Microsoft

Microsoft® .NET: Architecting Applications for the Enterprise

Dino Esposito
Andrea Saltarello

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2008935425

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 3 2 1 0 9 8

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Project Editor: Lynn Finnel
Editorial Production: S4Carlisle Publishing Services
Technical Reviewer: Kenn Scribner ; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X15-12272

To Silvia, Francesco, and Michela who wait for me and keep me busy.
But I'm happy only when I'm busy.

—Dino

To Mum and Depeche Mode.

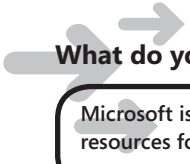
—Andrea

"Any sufficiently advanced technology is indistinguishable from magic."

—Arthur C. Clarke

Contents at a Glance

Table of Contents



What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Acknowledgments

For at least two years, Andrea didn't miss any opportunity to remind Dino about the importance of a .NET-focused architecture book covering the horizontal slice of a multitier enterprise system. And for two years Dino strung Andrea along with generic promises, but absolutely no commitment. Then, suddenly, he saw the light. During a routine chat over Messenger, we found out that we repeatedly made similar statements about architecture—too many to mark it down as a simple coincidence. So we started thinking, and this time seriously, about this book project. But we needed a team of people to do it right, and they were very good people, indeed.

Ben Ryan was sneakily convinced to support the project on a colorful Las Vegas night, during an ethnic dinner at which we watched waiters coming up from and going down to the wine-cellar in transparent elevators.

Lynn Finnel just didn't want to let Dino walk alone in this key project after brilliantly coordinating at least five book projects in the past.

Kenn Scribner is now Dino's official book alter ego. Kenn started working with Dino on books back in 1998 in the age of COM and the Active Template Library. How is it possible that a book with Dino's name on the cover isn't reviewed and inspired (and fixed) by Kenn's unique and broad perspective on the world of software? The extent to which Kenn can be helpful is just beyond human imagination.

Roger LeBlanc joined the team to make sure that all these geeks sitting together at the same virtual desktop could still communicate using true English syntax and semantics.

We owe you all the (non-rhetorically) monumental "Thank you" for being so kind, patient, and accurate.

Only two authors and a small team for such a great book? Well, not exactly. Along the project lifetime, we had the pleasure to welcome aboard a good ensemble of people who helped out in some way. And we want to spend a word or two about each of them here.

Raffaele Rialdi suggested and reviewed our section in Chapter 3 about design for security. **Roy Osherove** was nice enough to share his enormous experience with testing and testing tools. **Marco Abis** of ThoughtWorks had only nice words for the project and encouraged us to make it happen. **Alex Homer** of Microsoft helped with Unity and Enterprise Library. And the whole team at Managed Design (our Italian company) contributed tips and ideas—special thanks go to **Roberto Messora**.

It's really been a pleasure!

—Andrea and Dino

Dino's Credits

This is the first book I have co-authored in 8 or 9 years. I think the last was a multi-author book on data access involving COM and OLE DB. In the past, co-authoring a book for me meant accepting to write a few chapters on specific topics, while having only a faint idea of what was coming before and after my chapters.

This book is different.

This book has really been written by a virtual author: a human with the hands of Dino and the experience of Andrea. I actually did most of the writing, but Andrea literally put concepts and ideas into my keyboard. If it were a song, it would be described as lyrics by Dino and music by Andrea.

This book wouldn't exist, or it wouldn't be nearly as valuable, without Andrea. Andrea has been my personal Google for a few months—the engine to search when I need to understand certain principles and design issues. The nicest part of the story is that I almost always asked about things I (thought I) knew enough about. My "enough" was probably really enough to be a very good architect in real life. But Andrea gave me a new and broader perspective on virtually everything we covered in the book—ISO standards, UML, design principles, patterns, the user interface, business logic, services, and persistence. I've been the first hungry reader of this book. And I've been the first to learn a lot.

It was so fun that I spent the whole summer on it. And in Italy, the summer is a serious matter. I smile when I get some proposals for consulting or training in mid-August. There's no way I can even vaguely hint to my wife about accepting them.

So, on many days, I reached 7 p.m. so cloudy minded that running, running, and running—which was more soothing than my favorite pastime of trying to catch up to and hit a bouncing tennis ball—was the only way to recover a decent state of mind. On other days, my friends at **Tennis Club Monterotondo** helped a lot by just throwing at me tons of forehands and passing shots. One of them, **Fabrizio**—a guy who played Boris Becker and Stefan Edberg and who now wastes his time with my hopeless backhand slice—has been my instructor for a while. He also tried to learn some basic concepts of Web programming during what often became long conversations while changing ends of the court. But just as I keep on twirling the wrist during the execution of a backhand slice, he still keeps on missing the whole point of HTTP cookies.

My friend **Antonio** deserves a very special mention for organizing a wonderful and regenerative vacation in the deep blue sea of Sardinia, and for being kind enough to lose all the matches we

played. It was just the right medicine to rejuvenate a fatigued spirit after a tough book project. He tried to initiate me into the sport of diving, too, but all I could do was snorkel while the kids got their Scuba Diver certification.

My kids, **Francesco** and **Michela**, grow taller with every book I write, and not because they just hop on the entire pile of dad's books. They're now 10 and 7, and Michela was just a newborn baby when I started working on my first .NET book for Microsoft Press. I really feel a strong emotion when attendees of conferences worldwide come by and ask about my kids—loyal readers of my books have been seeing their pictures for years now.

For me, this book is not like most of the others that I have written—and I do write about one book per year. This book marks a watershed, both personal and professional. I never expressed its importance in this way with **Silvia**, but she understood it anyway and supported me silently and effectively. And lovingly. And with great food, indeed!

Life is good.

—Dino

Andrea's Credits

This is my first book. More precisely, this is my first serious publication. The seeds for this book were sowed in November 2004 when a rockstar like Dino approached me and proposed that we work together.

We started a successful business partnership, and we delivered a number of classes and some articles—including one for MSDN Magazine—and took a number of industry projects home to ensure our customers were happy.

In all these years, Dino impressed me especially with his unique ability of going straight to the point, and being a terrifically quick learner of the fundamentals of any topics we touched on. More, he also showed an unparalleled ability to express any concept precisely and concisely. Countless times during this book project, I found my own wording hard to read, nebulous, and even cryptic. A few days later, instead, massaged by Dino, the same text looked to me magically fluent and perfectly understandable—just like any technical text should always be.

(OK, I admit. Sometimes I thought "I hate this man," but it was an unusual and unconfessed way to look up to Dino with admiration.)

More than everything else, what initially was a simple although successful professional collaboration turned into friendship. This book, therefore, is not a finish line. It is, instead, the starting point of a common path. I really don't know either where we're going or how long it will take, but I'm going to be happy to take the walk.

Being a full-time consultant, it was very hard for me to set aside the time needed for writing this book. So I had to start living a double life, resorting to writing in what you would define as "spare time": evenings and weekends, and suddenly the summer also became standard working time. Every now and then, it has been a little frustrating, but I found new strength and inspiration due to the love and support I was blessed with by my guardian angels: my **mom** and **Laura**. I'd like to say to them that words cannot express how precious your caring is. I love you.

Now, this is fun.

—Andrea

Introduction

Good judgment comes from experience, and experience comes from bad judgment.

—Fred Brooks

Every time we are engaged on a software project, we create a solution. We call the process *architecting*, and the resulting concrete artifact is the *architecture*. Architecture can be implicit or explicit.

An *implicit* architecture is the design of the solution we create mentally and persist on a bunch of Microsoft Office Word documents, when not on handwritten notes. An implicit architecture is the fruit of hands-on experience, the reuse of tricks learned while working on similar projects, and an inherent ability to form abstract concepts and factor them into the project at hand. If you're an expert artisan, you don't need complex drawings and measurements to build a fence or a bed for your dog; you can implicitly architect it in a few moments. You just proceed and easily make the correct decision at each crossroad. When you come to an end, it's fine. All's well that ends well.

An *explicit* architecture is necessary when the stakeholder concerns are too complex and sophisticated to be handled based only on experience and mental processes. In this case, you need vision, you need guidance, and you need to apply patterns and practices that, by design, take you where you need to be.

What Is Architecture?

The word *architecture* has widespread use in a variety of contexts. You can get a definition for it from the Oxford English Dictionary or, as far as software is concerned, from the American National Standards Institute/Institute of Electrical and Electronics Engineers (ANSI/IEEE) library of standards. In both cases, the definition of architecture revolves around planning, designing, and constructing something—be it a building or a software program. Software architecture is the concrete artifact that solves specific stakeholder concerns—read, *specific user requirements*.

An architecture doesn't exist outside of a context. To design a software system, you need to understand how the final system relates to, and is embedded into, the hosting environment. As a software architect, you can't ignore technologies and development techniques for the environment of choice—for this book, the .NET platform.

Again, what is architecture?

We like to summarize it as the art of making hard-to-change decisions correctly. The architecture is the skeleton of a system, the set of pillars that sustain the whole construction.

The architect is responsible for the architecture. The architect's job is multifaceted. She has to acknowledge requirements, design the system, ensure the implementation matches the expectation, and overall ensure that users get what they really need—which is not necessarily what they initially accept and pay for.

Software architecture has some preconditions—that is, design principles—and one post condition—an implemented system that produces expected results. Subsequently, this book is divided into two parts: principles and the design of the system.

The first part focuses on the role of the architect: what he does, who he interacts with and who he reports to. The architect is primarily responsible for acknowledging the requirements, designing the system, and communicating that design to the development team. The communication often is based on Unified Modeling Language (UML) sketches; less often, it's based on UML blueprints. The architect applies general software engineering principles first, and object-oriented design principles later, to break down the system into smaller and smaller pieces in an attempt to separate what is architecture (points that are hard to change) and what is not. One of the purposes of object-oriented design is to make your code easy to maintain and evolve—and easy to read and understand. The architect knows that maintainability, security, and testability need to be built into the system right from the beginning, and so he does that.

The second part of the book focuses on the layers that form a typical enterprise system—the presentation layer, business layer, and data access layer. The book discusses design patterns for the various layers—including Domain Model, Model-View-Presenter, and Service Layer—and arguments about the evolution of technologies and summaries of the new wave of tools that have become a common presence in software projects—O/R mappers and dependency injection containers.

So, in the end, what's this book about?

It's about the things you need to do and know to serve your customers in the best possible way as far as the .NET platform is concerned. Patterns, principles, and techniques described in the book are valid in general and are not specific to particularly complex line-of-business applications. A good software architecture helps in controlling the complexity of the project. And controlling the complexity and favoring maintainability are the sharpest tools we have to fight the canonical Murphy's Law of technology: "Nothing ever gets built on schedule or within budget."

The expert is the one who knows how to handle complexity, not the one who simply predicts the job will take the longest and cost the most—just to paraphrase yet another popular Murphy's Law.

Who This Book Is For

In the previous section, we repeatedly mentioned architects. So are software architects the ideal target audience for this book? Architects and lead developers in particular are the target audience, but any developers of any type of .NET applications likely will find this book beneficial. Everyone who wants to be an architect may find this book helpful and worth the cost.

What about prerequisites?

Strong object-oriented programming skills are a requirement, as well as having a good foundation of knowledge of the .NET platform and data access techniques. We point out a lot of design patterns, but we explain all of them in detail in nonacademic language with no weird formalisms. Finally, we put in a lot of effort into making this book read well. It's not a book about abstract design concepts; it is not a classic architecture book either, full of cross-references and fancy strings in square brackets that hyperlink to some old paper listed in the bibliography available at the end of the book.

This is (hopefully) a book you'll want to read from cover to cover, and maybe more than once—not a book to keep stored on a shelf for future reference. We don't expect readers to pick up this book at crunch time to find out how to use a given pattern. Instead, our ultimate goal is transferring some valuable knowledge that enables you to know what to do at any point. In a certain way, we would happy if, thanks to this book, you could do more *implicit* architecture design on your own.

Companion Content

In the book, we present several code snippets and discuss sample applications, but with the primary purpose of illustrating principles and techniques for readers to apply in their own projects. In a certain way, we tried to teach fishing, but we don't provide some sample fish to take home. However, there's a CodePlex project that we want to point out to you. You find it at *http://www.codeplex.com/nsk*.

This book also features a companion Web site where you can also find the CodePlex project. You can download it from the companion site at this address: *http://www.microsoft.com/mspress/companion/9780735626096*.

The Northwind Starter Kit (NSK) is a set of Microsoft Visual Studio 2008 projects that form a multitier .NET-based system. Produced by Managed Design (*http://www.manageddesign.it*), NSK is a reference application that illustrates most of the principles and patterns we discuss in the book. Many of the code snippets in the book come directly from some of the projects in the NSK solution. If you're engaged in the design and implementation of a .NET layered application, NSK can serve as a sort of blueprint for the architecture.

Refer to the Managed Design Web site for the latest builds and full source code. For an overview of the reference application, have a look at the Appendix, "The Northwind Starter Kit," in this book.

Hardware and Software Requirements

You'll need the following hardware and software to work with the companion content included with this book:

- Microsoft Windows Vista Home Premium Edition, Windows Vista Business Edition, or Windows Vista Ultimate Edition
- Microsoft Visual Studio 2008 Standard Edition, Visual Studio 2008 Enterprise Edition, or Microsoft Visual C# 2008 Express Edition and Microsoft Visual Web Developer 2008 Express Edition
- Microsoft SQL Server 2005 Express Edition, Service Pack 2
- The Northwind database of Microsoft SQL Server 2000 is used by the Northwind Starter Kit to demonstrate data-access techniques. You can obtain the Northwind database from the Microsoft Download Center (http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en).
- 1.6 GHz Pentium III+ processor, or faster
- 1 GB of available, physical RAM.
- Video (800 by 600 or higher resolution) monitor with at least 256 colors.
- CD-ROM or DVD-ROM drive.
- Microsoft mouse or compatible pointing device

Find Additional Content Online

As new or updated material becomes available that complements this book, it will be posted online on the Microsoft Press Online Developer Tools Web site. The type of material you might find includes updates to book content, articles, links to companion content, errata, sample chapters, and more. This Web site is available at *www.microsoft.com/learning/books/online/developer* and is updated periodically.

Support for This Book

Every effort has been made to ensure the accuracy of this book and the contents of the companion CD. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article.

Microsoft Press provides support for books and companion CDs at the following Web site:

http://www.microsoft.com/learning/support/books

Questions and Comments

If you have comments, questions, or ideas regarding the book or the companion content, or questions that are not answered by visiting the sites above, please send them to Microsoft Press via e-mail to

mspinput@microsoft.com

Or via postal mail to

Microsoft Press
Attn: *Microsoft .NET: Architecting Applications for the Enterprise* Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that Microsoft software product support is not offered through the above addresses.

Part I

Principles

You know you've achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away.

—Antoine de Saint-Exupery, "Wind, Sand and Stars"

In this part:

Chapter 1

Architects and Architecture Today

The purpose of software engineering is to control complexity, not to create it.

—Dr. Pamela Zave

At the beginning of the computing age, in the early 1960s, the costs of hardware were largely predominant over the costs of software. Some 40 years later, we find the situation to be radically different.

Hardware costs have fallen dramatically because of the progress made by the industry. Software development costs, on the other hand, have risen considerably, mostly because of the increasing complexity of custom enterprise software development. Cheaper computers made it worthwhile for companies to add more and more features to their information systems. What in the beginning was a collection of standalone applications with no connection to one another that barely shared a database has grown over years into a complex system made of interconnected functions and modules, each with a particular set of responsibilities.

This situation has created the need for a set of precepts to guide engineers in the design of such systems. The modern software system—or the software-intensive system, as it is referred to in international standards papers—can be compared quite naturally to any construction resulting from a set of detailed blueprints.

Appropriated from the construction industry, the term *architecture* has become the appropriate way to describe the art of planning, designing, and implementing software-intensive systems. In software, though, architecture needs less artistry than in building. Well-designed buildings are pleasing to the eye and functional. Software architecture is less subjective. It either functions as required or it does not. There is less room for artistry and interpretation, unless you want to consider the artistry of a well-crafted algorithm or a piece of user interface.

One of this book's authors had, in the past, frequent interaction with an architecture studio. One day, a question popped up for discussion: What's architecture? Is it an art? Or is it just building for a client?

In software, the term *architecture* precisely refers to building a system for a client.

In this first chapter, we'll look at some papers from the International Organization for Standardization (ISO), the International Electrotechnical Commission (IEC), and the Institute of Electrical and Electronics Engineers (IEEE) that provide an architectural description of software-intensive systems. From there, we'll give our own interpretation of software architecture and voice our opinions about the role and responsibilities of software architects.

Note While some definitions you find in this book come from ISO standards, others reflect our personal opinions, experiences, and feelings. Although the reader might not agree with all of our personal reflections, we all should agree that software systems that lack strong architectural design and support are nearly guaranteed to fail. So having good architects on the team is a necessity. What's a "good" architect? It is one who is experienced, educated, and qualified.

Modern systems need more engineering and understanding, and less artistry and subjective guesswork. This is the direction we need to move toward as good software architects.

What's a Software Architecture, Anyway?

Herman Melville, the unforgettable author of *Moby Dick*, once said that men think that by mouthing hard words they can understand hard things. In software, the "hard" word *architecture* was originally introduced into the field to simplify the transmission and understanding of a key and "hard" guideline. The guideline was this: Care (much) more about the design of software systems than you have in the past; care about it to the point of guiding the development of a software system similar to guiding the development of a building.

It's a hard thing to do and probably beyond many developers' capabilities. But let's give it a try. Let's try to clarify what a "software architecture" is or, at least, what we intend it to be.

Applying Architectural Principles to Software

The word "architecture" is indissolubly bound to the world of construction. It was first used in the software industry to express the need to plan and design before building computer programs. However, a fundamental difference exists between designing and building habitable structures and designing and building usable software systems.

Intuitively, we care if the building falls on people. But software? There is always plenty of money to rewrite things, right? In construction, the design must be completed entirely up front and based on extremely detailed calculations and blueprints. In software, you tend to be more agile. A few decades ago, the up-front design methodology was common and popular in software, too. But, over the years, that approach increased development costs. And because software can be efficiently (and safely) tested before deployment, agility got the upper hand over up-front design.

Today the architectural parallelism between construction and software is not as close as it was a few years ago. However, many dictionaries currently list a software-related definition of the term "architecture." And a software architecture is described as "the composition, integration, and interaction of components within a computer system." It is certainly a definition that everybody would agree on. But, in our opinion, it is rather abstract.

We think that software professionals should agree on a more detailed explanation that breaks down that definition into smaller pieces and puts them into context.

Defining the Architecture from a Standard Viewpoint

Many seem to forget that a standard definition for software architecture exists. More precisely, it is in ANSI/IEEE standard 1471, "Recommended Practice for Architectural Description of Software-intensive Systems." The document was originally developed by IEEE and approved as a recommended practice in September 2000.

The document focuses on practices to describe the architecture of software-intensive systems. Using the definition in the standard, a *software-intensive system* is any system in which software is essential to implementation and deployment.

Stakeholders are defined as all parties interested or concerned about the building of the system. The list includes the builders of the system (architects, developers, testers) as well as the acquirer, end users, analysts, auditors, and chief information officers (CIOs).

In 2007, the ANSI/IEEE document was also recognized as a standard through ISO/IEC document 42010. Those interested in reading the full standard can navigate their browser to the following URL: *http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45991*.

Examining Key Architecture-Related Points in ANSI/IEEE 1471

The key takeaway from the ANSI/IEEE standard for software architecture is that a software system exists to meet the expectations of its stakeholders. Expectations are expressed as functional and nonfunctional requirements. Processed by the architect, requirements are then communicated to the development team and finally implemented. All the steps occur and exist to ensure the quality of the software. Skipping any step introduces the possibility for less software quality and the potential to not meet the stakeholders' expectations.

To design a software system that achieves its goals, you need to devise it using an architectural metaphor. Accepting an architectural metaphor means that you recognize the principle that some important decisions regarding the system might be made quite early in the development process; just like key decisions are made very early in the development of civil architecture projects. For example, you wouldn't build a skyscraper when a bridge was required. Similarly, requirements might steer you to a Web-oriented architecture rather than a desktop application. Decisions this major must be made very early.

A software architecture, therefore, is concerned with the organization of a system and lays out the foundations of the system. The system, then, has to be designed—which entails making some hard decisions up front—and described—which entails providing multiple views of the system, with each view covering a given set of system responsibilities.

Defining the System from a Standard Viewpoint

As mentioned, a software system is universally understood to be a collection of components composed and integrated to accomplish a specific set of functions.

A system lives in a context; and this context influences the design of the system by driving some developmental and operational decisions. A system exists to solve a problem and achieve its mission in full respect of the stakeholders' concerns. Stakeholders' concerns include functional and nonfunctional requisites as well as system aspects such as security, testability, performance, reliability, and extensibility.

Although it envisions the system as a composition of interconnected components, an architecture also establishes some firm points that are hard to modify later. In a way, expressing software development in terms of an architecture boils down to making some key decisions that affect the development life cycle and, ultimately, the quality of the resulting system.

Figure 1-1 illustrates the relationships between the system, architecture, and stakeholders as identified by ANSI/IEEE standard 1471. The content of Figure 1-1 is actually an adaptation of one of the figures in the document.

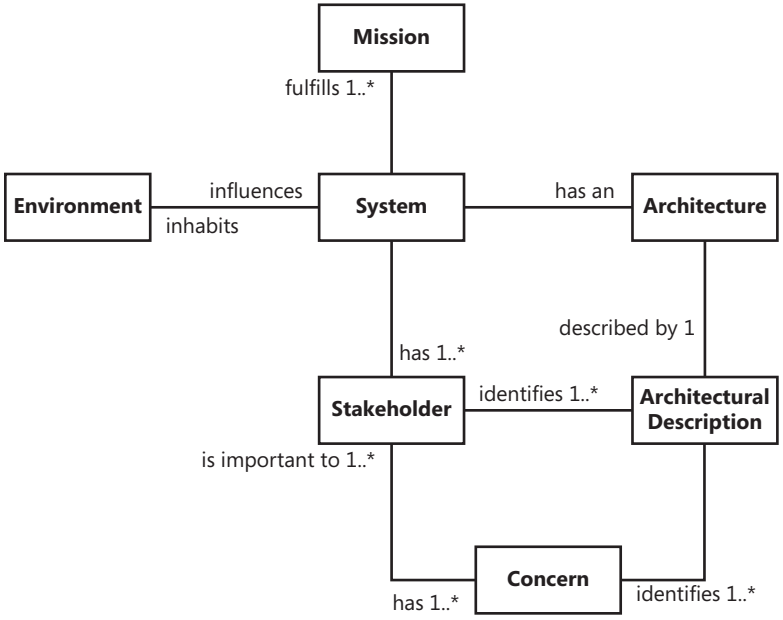


FIGURE 1-1 A model for creating an architectural description

The figure uses the notation of the Unified Modeling Language (UML) to express its concepts. We'll offer a refresher of UML in the next chapter. Even if you're new to UML, you should be able to read the figure quite comfortably. Each connector indicates an action

(with a specified name) that is executed by the element where the connector starts and that affects the target. So, for example, the System fulfills one or more Missions; the Environment (context) influences the System; a Concern is important to one or more Stakeholders; the System has an Architecture.

Describing the Architecture

As you can see in Figure 1-1, the Architecture is described by one Architectural Description. How would you describe the architecture to stakeholders?

The key messages about an architecture are its components and classes, their mapping onto binaries, their relationships and dependencies, usage scenarios, and detailed work flows for key operations. How would you render all these things?

A very popular choice is UML diagrams.

UML is also a recognized international standard—precisely, ISO/IEC 19501 released in 2005. You create UML *class diagrams* to show relationships between classes; you employ *use-case diagrams* to present usage scenarios; you create *component diagrams* to capture the relationships between reusable parts of a system (components) and see more easily how to map them onto binaries. In some cases, you can also add some *sequence diagrams* to illustrate in more detail the workflow for some key scenarios. These terms are defined and discussed in more detail in Chapter 2, "UML Essentials."

At the end of the day, you serve different and concurrent views of the same architecture and capture its key facts.

Note The same principle of offering multiple views from distinct viewpoints lies behind another vendor-specific model for architectural description—IBM/Rational's *4+1 views* model. The model defines four main views—nearly equivalent to UML diagrams. These views are as follows:

- The *logical view*, which describe components
- The *process view*, which describes mapping and dependencies
- The *development view*, which describes classes
- The *physical view*, which (if necessary) describes mapping onto hardware

The fifth, partially redundant, view is the scenario view, which is specific to use cases.

Validating the Architecture

How would you validate the design to ensure that stakeholders' concerns are properly addressed?

There's no magic wand and no magic formulas that take a more or less formal definition of an architecture as input and tells you whether it is appropriate for the expressed

requirements. To validate the design of a system, you can only test it—in various ways and at various levels.

So you will perform unit tests to validate single functionalities, and you will perform integration tests to see how the system coexists with other systems and applications. Finally, you'll run acceptance tests to verify how users actually feel about the application and whether the application provides the services it was created for. (Testing is one of the key topics of Chapter 3, "Design Principles and Patterns.")

What's Architecture and What's Not

When you think about creating or defining the architecture of a software system, you first try to identify a possible collection of interacting components that, all together, accomplish the requested mission. In international standards, there's no mention for any methodology you should use to decompose the system into more detailed pieces. Let's say that in the first step you a get a conceptual architecture and some different views of it. In a second step, you need to get closer to a functional and physical architecture. How you get there is a subjective choice, although a top-down approach seems to be a very reasonable strategy. You decompose components into smaller and smaller pieces, and from there you start building.

No System Is a Monolith

We've been told that, once upon a time, any piece of software was a monolith with an entry point and finish point. The introduction of structured programming, and the concept of a *subroutine*, in the early 1970s started shouldering such monoliths out of the way.

Since then, many software systems have been designed as a graph of components communicating in various ways and having various levels of dependency. In practical terms, designing a system consists of expanding the System element that was shown in Figure 1-1 into a graph of subsystems and defining communication policies and rules for each of them.

The process of breaking down these details should ideally continue until you have described in detail the structure and relationships for the smallest part of the system. Although fully completing the breakdown process up front is key for constructing habitable buildings, it is not that necessary for building software.

The actual implementation of the breakdown process depends on the methodology selected for the project—the more you are *agile*, the more the breakdown process is iterative and articulated in smaller and more frequent steps. (We'll return to the topic of methodologies later in the chapter.)

The output of the breakdown process is a set of specifications for the development team. Also, the content and format of the specifications depend on the methodology. The more you

are agile, the more freedom and independence you leave to developers when implementing the architecture.

Defining the Borderline Between Architecture and Implementation

The constituent components you identified while breaking down the system represent logical functions to be implemented in some way. The design of components, their interface, their responsibilities, and their behavior are definitely part of the architecture. There's a border, though, that physically separates architecture from implementation.

This border is important to identify because, to a large extent, it helps to define roles on a development team. In particular, it marks the boundary between architects and developers. Over the years, we learned that architects and developers are not different types of fruit, like apples and oranges. They are the same type of fruit. However, if they are apples, they are like red apples and green apples. Distinct flavors, but not a different type of fruit. And neither flavor is necessarily tastier.

You have arrived at the border between architecture and implementation when you reach a *black box of behavior*. A black box of behavior is just a piece of functionality that can be easily replaced or refactored without significant regression and with zero or low impact on the rest of the architecture. What's above a black box of behavior is likely to have architectural relevance and might require making a hard-to-change decision.

What's our definition of a good architecture? It is an architecture in which all hard-to-change decisions turn out to be right.

Dealing with Hard-to-Change Decisions

There are aspects and features of a software system that are hard (just *hard*, not *impossible*) to change once you have entered the course of development. And there are aspects and features that can be changed at any time without a huge effort and without having a wide impact on the system.

In his book *Patterns of Enterprise Application Architecture* (Addison-Wesley, 2002), Martin Fowler puts it quite simply:

> *If you find that something is easier to change than you once thought, then it's no longer architectural. In the end architecture boils down to the important stuff—whatever that is.*

To sum it up, we think that under the umbrella of the term *architecture* falls everything you must take seriously at quite an early stage of the project. Architecture is ultimately about determining the key decisions to make and then making them correctly.

Architecture Is About Decisions

When we talk about hard architectural decisions, we are not necessarily referring to irreversible decisions about design points that can be difficult and expensive to change later. Hard-to-change decisions are everywhere and range from the definition of a conceptual layers to the attributes of a class.

To illustrate our point, let's go through a few different examples of architectural points that can run into budget limits and deadlines if you have to touch them in the course of the project.

Changing the Organization of the Business Logic

In Chapter 4, "The Business Layer," we'll examine various approaches to organizing the business logic in the context of a layered system. Possible approaches for the design of the business logic include transaction script, table module, active record, and domain model. The selection of a pattern for the business logic is an excellent example of a design choice to be made very, very carefully. Once you have opted for, say, table module (which means, essentially, that you'll be using typed *DataSets* to store an application's data in the business logic layer), moving to an object model (for example, using the LINQ-to-SQL or Entity Framework object model) is definitely hard and requires nontrivial changes in the data access layer and in the application (service) layer, and probably also in the presentation layer. If you need to change this decision later in the project, you enter into a significant refactoring of the whole system.

Switching to a Different Library

Suppose you developed some functionality around a given library. One day, the client pops up and lets you know that a new company policy prevents the IT department from buying products from a given vendor. Now you have a new, unexpected nonfunctional requirement to deal with.

A change in the specifications might require a change in the architecture, but at what cost? In such a case, you have to comply with the new list of requirements, so there's not much you can do.

In the best case, you can get a similar tool from an authorized vendor or, perhaps, you can build a similar tool yourself. Alternatively, you can consider introducing a radical change into the architecture that makes that library unnecessary.

We faced an analogous situation recently, and the type of library was an Object/Relational mapping tool. With, say, a UI control library, it would have been much simpler to deal with. Replacing an Object/Relational mapping tool is not easy; it is a task that can be accomplished *only* by getting another tool from another vendor. Unfortunately, this wasn't possible. In other

words, we were left to choose between either of two unpleasant and painful options: writing our own Object/Relational mapping tool, or rearchitecting the middle tier to use a different (and much simpler) object model.

With over 500 presenters in the Model View Presenter–based user interface directly consuming the object model, having to make this decision was our worst nightmare. We knew it would require a huge amount of work on the middle tier, consuming both financial resources and time. We lobbied for more time and successfully stretched the deadline. Then we built our own tailor-made data access layer for a domain model. (After you've read Chapter 6, "The Data Access Layer," you'll have a clear picture of what this all means.)

Changing the Constructor's Signature

Don't think that architecture is only about high-level decisions like those involving the design and implementation of parts of the middle tier. A requested change in the signature of a class constructor might get you in a fine mess, too.

Imagine a scenario where you handle an *Order* class in your application's object model. You don't see any reason to justify the introduction of a factory for the *Order* class. It is a plain class and should be instantiated freely. So you scatter tons of *new Order()* instructions throughout your code. You don't see, though, that *Order* has some logical dependency on, say, *Customer*.

At some point, a request for change hits you—in the next release, an order will be created only in association with a customer. What can you do?

If you only add a new constructor to the *Order* class that accepts a *Customer* object, you simply don't meet the requirement, because the old constructor is still there and only new code will follow the new pattern. If you drop or replace the old constructor, you have tons of *new* statements to fix that are scattered throughout the entire code base.

If only you had defined a factory for the *Order* class, you would have met the new requirement without the same pain. (By the way, domain-driven design methodology in fact suggests that you always use a factory for complex objects, such as aggregates.)

Changing a Member's Modifiers

When you design a class, you have to decide whether the class is public or internal and whether it is sealed or further inheritable. And then you decide whether methods are virtual or nonvirtual. Misusing the *virtual* and *sealed* modifiers might take you along an ugly route.

In general, when you use the *sealed* and *virtual* modifiers you take on a not-so-small responsibility. In C#, by default each class is unsealed and each method on a class is nonvirtual. In Java, for example, things go differently for methods, which are all virtual by default.

Now what should you do with your .NET classes? Make them sealed, or go with the default option?

The answer is multifaceted—maintenance, extensibility, performance, and testability all might factor into your decision. We're mostly interested in maintenance and extensibility here, but we'll return to this point in Chapter 3 when we touch on design for testability and make some performance considerations.

From a design perspective, sealed classes are preferable. In fact, when a class is sealed from the beginning you know it—and you create your code accordingly. If something happens later to justify inheritance of that class, you can change it to unsealed without breaking changes and without compromising compatibility. Nearly the same can be said for virtual methods, and the visibility of classes and class members, which are always private by default.

The opposite doesn't work as smoothly. You often can't seal a class or mark a virtual method as nonvirtual without potentially breaking some existing code. If you start with most-restrictive modifiers, you can always increase the visibility and other attributes later. But you can never tighten restrictions without facing the possibility of breaking existing dependencies. And these broken dependencies might be scattered everywhere in your code.

To contrast these statements, some considerations arise on the theme of testability. A nonsealed class and virtual methods make testing much easier. But the degree of ease mostly depends on the tool you use for testing. For example, TypeMock is a tool that doesn't suffer from these particular limitations.

It's hard to make a choice as far as the *sealed* and *virtual* keywords are concerned. And whatever choice you make in your context, it doesn't have to be a definitive choice that you blindly repeat throughout your code for each class and member. Make sure you know the testability and performance implications, make sure you know the goals and scope of your class, and then make a decision. And, to the extent that it's possible, make the right decision!

Requirements and Quality of Software

The mission of the system is expressed through a set of requirements. These requirements ultimately drive the system's architecture.

In rather abstract terms, a *requirement* is a characteristic of the system that can either be functional or nonfunctional. A *functional* requirement refers to a behavior that the system must supply to fulfill a given scenario. A *nonfunctional* requirement refers to an attribute of the system explicitly requested by stakeholders.

Are the definitions of functional and nonfunctional requirements something standard and broadly accepted? Actually, an international standard to formalize quality characteristics of software systems has existed since 1991.

Examining the ISO/IEC 9126 Standard

As a matter of fact, failure to acknowledge and adopt quality requirements is one of the most common causes that lead straight to the failure of software projects. ISO/IEC 9126 defines a general set of quality characteristics required in software products.

The standard identifies six different families of quality characteristics articulated in 21 subcharacteristics. The main families are functionality, reliability, usability, efficiency, maintainability, and portability. Table 1-1 explains them in more detail and lists the main subcharacteristics associated with each.

TABLE 1-1 Families of Quality Characteristics According to ISO/IEC 9126

Family	Description
Functionality	Indicates what the software does to meet expectations. It is based on requirements such as suitability, accuracy, security, interoperability, and compliance with standards and regulations.
Reliability	Indicates the capability of the software to maintain a given level of performance when used under special conditions. It is based on requirements such as maturity, fault tolerance, and recoverability. Maturity is when the software doesn't experience interruptions in the case of internal software failures. Fault tolerance indicates the ability to control the failure and maintain a given level of behavior. Recoverability indicates the ability to recover after a failure.
Usability	Indicates the software's ability to be understood by, used by, and attractive to users. It dictates that the software be compliant with standards and regulations for usability.
Efficiency	Indicates the ability to provide a given level of performance both in terms of appropriate and timely response and resource utilization.
Maintainability	Indicates the software's ability to support modifications such as corrections, improvements, or adaptations. It is based on requirements such as testability, stability, ability to be analyzed, and ability to be changed.
Portability	Indicates the software's ability to be ported from one platform to another and its capability to coexist with other software in a common environment and sharing common resources.

Subcharacteristics are of two types: external and internal. An external characteristic is user oriented and refers to an external view of the system. An internal characteristic is system oriented and refers to an internal view of the system. External characteristics identify functional requirements; internal characteristics identify nonfunctional requirements.

As you can see, features such as security and testability are listed as requirements in the ISO standard. This means that an official paper *states* that testability and security are an inherent part of the system and a measure of its quality. More importantly, testability and security should be planned for up front and appropriate supporting functions developed.

Important If you look at the ISO/IEC 9126 standard, you should definitely bury the practice of first building the system and then handing it to a team of network and security experts to make it run faster and more securely. You can't test quality in either. Like security, quality has to be designed in. You can't hope to test for and find all bugs, but you can plan for known failure conditions and use clean coding practices to prevent (or at least minimize) bugs in the field.

It's surprising that such a practice has been recommended, well, since 1991. To give you an idea of how old this standard is, consider that at the time it was written both Windows 3.0 and Linux had just been introduced, and MS-DOS 5.0 was the rage, running on blisteringly fast Intel i486 processors. It was another age.

In the context of a particular system, the whole set of general quality requirements set by the ISO/IEC 9126 standard can be pragmatically split into two types of requirements: functional and nonfunctional.

Functional Requirements

Functional requirements indicate *what* the system is expected to do and provide an appropriate set of functions for such specified tasks and user objectives. Generally, a function consists of input, behavior, and output. A team of analysts is responsible for collecting functional requirements and communicating them to the architect. Another common source of functional requirements are meetings organized with users, domain experts, and other relevant stakeholders. This process is referred to as *elicitation*.

Requirements play a key role in the generation of the architecture because they are the raw input for architects to produce specifications for the development team. Needless to say, it is recommended by ISO/IEC that software requirements be "clear, correct, unambiguous, specific, and verifiable."

However, this is only how things go in a perfect world.

Nonfunctional Requirements

Nonfunctional requirements specify overall requirements of the final system that do not pertain specifically to functions. Canonical examples of nonfunctional requirements are using (or not using) a particular version of a framework and having the final product be interoperable with a given legacy system.

Other common nonfunctional requirements regard support for accessibility (especially in Web applications developed for the public sector) or perhaps the provision of a given level of security, extensibility, or reliability.

In general, a nonfunctional requirement indicates a constraint on the system and affects the quality of the system. Nonfunctional requirements are set by some of the system stakeholders and represent a part of the contract.

Gathering Requirements

The analyst is usually a person who is very familiar with the problem's domain. He gathers requirements and writes them down to guarantee the quality and suitability of the system. The analyst usually composes requirements in a document—even a Microsoft Office Word document—in a format that varies with the environment, project, and people involved.

Typically, the analyst writes requirements using casual language and adds any wording that is specific to the domain. For example, it is acceptable to have in a requirement words such as *Fund*, *Stock*, *Bond*, and *Insurance Policy* because they are technical terms. It is less acceptable for a requirement to use terms such as *table* or *column* because these technical terms are likely to be foreign terms in the problem's domain.

Again, requirements need to be clear and verifiable. Most importantly, they must be understandable, without ambiguity, to all stakeholders—users, buyers, analysts, architects, testers, documentation developers, and the like.

Note It is not uncommon that analysts write functional requirements using relatively abstract use cases. As we'll see in a moment, a use case is a document that describes a form of interaction between the system and its clients. Use cases created by the analysis team are not usually really detailed and focus on *what* the system does rather than *how* the system does it. In any case, it must come out in a form that stakeholders can understand. In this regard, a use case describes all the possible ways for an actor to obtain a value, or achieve a goal, and all possible exceptions that might result from that.

Specifications

Based on functional and nonfunctional requirements, specifications offer a development view of the architecture and are essentially any documentation the architect uses to communicate details about the architecture to the development team. The main purpose of specifications is to reach an understanding within the development team as to how the program is going to perform its tasks.

Note Typically, an architect won't start working on specifications until some requirements are known. In the real world, it is unlikely that requirements will be entirely known before specifications are made. The actual mass of requirements that triggers the generation of specifications depends mostly on the methodology selected for the process. In an agile context, you start working on specifications quite soon, even with a largely incomplete set of requirements.

Specifications for functional requirements are commonly expressed through *user stories* or *use cases*.

A user story is an informal and very short document that describes, in a few sentences, what should be done. Each story represents a single feature and ideally describes a feature that stands on its own. User stories work especially well in the context of an agile methodology, such as Extreme Programming (XP), and are not designed to be exhaustive. A typical user story might be as simple as, "*The user places an order; the system verifies the order and accepts it if all is fine*." When, and if, that user story gets implemented, developers translate it into tasks. Next, through teamwork, they clarify obscure points and figure out missing details.

A use case is a document that describes a possible scenario in which the system is being used by a user. Instead of *user*, here, we should say *actor*, actually. An *actor* is a system's user and interacts with the system. An actor can be a human as well as a computer or another piece of software. When not human, an actor is not a component of the system; it is an external component. When human, actors are a subset of the stakeholders.

When used to express a functional requirement, a use case fully describes the interaction between actors and the system. It shows an actor that calls a system function and then illustrates the system's reaction. The collection of all use cases defines all possible ways of using the system. In this context, a use case is often saved as a UML diagram. (See Chapter 2 for detailed UML coverage.) The scenario mentioned a bit earlier in this section, described through a use case, might sound like this: "*The user creates an order and specifies a date, a shipment date, customer information, and order items. The system validates the information, generates the order ID, and saves the order to the database*." As you can see, it is a much more detailed description.

The level of detail of a specification depends on a number of factors, including company standards currently in use and, particularly, the methodology selected to manage the project. Simplifying, we can say that you typically use user stories within the context of an agile methodology; you use the use cases otherwise.

Note Note that use cases you might optionally receive from analysts are not the same as use cases that you, as an architect, create to communicate with the development team. More often than not, use cases received by analysts are plain Microsoft Office Word documents. Those that get handed on to the development team are typically (but not necessarily) UML diagrams. And, more importantly, they are much more detailed and oriented to implementation.

Methodology and the Use of Requirements

Collected and communicated by analysts, requirements are then passed down the chain to the design team to be transformed into something that could lead to working code. The architect is the member on the design team who typically receives requirements and massages them into a form that developers find easy to manage.

The architect is the point of contact between developers and stakeholders, and she works side by side with the project manager. It is not unusual that the two roles coincide and the same person serves simultaneously as an architect and a project manager.

The project manager is responsible for choosing a methodology for developing the project. To simplify, we could say that the project manager decides whether or not an agile methodology is appropriate for the project.

The choice of methodology has a deep impact on how requirements are used in defining the architecture.

In the case of using an agile methodology, user stories are the typical output generated from requirements. For example, consider that a typical XP iteration lasts about two weeks. (An XP iteration is a smaller and faster version of a classic software development cycle.) In two weeks, you can hardly manage complex specifications; you would spend all the time on the specifications, thus making no progress toward the implementation of those specifications. In this context, user stories are just fine.

In the case of using a traditional, non-agile methodology with much longer iterations, the architect usually processes a large share of the requirements (if not all of them) and produces exhaustive specifications, including classes, sequences, and work flows.

Who's the Architect, Anyway?

As we've seen, architecture is mostly about expensive and hard-to-change decisions. And someone has to make these decisions.

The design of the architecture is based on an analysis of the requirements. Analysis determines *what* the system is expected to do; architecture determines *how* to do that. And someone has to examine the *what*s to determine the *how*s.

The architect is the professional tying together requirements and specifications. But what are the responsibilities of an architect? And skills?

An Architect's Responsibilities

According to the ISO/IEC 42010 standard, an architect is the person, team, or organization responsible for the system's architecture. The architect interacts with analysts and the project manager, evaluates and suggests options for the system, and coordinates a team of developers.

The architect participates in all phases of the development process, including the analysis of requirements and the architecture's design, implementation, testing, integration, and deployment.

Let's expand on the primary responsibilities of an architect: acknowledging the requirements, breaking the system down into smaller subsystems, identifying and evaluating technologies, and formulating specifications.

Acknowledging the Requirements

In a software project, a few things happen before the architect gets involved. Swarms of analysts, IT managers, and executives meet, discuss, evaluate, and negotiate. Once the need for a new or updated system is assessed and the budget is found, analysts start eliciting requirements typically based on their own knowledge of the business, company processes, context, and feedback from end users.

When the list of requirements is ready, the project manager meets with the architect and delivers the bundle, saying more or less, "This is what we (*think we*) want; now you build it."

The architect acknowledges the requirements and makes an effort to have them adopted and fulfilled in the design.

Breaking Down the System

Based on the requirements, the architect expresses the overall system as a composition of smaller subsystems and components operating within processes. In doing so, the architect envisions logical layers and/or services. Then, based on the context, the architect decides about the interface of layers, their relationships to other layers, and the level of service orientation the system requires.

Note At this stage, the architect evaluates various architectural patterns. Layering is a common choice and the one we are mostly pursuing in this book. Layering entails a vertical distribution of functionality. Partitioning is another approach, where all parts are at the same logical level and scattered around some shared entities—such as an object model or a database. Service-oriented architecture (SOA) and hexagonal architecture (HA) are patterns that tend to have components (services in SOA, adapters in HA) operating and interacting at the same logical level.

The overall design will be consistent with the enterprise goals and requirements. In particular, the overall design will be driven by requirements; it will not lead requirements.

The resulting architecture is ideally inspired by general guidelines, such as minimizing the coupling between modules, providing the highest possible level of cohesion within modules, and giving each module a clear set of responsibilities.

The resulting architecture is also driven by nonfunctional requirements, such as security, scalability, and technologies allowed or denied. All these aspects pose further constraints and, to some extent, delimit the space where the architect can look for solutions.

Finally, the architect also strategizes about tasking individual developers, or teams of developers, with each of the components resulting from the breakdown of the system.

Note There are no absolute truths in software architecture. And no mathematical rules (or building codes like in structural engineering) to help in making choices. Company X might find architecture A successful at the same time company Y is moving away from it to embrace architecture B. The nice fact is that both might be totally right. The context is king, and so is *gut* feeling.

Identifying and Evaluating Technologies

After acknowledging requirements and designing the layers of the system, the next step for the architect entails mapping logical components onto concrete technologies and products.

The architect typically knows the costs and benefits of products and technologies that might be related to the content of the project. The architect proposes the use of any technologies and products that he regards as beneficial and cost-effective for the project.

The architect doesn't choose the technology; based on his skills, the architect just makes proposals.

The architect might suggest using, say, Microsoft Windows 2008 Server for the Web server and a service-oriented architecture with services implemented through Windows Communication Foundation (WCF). The architect might suggest NHibernate over Entity Framework and Microsoft SQL Server 2008 over Oracle. And he might suggest a particular rich control suite for the Web presentation layer instead of, perhaps, an entirely in-house developed Silverlight client.

Who does make the final decision about which technologies and products are to be used?

Typically, it is the project manager or whoever manages the budget. The architect's suggestions might be accepted or rejected. If a suggestion is rejected, using or not using a given product or technology just becomes a new nonfunctional requirement to fulfill, and that might influence, even significantly, the architecture.

Formulating Specifications

The architect is ultimately responsible for the development of the system and coordinates the work of a team of developers. Technical specifications are the means by which the architect communicates architectural decisions to the developers.

Specifications can be rendered in various forms: UML sketches, Word documents, Microsoft Visio diagrams or, even, working prototypes.

Communication is key for an architect. Communication happens between the architect and developers, and it also happens between architects and project managers and analysts, if not users. A great skill for an architect is the clarity of language.

The interaction between architects and developers will vary depending on the methodology chosen. And also the involvement of project managers, analysts, and users varies based, essentially, on the level of agility you accept.

We'll return to the topic of methodologies in a moment.

How Many Types of Architects Do You Know?

There are many possible definitions of "architect." Definitions vary depending on how they factor in different roles and different responsibilities. In this book, we work with the ISO/IEC definition of an architect, which is the "person, team, or organization responsible for the system's architecture."

According to ISO/IEC, there are not various types of architects. An architect is an architect. Period.

Microsoft, however, recognizes four types of architects: enterprise architect (EA), infrastructure architect (IA), technology-specific architect (TSA), and solution architect (SA). The list is taken from the job roles recognized by the Microsoft Certified Architect Program. You can read more about the program and job roles at *http://www.microsoft.com/learning/mcp/architect/specialties/default.mspx*.

In our opinion, the distinctions offered by Microsoft are misleading because they attempt to break into parts what is ultimately an atomic, yet complex, role. It creates unnecessary categorization and lays the groundwork for confusing, who-does-what scenarios.

For example, who's responsible for security? Is it the SA or the IA? Ultimately, security is an ISO-recognized quality attribute of a software architecture and, as such, it should be planned from the beginning. Security should grow with the system's design and implementation. It cannot be added at a later time, by a separate team. Not if you really want security in the system.

Who's ultimately responsible for picking out a technology? Is it the SA? Is it the EA? Do both accept suggestions from a swarm of different TSAs? At the end of the day, it's not the architect who makes this decision. Instead, it's the customer, who holds the purse strings, that decides.

It is fine to have multiple architects on the same project team. Likewise, it is fine, if not desirable, that different architects have slightly different skills. But they remain just architects, working on the same team on the design of the same system. And architects also have a significant exposure to code. They work out the design of the system but then work closely with developers to ensure proper implementation.

As we see things, an architect is, among other things, a better and more experienced developer. We don't believe there's value in having architects who just speak in UML and Visio and leave any implementation details to developers. At least, we've never found it easy to work with these people when we've crossed paths with them.

Note This said, we recognize that titles like *enterprise architect*, *solution architect*, and perhaps *security architect* look much better than a plain *software architect* when printed out on a business card. But the terms are only a way to more quickly communicate your skills and expertise. When it comes to the actual role, either you're an architect or you're not.

Common Misconceptions About Architects

Although international ISO standards exist to define requirements, architecture, and architects, they seem not to be taken into great account by most people. Everybody seems to prefer crafting her own (subtly similar) definition for *something*, rather than sticking to (or reading) the ISO definition for the same *something*.

Try asking around for the definition of terms such as *architect*, *architecture*, or *project manager*. You can likely get distinct, and also unrelated and contrasting, answers.

Quite obviously, a set of misconceptions have grown out of the mass of personalized definitions and interpretations. Let's go through a few of them and, we hope, clear up a few of them.

The Architect Is an Analyst

This is a false statement. An architect is simply not an analyst.

At times, an architect might assist analysts during elicitations to help clarify obscure requirements or smooth improbable requirements. At times, an architect might participate in meetings with stakeholders. But that's it.

In general, an analyst is a person who is an expert on the domain. An architect is not (necessarily) such an expert. An analyst shares with an architect his own findings about how the system should work and what the system should do.

This common misconception probably originates from the incorrect meaning attributed to the word *analyst*. If the word simply indicates someone who does some analysis on a system, it is quite hard to deny the similarities between architects and analysts. Some 30 years ago, the term *system analyst* was used to indicate a professional capable of making design considerations about a system. But, at the time, the software wasn't as relevant as it is today, and it was merely a (small) part of an essentially hardware-based system.

Today, the roles of an analyst and an architect are commonly recognized as being different. And hardly ever does an architect play the role of an analyst.

Note Given that roles are neatly separated, anyway, in small companies, it can happen that the same person serves as an analyst and architect. It simply means that there's a person in the company who knows the business and processes well enough to come up with functional requirements and translate them into specifications for developers. The roles and responsibilities are still distinct, but the distinct skills for each can be found in the same individual.

The Architect Is a Project Manager

Is this another false statement? It depends.

The architect is responsible for the system's architecture and coordinates and guides the development of the system. The project manager represents stakeholders and manages the project by choosing, in the first place, a methodology. The project manager is then responsible for ensuring that the project adheres to the architecture while proceeding within the limits of the timeline and budget.

If we look at the role of the architect and the role of the project manager, we find out that they are distinct. Period.

However, it is not unusual that one actor ends up playing two roles. Like in the theater, this hardly happens in large companies, but it happens quite frequently in small companies.

In summary, if you want to be a software architect when you grow up, you don't necessarily have to develop project management skills. If you have skills for both roles, though, you can try to get double pay.

The Architect Never Writes Any Code

This is definitely an ongoing debate: Should architects write code? There are essentially two schools of thought.

One school thinks that architects live on the upper floor, maybe in an attic. Architects then step down to the developers' floor just for the time it takes them to illustrate, using UML diagrams, what they have thought about the system. After this, they take the elevator up, collect their things, and go out to play golf. When on the course, they switch off their cell phones and focus on the game. When done, if they missed a call or two, they call back and explain to dummy developers what was so clear in the diagram that nobody on the developers' floor could understand. According to this school of thought, architects never, ever dirty their hands with even the simplest C# statement. C#? Oh no, the latest language they've been exposed to is probably Pascal while in college and Borland Turbo Pascal at home.

Another school of thought thinks, instead, that every architect is a born developer. To take the metaphor one step further, we could say that the class *Architect* inherits from the class *Developer* and adds some new methods (skills) while overriding (specializing) a few others. Becoming an architect is the natural evolution in the career of some developers. The basic differences between an architect and a developer are experience and education. You gain experience by spending time on the job; you earn your education from studying good books and taking the right classes. In addition, an architect has the ability to focus her vision of the system from a higher level than an average developer. Furthermore, an architect has good customer-handling skills.

An architect might not write much production code. But she writes a lot of code; she practices with code every day; she knows about programming languages, coding techniques, libraries, products, tools, Community Technology Previews (CTPs); and she uses the latest version of Visual Studio or Team Foundation Server. In certain areas of programming, an architect knows even more than many developers. An architect might be able to write tools and utilities to help developers be more productive. And, more often than you might think at first, the architect is just a member of the development team. For example, an architect writing production code is an absolutely normal occurrence in an agile context. It is also a normal occurrence in small companies regardless of the methodology. At the same time, an architect who writes production code might be an absolutely weird occurrence in some large-company scenarios, especially if a traditional and non-agile methodology is used.

What about the two of us? To which school do we belong?

Well, Andrea is more of an architect than Dino because he lives on the fifth floor. Dino, on the other hand, is closer to development because he has quite a few highly technical ASP .NET books on his record and, more importantly, lives on the second floor. We don't play golf, though. Dino plays tennis regularly, whereas Andrea likes squash better. We just have been denied access to the first school of thought.

Note In no other area of engineering is the distinction between *those-who-design* and *those-who-build* as poorly accepted as it is in software. The distinction exists mostly through postulation rather than flowing from a public recognition of skills.

The canonical comparison is with civil architecture. Bricklayers have their own unique skills that engineers lack. No bricklayer, though, will ever dream of questioning designs or calculations, simply because the bricklayer lacks the skill to make the decisions himself. Bricklayers do their own work the best they can, taking full advantage of having the building work delegated to them.

In software, the situation is different because architects and developers have common roots. The more skilled a developer is, the more he feels encouraged to discuss design choices—and often with reason. The more the architect loses contact with everyday programming, the more he loses the respect of other developers. This generates a sort of vicious circle, which magically becomes better as you switch to an agile methodology.

Overview of the Software Development Process

For quite a few years, we've been highly exposed to the idea that writing software is easy, pleasant, and fun. You click this, you drag that, and the tool will write the code for you. You "declare" what you want, and the award-winning tool will do it for you. Admittedly, in this scenario everybody could gain the rank of architect, and the burden of writing code can be entirely delegated to the tool—aptly named the *wizard*.

Not all software is the same.

Writing a filing system to track the movies you've rented from Blockbuster is different from writing a line-of-business application to run a company. You probably don't need to work on an architecture to build a syndication reader; you probably need more than just architecture if you're working on the control system for, say, a chemical plant.

In some sectors of the industry (for example, in the defense sector), the need for a systematic approach to the various aspects of software—development, testing, operation, maintenance—was recognized long ago, as early as the 1960s. In fact, the term *software engineering* was first coined by Professor Friedrich L. Bauer during the NATO Software Engineering Conference in 1968.

Today, software engineering is a broad term that encompasses numerous aspects of software development and organizes them into a structured process ruled by a methodology.

The Software Life Cycle

Software development is a process created and formalized to handle complexity and with the primary goal of ensuring (expected) results. As in the quote at the top of the chapter, the ultimate goal is controlling complexity, not creating it.

To achieve this goal, a methodology is required that spans the various phases of a software project. Over the years, an international standard has been developed to formalize the software life cycle in terms of processes, activities, and tasks that go from initial planning up to the retirement of the product.

This standard is ISO/IEC 12207, and it was originally released in 1995. The most recent revision, however, was in March 2008.

Processes

According to the ISO/IEC 12207 standard, the software life cycle is articulated in 23 processes. Each process has a set of activities and outcomes associated with it. Finally, each activity has a number of tasks to complete.

Processes are classified in three types: primary, supporting, and organizational. The production of the code pertains to the primary process. Supporting processes and organizational processes refer to auxiliary processes, such as configuration, auditing, quality assurance, verification, documentation, management, and setup and maintenance of the infrastructure (hardware, software, tools). Figure 1-2 offers a graphical view of the software life cycle, showing specific processes.

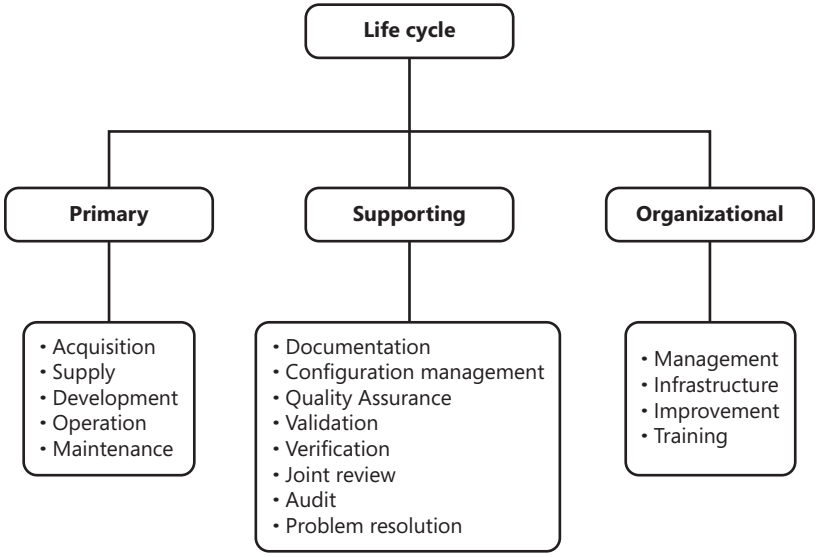


FIGURE 1-2 The overall software life cycle according to ISO/IEC 12207

Activities

The primary processes are those more directly concerned with the design and implementation of software. Let's briefly have a look at some of the activities for the primary processes.

The Acquisition process includes elicitation of requirements and evaluation of options, negotiations, and contracts. The Supply process is concerned with the development of a project management plan. Usually, architects are not involved in these steps unless they are also serving to some extent as project managers.

The Development process deals with the analysis of requirements, design of the system as well as its implementation, testing, and deployment. This is really all within the realm of the architect.

The Operation process essentially involves making the software operative within the company, integrating it with the existing systems, running pilot tests, and assisting users as they familiarize themselves with the software. Finally, the Maintenance process aims to keep

the system in shape by fixing bugs and improving features. This includes a set of activities that might require the architect to be involved to some extent.

Models for Software Development

Before starting on a software project, a methodology should be selected that is appropriate for the project and compatible with the skills and attitude of the people involved. A *methodology* is a set of recommended practices that are applied to the process of software development. The methodology inspires the realization and management of the project.

There are two main developmental models: traditional methodologies and agile methodologies. We'll also touch on a third model—the Microsoft Solutions Framework.

Traditional Methodologies

The best-known and oldest methodology is the *waterfall* model. It is a model in which software development proceeds from one phase to the next in a purely sequential manner. Essentially, you move to step N+1 only when step N is 100% complete and all is perfect with it. Figure 1-3 shows a sample of the waterfall model.

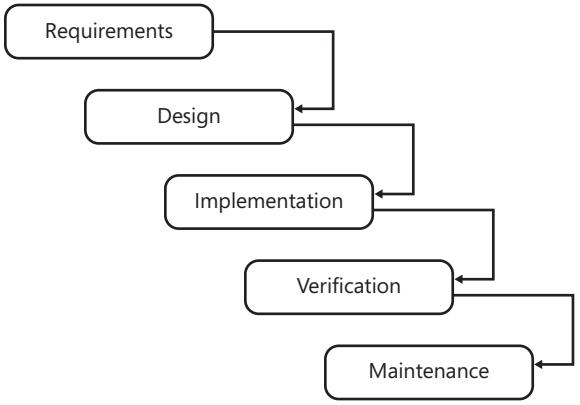


FIGURE 1-3 The waterfall model

After the team has completed the analysis of requirements, it proceeds with the design of the architecture. Next, coding begins. Next, testing is started, and it continues until the system is shipped.

The waterfall model goes hand in hand with the idea of software development paired to civil architecture. The primary characteristic of a waterfall model is BDUF—Big Design Up Front—which means essentially that the design must be set in stone before you start coding.

Waterfall is a simple and well-disciplined model, but it is unrealistic for nontrivial projects. Why? Because you almost never have all requirements established up front.

So you inevitably must proceed to the next step at some point while leaving something behind you that is incomplete.

For this reason, variations of the waterfall method have been considered over the years, where the design and implementation phases overlap to some extent. This leads us to a key consideration.

Ultimately, we find that all methodologies share a few common attributes: a number of phases to go through, a number of iterations to produce the software, and a typical duration for a single iteration. All phases execute sequentially, and there's always at least one iteration that ends with the delivery of the software.

The difference between methodologies is all in the order in which phases are entered, the number of iterations required, and the duration of each iteration.

After buying into this consideration, the step to adopting the agile methods is much smaller than you might think at first.

Note We could even say that when you move to an agile methodology, you have a much smaller *waterfall*, one that is less abundant and doesn't last as long. But it's more frequent and occurs nearly on demand. Not a waterfall. . .maybe a *shower*?

Agile Methodologies

Iterative development is a cyclic process that was developed in response to the waterfall method, and it emphasizes the incremental building of the software. After the initial startup, the project goes through a series of iterations that include design, coding, and testing. Each iteration produces a deliverable but incomplete version of the system. At each iteration, the team enters design changes and adds new functions until the full set of specifications are met. Figure 1-4 provides a graphical view of the iterative process.

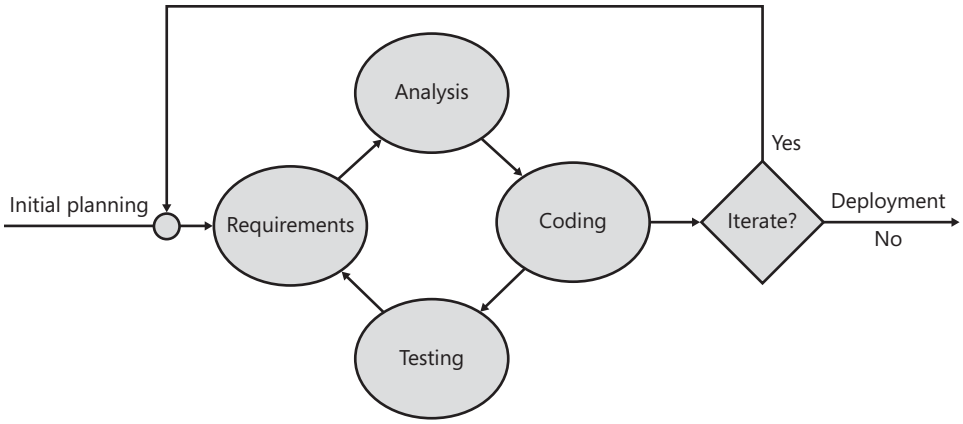


FIGURE 1-4 The iterative model

Iterative development forms the foundation of agile methodologies. The term *agile* was deliberately selected to symbolize a clear opposition to heavyweight methods such as the waterfall model. The principles behind agile methods are listed in the "Agile Manifesto," which you can find at *http://agilemanifesto.org*. The agile manifesto was first published in 2001.

Agile methodologies put individuals at the center of the universe. As stated on the home page of the manifesto, agile methods *focus on people* working together and communicating rather than on software building and processes. Change and refactoring are key in an agile methodology. User feedback is valued over planning, and feedback is driven by regular tests and frequent releases of the software. In fact, one of the agile principles states, "Working software is the primary measure of progress."

So in the end, how is an agile methodology different? And how does it work? Let's look at an example.

The project starts and only a few requirements are known. You know for a fact that many more will show up between now and the end. With an agile mindset, this is not an issue. You take a subset of the existing requirements that you can implement in a single iteration. And you go with the first iteration. During the iteration, you focus on a single requirement at a time and implement it. At the end of the iteration, you deliver a working piece of software. It might be incomplete, but it works.

Next, you go with another iteration that focuses on another set of requirements. If something changed in the meantime or proved to be wrong, refactoring is in order. And the process continues until there's nothing more to add.

Customers and developers work together daily; feedback is solicited and delivered on a timely basis; results are immediately visible; the architect is just one of the developers; and the team is highly skilled and motivated. The length of an iteration is measured in weeks—often, two weeks. In a word, an agile process is agile to react to changes. And changes in the business are the rule, not the exception.

Agile methodologies is a blanket term. When you refer to an agile methodology, you aren't talking very precisely. Which methodology do you mean, actually?

The most popular agile methodology for software development is Extreme Programming (XP). In XP, phases are carried out in extremely short iterations that take two weeks to terminate. Coding and design proceed side by side. For more information on XP, visit the site *http://www.extremeprogramming.org*.

Scrum is another popular agile methodology, but it is aimed at managing projects rather than developing code. Scrum is not prescriptive for any software development model, but it works very well with XP as the method to develop code. For more information on Scrum, have a look at *Agile Project Management with Scrum* by Ken Schwaber (Microsoft Press, 2004).

Microsoft Solutions Framework

Microsoft Solutions Framework (MSF) is another methodology for software development, like XP or waterfall. Like XP, MSF has its own principles, roles, and vocabulary. In particular, the roles in MSF are Program Manager, Architect, Developer, Tester, Release Manager, DBA, and Business Analyst. Typical terms are iteration, release, phase, and work item (to indicate an activity within the project).

MSF is a methodology that Microsoft developed and has used internally for the past ten years. Since 2006, it has also been supported by Team Foundation Server (TFS).

TFS is essentially a collection of Web services within an Internet Information Server (IIS) Web server that provide business logic specific to project management. Either through the TFS console (if you have proper administrative rights) or through a Visual Studio plug-in, you can create a TFS project and use TFS services to manage your project. Great, but what is the methodology being used?

TFS provides only one methodology out of the box: MSF. However TFS plug-ins exist to add other methodologies to TFS. In particular, plug-ins exist for the Rational Unified Process (RUP), Feature Driven Development (FDD), and Scrum.

When the project manager creates a TFS project, he is first asked to pick up an available methodology (say, MSF). When MSF is picked up, the project manager is also asked to choose which flavor of MSF he likes. There are two flavors: MSF for Agile, and MSF for CMMI.

Note CMMI is an acronym for Capability Maturity Model Integration. CMMI is a general methodology for improving processes within a company. CMMI focuses on processes and fights common misconceptions such as "good people are enough" and "processes hinder agility." CMMI proposes a set of best practices and a framework for organizing and prioritizing activities, with the purpose of improving processes related to the building of a product.

In essence, you opt for a model that is more agile or more rigorous. In the context of MSF, the word *agile* has exactly the meaning it has in an English dictionary. It is not necessarily related to agile methodologies.

For example, in MSF for Agile you don't give work items an explicit duration in terms of hours; instead, you use an integer to indicate the level of effort (order of magnitude) required. You have to use hours in MSF for CMMI, instead. In MSF for Agile, a Developer can assign a task to another Developer; in MSF for CMMI, only the project manager has a similar right. In an agile process, therefore, it is *assumed* that such an action is accomplished with due forethought. In MSF for Agile, a work item can be moved from one project area to another without problems. This might not be true for another methodology.

In general, MSF for Agile is designed for small teams working iteratively on a project. MSF for CMMI is more appropriate for large and heterogeneous teams, working on long iterations and particularly concerned with control of quality.

Summary

Architecture is a widely used term that has quite a few definitions. If you read between the lines, though, you mostly find variations of the same concept: architecture refers to identifying the software components that, when interacting, make the program work.

In the process of identifying these components, you encounter points of decision making. When you design an architecture, not all decisions you make have the same impact. The approach to the design of the business logic, for example, is something you can hardly change at a later time in an inexpensive way. So architecture is about components and hard-to-change decisions.

The design of an architecture is qualified by a number of quality parameters that are part of an international standard. The design of the architecture comes out of functional and nonfunctional requirements, gathered by business analysts and acknowledged by architects.

Who's the architect and what are his responsibilities? The role of the architect is different from that of an analyst or a project manager, but sometimes the same individual can play both roles in the context of a specific project. Does an architect write code? Oh, yes. In our vision, an architect is a born developer, and even if the architect doesn't write much, or any, production code, he definitely practices with deep code.

The role of the architect, and the way in which the architect and the development team work with requirements, largely depends on the methodology in use—whether it is agile or traditional.

In this chapter, we mentioned in some places the Unified Modeling Language (UML) as the primary notation to describe architectural aspects of a system. In the next chapter, we'll take a closer look at the UML language.

Murphy's Laws of the Chapter

Murphy's laws are the portrait of the real world. If anything happens repeatedly in the real world, it is then captured in a law. Software projects are a part of the real world and it is not surprising that laws exist to describe software-related phenomena. In all chapters, therefore, we'll be listing a few Murphy's laws.

- Adding manpower to a late software project makes it later.
- Program complexity grows until it exceeds the capability of the programmers who must maintain it.
- If builders built buildings the way programmers wrote programs, the first woodpecker that came along would destroy civilization.

See *http://www.murphys-laws.com* for an extensive listing of other computer-related (and non-computer-related) laws and corollaries.

Chapter 3

Design Principles and Patterns

Experienced designers evidently know something inexperienced others don't. What is it?

—Erich Gamma

In Chapter 1, "Architects and Architecture Today," we focused on the true meaning of architecture and the steps through which architects get a set of specifications for the development team. We focused more on the process than the principles and patterns of actual design. In Chapter 2, "UML Essentials," we filled a gap by serving up a refresher (or a primer, depending on the reader's skills) of Unified Modeling Language (UML). UML is the most popular modeling language through which design is expressed and communicated within development teams.

When examining the bundle of requirements, the architect at first gets a relatively blurred picture of the system. As the team progresses through iterations, the contours of the picture sharpen. In the end, the interior of the system unveils a web of interrelated classes applying design patterns and fulfilling design principles.

Designing a software system is challenging because it requires you to focus on today's requested features while ensuring that the resulting system be flexible enough to support changes and addition of new features in the future.

Especially in the past two decades, a lot has been done in the Information Technology (IT) industry to make a systematic approach to software development possible. Methodologies, design principles, and finally patterns have been developed to help guide architects to envision and build systems of any complexity in a disciplined way.

This chapter aims to provide you with a quick tutorial about software engineering. It first outlines some basic principles that should always inspire the design of a modern software system. The chapter then moves on to discuss principles of object-oriented design. Along the way, we introduce patterns, idioms, and aspect-orientation, as well as pearls of wisdom regarding requirement-driven design that affect key areas such as testability, security, and performance.

Basic Design Principles

It is one thing to write code that just works. It is quite another to write *good* code that works. Adopting the attitude of "writing good code that works" springs from the ability to view the system from a broad perspective. In the end, a top-notch system is not just a product of writing instructions and hacks that make it all work. There's much more, actually. And it relates, directly or indirectly, to design.

The attitude of "writing good code that works" leads you, for example, to value the maintainability of the code base over any other quality characteristics, such as those defined by International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) standard 9126. (See Chapter 1.) You adopt this preference not so much because other aspects (such as extensibility or perhaps scalability) are less important than maintainability—it's just that maintenance is expensive and can be highly frustrating for the developers involved.

A code base that can be easily searched for bugs, and in which fixing bugs is not problematic for anyone, is open to any sort of improvements at any time, including extensibility and scalability. Thus, maintainability is the quality characteristic you should give the highest priority when you design a system.

Why is software maintenance so expensive?

Maintenance becomes expensive if essentially you have produced unsatisfactory (should we say, sloppy?) software, you haven't tested the software enough, or both. Which attributes make software easier to maintain and evolve? Structured design in the first place, which is best applied through proper coding techniques. Code readability is another fundamental asset, which is best achieved if the code is combined with a bunch of internal documentation and a change-tracking system—but this might occur only in a perfect world.

Before we proceed any further with the basic principles of structured design, let's arrange a brief cheat-sheet to help us catch clear and unambiguous symptoms of bad code design.

Note Unsatisfactory software mostly springs from a poor design. But what causes a poor design? A poor design typically has two causes that are not mutually exclusive: the architect's insufficient skills, and imprecise or contradictory requirements. So what about the requirements problem, then? Contradictory requirements usually result from bad communication. Communication is king, and it is one of the most important skills for an architect to cultivate and improve.

Not surprisingly, fixing this communication problem drives us again straight to agile methodologies. What many people still miss about the agile movement is that the primary benefit you get is not so much the iterative method itself. Instead, the major benefit comes from the continuous communication that the methodology promotes within the team and between the team and the customers. Whatever you get wrong in the first iteration will be fixed quite soon in the next (or close to the next) iteration because the communication that is necessary to move forward will clarify misunderstood requirements and fix bad ones. And it will do so quite early in the process and on a timely basis. This iterative approach simply reduces the entry point for the major cause of costly software maintenance: poor communication. And this is the primary reason why, one day, a group of (perfectly sane) developers and architects decided to found the agile movement. It was pragmatism that motivated them, not caprice.

This said, you should also keep in mind that that agile methodologies also tend to increase development costs and run the risk of scope/requirements creep. You also must make sure everyone in the process is on board with it. If the stakeholders don't understand their role or are not responsive, or can't review the work between iterations, the agile approach fails. So the bottom line is that the agile approach isn't a magic wand that works for everyone. But when it works, it usually works well.

For What the Alarm Bell Should Ring

Even with the best intentions of everyone involved and regardless of their efforts, the design of a system at some point can head down a slippery slope. The deterioration of a good design is generally a slow process that occurs over a relatively long period of time. It happens by continually studding your classes with hacks and workarounds, making a large share of the code harder and harder to maintain and evolve. At a certain point, you find yourself in serious trouble.

Managers might be tempted to call for a complete redesign, but redesigning an evolving system is like trying to catch a runaway chicken. You need to be in a very good shape to do it. But is the team really in shape at that point?

Note Have you ever seen the movie *Rocky*? Do you remember the scene where Rocky, the boxer, finally catches the chicken, thus providing evidence that he's ready for the match? By the way, the scene is on *http://www.youtube.com/watch?v=o8ZkY7tnpRs*. During the movie, Rocky attempts several times to get the chicken, but he gets the chicken only when he has trained well enough.

Let's identify a few general signs that would make the alarm bell ring to warn of a problematic design.

Rigid, Therefore Fragile

Can you bend a piece of wood? What do you risk if you insist on doing it? A piece of wood is typically a stiff and rigid object characterized by some resistance to deformation. When enough force is applied, the deformation becomes permanent and the wood breaks.

What about rigid software?

Rigid software is characterized by some resistance to changes. Resistance is measured in terms of regression. You make a change in one module, but the effects of your change cascade down the list of dependent modules. As a result, it's really hard to predict how long making a change—any change, even the simplest—will actually take.

If you pummel glass or any other fragile material, you manage only to break it into several pieces. Likewise, when you enter a change in software and break it in various places, it becomes quite apparent that software is definitely fragile.

As in other areas of life, in the software world fragility and rigidity go hand in hand. When a change in a software module breaks (many) other modules because of (hidden) dependencies, you have a clear symptom of a bad design that needs to be remedied as soon as possible.

Easier to Use Than to Reuse

Imagine you have a piece of software that works in one project; you would like to reuse it in another project. However, copying the class or linking the assembly in the new project just doesn't work.

Why is it so?

If the same code doesn't work when moved to another project, it's because of dependencies. The real problem isn't just dependencies, but the number and depth of dependencies. The risk is that to reuse a piece of functionality in another project, you have to import a much larger set of functions. Ultimately, no reuse is ever attempted and code is rewritten from scratch.

This is not a good sign for your design. This negative aspect of a design is often referred to as *immobility*.

Easier to Work Around Than to Fix

When applying a change to a software module, it is not unusual that you figure out two or more ways to do it. Most of the time, one way of doing things is nifty, elegant, coherent with the design, but terribly laborious to implement. The other way is, conversely, much smoother, quick to code, but sort of a hack.

What should you do?

Actually, you can solve it either way, depending on the given deadlines and your manager's direction about it.

In summary, it is not an ideal situation when a workaround is much easier and faster to apply than the right solution. And it doesn't make a great statement about your overall design, either. It is a sign that too many unnecessary dependencies exist between classes and that your classes do not form a particularly cohesive mass of code.

This aspect of a design—that it invites or accommodates workarounds more or less than fixes—is often referred to as *viscosity*. High viscosity is bad, meaning that the software resists modification just as highly viscous fluids resist flow.

Structured Design

When the two of us started programming, which was far before we started making a living from it, the old BASIC language was still around with its set of GOTO statements. Like many others, we wrote toy programs jumping from one instruction to the next within the same monolithic block of code. They worked just fine, but they were only toy programs in the end.

Note Every time we looked at the resulting messy BASIC code we wrote, continually referring to other instructions that appeared a bunch of lines up or down in the code, we didn't really like it and we weren't really proud of it. But, at the time, we just thought we were picking up a cool challenge that only a few preordained souls could take on. Programming is a darned hard thing—we thought—but we are going to like it.

It was about the late 1960s when the complexity of the average program crossed the significant threshold that marked the need for a more systematic approach to software development. That signaled the official beginning of software engineering.

From Spaghetti Code to Lasagna Code

Made of a messy tangle of jumps and returns, GOTO-based code was soon belittled and infamously labeled as *spaghetti code*. And we all learned the first of a long list of revolutionary concepts: structured programming. In particular, we learned to use *subroutines* to break our code into cohesive and more reusable pieces. In food terms, we evolved from *spaghetti* to *lasagna*. If you look at Figure 3-1, you will spot the difference quite soon. Lasagna forms a layered block of noodles and toppings that can be easily cut into pieces and just exudes the concept of structure. Lasagna is also easier to serve, which is the food analogy for reusability.

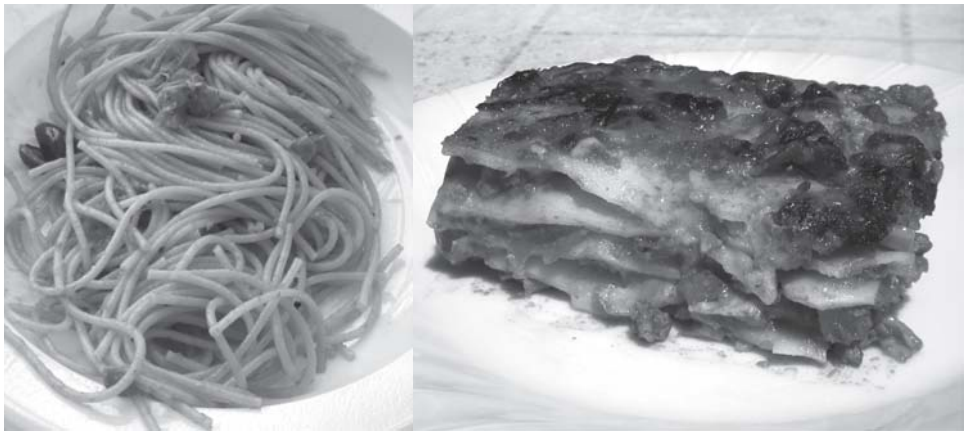


FIGURE 3-1 From a messy tangle to a layered and ordered block

Note A small note (and some credits) about the figure is in order. First, as Italians we would have used the term *lasagne*, which is how we spell it, but we went for the international spelling of *lasagna*. However, we eat it regardless of the spelling. Second, Dino personally ate all the food in the figure in a sort of manual testing procedure for the book's graphics. Dino, however, didn't cook anything. Dino's mother-in-law cooked the spaghetti; Dino's mom cooked the lasagna. Great stuff—if you're in Italy, and want to give it a try, send Dino an e-mail.

What software engineering really has been trying to convey since its inception is the need for some design to take place before coding begins and, subsequently, the need for some basic design principles. Still, today, when someone says "structured programming," immediately many people think of subroutines. This assumption is correct, but it's oversimplifying the point and missing the principal point of the structured approach.

Behind structured programming, there is structured design with two core principles. And these principles are as valid today as they were 30 and more years ago. Subroutines and Pascal-like programming are gone; the principles of *cohesion* and *coupling*, instead, still maintain their effectiveness in an object-oriented world.

These principles of structured programming, coupling and cohesion, were first introduced by Larry Constantine and Edward Yourdon in their book *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Yourdon Press, 1976).

Cohesion

Cohesion indicates that a given software module—be it a subroutine, class, or library—features a set of responsibilities that are strongly related. Put another way, cohesion measures the distance between the logic expressed by the various methods on a class, the various functions in a library, and the various actions accomplished by a method.

If you look for a moment at the definition of cohesion in another field—chemistry—you should be able to see a clearer picture of software cohesion. In chemistry, cohesion is a physical property of a substance that indicates the attraction existing between like molecules within a body.

Cohesion measurement ranges from low to high and is preferably in the highest range possible.

Highly cohesive modules favor maintenance and reusability because they tend to have no dependencies. Low cohesion, on the other hand, makes it much harder to understand the purpose of a class and creates a natural habitat for rigidity and fragility in the software. Low cohesive modules also propagate dependencies through modules, thus contributing to the immobility and high viscosity of the design.

Decreasing cohesion leads to creating modules (for example, classes) where responsibilities (for example, methods) have very little in common and refer to distinct and unrelated activities. Translated in a practical guideline, the principle of cohesion recommends creating extremely specialized classes with few methods, which refer to logically related operations. If the logical distance between methods grows, you just create a new class.

Ward Cunningham—a pioneer of Extreme Programming—offers a concise and pragmatic definition of cohesion in his wiki at *http://c2.com/cgi/wiki?CouplingAndCohesion*. He basically says that two modules, A and B, are cohesive when a change to A has no repercussion for B so that both modules can add new value to the system.

There's another quote we'd like to use from Ward Cunningham's wiki to reinforce a concept we expressed a moment ago about cohesion. Cunningham suggests that we define cohesion

as inversely proportional to the number of responsibilities a module (for example, a class) has. We definitely like this definition.

Important Strongly related to cohesion is the Single Responsibility Principle (SRP). In the formulation provided by Robert Martin (which you can see at *http://www.objectmentor.com/resources/articles/srp.pdf*), SRP indicates that each class should always have just one reason to change. In other words, each class should be given a single responsibility, where a responsibility is defined as "a reason to change." A class with multiple responsibilities has more reasons to change and, subsequently, a less cohesive interface. A correct application of SRP entails breaking the methods of a class into logical subsets that configure distinct responsibilities. In the real world, however, this is much harder to do than the opposite—that is, aggregating distinct responsibilities in the same class.

Coupling

Coupling measures the level of dependency existing between two software modules, such as classes, functions, or libraries. An excellent description of coupling comes, again, from Cunningham's wiki at *http://c2.com/cgi/wiki?CouplingAndCohesion*. Two modules, A and B, are said to be coupled when it turns out that you have to make changes to B every time you make any change to A.

In other words, B is not directly and logically involved in the change being made to module A. However, because of the underlying dependency, B is forced to change; otherwise, the code won't compile any longer.

Coupling measurement ranges from low to high and the lowest possible range is preferable.

Low coupling doesn't mean that your modules are to be completely isolated from one another. They are definitely allowed to communicate, but they should do that through a set of well-defined and stable interfaces. Each module should be able to work without intimate knowledge of another module's internal implementation.

Conversely, high coupling hinders testing and reusing code and makes understanding it nontrivial. It is also one of the primary causes of a rigid and fragile design.

Low coupling and high cohesion are strongly correlated. A system designed to achieve low coupling and high cohesion generally meets the requirements of high readability, maintainability, easy testing, and good reuse.

Note Introduced to support a structured design, cohesion and coupling are basic design principles not specifically related to object orientation. However, it's the general scope that also makes them valid and effective in an object-oriented scenario. A good object-oriented design, in fact, is characterized by low coupling and high cohesion, which means that self-contained objects (high cohesion) are interacting with other objects through a stable interface (low coupling).

Separation of Concerns

So you know you need to cook up two key ingredients in your system's recipe. But is there a supermarket where you can get both? How do you achieve high cohesion and low coupling in the design of a software system?

A principle that is helpful to achieving high cohesion and low coupling is separation of concerns (SoC), introduced in 1974 by Edsger W. Dijkstra in his paper "On the Role of Scientific Thought." If you're interested, you can download the full paper from *http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF*.

Identifying the Concerns

SoC is all about breaking the system into distinct and possibly nonoverlapping features. Each feature you want in the system represents a *concern* and an *aspect* of the system. Terms such as feature, concern, and aspect are generally considered synonyms. Concerns are mapped to software modules and, to the extent that it is possible, there's no duplication of functionalities.

SoC suggests that you focus on one particular concern at a time. It doesn't mean, of course, that you ignore all other concerns of the system. More simply, after you've assigned a concern to a software module, you focus on building that module. From the perspective of that module, any other concerns are irrelevant.

Note If you read Dijkstra's original text, you'll see that he uses the expression "separation of concerns" to indicate the general principle, but switches to the word "aspect" to indicate individual concerns that relate to a software system. For quite a few years, the word "aspect" didn't mean anything special to software engineers. Things changed in the late 1990s when *aspect-oriented programming* (AOP) entered the industry. We'll return to AOP later in this chapter, but we make the forward reference here to show Dijkstra's great farsightedness.

Modularity

SoC is concretely achieved through using modular code and making heavy use of *information hiding*.

Modular programming encourages the use of separate modules for each significant feature. Modules are given their own public interface to communicate with other modules and can contain internal chunks of information for private use.

Only members in the public interface are visible to other modules. Internal data is either not exposed or it is encapsulated and exposed in a filtered manner. The implementation of the interface contains the behavior of the module, whose details are not known or accessible to other modules.

Information Hiding

Information hiding (IH) is a general design principle that refers to hiding behind a stable interface some implementation details of a software module that are subject to change. In this way, connected modules continue to see the same fixed interface and are unaffected by changes.

A typical application of the information-hiding principle is the implementation of properties in C# or Microsoft Visual Basic .NET classes. (See the following code sample.) The property name represents the stable interface through which callers refer to an internal value. The class can obtain the value in various ways (for example, from a private field, a control property, a cache, the view state in ASP.NET) and can even change this implementation detail without breaking external code.

```
// Software module where information hiding is applied
public class Customer
{
   // Implementation detail being hidden
   private string _name;

   // Public and stable interface
   public string CustomerName
   {
        // Implementation detail being hidden
        get {return _name;}
   }
}
```

Information hiding is often referred to as *encapsulation*. We like to distinguish between the principle and its practical applications. In the realm of object-oriented programming, encapsulation is definitely an application of IH.

Generally, though, the principle of SoC manifests itself in different ways in different programming paradigms, and so it is for modularity and information hiding.

SoC and Programming Paradigms

The first programming paradigm that historically supported SoC was *Procedural Programming* (PP), which we find expressed in languages such as Pascal and C. In PP, you separate concerns using functions and procedures.

Next—with the advent of object-oriented programming (OOP) in languages such as Java, C++, and more recently C# and Visual Basic .NET—you separate concerns using classes.

However, the concept isn't limited to programming languages. It also transcends the realm of pure programming and is central in many approaches to software architecture. In a service-oriented architecture (SOA), for example, you use services to represent concerns. Layered architectures are based on SoC, and within a middle tier you can use an Object/Relational Mapping tool (O/RM) to separate persistence from the domain model.

Note In the preceding section, we basically went back over 40 years of computer science, and the entire sector of software engineering. We've seen how PP, OOP, and SOA are all direct or indirect emanations of the SoC principle. (Later in this chapter, we'll see how AOP also fits this principle. In Chapter 7, "The Presentation Layer," we'll see how fundamental design patterns for the presentation layer, such as Model-View-Controller and Model-View-Presenter, also adhere to the SoC principle.)

You really understand the meaning of the word *principle* if you look at how SoC influenced, and still influences, the development of software. And we owe this principle to a great man who passed away in 2002: Edsger W. Dijkstra. We mention this out of respect for this man.

For more information about Dijkstra's contributions to the field, pay a visit to *http://www.cs.utexas.edu/users/ewd*.

Naming Conventions and Code Readability

When the implementation of a line-of-business application is expected to take several months to complete and the final application is expected to remain up and running for a few years, it is quite reasonable to expect that many different people will work on the project over time.

With such significant personnel turnover in sight, you must pay a lot of attention to system characteristics such as readability and maintainability. To ensure that the code base is manageable as well as easily shared and understood, a set of common programming rules and conventions should be used. Applied all the way through, common naming conventions, for example, make the whole code base look like it has been written by a single programmer rather than a very large group of people.

The most popular naming convention is Hungarian Notation (HN). You can read more about it at *http://en.wikipedia.org/wiki/Hungarian_Notation*. Not specifically bound to a programming language, HN became quite popular in the mid-1990s, as it was largely used in many Microsoft Windows applications, especially those written directly against the Windows Software Development Kit (SDK).

HN puts the accent on the type of the variable, and it prefixes the variable name with a mnemonic of the type. For example, *szUserName* would be used for a zero-terminated string that contains a user name, and *iPageCount* would be used for an integer that indicates the number of pages. Created to make each variable self-explanatory, HN lost most of its appeal with the advent of object-oriented languages.

In object-oriented languages, everything is an object, and putting the accent on the value, rather than the type, makes much more sense. So you choose variable names regardless of the type and look only at the value they are expected to contain. The choice of the variable name happens in a purely evocative way. Therefore, valid names are, for example, *customer*, *customerID*, and *lowestPrice*.

Finally, an argument against using HN is that a variable name should be changed every time the type of the variable changes during development. In practice, this is often difficult or overlooked, leading developers to make incorrect assumptions about the values contained within the variables. This often leads directly to bugs.

You can find detailed design guidelines for the .NET Framework classes and applications at *http://msdn.microsoft.com/en-us/library/ms229042.aspx*.

Object-Oriented Design

Before object orientation (OO), any program resulted from the interaction of modules and routines. Programming was procedural, meaning that there was a main stream of code determining the various steps to be accomplished.

OO is a milestone in software design.

OO lets you envision a program as the result of interacting objects, each of which holds its own data and behavior. How would you design a graph of objects to represent your system? Which principles should inspire this design?

We can recognize a set of core principles for object-oriented design (OOD) and a set of more advanced and specific principles that descend from, and further specialize, the core principles.

Basic OOD Principles

To find a broadly accepted definition of OOD, we need to look at the Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) and their landmark book *Design Patterns: Elements of Reusable Object-Oriented Software*, (Addison-Wesley, 1994). (We'll make further references to this book as GoF, which is the universal acronym for "Gang of Four.")

The entire gist of OOD is contained in this sentence:

> *You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them.*

In GoF, we also find another excerpt that is particularly significant:

> *Your design should be specific to the problem at hand but also general enough to address future problems and requirements.*

Wouldn't you agree that this last sentence is similar to some of the guidelines resulting from the ISO/IEC 9126 standard that we covered in Chapter 1? Its obvious similarity to that standard cannot be denied, and it is not surprising at all.

The basics of OOD can be summarized in three points: find pertinent objects, favor low coupling, and favor code reuse.

Find Pertinent Objects First

The first key step in OOD is creating a crisp and flexible abstraction of the problem's domain. To successfully do so, you should think about things instead of processes. You should focus on the *what*s instead of the *how*s. You should stop thinking about algorithms to focus mostly on interacting entities. Interacting entities are your pertinent objects.

Where do you find them?

Requirements offer the raw material that must be worked out and shaped into a hierarchy of pertinent objects. The descriptions of the use cases you receive from the team of analysts provide the foundation for the design of classes. Here's a sample use case you might get from an analyst:

> *To view all **orders** placed by a **customer**, the **user** indicates the **customer ID**. The program displays an error message if the customer does not exist. If the customer exists, the program displays **name**, **address**, **date of birth**, and all outstanding **orders**. For each order, the program gets **ID**, **date**, and all **order items**.*

A common practice for finding pertinent objects is tagging all nouns and verbs in the various use cases. Nouns originate classes or properties, whereas verbs indicate methods on classes. Our sample use case suggests the definition of classes such as *User*, *Customer*, *Order*, and *OrderItem*. The class *Customer* will have properties such as *Name*, *Address*, and *DateOfBirth*. Methods on the class *Customer* might be *LoadOrderItems*, *GetCustomerByID*, and *LoadOrders*.

Note that finding pertinent objects is only the first step. As recommended in the statement that many consider to be the emblem of OOD, you then have to factor pertinent objects into classes and determine the right level of granularity and assign responsibilities.

In doing so, two principles of OOD apply, and they are listed in the introduction of GoF.

Favor Low Coupling

In an OO design, objects need to interact and communicate. For this reason, each object exposes its own public interface for others to call. So suppose you have a logger object with a method *Log* that tracks any code activity to, say, a database. And suppose also that

another object at some point needs to log something. Simply enough, the caller creates an instance of the logger and proceeds. Overall, it's easy and effective. Here's some code to illustrate the point:

```
class MyComponent
{
  void DoSomeWork()
  {
    // Get an instance of the logger
    Logger logger = new Logger();

    // Get data to log
    string data = GetData();

    // Log
    logger.Log(data);
  }
}
```

The class *MyComponent* is tightly coupled to the class *Logger* and its implementation. The class *MyComponent* is broken if *Logger* is broken and, more importantly, you can't use another type of logger.

You get a real design benefit if you can separate the interface from the implementation.

What kind of functionality do you really need from such a logger component? You essentially need the ability to log; where and how is an implementation detail. So you might want to define an *ILogger* interface, as shown next, and extract it from the *Logger* class:

```
interface ILogger
{
    void Log(string data);
}

class Logger : ILogger
{
    ⋮
}
```

At this point, you use an intermediate factory object to return the logger to be used within the component:

```
class MyComponent
{
  void DoSomeWork()
  {
    // Get an instance of the logger
    ILogger logger = Helpers.GetLogger();
```

```
    // Get data to log
    string data = GetData();

    // Log
    logger.Log(data);
  }
}

class Helpers
{
  public static ILogger GetLogger()
  {
    // Here, use any sophisticated logic you like
    // to determine the right logger to instantiate.

    ILogger logger = null;
    if (UseDatabaseLogger)
    {
        logger = new DatabaseLogger();
    }
    else
    {
        logger = new FileLogger();
    }
    return logger;
  }
}
class FileLogger : ILogger
{
   ⋮
}

class DatabaseLogger : ILogger
{
   ⋮
}
```

The factory code gets you an instance of the logger for the component to use. The factory returns an object that implements the *ILogger* interface, and the component consumes any object that implements the contracted interface.

The dependency between the component and the logger is now based on an interface rather than an implementation.

If you base class dependencies on interfaces, you minimize coupling between classes to the smallest possible set of functions—those defined in the interface. In doing so, you just applied the first principle of OOD as outlined in GoF:

> *Program to an interface, not an implementation.*

This approach to design is highly recommended for using with the parts of your code that are most likely to undergo changes in their implementation.

Note Should you use an interface? Or should you perhaps opt for an abstract base class? In object-oriented languages that do not support multiple inheritance—such as Java, C#, and Visual Basic .NET—an interface is always preferable because it leaves room for another base class of your choice. When you have multiple inheritance, it is mostly a matter of preference. You should consider using a base class in .NET languages in all cases where you need more than just an interface. If you need some hard-coded behavior along with an interface, a base class is the only option you have. ASP.NET providers, for example, are based on base classes and not on interfaces.

An interesting possibility beyond base classes and interfaces are mixins, but they are an OOP feature not supported by .NET languages. A mixin is a class that provides a certain functionality that other classes can inherit, but it is not meant to be a standalone class. Put another way, a mixin is like an interface where some of the members might contain a predefined implementation. Mixins are supported in some dynamic languages, including Python and Ruby. No .NET languages currently support mixins, but mixins can be simulated using ad hoc frameworks such as Castle.DynamicProxy. With this framework, you first define a class that contains all the methods you want to inject in an existing class—the mixin. Next, you use the framework to create a proxy for a given class that contains the injected methods. Castle.DynamicProxy uses *Reflection.Emit* internally to do the trick.

Real-World Example: *IButtonControl* in ASP.NET

In ASP.NET 1.*x*, there was no support for cross-page postbacks. Every time the user clicked a button, he could only post to the same page. Starting with ASP.NET 2.0, buttons (and only buttons) were given the ability to trigger the post of the current form to an external page.

To support this feature, the *Page* class needs to know whether the control that caused the postback is a button or not. How many types of buttons do you know? There's the *Button* class, but also *LinkButton* and finally *ImageButton*. Up until ASP.NET 2.0, these classes had very little in common—just a few properties, but nothing that could be officially perceived as a contract or a formal link.

Having the *Page* class check against the three types before posting would have limited the extensibility of the framework: only those three types of control would have ever been able to make a cross-page post.

The ASP.NET team extracted the core behavior of a button to the *IButtonControl* interface and implemented that interface in all button classes. Next, they instructed the *Page* class to check the interface to verify the suitability of a posting control to make a cross-page post.

In this way, you can write custom controls that implement the interface and still add the ability to make your own cross-page posts.

Favor Code Reuse

Reusability is a fundamental aspect of the object-oriented paradigm and one of the keys to its success and wide adoption. You create a class one day, and you're happy with that. Next, on another day, you inherit a new class, make some changes here and there, and come up with a slightly different version of the original class.

Is this what code reuse is all about? Well, there's more to consider.

With class inheritance, the derived class doesn't simply inherit the code of the parent class. It really inherits the context and, subsequently, it gains some visibility of the parent's state. Is this a problem?

For one thing, a derived class that uses the context it inherits from the parent can be broken by future changes to the parent class.

In addition, when you inherit from a class, you enter into a polymorphic context, meaning that your derived class can be used in any scenarios where the parent is accepted. It's not guaranteed, however, that the two classes can really be used interchangeably. What if the derived class includes changes that alter the parent's context to the point of breaking the contract between the caller and its expected (base) class? (Providing the guarantee that parent and derived classes can be used interchangeably is the goal of Liskov's principle, which we'll discuss later.)

In GoF, the authors recognize two routes to reusability—white-box and black-box reusability. The former is based on class inheritance and lends itself to the objections we just mentioned. The latter is based on *object composition*.

Object composition entails creating a new type that holds an instance of the base type and typically references it through a private member:

```
public CompositeClass
{
  private MyClass theObject;

  public CompositeClass()
  {
    // You can use any lazy-loading policy you want for instantiation.
    // No lazy loading is being used here ...
    theObject = new MyClass();
  }

  public object DoWork()
  {
    object data = theObject.DoSomeWork();

    // Do some other work
    return Process(data);
  }
```

```
  private object Process(object data)
  {
    ⋮
  }
}
```

In this case, you have a wrapper class that uses a type as a black box and does so through a well-defined contract. The wrapper class has no access to internal members and cannot change the behavior in any way—it uses the object as it is rather than changing it to do its will. External calls reach the wrapper class, and the wrapper class delegates the call internally to the held instance of the class it enhances. (See Figure 3-2.)

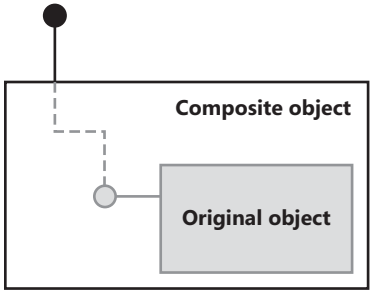


FIGURE 3-2 Object composition and delegation

When you create such a wrapper object, you basically apply the second principle of OOD:

> *Favor object composition over class inheritance.*

Does all this mean that classic class inheritance is entirely wrong and should be avoided like the plague? Using class inheritance is generally fine when all you do is add new functions to the base class or when you entirely unplug and replace an existing functionality. However, you should never lose track of the Liskov principle. (We'll get to the details of the Liskov principle in a moment.)

In many cases, and especially in real-world scenarios, object composition is a safer practice that also simplifies maintenance and testing. With composition, changes to the composite object don't affect the internal object. Likewise, changes to the internal object don't affect the outermost container as long as there are no changes to the public interface.

By combining the two principles of OOD, you can refer to the original object through an interface, thus further limiting the dependency between composite and internal objects. Composition doesn't provide polymorphism even if it will provide functionality. If polymorphism is key for you, you should opt for a white-box form of reusability. However, keep the Liskov principle clearly in mind.

Note In addition to composition, another approach is frequently used to contrast class inheritance—*aggregation*. Both aggregation and composition refer to a *has-a* relationship between two classes, whereas inheritance implies an *is-a* relationship. The difference between composition and aggregation is that with composition you have a static link between the container and contained classes. If you dispose of the container, the contained classes are also disposed of. With aggregation, the link is weaker and the container is simply associated with an external class. As a result, when the container is disposed of, the child class blissfully survives.

Advanced Principles

You cannot go to a potential customer and sing the praises of your software by mentioning that it is modular, well designed, and easy to read and maintain. These are internal characteristics of the software that do not affect the user in any way. More likely, you'll say that your software is correct, bug free, fast, easy to use, and perhaps extensible. However, you can hardly write correct, bug-free, easy-to-use, and extensible software without paying a lot of attention to the internal design.

Basic principles such as low coupling, high cohesion (along with the single responsibility principle), separation of concerns, plus the first two principles of OOD give us enough guidance about how to design a software application. As you might have noticed, all these principles are rather old (but certainly not outdated), as they were devised and formulated at least 15 years ago.

In more recent years, some of these principles have been further refined and enhanced to address more specific aspects of the design. We like to list three more advanced design principles that, if properly applied, will certainly make your code easier to read, test, extend, and maintain.

The Open/Closed Principle

We owe the Open/Closed Principle (OCP) to Bertrand Meyer. The principle addresses the need of creating software entities (whether classes, modules, or functions) that can happily survive changes. In the current version of the fictional product "This World," the continuous changes to software requirements are a well-known bug. Unfortunately, although the team is working to eliminate the bug in the next release, we still have to face reality and deal with frequent changes of requirements the best we can.

Essentially, we need to have a mechanism that allows us to enter changes where required without breaking existing code that works. The OCP addresses exactly this issue by saying the following:

> *A module should be open for extension but closed for modification.*

Applied to OOD, the principle recommends that we never edit the source code of a class that works in order to implement a change. In other words, each class should be conceived to be stable and immutable and never face change—the class is closed for modification.

How can we enter changes, then?

Every time a change is required, you enhance the behavior of the class by adding new code and never touching the old code that works. In practical terms, this means either using composition or perhaps safe-and-clean class inheritance. Note that OCP just reinforces the point that we made earlier about the second principle of OOD: if you use class inheritance, you add only new code and do not modify any part of the inherited context.

Today, the most common way to comply with the OCP is by implementing a fixed interface in any classes that we figure are subject to changes. Callers will then work against the interface as in the first principle of OOD. The interface is then closed for modification. But you can make callers interact with any class that, at a minimum, implements that interface. So the overall model is open for extension, but it still provides a fixed interface to dependent objects.

Liskov's Substitution Principle

When a new class is derived from an existing one, the derived class can be used in any place where the parent class is accepted. This is polymorphism, isn't it? Well, the Liskov Substitution Principle (LSP) restates that this is the way you should design your code. The principle says the following:

> *Subclasses should be substitutable for their base classes.*

Apparently, you get this free of charge from just using an object-oriented language. If you think so, have a look at the next example:

```
public class ProgrammerToy
{
    private int _state = 0;

    public virtual void SetState(int state)
    {
        _state = state;
    }

    public int GetState()
    {
        return _state;
    }
}
```

The class *ProgrammerToy* just acts as a wrapper for an integer value that callers can read and write through a pair of public methods. Here's a typical code snippet that shows how to use it:

```
static void DoSomeWork(ProgrammerToy toy)
{
    int magicNumber = 5;
    toy.SetState(magicNumber);
    Console.WriteLine(toy.GetState());
    Console.ReadLine();
}
```

The caller receives an instance of the *ProgrammerToy* class, does some work with it, and then displays any results. So far, so good. Let's now consider a derived class:

```
public class CustomProgrammerToy : ProgrammerToy
{
    public override void SetState(int state)
    {
        // It inherits the context of the parent but lacks the tools
        // to fully access it. In particular, it has no way to access
        // the private member _state.
        // As a result, this class MAY NOT be able to
        // honor the contract of its parent class. Whether or not, mostly
        // depends on your intentions and expected goals for the overridden
        // SetState method. In any case, you CAN'T access directly the private member
        // _state from within this override of SetState.

        // (In .NET, you can use reflection to access a private member,
        // but that's a sort of a trick.)
        ⋮
    }
}
```

From a syntax point of view, *ProgrammerToy* and *CustomProgrammerToy* are just the same and method *DoSomeWork* will accept both and successfully compile.

From a behavior point of view, though, they are quite different. In fact, when *CustomProgrammerToy* is used, the output is 0 instead of 5. This is because of the override made on the *SetState* method.

This is purely an example, but it calls your attention to Liskov's Principle. It *doesn't* go without saying that derived classes (subclasses) can safely replace their base classes. You have to ensure that. How?

You should handle keywords such as *sealed* and *virtual* with extreme care. Virtual (overridable) methods, for example, should never gain access to private members. Access to private members can't be replicated by overrides, which makes base and derived classes not semantically equivalent from the perspective of a caller. You should plan ahead of time which members are private and which are protected. Members consumed by virtual methods must be protected, not private.

Generally, virtual methods of a derived class should work out of the same preconditions of corresponding parent methods. They also must guarantee at least the same postconditions.

Classes that fail to comply with LSP don't just break polymorphism but also induce violations of OCP on callers.

Note OCP and LSP are closely related. Any function using a class that violates Liskov's Principle violates the Open/Close Principle. Let's reference the preceding example again. The method *DoSomeWork* uses a hierarchy of classes (*ProgrammerToy* and *CustomProgrammerToy*) that violate LSP. This means that to work properly *DoSomeWork* must be aware of which type it really receives. Subsequently, it has to be modified each time a new class is derived from *ProgrammerToy*. In other words, the method *DoSomeWork* is not closed for modification.

The Dependency Inversion Principle

When you create the code for a class, you represent a behavior through a set of methods. Each method is expected to perform a number of actions. As you specify these actions, you proceed in a top-down way, going from high-level abstractions down the stack to more and more precise and specific functions.

As an illustration, imagine a class, perhaps encapsulated in a service, that is expected to return stock quotes as a chunk of HTML markup:

```
public class FinanceInfoService
{
  public string GetQuotesAsHtml(string symbols)
  {
    // Get the Finder component
    IFinder finder = ResolveFinder();
    if (finder == null)
      throw new NullReferenceException("Invalid finder.");

    // Grab raw data
    StockInfo[] stocks = finder.FindQuoteInfo(symbols);

    // Get the Renderer component
    IRenderer renderer = ResolveRenderer();
    if (renderer == null)
       throw new NullReferenceException("Invalid renderer.");

    // Render raw data out to HTML
    return renderer.RenderQuoteInfo(stocks);
  }

  ⋮
}
```

The method *GetQuotesAsHtml* is expected to first grab raw data and then massage it into an HTML string. You recognize two functionalities in the method: the finder and the renderer. In a top-down approach, you are interested in recognizing these functionalities, but you don't need to specify details for these components in the first place. All that you need to do is hide details behind a stable interface.

The method *GetQuotesAsHtml* works regardless of the implementation of the finder and renderer components and is not dependent on them. (See Figure 3-3.) On the other hand, your purpose is to reuse the high-level module, not low-level components.

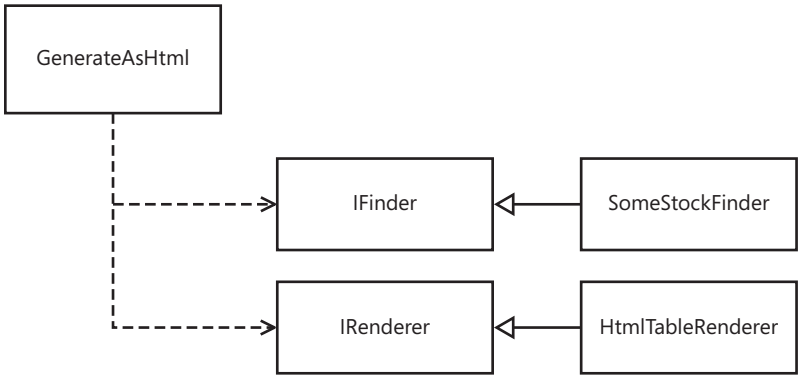


FIGURE 3-3 Lower layers are represented by an interface

When you get to this, you're in full compliance with the Dependency Inversion Principle (DIP), which states the following:

> *High-level modules should not depend upon low-level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.*

The *inversion* in the name of the principle refers to the fact that you proceed in a top-down manner during the implementation and focus on the work flow in high-level modules rather than focusing on the implementation of lower level modules. At this point, lower level modules can be injected directly into the high-level module. Here's an alternative implementation for a DIP-based module:

```
public class FinanceInfoService
{
  // Inject dependencies through the constructor. References to such external components
  // are resolved outside this module, for example by using an inversion-of-control
  // framework (more later).
  IFinder _finder = null;
  IRenderer _renderer = null;

  public FinanceInfoService(IFinder finder, IRenderer renderer)
  {
    _finder = finder;
    _renderer = renderer;
  }

  public string GetQuotesAsHtml(string symbols)
  {
    // Get the Finder component
    if (_finder == null)
      throw new NullReferenceException("Invalid finder.");
```

```
    // Grab raw data
    StockInfo[] stocks = _finder.FindQuoteInfo(symbols);

    // Get the Renderer component
    if (_renderer == null)
      throw new NullReferenceException("Invalid renderer.");

    // Render raw data out to HTML
    return _renderer.RenderQuoteInfo(stocks);
  }

  ⋮
}
```

In this case, the lower level modules are injected through the constructor of the DIP-based class.

The DIP has been formalized by Robert Martin. You can read more about it at *http://www.objectmentor.com/resources/articles/dip.pdf*.

Important In literature, the DIP is often referred to as *inversion of control* (IoC). In this book, we use the DIP formulation by Robert Martin to indicate the principle of dependency inversion and consider IoC as a pattern. In this regard, IoC and dependency injection are, for us, synonyms. The terminology, however, is much less important than the recognition that there's a principle about inversion of control and a practical pattern. We'll return on this in a moment with a more detailed explanation of our perspective.

From Principles to Patterns

It is guaranteed that by fulfilling all the OOD principles just discussed, you can craft a good design that matches requirements and is maintainable and extensible. A seasoned development team, though, will not be limited to applying effective design principles over and over again; members of the team, in fact, will certainly draw from the well of their experience any solutions for similar problems that worked in the past.

Such building blocks are nothing more than hints and the skeleton of a solution. However, these very same building blocks can become more refined day after day and are generalized after each usage to become applicable to a wider range of problems and scenarios. Such building blocks might not provide a direct solution, but they usually help you to find your (right) way. And using them is usually more effective and faster than starting from scratch.

By the way, these building blocks are known as *patterns*.

What's a Pattern, Anyway?

The word *pattern* is one of those overloaded terms that morphed from its common usage to assume a very specific meaning in computer science. According to the dictionary, a pattern is a template or model that can be used to generate things—any things. In computer science, we use patterns in design solutions at two levels: implementation and architecture.

At the highest level, two main families of software patterns are recognized: *design patterns* and *architectural patterns*. You look at design patterns when you dive into the implementation and design of the code. You look at architectural patterns when you fly high looking for the overall design of the system.

Let's start with design patterns.

Note A third family of software patterns is also worth a mention—*refactoring patterns*. You look at these patterns only when you're engaged in a refactoring process. Refactoring is the process of changing your source code to make it simpler, more efficient, and more readable while preserving the original functionality. Examples of refactoring patterns are "Extract Interface" and "Encapsulate Field." Some of these refactoring patterns have been integrated into Visual Studio 2008 on the Refactor menu. You find even more patterns in ad hoc tools such as Resharper. (For more information, see *http://www.jetbrains.com/resharper*.)

A good book to read to learn about refactoring patterns is *Refactoring to Patterns* by Joshua Kerievsky (Addison-Wesley, 2004).

Design Patterns

We software professionals owe design patterns to an architect—a *real* architect, not a software architect. In the late 1970s, Christopher Alexander developed a pattern language with the purpose of letting individuals express their innate sense of design through a sort of informal grammar. From his work, here's the definition of a pattern:

> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice.*

Nicely enough, although the definition was not written with software development in mind, it applies perfectly to that. So what's a design pattern?

A design pattern is a known and well-established *core* solution applicable to a family of concrete problems that might show up during implementation. A design pattern is a core solution and, as such, it might need adaptation to a specific context. This feature becomes a major strength when you consider that, in this way, the same pattern can be applied many times in many slightly different scenarios.

Design patterns are not created in a lab; quite the reverse. They originate from the real world and from the direct experience of developers and architects. You can think of a design pattern as a package that includes the description of a problem, a list of actors participating in the problem, and a practical solution.

The primary reference for design patterns is GoF. Another excellent reference we want to recommend is *Pattern-Oriented Software Architecture* by Frank Buschmann, et al. (Wiley, 1996).

How to Work with Design Patterns

Here is a list of what design patterns are *not*:

- Design patterns are not the verb and should never be interpreted dogmatically.
- Design patterns are not Superman and will never magically pop up to save a project in trouble.
- Design patterns are neither the dark nor the light side of the Force. They might be with you, but they won't provide you with any special extra power.

Design patterns are just helpful, and that should be enough.

You don't choose a design pattern; the most appropriate design pattern normally emerges out of your refactoring steps. We could say that the pattern is buried under your classes, but digging it out is entirely up to you.

The wrong way to deal with design patterns is by going through a list of patterns and matching them to the problem. Instead, it works the other way around. You have a problem and you have to match the problem to the pattern. How can you do that? It's quite simple to explain, but it's not so easy to apply.

You have to understand the problem and generalize it.

If you can take the problem back to its roots, and get the gist of it, you'll probably find a tailor-made pattern just waiting for you. Why is this so? Well, if you really reached the root of the problem, chances are that someone else did the same in the past 15 years (the period during which design patterns became more widely used). So the solution is probably just there for you to read and apply.

This observation prompts us to mention the way in which all members of our teams use books on design patterns. (By the way, there are always plenty of such books scattered throughout the office.) Design patterns books are an essential tool. But we never *read* such books. We *use* them, instead, like cookbooks.

What we normally do is stop reading after the first few pages precisely where most books list the patterns they cover in detail inside. Next, we put the book aside and possibly within reach. Whenever we encounter a problem, we try to generalize it, and then we flip through

the pages of the book to find a pattern that possibly matches it. We find one much more often than not. And if we don't, we repeat the process in an attempt to come to a better generalization of the problem.

When we've found the pattern, we start working on its adaptation to our context. This often requires refactoring of the code which, in turn, might lead to a more appropriate pattern. And the loop goes on.

Note If you're looking for an online quick reference about design patterns, you should look at *http://www.dofactory.com*. Among other things, the site offers .NET-specific views of most popular design patterns.

Where's the Value in Patterns, Exactly?

Many people would agree in principle that there's plenty of value in design patterns. Fewer people, though, would be able to indicate what the value is and where it can be found.

Using design patterns, per se, doesn't make your solution more valuable. What really matters, at the end of the day, is whether or not your solution works and meets requirements.

Armed with requirements and design principles, you are up to the task of solving a problem. On your way to the solution, though, a systematic application of design principles to the problem sooner or later takes you into the immediate neighborhood of a known design pattern. That's a certainty because, ultimately, patterns are solutions that others have already found and catalogued.

At that point, you have a solution with some structural likeness to a known design pattern. It is up to you, then, to determine whether an explicit refactoring to that pattern will bring some added value to the solution. Basically, you have to decide whether or not the known pattern you've found represents a further, and desirable, refinement of your current solution. Don't worry if your solution doesn't match a pattern. It means that you have a solution that works and you're happy with that. You're just fine. You never want to change a winning solution!

In summary, patterns might be an *end* when you refactor according to them, and they might be a *means* when you face a problem that is clearly resolved by a particular pattern. Patterns are not an added value for your solution, but they are valuable for you as an architect or a developer looking for a solution.

Applied Design Patterns

We said a lot about design patterns, but we haven't shown a single line of code or a concrete example. Patterns are everywhere, even if you don't realize it. As we'll see in

a moment, sometimes patterns are buried in the language syntax—in which case, we'll call them *idioms*.

Have you ever needed to use a global object (or a few global objects) to serve all requests to a given class? If you have, you used the Singleton pattern. The Singleton pattern is described as a way to ensure that a class has only one instance for which a global point of access is required. Here's an example:

```
public class Helpers
{
  public static Helpers DefaultInstance = new Helpers();

  protected Helpers() {}

  public void DoWork()
  {
    ⋮
  }

  public void DoMoreWork()
  {
    ⋮
  }
}
```

In a consumer class, you take advantage of *Helpers* through the following syntax:

```
Helpers.DefaultInstance.DoWork();
```

Swarms of Visual Basic 6 developers have used the Singleton pattern for years probably without ever realizing it. The Singleton pattern is behind the default instance of Visual Basic 6 forms, as shown here:

```
Form1.Show()
```

The preceding code in Visual Basic 6 invokes the *Show* method on the default instance of the type *Form1*. In the source, there's no explicit mention of the default instance only because of the tricks played by the Visual Basic runtime.

Tip Admittedly, the Singleton pattern on a class is similar to defining the same class with only static methods. Is there any difference?

With a Singleton pattern, you can actually control the number of instances because you're not actually limited to just one instance. In addition, you can derive a new (meaningful) class because the Singleton pattern has some instance-level behavior and is not a mere collection of static functions. Finally, you have more freedom to control the creation of the actual instance. For example, you can add a static method, say, *GetInstance*, instead of the static field and add there any logic for the factory.

Another interesting pattern to briefly mention is the Strategy pattern. The pattern identifies a particular functionality that a class needs and can be hot-plugged into the class. The functionality is abstracted to an interface or a base class, and the Strategy-enabled class uses it through the abstraction, as shown here:

```
public class MyService
{
  // This is the replaceable strategy
  ILogger _logger;

  public MyService(ILogger logger)
  {
    this._logger = logger;
  }

  public void DoWork()
  {
    this._logger.Log("Begin method ...");
    ⋮
    this._logger.Log("End method ...");
  }
}
```

The Strategy pattern is the canonical example used to illustrate the power of composition. The class *MyService* in the example benefits from the services of a logger component, but it depends only on an abstraction of it. The external logger component can be changed with ease and without risking breaking changes. Moreover, you can even change the component (for example, the strategy) on the fly. Try getting the same flexibility in a scenario where the implementation of the strategy object is hard-coded in the *MyService* class and you have to inherit a new class to change strategy. It's just impossible to change strategy in that case without recompilation and redeployment.

Architectural Patterns

Architectural patterns capture key elements of software architecture and offer support for making hard-to-change decisions about the structure of the system. As we saw in Chapter 1, software architecture is mostly about decisions regarding design points that, unlike code design, are not subject to refactoring.

Architectural patterns are selected and applied very early in the course of design, and they influence various quality characteristics of the system, such as performance, security, maintenance, and extensibility.

Examples of architectural patterns are Layers and SOA for modeling the application structure, Model-View-Controller for the presentation, Domain Model and Service Layer for the business logic, and Peer-to-Peer for the network topology.

Antipatterns

In physics, we have matter and antimatter. Just as matter is made of particles, antimatter is made of antiparticles. An antiparticle is identical to a particle except for the charge—positive in the particles of normal matter, and negative in an element of antimatter.

Likewise, in software we have patterns made of solutions, and antipatterns made of antisolutions. What's the difference? It is all in the "charge" of the solution. Patterns drive us to good solutions, whereas antipatterns drive us to bad solutions. The clearest definition for antipatterns we could find comes (again) from Ward Cunningham's wiki, at *http://c2.com/cgi/wiki?AntiPattern*:

> *An anti-pattern is a pattern that tells how to go from a problem to a bad solution.*

Put this way, one could reasonably wonder why antipatterns are worth the effort of defining them. For matter and antimatter, it's all about the thirst for knowledge. But developers and architects are usually more pragmatic and they tend to prefer knowledge with a practical application to their everyday work. What's the link that relates antipatterns to the real-world of software development?

The keystone of antipatterns is that they might, at first, look like good ideas that can add new power and effectiveness to your classes. An antipattern, though, is devious and insidious and adds more trouble than it removes. From Cunningham's wiki again:

> *In the old days, we used to just call these* bad ideas. *The new name is much more diplomatic.*

Designers and antipatterns, in some way, attract each other, but the experienced designer recognizes and avoids antipatterns. (This is definitely a characteristic that marks the difference between expert and nonexpert designers.) Because of the fatal attraction designers generally have toward antipatterns, a catalog of antipatterns is as valuable as a catalog of good design patterns.

A long list of antipatterns can be found at *http://c2.com/cgi/wiki?AntiPatternsCatalog* and also at *http://en.wikipedia.org/wiki/anti-pattern*. We like to briefly address a couple of them—one relates to architecture and the other relates to development.

The *Architecture-As-Requirements* antipattern refers to situations where a prominent and influential member of the design team has a pet technology or product and absolutely wants to use it in the project—even when there is no clear evidence of its usefulness and applicability in the customer's context.

The *Test-By-Release* antipattern refers to releasing a software product without paying much attention to all those boring and time-consuming chores related to unit and integration testing. Are users the final recipients of the product? Great, let's give them the last word on whether the software works or not.

Patterns vs. Idioms

Software patterns indicate well-established solutions to recurring design problems. This means that developers end up coding their way to a given solution over and over again. And they might be repeatedly writing the same boilerplate code in a given programming language.

Sometimes specific features of a given programming language can help significantly in quickly and elegantly solving a recurring problem. That specific set of features is referred to as an *idiom*.

What's an Idiom, Anyway?

An *idiom* is a pattern hard-coded in a programming language or implemented out of the box in a framework or technology.

Like a design pattern, an idiom represents a solution to a recurring problem. However, in the case of idioms, the solution to the problem doesn't come through design techniques but merely by using the features of the programming language. Whereas a design pattern focuses on the object-oriented paradigm, an idiom focuses on the technology of the programming language.

An idiom is a way to take advantage of the language capabilities and obtain a desired behavior from the code. In general, an idiom refers to a very specific, common, and eye-catching piece of code that accomplishes a given operation—as simple as adding to a counter or as complex as the implementation of a design pattern.

In C#, for example, the ++ operator can be considered a programming idiom for the recurring task of *adding to a counter variable*. The same can be said for the *as* keyword when it comes to *casting to a type and defaulting to null in case of failure*.

Let's see some more examples of programming idioms in C#.

Sample Idioms

Events are the canonical example of a programming idiom. Behind events, you find the Observer pattern. The pattern refers to a class that has the ability to notify registered observers of some internal states. Whenever a particular state is reached, the class loops through the list of registered observers and notifies each observer of the event. It does that using a contracted observer interface.

In languages such as C# or Visual Basic .NET that support event-driven programming, you find this pattern natively implemented and exposed through keywords. Consider the following code:

```
Button1.Click += new EventHandler(Button1_Click);
```

When it runs, a new "observer for the *Click* event" is added to the list maintained by object *Button1*. The observer in this case is a *delegate*—a special class wrapping a class method.

The interface through which observer and object communicate is the signature of the method wrapped by the delegate.

Similarly, the *foreach* keyword in C# (and *For*... *Each* in Visual Basic .NET) is a hard-coded version of the Iterator pattern. An iterator object accomplishes two main tasks: it retrieves a particular element within a collection and jumps to the next element. This is exactly what happens under the hood of the following code:

```
foreach(Customer customer in dataContext.Customers)
{
    // The variable customer references the current element in the collection.
    // Moving to the next element is implicit.
}
```

Finally, the most recent versions of C# and Visual Basic .NET—those shipping with the .NET Framework 3.5—also support a set of contextual keywords for Language Integrated Query (LINQ): *from*, *select*, *in*, *orderby*. When you apply the set of LINQ keywords to a database-oriented object model, you have LINQ-to-SQL. With LINQ-to-SQL, you ultimately use language keywords to query the content of a database. In other words, you programmatically define an object that represents a query and run it. This behavior is described by the Query Object pattern. And LINQ-to-SQL is a programming idiom for the pattern.

Idiomatic Design

We spent a lot of time pondering OOD principles and showing their benefits and applicability. We did it by reasoning in a general context and looking at the OO paradigm rather than by examining the concrete technology and platform. General principles are always valid and should always be given due consideration.

However, when you step inside the design, at some point you meet the technology. When this happens, you might need to review the way you apply principles in the context of the specific technology or platform you're using. This is called *idiomatic design*.

As far as the .NET Framework is concerned, a set of idiomatic design rules exists under the name of Framework Design Guidelines. You can access them online from the following URL: *http://msdn.microsoft.com/en-us/library/ms229042.aspx*.

Note *Framework Design Guidelines* is also the title of a book written by Krzysztof Cwalina and Brad Abrams from Microsoft (Addison-Wesley, 2008). Cwalina's blog is also an excellent source for tidbits and more details on guidelines. We definitely recommend it. The blog is *http://blogs.msdn.com/kcwalina*.

As an example, let's go through a couple of these guidelines.

Idiomatic Design: Structures or Classes?

When defining a type in a C# .NET application, should you use *struct* or *class*? To start out, a *struct* is not inheritable. So if you need to derive new classes from the type, you must opt for a class rather than a structure. This said, a class is a reference type and is allocated on the heap. Memorywise, a reference type is managed by the garbage collector. Conversely, a *struct* is a value type; it is allocated on the stack and deallocated when it goes out of scope. Value types are generally less expensive than reference types to work with, but not when boxing is required. In the .NET Framework, *boxing* is the task of storing a value type in an object reference so that it can be used wherever an object is accepted. As an example, consider the *ArrayList* class. When you add, say, an *Int32* (or a *struct*) to an *ArrayList*, the value is automatically boxed to an object. Done all the time, this extra work might change the balance between class and *struct*. Hence, the need of an official guideline on the theme shows up.

The guideline suggests that you always use a class unless the footprint of the type is below 16 bytes and the type is immutable. A type is immutable if the state of its instances never changes after they've been created. (The *System.String* type in the .NET Framework is immutable because a new string is created after each modification.) However, if the *struct* is going to be boxed frequently you might want to consider using a class anyway. (If you're looking for the list of differences between *struct*s and classes go here: *http://msdn.microsoft.com/en-us/library/saxz13w4.aspx*.)

Idiomatic Design: Do Not Use *List<T>* in Public Signatures

Another guideline we want to point out has to do with the *List<T>* type. Their use in the signature of public members is not recommended, as you can see in this blog post: *http://blogs.gotdotnet.com/kcwalina/archive/2005/09/26/474010.aspx*.

Why is this so?

One of the reasons behind the guideline is that *List<T>* is a rather bloated type with many members that are not relevant in many scenarios. This means that *List<T>* has low cohesion and to some extent violates the Single Responsibility Principle.

Another reason for not using *List<T>* in public signatures is that the class is unsealed, yes, but not specifically designed to be extended. This doesn't mean, though, that the class is not LSP-safe. If you look at the source of the class, you can see that using *List<T>* is absolutely safe in any polymorphic context. The issue is that the class has no protected and virtual methods for inheritors to do something significant that alters the behavior of the class while preserving the core interface. The class is just not designed to be extended.

It is therefore recommended that you use *IList<T>*, or derived interfaces, in public signatures. Alternatively, use custom classes that directly implement *IList<T>*.

Dependency Injection

As a design principle, DIP states that higher level modules should depend on abstractions rather than on the concrete implementation of functionalities. *Inversion of control* (IoC) is an application of DIP that refers to situations where generic code controls the execution of more specific and external components.

In an IoC solution, you typically have a method whose code is filled with one or more stubs. The functionality of each stub is provided (statically or dynamically) by external components invoked through an abstract interface. Replacing any external components doesn't affect the high-level method, as long as LSP and OCP are fulfilled. External components and the high-level method can be developed independently.

A real-world example of IoC is Windows shell extensions. Whenever the user right-clicks and selects Properties, Windows Explorer prepares a standard dialog box and then does a bit of IoC. It looks up the registry and finds out whether custom property page extensions have been registered. If any are registered, it talks to these extensions through a contracted interface and adds pages to the user dialog box.

Another real-world example of IoC is event-driven programming as originally offered by Visual Basic and now supported by Windows Forms and Web Forms. By writing a *Button1_Click* method and attaching it to the *Click* event of, say, the *Button1* control, you essentially instruct the (reusable and generic) code of the *Button* class to call back your *Button1_Click* method any time the user clicks.

What is dependency injection (DI), then?

From DIP to Inversion of Control

For the purpose of this discussion, IoC and DI are synonyms. They are not always considered synonyms in literature, as sometimes you find IoC to be the principle and DI the application of the principle—namely, the pattern. In reality, IoC is historically a pattern based on DIP. The term *dependency injection* was coined by Martin Fowler later, as a way to further specialize the concept of inversion of control.

IoC/DI remains essentially a pattern that works by letting you pass high-level method references to helper components. This injection can happen in three ways. One way is via the constructor of the class to which the method belongs. We did just this in the implementation of the *FinanceInfoService* class. Another way consists of defining a method or a setter property on the class to which the method belongs. Finally, the class can implement an interface whose methods offer concrete implementations of the helper components to use.

Today, IoC/DI is often associated with special frameworks that offer a number of rather advanced features.

IoC Frameworks

Table 3-1 lists some of the most popular IoC frameworks available.

TABLE 3-1 Main IoC Frameworks

Framework	More Information
Castle Windsor	*http://www.castleproject.org/container/index.html*
Ninject	*http://www.ninject.org*
Spring.NET	*http://www.springframework.net*
StructureMap	*http://structuremap.sourceforge.net/Default.htm*
Unity	*http://codeplex.com/unity*

Note that Ninject is also available for Silverlight and the Compact Framework. In particular, Microsoft's Unity Application Block (*Unity* for short) is a lightweight IoC container with support for constructor, property, and method call injection. It comes as part of the Enterprise Library 4.0. Let's use that for our demos.

All IoC frameworks are built around a container object that, bound to some configuration information, resolves dependencies. The caller code instantiates the container and passes the desired interface as an argument. In response, the IoC/DI framework returns a concrete object that implements that interface.

IoC Containers in Action

Suppose you have a class that depends on a logger service, such as the class shown here:

```
public class Task
{
  ILogger _logger;
  public Task(ILogger logger)
  {
    this._logger = logger;
  }
  public void Execute()
  {
    this._logger.Log("Begin method ...");
    ⋮
    this._logger.Log("End method ...");
  }
}
```

The *Task* class receives the logger component via the constructor, but how does it locate and instantiate the logger service? A simple and static *new* statement certainly works, and so does a factory. An IoC container is a much richer framework that supports a configuration section:

```
<configuration>
  <configSections>
    <section name="unity"
             type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
                   Microsoft.Practices.Unity.Configuration" />
  </configSections>
  ⋮
```

```
  <unity>
    <containers>
       <container>
          <types>
             <type type="ILogger, mdUtils"
                   mapTo="ManagedDesign.Tools.DbLogger, mdTools" />
          </types>
       </container>
    </containers>
  </unity>
</configuration>
```

The configuration file (*app.config* or *web.config*) contains mapping between interfaces and concrete types to be injected. Whenever the container gets a call for *ILogger*, it'll return an instance of *DbLogger*:

```
IUnityContainer container = new UnityContainer();
UnityConfigurationSection section = (UnityConfigurationSection)
                                    ConfigurationManager.GetSection("unity");
section.Containers.Default.Configure(container);
ILogger logger = container.Resolve<ILogger>();
Task t = new Task(logger);
⋮
```

IoC/DI is extremely useful for testing purposes and for switching between implementations of internal components. Frameworks just make it simple and terrific. In Chapter 6, "The Data Access Layer," we'll return to IoC/DI to show how to inject a data access layer (DAL) in the middle tier of a layered system.

To finish, here are a couple of brief remarks about IoC/DI containers. Through the configuration script, you can instruct the container to treat injected objects as singletons. This means, for example, that the container won't create a new instance of *DbLogger* every time, but will reuse the same one. If the *DbLogger* class is thread safe, this is really a performance boost.

In addition, imagine that the constructor of *DbLogger* needs a reference to another type registered with the IoC/DI framework. The container will be able to resolve that dependency, too.

Applying Requirements by Design

In Chapter 1, we saw that international standard ISO/IEC 9126 lists testability and security as key quality characteristics for any software architecture. This means that we should consider testability and security as nonfunctional requirements in any software architecture and start planning for them very early in the design phase.

Testability

A broadly accepted definition for testability in the context of software architecture describes it as the ease of performing testing. And testing is the process of checking software to ensure that it behaves as expected, contains no errors, and satisfies its requirements.

A popular slogan to address the importance of software testing comes from Bruce Eckel and reads like this:

> *If it ain't tested, it's broken.*

The key thing to keep in mind is that you can state that your code works only if you can provide evidence for that it does. A piece of software can switch to the status of *working* not when someone states it works (whether stated by end users, the project manager, the customer, or the chief architect), but only when its correctness is proven beyond any reasonable doubt.

Software Testing

Testing happens at various levels. You have *unit tests* to determine whether individual components of the software meet functional requirements. You have *integration tests* to determine whether the software fits in the environment and infrastructure and whether two or more components work well together. Finally, you have *acceptance tests* to determine whether the completed system meets customer requirements.

Unit tests and integration tests pertain to the development team and serve the purpose of making the team confident about the quality of the software. Test results tell the team if the team is doing well and is on the right track. Typically, these tests don't cover the entire code base. In general, there's no clear correlation between the percentage of code coverage and quality of code. Likewise, there's also no agreement on what would be a valid percentage of code coverage to address. Some figure that 80 percent is a good number. Some do not even instruct the testing tool to calculate it.

The customer is typically not interested in the results of unit and integration tests. Acceptance tests, on the other hand, are all the customer cares about. Acceptance tests address the completed system and are part of the contract between the customer and the development team. Acceptance tests can be written by the customer itself or by the team in strict collaboration with the customer. In an acceptance test, you can find a checklist such as the following one:

```
1)  Insert a customer with the following data ...;

2)  Modify the customer using an existing ID;

3)  Observe the reaction of the system and verify specific expected results;
```

Another example is the following:

```
1)  During a batch, shut down one nodes on the application server;

2)  Observe the reaction of the system and the results of the transaction;
```

Run prior to delivery, acceptance tests, if successful, signal the termination of the project and the approval of the product. (As a consultant, you can issue your final invoice at this point.)

Tests are a serious matter.

Testing the system by having end users poke around the software for a few days is not a reliable (and exhaustive) test practice. As we saw earlier in the chapter, it is even considered to be an antipattern.

Note Admittedly, in the early 1990s Dino delivered a photographic Windows application using the *test-by-poking-around* approach. We were a very small company with five developers, plus the boss. Our (patented?) approach to testing is described in the following paragraph.

The boss brings a copy of the program home. The boss spends the night playing with the program. Around 8 a.m. the next day, the team gets a call from the boss, who is going to get a few hours of very well-deserved sleep. The boss recites a long list of serious bugs to be fixed instantly and makes obscure references to alleged features of the program, which are unknown to the entire team. Early in the afternoon, the boss shows up at work and discusses improvements in a much more relaxed state of mind. The list of serious bugs to be fixed instantly morphs into a short list of new features to add.

In this way, however, we delivered the application and we could say we delivered a reliable and fully functioning piece of software. It was the 1994, though. The old days.

Software Contracts

A software test verifies that a component returns the correct output in response to given input and a given internal state. Having control over the input and the state and being able to observe the output is therefore essential.

Your testing efforts greatly benefit from detailed knowledge of the software contract supported by a method. When you design a class, you should always be sure you can answer the following three questions about the class and its methods in particular:

- Under which conditions can the method be invoked?
- Which conditions are verified after the method terminates?
- Which conditions do not change before and after the method execution?

These three questions are also known, respectively, as preconditions, postconditions, and invariants.

Preconditions mainly refer to the input data you pass; specifically, data that is of given types and values falling within a given range. Preconditions also refer to the state of the object required for execution—for example, the method that might need to throw an exception if an internal member is null or if certain conditions are not met.

When you design a method with testability in mind, you pay attention to and validate input carefully and throw exceptions if any preconditions are not met. This gets you clean code, and more importantly, code that is easier to test.

Postconditions refer to the output generated by the method and the changes produced to the state of the object. Postconditions are not directly related to the exceptions that might be thrown along the way. This is not relevant from a testing perspective. When you do testing, in fact, you execute the method if preconditions are met (and if no exceptions are raised because of failed preconditions). The method might produce the wrong results, but it should not fail unless really exceptional situations are encountered. If your code needs to read a file, that the file exists is a precondition and you should throw a *FileNotFoundException* before attempting to read. A *FileIOException*, say, is acceptable only if during the test you lose connection to the file.

There might be a case where the method delegates some work to an internal component, which might also throw exceptions. However, for the purpose of testing, that component will be replaced with a fake one that is guaranteed to return valid data by contract. (You are testing the outermost method now; you have tested the internal component already or you'll test it later.) So, in the end, when you design for testability the exceptions you should care about most are those in the preconditions.

Invariants refer to property values, or expressions involving members of the object's state, that do not change during the method execution. In a design for testability scenario, you know these invariants clearly and you assert them in tests. As an example of an invariant, consider the property *Status* of *DbConnection*: it has to be *Open* before you invoke *BeginTransaction*, and it must remain *Open* afterward.

Software contracts play a key role in the design of classes for testability. Having a contract clearly defined for each class you write makes your code inherently more testable.

Unit Testing

Unit testing verifies that individual units of code are working properly according to their software contract. A *unit* is the smallest part of an application that is testable—typically, a method.

Unit testing consists of writing and running a small program (referred to as a *test harness*) that instantiates classes and invokes methods in an automatic way. In the end, running a battery of tests is much like compiling. You click a button, you run the test harness and, at the end of it, you know what went wrong, if anything.

In its simplest form, a test harness is a manually written program that reads test-case input values and the corresponding expected results from some external files. Then the test harness calls methods using input values and compares results with expected values. Needless to say, writing such a test harness entirely from scratch is, at the very minimum, time consuming and error prone. But, more importantly, it is restrictive in terms of the testing capabilities you can take advantage of.

At the end of the day, the most effective way to conduct unit testing passes through the use of an automated test framework. An automated test framework is a developer tool that normally includes a runtime engine and a framework of classes for simplifying the creation of test programs. Table 3-2 lists some of the most popular ones.

TABLE 3-2 Popular Testing Tools

Product	Description
MSTest	The testing tool incorporated into Visual Studio 2008 Professional, Team Tester, and Team Developer. It is also included in Visual Studio 2005 Team Tester and Team Developer.
MBUnit	An open-source product with a fuller bag of features than MSTest. However, the tight integration that MSTest has with Visual Studio and Team Foundation Server largely makes up for the smaller feature set. For more information on MBUnit, pay a visit to *http://www.mbunit.com*.
NUnit	One of the most widely used testing tools for the .NET Framework. It is an open-source product. Read more at *http://www.nunit.org*.
xUnit.NET	Currently under development as a CodePlex project, this tool builds on the experience of James Newkirk—the original author of NUnit. It is definitely an interesting tool to look at, with some interesting and innovative features. For more information, pay a visit to *http://www.codeplex.com/xunit*.

A nice comparison of testing tools, in terms of their respective feature matrix, is available at *http://www.codeplex.com/xunit/Wiki/View.aspx?title=Comparisons*.

Unit Testing in Action

Let's have a look at some tests written using the MSTest tool that comes with Visual Studio 2008. You start by grouping related tests in a *text fixture*. Text fixtures are just test-specific classes where methods typically represent tests to run. In a text fixture, you might also have code that executes at the start and end of the test run. Here's the skeleton of a text fixture with MSTest:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
⋮

[TestClass]
public class CustomerTestCase
{
  private Customer customer;

  [TestInitialize]
  public void SetUp()
  {
    customer = new Customer();
  }

  [TestCleanup]
```

```
  public void TearDown()
  {
    customer = null;
  }

  // Your tests go here
  [TestMethod]
  public void Assign_ID()
  {
    ⋮
    ⋮
  }
  ⋮
  ⋮
}
```

It is recommended that you create a separate assembly for your tests and, more importantly, that you have tests for each class library. A good practice is to have an *XxxTestCase* class for each *Xxx* class in a given assembly.

As you can see, you transform a plain .NET class into a test fixture by simply adding the *TestClass* attribute. You turn a method of this class into a test method by using the *TestMethod* attribute instead. Attributes such as *TestInitialize* and *TestCleanup* have a special meaning and indicate code to execute at the start and end of the test run. Let's examine an initial test:

```
[TestMethod]
public void Assign_ID()
{
  // Define the input data for the test
  string id = "MANDS";

  // Execute the action to test (assign a given value)
  customer.ID = id;

  // Test the postconditions:
  // Ensure that the new value of property ID matches the assigned value.
  Assert.AreEqual(id, customer.ID);
}
```

The test simply verifies that a value is correctly assigned to the *ID* property of the *Customer* class. You use methods of the *Assert* object to assert conditions that must be true when checked.

The body of a test method contains plain code that works on properties and methods of a class. Here's another example that invokes a method on the *Customer* class:

```
[TestMethod]
public void TestEmptyCustomersHaveNoOrders()
{
  Customer c = new Customer();
  Assert.AreEqual<decimal>(0, c.GetTotalAmountOfOrders());
}
```

In this case, the purpose of the test is to ensure that a newly created *Customer* instance has no associated orders and the total amount of orders add up to zero.

Dealing with Dependencies

When you test a method, you want to focus only on the code within *that* method. All that you want to know is whether *that* code provides the expected results in the tested scenarios. To get this, you need to get rid of all dependencies the method might have. If the method, say, invokes another class, you assume that the invoked class will *always* return correct results. In this way, you eliminate at the root the risk that the method fails under test because a failure occurred down the call stack. If you test method A and it fails, the reason has to be found *exclusively* in the source code of method A—given preconditions, invariants, and behavior—and not in any of its dependencies.

Generally, the class being tested must be *isolated* from its dependencies.

In an object-oriented scenario, class A depends on class B when any of the following conditions are verified:

- Class A derives from class B.
- Class A includes a member of class B.
- One of the methods of class A invokes a method of class B.
- One of the methods of class A receives or returns a parameter of class B.
- Class A depends on a class that, in turn, depends on class B.

How can you neutralize dependencies when testing a method? This is exactly where manually written test harnesses no longer live up to your expectations, and you see the full power of automated testing frameworks.

Dependency injection really comes in handy here and is a pattern that has a huge impact on testability. A class that depends on interfaces (the first principle of OOD), and uses dependency injection to receive from the outside world any objects it needs to do its own work, is inherently more testable. Let's consider the following code snippet:

```
public class Task
{
  // Class Task depends upon type ILogger
  ILogger _logger;

  public Task(ILogger logger)
  {
    this._logger = logger;
  }

  public int Sum(int x, int y)
  {
    return x+y;
  }
```

```
  public void Execute()
  {
    // Invoke an external "service"; not relevant when unit-testing this method
    this._logger.Log("Begin method ...");

    // Method specific code; RELEVANT when unit-testing this method
    ⋮

    // Invoke an external "service"; not relevant when unit-testing this method
    this._logger.Log("End method ...");
  }
}
```

We want to test the code in method *Execute*, but we don't care about the logger. Because the class *Task* is designed with DI in mind, testing the method *Execute* in total isolation is much easier.

Again, how can you neutralize dependencies when testing a method?

The simplest option is using *fake* objects. A fake object is a relatively simple clone of an object that offers the same interface as the original object but returns hard-coded or programmatically determined values. Here's a sample fake object for the *ILogger* type:

```
public class FakeLogger : ILogger
{
    public void Log(string message)
    {
        return;
    }
}
```

As you can see, the behavior of a fake object is hard-coded; the fake object has no state and no significant behavior. From the fake object's perspective, it makes no difference how many times you invoke a fake method and when in the flow the call occurs. Let's see how to inject a fake logger in the *Task* class:

```
[TestMethod]
public void TestIfExecuteWorks()
{
  // Inject a fake logger to isolate the method from dependencies
  FakeLogger fake = new FakeLogger();
  Task task = new Task(fake);

  // Set preconditions
  int x = 3;
  int y = 4;
  int expected = 7;

  // Run the method
  int actual = task.Sum(x, y);

  // Report about the code's behavior using Assert statements
  Assert.AreEqual<int>(expected, actual);
  ⋮
}
```

In a test, you set the preconditions for the method, run the method, and then observe the resulting postconditions. The concept of *assertion* is central to the unit test. An assertion is a condition that might or might not be verified. If verified, the assertion passes. In MSTest, the *Assert* class provides many static methods for making assertions, such as *AreEqual*, *IsInstanceOfType*, and *IsNull*.

In the preceding example, after executing the method *Sum* you are expected to place one or more assertions aimed at verifying the changes made to the state of the object or comparing the results produced against expected values.

Note In some papers, terms such as *stub* and *shunt* are used to indicate slight variations of what we reference here as a *fake*. A broadly accepted differentiation is based on the fact that a stub (or a shunt) merely provides the implementation of an interface. Methods can just throw or, at most, return canned values.

A fake, on the other hand, is a slightly more sophisticated object that, in addition to implementing an interface, also usually contains more logic in the methods. Methods on a fake object can return canned values but also programmatically set values. Both fakes and stubs can provide a meaningful implementation for some methods and just throw exceptions for other methods that are not considered relevant for the purpose of the test.

A bigger and juicier differentiation, however, is the one that exists between fakes (or stubs) and mock objects, which is discussed next.

From Fakes to Mocks

A *mock* object is a more evolved and recent version of a fake. A mock does all that a fake or a stub does, plus something more. In a way, a mock is an object with its own personality that mimics the behavior and interface of another object. What more does a mock provide to testers?

Essentially, a mock allows for verification of the context of the method call. With a mock, you can verify that a method call happens with the right preconditions and in the correct order with respect to other methods in the class.

Writing a fake manually is not usually a big issue—all the logic you need is for the most part simple and doesn't need to change frequently. When you use fakes, you're mostly interested in verifying that some expected output derives from a given input. You are interested in the state that a fake object might represent; you are not interested in interacting with it.

You use a mock instead of a fake only when you need to interact with dependent objects during tests. For example, you might want to know whether the mock has been invoked or not, and you might decide within the text what the mock object has to return for a given method.

Writing mocks manually is certainly a possibility, but is rarely an option you really want to consider. For the level of flexibility you expect from a mock, you should be updating its source code every now and then or you should have (and maintain) a different mock for each

test case in which the object is being involved. Alternatively, you might come up with a very generic mock class that works in the guise of any object you specify. This very generic mock class also exposes a general-purpose interface through which you set your expectations for the mocked object. This is exactly what mocking frameworks do for you. In the end, you never write mock objects manually; you generate them on the fly using some mocking framework.

Table 3-3 lists and briefly describes the commonly used mocking frameworks.

TABLE 3-3 Some Popular Mocking Frameworks

Product	Description
NMock2	An open-source library providing a dynamic mocking framework for .NET interfaces. The mock object uses strings to get input and reflection to set expectations. Read more at *http://sourceforge.net/projects/nmock2*.
TypeMock	A commercial product with unique capabilities that basically don't require you to (re)design your code for testability. TypeMock enables testing code that was previously considered untestable, such as static methods, nonvirtual methods, and sealed classes. Read more at *http://www.typemock.com*.
Rhino Mocks	An open-source product. Through a wizard, it generates a static mock class for type-safe testing. You set mock expectations by accessing directly the mocked object, rather than going through one more level of indirection. Read more at *http://www.ayende.com/projects/rhino-mocks.aspx*.

Let's go through a mocking example that uses NMock2 in MSTest.

Imagine you have an *AccountService* class that depends on the *ICurrencyService* type. The *AccountService* class represents a bank account with its own currency. When you transfer funds between accounts, you might need to deal with conversion rates, and you use the *ICurrencyService* type for that:

```
public interface ICurrencyService
{
  // Returns the current conversion rate: how many "fromCurrency" to
  // be changed into toCurrency
  decimal GetConversionRate(string fromCurrency, string toCurrency);
}
```

Let's see what testing the *TransferFunds* method looks like:

```
[TestClass]
public class CurrencyServiceTestCase
{
  private Mockery mocks;
  private ICurrencyService mockCurrencyService;
  private IAccountService accountService;
```

```
  [TestInitialize]
  public void SetUp()
  {
    // Initialize the mocking framework
    mocks = new Mockery();

    // Generate a mock for the ICurrencyService type
    mockCurrencyService = mocks.NewMock<ICurrencyService>();

    // Create the object to test and inject the mocked service
    accountService = new AccountService(mockCurrencyService);
  }

  [TestMethod]
  public void TestCrossCurrencyFundsTransfer()
  {
    // Create two test accounts
    Account eurAccount = new Account("12345", "EUR");
    Account usdAccount = new Account("54321", "USD");
    usdAccount.Deposit(1000);

    // Set expectations for the mocked object:
    //   When method GetConversionRate is invoked with (USD,EUR) input
    //   the mock returns 0.64
    Expect.Once.On(mockCurrencyService)
                .Method("GetConversionRate")
                .With("USD", "EUR")
                .Will(Return.Value(0.64));

    // Invoke the method to test (and transfer $500 to an EUR account)
    accountService.TransferFunds(usdAccount, eurAccount, 500);

    // Verify postconditions through assertions
    Assert.AreEqual<int>(500, usdAccount.Balance);
    Assert.AreEqual<int>(320, eurAccount.Balance);
    mocks.VerifyAllExpectationsHaveBeenMet();
  }
}
```

You first create a mock object for each dependent type. Next, you programmatically set expectations on the mock using the static class *Expect* from the NMock2 framework. In particular, in this case you establish that when the method *GetConversionRate* on the mocked type is invoked with a pair of arguments such as "USD" and "EUR", it has to return 0.64. This is just the value that the method *TransferFunds* receives when it attempts to invoke the currency services internally.

There's no code around that belongs to a mock object, and there's no need for developers to look into the implementation of mocks. Reading a test, therefore, couldn't be easier. The expectations are clearly declared and correctly passed on the methods under test.

Note A mock is generated on the fly using .NET reflection to inspect the type to mimic and the CodeDOM API to generate and compile code dynamically.

Security

Located at Carnegie Mellon University in Pittsburgh, Pennsylvania, the CERT Coordination Center (CERT/CC) analyzes the current state of Internet security. It regularly receives reports of vulnerabilities and researches the inner causes of security vulnerabilities. The center's purpose is to help with the development of secure coding practices.

Figure 3-4 shows a statistic about the number of identified vulnerabilities in the past ten years. As you can see, the trend is impressive. Also, you should consider that the data includes only the first two quarters of 2008. (See *http://www.cert.org/stats/vulnerability_remediation.html*.)

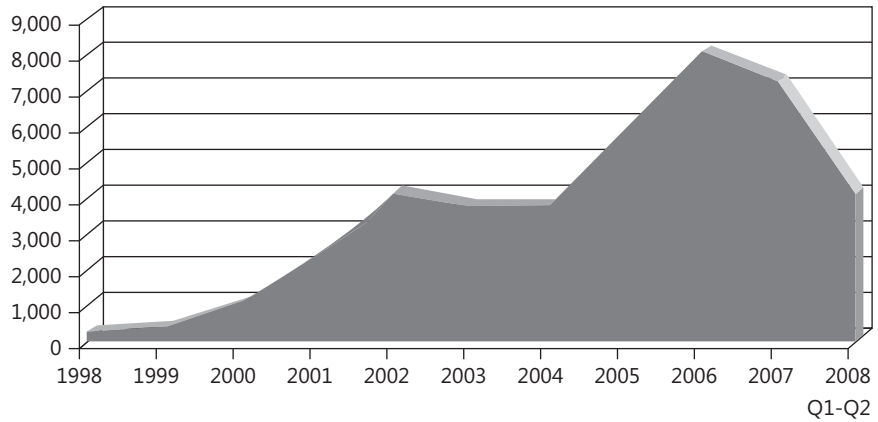


FIGURE 3-4 Identified security vulnerabilities in past ten years

It is broadly accepted that these numbers have a common root—they refer to software created through methodologies not specifically oriented to security. On the other hand, the problem of security is tightly related to the explosion in the popularity of the Internet. Only ten years ago, the big bubble was just a tiny balloon.

In sharp contrast with the ISO/IEC 9126 standard, all current methodologies for software development (agile, waterfall, MSF, and the like) hardly mention the word *security*. Additionally, the use of these methodologies has not resulted (yet?) in a measurable reduction of security bugs. To accomplish this, you need more than these methodologies offer.

Security as a (Strict) Requirement

We can't really say whether this is a real story or an urban legend, but it's being said that a few years ago, in the early days of the .NET Framework, a consultant went to some CIA office for a training gig. When introducing Code Access Security—the .NET Framework mechanism to limit access to code—the consultant asked students the following question: "Are you really serious about security here?"

Can you guess the answer? It was sort of like this: "Not only yes, but HELL YES. And you'll experience that yourself when you attempt to get out of the building."

Being serious about (software) security, though, is a subtle concept that goes far beyond even your best intentions. As Microsoft's senior security program manager Michael Howard points out:

> *If your engineers know nothing about the basic security tenets, common security defect types, basic secure design, or security testing, there really is no reasonable chance they could produce secure software. I say this because, on the average, software engineers don't pay enough attention to security. They may know quite a lot about security features, but they need to have a better understanding of what it takes to build and deliver secure features.*

Security must be taken care of from the beginning. A secure design starts with the architecture; it can't be something you bolt on at a later time. Security is by design. To address security properly, you need a methodology developed with security in mind that leads you to design your system with security in mind. This is just what the Security Development Lifecycle (SDL) is all about.

Security Development Lifecycle

SDL is a software development process that Microsoft uses internally to improve software security by reducing security bugs. SDL is not just an internal methodology. Based on the impressive results obtained internally, Microsoft is now pushing SDL out to any development team that wants to be really serious about security.

SDL is essentially an iterative process that focuses on security aspects of developing software. SDL doesn't mandate a particular software development process and doesn't preclude any. It is agnostic to the methodology in use in the project—be it waterfall, agile, spiral, or whatever else.

SDL is the incarnation of the SD^3+C principle, which is a shortcut for "Secure by Design, Secure by Default, Secure in Deployment, plus Communication." *Secure by Design* refers to identifying potential security risks starting with the design phase. *Secure by Default* refers to reducing the attack surface of each component and making it run with the least possible number of privileges. *Secure in Deployment* refers to making security requirements clear during deployment. *Communication* refers to sharing information about findings to apply a fix in a timely manner.

Foundations of SDL: Layering

The foundations of SDL are essentially three: layering, componentization, and roles.

Decomposing the architecture to layers is important because of the resulting separation of concerns. Having functionality organized in distinct layers makes it easier to map functions to physical tiers as appropriate. This is beneficial at various levels.

For example, it is beneficial for the data server.

You can isolate the data server at will, and even access it through a separate network. In this case, the data server is much less sensitive to denial of service (DoS) attacks because of the firewalls scattered along the way that can recognize and neutralize DoS packets.

You move all security checks to the business layer running on the application server and end up with a single user for the database—the data layer. Among other things, this results in a bit less work for the database and a pinch of additional scalability for the system.

Layers are beneficial for the application server, too.

You use Code Access Security (CAS) on the business components to stop untrusted code from executing privileged actions. You use CAS imperatively through *xxxPermission* classes to decide what to do based on actual permissions. You use CAS declaratively on classes or assemblies through *xxxPermission* attributes to prevent unauthorized use of sensitive components. If you have services, the contract helps to delimit what gets in and what gets out of the service.

Finally, if layering is coupled with thin clients, you have fewer upgrades (which are always a risk for the stability of the application) and less logic running on the client. Securitywise, this means that a possible dump of the client process would reveal much less information, so being able to use the client application in partial trust mode is more likely.

Foundations of SDL: Componentization

Each layer is decomposed to components. Components are organized by functions *and* required security privileges. It should be noted that performance considerations might lead you to grouping or further factorizing components in successive iterations.

Componentization here means identifying the components to secure and not merely breaking down the logical architecture into a group of assemblies.

For each component, you define the public contract and get to know exactly what data is expected to come in and out of the component. The decomposition can be hierarchical. From a security point of view, at this stage you are interested *only* in components within a layer that provide a service. You are not interested, for example, in the object model (that is, the domain model, typed *DataSets*, custom DTOs) because it is shared by multiple layers and represents only data and behavior on the data.

For each component, you identify the least possible set of privileges that make it run. From a security perspective, this means that in case of a successful attack, attackers gain the minimum possible set of privileges.

Components going to different processes run in total isolation and each has its own access control list (ACL) and Windows privileges set. Other components, conversely, might require their own AppDomain within the same .NET process. An AppDomain is like a virtual

process within a .NET application that the Common Language Runtime (CLR) uses to isolate code within a secure boundary. (Note, however, that an AppDomain doesn't represent a security barrier for applications running in full-trust mode.) An AppDomain can be sandboxed to have a limited set of permissions that, for example, limit disk access, socket access, and the like.

Foundation of SDL: Roles

Every application has its own assets. In general, an asset is any data that attackers might aim at, including a component with high privileges. Users access assets through the routes specified by use cases. From a security perspective, you should associate use cases with categories of users authorized to manage related assets.

A *role* is just a logical attribute assigned to a user. A role refers to the logical role the user plays in the context of the application. In terms of configuration, each user can be assigned one or more roles. This information is attached to the .NET identity object, and the application code can check it before the execution of critical operations. For example, an application might define two roles—Admin and Guest, each representative of a set of application-specific permissions. Users belonging to the Admin role can perform tasks that other users are prohibited from performing.

Assigning roles to a user account doesn't add any security restrictions by itself. It is the responsibility of the application—typically, the business layer—to ensure that users perform only operations compatible with their role.

With roles, you employ a unique model for authorization, thus unifying heterogeneous security models such as LDAP, NTFS, database, and file system. Also, testing is easier. By impersonating a role, you can test access on any layer.

In a role-based security model, total risks related to the use of impersonation and delegation are mitigated. Impersonation allows a process to run using the security credentials of the impersonated user but, unlike delegation, it doesn't allow access to remote resources on behalf of the impersonated user. In both cases, the original caller's security context can be used to go through computer boundaries from the user interface to the middle tier and then all the way down to the database. This is a risk in a security model in which permissions are restricted by object. However, in a role-based security model, the ability to execute a method that accesses specific resources is determined by role membership, not credentials. User's credentials might not be sufficient to operate on the application and data server.

Authorization Manager (AzMan) is a separate Windows download that enables you to group individual operations together to form tasks. You can then authorize roles to perform specific tasks, individual operations, or both. AzMan offers a centralized console (an MMC snap-in) to define manager roles, operations, and users.

Note AzMan is a COM-based component that has very little to share with the .NET Framework. The .NET-based successor to AzMan is still in the works somewhere in Redmond. The community of developers expects something especially now that Microsoft has unveiled a new claims-based identity model that essentially factors authentication out of applications so that each request brings its own set of claims, including user name, e-mail address, user role, and even more specific information.

Threat Model

Layering, componentization, and roles presuppose that, as an architect, you know the assets (such as sensitive data, highly privileged components) you want to protect from attackers. It also presupposes that you understand the threats related to the system you're building and which vulnerabilities it might be exposed to after it is implemented. Design for security means that you develop a threat model, understand vulnerabilities, and do something to mitigate risks.

Ideally, you should not stop at designing this into your software, but look ahead to threats and vulnerabilities in the deployment environment and to those resulting from interaction with other products or systems. To this end, understanding the threats and developing a threat model is a must. For threats found at the design level, applying countermeasures is easy. Once the application has been developed, applying countermeasures is much harder. If an application is deployed, it's nearly impossible to apply internal countermeasures—you have to rely on external security practices and devices. Therefore, it's better to architect systems with built-in security features.

You can find an interesting primer on threat models at the following URL: *http://blogs.msdn.com/ptorr/archive/2005/02/22/GuerillaThreatModelling.aspx*.

Threat modeling essentially consists of examining components for different types of threats. STRIDE is a threat modeling practice that lists the following six types of threats:

- **Spoofing of user identity** Refers to using false identities to get into the system. This threat is mitigated by filtering out invalid IP addresses.
- **Tampering** Refers to intercepting/modifying data during a module's conversation. This threat is mitigated by protecting the communication channel (for example, SSL or IPSec).
- **Repudiation** Refers to the execution of operations that can't be traced back to the author. This threat is mitigated by strong auditing policies.
- **Information disclosure** Refers to unveiling private and sensitive information to unauthorized users. This threat is mitigated by enhanced authorization rules.

- **Denial of service** Refers to overloading a system up to the point of blocking it. This threat is mitigated by filtering out requests and frequently and carefully checking the use of the bandwidth.
- **Elevation of privilege** Refers to executing operations that require a higher privilege than the privilege currently assigned. This threat is mitigated by assigning the least possible privilege to any components.

If you're looking for more information on STRIDE, you can check out the following URL: *http://msdn.microsoft.com/en-us/magazine/cc163519.aspx*.

After you have the complete list of threats that might apply to your application, you prioritize them based on the risks you see associated with each threat. It is not realistic, in fact, that you address all threats you find. Security doesn't come for free, and you should balance costs with effectiveness. As a result, threats that you regard as unlikely or not particularly harmful can be given a lower priority or not covered at all.

How do you associate a risk with a threat? You use the DREAD model. It rates the risk as the probability of the attack multiplied by the impact it might have on the system. You should focus on the following aspects:

- **Discoverability** Refers to how high the likelihood is that an attacker discovers the vulnerability. It is a probability attribute.
- **Reproducibility** Refers to how easy it could be to replicate the attack. It is a probability attribute.
- **Exploitability** Refers to how easy it could be to perpetrate the attack. It is a probability attribute.
- **Affected users** Refers to the number of users affected by the attack. It is an impact attribute.
- **Damage potential** Refers to the quantity of damage the attack might produce. It is an impact attribute.

You typically use a simple High, Medium, or Low scale to determine the priority of the threats and decide which to address and when. If you're looking for more information on DREAD, you can check out the following URL: *http://msdn.microsoft.com/en-us/library/aa302419.aspx*.

Note STRIDE and DREAD is the classic analysis model pushed by the Security Development Lifecycle (SDL) team and is based on the attacker's viewpoint. It works great in an enterprise scenario, but it requires a security specialist because the resulting threat model is large and complex. Another, simplified, model is emerging—the CIA/PI model, which stands for Confidentiality Integrity Availability/Probability Impact. This model is simplified and focuses on the defender's point of view. An interesting post is this one: *http://blogs.msdn.com/threatmodeling/archive/2007/10/30/a-discussion-on-threat-modeling.aspx*.

Security and the Architect

An inherently secure design, a good threat model, and a precise analysis of the risk might mean very little if you then pair them with a weak and insecure implementation. As an architect, you should intervene at three levels: development, code review, and testing.

As far as development is concerned, the use of strong typing should be enforced because, by itself, it cuts off a good share of possible bugs. Likewise, knowledge of common security patterns (for example, the "all input is evil" pattern), application of a good idiomatic design, and static code analysis (for example, using FxCop) are all practices to apply regularly and rigorously.

Sessions of code review should be dedicated to a careful examination of the actual configuration and implementation of security through CAS, and to spot the portions of code prone to amplified attacks, such as cross-site scripting, SQL injection, overflows, and similar attack mechanisms.

Unit testing for security is also important if your system receives files and sequences of bytes. You might want to consider a technique known as *fuzzing*. Fuzzing is a software testing technique through which you pass random data to a component as input. The code might throw an appropriate exception or degrade gracefully. However, it might also crash or fail some expected assertions. This technique can reveal some otherwise hidden bugs.

Final Security Push

Although security should be planned for from the outset, you can hardly make some serious security tests until the feature set is complete and the product is close to its beta stage. It goes without saying that any anomalies found during security tests lead the team to reconsidering the design and implementation of the application, and even the threat model.

The final security push before shipping to the customer is a delicate examination and should preferably be delegated to someone outside the team, preferably some other independent figure.

Releasing to production doesn't mean the end of the security life cycle. As long as a system is up and running, it is exposed to possible attacks. You should always find time for penetration testing, which might lead to finding new vulnerabilities. So the team then starts the cycle again with the analysis of the design, implementation, and threat model. Over and over again, in an endless loop.

Performance Considerations

You might wonder why we're including a sidebar on performance rather than a full "Design for Performance" section. Performance is something that results from the actual behavior of the system, not something you can put in. If you're creating a standalone, small disconnected program, you can optimize it almost at will. It is radically different when we move up in scope to consider an enterprise-class system.

Performance is not something absolute.

What is performance? Is it the response time the end user perceives? Is it resource utilization that might or might not penalize the middle tier? Is it network latency or database I/O latency? Is it related to caching or smarter algorithms? Is it a matter of bad design? Is it merely horsepower?

Too often, a design decision involves a tradeoff between performance and scalability. You release some performance-oriented improvement to achieve better scalability—that is, a better (read, faster) response when the workload grows. Performance is never something absolute.

In an enterprise-class system, efficiency and performance are certainly requirements to take into account, but they are not fundamental requirements.

In our opinion, a bad design influences performance, but there's no special suggestion we can share to help you to come up with a high-performance design. The design is either good or bad; if it's good, it sets the groundwork for good performance.

As we've seen in this chapter, a good design is based on interfaces, has low coupling, and allows for injection of external functionalities. Done in this way, the design leaves a lot of room for replacing components with others that might provide a better performance.

As Donald Knuth used to say, "Premature optimization is the root of all evil." So optimizing is fine and necessary, but you should care about it only when you have evidence of poor performance. And only when you know what is doing poorly and that it can be improved. Optimization is timely—it is never premature, never late.

Performance is hardly something that works (or doesn't work) in theory. You can hardly say from a design or, worse yet, from a specification whether the resulting system will perform poorly or not. You build the system in the best and simplest way you can. You adhere to OOD principles and code your way to the fullest. Then you test the system.

If it works, but it doesn't work as fast as it should, you profile the system and figure out what can be improved—be it a stored procedure, an intermediate cache, or a dynamic proxy injection. If the design is flexible enough and leaves room for changes, you shouldn't have a hard time applying the necessary optimization.

From Objects to Aspects

No doubt that OOP is currently a mainstream programming paradigm. When you design a system, you decompose it into components and map the components to classes. Classes hold data and deliver a behavior. Classes can be reused and used in a polymorphic manner, although you must do so with the care we discussed earlier in the chapter.

Even with all of its undisputed positive qualities, though, OOP is not the perfect programming paradigm.

The OO paradigm excels when it comes to breaking a system down into components and describing processes through components. The OO paradigm also excels when you deal with the *concerns* of a component. However, the OO paradigm is not as effective when it comes to dealing with *cross-cutting concerns*.

A cross-cutting concern is a concern that affects multiple components in a system, such as logging, security, and exception handling. Not being a specific responsibility of a given component or family of components, a cross-cutting concern looks like an *aspect* of the system that must be dealt with at a different logical level, a level beyond application classes. Enter a new programming paradigm: *aspect-oriented programming* (AOP).

Aspect-Oriented Programming

The inherent limitations of the OO paradigm were identified quite a few years ago, not many years after the introduction of OOP. However, today AOP still is not widely implemented even though everybody agrees on the benefits it produces. The main reason for such a limited adoption is essentially the lack of proper tools. We are pretty sure the day that AOP is (even only partially) supported by the .NET platform will represent a watershed in the history of AOP.

The concept of AOP was developed at Xerox PARC laboratories in the 1990s. The team also developed the first (and still most popular) AOP language: AspectJ. Let's discover more about AOP by exploring its key concepts.

Note We owe to the Xerox PARC laboratories many software-related facilities we use every day. In addition to AOP (which we don't exactly use every day), Xerox PARC is "responsible" for laser printers, Ethernet, and mouse-driven graphical user interfaces. They always churned out great ideas, but failed sometimes to push their widespread adoption—look at AOP. The lesson that everybody should learn from this is that technical excellence is not necessarily the key to success, not even in software. Some good commercial and marketing skills are always (strictly) required.

Cross-Cutting Concerns

AOP is about separating the implementation of cross-cutting concerns from the implementation of core concerns. For example, AOP is about separating a logger class from a task class so that multiple task classes can use the same logger and in different ways.

We have seen that dependency injection techniques allow you to inject—and quite easily, indeed—external dependencies in a class. A cross-cutting concern (for example, logging) can certainly be seen as an external dependency. So where's the problem?

Dependency injection requires up-front design or refactoring, which is not always entirely possible in a large project or during the update of a legacy system.

In AOP, you wrap up a cross-cutting concern in a new component called an *aspect*. An aspect is a reusable component that encapsulates the behavior that multiple classes in your project require.

Processing Aspects

In a classic OOP scenario, your project is made of a number of source files, each implementing one or more classes, including those representing a cross-cutting concern such as logging. As shown in Figure 3-5, these classes are then processed by a compiler to produce executable code.

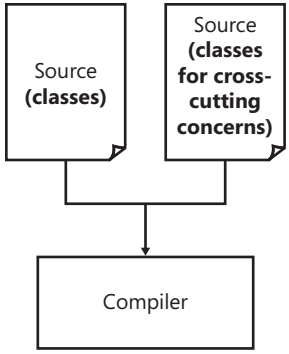


FIGURE 3-5 The classic OOP model of processing source code

In an AOP scenario, on the other hand, aspects are not directly processed by the compiler. Aspects are in some way merged into the regular source code up to the point of producing code that can be processed by the compiler. If you are inclined to employ AspectJ, you use the Java programming language to write your classes and the AspectJ language to write aspects. AspectJ supports a custom syntax through which you indicate the expected behavior for the aspect. For example, a logging aspect might specify that it will log before and after a certain method is invoked and will validate input data, throwing an exception in case of invalid data.

In other words, an aspect describes a piece of standard and reusable code that you might want to inject in existing classes without touching the source code of these classes. (See Figure 3-6.)

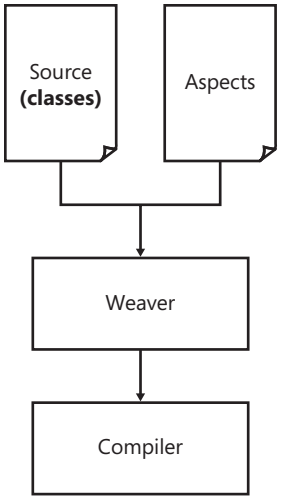


FIGURE 3-6 The AOP model of processing source code

In the AspectJ jargon, the *weaver* is a sort of preprocessor that takes aspects and weaves their content with classes. It produces output that the compiler can render to an executable.

In other AOP-like frameworks, you might not find an explicit weaver tool. However, in any case, the content of an aspect is always processed by the framework and results in some form of code injection. This is radically different from dependency injection. We mean that the code declared in an aspect will be invoked at some specific points in the body of classes that require that aspect.

Before we discuss an example in .NET, we need to introduce a few specific terms and clarify their intended meaning. These concepts and terms come from the original definition of AOP. We suggest that you do not try to map them literally to a specific AOP framework. We suggest, instead, that you try to understand the concepts—the pillars of AOP—and then use this knowledge to better and more quickly understand the details of a particular framework.

Inside AOP Aspects

As mentioned, an aspect is the implementation of a cross-cutting concern. In the definition of an aspect, you need to specify *advice* to apply at specific *join points*.

A *join point* represents a point in the class that requires the aspect. It can be the invocation of a method, the body of a method or the getter/setter of a property, or an exception handler. In general, a join point indicates the point where you want to inject the aspect's code.

A *pointcut* represents a collection of join points. In AspectJ, pointcuts are defined by criteria using method names and wildcards. A sample pointcut might indicate that you group all calls to methods whose name begins with *Get*.

An *advice* refers to the code to inject in the target class. The code can be injected before, after, and around the join point. An advice is associated with a pointcut.

Here's a quick example of an aspect defined using AspectJ:

```
public aspect MyAspect
{
  // Define a pointcut matched by all methods in the application whose name begins with
  // Get and accepting no arguments. (There are many other ways to define criteria.)
  public pointcut allGetMethods ():
         call (* Get*() );

  // Define an advice to run before any join points that matches the specified pointcut.
  before(): allGetMethods()
  {
    // Do your cross-cutting concern stuff here
    // for example, log about the method being executed
    ⋮
  }
}
```

The weaver processes the aspect along with the source code (regular class-based source code) and generates raw material for the compiler. The code actually compiled ensures that an advice is invoked automatically by the AOP runtime whenever the execution flow reaches a join point in the matching pointcut.

AOP in .NET

When we turn to AOP, we essentially want our existing code to do extra things. And we want to achieve that without modifying the source code. We need to specify such extra things (advice) and where we want to execute them (join points). Let's briefly go through these points from the perspective of the .NET Framework.

How can you express the semantic of aspects?

The ideal option is to create a custom language *a là* AspectJ. In this way, you can create an ad hoc aspect tailor-made to express advice at its configured pointcuts. If you have a custom language, though, you also need a tool to parse it—like a weaver.

A very cost-effective alternative is using an external file (for example, an XML file) where you write all the things you want to do and how to do it. An XML file is not ideal for defining source code; in such a file, you likely store mapping between types so that when a given type is assigned an aspect, another type is loaded that contains advice and instructions about how to join it to the execution flow. This is the approach taken by Microsoft's Policy Injection Application Block (PIAB) that we'll look at in a moment.

How can you inject an aspect's advice into executable code?

There are two ways to weave a .NET executable. You can do that at compile time or at run time. Compile-time weaving is preferable, but in our opinion, it requires a strong commitment from a vendor. It can be accomplished by writing a weaver tool that reads the content of the aspect, parses the source code of the language (C#, Visual Basic .NET, and all of the other languages based on the .NET common type system), and produces modified source code, but source code that can still be compiled. If you want to be language independent, write a weaver tool that works on MSIL and apply that past the compilation step. Alternatively, you can write a brand new compiler that understands an extended syntax with ad hoc AOP keywords.

If you want to weave a .NET executable at run time, you have to review all known techniques to inject code dynamically. One is emitting JIT classes through *Reflection.Emit*; another one is based on the CLR's Profiling API. The simplest of all is perhaps managing to have a proxy sitting in between the class's aspects and its caller. In this case, the caller transparently invokes a proxy for the class's aspects. The proxy, in turn, interweaves advice with regular code. This is the same mechanism used in .NET Remoting and Windows Communication Foundation (WCF) services.

Using a transparent proxy has the drawback of requiring that to apply AOP to the class, the class must derive from *ContextBoundObject* or *MarshalByRefObject*. This solution is employed by PIAB.

AOP in Action

To finish off our AOP overview, let's proceed with a full example that demonstrates how to achieve AOP benefits in .NET applications. We'll use Microsoft's Policy Injection Application Block in Enterprise Library 3.0 and higher to add aspects to our demo. For more information on PIAB, see *http://msdn.microsoft.com/en-us/library/cc511729.aspx*.

Enabling Policies

The following code demonstrates a simple console application that uses the Unity IoC container to obtain a reference to a class that exposes a given interface—*ICustomerServices*:

```
public interface ICustomerServices
{
    void Delete(string customerID);
}

static void Main(string[] args)
{
    // Set up the IoC container
    UnityConfigurationSection section;
    section = ConfigurationManager.GetSection("unity") as UnityConfigurationSection;
    IUnityContainer container = new UnityContainer();
    section.Containers.Default.Configure(container);
```

```
    // Resolve a reference to ICustomerServices. The actual class returned depends
    // on the content of the configuration section.
    ICustomerServices obj = container.Resolve<ICustomerServices>();

    // Enable policies on the object (for example, enable aspects)
    ICustomerServices svc = PolicyInjection.Wrap<ICustomerServices>(obj);

    // Invoke the object
    svc.Delete("ALFKI");

    // Wait until the user presses any key
    Console.ReadLine();
}
```

After you have resolved the dependency on the *ICustomerServices* interface, you pass the object to the PIAB layer so that it can wrap the object in a policy-enabled proxy. What PIAB refers to here as a *policy* is really like what many others call, instead, an aspect.

In the end, the *Wrap* static method wraps a given object in a proxy that is driven by the content of a new section in the configuration file. The section *policyInjection* defines the semantics of the aspect. Let's have a look at the configuration file.

Defining Policies

PIAB is driven by the content of an ad hoc configuration section. There you find listed the policies that drive the behavior of generated proxies and that ultimately define aspects to be applied to the object within the proxy.

```
<policyInjection>
  <policies>
    <add name="Policy">
      <matchingRules>
        <add type="EnterpriseLibrary.PolicyInjection.MatchingRules.TypeMatchingRule ..."
             name="Type Matching Rule">
          <matches>
             <add match="ArchNet.Services.ICustomerServices" ignoreCase="false" />
          </matches>
        </add>
      </matchingRules>
      <handlers>
        <add order="0"
             type="ManagedDesign.Tools.DbLogger, mdTools"
             name="Logging Aspect" />
      </handlers>
    </add>
  </policies>
</policyInjection>
```

The *matchingRules* section expresses type-based criteria for a pointcut. It states that whenever the proxy wraps an object of type *ICustomerServices* it has to load and execute all listed handlers. The attribute *order* indicates the order in which the particular handler has to be invoked.

From this XML snippet, the result of this is that *ICustomerServices* is now a log-enabled type.

Defining Handlers

All that remains to be done—and it is the key step, indeed—is to take a look at the code for a sample handler. In this case, it is the *DbLogger* class:

```
public interface ILogger
{
    void LogMessage(string message);
    void LogMessage(string category, string message);
}

public class DbLogger : ILogger, ICallHandler
{
    // ILogger implementation
    public void LogMessage(string message)
    {
        Console.WriteLine(message);
    }
    public void LogMessage(string category, string message)
    {
        Console.WriteLine(string.Format("{0} - {1}", category, message));
    }

    // ICallHandler implementation
    public IMethodReturn Invoke(IMethodInvocation input, GetNextHandlerDelegate getNext)
    {
        // Advice that runs BEFORE
        this.LogMessage("Begin ...");

        // Original method invoked on ICustomerServices
        IMethodReturn msg = getNext()(input, getNext);

        // Advice that runs AFTER
        this.LogMessage("End ...");

        return msg;
    }
    public int Order{ get; set; }
}
```

The class *DbLogger* implements two interfaces. One is its business-specific interface *ILogger*; the other (*ICallHandler*) is a PIAB-specific interface through which advice code is injected into the class's aspect list. The implementation of *ICallHandler* is fairly standard. In the *Invoke* method, you basically redefine the flow you want for any aspect-ed methods.

In summary, whenever a method is invoked on a type that implements *ICustomerServices*, the execution is delegated to a PIAB proxy. The PIAB proxy recognizes a few handlers and invokes them in a pipeline. Each handler does the things it needs to do before the method executes. When done, it yields to the next handler delegate in the pipeline. The last handler in the chain yields to the object that executes its method. After that, the pipeline is retraced

and each registered handler has its own chance to execute its postexecution code. Figure 3-7 shows the overall pipeline supported by PIAB.

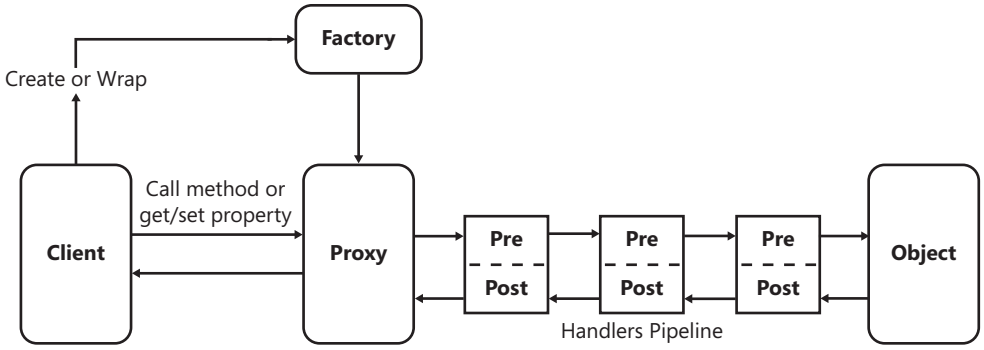


FIGURE 3-7 The PIAB handler pipeline

Note For completeness, we should mention that there are other AOP alternatives available for you to use, the most notable of which is COM+, although WCF exhibits AOP behavior as well. With COM+, you modify your aspects using Component Services. This assumes, of course, that you've taken the necessary steps to register the components by using *System.EnterpriseServices* or are using "services without components." (See *http://msdn.microsoft.com/en-us/library/ms686921(VS.85).aspx*.) Both COM+ and WCF AOP discussions are beyond the scope of this chapter, but by all means investigate the possibilities. (For a bit more information on WCF, see *http://msdn.microsoft.com/en-us/magazine/cc136759.aspx*.)

Practical Advice for the Software Practitioner

To design good software, general principles are enough. You don't strictly need patterns; but patterns, if recognized in a problem, are an effective and proven shortcut to get to the solution. Today, reinventing the wheel is a great sin, for yourself and your team.

Patterns are not essential to the solution of a problem. Using patterns won't make your code necessarily better or faster. You can't go to a customer and say "Hey, my product uses the composite pattern, a domain model, inversion of control, and strategy *à gogo*. So it's really great." Patterns, if correctly applied, ensure that a problem will be solved. Take an easy approach to patterns, and don't try to match a given pattern to a problem regardless of the costs of doing so.

Having mixed basic and OOD principles for years, we think we have now arranged our own few pearls of software design wisdom. These guide us every day, and we communicate them to all people we work with:

- Group logically related responsibilities and factor them out to classes. In the factoring process, pay attention to forming extremely specialized classes.

- Create concise and flexible abstractions of functionalities in classes. In this context, two other adjectives are commonly used in literature to describe abstractions: crisp and resilient.
- When it comes to implementing classes, keep in mind separation of concerns—essentially, who does what—and make sure that each role is played by just one actor and each actor does the minimum possible; this is not done out of laziness, but just for simplicity and effectiveness.

The emphasis on simplicity is never enough. Andrea has the following motto on a poster in the office, and Dino writes it as a dedication in each book he signs: *Keep it as simple as possible, but no simpler*. It's a popular quotation from Albert Einstein that every software professional should always keep in mind.

Often referred to as KISS (short for *Keep It Simple, Stupid*), the idea of "simplicity above all" emerges in various forms from a number of heuristic principles that populate software design articles and conversations. The most popular principles are these:

- **Don't Repeat Yourself (DRY)** Refers to reducing duplication of any information needed by the application, and suggests you store the same information only in one place.
- **Once and Only Once (OAOO)** Refers to reducing the number of times you write the code that accomplishes a given operation within an application.
- **You Aren't Gonna Need It (YAGNI)** Refers to adding any functionality to an application only when it proves absolutely necessary and unavoidable.

We often like to summarize the "simplicity above all" concept by paraphrasing people's rights in court: everything you write can and will be used against you in a debugging session. And, worse yet, it will be used in every meeting with the customer.

Summary

Just as an architect wouldn't design a house while ignoring the law of gravity, a software architect shouldn't design a piece of software ignoring basic principles such as low coupling and high cohesion. Just as an architect wouldn't design a house ignoring building codes that apply to the context, a software architect working in an object-oriented context shouldn't design a piece of software ignoring OOD principles such as the Open/Closed Principle (OCP), the Liskov Substitution Principle (LSP), and the Dependency Inversion Principle (DIP).

But other quality characteristics (defined in an ISO/IEC standard) exist—in particular, testability and security. These aspects must be taken care of at the beginning of the design process—even though magical testing tools and aspect orientation might partially alleviate the pain that comes from not introducing testability and security from the very beginning.

The fundamentals never go out of style, we can safely say, and they are always useful and essential at whatever level of detail.

Murphy's Laws of the Chapter

This chapter is about design and proving that your design is effective and meets requirements. If you think of acceptance tests, in fact, how can you help but recall the original Murphy's Law: "If anything can go wrong, it will." We selected a few related laws for the chapter.

- The chances of a program doing what it's supposed to do are inversely proportional to the number of lines of code used to write it.
- The probability of bugs appearing is directly proportional to the number and importance of people watching.
- An expert is someone brought in at the last minute to share the blame.

See *http://www.murphys-laws.com* for an extensive listing of other computer-related (and non-computer-related) laws and corollaries.

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Authors

Dino Esposito

Dino Esposito is an IDesign (*http://www.idesign.net*) architect and a trainer based in Rome, Italy. Dino specializes in Microsoft Web technologies, including ASP.NET AJAX and Silverlight, and spends most of his time teaching and consulting across Europe, Australia, and the United States.

Over the years, Dino developed hands-on experience and skills in architecting and building distributed systems for banking and insurance companies and, in general, in industry contexts where the demand for security, optimization, performance, scalability, and interoperability is dramatically high. In Italy, Dino and Andrea, together, run Managed Design (*http://www.manageddesign.it*), a premier consulting and training firm.

Every month, at least five different magazines and Web sites throughout the world publish Dino's articles covering topics ranging from Web development to data access and from software best practices to Web services. A prolific author, Dino writes the monthly "Cutting Edge" column for MSDN Magazine and the "ASP.NET-2-The-Max" newsletter for the Dr. Dobb's Journal. As a widely acknowledged expert in Web applications built with .NET technologies, Dino contributes to the Microsoft content platform for developers and IT consultants. Check out his articles on a variety of MSDN Developer Centers such as ASP.NET, security, and data access.

Dino has written an array of books, most of which are considered state-of-the-art in their respective areas. His more recent books are *Programming Microsoft ASP.NET 3.5* from Microsoft Press (2008) and *Programming Microsoft ASP.NET 2.0 Applications—Advanced Topics* from Microsoft Press (2006).

Dino regularly speaks at industry conferences all over the world (Microsoft TechEd, Microsoft DevDays, DevConnections, DevWeek, Basta) and local technical conferences and meetings in Europe and the United States.

Dino lives near Rome and keeps in shape playing tennis at least twice a week.

Andrea Saltarello

Andrea Saltarello is a solution architect and consultant at Managed Designs (*http://www.manageddesigns.it*), focusing on architecture and virtualization topics.

He has spoken at events and conferences in Italy and has also taught "Operating Systems" during the "Master in Editoria Multimediale" class organized by the university "Politecnico of Milan."

In 2001, Andrea co-founded UGIdotNET (*http://www.ugidotnet.org*), the first Italian .NET User Group, of whom he is the president.

Andrea is passionate about sports and music, and grew up playing volleyball and listening devotedly to Depeche Mode, a group he fell in love with after listening to "Everything Counts" for the first time.

These days he tries to keep in shape by catching up to balls on squash or tennis courts, and he enjoys going to as many live music gigs as he can.

Andrea has a blog at *http://blogs.ugidotnet.org/mrbrightside*.