THE EXPERT'S VOICE® IN SOFTWARE DEVELOPMENT

Practical Enterprise Software Development Techniques

Tools and Techniques for Large Scale Solutions

GET UP TO SPEED QUICKLY BY LEARNING INDUSTRY STANDARD PRACTICES AND COMMON TOOLS

Edward Crookshanks

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Purpose

The purpose of this book is to discuss and provide additional resources for topics and technologies that current university curriculums may leave out. Some programs or professors may touch on some of these topics as part of a class, but individually they are mostly not worthy of a dedicated class, and collectively they encompass some of the tools and practices that should be used throughout a software developer's career. Use of these tools and topics is not mandatory, but applying them will give the student a better understanding of the practical side of software development.

In addition, several of these tools and topics are the "extra" goodies that employers look for experience working with or having a basic understanding of. In discussions with industry hiring managers and technology recruiters, the author has been told repeatedly that fresh college graduates, while having the theoretical knowledge to be hired, are often lacking in more practical areas such as version control systems, unit testing skills, debugging techniques, interpreting business requirements, and others. This is not to slight or degrade institutional instruction, only to point out that there are tools and techniques that are part of enterprise software development that don't fit well within the confines of an educational environment. Knowledge of these can give the reader an advantage over those who are unfamiliar with them.

This guide will discuss those topics and more in an attempt to fill in the practical gaps. In some cases, the topics are code-heavy; in other cases, the discussion is largely a survey of methods or a discussion of theory. Students who have followed this guide should have the means to talk intelligently on these topics and this will hopefully translate to an advantage in the area of job hunting. Although it would be impossible to cover all tools and technologies, the ones covered in this guide are a good representative sample of what is used in the industry today. Beyond the theoretical aspects of computer science are the practical aspects of the actual implementation; it is this realm that this book attempts to demystify.

There are some "topics de jour" that have been left out; this is by design. To cover every emerging technology and technique would quickly overwhelm the reader and could have the effect of overemphasising a passing fad. Also, enterprise development is typically governed by architects and risk managers; in their minds, new technology is inheritly risky and, therefore, change in the enterprise occurs comparatively slowly. This book attempts to stick with techniques, tools, and concepts that are the basis for development in the enterprise; if these are understood, then layering new tools on top should be easy to do.

In short, it is hoped that this companion guide will help graduates and even relatively new hires overcome the "lack of practical experience" issue by becoming more familiar with industry standard practices and common tools. This volume cannot create experts, but it can at least provide enough cursory knowledge such that the reader can discuss the basics of each topic during an interview. With a little practice and exploration on their own, the student should realize that supplementing an excellent theoretical education with practical techniques will hopefully prove useful not only in writing better software while in school but also translate to an advantage when out of school and searching for a job.

Overview of Topics

The following topics and tools are discussed:

- How enterprise software development differs from academia and small business settings
- Writing requirements
- Design patterns and architecture
- Comparison of development methodologies
- Version control
- Unit testing and Test-driven development
- Refactoring
- Debugging
- Build tools, automated build engineering, and continuous integration
- Basic SQL statements and data frameworks

In addition, there are discussion questions for each chapter listed in Appendix B. In a classroom setting these can be used for review, classroom discussion, or as a starting point for further discussion.

Prerequisites

It is assumed the reader is already familiar with many facets of languages and tools. The typical student would have used Java, .NET, C++, or some other high-level language for course assignments in a typical computer science or software curriculum and is probably at the sophomore, junior, or senior level. The reader should also be familiar with the differences between console applications, GUI applications, and service/daemon

applications. The nuances of procedural, object-oriented, and event-driven programming should be understood at a high level. The examples will be kept as simple as possible where needed because the intent is not to teach CS topics and object-oriented design but instead how to use these particular tools and concepts to assist in implementing the problem at hand.

Disclaimer

The tools and techniques discussed in this guide are not the only ones on the market. If a particular tool is mentioned in this guide it does not mean that it is the only tool for the job, is endorsed in any way by the author or publisher, or is superior in any way to any of its competitors. Nor does mention of any tool in this publication insinuate in any way that the tool owners or resellers support or endorse this work.

Java and Java-based trademarks are the property of Sun Microsystems. Visual Studio® is a registered product of Microsoft and all other Microsoft product trademarks, registered products, symbols, logos, and other intellectual property is listed at http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx. Eclipse™ is property of the Eclipse Foundation. All other trademarks, registered trademarks, logos, patents, and registered names are the property of their respective owner(s). We are aware of these ownership claims and respect them in the remainder of the text by capitalizing or using all caps when referring to the tool or company in the text.

Any example code is not warranted and the author cannot be held liable for any issues arising from its use. Also, the references provided for each topic are most definitely not an exhaustive list and, again, their mention is not to be construed as an endorsement, nor is any resource left off the list to be considered unworthy. Many additional books, websites, and blogs are available on these topics and should be investigated for alternate discussions. Any mentions of names anywhere in the book are works of fiction and shouldn't be associated with any real person.

What Is NOT Included

Unfortunately in many enterprise development situations, the latest bleeding edge technology and tools are not frequently used. This can be for any number of reasons including risk, enterprise architecture standards and policies, and simple scale of change. For example, even though web systems using NodeJS and MongoDB are very popular in web development (as of early 2015), these technologies aren't nearly as common in the enterprise world. Large corporations may have hundreds of thousands of dollars invested in Windows Server farms running ASP.NET and SQL Server (not to mention large teams supporting them) and are not willing to change as readily as small development businesses. So although many of the concepts discussed in this book can be applied to any language, framework, or technology, many of the current "hot" technologies are not discussed here. This is not to slight them at all or say they have no place in enterprise development, but the purpose of this book is not to emphasize the latest trends in tools and technologies.

Software Notes

Examples are provided in a variety of languages and with different tools, all of which have some level of free software available. Enterprise versions of these tools may exist, or similar tools with stricter licensing models and slightly different semantics may exist; it is simply assumed that tools at the educational level will be closer to the free versions. Also, most hobby developers or recent graduates will probably make use of free tools instead of starting with an expensive development tool suite. If a particular topic/example is not given in a familiar language or with a familiar tool it should be easily translated into another environment. Where possible, notes on how different platforms solve different problems in different ways will be noted. Some of these tools may already be mandated by an employer, others may be free to choose which tools to use to start a practice discussed here. The development tools were originally released with tools available in early 2011 with the first edtion, namely, the Visual Studio 2010 Express Editions, and Eclipse 3.6. Other tools used in the text were obtained around the same time. In 2014, newer edtions were tested with success, including Visual Studio 2013, Eclipse 4.4, and Spring Tool Suite, and others. There are further notes in the downloadable source code. However, one common experience with enterprise software development is that versions are often at least one or two behind the current market standard. Usually this is not a great impact if the software being developed doesn't employ bleeding edge technology, and most risk managers is enterprise situations won't allow that.

Please note that the examples will be kept necessarily simple. In fact, most of the examples will be so short that the tools and techniques used on them will probably not seem worth it. However, in the context of much larger systems, enterprise systems, these tools and techniques are very useful.

A very special note on Chapter 5 on version control. Since the publication of the first edition of this book, a major paradigm shift has occurred in the repository world, and distributed version control, in particular GIT, has become commonplace. Many students and hobbyists alike have benefited from the true flexibility of GIT, its ease of use, and its power when sharing code among distributed teams and via GITHUB.com. However, large enterprises have been slower to adopt GIT as an enterprise-wide tool, not necessarily because of its distributed model, but often because of the sheer amount of retooling it would take to make the change. A small software shop with only a few developers and a single source repository has far more flexibility than an enterprise organization with four thousand software developers, multiple code repositories, and multiple layers of infrastructure to support all of that and the systems that interface with it such as continuous integration and automated deployment applications. So large organizations that have invested in a working centralized version control such as SVN, CVS, or Team Foundation Server will most likely not quickly move to GIT, as it potentially represents a large effort, and therefore a large risk. So to summarize an overused cliché, when it comes to version control (or any dedicated infrastructure component for that matter), many large organizations may be aware of the latest and greatest cool software, but will follow the "If it ain't broke, don't fix it" mentality.

Or at least fix it very slowly. In many cases, large "legacy" repositories would be left in place and their current integrated systems such as continuous build and deploy systems would be left as is. New repository systems, for example, Jira Stash, would be brought online completely separate from the legacy systems. New projects would use the new repository, and continuing projects would stay on the legacy system as migrated as time permits.

For that reason, the chapter on version control remains in place and mainly centers on SVN. Even if other tools are used, the concepts are easily demonstrated with SVN and the tortoise shell exentsion. Distributed version control is mentioned, but not emphasized because the concepts of check-in/check-out and the other main topics apply equally to GIT as to SVN.

CHAPTER 1

How Enterprise Software Is Different

Computer Science, software engineering, and software development are similar terms that are often used interchangeably. Likewise, there are many different types of educational opportunities available – Bachelor's programs, associate programs, trade schools, and high-intensity immersion programs. The intent of each of these is to educate the student in varying degrees of theory and turn out an individual capable of understanding and writing software.

Although each education program can produce students who can write software, often the scope of learning is either very broad (universities) to very narrow (immersion programs). A discussion of which education method is best is beyond the scope of this book; the intent here is to discuss and illustrate tools and techniques commonly used in professional enterprise software development. Some programs or professors may touch on some of these topics as part of a class, but individually they are mostly not worthy of a dedicated class, and collectively they encompass many of the tools and practices that should be used throughout a software developer's career in a team environment. In the case in which a dedicated class might exist (e.g., Design Patterns), a quick overview of the underlying principal is given but the majority of the discussion centers around how the principal is useful in the enterprise and why. The goal of the chapter is not to reteach patterns but to demonstrate how and why a few of them are important in an enterprise setting.

The Association for Computing Machinery (ACM) put out their most recent curriculum guideline in 2013. Although additional information was included concerning professional development and an added emphasis on including "professional practice," the recommendation was fairly vague and lacking deterministic details about exactly what those skills should be and how they should be taught. The topics selected for discussion here came from the author's personal experience with interviewing recent graduates and those developers with less than five years' experience. Also, in many discussions with industry hiring managers and technology recruiters, the author has been told repeatedly that fresh college graduates, while having the theoretical knowledge to be hired, oftentimes are lacking in more practical areas such as interpreting business requirements, unit testing skills, version control systems, debugging techniques, collaboration tools, and so on. This is not to slight or degrade institutional instruction, only to point out that there are tools and techniques that are part of enterprise software

development that don't fit well or are rarely taught within the confines of an educational environment. Nor are these consistently used in smaller software efforts such as startups or single-application firms. Knowledge of these principles can give the reader an advantage when going in to an environment where they are used.

Why are these practical topics important? As noted earlier, they are important because they are rarely formally discussed and the industry as a whole is continuously striving for a quicker "time to productivity" for new programmers. And although most new graduates are comfortable writing code, especially if they already have one or two years of actual experience, they may not be familiar with large team or enterprise practices.

Because both tools and techniques are covered, in some cases the topics are code-heavy; in other cases, the discussion is largely a survey of methods or a discussion of theory. Students and newcomers who have absorbed the material in this book should have the means to talk intelligently on these topics and this will hopefully translate to an advantage in the area of job hunting or quicker acclimation to a team environment. Although it would be impossible to cover all tools and technologies in various usage scenarios, the ones covered in this guide are a representative sample of what is used in the industry today. Beyond the theoretical aspects of computer science are the practical aspects of the actual implementation; it is this realm that this book attempts to demystify.

But isn't software development still software development no matter where it is written? The answer is both yes and no. Yes, at the end of the day the goal is to produce software that does what the end user wants. Even in large organizations, development efforts can be broken into small enough teams that coding seems very much like any other small team. And depending on management, the effect could be a small team feel as part of a much larger organization.

The "no" part of the answer stems from the corollary of "how do we get there?" In academia and even in small startup companies, the focus is usually on learning the theory or producing a website or product to sell and generate revenue. Typically the work is individual (academia) or in very small teams (startup); the final solution/product may be specified with the end goal of simply getting there at all costs.

In smaller companies, employees may wear many different hats, that is, being a developer, database administrator, tester, and deployment engineer all in one day. This is usually out of necessity and because of the limited resources available. When there are only seven people in the company, "many hats" is simply a fact of life. If the effort is around producing an app or service that contains mainly user-relevant data, some of those steps may not even be necessary. And although these experiences are good for understanding all the links in the chain, they may not be appropriate or even permissible in all organizations.

In enterprise development, there can be several additional factors present that are of little or no concern to small companies. Large organizations may be required by law or regulation to have more stringent processes around software development. Later discussions in this book involve multiple environments and separation of duties, but these concepts are usually in place in large organizations to reduce operational risk or a result of other corporate policy concerns. So understanding all links in the chain is important. For example, instead of changing a database entry manually, a SQL script may have to be sent to an administrator and the results verified without access to the database. In that case, knowing what the DBA has to do and how he will do it would be beneficial should problems arise.

Also, many enterprise development efforts are internally focused; therefore the "end user" is also an employee of the company and the developer is not directly responsible for generating revenue or producing a product. The software being developed could be for internal-only use (a payroll system), an operations system (scheduling plant maintenance), or to help an employee better serve an external customer (a bank teller application). And, as internal software, the entire infrastructure is usually kept internal as well, meaning that web servers and database servers are housed internally and all networking is internal via an intranet. Also, many systems may share a particular piece of infrastructure and changes to them, even for a single application, must be coordinated through them. This can be another very different aspect of enterprise software development.

To repeat and expand on the introduction, here again is the list of topics. A brief statement concerning the rational of each topic was chosen and how each is used in the enterprise or can differ from smaller development efforts is added to clarify the purpose of the discussions.

Chapter 2, Requirements – Large enterprises may still have formal project management teams in place even if development uses less formal methods. Being able to produce and understand the various documents is essential.

Chapter 3, Design Patterns and Architecture – There are often design pattern classes in academia. The purpose of this chapter is to (re)introduce patterns that are important in an enterprise environment and discuss why they are important. Also, enterprise architecture is discussed, both as a rational to why the design patterns are important and to highlight typical separation of duty issues.

Chapter 4, Comparison of Development Methodologies – This is a survey discussion of the two major methodologies, waterfall and agile. Agile is a broad brush and several of the different aspects are summarized. Again, like requirements, there may be different methodologies at different levels of the organization and knowing about each can be beneficial.

Chapter 5, Version Control – This chapter is a review of version control basics and a discussion about version control in an enterprise setting. With the recent popularity of Git and distributed collaborative efforts, students are generally more aware of the principles of version control, but enterprises may employ a slightly different model as a result of separation of duty concerns.

Chapter 6, Unit Testing and Test Driven Development – Test Driven Development (TDD) is a form of agile programming, but it is included in this section because of how tightly it is coupled with unit testing. Often in academic settings the practice is "code until it works" with minimal thought for ongoing modification and verification. This chapter discusses those ideas, how important they are for ongoing changes and maintenance, and illustrates some of the tools used.

Chapter 7, Refactoring – This is often done informally; this chapter formally defines some of the terms and acts as a complement to the TDD chapter. And again, refactoring in the enterprise is commonly done to increase extensibility or when new integration points need to be added, which is also the next step after "code until it works."

Chapter 8, Debugging – This is often not formally covered anywhere. In this chapter, some of the more powerful techniques are discussed and demonstrated. Also, logging is discussed as it is one of the only ways to "debug" in a production environment in which the developer has no access rights.

Chapter 9, Build Tools, Automated Build Engineering, and Continuous Integration – These tools and techniques are common in enterprise environments because of separation of duties and risk. Developers are often not allowed to compile and deploy code outside of the development realm, meaning that in the testing and production environments other teams handle this. They, in turn, are typically not developers and may make use of automation tools to accomplish the task. Knowing this process and preparing for lack of access outside of development is very beneficial.

Chapter 10, Basic SQL Statements and Data Frameworks – Although key in many projects, database theory and programming is not necessarily a required class, especially in associate programs or similar environments. Sometimes automated tools are used to create a data framework and do all the "behind-the-scenes plumbing" work. This chapter is a basic review/introduction to SQL statements to assist in understanding how to directly interact with a database. In addition, data framework programming is briefly touched on to demonstrate how relational data is presented in a language construct with some of the latest tools.

Summary

Enterprise software development can often be much more than just writing code. Collaboration with other teams and team members is important, as are the extra steps in documenting the process. In addition, the actual code can be different as well. Designing for integration, expandability, and flexibility in an environment that the developer may not have access to is a key consideration. Enterprises may also use various tools to automate certain processes and/or reduce risk. The remaining chapters attempt to highlight some of these important areas and prepare the reader for developing software in the enterprise.

CHAPTER 2

Software Requirements

Writing good requirements is difficult. Interpreting bad requirements is even tougher. But almost all projects start with them and thus they are extremely important. In this chapter, we will define requirements from a couple of different perspectives, discuss how the different perspectives work together, and list some general recommendations about each. We will also cover the roles the developers play in the requirements process and why developing and interpreting requirements are important skills. As with many of the topics in this book, there are countless permutations of the concepts that we are trying to cover and other works that are dedicated solely to the topic of requirements. Here we present a general overview with an emphasis on aspects that are important to developers from an enterprise or large organization perspective.

In our section on Test Driven Development and the Agile method, we will discuss minimal documentation and using index cards or story cards to capture desired behavior. Here we address the other extreme – documenting requirements in a more "traditional" project management manner so that both the business and technical project managers can track their status through the life of a project. This is often part of the waterfall process. In this chapter, we will use a human resources time-card application as our target system. All examples will be in the context of developing this as an internal application for internal users. Similar concepts apply to external and retail applications, however.

Business Requirements

Business analysts – those familiar with the day-to-day operations of the end users – will usually be instrumental in bridging the gap between business terminology and technical implementation. These analysts will assist in documenting requirements (described later) and will represent the business users to the technical staff in discussions.

At the highest level of abstraction is the Vision Statement, sometimes referred to as Primary Business Goal. This is a document or statement usually written from the business perspective describing the overall strategic goal of the software. Typically this statement is brief—only a few sentences or paragraphs long.

One theoretical goal statement is shown here. The statement is admittedly simple and brief, but the high-level purpose of the system is summed up in a single statement:

> *To provide employees and management with an easy-to-use, electronic time-card system that allows tracking of work, vacation, holidays, volunteer time, and other work-related activities.*

The vision statement describes the overall purpose of the application and may not necessarily need to be changed with each additional feature. However, for major changes (or when starting from scratch), the first document constructed is typically a Business Requirements Document, often referred to as a BRD. In this document, the business, with the assistance of the business analyst, spells out in business terms what the software should do. The document is typically written in business terms; the technical team will create their own document spelling out the technical details needed to accomplish what the business wants.

Use cases are another higher-level abstraction that begins to show interaction with the system. Actors are defined and are shown interacting with specific processes or components that produce a specific outcome. It is a standard convention that a use case begins with a verb before the component being acted on (Wiegers, 2006). There are several ways to define use cases; the two most common being a use case diagram in UML and a more detailed written description.

In Figure 2-1, an overly simplified use case diagram for some of the operations in the example time card system is shown. In this diagram there are four actors and six use cases; this shows that use cases may be shared among actors. Also note that each use case is a meaningful action that is carried out by each actor. The diagram doesn't show any restrictions or special conditions. When more detail is needed for a use case, the diagram is supplemented with a written description.

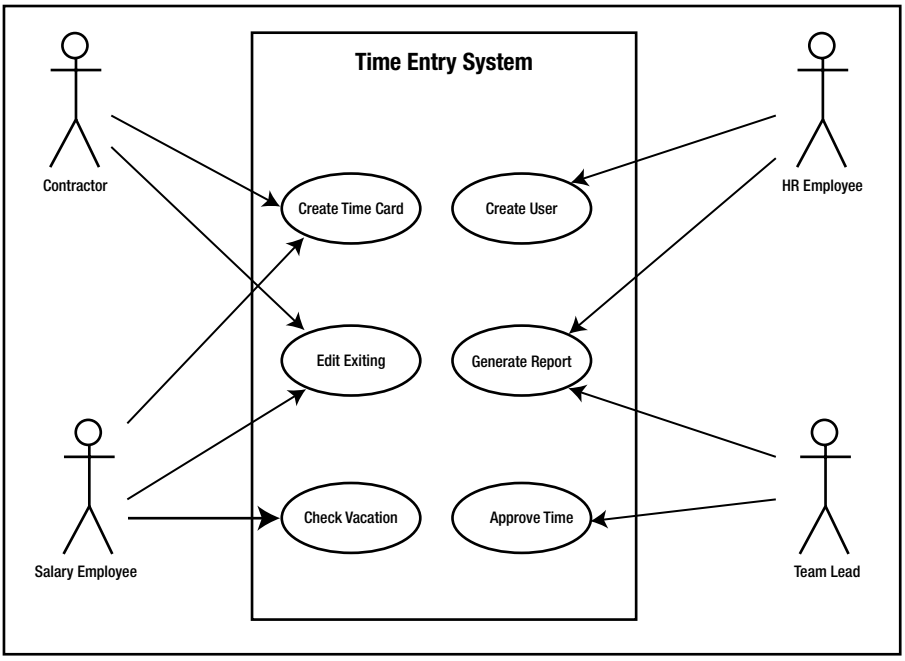

Figure 2-1. *Example use-case diagram*

Formatting for the written description varies widely and the example presented here is one of many formats. The amount of information and level of detail can differ depending on the complexity of the use case, the use of standard templates, or other organizational and/or team factors. Sometimes a use case can be an informal paragraph, but the example we show here is moderately more complex. In the example HR timecard system, each use case in the diagram would be matched by a corresponding use case description. The example shown in Table 2-1 is loosely based on the template in Wiegers (2006) but, as mentioned earlier, many formats are available and there is no single industry standard use case description template.

Table 2-1. *Example use case written description*

Use Case Name	**Create Time Card**
Created by:	EC
Date Created:	8-1-2011
Actors:	Contractor, Salaried Employee
Description:	Actors should be able to create a time card for a specific reporting period. They should be able to add tasks and hours for the period and mark it “ready for review” when done.
Priority:	1
Assumptions:	Actors set up with proper projects and task choices.
Rules:	Actors can save a time card without completing it—they may enter the entire period at once or update daily or as needed before completing. Before marking “ready for review” each work day must have an entry. Two time cards cannot be created for the same payroll period by the same person. Contractors are not allowed to enter vacation time.
Preconditions:	Actors successfully signed(?) onto the system.
Notes:	

In addition to or sometimes in place of use cases, the BRD may include more specific business user requirements spelled out in sentence form. These items may map more closely to the technical implementation. In fact, they may actually spell out some specific items to be implemented in the source code. In the written description, the “Rules” section most closely resembles these specific requirements but in a typical BRD these “line items” have identifiers that project managers and team leads can use to address a specific item. Table 2-2 shows a small sample of this type of BRD entry.

Table 2-2. *Example user requirements*

ID	Description	Priority	Status
R-11	Users should not have to sign into the system; their current network login should be used for identification.	Med	Done
R-12	The user should pick a project first; the tasks available are a derivative of the project.	High	Open
R-13	A full-time employee should not be able to submit a time card with less than 40 hours per week recorded.	High	Open
R-14	A contractor can submit any number of hours up to 60 without special approval.	Med	Open
R-15	A team lead can see his/her team's time cards before they are submitted but cannot approve them until the user submits it.	High	Open
R-16	Each approver should have a surrogate approver with the same approval rights.	High	Open
R-17	If the user attempts to submit a time card without an entry for each work day, the user should see an alert stating "Time card incomplete" and the action should fail.	High	Open
R-18	If a user is inactive for 5 minutes the system will timeout and kick the user off.	Low	Open

Note that the sample is an overly simple example, and most likely a real BRD would have additional columns. These might be a requirement category, such as "System," "Legal," "Performance," and so on. There could also be a "Notes" column for additional information, a "Date" for when the requirement was added, a desired release number if the project is in phases, and more. The main point is that the statements are from the business perspective and while not technical, care must be taken to understand exactly how a business expectation translates into a technical implementation. For example, a business requirement of "User should be able to search entire intranet for search term in less than one second" is likely unattainable and that sort of requirement should be tempered in consultation with the business analyst.

Functional Design

Once the business requirements have been completed, another typical piece of documentation is the Function System Design, or FSD. In many organizations, this serves two purposes: to address the coverage of the business requirements and to specify the

actual system implementation down to the class, object, and possibly even the sequence diagram level. In other scenarios, it simply focuses as a logical design document to ensure that all aspects of the Business Requirements are covered. As there are plenty of other references on classes, objects, object-oriented design, and sequence diagrams, here we will address the coverage of the business requirements. Later on, we will discuss technical specifications at a high level in a section on Technical Design.

As stated for the other items in this section, there are countless formats, templates, and structured documents that may be called an FSD. The focus in this section is to show a mapping between the requirements in Table 2-3 and a functional statement that implements the requirement in question. Oftentimes this is referred to as a "traceability matrix" and can provide a way of verifying that all requirements are covered by at least one functional design statement.

Table 2-3. *Sample FSD statement showing requirement number*

ID	Description	Bus Req	Status
F-23	The system will use Windows integrated authentication for identifying the user.	R-11	Assigned
F-24	The list of active tasks will not be retrieved until a project is selected. These will be child-parent tables in the database and can be associated.	R-12	Open
F-25	The session timeout will be configurable and default to 5 minutes.	R-18	Assigned

One representation involves the creation of a set of statements similar to Table 2-2; however, these statements describe technical aspects of the system. They would also have an additional column stating which requirement in the BRD is covered by this functional statement. These two columns could easily be extracted into a matrix to compare the mappings between business requirement ID and function requirement ID.

Another popular format is to lift the requirements table from the BRD and insert it directly into the functional design. An additional column is then added to the table, which shows the functional implementation directly alongside the requirement. This is shown in Table 2-4.

Table 2-4. *Combined FSD and BRD*

ID	BRD Description	Functional Solution
R-11 F-23	Uses should not have to sign into the system; their current network login should be used for identification.	The system will use Windows integrated authentication for identifying the user.
R-18 F-25	If a user is inactive for 5 minutes the system will timeout and kick the user off.	The session timeout will be configurable and default to 5 minutes.

Technical Design

Although the BRD and FSD are generally written at a high level so that both analysts and developers can interpret the information, "technical design" is a term given to documentation that is targeted toward developers. Whereas some teams may have a standard "Technical Design Document (TDD)" or "Software Design Document (SDD)," multiple documents can also make up the technical design documentation. The most basic of these would be the flowchart, but others are common as well. These could include:

- Database design diagram
- UML diagrams, such as class diagram, sequence diagram, and so on
- API documentation
- Architecture diagrams

An introduction to database design is discussed in Chapter 10: Just Enough SQL. A basic database diagram is shown in Figure 10-1. In short, the database diagram shows the structure of the database such as the table names and data types, constraints, and relationships. Refer to Chapter 10: for more information.

UML (Unified Modeling Language) is a way to express several aspects of a software project. A use-case diagram has already been shown in Figure 2-1; this section will discuss lower-level diagrams such as the class diagram and sequence diagram. There are many more possible diagrams and covering them all is beyond the scope of this book. The reader should refer to Appendix D for additional resources.

A class diagram is a representation of the static class structure of an application. It illustrates class names, attributes, operations, and relationships between classes. Figures 2-2 and 2-3 show the beginnings of a class diagram for the time tracker application discussed in this section. Figure 2-2 shows the beginnings of modeling the application users and includes classes and interfaces; Figure 2-3 shows how reports for the application could be represented as code classes.

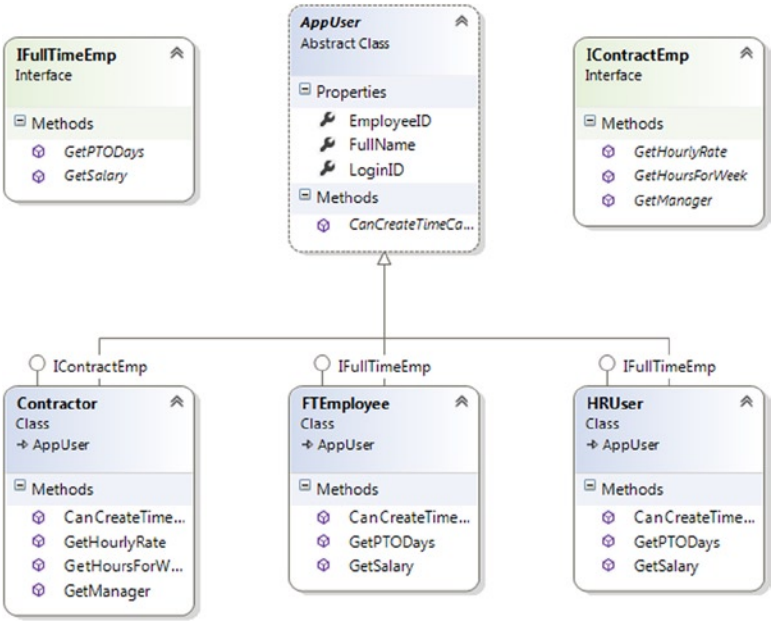

Figure 2-2. *Very simple class diagram of users*

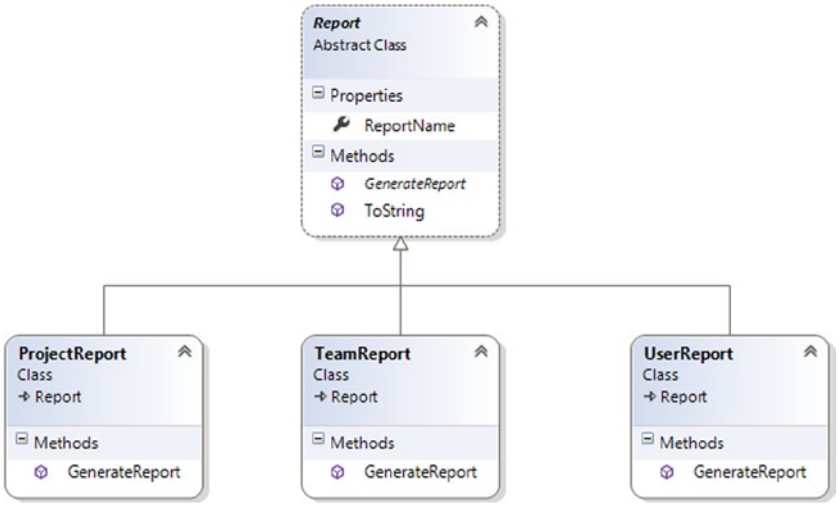

Figure 2-3. *Simple class diagram for report classes*

Obviously these are very early-on diagrams; as the application progresses, the diagrams may become much more complex. Keep in mind also that there are several different ways of generating these diagrams. Some development tools will automatically create diagrams based on the project structure, but diagrams are often done beforehand and maintained manually by using tools such as Microsoft Visio®.

Sequence diagrams represent the internal moving pieces of software. These diagrams illustrate what is going on at the object and method level. They highlight method calls, method return values, and object lifetimes. An example is shown in Figure 2-4.

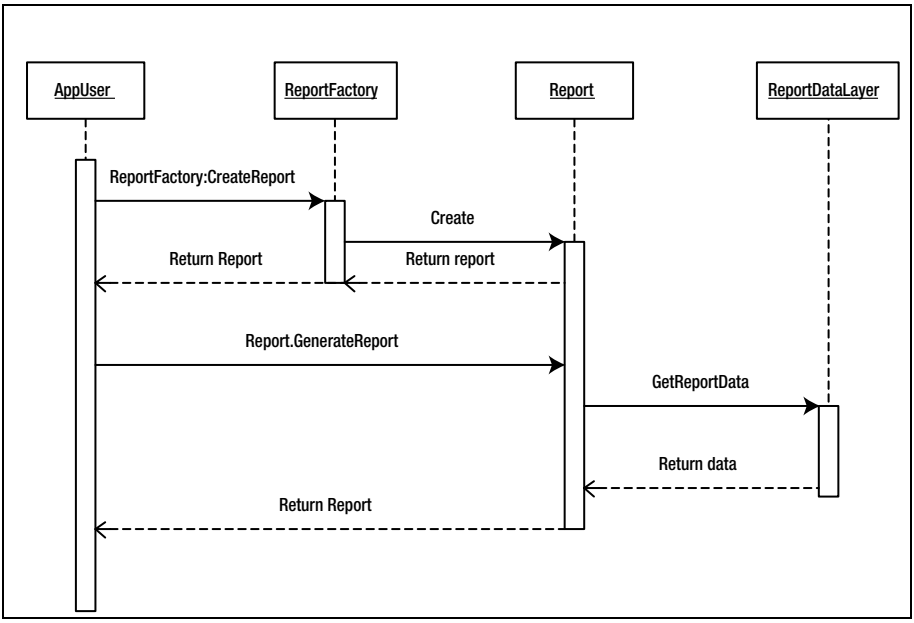

Figure 2-4. *Sequence diagram*

There are generally two levels of detail for sequence diagrams. Some are at a somewhat higher level and don't specify actual method names or parameters, only actions and interactions. These are useful for reviewing object interactions and getting a sense for how many objects are needed to produce a desired result. Other sequence diagrams can be very detailed, specifying actual method names with parameters, looping structures, and more.

API documentation is among the very lowest level of technical documentation. Simply put, every method of every class, its parameters and return value, and sometimes even usage context is included in this documentation. Numerous examples of this exist online, the most prevalent being Microsoft's MSDN documentation for the .NET library and Oracle's JavaDoc documentation.

As with class diagrams, this information can be generated manually or automatically. Manual creation of this documentation is done using a simple text editor. However, a more popular way of creating this type of documentation is through documentation comments in the source code itself. These are very similar to normal comments except for the special identifiers and tags and the fact that they begin with /** (two asterisks) instead of the normal one asterisk. Tools exist for taking these comments and producing standard format documentation in various formats such as HTML or XML.

Listing 2-1 shows an example of the comment format in Java that is used to produce the code documentation. By comparing with the generated page shown in Figure 2-5 with the tags (@author, @param, @return), it can be seen how the comments translate to generated information.

Listing 2-1. Java documentation comments

```
package com.nokelservices;

/**
 * How about an actual comment in here and not just the @author?
 * @author Author Name
 *
 */
public class SettingValidator
{
    /**
    * Validates the given key/value pair for allowable values
    * @param key key of the setting to validate
    * @param value value of the setting to validate
    * @return boolean indicating valid value or not
    */
    public boolean isValidSetting(String key, String value)
    {
        return (key.length() > 0 && value.length() > 0);
    }
}
```

These comments can be applied to the package, class, and methods, as well as other types such as enumerations. There are also more tags than are shown here; please refer to the online documentation for a complete list of these tags and their uses. Once these comments are processed, a series of web pages are generated that follow a standard format. Figure 2-5 is the page generated from the Java code and comments.

com.nokelservices

Class SettingValidator

```
java.lang.Object
  └ com.nokelservices.SettingValidator
```

```
public class SettingValidator
extends java.lang.Object
```

Author:
Author Name

Constructor Summary

SettingValidator()

Method Summary

boolean	isValidSetting(java.lang.String key, java.lang.String value) Validates the given key/value pair for allowable values

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

SettingValidator

```
public SettingValidator()
```

Method Detail

isValidSetting

```
public boolean isValidSetting(java.lang.String key,
                              java.lang.String value)
```

Validates the given key/value pair for allowable values

Parameters:
key - key of the setting to validate
value - value of the setting to validate
Returns:
boolean indicating valid value or not

Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Figure 2-5. *Sample JavaDoc page*

Other languages may have their own commenting standards and procedures for documenting code. In .NET, the format of the documentation comments is very similar to the preceding Java format. PHP and Perl also have standards for creating code from comments.

The architecture diagram is highlighted in , as part of the pattern discussion. These diagrams are also often included in technical documentation. Not only are they useful for "big picture" discussions, but they also are useful in determining which teams to partner with and how many other teams will be necessary. Also, special hardware situations such as clustering and firewalls may be depicted in these diagrams, which may or may not result in some special code considerations.

Change Control

In a system in which rigid documentation principles are followed, the frequent changing of requirements is typically not allowed to happen. In the past, this frequently led to a phenomenon known as "scope creep." One would start out with a simple design to turn on a light switch and end up with a functioning nuclear plant—very late and very much over the original budget! This necessitated an approach in which after the requirements and design documents are completed and approved, any changes in requirements or the design implementation must follow a formal change control process. This is not to prevent changes to the system but, rather, to properly analyze and document them (Weigers, 2003). This intentional review also helps to limit "scope creep." If these aren't properly vetted and documented, they will cause the project to be late or improperly implemented. Through this process, some may be implemented right away; some may be designated as a requirement in a future release of the product.

Again, there are countless policies for handling change control as well as countless ways of documenting the process. Some organizations may use an email trail, some may use a "mini-BRD/FSD" structure, and still others may blend the two based on the severity of the change. Ideally, all changes should follow the same system, and the discussion in this section only outlines a single change control process. Also, we only briefly discuss the surrounding process and focus mainly on the documentation of the change.

In the time card project, assume that the BRD is complete and approved, the FSD is complete and approved, and actual coding has begun. After reviewing the document and current practices in a legacy system, the business analyst comes to the development team and states that there was a scenario that was missed and could slightly change the flow.

The new requirement is this: "a team lead may need to complete and approve a time card on behalf of the team members." This leads to the creation of a change request (CR) that will formally document this new requirement as part of the change control process.

For clarity and tracking purposes, each change request should be documented in a single document. Like requirements, a CR is given a unique identifier and has a brief description. However, because it is arriving after the design is complete and the coding has started, many more questions and factors have to be considered before adopting the new requirement as part of the implementation. Again, documentation standards vary widely, but a sample CR with additional questions and sample answers is shown in Table 2-5.