Chapter 1

Introduction

What Is Software Engineering?

As its title indicates, this book is about software engineering. Thus, it is appropriate to introduce some definitions and describe some concepts of software engineering. This will also be useful in order to acquire the correct mindset and get the right perspective to better understand the rest of the material. Bear in mind, however, that, as presented in the Preface, the discussion focuses solely on the functional aspects of software engineering. In reality, the scope of the software engineering discipline is much broader than just the functional aspect, as you can discover by reviewing the documentation from the Software Engineering Body of Knowledge (SWEBOK; `www.swebok.org`). Nevertheless, this short introductory section focuses on the functional aspect. Though this section might look more like a philosophical discussion, my premise is that it will bring useful insight for the rest of the presentation.

Here is the working definition of software engineering that I will use, and which is most applicable to the discussion in this book. It is a definition in two parts, which are related and need to be considered as a whole:

- Refinement of knowledge through successive abstraction levels of representation.
- Traceability of each and every item of information between abstraction levels.

Abstraction Levels

The first part of the definition states that software engineering is about manipulating *knowledge*. Indeed, developing a software system is about expressing some knowledge somehow. It is about *refinement*, because the knowledge is taken from a high-level vision down to code and system configuration. This refinement is structured into *abstraction levels*, where on each level the knowledge is in the form of a specific set of representations. Note that when refining the knowledge there is an expected increase in the diversity of the types of its representation.

Figure 1-1 shows the structure of the abstraction levels. One important distinction made is the clear separation of the domain space and the solution space. The domain space deals with the description of the business and captures the knowledge of the business. This description exists independently of any automation solution that may be implemented in a specific organization. This knowledge is captured in business models and has to be maintained to accurately reflect the evolution of the definition of business functions and the related business processes. Capturing the knowledge of the business is also the prerequisite for any organization to move up on the ladder of the Capability Maturity Model.[1]

Note that in the literature you may find a slightly different wording for this duality, where the domain space is referred to as problem domain and the solution space as solution domain. I personally prefer and advocate the definition in this book, as it is more consistent with the everyday language spoken in the business world. Indeed, you will more often hear people speak of their business domain knowledge than about their business problem knowledge.

I also like to view the domain space in the two aspects of business functions and industries. Each organization is structured around a number of business functions that together collaborate to perform the business of the

1. More information on the Capability Maturity Model can be found at `http://www.sei.cmu.edu/cmm`.

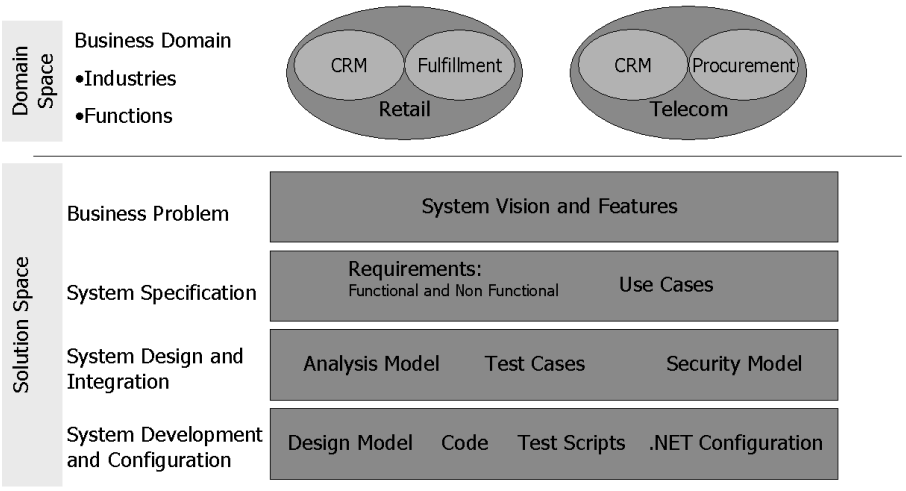

Figure 1-1: Abstraction levels.

organization. Examples of business functions are Customer Relationship Management (CRM), procurement, and fulfillment. Each organization is also a player within a particular industry, for example, Telecom, Retail, Banking, and others. Some business functions appear in every organization, irrespective of the industry they are part of.

It is interesting to note that while one business function is very likely to have a similar definition and terminology for all the organizations within one specific industry, the same business function may very well have a very different definition and terminology in a different industry. Moreover, within one specific industry, the more mature organizations are likely to have adopted a common definition and terminology for a specific business function. This awareness is important, as we will later consider the categorization of system features into business functions that may (or may not) be reusable, depending on the factors considered above. Appendix A, "Future Vision," shows you why this awareness is also particularly important for companies that produce prepackaged solutions for business functions.

In Figure 1-1, all the rest of the abstraction levels deal with the solution space. This is the business automation solution space, and it will be my main area of concern in this book.

I like to consider four levels of abstraction in the solution space. The *business problem definition* is likely to take the form of a system vision document, which documents a list of features. The knowledge at this abstraction level can be represented in a few pages of concise text. The *system specification* can be defined by the requirements and use case definition documents. This is where the refinement of the knowledge becomes clear. The knowledge representation (information) at the system specification level is obviously much more extended than at the business problem level. The knowledge continues to be refined through the *system design and integration* level to finally reach the *system development and configuration* level. At this level, the refinement of the knowledge is likely to have produced a huge amount of information, possibly in the form of millions of lines of code. The vision has turned into a solution, which automates a specific part of the business operations.

Traceability

Now let's consider the second part of the definition of software engineering, where we can find three concepts: *traceability*, *item of information*, and *abstraction levels*. These concepts connect the two parts of the definition in the following two ways:

- The concept of *knowledge* in the first part of the definition relates to the concept of *items of information* in the second part of the definition. An item of information "realizes" a piece of knowledge.[2] The concept of *item of information* is expressed throughout the book by the term "artifact."
- The concept of *abstraction levels* connects the two definitions by the means of *traceability*. Indeed, intuitively, we expect that the information items representing one abstraction level can somehow be connected to the information items of the next abstraction level. We have to view these

2. "Realization" is a word that will be used extensively throughout the text. Its meaning is defined by its usage within the UML language (e.g., a use case storyboard realizes a use case). A general definition of the verb, which accurately reflects the sense of its usage in this book, is "to bring into concrete existence."

connections as relationships of some sort that we have to formalize. Traceability is all about defining formalized relationships between the *items of information*.

Figure 1-2 shows information items of successive abstraction levels connected by arrows. These arrows define a relationship between two information items, which in turn represent the realization of knowledge at two successive levels of abstraction. These relationships represent the traceability of the information. What is very important to understand is that these relationships are not just arrows on a sheet of paper. Each of these relationships (i.e., each arrow) is "realized" by the means of a tool, method, or technique. Hence, each relationship represents a specific tool, method, or technique, which is defined by the software development process selected. The concept of traceability is central to this book, as through the chapters the discussion clearly presents the input and output artifacts of each stage of the process, and how they trace to each other by describing the thought process of going from the inputs to the outputs.

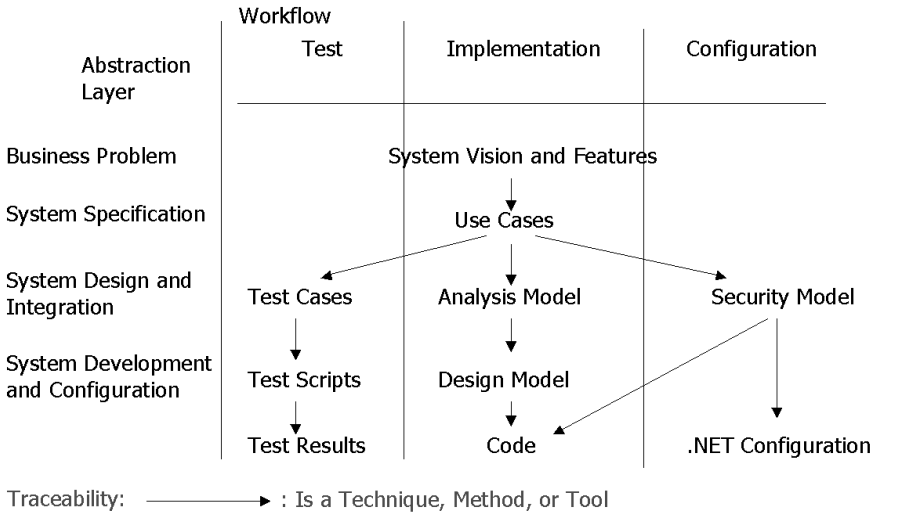

Figure 1-2: Traceability.

Practical Implications

The practical implications of the software engineering definition and the discussion above is that to achieve this vision we need to *implement* traceability between all the information items that will constitute a software automation solution. The means to implement traceability are

- Abiding by a clearly defined *process*.
- Describing a *technique* or *method* (e.g., use case analysis method or the technique of role-based security matrix, which you will see in Chapter 8).
- Using a *tool* (e.g., Rational XDE with Microsoft Visual Studio .NET) that enables automatic synchronization (hence tracing) of code to model, and the other way around.

By virtue of knowing and understanding these techniques, methods, or tools, you can relate all the information items produced in the course of a software development effort, with the ultimate goal being to demonstrate that the system does what it is meant to do. In other words, you can trace each and every feature the client wants, down to the system code and configuration. This ability defines a measure of functional coverage. Collectively, all this defines a measure of the quality of the system, which can now be measured in a very mechanical way, relying on the traceability of the items of information. And this is exactly the concept and vision of software engineering covered by the definition above.

The most prominent result of implementing traceability is the improved ability to execute impact analysis. It is easier and more accurate to identify the changes on any particular part of the system that will result from modifications in any level of system specification. In other words, when changing one artifact you can easily trace which other artifact(s) will be impacted downstream, down to the system code and configuration. Chapter 10 presents a case study that will demonstrate how traceability is put to good use in a situation where changes have to be made in the system specification, and how it preserves the quality of the solution.

Figure 1-3 completely covers the concepts found in the definition of software engineering. This is a schematic view in order to illustrate the concepts, while Figures 1-6 to 1-9 present a more precise and complete view, in depicting the practical process presented in this book. This process leverages

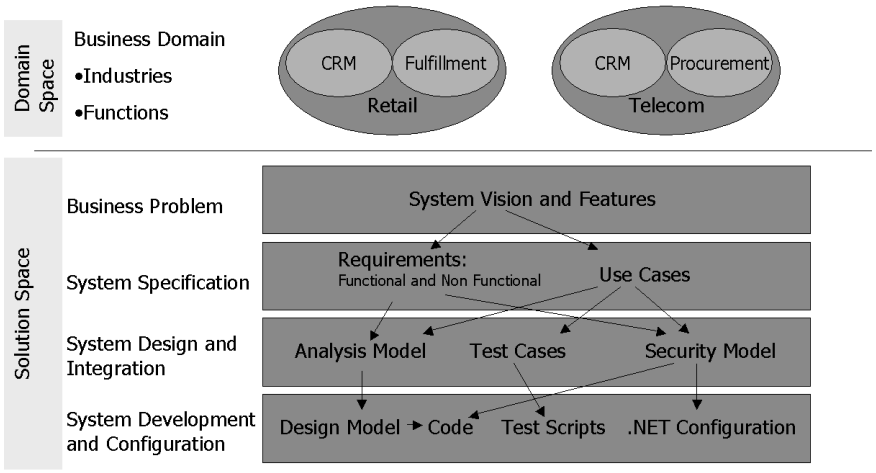

Figure 1-3: Abstraction levels and traceability.

the practical implications of the software engineering vision by appropriately using process and methods but also simple techniques and the capabilities of available tools.

The Case Study

The rest of this book will go through a case study of specifying, analyzing, designing, implementing, and testing a sample Web application for .NET. The application is a fictitious on-line bookstore named BooksREasy. Chapter 3 describes the features of this solution. There are six models produced as part of the process: business model, use case model, user experience (Ux) model, analysis model, design and implementation models, and database model.

The approach is to present you with a very simple but complete and comprehensive case study. The bookstore supports the whole buying experience from registration to login, browsing the catalogue, adding to the shopping cart, and checking out by paying with dummy credit cards. Using a simple case study ensures that all the artifacts produced during refinement are simple, and

it is easier for you to review and understand the whole solution in a holistic way.

At the same time, you get a complete working system that can serve as the basis (code and model) for your own solutions. Note that this sample is itself an offspring of the Pearl Circle Online Auction, the companion sample of the Rational XDE tool. The example is simple enough that you can conceive simple extensions to help you exercise the concepts and approach presented.

This book will also be useful as a tutorial for the design of the features you want to implement. It will take you through the refinement of one feature: the registration of a new user. Because every feature can follow the same refinement process, there is no need for a more complex example. This refinement process reviews the lifecycle of a single iteration of the software development process—specifically, the iteration that supports the development of the user registration feature.

Finally, you should also note that the emphasis is on software engineering, which is why I will not show extensive code examples. You can use a tool like Rational XDE to generate a very precise code structure, specifically targeted for .NET. To achieve this, the tool needs to understand the underlying technological platform. Code examples will appear where the modeling technique is tightly related to the .NET technology. This is the case in the persistence framework, where the design takes advantage of the .NET DataSets. It is also the case for the design of role-based security, which takes advantage of .NET's corresponding security feature. All the code is available for download at `www.booksreasy.com`. Finally, bear in mind that this is not a book on .NET; I refer you to the numerous titles covering all aspects of the .NET technology.

The Process

To restate and summarize in a process-like perspective the vision introduced earlier, this book is really about demonstrating the process of the refinement of knowledge from a business automation vision, expressed concisely in a few pages of text, to a whole application, with all the artifacts involved in the process. It also demonstrates methods and techniques for ensuring the traceability of each and every artifact produced in the process. In this perspective, it will achieve the vision of the definition of software engineering put forward at the beginning of this chapter.

The book structure itself supports the underlying process. This process is a lightweight version of the Rational Unified Process (RUP), focusing on the development-centric workflows. A very good and practical book on this process is *The Rational Unified Process: An Introduction.* Though it is primarily the description of a software development process, RUP is also a tool that can be used by all members of a team involved in a software development project.

A core RUP concept is the notion of Guidelines. You can think of these as the tools and methods/techniques mentioned earlier in the definition of software engineering. In this way, RUP covers all aspects of the practical implications of this definition: process, tools, and methods/techniques. Rational has also developed a simplified version of RUP, the Rational Unified Process .NET Developers Configuration (RNDC), which also focuses on development-centric workflows. RNDC effectively complements the integration of the Rational XDE modeling tool with Visual Studio .NET, as it also references Microsoft Developer Network (MSDN) resources.

The process that I will present is inspired by practical considerations and reflects a hands-on, straightforward approach to development, while using a necessary and sufficient level of rigor to achieve the software engineering vision presented earlier. Consequently, similar to RUP but in a concise way, this book will cover all aspects of the practical implications of this vision. Figures 1-6 to 1-9 present a graphical view of the process and cover the system specification, system analysis and design, system implementation, and system testing.

An Iterative Process

As for the Rational Unified Process, the process presented integrates an iterative approach to software development. Table 1-1 shows the four phases of a RUP project lifecycle.

RUP also specifies that each phase is composed of one or more development lifecycles (each lifecycle defining one iteration). Table 1-2 shows the activities of a development lifecycle.

Figure 1-4 presents the way the project and development lifecycles relate to each other (this diagram can also be found in *The Rational Unified Process: An Introduction*).

Table 1-1: The RUP project lifecycle.

Phase	*Main Activities*
Inception	Business modeling
	Requirements capture
Elaboration	Requirements refinement
	System analysis and design
Construction	Refinement of the design
	Implementation
Transition	Test
	Deployment
	Configuration and change management

Figure 1-4 makes clear that for each phase of the project lifecycle, all activities of a development lifecycle will take place. Two development lifecycles from two different project phases will differ only in the relative proportion of the four development lifecycle activities. Figure 1-5 (which is also from *The Rational Unified Process: An Introduction*) represents this fact clearly. The core disciplines marked in bold will be the focus of this discussion: Business Modeling (marked in italics because it is not directly related to

Table 1-2: RUP lifecycle activities.

Activity	*Main Artifacts*
Requirements Analysis (Referred to as "System Specification" in this book)	Business model Use case model
Analysis and Design	Analysis model
	Design model
	Data model
	Role-based security matrix
Coding and Unit Testing	Implementation model
	Unit stubs
	Unit tests
Integration and Testing	Test cases
	Test coverage matrix

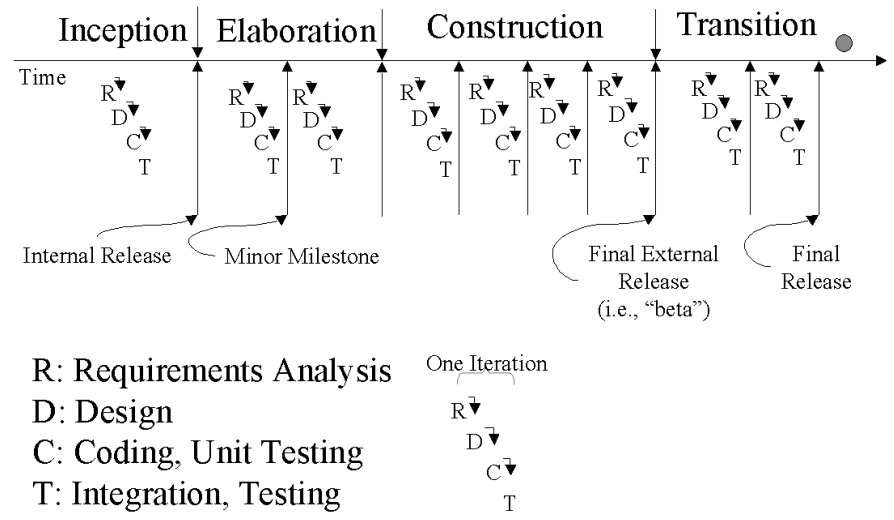

Figure 1-4: Project and development lifecycles in an iterative process.

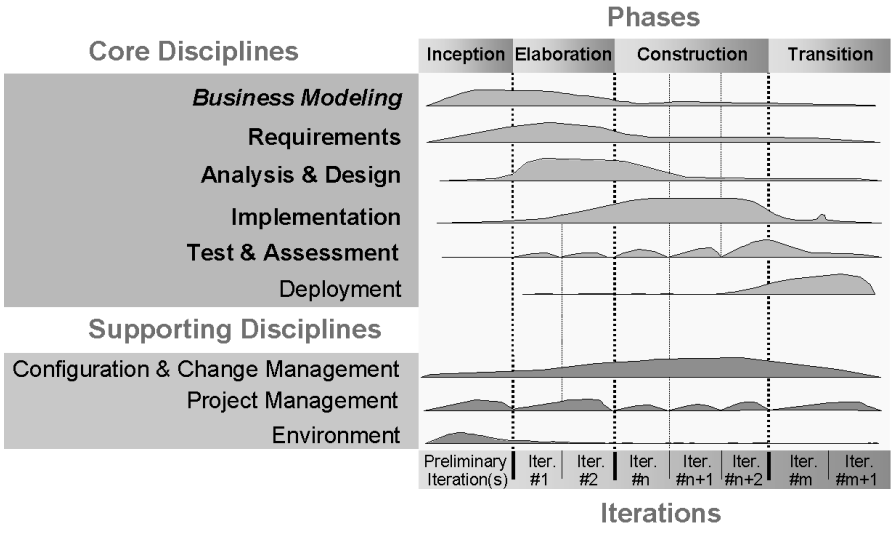

Figure 1-5: Distribution of effort for each activity during the project lifecycle.

development), Requirements, Analysis and Design, Implementation, and Test and Assessment.

The above discussion is a simplification of what is presented in RUP; I strongly encourage you to read *The Rational Unified Process: An Introduction* in order to get an accurate and complete understanding of these concepts.

This book focuses on development-centric activities, so I will not discuss the complete project lifecycle. Instead, I will consider a situation where a complete development lifecycle produces (from specification to code) one specific feature of the solution: the on-line registration of a new user. You can imagine yourself to be involved in a project where you will be playing all developer-centric roles through the project lifecycle.

The process diagrams that are pictured in the following sections present an artifact-centric vision of the process, depicting the dependencies between the artifacts. As such, it is important to understand that they do not imply a waterfall approach to software development. Any cohesive set of artifacts is likely to be used within a specific iteration (development lifecycle), though each of them may be produced and/or refined in different project phases (project lifecycle).

A practical implication of the above discussion is project planning, where you need to plan for architectural design activities even in the early stages of the project lifecycle, while focusing on system specification. Typically, the architectural design at this stage will involve the investigation and development of the architecturally significant mechanisms (as described in Chapter 6). Indeed, you do not need to have fully specified use cases in order to be able to design these mechanisms, which define the application infrastructure.

From the previous discussion it is clear that a complete development lifecycle for one development iteration would entail the following four areas of activities: system specification, system analysis and design, system coding, and system integration and testing. Because the focus of this book is on the functional aspects of system development, it will focus on the system specification and system design activities, while it skims through the system coding and system integration (and does not cover other activities like system deployment). The specific integration of the testing activity in Chapter 9 focuses on the functional aspects of testing and covers an area that is often left out by developers. Indeed, Chapter 9 will first present the correct approach to testing, and then show how simple techniques can produce the correct test

designs that will warrant the tracing of each test to a system use case (and permit a consistency check of the test coverage).

In the following sections I will give an overview of the various parts of the process, while presenting the relevant chapter where each part is discussed.

System Specification

The overall structure of the book defines two parts of the process, which represent two clearly separated concerns. In Part I, "System Specification," users are an integral part of the software development process and contribute to the definition of the various artifacts produced: business models and the vision document (obviously), but also use cases and the user experience model. They also decide the scope of the iterations. Part I also describes business process modeling activities, which are not an integral part of system specification, but present a complete and comprehensive approach to software engineering. Figure 1-6 presents an overview of the business modeling process, while Figure 1-7 presents the system specification part of the overall process. In these diagrams, as well as in Figures 1-8 and 1-9, all elements represent artifacts.

It is important to note that the process is defined by the artifacts and the activities that produce them. A solid line from an artifact A to an artifact B

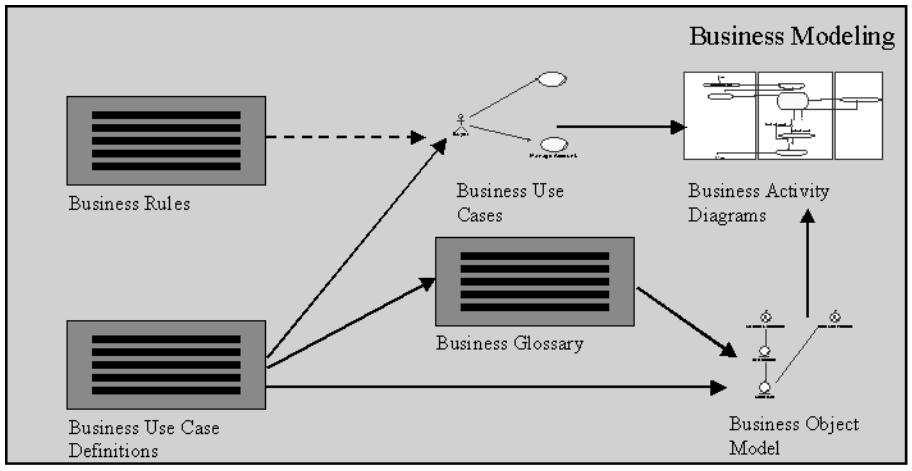

Figure 1-6: A practical process: Business modeling.

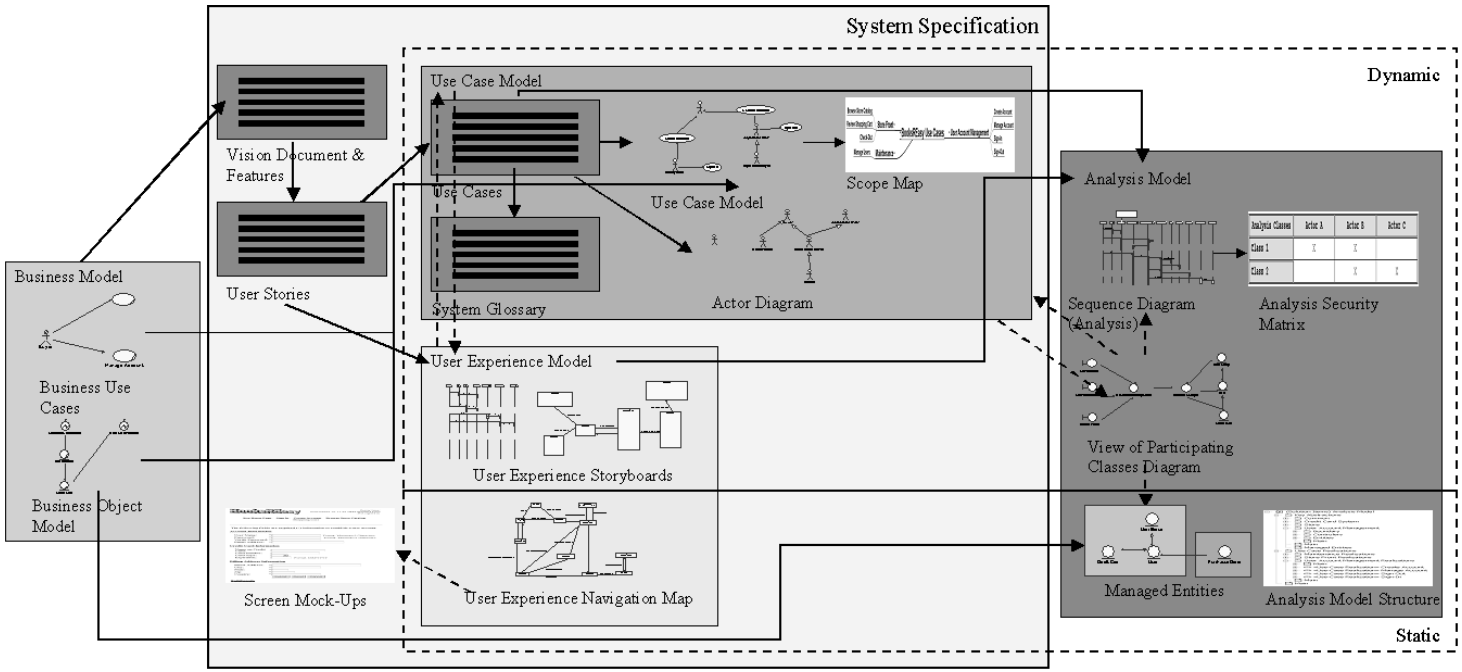

Figure 1-7: A practical process: System specification.

indicates that artifact A is a necessary input for creating artifact B. A dashed line indicates either an optional input artifact, or a situation where two artifacts (or sets of artifacts) need to be developed in parallel, each artifact feeding back in order to refine the other artifact (or sets of artifacts).

Chapter 2 considers business modeling activities, which result in the production of the artifacts presented in Figure 1-6, with their internal dependencies. This is the only chapter that is not directly relevant to development activities. As explained at the beginning of this chapter, business modeling activities and associated knowledge are not directly part of the solution space, which is my main concern in this book. Nevertheless, in this book, I am considering software development projects that aim to implement business automation solutions. The business requirements are thus the driving force of the system specification.

It is important to understand the starting point of the system development work, in order to put the rest of the process in context. This will help you gain a holistic vision of software engineering and also enable you to better communicate with the users who contribute to the system specification and who are focusing on the business implications of the solution you will be implementing.

There is at least one situation where business modeling occurs alongside system development. This happens when you do not yet have a model of the business for the specific function that is within the scope of the business automation solution. In that situation, you should also analyze the business function in order to model it, understand it, and possibly improve it, along with developing the corresponding automation solution. At the same time, this activity will bring a better understanding of the environment and the business in which the software will function. This will help create the best possible software for automating that business function.

Another important reason why it is useful to understand the business models is because they are documenting the glossary of the business. You should use this glossary when specifying the solution. Also, business modeling will produce the business object model, which will be used as the starting point for the static class model of the solution. Finally, the business model, along with the system requirements, will help identify an appropriate set of use cases. Because of the importance of the business glossary, business object model, and business rules artifacts for the rest of the process, teams should always carry out the activities involved in producing them, even in situations where there is no formal business-modeling phase defined.

Chapter 3 looks at how to specify the system requirements, starting with the system vision and features, as presented in Figure 1-7. This is the business problem definition, which is the highest level of abstraction in the solution space. It defines the scope of the business automation solution. We then refine the knowledge of the system vision and features into a list of requirements and use case definitions. Similar to RUP, the process followed is use case driven. As such, use case definition and use case analysis are both critical to the success of the solution.

The process presented defines two levels of use case descriptions. The first level is in the form of stories described by the users themselves: user stories. This is the same approach defined in Extreme Programming (XP), with the slight difference that in this book the user stories serve as the basis for the subsequent specification of the use cases. Read *EXtreme Programming EXplained* for more insight on that approach. After applying this approach to a few projects, I found it to be a very powerful and effective way to:

- Involve the users and give them a sense of ownership of the system.
- Get a head start on use case definitions.
- Serve as the basis for user experience discussions.
- Avoid a kind of paralysis that happens when users are not familiar with use cases and communication is difficult, especially when you interview them to get the information you need for the use case definitions.

The name of the artifact itself is carefully chosen to avoid any misconception from the users who are writing the use cases, thus giving you the leeway to mold the definitions into proper use cases, without any objections from the users. It is very important to note that the scope of this activity shall remain limited. Users are invited to write up, in simple terms and format, what they know of the related processes. Keep the schedule tight so users do not have the time to go into too much detail.

The user stories are used for two things: first for the use case model, then for the user experience model. From the user stories define a first draft of a use case model and use case descriptions. Then, in a second level of use case descriptions, the use cases are reworked with the users to a format more suitable for analysis. The diagram of actor relationships is carefully devised, as it is of critical importance to have a sensible model for the role-based security defi-

nition. The use case model is then produced. One thing to remember is that use case definitions and the use case model are the most important products of the system specification stage. In particular, use case definitions will be used as input to define test scenarios, driving the test artifacts (see further down in Figure 1-9).

The use case model is also the basic input for creating the so-called scope map, which is a diagram of all the identified use cases, organized in a hierarchy centered around business functions. This diagram helps prioritize the system features and allocate them into iterations. Note that the discussion revolves around functional requirements. Other types of requirements cover availability, reliability, manageability, performance, scalability, maintainability, and security. These nonfunctional requirements are also important and must be captured, but this book will not cover these concerns, in order to keep a limited and manageable functional scope that you can easily comprehend.

In Chapter 4 the user stories also serve as input to the definition of the user experience model, an activity that equally involves the users. This part is heavily inspired by RUP, though the discussion is kept concise and to the point. As presented in Figure 1-7, the screens identified from the user stories, and subsequently by reviewing the use cases, are organized into a series of navigation maps, which are diagrams of user experience model elements that represent the static relationships between the screens. The user stories, and subsequently the use cases, help create use case storyboards that describe the dynamics of the user interaction with the user experience model elements identified previously. You could also create screen mock-ups to attach to the screen definitions of the user experience model. The dashed arrows between the use case model and the user experience model indicate that you should use the information produced in one model to refine the artifacts of the other.

System Analysis and Design

In Part II, "System Analysis and Design," the software development team must find enough detail in the information produced in Part I to produce the rest of the artifacts, which will ultimately lead to the completed system. The development teams can use the artifacts produced during system specification as contracts for the implementation of the system. It is very important to understand that the separation of concerns does not mean a lack of communication.

And as much as the process is iterative, it is also very interactive. If attention and rigor have been applied in the system specification, you can expect to have very few incidents in the design and implementation stages where specifications are vague or inaccurate. Nevertheless, these incidents may occur, and in these situations you must go back to the corresponding system specification and review it with the users. This is also where traceability will play a major role. Each element of information at a specific level in the design should easily be traced to a specific element of information in the specification.

The next step in the process, presented in Chapter 5, is the development of the analysis model. As explained earlier, the overall process is divided into two main stages, system specification and system analysis and design (see Figure 1-8).

The analysis model, on the left part of Figure 1-8, is the first step in system design. It is an intermediary step that decouples the *What* of the system specification and the *How* of the system detail design, where the system will be modeled to the level of detail that will represent actual implementation classes. With the correct toolset, the design model can generate code in the appropriate language and technology. The main objective of the analysis model is to avoid the analysis paralysis phenomenon, which has been widely observed with teams using object-oriented analysis and design (OOAD) methods and UML for system design. Important points to keep in mind for the analysis model are the following:

- It serves as a decoupling of knowledge abstraction, between system specification and detailed design; that is, the analysis model can be seen as the output of system specification and the input to the design model.
- It is a high-level design model, which can be produced by either an analyst or a designer, without any emphasis on details. It's there to serve as a whiteboard to get a first cut of the responsibilities and attributes of the classes of the system. Still, it has to be designed with rigor, as we expect to identify most of the classes of the solution with their relationships.
- It serves as the glue between the use case model and the user experience model.
- Finally, it helps to control the completeness of the use case descriptions and model, specifically through the development of the View Of Participating Classes (VOPC) diagram. The central parts of the analysis model

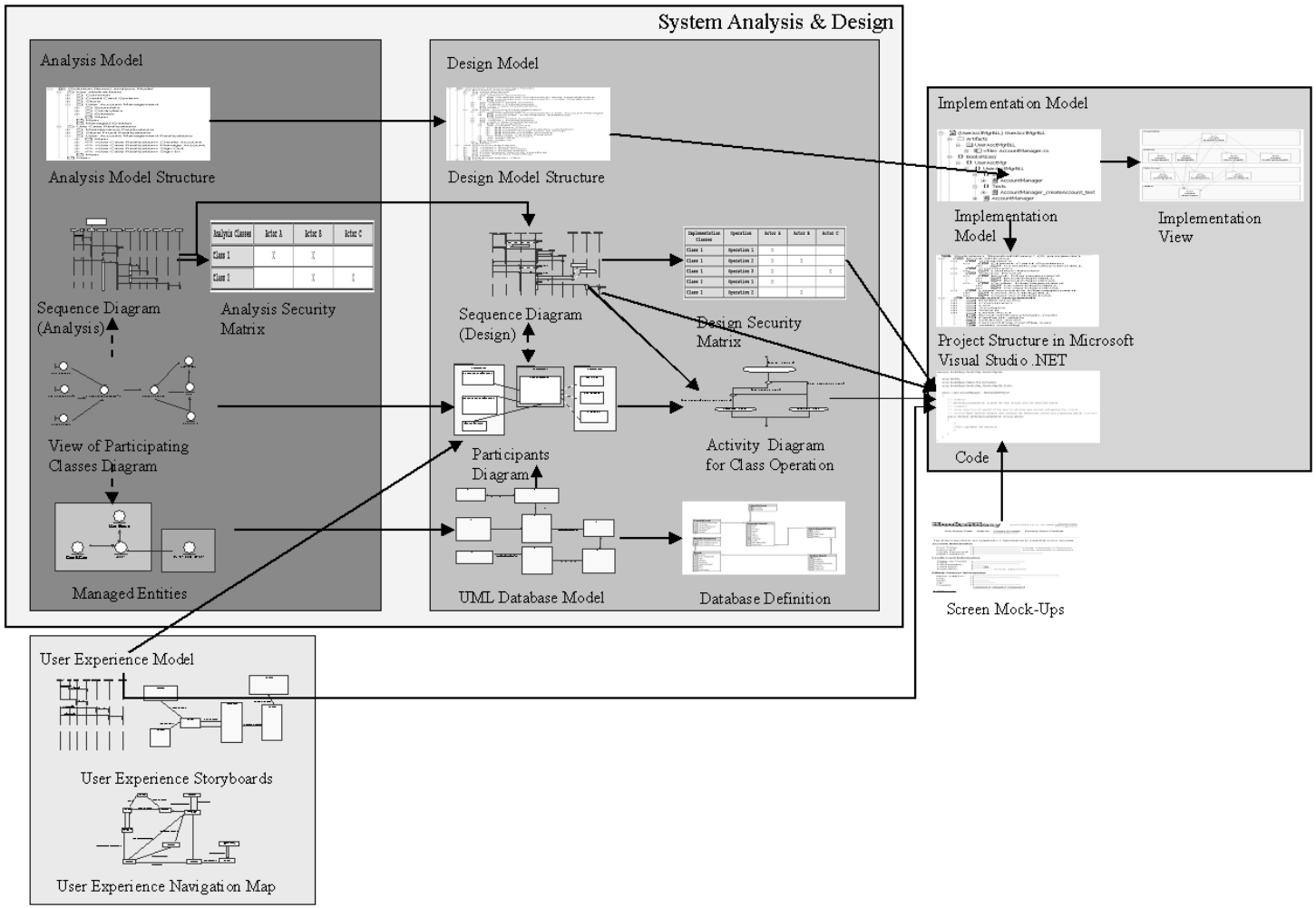

Figure 1-8: A practical process: System analysis and design.

are the VOPC diagram and the sequence diagrams that will help to make the first step in the detailed design.

The detailed design, pictured in the middle of Figure 1-8, is the next logical step in the process and is presented in Chapter 6. This part of the process is critical because it could lead to the generation of code. It is important to understand that, to achieve the software engineering vision defined earlier, you need to produce a complete design model. Along with analysis paralysis, this is the other plague of teams that use OOAD. Too often they stop short of creating a complete design model. The key input to the design model is the VOPC and sequence diagrams from the analysis model. The classes represented in these diagrams will all be part of the design model, with other classes possibly added to reflect the implementation details of the underlying technology used. In this book I will be using the .NET DataSets feature to implement data access.

Part of the detailed design stage is the development of a database model, which translates the key system entities relationships into a database model. Specifically, the approach will demonstrate the use of the UML Profile for Database Design. In the process presented, there is a lot of emphasis on the sequence diagrams that are considered core elements of the design model (and, for that matter, of the whole system design stage). These diagrams help identify the detail responsibilities of the implementation classes. An optional artifact in the design model is the activity diagram for class operations. These diagrams are useful to specify an algorithm or the flow of execution for a specific class operation.

In parallel, an important aspect of the design model is the model of the architecturally significant mechanisms. These are mechanisms that are of general use for the implementation of the functional features of the solution. They define the software infrastructure's application level, in contrast to its application server (.NET) or operating system (Microsoft Windows) level (which should not be represented in a model).

The final step, which relates directly to the production of code, is the implementation model (pictured on the right side of Figure 1-8). The related activities are presented in Chapter 7, which covers the practical implications of using modeling tools that integrate with integrated development environments. Specifically, I will demonstrate a set of best practices for the use of Microsoft Visual Studio .NET, from the perspective of the modeling tech-

niques applied. With the implementation model we allocate the classes into physical containers. The type of container depends on the target technology used. For the .NET platform we will assign the classes into .NET assemblies. The implementation model will show us the dependencies between the assemblies, and this, in turn, will be used to configure the solution in Microsoft Visual Studio .NET. These dependencies will also impose the build order. Notice that this approach can trace the .NET configuration of the solution back to the model of the system.

Chapters 8 and 9 are not directly related to the production of code. Chapter 8 provides a lot of insight into using the role-based security feature of the .NET framework. This feature is often not used in an optimal way (or not used at all). This chapter shows an engineered approach to the role-based security design, and you will learn a simple technique to derive the configuration of .NET role-based security. This technique demonstrates the far-reaching consequences of the definition of software development given earlier. This technique should be part of a wider software development process, because the implementation of role-based security should be an activity of every software development for the .NET platform. There are two levels of role-based security modeling: analysis and design. The related artifacts are in the form of matrixes, which are produced at the corresponding stage of the process, as presented in Figure 1-8. These are the analysis security matrix and the design security matrix.

As much as security is a core concern for every system, testing is equally important. Chapter 9 focuses on functional testing; you will learn how to derive test cases from use cases. Again, this will demonstrate how traceability can be implemented in this crucial area of software development. You will see how a simple technique can assure you that all functions of the system have been covered with a test case. This in turn defines a rigorous way to measure the quality of the system, which is exactly what the software engineering vision defined earlier is meant to achieve.

Similar to role-based security, the test definition step should be part of a wider software development process, as every system has to undergo rigorous testing. In the same chapter you will also see how to track coverage of unit tests and how developing unit tests is a preliminary step to implementing the class operations. Figure 1-9 presents an overview of the process related to the functional testing activities.

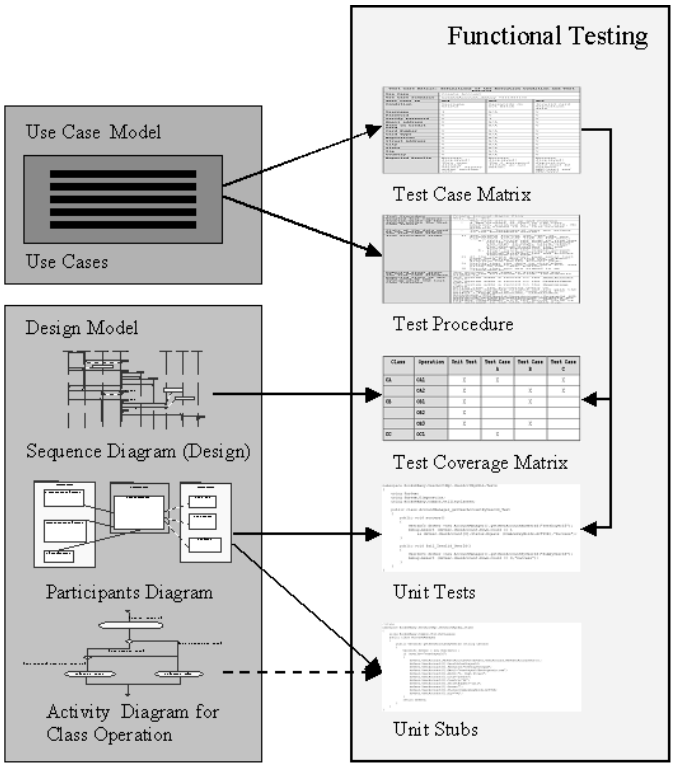

Figure 1-9: A practical process: Functional testing.

The presentation of a wider development process is not within the scope of this book, as the primary objective is to demonstrate basic software engineering principles (in particular, traceability). The process demonstrated is lightweight and hands on, covering the core objective of going from system vision to code. It is up to you to extend its use by "bolting on" every activity that you deem appropriate, as long as these activities maintain the traceability of each and every item of information between each stage of the process.

Roles in Software Development

As stated in the Preface, this book is of interest to all the software development roles, as the process presented demands the contribution of all. It is certain that each role will find a particular interest in the part where it contributes the most:

- Functional analysts, who in some organizations are named functional architects, will be interested in system specification, presented in Chapters 3 and 4. As a prerequisite they will also be interested in business modeling (Chapter 2) to understand its implications on system development. Also important for them is the development of the analysis model (Chapter 5), as well as system testing (Chapter 9). In reality, because the whole book focuses on the functional aspect of system development, functional analysts are likely to find interest in every aspect of the process.
- User interaction designers will be interested in understanding the system specification issues (Chapter 3) and developing the user experience model (Chapter 4), as well as the implications of the user interface and user experience on the overall system design.
- Software architects, or aspiring software architects, will be interested in every aspect of the process and will gain valuable insight on how elegant .NET systems can be produced by using a complete engineering approach.
- Developers/programmers will focus their interest on understanding the analysis model (Chapter 5), developing the design and implementation models (Chapters 6 and 7), modeling role-based security (Chapter 8), and testing—in particular, unit testing (Chapter 9). In reality, most of the developers will be interested in every aspect of the process, and by studying all the chapters they will gain valuable insight on object-oriented analysis and design and software development processes.
- Testers will be particularly interested in system specification (Chapters 3 and 4) and system testing (Chapter 9).

But, being hands-on, this book will benefit anyone playing any role in system development as described in the Rational Unified Process. It will bring valuable insight into what kind of deliverable is expected from their role, how

each role integrates with other roles, and what other roles do within the process. The structure of the chapters (Introduction, Approach, Case Study, and Summary) is very helpful in this perspective, as it enables each role to delve more or less deeply into the contributions of other roles. At the least, every role, including the ones mentioned above, should read the Introductions and Summaries. Other roles that do not directly contribute to the activities developed in this book, but will also benefit by reading it, include the following:

- Business analysts will benefit, especially if they have not yet had the opportunity to work within the framework of the Rational Unified Process. Note, however, that although Chapter 2 describes business modeling, it does not delve into sufficient detail to be practically used by a business analyst. It is more useful for functional analysts to help them understand the inputs to their activities.
- Project managers will gain insight into how all the roles collaborate at the technical level, as well as the practical implication on project planning of using the process.