Chapter 1

Introduction

The ICONIX process sits somewhere in between the very large Rational Unified Process (RUP) and the very small eXtreme programming approach (XP). The ICONIX process is use case driven, like the RUP, but without a lot of the overhead that the RUP brings to the table. It's also relatively small and tight, like XP, but it doesn't discard analysis and design like XP does. This process also makes streamlined use of the Unified Modeling Language (UML) while keeping a sharp focus on the traceability of requirements. And, the process stays true to Ivar Jacobson's original vision of what "use case driven" means, in that it results in concrete, specific, readily understandable use cases that a project team can actually use to drive the development effort.

The approach we follow takes the best of three methodologies that came into existence in the early 1990s. These methodologies were developed by the folks that now call themselves the "three amigos": Ivar Jacobson, Jim Rumbaugh, and Grady Booch. We use a subset of the UML, based on Doug's analysis of the three individual methodologies.

There's a quote in Chapter 32 of *The Unified Modeling Language User Guide*, written by the amigos, that says, "You can model 80 percent of most problems by using about 20 percent of the UML." However, nowhere in this book do the authors tell you which 20 percent that might be. Our subset of the UML focuses on the core set of notations that you'll need to do most of your modeling work. Within this workbook we also explain how you can use other elements of the UML and where to add them as needed.

One of our favorite quotes is, "The difference between theory and practice is that in theory, there is no difference between theory and practice, but in practice, there is." In practice, there never seems to be enough time to do modeling, analysis, and design. There's always pressure from management to jump to code, to start coding prematurely because progress on software projects tends to get measured by how much code exists. Our approach is a minimalist, streamlined approach that focuses on that area that lies in between use cases and code. Its emphasis is on what needs to happen at that point in the life cycle where you're starting out: you have a start on some use cases, and now you need to do a good analysis and design.

Our goal has been to identify a minimal yet sufficient subset of the UML (and of modeling in general) that seems generally to be necessary in order to do a good job on your software project. We've been refining our definition of "minimal yet sufficient" in this context for eight or nine years now. The approach we're telling you about in this workbook is one that has been used on hundreds of projects and has been proven to work reliably across a wide range of industries and project situations.

A Walk (Backwards) through the ICONIX Process

Figure 1-1 shows the key question that the ICONIX process aims to answer.

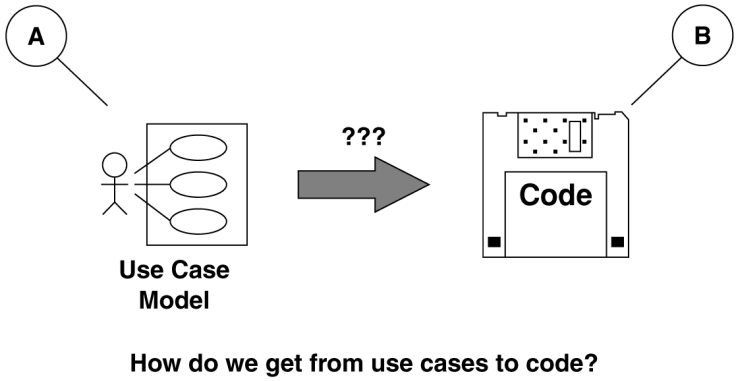


Figure 1-1: *Use Cases to Code*

What we're going to illustrate is how to get from point A to point B directly, in the shortest possible time. (Actually, we're not going to go all the way to code, but we'll take you close enough so you can taste it.) You can think of point A as representing this thought: "I have an idea of what my system has to do, and I have a start on some use cases," and point B as representing some completed, tested, debugged code that actually does what the use cases said it needed to do. In other words, the code implements the required behavior, as defined by the use cases. This book focuses on how we can get you from the fuzzy, nebulous area of "I think I want it to do something like this" to making those descriptions unambiguous, complete, and rigorous, so you can produce a good, solid architecture, a robust software design, then (by extension) nice clean code that actually implements the behavior that your users want.

We're going to work backwards from code and explain the steps to our goal. We'll explain why we think the set of steps we're going to teach is the minimal set of steps you need, yet is sufficient for most cases in closing the gap between use cases and code. Figure 1-2 shows the three assumptions we're going to make to start things off: that we've done some prototyping; that we have made some idea of what our user interface might look like; and that we might have some start in identifying the scenarios or use cases in our system.

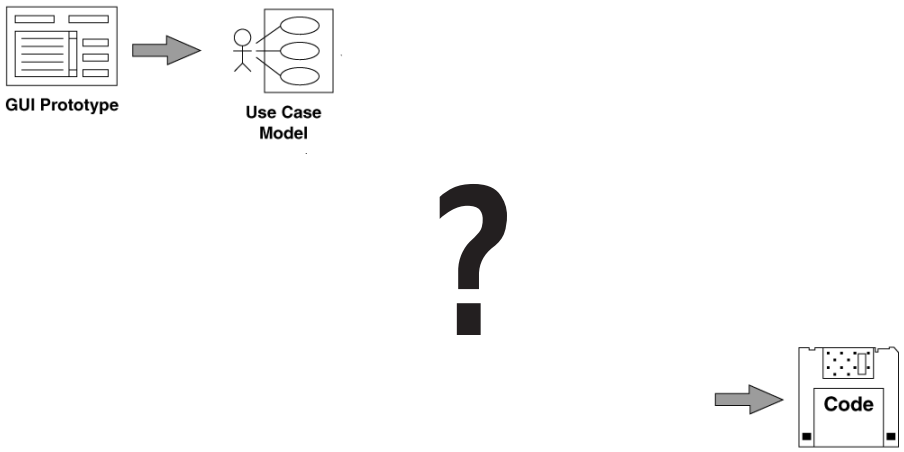


Figure 1-2: *Starting Off*

This puts us at the point where we're about to launch into analysis and design. What we want to find out is how we can get from this starting point to code. When we begin, there's only a big question mark—we have some nebulous, fuzzy ideas of what our system has to do, and we need to close this gap before we start coding.

In object-oriented systems, the structure of our code is defined by classes. So, before we write code, we'd like to know what our software classes are going to be. To do this, we need one or more class diagrams that show the classes in the system. On each of these classes, we need a complete set of attributes, which are the data members contained in the classes, and operations, which define what the software functions are. In other words, we need to have all our software functions identified, and we need to make sure we have the data those functions require to do their job.

We'll need to show how those classes encapsulate that data and those functions. We show how our classes are organized and how they relate to each other on class diagrams. We'll use the UML class diagram as the vehicle to display this information. Ultimately, what we want to get to is a set of very detailed design-level class diagrams. By **design-level**, we mean a level of detail where the class diagram is very much a template for the actual code of the system—it shows exactly how your code is going to be organized.

Figure 1-3 shows that class diagrams are the step before code, and there is a design-level diagram that maps one-to-one from classes on your diagram to classes in your source code. But there's still a gap. Instead of going from use cases to code, now we need to get from use cases to design-level class diagrams.

One of the hardest things to do in object-oriented software development is **behavior allocation**, which involves making decisions for every software function that you're going to build. For each function, you have to decide which class in your software design should be the class that contains it. We need to allocate all the behavior of the system—every software function needs to be allocated into the set of classes that we're designing.

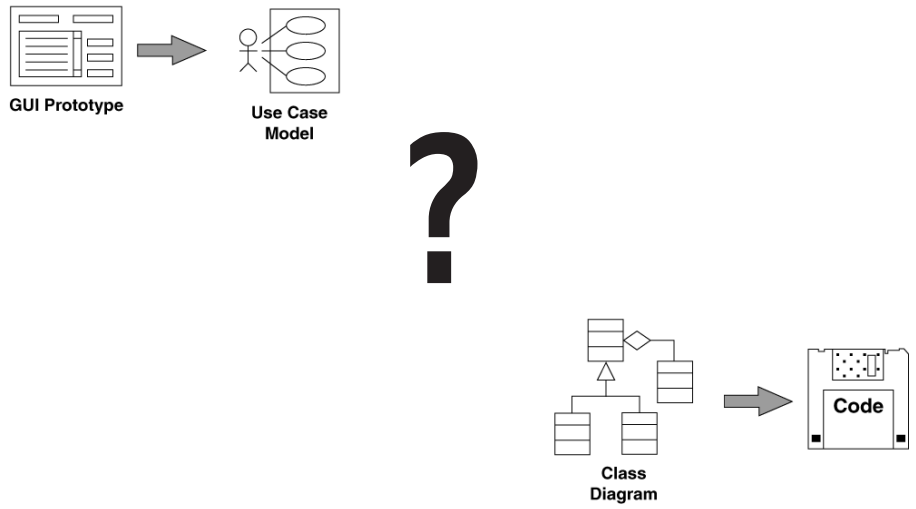


Figure 1-3: *Class Diagrams Map Out the Structure of the Code*

One UML diagram that's extremely useful in this area is the sequence diagram. This diagram is an ideal vehicle to help you make these behavior allocation decisions. Sequence diagrams are done on a per-scenario basis: for every scenario in our system, we'll draw a sequence diagram that shows us which object is responsible for which function in our code. The sequence diagram shows how runtime object instances communicate by passing messages. Each message invokes a software function on the object that receives the message. This is why it's an ideal diagram for visualizing behavior allocation.

Figure 1-4 shows that the gap between use cases and code is getting smaller as we continue to work backwards. Now, we need to get from use cases to sequence diagrams.

We'll make our decisions about allocating behavior to our classes as we draw the sequence diagrams. That's going to put the operations on the software classes. When you use a visual modeling tool such as Rational Rose or GDPro, as you draw the message arrows on the sequence diagrams, you're actually physically assigning operations to the classes on the class diagrams. The tool enforces the fact that behavior allocation happens from the sequence diagram. As you're drawing the sequence diagram, the classes on the class diagram get populated with operations.

So, the trick is to get from use cases to sequence diagrams. This is a non-trivial problem in most cases because the use cases present a requirements-level view of the system, and the sequence diagram is a very detailed design view. This is where our approach is different from the other approaches on the market today. Most approaches talk about use cases and sequence diagrams but don't address how to get across the gap between the fuzzy use cases and a code-like level of detail on the sequence diagrams. Getting across this gap between *what* and *how* is the central aspect of the ICONIX process.

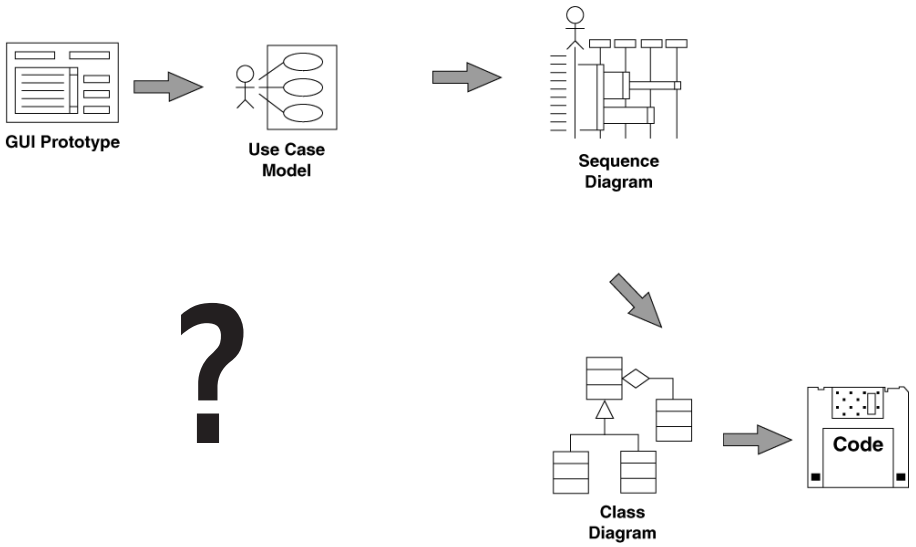


Figure 1-4: *Sequence Diagrams Help Us Allocate Operations (Behavior) to Classes*

What we're going to do now is close the gap between the fuzzy, nebulous use case and the very detailed and precise sequence diagram with another kind of diagram called a **robustness diagram**. The robustness diagram sits in the gap between requirements and detailed design; it will help make getting from the use cases to the sequence diagrams easier.

If you've been looking at UML literature, the robustness diagram was originally only partially included in the UML. It originated in Ivar Jacobson's work and got included in the UML standard as an appendage. This has to do with the history and the sequence of how Booch, Rumbaugh, and Jacobson got together and merged their methodologies, as opposed to the relative importance of the diagram in modeling.

Across the top of a sequence diagram is a set of objects that are going to be participating in a given scenario. One of the things we have to do before we can get to a sequence diagram is to have a first guess as to which objects will be participating in that scenario. It also helps if we have a guess as to what software functions we'll be performing in the scenario. While we do the sequence diagram, we'll be thinking about mapping the set of functions that will accomplish the desired behavior onto that set of objects that participate in the scenario.

It helps a great deal to have a good idea about the objects that we'll need and the functions that those objects will need to perform. When you do it the second time, it's a lot more accurate than when you take a first guess at it. The process that we're following, which is essentially Ivar Jacobson's process as described in his Objectory work, is a process that incorporates a first guess, or preliminary design, the results of which appear on what we call a robustness diagram. We refine that first guess into a detailed design on the sequence diagram. So, we'll do a sequence diagram for each scenario that we're going to build.

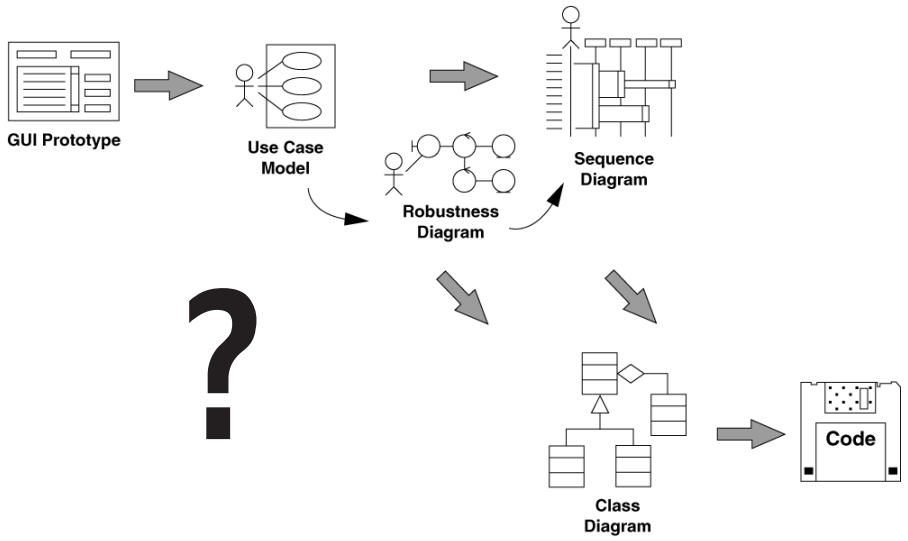


Figure 1-5: *Robustness Diagrams Close the Gap Between Requirements and Detailed Design*

Figure 1-5 shows that we're adding a diagram to our subset of UML. The robustness diagram was described in the original UML specs, but its definition was in an extra document called *Objectory Process-Specific Extensions*. What we've found over the past ten years is that it's very difficult to get from use cases to sequence diagrams without this technique. Using the robustness diagram helps avoid the common problem of project teams thrashing around with use cases and not really getting anywhere towards their software design. If you incorporate this step, it will make this process and your project much easier. We didn't invent robustness analysis, but we're trying to make sure it doesn't get forgotten. Robustness analysis has proven to be an invaluable aid in getting across the gap between requirements and design.

Robustness analysis sits right in the gap between what the system has to do and how it's actually going to accomplish this task. While we're crossing this gap, there are actually several different activities that are going on concurrently. First, we're going to be discovering objects that we forgot when we took our first guess at what objects we had in the system. We can also add the attributes onto our classes as we trace data flow on the robustness diagrams. Another important thing we'll do is update and refine the text of the use case as we work through this diagram.

We still have a question mark, though. That question mark relates to the comment we just made about discovering the objects that we forgot when we took our first guess. This implies that we're going to take a first guess at some point.

There's a magic phrase that we use to help teach people how to write use cases successfully: *Describe system usage in the context of the object model*. The first thing this means is that we're not talking, in this book, about writing fuzzy, abstract and vague, ambiguous use cases that don't have enough detail in them from which to produce a software design. We're going to teach you to write use cases that are very explicit, precise, and unambiguous. We have a very specific goal in mind when discussing use cases: we want to drive the software design from them. Many books on use cases take a different perspective, using use cases as more of

an abstract requirements exploration technique. Our approach is different because our goals are different. Remember, our mission is to help you get from use cases to code.

We'll start out with something called a domain model, which is a kind of glossary of the main abstractions—in other words, the most important nouns that are in our problem space (our problem domain). In the term **domain model**, the word "domain" comes from the idea of the problem domain. For example, if our problem domain is electronic commerce—as it is in the workbook, amazingly enough—we'll probably have a domain object like a catalog or a purchase order. We're going to call these nouns that belong to our problem space **domain objects**, and we're going to produce, at the very beginning of our analysis and design activities, something called a domain model, which lays all these domain objects out on one big UML class diagram.

On our robustness diagrams, we're also going to use something called **boundary objects**. Among the boundary objects, we find things like the screens of the system. In the text of our use cases, we want to explicitly reference both domain objects and boundary objects. We'll write about such things as how the users interact with the screens and how those screens interact with the domain objects, which often have some mapping onto a database that may sit behind the OO part of our system. Our use case text will get a lot more specific and a lot less ambiguous if we follow this guideline of describing how the system is used in the context of the object model as it evolves.

During domain modeling, we want to identify the most important set of abstractions that describe the problem space, or the problem domain of the system, that we need to build. For this task, we'll follow the methodology Jim Rumbaugh developed: the Object Modeling Technique (OMT), which is a very thorough treatment of some useful techniques for helping us do this domain model.

One difference between our approach and some of the other use case–oriented approaches you might run across is that we insist on starting the whole process with domain modeling. In writing our use cases against the set of nouns in the domain model, thus using that domain model as a glossary, we can unambiguously define a set of terms that we can reference within our use case text. This approach proves to be quite useful, especially when you're working in a team environment where there are multiple groups of people that are trying to describe scenarios in different parts of the system. If you get closure and agreement on what the important nouns in the system are, you eliminate whole layers of ambiguity in the use case models. For example, this enables you to be clear on what a purchase order is, what a line item is, and what a shopping cart is. All those things are clear from the beginning, due to the fact that we've defined a glossary of terms before we start writing our use cases.

In terms of the UML, the domain model is basically a class diagram, so it's the same kind of diagram as our design-level class diagram. Generally, on the domain model, we suppress quite a bit of detail; in particular, we don't show the attributes and the operations on the classes. The domain model is more of a global summary-level class diagram. In fact, it's a first guess at a class diagram, focusing entirely on the problem domain of the system we're building. We take this first guess at our class diagram, and then we work through all the details of our use cases and refine our view of the system. As we work through the scenarios, the first-guess class diagram evolves into our detailed static model for the system.

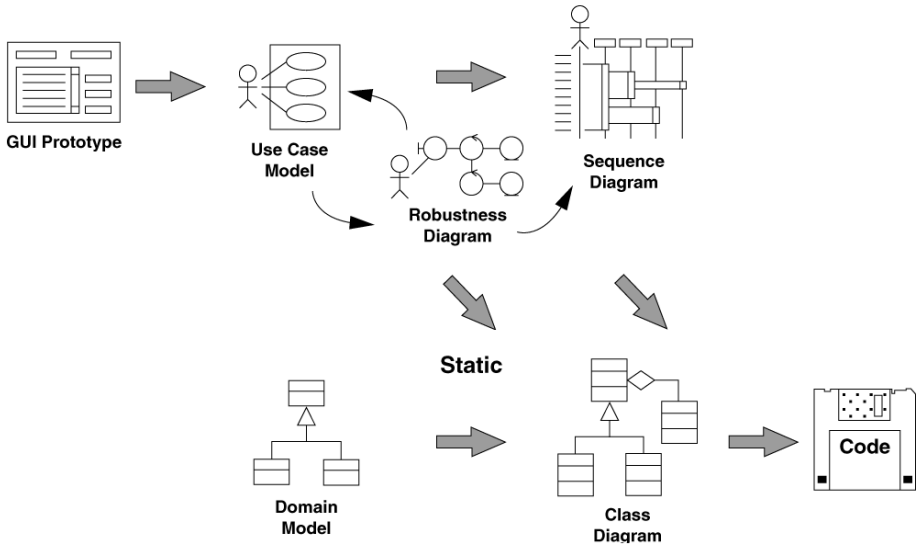


Figure 1-6: *Referencing Domain Objects by Name Removes Ambiguity from the Use Cases*

As you can see in Figure 1-6, we now have a fairly complete picture, with no big gaps in it, that helps us get from use cases and prototypes over on the left side to design-level class diagrams and source code over on the right side.

Note that we're using a very streamlined approach. We're only using four different kinds of UML diagrams. That's four out of a set of nine different kinds of diagrams that make up the UML. Generally, for most projects, most of the time you can do most of your work using less than half of the UML. Limiting your focus to this core subset of diagrams will make a significant impact on your learning curve as you learn how to do modeling with UML.

We're going to start off with the domain model, which is an analysis-level class diagram, as our first guess at the static structure of the system. We're going to continuously refine and add detail to this model, with the ultimate result being our detailed design. The class diagram, which is in the bottom half of Figure 1-6, is a static description of how the code is organized, whereas the use cases are a dynamic description of the runtime behavior.

We'll take the first guess at our static model, and then we'll spend most of our time working through use case after use case. Every time we work through a use case, we'll add some detail to the class diagram. After we work through all the scenarios that the system has to support, add in all the detail needed to make all those scenarios happen, and review what we've done a couple of times, we should have a design that meets the requirements, and we'll be well positioned to write code.

Figure 1-7 shows the "big picture" for the ICONIX process. This figure appears on the first page of every chapter in our book *Use Case Driven Object Modeling with UML*. The picture has two parts to it: The top part is the dynamic model, which describes behavior, and the bottom part is the static model, which describes structure.

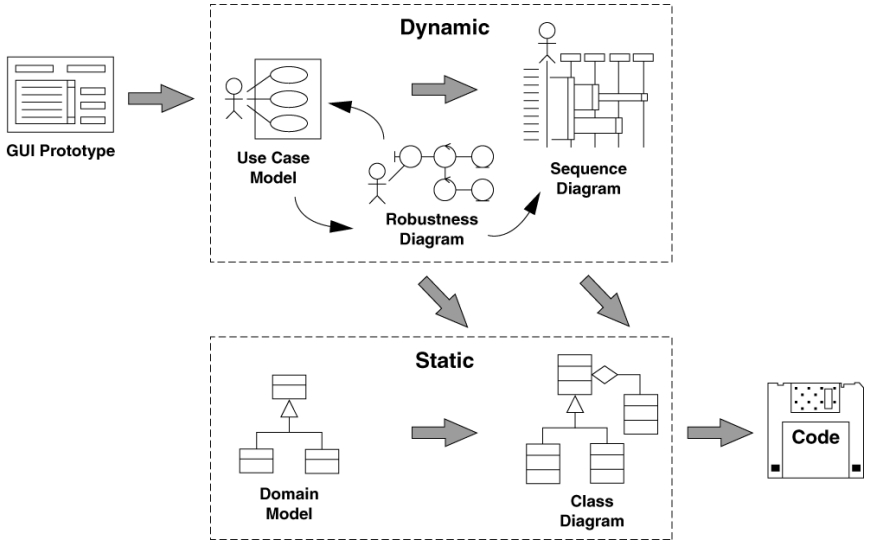


Figure 1-7: *The ICONIX Process—A Streamlined Approach to UML Modeling*

We might start with some prototypes, or perhaps simple line drawings of our screens. Then, after getting some assurance from users that we're on the right track, we can work from this beginning to identify use cases on our use case diagram, which shows all the scenarios that the system has to perform. Then we write the text of our use cases. We refine the use case text during robustness analysis. It's important to try to get the text stabilized and corrected during the preliminary design phase before moving into detailed design, which we do on sequence diagrams.

Many people complain about constantly changing requirements. Some use this as an excuse to start coding prematurely. We're willing to bet that the vast majority of these folks have never used robustness analysis, which is enormously helpful in getting those requirements stabilized.

By breaking exploration of the dynamic model into these three steps, we get two chances to review the behavior description; hopefully, by the time we've reviewed it the second time, our understanding of the required behavior is detailed and fairly stable, and we can start designing against it.

As you can see on the static part of the picture, we start with a quick first guess about objects based totally on the problem space description. We go through one long continuous refinement that's driven by our analysis of the dynamic runtime behavior of the system. We think in detail how one scenario is supposed to work, then update our class diagrams based on our improved understanding of that. Then, we go back and think more about what the behavior of the system should be.

Next, we refine our software structure accordingly. Our approach, which is derived 80 percent from Ivar Jacobson's work, is a very natural way to decompose systems along use case boundaries, and then use the results of the use case analysis to drive the object modeling forward to a level that's detailed enough to code from.

Key Features of the ICONIX Process

Figure 1-7 shows the essence of a *streamlined* approach to software development that includes a minimal set of UML diagrams, and some valuable techniques, that you can use to get from use cases to code *quickly* and *efficiently.* The approach is flexible and open; you can always elect to use other aspects of the UML to supplement the basic materials.

We'd like to point out three significant features of this approach.

First, the approach offers *streamlined usage of the UML*. The steps that we describe in the upcoming chapters represent a "minimalist" approach—they comprise the minimal set of steps that we've found to be necessary and sufficient on the road to a successful OO development project. By focusing on a subset of the large and often unwieldy UML, a project team can also head off "analysis paralysis" at the pass.

Second, the approach offers a high degree of *traceability.* At every step along the way, you refer back to the requirements in some way. There is never a point at which the process allows you to stray too far from the user's needs. Traceability also refers to the fact that you can track objects from step to step, as well, as analysis melds into design.

Third, the approach is *iterative* and *incremental*, although we might not be using these terms in the traditional sense. Multiple iterations occur between developing the domain model and identifying and analyzing the use cases. Other iterations exist, as well, as the team proceeds through the life cycle. The static model gets refined incrementally during the successive iterations through the dynamic model (composed of use cases, robustness analysis, and sequence diagrams). Please note, though, that the approach doesn't require formal milestones and a lot of bookkeeping; rather, the refinement efforts result in natural milestones as the project team gains knowledge and experience.

As we described in the Preface, we're going to demonstrate these aspects of the ICONIX process in the context of an on-line bookstore; the focus will be on the customer's view of the system.

The fact that we've been able to teach this process, with only minimal changes, over an entire decade, with it remaining useful and relevant today, is made possible because our process is based on finding the answers to some fundamentally important questions about a system. These questions include the following:

- Who are the users of the system (the actors), and what are they trying to do?
- What are the "real world" (problem domain) objects and the associations among them?
- What objects are needed for each use case?
- How do the objects collaborating within each use case interact?
- How will we handle real-time control issues?
- How are we really going to build this system on a nuts-and-bolts level?

We have yet to come across a system that doesn't need to have these basic questions answered (especially the first four questions), or one that couldn't use the techniques described in this book to help answer them using an iterative, incremental, opportunistic (when you see the answer, capture it) approach. Although the full approach presents the steps in a specific order, it's not crucial that you follow the steps in that order. Many a project has died a horrible death because of a heavy, restrictive, overly prescriptive "cement collar" process, and we are by no means proponents of this approach. What we are saying is that *missing answers to any of these questions will add a significant amount of risk to a development effort*.

Process Fundamentals

We believe that the best way to make process more attractive is to educate as many people as possible about the benefits of answering the questions we raised earlier, along with similar questions, and about the risks of failing to answer them. Building good object models is straightforward *if you keep ruthlessly focused on answering the fundamentally important questions about the system you are building and refuse to get caught up in superfluous modeling issues*. That philosophy lies at the heart of the ICONIX process.

The people who have to use the process, and management, are both customers of a software development process. We think of a process as a road map for a team to follow, a map that identifies a set of landmarks, or **milestones**, along the way to producing a quality product.

There are various paths a team can travel, depending on the capabilities and preferences of its members. But no matter which path they go down, at some point, they must reach the milestones. At these points in the process, their work becomes visible to management—during reviews of intermediate results. Passing the milestones does not guarantee a quality product, but it should greatly improve the chances.

We believe milestones for an object-oriented process should include, at a minimum, the following.

- The team has identified and described all the usage scenarios for the system it's about to build.
- The team has taken a hard look for reusable abstractions (classes) that participate in multiple scenarios.
- The team has thought about the problem domain and has identified classes that belong to that domain—in other words, the team has thought about reusability beyond just this system.
- The team has verified that all functional requirements of the system are accounted for in the design.
- The team has thought carefully about how the required system behavior gets allocated to the identified abstractions, taking into consideration good design principles such as minimizing coupling, maximizing cohesion, generality, and sufficiency, and so forth.

Beyond these milestones, there are at least four other fundamental requirements of a process.

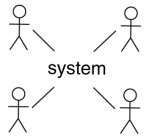
1. It has to be flexible enough to accommodate different styles and kinds of problems.
2. It has to support the way people really work (including prototyping and iterative/incremental development).
3. It needs to serve as a guide for less-experienced members of the team, helping them be as productive as possible without handcuffing more experienced members.
4. It needs to expose the precode products of a development effort to management in a reasonably standard and comprehensible form.

The Process in a Nutshell

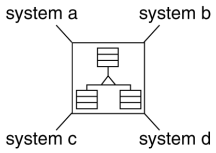
The basic steps that comprise the full ICONIX process and the associated milestones are presented in Figures 1-8 to 1-11. Note that the first three of these diagrams will appear again later in the text, to remind you where we are in the overall process. (We don't talk about implementation in this book, but we do have a chapter about implementation in the original book. Figure 1-11 is here for completeness.)

These diagrams together illustrate three key principles that underlie the process: inside-out, outside-in, and top-down, all at the same time.

1. Work inward from the user requirements.



2. Work outward from the key abstractions of the problem domain.



3. Drill down from high-level models to detailed design.

We'll reinforce these principles, in one way or another, in each subsequent chapter. We suggest that if you adopt them at the beginning of a software development project and stick with them, you will significantly increase your chances of success.

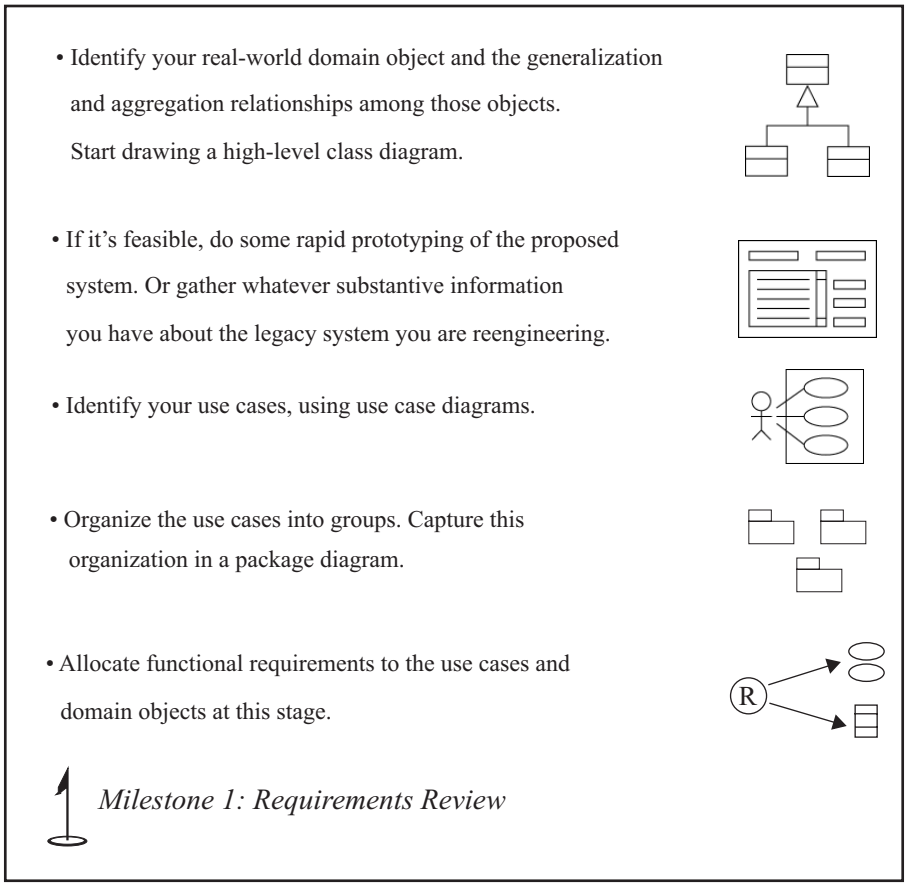


Figure 1-8: *Requirements Analysis*

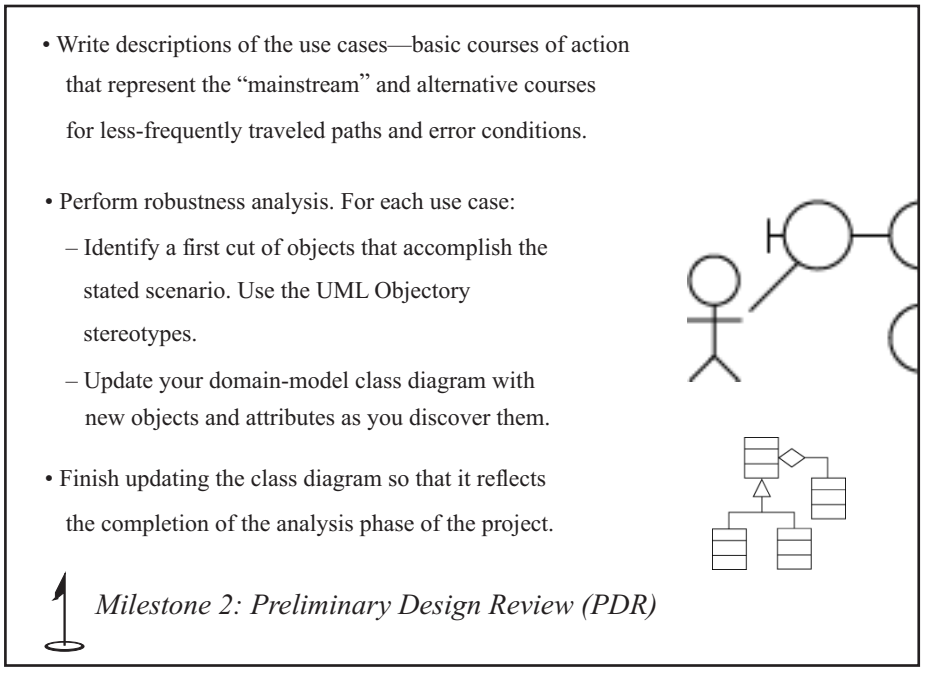
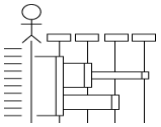
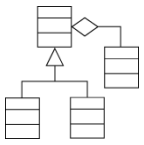


Figure 1-9: *Analysis and Preliminary Design*

- Allocate behavior. For each use case:
  - Identify the messages that need to be passed between objects, the objects, and the associated methods to be invoked. Draw a sequence diagram with use case text running down the left side and design information on the right. Continue to update the class diagram with attributes and operations as you find them.
  - If you wish, show, on a collaboration diagram, the key transactions between objects.
  - If you wish, use a state diagram to show the real-time behavior.



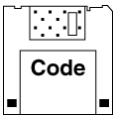


- Finish the static model by adding detailed design information (for instance, visibility values and patterns).
- Verify with your team that your design satisfies all the requirements you've identified.

Milestone 3: Critical Design Review (CDR)

Figure 1-10: *Design*

- As needed, produce diagrams, such as deployment and component diagrams, that will help you with the implementation phase.
- Write/generate the code.
- Perform unit and integration testing.
- Perform system and user-acceptance testing, using the use cases as black-box test cases for the latter.



Milestone 4: Delivery

Figure 1-11: *Implementation*

Requirements List for The Internet Bookstore

Starting in the next chapter, we're going to be following a running example, which we call The Internet Bookstore, through each phase of the process we've just outlined for you. The use cases we'll be working through, and the classes we'll discover, exist to satisfy certain requirements that our client (the owner of the bookstore we're going to build) has specified. These requirements include the following:

- The bookstore shall accept orders over the Internet.
- The bookstore shall maintain a list of accounts for up to 1,000,000 customers.
- The bookstore shall provide password protection for all accounts.
- The bookstore shall provide the ability to search the master book catalog.
- The bookstore shall provide a number of search methods on that catalog, including search by author, search by title, search by ISBN number, and search by keyword.
- The bookstore shall provide a secure means of allowing customers to pay by credit card.
- The bookstore shall provide a secure means of allowing customers to pay via purchase order.
- The bookstore shall provide a special kind of account that is preauthorized to pay via purchase order.
- The bookstore shall provide electronic links between the Web and database and the shipping fulfillment system.
- The bookstore shall provide electronic links between the Web and database and the inventory management system.
- The bookstore shall maintain reviews of books, and allow anyone to upload review comments.
- The bookstore shall maintain ratings on books, based on customer inputs.