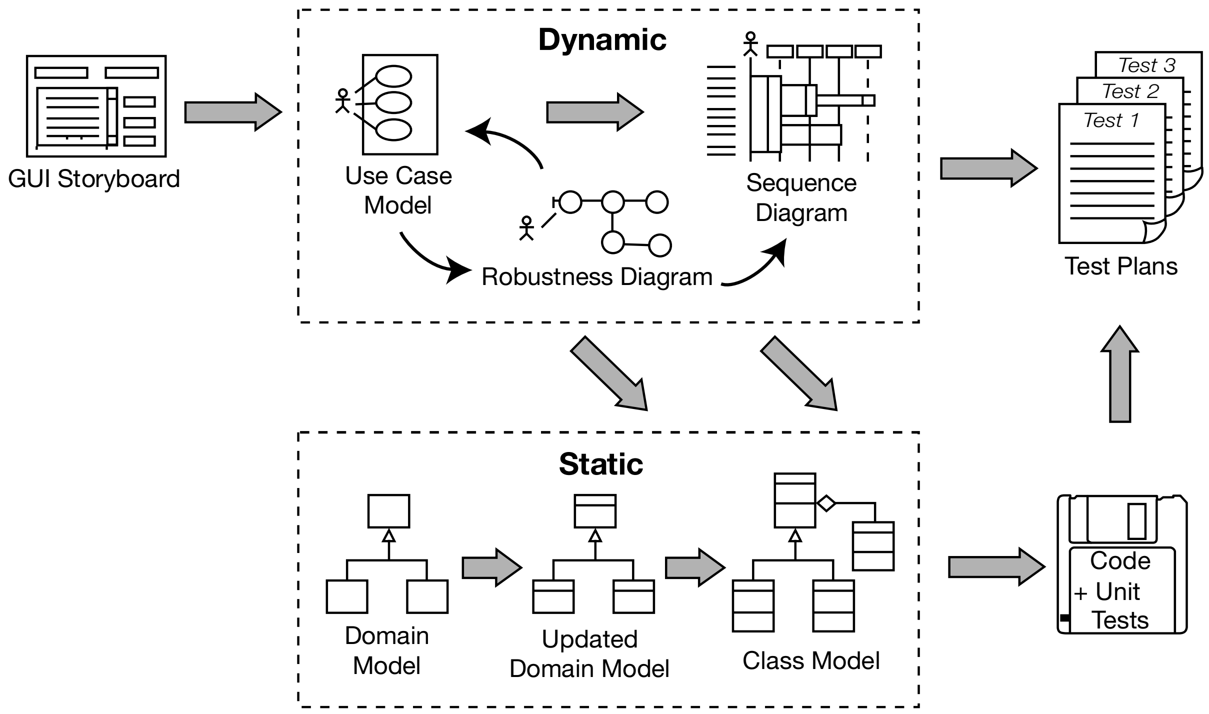
CHAPTER 1

Introduction to ICONIX Process



One process is much larger
And the other's way too small
And the full UML that OMG gives you
Is incomprehensible to all . . .

(Sing to the tune of "Go Ask Alice" by Jefferson Airplane)

In theory, every single aspect of the UML is potentially useful, but 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. ICONIX Process, as shown in the chapter's opening figure, 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 good analysis and design.

WHEN TO USE A COOKBOOK

There's a growing misconception in software development that cookbook approaches to software development don't work. We agree with this to an extent, because analysis and programming are massive, highly complex fields, and the number of different software project types is roughly equal to the number of software projects. However, we firmly believe that analysis and design can—and in fact should—be a specific sequence of repeatable steps. These steps aren't set in stone (i.e., they can be tailored), but it helps to have them there. In a world filled with doubt and uncertainty, it's nice to have a clearly defined sequence of "how-to" steps to refer back to.

Way back in the pre-UML days when Doug first started teaching a unified Booch/Rumbaugh/Jacobson modeling approach (around 1992/1993), one of his early training clients encouraged him to "write a cookbook, because my people like following cookbook approaches." While many have claimed that it's impossible to codify object-oriented analysis and design (OOAD) practices into a simple, repeatable set of steps (and it probably isn't possible in its entirety), ICONIX Process probably comes as close as anything out there to a cookbook approach to OOAD.

While there's still room for significant flexibility within the approach (e.g., adding in state or activity diagrams), ICONIX Process lays down a simple, minimal set of steps that generally lead to pretty good results. These results have proven to be consistent and repeatable over the last 12 years.

ICONIX Process in Theory

In this section we provide an overview of ICONIX Process, showing how all the activities fit together. We'll start with a very high-level view—kind of an overview of the overview—and then we'll examine each activity in more detail. As you're walking through the overview, keep referring back to the process diagram at the start of this chapter, to see how each part fits into the overall process.

Overview: Getting from Use Cases to Source Code

The diagram at the start of this chapter gives an overview of ICONIX Process. (We'll repeat this diagram at the start of each chapter, with the relevant section of the diagram shown in red.) As you can see from the diagram, ICONIX Process is divided into *dynamic* and *static* workflows, which are highly iterative: you might go through one iteration of the whole process for a small batch of use cases (perhaps a couple of packages' worth, which isn't a huge amount given that each use case is only a couple of paragraphs), all the way to source code and unit tests. For this reason, ICONIX Process is well suited to agile projects, where swift feedback is needed on such factors as the requirements, the design, and estimates.

Let's walk through the steps that we'll cover in the course of this book. The items in red correspond with the subtitles in this section (pretty slick, huh?).

As with any project, at some stage early on you begin exploring and defining the requirements. Note that within each phase there's a degree of parallelism, so all the activities in the requirements definition phase go on sort of overlapped and interleaved until they're ready.

Note There are many different types of requirements (e.g., nonfunctional requirements such as scalability). However, at a process level, we distinguish between *functional requirements* and *behavioral requirements*.

1. **REQUIREMENTS**

   a. Functional requirements: Define what the system should be capable of doing. Depending on how your project is organized, either you'll be involved in creating the functional requirements or the requirements will be "handed down from on high" by a customer or a team of business analysts.

   b. Domain modeling: Understand the problem space in unambiguous terms.

   c. Behavioral requirements: Define how the user and the system will interact (i.e., write the first-draft use cases). We recommend that you start with a GUI prototype (storyboarding the GUI) and identify all the use cases you're going to implement, or at least come up with a first-pass list of use cases, which you would reasonably expect to change as you explore the requirements in more depth.

   d. Milestone 1: Requirements Review: Make sure that the use case text matches the customer's expectations. Note that you might review the use cases in small batches, just prior to designing them.

Then in each iteration (i.e., for a small batch of use cases), you do the following.

2. **ANALYSIS/PRELIMINARY DESIGN**

   a. Robustness analysis: Draw a robustness diagram (an "object picture" of the steps in a use case), rewriting the use case text as you go.

   b. Update the domain model while you're writing the use case and drawing the robustness diagram. Here you will discover missing classes, correct ambiguities, and add attributes to the domain objects (e.g., identify that a `Book` object has a `Title`, `Author`, `Synopsis`, etc.).

   c. Name all the logical software functions (**controllers**) needed to make the use case work.

   d. Rewrite the first draft use cases.

3. Milestone 2: Preliminary Design Review (PDR)

4. **DETAILED DESIGN**

   a. Sequence diagramming: Draw a sequence diagram (one sequence diagram per use case) to show *in detail* how you're going to implement the use case. The primary function of sequence diagramming is to allocate behavior to your classes.

   b. Update the domain model while you're drawing the sequence diagram, and add operations[1] to the domain objects. By this stage, the domain objects are really domain classes, or *entities*, and the domain model should be fast becoming a *static model*, or *class diagram*—a crucial part of your detailed design.

   c. Clean up the static model.

5. Milestone 3: Critical Design Review (CDR)

6. **IMPLEMENTATION**

   a. Coding/unit testing: Write the code and the unit tests. (Or, depending on your preferences, write the unit tests and then the code.[2])

   b. Integration and scenario testing: Base the integration tests on the use cases, so that you're testing both the basic course and the alternate courses.

   c. Perform a Code Review and Model Update to prepare for the next round of development work.

For most of the rest of this chapter, we describe these steps in a little more detail. Throughout the rest of the book, we describe these steps in *much* greater detail, and provide lots of examples and exercises to help you understand how best to apply them to your own project.

Requirements

Figure 1-1 shows the steps involved in defining the **behavioral requirements**—that is, drawing the initial domain model and writing the first-draft use cases.

The steps shown in Figure 1-1 are covered in Chapters 2, 3, and 4.

1. Also called methods, functions, or messages, depending which programming language you use.
2. For Test-Driven Development (TDD) fans, in Chapter 12 we illustrate a method of incorporating the "test first" approach into ICONIX Process. The result is essentially "Design-Driven Testing."

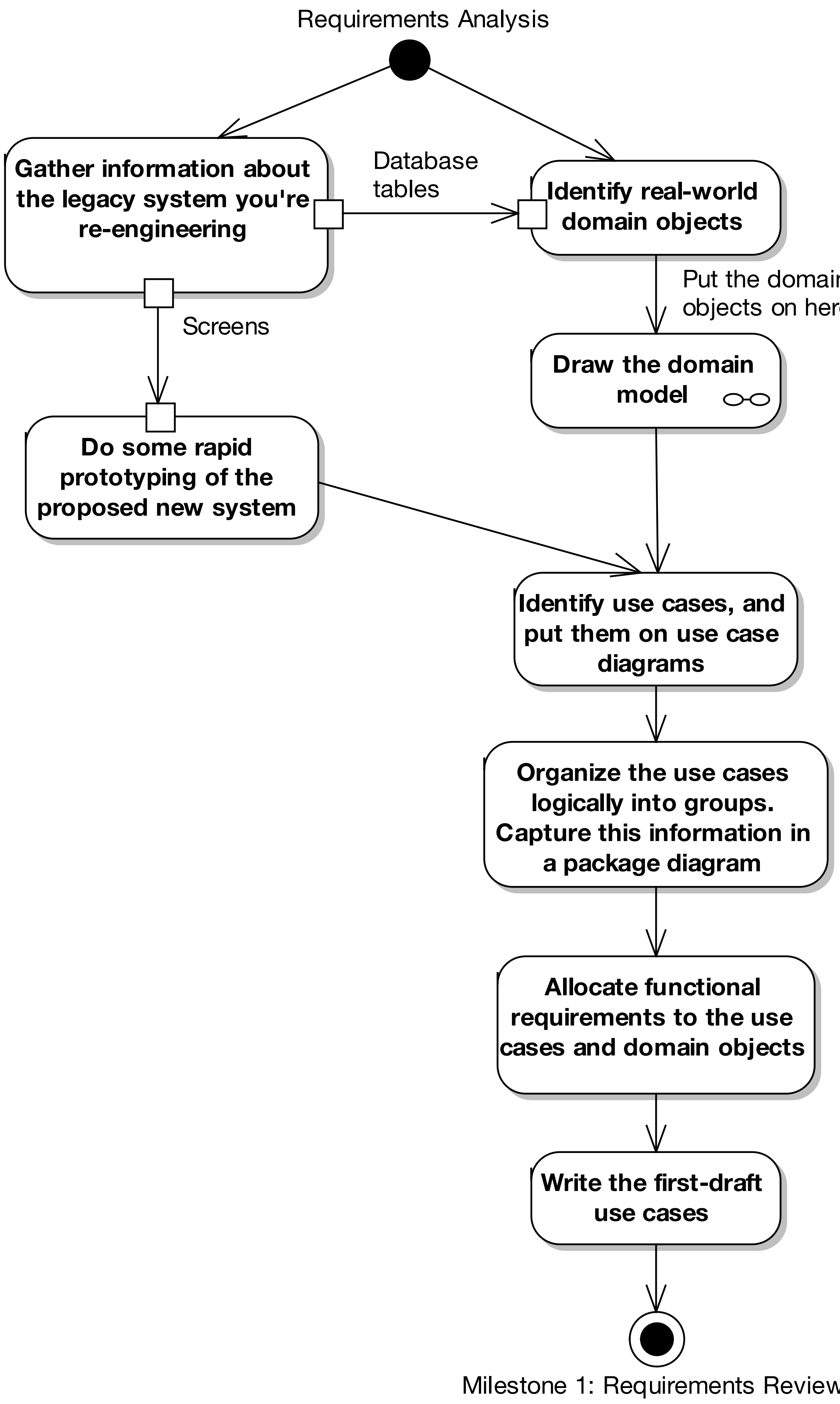

Requirements Analysis
Gather information about the legacy system you're re-engineering
Database tables
Identify real-world domain objects
Put the domain objects on here
Screens
Draw the domain model
Do some rapid prototyping of the proposed new system
Identify use cases, and put them on use case diagrams
Organize the use cases logically into groups. Capture this information in a package diagram
Allocate functional requirements to the use cases and domain objects
Write the first-draft use cases
Milestone 1: Requirements Review

Figure 1-1. *Requirements analysis*

Functional Requirements (What Will the System Be Capable Of?)

Right at the start of the project, somebody (possibly a team of business analysts) will be talking to the customer, end users, and various project stakeholders, and that person (or team) will most likely create a big Microsoft Word document packed to the brim with functional requirements. This is an important document, but it's difficult to create a design from (or to create an accurate estimate from, for that matter), as it tends to be quite unstructured. (Even if every requirement is numbered in a big document-length list, that still doesn't quite count as being structured.)

Note The initial stages of ICONIX Process involve creating a set of unambiguous behavioral requirements (use cases) that are "closer to the metal" than the functional requirements specification, and that *can* be easily designed from.

Creating functional requirements falls just slightly outside the scope of ICONIX Process, but we do offer some advice on the matter.[3] Probably the best way to describe our approach to requirements gathering is to list our **top 10 requirements gathering guidelines**. We describe these in more detail in Chapter 13.

10. Use a modeling tool that supports linkage and traceability between requirements and use cases.
9. Link requirements to use cases by dragging and dropping.
8. Avoid **dysfunctional requirements** by separating functional details from your behavioral specification.
7. Write at least one test case for each requirement.
6. Treat requirements as first-class citizens in the model.
5. Distinguish between different types of requirements.
4. Avoid the "big monolithic document" syndrome.
3. Create estimates from the use case scenarios, not from the functional requirements.
2. Don't be afraid of examples when writing functional requirements.
1. Don't make your requirements a technical fashion statement.

With the functional requirements written (whether by your team or by somebody else), you'll really want to do some additional analysis work, to create a set of **behavioral requirements** (use cases) from which you can create a high-level, preliminary design.

3. In Chapter 13, we show how to link your use cases back to the original requirements.

Domain Modeling

Domain modeling is the task of building a project glossary, or a dictionary of terms used in your project (e.g., an Internet bookstore project would include domain objects such as `Book`, `Customer`, `Order`, and `Order Item`). Its purpose is to make sure everyone on the project understands the problem space in unambiguous terms. The domain model for a project defines the scope and forms the foundation on which to build your use cases. The domain model also provides a common vocabulary to enable clear communication among members of a project team. Expect early versions of your domain model to be wrong; as you explore each use case, you'll "firm up" the domain model as you go.

Here are our **top 10 domain modeling guidelines**. We describe these in more detail in Chapter 2.

10. Focus on real-world (problem domain) objects.

9. Use generalization (is-a) and aggregation (has-a) relationships to show how the objects relate to each other.

8. Limit your initial domain modeling efforts to a couple of hours.

7. Organize your classes around key abstractions in the problem domain.

6. Don't mistake your domain model for a data model.

5. Don't confuse an object (which represents a single instance) with a database table (which contains a collection of things).

4. Use the domain model as a project glossary.

3. Do your initial domain model before you write your use cases, to avoid name ambiguity.

2. Don't expect your final class diagrams to precisely match your domain model, but there should be some resemblance between them.

1. Don't put screens and other GUI-specific classes on your domain model.

Once you have your first-pass domain model, you can use it to write the use cases—that is, to create your **behavioral requirements**, which we introduce in the next section.

Behavioral Requirements (How Will the User and the System Interact?)

ICONIX Process is a scenario-based approach; the primary mechanism for decomposing and modeling the system is on a scenario-by-scenario basis. But when you use ICONIX Process, your goal is to produce an object-oriented design that you can code from. Therefore, you need to link the scenarios to objects. You do this by writing the use cases using the domain model that you created in the previous step.

Storyboarding the GUI

Behavior requirements detail the user's actions and the system's responses to those actions. For the vast majority of software systems, this interaction between user and system takes place via screens, windows, or pages. When you're exploring the behavioral requirements, you capture the usage scenarios in narrative text form in the use cases, and these narratives have come from detailed conversations with customers and end users.

It's notoriously difficult for us humans to picture a proposed system in our mind's eye. So quite often it's easier for the customers and end users to relate to a visual aid, which often takes the form of a sequence of screens. These can be simple line drawings on paper, a PowerPoint slide show that sequences through the screens, an HTML prototype with core functionality left out—the exact form doesn't matter much. What's important is that they present a sequence of screens as they will appear to the users within the context of the usage scenarios being modeled.

It's also important that the screen mockups include details about the various buttons, menus, and other action-oriented parts of the UI. It's amazing how often a use case done without this sort of accompanying visual aid will omit alternate course behavior for events like "user clicks Cancel button," and how much better the use cases become when accompanied by a UI storyboard.

Use Case Modeling

Use cases describe the way the user will interact with the system and how the system will respond. Here are our **top 10 use case modeling guidelines**. We describe these in more detail in Chapter 3.

10. Follow the two-paragraph rule.
9. Organize your use cases with actors and use case diagrams.
8. Write your use cases in active voice.
7. Write your use case using an event/response flow, describing both sides of the user/system dialogue.
6. Use GUI storyboards, prototypes, screen mockups, etc.
5. Remember that your use case is really a runtime behavior specification.
4. Write the use case in the context of the object model.
3. Write your use cases using a noun-verb-noun sentence structure.
2. Reference domain classes by name.
1. Reference boundary classes (e.g., screens) by name.

Milestone 1: Requirements Review

Right at the end of Figure 1-1, you'll see the Requirements Review milestone. This vital step ensures that the requirements are sufficiently well understood by both the development team and the customer/users/project stakeholders.

Here are our **top 10 requirements review guidelines**. We describe these in more detail in Chapter 4.

10. Make sure your domain model describes at least 80% of the most important abstractions (i.e., real-world objects) from your problem domain, in nontechnical language that your end users can understand.
9. Make sure your domain model shows the is-a (generalization) and has-a (aggregation) relationships between the domain objects.
8. Make sure your use cases describe both basic and alternate courses of action, in active voice.
7. If you have lists of functional requirements (i.e., "shall" statements), make sure these are not absorbed into and "intermangled" with the active voice use case text.[4]
6. Make sure you've organized your use cases into packages and that each package has at least one use case diagram.
5. Make sure your use cases are written in the context of the object model.
4. Put your use cases in the context of the user interface.
3. Supplement your use case descriptions with some sort of storyboard, line drawing, screen mockup, or GUI prototype.
2. Review the use cases, domain model, and screen mockups/GUI prototypes with end users, stakeholders, and marketing folks, in addition to more technical members of your staff.
1. Structure the review around our "eight easy steps to a better use case" (see Chapter 4).

Once the requirements review is complete, you can move on to the **preliminary design** stage.

Analysis/Preliminary Design

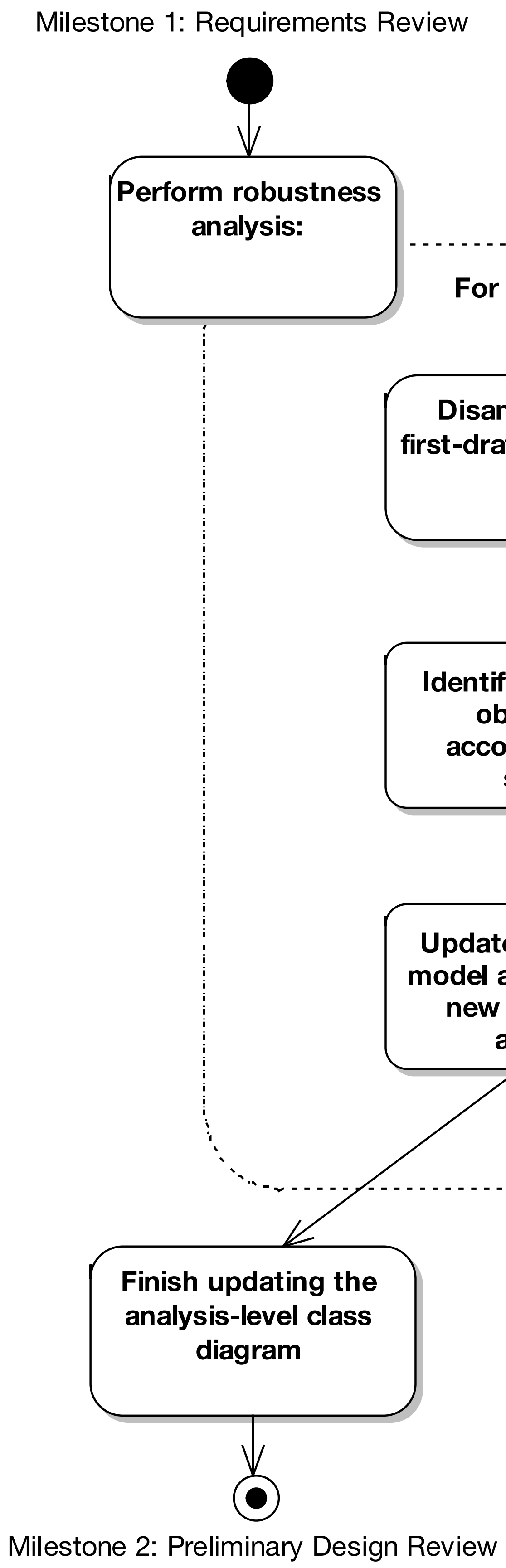
Analysis is about **building the right system**. Design is about **building the system right**. Preliminary design is an intermediate step between analysis and design.

Preliminary design explicitly recognizes something that many people recognize implicitly:

You usually can't fully understand the requirements that you're dealing with unless you do some exploratory design.

Figure 1-2 (which follows on from Figure 1-1) shows the **preliminary design** steps.

4. In Chapter 13, we introduce the term "intermangled" to describe use case text that has had functional requirements text mangled into it.



Milestone 1: Requirements Review
Perform robustness analysis:
For each use case . . .
Disambiguate the first-draft use case text
Identify a first cut of objects that accomplish each scenario
Update your domain model as you discover new objects and attributes
Finish updating the analysis-level class diagram
Milestone 2: Preliminary Design Review

Figure 1-2. *Analysis/preliminary design*

The preliminary design step (aka **robustness analysis**) involves doing the exploratory design you need to understand the requirements, refining and removing ambiguity from (aka disambiguating) those requirements as a result of the exploratory design, and linking the behavior requirements (use case scenarios) to the objects (domain model).

The steps shown in Figure 1-2 are covered in Chapters 5, 6, and 7.

Robustness Analysis

To get from use cases to detailed design (and then to code), you need to link your use cases to objects. Robustness analysis helps you to bridge the gap between analysis and design by doing exactly that. It's a way of analyzing your use case text and identifying a first-guess set of objects for each use case.

Here are our **top 10 robustness analysis guidelines**. We describe these in more detail in Chapter 5.

10. Paste the use case text directly onto your robustness diagram.
9. Take your entity classes from the domain model, and add any that are missing.
8. Expect to rewrite (disambiguate) your use case while drawing the robustness diagram.
7. Make a boundary object for each screen, and name your screens unambiguously.
6. Remember that controllers are only occasionally **real control objects**; they are typically **logical software functions**.
5. Don't worry about the direction of the arrows on a robustness diagram.
4. It's OK to drag a use case onto a robustness diagram if it's invoked from the parent use case.
3. The robustness diagram represents a preliminary conceptual design of a use case, not a literal detailed design.
2. Boundary and entity classes on a robustness diagram will generally become object instances on a sequence diagram, while controllers will become messages.
1. Remember that a robustness diagram is an "object picture" of a use case, whose purpose is to force refinement of both use case text and the object model.

With the preliminary design complete, your use cases should now be thoroughly disambiguated and thus written in the context of the domain model. The domain model itself should have helped to eliminate common issues such as duplicate names for the same item, and the classes on the domain model should also have attributes assigned to them (but not operations, yet).

In theory you should now be ready to start the detailed design, but in practice, it really helps to perform a quick **Preliminary Design Review** (PDR) first.

Milestone 2: Preliminary Design Review

The Preliminary Design Review (PDR) session helps you to make sure that the robustness diagrams, the domain model, and the use case text all match each other. This review is the "gateway" between the preliminary design and detailed design stages, for each package of use cases.

Here are our **top 10 PDR guidelines**. We describe these in more detail in Chapter 6.

10. For each use case, make sure the use case text matches the robustness diagram, using the highlighter test.
9. Make sure that all the entities on all robustness diagrams appear within the updated domain model.
8. Make sure that you can trace data flow between entity classes and screens.
7. Don't forget the alternate courses, and don't forget to write behavior for each of them when you find them.
6. Make sure each use case covers both sides of the dialogue between user and system.
5. Make sure you haven't violated the syntax rules for robustness analysis,
4. Make sure that this review includes both nontechnical (customer, marketing team, etc.) and technical folks (programmers).
3. Make sure your use cases are in the context of the object model and in the context of the GUI.
2. Make sure your robustness diagrams (and the corresponding use case text) don't attempt to show the same level of detail that will be shown on the sequence diagrams (i.e., don't try to do detailed design yet).
1. Follow our "six easy steps" to a better preliminary design (see Chapter 6).

With this review session complete, you can now be confident that the diagrams and the use case text match each other, and that both are complete and correctly represent the desired system behavior. It should now be a relatively straightforward matter to create the **detailed design**.

Detailed Design

Detailed design is about building the system right. We hope that by the time you get to this point, you have a pretty good understanding of what the "right system" is, because you've worked hard to develop that understanding. So now you're worrying about efficiency in terms of execution times, network loading, and memory footprint, and you're concerned with reusability of code where possible.

Figure 1-3 shows the steps involved in **detailed design**.

The steps shown in Figure 1-3 are covered in Chapters 8 and 9.

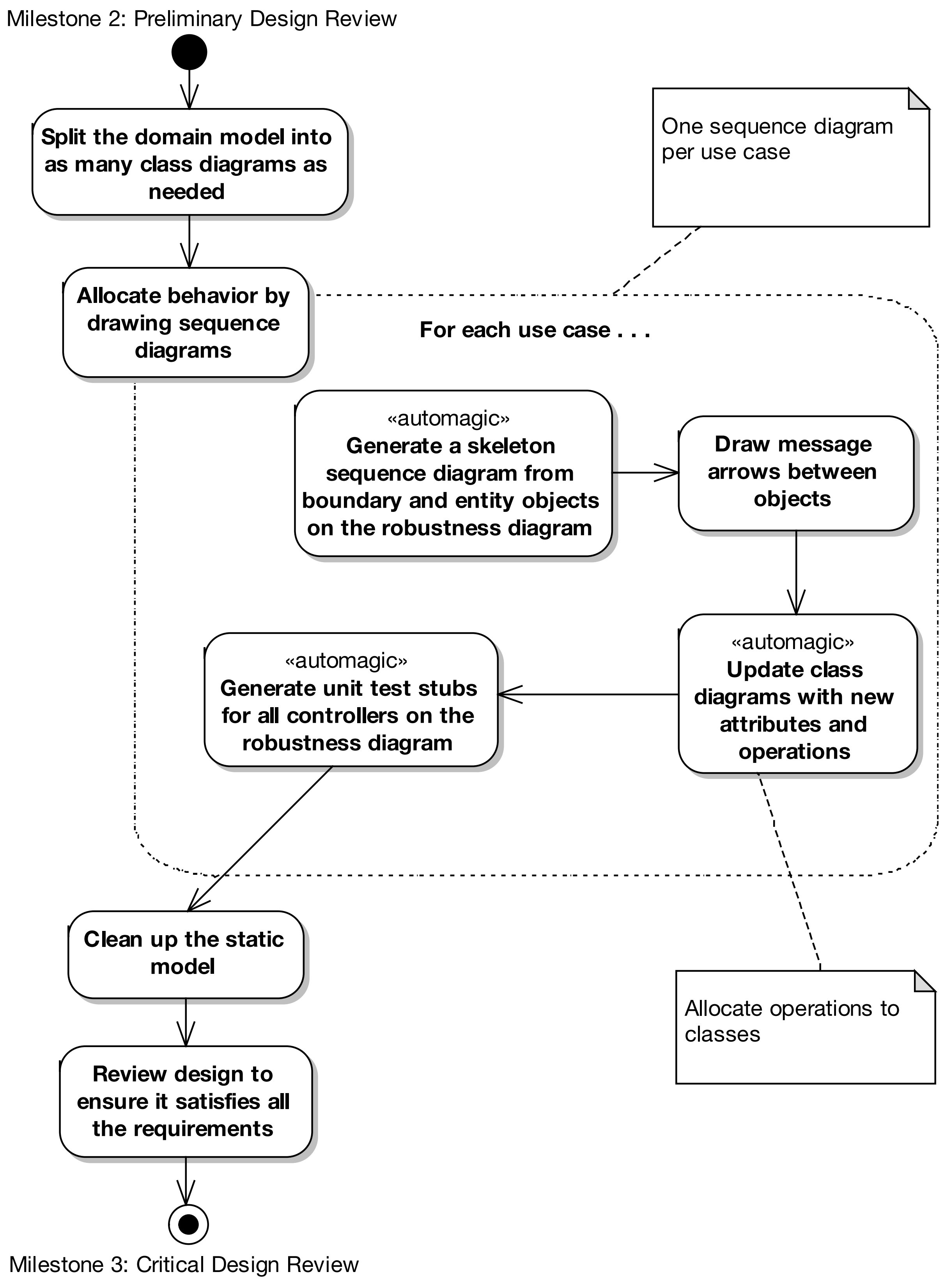

Milestone 2: Preliminary Design Review
Split the domain model into as many class diagrams as needed
Allocate behavior by drawing sequence diagrams
One sequence diagram per use case
For each use case . . .
«automagic» Generate a skeleton sequence diagram from boundary and entity objects on the robustness diagram
Draw message arrows between objects
«automagic» Update class diagrams with new attributes and operations
«automagic» Generate unit test stubs for all controllers on the robustness diagram
Clean up the static model
Allocate operations to classes
Review design to ensure it satisfies all the requirements
Milestone 3: Critical Design Review

Figure 1-3. *Detailed design*

Sequence Diagramming (Allocate Behavior to Classes)

ICONIX Process uses the **sequence diagram** as the main vehicle for exploring the detailed design of a system on a scenario-by-scenario basis.

In object-oriented design, a large part of building the system right is concerned with finding an optimal allocation of functions to classes (aka behavior allocation). The essence of this is drawing message arrows on sequence diagrams and allowing a modeling tool to automatically assign an operation to the class of the target object that receives the runtime message.

Here are our **top 10 sequence diagramming guidelines**. We describe these in more detail in Chapter 8.

10. Understand **why** you're drawing a sequence diagram, to get the most out of it.
9. Do a sequence diagram for every use case, with both basic and alternate courses on the same diagram.
8. Start your sequence diagram from the boundary classes, entity classes, actors, and use case text that result from robustness analysis.
7. Use the sequence diagram to show how the behavior of the use case (i.e., all the controllers from the robustness diagram) is accomplished by the objects.
6. Make sure your use case text maps to the messages being passed on the sequence diagram. Try to line up the text and message arrows.
5. Don't spend too much time worrying about focus of control.
4. Assign operations to classes while drawing messages. Most visual modeling tools support this capability.
3. Review your class diagrams frequently while you're assigning operations to classes, to make sure all the operations are on the appropriate classes.
2. **Prefactor** your design on sequence diagrams before coding.
1. Clean up the static model before proceeding to the CDR.

By now, you're almost ready to begin coding. You'll need to perform a Critical Design Review (CDR) first; but before that, it pays dividends to revisit the static model and clean it up.

Cleaning Up the Static Model

Take a long, hard look at your static model, with a view toward tidying up the design, resolving real-world design issues, identifying useful design patterns that can be factored in to improve the design, and so on. This should at least be done as a final step before proceeding to the CDR, but you can start thinking at this level in the design *before* drawing the sequence diagram.

By this stage, you should have an extremely well-factored design that works within the real-world constraints of your project's requirements, application framework design, deployment topology, and so forth. There's just one last stop before you begin coding: the CDR.

Milestone 3: Critical Design Review

The CDR helps you to achieve three important goals, before you begin coding for the current batch of use cases:

- Ensure that the "how" of detailed design matches up with the "what" specified in your requirements.
- Review the quality of your design.
- Check for continuity of messages on your sequence diagrams (iron out "leaps of logic" in the design).

Here are our **top 10 CDR guidelines**. We describe these in more detail in Chapter 9.

10. Make sure the sequence diagram matches the use case text.

9. Make sure (yes, again) that each sequence diagram accounts for both basic and alternate courses of action.

8. Make sure that operations have been allocated to classes appropriately.

7. Review the classes on your class diagrams to ensure they all have an appropriate set of attributes and operations.

6. If your design reflects the use of patterns or other detailed implementation constructs, check that these details are reflected on the sequence diagram.

5. Trace your functional (and nonfunctional) requirements to your use cases and classes to ensure you have covered them all.

4. Make sure your programmers "sanity check" the design and are confident that they can build it and that it will work as intended.

3. Make sure all your attributes are typed correctly, and that return values and parameter lists on your operations are complete and correct.

2. Generate the code headers for your classes, and inspect them closely.

1. Review the test plan for your release.

If you've gone through the detailed design for each use case and performed a CDR (as described in Chapter 9), then your design really should be fighting-fit now, and easily ready for coding.

Implementation

Figure 1-4 shows the steps involved in **coding and testing** (i.e., **implementation**).

Once you've made the effort to drive a model from use cases through detailed design, it would be lunacy to disregard the model and just start coding totally independent of the model you've produced.

Similarly, your modeling should provide a basis for knowing exactly what software functions will need to be unit tested, so you can drive the unit tests from the model in a similar manner to generating code from the detailed class diagrams.

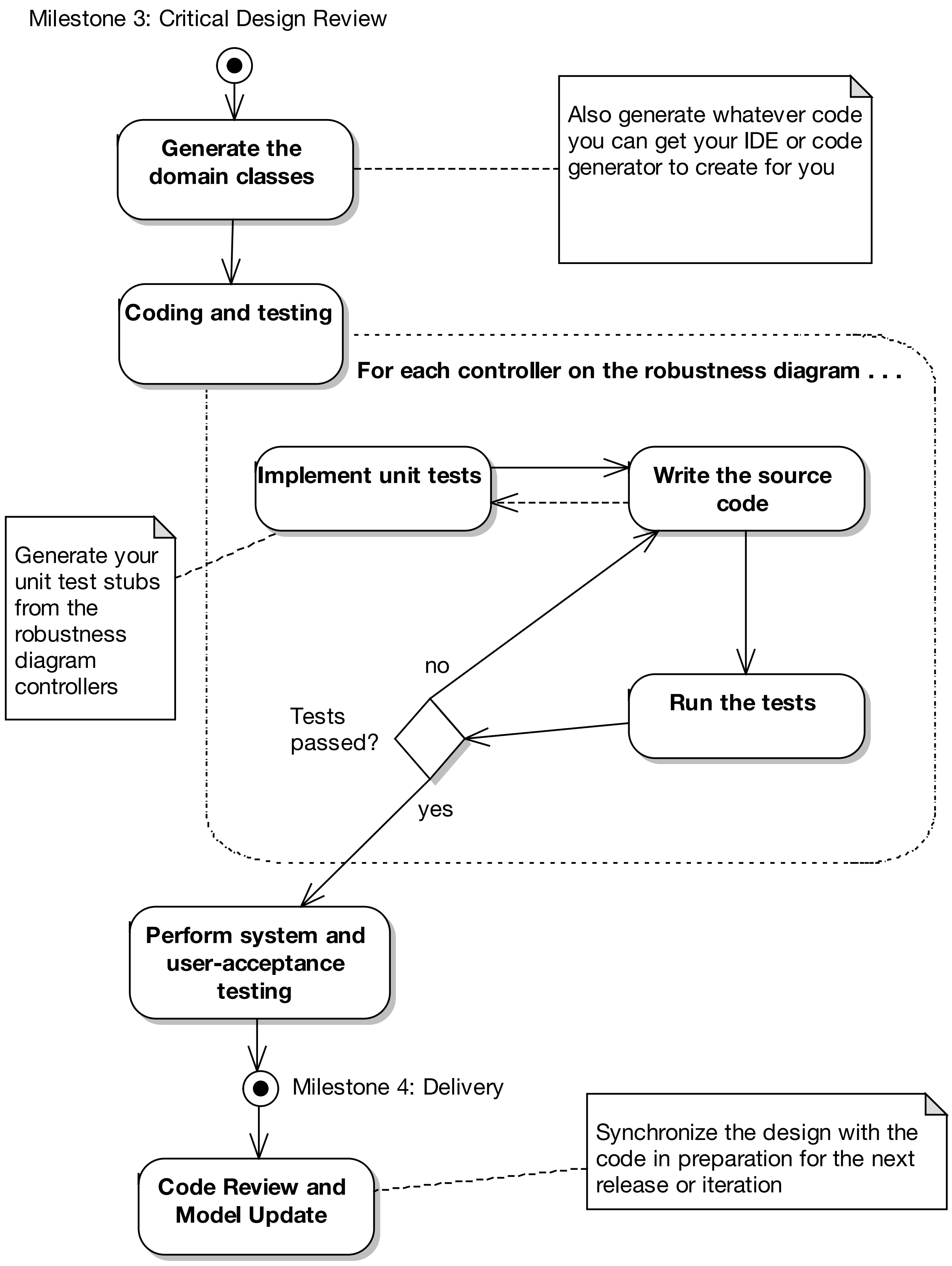

Milestone 3: Critical Design Review
Generate the domain classes
Also generate whatever code you can get your IDE or code generator to create for you
Coding and testing
For each controller on the robustness diagram . . .
Implement unit tests
Write the source code
Generate your unit test stubs from the robustness diagram controllers
no
Tests passed?
Run the tests
yes
Perform system and user-acceptance testing
Milestone 4: Delivery
Synchronize the design with the code in preparation for the next release or iteration
Code Review and Model Update

Figure 1-4. *Implementation*

Technology available in today's modeling tools (at least the ones we use) also provides for easy and convenient linkage between the UML model and the coding environment. We've extended ICONIX Process to leverage this exciting new technology.

The steps shown in Figure 1-4 are covered in Chapters 10, 11, and 12.

Implementation (Coding)

Here are our **top 10 implementation guidelines**. We describe these in more detail in Chapter 10.

10. Be sure to drive the code directly from the design.
9. If coding reveals the design to be wrong in some way, change it. But also review the process.
8. Hold regular code inspections.
7. Always question the framework's design choices.
6. Don't let framework issues take over from business issues.
5. If the code starts to get out of control, hit the brakes and revisit the design.
4. Keep the design and the code in sync.
3. Focus on unit testing while implementing the code.
2. Don't overcomment your code (it makes your code less maintainable and more difficult to read).
1. Remember to implement the alternate courses as well as the basic courses.

Unit testing is an important (and integral) part of implementation.

Unit Testing

While coding, you should also be writing unit tests that are tied into the use cases. These tests allow you to prove, in an automated and repeatable way, that the system behavior described in each use case has been implemented correctly. Essentially, you're testing all the software functions that you identified during robustness analysis.

Here are our **top 10 unit testing guidelines**. We describe these in more detail in Chapter 12.

10. Adopt a "testing mind-set" wherein every bug found is a victory and not a defeat. If you find (and fix) the bug in testing, the users won't find it in the released product.
9. Understand the different kinds of testing, and when and why you'd use each one.
8. When unit testing, create one or more unit tests for each controller on each robustness diagram.
7. For real-time systems, use the elements on state diagrams as the basis for test cases.

6. Do requirement-level verification (i.e., check that each requirement you have identified is accounted for).

5. Use a traceability matrix to assist in requirement verification.

4. Do scenario-level acceptance testing for each use case.

3. Expand threads in your test scenarios to cover a complete path through the appropriate part of the basic course plus each alternate course in your scenario testing.

2. Use a testing framework like JUnit to store and organize your unit tests.

1. Keep your unit tests fine-grained.

As we discuss in Chapter 12, other types of testing are performed on different project artifacts and at different stages in the project—in particular, **integration/scenario testing**.

Expand Threads for Integration and Scenario Testing

This activity involves expanding the sunny day/rainy day threads of the use cases. The integration tests come from the use cases, in the form of testing the following:

1. The entire sunny-day scenario (the basic course)

2. Part of the sunny-day scenario plus each individual rainy day scenario (the alternate courses)

For example, a use case with three alternate courses would need (at minimum) four integration test scenarios: one for the basic course and one for each alternate course (including whichever part of the basic course goes along with it).

With the code and tests written for a particular use case (and with the tests passing!), it's important to perform a **Code Review and Model Update**.

Code Review and Model Update

The main purpose of the Code Review and Model Update milestone is to synchronize the code and the model before the next iteration begins. This ongoing effort to keep the design tight prevents entropy, or **code rot**, from setting in as more and more functionality is added to a complex system. Once you've completed this milestone, the design and the code should be in a very good state, ready for development to begin on the next use case (or batch of use cases).

Here are our **top 10 Code Review and Model Update guidelines**. We describe these in more detail in Chapter 11.

10. Prepare for the review, and make sure all participants have read the relevant review material prior to the meeting.

9. Create a high-level list of items to review, based on the use cases.

8. If necessary, break down each item in the list into a smaller checklist.

7. Review code at several different levels.

6. Gather data during the review, and use it to accumulate boilerplate checklists for future reviews.
5. Follow up the review with a list of action points e-mailed to all people involved.
4. Try to focus on error detection during the review, not error correction.
3. Use an integrated code/model browser that hot-links your modeling tool to your code editor.
2. Keep it "just formal enough" with checklists and follow-up action lists, but don't overdo the bureaucracy.
1. Remember that it's also a Model Update session, not just a Code Review.

That about wraps up our overview of ICONIX Process. It probably seems as if there's a lot of information to absorb, but the process itself is actually very straightforward once you understand exactly why each step is performed.

Extensions to ICONIX Process

Although it's been used in hundreds of large-scale IT projects, the core ICONIX Process has stayed much the same in the last 10 to 15 years. However, in *Agile Development with ICONIX Process* (Apress, 2005), we published some extensions to the core process. These extensions include the following:

- Performing persona analysis
- Supplementing the process with Test-Driven Development (TDD)
- Driving test cases from the analysis model

Persona Analysis

Persona analysis as an interaction design technique makes actors and use cases more concrete and tangible for project stakeholders. Many people find actors and use cases too abstract, so this approach addresses the issue head-on.

A *persona* is a description of a fictional person: a prototypical target user. The person is given a name and a brief description of his or her job, goals and aspirations, level of ability—anything that might be relevant to how that person uses and perceives the product you're designing. You'd then write *interaction scenarios* (a form of use case that describes in more detail the user's motivations behind his or her interaction with the system), based around the persona you've defined. Using ICONIX Process, you would write a few detailed interaction scenarios to make sure the system is correctly focused on your target user, and then proceed to write the more minimal, ICONIX-style use cases for the system as a whole.

Test-Driven Development (TDD)

TDD is an increasingly popular method of designing software by writing unit tests. The design effectively "evolves" as you write the code. Teams have begun to realize that TDD by itself can be a long-winded design process (to say the least) and benefits greatly from some initial, up-front design modeling. ICONIX Process is a prime candidate for this, because its *robustness analysis* technique works well in collaborative design workshops with teams modeling on a whiteboard (or on a CASE tool hooked up to a projector).

Driving Test Cases from the Analysis Model

It makes sense to link your models as closely as possible to testing, and in fact to drive the testing effort from your use case–driven models. This extension to ICONIX Process drives the identification of test cases directly from the robustness diagram, in a parallel manner to the way we drive the creation of a skeleton sequence diagram from a robustness diagram. In short, the nouns (Entity and Boundary objects) from the robustness diagram become object instances on the sequence diagram, and the verbs (controllers) get test cases created for them. We discuss this process in depth in Chapter 12.

ICONIX Process in Practice: The Internet Bookstore Example

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.

When we get to the sequence diagramming (detailed design) stage in Chapter 8, we'll begin to implement the Internet Bookstore using the Spring Framework, a popular Java enterprise application framework. This book isn't primarily about Spring, so we won't dwell on the technical details regarding Spring itself (although we will suggest further resources, both print and online). Instead we'll focus on the ways in which ICONIX Process is used to create well-designed source code for a realistic web-based application.

The techniques we describe in this book should still be more than relevant to users of other application frameworks and other object-oriented programming languages.

The use cases we'll be working through and the classes we'll discover exist to satisfy certain requirements that our customer (the fictional owner of the bookstore we're going to build) has specified. We'll cover these requirements in Chapter 2, where we show how to derive our first-pass version of the **domain model** from the requirements.

Summary

In this chapter we introduced ICONIX Process and described its background and the driving principles behind it. We also described its key features and walked through the process, from use cases to code.

In the next chapter, we describe in detail the first major stage of ICONIX Process: domain modeling.