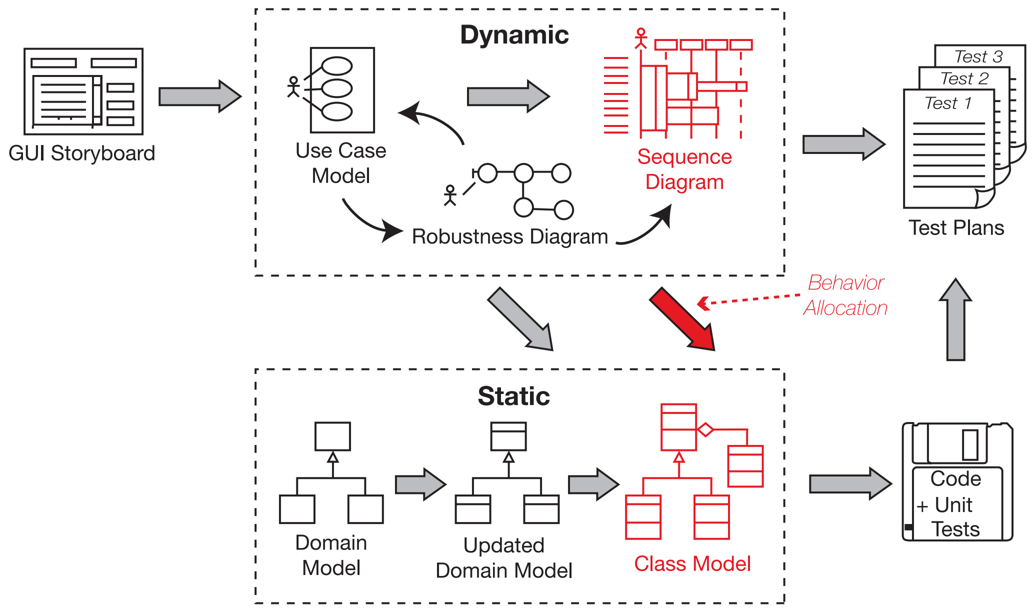
PART 3

Design and Coding

CHAPTER 8

Sequence Diagrams



Once you've finished robustness analysis, and you've held a preliminary design review, it's time to begin the detailed design effort. By this time, your use case text should be complete, correct, detailed, and explicit. In short, your use cases should be in a state where you can create a detailed design from them.

All the steps in the process so far have been preparing the use cases for the detailed design activity. Having completed robustness analysis and the PDR, you should now have discovered pretty much all of the domain classes that you're going to need. You also need to have the technical architecture (TA) nailed down by this stage.

The 10,000-Foot View

Before we leap into the details of sequence diagramming, let's take a step back (or up) and look at the bigger picture of object-oriented design (OOD).

Sequence Diagrams and Detailed OOD

If you figure that preliminary design is all about discovery of classes (aka **object discovery**), then detailed design is, by contrast, about allocating behavior (aka **behavior allocation**)—that is, allocating the software functions you've identified into the set of classes you discovered during preliminary design.

When you draw sequence diagrams, you're taking another sweep through the preliminary design, adding in detail.

With preliminary design, you made some informal first guesses at how the classes will interact with each other. Now it's time to make those statements very precise, to turn them into a detailed design that works within the TA that you've defined.

You use sequence diagrams to drive the detailed design. But note that we advocate drawing your sequence diagrams in a minimal, quite specific format (which we describe fully in this chapter). There's a direct link between each use case, its robustness diagram, and the sequence diagrams. Just as you drew one robustness diagram per use case, you'll also draw one sequence diagram per use case.

Sequence Diagram Notation

Before we dive into the best practices for drawing sequence diagrams from your use cases, it helps to understand the stuff that a sequence diagram is composed of (see Figure 8-1).

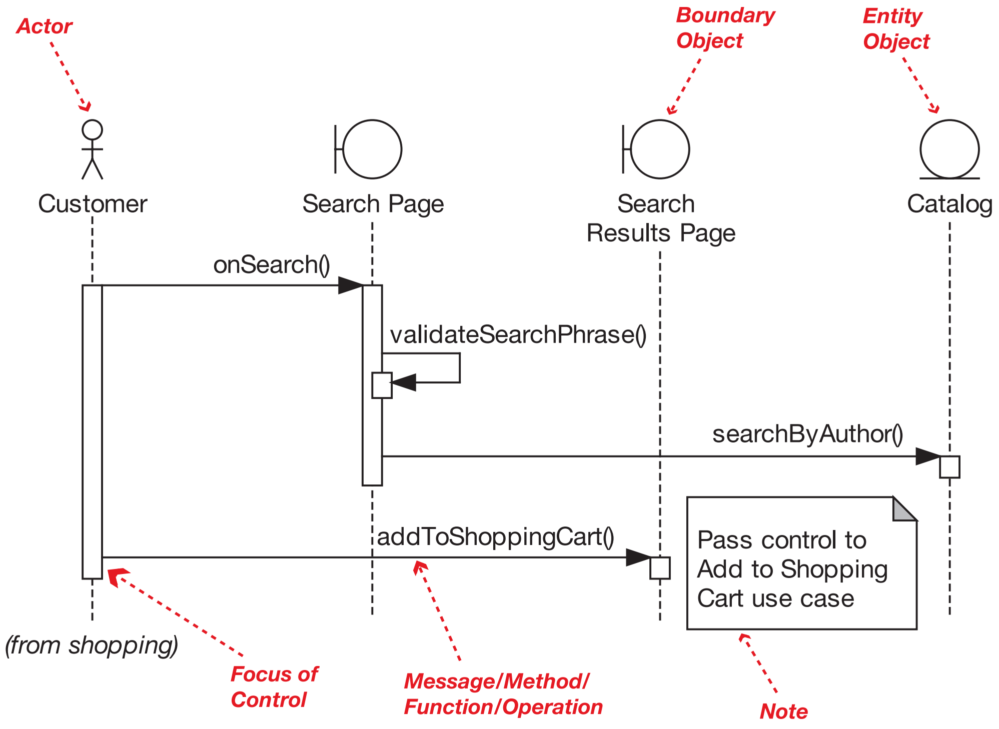


Figure 8-1. *Sequence diagram notation*

The objects across the top of the diagram (`Customer`, `Search Page`, etc.) are interacting with each other by passing **messages** back and forth. The vertical dotted lines (or *object lifelines*) represent time, so the process shown in the diagram begins with the topmost message (`Customer` calling `onSearch()` on `Search Page`).

An **actor** (the Customer in Figure 8-1) is the user whose interaction with the system you've described in each of the use cases. (See the "system boundary" diagram in Figure 3-2.) You should recognize the **boundary object** and **entity object** icons from robustness diagramming in Chapter 5. (In Figure 8-1, the boundary objects are `Search Page` and `Search Results Page`; the entity object is `Catalog`.)

However, notice that **there are no controller objects on the sequence diagram** (although there could be). This is because when you draw the sequence diagrams, the controllers (the verbs) are turned into messages on the boundary and entity objects (the nouns). Sometimes you'll find real controller classes, such as a "manager" or a "dispatcher" class, and sometimes a framework might tempt you to litter your design with dozens of tiny "controller classes," but as a general rule of thumb, 80% or so of the controllers from the robustness diagrams can be implemented as one or more operations on the entity and boundary classes. (More about this important aspect of sequence diagramming later.)

The **focus of control** represents the time that a particular method/function has control. It starts with the arrow going *into* the function and finishes when the function returns.

■**Note** You normally don't need to draw a return arrow, except in special circumstances (e.g., to show an asynchronous return value). That's because parameters can be passed back as arguments to the operation.

As you'll see in item 5 in the next section, the focus of control is best switched off, as it tends to be something of a distraction from what you're trying to achieve at this stage in the process.

Sequence Diagramming in Theory

In this section, we demonstrate how to use sequence diagrams as a mechanism for exploring and filling in the detailed OO design for each use case. We illustrate this theory with examples from the Internet Bookstore project. And we begin, as usual, with our top 10 guidelines list.

Top 10 Sequence Diagramming Guidelines

10. Understand **why** you're drawing a sequence diagram, to get the most out of it.
9. Draw 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.

Let's look at each of these top 10 guidelines in more detail.

10. Understand Why You're Drawing a Sequence Diagram

When drawing sequence diagrams, you're meticulously exploring the ins and outs of the **detailed design** for each use case, in microscopic detail. This means exploring not just the basic course, but also **all the alternate courses of action in each use case**. (We can't stress this point strongly enough!)

It's surprising how many design issues can be caught at this stage, saving time on refactoring your design later, so it pays to design and explore every facet of each use case, not just the sunny-day scenario.

Sequence diagramming has three primary goals in ICONIX Process:

- **Allocate behavior to your classes**: You identified these classes during robustness analysis. During sequence diagramming, the controllers (also discovered during robustness analysis) are turned into operations on the classes. However, you don't necessarily end up with a 1:1 correlation between the controllers and the operations. Often, a controller turns into two or more operations. (Check the examples later in this chapter to see how this happens.) And as we mentioned earlier, occasionally a controller may also be turned into a **controller class**.

- **Show in detail how your classes interact with each other over the lifetime of the use case**: When sequence diagramming, you should be exploring **how** the system will accomplish the behavior described in your use cases. You do this by thinking about and then depicting how your **objects** (runtime instances of a class) will interact with each other at runtime.

- **Finalize the distribution of operations among classes**: Having performed robustness analysis, you should by now have identified at least three-quarters of the attributes (the **data**) on your classes, but very few, if any, of the operations (the **behavior**). By now you've probably gathered that we advocate a two-pass approach to the design. The first pass (preliminary design) is driven by thinking about attributes while deliberately ignoring "who's doing what to whom." Then the second pass (the subject of this chapter) focuses all your attention on that exact question. That's because during preliminary design, the information just wasn't there to allocate operations without guessing. However, now that you're at the detailed design stage, you should have everything in place to correctly allocate the behavior among your classes.

Note Objects interact by sending messages to each other. In the Ruby programming language, the *message* paradigm is used literally, and all object interactions are considered to be messages. However, in other languages such as Java or C++, the messages you draw on sequence diagrams equate to method or function calls. To complicate things a little, in UML each message is also called an **operation** once it's assigned to a class.

Note Messages, methods, functions, operations, verbs, and controllers—these are all basically different versions of the same thing: the behavior that you allocate to a class (via sequence diagramming) and eventually implement and test.

DON'T TRY TO DRAW FLOWCHARTS ON SEQUENCE DIAGRAMS (FOCUS ON BEHAVIOR ALLOCATION INSTEAD)

UML 2.0 allows you to draw full-blown flowcharts on your sequence diagrams. However, even though the notation supports it, we consider the practice of drawing flowcharts on sequence diagrams to be inadvisable, because it puts emphasis on the wrong part of the problem.

In large part this is because ***drawing flowcharts simply misses the point of what you should be thinking about when you draw a sequence diagram***. If you're trying to drive a software design from use cases, it's vitally important to get the allocation of operations to classes correct. This allocation of operations to classes tends to be a make-or-break design issue.

In ICONIX Process, the primary purpose of the sequence diagram is to make this behavior allocation visible so that you get it right. If your mind is on drawing a flowchart, it's not going to be focused on this critically important set of behavior allocation decisions.

9. Do a Sequence Diagram for Every Use Case

It's pretty simple to make sure you've covered everything in your design, if you stick to these two simple rules:

- Write a use case for every scenario you're going to build in your current release (include basic and alternate courses in each use case).
- Draw a sequence diagram for each use case and use the sequence diagram to put the operations on the classes.

Simple but effective. You've then allocated all the software behavior you need, and presumably nothing you don't need, into your classes.

Tip When you're starting a sequence diagram, the very first thing you should do is paste the text of the use case into a Note on the left margin.

One question that we often get asked is, "Should I have a separate sequence diagram for each alternative course of action?" This question is frequently posed by people who have enormous use case templates and ten-page use cases.

Our preference is to keep the use case short (see the two-paragraph rule in Chapter 3), and thus be able to show the entire use case (sunny- and rainy-day scenarios) on a single sequence diagram. It's too easy to lose track of one or two alternate courses of action if you split up the use case and sequence diagrams. And losing track of alternate courses of action tends to be problematic.

8. Start from Where You Left Off with Robustness Analysis

You identified which objects will be collaborating together on the sequence diagram when you drew your robustness diagram. If your use case has a GUI, the boundary objects will represent screens and other UI elements. Entity classes from the domain model will collaborate with the GUI objects.

In many cases, controllers from the robustness diagram will not actually be "real controller objects" on a sequence diagram, although in some cases they will be. In the remaining cases, controllers will map to messages between objects on the sequence diagram.

Keep in mind that your sequence diagram shows the design at a much more concrete and detailed view than the idealized conceptual design shown on a robustness diagram. So your sequence diagram may show additional helper infrastructure objects, details of persistence storage mechanisms, and so forth.

7. Show How the Use Case's Behavior Is Accomplished by the Objects

Controllers on a robustness diagram generally map to "logical" software functions. Each logical function may be realized by one or more messages between objects that will be shown on the sequence diagram. As you draw messages, you are actually allocating operations to classes, so the sequence diagram is "about" allocating behavior among collaborating objects.

EXAMPLE SEQUENCE DIAGRAM: DISPLAYING THE BOOK DETAILS PAGE

Our *Write Customer Review* use case example (which we'll return to shortly) gets quite involved at the design stage, as it uses lots of different parts of Spring Framework, JSP, and our own controller logic. So it makes sense to ease you into sequence diagramming (and Spring Framework) with a simple example first.

We'll start out with a very small use case, *Show Book Details*. The use case text is as follows:

BASIC COURSE:
The Customer types in the URL for the bookstore's home page, which the system displays. Then the Customer clicks a link to view a Book. The system retrieves the Book details and displays the Book Details screen.
ALTERNATE COURSES:

Book not found: The system displays a Book Details Not Found screen.

We weren't kidding when we said this is a simple use case! Really, this barely qualifies as a use case (remember the two-paragraph rule from Chapter 3). However, it suffices for our purpose, which is to illustrate how we get from the use case to a finished sequence diagram.

We showed the robustness diagram for this use case back in Figure 5-11. The accompanying sequence diagram is shown in Figure 8-2 (the part of the diagram that handles the alternate course is shown in red).

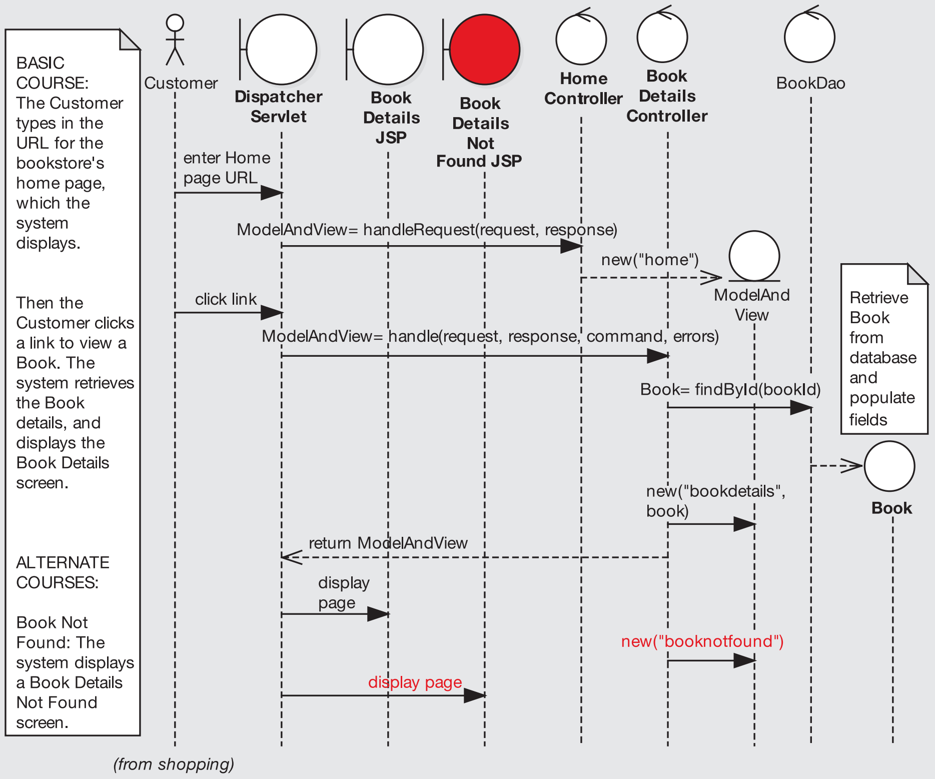

BASIC COURSE: The Customer types in the URL for the bookstore's home page, which the system displays.
Then the Customer clicks a link to view a Book. The system retrieves the Book details, and displays the Book Details screen.
ALTERNATE COURSES:
Book Not Found: The system displays a Book Details Not Found screen.
Customer
Dispatcher Servlet
Book Details JSP
Book Details Not Found JSP
Home Controller
Book Details Controller
BookDao
enter Home page URL
ModelAndView= handleRequest(request, response)
new("home")
ModelAnd View
click link
ModelAndView= handle(request, response, command, errors)
Book= findById(bookId)
Retrieve Book from database and populate fields
Book
new("bookdetails", book)
return ModelAndView
display page
new("booknotfound")
display page
(from shopping)

Figure 8-2. *Sequence diagram for the* Show Book Details *use case*

The processing in this diagram is quite detail-rich, so we've lumped it into Appendix B (in the section titled "'Show Book Details' Use Case"). If you're interested in the Spring plumbing, please do take a look there before moving on.

Later in this chapter, we walk through a step-by-step example of drawing a sequence diagram directly from its matching robustness diagram. As you'll see, when done properly, it's really a mechanical process allowing us to focus on the design details.

■Exercise The Book class in Figure 8-2 doesn't appear to do a huge amount, except exist. This doesn't seem especially domain-oriented, as domain classes are meant to contain both data and behavior. Looking at Figure 8-2, how could the design be improved to give Book a more prominent role?

■Note Although the design shown in Figure 8-2 seems fine on the surface—and (we regret to say) a *lot* of web-based applications are created this way—it actually violates a key principle of the ICONIX approach, namely that ***the domain class should be the central point of responsibility for operations relating to that domain class***. For example, all of the responsibilities related to a Book Review ought to be located in the Book Review domain class, even if that class delegates some of those responsibilities to helper classes.

Violating this principle often results in several superfluous one-line classes and no obvious starting point if you're trying to track something down in the code. We review this diagram in Chapter 9 to hammer the design into shape.

6. Map Your Use Case Text to the Messages on the Sequence Diagram

The use case text that appears on the left margin of the sequence diagram is really a contract between clients and programmers. It should unambiguously specify the runtime behavior requirements that satisfy the customer's needs. Since we have this contract, it makes sense to use the sequence diagram to show graphically how the design meets the behavior requirements. In fact, reading your sequence diagram should give you a visual trace from the design back to the requirements.

Whenever possible, try to make the message arrows and the use case text line up visually. This makes it much easier to review the diagrams and verify that the design meets the requirements.

5. Don't Spend Too Much Time Worrying About Focus of Control

Focus of control refers to the little rectangles that magically appear on the object lifelines when using a visual modeling tool. They indicate which object "has the focus," or is currently in control (see Figure 8-1).

In theory, this is a very useful thing to display on a sequence diagram. But in practice, with many visual modeling tools, it can be really annoying to try to get the focus-of-control rectangles to behave the way you want them to. Once again, this can become a distraction from the real purpose of the sequence diagram, which is (repeat after us) *allocation of behavior among collaborating objects*.

If focus of control starts to become a distraction, the easiest thing to do is just switch it off. You might find that your diagrams are cleaner and more readable without this extra detail anyway.

■Tip Using EA, simply right-click the diagram and choose Suppress Focus of Control.

ARE SEQUENCE DIAGRAMS DIFFICULT TO DRAW?

Many people find sequence diagrams rather tortuous to draw for the following reasons:

- Sequence diagrams can be difficult to draw and edit in some CASE tools (especially if the **focus of control** is visible).
- The diagrams are densely packed with information, which can make them as difficult to read as to draw, unless they're carefully organized.
- The people who draw sequence diagrams often don't really know *why* they're drawing them or what they're supposed to be getting out of the diagram.

To make matters worse, this is the time when you're also making the really hard design decisions—when you are presented with a bunch of analysis documents and are expected to somehow magically transform them into a detailed design that captures every last nuance of the business analyst's (often abstractly worded) requirements.

One issue we have with many approaches to UML is that there's generally a lack of emphasis on preliminary design, so developers must take a giant leap of faith between analysis and design, assuming (or praying) that the resultant design will somehow match up with the use cases. This is why many people find sequence diagrams so difficult to draw: they're trying to answer too many questions (i.e., juggle too many dust-bunnies) all at once.

ICONIX Process, however, is specifically geared toward laying the groundwork for sequence diagramming, so **if you've performed robustness analysis properly, drawing the sequence diagrams should be significantly easier**.

To answer the three issues we just listed:

- **Sequence diagrams are difficult to draw using a CASE tool**: This tends to be the case when someone doesn't really understand what it is he is trying to draw, so he spends most of his time struggling with the sequence diagram notation, wondering why the sequence diagram editor won't allow him to just draw whatever he wants. We've observed in practice that this frustration happens most often when ***the use case the person is trying to nail down with a sequence diagram is vague and ambiguous***, and he has to keep changing it around as he's in the middle of drawing the sequence diagram. The CASE tool often takes the blame for this frustration. At any rate, as you already know, we're strong advocates of completely and thoroughly disambiguating the behavior requirements (see Chapter 5) before starting detailed design.
- **Sequence diagrams are densely packed with information**: You do need to show lots of detail on each sequence diagram, because it's describing in detail ***how*** the use case behavior is going to be implemented. If necessary, you can split the diagram onto more than one page, if it makes it easier to draw; but even better, learn to keep your use cases short by factoring them on the use case diagrams using invokes and precedes.
- **Sequence diagram authors are unclear on their purpose and goals**: Having a clear idea of what you're trying to achieve when you draw a diagram definitely makes life easier. See "Understand Why You're Drawing a Sequence Diagram" earlier in this chapter.

Separating the "what" from the "how" (see Figure 5-1) is an essential task in software development, as is having a clear transition between them.

4. Assign Operations to Classes While Drawing Messages

Generally speaking, there are two ways to label messages on a sequence diagram:

- By simply typing a label on the arrow
- By using an explicit command such as a right-click or button-press near the arrowhead

The second option directly adds an operation to the class of the target. It's important to use the sequence diagram to drive the allocation of operations to classes. So, if that's your intent when drawing a message, make sure to use the second option.

3. Review Your Class Diagrams Frequently While You're Assigning Operations to Classes

Since you're now actively assigning operations to classes, and since it's very easy to make mistakes while drawing sequence diagrams, you should continually cross-check your sequence diagram and your class diagram to make sure that when you assign an operation to a class, you've done it in a way that makes sense.

An excellent reference that can help you to make good design decisions is Rebecca Wirfs-Brock's book *Object Design: Roles, Responsibilities, and Collaborations* (Addison-Wesley, 2002). This book teaches a very useful technique called *Responsibility-Driven Design* (we describe this technique in a little more detail later in this book).

2. Prefactor Your Design on Sequence Diagrams

Prefactoring your design on sequence diagrams saves massive amounts of refactoring after code. Many times, the need for code-level refactoring is the result of the programmer making suboptimal behavior allocation decisions. Refactoring techniques such as "Move Method," "Replace Method with Method Object," and (deep breath) "Consolidate Duplicate Conditional Fragments"[1] involve moving methods around among classes. The sequence diagram should be used as a tool to help you to *prefactor* your design and make these behavior allocation decisions correctly, **before** going to all the trouble of coding and unit testing.

If you use a sequence diagram for this purpose, you'll find that you spend a lot less time using refactoring techniques to fix the mistakes that you avoided making in the first place. The lost art of getting the design right the first time can still be practiced (and quite successfully) using sequence diagrams. And you don't even have to be a Druid to do it.[2]

1. Clean Up the Static Model Before Proceeding to the CDR

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 get started thinking at this level in the design even *before* drawing your sequence diagrams.

After you complete *preliminary* design, the class model should have fewer holes in it than it did before robustness analysis. Chances are that some additional classes have been discovered that were missing from the original domain model, and many of the classes

1. See *Refactoring: Improving the Design of Existing Code* by Martin Fowler (Addison-Wesley, 2000).
2. See our next book, *Prefactoring the Druid Way*. No goats required.

should be populated with attributes. However, even when you consider that the system behavior will be allocated to the classes as message arrows are drawn on the sequence diagrams, in most cases the class model will require additional work to complete it to the point that it's ready to code from.

Here are a few examples of the sorts of things that you may still need to deal with:

- Using infrastructure/scaffolding classes
- Using design patterns
- Meshing the design with application frameworks
- Completing parameter lists on operations
- Etc.

So, **plan on finalizing your class model during detailed design** along with doing your sequence diagrams. It's often a good idea to do a lot of this finalizing *before* the sequence diagram is drawn, so that the sequence diagram simply represents the way the code is going to work. An alternative strategy is to draw the sequence diagram at a slightly simplified level and then finalize the class model in the coding environment.

Tip New tools, such as MDG Integration from Sparx Systems (`www.sparxsystems.com`), make the synchronization task between model and code orders of magnitude easier than it used to be.

Now that we've described the **notation** you use to draw sequence diagrams, and we've examined the top 10 guidelines of sequence diagramming, we'll walk through the steps involved in drawing a sequence diagram effectively.

How to Draw a Sequence Diagram: Four Essential Steps

We've distilled the process of drawing a sequence diagram to four essential steps, which we describe in this section. We illustrate these steps by drawing the sequence diagram for the *Write Customer Review* use case.

The first three steps are completely mechanical in nature. This means they can be automated, which can be very useful in achieving momentum as you get serious about your design. The fourth step, **deciding which methods go on which classes**, is really what sequence diagramming is all about.

Figure 8-3 shows the four steps you perform when drawing sequence diagrams the ICONIX way. Next, we describe the four steps in more detail. To illustrate the steps, we return to the *Write Customer Review* use case we disambiguated using robustness analysis in Chapter 5.

Note In Chapter 12, we extend these steps to show how to systematically build a suite of test cases at the same time as your sequence diagrams (see Figure 12-2).

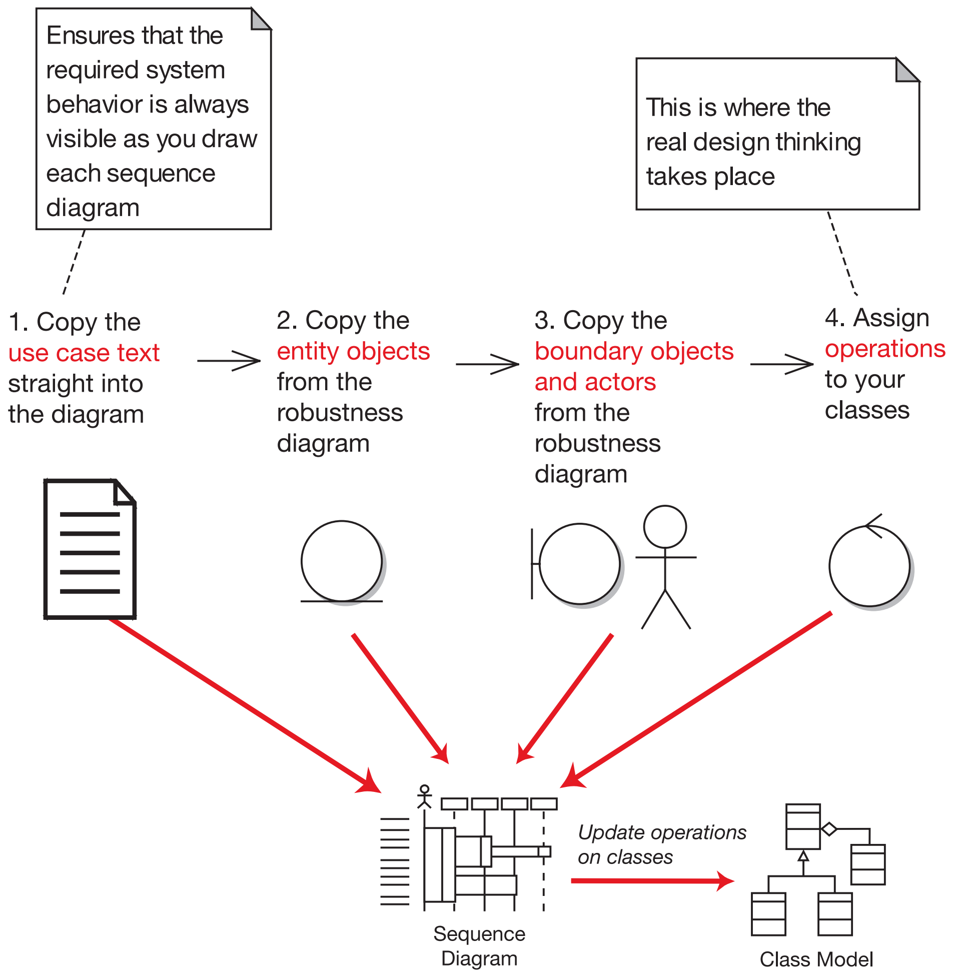

Ensures that the required system behavior is always visible as you draw each sequence diagram
This is where the real design thinking takes place
1. Copy the use case text straight into the diagram
2. Copy the entity objects from the robustness diagram
3. Copy the boundary objects and actors from the robustness diagram
4. Assign operations to your classes
Update operations on classes
Sequence Diagram
Class Model

Figure 8-3. *Building a sequence diagram in four essential steps*

IF THE STEPS ARE MECHANICAL, WHY NOT AUTOMATE THEM?

In fact, we've turned the first three steps of our process for drawing a sequence diagram into an executable script that automatically generates a skeleton of a sequence diagram. We originally did this some years ago for Rational Rose, and if you use Rose, you can download a copy of this script here: `www.iconixsw.com/RoseScripts.html`. More recently, we've created an upgraded version for the EA tool from Sparx Systems. The upgraded version also generates test cases for each controller on the robustness diagram (see Figure 12-7) and is available as an add-in on Doug's CD "Enterprise Architect for Power Users." (More information can be found at `www.iconixsw.com/EA/PowerUsers.html`.)

Scripts and add-ins such as this have proven to be very useful in achieving momentum as you get serious about your design. You get an immediate payback in time savings from the work you invested in your robustness diagrams, which can help to get some buy-in to the process from your team.

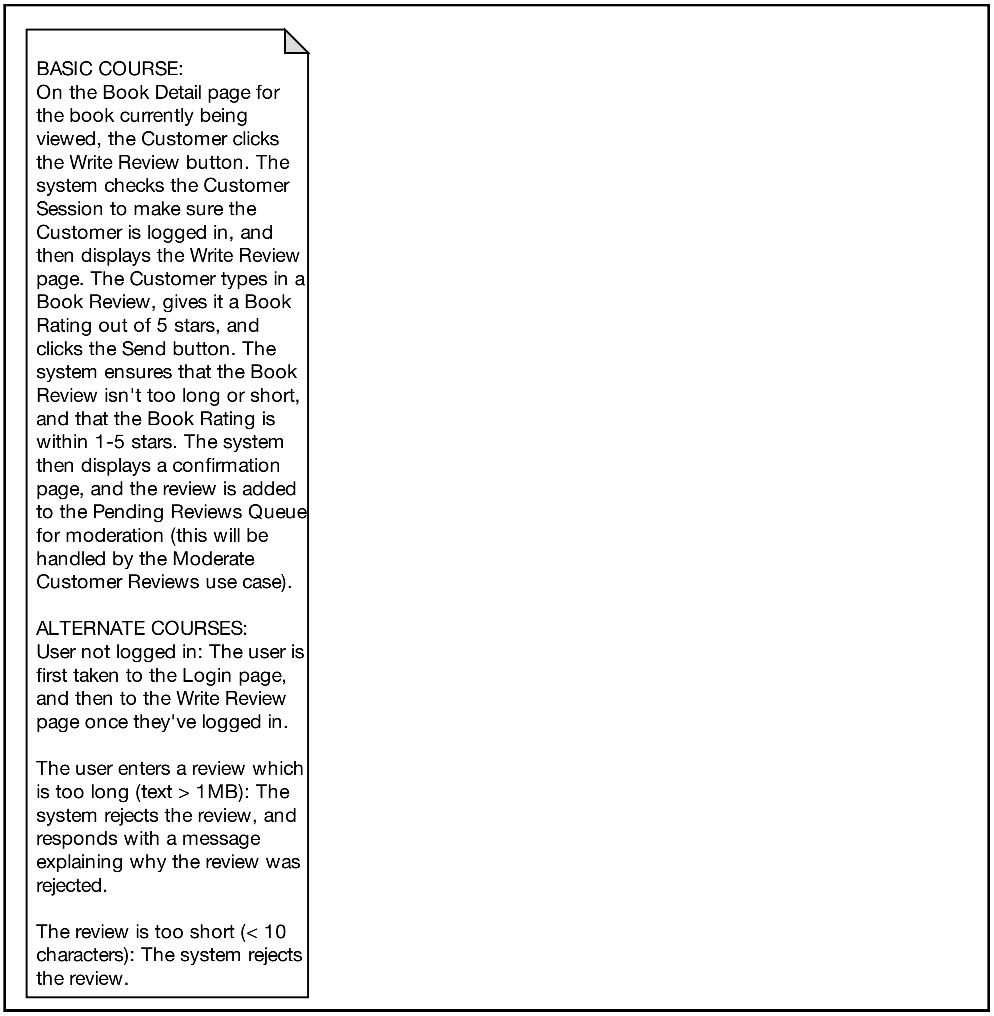
Step 1: Copy the Use Case Text Straight into the Diagram

After all the preparatory steps that you've gone through so far, the use case text should be remarkably fit and healthy. Having put all that effort into writing disambiguated use cases that you can design from, it makes sense to place the use case text directly on the design diagram so that, as you're doing the design, ***the required system behavior is right there on the diagram too***. The initial version of the *Write Customer Review* sequence diagram, with just this first step completed, is shown in Figure 8-4.

Remember that each use case is, essentially, a mini-contract that should have gotten sign-off from the project stakeholders. And because each use case is quite fine-grained (two paragraphs at most), it's a small capsule of discrete, self-contained functionality. Having the text there on the screen helps you to keep the sequence diagram focused on **just the steps described in those two paragraphs**.

Also remember that the robustness diagram you drew was an object picture of the given use case, and the process of drawing the robustness diagram caused you to rewrite the use case so that it uses the domain object names literally in the text. As a result, when you're designing, you should be able to look at the use case text and (when you get really good at it) visualize the named objects interacting with each other. For example, the text "The system places the Book Review on the Reviews Pending Queue" strongly suggests that there will be a `BookReview` class, a `ReviewsPendingQueue` class, and a message between the two probably called `addToQueue(bookReview)`.

■**Note** Because you're designing directly from the use case text, then it follows that if you didn't analyze the use case in detail and write down all its alternate courses, you shouldn't be doing detailed design yet. Designing from an incomplete use case means that you won't discover all of the necessary methods for your objects.



BASIC COURSE:
On the Book Detail page for the book currently being viewed, the Customer clicks the Write Review button. The system checks the Customer Session to make sure the Customer is logged in, and then displays the Write Review page. The Customer types in a Book Review, gives it a Book Rating out of 5 stars, and clicks the Send button. The system ensures that the Book Review isn't too long or short, and that the Book Rating is within 1-5 stars. The system then displays a confirmation page, and the review is added to the Pending Reviews Queue for moderation (this will be handled by the Moderate Customer Reviews use case).

ALTERNATE COURSES:
User not logged in: The user is first taken to the Login page, and then to the Write Review page once they've logged in.

The user enters a review which is too long (text > 1MB): The system rejects the review, and responds with a message explaining why the review was rejected.

The review is too short (< 10 characters): The system rejects the review.

Figure 8-4. *Building a sequence diagram, step 1*

Step 2: Copy the Entity Objects from the Robustness Diagram

Assuming you updated your static model during robustness analysis, the entity objects should now each have an equivalent class on one of the class diagrams (see Figure 8-5). (And if you *haven't* updated your static model yet, **be sure to do it now**.) The updated *Write Customer Review* sequence diagram is shown in Figure 8-6.

Until now, the entity objects have been little more than simple bags of data with lots of attributes but no behavior (read: no personality). This will change during sequence diagramming. Now that they're on the diagram, you'll soon start to allocate operations to them—in effect, assigning their classes with behavior. But first you need to get the remaining "nouns" (the boundary objects and actors) onto the diagram.

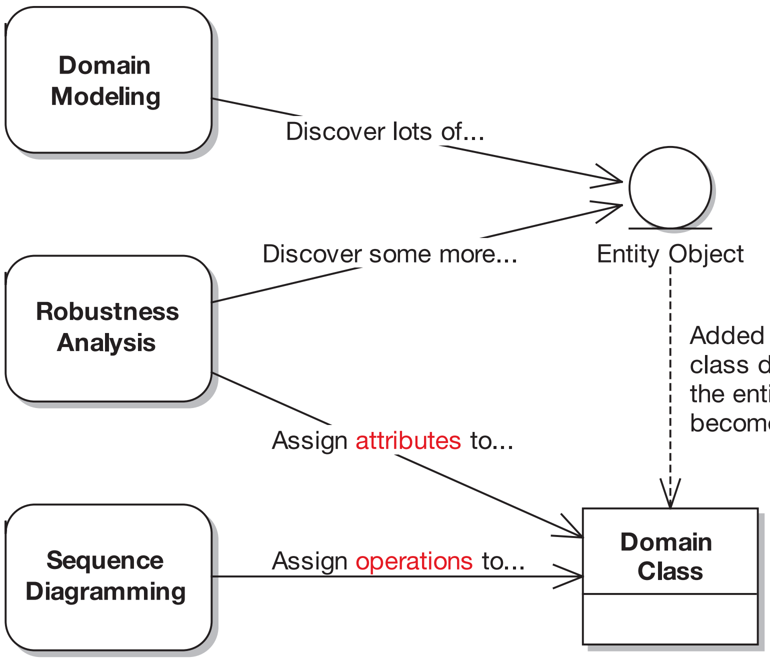


Figure 8-5. *How your entity objects evolve into full-fledged classes*

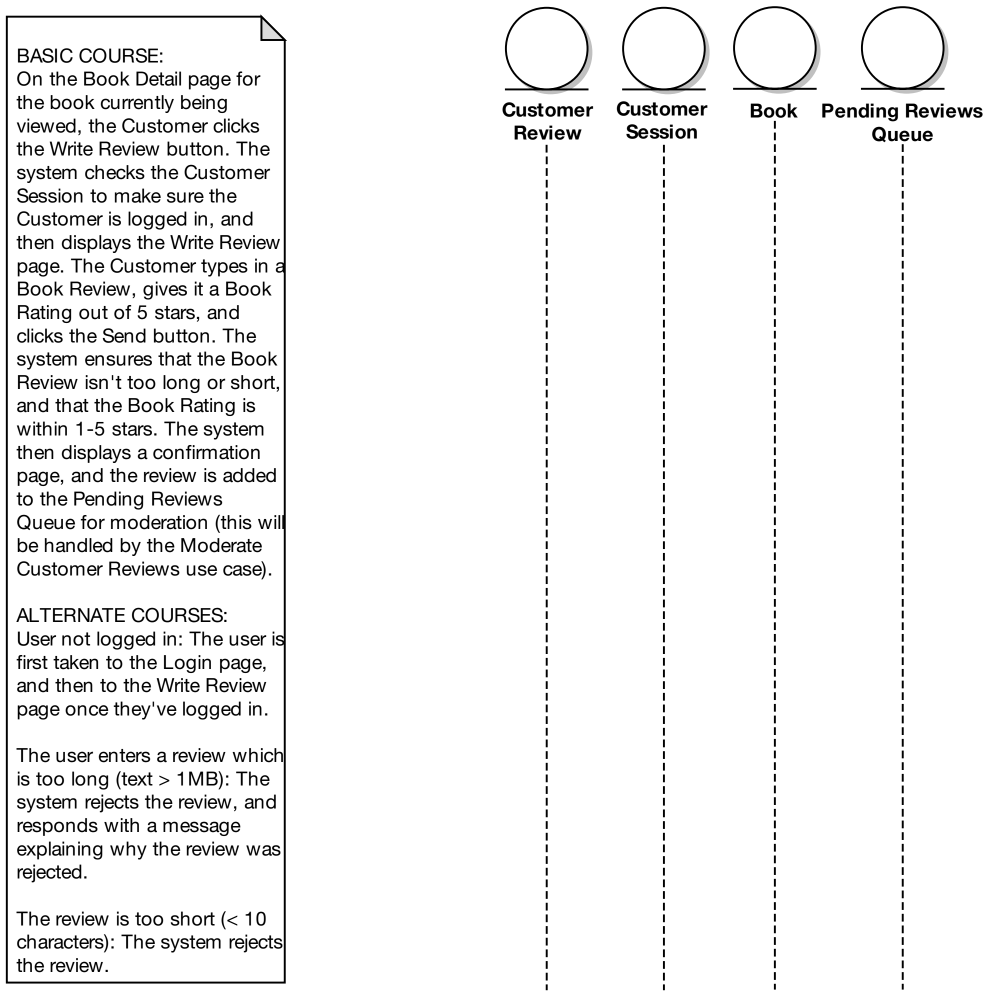

BASIC COURSE:
On the Book Detail page for the book currently being viewed, the Customer clicks the Write Review button. The system checks the Customer Session to make sure the Customer is logged in, and then displays the Write Review page. The Customer types in a Book Review, gives it a Book Rating out of 5 stars, and clicks the Send button. The system ensures that the Book Review isn't too long or short, and that the Book Rating is within 1-5 stars. The system then displays a confirmation page, and the review is added to the Pending Reviews Queue for moderation (this will be handled by the Moderate Customer Reviews use case).
ALTERNATE COURSES:
User not logged in: The user is first taken to the Login page, and then to the Write Review page once they've logged in.
The user enters a review which is too long (text > 1MB): The system rejects the review, and responds with a message explaining why the review was rejected.
The review is too short (< 10 characters): The system rejects the review.
Customer Review
Customer Session
Book
Pending Reviews Queue

Figure 8-6. *Building a sequence diagram, step 2*

Step 3: Copy the Boundary Objects and Actors from the Robustness Diagram

See Figure 8-7. The boundary objects and actors are the remaining "nouns" (see Chapter 5). There can be more than one actor on the sequence diagram, although typically there's only one, and it usually goes on the left of the diagram. The updated *Write Customer Review* sequence diagram is shown in Figure 8-8.

You win the ripest piece of fruit in the fruit bowl if you wondered why we're adding the boundary objects and actors from the robustness diagram and not from the domain model. In fact, the boundary objects and actors were never added to the domain model, as they're part of the solution space (the "how"). By contrast, **the entity objects that we've added *were* on the domain model**, as they're part of the problem space (the "what").

Depending on the type of GUI you're creating, the boundary classes usually turn into JSP or ASP pages. So you have the option to treat the boundary classes as "not real classes," and not allocate behavior to them. How GUIs are treated varies widely if you're doing JSP, ASP.NET, HTML, or a desktop application (for example). So during sequence diagramming, it's better to focus on allocating behavior into the domain/entity classes (see step 4).

Note We're not saying that you should *never* add attributes or operations to your view/boundary classes, but as a general rule, these classes (or pages) tend not to do any of their own processing. Your own experiences may vary depending on the GUI or web-application toolkit that you're using.

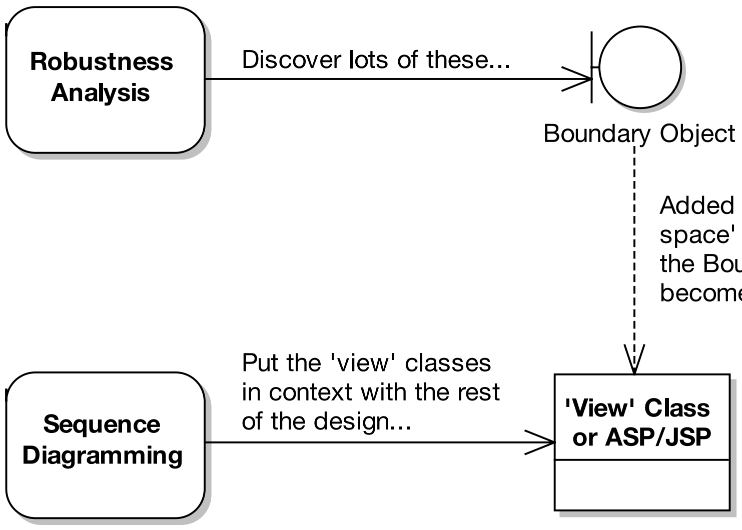

Robustness Analysis
Discover lots of these...
Boundary Object
Added to the 'solution space' class diagram, the Boundary Object becomes...
Sequence Diagramming
Put the 'view' classes in context with the rest of the design...
'View' Class or ASP/JSP

Figure 8-7. *How your boundary objects evolve into view classes or view pages (e.g., JSP pages)*

Tip It helps to maintain a pure domain model diagram, containing only the entity classes (but not showing any attributes or operations). But at some point (right about now, in fact, during detailed design), you'll need to draw some more detailed class diagrams that show both *solution space* classes and *problem space* classes. As you'll end up with some very large detailed class diagrams, you should split them up—for example, have one diagram per use case package. (Refer to Chapter 3 for a discussion of use case packages.)

Tip And another thing: These detailed class diagrams should use the same elements as on the sequence diagrams, so when you assign a message on the sequence diagram (the dynamic model), an operation is automatically added to the appropriate class in the static model.

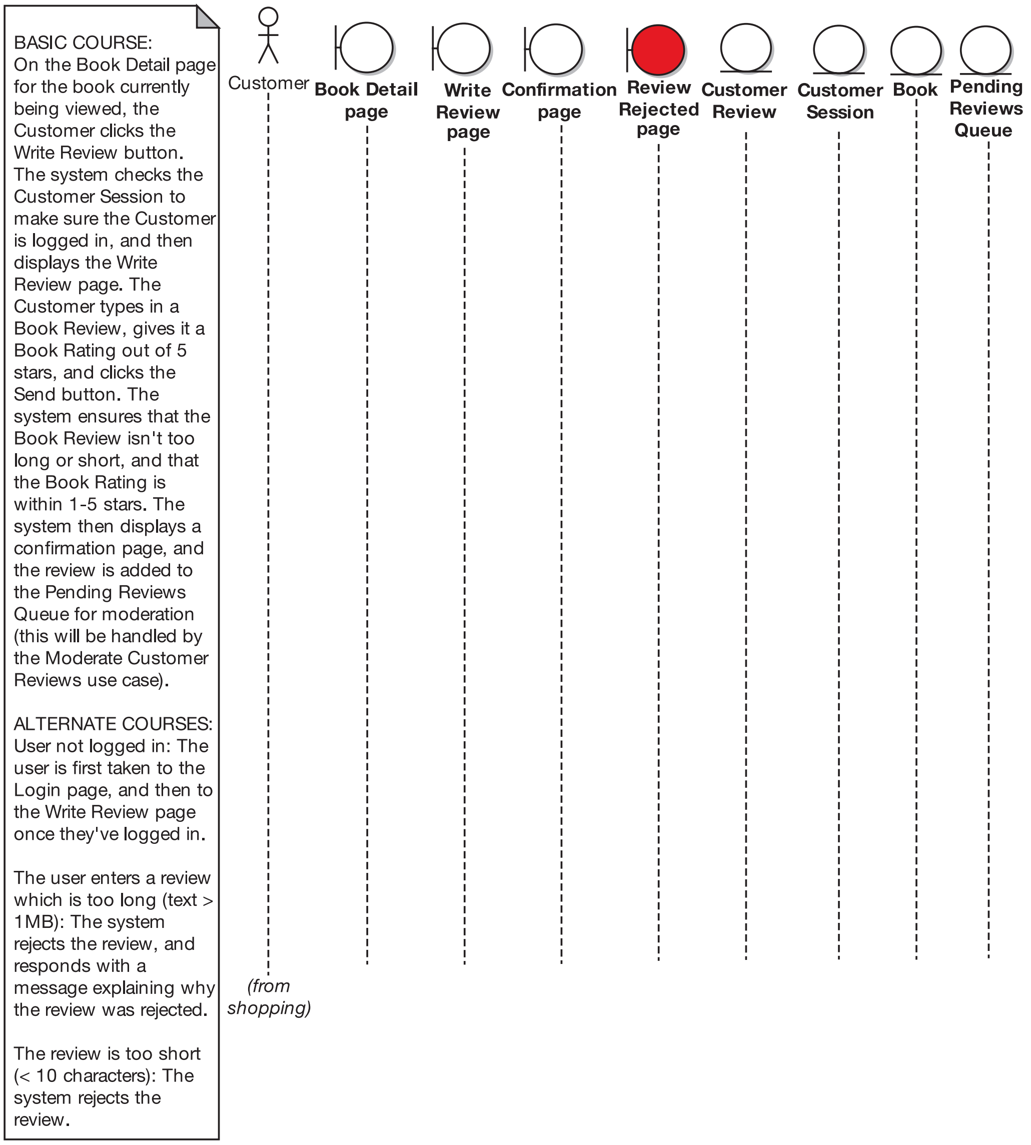

BASIC COURSE:
On the Book Detail page for the book currently being viewed, the Customer clicks the Write Review button. The system checks the Customer Session to make sure the Customer is logged in, and then displays the Write Review page. The Customer types in a Book Review, gives it a Book Rating out of 5 stars, and clicks the Send button. The system ensures that the Book Review isn't too long or short, and that the Book Rating is within 1-5 stars. The system then displays a confirmation page, and the review is added to the Pending Reviews Queue for moderation (this will be handled by the Moderate Customer Reviews use case).
ALTERNATE COURSES:
User not logged in: The user is first taken to the Login page, and then to the Write Review page once they've logged in.
The user enters a review which is too long (text > 1MB): The system rejects the review, and responds with a message explaining why the review was rejected.
The review is too short (< 10 characters): The system rejects the review.
Customer
(from shopping)
Book Detail page
Write Review page
Confirmation page
Review Rejected page
Customer Review
Customer Session
Book
Pending Reviews Queue

Figure 8-8. *Building a sequence diagram, step 3*

Once you've finished these steps, you're over the hump of getting your design started, and it's time to move on to the real work.

Step 4: Assign Operations to Your Classes

This step is where the real decision-making takes place (see Figure 8-9). Up until now, everything has been rather mechanical and aimed at prepping the sequence diagram so that the highly skilled surgeon (that's you) can come in and amaze everyone with his or her precise and scrupulous design work. A surgeon must make some high-pressure decisions during an operation (we should know, we've seen reruns of *Dr. Kildare*), and similarly, when you're designing software, you're faced with difficult decisions that mostly shouldn't be put off until later. You've done all the preparatory work leading up to this point, and this next step is the crux, the essence of detailed design.

Unfortunately, as you may have gathered, this step is also pretty hard (at least until you get the hang of it). Every decision, large and small, really counts. **Experience and talent are required to do a good job at detailed design.**

What you've done with this process so far is to clear the way so that, during detailed design, there are no distractions—all you need to think about is the design itself. But you still need to do the actual thinking, of course. On the bright side, the more you do it, the better you'll become. As your experience grows, you'll find it becomes easier to make good design decisions.

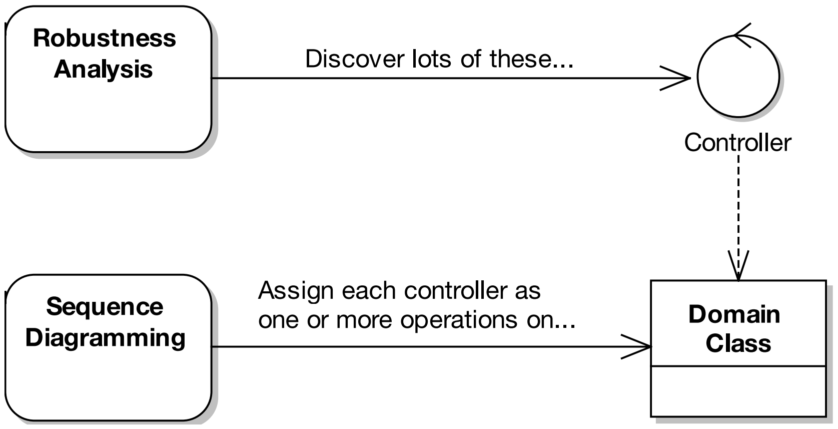

Robustness Analysis
Discover lots of these...
Controller
Sequence Diagramming
Assign each controller as one or more operations on...
Domain Class

Figure 8-9. *How your controllers evolve into operations on your domain classes*

The best place to get started with behavior allocation is to convert the controllers from the robustness diagram.

Converting the Controllers from Your Robustness Diagram

To allocate operations/methods to your classes, you need to convert the controllers from the robustness diagram into messages on your sequence diagram. It's possible to do this systematically. Step through each controller on the robustness diagram. For each one, draw the corresponding message(s) on the sequence diagram, and then check the controller off (rather like a checklist) and move on to the next controller.

■Tip Remember that when you step through each controller on the robustness diagram, each message should automatically be turned into an operation (or several operations) on the appropriate class(es). on the appropriate class. Make your CASE tool work harder, so that you can work smarter!

Remember that you've already checked the robustness diagram against the use case text. So by using the robustness diagram as a checklist for your sequence diagram, you're providing a level of assurance that you are designing precisely what the user needs—in other words, ***you're ensuring that your design matches up with the requirements***.

Although we recommend turning each controller into an operation, every now and then it makes sense to turn a controller into a full-fledged control class. How often this happens depends on whether you're building a real-time embedded system and whether the framework you're using more-or-less demands that you have lots of itty-bitty controller classes all over the place.

Tip Have both the sequence diagram and any relevant class diagrams at hand. You should be flipping between the sequence and class diagrams frequently while you assign operations and think about the design. It's a two-way process: sometimes you make a change to the class diagram and then feed the change back into the sequence diagram, and vice versa.

During this stage in the design, it pays to have a catalog of well-established design patterns to fall back on. The "classic set" of patterns can be found in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995). If you're developing enterprise software, we can also recommend *Patterns of Enterprise Application Architecture* by Martin Fowler (Addison-Wesley, 2002). Although the book's title suggests that it's about architecture, the patterns it describes are really design patterns for the enterprise.

However, it's also important to not become too obsessed with design patterns. Always try to remember that they're there for guidance—a starting point for when doing your own detailed designs. We've seen the unfortunate results of many projects that have become too complex for their own good, because of a tendency to overapply design patterns. As Groucho Marx once said, "I like my cigar, too, but I take it out of my mouth once in a while!"

Deciding Which Controllers to Assign to Which Classes

This is the big one, of course. To decide which controllers go on which classes, you must answer two questions:

- What are the classes?
- Which classes are *responsible* for which operations?

The second question has as its roots Rebecca Wirfs-Brock's Responsibility-Driven Design (RDD). A **responsibility** is a combination of things: knowledge or data (the attributes), behavior (the operations), and the major decisions made at runtime that affect others.

Usually, you'll find that the decision of which responsibilities to put in which classes is clear-cut. For example, it should be obvious that a `BookReview` will be responsible for validating its own data. Sometimes, however, the decision isn't quite so clear-cut. Responsibilities can be hazy or too complex to be comfortably managed by a single class. In *Object Design:*

Roles, Responsibilities and Collaborations, Wirfs-Brock describes how complex responsibilities should be allocated:[3]

> *An object has three options for fulfilling any responsibility. It can either:*
>
> ** Do all the work itself*
>
> ** Ask others for help doing portions of the work (collaborate with others)*
>
> ** Delegate the entire request to a helper object*
>
> *When you're faced with a complex job, ask whether an object is up to this responsibility or whether it is taking on too much. A responsibility that is too complex to be implemented by a single object essentially introduces a new sub design problem. You need to design a set of objects that will collaborate to implement this complex responsibility. These objects will have roles and responsibilities that contribute to the implementation of the larger responsibility.*

As we described earlier, assigning responsibilities to each class involves giving your class some *personality*, but it's also possible for a class to become overallocated and end up with perhaps a little too *much* personality. Dan Rawsthorne (a former ICONIX instructor) once summarized the responsibility-driven thought process to Doug as follows:

> *If you think of your objects (or classes) as people, then the set of behaviors (operations) that they are responsible for gives them a sort of personality. We want to* watch out for schizophrenic objects *(that is, objects with split or multiple personalities) because an object should be focused on a cohesive, related set of behaviors.*

A class with a split personality—or multiple personalities—is a class that has more than one responsibility. A clear sign of such a troubled class is if it contains *discordant* attributes—that is, attributes that don't seem to fit in with their peers. If you discover discordant attributes, use **aggregation** (see Chapter 2) to move them into separate, more appropriate classes. Most likely, these new classes will need to collaborate with each other.

Four Criteria of a Good Class

While you are making behavior allocation decisions, you are making decisions that affect the quality of the classes in your design. Grady Booch's *Object-Oriented Analysis and Design with Applications* (Addison-Wesley, 1994) introduces the *Halbert/O'Brien criteria* of reusability, applicability, complexity, and implementation knowledge.

If you follow the RDD thought process, you'll generally wind up with a set of behaviors that are all applicable (i.e., not schizoid), don't depend on implementation details of another

3. Rebecca Wirfs-Brock and Alan McKean, *Object Design: Roles, Responsibilities and Collaborations* (New York: Addison-Wesley, 2003), p. 132

class, aren't overly complex for any single class, and result in a reusable piece of code that maps nicely to a problem-domain abstraction.

Following a responsibility-driven thought process (done correctly) results in classes that meet the Halbert/O'Brien quality criteria. Or, to go back another 15 years or so: maximize cohesion, minimize coupling.

Continuing the Internet Bookstore Example

To illustrate the theory step by step, let's return to our Internet Bookstore example and the *Write Customer Review* use case. So far we've added the entities, actor, and boundary objects to our sequence diagram (see Figure 8-8). The next step, as described in the previous section, is to walk systematically through the controllers on the robustness diagram and apply them to the sequence diagram.

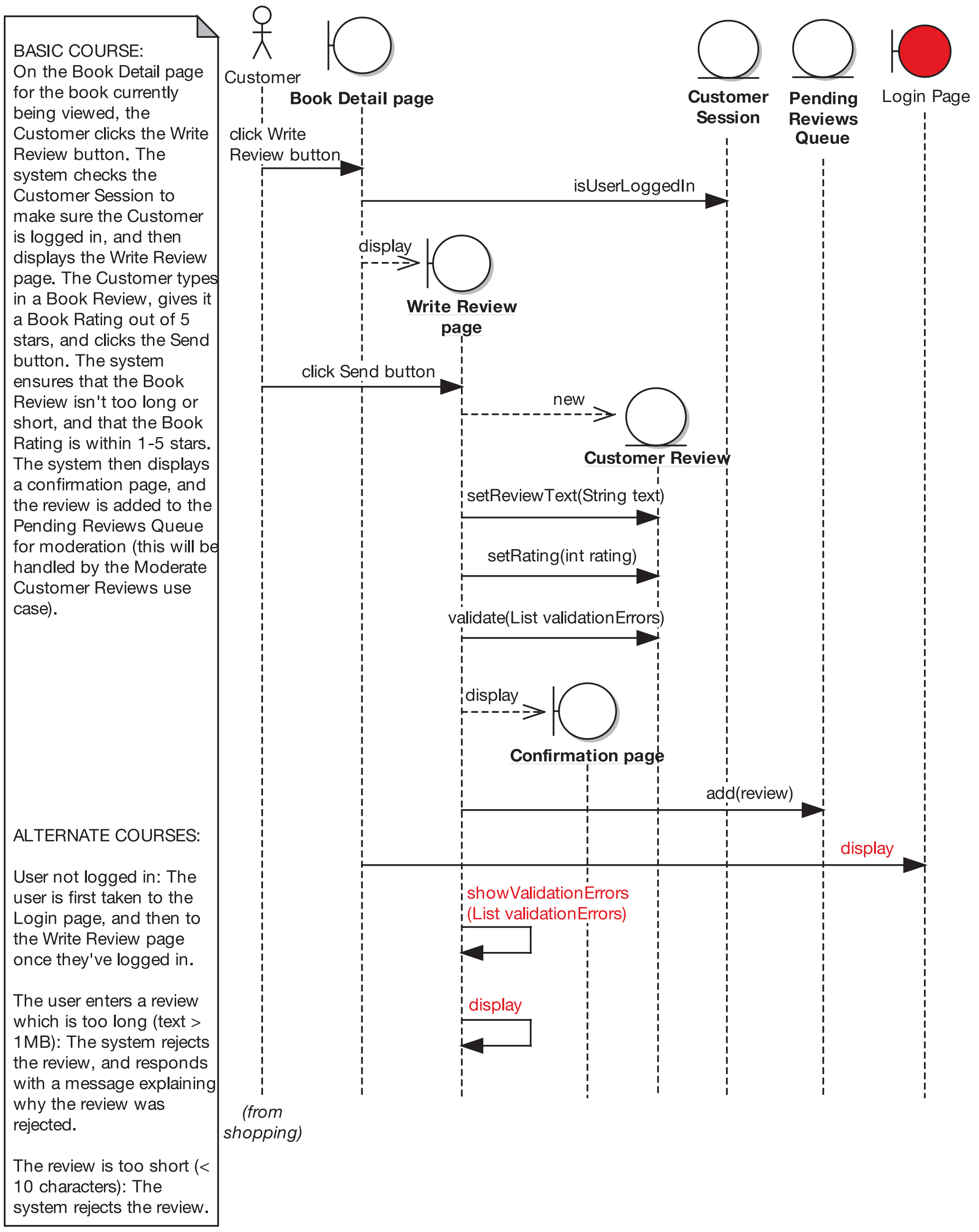
Referring back to the robustness diagram in Figure 6-7, you can see that the Customer clicks the Write Review button on the Book Detail page. The system then performs a check to see if the Customer is logged in. On the robustness diagram, this is represented as the "Is user logged in?" controller. This controller checks the Customer Session entity. On the sequence diagram, then, we'd have an `isUserLoggedIn()` method on a class called `CustomerSession`.

As we now need to be thinking in terms of the implementation details, we need to establish where the `CustomerSession` instance is found. In our Spring/JSP example, the `CustomerSession` would be an object contained in the HTTP Session layer (i.e., naturally associated with the current user session). When the user browses to the bookstore website, a new `CustomerSession` object is created and added as an attribute to `HttpSession`. This is then used to track the user's state (currently just whether the user is logged in). To make this explicit, we should show `HttpSession` on the sequence diagram, so that we can see exactly where `CustomerSession` is found.

Figure 8-10 shows the (nearly) completed sequence diagram. Notice that it's possible to read through the use case text on the left and simultaneously walk through the sequence diagram, tracing the text to the messages on the diagram.

■ **Exercise** In Figure 8-10, the "User not logged in" alternate course describes (at a high level) the process of logging in before proceeding with the current use case. Should the sequence diagram invoke a separate *Login* use case, and if so, how would this be represented on the sequence diagram? We discuss the answer in the CDR in Chapter 9.

As it turns out, a couple of objects weren't used at all. `Review Rejected Screen` wasn't used, although we originally (in the use case text) described the system taking the user to a separate screen to show that the review was rejected. However, on drawing the sequence diagram, it seemed like it would be much nicer to take the user back to the `Write Review Screen` and show the validation errors there, so that the user can simply correct the errors and resubmit the review. We should update the use case text ***at the earliest opportunity*** (this means right now) to reflect this, so that the use case doesn't fall out of sync with the design (and thus lose its usefulness).



BASIC COURSE:
On the Book Detail page for the book currently being viewed, the Customer clicks the Write Review button. The system checks the Customer Session to make sure the Customer is logged in, and then displays the Write Review page. The Customer types in a Book Review, gives it a Book Rating out of 5 stars, and clicks the Send button. The system ensures that the Book Review isn't too long or short, and that the Book Rating is within 1-5 stars. The system then displays a confirmation page, and the review is added to the Pending Reviews Queue for moderation (this will be handled by the Moderate Customer Reviews use case).

ALTERNATE COURSES:

User not logged in: The user is first taken to the Login page, and then to the Write Review page once they've logged in.

The user enters a review which is too long (text > 1MB): The system rejects the review, and responds with a message explaining why the review was rejected.

The review is too short (< 10 characters): The system rejects the review.

Figure 8-10. *Building a sequence diagram, beginning of step 4 (pure OO version of the completed sequence diagram)*

■**Caution** **You don't need to go back and redraw the robustness diagram** 17 more times once you're in the middle of sequence diagramming. You know the robustness diagrams aren't going to be perfect, and that's OK—it isn't important to make your preliminary design diagrams look like you clairvoyantly knew what the detailed design would look like (doing this would, in fact, cause analysis paralysis). It's only important that the preliminary design gets you properly started with detailed design.

Additionally, `Book` wasn't used at all so, after double, triple-checking to make sure we hadn't missed a critical detail, we removed it. We discuss the reasons why `Book` fell off the diagram when we get to the CDR (see Chapter 9).

■**Note** Figure 8-10 raises some other issues, which we'll also address during the CDR. In fact, most of these issues would have been caught using a ***design-driven testing (DDT)*** approach, which we describe in Chapter 12.

This version of the diagram follows a pure OO design. It's a "wouldn't it be great if . . ." diagram reflecting the OO design principles we've discussed. As it's quite high level, it could be transferred to a target platform other than Spring Framework without much (or indeed any) modification. In other words, so far we haven't tied the design in very closely with the nitty-gritty implementation details of our target framework—and we'll need to do that before we begin coding.

Because we're reusing a third-party web framework rather than designing our own, the design is dictated to an extent by the framework's creators. For example, in Figure 8-10 we show the validation taking place as a method on the entity class being validated. However, the approach in Spring Framework is to separate validation into a separate class. This isn't pure OO, as it breaks encapsulation and we potentially end up with lots of tiny "controller classes," each of which implements a single function. Having said that, there are practical, ground-based reasons why Spring does it this way.[4]

So, we need to explore the implementation details and fill in these details on the sequence diagram before we can really code from it. To complete the final step in our four essential steps, we need to add in more detail taking into account the framework and the technology being targeted.

Figure 8-11 shows the finished diagram. This is now something that we can code from directly (after the CDR, of course!).

The processing in this diagram is quite detail-rich, so we've moved the description into Appendix B (in the section titled "'Write Customer Review' Use Case"). If you're interested in the Spring details, please do take a look there before moving on.

4. Spring is actually one of the better frameworks for not imposing too many design constraints. For example, Struts (a popular MVC web framework) contains a much more rigid design, in which your classes must extend specific Struts classes. Because Java doesn't allow multiple inheritance, this can be a limiting factor.

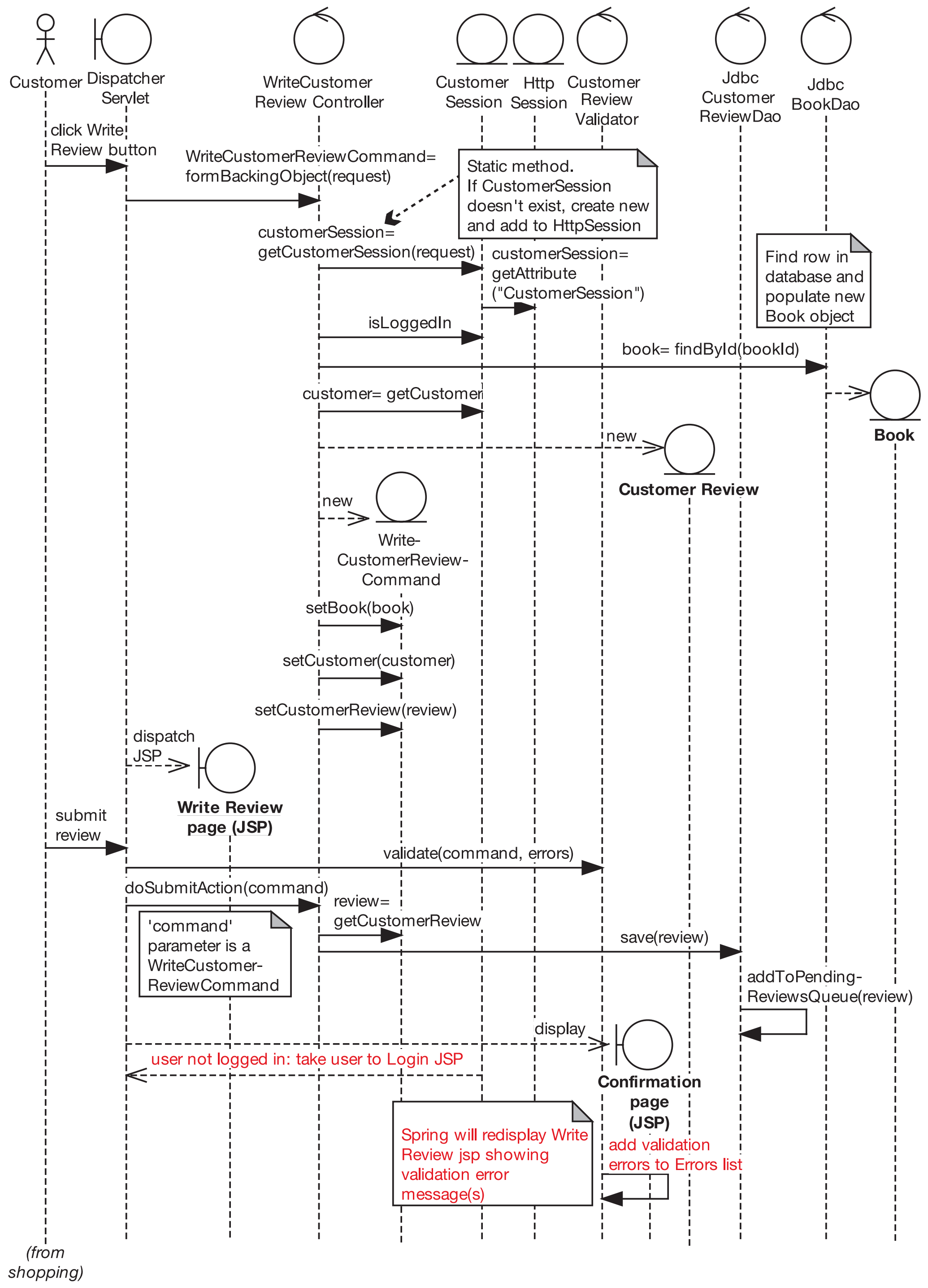

Customer
Dispatcher Servlet
WriteCustomer Review Controller
Customer Session
Http Session
Customer Review Validator
Jdbc Customer ReviewDao
Jdbc BookDao
click Write Review button
WriteCustomerReviewCommand= formBackingObject(request)
Static method. If CustomerSession doesn't exist, create new and add to HttpSession
customerSession= getCustomerSession(request)
customerSession= getAttribute ("CustomerSession")
Find row in database and populate new Book object
isLoggedIn
book= findById(bookId)
customer= getCustomer
Book
new
Customer Review
new
Write-CustomerReview-Command
setBook(book)
setCustomer(customer)
setCustomerReview(review)
dispatch JSP
Write Review page (JSP)
submit review
validate(command, errors)
doSubmitAction(command)
review= getCustomerReview
save(review)
'command' parameter is a WriteCustomer-ReviewCommand
addToPending-ReviewsQueue(review)
display
Confirmation page (JSP)
user not logged in: take user to Login JSP
Spring will redisplay Write Review jsp showing validation error message(s)
add validation errors to Errors list
(from shopping)

Figure 8-11. *Building a sequence diagram, completing step 4 (the completed sequence diagram)*

■Exercise `WriteCustomerReviewCommand` seems quite redundant, as it's simply there to hold data that gets set in the `CustomerReview` domain class anyway. What could be done to improve this part of the design? We reveal the answer during the CDR in the next chapter.

■Exercise The `Book` class in Figure 8-11 doesn't seem to do a huge amount. How could this design be improved to give `Book` a more prominent role? (Remember, we had the same problem with the *Show Book Details* sequence diagram in Figure 8-2.) Again, we reveal the answer during the CDR in the next chapter.

Now that we've completed the sequence diagram, it's time to bring the static model up to date. After walking through the theory, we show the updated static model for the Internet Bookstore.

Updating Your Class Diagrams As You Go Along

As you've probably gathered by now, you need to keep updating and refining your static model (the class diagrams) as you go along (see Figure 8-12).

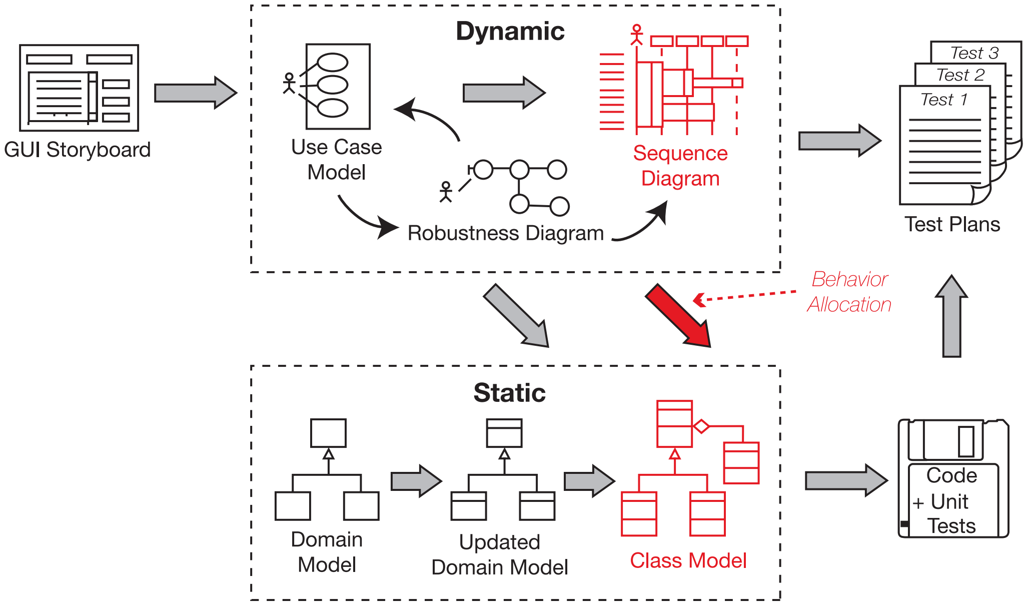


Figure 8-12. *Updating your static model, again*

Synchronizing the Static and Dynamic Parts of the Model

CASE tools take much of the burden out of keeping the static and dynamic models in sync by putting operations on classes as you draw message arrows on sequence diagrams. But we do recommend that, with each message arrow you draw on the sequence diagram, you take a peek at the class diagram to make sure it's updated correctly.

It's a good habit to check that the operations are added to the correct class each time you draw a message, and if you spot any missing attributes, add them as soon as you notice they're missing. Seeing each class evolve should also cause you to think about the class structure, and evolve the class design. You might spot an opportunity for generalization, for example; you may see a possible use of a design pattern; or you could find that a class has gained too many responsibilities and needs to be split in two using aggregation.

As you're adding implementation details to the sequence diagrams, don't be surprised if you start to see lots of new classes appear on the static model that were never there on the domain model during analysis. That's because you're now identifying the *solution space* classes. By contrast, the domain model shows classes only from the problem space.

The two spaces are now converging during detailed design. By updating the static model as you work through the sequence diagrams, you're now converging the problem space with the solution space.

So, as you add in more functionality from the use cases, you'll also come up with scaffolding and other types of infrastructure (e.g., "helper" classes).

Adding getters and setters to your class diagrams can be time-consuming and doesn't give you much in return, except a busy-looking class diagram. Our advice is to avoid adding them to your model. For now, just add the attributes as private fields and utilize encapsulation. As you only ever allow access to attributes via getters and setters, adding them is kind of redundant. When you generate code, you should be able to generate get and set methods from attributes anyhow. Pretty much all modern IDEs and diagramming tools are scriptable or have this sort of automation built in.

Although we suggested that you avoid adding too much design detail during domain modeling (since there just wasn't enough supporting information available at that time), now that you're at the detailed design stage, this really is the time to go wild (so to speak) and think the design through in fine detail. You and your team are shaping up the design—collaboratively, we hope—and getting it ready for coding.

In the next section, we illustrate how to synchronize the static and dynamic parts of the model for the Internet Bookstore.

Internet Bookstore: Updating the Static Model

In this section, we show three different versions of the static model updated from the sequence diagrams (review the previous version in Figure 5-19 to see how the static model has evolved so far). Figure 8-13 shows the static model updated from the pure OO sequence diagram shown in Figure 8-10.

`Book` and `BookCollection` are on the static model because they also appeared on the domain model; however, neither class has operations yet because we haven't drawn any sequence diagrams that allocate behavior to them.

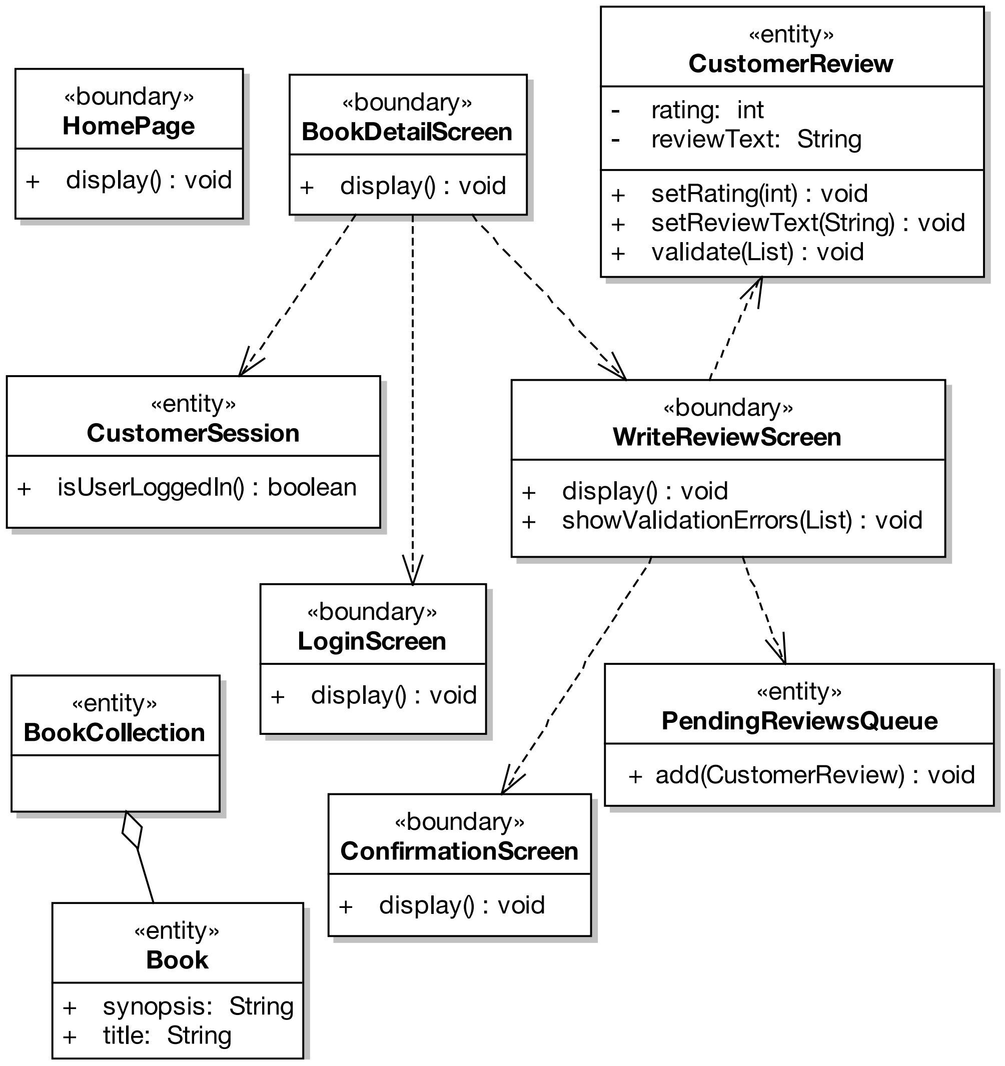


Figure 8-13. *Internet Bookstore static model based on the pure OO sequence diagram in Figure 8-10*

`CustomerSession` used to have a `loggedIn` attribute (see the domain model diagram in Figure 5-19), but after sequence diagramming, this has turned into a method, `isUserLoggedIn()`, without an attribute to back it (i.e., it's a "live" or calculated value that is rechecked each time the method is called).

Figure 8-13 reveals some commonality between some of the classes. Each of the boundary classes has its own `display()` method. So it would make sense to move this up to a new parent class. Figure 8-14 shows the result of this prefactoring (it's much quicker to do this sort of thing now, while we're looking at the bigger picture, before we have source code dependent on the class we're prefactoring).

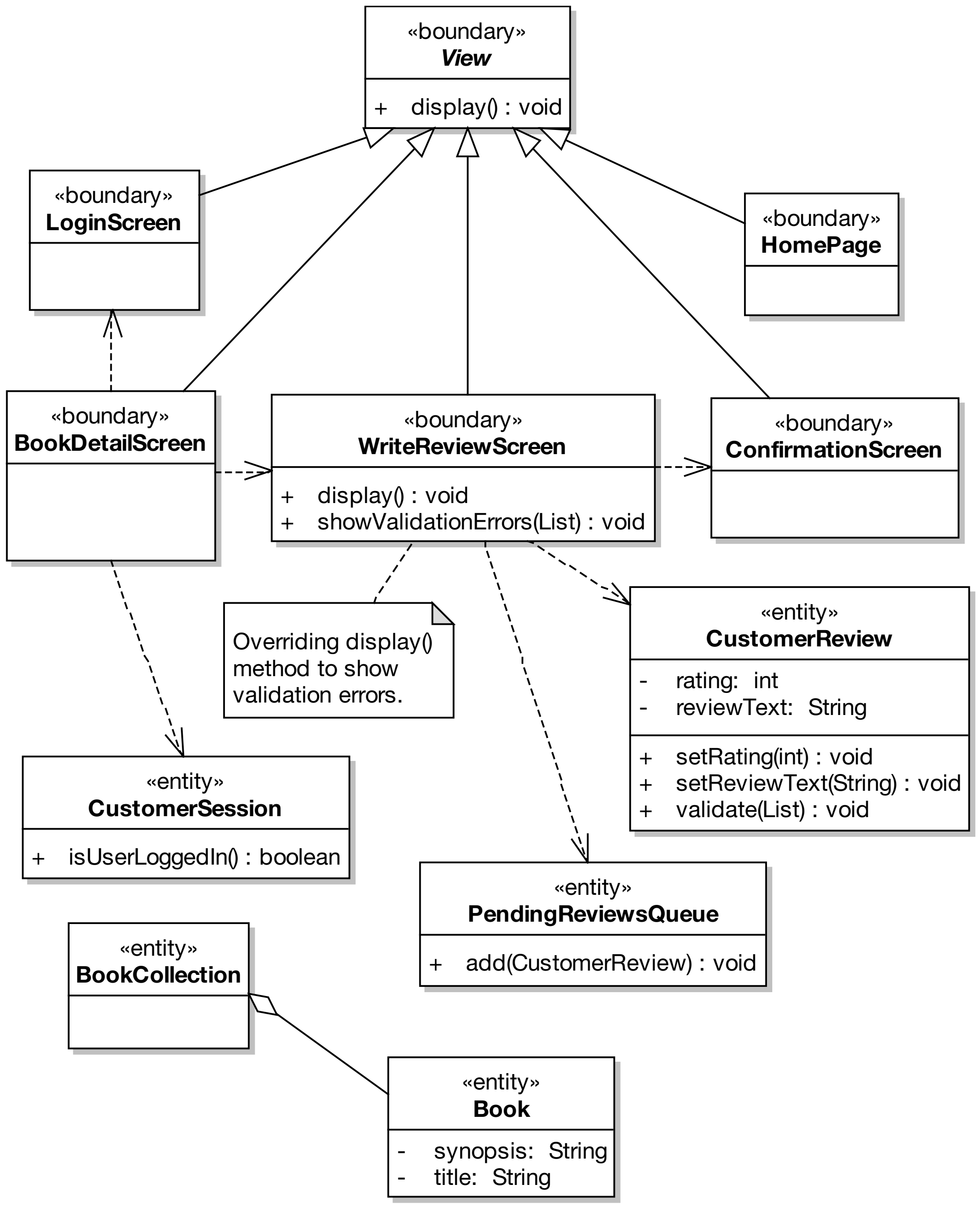


Figure 8-14. *Prefactored Internet Bookstore static model, still pure OO*

In Figure 8-14, we created a new abstract class called `View`, which contains the common `display()` method. The boundary classes only need to override this method if they're doing something special. In this case, `WriteReviewScreen` overrides it because it needs to display the validation errors set via `showValidationErrors()`.

We don't want to go any further without grounding the design in the reality of the target platform. What we especially want to avoid is designing our own framework (which we're starting to risk doing by creating a common `View` class), instead of just wrapping the design around the predesigned target framework. So let's correct that now: Figure 8-17 shows a much more detailed version of the static model, derived from the detailed, nitty-gritty version of the sequence diagram, taking into account the real-world constraints (and benefits!) of Spring Framework.

MULTIPLICITY

Multiplicity refers to the numbers that you often see on the lines between classes on class diagrams. For example, Figure 8-15 shows an excerpt from a class diagram showing multiplicity, and Figure 8-16 shows how the same diagram might be described in text form.

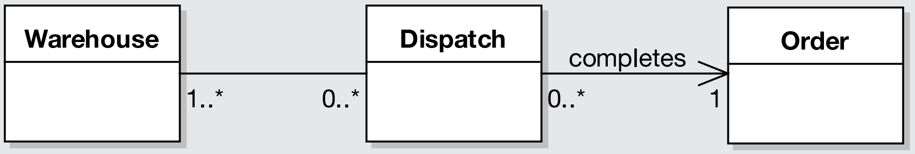


Figure 8-15. *Example class diagram notation for associations*

A Warehouse has zero to many Dispatch items

Each Dispatch belongs to one to many Warehouses

Dispatch completes an Order

Figure 8-16. *The same diagram in text form*

The level of detail shown in Figure 8-15 would probably be more appropriate for a relational data model than a class diagram. On relational data models (usually shown using entity-relationship [ER] diagrams), showing the precise multiplicity for each relationship is of paramount importance. However, for class diagrams it's much less important—optional, even. One possible exception to this is if you want to use multiplicity to indicate validation rules (e.g., a Dispatch *must* have at least one Order before it completes).

You might also want to use multiplicity to indicate whether a variable will be a **single** object reference or a **list** of object references, in which case, just showing **1** or ***** (respectively) would be sufficient. For example, in Figure 8-15, the Warehouse class can have many Dispatch items, so this would be represented in Java with a `List` (or some other `Collection` class):

```
private List<Dispatch> dispatches = new ArrayList<Dispatch>();
```

On the other hand, a Dispatch references only one Order, so this would be represented by a single object reference:

```
private Order order;
```

Figure 8-17 has been updated to show the details from the sequence diagrams for both the *Show Book Details* and *Write Customer Review* use cases.

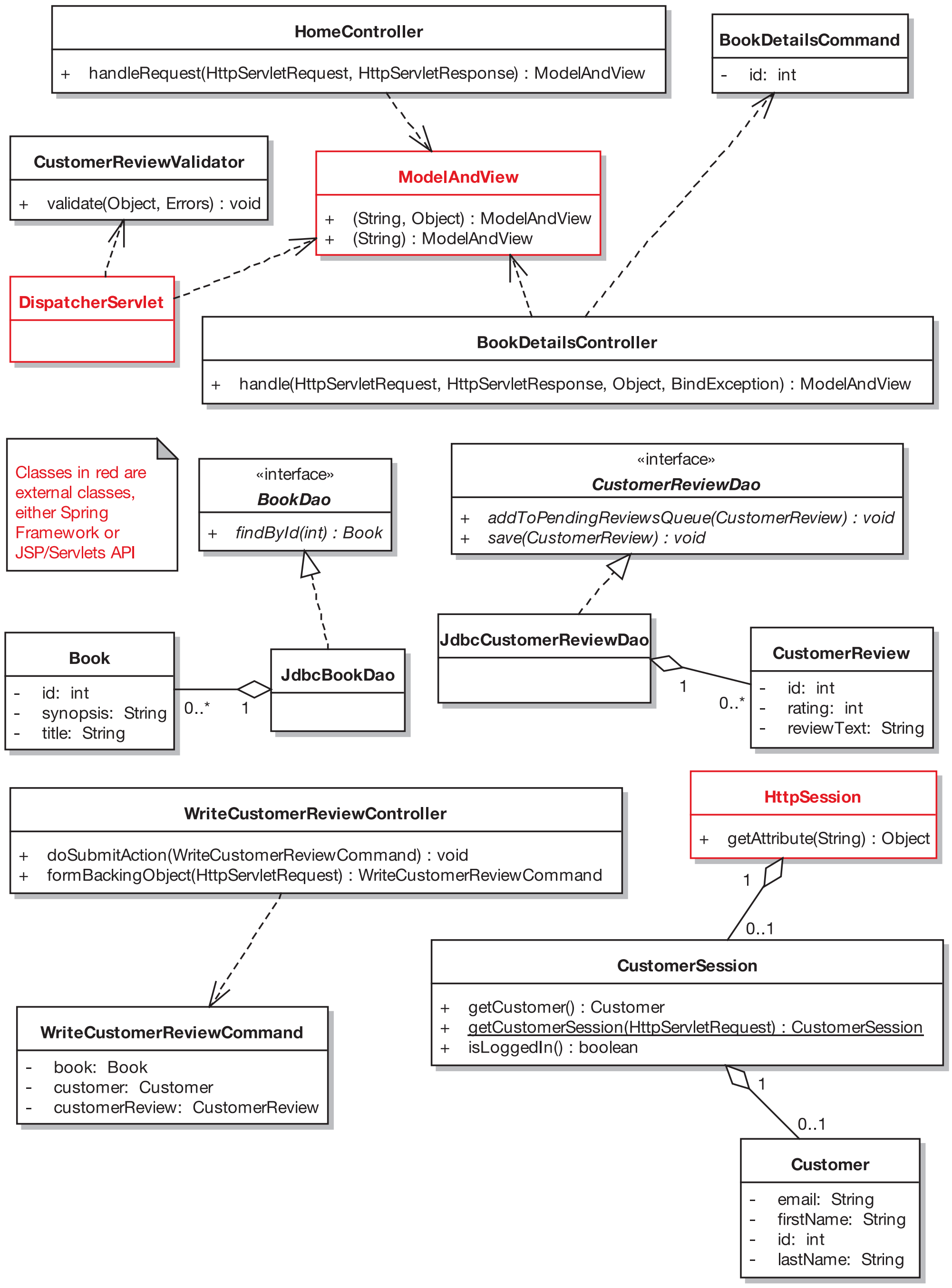

HomeController
+ handleRequest(HttpServletRequest, HttpServletResponse) : ModelAndView
BookDetailsCommand
- id: int
CustomerReviewValidator
+ validate(Object, Errors) : void
ModelAndView
+ (String, Object) : ModelAndView
+ (String) : ModelAndView
DispatcherServlet
BookDetailsController
+ handle(HttpServletRequest, HttpServletResponse, Object, BindException) : ModelAndView
Classes in red are external classes, either Spring Framework or JSP/Servlets API
«interface» BookDao
+ findById(int) : Book
«interface» CustomerReviewDao
+ addToPendingReviewsQueue(CustomerReview) : void
+ save(CustomerReview) : void
JdbcCustomerReviewDao
CustomerReview
- id: int
- rating: int
- reviewText: String
Book
- id: int
- synopsis: String
- title: String
JdbcBookDao
0..*
1
1
0..*
WriteCustomerReviewController
+ doSubmitAction(WriteCustomerReviewCommand) : void
+ formBackingObject(HttpServletRequest) : WriteCustomerReviewCommand
HttpSession
+ getAttribute(String) : Object
1
0..1
CustomerSession
+ getCustomer() : Customer
+ getCustomerSession(HttpServletRequest) : CustomerSession
+ isLoggedIn() : boolean
WriteCustomerReviewCommand
- book: Book
- customer: Customer
- customerReview: CustomerReview
1
0..1
Customer
- email: String
- firstName: String
- id: int
- lastName: String

Figure 8-17. *Internet Bookstore static model after drawing sequence diagrams for two use cases*

What's changed in Figure 8-17?

- We've introduced a couple of Spring classes (DispatcherServlet and ModelAndView) and a class from the Java Servlet API (HttpSession), and we've shown which of our own classes relate to them.
- The aggregation relationships now show their **multiplicity** (see the previous sidebar).
- We've filled in the properties that were identified in the use case, plus any supplementary specs (screen mock-ups, passive-voice functional specifications, etc.).
- Our Screen classes have been replaced with (mostly) Spring Controller classes. Which brings us to the next point . . .
- Some of the methods that originated from controllers on the robustness diagrams have escaped and turned into their own Controller classes. While we wouldn't normally recommend this, Spring requires that each form has its own Controller class. This class handles the details of extracting postvalidated data, processing it, and then telling the framework which JSP page to send back to the user.
- We've introduced a Command class (WriteCustomerReviewCommand), which represents the data extracted from the user's form post.

■**Note** In the next chapter, we revisit this part of the design and look at how to make it more "domain oriented" while lessening our reliance on separate Command and Controller classes *and* still fitting into the Spring design mold.

- We've got DAOs. BookDao, CustomerReviewDao—you name it, we've got it. Remember that a DAO (e.g., BookDao) is analogous to a database table, whereas a domain class (e.g., Book) is analogous to a row in a database table. A BookDao is a source of Books, so it's similar to an EJB Home object.
- BookCollection has disappeared. Instead we have BookDao, as mentioned in the previous point. In fact, BookDao will actually return collections of Books—though we haven't shown this yet, as we haven't drawn these Collection operations on any of the sequence diagrams.
- Because Spring has its own validation framework that we want to take advantage of, we've separated the CustomerReview validation into its own separate class, CustomerReviewValidator. (Note that although this separation of concerns seems like a good idea on the surface, it turns out to be not that good an idea after all. We discuss the reasons why in Chapter 11.)

Although it contains a lot of detail, Figure 8-17 contains purely the detail we've uncovered during domain modeling, robustness analysis, and sequence diagramming, and nothing more. There are no leaps of logic, leaps of faith, or leaps of any kind. The operations on each class are taken directly from the messages we drew on the sequence diagrams. The relationships between each class are also derived from the relationships in the domain model and from the operations. The attributes on each class are derived from the detail in the use cases and any supplementary specs that the use cases reference (e.g., screen mock-ups, data models, etc.).

It's tempting to add detail to the class diagram because you think it might be needed, but (as we hope we've demonstrated) it's better fill in the detail while drawing the sequence diagrams. If, after you've fleshed out the static model using the sequence diagrams, the static model still appears to be missing some detail, then you should revisit the sequence diagrams (and possibly even the robustness diagrams and use cases), as it's likely that something has been missed. Always trust your gut instinct, but use it as an indication that you need to revisit your previous diagrams, not that you need to second-guess yourself and add detail arbitrarily to the static model.

We've now finished the detailed design for the two use cases we're implementing. The next stage before coding will be the CDR, a "sanity check" involving senior technical staff to make sure the design is shipshape before the use cases are implemented.

Sequence Diagramming in Practice

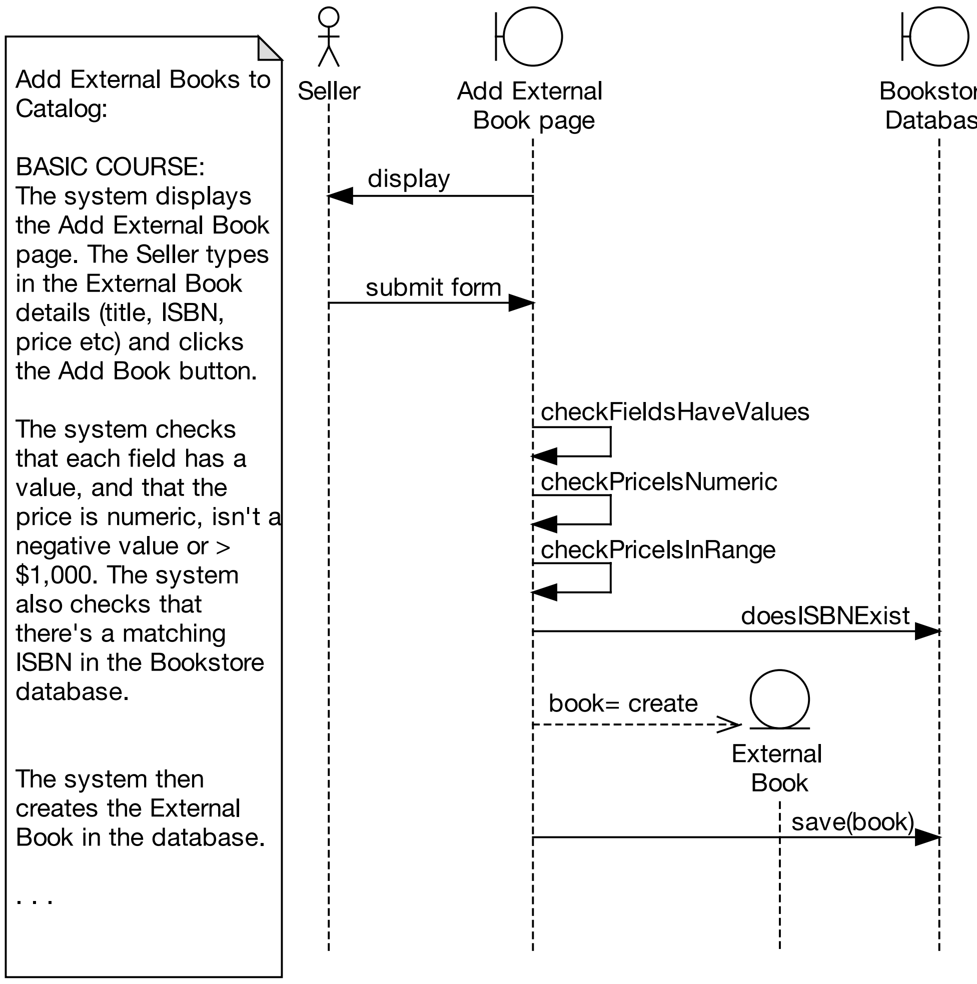
The following exercises, taken from the detailed design activities for the Internet Bookstore, are designed to test your ability to spot the most common mistakes that people make during sequence diagramming.

Exercises

Each of the diagrams in Figures 8-18 to 8-20 contains one or more typical modeling errors. For each diagram, try to figure out the errors and then draw the corrected diagram. The answers are in the next section.

Exercise 8-1

Figure 8-18 shows an excerpt from a sequence diagram for the *Add External Books to Catalog* use case (which you also encountered in the exercises in Chapter 5). It shows quite a few examples of a common behavior allocation error. (Hint: Which objects are the messages pointing to?) Try to explain why the behavior allocation in the diagram is wrong, and then draw the corrected diagram.


Add External Books to Catalog:

BASIC COURSE:
The system displays the Add External Book page. The Seller types in the External Book details (title, ISBN, price etc) and clicks the Add Book button.

The system checks that each field has a value, and that the price is numeric, isn't a negative value or > $1,000. The system also checks that there's a matching ISBN in the Bookstore database.

The system then creates the External Book in the database.

. . .
Seller
Add External Book page
Bookstore Database
display
submit form
checkFieldsHaveValues
checkPriceIsNumeric
checkPriceIsInRange
doesISBNExist
book= create
External Book
save(book)

Figure 8-18. *Excerpt from a sequence diagram showing several behavior allocation errors*

Exercise 8-2

Figure 8-19 shows an excerpt from a sequence diagram for the *Create New Book* use case (this use case is intended for Bookstore staff, so that they can add new Book titles to their online Catalog). There are a couple of problems with this diagram excerpt (one of which is repeated many times in the diagram). See if you can find them both.

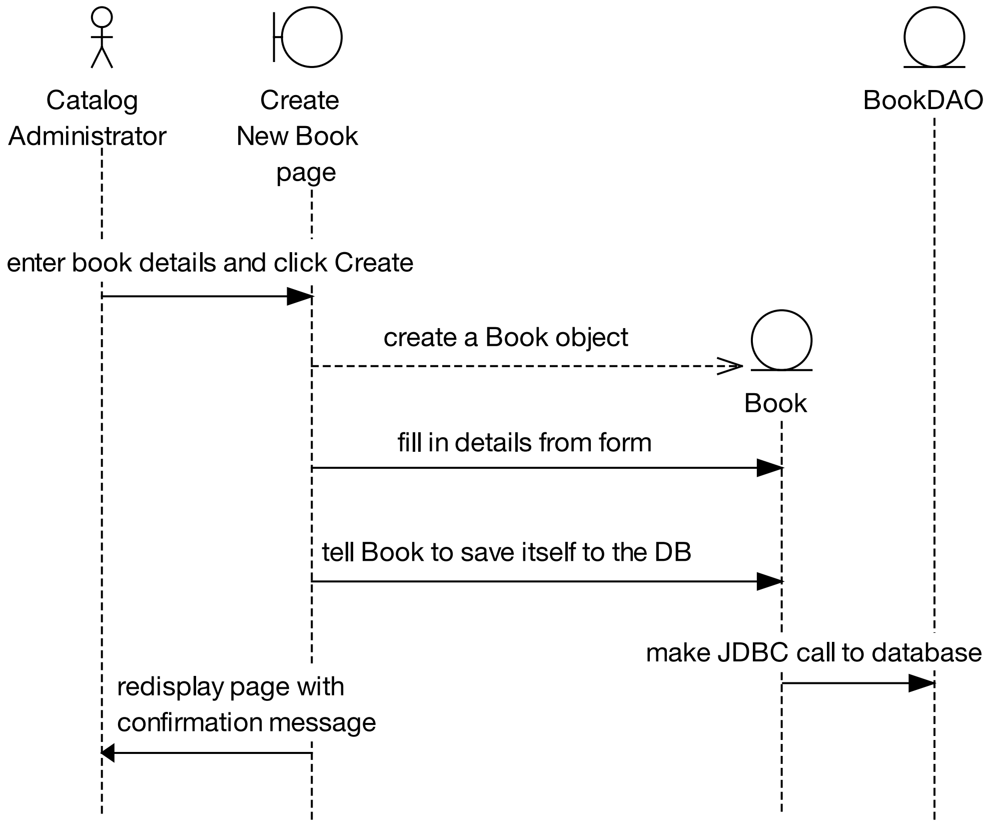

Catalog Administrator
Create New Book page
BookDAO
enter book details and click Create
create a Book object
Book
fill in details from form
tell Book to save itself to the DB
make JDBC call to database
redisplay page with confirmation message

Figure 8-19. *Excerpt from a sequence diagram showing a couple of pretty major problems*

Exercise 8-3

Figure 8-20 shows an excerpt from a sequence diagram for the *Edit Shopping Cart* use case. The problems with this diagram are partly related to the diagram showing too much detail in some aspects, but also (ironically perhaps) the diagram displays too little detail where it should be showing more of the design's "plumbing." There are also a couple of issues to do with the use case's scope (i.e., where it starts and finishes). In total, you should find six errors on this diagram. Good luck!

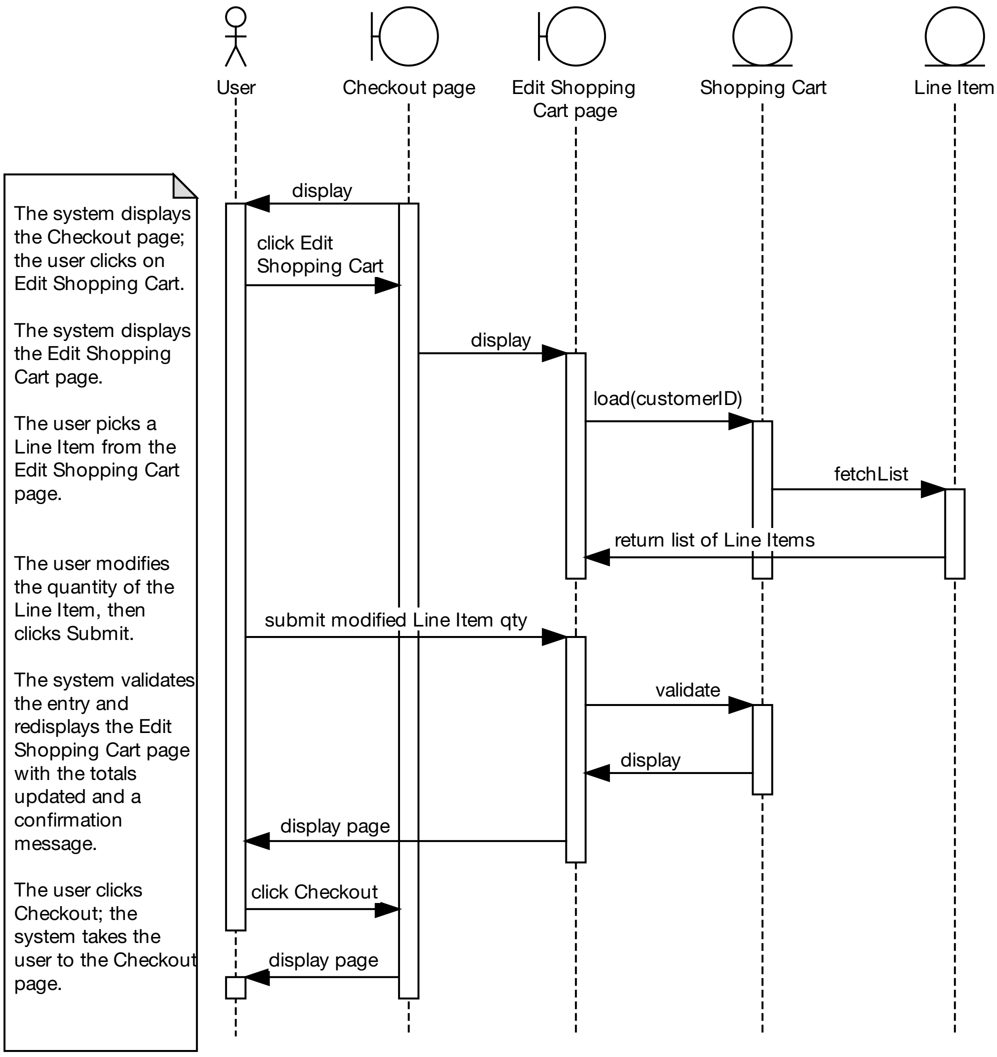

User
Checkout page
Edit Shopping Cart page
Shopping Cart
Line Item
The system displays the Checkout page; the user clicks on Edit Shopping Cart.
The system displays the Edit Shopping Cart page.
The user picks a Line Item from the Edit Shopping Cart page.
The user modifies the quantity of the Line Item, then clicks Submit.
The system validates the entry and redisplays the Edit Shopping Cart page with the totals updated and a confirmation message.
The user clicks Checkout; the system takes the user to the Checkout page.
display
click Edit Shopping Cart
display
load(customerID)
fetchList
return list of Line Items
submit modified Line Item qty
validate
display
display page
click Checkout
display page

Figure 8-20. *Excerpt from a sequence diagram showing both too much and too little detail*

Exercise Solutions

Following are the solutions to the exercises.

Exercise 8-1 Solution: Non-OO Behavior Allocation

Figure 8-21 highlights the parts of the sequence diagram where the messages were allocated incorrectly. The highlighted messages are really the responsibility of `ExternalBook`. In the current design, the validation checking goes on in the boundary object, and only after that's all done is the `ExternalBook` object created. Its only purpose in life is to be passed into `BookstoreDatabase` for saving.

Note that two of the methods haven't been highlighted: `checkFieldsHaveValues` and `checkPriceIsNumeric`. These are legitimately a part of the boundary object, as they're checking that the incoming data is both present and in the correct format. `checkPriceIsInRange`, on the other hand, is "genuine" data validation, so it belongs on `ExternalBook`.

On `BookstoreDatabase` (over on the right of Figure 8-21), two more methods have been highlighted: `doesISBNExist` and `save(book)`. In this case, it's the *caller* that's wrong—both methods should be called by `ExternalBook`.

Figure 8-22 shows the corrected diagram.

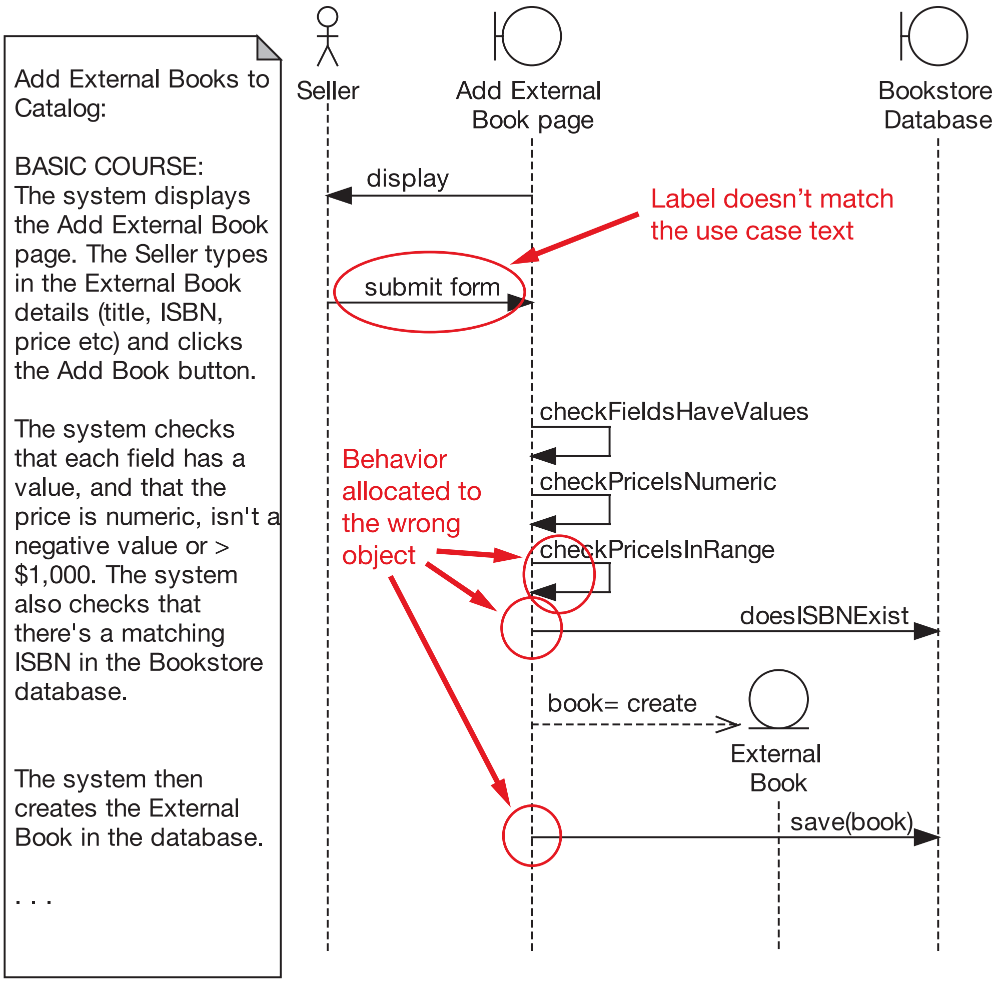
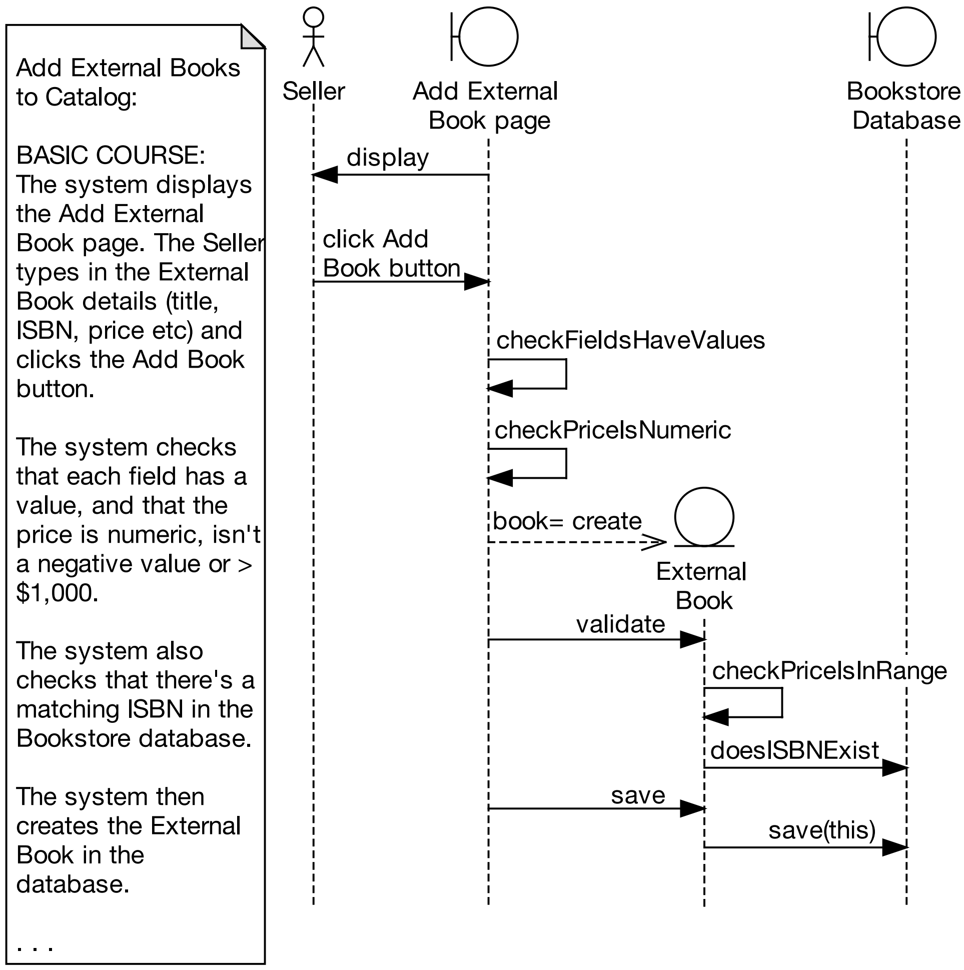

Add External Books to Catalog:
BASIC COURSE:
The system displays the Add External Book page. The Seller types in the External Book details (title, ISBN, price etc) and clicks the Add Book button.
The system checks that each field has a value, and that the price is numeric, isn't a negative value or > $1,000. The system also checks that there's a matching ISBN in the Bookstore database.
The system then creates the External Book in the database.
. . .
Seller
Add External Book page
Bookstore Database
display
Label doesn't match the use case text
submit form
checkFieldsHaveValues
Behavior allocated to the wrong object
checkPriceIsNumeric
checkPriceIsInRange
doesISBNExist
book= create
External Book
save(book)

Figure 8-21. *The sequence diagram excerpt from Exercise 8-1, with the errors highlighted*


Add External Books to Catalog:

BASIC COURSE:
The system displays the Add External Book page. The Seller types in the External Book details (title, ISBN, price etc) and clicks the Add Book button.

The system checks that each field has a value, and that the price is numeric, isn't a negative value or > $1,000.

The system also checks that there's a matching ISBN in the Bookstore database.

The system then creates the External Book in the database.

. . .
Seller
Add External Book page
Bookstore Database
display
click Add Book button
checkFieldsHaveValues
checkPriceIsNumeric
book= create
External Book
validate
checkPriceIsInRange
doesISBNExist
save
save(this)

Figure 8-22. *The corrected sequence diagram excerpt for Exercise 8-1*

Exercise 8-2 Solution: Flowcharting

Figure 8-23 highlights the parts of the sequence diagram that have gone wrong. The main issue is that the sequence diagram is being used as a flowchart, instead of for its primary purpose in life: to allocate behavior to classes.

Flowcharting on sequence diagrams isn't necessarily an evil thing in and of itself, and it is almost certainly better than not doing the sequence diagram at all. But we consider it to be (at best) a weak usage of a sequence diagram because it doesn't leverage the ability to assign operations to classes while drawing message arrows. Since, in our opinion, this activity is pretty much the fundamental place where "real OOD" happens, we've flagged it as an error. We think you can (and should) do better than just using the sequence diagram as a flowchart.

The second issue is that there's no validation performed on the incoming form data—and therefore no error handling code for rejecting bad data. Either the validation steps were left out of the use case or the designer didn't draw the sequence diagram directly from the use case text.

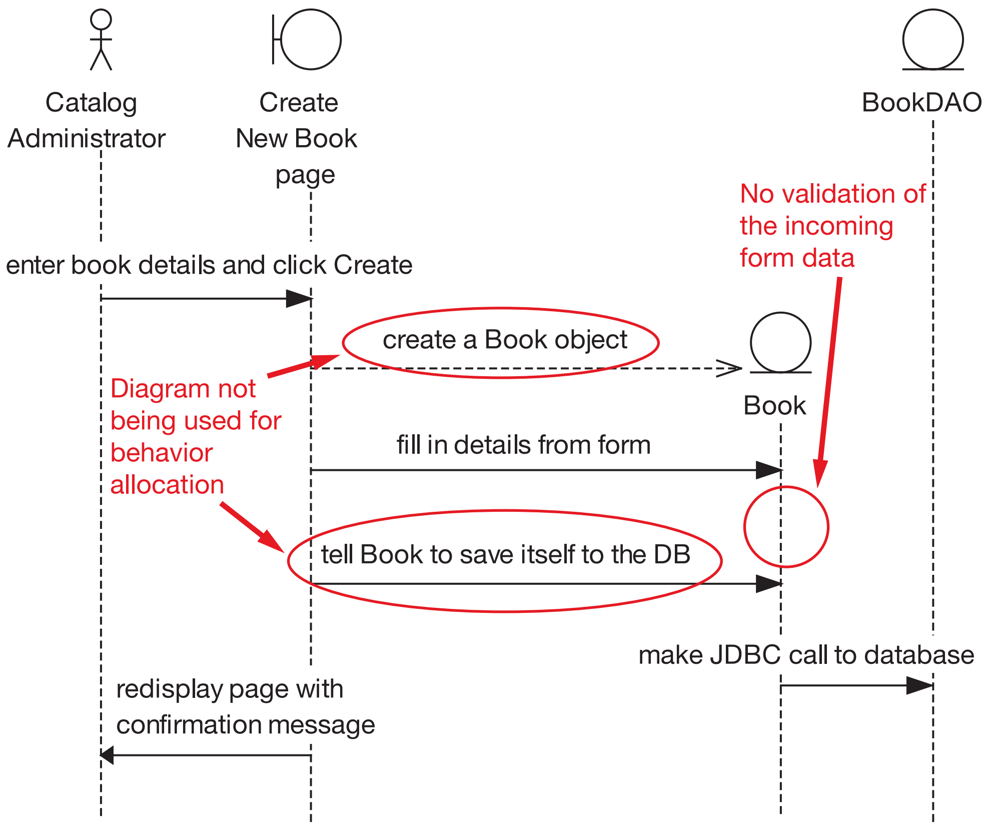

Catalog Administrator
Create New Book page
BookDAO
No validation of the incoming form data
enter book details and click Create
create a Book object
Book
Diagram not being used for behavior allocation
fill in details from form
tell Book to save itself to the DB
make JDBC call to database
redisplay page with confirmation message

Figure 8-23. *The sequence diagram excerpt from Exercise 8-2, with the errors highlighted*

Figure 8-24 shows the corrected diagram. The corrected version includes the alternate course (shown in red) for when the form validation fails.

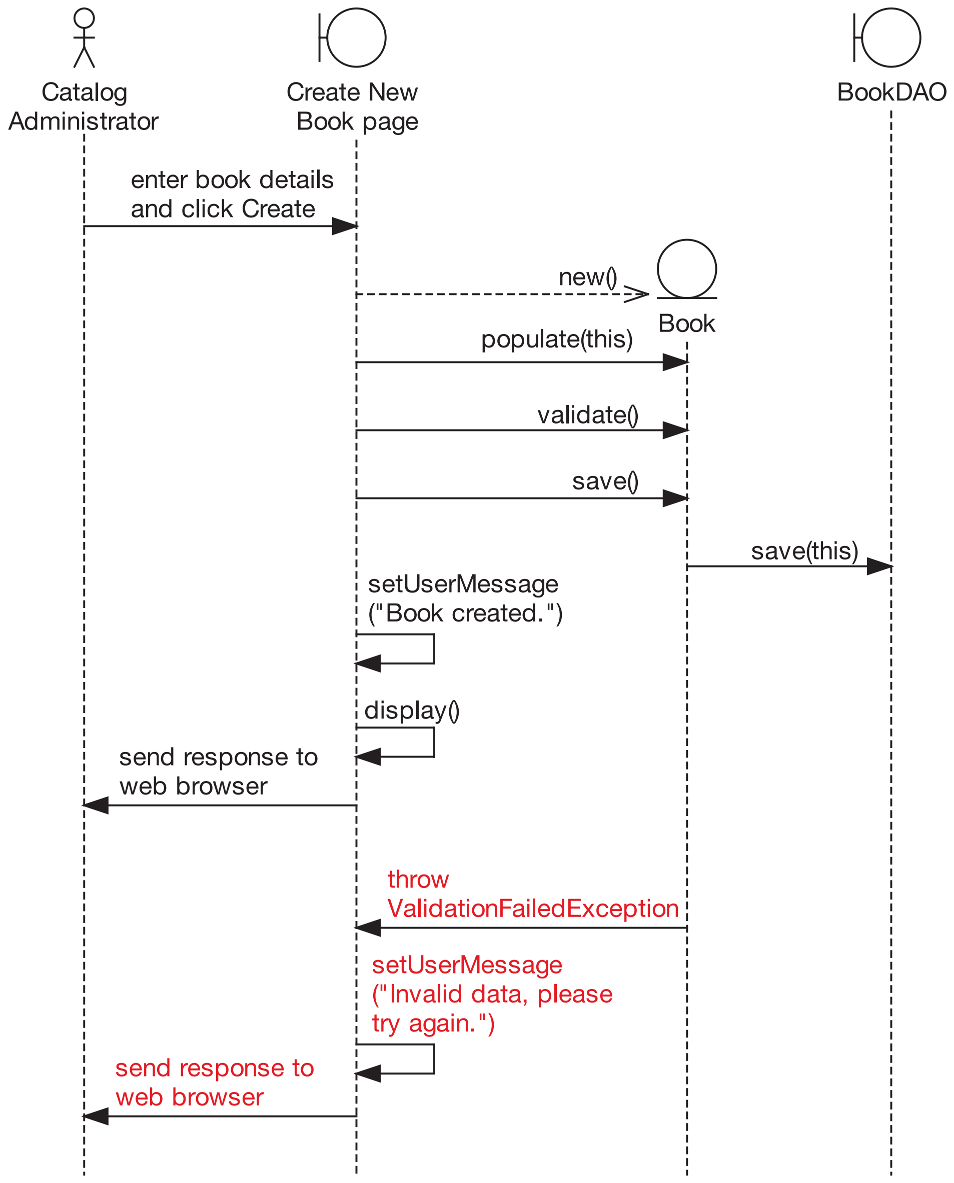

Catalog Administrator
Create New Book page
BookDAO
enter book details and click Create
new()
Book
populate(this)
validate()
save()
save(this)
setUserMessage ("Book created.")
display()
send response to web browser
throw ValidationFailedException
setUserMessage ("Invalid data, please try again.")
send response to web browser

Figure 8-24. *The corrected sequence diagram excerpt for Exercise 8-2*

Exercise 8-3 Solution: Plumbing

Figure 8-25 highlights the parts of the sequence diagram that have gone wrong.

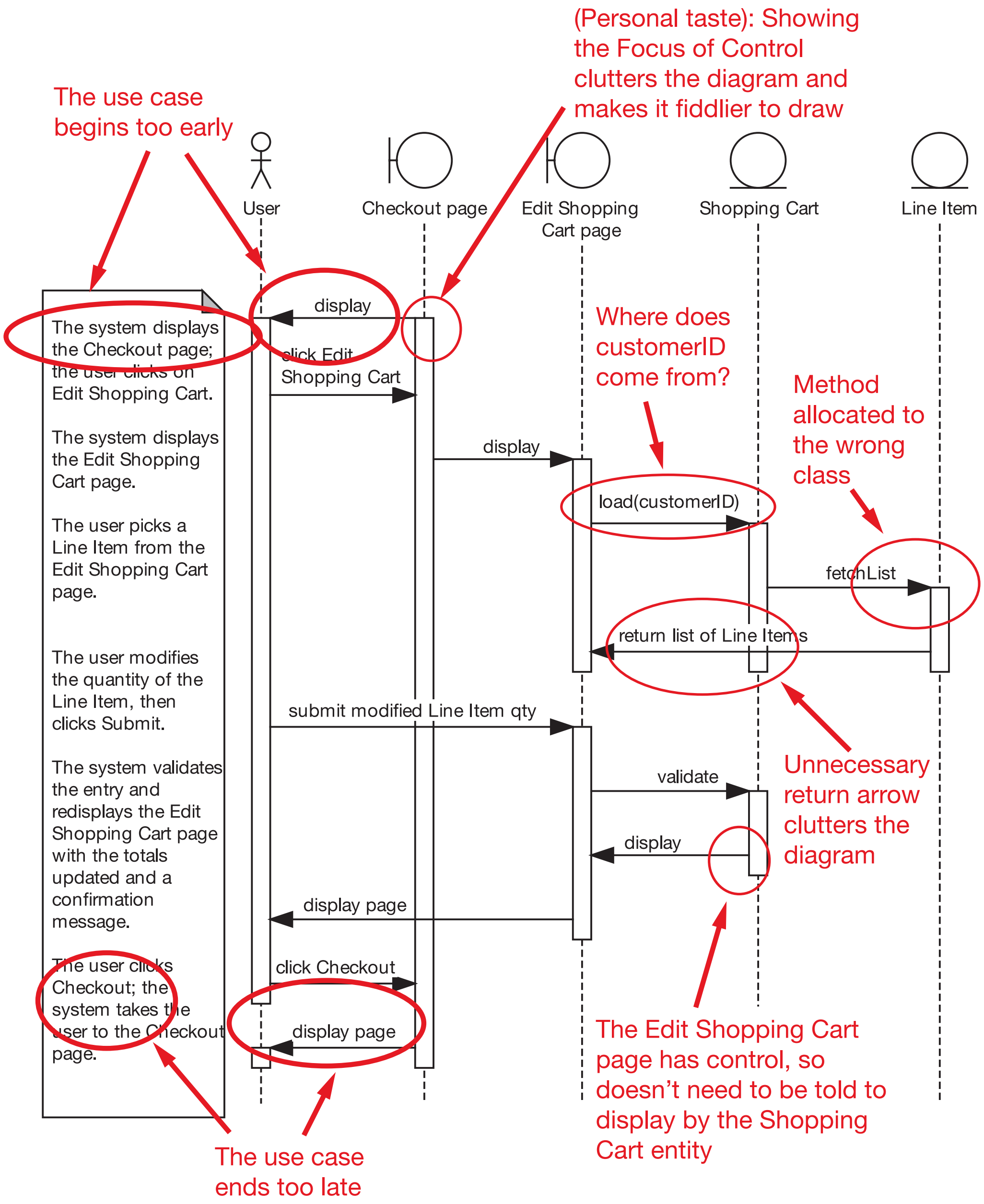

(Personal taste): Showing the Focus of Control clutters the diagram and makes it fiddlier to draw
The use case begins too early
User
Checkout page
Edit Shopping Cart page
Shopping Cart
Line Item
The system displays the Checkout page; the user clicks on Edit Shopping Cart.
The system displays the Edit Shopping Cart page.
The user picks a Line Item from the Edit Shopping Cart page.
The user modifies the quantity of the Line Item, then clicks Submit.
The system validates the entry and redisplays the Edit Shopping Cart page with the totals updated and a confirmation message.
The user clicks Checkout; the system takes the user to the Checkout page.
display
click Edit Shopping Cart
display
Where does customerID come from?
Method allocated to the wrong class
load(customerID)
fetchList
return list of Line Items
submit modified Line Item qty
validate
display
Unnecessary return arrow clutters the diagram
display page
click Checkout
display page
The Edit Shopping Cart page has control, so doesn't need to be told to display by the Shopping Cart entity
The use case ends too late

Figure 8-25. *The sequence diagram excerpt from Exercise 8-3, with the errors highlighted*

The use case starts and finishes at the wrong stages, suggesting a problem with the overall scope of the use case. It's a use case about editing the Shopping Cart, but it contains details about interacting with the Checkout page (i.e., the use case lacks focus). Luckily in this example, the problem is easy to fix, but normally you'd expect to catch this kind of scope issue during robustness analysis or the PDR at the latest. If you encounter a sequence diagram where the use case's scope is still wrong, you should take a look at the process and work out what's gone wrong.

The next issue is more a matter of personal taste than an egregious error. The sequence diagram is showing the focus of control (the rectangles that indicate the "lifetime" of each message). Note that this isn't necessarily an error, as some people do prefer to show these, but for the purposes of **behavior allocation**, our preference is to not show them, as they clutter the diagram and make it more difficult to draw, without giving a whole lot back in return.

On to the next issue, which is definitely a modeling error. In the `load(customerID)` message, you'd be forgiven for wondering where the `customerID` sprang from. Any time you see a leap of logic on a sequence diagram, where it isn't clear where something came from, then it's probable that a part of the design has been missed. In this case, the diagram is missing the `CustomerAccount` object, which should have been populated when the session began. This can then be retrieved from `CustomerSession`. These design details all need to be shown on the diagram, as it's essential "plumbing" work. (In this example, it would be reasonable to put a note on the diagram stating that `CustomerSession` and `CustomerAccount` are set up during the *Login* use case, so that the diagram remains focused on the use case that we're currently designing.) In fact, it would also make sense to retrieve the `ShoppingCart` from the `CustomerAccount`, instead of telling the `ShoppingCart` to "go find itself" based on the customer's ID.

Next up is the `fetchList` method, which `ShoppingCart` calls on `LineItem`. A quick check of the class diagram (assuming this is being automatically updated as you allocate messages on the sequence diagram) quickly reveals that `fetchList()` just doesn't belong on the `LineItem` class, as it returns a *list* of `LineItems`. Instead, it would make more sense for this method to be on `ShoppingCart` itself, but to be called from the `EditShoppingCart` boundary object.

Still on `LineItem`, the "return list of Line Items" arrow isn't needed. Normally, you don't need to draw an arrow to show return values, as the method returning is implied by the arrow that initially calls the method.

Finally, there shouldn't be a display method going from the `ShoppingCart` entity, as the boundary object is already in control, and the very next thing it does is display the page anyway.

Figure 8-26 shows the corrected diagram.

■**Exercise** Figure 8-26 still lacks some detail around populating the LineItems list component—for example, where does the Edit Shopping Cart page get the names and quantities from for each Line Item in the list? Try redrawing the diagram with this additional detail added.

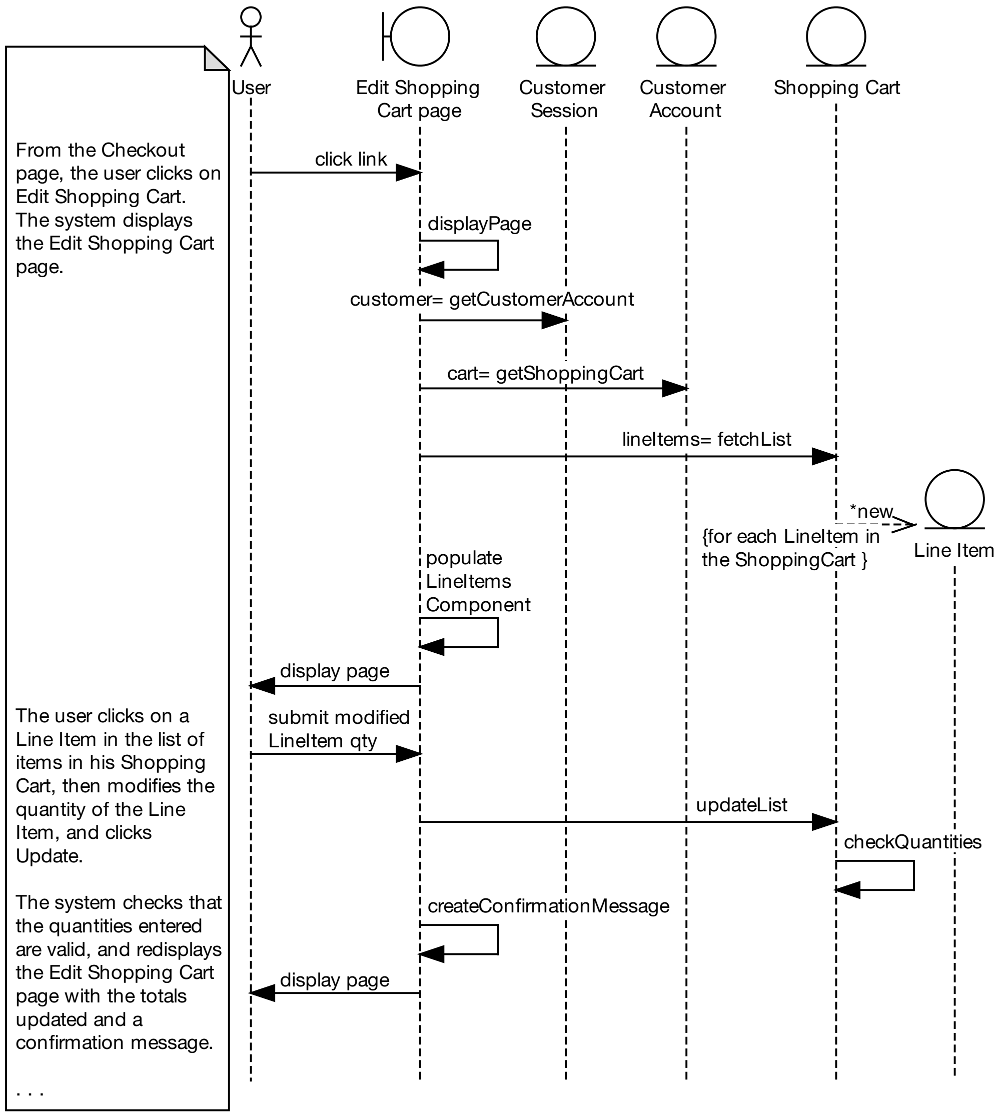

From the Checkout page, the user clicks on Edit Shopping Cart. The system displays the Edit Shopping Cart page.
The user clicks on a Line Item in the list of items in his Shopping Cart, then modifies the quantity of the Line Item, and clicks Update.
The system checks that the quantities entered are valid, and redisplays the Edit Shopping Cart page with the totals updated and a confirmation message.
. . .
User
Edit Shopping Cart page
Customer Session
Customer Account
Shopping Cart
click link
displayPage
customer= getCustomerAccount
cart= getShoppingCart
lineItems= fetchList
*new
{for each LineItem in the ShoppingCart }
Line Item
populate LineItems Component
display page
submit modified LineItem qty
updateList
checkQuantities
createConfirmationMessage
display page

Figure 8-26. *The corrected sequence diagram excerpt for Exercise 8-3*

More Practice

This section provides a list of modeling questions that you can use to test your knowledge of sequence diagramming. If you can't answer a question, review the relevant sections in this chapter until you *can* answer it.

1. The primary purpose of a sequence diagram should be
   - a) To show detailed flow of control for a use case
   - b) To specify real-time, finite-state behavior
   - c) To help make an optimal allocation of functions to classes within the context of a use case
   - d) To separate out alternate courses of action
2. A sequence diagram should always be
   - a) Cross-checked against a class diagram with operations shown on the classes
   - b) Reviewed to make sure all behavior specified in the use case is accounted for by messages between objects
   - c) Reviewed against a GUI prototype or storyboard to make sure all possible user actions are accounted for
   - d) All of the above
3. Readability of a sequence diagram is best accomplished by
   - a) Using detailed use case templates showing pre- and postconditions
   - b) Drawing it to the same level of detail as a robustness diagram
   - c) Following the two-paragraph rule and keeping the use cases short
   - d) Showing branching and conditional logic on the diagram
4. Which of the following statements is *not* true?
   - a) It's preferable to show all alternate course of actions on a single diagram rather than produce a separate diagram for each alternate.
   - b) It's OK to turn off "focus of control" if it gets in your way.
   - c) Sequence diagrams show essentially the same information as collaboration diagrams, in a different format.
   - d) You should always rewrite your use case text when drawing a sequence diagram.
5. What things should you consider when drawing a message between objects on a sequence diagram? List at least four criteria, and explain why you should consider each of them.
6. The allocation of behavior can be effectively accomplished by following a responsibility-driven thought process. Explain the premise of Responsibility-Driven Design (RDD). (Hint: A good book on RDD was written by Rebecca Wirfs-Brock.)
7. Is it a good idea to draw flowchart-level sequence diagrams that focus on branching and conditional logic? Why or why not?

8. How should the level of abstraction of the use case text that appears on the margin of the sequence diagram compare to the abstraction level of the diagram? (Hint: The diagram should show an object/message detailed design view.)

Discuss the pros and cons of each of the following possibilities:

a) Use case text should match the diagram's abstraction level.

b) Use case text should remain at the abstract, technology-free, implementation-independent requirements level as in the first-draft use cases.

c) Use case text should remain at the robustness diagram abstraction level (conceptual design), while the sequence diagram should show additional design detail.

Summary

In this chapter we covered **detailed design**, the second step in the two-step design process (the first step was preliminary design). Figure 8-27 shows where we are; the items covered in this chapter are shown in red.

Once you've drawn all the sequence diagrams for the use cases you're working on in the current release and updated your static model, then you can safely say that you've finished this stage in the process.

By now, you're almost ready to begin coding. There's just one last stop before code: the Critical Design Review (CDR), which we cover in the next chapter. It's an essential step, as it forms something of a reality check for your design.

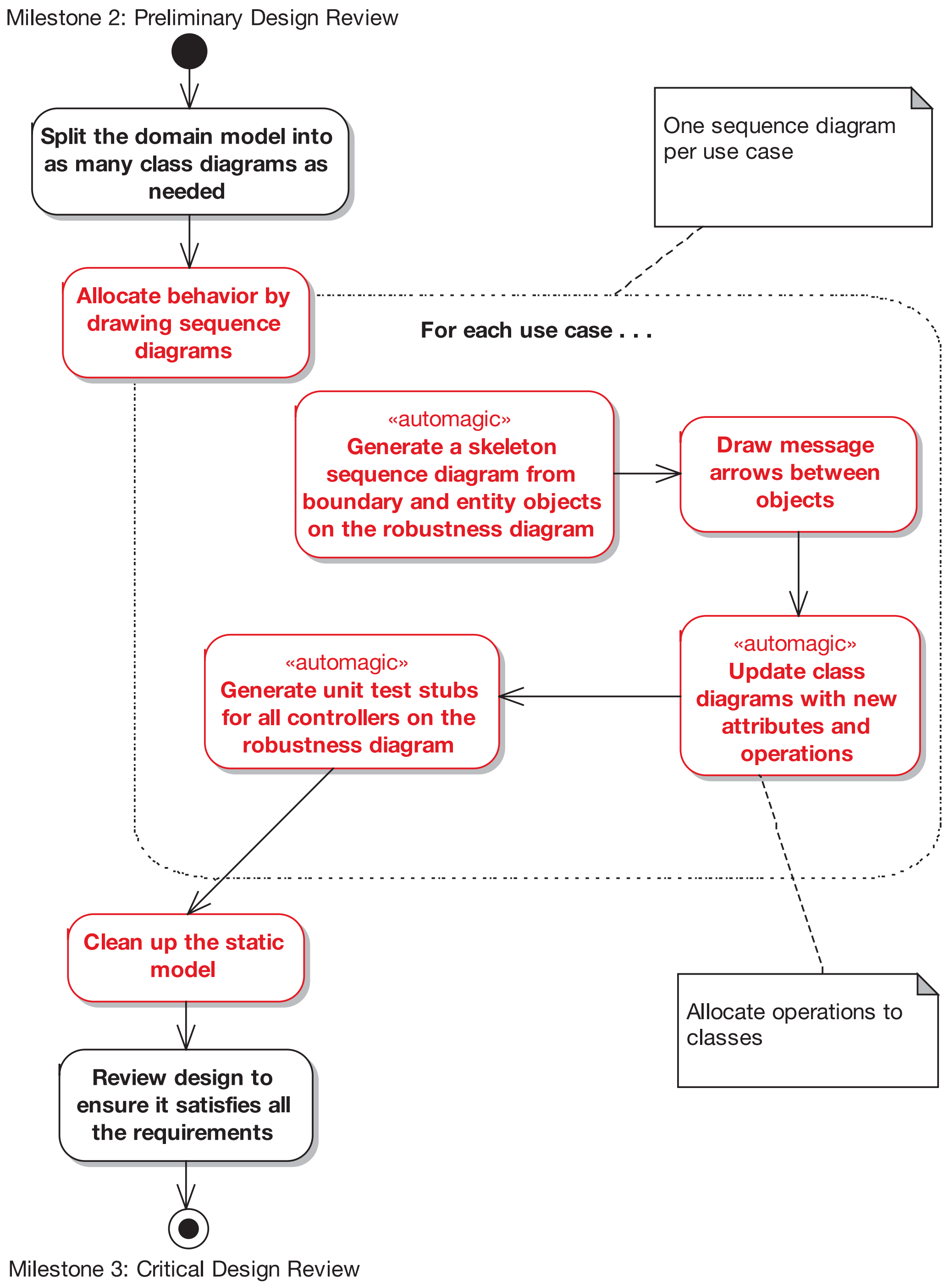

Milestone 2: Preliminary Design Review
Split the domain model into as many class diagrams as needed
One sequence diagram per use case
Allocate behavior by drawing sequence diagrams
For each use case . . .
«automagic» Generate a skeleton sequence diagram from boundary and entity objects on the robustness 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 8-27. *Activities during the detailed design stage*