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.

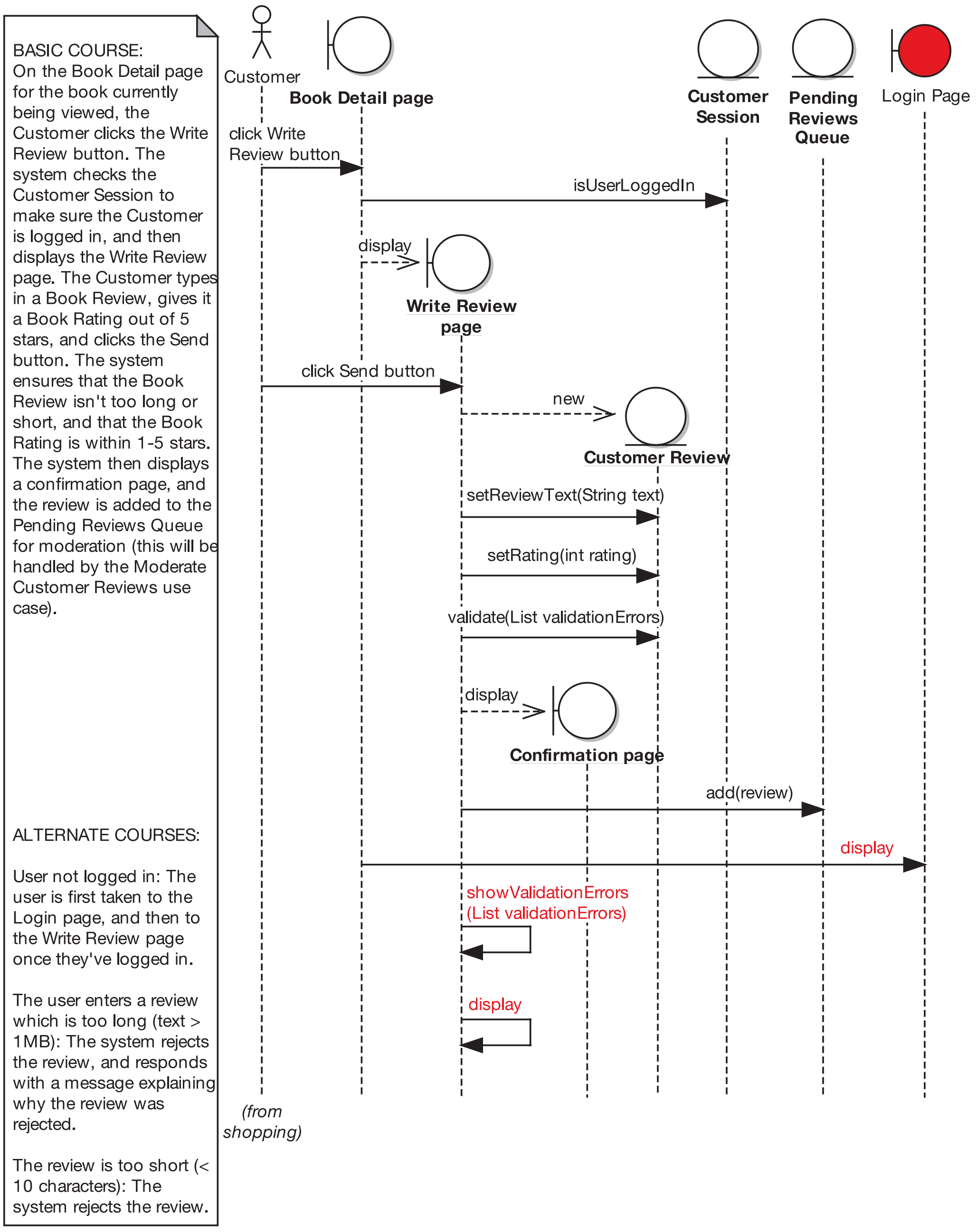
Referring back to the robustness diagram in Figure 6-7, you can see that the Customer clicks the Write Review button on the Book Detail page. The system then performs a check to see if the Customer is logged in. On the robustness diagram, this is represented as the "Is user logged in?" controller. This 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.

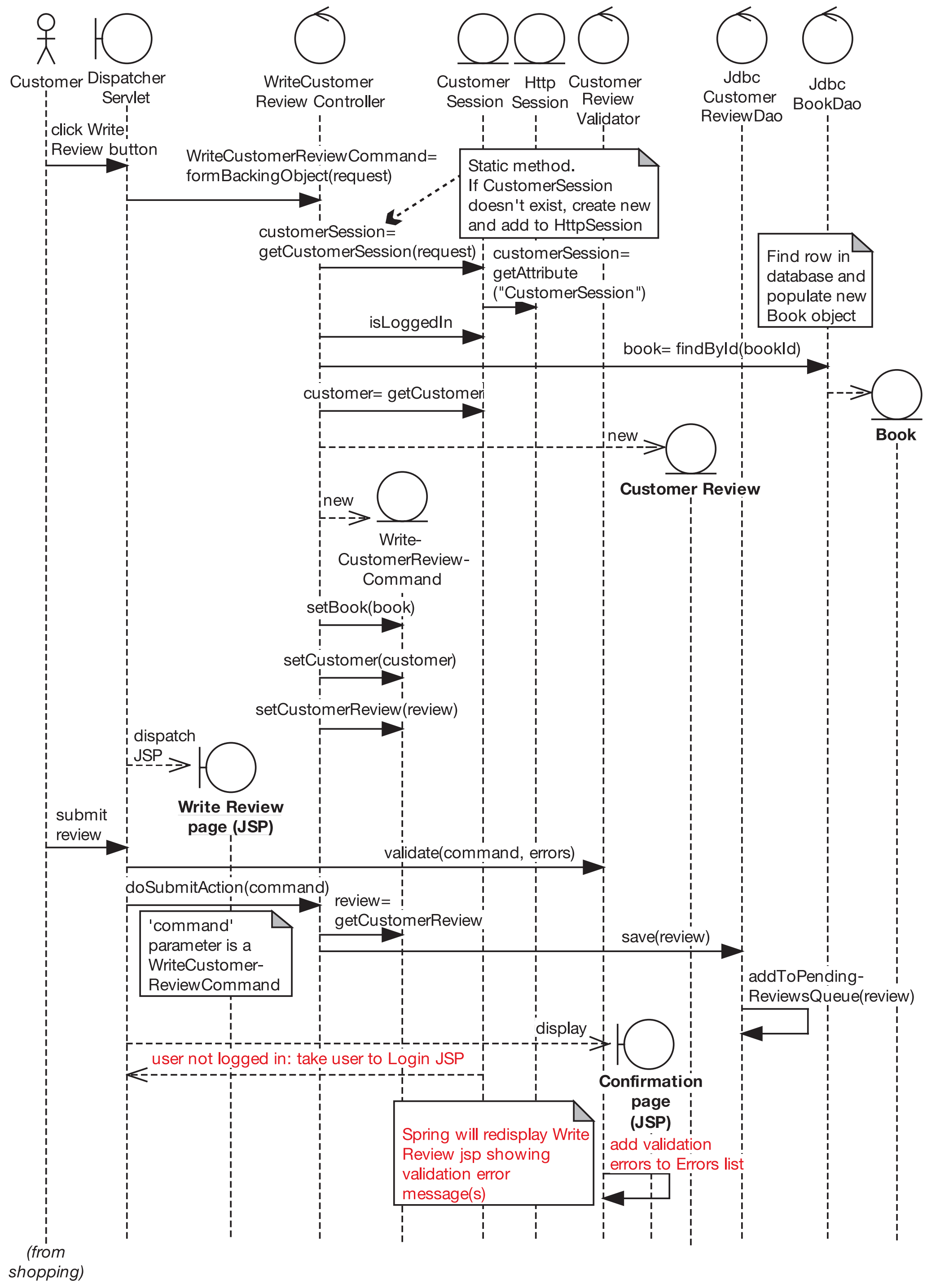

Customer
Dispatcher Servlet
WriteCustomer Review Controller
Customer Session
Http Session
Customer Review Validator
Jdbc Customer ReviewDao
Jdbc BookDao
click Write Review button
WriteCustomerReviewCommand= formBackingObject(request)
Static method. If CustomerSession doesn't exist, create new and add 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
'command' parameter is a WriteCustomer-ReviewCommand
save(review)
addToPending-ReviewsQueue(review)
display
user not logged in: take user to Login JSP
Confirmation page (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).

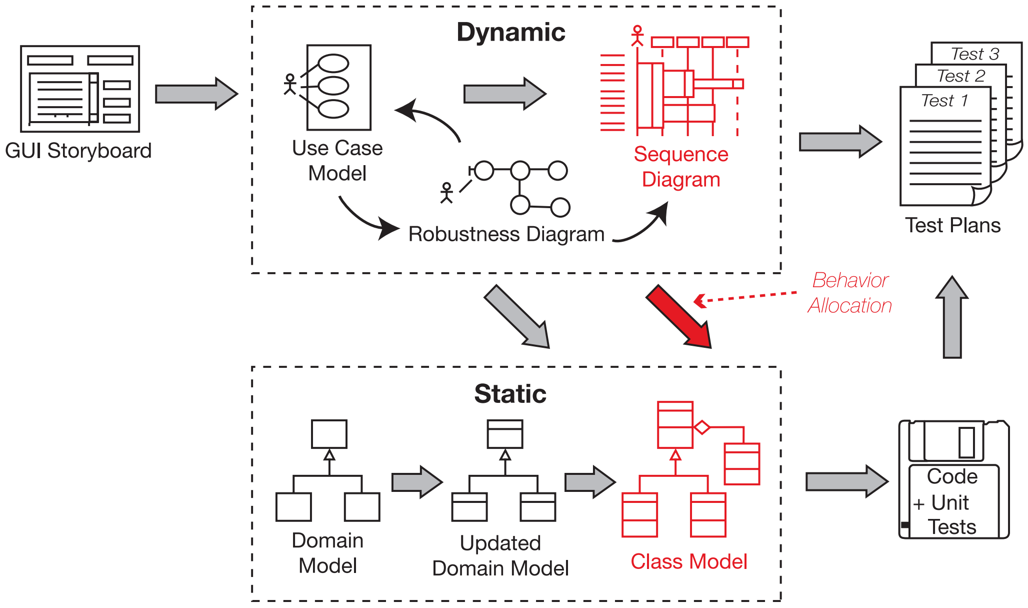


Figure 8-12. *Updating your static model, again*

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.

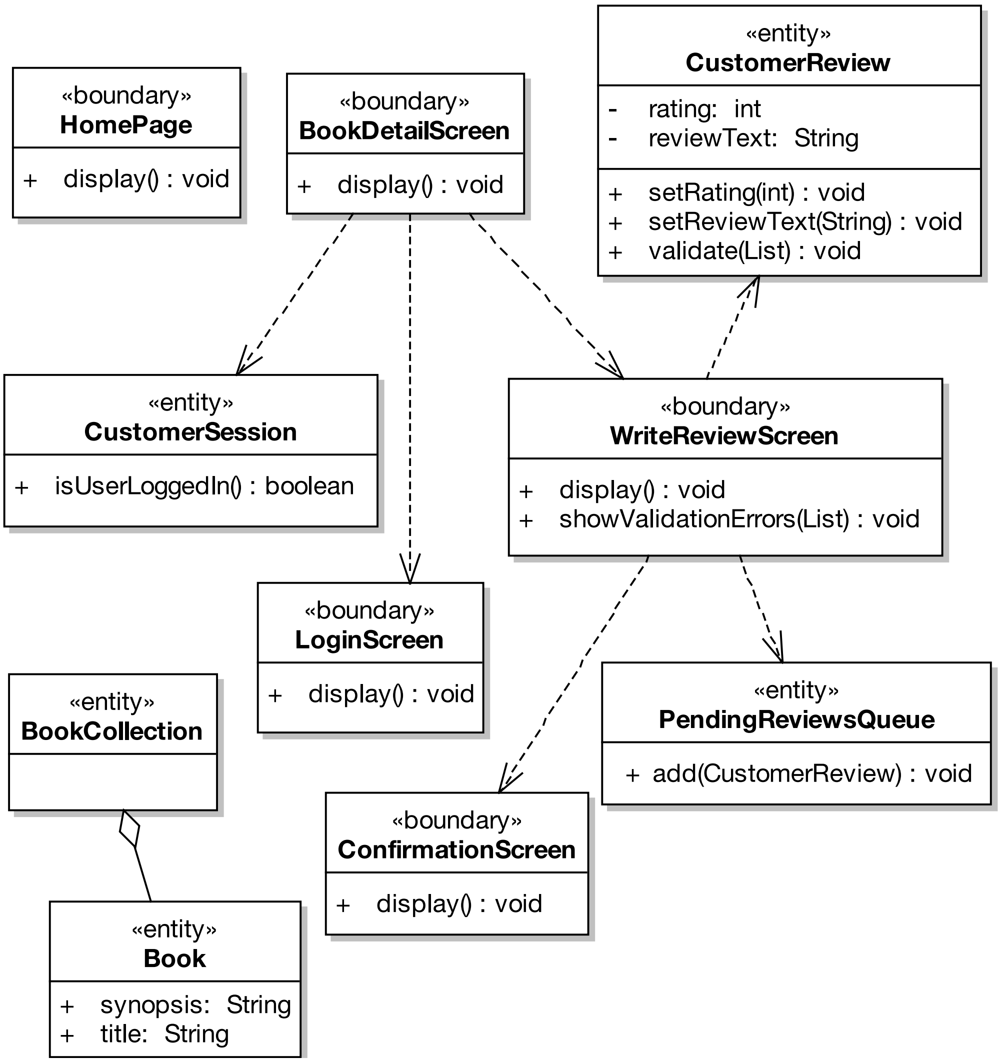


Figure 8-13. *Internet Bookstore static model based on the pure OO sequence diagram in Figure 8-10*

`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).

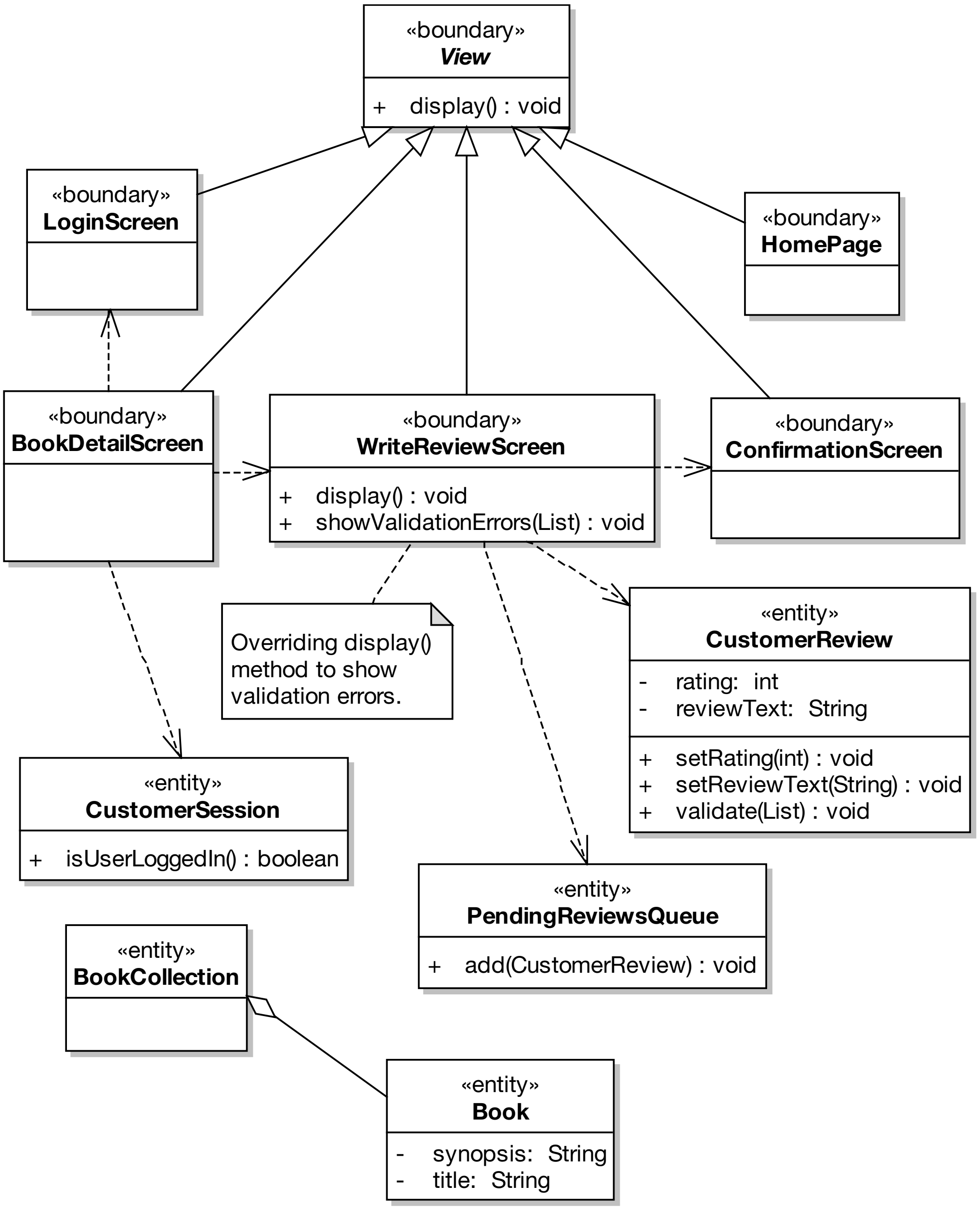


Figure 8-14. *Prefactored Internet Bookstore static model, still pure OO*

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.

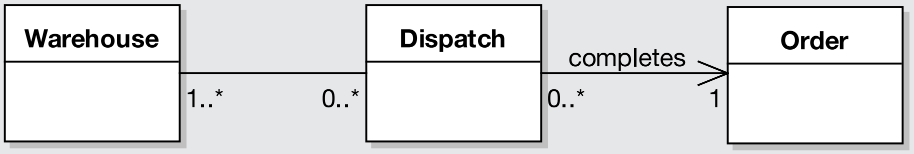


Figure 8-15. *Example class diagram notation for associations*

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.

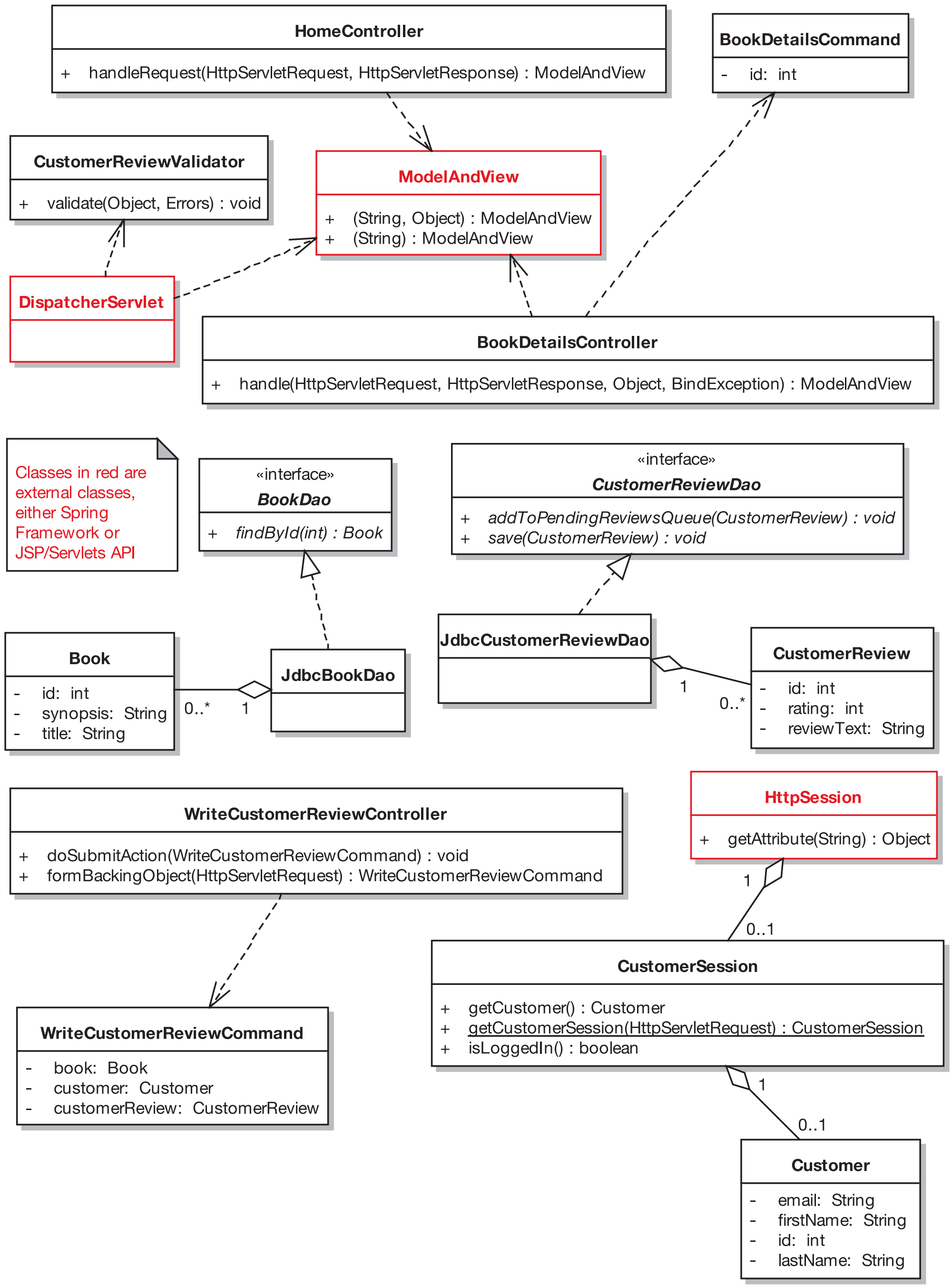

HomeController
+ handleRequest(HttpServletRequest, HttpServletResponse) : ModelAndView
BookDetailsCommand
- id: int
CustomerReviewValidator
+ validate(Object, Errors) : void
ModelAndView
+ (String, Object) : ModelAndView
+ (String) : ModelAndView
DispatcherServlet
BookDetailsController
+ handle(HttpServletRequest, 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
Book
- id: int
- synopsis: String
- title: String
0..*
1
JdbcBookDao
JdbcCustomerReviewDao
1
0..*
CustomerReview
- id: int
- rating: int
- reviewText: String
HttpSession
+ getAttribute(String) : Object
WriteCustomerReviewController
+ doSubmitAction(WriteCustomerReviewCommand) : void
+ formBackingObject(HttpServletRequest) : WriteCustomerReviewCommand
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.

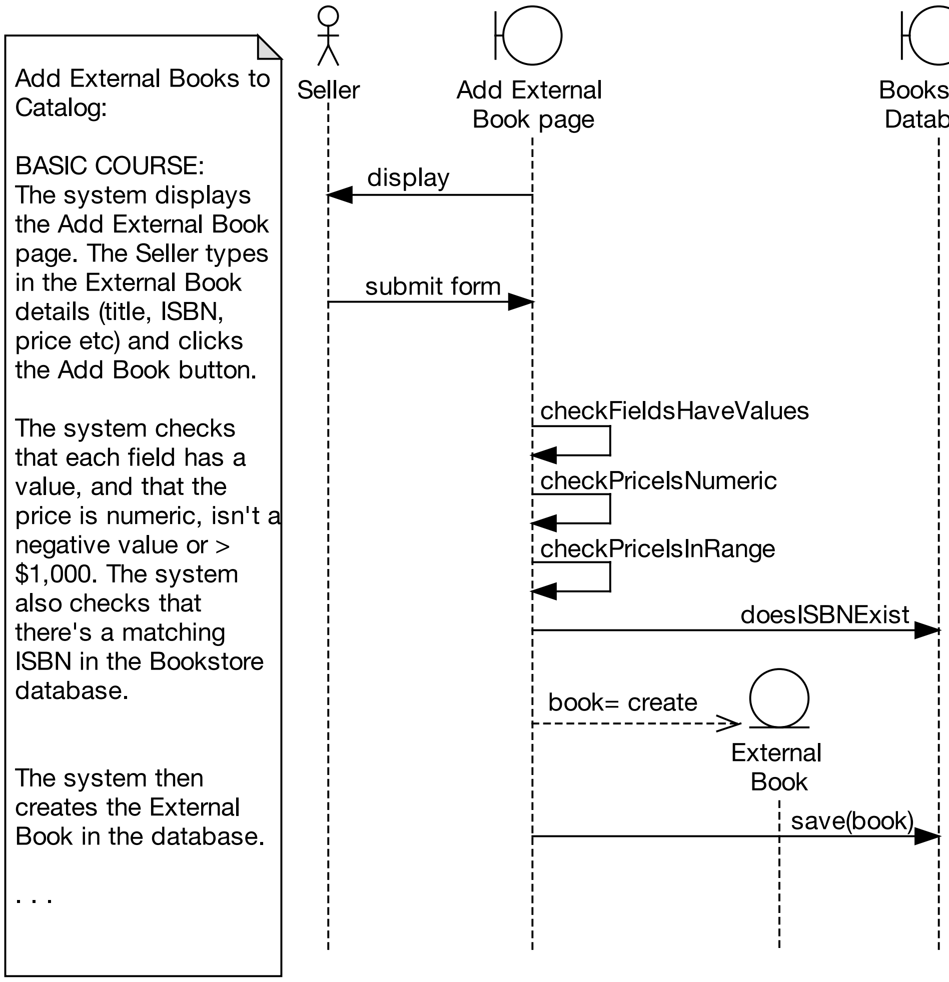

Add External Books to Catalog:
BASIC COURSE:
The system displays the Add External Book page. The Seller types in the External Book details (title, ISBN, price etc) and clicks the Add Book button.
The system checks that each field has a value, and that the price is numeric, isn't a negative value or > $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.

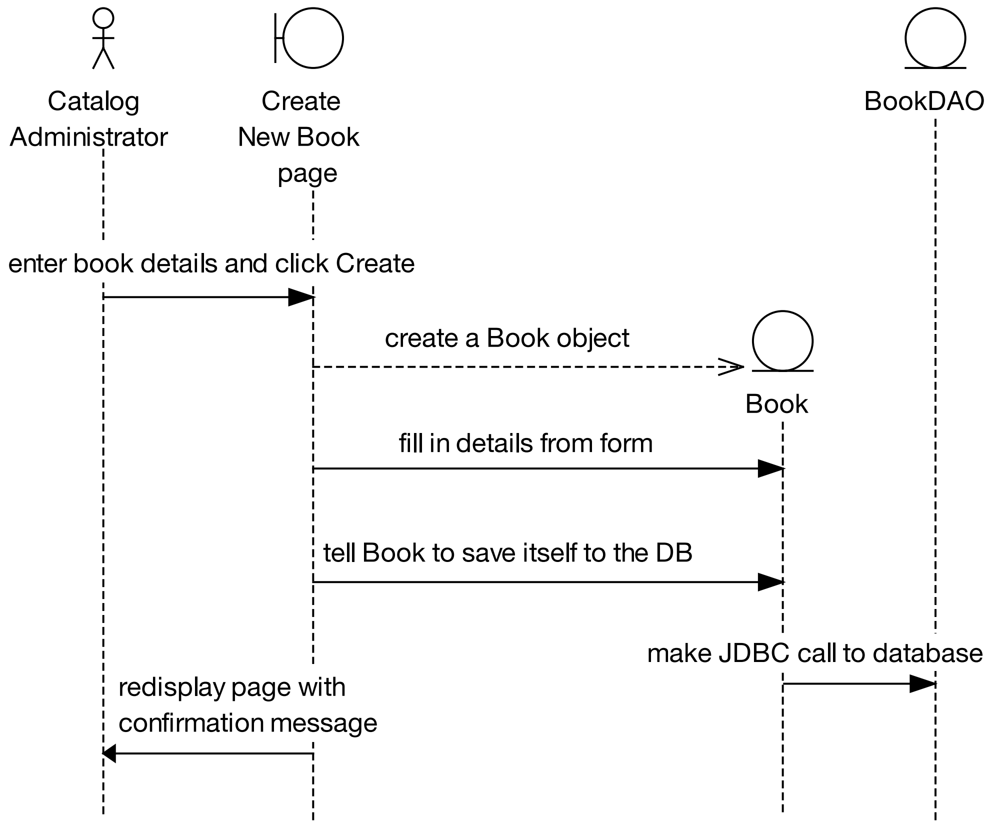

Catalog Administrator
Create New Book page
BookDAO
enter book details and click Create
create a Book object
Book
fill in details from form
tell Book to save itself to the DB
make JDBC call to database
redisplay page with confirmation message

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!

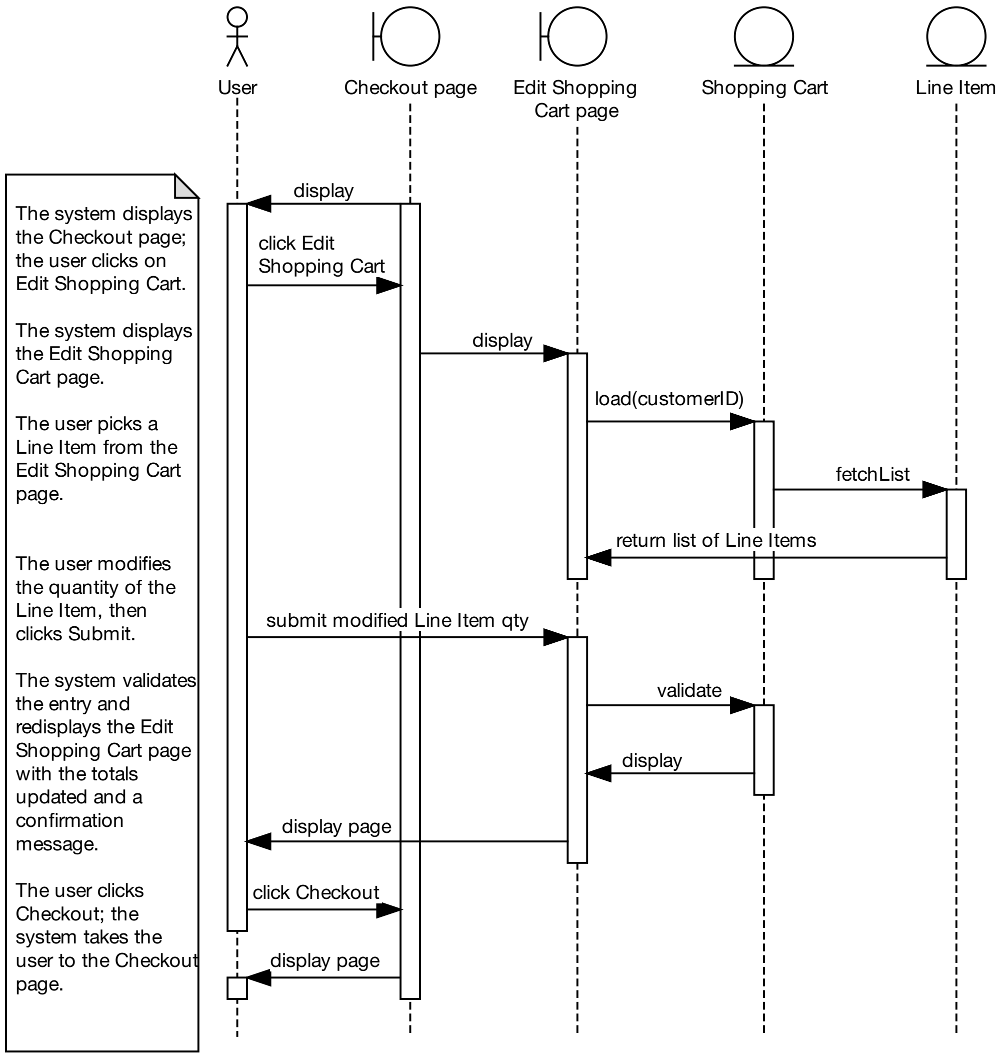

User
Checkout page
Edit Shopping Cart page
Shopping Cart
Line Item
The system displays the Checkout page; the user clicks on Edit Shopping Cart.
The system displays the Edit Shopping Cart page.
The user picks a Line Item from the Edit Shopping Cart page.
The user modifies the quantity of the Line Item, 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.

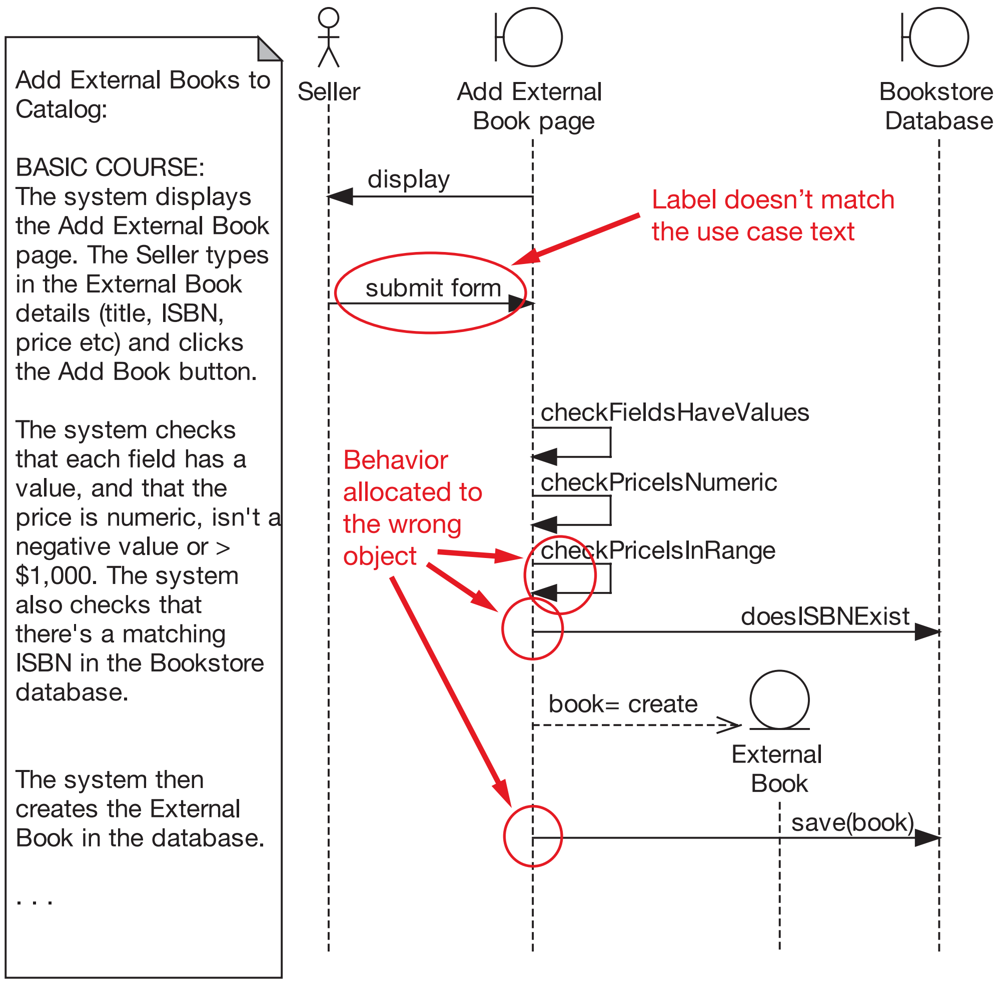
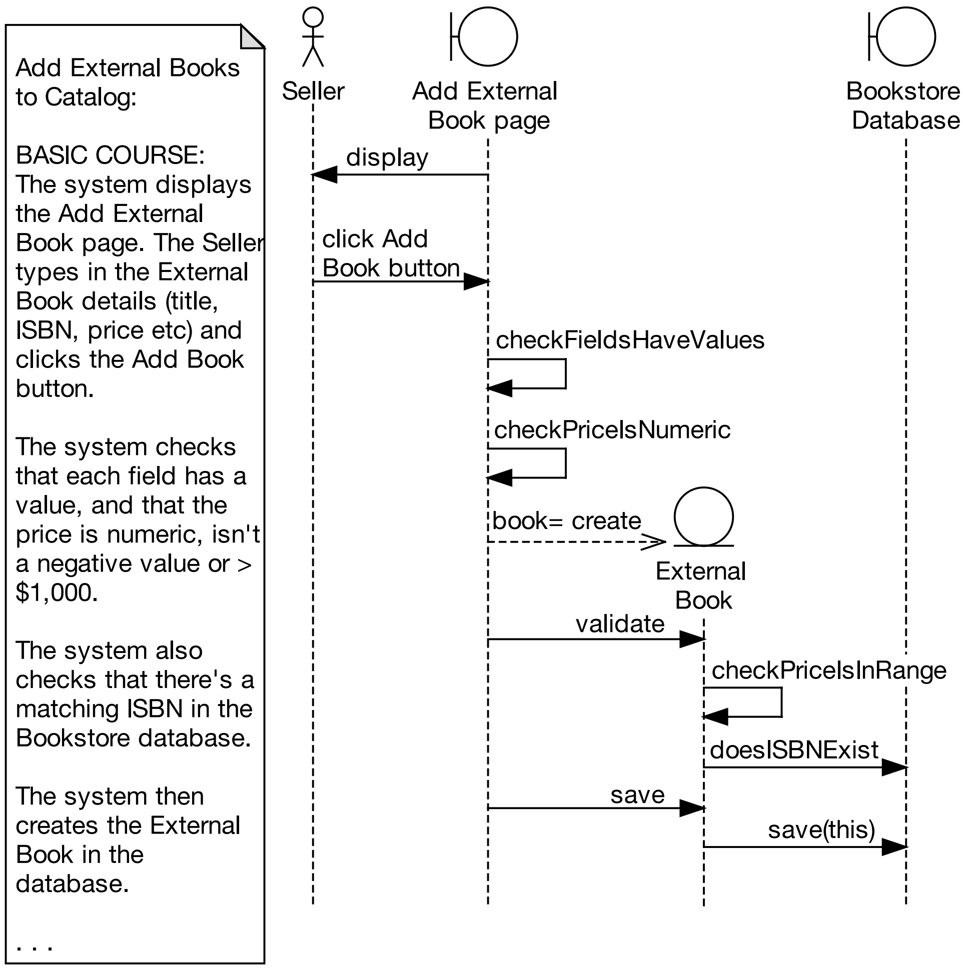

Add External Books to Catalog:
BASIC COURSE:
The system displays the Add External Book page. The Seller types in the External Book details (title, ISBN, price etc) and clicks the Add Book button.
The system checks that each field has a value, and that the price is numeric, isn't a negative value or > $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
checkPriceIsNumeric
checkPriceIsInRange
Behavior allocated to the wrong object
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.

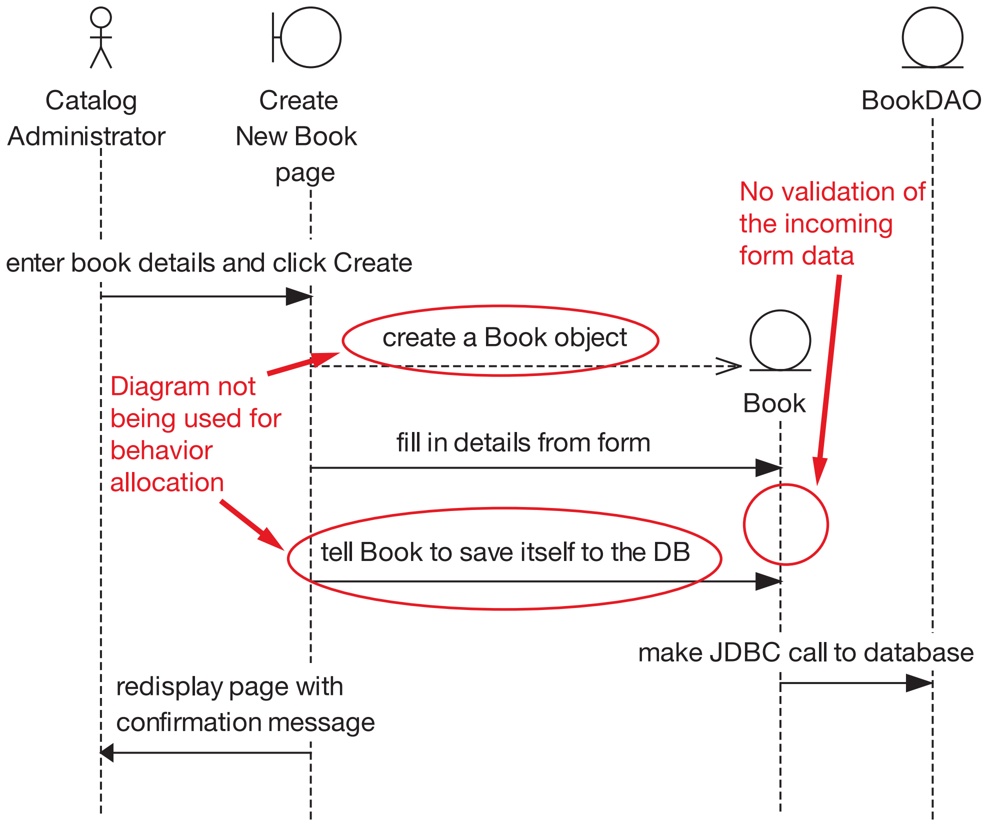

Catalog Administrator
Create New Book page
BookDAO
No validation of the incoming form data
enter book details and click Create
create a Book object
Book
Diagram not being used for behavior allocation
fill in details from form
tell Book to save itself to the DB
make JDBC call to database
redisplay page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

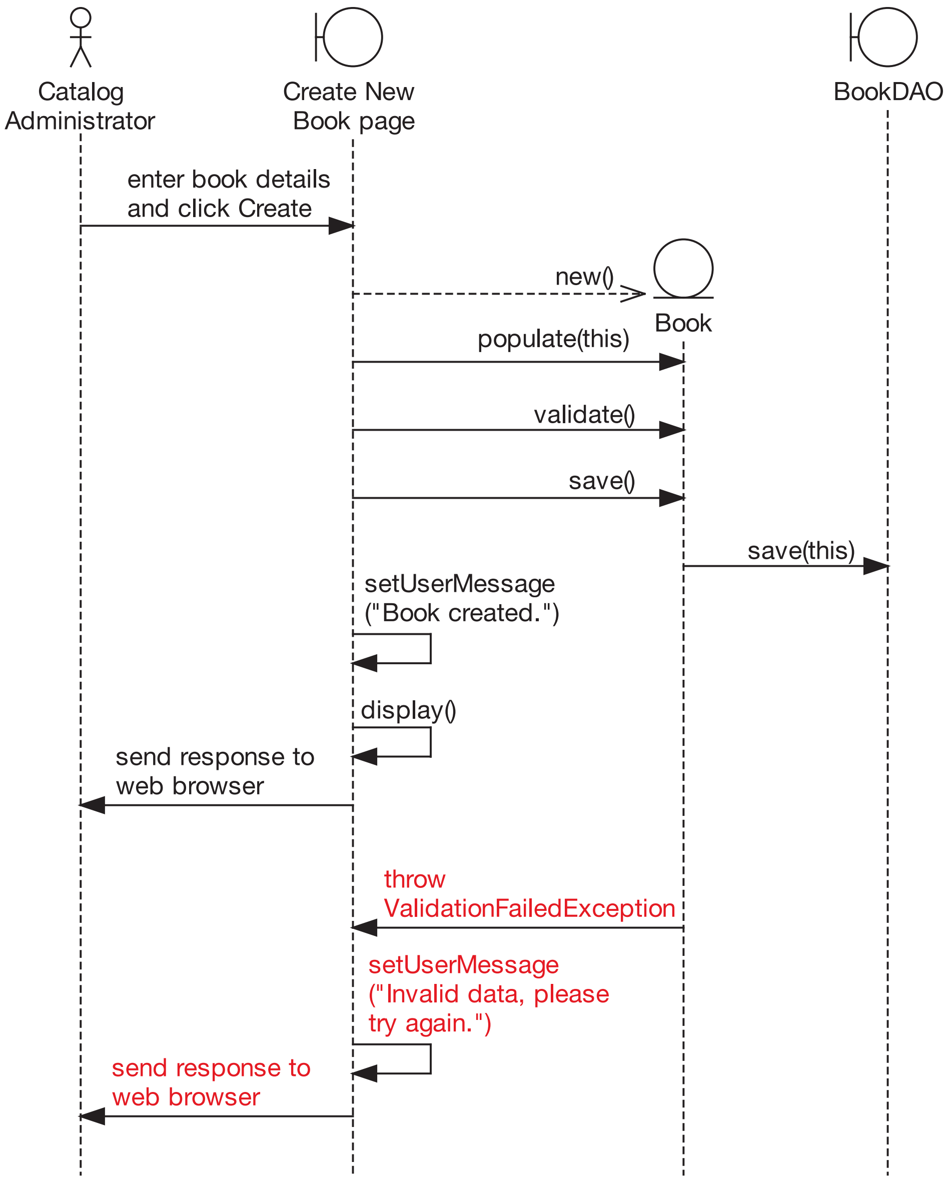

Catalog Administrator
Create New Book page
BookDAO
enter book details and click Create
new()
Book
populate(this)
validate()
save()
save(this)
setUserMessage ("Book created.")
display()
send response to web browser
throw ValidationFailedException
setUserMessage ("Invalid data, please try again.")
send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

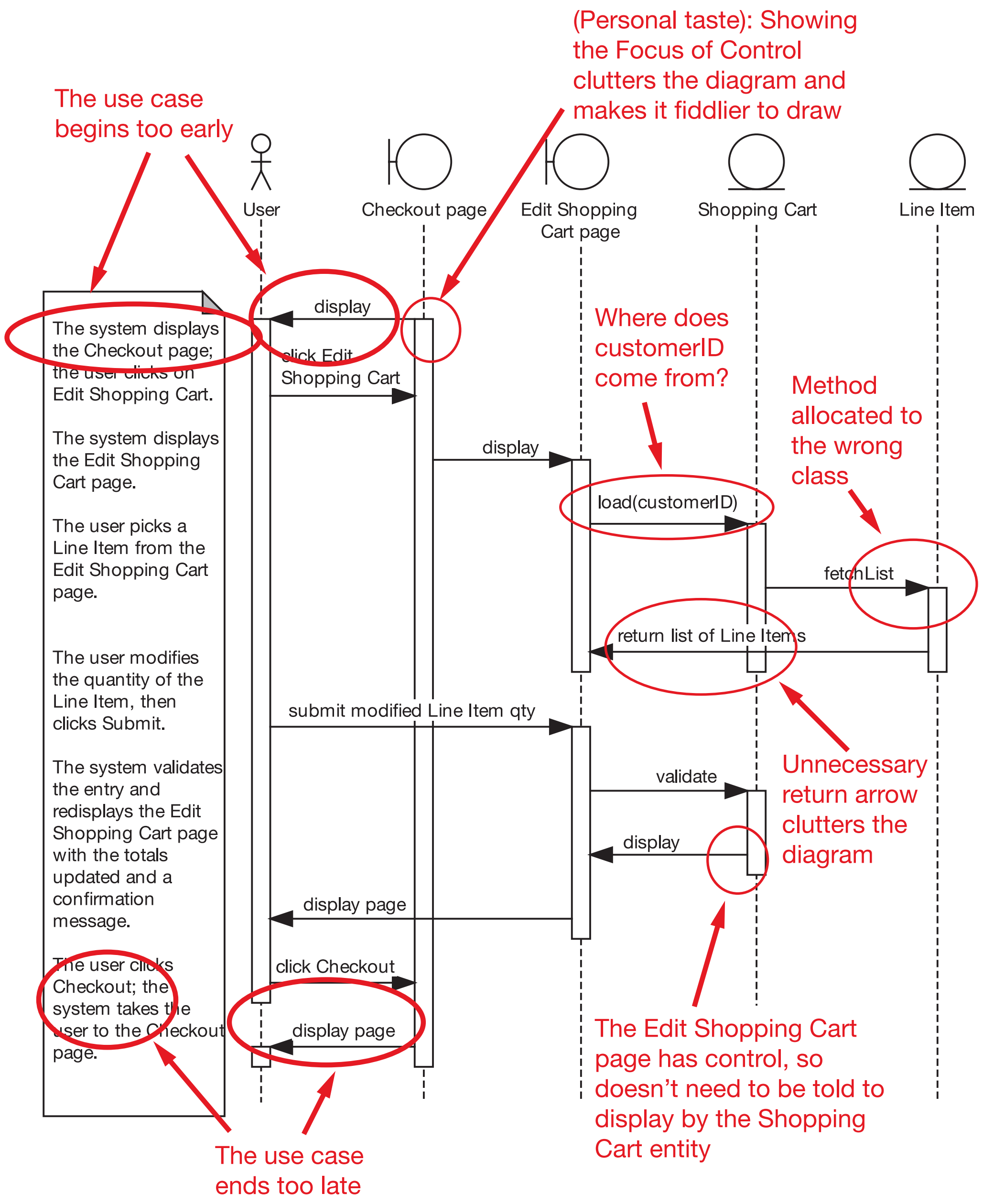

(Personal taste): Showing the Focus of Control clutters the diagram and makes it fiddlier to draw
The use case begins too early
User
Checkout page
Edit Shopping Cart page
Shopping Cart
Line Item
The system displays the Checkout page; the user clicks on Edit Shopping Cart.
The system displays the Edit 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?
load(customerID)
Method allocated to the wrong class
fetchList
return list of Line Items
Unnecessary return arrow clutters the diagram
submit modified Line Item qty
validate
display
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 `LineItem`s. 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.

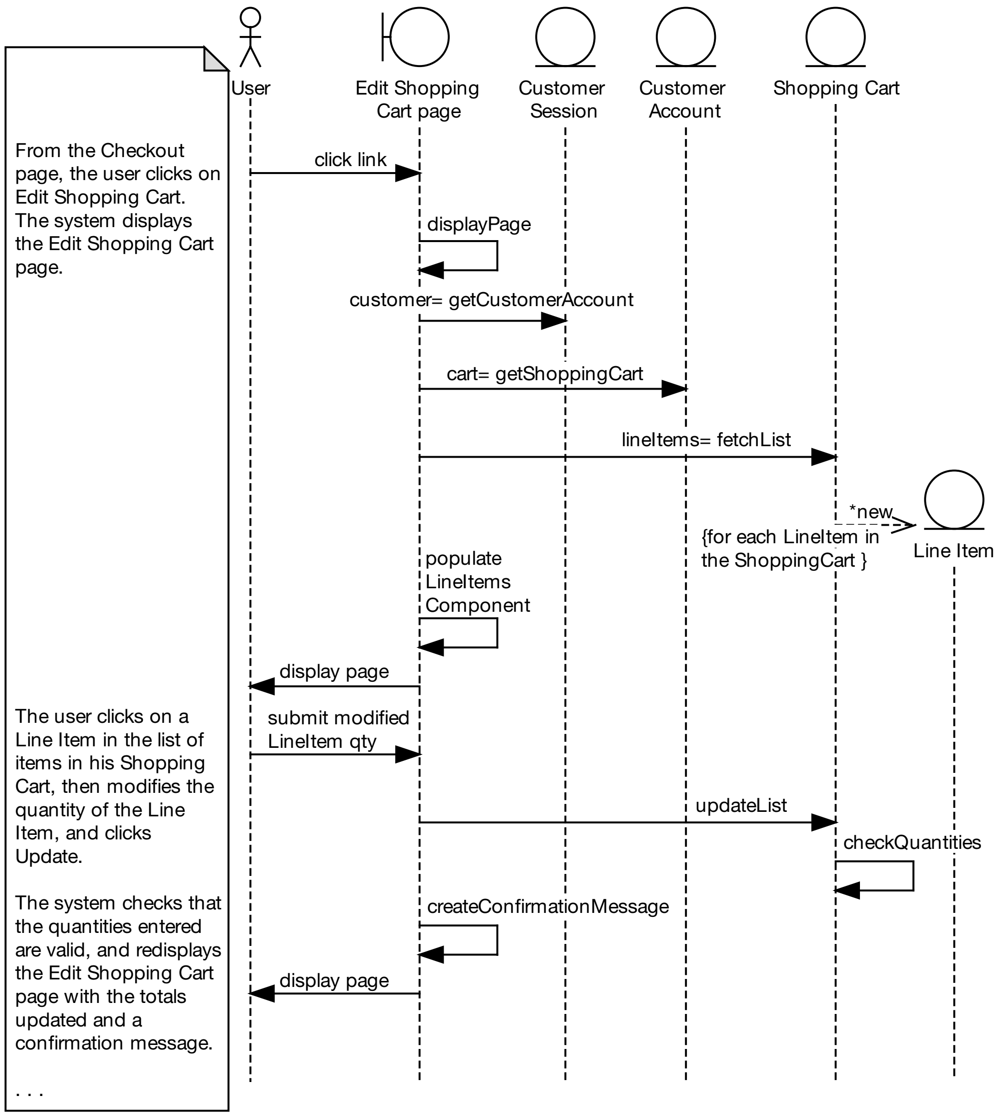

User
Edit Shopping Cart page
Customer Session
Customer Account
Shopping Cart
Line Item
From the Checkout page, the user clicks on Edit Shopping Cart. The system displays the Edit Shopping Cart page.
click link
displayPage
customer= getCustomerAccount
cart= getShoppingCart
lineItems= fetchList
*new
{for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