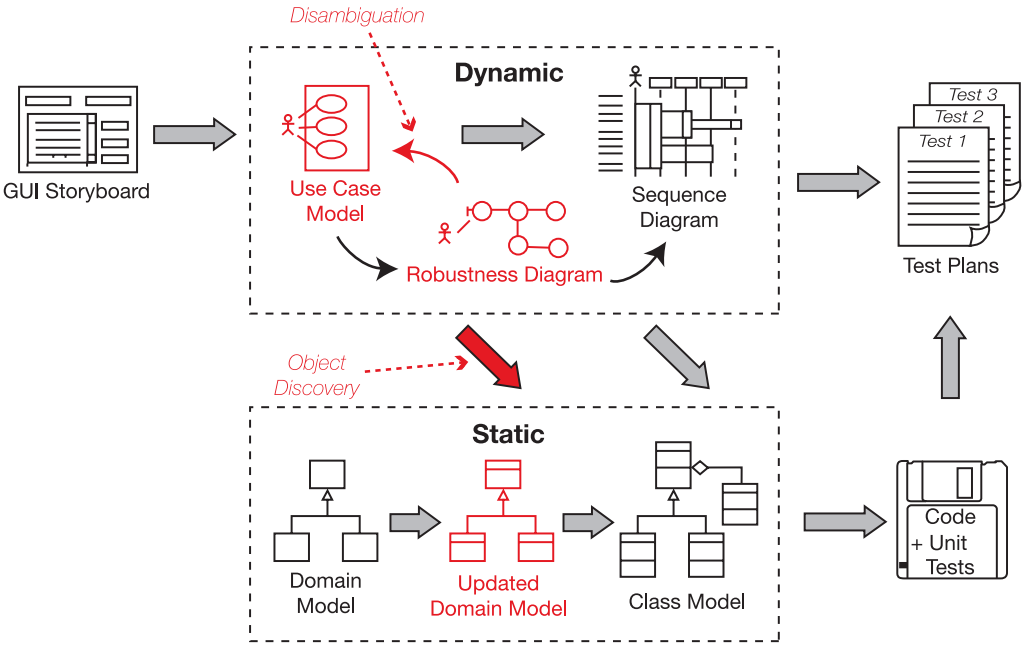
CHAPTER 5

Robustness Analysis



To get from use cases to detailed design (and then to code), you need to link your use cases to objects. The technique we describe in this chapter, robustness analysis, helps you to bridge the gap from analysis to design by doing exactly that. In a nutshell, it's a way of analyzing your use case text and identifying a first-guess set of objects for each use case. These are classified into boundary objects, entity objects, and controllers (which are often more like functions than objects).

The 10,000-Foot View

A robustness diagram is an object picture of a use case. The robustness diagram and the use case text have to match precisely, so the robustness diagram forces you to tie the use case text to the objects. This enables you to drive object-oriented designs forward from use cases, and this is really the "magic" of robustness analysis.

Drawing a robustness diagram ensures that the use case is **written in the context of the domain model**—that is, all the terms (nouns and noun phrases) that went into the domain model should also be used directly in your use case text.

Where Does Robustness Analysis Fit into the Process?

Looking at Figure 5-1, robustness analysis sort of takes place in the murky middle ground between analysis and design. If you think of analysis (i.e., the use cases) as the "what" and design as the "how," then robustness analysis is really preliminary design. During this phase, you start making some preliminary assumptions about your design, and you start to think about the technical architecture (also see Chapter 7) and to think through the various possible design strategies. So it's part analysis and part design.

It's also an important technique to remove ambiguity from (disambiguate) your use case text.

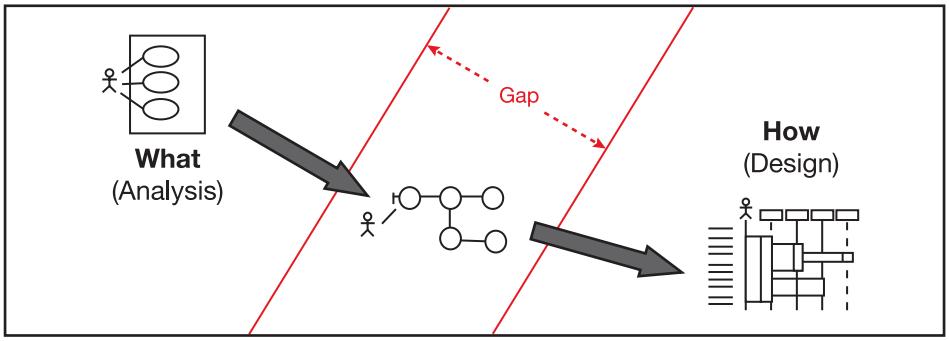


Figure 5-1. *Bridging the gap between "what" and "how"*

Like Learning to Ride a Bicycle

Learning this technique has a bit in common with learning to ride a bicycle. Until you "get it," robustness analysis can seem devilishly difficult, but once you do get it, it's really very simple. To jump-start your understanding, we'll walk through plenty of examples in this chapter. Experience has shown us that you usually need to draw six or so robustness diagrams before the penny drops and you suddenly get it. Just remember, **a robustness diagram is an object picture of a use case**.

Once you get the hang of it, you should be able to rattle off a robustness diagram in about ten minutes (or less) for each use case. Actually, as you'll see, **the trick is in writing your use case correctly**. If a robustness diagram takes more than ten minutes to draw, you can bet you're spending most of that time rewriting your use case text.

■ **Tip** Using a CASE tool can make your life easier, but robustness diagrams are really quick and simple diagrams that you can scribble on a piece of paper or a whiteboard. It's often very helpful to sketch your diagram on paper before attempting to draw it on the computer (especially when you're first learning the technique).

Anatomy of a Robustness Diagram

A robustness diagram is somewhat of a hybrid between a class diagram and an activity diagram. It's a pictorial representation of the behavior described by a use case, showing both participating classes and software behavior, although it intentionally avoids showing which class is responsible for which bits of behavior. Each class is represented by a graphical stereotype icon (see Figure 5-2). However, a robustness diagram reads more like an **activity diagram** (or a flowchart), in the sense that one object "talks to" the next object. This flow of action is represented by a line between the two objects that are talking to each other.

There's a direct 1:1 correlation between the flow of action in the robustness diagram and the steps described in the use case text.

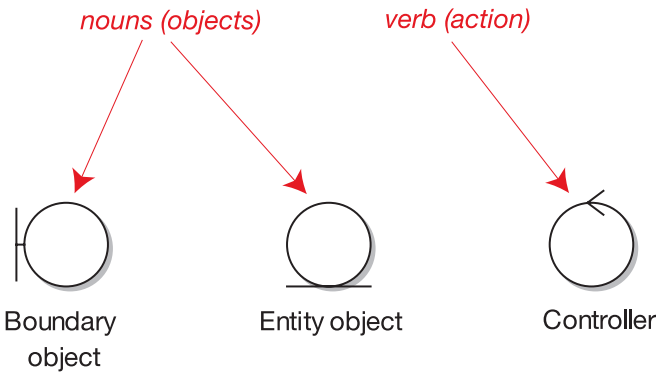


Figure 5-2. *Robustness diagram symbols*

The three class stereotypes shown in Figure 5-2 are as follows:

- **Boundary objects**: The "interface" between the system and the outside world (think back to Figure 3-2). Boundary objects are typically screens or web pages (i.e., the **presentation layer** that the actor interacts with).
- **Entity objects**: Classes from the domain model (see Chapter 2).
- **Controllers**: The "glue" between the boundary and entity objects.

It's useful to think of boundary objects and entity objects as being nouns, and controllers as being verbs. Keep the following rules in mind when drawing your robustness diagrams:

- Nouns can talk to verbs (and vice versa).
- Nouns can't talk to other nouns.
- Verbs can talk to other verbs.

We'll revisit these rules later in this chapter (see Figures 5-8 and 5-9).

■Exercise Two of the following are legal constructs, but which two?

a. Boundary ➤ Controller ➤ Entity

b. Entity ➤ Entity

c. Controller ➤ Controller

d. Boundary ➤ Boundary ➤ Controller

These rules **help to enforce a noun-verb-noun pattern in your use case text**. If your use case text follows this pattern, robustness diagrams are a snap to draw; if not, the diagrams can be really difficult to draw.

Think of this as an early warning signal: if you can't draw a simple ol' robustness diagram from a use case, how are you ever going to create a detailed design from it? **Sequence diagrams are completely noun-verb-noun in nature**: the objects are the nouns, and the messages that go between them are the verbs. So by getting your text in noun-verb-noun format now, you're making the detailed design task much easier than it would otherwise be.

Robustness analysis provides a sanity check for your use cases.

Robustness Analysis in Theory

In this section, we describe the theory behind robustness analysis, interspersed with examples from the Internet Bookstore project. We'll begin with our top 10 robustness analysis guidelines.

Top 10 Robustness Analysis Guidelines

The principles discussed in this chapter can be summed up as a list of guidelines. Our top 10 list follows.

10. Paste the use case text directly onto your robustness diagram.
9. Take your entity classes from the domain model, and add any that are missing.
8. Expect to rewrite (disambiguate) your use case while drawing the robustness diagram.
7. Make a boundary object for each screen, and name your screens unambiguously.
6. Remember that controllers are only occasionally **real control objects**; they are more typically **logical software functions**.
5. Don't worry about the direction of the arrows on a robustness diagram.
4. It's OK to drag a use case onto a robustness diagram if it's invoked from the parent use case.

3. The robustness diagram represents a preliminary conceptual design of a use case, not a literal detailed design.

2. Boundary and entity classes on a robustness diagram will generally become object instances on a sequence diagram, while controllers will become messages.

1. Remember that a robustness diagram is an "object picture" of a use case, whose purpose is to force refinement of both use case text and the object model.

Let's walk through the items in this list in more detail.

10. Paste the Use Case Text Directly onto Your Robustness Diagram

Doing this really helps to reinforce the fact that you're drawing an object picture of the events described in the use case. Plus, you'll **work through the use case a sentence at a time** as you draw the diagram, so it's handy to have the text nearby.

Figure 5-3 shows an example work-in-progress robustness diagram for the Internet Bookstore, for the *Login* use case. This is a snapshot of the diagram in its early stages. So far, only the first few sentences of the use case have been drawn onto the diagram.

In Figure 5-3, the use case text has been pasted directly into a note on the diagram. Because the robustness diagram is essentially a pictorial representation of the use case, it helps to have the text right there on the diagram: **they're two different views of the same thing**, so you should be able to walk through the text and trace it on the diagram (and vice versa).

■**Tip** Using a CASE tool such as EA, it's possible to create a live link between the use case and the note on the robustness diagram, so that if the use case is updated, the note on the diagram is updated automatically.

■**Exercise** We show the completed version of this diagram later, in Figure 5-5. But before you take a look, try completing the diagram, following the example of the controllers and message arrows that we've added so far in Figure 5-3. Simply follow the use case text, and draw the literal interpretation into the diagram (without trying to think about design details. Remember, you're not doing a real OO design yet—you're doing ***just enough preliminary design to validate that you understand the use case***).

(Hint: It's early days yet, as we've only barely introduced the basic concepts, so expect to make some mistakes. But don't be discouraged; the intention here is simply to give it a try, and then think about the diagram that you drew while you read the next few pages.)

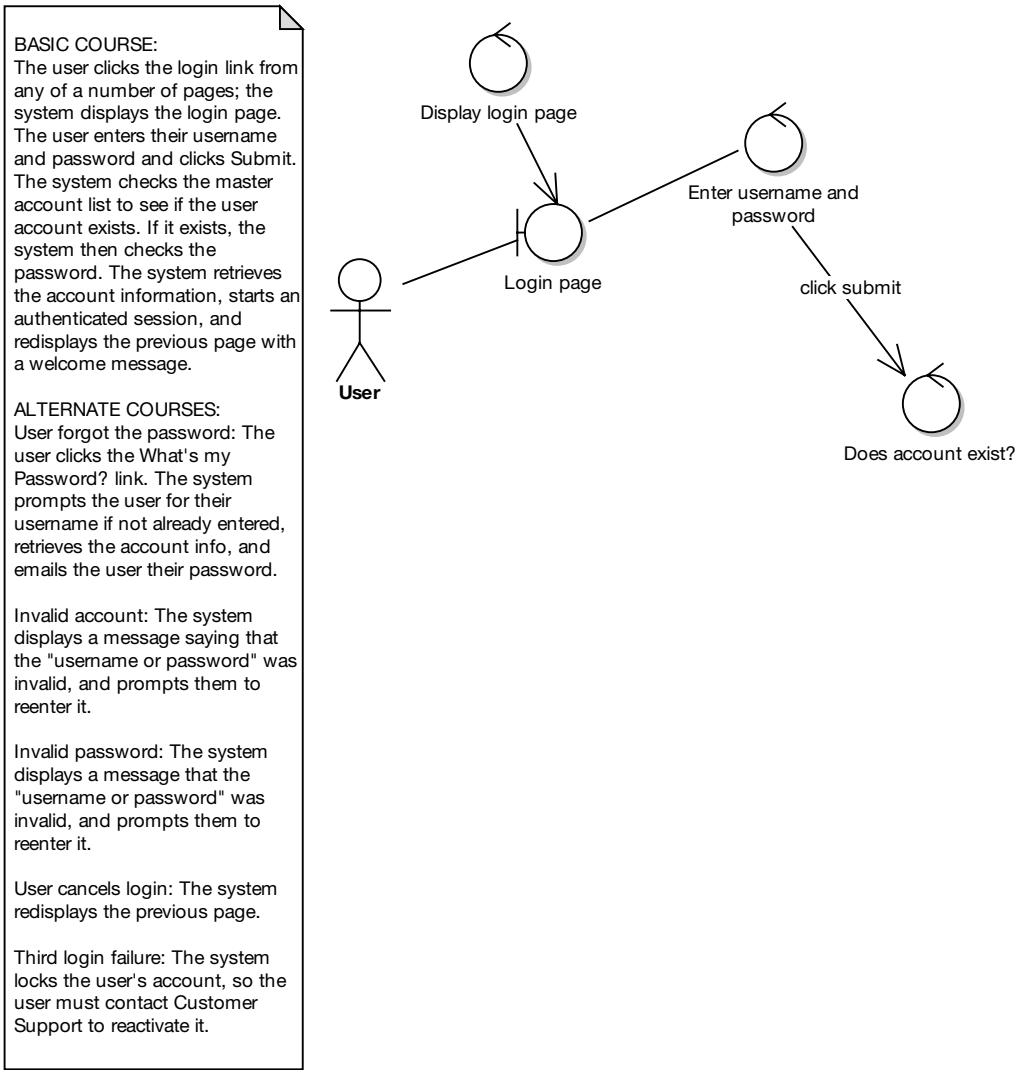


Figure 5-3. *Partially completed robustness diagram with the use case text pasted in*

9. Take Your Entity Classes from the Domain Model, and Add Any That Are Missing

Most of the entities on your robustness diagram will come from your domain model. However, since you time-boxed your initial domain modeling effort at a couple of hours, it's natural to expect that you might be missing some domain classes. When you're drawing robustness diagrams and this happens, make sure you **add the missing classes into the domain model**.

ICONIX Process **assumes that your initial domain model will be incomplete** and expects that missing objects will be discovered during robustness analysis. In this book, we refer to this process as ***object discovery***.

TYING YOUR USE CASE TO THE DESIGN

The robustness diagram ties three elements to your use case: the GUI, the domain classes, and an intended list of software functions (see Figure 5-4).

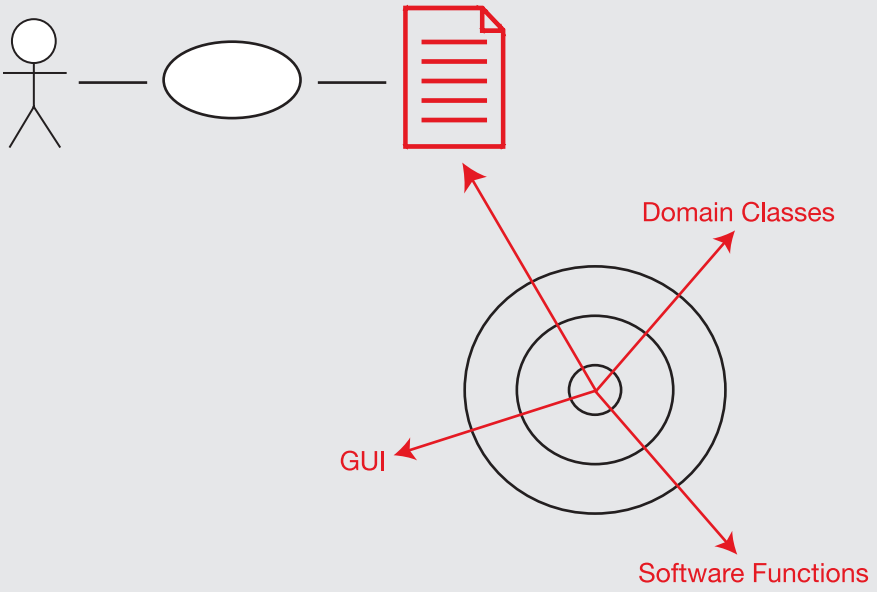


Figure 5-4. *Robustness diagrams tie three elements to your use case.*

As Figure 5-4 shows, the behavior requirements defined in your use case need to touch on several different aspects of the system, including how users interact with the GUI and manipulate the core objects from the problem domain. In between the GUI and the domain objects is the place where the software functions live.

On robustness diagrams, you describe GUI elements using boundary objects, software functions using controllers, and domain objects using entities. Note that this is substantially different from a collaboration diagram, which is sometimes confused with a robustness diagram. (Collaboration diagrams simply show object interactions.)

8. Expect to Rewrite Your Use Case While Drawing the Robustness Diagram

Experience has shown that **first-draft use cases** tend to exhibit the following characteristics: they **are typically vague, ambiguous, incomplete, and incorrect**. Small wonder that so many projects have struggled with use cases in the absence of a **disambiguation technique** like robustness analysis. Removing ambiguity from use cases is one of the primary purposes of this technique.

The "magic" of this technique is in reality hard work: **drawing a robustness diagram forces you to work through the use case one sentence at a time**. This simple act almost

always brings to the surface errors in the first-draft use case text, so it's important to rewrite the use case in parallel with drawing the robustness diagram.

7. Make a Boundary Object for Each Screen

Drawing a robustness diagram can enforce unambiguous naming (or, as we like to say, *disambiguated nomenclature* of your boundary objects[1]). If you see boundary objects labeled "web page" on a robustness diagram, stop, figure out the name of the page, and use the real name.

6. Remember that Controllers Are Typically Logical Software Functions

It's certainly possible to have control-intensive classes in your design (e.g., manager classes), and you can definitely represent these as controllers. However, don't presume that every controller on a robustness diagram will represent an actual control class. In many cases, a controller on a robustness diagram is simply used as a placeholder for a software function. Overuse of controller classes (e.g., one use case controller per use case) in a design can lead us back to functional decomposition, so controller classes should be used sparingly.[2] Showing a mix of objects and functions is one of the other ways in which a robustness diagram is substantially different from a collaboration diagram.

If you see a cluster of controllers on a robustness diagram that are all communicating with each other, then that's a good candidate for a manager class (especially if the finite state behavior is nontrivial). If you feel like you might need to draw a state diagram for the use case, you might also need a controller class, but the majority of your use cases generally aren't likely to be state-intensive (even in some real-time systems).

5. Don't Worry About the Direction of the Arrows on a Robustness Diagram

Remember that your robustness diagram has two main missions in life:

- To force you to **disambiguate your use case text**
- To help you to **discover missing objects** in your domain model

Which direction the arrowheads point on the robustness diagram does nothing to further either of these goals. As a consequence, the direction of the arrows just . . . doesn't . . . ***matter***. Seriously. Trust us. It really doesn't. Oh, and one other thing: it's not important.

Formally speaking, arrows on robustness diagrams represent *communication associations*. You can show either data flow or control flow and, in case we didn't mention it before, the direction of the arrows is **not** important.

---

1. For the benefit of any XPers who might be reading this book (!), that's "DisambiguatedNomenclatureOfYourBoundaryObjects."
2. As you'll see later, the current trend seems to be an even greater overuse of controller classes, where each software function actually has a controller class. It seems to us that the industry might have taken a giant step backward with this sort of thinking.

4. Show Invoked Use Cases on Your Robustness Diagram

It's OK to drag a use case onto a robustness diagram if it's invoked from the parent use case.

Not only is it OK to do this, but it's also the simplest way to show one case being invoked by another on a robustness diagram. In fact, it's the only reasonable way that we've found. Try it—it works really well.

3. The Robustness Diagram Represents a Preliminary Conceptual Design of a Use Case

Here are a couple of fundamental truths about system development:

- It's a good idea to fully understand the requirements before doing a design.
- It's often **impossible to fully understand the requirements without doing some exploratory design**.

These two statements may seem contradictory, but the solution is quite simple: you can do a conceptual design for the purpose of **validating the behavior requirements *before doing the real design***, which you're going to code from. The robustness diagram represents the conceptual design, whereas the real design is shown on the sequence diagrams.

Programmers often have trouble with robustness diagrams because they're accustomed to thinking in terms of concrete detailed designs, and they need to take a step back from being literal-minded and learn to think at a slightly more abstract, conceptual level. This can be tricky, as programming tends to be a very literal-minded skill. However, once you've mastered the skill of **manipulating designs at the conceptual level** of abstraction, you'll find that a number of benefits result, especially the ability to write precise and unambiguous use cases.

The good news is, mastering this skill doesn't require you to retire to a Tibetan monastery to meditate and practice for several years; it only takes a few hours of drawing diagrams.

2. Objects on Your Robustness Diagram Will "Morph" into the Detailed Design

Boundary and entity classes on a robustness diagram will generally become object instances on a sequence diagram, while controllers will become messages. It's also advisable to create test cases for the controllers.

Keep in mind that both boundary objects and entity objects are nouns, and that controllers are verbs (i.e., an action performed on an object). As such, it makes sense that the controllers (the actions) will become methods on the boundary and entity classes.

1. Remember That a Robustness Diagram Is an "Object Picture" of a Use Case

A robustness diagram is an "object picture" of a use case, whose purpose is to force refinement of both use case text and the object model. ***Robustness diagrams tie use cases to objects (and to the GUI).***

A robustness diagram isn't the same as a UML collaboration diagram. You show object-to-object communication on a collaboration diagram, but a robustness diagram is quite literally an object picture of a use case. Since the robustness diagram and the use case text have to match precisely, the robustness diagram forces you to tie the use case text to the objects, thus enabling you to drive object-oriented designs forward from use cases.

An important implication of all of this is that, because the robustness diagram must show all of the use case, ***it must show not just the basic course, but all the alternate courses as well***

(all on the same diagram). This is a good reason why your use cases should follow the two-paragraph rule (see Chapter 3).

Figure 5-5 shows the completed diagram for the *Login* use case (which we began earlier in this chapter), showing the basic course and the alternate courses.

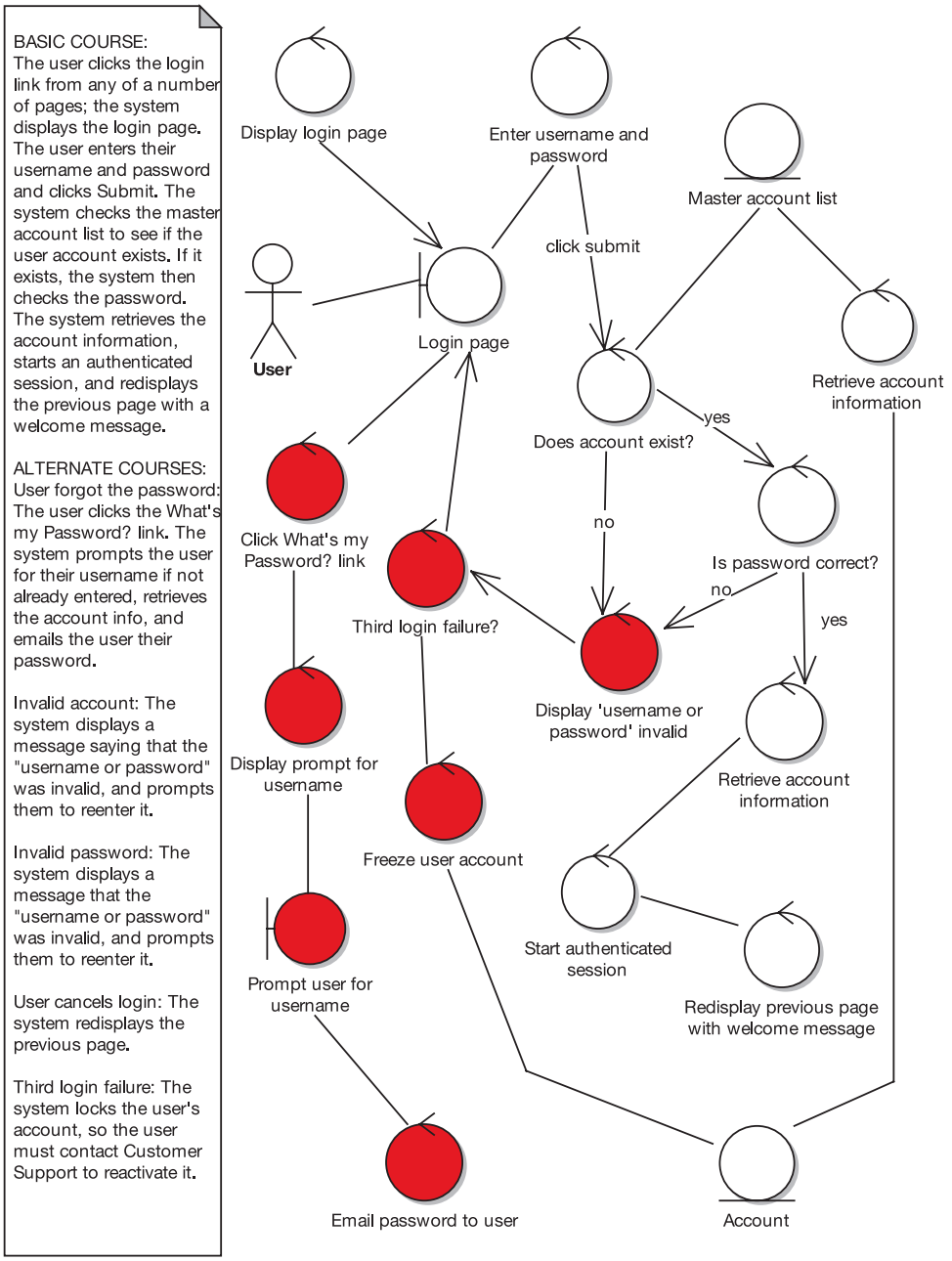


Figure 5-5. *Example robustness diagram*

Tip In Figure 5-5, some of the objects are shaded red. These are the objects (mainly controllers) for the alternate courses. Though it isn't essential, it's helpful to show the alternate courses in a different color from the basic course. The same effect can be achieved (and has an additional benefit as a form of review) by printing out the diagram and using different colored highlighter pens to trace the basic course and alternate courses.

DO I REALLY NEED ALL THOSE #$%^ DISPLAY CONTOLLERS?

A common issue that some people get concerned about when drawing robustness diagrams is that their diagram sometimes has a number of Display controllers.

Generally, if a controller is talking to a boundary object (as in Figure 5-6), then it wouldn't violate the diagram's noun-verb-noun rules to leave out the additional Display controller. The fact that the page is going to be displayed is already implied by the arrow from the Get Requested Addresses controller to the Delivery Address Page boundary object.

However, if you **think of Display as being "Initialize page,"** it makes sense to put the Display controller on the diagram wherever needed (see Figure 5-7). In fact, if it helps, call it "Initialize page" instead of "Display." In Figure 5-7, you can see that the system is getting the default delivery address and then initializing the page with the default settings (via the Display controller).

Display initialization code tends to be nontrivial, so it helps to place a Display controller explicitly on the robustness diagram. If you start leaving out the Display controllers, then you're **actually skipping a lot of initialization behavior in the use case**. For the most part, this is not stuff you want to forget about. When you draw a Display controller, ask yourself, "What gets displayed on this screen? Do I have to fetch it from the database?" (etc.)

In fact, when you begin drawing the sequence diagrams from the use case text (see Chapter 8), the Display controller usually becomes an operation on the boundary class, but that's a design decision (allocating operations to classes), and you shouldn't worry about design details while drawing the robustness diagrams.

As you'll see in Chapter 12, you can generate test cases directly from the controllers on your robustness diagrams, so this is another impetus to add that Display controller!

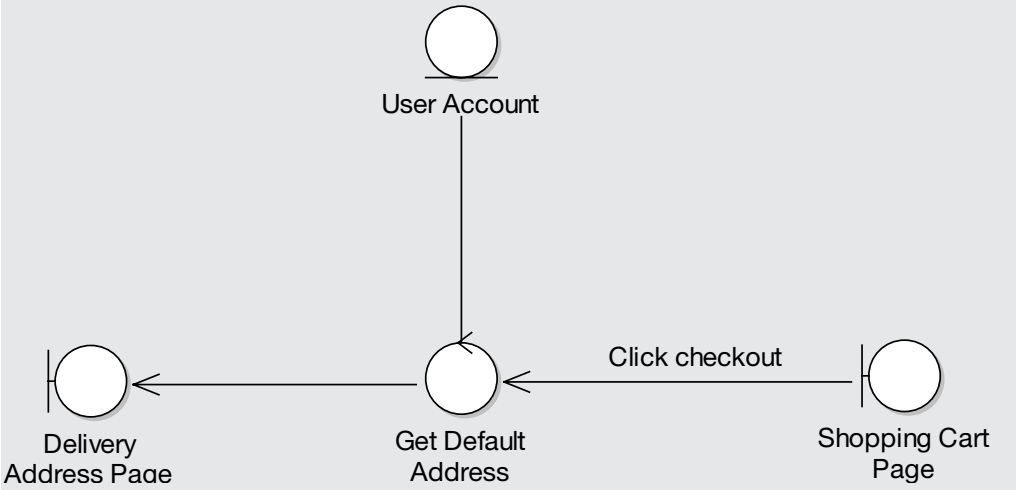


Figure 5-6. *Controller talking directly to the boundary object*

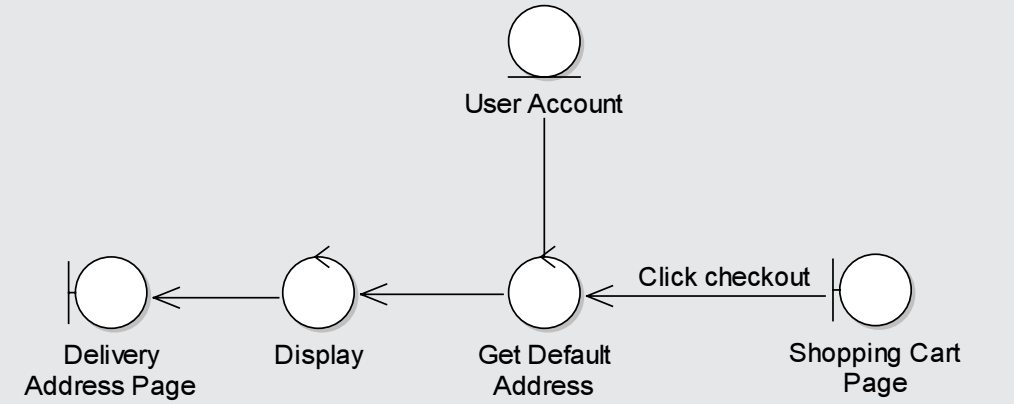


Figure 5-7. *Controller talking to the boundary object via a Display controller*

More About Robustness Diagram Rules

The robustness diagram noun-verb-noun rules may seem unnecessarily restrictive at first, but in reality they help you to prepare your use case text for the much more rigorous (if sometimes implicit) rules that you would need to apply when you create designs from your use cases. The robustness diagram rules are easy to learn, but there's now emerging tools support to catch rule violations. Figures 5-8 and 5-9 show our favorite modeling tool, EA, validating a couple of robustness diagrams for model errors. As far as we know, the folks at Sparx Systems (`www.sparxsystems.com`) are the only ones who have implemented rule checking for this diagram, as of the time of this writing.

In Figure 5-8, all of the possible valid relationships are shown (even though the diagram itself doesn't make a huge amount of sense), and in Figure 5-9, all of the possible *invalid* relationships are shown.

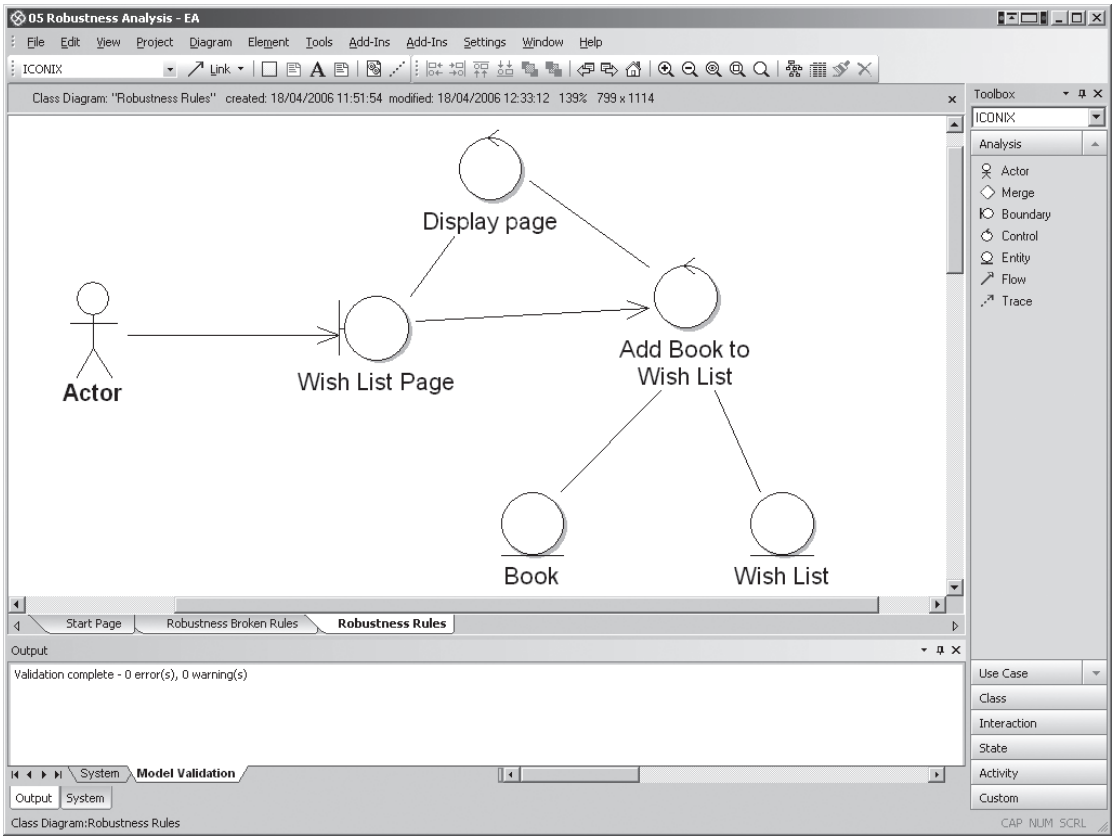


Figure 5-8. *All possible* ***valid*** *robustness diagram relationships*

The relationships shown in Figure 5-8 are allowed because

- An Actor can talk to a Boundary Object.
- Boundary Objects and Controllers can talk to each other (Noun <-> Verb).
- A Controller can talk to another Controller (Verb <-> Verb).
- Controllers and Entity Objects can talk to each other (Verb <-> Noun).

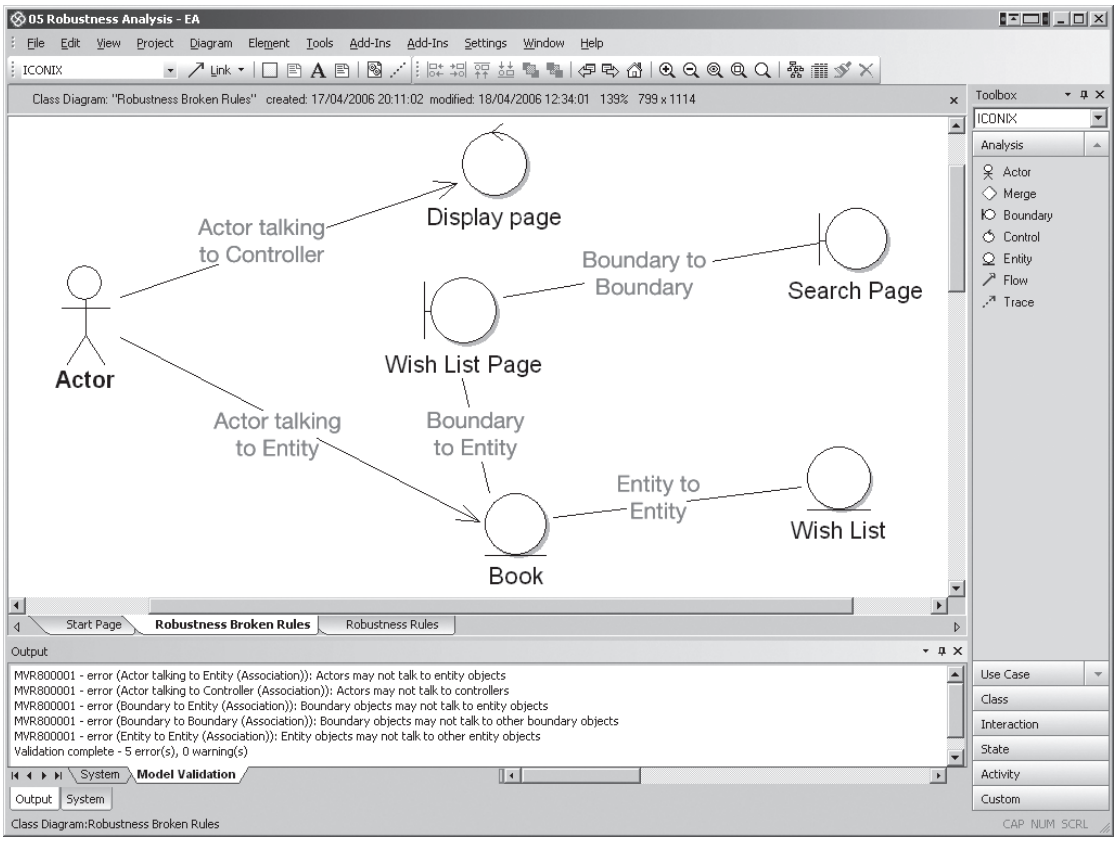


Figure 5-9. *Robustness diagram rule checker in EA showing all possible* ***invalid*** *relationships*

The relationships shown in Figure 5-9 are **not allowed** because

- An Actor can't talk directly to a Controller or an Entity (must talk to a Boundary Object).
- Boundary Objects and Entity Objects can't talk directly to each other (must go via a Controller).
- Entities can't talk directly to other Entities (must go via a Controller).
- Boundary Objects can't talk directly to other Boundary Objects (must go via a Controller).

How Do You Perform Robustness Analysis?

You perform robustness analysis for a use case by **working through the use case text, one sentence at a time**, and drawing the actor(s), the appropriate boundary and entity objects and controllers, and the connections among the various elements of the diagram. You should be able to fit the basic course and all of the alternate courses on one diagram.

Now for a couple of examples (both from the Internet Bookstore). We'll start with a completed robustness diagram, and then we'll walk through another diagram step by step.

Robustness Diagram for the "Show Book Details" Use Case

Figure 5-10 shows a first attempt at the robustness diagram for the *Show Book Details* use case, which we introduced in Chapter 4.

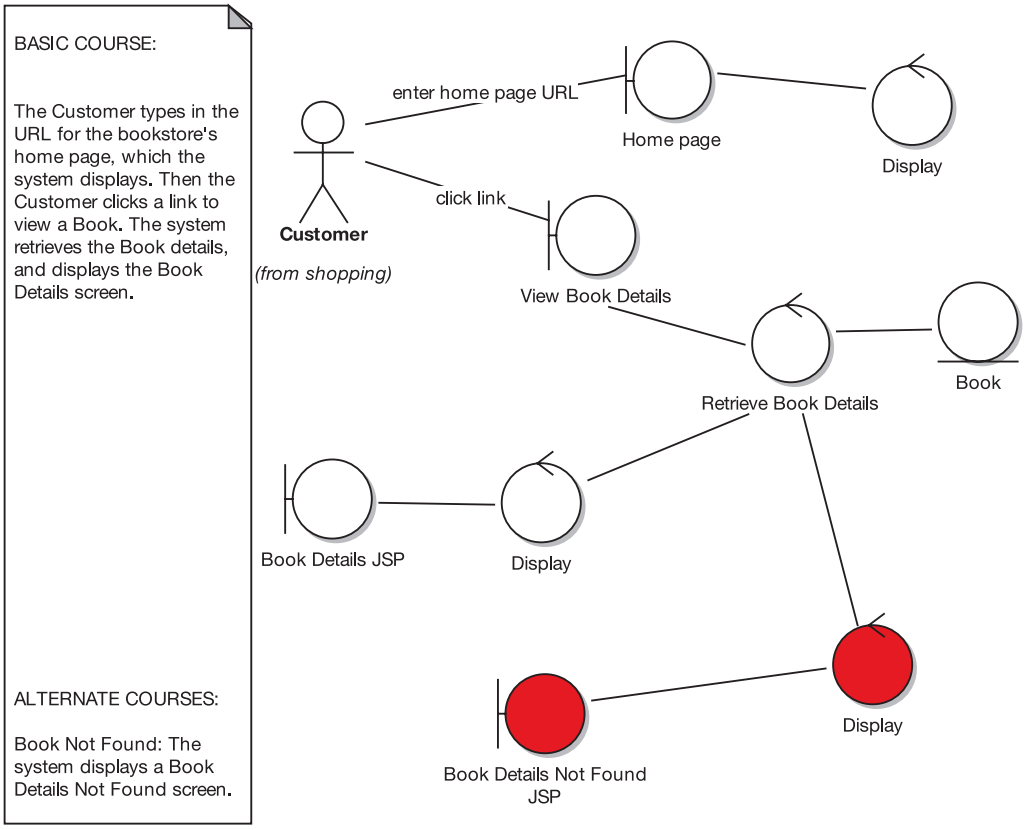


Figure 5-10. *A first attempt at a robustness diagram for the* Show Book Details *use case*

As you can see, it's possible to walk through the robustness diagram by reading the use case text and following along in the diagram itself. If you ever find that there's something in the use case text that isn't in the diagram, then the diagram is incomplete (and vice versa).

There are a few issues with this diagram that are worth clearing up:

- The "click link" should come from the home page boundary object and go to a controller that drives the display of the book details page (it's probably a "book details page" and not a "view book details page"). Always show the user actions coming *off* the boundary object and thus linking the previous screen/page to the next screen/page via a controller.
- Right now there are two boundary classes: "view book details page" and "book details JSP." Are there really two boundaries at the conceptual design level? Or do both of these represent the book details page? Is the "view book details page" boundary actually supposed to be a controller called "display book details page"?
- Don't call things JSPs (or ASPs, or whatever) on the robustness diagrams, as that's too technology specific for conceptual design. It's better to call them screens and pages (this is already done in the use case text).
- Try not to mix nouns and verbs as in "view book details page"—you will confuse yourself! **The name of the page should be a noun.**

Figure 5-11 shows the corrected version.

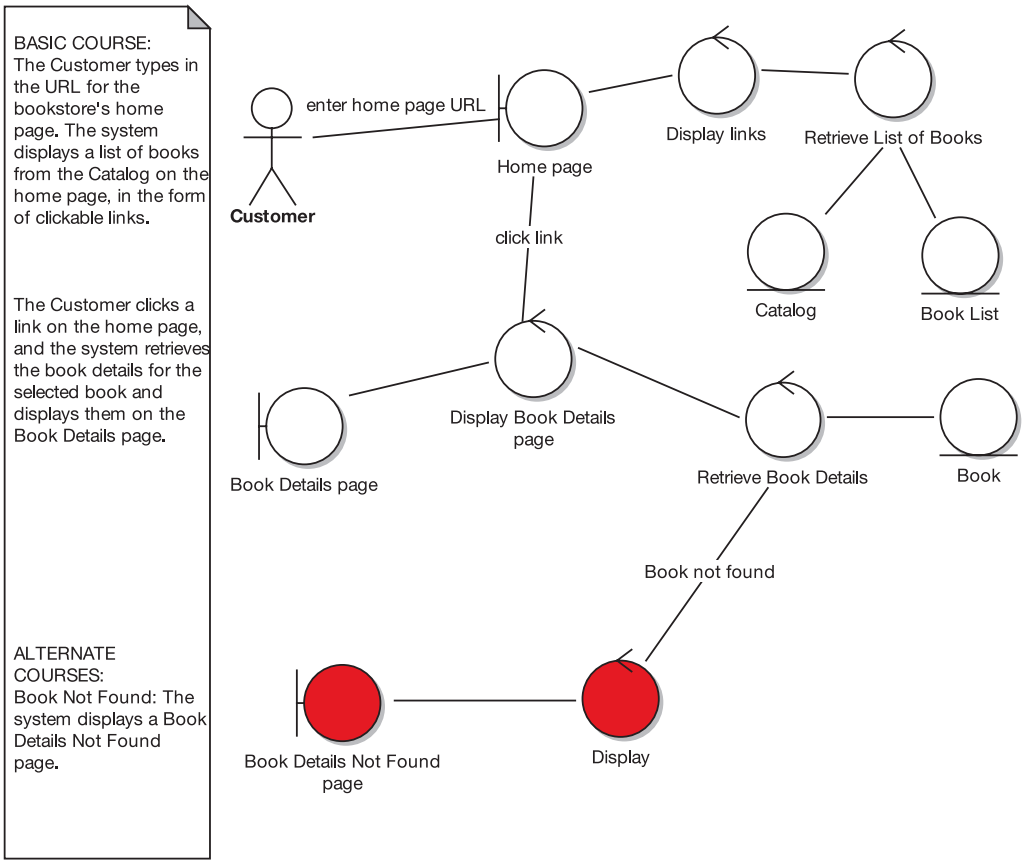


Figure 5-11. *Corrected robustness diagram for the* Show Book Details *use case*

We'll return to the *Show Book Details* use case periodically throughout the book, and we'll take it all the way to source code.

Of course, it's easy for us to simply show you a finished diagram and say, "There, that's how you do it!" So in the next section, we'll walk through the process of drawing a robustness diagram from the beginning.

Robustness Diagram for the "Write Customer Review" Use Case

We'll now walk through the robustness analysis process step by step for the *Write Customer Review* use case.

The first step is shown in Figure 5-12. We create a new, blank robustness diagram.

■ **Tip** Make the robustness diagram a child diagram of the use case you're modeling. (The same goes for the sequence diagram, which you'll add later.) Nest these diagrams "inside" the use case in the project browser.

Next, paste the use case text directly onto the diagram.

The next stage is to read through the first sentence of the basic course in the use case text:

> *The Customer clicks the Write Review button for the book currently being viewed, and the system shows the Write Review screen.*

The first thing referenced is the Customer, so we need to put a Customer actor onto the diagram.

■ **Tip** You can drag the actor directly from the tree view (project browser).

■ **Tip** If you're staring at the screen wondering how to begin . . . well, you aren't alone. When you're learning how to draw robustness diagrams, getting started on a new diagram is usually the trickiest part.

The easiest way to begin is simply to **start at the first sentence of the use case text and draw what you read**. If it just won't translate easily onto the diagram, then it's possible that the use case is starting at the wrong point (e.g., if it describes the actions leading up to the user's first action, then it's probably describing part of a different use case and should be rewritten).

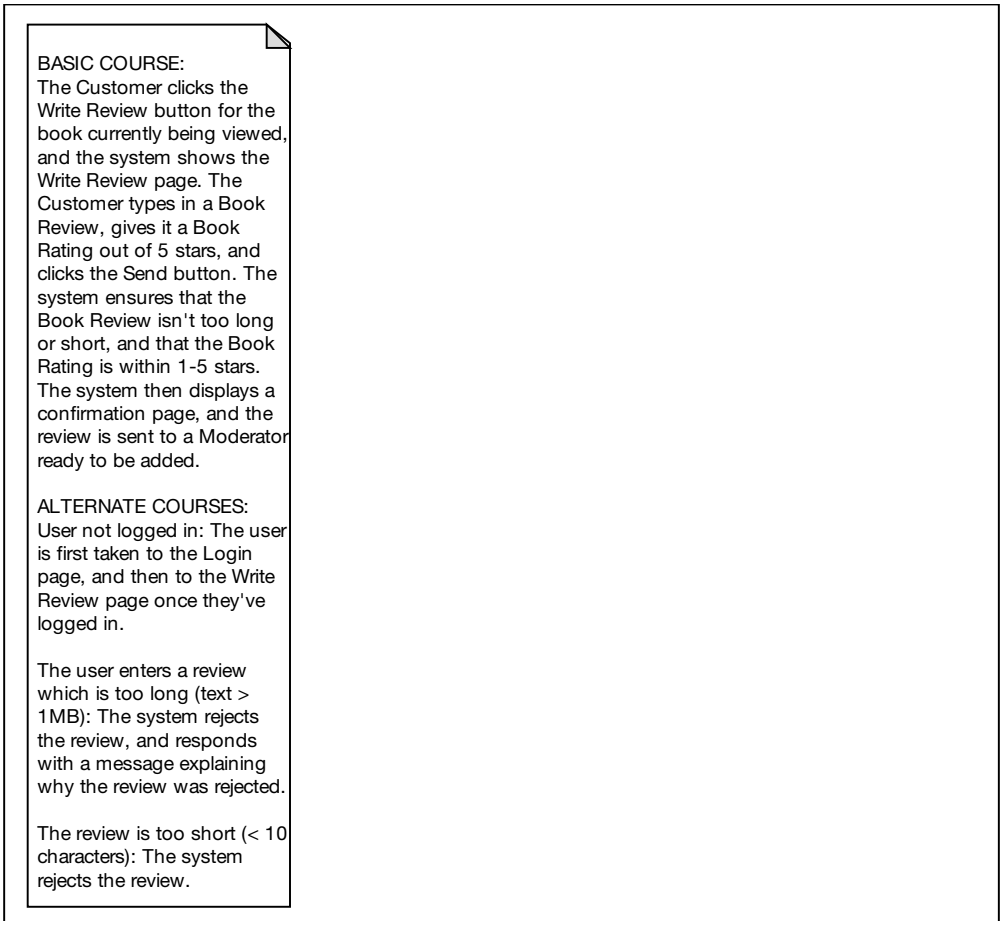


Figure 5-12. *Step 1: Create a new, blank* Write Customer Review *robustness diagram*

Next, it *seems* like the obvious thing to do would be to show the Write Review button as a boundary object and show the Customer interacting with it (see Figure 5-13).

You might wonder whether it's OK to put GUI widgets such as buttons on our robustness diagrams. In practice, we find that doing so opens up Pandora's box. If you include one GUI widget, then you start to think that you should include them all, which means . . . you'll be there all night drawing controllers and boundary objects for all the text fields, list boxes, buttons, labels, and so on for every screen. Yikes. It's better to avoid falling into that trap, and ***avoid drawing individual GUI widgets (below the screen/page/frame level) on the robustness diagrams***.

As you can see in Figure 5-14, we've removed the Write Review Button boundary object and relegated it to a message between the Customer and the Write Review Screen object. If the UI element absolutely must be mentioned explicitly (e.g., if you feel it makes the diagram clearer), then it could be included as a message label, as we've done in Figure 5-15. It isn't essential, though—in fact, the diagram would probably even be slightly clearer without it.

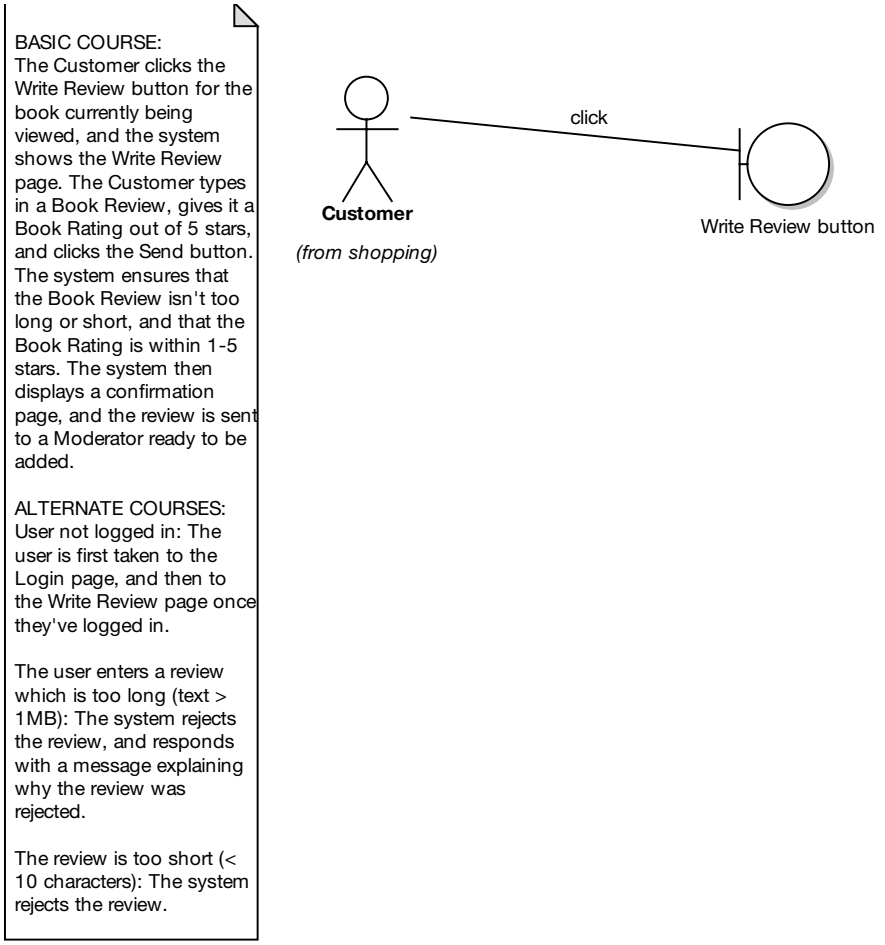


Figure 5-13. *Step 2: Spot the deliberate mistake.*

In Figure 5-14, we've also added a couple of controllers to represent the validation described in the use case text. The text represented in the diagram (so far) is as follows:

> *The Customer types in a Book Review, gives it a Book Rating out of five 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 one and five stars.*

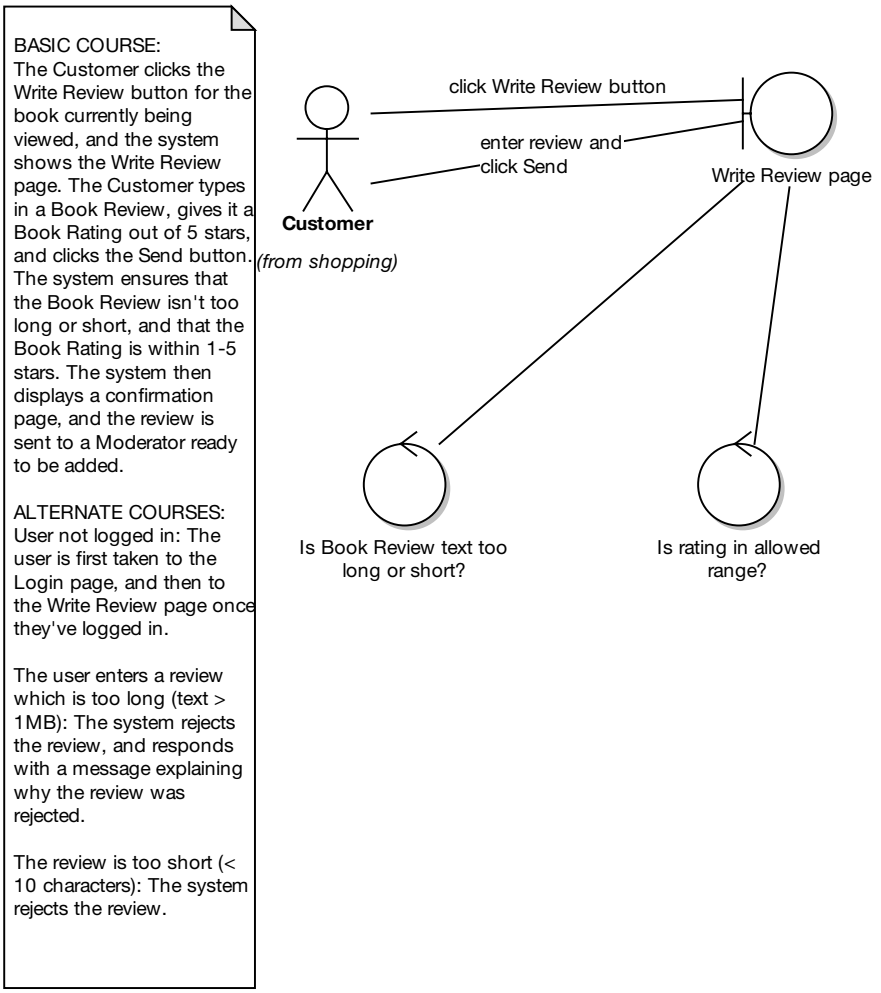


Figure 5-14. *Step 3: We've now corrected the mistake and added some validation controllers.*

There are a few issues with this diagram so far. The two messages between the Customer and the Write Review Screen boundary object are quite clumsy. In addition, this approach introduces some ambiguity into the diagram, as it isn't clear which controller is called when the user clicks the Write Review button versus when the user enters the review and clicks Send. As it turns out, the first sentence of the use case text ("The Customer clicks the Write Review button for the book currently being viewed") hints at a Book Detail screen, which isn't shown either in the text or on the diagram. If we add that screen, then we can have an arrow going from the Customer to the Book Detail screen, and we can also add a Display controller to show the Write Review screen being displayed.

Another issue is that the controller name "Is Book Review text too long or short?" is a tad long, so we could shorten this to "Is Book Review length OK?"