THE EXPERT'S VOICE® IN UML MODELING

Packed with examples and student exercises

Use Case Driven Object Modeling with UML

Theory and Practice

Fast-track your project from use cases to working, maintainable code

Doug Rosenberg and Matt Stephens

Apress®

Use Case Driven Object Modeling with UML

Theory and Practice

Doug Rosenberg and
Matt Stephens

Apress®

Use Case Driven Object Modeling with UML: Theory and Practice

ISBN-13 (pbk): 978-1-59059-774-3

ISBN-10 (pbk): 1-59059-774-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Gennick
Technical Reviewer: Dr. Charles Suscheck
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Senior Project Manager: Tracy Brown Collins
Copy Edit Manager: Nicole Flores
Assistant Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: Nancy Riddiough
Indexer: Toma Mulligan
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

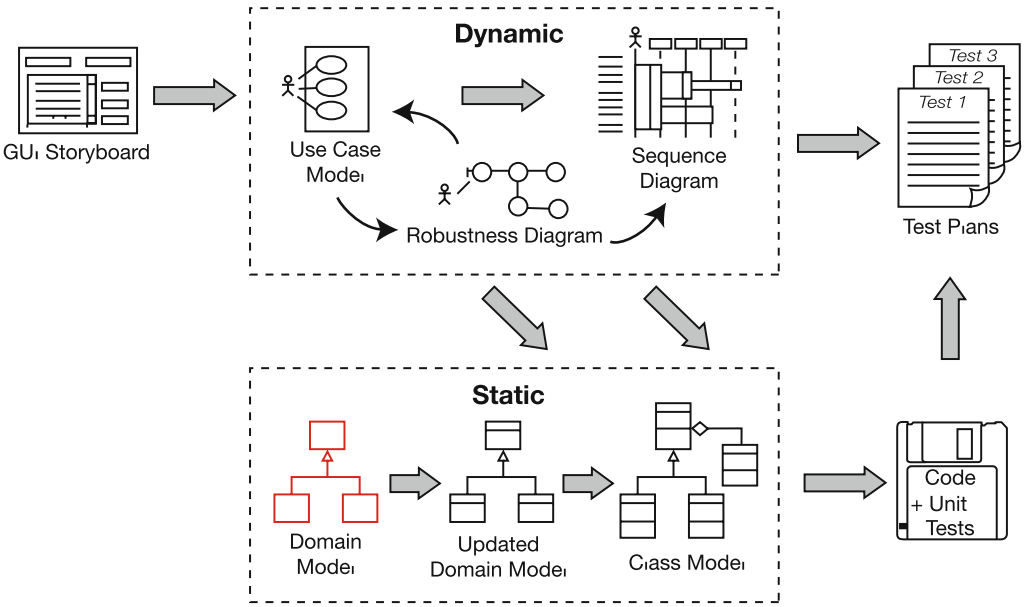
The UML model and source code for the example use cases in this book are available to readers at http://www.apress.com and http://www.iconixprocess.com/InternetBookstore.

PART 1

Requirements Definition

CHAPTER 2

Domain Modeling



Imagine if everyone on your team was talking a different language. Let's say you're speaking German, your teammate is speaking French, and someone else is speaking Swahili. Every time someone speaks, people glean whatever slivers of meaning they can, and then nod as if they've understood perfectly. They then walk away with a completely wrong interpretation of what the speaker was really trying to say.

In virtually all IT projects, the problem of miscommunication is rampant, but it's rarely noticed because everybody *thinks* they're speaking the same language. They're not. One person says "book review" and some people interpret this as meaning "editorial review" (a review written by an editorial team), whereas others might interpret it as meaning "customer review" (a review written by a customer and posted to the site). The results can be—and often are—catastrophic, as the system gets developed with everyone interpreting the requirements and the design differently.

The domain model is a live, collaborative artifact. It is refined and updated throughout the project, so that it always reflects the current understanding of the problem space.

In this chapter we'll look at domain modeling, which aims to solve the problem of miscommunication on projects by establishing a common vocabulary that maps out the problem space.

The 10,000-Foot View

Domain modeling is the task of building a project glossary, or a dictionary of terms used in your project. The domain model for a project defines the scope and forms the foundation on which to build your use cases. A domain model also provides a common vocabulary to enable clear communication between members of a project team. So even though this book is about use case–driven development, we have to begin at the beginning with domain modeling.

What's a Domain Model?

As just mentioned, a domain model is, essentially, a project glossary: a "live" dictionary of all the terms used in your project. But a domain model is better than a project glossary, because it shows graphically how all these different terms relate to each other. In practice it's a simplified class diagram, with lines drawn between the different classes (*domain objects*) to show how they relate to each other. The domain model shows aggregation and generalization relationships (*has-a* and *is-a relationships*) between the domain classes.

Figure 2-1 shows an excerpt from a domain model. Don't worry about the details for now—our purpose in presenting this figure is just so you can visualize what it is we're going to be talking about the rest of the chapter.

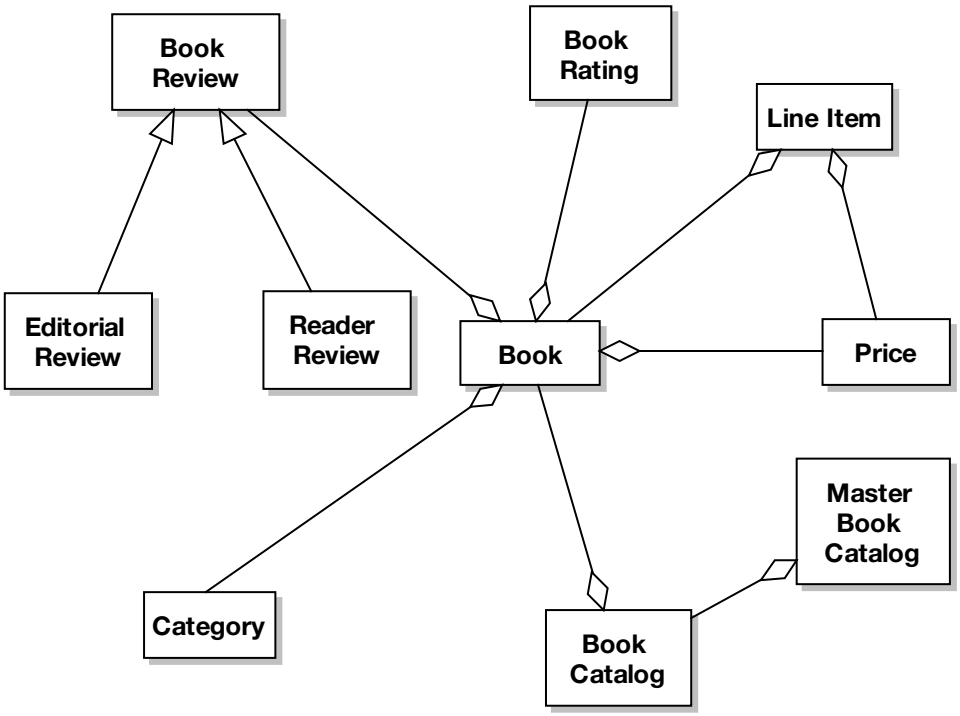


Figure 2-1. *Example of a domain model diagram*

Why Start with the Domain Model Instead of Use Cases?

You'll find that it really helps if you take a quick stab at a domain model right at the start of a project. When you write use cases, it's tempting to make them abstract, high-level, vague, and ambiguous. In fact, some gurus even recommend that you write your use cases this way (only they call it "abstract," "essential," "technology-free," etc.)—but more about that later. Our advice is pretty much the opposite: your use case text should be grounded in reality, and it should be very close to the system that you'll be designing. In other words, the use cases should be written in the context of the object model (i.e., the use case text needs to reference the domain objects by name). By doing this, you'll be able to tie together the static and dynamic parts of the model, which is crucial if you want your analysis and design effort to be driven forward by your use cases.

So before you write your use cases, you need to come up with a first-pass attempt at a domain model. The domain model forms the foundation of the static part of your model, while the use cases are the foundation of the dynamic part. The static part describes structure; the dynamic part describes behavior.

■ **Note** At the analysis level, the terms "object" and "class" are sometimes used interchangeably (an object is a runtime instance of a class). However, when we get to the more grounded design level, the distinction between objects and classes becomes more important.

Domain Modeling in Theory

As you read this book, you'll see that each chapter follows a familiar pattern. We start by describing an aspect of modeling "in theory," using our Internet Bookstore example to illustrate the points we make. Then we cover it "in practice," showing typical modeling errors and how to correct them, and presenting a number of exercises. Finally, we round off each chapter with "more practice."

Top 10 Domain Modeling Guidelines

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

10. Focus on real-world (problem domain) objects.
9. Use generalization (is-a) and aggregation (has-a) relationships to show how the objects relate to each other.
8. Limit your initial domain modeling efforts to a couple of hours.
7. Organize your classes around key abstractions in the problem domain.
6. Don't mistake your domain model for a data model.
5. Don't confuse an object (which represents a single instance) with a database table (which contains a collection of things).
4. Use the domain model as a project glossary.
3. Do your initial domain model before you write your use cases, to avoid name ambiguity.
2. Don't expect your final class diagrams to precisely match your domain model, but there should be some resemblance between them.
1. Don't put screens and other GUI-specific classes on your domain model.

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

10. Focus on Real-World Objects

When creating a domain model, be sure to focus on real-world objects within the problem domain. Try to organize your software architecture around what the real world looks like. The real world tends to change less frequently than software requirements.

CLASS NOTATION

Figure 2-2 shows two different types of class notation. On a full-blown detailed class diagram, you'd use the version on the left, with attributes and operations. However, during the initial domain modeling effort, it's too early to allocate these parts of a class. It's better to use the simpler notation shown on the right. This version only shows the domain class's name.

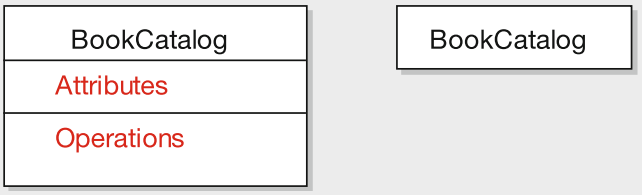


Figure 2-2. *Class notations*

9. Use Generalization (Is-a) and Aggregation (Has-a) Relationships

Over time, you'll flesh out your domain model with new domain classes, as and when you identify them. You'll also notice relationships (or *associations*) between them—for example, a `Book Review` belongs to a `Book`, and a `Purchase Order` and `Credit Card` are two of a kind, as they're both `Payment Types`.

The first relationship (*Book Review belongs to a Book*) is called aggregation (has-a, because a `Book` has a `Book Review`). The second relationship (*Purchase Order and Credit Card are both Payment Types*) is called generalization (is-a, because a `Purchase Order` is a `Payment Type`). Figure 2-3 shows an illustration of these concepts.

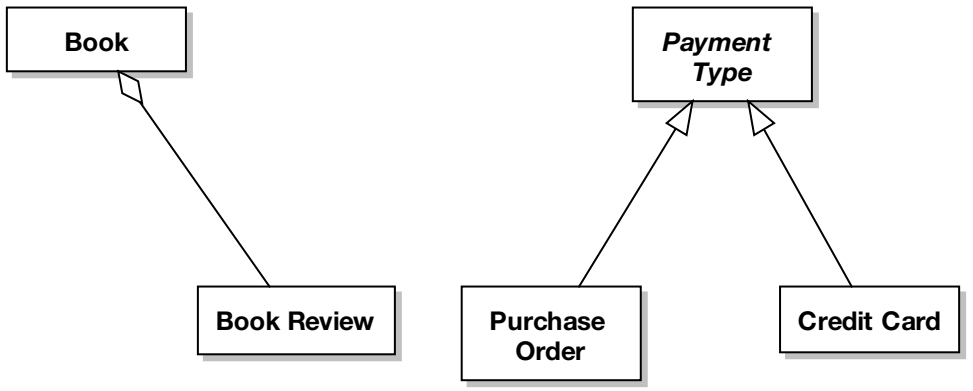


Figure 2-3. *Aggregation and generalization relationships*

These and regular (plain vanilla) associations are the most important relationships in your domain model. Ninety-five percent of your model's class relationships can be modeled using aggregation and generalization relationships.

■ **Tip** Wherever possible, place your associations so that they read left to right and top to bottom, just like regular text. This will improve the readability of your diagrams.

---

8. Limit Your Initial Domain Modeling Efforts to a Couple of Hours

We recommend that you establish a time budget for building your initial domain model. A couple of hours is all you should need. You're not going to make it perfect anyway, so do it quickly and expect to fix it as you proceed. You should be vigilant about making necessary adjustments to your analysis-level class model in response to discoveries made during robustness analysis and throughout the project.

You'll discover missing objects as you work through use cases and robustness diagrams. The use case–driven process *assumes that the domain model is incomplete* and provides a mechanism for discovering what was missed.

The initial domain modeling session is probably the most important two hours you'll spend on the project! It's likely that you'll discover 80% of your domain classes during that two-hour brainstorming session. If you can get 80% of your domain vocabulary disambiguated, then that's two hours well spent.

7. Organize Your Classes Around Key Abstractions in the Problem Domain

It's generally good practice to organize your classes around key abstractions in the problem domain. Remember that the domain model is a first-cut class diagram that becomes the foundation of your software architecture. This makes the model more resilient in the face of change. Organizing the architecture around real-world abstractions makes the model more resilient in the face of changing requirements, as the requirements will usually change more frequently than the real world does.

6. Don't Mistake Your Domain Model for a Data Model

Even though the diagrams might look similar, remember that what's good practice on a data model is not likely to be good practice on a class diagram (and vice versa). Classes are small and tables are bigger. A table in a relational database often relates a number of things. Conversely, classes are better designed if they're relatively small packets of data and behavior.

In a class diagram, it's likely that you'll have a class that manages a database table, and you might show some sort of `TableManager` class aggregating a regular domain class. The purpose of these `TableManager`-type classes is to hide the details of the database management system (DBMS) from the rest of the code base.

5. Don't Confuse an Object with a Database Table

An object represents a single instance of something. A database table represents a collection of things. You don't have to be as literal-minded as in the Enterprise JavaBeans (EJB) world, where an entity bean generally represents a single row in a table. Domain classes are similar, though. If you call a domain class `Book`, then you don't mean a book table—you mean a single book.

Columns in a table generally map to attributes on a class. However, database tables typically contain a lot more columns than a class contains attributes (tables often have foreign keys, as one example), so there may not be a direct 1:1 mapping between table rows and objects.

4. Use the Domain Model As a Project Glossary

If ambiguous requirements are the enemy, the domain model is the first line of defense. Ambiguous usage of names by "subject matter experts" is very common and very harmful. The domain model should serve as a project glossary that helps to ensure consistent usage of terms when describing the problem space.

Using the domain model as a project glossary is the first step toward disambiguating your model. In every Jumpstart workshop that Doug teaches, he finds at least two or three domain classes where students are using ambiguous names (e.g., "shopping cart," "shopping basket," or "shopping trolley").

3. Do Your Domain Model Before You Write Your Use Cases

Since you're using the domain model to disambiguate your problem domain abstractions, it would be silly to have your use cases written using ambiguous terms to describe domain classes. So spend that two hours working on the domain model before writing your use cases. Writing the use cases without a domain model to bind everything together stores up lots of problems for later.

2. Don't Expect Your Final Class Diagrams to Precisely Match Your Domain Model

The class diagrams will become a lot more detailed than the domain model as the design progresses; the domain model is deliberately kept quite simple. As you're designing (using sequence diagrams), detailed design constructs such as GUI helpers, factory classes, and infrastructure classes get added to the class diagram, and the domain model diagram will almost certainly be split out into several detailed class diagrams. However, it should still be possible to trace most classes back to their equivalent domain class.

1. Don't Put Screens and Other GUI-Specific Classes on Your Domain Model

Doing so opens up Pandora's box and leads to an overcrowded domain model containing lots of implementation-specific detail. Performance optimization classes, helper classes, and so on should also be kept out of the domain model. The domain model should focus purely on the problem domain.

Internet Bookstore: Extracting the First-Pass Domain Model from High-Level Requirements

When you're creating your domain model, a good source of domain classes includes the high-level requirements—the ones that are usually (but not always) written in the form "The system *shall* do this; the system *shall not* do that." It's useful to scan these requirements, extracting the nouns and noun phrases. You can then refine these to create the initial domain model.

With that in mind, let's go through the high-level requirements for the Internet Bookstore and extract some **domain classes** from them.

1. The **bookstore** will be web based initially, but it must have a sufficiently flexible architecture that alternative front-ends may be developed (Swing/applets, web services, etc.).
2. The bookstore must be able to sell **books**, with **orders** accepted over the **Internet**.
3. The user must be able to add books into an online **shopping cart**, prior to **checkout**.
    a. Similarly, the user must be able to remove **items** from the shopping cart.
4. The user must be able to maintain **wish lists** of books that he or she wants to purchase later.
5. The user must be able to cancel orders before they've shipped.
6. The user must be able to pay by **credit card** or **purchase order**.
7. It must be possible for the user to return books.
8. The bookstore must be embeddable into **associate partners**' websites using **mini-catalogs**, which are derived from an overall **master catalog** stored in a central **database**.
    a. The mini-catalogs must be defined in XML, as they will be transferred between this and (later to be defined) external systems.
    b. The **shipping fulfillment system** shall be carried out via Amazon Web Services.
9. The user must be able to create a **customer account**, so that the system remembers the user's details (name, address, credit card details) at login.
    a. The system shall maintain a **list of accounts** in its central database.
    b. When a user logs in, his or her **password** must always be matched against the passwords in the **master account list**.
10. The user must be able to search for books by various **search methods**—**title**, **author**, **keyword**, or **category**—and then view the **books' details**.
11. It must be possible for the user to post reviews of favorite books; the **review comments** should appear on the book details screen. The review should include a **customer rating** (1–5), which is usually shown along with the book title in **book lists**.

   a. **Book reviews** must be moderated—that is, checked and "OK'd" by a member of staff before they're published on the website.

   b. Longer reviews should be truncated on the book details screen; the **customer** may click to view the full review on a separate page.

12. It must be possible for staff to post **editorial reviews** of books. These should also appear on the book details screen.

13. The bookstore shall allow third-party **sellers** (e.g., second-hand bookstores) to add their own individual **book catalogs**. These are added into the overall **master book catalog** so that sellers' books are included in search results.

14. The bookstore must be scalable, with the following specific requirements:

   a. The bookstore must be capable of maintaining **user accounts** for up to 100,000 customers in its first six months, and then a further 1,000,000 after that.

   b. The bookstore must be capable of serving up to 1,000 simultaneous users (10,000 after six months).

   c. The bookstore must be able to accommodate up to 100 search requests per minute (1,000/minute after six months).

   d. The bookstore must be able to accommodate up to 100 purchases per hour (1,000/hour after six months).

These requirements are a rich source of domain classes. Let's put all the highlighted nouns and noun phrases into a list (in the process, we'll turn all the plurals into singulars, and put them all in alphabetical order):

Associate Partner
Author
Book
Book Catalog
Book Details
Book List
Book Review
Bookstore
Category
Checkout
Credit Card
Customer
Customer Account
Customer Rating
Database
Editorial Review
Internet
Item
Keyword
List of Accounts
Master Account List
Master Book Catalog
Master Catalog
Mini-Catalog
Order
Password
Purchase Order
Review Comment
Search Method
Search Results
Seller
Shipping Fulfillment System
Shopping Cart
Title
User Account
Wish List

There's quite a bit of duplication in this list; similar terms are being used for basically the same thing. But that's really the main benefit of the domain modeling approach: you get to identify and eliminate these duplicate terms early on in the project.

■ **Exercise** **Disambiguation via Grammatical Inspection**: We'll go through this list next, whipping it into shape and eliminating the duplicate terms. But first, try to identify the six duplicate pairs in the list. (Be careful: one pair *seems* like a duplicate but really isn't.)

Some of the items in the list are simply unnecessary because they fall outside the scope of the domain model, or they're actions sneakily masquerading as nouns.

Let's step through the list now and tune it up a bit:

- You'd *think* that the terms "Customer" and "Customer Account" are duplicates, but in fact they represent subtly different things: "Customer Account" is an entity stored in the database, whereas "Customer" is an actor (see the next item in this list).
- "Customer" and "Seller" are actors, and thus should be placed on use case diagrams. (See Chapter 3.)
- The terms "User Account" and "Customer Account" are duplicates. The choice of which one to keep is fairly arbitrary, so we'll go with "Customer Account."
- The terms "List of Accounts" and "Master Account List" are duplicates, so one of them should be removed. As we also have a "Master Book Catalog," the consistent thing would be to keep "Master Account List."
- The terms "Book Review" and "Review Comment" are duplicates, so we'll keep "Book Review."
- We have several different candidate terms for a catalog, or list of books: "Book Catalog," "Book List," "Mini-Catalog," and "Master Catalog." Catalogs and lists are probably different concepts. In fact, it seems that the requirements are trying to tell us something, which may just be implied in the text. When in doubt, *talk to the customer*. Ask questions until you get a clear, unambiguous answer.

  "Book Catalog" and "Master Catalog" are in fact the duplicates here, so we'll keep "Master Catalog," as it provides a good contrast with "Mini-Catalog." "Book List," meanwhile, is probably an umbrella term for different types of lists; we'll keep it in there for now and see how it fits in when we draw the domain model diagram.
- There's another duplicate in this area: "Master Catalog" and "Master Book Catalog." We'll delete "Master Catalog," as "Master Book Catalog" is the more descriptive term.
- The word "Internet" is too generic and doesn't add anything here.
- The word "Password" is a too small to be an object and would be shown as a UI element, so we should remove it from the domain model. If we start to include all the UI elements in the domain model, we're opening a serious can of worms and could be here all night backed into a corner, fighting them away with a large stick.

- Same goes for "Title" and "Keyword."
- Yet another duplicate is "Book" and "Book Details." We'll just keep "Book," as it's more concise than "Book Details," without losing any meaning.
- The word "Item" is just vague and fuzzy, but it does represent a valid concept: an item that's been added to the user's shopping cart. So we'll rename it "Line Item" and keep it in the list.
- The word "Bookstore" is a bit too broad and is unlikely to be referred to explicitly, so we can get rid of it.

Following is the updated list of candidate domain classes. Figure 2-4 shows those classes laid out in a class diagram.

Associate Partner	Customer Account	Order
Author	Customer Rating	Purchase Order
Book	Database	Search Method
Book List	Editorial Review	Search Results
Book Review	Line Item	Shipping Fulfillment System
Category	Master Account List	Shopping Cart
Checkout	Master Book Catalog	Wish List
Credit Card	Mini-Catalog	

As we mentioned earlier, although grammatical inspection techniques are useful to get a quick start, you shouldn't spend weeks or even days doing this. (As you'll see in Chapter 4, the rest of the objects for the Internet Bookstore were identified during robustness analysis.) A couple of hours is about the right amount of time to spend on the domain model before getting started writing the use cases.

■ **Caution** Don't get bogged down in grammatical inspection.

Figure 2-4 shows one type of relationship, *aggregation* (aka has-a), which we described earlier. As this is a first-pass attempt, not all the relationships shown are correct.

A helpful technique is to read the diagram aloud and include the term "has-a." For example, a Shopping Cart "has" Line Items. But, does an Order "have" Checkouts? Perhaps not. Notice that a few of the domain objects currently don't match up with anything else (namely, Associate Partner, Shipping Fulfillment System, Database, and Search Method). We've grouped these together over on the right for now; during robustness analysis, these may get linked to other objects, warped into something different, or removed altogether.

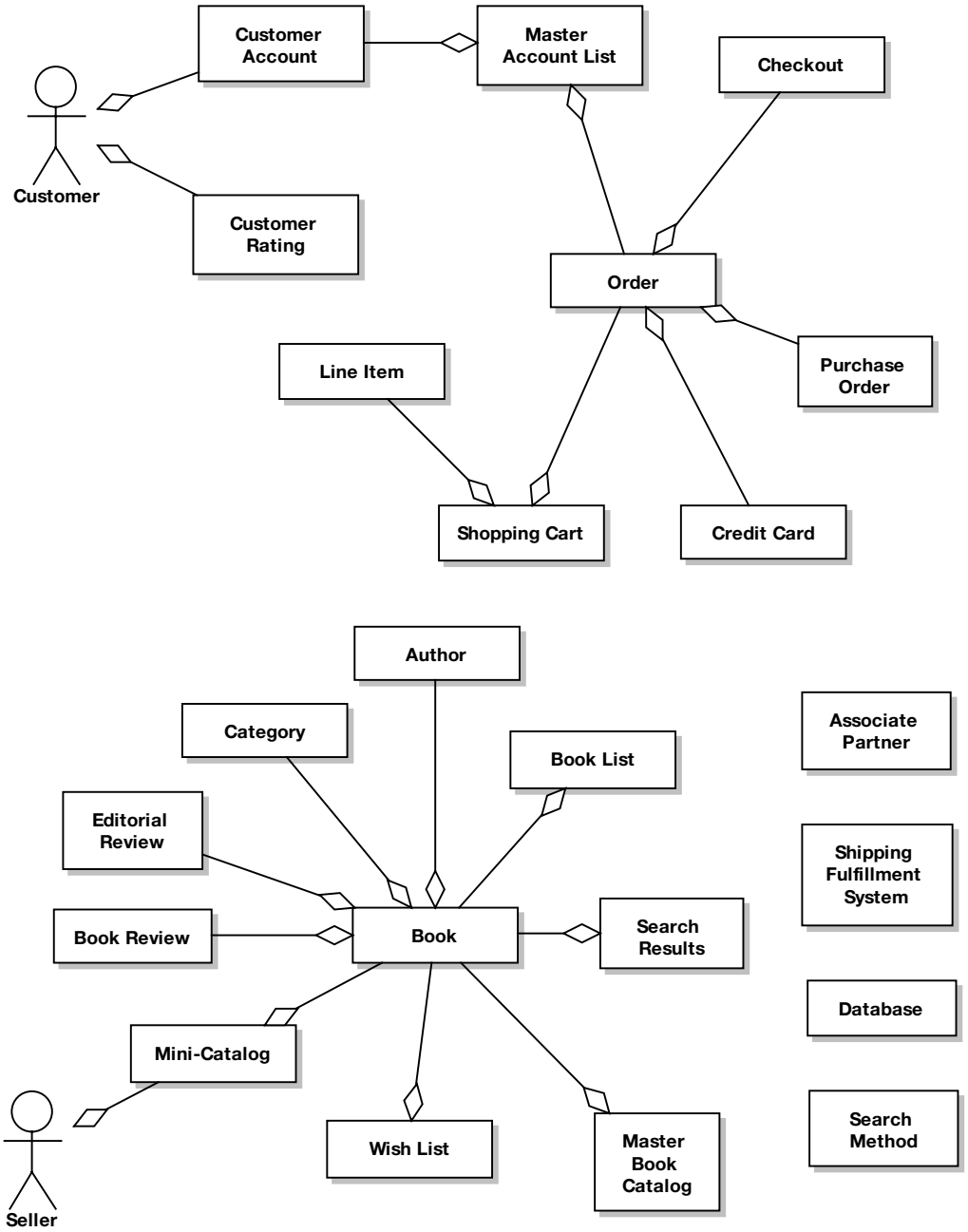


Figure 2-4. *First-pass domain model for the Internet Bookstore project*

There's still some work that needs to be done on this domain model before we're ready to move on to the next stage, so let's do some more tidying up work next. Hopefully, this will help to illustrate an important element of the ICONIX approach: ***continuous improvement via ongoing iteration and refinement***.

Internet Bookstore: Second Attempt at the Domain Model

When drawing up the domain model diagram, you're generally brainstorming as a team. Often the team will identify further domain objects that weren't in the requirements, but instead have been dredged from somebody's own understanding of the problem domain. To illustrate this, let's say we've discovered two additional domain objects: `Order History` and `Order Dispatch`. These weren't mentioned explicitly in the requirements, but they could still classify as minimum requirements for an Internet bookstore.

The updated diagram is shown in Figure 2-5, with the new domain classes shown in red.

In Figure 2-5, we've explored the concept of order fulfillment and dispatch. `Shipping Fulfillment System` still remains on the diagram, but we'll have to decide whether this is in scope for the current model or it's an external system that we need to interface with. **External systems are always modeled as actors**.

We've also removed `Checkout`, as on reflection this was really a verb in noun's clothing. And we've removed `Author`, as this is really just another field in the `Book` (i.e., it's too small to be a first-class object on the domain model[1]). Authors . . . who needs 'em?

There's some ambiguity around `Master Book Catalog`, which we've attempted to resolve. We've removed the link between `Book` and `Master Book Catalog`, and instead added a class called `Book Catalog` and linked `Book` to that instead. So we end up with a **tangle of relationships**: we're effectively saying that a `Book` belongs to a `Book Catalog`, and a `Book Catalog` belongs to a `Master Book Catalog` (i.e., a `Master Book Catalog` is really a catalog of catalogs). Ideally, a `Mini-Catalog` should also belong to the `Master Book Catalog`. But this tangle is getting complicated. What we really need is a simple way of saying that a `Book` can belong to `Book Catalogs`, and there can be various *types* of `Book Catalogs`. Luckily, a light sprinkling of ***generalization*** can work wonders on such relationship tangles, as you'll see in the next section.

1. Remember we're not creating a data model here. If this were a database design, we would almost certainly create a separate Authors table.

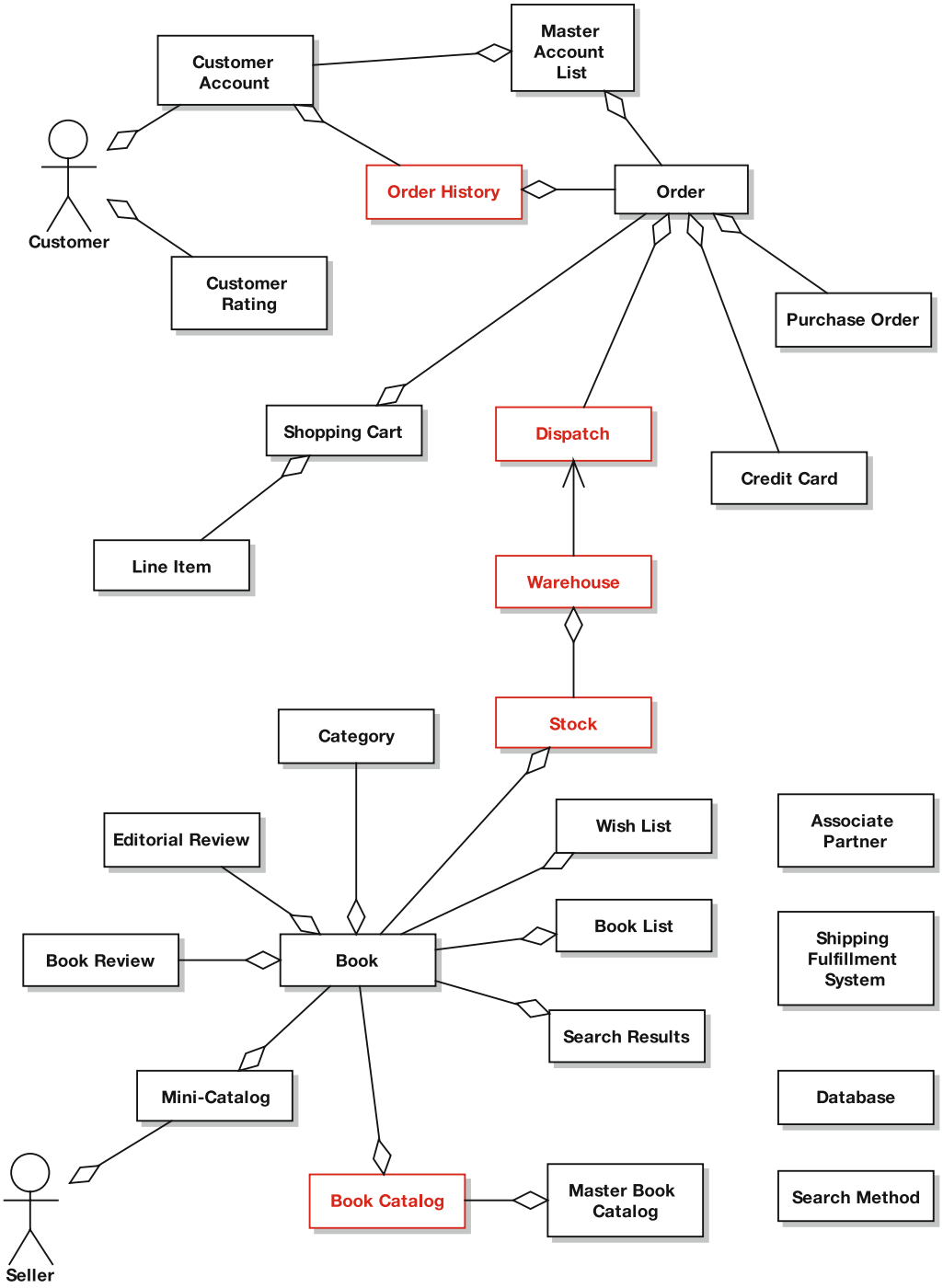


Figure 2-5. *Second snapshot of the evolving domain model for the Internet Bookstore project*

Internet Bookstore: Building Generalization Relationships

A *generalization relationship* is one in which one class is a "kind of" some other class—for example, a `Cat` is a kind of `Animal`. This is why generalization is often called an *is-a* relationship.

`Cat` is more specific than `Animal` (`Cat` is a "refinement" of the more general `Animal` class), hence the term "generalization." The more specific class is called the *subclass*, and the more general class is the *superclass*. Creating subclasses of more general classes is known as *subtyping*.

Within the Internet Bookstore, `Book Catalog` is a good candidate for subtyping, because doing so will help to "de-cloud" the relationship between `Mini-Catalog` and `Master Book Catalog`. `Book List` is also a good candidate for subtyping, because there may well be different types of accounts and different types of book lists.

As we delve more deeply into the user's needs for the Internet Bookstore system, we're beginning to identify different types of book lists: customer wish lists, recommendation lists, Related Books, Search Results, and so on. It's becoming clear that these are all simply lists of `Books`, so they could (conceptually, at least) have a common parent class. We've discovered that there are indeed aspects of `Wish Lists`, `Related Books`, and so on that are different enough to justify separate treatment, while they still have enough in common that they're all kinds of `Book List`. Figure 2-6 shows the notation for this generalization structure.

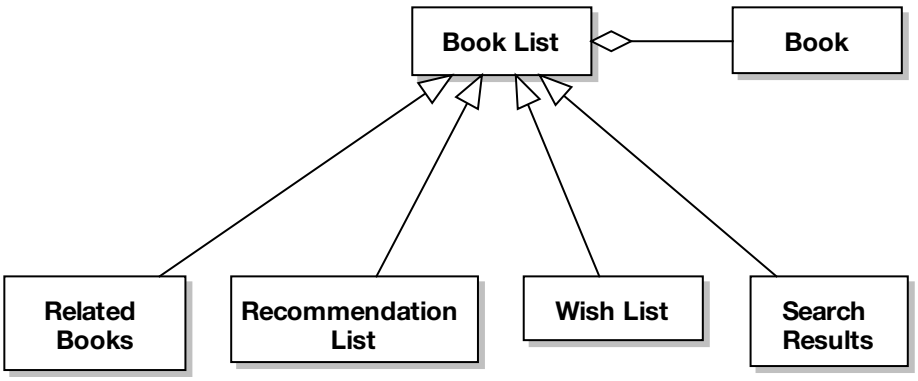


Figure 2-6. *Book Lists detail from the Internet Bookstore domain model*

The new classes (`Related Books`, `Recommendation List`, `Wish List`, and `Search Results`) inherit the attributes and operations that we define for `Book List`. Let's read this diagram out loud: A book list has books. Related Books is a book list. Recommendation List is a book list. Wish List is a book list. Search Results is a book list. All true statements that describe the problem space? Great, let's move on.

■ **Tip** You could also add additional specialized attributes and operations for each of the new classes. In other words, if you were to add an operation to `Related Books`, it would only be available to `Related Books`. However, if you add it to `Book List`, the new operation would be available to all of its subclasses.

Figure 2-7 shows the updated Internet Bookstore domain model, which makes good use of generalization to clarify the relationships between the domain classes. The new classes are shown in red.

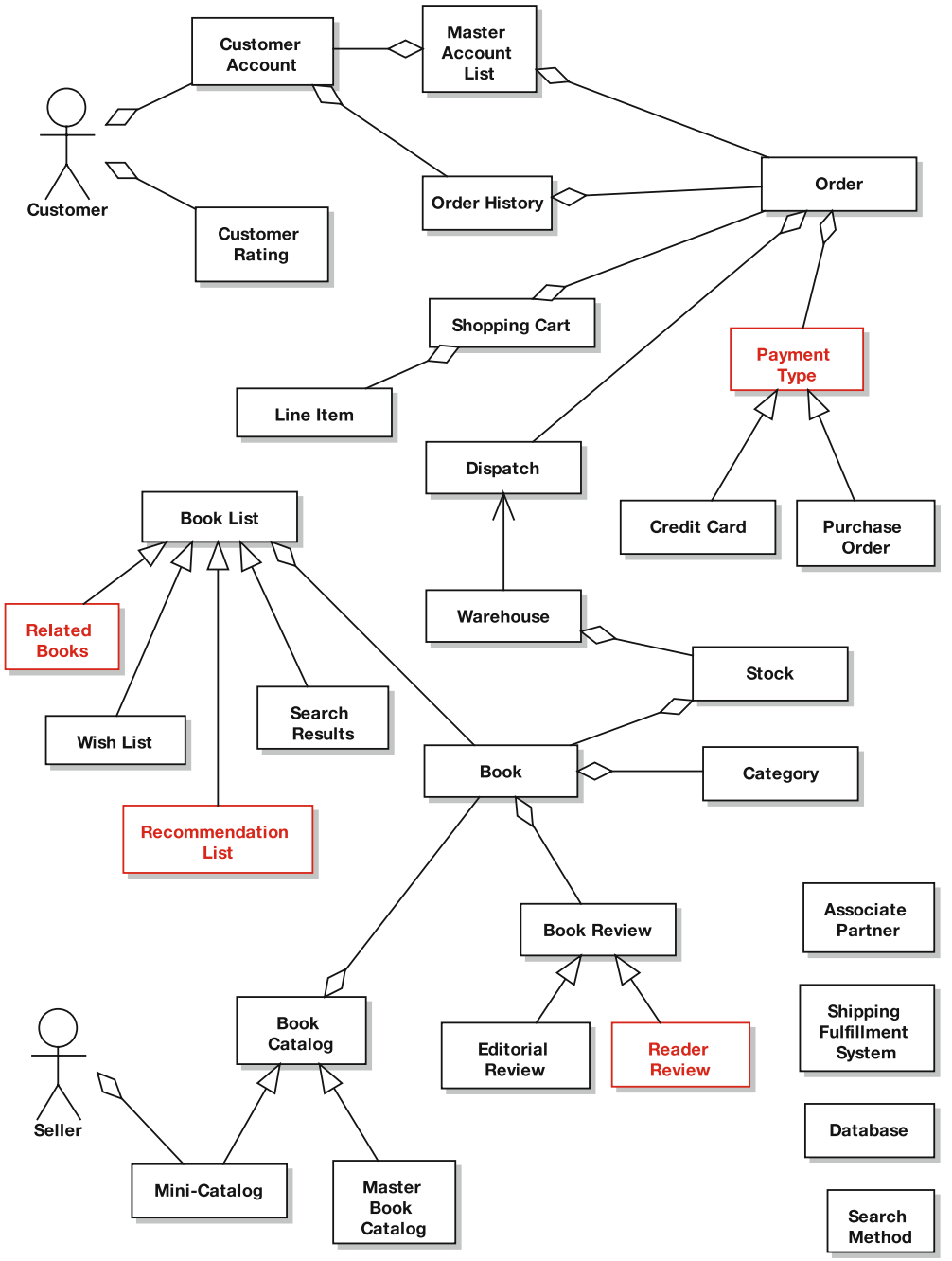


Figure 2-7. *Third snapshot of the evolving domain model for the Internet Bookstore project*

We've also changed the definition of `Book Review`, so that it's now the parent class for `Editorial Review` and the new class, `Reader Review`. And, finally, we've disentangled the relationships surrounding `Order` and its payment types (`Credit Card` and `Purchase Order`), by adding a new superclass, `Payment Type`.

■Tip If you need to, you can generalize to more than one level of subclass. Remember to look for is-a statements that are true in the real world.

Domain Modeling in Practice

The following exercises, taken from the domain model for the Internet Bookstore, are designed to test your ability to spot the most common mistakes that people make during domain modeling. After the exercises, you can find the diagrams with the errors highlighted on them, followed by the corrected diagrams.

Exercises

Each of the diagrams in Figures 2-8 to 2-11 contains one or more typical modeling errors. For each one, try to figure out the errors and then draw the corrected diagram. The answers are in the next section.

Exercise 2-1

Figure 2-8 shows a class diagram produced during the initial domain modeling effort. The UML syntax is correct, yet the diagram does point out a process-related error. Why is that? (Hint: The diagram is showing too much detail.)

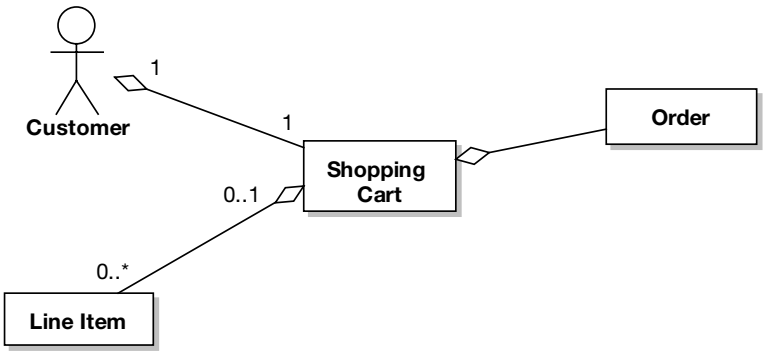


Figure 2-8. *Class diagram from the initial domain modeling effort*

Exercise 2-2

Figure 2-9 shows a domain model diagram with attributes on the Order class. What database-related problem does the diagram suggest?

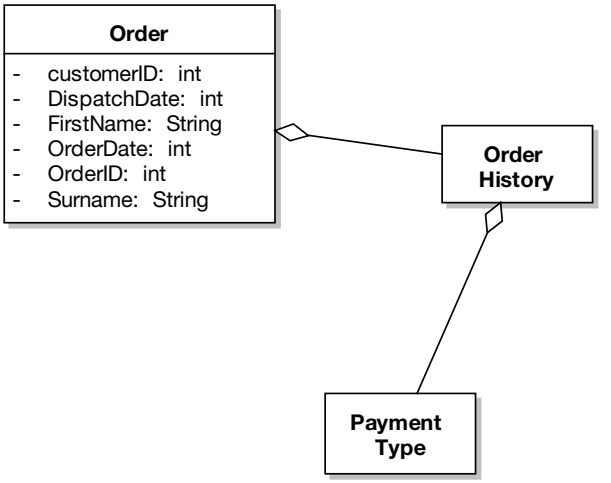


Figure 2-9. *Class diagram showing attributes*

Exercise 2-3

Figure 2-10 shows a domain model diagram in which the modeling team may have leapt ahead a little too soon. Which parts of the diagram were added too early in the process?

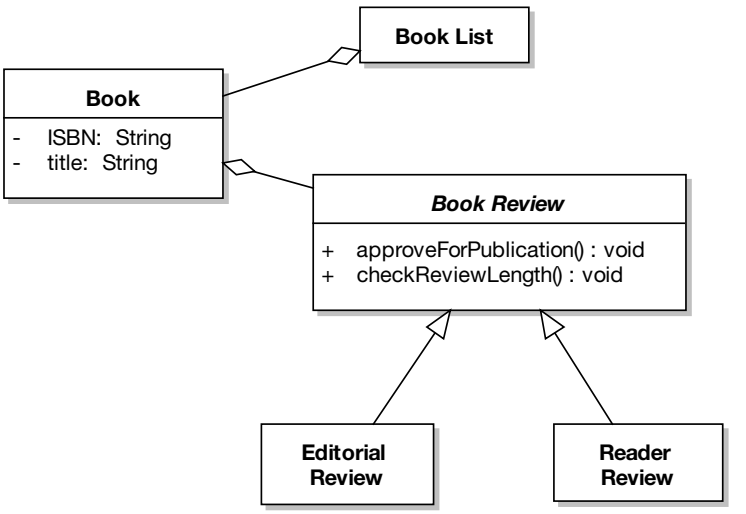


Figure 2-10. *Domain model diagram with details added too early*

Exercise 2-4

Figure 2-11 shows another domain model diagram in which the modeling team began thinking about certain details too early. What's gone wrong in this diagram?

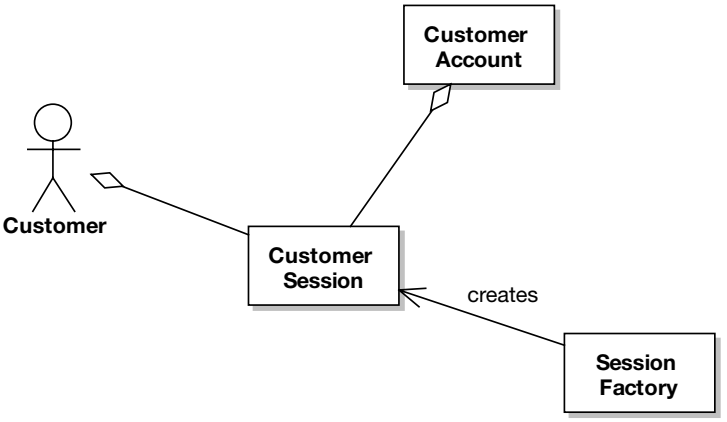


Figure 2-11. *Another domain model diagram with details added too early*

Exercise Solutions

Following are the solutions to the exercises.

Exercise 2-1 Solution: Multiplicity

Figure 2-12 highlights the errors in Figure 2-9. The initial domain modeling effort is way too early to start thinking about details like multiplicity. At this early stage, your main concern should be identifying domain objects and thinking at a broad level about how they relate to one another. Figure 2-13 shows the corrected diagram.

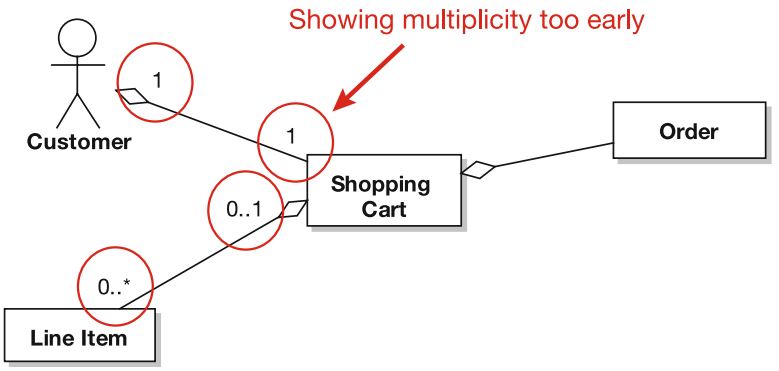


Figure 2-12. *Errors in Figure 2-9*

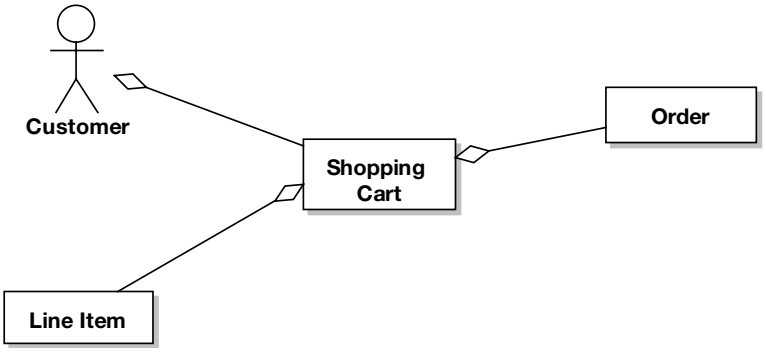


Figure 2-13. *Corrected version of Figure 2-9*

Exercise 2-2 Solution: Mapping Database Tables to Domain Classes

The `Order` domain class includes attributes that really don't seem like they belong in an `Order` class (see Figure 2-14). The most likely cause is that the modeler has mapped these domain classes directly from a relational database schema. Figure 2-15 gives the corrected diagram. The extra attributes have been separated out into their own domain classes (`Customer Account` and `Dispatch`). Note that we'd generally not show the attributes at all during domain modeling.

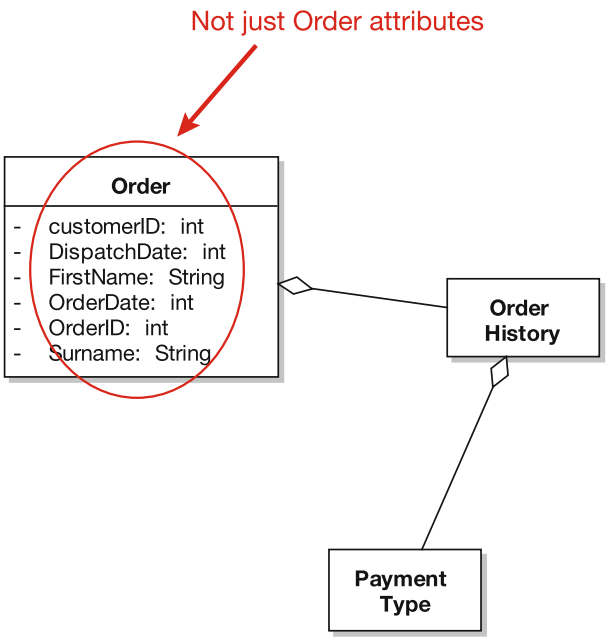


Figure 2-14. *Order class domain attributes erroneously influenced from database schema*

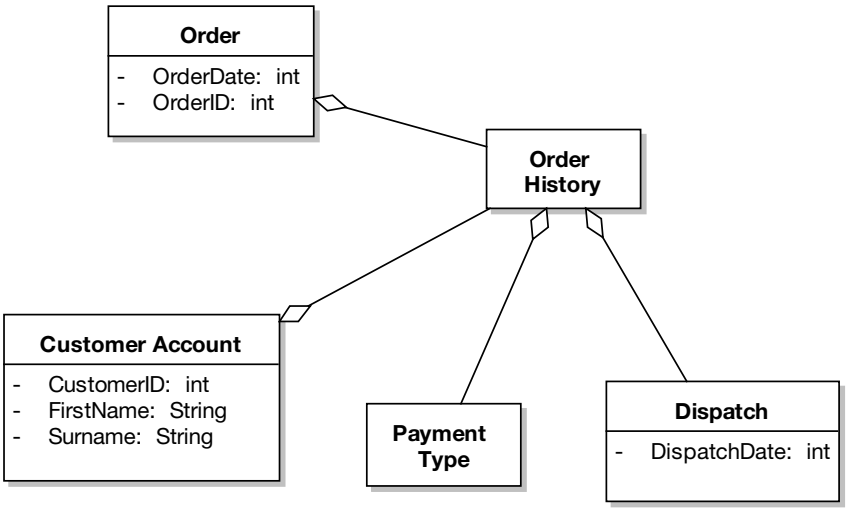


Figure 2-15. *Domain attributes from Figure 2-14, but this time in more appropriate classes*

Exercise 2-3 Solution: Operations and Abstract Classes

The domain model diagram shown in Figure 2-16 has a couple of problems. The first is that `Book Review` is an abstract class. While this isn't especially destructive, and the world probably won't end as a direct result, domain modeling is just a bit too early in the development process to be thinking about these sorts of design details.

Staying with `Book Review`, the other problem is that a couple of operations have been assigned: `checkReviewLength()` and `approveForPublication()`. Identifying and assigning operations to classes is very much a design thing—so again, domain modeling is just too early to be thinking about these sorts of details.

Figure 2-17 shows the corrected diagram.

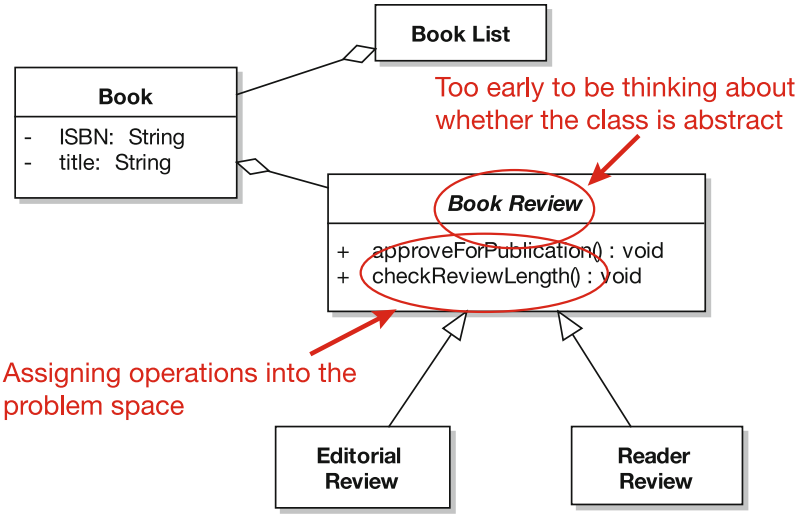


Figure 2-16. *Solution-space details (design) added into the problem space (domain model)*

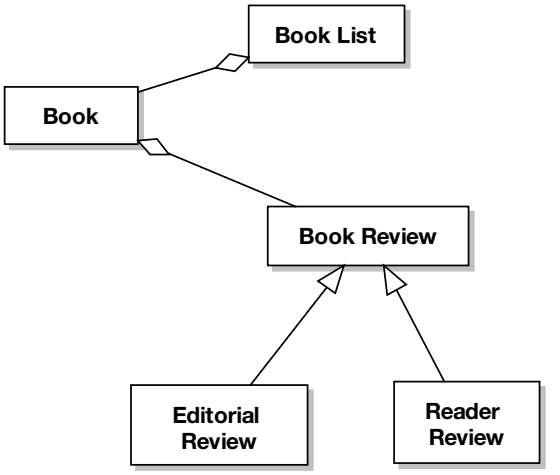


Figure 2-17. *Corrected version of Figure 2-16*

Exercise 2-4 Solution: Premature Patternization

In Figure 2-18, you can see the telltale beginnings of a Factory design pattern. Using this particular design pattern, the `SessionFactory` class will create new instances of `CustomerSession`. A `SessionFactory` is clearly part of the solution space, not the problem space, as are most design patterns. This sounds an awful lot like design, and the use cases haven't even been written yet, so again this is way too early to be thinking about implementation details.

Design patterns usually begin to emerge during robustness analysis (preliminary design), but domain modeling really isn't the time to be thinking about them.

Figure 2-19 shows the corrected diagram.

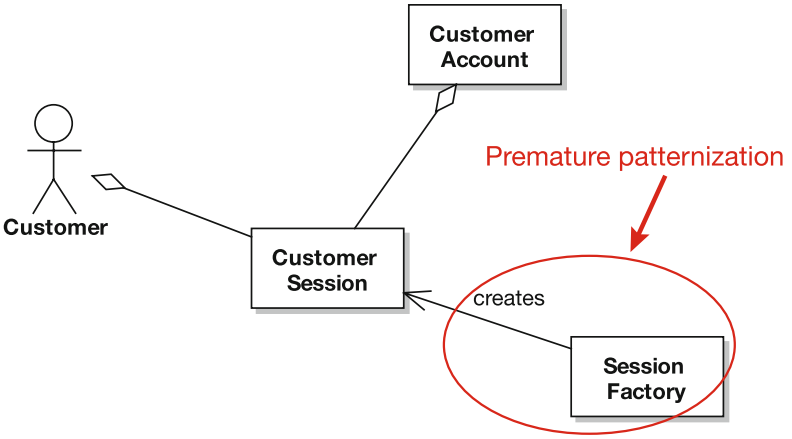


Figure 2-18. *Design details added too early in the project*

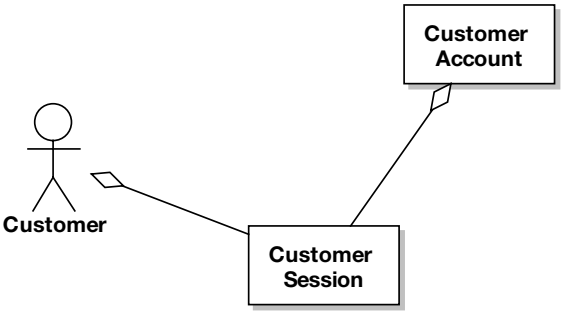


Figure 2-19. *Domain model diagram with the design details removed*

More Practice

This section provides a list of modeling questions that you can use to test your knowledge of domain modeling. The questions get progressively harder, but the answers can all be found by reading (and thinking about!) the domain modeling techniques described in this chapter.

1. Which of the following is ***not*** one of the four types of association in a domain model?
   - **a)** Has-a
   - **b)** Creates
   - **c)** Is-a
   - **d)** Knows about

2. When creating a domain class list, how do you tell when you have an attribute?
   a) An attribute has cardinality in all cases.
   b) An attribute can only be contained in instances with no behavior.
   c) An attribute has a value that is typically not compound.
   d) An attribute has a value that is made up of lots of other values.
3. What technique(s) can you use to figure out domain classes in a system?
   a) Noun phrase analysis
   b) Reverse engineering
   c) Class verb category
   d) Extreme Programming
4. What term is used to describe when a child class is an extension of a parent class?
   a) Aggregation
   b) Inheritance
   c) Composition
   d) Encapsulation
5. Draw a domain model for an online music store, first by trying to imagine how it works in the abstract, without looking at any screens, and then after looking at an example website such as iTunes or Napster. Which of your domain models is better? Explain.
6. Assume you could reverse engineer the database schema from Amazon.com and import this into a visual modeling tool. Would this be a good starting point for a domain model? What changes would need to be made to a reverse-engineered database schema to make it a good domain model?
7. Assume someone hands you some Java code for a GUI prototype of a new Internet bookstore and you reverse engineer it into UML. Would this be a good starting point for a domain model? What changes would need to be made to a reverse-engineered GUI prototype to make it a good domain model?
8. Assume you are working on Release 3 of a project, and you have a detailed set of class diagrams showing the complete implementation of Release 2 that has been reverse engineered from C# code. Release 3 involves migrating the system to a new GUI framework and a different DMBS. What changes would need to be made to your detailed class diagrams from the previous release to make it a good domain model for the current release?

Summary

In this chapter we described in detail the first major stage of ICONIX Process. Domain modeling forms the basis for the whole object modeling activity. As you'll see in the next chapter, the use cases are written in the context of the domain model, and (as you'll see in Chapter 5) robustness analysis helps to tighten up both the domain model and the use case text, bringing them closer together.

The activity diagram in Figure 2-20 shows how domain modeling fits into the overall requirements analysis effort. The activities we covered in this chapter are shown in red.