CHAPTER 8: CLASS AND METHOD DESIGN

Up until now, Alec, Margaret, and the development team members have been focusing on being sure that they captured the underlying behavior and structure of the evolving system. During this installment, Alec instructs the team members to make sure that the connascence is minimized at all levels of the design, to identify any opportunities for reuse, to consider restructuring and optimizing the evolving specification. Furthermore, he instructed them to identify any and all constraints that need to be modeled. He also suggested that they define the invariants in a separate text file and to define the preconditions and postconditions for all public methods using contracts. Finally, he instructed the team to specify every method using the method specification form.

Alec and his team began the detailed object design process by reviewing the class and package diagram for the problem domain layer (see Figures 7-A and 7-B). Alec made it clear that the team should be aware of the cohesion, coupling, and connascence design criteria and to review the models with those in mind. Furthermore, he insisted that they look to see if there were any additional specifications necessary, any opportunities for reuse that could be exploited, and any further restructuring of the design. Alec assigned Anne to review all results and to look for any possible optimizations that could be implemented. Finally, since the implementation would be in Java, he asked Anne to also ensure that the design could be implemented in a single-inheritance-based language.

Upon their review, it was discovered that there were quite a few many-to-many (*..*) association relationships on the class diagram. Alec questioned whether this was a correct representation of the actual situation. Brian admitted that when they put together the class diagram, they had decided to model most of the associations as a many-to-many multiplicity, figuring that this could be easily fixed at a later point in time when they had more precise information. Alec also questioned why this issue was not addressed during the verification and validation step. However, he did not assign any blame at this point in time. Instead, since Brian was the team member that was most familiar with structural modeling and was the analyst in charge of the data management layer (see Chapter 9), Alec assigned him to evaluate the multiplicity of each association in the model and to restructure and optimize the evolving problem domain model.

Figure 8-A shows the updated version of the class diagram. As you can see, Brian included both the lower and upper values of the multiplicity of the associations. He did this to remove any ambiguity about the associations. Since there is a one-to-one relationship between the CD class and the Mkt Info class, Brian considered merging them into a single class. However, he decided that not all CDs would necessarily have any marketing information associated with them for the CD to be included in an order. Consequently, he reasoned that the Mkt Info associated with every CD would be optional and he should keep the Mkt Info separate. He also realized that, even though the team had attempted to verify and validate the structural model, they had not gotten the multiplicities correct in many places. For example, he recognized that he should have known that an artist could be associated with multiple CDs. Consequently, without changing the multiplicities, the Artist Info would have been duplicated for each CD with which the artist was associated.

Upon reviewing the new revised class diagram and since Brian had already spent quite a bit of time on the classes in the CD package, Alec assigned it to him. The classes in the CD package were CD, Vendor, Mkt Info, Review, Artist Info, and Sample Clip. Since Anne was going to have to review all classes and packages from a more technical perspective, Alec

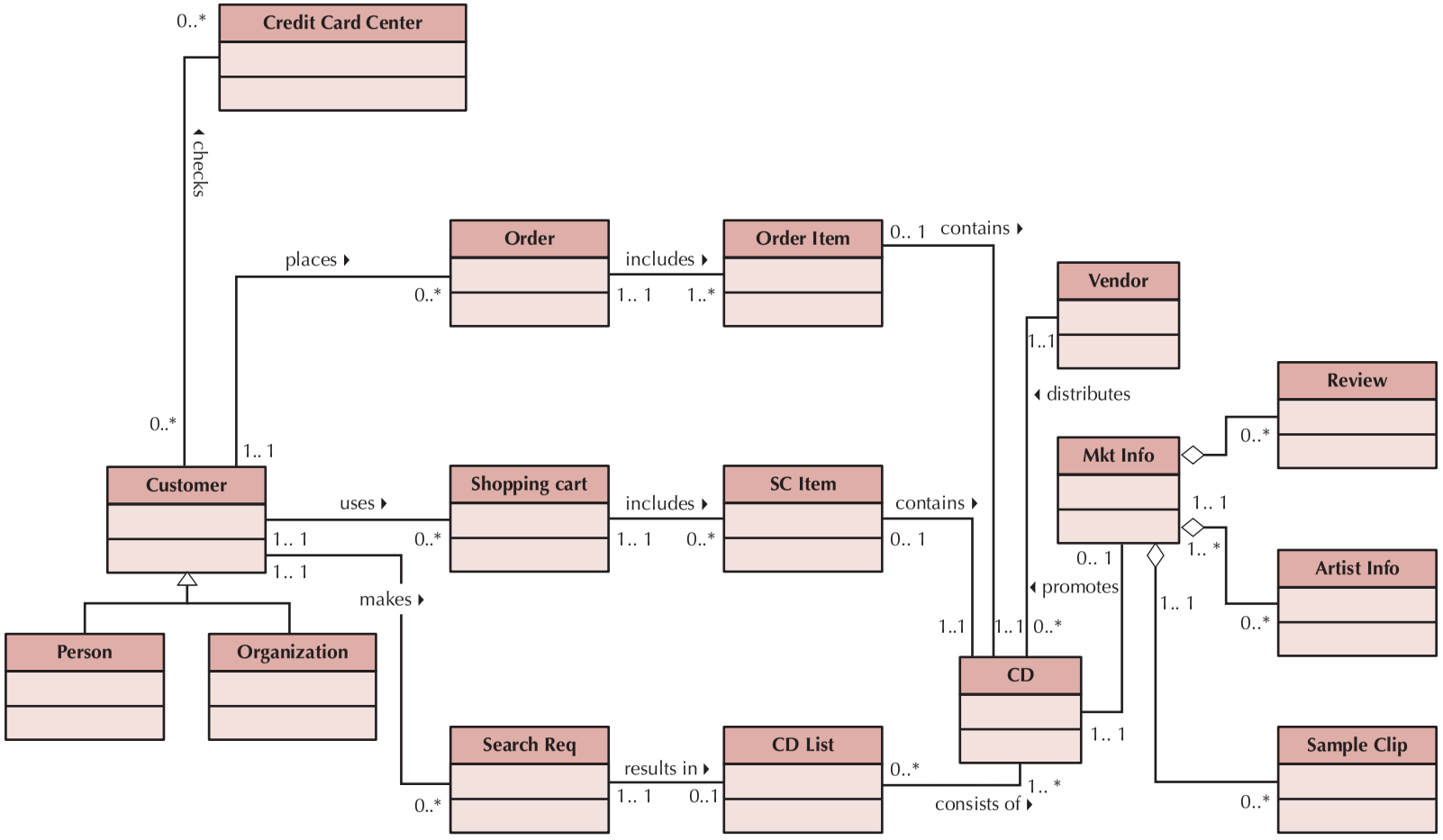

FIGURE 8-A Revised CD Selections Internet Sales System Class Diagram (Places Order Use Case View)

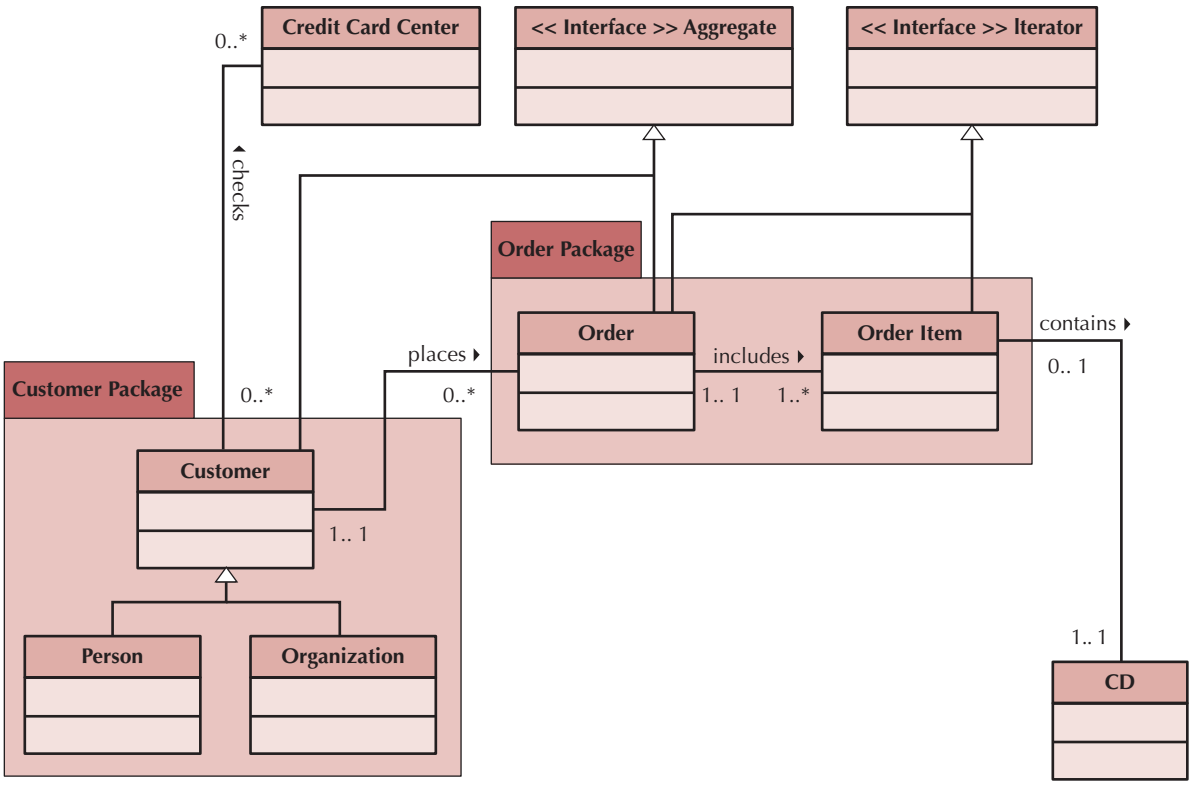

FIGURE 8-B Class Diagram for the Customer and Order classes in the Customer and Order Packages

decided to take on the Customer and Order packages himself (see Figures 7-A and 7-B). However, since the Shopping package could become rather technical, he assigned it to Anne.

Based on earlier projects, Alec suspected that there could be a set of patterns that should be useful in developing the Order Package. With a little work, Alec uncovered the Order, Product Order, and Product classes described in the textbook (see Figures 8-15, 8-19, and 8-20). By using these reusable classes, Alec was able to better define the Order, Order Item, and CD classes, respectively. He also discovered the Iterator and Whole-Part patterns (see Figures 8-13A, 8-13B, and 8-14). He decided that he should reuse the Iterator pattern in both the Customer and Order packages. He also recommended to Brian and to Anne to use the Product class as a basis for the CD class. Finally, Alec recommended to Brian to consider the Whole-Part pattern in the CD package for the Mkt Info class and to Anne, the use of the Iterator pattern for the Shopping Cart and CD List classes in the Shopping package. Alec hopes that by using these patterns the overall quality of the Internet Sales System will be improved in comparison to starting from scratch. Figure 8-B portrays the Order and Customer packages after Alec used the patterns.

Next, Brian added invariants, pre-conditions, and post-conditions to the classes and their methods. For example, Figure 8-C portrays the back of the CRC card for the CD class. He decided to add only the invariant information to the CRC cards and not the class diagram to keep the class diagram as simple and as easy to understand as possible. Notice the additional set of multiplicity, domain, and referential integrity invariants added to the attributes and relationships. Furthermore, Brian created contracts for each method. For example,

Back:

Attributes:

CD Number	(1..1)	(unsigned long)
CD Name	(1..1)	(String)
Pub Date	(1..1)	(Date)
Artist Name	(1..1)	(String)
Artist Number	(1..1)	(unsigned long)
Vendor	(1..1)	(Vendor)
Vendor ID	(1..1)	(unsigned long) {Vendor ID = Vendor.GetVendorID()}

Relationships:

Generalization (a-kind-of):

Aggregation (has-parts):

Other Associations: Order Item {1..1) SC Item (1..1} CD List {0..*} Vendor {1..1}

Mkt Info {0..1}

FIGURE 8-C Back of CD CRC Card

Figure 8-D portrays the contract for the GetReview() method associated with the Mkt Info class. Notice that there is a pre-condition for this method to succeed—Review attribute not Null. Given the overall simplicity of the contracts with the classes in the CD package, Brian decided not to use OCL like constraints (see Figure 8-18). He hopes when the team brings everything back together, that the use of English-like constraints will be sufficient.

Method Name: GetReview()	**Class Name:** Mkt Info	**ID:** 89
Clients (Consumers): CD		
Associated Use Cases: Place Order		
Description of Responsibilities: Return review objects for the Detailed Report Screen to display		
Arguments Received:		
Type of Value Returned: List of Review objects		
Preconditions: Review attribute not Null		
Postconditions:		

FIGURE 8-D Get Review Method Contract

Upon completing the CRC cards and contracts, Brian moved on to specifying the detailed design for each method. For example, the method specification for the GetReview() method is given in Figure 8-E. Brian developed this specification by reviewing the Place Order use case (see Figure 4-G), the sequence diagram (see Figure 6-B), and the contract (see Figure 8-D). Notice that Brian is enforcing the pre-condition on the contract by testing

Method Name: GetReview()	**Class Name:** Mkt Info	**ID:** 453
Contract ID: 89	**Programmer:** John Smith	**Date Due:** 7/7/12

Programming Language:

❑ **Visual Basic** ❑ **Smalltalk** ❑ **C++** ☑ **Java**

Triggers/Events:

Detail Button on Basic Report is pressed

Arguments Received: **Data Type:**	**Notes:**

Messages Sent & Arguments Passed: **ClassName.MethodName:**	**Data Type:**	**Notes:**

Argument Returned: **Data Type:**	**Notes:**
List	List of Review objects

Algorithm Specification:

```
IF Review Not Null
        Return Review
Else
        Throw Null Exception
```

Misc. Notes:

FIGURE 8-E Create Review Method Specification

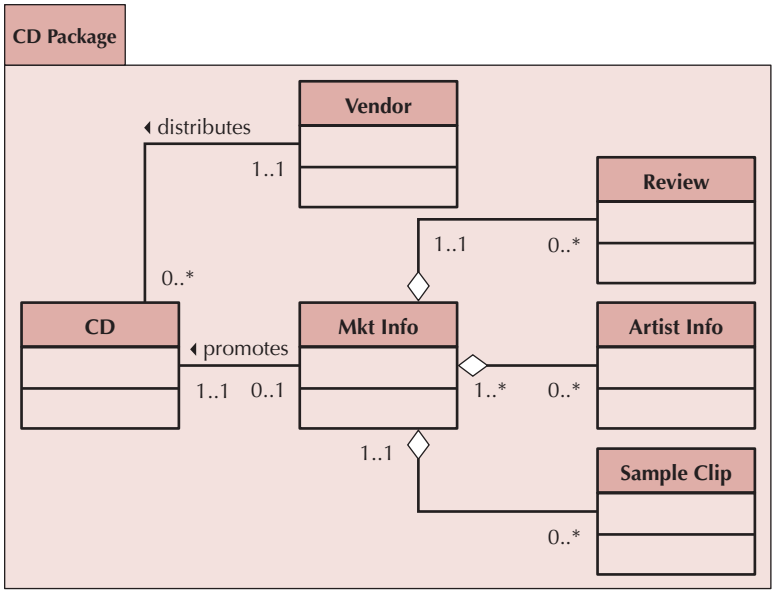

FIGURE 8-F Revised Package Diagram for the CD Package on the PD Layer of CD Selections Internet Sales System

to see whether the Review attribute that contains the list of reviews contains a value or not. Since the method is to be implemented in Java, he has specified that an exception is to be thrown if there are no reviews. Finally, Brian updated the class diagram for the CD package (see Figure 8-F). After looking at the diagram, Brian realized that there could be additional changes necessary depending on how the team decided to implement the data management layer (see Chapter 9). However, he decided that without any additional information regarding the data management layer, the current representation would have to suffice.