Chapter 11 – Object-Oriented Design: Use Case Realizations

Table of Contents
- Chapter Overview
- Learning Objectives
- Notes on Opening Case and EOC Cases
- Key Terms

Chapter Overview
This chapter is an extension of Chapter 10 and builds on the concepts presented there. The purpose of this chapter is to provide in-depth knowledge and skills on how to do low-level design of complex systems.

The primary focus of this chapter is on developing detailed object-oriented design models, which programmers then use to code the system. The two most important models that must be developed are design class diagrams and interaction diagrams, either sequence diagrams or communication diagrams.

Design class diagrams are developed for each layer (domain, view, and data access) of three-layer design to extend the domain model that was developed during analysis.

Interaction diagrams extend the system sequence diagrams, also developed during analysis. This chapter also discusses how to associate classes into package diagrams to show relationships and dependencies.

Finally, the chapter presents principles of good design as defined by design templates or design patterns. Design patterns is an important topic and a broad area – especially for those that begin their careers as programmers. Most of the development frameworks, e.g. Struts for Java, Zend for PHP, Visual Studio for .NET, are based on the Model-View-Controller (MVC) pattern. Very few Information Systems textbooks even address this subject, yet for programmers working in a team, it is a major component of design and programming discussions.

Learning Objectives
After reading this chapter, you should be able to:
- Explain the different types of objects and layers in a design
- Develop sequence diagrams for use case realization
- Develop communication diagrams for detailed design
- Develop updated design class diagrams
- Develop multilayer subsystem packages
- Explain design patterns and recognize various specific patterns
Notes on Opening Case and EOC Cases

Opening Case

New Capital Bank: Part 2: In this case the project leader, Bill Santora, reviews some of the methods that the project team is using to do detailed design. There is also some discussion of the benefits of this approach. Detail design is first done using CRC cards. Then when necessary for the more complex use cases, detailed design is done using sequence diagrams. Also a combined design class diagram was developed that is referenced by all the developers. Some of the benefits are that all programmers are using the same information, the same techniques, and the same approach to programming the system. Hence the system is much more consistent than if each programmer was allowed to program in his own way. A central repository of code enables all programmers to have access to the common pool of code.

EOC Cases

On the Spot Courier Services (running case): On the Spot is a small, but growing, courier service that needs to track customers, package pickups, package deliveries, and delivery routes. This cases extends the solutions developed in Chapter 10. You will create a first-cut DCD, a multilayer sequence diagram with data access classes, a multilayer sequence diagram with view layer classes, a final DCD, and a package diagram with dependency relationships.

Overview

Key Terms

- **use case realization** – the process of elaborating the detailed design with interaction diagrams for a particular use case

The method used to extend the process of detailed design is called **use case realization**. In use case realization, each use case is taken individually to determine all the classes that collaborate on it. As part of that process, any other utility or support classes are identified.

Detailed Design of Multilayer Systems

Key Terms

- **design patterns** – standard design techniques and templates that are widely recognized as good practice

Foundation

This chapter describes in depth the detailed design of all layers of a multilayer system. Figure 10-1 in Chapter 10, illustrates that there are three objects representing the three layers of a system. The input window object represents the view layer and has the primary responsibility of formatting and presenting student information on the screen. The student object represents the middle layer, or business logic layer, for the use case. The database access object represents the third layer in the
multilayer design. It is responsible for connecting to the database, reading the student information, and sending it back to the student object.

**Patterns and the Use Case Controller**

Standard design templates have become popular among software developers because they can speed object-oriented design work. The formal name for these templates is *design patterns*.

In Chapter 10, you were introduced to the concept of a use case controller. For any particular use case, messages come from the external actor to a windows class (that is, an electronic input form) and then to a problem domain class. Designers often define intermediary classes that act as buffers between the user interface and the domain classes.

Figure 11-1 provides a more formal specification for the use case controller pattern. Note that this specification has five main elements:

- Pattern name
- Problem that requires a solution
- Solution to or explanation of the pattern
- Example of the pattern
- Benefits and consequences of the pattern

A use case controller acts as a switchboard, taking input messages and routing them to the correct domain class. A use case controller also contains logic that controls the flow of execution for the use case. In the examples that follow, the authors define a controller class for each use case. A use case controller is a completely artificial class created by the person doing the system design. Sometimes, such classes are called artifacts or artifact objects.

**Use Case Realization with Sequence Diagrams**

**Key Terms**

- **sequence diagrams** – type of interaction diagram that emphasizes the sequence of messages sent between objects for a specific use case
- **communication diagrams** – type of interaction diagram that emphasizes the objects that send and receive messages for a specific use case
- **activation lifeline** – a representation of the period during which a method of an object is alive and executing
- **separation of responsibilities** – a design principle that recommends segregating classes into separate components based on the primary focus of the classes
- **persistent classes** – problem domain classes that must be remembered between program executions (i.e., require storage in a database)

Developing interaction diagrams is at the heart of object-oriented detailed design. The realization of a use case—determining what objects collaborate and the messages they send to each other to carry out
the use case—is done through the development of an interaction diagram. Two types of interaction diagrams can be used during design: **sequence diagrams** or **communication diagrams**.

In this section the authors first explain how to read and understand a sequence diagram. Then they explain the design process, after which they go through some detailed examples.

**Understanding Sequence Diagrams**

The starting point for the detailed design of a use case is always its System Sequence Diagram (SSD). Remember that the SSD only has two lifelines—one for the actor and one for the system. Figure 11-2 reviews the parts of an SSD. The syntax of an input message is:

```
* [true/false condition] return-value := message-name (parameter-list)
```

Figure 11-3 is a two-layer design for the *Create customer account* use case. Note the following points:

- The :System object is replaced by the internal objects, e.g. the ones inside of the system.
- The two-layer solution includes the view layer, e.g. the :CustomerForm, a controller object, and the business layer, e.g. :Customer.
- The createNewCustomer() message goes to the input form :CustomerForm, which sends it on to the controller, which creates the new aC:Customer object.
- The term aC is the identifier of the new :Customer object, and is returned to the controller as shown by the return value on the message. Thus the controller has visibility to the aC:Customer object.
- Another way to show return values is with a dotted arrow as indicated on the message returning from the controller to the view layer.
- The **activation lifeline** represents the time that the object is process an input message. It is represented by a narrow vertical box.

When a message is sent from an originating object to a destination object, in programming terms, it means that the originating object is invoking a method on the destination object. Thus, by defining the messages to various internal objects, you are actually identifying the methods of that object.

**Design Process for Use Case Realization**

Figure 11-4 describes the detailed steps of design by use case realization.

1. Develop the first-cut design class diagram showing navigation visibility.
2. Determine the class responsibilities and class collaborations for each use case using CRC cards.
3. Develop detailed sequence diagrams for each use case.
   (a) Develop the first-cut sequence diagrams.
   (b) Develop the multilayer sequence diagrams.
4. Update the DCD by adding method signatures and navigation information.
5. Partition the solution into packages, as appropriate.
First-Cut Sequence Diagram: Create customer account Use Case

This example goes through steps one and step three steps for this use case beginning with the SSD and the domain model. (The authors do not use step #2 with the CRC cards.)

1. First, the Customer class would have visibility to the other two classes: Account and Address.
2. Next create the first-cut sequence diagram.
   a. The first step in expanding an SSD is to place the problem domain objects in the diagram, along with the input messages from the SSD.
   b. The next step is to determine the internal messages that must be sent between the objects, including which object should be the source and destination of each message, using good design principles of coupling, cohesion, object responsibility and controllers.

Figure 11-8 illustrates the solution:
- Figure 11-8 focuses only on the domain classes.
- The :CustomerHandler controller receives the input messages, searches for the correct order object, and forwards the createNewCustomer message to the correct :Customer object. The :Customer object takes responsibility to save itself to the database based on the createNewCustomer input message. For the other messages—enterAddress and enterCreditCard—it also takes responsibility for creating these new objects.
- When identifying and creating messages, you must first determine the origin and destination objects for the message. The origin object is the one that needs information or help in carrying out a responsibility so it will initiate a message. The destination object is the one that has the information to help in the solution and will receive the message and process it.

Some design principles include:
- The use case controller provides the link between the internal objects and the external environment.
- The responsibility assigned to :Customer is to be in charge of creating itself and to control all the other required updates.
- The :Address and :Account objects create themselves and save themselves to the database. Coupling is straightforward, being basically vertical on the hierarchy.

First-Cut Sequence Diagram: Fill shopping cart Use Case

The process is the same as before. This use case is a just a different type of example.

Figure 11-10 shows that there are only two input messages to the system: adding an item and adding an accessory item. As you analyze the SSD, notice that adding an item to the shopping cart and adding an accessory to the cart are the same operation.

Begin this design by developing the first-cut DCD. The Customer, Cart, and CartItem classes are necessary because the use case will be adding items for this customer to the customer’s cart. To create a cart item, the system will need to know what product it is, if there are items in stock, and the price for
the item. Therefore, other classes that are required are InventoryItem, ProductItem, and PromoOffering. As you develop the solution, you may have to add classes.

Navigation visibility between these classes will be from the controller to the Customer class and to the Cart class once it has been created. The Cart class will be able to access the CartItem class. The CartItem class should have visibility to the other classes, such as ProductItem and InventoryItem, that contain the necessary information. Figure 11-11 shows the DCD.

Expanding the input messages to add the internal messages you get Figure 11-12 as the first-cut sequence diagram.

The input addItemToCart message originates from the Customer Actor (note: this is in error in textbook) and is directed to the :CartHandler controller object. The controller object determines if this is the very first item for this customer and, if so, sends a message to the :Customer object to create a new online cart. In UML, when a create message is sent to an object, it is often drawn directly to the object’s box and not to the lifeline. The reference to the cart—aCrt—is then passed back to the controller, which uses it to send the addItemToCart message to the :OnlineCart object. The :OnlineCart object processes that message by creating a :CartItem object. The :CartItem object then takes responsibility to get its price and description and see that the inventory is updated. It does this by sending appropriate messages to those objects.

The bottom part of the sequence diagram is similar except that adding accessory items may require multiple accessories to be added for each primary item. Hence, it is shown with the rectangular box for the loop notation.

Important considerations include:

- Which classes “own” other classes and hence have responsibility to create them?
- Which objects are the sources and which are the destinations for the messages?
- Which classes have visibility to which other classes? And how is that visibility supported?

**Guidelines and Assumptions for First-Cut Sequence Diagram Development**

**Guidelines:** The following three design tasks are done to produce the preliminary sequence diagram:

- Take each input message and determine all of the internal messages that result from that input. For that message, determine its objective. Determine what information is needed, what class needs it—the destination—and what class provides it—the source. Determine whether any objects are created as a result of the input.
- As you work with each input message, identity the complete set of classes that will be affected by that message. Select all the objects from the domain class diagrams and include use case preconditions and postconditions.
- Flesh out the components for each message by adding iteration, true/false conditions, return values, and passed parameters.
Assumptions: The development of the first-cut sequence diagram is based on several simplifying assumptions. The following are three of these assumptions:

- **Perfect technology assumption.** You first encountered this assumption in Chapter 5 when you were identifying business events. You continue that assumption here. You do not include steps such as the user having to log on or testing the availability of the network.

- **Perfect memory assumption.** You may have noticed that the authors just assumed that the necessary objects were in memory and available for the use case. They did not ask whether those objects were created in memory. They will change this assumption when they get to multilayer design. In multiple-layer design, the authors do include the steps necessary to create objects in memory.

- **Perfect solution assumption.** The first-cut sequence diagram assumes that there are no exception conditions. No logic is included to handle a situation in which the requested catalog or product is not found.

Developing a Multilayer Design

The development of the first-cut sequence diagram focuses only on the classes in the problem domain layer. In many instances, this may be sufficient documentation to program the solution—either by yourself or with another programmer.

Every system will need view layer classes to represent the input and output screens for the application. Data access layer classes aren’t always required. The data access layer is required when the business logic is fairly complex and should be isolated from the SQL statements that access the database.

**Designing the Data Access Layer:** The principle of separation of responsibilities is the motivating factor behind the design of the data access layer. Problem domain classes are also persistent classes, which means that their data values must be stored by the system even when the application isn’t executing.

In this chapter, The authors take a somewhat simplified design approach in order to teach the basic ideas without getting embroiled in the complexities of database access. They assume that every domain object has a table in a relational database.

There are two techniques for linking the domain layer to the data access layer. One way is to have the constructor method of each problem domain object invoke the data access object to get the necessary information to complete the instantiation of the new object. Another way is to send a message to the data access object and have it read the database and then instantiate a new problem domain object.

**The Data Access Layer for the Fill shopping cart Use Case:** To design the data access layer, you no longer assume that the objects are automatically in memory when you need them; that is, you disregard the perfect memory assumption. In Figure 11-13 notice that all the original internal messages are still there. However, associated with each problem domain object is a data access class, and associated with each original internal message are additional messages to retrieve data from or save data to the database via the data access object. It is important during this process to ensure that source objects have navigation visibility to destination objects so messages can be sent.
Comparing Figure 11-12 to Figure 11-13 you will find the following additions to the original messages:

- **[first Time]createCart**—The cart handler is going to send a message to a customer object to create a cart. First, it needs to ensure that there is a customer object in memory. It sends a findCustomer message to the `aC:Customer` object to find and create itself from the database. It does so by sending a message to the `:CustomerDA` object to read the database and return the appropriate customer object.

- **addItemToCart**—This message is initially the same in both figures. After `aCrt:CartItem` has been created and populated with data, a message is sent to the data access object to save the data to the database.

- **getPrice, getDesc, updateQty**—These three messages all access or update the database. Therefore, each also requires a previous message to find the appropriate data from the database, which is stored in a domain object in memory.

**Designing the View Layer:** Although adding the user interface classes may sound simple, it must be done in conjunction with the detailed design of the user interface forms, as described in Chapter 7. There are two sources of inputs for the design of the view layer. First, of course, is the user interface components that were designed during user interface design. The second source is either the first-cut sequence diagram or the sequence diagram with the data access classes identified.

In Figure 11-14, the authors have added the two view layer objects for searching items and viewing accessories. The first input message—`addItemToCart`—will go through the `:SearchItemWindow` object. The message then causes a detailed `:AddItemWindow` object to display and show the details to verify the addition to the cart. This later window will forward the message on to the `:CartHandler` object. The next three view layer objects—`:ViewAccessWindow`, `:AddAccessWindow`, and `:DisplayItem+AccessWindow`—function in a manner similar to the other view layer objects. The only difference is that the data includes the item and the accessories that have been added to the online cart.

Adding the view layer to your design is a good way to verify that the user interface that was developed with the users is consistent with the application design. All the input messages that were identified and documented on SSDs must be handled by the user interface.

**Designing with Communication Diagrams**

**Key Terms**

none

Communication diagrams and sequence diagrams are interaction diagrams, and they capture the same information. A communication diagram uses the same symbols for actors, objects, and messages as a sequence diagram. The lifeline and activation lifeline symbols aren’t used. However, a different symbol—the link symbol—is used. In a communication diagram, a link shows that two items share a message—that one sends a message and the other receives it.

The syntax of the message descriptor in a communication diagram is as follows:

\[
[true/false \text{condition}] \ sequence-number: \ return-value := \ message-name (parameter-list)
\]
The numbers on the messages indicate the sequence in which the messages are sent. The hierarchical dot numbering scheme is used when messages depend on previous messages.

When you compare the communication diagrams with the sequence diagrams, it should be evident that the focus of a communication diagram is on the objects themselves. Drawing a communication diagram is an effective way to get a quick overview of the objects that work together. However, as you look at the diagrams, you should see that it is more difficult to visualize the sequence of the messages.

**Updating and Packaging the Design Classes**

**Key Terms**

- dependency relationship – a relationship between packages, classes, or use cases in which a change in the independent item requires a change in the dependent item

Design class diagrams can now be developed for each layer. In the view layer and the data access layer, several new classes must be specified. The domain layer also has some new classes added for the use case controllers.

The first step in updating the DCD is to add the method signatures based on the information from the sequence diagrams. Three types of methods are found in most classes: (1) constructor methods, (2) data-get and data-set methods, and (3) use case-specific methods. To avoid information overload, most developers don’t include the get and set methods in the DCD.

The process of adding method signatures to a design class is to go through every sequence diagram and find the messages sent to that class. Each message indicates a method.

In addition to the method signatures, the DCD is updated with any new classes and any new navigation arrows.

**Structuring the Major Components with Package Diagrams**

As you learned previously, a package diagram in UML is simply a high-level diagram that allows designers to associate classes of related groups. In this example in Figure 11-19 the classes are placed inside the appropriate package based on the layer to which they belong.

The other symbol used on a package diagram is a dashed arrow, which represents a dependency relationship. The arrow’s tail is connected to the package that is dependent, and the arrowhead is connected to the independent package. A good way to think about a dependency relationship is that if one element changes (the independent element), the other (dependent) element might also have to be changed. Dependency relationships are used in package diagrams, class diagrams, and even interaction diagrams. Package diagrams can also be nested to show different levels of packages.

**Implementation Issues for Three-Layer Design**

Using design class diagrams, interaction diagrams, and package diagrams, programmers can begin to build the components of a system.

Based on the design principle “object responsibility,” it is possible to define which program responsibilities belong to each layer. If you follow these guidelines when writing code, a system will be
much easier to maintain throughout its lifetime. Let us summarize the primary responsibilities of each layer.

View layer classes should have programming logic to perform the following:

- Display electronic forms and reports.
- Capture such input events as clicks, rollovers, and key entries.
- Display data fields.
- Accept input data.
- Edit and validate input data.
- Forward input data to the domain layer classes.
- Start and shut down the system.

Domain layer classes should have responsibilities to perform the following:

- Create problem domain (persistent) classes.
- Process all business rules with appropriate logic.
- Prepare persistent classes for storage to the database.

Data access layer classes should have responsibilities to perform the following:

- Establish and maintain connections to the database.
- Contain all SQL statements.
- Process result sets (the results of SQL executions) into appropriate domain objects.
- Disconnect gracefully from the database.

Design Patterns

Key Terms

none

Systems that are based on good design principles aren’t only easier to develop and put into operation the first time, they are also much easier to maintain. Such concepts as object responsibility, coupling, cohesion, protection from variations, and indirection are key design principles that are implemented through the use of design patterns. In this section the authors introduce only a few design patterns to teach the concept. There are many more design patterns.

Adapter

The adapter pattern is also a good example of the design principles “protection from variations” and “indirection.” The adapter design pattern works just like the electrical adapter; it plugs an external class into an existing system. The method signatures on the external class are different from the method names being called from within the system, so the adapter class is inserted to convert the method calls from within the system to the method names in the external class. It is a powerful and elegant solution
to making a system more maintainable. Experienced developers use this pattern frequently—for foreign classes and for internally written classes that may need frequent upgrades.

**Factory**

In the discussions of detailed design, the authors have often expressed the need to have utility classes. What class should create these utility objects? In most situations, it doesn’t make sense for domain classes to create them because it isn’t a listed responsibility of domain classes. A popular solution in object-oriented programming is to have some classes that are factories. In other words, these classes instantiate objects of the utility classes.

**Singleton**

Some classes must have exactly one instance—for example, a factory class or the main window class. Because these classes are instantiated from only one place, it is easy to limit the logic to create only one object. Other classes must have exactly one instance but can’t be easily controlled by having only one place to invoke the constructor. Depending on the system’s flow of logic, a particular class might get instantiated from multiple locations. However, only one instance needs to be created, so the first one that needs it creates it, and every other class uses the one that was initially created. This common problem has a standard solution: the singleton pattern.

The singleton pattern has the same basic logic as the factory method pattern. The difference is that the singleton class has code that applies to itself as static methods.

**Design Summary**

Object responsibility is the foundation upon which all object-oriented methodology is based. Design is use-case driven. The use case controller class provides a common access point into the system logic for a given use case or business transaction.

As indicated in the chapter, the first step in this activity is to identify the core process required for the use case. By core, the authors mean only the classes from the domain model. If you identify this core process correctly, the rest of the design will evolve easily. This design activity consists of three steps, each of which corresponds to a layer in three-layer design. First, design the core process based on the domain model. Second, add the data access layer classes. Third, add the view layer classes (that is, the GUI).

The basic principle of design is to identify both the classes that are involved in the use case and the messages that must be passed between the objects to execute the use case. The system sequence diagram, in conjunction with the activity diagrams or fully developed use case workflow descriptions, drives this design process.

The system must process each input message from the actor to the system. Design involves defining of all the internal tasks that the system must perform to process an input message. For example, if the input message is a “verify” or “find” type of message, the system must find the right object and verify its existence or status or attribute values. The designer must identify all the objects that are associated with the task and each object’s responsibility. This is the central activity of doing object-oriented
design. You will need to practice this activity quite a bit to completely understand how it works. If you are new to object-oriented design, you will probably need to supplement the material in the textbook with other readings. Again, the authors recommend Craig Larman’s book (2nd edition or 3rd edition) as a fairly easy to read that can presented in small chunks.

The following questions can help you to develop a correct and comprehensive set of internal messages for each input message:

- What class has the primary responsibility to “receive and control the processes” for the input message? (Usually, this is the use case controller class.)
- What domain model class should take primary responsibility for the “core” processes of the service requested by this input message?
- What information is required? What comes in on the input message? What information is required to carry out the service request? From where does this information come (for example, which objects)?
- What objects need to be instantiated? What class has the responsibility to instantiate those objects?
- What relationships need to be built? What messages may be needed to establish these relationships?
- What information needs to be verified and checked? What other objects need to be involved to provide this verification?
- What information needs to be returned? What outputs are expected from the input message?
- What navigation visibility is available (from the first-cut design class diagram)? What other visibility will be required (and, therefore, built as part of the process)?
- Is the solution consistent with low coupling and high cohesion? Is it consistent with principles of good object responsibility?
- Is the solution complete—this is, does it satisfy the post-conditions described in the use case specifications?