Beginning Object-Oriented Programming with VB 2005

From Novice to Professional

Daniel R. Clark

Apress®

Beginning Object-Oriented Programming with VB 2005: From Novice to Professional

ISBN (pbk): 1-59059-576-9

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

Lead Editor: Jonathan Hassell
Technical Reviewer: Jon Box
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editors: Marilyn Smith and Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Janet Vail
Compositor: Kinetic Publishing Services, LLC
Proofreader: Christy Wagner
Indexer: Rebecca Plunkett
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 source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 4

Designing OOP Solutions: A Case Study

Designing solutions for an application is not an easy endeavor. Becoming an accomplished designer takes time and a conscious effort, which explains why many developers avoid it like the plague. You can study all the theories and know all the buzzwords, but the only way to truly develop your modeling skills is to roll up your sleeves, get your hands dirty, and start modeling. In this chapter, you will go through the process of modeling an office-supply ordering system. Although this is not a terribly complex application, it will serve to help solidify the modeling concepts covered in the previous chapters. By analyzing the case study, you will also gain a better understanding of how a model is developed and how the pieces fit together.

After reading this chapter, you should be familiar with the following:

- How to model an OOP solution using UML tools
- Some common OOP design pitfalls to avoid

Developing an OOP Solution

In the case-study scenario, your company currently has no standard way for departments to order office supplies. Each department separately implements its own ordering process. As a result, it is next to impossible to track company-wide spending on supplies, which impacts the ability to forecast budgeting and identify abuses. Another problem with the current system is that it does not allow for a single contact person who could negotiate better deals with the various vendors.

As a result, you have been asked to help develop a company-wide office-supply ordering (OSO) application. To model this system you will complete the following steps:

- Create an SRS.
- Develop the use cases.
- Diagram the use cases.
- Model the classes.
- Model the user interface design.

Creating the System Requirement Specification

After interviewing the various clients of the proposed system, you develop the SRS. Remember from Chapter 2 that the SRS scopes the system requirements, defines the system boundaries, and identifies the users of the system.

You have identified the following system users:

- **Purchaser:** Initiates a request for supplies
- **Department manager:** Tracks and approves supply requests from department purchasers
- **Supply vendor processing application:** Receives XML order files generated by the system
- **Purchase manager:** Updates supply catalog, tracks supply requests, and checks in delivered items

You have identified the following system requirements:

- Users must log in to the system by supplying a username and password.
- Purchasers will view a list of supplies that are available to be ordered.
- Purchasers will be able to filter the list of supplies by category.
- Purchasers can request multiple supplies in a single purchase request.
- A department manager can request general supplies for the department.
- Department managers must approve or deny supply requests for their department at the end of each week.
- If department managers deny a request, they must supply a short explanation outlining the reason for the denial.
- Department managers must track spending within their departments and ensure there are sufficient funds for approved supply requests.
- A purchase manager maintains the supply catalog and ensures it is accurate and up to date.
- A purchase manager checks in the supplies when they are received and organizes the supplies for distribution.
- Supply requests that have been requested but not approved are marked with a status of pending.
- Supply requests that have been approved are marked with a status of approved and an order is generated.
- Once an order is generated, an XML document containing the order details is placed in an order queue. Once the order has been placed in the queue, it is marked with a status of placed.

- A separate supply vendor processing application will retrieve the order XML files from the queue, parse the documents, and distribute the line items to the appropriate vendor queues. The vendor will retrieve the order XML documents from the queue.
- When all the items of an order are checked in, the order is marked with a status of fulfilled and the purchaser is informed that the order is ready for pickup.

Developing the Use Cases

After generating the SRS and getting the appropriate system users to sign off on it, the next task is to develop the use cases, which will define how the system will function from the users' perspective. The first step in developing the use cases is to define the actors. Remember from Chapter 2 that the actors represent the external entities (human or other systems) that will interact with the system. From the SRS, you can identify the following actors that will interact with the system:

- Purchaser
- Department Manager
- Purchase Manager
- Supply Vendor Processing Application

Now that you have identified the actors, the next step is to identify the various use cases with which the actors will be involved. By examining the requirement statements made in the SRS, you can identify the various use cases. For example, the statement "Users must log in to the system by supplying a username and password" indicates the need for a Login use case. Table 4-1 identifies the use cases for the OSO application.

Table 4-1. *Use Cases for the OSO Application*

Name	Actor(s)	Description
Login	Purchaser, Department Manager, Purchase Manager	Users see a login screen. They then enter their username and password. They either click Log In or Cancel. After login, they see a screen containing product information.
View Supply Catalog	Purchaser, Department Manager, Purchaser Manager	Users see a catalog table that contains a list of supplies. The table contains information such as the supply name, category, description, and cost. Users can filter supplies by category.
Purchase Request	Purchaser, Department Manager	Purchasers select items in the table and click a button to add them to their cart. A separate table shows the items in their cart, the number of each item requested and the cost, as well as the total cost of the request.

(*Continued*)

Table 4-1. (*Continued*)

Name	Actor(s)	Description
Department Purchase Request	Department Manager	Department managers select items in the table and click a button to add them to their cart. A separate table shows the items in their cart, the number of each item requested and the cost, as well as the total cost of the request.
Request Review	Department Manager	Department managers see a screen that lists all pending supply requests for members of their department. They review the requests and mark them as approved or denied. If they deny the request, they enter a brief explanation.
Track Spending	Department Manager	Department managers see a screen that lists the monthly spending of department members as well as the running total of the department.
Maintain Catalog	Purchase Manager	The purchase manager has the ability to update product information, add products, or mark products as discontinued. The administrator can also update category information, add categories, and mark categories as discontinued.
Item Check In	Purchase Manager	The purchase manager sees a screen for entering the order number. The purchase manager then sees the line items listed for the order. The items that have been received are marked. When all the items for an order are received, it is marked as fulfilled.
Order Placement	Supply Vendor Processing Application	The supply vendor processing application checks the queue for outgoing order XML files. Files are retrieved, parsed, and sent to the appropriate vendor queue.

Diagramming the Use Cases

Now that you have identified the various use cases and actors, you are ready to construct a visual design of the use cases using a UML modeling program. Figure 4-1 shows a preliminary use case model developed with Objecteering's UML Modeler, which was introduced in Chapter 2.

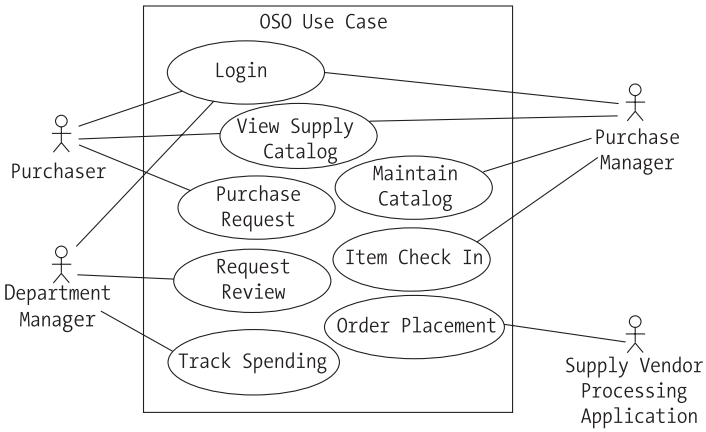

Figure 4-1. *Preliminary OSO use case diagram*

After you have diagrammed the use cases, you now look for any relationships that may exist between the use cases. Two relationships that may exist are the includes relationship and the extends relationship. Remember from the discussions in Chapter 2 that when a use case includes another use case, the use case being included needs to run as a precondition. For example, the Login use case of the OSO application needs to be included in the View Supply Catalog use case. The reason you make Login a separate use case is that the Login use case can be reused by one or more other use cases. In the OSO application, the Login use case will also be included with the Track Spending use case. Figure 4-2 depicts this includes relationship.

■ **Note** In some modeling tools, the includes relationship may be indicated in the use case diagram by the `uses` keyword.

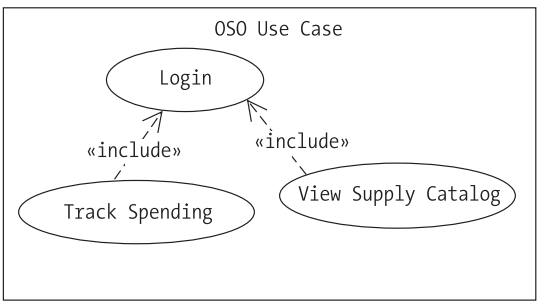

Figure 4-2. *Including the Login use case*

The extends relationship exists between two use cases when, depending on a condition, a use case will extend the behavior of the initial use case. In the OSO application, when a manager is making a purchase request, she can indicate that she will be requesting a purchase for the department. In this case, the Department Purchase Request use case becomes an extension of the Purchase Request use case. Figure 4-3 diagrams this extension.

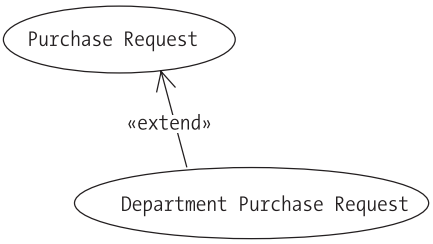

Figure 4-3. *Extending the Purchase Request use case*

After analyzing the system requirements and use cases, you can make the system development more manageable by breaking up the application and developing it in phases. For example you can develop the Purchase Request portion of the application first. Next, you can develop the Request Review portion, and then the Item Check In portion. The rest of this chapter focuses on the Purchase Request portion of the application. Employees and department managers will use this part of the application to make purchase requests. Figure 4-4 shows the use case diagram for this phase.

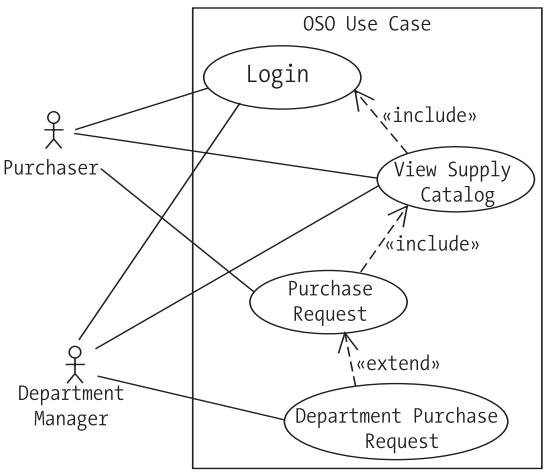

Figure 4-4. *Purchase Request use case diagram*

Developing the Class Model

Developing the class model involves several tasks. You begin by identifying the classes, and then add attributes, associations, and behaviors.

Identifying the Classes

After you have identified the various use cases, you can start identifying the classes the system needs to include to carry out the functionality described in the use cases. To identify the classes, you drill down into each use case and define a series of steps needed to carry it out. It is also helpful to identify the noun phrases in the use case descriptions. The noun phrases are often good indicators of the classes that will be needed.

For example, the following steps describe the View Supply Catalog use case:

1. User has logged in and been assigned a user status level. (This is the precondition.)
2. Users are presented with a catalog table that contains a list of supplies. The table contains information such as the supply name, category, description, and cost.
3. Users can filter supplies by category.
4. Users are given the choice of logging out or making a purchase request. (This is the postcondition.)

From this description, you can identify a class that will be responsible for retrieving product information from the database and filtering the products being displayed. The name of this class will be the `ProductCatalog` class.

Examining the noun phrases in the use case descriptions dealing with making purchase requests reveals the candidate classes for the OSO application, as listed in Table 4-2.

Table 4-2. *OSO Candidate Classes Used to Make Purchase Requests*

Use Case	Candidate Classes
Login	User, username, password, success, failure
View Supply Catalog	User, catalog table, supplies, information, supply name, category, description, cost
Purchase Request	Purchaser, items, cart, number, item requested, cost, total cost
Department Purchase Request	Department manager, items, cart, number, item requested, cost, total cost, department purchase request

Now that you have identified the candidate classes, you need to eliminate the classes that indicate redundancy. For example, a reference to items and line items would represent the same abstraction. You can also eliminate classes that represent attributes rather than objects. Username, password, and cost are examples of noun phrases that represent attributes. Some classes are vague or generalizations of other classes. User is actually a generalization of purchaser and manager. Classes may also actually refer to the same object abstraction but indicate a different state of the object. For example, the supply request and order represent the same abstraction before and after approval. You should also filter out classes that represent implementation constructs such as list and table. For example, a cart is really a collection of order items for a particular order.

Using these elimination criteria, you can whittle down the class list to the following candidate classes:

- `Employee`
- `DepartmentManager`
- `Order`
- `OrderItem`
- `ProductCatalog`
- `Product`

You can also include classes that represent the actors that will interact with the system. These are special classes called *actor classes* and are included in the class diagram to model the interface between the system and the actor. For example, you could designate a `Purchaser(UI)` actor class that represents the GUI that a Purchaser (`Employee` or `DepartmentManager`) would interact with to make a purchase request. Because these classes are not actually part of the system, the internal implementations of these classes are encapsulated, and they are treated as black boxes to the system.

You can now start formulating the class diagram for the Purchase Request portion of the OSO application. Figure 4-5 shows the preliminary class diagram for the OSO application.

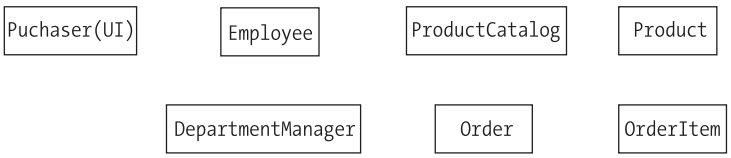

Figure 4-5. *Preliminary OSO class diagram*

Adding Attributes to the Classes

The next stage in the development of the class model is to identify the level of abstraction the classes must implement. You determine what state information is relevant to the OSO application. This required state information will be implemented through the attributes of the class. Analyzing the system requirements for the `Employee` class reveals the need for a login name, password, department, and whether the user is a manager. You also need an identifier such as an employee ID to uniquely identify various employees. An interview with managers revealed the need to include the first and last names of the employee so that they can track spending by name. Table 4-3 summarizes the attributes that will be included in the OSO classes.

Table 4-3. *OSO Class Attributes*

Class	Attribute	Type
Employee	EmployeeID	Integer
	LoginName	String
	Password	String
	Department	String
	FirstName	String
	LastName	String
DepartmentManager	EmployeeID	Integer
	LoginName	String
	Password	String
	Department	String
	FirstName	String
	LastName	String
Order	OrderNumber	Long
	OrderDate	Date
	Status	String
OrderItem	ProductNumber	String
	Quantity	Short
	UnitPrice	Decimal
Product	ProductNumber	String
	ProductName	String
	Description	String
	UnitPrice	Decimal
	VendorCode	String
ProductCatalog	None	

Figure 4-6 shows the OSO class diagram with the class attributes. I have left out the attributes for the DepartmentManager class. The DepartmentManager class will probably inherit the attributes listed for the Employee class.

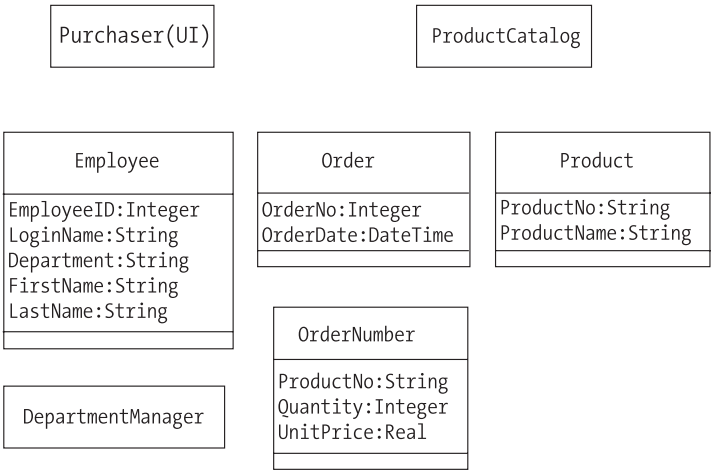

Figure 4-6. *OSO Purchase Request component class diagram with attributes added*

Identifying Class Associations

The next stage in the development process is to model the class associations that will exist in the OSO application. If you study the use cases and SRS, you can gain an understanding of what types of associations you need to incorporate into the class structural design.

■ **Note** You may find that you need to further refine the SRS to expose the class associations.

For example, an employee will be associated with an order. By examining the multiplicity of the association, you discover that an employee can have multiple orders, but an order can be associated with only one employee. Figure 4-7 models this association.

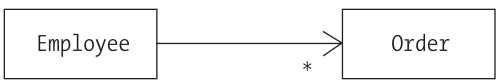

Figure 4-7. *Depicting the association between the Employee class and the Order class*

As you start to identify the class attributes, you will notice that the `Employee` class and the `DepartmentManager` class have many of the same attributes. This makes sense, because a manager is also an employee. For the purpose of this application, a manager represents an employee with specialized behavior. This specialization is represented by an inheritance relationship, as shown in Figure 4-8.

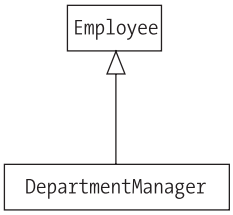

Figure 4-8. *The DepartmentManager class inheriting from the Employee class*

The following statements sum up the associations in the OSO class structure:

- An `Order` is an aggregation of `OrderItem` objects.
- An `Employee` can have multiple `Order` objects.
- An `Order` is associated with one `Employee`.
- The `ProductCatalog` is associated with multiple `Product` objects.
- A `Product` is associated with the `ProductCatalog`.
- An `OrderItem` is associated with one `Product`.
- A `Product` may be associated with multiple `OrderItem` objects.
- A `DepartmentManager` is an `Employee` with specialized behavior.

Figure 4-9 shows these various associations (excluding the class attributes for clarity).

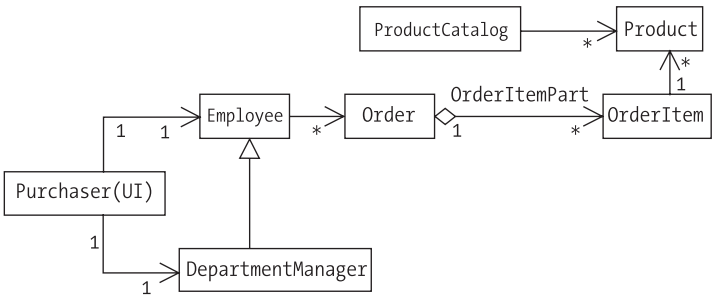

Figure 4-9. *The OSO Purchase Request component class diagram with associations added*

Modeling the Class Behaviors

Now that you have sketched out the preliminary structure of the classes, you are ready to model how these classes will interact and collaborate. The first step in this process is to drill down into the use case descriptions and create a more detailed scenario of how the use case will be carried out. The following scenario describes one possible sequence for carrying out the Login use case.

1. The user is presented with a login dialog box.
2. The user enters a login name and a password.
3. The user submits the information.
4. The name and password are checked and verified.
5. The user is presented with a supply request screen.

Although this scenario depicts the most common processing involved with the Login use case, you may need other scenarios to describe anticipated alternate outcomes. The following scenario describes an alternate processing of the Login use case:

1. The user is presented with a login dialog box.
2. The user enters a login name and a password.
3. The user submits the information.
4. The name and password are checked but cannot be verified.
5. The user is informed of the incorrect login information.
6. The user is presented with a login dialog box again.
7. The user either tries again or cancels the login request.

At this point, it may help to create a visual representation of the scenarios outlined for the use case. Remember from Chapter 3 that activity diagrams are often used to visualize use case processing. Figure 4-10 shows an activity diagram constructed for the Login use case scenarios.

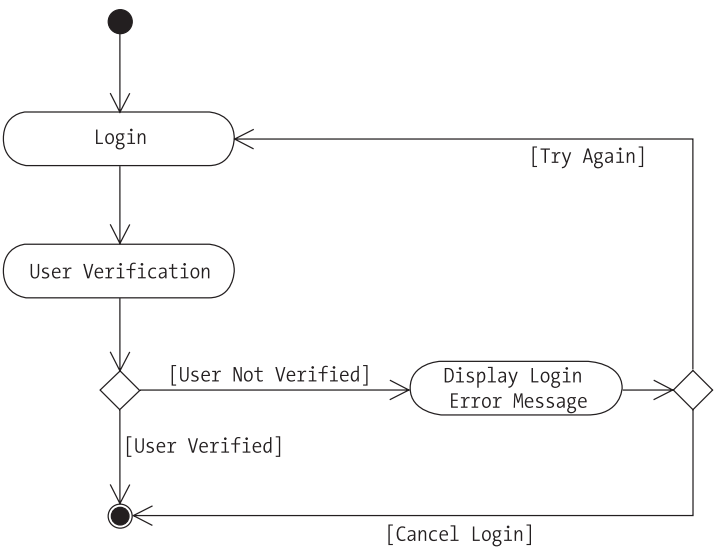

Figure 4-10. *An activity diagram depicting the Login use case scenarios*

After analyzing the process involved in the use case scenarios, you can now turn your attention to assigning the necessary behaviors to the classes of the system. To help identify the class behaviors and interactions that need to occur, you construct an interaction diagram. As discussed in Chapter 3, interaction diagrams can take the form of either a sequence diagram or a collaboration diagram. Sequence diagrams focus on the order of the object interactions taking place, and collaboration diagrams focus on the links occurring between the objects. Figure 4-11 shows a sequence diagram for the Login use case scenarios. The `Purchaser(UI)` class calls the `Login` method that has been assigned to the `Employee` class. The message returns information that will indicate whether the login has been verified.

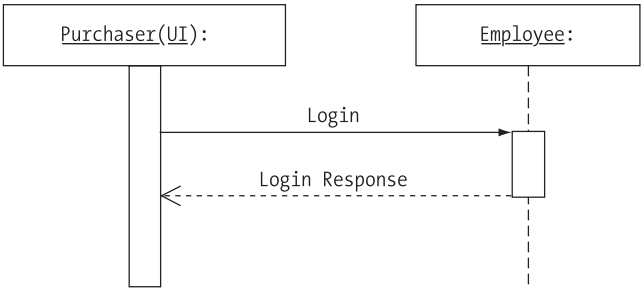

Figure 4-11. *A sequence diagram depicting the Login use case scenarios*

Next, let's analyze the View Supply Catalog use case. The following scenario describes the use case:

1. User logged in and has been verified.
2. User views a catalog table that contains product information, including the supply name, category, description, and price.
3. User chooses to filter the table by category, selects a category, and refreshes the table.

From this scenario, you can see that you need a method of the `ProductCatalog` class that will return a listing of product categories. The `Purchaser` class will invoke this method. Another method the `ProductCatalog` class needs is one that will return a product list filtered by category. The sequence diagram in Figure 4-12 shows the interaction that occurs between the `Purchaser(UI)` class and the `ProductCatalog` class.

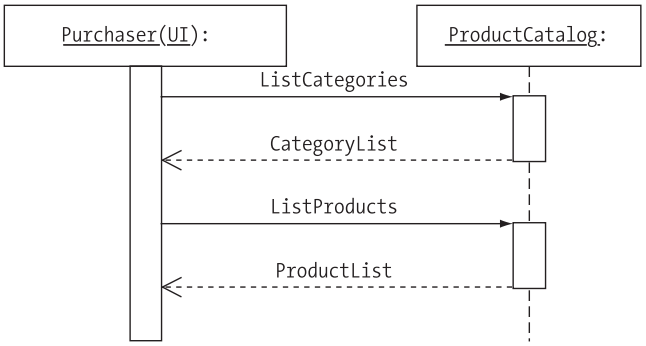

Figure 4-12. *A sequence diagram depicting the View Supply Catalog scenario*

The following scenario was developed for the Purchase Request use case:

1. A purchaser has logged in and has been verified as an employee.
2. The purchaser selects items from the product catalog and adds them to the order request (shopping cart), indicating the number of each item requested.
3. After completing the item selections for the order, the purchaser submits the order.
4. Order request information is updated, and an order ID is generated and returned to the purchaser.

From the scenario, you can identify an `AddItem` method of the `Order` class that needs to be created. This method will accept a product ID and a quantity and then return the subtotal of the order. The `Order` class will need to call a method of the `OrderItem` class, which will create an instance of an order item. You also need a `SubmitOrder` method of the `Order` class that will submit the request and the return order ID of the generated order. Figure 4-13 shows the associated sequence diagram for this scenario.

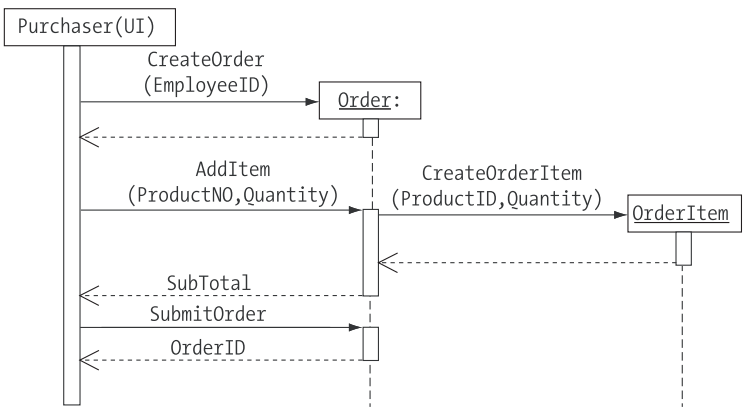

Figure 4-13. *A sequence diagram depicting the Purchase Request scenario*

Some other scenarios that need to be included are deleting an item from the shopping cart, changing the quantity of an item in the cart, and canceling the order process. You will also need to include similar scenarios and create similar methods for the Department Purchase Request use case. After analyzing the scenarios and interactions that need to take place, you can develop a class diagram for the Purchase Request portion of the application, as shown in Figure 4-14.

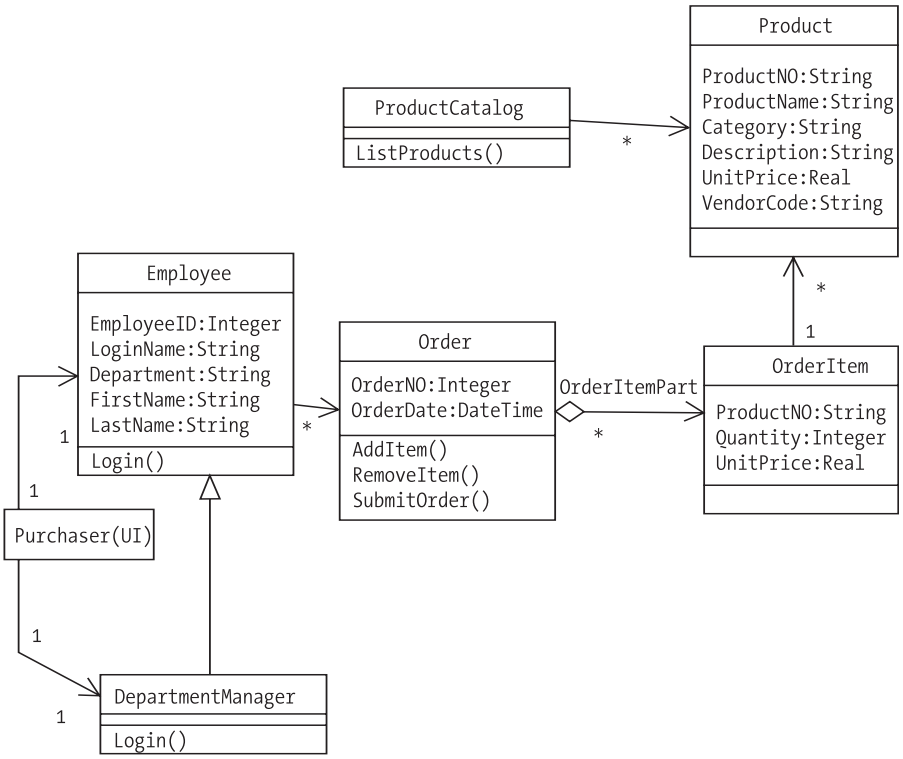

Figure 4-14. *OSO Purchase Request class diagram*

Developing the User Interface Model Design

At this point in the application design process, you do not want to commit to a particular GUI implementation (in other words, a technology-specific one). It is helpful, however, to model some of the common elements and functionality required of a GUI for the application. This will help you create a prototype user interface that you can use to verify the business logic design that has been developed. The users will be able to interact with the prototype and provide feedback and verification of the logical design.

The first prototype screen that you need to implement is the one for logging in. You can construct an activity diagram to help define the activities the user needs to perform when logging in to the system, as shown in Figure 4-15.

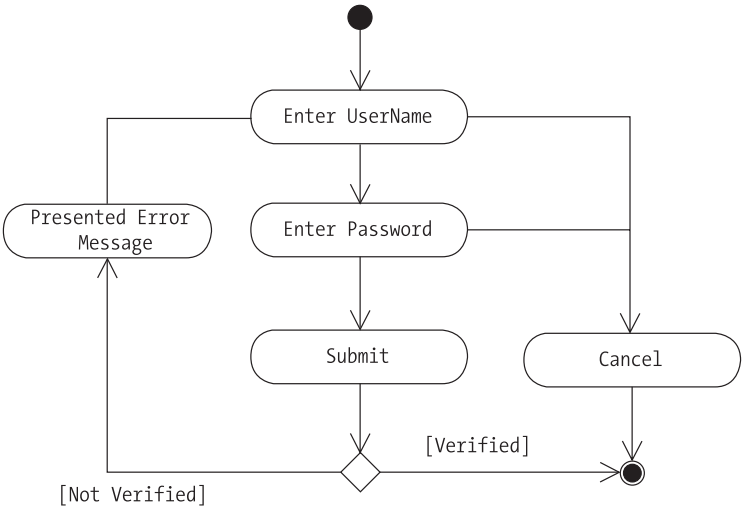

Figure 4-15. *An activity diagram depicting user login activities*

Analyzing the activity diagram reveals that you can implement the login screen as a fairly generic interface. This screen should allow the user to enter a username and password. It should include a way to indicate that the user is logging in as either an employee or a manager. The final requirement is to include a way for the user to abort the login process. Figure 4-16 shows a prototype of the login screen.

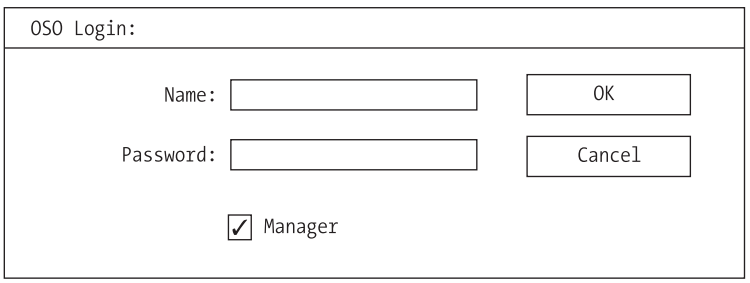

Figure 4-16. *Login screen prototype*

The next screen you need to consider is the product catalog screen. Figure 4-17 depicts the activity diagram for viewing and filtering the products.

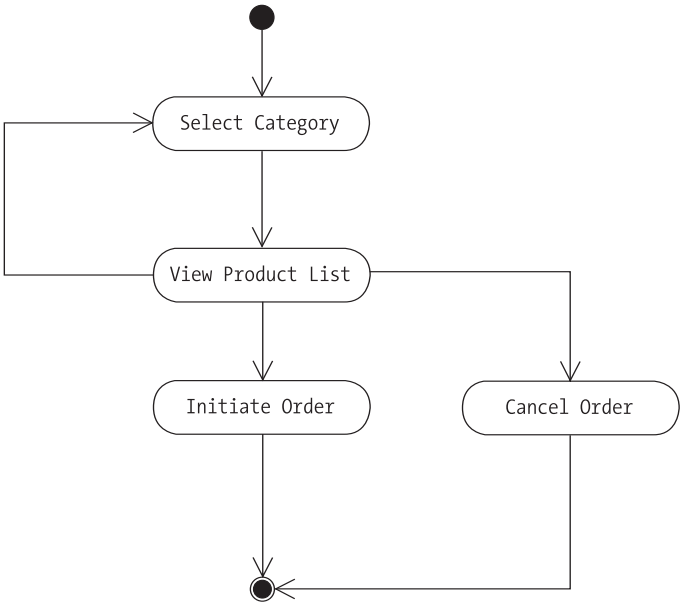

Figure 4-17. *An activity diagram depicting activities for viewing products*

The activity diagram reveals that the screen needs to show a table or list of products and product information. Users must be able to filter the products by category, which can be initiated by selecting a category from a category list. Users also need to be able to initiate an order request or exit the application. Figure 4-18 shows a prototype screen that can be used to view the products.

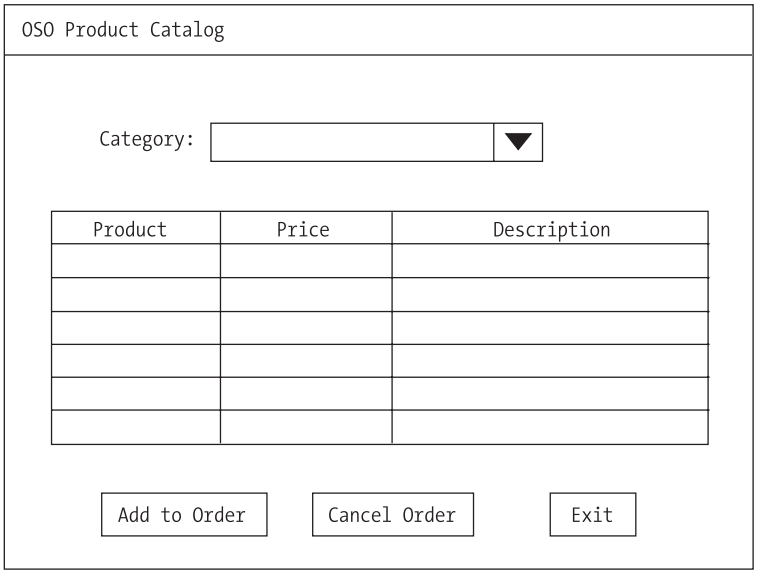

Figure 4-18. *View products screen prototype*

The final screen that needs to be prototyped for this part of the application is the shopping cart interface. This will facilitate the adding and removing items from an order request. It also needs to allow the user to submit the order or abort an order request. Figure 4-19 shows a prototype of the order request screen.

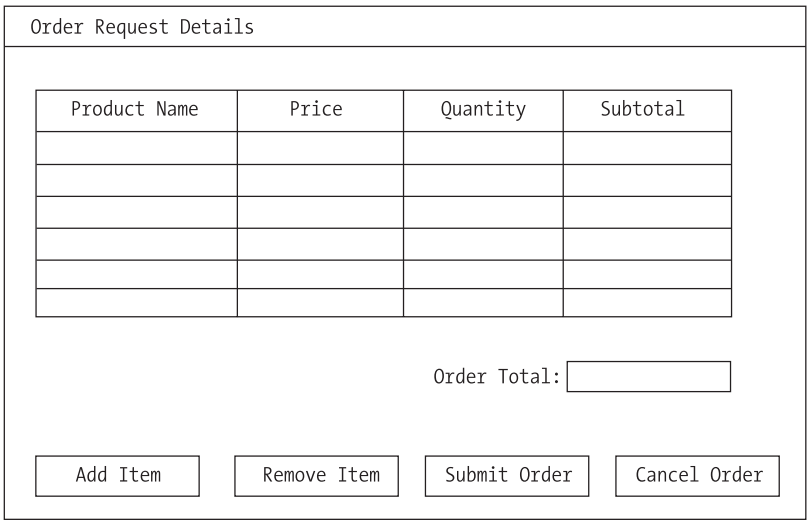

Figure 4-19. *Order request screen prototype*

That completes the preliminary design for this phase of the OSO application. You applied what you learned in Chapters 2 and 3 to model the design. Next, let's review some common mistakes to avoid during this process.

Avoiding Some Common OOP Design Pitfalls

When you start to model your own OOP designs, you want to be sure to follow good practice. The following are some of the common traps that you should avoid:

Confusing modeling with documenting: The main value in modeling is not the diagrams produced, but rather the process you go through to produce the diagrams.

Not involving the users in the process: It is worth emphasizing that users are the consumers of your product. They are the ones who define the business processes and functional requirements of the system.

Trying to model the whole solution at one time: When developing complex systems, break up the system design and development into manageable components. Plan to produce the software in phases. This will provide for faster modeling, developing, testing, and release cycles.

Striving to create a perfect model: No model will be perfect from the start. Successful modelers understand that the modeling process is iterative, and models are continuously updated and revised throughout the application development cycle.

Thinking there is only one true modeling methodology: Just as there are many different equally viable OOP languages, there are many equally valid modeling methodologies for developing software. Choose the one that works best for you and the project at hand.

Reinventing the wheel: Look for patterns and reusability. If you analyze many of the business processes that applications attempt to solve, a consistent set of modeling patterns emerge. Create a repository where you can leverage these existing patterns from project to project and from programmer to programmer.

Letting the data model drive the business logic model: It is generally a bad idea to develop the data model (database structure) first and then build the business logic design on top of it. The solution designer should first ask what business problem needs to be solved and then build a data model to solve the problem.

Confusing the problem domain model with the implementation model: You should develop two distinct but complementary models when designing applications. A *domain model* design describes the scope of the project and the processing involved in implementing the business solutions. This includes what objects will be involved, their properties and behaviors, and how they interact and relate to each other. The domain model should be implementation-agnostic. You should be able to use the same domain model as a basis for several different architecturally specific implementations. In other words, you should be able to take the same domain model and implement it using a Visual Basic rich-client, two-tier architecture or a C# (or Java, for that matter) n-tier distributed web application.

Summary

Now that you have analyzed the domain model of an OOP application, you are ready to transform the design into an actual implementation. The next part of this book will introduce you to the Visual Basic language. You will look at the .NET Framework and see how Visual Basic applications are built on top of the framework. You will be introduced to working in the Visual Studio IDE and become familiar with the syntax of the Visual Basic language. The next section will also demonstrate the process of implementing OOP constructs such as class structures, object instantiation, inheritance, and polymorphism in the Visual Basic .NET language. You will revisit the case study introduced in this chapter in Chapter 10, at which time you will look at transforming the application design into actual implementation code.