December 2003

Making the Transition from OO Analysis to OO Design with the Unified Process

John Satzinger
Southwest Missouri State University

Robert Jackson
Brigham Young University

MAKING THE TRANSITION FROM OO ANALYSIS TO OO DESIGN WITH THE UNIFIED PROCESS

John W. Satzinger
Southwest Missouri State University
jws086f@smsu.edu

Robert B. Jackson
Brigham Young University
rbj2@email.byu.edu

Abstract

The current momentum for OO development in industry makes OO techniques worthy of considerable attention. Consequently, information systems researchers and practitioners are increasingly using constructs such as use cases and class diagrams to define system requirements. A glaring weakness in the literature is the lack of useful guidelines and strategies for taking a relatively high level OO requirements model and translating it into an implementable architecture and detailed OO design. This tutorial demonstrates techniques for bridging the gap between OO requirements models and detailed OO design, drawing on the framework provided by the Unified Process (UP) and based on concepts and techniques developed by researchers working on OO design patterns. The examples provided illustrate the transition from requirements, to architecture, and to detailed design.

Keywords: Object-oriented analysis, object-oriented design, design patterns, unified process (UP), unified modeling language (UML)

Introduction

Most information systems development groups in industry are investigating OO development opportunities as are information systems (IS) academic programs. Most university IS programs are introducing OO concepts and teaching fundamental OO programming techniques. Many are introducing OO analysis and design concepts as part of the traditional analysis and design course. Some completely embrace OO throughout their curriculum and teach nothing but OO.

Although fundamental OO concepts and techniques are being addressed in academic programs, a glaring weakness in the literature is the lack of useful guidelines and strategies for taking a relatively high level OO requirements model and translating it into an implementable architecture and detailed OO design. This tutorial demonstrates techniques for bridging the gap between OO requirements models and detailed OO design. We demonstrate examples of iterations and models as defined in the Unified Process (UP) and draw on concepts and techniques developed by researchers working on OO design patterns. Design techniques and principles of good OO design as discussed by Larman (2002) and others, as well as the original design patterns identified by the "Gang of Four" (Gamma, Helm, Johnson, and Vlissides, 1995), are integrated into the set of information systems examples. The examples illustrate the transition from requirements, to architecture, and to detailed design.

The Unified Process

The Unified Process (UP) is a comprehensive OO system development methodology originally developed by Jacobson, Booch, and Rumbaugh (1999). The UP draws on accepted best practices such as iteration and model-driven development. It is widely accepted as a leading (if not *defacto* standard) OO development methodology. Since the UP includes specific OO models for modeling requirements and designs within an iterative development framework, it remains the best way to illustrate the transition from requirements models to architecture to detail design for OO development.

UP Phases, Iterations, and Disciplines

Like traditional system development lifecycles, the UP includes a set of four sequential phases: Inception, Elaboration, Construction, and Transition. What is different is that the UP abandons the notion that the phases follow the analysis-design-implementation waterfall pattern of the traditional SDLC. Instead, the sequential phases describe the emphasis of the project team and the project activities at any point in time.

Each iteration in each phase involves some mix of system development activities called disciplines. Disciplines include business modeling, requirements, design, implementation, test, deployment, configuration and change management, project management, and managing the development environment. Figure 1 shows the four phases with multiple iterations and the use of all disciplines (Larman, 2002). Note that all disciplines are involved in varying degrees in all iterations and in all of the phases.

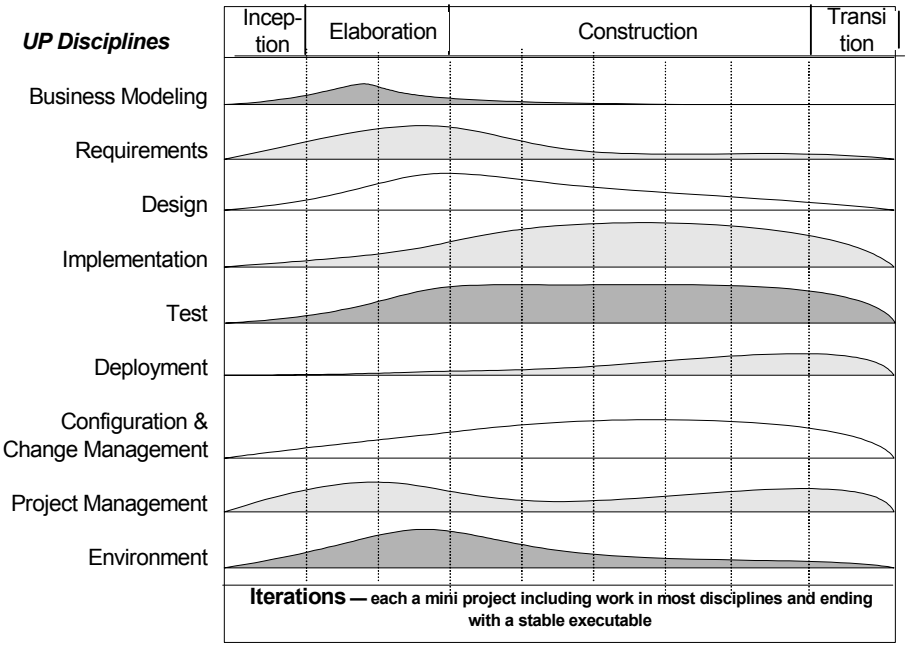

Figure 1. UP Phases, Iterations, and Disciplines

The Elaboration Phase iterations focus more on requirements, more on design, and less on implementation and testing, but some implementation and testing is completed in each iteration. Later, in Construction, some requirements modeling still occurs, but the focus is much more on design, implementation, and testing. The UP model shown in Figure 1 successfully portrays the sequential concepts needed for project management with the iterations required through the project that involve business modeling, requirements, design, implementation, and test disciplines.

OO Analysis Versus OO Design

Understanding the disciplines used throughout the development process is key to understanding the differences between OO analysis and OO design, a major theme of this tutorial. Business modeling and requirements are most often associated with OO analysis, but again, in the UP, these two disciplines are used throughout the project, not just in the initial phases.

Business modeling refers to modeling business processes for the entire enterprise and to modeling domain objects for the specific application. The domain model captures information about what is involved in the users' work – often called problem domain classes – modeled using the Unified Modeling Language (UML) class diagram.

The requirements discipline includes the analysis of functional requirements by identifying and writing use cases and by identifying non-functional requirements. A use case is a story that describes a case where the user interacts with the system to accomplish something of value. Use cases are modeled by starting with lists of use cases, writing brief descriptions of them, writing more detailed descriptions, and showing the collection of use cases graphically on a use case diagram. In addition, activity diagrams can be used to model the details of the user's required interactions with the system, and system sequence diagrams can be used to show the required input and output messages from the user and the system.

OO Design focuses on defining the software classes of objects and modeling how they collaborate to fulfill the requirements defined during OO analysis. The UP design discipline, therefore, shows how use cases are realized through the design of software classes and objects. It is here, during design, that design patterns discussed in the introduction are identified and applied to the problem.

Requirements Modeling with UML

In this section, we introduce the system case study used to illustrate OO analysis and OO design in this tutorial. We also describe the requirements models created for the case study to define what is included in the Inception Phase and in OO analysis.

The SailWorld Case Inception Iteration

The example used is the SailWorld Sailing School. SailWorld conducts sailing courses for beginning and advance sailors who travel to SailWorld locations for weekend or weeklong on-the-water training. Many customers make this trip their annual vacation. SailWorld sites are in desirable tourist resort locations. Many customers want to obtain a certification required by boat charter companies. SailWorld coordinates the certification for customers so they can take and pass a test to obtain the desired certification.

SailWorld wants an information system that provides information to customers about the sailing courses and course offerings and that allows a customer to make a booking. It is assumed the system provides direct Internet access. The system must also allow SailWorld management and employees to maintain information about sailing courses, course offerings, instructors, boats, and certifications.

The Inception Phase of the project provides an initial investigation into the proposed system. The key deliverables include an overview of the vision for the system, the business benefits, rough estimates of the cost, preliminary schedule, risk assessment, preliminary prototypes, and the list of key use cases that represent the scope. The Inception Phase should be brief, and key models might be started but not completed in any detail. The use cases initially listed and partially described include Maintain customer information, Maintain sail course information, Create course offering, Maintain registrations and bookings, Maintain information about boats, and Maintain information about certification exams.

Elaboration Phase Iteration 1

Once the Inception Phase is complete, the project moves on to a set of Elaboration Phase iterations. The first iteration would address the business modeling and requirements for a subset of the system, chosen based on the preliminary schedule plan and risk assessment plan. But we want to re-emphasize that the Elaboration Phase does not only involve requirements and OO analysis. The initial iteration will also complete much of the design, implementation, and testing for this subset of requirements.

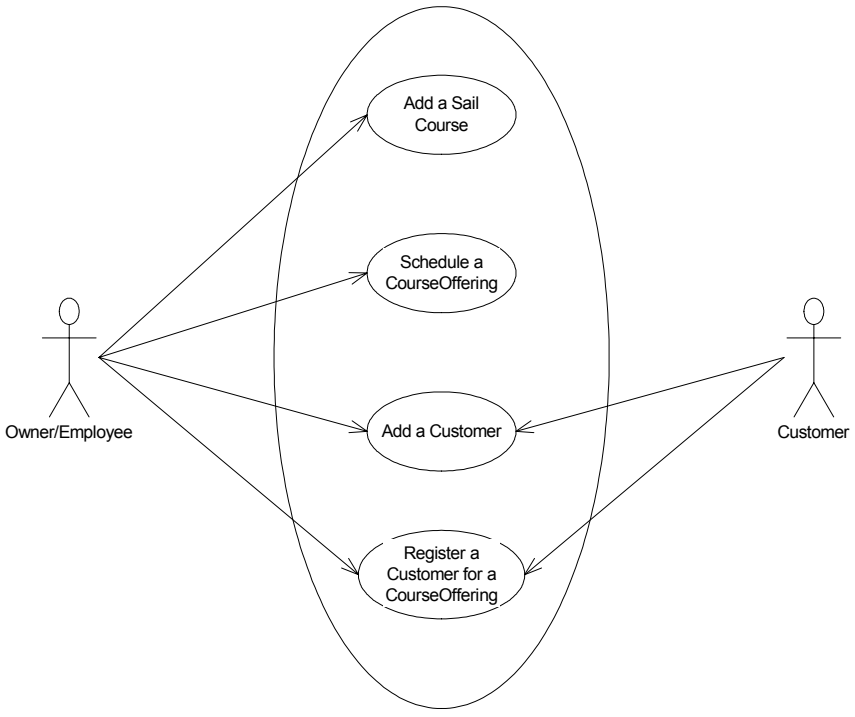

Figure 2. Elaboration Iteration 1: Use Case Model

The use case diagram shown in Figure 2 highlights the four use cases to be realized in the first iteration. Two actors are involved, owner/employee and customer, who would interact with the system via the Internet. Notice that the use cases and the domain model for this iteration are limited as compared to the entire SailWorld requirements.

The domain model for the first iteration is shown in Figure 3, a UML class diagram with four classes involved in the use cases for the iteration. Attributes and association relationships are shown (including an association class). Methods are not shown. Still focusing on requirements, each use case can be modeled using an activity diagram that highlights the activities completed by the actor and by the system.

A final diagram used to model a use case is a variation of the UML sequence diagram, called a system sequence diagram. The system sequence diagram models requirements rather than design details because it is limited to the actor and one object that represents the software system under construction. The details of the software system are not addressed.

Figure 4 shows the system sequence diagram for the use case Schedule Course Offering. The actor sends a message to the system requesting information on a SailCourse, and the system returns course information. The actor then sends a message to the system requesting that a CourseOffering be added for the course, and the system returns conformation information.

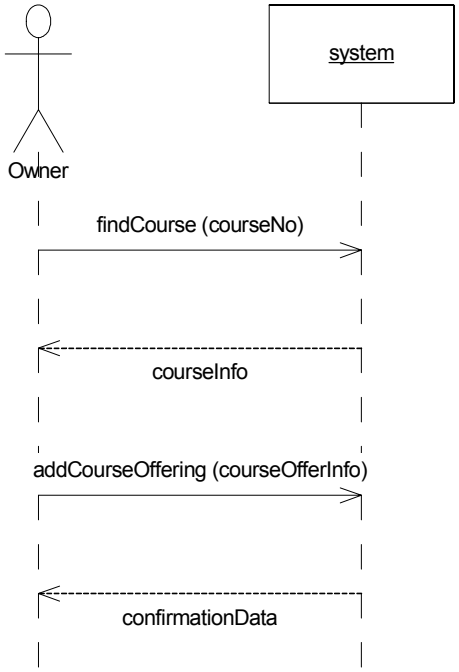

Figure 4. System Sequence Diagram for Use Case Schedule Course Offering

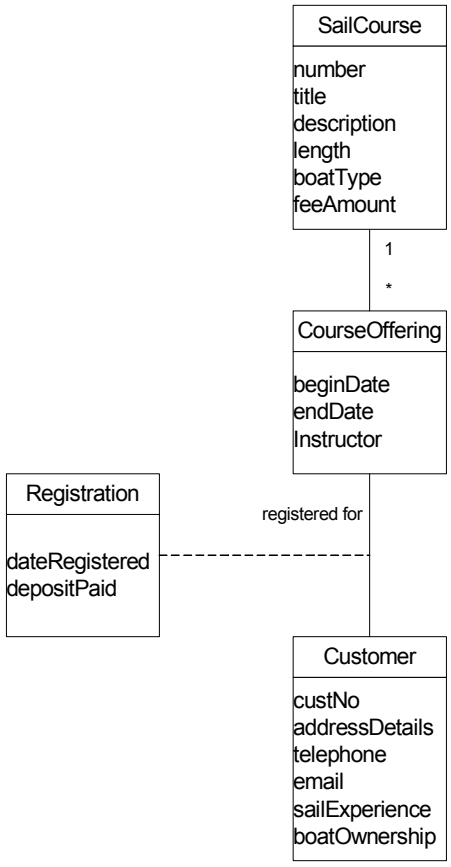

Figure 3. Elaboration Iteration 1: Domain Model

The class diagram (domain model), use case diagram, use case descriptions, activity diagrams, and system sequence diagrams complete the requirements modeling for the first iteration. Still in this first Elaboration Iteration, the UP design discipline will now extend the requirements models into more detailed design models, illustrating the transition from OO analysis to OO design.

OO Design Models with UML

Introduction — Why Do Design?

As we saw in the last section, three primary models are used to capture and document the user requirements: the domain model, the use case diagram and use case descriptions, and the system sequence diagrams. The other diagram mentioned is the activity diagram, which is used to help understand the business processes, but which is not directly used for system design.

The purpose of system design is to bridge that gap between the requirements models and the program code and database. Unfortunately, many information systems OO courses do not teach object-oriented systems design to the depth required for new information systems graduates to become proficient. Many information systems programs offer two or three programming classes and a fairly rigorous systems analysis course. Since little instruction is provided on how to do design, many new developers simply jump into OO programming after a brief effort at defining user requirements.

Architectural Layers

As discussed earlier, best practices are reflected in design patterns. The first design pattern to recognize is the N-Layer architecture that separates the user interface (UI) or view layer, the problem domain classes, and the data access classes and other technical services. This architecture is often referred to as three-tier design or as three-layer design. To the extent that design specifies the use case realization through software, three-layer design provides a framework for adding the user interface design details and the data access and database design details to the problem domain classes first modeled as requirements.

The Design Models

One of the main benefits of the object-oriented approach is that the design models are simply extensions of the analysis models. In object-oriented design, the primary models are (1) Design Class Diagrams and (2) Detailed Interaction Diagrams. A Design Class Diagram (DCD) (Figure 5) is an extension of the Domain Model (Class Diagram) developed during analysis. A Detailed Interaction Diagram is an extension of the System Sequence Diagram, also developed during analysis. Other models are used, but these two are the primary design components.

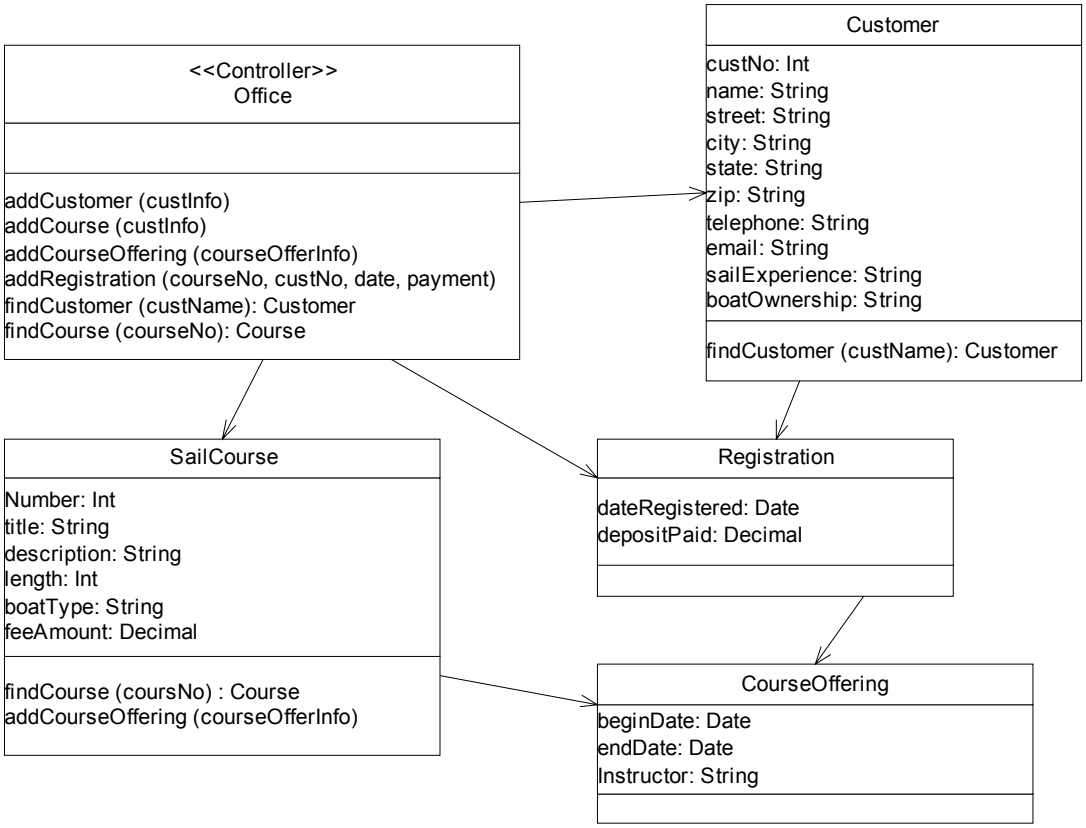

Figure 5. SailWorld Problem Domain Design Class Diagram (DCD)

Design Class Diagram

Notice that the SailWorld Design Class Diagram is very similar to the domain model developed during business modeling discipline activities. We will discuss three major additions. First, the attributes are defined more precisely by the addition of type information, and in some cases visibility information. We also identify class variables (shared in VB .NET and static in Java) with underlining. Second, method signatures are added. Method signatures include visibility, method name, parameter types, and return types. Third, navigation arrows are added. Navigation arrows indicate visibility from one class to another, meaning an object of one class is aware of and can send a message to an object of the other class.

The Design Class Diagram serves several purposes. First, it functions as a quality review document to ensure that the design is robust and complete. The other major objective of the DCD, of course, is to provide the basis for programming activities. As can be seen in Figure 5, every programming class is identified along with its typed attributes and method signatures.

Interaction Diagram

An interaction diagram documents the collaborative work done by several software objects to execute a singe use case (or even only a portion of a use case called a scenario). An interaction diagram for a use case identifies all of the objects that must "interact" or "collaborate" together to execute the system functions necessary for that use case or scenario. In UML the interactions are identified as "messages" between the collaborating objects. Obviously, the process of developing interaction diagrams is the foundation of OO system design. Two types of Interaction Diagrams are used for system design, (1) Sequence Diagrams and (2) Collaboration Diagrams. Both types of diagrams present information that is essentially the same, but from different views.

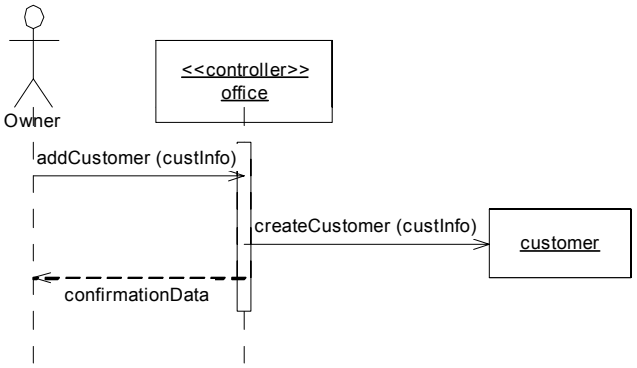

Figure 6. Sequence Diagram for Use Case *Create a Customer*

Figures 6 and 7 present examples of a sequence diagram and a collaboration diagram for the use case Create a Customer. Both diagrams contain the same messages. The message syntax is quite similar to method syntax in a programming language. In fact, the destination object for each message is required to have a method to handle the arrival of that message. The development of the messages on the interaction diagrams is the same process as defining the methods in the objects. Comparing the DCD in Figure 5 and the messages in Figure 6, you will note that there is a message *addCustomer(custInfo)* in the sequence diagram, and there is a corresponding method *addCustomer (custInfo)* in the Office class in the DCD.

The Design Process

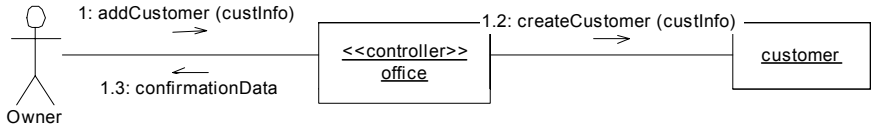

Figure 7. Collaboration Diagram for Use Case *Create a Customer*

System design is begun by selecting an individual use case and identifying the collaborating classes and messages required for that use case. The use case description and system sequence diagram created for requirements provides the foundation. System design is carried out use case by use case.

Another important point is to remember that design is also an iterative process. The UP iterations are macro-level iterations. Within each UP iteration, it is often beneficial to carry out design via micro-level iterations. One might first develop the detailed sequence diagram for the use case and only include domain model classes. Then additional micro-level iterations are done to add user interface view layer objects (e.g. windows) and data access layer objects, as discussed above for three layer design. During the design process we continually apply principles of good design and design patterns.

Micro-Level Iteration 1

We will explain the method of system design by presenting a simple design case. The first step is to select a use case to design. For this example, we will continue to work on a use case of intermediate complexity namely Schedule Course Offering. Our next step is to develop a detailed sequence diagram for this use case. Inputs to this process are the domain model and the system sequence diagram. As we look at the domain model, it appears that the domain classes that might be impacted are the SailCourse and the CourseOffering. At this point, we will also add a new object, one that is used to represent the system as a whole. (Later, we will learn about patterns and why this is a good design practice). This additional object will be called office. It will serve as a kind of switchboard to distribute messages that come from external points, called a controller (Figure 8).

The next step is to add the messages that were identified on the system sequence diagram (SSD) for this use case. The messages from the SSD are part of the user requirements and denote those tasks and data entry points that are initiated by the actor. The next step is to analyze each of these input messages and extend out the necessary internal messages required to complete that interaction. Figure 9 illustrates the completed sequence diagram for this use case for micro-level iteration 1 (without user interface or data access).

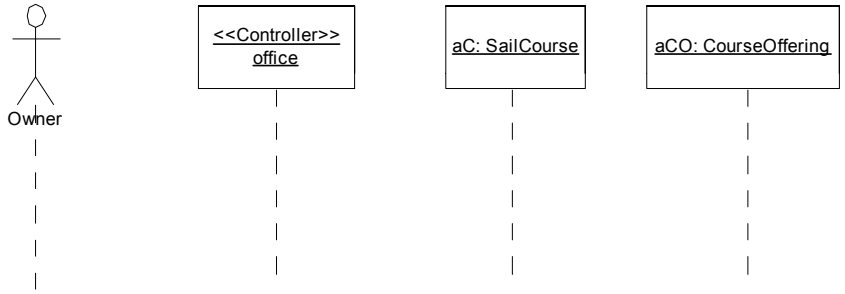

Figure 8. Preliminary Sequence Diagram Showing Actors, Objects and Lifelines

The findCourse message needs to be sent to some internal source of courses, find the right one, and return a reference to it. We illustrate this process by showing a multi-object called "aC:SailCourseS." The double boxes indicate that this is an internal array or set of courses, which were named sailcourseS. The "aC" indicates that a specific object, named "aC' is what is found based on the input parameter courseNo. At this point, we do not concern ourselves with how this internal array was populated. The message format now includes a return value to illustrate that the found course, "aC" is returned to the office object.

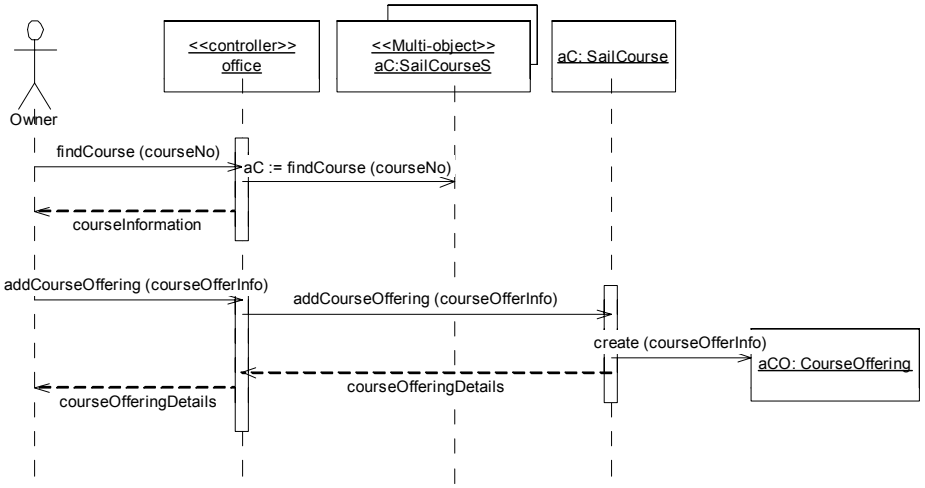

Figure 9. Schedule Course Offering Sequence Diagram

Finally, we extend the input message "addCourseOffering(courseOfferInfo)." This message originates from the owner actor and is sent into the system via the office object. During design, we are always asking questions such as who should be the creator of other objects and who should be the expert that should know about the other objects. In this instance, we decide that the course object should maintain control of the course offering objects. We base this decision partly on the domain model, and partly on some good design principles that will be discussed later. We add the return messages to indicate that the information is now also displayed back to the actor.

At this point in the design, we have a detailed sequence diagram for a simple version of the Schedule Course Offering use case. This diagram shows the domain classes that must be involved in the execution of the use case. We also recognize, however, that later micro iterations will be needed to add such things as windows objects and data access objects. We also did not include any error handling or exception conditions. Later iterations, either micro-level or perhaps even a complete UP iteration, will add messages to handle those complexities.

Once the detailed sequence diagram is complete, we elaborate the Design Class Diagram for the domain classes. Each domain class in the sequence diagram that contains a message destination must provide a method to handle that method. Looking at the sequence diagram, we identify two methods for the Office class, namely findCourse and addCourseOffering. For SailCourse we identify a method called addCourseOffering. Figure 10 illustrates the partially complete versions of the DCD classes Store and Course.

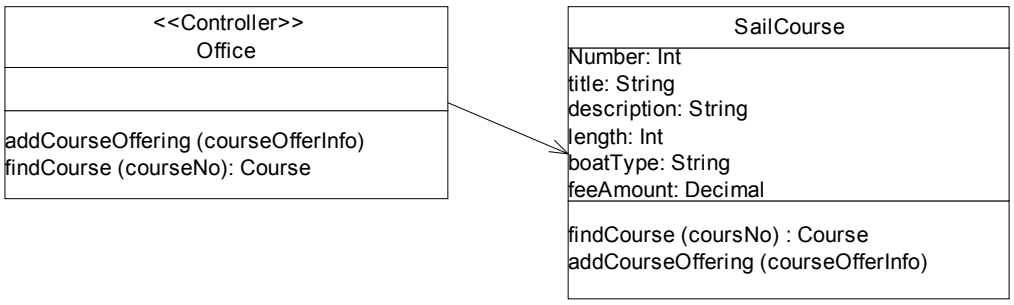

Figure 10. Partial Design Class Diagram derived from Sequence Diagram for Schedule Course Offering

This same process is done for every use case that was chosen to be part of the first UP elaboration iteration. The end result of this activity is a set of interaction diagrams and design classes that specify the interactions, methods, and responsibilities of the collaborating classes to carry out the defined use cases.

Micro-Level Iteration 2

In the previous example, we focused primarily on the problem domain model classes such as Customer and SailCourse. However, a system does not consist only of problem domain classes. As shown in Figure 8, one of the common design patterns used for today's systems is a three-layer design pattern (sometimes called model-view-data design pattern). Doing design with the problem domain classes is an important step in understanding the responsibilities of those classes; however, a complete design must also include classes in the user interface view layer and data layer, and the interactions among the three layers.

In Figure 11, we take the Schedule Course Offering use case and elaborate it by adding view layer and data access layer classes. In the view layer we add two classes, a MainWindow class and a ScheduleWindow class. The MainWindow simply gets the process started by creating a ScheduleWindow. The detail interactions from the Owner go through the ScheduleWindow. All messages from the view layer go through the controller class named office. It provides a single control point between the view layer and the model layer. In other words, in this iteration we add the user interface objects between the actor and the controller.

At this point in the design process, the solution for the Schedule Course Offering is essentially complete. Additional iterations would add any exception processes and error messages. As before, we can now transfer this information to the design class diagram. Figure 12 is an example of the design class diagram based on this sequence diagram. Note that each layer is identified as a separate UML package. In essence we identified three separate design class diagrams, one for each layer.

Applying Principles of Good OO Design

As indicated earlier, during the design process we also apply principles of good object-oriented design. In this tutorial we only have space to identify a few of the most important fundamental principles. These fundamentals should be understood by developers and also should be taught to students learning how to do OO design.

One major area of design is concerned with the identification and design of individual classes. As seen in the three-layer design, classes are designed to be self-contained units with data and methods that are focused and cohesive. A class with high cohesion is focused and the attributes and methods of that class are tightly related and contribute to a single focused function. Classes that are not cohesive are hard to comprehend and difficult to maintain.

A related design issue is the coupling between the classes in the system. A well-designed system uses limited and controlled coupling. Coupling between classes occurs when one class is visible y to another class and accesses the methods or attributes of that class. The previous two principles influence another principle identified in Larman (2002) as "protected variations." Protected variation says that the classes in the system should be designed so that they are protected as much as possible from variations, or from things that are subject to change.

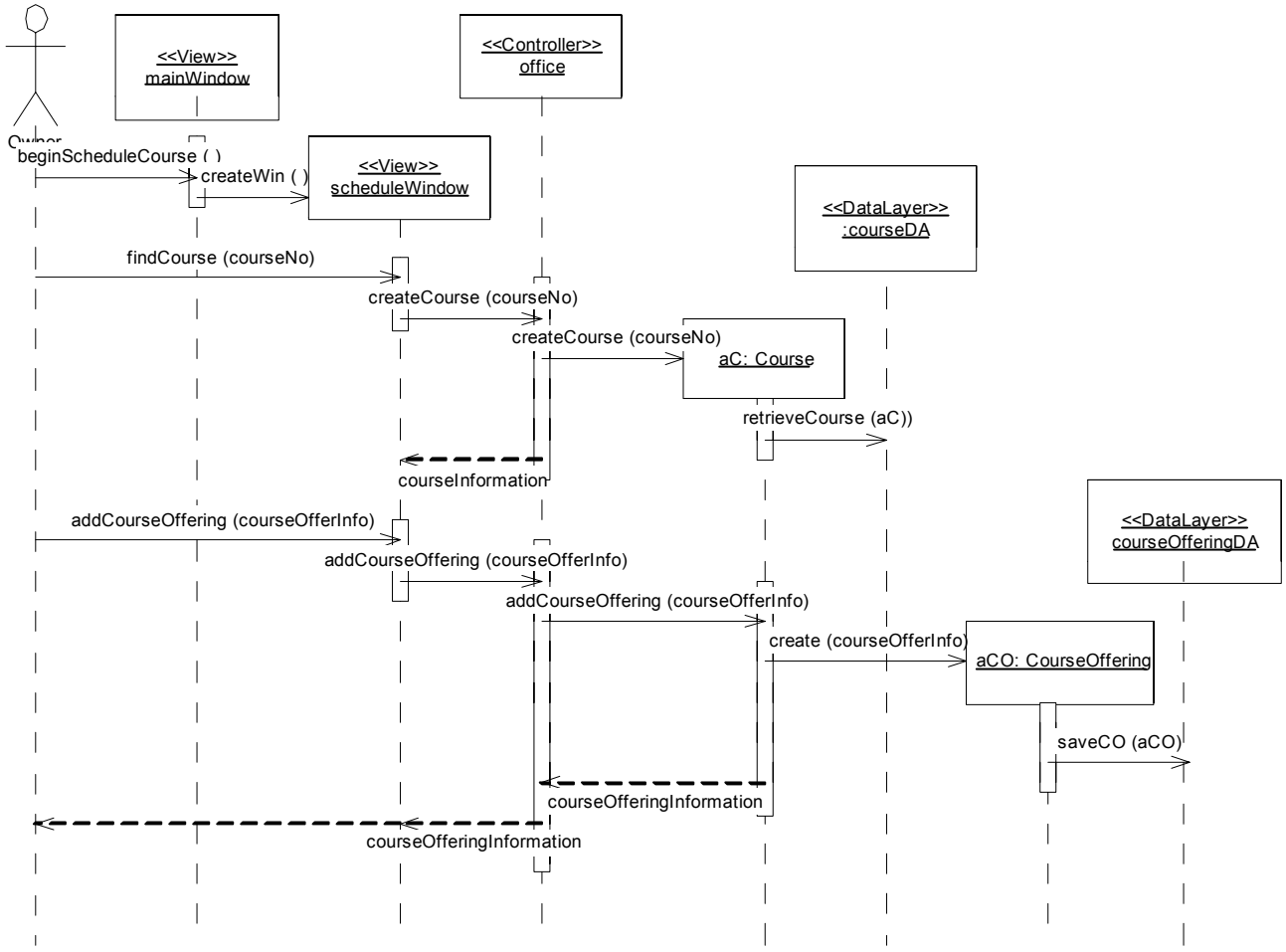

Figure 11. Three-Layer Sequence Diagram for Schedule Course Offering

Another important principle is assigning responsibilities to classes. These responsibilities include creating other classes, to access information, to provide visibility to other objects, and to accumulate information. In Figure 11, we see an example of creation responsibility. The question is asked, "Which class should be responsible for instantiation of new CourseOffering objects?" In this situation, since CourseOffering objects are dependent on the existence of a Course object and are subservient to it, then the Course class is responsible for creating the CourseOffering objects. Another question from Figure 11 might be, "Who should be responsible for retrieving all the information to fill in the fields for a new Course object?" The Course object is responsible to invoke the data access class to finish filling in the required information.

Many more basic design principles can be applied as the interaction diagrams and design classes are developed. It should be noted, that if developers go directly to code without design, these issues are rarely considered.

Design Patterns

System developers that employ good design principles will also be familiar with many standard design patterns. Design patterns exist for almost every facet of object-oriented design. There are many different ways to categorize the various design patterns such as structural, behavioral, architectural, and creational. Some design patterns are better used for specific languages and not for other languages. Other patterns apply to various platforms such as J2EE or .NET; some apply to local systems while others are for distributed systems.

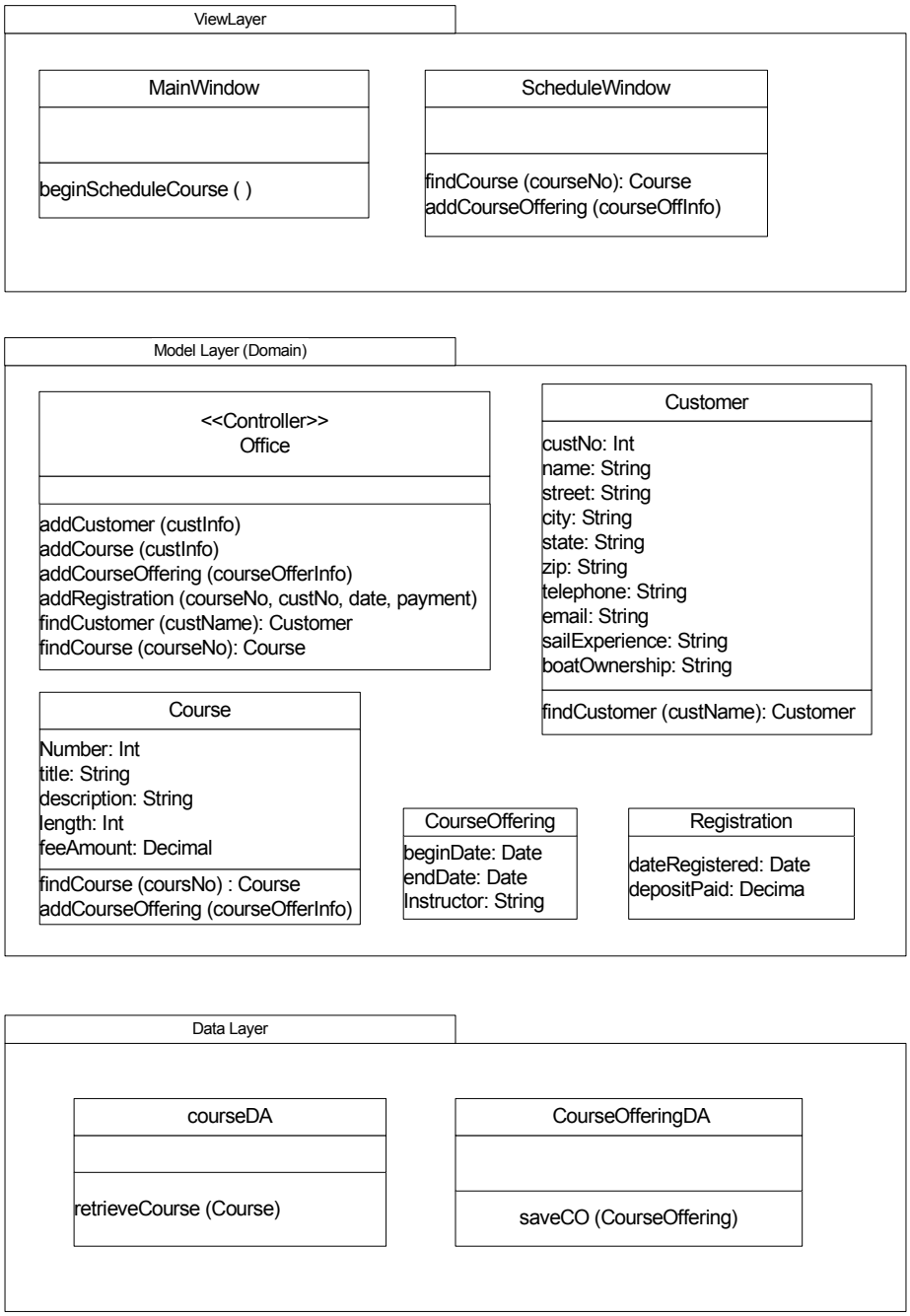

Figure 12. Design Classes by Package for Three-Layer Design

A few basic patterns that might be considered for those who are just learning object-oriented design include: Controller (Façade), Adapter, Factory, Singleton, Strategy, and Proxy. For a better understanding of the Windows event model an explanation of Publisher/Subscriber (a.k.a. Listener or Observer) should be taught. For understanding multi-tasking the Producer/Consumer pattern is important. As we already saw, multi-layer design pattern is an important and critical component of good object-oriented design.

In the examples provided earlier, we saw the use of the controller pattern represented by the Office class. The controller pattern is a type of façade pattern. A façade is a class that provides a front for another subsystem. In this instance, it reduces coupling

between the view layer and the model layer. The adapter pattern is useful for connecting a system or a class to another class with an API that is different than expected. For those of you who travel internationally, the adapter pattern works just like the adapters we use for electrical outlets. The factory pattern is another one that can be used effectively. One example of a factory class is a class that is responsible for instantiating the data access "utility" objects. The factory also becomes an example of the singleton pattern. Being a singleton means that there is only one instance of the class, and the class itself is responsible for ensuring that only one is ever created.

As new developers become familiar with the simple patterns, they gain a deeper understanding of what "good" object-oriented design means. Learning and using design patterns enhances and deepens one's ability to understand object-oriented design. Of course, it also creates better system solutions. Exposure to design patterns should be an integral part of any object-oriented developer's education.

Conclusions

With the widespread acceptance of object-oriented programming languages such as Java, VB .NET, C# .NET, and C+, the great majority of new systems being developed are written in these object-oriented languages. However, even though most educators and practitioners learned how to program with these object-oriented languages, substantial deficiencies exist in the abilities of both practitioners and educators in object-oriented systems design.

We find that some fundamental principles of object-oriented analysis, such as domain modeling and use case modeling, are fairly well understood and are being used in industry and being taught in the classroom. Likewise, the principles of object-oriented programming are being taught. However, a large gap remains in two major areas: (1) the processes, techniques, and artifacts (e.g. models) that bridge the gap from business modeling to programming, and (2) the total development process for building systems using the object-oriented paradigm.

In this tutorial paper, we addressed both these issues. It should be evident that design activities, including use case realizations, applying good design principles, identifying typical design problems and applying standard solutions (e.g. design patterns), are not only an important aspect of systems development, but they are also distinct from OO analysis and from programming. One conclusion that educators should seriously consider is whether a curriculum with only one course that covers both analysis and design can adequately teach the important issues related to design. In addition, students are sufficiently challenged just learning basic programming concepts that they would find it very difficult to grasp the more abstract concepts related to design and patterns. Hence, it is again concluded that a separate focus on design issues needs to be provided someplace in the curriculum.

References

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*, Reading, MA: Addison-Wesley, 1995.

Jacobson, I., Booch, G., and Rumbaugh, L. *The Unified Software Development Process*, Upper Saddle River, NJ: Addison-Wesley, 1999.

Jacobson, C., and Jonsson, O. *Object-Oriented Software Engineering: A Use Case Driven Approach*, Reading, MA: Addison-Wesley, 1992.

Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process* (2nd Ed), Upper Saddle River, NJ: Prentice-Hall, 2002.