Communications of the Association for Information Systems

Volume 12 Article 45

December 2003

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

Recommended Citation
Satzinger, John W. and Jackson, Robert B. (2003) "Making the Transition from OO Analysis to OO Design with the Unified Process," *Communications of the Association for Information Systems*: Vol. 12 , Article 45.
DOI: 10.17705/1CAIS.01245
Available at: https://aisel.aisnet.org/cais/vol12/iss1/45

Communications of the Association for Information Systems

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

ABSTRACT

The current momentum for object oriented (OO) development in industry makes OO techniques worthy of attention. 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 paper 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, to detailed design, and on to program code for one UP iteration.

Keywords: object-oriented analysis (OOA), object-oriented design (OOD), design patterns, Unified Process (UP), Unified Modeling Language (UML).

I. INTRODUCTION

The Java development environment and the newer Microsoft .NET platform focused considerable attention on object-oriented techniques and methodologies. Most information systems development groups in industry are investigating OO development opportunities. Information systems (IS) academic programs are also addressing OO development in various ways. Most university IS programs introduce OO concepts and teach 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.

Information systems instructors are now mostly familiar with OO concepts and fundamentals. Use cases, originally from Jacobson et al. [1992], although not strictly object-oriented, are increasingly being used to define functional requirements even in fairly traditional IS academic programs. Class diagrams are introduced as an alternative to entity-relationship models in database courses and to illustrate key OO concepts in introductory OO analysis lectures. Many additional system development approaches (that are not strictly object-oriented) are also being embraced by instructors. These approaches include the spiral model and the concept of risk [Boehm, 1988], rapid development [McConnell, 1996], eXtreme Programming (XP) [Beck, 2000], and agile modeling [Ambler, 2002]. Therefore, many new concepts and techniques are finding their way into system development courses, including OO concepts and fundamentals.

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. Most current system development books stop short of addressing this transition. Programming language books tend to focus on low-level design details without placing them in the broader context of an overall architecture or use case realization.

This paper 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, to detailed design, and on to program code in one UP iteration. Readers should gain a clearer understanding of how the OO approach can be used effectively from the beginning of the project through to the final implementation.

II. THE UNIFIED PROCESS

The Unified Process is a comprehensive OO system development methodology originally developed by Jacobson, Booch, and Rumbaugh [1999]. The UP draws on accepted best practices such as risk mitigation, iteration, and model-driven development and is now widely accepted as a leading (if not *defacto* standard) OO development methodology. The focus on risk and iteration is grounded in the spiral model developed by Barry Boehm [1988]. The spiral model changes the emphasis on the development project from a linear, waterfall process to a non-linear spiral process. Project requirements posing the greatest risk are addressed in the first iterations.

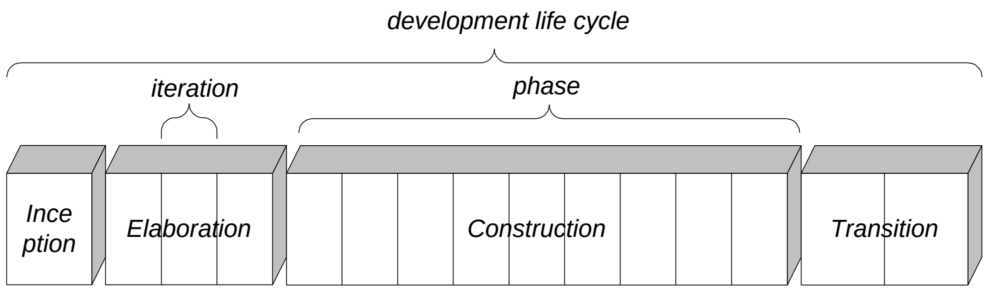
The model-driven focus of the UP follows from work on the Unified Modeling Language (UML) by Booch, Rumbaugh, and Jacobson [1999]. Since the UP includes specific OO models for modeling requirements and designs within an iterative development framework, it remains the best way to illustrate the transition from requirements models to architecture to detail design for OO development. The models, patterns, and design heuristics discussed in this tutorial do not specifically require the use of the UP, however.

UP PHASES, ITERATIONS, and DISCIPLINES

One of the innovations of the UP is its approach to an iterative lifecycle model. Many information systems professionals find the UP lifecycle model confusing at first because in some ways it resembles the spiral model and in other ways it resembles the tradition waterfall SDLC. Like traditional system development lifecycles, the UP includes a set of four sequential phases, but with new names:

- Inception
- Elaboration
- Construction
- 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, each phase includes one or more iterations as shown in Figure 1 [Larman 2002].



Phases are NOT analysis, design, and implement; instead, each
Iteration involves a complete cycle of requirements, design,
implementation, and test disciplines

Figure 1. UP Phases and Iterations (adapted from Larman, 2002)

After the Inception phase, which is much like the traditional SDLC project planning phase, each iteration involves defining requirements, design, and implementation. For example, the Elaboration phase in Figure 1 shows three iterations. The first iteration might involve implementing some of the core functionality of the system, perhaps one or two key use cases. The second iteration might implement the functionality of a few additional key use cases. To complete an iteration, the developers need to define some of the requirements in detail (analysis), design the software, and implement some code.

Most instructors who initially teach the traditional waterfall SDLC in their courses now also introduce iterative development techniques. It is always problematic to show a waterfall SDLC with sequential analysis-design-implementation *phases* while explaining that developers in practice actually complete a series of iterations, each including analysis-design-implementation *activities* so that design and implementation work is completed in violation of the waterfall SDLC. Students are left wondering how to actually apply the waterfall SDLC model for project management, if at all.

The UP lifecycle model solves this mismatch between sequential phases and iterative development. The sequential phases of the UP describe the emphasis or objectives of the project team and the project activities at any point in time. Therefore, the four phases do provide a project management framework to use to plan and track the project over time. It is possible to plan the project so that the Construction phase will begin on a specific date, for example. From a theoretical perspective, however, the division between each phase is weakly defined compared to the division between each waterfall SDLC phase.

The emphasis or objectives of the project team in each of the four phases is described briefly as follows [Larman, 2002]:

Inception – Like any project planning phase, develop an approximate vision of the system, make the business case, define the scope, and produce rough estimates for cost and schedule.

Elaboration – Refine the vision, identify and describe all requirements, finalize the scope, implement the core architecture and functionality, resolve high risks, and produce realistic estimates for cost and schedule.

Construction – Iteratively implement the remaining lower risk, predictable, and easier elements, and prepare for deployment.

Transition – Complete the beta test and deployment.

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 2 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.

The Elaboration phase iterations focus more on requirements, some on design, and less on implementation and testing, but some design, implementation, and testing is completed in each iteration. Later in the Construction phase, some requirements modeling still occurs, but there is much more focus on design, implementation, and testing. The UP model shown in Figure 2 successfully

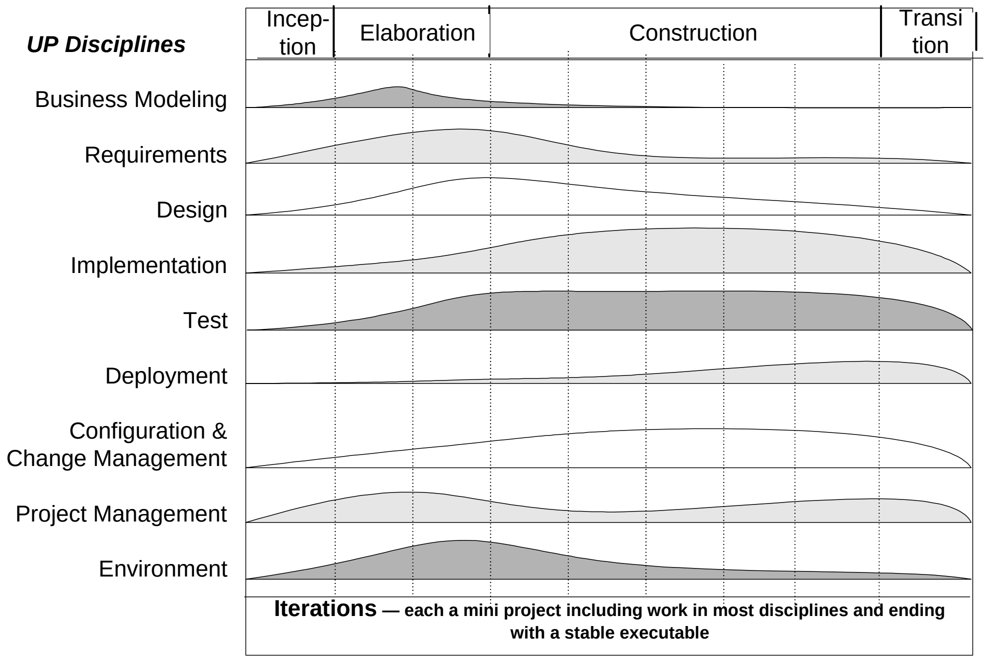


Figure 2. UP Phases, Iterations, and Disciplines

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. The activities in each iteration are consistent with the steps in the spiral model that result in a working prototype after each iteration.

OO ANALYSIS VERSUS OO DESIGN

Understanding the UP disciplines used throughout the development process is key to understanding the differences between what is often referred to as OO analysis and OO design, a major theme of this paper. 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 the types of things involved in the users' work – often called problem domain classes – modeled using the Unified Modeling Language (UML) class diagram. At this level, the classes do not represent software classes; rather, they are representations of real world concepts of importance to the users. Classes, key attributes, association relationships, aggregation, and inheritance can be shown in the domain model. Since business modeling does not consider the classes to be software classes, processing responsibilities are not yet assigned. Therefore, methods of classes are not considered part of the domain model.

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. Often, a use case is defined as a “goal” to be accomplished by the user with the system. 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. Additionally, 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. Examples of these two UML diagrams are shown in the next section. Note that the system sequence diagram is not as detailed as the sequence diagrams used for detailed design. In summary, OO analysis produces a domain model of objects and a use case model of functional requirements using the class diagram, use case descriptions, activity diagrams, and system sequence diagrams.

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. This process starts by defining a high level architecture of three of more tiers, or layers, of software components. Next, sequence diagrams are created by expanding the initial system sequence diagrams to reveal the design of object responsibilities and messaging in each use case. Design class diagrams add more detail about the software classes based on the responsibilities defined in the use cases. Design, therefore, focuses on software components and how use cases are realized in the physical system.

III. REQUIREMENTS MODELING WITH UML

In this section, we introduce the system case study used to illustrate OO analysis and OO design in this paper. The models shown are requirements models created during an initial Elaboration Phase iteration. Recall that the requirements discipline involves creating requirements models.

THE SAILWORLD CASE

The example used is based on the SailWorld Sailing School case study developed by the authors for a forthcoming text on OO analysis and design. 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, because 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 ITERATION

The Inception Phase of the project provides an initial investigation into the proposed system. Figure 3 lists key deliverables. The Inception Phase should be brief, and key models might be started but not completed in any detail. Some prototypes might be completed for proof of concept.

Inception Iteration

Business Case:

High level vision of the system
Business benefits
Rough estimate of the costs to develop
Preliminary schedule/iterations planned
Specific deliverables planned

Use Cases to Include:

List of all use cases
Descriptions of some of them

Prototypes:

Proof of concept prototype
Vision/scoping prototype
Development environment prototype

Potential Project Risks:

Risk assessment plan

Figure 3. Deliverables of SailWorld Project Inception Iteration

Most use cases should be identified, but only a few are described to an intermediate level of detail. The use cases for the complete system include the following:

- Add customer
- Add sail course
- Schedule course offering
- Register customer for course offeringRegister for certification exam
- Book lodging
- Process final payment
- Check in customer
- Cancel customer registration
- Process certification
- Sell merchandise
- Staff course offering
- Assign boat
- Maintain boats
- Maintain staff
- Maintain courses
- Maintain course offerings
- Maintain customers

- Maintain merchandise

ELABORATION 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.

The use case diagram shown in Figure 4 highlights the use cases to be realized in the first iteration. In this iteration, only four use cases are included. 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 compared to the entire SailWorld requirements. We limit the functionality addressed for two reasons. First and most importantly is that in the UP the requirements set "grows" via the iterations of the elaboration phase. In the UP, each iteration of the Elaboration phase should be limited to an amount of work that can be accomplished in a 4 to 8 week period. The set of use cases for the early iterations should be those core use cases. Also, any "high risk" use cases should be included in the early iterations so that the risks can be evaluated early in the project. The second reason is that we limited the use cases and domain model for purposes of this paper. Note with use cases, automation

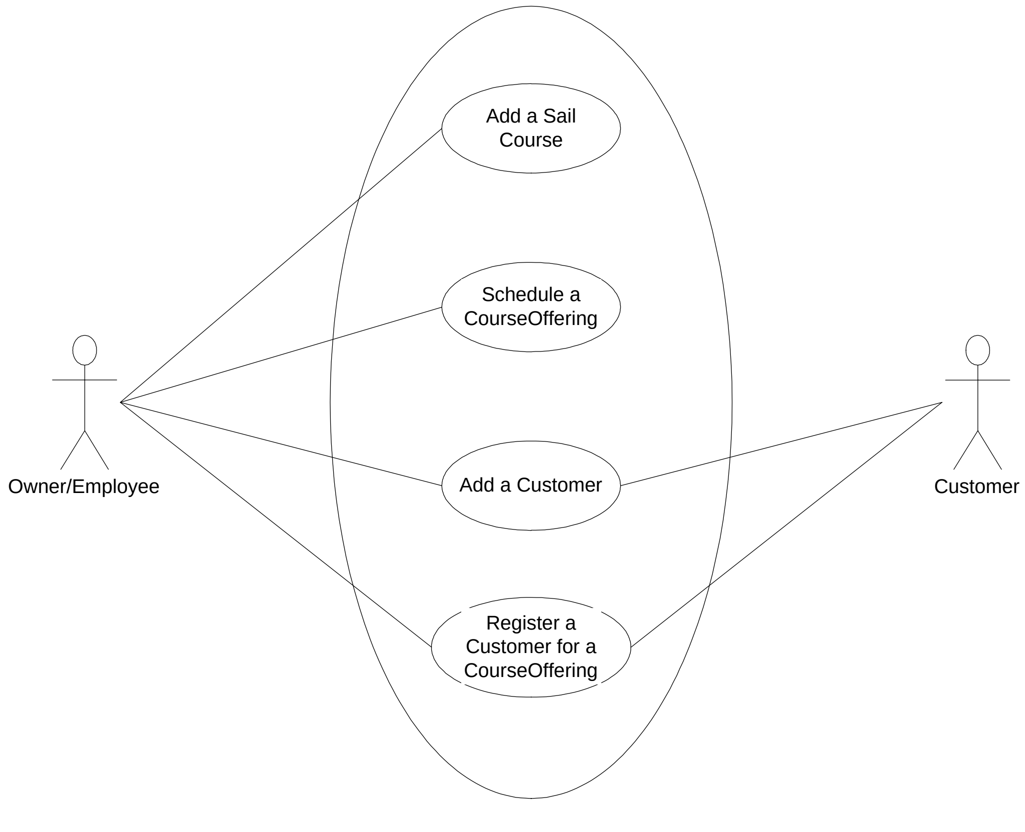

Add a Sail Course
Schedule a CourseOffering
Add a Customer
Register a Customer for a CourseOffering
Owner/Employee
Customer

Figure 4. Elaboration Iteration 1: Use Case Model

boundary decisions such as the decision to have direct customer interaction are made early. Each use case is described as a short narrative and some are described in more detail using a flow of events format.

The domain model for the first iteration is shown in Figure 5, a UML class diagram with four classes involved in the use cases for the iteration. Again, a limited scope is used for this first iteration. Attributes and association relationships are shown (including an association class). Methods are not shown. We assume readers are familiar with basic UML class diagram notation.

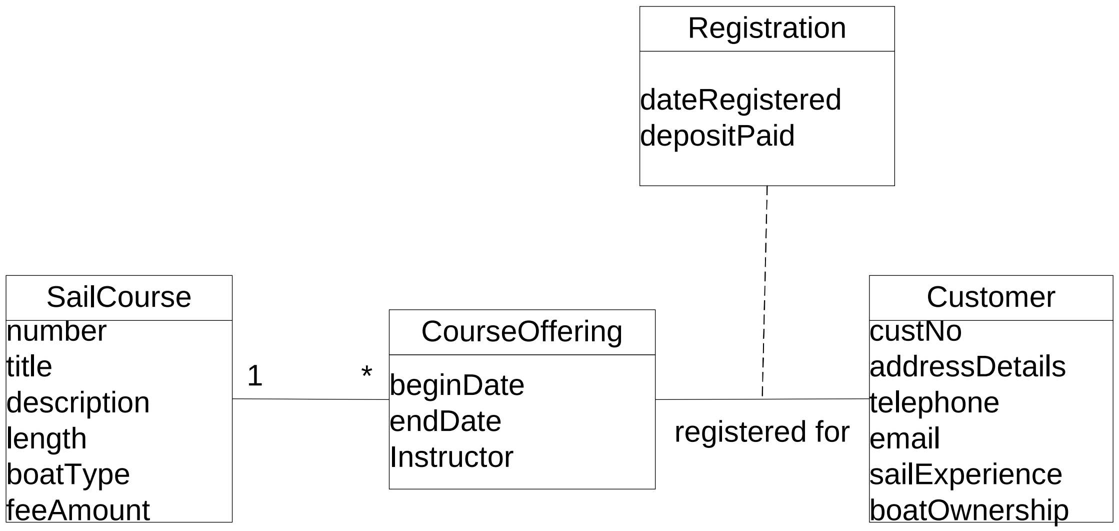


Figure 5. Elaboration Iteration 1: Domain Model

Still focusing on requirements, each use case can be modeled using an activity diagram or a use case description that highlights the activities completed by the actor and by the system. An example of an activity diagram for the use case Schedule Course Offering is shown in Figure 6a. An example of a use case description for the use case is shown in Figure 6b. Additional activity diagrams and use case descriptions are developed for each use case and as required for various scenarios or alternative processing options for a particular use case.

A final diagram used to model a use case is a variation of the UML sequence diagram, called a system sequence diagram. Most readers are probably familiar with sequence diagrams. They model the time-dependent behavior of the system as a sequence of messages from object to object. The rectangles represent objects, the dashed vertical lines represent a lifeline representing time, and the horizontal arrows represent messages (dashed arrows represent returned data in response to a message). 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. Only the input messages and returned output data are included.

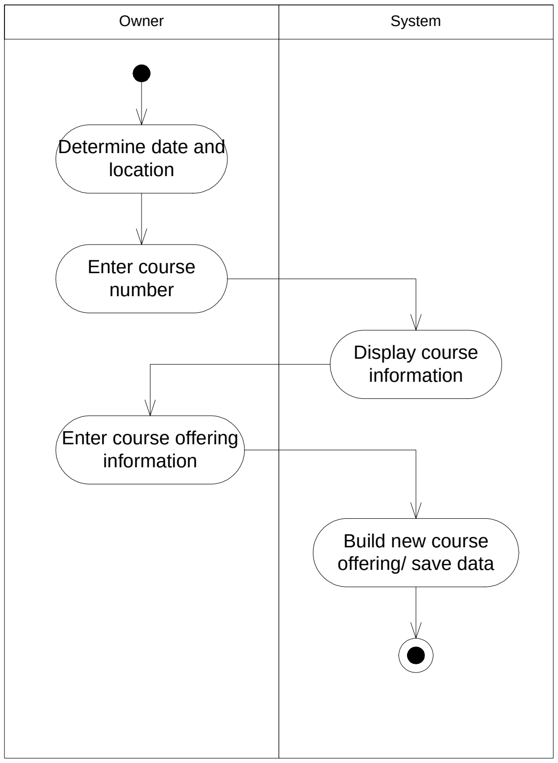

Owner
System
Determine date and location
Enter course number
Display course information
Enter course offering information
Build new course offering/ save data

Figure 6a. Activity Diagram for Schedule Course Offering Use Case

Use Case Name:	Schedule Course Offering	
Scenario:		
Triggering Event:	Owner decides to schedule a class	
Brief Description:	Owner searches for course in the system using a course descriptor, when it is displayed, he/she enters the details of the course offering, the system builds a new course offering	
Actors:	Owner, manager	
Stakeholders:	All office managers	
Preconditions:	Course must exist	
Postconditions:	Course offering must be created	
Flow of Events:	Actor	System
	1. Owner first collects all information for new offering 2. Owner enters course number to check validity of course 3. Owner enters information about the course offering (date, time, etc.)	2.1 System finds course and displays information 3.1 System creates new course offering record. 3.2 System displays information about the offering
Exception Conditions:	2.1 If course is not a currently offered course, the owner must enter course description information first	

Figure 6b. Use Case Description for Schedule Course Offering Use Case

Figure 7 shows the system sequence diagram that corresponds to the activity diagram for the use case Schedule Course Offering. The actor sends a message to the system requesting information

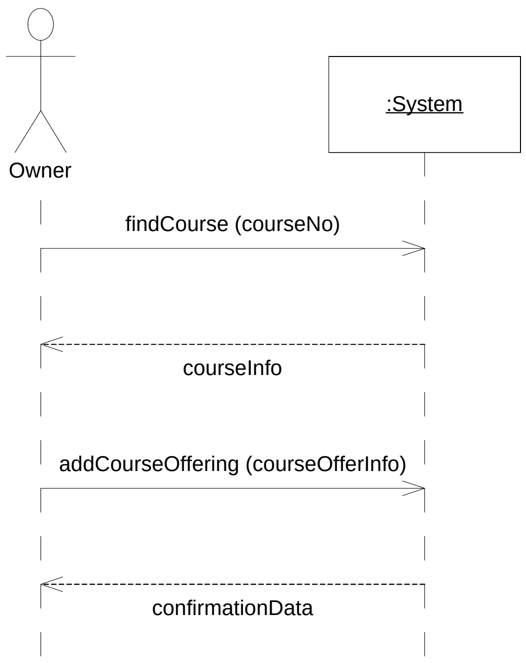


Figure 7. System Sequence Diagram for Use Case Schedule Course Offering

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. Again, it is important to realize that developers worry about being bogged down creating too many diagrams when defining requirements. However, a correct balance is important. Experienced developers recognize that the process of developing models is an essential step to understand and clarify user requirements. Too much time spent on “documentation,” however, detracts from the real objective, which is a thorough understanding of the user’s needs. A use case description, activity diagram, and system sequence diagram all define the same use case under study, but from differing perspectives so that a thorough understanding the use case is obtained.

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 within one iteration.

IV. OO DESIGN MODELS WITH UML

As we saw in Section III, three primary models are used to capture and document the user requirements in the requirements discipline:

- the domain model, which identifies the real world problem domain objects,
- the use case diagram and use case descriptions, which identify the processes that must be supported by the system, and
- the system sequence diagrams, which provides more details concerning the inputs into the system by the user and the outputs the system returns.

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.

WHY DO DESIGN?

The purpose of the design discipline 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 include 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. But many benefits are obtained by doing more formal OO design.

1. Going through the steps of systems design adds further clarification in understanding the user requirements. Those that do modeling, both at the requirements level and at the design level, know that the modeling activity itself raises questions about the new systems. Questions always come up during the construction of a model that are never thought about if the model was not created. Typical questions include: “What does this attribute or association really mean?” “How do these objects really interact?” “What should the system do when such-and-such happens?” The benefit, of course, is that the solution is more comprehensive and complete.

2. The solution system is higher quality, more robust, and more maintainable. Good design principles can be applied at the architectural level. Developing and reviewing design diagrams enable developers to identify common design issues and apply standard “best practice” solutions. This identification is the basis of all the work done in the development of OO system design patterns[1]. When developers jump right into programming without laying out the overall structure and design, then refinements and good design principles do not even get thought about. There is an old adage in systems development that the “first solution is usually the worst solution”, or “plan on throwing away your first solution.” However, once we commit something to code, we are reluctant to throw it away and do it right. We keep trying to “fix” what is already there. Having a solution only on a set of drawings that can easily be reviewed for quality, refined, and improved increases the probability of producing a high quality system.

3. Taking time for design saves time. Many developers think that they are making faster progress if they jump right into code without spending time to design. However, the design process can often be done quickly. It does not take long to lay out some diagrams. Areas of optimization, or shortcuts, or reuse can frequently be found that will expedite the coding. Most often, however, taking time for design will shorten the programming time due to fewer mistakes and fewer components that need to be redone. Taking the time to design and coordinate the designs of various subsystems and developer teams always saves time. System testing can also be done more effectively by using the design as a blueprint to develop the test plan.

These are just a few of the many benefits from taking the time to do system design before programming. The iterative nature of the UP permits doing architectural, high-level design, then some detailed design, then move into programming in one iteration. Then we iterate again, refine and add more design and more code. The complete design is not done on the first iteration, so there a long, drawn-out design activity is not needed before we can move into programming. However, to move directly to programming without addressing the issues of design almost always decreases the effectiveness of the programmers and the quality of the final system.

ARCHITECTURAL LAYERS

As discussed in the previous subsection, 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

[1] Design patterns are best practice solutions to common OO design problems that are cataloged in books and articles and available to developers to apply when needed.

services. This architecture is often referred to as three-tier design or as three-layer design. The term tier can imply a physical separation on separate processors, so many prefer the term layer implying a separate software component independent of location. We will use the term layer.

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. Therefore, OO analysis focuses on problem domain classes and use cases, and OO design focuses on physical design issues such as user interface design and database design. That is one way to think of the difference between OO analysis and OO design. This paper does not focus on the user interface design or the data access and database design; instead, it focuses on the design details of the problem domain classes (the middle layer) that also must be defined in detail. This three-layer architecture is shown in Figure 8. Later in this section, we briefly introduce some details of a three-layer design.

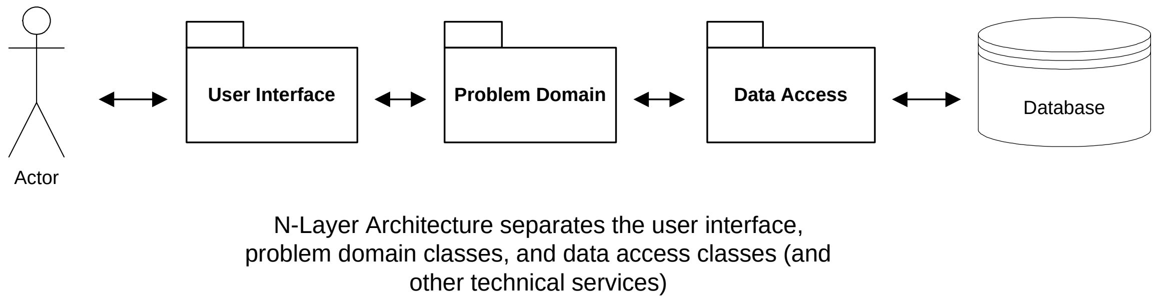

Actor
User Interface
Problem Domain
Data Access
Database
N-Layer Architecture separates the user interface, problem domain classes, and data access classes (and other technical services)

Figure 8. Three-Layer Architecture for Design

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 the structured technology, there is a big disconnect between the data flow models and the structure charts. Structured design necessitated a difficult and somewhat amorphous process to develop a structure chart from a set of data flow diagrams. In object-oriented design, the primary models are

1. Design Class Diagrams and
2. Detailed Interaction Diagrams.

A Design Class Diagram (DCD) 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.

Design Class Diagram

The following figure is an example of the SailWorld Sailing School Design Class Diagram (Figure 9). Notice that it is similar to the domain model developed during business modeling discipline activities. We will discuss. three major additions.

1. The attributes are defined more precisely by the addition of type information, and in some cases visibility information. Default visibility for attributes is invisible or private, meaning values cannot be seen by outside objects. We only add visibility notation using a plus sign when an exception occurs, so none are shown. We also identify class variables (shared in VB .NET and static in Java) with underlining, but there are none in the

example. In some cases, new attributes will be added to handle programming needs including attributes for vector arrays or status information.

Method signatures are added. Method signatures include visibility (default is public), method name, parameter types, and return types. Class level methods, i.e. shared in VB .NET and static in Java, are identified by underlining, but there are none in the example. The DCD usually does not include constructor methods, accessor methods, or mutator methods unless some specific uniqueness should be identified.

2. Navigation arrows are added. Navigation arrows indicate visibility from one class to another, that is, an object of one class is aware of and can send a message to an object of the other class. The actual implementation of this navigation in a programming language is with a variable that references another object. It should be noted that navigation is not the same as the association relationships in the domain model. Frequently, though, we can identify navigation requirements from the relationships. In Figure 9 note the navigation arrows from the controller named UseCaseController to the Customer and to the Course. Thus, instantiated controller objects have a reference variable that will point to the current customer object and the current course object.

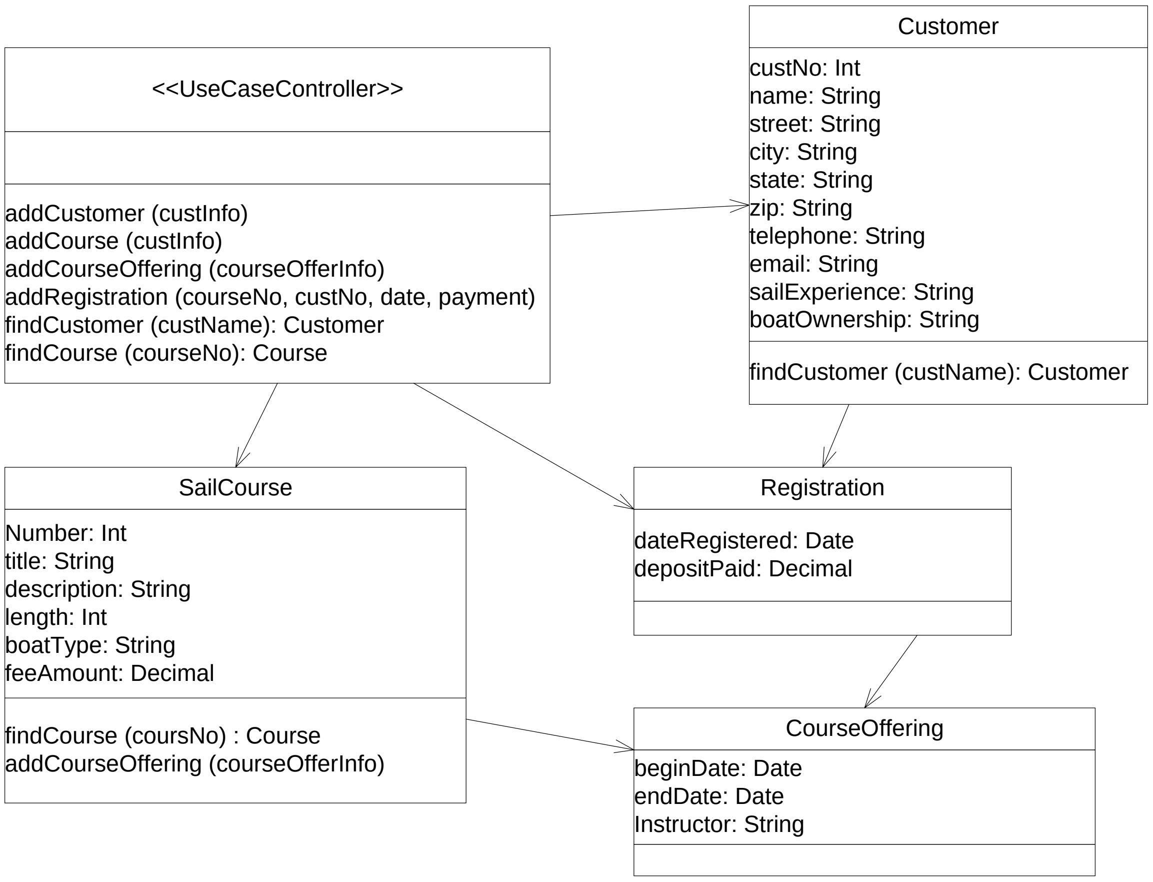


Figure 9. SailWorld Problem Domain Design Class Diagram (DCD)

The Design Class Diagram serves two purposes.

- It functions as a quality review document to ensure that the design is robust and complete.

- It provides the basis for programming activities. As can be seen in the diagram, every programming class is identified along with its typed attributes and method signatures.

Interaction Diagrams

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 will identify 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. When the UML model is translated to program code, these messages indicate that a method is invoked by an object. Thus the identification of all of the interacting objects and their respective messages is equivalent to identifying which methods will be invoked by which objects during the execution of the use case. The process of developing interaction diagrams is the foundation of OO system design.

The benefit of designing these interactions separately from programming is that good design principles can be applied and the design can be refined and improved until a correct and high quality approach is developed. The visual nature of design modeling assists in "seeing" good approaches to system structure from a broad perspective.

Two types of Interaction Diagrams are used for system design:

(1) Sequence Diagrams (Figure 10) and

(2) Collaboration Diagrams (Figure 11).

Both types of diagrams present information that is essentially the same, but from different views. A sequence diagram includes a "life line" for each object with the order of the messages indicated via a top-to-bottom, left-to-right reading. A collaboration diagram is more of a summary or overview of the collaborating objects with the order of the messages indicated by message numbers.

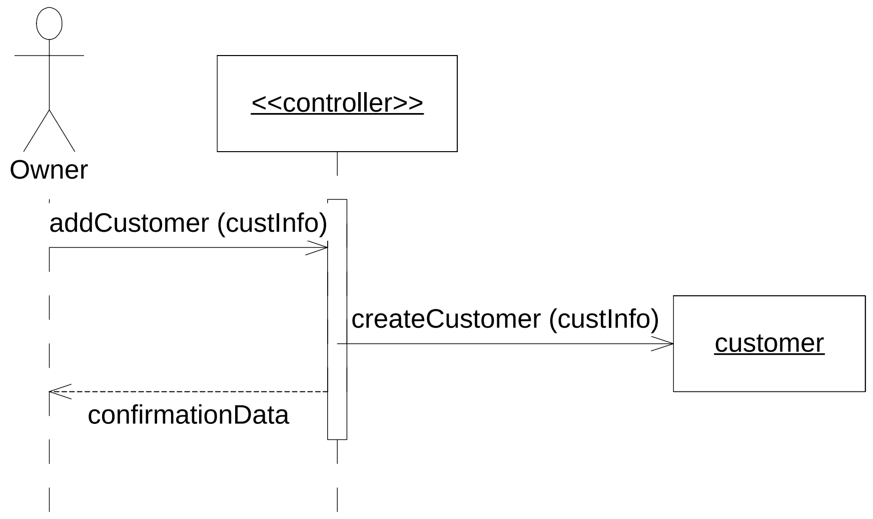


Figure 10. Sequence Diagram for Use Case *Create a Customer*.

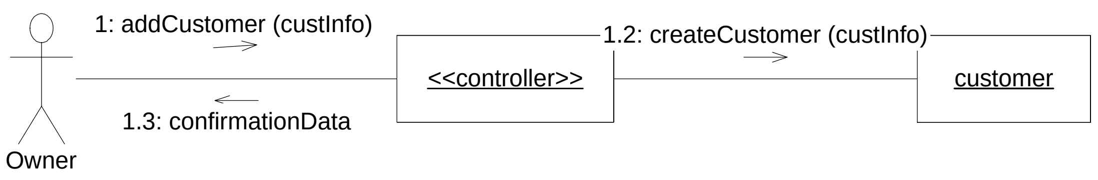


Figure 11. Collaboration Diagram for Use Case *Create a Customer*.

Figures 10 and 11 present examples of a sequence diagram and a collaboration diagram for the use case Add a Customer. As can be seen both diagrams show the same messages. In the sequence diagram, each object has a lifeline (dashed vertical line connected to the bottom of the object). The controller object also has an activation lifeline, illustrated by a long, narrow vertical rectangle on the lifeline. The activation lifeline indicates that the object is executing during that period. Each message includes a source object and a destination object. The order of the messages is read top to bottom. In the collaboration diagram, each pair of communicating objects is connected by a link that serves as the communication mechanism between the objects. Again each message includes a source object and a destination object. The order is indicated with sequence numbers. The dot numbering notation on message numbers indicates dependency that some messages occur after, and are dependent on prior messages.

The message syntax is quite similar to method syntax in a programming language. The destination object for each message is required to contain 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 9 and the messages in Figure 10, note that the message *addCustomer(custInfo)* in the sequence diagram goes to the controller object, and a corresponding method *addCustomer (custInfo)* is included in the controller class in the DCD.

THE DESIGN PROCESS

Many OO analysis and design books include detailed interaction diagrams as part of the analysis process. Because interaction diagrams identify the internal messages and methods of the system under development, creating detailed sequence diagrams is a design discipline and not a business modeling or requirements discipline.

The design discipline 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. At times a use case is complex, and design is done separately for all of the scenarios or variations of the use case. In other words, the design discipline is carried out use case by use case.

Another important point to remember is that design is also an iterative process. The UP indicates that Elaboration and Construction phases should be done through a process of iteration. The UP iterations are a macro-level iterations. Within each UP iteration, it is often beneficial to carry out design via micro-level iterations. As will be shown in the next subsection, 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.

- Experienced developers are familiar with principles such as low coupling and high cohesion and apply them.

- Common design patterns, such as controller and factory, are applied to the design to utilize best practices.
- Design improvements based on applying principles of good design and using design patterns are also achieved via micro-level iterations.

For purposes of this paper, we discuss each idea separately in the next three subsections.

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 continue to work on a use case of intermediate complexity: 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, both developed during business modeling and requirements definition tasks. As we look at the domain model, it appears that the domain classes that might be impacted are the SailCourse and the CourseOffering. We begin a sequence diagram by placing the Owner actor (from the use case), and a course object and a course offering object on the diagram (Figure 12). 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 this design pattern and why the added object is a good design practice). This additional object will be called controller. It will serve as a kind of switchboard to distribute messages that come from external points.

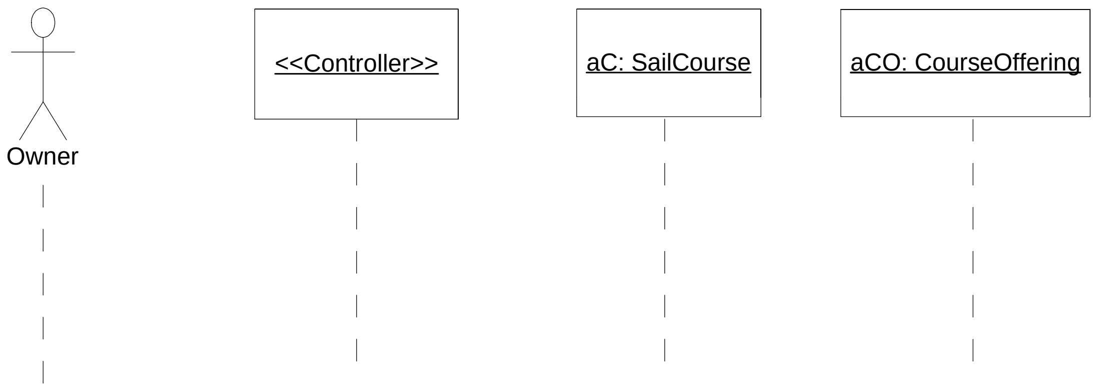


Figure 12. Preliminary Sequence Diagram Showing Actors, Objects and Lifelines.

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. It should be noted, however, that as more detail is developed, that some original user requirements may need to be modified and corrected based on better understanding of the problem and possible designs. In other words, use what was developed before, but recognize that it might not be completely correct.

The input messages from the SSD that we add are “findCourse (courseNo)” and “addCourseOffering(courseOfferInfo).” Those two messages are shown with an origin from the Owner actor and with a destination of the controller object. The next step is to analyze each of these input messages and extend out the necessary internal messages required to complete that interaction. Figure 13 illustrates the completed sequence diagram for this use case for micro-level iteration 1 (without user interface or data access).

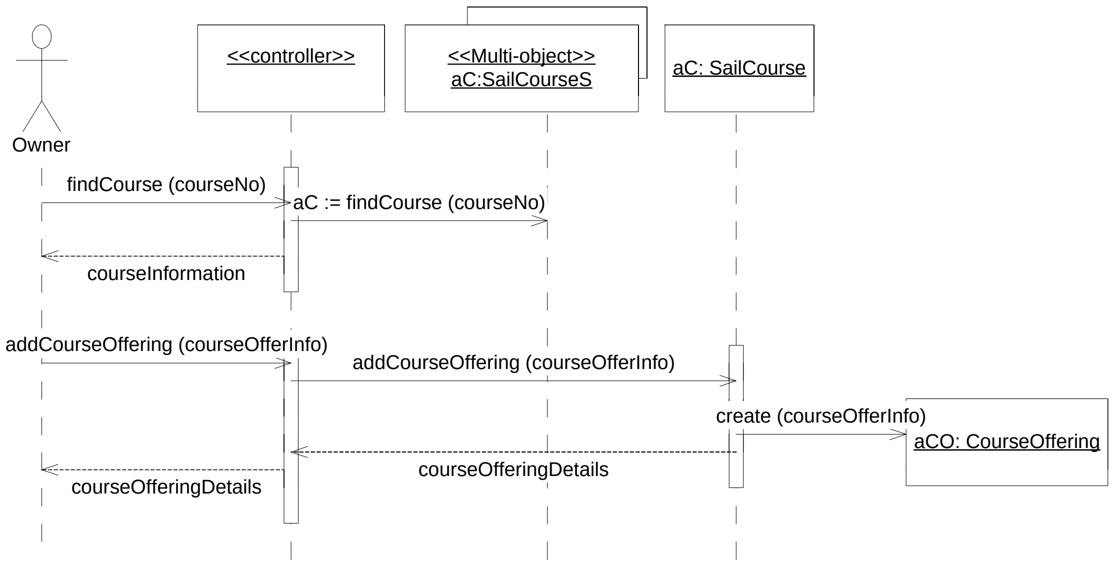


Figure 13. Schedule a CourseOffering Sequence Diagram.

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 at the top center of Figure 13 indicate that SailcourseS is an internal array or set of courses. 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. We will get to that in the next micro iteration.

The message format now includes a return value to illustrate that the found course, "aC" is returned to the controller object. Information about this course is returned to the owner actor as indicated by the dashed output message arrow.

Finally, we extend the input message "addCourseOffering(courseOfferInfo)." This message originates from the owner actor and is sent into the system via the controller 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 case, 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 are discussed later in this section. The final result of this decision is that the controller sends the addCourseOffering message to the course object, which, in turn, sends a create message to the course offering. In UML, we denote the instantiation of a new object with a create message. The create message indicates that a constructor will be invoked during execution. Create messages are normally shown sent directly to the object rectangle instead of the lifeline.

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 created 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. Neither did we included 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 with a message destination must be able to handle that message, so the domain class must include a method. Looking at the sequence diagram, we identify two methods for the Controller class, namely findCourse and addCourseOffering. For SailCourse we identify a method called addCourseOffering. We also note that there is a create message pointing to the CourseOffering class. We can add a constructor method to that class definition in the Design Class Diagram, or optionally we may leave it off. In this case, we decide to leave it off. Figure 14 illustrates the partially complete versions of the DCD classes Controller and Course.

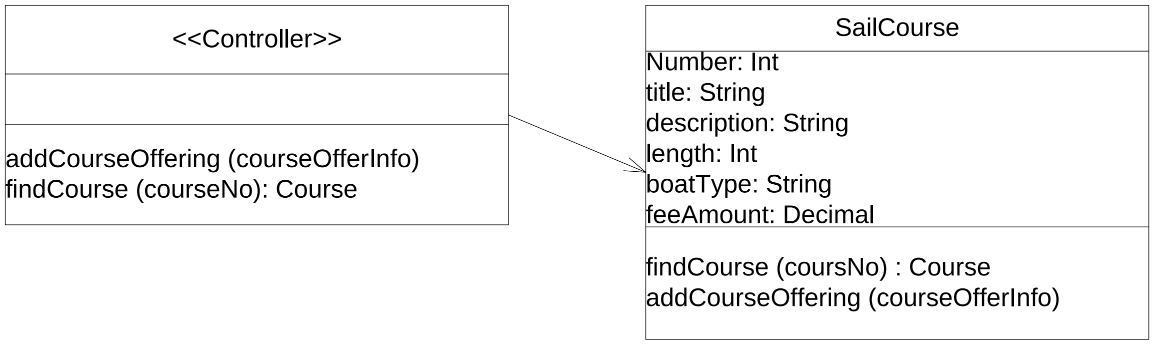

<<Controller>>
addCourseOffering (courseOfferInfo)
findCourse (courseNo): Course
SailCourse
Number: Int
title: String
description: String
length: Int
boatType: String
feeAmount: Decimal
findCourse (coursNo) : Course
addCourseOffering (courseOfferInfo)

Figure 14. Partial Design Class Diagram Derived from Sequence Diagram for Schedule Course Offering.

This same process is followed for every use case that was chosen to be part of the first UP elaboration iteration. The end result of this activity is to provide 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 15, 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 actor go through the ScheduleWindow. Then all messages from the view layer go through the controller. The controller 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.

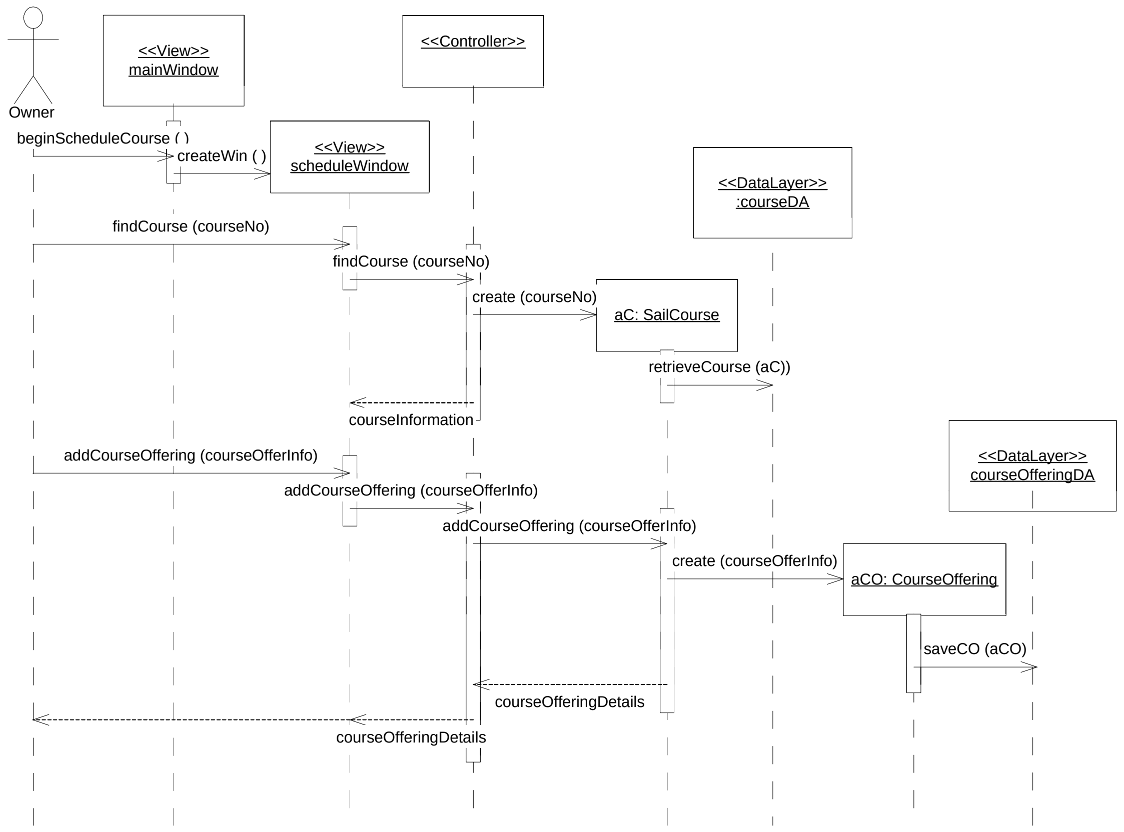


Figure 15. Three-layer Sequence Diagram For Schedule Course Offering.

Once the schedule window is created, the owner actor enters the course ID for the new offering. The schedule window sends a find request to the controller object, which creates a new course object. Here the multi-object ac:SailCourseS is replaced by a mechanism that creates one new course instance from data in the database to represent the requested course. The only information provided to the Course constructor is the course ID, so it accesses the data layer class, CourseDA, to go read the database and retrieve the rest of the information required. The CourseDA class contains all of the logic to connect to the database, execute an SQL statement, and extract the information from the result set and place it in the empty Course object that was provided. The data access layer classes contain all the logic to access the database and convert the result set, which is returned from the database system, back into domain objects.

After the correct course is found and displayed, the owner actor enters the information to schedule a new offering. That information goes from the window, to the controller object, to the course object, which then is responsible for creating a new courseOffering object. The courseOffering object is responsible to access the courseOfferingDA data access object to save the information to the database.

At this point in the design process, the solution for the Schedule CourseOffering is essentially complete except for any exception processes and error messages. As before, we can now transfer this information to the design class diagram. Figure 16 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.

To recap, we worked on the first iteration of the UP Elaboration Phase. The SailWorld project will require many more iterations as it moves through the Elaboration, Construction, and Transition phases. The first iteration included four key use cases. The requirements discipline defined each use case using an activity diagram, a use case description, and a system sequence diagram. The design discipline took each use case and developed interaction models (sequence diagrams or collaboration diagrams), design class diagrams, and package diagrams. Micro iterations for each use case designed the user interface (view) layer and the data access layer interactions. Although not shown in this paper, programming is based directly on the design models so that the iteration ends with a compiled executable.

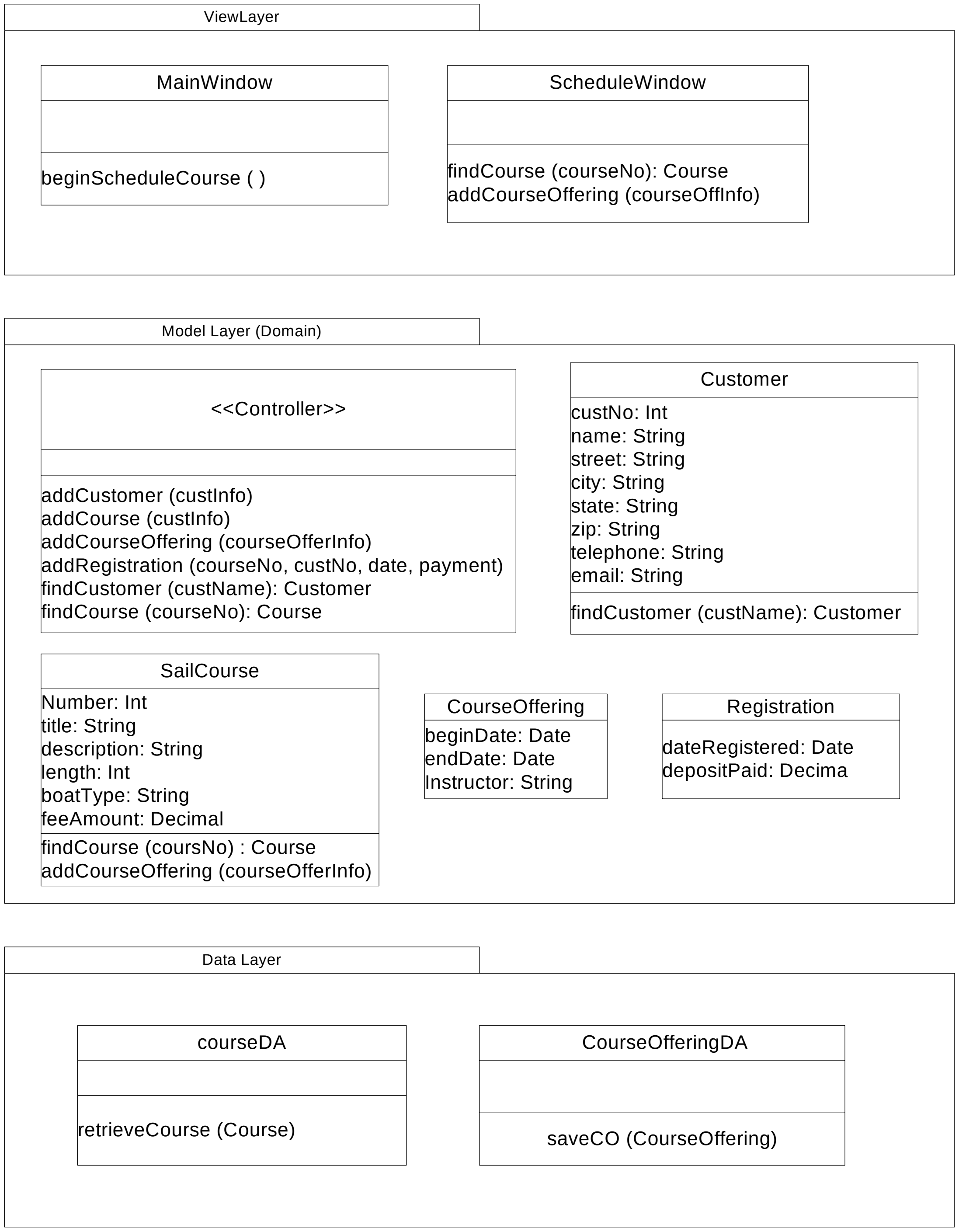


Figure 16. Design Classes by Package for Three-Layer Design

APPLYING PRINCIPLES OF 'GOOD' OO DESIGN

As indicated at the beginning of this section, during the design discipline we also apply principles of good object-oriented design. This topic is both broad and deep. In this tutorial space only permits briefly identifying 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.

Principle 1: 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. Well-designed classes are said to be highly 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. Figure 15 shows classes whose responsibility is to provide a graphical user interface, classes which only access the database, and classes which provide the processing to support the problem domain business logic. Each of these classes also is focused in that it works primarily on one type of object or data structure. Classes that are not cohesive are hard to comprehend and difficult to maintain. A system with many classes that are not cohesive tends to exhibit severe "ripple effects" when changes are made. Related to high cohesion are other principles of design such as encapsulation and information hiding.

Principle 2. Coupling between the classes in the system. In a well designed system, coupling is limited and controlled. Coupling between classes occurs when one class can see another class and accesses the methods or attributes of that class. For example in Figure 15, the SailCourse domain model class is coupled to the CourseDA class, but not coupled to the CourseOfferingDA class. As another example, the coupling between the view layer and the model layer is done through the Controller class.

Principle 3. "Protected variations". The previous two principles influence another principle: "protected variations" Larman [2002]. Protected variations means 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. For example, the effect of any changes to database system or data access will be minimal on the model and view layer classes. Those layers are protected from database changes by the data access classes. To consider low coupling and high cohesion during design produces a system with classes that are protected from variations. This fundamental principle should be applied throughout the design process.

Principle 4. Assigning responsibilities to classes. These responsibilities include responsibility to create other objects, to access information, to be visible to other objects, to accumulate information, among others. Figure 15 shows an example of create responsibility. The question is asked, "Which class should be responsible for instantiation of new CourseOffering objects?" In this situation, since CourseOffering objects depend on the existence of a Course object and are subservient to it, then the Course object is responsible for creating the CourseOffering objects.

Another question from Figure 15 might be, "Who should be responsible for retrieving all the information to fill in the fields for a new Course object?" In some solutions, we might see the controller class accessing the CourseDA class to retrieve the data and then instantiate a Course object. The solution provided offers a better assignment of responsibilities, however. The Controller class instantiates a Course object that only contains a course ID. The Course object is responsible for invoking the data access class to finish filling in the required information.

In this section, we illustrated the types of issues that should be considered while doing OO design. 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. The final system may execute and produce results, but the long term support and maintenance work will be difficult and painful.

DESIGN PATTERNS

One of the most important advances in object-oriented design is the recognition that certain design problems occur over and over and that "best practices" can solve these design issues. These best practice solutions are called design patterns. Numerous articles and books now identify and categorize many of these design patterns.

System developers that employ good design principles will also be familiar with many standard design patterns. A course that teaches good object-oriented design must not only include important design principles, but should also introduce the most widely used design patterns. Design patterns exist for almost every facet of object-oriented design. Design patterns are characterized in various ways 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 appropriate for distributed systems. Obviously, careful consideration needs to be given to decide which patterns are most appropriate for any given course on OO 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 already shown, the multi-layer design pattern is an important and critical component of good object-oriented design.

In the examples shown in Figures 9 through 16, we saw the use of the controller pattern represented by the Controller 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. By providing only a single entry point into the model layer, and by eliminating coupling from the model back up to the view layer, the windows and screens of the view layer can be changed with little or no impact on the rest of the system. We could, for example, create several different view layers, such as a Web GUI and a desktop GUI both functioning for the same 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. The adapter pattern works just like the adapters used for electrical outlets whose form and voltage differs from your own country's. Our computer plug expects a certain socket (API). When the wall socket is different (unique API), then an adapter lets us adapt our plug (method calls) to the unique socket.

The factory pattern is another one that can be used effectively. Consider the question presented previously:, "Whose responsibility is it to create all of the data access objects (CourseDA)?" Should an object that needs the data access service always instantiate a data access object? But what if several classes need the same data access service, then who should instantiate it? This common problem is resolved with the factory pattern. 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 to ensure that only one is ever created. Since we only want one instance of the factory, and since we cannot always predict which class will need the services of the factory first, we make the factory a singleton.

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 ones 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.

V. 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 today are written in these object-oriented languages. However, even though most educators and practitioners learned how to program with these object-oriented languages, many practitioners and educators are deficient in object-oriented systems design.

We find that some fundamental principles of OO 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. Therefore, most information systems professionals are familiar with the modeling done in the UP requirements discipline. Likewise, the principles of object-oriented programming are being taught and much object-oriented programming is being done. However, a large gap is left in two major areas:

1. the processes, techniques, and artifacts (e.g. models) that bridge the gap from requirements modeling to programming, and

2. the total development process for building systems using the object-oriented paradigm.

In this paper, we addressed both these issues. First, we explained some modeling related aspects of the Unified Process (UP), which is a total development process for building systems using object-oriented models. The UP is not the only development methodology that can be used to apply the OOA and OOD models and principles we discussed, but it is the new frame of reference or baseline upon which newer, more “lightweight” methodologies are being discussed [Beck, 2000; Ambler, 2002]. Therefore, it is important for educators and practitioners to understand the UP. Second, we showed the development process using Unified Modeling Language (UML) models to illustrate how to do analysis (requirements discipline) and design (design discipline) and how to move from requirements into and through architectural and systems design. It is somewhat paradoxical that in the UP the distinction between analysis and design, in the traditional sense of the word, tends to become fuzzy because of the multiple iterations. Yet there is a distinct difference in the modeling approach used for discovery and analysis, and the modeling approach used for design. In this paper, we carefully denoted these differences to illustrate the important thought processes and activities that are uniquely associated with OO systems development.

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 unique from OO analysis and 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, even though several widely used design patterns apply to low-level design, such as might be taught in a programming course, many architectural design patterns transcend programming. Generally students are sufficiently challenged just learning basic programming concepts that they would find it difficult to grasp the more abstract concepts related to design and patterns. Hence, a separate focus on design issues needs to be provided someplace in the curriculum.

Editor’s Note: This article is an expansion of the tutorial presented by the authors at the 2003 AMCIS meeting in August 2003 at Tampa, FL. The article was received on August 18, 2003 and was published on December ___, 2003. It was with the authors for six weeks for two revisions.

REFERENCES

Ambler, S. W. (2002) *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*, New York: Wiley

Beck, K. (2000) *Extreme Programming Explained*, Reading MA: Addison-Wesley.

Boehm, B. (1988). A Spiral Model for Software Development and Enhancement, *IEEE Computer*, (21)5, pp. 61-72.

Booch, G., J. Rumbaugh, and I. Jacobson, I. (1999). *The Unified Modeling Language Users Guide*, Reading, MA: Addison-Wesley.

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

Jacobson, I., G. Booch, and L. Rumbaugh (1999). *The Unified Software Development Process*, Reading, MA: Addison-Wesley.

Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*, Reading, MA: Addison-Wesley.

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

McConnell, S. (1996) *Rapid Development.* Redmond WA: Microsoft Press.

GLOSSARY

Accessor method – a method that retrieves attribute values of a class.

Class – a category or type of thing that includes attributes and methods that all objects belonging to the class contain.

Collaboration diagram - a UML interaction diagram that shows object interactions focusing on message coupling among objects.

Constructor method – a method of a class that is responsible for creating new instances.

Design class diagram – a version of a UML class diagram that shows design details about each class, including visibility, types, and method signatures.

Design pattern - best practice solutions to common OO design problems that are cataloged in books and articles and available to developers to apply when needed.

Mutator method – a method that updates attribute values of a class.

Object – a specific instance of a class

Object oriented - a view of software systems as being a collection of interacting objects.

OO analysis – the activities involving gathering information, defining requirements, and prioritizing requirements for an information system development project.

OO design – the activities involving defining the architecture, detailed classes, and object interactions that depict a multi-layer information system that satisfies the requirements.

Scenario – an instance or version of a use case that follows a specific path to completion.

SDLC – the traditional waterfall system development life cycle used as a project management framework for information system development projects.

Sequence diagram – a UML interaction diagram that shows object interactions in time sequence

System sequence diagram (SSD) – a version of a UML sequence diagram that shows the interactions of the actor and the system as messages, but leaving the system as a black box.

Unified modeling language (UML) – a set of constructs and diagrams used for modeling object-oriented systems accepted as a standard by the Object Management Group (OMG).

Unified process (UP) – a comprehensive and iterative system development methodology that used for object-oriented development with Inception, Elaboration, Construction and Transition phases.

Use case - a story that describes a case where the user interacts with the system to accomplish something of value

ABOUT THE AUTHORS

John W. Satzinger and **Robert B. Jackson** are co-authors (with Steven Burd) of *Systems Analysis and Design in a Changing World* (3rd Edition), a systems analysis and design text with balanced coverage of traditional and OO approaches to information systems development, and *Object-Oriented Analysis and Design with the Unified Process*, forthcoming in 2004. Dr. Satzinger is on the faculty of Southwest Missouri State University and previously taught at the University of Georgia and Cal Poly Pomona. He is also co-author of *The Object-Oriented Approach: Concepts, System Development, and Modeling with UML* (2nd Ed), *Object-Oriented Application Development Using Java*, *Object-Oriented Application Development Using VB .NET* and numerous articles about system development in journals including *Information Systems Research* and *Journal of MIS*. Dr. Jackson is on the faculty of the Marriott School of Management at Brigham Young University. He is a contributing author to the *Handbook of Object Technology* (CRC Press) and published articles on OO development in journals such as *IEEE Software* and *Information and Software Technology*.

Communications of the Association for Information Systems

ISSN: 1529-3181

EDITOR-IN-CHIEF
Paul Gray
Claremont Graduate University

AIS SENIOR EDITORIAL BOARD

Cynthia Beath Vice President Publications University of Texas at Austin	Paul Gray Editor, CAIS Claremont Graduate University	Sirkka Jarvenpaa Editor, JAIS University of Texas at Austin
Edward A. Stohr Editor-at-Large Stevens Inst. of Technology	Blake Ives Editor, Electronic Publications University of Houston	Reagan Ramsower Editor, ISWorld Net Baylor University

CAIS ADVISORY BOARD

Gordon Davis University of Minnesota	Ken Kraemer Univ. of California at Irvine	Richard Mason Southern Methodist University
Jay Nunamaker University of Arizona	Henk Sol Delft University	Ralph Sprague University of Hawaii

CAIS SENIOR EDITORS

Steve Alter U. of San Francisco	Chris Holland Manchester Business School	Jaak Jurison Fordham University	Jerry Luftman Stevens Institute of Technology

CAIS EDITORIAL BOARD

Tung Bui University of Hawaii	H. Michael Chung California State Univ.	Candace Deans University of Richmond	Donna Dufner U.of Nebraska -Omaha
Omar El Sawy University of Southern California	Ali Farhoomand The University of Hong Kong	Jane Fedorowicz Bentley College	Brent Gallupe Queens University, Canada
Robert L. Glass Computing Trends	Sy Goodman Georgia Institute of Technology	Joze Gricar University of Maribor	Ruth Guthrie California State Univ.
Juhani Iivari University of Oulu	Munir Mandviwalla Temple University	M.Lynne Markus Bentley College	Don McCubbrey University of Denver
Michael Myers University of Auckland,	Seev Neumann Tel Aviv University, Israel	Hung Kook Park Sangmyung University,	Dan Power University of Northern Iowa
Nicolau Reinhardt University of Sao Paulo,	Maung Sein Agder University College,	Carol Saunders University of Central Florida	Peter Seddon University of Melbourne Australia
Doug Vogel City University of Hong Kong,	Hugh Watson University of Georgia	Rolf Wigand University of Arkansas	Peter Wolcott University of Nebraska-Omaha

ADMINISTRATIVE PERSONNEL

Eph McLean AIS, Executive Director Georgia State University	Samantha Spears Subscriptions Manager Georgia State University	Reagan Ramsower Publisher, CAIS Baylor University