Teaching the Complete Object-oriented Development Cycle, Including OOA and OOD, with UML and the UP

Robert B. Jackson
School of Accountancy and Information Systems
Brigham Young University
Provo, Utah, 84602, U.S.A.
rbj2@email.byu.edu

John W. Satzinger
Computer Information Systems
Southwest Missouri State University
Springfield, Missouri, 65804, U.S.A.
jws086f@smsu.edu

Abstract

Many information system programs currently teach a combination of structured techniques and object-oriented techniques for system development. Very few programs teach complete OOA and OOD concepts based on UML and tie it in with OOP. Consequently many students are leaving the university with an inadequate set of OO skills. This paper describes a curriculum for teaching a complete set of skills for doing object-oriented development. Included are explanations for how to teach the unified process (UP), object-oriented analysis, and object-oriented design in such a way that it directly supports teaching object-oriented programming.

Keywords: object-oriented analysis, object-oriented design, unified modeling language, unified process, object-oriented curriculum

1. INTRODUCTION

System development techniques, and tools, continue to change and evolve. This fact presents an ongoing challenge to information systems programs to keep current with new technology. Some new technologies are passing fads and do not merit integration into a program. However, one fundamental change is occurring in systems development that does merit integration into information systems programs. That new technology is the move to object-oriented development.

Many information system programs currently teach a combination of structured techniques and object-oriented techniques. Most programs teach object-oriented programming (OOP) using languages such as Java, Visual Basic .NET, or C++. However, many programs are deficient in teaching the corresponding object-oriented analysis and design skills. Some schools teach only structured techniques in the systems analysis and design class. Others teach some structured and object-oriented analysis (OOA), but no object-oriented design (OOD). Very few pro-

grams teach complete OOA and OOD concepts based on UML and tie it in with OOP. Consequently many students are graduating and entering the work force with an incomplete set of OO skills.

One guideline that many information systems programs follow is the curriculum model presented in IS 2002 (Gorgone et al, 2002). For system development, IS 2002 includes one programming course, one analysis and design course, and one design and implementation course that includes database management. The programming course now addresses objects and object orientation as one part of the programming recommendation. The analysis and design course now acknowledges both the traditional structured approach and the newer object-oriented approach. An additional course covering design and implementation with newer technologies is included to allow covering development skills based on new approaches. IS 2002, like many college IS programs, tries to cover both traditional and OO system development to provide a broad exposure for students. However, information systems educators interested in focusing exclusively on OO development must extend the IS 2002 minimum recommendations to achieve a truly in-depth OO program.

In order for a student to become an effective systems developer using object-oriented techniques, he/she must first develop proficiency in the techniques and models of each component of OO development (OOA, OOD, and OOP) and second he/she must also be able to integrate the various techniques together into an integrated, complete, comprehensive development methodology such as that provided by the unified process (UP). As indicated above, many current IS programs have problems in both areas. Some critical components of OO development are not taught. And we do not integrate the various aspects of OO development into an integrated whole. We cannot expect students to be well educated in OO techniques until we remedy the holes in our programs.

However, as educators, we also know that a simple exposure to OO concepts is not adequate to develop a proficiency in our students. As indicated in studies about student proficiency, we believe that most programs have a goal to teach not just techniques but also higher level analytical and problem solving skills. This implies that we need to get students to a level three (Analytical) on Bloom's Learning Taxonomy scale (Bloom, 1956). One difficulty with teaching both structured and object-oriented when the number of courses is limited is that students will not develop in-depth skills in either paradigm.

One of the advantages of an integrated curriculum is that there are multiple opportunities to reinforce the important skills across several courses. As concepts from one course are reviewed and used in other courses, the entire set of skills is strengthened. Synergism begins to occur and students begin to understand the overall framework and to operate at a higher level. It is difficult to reinforce skills in an integrated set of courses if the curriculum is trying to cover both traditional and OO approaches. That is why we believe it is time to consider committing to OO development exclusively.

This paper presents a method for teaching a complete set of skills for doing object-oriented development. Included are explanations for how to teach the unified process (UP), object-oriented analysis (OOA), and object-oriented design (OOD) in such a way that it directly supports the object-oriented programming (OOP) concepts taught in programming courses. An integrated approach to teaching OO programming and OO analysis and design is required. Therefore, we propose a curriculum consisting of at least four integrated courses for teaching OO.

2. A CURRICULUM FOR TEACHING OOA, OOD, OOP

Our proposed curriculum is divided into two tracks: (1) a programming track, and (2) a modeling/methodology track. Each track consists of multiple courses with later courses building on the knowledge and skills of the earlier courses. Integration also occurs between the tracks to provide a totally integrated program. Note that we assume in the discussion below that a separate database management course and other key IS2002 MIS course are included in the degree program.

We divide the program into tracks for several reasons. The primary reason is that it allows an information systems department to begin implementing a more complete object-oriented program in a more gradual move. For example, in some universities it may be easiest to move into a two semester object-oriented programming sequence. At another university, it might be more prudent to first make changes in the systems analysis and design courses to move to object-oriented concepts.

Another reason for the tracks is that the faculty members that teach one track are typically not the same ones that teach the other track. In some instances the programming instructors also do systems analysis and design with modeling, but in most cases not. Thus, by focusing on separate tracks the faculty in each area can emphasize the skills they need to develop before trying to implement the fully integrated program. However, it should be noted that a fully integrated program will require cross education so that instructors in the modeling area have basic knowledge of OO programming and vice-versa. Only then can a truly integrated program be developed.

The programming track can be organized as shown in the Table 1. Many two-course programming sequences cover introductory programming in the first course and more advanced programming techniques in the second course. However, there are dozens of advance programming concepts that can make the course seem like an endless collection of unrelated concepts and techniques: one week Applets, then object serialization, then Swing components, and then multi-threading. Many instructors teach the advanced course by following an advanced textbook. Unfortunately, these textbooks are designed to cover all the unrelated concepts behind the language. They are typically not designed to cover information system development techniques in an integrated way.

The second programming course is design to teach not only advanced programming techniques, but also introduce some basic programming design patterns. In the Table 1, the items shown with an asterisk are those items that introduce the fundamentals of design. It should be noted that those topics are also the areas where the programming track and the modeling/methodology track can most easily be integrated.

Table 1: Two Course Programming Sequence

Course	Course Description / Possible Topics
Programming 1	• Basic programming skills—data types and structures, control structures (sequence, loops, decision), arrays. • Basics of objects, methods, instantiation. • Important to develop basic programming skills. Do not spend much time on GUI tools.
Programming 2	• Advanced object-oriented programming concepts—Inheritance and overriding, interfaces, GUI and Swing, serialization and database connectivity, exception handling, … • Object-oriented class libraries—String, Vector, Iterator, Array, … • *Multi-layer systems, layer design patterns • *Basic programming patterns—Singleton, Factory, Iterator, … • *Testing and iterative development

Integration starts during the second programming course with the introduction of basic multi-layer design pattern that separates the user interface classes, problem domain classes, and data access classes. Then we focus on how an object-oriented program actually works in practice. Students use OO design models at this point, as the design models are what students will learn to implement when they write programs. Ideally, the students will have been introduced to modeling concepts prior to or concurrent with this course. An introduction to the basics of design class diagrams and possibly sequence diagrams is necessary. In those instances where the two tracks are integrated, the programming course can move rapidly through concepts of design based on the UML models. Students read these models and write code based on them.

Systems analysis and design concepts and techniques are covered in the modeling/methodology track. It includes two development courses as shown in Table 2. The first course is much like a traditional analysis and design course in terms of objectives and outcomes, except it is taught iteratively and it covers UML modeling and the UP exclusively. The second course includes a project where students work in teams to complete a system development project using OOA, OOD, OOP and the UP and provides an excellent opportunity to integrate the modeling/methods track with the programming track.

Table 2: Two Course Modeling/ Methodology Sequence

Course	Course Description / Possible Topics

Modeling/ Methods 1	• The unified process (UP)
	• Business case analysis, project management, communication
	• Planning multiple iterations
	• Developing UML requirements models
	  ○ class diagram, use case diagram, use case descritions, system sequence diagrams, statecharts, activity diagrams
	• User interface design, security/controls, conversion, …
	• Introduction to UML design models
	  ○ design class diagram, interaction diagrams, detailed statecharts

Modeling/ Methods 2	• Advanced design concepts using UML design models • *Design patterns—Architectural design, programming patterns, desktop patterns, enterprise patterns, … • *Group project planning using the UP • *System development with multiple iterations through construction ○ OOA, OOD, OOP for each iteration

The approach to integration depends on the timing of the two semester programming track and the two semester modeling/methodology track. In fact, the content of the classes in the programming track will change slightly based on the timing. The two primary alternatives are shown in Table 3. Other alternatives exist that can work. The primary consideration is that during the last semester a major project is included that requires the students to integrate the entire modeling/methods and programming track concepts.

Alternative 1 covers four courses in three semesters, with the second programming course overlapping the first analysis and design course. This is a very effective approach because students understand basic programming as they learn modeling skills. Modeling skills can then be applied in the latter part of the programming class to help students program based on design models. Other MIS courses including a database management course must be included at some point, with database fitting best in semester 2.

Table 3: Two Alternatives for Scheduling Courses

Semester 1	Semester 2	Semester 3
Alternative 1 – Three semester sequence		
Programming 1	Programming 2 Modeling/ Methods 1	Modeling/ Methods 2
Alternative 2 – Two semester sequence		
Programming 1 Modeling/ Methods 1	Programming 2 Modelling/ Methods 2	

The second alternative allows students to complete both tracks in one semester. During the first semester the two tracks are kept separate and distinct. During the second semester, especially during the last half of the semester the two tracks can be very closely integrated. A major project can be required that includes concepts from both tracks. This alternative differs from the first in that the project becomes a component of both tracks concurrently.

The remainder of this paper focuses mainly on the modeling/methods track. In many information system programs, teaching the object-oriented Programming 1 course is more mature and developed. The primary need most IS instructors have is learning how to teach UML and object-oriented modeling so that it can be integrated into the systems analysis and design courses. In the next sections of this paper we present a simple, yet effective way to teach OO modeling and the UP. Once that step has been accomplished, the next step of an integrated OO program can be addressed by designing a Programming 2 course that is based on UML models and OO design patterns. We will discuss this approach in more detail in a later section of the paper.

3. UNDERSTANDING AND TEACHING THE UP

The Unified Process (UP) is a comprehensive OO system development methodology originally developed by Jacobson, Booch, and Rumbaugh (1999A, 1999B, 1999C). The UP draws on many of the best practices in software development such as iteration and model-driven development. It has become widely accepted as a leading (if not *defacto* standard) OO development methodology. The terminology used by the UP is somewhat new

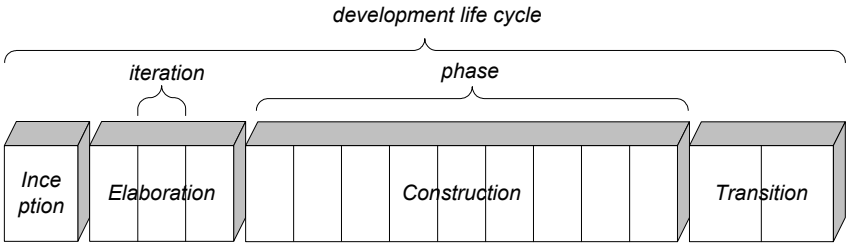
UP Iterations and Phases

The UP is fundamentally an iterative approach to software development. As with many other iterative approaches, the philosophy is to specify, design, and build a part of the system. Then with later iterations to specify, design and build more of the system so that it the solution evolves and grows into a final total solution. However, there are also some differences between the UP and other iterative approaches. The UP is also a model-driven design approach. Model building is an essential ingredient in the specification and design of the solution. So where some iterative approaches are based on prototyping techniques, e.g. build prototypes and build the system based on the users feedback of executing prototypes, the UP is somewhat more formal. The advantages of building models is well understood and usually provides a more robust and architecturally sound solution.

The point of confusion with the UP is the use of the term "phase." Historically, the idea of phases has came from the waterfall methods so that most of us think of phases as a group of activities of things like analysis or design activities. In the UP, the term phase means a "focus" or and "emphasis," and a phase is a grouping of iterations, not individual activities. In the UP, there are four system development life cycle (SDLC) phases: inception, elaboration, construction, and transition. So, for example, to say that a project is in the elaboration phase means that the current iteration, and probably the next few iterations, are concentrating on understanding the requirements. Lots of model building is being done, as well as construction of core pieces of the system. However, the requirements and design are still fluid at this point. To say that the project is in the construction phase means that the primary focus of this iteration, and the nearby iterations, is to flesh out the core system with all of the functionality and technical robustness for an industry strength system. During the construction phase, the models tend to be more design oriented to define all the technical issues such as exception handling. During the construction phase, there should be very little new specification based on user requirements.

The sequence of the iterations and how they are grouped as an emphasis or focus is shown in Figure 1. The emphasis in each of the four phases is the following: (Larman, 2002):

Inception – develop an approximate vision, business case, scope, and rough estimates. (It is, in essence, developing the business case and feasibility analysis.)
Elaboration – refined vision, iterative implementation of core architecture, resolution of high risks, identification of most requirements and scope, and realistic estimates. (The primary focus of these iterations is on system specifications.)
Construction – iterative implementation of the remaining lower risk and easier elements, and preparation for deployment. (The primary focus of these iterations is on quality, solid design, robustness, and comprehensive total solution.)
Transition – beta test, deployment. (These iterations include all those activities necessary to get it ready for deployment.)



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

UP Iterations and Disciplines

The output or deliverable of each iteration will vary somewhat depending on where it lies in the development cycle, i.e. which phase. However, in most cases the deliverable should include a set of models as well as executable system components. Since each iteration does include a working system or portion of a system, each iteration can be treated as a mini-SDLC. In other words, each iteration must have some requirements, some specification, some design, some project management, and so forth. In the UP, these are called **disciplines**. Disciplines include business modeling, requirements, design, implementation, test, deployment, configuration & 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, 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 there is much more focus on design, implementation, and testing. The UP model shown in Figure 2 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. One critical point about the UP iterations, is that they should be fairly short and focused. Iterations should range in length from about 4 to 6 or 8 weeks.

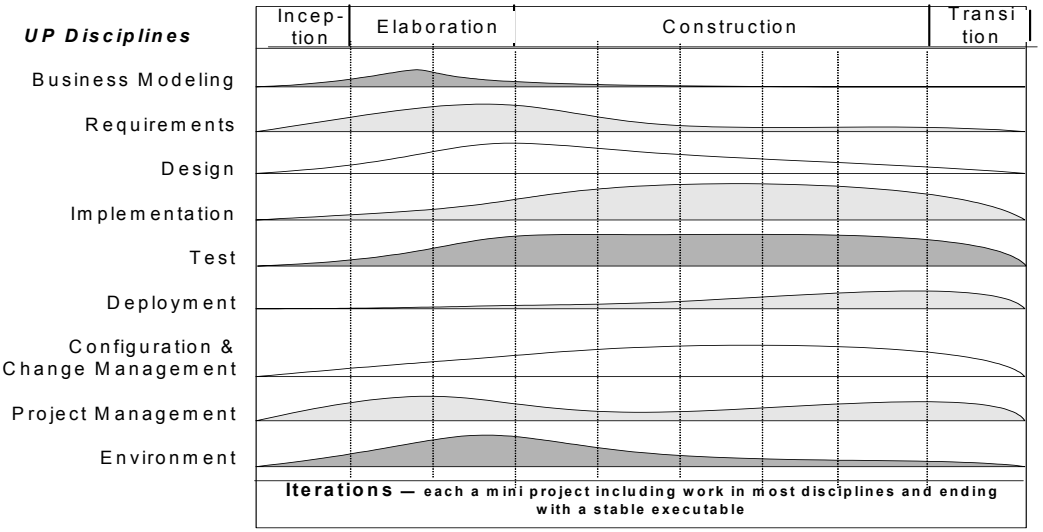


Figure 2. UP Phases, Iterations, and Disciplines

Integrating OOA, OOD, and OOP into the UP

The next sections in this paper describe OOA, OOD, and OOP. However, remember that the concepts of OOA, OOD, and OOP are no longer SDLC phases. We can still use the terms of analysis, design and programming, but they now refer to disciplines or activities within a single iteration. Since the UP is also a model building methodology, it is important that the students understand the models and understand how to use them. Those activities and models within the UP that focus on understanding and modeling the user requirements will still be called analysis. Other UP disciplines focus on designing the architecture and algorithms of the new system. Those activities and models we will call design. Other UP disciplines are oriented to programming and testing.

One benefit of object-oriented development, however, is that the OO models used for design are simply extensions of the models used in analysis. In other words, there is not the structural chasm between analysis and design that exists in the traditional structured approach. In some ways, OO development is more difficult to learn, primarily because there are many different models that are used to specify and design a system. However, once the models are understood, development is a smooth process that easily flows from analysis to design to programming. The objective of teaching OO development should be to bring students to a level or expertise that they understand and appreciate the gracefulness of these transitions.

One of the biggest problems with the way that most books present UML and OO development, as well as many instructors, is that they try to teach all of the sophistication of each model all at once. This normally overwhelms students. Not only are there many models to learn, but each has its own complexities and sophistication that is almost impossible to understand the first time through. One of the benefits of separating OOA from OOD is that very simple versions of the UML models can be used (and taught) for OOA. Then during the teaching of OOD, more of the complexities of the models can be added. Notice in the following sections how the models begin with the most simple and expand into more complexity as we move through a development iteration.

4. UNDERSTANDING AND TEACHING OOA WITH UML REQUIREMENTS MODELS

We use the term analysis in the traditional way. Analysis activities are those activities which determine the user's requirements and document them with narratives and models. Sometimes we refer to these activities as requirements definition. The basic objective of requirements definition is "understanding" – understanding the users' needs, understanding how the business processes are carried out, and understanding how the system will be used to support those business processes. In object-oriented development, system developers use a set of techniques, tools, and models to discover, understand and specify the requirements for a new system.

There are five interrelated models that can be used to define system requirements. Three of these models, the use case diagram, use case detail documents, and the system sequence diagram are used to describe the processes for the new system. Two models, the domain class diagram and statechart diagrams are used to describe the requirements relating to the data storage and structural portion of the system.

To illustrate how these models work together to specify the system requirements, let's use a common system that has been used in many previous examples, a video rental store. Assuming that we have all rented videos before, we will minimize the case description.

Based on the UP, the first iteration is the inception phase. The objective of the inception phase, and iteration, is do establish the business case and the project feasibility. We may do a little modeling, but usually only partial models with the use case diagram and possibly the domain class diagram are built. The details are not included. From inception, then we move to the first iteration in the elaboration phase.

One of the benefits of the UP, both for the development of real systems for teaching

UML is that the iteration approach allows us to limit the scope of the new system. For teaching purposes it is almost always necessary to limit the scope while students are learning. In the UP, it fits nicely to indicate that we will focus only on a few core use cases for the first iteration. Typically developers try to identify the "core" processes and develop those use cases through analysis, design and programming.

Use Case Diagram

The objective of use case modeling is to identify and define the business functions that require system support. The entire UP development process is based upon finding the use cases. This approach is called a use-case driven approach, and you will see that it drives the entire process (Jacobsen, 1992). We start the development of a use case diagram by identifying those people (or other systems) that will use the system. Those users are called actors. To ensure that we have the right level of abstraction, we will define actors only as those people (or things) that have direct contact with the automated system boundary. We emphasize that point by saying actors must have "hands." Next identify those core business functions done by each actor. A use case is a response that is done within the system in response to a request or input by a system user.

Figure 3 shows the actors and the use cases that we will include in the first iteration. We recognize that the video store system will need more use cases, but for the first iteration this will suffice. As you can see from the figure, we have decided that we need to have use cases to check out and return videos. We also need use cases to add movies to inventory. Finally we also need to add customers to our customer file. In later iterations we may expand the current use cases to be broader, such as removing movies and changing customers, but we keep it simple for the first iteration.

We emphasize that the actors in the diagram are those that are actually working with the system. Hence the checkout clerk is using the system to check out the movies for a customer. Since the end customer does not actually have contact with the automated system, it is not identified as an actor.

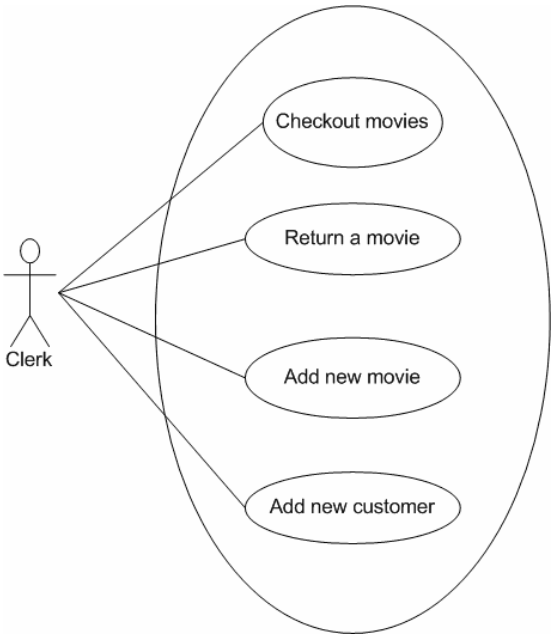


Figure 3. Use case diagram for Elaboration Phase first iteration.

Use Case Detail Model

We identify this as separate from the use case diagram to emphasize that the use case model is only a high-level scoping model. It does not provide enough details to accurately describe the steps in the business processes and the required system responses. The use case detail model looks inside the oval of a use case to describe what is happening within the confines of a use case. One way to describe the internal steps is with a simple narrative description. Another way is to use another UML diagram called an activity diagram.

An activity diagram is a type of workflow diagram that is used to describe the sequence of steps that make up the use case. An activity diagram is made up of ovals, representing activities, and connecting lines, representing the flow from activity to activity. Vertical boxes, called swimlanes, are used to identify which actor does which activity.

Figure 4 is an example of an activity diagram for the *Add new movies* use case. Remember during analysis we are focusing on understanding the requirements of the new system. Essentially we are defining the steps in the business processes that interact with the system. Therefore this activity diagram only has two swimlanes, one for the

actor and one for the system. At this point we do not try to describe what is going on inside the system, just that it needs to do something to respond to the actor.

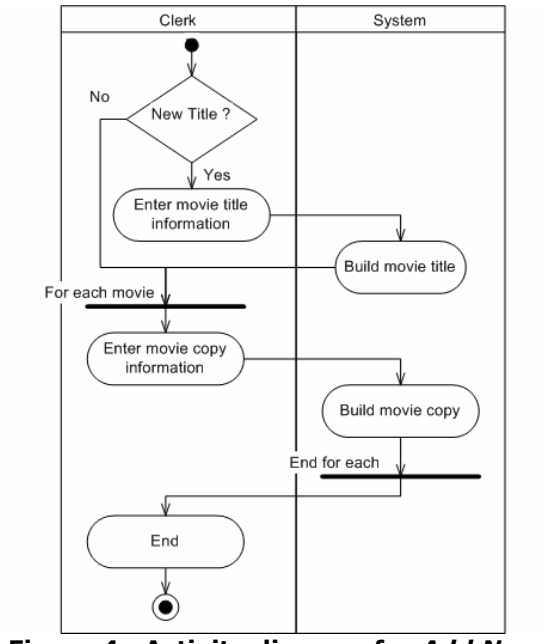


Figure 4. Activity diagram for *Add New Movie* use case.

One benefit of this approach is that the student is learning to focus on the business needs and the user activities. It also teaches about activity diagrams in a very simple context. At this point we expect the students to understand a use case diagram and a simple activity diagram. Students should also see the very close relationship between the two.

Problem Domain Class Diagram

Conceptual data modeling is a complex subject that usually takes several weeks in a database class. As appropriate, we build on the learning obtained in other classes. An important point to make for requirements is that the focus is on those real world things about which the system needs to maintain information. The term "problem domain" is used to describe those items associated with the business need or problem. We emphasize that for analysis the class diagram focuses only on the problem domain. Later on the design class diagrams will include other system objects and will become more complex. Students generally understand that the development of the class diagram is a requirements activity. It identifies the information requirements of the new system.

Figure 5 is an example of the domain class diagram for the video store. One popular technique to building the domain model is by using a "noun search" algorithm. The detail use case narratives can be searched to find those nouns that reflect objects that require class definitions.

In many information system programs, a database class is also offered and often taken concurrently. Hence in many cases students will already have been exposed to conceptual data modeling. Again, we emphasize that for this first iteration both the number of classes identified and the attributes within each class is kept simple. Later iterations will add complexity. However, even in its simplicity, there are numerous concepts that need to be understood. It is appropriate to teach such concepts as relationships, cardinality, inheritance, and even association classes.

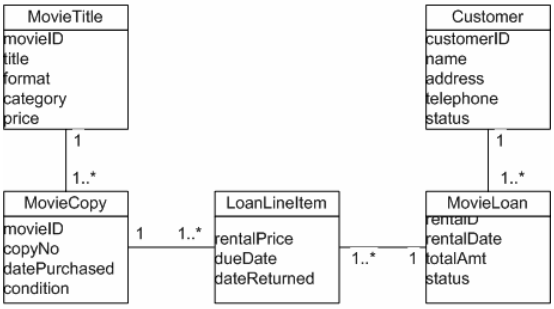


Figure 5. Domain class diagram for video rental store.

The class notation used is the simpler UML notation, which does not include the method compartment. We can discuss and illustrate method names in examples, but we do not require students to learn or understand details of methods. Obviously method specification is a design activity and not a system requirement.

From here the next point is what kind of detail information needs to be captured about the real world things that are included in the domain model. One type of detailed information is the properties or attributes that carry information about the objects. Another piece of information concerns the relationships of objects to other objects. Both of

these pieces of information are maintained in the domain class diagram. Another type of information may be the different status conditions of the objects. In the real world the real objects have different status conditions. The system may need to track those status conditions. This leads us to the final UML requirements model, the statechart diagram.

System Sequence Diagram

Typically at this point we will ask the students, "What about the input and output data? Where does it happen and what is it?" If you have the students go back and review the use case diagram and the activity diagrams, generally they identify the locations of inputs and outputs as the flow between the actor and the system on the activity diagram. So the activity diagram helps us identify the points that data must be entered and viewed, but it does not describe it. UML has another diagram that enables us to describe this flow of information, called a system sequence diagram.

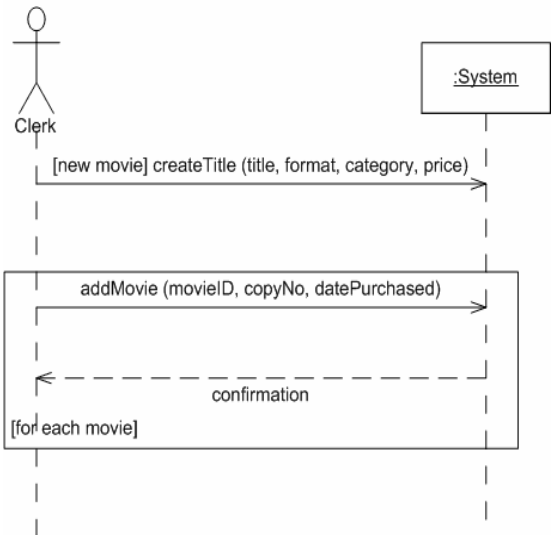


Figure 6. System Sequence Diagram for the *Add new movies* use case.

A system sequence diagram is simplified version of the more complex sequence diagram. However, at this point we keep the diagram simple and only present the basic concepts that are needed for requirements specification. Figure 6 is the system sequence diagram for the use case *Add new movies*. Notice that structurally it is very similar to the activity diagram. There are two components, the clerk and the system. Each has a lifeline, represented by the vertical dashed line, which is similar to the swimlane. The arrows show the movement of data. An arrow, along with its descriptor, is called a message. The message name describes the action requested by the actor, and the parameters describe the data that is being passed. Return messages are shown as dashed lines with only the data being returned as the message descriptor. The sequence of messages flows from top to bottom down the lifelines of actor and system.

At this point we have accomplished two goals. We have taught the students how to define user requirements based on business processes. We have also introduced and taught simple versions of four UML models. Students should not only have a good feel for doing requirements definition, but they should begin to see the elegance of using a set of interrelated models to do specifications.

Statechart Diagram

Many developers consider statechart diagrams to be an optional model for business systems. Statechart diagrams inherently are rather complex. They are a critical component for the definition of complex real-time systems. However, for many business database type systems, they are not as critical. Especially while the students are just beginning to grasp the ideas of OO modeling, we keep the examples and use of statecharts simple. As indicated earlier, for business systems just use statecharts to help track the various status conditions of the more complex objects.

Figure 7 is an example of a statechart for a movie copy. For this simple object class, we only track two conditions, Ready for checkout, and Checked out. A simple statechart can be used to show the relationship between these status conditions.

Note in the figure the ovals represent the states. The arrows are the transition between states. The states describe the various status conditions, and the transitions capture information about the events or messages that cause the object to change from one state to another.

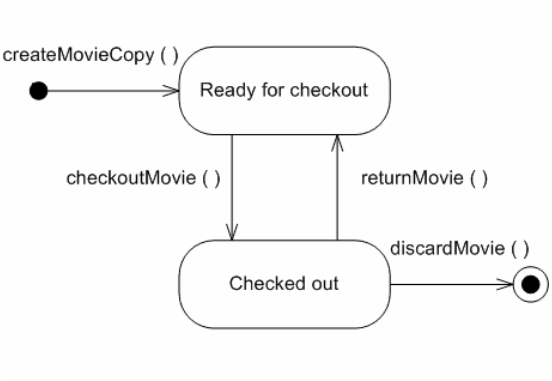


Figure 7. Statechart for Physical Movie Class.

OOA Conclusion

At this point we have a set of five models that captures the system requirements. Students should see how these five models work together to provide an integrated, multiple-view description of the requirements as identified by the users. A review of the five models and some comprehensive homework examples will help them get the big picture. As indicated earlier, it is a good idea to keep the difficulty of each model simple. Since students are learning five new models, it is just as important that they see the interconnections between the models as it is to learn the models. The complexity of each model can be added later as they become more proficient and more skilled. Also, it is important to always emphasize that requirements models are not completed in detail before design and implementation begins. They are always dealing with a subset of requirements for one iteration. That is another reason why simple examples understood in detail are better than one large example that represents the entire system.

5. UNDERSTANDING AND TEACHING OOD WITH UML DESIGN MODELS

A problem with many systems analysis and design courses is that they teach only the UML models and not the processes of development. This is especially true with object-oriented design. Very few programs actually teach how to do object-oriented design. However, thinking back through what we have learned in the preceding section, it should be clear that it is very difficult, if not impossible, for students to understand how to write object-oriented programs utilizing only the analysis models. Many information systems programs have 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 there are many benefits to doing more formal OO design.

Introduction

The purpose of system design is to bridge that gap between the requirements models and the program code and database. One benefit is that 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. In addition, teaching design also gives the students more practice with the models, thus raising their level of proficiency with the models.

Another benefit of teaching object-oriented design is that understanding good design principles is a necessary skill for systems developers. 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 is the basis of all the work that has been done in the development of OO system design patterns.

Interestingly enough, taking time for design also 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 very 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.

The Design Models

Since the purpose of object-oriented design is to bridge the gap between the analysis models and the code, it is helpful to start teaching design by reviewing the elements of an object-oriented system. In other words, the students need to have some experience writing object-oriented programs and understand classes, instantiating objects, methods, and method signatures. Given that to write an object oriented program, a developer needs to have thought about the necessary classes, the methods in those classes, and how the objects invoke methods of classes to carry out some system activity.

Given that a student understands the components of an object-oriented program, it is a straightforward step to identify the inputs to writing code. The inputs should be (1) a set of classes, (2) a description of the object interactions between objects that must occur to execute a given function, and (3) possibly some pseudocode or algorithm specifications for the methods.

The first input, i.e. a set of classes, is described using a design class diagram. Included in the design class diagram are both the attributes and the method signatures for those classes. The second input, i.e. the interactions, is described using interaction diagrams, either detailed sequence diagrams or collaboration diagrams. The interaction diagrams are organized around the use cases, just as the system sequence diagrams are. They are more detailed than system sequence diagrams. The system swimlane is replace by all of the individual objects that interact to carry out a use case or scenario. The messages, which are considered interactions, in reality invoke the methods of the participating objects. The third input, i.e. pseudocode, is described either just with simple pseudocode scripts or as action expressions in a statechart. The use of a statechart is still optional, even during design, but in some cases can add additional insight into the design.

Since the design models, especially the design class diagram, are different than the analysis models, we first illustrate each model with its important components. After showing the models, we explain the process of doing design.

The Design Class Diagram: The following figure is an example of the video system Design Class Diagram. Notice that it is very similar to the domain model developed during business modeling discipline activities. Note that the diagram is an extension of the problem domain class diagram developed during analysis. There are four major additions we will discuss.

First, the attributes have been defined more precisely by the addition of type information. Visibility information can also be added. Default visibility for attributes is invisible or private, meaning values cannot be seen by outside objects.

Second, method signatures have been added. Method signatures will include visibility, method name, parameter types, and return types. The DCD usually does not include constructor methods, accessor methods, or mutator methods unless there is some specific uniqueness that should be identified.

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 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.

Finally, we have added another class called the UseCaseController class. This new class acts as a controller class in that it is the focal point between the problem domain classes and the outside environment. In fact, the addition of the UseCaseController class is an illustration of the application of a design pattern called the controller pattern or a façade pattern.

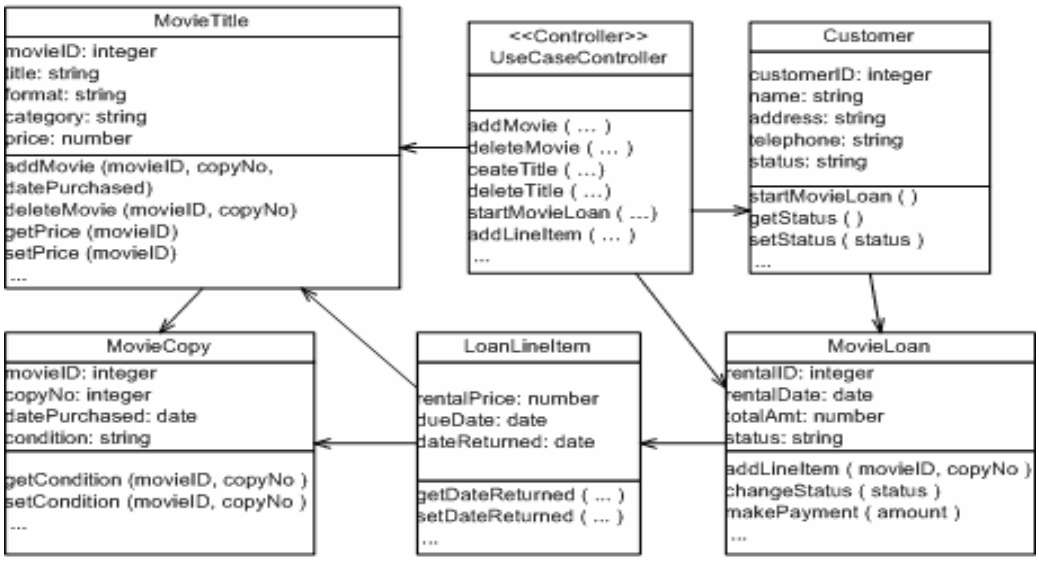


Figure 8. Video System Design Class Diagram

Detailed 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. Obviously, the process of developing interaction diagrams is the foundation of OO system design.

There are two types of Interaction Diagrams that 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. 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.

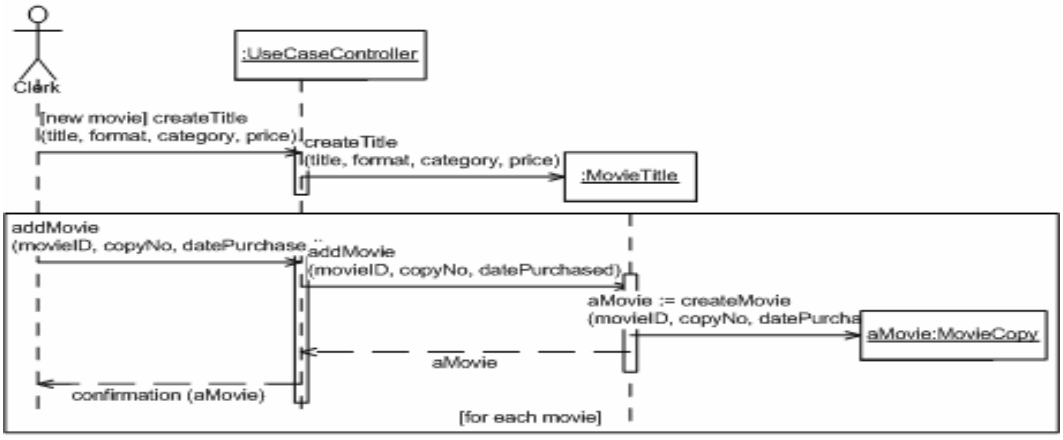


Figure 9. Sequence Diagram

Figure 9 presents an example of a sequence diagram and a collaboration diagram for the use case *Checkout movies*. In a sequence diagram, each object has a life line (dashed vertical line connected to the bottom of the object). The UseCaseController object also has activation lifelines, illustrated by a long, narrow vertical rectangle on the lifeline. An activation lifeline indicates that the object is executing during that period. Each message has a source object and a destination object. The order of the messages is read top to bottom. In a collaboration diagram, each pair of communicating objects is connected by a link that serves as the communication mechanism between the objects. Again each message has a source object and a destination object. The order of the messages is indicated with sequence numbers.

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 8 and the messages in Figure 9, you will note that there is a message *addMovie (movieID, copyNo, datePurchased)* in the sequence diagram going to the MovieTitle object, and there is a corresponding method *addMovie (movieID, copyNo, datePurchased)* in the MovieTitle class in the DCD.

The Design Process

The design class diagram discussed above provides the core structure for the new system. However, we recognize that an object-oriented system has many more classes than those that appear in the problem domain class diagram. Figure 8 only illustrated those classes that are derived from, or closely related to, the problem domain classes. Other classes, such as graphical user interface classes, database access classes, and possibly other utility type classes might also be required in a new system. Consequently, one of the first principles, and design patterns to teach is the multilayer design pattern.

Multi-Layer Design Pattern: An important design pattern 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. 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.

Figure 10 illustrates the components of a three-layer design. In Figure 1, we showed the various iterations of the UP. In the first iteration of the Elaboration Phase, we will select a few core use cases and develop a three layer design for those use cases. In other words, there will be view layer classes, domain layer classes, and data access layer classes, but only for those few selected use cases. All three layers are developed as part of a single UP iteration. In this case, the first UP iteration will produce the classes in these three layers.

Within a UP iteration, we will do some micro-level iterations, or design passes. By that we mean that in the first pass, we design the domain layer. In the second pass the view layer is designed, and in the third pass the data access layer is design. Not only is this multiple pass, micro-level iteration a good way to do design, it is also an excellent way to teach design to new students.

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 low complexity namely *Add new movie*. The objective of this design is to determine the objects that must collaborate together to execute this use case within the system.

Recall that one UP iteration might include just a few of the core use cases for the system, and after we complete the requirements modeling for those use cases (the OOA part) we will begin immediately with the design (OOD) and construction (OOP).

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

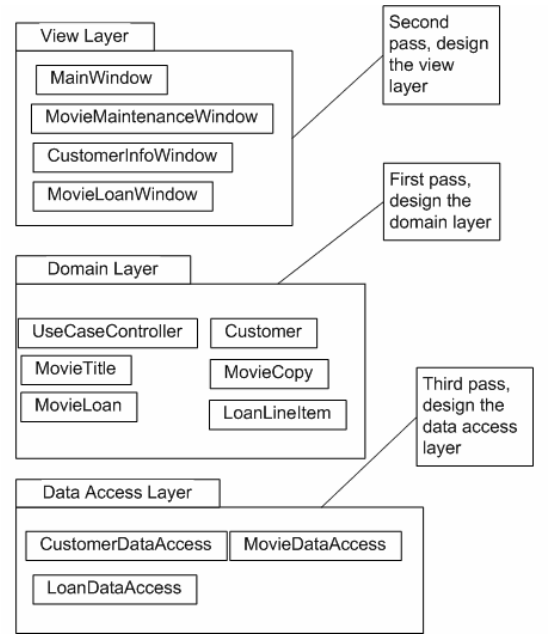


Figure 10. Multi-layer Architecture

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 *MovieTitle* and *MovieCopy* classes. We begin a sequence diagram by placing the Clerk actor (from the use case), and *MovieTitle* and *MovieCopy* objects on the diagram. At this point, we will also add a new object, one that is used to represent the system as a whole. This additional object will be called *UseCaseController*. It will serve as a kind of switchboard to distribute messages that come from external points, called a controller.

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 purpose of this preliminary iteration is to design the interacting objects and messages from the domain model. The results of this first iteration were shown in Figure 9.

Micro-level Iteration 2: The purpose of the second micro-level iteration is to add the other objects that are required from the view layer and the data access layer. These other classes are true design artifacts in that they are created by the designer to make the system work. The view layer classes are derived from the user interface screen layouts that are developed with the user. The system designer, takes those screen layouts, and possibly prototypes, and converts those to GUI classes with buttons, textboxes, graphics, and so forth.

The data layer classes are derived also design artifacts. The design of these classes is dependent on the data structures, including whether a flat file system or a relational database system is being used. If a relational database system is being used, then all the connectivity, SQL statements and result set processing is done in the data layer classes. An example of the final sequence diagram for the *Add new movie* use case is shown in Figure 11. Many message parameters have been omitted to enhance readability.

Information about the messages for this more complete sequence diagram will enable the designer to define the methods for classes in the view layer, the domain layer, and the data access layer. Note how this activity is truly design work, and that it directly supports programming. From a complete sequence diagram, and the corresponding design class diagrams, a programmer should be able to go right to code.

We have not discussed the inclusion of design patterns (Buschman, 1996, Gamma, 1995). In a curriculum that teaches development using models, and one that is use case driven, it is logical to include discussions of good design principles and design patterns. For example, earlier we mentioned one useful design pattern, called the controller pattern, which is used to reduce coupling between the classes of the view layer and the domain layer. Another important principle is that classes in the view layer have visibility to the domain layer, but the domain layer should not "know about" the particular classes in the view layer. There are many principles, such as coupling, cohesion, visibility, protected variations, creator, information expert, and so forth, which can be taught during discussion of creating good

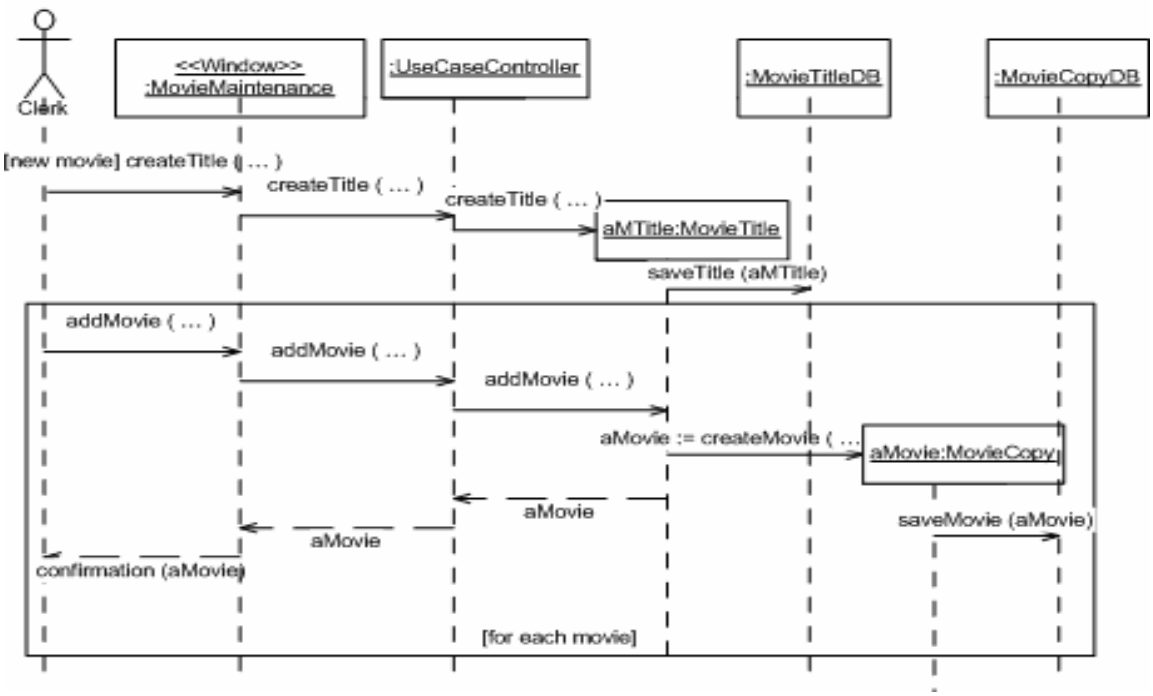


Figure 11. Final sequence diagram for Add new movie.

object-oriented design. Those topics usually are not discussed during either analysis or programming courses

6. UNDERSTANDING AND TEACHING OOP BASED ON OOD

Now that we have discussed the design models developed using the UP, it should be clearer why it is important to integrate the modeling/methods track with the programming track. In order to write programs that are meaningful problem solutions, the students must be taught how to read and interpret these design models just as a traditional structured programmer has always been taught how to read flowcharts, psuedocode, file layout specifications, report layout specifications, and structure charts. Just as structured courses show design documentation for each programming project, OOP courses must show design documentation. Each use case and scenario is documented by a design class diagram showing all problem domain classes required for the use case, a sequence diagram showing all interactions between the objects in the system, and some additional user interface design models showing the layout and components of each form used in the use case. That is how the student first learns the basics of typical design patterns. Programming students are not expected to create the design models. They are only expected to understand and implement the models. The focus stays on programming and testing, but it is all based on clear business system examples and architectures. In this manner, OO programming courses become much more than simply teaching a collection of advanced techniques that are typically not part of an integrated business system example.

There are several clear benefits to this approach. First, students are explicitly taught to implement code based on a documented design. The UML design models will therefore appear to be useful to students, and students will be more likely to be interested in learning the OOA and OOD processes that are followed to create the models. Second, the students will be exposed to design patterns that reflect good OO design solutions. As they implement the design, they will learn the underlying design pattern. Finally, the student will be learning UML in more than one iteration. They use the models and then go back and work with the models again in the Modeling/Methods 1 course. It takes more than one attempt at learning UML and the OO approach for most of us to really understand it.

7. REFERENCES

Bloom, B.S. (Ed.) (1956) Taxonomy of educational objectives: The classification of educational goals: Handbook I, cognitive domain. New York ; Toronto: Longmans, Green.

Booch, G., Rumbaugh, J. and Jacobson, I. (1999A). The Unified Modeling Language User Guide. Upper Saddle River, NJ: Addison-Wesley.

Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996), Pattern-Oriented Software Architecture: A System of Patterns. John Wiley and Sons.

Davis, G. B., J. T. Gorgone, J. D. Couger, D.L. Feinstein, and H.E. Longenecker, Jr. (1997). IS '97 Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems, ACM, New York, NY and AITP (formerly DPMA), Park Ridge , IL.

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

Gorgone, J. T., G. B. Davis, J. S. Valacich, H. Heikki, D. L Feinstein, H. E. Longenecker, Jr. (2002). Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems, ACM, AIS, and AITP.

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

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

Larman, Craig (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.

Martin, J. (1990). Object-Oriented Analysis and Design. Englewood Cliffs, NJ: Prentice Hall.

Rumbaugh, J., Jacobson, I., Booch, G. (1999C), The Unified Modeling Language Reference Manual. Upper Saddle River, NJ: Addison-Wesley.