Teaching Object-Oriented Systems Analysis and Design with UML

Robert V. Stumpf
rvstumpf@csupomona.edu

Lavette C. Teague
lcteague@csupomona.edu
Computer Information Systems Department
California State Polytechnic University, Pomona
Pomona, California 91768, USA

Abstract

The transition to object-oriented software presents a challenge to information systems (IS) educators, especially in the area of systems analysis and design, as familiar structured methods give way to the Unified Modeling Language (UML). This paper summarizes the principal similarities and differences between structured and object-oriented approaches and provides advice about strategies for teaching analysis and design with UML. Analysis strategies include: capturing the content and structure of inputs in the use case narratives, constructing the domain model one use case at a time, and expressing pre- and postconditions for the contracts in terms of the domain model. Strategies for teaching object-oriented design include: working one use case at a time, and starting with three basic design patterns.

Keywords: object-oriented analysis, object-oriented design, teaching UML, transition to objects

1. INTRODUCTION

Object-oriented software development has been in wide use for some time. There is now a stable, industry-standard notation for object-oriented analysis and design models – the Unified Modeling Language (UML) (Fowler 2004), (Rumbaugh (2005). There are also explicit, teachable methods for object-oriented analysis and design (Stumpf 2005). Yet universities have been slow to follow industry's move to objects. Few information systems programs offer courses in object-oriented analysis and design methods, and the number of curricula requiring or focused on these methods is still quite small.

Perhaps this situation is due to the difficulty (or perceived difficulty) of re-tooling current faculty so that they can comfortably teach the development of object-oriented software. The authors have been teaching object-oriented analysis and design for more than ten years and summarize here what they have learned in order to assist others in the transition from structured to object-oriented methods.

The paper is organized into three major parts. The first of these presents fundamental concepts and models of object-oriented systems analysis, describing the commonalities between it and structured analysis, followed by the differences between the two. The second part of the paper gives a similar treatment of object-oriented design. The third part, based on the authors' experience, offers strategies for teaching systems analysis and design using the UML models.

2. FUNDAMENTAL CONCEPTS OF OBJECT-ORIENTED ANALYSIS

Expressing users' requirements for software through the models of the UML is new. Nevertheless, the goals of requirements analysis and the nature of the task itself imply some inherent commonality with traditional, structured analysis.

 © 2005 EDSIG,

2.1 How Object-Oriented Analysis Is Like Structured Analysis

Like structured analysis, object-oriented analysis benefits from the use of event analysis for system decomposition. It also requires conceptual, or semantic, modeling of the application domain. In both approaches it is important to maintain a clear distinction between essential and implementation models (logical and physical models) as well as to separate the analysis models, which specify requirements, from the design models, which define a software solution.

2.1.1 Event Analysis for System Decomposition: Since at least the mid-1980s, event analysis (McMenamin 1985), (Page-Jones 1988), (Yourdon 1989) has been the preferred technique for decomposing a system into parts which respond independently to external or temporal stimuli. The results of event analysis are presented in an event table, from which the initial system models can be derived. In the structured approach, event analysis identifies a fundamental set of processes. In object-oriented analysis, each event leads to the discovery of a fundamental use case (see Section 2.2.1).

2.1.2 Conceptual (Semantic) Modeling of the Application Domain: In structured analysis, an entity-relationship diagram (ERD) provides the conceptual model of the entities and relationships in the application domain. It is derived by normalizing the data stores in the set of data flow diagrams (DFDs).

In object-oriented analysis, a UML ***domain model*** (Figure 1) plays a role similar to that of the ERD. The two models differ almost exclusively in the graphic conventions and the component names. Both models explicitly depict relationships (***associations*** in the UML). However, implementing the relational model requires foreign keys; these extraneous attributes are unnecessary in a domain model. The UML model incorporates generalization-specialization hierarchies, which are also included in extended entity-relationship diagrams (EERDs) (Teory 1986).

2.1.3 Clear Distinction Between Essential and Implementation Models: Systems analysis describes ***what*** users need but not ***how*** these needs are to be satisfied. Best practices in systems analysis have always stated users' requirements in a way which does not bias the design solution by incorporating details of an implementing technology.

Structured analysis captures users' requirements in a set of essential data flow diagrams, which is supplemented by a system dictionary containing definitions of the essential data flows, essential data stores, and descriptions of the procedures for the essential, primitive transformations. Object-oriented analysis with UML uses a different set of models, as described in Section 2.2.

Figure 1 – Domain model to support the use case Check Out Book.

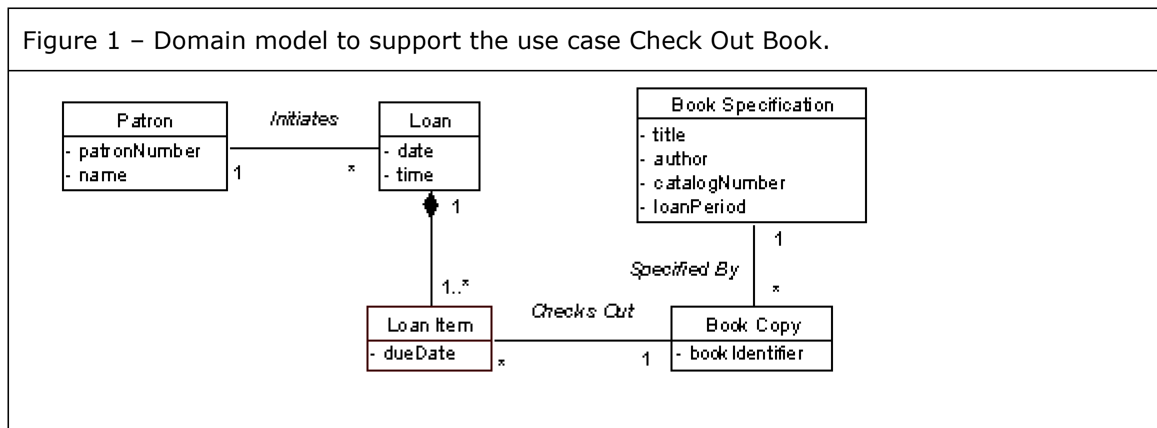
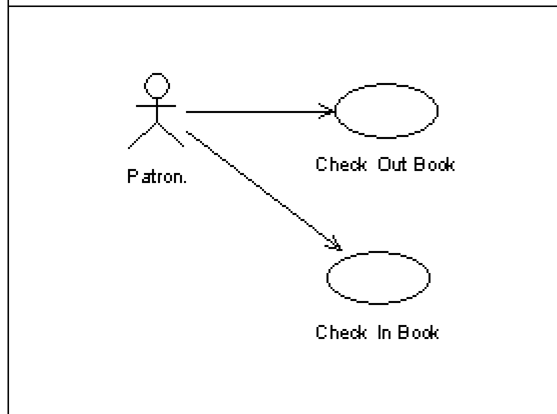


Figure 2 – Partial use case diagram for a public library.



2.1.4 Clear Distinction Between Analysis and Design:

Closely related to the important distinction between essential and implementation models is that between analysis and design. Maintaining the integrity of the requirements models permits the requirements to be traced through design and implementation. This is especially valuable when, as in current practice, development proceeds iteratively. When analysis and design models are obviously different, this distinction becomes easy to see. System developers need to be particularly careful to be aware of and highlight the differences when similar models are used in both activities.

2.2 How Object-Oriented Analysis Differs from Structured Analysis

Except for the event model and the conceptual model of the application domain, the UML models for object-oriented analysis differ from those of structured analysis. This section discusses the most important of these models — use case diagrams, use case narratives, system sequence diagrams, and system operation contracts. More detailed charts comparing the models are contained in the instructor's guide to Stumpf (2005).

2.2.1 Use Cases as the System-Level Units of System Requirements:

In the UML, the use case is the system-level unit for defining requirements. A ***use case*** is the sequence of actions which occur when an actor — a person, organization, or system — uses a system to complete a process. Normally, each use case corresponds to an event. Its structured analysis counterpart is an essential process associated with that event. Use case names are similar to process names in structured analysis — usually a verb followed by an object.

The UML lacks any counterpart of a context diagram. Instead, it has a ***use case diagram*** (Figure 2), which shows all the use cases (unless the system is large) and the ***actors***, equivalent to the external entities of structured analysis, who participate in each use case. Thus a use case diagram is the rough equivalent of Diagram 0. However, it does not show inputs or outputs.

2.2.2 Use Case Narratives for Requirements Specification:

Use case narratives replace the process descriptions of structured analysis as well as the data dictionary definitions of the system inputs and outputs. When well-written, they may be easier for users to understand than the structured models. Expanded essential ***use case narratives*** (Figure 3) describe the sequence of interaction between an actor or actors and the system. They should capture the sequence and detailed composition of the essential system inputs and outputs and also show the expected internal response of the system to each message from an actor.

2.2.3 System Sequence Diagrams for Interaction between the System and Its Environment:

In object-oriented analysis, a set of ***system sequence diagrams*** substitutes for a context diagram, but presents a finer level of detail. The system sequence diagrams (Figure 4) are based on the expanded essential use case narratives. As in a context diagram, what happens inside the system is not shown. In principle, there is a separate system sequence diagram for each message (system input) from an actor to the system. However, if the number of messages is small, one system sequence diagram per use case may suffice. As its name implies, this diagram will show the order in which the messages from the actor occur. The message format is similar to that of a procedure or function, showing the message (operation) name and a list of its parameters. These parameters are the essential data elements of the input.

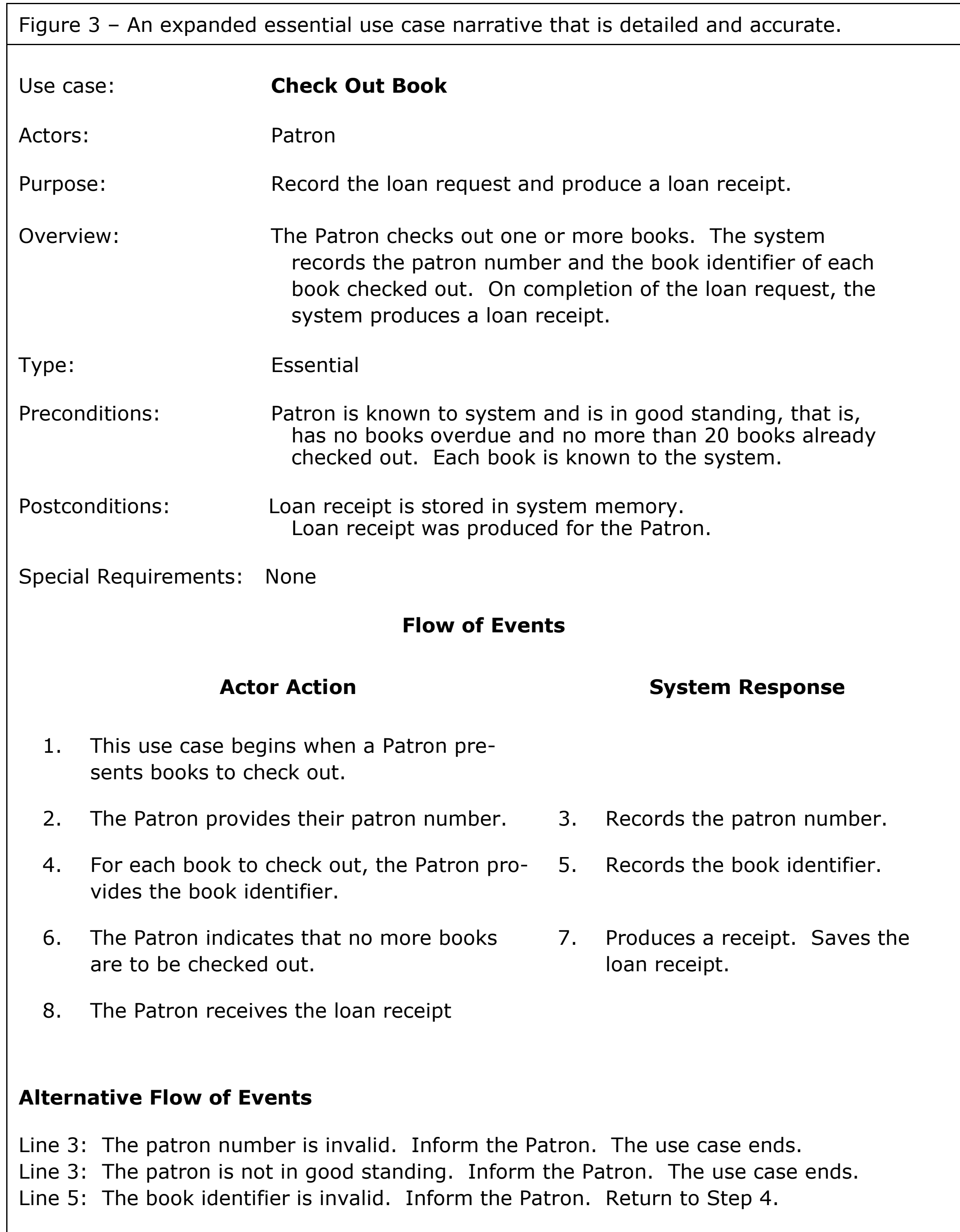


Figure 3 – An expanded essential use case narrative that is detailed and accurate.

Use case: **Check Out Book**

Actors: Patron

Purpose: Record the loan request and produce a loan receipt.

Overview: The Patron checks out one or more books. The system records the patron number and the book identifier of each book checked out. On completion of the loan request, the system produces a loan receipt.

Type: Essential

Preconditions: Patron is known to system and is in good standing, that is, has no books overdue and no more than 20 books already checked out. Each book is known to the system.

Postconditions: Loan receipt is stored in system memory.
Loan receipt was produced for the Patron.

Special Requirements: None

Flow of Events

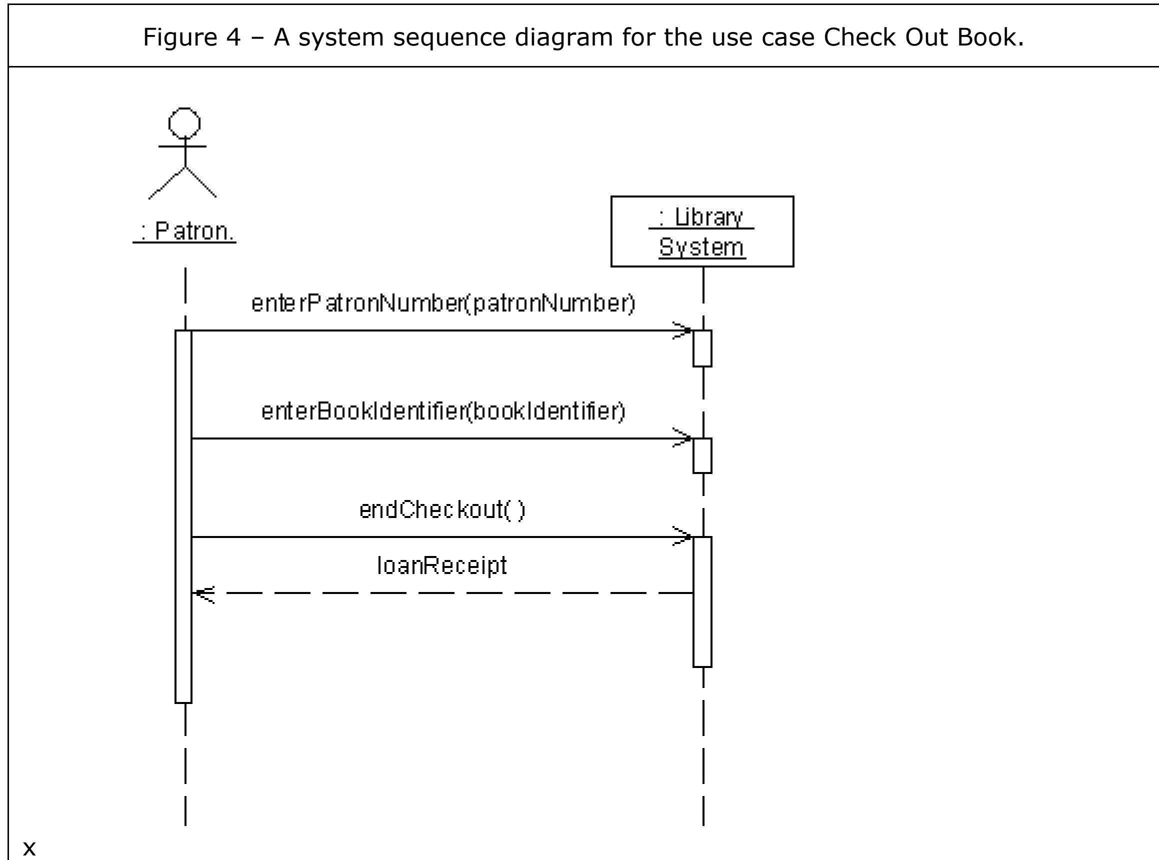
Actor Action	**System Response**
1. This use case begins when a Patron presents books to check out.	
2. The Patron provides their patron number.	3. Records the patron number.
4. For each book to check out, the Patron provides the book identifier.	5. Records the book identifier.
6. The Patron indicates that no more books are to be checked out.	7. Produces a receipt. Saves the loan receipt.
8. The Patron receives the loan receipt	

Alternative Flow of Events

Line 3: The patron number is invalid. Inform the Patron. The use case ends.
Line 3: The patron is not in good standing. Inform the Patron. The use case ends.
Line 5: The book identifier is invalid. Inform the Patron. Return to Step 4.

 © 2005 EDSIG,

Figure 4 – A system sequence diagram for the use case Check Out Book.



2.2.4 Contracts for the Specification of System Operations: Contracts for the system operations (Figure 5) link the UML analysis models to design. They enable the use of a design method known as ***design by contract*** (See Section 3.2.5).

Figure 5 – Contract for the system operation ***enterPatronNumber***.	
Use Case:	Check Out Book
Contract Name:	enterPatronNumber (patronNumber)
Responsibilities:	Verify the Patron and determine that they are in good standing.
Exceptions:	If the patron number is not valid, indicate an error. If the patron is not in good standing, indicate an error.
Output:	None
Pre conditions:	Patron is known to the system.
Post conditions:	A new Loan object was created. A new instance of the association Patron – Loan was created.

When an object-oriented system (or an object) receives a message, it executes an operation with the same name as that of the message. A ***system operation contract*** specifies the response of the operation to a message from an actor, as shown in a system sequence diagram. As an analysis model, it states ***what*** the system must do to respond, not ***how*** the response will be implemented. This is accomplished by writing the contract in terms of postconditions. These postconditions are expressed in terms of the domain model; they state which instances of concepts and associations have been added to (or deleted from) the domain model and which attribute values have been modified.

Thus each system operation contract is based on a message in a system sequence diagram, which in turn is derived from a use case narrative. The preconditions of the contract state what must be true for the operation to execute successfully in order to accomplish the desired system response. The postconditions state the required changes to the state of the domain model as a result of the execution of the system operation.

3. FUNDAMENTAL CONCEPTS OF OBJECT-ORIENTED DESIGN

The UML design models, described below, address the distinctive way in which object-oriented software is organized. Still, designers of both object-oriented systems and those implemented in more traditional procedural languages share some high-level goals.

3.1 How Object-Oriented Design Is Like Structured Design

One goal of software design has always been to specify systems which are easy to understand, to modify, and to maintain. Thus, both structured and object-oriented approaches share several principles of good design. These principles include layered system architecture, conservation of data flow, coupling and cohesion as design criteria, and the need to specify procedures or operations correctly and completely.

3.1.1 Layered System Architecture: Layered system architecture has long been a best practice in software design, dating at least as far back as the mid-1960s. This architecture provides interfaces which help minimize the effect of changes in one layer on the other layers. At a minimum, it implies separate layers for the user interface, the application programs, and the data (or objects) stored in a data base.

3.1.2 Coupling and Cohesion as Design Criteria: Similar considerations apply at the level of the program units. There, the criteria of coupling and cohesion also help the designer to minimize the impact of change. Coupling addresses how tightly the interconnections between program units are and thus how likely change is to propagate within the system. Cohesion addresses how strongly focused and relevant the features are within a program unit. In the case of object-oriented software, these features include attributes as well as operations.

3.1.3 Conservation of Data Flow: Conservation of data flow is an important principle of the structured approach. It assures that data does not magically appear or disappear, that the outputs of each process or module can be derived from its inputs, and that there is an unbroken path by which each data element of an input can flow to where it is used in the system. The designer of object-oriented systems has the same concerns when specifying internal message flows in the interaction diagrams (See Section 3.2.3).

3.1.4 Specification of Operations: Regardless of the program structure, each operation or procedure must be specified completely. Such a specification includes all the parameters of the operation with their data types as well as the algorithm or procedure for the operation and how to handle potential exceptions.

3.2 How Object-Oriented Design Differs from Structured Design

The paradigm shift from structured to object-oriented methods is most evident during design. This is largely due to the difference in the way that object-oriented software is organized — a difference which affects the designer's way of thinking as well as the design models themselves.

The most significant changes are in the units of program structure and the way in which they communicate with each other. The principal UML program design models are interaction diagrams and class diagrams. Moreover, the object-oriented approach incorporates the method of ***design by contract*** and uses patterns extensively. Similarities to structured methods will be noted in the course of the discussion.

3.2.1 Objects as the Units of Program Structure: In structured design the unit of program structure is a module, typically either a function or a procedure. In object-oriented systems, the unit of program structure is an ***object***, which encapsulates both its attributes and the related operations.

3.2.2 Peer-to-Peer vs. Hierarchical Communication: Objects collaborate to carry out the responses of the system. They communicate with each other by sending messages requesting services from other objects. By contrast, in the procedural programming environment of structured design, the structure of communication is hierarchical, and its tone is imperative.

3.2.3 Interaction Diagrams Model Dynamic Program Structure: The principal activity of object-oriented design is to assign operations to the objects to which they belong. This assignment involves two types of models of program structure — ***interaction diagrams*** and class diagrams. The former, as their name implies, show the sequence of internal messages and responses which are triggered by each message to the system from an actor. There is typically an interaction diagram (Figure 6) for each system operation.

3.2.4 Class Diagrams Model Static Program Structure: A ***class diagram***, on the other hand, shows the static structure of the program. It is like an expanded domain model — essentially, each class in the diagram replaces the corresponding concept in the domain model. This design class diagram (Figure 7) summarizes the results of the design process by showing the operations assigned to each class.

3.2.5 Design by Contract: The object-oriented design process is driven by the system operation contracts produced during analysis. The metaphor of the contract implies that the object receiving a message has an obligation to achieve the postconditions of the contract, provided that the object sending the message assures that the preconditions of the contract are satisfied.

Figure 6 – Collaboration diagram for the system operation ***enterPatronNumber***.

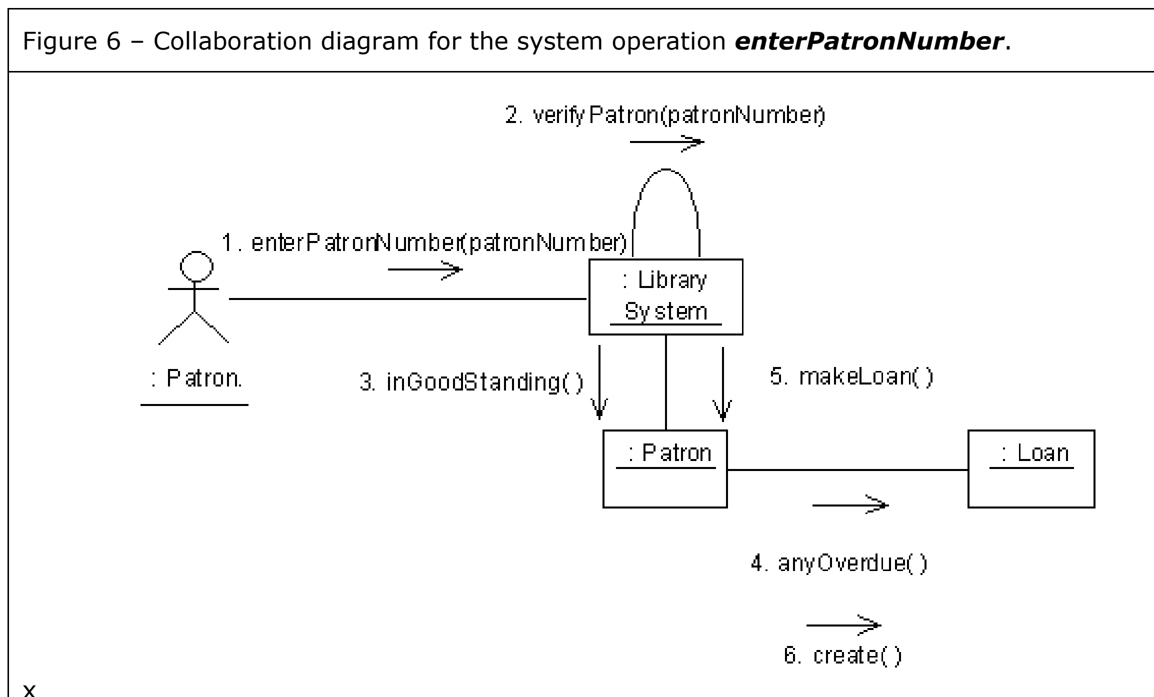
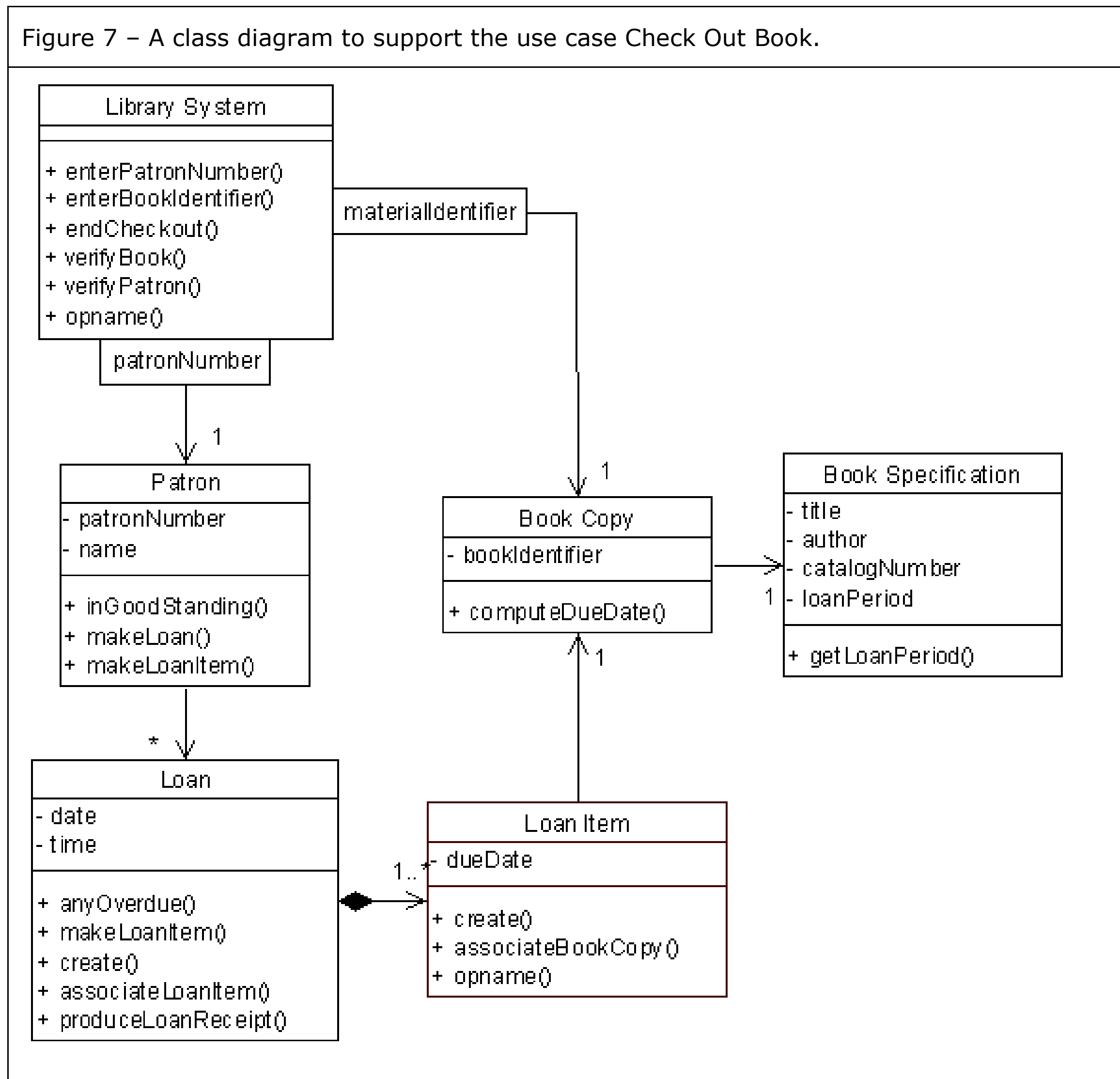


Figure 7 – A class diagram to support the use case Check Out Book.



Thus, each system operation must first check to see that all the preconditions of the contract are fulfilled and then must cause all the postconditions to be true. Clearly the quality of the design is highly dependent on the care with which the contracts were formulated.

3.2.6 The Use of Design Patterns: In assigning operations to objects, the designer is guided by ***patterns***. Collectively, these patterns record previously invented good solutions to design problems. A pattern states the problem, names the solution, and provides advice about using the pattern. An example is the Creator pattern, which solves the problem of which object should request a class to instantiate a new object. (One might view transform analysis and transaction analysis as patterns for mapping data flow diagrams into structure charts.)

4. TEACHING STRATEGIES

This discussion of teaching strategies for object-oriented analysis and design is presented using a public library system as an example.

4.1 Teaching Strategies for Analysis

The first step, event analysis, is the same for both structured and object methodologies. This step is important for identifying

Table 1 – Partial event table for the public library example.

Event Description	System Input	Actor Providing Input	System Output	Actor Receiving Output
Patron Checks Out Book	Loan Request	Patron	Loan Receipt	Patron

the use cases. An event table for one event for the public library example is shown in Table 1. Once an event analysis has been completed, a use case diagram (Figure 2) is drawn to provide a view of all the use cases on a single page. (Although relationships between use cases can be shown in the diagram, in our opinion these refinements are not valuable to beginning students and so are best omitted.)

The use case is the fundamental unit of analysis; therefore it is important for students to focus on a single use case at a time when building each of the analysis models. Our example here deals with a single use case – Check Out Book.

4.1.1 Capture the Content and Structure of the Inputs in the Expanded Use Case Narratives: The core of object-oriented systems analysis is the expanded essential use case narrative (Figure 3). Thus the instructor must strongly emphasize this segment of the process. A properly written use case that is detailed, complete, and accurate sets the stage for creating the remaining artifacts for requirements definition – domain model, system sequence diagram, and contracts.

One secret to success at this stage is to pay special attention to the data. For example, the terms loan request and loan receipt in the event table are not fully defined. Studying the actual loan receipt is a first step. For our example the loan receipt might be something like the one in Figure 8.

Using the following simple formula to determine output, we have:

Output = Input + Stored Data + Computed Data

Often it is useful to develop a simple table to help students to be sure they have not omitted any data. Such a table is shown in Table 2.

Figure 8 – Sample system output for the use case Check Out Book.

```
                              Loan Receipt
************************************************************************
                         Any City Public Library
                     Friday, May 13, 2005, 04:30 PM
************************************************************************
Patron:     29483761
            Louise Forbes
************************************************************************
Identifier   Title                         Author               Due Date
************************************************************************
1290349      Book of Running               Bigfoot, Amy         Jun 3
1340329      PC Computing- May 2005                             May 27
4203921      Who Did It                    Mystery, Writer      Jun 3
************************************************************************
To renew see website at: http/www.anycitylibrary.com
```

Table 2 – Analysis of loan receipt.

System Outputs	
Data	**Source**
Date & Time of Checkout	System
Patron Number	Patron
Patron Name	System
Book Identifier	Patron
Title	System
Author	System
Due Date	Computed

From this analysis, a use case narrative (Figure 3) can easily be constructed. Two of the more critical components are the pre- and postconditions. The patron's name must come from the system, thus the patron must be known to the system prior to the checkout. At this time the simple business rule can be introduced. The rule that no one may check out books if some are overdue or if the patron already has 20 books checked out is useful to show that the system must keep the essential data for the loan receipt in system memory. Also the book's title and author must be known to the system prior to the checkout. Listing the patron's number and book's identifier as known to the system in the preconditions insures that these requirements are met in later steps.

The postconditions are just as critical. The student can now see that to verify a patron's library standing, all the essential data in the loan receipts must be kept. To keep technology out of the requirements statement, the word *produced* is used.

Developing the use case narrative is often the most difficult part for the student. The authors strongly recommend the two-column format shown in Figure 3 for the Flow of Events section. This clearly separates what the actor does from what the system does and minimizes students' errors. Although the Actions 1 and 8 are boilerplate, the student must carefully consider the individual data elements in the messages from the actor as well as the corresponding system responses. Note that Actions 2 and 4 follow directly from system output analysis.

Since many business transactions have the hierarchical structure of a header and many detail lines, this is a good type of example for class lectures and exercises. Our library checkout use case satisfies this recommendation. To terminate the entry of repeated detail lines, a common solution is to use a separate message, as shown in Figure 4.

Lastly, the student must complete the exceptions, as illustrated in Figure 3. Only business-level errors are considered.

Based on the use case narrative, the system sequence diagram (Figure 4) may now be completed. It is nothing more than a graphic illustration of the sequence and structure of the messages sent to the system. The emphasis is on the data. Note that patron number, book identifier, and loan receipt come from the output analysis. At this time the definition of the input and output messages and their data is complete.

4.1.2 Construct the Domain Model One Use at a Time: Next, the student must model the data to be stored in the system by producing a domain model (Figure 5) for the use case. Since this step is virtually the same as producing an entity-relationship diagram, only the result will be shown here. (However, it is important to remember that foreign keys are not used in object analysis.) The UML graphic notation is slightly different, but, even for one unfamiliar with UML, a domain model is just as readable as an ER diagram. One must just get used to the terms ***concept*** and ***association*** in object parlance.

4.1.3 Express the Contracts in Terms of the Domain Model: The pre- and postconditions listed in the use case narrative must now be refined.

The preconditions require that the **Patron** and **Book** pre-exist. They are expressed rigorously in terms of the domain model. For example, in object terms, the specified **Patron** object pre-exists as well as the specified **Book Copy** object.

Figure 9 – Contract for the system operation ***enterBookIdentifier***.	
Use Case:	Check Out Book
Contract Name:	enterBookIdentifier (bookIdentifier)
Responsibilities:	Record the Book that is being checked out.
Exceptions:	If the book identifier is not valid, indicate an error.
Output:	None
Pre conditions:	Book Copy is known to the system.
Post conditions:	A new Loan Item object was created. A new instance of the association Loan – Loan Item was created A new instance of the association Loan Item – Book Copy was created.

Then the postconditions are expressed with the same rigor. In this case, both a **Loan** object and the **Loan Item** objects must have been created. But, just as importantly, the student must also realize that the associations are a critical part of the model. Thus the three associations, **Patron – Loan**, **Loan – Loan Item**, and **Loan Item – Book Copy** must also have been created.

Note that all these pieces are coordinated and contribute to the requirements. The two contracts are shown in Figures 5 and 9. This completes the requirements definition for the Check Out Book use case.

4.2 Teaching Strategies for Design

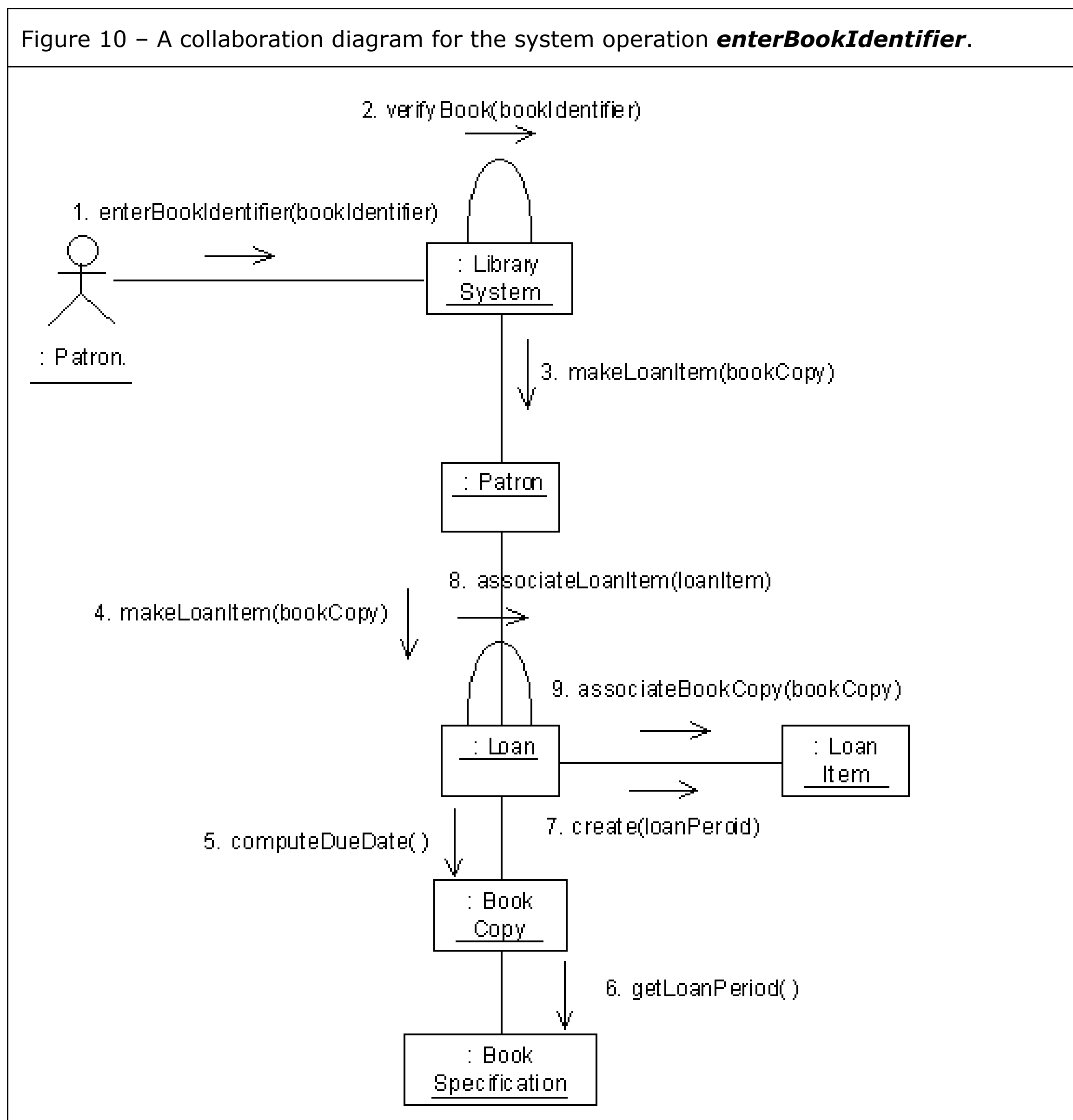
The UML analysis models do not necessarily depend on the implementing technology. Therefore it is appropriate to defer a presentation or review of object technology until the discussion of object-oriented program design. Teaching system design involves user interfaces and database design. Since these two topics are not different when using object modeling, they will be omitted in this discussion.

The major steps in program design are: developing an interaction diagram for each system operation and deriving a class diagram from the interaction diagrams. Since the assignment of data types to all of the arguments and returns of the operations is more of a programming problem, it is not discussed.

4.2.1 Use Patterns to Produce the Interaction Diagrams One Use Case at a Time: It has been said that the most important design task in object-oriented design is assignment of the responsibilities to classes. The best way to approach this process in the classroom is to emphasize patterns. In a beginning course, three common patterns may suffice. The Façade pattern provides a new class to represent the system. Its function is to receive messages from actors and request objects inside the system to carry out the system's response. In completing the interaction diagrams, we recommend using only the Expert and Creator patterns to assign responsibilities.

Expert Pattern for the system operation *enterPatronNumber*: In this system operation, the only behavior requiring attention is **inGoodStanding**. The Expert pattern merely says which object is best suited to do this. In general, this means the object that knows what is necessary to do this task. For example, **inGoodStanding** could be assigned to the object **Patron,** as it knows all of the books checked out and if they are overdue. Note that this information is in the concepts **Loan** and **Loan Item**.

Figure 10 – A collaboration diagram for the system operation ***enterBookIdentifier***.



Creator Pattern for the system operation *enterPatronNumber*: The Creator pattern states that the creation of an object should be requested by an object that has the knowledge to do so. The two candidates for creating **Loan** are **Patron** and **Book Copy**. Since **Book Copy** is not associated directly to **Loan**, **Patron** is the logical candidate. The resulting interaction (collaboration) diagram is shown in Figure 6.

One strategy for teaching is to draw this diagram with no name for the **Patron** object. Then point out the similarity of this diagram with the domain model. Also point out that two actions are necessary in this system operation in order to satisfy the postconditions. In Figure 5 the postconditions of the contract are: a new instance of **Loan** was created, and a new instance of the association **Patron** – **Loan** was created.

Expert Pattern for the system operation *enterBookIdentifer*: In this operation only the computation of the due date is required. In the domain model, this value is an attribute of the **Loan Item;** however, only the concept **Book Copy** has access to the data

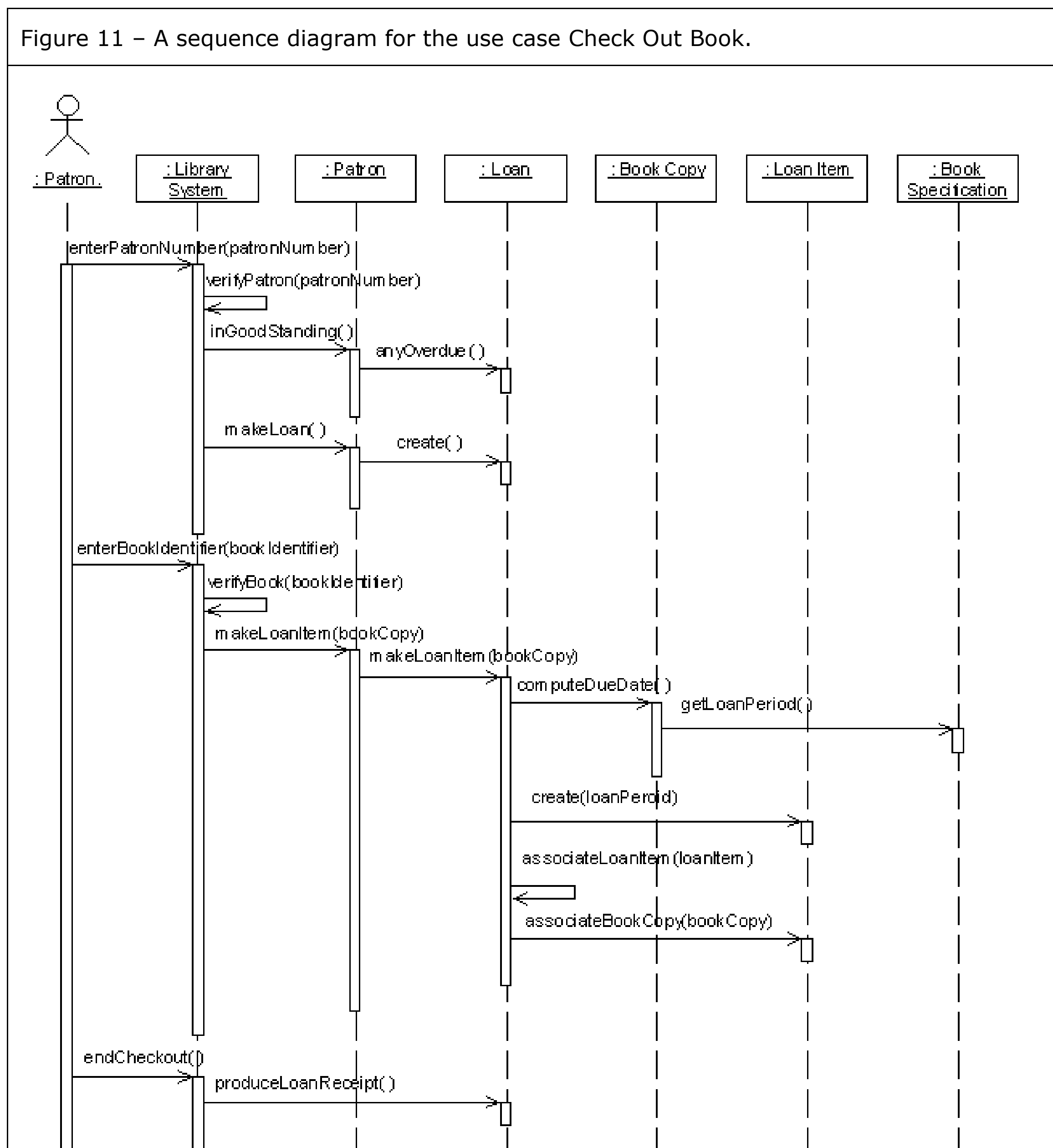


Figure 11 – A sequence diagram for the use case Check Out Book.

required to do this computation. Thus, ***computeDueDate ()*** is assigned to **Book Copy**, which references both **Book Specification** (where ***loanPeriod*** is stored) and **Loan Item** (where ***dueDate*** is stored).

Creator Pattern for the system operation *enterBookIdentifer*: Any time a composite object (**Loan** – **Loan Item**) is used, the composite should create the component objects. This means that **Loan** must create **Loan Item**. A simple name for this operation is ***makeLoanItem ()***. In order to be clear, we believe it is best not to abbreviate operation names. Figure 10 shows the resulting collaboration diagram.

Assembling the sequence diagram: If space permits, it may be helpful to combine the collaboration diagrams for a single use case into one sequence diagram (Figure 11). Since the same information is shown in both

a collaboration diagram and a sequence diagram, it is easy to convert from one to the other. It is recommended that the student do this manually in order to learn the correspondence even though tool sets will do the conversion automatically.

4.2.2 Create the Class Diagram from the Interaction Diagrams: This step is now mechanical – all the decisions have been made. That is why it is important to address the class diagram (Figure 7) after the interaction diagrams are developed. Otherwise students will become lost in designing the class diagram, as they have no idea where to place the operations in the class diagram and why.

5. SIGNIFICANT LITERATURE

The following references are useful sources for preparing to teach object-oriented systems analysis and design:

5.1 Classics of Structured Analysis and Design

Classic references for structured analysis are (McMenamin 1985) and (Yourdon 1989). The earliest presentation of event analysis (as "route mapping") is probably (Page-Jones 1980). Page-Jones (1988) presents structured design as well as a case study for both analysis and design. Teory (1986) summarizes the conventions and techniques for extended entity-relationship diagrams.

5.2 The Object Paradigm

A brief and simple introduction to object-oriented concepts is contained in (Taylor 1998).

5.3 UML

Fowler (2004) and Rumbaugh (2005) present an overview of the Unified Modeling Language.

5.4 Textbooks for Object-Oriented Systems Analysis and Design

Introductory texts include (Dennis 2005), (Larman 2005), (George 2004), and (Stumpf 2005). More advanced treatments are contained in (Page-Jones 2000), (Pooley 1999), and (Richter 1999).

6. CONCLUSIONS

Preparing to teach object-oriented systems analysis and design and the UML is perhaps not as difficult as some IS faculty fear. IS educators can take advantage of the similarities between structured and object-oriented approaches, especially during analysis. Experience in data modeling carries over directly to domain models. In the authors' experience, event analysis also remains valuable for high-level system decomposition.

Students should learn to work with one use case at a time when building the UML models. Contracts for system operations expressed in terms of a model of the application domain form the basis for design by contract, linking the UML analysis models to the design process.

Developing design models requires a basic understanding of the structure of object-oriented software. Nevertheless, three basic patterns are sufficient to help beginners construct acceptable initial interaction diagrams. The class diagrams should be derived from these interaction diagrams.

While industry has moved to object-oriented software development, information systems faculty have been slow to incorporate object-oriented analysis and design into the curriculum. Doing so would improve the currency of IS faculty and students and enhance the marketability of IS graduates.

7. REFERENCES

Dennis, Alan, Barbara Haley Wixom, and David Tegarden (2005) Systems Analysis and Design with UML 2.0: An Object-Oriented Approach. John Wiley & Sons, New York.

Fowler, Martin (2004) UML Distilled, Third Edition: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, Boston.

George, Joey F., Dinesh Batra, Joseph S. Valacich, and Jeffrey A. Hoffer, (2004) Object-Oriented System Analysis and Design. Prentice-Hall, Upper Saddle River, NJ.

Larman, Craig (2005) Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 3rd ed. Prentice-Hall, Upper Saddle River, NJ.

McMenamin, Stephen M, and John F. Palmer (1985) Essential Systems Analysis. Yourdon Press, New York.

Page-Jones, Meilir (1980) The Practical Guide to Structured Systems Design. Yourdon Press, Englewood Hills, NJ.

Page-Jones, Meilir (1988) The Practical Guide to Structured Systems Design, 2nd ed. Yourdon Press, Englewood Hills, NJ.

Page-Jones, Meilir (2000) Fundamentals of Object-Oriented Design in UML. Addison-Wesley, Boston.

Pooley, Rob and Perdita Stevens (1999) Using UML: Software Engineering with Objects and Components. Addison-Wesley, Boston.

Richter, Charles (1999) Designing Flexible Object-Oriented Systems with UML. Macmillan, New York.

Rumbaugh, James, Ivar Jacobson, and Grady Booch (2005) The Unified Modeling Language Reference Manual, 2nd ed. Addison-Wesley, Boston.

Stumpf, Robert V. and Lavette C. Teague (2005) Object-Oriented Systems Analysis and Design with UML. Prentice-Hall, Upper Saddle River, NJ.

Taylor, David A. (1998) Object Technology: A Manager's Guide, 2nd ed. Addison-Wesley, Reading, MA.

Teory, Toby J., Dongqing Yang and James P. Fry (1986) "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model." ACM Computing Surveys, v 18 n 2, September 1986. pp. 197-222. (See also the discussion and correction in v 19 n 2, June 1987. pp. 191-193.)

Yourdon, Edward (1989) Modern Structured Analysis. Yourdon Press, Englewood Hills, NJ.