Using UML Modeling to Facilitate Three-Tier Architecture Projects in Software Engineering Courses

SANDEEP MITRA, The College at Brockport

This article presents the use of a model-centric approach to facilitate software development projects conforming to the three-tier architecture in undergraduate software engineering courses. Many instructors intend that such projects create software applications for use by real-world customers. While it is important that the first version of these applications satisfy the customer by providing the functionality the customer expects and perform reliably and efficiently, it is equally important to be able to accommodate the customer's change requests over the period of the product's lifetime. The challenges in achieving these goals include the lack of real-world software development experience among the student developers and the fact that post-deployment change requests will almost certainly have to be handled by students who are not among the original developers. In this article, we describe how a model-centric approach using UML has been effective in enabling students to develop and maintain eight software applications for small businesses over a 9-year period. We discuss the characteristics of our modeling technique, which include the application of modeling patterns and quality check rules that enable students to create a model that can be clearly and consistently mapped to code. We also describe the nature of these mapping-to-code techniques, emphasizing how they reduce coupling among the implementation's classes. We then discuss our experiences in the classroom with these techniques, focusing on how we have improved our teaching over the years based on the analysis of student performance and feedback. Finally, we compare our approach to related work teaching modeling and the development and maintenance of code in software engineering courses with both extensive and minimal modeling.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms: Design, Documentation, Experimentation, Languages, Standardization

Additional Key Words and Phrases: Three-tier architecture, model-based development, Java, mapping techniques

ACM Reference Format:
Sandeep Mitra. 2014. Using UML modeling to facilitate three-tier architecture projects in software engineering courses. ACM Trans. Comput. Educ. 14, 3, Article 17 (October 2014), 31 pages.
DOI: http://dx.doi.org/10.1145/2635831

1. INTRODUCTION

Project work that expects students to create deliverables requiring a significant amount of work, executed over an extended period of time, and (usually) as part of a team, has long been considered an important part of the Computer Science curriculum. Fincher et al. [2001] present an exhaustive discussion of the numerous issues involved in setting up and managing projects, and in assessing the performance of the student teams

Author's addresses: S. Mitra, Department of Computer Science, The College at Brockport, State University of New York, Brockport, NY, 14420; email: smitra@brockport.edu.

DOI: http://dx.doi.org/10.1145/2635831

involved. In the context of undergraduate software engineering courses, such project work often involves developing a substantial software product with an external client. Various ways of managing the class' engagement with the external client have been discussed in the available literature. For example, Tenenberg [2010] presents an approach where the client allows a member of its team (an industry practitioner) to attend class sessions every week and provide critiques of the students' work from the professional's perspective. Tabrizi et al. [2009] discuss collaboration among geographically dispersed teams of students to develop software for a client who acts as the project manager of the student teams. Engagements such as these, in which the class and the client are fairly tightly coupled with each other, are possible in situations where the external client is a (possibly major) software development organization willing to invest its time and efforts into educating the next generation of its workers. In contrast, other authors [Gnatz et al. 2003; Khmelevsky 2009; Nurkkala and Brandle 2011] discuss projects where the client is essentially a customer for the product the students are developing. In these cases, the customer provides requirements, and gives feedback on the suitability of the product as it is developed. However, the customer does not manage the development or make any significant technical contributions. This is especially the case if the customer is a small (indeed, a micro) enterprise, with no significant in-house IT expertise. Such customers may be the only kind available to programs housed in small, liberal arts schools, which have not yet had the opportunity to develop a significant industrial partnership.

Software developed for regular use by a real-world customer is usually reasonably complex [Linos et al. 2003; Madey et al. 2005; Olsen 2008; Scorce 2010]. Therefore, many projects developing such software last beyond a single semester. Also, they require the programming knowledge and experience available in upper-division students. Such students, often on the verge of seeking employment in the field, find that a successfully completed real-world project strengthens their resumes. As a result, they are usually quite motivated to put in their best efforts into these projects. If the nature of the customer is such that the project fulfills societal goals/needs, student motivation is further enhanced [Hislop et al. 2009].

A real-world project is considered successful if the developed product largely meets the customer's initial requirements and performs reliably and efficiently. It is inevitable that perfective maintenance [Andrews and Lutfiyya 2000] requests (i.e., requests to handle new requirements) will follow. Small/micro-organizations tend to return to the original developers with such requests. This is a major challenge faced by small, undergraduate-only academic programs that lack access to personnel available over a longer period (e.g., graduate students or in-house programmers) who can obtain intimate knowledge of the code. In these settings, it falls on the instructors to work with a different group of students who have to learn the code and modify it appropriately, without compromising the reliability and efficiency of the original application.

Given this background, in this article, we discuss projects that involve developing software applications for small/micro organizations using upper-division undergraduate software engineering students in a small, liberal arts college. Appreciating the fact that students (not professionals) are developing the application, the customer is willing to wait several months (typically, up to the end of the academic year) for the first deployment. Similarly, customers are also willing to wait several months for their perfective maintenance requests to be handled. This article focuses on the role that software modeling has played in such an environment in enabling the success of projects created and maintained over a 9-year period.

1.1. The Problem-Domain: Three-Tier Architecture Systems

Applications that automate business processes usually conform to the three-tier architecture. In other words, they provide a Graphical User Interface (GUI) at the front end, have a middle tier containing the business logic, and have a database at the back end to permanently store the data needed for the effective operation of the business processes. Several instructors describe projects that conform to this architecture [Madey et al. 2005; Tan and Phillips 2005; Scorce 2010; Nurkkala and Brandle 2011; Fox and Patterson 2013]. Methodologies like the Rational Unified Process (RUP), including its educational version, the Unified Process for Education [Robillard et al. 2003], provide guidelines for the analysis and design of systems conforming to this architecture. Therefore, we conjecture that these systems are fairly widespread. The techniques and experiences that we report in this article apply in their fullest extent to these systems.

1.2. The Importance of Adopting a Model-Centric Software Development Methodology

Several authors [Dutson et al. 1997; Clear et al. 2001; Fincher et al. 2001; Dugan 2011] discuss a wide range of issues involved in the setup, management, evaluation, assessment, and overall impact of large projects. Dugan [2011] surveys a significant quantity of literature on the topic, reporting its results categorized by the principal issues the surveyed authors raise. These include models and goals for the project course (e.g., the time period over which the project should be completed and the skills the students expect to acquire from these courses); the sources from which appropriate projects may be obtained (e.g., conceived by instructor or obtained from industrial partners or local businesses), the setup, management, and evaluation of student teams (e.g., students self-selecting their team members vs. the instructor setting up teams; the instructor acting as a consultant to, or as the project manager of, the student teams; evaluating only the collective work of the team vs. also having a mechanism to evaluate individual efforts), and so on. Dugan [2011] also surveys various software process models used by instructors, which, in turn, impact the sequence of phases over which the project work is carried out. Instructors following a prescriptive, RUP-like process mention the use of UML modeling and the mapping of the model to code (e.g., Szmurlo and Smialek [2006]). However, we have found few articles whose primary purpose is to discuss the role played by software modeling in successfully completing projects. The importance of using a model-centric approach in industry is demonstrated by Forward and Lethbridge [2008], where the authors surveyed 113 software practitioners to uncover their attitudes and experiences regarding software modeling versus development approaches that avoid modeling. They found that their respondents believed that the following activities were easier to accomplish in code: fixing a bug, creating efficient software, creating a system as quickly as possible, and creating a prototype. On the other hand, activities like creating a usable system for end-users, creating a system that accurately meets requirements, comprehending a system's behavior, explaining a system to others, and modifying a system when requirements change were easier with models. As these are important goals in the environment in which we execute our projects, we view these results as supporting the emphasis on modeling in our approach.

Teaching modeling with UML at both the introductory and advanced levels has been discussed by a number of instructors [Burton and Bruhn 2003; Flint et al. 2004; Wei et al. 2005; Hai 2009]. The errors undergraduate students make in creating UML models are discussed in detail in Thomasson et al. [2006], Svetinovic et al. [2006], Bolloju and Leung [2006], Paterson et al. [2009], and Sien [2011]. In Section 5, we discuss details about the nature of these errors and the extent to which the techniques we present in this article reduce their occurrence. At this point, we will mention that we believe that a major reason for these errors is the fact that students do not appreciate the rationale for many modeling artifacts/features. For example, Boustedt [2012] indicates that students have different understandings of the purpose of class diagrams and the diamond symbols within them. Our approach emphasizes that the purpose of models is to provide a precise, high-level big picture that can be systematically mapped to code. We provide an overview of such a mapping technique in Section 2 of this article.

Table I. Roster of Projects

Project ID	Project Title/Description	Academic Year(s)	Number of Students (1st course)	Number of Students (2nd course)
P1	Video rental system for an ethnic mom-and-pop video store	2004–2006	19	6
P2	Patient encounter data maintenance system for a health care provider	2006–2007	27	3
P3	Inventory and transaction management system for a video game exchange store	2007–2008	13	8
P4	Inventory and rental management system for broadcast equipment for the Department of Communication on campus	2008–2009	11	3
P5	Graduating student exit interview data recording and management system for the Department of Computer Science on campus	2009–2010	14	2
P6	Inventory and rental management system for the bicycle borrowing program on campus	2010–2011	15	8
P7	Inventory and transaction management system for Christmas tree sales conducted by a Boy Scout troop	2011–2012	24	15
P8	Inventory management system for a small, independently owned restaurant	2012–2013	20	14

1.3. Where Do These Projects Fit into the Curriculum?

The author's affiliation is with an ABET-accredited Computer Science program in a small, liberal arts college. The curriculum has a two-course, upper division, software engineering–oriented sequence. The first course has a CS-2 pre-requisite, and covers traditional software engineering topics. It also covers UML extensively and requires students to create the model for the real-world project associated with this course. The second course is an elective and has the first course as a prerequisite. It is usually taken by the more motivated subset of students from the first course. The implementation work is done in this course. This article presents its results based on the author's experiences in the projects worked on in these courses, as shown in Table I.

87% of the students mentioned in Table I were traditional college age (18–22) students. Also, all of these students had either very little or no software modeling experience prior to taking the first course. The smaller number of students taking the second course in the years prior to 2010–2011 is due to the fact that this elective has only been offered in its current form since that year. Prior to that, students signed up for implementation work as an independent study/special topics course.

Section 2 of this article presents an overview of the modeling and mapping techniques used to complete the projects described in Table I. Section 3 presents pedagogical experiences with these techniques. In Section 4, we discuss issues that must be kept in mind if our approach is used in projects and also mention its limitations. Section 5 compares our approach to related work. We conclude with a reflection on our overall experience with these projects and possible directions for future work.

2. OVERVIEW OF THE MODELING AND MAPPING TECHNIQUE

Following a RUP-oriented development process, our first step involves writing use cases that describe the details of the business processes the system under development must automate. Our second step requires the creation of UML sequence diagrams from the use case workflow descriptions and the checking of these diagrams for quality. The next step consists of documenting the user interface to the system—we use a state diagram to depict the control flow among the user interface screens. Finally, we create CRC cards from the sequence diagrams. Starting with these cards and employing a technique that systematically maps model features to code, we implement the system.

2.1. Writing End-to-End Use Cases

Use case workflows are typically written as a sequence of request–response interactions between an external (human) user and an amorphous system entity (see Larman [2005]). Svetinovic et al. [2006] discuss how further modeling using such workflows can be problematic, as use cases in this form only capture some of the interchange data between system and actors. This data is hardly appropriate to capture the rest of the domain concepts associated with the system, which are needed to build a complete conceptual model. Influenced by similar experiences, our technique requires modeling use case workflows from end-to-end. In other words, our use cases not only indicate the requests received from, and responses sent to, the human user at the front end but also describe how the data at the back end is created, read, updated, and deleted between each request and its corresponding response. Also, in our technique, we have a key abstraction entitled the *main interface agent*. User requests are sent to this agent, who then interfaces with the backend, and generates the corresponding response. The main interface agent is thus a metaphor for the system under development and must be named appropriately. For example, in a hotel reservation system, it may be called a receptionist; in a bank ATM simulation system, it may be called a teller, and so on.

To write end-to-end use cases, it is vitally important that the modeler be aware of the data model associated with the system. We assume that the backend data is organized according to the relational model. To create a nontechnical mental picture of relational database tables, we advise considering them as rolodexes (with index cards in them) or as ledgers (with pages in them). Figure 1 shows a complete end-to-end use case describing the workflow of the Rent Videos service provided in Project P1 from Table I. The associated data model includes the following: A User Ledger containing information about registered customers, an Item Ledger containing information about videos in inventory, and an Invoice Ledger containing the user ID and the total cost of all videos rented by a customer in a single rental session. We also have a Rental Ledger that is linked to the Invoice Ledger in a one-many relationship and has information about the rental of each individual video.

Steps 3–6 of Figure 1 encapsulate a request–response pair sent to/from the clerk (the main interface agent). Within this pair, Step 4 describes how the clerk reads the relevant data from the backend ledgers. Similarly, within the request–response pair encapsulated in Steps 12–16, the clerk creates data in the ledgers. The order in which data is created is important—because the Invoice page has an auto-generated primary key, it must first be inserted. Then, the just-created Invoice's key is read and stored in the individual Rental pages, which are then inserted, as shown in Steps 14–15.

Correctly writing a workflow requires discovering the step-by-step process to provide a service and a thorough grasp of the data model. The steps and the data model are usually developed in tandem and over a number of iterations. Close consultation with the customer is also often required. The instructor's, customer's, and students' roles in developing use cases are discussed in Section 3.3.

2.2. Creating Sequence Diagrams

Figure 2 shows a sequence diagram for the Rent Items use case of Figure 1. In the first version of these diagrams, the only objects present are the main interface agent and objects created by reading from the ledger pages or created from user-supplied data for eventual insertion into the ledgers. The latter are called *persistable objects*.

WORKFLOW	
1.	The (human) user requests the clerk to rent items (videos).
2.	The clerk requests the user to provide his/her user ID.
3.	The user provides his/her user ID to the clerk.
4.	The clerk uses the user ID to retrieve the page in the User Ledger corresponding to this user ID (thus confirming that the user is already registered with the store).
5.	The clerk uses the information on the retrieved User page to verify that the user is allowed to borrow.
6.	The clerk requests the user to provide the ID of the item (video) he/she wishes to borrow.
7.	For each item ID that the user provides to the clerk: a. The clerk uses the item ID to retrieve the page in the Item Ledger corresponding to this item ID (thus confirming that the ID provided is a legitimate rentable item in its inventory). b. The clerk uses the information on the Item page, the User page, and the set of items rented so far in this session to verify that the user can borrow this item. c. The clerk creates a new Rental page and writes the user ID and item ID on it. d. The clerk computes rental data (which includes items like due date and time and cost of rental) using the information in the User Ledger and Item Ledger pages retrieved earlier, and writes all this data on the appropriate lines of the Rental page. e. The clerk adds the Rental page to the set of items rented so far in this session. f. The clerk requests the user to provide the ID of the next item he/she wishes to borrow.
8.	The user indicates to the clerk that he/she is done renting items.
9.	The clerk creates a new Invoice page and writes the user ID on it.
10.	The clerk computes the total rental cost of all items in the set of items rented so far in this session and writes this cost on the Invoice page.
11.	The clerk informs the user of the total rental cost and requests payment.
12.	The user provides the clerk with the payment amount.
13.	The clerk computes the amount of change to be returned to the user.
14.	The clerk inserts the Invoice page into the Invoice Ledger.
15.	For each Rental page in the set of items rented so far in this session: a. The clerk writes the key value of the Invoice page into the Rental page. b. The clerk inserts the Rental page into the Rental Ledger.
16.	The clerk informs the user of the change amount and of the fact that all the rentals have been successfully recorded.

Fig. 1. Workflow of the Rent Items use case.

Initially, the main interface agent executes all business logic. To effectively map these diagrams to code, the manner in which the main interface agent interacts with the backend ledgers must be clearly shown using modeling patterns. These patterns depict Create-Read-Update-Delete (CRUD) operations on the ledgers. They build on well-known concepts of object-relational mapping and use the "Active Record for Models" concept [Fox and Patterson 2013], which states that a persistable object can execute its own CRUD operations. We present two examples of such patterns in Figures 3 and 4. Pattern M1 (Figure 3) shows a read operation on a ledger, triggered by an "instantiate()" message to a persistable object. Pattern M2 (Figure 4) shows the creation of a collection of persistable objects into a ledger. Example applications of these patterns can be seen in Figure 2. For example, pattern M1 is used in the first instantiation of User object "u," and pattern M2 is apparent in the behavior associated with the manipulation of RentalCollection object "rentalColl." We have devised similar patterns for update and delete operations.

It is important that students form an appropriate mental image of how the workflow embodied in a sequence diagram is executed. Consequently, we emphasize that a sequence diagram is built as a sequence of request–response pairs. The main interface agent—in its human form—is considered to be idling before it receives a request. It is

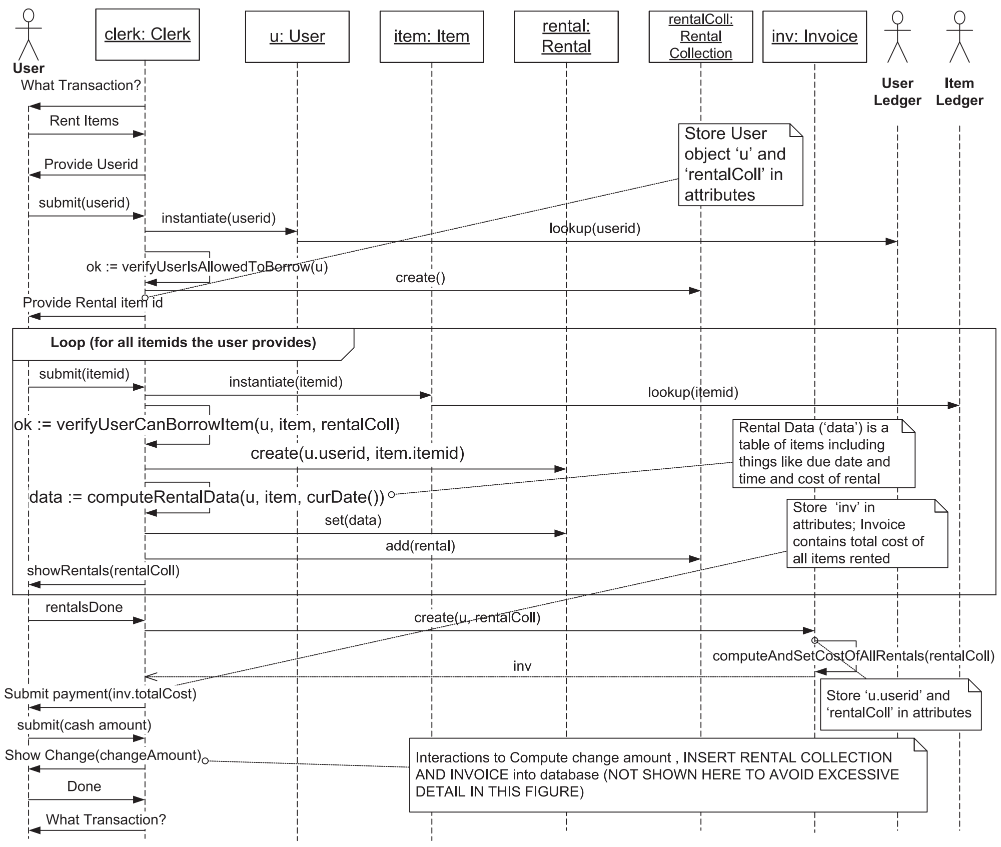
clerk: Clerk
u: User
item: Item
rental: Rental
rentalColl: Rental Collection
inv: Invoice
User
User Ledger
Item Ledger
What Transaction?
Rent Items
Provide Userid
submit(userid)
instantiate(userid)
lookup(userid)
Store User object 'u' and 'rentalColl' in attributes
ok := verifyUserIsAllowedToBorrow(u)
create()
Provide Rental item id
Loop (for all itemids the user provides)
submit(itemid)
instantiate(itemid)
lookup(itemid)
ok := verifyUserCanBorrowItem(u, item, rentalColl)
create(u.userid, item.itemid)
Rental Data ('data') is a table of items including things like due date and time and cost of rental
data := computeRentalData(u, item, curDate())
set(data)
Store 'inv' in attributes; Invoice contains total cost of all items rented
add(rental)
showRentals(rentalColl)
rentalsDone
create(u, rentalColl)
computeAndSetCostOfAllRentals(rentalColl)
inv
Submit payment(inv.totalCost)
Store 'u.userid' and 'rentalColl' in attributes
submit(cash amount)
Show Change(changeAmount)
Interactions to Compute change amount , INSERT RENTAL COLLECTION AND INVOICE into database (NOT SHOWN HERE TO AVOID EXCESSIVE DETAIL IN THIS FIGURE)
Done
What Transaction?

Fig. 2. Sequence diagram corresponding to the Rent Items use case.

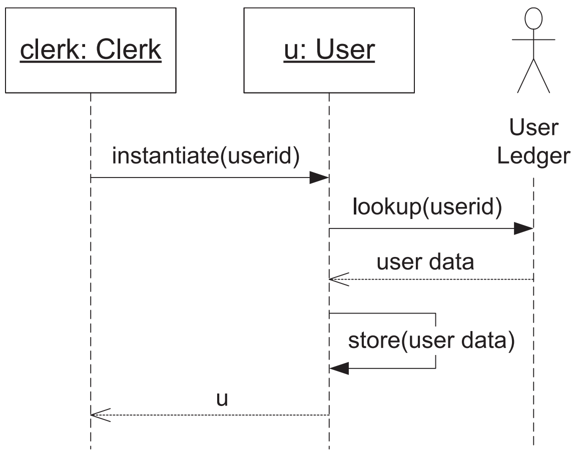

Fig. 3. Pattern M1: Instantiation of a single persistable object.

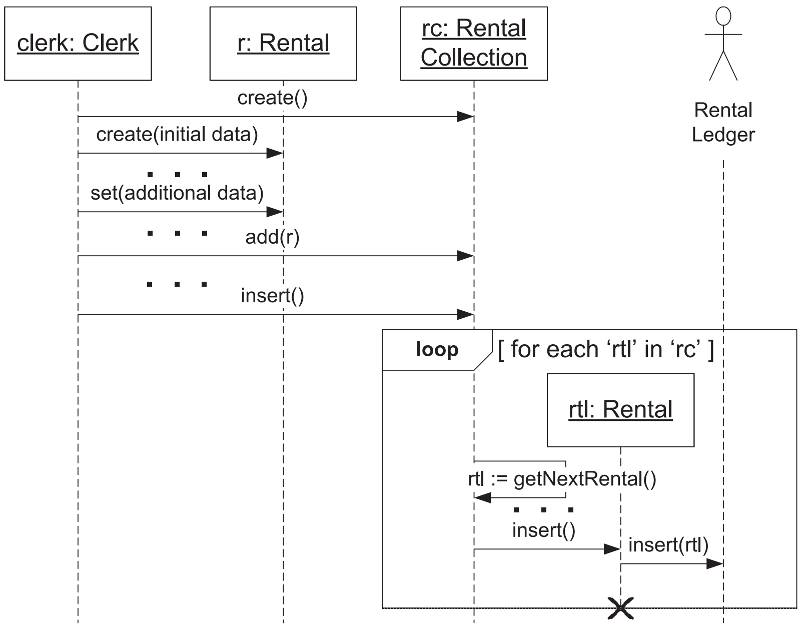

Fig. 4. Pattern M2: Creation of a collection of persistable objects into ledger.

perturbed by the request message, and it then has to become active. It processes the request by interfacing with the backend ledgers, and eventually calls back the user with a response message. Thus, for pedagogical purposes, we call a request–response pair a *perturbation*. As a result of a perturbation, the main interface agent's attributes may be altered—we discuss this in Section 2.2.1.

Each perturbation may include self-messages that depict algorithms whose details are usually not specified in the model. They serve to keep the model at a high-level and thus aid its understandability. Finally, our sequence diagrams are required to show how the associated use case is enabled. Note the "What Transaction?" message at the top of Figure 2, whose purpose is to enable the user to choose a use case. The diagram should end by enabling the user to choose the next use case they wish to execute; therefore, this same message appears at the bottom.

2.2.1. Analyzing the Quality of Sequence Diagrams. Effectively mapping sequence diagrams to code also requires these diagrams to be of high quality. This implies that they must be consistent in terms of data and control flows within them. Data flow consistency is based on the concept of "Designing for Visibility" [Larman 2005] and is illustrated by the following principle: Only talk to, and use, objects you know. An object "O1," in the scope of processing a message "M" it receives, can know another object "O2" not globally known if it either creates "O2," receives "O2" as a parameter of "M," or receives "O2" as a return value from another object "O3" it had talked to. If "O1" wishes to talk to, or use, "O2" after the scope in which it processes message "M" has terminated, it must store "O2" in its attributes. This last rule is used to discover attributes a class must have. We illustrate these discoveries in sequence diagrams using UML notes beginning with the label "Store..." (see Figure 2), and eventually in CRC cards. For example, in Figure 2, the topmost note states that object "clerk" needs to store object "u" in its attributes, because while "clerk" came to know "u" by instantiating it in the scope of processing

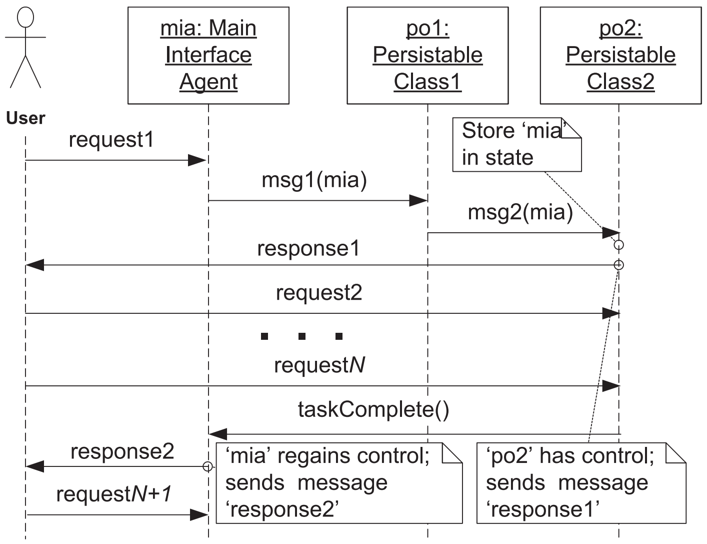

Fig. 5. Illustration of consistent control transfer.

request message "submit(userid)," it needs to use "u" in the scope of processing the later request message "submit(itemid)."

Control flow consistency is based on the principle that only the object currently in control may send the next message. The token embodying control is initially held by the external user. It is transferred to the main interface agent as a perturbation is initiated. Thereafter, it may be further transferred to another object, but it may only be returned to its sender. However, any object currently possessing the token may transfer it back to the external user, thus ending the perturbation. The next request will be sent to this object. These ideas are illustrated by Figure 5, and they form the basis of incorporating delegates into the model, as discussed in Section 2.4.

2.3. Creating State Diagrams

We use state diagrams to identify GUI screens and show how one screen transitions to another as a result of user actions. We build one state diagram from the full set of sequence diagrams created for a system. Each request message in a sequence diagram corresponds to a user action, whereas each response message corresponds to either the display of a new screen or an update of the contents of the currently visible screen. Two identically labeled responses always map to the same screen, even if they are present in two different sequence diagrams. Also, state diagrams contain at least one screen that serves as a pivot, from which it is possible to choose the next use case to execute. User actions on the pivot screen trigger a new use case, and a user action typically labeled "Done" ends the use case and redisplays the pivot.

Figure 6 shows a typical state diagram. Since multiple use cases may share the same GUI screen, it is necessary to envisage a history stack of user actions and visited screens that is maintained as a use case executes. This stack is used to disambiguate identically labeled user actions. For example, in Figure 6, the history stack is referenced in the two transitions labeled "Select User(user)" out of the "User Collection" screen. If the contents of the stack indicate that the Edit User use case is in progress, the "Edit User" screen is reached, whereas if they indicate that a Remove User use case in progress, the "Remove User" screen is reached.

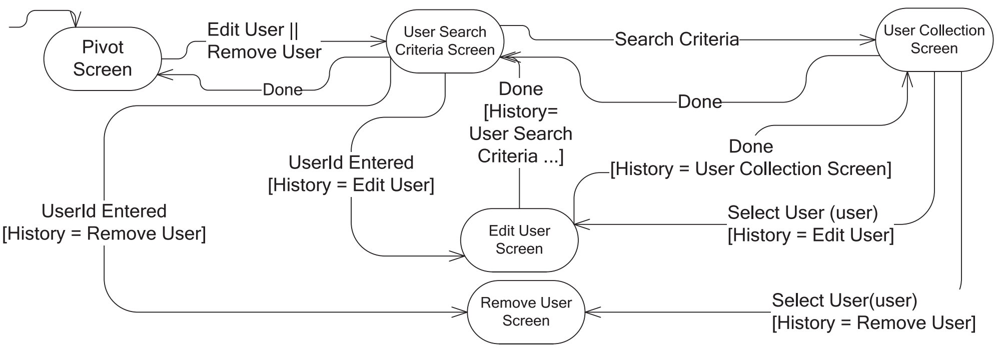
Pivot Screen
Edit User || Remove User
Done
User Search Criteria Screen
Search Criteria
User Collection Screen
Done
Done [History= User Search Criteria ...]
UserId Entered [History = Edit User]
Done [History = User Collection Screen]
UserId Entered [History = Remove User]
Edit User Screen
Select User (user) [History = Edit User]
Remove User Screen
Select User(user) [History = Remove User]

Fig. 6. An example of a state diagram.

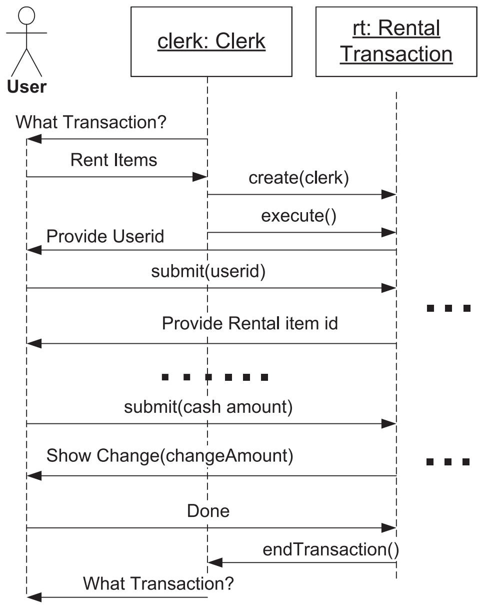

Fig. 7. Refactoring the sequence diagram in Figure 2 to include a delegate.

2.4. Delegating the Main Interface Agent's Responsibilities

The main interface agent can be made more cohesive by delegating some of its responsibilities. A good strategy is for the agent to transfer the responsibilities of executing the details of a use case to its own delegate. For example, following the "Rent Items" request message, object "clerk" in Figure 2 can create a delegate called "Rental Transaction" to handle all the details of the Rent Items use case. Figure 7 shows a refactoring of Figure 2 to use this delegate. It can be seen that the manner in which control is transferred to this delegate, and returned to "clerk" when the use case is complete, is consistent with the rules discussed in Section 2.2.1 and illustrated by Figure 5. Now, object "clerk" in Figure 7 is only responsible for creating the appropriate delegate for a certain use case and for allowing the user to choose the next use case. It should be noted that the history stack discussed in Section 2.3 can be realized by using delegates, using a strategy not discussed in this overview.

2.5. Mapping the Model to Code

Modeling is worthwhile only if the model provides a clear roadmap for writing the code. In this section, we describe the salient features of one possible mapping of a model created as described in Sections 2.1 through 2.4 to a desktop-based Java implementation environment. The Swing package is used to build the GUI and a possibly remote database is accessed using JDBC. Figure 8 shows a code skeleton generated using this mapping that seven of the eight projects in Table I utilized.

The modeling patterns of Section 2.2 are mapped to code using a locally developed framework for database access that uses JDBC and precludes the need to embed SQL code in Java when simple retrievals (i.e., those that use a condition that is only a conjunction of equality checks), inserts/updates of single rows into a table, and deletes of rows from a single table are needed. To use this framework, a class must extend the base class Persistable it provides (e.g., see class User in Figure 8). Class User's constructor uses the framework's features to access the appropriate row of the database table called User and populate its attributes with this row. On the other hand, class Rental's constructor simply initializes the class' attributes with its parameter values. The method “stateChangeRequest()” in class RentalTransaction uses this constructor to create a new Rental object as it processes an “itemId” sent by the user. This method will eventually invoke the Rental object's “save()” method, which will use the framework's features to insert its attributes into the database.

The mapping of the interactions with the front end to code is based on the Model-View-Controller (MVC) architecture. Classes extending Persistable map to MVC models. The main interface agent and the delegates it creates, as shown in Section 2.4, map to Controllers. Response messages map to either the display of a new View or an update of the current View (see Section 2.3). To reduce coupling, a Controller that needs to create and display a new View always uses a View Factory [Gamma et al. 1995], as shown in Figure 8. This implies that a View must implement a common interface called IView—in fact, it usually extends a framework-supplied class called View that implements this interface. Views are displayed in a global view container, made available to all Controllers using the Singleton pattern.

Response messages that map to an update of the currently displayed View are implemented as a callback to the View object currently showing in the container. Using the Observer pattern [Gamma et al. 1995], a View must subscribe itself to its associated Controller as an observer (e.g., see the “myCntrlr.subscribe(this)” statement in RentalsView's constructor in Figure 8). The IView interface mandates the “updateState(..)” method. At an appropriate time, the Controller invokes the framework-supplied “updateSubscribers()” method. This method, in turn, ensures that the “updateState()” method of each subscribed observer is called, with the attribute values of the Controller passed in via the Hashtable parameter of this method. Each View then updates the contents of their displays with these values. Thus, Controllers are not coupled to the implementation classes of the Views they create/update. Furthermore, in order to decouple the Views from the implementation class of their associated Controller, our framework requires a Controller to implement the interface called IModel (this is somewhat of a misnomer, but continues to be used for historical reasons). This interface mandates the “stateChangeRequest(String key, Object value)” method. All requests to a Controller are conveyed via this method. This can result in all business logic residing in a giant if-statement within this method, which is certainly undesirable. Appropriate use of the Command pattern can rectify this situation.

The overview provided in this section includes only the information about the modeling and mapping techniques needed to appreciate the pedagogical experiences described in Section 3. Full details are available to interested instructors on request from the author.

```
public class RentalsView extends View implements ActionListener {

    private JTextField itemIdField;

    public RentalsView(IModel modelOrController) {…
          myCntrlr.subscribe(this);
    }

    public void actionPerformed(ActionEvent evt) {
      // Line below shows the sending of entered itemId to controller
      myCntrlr.stateChangeRequest("ItemId", itemIdField.getText());
    }

    public void updateState(Hashtable data){
      // Use 'data' to update the values showing on your screen
    }

public class RentalTransaction extends EntityBase …{

    private User u;
    private RentalCollection rColl;

    public void stateChangeRequest(String key, Object value) {…
          if (key.equals("UserId")) {   // user submitted a 'user id'
             try {
                 u = new User((String)value);
                 // Call private method to see if user 'u' can borrow
                 rColl = new RentalCollection();
                 View rView = ViewFactory.create ("RentalsView",this);
                 // Show this view in the Singleton main display frame
             } (catch UserNotFoundException ex) {…}
          }
          else if (key.equals("ItemId")) {// user submitted 'item id'
           // Handle the sent itemId as shown in Figure 2

           Properties prop = new Properties();
           prop.setProperty("userId", u.getId());
           prop.setProperty("itemId", …);
           Rental r = new Rental(prop);
           // Compute rental data and set it into 'r'
           rColl.add(r);
           updateSubscribers(this);       // update 'Rentals View'
    }
}

public class User extends Persistable {

    public User(String userid) throws UserNotFoundException {
     // Use functionality provided by the Persistable class, which
     // works over JDBC to connect to database and retrieve data
    }
}

public class Rental extends Persistable {

    public Rental(Properties initialData) {
      // Set 'initialData' (key, value) pairs into local attributes
    }
    public void save(){//Insert/update local attributes into database}
}
```

Fig. 8. Code Skeleton mapped from Figures 2 and 7.

3. PEDAGOGICAL EXPERIENCES WITH THIS APPROACH

In this section, we first discuss the motivation for the modeling and mapping techniques described in Section 2. We then discuss how these are taught in the context of real-world projects over the two-course sequence discussed in Section 1.3. We present our evaluation scheme and discuss the manner in which we analyze the results of student evaluations, together with the feedback we get from students. We also discuss how we incorporate all these into improving our teaching.

3.1. Motivation for the Development of the Modeling and Mapping Techniques

The motivation for the modeling and mapping techniques described in Section 2 came from our early experiences with student projects. During these early years, we provided our students with the guidelines and checkpoints for RUP [Larman 2005; Robillard et al. 2003]. We found that our inexperienced students found these hard to implement in practice. Besides the problems encountered in creating the domain/data model, as we mentioned in Section 2.1, we noted that they particularly had a problem in deciding on the right set of business logic (aka Controller or RUP's Control) classes and distributing the required computations among them. Our students were not alone in being confused about the exact nature of these Control classes, as entries in this Wiki [Whatsa 2012] attest. In Project P1 of Table I, six students logged over 275 hours of teamwork, yet there was considerable code replication and tight coupling among the implemented classes. The instructor, with the help of a colleague from industry, had to refactor it considerably before deployment. It was during this process that we focused on developing a modeling technique that would, at the very least, better enable the identification of, and distribution of, business logic among Control objects. The technique should be teachable to undergraduate students. Our experiences over the years eventually led to the conception of ideas such as the "human" main interface agent, who first executes all the business logic and then delegates responsibilities systematically to other objects, as discussed in Section 2.4. The difficulties students encountered in ensuring that the implementation code is consistent with the model led at first to the concept of modeling patterns described in Section 2.2, then to the need to ensure data and control flow consistency in the model as described in Section 2.2.1, and finally to the mapping technique discussed in Section 2.5.

3.2. Course Content, Evaluation Scheme, and Obtaining Student Feedback

In the first semester course, students work on individual exercises requiring them to write use cases and create sequence diagrams and state diagrams. In addition, they work in teams that create these artifacts for the real-world term project. For each artifact, a student is evaluated according to the following scheme:

—Use cases:
  —The student writes end-to-end use cases from instructor-provided natural language problem descriptions individually in two in-class exams (one of which is the midterm exam) in the first half of the semester.
  —The student works with team members in writing up use cases for the term project.

—Sequence Diagrams:
  —The student creates sequence diagrams from instructor-provided end-to-end use cases individually in a take-home assignment and in two in-class exams (one of which is the final exam) in the second half of the semester.
  —The student works with team members in creating sequence diagrams for the term project.

—State Diagrams:
  —The student creates state diagrams from instructor-provided sequence diagrams individually in a take-home assignment, and in two in-class exams (one of which is the final exam) in the second half of the semester.
  —The student works with team members in creating state diagrams for the term project.

We take a case study–based approach [Ramnath and Dathan 2008] to teaching modeling. Therefore, we provide students with two complete models of systems created according to the techniques described in Section 2 and discuss them extensively in class. The first one is a fairly small model—of a bank ATM system [Mitra et al. 2005]. The second is a larger model and is actually one of the prior projects completed by an earlier group of students in Table I. Care is taken to ensure that the larger model provided is not from a project that is too close to the current project in terms of functional requirements. For example, Projects P1, P4, and P6 in Table I may all be categorized as applications requiring the development of a check-in/check-out system. Therefore, students working on any of these projects did not see the artifacts from any of the other projects in this group.

We also provide students with checklists, which by using they can gauge the quality of each artifact they create. We modify these checklists periodically, based on the frequently occurring modeling errors we note in each offering of the course and also using the qualitative feedback we receive from students. The rubrics we use to evaluate the artifacts are derived from these checklists. Since Fall 2007, we have used these rubrics to categorize our students into one of the following four categories: 1, Beginning; 2, Developing; 3, Competent; 4, Accomplished, with respect to their ability to create each artifact. In this article, we present quantitative data about these categorizations. We also present details of the frequently occurring errors we have identified over all the offerings mentioned in Table I. To obtain appropriate qualitative feedback, we require the students to write a "reflection document," an approach often used by instructors who teach software development methodologies and/or have their students participate in large projects [Hazzan and Dubinsky 2006; Rosmaita 2007; Szabo 2014]. While students reflect on many experiences concerning the project (e.g., teamwork), in this article we focus on their feedback about the issues they encounter in creating a model using the techniques described in Section 2. The "reflection document" is submitted anonymously, but it is mandatory, and simply submitting it enables the student to acquire 5% of the final exam grade.

In the second semester course, students are taught the details of the mapping technique outlined in Section 2.5. In this context, design patterns such as the Observer pattern are covered. The major evaluation exercises in this course are:

—Either individually or in small teams (of at most three members), complete a perfective maintenance exercise requested by a customer of a prior project. The students are provided with the original model and code. For maintenance requests that require new features to be added, using the techniques described in Sections 2.1 through 2.4, the instructor creates new artifacts and provides these to the students. For maintenance requests requiring the modification of current features, the instructor modifies existing artifacts and clearly identifies these modifications to the students. The students then apply the mapping techniques outlined in Section 2.5 to write new code and integrate it into the existing application and/or modify existing code.
—Work as part of a moderate-sized team (of at most five members) to complete the implementation of the project modeled during the previous semester. Using the set of student models from the previous semester, the instructor creates a model that

all teams work with as they begin to write the code. Again, students must use the mapping techniques of Section 2.5 to develop the code.

The primary evaluation technique in this class is code review by the instructor to ensure that the developed code is compliant with the mapping technique. This is conducted several times during the semester. Students get feedback, on the basis of which they revise their code. In this class, we also require anonymous qualitative feedback from students in a “reflection document,” where we mainly ask them to reflect on the mandate to use the model and map it to code.

3.3. Issues Involved in Incorporating the Term Project into our Course Sequence

How a suitable term project is found is beyond the scope of this article (see Clear et al. [2001] and Dugan [2011] for strategies). The chosen project must be one that can be implemented using the three-tier architecture. Prior to the start of the academic year in which the project work will be carried out, the instructor works closely with the customer to capture a basic set of functional requirements. This involves deciding on an initial set of use cases, and considering their workflows. An important goal of this exercise is to capture the data model suitable for the desired functionality. Obtaining the right data model can be challenging (see Section 2.1). As the major goals of our software engineering courses do not include data modeling, the instructor aims to present the students with a fairly complete data model, together with an outline of the initial set of use cases and their brief descriptions. Student teams begin modeling by writing end-to-end workflows of these use cases. This exercise may cause the initial data model to be modified. In our experience with the projects mentioned in Table I, these modifications have never been too extensive. Nevertheless, the instructor must be closely involved in these modifications, if any, and ensure that all teams use the same data model. Furthermore, it can prove beneficial to the customer if they are part of the discussion on the design of the data model, as they then get a better idea of how the data that is crucial to their business is, or should be, organized. We have found that the nontechnical mental picture of a relational database system that we mentioned in Section 2.1 helps our (usually) nontechnical customers to appreciate the data model better; in Projects P1, P6, P7 and P8, we found customers who were interested in the data model and largely understood it. Consequently, they were more engaged with the project.

The initial set of use case workflows is reviewed with the customer at least once. A typical customer is not interested in the details of how the data repository is accessed and modified—they are usually more concerned with how they would use the system. Therefore, the instructor manages this engagement with the customer, with the goal of enabling the customer to obtain a basic idea of the frontend interactions outlined in the workflows, and solicit their feedback. Similarly, the typical customer is usually not interested in, and is incapable of understanding, UML artifacts such as sequence/state/class diagrams and CRC cards. However, mockups of GUI screens corresponding to the states, and how they transition as a result of user actions, are appreciated by most customers. Therefore, the next customer interaction occurs after state diagrams and GUI mockups are created. Often, the customer requests additional/modified functionality after reviewing the GUIs and the instructor must be involved to carefully manage scope creep. During the second course, versions of the implementation as they are developed are demonstrated to the customer. The customer often requests several changes to the functionality, which again must be managed to prevent scope creep. Ultimately, at the end of the course, the customer chooses one of the implementations deemed successful by the instructor for deployment, mainly on the basis of the GUI. Further details about the term projects’ size and the workload these projects impose on the instructor and students are presented in Section 4.

3.4. Analysis of Student Performance and Feedback in Developing Model Artifacts

In this section, we present the most frequent errors made by students for each artifact (use cases, sequence diagrams, and state diagrams). We also present quantitative assessment data, quantitative data from student feedback and results of analyzing the qualitative feedback from the students' reflection documents.

3.4.1. Use Cases. The most frequent errors we have observed in writing end-to-end use cases are:

(1) Using vague terms to describe an action rather than describing it in terms of how the action is accomplished by conversing with the backend data repository. For example, a student writes: "The clerk verifies the user," rather than writing the following sequence of steps: "The clerk uses the user ID to retrieve the page corresponding to this ID from the User Ledger; The clerk verifies that the provided password matches the password on this page."
(2) Omitting the name of the backend ledger accessed during the workflow. For example, a student writes: "The clerk retrieves the page with the user ID" rather than writing: "The clerk retrieves the page from the User Ledger with the user ID."
(3) Omitting steps that describe, or incorrectly describing (e.g., describing in the wrong order), the filing of newly created pages into the backend ledger, or the refiling of retrieved pages whose contents were updated in prior steps of the workflow (e.g., Step 14 of Figure 1 is omitted).
(4) Erroneously referring to the data retrieved or created in an earlier step; this is especially seen in more complex use cases (e.g., Step 7(b) of Figure 1 refers to a collection of Rental pages using the phrase "set of items rented so far." This exact phrase should be, and is, used again in Steps 10 and 15 to refer to this collection).
(5) Failing to recognize the need for collections (i.e., not recognizing the fact that certain user-provided data results in the retrieval of a collection of pages rather than a single page).
(6) Omitting to describe how all new/changed pages of a collection are filed/refiled (relates to frequent error (3)) (e.g., Step 15 of Figure 1 is omitted).
(7) Omitting needed retrievals from the back end—the most common reason for this is that certain retrievals necessary to validate the user-supplied data (e.g., Step 7(a) of Figure 1) are omitted.

Based on these common errors, we have created a checklist for use cases. Among the guidelines it includes, the major ones are the following:

—Ensure that every step in the use case workflow describes either a request (including the associated data) from the external user to the main interface agent, a response (including the associated data) from the main interface agent to the external user, or an action the main interface agent executes to process the last request from the external user.
—For each action that the main interface agent executes, consider whether it needs to interact with the back end. For example, does "verifying a user" need the main interface agent to interact with the data repository? If yes, be sure to write these details down. If you are sure no interaction with the back end is needed, then this action must be an algorithm, to execute which the main interface agent already has the data it needs. Therefore, write the step in a way that describes the nature of this algorithm, the data it works with, and the expected result. (This guideline aims to reduce the occurrence of frequent errors (1) and (2).)
—Ensure that every new page you create is filed into the appropriate ledger at some step. If multiple new pages are to be filed, pay attention to the order in which they need to be filed (as required by the data model).

Table II. Assessment of Student Performance in Writing Use Cases

Semester	1 (Beginning)	2 (Developing)	3 (Competent)	4 (Accomplished)	Average
Fall 2007	2 (15%)	4 (31%)	1 (8%)	6 (46%)	2.85
Fall 2008	0 (0%)	4 (36%)	4 (36%)	3 (27%)	2.91
Fall 2009	1 (7%)	4 (29%)	5 (36%)	4 (29%)	2.86
Fall 2010	0 (0%)	0 (0%)	7 (47%)	8 (53%)	3.53
Fall 2011	0 (0%)	2 (8%)	11 (46%)	11 (46%)	3.46
Fall 2012	0 (0%)	2 (10%)	9 (45%)	9 (45%)	3.35

Table III. Student perception of easiness in writing use cases (0 = very easy, 4 = very hard)

Semester	Fall 2007	Fall 2008	Fall 2009	Fall 2010	Fall 2011	Fall 2012
Average	1.77	1.54	1.71	2.06	1.92	1.95
Std-Dev	1.42	1.37	1.16	1.23	0.86	0.80

—Think of retrieving a ledger page as making a copy of this page and bringing it forward to work on it. Therefore, if some step of the workflow has changed the content of this page, ensure that the workflow has a step in which this copy is refiled into the ledger, replacing the original. (These two guidelines aim to reduce the occurrence of frequent error (3).)
—Consider whether the data supplied by the external user guarantees the retrieval of a single ledger page or may result in the retrieval of multiple pages. In the latter case, describe the retrieval as the construction of a collection. (This guideline aims to reduce the occurrence of frequent error (5).)
—Ensure that all new/changed pages of collections get filed/refiled in some step (This guideline aims to reduce the occurrence of frequent error (6).).

The checklist as described above dates from Fall 2010, and has remained largely stable since that semester. In our last offering (Fall 2012), we noted that, among all observed errors in the last evaluation (midterm exam), the top two most frequently occurring errors were error (3) (28%) and error (2) (14%). The categorization of students since Fall 2007 based on overall performance in writing end-to-end use cases is shown in Table II. The column "Average" indicates the weighted average category of all students in that offering. Since Fall 2007, we have also asked students to report their perception of easiness in developing use cases, using a standard Likert scale (0 = very easy, 4 = very hard). Table III reports this data.

We ask students to reflect on their experience by answering the following questions:

(1) Outline, in detail, what you found most difficult about getting the use cases right as well as what you found particularly easy (if anything).
(2) Having created sequence and state diagrams from the use cases, what would you do differently (if anything) if asked to write use cases for another application?

A t-test comparison of performance between students from Fall 2009 and earlier (Group A) with those from Fall 2010 and later (Group B) is statistically significant (Group A vs. Group B, $p < 0.05$), whereas t-test comparisons of performance between any two subgroups within these two major groups are not statistically significant (e.g., Fall 2011 vs. Fall 2012, $p > 0.05$). This leads us to conclude that the use of the checklist, which is the principal difference in the manner in which these two major groups were taught, had a significant impact upon improving performance.

Considering the qualitative input from students, answers to Reflection Question 1 clearly indicate that students find achieving precision in the workflow descriptions challenging. They struggle to find the right words to unambiguously depict the interactions with the backend database, describe the interactions with the front end (external user), and specify the setup and processing of collections. In response to Reflection

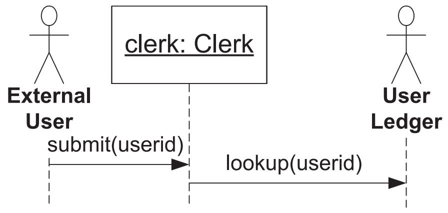

Fig. 9. Omitting the persistable object while accessing the back end.

Question 2, several students report that they would like to write the use cases at a level of precision that makes creating the sequence/state diagrams easier. In this context, a student in Fall 2011 wrote: “I should have written these requests and responses like this: “The user requests clerk to register a scout, waking him up from his restful slumber” and “The clerk tells the user to enter the barcode, and goes to sleep again.”” Through this statement, the student is indicating that (s)he would like to have the boundaries of perturbations in sequence diagrams more clearly identified in the use cases. Several students mention that after acquiring enough practice with sequence/state diagrams, they see how they could have written the use cases better.

Keeping these comments in mind, in the last two offerings, we provide detailed guidance and feedback on the language that should be used to describe steps of the workflow. For example, we require the use of one of the two words “file” or “insert” to indicate the addition of a new page to a ledger. Insisting on the use of standard words and phrases may be a reason for improved overall student performance since Fall 2010. It may also be the reason why, despite improved performance, the average of student perception of easiness has actually worsened, moving toward the 2.0 (neutral) value. Evidence for this is provided by the correlation between the average category of students in each offering shown in Table II and their respective perception of easiness shown in Table III (correlation coefficient: 0.87).

3.4.2. Sequence Diagrams. The most frequent errors we have observed in creating sequence diagrams are:

(1) Incorrectly accessing the backend ledgers when instantiating single persistable objects. Students, especially in the initial exercises, often erroneously show the main interface agent accessing the backend ledgers directly, instead of going through a persistable object (Figure 9). The frequency of this error led to the development of modeling patterns like M1 and M2 described in Section 2.2.
(2) Incorrectly labeling the message that brings a persistable object into existence. Persistable objects containing data already existing in the database are set up using an “instantiate()” message, whereas those containing externally (user-) supplied data are set up using a “create()” message. This is to distinguish the fact that an object containing external data must use an INSERT query to (eventually) save itself, whereas the other objects must use an UPDATE query to do the same. Perhaps because students do not experience this issue until the implementation phase, this error persists.
(3) Incorrectly modeling the retrieval, and saving, of collections. The frequent occurrence of this error led to the development of patterns such as M2.
(4) Not checking the diagram for control flow consistency, as described in Section 2.2.1. To rectify this, the following teamwork exercise is practiced: Assign students from a team to play the roles of the external user, each object, and ledger in the diagram. Embody the concept of the token representing the possession of control with a physical item (e.g., a little ball) that is initially present with the person playing the

Table IV. Assessment of Student Performance in Creating Sequence Diagrams

Semester	1 (Beginning)	2 (Developing)	3 (Competent)	4 (Accomplished)	Average
Fall 2007	2 (15%)	5 (38%)	2 (15%)	4 (31%)	2.62
Fall 2008	1 (9%)	5 (45%)	2 (18%)	3 (27%)	2.64
Fall 2009	3 (21%)	4 (29%)	3 (21%)	4 (29%)	2.57
Fall 2010	0 (0%)	4 (27%)	5 (33%)	6 (40%)	3.13
Fall 2011	2 (8%)	5 (21%)	7 (29%)	10 (42%)	3.04
Fall 2012	0 (0%)	2 (10%)	10 (50%)	8 (40%)	3.30

Table V. Student Perception of Easiness in Creating Sequence Diagrams
(0 = very easy, 4 = very hard)

Semester	Fall 2007	Fall 2008	Fall 2009	Fall 2010	Fall 2011	Fall 2012
Average	3.00	2.91	3.00	2.87	2.58	2.35
Std-Dev	0.96	1.31	1.13	0.96	0.95	0.85

external user. When the external user sends a request message, transfer the token to the person playing the object receiving the message. Only the person playing the role of the object that currently has the token can send the next message. Each message sender must remember to whom they sent the message. Sending a return implies that the "returner" asks for the identity of the sender of the corresponding message and returns the token to them. When a response message must go to the person playing the external user, the token is given to the latter, and everyone, other than the sender of the response message, forgets who they had sent the token to. The external user can then only send the token to the person who just sent it to them.

(5) Not checking the diagram for data flow consistency, as described in Section 2.2.1.

The checklist for this artifact really reflects much of what we described in Section 2 as techniques for creating sequence diagrams. It emphasizes the need to check whether the diagram is constructed as a sequence of perturbations, whether modeling patterns like M1 and M2 are properly used, and whether the consistency check rules of Section 2.2.1 have been applied. As with use cases, the current form of this checklist dates from Fall 2010. In the final exam of our last offering (Fall 2012), the most frequently occurring errors (excluding errors in UML syntax) were error (1) (50% of all observed errors), error (4) (16%), and errors (5) and (3) (8% each). The categorization of students since Fall 2007 based on their overall performance in creating sequence diagrams is shown in Table IV. Table V shows student perception of the easiness in drawing sequence diagrams.

A t-test comparison of performance between students from Fall 2009 and earlier (Group A), with those from Fall 2010 and later (Group B), is, again, statistically significant (Group A vs. Group B, $p < 0.05$), whereas t-test comparisons of performance between any two subgroups within these two major groups are not statistically significant (e.g., Fall 2011 vs. Fall 2012, $p > 0.05$). Therefore, we conclude that the use of the checklist is significant in enabling students to create better quality sequence diagrams.

Reflections on student experience are sought to two questions similar to those described for use cases in Section 3.4.1. Over the last three offerings, students clearly indicate that they found sequence diagrams difficult to create, with one student in Fall 2011 stating that creating them was "like coding, just with more structure, and a higher up (zoomed out) view." Especially in our earlier offerings, students found modeling the conversation with the backend data repository to be a challenge. They also indicate that "figuring out how loops are to be shown in communicating with the data" is difficult. These comments led us to the development of modeling patterns like M1 and M2. A significant number of students state that applying the data and control flow consistency rules requires effort. In response to the second reflection question, students state that after the experience of creating state diagrams they would pay greater attention to the labels on response messages, as these indicate the GUI screens that are present in state diagrams, which are important, as they need to be discussed with the customer.

Keeping student performance and comments in mind, we have sought to provide students more practice with sequence diagrams. We discuss a number of examples (at different levels of complexity) in class, and we have more individual assignments and test questions requiring creation of these diagrams. In our last two offerings, we spent 12–14 hours of class time discussing sequence diagrams. In Fall 2012, five (out of 20) students explicitly indicated that they found it easy to map use case workflow steps to sequence diagrams. One reason for this could be the emphasis we have laid on the use of standard words and phrases in use cases (see the last paragraph of Section 3.4.1). It is possible that the use of these words and phrases enables students to see the mapping from use case steps to modeling patterns more easily. Correlating the average category of students in each offering (Table IV) with their respective perception of easiness (Table V) yields a correlation coefficient of –0.84, thus indicating that as student performance has improved, they have also begun to perceive creating sequence diagrams as less difficult.

3.4.3. State Diagrams. The most frequent errors we have observed in creating state diagrams are:

(1) *Improperly identifying states*. Students do not observe the rule that two identically labeled response messages in two different sequence diagrams map to the same state. As a result, two different states that display the same content and enable the same user actions appear in the diagram.
(2) *Improperly identifying transitions out of a state*. The full set of possible user actions on a state is not considered. Students tend to forget to include the "Done" transition, which usually appears at the bottom of sequence diagrams and often simply takes the system back to a screen visited earlier (typically, the pivot screen allowing the next use case to be chosen).
(3) Not using the history stack to disambiguate user actions. Students often forget to disambiguate identically labeled transitions out of the same state with references to the history stack in the guards.

The checklist for this artifact, which also dates from Fall 2010, especially seeks to prevent error (1) by emphasizing that two identically labeled response messages must map to the same state. This emphasis was motivated by our observation in previous projects that ignoring this rule led to unnecessary code replication. Two identically labeled responses almost always present the same data to the user and allow similar user actions, so mapping them to different View classes replicates code among these classes. Because students do not actually experience this problem until the implementation phase, this error persists. We have continued to fine tune the checklist, and we provide examples that seek to prevent this error and also show how adhering to the rule that prevents this error leads to the use of the history stack. In the final exam of our last offering (Fall 2012), the most frequently occurring errors among all observed errors were error (3) (41%) and error (1) (26%). Categorizations of students' overall performance in creating state diagrams and student perceptions of easiness since Fall 2007 are shown in Tables VI and VII.

As with use cases and sequence diagrams, a t-test comparison of performances of the students in the two major groups (Fall 2009 and earlier (Group A) with Fall 2010 and

Table VI. Assessment of Student Performance in Creating State Diagrams

Semester	1 (Beginning)	2 (Developing)	3 (Competent)	4 (Accomplished)	Average
Fall 2007	3 (23%)	3 (23%)	3 (23%)	4 (31%)	2.62
Fall 2008	1 (9%)	2 (18%)	5 (45%)	3 (27%)	2.91
Fall 2009	6 (43%)	1 (7%)	4 (29%)	3 (21%)	2.29
Fall 2010	0 (0%)	1 (7%)	6 (40%)	8 (53%)	3.53
Fall 2011	2 (8%)	5 (21%)	6 (25%)	11 (46%)	3.08
Fall 2012	0 (0%)	1 (5%)	7 (35%)	12 (60%)	3.55

Table VII. Student Perception of Easiness in Creating State Diagrams
(0 = very easy, 4 = very hard)

Semester	Fall 2007	Fall 2008	Fall 2009	Fall 2010	Fall 2011	Fall 2012
Average	2.23	2.54	2.79	2.27	1.63	1.1
Std-Dev	1.37	0.99	1.21	1.18	0.86	0.70

later (Group B)) is statistically significant (Group A vs. Group B, $p < 0.05$), whereas t-test comparisons of performance between any two sub-groups within these two major groups are not statistically significant. Therefore, once again, we conclude that our checklist had an impact on improving our students' ability to create quality state diagrams.

In written feedback received since Fall 2010, students state that creating a quality state diagram requires starting with well-crafted sequence diagrams. They state that the modeler should devote some thought to the GUI screens the response messages in sequence diagrams map to. Students also state that they have to consider whether two distinct response messages map to the same or different states, and they have to keep the customer's desires in mind as they do so. Finally, they state that using the history stack in the guards correctly remains a challenge. However, in general, students comment that creating state diagrams is fairly mechanical and easy, provided the sequence diagrams they start with are well thought out. Correlating the average categorization of students in each offering (Table VI) with their respective perception of easiness (Table VII) yields a coefficient of –0.69.

The observations of the most frequent student errors that we have presented in this section have been in the context of our modeling technique described in Section 2. In Section 5, where we summarize observations of frequent student errors recorded by other instructors who (obviously) do not use our technique, we compare the differences between these two experiences.

3.5. Analysis of Student Feedback in the Implementation Phase

In the implementation phase (second course), code that conforms to the dictates of a technique that systematically maps model constructs to code constructs is written. It should be reiterated that there is not just one way of mapping models to high-quality code. In this context, we should mention that in Spring 2012, two A-level students from one team approached the instructor on their own initiative and criticized the implementation frameworks presented in Section 2.5. For example, they stated that while requiring every Controller to implement the same interface does allow any View to interface with any Controller, it makes tracing through the code harder, as it is not obvious which Controller a certain View is talking to. Consequently, they would rather have every Controller implement its own specific interface; for example, RentalTransaction in Figures 7 and 8 would implement an interface called IRental, which would contain methods to process submitted user ids, item IDs, cash amounts, and so on. Some Views would be coupled to this interface. This would make these

Views less flexible, but are there examples of Views that must be able to speak with absolutely any Controller? The instructor recognized the validity of their arguments, and gave them permission to use their own mapping technique, provided they could clearly document the mapping itself. This experience tells us that there could be several ways of systematically writing code from a model, with each technique having certain advantages and disadvantages over the others. The key is to choose a valid, well-documented technique and to strictly adhere to it. As outlined in Section 3.2, the evaluation process involves the instructor reviewing code written by each student team and insisting on revisions to ensure compliance with the chosen mapping technique. As a result of this approach, every student team working on each of the Projects P2–P8 of Table I has been able to develop a product to the customer's satisfaction. The relatively smaller number of student teams in this course (being an elective, this is usually taken by the more motivated students) has made this approach feasible.

Given the nature of our approach as described earlier, the means of obtaining appropriate quantitative measures of student performance in following a mapping technique was not evident. The existing literature contains few examples of assessing how students effectively create code from models. Therefore, we decided to gauge the effectiveness of our approach by identifying our students' perceptions of the appropriateness and effectiveness of our insistence in following a mapping technique. We undertook a process similar to the phenomenographic approach ideas presented by Kinnunen and Simon [2012] and also to that followed in more recent class projects involving large software development and maintenance [Szabo 2014]. In the years in which we conducted Projects P2–P5 of Table I, the author (who was also the instructor) and two other colleagues (who have an interest and some experience in students developing significant software products) independently reviewed the free-format reflection documents submitted by students on the use of the mapping technique. The team sought to identify the main themes that emerged from this feedback. Through this exercise, the three principal questions mentioned in Section 3.5.1 were identified. These questions were presented to students working on Projects P6–P8. Their (anonymous) reflections were independently analyzed by the author/instructor and one of the other two colleagues mentioned above. The responses were categorized to indicate whether the student was "positive," "neutral," or "negative" about the issues raised in the question, and the associated text was highlighted. There was more than 90% agreement between the two evaluators; in the few disputed cases, the third colleague was consulted for resolution.

3.5.1. Feedback from Students. The three principal questions used to solicit student feedback are:

(1) Do you think the UML artifacts were useful in helping you understand the functionality described by the project's use cases and implement them? Please comment in detail on which artifacts you found useful (or not), and why.
(2) The instructor insisted that you follow a mapping technique and keep the UML model and code consistent (i.e., the code is traceable from the model, and vice versa—implying that if you change one, you have to change the other). Did you perceive these as beneficial or problematic to the project? Please elaborate.
(3) The aim of the maintenance exercise was to provide you with experience in reading code written by other people, understanding it, and modifying it with your own code. Do you think this is a useful task? How difficult was it, and did the UML model and mapping techniques help you (e.g., were they better than just having inline comments)? Please respond in detail.

Question 1. In the context of Q.1 above, over our last three offerings (Projects P6–P8 of Table I), we categorized 24/37 (65%) of the students as being "positive," 12/37

(32%) as being "neutral," and 1/37 (3%) as being "negative" about the usefulness of UML artifacts. Students overwhelmingly identified the sequence diagrams and GUI mockups as the artifacts most useful in enabling them to write code. Students in the "positive" category stated that that the model was important in understanding the desired functionality to be implemented and completing the project on time. To quote two of these students:

—Although the time to create these models must be weighed against their utility, the project would have been much more difficult without the models and I would have felt less guided; eventually it would have taken me more time to complete the project.
—Sometimes I would have to take a break from coding for day or two, but when I came back to it, I could pick up from right where I left off, thanks to a quick review of the UML documents (mainly, sequence diagrams).

Among the students in the "neutral" category, four students were identified as stating that the models were useful in enabling them to appreciate how the workflow of the business process, outlined in the sequence diagram, maps to the "way the code flows among the implementation classes" (quote from one student). In other words, the model (and mapping) helped them identify a "coding pattern," To quote two other students from this set:

—Once these coding patterns were known, there was no need to keep referring back to the diagrams—you have the requirements in your head and can implement them.
—Use case workflows, if written correctly, are the most useful. There is no need for the sequence diagrams and other things. . .as a team, we should just use these workflows and talk about how the program should function.

Four other students in the "neutral" category were identified as stating that the model was helpful in "understanding the system's behavior, but not necessarily the code" (quote from one student). To quote another of these student: "In the code, the mapping technique says call the method "stateChangeRequest()" to convey data from view to controller, but the corresponding request message in the model has some fictitious name (fictitious because it never appears in code) .. this makes the model less helpful. The names used in the code should be the ones used in the model."

The one student in the "negative" category stated: "For a system at this level of complexity, UML diagrams are not needed. (I) can see the benefit in a "larger" system though. Main drawback: Hinders coding, especially changing the code to meet customer's changed requirements."

Question 2. Considering the responses to Question 2 over the last three offerings, we identified 23/37 (62%) of the responses as being "positive," 9/37 (24%) of the responses as "neutral," and 5/37 (14%) of the responses as "negative" about the instructor's emphasis on reviewing code and supervising the traceability between model and code. To quote from students in the "positive" and "neutral" categories:

—(Approach is) restricting but helps meet the challenge of getting the project done correctly and on time.
—Keeping the model and code in sync is problematic, mainly in terms of time, but the fact that the previous class had kept their UML model up to date with the code made working on their code (for the maintenance exercise) a lot easier. I see now why it has to be done this way.

Of the 32 students in the "positive" and "neutral" categories, 11 were identified as stressing the point in the last quote about the time it takes to keep the model and code in sync. This leads to the issue about the workload imposed by this course. In our

Table VIII. Information About the Size of Projects

	P1	P2	P3	P4	P5	P6	P7	P8
Use cases	17	10	5	32	15	22	24	20
LOC	17,260	17,864	6,754	33,004	2,311 (PHP)	16,587	18,540	17,389
Hours	275	129	88	255	56	116	182	197
Team size	6	3	4	3	2	4	5	5

latest offering (Project P8, Spring 2013), 5 out of 14 students mention the workload as a significant issue. To quote one of them: "This is only a three-credit course, and we still have to rapidly respond to at least some of the customer's new requirements." Workload was also mentioned by one student in the "negative" category, who stated that this approach hinders code refactoring and agility.

Also in the context of Question 2, in earlier years, students mentioned that a significant challenge in getting the mapping right was learning how to use the underlying frameworks correctly. Consequently, we introduced individual exercises intended to teach students the correct way to use framework APIs. In our last two offerings, when asked whether these exercises were useful, 15/29 (52%) students commented that they found these exercises useful. The remaining 14 students felt that these exercises, while useful, were unnecessary, and they could learn these by working on the maintenance and term projects. Their main complaint was again about available time (i.e., these exercises impact the time they can spend on the real-world projects).

Question 3. Evaluating the responses to Question 3, 29/37 (78%) responses were identified as feeling "positive" about the usefulness of the maintenance exercise, whereas the rest of the students, except one, were identified as having a "neutral" stance. Twenty-one of the students who felt positive about its usefulness also stated that they felt so because this exercise really gave them a flavor of what programming would be like in the real world, especially in their first jobs, and so this exercise helped them build a marketable skill. All but 2 of the 37 students stated that they are apprehensive about reading code written by others, so having the UML model and knowing how it is mapped to the code was beneficial. To quote one of these students: "The maintenance exercise should be harder, as then people will see the benefit of following the model closely in the term project." Four students addressed the issue about inline comments raised in the question, and stated that the model enabled them to see the "big picture" of the system in a way that inline comments could not possibly do. Two of the students in the "neutral" category stated that they disliked the existing code, and they would have personally written it very differently. But, they realize that modifying code they do not like has to be done in the real world, and they appreciated an exercise that required and also enabled them to read and understand "bad" code. The one student in the "negative" category did not really discuss the usefulness of the maintenance exercise. Rather, they stated that because it appeared early in the semester, they did not have the necessary practice with the mapping techniques and the use of the underlying frameworks that would have enabled them to tackle it effectively within the time available.

4. DISCUSSION ON LIMITATIONS OF OUR APPROACH

One characteristic of a real-world project is the need to manage the expectations of and overall relationship with the customer [Brooks 2008]. A major challenge for the instructor is to scope out a project that will satisfy the customer and still be feasible for the students to complete in the available time frame (two semesters).We try to let history be our guide. In Table VIII, we present information about the size of the projects in Table I, including the number of use cases, the LOC in the implementation chosen for deployment, and the time (hours) spent by the student team in developing this implementation (note: the time spent by the other teams has always been within about 20% of the hours reported for the teams in Table VIII). We note that we do not observe a direct relationship between the number of use cases and the eventual size (LOC) of the product, which is perhaps not surprising, as the size is impacted by the complexity of the use cases (a characteristic harder to measure) rather than their number. We have noted that most use cases describe CRUD operations on the database (e.g., Register a user, Add Item to Inventory, etc.), whose complexity is approximately the same (P4 is a notable exception). The complexity of a few key use cases describing the major business processes specific to the customer (e.g., the nature of a rental process) is what the instructor must gauge to judge the suitability of assigning the project. A similar approach is needed in assigning the maintenance exercise.

Authors such as Clear et al. [2001] and Dugan [2011] have discussed the increased workload on both instructors and students in project experiences. These include issues such as proper recognition of, and rewards provided for, project supervision (for instructors) and dealing with nonperforming team members (for students). We acknowledge the validity of all these issues. In addition, as is evident from the discussion in Section 3.5.1, the insistence on maintaining the traceability between model and code via the designated mapping technique adds to the students' workload. While a number of students appreciate that the instructor's "micromanagement" of their code is to facilitate the maintenance work their juniors will carry out, the mention of a higher workload, especially when compared to that in other three-credit courses offered at the same institution, does not fail to appear. Clearly, learning a technique that maps model constructs to constructs of an implementation framework (an exercise that requires spending time to learn the framework APIs as well), and adhering to this mapping strictly as you code is time-consuming. We keep emphasizing that spending the time to maintain consistency between model and code is an investment in the future. Some anecdotal evidence from this course's alumni provides us with a justification for our insistence on this approach. These alumni have, in informal conversations, informed us that they wished they had a set of documents like we created to tackle the maintenance tasks (including bug fixes) that their employers threw at them. We reported these stories to students in our latest offering. Yet, it must be stated that the workload issue and the scope of the project as it will impact the workload, must be kept in mind while adopting the approach outlined in this article.

Our approach also impacts the instructor's workload. In a small college setting, a class size of approximately 15 students in the implementation phase, which results in about three teams, has proved manageable for a single instructor. Larger class sizes would require additional instructional resources. In a large university setting, a cohort of trained graduate students may be available to conduct code review sessions with small groups, but in resource-limited small colleges, class sizes may have to be limited. Thus, some students may be deprived of the opportunity to participate in a real-world project experience. Also, since our approach stresses the maintenance experience, we receive the maintenance requests from our customers. As the number of deployed projects grows, we have more maintenance requests than we can handle. The inability to satisfy all these requests in a reasonably short period of time is a concern for a small department. To tackle this issue, approaches such as the Agile Software Factory [Chao and Randles 2009] are clearly the way to go, although these require a substantial commitment of certain resources as well.

Our insistence on maintaining a mapping from model to code also contributes to another drawback of our approach, which is that we can only tackle projects that clearly fit the three-tier architecture and where the customer is comfortable with the implementation technologies to which (at this point) we can map our model (i.e., Java for the desktop, and PHP for the web). Projects that require concurrency (e.g., the user interface for the project in Linos et al. [2003]), and those from other domains such as game development and embedded systems would require a different modeling technique, even when using UML. The mapping technique to another implementation environment (e.g., C# over .NET, Java-based web frameworks like GWT) would differ in its details from our current ones. We hope to tackle these in our future work.

5. RELATED WORK

Many authors describe common errors students make in creating UML artifacts. From Sien [2011], Bolloju and Leung [2006], and Thomasson et al. [2006], we obtain the following list of frequent errors for sequence diagrams: Missing initial trigger, missing a controller class, mismatch between class and sequence diagrams (one type of diagram refers to classes missing in the other type), inappropriate modeling of multiobjects (collections), errors with message parameters (parameters either missing, or used before they have values available), responsibilities delegated to the wrong object, and return of control to an object different from the caller. For class diagrams, errors include references to nonexistent classes, the presence of nonreferenced classes, and inappropriate relationships between classes. Using our approach, which starts with writing end-to-end use cases where the main interface agent is introduced as the principal "doer,", we have never encountered the problem of missing controller classes. The main interface agents and their delegates play the role of controllers. By emphasizing that a sequence diagram must start by showing how a use case is enabled and end by showing how the user can choose the next use case (see Section 2.2), we have rarely encountered the problem of missing initial triggers. Eliminating errors in the modeling of collections is a challenge—with the use of our modeling patterns and increased class time, we noted in Section 3.4.2 that these errors are still 8% of all errors observed in creating sequence diagrams. The same applies to errors with data and control flows (8% and 16% of all observed errors, respectively). But we should also state that by emphasizing the data and control flow consistency rules in Section 2.2.1, we have noted a declining trend in the absolute number of these errors over the years. Emphasizing the use of Larman's [2005] Information Expert principle to transfer responsibilities from the main interface agent to other objects has reduced the errors where responsibilities are assigned to the wrong object to negligible levels. Since we emphasize behavior modeling (sequence diagrams) and derive the attributes we put into the structural artifacts (CRC cards) by applying our data consistency check rules to the sequence diagrams, the occurrence of nonexistent and nonreferenced classes and methods in the structural artifacts is also negligible.

Few papers discussing real-world projects focus on the role modeling and mapping played (if any) in the project's implementation. Hai [2009] discusses the role of collaboration diagrams in analyzing use cases and identifying the initial set of RUP's Boundary, Control and Entity classes. Szmurlo and Smialek [2006] present a small example of mapping sequence diagrams to code. The basis of this mapping is the same as those discussed by several other researchers (e.g., Thongmak and Muenchaisri [2002]), where a message in the sequence diagram maps to a method call with the same name. The mapping we discuss in Section 2.5 is not quite as simple. For example, all user actions on views, shown as request messages (with different names) in sequence diagrams, map, in the interests of reducing coupling, to invocations of the same Java interface-mandated method called "stateChange Request()" on the view's controller. As we discussed in Section 3.5, some of our students disapproved of this and came up with an alternative mapping. We state that the mapping chosen must be well documented and must demonstrate how it facilitates the writing of extensible (and thus, maintainable) code.

Dugan [2011] presents a survey of how instructors teach maintenance and approach the maintenance tasks associated with real-world projects. We note that many authors discuss the process using which they have students execute maintenance tasks. For example, Beasley [2003] discusses a requirement that students work on maintenance tasks in a prior project before signing up for the project course, whereas Burge [2007] has two teams work on the same project, and exchange code bases for the maintenance release. Szabo [2014] requires the class to work on adding features to a student project from the previous year. The starting point for maintenance exercises appears to be an existing code base, plus inline documentation [Pierce 1997; Slimick 1997; Andrews and Lutfiyya 2000; Szabo 2014]. In more recent efforts, some external documentation in the form of diagrams and screen shots is available [Brazier et al. 2007]. In our approach, the instructor modifies the existing UML model to indicate the new or modified features desired by the customer. We emphasize the need for students to understand this changed high-level model, and then apply the mapping technique to either create new code or edit existing code appropriately. The new code is then integrated with the existing code base, and the system is tested again. In this sense, we view the maintenance tasks as practice exercises for students to learn the mapping technique well enough to apply it in their term project. With this approach, we have been able to accommodate a few maintenance tasks in the same semester as the main term project. These tasks were relatively small-scale ones—they have required modification or addition of at most three use cases, not of significant complexity (although still useful to the customer). Our students complete the maintenance exercise in 3 weeks, on code bases that range from approximately 6,000 to 18,000 Java LOC (Table VIII). Similarly, in the class mentioned by Szabo [2014], students are given 4 weeks to add one feature to a code base of 11,000 Java LOC and fix some bugs remaining from the earlier version. Larger maintenance exercises—say, those requiring the addition of 10 or more new features—are feasible with our approach, but would require a dedicated course on software maintenance. Alternatively, these features could be introduced over multiple offerings of our current course, although that could run into the customer satisfaction problem mentioned in Section 4.

6. CONCLUSIONS AND REFLECTIONS FOR THE FUTURE

In this article, we discussed how a model-centric approach has facilitated the successful completion of eight projects requiring the development of software applications conforming to the three-tier architecture. The key idea is to enhance an *essentially* RUP-based approach to modeling with certain techniques that permit the development of mapping strategies to code. The basic intention of the UML artifacts we create is to provide a high-level picture of the behavior of the application. This picture omits certain details; for example, algorithms shown as self-messages in our sequence diagrams are only elaborated in the code. Despite this fact, the artifacts must still attain a high level of precision; for example, our sequence diagrams must show objects interfacing with the front and back ends in a well-defined manner and should be internally consistent in terms of data and control flow. Only such precisely developed artifacts can be systematically mapped to code, thus creating an application that is understandable, and, therefore, maintainable by a later generation of students. It is in order to enhance the level of precision in students' models that we have begun to emphasize rigor in the modeling process from the earliest stage; for example, in the discussion on writing use case workflows (Section 3.4.1), we mentioned the emphasis on using standard words/phrases in the natural language descriptions of the main interface agent's interactions with the front and back ends. Thereafter, we noted the advantages of doing so when moving on to the creation of sequence diagrams (Section 3.4.2).

Taking such a rigorous approach to modeling and ensuring that consistency between the model and code is always maintained requires a significant time commitment on the part of both the instructor and students. Approaches categorized as Agile (such as in Hanks [2007] and Fox and Patterson [2013]) either ignore modeling or, echoing ideas put forth by Ambler [2001], suggest that only a minimal amount of modeling be done. These approaches advocate that the code and precisely written tests be the primary medium used to communicate the system's features and design among current and future team members. For such an approach to be successful, a typical student team should be able create reasonably high-quality code right from the start of the project, through the effective application of OOD principles such as Single Responsibility and Dependency Injection, and continually improve the code via refactoring. Moreover, they should be able to accomplish this within the time frame available for project completion, and within the constraints imposed by a typical college curriculum (e.g., several other courses to handle in the semester). We began to consider: What kind of training do we need to provide our students with so that they can be as successful with this approach as they have been with the use of our techniques? We assume that the expertise they have at the end of a typical CS-2 class is not enough.

We obtained some insight into this question after studying the concept of "notional machines" reviewed by Sorva [2013]. Broadly speaking, a notional machine is an abstract computer responsible for executing programs of a particular kind. Sorva [2013] discusses the need for students to build an appropriate mental model of a notional machine so that they correctly understand how a program in a particular language, or a set of related languages, is executed. Considering the experiences reported in this article along similar lines, we posit the idea of a notional machine suitable for a system that conforms to the three-tier architecture and automates business processes. Such a machine must include the following features: It must allow for the conception of functionality as a sequence of request–response interactions between an external user and entity serving as a façade to the system (namely, the main interface agent); this entity and its delegates are responsible for processing the data from the user request and generating the corresponding response after accessing and modifying a backend data repository using well-defined rules; and this entity should, if necessary, maintain state (i.e., remember and forget data) as it goes through a sequence of request–response pairs.

With this mindset, we could say that an important reason for teaching our modeling technique—a technique that requires the creation of a series of UML artifacts—is to enable students to gradually build the right mental model of how a three-tier architecture system operates. If students do have the right mental model, they can quickly conceive the correct components that constitute such a system, and map them to the desired implementation environment. For example, in a web-based system, they would correctly map the main interface agent to the web server, and to the server's delegates mandated by the web development framework being used. Similarly, they would map the persistable objects they have in mind to model classes that access a database. If such is the case, perhaps the creation of many of the artifacts outlined in Section 2 becomes unnecessary. Therefore, if the curriculum provides students with the practice necessary to build "robust mental models" [Sorva 2013] of three-tier architecture systems, then students need not create the artifacts discussed in this article when working on, say, a later capstone project that requires building such a system.

Let us re-examine some of the student responses to Question 1 (of the questions outlined in Section 3.5.1) that we categorized as "neutral." These students stated that they found many of the UML artifacts unnecessary because once they had identified how the "code flows among the implementation classes" (i.e., once they had a coding pattern in mind), then if they "had the requirements in their head," they saw "no need to keep referring back to the diagrams." We now think that these students responded this way because they had, without being conscious of it, been able to develop an appropriate mental model of a notional machine for three-tier architecture systems. However, these students are still in a minority. Therefore, we believe it is necessary to continue to teach the modeling technique outlined in this article, but possibly do so with the aim that, eventually, the technique will appear so natural to students that they will not need to build and update these artifacts for later projects. Consequently, we hope that the amount of time they spend on the implementation phase of the project will be reduced from the current levels to the extent possible.

We would like to direct our future efforts toward the possible achievement of this goal. We believe that we will need to provide students with more practice exercises in modeling and mapping-to-code, to which we provide detailed feedback. Challenges in incorporating such ideas are significant, especially given the numerous constraints an undergraduate setting in a small, liberal arts college places, such as limits on the number of credits that may be required by the curriculum. Due to such limits, adding a supervised lab component to our two courses, for example, could prove difficult. Nevertheless, based on the experiences reported in this paper, we suggest that a greater emphasis on modeling and mapping-to-code techniques in the Software Engineering components of the traditional computer science undergraduate curriculum may be in order. If these are taught in depth as early as possible, perhaps students will not need to explicitly create and document detailed models—at least for the kind of systems they have already developed—in later projects?

ACKNOWLEDGMENTS

The author would like to thank the numerous students of The College at Brockport who participated in the projects discussed in this article. Without their dedication and hard work, this effort would never have materialized. The author would also like to thank the anonymous referees whose feedback on earlier versions helped improve this article.

REFERENCES

Scott Ambler. 2001. Agile Modeling (AM) Home Page. Effective Practices for Modeling and Documentation. (2001). Retrieved April 6, 2014 from http://www.agilemodeling.com.

James H. Andrews and Hanan L. Lutfiyya. 2000. Experience report: a software maintenance project course. In *Proceedings of the 13th Conference on Software Engineering Education and Training (CSEET'00)*. IEEE Computer Society, Washington, DC, 132–139. DOI:http://dx.doi.org/10.1109/CSEE.2000.827031

Robert E. Beasley. 2003. Conducting a successful senior capstone course in computing. *Journal of Computing Sciences in Colleges* 19, 1 (October 2003), 122–131.

Narasimha Bolloju and Felix S. K. Leung. 2006. Assisting novice analysts in developing quality conceptual models with UML. *Communications of the ACM* 49, 7 (July 2006), 108–112. DOI:http://dx.doi.org/10.1145/1139922.1139926

Jonas Boustedt. 2012. Students' different understandings of class diagrams. *Computer Science Educ.* 22, 1 (2012), 29–62. DOI:http://dx.doi.org/10.1080/08993408.2012.665210

Pearl Brazier, Alejandro Garcia, and Abel Vaca. 2007. A software engineering senior design project inherited from a partially implemented software engineering class project. In *Proceedings of the 37th ASEE/IEEE Frontiers in Education Conference*. IEEE, F4D-7–F4D-12. DOI:http://dx.doi.org/10.1109/FIE.2007.4418071

Christopher H. Brooks. 2008. Community connections: Lessons learned developing and maintaining a computer science service-learning program. In *Proceedings of the 39th ACM Technical Symposium on Computer Science Education (SIGCSE'08)*. ACM, New York, NY, 352–356. DOI:http://dx.doi.org/10.1145/1352322.1352256

Janet Burge. 2007. Exploiting multiplicity to teach reliability and maintainability in a capstone project. In *Proceedings of the 20th Conference on Software Engineering Education and Training (CSEET '07)*, IEEE Computer Society, Washington, DC, 29–36. DOI:http://dx.doi.org/10.1109/CSEET.2007.22

Philip J. Burton and Russel E. Bruhn. 2004. Using UML to facilitate the teaching of object-oriented systems analysis and design. *Journal of Computing Sciences in Colleges* 19, 3 (January 2004), 278–290.

Joseph Chao and Mark Randles. 2009. Agile software factory for student service learning. In *Proceedings of the 22nd Conference on Software Engineering Education and Training (CSEET'09)*. IEEE Computer Society, Washington, DC, 34–40. DOI:http://dx.doi.org/10.1109/CSEET.2009.26

Tony Clear, Michael Goldweber, Frank H. Young, Paul M. Leidig, and Kirk Scott. 2001. Resources for instructors of capstone courses in computing. *SIGCSE Bulletin* 33, 4 (December 2001), 93–113. DOI: http://dx.doi.org/10.1145/572139.572179

Robert F. Dugan. 2011. A survey of computer science capstone course literature. *Computer Science Education* 21, 3 (2011), 201–267. DOI: http://dx.doi.org/10.1080/08993408.2011.606118

Alan J. Dutson, Robert H. Todd, Spencer P. Magleby, and Carl D. Sorensen. 1997. A review of literature on teaching engineering design through project-oriented capstone courses. *Journal of Engineering Education* 86, 1 (January 1997), 17–28. DOI:http://dx.doi.org/10.1002/j.2168–9830.1997.tb00260.x.

Sally Fincher, Marian Petre, and Martyn Clark. (Eds). 2001. *Computer Science Project Work: Principles and Pragmatics*. Springer-Verlag, London, UK.

Shayne Flint, Henry Gardner, and Clive Boughton. 2004. Executable/translatable UML in computing education. In *Proceedings of the 6th Australasian Conference on Computing Education - Volume 30 (ACE'04)*. Australian Computer Society, Inc., Darlinghurst, Australia, 69–75.

Andrew Forward and Timothy C. Lethbridge. 2008. Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals. In *Proceedings of the 2008 International Workshop on Models in Software Engineering (MiSE'08)*, ACM, New York, NY, 27–32. DOI:http://dx.doi.org/10.1145/1370731.1370738

Armando Fox and David Patterson. 2013. *Engineering Software as a Service* (2nd Beta Edition). Strawberry Canyon, LLC, San Francisco, CA.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, New York, NY.

Michael Gnatz, Leonid Kof, Franz Prilmeier, and Tilman Seifert. 2003. A practical approach of teaching software engineering. In *Proceedings of the 16th Conference on Software Engineering Education and Training (CSEET'03)*. IEEE Computer Society, Washington, DC, 120–128. DOI:http://dx.doi.org/10.1109/CSEE.2003.1191369

Lili Hai. 2009. The role of collaboration diagrams in OO software engineering student projects. In *Proceedings of the 22nd Conference on Software Engineering Education and Training (CSEET'09)*, IEEE Computer Society, Washington, DC, 93–100. DOI:http://dx.doi.org/10.1109/CSEET.2009.14

Brian Hanks. 2007. Becoming Agile using service learning in the software engineering course. In *Proceedings of the AGILE 2007 (AGILE'07)*, IEEE Computer Society, Washington, DC, 121–127.

Orit Hazzan and Yale Dubinsky. 2006. Teaching framework for software development methods. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, New York, NY, 703–706. DOI:http://dx.doi.org/10.1145/1134285.1134396

Gregory W. Hislop, Heidi J. C. Ellis, and Ralph A. Morelli. 2009. Evaluating student experiences in developing software for humanity. In *Proceedings of the 14th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'09)*. ACM, New York, NY, 263–267. DOI: http://dx.doi.org/10.1145/1562877.1562959

Youry Khmelevsky. 2009. SW development projects in academia. In *Proceedings of the 14th Western Canadian Conference on Computing Education (WCCE'09)*, ACM, New York, NY, 60–64. DOI: http://dx.doi.org/10.1145/1536274.1536292

Paivi Kinnunen and Beth Simon. 2012. Phenomenography and grounded theory as research methods in computing education research field. *Computer Science Education* 22, 2 (June 2012), 199–218. DOI:http://dx.doi.org/10.1080/08993408.2012.692928.

Craig Larman. 2005. *Applying UML and Patterns* (3rd ed.). Prentice Hall, Upper Saddle River, NJ.

Panagiotis K. Linos, Stephanie Herman, and Julie Lally. 2003. A service-learning program for computer science and software engineering. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'03)*. ACM, New York, NY, 30–34. DOI: http://dx.doi.org/10.1145/961511.961523

Gregory Madey, Curt Freeland, and Paul Brenner. 2005. A service learning program for CSE students. In *Proceedings of the 35th ASEE/IEEE Frontiers in Education Conference (FIE'05)*. IEEE, Los Alamitos, CA, F2F1–F2F6. DOI:http://dx.doi.org/10.1109/FIE.2005.1612055

Sandeep Mitra, T. M. Rao, and Thomas A. Bullinger. 2005. Teaching software engineering using a traceability-based development methodology. *Journal of Computing Sciences in Colleges* 20, 5 (June 2005), 249–259.

Tom Nurkkala and Stefan Brandle. 2011. Software studio: Teaching professional software engineering. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11)*. ACM, New York, NY, 153–158. DOI:http://dx.doi.org/10.1145/1953163.1953209

Anne L. Olsen. 2008. A service learning project for a software engineering course. *Journal of Computing Sciences in Colleges* 24, 2 (December 2008), 130–136.

J. H. Paterson, K. F. Cheng, and J. Haddow. 2009. PatternCoder: A programming support tool for learning binary class associations and design patterns. *ACM Transactions on Computing Education* 9, 3 (September 2009), 16:1–16:22. DOI: http://dx.doi.org/10.1145/1594399.1594401

Keith R. Pierce. 1997. Teaching software engineering principles using maintenance-based projects. In *Proceedings of the 10th Conference on Software Engineering Education and Training*. IEEE Computer Society, Washington, DC, 53–60. DOI:http://dx.doi.org/10.1109/SEDC.1997.592439

Sarnath Ramnath and Brahma Dathan. 2008. Evolving an integrated curriculum for object-oriented analysis and design. In *Proceedings of the 39th ACM Technical Symposium on Computer Science Education (SIGCSE'08)*. ACM, New York, NY, 337–341. DOI:http://dx.doi.org/10.1145/1352135.1352252

Pierre N. Robillard, Philippe Kruchten, and Patrick d'Astous. 2003. *Software Engineering Process with the UPEDU.* Addison-Wesley.

Brian J. Rosmaita. 2007. Making service learning accessible to computer scientists. In *Proceedings of the 38th ACM Technical Symposium on Computer Science Education (SIGCSE'07)*. ACM, New York, NY, 541–545. DOI:http://dx.doi.org/10.1145/1227310.1227493

Richard A. Scorce. 2010. Perspectives concerning the utilization of service learning projects for a computer science course. *Journal of Computing Sciences in Colleges* 25, 3 (January 2010), 75–81.

Ven Yu Sien. 2011. An investigation of difficulties experienced by students developing Unified Modeling Language (UML) class and sequence diagrams. *Computer Science Education* 21, 4 (2011), 317–342. DOI:http://dx.doi.org/10.1080/08993408.2011.630127

John Slimick. 1997. An undergraduate course in software maintenance and enhancement. In *Proceedings of the 10th Conference on Software Engineering Education and Training*, IEEE Computer Society, Washington, DC, 61–73. DOI:http://dx.doi.org/10.1109/SEDC.1997.592440

Juha Sorva. 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education* 13, 2, Article 8 (June 2013), 31 pages. DOI:http://dx.doi.org/10.1145/2483710.2483713

Davor Svetinovic, Daniel M. Berry, and Michael W. Godfrey. 2006. Increasing quality of conceptual models: is object-oriented analysis that simple? In *Proceedings of the 2006 International Workshop on Role of Abstraction in Software Engineering (ROA'06).* ACM, New York, NY, 19–22. DOI: http://dx.doi.org/10.1145/1137620.1137625

Claudia Szabo. 2014. Software projects are not throwaways: Teaching practical software maintenance in a software engineering course. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE'14)*. ACM, New York, NY, 55–60. DOI:http://dx.doi.org/10.1145/2538862.2538965

Robert Szmurlo and Michal Smialek. 2006. Teaching software modeling in a simulated project environment. In *Proceedings of the 2006 International Conference on Models in Software Engineering (MoDELS'06)*. Springer-Verlag, Berlin, 301–310.

Joo Tan and John Phillips. 2005. Incorporating service learning into computer science courses. *Journal of Computing Sciences in Colleges* 20, 4 (April 2005), 57–62.

M. H. N. Tabrizi, Carol B. Collins, and Vipul Kalamkar. 2009. An international collaboration in software engineering. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE'09)*. ACM, New York, NY, 306–310. DOI:http://dx.doi.org/10.1145/1508865.1508976.

Josh Tenenberg. 2010. Industry fellows: Bringing professional practice into the classroom. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)*. ACM, New York, NY, 72–76. DOI:http://dx.doi.org/10.1145/1734263.1734290

Benjy Thomasson, Mark Ratcliffe, and Lynda Thomas. 2006. Identifying novice difficulties in object oriented design. In *Proceedings of the 11th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'06)*, ACM, New York, NY, 28–32. DOI:http://dx.doi.org/10.1145/1140124.1140135

Mathupayas Thongmak and Pornsiri Muenchaisri. 2002. Design of rules for transforming UML sequence diagrams into Java code. In *Proceedings of the 9th Asia-Pacific Software Engineering Conference (APSEC'02).* IEEE Computer Society, Washington, DC, 485–494.

Fang Wei, Sally H. Moritz, Shahida M. Parvez, and Glenn D. Blank. 2005. A student model for object-oriented design and programming. *Journal of Computing Sciences in Colleges* 20, 5 (May 2005), 260–273.

Whatsa. 2012. Whatsa Controller Anyway. Retrieved on June 20, 2012 from http://c2.com/cgi/wiki?WhatsaControllerAnyway/.

Received June 2012; revised April 2014; accepted May 2014