A Simple Methodology for Applying UML to Database Design

Kevin Huggins[1], Eugene Ressler[2]

[1] Centre de Recherche en Informatique, École des Mines de Paris,
77300 Fontainebleau, France
Huggins@cri.ensmp.fr
http://www.cri.ensmp.fr
[2] Department of Electrical Engineering and Computer Science,
US Military Academy,
West Point, New York, 10996, USA
Ressler@usma.edu
http://www.eecs.usma.edu

Abstract. We present experience and assessment results for the use of UML in an undergraduate database design course. We employed an abbreviated iterative design methodology based on UML in a semester-long course project, with students learning about relational databases and SQL while simultaneously designing and implementing a simple system. UML replaced ER diagrams used in earlier versions of the course in order to support an effort to use UML across the curriculum. We found UML to be an adequate substitute for ER diagrams and that our thought processes in adopting UML led to enhancements of the course. We conclude that an undergraduate textbook employing UML to illustrate relational database design would be highly effective.

Keywords. database modeling, enterprise case studies.

1 Introduction

The Unified Modeling Language (UML) is a broadly accepted standard for modeling software applications. It is extremely rich and flexible, able to express diverse aspects of a development project. However, this flexibility and richness come with a price. The UML is large and complex, requiring significant effort to understand and apply. Moreover, the UML is not a design methodology. To use it effectively, practitioners must first find or develop their own. These challenges are often obstacles for students or practitioners seeking to use UML for the first time, when available effort to learn UML constructs is likely to be constrained. We recently faced these hurdles at the U.S. Military Academy Department of Electrical

Engineering and Computer Science, where UML has been adopted as a curriculum-wide standard design tool.

This paper concerns the experience of adopting UML to replace Entity-Relationship (ER) diagrams in a database design course, along with other supporting changes. In section 2, we describe the underlying structure of the course. Section 3 covers details of how we used UML. In section 4 we provide results and observations

2 Course Description

The primary desired outcome of our course is for students to be able to analyze a moderately complex user requirement, then design and implement in Structured Query Language (SQL) a well documented database that meets the requirement. This goal imparts three characteristics on the structure of the course. First, for the simple reason that demonstrating a working system appears to be the only way that goal accomplishment can be assured, this course is based on a semester-long design and implementation project. In order to engage students, we allow them, in teams of two or three, to propose their own project ideas. Refining the proposal into a problem statement that is both adequate and reasonable for students to accomplish in one semester is the first stage of design.

Second, the course schedule allows for two full project design iterations before final implementation late in the course. This choice is based on assessment of an earlier scheme that allowed for only one design iteration. Students new to databases and SQL inevitably make errors in analysis and design at the outset. Often these are "fatal"—they would certainly result in an unsuccessful implementation. If there is only one design iteration allowed, the instructor is left with a choice of two evils. One is to fix the error for the student, missing a great opportunity for the student to learn about his/her problem and design in general. Alternately, the instructor can allow the student to proceed with a flawed design and finish with a failed software system. In either case, the course goal is compromised. With two design iterations, the choice of two evils is unnecessary. Students have a mid-course opportunity to *independently* repair their bad choices made in the first iteration by employing their, by now, richer experience and confidence with databases and SQL. This maximizes the opportunity for each student to learn through discovery and to achieve the course goal of a successful implementation.

The third characteristic of the course is an abbreviated design methodology. This follows directly from the limited time available to complete each iteration. The methodology has four stages: Problem Definition, Analysis, Design and Implementation. Each stage culminates in a carefully structured and graded *minimal* documentation requirement. This documentation enforces discipline in each stage without excessively burdening the student or instructor. It uses UML as it is intended to be used—a succinct way of communicating analysis and design

information among designers and to others outside the design team, in this case the instructor.

3 UML Instruction

In this section, we detail the challenges of employing UML within the framework of small relational database system design and the effort-constrained environment of a single undergraduate course. We describe the techniques we used to overcome each of these challenges.

3.1 Challenges

UML is commonly described as a “box of tools,” where each tool is itself a language or convention for expressing some aspect of software design. It is meant to support large, complex projects in industrial and commercial setting. As such, its collection of tools is large and diverse. This largeness is both UML’s strength and its weakness. Given a software design problem, UML is likely to contain a tool that will express its solution. On the other hand, finding and learning the correct tool entails considerable effort. For a single undergraduate course, where effort is strictly limited, a student must be guided to the correct tool, and its usage must be communicated and mastered as efficiently as possible.

Therefore, our first challenge was to select a minimal appropriate UML subset to support design of small systems for SQL targets such as those in our course projects. We began by looking for UML equivalents for the constructs of ER diagrams, the well-known modeling technique we used previously for relational database problems. UML class diagrams served well in most cases, with UML classes standing for ER diagram entities. However, in a few instances, novel adaptations were necessary. For instance, ER diagrams include weak entities—those that have no unique identity without a related strong entity (one with a primary key). UML has no “weak classes.” Rather than create a non-standard extension to UML, we instead created a UML stereotype called “weak,” then composed this stereotype with classes corresponding to weak entities. An example is shown below.

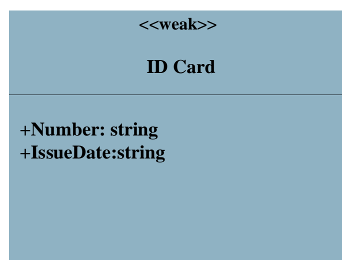


Fig. 1. This is an example of a UML class representation of an ER weak entity.

A second consideration was the timing for presentation of UML and its constructs in the course schedule. We decided on an initial overview covering the nature and purpose of UML during the introduction of the Analysis stage of design. Constructs are introduced "just in time" as needed. For instance, use cases are introduced first in order to illustrate the concept or system stakeholders. Object and class diagrams follow in order to provide a way for students to enumerate classes and objects, their attributes, and the ways they are related. However, only many-many relations are introduced at this early stage. More restrictive relations (1-to-1 and many-to-1, etc.) are best introduced later, i.e., after the concept of foreign keys.

Finally, we found a dearth of textbooks about UML suitable for people without a preexisting strong background in system design.

3.2 Implementation

The diagram below, which is presented to students during the second lesson of the 40-lesson course, shows the two design iterations of the project.

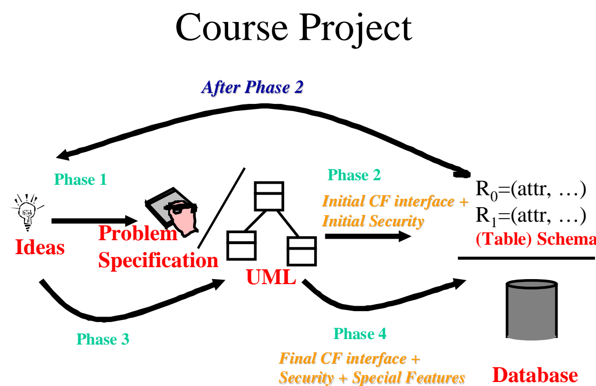


Fig. 2. The course plan shows the two-iteration, four-phased approach used.

Phase 1 ends with a written specification of the project chosen by the student team. It doubles as a contract with the instructor for the problem to be solved by the completed system. It includes UML object diagrams, use cases and a rudimentary class diagram based on the students' early learning in the course. The class diagrams frequently contain naïve errors. These are expected because the student, as yet, has only beginning experience with relational databases and SQL. The instructor may indicate the existence of these errors in feedback to the student, but

he or she *does not provide corrections.* These are for the students to discover later. During Phase 2, the class diagram is implemented as a database schema using only many-many relations. Prototype screens for a web interface, based on Cold Fusion or PHP technology, are also part of the Phase 2 submission. The design must include a proposed security mechanism for enforcing roles identified in use cases. Phase 3 requires refinement of use cases, classes, and relationships. Nearly all Phase 3 designs reflect significant overhauls of Phase 1 that are based on students' better appreciation for how they will ultimately be implemented in SQL tables. The Phase 3 submission is the instructor's final opportunity to influence designs prior to implementation. Our philosophy has been to coach each team to a design that has a feasible implementation by this point so that each team has at least the potential to produce a working system.

3.2.1 Object Diagrams

We introduce the Analysis stage of design by having students enumerate all nouns and verbs in various problem specifications, including their own projects. Nouns are potential classes and objects. Verbs are potential associations. Practice with several problems during class and in homework equips students to produce prospective object diagrams for project Phase 1. These are presented along with problem specifications in a face-to-face meeting approximately one week before Phase 1 is due. This "Interim Progress Review" has proven to be extremely valuable. It encourages prudent student time allocation to the project. It also allows for erroneous approaches to be corrected.

3.2.2 Use Cases

The use cases submitted with Phase 1 have two important purposes. First, they require the student to consider user roles and privileges in their systems. Hence, most designs naturally include a user class with privilege attributes. Second, they support early and careful thinking about the user interface screens that will ultimately be implemented in Cold Fusion or PHP in the finished system.

We found that neither written use cases nor the use case diagram alone was adequate. The written document encourages detailed thinking, while the diagram encourages an overall view of structure, especially for visual learners. Instructors learned to require cross-indexing information between the written cases and the diagram so that the latter could serve as a visual index for the former, a great efficiency for grading.

A representative use case diagram is shown below.

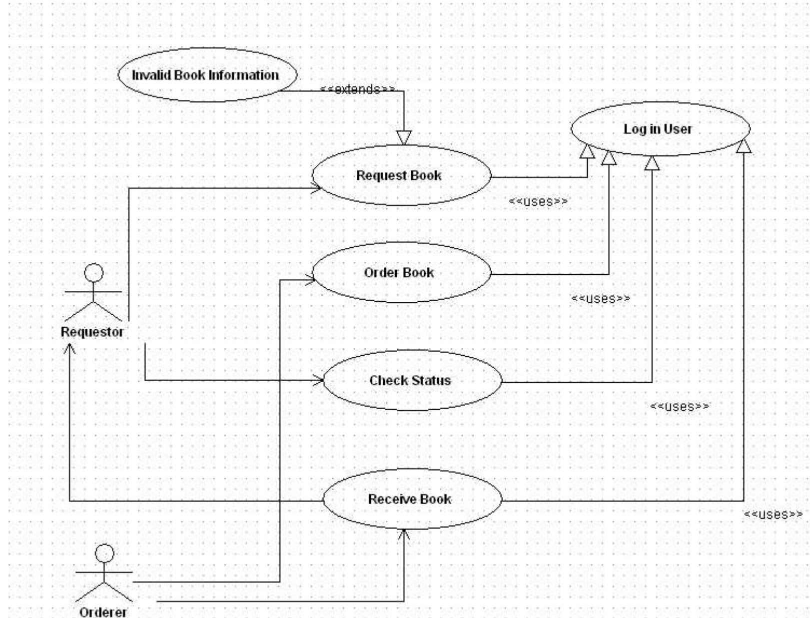


Fig. 3. Example of a Use Case diagram.

We prescribed an abbreviated and fairly structured “two-column” written use case format. An example is shown here.

Use case name: Request book
Actor: Requestor
Typical Course of Events:

Actor	System
1.Requestor enters log in credentials	
	2. Use <<Log In User >> (**this refers to another use case**)
3. Requestor inputs information on book	
	4. System checks to ensure Requestor's book information is valid. 5. System stores Requestor's book information 6. System notifies Orderer that a books needs to be ordered 7. System provides the user a confirmation
8. Requestor Logs off the system	

Fig. 4. Example of a Use Case write-up format we used in the course.

While UML admits a broad range of formats, this structured approach effectively provided “training wheels” for students writing use cases for the first time. The restricted format proved more than adequate for the moderate size of student projects. Some teams revised their restricted Phase 1 use cases into full narratives in Phase 3. We enthusiastically supported such initiative, since it reflected growing skills and appreciation for the value of analysis.

3.2.3 Class Diagrams

We used class diagrams to provide a logical view of the database design. Class diagrams replaced the ER diagram. As with use case diagrams, we employed only the minimal subset of class diagram structures available in UML.

During the first iteration, we introduced only the simplest concepts needed to express a model of a database. In particular, we covered classes and many-to-many associations. With these two features, the student teams made the initial model of their database.

During the second iteration, we introduced more advanced topics. We covered one-to-many associations, one-to-one associations, super-/sub-class associations, weak classes, and associations with existence dependencies. Since students had already had a chance to gain confidence with the simplified UML constructs from the first iteration, it was an easier step comprehension-wise for them to grasp the more advanced concepts.

3.2.4 UML to Schema Conversion

The bridge between design and implementation was the conversion of the UML diagram into schema, which our students finally used to create relational tables with SQL tools.

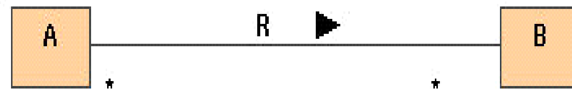
To impose structure on learning in this area, we developed a UML-to-Schema conversion guide. The guide consists of a set of pattern matching rules that allow a class diagram to be translated to a database schema almost automatically. The rules are organized with the most frequently used and necessary ones first. As such, students can construct usable databases from Phase 1 class diagrams after studying only two rules. These two most basic conversion rules, for strong classes and many-to-many associations, are shown here.

Convert strong classes to table schema

if key-attributes$_A$ = { keyAttr$_1$, …, keyAttr$_j$ }
and other-attributes$_A$ = { singleAttr$_1$, …, singleAttr$_k$ }
then A-schema = ($\underline{\text{key-attributes}}_A$, other-attributes$_A$)

Fig. 5. Guide to converting a strong class to table schema.

Convert all many-to-many associations to table schema.



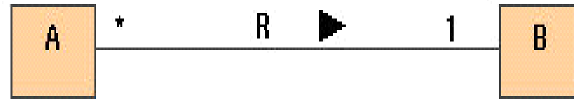
if A-schema = ($\underline{\text{key-attributes}}_A$, other-attributes$_A$)
and B-schema = ($\underline{\text{key-attributes}}_B$, other-attributes$_B$)
then R-schema = ($\underline{\text{key-attributes}}_A$, $\underline{\text{key-attributes}}_B$, other-attributes$_R$)
with a foreign key from R-schema.key-attributes$_A$ to A-schema.key-attributes$_A$
and a foreign key from R-schema.key-attributes$_B$ to B-schema.key-attributes$_B$

Fig. 6. Guide to converting a many-to-many association to table schema.

These rules alone support the student through preparation of their Phase 2 submissions, while they are simultaneously learning more complex associations and the concomitant conversion rules in preparation for later phases

In the last section of the guide, we provide ways to optimize the schema. An example, which is to drop an association table when the association is an existence dependency, is shown below.

Use existence dependencies to eliminate unneeded many-to-one table schema



if A-schema = (key-attributes$_A$, other-attributes$_A$)
and B-schema = (key-attributes$_B$, other-attributes$_B$)
and R-schema = (key-attributes$_A$, key-attributes$_B$, other-attributes$_R$)
with foreign keys as defined earlier
then drop R-schema
and alter A-schema = (key-attributes$_A$, other-attributes$_A$, key-attributes$_B$, other-attributes$_R$)
with a foreign key from A-schema.key-attributes$_B$ to B-schema.key-attributes$_B$

Fig. 7. Guide to optimizing the schema by eliminating unneeded many-to one table schema. If the many-to-one association was an existence dependency association, the association table could be dropped and the table schema representing the class on the many side of the association (in this case A-schema0 would be altered.

4 Results and Observations

Students reacted very favorably to the methods described in this paper. During the first two years, all 60 teams who took the course completed acceptable databases. Approximately 95 per cent completed successful web interfaces. Our Academy-wide student feedback system results for this course placed it substantially above Department and Academy averages for the following questions:

– *My motivation to learn increased because of this course.*
– *The instructor stimulated my thinking.*
– *In this course my critical thinking ability increased.*
– *The instructor used effective techniques.*

Finally, we found that many teams spent far more than the hours planned for project completion. These teams produced near production-quality web interfaces, when the expectation was merely for a prototype capable of inserting, updating, and accessing test data.

It became clear after one year that nearly all projects entailed user authentication to enforce user roles. Therefore we now provide a carefully restricted example as a template in order to reduce implementation time for this particular feature.

While hard to quantify, the general quality of student projects clearly increased with respect to earlier versions of the course that employed ER diagrams. We credit

this to the combination of use cases, which add discipline to student thinking about design early on, and the move from one to two design iterations. While these could both have been employed with ER diagrams, we devised the changes as a result of thoughts about design methodology with UML constructs as tools. Therefore, it is still fair to say the UML led to the improvements we experienced.

Finally, anecdotal evidence suggests that students completing the course are both capable of producing useful systems and confident they can do it. On average, two teams per semester request that their systems remain in operation because they are serving a production purpose of some kind. In this, we oblige as resources are available. Other instructors in the computer science program report that students spontaneously reach for database tools when confronted with data search and report generation-type problems.

The negative aspects of the methods presented here include the high level of instructor experience and effort necessary to guide a diverse array of student projects to successful completion without compromising learning in any way. Consistent grading of diverse projects is another aspect of the course that demands practice and experience. We have found that even veteran instructors must teach in this course for two semesters before developing good judgment on what makes a suitable project and how to provide feedback on designs at each Phase in a way the maximizes learning. Team teaching, where a new instructor gradually takes over for an experienced one, is the best way to transition between faculty members teaching the course.

A second negative remains the dearth of textbooks that present design using UML at a level suitable for a first course in databases. Our reading list for this course includes a traditional database textbook [5] and a "how to use UML" reference book, [4] from which we draw applicable sections. We drew from other resources for to enhance the course preparation. [1], [2] The UML-to-Schema document ties these together logically. While these are adequate for our purposes, a unified treatment in a single textbook would be greatly preferable.

Finally, we note that technical support for many separate web-fronted database implementations is a significant task. Careful planning is necessary, and the likelihood that one poorly written query will crash the database for all students in the heat of project completion should be considered in advance.

5 Conclusion

We have presented our methodology for applying UML in a database design course. Our thought process in considering how to do this led us to strengthen our treatment of the Analysis stage of design by employing use cases. It also encouraged us to try completing two design iterations within the time of a single semester course. Both of these changes proved beneficial. We found that UML class diagrams with some novel adaptations make good substitutes for traditional ER diagrams. We noted good student feedback on the resulting course, as well as improved outcomes on projects and anecdotal evidence that students were

transferring their knowledge to other coursework and to applications outside class. We look forward to a textbook that better supports the structure of this course than those currently available.

References

1. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley (2000)
2. Muller, R.: Database Design for Smarties: Using UML for Data Modeling. Morgan Kaufmann (1999)
3. Naiburg, E, Maksimchunk, R: UML Database Design. Addison-Wesley Profession (2001), Berlin (1995)
4. Schmuller, J.: Sams Teach Yourself UML in 24 hours. Sams (1999), Berlin (1995)
5. Silberschatz, A, Korth, H, Sudarshan, S.: Database System Concepts. McGraw-Hill (1998)