TEACHING SOFTWARE ENGINEERING USING A TRACEABILITY-BASED DEVELOPMENT METHODOLOGY

Sandeep Mitra, T.M. Rao, Thomas A. Bullinger[1]
Department of Computer Science
SUNY Brockport
Brockport, NY 14420
585-395-2234
smitra@brockport.edu

ABSTRACT

Instructors of Software Engineering (SE) classes in small Computer Science programs face the challenge of selecting a *coherent set* of SE concepts that can be taught in a one-semester course. This challenge is more significant when the students' background lacks significant exposure to Object-Oriented (OO) design. A second challenge is the design of "right-sized" case studies that rigorously illustrate the application of the selected SE concepts, can be comprehensively presented in the classroom and assigned as term projects. In this paper, we address the first challenge by proposing the use of a *traceability-based* methodology for SE instruction. We address the second challenge by advocating the use of a particular type of problem – namely, "GUI-based workflow applications". We present an ATM case study in class and focus on how the issues inherent in this problem can be tackled by choosing appropriate OOSE concepts. We have found that this approach allows us to teach concepts such as design patterns and coding idioms as they *naturally arise* in the problem. Our experience with a small group of students indicates that, with this approach, they were able to better understand the SE process itself and come up with high-quality software designs in a similar term project.

INTRODUCTION

The Accreditation Board for Engineering and Technology (ABET) requires a Software Engineering (SE) course in the undergraduate Computer Science curriculum. In a small liberal arts college like SUNY Brockport, the curriculum can usually accommodate at most one or two courses related to SE. Typically, these courses are at the junior/senior level, and assume CS-1/CS-2 as pre-requisites. CS-1/CS-2 courses need to cover three interrelated types of knowledge of programming: syntactic

---

[1] ArchSynergy, Ltd., 2500 Turk Hill Road, Victor, NY 14654, tom@archsynergy.com

(specifics of the language and its rules for use), conceptual (constructs and principles of programming), and strategic (general problem-solving skills) [2]. Consequently, CS-1/CS-2 instructors primarily focus on language details and algorithms so that students can effectively develop *working* programs at the end of the course. With the recent emphasis on taking an Objects-first approach, and the use of OO languages like Java for instruction, students are exposed to the basic OO concepts such as classes, objects, interfaces and inheritance in CS-1/CS-2. Despite this, the students do not have any *significant experience* in designing OO systems. The first SE course is where they will begin to learn how to design high-quality OO systems. Consequently, the SE course not only has to teach the traditional SE material, such as the standard software development life cycle model, but it also has to ensure that the students learn how to *apply* the proper techniques at each phase of the life cycle so that they create a high-quality artifact at the end of that phase.

Most SE courses traditionally assign a significant *group project* whose goal is to provide practical experience to students. Such a project helps in emphasizing the importance of teamwork, and ultimately enables creation of a product that gives the students a sense of accomplishment. All of this has to be achieved in the space of one 14-week semester!

Designing the term project has always been a significant challenge during the six years that the first two authors have taught SE related courses at SUNY Brockport. The project should not be too trivial, for the students then feel that they could simply write the code itself in a couple of days, and any other activities (e.g., writing a UML model) are simply additional documentation work that is a burden. On the other hand, an "industrial-strength" project is not an option in an environment like ours, with a significant portion of part-time and commuter students who have multiple commitments, both academic and non-academic. Achieving anything of quality in a project of this magnitude at the end of a 14-week semester is difficult.

We have examined many SE textbooks searching for a "right-sized" group project. Most books cover a wide variety of concepts, including life cycle models (waterfall, spiral), methodologies (Rational Unified Process – RUP [10], Agile Methods such as eXtreme Programming – XP [3]), and related topics such as methods for capturing customer requirements, managing risks, handling change, etc. Almost all books advocate the adoption of the OO approach, and use UML for modeling activities (even in an Agile environment [1]!). Many examples are used to illustrate different aspects of software engineering, however none of the books presented a single coherent case-study and developed it from the initial problem statement, through all the stages of development, and finally to code. Further, the wide variety of concepts presented makes it impossible to cover in a one-semester course in an environment like ours.

USING THE SEEM™ APPROACH

The basic curriculum of our SE course requires us to teach the phases of a traditional software development life cycle – requirements gathering, system analysis, system design, implementation and testing. We focus on a *prescriptive process* – i.e., one in which we create models and artifacts corresponding to each phase. Our challenge, as college professors, is to identify the right set of concepts that need to be taught to illustrate the significance of each phase. This set of concepts should provide a good understanding of the nature of activities and deliverables expected in each phase. Moreover, such a set should also be "scalable" by providing a strong foundation on which our students can build further. For example, while we might not be able to

discuss change management in great detail in the course, we expect that the students will have a good understanding of process so that they can appreciate where issues such as risk analysis, rationale management, etc. will need to be done in order to handle a particular change properly.

In 2002, we began collaborating with ArchSynergy, Ltd. – a hi-tech startup company in our area focusing on software development processes and architectures. ArchSynergy's Software Engineering Effectiveness Model (SEEM™) provided us with the right set of concepts. A full discussion of SEEM™ is beyond the scope of this paper, and is available at http://www.archsynergy.com. We present below the most important aspects of the SEEM™ approach that enabled us to be effective in teaching our SE course:

i. We need to pay significant attention to *understanding* the *problem* to be solved. Documenting and communicating this understanding using artifacts that enhance precision, unambiguity and consistency is also very important. Consequently, we need to use modeling notations like UML at the very *first* stage of the development process. SEEM™ provides a well-defined set of guidelines to make such modeling *actionable*, as we discuss later in this paper.

ii. We need to take the problem domain model forward to the solution domain in a precise and systematic manner. *Traceability* between the problem domain model, solution domain model, and the implementation model must be clearly and obviously maintained (as emphasized by Jacobson in [9]). Only by ensuring such traceability can we teach the techniques to arrive at good OO designs.

    SEEM™'s approach to traceability differs from the traditional *traceability matrix* ([12]) approach, which can only be used to check for consistency between two independently developed models. The SEEM™ approach considers traceability as a *mapping strategy* that is used to create the more detailed, later phase, model(s), from the more abstract earlier phase models. The ultimate result is implementation code that contains no surprises, because it flows naturally from the model artifacts. SEEM™ makes these mapping strategies as *explicit* as possible for a certain type of system, as we show in this paper. Such an approach can be a very good pedagogical tool, as student comments at the end of this paper indicate.

iii. Testability of the artifacts illustrating the individual models must be ensured at each phase. In earlier phases, testability implies that it should be possible to analyze the models themselves to draw conclusions about consistency and correctness. Space constraints do not allow us to focus on this aspect of the course in this paper.

We focused on teaching concepts pertaining to (i) and (ii) above by illustrating them *fully* and *completely* in a *single* case study. The need to use a single case study is very important, for it provides the focal point for illustrating otherwise diverse concepts. Our approach to traceability, which stresses the continuity from initial problem model to code, makes it feasible to use a single case study, something that we have not seen in much of the available literature. The choice of the case study is very important, as it must focus on a particular "type" of system. We ensure that the group project given to the class is also of the same type.

The rest of this paper describes a case study that we have used: a system that simulates a bank ATM. The problem description requires letting the user check the balance,

deposit, withdraw and transfer money between his/her accounts. We call such systems "GUI-oriented workflow systems" – systems that will eventually be implemented completely on a desktop computer, have a GUI at the front-end, and a repository at the back-end for persistent storage

THE BANK ATM CASE STUDY

Defining the Problem

The SEEM™ approach to defining the problem is similar to RUP/UML, except that SEEM™ uses the XP term *user stories* (instead of use cases), in order to emphasize the fact that the user stories and descriptions should be entirely in the problem domain, with no hint whatsoever of solution domain entities. In order to ensure rigorous compliance with this need, the students are asked to write up the sequence of events in each user story thinking on the lines of how it would happen *completely manually* – with no hint of a computer whatsoever. In this context, SEEM™ also strongly recommends the use of the XP approach of discovering a *metaphor* for the system under development. This is not an easy task, and the students are asked to tackle it by considering what real world, day-to-day familiar situation is most similar to the project problem. For the bank ATM system, the metaphor is the bank of yesteryear, with no computers around. Obvious user stories for such a bank are: "Deposit Money," "Withdraw Money," "Balance Inquiry" and "Transfer Money." The sequence of events associated with the "Withdraw Money" user story would be: (i) Customer presents identification to Teller; (ii) Teller validates identification and informs the customer; (iii) Customer indicates to Teller which account she wishes to withdraw from, and the amount; (iv) Teller looks up the account information in a ledger; (v) Teller verifies that the account has enough balance to cover the withdrawal and informs the customer; (vi) Teller withdraws the money and updates the ledger with the new balance; (vi) Teller presents withdrawn money to the customer.

This sequence needs to be modeled further at the Problem Analysis stage using a UML sequence diagram. In order to ensure that such modeling will maintain traceability to the defined sequence, the students are asked to check that each element of the sequence is written in active voice, and *also* in the following form: WHO does WHAT to WHOM. With this approach, the desired behavior, as well as the entities associated with that behavior, is identified simultaneously. We have seen that if the students realize the need to identify behavior and associated entities together at this phase, then they come up with a more natural and therefore maintainable/extensible object model at later phases.

Problem Analysis – building a precise and unambiguous Problem Model

Having written the user stories, the next step is the creation of a UML sequence diagram that pictorially depicts, and further clarifies, the sequence of events for each user story. It is very important that *no* solution domain entities should be considered at this phase. Figure 1 shows a UML sequence diagram for the "Withdraw Money" user story considering only problem domain entities.

In identifying the entities in the diagram, the approach is to model any "eternally existing" entities as actors. Since the customer, teller and repository will always exist irrespective of whether this transaction happens or not, they are actors. Other entities are created for the duration of the transaction. When the *system boundary* [11] for

implementation is identified at the design phase, some of these actors may be transformed to system entities (objects).

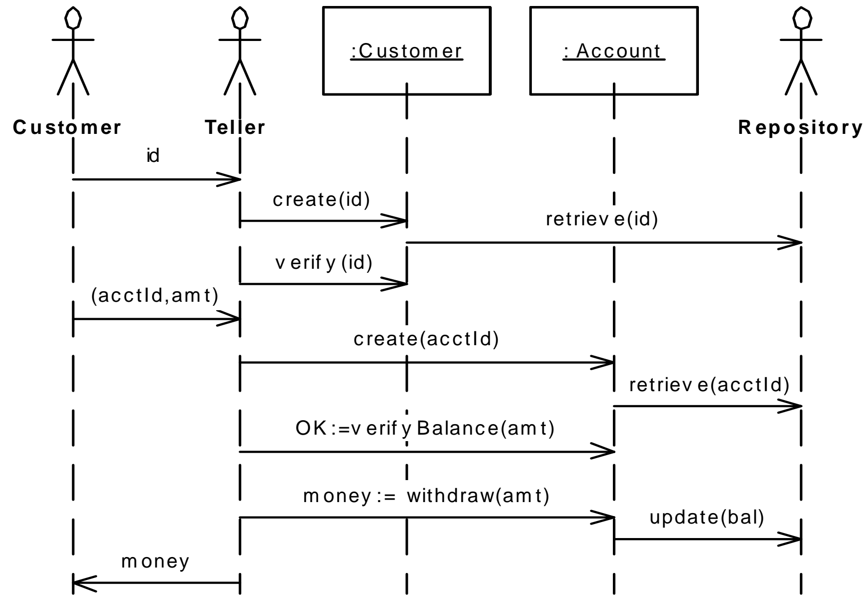

Figure 1. UML Sequence diagram for "Withdraw Money"

Note that the workflow shown in the sequence diagram traces exactly to the sequence of user story events indicated in the previous section. The students are taught that if such traceability is not obvious – that each element of the sequence cannot be mapped obviously to one or more interactions in the sequence diagram, then you need to reconsider one or both of them. A review of the artifacts (user story, sequence diagram) will detect any traceability problems. SEEM™ advocates "Progressive Learning" [5] – you may learn as a result of the review that a particular element of the sequence (which you wrote up in natural language, after all!) cannot be achieved in a more formal setting like UML.

In addition, each sequence diagram must be checked for *internal consistency*. Such a check involves verifying that every object interacts with only those objects it "knows about". For example, in Figure 1, the *Teller* can ask the *Account* object to withdraw money because the *Teller* instantiated this object after the customer provided it with the account Id. Consequently, the *Teller* can "remember" this object and talk to it when needed. Every interaction must be analyzed in this manner, and detected inconsistencies resolved, perhaps by going back to the user story's flow of events.

The principal objective of this phase is to clearly identify the *workflow* associated with each user story precisely and unambiguously in the sequence diagrams. Also, the workflow(s) must be abstract enough to be simple and understandable. If this goal is accomplished, these workflows can be *systematically mapped* to the design level artifacts. Besides the sequence diagrams, class diagrams showing discovered relationships between entities are created. For example, in this case, the *Customer* and *Account* objects must be *Persistable* objects – namely, those that can be written to/read from a repository. This is shown in a class diagram. CRC cards [7] are also created, but these are more important to the design stage as discussed in the next section.

Solution Design

This is the phase where aspects of the solution domain (i.e., the implementation environment) have to be considered. As this is a GUI-oriented workflow system, the execution of a user story is initiated from a GUI. In order to design the GUI correctly, the *system boundary* must be identified. This task results in the realization that the *Teller* “actor” in Figure 1 is an entity that needs to be part of the system. Therefore, we transform the *Teller* actor into a *Teller* “object” as shown in Figure 2. The repository remains outside the system – it will be accessed/updated using a repository access subsystem.

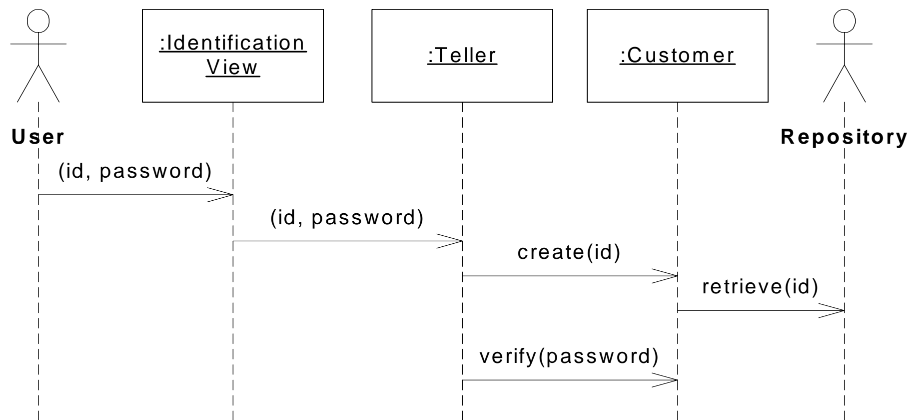

Figure 2. Mapping using MVC

At this point, a GUI mockup of the entire implementation is created. Students are required to create these mockups using pen-and-paper, or a non-programming tool like PowerPoint/HTML, but *not* use a programming tool like VB! Otherwise, the temptation to somehow reuse the code at the actual implementation phase is too great, and this can break traceability. Due to space considerations, we cannot show the GUI mockups here. For the “Withdraw money” user story above, the first GUI screen is the one in which the user enters her id and password, and after these are verified, she is presented with a screen on which she enters (or chooses) her account number and enters the withdrawal amount.

The main question now is to figure out how the behavior in Figure 1 will be realized in the solution domain consisting of a computer showing GUIs. The critical requirement is that the design phase sequence diagram be traceable from, and to, the analysis phase diagram. *Mapping strategies* need to be used here. One such strategy that applies for GUI-oriented workflow systems is the use of the Model-View-Controller (MVC) [6] pattern. Students are taught that any *system entity that an external user interacts with is a Model object and it has an associated view and a controller*. The user actually interacts with the view/controller. Since the customer interacts with the *Teller*, *Teller* is therefore a Model object and its initial associated view/controller, as shown in Figure 2, is the *IdentificationView* object. Thus, the *Teller* actor in Figure 1 maps to the *Teller* model and *IdentificationView* view/controller objects in Figure 2.

The correctness of the mapping of the behavior shown in Figure 1 to Figure 2 can be verified by realizing that the id provided by the *Customer* (Figure 1) maps on to an id and password combination, which is actually typed in to *IdentificationView* (Figure 2). Upon user action such as a button click this information is passed on to the associated model – i.e., the *Teller*. Thus, the first interaction in Figure 1 from *Customer* to *Teller* maps on to a sequence involving *User* (map of *Customer* actor) and the *IdentificationView* **and** *Teller* combination (map of *Teller* actor). The interactions showing the instantiation of the *Customer* object and the need to go to the repository

remain. The "verify" interaction at this phase checks the password, whereas at the analysis stage it had referred to a manual activity like matching the face on the driver's license with the customer.

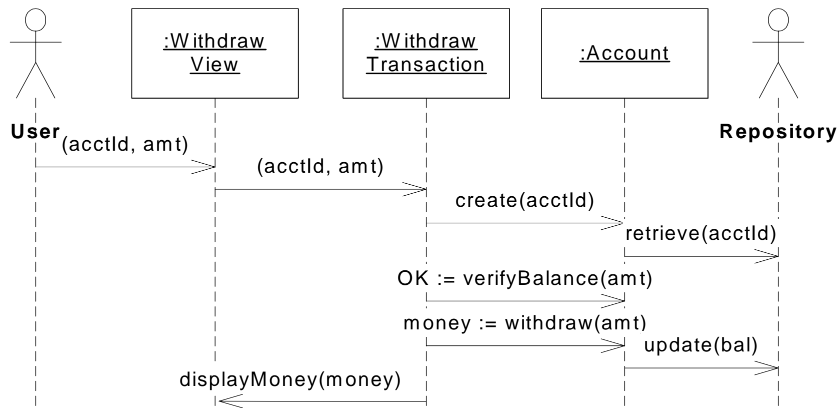

Figure 3. Mapping using "responsibility redistribution"

CRC cards are created at this phase as a "union" of all responsibilities for each entity, considering the interactions that come in to this entity in *all* the sequence diagrams that that entity participates in. Considering Figure 1 and similar diagrams for other transactions, the CRC card for *Teller* reveals that the "bulk" of the responsibility for carrying out *all* transactions (withdraw, deposit, transfer, etc.) is borne by the *Teller*. Relative to other entities, the *Teller* carries too many responsibilities, implying that its code might become too complex. A "responsibility redistribution" strategy can be used to rectify this situation. We now show how we accomplished such redistribution.

The main reason for the complexity is that the protocols between the *Teller* object and the *Account* object(s) are complex, due to the fact that the sequence of interactions between *Teller* and *Account* are different depending on the transaction. We need to re-distribute the protocol to other object(s). Consider the *WithdrawTransaction* object and its associated view shown in Figure 3, which is a continuation of the sequence diagram shown in Figure 2. Upon verification of the password (end of Figure 2), the *Teller* object (present, but not shown in Figure 3) actually creates a *WithdrawTransaction* object, passing it the *Customer* object. The *Teller* then simply asks the *WithdrawTransaction* object to "do its stuff", which results in this object showing its view to the user. Subsequent behavior is shown in Figure 3. It is to the *WithdrawView* object that the user enters her account number and withdrawal amount. We can now see that (according to Figure 1) the (*acctId*, *amt*) interaction that would have gone to the *Teller* object (or its view) now goes to the *WithdrawTransaction* object (via its view). Also, some interactions that would have come from the *Teller* object (such as the creation of the *Account* object and subsequent balance verification and withdrawal operations on it) now come from the *WithdrawTransaction* object. Thus, we say that we have decomposed the problem domain *Teller* entity and mapped it to *both* the *Teller* and *WithdrawTransaction* objects, and their views (Figures 2 and 3). Correctness of such decompositions are checked by ensuring that all interactions that came *in* to the original object in the earlier (analysis) phase come in to one of the mapped objects; similarly, all interactions that came *out* of the original object come out from one of the mapped objects. If, at the design level, an interaction is identified as having a different source or destination from that shown at the analysis level, and the different source/destination is not something that can be identified as a mapped object, then this is a *traceability mismatch*. This can be rectified by re-considering all existing project artifacts.

In our ATM example, the *Transaction* object serves as an *adapter* to the *Account* object that is used by the *Teller*. As a result, the *Teller*'s responsibilities are reduced significantly, thus creating a more *cohesive Teller* object. There is a different *Transaction* object specific to each transaction, and since in every user story the *Teller* object simply asks the *Transaction* object to "do its stuff", the *Teller* has low *coupling* to the *Transaction* object.

This exercise illustrates how we can introduce the MVC and Adapter patterns [8] in the *context of the problem to be solved*. By observing design patterns this way, students appreciate the real benefit of using the patterns – e.g., they see how the pattern can be used to achieve better cohesion and lower coupling.

Implementation

The CRC cards of the design phase are the starting points of the implementation. The challenge here is the mapping of the design sequence diagrams to the code in class methods. Again, we take recourse to strategies based on patterns. Patterns used in this mapping are called *coding idioms* [12], and they illustrate the realization of design patterns in code (e.g., Java). We present one particular example of using such patterns based on Figure 3. Our objective is to achieve the implementation of MVC logic in a high cohesion/low coupling manner. Consider the workflow in Figure 3 where the *WithdrawView* sends an interaction containing the account number and withdraw amount to its corresponding model object – i.e., *WithdrawTransaction*. The Transaction object processes the data and calls the view back with the results to display – i.e., the amount withdrawn. We first realize that the code of the *WithdrawView* class needs to know (i.e., be coupled to) the *WithdrawTransaction* class so it can send it the user entered data. Let us assume that this is acceptable, and *WithdrawView* knows *WithdrawTransaction* as its associated model. However, the callback via the invocation of the *displayMoney()* method appears to imply that the *WithdrawTransaction* class *also* needs to be coupled to the *WithdrawView* class, which hinders the view "replace-ability" benefit of using MVC. Now, if we use the Observer design pattern [8], and the Java classes/interfaces *java.util.Observer* and *java.util.Observable* that enable the realization of this pattern [4], then the code of the Transaction object would extend the *Observable* class, and the view would implement the *Observer* interface. The view would then need to ensure that it adds itself as an *Observer* to the Transaction, and the call *displayMoney()* would actually be a call to the *update()* method of the *Observer* interface, thus reducing coupling. Another possible alternative is to use Java's *PropertyChangeListener* infrastructure, as outlined in [11]. Irrespective of which implementation strategy is used, note that we are essentially implementing a *workflow*, shown in a "picture", whose feature is a "call forward" from the view/controller to the model and later a "call back" from the model to the view. Such workflows imply use of the Observer design pattern, and you may choose any appropriate coding idiom for coding this pattern.

Students have indicated that learning design patterns/coding idioms with the main emphasis on the context in which they apply is very useful in enhancing their understanding. In fact, one of our students found a Senior Thesis idea after this course, where he began to research criteria to use to evaluate alternative coding idioms for the same design pattern.

We cannot provide the actual Java code here due to space restrictions. We end here by mentioning that the Persistable classes use a repository access subsystem via a standard interface that is – again – *traceable* from the problem model because it

enables the realization of the *retrieve()* and *update()* methods in Figure 1. The standardization of the interface implies that that the same subsystem can be used in any similar implementation, irrespective of the number of Persistable classes and the number of files in the repository (our SE course does not have a relational database pre-requisite). The fact that files with different content can be accessed via a standard interface is an important lesson learned here and serves to further illustrate the benefits of the strict traceability-oriented approach.

REACTIONS FROM STUDENTS

We have used the above approach to teach SE using the ATM simulation as the comprehensive in-class case study, and a simple *Hotel Reservation System* as the term project. The students were given a problem statement during the first week of the semester (indicating the need to reserve rooms, cancel reservations, check in/out, etc.) and asked to come up with an initial design of classes. At the end of the semester they were asked to compare these to their final designs. A sampling of the student reactions is paraphrased below:

i. "I was familiar with Use Cases and UML. However, I never knew how to make the transition from the use cases to the classes in the design. ... [This approach is] much more systematic and really let me get a good handle on process."
ii. "I really like the traceability. I had knowledge of process (RUP) in my work place, but no one there really understood it, or could teach me its essentials. With SEEM's approach to strict traceability, I realize how you can use design patterns in the right context to get a good design."
iii. "I have had a design pattern class before. But while my teacher there was really good and I really understood what the design patterns were, I never knew how they applied to a problem that was written in English."
iv. "…When we wrote the code for the Hotel Reservation system, we found a bug that we tried to fix in the code itself, like we have done in our other classes, and got totally lost. We seemed to be breaking things that were not broken before. Eventually, we gave up on that, and looked at where we went wrong in the first (analysis level) sequence diagram…."
v. "Using this process … code is a no-brainer."

The dominant theme in the students' self-assessment was that the SEEM™ process is "easily actionable", largely thanks to *widely applicable guidelines* it provides. A comparison of the designs done at the beginning of the semester with those done at the end also demonstrates that design skills have significantly improved.

CONCLUSIONS AND FUTURE WORK

We conclude that proper problem modeling and maintaining strict traceability are two of the most important principles to teach in an SE class. Critical to student success has been the use of the *same* case study throughout the class. No concept is ever taught without showing how it applies to the one system under development, and students clearly see how to apply these concepts to their own (similar) term projects, thus reinforcing learning.

The core base described in this paper can be further extended in advanced SE classes. Risk management, for example, can be taught by considering as high risk problems for

which the traceability mappings sequences are unknown. Change management is taught in the context of understanding the first artifact that will be impacted by change, and how it percolates through the artifacts up to the implementation via the traceability mappings. It is our intention to further explore the use of this approach to teach other SE concepts.

REFERENCES

[1] Ambler, S. W. *The Object Primer: Agile Model-Driven Development with UML 2.0.* Cambridge University Press, Cambridge, United Kingdom, 2004.

[2] Baldwin, L., and Kuljis, J., Learning Programming Using Program Visualization Techniques. In *Proceedings of the 34th Hawaii International Conference on Systems Sciences,* Maui, HI, January 3 – 6, 2001 (on CD-ROM).

[3] Beck, K. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, Reading, MA, 1999.

[4] Braude, E., *Software Design: From Programming to Architecture.* John Wiley & Sons, Hoboken, NJ, 2004.

[5] Bullinger, T. and Mitra, S. A holistic approach to Embedded Systems Development. In *Proceedings of the Embedded Systems Conference (West)* (San Francisco, CA, March 29 – April 1, 2004). CMP Media, Manhasset, NY, 2004 (available online and on CD-ROM).

[6] Deitel, M., and Deitel, P., *Java: How to Program.* Prentice-Hall, Upper Saddle River, NJ, 2004.

[7] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley – Pearson Education, Boston, MA, 2004.

[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[9] Jacobson, I. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Reading, MA, 1992.

[10] Kruchten, P. *The Rational Unified Process: An Introduction.* Addison-Wesley, Reading, MA, 2003.

[11] Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice-Hall, Upper Saddle River, NJ, 2000.

[12] Pressman, R. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New York, NY, 2000.