DESIGN PATTERNS: EVOLVING FROM PASSIVE TO ACTIVE LEARNING

James Vallino[1]

Abstract *- Students in Rochester Institute of Technology's Software Engineering program gain an appreciation for the importance of design in their second year when they work on a term-long team-based software project. Student comments often express an eagerness to be taught more about the design of larger software systems. Our next course, Engineering of Software Subsystems, aims to achieve that outcome. This paper describes the evolution of this design course. The course was initially delivered as three one-hour lectures and one two-hour lab per week. Particularly in lectures, the students were not engaged to actively learn the material. The course has taken several evolutionary steps moving from its initial low level of active learning to where it now is mostly under the control of student teams participating as active learners. Data from one offering of the course suggests improved course evaluation ratings and a noticeable increase in student appreciation for the textbook.*

Index Terms – Active learning, Design patterns, Problem-based learning, Software engineering education.

SOFTWARE DESIGN IN THE RIT PROGRAM

The program at Rochester Institute of Technology (RIT) leading to the Bachelor of Science in Software Engineering was the first undergraduate software engineering program in the United States. This program is administered by the Department of Software Engineering which is within the Golisano College of Computing and Information Sciences. The program graduated its first class in 2001 and was among the first software engineering programs to undergo ABET accreditation visits in the fall of 2002. From its inception the emphasis of our program has been to educate students with strong technical software engineering skills so they can start work as productive members on a software development team. In our view this requires a balance between the design of software systems and the process for developing those systems.

The students in our program spend their first year and one quarter of their second year studying the fundamentals of object-oriented programming. These courses cover topics in basic programming, object-oriented technology, data structures, and algorithms with simple complexity analysis. The courses are offered by the Department of Computer Science and are common for the Computer Science, Computer Engineering and Software Engineering programs. We expect this sequence of four 10-week courses to develop solid programming skills in the students. Design discussions stay at rather low levels considering questions, such as, which nouns might represent objects in the system or state within the objects and which verbs are behaviors in an object. Beyond this there is little discussion of overriding principles motivating the design activity.

The next course in the sequence is SE361 Software Engineering. The main component of this course is a term-long team-based project. Teams of 5 or 6 students are the norm. This course is an introduction to software engineering practice. It covers topics such as roles on a software development team, software development lifecycles, requirements specification, design principles, and user interface design. Each team develops a product following the lifecycle from requirements through to product delivery. It is in this course when they develop larger scale systems that our students first begin to appreciate the importance of design. In reflective comments at the end of the course students identify the need for the team to spend more time in design discussions before starting to write code. They also identify that their design skills are not really adequate to handle the design of these larger systems with a larger number of classes interacting in new ways. The students express an eagerness to be taught more about designing larger software systems.

The second course in our software engineering program is SE362 Engineering of Software Subsystems. The course is commonly referred to as the "patterns course" since design patterns are at the course's core. The course work is based on [1]. Since design is one of the pillars supporting our curriculum and the students are eager to gain more knowledge in the design of software systems it is very important that this course "gets it right." In this course we begin to cultivate an engineering perspective for the development of software systems.

SOFTWARE DESIGN PATTERNS COURSE

The overall objectives for our Engineering of Software Subsystems course were described in [2]. We define the learning objectives for all of our software engineering courses in terms of Bloom's Taxonomy of cognitive learning [3]. It is common for our lower-division courses to have learning objectives at only the lower four Bloom taxonomy levels. As seen in Table I this is indeed the case for the course discussed in this paper. Throughout the evolution of the pedagogy for this course these learning objectives have remained constant.

[1] James Vallino, Rochester Institute of Technology, Department of Software Engineering, 134 Lomb Memorial Dr., Rochester, NY 14623
jvallino@mail.rit.edu

0-7803-7444-4/03/$17.00 © 2003 IEEE **November 5-8, 2003, Boulder, CO**
33rd ASEE/IEEE Frontiers in Education Conference

TABLE I
LEARNING OBJECTIVES FOR RIT'S ENGINEERING OF SOFTWARE SUBSYSTEMS COURSE

Taxonomy Level	Learning Objectives
Knowledge	1. list the design pattern classifications 2. identify the classification of a pattern
Comprehension	1. contrast different implementations of a pattern 2. contrast the difference in intentions between structurally similar patterns 3. discuss the general effects of design pattern usage on design principles such as cohesion and coupling
Application	1. demonstrate the use of patterns in isolated software subsystems 2. apply appropriate patterns in the design of a small software system
Analysis	1. analyze the design of a software system to identify logical components 2. select appropriate design patterns to refactor an existing design 3. compare design tradeoffs between different patterns and/or different implementations of the same pattern 4. compare the benefit of pattern usage versus non-usage

REVIEW OF THE INITIAL APPROACH

The course described in [2] was initially taught in a traditional lecture/lab format. This format was "another day, another pattern" with each lecture covering material for another pattern. Labs in the first half of the 10-week term were one week exercises highlighting individual patterns. Students worked on a team project in lab activities through the second part of the term. The pedagogy for the lecture component was primarily didactic passive learning.

Our first step toward engaging the students as active learners occurred when we started teaching the course in a studio classroom. Each two-hour class session started with a traditional lecture on a pattern and immediately reinforced the material with a class exercise. This placed a constraint on the size of the problems that could be posed. After the initial lecture, there was often only 45 minutes for groups of 2 or 3 students to work on a class exercise. By necessity many of the exercises were guided programming assignments where the students were told to "put your code here". The groups had a strong focus on completing the exercise even though a 25% grading curve was meant to allow teams to explore the problem space and not get complete solutions. The interactions with the instructor were almost exclusively aimed toward solving the specific exercise and only rarely sought deeper insights into the design pattern being discussed. Also, because of the limited scope of each exercise it was not easy to direct the students toward reflection on what they had done or to make comparisons between different pattern topics.

This redesign of the course benefited from the immediate reinforcement of the material discussed in the first part of the class period by the exercises completed in the second half of each class period. This format, however, still had the instructor doing a large amount of lecturing on pattern topics at lower levels of the Bloom taxonomy. During the lecture time class assessment techniques such as "one minute papers" and "think-pair-share" were often used to move the students from a passive to active mode of learning. Still the students would come to class without doing any preparatory reading even though they knew that they would complete a class exercise on the new material. They counted on the lecture to provide the necessary background.

The insight for a potential way in which to address these lingering concerns came at the "How to Engineer Engineering Education" Workshop[4] in July 2002. Bucknell University sponsors this workshop as part of their NSF-sponsored Project Catalyst[5]. One of the last sessions at the workshop discussed the advantages and disadvantages of Problem-Based Learning (PBL)[6]. The next section provides background into problem-based learning pedagogy. Following that are the details of how our Engineering of Software Subsystems course adapted PBL as its pedagogy with "lectures on-demand."

PROBLEM-BASED LEARNING

Problem-based learning is methodology which fosters active learning by the students. This pedagogy has been applied in medical education[7] and there have been previous reports of its application to engineering education[8, 9]. Like all active learning approaches it centers learning on the student rather than the instructor. Using the solution of problems posed by the instructor to motivate learning, it shifts the instructor from the "sage on the stage" to a "guide on the side." There are several characteristics of PBL[6]. To gain the benefit of PBL small teams of students work on a set of open-ended problems. The instructor identifies suggested reference materials. Depending on the level of the students and the objectives for the course the instructor can provide more or less in the way of reference materials. For this second-year course it is absolutely essential that the students have good resources. For the course discussed in this paper [1] is an excellent reference. The student teams direct their learning activities to acquire the additional knowledge that the team believes is needed to generate a solution to the problem. Through this process the instructor is available as an additional resource and to ensure that a team does not get misdirected by its own efforts.

The PBL methodology mimics the professional practice of the engineering disciplines where most learning is motivated by efforts to solve a problem. With this methodology, the instructor has a mentoring role providing knowledge that is sought by active student learners.

PBL APPLIED TO TEACHING DESIGN PATTERNS

As noted in the review of the initial delivery in the studio classroom, too much passive learning remained. The first

part of most class sessions felt like a series of "another day another pattern" lectures.

Currently the course is divided into four units of material. For each unit, the students receive a list of reading assignments, a detailed specification of the learning outcomes for that particular unit and a unit assignment. The outcomes contribute to the overall outcomes for the course. Lectures are planned at the start of each unit. For the most part, additional lectures are delivered "on-demand" when a student places a lecture request in a course discussion group one day prior to class. Class time is spent with the unit teams working on unit activities.

Grading is evenly split between individual and team assessment. The individual assessment is a system design in unit 1 and a unit quiz at the end of the other three units. There are also individual mid-term and final exams. Students receive a team grade for all unit team activities. This grade is adjusted individually based on peer evaluations that are done by each unit team member.

The instructor's time during class is spent in discussions with each unit team. This is the instructor's time to gauge the progress of the team overall and individuals on the team. During these discussions the instructor can ask questions of the team or individuals for both class assessment and to stimulate thinking and learning about the material. As a class assessment the instructor may decide that the class is generally missing a particular point and this could trigger a lecture or discussion of some examples in the next class session. The team uses this time to ask questions for clarification of the unit assignments and to receive early feedback from the instructor on those assignments. The intention is that the students are responsible for reading the reference material. The discussions that take place during class time are, for the most part, addressing higher-level issues with software design using the design patterns under study in the course.

PATTERNS COURSE SYLLABUS

As noted above, RIT's Engineering of Software Subsystems course is divided into four units. The first unit (1 week) motivates the use of patterns. Each of the following three units (3 weeks) addresses specific design patterns. Our current syllabus covers the topics listed in Table II.

TABLE II
CURRENT COURSE TOPICS

Unit	Topics
1	Course Basics; Design Principles
2	Adapter, Iterator, Composite, Singleton, Factory Method, Observer, Builder, Template Method
3	Facade, Command, Memento, Mediator, Visitor
4	Decorator, State, Strategy, Chain of Responsibility, Proxy

The Unit 1 activity is a design problem. The material gets the students working on design and working in teams from the start. The work also introduces the notion that there is no right or wrong design necessarily. There are ten learning outcomes for the first unit. Samples of these outcomes are shown in Table III. Bloom's Taxonomy is not used to classify the unit learning outcomes though they do map onto the course's learning outcomes that are classified with Bloom's Taxonomy.

TABLE III
SAMPLE UNIT 1 LEARNING OUTCOMES

After completing this unit the student will be able to:
1. Describe the logistics for the running of this course
2. List the assessment mechanisms that will be used in this course along with their percentage contribution to the final course grade
3. Define the principles of coupling and cohesion
4. Explain why coupling and cohesion are antagonistic principles
5. List the names of some common design patterns

In the first class the instructor covers the organization of the course and discusses design principles that will be considered throughout the course. The design principles review some that were covered in their first software engineering course. After that short introduction, the students are given a design problem to solve using the design skills that have been developed through 4 quarters of computer science programming courses and 1 software engineering course. The students are encouraged to discuss the problem with one or two other students. At the beginning of the next class each student will individually submit a first cut at an object-oriented design for the problem. Students are assured that they will receive most credit for the assignment if they exhibit due diligence in completing it. A full and complete design is not sought. The second class is divided into three parts. First, groups of three or four students will work together to create a consensus design incorporating the best aspects of the individual designs. Next, some of these designs are presented to the entire class. In the last part of this second class, the instructor leads a discussion of ways in which groups of classes in these designs relate to each other and to the solution of the problem. Especially where there are commonalities, the instructor will point out where established patterns were used by one or more designs and motivate the advantage of discussing the design at this subsystem level rather than an individual class level.

The remainder of the term is divided into three units which each cover a set of patterns as shown in Table II. Each unit has the same structure composed of three components: 1) a detailed list of learning outcomes, 2) a set of questions related to the patterns being studied in the unit, and 3) a design and implementation exercise.

The learning outcomes for the unit have some that are common to all patterns and additional outcomes specific to the individual patterns in the unit. A selection of the learning outcomes for unit 2 is shown in Table IV. The unit outcomes cover the range of levels where this course is positioned in Bloom's Taxonomy. Some unit outcomes are pure knowledge memorization. The outcomes common to

all patterns fall into this category. Students have a clear idea of what they are expected to know for each section of the course and for each individual pattern covered.

The second and third components of each unit make up the unit team activity. A unit team is composed of 3 or 4 students. Unit teams change for each unit in the course. Creation of the unit teams is both student self-selected and instructor assigned. For the future, we are considering using learning style inventories to guide formation of the team. The unit team submits a solution for the unit questions and design activity. All students on the team receive a common grade for the assignment individually adjusted by a factor computed from peer evaluations.

TABLE IV
SELECTION OF UNIT 2 LEARNING OUTCOMES

At the end of the unit, for each pattern studied, you should be able to:

1. state the intention
2. state the motivation
3. draw the structure of the pattern
4. identify the participants and describe their responsibilities
5. specify the applicability of the pattern
6. suggest sample application areas including at least one application not discussed in the textbook

Adapter

1. explain the tradeoffs of class vs. object adapters
2. contrast the ease of overriding adaptee behavior with class and object adapters
3. explain the use of two-way adapters

Singleton

1. describe how the number of instances is controlled by a singleton
2. describe the client collaborations with the singleton object
3. explain why a singleton is equivalent to a global variable

The unit questions are written so that some are a straightforward discussion of textbook material. Other questions require students to expand on concepts presented in the textbook or relate ideas that are not directly compared by the authors. Answers to these questions require more thought but can still be obtained from the reference material. At this second-year level we provide a more structured problem set than may be the case for typical problem-based learning exercises. Each unit will have between 10 to 20 questions that the unit team must answer. A list of some typical unit questions is shown in Table V.

TABLE V
TYPICAL UNIT QUESTIONS

1. Iteration over a recursive composite structure can be tricky using an external iterator. What are the problems with this? How can you accomplish this?
2. The C++ implementation of class adaptation specifies the use of multiple inheritance. How would this be implemented in Java?
3. Let **methodA()** and **methodB()** both be methods declared in the current class or one of its super classes, and assume that **methodA()** calls **methodB()**. Is **methodA()** always, sometimes, or never an example of the Template Method pattern? Justify your answer in terms of the pattern as presented in the text.
4. Consider the interface **java.util.Collection**, which is implemented by the java collection classes. One of the methods defined in the interface is **iterator()**. Is **iterator()** a Factory Method? Why or why not?
5. One of the methods all classes inherit from **java.lang.Object** is **toString()**, which can be overridden to provide a suitable string representation for any object in a given class. Is **toString()** a Factory Method? Why or why not?
6. How can a Builder enforce semantic constraints, i.e. certain parts are valid only when within another part, certain parts must be installed before/after other parts? Suggest methods that a Builder can use to handle a violation of semantic constraints.

The unit design and implementation activities are created to highlight the current unit's patterns. The instructor emphasizes to the students that the design should make appropriate use of design patterns and clearly show its capability for expansion and coverage of all problem requirements. The implementation is intended to be a proof of the design concept and will have requirements that are a subset of the overall design requirements. For example, one activity was the design of a drawing editor. The team was required to have a design that could accommodate many drawing elements, different persistent storage file formats, and several interaction modes. The design was graded for its ability to cover this span of requirements. The implementation was required to handle a subset of the design requirements, namely, rectangles and lines with a small set of properties, one very simple file format and an interface with only menus and toolbars. One unit implementation is in Java and another is in C++.

The design and implementation activity for the last unit is different. All too often students can make it through an entire computing curriculum having only done "greenfield" assignments. Rarely will students have class assignments that do not start with a fresh sheet of paper. Students in RIT's computing programs typically gain experience with non-greenfield projects on their co-op assignments. In reviewing co-op evaluations, students will often comment that they were completely unprepared to work within an established code base. The Unit 4 design activity addresses this weakness. Each unit team is given the final code and documentation for a student project submitted in the introductory software engineering course. This is the team-based project course that most students in Engineering of Software Subsystems completed within the last term or two. These projects are typically 1 – 1.5 kSLOC in size. The project is from one or two years back so that few students had this as their project and the instructor absolutely ensures that for those students who had done this project their submission is not the one selected.

All teams work with the same code base. The first task is to reverse engineer the code and extract the design identifying any pattern usage that is found. Having been prepared by a student team, the documentation may be of limited value for this part of the activity. After gaining an understanding of the as-built design each team will then propose a refactoring of the code base following the design principles that are stressed in the course and applying their

newly gained knowledge of design patterns. The unit team is not required to implement the refactored design.

COURSE GRADING

Collaborative learning activities should provide the opportunity for mutual interdependence and individual assessment. The grading structure that this course uses does provide that since even though all work for the course is done in unit teams the course grading is based on both individual and team assessment. Table VI shows the course grade structure. Individual and team assessments are equally weighted in the computation of a student's final course grade. The individual unit quizzes help assure that teams will not solely use a "divide-conquer-copy-paste" approach to answering the unit questions. Each student is individually responsible for achieving the learning outcomes for each unit and must demonstrate this on the unit quiz. The unit quizzes have questions that primarily address the knowledge and comprehension levels of outcomes for the unit. Mid-term and final exams pose large scale design problems that each student must individually solve. Grading emphasis is on appropriate application of design patterns and adherence to design principles such as coupling and cohesion.

TABLE VI
COURSE GRADING STRUCTURE

Individual Components	Percent
Exams (10, 20)	30
Unit 1 design problem	5
Unit quizzes (3 * 5)	15
Team Components	
Unit questions (3 * 5)	15
Unit design/implementation exercises (2 * 10)	20
Refactoring exercise	15

CLASSROOM SESSIONS

The unit teams use the classroom sessions primarily to work on their unit activities. This is a guaranteed time that unit teams can use for collaborative interaction. Other scheduled class activities include: unit quizzes, mid-term exam, discussion of unit questions at the end of each unit, and team presentation of unit design and implementation solutions. The classroom activities in unit 1 were already discussed. Each pattern unit starts with a short overview of the patterns for that unit. Lectures are also scheduled on refactoring and anti-patterns. Beyond that all other lecturing is done "on-demand." Students request to hear discussion of a particular topic by placing a lecture request in a course electronic discussion group within 24 hours of the class. If no request is made the unit team has the entire 2-hour class session available for their use.

During the class session the instructor actively works with each unit team for short periods of time. During these interactions the unit team can ask questions of the instructor or request feedback on their design and unit question answers. The instructor also asks questions to guide the team in desired directions or to assess the understanding of the unit material by individual members of the team. Having provided the students with a detailed list of the learning outcomes for each unit the responsibility is placed on the students to achieve those outcomes. The necessary information can be found in the resource materials or the student can make a lecture request to use the instructor as an additional resource. The instructor's time in class is freed from lecturing on topics which are low on the cognitive scale. The responsibility for that has been shifted to the students. Lecturing is only done in response to an expressed demand. The discussion is more effective since it is presented to an interested audience. With lecturing greatly diminished (rarely more than 3 hours over a 12 hour unit) class time is more productively used in engaging discussions with the student teams. Questions from the students are motivated by their need to answer the unit questions or use newly learned knowledge of design patterns in a real design. The discussions are most often related to whether a particular approach in the team's design is an appropriate use of a pattern or if it is use of a pattern at all. The instructor's interactions guide the students to explore the fundamental intentions of the design patterns and to make comparisons of different approaches. This has the students actively considering the tradeoff decisions they must make in their designs. The students are beginning to develop their core engineering skill of tradeoff analysis. The development of software is moving from being simply a programming task to a higher-level engineering task.

DISCUSSION

There is both anecdotal and quantitative evidence showing the success of problem-based learning delivery of this course material. Conversations with students and a review of written comments on course evaluation forms show strong support for the format. The negative comments received mostly were complaints about the course workload. This has been addressed by the course syllabus presented in this paper. The first PBL version of the course had a multi-phase term-long project running concurrent with the unit activities. This was too much activity and the students had great difficulty keeping track of the various activities. The current syllabus eliminated the term project and merged some of those activities into the implementation activity in each unit. Few comments (< 10%) were negative about the problem-based learning delivery itself. More typical were comments such as the following:

> "PBL is a great way to teach a course but the work was a little much. I think students are more interested and learn more in PBL. All CS and SE courses should be PBL."

> "I thought the PBL was a very good idea because that is how I learn best. I felt like I learned much more in 362 than I did in 361."

> "The Problem-Based Learning methodology is quite effective and I believe it goes a long way towards assisting the learning and retention of concepts and skills. While the work was a little overbearing in regard to my other classes, a slight reduction in this would make the course ideal."

> "The area of the instructor's performance that I liked best was when he would sit down with a group of individuals and have a very helpful discussion about course topics."

Quantitatively there are also indications of the benefits of PBL. Student course evaluations indicate that the PBL course yielded statistically significant improvements at the 95% confidence level when compared to the studio-based lecture and class exercise format. To the question "Overall, how would you rate this course?" the PBL course received 4.19 vs. 3.94 on a quality scale of 0 to 5. To the question "What is your opinion of the principle textbook of this course?" PBL was 4.84 vs. 3.88. The textbook [1] used in this course is generally considered to be an excellent book. The improved opinions of the textbook are attributed to the students now actually reading the book. With PBL the responsibility for understanding the material is clearly shifted to the student with the textbook as the primary reference. A student cannot survive in a PBL course without reading the textbook. Other evaluation questions showed improvement trends though were not statistically significant.

An analysis of grades to determine if the PBL format has improved student performance is difficult because the grading components are very different in the two class formats taught by this instructor. The final exams were a similar format that can be compared. The average grade on the final for the PBL course showed an improvement trend (88.4 vs. 82.1). This result however was not statistically significant at the 95% confidence level ($p <= 0.06$).

CONCLUSIONS

The first course in RIT's software engineering program that is devoted to software design has evolved from a traditional delivery of lecture and lab to its current format. Through this evolution each version maintained the same learning outcomes. The students rate the current format using a problem-based learning methodology favorably compared to the delivery of other courses in a traditional lecture/lab format. The students report an increased appreciation for the material in the textbook even compared to an intermediate delivery format of lecture immediately followed by exercises in a studio lab. Anecdotally, the student unit teams are engaged with the course's material on design patterns and the interactions and discussions with the instructor are more often at levels further up in the Bloom Taxonomy.

The author has also modified an upper-division elective course to use PBL delivery with results similar to what are reported here. Based on successes with the two courses the department faculty are considering using similar pedagogy in other courses in the software engineering curriculum. All software engineering courses are now scheduled in our three studio classrooms with conversion of our courses to studio delivery or problem-based learning to follow over the upcoming years.

ACKNOWLEDGMENT

I would like to acknowledge Mike Lutz as the developer of the original version of the Engineering of Software Subsystems course. It was from that solid base that I was able to attempt these experiments in changing the delivery format. The students, who were unwitting participants in the evolution of the delivery methodology for this course, deserve my acknowledgment also. They had to suffer through my misjudgments that usually were on the side of too much work for them to do. Each term I noted their comments and adjusted how I next delivered the course.

REFERENCES

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

[2] M. J. Lutz, "Using Patterns to Teach Software Subsystem Design," ASEE/IEEE Frontiers In Education Conference, San Jaun, pp. 21 - 24, 1999.

[3] Distance Learning Resource Network, "Bloom's Taxonomy," U.S. Department of Education, 2000, http://www.dlrn.org/library/dl/guide4.html.

[4] Project Catalyst, "How to Engineer Engineering Education," Bucknell University, 2002, http://www.departments.bucknell.edu/projectcatalyst/summer2002workshop.htm.

[5] B. Hoyt, M. Hanyak, M. Vigeant, W. Synder, M. Aburdene, et al., "Project Catalyst: Promoting Systemic Change in Engineering Education," ASEE/IEEE Frontiers in Education, Reno, NV, pp. 8 - 12, 2001.

[6] K. M. Edens, "Preparing Problem Solvers for the 21st Century through Problem-Based Learning," *College Teaching*, vol. 48, pp. 55-68, 2000.

[7] H. S. Barrows and T. R. N, *Problem-based Learning: An Approach to Medical Education*. New York: Springer, 1980.

[8] J. J. Kellar, W. Hovey, M. Langerman, S. Howard, L. Simonson, et al., "A Problem Based Learning Approach for Freshman Engineering," ASEE/IEEE Frontiers In Education Conference, Kansas City, 2000.

[9] M. C. LaPlaca, W. C. Newstetter, and A. P. Yoganathan, "Problem-Based Learning in Biomedical Engineering Curricula," ASEE/IEEE Frontiers In Education Conference, Reno, pp. 16-21, 2001.