2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Teaching Software Architecture to Undergraduate Students: An Experience Report

Chandan R. Rupakheti and Stephen Chenoweth
Department of Computer Science and Software Engineering
Rose-Hulman Institute of Technology
Terre Haute, Indiana 47803
{rupakhet, chenowet}@rose-hulman.edu

***Abstract*—Software architecture lies at the heart of system thinking skills for software. Teaching software architecture requires contending with the problem of how to make the learning realistic – most systems which students can learn quickly are too simple for them to express architectural issues. We address here the ten years' history of teaching an undergraduate software architecture course, as a part of a bachelor's program in software engineering. Included are descriptions of what we perceive the realistic goals to be, of teaching software architecture at this level. We go on to analyze the successes and issues of various approaches we have taken over the years. We finish with recommendations for others who teach this same subject, either as a standalone undergraduate course or integrated into a software engineering course.**

***Index Terms*—Software Architecture; Project-Based Learning; Course Evolution**

I. Introduction

Software architecture remains a difficult subject to teach at any level, unless the instructors and the audience are practicing software designers. By being in charge of the high-level design for systems, the scope of the architect's work is technically different, and it is not even all-technical. Desired virtues for a software architect include the following [1] [2]:

1) The *artistic skill* to make seemingly simple designs that are easy to grasp and yet which solve complex problems.
2) *Analytical skills*, such as an ability to find root causes for high-level problems in existing designs, like why a system runs too slowly or is not secure.
3) A *systems engineer's understanding* of the business, social, and operational environment in which a system needs to operate. An understanding of the domain in which a system will live, as well as technical knowledge.
4) Ability to *relate to a wide range of stakeholders*, including the client, users, system administrators, and suppliers.
5) *Communication skills* - such as listening to and working out issues with those implementing a design.
6) A *working knowledge of the capabilities of the development organization*, and of how to describe to them what to do. This normally includes a high-level of adeptness at using the required technologies. The architect often writes a framework, which is developed further by others in the team.
7) *Knowledge of multiple technologies*, from which to choose the right ones for the next job.

The need to develop these traits resonates only with people who have a certain amount of direct experience already. The goals of a course in software architecture, therefore, should either recognize the limitations of the intended audience, or else work to instill pieces of the background in students during the course.

The biggest challenge is surely teaching software architecture to undergraduate students who, usually, have very limited experience in the software industry. One needs to be modest about how many of the above seven skills can be checked-off at the conclusion of a software architecture course.

An example of a hinge question in teaching software architecture is whether or not students should attempt an architectural project. Without this, expectations of learning through experience surely need to be reduced. With this, the running of a course could be much more complex, because study and development of actual large systems require extensive work by students and instructors alike, and results can be unpredictable.

A systematic problem in getting architecture across, to students lacking real-world design experience, is this – the high-level design principles are heuristics lacking the absoluteness of first principles. The answer to the question, "Should we use this solution idea here?" is inevitably "It depends …". For practitioners, this is a grim joke that they live with. For inexperienced students, it is a frustrating divergence from what they are used to valuing.

A second structural problem in teaching software architecture is that the background of the instructors should include a large chunk of diverse real-world experience. The equivalent issue would be having someone teach the architecture of buildings who had not actually designed various kinds of buildings. E.g., A seasoned software architect knows that fashionable theoretical approaches may or may not work in a given practical situation. We are empiricists like David Hume, not rationalists like Rene Descartes.

At the undergraduate level, the overall design of any software architecture course should consider how ambitious the course developer's goals will be. Unless they start their own company, undergraduate students are unlikely to walk into a first job where they are the lead designer. Thus, the problem of "what to teach" here is analogous to that problem for a

[1] http://bredemeyer.com - All URLs verified on October 20, 2014.

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.177

course in project management. If an alumnus will not have to do that work right away, how much appreciation, conceptual understanding, or actual practice should be taught, to prepare them for an eventual likelihood?

Over ten years at Rose-Hulman, we have developed an undergraduate software architecture course to meet the needs of fourth year software engineering majors who are just starting their customer-based senior project. It is the final course in a sequence of seven courses specifically about software engineering practices and processes. The primary goal of the current course is to prepare students for high-level design situations in industry, such as the following:

1) Employing new, pervasive technologies and processes in their projects, and
2) Correcting architectural issues in existing systems.

An additional goal of the course is to prepare students for the initial design work in their senior project. The paper makes following two key contributions:

1) It discusses several components of a software architecture course that has a rich history of ten years.
2) It describes the evolution of the course, comparing different variants we have tried, and gathering useful conclusions for others working on software architecture education.

The rest of the paper is organized as follows: We present related work in Section II. The evolution of the course from 2004/05 to 2013/14 is discussed in Section III. Improvements made to the course are quantitatively assessed in Section IV. Section V discusses the lesson learnt. Threats to validity of our result are discussed is Section VI and we conclude the paper with observations, in Section VII.

II. Related Work

Awareness of the need to teach system-level skills realistically: At the end of CSEE&T 2004, a *Key Considerations in Teaching Software Architecture* workshop was given by Jim Tomayko, Linda Northop, and others [17]. This workshop served the same purpose then as the current paper, to recap experiences in teaching this subject, for the benefit of others doing the same. The major lesson of the 2004 workshop was that most of software design education focused on teaching how to create the functional attributes of a system, and this emphasis had to be different for teaching architecture. Different concepts and processes needed to be stressed, to deliver the "quality attributes" (QAs) expected in large systems. For example, functional interactions with users can be described with use cases or user stories. This is not so for more pervasive attributes of a system.

Karam et al. described an implementation of the quality attribute approach to an undergraduate course in software architecture and design [11]. This course included examples comparing the advantages of different architectural styles, in the sense of Garlan and Shaw [8], regarding QAs delivered based on high-level design. The course provided exercises and projects in doing such design work. It also had practitioners visit to explain the designs of systems they had built.

In 2005, Shaw and Vliet in [15] discussed how to make software architecture education more realistic, as well as how to teach the required non-technical competencies (reference the list of skills in Section I). Good examples of well-documented architectures were felt to be critical to this teaching. The specific issue was highlighted, of making examples large enough to demonstrate the complexity involved in architectural decisions. A library of such examples was recommended, and open source projects were suggested. To get across the value of "soft skills," an example of a case study approach was given [5].

Adding architecture-level technologies: Alrifai [1] described teaching software architecture so as to include lessons on new technologies combined with understanding QAs, in a software engineering course. In his case, the new technology was web services. His approach was to provide learning via problem solving in lectures, team projects, and individual assignments. UML was used to highlight the behavior induced by design patterns. Realistically, in the team projects, the students designing parts of a large system had to explain their technical decisions to non-software "management" members of their team, and accept the managers' decisions about high-level requirements.

Refining quality attribute descriptions: The use of scenarios in a class, to document and design for quality attributes, was described by Tempero [16]. These scenarios are a close equivalent, for QAs, of use cases for functional requirements. Both express requirements in an operational form. The scenarios were introduced by Bass, Clements and Kazman in [2]. As with use cases, the scenarios are expected to "live" throughout development, guiding design, implementation, and testing. Tempero taught a software architecture course as part of an undergraduate specialization in software engineering. It was accompanied by a separate project course, as well as courses in networks and operating systems. He chose to teach a part of the course on "middleware" first, for the following reason: *"... combining middleware with software architecture helped with a problem I had encountered when first teaching this course. Students in their third year of university generally have had little experience with large software systems and so have difficulty appreciating discussions on the architecture of systems. They have also had little experience with quality attributes such as reliability and availability, both of which are important to middleware discussions. By following on from the middleware part, I have some nice examples on which to base discussion on software architecture."*

The author noted this was not the first time middleware had been used as a vehicle for discussing higher-level architectures [14]. He commented that teaching both topics in a single course cut down on the amount of time he could spend getting students to wrestle with architectural decisions. This was a part of Tempero's general caution that it is not possible to give students at this level a complete picture of what is needed to architect systems successfully. He further found that it was hard to include a realistic problem which was fully detailed for architecting. This left students with having to "make

up" information in order to define needed quality attributes, reducing the perceived value of a systematic analysis. His conclusion was that, for students struggling to understand the principles involved, the complexity of architectural decisions was baffling.

Improving problem-solving skills: Garg and Varma [7] took a reverse strategy, to the caution that architectural problem solving is messier than undergraduates are accustomed to. They pointed out that *"... inculcating problem solving skills should be one of the learning objectives of SE academic and training programs. But structured problem solving is usually latent or missing in most of the current curriculum."* They recommended a problem-based software architecture course as the remedy for this impedance mismatch. Their course included case studies, a *cognitive apprenticeship*, the combination of analytical and synthetic approaches, and use of Polya's Problem Solving Model [13]. A part of the latter includes reflecting on the effectiveness of a problem-solving process used.

Finding large-enough, interesting systems: The use of open source projects to teach software architecture was described by Costa-Soria and Perez [4]. Employing an existing, large system gets around the problem of having a system for students to work on, where architectural decisions make a difference. The authors had students reverse-engineer open source projects to analyze their architectures and other sides to their software engineering. The reverse engineering activities required students to conceptualize the architectural intent. It also gave them insights into the way large systems are maintained, something not found in most other undergraduate software courses. Students made real changes to the open source systems, to validate their ideas about the design.

Wang utilized a game development project as a vehicle for teaching software architecture [18]. His student teams established both functional and quality requirements; designed, evaluated, implemented, and tested the software architecture; then did post-mortem analyses. His results show that *there are both positive and negative effects of teaching software architecture in the context of a game development project. Students found it motivating to learn about software architecture through game development, but some students found it hard to apply the theory when developing the game. Most students were very positive to learn about new game technology as a part of the course and it was very stimulating to create an actual product.* The students' complaints sound familiar: They ran out of time, disliked documentation, and found some of the architecture process worthless. Regarding the latter, they were asked to evaluate how well the architecture delivered required quality attributes, then to base their architecture on these results. The teams evaluated one another's architectures. The author concluded that they lacked sufficient experience to play the role of evaluator and mistrusted their own results.

This study was an example of a somewhat negative outcome from teaching software architecture to undergraduates. It would be helpful in furthering this type of education, if more such outcomes were known, in addition to the positive ones.

Giraldo et al. [9] had architecture students not only build a distributed system, but also develop it across schools using distributed tools, a realistic process goal for a large project. Six universities participated. The project was to design the software architecture for a secure communication system. In order to promote strong teamwork, the instructors used principles going back to Vygotsky [6] – positive interdependence, close interaction to achieve goals, recognition of individual contributions, and teamwork monitoring. The researchers considered experimental variables such as distance between team members in assessing the success of the class project. Their model for student collaboration was tested, using a hypothesis that the activities and team results would be sufficiently close to the software industry. The study was done primarily to test the distributed learning environment, but the authors felt the project outcomes were sufficiently favorable, despite the complexities of the architecture task for undergraduates.

Classroom strategies: In 2012, Christensen and Corry abandoned lecturing about software architecture, replacing 3 hours of weekly lecture with "active seminars" featuring storytelling [3]. While this was for a "master level course" in architecture, many of the problems the authors overcame are like those of undergraduate courses. They needed to pick a case study with sufficient complexity that software architecture design tools would not be seen as wasted on a toy problem. They used a popular MMORPG (massively multiplayer online role-playing game) where precise and responsive control must be exerted over a player's avatar, in "a world that needs to be rendered in high resolution at high frame rates to provide a meaningful experience." Students were presented with architectural problems in this world, which they then worked on, during class, in teams, with full-class review of team solutions at the end.

One sample architectural problem for an active seminar was this: "Fluent user experience of controlling his/her avatar in the virtual world – 1600x1200 full color rendering at minimum 30 frames per second."

The authors used exam scores to rate learning. They found that the mean scores for the class with active learning were somewhat higher than in four previous years.

III. Evolution of the Course

The Software Architecture course at Rose-Hulman has a rich history. From being a junior-level course, CSSE 374 and then CSSE 377, to a senior-level course, CSSE 477, it has gone through many changes and improvements. In this section, we discuss how the course evolved over the last ten years. A general overview of the changes is presented in Section III-A, which is followed by a discussion on the evolution of the instruments used in the course (Section III-B). A discussion on coverage of soft skills required for a software architect is presented in Section III-C. Finally, coverage of architectural patterns and quality attributes is discussed in Sections III-D and III-E, respectively.

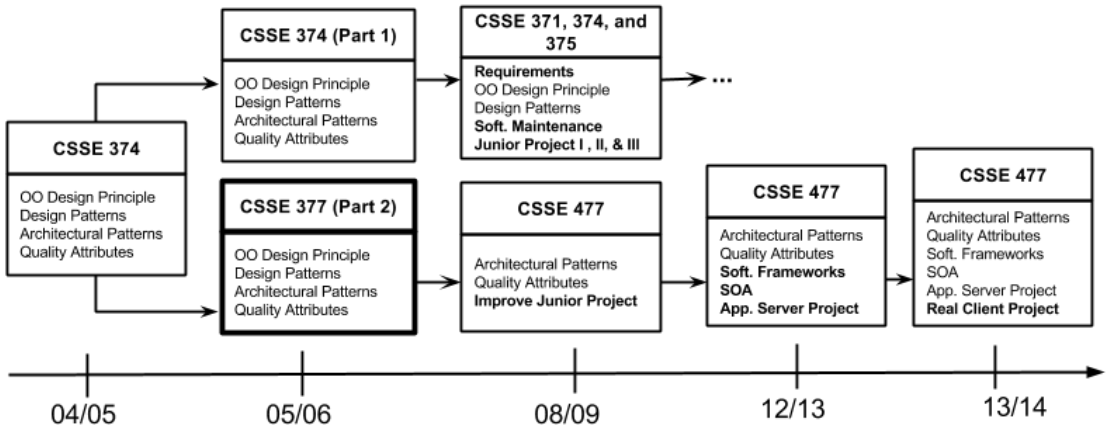

Fig. 1. Evolution of the course from acadmic year 2004/2005 through 2013/20014

A. An Overview of the Change History

The course began in 2004 as a combined software architecture and design course (CSSE 374) for undergraduates taking Rose-Hulman's newly created software engineering major. There were short design projects in this course, and efforts were made to include a larger team project which would reveal issues with quality attributes. Figure 1 shows the evolution of the course from its inception.

It was clear that both mid-level design and architecture were too much for a single, one-quarter course. The next year, the course was increased to two semesters, with a larger team project. This is when a separate software architecture course was born (CSSE 377, highlighted in Figure 1). Both courses included architecture as about half of their content, so that students had more time for separate topics such as performance, usability, security, and comparison of architectural styles.

In 2008, the two courses were divided differently, with the first becoming an object-oriented design course (CSSE 374 - Software Design) using Craig Larman's text [12], and the second being strictly a software architecture course (CSSE 477 - Software Architecture), using Bass, et al's book [2].

At this same time, the software engineering major instituted a "junior project" spread over three terms. The fall term focused on requirements engineering (CSSE 371), the winter term focused on object-oriented design principles and patterns (CSSE 374), and the spring term focused on software evolution and maintenance (CSSE 375). Immediately succeeding these courses was the new software architecture course (CSSE 477) to be taken at the start of the senior year (fall term). Thus, the students entering the architecture class already had been building, for a year, a system larger than what they would have encountered in most undergraduate classes. We used these systems as the basis for multiple projects in the architecture class.

Typical of these projects, each a week or two in duration, were the following (in short):

1) Make the user interface run twice as fast.
2) Make a significant reliability enhancement and prove that it improved system availability.
3) Do a user study in the usability lab, and create a conceptual model of the way the users understand your system.
4) Redesign a significant part of your system so that testing time is cut in half, for that part.

These architectural exercises were designed to be like what the students most likely would encounter in the first few years of their careers, something short of designing an entire system.

The systems students used were typically amenable to such architectural activities. They had been developed incrementally using agile methods, resulting in several thousand lines of code in each system. There were many points where design decisions had been made, which could now be a source of reflection. Most of these systems involved client-server or service-oriented architecture connections; thus, performance, reliability, and error-handling issues were prime targets to explore.

From 2009-11, the architecture course was stabilized around having such separate projects in quality attribute areas, plus exams and a short research paper in areas of the students' choice.

The projects, built on these systems that students already knew, were "semi-real-world." They had had real clients the year before, when the same students had worked on them. However, the assignments in the architecture class generally were done after those clients had lost interest, or taken a version of the system at the end of the previous year.

In 2012-13, we started realizing that, as the junior course sequence (CSSE 371, CSSE 374, and CSSE 375) achieved stability, the use of junior project to practice architectural improvements had become redundant. First, the junior projects were already well-refactored in CSSE 375. Second, only a few of the architectural improvement tactics were applicable to each project, thus, limiting opportunity to practice meaningful architectural enhancements.

TABLE I
EVOLUTION OF COURSE TOOLS

Dimension	Inception (04/05 - 07/08)	Stable (08/09 - 11/12)	Experimental (12/13-13/14)
Daily Quizzes	Not done.	Daily quizzes were used to support learning from lectures. These were completed by students during class.	Daily quizzes were used to pose analytical questions on architectural design, principles, and practices. Sometimes they required in-class group work.
Homework	Primarily based on required readings.	Primarily based on required readings.	Mostly centered around designing and implementing software using various patterns covered as well as doing tutorials on new software frameworks.
Paper Reviews	Students selected research papers and reported on these in class.	Students selected research topics and wrote approximately 5 pages (per student) on teams about a subject of interest.	The papers mostly focused on design and implementation of large-scale frameworks.
Project Work	Students had a single project in which they developed a framework for a system and a way to test that.	Students did six projects of 1-2 weeks each, essentially one on each of six quality attributes discussed shortly in Section III-E.	Students did a project with a real client as well as a side project (disguised as homework) to implement a Java Application Server that addressed the six quality attributes.
Tests	Biweekly quizzes with short answer questions, lasting approximately 30 minutes.	Biweekly quizzes with short answer questions, lasting approximately 30 minutes.	No test administered. Students were graded based on their performance on daily quizzes, homework, paper reviews, and project work.

Hence, we focused our attention to develop hands-on exercises on top of modern-day software frameworks and web services rather than trying to force superficial improvements on the junior projects. Furthermore, we wrote a partially implemented web server and asked students to utilize architectural improvement tactics on the server. This change allowed them to practice applying most of the quality attribute tactics they had learnt in the classroom, particularly, for improving availability, interoperability, modifiability, performance, security, and testability.

Architectural principles and patterns are about large-scale systems. After introducing changes in 2012-13, we wanted to experiment with a project in 2013-14 that was larger than the web server project and was in production as a mainstream server of a software company. We reached out to an industry partner who graciously agreed to help us with this experiment by letting our students implement components over their production server. Two of the students projects were so successful that they got rolled-in as features of the production server.

In this way, we have been striving to move away from a lecture-centric model of teaching to more hands-on, project/practice-based learning in the course. We hope to keep pushing in this direction in future offerings as well.

B. Evolution of Course Tools

As the course went through changes, the assessment tools used in the course were altered correspondingly. In order to better describe the changes made to the course, we have divided the entire history of the course into three phases: Inception (04/05 - 07/08), Stable (08/09 - 10/11), and Experimental (11/12-13/14). As the names suggest, the course was fairly new during the inception phase and there were lots of changes being made to the course. As the course progressed, it became more stable. We have recently started to make significant changes to the course to make it more modern. We are tweaking and testing various aspect of the course, hence, the name "Experimental" is given to the last phase.

The course since its inception had five key assessment tools: daily quiz, homework, paper review, project work, and test. A daily quiz is a set of questions with short answers that students use to focus on the key concepts of the ongoing lecture. It is a way for an instructor to emphasize such key points in the lecture and conduct active learning within the regular lecturing session. The other assessment tools are self-explanatory and their description is elided for brevity. Table I illustrates the key changes to the assessment tools during the three phases of the course history.

C. Coverage of Soft Skills

As listed in Section I, there are non-technical skills that an architect should exhibit such as leadership, communication, and interpersonal skills. An architect must also be aware of current market trends, product value, business rules, and other aspects of economics. Knowledge of software development processes, product lines, and other software engineering processes is a crucial aspect of their job function. They must also know how to document and communicate their ideas to both development teams and clients. We have covered many aspects of an architect's roles and responsibilities in this course. Table II details variations going from inception to the experimental phase of the course.

D. Coverage of Architectural Patterns

Architectural patterns are at the heart of the architecture course. Table III illustrates how the coverage of the patterns evolved through the different phases of the course. At the inception, most of these were covered through lecture slides, daily quizzes, and paper reviews. As the course evolved, the experimental phase of the course had more hands-on labs (extended as homework) to implement these patterns or apply them to some interesting problems.

E. Coverage of Quality Attributes (QA)

QA's, or non-functional requirements, dictate the architecture of a software system. The course covers quality attributes in detail, as shown in Table IV. The specification of the

TABLE II
COVERAGE OF SOFT SKILLS

Dimension	Inception (04/05 - 07/08)	Stable (08/09 - 11/12)	Experimental (12/13-13/14)
Qualities of an Architect	Taught along with design roles.	Separate material describing this. Major topic of the social role of the architect.	Minimum coverage. Technical qualities of an architect were the key focus.
Economics of Architecture	Not strongly emphasized.	Strong emphasis on constraints and on the effects of architectural decisions.	Reduced emphasis. Limited to a lecture and a daily quiz.
Process	Homework questions from the chapter on Product Lines in Bass's book [2]	Students envisioned another product related to the one they used in labs, and how a platform could be extracted to support both.	Agile architecture and software product lines were covered through lecture and daily quizzes. Microsoft Office Suite was used as a case study.
Documentation	Students created an architecture document in the style recommended in Bass's book. This was done top-down to a significant depth.	Students did a high-level design document only. They had done UML in a previous software design class.	Focus was on Agile-style documentation using UML to document high-level design as necessary in the homework/project works.

TABLE III
COVERAGE OF ARCHITECTURAL PATTERNS

Dimension	Inception (04/05 - 07/08)	Stable (08/09 - 11/12)	Experimental (12/13-13/14)
Layered	Typically a part of the team project developing a framework.	Used as a frequent class example.	Covered through lecture slides and daily quizzes.
Model-View-Controller	Used as a frequent class example.	Used as a frequent class example.	Covered through lecture slides, daily quizzes, and homework. Students solved a problem using Java Swing.
Plugin Architecture	Typically a part of the team project developing a framework.	Not done.	Covered through lecture slides, daily quizzes, homework, and project work. Students developed a pluggable framework from scratch and developed few extension plugins for the framework. Students also developed a plugin for the Eclipse IDE.
Pipe-and-Filter	Some student projects included or used this.	Some student projects included or used this.	Covered through lecture slides and daily quizzes.
Client-Server	Many student projects included or used this.	Many student projects included or used this.	Covered through lecture slides, daily quizzes, homework, and project work. Students implemented a Java application server.
Peer-to-Peer	Some student projects included or used this.	Some student projects included or used this.	Covered through lecture slides and daily quizzes.
Shared-Data	Not done.	Not done.	Covered through lecture slides and daily quizzes. The required database course enforces this pattern.
Broker	Some student projects included or used this.	Some student projects included or used this.	Covered through lecture slides, daily quizzes, homework. Student solved a computation problem using Java Remote Method Invocation (RMI).
Publish-Subcribe	Typically a part of the team project developing a framework.	Some student projects included this.	Covered through lecture slides, daily quizzes, homework. Student solved a problem using Rabit MQ.
Service-Oriented Architecture	Not covered	Introductory team project.	Covered through lecture slides, daily quizzes, homework, and project work. Student solved several problems using SOAP and RESTful APIs.
Multi-tier Pattern	Not covered	Not covered	Covered through lecture slides, daily quizzes, homework, and paper review.
Map-Reduce	Not covered	Not covered	Covered through lecture slides, daily quizzes, and paper review. The elective big data course specializes students in the Map-Reduce pattern using Hadoop ecosystem.

attributes are done using *scenarios* [2]. In the stable phase, the course project centered around understanding and applying different hardware and software tactics to improve the quality attributes. With the introduction of the Java Application Server project in the experimental phase, students found ample opportunities to apply software specific QA improvement tactics.

IV. ASSESSMENT OF THE COURSE CHANGES

Rose-Hulman's Office of Institutional Research, Planning and Assessment (IRPA [2]) ran two studies, looking for variations in outcomes from the different approaches we took to teaching the architecture course, one during the stable phase and the other during the experimental phase of the course:

1) They compared student achievement in the classes, as measured by the instruments we used in the classes - daily quizzes, projects, homework, and tests. A section of the course in the experimental phase was compared to a section of the course in the stable phase in this study.
2) They compared the students' self-assessment of their learning in the class from responses students provided on the course evaluations.

[2]http://www.rose-hulman.edu/irpa/

TABLE IV
COVERAGE OF QUALITY ATTRIBUTES IN THE COURSE

Dimension	Inception (04/05 - 07/08)	Stable (08/09 - 11/12)	Experimental (12/13-13/14)
Availability	Software issues were described, along with review of hardware methods to improve availability.	Students did a project on improving availability. E.g., what to do if user device became detached from the server. Most projects were about improving error handling.	Both hardware and software methods were discussed to improve availability. Homework/Project required application of the techniques covered in lecture.
Interoperability	Not covered.	Not covered.	Homework on Java's Remote Method Invocation (RMI). Homework on XML and JSON based web services using Glass Fish server (Java EE).
Modifiability	Homework questions from the chapter on this in Bass's Software Architecture in Practice, 2nd Ed.	Students did a lab to reduce the time to make a category of change. It was an experiment in which they made changes in both "before" and "after" versions of the system to measure improvements.	Lecture on coupling, cohesion, and dependency graph analysis. Pluggable frameworks were covered using Eclipse and Firefox as case study. Homework/Project work required developing pluggable framework as well as plugin as extension.
Performance	Homework questions from the chapter on this in Bass's Software Architecture in Practice, 2nd Ed.	Student project was to speed-up an existing system they knew. Typical results included improving user interfaces as well as database changes.	Lecture on both hardware and software methods for performance improvement. Homework/project problems required them to apply the software methods.
Security	Homework questions from the chapter on this in Bass's Software Architecture in Practice, 2nd Ed.	Goal was to find a security issue in a system students had built, and fix this. Typical examples - insecure connections and unencrypted files.	General coverage as there was already a security course offered on-campus. Mostly software methods covered with discussion on detecting, resisting, reacting, and recovering from attack for an application server. The project and homework required them to implement security measures.
Testability	Homework questions from the chapter on this in Bass's book	Project involved making some aspect of a system systematically more testable. Most create a test harness.	Minimum coverage, as there was already a Software Quality Assurance course offered. The project/homework required them to apply the concepts learnt in that course.
Usability	Homework questions from the chapter on this in Bass's Software Architecture in Practice, 2nd Ed.	A usability study was done in which students derived a conceptual model of the users' approach to interacting with their system.	Tactics on improving usability was discussed. A homework problem required them to evaluate usability of an existing software system in a controlled setting.

We instructors had had prior industry experience in designing systems, and we believed that we had used close to "industry standards" in deciding how well students were able to perform their assigned class activities. Although the assignments varied, as already described, these were made to be as close as possible to things we believed they would encounter in their first few years on the job. For example, the exercise in trying to improve performance of a system with a few thousand lines of code was much like a vocational task they would need to do, in our opinion. If the students were able to analyze the issues, discover root causes, and fix these, that had some face validity for doing similar tasks after graduation.

Even with our best efforts, there were some differences in the grading scales in assignments that needed to be accounted for. Therefore, before sections could be compared, a number of steps were taken. First, within each course, student averages were calculated for each assignment category. Then these averages were averaged to create a course objective score by student for each of the course objectives. Since the grade scales varied between the courses, (i.e. quizzes in one section worth 100 points and only worth 10 in another), the objective scores varied as well. In order to compare these, they first had to be standardized.

Z scores (standard scores) were computed for each section independently. This process adjusts the mean of student scores to 0 in both cases and each score is represented as the standard deviation from the mean [10]. Once the Z scores had been calculated, the scores for both sections were transformed so that student average for each course objective was on the same scale regardless of instructor. These transformed scores were used in an independent t-test analysis to compare student learning in CSSE 477 across courses.

The results of this study were a little surprising at first: The assessors found, *"There were no statistically significant differences in student scores on course objectives across sections. Therefore, the changes made in [the experimental phase of the course] did not impact evidence of student learning differently than previous iterations [the stable phase] of the course."* This result could be explained by the fact that the teaching methods used in the two offerings were doing an almost equivalent job in satisfying the course objectives in the respective phase of the course.

Not seeing a significant difference in students' learning based on the assessment of the course outcomes and students' grades, we wanted to understand if there was anything that we gained by evolving the course to its current format. We asked IRPA to conduct another study to understand student's perception of learning based on their self-assessment on the course evaluations. In particular, four ratings (Likert scale) about the course were analyzed from academic year 2005/06 through 2013/14 excluding 2009/10 when it was not offered:

1) The students' overall quality of learning in the course
2) The laboratory assignments and course materials reinforced one another

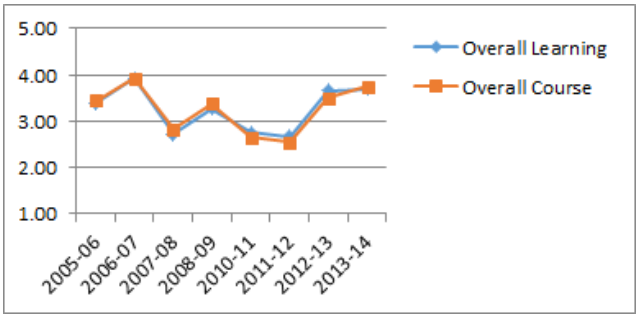

Fig. 2. Students rate the course and learning in the course similarly

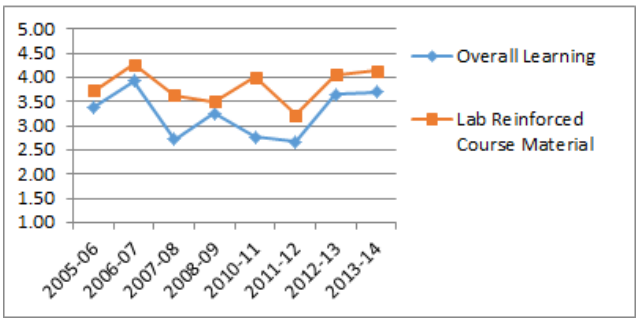

Fig. 3. Strong correlation between overall learning and better labs

3) The work load for the course in relation to other courses of equal credit
4) The overall rating of the course

This study produced some exciting results. First, there was a strong correlation (0.9789, $p < 0.0005$) between the overall perceived learning in the course and the overall rating of the course as depicted in Figure 2. Hence, it would be safe to drop overall course rating and just use overall learning to analyze other aspects of the course. Second, there was also strong correlation (0.717, $p < 0.025$) between students' learning and course materials reinforced by lab as depicted in Figure 3. While this evidence is encouraging, we should not hastily conclude that the labs helped improve the learning in the course. We were not able to run a regression test due to the limited data set to make such a claim. Nevertheless, the data looks encouraging as it indicate that the labs may have contributed to the improved learning of the students. Finally, Figure 4 presents averages of course evaluation ratings for all of the course evolution phases. Based on this figure, it would be safe to conclude that the changes introduced in the experimental phase have labs that reinforced the course materials, improved the course rating, improved student learning, while still making them work harder through increased course load.

We also checked correlations between i) class size and the overall learning and ii) load in the course and overall learning. We found both to be insignificant for this course.

The results of the study of a student self-assessment also showed a confounding outcome. The similar course designs from one year to the next yielded unexplained variability (inception vs. stable phases in Figure 4), as much as the years of

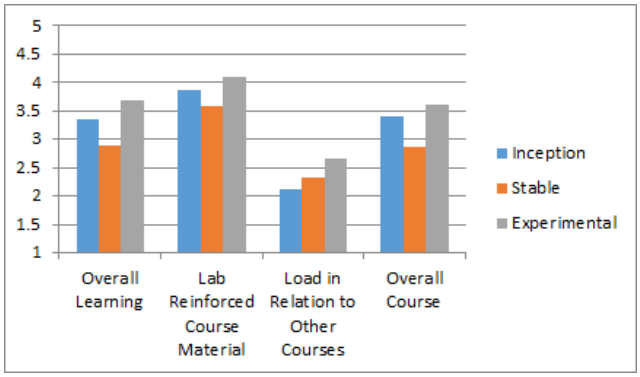

Fig. 4. Trend of the course

significant course change appeared to cause variability (stable vs. experimental phase in the figure). During some of the years when students reported, on average, the lowest perceived learning, all the project and homework was done on multi-term projects they had built for real-world clients. Combined with the outcome, in the first study, that students perceived more value from the controlled homework assignments, when they had both kinds, we tentatively conclude that our undergraduate students find greater value in the crisper, smaller projects they are used to tackling in school.

One current student summarized that, *"I thought the homeworks were much more valuable for the time spent on them. I learned a bit from doing the project, but it wasn't worth the effort."*

We would like to conclude this section by presenting three selected remarks made by students in the course evaluation of the recent versions of the course (the experimental phase):

Anonymous student 1: *"I'm typically not a fan of the SE courses, but this was rather good in terms of content. I find SE courses to be very high on lecture with less concentration on actual coding, which made this course a nice change of pace. Undoubtedly, this course has been the most work, but I find it to be worth it. I found myself bringing up the topics that I had learned in-class in interviews with companies, which was very useful for me. I also found myself applying some of the practices in my other courses which required programming."*

Anonymous student 2: *"The exposure to all of the web services really helps and I think it is very useful in software today. It is an all-encompassing course which ties together everything we have learned in the past 4 years."*

Anonymous student 3: *"1 - Learned a lot of important software architectures. 2 - Designed, implemented, and tested the concepts and architectures that we studied which reflected positively on my understanding of the topics. 3 - Working with a real client and a real code base gave me a big lesson on how things are different in industry than in class. 4 - How to build on top of existing systems. 5 - How to build systems using the right architectures."*

V. Lesson Learnt

Systems abound with components that are too slow, insecure, unstable, unusable, untestable, or unmaintainable. These systems are built and kept going by people with bachelor's degrees in computer science or software engineering. The graduates are not trained, in most such college programs, to build systems delivering high-level quality attributes.

At the inception of our undergraduate software engineering program, we wanted to help fix this pervasive issue of the software industry. We believed that our program needed a pragmatic course in software architecture. We then evolved the course to address areas of greatest need, always testing the limits of how much realism we could include.

We emphasized projects where architectural tactics played a strong role in success. Especially, we have focused on QA's accomplished via server-side programming. We searched to find projects expressing that side of design, in sufficient complexity that useful and memorable learning was achieved.

We adjusted the course also so that it taught large scale, modern-day tools and technologies. These are a moving target. At one time "frameworks" were a desirable goal in themselves. Then service-oriented architectures became a goal. We now teach both these topics in lower-level courses. Hence, being up-to-date in terms of both tools and techniques has become more critical than ever for the instructor of the software architecture course.

VI. Threats to Validity

We have found that undergraduate students prefer canned homework and tutorials; they are nervous about open-ended problems. The software architecture instructor has to juggle what students should be taught versus what they can appreciate at the time.

In all the sections of this course, throughout the years, students have been asked to do assignments having as much real-world value as we could put into the course. This meant that they were working on different assignments every year as their major projects. Thus, we could not do precise comparisons of either their success or their self-assessment of that success – the bases were always a bit different.

Software students very much react, in rating their own learning, based on whether their project activities were successful. Thus, for example, students who were asked to improve on a particular quality attribute may have done a very deep analysis of their work, and why it didn't provide the system changes they had hoped for. We would rate them highly for this, but they concluded they didn't learn much, because of the failure to achieve what they intended.

VII. Conclusion

We believe that the nature of software engineering in practice may be sufficiently messier, than the normal school work of undergraduate students, that these students systematically have trouble dealing with the obdurate realities. We found this problem to be true in teaching software architecture. Very likely, this high-level design work is about as fuzzy as it comes: Architecture could be a worst-case scenario. However, a solution also can be envisioned over the whole curriculum. A greater amount of imprecision could be put into learning software generally – by making the problems larger, by allowing multiple correct answers, and by providing voices more like stakeholders in deciding the success of school projects.

It could help students grasp the heuristics of software architecture if this were not the first time they had to contend with such cloudy software issues.

We strongly favor a project-based version of teaching architecture, even at the undergraduate level. Though the students may not understand or appreciate the nebulous nature of attempting high-level design work, a guided introduction to it, in which they have to struggle, is much better than no introduction at all. They are very likely to be put into a position of doing the architecture, or of modifying it, during their careers. Most often, this exposure will be without further formal training in architecture per se.

Studies in the literature show that combining architecture with the basics of software design, in a single course, gives insufficient time for either. This has also been our experience. We further believe the combining of all of software engineering into a single course, or even into a full year, gives even less of the needed time to cover this topic. Architecture requires reflection because design work significantly uses the "other" half of the brain, where ideas come to the designer during non-focused intervals. The learning of architecture needs to provide a framework for such absorption; the budding architect should learn how not to be rushed into design decisions.

If one's curriculum demands that architecture be one of many topics in a software engineering course, we recommend that student projects be based on large systems they already are familiar with, whose architectural problems stand out to illustrate design issues. If students do not come immediately from the same prerequisite courses, where such a system could be developed or worked on ahead of the architecture part of the class, this need for an example system probably means that architecture is done last, rather than toward the beginning of the software engineering course. That way, the system being built during earlier parts of the course could become the basis for architectural lessons learnt.

References

[1] R. Alrifai, "A project approach for teaching software architecture and web services in a software engineering course," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 237–240.

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., 2003.

[3] H. B. Christensen and A. Corry, "Lectures abandoned: Active learning by active seminars," in *ITiCSE*, 2012, pp. 16–21.

[4] C. Costa-Soria and J. Pérez, "Teaching software architectures and aspect-oriented software development using open-source projects," *SIGCSE Bull.*, vol. 41, no. 3.

[5] T. H. Davenport, "The case of the soft software proposal," *Harvard Business Review*, May 1989.

[6] R. M. Felder and R. Brent, "Cooperative learning," in *in Technical Courses: Procedures, Pitfalls, and Payoffs, ERIC Document Reproduction Service, ED 377038*, 1994.

[7] K. Garg and V. Varma, "An effective learning environment for teaching problem solving in software architecture," in *ISEC*, 2009, pp. 139–140.

[8] D. Garlan and M. Shaw, "An introduction to software architecture," in *Advances in Software Engineering and Knowledge Engineering*, 1993, pp. 1–39.

[9] F. Giraldo, S. Ochoa, M. Herrera, C. Clunie, A. Neyem, S. Zapata, J. Arciniegas, and F. Lizano, "Applying a distributed cscl activity for teaching software architecture," in *i-Society*, 2011, pp. 208–214.

[10] F. J. Gravetter and L. B. Wallnau, *Statistics for the behavioral sciences*, 9th ed., 2013.

[11] O. Karam, K. Qian, and J. Diaz-Herrera, "A model for swe course "software architecture and design"," in *FIE 2004*, 2004, pp. F2C–4–8 Vol. 2.

[12] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 2nd ed., 2004.

[13] G. Polya, *How to solve it, a new aspect of mathematical method*, 2nd ed. Princeton, 1973.

[14] W. Royce, B. Boehm, and C. Druffel, "Employing unas technology for software architecture education at the university of southern california," in *WADAS*, 1994, pp. 113–121.

[15] M. Shaw and H. van Vliet, "Software architecture education session report," in *WICSA*, 2005, pp. 185–190.

[16] E. Tempero, "Experiences in teaching quality attribute scenarios," in *ACE*, 2009, pp. 181–188.

[17] J. Tomayko, L. Northrop, S. Chenoweth, M. Sebern, and D. Suri, "Key considerations in teaching software architecture," in *CSEE&T*, 2004, pp. 174–174.

[18] A. Wang, "Post-mortem analysis of student game projects in a software architecture course," in *ICE-GIC*, 2009, pp. 78–91.