A Relook at the Introduction to Software Engineering Course

The Reality of the Course

The reality of undergraduate computing education is that the vast majority of students do not go through software engineering curricula where there is time to address the breadth of the software engineering body of knowledge. The Software Engineering 2014 Curriculum Guidelines [1] lists nine top-level knowledge areas for software engineering programs with a minimum of 467 "lecture" hours of material. For the majority of students who are in computer science, computer engineering, or other computing programs, they will receive their software engineering education in a single course, Introduction to Software Engineering, which is expected to cover software engineering as a topic. Some of the topics in the software engineering Knowledge Areas may be covered by other courses in the non-software engineering student's program, but if you look at for example, the Computer Science 2013 Curriculum Guidelines [2], the task of covering software engineering is still daunting. These guidelines define 18 Knowledge Areas three of which, Software Development Fundamentals (SDF), Software Engineering (SE), and Social Issues and Professional Practice (SP), contain knowledge that falls into the software engineering realm. Guideline comments identify the SE and SP knowledge areas as specific curricula areas where teamwork and communication soft skills will be learned and practiced. The Software Engineering Knowledge Area, which at 14 pages is the longest non-cross-cutting Knowledge Area in Computer Science 2013, identifies 60 Core topics with 69 Learning Outcomes, and 54 Elective topics with 56 Learning Outcomes. It will be a difficult syllabus design task to cover that material at any level of detail in a single course.

The breadth and depth of the software engineering Knowledge Areas leads to a lament that is often heard at software engineering education conference sessions. The faculty members responsible for software engineering in the curriculum ask “How am I going to fit the core SE topics and the ‘soft’ teamwork and communication skills in the single software engineering course in our computer science curriculum?” The Introduction to Software Engineering course often addresses this by being taught as a broad overview of topics in software engineering. The course uses one of the classic software engineering encyclopedic textbooks to cover this broad range of topics. There is a term-long team-based project. In many computing curricula, this is the first, and often only, coursework where students tackle a large team project. In the project work, students demonstrate their transference of knowledge of software engineering principles and practices gained from lecture material and reading the textbook to actual project work.

Our Experiences with the Course

Of all the courses in Rochester Institute of Technology software engineering curriculum, our SWEN-261 Introduction to Software Engineering course is the one course that we never feel we have done correctly. The students take the course in their second year after a first-year computer science sequence. This is perhaps a bit earlier in our students' program than at other institutions. The course has a broader constituency than just software engineering students because it is also required in the computer science, computer engineering, and computational math programs. In the course's greater than twenty year history, we have reworked it at least eight times with most of those reworks being extensive redevelopment of most of the course.

We have had versions that used the Personal Software Process (did we get pushback from students on that) to a full-on agile methodology [3] (it seemed like the students were not learning enough); were heavy on process with less than a 1,000 lines of code in the project (students nicknamed the course Document Engineering) to minimal process, heavy design, and >10,000 line project implementations (the last three weeks of the term were just a hack fest); used no textbook and our own lecture notes (that was a lot of preparation work) to a classic software engineering tome as the textbook (which only half the students bought and few of them read); had teams create full use case requirements, specification, design, and test plan documents (Document Engineering again) to preparing user stories, design, and simple acceptance test specifications (this seems like about the right mix); in different versions, we had the project implemented in C++, Java, and Python and as a desktop application and a web application; it has been delivered blended [4] and completely face-to-face. In one version of the course, we choose one of the classic texts with the idea that in the Introduction to Software Engineering course we would do only surface coverage. The software engineering students would keep it through their entire undergraduate experience, and we would be able to use it for reference material in most subsequent courses in our curriculum. That idea turned out to be a non-starter for several reasons. The students, especially the non-software engineering students, did not buy into it, or buy the text, and our faculty realized pretty quickly that "if everyone used the same textbook" was not an approach that would work across our curriculum.

The faculty in our program struggled with this course because we never felt like we had gotten it right, and we were tired of redeveloping the course every two to three years. The students also did not think that we got the course right. In addition the issues mentioned above that students expressed with the course, there were several perennial themes in end-of-term course evaluation comments. These included:

- There is no connection between the lectures and the project work.
- The exams are too much about buzzwords and memorization.
- There was a steep learning curve for the framework tools used in the project work. In various versions of the course this included C++ Xforms, Java Swing, and Django.

An additional problem that we found with the last version of the course was that students were not building their object-oriented design skills in the course. This course used a web-based project written in Python on top of the Django framework. The low level of design skills was noted by faculty teaching the following design course, SWEN-262 Engineering of Software Subsystems, which emphasizes design abstraction and design patterns. Students were talking about design information learned in their first-year CS course and not the Introduction to Software Engineering course. The department's Industrial Advisory Board also identified this as a problem after having discussions during our annual meeting with students at all year levels.

Learning Goals for the Course

In considering yet another redesign of our Introduction to Software Engineering course, we gave some reflection on why we had so much trouble getting this course "right". We were continually balancing multiple requirements for the course including it needing to be an introduction to the breadth of software engineering, and a significant team project experience for the students. In reviewing the course's history, we decided that the reason this course was changed so frequently

is that with each redesign we always started with the same basic premises for the course, namely, it needed to provide a broad overview of the software engineering discipline, and it would use one of the classic software engineering textbooks that covers all of those areas. For this redevelopment, we dropped both of those requirements.

What are the key software engineering learning outcomes that the course should deliver? In many respects, the design of this course is more important for the other computing programs that require it than for the software engineering program because this is often the only exposure to software engineering principles that the non-software engineering students get. It will be OK if some topics found in classic introduction courses and considered essential for a full understanding of software engineering are left out. Those that we had covered previously were not being covered at a level that imparted that full understanding to begin with, and the software engineering students would see the full breadth and depth of those topics later in their program.

With our redevelopment guidelines in mind, we set the primary goals for the course to be:

- Instill good entry-level software engineering practices in the Computer Science and Computer Engineering students. Of course, the Software Engineering students would benefit from this also. This is particularly important at our institution with all these programs having required co-op that students start the summer after taking this course.
- Reinforce and expand on basic object-oriented design skills introduced in the first-year CS sequence. Particularly important for the software engineering students as preparation for their design courses.
- Design the course around what we have said countless times in recruitment and open house activities, namely, that software engineering is about engineering design of software, software product development, teamwork, and communication.
- Have students follow contemporary software development practices and use up-to-date development tools on their term-project.

Guidelines for the Course Redesign

The Introduction to Software Engineering course is a large service course for our department. As already mentioned, it is required by several computing programs. Students typically take the course in their second year after a first-year CS sequence that includes object-oriented programming. The programs have this course as a prerequisite for going on co-op. That last point meant that the course topics should impart knowledge and skills that could be immediately applied during interviews and on co-op. Over 450 students enroll in this course each year. It is taught on both twice a week and three times a week semester schedules each term with more than 10 sections each term. This is a large number of adjuncts teaching sections. We designed course content in 25 minute "chunks" which provided 6 chunks of course content per week for all weekly schedules. Different activities occur in a course chunk including lecture and individual or team exercises. All but a few of the lectures stay within one chunk of class time.

We took an engineering approach to the design of the course specifying a requirement and how to address it. We set a requirement for the topics to be roughly distributed as 35% design, 35% process, 15% teamwork, and 15% communications. Previous versions of the course concentrated on design and process in varying degrees. In all those versions, we put the students on teams and

asked them to tackle communications tasks, but we never explicitly spent course time discussing either of those areas except for a single one hour Teaming lecture.

The selection of course topics involved a lot of compromise and judicious choice/elimination of topics. In selecting course topics and planning the schedule, we wanted to maintain our software engineering program's pedagogical philosophy which is to emphasize class exercises, a collaborative environment, and faculty interaction with teams. The topic coverage was to concentrate on the most important topics with coverage and practice in depth rather than breadth. We applied a Test-First consideration when selecting topics: For every topic, we asked what we will, or currently did, hold the students responsible for, and if it did not rise above Bloom Taxonomy Level 1 - Remembering [5] we considered dropping the topic. This addressed the student perception that the exams were buzzword and memorization exercises.

The design topics were selected to elevate the student's object-oriented design skills to a level beyond the design of single classes. The process topics were selected to describe the Scrum practices that the teams would do and how those practices supported the overall productivity of the team. Students would either individually or on the term-project team experience all of the design and process concepts that were discussed in class. Emphasizing practice in depth helped to reinforce the connection between the classroom lectures and the project work that the students were doing.

We settled on the following 31 topics to include in the course. Each topic is in what we consider its primary area though several topics will overlap between two or more areas. Within a topic area, the list is in chronologic order through the term. Fuller details about the course topics can be found on the course's public website [6].

Design Topics	**Process Topics**
Domain analysis Review OO concepts Object-oriented design I Appreciation for software architecture Web architecture and development Domain-driven design State-based behavior I Unit testing Object-oriented design II Sequence diagrams State-based behavior II Code metrics Appreciation for usability	Appreciation of software development process Introduction to OpenUP phase X Defining project requirements Sprint planning Acceptance testing Version control concepts Backlog refinement and estimation Code coverage Code review
Communications Topics	**Teamwork Topics**
Effective team communications Design and code communication Sprint demos Design documentation Project presentations	Team Formation Personality types Sprint retrospective Professional responsibility

The final count of topics does not exactly adhere to our percentage goals, but generally does achieve the desired balance. This selection of course topics required the elimination of several topics which are traditionally seen in Introduction to Software Engineering courses and we had covered in previous versions of our course. Eliminated topics included: discussion of multiple software process methodologies, discussion of multiple software architectures, details of requirements engineering with use case specification of requirements, functional specifications, risk management, and discussion of design patterns. In some cases, we pulled material from courses later in the software engineering curriculum and placed it in this course because we felt it was important for all students taking this course to be exposed to it and not just the software engineering majors. This was particularly done in the next design course which had material added to it because the students were not coming in with adequate object-oriented design skills from the previous version of this Introduction to Software Engineering course.

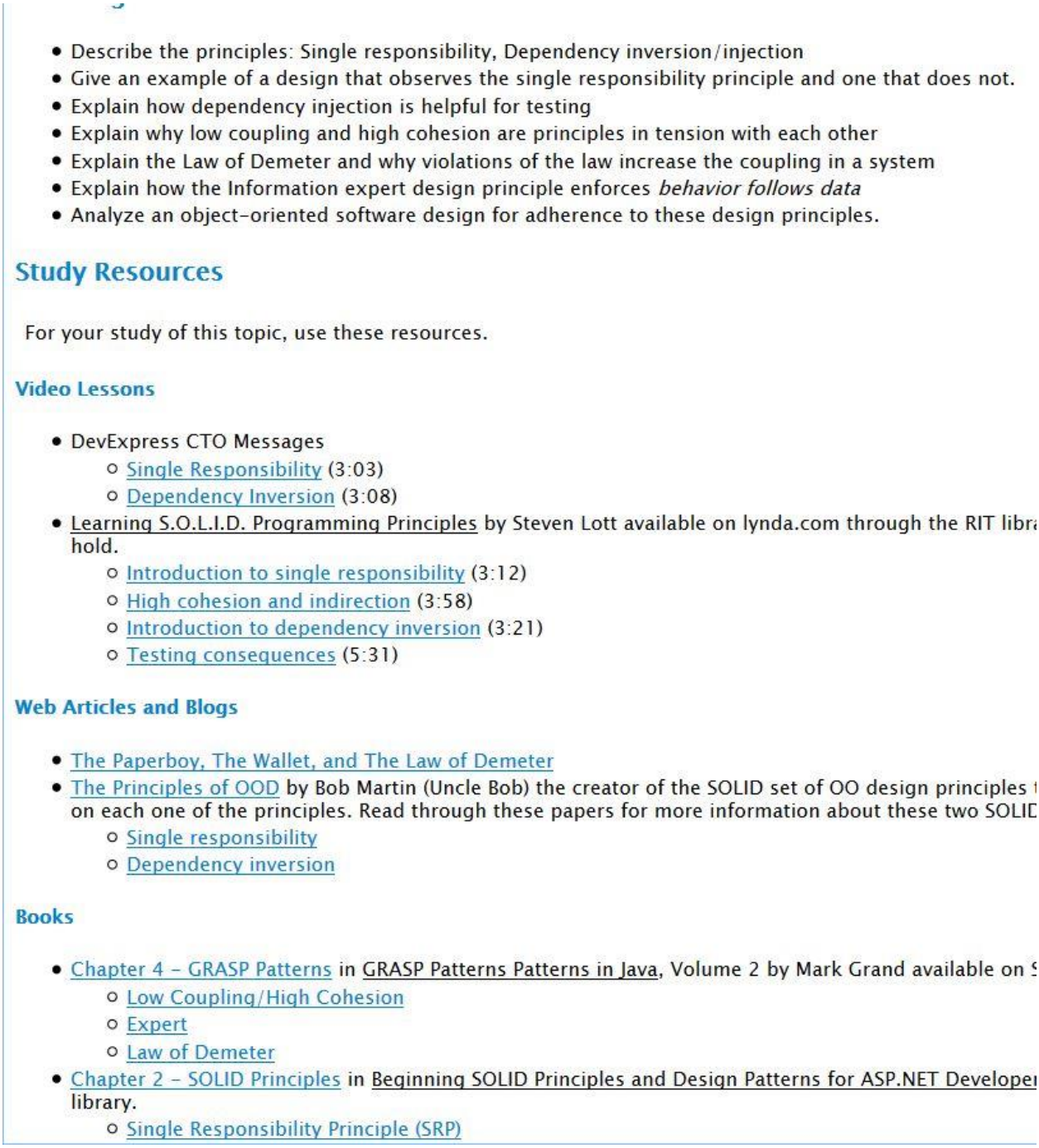
The topics that are listed as *Appreciation of X* are designed to not cover those areas to any level of detail as was done in previous versions of this course. The learning outcome was to have the student gain an appreciation for what the area was and why it was important. For example, a learning outcome for *Appreciation of Software Development Process* is "Identify the benefits of a formal software development process." We want the student to appreciate the need for following a defined software development process. We do not cover any methodologies other than OpenUP [7] which the students use as a strategic process to define the team's focus through major project phases. The project teams use Scrum practices (user stories, sprint planning, backlog refinement, daily standups, sprint retrospectives) as a tactical process. For software architecture our aim is similar wanting the students to understand the benefits of having a defined software architecture. We do not discuss architectures and remove all architectural design by imposing a three-tier (User Interface, Application, Model) architecture on the project.

We did not believe that there was a textbook that aligns with this set of topics which led us to abandon the use of a classic software engineering encyclopedic textbook in favor of resources that were web-based or we authored ourselves. Fortunately, all current students have access to a full-suite of lynda.com [8] videos and Skillsoft [9] books. We make extensive use of short videos and specific relevant sections of books from these sources along with YouTube videos and tapping into a wealth of websites.

Each course topic is described on a topic page which contains the following sections:

- Introduction – general description of what the lesson is covering
- Learning Outcomes – a lesson typically has 4 to 6 learning outcomes
- Study Resources – the number and type of resources varies from lesson to lesson. The types of resources include: Videos, Web Articles and Blogs, Books, Wikipedia (since this is the first place that the students go we felt compelled to include Wikipedia references) Each resource provides a link to the online resource.
- Class Lecture – lectures are typically 10+/- slides. A few lectures extend to take two class chunks. Each topic page has a link to a PDF of the lecture.
- Exercises – the exercises are both individual and team exercises that are done before, during, or after class. The timing and nature of the exercises varies from lesson to lesson. Individual before-class exercises are often a viewing/reading assignment with completion of a short (5 minute) on-line quiz by the start of the class when the lecture will be given.

A screen capture of a typical topic page showing the learning outcomes and resources is shown as Figure 1.



- Describe the principles: Single responsibility, Dependency inversion/injection
- Give an example of a design that observes the single responsibility principle and one that does not.
- Explain how dependency injection is helpful for testing
- Explain why low coupling and high cohesion are principles in tension with each other
- Explain the Law of Demeter and why violations of the law increase the coupling in a system
- Explain how the Information expert design principle enforces *behavior follows data*
- Analyze an object-oriented software design for adherence to these design principles.

Study Resources

For your study of this topic, use these resources.

Video Lessons

- DevExpress CTO Messages
  - Single Responsibility (3:03)
  - Dependency Inversion (3:08)
- Learning S.O.L.I.D. Programming Principles by Steven Lott available on lynda.com through the RIT libra hold.
  - Introduction to single responsibility (3:12)
  - High cohesion and indirection (3:58)
  - Introduction to dependency inversion (3:21)
  - Testing consequences (5:31)

Web Articles and Blogs

- The Paperboy, The Wallet, and The Law of Demeter
- The Principles of OOD by Bob Martin (Uncle Bob) the creator of the SOLID set of OO design principles t on each one of the principles. Read through these papers for more information about these two SOLID
  - Single responsibility
  - Dependency inversion

Books

- Chapter 4 – GRASP Patterns in GRASP Patterns Patterns in Java, Volume 2 by Mark Grand available on S
  - Low Coupling/High Cohesion
  - Expert
  - Law of Demeter
- Chapter 2 – SOLID Principles in Beginning SOLID Principles and Design Patterns for ASP.NET Developer library.
  - Single Responsibility Principle (SRP)

Figure 1 - Screen capture of part of typical topic page

Elements of the Course and Grade Breakdown

The overall course grading is composed of 57% individual and 43% team-based activities. The individual activities include two in-class 50 minute exams, a two-hour final exam, and a collection of individual exercises. The team-based activities are all associated with a term-long team project worked by a team of 4 or 5 students. The project work is delivered over 5 sprints along with a number of shorter team exercises that get submitted. The contribution of these elements to the final grade is shown in Table 1 below.

Table 1 - Contribution of Course Components to Final Grade

Course Component	Percentage of Final Grade
Two in-class exams	10% each
Final exam	25%
Term project	43% (40% over 5 sprints, 3% for team exercises)
Individual Exercises and online discussions	12%

The three exams are a combination of short answer questions and longer case study questions which result in the generation of one or more project artifacts. For all three exams, we allow the students to bring one 8.5" x 11" information sheet with whatever information is desired. The short answer questions are designed to go beyond memorization and recall. The following is an example of a question asking about the sprint planning process.

1. **(5 points)**
   What is the purpose of the sprint planning meeting? How does a team select the user stories to develop in the next sprint? Why might a team skip a story for a sprint?

The longer case study questions are based on a description of a small software system. A question may require the student to create user stories, a class structure diagram, a sequence diagram for a system feature, or a statechart for a web application interface or to define the behavior of a class. The case study questions that require a class decomposition have a complexity in the range of 10 classes.

The team and individual exercises vary in complexity and difficulty. Some are as simple as doing a screen shot to show that an online tool, such as a Trello planning board, has been setup. Others require team discussion or individual programming activity. Most of the Before-Class individual exercises are completion of a small quiz assessing a minimal knowledge of the topic based on reading and viewing a subset of the resources provided on the topic page. Most exercises are worth one point and are graded more on engagement than having a completely correct answer. Some exercises, particularly the ones involving programming activity, are graded more rigorously and are worth multiple points. An example of two individual exercises is shown in Figure 2.

There are a total of 39 individual exercises and 17 team exercises. The individual exercises get graded on a stepwise scale based on the percentage of points obtained: >80% - 12 points; > 60% - 9 points; >40% - 4 points; <= 40% - 0 points. The team exercise grade is computed as a straight percentage of the 3 points that they contribute to the final grade.

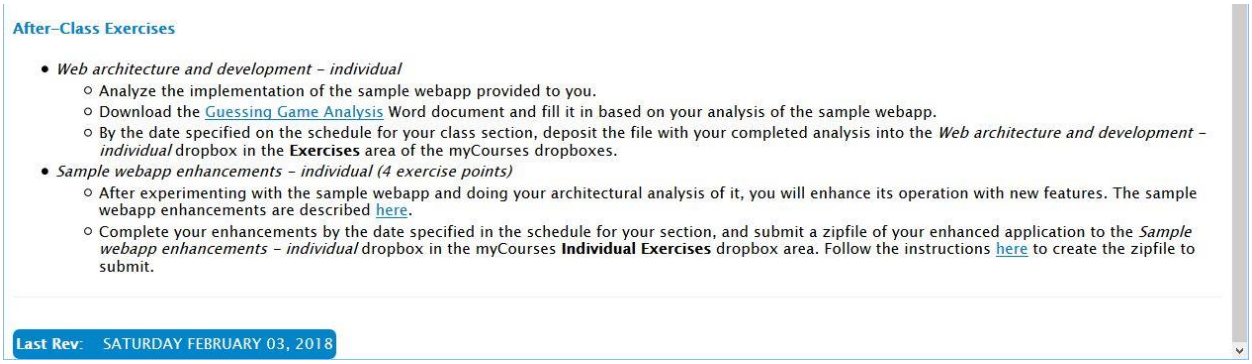


Figure 2 – Example individual exercises

To help ensure that the students are aware of the benefits of OpenUP as a strategic process, at the beginning of each phase, we review what the focus had been in the last phase and how the course topics and team activities supported that focus. With OpenUP the four phases and their focus are: Inception: reducing requirements risk; Elaboration – verifying proposed software architecture and reducing architectural risks; Construction – building out the product; Transition – final deployment and team shutdown.

Term Project Guidelines

There were several guidelines that were set for the term-project that students would work on through the course. These were set by the faculty when the redevelopment of this course was discussed during faculty and curriculum meetings. These guidelines included:

- The team-based project should run through the entire term with teams of 4 or 5 students.
- The project should use be a web-based application using Java.
- Any web framework that is used should be lightweight and not dictate the design thus requiring all students to do notable object-oriented design.
- The project does not have to include all of the elements of current web applications since the focus of this course is not web application development.

Considering these guidelines, we decided on the following requirements for the project:

- The project will be implemented on the Sparkjava [10] web micro-framework.
- Client-side work will be only minimal HTML or possibly some CSS, i.e. there will be no UI design, anything beyond that which the project needs will be provided to the team.
- The project's implementation focus will be on backend Java.
- The project will not require persistent storage, i.e. there would be no database work.
- Following Scrum practices, there will be no defined roles on the team.
- The team will use contemporary tools: GitHub/git for artifact control, Trello for planning, Slack for intra-team communications.

Teams deliver their project work in five sprints tied to the four OpenUP project phases. There are two sprints during the Construction phase. Each sprint is approximately three-weeks long. Rubrics are used to perform the sprint grading with the exact dimensions varying from sprint to sprint. For example, the first sprint in the Construction phase (Sprint 2) has a documentation deliverable but no functionality deliverable. The second sprint in that phase delivers the final functionality, a demo, but no documentation. Sprint 4 in the Transition phase delivers the final documentation. An example of three of the nine dimensions in the rubric grading for Construction - Sprint 3 is shown in Table 2.

Table 2 – Example Rubric Dimensions for Sprint Grading

Functionality 50%	Minimum viable product (MVP) feature set is bugfree. At least two enhancement features are bugfree.	MVP feature set is bugfree. At least one enhancement feature is bugfree.	MVP feature set is bugfree.	MVP feature set has notable bugs.	Little functionality seen in the product.
Adherence to Architecture and Design Principles 10%	High adherence to architectural separation and OO design principles.	Good adherence to architectural separation and OO design principles with only a few issues found.	Adherence to architectural separation and OO design principles is apparent but consistency is lacking.	There are multiple major issues with adherence to architectural separation and OO design principles.	There is little evidence of adherence to architectural separation or OO design principles.
Unit Tests and Code Coverage 5%	Full set of high quality unit tests with good mechanics providing over 90% coverage.	Full set of unit tests with a few issues with mechanics providing over 80% coverage.	Unit tests omit tests in several areas, have notable issues with mechanics, or provide less than 70% coverage.	Significant unit tests are missing, very poor mechanics, or provide less than 60% coverage.	Minimal set of unit tests with minimal use of JUnit, Mockito, and other mechanics, or providing less than 50% coverage.

In each sprint, the team is held accountable for sprint planning, appropriate use of version control including feature branching, and intra-team communication. The instructor uses the digital audit trail available in the tools used for these activities—Github/git, Trello, and Slack, respectively. In addition to peer evaluations that are done mid-project and end-of-project, the instructor uses this evidence to make individual positive or negative adjustments to team grades based on an individual student's contribution. The mid-project peer evaluations are mostly considered formative and do not affect the student's grade immediately. The instructor provides feedback to each student individually at that point. After the end-of-project peer evaluation, the instructor will compute the individual adjustment factor for each student, if needed, and apply it to the entire project grade for the student. Our experience is that after the first peer evaluation, poor performing students will step up their game with the end-of-project peer evaluation showing a more even distribution of the contributions.

Term Project Resources

There are a large number of technologies that teams will use during the project. Most students have experience with Java and source control from prerequisite coursework. To help students with the others, we provide a Resources page [11] which lists web resources for tutorials and other information to help the students learn the technologies. Through the project the students will be required to work with these technologies:

- Java 8 – implementation language
- Sparkjava web micro-framework
- FreeMarker – HTML template engine
- Maven – build tool
- Junit – unit testing
- Mockito – mock objects for unit testing
- Jacoco – code coverage
- MetricsReloaded – code metrics
- Trello – planning
- Slack – team communications
- GitHub/Git – artifact control
- Development IDE
- pandoc/MiKTeX – markdown to PDF

At the start of the Elaboration phase, we introduce the students to the "sample webapp". This is an example web application, a simple number guessing game, which we regularly use as a reference point. The implementation shows how to work within the Sparkjava web framework with good coding style and architectural separation. All of the basic interactions including some features of the FreeMarker template engine are demonstrated for the students. Only Ajax exchanges are not demonstrated.

One of the requirements for the project was that there was a shallow learning curve for working with the chosen web framework. This was not the experience that we had when we had a project that was Python/Django-based. Our student club had to offer many tutoring sessions to help students get up to speed in the Django environment. In this version of the course using Sparkjava and our sample webapp, in 15 to 20 minutes during the *Web architecture and development* class session we take a class of 20 from downloading a zipfile, to building the sample webapp, running it locally and guessing numbers in a browser, making a modification to server-side Java code, and seeing the result in a new running webapp. This is certainly a shallow learning curve for the web application technology.

We also use the sample webapp to get the students familiar with the Sparkjava and FreeMarker technologies. This is done by having each student implement a number of enhancements to the sample webapp. All the students are required to implement two enhancements and then can opt to get extra exercise points by implementing up to two additional enhancements. The four enhancements are described via user stories and acceptance criteria written in the format that we require the students to use on their term-project. An example of one of the required enhancements that each student must implement is:

> ***User Story***
>
> As a player I want to see the percentage of games I have won in my current session.

Acceptance Criteria

1. **Given** that the player is connecting to the web application for a new session **when** the Home page displays **then** the message "No game stats yet" will be shown.
2. **Given** that the player has not won any games **when** the player loses a game **then** the message "You have not won a game, yet. But I *feel* your luck changing." will be shown on the Home page
3. **Given** that the player has not won any games **when** the player wins a game **or** **Given** that the player has already won a game **when** the player finishes a game **then** the message "You have won an average of X% of this session's N games" will be shown on the Home page where X is the percentage of wins rounded to the nearest whole percentage and N is the total number of games for this player.

When the course covers unit testing with mock objects, the students get a copy of the sample webapp as an exemplar of a full set of high-quality unit tests. Each term, we have added to the sample webapp to move it toward being an end-to-end example of all the artifacts that the teams will be required to generate including user story requirements with acceptance criteria and solution tasks, an acceptance test plan, and design documentation. We are not there yet.

WebCheckers

Given the requirements that we set for the project, we decided to resurrect in a web-based version of an older Checkers desktop application that we had used in the 2005 timeframe. The application is called WebCheckers and is presented to the students with three required top-level features and several possible enhancements. A team is required to implement two of the enhancements to receive the full functionality credit. The minimum viable product (MVP) implementation must be bugfree before enhancement functionality is considered (see the first rubric dimension in Table 2 above). The full extent of our description of the requirements for this project is:

> The Product Owner desires a minimal viable product (MVP) which includes these features:
>
> 1. Every player must sign-in before playing a game, and be able to sign-out when finished playing.
> 2. Two players must be able to play a game of checkers based upon the American rules.
> 3. Either player of a game may choose to resign, at any point, which ends the game.
>
> The enhancements of interest to the Product Owner, in no order of preference are:
>
> 1. *AI Player*: Players may play a game against an artificial intelligence player.
> 2. *Asynchronous Play*: Players can play asynchronously.
> 3. *Multiple Games*: A player may play more than one game at a time.
> 4. *Player Help*: Extend the Game View to support the ability to request help.
> 5. *Replay Mode:* Games can be stored and then replayed at a later date.
> 6. *Spectator Mode:* Other players may view an on-going game that they are not playing.
> 7. *Tournament Play*: Players can enter into checkers tournaments including player statistics.

From this project description the teams develop user stories for MVP and for two enhancements to the product during Inception – Sprint 0. To get all the teams initially headed in the same direction, we define the two user stories and two technology spike stories that all the teams will complete for their initial implementation sprint in Elaboration – Sprint 1. These user stories are:

- User stories
    - Player Sign-in: As a Player I want to sign-in so that I can play a game of checkers.
    - Start a Game: As a Player I want to start a game so that I can play checkers with an opponent.
- Spike stories
    - Web Architecture: As a developer I want to learn the Web technologies Spark and FreeMarker so that I can develop the product.
    - Domain-driven Design: As a developer I want to learn Domain-driven Design so that I can develop the product.

Each team receives a GitHub repository with the essentials of a build configuration file, a home page that simply welcomes the user to WebCheckers, and all of the client-side template files and JavaScript code (~1900 LOC) for display of and interaction on a checkers board. The gameplay interaction includes identification of active player and moving that player's pieces onto empty squares. Validation of the moves is done through an Ajax exchange with the server. There is a description of the data structure that the UI-tier code must create for the view of the checkerboard to display correctly, and a statechart description of the Ajax exchanges that the server-side must process to effect gameplay. Teams can do client-side JavaScript coding only with instructor approval and it is rarely given because there is no need for it. The focus of the project is object-oriented server-side design and implementation, and that is where the instructors keep the teams focused.

For WebCheckers Elaboration – Sprint 1, which is validating and reducing architectural risks, the two user stories require the team to work the entire web application stack and understand the data structure required for display of a checkerboard. The only element that they will need but do not get to implement is the server-side Ajax exchange. The two spike stories are done as part of individual exercises adding feature enhancements to the sample webapp and other individual exercises associated with course lessons. The teams then continue in the Construction phase to implement the rest of the WebCheckers functionality. Our experience through two terms has been that a large majority of teams get to the point of implementing all of MVP and at least one enhancement. The teams that do not achieve that level of completion usually suffer from the standard teaming issues that are seen on team-based student projects.

Our Approach for Developing the Course

One unique approach that we took while developing the course material itself was to treat it like a software development project and follow practices similar to what the students were required to do on their term-project work. This gave us valuable insight into the problems that the students would encounter while doing their project work. Each course topic was created as a "project task" or user story as the students would within a Trello planning board. We created a Product Backlog of all the course topics. As development progressed, the topic cards moved through Design, Design Review, Development, Content Review, Ready to Merge, and Done. This

mirrored how the student teams would plan and track the development of user stories in their project. A screen shot of our Trello planning board during course development is shown in Figure 3.

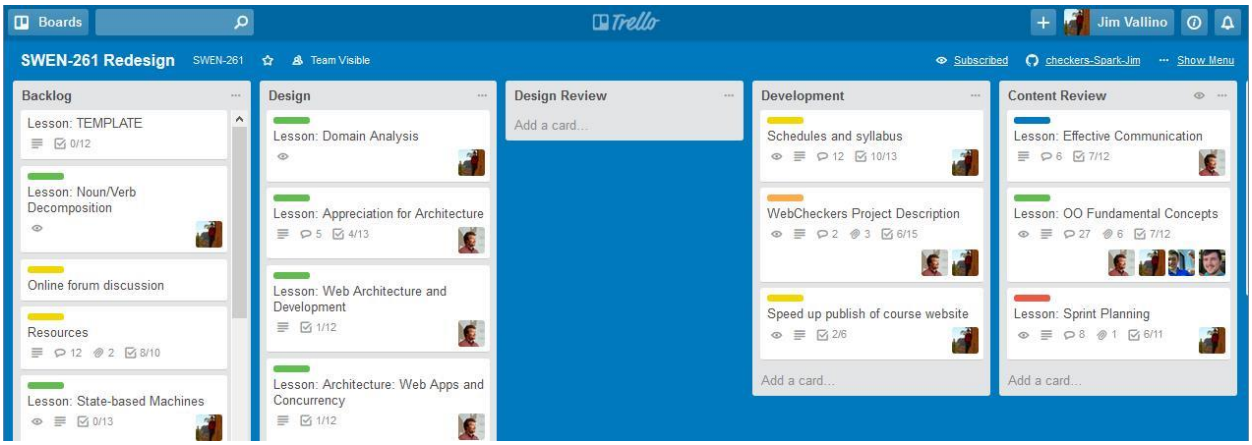


Figure 3 – Trello planning board for course development

In our department, we maintain common git repositories for course materials for almost all undergraduate courses. This is an invaluable resource for instructors of the course particularly new instructors and adjuncts teaching the course. While developing this course, we followed the same approach that we would ask the students to do in their project work. We developed each course topic (user story) as a separate feature branch in the course repository. When the story had gone through the full development, we needed to merge the new material into the public website along side of the material for the prior version of the course which was running concurrently in 10 other course sections in the term that this version was piloted. This gave us the experience of doing independent development and then the problems that the students would see when merging independent work into the main development path. **Error! Reference source not found.** is a picture of our git course repository at one point during the course development.

Finally, to maintain communication throughout the development and with continuing fixes we were in daily contact using a Slack workspace, exactly the same approach that we would ask the students to use. This also provided a great insight into the mechanics of using this technology.

Results of the Redevelopment

We have analyzed data from the pilot term in spring 2017 (10 original sections and 2 pilot sections) and for the first term rolled out to all sections of Introduction to Software Engineering in fall 2017 (13 new sections). Because of the large shift in course content and the level of coverage, there were few assignments or exam artifacts that we thought would provide insight for comparing student achievement between the original version and the new version. We had to rely on student perceptions as measured in course evaluation data. Since a primary goal for undertaking this redevelopment was to eliminate the perennial student complaints seen in course evaluation comments and heard in discussions with academic advisors, using an instrument which measures student perception appears valid. We chose to analyze data from six questions that are an evaluation of the course and not the instructor though there is no way for us to tell that the students have not conflated these two effects in their evaluations.

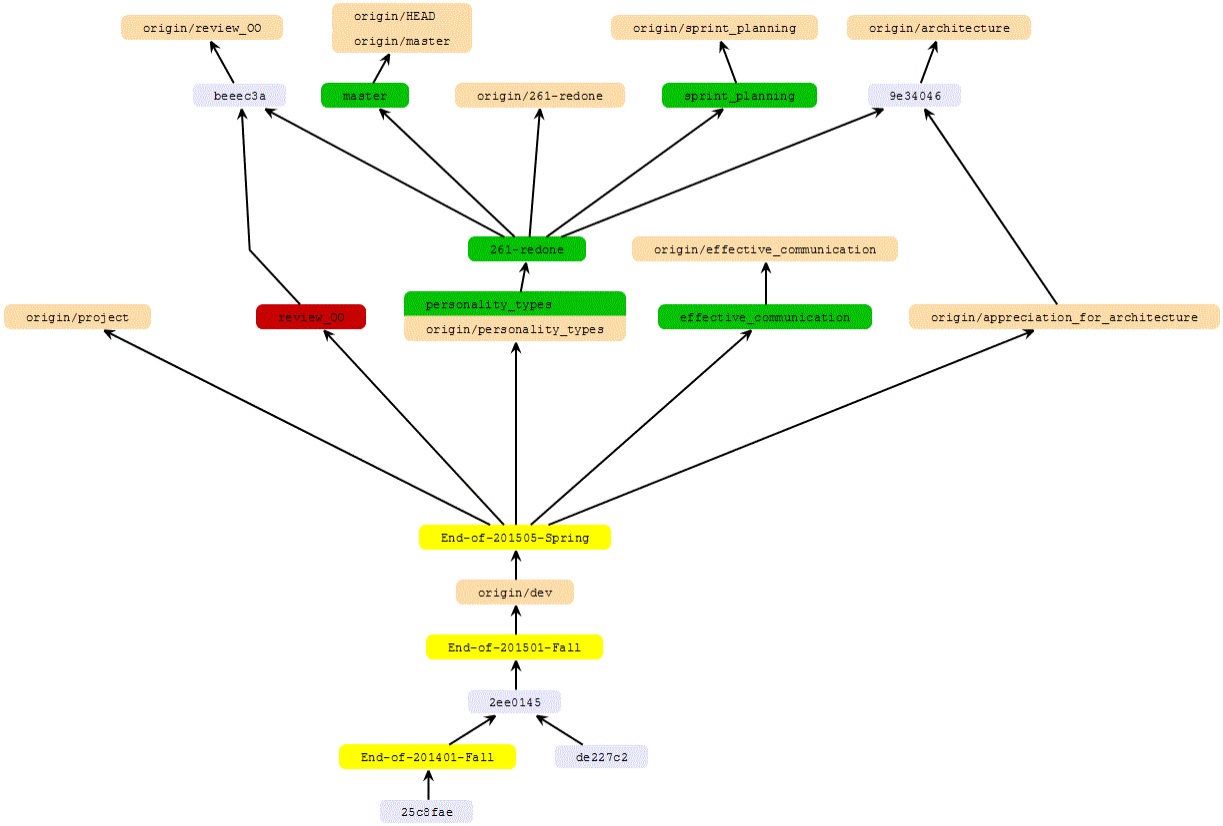


Figure 4 - Feature branches for course topic development

Table 3 below shows the analysis of the course evaluation data. For each question there is data for the three sample sets (10 sections taught with previous material, two pilot sections of the new material, 13 sections taught with new material) which includes the count of course evaluation responses received, the percentage of responses that were Agreeing/Neutral/Disagreeing, and a Kruskall-Wallis test P-value adjusted for repeats. This is a standard test used for the analysis of course evaluation data at our institution because it does not require a normally distributed data set which is not how we expect for course evaluation data to be distributed.

For each question, we tested whether the median of the responses for each set of data for the new course differed from that given by students in the previous version of the course. A P-value less than 0.05 indicates a statistical difference in the two populations that is significant.

- *Advanced my understanding*: Both analyses indicate a change in the agreement level for the course advancing the student's understanding of the subject. For the pilot sections, there was a higher agreement level for advancing the student's understanding. For the full rollout term, the previous version of the course had a higher agreement level.

- *Learned something of value*: In the pilot term there was greater percentage of agreement that the student learned something. In the full rollout term, there was no change in the agreement level between the new and old versions of the course.

- *Course was well organized*: Both analyses indicate that the students had a higher agreement level for the old course being organized compared to their view of the new course's organization. During the pilot term, the new course material was rolling out only hours before the students needed it, and sometimes later than originally scheduled. There

were also several shuffles of the schedule done during the term. In the full rollout term, the syllabus and schedule were mostly stable, but for 10 of the 12 instructors it was all new material so they may have presented the course in a less organized manner.

Table 3 – Course evaluation data analysis

Question	Course	n	Strongly Agree/ Agree	Neutral	Strongly Disagree/ Disagree	Kruskall-Wallis P-value
This course advanced my understanding of the subject.	Previous S2017	134	83.6%	12.7%	3.7%	
	Pilot S2017	34	100.0%	0.0%	0.0%	0.0117
	Full F2017	173	74.0%	12.7%	13.3%	0.0394
I feel that I learned something of value from this course.	Previous S2017	136	85.3%	9.6%	5.1%	
	Pilot S2017	34	100.0%	0.0%	0.0%	0.0178
	Full F2017	173	77.5%	12.7%	9.8%	0.0734
The course was well organized.	Previous S2017	134	68.7%	20.1%	11.2%	
	Pilot S2017	34	50.0%	29.4%	20.6%	0.0378
	Full F2017	173	42.2%	18.5%	39.3%	<0.0001
The objectives as stated on the course syllabus for this course are valuable.	Previous S2017	134	84.3%	9.7%	6.0%	
	Pilot S2017	34	88.2%	8.8%	2.9%	0.5463
	Full F2017	173	77.5%	16.2%	6.4%	0.1557
Overall, I would recommend this course to other students.	Previous S2017	136	47.8%	36.8%	15.4%	
	Pilot S2017	34	64.7%	20.6%	14.7%	0.1453
	Full F2017	173	40.5%	27.7%	31.8%	0.0171
			Much more than/More than most courses	**About average**	**Much less than/less than most courses**	
Compared to similar courses, the amount of work (reading, writing, etc.) in this course was:	Previous S2017	136	72.1%	25.0%	2.9%	
	Pilot S2017	34	91.2%	8.8%	0.0%	0.0189
	Full F2017	173	79.8%	18.5%	1.7%	0.1099

- *Objectives are valuable*: Both analyses indicate the same agreement level for the course objectives being valuable. The statement of the course objectives was not changed, and the new content and delivery did not change the students' perception of their value.

- *Would recommend the course*: In the pilot term, there was no difference in the agreement level to recommend the course. In the full rollout term, the students were less likely to recommend this course than they were for the previous version of the course.

- *Amount of work required*: In the pilot term, the students had a higher agreement level that the course involved more work than the students in the previous version. In the full

rollout term, the students did not differ in their agreement level. This course has always had a large workload in addition to the work/effort involved with the team activities which is new for many students. In the full rollout term, there were significantly improved resources to help the students get through aspects of the course and project development. These were added as a result of lessons learned in the pilot term.

The results of this analysis are as we expected and positive for an improvement in student perception of the new course, except course organization, during the pilot term. During the full rollout term, the analysis is not as universally positive for the new version of the course. There are two areas of student perception of the new course in the full rollout term that we will continue to track. These are *Advanced understanding* and *Recommending the course*. There are several factors that might explain why the students do not have a high a perception of the new course in these areas compared to the previous version and compared to the section in the pilot term. The most likely factor is that for 10 of the 12 instructors in the full rollout term, this was the first time that they were seeing the full details of the course and teaching it. There are typically many rough spots when an instructor teaches a course for the first time especially one that they did not develop. It was not made easier for the instructors that significant fixes were being put in place in close to a just-in-time delivery. A second possible factor is that the technical design content in the course has increased markedly from the old version. Many of the adjuncts who teach the course are not active developers and this material may be outside of the areas of expertise. We do not think that this will prevent them from becoming proficient instructors for this course, but we will continue to track this to see if the perceptions change.

A review of the text comments made in the course evaluations showed that the problems with memorization for exams, there being no connection between class lecture and the project, and the steep learning curve for the technologies has been eliminated. There were suggestions for aspects of the project and the technologies that needed clarification. There were requests for more interactive activities and team project work during class time. Not unexpectedly, there were complaints about keeping track of all the moving parts in the course schedule. These comments were all folded into FIXs in the course development Trello board. Some of them get addressed with each subsequent offering of the course.

A number of students also commented about being required to do some reading or video watching to answer a quiz before class, and then hear about that material in a lecture during class. When we started developing the course, we debated using a totally flipped classroom approach. What we have is a hybrid. There is a small amount of before-class work to answer several simple questions in an on-line quiz. These questions require minimal study of the required resource material. The lecture discussion, which typically is less than 25 minutes, reinforces the material and emphasizes the approach that the students will use on their project work. One concern that we had with a fully flipped classroom for this course is how well this could be handled by the large number of adjuncts teaching the course. Our adjuncts typically look for clear defined guidance on what has to be taught each class or each week. The flipped classroom has a lot looser structure that we think will not work well with many of our adjuncts.

There are additional anecdotal results that indicate we have achieved the goals for this redevelopment effort. With the previous version of the course, the mentoring hours that our student club held had many requests for help with the Django web framework. This demand

caused the club to create special tutoring sessions that were held multiple times in the early weeks of the term. When the new version fully rolled out, the club's Mentoring Head noticed that there was a sharp drop-off in all requests for mentoring help for the Introduction to Software Engineering course. They have also seen a reduction by 50% of the number of students attending exam review sessions.

Several instructors have received comments from students taking the course that indicated the material was very helpful during a co-op interview and when working in a co-op position. Comments such as the following have been received:

> I just started a new job at a software startup in Boston, called Burst. I have been applying almost everything we learned in class this semester. Our team uses Agile and Scrum processes. I have been attending daily standups and doing sprint planning using JIRA. Honestly, I just wanted to thank you for teaching such a useful real-world course. I went in day one and felt like I already knew how our software team was operating. I would have been extremely lost without my knowledge of Agile processes.
>
> I was recently interviewed for a Software Developer position at a company called CloudCheckr. They use the scrum process as well, and use similar technologies that we use in class. So first of all, I'd like to thank you for preparing me for the interview (they asked about SOLID, backlog refinement, etc.).
>
> I wanted to tell you what happened at the co-op interview that I just on. The technical person asked me a lot of questions about SOLID and the other design principles that we learned about in class. At the end of the interview, when everyone was saying goodbye to me, he pulled me aside and said that I had given the best answers to design questions of any student he ever interviewed.

Discussion

The results that we have achieved continue to be positive. This was a complete rework of a major service course in our curriculum required for students in four programs. The data for student perception of the course as given in course evaluations does not show a universal improvement from the previous version of the course, but it may too early to see the steady-state student sentiment because the bulk of the instructors are on their own learning curve with the course material. The anecdotal evidence in terms of student comments has been positive with several instructors hearing comments of how students immediately put the knowledge and skills learned in the course into use on co-op interviews and positions. To the extent that preparing student for their first co-op was a goal, this redevelopment was successful. The perennial complaints (buzzword memorization exams, no connection between lecture and project, steep learning curve for tools used on project) through several previous versions of the course are no longer being heard.

The course does have areas where there needs to be continued development. Each term the course instructors create FIX cards in the course redevelopment Trello board. At an end-of-term

retrospective meeting the next term's backlog is filled with the improvements that we think can be implemented. Some of the new student complaints have to do with how many "moving parts" there are in the course. The students are guided through this by an instructor continually pointing out what exercises are due and what is coming up in the next one to two weeks. Many instructors have taken to displaying a slide listing this information as the students are coming into the classroom. With most submissions done through our course management system, the students can check the next due dates and setup that they receive notifications of deadlines by email.

Keeping track of the "moving parts" is also a challenge for new instructors teaching the course. We have noticed now in the second term of full rollout of the course that there are many fewer questions coming from the instructors who are teaching this for the second time. Over the next term, we plan to develop a Trello list for the course coordinator and the instructors with one card for each course topic. The card would provide instructor guidance for before class, in class, and after class. There would be a checklist of tasks to complete. At the start of the term, an instructor copies the list and uses it to keep track of what needs to be done. We will continue to have a weekly coordination meeting with all the instructors as another mechanism for keeping everyone in sync with the course.

One thing that is significantly more difficult with this version of the course is creating a new project. Because a guideline for the term project is that it involves very minimal user interface work, creating a new project is not simply writing a vision statement for it. The WebCheckers vision statement is the set of MVP features and enhancements along with a statement of the web technologies and software development process to use. Developing the client-side implementation of the WebCheckers gameplay and the documentation describing it was a very significant undertaking. A different project might have a simpler interface and we certainly can envision applications whose interface can be specified entirely with HTML templates and CSS which would make the client-side work much easier. Even so, if the guideline of no user interface work is maintained bringing on such a project is still significantly more effort than writing a one screen vision statement.

Conclusions

This course was developed through the summer and fall of 2016 and delivered for the first time as two pilot sections of Introduction to Software Engineering in the spring of 2017. It progressed to its full implementation in 13 course sections in fall 2017. There continue to be fixes and tweaks done to the material to improve it as we gain more experience with large groups of students and new instructors being brought up on the material. The evidence shows that we accomplished the goals that we set out for the redevelopment of this foundational software engineering course. The course eliminated the perennial complaints that students had with the course, emerged the students in significant discussion and practice of object-oriented design, and have the students practicing contemporary software development techniques using up-to-date tools. With the full rollout to all course sections, we have shown that the course has legs and is not just dependent on the energy of the two original instructors who took on this massive redevelopment project building the course from the ground up.

References

[1] Joint Task Force on Computing Curricula IEEE Computer Society/Association for Computing Machinery, "Software Engineering 2014 – Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering", 2014. https://www.acm.org/binaries/content/assets/education/se2014.pdf

[2] Joint Task Force on Computing Curricula Association for Computing Machinery (ACM)/IEEE Computer Society. Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science, 2013. http://ai.stanford.edu/users/sahami/CS2013/final-draft/CS2013-final-report.pdf

[3] Reichlmayr, T, "The agile approach in an undergraduate software engineering course project", Proceedings of the Frontiers in Education Conference, 2003.

[4] Reichlmayr, T, "Enhancing the student project team experience with blended learning techniques", Proceedings of the Frontiers in Education Conference, 2005.

[5] B. Bloom, "Taxonomy of Educational Objectives: The Classification of Educational Goals," Mackay, 1956.

[6] http://www.se.rit.edu/~swen-261/

[7] http://epf.eclipse.org/wikis/openup/

[8] https://www.lynda.com/

[9] http://www.skillsoft.com/

[10] http://sparkjava.com/

[11] http://www.se.rit.edu/~swen-261/resources/resources.html