Undergraduate Program Learning Outcomes Assessment Summary Report 2016

Department, Program & Degree: Computer Science Department, Undergraduate Program, Bachelor’s Degree

Contact Person: William Gasarch

Date: 10/10/2016

This Undergraduate Program Learning Outcomes Assessment Summary Report is due to the Provost from each undergraduate degree program on October 19. [Colleges will collect these reports prior to this deadline and submit together on behalf of each Dean, and may set prior internal deadlines accordingly.] Please limit response to each question to approximately 100 words and limit the use of undefined acronyms. Attach supporting documents as appendices. Please include examples of assessment tools (prompts used to generate student work that was assessed such as pre/post test questions, questions sets, assignment instructions, etc.) and rubrics or statements of criteria used to assess that student work.

Actions Taken as a Result of Past Assessments.

1. Consider findings from past learning outcome assessments and feedback from the Provost’s Commission on Learning Outcomes Assessment. In the last academic year (Fall 2015-Spring 2016):
   a. What decisions were reached and/or what actions were taken (changes to courses, curricula, academic structure) to improve student learning?

   **Intro Sequence 131, 132, 216**: The main upshot of the prior evaluation is that students need to be better trained in in terms of program style. Since then some sections of these courses emphasize style more in terms of both grading and feedback from the TAs.

   **400-level courses**: The main upshot of the prior evaluation was that the faculty needs to reach a consensus about what our goals should be, with perhaps an eye towards being more real-world. The faculty had not done this. Individual courses have been updated and some new courses that are more real-world (e.g., CMSC 436 Mobile Computing) have been added.
**Theory Courses (250, 351, 451):** The main upshot was that the people who teach theory course should meet and discuss the contrasting approaches of applications-first or theory-first in courses. CMSC 351 has gone to apps-first, CMSC451 already was that way, and for CMSC250 this is still being debated.

Click or tap here to enter text.

b. What, if any, changes have you made to your assessment process and why?

We have made no changes to the assessment process.

c. If you received any scores of “unsatisfactory” (0 or 1), describe how you have addressed concerns raised in the feedback. There were no such cases.

### Four-Year Assessment Plan (Fall 2014 – Spring 2018)

2. Consider your 4-year assessment plan for Fall 2014 – Spring 2018.

   a. List all of the learning outcomes for your program.
      
      1) Graduates should be able to create, augment, debug, and test computer software. This should be accomplished after the three course intro sequence
      
      2) Graduates should have mathematical and analytical reasoning skills, and be able to apply them to algorithms and protocols.
      
      3) Graduates should experience design and implementation of programming projects that are similar to the real world environment
      
      4) Graduates should be able to work with people: on a team, getting user feedback, giving presentations.
      
      5) Graduates should have exposure to either academic research or the business environment or both.

b. Briefly summarize your 4-year assessment plan for AY15-AY18 (to provide context for your results). Please note any departmental or programmatic special circumstances that provide context for this plan or the work of this cycle.
Note that above we have five learning outcomes. We will evaluate 3 then 2 then 3 then 2 etc. This year we did 1,3,4 so next time we’ll do 2,5. We hope to start much earlier, the prior semester, so that the information is more available.

Results, Conclusions, and Implementation from Last Academic Year (Fall 2015-Spring 2016)

3. Describe the engagement of faculty and others (e.g., staff, students, alumni and or outside professionals of the field) in the assessment process. Examples might include participation in planning, collection, analysis, review, or decision making.

A seven person committee of 5 professors and 2 lecturers did the assessment. While doing so they interacted with the lecturers who taught these courses.

[Repeat items 4 - 8 for each additional outcome. To repeat this section for additional outcomes, click the document somewhere before #4 and #8 and select the blue ‘+’ symbol on the right.]

4. Please state the outcome you discuss below.

Graduates should be able to create, augment, debug, and test computer software

5. How did you measure student learning for this outcome?

For CMSC 131,132,216 we looked at three projects: one early in the term, one in the middle, and one late in the term. For each of those projects we looked at 10 student submissions picked at random. We looked at the submissions for how good they were in terms of style and correctness. We also asked the lecturers about how much testing the students do of code and any other issues they wanted to bring up.

6. What were the results of your assessment? What did you find?
The student’s code ranged from acceptable to very good. Talking with the teachers we find that the students in CMSC 132 have many assignments that involve testing. Anecdotally we also heard (and other sources like teaching evaluations back this up) that students who come in with less of a programming background are at a severe disadvantage in these courses, and that for all students the jump from 132 to 216 is rather steep.

7. How do you interpret these results? What conclusions did you draw?
   This is covered in the next question.

8. What was the consensus of your program’s discussion of these results? What actions are you going to take as a result of your discussion and analysis?
   We recommend that the department have a course intended for majors who do not have a programming background. Also, the issue of the jump to 216 needs to be discussed more.

4. Please state the outcome you discuss below
   Graduates should experience design and implementation of programming projects that are similar to the real world environment.

5. How did you measure student learning for this outcome?

   CMSC 430, introduction to compilers, is a course where students write a compiler. This is a large scale project similar to the type of project one would do in the real world.

   Two assistant professor (David van Horn, Tom Goldstein) looked at two projects from the course when it was taught in Fall 2015 (by Jeff Foster). For each project they looked at 10 student submissions. They judged each one on Code Quality and Comment Quality. Code quality is obviously important in all
settings. Comment quality is important in the real world where more people see your code. They judged both on a scale of 1 (poor) to 5 (excellent)

6. What were the results of your assessment? What did you find?

The average quality of code was 3.7 (fair) in the second project and increased to 4 (good) on the fifth project.

The average quality of comments was 2.7 (poor) in the first project and increased to 4 (good) on the fifth project.

7. How do you interpret these results? What conclusions did you draw?

Coming into the course the students’ skills for coding was fair and at commenting were poor. Over the course of the semester both improved a lot.

8. What was the consensus of your program’s discussion of these results? What actions are you going to take as a result of your discussion and analysis?

While it is tempting to say that the introductory courses should spend more time on comments this is not feasible given their size and what they already do.

The upper level courses should continue to stress code quality and comments. We will discuss with other teachers of 400 level courses if they are stressing code quality and comments and if not then urge them to do so.

4. Please state the outcome you discuss below.

Graduates should be able to work with people: on a team, getting user feedback, giving presentations.

5. How did you measure student learning for this outcome?
The course CMSC 436, Mobile Devices, has a team project that involves talking to users about what they want. The project is presented to the class. Hence we chose this course to evaluate in terms of the learning outcome above.

A committee of one lecturer (Evan Golub) and one professor (Adam Porter) looked at the project assignments in CMSC 436 from Spring 2016. The final projects were live presentations and hence were not available. The committee looked at three videos made about the presentations which involved 14 students. They also talked with the instructor (Vibha Sazawal) about how the students did.

6. What were the results of your assessment? What did you find?

The project involved both working on a team and soliciting input from users. The students did very well at both.

This was established in two ways: (1) the three videos showed that those students did an excellent job, and (2) The instructor says that of the 72 students in her class, 66 contributed to their projects in substantial ways.

7. How do you interpret these results? What conclusions did you draw?

The students are achieving the desired learning outcome of being able to work closely with other people.

8. What was the consensus of your program’s discussion of these results?

What actions are you going to take as a result of your discussion and analysis?

As far as the course is concerned, no action is needed. In terms of evaluating the course we will ask the Middle States Administrators to give us much more lead time so that we can evaluate courses while they are happening rather than after the fact.

**Plans for This Academic Year (Fall 2016-Spring 2017)**

9. For which outcomes will you be collecting information over this academic year (Fall 2016-Spring 2017)?
Mathematical reasoning, Undergraduate Research, Business exposure.

10. How will you measure student learning for these outcomes?

For math reasoning we will look at final exams in the courses Discrete Math (250), Algorithms (351), Design of Algorithms (451), and Cryptography (456). On the exams we will see if the students can think rigorously. For research we will look at finished undergraduate research projects and judge their quality. We will also talk to some of the mentors about how they recruit students and how they are doing. For the business exposure we will examine the corporate scholars program though honestly we’ll know more how to evaluate it after we talk to faculty that have been involved.
Assessment of the Introductory Sequence (131/132/216)
Bill Gasarch, Fawzi Emad, and Clyde Kruskal

Overview
The purpose of this report is to measure how effective our introductory courses (CMSC 131, CMSC 132, and CMSC 216) are in teaching students to write, test, and debug programs. We also consider how well our courses progress (as a sequence) and offer some suggestions for improvement. This assessment is based on samples of student code, and feedback from instructors who taught these courses during the last academic year. We have divided our effort into three sections: (1) student projects; (2) program testing; and (3) program debugging.

Part I: Analysis of Sampled Projects
For each course under consideration, we analyzed three programming projects: One from the beginning of the term, one from the middle, and one from the end. We gathered a random sample of ten student implementations for each project. We evaluated these samples independently from the grades that were earned by the students.

CMSC131 Projects

Early Project
Students write a program that asks the user a few simple questions about monetary denominations (e.g. whose picture appears on a twenty dollar bill), and decides whether or not they have answered correctly. The goals are for students to practice using variables, nested conditional statements, input using Java’s Scanner, and logical operators.
Mid-Semester Project
Students write two Java classes: One represents a postal address, the other simulates a photo processing system in which a customer can request that a photo be manipulated in various ways. (The actual photo manipulation is handled by instructor-written code.) The goals of this project are for students to practice writing complete Java classes, use the keyword “this”, practice exceptions, learn the StringBuffer class, and parse Strings.

End of Term Project
Students implement a simulation of a system that allows customers to rent movies and music, and receive them in the mail. This project focuses on students designing much of the project themselves. They also practice inheritance, the Java ArrayList class, and the Javadoc automated documentation utility.

Summary/Analysis
The 131 projects seem to be at the right level and build nicely from beginning to end.

Early Project: The first project seems simple enough, but for students who are new to programming, we can see that nesting of conditional statements and just learning basic Java syntax can be quite challenging. With help from the teaching assistants, most of the students who were struggling managed to submit working code; in fact, 90% of our samples demonstrated mastery of the project goals.

Mid-semester Project: Although the mid-semester project involves much code that is written by the instructor, it is fun, and students learn how their work can be integrated with work done by others. Of the sampled projects, 80% showed an understanding of the material; however, even among those projects there were still some rough points.

End of Term Project: The last project serves as a nice capstone, allowing students to combine many of the concepts they’ve learned and to be creative and design something of their own. The quality of these projects is harder to assess because students had a lot of leeway to choose their own design. However, our estimation
is that approximately 80% of the implementations demonstrate a satisfactory level of understanding.

After looking at many student implementations, we see that students exiting this course have mastered the basics of programming: variables, conditional statements, iteration, arrays, and even some intermediate Java features such as ArrayList and Exceptions. However, anecdotal evidence suggests that students who enter the course having had little or no previous programming experience struggle, and often drop or fail to earn a passing grade.

**CMSC132 Projects**

*Early Project*
Students write a simulated “submission server” in which records are maintained for “project submissions” (really just Strings) and corresponding scores (integers). Various simple methods are implemented. The students are free to choose how the underlying data is stored, although the instructions are specified precisely. The goals of the project are for students to learn about the project submission process and to review much of the material that they should already know prior to entry into this course (processing ArrayLists, Exceptions, etc.)

*Mid-Semester Project*
Students implement a sorted linked list using Java generics. All of the usual linked list methods are present (adding, removing, search, etc.) plus a few extras like sublist. Students also write an iterator() method that returns a Java Iterator satisfying the usual contract. The goals of the project are for students to practice Java generics, operations on a linked list, Iterators, throwing exceptions (checked and unchecked), and inner classes.

*End of Term Project*
This project builds on two previous projects in which students have written programs that simulate a “submission server”. This one adds the ability to process multiple submissions concurrently using separate threads. Part of the challenge of the project is that students are forbidden from using collection classes that
automatically provide synchronization, so they must practice carefully writing code that is thread-safe despite the threads sharing common data. This project is not fully specified, so students are allowed a great amount of leeway to think about and create their own design for accomplishing the project goals.

**Summary/Analysis**

Student implementations reveal that the goals of these projects are being satisfied by the majority of students. The 132 projects are organized in a very efficient and logical way, and although the project descriptions seem a bit verbose, the difficulty-levels of the actual implementations are not overly demanding: The assignments are challenging, but not unreasonably so.

*Early Project:* The first project may be a bit on the easy side, as it was largely meant as a review of previous material. 90% of student submissions illustrated competency with the project goals.

*Mid-Semester Project:* This project was somewhat complicated and most of the submissions were (appropriately) long. Despite this, the students seemed to understand the material, and 80% of the project submissions were organized well.

*End-of-Term Project:* This project built on the early project (as well as another project in between). The student implementations varied widely since this was a somewhat open-ended assignment. The multi-threaded aspect of the project could have been more challenging. 80% of the project implementations were satisfactory.

We like the way projects #1, #6, and #8 build upon each other – it’s good for students to see how the same core ideas can be developed in increasingly complex environments. Judging from the student implementations, we see that students are learning to write interesting code using many of the intermediate and higher-level features of the Java language. These are quality projects.
CMSC216 Projects

**Early Project**
In this project, students implement a hash table with open addressing (linear probing). Goals of this project are for students to learn about hashing, practice basic C constructs such as pointers, const, and memory management. Students also practice using Makefiles.

**Mid-Semester Project**
This project again requires students to implement a hash table, but this time using closed addressing (linked lists are used as buckets). The goals of this project are for students to create a data structure that is fairly complex and requires lots of dynamic memory allocation (and deallocation). Students also have to think carefully about data types and some of the subtleties that are unavoidable when creating this kind of structure in a raw language like C.

**End of Term Project**
Students write a shell that can handle operations such as pipes, input and output redirection, and the “&&” operator. The students must implement the “executor”, which takes a tree from a parser and executes the commands. Goals of the project are for students to practice system level calls, and to think carefully about how to handle the many different cases that arise in parsing this kind of tree.

**Summary/Analysis**

**Early Project** – We found that 80% of sampled projects demonstrated mastery of the material. The difficulty of the project overall is not that high, but since students are learning C, they struggle with this first project.

**Mid-Semester Project** – We found that 70% of sampled projects demonstrated mastery of the material. However, even those that mastered it still had some rough spots. This project builds on the early project, but requires more in depth knowledge of hashing and more advanced coding in C.

**End-Term Project** – Again, we found that 70% of student implementations demonstrated mastery of the material. This is an excellent, though rather difficult project. The number of lines of code in these implementations is not large,
however it requires in depth knowledge of more advanced C programming as well as system-level considerations.

Based on student implementations, we feel that 70% are reaching the goals of these projects, but those with lower grades may be struggling with some of the fundamentals. The 216 projects are a bit of a departure from the 131/132 projects: They are somewhat more demanding, the specifications might be a bit harder to read, and the code is in some cases fundamentally more difficult to write. That being said, the projects are not preposterous or absurdly hard – just a bit harder than those in the previous two semesters. Students who successfully implement these assignments are learning the nuances of C programming, gaining some exposure to system-level programming, and are mastering some of the nuts and bolts that underlie programming in higher-level languages. The projects are very solid, but perhaps just a bit on the challenging side.

Suggestions for Improvement
Some of the project descriptions seem unnecessarily verbose. Students may find it difficult to orient themselves when presented with such a deluge of information all at once. A couple of the project descriptions were a bit confusing to read (in particular the “end of term” 216 assignment). It seems that in some cases like this one, it might be helpful to make it more clear what parts of the project are provided by instructors, and what will be implemented by the student. Also, it would be helpful to find a concise statement of the goals of the project somewhere in the beginning – this would help students to understand exactly what they are facing and what is expected from them.

Part II – Student-Written Tests
One of the fundamental goals of the introductory course sequence is teaching students to write thorough and effective unit tests to check the correctness of their own code. We looked at how these “student-written tests” are incorporated into our course offerings.
Projects Requiring Student-Written Tests

<table>
<thead>
<tr>
<th>Course</th>
<th>Projects requiring Student test submissions</th>
<th>Fraction of grade from Student-Written Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMSC 131</td>
<td>1 of 8</td>
<td>10% of the project grade</td>
</tr>
<tr>
<td>CMSC 132</td>
<td>5 of 8</td>
<td>10% of the project grade</td>
</tr>
<tr>
<td>CMSC 216</td>
<td>2 of 7</td>
<td>12% of the project grade</td>
</tr>
</tbody>
</table>

The data in the chart above suggests that student-code is not being evaluated and graded on a regular basis in 131 and in 216. (Apparently in 132 the student-written tests are being collected more frequently). There may be a bit more to the story though... (See section below on “Secret Testing”.)

Quality of Student-Written Tests

We measured both the number of lines and “quality” of student-written tests from the Introductory sequence, and those numbers appear below. The “Average Quality” ratings are based on a subjective scale of 1 to 5, and represent the average of the opinions of the committee. We chose to report the median for lines of test code since there were a few outliers that would have yielded a misleading mean.

<table>
<thead>
<tr>
<th>Course</th>
<th>Median number of lines of test code per submission</th>
<th>Average quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMSC 131</td>
<td>220</td>
<td>3.4</td>
</tr>
<tr>
<td>CMSC 132</td>
<td>265</td>
<td>3.6</td>
</tr>
<tr>
<td>CMSC 216</td>
<td>490</td>
<td>3.0</td>
</tr>
</tbody>
</table>

The “average quality” figures should be viewed skeptically, since the sample size was relatively small, and the values given are quite subjective. However, at face value it seems disappointing that the quality of tests drops off substantially in CMSC 216. This is, in part, due to the fact that the code in this course is being written in C (instead of Java). Also, the nature of the projects under review may play a factor in the difficulty of writing decent tests. But the data does suggest
that more effort could be made (perhaps in the lab sessions) to improve CMSC 216 students’ ability to write useful test code in C.

The role of “Secret Tests”

Secret testing means that the student is not told whether or not he/she is passing the instructor-written tests that are being used to grade projects until after the project due date has passed. One of the benefits of secret testing is that it encourages students to write their own tests, since they have no other way of knowing how well their code is working prior to the project due date.

Utilization of “Secret” Testing (All Projects)

<table>
<thead>
<tr>
<th>Course</th>
<th>Percentage of projects with some secret tests</th>
<th>Overall Percentage of testing that was “secret”</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMSC 131</td>
<td>13%</td>
<td>3%</td>
</tr>
<tr>
<td>CMSC 132</td>
<td>37%</td>
<td>9%</td>
</tr>
<tr>
<td>CMSC 216</td>
<td>100%</td>
<td>63%</td>
</tr>
</tbody>
</table>

Suggestions for Improvement

In CMSC 131 there is a project in which secret testing is used for 20% of the project grade. Although this offers a bit of encouragement for students to write their own tests, we feel that it is not enough, and that more test code should be collected and graded in 131 where students are only beginning to learn how tests should look. This kind of feedback would be very useful for beginners to learn how to write a wide variety of tests, including those that cover cases and a thorough mix of more traditional cases. Further, once students have been taught how to write their own test cases, utilizing some additional amount of secret testing in projects might encourage students to be more serious about writing tests of their own.

In CMSC 132, in addition to the thorough grading of student-written test-cases, Secret testing is used for nearly all of the student’s grade on two of the remaining projects. We feel like this is a nice blend that provides much useful feedback to students on the graded tests while also getting students used to the secret testing
that will be used for much of the grading in 216 (see below). We have no suggestions for 132, with regard to student-written tests.

In 216 the **majority** of the grade on every project is from secret testing! So if students in 216 are not writing effective tests, they are very unlikely to submit project implementations that will earn passing grades. Although this seem fine, we feel there would still be much to gain if more of the projects required students to submit their tests for grading. Writing test code in C is very different from the JUnit tests that students are used to writing in CMSC 131 and 132. It seems like additional feedback from graders could be of value here.

**Part III – Debugging**

Encouraging and teaching students to use a debugging tool effectively is another fundamental goal of the introductory sequence. We asked the instructors a little bit about how use of the debugger was incorporated into their courses.

**Classroom time spent on Debugging**

<table>
<thead>
<tr>
<th>Course</th>
<th>Lecture Time</th>
<th>Discussion Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMSC 131</td>
<td>2 hours</td>
<td>4 hours</td>
</tr>
<tr>
<td>CMSC 132</td>
<td>0 hours</td>
<td>0 hours</td>
</tr>
<tr>
<td>CMSC 216</td>
<td>2 hours</td>
<td>4 hours</td>
</tr>
</tbody>
</table>

The Eclipse debugger is demonstrated in CMSC131, and one of the available command-line debuggers is demonstrated in CMSC216. Since no new debugging tool is being introduced in CMSC132, we see that no classroom time is dedicated to this purpose there.

Instructors report that although it is necessary to teach the basic operation of the available debugger in the classroom, it is very difficult to show students how the debugger will be utilized in “real world” situations because any example that is small enough to demonstrate in class is probably so simple that the use of a debugger is not really of much value. (Again, there is more to the story... See the next section.)
The Role of Guided Office-Hour Debugging

Instructors report that although minimal time is spent in the classroom demonstrating how to use a debugger effectively, the most valuable demonstrations of how to use the debugger in real-world situations frequently occur during office hours. In all three courses, instructors report that students who are stuck on fixing a project bug are encouraged to arrive at office hours having prepared (if possible) a test case that reliably demonstrates failure. Given this starting-point, instructors or teaching assistants can then help the student to use the debugger to figure out what is going wrong. Although this model doesn’t always work, in many cases it does, and it is certainly a valuable experience for a student to sit down with an experienced programmer to see how to make the most of one of the available debuggers.

Suggestions for Improvement

We feel that instructors should consider writing more assignments (perhaps lab exercises rather than official projects) in which students are given some sample of buggy code, and then are expected to use the debugger to find and correct the errors. Although the code couldn’t be too intricate, it still might be useful for students to practice using the debugger in this environment where an experienced TA (or even more experienced students) can help. Using the debugger can be confusing and daunting for students who have never tried it before; we want them to become comfortable enough with the debugging tools that they will naturally turn to them on their own when they run into bugs in their own code.

Conclusion and Further Suggestions

What we Liked

Regarding the progression of difficulty: The transition from 131 to 132 is seamless, almost as if the two courses were halves of a one-year sequence. The difficulty of the projects builds gradually and comfortably. Students should not find any abrupt surprises here.
Suggestions for Improvement

Students with little or no prior programming experience have a hard time adjusting to the pace of CMSC 131. For this reason, we strongly recommend a new course that could serve as a prelude to 131 for students who have little or no prior programming experience but are considering majoring in Computer Science.

The transition from 132 to 216 may be too big of a jump. This is partly due to the intrinsic differences in Java programming versus C programming, but it could also be the case that the 216 project descriptions could be made a bit more clear, and that some of the projects may be a bit too challenging for third-semester students. We recommend that the 216 syllabus be re-examined with an eye toward making the transition from 132 to 216 smoother.
Evaluating "Human Interaction" experiences of Computer Science Majors using Middle States criteria

by

Evan Golub and Adam Porter

The student learning outcome listed for this course was, "Graduates will be exposed to working closely with other people. This human interaction is manifested in several ways: design of software/hardware based on user input and feedback, working as a member of a programming team, and making presentations to groups about what has been designed and/or implemented."

After reviewing the course syllabus and talking with Vibha Sazawal, the faculty member who taught the course, we determined that we could assess the above-mentioned outcome by analyzing the semester-long, team-based project given in CMSC436 during the Spring 2016 semester. The project involved several stages, specifically; project design and documentation, peer review of wireframes, and a final demonstration. We also gathered input from the faculty member's assessment of individual participation within the project teams.

The project design document asked students to describe the mobile app they would build, what goals it would help a user accomplish (with at least four discussed), and why a user would be motivated to download the app. If the potential users were not college students, the team was required to speak with example users to inform this document. For this particular semester, many of the apps were aimed at college-aged students, but we did review one, focused on food recovery efforts, where the target audience required the students to reach out to non-student potential users. In designing that app the project team spoke to two employees at stores that donate leftover food, and a member of an organization that reaches out to potential sources of leftover food to then distribute it for free.
As part of the design of their projects students built app wireframes. The students later participated in peer reviews of these app wireframes. Since this was done "live" in the classroom, we could not directly assess that activity. However, based on the description provided to us by the faculty member, all teams appeared to have participated in this process and thus received critical feedback from other students through it.

At the end of the semester students prepared and delivered a live demonstration to the faculty member, including a video demo. We reviewed three of these video demos (e.g., the ones for Team Explore, PSous, and Beacon). These teams created non-trivial presentation videos that told the "story" of their app and its usage. In their own ways, each team’s video showed how their app might be used in context and presented details of typical user interactions individual screens of the app. While the quality of the videos differed in style and polish, they all showed true understanding of how to present their work to others.

Finally, though there was no explicit peer-evaluation for teams, the faculty member was in communication with the students and was able to get feedback regarding the student experiences working as a member of a team. Of the 72 students in the class, her assessment was that at least 66 students were contributing members of their team, while at most 6 were not.

Overall, there were thirteen teams comprising a total of 72 students at the end of the semester. The three teams whose videos we reviewed represented 14 of these students. The selection was based in large part on which teams' final presentation videos were still available online.

We feel there is no reasonable way to independently assess the individual students in retrospect in a course like this, unlike categories where written exams can be used after the fact. However, using what we have as a sample, and drawing reasonable inferences, we are confident that the course helps students achieve the goals of being able to work with other people.
**Evaluation of Upper-Division Real World Programming Project**

Tom Goldstein & David Van Horn

**Overview**

We analyzed projects that were completed in CMSC430 (introduction to compilers) to determine whether programming-intensive projects enhance students’ software development skills, and also to provide suggestions on how projects can be improved in the future. The assessment was performed by two assistant professors, one who has taught this course in the past and one who has not seen students’ performance in this course before.

**Selection Criteria**

We chose to select two projects from the course for evaluation. Because our goal was to access how students gain skills over the course of the semester, we chose one early-semester project, and one from late in the semester. The course consisted of 6 projects, some of which formed a sequence that built upon one another. We chose to review project 2 as the early–semester project because it was much more coding intensive than project 1, and thus provided a more reliable assessment. We chose project 5 as the late-semester project because it was the latest project to be part of the main cumulative project sequence.

For each project, we examined the work of ten students. Each project was given 2 scores: one for the quality of the code, and one for the quality of the comments. We find that comment quality, while highly important in industry, is often not emphasized in academic environments. Students are typically graded
primarily on how well their code passes well-defined tests that measure the behavior of the code relative to the assignment. Our rubric, on the other hand, focuses on more qualitative measures, such as modularity, clarity, and extensibility.

**Rubric**

To assess the quality of student written code, each project was ranked using the following five-point scale:

1. Code is poorly written or inadequate.
2. Code is serviceable, but not well crafted. May be hard to understand or maintain.
3. Code is solid. Lacks clever ideas that might make it more modular, compact or effective.
5. Exceptional programming skill. Code is unusually clear and concise. Excellent use of abstractions and modularity.

Student written comments were accessed using the following criteria:

1. Comments are absent or nearly absent.
2. Code has some comments, but they are unclear or inadequate.
3. Comments are widely present, but some comments are unclear and lacking details.
4. Comments are adequate and clearly written.
5. Exceptional and extensive comments. Comments address both low-level details of code functionality and also high-level issues like how the code is to be properly used.

**Raw Data**

Results from the survey appear in the table below. In addition to raw scores, we also report the change in score between the two projects, and also marginal averages of scores.
<table>
<thead>
<tr>
<th>Student</th>
<th>Code</th>
<th>Comment</th>
<th>Code</th>
<th>Comment</th>
<th>Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>-2</td>
<td>-3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Mean</td>
<td>3.7</td>
<td>2.7</td>
<td>4</td>
<td>4</td>
<td>0.3</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Note that all students increased their code and comment scores over the semester with the exception of student 3. Student 3’s performance on project 5 is uncharacteristic of his work on other projects, and is likely an outlier due to poor time management. When student 3 is removed, code scores increased on average by 0.6 units, and comments by 1.7 units.

**Analysis and Conclusions**
While both code and comment scores increased, the improvement in students’ use of comments was the most dramatic. This is likely due to the fact that
students are not used to being graded on comments, and likely started from a lower baseline and improved through the semester by getting feedback from the graders.

Improvement in code scores was fairly modest, with most students starting out with fairly good marks even in the earlier project. This indicates that lower-division courses were likely successful at teaching basic coding skills, and that students were probably expecting coding style as a grading criteria.

Despite this, nearly half of the students scored less than a 4 for code quality on the late-semester project. Ideally, by the end of an upper-division course a larger fraction of students should be receiving high marks in this area. This indicates that it is important to keep code style and structure as a grading criteria, even for upper-division classes. One of the dangers of automated grading is that it makes it difficult for students to get feedback on code structure and style.