Design and Evaluation of "The Missing CS Class," A Student-led Undergraduate Course to Reduce the Academia-industry Gap

Grant Gilson, Stephen Ott, Noah Rose Ledesma, Aakash Prabhu, and Joël Porquet-Lupine*
{ggilson,stott,roseledesma,aakprabhu,jporquet}@ucdavis.edu
Department of Computer Science
University of California, Davis
Davis, California, USA

ABSTRACT

One notable part of the academia-industry gap is the deficiency in computing ecosystem literacy, which may result in college graduates exhibiting little technical knowledge of software development tools and practices commonly used in industry. This paper presents our experience developing and teaching "The Missing CS Class," the student-led 1-unit course that we created at our university to address computing ecosystem literacy. This course primarily targets lower-division students and, based on our observations as peer tutors, covers four common but crucial gaps in technical knowledge: (1) Unix-like command-line environments and tools, (2) Software testing and debugging, (3) Scripting, and (4) Version control. Based on the collected feedback from two consecutive offerings of this course during the winter and spring quarters of 2021, most surveyed students reported having increased their self-efficacy on all course topics and incorporated them into their software development workflow.

To benefit the community at large, we have published all the lecture materials online at https://missing.cs.ucdavis.edu.

CCS CONCEPTS

• **Applied computing → Education**; • **Social and professional topics → Computer science education**.

KEYWORDS

Computer Science Education; Academia-industry Gap; Student-led Undergraduate Course; Computing Ecosystem Literacy

ACM Reference Format:
Grant Gilson, Stephen Ott, Noah Rose Ledesma, Aakash Prabhu, and Joël Porquet-Lupine. 2022. Design and Evaluation of "The Missing CS Class," A Student-led Undergraduate Course to Reduce the Academia-industry Gap. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499422

*Gilson, Ott, and Rose Ledesma are the undergraduate students who equally contributed in defining the course topics, creating the course materials, teaching the course twice, and writing an early draft of this paper, under the active guidance of Prabhu and Porquet-Lupine. The final version of this paper was written by Porquet-Lupine.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SIGCSE 2022, March 3–5, 2022, Providence, RI, USA.
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9070-5/22/03.
https://doi.org/10.1145/3478431.3499422

1 INTRODUCTION

In traditional introductory computer science courses (i.e., CS1 and CS2) instructors typically focus on fundamental topics such as programming, algorithms, and data structures, in accordance with official computing curriculum guidelines [5, 8]. While this approach establishes a core of applicable knowledge for students, the training of practical skills, such as Unix-like command-line tools, testing, debugging, etc., often tends to be neglected. This deficiency in *computing ecosystem literacy* contributes to reinforcing the academia-industry gap [14–16], as it may result in students with solid abilities to solve well-formatted programming problems but little technical knowledge of software development tools and practices commonly used in industry.

This widening gap in technical knowledge became apparent to us during our experience as undergraduate peer tutors. We assisted plenty of students who were familiar with the concepts of computing ecosystem literacy but showed a lack of practical skills and self-efficacy [17]. For example, students would understand that a `Makefile` can efficiently automate the building of their code, but were too afraid to part from the plain "compile-and-run" button of their graphical IDE. Further informal interactions with upper-division instructors also revealed numerous complaints that students lacked proficiency with Unix-like environments and tools. To confirm this issue's predominance throughout our CS undergraduate population, we conducted a survey in February 2020 and collected 220 student responses. Among them, only two-thirds of respondents reported being confident in their ability to navigate a filesystem via a command-line interface, and less than half reported being comfortable with more advanced utilities such as `grep` or `find`. Similarly, only a third reported practical knowledge of shell scripting. However, it appeared that students recognized their non-proficiency in computing ecosystem literacy as a serious issue, since an overwhelming ~90% of respondents expressed an interest in taking a course that would focus on it.

This paper presents our experience developing and teaching "The Missing CS Class," the student-led 1-unit course that we created to address computing ecosystem literacy. This course primarily targets lower-division students and, based on our observations as peer tutors, covers four common but crucial gaps in technical knowledge: (1) Unix-like command-line environments and tools, (2) Software testing and debugging, (3) Scripting, and (4) Version control. The student feedback from two consecutive offerings of this course during the winter and spring quarters of 2021 was very positive. Most surveyed students reported having increased their self-efficacy on all course topics and incorporated them into their own software development workflow.

This paper is organized as follows. First, in Section 2, we provide an overview of related courses from other institutions, including their goals and methodologies. Then, in Section 3, we explore the principles of our course design. In Section 4, we present a detailed layout of our course. We discuss the course offerings and end of quarter survey results in Section 5. Finally, we conclude and suggest potential next steps in Section 6.

2 RELATED WORK

While most institutions offer core courses that introduce some computing ecosystem literacy, it is typically covered as supplementary to the main course topics. For example, the introduction to Unix-like environments and associated tools may be a single lecture or lab in a computer systems course [4, 18], as may be the introduction of unit testing in a programming course [6, 13]. The few existing courses which we found that focus exclusively on computing ecosystem literacy tend to be student-run and not made a degree requirement for undergraduates, exactly like ours.

In January 2019 and 2020, students at the Massachusetts Institute of Technology (MIT) taught "The Missing Semester of Your CS Education" [1]. This class, which greatly inspired our initiative, was offered during MIT's "Independent Activities Period", a short one-month semester featuring a variety of student-run classes. It aimed to provide a hands-on introduction to tools and techniques, topics that these MIT students also felt were insufficiently addressed by their school's curriculum. While some of the contents covered in this class overlap with our course, they seem to favor breadth over depth. For example, the class brushes on multiple additional topics such as security and cryptography, build systems, continuous integration, etc. In contrast, our course dives deeper into software testing and debugging, and does not limit itself to shell scripting.

While initially created by an undergraduate student in 2011, Carnegie Mellon University's "Great Practical Ideas for Computer Scientists" [3] has become an official 2-unit elective course generally offered once a year. The course presents "the common tools that computer scientists use," which includes `Vim`, `Bash`, `Git`, and LaTeX. It also offers additional workshops and talks outside of the required class material. There is again some overlap between this course and ours, especially regarding some of the presented tools. However, while this course spends more time on certain tools (e.g., 2 weeks on `Vim`), our course also covers software development techniques and practices, such as an entire 3-week module on software testing and debugging.

Kendon and Stephenson [9] described a six-hour non-credit course that runs on a single day, which aims to better prepare incoming first-year students for introductory CS courses at the University of Calgary. In addition to providing hands-on instruction on the basic use of the command-line interface, this mini-course also covers other practical and soft skills. For example, it shows students how to write effective emails to teaching assistants or course instructors, and presents different valuable services that the university offers. In comparison, our course provides a broader, deeper exploration of software development tools and techniques.

3 DESIGN PRINCIPLES

At our school, student-led CS courses are always electives, usually have a low unit count, and only offer a Pass/No Pass grading option. These characteristics typically make them rather low-stakes, which means that students enrolling in such courses often do so out of genuine interest. Wanting to capitalize on this curiosity-driven motivation, we aimed to design our course to be as accessible, low-burden, and hands-on as possible.

First, we ensured that the course would be accessible to students right after the introductory sequence's first course (i.e., CS1), as our goal is for students to develop good software development habits early. Then, we opted for the class to be one unit, the lowest possible unit count, to guarantee that the workload would not interfere with students' other coursework; one unit represents 1 hour of weekly lecture and assumes up to 3 hours of additional weekly homework. Our course lectures are often a mix of slides, to present and discuss the topic(s) at hand, and demos, to show how the discussed concepts can be applied. For the most part (see details in next section), the course homework offers students hands-on experiences with the topics studied in lecture. The homework assignment for a particular week is made available right after the weekly lecture, and students are given one week to complete it on their own. To maximize the rate of homework completion, student submissions are autograded whenever possible, in which case students know their score immediately, even before the submission deadline.

Our school is on the quarter system, which makes for exactly 10 weeks of instruction. In order to balance breadth and depth, we decided to divide the course contents into three equally-sized 3-week modules, each of which dives into a set of related topics. We initially left the 10th and last week open to allow for unforeseen issues. The course topics were determined by three main factors: (1) Observations of the undergraduate curriculum from our own points of view as undergraduate students, (2) Experiences as peer tutors while assisting other undergraduate students, (3) Informal discussions with upper-division instructors whom we had heard complain about technical gaps they felt students exhibited. We quickly established that the central struggle point regarding computing ecosystem literacy was the students' general lack of familiarity with the command-line interface, which is why we designed our course with this aspect at its underlying core.

The first module focuses on the command-line interface (CLI) by introducing students to Unix-like environments and common utilities. Based on our observations, students are usually familiar with the idea of the terminal after CS1 and know very basic commands such as `cd` and `ls` but are often too intimidated to use the terminal regularly. The main goal of this module is to empower students to interact with Unix-like computer systems via the CLI. Topics range from basic (e.g., navigating the filesystem, interacting with files, using options to customize commands) to advanced (e.g., input and output redirections, composing complex commands using pipes).

The second module addresses software testing and debugging, as we noticed that students often struggle to detect and troubleshoot bugs in their code. As with CLI usage, they may receive exposure to the related concepts in their programming classes but may still lack the self-efficacy to apply them fully. For instance, if they are taught how to write unit tests in a prior course, they are not necessarily

told what constitutes a *good* test case. Similarly, most students know how to start a debugger, but they often lack the intuition of *where* to place a breakpoint. This module covers software testing, debugging methodologies, and presents a CLI-based debugger.

The third module is meant to further develop the students' software development practices by presenting scripting and showcasing a few convincing examples. We noticed that students were not always equipped to manage highly repetitive tasks, and their typical reluctance towards the CLI often led them to dismiss some of the most efficient tools available. This module presents a seemingly loosely-connected group of topics: shell scripting using `Bash`, regular expressions, and text processing using `sed` and `awk`. It is important to note that the module's goal is less to teach these specific tools/utilities than to illustrate how to solve certain complex problems using an adapted (CLI-based) approach.

As mentioned above, the 10th and last week of instruction was initially left open in case of unforeseen issues. Luckily, no major issues happened during the first offering of the class, so we decided to ask our first class of enrolled students which topic they would like to study during their last week. The two proposed options were to either extend one of the three existing modules (e.g., we suggested extending the second module with a topic on profiling) or introduce a fourth module about version control using `Git`, which has become an indispensable tool for software development. The majority of students chose the latter option.

4 COURSE CONTENTS

In this section, we describe the course implementation. There are ten pairs of lectures and associated homework assignments covering the four modules presented in the previous section.

4.1 Module 1: Unix-like command-line environments and tools

4.1.1 Week 1: Unix-like computer systems. Apart from the first ten minutes, which are devoted to explaining the logistics of the class by briefly going over the syllabus, the lecture is mostly meant to describe various, convincing motivations for using Unix-like environments, such as `GNU/Linux` based computer systems, and their associated command-line utilities. For example, we show how ubiquitous such systems have become in a vast range of devices, from gaming consoles, to smartphones, to IoT devices, etc. We also demonstrate how to perform a number of common software development practices in the terminal in a very efficient way: e.g., compiling and running C code, automatically formatting code to a given coding convention, installing software libraries using a package manager, etc. Finally, we point out that an understanding of these environments and tools is often a prerequisite to both academic coursework and industrial software development.

For the associated homework assignment, students are given two tutorials. The first tutorial describes how to remotely access the Ubuntu-based machines provided by our CS department's computing lab using SSH. The second tutorial provides students with instructions for installing and configuring an Ubuntu-based virtual machine on their computers. The goal of this assignment is twofold: (1) making sure that students have access to the environment required for this course, (2) encouraging them to start using a Unix-like system on their own computer more regularly. The autograded deliverables for this assignment are files produced by a couple of scripts, showing that students were able to run these scripts both on a lab machine and on their freshly installed virtual machine.

4.1.2 Week 2: Command-line interface basics. The lecture gives an overview of the command-interface interface (CLI) and its benefits. In this overview, we mostly discuss the interaction between the user and the filesystem. The lecture begins by explaining the structure of the filesystem in Unix-like systems and the concept of a file path (e.g., absolute vs. relative). We introduce a first set of utilities used to navigate the filesystem (e.g., `ls`, `cd`, `pwd`), also taking the opportunity to mention the concept of command-line options to customize the behavior of these utilities (e.g., `ls -a` to print hidden files). Next, we present a second set of utilities used to interact with files (e.g., `mv`, `cp`, `rm`), and introduce the concept of aliases to create useful shortcuts (e.g., `alias rm='rm -i'` to force a confirmation before deleting files). We also briefly discuss different ways to access files, either in the terminal directly (e.g., using `cat` or `less`) or via a text editor (e.g., using `Vim` or `Nano`). Finally, the lecture emphasizes that learning about the different utilities and their options can be self-driven, by using the system reference manuals (i.e., `man` pages) and with tools such as `tldr` that provide simplified help pages.

The homework assignment is an asynchronous quiz, counting about 15 questions. Most of the questions provide students with a plausible scenario related to using the CLI (e.g., assuming a given file tree and the user's current location in the tree), and ask them to solve a relatively easy problem which often involves using one of the CLI utilities presented in lecture (e.g., copying only updated files from one location to another). Many questions require students to use command-line options not presented in lecture, to encourage them to refer to the related man pages. The quiz has no time limit, and feedback is made available to students after the submission deadline.

4.1.3 Week 3: Advanced command-line interface. While week 2 limits itself to ordinary CLI interactions users could otherwise perform using a graphical user interface (GUI), week 3 mainly focuses on demonstrating more complex scenarios in which the CLI can exceed the GUI. The lecture introduces the notion of input and output streams and shows how commands' streams can be redirected from or to files, or redirected to other commands using pipes. We also present a few utilities often found in piped commands, such as `head`, `grep`, and `cut`. All of these concepts are illustrated via a thorough demo that shows how a CSV-formatted gradebook can, for example, be processed using a well-constructed piped command to determine the frequency of each assigned letter grade. We end the lecture by presenting a couple of administration-related topics, such as `sudo` (e.g., to run commands with superuser privileges) and the concept of job control (e.g., to run commands in the background or forcefully stop them).

The homework assignment puts students in a credible scenario for which using the CLI environment and tools is the most appropriate approach. We first provide them with a labeled data set of images from the Mars Curiosity rover mission [11]. They are then asked to write and submit a few "one-liners" (single piped commands), which are meant to process these images according to

some instructions. For example, the most complex one-liner that students are requested to write should copy all of the images labeled as "Martian horizon" (label number 9) to subdirectory `horizons`, and can be written as: `cat image_labels.txt | grep '^9' | cut -d ' ' -f2 | xargs -I % cp % horizons/`. This fully autograded homework solidifies the understanding and usage of piped commands and encourages students to discover additional useful utilities by themselves, such as `uniq` and `xargs`.

4.2 Module 2: Software testing and debugging

4.2.1 Week 4: Software testing. The lecture describes methods for evaluating the correctness of a program, which we define by its input-output behavior (i.e., any input gives the expected output). We first briefly mention the intuitive idea of manual testing, in which the programmer manually enters a set of inputs and visually verifies the corresponding outputs, but argue that thorough testing quickly becomes too time-consuming. The lecture then focuses on automated testing; that is, the idea of writing additional code along with a predefined set of inputs and expected outputs to automatically and efficiently test a program. The concept of automated testing is refined by presenting unit testing, which helps provide more details as to which component(s) in the tested program may be faulty. Finally, the lecture introduces code coverage, which determines the percentage of the tested program that is covered by a test suite.

The homework assignment trains students' software testing skills via two commonly encountered programming use-cases. First, students are provided with a precompiled program (as an object file) for which they know the interface (the public functions are documented in a header file) but do not have access to the code. This program contains a few bugs, which students have to discover by writing a test suite. This problem aims to teach students how to determine test cases solely based on a specification. Second, students are provided with the source code of an (assumed) bug-free program and are tasked to write a test suite that reaches 100% of code coverage. Through this problem, students learn how to account for all possible execution paths when writing test cases. The deliverables for this homework include both test suites and a report listing the bugs students found in the first program.

4.2.2 Week 5: Debugging methodologies. While debugging is recognized as a critical skill for software development, explicit strategies and skills are rarely taught [12]. Our lecture briefly presents the difficulties and pitfalls associated with debugging and details a four-step problem-solving framework to encourage students to be thoughtful and aware of their debugging practices. We adapted the work from Li et al.[10] and coined our framework *MINS*, an acronym where each letter represents a step within the debugging process. (1) Construct a **M**ental model of the faulty program. We present creating flow charts as a tool for serializing the understanding of a program. (2) **I**dentify the bug. We explain how to observe the program's inconsistency and determine its actual and intended behavior given a certain input. We suggest that students form "plain english" descriptions of this behavioral difference and find other inputs that create similar inconsistencies. (3) **N**arrow down the problem scope. We show that students can progressively hypothesize the bug's location until finding its exact location by posing answerable questions about the code and its behavioral inconsistencies. (4) **S**olve the bug per se. This entails generating a solution and verifying it. At the end, if the bug is still not resolved, the debugging process can be reapplied. While small specific examples illustrated each step of the presented MINS framework, the lecture ends with a more comprehensive demo.

The associated homework assignment asks students to apply MINS on two broken programs. First, based on the provided source code, they have to generate flow charts representing each program's intended behavior. Then, they are instructed to write "bug reports" that specify the intended behaviors and actual –erroneous– behaviors for each program, given certain inputs. In these reports, students should also try to generalize these differences. Finally, students can fix the bugs that they found in the broken programs. While the two first deliverables (i.e., the flowcharts and bug reports) are manually graded, the third deliverable (i.e., the fixed programs) is autograded.

4.2.3 Week 6: Using the GNU debugger. The lecture is a hands-on tutorial on the usage of the popular, terminal-based GNU Debugger (GDB). After a brief introduction of the tool, the lecture focuses on a couple of scenarios for which GDB can be helpful. For example students can use the command `backtrace` to investigate segmentation faults based on the detailed sequence of function calls. We also remind students that using assertions to check the argument passed to a function (such as `NULL` pointers in C) can help prevent unintended bugs. For behavioral bugs not resulting in segmentation faults, students are shown how and where to place breakpoints and how to inspect and print variables. The rest of the lecture presents many other useful GDB commands for carrying out actions such as controlling the execution of the debugged program, setting up watchpoints on specific variables, and printing the values of variables using different representations.

The homework assignment comes in the form of a complete code project that students need to debug. The project's codebase is meant to mimic an authentic project; the code is split over multiple files, a Makefile performs the compilation, and there is a test suite that uses sample data. The goal is to immerse students in a codebase that they have not written but need to quickly apprehend, as is often the case when working in industry. Students are invited to use GDB in order to find and fix four bugs, one of which is a segmentation fault, and all of which are revealed by the test suite. The expected deliverable is a fully functional codebase, which is verified by our autograder.

4.3 Module 3: Scripting

4.3.1 Week 7: Shell scripting. The lecture introduces students to shell scripting using `Bash`. We consider a realistic scenario in which a user wants to rename an extensive collection of JPEG images (from `IMGXXXX.jpg` to `imageXXXX.jpg`), and we discuss three incremental solutions. The first solution is for the user to manually rename each file individually (e.g., using `mv`); we argue that it is hardly a viable solution and that such repetitive tasks should be automated. We then present the basics of shell scripting, such as variables, string interpolation and substitution, wildcards, and for loops, which enables us to present a much better second solution based on a shell script. In the third and last step, we introduce more

shell scripting features, such as if and case statements, exit codes, functions, and command-line arguments, and present a complete, scalable third solution. This final script allows the user to specify both the `find` and `replace` patterns, the list of files to rename, and supports a couple of useful flags (e.g., `'-d'` to perform a dry-run that does not rename any files). We conclude the lecture by briefly mentioning `ShellCheck` [7], a static analysis tool that helps find potential shell script bugs.

The homework assignment addresses an engaging, real-life scenario. We provide students with a sample collection of music files[1], in which files are randomly named and encoded in various audio formats. Students are tasked to write a shell script that (1) re-encodes all of the music files in mp3 and (2) renames them according to the pattern `<artist>_<song>.mp3`. A typical solution script includes using most of the scripting features seen in lecture. It also requires students to use some external tools (from the FFmpeg project [2]) for the audio conversion and for extracting the artist and song names from the music files. We also ask students to run `ShellCheck` on their script and fix any reported errors. Student scripts are autograded by running them against additional collections of music files to ensure that the submitted scripts are generic.

4.3.2 Week 8: Regular expressions. The lecture introduces students to regular expressions, a transversal topic to many of the course's themes, as regular expressions are often used with many CLI utilities (e.g., `grep` or `find`) and are generally central to bash scripting as well as text processing. To illustrate this introduction, we focus on writing an email address validation script, a classic example of string validation. This example enables us to present many of the concepts involved in building simple but powerful regular expressions: wildcards, character classes, quantifiers, boolean OR'ing, etc. We also mention more advanced features, such as capture groups, to show students how to extract substrings from an input. The email address validator example is modified so that the three usual fields composing a valid email address (i.e., username, second-level domain, and top-level domain) can be extracted. The lecture ends by briefly discussing the most common flavors of regular expressions –Basic Regular Expressions, Extended Regular Expressions, and Perl-Compatible Regular Expressions– and how they can be used in various programming languages, from Python to C.

In the associated homework assignment, students are confronted with a realistic problem in which they are asked to complete a bash script that validates the billing information for all of the customers of a fictitious company. This billing information is composed of twelve different fields, such as names, addresses, credit card information, etc., which are commonly found in industrial customer databases, and provide students with the opportunity to write many regular expressions of various difficulties. Their submitted script is thoroughly evaluated via an autograder that counts a total of fifty different test cases.

4.3.3 Week 9: Text processing with `sed` *&* `awk`*.* The lecture gives an introduction to scripted text processing using the popular tools `sed` and `awk`. This introduction revolves around a realistic scenario in which a professor wants to manipulate a CSV-formatted gradebook. We show how `sed` can be used in lieu of `grep` to print selected entries, how it can insert new students in the gradebook or delete existing ones, and how it can replace fields within an entry. We then present a few of `awk`'s basic features, including variables, conditionals, for loops, and functions. We bring it all together with a simple `awk` script that can automatically compute each student's final grade, by dropping their lowest score from the gradebook and averaging their other scores. The lecture ends by comparing `grep`, `sed`, and `awk`, and the situations for which they each are best suited.

The homework assignment includes two problems, each illustrating a relevant use case for either `sed` or `awk`. The first problem is to write a bash script that modifies an `INI` configuration file using `sed` commands. The script can append, delete, or substitute fields in the configuration file. The second problem is to write an `awk` script that computes various tax information (e.g., average yearly earnings, highest monthly earnings, etc.) for each employee of a fictitious company, based on a CSV-formatted input file. All of the submitted scripts are autograded using a set of auto-generated data.

4.4 Module 4: Version control

4.4.1 Week 10: Using `Git` *for version control.* The lecture is a demo-based tutorial on the usage of `Git`, the most popular version control system. After briefly introducing the tool and its history, the lecture is split into three main sections. In the first section, we present the basic file tracking features that `Git` offers. We explain how to add new files to the repository, stage modified files and commit the modifications to the repository, view the history, etc. In the second section, we talk about the distributed model of `Git` and introduce the push/pull operations. We also mention conflict resolutions. The third and last section introduces branching. Small examples are shown to illustrate during each section of the tutorial.

For this homework assignment, students are provided with a git repository that contains a fictitious autograder across multiple branches. Students first need to understand the repository's structure, then merge all the branches into the main branch in the proper order –and solve merge conflicts, if any–, and finally complete the grading script in order to reconstitute a fully functional autograder. The submitted git repository is autograded on two different aspects; we check that the merge operations were successful by examining the repository's history and that the grading script was properly completed. Students are also required to write answers to various questions, which is meant to guide them through the different phases. Their written answers are manually graded.

5 COURSE OFFERINGS AND FEEDBACK

Our course was developed in spring and fall of 2020 and offered twice consecutively in the winter and spring quarters of 2021. Because of the pandemic, the course was offered online, mostly asynchronously. The lecture videos were uploaded to the course website, and students had continuous access to an online forum. The only synchronous part of the class were office hours.

5.1 Student enrollment

For both offerings of the course, the enrollment was limited to 50 students, which is typical of elective courses at our university. Although there was a short waiting list at the beginning of each quarter, the enrollment usually settled at around 45 students, indicating

[1] Found on the Free Music Archive (https://freemusicarchive.org/)

that the student demand was met. The exact class composition for each quarter is shown in Figure 1.

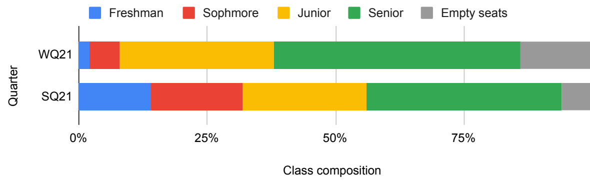

Figure 1: Class composition per course offering

The course was intrinsically designed to be accessible as soon as after CS1. We were surprised when the first offering counted an overwhelming 90% of upper-division students, including 56% of seniors alone. That said, it confirmed two of our initial assumptions which motivated the creation of this course: (1) students recognize the importance of computer ecosystem literacy for their professional success, especially as they are about to graduate, and (2) students feel like this literacy is not being sufficiently covered during their undergraduate education. Such a big share of upper-division students is also likely due to the fact that they have priority for registration.

To ensure that the second offering would be more balanced, we delayed enrollment until the open registration period, during which students of all levels have equal access to available courses. While seniors still held the most significant enrollment, the share of lower-division students dramatically increased (from 9% to 37%).

We are confident that as our course keeps being regularly offered, the proportion of lower-division students will steadily increase and soon become the majority, as we had initially expected.

5.2 Student feedback

At the end of each quarter, we asked students to complete an optional survey, to gather feedback about their experience. Unfortunately, the survey completion rate for the first class was relatively low, with only about 28% of students responding. For the second class, we gave a tiny amount of extra participation credits for completing the survey which boosted the completion rate to about 60%. All the results shown in this subsection represent the cumulative percentage across the two classes.

The main question that our survey aimed to investigate was whether students had increased their knowledge and self-efficacy in the topics covered by the course. We first asked them to rate how familiar with these topics they felt prior to taking our course on a 4-point Likert scale. As shown in Figure 2, the results unsurprisingly matched our original February 2020 survey, which prompted the creation of this course; for most topics, a majority of students reported having a low level of familiarity or no familiarity at all.

As shown in Figure 3, we then asked them to rate their perceived progress after having completed the course, also on a 4-point Likert scale. The vast majority of students reported a high or moderate level of progress across all the course topics.

In addition to evaluating students' overall progress in the course topics, we finally wanted to determine to what extent they left the course with a growth mindset. We therefore asked students if

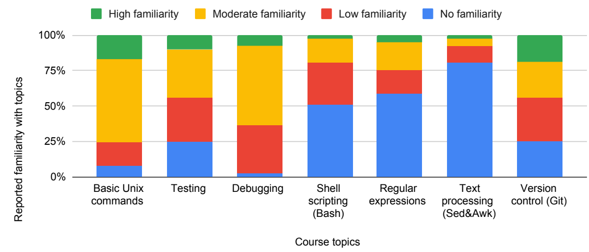

Figure 2: Student familiarity with course topics prior to taking class

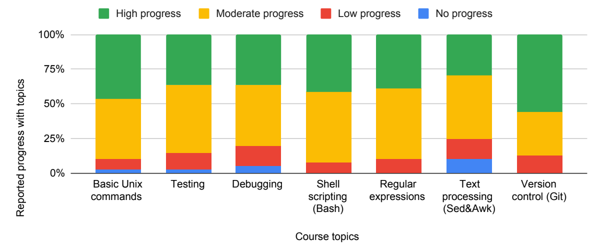

Figure 3: Student progress on course topics after taking class

they had incorporated some of these new software development practices into their workflow and if they had become more curious about CLI environments and tools. As shown in Figure 4, the vast majority of students reported that they had, for both metrics.

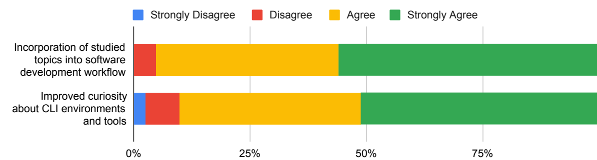

Figure 4: Growth mindset after taking class

6 CONCLUSION AND FUTURE WORK

In this paper, we described "The Missing CS Class," a student-led undergraduate course that covers topics aiming to increase students' computing ecosystem literacy and therefore helps reduce the academia-industry gap. We provided a detailed description of the course contents, which we hope others can adapt to their needs. The very positive feedback we received from students indicates that we accomplished our objectives of inspiring curiosity and encouraging independent learning.

Our long-term objective is now for this course to continue as a generational, student-led effort that can stand on its own after we graduate. We are currently looking into a rolling approach in which formerly enrolled students later become the ones teaching the course. Finally, to benefit the community at large, we have published all the lecture materials online at https://missing.cs.ucdavis.edu.

REFERENCES

[1] Anish Athalye, Jon Gjengset, and Jose Javier Gonzalez. January 2019-20. The Missing Semester of Your CS Education. Massachusetts Institute of Technology. Retrieved 2021/08/11 from https://missing.csail.mit.edu/

[2] Fabrice Bellard et al. 2000. FFmpeg. Retrieved 2021/08/11 from https://www.ffmpeg.org/

[3] Adam Blank, Josh Zimmerman, and Jake Zimmerman. Fall 2020. 07-131 – Great Practical Ideas in CS. Carnegie Mellon Unversity. Retrieved 2021/08/11 from https://www.cs.cmu.edu/~07131/f20/

[4] Michael Chang, Julie Zelenski, Chris Gregg, and Nick Troccoli. Spring 2021. Assignment 0: Intro to Unix and C. In *CS107: Computer Organization & Systems*. Stanford University. https://web.stanford.edu/class/archive/cs/cs107/cs107.1216/assign0/ (2021/08/11).

[5] CC2020 Task Force. 2020. *Computing Curricula 2020: Paradigms for Global Computing Education*. Association for Computing Machinery, New York, NY, USA.

[6] Paul Hilfinger. Spring 2021. Lab 6: Unit Testing and Integration Testing for Enigma. In *CS 61B: Data Structures*. University of California, Berkeley. Retrieved 2021/08/11 from https://inst.eecs.berkeley.edu/~cs61b/sp20/materials/lab/lab6/index.html

[7] Vidar Holen et al. 2013. ShellCheck - shell script analysis tool. Retrieved 2021/08/11 from https://www.shellcheck.net

[8] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, USA.

[9] Tyson Kendon and Ben Stephenson. 2016. Unix Literacy for First-Year Computer Science Students. In *Proceedings of the 21st Western Canadian Conference on Computing Education* (Kamloops, BC, Canada) *(WCCCE '16)*. Association for Computing Machinery, New York, NY, USA, Article 14, 4 pages. https://doi.org/10.1145/2910925.2910930

[10] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2019. *Towards a Framework for Teaching Debugging*. Association for Computing Machinery, New York, NY, USA, 79–86. https://doi.org/10.1145/3286960.3286970

[11] Steven Lu and Kiri L. Wagstaff. 2020. *MSL Curiosity Rover Images with Science and Engineering Classes*. https://doi.org/10.5281/zenodo.3892024

[12] Renée Mccauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: A review of the literature from an educational perspective. *Computer Science Education* 18 (06 2008). https://doi.org/10.1080/08993400802114581

[13] Nick Parlante. Fall 2007-08. Unit Testing. In *CS108: Object Oriented Programming*. Stanford University. https://web.stanford.edu/class/archive/cs/cs108/cs108.1082/handouts/06UnitTesting.pdf (2021/08/11).

[14] Alex Radermacher and Gursimran Walia. 2013. Gaps between Industry Expectations and the Abilities of Graduates. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 525–530. https://doi.org/10.1145/2445196.2445351

[15] Alex Radermacher, Gursimran Walia, and Dean Knudson. 2014. Investigating the Skill Gap between Graduating Students and Industry Expectations. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 291–300. https://doi.org/10.1145/2591062.2591159

[16] Alex D. Radermacher. 2012. *Evaluating the gap between the skills and abilities of senior undergraduate computer science students and the expectations of industry*. Ph. D. Dissertation. https://www.proquest.com/dissertations-theses/evaluating-gap-between-skills-abilities-senior/docview/1018399283/se-2?accountid=14505 Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2021-05-25.

[17] Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. 2004. Self-Efficacy and Mental Models in Learning to Program. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) *(ITiCSE '04)*. Association for Computing Machinery, New York, NY, USA, 171–175. https://doi.org/10.1145/1007996.1008042

[18] Nicholas Weaver. Spring 2019. Lab 0: Setup. In *CS 61C: Great Ideas in Computer Architecture (Machine Structures)*. Stanford University. https://inst.eecs.berkeley.edu/~cs61c/sp19/labs/lab0.pdf (2021/08/11).