Hiring Millennial Students as Software Engineers

A Study in Developing Self-Confidence and Marketable Skills

Scott Heggen
Berea College
Berea, Kentucky
heggens@berea.edu

Cody Myers
Berea College
Berea, Kentucky
myersco@berea.edu

ABSTRACT

Software engineering courses and internships aim to equip students with skills that are vital in the software engineering industry. Millennial students are expected to emerge from an undergraduate education ready to step directly into software developer positions and succeed. And yet, for a variety of reasons, these experiences often fail to adequately prepare students for the ambiguity of industry. The capabilities of a typical undergraduate simply do not align well with the expectations of the industry, resulting in disappointed employers, unhappy employees, and a poor reputation for the quality of a higher education in software engineering. This paper describes the Student Software Developers Program, where students are developing real-world applications that solve business needs at various departments on campus, leveraging those departments as customers. Students are immersed in the program for a full year, providing them with adequate time to experience both the depth and breadth of skills desired by the curriculum and by industry. Our evaluation shows the program provided students with confidence in their engineering abilities, a wealth of hard and soft skills valuable in industry, all while learning software engineering in a way that aligns with the values of their generation.

CCS CONCEPTS

• **Social and professional topics → Software engineering education**; • **Software and its engineering → Designing software**; *Software development process management*; Programming teams;

KEYWORDS

Software Engineering Education, Agile, Software Development

ACM Reference Format:
Scott Heggen and Cody Myers. 2018. Hiring Millennial Students as Software Engineers: A Study in Developing Self-Confidence and Marketable Skills. In *Proceedings of ACM Software Engineering Education for Millennials (SEEM 2018).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Undergraduate software engineering courses often have high expectations of what they want to accomplish. Typically, the development of a software project is expected. In addition, instructors want students to experience what it's like working in a software engineering team, following software engineering best practices, and developing good documentation and coding practices. Depending on the project, students may also need to learn supporting skills (i.e., marketable skills) like version control, new programming languages, and new libraries or APIs. Lastly, software engineering courses are also a useful place in a curriculum to teach soft skills, such as communication within a team, communication with a customer, requirements gathering, and resolving conflicts [21]. For a single course in a college or university setting, getting students to deeply engage in all of these subjects at a rigorous level is challenging, if not impossible.

An instructor also has the difficult task of choosing projects. Letting the students pick their own project gives them a great sense of ownership and pride in their product, but the projects are usually poorly defined, have no real-world stake, no customer to provide requirements or feedback, or are scaled back significantly to be completed in the allotted time. Working with outside businesses or non-profit organizations provides real stake in the project, as a company is depending on the students to complete the project, but comes with the additional challenges of coordinating with the representatives of the company, gathering requirements in a foreign context, and facing challenges with customers who have little or no technical expertise. These projects often end with a rough prototype, but not a real final product that is usable by the customer, leaving them unsatisfied. Other courses may leverage open source projects, which typically have real-world stake, low negative impact if they are not successful, and rigorous requirements for the students to follow. However, these projects are often supported by volunteer developers in the open source community who support the software as a hobby, making them poor collaborators or customers. Students can also select issues and features beyond their capabilities. The projects can have difficult or inaccurate installation and usage instructions. Worst of all, the community around the software can be an unwelcoming or even toxic on-line environment for novice programmers, driving away students with less self-confidence. Selecting open source projects requires caution taken by the instructor. In short, all software engineering projects come with pros and cons, and constraints that limit the course to a subset of the learning goals desired by the instructor.

Shaw [20] indicates three challenges in the design of a software engineering curriculum. First, there must be well-defined roles for all members of the team, similar to how roles are separated in the software engineering industry. As Shaw notes, "Available knowledge about software development far exceeds what any one person can know." The result is depth in a single software engineering role,

SEEM 2018, June 2018, Gothenburg, Sweden

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

but a deficiency in other roles. Second, Shaw claims that software education needs to be a long-term investment. A single course is simply not enough time to fully engage in software engineering. Third, Shaw notes that specialization training to develop mastery in a subject area within software engineering results in opportunities for the students. Again, specialized training takes time. Industry often takes on this burden by training new employees in their first few weeks, due to academia failing to provide them with the specific, necessary skills [4].

Sometimes software engineering learning is pushed outside the classroom, such as through internships, research experiences for undergraduates (REU's), or co-ops. While some of these experiences are stellar for students, others return to campus with mixed reviews. Some feel defeated or overwhelmed by the work they were assigned, while others may feel underutilized by the organization. Some have bad experiences because of inexperienced supervisors. Other students aren't able to secure an internship at all, while others can't afford to not make money or move to a new city, and end up wasting valuable learning time during a summer working jobs to make ends meet. In all, internships are a gamble for students, and are not a reliable source for gaining skills that align with a computer science or software engineering program's curriculum.

This paper describes a program which blends the best parts of a software engineering course and an external experience like internships. Through our work-study program, students are employed by the computer science department to develop software for a full year. Students start in the summer, working approximately 400 hours under the supervision of a faculty member, where they are trained on the software engineering principles described above. In addition, they begin developing their first software system. The projects are proposed by the campus community (a.k.a. the customer) and are selected by the supervising faculty member. These projects are often tools requested by the customer to help them complete their daily work, thus, the software becomes an integral part of the customer's job. Students finish the beta version of the product by the end of summer. During the remainder of the academic year, the students work an additional 300 or more hours, where they maintain the software, including implementing new feature requests, fixing bugs, and performing customer support.

The remainder of this paper is organized as follows: the Related Work section describes research that influenced our program. Next, the software projects and the process created by students in the program are discussed. Next, an evaluation of the program is presented in the Results section. We conclude with future work and closing thoughts on the effectiveness of the program.

2 RELATED WORK

Interacting with millennials, those born after 1980 [14], has challenged instructors, particularly those from other generations, to adapt their teaching styles. According to Partridge et al. [15], the millennial generation can be characterized by the following traits: they place value on building strong relationships; prefer close-knit communities; are socially-responsible and aware; view their parents as friends more than authority figures; enjoy teamwork; bring a mentality of "leave no one behind"; are pragmatic but optimistic; and probably most relevant, they are a generation that has always had digital technology as part of their lives, and see it as an essential part of their daily living. The millennial generation is also characterized as a protected generation, who are used to structure in every aspect of their lives. They are not known for handling ambiguity well, lean heavily on mentors, and are considered materialistically spoiled, particularly with regard to technology. Boredom is their worst fear.

When you imagine your typical course in college, you might envision an instructor lecturing to a class; not exactly your most engaging, active environment. In fact, Freeman et al. [7] showed that lecturing results in 1.5x more students failing a course than through the use of active learning. This is not the only difference an instructor might notice as they interact with millennial students. Partridge et al. [15] notes that millennials learn best when they are recognized as individuals; have a voice in class decisions; are able to establish a rapport with instructors; have group interactions; use class time as a social and educational experience; have a non-judgmental sounding board; are not passive recipients of information; are engaged to retain information; are given a variety of learning experiences; are given course work that is relevant to the real world; are learning marketable skills; and are engaged with current information.

In software engineering courses specifically, educators who are keyed into understanding how millennials learn have discovered that there are better ways of teaching than purely lecturing. Razmov and Anderson [17] incorporated modern technology such as tablets into the delivery of course materials and found that it resonated well with students who are already very comfortable with technology. Gannod et al. [8] followed a flipped classroom approach, driving the theoretical learning out of the classroom and focusing that time on active development of a project. Liu's [12] approach engaged students with real-world projects through service-learning and contributions to open source projects. Numerous other examples exist of active learning being used to improve the way millennial students are learning software engineering beyond the traditional lecture.

Although curricula are becoming more engaging, software engineering as a discipline is broad and requires many skills. According to Shaw [20], "...design, management, programming, validation, analysis, user studies, documentation, system integration, and property-specific techniques such as design for security and reliability" are all key principles in software engineering. However, it is inconceivable to expect students to develop all of these skills within a single course. Therefore, the curriculum relies on teaching "...essential, durable content that will serve the student for several decades" [20], meaning scalable knowledge will consistently be favored over immediate learning such as current industry standards (i.e., marketable skills). As a result, marketable skills such as Agile [3], lean in software development [16], and version control tools, to name a few, are omitted from the course, and are left in the hands of the students to learn through their own curiosity. Shaw [20] says it best when she states, "Curriculum design is at heart a resource allocation problem, with curriculum space (as measured by courses, hours of study, number of homework problems and projects, etc.) as the scarce resource."

Naturally, students seek out these marketable skills outside of the classroom, such as through internships. As Schambach and Dirks

note [18], internships can provide numerous potential benefits to students, such as reinforcing conceptual learning; establishing self-confidence; an increase in marketable job skills; improving their ability to problem solve in an environment where defining the problem is an essential part of the job; revealing the importance of soft skills such as social interactions; effective communication; and teamwork. Because internships are competitive, students that are further in their education are more likely to get these experiences. Furthermore, the majority of internships are offered during the summer, meaning these experiences are typically short. Tvedt et al. [23] summarize the major pitfall of internships well when they say, "[An internship] does not provide the depth of experience to appreciate the responsibilities of the roles and the implications of their decisions on future development. Nor, do the students have the opportunity to learn from their mistakes and apply their experience to future projects."

Both Martincic [13] and Keogh et al. [9] present multi-semester experiences, blending software engineering courses with industry-sponsored internships. Their studies showed deep software engineering and soft skill development by the students. However, the logistics of such programs proved one of the more challenging aspects of these programs, particularly in coordinating with multiple industry collaborators. Our Student Software Developers Program most closely aligns with these two works, with a few exceptions which we will cover in the next section.

3 METHODOLOGY

This section describes the Student Software Development Program. First, the software projects and their unique purpose are described. Then, the structure of the program, framed around the software engineering learning goals, are presented.

3.1 Software Projects

The software projects developed by our students are all custom web applications that provide technical solutions to problems unique to our institution. For example, at our institution every student must participate in a work-study program known as the Labor Program, where they are employed by the college for 10 to 15 hours per week to do the work of the college. Students get jobs in every department on campus, from maintaining the grounds to writing software, such as the students who are participating in the Student Software Development Program. To hire all of these students, the labor department was using an antiquated paper-based process, typically requiring students to walk around campus obtaining signatures from multiple supervisors and academic advisors. The Student Software Developers team created a custom web application to replace this process, eliminating time wasted by students chasing down signatures from labor supervisors, significantly reducing rework by eliminating errors from the paper-based form, and removing paper waste from the campus. The software is used by every labor supervisor (nearly every staff and faculty member) to hire new students, as well as manage many other aspects of their labor position (such as releasing students from their position or making corrections). Our software is fully in production and is the primary tool supervisors use for managing their student labor.

All of our application requests follow a similar process: a department on campus identifies an inefficient process within their department, they discuss their challenges with the student software developers, and determine if custom software will reduce or eliminate that burden. The student developers then design, develop, and deploy the application for live use. We also manage the software for as long as it is being used, providing technical support and continually upgrading the software as new features are requested, bugs in the software are discovered, and new processes are implemented.

The program began 3 1/2 years ago, and we have since worked on nine other systems like the one described above. Space limitations prevent describing all ten systems. However, we briefly describe the Course Administration and Scheduling (CAS) system to demonstrate one more application which has a significant impact on the campus community.

3.1.1 The Course Administration and Scheduling (CAS) System. At roughly 1600 students, our institution runs about 550 courses per term. The scheduling of classes and selection of rooms has been a challenge for the Registrar. Using the old process, the Registrar's Office would email all department chairs requesting schedules with instructor, date, and time information entered into a spreadsheet. Each department chair would enter their courses into the spreadsheet then send it back to the Registrar's Office, who would transfer each department's spreadsheet into a master spreadsheet. This process would repeat each week as the master schedule changed, allowing department chairs to identify conflicts in the schedule (i.e., conflicts between two required classes in a major, but in different departments, such as a math requirement for a computer science major). Once the master schedule is set, the process repeats for room assignments. Conflicts arose when multiple faculty needed a particular room, and the Registrar's Office would make the changes and resend the spreadsheet to the community for further feedback.

The old process took roughly 70 hours for the Registrar's Office staff to do data manipulation, plus over four weeks of back-and-forth emailing of spreadsheets between department chairs and the Registrar. Worse yet, this process resulted in numerous errors as spreadsheets were lost, data entry or data migrations were done incorrectly, spreadsheet versions became out of order, emails weren't sent in a timely manner, and a multitude of other potential pitfalls.

The Registrar's Office approached the team about developing software to aid in solving the aforementioned challenges. The first version of the software was developed in a summer by two students (one sophomore and one junior), and deployed in the following term as a beta product. The original software provided an interface for department chairs to enter course information. All members of the faculty were able to view the entire schedule as it was being built, allowing department chairs greater visibility into the schedules of other departments. The improved visibility reduced the number of scheduling errors and fixes, as each department was responsible for their own data entry. The system improved the collaboration between departments in scheduling courses that should not run at the same time. Furthermore, it exported cleanly into the college's system for managing the official schedule (this tool was intended as a planning tool only). The new process eliminated the 70 hours required by the Registrar's Office staff in data manipulation entirely. While department chairs did not report a savings in time

on task, they reported clearer visibility in the schedule as it was being developed. Most importantly, it eliminated nearly all errors in scheduling caused by the Registrar's Office.

Subsequent versions of the CAS software added new features which have further improved the utility of the system. The system is now capable of assisting the Registrar in scheduling rooms, as it clearly highlights conflicts for each room. The Registrar can also manage terms, manage users, and manage new course catalog entries and special topics courses (student programmers were directly inserting them into the database in earlier versions).

3.2 Software Engineering Process and Learning Goals

Part of the success of Martincic [13] and Keogh et al. [9] was the immersive nature of their programs; students actively participated in software engineering for more than a single semester. The Student Software Developers Program takes a similar approach, but breaks their time into two phases: the summer term, where the students work 40 hours per week for 10 weeks, and the academic term, where the students work 10 to 15 hours per week for the entire academic year. This provides the students with a minimum of 700 hours over the course of the year engaging with software engineering principles. Through the college's Labor Program, the students are paid an hourly rate for their time.

The summer term is dedicated to the design, prototyping, testing, and deploying of the first version of a system. Using Scrum principles [19], the team breaks down the summer into a series of weekly sprints. The students rotate roles (e.g., scrum master, database engineer, front-end developer, etc.), allowing them to experience multiple software engineering roles. As they begin coding, the students practice pair programming [24] to meet the goals of each sprint, giving them the opportunity to develop good team practices and leverage each other for questions as problems arise. Lastly, we employ just-in-time teaching to give the students the skills they need at the moment they need it. For example, when the students are beginning to think about the interactions between their front-end user interface, the back-end database, and the controller that connects them, we launch into a discussion about Model-View-Controller (MVC) [10].

The next four sections describe the different phases of the Student Software Developers Program. While we describe the phases linearly and rigidly, they follow an Agile [11, 19] mentality, where the students will rotate through each phase in short, typically one week cycles, and naturally flow from one phase to another.

3.2.1 Initial Design Sprint. The process typically begins with students meeting with their customer to gather requirements. The students then engage in design thinking exercises. For example, they map the customer's requests to a set of features, by creating a software requirements specification document and use case diagrams [6]. From these documents, the students can begin thinking about the main user interfaces in the system, which is then paper prototyped [22] in small teams. The paper prototypes are reviewed by the entire team, extracting the best features from each group into a final prototype design. The team then begins to think about the data models that support their user interface, and build the entity relationship models. Much of the first week in the design of a system is not spent coding, but thinking about the system from a user's perspective, and determining the best way to implement it.

The team, led by the scrum master for that sprint, have daily stand-up meetings for design reviews. In each meeting, the students review each others' most recent work. By actively questioning each other's design at each stage of the process, the students learn to work as a team to identify the problems quickly, modify the design early (before too much code has been written), and resolve design flaws before they reach the customer. When a sprint comes to an end, a new scrum master is chosen, and the students switch pair programming partners. Guaranteeing that everyone's voices and ideas are incorporated into the design allows for each student to gain ownership of the project and see the big picture goals. Students see early on that the system will constantly change, helping them realize that software engineering isn't just about defining a product then building it. Software engineering is much more organic, allowing the system to evolve as design decisions are made.

3.2.2 Implementation Sprint. As implementation begins, the students break down the design into issues inside of an issue tracker. The team meets, creates a sprint list, and decide on the most important features to work on for that sprint. The pair programming teams then create self-imposed deadlines estimating their development time, which helps them track their progress and learn to estimate development time. Each workday during the sprint begins with the scrum master asking three questions: what did you accomplish yesterday, what do you plan to accomplish today, and what issues are you stuck on? These three questions create a culture within the team that constant progress is expected, getting stuck on a single problem for multiple days is discouraged, don't feel inferior for not being able to do something, and there's no shame in asking for help. Instead, getting stuck and asking for help early is actively encouraged, and results in better productivity and learning overall.

The students follow git workflow [1] to create pull requests by the end of each sprint. During the sprint review, the team will go through each pull request, showing what was accomplished and performing a team code review before merging code into a "development" branch. After all of the pull requests have been merged into development by the scrum master, the students spend a day on team testing, where the students try to break each other's code. This testing phase emphasizes the importance of good coding practices, such as input validation and sanitizing. After the team approves the changes, the development branch is merged into the master branch, thus ending the implementation sprint. The code is then pulled into the production environment and considered live.

3.2.3 Design Checks. During implementation, any member of the team can request a design check. These checks provide an opportunity for the team to step back and ensure that their current design is still valid and meets the customer's needs. The question is either answered by the team, or the design is modified. For example, students might discover the database needs restructuring to handle relationships correctly. The team discusses, and if they agree on a modification to the design, it is appended to the current implementation sprint (if critical), or it is prioritized in a future sprint.

3.2.4 Software Maintenance & Upgrades. During the academic terms, our work-study program requires students to work for 10 to 15 hours per week in addition to their regular course load. The new schedule shifts our focus from a development mindset to a maintenance mindset. Duties include bug fixes, new minor features, and general customer support. The benefit is students experience the entire life-cycle of software development, particularly the later stages of maintenance and upgrades, which students rarely see in courses and internships (or worse, it's the only thing they see in their internship). Through working customer support, the students learn how to debug those unique, fringe issues that only occur when a system is in production. Additionally, having to develop new features to a live system reveals the importance of a scalable design. The value of good coding practices becomes apparent; they are often looking at their peer's code. If the team's self-imposed coding standards are not followed, or the software was not designed in a manner that makes it easy for new upgrades, their own workload and frustration levels increase. A culture of properly structuring code, providing good documentation, commenting, and testing emerges, as students don't want to spend their precious hours debugging code because of poor implementations.

4 RESULTS

To evaluate the effectiveness of the Student Software Developers Program, we issued a survey to all past and present students in the program. The program has had a total of 22 participants; 16 completed the survey (73%). The newest students in the program started in 2017, approximately nine months before the survey was issued. The amount of time students spend in the program varies from a single summer (originally, we allowed this; now, all students are expected to work for a full year) to the most senior student being in the program for 3 years now. All data was collected in January 2018. Table 1 summarizes the demographics of the 16 participants in the study. All participants fall in the millennial age range, being born after 1980.

Table 1: Demographics of the 16 students participating in the Student Software Developers Program.

Demographic	% of Participants
Male	68.8%
Female	31.3%
White	43.8%
Asian	18.8%
Black	25.0%
Hispanic	12.5%

The survey aimed at capturing three metrics from the students: their development of computer science and professional skills before and after the program; their self-confidence as computer scientists; and their perceptions of the program and how well its structure compares to the values of the millennial generation.

4.1 Software Engineering Skills

First, the survey asked the students to rate their abilities across 15 software engineering skills, summarized in Table 2. Based on Bandura [2], a 0 to 100-point scale was used for self-reporting of skills because it is a stronger predictor of performance, compared to a typical Likert 5-point or 6-point scale. The students were asked to rate themselves based on their self-perception prior to participating in the program. The students were then asked the same set of questions, but in regard to their skills after having participated in the program (or for those still in the program, rate their current perception of their own skills). Their ratings of prior to and during/after the program were not viewable at the same time as they completed the survey.

As Table 2 shows, the results are extremely positive. A two-sample paired t-test (alpha = 0.05) was performed on the data, and on every skill, they rated their skills significantly higher during and after the program compared to before joining the program. The highest p-value reported by the students was for testing software, but is still well within the alpha of 0.05. Our interpretation of these results is rather straightforward: the students are gaining all of the software engineering skills set forth by the program, and with great success.

4.2 Marketable Hard and Soft Skills

The survey also asked students to report the top five hard skills they learned in the program, and similar for soft skills. For hard skills, their responses fell into three categories: Software engineering-specific skills (e.g., requirements gathering, debugging, git, scrum, agile, etc.); programming and markup languages (e.g., Python, C, HTML, Javascript, SQL, etc.); and frameworks and APIs (e.g., Flask, ASP.NET, Ruby on Rails, Ansible, etc.). Table 3 summarizes their responses. In all, the students reported 15 unique software engineering-specific skills, and these skills were mentioned 30 times total. For programming and markup languages, the students reported 10 unique languages, and these languages were mentioned 26 times total. Lastly, 9 frameworks and APIs were reported, appearing 19 times in the students' responses. The breadth of hard skills mentioned by the students (34 unique skills) coupled with the average number of responses per skill (between 2.0 and 2.6 for all three categories) indicates that the students were both learning lots of skills, and multiple students were learning the same set of skills (i.e., students were learning about multiple software engineering roles). Git (12 mentions), Javascript (9 mentions), and Flask (6 mentions) were the three most mentioned hard skills learned.

For soft skills, their responses fell into four categories: project management (e.g., writing issue queues, designing, time management, etc.), teamwork and communication, professionalism (e.g., leadership, independent learning, working with ambiguity, seeing the big picture, etc.), and client communication. Table 4 summarizes the four categories and the student responses. Overall, the students identified learning about 26 unique soft skills, with significant overlap within each category. For instance, every student mentioned teamwork and communication as a soft skill learned in the program, as well as at least one project management skill.

Table 2: Summary of software engineering skills before and during/after participation. Each value is the average of all responses by students, on a scale of 0 (did not believe they could do at all) to 100 (had absolute confidence they could do).

Software Engineering Skill	Before the Program		During/After the Program		p-value
	Avg.	St.D.	Avg.	St.D	
Design a new software system:	38.1	25.6	88.9	11.8	5e-8
Create a new software from scratch	27.0	24.0	86.1	13.4	1e-9
Communicating with a customer:	47.2	26.3	92.5	9.3	4e-7
Capture a customer's requirements:	56.9	30.3	93.1	7.9	7e-5
Document software requirements:	47.2	28.5	91.9	8.5	1e-6
Document a software's structure with diagrams:	47.2	26.6	94.0	6.9	1e-7
Design software in a team:	53.8	16.0	93.2	10.0	2e-9
Develop software in a team:	51.6	16.0	92.8	11.9	7e-9
Test software:	43.1	25.7	79.6	20.7	1e-4
Fix a bug in a software system:	51.6	27.1	92.8	7.9	2e-6
Use version control, such as git:	35.6	38.4	92.3	12.8	4e-6
Push changes to a production environment:	29.9	34.4	86.9	18.5	2e-6
Lead a scrum meeting:	23.8	29.0	81.5	23.8	9e-7
Prioritize tasks related to software:	46.6	26.0	91.1	15.7	2e-6
Lead a team of software developers:	34.4	33.7	87.6	15.4	3e-6

Table 3: Summary of top hard skills categories learned as reported by the students.

Hard Skill	# unique resp.	# total resp.	Avg. resp. per skill
Software eng.-specific skills	15	30	2.0
Programming/markup lang.	10	26	2.6
Frameworks & APIs	9	19	2.1

Table 4: Summary of top soft skills categories learned as reported by the students

Soft Skill	# Resp.
Project Management	21
Teamwork and Communication	21
Professionalism	19
Client Communication	13

4.3 Confidence

The participants were then asked to respond to several different questions about the program's impact on their confidence. Confidence in themselves as computer scientists is a salient metric because it affects a student's "…motivation, interest, and achievement towards specific tasks" [5]. Therefore, we first measured confidence by asking the students to self-report how they felt prior to joining the program. The question was intentionally made open-ended so they had the freedom to answer in a way that suits their experience, without leading them to a positive (e.g., "I felt confident") or a negative (e.g., "I felt nervous") response. Themes emerged as we parsed their responses. First, the results revealed that 75% (12/16) of the students felt they lacked the experience needed to develop software systems prior to joining the program. Additionally, 63% (10/16) of students were nervous about whether or not their current skill level would allow them to make a meaningful contribution. Conversely, only 25% (4/16) of the students indicated that they were optimistic in their own abilities to develop software systems.

We also explicitly asked the students to compare themselves to their peers to see if they felt they had more, less, or equivalent software engineering experience than their peers in the program with them. This metric revealed that 69% (11/16) of our students felt they had less experience, 18% (3/16) percent felt that had an equivalent experience, and only 6% (1/16) felt that had more experience than their peers. One student reported they were unaware of their peer's skills, which resulted in the student feeling unqualified to answer the question. This clearly demonstrated that the majority of students began the program with a lack of self-confidence in their technical abilities.

To evaluate their confidence during and after participating in the program, we look back at Table 2. The students indicated they were least confident in their ability to perform testing at an average rating of 79.6. However, 79.6 on a 100-point scale is by no means a low rating in the authors' opinions, and is statistically significantly higher than their rating of 43.1 prior to joining the program ($p = .0001$). Similarly, the second lowest skill is leading a scrum meeting, rated on average at 81.5 during and after the program. Compared to the 23.8 rating prior to the program ($p = .0000009$), we are comfortable in saying that students are far more confident in their leadership ability after participating in the program. The 13 other skills from Table 2 follow a similar trend. The students feel significantly more confident in their software engineering capabilities after participation in the program.

Digging deeper into the students' comments, we see some quotes that also indicate effectiveness in impacting confidence. For example, one student stated that prior to the program, "I was super nervous because I felt like I wasn't ready [for the program]. I had

taken only three [computer science] classes and I wasn't certain what I was doing. I felt I wasn't as qualified compared to my peers. I always had an issue with doubting myself so I wasn't surprised that I felt this way." However, after the student had participated in the program for two years, we asked if the program helped prepare them for their current job in industry. The student responded with, "[The program] was my first actual software development experience. This was my stepping stone to get other internships and helped me a lot with my resume building. I am very grateful that [I was given] this opportunity because I am certain this was a huge reason I decided on this career path and the reason I was able to get a job at Wall Street. I can explain how big of an impact it had on my self esteem, academic life, and life in general."

Another student admitted that, "…at the beginning I was feeling a bit intimidated by my peers, and I thought my skills weren't as advanced as theirs. Being a freshman who has never had any real-life experience of that kind I thought I am behind the rest…I was a bit nervous about whether I'll be able to learn all that I need to learn, and whether or not I'll be a valuable member of the team." After two years in the program, the student shared that, "I love every part of this program. It was an extraordinary experience for me, and it was one of the most important things in my life. It helped me in many ways, and I think many things in my life wouldn't be possible without it…[The program] helped me get over the fear that I won't be able to learn, or I won't be good enough. Right now I feel prepared to take on any challenge they give me, believing in myself and both my technical and non-technical skills."

There are many more comments that reveal how the program has impacted the students' confidence. For the sake of space, here are just a few: "Working in the program had given me enormous experience, and stories to talk about confidently in job interview" and "working in the program helped me be more comfortable with working in a group, sharing my ideas, and not being afraid to even suggest that someone might be wrong when I feel that they are. I fully credit [the program] for my current [career] and for placing me in an advantageous position in the pursuit of lifelong learning."

4.4 Millennial Values

We were also interested in how well the Student Software Developers Program aligned with the values of the millennial generation. Table 5 shows the eleven questions related to millennial values, which are based on the list in Section 2 as defined by Partridge et al. [15]. A high response (i.e., closest to 100) correlates to a value held by millennials, while a low response (i.e., closest to 0) correlates to a value not held by millennials.

The student responses were very promising; on average, students rated above 85 out of 100 on every question, indicating the design of the program aligned well with the way the students learned best. Out of the 11 questions, six questions rated above 90. The program is most adept at providing the students with work that is relevant to the real world (rated 96.67). The other five metrics over 90 worth mentioning: the program provides opportunity to establish a rapport with their mentor (rated 93.1); the work was engaging (rated 92.8); they got a variety of learning experiences (92.5); and they learned marketable skill (91.9). Interestingly, the students said they developed a rapport with their mentor(s) (93.1) almost 5 points higher than they reported developing a rapport with each other (88.3).

Table 5: Results of how the students felt during their time as student programmers. Responses were on a scale of 0 (absolutely not true) to 100 (absolutely true).

During your time as a student programmer, did you feel you:	Avg. Resp.	St. Dev
were recognized as an individual.	88.6	16.1
had voice in decisions.	89.1	17.0
established a rapport with your mentor(s).	93.1	15.0
established a rapport with your peers.	88.3	17.9
had group interactions.	85.6	22.1
were engaged in your work.	92.8	11.4
got a variety of learning experiences.	92.5	9.5
did work that was relevant to the real world.	96.6	6.2
did work that was relevant to yourself.	91.6	12.1
learned marketable skills.	91.9	14.4
gained information that was current.	89.6	13.8

Overall, our interpretation of these results is that students feel the Student Software Developers Program is providing them with an experience that engages them in a way that aligns closely with the values of the millennial generation.

4.5 Community Impact of the Software

While our primary goal of the Student Software Developers Program is to provide students with an immersive, high impact program for growing as software engineers, a secondary goal of the program is to have an impact on the community through the software we develop. In the 3 1/2 years the program has been running, we have deployed and are maintaining eight applications, with two more in development.

All of the systems are designed as tools to allow departments to do their work, so we are most interested in estimating how much time the software has saved the customers. To extract time savings, we interviewed our customers asking them to report the number of hours the new process has saved them. The sum of these time savings translates to a conservative estimate of 2700 hours per term of faculty, staff, and student time that is no longer spent on inefficient, error prone, and antiquated processes. Put another way, assuming the average salary of all faculty and staff and students (who are all paid at our institution through our work study program) is $25.00 an hour, the college is saving roughly $135,000 per year because of our software. Both the estimate of hours and assumed average salary are very conservative here; the actual cost savings is certainly higher. However, we aren't interested in exact values in this work; these figures are presented merely give a sense of the impact our systems are having on our campus community.

5 FUTURE WORK

The Student Software Developers Program is still relatively new. Because we understand and value lean principles [16], we know the program will continuously improve with each new student cohort.

The students are encouraged to be actively involved in understanding the larger process, and to have a voice in making decisions about changes to the workflow. One example of this is how the team was using git to manage versioning. The students were creating merge conflicts due to poor branching practices. They researched and implemented a new process to solve these merge issues, reducing the number of merge conflicts dramatically. By empowering the students to make changes to process and giving them a voice in the direction of the program, we've created an environment where the students are deeply invested in the success of the program. Since the program's inception, our processes have changed dramatically, and will continue to do so because the students are given the tools to identify their own inefficiencies and devise solutions to their problems.

6 CONCLUSION

The Student Software Developers Program has provided students with a new way of engaging with software engineering. Students spend a minimum of one year in a work-study position developing software that serves the needs of the campus community. The software serves the community by providing departments with the tools they need to conduct their daily business with greater efficiency. The program serves the students by providing them with the time and skills needed to fully understand the complexity of software engineering. The students in the program, all from the millennial generation, indicate the program engages them with software engineering in a way that aligns well with their values and beliefs. Furthermore, as the students progress through the program, they are feeling significantly more confident in their software engineering skills, gaining marketable skills, feeling more confident in their soft skills, and feel like the work they are doing is having a real-world impact.

ACKNOWLEDGMENTS

The authors would like to thank the entire Berea College community for their support in realizing this program. Through their willingness to serve as our customers, providing feedback and patience, our students are able to develop software that has a real impact in an environment that encourages them. Special thanks to the Computer Science Department for their hands-on support, and to Dr. Matt Jadud, who provided significant mentorship to students early in the program.

REFERENCES

[1] Atlassian. 2018. Gitflow Workflow. (2018). Retrieved February 2, 2018 from https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[2] Albert Bandura. 2006. Guide for constructing self-efficacy scales. *Self-efficacy beliefs of adolescents* 5, 1 (2006), 307–337.

[3] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. 2001. Manifesto for agile software development. (2001).

[4] DJ Besemer, KS Decker, DW Politi, and JF Schnoor. 1989. A synergy of industrial and academic education. In *Issues in software engineering education*. Springer, 399–413.

[5] Tebring Daly. 2011. Minimizing to maximize: an initial attempt at teaching introductory programming using Alice. *Journal of Computing Sciences in Colleges* 26, 5 (2011), 23–30.

[6] Remco Matthijs Dijkman and Stephanus Maria Mathias Joosten. 2002. Deriving use case diagrams from business process models. (2002).

[7] Scott Freeman, Sarah L Eddy, Miles McDonough, Michelle K Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat Wenderoth. 2014. Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences* 111, 23 (2014), 8410–8415.

[8] Gerald C Gannod, Janet E Burge, and Michael T Helmick. 2008. Using the inverted classroom to teach software engineering. In *Proceedings of the 30th international conference on Software engineering*. ACM, 777–786.

[9] Kathleen Keogh, Leon Sterling, and Anne Therese Venables. 2007. A scalable and portable structure or conducting successful year-long undergraduate software team projects. *Journal of Information Technology Education: Research* 6 (2007), 515–540.

[10] Glenn E Krasner, Stephen T Pope, et al. 1988. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming* 1, 3 (1988), 26–49.

[11] Guido Lang. 2017. Agile Learning: Sprinting Through the Semester. *Information Systems Education Journal* 15, 3 (2017), 14.

[12] Chang Liu. 2005. Enriching software engineering courses with service-learning projects and the open-source approach. In *Proceedings of the 27th international conference on Software engineering*. ACM, 613–614.

[13] Cynthia J Martincic. 2009. Combining real-world internships with software development courses. *Information Systems Education Journal* 7, 33 (2009), 1–10.

[14] Eddy SW Ng, Linda Schweitzer, and Sean T Lyons. 2010. New generation, great expectations: A field study of the millennial generation. *Journal of Business and Psychology* 25, 2 (2010), 281–292.

[15] Helen Partridge and Gillian Hallam. 2006. Educating the millennial generation for evidence based information practice. *Library hi tech* 24, 3 (2006), 400–419.

[16] Mary Poppendieck. 2007. Lean software development. In *Companion to the proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 165–166.

[17] Valentin Razmov and Richard Anderson. 2006. Pedagogical techniques supported by the use of student devices in teaching software engineering. In *ACM SIGCSE Bulletin*, Vol. 38. ACM, 344–348.

[18] Thomas P Schambach and Jim Dirks. 2002. Student Perceptions of Internship Experiences. (2002).

[19] Ken Schwaber and Mike Beedle. 2002. *Agile software development with Scrum*. Vol. 1. Prentice Hall Upper Saddle River.

[20] Mary Shaw. 2000. Software engineering education: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. ACM, 371–380.

[21] Mary Shaw, Jim Herbsleb, Ipek Ozkaya, and Dave Root. 2005. Deciding what to design: Closing a gap in software engineering education. In *International Conference on Software Engineering*. Springer, 28–58.

[22] Carolyn Snyder. 2003. *Paper prototyping: The fast and easy way to design and refine user interfaces*. Morgan Kaufmann.

[23] John D Tvedt, Roseanne Tesoriero, and Kevin A Gary. 2001. The software factory: combining undergraduate computer science and software engineering education. In *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*. IEEE, 633–642.

[24] Laurie Williams, Robert R Kessler, Ward Cunningham, and Ron Jeffries. 2000. Strengthening the case for pair programming. *IEEE software* 17, 4 (2000), 19–25.