Employing Students for the Creation of Institutional Software*

Bria Williams[1], Guillermo Cruz[1], Scott Heggen[1]
[1]Computer Sciences
Berea College
Berea, KY, 40403
{williamsbri, cruzg, heggens}@berea.edu

Abstract

Higher educational institutions, like most all large businesses, need to purchase third-party software to manage their daily operations. Because of limited budgets, the costs to purchase some needed software is prohibitive, resulting in the purchase of less apt software, or none at all. These limitations lead to interfaces that are sometimes difficult to use, poorly developed, not fully-featured for the needs of the institution, or simply the wrong tool for the job. Worse yet, contracts lock institutions into this software for multiple years, putting a strain on the faculty, staff, and students who rely on the software. The tech support by these companies is not always easily accessible and costs the institution more money. As an alternative solution, a team of student developers led by a faculty and minimal staff support can create custom, institution-specific software that meets the college's unique needs while providing students with valuable work experience building their skills and preparing them for software engineering positions after graduation. This paper highlights the successes of a five-year experiment employing college students as software developers creating software for a higher education institution.

1 Introduction

The growing complexity of higher education institutions means an increasing reliance on technology, in particular, software. Most institutions rely heavily on databases and management platforms, such as Banner [3], to manage the complexity. Similarly, learning management systems such as Moodle [9] and Blackboard [2] improve the educational experience of students and faculty by facilitating and automating common tasks such as grading and distributing information to classes. Some of the software is open source (e.g., Moodle), which means the software itself is free, but the customization and support of the software costs the institution. Proprietary software typically has a combination of an initial purchase cost, a maintenance cost, and a multi-year contract. While the software can receive good support from the developers and institution-specific customization as-needed, the annual costs can creep into the thousands of dollars per application. According to a 2015 report about higher education spending, it is estimated that higher education institutions spend on average 4.2% of their entire budget on information technology alone [19].

Most purchasable software is built to meet the general needs of managing higher education, such as course registration and scheduling, but do not handle the nuanced differences that exist at every institution. For example, some institutions operate on a typical semester calendar consisting of a Fall, Spring, and Summer term. Other institutions may operate on a quarterly calendar, or they may have other terms such as a January short-term, multiple summer terms, online courses without specific start and end dates, or not operate on a calendar system at all. These institution-specific modifications are supported by software developers through "additional services" they provide. In other words, they are going to cost the institution on top of the base cost of the software.

Worse yet, each institution is unique in its needs, and sometimes the software simply doesn't exist. For example, there are nine work colleges in the United States [4, 1]. A work college requires every student to be employed while attending, and student employment is considered a part of the institution's academic mission. Given there are only nine work colleges, there isn't a market for the creation of large-scale software to support work colleges. However, there is a lot of need for software to manage a work college, such as tracking student employment, time-sheets for clocking hours, reporting back to the federal government and other funding sources, and evaluating student and supervisor performance, to name a few. Using existing software sometimes solves parts of the institutions needs (e.g., there's plenty of time-sheet solutions out there already), but often is missing key features that are institution-specific. Again, custom software and modifications to existing software is expensive and

adds to the cost of managing the institution. In the case of work colleges, which tend to be smaller institutions, this cost is particularly prohibitive.

However, an institution's unique needs can be met by a team of students crafting custom, institution-specific software. Instead of hiring an entire software engineering team (which would be quite expensive), a faculty or staff member can serve as a project manager to a team of students, guiding them in the development of software. The institution can avoid the high cost of software and maintenance, but still get applications that are tailored specifically to meet their needs.

There are a plethora of examples of students creating real-world (i.e., software that is eventually used by the product owner to do their business) software in courses [22, 28], capstone experiences [16, 8], and internships outside of the institution [23]. Kaminar [15] summarizes a few instances where students developing software that was eventually adopted by the institution. These adoptions occurred organically however, and were typically one-off ventures. To our knowledge, no examples were found of institutions hiring students to develop custom software solutions in a multi-semester, internship model, making this program the first of its kind.

One of the primary benefits of a program such as this is that the students gain valuable experience, preparing them for the software engineering industry after graduation. The students learn numerous technical skills [24], such as specific programming and markup languages, software engineering principles, libraries, frameworks, web development, MVC [27], and more. Students also learn a number of valuable soft skills, such as critical thinking, leadership, project and time management, teamwork, and problem solving; skills deemed valuable for new graduates to have by employers [20]. Previous work has shown that these benefits are easily obtainable through the development of real-world applications, regardless of the setting with which the application is developed (course, capstone, internship, etc.) [14, 21, 5].

This paper describes the framework for a program which blends the best parts of a software engineering course, a capstone experience, and an external experience like an internship. Through a work-study program, students are employed by the Computer Science department to develop software for the institution. Teams of 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 and apply them to in-progress projects or new ones. The projects are proposed by the campus community (a.k.a. the product owner) and are then selected by the team. These projects are often tools requested by the product owner to help them complete their daily work, and thus, the software becomes an integral part of their job. Students intend to finish the beta version of the product by the end of summer. During the

subsequent academic year (i.e., Fall and Spring terms), the students work an additional 300 or more hours, where they maintain the software, implementing new feature requests, fixing bugs, and performing customer support for their applications.

The remainder of this paper is organized as follows: Section 2 describes the framework for employing students as software developers for an academic institution; Section 3 describes the software, how it is selected, and provides examples of software already created; Section 4 presents results from surveys issued to product owners about the effectiveness of the solutions developed by the students; and Section 5 concludes the paper with closing remarks.

2 The Student Software Development Framework

Over a five year period, a process was created and refined that informed the current framework, which is designed for employing students as software developers for the purpose of developing applications for an academic institution. Following best practices in lean thinking, the framework is constantly being evaluated, redesigned, and optimized each year. Thus, the framework presented in this section represents its current state, but has been generalized as much as possible. It will certainly change as needs of the team and the institution evolve. Figure 1 illustrates the framework, including both the summer and regular term, each of which have different structures as described in the remainder of this section.

The team of students follow Agile principles [7] using a modified version of Scrum [25]. The Scrum methodology dictates that there are three key roles in which all those involved with the development of a software fall under: product owner, Scrum Master, and the development team. The product owner, in relation to the team, is the individual (or group of people) who makes a software request and are the customers who determine the vision and purpose of the software product. Newly added features, fixed bugs, and completed software must be approved and accepted by the product owner before it is considered complete. The Scrum Master is charged with the responsibility of facilitating the development team and communicating the needs of the product owner. The Scrum Master supports and promotes the development team and ensures that the team understands and follows Scrum principles. In our framework, the Scrum Master is not typically a student, unless they have been on the team a significant number of years and can handle the responsibility of guiding the development of the software. Typically, the Scrum Master is a full-time employee of the college (i.e., faculty or staff) who also acts as the team's supervisor. Finally, the development team in this framework consists of undergraduate students who are "structured and empowered by the organization to

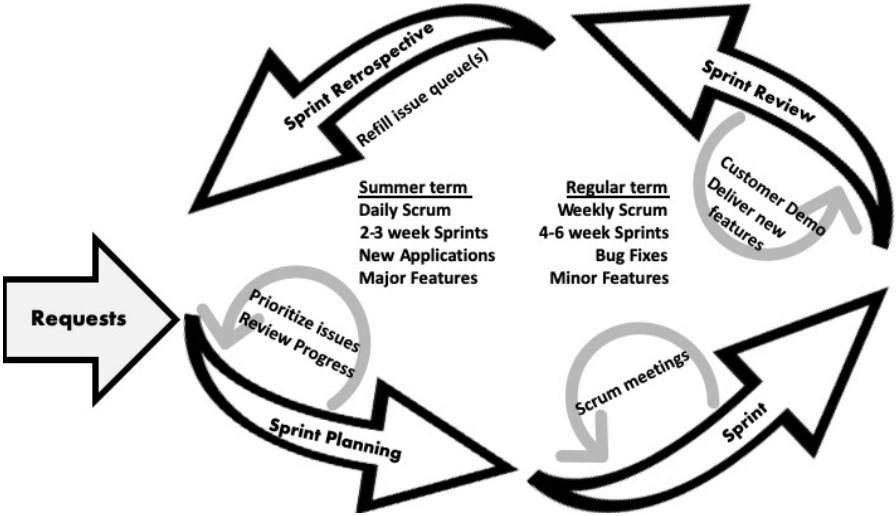

Figure 1: The Student Software Development Framework

organize and manage their own work" [25]. Following Scrum theory provides an iterative, yet incremental approach to cut down on risks and enhance the predictability of the project for the product owner and the development team.

Scrum also prescribes four events in the software engineering process: sprint planning, the sprint, sprint review, and sprint retrospective. These four events are implemented differently in the framework depending on the term. In the summer, the team follows a more traditional Scrum model with a product backlog and work in progress determined during spring planning, daily Scrum meetings occurring during the sprint, and a Sprint Review and Retrospective happening every two to three weeks before starting the next cycle. In the regular term (i.e., a Fall and Spring term consisting of 16 weeks each), the Scrum time-frame is stretched out due to students working only 10 hours per week instead of 40 hours per week. Daily Scrums become weekly, and Sprints take four to six weeks. Because the regular term is significantly less hours, we often save new systems and major features for the upcoming summer term, while smaller features and bug fixes happen during the regular term.

This next section will first describe our framework during the summer term, coined the Summer Internship Phase. Then, the following section will describe the framework during the fall and spring terms, coined the Maintenance and Customer Support Phase.

2.1 The Summer Internship Phase

Starting in the summer term, students are hired into the Student Software Development Team (SSDT). Students apply in the previous spring term through an open application process. The students are then hired based on a few metrics, including grades in key classes, such as the CS1 course; their ability to work well with others (based on their interactions with other students in classes, which follow a team-based [29], active learning pedagogy [17]; and the perceived value they will attain from participating in the program (i.e., the best computer science students are not always selected, as they may not benefit as much from the program), to name a few.

A typical summer operates much like an internship outside of an academic institution. Students are employed for 40 hours per week for eight to ten weeks, resulting in up to 400 hours of software engineering experience just in the summer alone. During this time, their expectations mirror an internship in many ways, in that they are expected to show up at work on time, be productive throughout the day, and report back to supervisors of their progress. Each summer consists of six to ten students managed by 2 supervisors (one faculty, one staff). The students typically work in pairs as they design interfaces and develop code.

Early in the summer, students have varying skill levels, but mostly all are untrained on any software engineering principles. All student recruited into the program will have passed the CS1 course, but otherwise, their experience varies. For example, rising seniors will have completed CS2 and maybe a few upper level CS courses, while rising sophomores may have only had CS1. Very rarely will a student have been explicitly trained on software engineering principles, such as Agile methodologies, Model-View-Controller (MVC) or similar frameworks, or large-scale application development.

Contrary to what might seem obvious, the students do not spend a significant amount of time in "training." Instead, we follow a "just-in-time" learning principle, where the students are given only as much theory and knowledge about software engineering as is needed to begin the task at hand. For example, one of the first skills students must develop is the ability to work in a version control system, in our case, Git. Students are walked through Git basics, such as how to clone the repository, make commits, and issue pull requests, and then they are given their first issue from the issue queue. Their first issue is often selected very intentionally by the supervisors; for example, finding and fixing a typo in a README of one of the repositories. This gives them the confidence to make a change and issue a pull request to the repository owner, without the fear of breaking any system.

A key skill needed and learned by our students very early in the summer internship is the ability to translate customer's needs into usable software (i.e.,

design). The first step in designing new software is to paper prototype [26] the software before spending significant hours writing code. This design process consists of many iterations of interfaces on paper; designs are critiqued and modified until there is a group consensus to move forward. Early the beginning of this process, when students are still very new to the idea, the supervising faculty member will act as the "driver," meaning they "click-through" the interface drawn on paper by the students, while those who designed the prototype will move pieces of paper, representing new parts of the interface, to act out its functionality. The first draft typically acts as an eye-opener for the students; the driver will find many flaws and functionalities that their designs are missing. These design mistakes are noted, and the students return to the drawing board to build another version on paper. After multiple iterations, review sessions, and the group approves of the interfaces, the product owner is invited to demo the entire paper prototyped application. Once the product owner's feedback is recorded and taken into account when making the next version. When the interface is finally approved, the paper prototypes are video recorded for later reference (i.e., during coding).

After paper prototyping is complete, the students review their interfaces and begin to construct the database supporting their prototypes. Again, many of the students have never taken a database design course, requiring more just-in-time training. As a group, they participate in a model building session in which they propose the underlying data structures of the application and the relationships between parts of the model. Multiple design iterations along with guidance from the supervisors lead to the students designing a working database which will support their paper prototypes. One of the key benefits of the students going through this process is they have a much better understanding of the models when they begin implementing their prototypes, reducing confusion about how data is saved and retrieved.

Having a design in place, plus an understanding of the data model, the students are ready to begin building the application. Students are first given a virtual machine (VM) to develop on, which mirrors the production environment, but also requiring them to learn some basic Linux commands and usage (again, a skill they may not have learned in their courses yet). In pairs, students clone a bare-bones template for the application, with only as much code written for them as is necessary to begin writing code. For example, a main page may be constructed, with minimal content in it, and a basic query to the database to fill it with some dynamic content. From there, the student pairs must replicate the structure of the view and controller into their own interface.

Throughout the process described above, students meet every morning for a daily Scrum. It is encouraged and expected that students will have plenty of questions early on in the summer internship. In fact, most of the just-

in-time learning described above happens as a result of a student asking a question. As the summer progresses and the students' capabilities improve, the questions become more technical in nature. In these morning Scrum meetings, demonstrations may also occur, as interfaces need group review. The students are encouraged to present their incomplete interfaces early, so they can get feedback about components they may impact future code. Lastly, code is often brought up to demonstrate good and bad coding practices, and correct them before the problems become too large to handle, to avoid large amounts of refactoring. The intention is to catch flaws, misconceptions, and problems early on to keep our programs agile and easily adaptable.

As the summer progresses, the supervisors take an increasingly hands-off approach, expecting the students to ask each other for help first. This promotes sharing of knowledge among teams, as pairs begin to solve problems that others have already solved (or those similar), and know some of the pitfalls to avoid. To maintain organization and visibility about what features each team is working on, we leverage Kanban boards, or Work in Progress (WIP) boards [6]. Applying the Kanban model in tandem with the Scrum framework aids in managing the overall flow of the project. The Kanban model prioritizes three essential principles: visualize what will be done today, reduce the amount of work that is in progress, and properly manage the flow. Kanban encourages continued cooperation and creates an environment that promotes ongoing learning by describing the optimal team workflow. The blending of Kanban and Scrum is also known as Scrumban [18]. Figure 2 shows the most recent iteration of our Kanban board (which has changed many times as the team finds ways to improve its usefulness).

Another important tactic in promoting the sharing of knowledge involves rotating teams. The team lead and supervising faculty member conduct a meeting a few weeks into the summer term to evaluate each pair's performance. They evaluate each developer's strengths and weaknesses, then redistribute the teams. Typically one developer stays on the interface they were already working on developing, and gain a new partner who is new to the interface. Shuffling the pairs helps students be flexible, learn to collaborate with others, and most importantly, reveals the importance of well-documented code and following coding standards adopted by the team. It also encourages the students to be able to explain their progress, clearly articulate their goals, and know exactly what their code does as they explain it to their new partner.

As interfaces get close to being completed, the pairs are expected to conduct a usability test [13] to ensure that their interface is user-friendly. The pair will run the usability test first with another team who has not used their interface prior. The hosting team provides a set of tasks or scenarios to complete, then observe the user. Usability tests showcase previously unseen faults or uncon-

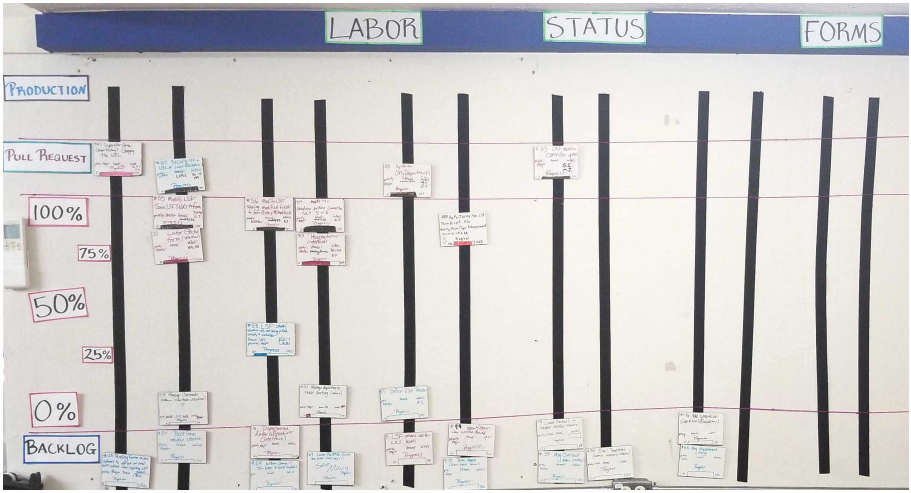

Figure 2: Kanban board for organizing team progress

sidered scenarios in their interfaces thus far. For example, during development, teams will commonly only test the expected use case. The usability test might reveal that there is another set of operations that the user can do which causes a crash. After fixing these new bugs or improvements, they will conduct another usability test, typically with the supervisors as the driver. It is expected that there will be less (but rarely none) new bugs or usability problems to fix. A third (or more) usability test is then conducted with the product owner, to get their final approval of the implementation.

Ideally, by the end of summer, the product is delivered to the customer fully implemented. As this is not always the case, constant communication with the product owner as well as clearly setting expectations very early alleviate issues with surprises about delays in deployment. Any features that aren't completed in the summer term become issues for the team during the regular term, after critical bugs are resolved (more on this in the next section). The team reflects on its progress, what worked well and what did not. Each interface is thoroughly documented of its state and what is left to be done so the next set of developers who take it on will have a good idea of what direction to go in.

2.2 The Maintenance and Customer Support Phase

As the Fall term begins, the team shifts from working 40 hours to 10 hours per week, as the students are now also responsible for attending classes and other

academic responsibilities. As expected, the productivity of the students shifts dramatically between summer and the regular terms. The framework takes this into consideration. Whereas summer terms focus heavily on designing new, major features or entire software systems, the regular fall and spring terms focus more on maintaining existing software and responding to customer needs (e.g., bug fixes or small feature requests). The expectation is that the regular term provides students with the valuable experience of having to maintain their software after deployment, a skill rarely taught in software engineering courses after projects are "completed" and delivered to customers.

Students meet with the supervisors at the beginning of the Fall term to schedule their hours around their courses and other academic responsibilities, with the goal of ensuring no student is working alone for extended periods of time. Students who work alone for too much of their time develop bad coding habits without their code being reviewed frequently, leading to wasted effort as the code they create doesn't make it to the final product. Similarly, students working alone often make poor design and usability choices as they implement, leading to useless software. Lastly, if the student gets stuck on a problem, there is no one there to brainstorm solutions with them. The supervisor, in our case a faculty also teaching courses, isn't as available to assist the students as quickly, meaning they must rely on each other to get unstuck more-so than during the summer. It is crucial that students work with other students to fix bugs and solve problems in order to maintain steady progress in development.

Work duties are mostly determined by the users as they encounter bugs, and the product owner, as they start thinking of new features they want in their software. Customers report bugs or new feature requests via email to the supervisors, who evaluate the priority and difficulty of the issue. The issue is then added to the issue queue for that application. If the issue is urgent (e.g., causing crashes or rendering an interface unusable), the issue is taken by the first student that is available (i.e., the next one to come to work). In some cases, the bug warrants one or more of the supervisors to work on the issue immediately. For non-critical issues, the issue is discussed in the weekly scrum meeting and added to a team's work in progress, or left in the issue queue if it's not considered the most pressing issue for that week.

Similar to summer, the Kanban board is instrumental to keeping the team organized and aware of each others' work, particularly since the students' schedules become more spread out. Students check the Kanban wall for new issues, claim them, and keep record of their progress on the wall. This flow helps developers visualize their progress and estimate their time, which is essential given they only have 10 hours per week to work. When the issue is resolved and tested by multiple student developers, a pull request is issued. Pull requests are reviewed by the supervisors, and returned if there are errors or standards

not followed. An accepted pull request must pass our guidelines for coding structure, naming conventions, security, and best practices. The supervisors are responsible for the last mile; integrating the feature into the production environment. Sometimes bugs are found during integration, which are pushed back down into the repository, requiring all developers to pull in these changes (as well as the new feature itself) to synchronize the everyone's local environments.

3 The Software

This section begins by describing the process for selecting new requests for software, followed by a brief summary of the structure the current software that has been generated by the SSDT has been built upon. Lastly, two examples of applications built by the SSDT are presented.

3.1 Selecting Software Requests

The team receives software requests from departments and offices at the institution who are aware of the Student Software Development Team and think we can help them. In some cases, the department needs to replace old or inadequate software they already own. In other cases, they are still relying on inefficient paper processes that could easily be replaced with a software solution. It is also not unlikely for team members to reach out to departments and inquire about their needs. When a software request is received, the team reviews the request and determines which projects are feasible and which lie outside of the mission of the team. Selected projects go into a backlog to be considered in the upcoming summer. A number of factors determine which systems we will implement, including the request's urgency, value to the institution, and the expected effort required to complete the project. All of these factors are weighed in order to select the request that best fits our capabilities and current capacity.

3.2 Software Built by the Student Software Development Team

The vast majority of the applications created by the SSDT are browser-based web applications. The SSDT follows an MVC architectural model to structure all of the applications. The team uses Flask, a web framework that uses Python as a back-end language (i.e., the controller), designed to make the start-up of a web application quick and easy. Coupled with two Python libraries, Jinja for template rendering (i.e., the view) and Peewee for database abstraction and integration (i.e, the model), the team has all the tools needed to begin building web applications. Since most of our applications are intended to be tools for

doing work (as opposed to web sites for selling products or advertising services), we leverage Bootstrap for web page styling and simplifying the implementation of the front-end of the application, particularly related to mobile-readiness and responsiveness.

In the five years since the creation of the SSDT, we have started twelve software systems. Of those twelve, three are retired and one is not maintained by our team any longer. The remaining eight applications are regularly maintained by the SSDT. Communications with the product owners occur at a minimum twice a year, where new features are proposed by them as they discover more ways in which the software can support their work. Otherwise, customers report bugs and issues via email to the SSDT supervisor. Each system was initially designed to serve a specific need within the product owner's department. For brevity's sake, we will describe two of these applications that are of relevance to many academic institutions.

3.2.1 The Chemical Inventory System

One responsibility of the Environmental Health and Safety department is to track and control the movement of hazardous chemicals on campus. Chemicals must be stored in a special bunker, and certain regulations exist for certain chemicals. For example, some hazardous chemicals cannot be stored above specific quantities, depending on which floor the chemical is on.

The institution was managing the chemical inventory through dated software which was slated for retirement. The department needed a new solution, but was not given an adequate budget to purchase existing solutions that served their exact needs. The SSDT was approached to implement a solution which solved the needs specifically for the department.

The SSDT created a web-based application which allowed the department to track the purchase, movement, and disposal of all chemicals on campus. The application had the added benefit of providing controlled access to interested individuals, such as chemistry faculty, to be able to view the current stock of each chemical, as well as see where they are currently stored and the age of the chemical. The system was extended further to include the ability to quickly generate a report for a building (in case of emergencies, such as a fire), as well as document the disposal of chemicals and track the history of a chemical as it moved to different places. In all, the software made the campus a safer place to work, improved visibility about the chemical inventory, and resolved issues of missing chemicals and data inconsistencies from years of use by the prior tool.

3.2.2 Course Administration and Scheduling (CAS)

The Course Administration and Scheduling (CAS) application was first requested by the Registrar's Office at our institution. Prior to the system, the Registrar's Office would send an email out to all of the department chairs at the institution requesting the schedule for upcoming terms. Included in the schedule were the course title, prefix and number, the faculty member(s) who would teach the class, the number of students who could take it, which room on campus was needed, if any, and the time of the course, to name a few. Department chairs would then respond to the email with their drafted schedule. This data would get get aggregated into a single spreadsheet by the Registrar, then emailed back out to the department chairs for review. This process would repeat for multiple weeks as department chairs resolved conflicts in scheduling (e.g., ensuring required courses from other departments not overlapping), each week requiring the Registrar to update the spreadsheet. To further complicate the process, one additional chair would need to approve all courses for their division (i.e., multiple departments), and some courses are not as straightforward as others (e.g., special topics courses, cross-listed courses, team-teaching situations, prerequisites, co-requisites, to name a few).

This process was error prone and inefficient for the Registrar's Office staff as well as the department chairs. The SSDT was approached to develop a software solution that would improve the situation for everyone.

The SSDT developed CAS, a web application that allowed department chairs to enter in all of the course information for their department. Each department chair had access to edit their own department, and could view the schedule of all departments as the schedule was being constructed. The Registrar's Office staff had access to edit all courses in all departments (i.e., administrator access). The application provided data validation tools to ensure only valid inputs were allowed (e.g., course numbers and titles matched the institution's course catalog), as well as the ability to indicate all of the non-standard courses, as mentioned above. The Registrar was given the ability to lock the term at a specific date, at which point no additional changes were allowed in the system. The full list of courses was then exported from our application and imported into the official schedule for that term.

The application was well-received, particularly by the Registrar's Office, who was no longer spending significant hours managing spreadsheets via email. The application has also survived multiple staffing changes, including a new Registrar and two new Assistant Registrars (who interacted with the software the most). In each case, the new staff member was able to learn the software quickly and integrated it into their workflow. The Registrar's Office has also requested new features and changes to existing features to be integrated into the system since its creation. For example, a matching algorithm was added

to facilitate a more unbiased approach to assigning rooms to classes, particularly in popular rooms where multiple faculty would lobby to the Registrar to be given that room. All faculty are now able to enter their preferences for rooms, and the system will attempt to assign everyone a room based on those preferences, without creating any room conflicts.

The CAS application started as a simple scheduling tool, and has evolved over the last four years into a much larger application for managing multiple aspects of course scheduling, empowering faculty to have more voice in their room placement, and improving visibility of the larger schedule to the entire faculty and staff.

4 RESULTS

4.1 Study Design

In evaluating our framework for leveraging undergraduate students for the development of institutional software, a survey was issued to all of our product owners asking them to evaluate the value of the software to them. Several factors were identified to gauge software success. The metrics for software success were informed primarily by DeLone and McLean's Model for Information Software Success [11]. Their model consists of six measures of success: system quality, information quality, use, user satisfaction, individual impact, and organizational impact. Questions were developed with aligned with three of the six components from DeLone and McLean (system quality and information quality were excluded, since product owners are not always able to accurately judge these metrics; use was excluded due to the controversial nature of its applicability to measuring software success [12]).

4.2 Participants

Participants in this study included 14 institutional faculty and staff who all represented product owners, in that they were administrators in the systems and had the largest amount of interactions with the software. Ten out of the fourteen clients responded to the survey. The ten respondents to the survey represented product owners from six of the eight applications.

4.3 Evaluation of Software Success

4.3.1 User satisfaction

To measure user satisfaction, the product owners were asked a set of four questions about how easy it was to use and learn the software. The questions asked were: how difficult was it to complete specific tasks in the software; how

Table 1: User Satisfaction - Survey Results. The scale is: Strongly agree (SA), Agree (A), Neutral (N), Disagree (D), and Strongly Disagree (SD)

Question:	SA	A	N	D	SD
Software easy to use/learn	80%	20%	0%	0%	0%
Makes job better	70%	20%	10%	0%	0%
I feel confident using software	60%	40%	0%	0%	0%
Software useful to my job	90%	10%	0%	0%	0%
My level of software competency irrelevant	20%	20%	50%	10%	0%

long did it take to learn the software; how useful were certain features in the software; did the application make their job better; and did they feel more confident while using the application. Table 1 summarizes the product owner responses.

Respondents were overwhelmingly positive about user satisfaction; All ten product owners indicated the features were easy to use, was useful to their job, and they were all confident in their ability to use the software. 90% also indicated the software made their job better. When asked if the product owner's prior knowledge played into their ability to use the software, the responses were split; 50% of respondents were neither in agreement or disagreement with the statement, while 40% indicated their prior experience did not play a role in their ability to use their software. One respondent indicated that their prior software competency was important to their ability to use the software.

Despite undergraduates designing and building the software, some with little to no design or web development experience coming into the experience, the students are still capable of producing software that is satisfactory to the needs of the customers; in our case, faculty, staff, and administrators at the academic institution.

Within the four questions, the users were asked to rate how useful the primary features of the software were to them and to their daily tasks. Respondents were overwhelmingly positive about user satisfaction; 80% of users indicated that they "strongly agreed" that the features in the application were easy to use and two answered that they "agreed" that features were easy to use. Easy of use and learnability are important components of the user satisfaction measure when designing software for a customer because it is critical to the SSDT that it does not provide a "solution" that only makes the users' work-life more difficult. The results of this component of the survey also showed that nine out of ten respondents answered that they "strongly agree" that the primary feature within the respective softwares is useful; the tenth respondent

marked that they “agree” that primary feature was useful. One participant of the survey added in the optional commentary that their favorite aspects of the syllabi archival software was that it allowed them to “find the syllabi that I need” and “I can go back to previous terms pretty easily”. This participant in particular is an administrator of the syllabi software who provides support to the professors at the institution who use the non-administrative version of the application; they went on to say “I can easily tell who hasn’t submitted their syllabi". This user’s testimony further substantiates the results of the initial question about the software’s ease of use, overall learnability, and usefulness.

The final two question of the user satisfaction portion of the survey asked the product owners to expresse the extent of their agreement with the following two statements as it pertains to the particular application that they utilize: “The application makes my job better” and “I feel confident while using the application.” Out of the ten respondents, eight product owners said that they “Strongly Agree” that the application makes their job better. One of the two other users answered that they “Agree” with the statement and the last remained “Neutral” on the question. Dalcher [10] suggests that quality solutions emerge from the consideration of effectiveness and that to attain project success one needs to relate the quality aspects and perspectives related to the effectiveness of the project. The software built by the SSDT prioritzes the effectiveness of the solution in order to provide a more favorable product to the users. In response to the statement about their confidence in using the application, seventy percent of the participants answered that they “Strong Agree" with the statement whilst the other three all answered that they “Agree” with the statement. The user’s confidence in their usage of the application attributes to the user’s satisfaction by allowing the user to complete their everyday tasks without worry of errors or confusion.

Last, the users were asked how long they typically engage with the software whe they use it to do their work. Fifty percent of the users answered that they use the software between 15 to 30 minutes whilst 40% of the users answered that they engaged with the software from 30 minutes to an hour at a time. Only one user answered that they used the software for fifteen minutes or less.

4.3.2 Individual Impact

Individual impact is loosely based on the user’s personal interactions with the software and its effects on them. Individual impact includes the user’s confidence when using the software, improved personal productivity, the amount of time it takes to complete a task as compared to when the user did not have the software, as well the amount of effort that is put into using the software. Product owners were asked to estimate how much time they spent completing tasks before the software was implemented, and how much time it took to complete

Table 2: Individual Impact - Survey Results. The scale is: Strongly agree (SA), Agree (A), Neutral (N), Disagree (D), and Strongly Disagree (SD)

Question:	SA	A	N	D	SD
Before software, tasks took long time	50%	20%	10%	20%	0%
After software, tasks became quicker	70%	0%	10%	20%	0%
Application meets my needs	50%	40%	10%	0%	0%
Allows me to do my job faster	80%	10%	0%	0%	10%

the same task using the software. The product owner was also asked if the software met their needs as defined in their initial request. Also, product owners were asked if the application helped make their job go faster. Finally, they were asked how much of their day-to-day responsibilities and work depended on their ability to use the newly developed application. Table 2 summarizes the product owner's responses to each question.

Respondents overwhelming agreed that the software met their individual needs, with 90% positive responses. The survey revealed that 90% of responses felt the software allowed them to do their job faster. When comparing the questions related to length of time spent doing tasks before and after the software was implemented, 70% of respondents agreed that they were both now quicker than before, and tasks took too long prior to the software.

4.3.3 Organizational Impact

Lastly, organizational impact evaluates the organizational goals of the institution, cost reduction, and the efficiency and effectiveness of software to the organization. The product owners gauged organizational impact on six questions. Four of the questions asked the product owner about the cost in terms of money, stress, errors, and time if the new software was suddenly completely taken down and they had to revert back to their previous processes; the fifth questions asked product owners to evaluate the significance of the software to their daily duties, an indicator of the importance of the software to that department's operation. Lastly, product owners were asked to directly evaluate if the software improved the organization as a whole. Table 3 summarizes the product owner's responses.

When considering the cost of the software in terms of time, money, stress, and errors were the system to be removed, responses aligned with our expectations; over 80% of respondents agreed with a high cost. Regarding monetary

Table 3: Organizational Impact - Survey Results. The scale is: Strongly agree (SA), Agree (A), Neutral (N), Disagree (D), and Strongly Disagree (SD)

Question:	SA	A	N	D	SD
Daily work depends on software	60%	40%	0%	0%	0%
Software improved the inst. as whole.	70%	20%	0%	10%	0%
Removing software would cost time.	80%	10%	10%	0%	0%
Removing software would create stress.	40%	40%	20%	0%	0%
Removing software would cost money.	50%	10%	10%	0%	30%
Removing software would create errors.	40%	40%	20%	0%	0%

costs, 60% of respondents agreed there would be a high cost penalty were the software eliminated. This slightly lower metric is likely due to the fact that none of the software created by SSDT earns any money; they all improve processes within departments, which saves money through reduced labor costs doing tasks that can be automated through software. All respondents indicated their daily work depended on the software, indicating there is a lot of value derived by our product owners from having this software. Lastly, 90% indicated that the software improves the institution as a whole. Clearly, our product owners believe in the value of the software generated by the SSDT.

5 Conclusion

In conclusion, higher educational institutions, like most all large businesses, need to purchase third-party software to manage their daily operations. Because of limited budgets, the costs to purchase some needed software is prohibitive, resulting in the purchase of less apt software, or none at all. These limitations lead to interfaces that are sometimes difficult to use, poorly developed, not fully-featured for the needs of the institution, or simply the wrong tool for the job. Worse yet, contracts lock institutions into this software for multiple years, putting a strain on the faculty, staff, and students who rely on the software. As an alternative solution, a team of student developers led by a faculty and minimal staff support can create custom, institution-specific software that meets the college's unique needs while providing students with valuable work experience building their skills and preparing them for software

engineering positions after graduation. This can be accomplished through a year long program with two modes of operation: Summer Internship and the Academic Year. During the Summer Internship Phase, students learn crucial skills and computer science concepts, such as frameworks, programming languages, and Agile methodologies, and work to build large new features and interfaces. Group and individual process is monitored through daily Scrum meetings and students practice pair programming to bring their designs to fruition. During the Fall and Spring terms, the team will transition to the Maintenance and Customer Support Phase. There, they will focus on smaller feature delivery and providing customer support as needed. Detractors are quick to indicate that trusting undergraduates with this task, even under the leadership of faculty or staff, is an unacceptable risk to the institution. However, the Student Software Development Team at Berea College utilizes this program to design, craft, and maintain software for a range of offices on campus. The applications have quickly became an integral part of customers' jobs, assisting in simplifying institutional management processes across campus.

References

[1]

[2] Education technology services, Oct 2019.

[3] Ellucian banner, Oct 2019.

[4] Member colleges, 2019.

[5] Zakarya Alzamil. Towards an effective software engineering course project. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 631–632. IEEE, 2005.

[6] David J Anderson. *Kanban: successful evolutionary change for your technology business.* Blue Hole Press, 2010.

[7] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001.

[8] AT Chamillard and Kim A Braun. The software engineering capstone: structure and tradeoffs. In *ACM SIGCSE Bulletin*, volume 34, pages 227–231. ACM, 2002.

[9] Chris. Moodle lms - online learning with the world's most popular lms, Oct 2019.

[10] Darren Dalcher. Software project success: moving beyond failure. *Upgrade*, 2009.

[11] William H DeLone and Ephraim R McLean. Information systems success: The quest for the dependent variable. *Information systems research*, 3(1):60–95, 1992.

[12] William H Delone and Ephraim R McLean. The delone and mclean model of information systems success: a ten-year update. *Journal of management information systems*, 19(4):9–30, 2003.

[13] Joseph S Dumas, Joseph S Dumas, and Janice Redish. *A practical guide to usability testing*. Intellect books, 1999.

[14] Scott Heggen and Cody Myers. Hiring millennial students as software engineers. *no. June*, 2018.

[15] Ariel Kaminer. Student-built apps teach colleges a thing or two. *New York Times*, Aug 2014.

[16] Kathleen Keogh, Leon Sterling, and Anne Therese Venables. A scalable and portable structure or conducting successful year-long undergraduate software team projects. *Journal of Information Technology Education: Research*, 6(1):515–540, 2007.

[17] Clifton Kussmaul. Process oriented guided inquiry learning (pogil) for computer science. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 373–378. ACM, 2012.

[18] Corey Ladas. *Scrumban-essays on kanban systems for lean software development*. Lulu. com, 2009.

[19] Leah Lang. *2015 CDS Benchmarking Report*. March 2016.

[20] Ilana Lavy and Aharon Yadin. Soft skills-an important key for employability in the" shift to a service driven economy" era. *International Journal of e-Education, e-Business, e-Management and e-Learning*, 3(5):416, 2013.

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

[22] Stephanie Ludi and James Collofello. An analysis of the gap between the knowledge and skills learned in academic software engineering course projects and those required in real: projects. In *31st Annual Frontiers in Education Conference. Impact on Engineering and Science Education. Conference Proceedings (Cat. No. 01CH37193)*, volume 1, pages T2D–8. IEEE, 2001.

[23] Michael J Lutz, J Fernando Naveda, and James R Vallino. Undergraduate software engineering. *Commun. ACM*, 57(8):52–58, 2014.

[24] Vreda Pieterse and Marko Van Eekelen. Which are harder? soft skills or hard skills? In *Annual Conference of the Southern African Computer Lecturers' Association*, pages 160–167. Springer, 2016.

[25] Ken Schwaber and Jeff Sutherland. The scrum guide. *Scrum Alliance*, 21:19, 2011.

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

[27] Chanchai Supaartagorn. Php framework for database management based on mvc pattern. *International Journal of Computer Science & Information Technology (IJCSIT)*, 3(2):251–258, 2011.

[28] Nasser Tadayon. Software engineering based on the team software process with a real world project. *Journal of Computing Sciences in Colleges*, 19(4):133–142, 2004.

[29] Laurie Williams, Eric Wiebe, Kai Yang, Miriam Ferzli, and Carol Miller. In support of pair programming in the introductory computer science course. *Computer Science Education*, 12(3):197–212, 2002.