A Long-Term Study of Software Product and Process Metrics in an Embedded Systems Design Course

Dr. J.W. Bruce, Tennessee Technological University

J.W. Bruce is with the Department of Electrical & Computer Engineering at Tennessee Technological University in Cookeville, Tennessee USA

Dr. Ryan A. Taylor, University of Alabama

Dr. Taylor received his Ph.D. in Electrical and Computer Engineering from Mississippi State University in 2018. He is currently an assistant professor at the University of Alabama in Tuscaloosa, Alabama. His research interests revolve around remote sensing and engineering education.

©American Society for Engineering Education, 2019

A Long-Term Study of Software Product and Process Metrics in an Embedded Systems Design Course

In response to input from advisory employers, market demands, and academic studies [1], many computer engineering programs have increased focus on embedded computer systems. Embedded systems form a rich application through which computer engineering education can be made relevant. Embedded computer systems are a timely subject that is immediately useful to students in their senior capstone design projects. Furthermore, a large number of our computer engineering graduates currently use or design embedded computer systems in their jobs.

A team-based progressive embedded systems design course was developed that, in addition to providing the technical embedded systems knowledge, develops team and communication skills in situations emulative of industry [2]. The course was a success by many accounts; however, student teams abandoned sound design practices in attempt to meet the demanding 15-week “time-to-market” constraint. Team members produced defect-riddled designs and the design schedules slipped due to an unproductive test-redesign-test development cycle.

Afterwards, the course was retooled [3] to use a lightweight design process based loosely on proven software engineering standards [8], [9], [10] which detect defects during design. This development process has been used with success in the subsequent offerings of the design course based on a more complex project [6]. The resulting student designs are typically on time and of high quality. Furthermore, students report satisfaction with the experience, because of both the visible results at course end, and the perceived relevance of the processes that they used.

All of these course offerings have made a visible impact on the computer engineering program at the authors' institution. Computer engineering student projects in the capstone design course have greater complexity and are of higher quality compared to previous years. It was decided that these useful embedded system design skills be introduced to students earlier in the program, so a simplified embedded systems approach was adopted in the basic microprocessors course [5]. With fundamental embedded systems content being taught at the junior level, the senior-level embedded systems course was freed to concentrate more on system concepts and integration issues that are more common in engineering practice and industry: team-based design [2], industry-based software engineering standards [3], and the higher-order design constraints [6] that students were lacking in their capstone designs. This paper briefly describes the structure of the authors’ approach to the embedded systems course, the course’s evolution over twelve years of the study, and the process and product measures collected by students in the course. Analysis and discussion of the data is provided.

The Course – Motivation and Evolution

Over the years, the authors have reported on the evolution of the embedded systems design course. The authors' college of engineering has required students to purchase laptops since 1999

and many courses make substantial use of these valuable student-provided resources. The mission of the course described in this paper is to be a culminating design experience focused on embedded systems. This course offers a directed, and reasonably tightly controlled, design experience to prepare the students for the much more unstructured design experience in their capstone senior design projects. A secondary mission of the course described here is to introduce the students to practices commonly found in the commercial embedded systems design environment. Since embedded system designs are so naturally adapted to the object-oriented design paradigm, the course has always tended toward a high-level, reusable and modular approach.

The early Java years: before 2007

The earliest offerings [4] of the course used a specialized (read: expensive) development board with a high-performance 8051-compatible microcontroller running a custom Java virtual machine. Embedded code development was done in Java with a communications interface to an intermediate PC running Java code. The Java-enabled 8051 development board read a variety of weather sensors with results being sent to the student's laptop. Ultimately, weather sensor data was stored in a MySQL database on a centralized campus server with Java servlets allowing Apache Tomcat providing the data to the world via dynamic web pages.

The 8-bit years: 2007-2009

The first two offerings (2007-2008) of the course discussed in this paper largely followed the design specification of the previous years: a microcontroller reading a weather sensor suite with communications to a laptop PC for additional processing and ultimately the storing of the data in a MySQL database for dynamic web page creation. The key difference in the 2007 approach was that the course was altered to deal with the high costs and fragility of the development board. The 2007 offering adopted a student-developed design for microcontroller interfaces to the hardware and sensors. Students developed the system on a breadboard and eventually created their own professionally-manufactured printed circuit boards. Parts were obtained from parts kit purchased by students in the prerequisite microprocessors course [5]. The low-cost 8-bit microcontroller ran C code and provided the RS-232 communications link to the PC running Java as in the earlier iterations. The client-server architecture from the pre-2007 versions remained in place. Although the first-half of the semester was dramatically different than the pre-2007 iterations, design output in the the second-half of the course was almost completely unchanged. Once the data was successfully transmitted to the PC, the existing Java code and client-server architecture worked identically.

The 2009 offering adopted the model of the 8-bit microcontroller using USB communications instead of RS-232. Since USB requires a software stack and housekeeping on both endpoints, a cooperative multitasking operating system named "Embedded Systems Operating System" (ESOS) was written by the first author for use on the 8-bit microcontroller to facilitate faster firmware development cycles and give students a ready-built USB service. The provided ESOS and USB infrastructure allowed students to concentrate on the specific design requirements like previous semesters. While the general form of the client-server structure remained in place, students were given the option to develop the client-server portion with a much simpler, albeit

less robust and secure, Python implementation instead of Java servlets. It was becoming clear that Python was a powerful language with which computer engineering students should gain more experience.

Stepping up to 16-bits: 2010-2011

The 2010 offering of the course was identical to the previous except the microcontroller was changed to a 16-bit model and the Python back-end was mandatory. ESOS was ported to the 16-bit microcontroller and the OS structure and API were harmonized. The resulting operating system was more logical and consistent, and could be easily ported to a number of architectures if desired. Students continued to initially develop on their own breadboard approaches to interface with weather sensors and ultimately design their own PCB for use in the last few design tasks. Transitioning from an 8-bit to a 16-bit MCU for hardware interfacing is potentially a very dramatic change. Surprisingly little changed from the student's perspective because ESOS had been ported to the new processor. The ESOS API hid much of the change of the data size beneath and only minor documentation changes were required of the instructor/author. The 2010 version of the design introduced a simple task where the Controller Area Network (CAN) bus was used to communicate between student designs locally. Student designs would share limited sensor information over CAN. In 2011, students were required to extend ESOS to include a simplistic service for managing a Hitachi 44780 compatible LCD character module.

Standardized Hardware and ESOS extensions: 2012-2015

Prior to 2012, students developed their own microcontroller designs on breadboards and eventually a student team-designed PCB. This approach provides valuable experience to the students, but led to a great deal of variation and incompatibility between student team approaches. As class roster changes during the semester are inevitable due to student course withdrawals, team compositions were adjusted to maintain balance between team roster sizes. When a student was assigned to a different team mid-semester, the new team’s approach could be dramatically different than the one used in the student’s previous team. These experiences tended to frustrate students. Furthermore, the addition of CAN bus communications two years earlier was also much complicated by the variations between student designs. To alleviate these problems, students in the 2012 course were given an entire development board design kit consisting of PCBs and parts. As the design experience progressed, students would increasingly populate and test portions of the PCB. By semester’s end, the board would be fully populated, tested, and executing student code. Furthermore, the 2012 course removed the design tasks that required interfacing with the weather sensor suite. Class enrollments had grown, and the weather sensor suite located on the building roof was now more than 10 years old. With the removal of the weather sensor requirements and with common hardware used across the entire class, CAN bus design tasks were increased and the LCD character module OS service design was made more elaborate.

The 2013 course design tasks were identical to 2012. However, the design experience shifted to fully exploring the provided hardware and ensuring solid OS support for its capabilities. In addition to creating the LCD character module service as in the previous three years, students also had to extend the OS services to manage a variety of sensors and processing, and to provide

a robust human-machine interface service to manage the variety of input devices on the PCB . The CAN bus development remained much the same as before.

By 2013, the course structure and design specification had been informed by more than a decade of teaching a team-based, experiential, progressive design course on embedded systems. The approach was deemed solid, and remained largely unchanged. During the 2013-2015 period, the design specifications were altered only slightly, and some tasks were replaced by other tasks of similar complexity.

Gap Learning: 2016-2017

The two offerings in 2016 and 2017 were identical to each other and used a design specification that was very similar to 2013-2015 classes. The only real change was that the authors employed a design education approach called gap-learning [7]. Instead of asking students to use the OS API to implement the desired application functionality in any way they choose, gap learning has the instructor and lab assistant develop a common framework for how the system can best be implemented. Once this framework is determined, it is partially implemented, leaving key portions of the design missing. Students are then assigned the task of filling in the design's missing gaps. This approach requires the student to approach the design first with an inquisitorial attitude, searching to understand the design framework that has been set up for them. Once this understanding is complete (or sufficient), the student and his or her teammates are able to embark upon the completion of the design requirements. In a coarse view, gap learning is a "fill-in-the-blanks" design, except the blanks may be single lines of code, a block of code, a subroutine, or an entire API with only function calls defined.

Continued gap learning, 32-bit processor, and a different institution: 2019

In 2018, the embedded systems course was not offered as the lead author had moved to a new institution. In 2019, the course was offered for the first time at the new institution using approach described in this paper. During the 2007-2017 period, the prerequisite "microprocessors" course used a smaller processor from the same family line of the microcontroller used in the embedded systems course. The prerequisite "microprocessors" course at the author's new institution employed a different processor, and this processor was deemed dated and unsuitable for use in the embedded systems course described here. Furthermore, the prerequisite microprocessors course at the new institution has a strong emphasis on low-level assembly language programming with limited coverage of hardware interfacing topics. Timers and interrupts were the only microcontroller hardware devices covered in any detail. Students in the 2019 course had little to no knowledge of synchronous and asynchronous serial communications protocols, data converters, and other common microcontroller peripherals. The 2019 course would need to introduce the fundamental concepts and basic microcontroller peripheral operations before systems-level topics could be addressed. The 2019 version of the design experience compared similarly to the 2016 and 2017 offerings using the gap-learning techniques. The 2019 course adopted a very capable 32-bit microcontroller with a floating-point unit, a DMA engine, and multiple instances of nearly every hardware peripheral imaginable. Over the course of the semester, the authors ported, first, the core of the ESOS code, then, various services to a 32-bit ESOS implementation. Student design assignments were to "fill in the gaps" of a mostly

complete and ported OS code and services, then develop a demonstration application using their results.

Assessment

The foremost assessment of the course's effectiveness is that all teams must successfully complete the project requirements and submit working designs. Although the lab's design specifications are ambitious, there has never been a team fail to finish the lab design. By informing students early and often that failure is not an option, the teams always seem to find a way to make their designs work.

Software development metrics can increase productivity, identify development process shortcomings, increase software quality, and aid in development planning [11], [19]. Student confidence can be increased by comparing their metrics to those published in the literature. Furthermore, the instructor gains insight from collecting metrics on the students’ software process. The instructor can observe the improvement in individual and class abilities, as well as acquire indicators of the relative complexity of homework and design assignments. Many software measures in IEEE Std. 982.1 are obtained very naturally during the development process described in the previous section [3]. Throughout the design activities and the inspection process, students are asked to record:

- Defects -- identified by author, design task, defect type, and severity
- Person-hours -- recorded by team member, task, and activity (design, review, deployment, or testing)
- Output -- lines of code (LoC) identified by author, design task, and software routine

Each team member maintains his or her own records and is required to compute several additional measures. For example, coding efficiency is computed as LoC per workday where a workday is defined as eight person-hours of effort, development costs per LoC are computed assuming a hourly rate of $75/hour, and a code quality measure is computed as number of identified defects per thousand LoC. Data collection is facilitated by forms for recording time spent in each activity, recording details of defects found, and summarizing inspection findings. A spreadsheet is provided to compute all measures by individual, team, and design task. Example data collection forms and spreadsheets can be obtained by contacting the author.

The data disclosed in this paper is self-reported by the students, with each student and team having a different standard by which they choose to measure themselves. Large variations in the data may be attributed to variation in discipline and expectations on the part of the students. Figure 1 shows the student-reported hours spent on course tasks. In total, the 164 student participants over the period 2007-2019 invested 12,394 person-hours of effort. In classroom discussions, we assigned a loaded compensation rate of US$75 per hour. Therefore, the authors’ mythical design team incurred non-recurring engineering (NRE) costs of nearly US$930,000.

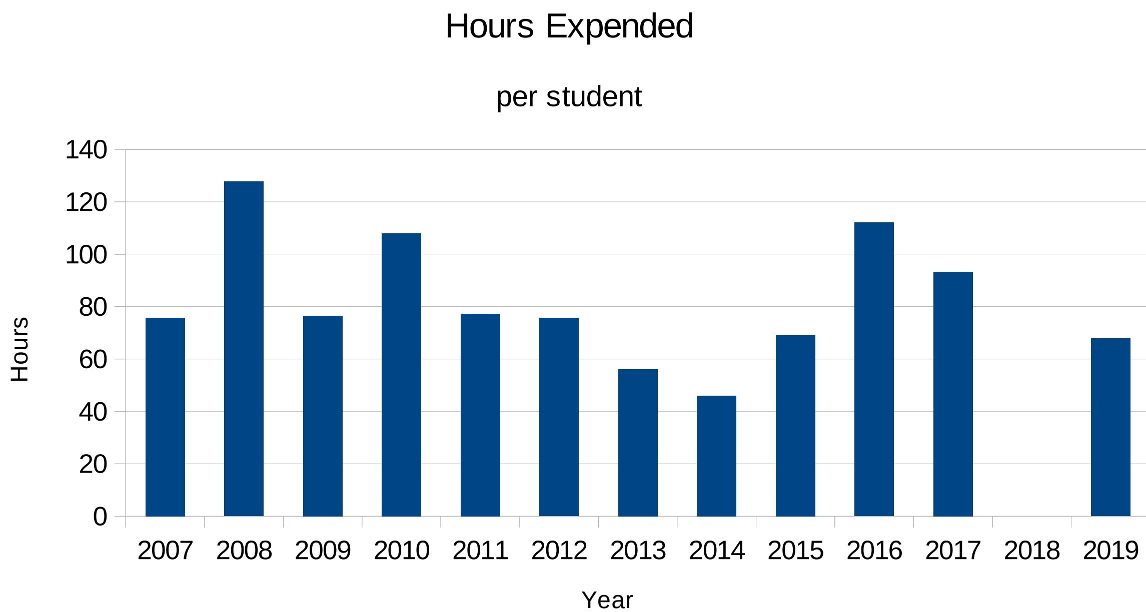

Figure 1 Average student effort for course activities over the semester

Each year, the lab design specification was changed in subtle ways. Some years, entire tasks were completely changed. These changes in the lab design specification were made to provide variety, address new topics introduced into the course, prevent academic dishonesty, and deal with changes to target hardware components. Furthermore, end-of-semester metrics were analyzed and adjustments made for the following offering of the course – with changes often made to reduce the student workload. By the last offerings in 2016-2019, the lab specification was dramatically different than the first specification in 2007. Over the twelve years of data presented in this paper, the 165 students reported an average of 77 hours spent on lab design tasks with a standard deviation of 45. Only twice during 2007-2019 did the average student report spending less than 60 hours on design activities. This level of effort is one of the authors' goals as it represents approximately 4 hours per week over the 15-week semester.

The overall trend from 2011 to 2014 shows a trend of reducing student effort. In 2016, a gap learning [7] approach was introduced in an attempt to reduce student workload further while still allowing completion of more complex designs. Gap learning is similar in concept to "guided notes" [15][16]. While guided notes – incomplete lecture notes – are typically provided for use in a lecture, our gap learning technique provided an incomplete design for a design activity. A design framework or a full reference design is created for each design milestone. Students are given a partially implemented version, where small but key parts of the design are omitted. The omitted portions of the provided design is such that diligent study of the provided information makes obvious what is missing. With so much of the design already specified in detail, there is little latitude for the students to do anything else but supply the missing information, code, etc. It is a "fill-in-the-blank" design. This approach requires the student to approach the design first

with an inquisitorial attitude, searching to understand the framework that has been set up for them. Once this understanding is complete (or sufficient), the student is able to embark upon the completion of the design requirements. It was hoped that this technique would affect multiple benefits. First, the techniques would allow the students to see the framework of a successful design before beginning their own implementation. This helps to visualize a successful design as a team before they are thrown into the throes of their senior capstone design project. Second, the techniques would remove some of the tedious work that should be covered in prerequisite courses. On a quick timeline such as a 15-week semester, precious time cannot be spent rehashing material that the students should be comfortable with already. This also reduces the overall workload from a course that is naturally "work-heavy." Third, these gap learning techniques would allow for students to be able to immediately see the heart of the concept that is being addressed with each lab milestone. Added possible tertiary benefits included a lower stress level for students and potentially smoother team interaction. While data collected [7] implied that the first two objectives were largely met, the gap learning approach does not seem to save time. While students do not have to form system architectures or write and debug many lines of code, the student must read, study, understand, and internalize the entirety of the design and associated device data. An improvement in student workload was noted in 2017. It is believed that the improvement in 2017 was largely due to the instructor being more skilled with the approach. Based on the 2016-2017 experience, the 2019 revision of the course was created where the "gaps" were smaller and very well-defined. Functionality of each of the lab tasks was still quite complex with a very large percentage of the code provided to students in the design assignment. Student-generated code was limited and very localized. The authors are continuing research into gap learning efficacy and ways to streamline the process for students.

Figure 2 shows the average number of lines of code (LoC) written by a student in each offering of the course. It is widely held that LoC is a poor product measure. Use of high-level languages such as Java or Python compared to low-level languages like C make comparisons of LoC in production difficult. Furthermore, it is true that a few well-written LoC are more useful, better, and often more time-consuming to write than a large numbers of lines of poorly written code. In the data presented here, most of the students' output was straight ANSI C language running on a small microcontroller, so comparisons between students and semesters is not entirely unreasonable. Once again, data shown here was self-reported by students. Students were instructed to only report LoC developed in the submitted design milestone, and not to count previously written LoC and LoC provided in libraries, gap learning frameworks, etc. The authors believe that student reported measures for this metric are suspect, as student teams submitted designs to identical specifications with LoC product measures that vary significantly.

In total, the 164 student participants over the period 2007-2019 reported creating 185,808 LoC. Again using our loaded compensation rate of US$75 per hour, the twelve year design effort averages to a software production cost of US$5.00 per LoC. One software manager's blog [20] reported US$3.98 per LoC for a traditional programming design team that he personally served as the design architect and manager. Several other studies [19] report software development costs ranging from $5-100 per LoC.

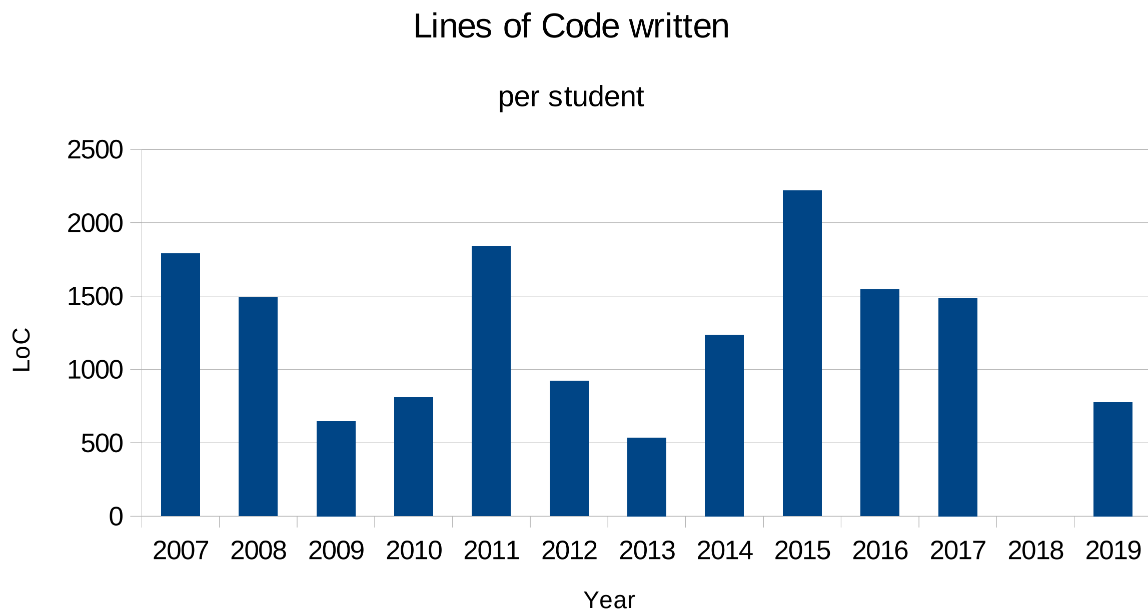

Figure 2: Average student output for a semester-long effort for 2007-2019

Changes in the course’s design requirements are reflected in Figure 2. For example, the design requirements changed only slightly from 2009 to 2010. In 2011, the design specification remove much of the client-server requirements where MySQL transactions and server-side Java servlet (high-level language interaction with dynamic web server frameworks) were replaced by more hardware-focused microcontroller code (C language control of registers and student-written data structures) to control external devices. The 2013 offering removed the laborious task introduced in 2012, replacing it with a requirement that students develop a new service for the OS. That change was well-received, so low-level microcontroller code development was removed and replaced by a task for students to develop a second service for the OS. Unfortunately, that service was a bit more complex than the 2013 addition. In 2015, CAN bus functionality was added to the 2014 specification. The CAN bus is a robust but complex network protocol, and student-reported development peaked. Apart from the heavy workload, students reported satisfaction with the experience as they found the CAN bus interesting and saw the utility of having experience with it for future courses and their careers. The 2016 design was fundamentally identical to the 2015 variation but with the introduction of gap learning. A clear reduction in LoC reported is seen. The 2017 offering was nearly identical to the 2016 version and reported LoC was almost unchanged. The effort in 2019 to reduce student workload by providing more of a complete design framework and limiting student requirements is clearly seen in Figure 2. Also, the authors are likely more adept with gap learning as the 2019 was the third course iteration using the method.

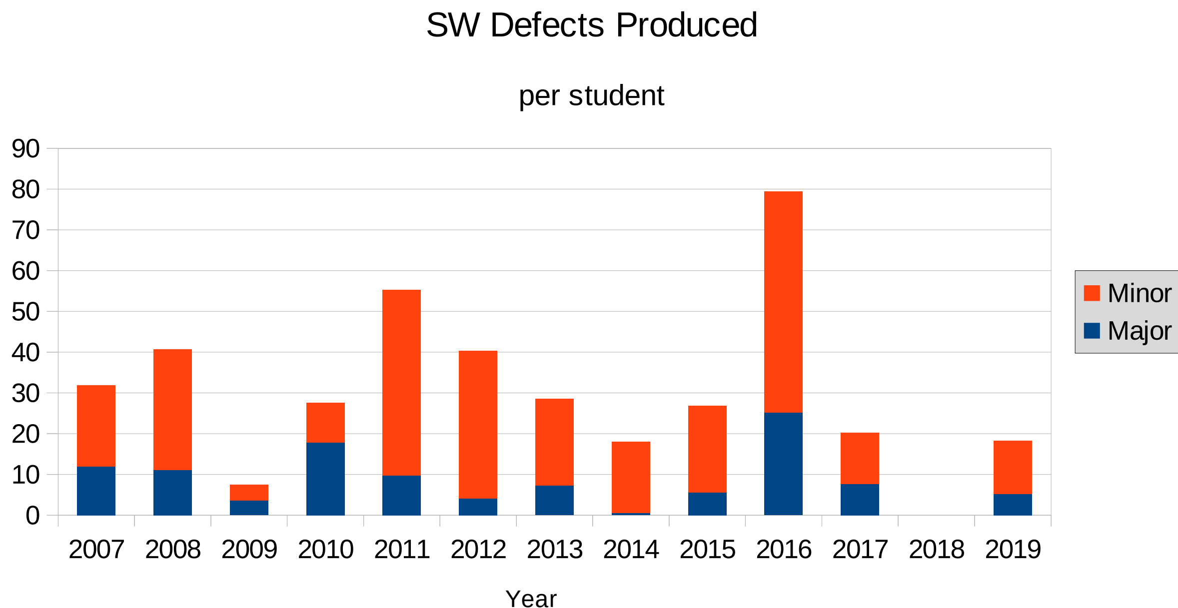

Figure 3: Average number of software defects reported by severity for period 2007-2019

The average number of software defects reported by students over the semester experience is given in Figure 3. Students were instructed to classify defects as major or minor. A major defect is one that will result in a problem that the customer will see or will cause system functionality to be compromised. Minor defects are those that include spelling errors, non-compliance with design conventions, and poor workmanship that does not lead to a major defect. Major defects are fairly obvious as being major – something is broken – and so they are more consistently reported year on year. Minor defects are a bit more subjective, so the data reported varies widely. In fact, every semester there was some number of students who reported no minor defects at all. Clearly, that is not reality.

Figures 1-3 detail student-reported metrics. Changes in the assigned lab tasks make comparison difficult. However, using the student-reported time invested in the design experience, student coding productivity and product defect density can be calculated. Industrial averages for these number exist in the literature and allow for a comparison of our novice embedded systems designers with seasoned professionals [17]. Just such a comparison was performed during the last class of the semester. Validation of their own efforts against real-world engineers was definitely well-received. Students report satisfaction with the experience and increased confidence knowing that their abilities and output are near industry norms.

Figure 4 presents the average student's coding productivity in LoC per day. The average coding productivity for the 165 student participants from 2007-2019 was 125 LoC per standard eight-hour workday.

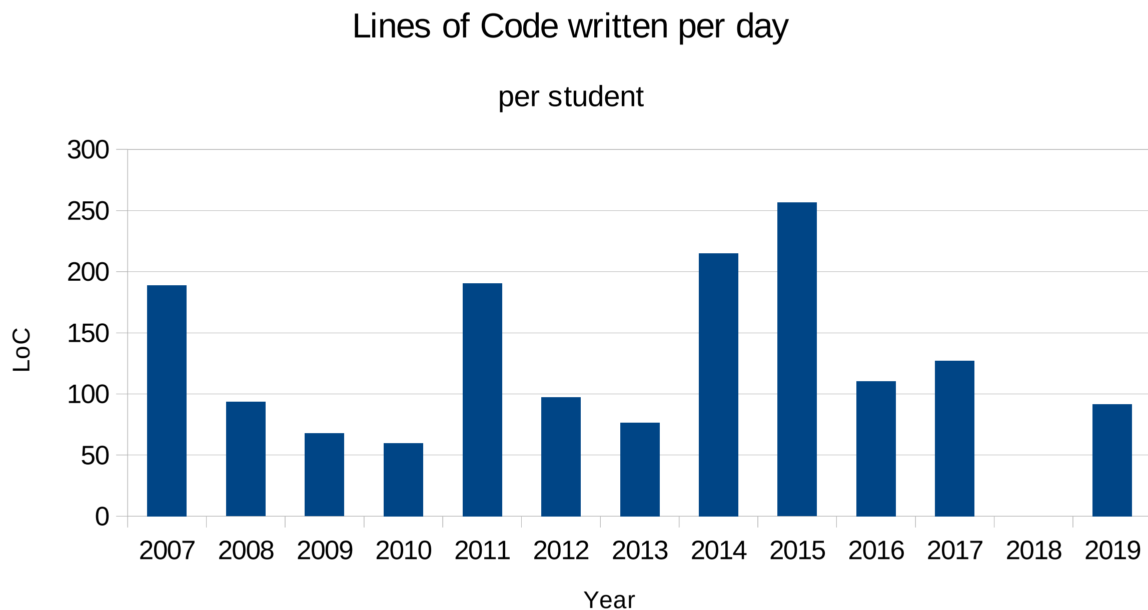

Figure 4: Average student coding productivity 2007-2019

So how many LoC do “real” programmers average per day? Brooks reported in his famous book that OS/360 programmers averaged 10 LoC per day [17]. Jones found coding productivity 16 to 38 LoC per day across a range of projects [18]. McConnell measured productivity of 20-125 LoC per day for small projects (<10 kLoC) dropping down to 1.5-25 LoC per day for large projects (10 MLoC) [19]. Anecdotal reports by a few software developers are 50-100 LoC per day determined by dividing their lifetime coding output by estimates of time spent working over their decades-long career. With these benchmark numbers in mind, the student-reported coding productivity in Figure 4 is at least reasonable but likely somewhat optimistic. Team projects averaged just over 4.5 kLoC over the 2007-2019 period. Projects of this size would be considered small by McConnell [19], and the student productivity of 125 LoC per day would be at the upper range of his findings for small projects. Data reported in years 2008-2010, 2012-2013, and 2016-2017 may be fairly accurate with the other years considered outliers. Clearer guidelines to the students on what constitutes a countable LoC would likely have improved the data in Figure 4.

Students reported their programming output and productivity at the conclusion of each design task (the semester-long design experience was divided into seven to ten design tasks). Not presented here, but worth noting is the students' maturation as software developers was often apparent in the data. Students tended to become more productive coders as the semester progresses. In general, students produced more LoC/day later in the semester than earlier. Also, some of the design tasks that were heavy on systems integration were characterized by lower coding productivity than tasks involving development of stand alone processing. This is logical as system integration design deals largely with module interfaces and is time-consuming to write, but often has relatively few LoC.

What is the relative quality of the student-generated software product? Figure 5 shows the defect density in defects per 1000 LoC over the twelve years of the course offerings. As mentioned earlier, the reported minor defect quantity varies widely, and is likely not very accurate. Assuming that major defect reporting is more accurate, the major defect density is more tightly bound. Variations seen in Figure 5 are attributed to varied reporting by students and design specification differences from year to year. Over the entire twelve year study, the average major defect rate was 7.6 defects per kLoC, the minor defect rate was 17.9 defects per kLoC, and the total defect rate was 25.4 defects per kLoC. Industry averages for defect density are 15-50 error per 1000 LoC delivered [19]. The student data reported here falls within that range. Jones reports that defect density increases as a function of project size [18]. The small project described here were always less than 10 kLoC (as reported by the students) and Jones' results would indicate a defect densities of <40 defects per kLoC. The student reports fall squarely in the middle of Jones' range for industry products.

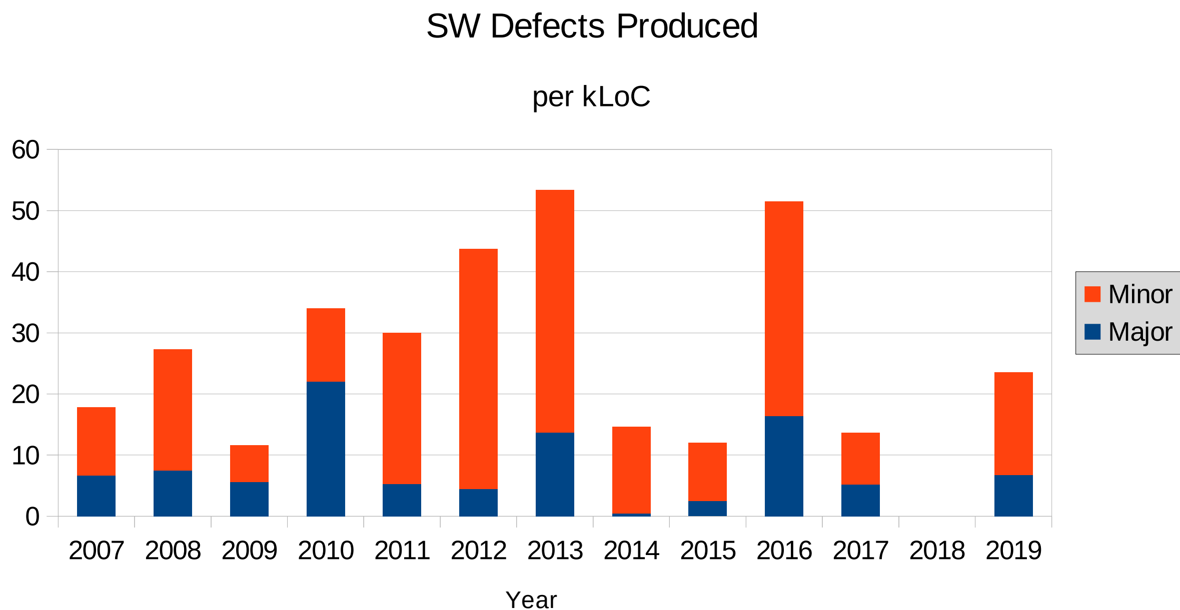

Figure 5: Software defect density as reported by severity

Discussion and Conclusions

At several points during the semester, design teams are required to forward their measures data to the instructor. Individual and team data is disseminated to the class. This disclosure allows the instructor to (i) give frequent feedback to ensure quality data collection, (ii) identify teams with a poor team dynamic, (iii) promote a friendly competition between teams to operate with maximum efficiency, and (iv) motivate engaging classroom discussion on ethical, economic, and design method issues.

As might be expected, some students resisted the design processes described here as "a complete waste of time". Students argued that designers are "born, not created". Many examples from the literature to support quantitatively the effectiveness of development process were given in counter argument. Students are asked to follow the prescribed procedure for a few weeks. A promise to discuss, evaluate, and incorporate any suggested improvements usually sways stalwart resistors (this is an excellent way to give students ownership and responsibility of their own learning). After the first design milestone during one semester, an elated team gave a class-time testimonial about how the design process and inspections identified all of their design defects in the preparation and design review meeting. The team declared the process valuable since their design worked the first time upon consolidation of their individual efforts. For the remainder of the semester, this team insisted on following the design procedures exactly. This team consistently finished assignments first and with high quality. Clearly, this team had internalized the development process and made it a part of their "professional persona". Throughout the semester, several other design teams reported similar experiences.

This paper details student-reported process and product metrics from an embedded systems design experience over an twelve year period. The course was offered in an electrical and computer engineering department and was taken by mostly computer engineering majors along with some electrical engineering majors. Occasionally, software engineering majors or computer science majors would enroll. The course utilized different processors and was offered at two different institutions. While the exact design tasks varied from year to year, the design experience involved a team-based (3 or 4 students), progressive design to specification using design inspections. Design tasks assigned include writing low-level C language routines to interface with sensors and control microcontroller peripherals, data analysis and processing software, creation or modification of operating system code, and writing Java or Python code on student laptops to process, display, or store data on client-server systems. Although the projects are ambitious, all teams were successful in completing the design requirements. Student coding productivity measures are comparable to data reported in the literature from industrial settings. Software defect production appears to be very much in the center of the range of reported industrial norms. Finally, qualitative feedback from students was mostly positive apart from concerns about heavy time investment. The authors created a design education method called gap-learning in attempt to reduce the required student effort while maintaining the design complexity. Post-semester discussions with most teams who experienced this method of learning were positive, with students reporting a more clear idea of what was expected from them and a better overall team experience. Most students report that the experience of this embedded systems course was "realistic" and "help them to know what to expect upon employment". Many

students participating the course described here accepted jobs in industry as embedded systems designers. Alumni participants report that the experience was an excellent preparation for their professional responsibilities.

References

[1] ACM/IEEE, *Computer Engineering Curricula 2016: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering*, Dec. 2016.

[2] J.W. Bruce, J.C. Harden, and R.B. Reese, “Cooperative and progressive design experience for embedded systems,” *IEEE Trans. Educ.* vol. 47, no. 1, pp. 83-92, 2004.

[3] J.W. Bruce, “Design Inspections and Software Product Metrics in an Embedded Systems Design Course”, *Proc. ASEE Annual Meeting and Exposition*, 2004.

[4] J.W. Bruce and J. Goulder, “First Look at an Internet-enabled Embedded Systems Design Course”, *Proc. ASEE Annual Meeting and Exposition*, 2005.

[5] R.B. Reese, “Embedded System Emphasis in an Introductory Microprocessor Course”, *Proc. ASEE Annual Meeting and Exposition*, 2005.

[6] J.W. Bruce and L. Hathcock, “An Approach For Vertically Integrated Embedded Systems Design”, *Proc. ASEE Annual Meeting and Exposition*, 2008.

[7] J.W. Bruce and R.A. Taylor, “A Using Information Gap Learning Techniques in Embedded Systems Design Education”, *Proc. ASEE Annual Meeting and Exposition*, 2017.

[8] IEEE Std. 1028-1997, IEEE Standard on Software Reviews. Section 6.

[9] T. Gilb and D. Graham, Software Inspections. Addison-Wesley, 1993.

[10] IEEE Std. 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software.

[11] R.S. Pressman, *Software Engineering: A Practitioner’s Approach 5/e*, Mc-Graw Hill, 2001.

[12] http://www.sics.se/~adam/pt/

[13] http://en.wikipedia.org/wiki/Duff's_device

[14] C. Jones, Programming Productivity, McGraw-Hill, 1986.

[15] M. Konrad, L.M. Joseph, and E. Eveleigh, E. “A meta-analytic review of guided notes.”, *Education and Treatment of Children*, 32, 421-444, 2009.

[16] B.D. Lazarus, “Flexible skeletons: Guided notes for adolescents with mild disabilities”, *Teaching Exceptional Children*, 28(3), 36-40, 1996.

[17] F. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1995.

[18] C. Jones and O. Bonsignour, *The Economics of Software Quality*, Addison-Wesley, 2011.

[19] S. McConnell, *Code Complete, 2/e*, Microsoft Press, 2004.

[20] https://www.yegor256.com/2014/04/11/cost-of-loc.html