# The Journal of Computing Sciences in Colleges

**Papers of the 26th Annual CCSC Midwestern Conference**

October 4th-5th, 2019
Benedictine University
Lisle, IL

Baochuan Lu, Editor
Southwest Baptist University

Saleh Alnaeli, Regional Editor
University of Wisconsin-Stout

**Volume 35, Number 5**

**October 2019**

*The Journal of Computing Sciences in Colleges* (ISSN 1937-4771 print, 1937-4763 digital) is published at least six times per year and constitutes the refereed papers of regional conferences sponsored by the Consortium for Computing Sciences in Colleges. Printed in the USA. POSTMASTER: Send address changes to Susan Dean, CCSC Membership Secretary, 89 Stockton Ave, Walton, NY 13856.

## Table of Contents

# The Consortium for Computing Sciences in Colleges Board of Directors

## CCSC National Partners

The Consortium is very happy to have the following as National Partners. If you have the opportunity please thank them for their support of computing in teaching institutions. As National Partners they are invited to participate in our regional conferences. Visit with their representatives there.

### Platinum Partner

*Turingscraft*
*Google for Education*
*GitHub*
*NSF – National Science Foundation*

### Silver Partners

*zyBooks*

#### Bronze Partners

*National Center for Women and Information Technology*
*Teradata*
*Mercury Learning and Information*
*Mercy College*

## Foreword

The following five CCSC conferences will take place this fall.

| | |
|---|---|
| Midwestern Conference | October 4-5, 2019 |
| | Benedictine University in Lisle, IL |
| Northwestern Conference | October 4–5, 2019 |
| | Pacific University, Forest Grove, OR |
| Rocky Mountain Conference | October 11-12, 2019 |
| | University of Sioux Falls in Sioux Falls, SD |
| Eastern Conference | October 25-26, 2019 |
| | Robert Morris University in Moon Township, PA |
| Southeastern Conference | October 25-26, 2019 |
| | Auburn University in Auburn, AL |

The papers and talks cover a wide variety of topics that are current, exciting, and relevant to us as computer science educators. We publish papers and abstracts from the conferences in our JCSC journal. You will get the links to the digital journals in your CCSC membership email. You can also find the journal issues in the ACM digital library and in print on Amazon.

Since this spring we have switched to Latex for final manuscript submission. The transition has been smooth. Authors and regional editors have worked hard to adapt to the change, which made my life a lot easier.

The CCSC board of directors have decided to deposit DOIs for all peer-reviewed papers we publish. With the DOIs others will be able to cite your work in the most accurate and reliable way.

Baochuan Lu
Southwest Baptist University
CCSC Publications Chair

## Welcome to the 2019 CCSC Midwestern Conference

Welcome to the 26th CCSC Midwestern Conference held on October 4-5, 2019 at beautiful Benedictine University in Lisle, IL. It is my pleasure to be the Chair for this conference for a second time. Much has changed since I chaired this conference fifteen years ago. There are new languages like Kotlin, Rust, and Go. Software engineering has become a discipline of its own with cybersecurity well on its way. Interactive lab environments with auto-grading are now ubiquitous. One thing that has not changed is that the CCSC conferences provide a fertile ground for sharing ideas and techniques in our ever-changing world. There is always something for everyone, and I hope you will find something that will be helpful to you.

This year we have a strong program of speakers, papers, panels, tutorials, and workshops. We accepted 9 papers out of 15 submissions for a 60% acceptance rate. Our keynote speaker is Roman Lysecky who is a noted Professor in Electrical and Computer Engineering at the University of Arizona and Head of Content at zyBooks. He will be speaking on "Improving CS/CE Education: Recent Research and Experiences". Our banquet speaker is Casey O'Donnell who is an Associate Professor in Media and Information at Michigan State Unviersity. His talk will be on "Friendship Games: Designing Feedback Loops that Foster Friendship".

In addition, we accepted 3 tutorials. Our pre-conference workshop is a session by National Partner Google for Education and will explore using Google Cloud servers and databases in the classroom. In addition, there will be several other National Partner vendor sessions. Our student program includes a project showcase competition and a programming contest as well as an employer panel.

This conference as with all the CCSC conferences would not be possible without volunteer efforts of the Conference Committee and the reviewers. Their names and roles can be found in proceedings. Thank you for the work done to make this conference a success. If anyone would like to become part of this group, we are always looking for new volunteers.

Deborah Hwang
University of Evansville
Conference Chair

## 2019 CCSC Midwestern Steering Committee

## 2019 CCSC Midwestern Conference Committee

## Reviewers — 2019 CCSC Midwestern Conference

| | |
|---|---|
| David Bunde | Knox College, Galesburg, IL |
| Kevin Coogan | Blackburn College, Carlinville, IL |
| J. William Cupp | University of Alabama at Birmingham, Birmingham, AL |
| Lawrence D'Antonio | Ramapo College, Mahwah, NJ |
| John Finnegan | Purdue Polytechnic New Albany, New Albany, IN |
| Richard Fox | Northern Kentucky University, Highland Heights, KY |
| Paul Gestwicki | Ball State University, Muncie, IN |
| Janet Helwig | Dominican University, River Forest, IL |
| Mickey Hendon | Bloomsburg University, Bloomsburg, PA |
| Mahmood Hossain | Fairmont State University, Fairmont, WV |
| Brian Howard | DePauw University, Greencastle, IN |
| Deborah Hwang | University of Evansville, Evansville, IN |
| Zachary Kurmas | Grand Valley State University, Allendale, MI |
| David Largent | Ball State University, Muncie, IN |
| Jeffrey Lehman | Huntington University, Huntington, IN |
| Qian Liu | Rhode Island College, Providence, RI |
| Mohamed Lotfy | Regis University, Denver, CO |
| Saverio Perugini | University of Dayton, Dayton, OH |
| Nicholas Rosasco | Valparaiso University, Valparaiso, IN |
| Nancy Van Cleave | EIU, Charleston, IL |
| Greg Wolffe | Grand Valley State University, Allendale, MI |
| David Woods | Miami University, Hamilton, OH |
| Yingbing Yu | Austin Peay State University, Clarksville, TN |

# Improving CS/CE Education: Recent Research and Experiences[*]

**Keynote Speech**

*Roman Lysecky*
*Electrical and Computer Engineering, University of Arizona*
*Head of Content, zyBooks - A Wiley Brand*
`rlysecky@email.arizona.edu`

## Abstract

Online active-learning content and program auto-grading with immediate feedback have enabled new approaches to teaching lower-division computer science/engineering courses. Having started with the goal of reducing failure rates in lower-division CS/CE courses by replacing existing textbooks/homework with web-native, integrated, active-learning content, zyBooks now cover more than 17 CS/CE courses and have been used by more than 600 universities and 500,000 students. This talk briefly introduces the web-native, active-learning learning content that consists of aggressively-minimized text, animations, interactive learning questions, and auto-graded programming assignments.

We summarize published research findings that highlight results on student learning outcomes, student earnestness in completing reading activities, student struggle rates and stress, and student engagement in class. We further present a case study of using many small programming labs in an introductory programming courses instead of the more traditional one large programming assignment per week, which resulted in higher performance on exams, reduced student stress, and continued performance in subsequent courses.

[*]Copyright is held by the author/owner.

## Speaker Bio

Roman Lysecky is a Professor of Electrical and Computer Engineering at the University of Arizona and Head of Content at zyBooks - A Wiley Brand. He received his Ph.D. in Computer Science from the University of California, Riverside in 2005. His research focuses on embedded systems with emphasis on medical device security and on computer science/engineering education. He is an inventor on one US patent. He has authored eight textbooks and contributed to several more on topics including C, C++, Java, Data Structures, Digital Design, VHDL, Verilog, Web Programming, and Computer Systems. His recent books with zyBooks utilize a web-native, active-learning approach that has shown measurable increases in student learning and course grades. He has also authored more than 100 research publications in top journals and conferences. His research has been supported by the National Science Foundation (including a CAREER award in 2009), the Army Research Office, the Air Force Office of Scientific Research, and companies such as Toyota. He received the Outstanding Ph.D. Dissertation Award from the European Design and Automation Association (EDAA) in 2006, eight Best Paper Awards, and multiple awards for Excellence at the Student Interface from the College of Engineering at the University of Arizona.

# Friendship Games: Designing Feedback Loops that Foster Friendship[*]

## Banquet Speech

*Casey O'Donnell*
*Michigan State University*

Friendships are deeply important to human health and happiness. Research shows that, “individuals with adequate social relationships have a 50% greater likelihood of survival compared to those with poor or insufficient social relationships. The magnitude of this effect is comparable with quitting smoking and it exceeds many well-known risk factors for mortality (e.g., obesity, physical inactivity).” (Holt-Lunstad et al., 2010) Friendship is also strongly correlated with increased “life satisfaction” (Amati et al., 2018) and depression is lower overall for people with rich friend networks (Helliwell Putnam, 2004).

With that said, the majority of games (and other kinds of applications as well) actually avoid the kinds of activities that lead to richer and deeper connections or those that help us make new friends. One can simply look to something like Facebook social games or massively multiplayer online games where connections do not necessarily encourage players to interact with one another in any meaningful way. Players often enter and leave games as strangers. While friendships may form, it is often an accident or byproduct of a game. Or players enter as friends and simply share the space as they might any other.

In the design of Friendship Games, one must examine: designing for consent, designing for low trust activities that can be scaled up to higher level trust activities, building opportunities for consensual reciprocity through reciprocation loops, then link these loops together through an escalating structure to provide players an opportunity to build, connect and maintain connections with one another over time. Often times this focuses on building games for much smaller collaborative groups. Players clustered together into "cohorts" have more reasons to connect with one another through shared goals and situations. It becomes possible through closely linking players through high-concurrency events or asynchronous activities players work together to solve problems or share experiences.

[*]Copyright is held by the author/owner.

Players are intrinsically motivated to build new friendships and meet new people, particularly when their various social needs are not already being met. Friendships are long-term relationships, based on trust and shared values and are mutually beneficial. Without designing for friendship building, players treat one another as interchangeable, disposable or abusable. By making games that build connections between players, designers work to improve the lives of players. There is certainly nothing more human than our connections to others. If it is possible to design systems in such a way that it fosters friendship rather than suppress it, isn't there a moral obligation for designers to pursue that goal?

## References

[1] Viviana Amati, Silvia Meggiolaro, Giulia Rivellini, and Susanna Zaccarin. Social relations and life satisfaction: The role of friends. *Genus*, 74(1):7, 2018.

[2] John F Helliwell and Robert D Putnam. The social context of well–being. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 359(1449):1435–1446, 2004.

[3] Julianne Holt-Lunstad, Timothy B Smith, and J Bradley Layton. Social relationships and mortality risk: a meta-analytic review. *PLoS medicine*, 7(7):e1000316, 2010.

# Examine the Adoption of a Virtual World to Improve the Hybrid Courses[*]

*Imad Al Saeed*
*Computer Science Department*
*Saint Xavier University*
*Chicago, IL 60655*
*alsaeed@sxu.edu*

**Abstract**

The purpose of this study is to examine the effectiveness of adopting a virtual world's environment to enhance the teaching and learning processes in hybrid courses. The study includes a target population of graduate students of both genders who are studying Mobile Applications Programming in the summer 2017 and the summer 2018 semesters at Saint Xavier University. The general findings indicate that adopting a virtual world's environment, such as Second life, increases students' engagement, encourages them to ask questions, enhances their participation within the class discussion activities, and improves their grades.

## 1 Introduction

The Computer Science department (CS) at Saint Xavier University offers face-to-face courses and full internet courses (no campus meetings) during the Fall and Spring semesters as 16-week courses. To provide students with a more flexible schedule, existing 16-weeks courses, such as Software Engineering, Data Visualization, and Mobil Applications were re-designed and delivered in a hybrid format in the summer semesters only as 8-week courses. Hybrid courses are a combination of traditional face-to-face and online activities [12]. Hybrid courses are the best alternative solution that provide the opportunity for face-to-face interaction and the benefit of reducing the classroom's seat time [12].

[*]Copyright ©2019 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

The CS department encourages faculty to develop the 8-weeks hybrid courses version to cover the exact same materials as a 16-week course. The purpose of this research study is to examine the adoption of the Virtual World (VW) to enhance the teaching and learning processes in hybrid courses.

## 2 Related Work

Unlike online courses, hybrid courses are a combination of a traditional face-to-face class with online discussion activities that provide the opportunity for students to contact their course instructors and a reduced-contact schedule at the same time [2, 14]. The study published by the Learning Technology Center in 2014 [9] indicates that hybrid courses share three important features: "(an) online learning activities are used to complement in-person activities; (b) time in the classroom is reduced, but not eliminated; and (c) online and in-person instructional elements of the courses are designed to interact and benefit from the strengths of each." This research study followed and implemented these three key distinguishing features in designing the pilot hybrid course for this experimental study.

In 2010, Ertmer et al. [3] published a paper that examines the online interactive discussion and how it improves education in computer science courses. The results of their experiment provide insights on how the online discussion provides students with the opportunity to communicate outside the physical classroom and build an effective online community [7]. On the other hand, online discussions through Canvas, Backboard, Moodle, and/or any other similar content management system provides the opportunity for students to recognize personal knowledge gaps [3, 10].

It is true that hybrid courses are the fastest growing courses in higher education, but it is a challenge to design such courses and provide online discussions that interest all students at all knowledge levels [1]. In 2008, Jackson  Helms [6] published a paper that focused on student perceptions in hybrid courses. The authors used strengths, weaknesses, opportunities, and threats methodology to examine if traditional hybrid formats meet student expectations. They argued that it is very difficult to maintain student engagement and develop suitable online learning experiences. The researchers concluded that hybrid courses have been criticized for not offering enough teacher-student interaction or student-student cooperation. Their findings indicate that students are not able to understand what they are supposed to do with the online discussion assignments and how to find the answers for the discussion questions [5, 6]. Their findings suggest that students still need to interact with their instructors and other students and ask questions because most of them cannot work independently.

On the other hand, there is a big concern about the course completion rates and student satisfaction [1, 5]. According to the 2013 Managing Online Education Survey [4], conducted by the Western Interstate Commission for Higher Education, completion rates of hybrid courses are approximately 5% below the face-to-face courses [12]. Since there are many factors for the lower completion rate, the reduced interaction time and higher degree of self-motivation might be one of the essential factors

To succeed, hybrid courses must be improved in a way that can provide more interaction time for students. This research study responded to these studies by examining the adoption of an open-source virtual world (Second Life) to be used as an effective and cost-efficient tool to increase the interaction time between instructor-student and student-student in hybrid courses.

## 3 Why Virtual World?

The Virtual World, such as Second Life, is a unique, low cost, and safe online environment for teaching and learning purposes that could be used for supporting and training distributed groups through the virtual classroom simulation [11]. Currently, the virtual worlds are widely used for educational purposes to facilitate student learning activities [8]. The Virtual worlds allow users to create their own avatars, which are referred to their residents and offer them the platform they need to interact with each other easily. The Virtual World is a playground for imagination because it expands the boundaries of users' creativity in exploring, creating, designing, modeling real environments, building, coding, document sharing, recording facilities, and collaboration [13].

## 4 Approach

Typical weekly discussion board (DB) assignments at a graduate level were posted with the weekly modules in Canvas. Students are required to write 600 to 800 words that respond to the discussion questions with their thoughts, ideas, and comments. The DB assignments are related to the focusing topic(s) and the reading assignments for that week. Students are required to research, follow APA guidelines with their academic writing, provide an in-text citation, and respond to at least two of their fellow classmates with at least a 100-word reply for each response. The study assumes that the students know how to write using APA styles and how to find peer viewed references. Below are two examples of the DB assignments.

## 5 Example Assignments

### 5.1 Discussion Board Assignment1

Write 600–800 words that respond to the following questions with your thoughts, ideas, and comments. Use the school library and the Internet to answer the following questions:

1. Research a model of a popular Android smartphone or tablet device and write a paragraph on this device's features, specifications, price, and manufacturer.
2. Name five Android mobile device features not mentioned in Chapter 1.
3. What is the current annual cost for a developer's account at the Windows Store? Even though the mobile side of the Windows Store is marginal, why are many app developers learning to create Windows Store apps?

### 5.2 Discussion Board Assignment2

Write 600–800 words that respond to the following questions with your thoughts, ideas, and comments. Use the library and the Internet to answer the following questions:

1. Linear and Relative layouts are not the only types of Android layouts. Name three other types of layouts and write a paragraph describing each type.
2. How much does an average Android app developer profit from his or her apps? Research this topic and write 150–200 words on your findings.
3. Research the most expensive Android apps currently available at Google Play. Name three expensive apps, their price, and the purpose of each.

## 6 Hypotheses

This study focuses on the online discussion activities portion of the hybrid course. There are eight DB assignments involved in the course. The research study hypothesizes the following:

1. Students who are accepted to participate within the experimental study, attend the Virtual World meeting, participate in the online discussion activities, and submit their written responses to the discussion questions through Canvas score better than other students who are in the traditional hybrid course format.
2. Students who are in a traditional hybrid classroom setting and answer the DB independently score lower than students who participate in the experimental study.

To address these hypotheses, measuring the learning is needed. For this study, students were given the DB questions and were looked at the quality and timing of their initial responses and their responses to their peers in both classroom settings (with and without the use of VW). The research study assumed that the DB questions were written well, assessed what they are intended to assess, and graded fairly. Also, the research study assumed that all students followed the school policy on academic integrity in doing their own work by themselves.

To eliminate any bias, two experienced faculty members with Mobile Applications Programming were employed to grade the students' DB assignments by using the rubric in Table 1.

Table 1: Discussion Board Assignments Rubric

| Criteria | Pts |
|---|---|
| Task Requirements | 30.0 pts |
| Demonstrate and application of knowledge | 50.0 pts |
| Academic writing and format | 20.0 pts |

All of the students' names were hidden from the two expert teachers. For example, each participant was assigned a randomized ID number to keep the results coordinated and anonymous. The students' grades collected from both classroom settings (hybrid course with and without VW) were compared and used to test the research hypotheses. Student participation in this study is strictly voluntary. Participants may also withdraw from the study at any time, even if they signed the consent paperwork. Students who choose not to participate will follow the traditional hybrid class approach to complete their weekly class activities including the online discussion assignments.

## 7 Logistics of Data Collection

The data consists of three categories: The DB assignment rubric (see table1), assess the students' understanding of the subject matter through answering the assigned questions and responding to their peers, and responses to the survey's questionnaires. The questionnaire is administered to the students by the end of the course. The researcher used an online publication within a Web-based survey engine (survey monkey) in an HTML format to distribute the questionnaire (as open-ended questions) to the students. The score data was assembled and anonymized after the grades are in. The data does not identify the students. No personally identifiable data was collected.

## 8 Logistics of Data Analysis

Since this is a qualitative research, rigorous and systematic data collection methods were used to analyze the data. Open-ended questions were used to gain further detail about the research problems and research goals. Nvivo studio will be used for data analysis. The records of this study were kept private. In any report of this study that might be published, the researcher did not include any information that will make it possible to identify students.

## 9 Experiment

The experiment took place in the summer of 2018 semester using Mobile Apps as a pilot course. There are two sections of the course, and each section consisted of 20 students (total of 40 students). The study compared the student's grades collected from Summer 2018 semester (with VW) with the student's grades collected from Summer of 2017 semester (without VW) for the same hybrid course. All 40 students who enrolled with Summer 2018 semester were chosen to participate in the research study and sign the consent form.

The online discussion activities were conducted at the Virtual World environment - Second Life. Second life is an open source online virtual environment that can be customized in different ways [13]. No letter of approval to conduct the off-site research was needed. The researcher rented a land (the Private Region within Second Life) that runs on servers hosted specifically by the Linden Lab to build his classroom environment and pay the (see 1). No one can access the researcher's land without his permission. All access requires a login token to establish the identity and permissions, including land access. This protects the students' anonymity as well. The researcher used the Second Life tools and the Linden Lab programming language (mixed between Java and C++) to create a secure virtual classroom environment and provided an access code for the students to access to the virtual classroom in Second Life. Each student received his/her own code that cannot be shared with others. To ensure privacy, participants are required to create their own avatars with fake names. The participants will be attending a secured virtual classroom using their avatars in Second Life. No outside users can access, see, or listen to other classroom discussions.

A weekly meeting was conducted in Second life (see 2). The researcher discussed the subject matter and ensured the students' participation through the following methods: Lecturing, Demonstrating, Collaborating, Classroom discussion, and Debriefing. The researcher provided discussion questions, explained them to the students, and the posted them on Canvas. Students attended the meeting using their own avatars, listened to the lecture, participated in the class discussion, asked questions about the DB assignments, and participated with other classroom activities. All students were exposed to the same lecture and assigned the same discussion questions. Students submitted their written responses on the Canvas course to be graded.

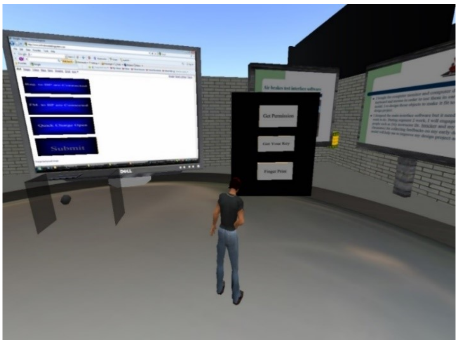


Figure 1: Classroom Initial Setup in Second Life.

Figure 2: Students Attending Online VW Classroom.

All students' initial post and their peers' responds were collected and sent to the two expert faculty members for grading without students' identifications. To collect enough data, all students were surveyed. Survey responses were keyed to individual students so that results may be coordinated with other data in the research project. All student names were replaced by random numeric keys to anonymize the data.

## 10 Results

In this analysis, the study included eight DB assignments (one DB assignment per week) graded by two experienced teachers. The reason for that is to examine students' knowledge level with all DB assignments in the course and to eliminate any bias for choosing specific discussion DB than the others. To evaluate the hypotheses, the study examined the proportion of the students who did well on responding to the DB questions. "Did Well on the DB assignment" was defined as scoring at 80% or above, and "Did Poorly on the DB assignment" as scoring 60% or below. Tables 2, 3, 4 and 5 below showed the average students grades for the two semesters graded by two expert semesters.

Table 2: Summer 2018 semester – section 1 (with VW)

| **No. of students** | **20** | **20** | **20** | **20** | **20** | **20** | **20** | **20** |
|---|---|---|---|---|---|---|---|---|
| **DB assignments** | Week1 | Week2 | Week3 | Week4 | Week5 | Week6 | Week7 | Week 8 |
| **Instructor 1** | 82% | 88% | 85% | 83% | 87% | 85% | 92% | 97% |
| **Instructor 2** | 80% | 84% | 87% | 86% | 85% | 90% | 89% | 95% |

Table 3: Summer 2018 semester – section 2 (with VW)

| **No. of students** | **20** | **20** | **20** | **20** | **20** | **20** | **20** | **20** |
|---|---|---|---|---|---|---|---|---|
| **DB assignments** | Week1 | Week2 | Week3 | Week4 | Week5 | Week6 | Week7 | Week8 |
| **Instructor 1** | 79% | 82% | 80% | 88% | 87% | 82% | 92% | 94% |
| **Instructor 2** | 81% | 84% | 81% | 87% | 85% | 88% | 95% | 97% |

Table 4: Summer 2017 semester – section 1 (without VW)

| **No. of students** | **15** | **17** | **18** | **15** | **19** | **19** | **17** | **19** |
|---|---|---|---|---|---|---|---|---|
| **DB assignments** | Week1 | Week2 | Week3 | Week4 | Week5 | Week6 | Week7 | Week 8 |
| **Instructor 1** | 70% | 66% | 75% | 77% | 79% | 78% | 82% | 82% |
| **Instructor 2** | 66% | 68% | 69% | 78% | 79% | 81% | 82% | 83% |

For statistical analysis, a repeated-measure ANOVA was used on the assignment score for students who completed all eight DB assignments (n=40). See tables 2, 3, 4 and 5 for mean DB scores. The analysis showed that the distribution of scores on these DB assignments for students who participate in the weekly virtual meeting was significantly different ($p<0.001$). The distribution of score and the statistical analysis supported hypotheses 1. Also, the analysis showed that the distribution of scores on these DB assignments for students in

Table 5: Summer 2017 semester – section 2 (without VW)

| **No. of students** | **20** | **18** | **18** | **15** | **19** | **19** | **18** | **19** |
|---|---|---|---|---|---|---|---|---|
| **DB assignments** | Week1 | Week2 | Week3 | Week4 | Week5 | Week6 | Week7 | Week8 |
| **Instructor 1** | 74% | 79% | 67% | 78% | 72% | 80% | 82% | 80% |
| **Instructor 2** | 71% | 76% | 71% | 80% | 71% | 81% | 81% | 81% |

a traditional hybrid course was significant ($P<0.001$). The distribution of score and the statistical analysis supported hypotheses 2. In other words, students who attended the virtual weekly meeting scored better on the DB assignment than students with regular hybrid classroom setup (without VW). The results also showed that not all students who enrolled in the Summer 2017 semester (without VW) submitted their initial posts for the DB questions each week compared to students who enrolled in summer 2018 semester (with VW).

The survey asked the students ten open-ended equations about the experiments. All students who participated in this experimental study were required to answer the survey questions. Qualitative analysis showed that students were satisfied with the weekly meeting because it gave them a chance to interact with their course instructor and their peers and ask them questions to enforce with what they learned during the virtual online discussion. For example, one student indicated that "participating in the VW assisted me asking questions directly with no hesitating and it is not boring like traditional." Another student indicated that "VW helped boost understanding and more of a feeling of a class participating together rather than just replying one at a time to the presentation... [and] The Virtual world discussion gave us interest in doing the discussion rather than participating in typical traditional online discussion." Another student responded that "the online virtual discussion has saved me a lot of time looking for answers. So, I asked questions and received an immediate answer from my teacher." This is considered as evidence that hybrid courses with VW has met the students' expectations and increased their knowledge at the same time. The only drawback was the learning time required to learn how to navigate the second life and master the use of Second Life tools. For example, technical problems such as the internet connection, the quality of the microphone, and the quality of the computer speakers may cause some issues for a few students.

## 11 Conclusion

The essential purpose of this research study is to examine the effectiveness of adopting a virtual world's environment within a hybrid course to enhance the teaching and learning processes of computer programming concepts using virtual discussions. The results of this experiment showed an indication that adopting VW such as Second Life has enhanced students' knowledge within the subject matter, increased their engagements in class, encouraged to ask questions, and improved their grades. The study also showed evidence that hybrid courses were improved in a way that can provide more interaction time for students using VW. The finding also indicated that the only potential drawbacks include the learning time required and technological issues involved in using the online virtual environment. This research study provides another way for students to have a highly personalized learning experience that enables them to improve their understanding and confidence related to online activities.

## References

[1] Stephanie Babb, Cynthia Stewart, Ruth Johnson, et al. Constructing communication in blended learning environments: Students' perceptions of good practice in hybrid courses. *MERLOT Journal of Online Learning and Teaching*, 6(4):735–753, 2010.

[2] The Sloan Consortium. Blended/hybrid courses. `http://commons.sloanconsortium.org/discussion/blendedhybrid-courses` Retrieved on March 31st, 2014.

[3] Peggy A Ertmer, Ayesha Sadaf, and David J Ertmer. Student-content interactions in online courses: The role of question prompts in facilitating higher-level engagement with course content. *Journal of Computing in Higher Education*, 23(2-3):157, 2011.

[4] WICHE Cooperative for Educational Technologies. Managing online education 2013: Practices in ensuring quality. `http://wcet.wiche.edu/wcet/docs/moe/2013ManagingOnlineEducationSurveyFinalResults.pdf` Retrieved on March 9th, 2019.

[5] Stan G Guidera. Perceptions of the effectiveness of online instruction in terms of the seven principles of effective undergraduate education. *Journal of Educational Technology Systems*, 32(2-3):139–178, 2003.

[6] Mary Jo Jackson and Marilyn M Helms. Student perceptions of hybrid courses: Measuring and interpreting quality. *Journal of Education for Business*, 84(1):7–12, 2008.

[7] Radu P Mihail, Beth Rubin, and Judy Goldsmith. Online discussions: Improving education in CS? In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 409–414. ACM, 2014.

[8] Shailey Minocha and Christopher Leslie Hardy. Designing navigation and wayfinding in 3D virtual learning spaces. In *Proceedings of the 23rd Australian Computer-Human Interaction Conference*, pages 211–220. ACM, 2011.

[9] University of Wisconsin Milwaukee. Hybrid courses. `http://www4.uwm.edu/ltc/hybrid/about_hybrid/index.cfm` Retrieved on March 10th, 2019.

[10] Murat Oztok, Daniel Zingaro, Clare Brett, and Jim Hewitt. Exploring asynchronous and synchronous tool use in online courses. *Computers & Education*, 60(1):87–94, 2013.

[11] Sarah Smith-Robbins. Are virtual worlds (still) relevant in education? *eLearn*, 2011(12), December 2011.

[12] Jeffrey A Stone and Tricia K Clark. Experiences with a hybrid CS1 for non-majors. *Journal of Computing Sciences in Colleges*, 30(3):47–53, 2015.

[13] Maria Vargas-Vera. Reflections on the Second Life platform used in the development of a virtual university campus. *International Journal of Knowledge Society Research (IJKSR)*, 7(4):12–23, 2016.

[14] Barbara Ward. The best of both worlds: A hybrid statistics course. *Journal of Statistics Education*, 12(3), 2004.

# Using Real Examples to Motivate Automata Theory[*]

*David P. Bunde*
*Computer Science Department*
*Knox College*
*Galesburg, IL 61401*
`dbunde@knox.edu`

**Abstract**

Many institutions have a course on automata theory that studies the limits of computation by examining successive computational models, including deterministic finite automata (DFAs), context-free grammars (CFGs), and Turing machines (TMs). Some students resist this material because they see it as overly theoretical, but much of it has important practical applications. In this position paper, we discuss some of these and advocate a view of the course with applications emphasized to provide motivation.

## 1 Introduction

In the automata theory class included in many CS degree programs, the emphasis is on fundamental questions such as "What is computing?" and "What is computable?". The subject also asks about the power of non-determinism, which is a key question in polynomial-time complexity and the study of algorithms. These are important questions, but students sometimes struggle to see the motivation for this abstract material.

With the issue of motivation in mind, we propose a different approach to teaching automata theory, one which foregrounds applications and practical concerns even as it covers the standard material. This approach recognizes that the material also has important uses in text processing (using regular

[*]Copyright ©2019 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

expressions) and parsing. By emphasizing these applications, we attempt to make the course more accessible to students with an applied focus while still teaching abstract topics like computability and non-determinism.

This paper presents our efforts to follow this approach; we have taught using it several times and endeavor to increase the number and prominence of applications each time. Due to the course's evolving nature and the relative infrequency with which we have taught it (roughly every other year), we do not have formal evidence of the effectiveness of our approach. Thus we present our ideas as a position paper.

The remainder of the paper discusses our ideas roughly in the order in which they are used in a standard course, followed by brief discussions of the context of our course, related work, and possible future directions.

## 2 Regular languages

We think of an automata theory course as organized into 3 parts, each corresponding to a different class of languages and machines. The first part covers regular languages, represented by regular expressions, deterministic finite automata (DFAs), and non-deterministic finite automata (NFAs). Thus, we begin by presenting applications related to this material.

**Practical notation for regular expressions.** Regular languages are actually easy to motivate because many programming languages implement a version of them, though with a slightly different notation than in standard automata textbooks. The latter define a regular expression using three base cases: $\varnothing$ (the empty set), $\epsilon$ (the empty word), and a single character. Then, regular expressions can be combined using operators for choice ($+$) and repetition ($*$).

Unfortunately, this notation is somewhat different than that used in regular expressions as implemented in practical programming languages. This forms an unnecessary barrier for students who have already used regular expressions and also makes it harder for students learning about them in the class to take their skills outside. Thus, we change the standard notation to something closer to that used in languages such as Python. Specifically, we adopted $|$ rather than $+$ for a choice, but added $\epsilon$ and $\varnothing$ since the languages $\{\epsilon\}$ and $\{\}$ are otherwise difficult or impossible to represent. Having introduced regular expressions as a practical tool, it then becomes natural to show that other common regular expression notation is syntactic sugar. For example, other types of repetition available in Python can be represented using our limited set of operators:

| Symbol | Meaning | Example | Equivalent to |
|---|---|---|---|
| ? | 0 or 1 times | a? | a \| $\epsilon$ |
| + | 1 or more times | a+ | aa* |
| {n} | n times | a{5} | aaaaa |
| {n,m} | n to m times | a{1,3} | a \| (aa) \| (aaa) |

These become examples done in class or they could be used as homework problems.

Another useful type of syntactic sugar are character classes. For example, [a-z] matches any single lowercase letter and [A-Za-z] matches any letter. Students are quick to see that these expressions can easily be created using a list of the options, but appreciate the concise representation provided by the character class. More puzzling, but equivalent in a finite alphabet, are character classes with negations: [^a-z] matches any single character that is not a lowercase letter. Again, all of these can be used as examples or exercises while giving students more powerful regular expression notation for use either in class or in their programs.

**Practical regular languages.** Although the syntactic sugar discussed above does not technically increase the power of regular expressions, using it (particularly character classes) does make it reasonable to express regular languages of practical importance. An obvious possibility is programming language identifiers, which have rules like needing to start with a letter. Limited ranges of numbers are also natural and constitute a nice challenge; creating a regular expression for a number with a fixed number of digits (like 2) is easy, but making one for a range of values like 0–255 (one part of an IPv4 address) is surprisingly tricky since the values allowed for each digit depend on the values of the previous digits. One regular expression for numbers in this range (without leading 0s) is

([1-9]?[0-9]) | (1[0-9][0-9]) | (2[0-4][0-9]) | (25[0-5])

The separation into multiple terms prevents (i) leading 0s and (ii) a ones digit above 5 in three digit numbers if the first two digits are 25. Obviously, there are other ways to create regular expressions for this language, but all of them seem to be surprisingly complicated. This is something for the instructor to be careful about; the first time the author used numbers as an example, he asked for the range of a `short` in C (-32,768 to 32,767), which requires a very complicated expression.

**Code counting.** Our favorite example of using a DFA is writing a program to count lines of code (LOC). Integrated editors will provide a line count, as will utilities such as Unix's `wc`, but their counts include lines that are blank or contain only comments. Instead, we ask students to write a program or give a DFA with output that counts lines while excluding blank lines and Java comments (`//` to ignore the rest of the line and `/*` to ignore until the next `*/`). Consider the following Java code:

```
 1  public class TestProgram \{
 2      public static void main(String[] args) { //comment
 3          /* Comment at head of line */ int var = 0;
 4
 5          /* Hard case //with nested comments and break
 6         mid-comment */ System.out.println("Hello!");
 7
 8          //Here the comments nest the other way /*
 9          var++;
10      }
11  }
```

In this example, lines 5 and 8 contain only comments, while lines 4 and 7 are blank. This leaves 7 lines of actual code. Note that lines of code can contain either kind of comment and that `//` can appear inside a `/*` comment and vice versa. Because of these complications, it is tricky to write a code-counting program without the idea of states. Code counting is not a highly practical problem, but it is a familiar idea for students and the program's output does provide a primitive measure of program complexity. (Despite the flaws of LOC as a measure, the author was asked to write a program to count it in a summer internship; we share this with the students to help motivate the task.)

## 3 Context-free languages

The second part of an automata course covers context-free languages, represented by context-free grammars (CFGs) and pushdown automata (PDAs). Unlike regular expressions, these are not commonly integrated into programming languages. Grammars are, however, commonly used to represent and parse programming languages. Thus, the examples for this material tend to be drawn from programming constructs.

**Printf calls.** One of the canonical examples of a language that is context-free but not regular is $0^n1^n$. The difficulty of this language comes from ensuring that the numbers of 0s and 1s are equal, something which regular expressions and finite automata are unable to do. The language $0^n1^n$ distills this key limitation into a simple example, but it is hardly motivating without any context or rationale. Instead, we use a programming-based example: invocations of `printf` where the number of conversion specifications (limited to `%d` for simplicity) in the format string must match the number of other arguments. Again, the key aspect of this language is basically to make a 1-to-1 correspondence between a feature of the first part of the word (occurrences of `%d`) and a feature of the end of the word (the other arguments). Using `printf` makes the example messier since there are other characters to consider (the word "`printf`", parentheses, commas, the variable names, etc), but also much more interesting since checking the number of arguments is a task that the compiler actually has to perform. (Of course, this is not how `printf` is parsed in practice; the parser would simply identify the function name and argument list, with the matching done afterwards.)

**Non-regular "regular expressions".** For another example, we return to regular expressions. Some implementations of these provide features that are more than syntactic sugar, actually expanding the set of languages that can be represented. For example, Python provides a matching feature; putting part of the expression in parentheses binds whatever matches that part of the expression to a numerically-named variable. These can be accessed using \1, \2, ... later in the regular expression. Thus, the expression

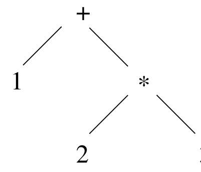
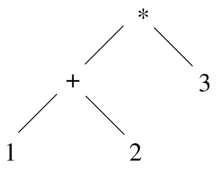
([0-9])x\1

matches words of the form $d\mathrm{x}d$, where $d$ is a decimal digit. Since the parentheses can contain an arbitrary regular expression, it is easy to show that this feature allows Python's "regular expressions" to match at least some languages that are not even context-free. (The basis for a nice homework problem...)

**Parsing ambiguity.** The next application is parsing arithmetic expressions. A naive way to generate expressions with the four basic operators is with a single rule:

$$E \longrightarrow E + E \mid E - E \mid E * E \mid E \;/\; E \mid \text{number}$$

This rule generates the desired expressions, but can generate parse trees that do not respect operator precedence and associativity. For example, the expression $1 + 2 * 3$ can yield either of the following trees:

The left tree evaluates as $1+(2*3)$ while the right evaluates as $(1+2)*3$. Obviously, the left tree is correct for the standard precedence order. Even without precedence differences, one order is preferable due to operator associativity; the expression $1-2-3$ should evaluate to $(1-2)-3$ rather than $1-(2-3)$. Standard precedence order and associativity can be enforced by changing the grammar as follows:

$$\begin{array}{lcl} E & \longrightarrow & E + A \mid E - A \mid A \\ A & \longrightarrow & A * B \mid A \;/\; B \mid B \\ B & \longrightarrow & \text{number} \end{array}$$

This is a standard example, with solutions in textbooks and on the web. To a lesser degree, the same is true of the "dangling else" problem (when `if` statements are chained and the compiler needs to select one of them to associate with a trailing `else`). Thus, these examples are shown in class, while homework features an equivalent but less-popular example such as exponentiation (which is right associative) or boolean operators (where left associativity is needed for correct short circuiting).

Some automata theory textbooks discuss this issue under the name *ambiguity*, but typically devote only a few pages to the topic. In many ways it is a programming language topic, but a very low-level one. Note that in a practical context, precedence and associativity are likely to be established by means other than the grammar; compiler generation tools such as flex and bison [3] provide directives to specify them without changing the grammar. That said, they can be established in the grammar itself and doing so illustrates the connection between interpreting computer code and grammars.

**Assignment and equality.** Considering the need to parse expressions also helps motivate the use of separate symbols for assignment and testing for equality, such as `=` and `==` in C/C++/Java or `:=` and `=` in Pascal. Both assignment and equality testing can be expressed with the English word "equals" and confusion between `=` and `==` has been found to be one of the most common errors by novice programmers learning Java [1], but looking at them from a parsing perspective quickly motivates the use of different symbols.

## 4 Turing machines

Finally, the third part of a typical automata course uses Turing machines (TMs) to cover recursive and recursively enumerable languages as well as complexity topics such as NP-completeness and the polynomial hierarchy. This is very abstract stuff, but there are still some practical connections to be drawn.

**Undecidability of program analysis.** A particularly abstract part of this material is reductions, particularly those proving undecidability. A common structure for these reductions involves describing a TM whose input is itself the representation of a TM. The TM being described manipulates this representation and then provides it as input to another TM being simulated. This is confusing on several levels. One source of confusion is just the idea of TMs taking the representations of TMs as input. We have found it helpful to point out that students use something like this all the time, namely a compiler, which is a program that takes the description of a program as input.

Building on this, we try to focus on undecidable problems of practical interest. The Halting problem, deciding whether a program will ever halt, is often presented in practical terms. We take this further with other undecidable problems that come from analyzing programs such as identifying dead code (does any input cause a TM to enter a particular state?).

**Historical background for programming terminology.** In addition, this material provides historical context for things they may have seen in other CS settings. We include a couple of days on lambda calculus as another attempt to answer to the question "what can be computed?" and its equivalence to TMs while being so different as support for the Church-Turing Thesis. On one hand, lambda calculus is clearly an automata theory topic, albeit one with a different flavor than the rest of the material. On the other hand, "lambdas" are listed as exciting new features of both C++11 and Java 8. Students with experience in functional programming have likely also seen the word there as well as function calls formatted in the same way as lambda expressions.

## 5 Our Experiences

The genesis of our work on a practically-oriented automata course was as a new assistant professor assigned to teach "Automata theory and programming languages", a course whose creation was motivated by the use of automata in parsing, but which no one had previously taught. The first offering of this course did not include many lectures explicitly combining the two topics, but gradually the perspective discussed above was developed and utilized. Later, the department decided to create separate courses in programming languages and automata theory in order to give more coverage to each area. The new automata-only course has been taught once.

Both versions of the course were taught as upper-division electives, of which 3 are required for the CS major and one for the CS minor. The courses were both taught every other year and typically taken by majors as one of their last courses, though the official prerequisites were just Discrete Mathematics and

CS 2 (Data Structures). Class sizes were 5–12, with an average just below 7.

The combined course was typically taught without a textbook because of the lack of books that covered both the automata and programming languages material. For the automata-only offering, we used a book by Webber [5] which does not directly take our approach, but which is an accessible treatment of the standard automata material. The focus on applications and practical concerns we advocate came through in lecture and assignments. The most awkward change was the use of different notation for regular expressions, but this is mitigated by the students being upperclassmen.

Because the approach described in this paper evolved gradually in a course taught only every other year to relatively few students, we do not have any kind of formal assessment comparing it to a more traditional automata course. Our sense is that explicitly connecting to practical regular expressions is helpful, particular since a couple of students in the class often speak up about how useful regular expressions are when the topic is first introduced. Similarly, the applications of regular expressions seem to be well received. Ambiguity is a somewhat challenging topic since it asks students to relate a grammar to the set of parse trees it creates. Undecidability also remains a challenge, though we believe that mentioning the compiler and focusing on languages that can be interpreted as code analysis tasks gives students a slightly more concrete frame of reference.

## 6 Discussion

Given the breadth of automata-related applications, we are not the first to notice many of them. Several books explicitly introduce some of these applications. Hopcroft et al. [2] use Unix regular expressions as an example of practical regular expressions. They also include sections on parse trees (including the use of `yacc`, a predecessor to `bison`) and ambiguity. Rich [4] includes nearly 200 pages on applications in the appendices as well as sections on parsing, parse trees, and ambiguity. Neither of these texts fully integrates applications throughout the course as we envision or make changes such as using the practical regular expression syntax, but these books would support the kind of approach we propose here if the practice-oriented material were included in the course. They are also resources that instructors could use to identify interesting applications to include in courses based on other textbooks.

Obviously, much can be done to integrate applications into textbooks and course materials as well as finding other ways to motivate the material. In addition, we are very interested in an assessment of this approach: Do students find our examples more interesting than concise language descriptions such as $0^n1^1$? Does this lead them to enjoy the course more or learn more from it?

## References

[1] N.C.C. Brown and A. Altadmri. Investigating novice programming mistakes: Educator beliefs vs student data. In *Proc. 10th Ann. Conf. Intern. Computing Education Research (ICER)*, pages 43–50, 2014.

[2] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introdution to automata theory, languages, and computation.* Addison Wesley, 2nd edition, 2001.

[3] J. Levine. *flex & bison.* O'Reilly Media, 2009.

[4] E. Rich. *Automata, computability, and complexity.* Pearson Education, 2008.

[5] A.B. Webber. *Formal language: A practical introduction.* Franklin, Beedle & Associates, Inc., 2011.

# Instructor-Formed Capstone Teams Based on Interest and Technical Experience: The Road to Success*

*Huseyin Ergin*
*Computer Science Department*
*Ball State University*
*Muncie, IN 47306*
*hergin@bsu.edu*

**Abstract**

In this paper, we adopt many good-enough capstone practices in the literature, apply it in our capstone class for one year and share our experiences. Our capstone classes take two semesters, consist of junior and senior students, and the teams have to work with a real client. We form our capstone teams based on homogeneous interests and heterogeneous technical skill level of the students. Students fill a survey at the beginning of the class to specify their technical expertise and project interest (without revealing the projects, only using the domain and technological keywords of the projects). This approach yields a more successful result for both the students, the instructors, and the real clients. We also adopt a personal contribution weight (PCW), which is a combination of various aspects we gather throughout the capstone class, to distinguish individual team member's grades.

## 1 Introduction

Project-based capstone classes are at the heart of computer science education [8]. A considerable amount of literature has been published on capstone classes. These studies usually focus on whether the class projects must be

*Copyright ©2019 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

individual or team-based, use industry or made-up projects, take one or two semesters, be managed by the students or the instructors. Regardless of these various options, capstone classes always have been an essential part of the curriculum [17]. These classes typically target undergraduate junior or senior students and are group-based to meet the requirements of accreditation and employers' demands.

In our capstone classes, we try to adopt the many good-enough practices that are extracted from the various studies published in this field [4, 5, 17]. In this paper, we mostly focus on team formation aspect because we believe this is one of the most critical aspects concerning the success of the team both for the instructor and the client partner. However, we still reveal all details of the class structure to put the team formation into the right context. We adopt an instructor-formed team formation based on technical experiences and interests of the students. One of the first systematic study of team formation is reported by Brickell et al. [1]. In their study, the authors find out that the best group settings in terms of group performance are homogeneous interests and heterogeneous GPAs. In our classes, we interpret GPA as a technical skill level and interest as the students' willing to do a project more than another in terms of the domain and technological features of the project. In addition to that, we also ensure that there is at least one high-skilled student in every team in terms of technical skill level.

In our experience, we see that this kind of team formation strategy yields better team success when combined with the good-enough practices of capstone class design.

## 2 Background

Team formation is a highly studied area both for capstone and other group-based projects in various disciplines.

In terms of forming the teams, there are two competing approaches: 1) instructor-formed, 2) student self-formed. Various studies mention the problems of self-formed teams [1, 15, 19]. Among these problems, the most crucial ones are: 1) students' not selecting diverse team members and only focusing on teammates like them in gender, ethnicity, knowledge, and ability, and 2) students' not learning how to work with unfamiliar people. The opposing argument of instructor-formed team formation is more straightforward alongside negating the problems above: instructor-formed teams are more successful in terms of class objectives [1, 7, 9, 20].

"Instructor-formed teams" approach has one big decision to make: Based on what should we arrange those teams? Studies on these selection guidelines [1, 6, 7, 9, 14] summarize what kind of factors may affect team formation strategy.

These are: 1) academic strength (i.e., GPA), 2) interest in the project, 3) work experience, 4) personality, and 5) diversity elements (i.e., gender, ethnicity).

All studies have valid points and evaluation criteria for the teams. However, we believe standardization is crucial in these kinds of studies. A study by Brickel et al. [1] provides the most standard way possible[1] to evaluate different team formation approaches. They study five different team formation approaches on 442 students enrolled in 24 sections of the same course taught by eight different instructors. The study takes place under the knowledge of instructors. However, the students are not aware of the study. They put students into teams using these approaches: 1) Heterogeneous GPA and Heterogeneous Interest, 2) Heterogeneous GPA and Homogeneous Interest, 3) Homogeneous GPA and Heterogeneous Interest, 4) Homogeneous GPA and Homogeneous Interest, and 5) Self-select. The findings are not surprising and are consistent with other studies. Teams with the same interests and different GPAs seems to be the most successful concerning team grades and attitudes about the course.

In this paper, we follow this approach to create our capstone teams: instructor-formed based on technical experience (as a replacement to GPA) and similar interest. However, we tweak the technical experience part by putting a high-skilled student into each team to guarantee the success of the projects, which we justify in the related subsection (Section 3.3).

## 3 Capstone Class Details

We have to look at the capstone class details to interpret the project-related data in the next section. Our capstone class is called "Software Engineering." It spans two semesters. The class has formal software engineering contents and the capstone project running in parallel. We are aware of the risks putting them that way (i.e., it is becoming late to adopt a technique when the teams learned it) and it is one of our plans to revise capstone class structure in the upcoming semesters. Current capstone class objectives are depicted in Table 1 (directly from the master syllabus).

The teams work with a client partner according to the master syllabus. For the sake of satisfying the client on the final product and achieving the class objectives, every aspect of the capstone project is tightly under control of the instructor while giving enough freedom for the teams to manage the project themselves and be creative.

[1]We feel comfortable saying this because the study has taken place in Air Force Academy in Colorado Springs, CO.

Table 1: Current Capstone Class Objectives

| Objective |
|---|
| -Identify, compare, and contrast the requirements capture, requirements analysis, design, implementation, and quality assurance phases of software engineering; |
| -Coordinate a requirements capture and analysis; |
| -Design a nontrivial software system; |
| -Implement a software system from a design specification; |
| -Conduct quality assurance (such as testing) on systems, including a student's own systems as well as others'; |
| -Manage a software development project; |
| -Communicate effectively within a software development team and with clients; |
| -Write effectively relevant technical documentation (use cases, UML diagrams, bug reports, user manuals, etc.) |

### 3.1 Projects

We collect all projects through the personal connections of the instructor. Eventually, we come up with a nice list of variety in terms of clients. We get in touch with clients before the semester begins and prepare initial descriptions of the projects. We also create a set of domain and technological keywords for each project. We distribute these keywords (nothing else about project) to students in the form a survey to guide assigning students to each project later on. Table 10[2] lists clients, brief project descriptions and associated keywords. We omit the names and any revealing information for the sake of anonymity.

Table 2: Projects

| ID | Client | Description | Keywords |
|---|---|---|---|
| A | Owner of two local restaurants | Inventory management software specialized in tracking the food ingredients and cooked food | Domain: Restaurant, Ingredients, Food. Technological: Client-Server, Frontend, Backend, Users, Database, Notifications, Native. |
| ... | ... | ... | ... |

Primarily, the students manage the projects themselves, and the instructor

[2]The remainder of the projects are in Appendix A.

is involved by using the grading mechanisms mentioned in Section 3.4 to make sure everything is on track.

In terms of development methodology, the class' nature suggests a waterfall approach. There are also various studies that investigate the advantages of Agile methodologies [11, 10, 16, 18, 21]. We opt in for Agile methodology given it is the current trend in the industry. However, we select a modified Agile by taking the criticisms of Meyer [13] into account. His biggest criticism is about Agile's leaving out the upfront design. We resonate with his arguments. Therefore, we decide to spend the first-semester doing requirement analysis, design, and prototyping. The main output of these steps are not comprehensive documentation like waterfall's software requirement specification but a correct amount of user stories and just enough documentation to understand the domain, functionality, and technical aspects of the project. In the second semester, we focus on producing the project in three monthly releases in an Agile way followed by another final release to transfer the product to the client.

The primary source of collaboration takes place over one central GitHub repository for each team. They store everything except code-base of the projects in these central repositories (i.e. user stories, documentation, weekly meeting notes). The teams also should meet with their clients monthly and, then, present their meeting details and project progress in the class. This helps teams learn from other teams [4]. They store client meeting notes and these monthly presentations in their central repositories as well.

## 3.2 Students

There are 21 students enrolled in our capstone class. Eighteen of them enrolled in the Fall semester, and three more transferred from another section of the class in the Spring semester because of a conflict in their schedule. However, any data we reveal is about the initial 18 students.

At the beginning of the semester, a survey is distributed to know students better and to assign each one to the projects. This is in parallel to the studies by Emanuel et al. [6] and Oakley et al. [14]. Students are not informed with what projects are available, and they are only exposed to the keywords of projects in a random manner in the survey. In this survey, we ask questions about their keyword preferences both domain and technological, programming experience, top programming languages, top IDEs, internship and work experiences. We use some of the questions as a distraction not to reveal the main purpose of the survey such as the programming language and IDE choices.

As a result of the survey, we record each student's keyword choices and classified their technical abilities with respect to the scale depicted in Table 3.

Table 3: Scales and Explanations

| Scale | Explanation |
|---|---|
| High | Has lots of experience and skill and can drive the team to success |
| Medium | Enough technical skill to support a good technical leader |
| Low | Not enough technical skill and have to follow the rest of the team to contribute |

According to the survey: 1) 39% of the students are high-skilled, 2) 28% of the students are medium-skilled, and 3) 33% of the students are low-skilled.

### 3.3 Teams

Our capstone class enforces to work within team settings in the master syllabus. Regardless of that, most of the students who plan to work in the industry will work in a team setting after graduation. We decide to keep the size of the teams at three mainly because we do not want the number of communication lines hinder the success of the projects [2]. Richards [17] observes that teams of three often delivers better solutions, probably because of suffering less from the communication-related problems. Although we do not have hard evidence on this, we believe teams of size more than three have a risk of dividing themselves into subteams which makes the communication harder among them. However, in teams of size three, if two team members are working, the third member has no chance but to work together with the rest of the team.

#### 3.3.1 Team Building

According to McConnell [12], teams require around four to seven weeks to be effective to start producing, which depends on the frequency of group interactions. Fiechtner and Davis [7] also suggest that the teams must be cohesive to be productive. To improve the cohesiveness and reduce the time to do it, we adopt team building exercises on the first month of the capstone class before revealing the capstone project details. Some of these team building exercises are not even coding related, but, aim to make students get used to working with non-familiar students. The exercises we do are listed in Table 4.

#### 3.3.2 Capstone Teams

We use the results of the survey we have distributed at the beginning of the class (see Section 3.2) to form the capstone teams. We decide to put at least one

Table 4: Team Building Exercises

| Exercise |
|---|
| -Drawing parts of a frog on a piece of post-it note and trying to complete the frog picture. |
| -Dividing a swirl image for each team member, drawing five times bigger version of it and combining them. |
| -Creating a repository for and testing a roman number conversion system. |
| -Creating use cases for a system that is described with only two sentences. |
| -Using GitHub to put their profile information and doing it using pull requests. |

high-skilled student to each team according to the study by Webb et al. [22], which basically can be summarized as "the team performance is correlated to the team member with the highest ability level." In the same study, a disadvantage of that is listed as follows: "Heterogeneous groups provide a greater benefit for below-average students than they impose a detriment on high-ability students." This detriment is a disadvantage we chose to dismiss for the greater good of the team and the projects. From what we find, our final decision is to create heterogeneous teams based on their technical skills but with common interests. By using the project (domain) and technical interests at the same survey (see Section 3.2), we ensure the teams are composed of students with homogeneous (domain) interests in terms of what the project is about and heterogeneous in terms of skill level.

Table 5 shows the distribution of students in terms of their skill level in each project. We foresee that the high-skilled student would drive the team

Table 5: Students' Technical Skills in Each Project

| Project ID | Student1 | Student2 | Student3 |
|---|---|---|---|
| Project A | High | Medium | Low |
| Project B | High | Medium | Low |
| Project C | High | High | Low |
| Project D | High | Medium | Low |
| Project E | High | Medium | Low |
| Project F | High | Medium | Low |

in terms of succeeding in the project and other will learn a lot from him/her. However, this information is not shared with the teams or any team members.

## 3.4 Grading

According to Richards [17], regardless of what is the major focus of the grading (process or product), the project itself is assessed. We take every measure to keep the teams on track about the product and the process following the multi-source assessment of Dominick et al. [3]. As we explain in Section 3.1, the teams should meet with their clients monthly in both semesters. These meetings both help us track the team's progress more frequently and grade them fairly. Table 6 summarizes the grading scheme of the overall capstone class.

Table 6: Grading Scheme

| Item | Ratio |
|---|---|
| Project | 60% |
| Midterm exam | 10% |
| Final exam | 10% |
| In-class exercises | 15% |
| Class participation | 5% |

Table 7 depicts the pieces to grade in each project.

Table 7: Project Grading Details

| Item |
|---|
| -At least three monthly client meetings every semester |
| -Progress presentation after every client meeting |
| -Peer evaluations after every iteration |
| -Client evaluations at the end of semesters |
| -Weekly team meeting notes |
| -Quality of the produced materials such as documentation, sketches, prototypes |
| -Structure of the main repository |
| -Structure of the code repositories |
| -Quality of the produced code |
| -Quality of the end product |

### 3.4.1 Personal Contribution Weight

Fiechtner and Davis [7] and some other researchers [10, 20] find out that peer evaluations are an important part of a grade that leads to a fair grading in the class. Peer evaluation-based individual grades for projects eliminate the "free riders." However, peer evaluations can also lead to some problems if team member's peer scores have too much impact on one's grades. By taking these threats into account, we decide not to directly take peer evaluations' results but use them to calculate a better number that represents the contribution of each team member in a team.

We call it personal contribution weight (PCW). PCW is a number from 0.0 to 1.0, and it is a combination of various factors of the tasks an individual team member has done in the team. The factors that affect a team member's PCW are depicted in Table 8.

Table 8: Factors Affecting PCWs

| Factor |
|---|
| -The number of commits of the team member in the code repositories |
| -The frequency of commits of the team member in the code repositories |
| -The contents of commits of the team member in the code repositories |
| -Peer evaluation scores of the team member by all team members (including themselves) |
| -The role of the team member logged in the weekly team meeting notes |
| -The task distribution of the team member that is presented by the team at every iteration |
| -The instructor's perception of the team member |
| -The client's perception of the team member |

We experimentally adopt PCW to differentiate each team member's project grade. PCW is not created to punish team members but to encourage them contribute more and for us to be fair in grading their work. Therefore, we use the first PCW as a warning to the team member and, at every iteration, we recalculate PCWs. Then, PCWs are used as a multiplier to the team iteration grades to find each team member's project grade (i.e. Project Grade = Sum of PCWs * Iteration Grades).

## 4 Current Status & Discussion

In industry-sponsored projects, one of the main objectives is successfully delivering the right software [8]. Therefore, we analyze the current status of the project concerning their completion level, and functionality promised to the client at the beginning of the capstone class. We also analyze the in-team progress and the roles of the students with different skills levels. Are the students with high technical skills driving the teams to success?

According to the interviews with our clients and progress presentations after client meetings, all projects look like they are on the right track to deliver the right software. Table 9 displays the PCWs of each team members along with their skill levels.

Table 9: Personal Contribution Weights at the End of the Class

| Project ID | Student1 | PCW | Student2 | PCW | Student3 | PCW |
|---|---|---|---|---|---|---|
| Project A | High | 1.0 | Medium | 0.9 | Low | 0.8 |
| Project B | High | 1.0 | Medium | 1.0 | Low | 0.9 |
| Project C | High | 1.0 | High | 0.9 | Low | 0.6 |
| Project D | High | 1.0 | Medium | 1.0 | Low | 0.4 |
| Project E | High | 1.0 | Medium | 0.6 | Low | 0.9 |
| Project F | High | 1.0 | Medium | 0.3 | Low | 0.5 |

A critical interpretation of Table 9 is that high-skilled students whom we foresee to drive the teams to success reflected these efforts in their PCWs. We already take the peer evaluation scores into account to calculate PCWs. However, we recheck the peer evaluations to see what other team members are thinking about the high-skilled students in the team. The high-skilled students receive higher grades from their peers in terms of commitment, participation, communication, and technical contribution. According to the free form comments that students only share with the instructor, high-skilled students are “doing good job to keep the team on track”, “enjoyable to work with positive attitudes”, and “considered very valuable to the team.”

## 5 Lessons Learned

Teamwork is not without problems even though assigning a high-skilled student to each team guarantees success in the project. Here are some lessons learned on the way:

- Some high-skilled students are too high-skilled. They can not easily stop themselves from making project contributions. High-skilled students' work ambition sometimes lead other team members to feel their work is worthless because they can not keep up with the high-skilled students. In this case, we have to balance the team during the semester by convincing the high-skilled student to triage more tasks to other team members and be patient while the rest are working.

- Some low-skilled students are not affected by their high-skilled peers. Therefore, they have a lower PCW as a result.

- Some high-skilled students complain about their low-skilled peers. We advise the high-skilled student to convince peers to work to reduce their workload instead of losing hope and giving up on the low-skilled students.

Some high-skilled students mention the following in the free form comments section of peer evaluation forms that is only shared by the instructor[3]:

- “I feel as though a lot of the pressure for our project has fell on my shoulders.”
- “I sometimes have doubts if the team would do anything unless I asked first.”
- “I do feel like I carry this team a little too hard at times and organize everything.”

These thoughts by the high-skilled students confirm that they feel they are the driving force behind the teams and they feel the pressure. We already know that high-skilled students are affected negatively for the sake of the greater good of the team [22]. How powerful the effect of this on the high-skilled students’ performance is probably a topic of another study.

## 6 Conclusion & Future Work

In this paper, we have shared our experiences with a team formation based on homogeneous interests and heterogeneous technical skill level. We have also modified this approach a little bit and ensured that there is at least one high-skilled student in each team. We have six teams that we have formed using this approach. In all of the teams, we have found out that each team to become successful concerning the promises made to the client at the beginning of the capstone class.

In addition to that, we have adopted a personal contribution weight (PCW) that is a combination of a couple of aspects to assign each student individual team grades for their contributions. As a future work, we plan to analyze the advantages and disadvantages of PCWs.

As a broader goal of our department, we have some other plans as well. We are working on redesigning our capstone class experience by taking the many studies into account [4, 5, 17]. We also want to provide a more sustainable way of acquiring real clients for capstone projects and work with a local incubation center for start-up companies to form a new kind of capstone project center.

[3]We have not fixed the grammatical errors of students.

## References

[1] Lt Col James L. Brickell, Lt Col David B. Porter, Lt Col Michael F. Reynolds, and Capt Richard D. Cosgrove. Assigning Students to Groups for Engineering Design Projects: A Comparison of Five Methods. *Journal of Engineering Education*, 83(3):259–262, jul 1994.

[2] Frederick P. Brooks Jr. *The Mythical Man Month and Other Essays on Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1975.

[3] P. Dominick, M. Besterfield-Sacre, J. McGourty, L. Shuman, and H. Wolfe. Improving student learning through the use of multisource assessment and feedback. In *30th Annual Frontiers in Education Conference. Building on A Century of Progress in Engineering Education. Conference Proceedings (IEEE Cat. No.00CH37135)*, volume 1, pages F1A/7–F1A11. Stripes Publishing, 2002.

[4] Robert F. Dugan. A survey of computer science capstone course literature. *Computer Science Education*, 21(3):201–267, sep 2011.

[5] Alan J. Dutson, Robert H. Todd, Spencer P. Magleby, and Carl D. Sorensen. A Review of Literature on Teaching Engineering Design Through Project-Oriented Capstone Courses. *Journal of Engineering Education*, 86(1):17–28, jan 2013.

[6] Emanuel and Worthington. Team oriented capstone design course management: a new approach to team formulation and evaluation. In *Proceedings Frontiers in Education Conference*, pages 229–234. IEEE, 2003.

[7] Susan Brown Fiechtner and Elaine Actis Davis. Republication of "Why some groups fail: A survey of students' experiences with learning groups" (original publication 1984). *Journal of Management Education*, 40(1):12–29, 2016.

[8] Sally Fincher, Marian Petre, and Martyn Clark. *Computer science project work : principles and pragmatics*. Springer, 2001.

[9] David Hunkeler and Julie E. Sharp. Assigning Functional Groups: The Influence of Group Size, Academic Record, Practical Experience, and Learning Style. *Journal of Engineering Education*, 86(4):321–332, oct 2013.

[10] Jon G Kuhl. Incorporation of Agile Development Methodology into a Capstone Software Engineering Project. In *Proceedings*, 2018.

[11] Phillip A. Laplante. An agile, graduate, software studio course. *IEEE Transactions on Education*, 49(4):417–419, nov 2006.

[12] JJ McConnell. Active and Cooperative Learning: Further Tips and Tricks (Part 3). *ACM SIGCSE Bulletin*, 38(2):24, 2005.

[13] Bertrand Meyer. The Ugly, the Hype and the Good: an assessment of the agile approach. In *Agile!*, pages 149–154. Springer International Publishing, Cham, 2014.

[14] Barbara Oakley, Richard M Felder, R Brennt, and Imad Elhajj. Turning Student Groups into Effective Teams. *Journal of Student Centered Learning*, 2(1):9–34, 2004.

[15] Michael A. Redmond. A computer program to aid assignment of student project groups. *ACM SIGCSE Bulletin*, 33(1):134–138, 2004.

[16] Thomas Reichlmayr. The agile approach in an undergraduate software engineering course project. In *Proceedings - Frontiers in Education Conference, FIE*, volume 3, pages S2C13–S2C18. IEEE, 2003.

[17] Debbie Richards. Designing Project-Based Courses with a Focus on Group Formation and Assessment. *ACM Transactions on Computing Education*, 9(1):1–40, mar 2009.

[18] Diane Rover, Curtis Ullerich, Ryan Scheel, Julie Wegter, and Cameron Whipple. Advantages of agile methodologies for software and product development in a capstone design project. In *Proceedings - Frontiers in Education Conference, FIE*, volume 2015-Febru, pages 1–9. IEEE, oct 2015.

[19] Rebecca H. Rutherfoord. Using personality inventories to help form teams for software engineering class projects. *ACM SIGCSE Bulletin*, 33(3):73–76, 2004.

[20] Jim Sibley and Dean X. Parmalee. Knowledge is no longer enough: Enhancing professional education with team-based learning. *New Directions for Teaching and Learning*, 2008(116):41–53, sep 2008.

[21] Tucker Smith, Kendra M.L. Cooper, and C. Shaun Longstreet. Software engineering senior design course. In *Proceeding of the 1st international workshop on Games and software engineering - GAS '11*, page 9, New York, New York, USA, 2011. ACM Press.

[22] Noreen M. Webb, Kariane M. Nemer, Alexander W. Chizhik, and Brenda Sugrue. Equity Issues in Collaborative Group Assessment: Group Composition and Performance. *American Educational Research Journal*, 35(4):607–651, jan 2008.

## A Projects

Table 10: Projects

| ID | Client | Description | Keywords |
|---|---|---|---|
| A | Owner of two local restaurants | Inventory management software specialized in tracking the food ingredients and cooked food | Domain: Restaurant, Ingredients, Food. Technological: Client-Server, Frontend, Backend, Users, Database, Notifications, Native. |
| B | Director of Interactive Learning Spaces | Analyzing students' movements in a classroom by using an app and Bluetooth communication | Domain: Students, Seating, Relocation, Indoor. Technological: Surveys, Location, Data collection, Mobile, Client-Server, Notifications, Reporting, Backend, Frontend. |
| C | Same client as above | Analyzing students' movements in a classroom by using an admin only app that lets drawing the layout for classroom | Domain: Students, Classroom, Modeling. Technological: User interface, Native, Reporting, Database. |
| D | Supervisor of shuttle bus operations in our university | Collecting and analyzing shuttle bus statistics about how many students are getting on the bus at each stop and how many are left | Domain: Transportation, Buses, Drivers. Technological: Offline, Client-Server, Frontend, Backend, Database, Location, Mobile, Users, Caching, Reporting, Charts, Data collection. |
| E | CEO of a local software development company | A web-based travel content management solution to provide an efficient platform to edit, showcase, and organize B2B client contents | Domain: Travel, Routes, Places, Tours, Tourism, Map. Technological: Azure, Responsive, Frontend, Backend, Location, Database. |
| F | Same client as above | An app to record tour paths and let create shared tours among peers in real time | Domain: Travel, Map, Path. Technological: Real-time, Azure, Client-Server, Mobile, Frontend, Synchronization, Backend, Location. |

# CS+ Creating a Community Outreach Group in Computing from the Ground Up[*]

*Brian Krupp, Sydney Leither, Zachary Egler,*
*Tyler Hardy, Paul Peters*
*Computer Science Department*
*Baldwin Wallace University*
*Berea, OH 44017*
{bkrupp,sleither17,zegler16,thardy15,ppeters13}@bw.edu

**Abstract**

Large scale efforts that exist to attract more students to computer science have made significant progress. However, while these efforts have been successful in varying degrees, access to these opportunities are fragmented and inequitable. While contributing to these mainstream programs such as CS4All and Hour of Code should continue, grassroots initiative within local colleges and organizations can also help provide opportunities that national efforts do not or supplement their efforts. In this paper, we share our experiences of building such a program at a primarily undergraduate institution. We share how students within our computer science program created and maintained a community outreach program that delivers opportunities to the community for students to learn computer science in various formats. We provide detailed information into the structure of the organization, recruitment of students, programs that we offer, and lessons learned. We also provide best practices so that similar institutions can offer such programs and utilize the resources we have made available to start a similar program. While our efforts alone may have a small impact nationally, if other institutions start similar efforts, collectively the impact we have can be far greater and personalized for each community we serve. We hope these efforts

[*]Copyright ©2019 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

will work towards increasing the diversity in our field and continue to build confidence in students that want to pursue a career in computer science.

## 1 Introduction

Throughout the computing community, large scale initiatives have been created to help promote computer science as a standard course offering. While these efforts have had a significant impact, there are still schools that lack access to computing [1]. Currently, only 35% of schools teach computer science [1] while 90% of parents want their child to study computer science [4]. This access is puzzling given that in 2019, there are approximately 500,000 open computing jobs nationwide [1] and computer science graduates can expect to earn 40% more than other college graduates [7]. While national programs have been effective in increasing overall participation [9], [10], [6] given the computing jobs available and lack of access to computing [5], we believe there is a strong demand for local programming as well. Inequitable access to computing continues to be a challenge where students from lower income backgrounds [2] and in rural areas [8] have less access to learn computer science. In this paper, we describe how we created a community outreach group that promotes computer science within the Northeast Ohio area. We describe how the organization was formed in collaboration with students and faculty at Baldwin Wallace University. We share how the organization recruits to promote scalability and ensure continuity, and also describe the interview process for students that want to join the organization. We then discuss various programs that are offered within the organization including School Visits, Tech Camps, and partnership programs including a Tech Club hosted within an elementary school and a Programming Camp at a juvenile correction center. We include best practices and lessons learned from our experiences so that students and faculty at other institutions can start similar efforts.

## 2 Organization

### 2.1 Founding

CS+ was founded in partnership between computer science students and faculty of Baldwin University out of a desire to expand and enhance interest in computing for future generations. Founded in 2015, members of CS+ worked to build an identity for the organization and plan for future development. This included identifying engaging activities for young adults in the Northeast Ohio Area that can be easily replicated by other institutions across the nation.

Members of the organization are not paid for participating nor do they receive any financial benefit. With many ideas discussed amongst students and faculty, CS+ members knew not all ideas would be easily incorporated. Founding members documented long-term goals while working to implement first-priority engaging activities for the community. The first programs implemented include presentations at local schools (to build lasting partnerships with local educators) and monthly tech camps to provide opportunities for students to learn computing and build their confidence in the discipline. To support these efforts, the initial group consisted of four students and one faculty member. The organization found it important to create a vision and mission for the group as a compass in the creation of future programs and for recruiting new members. Creating a vision and mission statement was proposed by students within the organization and included themes such as equitable access, engagement, and broadening participation.

## 2.2 Recruitment and Roles

To ensure the sustainability of the program and scale in future efforts, student members of the organization recruited additional students by visiting computer science courses at our university and discussing the organization, what students can expect in joining the organization, and how to apply. Applicants submit a Google Form that includes basic contact information and two questions that are used to determine if the applicant is a good fit for the organization: 1. Why are you interested in joining? and 2. What skills do you possess that would make you a valuable member to the team? If the group determined that an application should move forward, a brief ten-minute interview is conducted with the candidate. Primarily, the team is attempting to determine if the candidate would be a good fit within the current team and if they would be suitable for working with students that attend our programs. Two interview questions were used to help determine this: 1. Can you provide an example of how you helped someone who was struggling with something? This question helps reveal their problem-solving skills and ability to handle stress and questions from attendees. 2. How do you seek to develop yourself through CS+? This question shows that they are focused on the mission of the organization and not just as a resume builder.

Long-term success for the organization played a large role in discussions on recruitment and sustainability. Roles were developed by founding members with the assumption that roles would be flexible for students and the changing demands of their schedules, and also to include the necessary work for later programs that might not yet exist. Below are the roles that were created:

- Faculty Advisor - Maintain relationships with schools, members, and participants. Mentor current facilitators and serve as a resource for students.

- Senior Development Facilitator - Accomplished all goals as development facilitator and in addition create and update tech camp material, create school presentation and workshops, and speak to students on joining CS+. They are also responsible for reviewing applications.
- Development Facilitator - First year within organization, assists with five events per year and regularly attends staff meetings.
- Alumni Advisor - Graduate that previously held position as Senior Development Facilitator. Shows continued interest in program and provides advice.

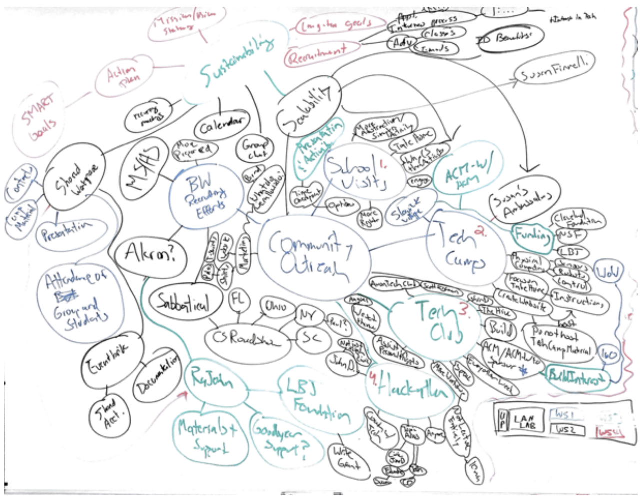


Figure 1: Mind Map Session Creating Goals for Group

## 2.3 Goals

Founding members documented long-term goals for CS+ and generated ideas through various brainstorming sessions. One of these included the creation of a mind map that visually details future opportunities for expansion and ensuring continuity and scalability of the organization (Figure 1). Through this session, student organizers and faculty mentors branched from a central theme and generated any possible idea for the group without questioning the feasibility. Using this idea generation technique, members were able to identify goals for the following year and create a plan for achieving these goals as well as identify future programs and opportunities.

## 3 Programs

### 3.1 School Visits

A vital component to starting CS+ was to visit schools to promote Computer Science and allow students to participate in a "mini" Tech Camp. In visiting the schools, partnerships are created with local middle school and high school teachers. Through these partnerships, we learn about opportunities provided at each location and can help the teacher navigate different resources and programs that are available locally. Additionally, by visiting the schools this allows us an opportunity to promote our Tech Camp programs (Figure 2).

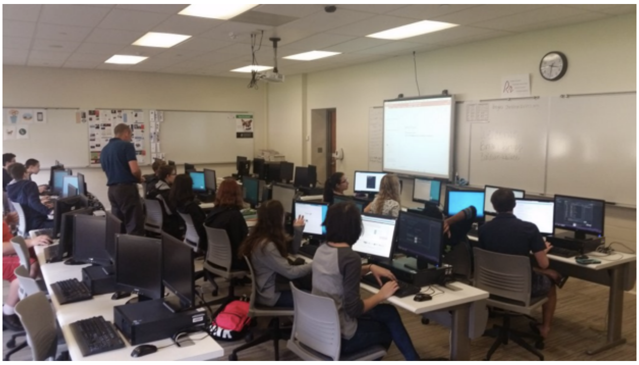
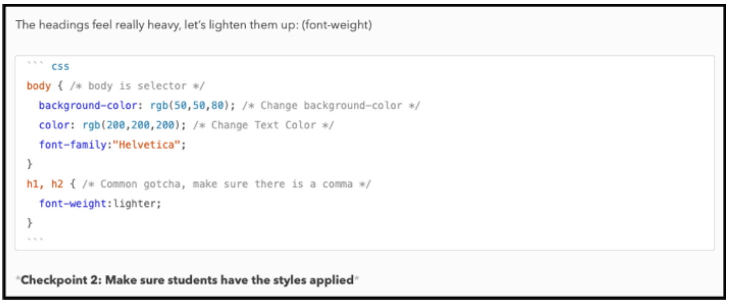


Figure 2: Running a Mini Tech Camp During a School Visit

### 3.2 Tech Camps

Tech Camps are designed to introduce students to a specific area in computing by allowing them to produce different take-home projects, such as a program, game, web page, or other artifact, within a two-hour session. Typically camps run weekdays from 5:30pm to 7:30pm. Most camps are "sold out", meaning that all seats (24) are reserved well in advance of the day of the camp. On average, we experience that approximately two thirds of those who register attend the camp as there is no risk with a free ticket. However, on one occasion we overbooked a camp providing more seats that were available and had to bring additional chairs and share computers. Since that event, we have not overbooked a camp. Attendees to Tech Camps may drive over an hour to attend, which provides evidence for the need of similar programs in our region. While we do not maintain a list of locations attendees are from, during basic introductions with attendees we ask to include their city. From this introduction, we estimate that a third of our students travel between 20-25 miles to attend the camp. Considering that parents transport them a considerable distance and wait for two hours demonstrates the desire of these programs.



The headings feel really heavy, let's lighten them up: (font-weight)

~~~
``` css
body { /* body is selector */
  background-color: rgb(50,50,80); /* Change background-color */
  color: rgb(200,200,200); /* Change Text Color */
  font-family:"Helvetica";
}
h1, h2 { /* Common gotcha, make sure there is a comma */
  font-weight:lighter;
}
```
~~~

*Checkpoint 2: Make sure students have the styles applied*

Figure 3: Snapshot of Tech Camp Material for Instructors with Checkpoint Included

Each Tech Camp starts with students introducing themselves to one another where they provide their name, where they are from, and why they attended the camp. Instructors then do the same once all students have introduced themselves. Following introductions, basic rules are introduced to ensure that students get the most out of the camp and can receive uninterrupted help from the instructors. Camps are typically led by a lead instructor or pair of instructors where additional members of CS+ will float around the classroom to assist attendees. Each instructor or member assisting is given a handout that includes instructions the lead instructor is following. Each set of instructions contains checkpoints that all attendees reach before the camp moves forward (Figure 3). It is important to note volunteers do not need to be Computer Science majors nor is a background in the material required. Since each instructor is given a handout with step by step instructions, almost anyone is able to follow the steps as an attendee would. Using this approach, it allows anyone in the group to participate and also learn while the camp is being run. While most of the members of the group are Computer Science majors, we also have several Education majors that assist with camps where their classroom management skills have been beneficial on several occasions.

### 3.3 Available Camps

The primary goal of a Tech Camp is to provide exposure to a broad range of topics in computing that are engaging for attendees. They also aim to build confidence where attendees can create something meaningful within a two-hour timeframe and continue to build upon it outside of the camp. Most camps "Sell

Out", meaning all tickets are reserved where a majority of camps have 24 seats available. Below is a sample list of Tech Camps that have been offered:

- Building Web Pages with HTML and CSS (Parent + Child Team Camp)
- Programming Self Driving Robots
- Creating a Halloween Game in Scratch
- Programming Electronics with Raspberry Pis
- Interacting with Embedded Systems with Web APIs
- Intro to Python and Data Processing

Unsolicited feedback from attendees are generally positive and indicate the programs are providing value to the community. One attendee's parent provided the following supporting the structure of our camps: "Thank you!!! Great class & thank you to your volunteers that took their time to help out, very much needed." Even with students that have access to computing, the following feedback indicates that the camps provide an opportunity to apply what they learn: "This is a really great thing that you are doing. Currently, other than for his AP comp sci class at school, my son spends his time coding cheats for his video games, so I'm happy for him to have an opportunity to learn another productive use of coding". Other feedback received indicates a positive impact in considering careers in computing: "Thank you for hosting this event! My son had a really nice time. He just did a report last week on choosing a STEM job and what it would entail. He chose Computer Coding of all things!".

## 3.4 Additional Camp Information

To advertise camps and manage reservations for seats, we use Eventbrite. Eventbrite allows us to not only manage reservations for seats, but we also communicate future events to past attendees. We also communicate upcoming events on Twitter, LinkedIn, and through our university's internal homepage. With the camps we offer, we intentionally include tools that are free for students to use. For creating web pages and learning JavaScript, students will use Notepad++ and a web browser. We found from these two simple tools there are many different camps we can create. For developing games in Scratch, we have students create the games right in the web browser (https://scratch.mit.edu). To program robots, we use the Arduino IDE and to document instructions for the circuit camps we used Fritzing (http://www.fritzing.org) to create the diagrams. We are also intentional in using low cost computing platforms such as the Raspberry Pi which costs approximately $35 USD per board [3].

A variation of the camps that we offered included a combined "Parent + Child" camp. The goal of this is to allow the parent to be involved in the process and have an opportunity for them to spend time with their children on something they are interested in. We hoped this opportunity would lead to future support from the parent.

## 4 Partnership Programs

### 4.1 The Hive Tech Club

Aligning with our vision, we partnered with community organizations including the Boys & Girls Club, Open Doors Academy, P16, and a Cleveland Metropolitan School: Mound STEM, within the Slavic Neighborhood Village in Cleveland to create "The Hive" Tech Club. The club is an after-school program where students meet every other week for an hour and a half to learn a topic in computing. At the end of the academic year, through grant funding from "Friends of Slavic Village", we were able to run a week long robotics camp. Students learned how to program a "self driving" robot using various sensors and motors to control the robot. At the end of the camp, they were able to take the robot home and received additional instructions on programming the robot. Participants in the club include students from age 8 - 12.

### 4.2 Cuyahoga Hills Programming Camp

An additional effort working towards our vision includes partnering with Cuyahoga Hills Juvenile Correctional Facility to offer a week long programming camp in the Summer of 2019. This camp will allow students to learn basic programming skills using an object-oriented programming language. Towards the end of the camp, students will work on their own programming project. This partnership requires additional planning as there are limitations on devices that can be brought into the facility as well as access to the Internet. However, providing this opportunity not only allows students access to learning computing but also provides a potential pathway for students once they are released from prison to pursue a career in computing.

## 5 Discussion

To encourage other institutions to start similar organizations, we describe some best practices from our experiences here. An initial step to creating a local organization is to identify several students that demonstrate commitment and tenacity. Once a founding group is formed, visit local schools within to discuss computing as a career and run a mini tech camp. This tech camp can be as simple as making a web page in HTML and CSS as all that is needed is a simple text editor and a browser. If your institution has an education department, ask your colleagues to provide contact information for schools in the area.

Once several schools have been visited, you can host a Tech Camp locally at your institution by reaching out to several of the schools you visited and have students sign up for the camp. While our group offers one camp a month, we plan to host them every three weeks beginning the 2019-2020 academic year as camps quickly "sell out" and more students are becoming aware of the offering. To host Tech Camps, we find that Eventbrite allows us to easily notify past attendees and manage reservations.

Once several camps are hosted, recruitment within the organization is a key factor in ensuring sustainability and scalability. Students within the organization can visit classrooms and provide an overview of the group and provide access to the application. Using Google Forms, you can set up notifications when a new application is submitted. Once applications are reviewed and applicants are interviewed, you want to provide roles to the new members so they can contribute immediately. When interviewing candidates, you want to ensure that they will be a good fit not only working with the group but also that they will be patient and encouraging for attendees as a student volunteer can do just as much damage to a learner's confidence as they can do good.

As the organization is formed, it is important to ensure constant student involvement and ownership. While our organization meets bi-weekly, meetings do not need to be as frequent or formal in the beginning. However, student members should be involved in creating the tech camps or leading them, planning future goals for the group, and reviewing applications for incoming members. Finally, when hosting Tech Camps, we encourage to use free programs whenever possible so that students can work on material they learn in the camp at home. Additionally, if you have a website available you may want to post materials that helps identify resources for students to utilize between camps.

## 6 Conclusion

Large scale efforts have had an incredible impact in getting more students interested and engaged in computer science. However, there still exists a substantial need for these programs and not every learner has access to them. In this paper, we provided details for a group that was started at Baldwin Wallace University that has experienced an incredible amount of success in the first three years. By visiting schools and offering various tech programs, we have been able to inspire future computer scientists and build confidence in their abilities. By sharing our experiences and organizational structure, we hope that other institutions will be inspired to start similar efforts. While one group cannot solve this problem alone, collectively, we can make an impact as every effort contributes towards the goals of national movements. This is especially important to ensuring that talent is not centralized to well-known tech areas and that companies with a focus in technology have the talent pool available locally. Every student should have an opportunity to learn computing. While our community outreach group provides opportunities, there are many students even within our own region that do not have access to attend these opportunities. Therefore, while we have made significant progress in our efforts, there is plenty of work to be done.

## References

[1] Code.org. Computer science education stats, March 2019. `https://code.org/promote`.

[2] K-12 Computer Science Framework Steering Committee. K-12 computer science framework. Technical report, New York, NY, USA, 2016.

[3] Raspberry Pi Foundation. Raspberry Pi 3 Model B+, March 2019. `https://www.raspberrypi.org/products/`.

[4] Gallup. Pioneering results in the blueprint of U.S. K-12 computer science education. `https://csedu.gallup.com/home.aspx`.

[5] Google Inc and Gallup. Searching for computer science: Access and barriers in U.S. K-12 education report, 2015. `https://services.google.com/fh/files/misc/searching-for-computer-science_report.pdf`.

[6] Hadi Partovi. A comprehensive effort to expand access and diversity in computer science. *ACM Inroads*, 6(3):67–72, August 2015.

[7] The Hamilton Project. Career earnings by college major, March 2019. `http://www.hamiltonproject.org/charts/career_earnings_by_college_major/`.

[8] Jayce R. Warner, Carol L. Fletcher, Ryan Torbey, and Lisa S. Garbrecht. Increasing capacity for computer science education in rural areas through a large-scale collective impact model. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE ’19, pages 1157–1163, New York, NY, USA, 2019. ACM.

[9] Cameron Wilson. Hour of Code—a record year for computer science. *ACM Inroads*, 6(1):22–22, February 2015.

[10] Cameron Wilson. Hour of Code: Maryland, washington and san francisco move to support computer science. *ACM Inroads*, 6(3):14–14, August 2015.

# Revitalizing the Linux Programming Course with Go[*]

*Saverio Perugini*
*Department of Computer Science*
*University of Dayton*
*Dayton, Ohio 45469*
*saverio@udayton.edu*

**Abstract**

We present the design of a contemporary *Linux Programming* course, which includes the use of the Go programming language. To foster course adoption and adaptation, we discuss the design of the course, which includes progressive and thematic content modules, a series of supportive programming assignments, and a culminating, final project experience. The goal of this article is to revitalize the *Linux Programming* course with the use of Go, generate discussion in the community around it, and inspire and facilitate a similar use of Go.

## 1 Introduction

Nearly thirty years have passed since J. Wolfe published on the topic of reviving systems programming [10]. Given recent trends in big data and cloud computing, web frameworks, and concurrent programming models, we believe the time is ripe to revisit and revitalize the *Linux Programming* course—an enabling and support course for those and similar areas of computing in science and engineering. While many of the concepts and topics in this course have remained stable over the past thirty years, we have attempted to revitalize the course through a focus on concept and tool integration, coverage of Git as version control software, and, particularly, the use of the *Go programming language*[1] [9] as an improved C, especially to reinforce that Linux system calls

[*]Copyright ©2019 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

[1]Developed by Robert Griesemer, Rob Pike, and Ken Thompson, the latter two of which were involved in the original work on UNIX and C; see `http://golang.org/`.

can be called from any programming language.

*Linux Programming* is a course that provides an accessible introduction to programming in the Linux environment, especially in C and Go, and prepares students for developing software in that environment using those languages. Topics include libraries and system calls, shells, operating system structures and internals, concurrency, interprocess communication (pipes and signals), the client-server model, configuration and compilation management, regular expressions, pattern matching and filters, shell programming, and automatic program generation. The course distills these topics and concepts through a survey of various software tools supporting Linux programming, including `gcc`, `gdb`, `make`, `git`, `sed` and `awk`, and `lex` and `yacc`, with a thematic focus on the programming environment that these tools collectively foster which is calibrated toward productivity. The course does not aim to be comprehensive and focuses more on breadth than depth. Assignments are designed to provide students with a pragmatic exposure to these tools as well as issues faced by modern practitioners.

*Linux Programming* is a programming-intensive course. The prerequisite for the course is an operating systems course. The course assumes no prior experience with Linux, C, Go, or any other language used in the course, but expects that students are familiar with programming in some block-structured language.

## The Linux Philosophy

Among the multiple themes constituting the Linux philosophy we simply mention the following two; it is our hope that students acquire an appreciation for Linux through observation of these recurring themes, and others, of Linux in this course.

**Concurrency and Communication:** Often in Linux programming we compose a solution to a problem by combining several small, atomic programs in creative ways through interprocess communication mechanisms such as pipes. Atomic programs are the building blocks; communication mechanisms are the glue. Such programs are easier to develop, debug, and maintain than large, all-encompassing, monolithic systems.

> If you give me the right kind of Tinker Toys, I can imagine the building. I can sit there and see primitives and recognize their power to build structures a half mile high, if only I had just one more to make it functionally complete. — Ken Thompson, creator of UNIX and the 1983 ACM A.M. Turing Award Recipient, in *IEEE Computer*, 32(5), 1999.

**Uniform style of I/O:** For instance, the function `fprintf` and the system call `write` can be used to write to standard output, a file, or a pipe.

**The Student Learning Outcomes are:**

- Establish a comfort with and proficiency in Linux and C/Go as a programming language/environment.
- Survey various important system-oriented software tools (`gcc`), including debuggers (`gdb`), and compilation (`make`) and configuration (`git`) managers.
- Establish an understanding of the power of a programmable shell.
- Establish an understanding of the power of the Linux filter style of concurrent system construction as a composition of atomic processes using pipes as the glue in stark contrast to the construction of a monolithic sequential program.
- Establish an understanding of the design and development of systems software, such as command interpreters (`ksh`) and compilers (`gcc`), through the study of system libraries (`libc`), pattern matching and filters (`grep`, `sed`, and `awk`), interprocess communication (pipes and FIFOs), automatic program generation (`lex` and `yacc`), and signals (`SIGINT`).
- Establish a competency in Linux internals (e.g., inodes) and establish an understanding of Linux system calls (e.g., `open`, `close`, `read`, `write`, `fork`, `wait`, `execvp`).

## 2 Course Design

An instructor-authored book on Linux and C, made available to students for free, is the only required textbook. The recommended textbooks are [2, 4, 5, 7, 8, 9] and students have access to each through the University library.

This course tells a story: the first half of the course (Modules I and II) progressively cover the fundamentals of Linux and C/Go, and Linux systems programming, while the second half of the course demonstrates how the atomic tools and utilities covered in Module I can be creatively combined and composed with each other using the interprocess communication mechanisms, covered in Module II and built into the shell, to solve practical programming problems within an environment which supports software development in a timely fashion. The main themes running throughout Modules I and II are the uniform style of I/O in Linux, C, and Go; the interface of Linux and C/Go; and

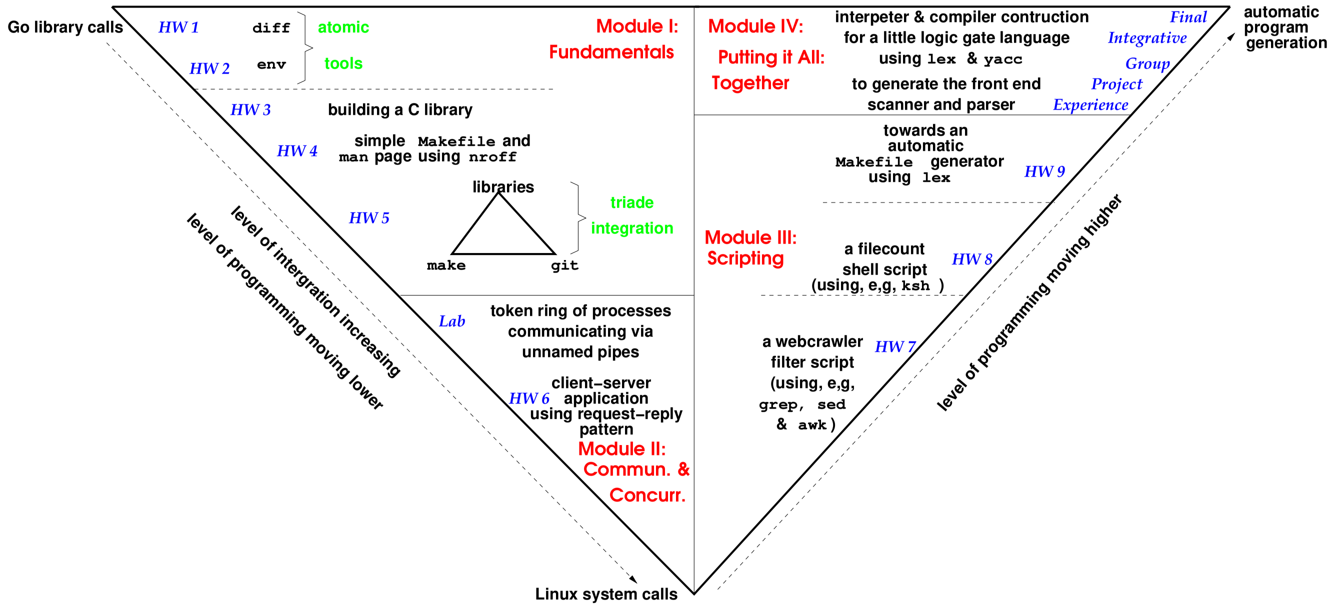


Figure 1: Graphical view of the sequence of course content modules, conceptual topics within each module, and the programming assignments therein.

the well-designed nature of the Linux OS and shell which fosters a powerful, stable, programming environment which has endured for half a century. The main theme running throughout Modules III and IV is the Linux filter style of programming (i.e., constructing software systems, such as specialized tools and utilities, by dynamically and creatively arranging multiple atomic existing tools as building blocks, using pipes as the interprocess communication mechanism). The central focus of the course is on establishing an understanding of these themes and ways of putting them into practice through supportive programming assignments. In other words, helping connect themes to applications. We can think of Modules II and IV as taking Modules I and III to a deeper level, respectively.

Homework assignments are programming exercises each of which require a fair amount of critical thought and design, and approximately 150 lines of code. The following is a list of sample assignment synopses, intended to relate the content and form of assignments that might be helpful to instructors inspired to teach a similar course. While adopters of this approach will undoubtedly tailor assignments, the guiding theme of the assignments as a whole is to serve as an evaluation mechanism within this course template. The series of assignments presented here represent one particular instantiation of that template, depicted graphically in Figure 1, and serve as a vehicle to convey the course motif. Thus, any series of programming and conceptual assignments that fits that general, or a similar, structure is appropriate.

(*MODULE I: FUNDAMENTALS*)

**Homework #1** involves implementing a simplified version of the Linux file comparison utility `diff` **in Go**, which will accept input from standard input or file input or a combination of both. This first assignment helps (re-)orient students to programming with standard libraries, especially for I/O and text processing and manipulation.

**Homework #2** has two parts—a conceptual exercise and a programming exercise—and both deal with the process environment. The conceptual exercise involves customizing the user environment through the setting and modification of shell variables (e.g., `ENV`, `PS1`, and `PATH`), exploring startup files (e.g., `.profile`, `.kshrc`, and `.vimrc`) and adding both Linux commands and shell-builtin commands to them (e.g., `alias` for creating command aliases). This is a fun exercise for students to tailor their programming environment to their tastes to help improve their productivity. We often spend class time as an activity-learning exercise for this part of the assignment. The programming exercise involves building the Linux utility `env` in Go (and is a modified version of [7, § 2.12 Exercise: An `env` Utility; pp. 54–55] in Go). This second part of the assignment introduces students to Linux system calls and the system call interface in Go (i.e., `package syscall`) and gives them experience with spawning and executing processes.

**Homework #3** involves **building an application programming interface (API) in Go** (as a `package`) to a simple linked-list, which can be used for purposes of issue or bug tracking in the development of a software system. Students are given both the interface containing the signatures of the functions, which they must define, as well as a sample application which must be linked to their library. Neither must be modified. This assignment reinforces to students the idea of programming with an API and the idea of factoring a system into three components: i) a public interface (e.g., a `.h` header file), ii) a private library implementation (e.g., a `.a` `ar` file archive of a collection of pre-compiled `.o` object files), and iii) a client application containing the main program. This programming exercise is a modified version of [7, § 2.12 Exercise: Message Logging; pp. 55–56] in Go.

**Homework #4** involves defining **a simple `Makefile`** for building both a utility `flip`, which converts DOS to UNIX newlines and verse versa, and its `man`page using `nroff`. Students are given multiple requirements on both the operation and style of the `Makefile` (i.e., naming conventions for targets such as `all` and `clean` and the use of variables such as `CC` to render it more readable and easily modifiable).

With this foundation in place, students are well equipped to start integrating some of the concepts and tools they now know.

**Homework #5 integrates three important topics explored in this module: i) implementation of a library and its use in a client application, ii) compilation management (`make`), and iii) configuration management (`git`).** Students progressively refine a `Makefile` for an application that utilizes two libraries for interacting with a linked-list data structure. Students [i)]

write and iteratively refine a `Makefile` for the project,

factor out some of the functions in the codebase and create a library from them, and

maintain versions of each iteration using Git on a local repository connected to a remote origin hosted in BitBucket.

Armed with this foundation of fundamentals, students are now ready to construct concurrent systems, which are built using the fundamental concepts from prior assignments (e.g., libraries), whose entities communicate with each other through interprocess communication mechanisms.

(*MODULE II: COMMUNICATION AND CONCURRENCY*)

**Homework #6** involves **implementing and experimenting with a client-server application in Go**. The application uses a synchronization barrier, where the barrier is implemented as a server to which clients communicate through named pipes. Clients communicate with the server using a library (i.e., a `package` in Go) which students implement and install (leveraging their experience with implementing, packaging, and linking libraries in Go in Homework #3). This programming exercise is a modified version of [7, § 6.8 Exercise: Barriers; pp. 221–222] in Go.

Equipped with the knowledge of building integrated and concurrent Linux programming structures, we turn our attention from the low-level details of constructing those structures to harnessing off-the-shelf structures/mechanisms to creativity solve a variety of practical programming problems—a higher-level activity.

(*MODULE III: SCRIPTING*)

**Homework #7** involves **implementing a Linux filter script** to scrape data from a webpage and apply a series of data transformations, using Linux filters, such as `cut`, `paste`, `join`, `sort`, `uniq`, `tr`, `grep`, `sed`, and `awk`, among a suite of others, to prepare the data for importation into a database system. Students are only given the final output; it is up to them to decide both which transformations to apply and concomitantly which filters to use. The goal of this assignment is to convey to students the plug-and-play flexibility (of the atomic processes) through the Linux filter style of programming. Another goal is to contrast the monolithic, construction of a (typically) sequential program versus the construction of a concurrent system as a composition of atomic processes using pipes as the glue. It is also important for students to understand that the processes in a pipeline execute concurrently, not sequentially, while all remain in synchronization due to the automatic blocking nature of Linux pipes.

**Homework #8** involves **building a version of the Linux `find` utility** as a Korn shell script which traverses a series of directories given at the command line and reports the number of plain files, executables, directories, and symbolic links encountered therein. The goal of this assignment is to convey to students the power of a programmable shell, and to contrast (sequential) shell programming with (concurrent) filter style programming (in Homework #7).

**Homework #9** involves **using `lex` to automatically generate a filter** capable of parsing a codebase of C and C++ source files and not only extracting, but also, differentiating between uncommented and commented header files therein. This assignment is the basis of an automatic `Makefile` generator, which is itself automatically generated using `lex`. The goal of this assignment is to introduce students to automatic program generation and lexical analysis as well as to expose them to yet another approach to filter construction.

(*MODULE IV: PUTTING IT ALL TOGETHER*)

The entire course is structured to prepare students for the **final, culminating project experience**, which ties many of a course concepts and themes into a robust and compelling, yet manageable, final project. The project involves building an interpreter and a compiler for a small logic language to C++. Students are advised to factor their system into the following three components: [i)]

a front end (i.e., a shift-reduce parser, automatically generated with `lex` and `yacc`, which produces a parse tree);

an interpreter (i.e., an expression evaluator); and

a compiler (i.e., a translator to C++).

## 3 Discussion

We have used and refined the approach espoused in this article in the *Linux Programming* course at the University of Dayton for nine consecutive offerings of it since Fall 2013 with documented student feedback. Feedback from anonymous student evaluations has revealed that this approach and, especially, the implementation-oriented nature of it, is effective at reinforcing core Linux concepts and themes.

We maintain a set of primary and supplemental material for this course online at `http://academic.udayton.edu/SaverioPerugini/LCP/`. The course webpage for the most recent offering of the *Linux Programming* course at the University of Dayton (Spring 2019), which contains links to the syllabus, course notes, readings, homework assignments, and projects, is available at `http://perugini.cps.udayton.edu/teaching/courses/Spring2019/cps444/`.

There is scope for customization within the general course framework presented here in both content and delivery. For instance, an instructor can make use of interactive learning and assessment tools such as *uAsssign* [1]. A focus on the Linux kernel through which to cover system calls (as opposed to the client-sever model as used here) is an alternate approach [3]. Similarly, more coverage of cybersecurity or mobile devices/OS can be infused into course [6].

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Numbers 1712406 and 1712404. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

[1] J. Bailey and C. Zilles. uAssign: Scalable interactive activities for teaching the Unix terminal. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 70–76, New York, NY, 2019. ACM Press.

[2] S.P. Harbison and G.L. Steele Jr. *C: A Reference Manual.* Prentice Hall, Englewood Cliffs, NJ, fourth edition, 1995.

[3] R. Hess and P. Paulson. Linux kernel projects for an undergraduate operating systems course. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 485–489, New York, NY, 2010. ACM Press.

[4] B.W. Kernighan and R. Pike. *The UNIX Programming Environment.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1984.

[5] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[6] J.-F. Lalande, V. Viet Triem Tong, P. Graux, G. Hiet, W. Mazurczyk, H. Chaoui, and P. Berthomé. Teaching android mobile security. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 232–238, New York, NY, 2019. ACM Press.

[7] K.A. Robbins and S. Robbins. *UNIX Systems Programming: Communication, Concurrency, and Threads.* Prentice Hall, Upper Saddle River, NJ, second edition, 2003.

[8] W. Schotts. *The Linux Command Line: A Complete Introduction.* No Starch Press, 2019. `http://linuxcommand.org/tlcl.php` [Last accessed: 12 June 2019].

[9] M. Summerfield. *Programming in Go: Creating applications for the 21st century.* Addison Wesley, Boston, MA, 2012.

[10] J.L. Wolfe. Reviving systems programming. In *Proceedings of the 23rd ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 255–258, New York, NY, 1992. ACM Press.

# An Introduction to Concatenative Programming in Factor[*]

*Zackery L. Arnold and Saverio Perugini*
*Department of Computer Science*
*University of Dayton*
*Dayton, Ohio 45469*
`saverio@udayton.edu`

**Abstract**

We provide an overview of Factor—a stack-based and concatenative programming language like FORTH. Factor incorporates the core concepts of the object-oriented (e.g., classes, polymorphism) and functional (e.g., higher-order functions) paradigms of programming. These features among others render Factor an emerging language worthy of wider consideration.

## 1 Introduction

Factor[1] is a concatenative programming language that incorporates the core concepts of the object-oriented (e.g., classes, polymorphism) and functional (e.g., higher-order functions) paradigms of programming [2]. Factor programs are written using chains of higher-order functions that manipulate data on a stack. Programs are developed using an interactive development environment called the Factor Listener. Code written by the user is immediately compiled by Factor's optimizing compiler and stored within the currently operating image of the language—mixing the ease of experimentation of interpreted languages with the performance of a compiled one. These features and rich libraries of helpful functions render Factor an emerging

[*]Copyright ©2019 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

[1]`http://factorcode.org/`.

language worthy of wider consideration. We explored Factor in an emerging programming languages course at the University of Dayton (see `http://perugini.cps.udayton.edu/teaching/courses/Spring2017/cps499/`).

## 2 Concatenative Programming

Concatenative programming languages, in the tradition of the language FORTH [4], involve the composition of multiple functions that cooperate to transform data [3]. Concatenative programming is similar to *pipelining* in UNIX systems. Programs in Factor are made up of higher-order functions that are written from left to right. Functions are identified using any sequence of non-whitespace characters, resulting in a line of code that nearly resembles a sentence in a natural language. Hence, in Factor, functions are typically referred to as *words* with whitespace serving as the *concatenation operator* [5]. The following is an example of concatenative Factor code that calculates `12!`.

```
12 [1,b] 1 [ * ] reduce
```

## 3 The Data Stack

Factor is also a stack-based programming language—all data in a program is loaded onto a shared data stack. Words that use this data pop off the stack and the results of the operation are pushed back onto it. Concatenated words, then, work together through the use of the shared stack to morph the input data into the desired output. The shared data stack is is the traditional run-time call stack of activation records used to implement functions [1]. Furthermore, the data stack itself is not directly related to a typical stack data structure, as the data stack does not encapsulate any methods like `push` or `pop`. Instead, the data stack is fully managed by the words used in the program. Manipulation of the data stack is fundamental to programming in Factor.

### 3.1 Stack Shuffling and Combinators

There are multiple words that are readily available to the programmer, such as `swap` (which swaps the two topmost elements), `drop` (which discards the top element), and `dup` (which duplicates the top element). While these words are vital to the success of more complicated Factor programs, they render the code verbose and less readable. As a result, the programmer is encouraged to adopt more powerful, higher-order functions that abstract some of the details associated with the language. Words in Factor that achieve this goal, such as `reduce` and `fold`, are called *combinators*.

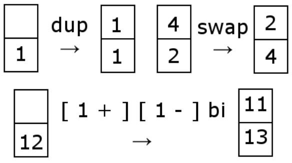


Figure 1: A visual representation of a few commonly used words in Factor code.

One commonly used combinator is the word `bi`, which takes a single value off of the stack and applies two separate functions to the value, leaving two results on the stack. For example, `12 [ 1 + ] [ 1 - ] bi` leaves the two separate values `13` and `11` on the top of the stack. The word `bi` may also be used for the evaluation of an `or` word as in the program `[ even? ] [ odd? ] bi or`. Other useful combinators include `cleave` (which applies an arbitrary number of words to a given value), `each` (which applies a word to all items of a list), `filter` (which returns the elements of a list that pass a given filter), and `map` (which applies a word to a list to receive a new list). Figure 1 illustrates some of these common stack shuffling and combinator words.

## 3.2 Stack Effect and the Stack Checker

A challenge associated with programming in a stack-based language is addressing the bugs that arise when words that manipulate the stack leave a varying number of items on the stack after completion. As a result, words defined in Factor have an explicit declaration that specifies how many items are popped off of and pushed onto the stack. This declaration, called a *stack effect*, is notated with `( a - b )`, where the words `a` and `b` are representative of the *effect* that the word has on the stack. In addition to program documentation, stack-effect declarations inform the program compilation process. For instance, words can only be compiled if they strictly follow the notation of their corresponding stack effect. Stack effects, in general, perform a simple role that is similar to pattern matching in other languages, such as Haskell. However, unlike Haskell, the words used to represent items on the stack are purely symbolic, meaning that they cannot be referenced in the body of the word like a lexically-scoped variable.[2]

The *stack checker* in Factor is a tool that acts similar to a type system [5]. The stack checker performs a brief simulation of the program before passing it to the compiler—checking that each branch of control in the program leaves the

[2]Factor does have library-support for lexically-scoped variables, but they are unrelated to the notion of stack effect.

stack at equal heights. If the word does not leave the stack with a consistent height, then a compile-time error is raised and the program is rejected. However, Factor does make some exceptions to this rule, specifically with regards to *row polymorphism.* In some situations the programmer may wish to use a combinator with sets of words that have different stack effects. Row polymorphism in Factor permits this, so long as the quotations (defined below) are only operating on data *below* itself on the stack—resulting in the same stack height as the other words used in control flow.

## 4 Other *Factors*

### 4.1 Quotations and Vocabularies

As mentioned above, functions in Factor are idiomatically called words. An *anonymous word*, called a *quotation*, is a word that is anonymously constructed from the composition of other words wrapped with the `[` and `]` *parsing words.* Table 1 lists some of the most common parsing words in Factor programs. Some words, such as `if`, take a quotation as an argument and use that quotation to transform the stack. Quotations can also be constructed using values already on the stack when using a pair of alternate words. For instance, `10 '[ _ = ]` results in the equality quotation `[ 10 = ]`.

Words can also be defined with formal names. New words are defined with a sequence beginning with the `:` parsing word, followed by the name given to the word, the stack effect of the word, the body of the word, and, finally, ending with the enclosing `;` parsing word that marks the end of a definition. For example, consider the following word `product` which multiplies the contents of a sequence:

```
: product ( seq –– p ) 1 [ * ] reduce ;
```

Multiple words like `product` above can be defined in a collection known as a *vocabulary*—the Factor analog of a library. Vocabularies are stored within a source code file that is in a sub-directory structure within the Factor installation directory. Each vocabulary must explicitly reference the vocabularies upon which it relies for operation. For example, a separate vocabulary that wishes to use the `product` word above would need to include the vocabulary in which `product` is defined. Explicit declaration of necessary vocabularies allow the Factor compiler to ensure minimization of the compiled code. Compiled Factor programs only include the bare minimum code needed for successful program execution.

Table 1: Commonly encountered parsing words.

| **Parsing Word** | **Semantics** |
|---|---|
| `: ... ;` | Denotes the start and end of a named word. |
| `( ... -- ... )` | Denotes a stack effect of a word. |
| `[ ... ]` | Denotes a quotation or anonymous word. |
| `{ ... }` | Denotes an array or vector sequence. |
| `!` | Denotes a comment until end of line. |

## 4.2 Object-Oriented Implementation

Factor is a *purely* object-oriented programming language—every value used is an object. Classes are divided into three primary types: *primitive classes*, *tuple classes*, and *derived classes* [5]. Primitive classes cannot be sub-classed and are used for types such as strings, numbers, and words. Tuple classes are more advanced and may contain instance variables that, unlike generic words, are owned by the class. Derived classes are built from other preexisting classes and can take multiple forms. For example, predicate, union, and intersection classes can be created in the same way one might interpret sets from set theory. A special example is a *mixin* class that is used often in Factor to collect groups of classes under a common interface [5].

Another unique feature of Factor is the way in which objects and object-specific words interact. Usually in object-oriented languages an object has direct access to methods that are intrinsic to it. Factor instead uses *generic words* that are redefined as needed to handle a given object-specific implementation—serving a purpose similar to template class functions in other languages, but with a less restrictive implementation due to the absence of the concept of ownership.

## 4.3 Development Workflow

The *Factor Listener* is the interactive development environment (IDE) and provides an interpretive command-line interface for the programmer to dynamically test new words without writing a complete program. When the programmer is ready to commit to a source file, the programmer can create a new vocabulary using the `scaffold` tools vocabulary and the `scaffold-work` word. A new directory is created with an empty source file and ready for the programmer. Once the file is successfully edited, the programmer can update Factor automatically with the new vocabulary by using either the `F2` key or the `refresh-all` word.

The Factor Listener provides other helpful features. For instance, unit tests are defined and automatically evaluated with the use of the `unit-test` vocabulary. Documentation files are created with the use of the `scaffold-docs` vocabulary and multiple words from the auto-included `help` vocabulary. In addition to user documentation, all of the Factor documentation is available from within the IDE through the help button at the top of the interface. Word execution is benchmarked at any time with the use of either the *<ctrl-t>* keystroke or the use of the `time` word. Because Factor uses an image-based compilation system, the `save-image` word is used to save the state of the current Factor instance to a file. Images then are loaded when the IDE is launched, resulting in a faster development experience without the need of reloading key vocabularies for every launch. Factor is also capable of deployment as a standalone executable with support for Windows, Linux, and Mac OS through the use of the `deploy` word. Moreover, Factor itself also has capable vocabularies that implement concepts from foreign function interfaces and UI toolkits to direct HTTP and SMTP support [5].

## 5 Discussion

Other concatenative programming languages include Joy, Retro (FORTH), PostScript, and Gershwin—a version of Clojure supporting a concatenative style of programming. The strengths of both the object-oriented and functional paradigms of programming are blended in Factor. An expressive syntax, a set of vocabularies, abstract classes, higher-order functions, and related features in Factor support the programmer in solving complex problems (e.g., an ID3 parser) in an elegant way. Applications of Factor include command-line utilities, graphical user interfaces, and web applications [2]. We hope that this article has inspired instructors to consider exploring concatenative programming in a language like Factor—both as an example of an approach which blends concepts of FORTH and LISP and in contrast to more typical applicative programming—in a programming languages course. A presentation on Factor by the student author is available at `https://www.youtube.com/watch?v=FjW4-5tGidk`. Moreover, course notes on Factor developed by the student author are available at `http://perugini.cps.udayton.edu/teaching/courses/Spring2017/cps499/Languages/notes/Factor.html`. In summary, in a *word*:

```
: Factor ( -- is ) "Object Oriented" "Functional Execution" + ;
```

## References

[1] Concatenative language. Available: `http://concatenative.org/wiki/view/Concatenative\%20language` [Last accessed: 12 June 2019].

[2] F. Daoud. Chapter 2: Factor. In J. Carter, editor, *Seven More Languages in Seven Weeks: Languages That Are Shaping the Future*. Pragmatic Bookshelf, Dallas, TX, 2014.

[3] D. Herzberg and T. Reichert. Concatenative programming: An overlooked paradigm in functional programming. In *Proceedings of the Fourth International Conference on Software and Data Technologies (ICSOFT)*, 2009.

[4] P. Koopman. A brief introduction to Forth. In *Proceedings of the Second ACM SIGPLAN conference on History of Programming Languages*, pages 357–358, New York, NY, 1993. ACM Press. Also appears in *ACM SIGPLAN Notices*, 28(3), 1993.

[5] S. Pestov, D. Ehrenberg, and J. Groff. Factor: A dynamic stack-based programming language. In *Proceedings of the Sixth Dynamic Languages Symposium (DLS)*, pages 43–58, New York, NY, 2010. ACM Press. Also appears in *ACM SIGPLAN Notices*, 45(12), 2010.

## Appendix: Factor Exercises

The following are some programming exercises that incorporate essential concepts in Factor. Solutions to these exercises are available in a Git repository at `https://github.com/saverioperugini/Emerging-Languages-Spring-2017/tree/master/Factor/src/`.

1) Define a word `caesar` in a vocabulary `exercises` that takes a string of alphabetical characters and an integer and applies the integer to each character in the string. The program must handle both positive and negative offset values. Only include uppercase and lowercase alphabetical characters in the output. For example, `ABCD` with an offset of 4 should produce `EFGH`. Divide the program into a primary word `caesar` and set any helper words as private. This exercise is solved in less than ten lines of code.

Examples:

```
> "SEESPOTRUN" 26 caesar
.
"seespotrun"
```

```
> "SEESPOTRUN" −26 caesar
.
"seespotrun"
```

```
> "ABCDEFG" 7 caesar .
"HIJKLMN"
```

2) Define a new tuple class `novel` that represents a fictional literary work. Include member variables that correspond to the title, author, genre, publisher, year of publication, and identification number. Use strings for the first four variables and integers for the last pair. Include a constructor `<novel>` that takes values for each variable as arguments and sets them automatically. Also include a word `book-print` that takes a novel as an argument and prints its details in a simple format.

Examples:

```
> "The Lion, the Witch, and the Wardrobe"
  "C.S. Lewis" "Fantasy"
  "Geoffrey Bles" 1950 1 <novel> .

--- Data stack:
T{ novel f "The Lion, the Witch,
and the Wardrobe" "C.S. Lewis"
"Fantasy" "Geoffrey Bles" 1950...
```

```
> book-print
Class: Novel
Title: The Lion, the Witch, and
the Wardrobe
Author: C.S. Lewis
Genre: Fantasy
Publisher: Geoffrey Bles
Year: 1950
ID: 1
```

3) Extend the solution to previous exercise with two new classes of books—`textbook` and `article`. For textbooks, change the genre field to subject. For article, change the genre field to discipline and add journal and volume fields. Then, with the three classes, define a `mixin` class called `library` that represents the union of different types of books. Adjust the `book-print` word from before to be generic with templates for each type of book. Publish the `library` and each class in a `library` vocabulary.

Examples:

```
> USE: library
> "The C Programming
     Language"
  "Ritchie, D. and
       Kernighan, B."
  "Computer Science"
  "Prentice Hall"
  1988 2 <textbook> .

--- Data stack:
T{ textbook f "The C
     Programming
Language"...

> book-print
Class: Textbook
Title: The C Programming
     Language
Author: Ritchie, D. and
          Kernighan, B.
Subject: Computer Science
Publisher: Prentice Hall
Year: 1988
ID: 2
```

```
> "Factor: A Dynamic
     Stack-based
  Programming Language"
  "Pestov, S., Ehrenberg,
       D., and
   Groff, J." "Computer
       Science"
  "ACM SIGPLAN Notices"
       45
  "ACM Press" 2010 3 <
       article> .

--- Data stack:
T{ article f...

> book-print
Class: Article
Title: Factor: A Dynamic
Stack-based Programming
     Language
Author: Pestov, S.,
     Ehrenberg, D.,
          and Groff, J.
Discipline: Computer
     Science
Journal: ACM SIGPLAN
     Notices
Volume: 45
Publisher: ACM Press
Year: 2010
ID: 3
```

# An Interactive, Graphical CPU Scheduling Simulator for Teaching Operating Systems[*]

*Joshua W. Buck and Saverio Perugini*
*Department of Computer Science*
*University of Dayton*
*Dayton, Ohio 45469*
*joshua.buck993@gmail.com, saverio@udayton.edu*

(A comprehensive version of this article, including screen captures for all figures referenced herein, is available as an arXiv technical report at `https://arxiv.org/abs/1812.05160` [1]).

**Abstract**

We present a graphical CPU scheduling simulation tool for visually and interactively exploring the processing of a variety of events handled by an operating system when running a program. Our graphical simulator is available for use on the web as well as locally by both instructors and students for purposes of pedagogy. Instructors can use it for live demonstrations of course concepts in class, while students can use it outside of class to explore the concepts. The graphical simulation tool is implemented using the React library for the fancy UI elements of the `Node.js` framework and is available as a web application at `https://cpudemo.azurewebsites.net`. The goals of this paper are to showcase the demonstrative capabilities of the tool for instruction, share student experiences in developing the engine underlying the simulation, and to inspire its use by other educators.

[*]Copyright ©2019 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

## 1 Introduction

We present a graphical simulation tool for visually and interactively exploring the processing of a variety of events handled by an operating system when running a program [2]. Our tool graphically demonstrates a host of concepts of preemptive, multi-tasking operating systems, including scheduling algorithms, I/O processing, interrupts, context switches, task structures, and semaphore processing [3].

Our graphical tool was designed to run on top of a solution to a text-based course programming project—here after referred to as the *underlying simulation engine*—in which students design and implement a system that simulates some of the job and CPU scheduling, and semaphore processing of a time-shared operating system (OS). The complete project specification for the underlying simulation engine is available at `http://perugini.cps.udayton.edu/teaching/courses/Fall2015/cps356/#midterm`. The architectural view of the underling simulation engine, which also serves to convey some of the project/system requirements, is shown in Figure 1. Students are familiar with the concepts of job and processing scheduling, non-preemptive and preemptive scheduling algorithms, as well as semaphores prior to working on this project. See `http://perugini.cps.udayton.edu/teaching/courses/cps346/lecture_notes/scheduling.html` (scheduling) and `http://perugini.cps.udayton.edu/teaching/courses/cps346/lecture_notes/semaphores.html` (semaphores) for more information.

While demonstrating OS concepts using physical computer hardware and real operating systems is effective, a software simulation is less expensive to develop and more easily configurable. For instance, users of our tool have control over both the time-based events and the parameters of the system (e.g., quantum size and main memory constraints) that are less easily controllable at the user level in a real computing system. Moreover, a visual simulation graphically reveals the internal processing of and event handling within an OS from which the user is typically shielded. The ability to step through the handling of events (e.g., new process creation, process termination, I/O completion, a context switch) enabled by our tool in user-defined steps of CPU time is formative in students' conceptualization, comprehension, and visualization of these complex processes at work within an OS.

The graphical simulation tool is implemented using the React library for the fancy UI elements of the `Node.js` framework and is available as a web application at `https://cpudemo.azurewebsites.net/`. Our graphical simulator is available for use on the web as well as locally by both instructors and students for purposes of pedagogy. Instructors can use it to for live demonstrations of course concepts in class, while students can use it outside of class to explore the concepts. Assigning the development of the underling text-based simula-

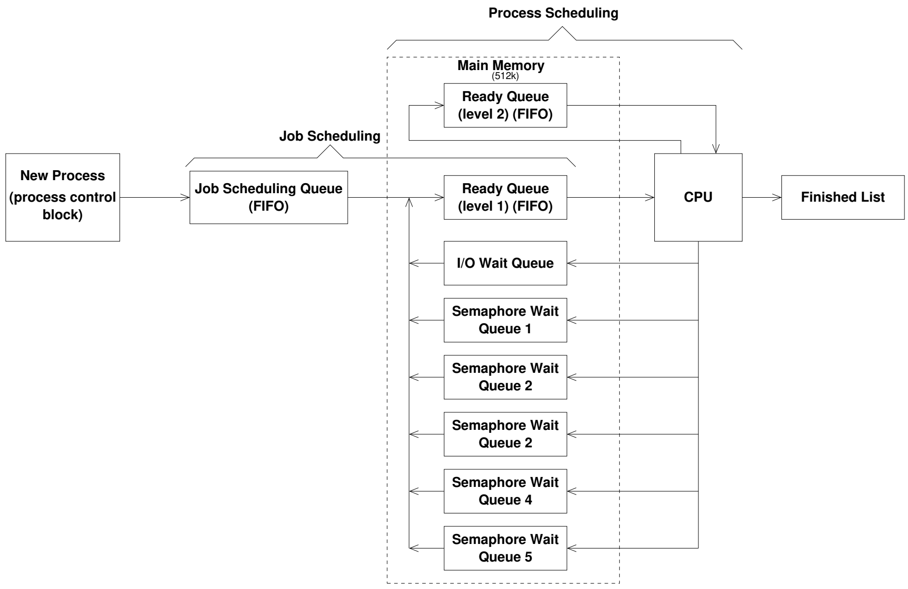


Figure 1: Architectural view of the underlying simulator depicting the transitions processes can take as they move throughout the various queues and the CPU of the system.

tion engine, on which the graphical simulator runs, to students as a course project is also an effective approach to teach students the concepts—more on this below.

## 2 Simulation Details

The top of the left-most column of the tool, shown in Figure 2, contains a list of incoming external events in the current session. Below the incoming external event list is a list of jobs rejected by the system because each requires more memory than the total system memory. At the bottom of the left-most column of the tool is the job scheduling queue, which lists all jobs that are waiting for main memory to become available before they can be moved to the ready queue. The middle columns of the tool contain the multi-level, FIFO ready queue, the I/O wait queue, and the semaphore wait queues. The right-most column of the tool contains a block representing the CPU, and a list of the completed processes. Above the CPU, the available system memory is shown.

The user can step through events in multiple ways. The top of the tool shows the current simulation time, which the user can modify at any time. Alternatively, the user can use the slider bar handle to advance or subtract up to 250 units of simulation time at once. Upon changing the time with the slider bar, the handle resets to the middle position and the user can again advance or subtract up to 250 units of time. There are also controls to allow the user to step forward to the next or backward to the previous simulation event. Finally, there is a run-until-complete option to fully run the simulator and produce the output (i.e., a variety of turnaround and wait time statistics) of the underlying text-based simulation engine in one stroke.

There are seven events in this simulator: four external events (i.e., given in the incoming external event list) and three internal events. The four external events are a new job arrival, an I/O request, and a semaphore wait and signal. The three internal events are process termination, quantum expiration, and I/O completion. If an external and internal event collide (i.e., occur at the same time), the internal event is processes first. The events are automatically processed in the background and the event navigation buttons allow a user to see every change that occurs in the simulation. At any point, the user may click the button labeled 'Reset Run' to return the simulation time to 0.

## 2.1 Customization: Tuning the Simulation Parameters

There are several ways in which a user can customize the simulation. The settings button opens a dialog window, shown in Figure 3, with several simulation variables that can be tuned. The user can set the quantum for each level of the FIFO ready queue. Setting the quantum to of a particular level of the ready queue to 0, sets the scheduling algorithm to be 'first-come, first-served' (FCFS)—a non-preemptive algorithm—for that level. The user can also select a simulation scenario (i.e., a sequence of incoming external events) from a drop-down menu of canned event scenarios. Alternatively, a user can upload their own simulation scenario as a text file. Creating and importing such custom sequences of simulation events is helpful for demonstrating specific scenarios, especially event collisions. The user can also change the current values of any of the semaphores and set the maximum memory available to the simulation. Jobs that require more memory than the system maximum are rejected and placed in the list of jobs rejected by the system. When a job is rejected by the system for any reason, an alert is displayed in the tool. There is a toggle available for disabling or enabling these alerts. Finally, another toggle is available that will simplify the tool layout by hiding the semaphore queues and enlarging the other queues. The simplified layout is shown in Figure 4.

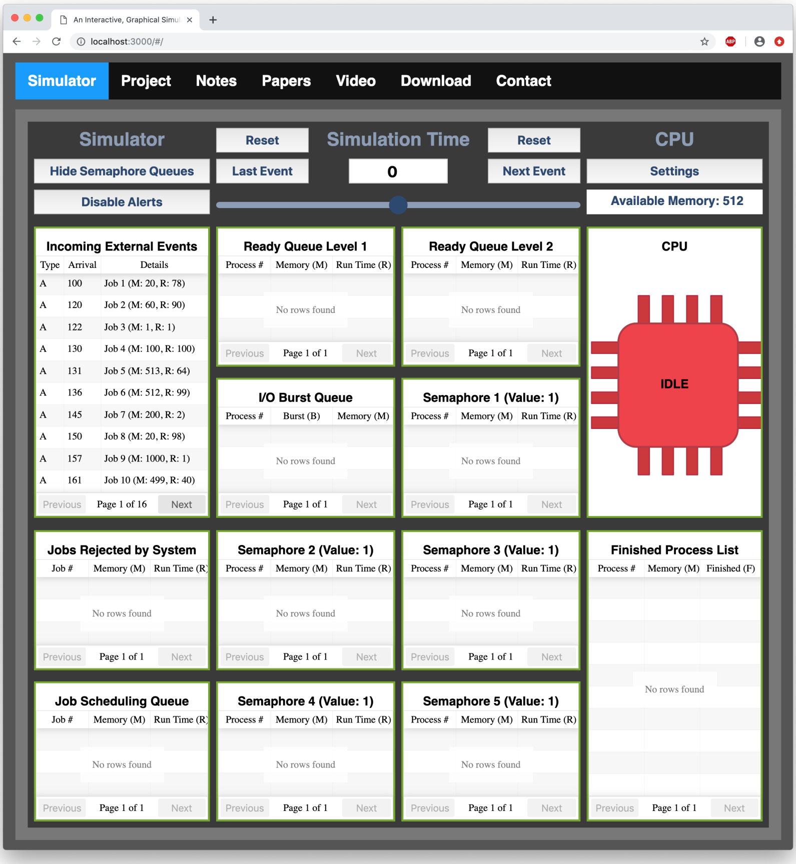
## 2.2 Example Simulation Scenario

(All of the screen captures referenced in this subsection are available in an arXiv technical report at `https://arxiv.org/abs/1812.05160` [1]).

Figure 2 shows the system before any events occur. In Figure 5 , 100 units of time have elapsed and the first event occurs—a new job arrival—identified by the letter 'A' in the first column of the incoming external events list (see Figure 2). Note that the job id, and runtime and memory requirements are also provided in the incoming external events list. The new job immediately moves into the job scheduling queue, which triggers the job scheduling algorithm. Since job 1 requires 20 units of memory and 512 units are available, job 1 is immediately loaded into the first level of the ready queue. Since the CPU is idle at this point, the arrival of a process in the ready queue triggers the CPU scheduling algorithm, which moves process 1 from the ready queue and onto the CPU with a quantum of 100 units of time. While this simulation does not currently factor in the time required for the overhead of a CPU context switch, process 1 does not run until the next clock cycle of the CPU. A summary of time stamp 100 is: the first job arrived and was instantaneously loaded onto the CPU (through the job queue and first-level ready queue). At time 101, shown in Figure 6, process 1 runs for 1 unit of the 78 required units of time before completion and has a remaining quantum of 99 clock cycles. Since the remaining quantum for process 1 is 99 units of time, the process would finish without time-slicing, but may require I/O or a semaphore in the interim. The incoming external events list in Figure 6 shows that the next incoming external event—another new job arrival—occurs at simulation time 120.

Clicking the next event button to the right of the simulation time automatically advances the time to the next event. At this point, the next event is the next incoming external event as opposed to an internal event. At time 119, process 1 has run for a total of 19 units of time, and requires 59 additional units of time until completion. During the next unit of time, shown in Figure 7, a new job arrives. It is moved into the job queue, which triggers the job scheduling algorithm. Since job 2 requires 60 units of memory, and there are 492 units available, job 2 is immediately loaded into the first level of the ready queue. Since the CPU is busy with process 1 at this point in time, process 2 must wait in the ready queue until the CPU is available.

Clicking the next event button twice more brings the simulation to time 130, shown in Figure 8, where two new jobs have been loaded into the ready queue. At time 131, there is an incoming external event for the arrival of job 5. However, job 5 requires 513 units of memory, which is greater than the main memory capacity of the system (i.e., 512 units of total memory that the simulation supports). In this simulation, since there is not enough memory to accommodate job 5, it can never run and, thus, is rejected with the alert



**Incoming External Events**

| Type | Arrival | Details |
|---|---|---|
| A | 100 | Job 1 (M: 20, R: 78) |
| A | 120 | Job 2 (M: 60, R: 90) |
| A | 122 | Job 3 (M: 1, R: 1) |
| A | 130 | Job 4 (M: 100, R: 100) |
| A | 131 | Job 5 (M: 513, R: 64) |
| A | 136 | Job 6 (M: 512, R: 99) |
| A | 145 | Job 7 (M: 200, R: 2) |
| A | 150 | Job 8 (M: 20, R: 98) |
| A | 157 | Job 9 (M: 1000, R: 1) |
| A | 161 | Job 10 (M: 499, R: 40) |

Previous Page 1 of 16 Next

**Ready Queue Level 1**

| Process # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Ready Queue Level 2**

| Process # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**I/O Burst Queue**

| Process # | Burst (B) | Memory (M) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Semaphore 1 (Value: 1)**

| Process # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Jobs Rejected by System**

| Job # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Semaphore 2 (Value: 1)**

| Process # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Semaphore 3 (Value: 1)**

| Process # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Finished Process List**

| Process # | Memory (M) | Finished (F) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Job Scheduling Queue**

| Job # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Semaphore 4 (Value: 1)**

| Process # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

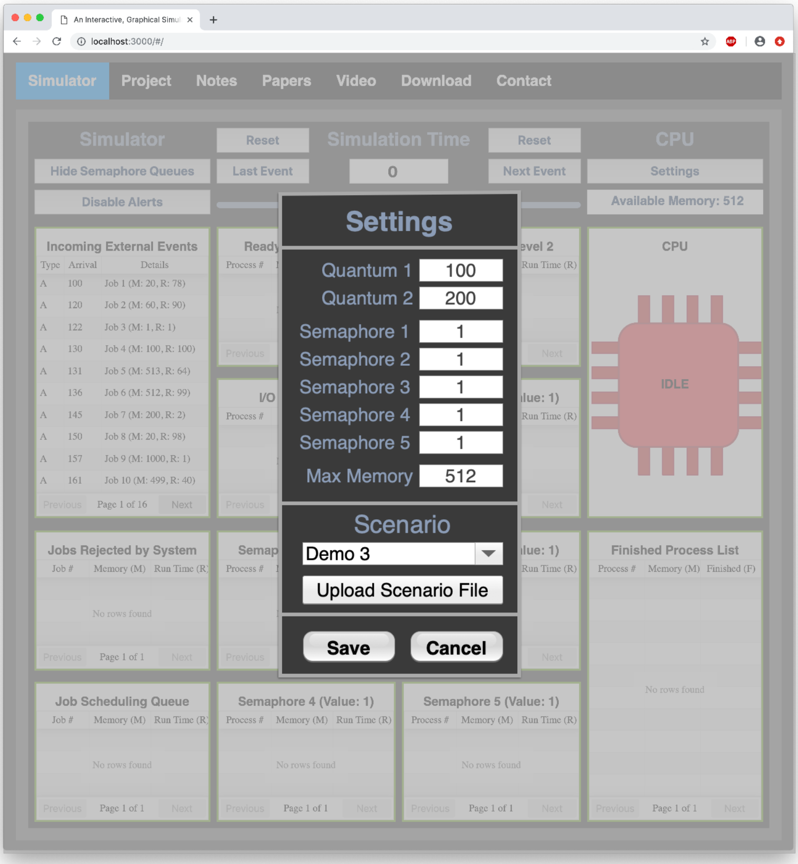
**Semaphore 5 (Value: 1)**

| Process # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

Figure 2: Main window of the simulator at time zero.



Incoming External Events

| Type | Arrival | Details |
|---|---|---|
| A | 100 | Job 1 (M: 20, R: 78) |
| A | 120 | Job 2 (M: 60, R: 90) |
| A | 122 | Job 3 (M: 1, R: 1) |
| A | 130 | Job 4 (M: 100, R: 100) |
| A | 131 | Job 5 (M: 513, R: 64) |
| A | 136 | Job 6 (M: 512, R: 99) |
| A | 145 | Job 7 (M: 200, R: 2) |
| A | 150 | Job 8 (M: 20, R: 98) |
| A | 157 | Job 9 (M: 1000, R: 1) |
| A | 161 | Job 10 (M: 499, R: 40) |

Page 1 of 16

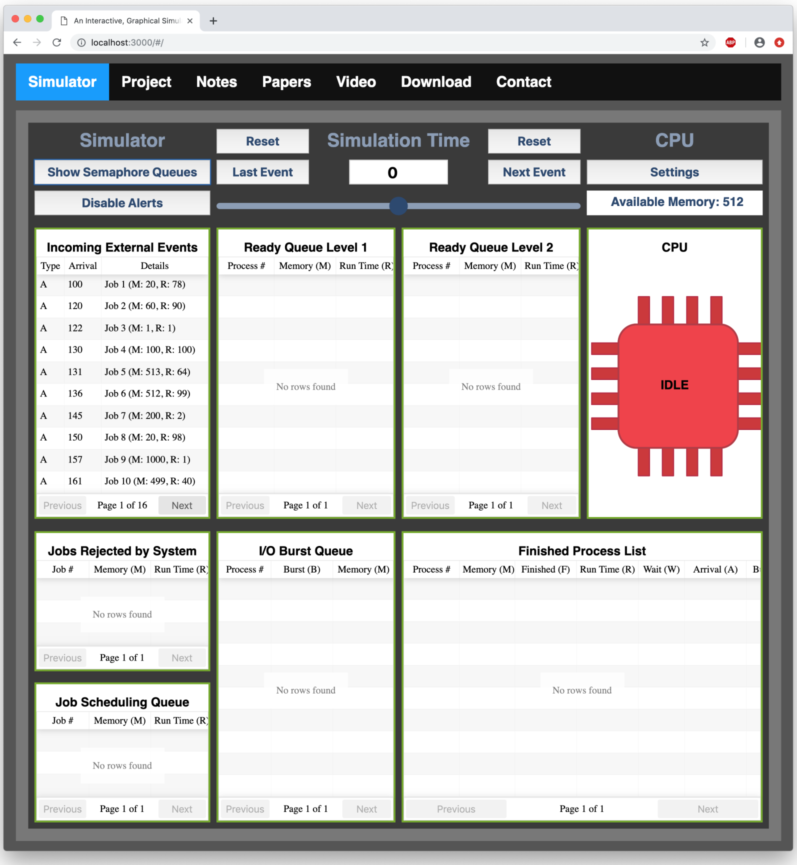
Settings

| | |
|---|---|
| Quantum 1 | 100 |
| Quantum 2 | 200 |
| Semaphore 1 | 1 |
| Semaphore 2 | 1 |
| Semaphore 3 | 1 |
| Semaphore 4 | 1 |
| Semaphore 5 | 1 |
| Max Memory | 512 |

Scenario

Demo 3

Figure 3: Simulator configuration settings available to users.



**Incoming External Events**

| Type | Arrival | Details |
|---|---|---|
| A | 100 | Job 1 (M: 20, R: 78) |
| A | 120 | Job 2 (M: 60, R: 90) |
| A | 122 | Job 3 (M: 1, R: 1) |
| A | 130 | Job 4 (M: 100, R: 100) |
| A | 131 | Job 5 (M: 513, R: 64) |
| A | 136 | Job 6 (M: 512, R: 99) |
| A | 145 | Job 7 (M: 200, R: 2) |
| A | 150 | Job 8 (M: 20, R: 98) |
| A | 157 | Job 9 (M: 1000, R: 1) |
| A | 161 | Job 10 (M: 499, R: 40) |

Previous Page 1 of 16 Next

**Ready Queue Level 1**

| Process # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Ready Queue Level 2**

| Process # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Jobs Rejected by System**

| Job # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**I/O Burst Queue**

| Process # | Burst (B) | Memory (M) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Finished Process List**

| Process # | Memory (M) | Finished (F) | Run Time (R) | Wait (W) | Arrival (A) | B |
|---|---|---|---|---|---|---|

No rows found

Previous Page 1 of 1 Next

**Job Scheduling Queue**

| Job # | Memory (M) | Run Time (R) |
|---|---|---|

No rows found

Previous Page 1 of 1 Next

Figure 4: A simplified view of the simulator with semaphore queues hidden.

popup message shown in Figure 9. This presents an opportunity to mention to students that in a virtual memory management scheme—a forth-coming topic—this job would be runnable, even though it exceeds the total amount of system memory. To disable alerts, a user can toggle the button labeled 'Disable Alerts' above the incoming external events list (see Figure 8). After closing the alert, the simulation time is 131, shown in Figure 10, where the rejected job 5 is in the list of jobs rejected by the system. At time 136, job 6, which requires exactly 512 units of memory, arrives. While this job enters the job queue (which is in secondary memory), it will not be permitted to enter the ready queue (which is in main memory) until all processes in main memory have fully completed (i.e., terminated). At that point, the full 512 units of memory are free and available for job 6.

In Figure 11, the simulation time is now 177 and process 1 requires only 1 unit of time before its completion. Figure 12 shows the result of running the simulation for 1 more unit of time after the button labeled 'Next Event' is clicked. Note that process 1 has moved to the finished process list, and process 2 has been loaded onto the CPU and requires 90 units of time to complete.

Moving forward to time 779, shown in Figure 13, we see that several processes have finished, several are in the first level of the ready queue, and many jobs are waiting in the job scheduling queue. In the incoming external events list, we see that the next event, identified with the symbol 'I,' occurs at time 780 and is a request for I/O by the process on the CPU—in this case, process 13. At time 780, shown in Figure 14, process 13 leaves the CPU and is moved to the I/O wait queue for the specified I/O burst. Once the I/O burst is complete, the process is moved back to the first level of the ready queue. Many other additional I/O events occur over the next few time steps.

Up to this point, the allotted quantum of 100 units has been sufficient for each process to finish before a quantum expiration. Thus, the second level ready queue is yet used. At time 1,569, shown in Figure 15, process 26 requires 2 units of time before completion, but its remaining quantum is only 1 unit of time. In Figure 16, process 26 is moved to the second level of the ready queue at time 1,570, and will not get back on the CPU again until the first level of the queue is empty since processes on the first level of the ready queue have priority over processes on the second level of the queue. While process 26 requires only 1 more unit of time to complete, it must now wait (potentially indefinitely leading to starvation due to the priority policy between the levels of the ready queue) for more time on the CPU. Students often raise questions about the efficiency of such a scheduling scheme. We use this opportunity to discuss the tradeoffs of scheduling algorithms and address potential solutions to starvation such as aging.

We now demonstrate the use of the system semaphores. We can toggle the button labeled 'Show Semaphore Queues' to restore the semaphore queues to the display. The first semaphore event is a signal which is identified by the 'S' symbol in the incoming external events list. This signal occurs at time 7,068 and signals semaphore 5 (see Figure 16). The availability of a semaphore is tracked next to the semaphore labels in the semaphore wait queues (see Figure 17). When a semaphore wait event, identified with a 'W' symbol, occurs, it causes the process on the CPU to acquire or wait on the identified semaphore. If that semaphore is available, its value is decremented and the process acquire the semaphore and, thus, remains on the CPU. If is that semaphore is unavailable (i.e., has a value of 0), the process on the CPU must block and, thus, is moved to the particular semaphore wait queue until the semaphore is available (through a subsequent signal). At time 7,449, shown in Figure 17, the simulation is 1 unit of time away from an incoming external event requiring process 57 on the CPU to acquire semaphore 4. Since the value of semaphore 4 is 0, process 57 blocks, leaves the CPU and must wait in the wait queue for semaphore 4 at time 7,450 (see Figure 18). For additional sample simulation scenarios, see a YouTube video of a text-based demonstration of the underlying simulation engine available at `https://youtu.be/eRU8h-5aM0s`.

## 3 Related Tool

A similar, albeit non-graphical, CPU scheduling simulator is available at `http://classque.cs.utsa.edu/classes/cs3733s2015/notes/ps/index.html` as a Java applet: `appletviewer http://classque.cs.utsa.edu/classes/cs3733/scheduling2/index.html`. This tool allows the user to explore a variety of CPU scheduling algorithms (e.g., FCFS, SJF, PSJF, and RR). The user chooses an algorithm, sets a quantum, and inputs an arrival time, CPU burst time, and I/O burst time for each process in a set of user-defined processes. The tool then produces a text-based Gantt chart. When a change is made to any of these inputs, the Gantt chart is updated. There are some key differences between this tool and our tool with implications on student learning. While our tool, like this tool, does permit the user to dynamically tune the quantum, unlike this tool, our tool does not permit the user to select the scheduling algorithm—the RR scheduling algorithm is fixed in our tool. However, and more importantly, unlike this tool, our tool i) simulates and displays the variety of queues involved in CPU scheduling and I/O and semaphore processing, ii) supports the user in stepping through the simulation at user-defined increments of time, and iii) allows the user to dynamically tune more simulation parameters than just quantum. In short, unlike our tool, this related tool does not capture the transitions from one unit of time to the next. (It also has an upper bound on how many processes can be simulated before the textual Gantt chart becomes unreadable. Also, the Java web applet is not secure and is blocked by most modern web browsers by default.)

Our tool allows users to interactively step through the simulation by any increment of time and observe both the state and location of all processes. Rather than displaying only the state of processes, our tool also illustrates the queues in which each process resides throughout its lifecycle. For example, in our tool, users can monitor a process as it is loaded into memory, inserting into the ready queue, granted access to the CPU, preempted to a semaphore or I/O wait queue, time sliced and moved to the second-level ready queue, and eventually completed and flushed out of the system onto the list of finished processes. Moreover, our tool supports the dynamic tuning of many of the simulation parameters. For instance, users can inject or edit incoming events at any time (e.g., altering the semaphore signals at any increment of time along the simulation timeline). The ability to step backwards also allows users a convenient way to explore multiple scenarios from a given point in time. This level of interaction supported by our tool provides ample scope for students to explore CPU scheduling in a variety of user-created scenarios. Lastly, unlike this tool, our tool also involves memory usage and multi-level feedback queues.

## 4 Student Feedback and Discussion

Students across a wide range of offerings of the OS course have found the project to develop the underlying simulation engine helpful for discerning and acquiring an appreciation of the difficulty in the copious event processing an OS must handle. It also gives them a feel for the operations management nature of an OS. The following is a sample of anonymous student quotes from a course evaluation.

The project really nailed in the main concepts of operating systems in general.

The project was also an interactive and engaging experience that demonstrated and explained concepts we were working on in class.

I found that the project really helped me learn how an operating system scheduler worked.

Also I found the project to be really fun, I actually enjoyed working on it.

. . . mostly the project that we did halfway through the semester was very beneficial to my learning looking back at it.

Another use of our tool is as an aid to students working on conceptual, pencil-and-paper process scheduling exercises (such as those in [3][Chapter 5]), particularly for verifying the correctness of their work. In addition to its use for exploring the demonstrated concepts, we have discovered an unintended use of this graphical tool—students use of it as a tool to debug their underlying simulation engine. Observing the operation of their underlying simulation graphically helps students identify bugs in their implementation more quickly than wading through pages of textual dumps of their system queues, e.g., to identify a process that went awry.

Approximately three hundred and twenty students have completed the underlying simulation engine project since the Fall 2009 semester. Students are permitted to use any programming language of their choice for implementation. Students have primarily used Java (210) and C++ (95)—the languages used in our introductory sequence.[1] Students have also used Python (12), C$^\sharp$ (2), and Perl (1). Two students re-implemented their projects—one in Scheme and one in Elixir. (The Elixir program was multi-threaded—see below.)

There are multiple extensions to the underlying simulation engine that can be assigned to students as follow-on projects. For instance, students can implement a memory management scheme (e.g., paging) to the organization of the ready queues and/or simulate a multi-core processor.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Numbers 1712406 and 1712404. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The source of the underlying simulation engine is a course project designed by John A. Lewis in which students design and implement a program that simulates some of the job and CPU scheduling, and semaphore processing of a time-shared operating system.

## References

[1] J.W. Buck and S. Perugini. An interactive, graphical CPU scheduling simulator for teaching operating systems. Technical Report arXiv:1812.05160 [cs.OH], Cornell University Library: Computing Research Repository (CoRR), 2019. Available at `http://arxiv.org/abs/1812.05160`.

[2] J.W. Buck and S. Perugini. An interactive, graphical simulator for teaching operating systems. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*, New York, NY, 2019. ACM Press. Demonstration; DOI: `http://doi.acm.org/10.1145/3287324.3293756`.

[1]We switched from C++ to Java in Fall 2014, but C++ was phased out progressively over the span of a few semesters in the three-course sequence.

[3] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating system concepts.* John Wiley and Sons, Inc., Hoboken, NJ, tenth edition, 2018.

# Stepwise Refinement in Block-Based Programming[*]

*Josiah Krutz, Harvey Siy, Brian Dorn, Briana B. Morrison*
*Department of Computer Science*
*University of Nebraska at Omaha*
*Omaha, NE 68182*
{jkrutz,hsiy,bdorn,bbmorrison}@unomaha.edu

**Abstract**

With the popularity of block-based programming in CS0 courses, a growing number of students will learn a block language in their first exposure to programming. Studies indicate that blocks make it easier for novices to complete simple tasks and help them maintain or increase interest in computer science. But as they attempt more complicated programs, the mere ease of dragging and dropping blocks will not be sufficient to help them think through the process of composing blocks into a working program. We propose to incorporate stepwise refinement into block programming environments as an approach for novices to work out the computational thinking processes needed to write more complex programs. We present a prototype, developed by modifying the Google Blockly platform, to facilitate stepwise refinement. We conduct a small exploratory study with high school AP CS0 teachers to determine the feasibility of this approach and report on the teachers' feedback.

## 1 Introduction

In our experiences teaching Computer Science Principles [1], a CS0 course for college freshmen, one of the most common frustrations expressed by students who are learning to program is, "I know what I want to do, but I don't know how to tell the computer how to do it." This mismatch between the idea

[*]Copyright ©2019 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

they have in mind (their mental model) and the capabilities afforded by the programming environment (its computational model) is an age-old problem.

Stepwise refinement [17] is a traditional approach for addressing this mismatch. It is an interative process where the specification of a program is successively refined until a complete program is coded. A high level natural language statement is decomposed into statements of smaller tasks which, when completed, will accomplish the statement. The process is repeated, with program elements being slowly incorporated into finer-grained statements, until only program elements are exclusively used. Also known as top-down programming, top-down design, or top-down decomposition, this approach was widely used in teaching programming in the 1970's and 1980's [12, 2]. Its pedagogical prominence may have declined somewhat over the years, especially with the advent of object-oriented programming languages that shifted the design focus to objects and their interactions, leading to a bottom-up programming approach [2]. Nevertheless, stepwise refinement is still useful in the programming of algorithmic steps inside functions and methods.

We aim to enable stepwise refinement as a first-order activity in block-based programming. A growing number of beginning programmers use a block language as their first programming language. While blocks ease the learning curve of beginning programming, block programming by itself does not address the programming barrier described above. Moreover, the ease of dragging and dropping blocks could encourage a trial-and-error approach to compose programs and make it difficult to instill a discipline for systematic program development, such as iterative refinement [8]. Current block programming environments do not provide features that facilitate stepwise refinement, which utilize informal statements, e.g., pseudocode, in intermediate stages.

Our contributions are as follows:

1. We introduce new features to a block-based programming environment that enable top-down stepwise refinement to help beginners express what they want to do within the block environment.
2. We report on an exploratory user study to determine the feasibility of using this implementation.

## 2 Background and Related Work

Block programming has emerged as a widely accepted approach for teaching programming to students with no prior experience. The block programming language Scratch [10] came onto the scene just as there was growing public awareness of the importance of K-12 computer science education [16]. Coupled with the presence of earlier block programming languages like Alice [3], the advent of Scratch led to the nascent adoption of block programming as the *de* *facto* approach for teaching beginning programming, as well as the proliferation of many block programming languages.

### 2.1 Empirical Studies of Block-Based Programming

There have been many studies conducted to assess the efficacy of using block programming languages for teaching beginners compared to conventional text-based programming languages (e.g., [15], [9]). Results indicate that block-programming does lower the barrier to learning simple programming but also limits the complexity of the programs students can write. At the same time, student engagement increased and they show higher levels of interest in pursuing future computing pathways.

On the other hand, Schanzer, et al. [11] observed that blocks do not lend themselves well to paper, being harder to draw than writing text. They conclude that block environments are an all-or-nothing approach; the programmer has to work out her thinking exclusively in the environment, losing the flexibility of writing out and thinking through problems on paper.

### 2.2 Challenges

Studies also report challenges related to student perceptions that block programming lacks both authenticity and expressive power [14]. While block programming raised the level of abstraction of programming, the intrinsic cognitive load of writing non-trivial programs remains [7]. Given a complex enough problem, there will always be a gap between the mental models held by novice programmers and the computational models afforded by the blocks in existing block programming environments. The all-or-nothing nature of block environments [11] suggests that effective approaches for bridging this gap must occur with additional affordances provided within the environment itself. We begin to address the mental to computational model gap by providing additional tools to programmers to bridge the gap.

## 3 Implementing Stepwise Refinement with Blocks

To explore stepwise refinement as a first-order task, an initial prototype was developed within the Blockly platform [5]. Two new blocks were added: a pseudocode block and an adapter block.

### 3.1 Pseudocode Blocks

The main addition is the pseudocode block, a block that lets the programmer write a natural language pseudocode sentence (see Figure 1). A programmer

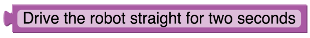


Figure 1: A pseudocode block example.

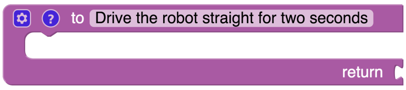


Figure 2: A pseudocode definition block example.

can refine a pseudocode block by opening its definition (see Figure 2) and adding executable blocks or more detailed pseudocode.

The pseudocode block reimplements the function call block. The difference from Blockly's functions is the programmer does not have to perform the extra step of dragging a function definition first before using the function call block. Because pseudocode blocks are functions, they can be parameterized as well. A pseudocode block can also be reused, sharing the same definition.

A pseudocode definition block is created automatically when a pseudocode block is dragged from the toolbox, but it is initially hidden so the programmer can focus on writing the pseudocode statement. The definition can be opened on demand beside the pseudocode block if a programmer wishes to refine it.

Implementing pseudocode sentences as functions is logical; pseudocode sentences are abstractions that hide more detailed procedures just as functions are abstractions that hide more detailed behavior. Alternatives considered, such as implementing as a comment or a special "no-op" block, lack the ability to be refined.

## 3.2 Adapter Blocks

Pseudocode blocks can be used wherever expressions are allowed (Figure 3). As expressions cannot be used where statements are needed, an adapter block is designed which effectively turns an expression into a statement (see Figure 4). This eliminates having a separate "pseudocode statement" block and simplifies novice programmers' choices. Quite often, novices may not know in advance if the pseudocode has to be an expression or a statement. If she creates a pseudocode statement block and later changes her mind because an expression is needed, (i.e., the pseudocode needed to return a value) then she has to create a new pseudocode expression block and copy the statement and definition, and delete the old block and its definition. Eliminating this upfront decision (statement vs. expression) makes it less onerous for the novice to perform steppwise refinement.

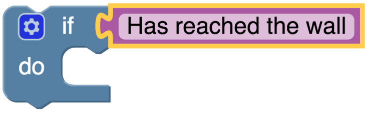


Figure 3: A pseudocode block used as condition in an *if* statement.

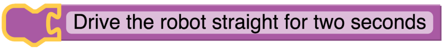


Figure 4: Adapter block transforms an expression into a statement

## 4 Exploratory User Study

To determine the readiness of the implementation for classroom use, we conducted a small exploratory study. Specifically we wanted to evaluate this proof-of-concept design by asking:

1. What usability issues exist in the current prototype that should be addressed before conducting classroom studies?
2. What pedagogical insights do teachers have about introducing stepwise refinement to their students using this tool?

### 4.1 Participant Selection

We turned to our high school computing colleagues teaching equivalent AP Computer Science Principles courses, who provided expert feedback. While this constitutes an inherently small pool of potential participants, we felt they would be well equipped to give constructive feedback to improve the current implementation and reflect on the feasibility of using it to teach stepwise refinement in a block programming-based class. As the implementation matures, we anticipate additional studies with novice students and their teachers.

### 4.2 Procedures

For the purpose of the study, we added the pseudocode block extensions to Blockly Turtle [6], an environment for creating turtle graphics on Blockly.

We conducted a separate session with each participant. We first walked them through a worked programming example to draw 5 stars. We then briefly explained the stepwise refinement process and illustrated it by walking them through the same example but using pseudocode blocks. We then gave them a similar programming task: draw 5 snowmen. The participants were asked to work through the task using the think aloud protocol [4], explaining their actions as they worked. A short debriefing interview was conducted at the end to gather opinions on the usability of the implementation as well as its feasibility for teaching.

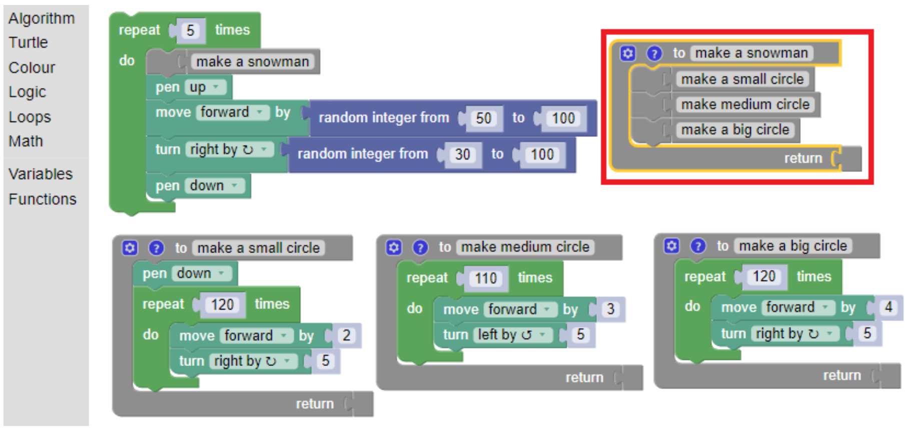
## 4.3 Results

Five high school teachers participated in this evaluation. All of them are under 40 years old and have started teaching high school AP computer science courses in the last 5 years, with education degrees from a variety of disciplines: math, business, and English. None of them teach stepwise refinement *per se*, but they do touch on this idea implicitly alongside related concepts of algorithm writing and problem decomposition. All of them completed the task successfully in less than 10 minutes. In the subsections to follow, we organize participant feedback somewhat thematically to present a view of their first impressions interacting with the pseudocode block features as a proof-of-concept rather than a formal qualitative analysis of actual classroom deployment. We include direct quotes from the participants where appropriate.

### 4.3.1 Comments related to the Implementation

**Relationship to functions.** Four participants picked up on the similarity with functions. One participant was concerned that there may be potential confusion with having two blocks that essentially do the same thing, since he covered functions early in his class. Two participants suggested putting the two blocks next to each other in the toolbox and explaining the difference to students. One participant suggested removing functions altogether and simply renaming the pseudocode block as the function block. Future studies would be needed to explore the advantages and disadvantages these alternatives.

**Parameter usage.** Though participants picked up on the similarity to functions, we observed only one participant actually make use of function parameters. The task (to draw a snowman) had them draw 3 circles of differing sizes. This naturally lends itself to programmers defining abstractly how to draw a circle once and then using the same definition with different radii. Two other participants did consider parameters but did not use them, preferring to use a global variable to pass the radius. In the post-test debrief, several commented that parameters were not obvious since they can only be added in the definition, which is Blockly's default behavior for functions. They suggested that parameters should be an option on the pseudocode block itself.



```
repeat 5 times
do
    make a snowman
    pen up
    move forward by random integer from 50 to 100
    turn right by ↻ random integer from 30 to 100
    pen down

to make a snowman
    make a small circle
    make medium circle
    make a big circle
    return

to make a small circle
    pen down
    repeat 120 times
    do
        move forward by 2
        turn right by ↻ 5
    return

to make medium circle
    repeat 110 times
    do
        move forward by 3
        turn left by ↺ 5
    return

to make a big circle
    repeat 120 times
    do
        move forward by 4
        turn right by ↻ 5
    return
```

Figure 5: A participant's snowman program. The box shows pseudocode for drawing a snowman.

### 4.3.2 Comments related to Stepwise Refinement

**Stepwise refinement in practice.** Three participants struggled to follow the stepwise refinement practice, first creating the program blocks for drawing a circle and upon thinking of the second circle, created the corresponding pseudocode block and moved the program blocks into the pseudocode definition, essentially a bottom-up approach. On the other hand, two participants created 3 different pseudocode blocks, one for each circle (see, e.g., Figure 5) and performed stepwise refinement to define their behavior.

**Teaching stepwise refinement.** Participants supported introducing stepwise refinement formally to their students as a systematic program development process. One teacher commented that students struggle with planning code from scratch. Three felt that it would help students think through the steps needed before actually writing code:

- "It makes students think about the big picture first. They can focus on the bigger steps and worry about the details later."
- "I want to do this, now this is how I will do it. Establishing the outcome first is beneficial."
- "It is a way to teach students to break things up like functions."

**Teaching functions and decomposition.** Closely related to stepwise refinement is the concept of decomposition. Our participants viewed functions more as a way of consolidating repetitive code and less as a way of encouraging decomposition. Two participants commented that the pseudocode block would make it easier to encourage students to use functions as they do not have to do the separate step of dragging the definition first. Another participant reflected that students often get careless in naming functions when being preoccupied with the function definition, so pseudocode blocks might encourage students to provide a more descriptive name for their function before actually writing its contents, "I have to name this thing before I start being concerned with the body."

**Using the modified Blockly platform in class.** The overall impressions of the participants on the prototype were positive as a replacement for hand-writing pseudocode on paper. One remarked that the system could be used during the very first examples of algorithm writing. With students in CS often preferring to use computers to paper, another felt that allowing students to write pseudocode on the computer might help them take it more seriously. Using a single platform for writing algorithms and having them seamlessly be refined into programs may instill the discipline of writing algorithms first while providing greater authenticity than paper-based planning. Exploring the perceived value of these alternatives to planning would be promising future work.

## 5 Conclusion

We presented a prototype block programming environment that supports top-down stepwise refinement. Though stepwise refinement is an old idea, there is no mechanism to explicitly support it in most programming environments. In particular, block-based environments make it even harder to sketch out pseudocode by restricting available program statements to a palette. With our implementation, stepwise refinement can be carried out in top-down fashion by successively filling in pseudocode definitions.

While these features enable a greater emphasis on top-down planning in block-based programming, we recognize the importance of epistemological pluralism [13] in constructionist learning environments and do not mean to imply that top-down work is inherently superior to bottom-up explorations in computing, especially in the early contexts in which block-based systems are regularly used. Indeed, our pseudocode can be written in a mix of programming constructs and natural language rather than focusing on formal function definitions upfront. As each step defined in natural language is successively refined in a top-down fashion, a record of the computational thinking process that occurred during the refinement process can be maintained and reviewed.

At the same time, it supports bottom-up refactoring to create named abstractions in the learner's own language. Ultimately, our objective is to enable both natural top-down and bottom-up modes for a learner while writing block code.

While we have not yet shown if this environment can indeed help novice programmers bridge the gap between ideas and their realization, our exploratory user study here is encouraging. The teachers agreed that students should benefit from learning stepwise refinement. Moving pseudocode writing from paper to the programming environment may encourage students to write algorithms that can be refined into working programs.

Our future work is to prepare a larger study with novice programmers in CS0 courses after addressing the usability isuses identified here and integrating our system in a more general Blockly environment. We also plan to design lessons to introduce stepwise refinement in the context of block programming.

## Acknowledgment

This work is funded in part by the National Science Foundation under grant DRL-1433788. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

## References

[1] Andrea Arpaci-Dusseau et al. Computer Science Principles: analysis of a proposed advanced placement course. In *Proc. of Tech. Symp. on Computer Science Education (SIGCSE)*, pages 251–256. ACM, 2013.

[2] Michael E Caspersen and Michael Kolling. STREAM: A first programming process. *ACM Trans. on Computing Education (TOCE)*, 9(1):4, 2009.

[3] Stephen Cooper, Wanda Dann, and Randy Pausch. Alice: a 3-D tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116, 2000.

[4] Karl Anders Ericsson and Herbert Alexander Simon. *Protocol analysis.* MIT Press, 1993.

[5] Google. Blockly | Google Developers. `https://developers.google.com/blockly`, 2018.

[6] Google. Blockly Games: Turtle. `https://blockly-games.appspot.com/turtle`, 2018.

[7] Raymond Lister. Programming, syntax and cognitive load. *ACM Inroads*, 2(2):21–22, 2011.

[8] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Habits of programming in Scratch. In *Proc. of Annual Joint Conf. on Innovation and Technology in Computer Science Education (ITiCSE)*, pages 168–172. ACM, 2011.

[9] Thomas W. Price and Tiffany Barnes. Comparing Textual and Block Interfaces in a Novice Programming Environment. In *Proc. of Intl. Conf. on Computing Education Research (ICER)*, pages 91–99. ACM, 2015.

[10] Mitchel Resnick et al. Scratch: programming for all. *Comm. of the ACM*, 52(11):60–67, 2009.

[11] E. Schanzer, S. Krishnamurthi, and K. Fisler. Blocks versus text: Ongoing lessons from Bootstrap. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 125–126, October 2015.

[12] Elliot Soloway. Learning to program = learning to construct mechanisms and explanations. *Comm. of the ACM*, 29(9):850–858, 1986.

[13] Sherry Turkle and Seymour Papert. Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, 11(1):3–33, 1992.

[14] David Weintrop and Uri Wilensky. To block or not to block, that is the question: Students' perceptions of blocks-based programming. In *Proc. of Intl. Conf. on Interaction Design and Children (IDC)*, pages 199–208. ACM, 2015.

[15] David Weintrop and Uri Wilensky. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Trans. on Computing Education (TOCE)*, 18(1):3, 2017.

[16] Jeannette M Wing. Computational thinking. *Comm. of the ACM*, 49(3):33–35, 2006.

[17] Niklaus Wirth. Program Development by Stepwise Refinement. *Comm. of the ACM*, 14(4):221–227, April 1971.

# $1000_{(binary)}$ Ways to Help New, Visiting, and Adjunct Faculty[*]

*Henry M. Walker*
*Department of Computer Science*
*Grinnell College*
*Grinnell, Iowa 50112*
*walker@cs.grinnell.edu*

**Abstract**

With increasing enrollments in computing courses nationwide, many schools are hiring more computing teachers—new permanent faculty, visiting faculty, adjunct faculty, etc. Each new teacher, of course, requires some orientation and initial introduction to the school, department, overall curriculum, and specifics of courses to be taught. Altogether, smooth entry into this educational framework requires addressing numerous matters, ranging from low-level logistical details to high-level educational and administrative frameworks. This paper reviews many of the practical matters faced by beginning faculty, providing a type of annotated checklist to aid both departments and new faculty themselves in easing the newcomers into the ongoing operation of a school.

## Introduction

Colleges and universities throughout the United States (and beyond) are experiencing exploding enrollments within computing programs. For example, in October 2017, the National Academies of Science, Engineering, and Medicine released a prepublication report on "Assessing and responding to the growth of computer science undergraduate enrollments" [1]. Today, many schools are increasing class sizes, increasing teaching staffs, limiting enrollments, or finding other creative approaches to accommodate more students. Various resources

[*]Copyright ©2019 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

have emerged to address issues of high or increasing enrollments, and Eric Roberts has developed a particularly helpful Web page, “Resources for the CS Capacity Crisis”, with links to recent studies and reports [2].

One common approach to handle large enrollments involves hiring new, visiting, and adjunct faculty. Although these teachers support new and expanded sections of courses and accommodate student needs, a new teacher may find the first several weeks to be particularly stressful and frantic:

- how do logistics work, how does one get paid, what support structures are available,
- what policies and practices are in place at the school or program, and how do they apply to a new faculty member’s teaching,
- what computing equipment is available in one’s office, classrooms, or labs,
- what classes will be taught and how do they fit into the broader curriculum,
- what are the expectations for course planning, syllabi, schedules, etc.,
- what are the expectations and policies for the evaluation of students and faculty, and
- what support services (e.g., tutoring, medical or mental health assistance) are available?

At some schools, orientation programs help address such matters. However, in this author’s experience, schools vary greatly regarding initial contacts and guidance for new faculty, and sometimes elements are missed (e.g., will be new person be paid on time).

This paper addresses such matters by providing an annotated checklist of topics, issues, policies, and practices, relevant as new faculty member begin their work. Naturally, an exhaustive list of all possible details at the start of a teaching position could fill volumes. Instead, this paper compiles within a reasonably concise framework many of the points that the author wishes had been tabulated when starting several recent visiting positions.

## 000: Logistics

A new person arriving on campus faces many basic elements of everyday life. For faculty and staff already established, logistical details may seem automatic and clear. In this author’s experience, orientation programs and on-going departmental colleagues often try to cover many of the day-to-day details, but some pieces inadvertently may be skipped, leading to substantial expenditures of time and effort.

The following subjects likely are handled differently from department to department and school to school, with different offices and individuals responsible for various tasks. In order to get started, a new person must navigate each of these areas at an early stage.

## 000.000: Human Resources Forms

A new faculty member likely must complete forms, such as proof of citizenship, tax forms, benefits, insurance, deductions, automatic deposit of paychecks. Although a Human Resources Office often handles these tasks, sometimes a Treasurer's Office or Dean's Office processes paperwork for salaries or stipends.

## 000.001: ID, Keys

Typically, new faculty require school IDs and keys to relevant buildings, offices, and classrooms. Such services may be handled by a wide range of offices.

## 000.010: Campus Layout, Parking, Transportation

Once on campus, a new faculty member must know options for parking (if they expect to drive to work)—but where are the relevant buildings and how might one get from one destination to another? On some campuses, buildings may be labeled (other campuses may have little labeling), but building names may or may not indicate where departments and offices might be located.

## 000.011: Office and Telephone

A faculty member typically has an office: where might it be, will it be shared, what furnishings are provided, and what telephone number has been assigned? Also, when a new faculty member arrives, how might boxes and equipment be moved in?

## 000.100: email, Computing Equipment, Printing

An incoming faculty member may be issued an email address and computer, but what type (e.g., desktop or laptop), what operating system, what installed software, what process is used for printing, etc.? Also, what accounts and passwords are needed to access school directories/records and one's course materials and professional work?

## 000.101: Copy Machine(s)

Coursework and professional activities often require printing and copying. What capabilities are available, where, what codes are needed, what costs or limitations apply?

### 000.110: Mail Box and Mailing Privileges/Options

Where is the faculty member's mailbox? Does it have a key or combination? What processes are needed to mail a letter, form, or conference submission?

### 000.111: Supply Options

Class preparation, office hours, scholarship, and service require office supplies, such as paper, pens, markers, chalk, tape, scissors, rulers, file folders, etc. Some schools allow faculty to charge materials to department or faculty accounts at campus shops, bookstores or local merchants; or faculty may receive employee discounts. Whatever the process, a new faculty member should learn procedures and needed account numbers.

## 001: School/Departmental Policies and Practices

Teaching does not happen in a vacuum (too hard to breathe). Rather schools create policies and practices, sometimes gathered in a Faculty Handbook or Visiting Faculty Handbook to address matters, such as administrative and academic structures, college committees, voting rights and responsibilities, appointments, promotion, tenure, salary reviews, dismissal for cause, leaves (e.g., parental, sabbatical, leaves of absence, jury duty, military duty), academic freedom, financial/personal emergencies, attendance, and outside employment.

In addition, practices may include such areas as the posting of office hours, scheduling of help sessions, notice required for tests or other exercises, procedures for suspected academic dishonesty, student absences due to athletics or other circumstances, professional travel, faculty attendance at conferences, etc.

Many policies and practices (e.g., scheduling and proctoring of tests) may have direct impact on a new faculty member's initial course planning, so need to be identified early—perhaps before the start of an academic term.

## 010: Support and Resources

Faculty may or may not be able to obtain help for various tasks. Do individuals or offices provide clerical help, how does one obtain reimbursement for supplies or professional travel, what library help is available (e.g., online indexes, interlibrary loan, assistance with bibliographic searching)?

Beyond day-to-day tasks, new faculty should know whom to contact for building problems (e.g., leaks, fire, electrical failures, burned out lights), maintenance of hardware and/or software, repair of buildings or furniture, accumulation of trash, etc. Such matters may seem routine, until something goes wrong. (I know of one classroom within a campus I will not name, in which

17 of the 18 dimmable light bulbs had burned out—apparently no one took responsibility for reporting and fixing the trouble!)

## 011: Options for Software and Lab Environments

Teaching faculty require hardware and software for their offices, classrooms, teaching labs, and [often] research labs. Efficiency suggests a common computing environment for all of these settings, but schools and departments vary dramatically. For example, at one school, departmental labs contain a complete suite of compilers and tools for C, Java, and other languages, whereas college-wide labs have limited functionality (e.g., no debuggers or integrated development environments). If a faculty member moves from one setting to another for teaching, surprises can arise when expected utilities and compilers are absent. Further, if additional software is needed, responsibilities and practices should clarify who can do what and when. (If the faculty member must do installation or maintenance, what permissions are needed, and how are those obtained?)

Similarly, schools differ regarding their reliance on student laptops for computing power within a classroom or lab setting. If student machines are to be used, what understandings exist for the creation and maintenance of needed software, and what practices are in place for correcting difficulties—especially within a teaching lab setting?

## 100: Course Content and Planning

An incoming faculty member often has been hired with the expectation of teaching one or more specific courses. If the course is an upper level elective, course content and format may be reasonably flexible. However, if the course is a prerequisite for later courses, if it builds on previous courses, if it satisfies specific requirements, or if it addresses stated departmental goals and objectives, then the course must fit within a specific context. Overall, an instructor should be told the context of each course he/she will teach; typically dialog and explicit interaction with other faculty is essential.

Once a course's goals and objectives are identified, an instructor should prepare both a syllabus and day-by-day schedule. Today, schools often dictate specific elements of syllabi, including matters of assessment, academic honesty, and accommodations for disabilities. [3] identifies additional elements for discussion with incoming faculty.

Similarly, in developing a day-by-day schedule, an incoming teacher needs a sense of the pace, breadth of coverage, rigor, and student engagement with the material. See [4] and [5] for additional discussion of planning and scheduling.

## 101: Classroom Options and Expectations

During numerous school visits, the author has observed that different schools have remarkably different traditions and expectations regarding teaching. For example, although schools often celebrate the concept of academic freedom, courses may be limited to specific class meeting times and course durations. Commonly for computing, a course might meet 1 hour for 3 times a week, perhaps with a 3 hour lab (6 contact hours total). However, the school might not allow a course to meet 3 times a week for 2 hours at each meeting (still 6 contact hours total). In some settings, a course may be split among faculty-led lecture to an entire class, small recitation sections with teaching assistants, and labs. Incoming teachers may have expectations regarding such timing options, but the customs at one school may not match those at other schools. Since initial faculty assumptions may be incorrect, early notice of possibilities is essential for new teachers (e.g., before the semester course schedule is finalized, if at all possible).

Similarly, teaching environments vary substantially:

- Do classrooms have sliding whiteboards, blackboards, side boards, etc.?
- Do projectors connect to a built-in computer, laptop, DVD players, other equipment? What operating system(s) and software are present?
- Are teaching labs designed for individual or collaborative work? Are projection equipment and/or whiteboards/blackboards available?
- What motivates students (e.g., grades, challenge of problem solving, intellectual curiosity, career prospects, etc.)?
- What frameworks might encourage active student engagement?
- To what extent do the school and the students expect lecturing or student engagement?
- What types of active learning techniques seem common?

Such matters can have dramatic impact on the effectiveness of classes, so incoming faculty should receive briefings as they plan courses, activities, exercises, and tests.

In addition, with such variations among student expectations and learning environments, a department chair might encourage new faculty to invite an observer to attend class from time to time (including early visits).

## 110: Student Assessment and Faculty Evaluation

Assessment has become a vital part of all aspects of school activity. In assessing students, incoming faculty should know some basics:

- What are common forms of assessment in the department's classes (e.g., tests, programs, exercises, projects, written assignments, presentations, portfolios, etc.)?
- What grading standards are commonly used, and is there an expectation regarding final grade distribution?
- What mechanisms are available to detect and report suspected cases of dishonesty?
- To what extent are graders, teaching assistants, or others available to help in grading?
- Does an instructor report only final grades, or are mid-semester grades expected for some or all students? Are other interim reports encouraged or required?
- Should students having difficulty be reported? How, and to whom?

Similarly, although incoming faculty may not need to know all of the details, they should understand the elements of faculty evaluation for contract renewal, promotion, tenure, and salary reviews. Many schools utilize formal end-of-course evaluations, but other instruments also may be involved (e.g., class visits or student interviews). And, once data are collected, faculty should know how that data might be used to evaluate faculty?

## 111: Student Support Services

In recent years, schools are paying considerable attention to the mental and physical health of their students. Also, the Americans with Disability Act and other legislation mandate that schools and faculty find ways to accommodate students with a range of circumstances. In addition, faculty must learn how to respond to issues of assault and harassment. Although many incoming faculty are not trained to handle the vast range of possible student conditions, they should be apprised of what resources are available.

- What resources are available to address mental and physical wellbeing?
- What requirements and procedures are in place to respond to student reports of discrimination, harassment, and assault (e.g., Title IX reporting)?
- What advising and tutoring services are available?
- What offices help address accommodations for students with disabilities?

Incoming faculty may not know all of the answers when talking with students, but faculty should know what staff, offices, and resources to contact to obtain help.

## Conclusions

This paper identifies areas of immediate concern for new faculty. Different schools may address such matters in orientation sessions, conversations with formal or informal mentors, interactions with department chairs, etc. Unfortunately, at various schools, some of these details may be missed, resulting in stress, disrupted teaching and learning, failed labs, etc. Overall, whatever the mechanism, the early success of incoming faculty may depend upon the organized attention to these basic elements.

## References

[1] National Academies of Science, Engineering, and Medicine. *Assessing and Responding to the Growth of Computer Science Undergraduate Enrollments: Prepublication Report*, November 2017. `https://www.nap.edu/catalog/24926/assessing-and-responding-to-the-growth-of-computer-science-undergraduate-enrollments`.

[2] Eric Roberts. *Resources for the CS Capacity Crisis*, November 2017. `http://cs.stanford.edu/people/eroberts/ResourcesForTheCSCapacityCrisis/`.

[3] Henry M. Walker. What should be in a syllabus. *SIGCSE Bulletin*, 37(4):19–21, December 2005.

[4] Henry M. Walker. Course planning: the day-to-day schedule. *ACM Inroads*, 3(3):22–24, September 2012.

[5] Henry M. Walker. Planning and organizing a course for the first time. *ACM Inroads*, 7(4):12–17, December 2016.

# Unreal Engine 4 for Computer Scientists*

## Conference Tutorial

*Paul Gestwicki*
*Computer Science Department*
*Ball State University*
*Muncie, IN 47306*
*pvgestwicki@bsu.edu*

This tutorial will provide an introduction to Unreal Engine 4 (UE4), focusing on properties that would be of interest to Computer Science students and faculty. UE4 is a popular game engine, the fourth generation of an engine technology whose development started in the mid 1990s. As a game engine, it provides common functionality that is used in entertainment software, including a 3D graphics rendering system, real-time audio, input management, networking infrastructure, and artificial intelligence support. It supports development for numerous platforms, including desktop and mobile operating systems, game consoles, and virtual reality devices. The engine is also becoming more popular for use in film as well as architecture.

Getting started with UE4 can be a challenge. As in many powerful, professional-grade tools, it is easy for novices to get lost in the multitude of possibilities. The first part of this tutorial, then, will provide a introduction to commonly-used features for beginners. Tutorial attendees will need no previous experience with game design or game development, although general understanding of programming principles will be expected.

The focus of the tutorial will be on interesting properties of the engine that manifest core principles of Computer Science. I believe that this will provide a valuable complement to the many resources available that present UE4 from the perspective of game design or game development. For example, the Blueprint visual scripting language of UE4 may at first blush resemble the blocks-based programming of systems such as Scratch; however, it also provides powerful syntactic structures that are not found in text-based programming languages. Seeing the Blueprint and C++ implementations side-by-side provides insight not only into the implications of language design but also of the nature of design choices a developer can make. The system of object-orientation in UE4

*Copyright is held by the author/owner.

draws upon its C++ foundations, providing class-based implementation inheritance, but the engine also makes heavy use of both the Component and Flyweight patterns [3] through its component system and class default objects, respectively.

I hope that this Computer Science perspective on UE4 will inspire attendees in several ways. Student attendees will learn how these professional-grade tools are available to them and how their knowledge of Computer Science empowers them to make good use of them. Faculty who have never explored game development will gain an understanding of how contemporary tools support the endeavor, and they may appreciate new examples for use in their own curricula. Those faculty who teach game development with other technologies should be able to see how to get started quickly with UE4, should they wish to try it themselves or mentor students in its use.

To deliver this tutorial, I will need a room in which I can project my laptop screen and potentially slides and other resources. If the conference facility has a lab where UE4 can already be installed and configured, that would facilitate participation in the tutorial. However, I expect this not to be the case, and that tutorial attendees would be expected to have installed the latest stable release of UE4 onto their laptops. I expect to have handouts available for tutorial attendees to point them toward my favorite resources for continued learning.

UE4 has a licensing model that is friendly for academic use. The engine and all of its source code are free to use for non-commercial products. Those who wish to sell products developed with UE4 pay a 5% royalty on gross revenue after the first $3000 per calendar quarter [1]. This means that most academic projects can use UE4 technology with no cost. The source code is available to read and modify under a liberal license as well, although there are restrictions against incorporating code with "copyleft" licenses (such as the GNU General Public License [2]) into the UE4 source.

## References

[1] Epic Games. Unreal Engine End User License Agreement, n.d. Retrieved March 12, 2019, from `https://www.unrealengine.com/en-US/eula`.

[2] Free Software Foundation. GNU General Public License, Version 3, 2007. Retrieved March 12, 2019, from `https://www.gnu.org/licenses/gpl-3.0.en.html`.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

# Concurrent Programming with the Actor Model in Elixir[*]

**Conference Tutorial**

*Saverio Perugini*[1] *and David J. Wright*[2]
[1]*Department of Computer Science*
[2]*Learning Teaching Center*
*University of Dayton*
*Dayton, Ohio 45469*
`{saverio,dwight1}@udayton.edu`

*Project Site:* `http://sites.udayton.edu/operatingsystems`

## 1 Tutorial Summary

This tutorial fosters and facilitates discussion and innovation around the topic of *teaching concurrency and synchronization* in an undergraduate *operating systems* (OS) course [2]. In particular, we lead participants through a variety of active-learning exercises for teaching a 'concurrent programming and synchronization' module of an undergraduate OS course using the Actor model of concurrency—a concurrent programming model that is increasing in popularity, especially in the Elixir programming language. The model naturally focus the programmer's locus of attention on the interaction between concurrent entities to collaboratively solve a problem—the primary learning outcome—rather than the level-low details of language syntax and the minutia of which individual data to protect/lock and how to protect/lock it. In this tutorial, classical (e.g., sleeping barber) and modern (e.g., chat server) problems of synchronization are demonstrated through the lens of the Actor model using in-class laboratory plans that attendees can adopt. Participants are exposed to concurrent programming in Elixir[1] through asynchronous communication via message

[*]Copyright is held by the author/owner.
[1]`http://elixir-lang.org/`

passing and mailboxes. Participants should have working knowledge of concurrency/synchronization in a language such as C or Java and are encouraged to bring laptop computers, especially to follow the labs demonstrated.

## 2 Broadening Aspects

This tutorial is part of a three-year NSF-funded IUSE (Improving Undergraduate STEM Education) project titled "Engaged Student Learning: Reconceptualizing and Evaluating a Core Computer Science Course for Active Learning and STEM Student Success" (2017–20) whose goal is to foster innovation, in both content and delivery, in teaching OS through the development of a contemporary model for an OS course that aims to resolve significant issues of misalignment between existing OS courses and employee professional skills and knowledge requirements.

While an OS course is a nexus in a CS program (connecting the introductory programming sequence to upper-level electives), the typical pedagogical approach to it has become dated and stale—most of the recent focus has been on improving the introductory programming sequence primarily for retention—because it has not been responsive to the transformed landscape of modern computing platforms (e.g., from desktop to mobile; from single- to multi-core architectures), the job market, and the concomitant progress in active, student-centric learning (e.g., from a traditional, content-centric, purely lecture-based course to an active, learner-centric, hybrid lecture/lab-based format). Addressing this issue is the rationale for our NSF-funded project. Thus, a broader objective of this tutorial is to promote and facilitate use of our re-conceptualized course model for OS content and pedagogy. Specifically, we aim to help faculty at other institutions adopt this model in a systematic and simplified way.

The course model involves three progressive modules: 1) mobile OSs and Internet of Things, 2) concurrent programming and synchronization, and 2) cloud computing and big data processing. This tutorial focuses on the 'concurrent programming and synchronization' module of the model, which is the most easily adoptable of the three modules, because it is not dependent on the presence of mobile devices or external resources (e.g., Amazon Web Services). However, participants can also expect an introduction to our re-conceptualized course model; a demonstration of a set of re-usable, in-class laboratory plans; discussion of benefits and tradeoffs of employing the model; and an invitation to participate more actively in the project beyond the tutorial (see below).

### 2.1 Laboratory Manual

In a 2019 SIGCSE birds-of-a-feather discussion that we lead [2], faculty teaching OS found the ability to plug-and-play with the active-learning, laboratory plans attractive and see significant value in making use of them in their courses and teaching activities, especially since developing real-world lab and project plans requires substantial effort and time. Tutorial attendees will be granted access to our Laboratory Manual, which contains these active-learning, exercises, each categorized into one of the three topic modules. Moreover, we intend to provide stipends from our grant to attendees who both adopt the model, or parts thereof, in Spring 2020 and collect pre- and post-evaluative data for a semester.

## 2.2 Community of Practice

We are also establishing an OS teaching community of practice to share both materials and experiences in the teaching of OS. To support the community in sharing expertise and perspective, we have developed an open-access, Git repository of items related to teaching OS The items collected in the repository are shared and accessed through our GitHub Pages portal site: `https://saverioperugini.github.io/Teaching-Operating-Systems-Community-of-Practice/`). The community includes members who attended our 2018 CCSC:MW tutorial [1] and 2019 SIGCSE birds-of-a-feather [2]. We plan to invite 2019 CCSC tutorial attendees to share both their perspective and materials to this repository.

## 2.3 Advisory Group

We are also establishing an *advisory group* of computer science faculty members for this project for an external perspective on the model and its adoption. Another goal of this tutorial is to expand the participation of regional faculty in the advisory group. Members of this group serve as advisors and are asked to provide an external perspective on both the module content and the laboratory plans.

## 3 Schedule of Activities

- Introduce attendees to the course model, the content modules, and the active-learning exercises in the Laboratory Manual.
- Work through lab plans of the Actor model of concurrency in Elixir.
- Share our experience in teaching the OS course using this model.
- How to adopt the course model and use the active-learning labs in the Laboratory Manual.

- Invite participation in our community of OS educators.
- Discussion: Questions and Answers.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Numbers 1712406 and 1712404. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

[1] S. Perugini and D.J. Wright. Developing a contemporary operating systems course. *Journal of Computing Sciences in Colleges*, 34(1):155–158, 2018. Conference Tutorial.

[2] S. Perugini and D.J. Wright. Developing a contemporary and innovative operating systems course. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*, page 1248, New York, NY, 2019. ACM Press. Conference Birds-of-a-Feather; DOI: `http://doi.acm.org/10.1145/3287324.3293734`.

# Google Cloud Workshop - Serverless and Databases*

## Conference Workshop

*Ryan Matsumoto*
*Google for Education*

Learn more about cloud computing on Google Cloud and how you can use it in the classroom. In this interactive workshop, you'll start by learning how to deploy applications using Compute Engine, App Engine, and Cloud Functions. Next, you'll learn how to store and retrieve data using Google Cloud's SQL, NoSQL, and Big Data databases. Finally, we'll cover Google Cloud's education grants program and online learning resources.

*Copyright is held by the author/owner.

# Python, DevNet and the Cloud[*]

## Conference Workshop

*Matthew Cloud*
*Ivy Tech Community College of Indiana*
`mcloud3@ivytech.edu`

Learn how to get started with Python and the CISCO DevNet workshops for your classroom and why they are important. Understand how to work with AWS Cloud 9, DevNet, Python IDLE design environments and how they play into the future of networking and cybersecurity. Develop demo Python programs on API integration with chatbots and see how it can work with Big Data in real time.

[*]Copyright is held by the author/owner.

# Computer-Based Proofs and Disproofs*

**Work-in-Progress**

*Ramachandra B. Abhyankar*
*Department of Mathematics and Computer Science*
*Indiana State University*
*Terre Haute, Indiana 47809*
`R.B.Abhyankar@indstate.edu`

## Computer-Based Proofs and Disproofs

I am exploring the teaching of Computer Based Proofs and Disproofs to Computer Science students, using tools, such as Proof Checkers, SAT and SMT solvers, Model Builders and Theorem Provers. Often proofs are presented in an informal manner, and computer-based proofs can increase the confidence in the proofs.

Smullyan popularized Propositional, First-Order and Combinatory Logic, topics of importance in Computer Science, through his books [1, 2]. In his books, Smullyan offers "informal" proofs of his results. The following problems are considered , for which computer-based proofs and disproofs are given, using Proof Checkers, Model Builders, SAT and SMT Solvers, and Theorem Provers. Students are able to carry out such computer-based proofs.

## Propositional and First-Order Logic

On an Island of Knights and Knaves every inhabitant is either a knight or a knave. A knight always tells the truth and a knave always lies. There is a cluster of such islands.

1. On one such island, all inhabitants said the same thing: "All of us here are the same type."
2. On the next island, all inhabitants said the same thing: "Some of us are knights and some of us are knaves. What can be deduced about the inhabitants of the island?"

*Copyright is held by the author/owner.

3. On the third island, an inhabitant says, “I am a knight if and only if there is gold on the island.” Show that it cannot be deduced that the inhabitant is a knight.

## Combinatory Logic

By an applicative system is meant a system C that assigns to each element x and each element y, an element denoted (xy). The elements of C are called combinators. C contains more than one element.
An element y is called a fixed point of x if xy = y.
For every element x and every element y, there exists an element z such that for all elements u,
zu = x(yu)
This is called the Composition Condition.
An element M is called a Mocker if Mx = xx for all x.
Show that if the Composition Condition holds, and if a Mocker exists, then every element has a fixed point.

## References

[1] Raymond Smullyan. *Logical labyrinths.* AK Peters/CRC Press, 2008.

[2] Raymond Smullyan. *A Beginner’s Further Guide to Mathematical Logic.* World Scientific, 2017.

# Mapping between the Computer Science Body of Knowledge and Fundamentals of Game Design[*]

## Work-in-Progress

*Paul Gestwicki*
*Computer Science Department*
*Ball State University*
*Muncie, Indiana 47306*
pvgestwicki@bsu.edu

I have been exploring the relationship between Computer Science and Game Design. Recently, I have been intrigued by the intersection of these fields' relative epistemologies: what does a computer scientist know, and what does a game designer know, and how can one inform the other?

Alistair Cockburn [2] proposed that software development is a cooperative game of invention and communication. The outcome of a successful game is a working software system. From this perspective, then, the design of a team's methodology is a problem of game design: creating a set of rules that most effectively move the team into a winning state.

Game design, like Computer Science, is an integrative and creative discipline. There are some core principles of game design, although the field lacks a standardizing body that has codified them: read different designers' textbooks and one finds similarities in philosophy but differences in epistemology. However, all game design educators seem to agree that a well-prepared game designer should be broadly educated. In the first chapter to his celebrated text, "The Art of Game Design," Jesse Schell [3] identifies twenty different domains that a game designer can draw upon. Most of these constitute their own undergraduate majors, including business, creative writing, engineering, and mathematics.

In this presentation, I will share my progress in mapping the Computer Science body of knowledge, as expressed by the ACM/IEEE recommendations [1], to the conventional topics of introductory game design education. One of the

[*]Copyright is held by the author/owner.

goals of this work is to identify what core Computer Science knowledge a learner might develop by studying fundamentals of game design. From here, then, one might explore how that knowledge could be used in the design and evaluation of software development methodologies, following Cockburn's Cooperative Game theory.

It is rare that a computer scientist would work only with and for other computer scientists. Ours is an integrative discipline that always has us working with experts from a variety of business, engineering, and scientific fields. A well-trained computer scientist should be a lifetime learner who is able to learn not just new technologies but also new domains of application.

## References

[1] ACM/IEEE Joint Task Force on Computing Curricula. Computer science 2013. Technical report, ACM Press and IEEE Computer Society Press, 2013.

[2] Alistair Cockburn. *Agile Software Development: The Cooperative Game.* Addison-Wesley, second edition, 2006.

[3] Jesse Schell. *The Art of Game Design: A Book of Lenses.* CRC Press, second edition, 2014.