An AI System for Coaching Novice Programmers

Gilbert Cruz, Jacob Jones, Meagan Morrow, Andres Gonzalez, and Bruce Gooch(✉)

Texas A&M University, College Station, USA
{gilbertcruz, meagmccright, agonzalez95, Gooch}@tamu.edu, jtjonesjt@gmail.com

Abstract. We inhabit a century where every job will be technical. In the 21st century, learning to program a computer is empowerment. Programming instruction teaches procedural and functional thinking, project management and time management, skills that are essential components of an empowered individual. Programming is the power to create, the power to change and the power to influence. Today's students regardless of their field of study or need this fundamental knowledge.

Rapidly giving students meaningful feedback is a fundamental component of an effective educational experience. A common problem in modern education is scalability, as class size increases an instructor's ability to provide meaningful feedback decreases. We report on an online Artificial Intelligence (AI) system capable of providing insightful narrative based coaching to beginning programmers. We document system tests to ensure that: it generates a unique response to every input, makes responses in real time, and is deployable online.

Keywords: Feedback · Coaching · Programming · Education

1 Introduction

Statistics show that number of students pursuing computer science as a major in American colleges and universities has dropped by as much as 60% [1, 2]. These statistics indicate decreasing interest of college freshmen in computer science at a time when industry advertises employment opportunities in computer related fields. Several misconceptions deter college students from pursing computer science as a major, one being that computer scientists are males who lack social skills and whose lives revolve around sitting in front of the computer all day long; such misconceptions deter college students from pursing computer science as a major [3–5]. Industry has voiced its concern about the decreasing supply of qualified employees in computer related fields, deciding to partner with academia to find solutions to address the issue of recruitment and retention of computer science majors. One solution involves invigorating traditional core computer science curriculum with pedagogical strategies that leverage the use of digital media as a means to generate interest in courses perceived to be designed exclusively for techies, geeks and programming gurus.

© Springer International Publishing AG 2017
P. Zaphiris and A. Ioannou (Eds.): LCT 2017, Part II, LNCS 10296, pp. 12–21, 2017.
DOI: 10.1007/978-3-319-58515-4_2

2 Curriculum

Teachers often struggle with creative and meaningful ways to assist students in developing their programming skills and provide ample support for beginning programmers [6]. In addition, first year Computer Science carries the highest cognitive load of any academic disciple [1–6]. Students have to learn the problem the solving skills of Engineers and a new form of language. Compounding this problem is the widespread use of Mathematics pedagogy. Exercises and tests use a "fill in the blank" model. Students receive nonworking computer code or a specification and expected to generate a solution.

We have found more practical methods of teaching introductory computer science by incorporating cognitive linguistics research [13–18]. The most useful result is changing the Exercise format from "fill in the blank" to creative writing. Students receive working code and educational scaffolding based on industry practice. Students extend the starter code to increase functionality or efficiency. Then test their code using data they are responsible for creating (Fig. 1).

Fig. 1. Students must Credit their source not Plagiarize. Students may Transform the work of others, not Imitate. Students may Remix the work of others, not Rip-Off. Banksy by Information of New Orleans used under Creative Commons Attribution 2.0 http://commons.wikimedia.org/wiki/File:Banksy_NOLA_SimpsonsA.jpg Jan. 15 2015.

To support this effort, we develop an infrastructure that accommodates novice programmers. The infrastructure includes:

- Sixty creative writing style programming assignments.
- Laboratory programming assignments that emphasize team based enquiry and learning.
- Built-in software evaluation tools to assist students with developing good programming skills.

2.1 Assignments

For each assignment students will be shown how to write Java code that solves a well defined and interesting problem using a programing topic that is the learning objective for that assignment. Students will be responsible for modifying and extending the code to solve a similar problem that interests them. Examples will be provided.

This is a Studio course all assignment code will be posted in an open forum. Grading will be based on three factors; is an individuals code different from the code provided by the instructor, is the code different from that of all the other students, and the instructor may award bonus points for innovation, novelty, and coolness.

Students are not only encouraged they are required to use, modify and communicate with others about computer code they did not write. Students may include copyright free code in their assignments so long as they provide the source of the code and the date in a comment. Assignments must contain 25% original code. Changing variable names and values does not constitute "original code."

2.2 Team Based Learning

Traditional computer science curriculum emphasizes individual achievement, discouraging students from working together for fear of unethical practices such as cheating. In contrast, industry values teamwork as a necessary practice. To facilitate students' ability to work with others and strengthen their interpersonal skills, we incorporate teamwork as an essential component of computer science curriculum and present a team-based learning model applicable to programming assignments. The team-based learning model promotes collaborative learning among students, dividing them into groups for the purpose of completing a programming assignment that is virtually impossible for one student to do given specific time constraints.

We define the process of team-based programming according to the following steps:

Students review the rubric for each programming assignment.

We equate rubric to the course requirements, providing a framework for assignments, learning objectives and methods for student assessment. A well-defined rubric serves as a guide for students' learning and communicates the teacher's expectations for each programming assignment [5]. Therefore, we post the rubric for each module on the course wiki so that the information is available in a public place.

Teams participate in a top-down design approach to derive a solution for the programming assignment, breaking down the design of the game module into manageable pieces (e.g. background scenery, user interface, etc.). Team members negotiate the portions of the assignment that each student will complete; member assignments are posted on the wiki for review. The act of negotiating engages team members in supportive communicative practices such as sharing information that builds group knowledge, acknowledging peers' ideas, mediating conflict when group members disagree, and asking for assistance.

These social interactions form the basis for building a high performance team that works together to accomplish the common goal of designing a game module.

Individual students write code and complete assessment. Although team-based learning emphasizes collaboration among group members to acquire object-oriented design skills, it is not a substitute for individual students mastering these skills. Students write code for their portion of the team programming assignment. Each student then uploads individual submitted code segments to the course wiki and completes an online quiz. The quiz lists open-ended questions and the answers are available only to the instructor (Fig. 2).

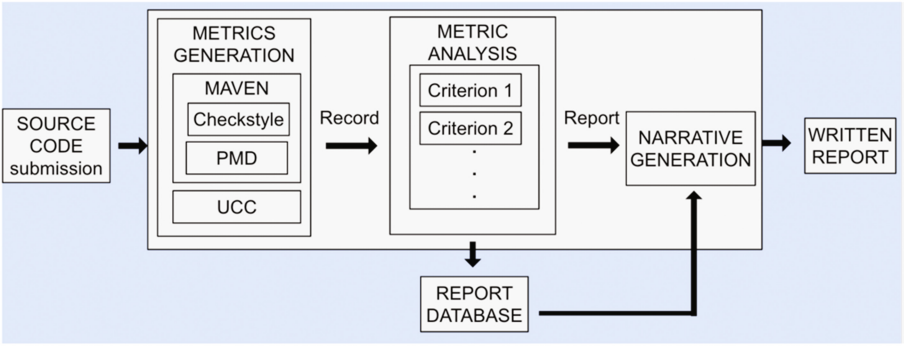

Fig. 2. A Diagram of the system. Code is submitted by students, if the code passes the compile and copy process it moves to metric generation and metric record is generated. The record is stored in a report database that tracks individual student progress and may be used to enhance the compile and copy process. The metric analysis system creates a report that is passed to the narrative generation process. The narrative generation software builds a written report from the code metrics. The report aids in coaching a student based on their personal improvement rather than in comparison to fellow students.

3 Software Evaluation Tools

We used a bottom-up design process shown to be useful in Computational Linguistics [24], Applied Computing [23] and Machine Learning [22]. We first built components. We linked them together to form subsystems. Finally we connected subsystems to develop the top-level system. Our top-level system requires three subsystems; advanced metrics, pattern matching, and narrative generation. The pattern matching

subsystem finds similarities, while the advanced metrics subsystem finds differences. Their output forms the input to the narrative generation subsystem. In order to reach multiple goals and manage our project development, our team implemented a project management task known as scrum. This allowed for the building and testing of the AI system to have a designated timeline. Our team also used our personal past assignments as a sample material. These assignments were used as a testing aid when implementing the system.

Current auto-grading software uses a Compile, Correct, Copied model [19]. The software checks that submissions compile, run, produce correct output, and are not plagiarized. Such software is not enough to provide meaningful feedback [20]. Our Coaching AI first performs the standard Compile, Correct, Copied checks. If the submission passes, we compare with a database of the student's previous work. We generate Feedback in the form of a unique narrative. The story details the effects of the student's modifications (speed, functionality, memory usage, etc.). It also contains, tips on improving the functionality and readability of their code. A story may go on to detail how their work is novel compared with their peers. An evaluation of how their coding ability is improving is always included.

The advanced metrics subsystem classifies code and compares it with two databases. First are a general database and then a personal submissions database. The first database is a collection of old coding assignments. The first database allows us to check for plagiarism. We compare submissions with this database to check for similarity with previously submitted work. We wrote a Java program to make a Compile, Correct, Copied part using CCCC [25], Unified Code Count [26], and RSM [27] and Maven. The second database composed of the Compile, Correct, Copied output for a single individual. It allows us to compute metrics such as increased complexity from two code submissions. We can also find trends, records, deltas, and streaks. We use it to track student improvement over time. Every student's progress is stored for the purpose of creating new reports without making them sound too familiar to previously generated reports. We used the MariaDB system to create and manage the databases. We use a Java program to accesses and inject SQL code into the databases remotely. The Java code is included as metadata in the report. The report uses the Java code to access the tables in individual students information.

The pattern matching subsystem performs lexical analysis to generate an abstract a syntax tree. Abstract syntax trees are analogous to sentence diagrams with parts of speech labeled. Machine learning methods find correlations between the abstract syntax tree of a submission and those stored in a database. These allow us to find improvements like repeated errors, best practice use, or readability.

The narrative subsystem uses natural language generation to tell a story around insights obtained by the advanced metrics and pattern matching subsystems [37]. We used the KPML system from the University of Bremen [28]. The KPML system offers a robust, mature platform for large-scale grammar engineering. Both flexibilities of expression and speed of generation are issues for us. KPML is particularly useful in applications like ours. The finalized report is designed in an easy to read manner that will hold specific detail that is relevant to the students as individuals.

4 Development Plan

Design and development planning culminated in a plan that identifies necessary tasks, procedures for anomaly reporting and resolution, resource requirements, and design review requirements. A software life cycle model and associated activities will be determined, as well as those tasks necessary for each software life cycle activity. We used a test first iterative waterfall software life cycle model.

The student's software creation plan identified the personnel, the facility and equipment resources for each task, and the role that risk (hazard) management will play. A configuration management plan was developed to guide and control development activities and ensure proper communication and documentation. Version control software is necessary to provide active and correct correspondence among all approved versions of the specifications documents, source code, object code, and test suites that comprise the software system. The project used controlled release system software (CVS) to ensure exact correspondence between software versions. The CVS software will also provide accurate identification of, and access to, the currently approved versions. Procedures were be created for reporting and resolve software anomalies found through validation or other activities. Throughout the software development process the International Organization for Standardization (ISO) software creation, documentation, and testing standards were met.

We tested and evaluated the software using the HP software evaluation tools [39]. We ensured that our software; it meets the design and development requirements. It responds to input correctly. It performs in real-time. It is sufficiently usable and can be installed and run online.

Requirements development includes the identification, analysis, and documentation of information about the device and its intended use. Areas of special importance include allocation of system functions to hardware/software, operating conditions, user characteristics, potential hazards, and anticipated tasks. In addition, the requirements will clearly state the intended use of the software. The software requirements specification document will contain a written definition of the software functions. Typical software requirements specify the following: All software system inputs; All software system outputs; All functions that the software system will perform; All performance requirements that the software will meet, (e.g., data throughput, reliability, and timing); The definition of all external and user interfaces, as well as any internal software-to-system interfaces; How users will interact with the system; What constitutes an error and how errors should be handled; Required response times; The intended operating environment for the software, if this is a design constraint (e.g., hardware platform, operating system); All ranges, limits, defaults, and specific values that the software will accept; and All safety related requirements, specifications, features, or functions that will be implemented in software. We will develop the requirements for the software in the first months of working on the project and post them on the project webpage.

Software safety requirements were derived from a technical risk management process that is closely integrated with the system requirements development process. Software requirement specifications will identify clearly the potential hazards that can

result from a software failure in the system as well as any safety requirements to be implemented in software. The consequences of software failure will be evaluated, along with means of mitigating such failures (e.g., hardware mitigation, defensive programming, etc.).

In the design process, the software requirements specification is translated into a logical and physical representation of the software to be implemented. The software design specification is a description of what the software should do and how it should do it. The completed software design specification constrains the programmer/coder to stay within the intent of the agreed upon requirements and design.

The activities that occur during software design have several purposes. Software design evaluations are conducted to determine if the design is complete, correct, consistent, unambiguous, feasible, and maintainable. Software design evaluations will include analyses of control flow, data flow, complexity, timing, sizing, memory allocation, criticality analysis, and other aspects of the design. A traceability analysis will be conducted to verify that the software design implements all of the software requirements. The traceability analysis should also verify that all aspects of the design are traceable to software requirements. An analysis of communication links will be conducted to evaluate the proposed design with respect to hardware, user, and related software requirements. The software risk analysis will be re-examined to determine whether any new hazards have been introduced by the design.

At the end of the software design process, a Formal Design Review was conducted to verify that the design is correct, consistent, complete, accurate, and testable, before moving to the coding phase of the project.

Software may be constructed either by coding (i.e., programming) or by assembling together previously coded software components (e.g., from code libraries, off-the-shelf software, etc.) for use in a new application. Coding is the software activity where the detailed design specification is implemented as source code. Coding is the last stage in decomposition of the software requirements where module specifications are translated into a programming language. The project will be coded in the Java programming language using off the shelf OpenGL libraries to drive the graphics accelerator cards. The system will be run on the Linux operating system.

The project used the TogetherJ development environment. The TogetherJ tool from TogetherSoft is a development environment for building object-based systems using the Java software language. It combines design and coding, letting developers effortlessly switch between the two, and facilitating an iterative style of software development. TogetherJ supports and promotes, a “design, code, test” approach to object-oriented design.

Software testing entails running software products under known conditions with defined inputs and documented outcomes that can be compared to their predefined expectations. Throughout the project we will followed the testing guidelines put forth in NUREG/CR-6293, Verification and Validation Guidelines for High Integrity Systems.

Testing at the user site is an essential part of software validation. Project site testing took place at a user’s site with the actual hardware and software that will form an installed system configuration. User site testing followed a pre-defined written plan with a formal summary of testing and a record of formal acceptance. The testing plan

specified testing throughout the full range of operating conditions and should specify continuation for a sufficient time to allow the system to encounter a wide spectrum of conditions and events in an effort to detect any latent faults that are not apparent during more normal activities. All testing procedures such as test input data, and test results will be documented and retained. In addition to an evaluation of the system's ability to properly perform its intended functions, there was an evaluation of the ability of the users of the system to understand and correctly interface with the workstation (Fig. 3).

Metrics

Lines	20
Comemnts	5
Cyclomatic Complexity (CC)	2
Whitespace	27

Criteria

	Great	Good	Neutral	Bad
Comments/Lines	X < .25	.25 <= X<= .30	.30 < X <= .35	X > .35
CC	X < 3	3 <= X <= 6	6 < X <= 9	X > 9
Whitespace	X < 5	5<= X <= 15	15 < X <= 25	X >25

*These values are for illustrative purposes only.

Example Report: This week you did very well in commenting your code, you have greatly improved your commenting skills in the past 3 weeks! In the past, you've had trouble with the amount of nested loops, but this week there was great improvement. You seem to have a little trouble with whitespace in your code. Additional whitespace is being put in unnecessary places; removing this whitespace will give your code a cleaner look. Overall, this week you have improved a lot more than any other week. Keep up great work!

Fig. 3. Shows some of the possible metrics and criteria that are generated in the statistical analysis programs and an example report based on the illustrative metrics and criteria.

5 Program Evaluation

We performed both formative and summative evaluation of the assignments and team based learning laboratories. Formative evaluation metrics included pre-tests administered to students prior to participation in the project, and baseline attitude surveys administered to student participants. The surveys were repeated at the end of the course and six months after the course.

The evaluation instruments were designed to answer the following questions:

- Has the students' knowledge, of java programming, increased since the pre-test.
- Have their attitudes toward studies and careers changed since learning to program?

Over the past two years we have evaluated six cohorts in all six cases the answers we received are yes and yes. Student knowledge of Java programming has increased and tends to increase after the course. We found that students tend to write small, less than one thousand lines of code, programs to solve homework problems in other classes. We have also found that students who listed Java programming experience on resumes were more likely to receive internships. We have anecdotal evidence that a significant number of students have transferred into the CS major, but at this time we lack a rigorous longitudinal study. The most telling statistic is the fact that over three years we have less that 1% dropout from the course and a less 1% failure rate. This compares to the traditionally taught course with a 30% dropout rate over the same time period.

6 Future Work

To determine the effectiveness of our platform we are currently conducting a multiyear longitudinal study. We will compare students' performance in our course to students who enroll in a traditional object oriented programming class. In phase one groups of students will complete a pre and post assessment that measures their understanding of object-oriented programming concepts and ability. The assessment will help us to recognize factors that contribute to learning outcomes for each group. In phase two we are tracking the student performance in second year of programming courses. We also plan to compare fan-in and fan-out. (i.e. do non-majors who take this class change their major to Computer Science and what proportion of Computer Science majors change after taking this class versus a standard class.)

7 Conclusion

The computer's role in culture has expanded from a calculating machine used by governments to the iPod as a fashion accessory, the Smart Phone as a companion, and the Internet as a medium of self-expression. In the 1950s, the idea of dedicating a computer to entertainment was unthinkable; today revenues from the computer game industry exceed Hollywood. More humans own a computer than own a toothbrush and Apple is the worlds largest company having overtaken Oil, Agriculture and Manufacturing. From if conditionals to for loops, knowing the basics of programming allows one to understand the way the modern world works.

References

1. Snyder, N.: Universities see a sharp drop in computer science majors
2. Vesgo, J.: Interest in CS as a major drops among incoming freshman
3. Guzdial, M., Forte, A.: Design process for a non-majors computing course
4. Guzdial, M., Soloway, E.: Teaching the Nintendo generation to program: preparing a new strategy for teaching introductory programming. Commun. ACM **45**, 17–21 (2002)
5. Margolis, J., Fisher, A.: Unlocking the Clubhouse: Women in Computing. MIT Press, Cambridge (2002)
6. Layman, L., Williams, L., Slaten, K.: Note to self: make assignments meaningful. In: Proceedings of the Thirty-Eighth SIGCSE Technical Symposium on Computer Science Education, pp. 459–463. ACM Press (2006)
7. Tuovinen, J., Sweller, J.: A comparison of cognitive load associated with discovery learning and worked examples. J. Educ. Psychol. **91**(2), 334–341 (1999)
8. Sweller, J.: Cognitive load theory learning difficulty, and instructional design. Learn. Instr. **4** (4), 295–312 (1994)
9. Pea, R.D.: Midian Kurland, D.: On the cognitive effects of learning computer programming. New Ideas Psychol. **2**(2), 137–168 (1984)
10. Jenkins, T.: On the difficulty of learning to program. In: Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences, vol. 4 (2002)

11. Pennington, N.: Stimulus structures and mental representations in expert comprehension of computer programs. Cogn. Psychol. **19**(3), 295–341 (1987)
12. Mayer, R.E.: Different problem-solving competencies established in learning computer programming with and without meaningful models. J. Educ. Psychol. **67**(6), 725 (1975)
13. Soloway, E., Spohrer, J.C.: Studying the Novice Programmer. Psychology Press, Hove (2013)
14. Gomes, A., José Mendes, A.: Learning to program-difficulties and solutions. In: International Conference on Engineering Education–ICEE, vol. 2007 (2007)
15. Du Benedict, B.: Some difficulties of learning to program. J. Educ. Comput. Res. **2**(1), 57–73 (1986)
16. Lahtinen, E., Ala-Mutka, K., Järvinen, H.-M.: A study of the difficulties of novice programmers. ACM SIGCSE Bull. **37**(3), 14–18 (2005). ACM
17. Mayer, R.E.: Teaching and learning
18. Tan, P.-H., Ting, C.-Y., Ling, S.-W.: Learning difficulties in programming courses: undergraduates' perspective and perception. In: International Conference on Computer Technology and Development, ICCTD 2009, vol. 1. IEEE (2009)
19. Ala-Mutka, K.: A survey of automated assessment approaches for programming assignments. Comput. Sci. Educ. **15**(2), 83–102 (2005)
20. Ihantola, P., et al.: Review of recent systems for automatic assessment of programming assignments. In: Proceedings of the 10th Koli Calling International Conference on Computing Education Research. ACM (2010)
21. Morris, D.S.: Automatic grading of student's programming assignments: an interactive process and suite of programs. In: Proceedings of 33rd Annual Frontiers in Education, FIE 2003, vol. 3. IEEE (2003)
22. Califf, E.: M., Mooney, R.J.: Bottom-up relational learning of pattern matching rules for information extraction. J. Mach. Learn. Res. **4**, 177–210 (2003)
23. Chang, B.-M., Choe, K.-M., Giacobazzi, R.: Abstract filters: improving bottom-up execution of logic programs by two-phase abstract interpretation. In: Proceedings of the 1994 ACM Symposium on Applied Computing, pp. 388–393. ACM Press (1994)
24. Erbach, G.: Bottom-up early deduction. In: Proceedings of the 15th Conference on Computational Linguistics, pp. 796–802. Association for Computational Linguistics (1994)
25. http://sourceforge.net/projects/cccc/
26. http://sunset.usc.edu/ucc_wp/
27. http://msquaredtechnologies.com/m2rsm/rsm_demo.php
28. http://www.fb10.unibremen.de/anglistik/langpro/kpml/readme.html
29. The Boyer Commission on Educating Undergraduates: Reinventing undergraduate education: a blueprint for America's research universities (1998)

http://www.springer.com/978-3-319-58514-7

Learning and Collaboration Technologies. Technology in Education
4th International Conference, LCT 2017, Held as Part of HCI International 2017, Vancouver, BC, Canada, July 9-14, 2017, Proceedings, Part II
Zaphiris, P.; Ioannou, A. (Eds.)
2017, XXII, 499 p. 174 illus., Softcover
ISBN: 978-3-319-58514-7