Continuous Feedback - automated real-time feedback for software development oriented project-based learning

Mark E. DeYoung, Capt, USAF

Abstract—A common goal of many Computer Science programs is to enable students to develop conceptual knowledge and procedural skills to analyze, design, test, and build a software oriented solution to an ill-defined problem. Modern continuous software development practices offer techniques, tools, and procedures that could be harnessed in software production oriented project-based learning to enhance student learning. In this paper we review selected topics and resources that could provide a basis for future research on means and methods to provide near real-time automated feedback to students – what we define as continuous feedback for software development oriented project-based learning.

BACKGROUND

Some of the student outcomes we desire from Computer Science majors at USMA are:

- An ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer-based systems in a way that demonstrates comprehension of the trade offs involved in design choices
- An ability to design, implement, and evaluate a computer-based system, process, component, or program to meet desired needs
- An ability to use current techniques, skills, and tools necessary for computing practice
- An ability to apply design and development principles in the construction of software systems of varying complexity.

In essence, we desire our graduates to have the conceptual knowledge and procedural skills that enable them to analyze ill-defined problems then design, test, and build a software oriented solutions.

In this paper we review selected topics and resources that frame current state of practice in automated grading of software artifacts. Additionally, we outline some concepts that could provide a basis for future research on means and methods to provide semi-automated (or potentially fully automated) near real-time feedback to students – what we define as continuous feedback for software development oriented project-based learning.

SOFTWARE ENGINEERING EVOLUTION

Software engineering practices have evolved dramatically over the last two decades. Agile software engineering methodologies fueled the growth of the Internet during the last decade of the twentieth century are commonly utilized in today's efforts. The first decade of the twenty-first century witnessed the refinement of multiple agile development methodologies focused on iterative and incremental changes into tools, techniques, and procedures (TTPs) that radically reduced time to market. The same agile TTPs can be, and often are, integrated into large scale software development efforts[6].

We are already in the midst of the next software engineering evolution. Modern highly automated TTPs such as continuous integration and continuous delivery are critical enablers to continuous deployment workflows like that shown in Figure 2.

Instead of producing software in discrete versioned increments over months or years; today's software development projects strive for accelerated change cycles. Enabled by ubiquitous Internet connectivity a new normal of nearly continuous software deployment has emerged. Practitioners in large and small efforts have adopted at least portions of the continuous methods shown in Figure 1 and Figure 2. Corporations that produce software used every day around the world (like Microsoft, Google and Facebook) are using or shifting towards continuous deployment over discrete deployment. Continuous methodology has assimilated and supplanted agile methodology in many software development efforts.
Modern continuous software development practices offer techniques, tools, and procedures that could be harnessed in software production oriented project-based learning to enhance student learning. Some example systems that include at least partial support for continuous delivery methods include:

- Assembla - http://www.assembla.com
- Gitorious - http://www.gitorious.org
- OpenSUSE Build Service - http://build.opensuse.org
- SourceForge Allura - http://sourceforge.net/projects/allura/

Details of several systems and the specific TTPs they support are briefly described in [10].

AUTOMATED GRADING

Computer Science educators, as practitioners of the discipline, frequently conceive of systems to automate the tedium of grading. Automated grading systems for programming language assignments have been proposed and used for nearly five decades[4]. Automation efforts range from assistive automation, semi-automated grading, to fully automated grading. The techniques proposed generally mirror industry state-of-practice software engineering for their respective era. Unsurprisingly, the last several years attempts have attempted to pick up agile methodology[1]. Some have attempted the use of peer-reviewed test cases [15], cross-utilization and competitive use of student generated test cases [5], or focusing on test-driven development[3][9].

Other efforts are more comprehensive introducing suites of software to support automation. Three arbitrary examples are WebCoM[14] introduced in 2001 is a web-based course manager, PETCHA[13], and FLOP[11] and Jutge.org [12] which include collections of introductory and intermediate programming problems. An overview of automated assessment systems and approaches from the mid 1960s through 2005 is available in [4]. A literature review of automatic assessment tools from 2006 to 2010 is available in [8].

It is the opinion of the author that automated grading techniques are a better fit with course work that is focused on artifact production over process execution. We suspect that there could be some correlation between procedural skills and artifact production oriented learning activities. This could be a limited form of case-based learning where the cases are severely restricted in scope and scale. Auto-graders are likely to enhance learning with software projects where a solution is already specified to some extent – the students are expected to implement a completely specified problem and will not perform analysis. The learning experience is likely to be procedurally focused instead of conceptually focused and students will likely not be exposed to a full engineering design process where they analyze, design, test, and build a solution.

PROJECT-BASED LEARNING AND CONTINUOUS FEEDBACK

Project-based learning (PBL) is intended to expose students to ill-defined problems and allow exposure to conceptual (domain knowledge, theory, principles and thinking strategies) as well as procedural skills. A critical problem with a PBL approach is that much of the learning activity will occur unobserved by knowledgeable persons (e.g. the instructor). If the activity is process based instructors will not often observe performance and will not be able to provide corrective feedback during execution. We can compensate for this by using product/artifact based grading or attempting to inject visibility into the execution of the process. We can also seek pedagogical techniques to enhance observability and tighten the feedback loop. As an example we use in-class exercises for process based activities in several junior and senior level Computer Science classes. This allows instructors to directly observe and provide on the spot correction and guidance.

We believe that the proposed concept of continuous feedback is a natural evolution of existing automated grading efforts. It is the authors opinion that several aspects of agile
development, especially continuous deployment (shown in the innermost loop of figure [1], can be adapted in some project based learning efforts to enhance observability of student performance. We echo the assertions of our predecessors...when a preponderance of the learning activity is the production of working software and the primary project artifacts are software solutions it should be possible to build an environment that provides continuous feedback.

The intent of any such effort would be to increase educators ability to observe students application of software development processes instead of solely relying on post-execution artifacts.

During our literature review we discovered several projects that implement some level of automated grading of software artifacts and a wide range of continuous delivery TTPs. It is this intent of this literature review to provide a limited overview of a few that are potential sources for future research efforts.

ANOTATED REFERENCES


The authors provide a review of manual, rubric focused, grading. The authors then semi-automate the grading process by introducing a tool called ALOHA. The primary intent of ALOHA is to enhance the use of rubrics to increase the consistency and objectivity of written feedback. The authors indicate that the tool impacted the quality of feedback to students positively. They found that the tool did not negatively impact the objectivity of grading and did not cause graders to default to pre-defined feedback phrasing. They caution that any template phrasing must be carefully designed but do not provide any formal guidance or recommended practices.


The author provides a high level overview of a continuous deployment workflow. The workflow presented has several checkpoints for correctness:

• Code Review
• Continuous Integration (into staging environment)
• User Acceptance Testing / Quality Assurance
• Continuous Integration (into release environment)
• Deployment to Production

Continuous integration of source code into the projects staging and release environments relies on automated software builds and unit testing. Continuous delivery is used between workflow/pipeline stages with failure at any stage being sent back to developers.


The authors summarize test driven development (TDD) experiments conducted at several universities. Limitations on test coverage, applicability of unit testing, and scalability of TDD are mentioned but not fully explored. The paper covers many different methodologies: case studies, surveys, experience reports, and controlled experiments. Some of the results highlighted are:

• nine controlled experiments with sample sizes from 4 to 118 in CS1 and CS2 courses

It must be noted that all of the experiments except one used the Java programming language – which accurately reflects industry accepted practice at the time of writing. The paper was written at the apex of agile methodology pre-dating the conception of continuous methodology by a few years.


Douce, et al. provide an overview of several generations of automated assessment.

• First Generation - early assessment systems; 1960s - mid-1970s. In some cases compilers and operating systems were required to support automation. Generally, a grader would execute a grading program on student submissions or the student’s work would be designed to produce very specific outputs for direct observation. The systems described are artifact based.

• Second Generation - tool-oriented systems; 1980s - mid-1990s. Some of the proposed automated assessment systems in this generation start to use test driven and test based techniques. Others use pre-specified criteria to analyze student code for correctness. Most are artifact based while a few attempt to trace (but not check) a students ability to complete stages of development process. A few of the tools are positioned as automated tutoring systems. Often the correctness of a solution is determined by comparing student produced outputs to expected outputs - a standard black box testing technique. The majority of the systems

• Third Generation - web-oriented systems; mid-1990s - 2005.

Web based systems

The authors first and second generation assessment systems coincide with the early eras of software engineering methodology (i.e. centralized control, waterfall). The third generation is associated closely with the emergence of the World-Wide-Web, dot com bubble, and perhaps most importantly the progression of agile software engineering methodology. The authors also discuss the pedagogic merits of several of the systems. The also point out a significant area of concern: automated assessment systems require carefully specified assignments. This is a severely limiting factor for automated grading because it is likely to curtail higher level process based activities and conceptual learning. We consider reading of this paper essential to understanding the evolution of automated grading systems over the last several decades.


The authors notionally propose adopting and adapting test-drive development into student coursework. They somewhat naturally consider implementing an automated grading solution to execute student code a test cases to be highly desirable. The authors piloted an approach to this using the Web-CAT Grader. The distinctive emphasis is on on evaluating students test cases for validity and correctness. The student test cases are executed against a reference implementation to determine if they are correct. Finally, the authors checked student submissions for completeness (code coverage) and style/quality with static analysis tools. The tools used for code coverage and static analysis are both open source projects which are available to students and educators. The authors did not provided any conclusions or experimental evidence that demonstrates the efficacy of their assessment methods.


The report recommends using Agile development methods alongside the Capability Maturity Model Integration CMMI.
They propose that the best practices of both can co-exist and that using both agile methods and the model can improve software production processes.


This paper is derived from thesis work where the author attempted to develop a set of lecture notes to teach domain testing. This paper is not directly relevant to the topic of continuous feedback. It does describe some ways that lesson plans could be analyzed empirically.


The authors developed an automated assessment/automated grading system called FLOP. This paper is particularly useful because they discuss existing systems and explain why the did not choose each. Unfortunately, while the FLOP system is deployed the authors only documented personal observations of the results and have not performed experimental studies. Empirical evidence of the efficacy of the systems impact on student learning would improve the quality of the authors work.


The authors propose peer review as a way to motivate student learning. The authors also attempt to address issues such as:

- fairness
- anonymity
- academic integrity
- increased workload

The proposed method is to use a double-blind approach so students do not know who’s work they are evaluating. While the technique provides some automation it is not likely to be a fundamentally useful enhancement above that provided by standard code review systems that support peer review (i.e. Gerrit - http://code.google.com/p/gerrit/).