Teaching Programming Skills Through Learner-Centered Technical Reviews for Novice Programmers

XIAOHONG (SOPHIE) WANG
Salisbury University

This article discusses the experience of teaching programming skills to novice programmers through learner-centered technical reviews. The article first addresses the challenges in teaching programming skills to novice programmers in an educational setting. Learner-centered technical reviews and their benefits in teaching programming skills are discussed, and the observations and specific strategies used during learner-centered technical reviews are addressed. The last section summarizes the experiments of adopting learner-centered technical reviews in teaching programming skills and presents a few highlights for future work.

Key words

learner-centered, programming courses, programming skills, technical review

SQP References

Teaching Software Quality Assurance in an Undergraduate Software Engineering Program
Vol. 9, issue 3
Claude Y. Laporte, Alain April, and Khaled Bencherif

Teaching Challenges: Testing and Debugging Skills
Vol. 11, issue 1
Fereydoun Kazemian and Trudy Howles

INTRODUCTION

Novice programmers face many challenges in learning the basic skills required to design and implement programs. First, programming requires thinking with abstract concepts, which is difficult for novices. Second, programming includes many different tasks, such as problem solving, algorithm and data structure design, programming language comprehension, testing, and debugging. Weakness in any one of these areas can lead to inefficient programming experiences and buggy programs. Mastering all of these skills at once, however, can be overwhelming for beginners and requires a lot of training and practice.

McCauley et al. (2008) viewed programming as a process of building a plan, in the form of source code, to achieve a certain goal. When the plan breaks down, a bug occurs. The cause of the breakdown, that is, the cause of bugs, can stem from students' preconceptions, misconceptions, and the fragility of their knowledge and understanding of programming itself and of the programming languages being used.

Kazemian and Howles (2008) conducted a survey to understand the programming behaviors of novice programmers and found that among the students they surveyed: 1) only 5 percent of the students always developed a design and/or a plan before they started programming; and 2) only one-third of the students always static checked their work, and even less than one-third of

the students always performed a unit test on their work. The reason for not checking/testing was lack of time. The students spent so much time trying to get their programs to compile and run that they had no time left for testing. Their survey results showed that the students were not aware that some forethought could help them identify problem areas and result in fewer bugs and less rework. The results also indicated the students needed better time management and planning skills, better knowledge of programming languages, and better debugging skills.

Lu and Wang (2006) adopted learner-centered technical reviews for programming courses by tailoring industry technical reviews to their special educational needs. Their results indicate that learner-centered technical reviews are effective in an existing computer science curriculum. In this article, descriptions are given of strategies and experiences in teaching and improving beginning programmer programming skills by conducting learner-centered technical reviews in programming courses. As such, the article includes the following: descriptions of learner-centered technical reviews and their benefits; case studies of applying learner-centered technical reviews to teach programming skills in 200- and 300-level programming courses; results of experiments; and plans for future work. The narrative in this article describes practices employed by the author and other colleagues from two different universities.

LEARNER-CENTERED TECHNICAL REVIEWS FOR TEACHING PROGRAMMING SKILLS

Background

Technical reviews are still the most effective method for discovering defects in software artifacts in industry. When learner-centered technical reviews were designed, the artifacts, review tasks, role plays, expectations, format, and frequency of traditional industry technical reviews were modified and adjusted to suit educational needs. Since the audiences of technical reviews in an educational environment are students who vary in their academic background, maturity level, and experience, the goals and strategies for learner-centered technical reviews at each programming course should be determined by the level of the programming course. One important fact that should be pointed out is that, in real-world technical reviews, people only identify problems—they do not solve them. To serve learning goals for students, however, a modification was made to this practice by incorporating problem-solving processes, such as the defect correction process, into learner-centered reviews. For example, the instructors would like students to see the debugging process with their own eyes. The instructors' experience and other research (Anewalt 2005; Gehringer et al. 2005) suggested that the adjusted traditional technical reviews have a very high educational value. For instance, they can help students improve their soft skills, such as giving constructive criticisms and being receptive and open to suggestions. Second, they can motivate students to do their best in their work, and hence train them to strive for excellence in their future career. Regular review sessions also force students to plan early for their assignments and make steady and incremental progress. This not only improves the quality of their work, but it also makes them appreciate the importance of planning in software development. Overall, learner-centered technical reviews can be an excellent learning process for students.

The following briefly summarizes the strategies (in terms of review artifacts, review tasks, role plays, and expectations) of learner-centered technical reviews. More details can be found in Table 1.

100-level course technical reviews

At this level, reviews are focused on how to set up a project plan and how to initially decompose a given programming problem into manageable tasks. The instructors also ensure that steady progress is made by tracking task completion during each review session. When students start writing code, the instructors begin to conduct source code reviews and demonstrate how to use good programming styles to avoid common errors. Only the advanced students are selected to show their work during review sessions. The instructors believe that, at this early stage of learning to write programs, it is vital to encourage and inspire students. A gentle and moderate approach makes students feel comfortable with the review process. The expectations for most students are to observe and to learn. The instructors play instrumental roles at this stage and act as moderators and major reviewers to drive and control each review session by asking questions, pointing out issues, and providing detailed guidance.

TABLE 1 Summary of Learner-Centered Technical Review Strategies

	100 level	200 level	300 level	400 level
Artifacts	• ideas • plans • source code	• data structure • algorithm • source code	• source code	• requirement document • user interface design • architecture design • database design
Tasks	• review ideas • adjust plans • track progress • detect bugs • review styles	• review data structure • review algorithm • track progress • detect bugs • review styles	• detect bugs • review styles • review efficiency	• confirm requirements • detect design flaws
Producer	volunteers or good students	individuals or every team	every team	every team
Expectation for audience	• learn how to decompose problem • understand good programming styles • observe common mistakes	• learn how to design data structure • learn how to design algorithm • understand what good programming styles are • observe common mistakes • provide feedback	• find defects • provide feedback	• find defects • provide feedback
Instructor's role	• moderator • major reviewer	• moderator • major reviewer	• moderator • reviewer	• reviewer
Format	in class	in class	in class	open to public
Frequency	one hour/week	one hour/week	one hour biweekly	before major milestones

© 2010, ASQ

200-level course technical reviews

The review artifacts for 200-level courses include data structures, algorithms, and source code. Since the concepts of data structure and algorithm are introduced at this level, the instructors want to see how well these concepts are understood and used by students. Progress tracking, error detection, and critiques on programming style are still important tasks, and discussions of these topics are continued. Every student and/or team is required to give a presentation at this stage. By this time, all students should feel comfortable with the review process. Besides observing and learning, students are also expected to be actively engaged in each review session by asking questions, identifying issues, and offering suggestions. Although the instructors still act as moderators and major reviewers, more and more student participation is expected.

300-level course technical reviews

In 300-level courses, the major technical review artifact is source code. At this stage, students should be familiar with some basic approaches and tools for effective debugging. The instructors can focus more on common logic errors, programming styles, and source code efficiency. The instructors' roles as moderators and reviewers should be significantly replaced by active student involvement.

400-level course technical reviews

Students enrolled in 400-level programming courses are likely to work on large software development projects. The instructors try to make their review sessions mirror real-world technical reviews as much as possible. Some review sessions at this stage are open to wider audiences (across different courses and different majors) and focus on higher-level software artifacts such as requirement and database and architecture design documents.

Benefits

The activities involved in learner-centered technical reviews include reviewing of project plans, task division, requirements and design documents, tracking progress, bug detection and correction, data structure and algorithm design, program efficiency, and critiques of programming styles. To engage students more effec-

tively, different activities are given different emphases, depending on the level of a course.

Reviews on project plans, task division, and project progress help students learn and practice planning and think ahead before programming, which are key to the development of high-quality products. One obstacle for most freshmen in computer science programs is figuring out where and how to start when they are given a software project. It is hard for them to relate a real-world problem to a computer program. Besides, they have no clue as to how long it will take them to complete a given programming task due to lack of programming experience. Sometimes, they do not understand the requirements for a project initially. Learner-centered technical reviews can demonstrate to them how to plan early for their programming assignments and make steady and incremental progress.

Reviews on error detection and correction, data structure and algorithm design, and program efficiency allow students to have firsthand experience with the debugging process, clarify their understanding of programming language components, and identify common syntax and logic mistakes. By participating in these activities, students learn basic debugging techniques, improve program comprehension skills, understand how memory is used for storing variables, visualize program execution dynamically, and identify possible discrepancies using debugging tools. They also see the examples and consequences of good and bad programming styles. Using critiques on programming styles also teaches them about bug prevention and why and how to develop good programming styles.

Other benefits of learner-centered technical reviews have also been identified. For example, unlike the traditional project grading process and regular lab sessions, learner-centered technical reviews allow an instructor's feedback and suggestions on one person's work to benefit a whole class. Since learner-centered technical reviews are conducted as part of a programming course, the goals of teaching programming skills and infusing software quality assurance are achieved with no extra credit hours added to an already busy curriculum.

CASE STUDIES

In this section, the author describes the experience with learner-centered technical reviews among 100- to 300-level programming courses. Learner-centered technical reviews at these levels put much emphasis on programming skills such as project planning, time management, and debugging skills.

Improve Planning and Time Management Skills

As shown in Table 1, project plans are part of the technical review artifacts. A typical project plan includes task division and assignments and a tentative schedule for a project. During the first few learner-centered technical review sessions, the following typical student behaviors were observed:

A couple of project teams had clear plans and made steady progress. Some teams had a plan but the plan was not followed properly and the progress was inconsistent. Some teams had no plan and had a slow start. A couple of teams did not come to technical reviews.

When the instructors assigned programming projects, they often allocated a portion of a project grade to planning related activities such as having a tentative schedule and making steady progress. They hoped to use this incentive to motivate students and avoid procrastination. Being motivated, however, was only part of the story. Some students, particularly novice programmers, did not even know what a project plan was or how to come up with one. They were not experienced in work load and time estimation either. During the first few review sessions, the instructors usually went over the project plan for each team and provided feedback and advice on how tasks were divided, how time estimations were calculated, and how these estimations could be improved by using correlation and analog. When students reviewed their schedule, the instructors provided advice on how to adjust the schedule according to newly updated task division and time estimation, and what to do when a deadline approached.

Interesting changes occurred after the first few review sessions. Some students seemed to learn quickly and started to have a plan with better clarity and, most importantly, tried to use the plan to direct and regulate their time and effort. Although they still could not completely follow through the plan, the fact that they realized what needed to be done for the project, and their efforts in trying to complete the tasks before each deadline, became very obvious and indicated they were making progress.

Toward the end of a programming course, the instructors observed the following student behaviors:

Teams with clear plans made steady progress according to their plan and completed the project with good quality. Most teams had a plan, and even though the plan was not followed consistently, steady progress was made. In the end, they completed their projects with some struggle (for example, programming overnight before deadlines and leaving no time for testing).

The aforementioned observations were encouraging. They made the instructors realize that, while allocating a portion of the project grade to planning activities did motivate students, learner-centered technical reviews provided them with many effective learning opportunities to improve their planning and time management skills.

Teach Debugging Skills

Besides project plans, another major group of review artifacts for 100- to 300-level programming courses includes data structures, algorithms, and source code. When the instructors reviewed these artifacts, they identified many problems that were directly or indirectly related to student debugging skills. Whenever these problems were identified, they seized the opportunity during review sessions to introduce debugging techniques. The following behaviors were often observed during learner-centered technical review sessions:

Students entered many lines of source code without any attempt to compile. This led to a lot of frustration when they saw a long list of compilation errors after they finally compiled their program.

This was a very common behavior among beginning programmers. To a certain extent, this behavior reflected their lack of confidence when they faced both a new programming environment and a new programming language. Typing as much source code as they could provided them with some comfort, even though real progress might not be made. During review sessions, the instructors suggested that students use an incremental development approach by typing in a small portion of source code first, and then compiling and running it to make it bug free before adding more lines of code. Successfully going through that process alone gave students a sense of achievement, gradually built their confidence, and eventually helped them become acquainted with the new programming environment and the language. This also prepared them for understanding and using backtracking techniques when they debugged their programs. When students used an incremental development approach, they had a better idea of when and where a program stopped working properly by looking at the changes made recently.

Students did not understand compiler error messages and did not know how to use the helpful features provided by a programming environment.

This was also very common among students in lower-level programming courses. Most compiler error messages use technical nomenclatures with which beginning students are not familiar. The existence of other useful information within a compiler error message, such as the line number of the error, does not get noticed initially. Most integrated development environments have many user-friendly features, but those nice features will not benefit beginners if they don't know how to use them. The instructors took this opportunity to teach students how to interpret a compiler error message and familiarized them with these useful features. The instructors also introduced basic strategies on how to avoid those errors. For instance, missing the closing parenthesis or brace was a very common error, and this error could be avoided by putting down both opening and closing parentheses or braces at the same time before adding anything between the pair.

Many common logic errors were identified. For example, in a conditional statement "if (x==y)," a single equal sign "=" was used instead of "==." Other examples were accessing an array beyond its boundary (caused by misunderstanding of the concepts of array size and array index); misusing multiple "if" statements and "if/else if" statement; and using a variable to accumulate a total without initializing the variable to zero first.

When these errors were identified during learner-centered technical reviews, the instructors started by reinforcing related programming language concepts and then demonstrating how a program behaved differently with and without the existence of those bugs using a debugging tool. This was a great time to teach how to use debugging tools. After seeing the contrasting behaviors a subtle difference in source code could make, students grew to attend more details. The instructors also used debugging tools to show how the values of variables were changed as a result of the execution of a line of source code, and how these changes affected the overall behavior of a program, that is, going into different branches. Those live demonstrations allowed students to gain a deeper understanding of the dynamic behaviors

of each language component and begin to build mental pictures of program executions.

When a program was compiled successfully but did not produce expected results, students did not know what to do. They did not know where and how to look for the errors.

This was a frequently observed behavior in regular lab sessions. It was also the most difficult one to tackle since there could be many different scenarios for a program to misbehave and strategies to be selected depended on specific scenarios. This was a perfect opportunity, however, to teach debugging strategies and techniques so students could learn to reproduce, isolate, track, and fix problems. The instructors introduced techniques relevant to the programming problem in question and demonstrated how to select and apply a specific technique. For example, the first step in debugging was to reproduce the problem, that is, to create a test case that caused the program to fail. The strategies and techniques associated with reproducing a problem were introduced. The next step was to find out what was relevant, that is, to create simplified test cases that contained only the relevant circumstances. The best scenario would be a simplified test case that immediately pinpointed the defect. The strategies and techniques on how to achieve this were introduced. Once the instructors had reproduced and simplified the problem, they tried to help students understand how the failure came to be. The basic techniques for creating and verifying hypotheses, making experiments, and conducting the process in a systematic fashion were discussed.

After the introduction to debugging strategies and the demonstration of using those strategies by the instructors, students were asked to apply what they had learned to similar problems in their own programs. The instructors found that this was a very productive process compared with the one-to-one (one instructor to one student) debugging during regular lab sessions, since the demonstration and feedback by the instructors were seen by all students in the class.

Poor programming styles were observed frequently. For example, there was no comment in source code; duplicate code segments were found within a program; source code was not properly aligned; a huge main program was written instead of breaking it into modules; and meaningless variable names were used often.

Like any other practice, good programming habits developed in an early stage of learning go a long way. With so many skills for novice programmers to master, however, programming styles tend to become the least important ones to be considered. Although a portion of a project grade was usually assigned to programming styles, the instructors found that, even though some students handed in "nicely commented" programs, they added these comments at the last minute either "to make the instructors happy" or "to get the credit." This was understandable, since the novices did not have enough knowledge to know the differences between good and bad styles. So it was important for them to see comparisons of different styles and realize the consequences of programming with different styles. At this stage, it was the instructors' responsibilities to show examples of good and bad code and the significant consequences they led to. The instructors also showed students how common errors could be avoided with good programming habits and encouraged the development of good programming habits. Through those training exercises, students learned to achieve bug prevention by becoming defensive programmers.

DISCUSSIONS AND FUTURE WORK

Learner-centered technical reviews proved to be very effective in achieving the educational objectives, particularly in teaching programming skills to beginning programmers.

In the surveys conducted for learner-centered technical reviews at the end of each course, students strongly agreed that learner-centered technical reviews helped them become more motivated and encouraged them to develop better plans for their projects. Learner-centered technical reviews also helped them learn and improve their programming skills. They thought the quality of their final projects had been improved. Most students felt comfortable sharing their incomplete work, or sometimes, their mistakes with the rest of the class during review sessions.

According to the survey results, source code reviews and debugging process demonstrations were the two most beneficial activities. Students agreed that the time spent on the reviews was worthwhile. Students believed that a few minutes spent on reviews often saved them from getting stuck for hours when they worked on their own. Students strongly supported regular review sessions, such as weekly or bi-weekly, if possible.

To utilize the full potential of this practice, the author and her colleagues plan to work on the following areas.

First, although debugging techniques are introduced during learner-centered technical reviews, this is done on an ad-hoc basis. They believe that it is important for students to have systematic knowledge and understanding of modern debugging techniques. An online tutorial for debugging techniques and tools is being developed. This tutorial will introduce students to comprehensive and in-depth debugging techniques. Students will be required to go through the online tutorial as part of their programming courses.

Second, it is a serious commitment both in time and effort to incorporate technical reviews into programming courses. So far, the effectiveness of this approach to improve student programming skills has been observed through the improvements of student programming behaviors and the quality of their programming projects. Some systematic and quantitative measurements of the effectiveness of learner-centered technical reviews on work quality improvement would be very useful, for example, the percentage of teams whose product plans improve as a result of technical reviews, or grade improvements on project deliverables after reviews. An online tool that can be used to assess programming knowledge and skills is currently being developed. Once it is completed, students can take assessment tests several times: at the beginning, during, and at the end of each programming course. The assessment results should provide better quantitative measurements of the effectiveness of learner-centered technical reviews. Incorporating training material into this online assessment tool and providing appropriate programming skill training practices based on the assessment results is another direction for future work.

Lastly, the author and her colleagues would like to further explore learner-centered technical reviews across different levels of programming courses. They believe students in different level programming courses can benefit significantly from each other. Not only can students in lower-level courses learn from their senior peers technically, they can also be inspired. They can envision where they will be and what they can do later in their undergraduate experience. This can help them become more conscious about what they should do presently to avoid the mistakes made by their senior peers.

Acknowledgments

The author would like to express her gratitude to all colleagues who participated in this practice. Drs. Homer Austin and Harel Barzilai from the department of mathematics and computer science at Salisbury University and the anonymous reviewers are thanked for their constructive comments and suggestions. The State of Maryland Higher Education Commission provided the Welcome Fellowship for this work.

References

Anewalt, K. 2005. Using peer review as a vehicle for communication skill development and active learning. *Journal of Computing Sciences in Colleges* 21, no. 2:148-155.

Gehringer, E. F., D. D. Chin, M. A. Perez-Quinones, and M. A. Ardis. 2005. Panel: Using peer review in teaching computing. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* 37, no.1.

Kazemian, F., and T. Howles, 2008. Teaching challenges: Testing and debugging skills for novice programmers. *Software Quality Professional* 11, no. 1:5-12.

Lu, H., and X. Wang. 2006. Learner-centered technical review in programming courses. In *Proceedings of the International Conference on Software Engineering Research and Practice*, June 23-26, 702-709.

McCauley, R., S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander. 2008. Debugging: A review of the literature from an educational perspective. *Computer Science Education* 18, no. 2:67-92.

Biography

Xiaohong (Sophie) Wang is an associate professor in the department of mathematics and computer science at Salisbury University in Maryland. She has more than 10 years of industrial and academic experience. She received her doctorate degree in computer science from the University of Victoria, Canada. She can be reached at xswang@salisbury.edu.

Join us in San Diego, CA, February 7-10, 2011, for the

International Conference on Software Quality (ICSQ 2011)

http://www.asq-icsq.org

Theme: High Reliability and Human Safety Critical Software