This article discusses some issues that many educators in computing disciplines encounter: teaching testing and debugging skills to novice programmers. The first part of the article addresses the issue of presenting debugging as part of a larger context of educating future professionals committed to providing quality results. Discussed are the results of a survey designed to identify student programming and debugging practices, and suggestions to help infuse quality practices into courses and assignments. The second part of the article addresses the introduction of noncredit seminars into the curriculum. It describes a pilot program that will develop short seminars to help ease the time crunch felt in many computer disciplines. The seminars will be offered in a laboratory environment where students may experience a hands-on approach to learning, and the seminars will not be connected to course work or grades, providing a low stress environment for learning outside the classroom.

Key Words
computer science education, debugging, programming, quality

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

Designing a Software Quality Assurance Course: An Effective Framework for Teaching
vol. 5, issue 3
Daniel J. Zrymiak and Abhijit Sen

INTRODUCTION
A recent issue of SQP featured an article by Laporte, April, and Bencherif (2007) that described the challenges of teaching software quality assurance (SQA) topics in an undergraduate software engineering (SE) program. In the article, the authors cited various obstacles, including the cost of materials and tools. Perhaps most important, they wrote that SQA activities are given a low priority in a typical SE curriculum.

The authors of this article are educators in an undergraduate computer science program and have faced similar challenges in infusing software quality practices into their program. As Laporte, April, and Bencherif (2007) cited, the authors agree that tools and materials are often cost prohibitive. The most critical issues they have identified in the computer science curriculum, however, are a lack of time and a low perception of the importance of quality practices.

The issue of time is a constant struggle for several reasons. First, in the fall of 1998, the IEEE Computer Society and the Association of Computing Machinery (ACM) joined forces to update the Computing Curricula 1991 (Roberts and Engel 2001). This effort created 20 task forces, and each was charged with
defining the body of knowledge for computer science undergraduates. The task forces studied both two- and four-year college programs.

The task forces addressed both the technological and cultural changes since the publication of the *Computing Curricula 1991*. As a result, many important topics were added, including the Internet and its applications, networking, graphics and multimedia, embedded systems, data management, human-computer interaction, the object-oriented (OO) paradigm, software safety, security, and cryptography. The task groups also addressed educational issues resulting from the influx of technology, the overall growth of computer applications, economic influences, and the broadening discipline. In short, many topics were added or expanded, but few were removed. Consequently, it is difficult to expand on existing topics or add new topics because there is no time to teach them. Many institutions are working to reduce the time to graduation in order to remain competitive and flexible. Adding more required courses or increasing the number of required credit hours for graduation are not desirable solutions to the problem.

Another factor, similar to what Laporte, April, and Bencherif (2007) identified in an SE program, is that software quality also appears to be a low priority topic in most computer science programs. For example, both the IEEE and the ACM emphasize the fundamental need for testing and debugging skills (Roberts and Engel 2001), but verification and validation, maintenance, and quality issues are given little emphasis and are allocated very few hours in the computer science curriculum. There appears to be a perception that computer science students do not need rigorous, in-depth courses in verification and validation, software quality, and reliability—areas typically considered as “software engineering” and not “computer science.”

This article focuses on one quality-related issue the authors have identified as critical to entry-level students and important to student success in later courses—teaching defect prevention and debugging techniques to novice programmers. The first part of this article discusses the importance of teaching defect prevention as part of a larger context of educating future professionals committed to providing quality results. The emphasis focuses on a quality-centered approach to software development beginning with “Computer Programming 101.” This approach creates an environment where students are allowed to learn within a culture that emphasizes a preventive approach to defects. Included are the results of a student survey designed to identify student programming and debugging practices, and suggestions for educators to help infuse quality practices into courses and assignments.

The second part of this article presents strategies for addressing the problem of time. Computer science programs are already stressed to cover the current material and topics, and the constantly expanding field makes it difficult to add new courses or add material to existing courses. A possible solution may be found in a current proposal to offer laboratory based hands-on noncredit seminars. These seminars would provide additional learning opportunities in a low stress environment not connected to course grades.

**TEACHING DEFECT PREVENTION**

The first step in determining strategies for developing students’ testing and debugging skills was to understand their current practices. A short survey was designed to try to understand how students approach programming assignments. The survey was given to 168 students, of which 93 were in their first year and the remaining 75 were in their second. Each survey question was targeted to a specific behavior or attitude. Some of the most interesting responses are discussed here. Student response options were “always,” “sometimes,” and “never” for each of the questions noted.

Students were asked if they static tested their code. In other words, did the students trace the logic flow of their code to identify problems even before they started active testing? The survey determined that only 39 percent of the students always checked their work, 44 percent sometimes static tested work products, and 16 percent never performed any static testing at all.

Related to this question is the issue of resource utilization. Developers’ time is the most expensive resource on most development projects. By static testing, obvious defects can be found and removed before the involvement of other resources. Although it may not be reasonable to static test everything, novice students are strongly encouraged to static test loops and conditionals, which are common sources of semantic errors. Students must learn in a culture that focuses on work integrity so they take pride in high-quality work and learn to respect the time of others in the work group.
A second question asked if students unit tested their work products. Some departments may be guilty, in a sense, of discouraging unit testing because of automatic grading programs. The grading tool used in the authors’ computer science department automatically runs instructor-provided test scenarios and provides selective feedback to students. The suspicion is that this “testing” is the only form of testing many students routinely execute on their code. Although the automated testing program certainly has been helpful in redirecting students who may have had gross calculation mistakes or omitted critical functionality, it has also created a culture where students rely on the grading tool to perform ad hoc unit testing and never think outside the box to create their own test scenarios.

The survey found that only 23 percent of the students always unit tested their code. Most students (70 percent) admitted to only sometimes testing; 8 percent did not unit test. Not testing code implies a “cross your fingers” mentality—write it and hope it works. Many students do not understand the importance of rigorous unit testing until their work becomes a part of a collaborative product. This is why team projects are an essential part of each student’s personal development process. The integrity of one’s work takes on a greater importance when it is a part of a larger project—and when it has the potential to be the weak link in the chain if not done correctly.

Assignments were modified so that students were frequently required to submit a working unit test program. This is especially easy with OO languages. When students implement a new class, they know they must also implement (and submit as a required deliverable) a thorough unit test.

**CODE DESIGN ISSUES**

In most courses, even introductory courses, students are exposed to software design techniques that focus on how a system will be built. But how many students are taught to plan an approach for selecting the language constructs they will use to write the code itself? Here are two examples:

1. A student selects an array data structure to store input data and then spends hours trying to figure out how to expand the array at run-time to accommodate the unknown volume of data.

2. A student uses a data structure to store account transactions identified by an account number, then realizes the structure won’t allow the storage of duplicate transactions with the same account number.

These are some actual code design flaws requiring students to restart projects because they failed to think about the design of the code in terms of the language constructs available.

As with formal system component designs, many students are overly anxious to get started and fail to see the benefits of thinking through a design. Some forethought results in less rework and helps identify problem areas and rule out inappropriate data structures before coding begins. This practice is perhaps most helpful for novice programmers. The survey found that only 5 percent of students always developed a design, 60 percent sometimes did, and 35 percent never developed a design or plan for how they would implement the code. Assignments have been restructured so that students pair together to discuss implementation strategies. They are encouraged to ask inventive “what if” questions to ensure data structures and coding strategies are sound.

**RETHINKING ASSIGNMENTS**

When students are not required to detect, trace, and correct defects, they miss an important skill development opportunity and fail to appreciate the amount of effort, time, and cost involved in finding, fixing, and documenting defects. Many students don’t understand the implications when defects are not found. They often view deliverables as a “throw away” assignment—important only until they get a grade, making it difficult for them to see the long-term impact of their work and the importance of having the entire deliverable correct. Students willing to accept a grade that is good enough, supported by instructors willing to reward mediocre work, have an even more difficult time seeing its importance. Incremental grading, find and fix, and root-cause analysis provide helpful improvement strategies. Each of these practices is designed to help drive the effort from defect detection to defect prevention, and instill a continuously improving quality culture in students’ attitudes.

Incremental grading is one way to force a higher level of accountability and accuracy. Using this model, students must completely and correctly satisfy very specific functionality sets. Students are not allowed to
submit solutions to higher levels of functionality sets until lower levels are passed. For example, students may submit solutions for functionality set 2 only after all of the requirements for functionality set 1 have been met. At each set, students are assigned a grade of acceptable or not acceptable.

Another assignment approach is to require that students who submit incorrect work find and fix all of their defects. This includes making the code corrections and the necessary modifications to test suites. Students may resubmit corrected deliverables for a reduced grade based on the original credit. Students are also required to examine and document how and why their programs failed specific functionality tests, and describe a strategy for improving their test scenarios to prevent similar defects in the future.

**COMPILE ERROR MESSAGES**

Through the survey and one-on-one discussions with students, another interesting conclusion was reached: students don’t adequately test because they don’t have enough time. For anyone who has ever worked on a commercial development project, this probably sounds familiar. Many students admitted that they did not effectively manage their time and simply did not have time to test. Fixing this problem lies in developing time management skills and is not addressed here. Other students, however, indicated that they spent so much time getting the program to compile and launch, they didn’t have enough time for testing. Addressing the amount of time students spend battling compilation errors is a factor the authors are addressing.

Regardless of the language used to teach programming skills, students must learn enough about the compiler to understand what the compiler is trying to tell them through error messages or warnings. Developing the ability to understand compiler messages is difficult for many students, and this appears to be especially true when learning Java. Many college programs use Java, and Java is the current language used by the College Board’s Advanced Placement course for high school students. Java is a powerful and versatile language, but it is also very large and complex (Reges 2006; Roumani 2006). Previously, a typical textbook used to teach the first course in computer science, Computer Science 1 (CS1) using a procedural programming language (such as Fortran) averaged about 380 pages. In fact, the Kernighan and Ritchie (1978) classic, *The C Programming Language*, described the entire C language in just more than 200 pages. Today, a typical CS1 textbook for a Java-based course averages about 866 pages (Beaubouef and Mason 2005). Roberts (2004) dramatized the complexity of Java—there are more than 50,000 public methods, code segments that may be called from within programs, released with the language. Clearly, novice programmers could use some help in understanding error messages and identifying common oversights and problems.

A simple Internet search revealed many sites dedicated to identifying typical syntax and semantic errors encountered by programmers; it was not surprising that students struggle with many of the same errors. The authors are in the process of combining the content on several sites, journal articles, and their own department data to produce a Web page of common errors and fixes. The list of the initial “top 12” most common problems the authors have identified in students’ code includes:

1. Throwing null pointer exceptions
2. Using “=” instead of “==”
3. Using “==” instead of .equals when comparing objects
4. Referencing arrays out of bounds
5. Referencing a method call like an instance variable (missing parentheses). Example: toString instead of toString()
6. Mismatching parentheses or missing semicolons
7. Mistyping identifier names—case sensitivity
8. Misnaming file name (does not match class name)
9. Incorrectly naming methods when overriding methods or implementing interfaces
10. Confusing pass by value with pass by reference
11. Introducing off-by-one errors in loops
12. Using static and nonstatic references incorrectly

The authors' Web site will identify common errors and provide a list of questions to help students determine the problem. For example, “null pointer” will be described and will include a series of questions to help students identify the source of the problem. An example is shown in Figure 1.
Another interesting effort designed to help novice programmers understand compile-time syntax errors and run-time semantic errors is the use of short videos (Simon, Fitzgerald, and McCauley 2007). This project involves producing videos that demonstrate typical programming errors and documenting the strategy for determining the cause. Many of the videos were produced without a script, so the resulting video demonstrates an informal and natural approach—making mistakes and determining the fix. Work on the project is continuing through 2008, and the authors expect to leverage this repository once it is made public.

The authors also are developing a rubric that will be used to evaluate student code and unit tests. The rubric shown in Figure 2 is used to evaluate the classes and unit tests students implement in the first course, CS1. In addition to correctness, the rubric evaluates other features of the code—notice the emphasis on documentation, the use of a version control tool (RCS is used in the introductory courses), and enforcing encapsulation. Students are required to follow a coding standard and are expected to submit a main method to demonstrate the scope and depth of testing for each implemented class.

Initially, the 5-3-0 rating score will be used to help students identify the important constructs of their tests. Students will be given some credit for partially adhering to the expected standards. As they progress, however, the rating scale will be reduced to yes or no, as shown in the last few entries of the rubric in Figure 2. The yes or no rating translates to high stakes—zero or five points on each item—so students will quickly learn to do things correctly.

### Figure 1

Questions to help identify the source of an error

<table>
<thead>
<tr>
<th>Error</th>
<th>Possible causes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null pointer exception. Cause: You have referenced an object that is a null, for example: <code>someObject.toString()</code></td>
<td>1. Declaring the object but forgetting to call its constructor.</td>
</tr>
<tr>
<td></td>
<td>2. The object has gone out of scope.</td>
</tr>
<tr>
<td></td>
<td>3. The object reference was assigned to another name.</td>
</tr>
<tr>
<td></td>
<td>4. Forgetting to return the newly created object if created in a different scope (a method).</td>
</tr>
</tbody>
</table>

### Figure 2

Rubric for simple Java class development

<table>
<thead>
<tr>
<th>Unit test criteria</th>
<th>5</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class header (refer to the coding standards document for details)</td>
<td>Class contains a complete class header</td>
<td>Class contains a class header but some data are missing</td>
<td>Class header is missing</td>
</tr>
<tr>
<td>File header (refer to the coding standards document for details)</td>
<td>Class contains a complete file header</td>
<td>Class contains a file header but some data are missing</td>
<td>File header is missing</td>
</tr>
<tr>
<td>RCS</td>
<td>Class contains RCS check-in information</td>
<td>Class contains partial RCS check-in information</td>
<td>RCS check-in information is missing</td>
</tr>
<tr>
<td>Use of constants</td>
<td>Class uses properly named constants</td>
<td>Some properly named constants are used</td>
<td>Instance variables are missing private declaration</td>
</tr>
<tr>
<td>Instance variables declared private to enforce encapsulation</td>
<td>Class contains private instance variables</td>
<td>Some instance variables are properly declared</td>
<td>Instance variables are missing private declaration</td>
</tr>
<tr>
<td>Correct use of static modifier</td>
<td><code>static</code> is properly used on variables and methods</td>
<td><code>static</code> is properly used on some variables and methods</td>
<td><code>static</code> is incorrectly used on variables or methods</td>
</tr>
<tr>
<td>Constructors</td>
<td>Unit test exercises all available constructors</td>
<td>Unit test exercises most available constructors</td>
<td>Unit test does not adequately exercise all available constructors (only calls default)</td>
</tr>
<tr>
<td>Method calls</td>
<td>Unit test exercises all available method calls</td>
<td>Unit test exercises most available method calls</td>
<td>Unit test misses many methods</td>
</tr>
<tr>
<td>All instance variables declared in the code are correctly used and should not be declared in local scopes</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>More than one object instance generated and tested in unit test</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Generating the javadoc for this class creates complete documentation (tags are not missing in code)</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Main method is included in code to facilitate easy regression testing</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>
right the first time and focus on the details, habits that are important in real-world software development.

**FINDING TIME IN THE COMPUTER SCIENCE CURRICULUM**

A proof-of-concept proposal is currently under development to use noncredit seminars to teach testing and debugging techniques. The seminars will be offered to first-year students so they are prepared to handle the demands of upper-level courses where programming projects become larger and more complex. The seminar will be four hours long and offered over two days. Students will be expected to attend both sessions, and the seminars will be offered at no charge.

The seminars will take place in a laboratory environment where students will experience a hands-on approach to learning. The instructional approach will emphasize peer learning, and student assistants will be used to help during the sessions. The environment will be low stress, since the seminars will not be connected to course work or grades. Offering the seminars outside of class time will also emphasize that some learning needs to continue outside the classroom.

Two pilot seminars will be scheduled in the 2009 academic year, and two groups will be formed: one group will be open to all students on a first-come, first-served basis, and the second will consist of invited students based on instructors’ recommendations. Progress of students completing the seminar will be studied in the following terms, and their programming, testing, and debugging ability will be compared to a control group of students who did not participate in the seminar. The study will determine if a statistically significant difference exists in the programming project grades of students attending the seminar. These data will help determine if the seminars have a positive impact on student performance, and if the seminars should continue with additional support from the department.

Initially, the authors are considering covering the following topics during these seminars:

1) **An introduction to debugging and testing and their importance as part of software development.** Novice programming students seem to confuse testing and debugging. The authors use Myers’ (1979) definitions of testing as the process of executing a program with the intent of finding errors, and debugging as determining the exact nature and location of the suspected error within the program and fixing or repairing the error. This seminar will highlight high-profile software failures and the importance of software quality in a world where more and more applications and consumer products rely on high-quality software.

2) **An introduction to a simple debugging tool.** Because of time constraints, students are not formally trained to use a debugger or a software development environment. In this seminar, students will be provided a hands-on environment in which they can explore various tools or debuggers.

3) **An introduction to a systematic approach to debugging.** Learning a systematic approach will show students a more effective and less time-consuming approach to debugging. In this seminar, students will learn debugging using a deductive approach similar to problem solving. The process starts by: 1) identifying the error (this step can be made more explicit by listing the possible causes of the error under consideration); 2) collecting information about each cause listed in no. 1; 3) forming a hypothesis, that is, making an informed guess about the cause of the error based on the collected information; and 4) testing each hypothesis to find the actual cause or eliminate it from the list of possible causes (Zeller 2006). Eventually, this will lead to the identification of the error and its repair or removal. If as part of this process no single hypothesis remains, new hypotheses will be formed and the process will be continued.

4) **An introduction to the use of assertions.** Assertions are not explicitly taught in the authors’ programming sequence because the courses emphasize learning a language as a tool, and do not explicitly focus on learning the details of a specific language. The built-in assertions in Java, however, can be helpful for novice programmers. Using assertions can force students to think about conditions or states that must exist at certain points in a program, and can be helpful in minimizing
Teaching Challenges: Testing and Debugging Skills for Novice Programmers

frustrated with the required programming projects. Once students became overly frustrated, their confidence diminished, resulting in increased attrition. Specifically, they found that students spent many hours on projects and were increasingly frustrated when solutions did not work. This in part may be attributed to a lack of debugging and testing skills. Students who became frustrated lost confidence in their ability to be successful and easily became “turned off to computer science entirely.”

Programming itself does not define the computer science discipline, and many computer science educators are advocating for a decreased focus on programming. The truth, however, is that programming is an essential tool for computer science students who need programming expertise to succeed in the major. In addition, programming is often a requirement for those computer science graduates advancing to professional jobs in various software development positions.

Clearly, programming is critical to the curriculum and a necessary tool for understanding and applying concepts in higher-level courses (Roberts and Engel 2001, 49). Because programming is typically taught the first year, and the first year results in the highest rate of attrition from computer science programs (Carter 2006; McKinney and Denton 2004; Reges 2006; Ventura and Ramamurthy 2004), the aforementioned strategies and other initiatives should continue to be studied and assessed to determine if they are successful at retaining more students and producing working professionals with stronger testing and debugging skills.

REFERENCES


5) An awareness of root-cause analysis. Using a formal approach, students will examine defects in their programs and trace them back to a specific cause so the appropriate action can be taken to prevent the problem’s reoccurrence. It is difficult to prevent future defects without understanding how and why they are happening, and the same mistakes tend to occur again and again unless there is a focus on the cause. Learning from previous mistakes and deriving a plan for correction and improvement may help instill a “do it right the first time” mentality. Students also will be encouraged to maintain defect journals, where they learn to document defects and their causes and routinely prepare postmortem reports to document what they did well and what they need to improve.

6) Developing skills to better understand compiler error messages. Many students taking a programming course for the first time feel frustrated because it is difficult to understand error messages produced by a compiler. Students will start with a correctly working program and systematically introduce common defects. Activities will provide practice interpreting a stack trace to understand the call chain back to the source of the problem. Errors seeded into the working code will be based on the instructors’ feedback about the most frequently occurring errors that are seen in novice students’ programs. Students will be encouraged to record their experiences in their defect journals.

In addition, the noncredit seminar model has been proposed as a scaffolding to teach other topics for which there is little allocated time in the curriculum. Proposals include business ethics, professionalism, internationalization, human factors, and basic problem-solving skills.

SUMMARY

Computer science enrollments have been declining over the last few years for many reasons that will not be addressed here. One reason cited by Hansen and Eddy (2007), however, was that students easily became frustrated with the required programming projects. Once students became overly frustrated, their confidence diminished, resulting in increased attrition. Specifically, they found that students spent many hours on projects and were increasingly frustrated when solutions did not work. This in part may be attributed to a lack of debugging and testing skills. Students who became frustrated lost confidence in their ability to be successful and easily became “turned off to computer science entirely.”

Programming itself does not define the computer science discipline, and many computer science educators are advocating for a decreased focus on programming. The truth, however, is that programming is an essential tool for computer science students who need programming expertise to succeed in the major. In addition, programming is often a requirement for those computer science graduates advancing to professional jobs in various software development positions.

Clearly, programming is critical to the curriculum and a necessary tool for understanding and applying concepts in higher-level courses (Roberts and Engel 2001, 49). Because programming is typically taught the first year, and the first year results in the highest rate of attrition from computer science programs (Carter 2006; McKinney and Denton 2004; Reges 2006; Ventura and Ramamurthy 2004), the aforementioned strategies and other initiatives should continue to be studied and assessed to determine if they are successful at retaining more students and producing working professionals with stronger testing and debugging skills.


**QMD Presents:**

Software presentations designed specifically for quality managers!

The Quality Management Division has partnered with the Software Division to bring you an unprecedented lineup of software quality presentations. Our new software track features eight expert presenters who make software quality issues come alive in a non-technie way!

- **“Teamspeak”—Communication for Global Software Teams:** Discover the challenges that global software development teams face and how best to avoid them. Presented by Carol Dekkers, one of *Quality Progress’s* new voices of quality for the 21st century.

- **Learning From Software Disasters:** What a novel concept: To learn from previous mistakes! Join presenter Patricia McQuaid, a visionary professor from California Polytechnic in San Luis Obispo for her insights in this important management area.

- **The Capability Maturity Model Integration (CMMI)® Made Practical:** Learn about the U.S. developed model now used worldwide to instill higher quality in software development. Presented by Bob Stoddard, a leading expert from the Software Engineering Institute (SEI).

- **Software Is a Risky Business:** Learn the basic concepts, principles, and processes for ensuring that your next software project adequately addresses the inevitable risk factors. Presented by Linda Westfall, a former Software Division chair.

- **How to Setup IT Dashboards:** Learn a straightforward and structured technique to build information technology (IT) dashboards that directly align with the voice of your customers! Join Kaiser Permanente expert Kandy Senthilmaran and see how easy it can be to build a reliable dashboard.

- **Software Quality Acceptance Sampling Inspection by Attributes:** We’ve all heard about using sampling techniques as a means of increasing the quality of products out the door, but how does it apply to software? Presented by Bill Trest, one of Lockheed Martin’s quality gurus and incoming Software Division chair.

- **Quality Management Think & The Quality Assurance Group:** Find out why Europe and Asia bring international consultant Tim Kasse in again and again to provide advice to their quality management initiatives.

- **Culture Context and Organizational Competence:** Quality managers know that there is a delicate balance between people, processes, and performance that governs projects where quality is important. Join presenter Stephen Lawrence as he explores the relationships specific to software intensive projects.

Visit [www.asq-qm.org](http://www.asq-qm.org) for more information about this conference.

---

**BIOGRAPHIES**

**Fereydoun Kazemian** is an associate professor in the Department of Computer Science at the Rochester Institute of Technology in Rochester, New York. He has more than 22 years of industrial and academic experience. He received his doctorate in computer science from Kansas State University. As a faculty member in the Department of Computer Science, he has developed a number of courses in computer science and software engineering, and has been a consultant and trainer in software development for local industries. He can be reached by e-mail at fsk@cs.rit.edu.

**Trudy Howles** is an associate professor in the Department of Computer Science at the Rochester Institute of Technology in Rochester, New York. She is a Certified Software Quality Engineer and the chair of ASQ’s Rochester Section Software Task Group. Prior to entering the teaching field, Howles worked in industry for more than 25 years. She held positions as a senior software engineer, project manager, and consultant at Bankers Trust, Eastman Kodak Company, the former Digital Equipment Corporation, and General Motors. Howles can be reached by e-mail at tmh@cs.rit.edu.