METHODOLOGIES AND TOOLS FOR THE SOFTWARE QUALITY ASSURANCE COURSE*

Vladimir V. Riabov
Department of Mathematics and Computer Science
Rivier College
Nashua, NH, 03060
603 897-8613
vriabov@rivier.edu

ABSTRACT

Tutorials, labs, projects, and homework assignments were designed to help students explore modern techniques of software quality assurance; debugging C/C++ and Java codes; and developing high-quality computer projects. Different methods (predicate-logic and topological approaches of graph theory; metric theory of algorithms, and object-oriented methodology of rapid prototyping) have been explored by using various tools in analyses of complex computing code. Applications cover software test strategies, code reusability issues, and ways to significantly reduce code errors and maintainability effort. The related course materials provide students with knowledge, instructions and hands-on experience, and motivate them in their research studies.

1 MOTIVATION

Students pursuing careers in software development should be familiar with various methods of software quality assurance (SQA). A year ago we launched a new SQA course that addresses the issue of quality throughout the software development process, including system analysis and design, rapid prototyping, implementation, testing, and delivery. Special attention is given to setting quality standards [1], developing quality measurement techniques, writing test plans, and testing the user interfaces. It becomes a challenge for an instructor to provide students with the state-of-the-art hands-on technology-exploration experience in this field.

Topics covered include the integration of quality activities into the software project life cycle, CASE tools overview, structured testing methodology, complexity and object-oriented metrics, configuration management, capability maturity models, software engineering practices and standards, code re-engineering strategies, and miscellaneous topics.

Students were encouraged to examine various software-development projects. Exploring different testing strategies, they analyzed computer code by using various software-organization metrics including cyclomatic complexity [2], Halstead's [3] and object-oriented metrics [4], and re-designed the code with lower risk level and errors.

This paper contains an overview of the SQA software tools, tutorials, lab manuals, homework assignments, project reports, and research papers of students who took the Software Quality Assurance course. The advantages of using these tools for instruction in online and hybrid courses are also discussed.

2 TOOLS FOR SOFTWARE QUALITY ANALYSIS

2.1 Industrial SQA Tool Used in Software Engineering

One of the most popular SQA tools, the McCabe™ IQ software package, was selected for exploring various study cases and projects in the course on software quality assurance. This industrial tool became available for students under the free-license agreement with the McCabe™ University Program. The tool allows them to explore McCabe's structured testing methodology [2] that became a widely used method in code analyses, unit and integration test planning, and test-coverage estimations. Following specially designed computer-lab assignments and using the McCabe™ IQ tool, students study how to apply the theory of graphs for the complexity code analysis, develop test strategies, and predict possible errors [3, 6] in the code developed by themselves and companies. Unfortunately, the McCabe™ IQ package could be used only on campus; therefore, other Open Source free-license SQA packages were evaluated for use by students at home, specifically when taking courses online or in the hybrid format.

2.2 Open Source Free-License SQA Tools

There are several Open Source free-license SQA tools available for students. The Java Source Metric™ package [7] has been used to analyze Java source code with quality metrics like the Inheritance Depth, Lines of Code, and McCabe Complexity Metric suite. The CCCC™ tool [8] generates a report on various metrics (including the Lines of Code and McCabe's complexity) of C/C++ code. The freeware program SourceMonitor™ [9] has been used for code analysis to identify the relative complexity of code modules. SourceMonitor™ measures metrics for source code written in C++, C, C#, VB.NET, Java, Delphi, Visual Basic (VB6), and HTML. It operates within a standard Windows GUI and exports metrics to XML or CSV (comma-separated-value) files for further processing with other tools. The COCOMO-II™ tool [10] was used by students to estimate the cost, effort, and schedule associated with their software development projects.

3 LAB AND HOMEWORK ASSIGNMENTS

The main goal of labs and homework assignments is to introduce software quality metrics and help students build their individual skills of code analysis, testing, and redesign to improve code quality and enable possible reuse in other projects.

3.1 Introducing the Structural Testing Methodology

The first set of lab and homework assignments deals with implementation of the structured testing methodology offered by McCabe [2]. The approach is based on graph-theoretical complexity-measuring techniques in code studies and control of program complexity. Using the experimental results of Miller [5], McCabe suggests that code modules approach zero defects when the module cyclomatic complexity is less than 10. During lectures, the instructor provided an overview of the graph-based complexity metrics and the results of his systematic metric analyses of software for two industrial networking projects [6]. Following the lab assignments, students explored the McCabe ™ IQ tool and used it to perform metric analyses of several codes by applying cyclomatic complexity (*v*), essential complexity (*ev*), module design complexity, system design complexity, and system integration complexity metrics [2] in order to understand the level of complexity of a code module's decision logic, the code's unstructured constructs, a module's design structure, and the amount of interaction between modules in a program, as well as to estimate the number of unit and integration tests necessary to guard against errors.

3.2 Estimating the Number of Code Errors and Efforts to Fix the Errors

The second group of the lab and homework assignments was designed to introduce students to the comparative analyses of algorithm implementations in different languages (FORTRAN, C, C++, Java and some others). Following Halstead's procedures [3], students identified all operators and operands, their frequencies, and estimated the program length, vocabulary size, volume, difficulty and program levels, the effort and time amounts to implement and understand the program, and the number of delivered bugs (possible errors), *B*. They compared their findings with values calculated by using SQA tools (McCabe™ IQ, Java Source Metric™, CCCC™, SourceMonitor™, and COCOMO-II™), and found that the results are sensitive to the programming language type (procedural or object-oriented).

In particular, students found that efforts to implement and understand the program were higher for procedural languages (FORTRAN and C) than for the object-oriented language (Java), even for simple algorithms, like Euclid's algorithm for calculating the Greatest Common Divisor. They also found that large C/C++ source files [6] contain more actual errors than the number of delivered bugs (*B*) suggested [3].

3.3 Interpreting Object-Oriented Metrics

The third group of the lab and homework assignments was developed to help students identify clusters of object-oriented metrics that would better describe the major

characteristics of object-oriented systems (properties of classes, polymorphism, encapsulation, inheritance, coupling, and cohesion) implemented in computer code written in C++ and Java. Their findings are summarized in Table 1 below.

Metric	Class	Polymor-phism	Encapsu-lation	Inheri-tance	Coup-ling	Cohe-sion
Weighted Methods Per Class	Yes	Yes				
Response For Class	Yes	Yes				
Percentage of Public Data			Yes			
Accesses to Public Data			Yes			
Lack of Cohesion of Methods	Yes		Yes			Yes
Number of Children				Yes		
Number of Parents				Yes		
Depth in Inheritance Tree				Yes		
Coupling Between Objects				Yes	Yes	
Attribute Hiding Factor	Yes		Yes			
Method Hiding Factor	Yes		Yes			
Polymorphism Factor	Yes	Yes				

Table 1. Clusters of Object-Oriented Metrics

Students also identified some specific object-oriented metrics (Weighted Methods per Class, Response from a Class, Lack of Cohesion between Methods, and Coupling between Objects) that are the important factors for making a decision about the code module/class re-usability.

3.4 Comparing Two Releases of the Code

The last, fourth set of the lab and homework assignments was offered to students to identify major factors that forced programmers to change the code [6] in the project redesign efforts. After analysis of the project software (about 300,000 lines of C-code) using the network-protocol approach 271 modules of the old Code Release 1.2 [6] were recommended for redesign. The re-engineering efforts resulted in the deletion of 16 old modules and in the addition of 7 new modules for the new Code Release 1.3. Analyzing the deleted modules, students found that 7 deleted modules were unreliable ($v > 10$) and 6 deleted modules were unmaintainable ($ev > 4$). Also, 19% of the deleted code was both unreliable and unmaintainable. Moreover, all seven new modules are reliable and maintainable.

After redesign, code changes resulted in the reduction of the code cyclomatic complexity by 115 units. 70 old modules (41% of the code) were improved, and only 12 modules (about 7% of the code) deteriorated. This analysis demonstrates a robustness of the structured testing methodology and successful efforts in improving the quality of the Code Releases. Studying the relationship between software defect corrections [6] and cyclomatic complexity, students found a positive correlation between the numbers of possible errors, unreliable functions (with $v > 10$), and error submissions [6] from the

Code Releases (see Fig. 1) in the implementation efforts for six network protocols (BGP, FR, IP, ISIS, OSPF, and RIP).

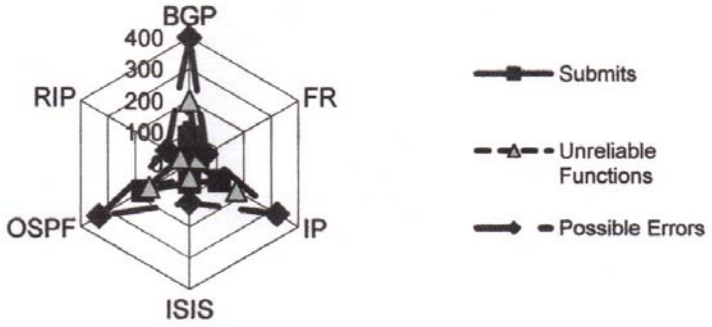

Figure 1: Correlation between the Number of Error Submits, Number of Unreliable Functions ($v > 10$), and the Number of Possible Errors for Six Network Protocols.

4 STUDENTS' PROJECTS AND RESEARCH STUDIES ON SQA

As the main assignment for the SQA course, students were asked to develop their individual or team projects on the quality analysis of moderate-size computer programs written in an object-oriented language (C++ or Java) and compare different releases of the code. The project assignment included the study of the code complexity and quality based on the analysis of cyclomatic complexity metrics, Halstead's metrics, and object-oriented design metrics by using available SQA tools (McCabe™ IQ, Java Source Metric™, CCCC™, SourceMonitor™, and COCOMO-II™). The project reports included the code structure chart, "Battlemap"; McCabe's Complexity Metrics summary; examples of flowgraphs with low, moderate, and high complexity; scatter diagrams with identification of unstructured-unreliable, reliable-structured, unreliable-structured, and reliable-unstructured modules; examples of flowgraphs with the independent paths for the unit tests; Halstead's Metrics report with estimation of the total number of the delivered errors (B); Object-Oriented Metrics report with basic interpretation of the metrics; and recommendations to improve the code.

Michael Jeffords in his project, "Using SQA Metrics to Guide Refactoring of Medium to Large Scale Projects", developed a plan to reduce the overall complexity of the code while adding important functionality to the system that provided visualization of geophysical data. His code refactoring efforts took approximately 20 hours, but have reduced the number of tests by about 200 unit tests. Summarizing these efforts, he offered two effective methods of refactoring. In the first method, he used weighting factors to "boil" McCabe's and Halstead's metrics down using a formula $M = v/10 + ev/4 + V/3000$. This approach gives equal weights to v-cyclomatic and ev-essential complexity metrics. Combined they have double the weight of Halstead's V-metric of the program volume [3]. In the second method, three thresholds for cyclomatic complexities were used to chart improvement over time. His goal was to achieve 100% of methods below the cyclomatic complexity of 20 and less than 97% of methods below the cyclomatic complexity of 10. He achieved 100% for $v < 20$; 99.67% for $v < 15$, and 98.01% for $v < 10$. Michael described three stages of refactoring: 1) He started by refactoring classes to remove

public variables and adding accessors; 2) Next, he chose any of the methods that ended up on the "hot" list that he could understand the basic methodology, and then tried to tighten up the algorithms; 3) Finally, he worked on the hardest methods once he felt that his plan had taken shape and he could see a direction for fixing the architecture of the worst classes.

In the other project, Timothy Houle and Douglas Selent analyzed the complexity of the Light-Up Puzzle program [11], its maintainability, testability and metrics related to the quality of the software program's design. The application's vulnerabilities were identified by using the McCabe™ IQ tool. This approach helped them to identify vulnerable code areas, reduce error rates, shorten testing cycles, improve maintainability, and maximize reusability. In order to verify the effectiveness of the McCabe™ IQ tool, they re-factored the program in the areas reported to be highly complex and error-prone. After this, they compared the McCabe's Metrics reports on the initial analysis to the reports on the re-factored analysis and charted results to clearly indicate the improvements made to decrease complexity. In addition to the McCabe metrics, the students also added various UML diagrams [1], which helped them understand the concept of the program structure and identify areas where object-oriented design principles could be applied to increase code reusability and maintainability.

5 EFFECTIVENESS OF THE COURSE AND STUDENTS' RESPONSE

All 16 computer-science graduate students that took the SQA course in the spring of 2010 expressed in their course evaluations full satisfaction with course organization, content, and material delivery. The overall course-evaluation score was high (4.71 out of maximum possible 5.0). In their anonymous comments, students shared mostly favorable observations, e.g., "This is a practically-oriented course and it has a lot of demand in the job market…"; "The instructor opened my eyes to the importance and usefulness of SQA metrics…"; "I learnt a lot of how the SQA is handled in the software development process…"; "Great experience…", and "I met my expectations more than I thought." Later these students effectively used the SQA techniques to improve their software programs in other courses, including the final capstone projects. All the students are currently employed by local computer companies.

6 CONCLUSIONS

The author has described the challenges and experience of running software quality assurance courses for undergraduate seniors and graduate students. The knowledge of graph theory and its applications in software engineering is beneficial for students with Computer Science majors. Detailed analysis of code complexity reveals areas of code structures that should be revised. The code revision allows students to find those code areas with potential errors and to improve code design and testing practices. In particular, the resulting analysis can be used in identifying error-prone software, measuring optimum testing efforts, predicting the effort required to maintain the code and break it into separate modules, reallocating possibly redundant code, and providing a fundamental basis for unit and integration testing. The complexity code analysis and structured testing

methodology should become a necessary attribute of software design, implementation, testing, sustaining, and re-engineering practice and training.

REFERENCES

[1] Galin, D., *Software Quality Assurance: From Theory to Implementation*, New York: Pearson Education, 2004.

[2] Watson, A. H., McCabe, T. J., Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, *NIST Special Publication*, No. 500-235. Gaithersburg, MD: National Institute of Standards and Technology, 1996.

[3] Halstead, M. H., *Elements of Software Science*. New York: North Holland, 1977.

[4] Rosenberg, L.H., Applying and Interpreting Object-Oriented Metrics. In *Proceedings of the Tenth Annual Software Technology Conference, April 18-23, 1998.* Salt Lake City, UT, 1998.

[5] Miller, G., The Magical Number of Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. In *Psychological Review*, 63 (2), 81-97, 1956.

[6] Riabov, V. V., Networking Software Studies with the Structured Testing Methodology. In *Computer Science and Information Systems*. Athens, Greece: ATINER, 2005, pp. 261-276.

[7] Java Source Metric, SourceForge.Net, 2009, http://jsourcemetric.sourceforge.net/, retrieved January 25, 2011.

[8] Littlefair, T., CCCC: C and C++ Code Counter, SourceForge.Net, 1998, http://cccc.sourceforge.net/, retrieved January 25, 2011.

[9] SourceMonitor Freeware Program, Campwood Software, Inc., 2010, http://www.campwoodsw.com/sourcemonitor.html, retrieved January 25, 2011.

[10] COCOMO II, University of South California, 2010, http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html, retrieved January 25, 2011.

[11] Light-Up Puzzle Program, Puzzle-Loop.Com, 2009, http://www.puzzle-light-up.com/, retrieved January 25, 2011.

The Journal of Computing Sciences in Colleges

Papers of the Sixteenth Annual CCSC Northeastern Conference

April 15-16, 2011
Western New England College
Springfield, Massachusetts

Volume 26, Number 6 **June 2011**