DEVELOPING INSTRUCTIONAL MATERIAL FOR TEACHING SOFTWARE COMPREHENSION/MAINTENANCE

By
MILTON A. AUSTIN III

Bachelor of Science

Oklahoma Christian University

Oklahoma City, OK

1999

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May 2006

DEVELOPING INSTRUCTIONAL MATERIAL FOR

TEACHING SOFTWARE COMPREHENSION/MAINTENANCE

Thesis Approved:

M. H. Samadzadeh
Thesis Advisor

B. E. Mayfield

H. K. Dai

A. Gordon Emslie
Dean of the Graduate College

PREFACE

Software maintenance is a costly problem for industry, typically taking up to 50-75% of the cost of software development [Sommerville 04]. Traditional Computer Science programs often do not prepare students to face this problem. Since a large part of software maintenance is software comprehension, better comprehension methods are a major part of the answer to the problem. Students often do not know how to comprehend already written code and do not know how to work in groups, hence new graduates are typically forced to learn the very important skill of software comprehension on the job. It is proposed that students should be taught a standardized way of software comprehension in preparation for the software maintenance jobs most will have.

The comprehension/maintenance area of computer science education has not been extensively covered as a research topic. This work is a detailed proposal for a software maintenance course, using techniques utilized by other researchers in their efforts to teach software maintenance as well as the new idea of teaching software comprehension techniques in a required course. This work outlines a course that will be designed to better prepare students for work in the area of software maintenance by teaching them software comprehension methods. The course includes best practices, a large-scale project, and focuses primarily on code comprehension methods. This course represents a standardized way to teach students software comprehension skills that are needed in the industry.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. M. H. Samadzadeh. His ideas and guidance have been extremely helpful. His attention to detail is amazing and has drastically improved the writing quality of any paper he has reviewed for me. I hope that I will be able to collaborate with him many times in the future.

I would like to thank my committee members, Dr. Blayne E. Mayfield and Dr. H. K. Dai, for their serving on my committee.

I would like to thank Dr. John P. Chandler for giving me good advice throughout my graduate studies.

I would also like to thank Mr. James F. Cain, who helped me as a sounding board for ideas. Many of the ideas in this paper were bounced off him first.

I would also like to thank Professor Malcolm Munroe from Durham University for providing me with a citable version of a paper that one of his graduate students wrote several years ago. While I didn't end up using that paper in the final version of this work, it was a valuable starting point for someone new to software comprehension.

TABLE OF CONTENTS

Page

CHAPTER I

INTRODUCTION

1.1 Introduction to Software Comprehension

Software comprehension deals with studying how a programmer learns what computer code does. There are a number of theories on comprehension.

One popular theory is that a programmer makes a mental map of the code by looking at and recognizing various knowledge structures [Pennington 87]. These knowledge structures include specific domain knowledge as well as recognized structures in the code. For example, let us assume that a programmer sees a while loop in some code that he/she is trying to comprehend. The programmer will probably immediately look for the condition to exit the loop, how the condition is changed, and the end of the loop, because that is how while loops are structured in general. The mental map of the code is then used to predict what will happen next in the code [Pennington 87]. This mental map is from classical knowledge structure comprehension in psychology and is not necessarily related to a flowgraph (though the knowledge structure could be a flowgraph).

Knowledge in this area of software comprehension is usually gained through the scientific method and double blind experiments, which is somewhat unusual in the field of computer science. Knowledge structure comprehension is done in one of three ways:

top-down, bottom-up, or a mixture of both (called metacognition) [Pennington 87] [Shaft 95].

Another popular area of software comprehension research is measured comprehension. This area of software comprehension research uses graph-theoretic software models and some software metrics to measure the comprehendability of programs. McCabe's cyclomatic complexity [McCabe 76] is one of the best known graph-theoretic metrics. This metric can be used to devise a methodology for structured testing [McCabe 76]. Intuitively, programs with a higher cyclomatic complexity number should be more difficult to understand because they have more control flow branches. Other measures have also been suggested as possible measures to gauge program complexity. Some examples are: lines of code, scope metric, and various object-oriented metrics [Mathias et al. 99].

Yet another popular area of software comprehension is automatic program comprehension. In automatic program comprehension, the idea of "slicing" a program is used to break a large program into more manageable slices or smaller parts of the given program. So instead of trying to comprehend the program as a monolith, the programmer can try to comprehend the slices. By definition, a slice is a set of statements related by d4ata and control flow [Weiser 82]. Since identifying program slices is something that can be done programmatically, this process can be automated. Slicing can be useful for debugging of computer programs and during program comprehension [Agrawal and Horgan 90]. Research into this field generally deals with creating slicing tools [Wichaipanitch 03] [Wichaipanitch et al. 05] and making slicing algorithms more efficient [Larsen and Harold 96].

1.2 Statement of the Problem

Software maintenance is a problem that plagues the software industry. Since maintenance takes up approximately 50-75% of the cost of software development [Sommerville 04], anything that can be done to reduce that cost would be helpful. Program comprehension is the central to maintenance because a programmer who is working on a piece of code is generally not the original programmer of that software, so the programmer must take the time to learn what the code does. Even if the original programmer works on his/her own code during maintenance, that programmer may not remember what that code does, so still software comprehension manifests itself as a critical issue.

It is generally known that traditional computer science education does not adequately prepare a future programmer for software maintenance. Maintenance skills are usually learned on the job, and perhaps not learned well enough. To address this deficiency, a course was developed as part of this thesis work to teach programmers how to comprehend code to improve the process of software maintenance. Based on the programming maturity of the students and the courses taken previously, it seems that a course on software maintenance could be a required one-hour course, preferably a junior or senior level class. By teaching students comprehension techniques, students can be better prepared for their careers as programmers and software engineers.

CHAPTER II

MAINTENANCE EDUCATION

2.1 Current Status

Computer Science curricula generally do a great job of preparing students for being productive members of software development teams. However, computer science programs do less well at preparing students for the inevitable software maintenance task. While going through the courses at the Computer Science Department at OSU (and most other programs), generally students receive little or no training or practice on how to do software maintenance. The focus seems to be on teaching programming languages, programming skills, and the formal models and abstractions necessary for problem solving. However, to be productive in the software industry, students need to learn about software maintenance also. This thesis outlines a course to teach students how to write maintainable code, how to test that code, and how to understand a piece of code they are given.

It appears that software comprehension is the common underlying problem. Computer Science programs prepare students for new software development. However, students may not know how to test existing code systematically, correct it, or add to it because they might not have understood how the code works. Teaching students how to

comprehend code and write code that is maintainable is so important that it should be required and should be taught early before students are allowed to acquire bad or questionable programming practices and habits.

2.2 Previous Research into Software Maintenance Education

Software maintenance education has been studied before, but not extensively. Cornelius and his colleagues [Cornelius et al. 1989] advocated the idea that maintenance education should be a project-based workshop where students are given a large software project and are required to maintain it. They recommended that students work in small groups to mirror what is typically done in the software industry. They recommended no formal lectures but instead holding weekly meetings with the instructor. The workshop was divided into five phases: familiarization with the system, selection of a number of enhancements together with design of the selected enhancements, reviews, implementation of the enhancements, and new function testing and regression testing [Cornelius et al. 1989]. This class was available only to honors students in the second semester of their sophomore year.

Postema and her colleagues [Postema et al. 2001] also had a project-based class to teach maintenance as part of their regular software engineering course. Students were responsible for maintaining a large software project that was passed on from semester to semester and class to class. The project involved simulations. Part of the code may have been reused on different projects. This approach seemed to be mostly concerned with the four different types of maintenance (corrective, adaptive, perfective, and preventive) and not with the actual understanding of the code or how understanding the code could help in driving down the cost of software maintenance.

Allen and his colleagues [Allen et al. 03] suggested that maintenance can be learned by having the students work on a program that is being used by real customers. Their program was DrJava (available at drjava.sourceforge.net). Their students made enhancements to this open source Java IDE. This approach appears to be especially helpful in the field of comprehension because open source code is not always documented well. Perhaps the technique advocated by Allen and his colleagues [Allen et al. 03] could be adapted to a program comprehension course. However, Allen and his colleagues were not investigating comprehension. They were looking at teaching extreme programming to their students.

Collofello [Collofello 89] did not believe that the topic of maintenance would take a whole class. He taught the first four lectures of his software engineering class at Arizona State University on maintenance. He did bring up some valid issues with maintenance as far as a student's view of it is concerned. He dedicated a whole lecture to the background of maintenance and the myths about it. He spent a single lecture on going over top-down and bottom-up comprehension, a lecture on changing software and documenting changes, and a lecture on validating software changes. However, it seems that software maintenance experience is something that should be taught separately from a software engineering course.

Perhaps the most interesting and radical suggestion was put forth by Stephan H. Edwards [Edwards 03]. His idea was to rethink all of computer science education in terms of a test-first mentality rather than a code-first mentality. What makes his ideas so radical is that he proposed to first teach students how to test code, not how to write it. His argument is that this approach allows students to better understand how code works.

The main argument made is that students in introductory programming classes often only test the code on the sample data provided, which provides no guarantee that their program will work for other valid data. Code testing is a very important part of comprehension, and knowing how to test code helps a person to better understand how it works [Edwards 03]. However, the idea that students need to learn how to test their code before they know how to write code seems too extreme.

CHAPTER III

PROPOSED COURSE

3.1 Basis of the Course

The proposed comprehension course would combine some of the research that was discussed previously in Chapter II into a one-hour course. This course would be a junior level class. This course would have Computer Science II as a prerequisite. Other courses such as Data Structures I and Discrete Mathematics would be helpful to have as prerequisites, but they seem to be at too high a level for this class to easily fit into the current system. The Discrete Mathematics background that is needed in the proposed class, namely basic graph theory, can be taught in a single lecture, and the Data Structures background that might be helpful for the programming assignments can be worked around by using simpler data structures.

An ideal textbook for this course would be [Arthur 88]. The first four chapters of the book could be covered directly. The book has assignments in each chapter that reinforce the material. Also the book has great examples of various software engineering documents and, in the author's opinion, they are better than the examples offered by Sommerville [Sommerville 04]. However, this book is out of print. Sommerville's textbook [Sommerville 04] is a viable replacement. The two chapters on test planning and software testing could be covered directly in this course. A number of other topics

that are recommended to be included in the proposed course are also in Sommerville's book [Sommerville 04]. However these topics are not covered in specific chapters of the book.

The course would have a large-scale project that emphasizes working in teams. Several assignments would also be included in the course. These assignments correspond to each specific section of the course. It is assumed that the semester is going to consist of twelve 1-hour sessions, which includes time for tests. Ten sessions have been planned: Introduction to Software Maintenance, Introduction to Software Comprehension, Introduction to Software Documents (2 parts), Introduction to Graph Theory and Measuring Software Comprehensibility, Introduction to Software Testing, Planning for Successful Testing, and The Testing Toolbox (3 parts). The details of these sessions can be found in Appendix C. One session is to be used as the extra class that will be used for a guest lecturer.

3.2 Overview of the Lectures

The class will have a total of 13 class meetings. One class meeting will be taken up by the midterm exam, and the last class meeting will be used for the final exam. Eleven of the class meetings will be lectures. The lectures will cover topics in software engineering that aid in software maintenance with focus on software comprehension. The topics to be covered include: text structure comprehension, applying text structure comprehension to software engineering documents, measuring software comprehendability, and software testing. Lectures will be covered in more detail in Chapter IV and even more detail in Appendix C.

3.3 Overview of the Assignments

The homework in this class is going to be mostly the class project, although some short-answer homework assignments will be given. The short-answer assignments would be given either every week or as a larger assignment every other week.

The semester project would be a large-scale group programming project with an emphasis on reengineering. A large portion of the course grade should come from the project because it will make up the majority of the work in the class for the students.

The tests would also make up a large portion of the grade for the course. As designed, this course currently has two tests: a midterm and a final exam. The final exam would be cumulative as is appropriate. The coursework will be discussed in further detail in Chapter V.

CHAPTER IV

LECTURES

4.1 Introduction to Maintenance

In this section of the class, the students will be introduced to software maintenance and some of the myths about software maintenance. Students will be made aware of how much of the work in the software industry involves software maintenance. Also, students will be introduced to the four types of software maintenance: corrective, adaptive, perfective, and preventative [Pressman 05].

This section of the course is not meant to take up much time, and will merely introduce students to the underlying theme of software maintenance. The approach taken in this course is that software maintenance is a big problem that can be tackled with various "tools" at a programmer's disposal, and the rest of the course will be spent teaching the students how to use those tools.

The most useful tool to use during software maintenance is software comprehension. Students likely have not been exposed to software comprehension (and the ways to tackle the problem of software comprehension) before this course. Students will already know how to design and code computer programs in several different languages by the time they get to this course. Therefore, software comprehension is probably the most important tool missing from the students' toolset.

4.2 Software Comprehension Strategies

This section of the course will go over text comprehension theories and how to apply them to software in particular and also software documents in general. Text structure comprehension [Pennington 87] is probably one of the easiest comprehension strategies to understand, and that it is the first strategy to be covered. The theory of text structure comprehension is quite intuitive and already well proven in the field of psychology [Pennington 87]. What is new about the theory is applying it to computer programs. So, first students will be taught the general theory of text structure comprehension.

Since a large part of test structure comprehension theory deals with domain knowledge, students must have a way to gain that domain knowledge. Here, the domain is software engineering. Therefore, students will be taught how to read software engineering documents and the abstractions that are common in them. Since text structure comprehension applies to both software documents and code, exposing students to software documents will give them another chance to reinforce the material on text structure comprehension.

In the last part of this section, software metrics will be introduced as measures of comprehendability of software. Unfortunately, since this class does not have Discrete Mathematics as a required prerequisite, basic graph theory must be covered. The metrics that will be covered include lines of code, McCabe's cyclomatic complexity [McCabe 76] and several object-oriented metrics [Mathias et al. 99] the most important of which is lack of cohesion. The assignments would show the strengths and weaknesses of each measure (e.g., having the students calculate the cyclomatic complexity of a very large

program with a very simple call graph). Metrics used for comprehension are important because they are an objective way to measure the comprehendability of a program.

4.3 Using Software Testing

In this section of the class, students will learn comprehension through writing test plans for sample programs that are difficult to comprehend. This activity may not at first seem conductive to teaching program comprehension, but Edwards [Edwards 03] has shown that testing helps students understand how programs work. By learning to better test programs, students will learn a way to approach the comprehension task of any program given to them. All programming assignments after this point will require test plans. Also, students will be introduced to using various software engineering documents as a way to test software. Students will learn how to refer to the specification document to attempt to figure out how something is supposed to work. Assignments from this section of the course will require that all programs have a written specification and a test plan in addition to the program. In order to make it a little more realistic, students will be given other students' specifications and test plans.

The testing part of the course will be the largest section. First, a lecture will be spent introducing testing. Also, the topic of test plans must be covered in the following lecture. Then the subsequent three lectures will be spent introducing students to different types of testing and how they can be used to further software comprehension. Even if these skills are not used for software comprehension, they are still a useful skill to have as most computer science programs do not spend any time on teaching software testing techniques. The assignments in this section of the course will reinforce the various types

of testing and when they should and should not be used. Perhaps the most important concept that is to be reinforced is equivalence partitioning [Sommerville 04]. Equivalence partitioning is useful to almost all types of tests and can really help students (i.e., future software developers) design their tests so that they cover the code better.

4.4 Extra Lecture

The proposed schedule has time available for an extra lecture. This lecture should be the last one. Ideally, this lecture would be given by an experienced software engineer from industry, relating his or her experiences in software maintenance. If the teacher cannot find a guest lecturer, a lecture on the most extreme type of software comprehension, software reverse engineering, might be a good substitute. Showing students the most extreme example of software comprehension should show students how the skills they have learned in the class can be put to use. Also, the possibility exists that the time used in each lecture for the project will eat into the time allotted for the other lectures so that no time will exist for the extra lecture, in which case, the extra lecture would just be omitted.

4.5 Topics Not Covered

Due to the short nature of the course, many important topics in the research areas of software comprehension cannot be covered. Among those topics not covered is the area of automated software comprehension. Automated software comprehension is an important subfield of software comprehension. While automatic software comprehension shows great promise with respect to improving debugger functionality and ease of use, there is not enough time in the course to cover this topic. Also, the author believes that future debuggers might have slicing and dicing tools built in due to how useful they have

shown to be in the debugging process [Weiser 82]. It can be argued that, while interesting, the research field of automated software comprehension is not as useful as a skill as the other software comprehension techniques taught in this class. Students would need to create their own slicing tool in order to make the most use of slicing, probably one tool per language that the student programs in. Also, this topic is fairly advanced as compared to the other topics covered in this course. It is likely that many students would not understand this subject well enough given the compressed nature of the proposed course.

.

CHAPTER V

STUDENT COURSEWORK

5.1 Coursework Value

The bulk of the grade points available for the proposed course will be the two tests (20% for the midterm exam and 30% for the final exam) and the semester project (40%). Homework will have some grade points assigned to it (10%), but the number of points will be minor due to the relative ease in which students can resort to questionable collaboration on short answer questions. However, since the main goals of the homework sets are to reinforce the just-taught lessons and prepare students for the tests, having the homework worth set at such a small amount is acceptable.

5.2 Homework

In addition to the large scale project, students will be given homework. As in most software engineering courses, the format of the homework would be short answer questions (i.e., the question could be answered with a sentence or two of verbiage), with the exception of some graph theory questions. These short answer assignments should be given frequently, at least one every other class period. These assignments reinforce the core course material in ways the project could never reinforce them. For example, a student might never get an opportunity to do top-down comprehension in the large scale project. Students would be expected to pick a single text structure comprehension

strategy and stick with it. However, students will still need to know the difference between the different text structure comprehension strategies.

The short answer questions would be similar to the questions asked on tests. The intent of asking similar questions is to prepare students for the tests that are worth half of the total points in the class. Like the software engineering course taught here and at many other universities, there are many terms and definitions to be learned. The homework could reinforce the learning of those terms and definitions. Appendix D is a test/homework question bank consisting mostly of short answer questions.

Some topics in the class do not make for good short answer questions. For example, most of the section on measuring software comprehension and graph theory would not fit the short answer question template. For those sections of the course, it would be more appropriate to have more classic math-style homework. For example, one can give the student a directed acyclic graph (DAG) representing a program flowgraph and ask what the cyclomatic complexity is. Some questions in Appendix D are examples of this type of homework.

5.3 Semester Project

The project will include all aspects of software maintenance, including updating the software documents where appropriate. The project will give students a chance to observe and experience software maintenance firsthand. The project's program would be a large program, likely an open source program. If enough enhancements/software defect reports do not exist to support the project in the semester, the professor does have the option of forking the development tree and creating a copy of the program's code database. Then, the professor could assign enhancements/software defect fixes that

already have had programmers assigned to them. This solution does entail more work on the part of the professor. Care would have to be taken to make sure students do not copy code from the original source code.

The project must be something the students have never seen before in a programming sense. The important characteristics a program must have to be a good candidate project for this course are: an accessible code database (that includes licensing – the project must be legal), documentation having been written by multiple authors, being quite large, and, ideally, one that has an accessible software defect reporting/enhancement request interface, also known as a defect tracking system (like Bugzilla) [Mozilla 05]. Professor control of the code database is desirable because the professor may want to interject software defects to simulate latent software defects (a software defect that already exists from a previous software version in the code database) often existing in commercial code databases without affecting the code database used by others. What makes most open-source projects desirable as class projects is that they already meet most of those desirable traits. Some programs may lack documentation, and, if the professor does decide to fork the code database, the professor would have to setup the software defect reporting/enhancement request software locally (which is available from the Mozilla Organization [Mozilla 04] in Bugzilla).

DrJava [DrJava 05a], which was used successfully by Allen and his colleagues [Allen et al. 03] to teach extreme programming, would be a good choice as the project for the proposed course. This program could be interesting to students because it is a Java Integrated Development Environment that is written in Java. There are a large number of enhancement requests and software defect reports currently available for assignment.

DrJava is “a lightweight programming environment for Java designed to foster test-driven software development” [DrJava 05a]. The features most interesting from a teaching standpoint are the source code level debugger and the unit testing tool. Students could use the tool for the class and also as part of the project work on the tool. DrJava is an ideal tool to use because it has some documentation, has a Bugzilla site set up (if the professor doesn’t wish to fork), has been developed with contributions by many developers (the current team has 36 members), the code is quite large, and the code database is accessible (a fairly lenient license). The only problem with using DrJava is that the professor will not have local control and will not be able to introduce software defects. Also, there are many known unfixed software defects (well over 100 as of November 7, 2005) [DrJava 05b]. These known unfixed software defects could be a source of work for the class project.

The class would be broken into multiple groups to emulate a typical programming environment in industry. The project would include some coding, but since other skills taught in the course would need to be reinforced in the project, coding would not be the main focus. The students would need to work from and update a specification and some test plans, do the testing (either to conduct testing after they code something or to do the testing on code that is provided to them), and then measure the code to see how comprehendible it is. A good example of code comprehension exercise that this project can contain would be to make the groups responsible for testing one another's code.

Each task in the project will have a point value assigned to it. Each student will be required to earn a certain number of points to get full credit for the semester project. The professor will serve as the technical lead for all of the teams. The author initially

thought that a student could fill that role, but giving students experience in a technical lead role is beyond the scope of this course and also is risky because most students would have no management experience. The class will be broken up into several small groups of three to five members. Also, while not scheduled, discussions on the project would likely eat into the time allocated for the class, so some time at the end of each class will be allocated for the project.

5.4 Exams

There will be two exams in this course: a midterm exam and a final exam. The midterm exam will have material from lectures one to six. The final exam will be heavily weighted towards lectures seven to eleven (which includes material from the extra lecture), but it would have some material from the midterm.

The exams would be closed book and closed notes. The tests in this course would contain many term definitions. Allowing the book or notes at exam time would defeat the purpose of making sure the students learn the terms. Formulas for the comprehension metrics would be included on the exams. The intent of teaching the comprehension metrics is to give students a method to measure the comprehendability of a computer program. Of course, the most important part of comprehension metrics is knowing how to apply the metrics as tools and not merely knowing the formulas for the metrics.

The midterm exam would take up most of one class period. The first several minutes of the class period would be used for class project questions, but the rest of the time would be used on the exam. The midterm exam would have at least 10 short answer questions from the question bank in Appendix D. Those questions would make up 70% of the available points. There would be a single relatively complex cyclomatic

complexity question (similar to the one in Appendix D, but with more nodes). In this problem, students would be given the code to a method with a complex call graph. Students would be required to create the call graph and calculate the cyclomatic complexity of the method. This question would be worth 20% of the available points for the exam. The remaining 10% of points left would to graph theory questions.

At OSU, final exams are scheduled for around two hours. That extra time would be used to make a longer exam. Approximately 75% of the final exam (10 short answer questions) would be on the material from lectures seven to eleven. The remaining 25% of the exam (4 short answer questions and 1 cyclomatic complexity problem) would cover important topics from the first exam including (but not limited to): measured software comprehension, definition of text structure software comprehension, abstractions common in software documents, and software documents. Even though it would take more time than a typical short answer question, a cyclomatic complexity problem similar to the one on the midterm exam would be on the final exam because it is a very important topic.

CHAPTER VI

SUMMARY AND FUTURE WORK

6.1 Summary

Computer science education does not adequately prepare students for programming in an industrial environment. Students often learn nothing about software maintenance. Since software comprehension is the central issue in software maintenance, students should be taught good software comprehension techniques. Also since software testing is an important tool that can be used to help understand a program, a large amount of time should be spent on teaching software testing.

This thesis work proposes a course on the topic of software maintenance with an emphasis on software comprehension. Chapters III, IV, and V provide the details of the proposed course.

The material in this course will be comprised mostly of lectures, short answer homework assignments, the semester project, and two exams. The students will be given the opportunity to demonstrate and use the knowledge they have learned in the class on the homework assignments, which are modeled after the tests. Since there are only two exams, frequent homework assignments are a good way to reinforce the material being taught in class. Some graph theory problems would be included in the homework sets dealing with the measured software comprehension section of the class.

As discussed in Chapter V, a large-scale class project will be used to reinforce the software maintenance and software comprehension techniques learned in this class. It is argued that an open-source tool would be ideal for this project because it would give students a chance to see their code in action. An interesting side note is that the students can use the software tool they are working on for their other computer classes as well.

As explained in Chapter V, students' knowledge will be tested with exams that reinforce the course material. A question bank is given in Appendix D. The questions in the question bank include questions on software comprehension, software testing, various metrics, and the graph theory concepts underpinning some of the metrics.

6.2 Future Work

Future work in this field includes actually offering and conducting the course that is suggested in this thesis. Funding might be able to be procured from NSA to do a trial run of this course. The students in the class would be extensively surveyed to provide data to make the next incarnation of the course even better. The results of those surveys could be used as the bases for a future pedagogical research paper. Also, a textbook could be written that better fits the flow of the course. Currently, there seems to be no textbook that would be a perfect fit for the proposed course. The best candidate I have found, [Sommerville 04], does not quite fit the order of what is covered in this course. Also, Sommerville's book does not cover all of the topics that the author believes should be taught in a software comprehension course. A companion textbook to go with this course would be a valuable tool for students who would take the course. Most of the skills they need to learn would be in a single location rather than in the several books that are used as the sources for the material in this course.

REFERENCES

[Agrawal and Horgan 90] H. Agrawal and J. R. Horgan, "Dynamic Program Slicing", *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pp. 246-256, White Plains, NY, March 1990.

[Allen et al. 03] E. Allen, R. Cartwright, and C. Reis, "Production Programming in the Classroom", *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, pp. 89-93, Reno, NV, February 2003.

[Arthur 88] L. J. Arthur, *Software Evolution: The Software Maintenance Challenge*, John Wiley & Sons, New York, Toronto, and Singapore, 1988.

[Belady and Lehman 72] L. Belady and M. Lehman, "An Introduction to Growth Dynamics", *Statistical Computer Performance Evaluation*, W. Freiberger ed, Academic Press, New York, 1972.

[Brooks 83] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs", *International Journal of Man-Machine Studies*, Vol. 18, Issue 6, pp. 543-555, 1983.

[Collofello 89] J. S. Collofello, "Teaching Practical Software Maintenance Skills in a Software Engineering Course", *Proceedings of the 20th SIGCSE Technical Symposium on Computer Science Education*, pp. 192-184, Louisville, KY, February 1989.

[Cornelius et al. 89] B. J. Cornelius, M. Munro, and D. J. Robson, "An Approach to Maintenance Education", *Software Engineering Journal*, Vol. 4, No. 4, pp. 233-236, July 1989.

[DrJava 05a] "DrJava Project Info," *Sourceforge.net*, http://sourceforge.net/projects/drjava/ , Date Accessed: May 5, 2005, Date Created: Unknown.

[DrJava 05b] "DrJava Software defect List," *Sourceforge*.net, http://sourceforge.net/tracker/?group_id=44253&atid=438935, Date Accessed: November 7, 2005, Date Created: November 7, 2005.

[Edwards 03] S. H. Edwards, "Rethinking Computer Science Education from a Test-First Perspective", *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Languages, and Applications, OOPSLA*, pp. 148-155, Anaheim, CA, October 2003.

[Johnsonbaugh 04] R. Johnsonbaugh, *Discrete Mathematics*, 6th edition, Prentice Hall, Upper Saddle River, NJ, 2004.

[Kappleman et al. 98] L. A. Kappleman, D. Fent, K. B. Keeling, and V. Prybutok, "Calculating the Cost of Year-2000 Compliance", *Communications of the ACM*, Vol. 41, Issue 2, pp. 30-39, February 1998.

[Koenemann and Robertson 91] J. Koenemann and S. P. Robertson, "Expert Problem Solving Strategies for Program Comprehension", *Proceedings of the SIGCHI conference on Human Factors in Computer Systems*, pp. 125-130, New Orleans, LA, April 27 – May 2, 1991.

[Kent 99] Clement Kent, "The Chronology of Y2K Problems," *ACM SIGAPL APL Quote Quad*, Volume 26, Issue 1, pp. 6-7, January 1999.

[Larsen and Harold 96] L. Larsen and M. J. Harold, "Slicing Object-Oriented Software", *Proceedings of the 18th International Conference on Software Engineering*, pp. 495-505, Berlin, Germany, March 1996.

[Mathias et al. 99] K. S. Mathias, J. H. Cross, T. D. Hendrix, and L. A. Barowski, "The Role of Software Measures and Metrics in Studies of Program Comprehension", *Proceedings of the thirty seventh ACM Southeast Regional Conference*, originally distributed on cd-rom, available on line only at http://portal.acm.org/citation.cfm?id=306381, no page numbers available, Mobile, AL, April 1999.

[McCabe 76] T. M. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. SE-2, pp. 308-320, December 1976.

[MITRE 03] MITRE, "The Y2K Scorecard", *Y2K Home*, http://www.mitre.org/tech/y2k/docs/TRIAGE_SCORECARD.html, Date Accessed: April 6, 2005, Date Created: May 19, 2003.

[Microsoft 05] Microsoft, "Product Lifecycle Dates – Windows Product Family", *Microsoft Windows Help and Support*, http://support.microsoft.com/gp/lifewin, Date Accessed: April 6, 2005, Date Created: January 15, 2005.

[Mozilla 04] Mozilla Organization, "About::Bugzilla::bugzilla.org", http://bugzilla.org/about/, Date Accessed: December 01, 2005, Date Created: September 21, 2004.

[OMG 05] Object Management Group, "Introduction to OMG UML", *UML.org*, http://www.omg.org/gettingstarted/what_is_uml.htm, Date Accessed: April 21, 2005, Date Created: January 19, 2005.

[Pennington 87] N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs", *Cognitive Psychology*, Vol. 19, pp. 295-341, July 1987.

[Pressman 05] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th edition, McGraw-Hill Higher Education, New York, 2005.

[Postema et al. 2001] M. Postema, J. Miller, and M. Dick, "Including Practical Software Evolution in Software Engineering Education", *Proceedings of the 14th Conference on Software Engineering Education and Training*, pp. 127-135, Charlotte, SC, February 2001.

[Shaft 95] T. M. Shaft, "Helping Programmers Understand Computer Programs: The Use of Metacognition", *ACM SIGMIS Database*, Vol. 26, No. 4, pp. 25-56, November 1995.

[Sommerville 04] I. Sommerville, *Software Engineering*, 7th edition, Pearson Education Limited, Essex, England and Boston, MA, 2004.

[Swanson 76] E. B. Swanson, "The Dimensions of Maintenance", *Proceedings of the 2nd International Conference on Software Engineering*, pp. 492-497, San Fransisco, CA, October 1976.

[Weiner 78] L. Weiner, "The Roots of Structured Programming", *ACM SIGCSE Bulletin*, Volume 10, Issue 1, pp. 243-254, February 1978.

[Weiser 82] M. Weiser, "Programmers Use Slices When Debugging", *Communications of the ACM*, Vol. 26, No. 7, pp. 446-452, July 1982.

[Wichaipanitch 03] W. Wichaipanitch, "An Interactive Debugging Tool for C++ Based on Dynamic Slicing and Dicing", Computer Science Department, Oklahoma State University Doctoral Dissertation, UMI Number 3105798, August 2003.

[Wichaipanitch et al. 05] W. Wichaipanitch, M. H. Samadzadeh, and S. Tangsripairoj, "Development and Evaluation of a Slicing-Based C++ Debugger", Proceedings of International Conference on Information Technology: Coding and Computing (ITCC 2005), pp. 473-478, Las Vegas, NV, April 2005.

APPENDICES

APPENDIX A

GLOSSARY

Bottom-up Comprehension	A line-by-line analysis of code [Shaft 95].
CC	Cyclomatic complexity. CC is a complexity measure that represents the number of independent paths through a computer program's call graph [McCabe 76]. Cyclomatic complexity is a very important metric that has many uses that include: structured testing, maintainability, and software comprehensibility.
COTS	Commercial Off-The-Shelf. A term used to describe software or software components that can be purchased.
DIT	Depth of Inheritance Tree. An object-oriented metric that counts the length in a direct path from any given class in an inheritance tree to the root of the inheritance tree. This metric is believed to be a good estimator of software comprehensibility because each inherited method adds confounding factors and testing complexity [Mathias et al. 99].
IDE	Integrated Development Environment, a graphical program that helps its users develop code by including documentation help and debugging tools.
Knowledge Structure	Hypotheses about what is being read, they are based on how the category of things being read usually progresses and the specific domain knowledge; these hypotheses are either proven or disproven, and replaced as comprehension takes place [Pennington 87].

LCOM	Lack of Cohesion Metric. This metric measures the sharing of instance variables in a class, also called cohesion. This serves as a measure of comprehensibility because, intuitively, the more instance variables are shared in a class, the easier it is to understand. A lower measure indicates more cohesion with zero as the minimum [Mathias et al. 99].
Metacognition	Thinking about thinking, metacognition involves a conscious choice to change comprehension strategies [Shaft 95]
NOC	Number of Children. NOC is an object-oriented metric measuring the number of direct children classes a class has. Child classes increase the complexity of a class because they increase the number of interfaces that must be tested [Mathias et al. 99].
OMG	Object Management Group. A not-for-profit consortium that has created specifications for several modeling languages including but not limited to UML, XMI (a combination of UML and XML), and CORBA [OMG 05].
OOP	Object-oriented programming. Object-oriented programming represents a different computer architecture in which parts of the programs are organized into “objects”, independent computer program components that have data and methods associated with them. An object-oriented program works by having a set of objects work together towards some end.
Software Comprehension	The process of understanding code unfamiliar to the programmer [Koenemann and Robertson 91]

Software Maintenance	The process of making corrective, adaptive, perfective, or preventive changes to a software system.
Top-down Comprehension	A Code Comprehension method where a programmer develops a hypothesis about the structures and operations in a program by looking at the program as a whole or in large pieces, and then attempts to either verify or disprove that hypothesis [Shaft 95].
UML	Universal Modeling Language. UML is a modeling language that allows a user to create abstractions of many different types of applications. Its primary use is in requirements and specification documents, though it can also be used to model business processes [OMG 05].
WMC	Weighted Methods per Class. An object-oriented metric that is the sum of a complexity measure for each method in a class. Any complexity measure that can be used on a method can be used as long as the same complexity measure is used for each method. Common choices are source lines of code (SLOC) or cyclomatic complexity. This metric is believed to be a good estimator of the complexity of a class [Mathias et al. 99].

APPENDIX B

CLASS SCHEDULE

The proposed course is planned to have 13 class meetings. The last meeting is assumed to be an extended class period and will be used as time for the final exam. The midterm exam will take place in the seventh class meeting. The remaining eleven class periods will be as lecture time. One of the eleven lecture periods will feature a guest speaker (see section 4.4). The schedule below refers to lecture numbers, not class meeting numbers.

Lecture 1: Covering the syllabus, forming groups, introduction to Software Maintenance.

Lecture 2: Introduction to software comprehension.

Lecture 3: Introduction to software documents.

Lecture 4: Introduction to software documents, part 2.

Lecture 5: Introduction to graph theory and measuring software comprehension.

Lecture 6: Introduction to testing.

Lecture 7: Planning for successful testing.

Lecture 8: The testing toolbox.

Lecture 9: The testing toolbox, part 2.

Lecture 10: The testing toolbox, part 3.

Lecture 11: Extra lecture (see section 4.4).

Midterm exam: Between lectures 6 and 7.

Final: Covers mostly lectures 9-12 with some material from earlier lectures.

APPENDIX C

OUTLINE OF LECTURES

Appendix C contains the outlines for each lecture. Please note, only ten lecture outlines are in this appendix because one lecture is proposed to be done by a guest speaker from industry relating his or her experiences in the area of software maintenance (see section 4.4).

1. INTRODUCTION TO SOFTWARE MAINTENANCE

Note: This lecture is shortened due to the expected time required to go over the syllabus and to form project teams or groups.

- 1.1 What is software maintenance?
  - 1.1.1 Defined as "the process of changing software once it has gone into use" [Sommerville 04].
  - 1.1.2 Many software projects generally have a long lifetime after they are released.
    - 1.1.2.1 E.g.: Microsoft operating systems are usually supported for at least 5 years after release [Microsoft 05].
  - 1.1.3 A large maintenance problem – Y2k.

1.1.3.1 Problem happened because some older computer programs didn't handle the year 2000 [Kent 99].

1.1.3.2 Many of the systems involved were not expected to be in use at the turn of the century [Kent 99].

1.1.3.3 A very costly problem – estimated to have cost $125 billion for the private sector alone [Kappleman et al. 98].

1.1.3.3.1 Maintenance almost always costs more per line of code to do than original development [Pressman 05].

1.2 Four types of maintenance [Swanson 76] [Pressman 05]

1.2.1 Corrective maintenance.

1.2.1.1 Involves fixing a defect.

1.2.1.2 Defect can be of any kind including incorrect output, code not matching the documentation, and working around a hardware defect.

1.2.2 Adaptive maintenance.

1.2.2.1 Involves changing the software to work in a new environment.

1.2.2.2 E.g., Porting a program to a new OS or processor.

1.2.3 Perfective maintenance.

1.2.3.1 Involves adding enhancements to software already released.

1.2.3.2 E.g., introducing a new feature or attempting to improve the performance of software.

1.2.4 Preventive maintenance.

1.2.4.1 Involves making changes to software that do not constitute a functional change but instead make the software easier to maintain.

1.2.4.2 E.g., reducing the complexity of a method that still does the same thing after the change.

1.2.4.3 E.g., adding documentation to or improving the documentation of a previously undocumented/poorly documented part of a program.

1.3 The process and challenge of software maintenance.

1.3.1 The software life cycle.

1.3.2 As much as 75% of the cost of a software project comes from software maintenance [Sommerville 04].

1.3.3 The high cost of software maintenance (from [Arthur 88])

1.3.3.1 It is not unbelievable for software to cost over 40 times per line of code for a maintenance project versus its original development cost.

1.3.3.2 Each new project adds to the maintenance burden.

1.3.3.3 Demand for maintenance outpaces most software development organizations' ability to provide maintenance.

1.3.3.4 Difficult to maintain software projects are sometimes rewritten at a later point at great cost.

1.3.4.5 Software changes are often poorly designed and documented.

1.3.4.5.1 Design documents are often not updated to reflect changes.

1.3.4.6 The first two years following the release of a software system are often used to bring the software up to the users' original expectations.

1.3.4.7 Changing the software often interjects new software defects that later must be repaired.

1.3.4.7.1 Each time the code is touched for a change, the maintainer risks introducing new defects.

1.4 The process of software maintenance (show dataflow diagram for software maintenance, figure 1.2 from [Arthur 88]).

1.4.1 Change management.

1.4.1.1 Purpose: “to uniquely identify, describe, and track the status of each requested change” [Arthur 88].

1.4.1.2 Major activities [Arthur 88].

1.4.1.2.1 Enter change request: Software developers receive a request for a change, analyze the change, and generate a change request.

1.4.1.2.2 Track change request: Provide reports on the status of the change request.

1.4.1.2.3 Auditing: Provide a paper trail of the changes.

1.4.1.2.4 Information: Provide information to project management and quality assurance personnel.

1.4.1.4. Impact analysis.

1.4.1.4.1 Perhaps the most important part of the change management process.

1.41.4.2 Provides analysis opportunity.

1.4.1.4.3 What will the change affect? Literally means anything the change can affect from data structures to human users [Arthur 88].

1.4.1.4.4 How much will this change cost?

1.4.1.4.4.1 Includes intangible costs which include the following items (from [Pressman 05]):

1.4.1.4.4.1.1 How much customer satisfaction is lost if the change is not made?

1.4.1.4.4.1.2 How much of a decrease in overall software quality will happen due to defects introduced while making this change?

1.4.1.4.4.1.3 How much will making the change impact projects currently in development if staff must be reassigned to work on this problem?

1.4.1.4.5 Not all changes that are requested are made.

1.4.2 Release planning.

1.4.2.1 Includes selecting and ranking enhancements to be included in the next release package [Arthur 88].

1.4.2.2 The document is placed under configuration management.

1.4.2.3 Work and development resources are then scheduled to work on the release.

1.4.2.4 Document is updated to the status of the tasks planned for the release.

1.4.3 Development work.

1.4.3.1 Includes design, coding, and testing.

1.4.4 Release.

1.4.4.1 The software with the new changes is packaged and sent to the user.

1.4.4.2 Included with this software is any documentation changes that are required.

2. INTRODUCTION TO SOFTWARE COMPRHENSION

2.1 What is software comprehension?

2.1.1 Defined as "the process of understanding program code unfamiliar to the programmer" [Koenemann and Robertson 91].

2.2 Why is software comprehension important?

2.2.1 Comprehension is a central issue to software maintenance.

2.2.2 Software maintenance takes up 50-75% of all software development costs [Sommerville 04].

2.2.3 Cost of software maintenance is defined as: $M = p + K^{(c-d)}$ where p is productive effort, and the second term is nonproductive work or "wheel turning" [Belady and Lehman 72] [Pressman 05].

2.2.3.1 Specifically, K is a constant, c is the measure of complexity as a result of poor design and documentation, and d is a measure of familiarity. [Belady and Lehman 72].

2.2.3.2 Good software comprehension techniques can increase d, leading to higher productivity in software maintenance.

2.3 Why should you care?

2.3.1 Less time spent on non-productive activities means more time left for working on productive activities.

2.3.2 Good software comprehension techniques give you an advantage over your coworkers which could possibly lead to quicker promotions and more pay.

2.3.3 Non-productive time during software comprehension is frustrating.

2.4 Software comprehension strategies.

2.4.1 Intent is to give students a comprehension toolbox to use when in maintenance.

2.4.2. Text structure comprehension.

2.4.2.1 Based off classical psychology theories on how one learns when one reads [Pennington 87].

2.4.2.2 The theory is that as a person reads something, she/he recognize the various knowledge structures that include specific domain knowledge (what she/he knows about how this sort of program works) and recognizes structures in the code [Pennington 87].

2.4.2.3 E.g., a programmer sees a while loop in some code the programmer is trying to understand.

2.4.2.3.1 The programmer uses her/his knowledge of how while loops work to look for the stop condition, how that condition changes, and the end of the loop because that is how while loops are structured in general.

2.4.2.3.2 The programmer then forms a mental map of the code to predict what happens next.

2.4.2.5 Three techniques to form the mental map.

2.4.2.5.1 Bottom-up comprehension: The programmer creates a general hypothesis about a program as a whole and then uses that hypothesis to create more specific hypotheses [Brooks 83].

2.4.2.5.2 Top-down comprehension: line-by-line analysis of the code [Pennington 87].

2.4.2.5.3 Note: Metacognition, the process of changing comprehension strategies from bottom-up to top-down or vice versa in an attempt to better comprehend the code, has not been shown to be helpful [Shaft 95].

2.4.2.6 The uses of text structure comprehension.

2.4.2.6.1 Can be used in both code and the documents related to coding such as specifications and test plans.

3. INTRODUCTION TO SOFTWARE DOCUMENTS

3.1 What is a software document?

3.1.1 A software document can be any document produced during the software development process.

3.1.2 Most important ones in terms of software comprehension are the requirements document and the design document.

3.1.3 Both offer valuable clues into how software works: the requirements document documents how software is supposed to work and the design document lays out the design.

3.1.4 Can be used in a text structure comprehension strategy.

3.1.5 One can predict how these documents are laid out with enough practice.

3.2. Abstractions.

3.2.1 What is an abstraction?

3.2.1.1 Abstractions means that a software document is simplified or displayed in a manner that might be easier to understand.

3.2.2 Abstractions are common in software documents.

3.2.3 Dataflow diagram or DFD [Sommerville 04].

3.2.3.1 Functional transformations that transform input to output

3.2.3.2 Advantages: Supports the reuse of transformations, intuitive (many people can think of work in terms of input and output), adding new transformations is easy, making changes in the DFD in the process of maintenance/evolution is easy, and it is easy to implement a concurrent or a sequential system.

3.2.3.3 Disadvantages: Programs with I/O with complicated interfaces are too complex to model as a DFD and there must be a common format for data transfer among transformations.

3.2.4 Object models [Sommerville 04].

3.2.4.1 The overall system is turned into a set of objects with well defined interfaces with each other.

3.2.4.2 Object models are concerned with the classes, their attributes, and their methods.

3.2.4.3 Advantages: Objects can be changed without affecting other objects (since the objects are loosely coupled); objects are usually representations of things in the real world, so the model is generally naturally

understandable; objects are reusable; and object-oriented languages exist to help provide direct implementation of the models.

3.2.4.4 Disadvantages: Interface changes often require extensive analysis to identify the objects that use that interface and make sure no other objects are affected, and complex real-world entities are difficult to represent as objects.

3.2.5 Unified Modeling Language or UML [Sommerville 04]

3.2.5.1 A powerful modeling language that can be used to abstract many activities.

3.2.5.2 Usually used in conjunction with a software tool.

3.2.5.3 We will look at only a few types of UML diagrams since this is a large subject

3.2.5.4 UML State Diagrams.

3.2.5.5 UML Inheritance Diagram.

3.2.5.6 UML Use Cases.

3.2.5.7 Advantages: UML is a powerful model, capable of modeling many processes and things.

3.2.5.8 Disadvantages: Changing your requirements and design methodology may require the purchase of a new UML tool since UML tools are usually targeted at a specific methodology [OMG 05].

4. SOFTWARE ENGINEERING DOCUMENTS PT 2

4.1 Continuation of program design abstractions.

4.1.1 Flowcharts.

4.1.1.1 One of the oldest program design abstractions.

4.1.1.2 Advantages: They are easily understood by most programmers (since most have received training in using flowcharts in college), and they depict flow control quite well

4.1.1.3 Disadvantages: Flowcharts can be bulk insensitive and insensitive to flow of the data.

4.1.2 There is no best abstraction.

4.1.3 Usually, more than one abstraction is used in software documents due to the differing strengths and weaknesses of different types of abstractions.

4.2 Requirements documents [Sommerville 04].

4.2.1 A requirements document contains the user and system requirements for the software.

4.2.2 User requirements.

4.2.2.1 A high-level abstraction.

4.2.2.2 Usually contains natural language with diagrams and defines what the software is supposed to do and not to do as well as the conditions under which it must operate.

4.2.3 System requirements.

4.2.3.1 A “detailed description of what the system should do” [Sommerville 04].

4.2.3.2 These are the low-level requirements.

4.2.3.3 A document with all of the low level requirements is sometimes called a functional specification.

4.2.3.4 Can serve as a contract between the software developer and the purchaser of the software.

4.2.3.5 The functional specification is often turned into the software design specification by adding more detailed design information.

4.2.4 Functional and nonfunctional requirements.

4.2.4.1 A functional requirement is something the software must do functionally.

4.2.4.1.1 E.g., the software must create an unique user id for each new user and a unique session id for each session.

4.2.4.2 A nonfunctional requirement is something the software that is not directly related to what the software must deliver functionally.

4.2.4.2.1 E.g., performance requirements and time/space restrictions that the software must operate under.

4.2.5 Requirement documents are usually in one of 4 forums: Structured Natural Language, Graphical Notations, Design Description Language (DDL), or Mathematical Specifications [Sommerville 04].

4.2.6 Why are requirements documents useful?

4.2.6.1 They document how the software is to behave.

4.2.6.2 Even requirements documents for past versions of the software are useful since old requirements are usually not entirely superseded by new requirements.

4.2.6.3 New requirements due to perfective maintenance can but do not always supersede the old requirements.

4.2.6.4 More often than not, software after a maintenance change must still meet the old requirements.

4.2.7 If there is a change made to the requirements, the requirements document will need to be updated accordingly.

4.3 Design documents [Pressman 05].

4.3.1 Usually heavily based on the functional specification.

4.3.2 Usually contain abstractions of some sort.

4.3.3 Contains information about the models, objects, data, file structures, and initial integration test plans.

4.3.4 Go over Table 10.1 (FIND OUT WHAT THIS TABLE IS DREW) in [Pressman 05].

4.3.5 Perhaps the most important section is requirements-to-module matching.

4.3.5.1 Makes sure all requirements are covered.

4.3.6 Like the requirements documents, this document must be updated when a change is made to the software.

5. INTRODUCTION TO GRAPH THEORY AND MEASURING SOFTWARE COMPREHENDABILITY

5.1 Graph theory (from [Johnsonbaugh 04])

5.1.1 A short lesson in graph theory is needed to lay out some of the basics needed for a specific and important software engineering metric.

5.1.2 Graph (show example).

5.1.2.1 Circles are nodes.

5.1.2.2 Lines are edges.

5.1.2.3 A path is a list of nodes where the end node is connected to the rest.

5.1.2.4 Path example.

5.1.3 Cycles.

5.1.3.1 Acyclic graphs.

5.1.3.2 Example.

5.1.4 Directed graphs.

5.1.4.1 Example.

5.1.5 Strongly-connected graphs.

5.1.5.1 There is a path between any pair of nodes.

5.1.6 Cyclomatic number or nullity of a graph

5.1.6.1 Nullity of a graph is the number of edges that must be removed from the graph to make it acyclic.

5.1.6.2 Formula: C = e – n + l for a graph with e edges, n nodes, and p connected components.`

5.2 Flowgraphs.

5.2.1 A flowgraph is a directed graph that depicts the control flow of a program.

5.2.2 Each node consists of sequential blocks of code and the edges are transfers of control.

5.2.3 Control flow changes include method calls, loops, and conditional statements.

5.2.4 Flowgraphs are not strongly connected.

5.3 Measuring software comprehensibility.

5.3.1 Some software metrics can be good estimators of how hard a piece of software is to understand.

5.3.2 Not all metrics measure the same thing, and a mix of some of these measures should be used because some have glaring weaknesses.

5.3.3 Lines of code.

5.3.3.1 A classic measurement that is used to depict the size of a program.

5.3.3.2 Insensitive to flow complexity.

5.3.4 Cyclomatic complexity (from [McCabe 76]).

5.3.4.1 Uses the flowgraph of a program.

5.3.4.2 A good measurement for the flow complexity of a program.

5.3.4.3 Insensitive to bulk.

5.3.4.4 How to calculate?

5.3.4.4.1 Make the flowgraph of the program.

5.3.4.4.2 CC = C + 1 or Cc = e – n + 2p, for a graph with e edges, n nodes, and p connected components

5.3.4.4.3 Why +1? Because McCabe suggested adding an extra edge to make the flowgraph strongly connected in order to be able to calculate the cyclomatic complexity.

5.3.4.4.4 The cyclomatic number measures the number of unique paths in a strongly connected graph.

5.3.4.4.5 The extra edge is added from the ending node to the beginning node of each component.

5.3.5 Object-oriented metrics (from [Mathias et al. 99]).

5.3.5.1 Object-oriented programs have a different set of metrics that can be used.

5.3.5.2 Weighted Methods per Class (WMC).

5.3.5.2.1 WMC = sum of complexities of all methods in a class.

5.3.5.2.2 Based on the belief that classes with a large number of methods are more difficult to maintain.

5.3.5.2.3 Any complexity measure can be used for the methods.

5.3.5.3 Depth of Inheritance Tree (DIT).

5.3.5.3.1 The more inherited methods there are along a hierarchy, the more complex the class is.

5.3.5.3.2 Count the number of classes in a maximum length path from the root of the inheritance tree to the current class.

5.3.5.3.3 A good measure of complexity since integration testing will be required due to the fact that the inherited methods must be integration tested with the current class.

5.3.5.4 Number of Children (NOC).

5.3.5.4.1 The more child classes there are, the more complex a class is.

5.3.5.4.2 Count the number of direct child classes.

5.3.5.4.3 A good measure of complexity since integration testing will be required due to the fact that the inherited methods from the current class must be integration tested with the child classes.

5.3.5.4.4 When using this metric to show integration test complexity, do not also use DIT.

5.3.5.5 Lack of Cohesion Metric (LCOM)

5.3.5.5.1 Checks to see if the methods in a class use the same instance variables.

5.3.5.5.2 If the instance variables are shared, the class has good cohesion and is therefore less complex.

5.3.5.5.3. Defined mathematically as follows.

5.3.5.5.3.1 Let a class C have n methods, M_1, M_2, ..., M_n.

5.3.5.5.3.2 Let $\{I_j\}$ be the set of instance variables used by method M_j.

5.3.5.5.3.3 There are n such sets $\{I_1\}$, $\{I_2\}$, ... $\{I_n\}$.

5.3.5.5.3.4 Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset, i \neq j \text{ and } 1 \leq i, j \leq n\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset, i \neq j \text{ and } 1 \leq i, j \leq n\}$

5.3.5.5.3.5 If all n sets $\{I_1\}$, $\{I_2\}$, ... $\{I_n\}$ are Ø, then $P = \emptyset$.

5.3.5.5.3.6 LCOM = $|P| - |Q|$, if $|P| > |Q|$ otherwise 0.

6. INTRODUCTION TO SOFTWARE TESTING [Sommerville 04]

6.1 Software testing deals with verification and validation of the software.

6.1.1 Verification refers to the software matching its requirements.

6.1.2 Validation means that the software works correctly.

6.2. Types of software testing.

6.2.1 Unit testing is the testing of individual components.

6.2.1.1 Usually done by exercising the interface of an individual component with known good and bad input.

6.2.1.2 E.g., testing the interface of a single method with valid and invalid parameters.

6.2.2 Module testing is the testing of a group of related units.

6.2.2.1 Usually done to test how a group of components work together.

6.2.2.2 E.g., an abstract data type or an object.

6.2.3 Sub-system testing is the testing of a group of related modules.

6.2.3.1 Done to test the interactions between various related modules, specifically the interfaces between the various modules.

6.2.3.2 E.g., testing an object that uses another object or set of objects.

6.2.4 System testing is a type of testing that involves testing the whole system.

6.2.4.1 Also called integration testing.

6.2.4.2 Concerned with two areas mostly.

6.2.4.2.1 Sub-system to subsystem interactions. The interfaces between the various subsystems work as expected.

6.2.4.2.2 Requirements: The software performs as the functional and nonfunctional requirements specify it must

6.3 Different goals of testing.

6.3.1 Defect testing.

6.3.1.1 Defect testing is used to find inconsistencies between the requirements specification and the program [Sommerville 04].

6.3.1.2 This type of testing focuses on trying to break the program.

6.3.1.3 Operational reliability can be estimated by counting the number of defects found with defect testing

6.3.2 Statistical testing.

6.3.2.1 This type of testing focuses on the program's performance and reliability.

6.3.2.2 Performance is a measure of how responsive a program is and how fast it is able to process sample data.

7. PLANNING FOR SUCCESSFUL TESTING

7.1 Introduction to test plans.

7.1.1 Each part of the testing process must be planned in detail.

7.1.2 A document is created to plan the testing of the program.

7.1.2.1 Like other software documents, this document evolves through the life cycle of the software project.

7.1.3 Test planning is more concerned with documenting the standards for software testing than the actual product tests.

7.1.4 Structure of a test plan (taken directly from [Sommerville 04]).

7.1.4.1 The testing process,

7.1.4.1.1 A description of the major phases of the testing process.

7.1.4.1.2 Requirements traceability.

7.1.4.1.2.1 Do the requirements map directly to design?

7.1.4.1.2.2 Each requirement should be individually tested.

7.1.4.1.3 Tested items.

7.1.4.1.3.1 The items in the software that are to be tested should be specified.

7.1.4.1.4 Testing schedule.

7.1.4.1.4.1 When are the tests going to be done and in what order?

7.1.4.1.4.2 Also includes resource allocation for the testing (which could include developers or specific machines).

7.1.4.1.4.3 Tied directly to the overall development schedule.

7.1.4.1.5 Test recording procedure.

7.1.4.1.5.1 How are the testing results recorded?

7.1.4.1.5.2 The results must be recorded so that they may be audited at a later date if necessary.

7.1.4.1.6 Hardware and software requirements.

7.1.4.1.6.1 This section refers to what is required for the actual testing, not to the software that is being tested.

7.1.4.1.6.2 Does new testing software need to be developed?

7.1.4.1.6.3 What hardware is required for testing?

7.1.4.1.7 Constraints.

7.1.4.1.7.1 Anything that could potentially negatively affect the testing process should be listed here.

7.1.4.1.7.2 Should include a plan of how to get around the problem.

7.1.4.1.7.3 Risk management activities should also be listed under constraints.

7.1.4.1.7.4 Some example problems are: staffing shortages, test machine failure, and test development not being done in time.

7.1.4.1.8. Resources.

7.1.4.1.8.1 What is needed to physically run the test?

7.1.4.1.8.2 Includes people as well as equipment.

7.1.4.1.8.3 Used for planning purposes.

7.1.4.1.8.4 E.g., a specialized piece of equipment may be needed to run the test.

7.1.4.1.8.5 E.g., temporary test operators may need to be hired.

7.1.4.1.8.6 Should include the cost of creating and running the tests.

7.1.4.1.9. Tests.

7.1.4.1.9.1 What tests are possible to be run?

7.1.4.1.9.2 How is a test considered to be passed?

8. THE TESTING TOOLBOX

8.1 This section is meant to expose the students to different testing strategies.

8.2 Inspections – a very different kind of testing (from [Sommerville 04]).

8.2.1 Inspections are always carried out by software developers, testing could be done by either by developers or testers.

8.2.2 Inspections do not necessarily require the program to be executed.

8.2.3 Inspections are an inexpensive way to find software defects early in the software development process.

8.2.3.1 The earlier software defects are found, the less cost to fix them

8.2.4 Inspection advantages:

8.2.4.1 A number of defects can be found in a single inspection.

8.2.4.1.1 Testing can sometimes only find one error per test because a defect may cause the program to crash or cause other defects not to be shown.

8.2.4.2 They reuse domain and programming language knowledge.

8.2.4.2.1 Inspectors are likely to have seen the types of errors that are common in the programming language used and also in the type of application under consideration.

8.2.4.2.2 Using knowledge structure comprehension theory, inspectors are likely to be able to predict how the program should work and are well suited to detecting and locating abnormalities.

8.2.5 Inspections should not replace system testing.

8.2.5.1 System testing is still necessary because the inspectors are not likely to find all the software defects or to verify that all requirements have been covered.

8.2.5.2 Because they occur around the same time as the unit test, inspections are just a way to find defects fairly early in the development process.

8.2.5.3 Inspections and testing have different advantages and disadvantages. They compliment each other well, but they do not replace each other.

8.3 Black box testing.

8.3.1 The tests are derived from the specifications.

8.3.2 The tests are scalable – black box testing can be used on software items as small as a single method or on large items like classes or sets of interfacing classes.

8.3.3 The software item being tested is viewed as a “black box,” which means that the internal parts are not known and do not matter.

8.3.4 The item being tested is fed input data and then the results are compared to what they should be from the specification.

8.3.5 Test input data is often selected using the tester's domain knowledge.

8.3.5.1 The tester might know of certain input data values that have been a problem in the past

8.3.6. Test input data selection can be improved by using Equivalence Partitioning.

8.3.7 Equivalence Partitioning.

8.3.7.1 Input to a program often falls into several classes.

8.3.7.2 E.g., all positive numbers, all negative numbers, etc.

8.3.7.3 Programs will behave in a similar way for all members of each class.

8.3.7.4 Identifying all of the equivalence classes is a way to do systematic testing.

8.3.7.5 Once all equivalence classes have been identified, test cases can be designed so that a specific input data is inside or outside a particular equivalence class.

8.3.7.6 Boundaries between equivalence classes, as well as data input members right before and after the boundaries, are good candidates for test cases.

8.3.7.7 Show example.

9. THE TESTING TOOLBOX, part 2

9.1 White box testing.

9.1.1 This testing uses knowledge of the structure of the code to derive tests and test data.

9.1.2 Useful typically only for small testing units.

9.1.2.1 Another testing technique that greatly benefits from Equivalence Partitioning.

9.1.2.2 Sometimes knowledge of the code structure can lead to further partitions of data.

9.1.3 White box testing is quite similar to black box testing.

9.1.4 Show example

9.2 Static analysis [Sommerville 04]

9.2.1 Uses a software tool to scan the source code for possible faults.

9.2.2 Searches for a limited set of errors including data faults, control faults, I/O faults, interface faults, and storage management faults [Sommerville 04].

9.2.3. Helps to combat error-prone features of a language, such as C's ability to cast anything as almost anything else.

9.2.4 Included on all Linux and Unix systems – the LINT utility.

9.2.5 Also available as a COTS for many other languages and operating systems.

9.2.6 Not as valuable for more modern, strongly typed languages like Java.

9.2.7 Can produce voluminous output, not all of it being actual defects.

9.2.8 The author's experience is that engineers tend to ignore the output of a static analysis tool if it isn't set up properly (i.e., if it produces many errors).

9.3 Path testing.

9.3.1 Path testing is a testing strategy that is concerned with testing all paths in a program.

9.3.2 Path testing also ensures that the branch conditions (looping and if conditions) are tested.

9.3.3 E.g., if the code from the “if” and “else” part of an “if... else” block is run in separate tests, then one can safely assume the condition was tested since all possible branches were tested.

9.3.4 Usually uses a dynamic program analyzer to discover the flowgraph of the program.

9.3.5 Then the test cases are derived to test as many paths as feasible.

9.3.6 Not all paths are tested.

9.3.6.1 There could potentially be an infinite number of path combinations in some programs.

9.3.6.2 For programs with a high cyclomatic complexity, it may not be feasible to test all paths, especially if some are trivial.

9.3.7 A common software engineering metric is test path coverage.

9.3.7.1 Test path coverage = # paths tested / cyclomatic complexity.

9.3.7.2 Show example.

9.4 Integration testing.

9.4.1 Integration testing is concerned with testing either the whole or a partial system.

9.4.2 One problem with integration testing is finding out where discovered defects originate.

9.4.3 Incremental testing can be done to get around this problem.

9.4.3.1 Incremental testing tests an initially small system and then adds on other parts until the whole system is tested.

9.4.4 Integration testing can be useful for software comprehension, specifically reverse engineering of a large system.

9.4.5 Several approaches to integration tests are possible

10. THE TESTING TOOLBOX, part 3

10.1 Integration testing (continued) (from [Sommerville 04]).

10.1.1 Top-down vs. bottom-up testing.

10.1.2 Bottom-up testing tests the smallest components first and then moves up until the whole system is tested.

10.2.3 Top-down testing tests the overall system first and then moves down until the smallest components are tested.

10.2.3.1 Top-down testing requires that smaller program components that are not yet implemented be simulated by stubs.

10.2.3.2 Stubs have the same interface as the smaller component, but have very little actual functionality.

10.2.3.3 Stubs are usually either a very simplified version of the final component or a small program that allows testers to interject return values based on the inputs with I/O.

10.2.3.4 E.g., the return values from a method could be simulated by user input.

10.2.4. Comparing top-down and bottom-up testing.

10.2.4.1 Top-down testing is much better at discovering defects in the system architecture and high level design.

10.2.4.1.1 An integration test that is focused solely on bottom-up testing cannot run until large parts of the program are completed.

10.2.4.1.2 This means that these specific software defects are found later in the development and are more costly to fix than if they had been found earlier.

10.2.4.2 When top-down development is used, a working system is available early on, although the system would be missing much of its functionality.

10.2.4.2.1 Top-down testing is more difficult to implement than bottom-up testing because the program stubs must be created.

10.2.4.3 Both testing strategies run into difficulties sometimes in test protocol observation.

10.2.4.3.1 The software being tested is not often designed to allow for easy viewing of the internal operation (e.g., return values of methods, etc.).

10.2.4.3.2 Testers usually have to create an artificial testing environment to get the results of the tests.

10.2.4.3.3 E.g., running a program in Visual Studio's debug mode.

10.2 Interface testing.

10.2.1 Interface testing is used to test any defined interfaces in the system.

10.2.2 Interfaces are usually method calls but can be object interaction as well.

10.2.3 Interface testing is intended to defect faults due to interface errors or faulty assumptions about interfaces [Sommerville 04].

10.2.4 Interface testing benefits from equivalence partitioning.

10.2.5 Interface testing is especially important in a weakly-typed language, such as C, where the compiler may not catch many of the interface errors.

10.2.6 Can use static analysis to find some interface problems.

10.2.7 General guidelines are:

10.2.7.1 List each call to an external component. Design a set of test cases where the values of the parameters are at the extreme ends of the ranges because "Extreme values can cause interface inconsistencies" [Sommerville 04].

10.2.7.2 When pointers are passed, test the interface with null pointers.

10.2.7.3 When a component is called through a procedure, design a test that should cause the component to fail because "Differing failure assumptions" can cause "specification misunderstandings" [Sommerville 04].

10.2.7.4 In a message passing system, design a test that will generate many more messages than are expected. This is called stress testing, and, in a message passing system, this will find timing problems.

10.2.7.5 When shared memory is used, design tests to change the order in which components access the shared memory. These tests could reveal incorrect assumptions about the order in which the shared memory is used [Sommerville 04].

10.2.5 Show Example.

10.3 Stress testing.

10.3.1 Stress testing is intended to expose the software to extreme conditions it would likely not encounter.

10.3.2 Some software is designed to handle a certain load.

10.3.3 E.g., operating systems.

10.3.4 Stress testing tests the failure behavior of the system.

10.3.5 Stress testing will cause the software to fail.

10.3.5.1 Failure should not result in data corruption or unexpected loss of service.

10.3.5.2 Stress testing helps to prove that a system that was designed to fail does so gracefully.

10.3.6 Stress testing also can cause other defects to show up that might not be seen under a normal testing load.

10.3.7 However, defects found with stress testing may not be likely to occur normally and may also be hard to track down.

10.3.8 Show example.

APPENDIX D

TEST/HOMEWORK QUESTION BANK

This test/homework question bank is intended to give the professor a series of questions to choose from for assignments and tests. More questions are given than are intended to be assigned. The intention is that every two class periods a new written assignment should be given. The questions are broken out by class period. Answers are also given here.

While there are thirteen class meetings planned, only ten of those class meetings will be used as planned lectures. One will be used for the extra lecture (see section 4.4), and two will be used as tests. There will be no assignments to reinforce the extra lecture.

1. Introduction to Software Maintenance

Define software maintenance.

Software maintenance is defined as "the process of changing software once it has gone into use" [Sommerville 04].

Why was the Y2k maintenance issue such a large problem?

Older computers couldn't handle the year 2000 properly because many of them truncated the first two digits of the year. Also, many of these older systems were not expected to be in use for so long [Kent 99]. The high cost of addressing this problem ($125 billion for the private sector [Kappleman et al 98]) makes it unique.

What are the four types of software maintenance? Describe each type of software maintenance.

The four types of software maintenance are corrective (involves fixing a specific defect), adaptive (involves changing the software for it to work in a new environment), perfective (involves making new enhancements to post-release software), and preventive (involves changes to software that are not a function change but instead are intended to make the software easier to maintain).

Why is software maintenance so expensive?

Software maintenance can be looked at as an organizational problem. Older software is still in use well after its intended end date. Often this software must still be maintained. Each new project a software organization makes adds to this maintenance burden. The demand for maintenance usually is greater than a software organization's ability to provide maintenance.

Also, maintenance can be looked at as a people problem. Software changes are usually poorly designed and documented. Sometimes the supporting documentation (design documents, etc.) are not updated to reflect new changes. Even if the same programmer works on the maintenance of a certain section of code, the programmer will probably not remember how it works.

Software maintenance is expensive also because it impacts the quality of the software. New software defects can be, and usually are, introduced during and as a result of software maintenance. These software defects must be fixed, thus adding to the maintenance required.

What are the steps in the process of change management?

In change management, first the change request is entered. Then changes to that change request are tracked. Also happening during this process is auditing, i.e. a paper trail. Information on the change is provided to project management and quality assurance personnel. An impact analysis is performed. If the change is going to be made, release plans are made, the code is designed, written, and tested, and then the software is released [Arthur 88].

Why is impact analysis so important?

Impact analysis is important because it is crucial to fully understand the impact a change will have on a software project. Impact analysis also covers the cost of making a change and the cost of not making a change (how much customer satisfaction is lost). Impact analysis provides managers information they need to make a decision on whether a change will be made or not.

2. Introduction to Software Comprehension

Define software comprehension.

Software comprehension is "the process of understanding program code unfamiliar to a programmer" [Koenemann and Robertson 91].

Why is software comprehension important?

Software comprehension is important because it is a major part of software maintenance. Any increase in software comprehension effectiveness would result in a direct increase in software maintenance effectiveness. Less time spent on software comprehension results in more time spent working on productive activities.

Give the equation that describes software comprehension and describe its terms and variables.

$M = p + K^{(c-d)}$ *where p is productive effort and the second term is nonproductive work, or "wheel turning" [Belady and Lehman 72] [Pressman 05]. K is a constant, c is a measure of complexity due to poor design and documentation, and d is a measure of familiarity.*

Describe text structure comprehension.

Text structure comprehension is rooted in classical text comprehension from psychology [Pennington 87]. The theory states that as programmers read computer code, they recognize various knowledge structures which include specific domain knowledge and recognized code structures. Programmers then make a mental map of the code.

Describe the difference between bottom-up and top-down comprehension.

In top-down comprehension, the programmer creates a general hypothesis about how a section of code works then goes deeper into the code to prove or disprove that hypothesis. In bottom-up comprehension, the programmer starts at the lowest level then does a line-by-line analysis of the code.

3. Introduction to Software Documents

What is a software document?

A software document is any document produced during the development of a software project. It includes design documents, test plans, requirements documents, code inspection records, and any other document generated.

What is an abstraction?

An abstraction is a simplification to a form that is easier to understand.

What is a dataflow diagram? What are some of its advantages and disadvantages?

A dataflow diagram is an abstraction that shows how data in a program/module is transformed from input to output. It is easy to understand, but it is difficult to use for programs with complex I/O interfaces [Sommerville 04].

What is an object model? What are some of its advantages and disadvantages?

An object model is the overall system turned into a set of objects with defined interfaces. Object models are similar to the idea of object-oriented programming, and in fact model OOP quite well. An advantage this abstraction shares with OOP is loose coupling: any object can change without affecting the others as long as the interface doesn't change. However, a major disadvantage of object models is that interface changes usually require an extensive impact analysis to discover everything else that must also be changed as a result [Sommerville 04].

What is UML? What are some of its advantages and disadvantages?

UML is a powerful modeling language that can be used to abstract different activities. UML is usually used in conjunction with a software tool. Changing a design methodology may require the purchase of a new UML tool since the tools are methodology specific [OMG 05].

4. Software Documents

What is a flowchart? What are some of its advantages and disadvantages?

Flowcharts are one of the oldest program abstractions. A flowchart is a set of shapes and lines that represents the control flow of a program. An advantage is that flowcharts are very easy to read since most programmers are trained in how to use them in college. Unfortunately, flowcharts are generally bulk and dataflow insensitive.

Describe the two levels of requirements.

User requirements are the highest level requirements abstractions and usually are in a natural language. These requirements detail what the software is supposed to do and the conditions under which it must operate.

System requirements are the low level requirements. These requirements can serve as a contract between a developer and the customer.

What is the difference between functional and nonfunctional requirements?

Functional requirements are requirements that the software must deliver. For example, " an entry must be created in a database table when the user completes input" would be an example of a functional requirement. Nonfunctional requirements cover things that are not directly related to what the software must deliver. For example, performance requirements are nonfunctional requirements.

Why are requirements documents useful?

Requirements documents are useful because they document how the software is to behave. Requirements documents for past versions of software are also useful because usually the old functionality that is still in the software is not detailed in the requirements document for the current version. Also even after maintenance changes, the software still must behave as the requirements document says it should except maybe when there is a major perfective change.

What is a design document?

A design document is usually heavily based on the requirements document. It contains abstractions that describe how the software is going to implement the system requirements.

5. Introduction to Graph Theory and Measuring Software Comprehensibility

What is an acyclic graph?

An acyclic graph is a graph with no cycles, e.g., a tree or a DAG.

What is a DAG?

A DAG is a directed acyclic graph. Its edges are directed, and the graph has no cycles. It is different from a tree in that some nodes may have more than one predecessor.

Describe the formula for the cyclomatic number or nullity of a graph.

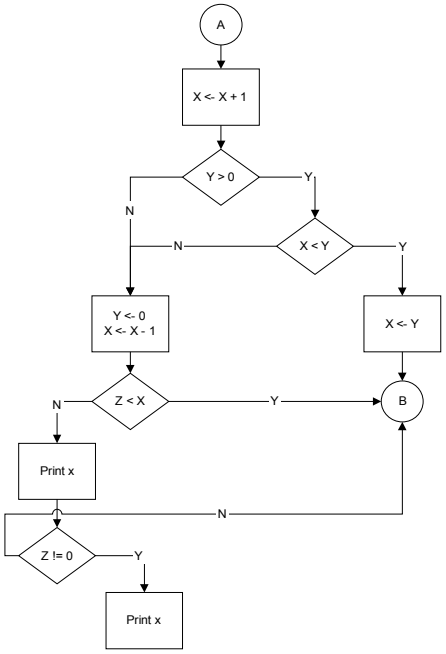
$C = e - n + p$, where C is the cyclomatic number, e is the number of edges, n is the number of nodes, and C is the cyclomatic number, and p is the number of connected components.

What is the formula for cyclomatic complexity?

CC = C + 1 or CC = e – n + 2p

Draw a flowgraph for the following pseudocode. Assume all variables have been initialized.

```
1 A:
2 X = X + 1
3 If Y > 0
4   If X < Y
5       X = Y
6       Goto B:
7   End if
8 Else
9   Y = 0
10  X = X – 1
11  If Z < X
12      Goto B:
13  Else
14      Print X
15      If Z != 0
16          Print X
17          Goto A:
18      End if
19  End if
20 B:
21 End
```



What is the cyclomatic complexity of the pseudocode in the last question?

CC = 14 – 11 + 2 = 5

Given the following complexities of the methods in a class, what is the WMC? Why is WMC a good metric for object-oriented programs?

Constructor = 5
Print = 3
Compute = 10
Destructor = 2
Method1 = 7

WMC = 27. WMC is a good metric for object-oriented programs because it is biased to be larger for methods with a larger number of methods. Intuitively, a class with many methods should be harder to understand than a class with a few large methods.

Describe the depth of inheritance tree and the number of children metrics and how they are useful.

Depth of inheritance is the number of classes in a maximum length path from the root of the inheritance tree to the current class. Number of children is the number of direct child classes. Both are important for integration testing. They count the integration interfaces that must be tested with the current class. Also, the current class needs to be retested any time any of those interfacing classes change.

Describe the lack of cohesion metric.

LCOM measures the methods' usage of instance, class-level variables. Intuitively, if the methods in a class share its instance variables, the class is more cohesive and easier to understand.

6. Introduction to Software Testing

What is the difference between verification and validation?

Verification means making sure that software meets its requirements. Validation means making sure that software is free of defects.

Describe unit testing.

Unit testing tests individual software components.

Describe module testing.

Module testing is the testing of a group of related units. It is done to test how a group of components work together.

Describe sub-system testing.

Sub-system testing is done to test the interaction between various modules, specifically the interaction with respect to their interfaces.

Describe system testing.

System testing is done to test the operation of the whole system. System testing is concerned with the software operating according to its requirements and also testing the interfaces between various subsystems. This sort of testing is also called integration testing.

7. Planning for Successful Testing

What is the main goal of a test plan?

The main goal of a test plan is to document the standards for the software testing process. Tests used are documented, but they are not the main focus of the test plan.

What is requirements traceability and why is it important?

Requirements traceability is the ability to trace requirements to specific tests. Requirements traceability is important because it is used to make sure each and every requirement is covered by the software and works as is specified.

Why are resources listed in a test plan?

Resources are listed so that everything required for the testing to be successful can be planned for and will be present when testing begins. Late delivery of a key resource could mean that testing is not likely to finish in the required time.

8 The Testing Toolbox

What is software inspection?

Software inspection is a type of testing that is carried out by software developers. In software inspection, a code change is looked at by a group of software developers knowledgeable about the change. In code inspection, it is important to note that the change is what is being inspected, not the programmer who made the change.

Name the advantages of software inspections.

Software inspection can find many defects in a single inspection. Conducting inspections is an inexpensive way to find software defects. Inspections reuse the domain and programming language knowledge of the software developers. Inspectors generally know what sort of errors are common to the problem the software change is trying to solve.

Why should inspection testing not replace system testing?

Inspectors cannot find all software defects or verify that all requirements have been covered. The thorough and exacting nature of system tests ensure that most software defects or requirements compliance issues will be found if the tests are designed correctly. Inspections compliment system testing, they do not replace it.

What is black box testing?

Black box testing is testing a piece of code without regard to the internal workings of that code. The outputs are compared to what they should be for a specific set of inputs. For example, a method can be tested by sending it parameters and then comparing the return value of that method to the expected return value.

What is equivalence partitioning and why is it important to software testing?

Equivalence partitioning is the idea of grouping input values together based on an equivalence function. All members of each set would behave similarly. As a result, only one input value from each set (equivalence class or partition) needs to be tested. Equivalence partitioning is important because it allows a tester to limit the input values being fed into a test. Also, it allows a tester to know that all possible types of input have been tested.

9. The Testing Toolbox, part 2

What is white box testing?

White box testing is similar to black box testing, but the internal structure of the code is taken into account when selecting test inputs. By using this knowledge of the code, a tester can select specific input values or equivalence classes which the tester believes the code would have difficulty processing.

What is static analysis?

Static analysis is examining the syntax or surface structure of a program to determine its quality or complexity. Static analysis can be done manually or with the aid of static analysis tools. Static analysis also refers to scrutinizing the test of a program

for possible sources of faults such as missing declarations, incorrect scooping, and variable redefinitions.

What is the main challenge of static analysis?

Static analysis tools generally produce voluminous output, leading to quite a bit of time just to interpret the results of the analysis.

What is the main goal of path testing?

Path testing uses McCabe's cyclomatic complexity [McCabe 76]. Path testing is concerned with calculating a test coverage metric that shows the test coverage of the code. Path testing usually uses a software tool to discover the flowgraph of a program. Then, the testers can designate specific branches to test.

10. The Testing Toolbox, part 3

What is integration testing?

Integration testing is testing to see how the smaller parts of a program come together. Integration testing usually looks at the interfaces between smaller parts of a program. A challenge with integration testing is that sometimes it is difficult to find where a defect is originating.

Describe the two different methods of integration testing.

Bottom-up integration testing starts with testing the interfaces of the smallest components and moves up until the entire program is integrated. Large parts of the program must have been finished to run a bottom-up integration test.

Top-down integration testing starts the integration testing with the overall program. Smaller parts of the program that are not complete yet must be simulated with stubs, which are test programs or very simplified version of the final components that have the same interface as the smaller incomplete parts of the program.

What are the main strengths and weaknesses of top-down integration testing?

Top-down integration testing is very good at discovering defects in the architecture and high-level design. Top-down integrations tests can be run much earlier in the software development process because stubs can be created to represent incomplete parts of the program. Finding software defects earlier decreases cost, however the creation of the stubs does require some effort.

What is the main disadvantage both types of integration tests, bottom-up and top-down, have in common?

Programs are usually not designed to allow easy viewing of the return values of parts of the program, which means some sort of artificial testing environment must be created.

What are the general guidelines to use in interface testing?

1. Test the boundary values of the ranges of input.
2. When pointers are used, test passing a null parameter.
3. When a component is called through a procedure, design a test that should cause the component to fail.
4. If the program is in a message passing system, generate a test that generates far more messages than are expected.
5. When shared memory is used, design a test to change the order in which components access that shared memory

Why is "how the software behaves in a stress test" important?

Software must be able to fail gracefully. Programs should not cause a user to lose data when they fail. Stress testing will cause failures and is thus a good mechanism to test the failure behavior of software.

VITA

Milton A. Austin III

Candidate for the Degree of

Master of Science

Thesis: DEVELOPING INSTRUCTIONAL MATERIAL FOR SOFTWARE COMPREHENSION/MAINTENANCE

Major Field: Computer Science

Biographical:

Education: Graduated as a valedictorian from Mustang High School, Mustang, Oklahoma; in May 1995; received Bachelor of Science in Computer Science from Oklahoma Christian University, Oklahoma City, Oklahoma with magna cum laude honors in April 1999; completed the requirements for Master of Science at the Computer Science Department at Oklahoma State University in May 2006.

Experience: Employed from 1998 - 2001 as a engineering assistant, development engineer, development engineer advisory at Seagate Technology doing software engineering support on low and high end SCSI hard disk drives. Employed since May 2005 as a software engineer journeyman with Techrizon (formerly TELOS•OK, LLC, TELOS, TELOS Corporation, TELOS Federal Systems) doing software engineering support/project management on tactical Fire Support Systems for the US Army. The systems the author works with are: Meteorological Measuring Set (MMS), Forward Observer System (FOS), and Firefinder Radar (FF-EU and FF-Q37).

Name: Milton A. Austin III Date of Degree: May 2006

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: DEVELOPING INSTRUCTIONAL MATERIAL FOR TEACHING SOFTWARE COMPREHENSION/MAINTENANCE

Pages in Study: 72 Candidate for the Degree of Master of Science

Major Field: Computer Science

Scope and Method of Study: Software maintenance is a costly problem for industry, typically taking up to 50-75% of the cost of software development [Sommerville 04]. Traditional Computer Science programs often do not prepare students to face this problem. Since a large part of software maintenance is software comprehension, better comprehension methods are a major part of the answer to the problem. Students often do not know how to comprehend already written code and do not know how to work in groups, hence new graduates are typically forced to learn the very important skill of software comprehension on the job. It is proposed that students should be taught a standardized way of software comprehension in preparation for the software maintenance jobs most will have.

Findings and conclusions: The comprehension/maintenance area of computer science education has not been extensively covered as a research topic. This work is a detailed proposal for a software maintenance course, using techniques utilized by other researchers in their efforts to teach software maintenance as well as the new idea of teaching software comprehension techniques in a required course. This work outlines a course that will be designed to better prepare students for work in the area of software maintenance by teaching them software comprehension methods. The course includes best practices, a large-scale project, and focuses primarily on code comprehension methods. This course represents a standardized way to teach students software comprehension skills that are needed in the industry.

ADVISOR'S APPROVAL: M. H. Samadzadeh