# **Imprinting Community College Computer Science Education with Software Engineering Principles**

by

Jacqueline Holliday Hundley

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 7, 2012

Keywords: computer science, software engineering, CS1/CS2,
curriculum, higher education, community college

Copyright 2012 by Jacqueline Holliday Hundley

Approved by

David Umpress, Chair, Associate Professor of Computer Science and Software Engineering
James Cross, Professor of Computer Science and Software Engineering
Dean Hendrix, Associate Professor of Computer Science and Software Engineering

Abstract

Although the two-year curriculum guide includes coverage of all eight software engineering core topics, the computer science courses taught in Alabama community colleges limit student exposure to the programming, or coding, phase of the software development lifecycle and offer little experience in requirements analysis, design, testing, and maintenance. We proposed that some software engineering principles can be incorporated into the introductory-level of the computer science curriculum. Our vision is to give community college students a broader exposure to the software development lifecycle. For those students who plan to transfer to a baccalaureate program subsequent to their community college education, our vision is to prepare them sufficiently to move seamlessly into mainstream computer science and software engineering degrees. For those students who plan to move from the community college to a programming career, our vision is to equip them with the foundational knowledge and skills required by the software industry.

To accomplish our goals, we developed curriculum modules for teaching seven of the software engineering knowledge areas within current computer science introductory-level courses. Each module was designed to be self-supported with suggested learning objectives, teaching outline, software tool support, teaching activities, and other material to assist the instructor in using it.

Acknowledgements

I would like to take this opportunity to express my sincere appreciation to my family for their support and encouragement through my pursuit of this PhD degree.

I offer a special thanks to my advisory committee chair, Dr. David A. Umphress, for his guidance and encouragement throughout the development of this research. I thank my committee members, Dr. James H. Cross and Dr. Dean Hendrix for their assistance and input into this thesis work.

I want to extand an appreciation to the graduate students in the Teaching Software Engineering course for their contributions during the class and participation in the faculty workshop.

Table of Contents


Abstract .......... ii

Acknowledgements .......... iii

List of Tables .......... vii

List of Figures .......... ix

List of Acronyms .......... x

1 Introduction .......... 1

2 Background .......... 6

2.1 Computing .......... 6

2.2 Software Engineering Principles [Pressman 2010] .......... 9

2.3 Industry and Software Engineering .......... 15

2.4 Curriculum Guidelines .......... 19

2.4.1 Computing Curricula 2001: Computer Science [Chang, et al. 2001] .......... 19

2.4.2 Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science [Campbell, R. (chair) et al. 2003] .......... 21

2.4.3 Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering [Le Blance and Sobel 2004] .......... 24

2.4.4 Computing Curricula 2005: Guidelines for Associate-Degree Transfer Curriculum in Software Engineering [Campbell, et al. 2005] .......... 25

2.4.5 Computing Curricula 2005: Overview Report on Computing Curricula [Shackelford, et al. 2005] .......... 26

2.4.6 Computer Science Curriculum 2008: An Interim Revision of the CS 2001 [McCauley and McGettrick 2008] .......... 28

2.4.7 Computing Curricula 2009: Guidelines for Associate-Degree Transfer Curriculum in Computer Science [Hawthorne, et al 2009] .......... 29

2.4.8 The Guide to the Software Engineering Body of Knowledge [Tripp, et al. 2004] .......... 33

2.5 Software Development Tools ........ 34
2.5.1 Professional Integrated Design Environment ........ 36
2.5.2 Pedagogical Integrated Design Environments ........ 39
2.5.3 Microworlds ........ 43
3 Community Colleges ........ 46
3.1 Higher Education and Community College Demographics ........ 47
3.2 STEM in Community Colleges ........ 48
3.3 Alabama Community College System ........ 49
3.4 Alabama Articulation and General Studies Committee ........ 50
3.5 Alabama Community College Computer Science Curricula ........ 52
4 SIGCSE 2011 Birds-of-a-Feather: Introducing Software Engineering Principles in the First Two Years of Computer Science Education ........ 59
5 Survey of Software Engineering Principles and Concepts ........ 63
5.1 Survey Results ........ 64
5.1.1 Software Engineering Knowledge Area Results ........ 64
5.1.2 Integrated Development Environment and Programming Language Results ........ 71
5.1.3 Other Results ........ 71
5.2 Survey Results Summary ........ 72
6 Teaching Software Engineering Course ........ 75
7 Teaching Software Engineering Principles in Introductory Computer Sciences Courses Workshop ........ 77
8 Curriculum modules ........ 79
8.1 Software Process Curriculum Module ........ 80
9 Conclusion and Future Work ........ 96
9.1 Summary of Research ........ 96
9.2 Future work ........ 100
References ........ 102
Appendix A ........ 110

SWEBOK Software Engineering Knowledge Areas (KAs) .... 111
Bloom's Taxonomy Levels .... 115
Appendix B .... 116
Alabama Public Community Colleges’ Reference Information. .... 117
Alabama Public 4-year Universities’ Reference Information. .... 122
Appendix C .... 124
Survey of Usage of Software Engineering Principles and Concepts .... 125
Appendix D .... 133
Teaching Software Engineering .... 134
Appendix E .... 138
Software Process Curriculum Module .... 139
Software Testing Curriculum Module .... 151
Software Construction Curriculum Module .... 162
Software Design Curriculum Module .... 169
Software Quality Curriculum Module .... 189
Software Requirements Engineering Curriculum Module .... 197
Software Configuration Management Curriculum Module .... 207

List of Tables

Table 2.1 General principles of software engineering [Hooker 1996] ........ 9

Table 2.2 Core principles that guide process [Pressman 2010] ........ 10

Table 2.3 Core principles that guide practice [Pressman 2010] ........ 10

Table 2.4 Communication principles [Pressman 2010] ........ 11

Table 2.5 Planning principles [Pressman 2010] ........ 11

Table 2.6 Modeling principles [Ambler and Jefferies 2002] ........ 12

Table 2.7 Operational principles [Pressman 2010] ........ 12

Table 2.8 Design principles [Pressman 2010] ........ 13

Table 2.9 Coding principles [Pressman 2010] ........ 13

Table 2.10 Testing principles [Pressman 2010] ........ 14

Table 2.11 Deployment principles [Davis 1995] ........ 15

Table 2.12 Software engineering technical skills ........ 17

Table 2.13 Software engineering soft skills ........ 19

Table 2.14 Software engineering core topics and coverage hours ........ 20

Table 2.15 Software engineering unit coverage hours in CS2001 introductory tracks ........ 22

Table 2.16 Software engineering unit coverage hours in CC2003 [Campbell, et al. 2003] ........ 23

Table 2.17 Computer science associate-degree program outcomes [Hawthorne 2009] ........ 30

Table 2.18 Program outcomes and supporting coursework [Hawthorne 2009] ........ 31

Table 2.19 Computer science sequence topics [Hawthorne 2009] ........ 32

Table 2.20. The SWEBOK Knowledge Areas ........ 34

Table 2.21. Core capabilities of rational products and services [Rational 2009] ........ 38

Table 2.22. Software engineering curriculum and software development environments .......................... 45

Table 3.1. Percentages of postsecondary enrollment increase .................................................................. 46

Table 3.2. Alabama new undergraduate transfers summary ...................................................................... 47

Table 3.3. CIS courses in community college catalogs related to software engineering ........................... 54

Table 3.4. Concepts, Techniques, and Requirements in CIS Course Numbers and Descriptions .............. 56

Table 3.5. Area V required CIS courses for transfer to four-year university ............................................ 58

Table 5.1. Results of Wilcoxon signed rank test (*p* values) ....................................................................... 74

List of Figures

Figure 2.1 Computer Science [Shackelford, et al. 2005]. 27

Figure 2.2 Software Engineering [Shackelford, et al. 2005]. 27

Figure 4.1. SIGCSE 2011 Birds-of-a-feather small group results 61

Figure 4.2. SIGCSE 2011 Birds-of-a-feather large group discussion 62

Figure 5.1. Software engineering knowledge areas included in the survey and education objectives in all programs 65

Figure 5.2. Software engineering knowledge areas included in the survey and education objectives for the programs that teach the principle 66

Figure 5.3. Software engineering terms and concepts included in the survey and education objectives in all programs 67

Figure 5.4. Software engineering terms and concepts included in the survey and education objectives in programs that teach the concept 68

Figure 5.5. Integrated development environments (IDEs) used in respondents' programs 69

Figure 5.6. Computer programming languages taught in respondents' programs 70

Figure 5.7. Where do two-year graduates go 71

Figure 5.8. Other jobs for two-year graduates 71

Figure 5.9. Two-year respondents familiarity with curriculum guides 72

Figure 5.10. Four-year respondents familiarity with curriculum guides 72

Figure 8.1-15. Software process curriculum module 81-95

## List of Acronyms

| | |
|---|---|
| AACC | American Association of Community Colleges |
| ACCS | Alabama Community College System |
| ACHE | Alabama Commission on Higher Education |
| ACHE | Alabama Commission on Higher Education |
| ACM | Association for Computing Machinery |
| ACMTYC | ACM Two-Year College Education Committee |
| ADPE | Alabama Department of Postsecondary Education |
| AGSC | Alabama Articulation and General Studies Committee |
| API | Application Programming Interface |
| CASE | Computer-Aided Software Engineering |
| CC2003 | *Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science* [Campbell, R. (chair) et al. 2003] |
| CC2005 | *Computing Curricula 2005: Guidelines for Associate-Degree Transfer Curriculum in Software Engineering* [Campbell, et al. 2005] |
| CC2009 | *Computing Curricula 2009: Guidelines for Associate-Degree Transfer Curriculum in Computer Science* [Hawthorne, et al 2009] |
| CIS | Computer Information Science |
| CS2001 | *Computing Curricula 2001: Computer Science* [Chang, et al. 2001] |
| CS2008 | *Computer Science Curriculum 2008: An Interim Revision of the CS 2001* [McCauley and McGettrick 2008] |
| CSD | Control Structure Diagram |
| CVS | Concurrent Versions System |
| GAO | U.S. Government Accountability Office |
| GSAC | [AGSC] General Studies Academic Committees |
| IDE | Integrated Design Environment |
| IEEE | Institute for Electrical and Electronic Engineers |
| IEEE-CS | Computer Society of the Institute for Electrical and Electronic Engineers |
| KA | Knowledge Area |

| | |
|---|---|
| PAC | [AGSC] Pre-Professional Academic Committees |
| SE2004 | *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* [Le Blanc, et al. 2004] |
| SEEK | Software Engineering Education Knowledge |
| SPAC | State Planning Advisory Council, Alabama Commission on Higher Education |
| STARS | [Alabama] Statewide Transfer and Articulation Reporting System |
| STEM | Science, Technology, Engineering, and Mathematics |
| SWEBOK | *The Guide to the Software Engineering Body of Knowledge* [Trip, et al. 2004] |
| SWECC | Software Engineering Coordinating Committee |
| UML | Unified Modeling Language |

# 1Introduction

Every year, software disasters cost the United States billions of dollars. Statistics indicated that 40-50% of programs contain nontrivial failures [Stiller and LeBlance 2002, Boehm 2006]. In many cases, the causes of failure originate with a misunderstanding of requirements; mismatches in system design and implementation; overly ambitious development and implementation plans; unrealistic expectations; bad project planning; and indecisive customers [Pfleeger 1998, Burgess 1995]. Other failures of software projects can be linked to the lack of version control, thorough unit testing, or proper monitoring of daily progress and activities [Hunt and Thomas 2004].

The failures noted above imply a lack of an adequate procedure to assess the problem and design the solution. Software engineering strives "to deliver on-time, high-quality, operational software that contains functions and features that meet the needs of all stakeholders." Guidelines are needed to successfully produce a software product of this caliber. During the past fifty years, the principles of the software engineering discipline have been developed and provide guidelines for a solid approach to software engineering. [Pressman 2010]

In the computer science curriculum, students receive considerable experience in the programming, or coding, phase of the software lifecycle [Pressman 2010]. Their projects are usually limited to small problems in which there is little need for requirements analysis, design, testing, and maintenance [Myers 2000]. Students are taught to write computer programs, but few can develop large software systems [Long 2008]. In contrast, industry needs software engineers equipped with skills that go

well beyond the coding activity. It needs engineers that can use procedures, paradigms, tools, and techniques to produce quality software products [Pfleege and Altee 2006].

The Joint Task Force on Computing Curricula of the Computer Society of the Institute for Electrical and Electronic Engineers (IEEE-CS) and the Association for Computing Machinery (ACM) joint task force produced *Computing Curricula 2001: Computer Science* [Chang, et al. 2001] to outline curricular guidelines for undergraduate programs in computer science. Additional volumes of this report present undergraduate curricula for specific disciplines including *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* [Le Blanc, et al. 2004], which presents a set of curriculum recommendations for baccalaureate software engineering programs. Because of rapid changes in the field of computer science, an ACM and IEEE-CS Interim Review Task Force was formed to conduct an interim review of CS2001. The document resulting from the review is *Computer Science Curriculum 2008: An Interim Revision of the CS 2001* [McCauley and McGettrick 2008].

Curricula guides are also available for associate-degree programs designed for students planning to transfer into baccalaureate programs. *Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science* [Campbell, et al. 2003] and *Computing Curricula 2005: Guidelines for Associate-Degree Transfer Curriculum in Software Engineering* [Campbell, et al. 2005] were developed by the ACM Two-Year College Education Committee and Joint Task Force on Computing Curricula of IEEE-CS and ACM. In 2009, the ACM Two-Year College Education Committee (ACMTYC) developed *Computing Curricula 2009: Guidelines for Associate-Degree Transfer Curriculum in Computer Science* [Hawthorne, et al. 2009].

The existence of software engineering curriculum guidelines reinforces the need for teaching software engineering principles in two- and four-year undergraduate programs. The problem with adding a software engineering curriculum to two- and four-year computer science programs is resources.

Because there is overlap of material in computer science and software engineering curriculum guidelines, it may be possible to include software engineering in an existing computer science program with little or no additional resources.

The use of software tools and programming environments can enhance the teaching and learning of software engineering and computer science skills and principles. There are many programming tools available for teaching introductory-level computer science courses. Professional integrated development environments (IDEs) support the teaching of software engineering principles in all phases of the software development life cycle. However, many instructors feel that the number of features in professional level IDEs is a distraction for students, the learning curve is too steep, and they are too costly. Other IDEs are designed specifically for pedagogical purposes. The disadvantage of these is their application is limited to one or a few phases of the software lifecycle. [Burch 2009, Chen and Marx 2005]

Community colleges' open-door admission policies, reduced costs, convenient campus locations, and comprehensive course offerings offer a diverse population of students an alternative to the traditional four-year universities. Over the past 40 years, public community college enrollment has increased at a much faster rate than at the public four-year universities, with the percentage of women enrolled in community colleges surpassing that of men. Because of the low cost and accessibility, racial and ethnic minorities have become an increasing proportion of all students enrolled at community colleges [Kasper 2002]. In the time of a recession, community colleges experience an abnormal increase in student enrollment as unemployed workers seek to continue their education or change career fields [Tirrell-Wysocki 2009].

The Computer Information Science (CIS) courses listed in the computer science programs of the Alabama public community college catalogs use the same course numbering system; however, the same course number represents multiple course names and descriptions in various catalogs. In short, students are not guaranteed that courses with the same number and title will convey the same computer science

concepts, much less complementary engineering concepts. This situation also hinders a smooth transfer from community college program to the junior year of a baccalaureate program at a four-year university.

We proposed that some software engineering principles can be incorporated into the introductory-level of the computer science curriculum. Our vision is to give community college students a broader exposure to the software development lifecycle. For those students who plan to transfer to a baccalaureate program subsequent to their community college education, our vision is to prepare them sufficiently to move seamlessly into mainstream computer science and software engineering degrees. For those students who plan to move from the community college to a programming career, our vision is to equip them with the foundational knowledge and skills required by the software industry.

By placing our emphasis on the Alabama public community college system, we sought a broader impact by taking our efforts to the institutions that provide open and low cost education to those who are underrepresented in the general computer science student population in higher education. This research offers further the exposure of women and minorities to STEM areas of study. This research, also, assists with articulation between the two- and four-year public institutions in the region.

This research seeks intellectual merit through emphasizing the incorporation of software engineering principles required by industry into the introductory-level of curriculum where this knowledge can mature and better benefit the students throughout the curriculum and into the work force. It provides guidance to the faculty of the community colleges through application of this educational experience.

In the following chapters, we provide information relating to the examination of the background material and the proposed goals and plans for this research. Chapter 2 provides discussions on the computing field of study, the purpose of using software engineering principles, the need for software engineering knowledge in industry, the curriculum guides for computer science and software engineering, and the software development tools. Chapter 3 presents the influence of community colleges

in higher education and discusses the Alabama community college system and curriculum. In chapter 4, we presents ideas collected from faculty from two- and four-year computer science programs while in a SIGCSE 2011 Birds-of-a-Feather session. This discussion assisted in determining topics that were later included in a online survey. The survey was used to collect information about which software engineering principles and concepts are currently being taught in the Alabama public colleges and universities. The survey findings are reported in Chapter 5, and a copy of the survey is presented in Appendix C. In Chapter 6, we describe a special topics graduate course, Teaching Software Engineering, which examined software engineering from an instructional perspective. During this course, students were exposed to explaining fundamental software engineering concept to those new to the field. The information collected during the literature search, SIGCSE 2011 Birds-of-a-Feather session, and special topics course assisted the creation of curriculum modules for teaching software engineering at the introductory-level. Chapter 5 presents the Software Process Curriculum Module as a sample teaching modules. All of the modules can be found in Appendix D. At the end of the Teaching Software Engineering course, a faculty workshop was held to present the teaching to modules to local area faculty who teach introductory-level computer science. During the workshop, attendees were asked to evaluate the modules. The result of the evaluations is presented in Chapter 7. In Chapter 8 this research is A summary of this research is presented in Chapter 8 as well as ideas for further work and studies that will enhanced and continue this research.

# 2Background

## 2.1Computing

Computing is one of the most significant advancements of the twentieth century. It is a product of human ingenuity and provides unlimited intellectual challenges. Computing promotes innovations and creativity that require a disciplined approach to problem solving. Although its important components are invisible to the naked eye, computing has been applied to a diverse range of applications and has become a significant part of everyday life. [QAA 2000]

The Association for Computing Machinery (ACM) categorizes the computing discipline into five sub-disciplines: computer science, information systems, software engineering, computer engineering, and information technology [Hawthorne, et al. 2009]. The activities of these topics are often misunderstood and the names misused. Titles such as programmer, computer scientist, system analyst, and software engineer are often used for positions with the same job description. This confusion is reflected in academia, where there is little consistency in naming departments, curricula, and courses. Many lay people do not understand the field of computing, and students chose to study computing without fully understanding what it is. Computer science is a valid field of study, but does it provide the graduates with the knowledge and skill that are needed in industry?

Computer science tends to be the generic term for computing [Vaughn 2000]. Many consider programming to be the core of computer science; however, it is only one of four core practices of computer science along with systems thinking, modeling, and innovation [Denning 2004].

Computer science offers a comprehensive foundation that allows graduates to adapt to new technologies and new ideas. Through its theoretical and algorithmic foundations, computer science includes the developments of robotics, computer vision, intelligent systems, bioinformatics, and many other areas. The work of computer scientists can be divided into three categories: (1) designing and implementing software (programmers who keep up with new approaches), (2) devising new ways to use computers (researchers who work with other scientists to develop practical and intelligent robots, to use databases to create new knowledge, or to use computers to help decipher DNA), and (3) developing effective ways to solve computing problems (determine the best performance possible and develop new approaches that provide better performance). [Shackelford, et al. 2005]

In the computer science curriculum, students receive considerable experience in the programming, or coding, phase of the software lifecycle [Pressman 2010]. Their projects are usually limited small “systems” offering little experience in requirements analysis, design, testing, and maintenance [Myers 2000]. Many graduates can write computer programs, but few can develop large software systems [Long 2008].

The introduction of integrated circuit computers enabled software to be much larger and more complex causing what some called a “software crisis” [Sommerville 2004]. Software systems continue to grow quickly in size and complexity with their success becoming a factor of lives as well as economics. To produce these systems, procedures and guidelines are needed to translate requirements into working systems, to assess and manage risk, to systematically locate and eliminate errors, to organize and manage development teams, and to satisfy customers. In 1968, the recognition of an engineering process led to a new discipline, software engineering. [Denning 2004]

“Engineering disciplines are concerned with the construction of devices that can be relied upon to perform a function. …[A]n engineer approaches a design task with a collection of techniques, tools, and previous designs which make it possible to create reasonably reliable devices at reasonable cost with

a reasonable amount of effort." [Richard Karp in Denning 1989] Software engineering refers to the discipline application of engineering, scientific, and mathematical principles to the economical production of quality software [Gibbs 1989].

Software engineering is the part of computing concerned with a system view of software and all phases of the software lifecycle as well as general design and architecture issues of the system that contains the software [Werth and Werth 1991]. Software engineers aim to develop and maintain software systems that (1) behave reliably and efficiently; (2) are affordable; and (3) satisfy all the customers' requirements [Shackelford, et al. 2005]. They build on and apply the body of knowledge found in the computer science curriculum; but they also need additional education in some or all of the following areas: software development process, software project management, requirements analysis, technical communication, computer engineering, systems engineering, embedded and real-time systems, configuration management, quality assurance, formality, performance analysis, metrics, standards, verification and validation, security, human factors and specialized applications domains. [Gibbs 1989]

The body of knowledge needed in software engineering programs differs from that of computer science programs. The goal of software engineering programs should be that of an engineer constructing a useful artifact rather than that of a scientist discovering or refining new knowledge. [Mitchell 2004] Brooks [1995] puts it this way: "The scientist builds to learn; the engineer learns in order to build."

Computer science is a science [Werth and Werth 1991, Mitchell 2004, Mead, et. al. 2000], meaning it is concerned with the underlying theories and methods of computers and software systems; software engineering is concerned with the practical problems of producing software. Real, complex problems often require more than the theories of computer science. There has been concern in industry and some application areas that the gap between academic computer science and the actual needs of industry has become too great; and computer science programs do not provide the fundamental knowledge needed for long-term professional growth. [Lewis 1989, Parnas 1990]

## 2.2Software Engineering Principles [Pressman 2010]

During the past fifty years, much knowledge has been collected from experience gained through the observation of thousands of software projects. From this knowledge, the principles of the software engineering discipline have been developed providing guidelines for a solid approach to software engineering. Some may view principles as common sense; however, documenting them allows all software project development teams to benefit from the knowledge of others. Roger Pressman included a comprehensive list of software engineering principles in the seventh edition of *Software Engineering: A Practitioner's Approach* [Pressman 2010]. This list is used in the following discussion of the software engineering principles.

Software engineering principles provide guidance at different levels of abstraction. Some, as shown in Table 2.1, focus on software engineering as a whole, and others focus on a general framework activity or specific actions and tasks. The generality of the principles presented in Tables 2.1, 2.2, and 2.3 shows how the development of good software begins long before where traditional computer science courses start, coding.

**Table 2.1 General principles of software engineering [Hooker 1996]**

1. Remember that a software system exists to provide value to its user.
2. Keep designs as simple as possible but no simpler.
3. Attain a clear vision to ensure the success of a software project.
4. Always specify, design, and implement knowing someone else will have to understand what you are doing.
5. Never design yourself into a corner; consider the "what if."
6. Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the system into which they are incorporated.
7. Placing clear, complete thought before action almost always produces better results.

The primary levels of core principles guide the application of software process and the execution of effective software. Core principles, shown in Table 2.2, can be applied to any type software process model: linear or iterative, prescriptive or agile. They help establish a philosophy that will guide a software team through the activities necessary to produce a working software product.

**Table 2.2 Core principles that guide process [Pressman 2010]**

1. Be agile. Emphasize economy at every step.
2. Focus on quality at every step.
3. Be ready to adapt.
4. Build an effective team.
5. Establish mechanisms for communication and coordination.
6. Manage change.
7. Assess risk.
8. Create work products that provide value for others.
9. Use an appropriate process model [Davis 1995].

At the practice level, there are core principles that guide the technical work. These principles apply regardless of the analysis and design methods; construction techniques; or verification and validation approach used. The core principles establish rules and values that will guide the problem analysis; solution design, implementation and testing; and product deployment. Table 2.3 lists a set of core principles that are fundamental to the practice of software engineering.

**Table 2.3 Core principles that guide practice [Pressman 2010]**

1. Divide and conquer, or separation of concerns.
2. Understand the use of abstraction.
3. Strive for consistency.
4. Focus on the transfer of information.
5. Build software that exhibits effective modularity
6. Look for patterns.
7. Represent the problem and its solution from a number of different perspectives, when possible.

A refinement of the general process and practice principles guide generic framework activities of the software process. The following tables present the core principles at a lower level of abstraction for each framework activity. These are not exhaustive lists of software engineering principles, but they are a good representation of the type of principles that are necessary to achieve quality products.

Good communication is a top priority for good software development. Knowing a customer's problem may not be obvious. Through the careful communication of technical peers, team, customers,

stakeholders, and project managers, the true problem can be established. Table 2.4 presents some of the principles that apply to communication during a software project.

**Table 2.4 Communication principles [Pressman 2010]**

1. Listen.
2. Prepare before you communicate.
3. Designate an activity facilitator.
4. Communicate face-to-face for best results.
5. Take notes and document decisions.
6. Create a glossary and index [Davis 1995].
7. Strive for collaboration.
8. Stay focused; modularize your discussion.
9. Draw a picture, if something is unclear.
10. (a) Move on, once you agree to something. (b) Move on, if you can't agree to something. (c) Move on, if a feature or function is unclear and cannot be clarified at the moment.
11. Negotiate not compete. It works best when both parties win.

After defining the overall goals and objectives, a plan for how these goals and objects will be met is necessary. To develop an effective plan, everyone on the software team should participate. The plan, or road map, to the solution of a problem is where the real work of software development begins. No matter which approach of planning is used the principles in Table 2.5 apply.

**Table 2.5 Planning principles [Pressman 2010]**

1. Understand the scope of the project.
2. Involve stakeholders in the planning activity.
3. Recognize that planning is iterative.
4. Estimate based on what you know.
5. Consider risk as you define the plan.
6. Be realistic.
7. Adjust granularity as you define the plan.
8. Define how you intend to ensure quality.
9. Describe how you intend to accommodate change.
10. Track the plan frequently and make adjustments as required.

Models are used to ensure the understanding of the requirements and the product to be built. The models must represent (1) the information that the software manipulates, (2) the architecture and functions that manipulate the information, (3) the features and functions that enable the information, (4) the users' desired features, and (5) the behavior of the system. Models should describe software from the

customers' and the technical points of view. In software engineering, there are two classes of models. Requirement models represent the customer requirements in three domains: information, functional, and behavioral. Design models represent the software characteristics: the architecture, the user interface, and component-level detail. Principles for the modeling of actions and tasks presented in Table 2.6 are appropriate for all types of process modeling.

**Table 2.6 Modeling principles [Ambler and Jefferies 2002]**

| |
|---|
| 1. Create models that lead to the primary goal, software.<br>2. Travel light—don't create more models than you need.<br>3. Strive to produce the simplest model that will describe the problem or the software.<br>4. Build models in a way that makes them amenable to change.<br>5. Be able to state an explicit purpose for each model that is created.<br>6. Adapt the models you develop to the system at hand.<br>7. Try to build useful models, but forget about building perfect models.<br>8. Don't become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary.<br>9. Be concerned when your instincts tell you a model isn't right even though it seems okay on paper.<br>10. Get feedback as soon as you can. |

There are a variety of analysis modeling notations and heuristics that have been developed to assist in identifying requirements problems and causes and how to overcome them. Table 2.7 presents a set of operational principles that applies to any analysis method.

**Table 2.7 Operational principles [Pressman 2010]**

| |
|---|
| 1. The information domain of a problem must be represented and understood.<br>2. The functions that the software performs must be defined.<br>3. The behavior of the software (as a consequence of external events) must be represented.<br>4. The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.<br>5. The analysis task should move from essential information toward implementation detail. |

**Table 2.8 Design principles [Pressman 2010]**

1. Evaluate design alternatives [Davis 1995].
2. Design should be traceable to the requirements model.
3. Always consider the architecture of the system to be built.
4. Design of data is an important as design of processing functions.
5. Design interfaces (both internal and external) with care.
6. Adjust user interface design to the needs of the end user. However, in every case it should stress ease of use.
7. Create functionally independent component-level design.
8. Construct components such that they are loosely coupled to one another and the external environment.
9. Design representations (models) should be easily understandable.
10. Develop the design iteratively. With each iteration, the designer should strive for greater simplicity.

**Table 2.9 Coding principles [Pressman 2010]**

Preparation principles

1. Understand the problem you are trying to solve.
2. Understand basic design principles and concepts.
3. Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
4. Use different language for different phases, if needed for optimization of the system [Davis 1995].
5. Select a programming environment that provides tools that will make your work easier.
6. Create a set of unit tests that will be applied once the component is completed.

Programming Principles

1. Constrain your algorithms by following structured programming practice.
2. Consider the use of pair programming.
3. Select data structure that will meet the needs of the design.
4. Understand the software architecture and create interfaces that are consistent with it.
5. Keep conditional logic as simple as possible.
6. Create nested loops in a way that makes them easily testable.
7. Select meaningful variable names and follow other local coding standards.
8. Write code that is self-documenting.
9. Create a visual layout (e.g., indentation and blank lines) that aids understanding.

Validation Principles

1. Conduct a code walkthrough when appropriate.
2. Perform unit tests and correct errors you've uncovered.
3. Refactor the code.

Design models represent an overview of the whole system from a variety of views. They must include both the factors observed by the user and the factors that are important to software engineers. A set of design principles is shown in Table 2.8.

Coding and testing are software construction activities that produce working and deliverable software. Coding may be directly created source code, automatically generated source code using a design component, or automatically generated executable code. Fundamental coding principles, presented in Table 2.9, are associated with all programming style, languages, and methods.

Testing is performed at several levels during software construction: (1) unit testing at the component level, (2) integration testing as more components are added, (3) validation testing to determine if the system meets the requirements, and (4) acceptance testing by the customer. Successful testing will not only uncover errors, but it will demonstrate that the software functions seem to be working according to the specifications and that the behavioral and performance requirements seem to be met. A set of testing principles is listed in Table 2.10.

**Table 2.10 Testing principles [Pressman 2010]**

1. All tests should be traceable to customer requirements.
2. Tests should be planned long before testing begins.
3. The Pareto principle applies to software testing.
4. Testing should begin “in the small” and progress toward testing “in the large.”
5. Exhaustive testing is not possible.

The deployment activity of software development contains three actions: delivery, support, and feedback. Because process models are incremental, there may be several product deployments. Each delivery of a software increment represents an important milestone for a software project and should be accompanied by the appropriate support to enable proper feedback. The feedback is used to modify the functions, features, and approaches before continuing to the next step. Table 2.11 lists key principles that should be followed as the team prepares to deliver and increment.

**Table 2.11 Deployment principles [Davis 1995]**

1. Customer expectations for the software must be managed.
2. A complete delivery package should be assembled and tested.
3. A support regime must be established before the software is delivered.
4. Appropriate instructional materials must be provided to end users.
5. Delivery should be delayed until errors have been corrected. (McConnell 1996).

## 2.3Industry and Software Engineering

With the rapid growth of the software industry, there is an increasing importance that software development results in more products with fewer errors [Tilley and Wong 1993]. Because software systems are becoming larger and more complex, a defined process is essential in ensuring that the true goals, objectives, and requirements of a software system are not missed or misinterpreted. IEEE defines software engineering as "the application of a systematic disciplined, quantifiable approach to the development, operation, and maintenance of software" [Le Blanc, et al. 2004].

To apply the processes and procedures required for successful software development, one needs the appropriate skills. As a result of this literature survey, software engineering skills required by industry for successful software development have been identified. Behind these skills are software engineering principles that guide the tasks and activities. Lists of these technical and soft skills are presented in Tables 2.12 and 2.13 along with an indication of which software engineering principles guide the application of the skill.

Core software engineering principles (Table 2.3) at the general, process, and practice levels provide guidelines for software engineering are skills such as communication, teamwork, ethics, documentation, flexibility, critically thinking, problem solving, abstraction, project management, data management estimation, etc. As the process moves through the lifecycle, the previously mentioned skills continue to be used with more specific ones others added as needed. Communication (Table 2.4), both verbal and visual, continues to play a major part at all phases. Effective communication among the

customers, users, development team and other stakeholders is necessary to produce the system goals, objectives and requirements from which all other work evolves. It is through communication that the iterative deployment and feedback at one phase can be applied to the next phase of development.

Only after the overall goals and objectives have been established does the planning begin. Planning principles (Table 2.5) guide scheduling, cost estimating, abstraction adjustment, quality assurance, change management, requirements tracking, process management, configuration management, metrics and measurement selection, etc. From the plans, models can be developed. Modeling principles (Table 2.6) apply to skills such as model selection, data management, creativity, algorithm development, design tradeoffs, relational database systems, interfaces, etc. The operational principles (Table 2.7) influence software architecture and design, interfaces, data management, process management, etc.

Teamwork, problem solving, conceptualization, creativity, algorithm development, flexibility, abstraction and concretization, software architecture, prototypes, interfaces, design tradeoff, etc. are skills that apply to the design phase. The design principles (Table 2.8) provide guidance for all design views relevant to users and software engineers.

Missing from the discussion of software engineering skills and principles thus far is coding. Coding is where novice software developers think the process begins. In an industry setting, software systems are large and complex and ignore the lifecycle phases prior to coding can be, and has been, disastrous. Yet, it is this phase that many community colleges and universities emphasize, almost to the exclusion of all others.

Many of the previous skills continue to be use at the coding phases. These and additional skills are guided by three levels of coding principles (Table 2.9): preparation, programming, and validation. Some of the skills used during the coding phases are programming language selection, integrated development environment selection, software development aids, teamwork, interfaces, software

architecture, data management, documentation, prototypes, code verification and validating, and unit testing.

Testing principles (Table 2.10) apply to test making and application as well as other skills such as defect tracking, metrics and measurement, and visual monitoring of the progress are guided by a set of testing principles. Deployment principles (Table 2.11) guide product deployment which can, or should, take place integrally during the development process. At each delivery, a package should be assemble and tested.

**Table 2.12 Software engineering technical skills**
[Conn 2002, Crnkovic, et al. 2003, Johnson and Jones 2006, Kornecki, et al. 2003, Lang 1999, Long 2008, Reifer 2005, Tilley and Wong 1993, Veraat et al. 1997]

| Software Engineering Technical Skills | General | Process | Practice | Communication | Plan | Model | Operational | Design | Coding | Testing | Deployment |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Abstraction, concretization | | | x | | | | x | x | x | | |
| Artifacts | | | | | x | | | | | | |
| CASE | | | | | x | x | x | x | x | x | |
| Change management | | x | | | x | | | | | | |
| Code testing | | | | | x | | | | x | x | |
| Code v & v | | | | | x | | | | x | x | |
| Configuration management | | | | | x | | | | | | |
| Cost estimating | | | | x | x | | | | | | |
| Data management | | | x | | | | x | | x | | |
| Defect tracking | | | | | x | | | | | | |
| Design documents | | | x | | | x | x | | | | |
| Develop algorithms | | | x | | x | x | x | x | x | | |
| Documentation | x | x | x | x | x | x | x | x | x | x | x |
| Estimation | x | x | | | x | | | | | | |
| Implementation | x | x | | | x | | | | | | |
| Integrated development environment | | | | | | | x | | | | |
| Language | | | | | | | | | x | | |
| Life cycle models | | x | | | | | | | | | |
| Maintenance | | | | | | | | | | | x |
| Metrics & measurement | | | | | x | | | | | | |
| Perform design tradeoffs | | | | | x | x | | | | | |
| Process management | | x | | | x | | | | | | |
| Process tools | | x | | | | | | | | | |
| Product development | | | | | x | | x | | | | |
| Project management | | x | x | x | x | | | | | | |
| Project tracking | | x | | | x | | | | | | |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Prototypes | | | | x | | x | | | | | |
| Relational database systems | | | | | | | x | | | | |
| Requirements engineering | x | | x | | x | x | | | | | |
| Requirements tracking tools | | | | | x | | | | | | |
| Scheduling | | x | | x | x | | | | | | |
| Software architecture | | | | | | | x | | x | | |
| Software design | x | | x | | x | | x | x | | | |
| Software development aids | | | | | | x | x | | | | |
| Specify interfaces | | | x | | | | x | x | | | |
| System analysis | x | | | | x | | | | | | |
| Visual modeling | | | | x | x | x | | | | | |
| Visual monitoring | | | | | | | | | x | x | |

**Table 2.13 Software engineering soft skills**
[Bailey and Stefaniak 2002, Conn 2002, Crnkovic, et al. 2003, Lang 1999,
Long 2008, Reifer 2005, Veraat et al. 1997]

| Software Engineering Soft Skills | General | Process | Practice | Communication | Plan | Model | Operational | Design | Coding | Testing | Deployment |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Candor | | x | | x | | | | | | | |
| Commitment | | x | | x | | | | | | | |
| Communicate with people from other engineering disciplines | x | | x | x | | | | | | | |
| Communication | x | x | x | x | x | x | x | x | X | x | x |
| Conceptual skills | x | x | x | | x | x | | | | | |
| Continuous improvement | x | | | | x | | | | | x | |
| Creating and managing change | | x | | | x | | | | | | |
| Creativity | x | x | x | x | x | x | x | x | X | x | x |
| Critical thinking | x | x | x | x | x | x | x | x | X | x | x |
| Flexibility | | x | | x | x | x | | | | | |
| Interface with user | x | x | | x | x | | | | | | |
| Manage people | | x | | x | | | | | | | |
| Marketing and sales | | | | | | | | | | | |
| Monitor ethical responsibility of team | | x | | x | | | | | | | |
| Organization and business knowledge | | x | | x | | | | | | | |
| Problem solving | x | | x | | x | x | | x | | | |
| Role modeling abilities | | | | x | | x | | | | | |
| Sense of ethics | | x | | x | | | | | | x | x |
| Strategic management | | x | | | x | | | | | | |
| Strong interpersonal skills | | x | | x | x | | | | | | |
| Teamwork | x | x | | x | | | | | X | | |

## 2.4Curriculum Guidelines

*Computing Curricula 2001: Computer Science* [Chang, et al. 2001] is the final report of Joint Task Force on Computing Curricula of the Computer Society of the Institute for Electrical and Electronic Engineers (IEEE-CS) and the Association for Computing Machinery (ACM). This volume of the report outlines curricular guidelines for undergraduate programs in computer science. Given the breadth of computing, the task force recommended that the report consist of additional volumes for specific disciplines, including computer engineering, software engineering, and information systems. In 2004, *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* [Le Blanc, et al. 2004] was published. This volume presents a set of curriculum recommendations for baccalaureate software engineering programs. As the result of the increasingly fast changing field of computer science, a ACM and IEE Computer Society Interim Review Task Force was formed to conduct an interim review of CS2001. The document resulting from the review is *Computer Science Curriculum 2008: An Interim Revision of the CS 2001* [McCauley and McGettrick 2008].

Other curriculum guidelines include volumes for associate-degree programs designed for students planning to transfer into baccalaureate programs. *Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science* [Campbell, et al. 2003] and *Computing Curricula 2005: Guidelines for Associate-Degree Transfer Curriculum in Software Engineering* [Campbell, et al. 2005] were developed by the ACM Two-Year College Education Committee and Joint Task Force on Computing Curricula of IEEE-CS and ACM. In 2009, the ACM Two-Year College Education Committee (ACMTYC) developed *Computing Curricula 2009: Guidelines for Associate-Degree Transfer Curriculum in Computer Science* [Hawthorne, et al. 2009].

### *2.4.1Computing Curricula 2001: Computer Science [Chang, et al. 2001]*

*Computing Curricula 2001: Computer Science* (CS2001) presents a set of recommendations for undergraduate programs in computer science. In addition to curriculum models and course descriptions,

this report identifies the body of knowledge appropriated for undergraduate computer sciences programs and a set of learning objectives for each of the units in the body of knowledge.

Multiple approaches to the structure of the introductory computer science courses have been developed through the years. Because introductory programs differ in goals, structure, resources, and audiences, the CS2001 Task Force acknowledged that there is no one-size-fits-all approach for all institutions and did not recommend an approach. This report presents three implementations of a programming-first model and three of an alternative paradigm. The programming-first implementations are imperative-first, objects-first, and function-first. The three alternative methods are breadth-first, algorithms-first, and hardware-first. All of these approaches have proven successful in the more traditional two-semester packaging, and the C2001 Task Force believes that the three-semester implementations will achieve similar levels of success. Although the role of programming in introductory computer science education is still a topic of debate, the programming-first model continues to be used in the majority of institutions, and the CS2001 Task Forces expects it to remain dominant for the foreseeable future.

**Table 2.14 Software engineering core topics and coverage hours**
[[1] Chang, et al. 2001, [2] Campbell, et al. 2003]

| Software Engineering Core Topics | [1] CS2001 4-year Minimum Core Hours | [2] CC2003 2-year Coverage Time (Hours) |
|---|---|---|
| SE1. Software design | 8 | 3-7 |
| SE2. Using APIs | 5 | 1-2 |
| SE3. Software tools and environments | 3 | 0-2 |
| SE4. Software processes | 2 | 0-1 |
| SE5. Software requirements and specifications | 4 | 0-1 |
| SE6. Software validation | 3 | 0-2 |
| SE7. Software evolution | 3 | 0-1 |
| SE8. Software project management | 3 | 0-1 |
| | 31 | 4-17 |

Another controversy in designing introductory computer science curricula is the length of the sequence. The CS2001 Task Force endorses a three-course introductory sequence; however, it recognizes

that it and the traditional two-course sequence have advantages for being used in a particular institution's program. Two- and three-course introductory sequences were developed for the imperative-first and objects-first tracks. TShe functional-first, algorithms-first, and hardware-first approaches exists only in the two-semester form. If the approach proves popular, it may be appropriate to consider a three-semester implementation. The task force proposed two implementations of a breadth-first approach. The first is simply to include an overview course (CS100B) before a more conventional programming sequence. The second is to expand the introductory curriculum into a three-semester sequence (CS101B-102B-103B) so that there is time for the additional topics.

CS2001 recognizes Software Engineering as a computer science body of knowledge core topic with a minimum coverage of 31 core hours, i.e., in-class hours. Table 2.14 lists the core units for the Software Engineering Body of Knowledge with the teaching topics and minimum core coverage time for each. Table 2.15 shows the coverage of the software engineering core hours by each of the six introductory course sequences.

### *2.4.2Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science [Campbell, R. (chair) et al. 2003]*

*Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science* (CC2003) shares goals and outcomes with CS2001. This report focuses on associate-degree computer science programs designed for students who intend to transfer into baccalaureate programs by presenting guidelines to enable students to transfer as smoothly as possible. The report promotes articulation by enabling a topical comparison by using this report together with CS2001.

**Table 2.15 Software engineering unit coverage hours in CS2001 introductory tracks**
[Chang, et al. 2001]

| | Software Engineering Unit No. | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Total hrs | | | | | | | | |
| | Min. coverage hours | 31 | 8 | 5 | 3 | 2 | 4 | 3 | 3 | 3 |
| **Imperative-first** | | | | | | | | | | |
| CS101I | Programming Fundamentals | 13 | 3 | | 2 | 1 | | | | |
| CS102I | Object-Oriented Paradigm | | 1 | 2 | | | 1 | 1 | 1 | |
| CS103I | Data Structures and Algorithms | | | | | | | 1 | | |
| CS111I | Introduction to Programming | 11 | 2 | | 1 | | 1 | 1 | | |
| CS112I | Data Abstraction | | 2 | 2 | 2 | | | | | |
| **Objects-first** | | | | | | | | | | |
| CS101O | Intro. to Object-Oriented Programming | 13 | | | 1 | | | 1 | | |
| CS102O | Objects and Data Abstraction | | 3 | | 1 | | 1 | 1 | 1 | |
| CS103O | Algorithms and Data Structures | | 1 | | | | | | | 3 |
| CS111O | Object-Oriented Programming | 10 | 2 | 1 | 2 | | | | | |
| CS112O | Object-Oriented Design and Methodology | | 2 | 1 | | | 1 | 1 | | |
| **Functional-first** | | | | | | | | | | |
| CS111F | Intro. to Functional Programming | 10 | 1 | | 1 | | | | | |
| CS112F | Objects and Algorithms | | 3 | 2 | 1 | | 1 | 1 | | |
| **Breadth-first** | | | | | | | | | | |
| CS100B | Preview of Computer Science | 10 | | | | | | | | |
| CS101B | Intro. to Computer Science | | | | | | | | | |
| CS102B | Algorithms and Programming Techniques | | 2 | | 1 | | 1 | 1 | | |
| CS103B | Principles of Object-Oriented Design | | 2 | 2 | 1 | | | | | |
| **Algorithms-first** | | | | | | | | | | |
| CS111A | Intro. to Algorithms and Applications | 11 | 2 | | | | 1 | | | |
| CS112A | Programming Methodology | | 2 | 2 | 2 | | 1 | 1 | | |
| **Hardware-first** | | | | | | | | | | |
| CS111H | Intro. to the Computer | 10 | 2 | | 1 | | | 1 | | |
| CS112H | Object-Oriented Programming Techniques | | 2 | 2 | 1 | | 1 | | | |

**Table 2.16 Software engineering unit coverage hours in CC2003 [Campbell, et al. 2003]**

| | Software Engineering Unit No. | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Total hrs | | | | | | | | |
| | Coverage hours | 4-17 | 3-7 | 1-2 | 1-2 | 0-2 | 0-1 | 0-2 | 0-1 | 0-1 |
| **Imperative-first** | | | | | | | | | | |
| CS101I | Programming Fundamentals | | 3 | | 2 | 1 | | | | |
| CS102I | Object-Oriented Paradigm | 16 | 1 | 2 | | | 1 | 1 | 1 | |
| CS103I | Data Structures and Algorithms | | | | | | | 1 | | 1 |
| **Objects-first** | | | | | | | | | | |
| CS101O | Intro. to Object-Oriented Programming | | | | 1 | | | 1 | | |
| CS102O | Objects and Data Abstraction | 15 | 3 | 2 | 1 | | 1 | 1 | 1 | |
| CS103O | Algorithms and Data Structures | | 3 | | | | | | | 1 |
| **Breadth-first** | | | | | | | | | | |
| CS101B | Computing Science I | | | | | | | | | |
| CS102B | Computing Science II | 3 | 2 | | | | | | | |
| CS103B | Computing Science III | | | | | 1 | | | | |

The Software Engineering Body of Knowledge units, shown in Table 2.14, recommended for the two-year college programs are the same as the core units recommended in CS2001. The CC2003 coverage time of each unit is given as a range and determined by the instructional paradigm and the chosen electives. [Chang, et al. 2001, Campbell, et al. 2003]

The introductory courses in CC2003 are implemented in three approaches: imperative- first, objects-first, and breadth-first. Each three-course sequence is fundamentally equivalent to the corresponding sequence in CS2001. Each is designed to be at least equivalent to the corresponding two-course sequence in CS2001. The differences in the courses are the additional time and preparatory material provided in the two-year college setting, as shown in Table 2.16.

#### ***2.4.3Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering [Le Blance and Sobel 2004]***

*Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* (SE2004) presents guidance for a baccalaureate software engineering education program. It contains the body of Software Engineering Education Knowledge (SEEK) and a curriculum describing how this knowledge can be taught. SEEK provides the foundation of the educational units that make up a software engineering curriculum. Although SE2004 concentrates on knowledge and pedagogy associated with a software engineering curriculum, there is overlap with material contained in other computing curriculum reports and guidance for the incorporation of software engineering in other disciplines.

Many software engineering topics require maturity. Introducing material early allows for subsequent reinforcing and expanding in later courses. Rather than the details of specific tools, the underlying principles of software engineering should be emphasized. A software engineering program must allow its graduates to feel confident in their ability when entering the workforce.

SE2004 presents two sequences for the introductory-level of the software engineering curriculum: software engineering-first and computer science-first. The computer science-first approach is the more common, but there are advantages and disadvantages for both approaches. The software engineering-first approach allows the student to focus on a problem and the way it can be solved without thinking primarily in terms of code. It also gives an early idea of what software engineering is. The computer science-first allows students to begin practicing their programming skills early. Most textbooks are written for teaching computer science-first. Another factor to consider is that students who know little about computers and programming may have trouble grasping software engineering concepts during the first year.

Although undergraduate software engineering and computer science degrees differ, the introductory-levels have much in common. The SE2004 computer science-first sequence begins with

CS101I Programming Fundamentals, CS102I The Object-Oriented Paradigm, and CS103I Data Structures and Algorithm. Other courses in CS2001 can be substituted for these. The first year of the software engineering-first sequence contains SE101 Introduction to Software Engineering and Computing 1 and SE102 Software Engineering and Computing 2.

#### *2.4.4Computing Curricula 2005: Guidelines for Associate-Degree Transfer Curriculum in Software Engineering [Campbell, et al. 2005]*

*Computing Curricula 2005: Guidelines for Associate-Degree Transfer Curriculum in Software Engineering* (CC2005) provides guidelines for an associate-degree software engineering curriculum designed for students intending to transfer into software engineering baccalaureate programs. It is specifically designed to promote articulation of curricula for two-year colleges and baccalaureate institutions. CS2001, CC2003, and SE2004 provide the foundation for this work. This report is based on the computer science-first approach for the following reasons: (1) students with limited knowledge of programming may not have the necessary background for the study of software engineering concepts; (2) the current guidelines for foundation computer science curricula include concepts and programming paradigm that must be mastered; and (3) the software engineering curriculum track can be implemented easily for those institutions with computer science curricula based on current ACM standards.

This report indentifies two of the three introductory computer science paradigms presented in CC2003 as being appropriate in an associate degree software engineering curriculum: imperative-first and objects-first. By using SE2004's SE201 Introduction to Software Engineering as a suggested second-year elective, the software engineering track adapts well into the computer science transfer degree program.

#### *2.4.5Computing Curricula 2005: Overview Report on Computing Curricula [Shackelford, et al. 2005]*

In addition to the set of reports that cover the computing-related disciplines, CS2001 requested an Overview Report to summarize the content of the discipline-specific reports. This report provides the perspective needed to understand the major computing disciplines and how the undergraduate degree programs compare and complement each other. *Computing Curricula 2005: Overview Report on Computing Curricula* (CC2005-Overview), summarizes the body of knowledge of the undergraduate programs in each of the major computing disciplines (computer engineering, computer science, information systems, information technology, and software engineering), highlights their commonalities and differences, and describes the performance characteristics of graduates from each kind of program.

Computer science and software engineering degree programs have many courses in common. While computer science students are exposed to software reliability and maintenance, software engineering students focus on software reliability and quality during the complete software development cycle. Engineering knowledge and experience are applied to develop software that is correct, genuinely useful, and usable by the customer. Figures 2.1 and 2.2 are graphical representations of the computer science and software engineering disciplines, respectively. The shaded area of Figure 2.1 shows that computer scientists are concerned with the whole spectrum of computing from the software that enables devices to work and to the information systems that help organizations to operate. The fact that the shaded area narrows and stops before reaching the right edge indicates that computer scientists create the capabilities, but they do not manage the deployment of them. The shaded area in Figure 2.2 shows that software engineers cover software development from the conception to the deployment as they oversee large software system development. The vertical range shows that software engineers develop software infrastructure and design and develop information systems that are appropriate to the client's organization.

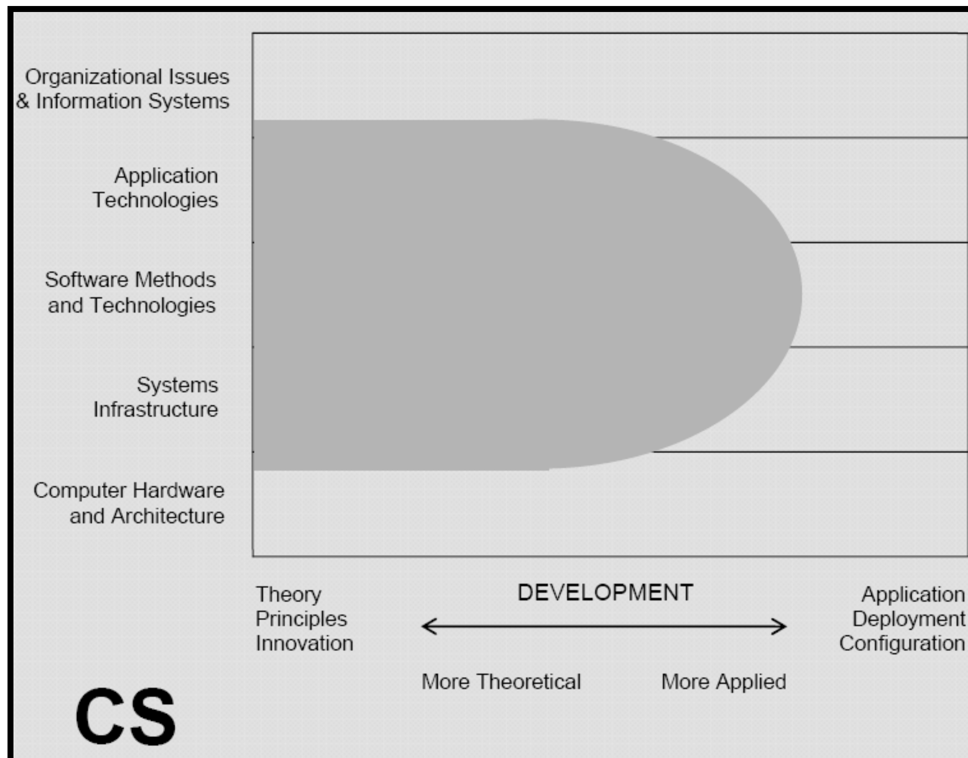

**Figure 2.1 Computer Science [Shackelford, et al. 2005].**

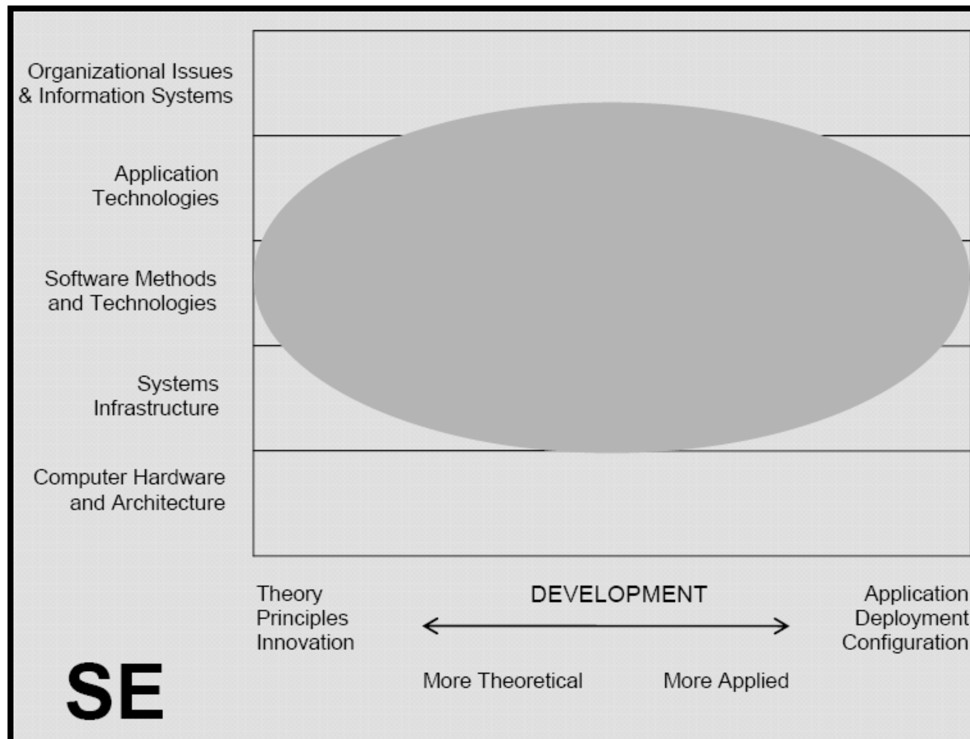


Figure 2.2 Software Engineering [Shackelford, et al. 2005

***2.4.6Computer Science Curriculum 2008: An Interim Revision of the CS 2001 [McCauley and McGettrick 2008]***

As the result of the increasingly fast changing field of computer science, an Interim Review Task Force (RTF) was form to conduct an interim review of CS2001. The CS2008 Review Task Force is a joint task force of the ACM and IEEE Computer Society. The document resulting from the review is *Computer Science Curriculum 2008: An Interim Revision of the CS 2001* (CS2008). Released in December 2008, this report includes an update of the CS2001 body of knowledge as well as a commentary on resent developments and trends in the computer science discipline. The RTF was directed to solicit and consider feedback from the industrial, the two- and four-year academic communities, and individuals.

Some of the relevant trends found to influence the evolution of computer science were (1) the emergence of security as a major area of concern, (2) the growing relevance of concurrency, (3) the persistent nature of net-centric computing, and (4) the stronger development of the concept of systems issues. There were no changes made to the set of software engineering core topics listed in CS2001.

However, changes were made to individual knowledge areas to reflect the greater emphasis on security and the updating of net-centric computing. There is also a more detailed list of topics and learning objects for each knowledge area.

This report addresses the debate of programming languages and paradigms. Although the RTF was divided in its agreement with the SIGPLAN proposal [SIGPLAN 2008], it did agree that students need to be exposed to more than one programming paradigm. The task force did not agree that the functional programming paradigm needed to be required in all undergraduate computer science curricula. Rather, the RTF chose to add a new requirement in Chapter 9 (“Completing the Curriculum”) of the CS2001 report. This requirement recognized that, because professionals frequently used different programming languages, students must recognize the benefits of learning and applying new programming languages. Therefore, the committee recommended that all students must learn to program in more than one paradigm. The choice of the secondary paradigm would depend on the character and educational goals of the institution.

Another concern addressed by the RTF was the enrollment and retention crisis in computing. This issue was considered important enough to warrant a new chapter, “Reflections on the Computing Crisis,” to address (1) finding new and better ways of teaching computer programming and (2) trying to present computing in a perspective that would motivate and inspire students.

#### *2.4.7Computing Curricula 2009: Guidelines for Associate-Degree Transfer Curriculum in Computer Science [Hawthorne, et al 2009]*

The ACM Two-Year College Education Committee (ACMTYC) developed a set of curriculum guidelines that provide guidance for associate-degree programs that are similar to those in the ACM Computing Curricula Series for baccalaureate programs. *Computing Curricula 2009: Guidelines for Associate-Degree Transfer Curriculum in Computer Science* (CC2009) provides discussion on articulation as the key consideration when designing courses and programs that facilitate transfers by

student between two- and four-year institutions. Efficient and effective articulation requires well-defined courses and program outcomes as well as meaningful communication and cooperation between institutions and faculty.

The computer science associate-degree transfer program calls for a blended approach with object-oriented programming emphasized in the latter part of CS1. CS1 includes algorithms and fundamental programming constructs consistent with the Bohm-Jacopini theory for procedural programming. Ethics and professionalism, security, and software engineering principles are presented in CS1 using the breadth-first approach. These topics are covered deeper throughout the CS1-CS2-CS3 series. Software engineering principles are essential in the curriculum to ensure a disciplined, controlled approach to software evolution and reuse. The coverage of security topics include encapsulation in CS1, exception handling in CS2, and developing attack-resistant code in CS3. The third emphasis, professionalism and ethics, begins in CS1 by examining the computing and ethical conduct and individual behaviors. CS2 continues with consideration of societal impacts of computing. In CS3, students begin to internalize the importance of professional and ethical behavior.

**Table 2.17 Computer science associate-degree program outcomes [Hawthorne 2009]**

| Group 1 – **Critical thinking, problem solving, and theoretical foundations** | |
|---|---|
| Outcomes | A. An ability to apply knowledge of computing and mathematics appropriate to the discipline.<br>B. An ability to think critically and apply the scientific method.<br>C. An ability to analyze a problem and craft an appropriate algorithmic solution.<br>D. An ability to design, implement and evaluate an appropriate and secure computer-based system, process, component, or program to satisfy required specifications. |
| Group 2 – **Communication and interpersonal skills** | |
| Outcomes | A. An ability to read and interpret technical information, as well as listen effectively to, communicate orally with, and write clearly for a wide range of audiences.<br>B. An ability to function effectively as a member of a team to accomplish common goals. |
| Group 3 – **Professionalism and ethics, social awareness and global perspective** | |

| Outcomes | |
|---|---|
| Outcomes | A. An ability to engage in continuous learning as well as research and assess new ideas and information to provide the capabilities for lifelong learning.<br>B. An ability to exhibit professional, legal and ethical behavior.<br>C. An ability to demonstrate social awareness, respect for privacy and responsible conduct.<br>D. An ability to analyze the global impact of computing on individuals, organizations, and society. |

The ACMTYC recommends that the entire CS1-CS2-CS3 core sequence presented in this report be completed at the same educational institution with the programs of study having well-defined exit-points. Two- and four-year institutions are advised to work together to design compatible and consistent programs of study that enable students to transfer easily from associate-degree programs into baccalaureate-degree programs.

Clearly defined program outcomes at the course and program levels are essential to developing effective articulation agreements. CC2009 developed three groups of outcomes, shown in Table 2.17, which students should demonstrate upon successful completion of the computer science associate-degree program. In addition to the program outcomes, CC2009 defines an assessment rubric for the student learning outcomes for each of the three core CS courses.

**Table 2.18 Program outcomes and supporting coursework [Hawthorne 2009]**

| Program Outcomes | Critical thinking, problem solving, and theoretical foundations | | | | Communications and interpersonal skills | | Professionalism and ethics, social awareness and global perspective | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Group 1 | | | | Group 2 | | Group 3 | | | |
| | A | B | C | D | A | B | A | B | C | D |
| CS1 | x | x | x | x | x | | x | | x | x |
| CS2 | x | x | x | x | x | x | x | | x | x |
| CS3 | x | x | x | x | x | x | x | x | x | x |
| Discrete Structures | x | x | x | x | x | | x | | | |
| Calculus I | x | x | x | | | | x | | | |

The ACMTYC recommends that an associate-degree transfer curriculum in computer science includes a core computer science sequence, CS1-CS2-CS3, and foundational mathematics courses. Table

2.18 summarizes the program outcomes and identifies the underlying support provided by the CS core sequence and the foundational mathematics courses.

In CS1, students develop fundamental programming skills using a language that supports an object-oriented approach. In CS2 and CS3 students continue with the students developing intermediate and advance programming skills using a language that supports an object-oriented language. It is recommended that each core computer science course has 42 minimum contacts hours. The course topics for each of the core computer science course are listed in Table 2.19 along with the recommended hours per topic heading.

**Table 2.19 Computer science sequence topics [Hawthorne 2009]**

| **CS Core Sequence Topic Headings** with recommended hours per topic in ( ) | | | **ACM Computing Ontology Topic Classifiers** |
|---|---|---|---|
| **CS 1** | **CS 2** | **CS 3** | |
| Social & historical context of computing (1) | Ethical conduct (1) | Professionalism (1) | Ethical Social; History Computing |
| Programming languages (1) | Event-driven programming (4) | | Programming Languages |
| IDE & software tools (2) | | | Programming Languages |
| Fundamental programming constructs (11) | Intermediate programming constructs (3) | Recursion (7) | Programming Fundamentals; Programming Languages |
| Machine level representation of data (1) | | | Computer Hardware Organization |

| | | | |
|---|---|---|---|
| Fundamental algorithms & problem-solving (6) | Intermediate computing algorithms (5) | Formal computing algorithms (8) | Algorithms Complexity; Discrete Structures |
| | Object-oriented design & modeling (5) | Software reuse | Conceptual Modeling; Information Topics |
| Object-oriented principles (6) | Object-oriented programming (7) | | Programming Languages |
| Fundamental Data Structures (6) | Intermediate data structures (7) | Canonical data structures (7) | Programming Languages; Algorithms Complexity |
| Secure code (2) | Software assurance (3) | Software & information assurance (3) | Security Topics |
| Overview of operating systems (1) | | Concurrency (2) | Computing & Network Systems |
| Human-computer interaction (1) | Human-computer interaction (2) | Human-computer interaction (2) | User Interface; Graphics, Visualization, Multimedia |
| | Simple database integration (1) | | Information Topics |
| Program development (3) | Software development (4) | Software engineering (4) | Software Engineering |
| | | Basic algorithmic analysis (3) | Algorithms Complexity |
| | | Algorithmic strategies (3) | Algorithms Complexity |

#### *2.4.8The Guide to the Software Engineering Body of Knowledge [Tripp, et al. 2004]*

*The Guide to the Software Engineering Body of Knowledge* (SWEBOK) is the result of a project initiated by the Software Engineering Coordinating Committee (SWECC), a joint effort of IEEE-CS and ACM. The objectives of this guide are to (1) promote a worldwide, consistent view of software engineering; (2) set the boundary of software engineering with respect to other disciplines such as computer science, project management, computer engineering, and mathematics; (3) determine the contents of the software engineering body of knowledge; (4) provide a topical access to the software engineering body of knowledge, and (5) provide a foundation for curriculum development and individual certification and licensing material.

SWEBOK centers on organizing the software engineering body of knowledge into ten Knowledge Areas (KAs) which are listed in Table 2.20. Each KA is divided into subtopics that were selected with consideration to (1) compatible with major schools of thought, (2) breakdowns generally found in industry, and (3) breakdowns in software engineering literature and standards. The topics descriptions convey what is needed to understand the

generally accepted nature of the topics and are not related to particular application domains, business uses, management philosophies, development methods, etc.

**Table 2.20 The SWEBOK Knowledge Areas**
[Tripp, et al. 2004]

| |
|---|
| Software requirements |
| Software design |
| Software construction |
| Software testing |
| Software maintenance |
| Software configuration management |
| Software engineering management |
| Software engineering process |
| Software engineering tools and methods |
| Software quality |

Each KA subtopic has been identified with a proposed Bloom's taxonomy level appropriate for a "generalist" software engineer graduate with four years of knowledge. Bloom's taxonomy [Bloom 1956] is a well-accepted and widely used classification of cognitive educational goals. An explanation of the Bloom's taxonomy levels and the list of the KA subtopics are presented in Appendix A. The Bloom's taxonomy levels were included in SWEBOK to assist with course and curricula development, university accreditation criteria, job descriptions, software engineering process role descriptions, professional development and training programs, etc. Because a four-year software engineering graduate lacks management experience, the management-related topics are not given a high priority in the taxonomy levels. SWEBOK, also, assumed that graduates would have less knowledge of life cycle topics related to software requirements than for more technically-oriented topics like software design, software construction, or software testing.

## 2.5Software Development Tools

Software development tools and programming environments have existed since the early days of computer programming. The demands for more complex software in less time have made tools and environments more crucial than ever expected. All software engineers use tools. Some use stand-alone

tools while others use integrated collections of tools. Traditional tools are editors, compilers, and debuggers. Today, tools are being developed to provide a broader coverage of the software engineering lifecycle. These tools aid in requirements gathering, design, building GUIs, generating queries, defining messages, architecture systems and connecting components, testing, version control and configuration management, administering databases, reengineering, reverse engineering, analysis, program visualization, and metrics gathering. Others are full-scale, process-centered software environments that cover all, or a significant part, of the life cycle. [Harrison, et al. 2000]

The development of large, complex software systems require tools that help (1) trace connections among products to monitor change impact analysis, (2) measure progress of product development, (3) simulate and understand parts of a problem to select the correct solution, and (4) support reuse so we can easily extract material from existing developments and incorporate it into current products. [Pfleeger and Atlee 2006]

Computer-Aided Software Engineering (CASE) automates some of the software process and provided information about the software being developed. [Sommerville 2004] The power of CASE is its integrated environment that (1) allows smooth transfer of information from one task to another; (2) reduces the effort required for tasks such configuration management, quality assurance, and creating documentation; (3) increases project control through better planning, monitoring, and communication; and (4) coordinates work effort on a large software project. [Pressman 2001]

Programming environments are a collection of tools that support coding activities and included one or more compilers, language-sensitive editors, debuggers, and, sometimes, testing or documentation utilities. Each of these environments supports only one software engineering activity and its artifacts, implementation and code, respectively. The need for integrated support throughout the software engineering lifecycle led to integrated design environments (IDE). [Harrison, et al. 2000]

Pedagogical programming environments should facilitate learning to program effectively and efficiently and help students understand problem-solving strategies. For pedagogical purposes, an IDE should satisfy several fundamental requirements: (1) assist students in writing correct syntax in a language in which they may not be proficient; (2) provide a simple interface to the language compiler and provide a visual flag for syntax errors; and (3) provide an alternative to command line interface for running a program. In addition to these, an IDE used in teaching an introductory-level programming course should (1) be simple and non-intimidating; (2) provide simple mechanisms for working around complicated aspects of the Java language; and (3) be able to run on older, less capable hardware. [Reis and Cartwright 2003]. A short learning curve for a new software tool or programming environment is an important factor for integration into an educational setting [Kouznetsova 2007, Sanders and Heeler 2001, Bouillon, et al. 2003, Reis and Cartwright 2003, Roy 2006, Boloix and Robillard 1998].

Many choices and types of programming environments are available for teaching CS1 and CS2 courses. Many instructors feel that the number of features in professional level IDEs--Eclipse, IBM Rational, JCreator, Netbeans, Borland JBuilder, and Microsoft Visual--is a distraction for student; the learning curve is too steep; and they are too costly. Environments have been design for specifically for pedagogical purposes. The most prominent of these are BlueJ, DrJava, and jGRASP. [Burch 2009, Chen and Marx 2005]

### *2.5.1Professional Integrated Design Environments*

*Eclipse*. Released by IBM in 2004, Eclipse is an open-source, professional IDE with extensible frameworks, tools, and runtimes for building, deploying, and managing software across the software lifecycle [Eclipse 2009]. The Eclipse plug-in architecture makes the IDE extensible for multiple programming languages with a consistent look and feel [Czyz and Jayarman 2007].

Eclipse is a platform for tool integration allowing modeling, design, programming, and testing tools to come together. Eclipse is supported by several platforms including Windows, Linux, Apple Mac OS, and most major UNIX systems.

The Eclipse Java Development Environment (JDT) includes an editor, a debugger, and a variety of refactoring operations. JDT contains integrated support for the Java build tool Apache Ant, a unit-testing tool JUnit and a documentation tool JavaDoc. [D'Anjou, et al. 2005]

Eclipse does not satisfy the previously mentioned requirements for introductory programming course, but it may be appropriate for intermediate and advanced courses [Reis and Cartwright 2003]. While Eclipse has large established usage, the availability of education material is lacking. Because of its abundance of windows and unfamiliar concepts like perspectives, Eclipse has a steep learning curve [Bouillon, Burger, and Zeller 2003, Reis and Cartwright 2003, Czyz and Jayarman 2007, Deugo 2008, Rubel 2006] but, once mastered, it is a powerful development environment [Bouillon, et al. 2003, Czyz and Jayarman 2007]. Although it is a powerful professional tool, it has better complier error messages than BlueJ or DrJava and detects most syntax errors as the user types [Olan 2004].

*NetBeans*. Similar to Eclipse, the NetBeans project consists of an open-source IDE and an application platform that enable developers to create web, enterprise, desktop, and mobile applications using the Java platform. This IDE, also, supports JavaFX, PHP, JavaScript and Ajax, Ruby and Ruby on Rails, Groovy and Grails, and C/C++. NetBeans is a collaborative where a team of developers can check out, edit, debug, build, discuss, and commit code through on interface. It provides integrated file version control through easy access to Concurrent Versions Control (CVS), Mercurial, or Subversion; documentation via Javadoc; and unit testing with JUnit. [Netbeans 2009]

**Table 2.21 Core capabilities of rational products and services [Rational 2009]**

| **Architecture management:** |
|---|
| •Design, model, develop and deliver software and systems and solutions help manage software quality throughout the lifecycle.<br>•Control reuse and automation capabilities.<br>•Integrate with existing software development tools, like a visual modeling design tool for designing with Unified Modeling Language (UML) and automate design-to-code translation. |
| **Change and release management:** |
| •Improve software delivery and lifecycle traceability, from requirements through deployment.<br>•Unify distributed teams, automate software assembly processes, and provide traceability across the software development lifecycle.<br>•Collaborative software delivery through integrated version control and automated workflows.<br>•Identify code-level issues through static analysis.<br>•Provides version control, workspace management, and parallel development support.<br>•Provides flexible defect and change tracking, process automation, reporting, and lifecycle traceability for better visibility and control of change and the development lifecycle. |
| **Enterprise architecture management:** |
| •Link, consolidate, and analyze information concerning strategy, business architecture, information systems, and technology infrastructure.<br>•Drive reuse by collecting and maintaining current information about enterprise building blocks. |
| **Integrated requirements management:** |
| •Define and manage requirements.<br>•Provide traceability and alignment with business procedures.<br>•Offer best practices in requirements definition and requirements management.<br>•Control and manage changes to requirements.<br>•Measure the impact of changes as they occur.<br>•Demonstrate compliance by ensuring full traceability of requirements. |
| **Product, project, and portfolio management:** |
| •Align business goals, best practices, and projects for improved productivity and predictability.<br>•Automate proven governance and delivery process for consistency.<br>•Manage value and delivery risk across the lifecycle. |
| **Quality management:** |
| •Ensure software functionality, reliability, compliance, security, and performance throughout development and production.<br>•Automate software functional testing, load testing, performance testing, scalability testing, run-time analysis, memory leak detection, performance profiling, and component unit testing. |

*IBM Rational*. The IBM Rational software architect, built on Eclipse, is a process-centered software environment (PSEE) integrates process, tools, and automated tasks. PSEEs integrate tool support for software artifact development and support the modeling and executing of the software

engineering processes that produce the artifacts. The representation of processes, their products, and their interactions is the foundation on which modern integrated development environments, such as IBM Rational, are built. [Harrison, et al. 2000]

IBM Rational provides an approach to iterative lifecycle management for faster and more consistent software development. The core capabilities (Table 2.23) of Rational products and services assists in all software development needs.

### *2.5.2Pedagogical Integrated Design Environments*

*jGRASP*. Developed by Auburn University, jGRASP is a lightweight development environment that is implemented in Java and runs on all platforms with a Java Virtual Machine (JVM). It provides for the automatic generation of software visualizations to improve software comprehension. The Object Workbench, Debugger, and Interactions features are tightly integrated with the visualizations: Control Structure Diagrams (CSDs), UML Class Diagrams, and Viewers. Each is described below:

- The Object Workbench works with the UML class diagram and CSD window, allowing the user to create instances of classes and invoke their methods. An object on the workbench can be viewed and changes observed as methods are invoked.
- The integrated Debugger works with the CSD window, UML window, and the Object Workbench. Users can use the Debugger to examine their program step by step.
- The Interactions features allow the user to execute or evaluate most Java statements and expressions as they are entered.
- Control Structure Diagrams are available for Java, C, C++, Objective-C, Ada, and VHDL to depict control constructs, control paths, and overall structure of each program unit. The CSD window supports editing, compiling, running, and debugging programs. [jGRASP 2009b] The jGRASP CSD visualizes the dynamic behavior of code which allows students to quickly comprehend the

meaning of the program they are writing or reading. This is especially helpful in understand nested control structures. [Buck and Stucki 2001]

- UML Class Diagrams for Java depict with arrows dependencies among classes. UML class diagrams can be generated for Java source code from all Java class and jar files in the current project. Class members and class dependencies can be displayed. The UML diagram assists in understanding the dependencies among classes when using object-oriented software. [jGRASP 2009b]
- Dynamic Viewers were developed primarily for students in an introductory-level data structure and algorithms course and help increase the accuracy and reduce the time taken to write programs implementing [jGRASP 2009a]. The object viewers for Java include a data structure identifier mechanism that recognizes traditional data structures objects such as stacks, queues, linked lists, binary trees, and hash tables, and displays them in a textbook-like view. The viewers provide dynamic visualizations of objects and primitives as the user steps through a program in debug mode or invokes methods from an object on the workbench. When a viewer is opened, the structure identifier attempts to automatically recognize linked lists, binary trees, and array wrappers (list, stacks, queues, etc.) during debugging or workbench use. [jGRASP 2009b]

Available at no cost, jGRASP is currently being used in almost 300 educational institutions world-wide including 37 high schools and districts, 48 community colleges, and 209 colleges and universities [jGRASP 2009]. Many instructors have had positive anecdotal response to the use of these viewers in CS1 and CS2 courses [Cross, et al. 2007].

*BlueJ*. BlueJ is a Java pedagogical IDE designed for teaching an object-oriented first introductory course. Used by hundreds of educational institutions world-wide, BlueJ is based on the Blue system, which is an integrated teaching environment and language developed at the University of Sydney and Monash University, Australia. [Sanders 2001, BlueJ 2009]

Some of the characteristics of BlueJ that allow it to be a good approach for teaching introductory-level object-oriented and data structure courses are: it is free of charge; it is easy to use with short learning curve; it supports automatic construction of class diagrams; it allows customizable templates for class skeletons; it has the ability to instantiate objects and test methods without a driver program; it is an integrated debugger; it previews or creates HTML documentation (via javadoc); it can import packages that are not created with BlueJ; and it has an automatic make utility. [Smith and Boyd 2001, Xinogalos, et al. 2007, Paterson, et al. 2005, Sanders 2001]

BlueJ eliminates the dependence on Java's main method and console input and output. It uses class diagrams and a graphical "workbench" to allow students to interact visually with their programs without writing code. [Reis and Cartwright 2004, Kolling 2003]. BlueJ supports a fully integrated environment; graphical class structure display; graphical and textual editing; built-in editor, compiler, virtual machine, debugger, etc.; an easy-to-use interface; interactive object creation and calls; interactive testing; and incremental application development [Sanders 2001, BlueJ 2009]. Creating an object displays a UML object diagram in BlueJ's object bench [Olan 2004].

Unit testing in BlueJ combines BlueJ's interactive testing functionality with the regression testing of JUnit. New functionality resulting from the combination of the two systems allows interactive test sequences to be recorded automatically creating JUnit test methods for later regression testing. [Kolling 2009]

The BlueJ support tools present a limited subset of the professional version control systems with a significantly simplified interface. When teamwork tools are enabled, users can check out or share a project, update from or commit changes to a repository, and obtain file status and project history. [Fisker, et al. 2008]

BlueJ describes programs using both UML diagrams and text, which can make the environment more complex than other pedagogical IDEs. To use BlueJ, the user must learn both Java and the

protocols for using the graphical programming interface. [Allen, et al. 2002] It does not scale to large project or computations and is limited to developing small programs in introductory courses [Reis and Cartwright 2004, Allen, et al. 2002].

*DrJava*. DrJava is a lightweight Java development environment which provides the ability to interactively evaluate Java code. Developed at Rice University, DrJava is available for free under the BSD License [DrJava 2009] and is available on multiple platforms, including Windows, Linux, and Macintosh [Olan 2004].

DrJava's interactions are based on a read-eval-print interpreter similar to Scheme and evaluate Java expressions and statements interactively. Users can experiment with Java constructs by typing an expression or statement and having it evaluated immediately, without having to write a full Java program. The interface is text-based and requires Java syntax. [Olan 2004, Smith and Boyd 2001]

As a pedagogical IDE, DrJava's most important benefits are its simplicity and its interactive interface. The user interface is designed to be accessible to beginners, with clearly labeled buttons and few distractions in a simple graphical layout. It consists of three panes: (1) a Definitions Pane used to enter program text, (2) an OpenFiles panel listing the open files and highlighting the one selected for display in the Definitions Pane, and (3) an Interactions Pane used to evaluate arbitrary statements and expressions in the context of the files listed in the OpenFiles Pane. [Reis and Cartwright 2004]

When a class is compiled in the Definitions Pane, it is immediately available for use in the Interactions Pane. An interaction can be copied into the editor providing a convenient way to move experimental tests into a JUnit test to make them repeatable. Because each class method can be executed independently, the Interactions Pane is an effective tool for simple testing and debugging. DrJava also includes a source-level debugger which supports setting breakpoints and defining watches. The DrJava compiler parses the entire file and reports all syntax errors. The editor provides automatic indentation, parenthesis/braces matching and quotation/comment highlighting that is updated with every keystroke.

Other features of DrJava include built-in support for JUnit test cases; generation of Javadoc documentation; Javadoc preview for the current document; single medium for program development – text; and a history list of statements/expression in the Interactions Pane. [Olan 2004, Reis and Cartwright 2003] DrJava is effective on sizable projects, but the environment lacks built-in refactoring tools [Reis and Cartwright 2003].

The first stage of a DrJava plug-in for IBM's Eclipse is available. This plug-in provides a fully-functional Interactions Pane and simplified user interface to Eclipse. The next stage of development will integrate Eclipse's debugger with the Interactions Pane, allowing users to interact with their programs while at a breakpoint. [DrJava 2009]

#### *2.5.3Microworlds*

The objects-first strategy for teaching programming has spawned the development of a new type of educational software tools that is intended to reduce the gap between students' mental models and the programming language. These software tools are programming microworlds based on a physical metaphor. JKarelRobot, Alice, and Greenfoot are this type of software tool that are used in educational settings. [Xingalos 2006]

*JKarelRobot*. Karel the Robot, created at Carnegie Mellon University, uses a simple set of primitives and contains branching and looping structures as well as procedural abstraction. Rather than using a traditional programming language, each program has a robot execute a task. The language is block structured and has basic branching, looping and procedure abstraction control structures. [Barnett 2009] To aid in the comprehension the meaning of programs, especially nested control structures, JKarelRobot supports CSDs. JKarelRobot, written in Java, is platform and language/paradigm independent, supporting Pascal, Java, and Lisp-style environments [Buck and Stucki 2001].

Karel can only exist at discrete coordinates in a discrete 2D world. The abstract notion of the state of variables is replaced with the state of the world that Karel occupies. JKarelRobot expands on the

Karel the Robot language and environment (1) to support more directly the primitive levels of cognitive development and (2) to teach more concepts and support more of the curriculum. JKarelRobot allows students to learn programming concepts without the syntactical baggage or the complexities of a real programming environment. It does so at the cost of only supporting the single metaphor of manipulating a robot on a 2D gridded space. [Buck and Stucki 2001]

*Greenfoot*. Greenfoot is a free programming environment that is suitable for novice programmers. It is a tool for modeling and simulating single objects on a 2D space. Greenfoot contains the typical elements of a integrated development environment: source code editor, class browser, compilation, execution control, and debugger. The greenfoot framework (1) makes it easy to create representations of objects and (2) controls the execution of a simulation loop. Greenfoot can visualize and directly interact with objects from a Greenfoot scenario. [Greenfoot 2009] In contrast to some microworld's single scenario, Greenfoot provides an interactive microworld meta-framework which allows multiple scenarios and iconic objects in the world. The Greenfoot runtime and compiler uses standard Java, and Greenfoot classes are standard Java classes. The Greenfoot implementation is based on the BlueJ system and many BlueJ tools are available in Greenfoot. [Henriksen and Kolling 2004]

*Alice*. Alice is a free 3D programming environment created at Carnegie Mellon University to be students' first exposure to object-oriented programming. Students can learn fundamental programming concepts while populating a virtual world with 3D objects and creating animated movies and simple video games. Its interactive interface allows students to drag and drop tiles that represent tokens of a programming language and immediately see how their animated programs run. [Alice 2009] Alice is built on top of the Python programming language and uses many Python features. Alice functions and decisions are supported through the underlying Python language. [Cooper 2000]

**Table 2.22 Software engineering curriculum and software development environments**

| Software Development Environments / CC2001 Software Engineering Core Areas of Study | Professional | | | Pedagogical | | | |
|---|---|---|---|---|---|---|---|
| | Eclipse | NetBeans | Rational | jGRASP | BlueJ | DrJava | Micro-worlds |
| SE1. Software design | x* | | x | x | | | |
| SE2. Using APIs | x | x | x | x | x | x | x |
| SE3. Software tools and environments | x | x | x | x | x | x | x |
| SE4. Software processes | | | x | | | | |
| SE5. Software requirements and specifications | | | x | | | | |
| SE6. Software validation | x | x | x | | x | x | |
| SE7. Software evolution | | | x | | x | | |
| SE8. Software project management | x | x | x | | x | | |
| * The software development environment covers at least part of the curriculum topics in the core area of study. | | | | | | | |

3Community Colleges

Community colleges are a 100-year-old American invention that put publicly funded higher education at close-to-home facilities. Since their beginning, they have been inclusive institutions welcoming all who desire to learn, regardless of wealth, heritage, or previous academic experience. [AACC 2009, Kasper 2002] Making higher education available to the maximum number of people continues at 1,173 public and independent community colleges with branch campuses bringing the number to about 1,600 across the United States. [AACC 2009]

**Table 3.1 Percentages of postsecondary enrollment increase**
**[[1]ACHE 2009a, [2]GAO 2007]**

| | [1]Alabama 1999 to 2007 | | | [2]United States 2000-01 to 2006-07 |
|---|---|---|---|---|
| | 4-yr | 2-yr | Average | |
| Female | 21.4% | 18.1% | 19.7% | na |
| African Am | 24.2% | 15.1% | 19.7% | 15.0% |
| White/ non-Hisp | 9.0% | 9.1% | 9.1% | 3.0% |
| Am Indian/ Alaskan Native | 17.9% | 21.6% | 19.8% | na |
| Asian/ Pacific Island | 43.3% | 45.2% | 44.2% | 15.0% |
| Hispanic | 97.3% | 107.5% | 102.4% | 25.0% |
| Total enroll | 16.2% | 11.9% | 14.0% | *22.5% |
| | | | | * Average of 21.0% 4-yr and 24.0% 2-yr |

Community colleges have become an important part of postsecondary education in the United States. Over the past 40 years, public community college enrollment has increased at a much faster rate than at the public four-year universities, with the percentage of women enrolled in community colleges

surpassing that of men. Because of the low cost and accessibility, racial and ethnic minorities have become an increasing proportion of all students enrolled at community colleges. [Kasper 2002] In the time of a recession, community colleges experience an abnormal increase in student enrollment as unemployed workers seek to continue their education or change career fields [Tirrell-Wysocki 2009].

**Table 3.2 Alabama new undergraduate transfers summary**
[ACHE 2008a]

| Fall | AL Public 2yr to 4yr | Enrollment Increase | Percent Increase |
|---|---|---|---|
| 2008 | 6001 | 400 | 7.1% |
| 2007 | 5601 | 640 | 12.9% |
| 2006 | 4961 | 449 | 10.0% |
| 2005 | 4512 | -85 | -1.8% |
| 2004 | 4597 | 243 | 5.6% |
| 2003 | 4354 | 206 | 5.0% |
| 2002 | 4148 | -157 | -3.6% |
| 2001 | 4305 | 436 | 11.3% |
| 2000 | 3869 | 236 | 6.5% |
| 1999 | 3633 | Not available | Not Available |
| | | | |
| Increase 1999-2008 | | 2368 | 65.2% |

3.1Higher Education and Community College Demographics

The November 2007 U.S. Government Accountability Office (GAO) report on higher education in the United States indicateded that, for the 2006-2007 academic year, the public two-year colleges' share of new enrollment was 47% compared with 24% for the four-year universities. During the decade prior to the 2006-2007 academic year, public two-year college enrollment increased by 24% and four-year by 21%. There was also a shift toward two-year colleges for some minority groups. For public two-year colleges, the Hispanic enrollment increased by 4% and Black students 3%. These two groups' enrollment in public four-year universities decreased 2% and 3%, respectively. The changes in enrollment for other minority groups were less than 2%. For 2006-2007, the percentages of students by race who were

enrolled in a two-year college were: Hispanic 58%, Black 50%, Asian/Pacific Islander 50%, Alaskan Native 50%, and White/non-Hispanic 43%. [GAO 2007] Table 3.1 presents the increases in higher education enrollment by gender and race that was experienced in higher education in Alabama and the U.S. over the past decade.

In Alabama, the percentage of students who begin their college education in a two-year institution and transfer to a four-year university has increased greatly over the past decade, as shown in Table 3.2 [ACHE 2008a].

## 3.2STEM in Community Colleges

The lack of American students qualified to fill professional careers in basic science, technology, engineering, and mathematics has been identified as a danger to our country over the next quarter century. The American education system needs to produce significantly more scientists and engineers, including four times the current number of computer scientists, to meet anticipated demand. [USCNS/21 2001] The STEM (Science, Technology, Engineering, and Mathematics) Education Coalition along with federal and state education systems are in a heightened state of concern for the need to inspire young people, especially those from underrepresented or disadvantaged groups, to pursue careers in STEM fields [STEM 2009]. In the 1999-2000 academic year, computer and information science was the seventh most popular field of study for community college associate degrees. During the decade preceding 1999-2000, the number of associate degrees awarded by community colleges increased 21%. One of the fastest growing fields of study for associate degrees was computer and information science with an increase of 93%. [Kasper 2002] During the 16 years of the National Science Foundation's Advanced Technological Education (ATE) program, of those who earned bachelors and/or masters degrees in STEM disciplines, roughly 44% got started on their undergraduate studies at a community college [Navarro, et al. 2008]. One of the top priorities of the *State Plan for Alabama Higher Education 2009-2014* is to increase the

number of graduates in STEM from Alabama colleges and universities. This initiative includes working with two-year schools to prepare more students to transfer into four-year STEM programs. [SPAC 2009]

### 3.3Alabama Community College System

The Alabama Community College System (ACCS) consists of 22 comprehensive public community colleges and four public technical colleges; Athens State University; and workforce development initiatives such as the Alabama Industrial Development Training Institute and the Alabama Technology Network. The ACCS ensures access to education through statewide geographical locations, open enrollment, and low-cost tuition. It provides general education and other collegiate programs at the freshman and sophomore levels to prepare students for transfer to four-year institutions to complete baccalaureate degrees. [ACCS 2009a]

The Alabama Commission on Higher Education (ACHE) is the state agency responsible for statewide planning and coordination of higher education in Alabama. This includes on- and off-campus instruction programs as well as nonresident institutions[1] operating in Alabama. [ACHE 2009b] The responsibilities of the ACHE include (1) approval of new units of instruction including new institutions, mergers, branch campuses, colleges, schools, division, and departments; (2) approval of all new academic programs; (3) facilitating the planning for higher education including the development of a statewide plan; (4) reviewing and making recommendations concerning existing programs; (5) collecting and compiling information concerning higher education in Alabama; and (6) conducting studies on

---

[1] Non-resident institutions are defined as postsecondary institutions or corporations with main campuses or headquarters located outside the state that offer educational programs in Alabama [ACHE 1975].

higher education issues and making recommendation to the institutions, the Legislature, and the Governor. [ACHE 2009c]

### 3.4Alabama Articulation and General Studies Committee

A part of the ACHE, the Alabama Articulation and General Studies Committee (AGSC) was created to oversee articulation between public institutions of higher education. The AGSC developed and implemented a statewide general studies and articulation program that facilitates the transferability of coursework among Alabama community colleges and universities. The AGSC was charged by the 1994 Alabama legislature with four tasks:

1) Develop a statewide freshman- and sophomore level general studies curriculum for all public colleges and universities.
2) Develop a statewide articulation agreement for the transfer of freshmen and sophomore year credit among all public institutions of higher education in Alabama.
3) Examine the need for a uniform course numbering system, course titles, and course descriptions.
4) Resolve problems concerning the administartion and interpretation of the articulation agreement of the general studies curricula.

The AGSC has completed charges 1) and 2). A uniformed numbering system for the whole state was determined not to be needed at this time. The fourth charge is an ongoing responsibility of the committee. [AGSC 2009a]

The AGSC oversees the Statewide Transfer and Articulation Reporting System (STARS), a web-based database where students, advisors, faculty, and administrators can obtain the most current AGSC approved information. STARS allows students in Alabama public two-year colleges to obtain a transfer guide/agreement for the major of their choice. This guide/agreement directs the student through the first two years of coursework and prevents loss of credit hours upon transfer to the selected Alabama public four-year university.

The AGSC course structure for the freshman and sophomore years is divided into four areas of general studies: (I) written composition; (II) humanities; (III) natural science and mathematics; and (IV) history, social science, and behaviorial science. A fifth area of study contains pre-professional, pre-major, and elective courses that prepare students for transfer to a baccalaureate program at a four-year university. Prospective transfer students earn 41 semester hours in Areas I-IV of general studies and 19-23 semester hours in Area V. It is in Area V where a student begins a computer science program of study. One or more computer science courses are required by the two-year program with the remaining hours in Area V being determined by the institution to which the student plans to transfer. [AGSC 2009b]

The AGSC has two groups of academic committees, the General Studies Academic Committees (GSACs) and the Pre-Professional Academic Committees (PACs) [AGSC 2009c]. There is a GSAC for each of the 21 general studies disciplines in Areas I-IV. Each GSAC is responsible for the review and recommendation for approval and disapproval of new courses in the two- and four-year institutions. For each field of study in Area V, a PAC is responsible for the annual review of the discipline templates as well as the consideration of proposals from institutions for the development of new templates. Templates are used to establish degree requirements for Areas I-V for each major offered through STARS. Once a template is ratified by the AGSC, the STARS office creates a transfer guide based on the specific requirements listed in the ratified template. [AGSC 2009d]

Community college students are encouraged to use STARS to ensure that they are taking the courses necessary to continue to the Alabama four-year university of their choice. Each community college website has information about and a link to STARS as well as contact information to assist with the transfer process. In addition, each university offers an Area V webpage which contains initial information about its transfer requirements as well as contact information for transfer assistance. To obtain a STARS Guide, students provide the name of the community college they are currently attending, the area of concentration in which they would like to study, and two universities they would like to

attend. Correct information is necessary to ensure that a student has the correct guide. The guide identifies the courses and hours needed in each area (Area I-Area V) to transfer from a specific community college to a specific university. Not following the guide may result in the university rejecting credits. [STARS 2009b] The *Alabama Commission on Higher Education 2008 Accountability Report* [ACHE 2008b] communicated that the AGSC continues to give priority (1) to reviewing the approved/ratified course and templates; (2) to the education of administrators, faculty, parents, legislators, and the general public about STARS; and (3) to improve four-year institutions Area V webpage to better assist in the transfer of students from two-year to four-year institutions.

### 3.5Alabama Community College Computer Science Curricula

Nineteen of the Alabama public community colleges offer an associate and/or applied associate degree in computer science which prepare students for the work environment or for transfer to a four-year institution. Each of these institution's Area V STARS transfer guide includes one or more Computer Information Science (CIS) courses which are required for a student wanting to transfer to an Alabama public four-year university. The four-year university that is listed in a student's transfer guide will specify the additional CIS, and possible other discipline, courses that a student is required to take at the community college prior to transferring to its institution. The links for the community colleges' STARS transfer guides and the universities' Area V are presented in Appendix B.

The creation and approval of the CIS courses for the Alabama community colleage system are the responsibility of the AGSC. Course directories are available for each accademic discipline of program. The course directory provides the course number, title and course description for each approve course to create consistancy among the community colleges. [ACCS 2009c] A standardized syllabus is available for most courses [ADPE 2005].

From the course catalogs of Alabama community colleges, we obtained information on the computer science curriculum, the CIS courses taught, and the CIS course prerequisites (Appendix B).

The CIS courses offered and required for a computer science degree and the prerequisites vary by the community college. Although these courses have the same CIS course numbers in each community college catalog, some course numbers represent courses with different titles and course descriptions. For the purpose of this research, only the CIS courses related to software engineering and software development and design were selected for review. They are listed in Table 3.3., where the various course names are given for each course number. The number of community colleges using a course name in their catalog and the number listing the course as a degree requirement are indicated. The course descriptions are summarized in Table 3.4. Note that there are multiple course descriptions given for CIS 251, 252, and 255.

The Alabama Department of Post Secondary Education, which represents Alabama's public two-year college system, provides standardized syllabi. A review of the available CIS syllabi found that some contained incorrect information and inconsistencies. Of particular interest to this research is that the syllabi for CIS 251 *C++ Programming* and CIS 255 *Java Programming* do not mention object-oriented programming [ADPE 2005].

The inconsistances in the CIS courses offered by Alabama public community colleges are evident in the course descriptions shown in Table 3.4. CIS 251 is listed as a required course in Area V of four four-year universities; yet, it has three different course descriptions from the community college catalogs. The differences in the course descriptions and computer science curricula offered at the community colleges convey significant inconsistancy among institutions. This can put some students at a disadvantage in the workplace, but, more in line with this research, students can be at a disadvantage when entering a baccalaureate program at a public four-year university.

Although the CIS course numbers are consistent in the community college catalogs, the corresponding course numbers in the four-year university Area V requirements, shown in Table 3.5, are most likely not the same number. A student is dependent on a STARS Tranfer Guide to provide the

connectivity of a course taught at a community college with the cooresponding course needed at the four-year institution. It should be noted that in most cases the Area V webpages indicate that assistance of an advisor at the community college or university is needed to ensure that a tranfer guide is complete and correct.

**Table 3.3 CIS courses in community college catalogs related to software engineering and software development and design**

| CIS No. | Course Title(s)<br>*All 3 semester hours* | Cat | Deg Req | Prerequisite | Transfer Code |
|---|---|---|---|---|---|
| 110 | Intro to Computer Logic & Programming | 15 | 7 | Varies by college | C |
| 150 | Computer Program Logic<br>Intro to Computer Logic & Programming | 1<br>1 | -<br>- | Varies by college | |
| 185 | Computer Ethics | 6 | 2 | Varies by college | C |
| 191 | Intro to Computer Programming<br>Intro to Programming Concepts<br>Intro to Computer Programming Concepts<br>Intro to Computer Science | 1<br>1<br>5<br>3 | -<br>-<br>1<br>- | Varies by college | B |
| 192 | Advanced Computer Programming Concepts | 4 | - | Varies by college | C |
| 201 | Intro to Computer Programming<br>Intro to Computer Programming Concepts | 1<br>2 | 1<br>1 | Varies by college | C |
| 211 | Basic Programming | 1 | 1 | | |
| 212 | Visual Basic<br>Visual Basic Programming<br>Visual Basic Programming (VisualBasic.net) | 4<br>16<br>1 | 2<br>6<br>1 | Varies by college | B |
| 213 | Advanced Visual Basic Programming<br>Adv Visual Basic Prog(Adv VisualBasic.net) | 15<br>1 | 3<br>1 | Varies by college | C |
| 251 | C Programming<br>C++ Programming<br>C++ Programming Language | 4<br>13<br>1 | 1<br>7<br>1 | Varies by college | B |
| 252 | Advanced C Programming<br>Advanced C++ Programming | 2<br>9 | 2<br>2 | Varies by college | C |
| 255 | JAVA Programming | 14 | 4 | Varies by college | B |
| 256 | Advance JAVA | 6 | 1 | Varies by college | C |
| 281 | System Analysis & Design | 10 | 4 | Varies by college | C |
| 285 | Object-Oriented Programming | 9 | 3 | Varies by college | B |

| Transfer Code | |
|---|---|
| | Code A: AGSC-approved transfer courses in Areas I-IV that are common to all institutions.<br>Code B: Area V deemed appropriate to the degree and pre-major requirements of individual students.<br>Code C: Potential Area V transfer courses subject to approval by respective receiving institutions. |
| Cat (Catalog) | Number of community colleges with course in catalog |
| Deg Req (Degree requirement) | Number of community colleges with course as a degree requirement |

The first computer science courses taught in a computer science baccalaureate curriculum are usually a programming course using a specific programming language. From the catalogs of the four-year

universities (Appendix B), some programs specify C++, some specify Java, and others do not specify a language. In Table 3.4, CIS 285 is described as an object-oriented programming course, and CIS 191 is described as structured programming course. Neither of these courses indicate a programming language in the community college course descriptions. Inconsistency in this type of information and the Area V course numbers listed in Table 3.5 that universities that teach using Java or C++ do not specifically list the corresponding community college course in their Area V.

Through the creation of the Articulation and General Studies Committee and STARS, the Alabama Commision of Higher Education has laid the ground work for a smooth transistion from a two-year community college program to a four-year baccalaureate program. However, as noted in the previous discussion, there are significant inconsistencies in course numbering, titles, and descriptions that can interfere with an effective articulation system.

**Table 3.4 Concepts, Techniques, and Requirements in CIS Course Numbers and Descriptions**
(Alabama Community College Course Catalogs. See Appendex A.)

| | |
|---|---|
| 110 | Logic, design, and design solving techniques<br>Flowcharts, structure charts, and pseudocode |
| 150 | Logic, design, and design solving techniques<br>Flowcharts, structure charts, and pseudocode |
| 185 | Computer ethics issues |
| 191 | Algorithm approach to problem solving via design and implementation of programs in selected programming languages<br>Structured programming techniques: input/output, conditional statements, loops, files, arrays, structures, simple data structures<br>Write programs |
| 192 | Algorithm specifications<br>Structured programming<br>Data representation<br>Searching and sorting<br>Recursion<br>Simple data structures<br>Language description<br>Problem testing<br>Develop problem solving skills<br>Programming projects |
| 201 | Fundamental programming concepts<br>Problem solving and algorithms<br>Design tools<br>Programming structures<br>Variable data types and definitions<br>Modularization<br>Selected program languages<br>Develop programs |
| 211 | Intro to BASIC programming lanuage<br>File processing<br>Internal sorts<br>Data structures<br>Programming projects |
| 212 | BASIC program using graphical user interface<br>Graphical user interface<br>Advanced file handling<br>Simulation<br>Programming projects |
| 213 | Continuation of BASIC programming with emphasis on understanding techniques and procedures for developing projects using an object oriented language |
| 251 | (*Course number has multiple descriptions*.) |
| | Intro to C programming languageAlgorithm approach to problem solvingStructure programmingUsing functions and macrosSimple data structuresFile input and outputProgramming projects |
| | Intro to C++ programming lang.<br>Object-oriented programming<br>Problem solving and design<br>Control structures<br>User interface construction<br>Document and program testing |
| | Intro to C++ programming language<br>First course in problem solving<br>Program style<br>Algorithm<br>Data structuring<br>Modularization<br>Programming projects |

**Table 3.4**. (cont.). Concepts, Techniques, and Requirements in CIS Course Descriptions with CIS Course Numbers

| Course | Description |
|---|---|
| 252 | (*Course number has multiple descriptions*.) |
| | Continue C programming<br>Improvement of application and systems programming<br>Memory management<br>C library functions<br>Debugging<br>Portability and reusable code<br>Programming projects |
| | Advanced object-oriented programming<br>Advanced program development in context of an object-oriented language<br>Object-oriented analysis, encapsulation, polymorphism (operator and function overloading), information hiding, abstract data types, reuse, dynamic memory allocation, and file manipulation<br>Develop hierarchical class structure<br>Implementation of an object-oriented software system |
| | Introduce the C++ programming language<br>Problem solving and design, control structures, objects and events, user interface construction, documentation and program testing |
| | Continue C++ programming<br>Improvement of application and systems programming<br>Memory management,<br>C library functions<br>Debugging<br>Portability and reusable code<br>Programming projects |
| 255 | (*Course number has multiple descriptions*.) |
| | Intro to the Java programming language<br>Object-oriented programming<br>Webpage applet development<br>Class definitions<br>Threads, events, and exceptions<br>Programming projects |
| | Intro to Java programming language<br>Hands-on programming assignments<br>Java program structures<br>Java's built-in class libraries<br>Data types, control structures, and object-oriented programming |
| 256 | Advanced Java programming language<br>Sun's Swing GUI components, JDBC, JavaBeans, RMI, servlets, and Java media framework<br>Programming projects |
| 281 | Study of contemporary theory and systems and design<br>Investigating, analyzing, designing, implementing, and documenting computer systems<br>Programming projects |
| 285 | Advanced object-oriented programming in the context of an object-oriented language, such as C++ or Java [Note: Not all include the language list.]<br>Object-oriented analysis and design, encapsulation, inheritance, polymorphism (operator and function overloading), information hiding, abstract data types, reuse, dynamic memory allocation and file manipulation<br>Develop a hierarchical class structure necessary to the implementation of an object-oriented software system |

**Table 3.5 Area V required CIS courses for transfer to four-year university**

| | Area V Website Provider | BS Degree | Area V Required Courses |
|---|---|---|---|
| | STAR's Transfer Guide | CS | CIS 251 or 285<br>CIS 251 - C Programming (3sh)<br>CIS 285 - OOP (3sh) |
| ASU | Alabama State U | CS | CSC 210, 211 or 447<br>CSC 210 - Intro to CS (3sh)<br>CSC 211 - Programming Concepts, Standards, & Methods (4sh)<br>CSC 447 - OOP (4sh) |
| Athens | | CS | prereq courses<br>Microcomputing Apps (3sh)<br>C++ Programming (3sh)<br>Computer Programming courses (6sh) |
| AU | Auburn U | CS, SE | see Engineering Articulation Guide |
| AUM | Auburn U Montgomery | Math(CS) | CSCI 1200, 2000<br>CSCI 1200 - Scientific Prog<br>CSCI 2000 - Structured Prog |
| JSU | | CS | CS 201, 230, 231 [CIS 146, 201, 251] (3sh)<br>CS 201 - Intro to Information Tech [CIS 146]<br>CS 230 - Fund. Of Computing [CIS 201]<br>CS 231 - Computer Programming I [CIS 251] |
| Troy | Troy U | CS | CIS 146 (3sh) |
| UA | U of Alabama | CS | CS 114, 116 [CIS 191, 193]<br>CIS 191 - Intro to Computer Programming (3sh)<br>CIS 193 - Intro to Computer Programming Lab (1sh) |
| UAB | U of Alabama, B'ham | CIS | CIS 285 - OOP (3sh) |
| UAH | U of Alabama, Huntsville | CS | CIS 285 or 251 - Intro to C++ Programming (3sh)<br>CIS 285 - OOP<br>CIS 251 - C Programming |
| UNA | U of North Alabama | CS | CS 155 [CIS 191 or 251] (3sh)<br>CIS 191 - Intro to Computer Programming<br>CIS 251 - C Programming |
| USA | U of South Alabama | CS | CIS 115 [CIS 197 or 211 or 212] (3sh)<br>CIS 197 - Adv Commercial Software<br>CIS 211 - Programming Concepts, Standards, & Methods<br>CIS 212 - Intro to Visual Basic |

## 4SIGCSE 2011 Birds-of-a-Feather: Introducing Software Engineering Principles in the First Two Years of Computer Science Education

The purpose of the Birds-of-a-Feather session was to identify and discuss software engineering concepts that can be pushed down into the introductory-levels, CS1 and CS2. For this discussion, CS1 and CS2 referred to the first two courses in the introductory sequence of computer science. CS1 and CS2 were further defined by a list of suggested teaching topics for each course. [Hundley 2011]

The eleven participants in the session included three faculty from two-year academic institutions, seven from four-year institutions, and one non-academic participant from around the United States. Small groups were formed by the length of the computer science program, i.e., two- or four-year.

Two sets of wall charts, one for two-year and one for four-year, with the identifying teaching topics printed at the top were used to collect information for the large group discussion. The groups were asked to discuss one course level, CS1 or CS2, at a time and record their ideas for software engineering concepts that might be used.

There were one two-year and two four-year faculty groups that brainstormed about what software engineering principles and concepts could be taught in CS1 and CS2. Each four-year groups' responses were listed separately with duplicate ideas marked with a checkmark on the four-year second part of the first year chart. The information from the wall charts is presented in Figure 4.1.

At the end of the allotted time, the large group discussed the topics written on each wall chart and the pros and cons of how each topic may be used in teaching CS1 and CS2. At the bottom of the Figure 4.1, the "During Discussion" sections list topics and clarifications from the large group

discussion. The group agreed that testing and conventions were the top priority topics to include CS1 and CS2. Figure 4.2 shows some of the comments made during the large group discussion.

The participants of this session expressed much interest in what and how software engineering principles can be introduced early in the computer science curriculum. During the discussion, several software engineering principles and concept that can be introduced at the introductory-level were identified. The principles and concept list on the wall sheets (Figure 4.1) were used when designing online survey that was sent to two- and four-year faculty in Alabama public community colleges and universities. The information collected was also considered to determine which software engineering knowledge areas to include in the set of teaching modules.

| **2-year programs:**<br>**Intro course for CS students [CS1]**<br>-I/O -control structures<br>-syntax -functions/methods<br>-foundation OO<br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br>follow a requirement spec<br>create a requirement spec?<br>testing, what form?<br>code of ethics<br>documentation<br>conventions/ standards | **2-year programs:**<br>**2nd part of 1st year for CS students [CS2]**<br>-data structures<br>-building fundamental algorithms<br><br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br>create requirements<br>testing<br>architectural design<br>documentation<br>following conventions |
|---|---|
| **4-year programs:**<br>**Intro course for CS students [CS1]**<br>-I/O -control structures<br>-syntax -functions/methods<br>-foundation OO<br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><u>1st group</u><br>Assumption<br>starting point: not OO<br>Things we do now<br>pair programming<br>code reviews/ readings*<br>test-driven development/ unit test/ negative test<br>time tracking**<br>validation<br>Could do?<br>source code control***<br>tools/ collaboration<br><br><u>2nd group</u><br>black box test plans<br>documentation<br>presentations:<br>code, project plans, completed project<br>flow charts<br>pseudocode<br>team projects<br>student defined project<br><br><u>During discussion</u><br>*give buggy program at end of semester<br>**PSP<br>*** CS2? | **4-year programs:**<br>**2nd part of 1st year for CS students [CS2]**<br>-data structures<br>-building fundamental algorithms<br><br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><u>1st group</u><br>performance analysis (algorithms)<br>testing (black-box and white-box)<br>✓interface design<br>✓documentation<br>✓design<br>plan<br>inspections (requirements, design, code)<br><br><u>2nd group</u><br>static analysis<br>UML<br>design process alternatives<br>validation<br>modularity => scale, complexity<br>program management<br><br><u>During discussion</u><br>basic design patterns<br>design check list<br>class diagram description<br>interfaces<br>no inheritance?<br>no polymorphism? |

**Figure 4.1. SIGCSE 2011 Birds-of-a-feather small group results**

**Prioritites from the 2-year chart:**

- Easy to employ and supportable: (1) conventions, (2) documentation, and (3) test-driven development (TDD).
- An opposing view suggested TDD should be introduced early because tests do not lie, i.e. comments are lies meaning not executable.
- Pair programming can work early but the partners must switch frequently. This can include small group programming.

**Code review:**

- CS1: instructor (so students aren't being mean to each other) provides code (with bugs); students review improve and CS2: students review own code
- Done in industry and most new hires haven't experienced
- Validation: students have to read each other's code (code can't have comments). Discuss if code satisfies requirements

**Time tracking:**

- Industry basis for estimation
- Later: track time + interruptions

**Source control:**

- In CS2; too early in CS1 due to overload of material
- Consensus: source code control is iffy. Students use it once at submission time.

**Breadth-first approach:** Expose to a lot of practices at shallow level

**UML:** Can be introduced early in a simplified form

**Engineering approach:**

- CS1and CS2 are really engineering 1 and engineering 2
- Teach problem solving as engineering approach
- Engineering design allows multiple alternatives which SWE typically do not do
- Keep students away from keyboard, i.e. plan before coding

**Interfaces:** Introduce early

**Modularity:** Typically enforce modularity but students revert to giant main when not enforced

**Good practices:** Many software engineering practices taught later in curriculum; students ask why they weren't exposed to them earlier

**Figure 4.2. SIGCSE 2011 Birds-of-a-feather large group discussion**

# 5Survey of Software Engineering Principles and Concepts

For the research, the preliminary information about the Alabama public community colleges' computer science programs was obtained from the school websites. Additional information about the two-year programs was available on the Alabama Community College System and Alabama Articulation and General Studies Committee websites [ACCS 2009b, AGSC 2009a]. This information is included in Chapters 2 and 3 of this document. To collect more current and specific information about what software engineering principles and concepts are being taught these two-year programs, an online survey was created using Qualtrics Survey Software [Qualtrics 2011]. An invitation email containing the URL of the survey was sent to the computer science faculty at the 19 two-year schools that have computer science programs. To obtain comparative data, the invitation email was also sent to faculty who teach CS1/CS2 level courses in the six four-year public universities that offer a computer science program.

Because of the discrepancy of the use of CS1, CS2, and CS3 in teaching computer science, information was requested for the first three semesters the computer science courses in the programs. The questions centered on the educational objectives of each respondent's school's educational objectives for teaching five software engineering knowledge areas and a list of software engineering terms and concepts. The survey also collected information on the integrated development environment (IDE) and computer programming language used during the first three semesters.

The respondents were asked about personal familiarity with the Association for Computing Machinery (ACM), the IEEE computer Society (IEEE-CS), and/or Two Year College Education

Committee computer science curriculum guides. They were also asked  where students who complete the computer science go after graduations.

## 5.1Survey Results

Because the responses were anonymous, it is not possible to track whether all two-year and four-year programs are represented in the survey results. There were nine responses from four-year programs and 20 responses from two-year programs. Although the sample is small, the accumulated information does provide insight into the Alabama state public two- and four-year computer science programs.

### *5.1.1Software Engineering Knowledge Area Results*

For each software engineering knowledge area and the software engineering term and concepts, the respondents  were asked to select the level of education objective expected for a student who finishes the first two semesters of their computer science program. The Bloom's Taxonomy was used as guide for the education objective choices: remember, understand, and apply [Bloom and Krathwohl 1956]. The fourth response choice was "not used in courses." The results analysis of each software engineering knowledge areas and the software engineering terms and concepts results is presented in two charts: (1) all responses including "not used in teaching" (See figures 5.1 and 5.3.) and (2) only the programs that use the software knowledge area and terms in teaching (See figures 5.2 and 5.4.).

These results are represented by horizontal bar charts to show the relative levels of the teaching objectives used. In each figure, the responses are presented for the two- and four-year programs. The graphs show the data with and without the "not used in courses" responses. The latter provides a better comparison of the teaching objectives. For each item, the average for the weighted responses was used as data for the charts.

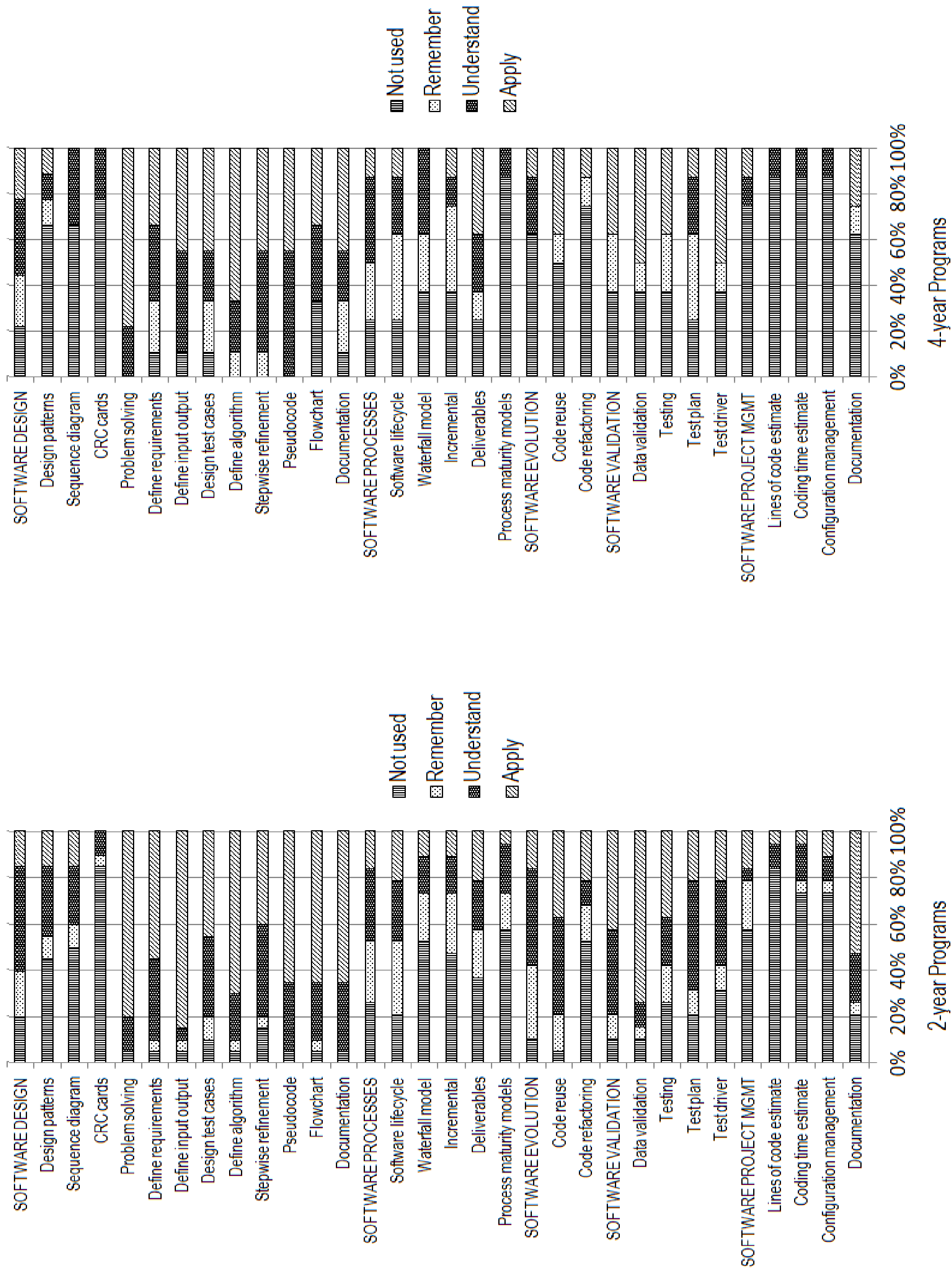


**Figure 5.1. Software engineering knowledge areas included in the survey and education objectives in all programs**

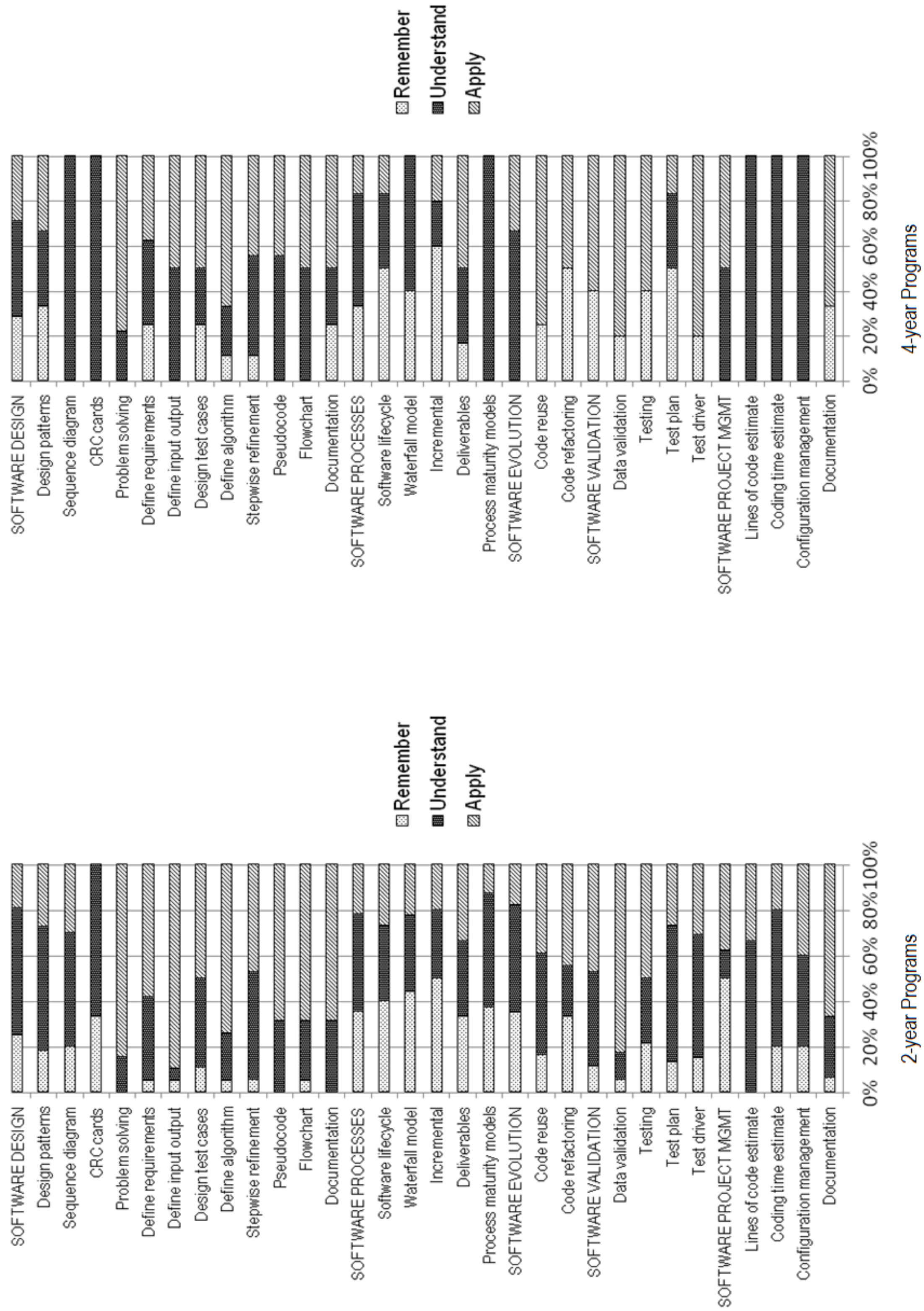


**Figure 5.2. Software engineering knowledge areas included in the survey and education objectives for the programs that teach the principle**

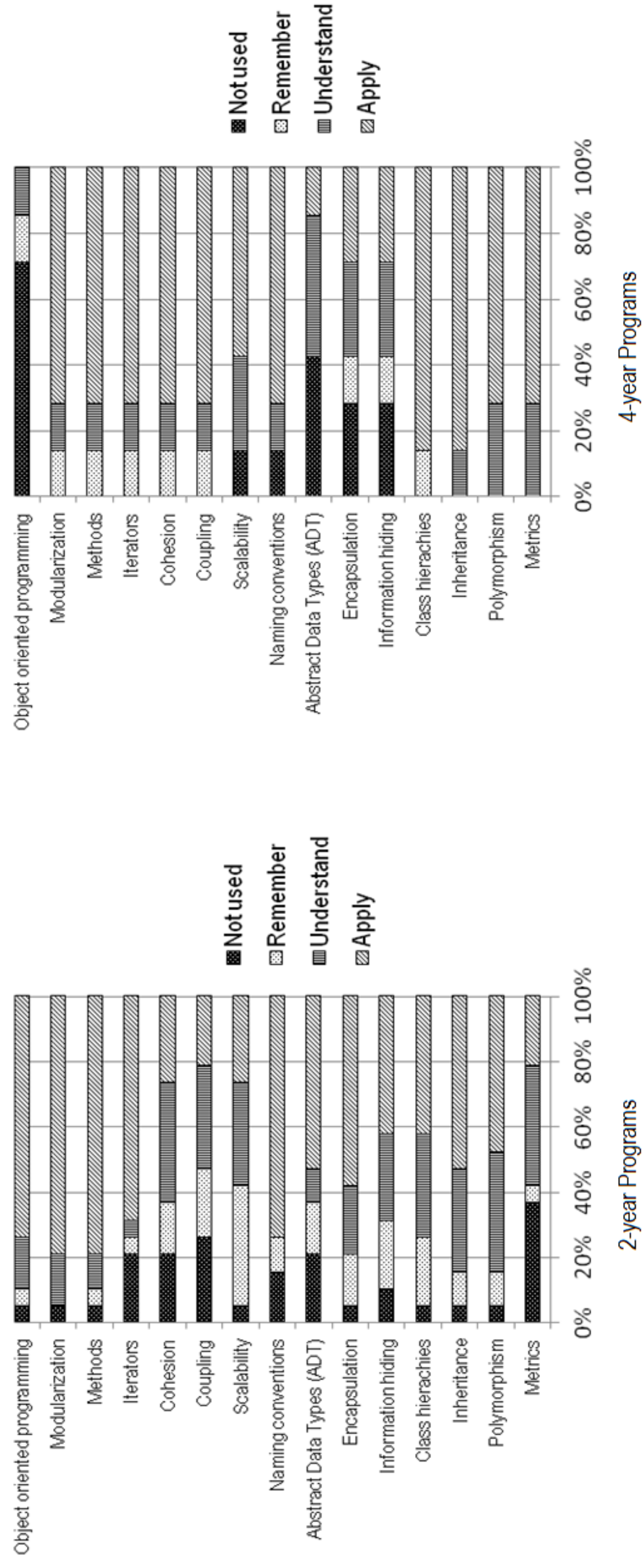


**Figure 5.3. Software engineering terms and concepts included in the survey and education objectives in all programs**

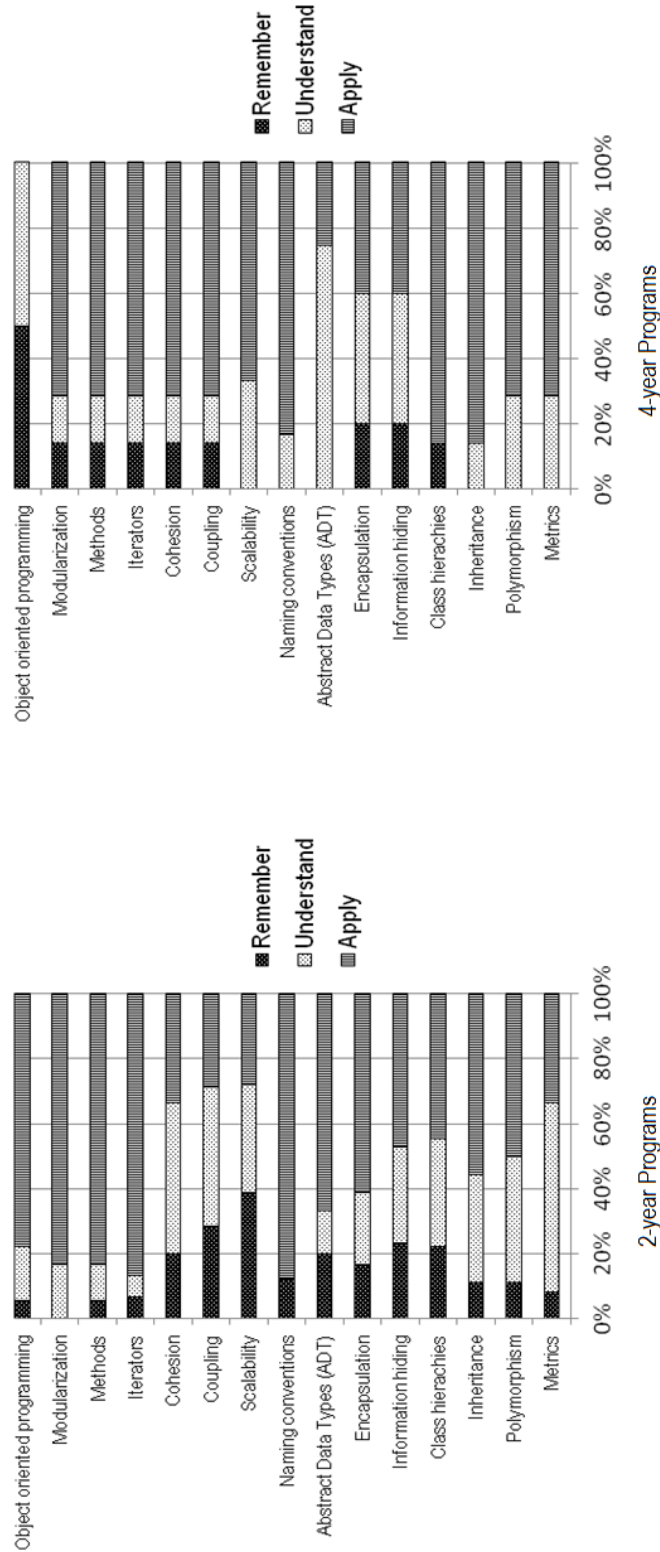


**Figure 5.4. Software engineering terms and concepts included in the survey and education objectives in programs that teach the concept**

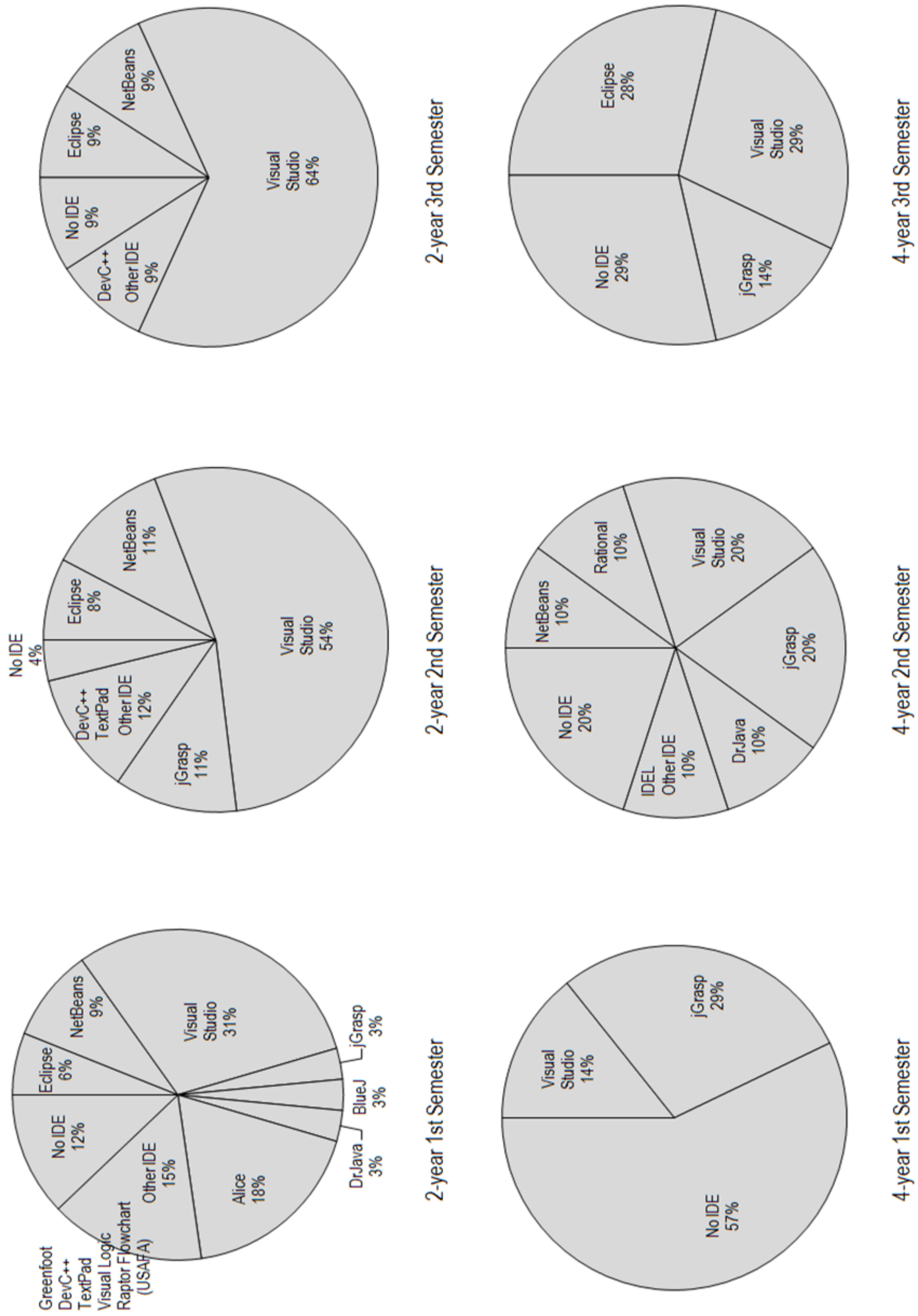


**Figure 5.5. Integrated development environments (IDEs) used in respondents' programs**

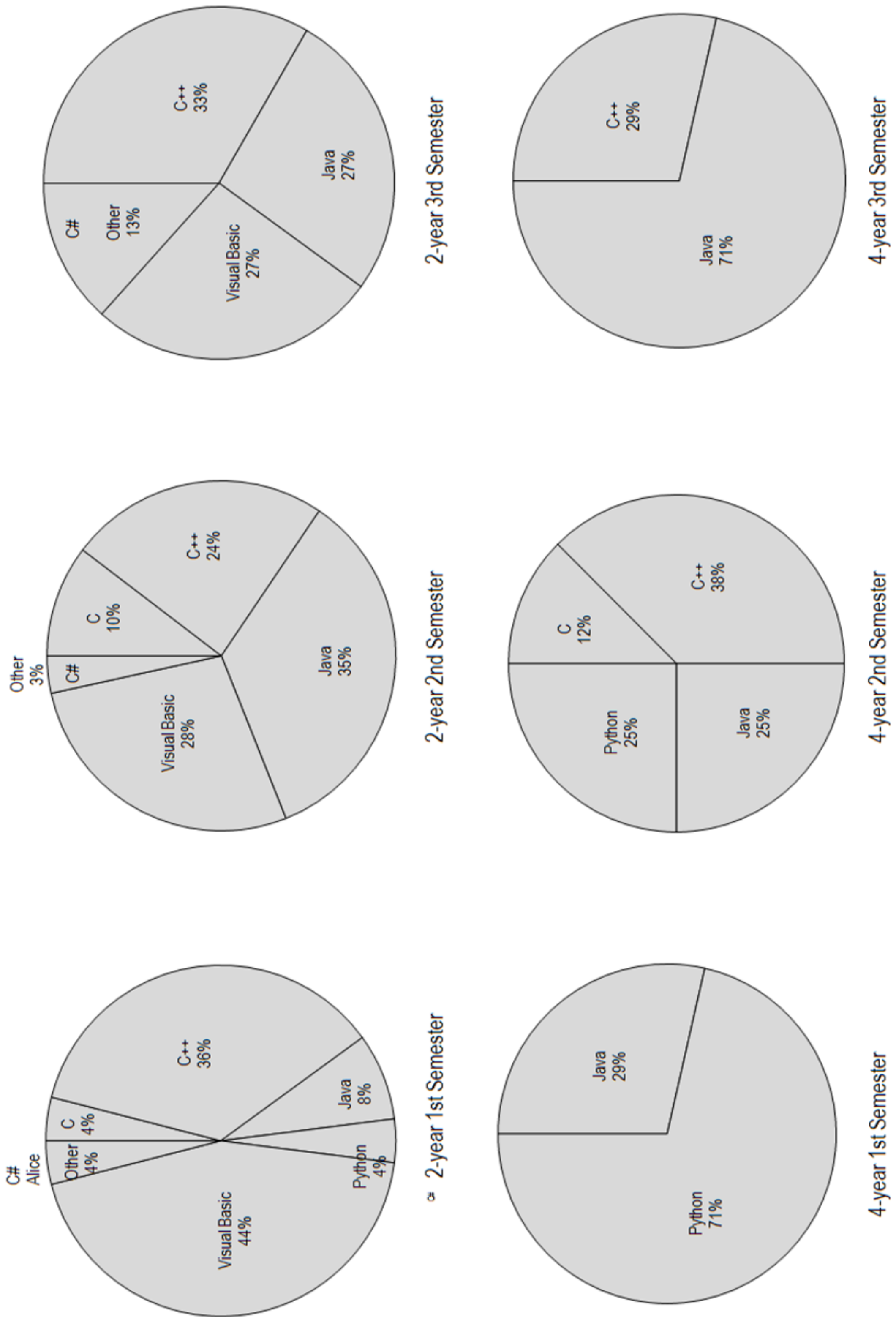


**Figure 5.6. Computer programming languages taught in respondents' programs**

### *5.1.2Integrated Development Environment and Programming Language Results*

The survey included a list of Integrated Development Environments (IDEs) and Programming Languages. Each list included an "other" option with a text box for recording IDEs and languages not listed in the survey. The choice of no IDE was also included. The respondents indicated which IDE and programming language was used in the first, second, and third semesters of the computer programming courses.

The pie charts display the IDEs and languages used in each of the three semesters for the two- and 4-year programs. These charts are shown in Figures 4.7 and 4.8 with lists of text responses given for the "other".

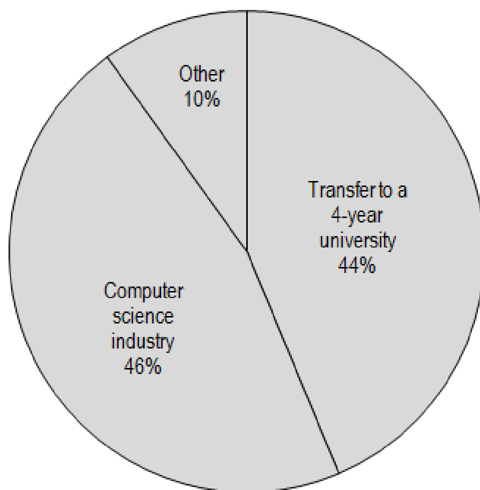


**Figure 5.7. Where do two-year graduates go**

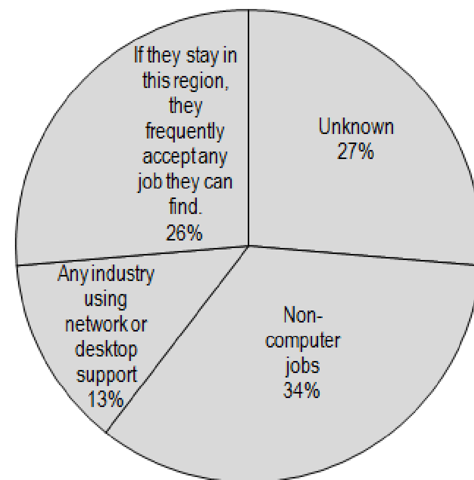


**Figure 5.8. Other jobs for two-year graduates**

### *5.1.3Other Results*

The survey asked the two-year respondents where the students who completed their computer science program went after graduation. The responses were given as percentage for the categories: four-year university, computer science industry, and other which included a text box. These results are presented in a pie chart in Figure 5.7. A second pie chart, Figure 5.8, shows the proportions of the "other" option text responses, only.

In the final section, the respondents recorded their familiarity with the following curriculum guides created by the ACM, IEEE-CS, and/or Two-Year College Education Committee. These responses are presented in Figures 5.9 and 5.10.

- Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science
- Computing Curricula 2009: Guidelines for Associate-Degree Transfer Curriculum in Computer Science
- Computing Curricula 2001: Computer Science
- Computer Science Curriculum 2008: An Interim Revision of CS-2001

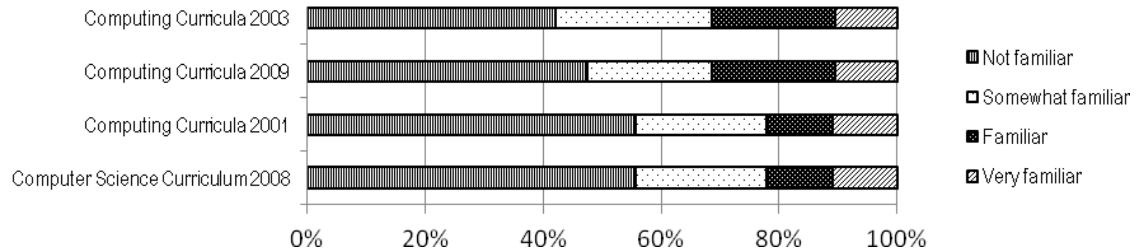


**Figure 5.9. Two-year respondents familiarity with curriculum guides**

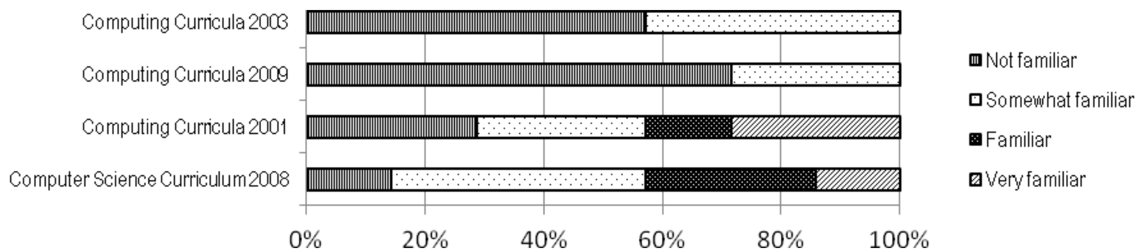


**Figure 5.10. Four-year respondents familiarity with curriculum guides**

5.2Survey Results Summary

The samples were small and the completeness of the coverage of the two- and four-year computer science programs in Alabama is not verifiable. The previous charts and graphs of the survey data are given to allow the readers to extract their own conclusions from the information gathered.

The responses indicated that many programs do not include the teaching software engineering principles and concepts at the introductory-level. There is a signification lack of familiarity with the computing curricula guidelines provided by the ACM, IEEC-CS, and Two-Year College Education Committee. This can lead to the lack of attention giving to teaching software engineering principles and concepts in the introductory courses. The lack of familiarity could mean a lack of familiarity of terms in the survey which could have resulted in the high number of responses of "Not Used in Course" for the software engineering knowledge areas.

An attempt was made to extract some statistical analysis from the survey responses. Because the data collected was determined to not be normally distributed, the Wilcoxon signed rank test was use attempt to statistically compare the two- and four-year computer science programs. The Wilcoxon signed rank test null hypothesis was that the population mean ranks did not differ, and the alternative hypothesis was that the population mean ranks differ. Because of the small sample sizes, the resulting information may or may not be conclusive.

**Table 5.1. Results of Wilcoxon signed rank test (*p* values)**

| | Usage average including "not used" | Usage average without "not used" | Percent "not used" |
|---|---|---|---|
| All KAs and subtopics | ***0.0007*** | 0.1641 | ***0.0047*** |
| Software Design | 0.0652 | ***0.0134*** | 0.3384 |
| Software Process | 0.9063 | 1.0000 | 0.6875 |
| Software Evolution | 0.2500 | 0.5000 | 0.2500 |
| Software Validation | 0.1250 | 0.4375 | 0.0625 |
| Software Project Management | 0.0625 | 0.8750 | 0.0625 |
| | | | |
| Terms and Concepts | 0.1038 | 0.0617 | 0.3243 |

The computations were done using the signrank function in MATLAB which compares two vectors of values and returns two values: the *p* value and 0 or 1 where 1 indicates a rejection of the null hypothesis at the 5% significance level. Shown in Table 5.1, most of the test results did not reject the null hypothesis. For "All KAs and subtopics" that included the "not used" response option and the percent of respondents not using the KAs in teaching, the results rejected the null hypothesis indicating that there was a difference in the number of two- and four-year respondents who did not include software engineering knowledge areas in teaching. The only other difference appeared in those who included software design in their teaching.

# 6Teaching Software Engineering Course

A special topics graduate course, Teaching Software Engineering, was taught in Summer 2011 in the Auburn University Computer Science and Software Engineering department. The course examined software engineering from an instructional perspective. Its purpose was to give graduate students exposure to explaining fundamental software engineering concepts to those new to the field and to explore which software engineering principles can be introduced into the first two computer science courses, CS1 and CS2.

The rationale of the course was "Postgraduate instruction traditionally focuses on developing advanced subject specialty skills, offering research experiences, and fostering methods of disciplined thought. Graduate students look to careers in higher education or industry but never receive training on how to explain the complex concepts of engineering software to students, coworkers, supervisors, subordinates, etc. This course was designed to provide insight into how to teach software engineering concepts at the introductory-level." See the course syllabus in Appendix D.

The course began with (1) a discussion of the difference between computer science and software engineering [Shackelford, et al. 2005], (2) an examination of Software Engineering Body of Knowledge Project (SWEBOK) [Tripp, 2004], and (3) an examination of model curricula for software engineering and computer science [Campbell, et al. 2005, Hawthorn, et al. 2009, LeBlanc and Sobel 2004]. Eight SWEBOK software engineering knowledge areas were selected for closer examination.

- Software engineering process
- Software construction

- Software design
- Software testing
- Software quality
- Software requirements
- Software configuration management
- Software engineering management

Each week, course participates focused on a specific knowledge area. The presentation and class discussion were lead by a student with a video contribution from one or more distance-learning students. The discussion began with a brief primer of the knowledge area to identify its principal elements and the prerequisite skills required to apply it.

Outside readings were used to identify sources of information on how to explain the knowledge area and what others have done to teach it. Emphasis was also given to identifying instructional pitfalls to avoid when explaining the knowledge area and how to teach it in the context of the novice instructor, the CS1/CS2 student, and the adult learner.

The availability of software tool support for teaching the material in a knowledge area was presented and discussed via video by distance-learning student(s). Sample learning activities for teaching the key concepts of the knowledge area in the CS1/CS2 were identified.

Artifacts from the course included teaching modules for each knowledge area. These are explained in Chapter 5 and can been seen in Appendix E. For additional requirements, students were to (1) contribute to a class discussion board about the material, (2) maintain a journal of teaching reflections over the course of the semester and (3) interview an instructor and a novice student or lay person on their respective perspectives of a software engineering knowledge area.

## 7Teaching Software Engineering Principles in Introductory Computer Sciences Courses Workshop

The culmination of the efforts of the Teaching Software Engineering course was a workshop for nearby faculty who teach introductory computer science. At the workshop, students from the summer class presented a teaching module for each knowledge area.  It should be noted that because the workshop was on Saturday, a non-class day, attendance was not required. However, the students volunteered to attend, give the presentations, and participate in the discussions.

As each teaching module was presented, the attendees were asked to make comments and evaluate the module using an evaluation form. Each module was rated on five different aspects:

- The module as a whole covers as much of SWEBOK guidelines as it should for teaching the CS1 and CS2 level courses.
- The outline is realistic in covering the KA for the CS1 and/or CS2 teaching level.
- The outline is usable in teaching the KA at the CS1 and/or CS2 level.
- The suggested course activities are realistic for teaching CS1 and/or CS2 students.
- The suggested course activities are usable for teaching CS1 and/or CS2 students

The evaluations for all modules were rated Strongly Agree, Agree or Neutral. Most of the comments were positive and agreeable with the material in the modules. The teaching of software engineering principles early was considered beneficial for the students and allowed for time students to mature in using the skills during their academic career. Caution was expressed by some that some modules may be too large and could interfere with the normal contents of the curriculum being taught. It

was suggested that some activities be used with multiple modules for more continuity and ease of covering more without too much extra work.

The suggestions were used to edit the modules. The revised modules are presented in Chapter 8 and Appendix E of this document.

# 8Curriculum modules

In the summer, 2011, the Computer Science and Software Engineering Department at Auburn University offered a special topics graduate course, Teaching Software Engineering. The purpose of the course was to examine software engineering from an instructional perspective and to give students an exposure to explaining fundamental software engineering concepts to those new to the field. Traditionally, postgraduate instructions focus on developing advanced subject specialty skills, offering research experiences, and promoting methods of disciplined thought. Graduate students look to careers in higher education or industry but do not receive training on how to explain complex concepts of engineering software to students, coworkers, supervisors, subordinates, etc. This course was designed to provide insight into how to teach software engineering concepts at the introductory-level.

During the course, eight knowledge areas of software engineering as established in SWEBOK [Tripp, et al. 2004] were examined. Each class was led by a student who presented and led the discussion about the knowledge area. The leader initiated small group discussions to identify possible teaching activities that would enhance the learning experience. Other students identified and presented automated tools that can assist the teaching and learning of the knowledge area. The software tools were considered with respect to functionality, acquisition expense, and effort to introduce into the classroom environment.

From the class discussions and further research, the students developed curriculum modules for teaching a software engineering knowledge area at the introductory-level. The modules include recommendations of which sub-topics of the knowledge area can be integrated into CS1 and CS2,

teaching activities to reinforce learning the topics, and suggested tools for teaching and learning. The these modules were considered when creating the curriculum modules included in this research.

## 8.1Software Process Curriculum Module

The Software Process Curriculum Module is shown in Figures 8.1-8 as an example of those produced during the course. The additional curriculum modules are included in Appendix E. It should be noted that the purpose of these teaching modules is to demonstrate how software engineering knowledge area and principles can be imprinted into teaching computer science at the CS1 and CS2 levels not to replace material and topics that are necessary in the curricula. It is hoped that the information presented in the module will enhance the learning experience of the students.

Each module begins with a module description of and the philosophy for teaching the knowledge area as shown in Figure 8.1. The description and subtopics are given as established by SWEBOK [Tripp, et al. 2004]. The subtopics of the knowledge area are listed and mapped as considered being appropriate to be introduced into CS1 and/or CS2. The philosophy explains the importance for including the knowledge area at the introductory computer science curricula.

**Software Process Curriculum Module**

**Preface**

The purpose of these teaching modules is to demonstrate how software engineering knowledge area and principles can be imprinted into teaching computer science at the CS1 and CS2 levels. It is not intended to replace material and topics that are necessary in the curricula. It is hoped that the information presented in this module will enhance the learning experience of the students.

**Module Description**

This module presents an introduction software process. Software engineering process refers to the technical and managerial activities that are performed during software acquisition, development, maintenance, and retirements. It is concerned with meta-data: definition, implementation, assessment, measurement, management, change, and improvement. (SWEBOK)

In SWEBOK, Software Engineering Process is divided into the sub-knowledge area topics show below. This module will provide assistance for introducing the some topics at the CS1 and CS2 levels.

- Process Implementation and Change
  - Process Infrastructure
  - Software Process Management Cycle
  - Models for Process Implementation and Change
  - Practical Considerations
- Process Definition
  - Software Life Cycle ModelsCS1
  - Software Life Cycle ProcessesCS1
  - Notations for Process Definitions
  - Process Adaptation
  - Automation
- Process Assessment
  - Process Assessment Models
  - Process Assessment Methods
- Process and Product Measurement
  - Process Measurement
  - Software Products MeasurementCS2
  - Quality of Measurement Results
  - Software Information Models
  - Process Measurement Techniques

**Philosophy**

Software process is an integral part of software development. It can assist in learning and teaching by providing:

- a set of steps for approaching software development
- a mechanism for accountability
- an engineering mindset of problem solving
- a factory of artifacts
- a reminder of best practices
- a communication tool

**Figure 8.1. Software process curriculum module**

The learning outcome, Figure 8.2, identifies the expected cognitive and performance skills students can obtain from using the module to teach the knowledge area. The prerequisite knowledge expected of the student before starting the module is provided to assist the correct placement of the module.

**Outcomes**

Through the material covered in this module, students should:

- Identify a problem, define solutions, and develop algorithms to attain the optimal solution.
- Recognize that software systems can be produced according to a systematic model.
- Explain alternative ways to organize software development efforts
- Describe the software engineering process using standard metrics.

**Prerequisite Knowledge**

The CS1 level of subject matter presented in this module requires no computer science prerequisite. CS1 is the prerequisite for CS2 level.

**Figure 8.2. Software Process Curriculum Module (continued)**

A teaching outline, Figure 8.3, is included in the curriculum module to provide a guide for teaching the knowledge area topics and subtopics. The brief outline is an overview and a guide for presentation slides. The annotated outline, Figure 8.4, provides more substance to assist the instructor's discussion about the knowledge area and the inclusion teaching activities in the curriculum module in the teaching process.

**Outline**

1) CS1
   a) Introduction
      i) Software Engineering
      ii) Software Process
      iii) Software Process Helps
      iv) A Software Engineering Process
   b) A Problem Solving Approach
   c) Use CS1 Activity 1

2) CS2
   Recap the CS1 introduction
   a) Software Metrics
   b) Vocabulary
      i) Measure
      ii) Measurement
      iii) Metrics
      iv) Indicator
   b) Measurable Attributes of Software Engineering
   c) Measuring Individual Performance - CS2 Activity

**Figure 8.3. Software process curriculum module (continued)**

**Annotated Outline**

1) CS1
   a) Introduction
      i) Software Engineering
         Applies a systematic, disciplined, quantifiable approach (or process) to the development, operation, and maintenance of software.
      ii) Software Process
         The sequence of steps to develop or maintain software
      iii) Software Processes Help
         (1) Boost the probability of product quality
         (2) Identify the principle activities of doing a job
         (3) Separate routine from complex tasks
         (4) Facilitate tracking and measuring performance
         (5) Provide orderly mechanism for learning
         (6) Establish corporate memory
         (7) Create a defined baseline for improvement
         (8) Put everyone on the same page
      iv) A Software Engineering Process
         (1) Define the function of the program
         (2) Sketch out a design
         (3) Pseudo code – not ready to write source code (a program), yet
         (4) Discuss with all parties
         (5) Modify
         (6) Repeat
         (7) After the design is agreed upon,
            (a) Write the real program using a computer programming language
            (b) Test – run the program with known data
            (c) Modify – correct defects (errors)
            (d) Repeat
   b) A Problem Solving approach
         A simple introduction to the process of software development is using a systematic approach to problem solving.
      i) Understand the problem.
         *Learn about the problem domain. If necessary, break a large task into multiple smaller tasks*
      ii) Analyze the problem requirements.
         *Specify input values (knowns) and required output values (unknowns). Include the units. Identify the relevant formulae needed for computations and necessary constants values, e.g., gravity or pi.*
      iii) Work a hand example.
         *This will (1) identify the steps needed to solve the problem and (2) a set of input and resulting output that can be used to test your software, later.*
      iv) Develop an algorithm to solve the problem.
         *Record the steps used to solve the hand example. If necessary, divide steps into multiple simpler steps to provide a clear solution.*
      v) Implement the algorithm.
         *Now, it is time to write a computer program that follows the steps in the algorithm to solve the problem. The statements in the algorithm can be used as comments as a guide for writing code in the program.*
      vi) Test and verify the program solution.
         *Run the program correcting any errors that exists. Use the input values from the hand example to verify that the solution is correct.*
      vii) Maintain and update the program.
         *This step is necessary when new requirements are added or there is a policy change that affects the problem solution.*
   c) Use CS1 Activity 1 to demonstrate the problem solving approach. See the Activities section below. Note: Activities 2 and 3 may be use later with the introduction of selection and repetition.

**Figure 8.4. Software process curriculum module (continued)**

**Annotated Outline (continued)**

2) CS2
   a) Recap the CS1 introduction
   b) Software Metrics
      i) A key element of any engineering process is measurement. Measures help to better understand the attributes of a product and to assess its quality. Unlike other engineering disciples, software engineering is not grounded in the basic quantitative laws of physics, like voltage, mass, velocity, or temperature. What are the measurable attributes of software engineering work products?
      ii) What are software engineering products? requirements and design models, source code, and test cases.
   c) Vocabulary
      i) In software engineering, **measure**, **measurement**, and **metrics** are often used interchangeably.
      ii) A **measure** provides a quantitative **indication** of the extent, amount, dimension, capacity, or size of some attribute of a product or process.
      iii) **Measurement** is the act of determining a measure.
      iv) **Metric** is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
      v) A software engineer collects measures and develops metrics so that indicators will be obtained. An **indicator** is a metric or combination of metrics that provides insight into the software process, a software project, or the product itself
   d) Measurable attributes of software engineering
      i) Lines of code (LOC) and LOC per hour are metrics for planning software development
         (1) What are the measurable attribute of software engineering work products? We will look at source code because students are familiar with this product. Source code has size. If we know the average length of a program for solving a particular type problem and the average number of lines of code we write in an hour, we can estimate how long it would take to produce this type product.
      ii) Number and type of mistakes (defects) are also metrics to track improvement
         (1) If we always wrote code with no defects, our production level of producing code would be pretty good. But, we all make mistakes. Finding and correcting them take time and lowers the actual number of LOC per hour.
   e) Measuring Individual Performance - CS2 Activity
      i) How can we improve our LOC per hour? The obvious way is to make fewer mistakes. To help us reduce the number of mistakes, we need to note the types of mistakes that we make and try to not make them. One way to approach reducing the number of defects in our code is to keep a log of the defects…and how many. See the tables below for the defect log and instructions.
      ii) Completing an assignment is not (usually) done in one seating without interruptions. A time log will help you record how much time is spent in each stage. See the tables below for the time log and instructions.
      iii) Maintaining a record of LOC, time and defects, we can monitor improvement.

**Figure 8.5. Software process curriculum module (continued)**

The curriculum module contains a list of teaching resources in a ready-to-use state. These resources are included in the teaching activities section of the module. The teaching techniques present suggestion of how to convey the material found in the module, e.g. lecture, worksheets, small groups, role play, etc.

**Teaching Resources**

Process Worksheet
Defect Recording Log
Time Recording Log

**Teaching Techniques**

CS1 activities

- Lecture with slides
- Blank worksheet to guide the students through the process of problem solving. Lead a class discussion the solutions using a document camera with students providing the needed information. Students can be asked to lead the discussion or report on their solution.

CS2 activity

- Provide a section of code with defects and lead the students in finding and typing the defects.

**Figure 8.6. Software process curriculum module (continued)**

A list of automated tools, Figure 8.7, that can assist in the teaching and learning of the knowledge area included in the module. These tools were considered with respect to functionality, acquisition expense, and effort to introduce into the classroom environment.

A glossary and bibliography, Figure 8.8, are included to clarified terms used in the module that may be unfamiliar to the instructor and to provide the referenced material and other resource material that may enhance the teaching of the knowledge area.

The curriculum module includes one or more teaching activities that reinforce the concepts presented in the module. The activities are self-contained with instructions and teaching resources needed by the instructor. The suggested course activities in the Software Process Curriculum Module are presented in Figure 8.9-15.

**Tool support**

Process Dashboard – used with PSP

- Not User Friendly
- Describes Psp Scripts
- Does Calculations For You
- Better than using PSP manually

Eclipse Process Framework

- OpenUP process (also XP and scrum)
- Describes steps to follow
- Can attach tools to framework
- More sufficient than dashboard
- Helps enact process
- Guides you through process correctly
- Umbrella tool that walk you through a process

**Figure 8.7. Software process curriculum module (continued)**

The first CS1 teaching activity uses the well-known problem of solving for the real roots of a quadratic equation. A process worksheet, Figure 5.9, shows a problem solving approach to solving this familiar problem. After a walkthrough using this worksheet, the students can work individually on in small groups and solve another familiar problem using the blank process worksheet, Figure 8.19. A good problem to use is one that can grow with the following activities. The second CS1 activity, Figure 8.11, modifies the problem presented in CS1 Activity 1 by adding restrictions to the input and introduces input validation loops. CS1 Activity 3, Figure 8.12, expands to allow multiple sets of data input.

After students have some knowledge of problem solving, coding and types of errors, they can work toward improving their software development skills. In the CS2 activity, Figure 8.13, the student records the errors and time spent making corrections and time spent completing the assignment. Prior to this activity, students need to understand the types of errors: syntax, logic and, runtime. The goal of this activity is for student to reduce common errors by being more aware of them during the coding process. Their progress can be track during the semester using the provided defect record log, Figure 8.15, and time record log, Figure 8.15.

**Glossary**

Measure - provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process.

Measurement - the act of determining a measure.

Metric - a quantitative measure of the degree to which a system, component, or process possesses a given attribute.

Indicator - a metric or combination of metrics that provides insight into the software process, a software project, or the product itself

Software life cycle - a typical sequence of phased activities that represent the various stages of engineering through which software system passes

Software process - the network of object states and transitional events that represent the production of a software system in a form suitable for computational encoding and processing

**Bibliography**

HUMPHREY, W. 2000. "The Baseline Personal Process" in A Discipline for Software Engineering. Addison-Wesley, Boston.

LE BLANC, R. and SOBEL, S. (chairs) et al. 2004. Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. IEEE Computer Society Press and ACM Press (23 August 2004). Available at http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/education/cc2001/SE2004Volume.pdf

TRIPP, L. (chair), et al. 2004. IEEE Computer Society Professional Practices Committee. 2004. Guide to the software engineering body of knowledge Project (SWEBOK) and the Guide. IEEE Computer Society Press, Los Alamitos, CA, (2004). Available at http://www.swebok.org.

**Figure 8.8. Software process curriculum module (continued)**

**Suggested Course Activities**

No software engineering tools other than the IDE will be introduced for this series of activities.

**CS1 Activity 1**

First assignment is solving a problem that involves an equation. Introducing the assignment should include a class discussion of the steps necessary to solve this problem.

To introduce a systematic problem solving strategy, talk through solving an example using the steps. Because solving for the roots of a quadratic equation is familiar, it is a good example to use at multiple stages during the course. These multiple stages present a sequence of activities that allows students to revisit and modify existing code and observe how changes in requirements affect the code.

Understand the problem.
Find the real roots of a quadratic equation: $ax^2 + bx + c = 0$

Analyze the problem requirements.
3 coefficients: a, b, c

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Work a hand example.
Results from hand calculations:

| input | | | output |
|---|---|---|---|
| a | b | c | x1x2 |
| 1 | 3 | -4 | -41 |
| 2 | -4 | -3 | -0.582.58 |

Develop an algorithm.
Get coefficients: a, b, c.
Compute roots: x1, x2
Display results

Implement the algorithm.
This is where the program is written. The algorithm can be used comments in the program write the computer program statements.
Using the IDE that the students use, type the program.

NOTES:
1st time, use assignment statement for input
2nd time, use user input
Later, functions can be used for each step
These is an example of design alternatives.

Test and verify the program solution.
This can be an opportunity to discuss types of errors by including errors in the program.
Compile program and correct errors.
Run program using input from hand example.
If results are not correct, review set step in algorithm and program.

Maintain and update the program.
There will probably not be a required response for this step.

**Figure 8.9. Software process curriculum module (continued)**

**Process Worksheet**

Understand the problem.

Analyze the problem requirements.

Work a hand example.
Show work and results from hand calculations:

Develop an algorithm.

Implement the algorithm.
This is where the program is written. Start by copying and pasting the algorithm into the IDE editor window and marking the statements as comments. These comments will be a guide for writing the computer program statements.

Test and verify the program solution.
This is where students will run the program to determine if it solves the problem correctly.

Maintain and update the program.
No required response for this step.

**Figure 8.10. Software process curriculum module (continued)**

**CS1 Activity 2**

To introduce Selection, reuse the CS1 Activity 1 example and include the restrictions on the coefficients to find the real roots of a quadratic equation.

Understand the problem.
Find the real roots of a quadratic equation: $ax^2 + bx + c = 0$

Analyze the problem requirements.
3 coefficients: a, b, c
Restrictions on input:
a != 0
D >= 0

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Work a hand example.
Results from hand calculations:

| input | | | output |
|---|---|---|---|
| a | b | c | x1x2 |
| 1 | 3 | -4 | -41 |
| 0 | 7 | 6 | not a quad eq |
| 1 | 3 | 3 | -sqrt, not a real root |

NOTE: Sample input includes values to test restrictions

Develop an algorithm.
Get coefficients: a, b, c.
If a != 0, compute D
If D >= 0, compute roots: x1, x2
display results

Implement the algorithm.
This is where the program is written. Start by copying and pasting the algorithm into the IDE editor window and marking the statements as comments. These comments will be a guide for writing the computer program statements.

NOTES:
Use user input to prepare students for input validation loops, next time.
Later, functions can be used for each step

Test and verify the program solution.
This can be an opportunity to discuss types of errors by including errors in the program.
Compile program and correct errors.
Run program using input from hand example.
If results are not correct, review set step in algorithm and program.

Maintain and update the program.
There will probably not be a required response for this step.

**Figure 8.11. Software process curriculum module (continued)**

**CS1 Activity 3**

To introduce Repetition, reuse the CS1 Activity 2 and include the restrictions on the coefficients to find the real roots of a quadratic equation. Ask user to re-enter invalid coefficients values.

Understand the problem.
Find the real roots of a quadratic equation: $ax^2 + bx + c = 0$

Analyze the problem requirements.
3 coefficients: a, b, c
Restrictions on input:
a != 0
D >= 0

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Work a hand example.
Results from hand calculations:

| input | | | output | |
|---|---|---|---|---|
| a | b | c | x1 | x2 |
| 1 | 3 | -4 | -4 | 1 |
| 0 | 7 | 6 | not a quad eq | |
| 1 | 3 | 3 | [-sqrt] | |

NOTE: Sample input includes values to test restrictions

Develop an algorithm.
While a == 0, get a
    Get coefficient b, c
    Compute D
    If D < 0,
    else need new a, b, c
Compute roots: x1, x2
Display results

Design alternatives can be introduced at this stage of this example.

After giving student the steps to solving the assignment problems for the first few assignments, ask them to write and submit their own software development plan for the assignments.

**Figure 8.12. Software process curriculum module (continued)**

**CS2 Activity**

After students have knowledge of problem solving, coding and types of errors, they can work on improving their software development skills. They will record the errors and time spent making corrections and time spent completing the assignment. Prior to this activity, students need to understand the types of errors: syntax, logic and, runtime. The goal of this activity is for student to reduce common errors by being more aware of them during the coding process. Progress can be track during the semester.

The logs and instructions for using the logs used in this activity are an adaption of those found in HUMPHREY, W. 2000. "The Baseline Personal Process" in A Discipline for Software Engineering. Addison-Wesley, Boston.

| DEFECT RECORDING LOG INSTRUCTIONS | |
|---|---|
| Purpose | This form holds the data on each defect as you find and correct it. |
| General | Record in this log all defects found in review, compile, and test.<br>Record each defect separately and completely.<br>If you need additional space, use another copy of the form. |
| **Column** | |
| No. | Enter the defect number. For each program, this should be a sequential number starting with, for example, 1 or 001. |
| Date | Enter the date when the defect was found. |
| Type | Enter the defect type from the defect type list. Use your best judgment. |
| Fix defect | If you injected this defect while fixing another defect, record the number of the previously improperly fixed defect. |
| Fix time | Enter you best judgment of the time you took to fix the defect, i.e., in seconds, minutes. |
| Description | Write a brief description of the defect that is clear enough to later remind you about the error and help you to remember why you made it. |

| TIME RECORDING LOG INSTRUCTIONS | |
|---|---|
| Purpose | This form is for recoding the time spent doing the project. |
| General | Record all the time you spend on the project<br>Record the time in minutes.<br>Be as accurate as possible.<br>If you need additional space, use another copy of the form. |
| **Column** | |
| Date | Enter the date when the entry is made. |
| Start Time | Enter the time when you start working on a task. |
| Stop Time | Enter the time when you stop working on the task. |
| Interruption | Record any interruption time that was not spent on the task and the reason for the interruption.<br>It you have several interruptions, enter their total time. |
| Work Time | Enter the clock time you actually spent working on the task, less the interruption time. |
| Comments | Enter reasons for interruptions and other comments that may remind you of any unusual circumstances regarding this activity. |

**Figure 8.13. Software process curriculum module (continued)**

DEFECT RECORDING LOG*

Student ____________________________________ Total # defects _______ Start date___________

Class ________________ Assignment # ________ Total fix time _______ End date___________

| No. | Date | Fix defect | Fix Time | Descriptions |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

* adaption of defect log found in HUMPHREY, W. 2000. “The Baseline Personal Process” in A Discipline for Software Engineering. Addison-Wesley, Boston.

**Figure 8.14. Software process curriculum module (continued)**

**TIME RECORDING LOG***

Student ________________________________Total # time _______Start date___________

Class ________________ Assignment # ________End date___________

Work time = Stop time – Start time – Interruption
Comments may be use explain interruptions

| Date | Start Time | Stop Time | Interr-uption | Work Time | Comments |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

* adaption of time log found in HUMPHREY, W. 2000. “The Baseline Personal Process” in A Discipline for Software Engineering. Addison-Wesley, Boston.

**Figure 8.15. Software process curriculum module (continued)**

# 9Conclusion and Future Work

## 9.1Summary of Research

Software engineering strives "to deliver on-time, high-quality, operational software that contains functions and features that meet the needs of all stakeholders." Guidelines are needed to successfully produce a software product of this caliber. During the past fifty years, the principles of software engineering have been matured and provide guidelines for a solid approach to developing software solutions. [Pressman 2010]

In the computer science curriculum, students receive considerable experience in the programming, or coding, phase of the software lifecycle [Pressman 2010]. Their projects are usually limited to small problems in which there is little need for requirements analysis, design, testing, and maintenance [Myers 2000]. Students are taught to write computer programs, but few can develop large software systems [Long 2008]. For software engineers, computers and programming languages are tools to be used in designing and implementing a solution to a problem. They use procedures, paradigms, tools, and techniques to produce quality software products [Pfleege and Altee 2006].

The existence of software engineering curriculum guidelines reinforces the need for teaching software engineering principles in two- and four-year undergraduate programs. The problem with adding a software engineering curriculum to two- and four-year computer science programs is resources. Because there is overlap of material in computer science and software engineering curriculum guidelines,

it may be possible to include software engineering in an existing computer science program with little or no additional resources.

Community colleges' open-door admission policies, reduced costs, convenient campus locations, and comprehensive course offerings offer a diverse population of students an alternative to the traditional four-year universities. Over the past 40 years, public community college enrollment has increased at a much faster rate than at the public four-year universities, with the percentage of women enrolled in community colleges surpassing that of men. Because of the low cost and accessibility, racial and ethnic minorities have become an increasing proportion of all students enrolled at community colleges [Kasper 2002]. In the time of a recession, community colleges experience an abnormal increase in student enrollment as unemployed workers seek to continue their education or change career fields [Tirrell-Wysocki 2009].

The state of Alabama has an extensive network of community colleges that provides an important and accessible source of higher education to their communities. The goal of the computer science programs in these institutions is to provide students with an opportunity to prepare for the work force or transfer to a four-year computer science program. The [Alabama] Statewide Transfer and Articulation Reporting System (STARS) is an academic planning tool between students in a community college and the four-year universities for academic programs. STARS assists community college student in having the prerequisites to transfer to a specific program and a specific four-year program [STARS 2009b].

The first objective of this research was to determine the state of computer science teaching in community colleges. Information about the computer science programs in Alabama public community colleges was collected from the schools' websites. In the websites, we found that the CIS courses offered and required for a computer science degree and the prerequisites vary by the community college. Although these courses have the same CIS course numbers in each community college catalog, some course numbers represent courses with different titles and course descriptions. This can put some

students at a disadvantage in the workplace, but, more in line with this research, students can be at a disadvantage when entering a baccalaureate program at a public four-year university.

Additional information about computer science programs in Alabama state community colleges was obtained through an online faculty survey. The survey invitation was also sent to the computer science faculty of Alabama public four-year universities for comparison. The survey responses indicated that many programs do not include the teaching software engineering principles and concepts at the introductory-level. It revealed a signification lack of familiarity with the computing curricula guidelines provided by the ACM, IEEC-CS, and Two-Year College Education Committee. This could lead to inconsistences in what and how computer science is taught and the lack of the inclusion of software engineering principles.

The second objective was to determine which principles and concepts of software engineering can be pushed down into the introductory-level computer science courses and to create a curriculum for teaching software engineering in existing introductory computer science courses. The collection of ideas and suggestions for the curriculum began during a SIGCSE 2011 Birds-of-a-Feather session. Topics, such as requirements, design, testing, documentation, inspection, and problem solving, were discussed as they apply to the CS1/CS2 courses in two- and four-year computer science programs. The information gathered during birds-of-a-feather discussion contributed to the selection of terms and concept included in the online survey that was sent to computer science faculty in Alabama public colleges and universities.

The third objective was to create a set of teaching modules for teaching software engineering knowledge areas in current computer science courses. The previously collected information was instrumental in the selection of the eight software engineering knowledge areas covered in a special topics course, Teaching Software Engineering. The course research and discussion lead to the

accumulation of information used to create teaching modules for the following software engineering knowledge areas.

- Software process
- Software testing
- Software construction
- Software design
- Software quality
- Software requirements
- Software configuration management

Each teaching module contains:

- Description of the Knowledge Area
- Philosophy…why is it important to include in CS1/2
- Teaching Outcomes
- Prerequisite Knowledge
- Teaching Outline and Annotated Outline
- Teaching Resources
- Teaching Techniques
- Tool Support
- Suggested Course Activities
- Glossary
- Bibliography

The teaching modules, which are available in Appendix E, of the document are intended to be supplemental and not to replace the existing computer science course curricula. Each module was designed to be self-supportive with suggested learning objectives, a teaching outline, software tool

support, teaching activities, and other material to assist the instructor. The teaching modules were evaluated by faculty and graduate students from area colleges and universities during a workshop that was held at the end of the summer course. The comments from the module evaluations were positive and agreeable with the material in the modules with the teaching of software engineering principles early being considered beneficial for the students and allowing for time students to mature in using the skills during their academic career. Caution was expressed by some that some modules may be too large and could interfere with the normal contents of the curriculum being taught. It was suggested that some activities be used with multiple modules for more continuity and ease of covering more without too much extra work.

## 9.2Future work

The teaching modules included in this research are a beginning. Additional refinement is needed to incorporate the suggestion of consolidating activities into multiple modules to allow for continuity and time efficiency. A further analysis on the software tools in the modules will help identify the most effective ones or possibly identify the need for creating a new one. Attention will be given to Computer Science Curricula 2013: Strawman Draft (Sahami, Roach, et al., 2012) when refining the teaching modules.

The evaluation and finalizing of the teaching modules is an ongoing process. During the research, a voluntarily submitted list of names and email addresses was establish of faculty who participated in the events. This list provides a means for sharing the modules and getting additional feedback. Introducing a pilot course at a local community college and asking the instructor and students to evaluate the course via a survey would provide information to further hone the content of the teaching modules. Metrics need to be developed to be used by community college faculty and students to assist in validating the teaching modules.

Once the content of the teaching modules is stable, it can be distributed to community college faculty. On-site faculty workshops can be used to assist with incorporating the new curricula into the existing courses. With the support of one or more community college programs, a grant can be written to support ongoing community college faculty training workshops and to expand the focus of this research by surveying community colleges outside Alabama..

# References

AACC. 2009. About community colleges. *American Association of Community Colleges*. Retrieved on 15 August 2009 from http://aacc.nche.edu

ACCS. 2009a. Powers of State Board of Education. Alabama State Board of Education. Retrieved on 28 May 2009 from http://www.accs.cc/BoardCitation.aspx

ACCS. 2009b. System Overview. *Alabama Community College System*. Retrieved on 28 May 2009 from http://www.accs.cc/aboutaccs.aspx

ACHE. 1975. Non-Resident Institutional Review. *Alabama Commission on Higher Education*. Retrieved on 07 Novemer 2009 from http://www.ache.alabama.gov/Nonresident/ index.htm

ACHE. 2008a. Transfer/Migration Reports, 1999-2008. *Alabama Commission on Higher Education*. Retrieved on 25 May 2009 from http://www.ache.alabama.gov/studentdb/ index.htm

ACHE. 2008b. Alabama Commission on Higher Education 2008 Accountability Report. *Alabama Commission on Higher Education* (12 December 2008). http://www.ache.alabama.gov/Publications/Accountability%20Report%202008.pdf

ACHE. 2009a. Institutional Student Profiles Fall 2007. *Alabama Commission on Higher Education*. Alabama State Data Center, University of Alabama. Retrieved on 25 May 2009 from http://www.ache.alabama.gov/profiles/2007%20Profiles/2007%20Institutional %20Student%20Profile.pdf

ACHE. 2009b. Mission Statement. *Alabama Commission on Higher Education*. Retrived on 28 May 2009 from http://www.ache.alabama.gov/aboutus/mission.htm

ACHE. 2009c. Responsibilities. *Alabama Commission on Higher Education*. Retrieved on 28 May 2009 from http://www.ache.alabama.gov/aboutus/responsibilities.htm

ADPE. 2005. CIS Syllabi. Alabama Department of Postsecondary Education. Retrieved on 28 May 2009 from http://www.nacc.edu/assessment/syllabi/ComputerScience_AreaV.htm

AGSC. 2009a. What is the AGSC? *Alabama Articulation and General Studies Committee*. Retrieved on 28 May 2009 from http://stars.troy.edu/agsc/what_agsc.htm

AGSC. 2009b. Articulation and general studies committee approved general studies curriculum. *Alabama Articulation and General Studies Committee.* Retrieved 28 May 2009 from http://stars.troy.edu/agsc/what_agsc.htm#AREAS

AGSC. 2009c. AGSC Academic Committees. *Alabama Articulation and General Studies Committee.* Retrieved on 28 May 2008 from http://stars.troy.edu/agsc/academic.htm

AGSC. 2009d. AGSC template ratification process. *Alabama Articulation and General Studies Committee.* Retrieved from 02 June 2009 from http://stars.troy.edu/agsc/template_process.htm

ALICE. 2009. Alice. Available at www.alice.org

ALLEN, E., CARTWRIGHT, R., and STOLER, B. 2002. DrJava: a lightweight pedagogic environment for Java. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky, February 27 - March 03, 2002). SIGCSE '02. ACM, New York, NY, 137-141. DOI= http://doi.acm.org/10.1145/563340.563395

AMBLER, S., and JEFFERIES, R. 2002. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. Wiley, New Jersey.

ANDERSON, L. W., and KRATHWOHL, D. R. (Eds.). 2001. A Taxonomy for Learning, Teaching and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives. Addison Wesley Longman, New York.

BAILEY, J. L., and STEFANIAK, G. 2002. Preparing the information technology workforce for the new millennium. *SIGCPR Computer Personnel* 20, 4 (Aug. 2002), 4-15. DOI= http://doi.acm.org/10.1145/571475.571476

BARNETT, D. D. 2009. Karek the Robot. Available at http://home.att.net/~David.D.Barnett/karel-home.html

BLOOM, B. S., and KRATHWOHL, D. R. 1956. Taxonomy of Educational Objectives: Handbook I: Cognitive Domain. New York: David McKay.

BLUEJ. 2009a. BlueJ--The interactive java environment. Available at http://www.bluej.org.

BOEHM, B. 2006. A view of 20th and 21st century software engineering. In *Proceeding of the 28th International Conference on Software Engineering* (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM Press, New York, NY, 12-29. DOI= http://doi.acm.org/10.1145/1134285.1134288.

BOLOIX, G., and ROBILLARD, P. N. 1998. CASE tool learnability in a software engineering course. *IEEE Transactions on Education.* 41, 3 (Aug. 1998), 185-193.

BOUILLON, P., BURGER, M., and ZELLER, A. 2003. Automated debugging in Eclipse: (at the touch of not even a button). In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology Exchange* (Anaheim, California, October 27 - 27, 2003). Eclipse '03. ACM, New York, NY, 1-5. DOI= http://doi.acm.org/10.1145/965660.965661

BROOKS, F. P. 1995. *The Mythical Man-Month: Essays on Software Engineering*, Boston: Addison-Wesley.

BUCK, D. and STUCKI, D. J. 2001 JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. *SIGCSE Bulletin* 33, 1 (Mar. 2001), 16-20. DOI= http://doi.acm.org/10.1145/366413.364529

BURCH, C. 2009. Jigsaw, a programming environment for Java in CS1. *Journal of Computing in Small Colleges* 24, 5 (May. 2009), 37-43.

BURGESS, L. 1995. No easy way to reform the FAA. *Journal of Commerce*. (Oct. 30, 1995), 14.

CAMPBELL, R. (chair) et al. 2003. Computing curriculum 2003: guidelines for associate-degree curricula in computer science. *IEEE Computer Society Press and ACM Press*, (December, 2002). Available at http://www.acmtyc.org/reports/TYC_CS2003_report.pdf

CAMPBELL, R. (chair) et al. 2005. Computer curricula 2005: Guidelines for associate-degree transfer curriculum in software engineering. *IEEE Computer Society Press and ACM Press*, (August, 2005). Available at http://www. acmtyc.org/reports/TYC_SE_report.pdf

CHANG, C., DENNINGS, P., et al. 2001. Computing curricula 2001: Computer science. Final report (December 15, 2001). *IEEE Computer Society Press and ACM Press* (Dec. 15, 2001). Available at http://www.acm.org/education/curric_vols/cc2001.pdf.

CHEN, Z. and MARX, D. 2005. Experiences with Eclipse IDE in programming courses. *Journal of Computing in Small Colleges* 21, 2 (Dec. 2005), 104-112.

CONN, R. 2002. Developing software engineers at the C-130J software factory. *IEEE Software* (Sep/Dec, 2002), 25-29.

COOPER, S., DANN, W., and PAUSCH, R. 2000. Alice: a 3-D tool for introductory programming concepts. *In Proceedings of the Fifth Annual CCSC Northeastern Conference on the Journal of Computing in Small Colleges* (Ramapo College of New Jersey, Mahwah, New Jersey, United States). J. G. Meinke, Ed. Consortium for Computing Sciences in Colleges. Consortium for Computing Sciences in Colleges*, 107-116.*

CRNKOVIC, I., LAND, R., and SJOGREN, A. 2003. Is software engineering training enough for software engineers? In *Proceedings of the 16th Conference on Software Engineering Education and Training* (20-22 March 2003). CSEET'03. 140-147. DOI=10.1109/CSEE.2003.1191371

CROSS, J. H., HENDRIX, T. D., JAIN, J., and BAROWSKI, L. A. 2007 Dynamic object viewers for data structures. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (Covington, Kentucky, USA, March 07 - 11, 2007). SIGCSE '07. ACM, New York, NY, 4-8. DOI= http://doi.acm.org/10.1145/1227310.1227316

CZYZ, J. K. and JAYARAMAN, B. 2007. Declarative and visual debugging in Eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange* (Montreal, Quebec, Canada, October 21 - 21, 2007). eclipse '07. ACM, New York, NY, 31-35. DOI= http://doi.acm.org/10.1145/1328279.1328286

D'ANJOU, J., et al. 2005. The Java Developer's Guide to Eclipse, 2e. Addison-Wesley, Boston, MA.

DAVIS, A. 1995. *201 Principles of Software Development*. McGraw-Hill, New York, NY.

DENNING, P J., Ed. 1989. A debate on teaching computing science. *Communications of the ACM* 32, 12 (Dec. 1989), 1397-1414. DOI= http://doi.acm.org/10.1145/76380.76381

DENNING, P. J. 2004. The field of programmers myth. *Communications of the ACM* 47, 7 (July, 2004), 15-20.

DEUGO, D. 2008. Using eclipse in the classroom. *SIGCSE Bulletin* 40, 3 (Aug. 2008), 322-322. DOI= http://doi.acm.org/10.1145/1597849.1384365

DRJAVA. 2009. DrJava. Available at http://www.drJava.org.

ECLIPSE. 2009. Eclipse. Available at http://www.eclipse.org.

FISHER, K., KRINTZ, C. (chairs), et al. 2008. 2008 SIGPLAN Programming Language Curriculum Workshop Report. *2008 SIGPLAN Workshop on Programming Language Curriculum*, ACM SIGPLAN Notices 43, 11 (Nov. 2008).

FISKER, K., MCCALL, D., KÖLLING, M., and QUIG, B. 2008. Group work support for the BlueJ IDE. In *Proceedings of the 13th Annual Conference on innovation and Technology in Computer Science Education* (Madrid, Spain, June 30 - July 02, 2008). ITiCSE '08. ACM, New York, NY, 163-168. DOI= http://doi.acm.org/10.1145/1384271.1384316

GAO. 2007. Higher education: Tuition continues to rise, but patterns vary by institution type, enrollment, and educational expenditures (GAO-08-245). U.S. Government Accountability Office report to the Chairman, Committee on Education and Labor, House of Representatives, Washington, DC, 1-30.

GIBBS, N. E. 1989. The SEI education program: the challenge of teaching future software engineers. *Communications of the ACM* 32, 5 (May 1989), 594-605.

GREENFOOT. 2009. Greenfoot. Available at http://www.Greenfoot.org.

HARRISON, W., OSSHER, H., and TARR, P. 2000. Software engineering tools and environments: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (Limerick, Ireland, June 04 - 11, 2000). ICSE '00. ACM Press, New York, NY, 261-277. DOI= http://doi.acm.org/10.1145/336512.336569

HAWTHORNE, E. (chair) et al. 2009. Computing curricula 2009: Guidelines for associate-degree transfer curriculum in computer science. *ACM Two-Year College Education Committee. ACM and IEEE Computer Society* (2009). Available at http://www.acmtyc.org

HENRIKSEN, P. and KÖLLING, M. 2004. greenfoot: combining object visualisation with interaction. In *Companion To the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, BC, CANADA, October 24 - 28, 2004). OOPSLA '04. ACM, New York, NY, 73-82. DOI= http://doi.acm.org/10.1145/1028664.1028701

HOOKER, D. 1996. Seven Principles of Software Development (September 1996). Available at http://www.c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment.

HUNDLEY, J. 2011. Birds-of-a-Feather: Introducing Software Engineering Principles in the First Two Years of Computer Science Education. SIGCSE 2011 *the 42nd ACM Technical Symposium on Computer Science Education*. Dallas TX, USA, March 09-12, 2011.

HUNT, A. and THOMAS, D. 2004. Three essential tools for stable development. *CrossTalk: The Journal of Defense Software Engineering* 17, 11 (Nov. 2004). 22-25.

JGRASP. 2009a. jGRASP. Available at http://jgrasp.org

JGRASP. 2009b. Overview of JGRASP and the tutorials (2 September 2009). Available at http://jgrasp.org/tutorials187/00_Overview.pdf

JOHNSON, D. W., and JONES, C. G. 2006. IS education: the changing complexity of relevance. *Issues in Information Systems* 7, 1 (2006), 188-192.

KASPER, H. 2002. The changing role of community college. *Occupational Outlook Quarterly* (Winter 2002-03), 14-21.

KOLLING, M. 2009. The BlueJ tutorial, version 2.0.1 for BlueJ version 2.0.x. *Maersk Institute, University of Southern Denmark*. Available at http://bluej.org/tutorial/tutorial-201.pdf

KÖLLING, M., QUIG, B. PATTERSON, A., and ROSENBERG, J. 2003. The BlueJ system and its pedagogy," *Journal of Computer Science Education*, 13, 4 (December 2003), 249-268.

KORNECKI, A.J.; S. KHAJENOORI, D. GLUCH, and N. KAMELI. 2003. On a partnership between software industry and academia. In *Proceedings of the 16th Conference on Software Engineering Education and Training* (20-22 March 2003). CSEET'03. 60-69. DOI=10.1109/CSEE.2003.1191351

KOUZNETSOVA, S. 2007. Using BlueJ and Blackjack to teach object-oriented design concepts in CS1. *Journal of Computing in Small Colleges* 22, 4 (Apr. 2007), 49-55.

LANG, J. 1999. Industry expectations of new engineers: A survey to assist curriculum designers. *Journal of Engineering Education* 88, 1 (Jan. 1999), 43-51.

LE BLANC, R. and SOBEL, S. (chairs) et al. 2004. Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. *IEEE Computer Society Press and ACM Press* (23 August 2004). Available at http://www.computer.org/ portal/cms_docs_ieeecs/ieeecs/education/cc2001/SE2004Volume.pdf

LEWIS, P. M. 1989. Information Systems is an Engineering Discipline. *Communications of the ACM* 32, 9 (Sept. 1989), 1045-1047.

LONG, Lyle N. 2008. The critical need for SE education *CrossTalk: The Journal of Defense Software Engineering* 21, 1 (Jan 2008), 6-10.

MCCAULEY, R. and MCGETTRICK, A. 2008. Computer Science Curriculum 2008: An Interim Revision of the CS 2001, a report from the interim review task force. *IEEE Computer Society Press and ACM* (December, 2008). Available at http://www.acm.org//education/curricula/ComputerScience2008.pdf

MCCONNELL, S. 1996. Best Practices: Daily and Smoke Test. *IEEE Software* 13, 4 (July 1996), 143-144.

MEAD, N.R., SAIEDIAN, H., RUHE, G., and BAGERT, D.J. 2000. Panel: shortages of qualified software engineering faculty and practitioners: challenges in breaking the cycle In *Proceedings of the 2000 International Conference on Software Engineering* (4-11 June 2000), 665–668.

MITCHELL, W. 2004. Is software engineering for everyone? In *Proceedings of the 2nd Annual Conference on Mid-South College Computing* (Little Rock, Arkansas, April 02 - 03, 2004). ACM International Conference Proceeding Series, vol. 61. Mid-South College Computing Conference, Little Rock, Arkansas, 53-64.

MYERS, J. P. 2000. Software engineering throughout a traditional computer science curriculum. In *Proceedings of the Second Annual CCSC on Computing in Small Colleges Northwestern Conference* (Oregon Graduate Institute, Beaverton, Oregon, United States). Consortium for Computing in Colleges, 31-40.

NAVARRO, D., HORN, T., and SALINGER, G. 2008. ATE centers and community colleges: Increasing underrepresented minorities participating in STEM fields: A forum (21 November 2008). Retrieved on 09 June 2009 from http://www.aypf.org/forumbriefs/2008/fb112108.htm

OLAN, M. 2004. Dr. J vs. the bird: Java IDE's one-on-one. *Journal of Computing in Small Colleges* 19, 5 (May. 2004), 44-52.

PARNAS, D. L. 1990. Education for Computing Professionals. *Computer* 23, 1 (Jan. 1990), 17-22. DOI= http://dx.doi.org/10.1109/2.48796

PATERSON, J. H., HADDOW, J., BIRCH, M., and MONAGHAN, A. 2005. Using the BlueJ IDE in a data structures course. In *Proceedings of the 10th Annual SIGCSE Conference on innovation and Technology in Computer Science Education* (Caparica, Portugal, June 27 - 29, 2005). ITiCSE '05. ACM, New York, NY, 349-349. DOI= http://doi.acm.org/10.1145/1067445.1067548

PFLEEGER S. L. 1998. *Software Engineering, Theory and Practice*, Prentice-Hall, Inc., 1998.

PFLEEGER, S. L. and ALTEE, J. M. 2006. *Software Engineering, Theory and Practice*, Prentice-Hall, Inc., Upper Saddle River, NJ.

PRESSMAN, R. 2001. Software Engineering: A Practitioner's Approach, 5e. McGraw Hill, New York, NY.

PRESSMAN, R. S. 2010. *Software Engineering: A Practitioner's Approach*. 7e. McGraw-Hill, New York, NY.

QAA. 2000. Quality assurance agency for higher education: A report on benchmark levels for computing. *The Quality Assurance Agency for Higher Education*. Southgate House, Southgate Street, Gloucester GL1 1UB. Available at www.qaa.ac.uk.

QUALTRICS. 2011. Qualtrics: Survey Research Suite. Available at www.qualtrics.com.

RATIONAL. 2009. IBM Rational Software. Available at http://www-01.ibm.com/software/rational/

REIFER, D. J. 2005. Educating software engineers: an industry viewpoint. *SIGSOFT Software Engineering Notes* 30, 3 (May. 2005), 8-9. DOI= http://doi.acm.org/10.1145/1061874.1061876

REIS, C. and CARTWRIGHT, R. 2003. A friendly face for Eclipse. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology Exchange* (Anaheim, California, October 27 - 27, 2003). eclipse '03. ACM, New York, NY, 25-29. DOI= http://doi.acm.org/10.1145/965660.965666

REIS, C. and CARTWRIGHT, R. 2004. Taming a professional IDE for the classroom. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (Norfolk, Virginia, USA, March 03 - 07, 2004). SIGCSE '04. ACM, New York, NY, 156-160. DOI= http://doi.acm.org/10.1145/971300.971357

ROY, G. G. 2006. Designing and explaining programs with a literate pseudocode. *Journal on Educational Resources in Computing* 6, 1 (Mar. 2006), 1. DOI= http://doi.acm.org/10.1145/1217862.1217863

RUBEL, D. 2006. The Heart of Eclipse. *Queue* 4, 8 (Oct. 2006), 36-44. DOI= http://doi.acm.org/10.1145/1165754.1165767

SAHAMI, M. (ACM Delegation chair), ROACH, S. (IEEE-CS Delegation chair), et al. 2012. Computer Science Curricula 2013: Strawman Draft. Retrieved on March 7, 2012 from www.cs2013.org.

SANDERS, D. and HEELER, P. 2001. Introduction to BlueJ: a Java development environment for CS1 and CS2. In *Proceedings of the Seventh Annual Consortium For Computing in Small Colleges Central Plains Conference on the Journal of Computing in Small Colleges* (Branson, Missouri, United States). J. G. Meinke, Ed. Consortium for Computing Sciences in Colleges, 115-116.

SHACKELFORD, R. (chair) et al. 2005. Computing curricula 2005: Overview report on computing curricula. *IEEE Computer Society Press and Association of Computing Machinery Press* (Sep. 30, 2005). Available at http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf.

SMITH, P. A. and BOYD, G. 2001. Introducing OO concepts from a class user perspective. *Journal of Computing in Small Colleges* 17, 2 (Dec. 2001), 152-158.

SOMMERVILLE, I. 2004. *Software Engineering,* 7ed. Pearson Education Limited, Edinburgh, England.

SPAC. 2009. Forging Strategic Alliances: State Plan for Alabama Higher Education 2009-2014, Draft. *State Planning Advisory Council, Alabama Commission on Higher Education*, Montgomery, AL. Retrieved on 09 June 2009 from http://www.highered.alabama.gov/Portals/9/Documents/COP%20handout.ppt

STARS. 2009a. What is STARS? *Statewide Transfer and Articulation Reporting System*. Retrieved on 28 May 2009 from http://stars.troy.edu/stars/what_stars.htm

STARS. 2009b. Statewide transfer and articulation reporting systems. *Statewide Transfer and Articulation Reporting System.* Retrieved on 28 May 2009 from http://stars.troy.edu/stars/stars.htm

STEM. 2009. STEM Education Coalition. Retrieved on 09 June 2009 from http://www.stemedcoalition.org

STILLER, E. and LEBLANCE, C. 2002. Effective software engineering pedagogy. *Journal of Computing in Small Colleges* 17, 6 (May 2002), 124-134.

TILLEY, S.R., WONG, K. 1993. Report on NWSEE '93. *The 1993 [Canadian] National Workshop on Software Engineering Education* (Aug 27, 1993). Available at www.cs.ualberta.ca/~kenw/papers/nwsee93-rep.pdf.

TIRRELL-WYSOCKI, D. 2009. Recession sending more students to community colleges. *The Seattle Times* (08 February 2009). Retrieved on 09 June 2009 from http://seattletimes. nwsource.com/html/nationworld/2008721603_apmeltdowncommunitycolleges.html

TRIPP, L. (chair), et al. 2004. IEEE Computer Society Professional Practices Committee. 2004. Guide to the software engineering body of knowledge Project (SWEBOK) and the Guide. *IEEE Computer Society Press*, Los Alamitos, CA, (2004). Available at http://www.swebok.org.

USCNS/21. 2001. Recapitalizing America's strengths in science and education in Road map for National security: Imperative for change: The phase III report of the U.S. Commission on National Security/21st century (15 February 2001), 30-46.

VERAAT, V., HILTON, M., SAMY, N., GRANT, D., and GREENING, T. 1997. Software Engineering Education - Is It Meeting Industry Needs? Can Industry Needs Be Met? In *Proceedings of Australian Software Engineering Conference* (29 Sep-2 Oct 1997). ASWEC 97, 184-187.

WERTH, J. and WERTH, L. 1991. Directions in software engineering education. In *Proceedings of the 13th international Conference on Software Engineering* (Austin, Texas, United States, May 13 - 17, 1991). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 353-357.

XINOGALOS, S., SATRATZEMI, M., and DAGDILELIS, V. 2006. An introduction to object-oriented programming with a didactic microworld: objectKarel. *Computers & Education* 47 (2006), 148-171.

XINOGALOS, S., SATRATZEMI, M., and DAGDILELIS, V. 2007. Teaching java with BlueJ: a two-year experience. In *Proceedings of the 12th Annual SIGCSE Conference on innovation and Technology in Computer Science Education* (Dundee, Scotland, June 25 - 27, 2007). ITiCSE '07. ACM, New York, NY, 345-345. DOI= http://doi.acm.org/10.1145/1268784.1268914

Appendix A

SWEBOK Software Engineering Knowledge Areas (KAs) [Pressman 2010]

| SOFTWARE REQUIREMENTS | |
|---|---|
| **1. Software requirements fundamentals** | |
| Definition of software requirement | C |
| Product and process requirements | C |
| Functional and functional requirements | C |
| Emergent properties | C |
| Quantifiable requirements | C |
| System requirements and software requirements | C |
| **2. Requirement process** | |
| Process models | C |
| Process actors | C |
| Process support and management | C |
| Process quality and improvement | C |
| **3. Requirements elicitation** | |
| Requirements sources | C |
| Elicitation techniques | AP |
| **4. Requirements analysis** | |
| Requirements classification | AP |
| Conceptual modeling | AN |
| Architectural design and requirements allocation | AN |
| Requirements negotiation | AP |
| **5. Requirements specification** | |
| System definition document | C |
| System requirements specification | C |
| Software requirements specification | AP |
| **6. Requirements validation** | |
| Requirements reviews | AP |
| Prototyping | AP |
| Model validation | C |
| Acceptance tests | AP |
| **7. Practical considerations** | |
| Iterative nature of requirements process | C |
| Change management | AP |
| Requirements attributes | C |
| Requirements tracing | AP |
| Measuring requirements | AP |

| SOFTWARE DESIGN | |
|---|---|
| **1. Software design fundamentals** | |
| General design concepts | C |
| Context of software design | C |
| Software design process | C |
| Enabling techniques | AN |
| **2. Key issues in software design** | |
| Concurrency | AP |
| Control and handling of events | AP |
| Distribution of components | AP |
| Error and exception handling; fault tolerance | AP |
| Interaction and presentation | AP |
| Data persistence | AP |
| **3. Software structure and architecture** | |
| Architectural structures and viewpoints | AP |
| Architectural styles (macro-arch patterns) | AN |
| Design patterns (micro-architectural patterns) | AN |
| Families of programs and frameworks | C |
| **4. Software design quality analysis and evaluation** | |
| Quality attributes | C |
| Quality analysis and devaluation techniques | AN |
| Measures | C |
| **5. Software design notations** | |
| Structural descriptions (static) | AP |
| Behavioral descriptions (dynamic) | AP |
| **6. Software design strategies and methods** | |
| General strategies | AN |
| Function-oriented (structured) design | AP |
| Object-oriented design | AN |
| Data-structure centered design | C |
| Component-based design (CBD) | C |
| Other methods | C |

| SOFTWARE CONTRUCTION | |
|---|---|
| **1. Software construction fundamentals** | |
| Minimizing complexity | AN |
| Anticipating change | AN |
| Constructing of verification | AN |
| Standards in construction | AP |
| **2. Managing construction** | |
| Construction methods | C |
| Construction planning | AP |
| Construction measurement | AP |
| **3. Practical consideration** | |
| Construction design | AN |
| Construction languages | AP |
| Coding | AN |
| Construction testing | AP |
| Construction quality | AN |
| Integration | AP |

| SOFTWARE TESTING | |
|---|---|
| **1. Software testing fundamentals** | |
| Testing-related terminology | C |
| Key issues | AP |
| Relationships of testing to other activities | C |
| **2. Test levels** | |
| The target of the tests | AP |
| Objectives of testing | AP |
| **3. Test techniques** | |
| Based on tester's intuition and experience | AP |
| Specification-based | AP |
| Code-based | AP |
| Fault-based | AP |
| Usage-based | AP |
| Based on nature of application | AP |
| Selecting and combining techniques | AP |
| **4. Test-related measures** | |
| Evaluation of the program under test | AN |
| Evaluation of the tests performed | AN |
| **4. Test process** | |
| Management concerns | C |
| Test activities | AP |

| SOFTWARE MAINTENCE | |
|---|---|
| **1. Software maintenance fundamentals** | |
| Definitions and terminology | C |
| Nature of maintenance | C |
| Need for maintenance | C |
| Majority of maintenance costs | C |
| Evolution of software | C |
| Categories of maintenance | AP |
| **2. Key issues in software maintenance** | |
| Technical | |
| Limiting understanding | C |
| Testing | AP |
| Impact analysis | AN |
| Maintainability | AN |
| Management issues | |
| Alignment with organizational issues | C |
| Staffing | C |
| Process issues | C |
| Organizational | C |
| Maintenance cost estimation | |
| Cost estimation | AP |
| Parametric models | C |
| Experience | AP |
| Software maintenance measurement | AP |
| **3. Maintenance process** | |
| Maintenance process models | C |
| Maintenance activities | |
| Unique activities | AP |
| Supporting activities | AP |
| **4. Techniques for maintenance** | |
| Program comprehension | AN |
| Reengineering | C |
| Reverse engineering | C |

| SOFTWARE CONFIGURATION MANAGEMENT | |
|---|---|
| **1. Management of the SCM management** | |
| Organizational context for SCM | C |
| Constraints and guidance for SCM | C |
| Planning for SCM | |
| SCM organization and responsibilities | AP |
| SCM resources and schedules | AP |
| Tool selection and implementation | AP |
| Vendor/subcontractor control | C |
| Interface control | C |
| Software configuration management plan | C |
| Surveillance of software configuration mgmt | |
| SCM measures and measurement | AP |
| In-process audits of SCM | C |
| **2.Software configuration identification** | |
| Identifying items to be controlled | |
| Software configuration | AP |
| Software configuration items | AP |
| Software configuration item relationships | AP |
| Software versions | AP |
| Baseline | AP |
| Acquiring software configuration items | AP |
| Software library | C |
| **3. Software configuration control** | |
| Requesting, evaluating and approving software changes | |
| Software configuration control board | AP |
| Software change request process | AP |
| Implementing software changes | AP |
| Deviations and waivers | C |
| **4. Software configuration status accounting** | |
| Software configuration status information | C |
| Software configuration status reporting | AP |
| **5. Software configuration audit** | |
| Software functional configuration audit | C |
| Software physical configuration audit | C |
| In-process audits of a software baseline | C |
| **6. Software release management and delivery** | |
| Software building | AP |
| Software release management | C |

| SOFTWARE ENGINEERING MANAGEMENT | |
|---|---|
| **1. Initiation and scope definition** | |
| Determination and negotiation of requirements | AP |
| Feasibility analysis | AP |
| Process for requirements review/revision | C |
| **2. Software project planning** | |
| Process planning | C |
| Determine deliverables | AP |
| Effort, schedules, and cost estimation | AP |
| Resource allocation | AP |
| Risk management | AP |
| Quality management | AP |
| Plan management | C |
| **3. Software project enactment** | |
| Implementation of plans | AP |
| Supplier contract management | C |
| Implementation of measurement process | AP |
| Monitor process | AN |
| Control process | AP |
| Reporting | AP |
| **4. Review and evaluation** | |
| Determining satisfaction of requirements | AP |
| Review and evaluating performance | AP |
| **5. Closure** | |
| Determining closure | AP |
| Closure activities | AP |
| **6. Software engineering measurement** | |
| Establish and sustain measurement commitment | C |
| Plan the measurement process | C |
| Perform the measurement process | C |
| Evaluate measurement | C |

| SOFTWARE ENGINEERING PROCESS | |
|---|---|
| **1. Process implementation and change** | |
| Process infrastructure | |
| Software engineering process group | C |
| Experience factory | C |
| Activities | AP |
| Models for process implementation and change | K |
| Practical considerations | C |
| **2. Process definition** | |
| Life cycle models | AP |
| Software life cycle processes | C |
| Notations for process definitions | C |
| Process adaptations | C |
| Automation | C |
| **3. Process assessment** | |
| Process assessment models | C |
| Process assessment methods | C |
| **4. Product and process measurement** | |
| Software process measurement | AP |
| Software product measurement | |
| Size measurement | AP |
| Structure measurement | AP |
| Quality measurement | AP |
| Quality of measurement results | AN |
| Software information models | |
| Model building | AP |
| Model implementation | AP |
| Measurement techniques | |
| Analytical techniques | AP |
| Benchmarking techniques | C |

| SOFTWARE ENGINEERING TOOLS AND METHODS | |
|---|---|
| **1. Software tools** | |
| Software requirements tools | AP |
| Software design tools | AP |
| Software construction tools | AP |
| Software testing tools | AP |
| Software maintenance tools | AP |
| Software engineering process tools | AP |
| Software quality tools | AP |
| Software configuration management tools | AP |
| Software engineering management tools | AP |
| Miscellaneous tool issues | AP |
| **2. Software engineering methods** | |
| Heuristic methods | AP |
| Formal methods and notations | C |
| Prototyping methods | AP |
| Miscellaneous method issues | C |

| SOFTWARE QUALITY | |
|---|---|
| **1. Software quality fundamentals** | |
| Software engineering culture and ethics | AN |
| Value and costs of quality | AN |
| Quality models and characteristics | |
| Software process quality | AN |
| Software product quality | AN |
| Quality improvement | AP |
| **2. Software quality management process** | |
| Software quality assurance | AP |
| Verification and validation | AP |
| Reviews and audits | |
| Inspections | AP |
| Peer reviews | AP |
| Walkthroughs | AP |
| Testing | AP |
| Audits | C |
| **3. Practical considerations** | |
| Application quality requirements | |
| Criticality of systems | C |
| Dependability | C |
| Integrity levels of software | C |
| Defect characterization | AP |
| Software quality management techniques | |
| Static-techniques | AP |
| People-intensive techniques | AP |
| Analytic-techniques | AP |
| Dynamic techniques | AP |
| Software quality measurement | AP |

## BLOOM'S TAXONOMY LEVELS [Bloom 1956]

**Knowledge (K)**: Recall data

**Comprehension (C)**: Understanding the meaning, translation, interpolation, and interpretation of instructions and problems; state a problem in one's own words.

**Application (AP)**: Use a concept in a new situation or use an abstraction unprompted; apply what was learned in the classroom to novel situations in the workplace.

**Analysis (AN)**: Separate material or concepts into component parts so that its organizational structure may be understood; distinguish between facts and inferences.

**Synthesis (S)**: Build a structure of pattern from diverse elements, put parts together to form a whole, with emphasis on creating a new meaning or structure.

**Evaluation (E)**: Make judgments about the value of ideas or material.

Appendix B

Alabama Public Community Colleges' Reference Information.

| | **Legend** |
|---|---|
| CC URL | Community college web address |
| CC City | Community college city |
| Dept Name | Department name associated with the computer science curriculum or courses |
| Dept URL | Department web address |
| Cat Name | Course catalog name |
| Cat URL | Course catalog web address |
| Curriculum | Computer science curriculum catalog page number(s) or web address |
| STARS | Community college web page with information about the STARS program |

| | **Bevill State CC (BEV)** |
|---|---|
| CC URL | www.bscc.edu |
| CC City | Sumiton |
| Dept Name | Computer Science |
| Dept URL | http://www.bscc.edu/pos_computer.php |
| Cat Name | Course Descriptions: Computer Science (webpage) |
| Cat URL | http://www.bscc.edu/course_computer.php |
| Curriculum | http://www.bscc.edu/pos_computer_cr.php |
| STARS | http://www.bscc.edu/academics.php |

| | **Bishop State CC (BIS)** |
|---|---|
| CC URL | www.bishop.edu |
| CC City | Mobile |
| Dept name | Computer Information Systems (CIS) |
| Dept URL | http://www.bishop.edu/business.html |
| Cat Name | General Catalog 2008-2009 |
| Cat URL | http://www.bishop.edu/PDFs/bscat08.pdf |
| Curriculum | pp.33-35 of catalog |
| STARS | http://www.bishop.edu/resources.html |

| | **Calhoun State CC (CAL)** |
|---|---|
| CC URL | www.calhoun.edu |
| CC City | Decatur |
| Dept name | Computer and Office Information Systems |
| Dept URL | http://www.calhoun.edu/Bus_Div/cis.htm |
| Cat Name | 2008-2009 Catalog |
| Cat URL | http://www.calhoun.edu/Acrobat/catalog2008/Index.html |
| Curriculum | http://www.calhoun.edu/Bus_Div/cisprograms/AS.htm |
| STARS | http://www.calhoun.cc.al.us/Stars/index.html |

| | **Central Alabama CC (CEN)** |
|---|---|
| CC URL | www.cacc.edu |
| CC City | Alexander City |
| Dept name | na |
| Dept URL | na |
| Cat Name | 2008-2009 General Catalog |
| Cat URL | http://www.cacc.me/clientuploads/catalog/2008_2009_Catalog_complete.pdf |
| Curriculum | pp. 68-72 of catalog |
| STARS | home page menu link to stars.troy.edu |

| | **Chattahoochee Valley CC (CVCC)** |
|---|---|
| CC URL | www.cv.edu |
| CC City | Pheonix City |
| Dept name | Computer Information Systems |
| Dept URL | http://www.cv.edu/content/view/157/236/ |
| Cat Name | Catalog and Student Handbook 2008-2009 |
| Cat URL | http://www.cv.edu/component/option,com_wrapper/Itemid,132/ |
| Curriculum | http://cv.edu/external/catalogs/catalog_viewer.asp?109 |
| STARS | home page menu link to stars.troy.edu |

| | **Enterprise-Ozark CC (ENT)** |
|---|---|
| CC URL | www.eocc.edu |
| CC City | Enterprise |
| Dept name | Computer and Information Science |
| Dept URL | http://www.eocc.edu/divisions/cis_div/cis_home.html |
| Cat Name | College Catalog and Student Handbook 2008-2009 |
| Cat URL | http://www.eocc.edu/adminoffices/registrar/catalogs/CollegeCatalog.htm |
| Curriculum | pp. 81-82 of catalog |
| STARS | home page menu link to stars.troy.edu |

| | **Faulkner State CC (FSC)** |
|---|---|
| CC URL | www.faulknerstate.edu |
| CC City | Bay Minette |
| Dept name | Computer Science |
| Dept URL | http://www.faulknerstate.edu/majors |
| Cat Name | College Catalog and Student Handbook 2008-2009 |
| Cat URL | http://www.faulknerstate.edu/admissions/catalog0809 |
| Curriculum | pp. 78-84 of catalog |
| STARS | p. 10-11 of catalog |

**Gadsden State CC (GAD)**

| | |
|---|---|
| CC URL | www.gadsdenstate.edu |
| CC City | Gadsden |
| Dept name | Information Technology |
| Dept URL | http://www.gadsdenstate.edu/it/index.html |
| Cat Name | Catalog and Student Handbook 2008-2009 |
| Cat URL | http://www.gadsdenstate.edu/catalog/catalog0809.pdf |
| Curriculum | p. 98 of catalog |
| STARS | http://www.gadsdenstate.edu/enrolled.html |

**Jefferson State CC (JSC)**

| | |
|---|---|
| CC URL | www.jeffstateonline.com |
| CC City | Birmingham |
| Dept name | Computer Information Systems Technology |
| Dept URL | http://www.jeffstateonline.com/Business/index.aspx |
| Cat Name | Catalog and Student Handbook 2008-2009 |
| Cat URL | http://www.jeffstateonline.com/Catalog/PDFs/0809JSCCCatalog.pdf |
| Curriculum | pp. 96-98 of catalog |
| STARS | http://www.jeffstateonline.com/stars/index.aspx |

**Lawson State CC (LAW)**

| | |
|---|---|
| CC URL | www.lawsonstate.edu |
| CC City | Birmingham |
| Dept name | Computer Science - Math Degree |
| Dept URL | http://www.lawsonstate.edu/academics/computerscience/bit_index_computer.html |
| Cat Name | 2007-2009 Student Catalog and Handbook |
| Cat URL | http://www.lawsonstate.edu/catalogs/LSCC%202007-2009--<br>Electronic%20Student%20Catalog%20&%20Handbook.pdf |
| Curriculum | pp. 103, 185-185 of catalog |
| STARS | pp. 68-78 of catalog |

**Lurleen B. Wallace CC (LBW)**

| | |
|---|---|
| CC URL | www.lbwcc.edu |
| CC City | Andalusia |
| Dept name | Business-Information Technology/Social Science |
| Dept URL | http://www.lbwcc.edu/cms/page.aspx?pageid=528 |
| Cat Name | College Catalog 2007-2009 |
| Cat URL | http://www.lbwcc.edu/cms/Storage/Files/2007-2009%20College%20Catalog.pdf |
| Curriculum | http://www.lbwcc.edu/cms/page.aspx?pageid=446 |
| STARS | http://www.lbwcc.edu/cms/page.aspx?pageid=309 |

**Northeast Alabama CC (NEC)**

| | |
|---|---|
| CC URL | www.nacc.edu |
| CC City | Rainsville |
| Dept name | Business and Computer Science |
| Dept URL | http://www.nacc.edu/study/business_computer_science.htm |
| Cat Name | Catalog 2008-2009 |
| Cat URL | http://www.nacc.edu/catalog/catalog09.htm |
| Curriculum | http://www.nacc.edu/assessment/program_requirements/AS0809_1.pdf |
| STARS | http://www.nacc.edu/study/stars.htm |

**Northwest-Shoals CC (NWS)**

| | |
|---|---|
| CC URL | www.nwscc.edu |
| CC City | Muscle Shoals |
| Dept name | Computer Information |
| Dept URL | http://www.nwscc.edu/cisweb/cishome.htm |
| Cat Name | 2008-2009 Catalog and Student Handbook |
| Cat URL | http://www.nwscc.edu/Catalog0809/catalog.html |
| Curriculum | http://nwscc.edu/catalog0607/transfer_cis.pdf |
| STARS | http://www.nwscc.edu/students.html |

**Shelton State CC (SHC)**

| | |
|---|---|
| CC URL | www.sheltonstate.edu |
| CC City | Tuscaloosa |
| Dept name | Business |
| Dept URL | http://www.sheltonstate.edu/content.aspx?PageID=182 |
| Cat Name | College Catalog Fall 2007-Summer 2009 |
| Cat URL | http://www.sheltonstate.edu/userfiles/File/catalog/Fall%202007%20-%20Summer%202009/Shelton%20College%20Catalog%2007_09.pdf |
| Curriculum | http://www.sheltonstate.edu/userfiles/File/catalog/Fall%202007%20-%20Summer%202009/degree%20and%20cert%20requirements.pdf |
| STARS | http://www.sheltonstate.edu/content.aspx?PageID=139 |

**Snead State CC (SND)**

| | |
|---|---|
| CC URL | www.snead.edu |
| CC City | Boaz |
| Dept name | Math and Technology |
| Dept URL | www.snead.edu/academics/departments.asp |
| Cat Name | Official General Catalog (May2009) |
| Cat URL | http://www.snead.edu/ContentDocMaint/GetDocument.asp?ID=293 |
| Curriculum | pp. 95-96, 104-105 if catalog |
| STARS | pp. 69-70 of catalog |

**Southern Union State CC (SOU)**

| | |
|---|---|
| CC URL | www.suscc.edu |
| CC City | Wadley |
| Dept name | Academic Division |
| Dept URL | http://www.suscc.edu/SubTopicPages/InstructionalDivisions/Academic/AcademicDivHomePage.cfm |
| Cat Name | 2008-2009 Offical College Catalog |
| Cat URL | http://www.suscc.edu/PDFFiles/SUcatalog2008.pdf |
| Curriculum | pp. 142-143 of catalog |
| STARS | link in Enrolled Students menu |

**Wallace State CC Dothan (WSD)**

| | |
|---|---|
| CC URL | www.wallace.edu |
| CC City | Dothan |
| Dept name | Educational Programs - Academic |
| Dept URL | http://www.wallace.edu/programs/academic/ |
| Cat Name | 2008/2009 College Catalog and Student Handbook |
| Cat URL | http://www.wallace.edu/programs/Catalog.pdf |
| Curriculum | p. 61 of catalog |
| STARS | http://www.wallace.edu/current_students.htm |

**Wallace State CC Hanceville (WSH)**

| | |
|---|---|
| CC URL | www.wallacestate.edu |
| CC City | Hanceville |
| Dept name | Computer Science |
| Dept URL | http://www.wallacestate.edu/programs/academic/computer-science.html |
| Cat Name | Catalog 2007-2008 |
| Cat URL | http://www.wallacestate.edu/fileadmin/user_upload/WallaceState/documents/general/WSCC_2007-2008_catalog_cover.pdf |
| Curriculum | pp. 73-75 of catalog |
| STARS | http://www.wallacestate.edu/stars-guide.html |

**Wallace State CC Selma (WSS)**

| | |
|---|---|
| CC URL | www.wccs.edu |
| CC City | Selma |
| Dept name | Computer Information Systems (CIS) |
| Dept URL | http://www.wccs.edu/index.php?pages/cisdept.html |
| Cat Name | General Catalog and Student Handbook 2007-2010 |
| Cat URL | http://www.wccs.edu/files/2007-2010%20Catalog%20as%20modified%20on%207.11.07.pdf |
| Curriculum | p. 45 of catalog |
| STARS | http://www.wccs.edu/index.php?pages/sss.html |

Alabama Public 4-year Universities' Reference Information.

| | **Legend** |
|---|---|
| Univ URL | University web address |
| Area V | University STARS Area V information web address |
| Catalog | University course catalog web address |

| | **Alabama A&M U (AA&MU)** |
|---|---|
| Univ URL | www.aamu.edu |
| Area V | http://www.aamu.edu/Admission/STARS/transfer_info.htm |
| Catalog | http://www.aamu.edu/acadaffairs/BULLETIN--_2008-2011.pdf |

| | **Alabama State U (ASU)** |
|---|---|
| Univ URL | www.alasu.edu |
| Area V | http://www.alasu.edu/areav/default.aspx?id=82 |
| Catalog | http://www.alasu.edu/records/applications/documentlibrary/18673%20ASU%202004-2006%20LR.pdf |

| | **Athens State U (ATHENS)** |
|---|---|
| Univ URL | www.athens.edu |
| Area V | http://www.athens.edu/admissions/transfer.php |
| Catalog | http://www.athens.edu/catalog/index.html |

| | **Auburn U (AU)** |
|---|---|
| Univ URL | www.auburn.edu |
| Area V | http://www.auburn.edu/areav/engine.htm |
| Catalog | http://www.auburn.edu/student_info/bulletin/2009_bulletin.pdf |

| | **Auburn U Montgomery (AUM)** |
|---|---|
| Univ URL | www.aum.edu |
| Area V | http://www.aum.edu/uploadedFiles/Student_Life/Student_Services/Student_Records/Math%20MajorComputer%20Sciences.pdf |
| Catalog | http://www.aum.edu/uploadedFiles/Academics/Catalogs/Cat_UG_Sciences_08.pdf |

| | **Jacksonville State U (JSU)** |
|---|---|
| Univ URL | www.jsu.edu |
| Area V | http://www.jsu.edu/transfer/cs_se.html |
| Catalog | http://www.jsu.edu/depart/undergraduate/catalog/ |

| | **Troy U (TROY)** |
|---|---|
| Univ URL | www.troy.edu |
| Area V | http://www.troy.edu/area5/majors/Computer%20Science.html |
| Catalog | http://www.troy.edu/catalogs/0809undergrad/index.html |

**U of Alabama (UA)**

| | |
|---|---|
| Univ URL | www.ua.edu |
| Area V | http://coeweb.eng.ua.edu/future_students/computerscience.htm |
| Catalog | http://catalogs.ua.edu/catalog08/ |

**U of Alabama , Birmingham (UAB)**

| | |
|---|---|
| Univ URL | www.uab.edu |
| Area V | http://www.app.uab.edu/Area_V/Computer_Science.pdf |
| Cat | http://www.catalog.uab.edu/ |

**U of Alabama , Huntsville (UAH)**

| | |
|---|---|
| Univ URL | www.uah.edu |
| Area V | http://www.uah.edu/main/transfer1/areaV/computerSci.html |
| Catalog | http://www.uah.edu/main/catalogs/Cat07_09/ugCat07_09.pdf |

**U of Montevallo (UM)**

| | |
|---|---|
| Univ URL | www.montevallo.edu |
| Area V | na |
| Catalog | http://www.montevallo.edu/undergrad/ |

**U of North Alabama (UNA)**

| | |
|---|---|
| Univ URL | www.una.edu |
| Area V | http://www.una.edu/areav/arts-sciences/computer-science.html |
| Catalog | http://www.una.edu/catalog/catalogs/UNACatalog2009-2010.pdf |

**U of South Alabama (USA)**

| | |
|---|---|
| Univ URL | www.southalabama.edu |
| Area V | http://www.southalabama.edu/admissions/transfer/al/cs.html |
| Catalog | http://www.southalabama.edu/bulletin/ |

**U of West Alabama (UWA)**

| | |
|---|---|
| Univ URL | www.uwa.edu |
| Area V | http://www.uwa.edu/academics/areas/Registrar/transfers/mathcis.aspx |
| Catalog | http://www.uwa.edu/academics/catalog/undergraduate.aspx |

# Appendix C

Survey of Usage of Software Engineering Principles and Concepts

I invite you to participate in my research study to determine what software engineering principles are taught in the first two semesters in Alabama state community college and university computer science programs.

The survey will take 10-20 minutes of your time. You will be asked to select the highest level of usage of Software Engineering principles and related terms in the computer programming courses at your institution.

The collected information will help identify possible revision to existing computer science curricula to include software engineering. The curricula will be made available to computer science faculty.

Data obtained from this study will remain anonymous. The summary of the information collected from this survey will be used to fulfill an education requirement, published in a professional journal, and presented at a professional meeting. If optional identification information is given, it will be stored separately from the collected survey data and not part of the survey results. **There are 10 Sections in this survey.**

In Sections 1-8, you are asked to select the highest level of usage for several terms related to Software Engineering.
In Section 9, you may provide comments.
In Section 10, you may provide (optional) contact information for further communication.

To start the survey or choose not to participate, click "Yes" or "No" below

- ○ Yes, I would like to participate in the survey.
- ○ No, I do not want to participate in the survey. You will advance to the end of the survey.

[ >> ]

Please indicate if you teach at a 2-year or 4-year college or university before continuing with the survey.

- ○ 2-year, or less than 4 year program
- ○ 4-year, or more years program

[ >> ]

**There are 10 Sections in this survey.**
In Section 9, you may provide comments.
In Section 10, you may provide (optional) contact information for further communication.
**Thank you for your participation.**

**A response is required for each term in this section.**

## Section 1: Software Engineering Principles terms and concepts

**For each term, indicate the level of educational objective expected for a student who finishes the first two semesters of your computer science program.**

| | Not used in courses | Remember | Understand | Apply |
|---|---|---|---|---|
| SOFTWARE DESIGN | ○ | ○ | ○ | ○ |
| Design patterns | ○ | ○ | ○ | ○ |
| Sequence diagram | ○ | ○ | ○ | ○ |
| CRC cards (Class Responsibilty Collaboration) | ○ | ○ | ○ | ○ |
| Problem solving | ○ | ○ | ○ | ○ |
| - Define requirements & specifications | ○ | ○ | ○ | ○ |
| - Define input & output | ○ | ○ | ○ | ○ |
| - Design test cases | ○ | ○ | ○ | ○ |
| - Define algorithm | ○ | ○ | ○ | ○ |
| Stepwise refinement | ○ | ○ | ○ | ○ |
| Pseudocode | ○ | ○ | ○ | ○ |
| Flowchart | ○ | ○ | ○ | ○ |
| - Documentation | ○ | ○ | ○ | ○ |

>>

**A response is required for each term in this section.**

**Section 2: Software Engineering Principles terms and concepts continued**

For each term, indicate the level of educational objective expected for a student who finishes the first two semesters of your computer science program.

| | Not used in courses | Remember | Understand | Apply |
|---|---|---|---|---|
| SOFTWARE PROCESSES | ○ | ○ | ○ | ○ |
| Software lifecycle | ○ | ○ | ○ | ○ |
| - Waterfall model | ○ | ○ | ○ | ○ |
| - Incremental | ○ | ○ | ○ | ○ |
| Deliverables | ○ | ○ | ○ | ○ |
| Process maturity models | ○ | ○ | ○ | ○ |
| SOFTWARE EVOLUTION | ○ | ○ | ○ | ○ |
| Code reuse | ○ | ○ | ○ | ○ |
| Code refactoring | ○ | ○ | ○ | ○ |

>>

**A response is required for each term in this section.**

**Section 3: Software Engineering Principles terms and concepts continued**

For each term, indicate the level of educational objective expected for a student who finishes the first two semesters of your computer science program.

| | Not used in courses | Remember | Understand | Apply |
|---|---|---|---|---|
| SOFTWARE VALIDATION | ○ | ○ | ○ | ○ |
| Data validation | ○ | ○ | ○ | ○ |
| Testing (unit, integration, systems & acceptance) | ○ | ○ | ○ | ○ |
| Test plan | ○ | ○ | ○ | ○ |
| Test driver | ○ | ○ | ○ | ○ |
| SOFTWARE PROJECT MANAGEMENT | ○ | ○ | ○ | ○ |
| Lines of code estimate | ○ | ○ | ○ | ○ |
| Coding time estimate | ○ | ○ | ○ | ○ |
| Configuration management-version control | ○ | ○ | ○ | ○ |
| Documentation | ○ | ○ | ○ | ○ |

>>

A response is required for each term in this section.

## Section 4: Good practices terms and concepts

For each term, indicate the level of educational objective expected for a student who finishes the first two semesters of your computer science program.

| | Not used in courses | Remember | Understand | Apply |
|---|---|---|---|---|
| Object oriented programming | ○ | ○ | ○ | ○ |
| Modularization | ○ | ○ | ○ | ○ |
| Classes & methods | ○ | ○ | ○ | ○ |
| Iterators | ○ | ○ | ○ | ○ |
| Cohesion | ○ | ○ | ○ | ○ |
| Coupling | ○ | ○ | ○ | ○ |
| Scalability | ○ | ○ | ○ | ○ |
| Naming conventions | ○ | ○ | ○ | ○ |
| Abstract Data Types (ADT) | ○ | ○ | ○ | ○ |
| Encapsulation | ○ | ○ | ○ | ○ |
| Information hiding | ○ | ○ | ○ | ○ |
| Class hierachies | ○ | ○ | ○ | ○ |
| Inheritance | ○ | ○ | ○ | ○ |
| Polymorphism | ○ | ○ | ○ | ○ |
| Metrics | ○ | ○ | ○ | ○ |
| Naming conventions | ○ | ○ | ○ | ○ |

**A response is required for each curriculum guide in this section.**

**Section 5: The following are curriculum guides developed by the ACM, IEEE-CS, and/or Two-Year College Education Committee.**

How familiar are you with each?

| | Not Familiar | Somewhat familiar | Familiar | Very familiar |
|---|---|---|---|---|
| Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science | ○ | ○ | ○ | ○ |
| Computing Curricula 2009: Guidelines for Associate-Degree Transfer Curriculum in Computer Science | ○ | ○ | ○ | ○ |
| Computing Curricula 2001: Computer Science | ○ | ○ | ○ | ○ |
| Computer Science Curriculum 2008: An Interim Revision of CS 2001 | ○ | ○ | ○ | ○ |

>>

**In this section, check all that apply.**
**Selecting "Other" requires a text response.**

**Section 6: Which Integrated Design Environment(s) (IDE) is/are used in the first and second computer programming courses?**

| | Used in 1st semester computer programming course | Used in 2nd semester computer programming course | Used in 3rd semester computer programming course |
|---|---|---|---|
| Eclipse | ☐ | ☐ | ☐ |
| NetBeans | ☐ | ☐ | ☐ |
| Rational | ☐ | ☐ | ☐ |
| Visual Studio | ☐ | ☐ | ☐ |
| jGrasp | ☐ | ☐ | ☐ |
| BlueJ | ☐ | ☐ | ☐ |
| DrJava | ☐ | ☐ | ☐ |
| JKarelRobot | ☐ | ☐ | ☐ |
| Alice | ☐ | ☐ | ☐ |
| Other IDE, enter name | ☐ | ☐ | ☐ |
| No IDE, Run from command line | ☐ | ☐ | ☐ |

>>

In this section, check all that apply.
Selecting "Other" requires a text response.

**Section 6: Which Integrated Design Environment(s) (IDE) is/are used in the first and second computer programming courses?**

| | Used in 1st semester computer programming course | Used in 2nd semester computer programming course | Used in 3rd semester computer programming course |
|---|---|---|---|
| Eclipse | ☐ | ☐ | ☐ |
| NetBeans | ☐ | ☐ | ☐ |
| Rational | ☐ | ☐ | ☐ |
| Visual Studio | ☐ | ☐ | ☐ |
| jGrasp | ☐ | ☐ | ☐ |
| BlueJ | ☐ | ☐ | ☐ |
| DrJava | ☐ | ☐ | ☐ |
| JKarelRobot | ☐ | ☐ | ☐ |
| Alice | ☐ | ☐ | ☐ |
| Other IDE, enter name | ☐ | ☐ | ☐ |
| No IDE, Run from command line | ☐ | ☐ | ☐ |

>>

In this section, check all that apply.
Selecting "Other" requires a text response.

**Section 7: Which programming language(s) is/are taught in the first and second computer programming courses?**

| | Taught in 1st semester computer programming course | Taught in 2nd semester couputer programming course | Taught in 3rd semester computer programming course |
|---|---|---|---|
| C | ☐ | ☐ | ☐ |
| C++ | ☐ | ☐ | ☐ |
| Cobol | ☐ | ☐ | ☐ |
| Fortran | ☐ | ☐ | ☐ |
| Java | ☐ | ☐ | ☐ |
| Python | ☐ | ☐ | ☐ |
| Visual Basic | ☐ | ☐ | ☐ |
| Other, enter name ____ | ☐ | ☐ | ☐ |

>>

In this section, enter an approximate percentage for each catagory.
Selecting "Other" requires a text repsonse.
The total should equal 100%.

**Section 8: Where do student who complete your computer science program go after graduation?**

| | |
|---|---|
| Transfer to or continue at a 4-year university | 0 |
| Computer science industry | 0 |
| Other ____ | 0 |
| **Total** | 0 |

>>

**Section 9: This section allows you to enter comments about this survey, teaching the first two semesters of computer programming, teaching software engineering, etc.**
**Comments are appreciated but optional.**

**Section 10: Thank you for your participation in this survey.**

**A summary of the information collected via this survey will be publish on my website:**

http://www.eng.auburn.edu/user/hundljh/

**You can receive a copy via email by providing the following information.**
**Providing contact information is optional and will be stored independently of the survey data.**

Email

Name

We thank you for your time spent taking this survey.
Your response has been recorded.

# Appendix D

Teaching Software Engineering

Summer 2011 Special Topics Course

COMP 7976 COMP8970
3-6:30pm Wednesdays 1120 Shelby Center

**INSTRUCTOR:**

**TEACHING ASSOCIATE:**

**COURSE GOALS AND OBJECTIVES:** This course examines software engineering from an instructional perspective. Its purpose is to give students an exposure to explaining fundamental software engineering concepts to those new to the field.

**PREREQUISITES:**
• Graduate standing
• Familiarity with general software engineering

**RATIONALE:**
Postgraduate instruction traditionally focuses on developing advanced subject specialty skills, offering research experiences, and fostering methods of disciplined thought. Graduate students look to careers in higher education or industry but never receive training on how to explain the complex concepts of engineering software to students, coworkers, supervisors, subordinates, etc. This course is designed to provide insight into how to teach software engineering concepts at the introductory level.

**REQUIRED RESOURCES:**

TRIPP, L. (chair), et al. 2004. IEEE Computer Society Professional Practices Committee. 2004. Guide to the software engineering body of knowledge Project (SWEBOK) and the Guide. *IEEE Computer Society Press*, Los Alamitos, CA, (2004). Available on Blackboard and at http://www.swebok.org.

CAMPBELL, R. (chair) et al. 2005. Computer curricula 2005: Guidelines for associate-degree transfer curriculum in software engineering. *IEEE Computer Society Press and ACM Press*, (August, 2005). See Blackboard.

HAWTHORNE, E. (chair) et al. 2009. Computing curricula 2009: Guidelines for associate-degree transfer curriculum in computer science. *ACM Two-Year College Education Committee. ACM and IEEE Computer Society* (2009). See Blackboard.

LE BLANC, R. and SOBEL, A. (chairs) et al. 2004. Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. *IEEE Computer Society Press and ACM Press* (23 August 2004). Available on Blackboard and at http://sites.computer.org/ccse/SE2004Volume.pdf

More information at ACM Committee for Computing Education in Community Colleges. http://www.acmtyc.org/

**COURSE STRUCTURE:**
Class 1 will entail 1) a discussion of the difference between computer science and software engineering; 2) an examination of SWEBOK; and 3) an examination of model curricula for software engineering.

Classes 2 through 9 will be structured as follows:

| Activity | Desired Outcomes | Responsibilities |
|---|---|---|
| Primer | °To bring everyone up to speed on the knowledge area.<br>°To identify cardinal elements of the knowledge area. | COMP8970 Team:  Presentation. |
| Prerequisite Knowledge Graph | °To identify prerequisite skills required to apply the knowledge area. | COMP8970 Team:  Class discussion |
| Pedagogy | °To identify what others have done to teach the knowledge area<br>°To identify where to find useful information on how to explain the knowledge area.<br>°To identify instructional pitfalls to avoid when explaining the knowledge area.<br>°To place teaching the knowledge area in the context of 1) the novice instructor, 2) the CS1/CS2 student, and 3) the adult learner. | COMP8970 Team:  Presentation |
| Tool Support | °To identify how automated tools can assist carrying out the knowledge area.<br>°To identity available tools, to include functionality, acquisition expenses, and effort to inject into the classroom environment. | COMP7976 Team:  Presentation |
| Sample Learning Activities | °To propose one or more activity that fortifies learning.<br>°To identify activities which can be used in the workshop. | COMP8970 Team:  Class discussion |
| Curriculum Integration | °To map key knowledge area concepts into CS1/CS2. | COMP8970 Team:  Brainstorm |
| Reading | °To introduce the next topic and how it can be integrated into CS1/CS2. | Instruction Team:  Class discussion |

Class 10 will entail planning a workshop for regional community college instructors on how fundamental software engineering principles can be taught at the introductory level.

**CALENDAR (subject to change):**

| Class | Topic | Class | Topic |
|---|---|---|---|
| - | Pre-assigned readings | 29 June | Software quality |
| 25 May | Introduction | 6 July | Software requirements |
| 1 June | SwE Process | 13 July | Software Configuration Managemen |
| 8 June | Software testing | 20 July | SwE Management |
| 15 June | Software construction | 27 July | Workshop planning |
| 22 June | Software design | 30 July | Workshop |

**COURSE REQUIREMENTS -- COMP8970**

**All of the following must be completed for a grade of B:**

- Discussion Leader: Gather discussion items relating to an assigned SwE knowledge area, assign readings, and lead the class discussion for the entire period, prepare supplemental material. Students classified as "PHD" may be asked to lead more than one discussion depending on the number of people in the class.
- Educational Autobiography: Prepare a 4-5 page description of your educational/professional history describing factors you think were most influential in shaping your learning and behavior with respect to software engineering.
- Journal of Teaching Reflections: Maintain a journal over the course of the semester that documents reflections on material related to class. At least two entries must be written each week.
- Interviews: Interview an instructor and a novice student on their respective perspectives on an assigned SwE knowledge area. Deliver a 10-15 minute report on your interviews to the class. Prepare a 4-5 page summary of the interviews.
- Participation: Miss no more than one class. Make at least three contributions a week to the course discussion forum.

**Additional requirements for a grade of A:**

- Activity Design: Develop a curriculum module for an assigned SwE knowledge area that includes recommendations as to what concepts can be integrated into CS1/CS2, where those concepts can be injected into CS1/CS2, ideas for activities that reinforce the concept, at least one detailed description of a sample reinforcing activity, suggestions for tools. The curriculum module must be suitable for the end-of-course workshop.

**COURSE REQUIREMENTS -- COMP7976**

**All of the following must be completed for a grade of B:**

- Video Presentation: Submit a 15-25 minute prerecorded presentation on tools that support an assigned knowledge area. Include information on tool functionality, acquisition expenses, and efforts to inject into the classroom environment. Tool demonstrations are highly encouraged.
- Educational Autobiography: Prepare a 4-5 page description of your educational/professional history describing factors you think were most influential in shaping your learning and behavior with respect to software engineering.
- Journal of Teaching Reflections: Maintain a journal over the course of the semester that documents reflections on material related to class. At least two entries must be written each week.
- Participation: Make at least three contributions a week to the course discussion forum.

**Additional requirements for a grade of A:**

- Interviews: Interview a trainer and a novice learner on their respective perspectives on an assigned SwE knowledge area. Prepare a 4-5 page summary of the interviews.
- Activity Design: Develop a curriculum module for an assigned SwE knowledge area that includes recommendations as to what concepts can be integrated into CS1/CS2, where those concepts can be injected into CS1/CS2, ideas for activities that reinforce the concept, at least one detailed description of a sample reinforcing activity, suggestions for tools.

**ADDITIONAL COURSE INFORMATION:**

**Academic Honesty:** You will be held accountable to the Academic Honesty policies described in the Tiger Cub. Cheating will not be tolerated. Unless otherwise directed, it is considered cheating to give or receive material that is part of an assignment solution; work so closely with someone that you ideas, solutions, and work are indistinguishable from theirs; use, i.e., copy, the work of others as your own. This includes material copied from the Internet.

**Special Accommodations:** Students needing special accommodations (for school events, personal circumstances, disabilities, etc.) should bring that need to my attention as soon as possible, along with the appropriate written verification.

**Electronic Devices:** Electronic devices such as cell phones, pagers, and alarms should be turned off or set to silent mode throughout class. If your phone rings audibly, it is to be answered by your neighbor. Please do not text message during class. Laptops may be used in class, but only for purposes relating to the course itself. Please do not play games, answer e-mail, do homework, browse the web, etc. during class.

**Civility Statement:** Honest, open, and candid opinions are welcome; however, everyone is expected to show respect.

**Attendance:** I expect you to attend all classes and to particulate in all aspects of class discussion. Only valid university excuses will be accepted as legitimate reasons for missing class. These include illness (with a written medical excuse); personal or family emergencies; religious holidays (with advance notice); subpoena for court appearance (with written documentation); and university-related travel (with an official letter). Missing two or more classes with an unexcused absence will result in a grade of "FA".

**Email policy:** Please observe conventional rules of e-mail etiquette when communicating with me electronically. In particular, please

- be courteous
- sign your e-mail
- proof-read your e-mail
- don't expect me to pre-grade your assignments
- don't flag e-mails as urgent unless they are truly so
- be reasonable about when you expect me to respond to e-mail. I try to respond within one business day

Appendix E

# Software Process Curriculum Module

## Preface

The purpose of these teaching modules is to demonstrate how software engineering knowledge area and principles can be imprinted into teaching computer science at the CS1 and CS2 levels. It is not intended to replace material and topics that are necessary in the curricula. It is hoped that the information presented in this module will enhance the learning experience of the students.

## Module Description

This module presents an introduction software process. Software engineering process refers to the technical and managerial activities that are performed during software acquisition, development, maintenance, and retirements. It is concerned with meta-data: definition, implementation, assessment, measurement, management, change, and improvement. (SWEBOK)

In SWEBOK, Software Engineering Process is divided into the sub-knowledge area topics show below. This module will provide assistance for introducing the some topics at the CS1 and CS2 levels.

Process Implementation and Change
Process Infrastructure
Software Process Management Cycle
Models for Process Implementation and Change
Practical Considerations
Process Definition
Software Life Cycle ModelsCS1
Software Life Cycle ProcessesCS1
Notations for Process Definitions
Process Adaptation
Automation
Process Assessment
Process Assessment Models
Process Assessment Methods
Process and Product Measurement
Process Measurement
Software Products MeasurementCS2
Quality of Measurement Results
Software Information Models
Process Measurement Techniques

## Philosophy

Software process is an integral part of software development. It can assist in learning and teaching by providing:

- a set of steps for approaching software development
- a mechanism for accountability
- an engineering mindset of problem solving
- a factory of artifacts
- a reminder of best practices
- a communication tool

**Outcomes**

Through the material covered in this module, students should:

- Identify a problem, define solutions, and develop algorithms to attain the optimal solution.
- Recognize that software systems can be produced according to a systematic model.
- Explain alternative ways to organize software development efforts
- Describe the software engineering process using standard metrics.

**Prerequisite Knowledge**

The CS1 level of subject matter presented in this module requires no computer science prerequisite. CS1 is the prerequisite for CS2 level.

**Outline**

3) CS1
    d) Introduction
        i) Software Engineering
        ii) Software Process
        iii) Software Process Helps
        iv) A Software Engineering Process
    e) A Problem Solving Approach
    f) Use CS1 Activity 1

4) CS2
Recap the CS1 introduction
    a) Software Metrics
    b) Vocabulary
        i) Measure
        ii) Measurement
        iii) Metrics
        iv) Indicator
    d) Measurable Attributes of Software Engineering
    e) Measuring Individual Performance - CS2 Activity

**Annotated Outline**

3) CS1
    a) Introduction
        i) Software Engineering
Applies a systematic, disciplined, quantifiable approach (or process) to the development, operation, and maintenance of software.
        ii) Software Process
The sequence of steps to develop or maintain software
        iii) Software Processes Help
            (1) Boost the probability of product quality
            (2) Identify the principle activities of doing a job
            (3) Separate routine from complex tasks
            (4) Facilitate tracking and measuring performance

(5) Provide orderly mechanism for learning
(6) Establish corporate memory
(7) Create a defined baseline for improvement
(8) Put everyone on the same page

iv) A Software Engineering Process
(1) Define the function of the program
(2) Sketch out a design
(3) Pseudo code – not ready to write source code (a program), yet
(4) Discuss with all parties
(5) Modify
(6) Repeat
(7) After the design is agreed upon,
(a) Write the real program using a computer programming language
(b) Test – run the program with known data
(c) Modify – correct defects (errors)
(d) Repeat

b) A Problem Solving approach

A simple introduction to the process of software development is using a systematic approach to problem solving.

i) Understand the problem.
*Learn about the problem domain. If necessary, break a large task into multiple smaller tasks*

ii) Analyze the problem requirements.
*Specify input values (knowns) and required output values (unknowns). Include the units. Identify the relevant formulae needed for computations and necessary constants values, e.g., gravity or pi.*

iii) Work a hand example.
*This will (1) identify the steps needed to solve the problem and (2) a set of input and resulting output that can be used to test your software, later.*

iv) Develop an algorithm to solve the problem.
*Record the steps used to solve the hand example. If necessary, divide steps into multiple simpler steps to provide a clear solution.*

v) Implement the algorithm.
*Now, it is time to write a computer program that follows the steps in the algorithm to solve the problem. The statements in the algorithm can be used as comments as a guide for writing code in the program.*

vi) Test and verify the program solution.
*Run the program correcting any errors that exists. Use the input values from the hand example to verify that the solution is correct.*

vii) Maintain and update the program.
*This step is necessary when new requirements are added or there is a policy change that affects the problem solution.*

c) Use CS1 Activity 1 to demonstrate the problem solving approach. See the Activities section below. Note: Activities 2 and 3 may be use later with the introduction of selection and repetition.

4) CS2

a) Recap the CS1 introduction

b) Software Metrics

i) A key element of any engineering process is measurement. Measures help to better understand the attributes of a product and to assess its quality. Unlike other engineering disciples, software engineering is not grounded in the basic quantitative laws of physics, like voltage, mass, velocity, or temperature. What are the measurable attributes of software engineering work products?

ii) What are software engineering products? requirements and design models, source code, and test cases.

c) Vocabulary

i) In software engineering, **measure**, **measurement**, and **metrics** are often used interchangeably.

ii) A **measure** provides a quantitative **indication** of the extent, amount, dimension, capacity, or size of some attribute of a product or process.

iii) **Measurement** is the act of determining a measure.

iv) **Metric** is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.

v) A software engineer collects measures and develops metrics so that indicators will be obtained. An **indicator** is a metric or combination of metrics that provides insight into the software process, a software project, or the product itself

d) Measurable attributes of software engineering

i) Lines of code (LOC) and LOC per hour are metrics for planning software development

(1) What are the measurable attribute of software engineering work products? We will look at source code because students are familiar with this product. Source code has size. If we know the average length of a program for solving a particular type problem and the average number of lines of code we write in an hour, we can estimate how long it would take to produce this type product.

ii) Number and type of mistakes (defects) are also metrics to track improvement

(1) If we always wrote code with no defects, our production level of producing code would be pretty good. But, we all make mistakes. Finding and correcting them take time and lowers the actual number of LOC per hour.

e) Measuring Individual Performance - CS2 Activity

i) How can we improve our LOC per hour? The obvious way is to make fewer mistakes. To help us reduce the number of mistakes, we need to note the types of mistakes that we make and try to not make them. One way to approach reducing the number of defects in our code is to keep a log of the defects…and how many. See the tables below for the defect log and instructions.

ii) Completing an assignment is not (usually) done in one seating without interruptions. A time log will help you record how much time is spent in each stage. See the tables below for the time log and instructions.

iii) Maintaining a record of LOC, time and defects, we can monitor improvement.

**Teaching Resources**

Process Worksheet
Defect Recording Log
Time Recording Log

**Teaching Techniques**

CS1 activities

- Lecture with slides
- Blank worksheet for students to complete individually or as a group to solve another familiar problem, like "find the distance between two points." Lead a class discussion the solutions using a document camera with students providing the needed information. Students can be asked to lead the discussion or report on their solution.

CS2 activity

- Provide a section of code with defects and lead the students in finding and typing the defects.

**Tool Support**

Process Dashboard – used with PSP

- Not User Friendly
- Describes Psp Scripts

- Does Calculations For You
- Better than using PSP manually
-

Eclipse Process Framework
- OpenUP process (also XP and scrum)
- Describes steps to follow
- Can attach tools to framework
- More sufficient than dashboard
- Helps enact process
- Guides you through process correctly
- Umbrella tool that walk you through a process

**Glossary**

Measure – provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process.

Measurement – the act of determining a measure.

Metric – a quantitative measure of the degree to which a system, component, or process possesses a given attribute.

Indicator – a metric or combination of metrics that provides insight into the software process, a software project, or the product itself

Software life cycle – a typical sequence of phased activities that represent the various stages of engineering through which software system passes

Software process –
the network of object states and transitional events that represent the production of a software system in a form suitable for computational encoding and processing

**Bibliography**

HUMPHREY, W. 2000. “The Baseline Personal Process” in A Discipline for Software Engineering. Addison-Wesley, Boston.

LE BLANC, R. and SOBEL, S. (chairs) et al. 2004. Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. IEEE Computer Society Press and ACM Press (23 August 2004). Available at http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/education/cc2001/SE2004Volume.pdf

TRIPP, L. (chair), et al. 2004. IEEE Computer Society Professional Practices Committee. 2004. Guide to the software engineering body of knowledge Project (SWEBOK) and the Guide. IEEE Computer Society Press, Los Alamitos, CA, (2004). Available at http://www.swebok.org.

**Suggested Course Activities**

**CS1 Activity 1**
First assignment is solving a problem that involves an equation. Introducing the assignment should include a class discussion of the steps necessary to solve this problem.

To introduce a systematic problem solving strategy, talk through solving an example using the steps. Because solving for the roots of a quadratic equation is familiar, it is a good example to use at multiple stages during the course. These multiple stages present a sequence of activities that allows students to revisit and modify existing code and observe how changes in requirements affect the code.

No software engineering tools other than the IDE will be introduced for this series of activities.

Understand the problem.
Find the real roots of a quadratic equation: $ax^2 + bx + c = 0$

Analyze the problem requirements.
3 coefficients: a, b, c

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Work a hand example.
Results from hand calculations:

| input | | | output |
|---|---|---|---|
| a | b | c | x1x2 |
| 1 | 3 | -4 | -41 |
| 2 | -4 | -3 | -0.582.58 |

Develop an algorithm.
Get coefficients: a, b, c.
Compute roots: x1, x2
Display results

Implement the algorithm.
This is where the program is written. The algorithm can be used comments in the program write the computer program statements.
Using the IDE that the students use, type the program.

NOTES:
1st time, use assignment statement for input
2nd time, use user input
Later, functions can be used for each step
These is an example of design alternatives.

Test and verify the program solution.
This can be an opportunity to discuss types of errors by including errors in the program.
Compile program and correct errors.
Run program using input from hand example.
If results are not correct, review set step in algorithm and program.

Maintain and update the program.
There will probably not be a required response for this step.

**Process Worksheet**

Understand the problem.

Analyze the problem requirements.

Work a hand example.
Show work and results from hand calculations:

Develop an algorithm.

Implement the algorithm.
This is where the program is written. Start by copying and pasting the algorithm into the IDE editor window and marking the statements as comments. These comments will be a guide for writing the computer program statements.

Test and verify the program solution.
This is where students will run the program to determine if it solves the problem correctly.

Maintain and update the program.
No required response for this step.

CS1 Activity 2

To introduce Selection, reuse the CS1 Activity 1 example and include the restrictions on the coefficients to find the real roots of a quadratic equation.

Understand the problem.
Find the real roots of a quadratic equation: $ax^2 + bx + c = 0$

Analyze the problem requirements.
3 coefficients: a, b, c
Restrictions on input:
a != 0
D >= 0

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Work a hand example.
Results from hand calculations:

| input | | | output |
|---|---|---|---|
| a | b | c | x1x2 |
| 1 | 3 | -4 | -41 |
| 0 | 7 | 6 | not a quad eq |
| 1 | 3 | 3 | -sqrt, not a real root |

NOTE: Sample input includes values to test restrictions

Develop an algorithm.
Get coefficients: a, b, c.
If a != 0, compute D
If D >= 0, compute roots: x1, x2
display results

Implement the algorithm.
This is where the program is written. Start by copying and pasting the algorithm into the IDE editor window and marking the statements as comments. These comments will be a guide for writing the computer program statements.

NOTES:
Use user input to prepare students for input validation loops, next time.
Later, functions can be used for each step

Test and verify the program solution.
This can be an opportunity to discuss types of errors by including errors in the program.
Compile program and correct errors.
Run program using input from hand example.
If results are not correct, review set step in algorithm and program.

Maintain and update the program.
There will probably not be a required response for this step.

**CS1 Activity 3**

To introduce Repetition, reuse the CS1 Activity 2 and include the restrictions on the coefficients to find the real roots of a quadratic equation. Ask user to re-enter invalid coefficients values.

<u>Understand the problem</u>.
Find the real roots of a quadratic equation: $ax^2 + bx + c = 0$

<u>Analyze the problem requirements</u>.
3 coefficients: a, b, c
Restrictions on input:
a != 0
D >= 0

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

<u>Work a hand example</u>.
Results from hand calculations:

| input | | | output | |
|---|---|---|---|---|
| a | b | c | x1 | x2 |
| 1 | 3 | -4 | -4 | 1 |
| 0 | 7 | 6 | not a quad eq | |
| 1 | 3 | 3 | [-sqrt] | |

NOTE: Sample input includes values to test restrictions

<u>Develop an algorithm</u>.
While a == 0, get a
Get coefficient b, c
Compute D
If D < 0,
    else need new a, b, c
Compute roots: x1, x2
Display results

Design alternatives can be introduced at this stage of this example.

After giving student the steps to solving the assignment problems for the first few assignments, ask them to write and submit their own software development plan for the assignments.

**CS2 Activity**

After students have knowledge of problem solving, coding and types of errors, they can work on improving their software development skills. They will record the errors and time spent making corrections and time spent completing the assignment. Prior to this activity, students need to understand the types of errors: syntax, logic and, runtime.

The goal of this activity is for student to reduce common errors by being more aware of them during the coding process. Progress can be track during the semester.

The logs and instructions for using the logs used in this activity are an adaption of those found in HUMPHREY, W. 2000. "The Baseline Personal Process" in A Discipline for Software Engineering. Addison-Wesley, Boston.

| DEFECT RECORDING LOG INSTRUCTIONS | |
|---|---|
| Purpose | This form holds the data on each defect as you find and correct it. |
| General | Record in this log all defects found in review, compile, and test.<br>Record each defect separately and completely.<br>If you need additional space, use another copy of the form. |
| **Column** | |
| No. | Enter the defect number. For each program, this should be a sequential number starting with, for example, 1 or 001. |
| Date | Enter the date when the defect was found. |
| Type | Enter the defect type from the defect type list. Use your best judgment. |
| Fix defect | If you injected this defect while fixing another defect, record the number of the previously improperly fixed defect. |
| Fix time | Enter you best judgment of the time you took to fix the defect, i.e., in seconds, minutes. |
| Description | Write a brief description of the defect that is clear enough to later remind you about the error and help you to remember why you made it. |

| TIME RECORDING LOG INSTRUCTIONS | |
|---|---|
| Purpose | This form is for recoding the time spent doing the project. |
| General | Record all the time you spend on the project<br>Record the time in minutes.<br>Be as accurate as possible.<br>If you need additional space, use another copy of the form. |
| **Column** | |
| Date | Enter the date when the entry is made. |
| Start Time | Enter the time when you start working on a task. |
| Stop Time | Enter the time when you stop working on the task. |
| Interruption | Record any interruption time that was not spent on the task and the reason for the interruption.<br>It you have several interruptions, enter their total time. |
| Work Time | Enter the clock time you actually spent working on the task, less the interruption time. |
| Comments | Enter reasons for interruptions and other comments that may remind you of any unusual circumstances regarding this activity. |

## DEFECT RECORDING LOG*

Student ___________________________________Total # defects_______Start date___________

Class ________________ Assignment # ________Total fix time_______End date___________

| No. | Date | Fix defect | Fix Time | Descriptions |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

* adaption of defect log found in HUMPHREY, W. 2000. “The Baseline Personal Process” in A Discipline for Software Engineering. Addison-Wesley, Boston.

## TIME RECORDING LOG*

Student ________________________________ Total # time ______ Start date__________

Class _______________ Assignment # ________ End date__________

Work time = Stop time – Start time – Interruption
Comments may be use explain interruptions

| Date | Start Time | Stop Time | Interr-uption | Work Time | Comments |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

* adaption of time log found in HUMPHREY, W. 2000. "The Baseline Personal Process" in A Discipline for Software Engineering. Addison-Wesley, Boston.

Software Testing Curriculum Module

**Preface**

The purpose of these teaching modules is to demonstrate how software engineering knowledge area and principles can be imprinted into teaching computer science at the CS1 and CS2 levels. It is not intended to replace material and topics that are necessary in the curricula. It is hoped that the information presented in this module will enhance the learning experience of the students.

**Module Description**

Contained in this module is an introduction to software testing. IEEE's SWEBOK describes software testing as "an activity performed for evaluating product quality, and for improving it, by identifying defects and problems," which includes verifying the behavior of a program based on an appropriately selected set of test cases.

The SWEBOK divides Software Testing into the sub-knowledge area topics listed below. This module will provide assistance for introducing Software Testing concepts at the CS1 and CS2 levels.

Software Testing Fundamentals *
Testing-Related Terminology
Key issues
Relationships of Testing to Other Activities
Test Levels *
The Target of the Test CS1
Objectives of Testing
Test Techniques
Based on the Tester's Intuition and Experience CS1 and CS2
Specification-based CS2
Code-based CS1
Fault-based
Usage-based
Based on Nature of Application
Selecting and Combining Techniques CS1 and CS2
Test Related Measures
Evaluation of the Program Under Test
Evaluation of the Test Performed CS1 and CS2
Test Process
Practical Considerations
Test Activities CS1 and CS2

* Various concepts in the starred subsections are introduced at both the CS1 and CS2 levels.

**Philosophy**

Software testing is vital to the development of reliable software. Many students will test their programs in some fashion prior to submission but will often not have the a sufficient background in software testing to find even major faults in their final product. Teaching software testing techniques starting at the CS1 level not only increases the quality of their software but offers the following unique pedagogical benefits:

- Increases student confidence in proving the functionality of their software
- Decreases frustration and time spent on debugging when students test individual components early on

- Reinforces software design principles with a focus on program testability
- Increases student comprehension of their programs by focusing on problem solving
- Familiarity with software testing gives students an edge in later classes and upon entering the workforce

**Outcomes**

The material covered in this module is designed to provide students with the ability to:

- Select suitable inputs in order to verify the behavior of a program.
- Identify failures early and prevent common software faults
- Design and develop software with a focus on quality and testability.
- Demonstrate knowledge in the fundamentals of testing and its role in software development.

**Prerequisite Knowledge**

The CS1 material in this module requires no computer science knowledge prior to entering CS1. The CS2 level material assumes familiarity with the concepts introduced during CS1. Because the suggested course activities are designed for varying levels of experience, each activity includes specific prerequisite knowledge requirements.

**Outline**

1) Early CS1
    a) Introduction
        i.) What is software testing?
        ii) Why is software testing important?
        iii) Components of software testing
    b) Calculating expected outputs
    c) Fault log
    d) CS1 Activity 1: Calculating expected outputs and fault logging

2) Advanced CS1
    a) Levels of testing
    b) Unit testing
    c) Input selection: boundary conditions
    d) CS1 Activity 2: Unit testing and input selection

3) Early CS2
    a) Review: Unit testing and input selection
    b) Automated Unit Tests
    c) Test-Driven Development
    d) CS2 Activity 1A: TDD

4) Advanced CS2
    a) Review: Automated Unit Tests
    b) Non-functional requirements
    c) Regression testing
    d) CS2 Activity 1B

**Annotated Outline**

1) Early CS1
    a) Introduction
        i.) What is software testing?
        *Testing evaluates and improves the quality and reliability of software by detecting and preventing software defects.*
        ii) Why is software testing important?
        *Testing helps to ensure software quality and reliability. Testing saves time and money; faults that remain undetected until after the release of software are difficult to fix and can result in loss of business, law suits, and worse. Familiarity with software testing is an essential skill for software developers. Entire careers are centered around software testing and quality assurance.*
        iii) When should testing be performed?
        *Testing should play a part in software development at the earliest stages of planning. It is important to design programs for testability and to identify potential faults as early as possible. Testing should be carried out as the software is being developed, after the software is complete, and again when the software is released.*
    b) Calculating expected outputs
    *An essential component of testing your software is to choose a meaningful set of input values. For each input, the expected output should be calculated and compared against the actual output of your program. Test inputs and expected outputs can be determined before coding a piece of software begins.*
    c) Fault log
    *When the actual output of your program does not match the expected output, the underlying fault will need to be corrected. A fault log keeps track of the failures that occurred during testing and the solution that was implemented to correct the underlying fault. Not only will a fault log track changes to the program, but it can be used as a reference when similar failures occur.*
    c) CS1 Activity 1: Calculating expected outputs and fault logging

2) Advanced CS1
    a) Levels of testing
    *Programs should be designed in individual components that can be tested before they are integrated into the software system. Unit tests test individual components, integration tests combine components and test them as a group, and system tests determine whether the system as a whole is functioning as expected.*
    b) Unit testing
    *One of the benefits of dividing a program into components (classes and methods) is that each component can be tested individually and can be tested before the other components are complete. Unit testing is the process of testing each component individually before the completed program is tested.*
    c) Input selection: boundary conditions
    *One method of determining effective test inputs is to determine boundary conditions. Boundary conditions are extreme values or those near defined boundary conditions in the program's specification.*
    d) CS1 Activity 2: Unit testing and input selection

3) Early CS2
    a) Review: Unit testing and input selection
    b) Automated Unit Tests
    *Many frameworks will compare expected outputs to a program to actual outputs and alert the programmer if a failure has occurred. The unit tests can be coded before the component is complete and then run to make sure the component follows the specifications. Automated unit tests can also be run again when a change is made to ensure that the software is still functioning as expected.*
    c) Test-Driven Development
    *During TDD, a programmer first creates unit tests that verify a component based on the specifications. The unit tests initially fail, as the component's functionality has not yet been implemented. The component is*

*then coded and run with the tests until all tests pass. The programmer can then refactor (improve) the component's code and rerun the unit tests to ensure that no bugs have been injected.*

   d) CS2 Activity 1A: TDD

4) Advanced CS2
   a) Review: Automated Unit Tests
   b) Non-functional requirements
      *Functional requirements are used to determine whether a program performs a desired behavior. Non-functional requirements specify criteria used to judge whether the operation was performed in an acceptable manner. For example, a functional requirement may specify that a webpage loads 10 pictures from a photo album. A non-functional requirement may specify that the action must be performed in less than 3 seconds. If the website took 10 minutes to load 10 pictures, it has passed the functional requirement but has not passed the non-functional requirement.*
   c) Regression testing
      *Once components in a system are complete, they may need to be changed to satisfy a new requirement, improve the functionality of the code, etc. Tests will then need to be run again to ensure that no faults have been added to the code during the change.*
   d) CS2 Activity 1B

**Teaching Resources**

Worksheets for all assignments

**Teaching Techniques**

CS1 Activity 1

- Lecture with slides
- Students fill out expected values on a worksheet
- Once worksheet 1 is complete and checked for correctness, students run tests on the program that they receive and log which inputs failed. Students then determine the faults associated with each failure and make necessary changes to the program to correct those faults. The assignment can be completed in the lab or on their own.

CS1 Activity 2

- Lecture with slides
- In-class discussion of selection criteria and perceived advantages of unit testing
  - Brainstorm: Introduce the project design to students.
    - Have the class determine possible boundary conditions for each unit test and write them down on their worksheet.
- Students then begin coding and test their code according to the conditions discussed in class. After students submit their assignment for either manual grading or automated grading, they are scored based on how well their program performs under various conditions.

CS2 Activity 1A

- Lecture with slides - automated unit testing & TDD
- Students create unit tests given inputs and expected outputs
- Once unit tests are complete and checked for correctness, students begin coding assignment. Once all unit tests pass, students must refactor their code and run tests to verify correctness. Assignments are graded based on performance in instructor-designed JUnit tests and code style.

CS2 Activity 1B

- Lecture with slides - Non-functional requirements and regression testing
- Students follow assignment to modify their code from Activity 1A. Existing unit tests are run to verify correctness. Assignments are graded based on performance in instructor-designed JUnit tests and code style.

**Tool Support**

All IDEs listed are completely free of charge.

jGRASP

- User friendly IDE for compiling, running, and debugging code in multiple languages
- Interactions pane where students can experiment with code and test their programs without creating a separate executable program (Java)
- Easy to configure integration with junit, Checkstyle, and Web-CAT
- Includes dynamic visualizations and animations of data structures
- Detailed tutorials in both text and video format

BlueJ

- User friendly IDE for compiling, running, and debugging code in multiple languages
- Focused on object orientation
- Tutorials for students to set up and use features
- Integration with junit
- API for creating custom plug-ins

Eclipse

- Widely-used professional IDE, but may be difficult to use for students not experienced in programming
- Includes integration with junit, Checkstyle, Web-CAT, and much more
- Vast library of custom plug-ins

**Glossary**

Failure –
a situation in which a piece of software's behavior does not match its expected behavior / specifications
Fault –
the cause of a failure in a piece of software. Also called a defect or a bug
Functional requirements –
the desired behaviors of a system given a set of inputs
Input selection –
narrowing down a potentially infinite set of test inputs to a subset that is effective during testing
Non-functional requirements –
requirements for the performance of a system that are independent of specified behaviors
Testability –
a property of a software's design that allows the software to be tested early and easily
Validation –
ensuring that a piece of software matches what was required by the customer
Verification –
ensuring that a piece of software performs as expected in its functional specification

**Bibliography**


ADAMS, J. "Test-driven data structures: revitalizing CS2", Proceedings of the 40th ACM technical symposium on Computer science education - SIGCSE '09, Chattanooga, TN, USA New York, New York, USA, ACM Press, pp. 143, 2009.

JANZEN, D. and SAIEDIAN, H. Test-Driven Learning: Intrinsic Integration of Testing into the CS/SE curriculum. Technical Symposium on Computer Science Education (SIGCSE'06), March, 2006, Houston, TX.

TRIPP, L. (chair), et al. 2004. IEEE Computer Society Professional Practices Committee. 2004. Guide to the software engineering body of knowledge Project (SWEBOK) and the Guide. IEEE Computer Society Press, Los Alamitos, CA, (2004). Available at http://www.swebok.org.

**Suggested Course Activities**

CS1 Activity 1

**Prerequisites**: Students at this level are assumed to have performed basic mathematical calculations and used if statements. The example uses a Java program in a main method, but the activity can be adapted for any programming language. If students are familiar with object orientation, the program can be modified accordingly.

Students must also be given a program to test that calculates BMI that includes several faults, one of which is not identified by the tests. When students find the fault that did not cause a failure during testing, they can answer the last question.

CS1 Activity 2

**Prerequisites**: Students should know how to create classes and methods / functions.

After students complete the worksheet either in class or on their own, they can create their program and run the unit tests. Students can be asked whether any of their unit tests failed or whether any perceived benefits of calculating boundary values before the program was created. Students can also be asked to write a program that will use the methods to convert a dollar amount to change and prints the information to the user (at which point system testing could be carried out).

CS2 Activity 1A & 1B

**Prerequisites**: Students should know how to set up JUnit tests.

**CS 1 Activity 1: Worksheet**

During this activity you will be given a program that you will verify through testing. If any **failures** occur during testing, you will need to find and repair the **faults** (errors) in the program.

Program description: A program has been designed for a gym that will calculate a user's BMI. Users enter their **weight** in pounds and their height in **feet** and **inches**. All numbers must be entered as integer values, or the program will generate a run-time error. If the user enters a value less than or equal to 0 for weight or feet or a value less than 0 for inches, the program will print an error message and end execution. The user's BMI is then calculated according to the following equation.

$$BMI = 704 * \frac{weight}{(12 * feet + inches)^2}$$

The program then displays the user's BMI **rounded to 2 decimals** and weight categorization as follows:

- BMI is 18.4 or below: Underweight
- BMI from 18.5 to 24.9: Optimal weight
- BMI is 30 or above: Overweight

Fill out the expected output of the program in the following test log (use a calculator for calculations). You can write "Error Message" for all invalid input errors. Do not fill out the 'passed' column, the actual output, or the fault description until you receive the program. Input includes the weight and height in feet and inches of the user. For example, someone who is 5'7" 125 lbs would be in the format 125 5 7 on the worksheet.

| Input * | Expected Output | Actual Output | Passed? | Fix Description |
|---|---|---|---|---|
| 0 5 7 | Error Message | | | |
| 125 0 4 | Error Message | | | |
| 169 4 4 | BMI = 44<br>Overweight | | | |
| 140 6 0 | | | | |
| 145 5 -1 | | | | |
| 120 5 8 | | | | |
| 160 5 2 | | | | |

*Input includes the weight and height in feet and inches of the user. For example, someone who is 5'7" 125 lbs would be in the format 125 5 7 on the worksheet.

**After you are finished testing**: Look carefully at the code after you are finished testing. Are there any logic errors that still exist? If so, what would you do in the future to improve the tests that are selected?

**CS 1 Activity 2**

**Part 1:** You will be creating unit tests for a class called DollarsToQuarters with three methods. The constructor to the class should take a dollar amount as a double. The method's functionality is described below. Specify possible boundary conditions for each method:

- maxQuarters: Returns an integer representing the maximum number of quarters in the dollar amount specified in the constructor. Returns -1 if the dollar amount is less than 0.
  Boundary values:

- centsLeft: Returns an integer representing the amount of change left after quarters are taken out. Returns -1 if the dollar amount is less than 0.
  Boundary values:

- toString: Returns a String containing the dollar amount. The dollar amount should always be displayed with 2 decimal places and a dollar sign (example: $2.00 or $2.34). The method should return a String "Invalid dollar amount" if the dollar amount is less than 0.
  Boundary values:

**Part 2**: Based on your boundary values above, write unit test inputs for each method. Below you can specify the dollar input and the output of the specified method under that condition:

maxQuarters

| Input ($) | Expected Output |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |

toString

| Input ($) | Expected Output |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |

centsLeft

| Input ($) | Expected Output |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |

**CS2 Activity 1A**

Create a class called SimpleList that will store descriptions of inventory items. The list must be implemented using an Array that starts with an initial capacity of 5 (size 0) and expands by 3 when necessary. The constructor should take no parameters. The class should include the following methods:

- add(String item): adds the element to the end of the list. Returns true if the element was added (elements with value null should not be added to the list and should result in a false value).
- size(): returns the size of the list as an integer value.
- get(int index): returns the element at the specified index as a String. Returns null if the index is invalid.
- remove(int index): removes the element at the specified index from the list and returns true if the index was valid (in bounds of the list) and false otherwise.

Write the unit tests in JUnit prior to implementing the methods below as specified. After you have written a test set, implement the method and run the tests. Assume that list is a variable referring to an instance of SimpleList (can be set up in the Before method).

| Method(s) | Setup / Input(s) | Expected output | Actual output calculation |
|---|---|---|---|
| **add** | Add the element "Item" to a list. | true | list.add("Item") |
| **add** | Add the element null to a list. | false | list.add(null) |
| **size** | No elements should be added to list | 0 | list.size() |
| **size** | Add 12 valid elements to the list | 12 | list.size() |
| **get** | Add 8 valid elements to the list:<br>"1", "2", "3", "4", "5", "6", "7", "8" | get(-1)<br>get(0)<br>get(7)<br>get(8) | null<br>"1"<br>"8"<br>null |
| **remove** | Add 6 valid elements to the list:<br>"aa", "bb", "cc", "dd", "ee", "ff" | remove(-1)<br>remove(6)<br>remove(5)<br>remove(0) | false<br>false<br>true<br>true |
| **remove** | Add 6 valid elements to the list:<br>"aa", "bb", "cc", "dd", "ee", "ff"<br>Remove elements at indices 0, 2, and 5 | size()<br>get(0)<br>get(1)<br>get(2)<br>get(3) | 3<br>"bb"<br>"dd"<br>"ee"<br>null |

Write 2 more tests sets that you create yourself. You may test the functionality of any of the methods. Describe your tests below.

| Method(s) | Setup / Input(s) | Expected output | Actual output calculation |
|---|---|---|---|
| | | | |
| | | | |

**CS2 Activity 1B**

Storing descriptions of inventory items using your SimpleList class finds that the system is running too slowly, especially when adding a large number of items to the list. First, SimpleList will need to be modified to use a doubly linked implementation rather than an array (your final class should NOT include any array references). In addition, the add method must add directly to the end of the list rather than iterating through all nodes.

Because your functional requirements of SimpleList have not changed, you can use the same JUnit tests that you used in Activity 1A after you have implemented all changes. Your program will be graded based on whether all of your tests pass and whether the non-functional requirements above have been satisfied.

**Questions:**

What is the difference between functional requirements and non-functional requirements? Describe a functional requirement and non-functional requirement presented in the activity.

Suppose that the SimpleList needs an additional method that prints out the entire list to standard out. Is this an example of an additional non-functional or functional requirement? Explain your answer.

Describe the concept of regression testing and how it applied to this project.

Software Construction Curriculum Module

**Preface**

The purpose of these teaching modules is to demonstrate how software engineering knowledge area and principles can be imprinted into teaching computer science at the CS1 and CS2 levels. It is not intended to replace material and topics that are necessary in the curricula. It is hoped that the information presented in this module will enhance the learning experience of the student.

**Module Description**

This module presents an introduction to software construction. Software construction refers to "the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging." The Software Construction module is strongly related to Software Design and Software Testing. This is because the software construction process itself involves significant software design and test activity. It also uses the output of design and provides one of the inputs to testing. The relationship between design, construction, and testing (if any) depends on the software life cycle processes that are used in a project. (SWEBOK)

In SWEBOK, Software Construction is divided into the sub-knowledge area topics shown below. This module will provide assistance for introduction some topics at the CS1 and CS2 levels.

- Software Construction Fundamentals
    - Minimizing ComplexityCS1
    - Anticipating ChangeCS2
    - Construction for Verification
    - Standards in ConstructionCS1
- Managing Construction
    - Construction Models
    - Construction PlanningCS2
    - Construction Measurement
- Practical Considerations
    - Construction Design
    - Construction Languages
    - CodingCS1
    - Construction Testing
    - ReuseCS2
    - Construction Quality
    - Integration

**Philosophy**

Corporate standards, quality assurance procedures, and software methods determine the path that software development takes. Developing skills to construct software in an explainable and standardised format is important. Software construction is one of the major concepts in software development in both industry and education.

Software construction can help students learn by providing them with:

1. Standards and best practices to follow during development
2. A plan of action to follow during development
3. Tools to characterize the software developed
4. Tools to aid in debugging and testing

**Outcomes**

After covering the material in this module, students should be able to:

1. Follow a standard for source code readability
2. Create construction artifacts such that complexity is minimized
3. Plan for changes to software artifacts
4. Create and explain the purpose of reusable code

**Prerequisite Knowledge**

For this module, we assume that there are no prerequisites for CS1 and that CS1 is the only prerequisite for CS2.

**Outline**

1. CS1
   a. Introduction
      i. Software Engineering
      ii. Software Construction
      iii. Reasons for Software Construction
   b. Minimizing Complexity while Coding
   c. Standards in the Classroom
   d. Use on-going standard with peer reviews

2. CS2
   a. Re-cap introduction from CS1
   b. Continue on-going standard from CS1
   c. Code Reuse
   d. Anticipating Changes to Software
   e. Modified Code Activity
   f. Software construction involves team work
   g. Working in a Team Environment Activity

**Annotated Outline**

1. CS1
   a. Introduction
      i. Software Engineering - applies a systematic, disciplined, quantifiable approach (or process) to the development, operation, and maintenance of software.
      ii. Software Construction – "the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging" (SWEBOK)
      iii. Reasons for Software Construction
         1. Provides standards for software development
         2. Provides methods to plan ahead and smooth the software process (construction for verification, anticipating change, integration, testing, etc.)
         3. Provides a series of steps to execute during construction
         4. Increases readability and understanding of software by minimizing complexity
   b. Minimizing Complexity while Coding

i. Limits of human memory - As code becomes increasingly complex, the ability of an individual to grasp it becomes more difficult. There are widely accepted techniques used in order to reduce this complexity.
ii. Readability - One benefit of minimizing complexity is making the source code easier to read. This includes a wide range of topics such as tabbing blocks of code, variable naming conventions, and the proper use of functions.
iii. Code Modules - Grouping related lines of source code increases readability in a file. Putting several lines into a function or grouping lines into input, calculation, and output provides a system for easily reading through source code.

c. Standards in the Classroom

i. Standards provide a common system to perform some activity. These can be process standards, coding standards, testing standards, etc.
ii. External standards - Some standards are written and agreed upon by general public. These standards must be followed if you hope to integrate with real-world systems.
iii. Internal standards - Some standards are written to improve the work-flow process internally and agreed upon typically by a particular work group. These standards are followed to provide a common consensus on when and where to do what.
iv. Example standards - Visiting a website such as a university website will provide students with a better visual example of external and internal standards. The website has to conform to standards such as HTTP while having its own internal user interface standard to make navigation of the website easier.

d. **CS1 Activity** 1. Use on-going standard with peer reviews.

2. CS2

a. Re-cap introduction from CS1
b. Continue on-going standard from CS1
c. Code Reuse

i. Efficiency - Code reuse allows for a developer to save time by simply using source code that's already been verified.
ii. Common Standard - Code reuse also allows for developers to conform to a specified interface, whether it's a function, object, or something else.

d. Anticipating Changes to Software

i. External changes - Sometimes external changes will affect how you develop software. A change to a standard or perhaps customer requirements. If software has been developed to anticipate these changes, then adapting to the change will be easier.
ii. Compartmentalization - By isolating components of a system into smaller categories (functions or objects typically), you can minimize the impact of a change. This allows a developer to separate the parts of the system more likely to change from those that are more stable and create common interfaces between them.

e. **CS2 Activity** 1. Modified Code Activity
f. Software construction involves team work

i. Communication
ii. Documentation

g. **CS2Activity** 2. Working in a Team Environment

**Teaching Resources**

CS1 Activity 1
CS2 Activity 1
Example Peer Review Document

**Teaching Techniques**

Set 1 CS1 activities:
- Lecture to convey ideas
- Weekly additions to on-going standard of skills related to normal course content

Set 1 CS2 activities:
- Lecture to convey ideas
- Simple example of code reuse - prior to CS2 activity 1
- Simple example of anticipating a code change - prior to CS2 activity 1

**Tool Support**

Eclipse
- Cross Platform
- Supports Various programming languages
- Open Source
- UML tool available
- http://www.eclipse.org

Netbeans
- Cross Platforms
- UML tool available
- Software Bundles
    - Web and Java EE
    - Ruby
    - Java ME
    - PHP
    - Java FX
    - Complete
- http://www.netbeans.org

JGrasp
- Cross Platform
- Open Source
- Runs on platform of Java Virtual Machine
- http://www.jgrasp.org

BlueJ
- Cross Platform
- Open Source
- Multilingual
- http://bluej.org

Greenfoot
- Cross Platform
- Open Source
- Combination of Framework/Development Environment
    - Great for Gaming
    - Can be used with Microsoft Kinect Sensor
- http://www.greenfoot.org

DrJava
- Cross Platform
- Open Source
- http://www.drjava.org

Microsoft Visio
- Uses vector graphics to create diagrams

- ▪Available in three editions
- ▪Costs (available free through MSDN Alliance)
- ▪http://office.microsoft.com/en-us/visio/

**Glossary**

IDE – Integrated Development Environment, a software application that provides useful tools to computer programmers during software development usually including a text editor, compiler, debugger, and run-time environment

Artifact – any item created from following a software process; documentation, source code, diagrams, etc.

Standard – a set of guidelines used when creating an artifact such that it follows a well-known pattern

Peer review – the process of having a fellow co-worker or student analyze your work in order to detect flaws or point out odd things within an artifact

Reuse – the use of existing software, or software knowledge, to build new software

Standards – a description of how procedures are to be described, how objects and activities are named, and how processes are to be executed.

**Bibliography**

BAUER, C, et al. The student view on online peer review. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '09, page 26-30, New York, NY, USA, 2009. ACM.

TREMBLAY, G, et al. 2007. Introducing students to professional software construction: a "software construction and maintenance" course and its maintenance corpus. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '07, pages 176-180, New York, NT, USA, 2007. ACM.

TRIPP, L. (chair), et al. 2004. IEEE Computer Society Professional Practices Committee. 2004. Guide to the software engineering body of knowledge Project (SWEBOK) and the Guide. IEEE Computer Society Press, Los Alamitos, CA, (2004). Available at http://www.swebok.org.

**Suggested Course Activities**

CS1 Activity 1 - On-going Standard with Peer Reviews

*Description.* In software engineering, there are plenty of typical readability standards in place keep code uniform and understandable. One of the difficulties with standards is that they can be a lot of information to take in at once. Instead of hitting students with one large standard, this approach provides an evolving standard that can go concurrently with normal coursework and uses minimal amount of class time.

After an assignment is turned in, the students can then perform peer reviews (BAUER, 2009) on another student's work while basing their comments on this standard. These reviews can be performed in person or anonymously over some other medium and they can be performed with any language or code-based homework assignment. A simple peer review sample is located at the end of this document.

*Concepts Introduced.*

1. Construction Standards
2. Coding Concepts
3. Minimizing Complexity
4. Peer Review

*Example Standard Items.* This section is intended to provide some example standard items that can be easily introduced in an ever-growing document that reflects the current content in the course. The format of this document provided to the students is up to the instructor. Presentation of the document can be done briefly during class time as a ``do this, not that" portion of class. The current state of this document should be available to students as a reference point for applying standards to assignments. The following list contains examples of some standards that can be applied in a CS1 course. Others can be added as the students acquire new skills.

1. Constant Naming Conventions - for constant values use all capitalized letters, `_', and/or numerical values. Examples include DEGREES_TO_RADIANS, HOURS_IN_DAY, FEET_TO_METERS.
2. Variable Naming Convention - start all variables with a lowercase letter, use camelbacking or `_' to separate words in a variable name. Examples include myName, my_name, tempOutside.
3. Comments - provide comments for every section of code, especially functions, objects, and the beginning of programs
4. Separate Sections - whenever possible, separate similar lines of code into sections such as input, calculations, and output.
5. Tabbing Conventions - offset blocks of code by an indenting, this applies to loops and conditionals in particular. For nested loops or conditionals, indent multiple times to make the document more readable.

*Sample Peer Review Document.* Instructions: Review each of the following items from the class standard. Rank your peer's compliance with the standard each of those items on a scale from 1-5 where 1 is non-compliant and 5 is completely compliant. Also comment on places where your peer is non-compliant with line numbers and examples.

1. Constant Naming Conventions:

2. Variable Naming Conventions:

3. Comments Throughout:

4. Separation of Sections of Code:

5. Tabbing Conventions:

CS2 Activity 1 - Modifying Code Activity.

*Description.* This activity is based on two major ideas: reusing code and anticipating changes that may need to be made to a program. To accomplish this task, there will need to be at least two assignments, one where some task is accomplished and then a second one where that task has to be reused or modified because of some new requirement (Tremblay, 2007). Alternatively, you could try to make a three assignment group with the original assignment, a reuse assignment, and a change assignment.

*Concept Introduced.*

1. Code Reuse
2. Anticipating Changes

*Example of Code Reuse.* This is based on the assumption that the CS2 course is primarily focused on data structures. The assignment itself is relatively simply and based on the concept of queues and priority queues. This could theoretically be done over the course one or two weeks and concurrently with the queue lectures. Consider the following for a first assignment given to the students to demonstrate code reuse:

**Assignment 1:** A business is hoping to provide better automation to its customers by implementing a new line system that will help allow customers to check in and have a seat instead of physically waiting in line. Create a wrapper class for a queue structure called StoreLine that will have the following functions:

1. void addCustomer(string name) - puts a customer at the end of the line
2. string lookAtNext() - returns the name of the next customer in line
3. string getNextInLine() - returns the name of the next customer in line and removes that customer from the line

Include in the program a simple interface to add a customer, look at who is next in line, and get the next person in line.

**Assignment 2:** Another business, after seeing what you did in assignment 1, has contacted you about making a custom line system where certain customers are paying for priority service. They have three levels of priority: 1, 2, and 3, where 1 is the highest priority customer. Using your unmodified StoreLine class, create another class called PriorityLine that includes the following functions:

1. void addCustomerToLine(string name, int lineNumber) - puts a customer at the end of the line with the given priority
2. string lookAtNext() - returns the customer of highest priority who is currently at the front of his/her line
3. string getNextInLine() - returns the customer of highest priority who is currently at the front of his/her line and removes that customer from the line

CS2 Activity 2 – Working in a Team Environment

Working in a team environment would more closely resemble real software development. Working on design documents and with a project management plan that the student did not developed would give the student an accurate appreciation of the real world.

In this activity, there is a single project with two sets of requirements. The challenge is for the students to first describe the changes to be done and then to execute them according to the plan. The students are given a completed project for which the requirements have changed. They are instructed to determine the changes needed to meet the new requirements. To emphasize the importance of communicating with future developers of the same software project, it is recommended that process of modifying the problem solution be done by multiple teams of students.

Each team would accomplish a phase in the software development process. For instance, team A produces the design document; team B enhances the document and produces the project plan; team C completes the quality assurance and test documents; team D programs; and team E tests.

Software Design Curriculum Module

**Preface**

The purpose of these teaching modules is to demonstrate how software engineering knowledge area and principles can be imprinted into teaching computer science at the CS1 and CS2 levels. It is not intended to replace material and topics that are necessary in the curricula. It is hoped that this information presented in this module will enhance the learning experience of the student.

**Module Description**

This module presents an introduction into software design. Software design refers to both (1) the process of defining the architecture, components, interfaces, and other characteristics of a system or component and (2) the result of [that] process. Software design must describe the software architecture – that is, how software is decomposed and organized into components – and the interfaces between those components. It must also describe the components at a level of detail that enable their construction (SWEBOK).

Software design is invaluable to the software engineering process as it allows to software engineer to describe in detail how components will function and interact with one another before any actual code is written. The artifacts produced by this process will provide a blueprint which will allow the software engineer to create a high quality end product.

The SWEBOK divides the software design process into the sub-knowledge area topics shown below. This module will provide assistance in introducing some of these topics into typical CS1 and CS2 level courses.

| Topic | Level |
|---|---|
| Software Design Fundamentals | |
| General Design Concepts | CS1, CS2 |
| The Context of Software Design | CS1, CS2 |
| The Software Design Process | |
| Enabling Techniques | |
| Key Issues in Software Design | |
| Concurrency | |
| Control and Handling of Events | CS1 |
| Distribution of Components | |
| Error and Exception Handling and Fault Tolerance | CS1, CS2 |
| Interaction and Presentation | CS1, CS2 |
| Data Persistence | CS1 |
| Software Structure and Architecture | |
| Architectural Structure and Viewpoints | |
| Architectural Styles (Macroarchitectural Patterns) | |
| Design Patterns (Microarchitectural Patterns) | CS2 |
| Families of Programs and Frameworks | CS2 |
| Software Design Quality Analysis and Evaluation | |
| Quality Attributes | |
| Quality Analysis and Evaluation Techniques | |
| Measures | |
| Software Design Notation | |
| Structural Descriptions | CS1 |
| Behavior Descriptions (Dynamic View) | CS1, CS2 |
| Software Design Strategies and Methods | |
| General Strategies | CS1 |
| Function-Oriented (Structured) Design | CS1 |

| | |
|---|---|
| Object-Oriented Design | CS1, CS2 |
| Data-Structure Centered Design | |
| Component-Based Design | |
| Other Methods | |

**Philosophy**

Software design is a necessary part of the software engineering process. By gaining a firm grasp on basic software design principles, the software engineering student will gain a better understanding of the overall software engineering process in addition to being able to produce software artifacts of a higher overall quality. The earlier that good software design principles are incorporated into a pupil's software engineering education the sooner they can begin to gain valuable tools and techniques that will benefit them for the rest of their professional careers. These include but are not limited to:

1. A better understanding of how the components of a software system interact with each other
2. The ability to finitely model abstract systems
3. A communication platform for discussing design alternatives and trade-offs
4. The ability to determine what a software system must accomplish
5. The knowledge and use of practical design patterns
6. The ability to design higher quality code
7. The understanding of code reuse and how to incorporate it into their software systems
8. Basic tools to help determine the quality of a software artifact
9. Common issues that reoccur in software development

**Outcomes**

Through covering the material included in this module, a CS1 or CS2 student should be able to:

1. Break a given problem statement into a set of components which can accomplish the given goal
2. Determine the interfaces between these software components
3. Produce a physical artifact which displays these components and their interface
4. Be able to effectively communicate the design of a given software system
5. Analysis the strengths and weaknesses of a given design and be able to discuss these attributes with peers
6. Begin to recognize recurrent issues in software design and solutions that apply

**Prerequisite Knowledge**

While software design principles can begin to be incorporated into the earliest stages of a software engineering education this module assumes that the CS1 student has a very basic understanding of an object oriented programming language. In addition, it is assumed that the CS2 student has taken a CS1 course in which good programming practices were enforced and where the basics of software design were covered at a very high level.

**Outline**

1) CS1
    a. Introduction
        i. What is software engineering?
        ii. What is software design?
        iii. Why is design important?
        iv. Can we determine the quality of a design?
    b. Introduction to Simple Design Tools

i. Pen and paper
ii. Mind map
c. Demonstration of Simple Design Artifacts (Structural)
d. Application

2) CS2
a. Recap of CS1 Knowledge
b. Behavioral Modeling
i. What is it?
ii. How does this differ from structural modeling?
iii. Why is this important?
c. Introduction to Design Patterns
i. What are design patterns?
ii. What value do they have?
iii. Simple design patterns
- Nested if statements
- Recursion
- Factory
d. Design Quality Analysis Introduction
i. Verification
ii. Validation
iii. Analysis of tradeoffs
e. Application

**Annotated Outline**

1) CS1
*This outline is designed to be used for teaching a lesson incorporating elements of software design in the last half of a typical CS1 course. Before approaching this topic the student should have a good grasp on programming paradigm being used in the course.*

a. Introduction
i. What is software engineering?
- Software engineering applies a systematic, disciplined, quantifiable approach (or process) to the development, operation, and maintenance of software.

ii. What is software design?
- Software design is the both (1) the process of taking a problem and then decomposing it into components and (2) the architecture consisting of interfaces and components as well as the artifacts produced by this process.
- In determining the place of software design in the software engineering lifecycle, software design should take place before any code is written but should not stop once coding begins.
- Software design differs from both requirements analysis and software construction.
- Types of design (from SWEBOK):
  - Software architectural design – top-level structure and organization and identifying the various components.
  - Software detailed design – describing each component sufficiently to allow for its construction.

iii. Why is design important?
- Software engineering is not the only field in which design is used. Software engineering can easily be compared to the design and building of a house or the writing of an English research paper. If more detail/explanation is needed

consider the case of a model home that will be used to create an entire neighborhood of slightly different homes.

- Design allows you to:
  - practically apply the principles of abstraction by building more and more complicated systems based on simpler ones
  - decompose a system into its various components and ascribe functionality to each
  - identify areas in which the practice of code reuse can be applied
  - ensure that a system is complete and will provide the functions that have been identified in requirements analysis
  - effectively communicate a design with coworkers or peers

iv. Can we determine the quality of a design?

- Discuss “ilities” and “nesses” at a high level
  - Maintainability – how easy it to perform maintenance on the given software system once it has been implemented?
  - Portability – how easily can the system be moved from one platform to another? How easily can the components of the system be reused in a completely separate system?
  - Testability – how easy will it be to test the software system once it is implemented?
  - Correctness – is the design correct? Does it meet all of the requirements that were defined in the requirements analysis phase of the engineering process?
  - Robustness – how robust is the code where errors and failures are concerned?
- Design reviews are a valuable tool that can be used to determine the quality of your design. Design reviews should occur though out the entire design process and should be done by someone qualified to complete the review. Findings from the review can then be integrated into the rest of the design process.

b. Introduction to Simple Design Tools

i. Pen and paper

- Extremely simple and watered down version of software design
- Good for mapping out ideas of how a software system will function before beginning to write any code
- At the CS1 level, this may be all the design that is really needed to complete many of the problems students will be given.
- Good practice to get into, even on simply problems. It is a lot easier to change a design on paper than it is once it has been coded.

ii. Free Mind

- Free software product which allows the students to produce designs at a very high level.
- Good for group presentations and for discussing topics in lab.
- More information can be found in the tool support section of this module.

c. Demonstration of Simple Design Artifacts (Structural)

At this point, the students should understand the basic principles of software design. Using a lab assignment that they have completed previously in the semester, start with the project requirements and demonstrate how the design process would have worked for this project. At this time, you can also introduce any standard notation or diagrams that you would like the students to use throughout the rest of the course. Focus on the structural aspects of design as these will be easiest for students to understand at this level. Basic behavioral design notations may be added as well.

d. Application

To reinforce what has just been taught in lecture, present the students with a normal lab assignment but incorporate aspects of software design into the assignment as well. See the CS1 suggested activity included at the end of this module for an example of how this can be accomplished.

2) CS2

*This module is designed to be used during the first half of a typical CS2 course. For students to fully understand the topics covered in this module a basic understanding of software design is required. In it is assumed that a student has taken a prior CS1 course and that the topic of recursion has already been covered in detail.*

a. Recap of CS1 Knowledge

See outline above. Refresh the key topics such as what software design actually is as well why software design is necessary and how you can judge the quality of software design. This can most easily be accomplished using a classroom discussion if the class size allow for it.

b. Behavioral Modeling

i. What is behavioral modeling?

- Focuses on the interaction of software components with one another as well as the outside world
- It presents a dynamic view of software – one that allow change over time
- Some common artifacts are data flow diagrams, decision tables, flow charts, and sequence diagrams.

ii. How does this differ from structural modeling?

- Structural modeling is static where as behavioral modeling is dynamic
- Structural modeling is concerned with the overall structure of a software system where as behavioral modeling is concerned with how the components defined in the structural model will interact with one another.
- Different artifacts are produced and compared to structural modeling due to the different nature of the information being modeled.

iii. Why is this important?

All software systems have both a structure and a behavior. If the software did not behave in some way, it would not be able to accomplish any work for the user and would be useless. Being able to understand and design the behavior of a piece of software is a critical component in the software engineering process.

c. Introduction to Design Patterns

i. What are design patterns?

A design pattern is a reusable strategy to meet the needs of some common problem that arises repeatedly in software engineering. It is not code or even pseudocode. Instead, it is a template which can be applied in any situation to solve a given problem.

ii. What value do they have?

- By studying design patterns and having a working knowledge of them many problems encountered when designing software can be dealt with easily and efficiently.
- Design patterns make it easier to communicate the idea of what you are trying to accomplish a given software design
- It is easier to understand and analyze the trade offs of various design patterns as opposed to a "roll you own" approach.

iii. Simple design patterns

All of the following design patterns would be best explained by using a "toy" problem that the class is familiar with. An excellent source of these would be snippets from previous assignments that they have already completed in the current

CS2 course. This allows the code to be trivial to the student and allows them to focus on the main point that you are trying to convey, what a design pattern is and how to apply it.

- Nested if statements

  The main purpose of introducing this design pattern is to show students that they have already been using various design patterns even if they did not realize it. Discuss the benefits and disadvantages of this pattern as well as comparing it to other solutions to the same problem.

- Recursion

  If the topic of recursion has yet to be covered in the CS2 curriculum, this design pattern should be skipped or replaced with another pattern so that the details of recursion do not take away from the discussion of the pattern itself. If recursion has been covered, focus on this pattern as you did the nested if pattern above. Show the students that they have already used design patterns even though they were unaware of it. Discuss the cost and pay offs of using recursion as opposed to a loop or a table. The classical example of this would be a factorial function.

- Factory

  This design pattern was chosen for its simplicity as well as the value it adds to a software system. The factory design pattern is an object-oriented design pattern which concerns itself with the creation of new objects. The factory design pattern uses a concrete creator class which inherits from a creator parent class. This concrete creator can then be used to create an object. More information about the factory design pattern as well as other design patterns can be found in the reference section of this module.

d. Design Quality Analysis Introduction

i. Verification

- The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [IEEE-STD-610]
- Checking to see if the software actually works

ii. Validation

- The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements [IEEE-STD-610]
- Checking to see if the software works correctly, that it meets the original project requirements

iii. Analysis of tradeoffs

- Considering different solutions to a given problem
- What makes one solution better than another
- Things to consider
  - a. Time efficiency
  - b. Space efficiency
  - c. Code reuse
  - d. Modularity
  - e. Etc.

e. Application

To reinforce what has just been taught in lecture, present the students with a normal lab assignment but incorporate aspects of software design into the assignment as well. See the CS2 suggested activity included at the end of this module for an example of how this can be accomplished.

**Teaching Resources**

Blank CS1 Project Assignment Worksheet
Blank CS2 Project Assignment Worksheet

**Teaching Techniques**

CS1

- Introductory lecture using the annotated outline included above
- In-class discussion of basic design concepts to gauge students understanding as well as any areas that might need to be covered again
- Have students break into groups of two. Assign a simple problem and have students individually sketch a structural design based on the notation covered in lecture. Have students exchange sketches and critique each other's work
- Review a solution to the design problem with the class
- Assign a programming assignment which also focuses on design in the form of the assignment included at the end of this module.

CS2

- Introductory lecture using the annotated outline included above
- In-class discussion of basic design concepts to gauge students understanding as well as any areas that might need to be covered again
- Assign a programming assignment which also focuses on design in the form of the assignment included at the end of this module.
- Once the project has been submitted have individual or groups of students present the design portion of their project to the class. They should include a description of the design, an analysis of one or more of the tradeoffs they looked at, a description of the design pattern they researched, and any issues they ran into during implementation of their design.
- Allow other students to ask questions of the group about their design to give them practice on communicating about design principles.

**Tool Support**

Pen and Paper

While it may seem that this option need not even be mentioned, simply having students quickly sketch out their design on paper before implementing it can add great value to a classroom assignment. In addition, these sketches can be easily turned in to an instructor for grading. There is no software that the novice software engineering student must learn; and there is no additional cost to either the department or the student. This also enforces good software engineering habits by showing the student that design can be performed for even the simplest project by using the tools on hand. By asking students to perform some type of design on every project in a very nonintrusive way, students will begin to form the good habit of thinking from a design perspective.

Free Mind

This is a free software product that allows students to create mind maps of a programming problem during the software design phase. The interface and options provided in this software package are very intuitive and can be picked up quickly. In addition, the options provided are limited so students will not be overwhelmed. While this software product is simple, it provides all the tools needed to produce effective design documentation at the CS1 or CS2 level. More information about Free Mind as well as downloading the software can be found at the following url: http://freemind.sourceforge.net/wiki/ index.php/Main_Page

jGRASP

jGRASP is an integrated development environment with visualizations for improving software comprehensibility. While this IDE features many nice visualizations that can be used to enhance the educational experience in CS1 and CS2 courses the one that applies most directly to software design is the automated UML class diagram production which the IDE can accomplish. By demoing instructional code in jGRASP instructors can easily show students an UML representation of the software being discussed. jGRASP, developed by Auburn University, is a free product and more information as well as the download can be found at the following url:
http://www.jgrasp.org/

**Glossary**

Software Design – both the process of defining the architecture, components, interfaces, and other characteristics of a system or component and the result of [that] process

Software Architectural Design – design of top-level structure and organization and identifying the various components of a software system

Software Detailed Design – describing each component of a software system sufficiently to allow for its construction

Structural Modeling – static model of a software system concerned with the overall structure of the system itself

Behavioral Modeling – dynamic model of a software system concerned with how components of a software system interact with one another and the outside world

Maintainability – the measure of how easy it to perform maintenance on the given software system once it has been implemented

Portability – the measure of how easily a software system be moved from one platform to another or how easily the components of the system be reused in a completely separate system

Testability – the measure of how easy will it be to test the software system during implementation

Correctness – the measure of how well a software system's design meets the requirements defined during the requirements analysis phase of the software engineering life cycle

Robustness – the measure of how well a software system can stand up to faults and failures

Design Pattern – a reusable strategy that meets the needs of some common problem that arises repeatedly in software engineering

Verification – the process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase

Validation – the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements

**Bibliography**

ALLISON, C and HARRISON, N. 2007. Teaching Design Patterns: A Matter of Principle. *Journal of Computing Sciences in Colleges.* 23, 1 (October 2007), 206-211.

GAMMA, E., HELM, R., JOHNSON, R., and VLISSIDES, J. 1995. *Design Patterns Elements of Reusable Object-Oriented Software*, Indianapolis, Indiana: Addison-Wesley.

GESTWICKI, P. and SUN, F. 2008. Teaching Design Patterns Through Computer Game Development. *Journal on Educational Resources in Computing* (JERIC). 8, 1 (March 2008 ), 1-22. [doi>10.1145/1348713.1348715]

HORSTMANN, C. 2010. *Big Java 4th Edition*, Hoboken, New Jersey: John Wiley and Sons Inc.

IEEE Std 610.12-1990 (R2002), IEEE Standard Glossary of Software Engineering Terminology, IEEE.

LEWIS, J. and LOFTUS, W. 2009. *Java Software Solutions Foundations of Program Design 6th Edition*. New York, New York: Pearson Education Inc.

LEWIS, T., ROSSON, M., and PÉREZ-QUIÑONES, M. 2004. What do the experts say?: teaching introductory design from an expert's perspective. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. Norfolk, Virginia, USA, 296-300. [doi>10.1145/971300.971405]

TRIPP, L. (chair), et al. 2004. IEEE Computer Society Professional Practices Committee. *Guide to the Software Engineering Body of Knowledge Project* (SWEBOK). IEEE Computer Society Press, Los Alamitos, CA. Available at http://www.swebok.org.

WICK, M. 2005. Teaching design patterns in CS1: a closed laboratory sequence based on the game of life. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*. St. Louis, Missouri, USA, 487-491. [doi>10.1145/1047344.1047499]

**Suggested Course Activities**

**CS1 Activity**

NAME: ________________________________

DATE: ________________________________

SECTION NUMBER:_____________________

*The first three fields of this document set up the assignment for the student. The amount of documentation presented in these fields should be determined by the compatancey level of the students in the class. For simplicity sake a simple test explanation is included here. In an actually assignment these fields could include pseudo code, uml diagrams, pictorial descriptions, sample input and output, method headers, or even source code provided to get the project started. The project provided should be appropriate during the last half of a typical CS1 course.*

**PROJECT DESCRIPTION:**
Create a program which can store information on various staff members for an company and compute a payroll report from this information. Staff members can either be volunteer workers or employees with employees being either hourly or salary workers. Each staff member should have an associated name, social security number, and phone number. In addition salary employees should have a bi-weekly salary while hourly employees have an hourly salary as well as the number of hours worked for the current pay period.

**USER INPUT:**
All input and output should be done from the command line. A user should be able to input a new employee or update any of the stored information for a current employee. A simple text menu should be presented to the user with the following options: input new employee, update current employee, display payroll information, exit program. A detailed explanation of what input is expected should be presented for each option that is selected.

**REQUIRED OUTPUT:**
The only output from this program should be a table of payroll information. This information should be displayed in the following order staff name, social security number, phone number, current pay period salary. On exiting the program all staff information will be lost.

**IN LAB SECTION:**

**SOFTWARE DESIGN:**
In the space provided below draw a simple diagram using the notation discussed in lecture describing the design of your software system.

*This section can be completed using the Free Mind Software described in the tool section above.*

**SOFTWARE DESIGN EVALUATION:**

Swap papers with a neighbor and evaluate each other's design based on the characteristics listed below:

Correctness:

Maintainability:

Testability:

Robustness:

EVALUATOR'S NAME:__________________________________

*The level of detail provided in the comments above should be at a high level but this will force students to start thinking with a design mind set and allow them to see other's designs as well as practice critical thinking skills in evaluating these designs.*

**SOFTWARE DESIGN EVALUATION – DESIGNER COMMENTS**

In the space provided below describe what you think about your evaluator's comments above. We the points made valid? Will your design change in any way due to these comments? Where there any issues you completely overlooked?

**AT HOME SECTION:**

**FINAL SOFTWARE DESIGN:**
Incorporating any changes mention above draw a new design diagram for this assignment in the space provided below (if your design did not change explain why):

**IMPLEMENTATION:**
Implement your design in code. When complete answer the following question: Did your design change any during implementation? If so why?

**TURN IN:**
Turn in all files required to compile and run your program as well as a readme file describing how to compile and run you program. In addition turn in this worksheet in the lab session immediately following the turn in date.

*A blank copy of this worksheet follows.*

**COURSE # - ASSIGNMENT #**
**Due Date**

NAME: ________________________________

DATE: ________________________________

SECTION NUMBER:_____________________

**PROJECT DESCRIPTION:**

**USER INPUT:**

**REQUIRED OUTPUT:**

**IN LAB SECTION:**

**SOFTWARE DESIGN:**
In the space provided below draw a simple diagram using the notation discussed in lecture describing the design of your software system.

**SOFTWARE DESIGN EVALUATION:**

Swap papers with a neighbor and evaluate each other's design based on the characteristics listed below:

Correctness:

Maintainability:

Testability:

Robustness:

EVALUATOR'S NAME:___________________________________

**SOFTWARE DESIGN EVALUATION – DESIGNER COMMENTS**

In the space provided below describe what you think about your evaluator's comments above. We the points made valid? Will your design change in any way due to these comments? Where there any issues you completely overlooked?

**AT HOME SECTION:**

**FINAL SOFTWARE DESIGN:**

Incorporating any changes mention above draw a new design diagram for this assignment in the space provided below (if your design did not change explain why):

**IMPLEMENTATION:**

Implement your design in code. When complete answer the following question: Did your design change any during implementation? If so why?

**TURN IN:**

Turn in all files required to compile and run your program as well as a readme file describing how to compile and run you program. In addition turn in this worksheet in the lab session immediately following the turn in date.

**CS2 Activity**

NAME: ________________________________

DATE: ________________________________

SECTION NUMBER: _____________________

*This assignment is set up much like the CS1 assignment to present a consistent format to the students and allow them to understand what is being asked of them quickly. The project mentioned below should be appropriate for a mid-semester CS2 assignment. Again any level of detail required can be provided in the sections below but for simplicity sake in this document only a simple text description is provided.*

**PROJECT DESCRIPTION:**

Implement a program which is able to solve a maze which is input as a text file. The program should find a solution for the maze. How the program achieves this goal is up to you and the program only has to find one solution to the maze, not necessarily the best solution. If no solution to the maze can be found the program should output a message conveying this to the user.

**PROGRAM INPUT:**

This program should receive input from the user in the form of a text file named input.txt. The first line of the program should be the size of the maze to be solved in the format width x height. For example if a maze of width 5 and height 7 is to be solved the first line should be 5 x 7. The next lines of the text file should contain the maze itself. Only the following symbols are allowed to be in these lines:

S – the start of the maze

F – the finish of the maze

* – a wall (impassable) section of the maze

= – a possible path through the maze

The program should not be allowed the exit the area of the maze input in the first line and the edges of the maze should be considered as walls. For example the following would be a valid input.txt file:

```
5 x 7
S * - - -
- * - - -
_ * * * _
- * F - -
_ * * _ _
- - - - -
* * * _ _
```

**REQUIRED OUTPUT:**

If there is a path through the maze the program should output the solved maze with the path marked by the character $. If there is no path through the maze the program should output a message convey this information to the user. For example one valid solution to the sample input file would be:

```
S * - - -
$ * - - -
$ * * * _
$ * F $ -
$ * * $ -
$ $ $ $ -
* * * _ _
```

**SOFTWARE DESIGN:**

In the space provided below draw out in detail you design strategy for this programming assignment using the notation described in class (be sure to include cover both structural design and behavioral design):

**SOFTWARE DESIGN DISCUSSION:**

Answer the following questions:

1) Did you use any of the design patterns mentioned in lecture in your design? If so, which one(s) and how were they applied? If not, do you think your design would benefit from any of these design patterns?

2) How can you validate your design before you begin implementing you code? Describe how you accomplished this for this assignment.

3) How can you verify your design before you begin implementing you code? Describe how you accomplished this for this assignment.

4) Analyze at least two different tradeoffs that you considered during designing your software system. Why did you choose the option that you did? What were the other alternatives?

**IMPLEMENTATION:**
Implement your design in code. When complete answer the following question: Did your design change any during implementation? If so why?

**TURN IN:**
Turn in all files required to compile and run your program as well as a readme file describing how to compile and run you program. In addition turn in this worksheet in the lab session immediately following the turn in date.

**POST PROJECT PRESENTATION:**
Prepare a 15 minute presentation on you project. In this presentation include the following: your initial design, a discussion of any design patterns used, a brief description of how you verified and validated your software system, any problems you encountered when implementing your design, a final evaluation of the quality of your design. Be prepared to hold a 5 minute Q&A session with the class after your presentation.

*If time does not permit for each student to give a presentation this can be done in teams or as a written report turned in with the project.*

*A blank copy of this worksheet follows.*

**COURSE # - ASSIGNMENT #**

**Due Date**

NAME: ______________________________

DATE: ______________________________

SECTION NUMBER: ___________________

**PROJECT DESCRIPTION:**

**PROGRAM INPUT:**

**REQUIRED OUTPUT:**

**SOFTWARE DESIGN:**

In the space provided below draw out in detail you design strategy for this programming assignment using the notation described in class (be sure to include cover both structural design and behavioral design):

**SOFTWARE DESIGN DISCUSSION:**

Answer the following questions:

1) Did you use any of the design patterns mentioned in lecture in your design? If so, which one(s) and how were they applied? If not, do you think your design would benefit from any of these design patterns?

2) How can you validate your design before you begin implementing you code? Describe how you accomplished this for this assignment.

3) How can you verify your design before you begin implementing you code? Describe how you accomplished this for this assignment.

4) Analyze at least two different tradeoffs that you considered during designing your software system. Why did you choose the option that you did? What were the other alternatives?

**IMPLEMENTATION:**

Implement your design in code. When complete answer the following question: Did your design change any during implementation? If so why?

**TURN IN:**

Turn in all files required to compile and run your program as well as a readme file describing how to compile and run you program. In addition turn in this worksheet in the lab session immediately following the turn in date.

**POST PROJECT PRESENTATION:**

Prepare a 15 minute presentation on you project. In this presentation include the following: your initial design, a discussion of any design patterns used, a brief description of how you verified and validated your software system, any problems you encountered when implementing your design, a final evaluation of the quality of your design. Be prepared to hold a 5 minute Q&A session with the class after your presentation.

Software Quality Curriculum Module

**Preface**

The purpose of these teaching modules is to demonstrate how software engineering knowledge areas and principles can be imprinted into teaching computer science at the CS1 and CS2 levels. It is not intended to replace material and topics that are necessary in the curricula. It is hoped that the information presented in this module will enhance the learning experience of the students.

**Module Description**

This module provides an introduction to software quality early in the computer science curriculum. Software quality is interleaved throughout many of the knowledge areas and is applicable throughout the software engineering life cycle, therefore, it makes it easily to introduce students to quality techniques in CS1 and CS2. The techniques discussed in the software quality knowledge area is more static than dynamic, which means that it does not require the execution of the software being evaluated (SWEBOK).

In the SWEBOK, Software quality is divided into three subareas and several topics. This module will demonstrate how these topics can be incorporating into a CS1 and/or CS2 curriculum.

- Software Quality Fundamentals
  - Software Engineering Culture and Ethics CS1, CS2
  - Value and Cost of Quality
  - Models and Quality Characteristics CS1, CS2
  - Quality Improvement CS1, CS2
- Software Quality Management Processes
  - Software Quality Assurance CS2
  - Verification and Validation CS2
  - Review and Audits CS2
- Practical Considerations
  - Application Quality Requirements
  - Defect Characterization CS2
  - Software Quality Management Techniques
  - Software Quality Measurement

**Philosophy**

Software Quality is define as "the degree to which a set of inherent characteristics fulfills requirements," according to ISO9001-00 (SWEBOK). Therefore, the use of software quality techniques throughout the software development life cycle is crucial to the success of a software project. It is important for software quality to be integrated into the CS1 and CS2 curriculum to assist in learning and teaching by providing the following:

- A basic understanding of software quality early in a student's learning computer science development
- A method for assuring requirement conformance
- The role of ethics in achieving software quality
- An attitude towards value from a customer perspective
- A set of quality standards and guidelines for software quality
- Exposure to good programming practices

**Outcomes**

Though the material covered in this module, students should:

- Define software quality
- Understand the difference between Software Quality and Software Testing
- Possess an positive attitude related to quality
- Identify not only functional requirements, but quality requirements as well
- Explain the verification as well as validation processes
- Firm understanding of the process of error/defect detection, removal, and prevention

**Prerequisite Knowledge**

The CS1 level of subject matter presented in this module requires no computer science prerequisite. Student must be familiar with basic word processing software. CS1 is the prerequisite for CS2.

**Outline**

1. **CS 1**
   a. Introduction
   b. Software Development Life Cycle
   c. Software Quality (Part 1)
      i. Quality Defined
      ii. Benefits of Software Quality
      iii. Software Quality Vocabulary
      iv. Error and defect detection, removal and prevention
   d. A Quality approach to software development
   e. **CS 1 Activity 1** Use the worksheet provided.
2. **CS 2** – A more in-depth over of software quality
   a. Recap of CS1 topics
   b. Software Quality (Part 2)
      i. Software Quality Vocabulary
      ii. Notions of Software Quality
   c. **CS 2 Activity 1**: Identify quality requirements for a project using the software quality notions above.
      i. Quality Standards and Models
   a. **CS 2 Activity 2**: Conduct a walkthrough of software development product project with the following objectives in mind:
      i. Find anomalies
      ii. Improve the software product
      iii. Consider alternative implementations
      iv. Evaluate conformance to standards and specifications

**Annotated Outline**

1) **CS 1**
   a) Software engineering applies a systematic, disciplined, quantifiable approach (or process) to the development, operation, and maintenance of software.
   b) Software Quality
      i) Measure of how the characteristic of products fulfill the requirements
      ii) Relevant to all phases of the software development life cycle, i.e., from identifying the problem and requirements through developing a solution to testing and acceptance
      iii) Software functional quality reflects how well it conforms to a given design based on functional requirements or specifications.

iv)Software structural quality refers to how it meets non-functional requirements .
c)Error and defect detection, removal and prevention
d)Error, Defect, and Bug are often used interchangeably by developers, but have very different meanings. See below:
i)**Error**: A mistake in the system under test; usually but not always a coding mistake on the part of the developer.
ii)**Defect**: Nonconformance to requirements or functional / program specification
iii)Defect Characterizations
(1)Error: "A difference...between a computed result and the correct result"
(2)Fault: "An incorrect step, process, or data definition in a computer program"
(3)Failure: "The [incorrect] result of a fault"
(4)Mistake: "A human action that produces an incorrect result"
iv)**Bug**: A fault in a program which causes the program to perform in an unintended or unanticipated manner.
e)**CS1 Activities 1 and 2**.

**2)CS 2**
a)Recap of CS1 topics
b)Software Development Products
i)Abstract and difficult to define how to measure quality
**ii)CS2 Activity 1.**
c)Non-functional requirements of Software Quality
i)Correctness
ii)Reliability
iii)Robustness
iv)Maintainability
v)Adaptability
vi)Testability
vii)Reusability
viii)Performance
ix)**CS2 Activity 5**.
d)Verification and Validation (V&V)
i)Determines whether or not the development products resulting conform to the requirements, and whether or not the final software product fulfills the intended purpose and meets the user requirements
(1)Verification: Is the product is built correctly?
(2)Validation: Is the right product built and does it meet the intended purpose?
ii)**CS2 Activity1a**.
e)Walkthrough
i)An informal review of a software product.
**ii)CS2 Activity 4.**

**Teaching Resources**

Requirements Worksheet
Coding styles

**Teaching Techniques**

- Lecture with slides.
- Use document camera to mark errors found by students and discussion points.
- Provide a section of code with defects and lead the students in finding and typing the defects. This can be code created by the instructor or anonymous code from a previous assignment.
- Activities maybe completed

- In class by individual students or small groups with full class discussion of findings
    - As homework
- Provide blank worksheet for students to complete individually or as a group.
- Pair students for walkthrough

**Tool Support**

Below is a list of tools that can be used within a CS1 and or CS2 to demonstrate software techniques.

1)Automated Testing Tools

a)Eclipse Test and Performance Tools Platform (TPTP)

(1)TPTP addresses the entire test and performance life cycle, from early testing to production application monitoring, including test editing and execution, monitoring, tracing and profiling, and log analysis capabilities

ii)http://www.eclipse.org/tptp/

b)Jelly – Functional Testing on NetBeans platform

(1)The NetBeans Platform's extension to Jemmy is named Jelly. It provides a set of operators that are tailored to UI components used specifically in the NetBeans Platform, such as TopComponentOperator

ii)http://wiki.netbeans.org/JellyTools

2)Bug Tracking

a)Mantis (http://www.mantisbt.org/)

3)A mix

a)XQual Studio (XStudio)

i)A 100% Free graphical and modular test management application that handles the complete life-cycle of your QA/testing projects from end to end: users, requirements, specifications, development projects (scrum oriented), SUTs, tests, testplans

ii)Using a MySQL database as principal storage, XStudio allows you to schedule or run directly fully-automated or manual tests. Because XStudio can be used with any kind of tests (C/C++, Java, C#, Python, Perl, XUnit, VBScript, JavaScript or any proprietary systems such as QTP, AutoIt, Selenium, VisualStudio, TestComplete, Sahi, Ranorex, Squish, TestPartner, JMeter etc.), anyone from any kind of industry can take advantage of it

(a)http://www.xqual.com/products/xstudio.html

(b)Java Launcher already provided with download

**Glossary**

Quality Assurance – planned or systematic actions necessary to provide adequate confidence that a product or service is of the type and quality needed and expected by the customer.

Product – any artifact which is the output of any process used to build the final software product

Final software and system performance

Entire system requirements specification

Software requirements specification for a software component of a system

Design module, code, test documentation, or reports produced

Error - a mistake in the system under test; usually but not always a coding mistake on the part of the developer.

Defect - nonconformance to requirements or functional / program specification

Defect Characterizations

Bug – a fault in a program which causes the program to perform in an unintended or unanticipated manner.

Error – a mistake in the system under test; usually but not always a coding mistake on the part of the developer.

Defect – nonconformance to requirements or functional / program specification

Bug – a fault in a program which causes the program to perform in an unintended or unanticipated manner.

Functional vs. non-functional requirements –

Software development life cycle – a sequence of phased activities that represent the various stages of engineering through which software development passes

- Requirement analysis
- Architecture design
- Plan
- Detailed design
- Construct
- Review
- Refactor
- Testing
- Post mortem

Verification – process of determining whether or not the products of a given phase of the software development cycle meets the implementation steps and can be traced to the incoming objectives established during the previous phase.

Validation – process of evaluating software at the end of the software development process to ensure compliance with software requirements.

Walkthrough – review of requirements, designs or code characterized by the author of the material under review guiding the progression of the review.

**Bibliography**

HUMPHREY, W. 2000. “The Baseline Personal Process” in A Discipline for Software Engineering. Addison-Wesley, Boston.

TRIPP, L. (chair), et al. 2004. IEEE Computer Society Professional Practices Committee, 2004. Guide to the software engineering body of knowledge Project (SWEBOK) and the Guide. IEEE Computer Society Press, Los Alamitos, CA, (2004). Available at http://www.swebok.org.

SOFTWARE QUALITY. http://en.wikipedia.org/wiki/Software_quality

**CS1 Activities**

- Lecture with slides
- These activities can be done individually or in small groups. After a specified time, discuss the students' finding as a class. The activities may also be a lab or homework assignment.
- The scenario and code could be anonymous submissions from a previous assignment.

- **Activity 1**: Display a set of code with issues. Instruct the students to record the syntax and style errors found. Start with simple standards and add at the students' skill level increases.
  6. Constant Naming Conventions:
  7. Variable Naming Conventions:
  8. Comments Throughout:
  9. Separation of Sections of Code:
  10. Tabbing Conventions:

- **Activity 2**: Give the students a simple scenario. Instruct them to record the functional requirements. Discuss the functional requirements as a class. Show the students a set of code for the scenario with requirement issues. Instruct them to review and then discuss as a class.

**CS2 Activities**

- Lecture with slides

- **Activity 1:** discussion on the two topics:
  - Elements that influence the other product quality
    Start the discussion of what are the common factors that influence the other product quality: materials, design, produce process, maintenance, usability…
  - Elements that influence software product quality
    Introduce some basic elements that will influence software product quality: the quality of software engineers, design, process management, usability…

- **Activity 2**: Include the students in determining the students in how assignment products will be validated and verified.

- **Activity 3**: Revisit CS1 Activity 2 with a more complex scenario. allow them to find additional functional requirements as well as identify requirement based on the quality concepts discussed within class

- **Activity 4**: Walkthrough
  Introduce by conducting a sample walkthrough of a piece of code. Allow the students to discuss and point out changes they think that needs to be made. Also, try to come up with an alternative solution for the problem and present that solution as well.
  - **Assignment:** Give students a set of requirements and a piece of code that follows those requirements. Allow them to identify some quality requirements while doing a walkthrough of the piece of code they were given. If they think the piece of code is an optimal solution, they must give evidence based upon the quality notions above and implement a less than optimal solution. If they think it is not an optimal solution, they must implement an optimal solution and tell why the solution they implement is better than the one they was given. (Group or individual assignment)
  - **Optional Assignment:** Peer review of the requirements identified by the students. Instructors can allow the students to work in groups of three or more to peer review each other work. (If you choose groups, each group could switch code with another group)

- **Activity 5**: Present a relativity complex scenario. Ask the students to determine the functional and non-functional requirements and record them on the Requirements Worksheet.
- The logs and instructions for using the logs used in this activity are an adaption of those found in HUMPHREY, W. 2000. "The Baseline Personal Process" in A Discipline for Software Engineering. Addison-Wesley, Boston.

## REQUIREMENTS WORKSHEET INSTRUCTIONS

| | |
|---|---|
| Purpose | This form is for writing both functional and non-functional requirements. |
| General | Record on this worksheet the requirements identified for the given assignment.<br>Describe each requirement with as must detail an necessary. One requirement may require multiple worksheets. |

| **Column** | |
|---|---|
| Project name | Enter the assignment name provided by the instructor |
| Project objective | Write a brief description of the project objectives |
| No. | Assign each requirement a sequential number. |
| Section | If the assignment is divided into sections, record the appropriate section. |
| Description | Write a brief description of the problem or function that needs to be implemented. |
| Functional or non-functional | Note whether the requirement is functional or non-functional. |
| Indentified by | Record the person who identified the requirement. This could be you, a teammate, or the instructor. |

## REQUIREMENTS WORKSHEET*

Student ______________________________Class ______________ Assignment # _______

Project Name: ____________________________________________________________________________

Project Objective:

| No. | Section | Descriptions | F or NF | ID By |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

* adaption of requirements worksheet found in HUMPHREY, W. 2000. "The Baseline Personal Process" in A Discipline for Software Engineering. Addison-Wesley, Boston.

## Software Requirements Engineering Curriculum Module

**Preface**
The purpose of these teaching modules is to demonstrate how software engineering knowledge areas and principles can be imprinted into teaching computer science at the CS1 and CS2 levels. It is not intended to replace material and topics that are necessary in the curricula. It is hoped that the information presented in this module will enhance the learning experience of the student.

**Module Description**

This module attempts to identify concepts and skills associated with software requirements engineering that can be introduced or encouraged at the CS1 and CS2 levels. Software requirements engineering is "... concerned with the elicitation, analysis, specification, and validation of software requirements." Software requirements "express the needs and constraints placed on a software product that contribute to the solution of some real-world problem." (SWEBOK)

The SWEBOK breaks the knowledge area of Software Requirements Engineering into seven areas of study and then further subdivides those into topics. Here is a tree of the topics by areas of study and an indication of whether or not we will discuss how each can be introduced at the CS1 or CS2 levels.

- Software Requirements Fundamentals
    - Definition of a Software Requirement CS1
    - Product and Process Requirements CS1
    - Functional and Non-Functional Requirements CS1
    - Emergent Properties
    - Quantifiable Requirements CS1
    - System Requirements and Software Requirements
- Requirements Process
    - Process Models
    - Process Actors
    - Process Support and Management
    - Process Quality and Improvement
- Requirements Elicitation
    - Requirements Sources CS2
    - Elicitation Techniques CS2
- Requirements Analysis
    - Requirements Classification CS1
    - Conceptual Modeling
    - Architectural Design and Requirements Allocation
    - Requirements Negotiation
- Requirements Specification
    - System Definition Document
    - System Requirements Specification
    - Software Requirements Specification CS1, CS2
- Requirements Validation
    - Requirements Reviews CS1
    - Prototyping

    - Model Validation
    - Acceptance Tests CS1
- Practical Considerations
    - Iterative Nature of Requirements Process CS1
    - Change Management
    - Requirements Attributes
    - Requirements Tracking CS2
    - Measuring Requirements CS1

**Philosophy**

Software Requirements Engineering (and Requirements Engineering in general) is often over looked in curriculum's and can lead to students failing to understand their importance when they reach the work force as well as to the students forming bad habits like making assumptions and allowing requirements or feature creep. The SWEBOK points out that projects that perform poor requirements engineering processes often result in poor software. Teaching requirements engineering prepares the students not only for a more advanced class on requirements engineering in the future, but also provides them with skills with which to handle requirements classification, elicitation, analysis, and negotiation in everyday assignments. This can encourage the students to analyze their assignment and to ask questions if things aren't clear enough rather than making assumptions. Additionally, it can help them identify what is being asked of them for the assignments and deliver the appropriate solution, which will lead to less frustration and less wasted time on both the educator and students.

Requirements drive the software development effort. Being able to understand the difference from a well-defined requirement and a poorly defined requirement is an important part of a software developer's ability.

**Outcomes**

The goal of this module is to provide some methods for introducing requirements engineering concepts into the CS1 and CS2 level courses. Specifically they should:

- Learn to identify requirements and classify them
- Determine how each requirement can be quantified
- Evaluate requirements using acceptance tests and measurements
- Identify requirement sources and basic requirements elicitation techniques
- Be able to explain the difference between a functional and non-functional requirement
- Learn some of the techniques for eliciting requirements
- Be able to write an unambiguous requirement
- Understand how requirements trace through the whole software lifecycle

**Prerequisite Knowledge**

Software Requirements Engineering requires only basic knowledge of software context. Requirements Engineering can be introduced with only previous life experiences. Therefore, these topics should be easily conveyed and understood by the students with no prerequisite knowledge.

**Outline**

1) CS1
   a) Introduction
      i) Software Engineering
      ii)Software requirement
   b)It is important that the whole class is on the same page as to what a requirement is. Once a basic definition of requirement is established, it is important to include and label requirements in future assignment descriptions. Often the problem will be presented as prose and a bullet list of deliverables. From this the problem statement and the requirements can be identified.
      i) Requirement
      ii) Software Requirement
      iii) Product Requirement
      iv) Process Requirement
      v) Functional Requirement
      vi) Non-Functional Requirement
      vii) Quantifiable Requirement
   c) Illustrating the difference between Product/Process and Functional/Non-Functional Requirements
      i)Product and process requirements
      ii)Functional and non-functional requirements
      iii)Well-written software requirements specification (SRS)
   d) Integrating Requirements Identification and Classification into the Curriculum

2) CS2
   a) Continuation of CS1 Components
   b) Identify Requirement Sources and Practice Requirements Elicitation
      i)Elicitation Techniques
      ii)Requirements Specification revisited
   c)Requirements Traceability

**Annotated Outline**

1)CS1
   a)Introduction
      i) Software Engineering
         Applies a systematic, disciplined, quantifiable approach (or process) to the development, operation, and maintenance of software.
      ii)Software requirement
         (1)A need or constraint place on a software product that contributes to the solution of a real-world problem. (SWEBOK)
         (2)A challenge of identifying software requirements is to find, communicate, and remember what is really needed, in the form that clearly communicates to the customer and development team members.
   b)It is important that the whole class is on the same page as to what a requirement is. Once a basic definition of requirement is established, it is important to include and label requirements in future assignment descriptions. Often the problem will be presented as prose and a bullet list of deliverables. From this the problem statement and the requirements can be identified.
      i)Requirement
      ii)Software Requirement
      iii)Product Requirement
      iv)Process Requirement

v)Functional Requirement
vi)Non-Functional Requirement
vii)Quantifiable Requirement

c)Illustrating the difference between Product/Process and Functional/Non-Functional Requirements

i)Another easy distinction that can be explained in class and reinforced in the description of homework assignments is the difference between product and process requirements. Simply dividing the assignment requirements into product and process requirements will keep the students familiar with the words and the distinction. This same process can be used to teach the concept of Functional vs. Non-Functional requirements.

ii)Explain difference between functional and non-functional requirements

iii)Well-written software requirements specification (SRS) (DAVIS, 1993)

(1)Should say *what*, not *how*.
(2)Correct: does what the client wants, according to specification
(3)Verifiable: can determine whether requirements have been met
(4)Unambiguous: every requirement has only one interpretation
(5)Consistent: no internal conflicts
(6)Complete: has everything designers need to create the software
(7)Understandable: stakeholders understand enough to buy into it
(8)Modifiable: requirements change

iv)See CS1 – Activity 1

d)Integrating Requirements Identification and Classification into the Curriculum

(a)To help reinforce the differences in requirements and to help teach requirements classification you can use a simple spreadsheet to list the requirements for an assignment.

(b)See CS1 – Activity 2

2)CS2

a)Continuation of CS1 Components

Rather than introduce tones of additional topics related to requirements engineering in CS2, it would be better to continue to provide requirements engineering components used in CS1 in the CS2 assignments, remembering to use the same vocabulary and layout in the assignment statements to provide a sense of familiarity.

b)Identify Requirement Sources and Practice Requirements Elicitation

Now that the students have had more than a semester of experience reading and classifying requirements, we can approach the topic of where requirements come from in real life and touch of the topic of requirements elicitation.

i)Elicitation Techniques

(1)Interviews
(2)Scenarios
(3)Prototypes
(4)Facilitated meeting
(5)See CS2 – Activity 1.

ii)Requirements Specification

(1)Review the qualities of a well-written SRS
(2)Revisit CS1 – Activity 1 and 2 with a more complex problem.

c)Requirements Traceability

i)Trace requirement from source to design to implementation to test
ii)Ensure that all requirements have been implemented
iii)Used to analyze adverse effects of planned software changes
iv)CS2 Activity 3 – Create RTM for programming assignment
v)See CS2 – Activity 2.

**Teaching Resources**

Requirements classification table

**Teaching Techniques**

CS1 Activities

- Lecture with slides
- Provide programming assignments and projects with various format of software requirements: use cases, textual shall statements, and user stories
- Provide SRS for final programming assignment / project

CS2 Activities

- Lecture with slides
- Provide students with SRS template for creating their own SRS
- Conduct a Requirements Elicitation meeting with individual students or teams
- Conduct a Requirements Review with individual students or teams
- Provide students with a RTM template

**Tool Support**

Spreadsheets

Using spreadsheets you can have "on paper" exercises to help the students get used to classifying and analyzing requirements that are given for assignments.

Google Forms

You can create a Google "Form" for free in the google docs suite, which will allow you to ask questions of the students, survey style, and have them put into a private spreadsheet. Here is an example one I have setup:
https://spreadsheets.google.com/spreadsheet/viewform?formkey=dFU5WXl4cDFMN3FZSEd5eHVjNzNtdHc6MQ
The results can be viewed here:
https://spreadsheets.google.com/spreadsheet/ccc?key=0AtK1DJ1zjC_kdFU5WXl4cDFMN3FZSEd5eHVjNzNtdHc&hl=en_US

Microsoft Word or any word processor – used to create SRS and RTM

Easy to use

Microsoft Visio – Create Use Case Diagrams

Easy to use
Freely available with academic msdn
Exposes students to UML diagrams
Can be used for design diagrams as well

Enterprise Architect

Higher learning curve compared to Visio and Word
Not freely available. Can get a free 30 day trial. Academic License available at a reduced price
Can be used for each phase of the software development lifecycle
Can document requirements in use cases or traditional shall statements

**Glossary**

Using consistent and correct vocabulary when you are describing the activities is very important. This introduces students to the vocabulary related to requirements engineering. Terms applicable to the requirements engineering are list in the Glossary as well as being further explanation given in this outline.

Requirement - is a singular documented need of what a particular product or service should be or perform.

Software Requirement - is a requirement specifically about a piece of software or the processes specific to that software. A property which must be exhibited by software developed to solve a particular problem.

Product Requirement - describes properties of a system or product.

Process Requirement - describe activities performed by the developing organization. For instance, process requirements could specify specific methodologies to be followed and constraints that the organization (or class) must obey.

Functional Requirement - describe the functionality that the system behaviors; for example, formatting some text or modulating a signal. They are sometimes known as capabilities.

Non-Functional Requirement - describe characteristics of the system that the user cannot affect or (immediately) perceive. Nonfunctional requirements are sometimes known as quality requirements or “ilities”, i.e. reliability, maintainability, portability, availability, etc.

Quantifiable Requirement - Requirements that are not vague and are verifiable or measurable.

Requirements Elicitation - is concerned with where software requirements come from and how the software engineer can collect them.

Stakeholder - Anyone who has a stake in the project, typically: Users, Customers, Market Analysts, Regulators, and Software Engineers.

Requirements Traceability Matrix – a document in the form of a table that correlates requirements, design, implementation and test to determine the completeness of the relationship.

Use case – description of steps or actions between a user and a software system.

**Bibliography**

CHANG, C., DENNINGS, P., et al. 2001. Computing curricula 2001: Computer science. Final report (December 15, 2001). *IEEE Computer Society Press and ACM Press* (Dec. 15, 2001). Available at http://www.acm.org/education/curric_vols/cc2001.pdf.

DAVIS, A. 1993. *Software Requirements: Objects Functions and States*, Prentice-Hall Inc.

TRIPP, L. (Chair), et,al, 2004. IEEE Computer Society Professional Practices Committee. Guide to the software engineering body of knowledge Project (SWEBOK) and the Guide. *IEEE Computer Society Press*, Los Alamitos, CA, 2004. Available at http://www.swebok.org.

WIKI. 2011. Requirements. *Wikipedia, The Free Encyclopedia*. Wikimedia Foundation, Inc. Viewed on 21 July 2011.

**Suggested Course Activities**

**CS1 – Activity 1**.

**Requirements Classification Exercise Example**

Then have the students go through and label each requirement as Product or Process requirement and then as a Functional or Non-Functional requirement. This assignment can be due a few days before the assignment is due and has the side effect of ensuring the students have read the requirements in advance of the due day and have to comprehend them enough to classify them. I think this will increase the clarification of the assignment in the minds of the students and help fend off procrastination. See the Requirements classification example in the Suggested Course Activities

In this exercise the students must look at the unclassified requirements of the assignment (or a list of requirements supplied to use this as a quiz or test question) and classify them as product or process and as functional or non-functional.

Below is example answer key to the exercise, the students would be given the requirements only and be asked to enter process or product in the second column and functional or non-functional in the third column.

| **Assignment Requirement** | **Product or Process?** | **Functional or Non-Functional?** |
|---|---|---|
| Your assignment should include usage documentation, according to the class standards. | *Process* | *Non-Functional* |
| Your program must be written in Python. | *Product* | *Non-Functional* |
| Your program must print the message `The number is zero.`, if the input is zero. | *Product* | *Functional* |

In class, present a problem scenario using a document camera, presentation slide, handout, etc.
Lead a class discussion or instruct the students to work individually or in small groups then discuss findings as a class to:

- Identify the requirements
- Classify each requirement as a product or process and functional or non-functional

Take home assignment could requirement students to identify and classify the requirements in an assignment scenario problem.

**CS1 – Activity 2**

**Iterative Nature of Requirements Process**

The SWEBOK repeated points out that the requirements engineering process is something that happens iteratively over the life cycle of the project. In order to reinforce this concept you can describe how requirements in a real life project can change often and that it is important to continually analyze and reevaluate the requirements of the project. This can either be demonstrated in a longer project or in a series of assignments that build on each other. The assignments below can be used to demonstrate how new requirements can affect the work you have done up until then.

**Interative Requirements Exercise**

Here are two exercises to illustrate the iterative nature of the requirements process by iterating on the previous assignment by only adding new requirements for the next assignment.

**Assignment 1**

Problem Statement

You must design a program in the course programming language that receives input from the user and print one of three messages. You must prompt the user. If the input is a positive number then your program must exactly output: "The number is positive." If the input is zero: "The number is zero." If the input is negative: "The number is negative."

Along with your program you must include usage documentation that describes how to run and use your program according to our class standard. Additionally, you are to keep track of your defects in the provided defect log. Your program is to be submitted via the class website using the normal assignment delivery procedure.

Requirements

Process Requirements

- The assignment artifacts (Program and Documentation) must be turned in via the class website using the normal assignment delivery procedure.

Product Requirements – Non-Functional

- Your program must be written in [course computer language].
- Your program must use the console to receive input and display messages.
- Your assignment must also contain usage documentation.

Product Requirements - Functional

- Your program must receive input from the command line, by prompting the user.
- Your program must print the message "The number is positive.", if the input is a positive integer or decimal.
- Your program must print the message "The number is zero.", if the input is zero.
- Your program must print the message "The number is negative.", if the input is a positive integer or decimal.

Example Output

```
>>> 16<return>
The number is positive.
>>> 0.0<return>
The number is zero.
>>> -16.0<return>
The number is negative.
```

**2)**

Solution program. Python is used here.

```
"""Assignment 1"""

def main():
   """This function implements assignment 1"""
   while True:
      user_input = input(">>> ")
      number = float(user_input)
      if number == 0:
         print("The number is zero.")
      elif number > 0:
         print("The number is positive.")
      elif number < 0:
         print("The number is negative.")

if __name__ == '__main__':
   main()
```

**Interative Requirements Exercise**

**Assignment 2**

Problem Statement

You must design a program in the course programming language that receives input from the user and print one of three messages. You must prompt the user. If the input is a positive number then your program must exactly output: “The number is positive.” If the input is zero: “The number is zero.” If the input is negative: “The number is negative.” New requirement: If the input is not a valid integer or decimal number then the message: “That is not a number.” should be printed. Along with your program you must include usage documentation that describes how to run and use your program according to our class standard. Your program is to be submitted via the class website using the normal assignment delivery procedure.

Process Requirements

- The assignment artifacts (Program and Documentation) must be turned in via the class website using the normal assignment delivery procedure.
- Defects must be logged in the provided defect log.

3)Product Requirements - Non-Functional

- Your program must be written in [course computer language].
- Your program must use the console to receive input and display messages.
- Your assignment must also contain usage documentation.

4)Product Requirements - Functional

- Your program must receive input from the command line, by prompting the user.
- Your program must print the message “The number is positive.”, if the input is a positive integer or decimal.
- Your program must print the message “The number is zero.”, if the input is zero.
- Your program must print the message “The number is negative.”, if the input is a positive integer or decimal.
- Your program must print the message “That is not a number.”, if the number is not a valid integer or decimal number.

Example Output

```
>>> 16<return>
The number is positive.
>>> 0.0<return>
The number is zero.
>>> -16.0<return>
The number is negative.
>>> q<return>
That is not a number.
```

Solution program**.** Python is used here.

```
"""Assignment 2"""

def main():
   """This function implements assignment 2"""
   while True:
      user_input = input(">>> ")
      try:
         number = float(user_input)
         if number == 0:
            print("The number is zero.")
         elif number > 0:
            print("The number is positive.")
         elif number < 0:
            print("The number is negative.")
      except ValueError as e:
         print("That is not a number.")

if __name__ == '__main__':
   main()
```

**CS2 - Activities**

**CS2 Activity 1**

Using Requirements Elicitation for Assignment Assessment

To demonstrate requirements elicitation, the instructor plays the part of a customer who has a project scenario that need to be solved. The students in the class play the part of developers and ask questions of the customer about the project and develop a list of requirements.

This can also be done with pairs of students with one being the customer and the other being the developer or a custom student and a small group of student developers.

**CS2 – Activity 2**

Using Requirements Measurement for Assignment Assessment

Students can be reminded of requirements and unsuspectingly taught the concept of trace-ability. Each error in their program (or each test case they failed) is tied back to a requirement (if applicable as things like logic errors don't really fall apply). For example, if the student's project failed a test case by giving a negative number to a function where the requirement that said the input must be positive or zero then that requirement could be reference with the error. This demonstrates that and error can be traced back to a specific requirement and reinforces the importance of requirements.

Assignment

Students are to make a table of the requirements for an assignment scenario as in CS1 – Activity 1.
As they test the solution program, they are to record errors and the requirement that applies.

Software Configuration Management Curriculum Module

**Preface**

This teaching module is to describe some of the ground work for students to understand Software Configuration Management. CS1 and CS2 are a great place to begin introducing SCM topics. The basic understanding of SCM principles can helps students with their managing their home as well as any projects they encounter.

**Module Description**

Software Configuration Management (SCM) is emphasizes the importance of configuration control in managing software development. It is a set of operations and tools to control your projects configuration.

The SWEBOK defines SCM as "a discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements."

The SWEBOK defines the following sub-sections of Software Configuration Management:

- Management of the SCM Process
  - Organizational Context for SCM
  - Constraints and Guidance for SCM Process
  - Planning for SCM CS1
  - Software Configuration Management Plan
  - Surveillance of SCM
- Software Configuration Identification
  - Identifying Items to be Controlled CS2
  - Software Library
- Software Configuration Control
  - Requesting, Evaluating and Approving Software Changes CS1
  - Implementing Software Changes CS2
  - Deviations and Waivers
- Software Configuration Status Account
  - Software Configuration Status Information
  - Software Configuration Status Reporting
- Software Configuration Auditing
  - Software Functional Configuration Audit
  - Software Physical Configuration Audit
  - In-Process Audits of a Software Baseline
- Software Release Management and Delivery
  - Software Building
  - Software Release Management

**Philosophy**

Software Configuration Management can teach students to better manage their projects by providing:

- an integral part of the software development process in all phases of the life cycle.
- an understanding of configuration identification
- a mechanism for controlling change
- a strategy for breaking down large projects into manageable components
- an understanding of tractability on bugs and feature requests
- a strategy for adapting to changes in requirements or specifications

**Outcomes**

Students who have had a proper introduction to Software Configuration Management should have:

- a basic understand of the need to manage a project
- a realization that SCM extends over the entire software development life cycle
- understand the term baseline
- the ability to explain why configuration management is required
- understand the difference between discrepancies and requested changes
- the ability to form a change request and implement those changes

**Prerequisite Knowledge**

Some prerequisite knowledge of Software Configuration is needed. Minimal understanding of some management processes would be helpful and knowledge of software requirements.

**Outline**

1) CS1
    a) Introduction
        i) Software Engineering
        ii) Software Configuration Management
    b) Configuration management as a controlling tool
    c) Types of changes
        i) Discrepancies
        ii) Requested changes
    d) Configuration identification

2) CS2
    a) Review the CS1 material.
    b) Version control
    c) Configuration Management Planning

3) CS3+
    a) Large scale group project
    b) Introduction of Basic Source Management Tools

**Annotated Outline**

4) CS1
   a) Introduction
      i) Software Engineering
         Applies a systematic, disciplined, quantifiable approach (or process) to the development, operation, and maintenance of software.
      ii) Software Configuration Management
         (1) a set of operations and tools to control your projects configuration
         (2) emphasizes the importance of configuration control in managing software development
         (3) identifies and documents the functional and physical characteristics of a configuration item
         (4) control changes to those characteristics
         (5) record and report change processing and implementation status
         (6) verify compliance with requirements
   b) Configuration management as a controlling tool
      i)Changes to one configuration can affect others.
      ii)Monitoring change and its effects helps maintain integrity of the development process
      iii)Change requests and discrepancy reports are evaluated before allowing the a change
      iv)Provides a way to track changes and prevent changes that cause problems
   c)Types of changes
      i)Discrepancies
         (1)Requirement errors: caused by incomplete or incorrect requirements
         (2)Development errors: caused by incorrectly implementing a requirement
      ii)Requested changes
         (1)Unimplemented requirements: a requirement is poorly or not implemented
         (2)Enhancements: additional requirements
         (3)Improvements: non-functional changes to improve the product
   d) Configuration identification
      i) Identify the configuration items that will be affected by a change
      ii) Change Request Forms
         (1) Describes an a possible change to the system or a configuration item
         (2) Documents the decision on whether or not the change will be implemented.
         (3) CS1 – Activity 1.

5) CS2
   a) Review the CS1 material.
   b) Version control
      i)Simultaneous update. If not monitored properly, one programmer or developer can make changes that will cancel or not work with another's change.
      ii)Tools for version control exist.
         (1)See Tool Support below.
         (2)Automated tools are not used in the activities of the module. The emphasis here is to teach concept.
   c) Configuration Management Planning
      i)Making multiple changes to a product or system requires coordination and communication to ensure a working product results
      ii)See CS2 – Activity 1.

6) CS3+
   a) Large scale group project
      i) Identify someone as the configuration manager who approves change requests and managing baselines.
      ii) Other students will act as programmers and be responsible for implementing changes
   b) Introduction of Basic Source Management Tools
      i) A basic introduction would be useful for students who are interested in managing their code from the start. Being introduced to simple tools such as a source code management tool at the beginning of their

first course could help the students with their homework as well as teach them a valuable tool for their career.

ii) See Tool Support below.

**Teaching Resources**

Change Request Forms

**Teaching Techniques**

Lecture using presentation or a document camera to introduce the material.
The CS1 activity may be done with a large group discussion in class or with individual or small group discussion with class discussion.
The CS2 activity may be started in class with the small group discussion out of class. The final discussion will involve everyone and should be done in a lab or class setting.

**Tool support**

- Source Management Tools:
  - git - A distributed version control system
  - Microsoft SourceSafe - Integrates well into Microsoft Visual Studio
  - subversion (otherwise known as svn) - A popular
  - fossil - A distributed version control system
- SCM Project Tools:
  - Redmine - A project management tool that contains multi-project support, bug tracking, feature tracking, wiki, documentation tracking, integration with source management tools and milestone tracking.
  - Trac - Another project management tool that contains bug tracking, feature tracking, wiki, documentation tracking, integration with source management tools and milestone tracking.
  - fossil - A distributed project management tool that contains bug tracking, feature tracking, a wiki, and milestone tracking within the project repository instead of in an external project.

**Glossary**

Baseline – the state of a configuration item that is agreed upon to be correct and will serves as the basis of future changes

Configuration control – managing change to a configuration item

Configuration item – an object that is created as part of the software engineering development process, like specification, design, code, and tests

Discrepancy – a software error caused by improper implementation of a requirement or failing to implement a requirement

Enhancement – a modification to a product to improve or expand its purpose

**Bibliography**

"Fossil - About", *Fossil SCM*. Website http://fossil-scm.org/

"About Git", *git - Fast Version Control System.* Website http://git-scm.com/

TRIPP, L. (chair), et al. 2004. IEEE Computer Society Professional Practices Committee. 2004. Guide to the software engineering body of knowledge Project (SWEBOK) and the Guide. IEEE Computer Society Press, Los Alamitos, CA, (2004). Available at http://www.swebok.org.

**Suggested Course Activities**

**CS1 – Activity 1**

Select a previous project or assignment that re-uses old code, documentation, tests, etc. Present change request and ask the students to identify which configuration items will be affected. After the changes have been identified and approved, instruct the students to implement the changes as a new assignment. Point out to the students that the original items given as the baseline from which to work.

**CS2 – Activity 1**

Revisit CS1 – Activity 1. This time divide the students into groups and give each group a different change request for the same old project or assignment. Allow each group to review the change and identify affected configuration items. Bring all groups together into a discussion of what and where changes are needed. Remember a change can affect multiple configuration items.

Note that multiple changes make be needed to the same configuration item and one change may affect another. Lead the students in a discussion of which changes should be implemented first and would that change affect the implementation of additional changes. All changes made should be documented as how they affected any of the configuration items.

The group implementing the first change to a configuration item is given copies of the baseline (original) products. At the acceptable end of the change, a new baseline is available for the next change and so on. Remember if complication arise during a change implementation that cannot be resolved that the baseline is always available to fall back on.

This illustrates the importance of configuration management. If there is no control over how and when changes are made the changes may conflict and create a worse product than before.

**Suggested Change Request Form (Outline)**

Project Name:
Change#:
Type of Request: [Enhancement] or [Defect] or [Other]
Submitted By:
Date Submitted:

Short Description:

Full Description:

Resolution: [Fixed] or [Rejected] or [Workaround] or [In Progress]
Closed On:
Assigned To:
Severity:
Dependencies:
Fixed in Revision: