**Instructor’s Guide**

*Software Engineering:*

*A Practitioner’s Approach*

Eighth edition

Roger S. Pressman, Ph.D.

Bruce R. Maxim, Ph.D.PREFACE

Software has become the dominant technology of this new century—dwarfing most “twentieth century” businesses, influencing nearly every aspect of our economy, differentiating products and services of all kinds, and (via the Internet) offering a completely new paradigm for the way we interact with one another at nearly every level. Software engineering—a field that combines principles of project management, computer science and engineering methodology—is essential for the construction of software-based systems. It is a topic that should be studied by all industry practitioners and taught in all serious computer science and computer engineering programs.

Prior to the first edition of *Software Engineering: A Practitioner's Approach* (SEPA), there were no textbooks that adequately treated the entire software engineering process, while at the same time presenting the subject in a style that is amenable to a University or industrial course format. There were fewer than 10 institutions in the United States that offered a single software engineering course. More than 30 years have passed since the first edition was published, and during that time software engineering courses (of one type or another) have proliferated and are now offered at well over 500 institutions in the US and in many hundreds of universities worldwide. Dozens of books on the subject are now published.

With this edition of *Software Engineering: A Practitioner’s Approach,* Bruce Maxim joins me (Roger Pressman) as a co-author of the book. The eighth edition of *Software Engineering: A Practitioner's Approach* represents our best attempt at providing a reasonably comprehensive guide for software engineering. A review of the second edition in *IEEE Software* (January, 1988) called the book "a software engineer's Baedecker's Guide." I thought this really did capture the essence of SEPA. My intent in each of the editions has been to provide a travel guide for computer science and engineering students as they begin their journey into this important technology. But the book is also designed as a guide for practitioners—industry professionals who must learn about this important subject.

A travel guide cannot provide street maps of every town or list every phone number of every inhabitant of a country. Similarly, a software engineering travel guide cannot present comprehensive coverage of every important topic or it will become prohibitively long (after all, this seventh edition requires well over 800 pages!). Both can, however, select those things that the authors believe are important and teach you enough about the language, the main travel routes, the customs, the laws and the taboos to enable you to make the journey a pleasant one. If you intend to be a native, more detailed study, conducted over many years, will be necessary.

I have been overwhelmed by the widespread acceptance and use of earlier editions of SEPA—more than 2 million copies, published in eight different languages, have been sold worldwide. In writing the eighth edition, Bruce and I have attempted to extend both the breadth and depth of earlier editions, while at the same time maintaining the attributes that have lead to the book's success. A comprehensive website complements the book. The website provides much information that will be of use to you, your students, and industry professionals alike. We hope that it proves to be a valuable supplement for your course.

During our years as professors, Bruce and I had the opportunity to review many instructors’ manuals. Our goal for this one has been to develop the kind of supplementary tool that we have often desired, but rarely encountered. We hope that you find this manual useful.

*Roger S. Pressman*

ACKNOWLEDGEMENTS

The eighth edition of *Software Engineering: A Practitioner's Approach* (SEPA8e) and the resultant content of this *Instructor's Manual* have been influenced by the comments and suggestions of colleagues and students too numerous to mention. A student's question, a colleague's request for additional information, or a Professor's suggestion (some from as far away as India, China and Australia) have each in their own way contributed to the content of both the book and this manual. Our thanks to all of you.

In addition, my sincere appreciation to Elaine Kant for allowing me to use her paper as a foundation for Segment 5 of this manual. Her work remains one of the best "how to" guides for conducting a software engineering course. Finally, my thanks to Professor Tim Lethbridge of the University of Ottawa who developed the comprehensive case study that your students will find extremely useful.

BRM: My thanks to my wife Norma for her understanding and support as I worked on another book.

RSP: Writing an instructor's manual comes as a shock to most authors. You've just finished one book and you have to write another one under significant time pressure. My wife, Barbara, understands the challenges and allows me to meet them. For that, my love and thanks.

FORETHOUGHT

A great teacher is my adversary, my conqueror, commissioned to chastise me. He leaves me tame and grateful for the new language he has purloined from other kings whose granaries are filled and who libraries are famous. He tells me that teaching is the art of theft: of knowing what to steal and from whom.

Bad teachers do not touch me; the great ones never leave me. They ride with me during all my days, and I pass on to others what they have imparted to me. I exchange their handy gifts with strangers on trains, and I pretend the gifts are mine. And the truly wonderful thing about them is they would applaud my theft, laugh at the thought of it, realizing they had taught me their larcenous skills well.

Pat Conroy

*The Lords of Discipline*

Houghton Mifflin Co., 1980

1 INTRODUCTION TO SEPA

*Software Engineering: A Practitioner's Approach* , 8/e (abbreviated SEPA or SEPA8e in this guide) provides a foundation for university and industry courses in software engineering. The book presents the entire software engineering process in a partitioned fashion that is amenable to term (quarter, trimester, or semester) courses at the university level. SEPA can also be used with success as the text for industry training courses ("short courses") in software engineering.

The topic that we call *software engineering* is both exciting and frustrating. Exciting because it draws on many technical disciplines and provides a harness that binds each discipline to the next. Frustrating, because it demands knowledge in a multitude of topic areas and seems to be infinitely expandable. SEPA does not purport to be *the* definitive volume on software engineering—no text can satisfy such a goal because the field is changing too rapidly. Rather, this book does present a thorough treatment of software engineering, allowing the instructor to emphasize and augment (with state-of-the-art material) those subjects that are relevant to course participants.

SEPA8e is accompanied by a comprehensive website that has been organized to provide you and your students with a wide variety of teaching options and resources. Over 808 Powerpoint slides for classroom use are provided to you to help in teaching this important subject. Both can be acquired through the website. In addition, a vast array of other resources is available at the website.

See <http://www.mhhe.com/pressman> for details.

This *Instructor's Manual* has been written to assist you in the development and presentation of a course in software engineering. The manual includes a description of teaching options, guidelines for the use of the website resources, course design considerations and a chapter-by-chapter discussion of the book.

* 1. **OVERVIEW OF THE BOOK**

The 39 chapters of the eighth edition have been divided into five parts. This has been done to compartmentalize topics and assist instructors who may not have the time or inclination to complete the entire book in one term. Different parts of the book can be used in conjunction with resources provided within the SEPA website(described in detail in segment 2 of this manual) and will enable you to create lecture or lab courses that emphasize those topics that are relevant to your curriculum.[[1]](#footnote-1)

**Part 1, *The Software Process***—presents an introduction to the software engineering milieu. It is intended to introduce the subject matter, and more importantly, to discuss important software process models, including the agile model. It can be used in conjunction with portions of other parts of the book.

**Part 2, *Modeling****—*presents comprehensive treatment of analysis and design methods with an emphasis on UML modeling for analysis and design. Although an object-oriented approach is emphasized, conventional concepts and methods are also presented. It can be used alone or in conjunction with Parts 1, 3, and 4 if your primary emphasis is technical methods for software development.

**Part 3, *Quality Management****—*presents a reasonably comprehensive discussion of software quality, quality assurance, and software testing strategies and tactics. It incorporates formal modeling and verification methods (using somewhat abbreviated presentations adapted from the sixth edition). It also includes a chapter on software configuration and content management.

**Part 4, *Project Management****—*presents topics that are relevant to those who plan, manage, and control a software development project. It can be used as the basis for a course dedicated to the management aspects of software engineering or portions may be used to complement more technical software engineering courses.

**Part 5, *Advanced Software Engineering Topics****—*presents dedicated chapters that address software process improvement, emerging trends, and the road ahead. Used in conjunction with Parts 1 – 4, this part of SEPA encourages student to look toward the future of software engineering.

In summary, the five-part organization of the eighth edition enables you to "cluster" topics based on available time and student need. An entire one-term course can be built around one or more of the five parts. By organizing the eighth edition in this way, We have attempted to provide you with a number of teaching options, while providing your students with a software engineering textbook (and accompanying website) that introduces them to the broad scope of the subject and can serve them as a reference in future years.

A more detailed discussion of each SEPA chapter is presented in segment 5 of this manual.

**1.2 DIFFERENCES BETWEEN THE SEVENTH and EIGHTTH EDITIONS**

**Format and Style**

The format and style of SEPA, 8/e have been further refined to make the book more reader-friendly. Each of the book's 39 chapters begins with a "Quick Look" that provides immediate short answers to six questions about chapter content:

* What is it?
* Who does it?
* Why is it important?
* What are the steps?
* What is the work product?
* How do I ensure that I've done it right?

Throughout the book, marginal icons and related text are used to present important complementary information:

* The **Keypoint** icon will help you to find important points quickly.
* The **Advice** icon provides pragmatic guidance that can help you make the right decision or avoid common problems while building software.
* The **Question Mark** icon asks common questions that are answered in the body of the text.
* The **Quote** icon presents interesting quotes that have relevance to the topic at hand.
* The **WebRef** icon provides direct pointers to important software engineering related web sites.

Like the seventh edition, SEPA8e features a collection of *sidebars* that are intended to enhance the reading experience by providing supplementary information that will be of use to the reader. A number of different types of sidebars are presented:

* SafeHome—presents the story of a project team as it works on a software project. In essence SafeHome is a “mini-play” that allows students to better understand the topics presented in many SEPA chapters.
* General Info—present complementary information that is relevant to chapter topics
* Task set—presents the tasks (or workflow) that is required to accomplish a specific software engineering activity.
* Tools—present a set of representative tools (along with Web references) that are relevant to the topic at hand.

**Content**

The eighth edition of *Software Engineering: A Practitioner's Approach* (SEPA8e) is considerably more than a simple rewrite and update of the 7th edition. The book has been revised and restructured to improve pedagogical flow and emphasize new and important software engineering processes and practices. The vast majority of Web engineering topics presented in the 7th edition are integrated into relevant SEPA, 8/e chapters.[[2]](#footnote-2)

SEPA8e contains expanded discussion of many topics presented in the 7th edition and introduces many additional topics. These new topics include human aspects of software engineering, Mobile App engineering, cloud computing, and security engineering.

Overall, chapters have been reorganized and restructured. The writing style throughout the book continues to use the slightly less formal tone of the 7th edition, and the SEPA website has been further enhanced.

Many new **Problems and Points to Ponder** have been added. The **Further Readings and Information Sources** sections have been revised extensively and many new figures and examples are used. The following table identifies key changes on a chapter by chapter basis.

**TABLE OF CHANGES FOR SEPA, 8TH EDITION**

**Chapter 8th Edition Revisions**



**1 The Nature of Software**

A *moderate revision* of SEPA7e, Chapter 1. Includes a discussion on the impact of the web, mobility, the cloud and product line software on modern software development.

**2 Software Engineering**

A *moderate revision* of SEPA7e, Chapter 1. A broader discussion of contemporary trends is presented.

**Part 1—The Software Process**

**3 The Software Process**

A *moderate revision* of SEPA7e, Chapter 2. Presents a discussion of a generic software process by identifying a set of framework activities that appear in all process models. Also addresses process patterns.

**4 Process Models**

A *minor revision* of SEPA7e, Chapter 2. Reorganizes process models. Introduces the Unified Process and PSP/TSP.

**5 Agile Development**

A *minor revision* of SEPA7e, Chapter 3. Discusses the concept of agility and the agile process models that have been proposed to achieve it. Agile modeling is also introduced.

**6 Human Aspects of Software Engineering**

***A new chapter.*** This chapter discusses the characteristics of successful software engineers and the psychology of software engineering work. It also describes the impact of social media, the cloud, and collaborative tools on team software engineering processes.

**TABLE OF CHANGES FOR SEPA, 8TH EDITION**

**Chapter 8th Edition Revisions**



**Part 2—Modeling**

**7 Principles that Guide Practice**

A *minor revision* of SEPA7e, Chapter 4.Presents core principles that are relevant to major software engineering activities, regardless of technology applied or methods used. Practices are organized based on the generic process framework presented in Chapter 3.

**8 Understanding Requirements**

A *minor revision* of SEPA7e, Chapter 5.Focuses on the actions that are required to gather and organize stakeholder requires, Introduces use cases as a foundation for the creation of the analysis model.

**9 Requirements Modeling: Scenario-based**

***A moderate revision*** of SEPA7e, Chapter 6 ***.*** The first of three chapters on requirements modeling, this chapter presents scenario-based modeling.

**10 Requirements Modeling: Class-based**

***A moderate revision*** of SEPA7e, Chapter 6 ***.*** The second of three chapters on requirements modeling. This chapter presents class modeling and CRC modeling.

**11 Requirements Modeling: Behavior, Patterns, and WebApps**

***A moderate revision*** of SEPA7e, Chapter 7***.*** The third of three chapters on requirements modeling. This chapter expands on elements SEPA7e, Chapter 6 by presenting behavioral modeling, requirements patterns, and analysis methods for WebApps and MobileApps. Data flow modeling has been dropped from SEPA8e.[[3]](#footnote-3)

**TABLE OF CHANGES FOR SEPA, 8TH EDITION**

**Chapter 8th Edition Revisions**



**12 Design Concepts**

A *minor revision* of SEPA7e, Chapter 8, this chapter defines design within the context of software engineering and presents the design process, design concepts and the core elements of the design model.

**13 Architectural Design**

A *moderate revision* of SEPA7e, Chapter 9. Material on architectural review and conformance checking has been added. Discussion on agility and architectural design as well as the use of architectural patterns is also discussed. Architectural mapping using data flow has been removed from SEPA8e.

**14 Component-Level Design**

A *minor revision* of SEPA7e, Chapter 10, Includes material on WebApp components and CBSE topics. .

**15 User Interface Design**

A *minor revision* of SEPA7e, Chapter 11, includes material on designing Web and MobileApp user indterfaces.

**16 Pattern-Based Design**

A *moderate revision* of SEPA7e, Chapter 13Includes discussion of architectural, component-level, interface, WebApp, and MobileApp design patterns.

**17 WebApp Design**

A *minor revision* ofSEPA7e, Chapter 13.

**18 MobileApp Design**

***A new chapter.*** Discusses the challenges of creating MobileApps for multiple hardware platforms and the importance of context in designing a high quality mobile user experience. The use of cloud storage and web services is also discussed.

**TABLE OF CHANGES FOR SEPA, 8TH EDITION**

**Chapter 8th Edition Revisions**



**Part 3—Quality Management**

**19 Quality Concepts**

A *minor revision* of SEPA7e, Chapter 14.

**20 Review Techniques**

A *minor revision* of SEPA7e, Chapter 15.

**21 Software Quality Assurance**

A *minor revision* of SEPA7e, Chapter 16.

**22 Software Testing Strategies**

A *minor revision* of SEPA7e, Chapter 17.

**23 Testing Conventional Applications**

A *minor revision* of SEPA7e, Chapter 18.

**24 Testing Object-Oriented Applications**

A *minor revision* of SEPA7e, Chapter 19.***A new chapter.*** Presents testing techniques presented in SEPA, 6/e, Chapter 13 with an emphasis on testing for object-oriented software.

**25 Testing Web Applications**

A *minor revision* of SEPA, 6/e, Chapter 20.

**27 Security Engineering**

***A new chapter.*** Discusses the challenges of security and privacy in an online world. Accounting for security risks and the security assurance are considered as part of the design process.

**28 Formal Methods**

A *minor revision* of SEPA, 7/e, Chapters 21..

**29 Software Configuration Management**

A *moderate revision* of SEPA, 7/e, Chapter 22. Configuration management for Web and Mobile Apps has been added.

**TABLE OF CHANGES FOR SEPA, 8TH EDITION**

**Chapter 8th Edition Revisions**



**30 Product Metrics**

A *moderate revision* of SEPA, 7/e, Chapter 23. The number of metrics discussed has been reduced to those that are most popular in the software engineering community.

**Part 4—Project Management**

**31 Project Management Concepts**

A *minor revision* of SEPA, 7/e, Chapter 24.

**32 Process Metrics**

A *minor revision* of SEPA, 7/e, Chapter 25.

**33 Estimation for Software Projects**

A *minor revision* of SEPA, 7/e, Chapter 26.

**34 Project Scheduling**

A *minor revision* of SEPA, 7/e, Chapter 27.

**35 Risk Management**

A *minor revision* of SEPA, 7/e, Chapter 28.

**36 Maintenance and reengineering**

A *minor revision* of SEPA7e, Chapter 29

**Part 5—Advanced Topics**

**37 Software Process Improvement**

A *minor revision* of SEPA7e, Chapter 30.

**38 Emerging Trends in Software Engineering**

An *extensive revision* of SEPA7e, Chapter 31Presents a discussion of the challenges we face in isolating technology trends and the typical technology evolution life cycle. Software engineering trends are presented.

**39 Concluding Comments**

A *moderate revision* of SEPA, 7/e, Chapter 32.

**1.3 AUDIENCES FOR THE BOOK**

SEPA may be used as a textbook for a course in software engineering for the following audiences:

1. *Undergraduate students* in computer science, engineering and management/information science. Students are expected to have a nontrivial (i.e., considerably more than an introductory programming course) background in software and would generally be at Junior/Senior level. Guidelines for courses for this audience are presented in Segment 4.

2. *Graduate students* in computer science, engineering and other related disciplines. A course for this audience may be tailored to the technical maturity and backgrounds of the students. Alternative modes of presentation are also discussed in Segment 4.

3. *Practicing professionals* from commercial, engineering, scientific and systems environments. A course for this audience (generally an intensive 3 to 5 day format) must be tailored to the technical background and interests of the participants. Alternative modes of presentation are discussed in Segment 5.

**1.4 TEACHING OPTIONS**

Because software engineering is such a broad subject, the instructor of a course may present a balanced treatment of all topics or may choose to emphasize one or more areas. Your background and the needs of your students will dictate this choice.

At the undergraduate computer science and engineering level, greater emphasis may be placed on development issues (analysis, design, coding, testing). For undergraduate courses directed toward Information Technology backgrounds, emphasis may shift to project management, information engineering, QA, and reengineering. Similar choices can be made for graduate courses.

In an academic setting, a software engineering course can be offered as:

1. a lecture, problem, test course;

2. a lecture, project course;

3. a project course (with readings from SEPA);

4. some combination of choices 1 – 3.

For maximum, pedagogical benefit, a course project is generally recommended. Segments 3 and 4 discuss options for course design.

In a continuing education/professional training setting, a software engineering course can be offered as:

1. a lecture, "laboratory" course;

2. a seminar or tutorial;

3. a lecture-project course.

Segment 5 discusses options for course design in this area.

**A note of caution:** You should resist the temptation to offer a classical "advanced programming course" that is called "Software Engineering." Software engineering is much more than advanced programming and unless your course has been explicitly designed to emphasize only one topic area, the course offering should clearly present topics relevant to each activity in the software engineering process. Emphasis may vary, but a presentation that is limited to "algorithm design, data design, programming and debugging" (however worthwhile as an independent course offering) is not an adequate presentation of software engineering.

**1.5 USE OF THIS MANUAL**

The *Instructor's Manual* for SEPA8e has been written to provide you with guidelines for course design, a description of the SEPA website, courseware that you can use to complement SEPA, and an explanation of chapter content, problems and points to ponder, and supplementary references and topics.

Segment 2 describes the SEPA website. Segment 3 discusses the *IEEE/ACM Computing Curriculum* and how SEPA can be used to fulfill many of the "knowledge units" described in the curriculum. Segment 4 provides detailed information for the design of a university level course. Segment 5 provides guidelines for those instructors who are contemplating the development of a software engineering short-course or seminar. Segment 6 provides valuable insight into the content of each chapter and serves as a support tool for those who have adopted SEPA.

2 THE SEPA WEBSITE

We've had a fair amount of experience in teaching software engineering and there are a few things we've learned along the way:

• Software engineering courses are not amenable to white board presentations—too much time is wasted with no benefit.

• Few instructors are expert in all subject areas; another point of view freshens the course.

• The more supplementary material readily available for students, the better the learning experience.

With these thoughts in mind, we decided to enhance and update the already comprehensive resources that were first developed to supplement the 5th through 7th editions of SEPA and to distribute them to instructors and students via the SEPA website.

To be honest, our thoughts while assembling the website for the SEPA8e were not completely altruistic. There are dozens of software engineering books on the market (some good, some not so good) and it's important to consider ways to differentiate SEPA. We are convinced that depth of coverage, style and overall presentation differentiate SEPA in its own right (but it's hard to be objective). However, we honestly do believe that the website for SEPA is far and away the most comprehensive collection of software engineering resources (directed at instructors, students and professionals) available for any book on the subject.

**2.1 INSTRUCTOR RESOURCES**

The SEPA Web Site contains the following resources designed explicitly for Instructors who have adopted the book for classroom use.

• **Powerpoint Slides** - A full set of MS Powerpoint slides may be downloaded to supplement your lecture presentations. The slides are organized by chapter.

IMPORTANT: *Please note that these copyright slides are intended for use ONLY by adopters of SEPA8e and ONLY for classroom use in a university setting. They may NOT be used for short courses or for consulting purposes without express written permission.*

• **Web-based Resources** — The SEPA8e website contains over 1,000 pointers, organized by software engineering topic, to a broad array of software engineering resources. You may use these to prepare lectures and as reference sources for student papers, projects and general research.

• **Reference Library** — The SEPA8e website contains over 600 pointers, organized by chapter topic, to a wide array of downloadable papers (most in PDF format). The reference library enables you to supplement SEPA7e content and provide more depth in topic areas that are only considered superficially. Many of the papers contained in the library would be appropriate for addition reading assignments at the graduate level.

• **Test Bank** — A collection of suggested test questions (with answers) is provided.

*Note: The Student Resources section of the Web site contains a set of student self-tests (multiple choice) that are intended to assist a student in assessing his or her understanding of chapter content. Self-tests (typically 12 - 20 questions) are provided for each chapter. You should encourage your students to use them.*

• **Sample exams** — A collection of sample exams are provided.

• **Tiny Tools** — A collection of simple software engineering tools that serve to implement approaches and techniques presented in SEPA, 6/e is provided.

IMPORTANT: These tools are provided without any warrantee, either express or implied. They are intended for educational purposes only. If your students develop useful tools that you feel might be added to the list of tiny tools, see the Tiny Tools pages for instructions on how to submit them for evaluation and inclusion (if accepted)

• **Page Out** - A tool that enables you to develop your own course specific website.

• **Message Board** - The message board allows communication among instructors who are using SEPA8e for teaching purposes.

**2.2 STUDENT RESOURCES**

The student resources section of the SEPA8e website has been designed to improve the student's learning experience. The following resources are available for student use:

• **Study Guide** - The study guide summarizes key points presented within each SEPA chapter. Be sure to emphasize that it is NOT a substitute for reading the chapter and that many important points contained within the book could not be included in the study guide for space reasons.

• **Self-Tests** – Multiple-choice self-tests allow students to test their knowledge of chapter content. Along with each answer, the self-test provides the section within SEPA where topic discussion may be found.

• **Web-based Resources** - The SEPA8e website contains over 1,000 pointers, organized by chapter, to a broad array of software engineering topics.

• **Reference Library** - The SEPA8e Web site contains over 600 pointers, organized by chapter topic, to a wide array of downloadable papers (most in PDF format).

• **Case Study** - The case study provides running examples of many important software engineering work products. These case studies were developed as part of student projects (blemishes and all). They should provide useful examples for students who are asked to produce similar document as part of their project work.

• **Supplementary Content** - The website contains a collection of supplementary materials that expand on topics presented within the book.

• **Message Board** - The message board allows Q&A among students and other and provides a useful mechanism for informal communication.

**2.3 PROFESSIONAL RESOURCES**

The professional resources section of the SEPA website has been designed to provide software practitioners and their managers with useful software engineering information. It should be noted that most, if not all, of the resources presented in this section can be of use to instructors and students. You should encourage your students to explore this section of the site in some detail.

The following professional resources are provided:

• **Documents/Work Product Templates** - Examples of the format and content of software engineering work products are provided for download.

• **Software Engineering Checklists** - When reviewing or assessing software engineering work products, it's not a bad idea to have a checklist to guide the evaluation. A wide variety of checklists are provided and links to still more are also included.

• **Tiny Tools** - There are lots of little things that lead to success software engineering work. These tools will help with some of them.

• **Professional Tools (CASE)** - The SEPA website provides links to hundreds of CASE tools and more than a few tools comparisons.

• **Software Engineering Resources** - - The SEPA website contains over 1,000 pointers, organized by chapter, to a broad array of software engineering topics.

• **Reference Library** — The SEPA website contains over 600 pointers, organized by chapter topic, to a wide array of downloadable papers (most in PDF format).

• **Adaptable Process Model** - The adaptable process model is a comprehensive process template that can be tuned to the need of a specific organization.

• **Distance Learning Curriculum** - An industry quality distance learning curriculum is available from EdistaLearning. See the SEPA website for details

• **Industry Commentary** - A collection of short essays, extracted from a number of industry sources, are provided to help spur thoughtful debate.

3 SEPA AND THE ACM/IEEE

COMPUTING CURRICULUM

The ACM/IEEE-CS Joint Task Force on Computing Curriculum has produced a lengthy report that introduces curriculum guidelines for software engineering entitled *Software Engineering 2004* (23 August 2004) in the following manner:[[4]](#footnote-4)

The primary purpose of this volume is to provide guidance to academic institutions and accreditation agencies about what should constitute an undergraduate software engineering education. These recommendations have been developed by a broad, internationally-based group of volunteer participants. This group has taken into account much work that has been done in software engineering education over the last quarter of a century. Software engineering curriculum recommendations are of particular relevance, since there is currently a surge in the creation of software engineering degree programs and an accreditation process for such programs has been established in a number of countries. The recommendations included in this volume are …

• SEEK: Software Engineering Education Knowledge - what every SE graduate must know.

• Curriculum: Ways that this knowledge and the skills fundamental to software engineering can be taught in various contexts.

The 8th edition of SEPA has been designed to assist you in presenting many important elements of SEEK.

In addition to **computing essentials** (the computer science foundations that support the design and construction of software products) and **mathematical and engineering fundamentals** (the theoretical and scientific underpinnings for the construction of software products with desired attributes), the report covers a number of important areas of knowledge that should be presented as part of a complete software engineering degree program (extracted from the ACM/IEEE-CS report):

1. **Continue here🡪professional practice**— concerned with the knowledge, skills, and attitudes that software engineers must possess to practice software engineering in a professional, responsible, and ethical manner.

2. **software modeling and analysis**— considered core concepts in any engineering discipline since they are essential to documenting and evaluating design decisions and alternatives.

3. **software design**— concerned with issues, techniques, strategies, representations, and patterns used to determine how to implement a component or a system.

4. **software verification and validation**— uses both static and dynamic techniques of system checking to ensure that the resulting program satisfies its specification and that the program as implemented meets the expectations of the stakeholders.

5. **software evolution**— is the result of the ongoing need to support the stakeholders' mission in the face of changing assumptions, problems, requirements, architectures and technologies.

6. **software process**— is concerned with knowledge about the description of commonly used software life-cycle process models and the contents of institutional process standards; definition, implementation, measurement, management, change and improvement of software processes; and use of a defined process to perform the technical and managerial activities needed for software development and maintenance.

7. **software quality**— Software quality is a pervasive concept that affects, and is affected by all aspects of software development, support, revision, and maintenance. It encompasses the quality of work products developed and/or modified (both intermediate and deliverable work products) and the quality of the work processes used to develop and/or modify the work products.

8. **software management**— Software management is concerned with knowledge about the planning, organization, and monitoring of all software life cycle phases.

In the sections that follow, I discuss how SEPA8e addresses the specific requirements outlined for each of these subject areas. These represent the core information required for an understanding of software engineering.

**3.1 Professional Practice**

*From the CCSE Report:* Professional Practice is concerned with the knowledge, skills, and attitudes that software engineers must possess to practice software engineering in a professional, responsible, and ethical manner. The study of professional practices includes the areas of technical communication, group dynamics and psychology, and social and professional responsibilities.

|  |  |  |
| --- | --- | --- |
| Reference | Topic | SEPA Citation |
| PRF.psy | *Group dynamics / psychology* |  |
| PRF.psy.1 | Dynamics of working in teams/groups | Chapter 6 |
| PRF.psy.2 | Individual cognition (e.g. limits) | not considered |
| PRF.psy.3 | Cognitive problem complexity | Chapter 2 |
| PRF.psy.4 | Interacting with stakeholders | Chapter 8 |
| PRF.psy.5 | Dealing with uncertainty and ambiguity | Chapter 8 |
|  |  |  |
| PRF.com | *Communications skills (specific to SE)* |  |
| PRF.com.1 | Reading, understanding and summarizing reading | not considered |
| PRF.com.2 | Writing | not considered |
| PRF.com.3 | Team and group communication | Chapters 6, 31 |
| PRF.com.4 | Presentation skills | not considered |
|  |  |  |
| PRF.pr | *Professionalism* |  |
| PRF.pr.1 | Accreditation, certification, and licensing | not considered |
| PRF.pr.2 | Codes of ethics and professional conduct | Chapter 39 |
| PRF.pr.3 | Social, legal, historical, and professional issues and concerns | Chapters 1, 37, 38 |
| PRF.pr.4 | The nature of, and role of professional societies | not considered |
| PRF.pr.5 | The nature and role of software engineering standards | throughout SEPA |
| PRF.pr.6 | The economic impact of software | Chapters 1, 39 |

**3.2 Software Modeling and Analysis**

*From the CCSE Report:* Modeling and analysis can be considered core concepts in any engineering discipline since they are essential to documenting and evaluating design decisions and alternatives. Modeling and analysis is first applied to the analysis, specification, and validation of requirements. Requirements represent the real world needs of users, customers and other stakeholders affected by the system and the capabilities and opportunities afforded by software and computing technologies. The construction of requirements includes an analysis of the feasibility of the desired system, elicitation and analysis of stakeholders' needs, the creation of a precise description of what the system should and should not do along with any constraints on its operation and implementation, and the validation of this description or specification by the stakeholders.

|  |  |  |
| --- | --- | --- |
| Reference | Topic | SEPA Citation |
| MAA.md | *Modeling foundations* |  |
|  |  |  |
| MAA.md.1 | Modeling principles (e.g. decomposition, abstraction, generalization, projection/views, explicitness, use of formal approaches, etc.) | Chapters 7, 12 |
|  |  |  |
| MAA.md.2 | Pre & post conditions, invariants | Chapters 14, 28 |
| MAA.md.3 | Introduction to mathematical models and specification languages | Chapter 28 |
| MAA.md.4 | Properties of modeling languages | Chapter 28, App. I |
| MAA.md.5 | Syntax vs. semantics | Chapters 8 – 18, 28 |
| MAA.md.6 | Explicitness | Chapters 8 - 11 |
|  |  |  |
| MAA.tm | *Types of models* |  |
| MAA.tm.1 | Information modeling (e.g. entity-relationship modeling, class diagrams, etc.) | Chapter 10 |
| MAA.tm.2 | Behavioral modeling (e.g. structured analysis, state diagrams, use-cases | Chapter 11 |
| MAA.tm.3 | Structure modeling (e.g. architectural, etc.) | Chapter 13 |
| MAA.tm.4 | Domain modeling | Chapters 10, 16, 12 |
| MAA.tm.5 | Functional modeling | Chapter 14 |
| MAA.tm.6 | Enterprise modeling | not considered |
| MAA.tm.7 | Modeling embedded systems | Chapters 11, 12 |
| MAA.tm.8 | Requirements interaction analysis | not considered |
| MAA.tm.9 | Analysis Patterns | Chapters 11, 16 |
|  |  |  |
| MAA.af | *Analysis fundamentals* |  |
| MAA.af.1 | Analyzing well-formedness (e.g. completeness, consistency, robustness, etc.) | not considered |
| MAA.af.2 | Analyzing correctness (e.g. static analysis, simulation, model, checking, etc. | Chapters 19, 20, 22 - 28 |
| MAA.af.3 | Analyzing quality (non-functional) requirements (e.g. safety, security, usuability, performance, etc. | Chapters 19, 21 |
| MAA.af.4 | Prioritization, trade-off analysis, risk analysis, and impact analysis | Chapters 8, 35 |
| MAA.af.5 | Traceability | Chapters 8, 29 |
| MAA.af.6 | Formal analysis | Chapter 28 |
|  |  |  |
| MAA.rfd | *Requirements fundamentals* 3 |  |
| MAA.rfd.1 | Definition of requirements (e.g. product, project, constraints, system boundary, external, internal, etc.) | Chapters 8 - 11 |
| MAA.rfd.2 | Requirements process | Chapter 8 |
| MAA.rfd.3 | Layers/levels of requirements (e.g. needs, goals, user requirements, system requirements, software requirements, etc.) | Chapters 8 - 11 |
| MAA.rfd.4 | Requirements characteristics (e.g. testable, non-ambiguous, consistent, correct, traceable, priority, etc.) | Chapter 8 |
| MAA.rfd.5 | Managing changing requirements | Chapter 5, 8, 29 |
| MAA.rfd.6 | Requirements management (e.g. consistency management, release planning, reuse, etc.) | Chapters 8 - 11 |
| MAA.rfd.7 | Interaction between requirements and architecture | Chapter 13 |
| MAA.rfd.8 | Relationship of requirements to systems engineering, human-centered design, etc. | Chapters 8 – 12, 15 |
| MAA.rfd.9 | Wicked problems (e.g. ill-structured problems; problems with many solutions; etc.) | only indirectly in SEPA |
| MAA.rfd. | COTS as a constraint | Chapter 14 |
|  |  |  |
| MAA.er | *Eliciting requirements* 4 |  |
| MAA.er.1 | Elicitation Sources (e.g. stakeholders, domain experts, operational and organization environments, etc.) | Chapter 8 |
| MAA.er.2 | Elicitation Techniques (e.g. interviews, questionnaires/surveys, prototypes, use cases, observation, participatory techniques, | Chapter 8 |
| etc.) |  |  |
| MAA.er.3 | Advanced techniques (e.g. ethnographic, knowledge elicitation, etc.) | not considered |
|  |  |  |
| MAA.rsd | *Requirements specification & documentation* |  |
| MAA.rsd.1 | Requirements documentation basics (e.g. types, audience, structure, quality, attributes, standards, etc.) | Chapter 8 - 11 |
| MAA.rsd.2 | Software requirements specification | Chapters 9 - 11 |
| MAA.rsd.3 | Specification languages (e.g. structured English, UML, formal languages such as Z, VDM, SCR, RSML, etc.) | Chapters 14, 28, App. II |
|  |  |  |
| MAA.rv | *Requirements validation* 3 |  |
| MAA.rv.1 | Reviews and inspection | Chapter 20 |
| MAA.rv.2 | Prototyping to validate requirements | Chapters 3, 4, 8 |
| MAA.rv.3 | Acceptance test design | Chapter 22 |
| MAA.rv.4 | Validating product quality attributes | Chapters 19 – 28 |
| MAA.rv.5 | Formal requirements analysis | Chapter 28 |

**3.3 Software Design**

*From the CCSE Report:* Software design is concerned with issues, techniques, strategies, representations, and patterns used to determine how to implement a component or a system. The design will conform to functional requirements within the constraints imposed by other requirements such as resource, performance, reliability, and security. This area also includes specification of internal interfaces among software components, architectural design, data design, user interface design, design tools, and the evaluation of design.

|  |  |  |
| --- | --- | --- |
| **Reference** | **Topic** | **SEPA Citation** |
| DES.con | *Design concepts* 3 |  |
| DES.con.1 | Definition of design | Chapter 12 |
| DES.con.2 | Fundamental design issues (e.g. persistent data, storage management, exceptions, etc.) | Chapter 12 |
| DES.con.3 | Context of design within multiple software development life cycles | Chapters 3 5, 128 |
| DES.con.4 | Design principles (information hiding, cohesion and coupling) | Chapter 8 |
| DES.con.5 | Interactions between design and requirements | Chapters 3-4, 9 - 11, 12 |
| DES.con.6 | Design for quality attributes (e.g. reliability, usability, performance, testability, fault tolerance, etc.) | Chapter 12 |
| DES.con.7 | Design trade-offs | Chapters 13 - 18 |
| DES.con.8 | Architectural styles, patterns, reuse | Chapters 13, 16 |
|  |  |  |
| DES.str | *Design strategies* |  |
| DES.str.1 | Function-oriented design | Chapter 14 |
| DES.str.2 | Object-oriented design | Chapters 13 - 18 |
| DES.str.3 | Data-structure centered design | Chapter 14 |
| DES.str.4 | Aspect oriented design | Chapter 3-4 |
|  |  |  |
| DES.ar | *Architectural design* 9 |  |
| DES.ar.1 | Architectural styles (e.g. pipe-and-filter, layered, transaction-centered, peer-to-peer, publish-subscribe, event-based, client/server, etc.) | Chapter 13 |
| DES.ar.2 | Architectural trade-offs between various attributes | Chapter 13 |
| DES.ar.3 | Hardware issues in software architecture | only indirectly in SEPA |
| DES.ar.4 | Requirements traceability in architecture | Chapter 13 |
| DES.ar.5 | Domain-specific architectures and product-lines | Chapter 13 |
| DES.ar.6 | Architectural notations (e.g. architectural structure viewpoints & representations, component diagrams, etc.) | Chapter 13 |
|  |  |  |
| DES.hci | *Human computer interface design* |  |
| DES.hci.1 | General HCI design principles | Chapter 15 |
| DES.hci.2 | Use of modes, navigation | Chapters 15, 17, 18 |
| DES.hci.3 | Coding techniques and visual design (e.g. color, icons, fonts, etc.) | Chapters 15, 17, 18 |
| DES.hci.4 | Response time and feedback | Chapter 15 |
| DES.hci.5 | Design modalities (e.g. menu-driven, forms, question-answering, etc.) | Chapters 15, 17, 18 |
| DES.hci.6 | Localization and internationalization | Chapters 15, 17. 18 |
| DES.hci.7 | Human computer interface design methods | Chapter 15 |
| DES.hci.8 | Multi-media (e.g. I/O techniques, voice, natural language, webpage, sound, etc.) | Chapters 15, 17, 18 |
| DES.hci.9 | Metaphors and conceptual models | Chapter 15 |
| DES.hci.10 | Psychology of HCI | Chapter 15 |
|  |  |  |
| DES.dd | *Detailed design* |  |
| DES.dd.1 | One selected design method | Chapter 14 |
| DES.dd.2 | Design patterns | Chapter 16 |
| DES.dd.3 | Component design | Chapter 15 |
| DES.dd.4 | Component and system interface design | Chapters 14, 15 |
| DES.dd.5 | Design notations (e.g. class and object diagrams, UML, state diagrams, etc.) | Chapters 12 – 18, App. I |
|  |  |  |
| DES.ste | *Design support tools and evaluation* 3 |  |
| DES.ste.1 | Design support tools (e.g. architectural, static analysis, dynamic evaluation, etc.) | Chapters 12 - 18 |
| DES.ste.2 | Measures of design attributes (e.g. coupling, cohesion, information-hiding, separation of concerns, etc.) | Chapter 30 |
| DES.ste.3 | Design metrics (e.g. architectural factors, interpretation, metric sets in common use, etc.) | Chapter 30 |
| DES.ste.4 | Formal design analysis | Chapter 28 |

**3.4 Software Verification and Validation**

*From the CCSE Report*: Software verification and validation uses both static and dynamic techniques of system checking to ensure that the resulting program satisfies its specification and that the program as implemented meets the expectations of the stakeholders. Static techniques are concerned with the analysis and checking of system representations throughout all stages of the software life cycle while dynamic techniques involve only the implemented system.

|  |  |  |
| --- | --- | --- |
| **Reference** | **Topic** | **SEPA Citation** |
| VAV.fnd | *V&V terminology and foundations* |  |
| VAV.fnd.1 | Objectives and constraints of V&V | Chapters 19, 21 |
| VAV.fnd.2 | Planning the V&V effort | Chapter 21 |
| VAV.fnd.3 | Documenting V&V strategy, including tests and other artifacts | Chapter 21 |
| VAV.fnd.4 | Metrics & Measurement (e.g. reliability, usability, performance,etc.) | Chapters 30, 32 |
| VAV.fnd.5 | V&V involvement at different points in the lifecycle | Chapters 3 - 5, 19 |
|  |  |  |
| VAV.rev | *Review* |  |
| VAV.rev.1 | Desk checking | Chapter 20 |
| VAV.rev.2 | Walkthroughs | Chapter 20 |
| VAV.rev.3 | Inspections | Chapter 20 |
|  |  |  |
| VAV.tst | *Testing* 21 |  |
| VAV.tst.1 | Unit testing | Chapters 22, 23 |
| VAV.tst.2 | Exception handling (writing test cases to trigger exception handling; designing good handling) | Chapters 23, 25 - 27 |
| VAV.tst.3 | Coverage analysis (e.g. statement, branch, basis path, multi-condition, dataflow, etc.) | Chapter 23 |
| VAV.tst.4 | Black-box functional testing techniques | Chapter 23 |
| VAV.tst.5 | Integration Testing | Chapters, 22, 23 |
| VAV.tst.6 | Developing test cases based on use cases and/or customer stories | Chapters 22 - 27 |
| VAV.tst.7 | Operational profile-based testing | Section 22 - 27 |
| VAV.tst.8 | System and acceptance testing | Section 22 |
| VAV.tst.9 | Testing across quality attributes (e.g. usability, security, compatibility, accessibility, etc.) | Chapters 19, 21 |
| VAV.tst.10 | Regression Testing | Chapter 22 |
| VAV.tst.11 | Testing tools | Section 22 - 27 |
| VAV.tst.12 | Deployment process | throughout SEPA |
|  |  |  |
| VAV.hct | *Human computer user interface testing and evaluation* |  |
| VAV.hct.1 | The variety of aspects of usefulness and usability | Chapters 15, 24, 25, 26 |
| VAV.hct.2 | Heuristic evaluation | Chapter 15 |
| VAV.hct.3 | Cognitive walkthroughs | Chapter 15, 24 |
| VAV.hct.4 | User testing approaches (observation sessions etc.) | Chapter 15, 24 |
| VAV.hct.5 | Web usability; testing techniques for web sites | Chapter 17 |
| VAV.hct.6 | Formal experiments to test hypotheses about specific HCI controls | not considered |
|  |  |  |
| VAV.par | *Problem analysis and reporting* 4 |  |
| VAV.par.1 | Analyzing failure reports | Chapter 21 |
| VAV.par.2 | Debugging/fault isolation techniques | Chapter 23 |
| VAV.par.3 | Defect analysis | Chapter 22 |
| VAV.par.4 | Problem tracking | Chapter 21 |

**3.5 Software Evolution**

*From the CCSE Report:* Software evolution is the result of the ongoing need to support the stakeholders' mission in the face of changing assumptions, problems, requirements, architectures and technologies. It is intrinsic to all real world software systems. Support for evolution requires numerous activities both before and after each of a succession of versions or upgrades (releases) that constitute the evolving system. Evolution is a broad concept that expands upon the traditional notion of software maintenance.

|  |  |  |
| --- | --- | --- |
| **Reference** | **Topic** | **SEPA Citation** |
| EVO.pro | *Evolution processes* 6 |  |
| EVO.pro.1 | Basic concepts of evolution and maintenance | Chapter 2, 36 |
| EVO.pro.2 | Relationship between evolving entities (e.g. assumptions, requirements, architecture, design, code, etc.) | Chapter 29 |
| EVO.pro.3 | Models of software evolution (e.g. theories, laws, etc.) | Chapter 36 |
| EVO.pro.4 | Cost models of evolution | Chapter 33 |
| EVO.pro.5 | Planning for evolution (e.g. outsourcing, in-house, etc.) | Chapter 29, 36 |
|  |  |  |
| EVO.ac | *Evolution activities* |  |
| EVO.ac.1 | Working with legacy systems (e.g. use of wrappers, etc.) | Chapter 36 |
| EVO.ac.2 | Program comprehension and reverse engineering | Chapter 36 |
| EVO.ac.3 | System and process re-engineering (technical and business) | Chapter 36 |
| EVO.ac.4 | Impact analysis |  |
| EVO.ac.5 | Migration (technical and business) | Chapter 36 |
| EVO.ac.6 | Refactoring | Chapter 5 |
| EVO.ac.7 | Program transformation | Chapter 36 |
| EVO.ac.8 | Data reverse engineering | Chapter 36 |
|  |  |  |

**3.6 Software Process**

*From the CCSE Report:* Software process is concerned with knowledge about the description of commonly used software life-cycle process models and the contents of institutional process standards; definition, implementation, measurement, management, change and improvement of software processes; and use of a defined process to perform the technical and managerial activities needed for software development and maintenance.

|  |  |  |
| --- | --- | --- |
| **Reference** | **Topic** | **SEPA Citation** |
| PRO.con | Process concepts |  |
| PRO.con.1 | Themes and terminology | Chapters 2, 3, 30 |
| PRO.con.2 | Software engineering process infrastructure (e.g. personnel, tools, training, etc.) | Chapters 3 - 5, 37 |
| k E |  |  |
| PRO.con.3 | Modeling and specification of software processes | Chapters 3 - 5, 37 |
| PRO.con.4 | Measurement and analysis of software processes | Chapters 32, 37 |
| PRO.con.5 | Software engineering process improvement (individual, team) | Chapter 37 |
| PRO.con.6 | Quality analysis and control (e.g. defect prevention, review processes, quality metrics, root cause analysis, etc.) | Chapters 21, 37 |
| PRO.con.7 | Analysis and modeling of software process models | Chapter 3 - 5, 37 |
|  |  |  |
| PRO.imp | *Process implementation* |  |
| PRO.imp.1 | Levels of process definition (e.g. organization, project, team, individual, etc.) | Chapter 37 |
| k E |  |  |
| PRO.imp.2 | Life cycle models (agile, heavyweight, waterfall, spiral, etc.) | Chapters 3 - 5 |
| PRO.imp.3 | Life cycle process models and standards (e.g., IEEE, ISO, etc.) | Chapter 3 - 4 |
| PRO.imp.4 | Individual software process (model, definition, measurement, analysis, improvement) | Section 37 |
| PRO.imp.5 | Team software process (model, definition, organization, measurement, analysis, improvement) | Section 3 - 4 |
| PRO.imp.6 | Process tailoring | Chapter 37 |
| PRO.imp.7 | ISO/IEEE Standard 12207: requirements of processes | Chapters 3 - 4, 37 |

**3.7 Software Quality**

*From the CCSE Report:* Software quality is a pervasive concept that affects, and is affected by all aspects of software development, support, revision, and maintenance. It encompasses the quality of work products developed and/or modified (both intermediate and deliverable work products) and the quality of the work processes used to develop and/or modify the work products. Quality work product attributes include usability, reliability, safety, security, maintainability, flexibility, efficiency, performance and availability.

|  |  |  |
| --- | --- | --- |
| **Reference** | **Topic** | **SEPA Citation** |
| QUA.cc | *Software quality concepts and culture* |  |
| QUA.cc.1 | Definitions of quality | Chapter 19 |
| QUA.cc.2 | Society's concern for quality | Chapters 1, 19 |
| QUA.cc.3 | The costs and impacts of bad quality | Chapter 19 |
| QUA.cc.4 | A cost of quality model | Chapters 19, 21 |
| QUA.cc.5 | Quality attributes for software (e.g. dependability, usability, etc.) | Chapters 12, 19 |
| QUA.cc.6 | The dimensions of quality engineering | Chapter 21 |
| QUA.cc.7 | Roles of people, processes, methods, tools, and technology | Chapter 21 |
|  |  |  |
| QUA.std | *Software quality standards* |  |
| QUA.std.1 | The ISO 9000 series | Chapter 21 |
| QUA.std.2 | ISO/IEEE Standard 12207: the "umbrella" standard | Chapter 37 |
| QUA.std.3 | Organizational implementation of standards | Chapter 37 |
| QUA.std.4 | IEEE software quality-related standards | Chapter 21 |
|  |  |  |
| QUA.pro | *Software quality processes* |  |
| QUA.pro.1 | Software quality models and metrics | Chapters 19, 30, 32 |
| QUA.pro.2 | Quality-related aspects of software process models | Chapters 3 - 5, 37 |
| QUA.pro.3 | Introduction/overview of ISO 15504 and the SEI CMMs | Chapter 37 |
| QUA.pro.4 | Quality-related process areas of ISO 15504 | not considered |
| QUA.pro.5 | Quality-related process areas of the SW-CMM and the CMM | Chapter 37 |
| QUA.pro.6 | The Baldridge Award criteria for software engineering | not considered |
| QUA.pro.7 | Quality aspects of other process models | Chapters 3 - 5, 37 |
|  |  |  |
| QUA.pca | *Process assurance* 4 |  |
| QUA.pca.1 | The nature of process assurance | Chapter 37 |
| QUA.pca.2 | Quality planning | Chapter 21 |
| QUA.pca.3 | Organizing and reporting for process assurance | Chapters 21, 37 |
| QUA.pca.4 | Techniques of process assurance | Chapter 21, 37 |
|  |  |  |
| QUA.pda | *Product assurance* |  |
| QUA.pda.1 | The nature of product assurance | Chapters 19, 21 |
| QUA.pda.2 | Distinctions between assurance and V&V | Chapter 21 |
| QUA.pda.3 | Quality product models | Chapter 30 |
| QUA.pda.4 | Root cause analysis and defect prevention | Chapter 21 |
| QUA.pda.5 | Quality product metrics and measurement | Chapter 30 |
| QUA.pda.6 | Assessment of product quality attributes (e.g. useability, reliability, availability, etc.) | Chapters 19, 21 |

**3.8 Software Management**

*From the CCSE Report:* Software management is concerned with knowledge about the planning, organization, and monitoring of all software life cycle phases. Management is critical to ensure that software development projects are appropriate to an organization, work in different organizational units is coordinated, software versions and configurations are maintained, resources are available when necessary, project work is divided appropriately, communication is facilitated, and progress is accurately charted.

|  |  |  |
| --- | --- | --- |
| **Reference** | **Topic** | **SEPA Citation** |
| MGT.con | *Management concepts* |  |
| MGT.con.1 | General project management | Chapter 31 |
| MGT.con.2 | Classic management models | Chapter 31 |
| MGT.con.3 | Project management roles | Chapter 31 |
| MGT.con.4 | Enterprise/Organizational management structure | not considered |
| MGT.con.5 | Software management types (e.g. acquisition, project, development, maintenance, risk, etc.) | Chapter 31 |
|  |  |  |
| MGT.pp | *Project planning* |  |
| MGT.pp.1 | Evaluation and planning | Chapters 31, 33, 34 |
| MGT.pp.2 | Work breakdown structure | Chapter 34 |
| MGT.pp.3 | Task scheduling | Chapter 34 |
| MGT.pp.4 | Effort estimation | Chapter 33 |
| MGT.pp.5 | Resource allocation | Chapters 33, 34 |
| MGT.pp.6 | Risk management | Chapter 35 |
|  |  |  |
| MGT.per | *Project personnel and organization* |  |
| MGT.per.1 | Organizational structures, positions, responsibilities, and authority | Chapter 31 |
| MGT.per.2 | Formal/informal communication | Chapter 31 |
| MGT.per.3 | Project staffing | Chapter 31, 33, 34 |
| MGT.per.4 | Personnel training, career development, and evaluation | not considered |
| MGT.per.5 | Meeting management | Chapters 5, 20 |
| MGT.per.6 | Building and motivating teams | Chapters 5, 31 |
| MGT.per.7 | Conflict resolution | Chapter 8 |
|  |  |  |
| MGT.ctl | *Project control* |  |
| MGT.ctl.1 | Change control | Chapter 29 |
| MGT.ctl.2 | Monitoring and reporting | Chapter 29 |
| MGT.ctl.3 | Measurement and analysis of results | Chapters 32, 33 |
| MGT.ctl.4 | Correction and recovery | Chapter 29 |
| MGT.ctl.5 | Reward and discipline | not considered |
| MGT.ctl.6 | Standards of performance | Chapters 21, 29 |
|  |  |  |
| MGT.cm | *Software configuration management* 5 |  |
| MGT.cm.1 | Revision control | Chapter 29 |
| MGT.cm.2 | Release management | Chapter 29 |
| MGT.cm.3 | Tool support | Chapter 29 |
| MGT.cm.4 | Builds | Chapter 29 |
| MGT.cm.5 | Software configuration management processes | Chapter 29 |
| MGT.cm.6 | Maintenance issues | Chapter 1 - 2, 36 |
| MGT.cm.7 | Distribution and backup | Chapter 29 |

It should be noted that the ACM/IEEE-CS Joint Task Force on Computing Curriculum Repo7e, 6/e addresses a variety of important Web engineering topics in considerable detail.

4 GUIDELINES FOR UNIVERSITY COURSES

Courses in software engineering at both graduate and undergraduate levels are a necessary element of computer science and computer engineering curricula. In segment 3, the current ACM/IEEE Computing Curriculum was presented and an indication of how SEPA fits into it was provided.

When the first edition of SEPA was written (1981) few institutions offered software engineering courses. Today, things are different. Most schools offer at least one course in software engineering, and it may be that the design of your course is fixed. If so, the content of this segment of the *Instructor's Manual* may be of little interest. However, if you are considering the redesign of your course, or are just starting one, or have the freedom to conduct the course as you see fit, the content of this segment may help.

In 1981, Elaine Kant (*ACM Software Engineering Notes,* August, 1981, vol. 6, no. 4, pp. 52-76) of Carnegie Mellon University published a detailed discussion of software engineering course design. Professor Kant gave me (Roger Pressman) permission to include her paper in the *Instructor's Manual* for the first edition of SEPA and it has been included in every subsequent edition. Kant's paper is pragmatic and based on actual experience. Although it is now almost 30 years old, it remains one of the best discussions of software engineering course design that we have encountered. Undoubtedly, some of the concepts Kant notes must be updated for current technology and modern students, but the overall recommendations are as valid today as they were when the paper was originally written.

The Kant paper is reproduced in Appendix I with permission.

**4.1 COURSE DESIGNS USING SEPA**

The 8th edition of SEPA has been designed to accommodate a number of different approaches for teaching software engineering. The book has been partitioned in a manner that will enable you to use it for courses with the following emphasis:

 a general course in software engineering

• conventional emphasis

• object-oriented emphasis

 a course that emphasizes methods

 an analysis-oriented course

* a design-oriented course
* a Web engineering course

 a management-oriented course

 an advanced topics course

In the sections that follow, course schedules, readings and other suggestions are provided for each of these courses.

**4.2 A General Course in Software Engineering**

A general course in software engineering is what I call a "soup to nuts" course. It introduces the student to all important aspects of the discipline, but of necessity, does not go into depth in every one. The schedule that follows is based on a 14 week term and assumes the use of SEPA as a required course text. The chapters of SEPA follow the lecture presentations closely; reading assignments from SEPA and other books are implied by the lecture topic; homework problems, if assigned, may be found at the conclusion of each chapter in SEPA.

Project deliverables should be distributed evenly throughout the term. In-class reviews are desirable but time pressure may cause problems if they are conducted.

**Week 1 The Product and Process**

An overview of software engineering

The Software Process

Discussion of project requirements and topics

*Readings:* Chapters 1 -5.

**Week 2 Project Planning and Organization**

Project planning issues

Software estimation techniques

Risk analysis

Project Abstract due

*Readings:* Chapters 31, 34, 35

**Week 3 Software Engineering Practice**

*Readings:*  Chapters 6, 7

**Week 4 Requirements Engineering**

*Readings:* Chapters 8

**Week 5 Analysis Methods**

Elements of the analysis model

UML notation

*Readings:* Chapters 9 - 11

**Week 6 Elements of software design**

Fundamental concepts

The design model

*Readings:* Chapter 12

**Week 7 Mid term exam**

Requirements specification due

**Week 8 Design Methods - I**

Architectural design

*Readings:* Chapter 13, 17, 18

**Week 9 Design Methods - II**

Component and interface design

*Readings:* Chapters 14, 15

**Week 10 Testing Strategies**

Test Specification due

*Readings:* Chapters 20, 22

**Week 11 Testing Methods**

White-Box and Black-box testing methods

*Readings:*  Chapter 23 - 26

**Week 12 Umbrella Activities**

*Readings:* Chapters 19, 21, 29, 36

**Week 13 Advanced topics**

*Readings:* Selections from Part 5

**Week 14 Project presentations**

All remaining documentation

Source listing due

A 2 - hour final exam is given as part of the course described above. Homework is suggested, but not graded.

The major challenge in any software engineering course is coordination of lecture material with knowledge required to complete project deliverables. Colleagues at other institutions have gone so far as to require that the entire SEPA text be read within the first month of the course (a bit much, we think) so that students will have been introduced to all necessary techniques and methods.

**4.3 A Software Engineering Course Emphasizing Methods**

A course of this type emphasizes methods to the exclusion of other important aspects of the discipline. We would strongly recommend that you assign readings in other areas (e.g., SQA, maintenance, SCM) even if you do not cover these topics in lecture. It's not a bad idea to test on these topics and to grade adequacy in methods using the course project.

**Week 1 Software Process and Practice**

Generic, prescriptive and agile process models

Software engineering principles and concepts

*Readings:* Chapters 1 – 7

**Week 2 Requirements Engineering**

*Readings:* Chapters 8

**Week 3 Analysis Modeling – I**

Scenarios, Data modeling, Analysis Classes

*Readings:* Chapter 9, 10

**Week 4 Analysis Modeling - II**

Behavioral modeling, patterns

*Readings:* Chapter 11

**Week 5 Design Concepts**

*Readings:* Chapter 12

**Week 6 Patterns**

*Readings:* Chapter 16

**Week 7 Mid term exam**

Analysis model due

**Week 8 Design Methods - I**

Architectural design

*Readings:* Chapter 13, 17, 18

**Week 9 Design Methods - II**

Component design

*Readings:* Chapter 14

**Week 10 Design Methods - III**

User interface design, patterns

*Readings:* Chapter 15, 16

**Week 11 Test Case Design - I**

Strategy and Conventional methods

*Readings:* Chapters 22, 23

**Week 12 Test Case Design - II**

OO methods methods

*Readings:* Chapters 24 - 26

**Week 13 Umbrella Topics**

*Readings:* Chapters 19, 20, 21, 29, 37

**Week 14 Project Presentations**

All remaining documentation

source listing due

A 2-hour final exam is given as part of the course described above. Homework is probably not in order because of time pressure. However, selected problems may be assigned to emphasize important points.

**4.4 An Analysis-Oriented Course**

The primary focus of this course is to cover analysis methods within the context of the software engineering process. A term project is assigned, but the deliverable is a complete software requirements specification and or prototypes that is considerably more detailed (and hopefully, higher quality) than those required in the course already described. A course of this type emphasizes analysis to the exclusion of other important aspects of the discipline. We would strongly recommend that you assign readings in other areas, even if you do not cover other topics in lecture.

SEPA is adequate as the only textbook for an undergraduate course of this type. However, a graduate offering should supplement SEPA with other readings from books dedicated to the subject of analysis. See *Further Readings and Other Information Sources* for suggestions.

**Week 1 Software Process and Practice**

Generic, prescriptive and agile process models

Software engineering principles and concepts

*Readings:* Chapters 1 – 7

**Week 2 Requirements Engineering**

*Readings:* Chapter 8

**Week 3 Analysis Modeling – I**

Use Cases

*Readings:* Chapters 9, 10

**Week 4 Analysis Modeling – II**

Information modeling

*Readings:* Chapter 9, 10

**Week 5 Analysis Modeling – III**

Class-based modeling

*Readings:* Chapter 10

**Week 6 Building a requirements model**

Class-based modeling

*Readings:* Chapter 10

**Week 7 Analysis Modeling – IV**

Class modeling

*Readings:* Chapter 9, 10

**Week 8 Mid-Term exam**

**Week 9 Analysis Modeling - V**

Behavioral modeling

*Readings:* Chapter 11

**Week 10 WebApp Analysis**

*Readings:* Chapter 11

**Week 11 Formal Methods of Specification**

*Readings:* Chapter 28

**Week 12 The Transition to Design**

*Readings:*  Chapter 128

**Week 13 The Transition to Testing**

*Readings:*  Chapter 17

**Week 14** **Software requirements model due**

Presentations

A 2-hour final exam is given as part of the course described above. Homework can be assigned to emphasize important points.

**4.5 A Design-Oriented Course**

The primary focus of this course is to cover design methods within the context of the software engineering process. A term project is assigned, but the deliverable is a complete software requirements and design specification that is considerably more detailed (and hopefully, higher quality) than those required in the courses 4.2.1 and 4.3 A course of this type emphasizes design (and to some extent analysis) to the exclusion of other important aspects of the discipline. We would strongly recommend that you assign readings in other areas, even if you do not cover other topics in lecture.

SEPA is adequate as the only textbook for an undergraduate course of this type. However, a graduate offering should supplement SEPA with other readings from books dedicated to the subject of design. See *Further Readings and Other Information Sources* for suggestions.

**Week 1 Software Process and Practice**

Generic, prescriptive and agile process models

Software engineering principles and concepts

*Readings:* Chapters 1 – 5

**Week 2 Practice and Quality Concepts**

*Readings:* Chapter 6, 7, 19

**Week 3 Requirements Engineering - I**

*Readings:* Chapters 8 - 11

**Week 4 Requirements Engineering - II**

*Readings:* Chapters 8 - 11

**Week 6 Design Engineering**

Fundamental concepts

The design pyramid

*Readings:* Chapter 12

**Week 7 Design Methods - I**

Architectural design

*Readings:* Chapter 13

**Week 8 Mid term exam**

**Week 9 Design Methods - II**

Component design

*Readings:* Chapter 14

**Week 10 Design Methods - III**

User interface design

Pattern-based design

*Readings:* Chapter 15, 16

**Week 11 Web and Mobile Design**

*Readings:* Chapter 17, 18

**Week 12 Complementary topics**

*Presentation focuses on the relationship of the topics to design.*

SQA—an overview

Testing—an overview

Test case design

*Readings:* SEPA, Part 3

**Week 13 Complementary topics**

*Presentation focuses on the relationship of the topics to design.*

Maintenance and Reengineering

*Readings:*  Chapters 36

**Week 14 Design project due**

Presentations

A 2-hour final exam is given as part of the course described above. Homework can be assigned to emphasize important points.

**4.6 A Web Engineering Course**

This course covers all aspects of Web engineering. A term project is assigned. Project deliverables should be distributed evenly throughout the term. In-class reviews are desirable but time pressure may cause problems if they are conducted.

SEPA can be used as the only textbook for an undergraduate course of this type. However, David Lowe and I (Roger Pressman) have written a book dedicated to the subject

Pressman, R. and D. Lowe, *Web Engineering: A Practitioner’s Approach,* McGraw-Hill, 2008.

A comprehensive Instructor’s manual is provided with *Web Engineering.* It includes a variety of course outlines and alternatives.

**4.7 A Management-Oriented Course**

Courses of this type are more likely to be offered in information systems or business programs that have a software engineering focus. Although we believe a course such as this would be important in a computer-science or computer engineering curriculum, the realities of an already overloaded curriculum preclude this.

The primary focus of this course is to cover management methods within the context of the software engineering process. A term project is assigned, but the deliverable is a project plan that is considerably more detailed (and hopefully, higher quality) than those required in the courses 4.2 and 4.3. A course of this type emphasizes planning, estimation, scheduling, risk analysis and other project management activities to the exclusion of other important aspects of the discipline. We would strongly recommend that you assign readings in other areas, even if you do not cover other topics in lecture.

Ideally, much of the project planning work that is part of the term project should be performed using a tool such as Microsoft *Project.*

SEPA is adequate as the only textbook for an undergraduate course of this type. However, a graduate offering should supplement SEPA with other readings from books dedicated to the subject of design.

**Week 1 Software and Software Engineering**

*Readings:* Chapters 1 - 2

**Week 2 The Software Engineering Process**

*Readings:* Chapters 3 - 5

**Week 3 Software Project Management Concepts**

Human aspects of software engineering

The management paradigm

Process and project measurement

*Readings:* Chapters 6, 31, 32

**Week 4 Project Planning**

Estimation techniques

Examples and exercises

*Readings:* Chapter 33

**Week 5 Risk Analysis**

Identification

Projection

Management and monitoring

*Readings:*  Chapter 35

**Week 6 Scheduling and Tracking**

Task networks and resource allocation

Acquisition

*Readings:*  Chapter 34

**Week 7 Maintenance and reengineering**

*Readings:*  Chapter 36

**Week 8 Mid term exam**

**Week 9 Software Quality Assurance**

Statistical SQA

Formal technical reviews

SQA management

*Readings:* Chapters 19- 21

**Week 10 Software Configuration Management**

Change and related management issues

SCM tasks

The impact of the project database

*Readings:*  Chapters 29

**Week 11 Analysis, Design and Testing Fundaments**

*Readings:*  Chapters 8, 12, 22

**Week 12 Software Process Improvement**

*Readings:*  Chapter 37

**Week 13 Advanced Topics**

*Readings:*  Chapter 38

**Week 14 Management presentations**

Project Plan due

A 2-hour final exam is given as part of the course described above. Homework can be assigned to emphasize important points.

**4.8 An Advanced Topics Course**

The primary focus of this course is to consider advanced topics in the context of the software engineering process. In general, such courses are offered at the graduate level. Reading in SEPA should be supplemented with other readings from books suggested in the *Further Readings and Other Information Sources* sections of chapters in SEPA Part V.

**Week 1 The Software Process**

*Readings:* Chapters 1 - 5

**Week 2 Software Engineering Practice**

*Readings:* Chapters 4

**Week 3 Advanced Management Topics**

*Readings:* SEPA Part 4

**Week 4 Requirements Management Topics**

*Readings:* Chapters 5 -

**Week 5 Design Topics - I**

*Readings:* Chapters 12 - 18

**Week 6 Design Topics - II**

*Readings:* Chapter 12 - 18

**Week 7 Quality management - I**

*Readings:* Part 3

**Week 8 Mid-term exam**

**Week 9 Quality Management - II**

*Readings:* Part 3

**Week 10 Advanced Topics**

*Readings:* Part 5

**Weeks 11 - 14 Advanced topics chosen by the instructor**

*Readings:*  from SEPA and supplementary sources

A 2-hour final exam is given as part of the course described above. Homework can be assigned to emphasize important points.

5 GUIDELINES FOR SHORT COURSES & SEMINARS

Over the past 30 years, we have had the opportunity to conduct short courses and seminars in software engineering that have been attended by over 25,000 software practitioners and managers. To be effective, a short course in software engineering must have the following attributes:

**Focus.** The course tone and presentation must be tuned to the audience; i.e., a course for managers must stress different topics than a course for technical practitioners.

**Practicality.** Although it is important to note research directions and assess their long-term impact, most short course participants are interested in practical tools and techniques that are viable *now.*

**Content.** All phases of the software engineering process should be presented, even if different emphasis is applied for different audiences.

**Problem solving.** Students should participate in "laboratory sessions" in which limited problem solving is used to reinforce concepts introduced during lectures.

**Supplementary materials.** One or more books (e.g. SEPA) should be used to reinforce course materials and provide depth that is otherwise impossible to include in a 3, 4 or 5-day short course.

Many universities and private organizations sponsor two to five day public seminars/courses in software engineering and related topics. In fact, this can be an extremely effective way for your department to establish better ties to local industry and to raise (what can amount to) substantial revenue for yourself and your department or institution. Many larger institutions direct market their short courses on a national basis. Today, an increasing number of Web-based course offerings are available.

SEPA can provide a foundation for software engineering training that can be tailored to specific market requirements and application areas.

When software engineering discipline taught in an industry short course setting, three specific audiences can be addressed:

1. Business managers

2. Technical managers

3. Practitioners

Business managers should be introduced to software engineering and to the importance of software as a critical system element for all aspects of business in the 21st century. Their decisions will affect the manner in which software engineering methods are applied within a company. Technical managers (and their counterparts in other engineering and customer organizations) must understand how to plan and control software development. Practitioners must be provided with a suite of methods and tools that may be applied during the definition, development and maintenance phases.

On the pages that follow, we have included outlines for a number of courses that are appropriate for one or more of the constituencies noted above. Each of these courses is supported by extensive courseware. Each is supplemented with SEPA.

**5.1 A SHORT COURSE: MANAGING SOFTWARE PROJECTS**

*Audience:* Managers and senior technical staff who must plan, control and track software development projects.

*Duration:* 3 days (an abbreviated version can be offered in two days)

*Abstract:* This course introduces modern software project management practices in the context of software engineering technology. The course has been designed to cover those “key process areas” that are essential for achieving SEI level 2 process maturity. Effective paradigms for software engineering are introduced; an brief overview of modern software engineering techniques is provided; risk management techniques are discussed; estimation and scheduling methods are considered; and tracking and control mechanisms are presented; management aspects of software quality assurance and software configuration management are described. Laboratory sessions will be conducted to help reinforce important concepts.

*Questions to be answered:*

• What are the key challenges for software project managers?

• What is the current state of software engineering practice in the industry?

• How do I establish scope?

• How do I develop a meaningful effort and time estimates?

• How do I assess risk at the beginning of a project?

• What can I do to manage and reduce risk?

• How can I control my projects more effectively?

• How will a software quality assurance program help?

• Is there a way to control and monitor change?

• How do I track project status?

*Outline:*

**Software & Software Engineering**

The management challenge

What is software?

Measuring the process and the product

**Software Engineering: Management Overview**

The elements of the technology

Process models for software engineering

Methods and tools

**Software Project Management**

The nature of software projects

Why projects fail?

Basic elements of project management

Project planning

Project control

Project tracking

**Software Project Planning**

Project scoping

Cost/effort estimation

The impact of software metrics

Conventional methods

Empirical models

Risk management

Risk assessment

Risk control

Documenting and managing risk

Creating a project schedule

Work breakdown structure

Establishing milestones

Using CPM and related tools

Organizing people for the project

One person projects

Small teams

Large projects

Creating a project plan

A typical plan

Reviewing the plan

**Managing Subcontractors**

Defining requirements

The specification activity

Requirements management

Ensuring the quality will be built

Evaluating subcontractor candidates

Determining progress toward completion

Review points

Other progress review mechanisms

**Software Project Control**

Answering the question "Where are we?"

Controlling progress

Managing to the schedule

Isolating problem areas

"Re-routing" the project

Controlling Quality—Software quality assurance

The elements of SQA

Using formal technical reviews

Statistical SQA

Controlling Quality—Software configuration management

SCM functions

Automating the process

**Software Project Tracking**

Conventional tracking mechanisms

The daily/weekly/monthly progress report

Micro-milestones

Holding Successful meetings

Recognizing when to revise (schedule, budget, expectations)

**Course Wrap-up**

**5.2 A SHORT COURSE: SOFTWARE ENGINEERING METHODS**

*Audience:* Technical staff and managers who must understand modern software engineering methods and how to apply them.

*Duration:* 4 days

*Course description:* This course presents modern software engineering methods. Paradigms for software development are emphasized. The bridge between the customer and the system is described. Analysis and design methods, with an emphasis on object-oriented techniques, are presented. Testing methods and strategies are presented. The impact of CASE tools across all methods is described. Mini-lab sessions are used to reinforce importance concepts.

*Prerequisite:* Basic understanding of software development and implementation (knowledge of software engineering principles is not assumed)

*Upon successful completion, the student will understand:*

• How software engineering paradigms are applied on real projects

• The formal techniques that can be used to help the developer and customer define requirements

• The notation, the tools, and the approach used for conventional analysis and design

• The principles underlying object-oriented thinking and the methods that are used to accomplish OO analysis and design.

• The fundamental "measures" of design quality

• How to ensure the quality of engineering models

• The strategy and tactics for software testing and some methods for "test case design?"

• What CASE tools are available to help

*Outline:*

**Software Engineering**

The elements of the technology

Paradigms for software engineering

**Collaborative Requirements Gathering Techniques**

Working with the customer

Gathering requirements

Use-cases and a requirements gathering mechanism

**Conventional Analysis Modeling**

Analysis principles

Creating the flow model

Control and process specifications

The requirements (data) dictionary

Specification Principles

**SQA Mechanisms**

Formal Technical Reviews

Technical metrics for engineers

**Conventional Software Design**

Design Fundamentals

Data design

Architectural design

Procedural design

Interface design

**Object Oriented Software Engineering**

Basic OO Concepts

Basic OOA Principles

Basic OOD Principles

Unified Modeling Language (UML)

OO Metrics

**Software Testing**

Basic objectives

Test case design

White-box methods

Black-box methods

Special issues for object-oriented testing

Testing strategies

Debugging

**Course Wrap-up**

**5.3 A SHORT COURSE: SOFTWARE QUALITY ASSURANCE**

*Audience:*  Intended for managers and senior technical staff who need guidance in establishing effective SQA activities.

*Duration:* 2 or 3 days (a workshop version is also available)

*Description:* This course introduces techniques and standards for achieving and ensuring high quality in computer software. Includes discussion of software quality and the metrics that are used to assess it; the activities that are required to establish an effective quality assurance approach including a variety of relevant standards, and the process related issues that must be implemented to achieve continuous quality improvements.

*Questions to be answered:*

• What is software quality?

• How is software quality measured?

• What is the primary objective of SQA?

• What activities are considered to be part of SQA?

• How is a pragmatic SQA Plan developed?

• How do we conduct effective formal technical reviews?

• When are statistical techniques worthwhile?

• What management techniques have an impact on quality?

• What software engineering methods have an impact on quality?

• How does SCM fit into the quality assurance picture?

• How do we establish a process that leads to high quality?

*Outline:*

**Software engineering and the Quality Context**

What is software engineering?

Software engineering components

**What is software quality?**

A definition

Quality factors

The quality trilogy

Quality metrics

**The Elements of SQA**

Standards and their use in implementing an affective SQA approach

The IEEE SQA Plan

The SEI CMMI and its implications

ISO-9001:2000

Measurement and metrics

Quality control checkpoints—formal technical reviews

Statistical SQA approaches

**The SQA Plan (IEEE Std. 730.1-1989 and related standards)**

Purpose of the plan

Individual topics considered

How to create an effective plan

**The SEI-CMM and SQA**

The SEI View of SQA in a repeatable process

SQA goals, commitment, abilities, and activities

**What We can Learn from ISO 9000**

The 20 ISO requirements

Documentation and record keeping

Technical activities

Auditing your process

**Quality Measurement**

Process metrics

Product metrics

**Formal Technical Reviews**

Why reviews are so important

The players

A review scenario

How many/how often?

**Statistical SQA**

A quantitative approach

Data collection and analysis

Isolating the “vital few”

**Managing software projects to achieve high quality**

Planning and control

Project monitoring & measurement

Risk analysis and management

**Engineering software to achieve high quality**

Analysis

Design

Construction

Testing

Maintenance and reengineering

**Software configuration management—an element of SQA**

What it is ... why it's needed...how it impacts quality

Change control disciplines

The CCB

The flow of control

Principles of configuration auditing

Status accounting principles

**Process Issues**

Establish a common process framework and integration SQA into it

SQA “checkpoints”

SQA and its application for various process models

The linear model

The prototypical model

Iterative, evolutionary and incremental models

**Course Wrap-Up**

**5.4 SEMINAR: THE CHALLENGE OF SOFTWARE DEVELOPMENT**

*Audience:* Business managers with an interest in software development; managers with direct and indirect responsibility for software development and maintenance; technical staff who require an overview of software engineering issues

*Duration: 3 -4 hours*

*Description:* This seminar discusses software as a strategy business issue, software engineering as an enabling technology, CASE, and guidelines for making the technology work within a company that adopts it.

*Learning Objectives:*

• to understand the nature of computer software and the software engineering process and why it requires management attention

• to view CASE as a tool set that forms an environment for software development

• to understand the impact of software engineering and CASE on productivity and quality

• to outline a strategy for instituting the technology

*Outline:*

**What is software and why is it important?**

defining software

management challenges

**Measurement and software engineering**

why measurement leads to good management

what do we measure?

how do we measure it?

what do we do with the measures?

**The challenge of change**

**Software engineering—a management view**

**Software process improvement**

the SEI CMM

ISO-9001:2000

Process improvement strategies

**Managing the transition to a more mature process**

An SPI process model

assessment

education

selection

installation

evaluation

**Seminar Wrap-up**

**5.5 SHORT COURSE: SOFTWARE TESTING**

*Audience:* Practitioners and technical managers who must understand and apply effective software testing methods and strategies.

*Abstract:* This course presents modern methods and strategies for software testing. Testing is discussed in the context of a software engineering approach. The role of testing as a quality assurance filter is discussed. Strategies for testing are presented. Black box and white box testing methods are presented and applied in laboratory sessions.

*Duration:* 3 days

*Outline:*

**Software Engineering**

The elements of the technology

Paradigms for software engineering—the role of testing in each

Methods—how testing play a part

CASE—the impact of testing tools

**Software Quality Assurance and Testing**

The elements of software quality

QA and software testing

The numbers and the competition

The SQA umbrella

Software engineering "filters"

Software reviews and testing

Quality metrics

the difference between "errors" and "defects"

statistical SQA and testing

common measures

achieving higher defect removal efficiency

Test planning

**Testing Fundamentals**

The objective

Who tests software?

When are we done?

The test spiral

Test case design—the options

Selective vs. exhaustive testing

**White-box testing**

The goal of the black box approach

Basis path methods

using cyclomatic complexity

the graph matrix

Condition testing

Data flow testing—an overview

Loop testing

**Black-box techniques**

The goal of the black box approach

Equivalence partitioning

class definition guidelines

Boundary value analysis

Logic based testing

using decision tables

Specialized testing methods

for human interfaces

stress testing, smoke testing

**Testing Strategy**

Unit testing

the unit test environment

who does it and when

critical modules

unit test worksheets and reports

Integration testing

differing strategies

stubs and drivers

top down and bottom up approaches

sandwich testing

High order testing

validation testing

system testing

alpha and beta testing

**Debugging**

The debugging process

Symptoms and causes

Debugging techniques

brute force

backtracking

induction/deduction

**The Use of CASE Tools**

**Course wrap-up**

6 CHAPTER GUIDELINES

*Software Engineering: A Practitioner's Approach* (SEPA8e) has been organized into five parts that address the primary concerns of most practicing software engineers and the topic focus of most professors who teach the subject. In this segment of the *Instructor's Manual,* guidelines for the use of each of the chapters in SEPA8e are presented.

The chapters in Part 1 of the book focus on defining the product and the process. A key issue in this part of the book is the variety of different software engineering paradigms—their strengths and weaknesses. SEPA8e emphasizes the agile process model (Chapter 5), something that may be of interest to you and your students.

The chapters in Part 2 of SEPA present the methods that encompass software engineering modeling. Analysis and design models are considered in detail. SEPA8e integrates conventional and object-oriented methods to provide a coherent approach to software engineering practice. In addition, UML is emphasized as the modeling technique of choice.

The chapters of Part 3 focus on quality management. The software engineering actions that are required to verify and validate the analysis and design models, as well as actual code are discussed.

The chapters of Part 4 present software project management and related topics. There is sufficient detail in this part to conduct a software project management course. In general, however, most professors will spend relatively little time here.[[5]](#footnote-5)

Finally, the chapters of Part 5 consider software process improvement and future trends in software engineering.

**6.1 AN ABBREVIATED JOURNEY THROUGH THE BOOK**

Before presenting a more detailed discussion of each chapter, we thought it might be worthwhile to provide you with a thumbnail sketch of the book. Reasonably detailed chapter notes may be found in the *Student Resources* section of the SEPA website at **http://www.mhhe.com/pressman.**

**Chapter 1** discusses the impact of software and provides a number of different ways to characterize software. It is intended to get the student thinking about software in a broader context and is mandatory reading for all introductory courses (and many advanced presentations as well).

**Chapter 2** presents software engineering as encompassing process, methods, and tools that enable complex computer-based systems to be built in a timely manner with quality. The software process incorporates five framework activities—communication, planning, modeling, construction, and deployment—that are applicable to all software projects. Software engineering practice is a problem solving activity that follows a set of core principles.

**Part 1—The Software Process**

**Chapter 3** presents a generic process model for software engineering that includes a set of framework and umbrella activities, actions, and work tasks. The intent is to identify a set of “framework activities” that are applicable regardless of the specific process model chosen for a project.

**Chapter 4** expands upon the generic model of the software process and explores the spectrum of software process models that are used throughout the industry. Some of these models are ‘old-school’ (e.g., the Waterfall) while other are being applied on thousands of projects every day (e.g., Incremental models).

**Chapter 5** introduces the agile philosophy and agile methods. This is a hot topic in the industry, but it really represents little that is novel. Everyone wants to be agile—the real question is, how do we get there. In this chapter, a number of ‘agile’ models are considered.

**Chapter 6** describes a successful software engineer as having good technical skills. But in addition, he must take responsibility for his commitments, be aware of the needs of his peers, be honest in his assessment of the product and the project, be resilient under pressure, treat his peers fairly, and exhibit attention to detail. A successful (“jelled”) software team is more productive and motivated than average. To be effective, a software team must have a sense of purpose, a sense of involvement, a sense of trust and a sense of improvement.

**Part 2—Modeling**

**Chapter 7** takes a generic view of software engineering practice and presents the basic concepts and principles that are applicable regardless of the specific methods that are applied.

**Chapter 8** presents requirements management and discusses the fundamental principles of all analysis activities, emphasizing requirements gathering and the analysis of the information domain.

**Chapter 9** discusses requirements modeling focusing on user scenarios And the way in which they can be used to derive requirements.

**Chapter 10** discusses class-based modeling using information derived from use cases and other written application descriptions to identify analysis classes. A set of class-responsibility-collaborator index cards can be used to define relationships between classes. A variety of UML diagrams are described.

**Chapter 11** discusses requirements modeling focusing on behavior, patterns and requirements modeling for WebApps and MobileApps. Analysis patterns are presented in the context of the requirements model.

**Chapter 12** presents design engineering and presents a set of basic design concepts that may be used as an indicator of software quality. The concepts of data, architectural, component-level and interface design are introduced, and the content of the design model is presented. Design patterns are introduced.

**Chapter 13** discusses the underlying concepts associated with software architecture. Styles and patterns are presented.

**Chapter 14** presents component-level design, with an emphasis on the principles of structured programming and the notations that are used to representprocedural designs.CBSE is also discussed in some detail.

**Chapter 15** introduces design considerations for user interfaces. "Golden rules" for UI design, task analysis and modeling, design approaches, and implementation tools are all discussed.

**Chapter 16** discusses design patterns in general and specific patterns for architectural, component-level, interface, and Web/Mobile App design.

**Chapter 17** presents design modeling for WebApps. Six design elements: interface, content, aesthetics, architecture, navigation and components are each considered. Hypermedia design patterns and OOHDM is introduced.

**Chapter 18** presents MobileApp design, encompassing both technical and non-technical activities that include: establishing the look and feel of the mobile application, creating the aesthetic layout of the user interface, establishing the rhythm of user interaction, defining the overall architectural structure, developing the content and functionality that reside within the architecture, and planning the navigation that occurs within the MobileApp.

**Part 3—Quality Management**

**Chapter 19** presents basic quality concepts including a variety of definition for software quality and generic methods for achieving it.

**Chapter 20** discusses review techniques with an emphasis on technical reviews and their impact on software quality.

**Chapter 21** focuses on software quality assurance (SQA) with specific emphasis on formal technical reviews. The basic factors that influence software quality are surveyed, the elements of an SQA approach are defined and the manner in which reviews are conducted is presented in detail.

**Chapter 22** presents the fundamental objectives, overall strategy, and specific methodology associated with software testing and also discusses the elements of software debugging.

**Chapter 23** focuses on a series of white-box and black-box testing techniques that can be applied to conventional software.

**Chapter 24** focuses on a series of testing techniques that can be applied to object-oriented software.

**Chapter 25** discusses testing strategies and tactics for WebApps. Content testing, database testing, user interface testing, component-level testing, navigation testing, configuration testing, security testing, and performance testing are each considered.

**Chapter 26** discusses testing strategies and tactics for MobileApps. The goal of Testing focuses on the quality elements such as content, function, structure, usability, use of context, navigability, performance, power management, compatibility, interoperability, capacity, and security. It incorporates reviews and usability assessments that occur as the MobileApp is designed and has been deployed on an actual device.

**Chapter 27** focuses on security engineering as being concerned with developing software that protects the assets that it manages from threats. Security risk management is concerned with assessing the impact of possible threats and deriving security requirements to minimize critical loses.

**Chapter 28** discusses the cleanroom software engineering paradigm. Functional specification, design refinement and verification and cleanroom testing are considered. In addition, the concepts and notation that are used to formally specify software..

**Chapter 29** presents a technical discussion of SCM. The four primary activities associated with this important software engineering "umbrella" activity are presented and the concept of configuration objects and the project database (the repository) are introduced.

**Chapter 30** presents a set of technical measures and metrics that can be used to assess the quality of analysis and design models as well as source code. Metrics for software testing and maintenance are also introduced. Very few software engineering books address technical metrics (an oversight in our opinion). Students should understand that engineering measures for software (although imperfect) do exist.

**Part 4—Managing Software Projects**

**Chapter 31** presents an introduction to software project management and discusses the “four P’s”—people, problem, project, and process. This chapter serves as an introduction to the remaining chapters in this part of the book.

**Chapter 32** emphasizes the importance of measurement and metrics applied to the process and to projects. Quality and productivity metrics are emphasized, with primary emphasis being placed on LOC and function point measures. This sets the stage for their use in project planning.

**Chapter 33** focuses on project planning, presenting important techniques for cost and effort estimation. Decomposition and empirical models for cost estimation are considered.

**Chapter 34** continues the discussion of project planning, but here I emphasize scheduling and the creation of a common process framework. The content *Project Plan* is also considered.

**Chapter 35** discusses risk management for software projects. A risk management paradigm is presented. It should be noted that risk management is a critically important project management activity, yet relatively few people apply it in the real world. Your students would benefit from learning something about it.

**Chapter 36** presents maintenance and reengineering with an emphasis on software related issues. Business process reengineering is introduced (to establish a context). Then, reverse engineering, restructuring and forward engineering methods are considered.

**Part Five—Advanced Topics in Software Engineering**

**Chapter 37** discusses software process improvement (SPI) with an emphasis on the CMMI and other SPI frameworks. Also considers the ROI for SPI and on-going trends.

**Chapter 38** takes a look at software engineering trends. The basic structure of technology trends and overall technology directions are presented.

**Chapter 39** looks at the road ahead for software engineering and also examines software engineering ethics.

**6.2 A CHAPTER-BY-CHAPTER DISCUSSION OF SEPA8e**

The discussion of each chapter begins with an indication of the overall intent of the chapter and the critical concept that should be emphasized. Our intent here is not to provide a chapter summary, but rather to suggest points of emphasis and discussion as well as supplementary topics that may be of interest to your students.

**Chapter 1 — The Nature of Software**

The goal of this chapter is to introduce software. Emphasize that software is a “product” designed and built by software engineers.

Software is the key element in the evolution of computer-based systems and products and one of the most important technologies on the world stage. Over the past 50 years, software has evolved from a specialized problem solving and information analysis tool to an industry in itself. Yet we still have trouble developing high-quality software on time and within budget.

Spend some time talking about the three major software characteristics. The “wear” discussion is critical—be sure your students understand it. Legacy software continues to present special challenges to those who must maintain it.

The nature of software is changing, WebApps and MobileApps have evolved from simple collections of information content to sophisticated systems that present complex functionality and multimedia content. Although these Apps have unique features and requirements, they are software nonetheless. Mobile applications present new challenges as apps migrate to a wide array of platforms. Cloud computing will transform the way it which software is delivered and the environment in which it exists. Product line software offers potential efficiencies in the manner in which software is built.

**Chapter 2 — Software Engineering**

The goal of this chapter is to introduce software engineering. You should indicate that software engineers have a moral and ethical responsibility to ensure that the software they build does no serious harm and meets the needs of the people who request it and those who use it. In fact, many adopters think it’s worthwhile to jump to Section 39.5 and discuss ethics more formally at the outset.

Be sure your students understand that software engineers are often concerned with the technical elegance of their software products, but other stakeholders tend to be concerned only with whether or not a software product meets their needs and is easy to use.

Recognize that any attempt to develop an all-encompassing definition of software engineering is an exercise in frustration. Dissect the IEEE definition of software engineering with your students. Be certain that they understand that a “systematic, disciplined, quantifiable approach” does *not* mean a bureaucratic, document-laden approach. The point here is to emphasize that software engineering is a “layered” technology. If you have the time and the inclination, you might want to introduce your students to SWEBOK **(http://www.swebok.org/)** as another source of software engineering information.

Even though it might seem to be a simple idea, many students struggle with the concept of a process, confusing process with methods and tools. Spend some time discussing the meaning of a process framework and the basic framework activities introduced here.

Polya’s work is pivotal to basic understanding of the software engineering approach. Spend time discussing his seminal ideas along with the questions that he posed at each step along the way.

The myths presented provide a good source of material for class discussion. Ask “why” people believe these myths and whether there are grains of truth in each (there are). Ask you students to think of one or two software myths of their own.

*SafeHome* is a useful case study that you’ll encounter throughout the book. It’s worth spending some time discussing how projects start in the real world. Based on the sidebar, ask students to develop a list of 10 questions that they would need answered about *SafeHome.*

**Part 1—The Software Process**

**Chapter 3 — Software Process Structure**

The intent of this chapter is to establish a definition for software engineering and to present a generic software process model that can be used as a template for all other process models presented in the next chapter. We present a process framework that encompasses five activities— communication, planning, modeling, construction, and deployment. Be certain that your students understand each of the framework activities and the fact that they are elaborated differently depending on the 3 P’s—problem, project, and people.

Note that the framework activities are applicable to all projects and all application domains, and they are a template for every process model discussed in this book.

Many students have trouble understanding “umbrella activities” and “task sets.” Be certain that you’ve discussed these concepts thoroughly. Spend some time on the first Task Sets sidebar, noting that a task set will vary depending on the characteristics of a project.

You’ll encounter the first appearance (of many) of “patterns” in this book. It might be worthwhile discussing patterns in general, their meaning, their intent, and the benefits of using them across many software engineering topics. If you are a patterns advocate, you might digress and discuss Alexander’s work, pattern templates and so on. If time permits, have your students suggest a pattern for a non-software engineering related process, e.g., something in a fast food restaurant serving process.

Many adopters will choose to skip this discussion of SPI, delaying it until the end of the course (Chapter 37).

**Chapter 4 — Process Models**

The intent of this chapter is to present an overview of several prescriptive process models.

Many people (and not a few professors) believe that prescriptive models are “old school”—ponderous, bureaucratic document-producing machines. Disabuse your students of this notion. Every prescriptive process presented in this chapter can be streamlined (made agile). Note that “prescriptive” simply means that the process model identifies a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance and change control mechanisms—for each project.

It is easy for students to become so lost in the details of the various process models that they fail to see the features the models have in common with each other. Refer back to the framework activities introduced in the previous chapter to emphasize generic similarities. Another difficulty students have is their belief that each activity within the process is performed completely independently of the other activities. The reality is that there tends to be lots overlap among these activities.

We’ve included the UP in this chapter for completeness and because we use UML as a modeling notation throughout this book. The phased approach noted can be mapped nicely into the generic framework introduced earlier in this chapter. If you consider the UP, you should discuss what use-cases are (this is covered in detail in later chapters) and why a “use-case driven process” has merit.

Two important “personal” process models (PSP and TSP) are discussed. Your students may find it helpful for you to provide examples of how development might proceed under each model. The PSP model is good from the perspective that an individual software engineer can use it to improve his or her personal productivity and work product quality. Both models are largely iterative or evolutionary in their approach to software development.

PSP and TSP are interesting, but are not pivotal to an understanding of process issues. If time permits and your students (or you) have an interest, a discussion is worthwhile. The key point to emphasize is that individuals and teams should measure their work and the errors they make and act to improve their approach so that the causes of errors are.

**Chapter 5 — Agile Development**

This intent of this chapter is to discuss the “agile” philosophy and to present a variety of agile process models and methodologies. The overriding theme of this chapter is that everyone wants an agile process—the only debate is on how to achieve one and the level of discipline that is incorporated into such process models. Most students love the agile approach for the wrong reasons. They interpret it as glorified hacking. Be certain that you emphasize that engineering discipline and agility are NOT mutually exclusive.

Be sure to emphasize that agility is more than change management. Agility means that customers and developers need to work together as collaborators on the development team and try to build products that can be adapted to a rapidly changing market place. Part of this adaptation is learning when and how to streamline the production of work products and focus on the development of incremental operational prototypes.

Spend some time talking about the ramifications of the Agile Manifesto (noted in the chapter introduction and then note that none of this implies that discipline is discarded. Software engineering emphasizes each element noted in the manifesto. Also note that the agile philosophy is seductive, but must be tempered by the demands of real systems in the real world. On the positive side, note that many of the ideas espoused as part of the agile approach are excellent and worth considering regardless of the process model a team adopts.

All agile processes are adaptable to manage unpredictable changes that take place during software development projects. Agile processes rely heavily on customer feedback generated by their evaluation of operational prototypes. The focus of agile processes is on the delivery of software increments in relatively short timeframes. It is important for students to be exposed to the arguments for agile development (product is more important than documentation) and against agile development (rapidly produced prototypes do not always scale up to enterprise-wide software applications). The idea of tradeoffs is an important point to emphasize here. A third part of this section deals with human factors and group dynamics of agile teams. Students should not underestimate the potential problems that can result if teams do not function well.

Each of the agility principles presented should be discussed at length. Ask your students how they would achieve each. One of the best debates about agile development appeared in the June 2003 issue of *IEEE Computer.* You might want to assign it as additional reading if you’re emphasizing agility.

Several agile process models are discussed. Your students may have some misconceptions about some of these models from experiences prior to this course. For example, Extreme Programming does not mean writing code without documentation or testing. Many of these process models make references to chapters later in the text. Your students may need additional background to appreciate some of the points made about this process models. The point of this section is to expose students to the philosophies and activities associated with these agile process models. They should be encouraged to look for the common elements found in each. For the most part students will not be able to apply these process models to their own projects at this point in the course. They should have enough background to apply these models are reading the text chapters on analysis modeling, object-oriented analysis, and testing strategies.

There is little debate that XP is the most dominant of all agile models (at this point in time), and yet, the other models summarized in this chapter have characteristics that are worth noting (e.g., the Scrum stand-up meeting). It’s unlikely that you’ll have time to cover each of the models presented. Therefore, I suggest presenting XP and emphasizing user stories, pair-programming, refactoring, and continuous integration as important XP concepts and then supplementing with elements of the other models that you consider to be noteworthy.

The terms “collaboration” and “self-organizing teams” are used repeatedly when agile development is presented. You might discuss the meaning of these terms with your students.

**Chapter 6 — Human Aspects of Software Engineering**

This intent of this chapter is to present the principles that serve as a foundation for software engineering as a collaborative activity. Software engineering can also be viewed as a helping profession. Without problems to solve or stakeholders to work with software development is an academic activity at best.

A successful software engineer must have technical skills. But in addition, he must take responsibility for his commitments, be aware of the needs of his peers, be honest in his assessment of the product and the project, be resilient under pressure, treat his peers fairly, and exhibit attention to detail. We require our students to rate each team members contributions to any team product being graded in our courses as a way of driving this point home.

The psychology of software engineering includes individual cognition and motivation, the group dynamics of a software team, and the organization behavior of the company. In order to improve communication and collaboration, members of a software team can take on boundary spanning roles.

A successful (“jelled”) software team is more productive and motivated than average. To be effective, a software team must have a sense of purpose, a sense of involvement, a sense of trust and a sense of improvement. In addition the team must avoid “toxicity” that is characterized by a frenzied and frustrating work atmosphere, an inappropriate software process, an unclear definition of roles on the software team, and continuous exposure to failure. You might want to present case study illustrating the impact of various elements of team toxicity.

Social media is becoming an integral part of many software projects. Blogs, microblogs, forums, and social networking capabilities help to form a software engineering community that communicates and coordinates more effectively. You might encourage students to create Google+ or Facebook groups to facilitate communication during your course.

Cloud computing has the potential to influence the manner in which software engineers organize their teams, the way they do their work, the manner in which they communicate and connect and the way software projects are managed. In situations in which the cloud can enhance the social and collaborative aspects of software development, its benefits far outweigh its risks. You may wish to encourage students to make use of Google docs, sky drives, or code repositories (e.g. CVS) as they work on team projects in your course.

**Part 2—Modeling**

**Chapter 7 — Principles that Guide Practice**

The intent of this chapter is to present the principles that serve as a foundation for software engineering practice. The principles presented in this chapter are applicable regardless of the specific process that is used, methods that are chosen or tools that are applied.

In many cases, the principles discussed here should be revisited as each software engineering activity is presented during the course. For example, modeling principles can be revisited when the chapters on software analysis and design are covered.

Be sure to cover each of the core principles presented in the introduction of Section 7.2. They represent an overarching philosophy for software engineering practice and represent the basis for all further discussion throughout Part 2 of the book. The entire chapter represents a generic software engineering mini-course in the sense that it presents the common sense ideas that all students must understand. Be sure to spend time here.

The principles that guide process can be considered in light of what has already been learned in Chapters 3 to 5. Be sure to emphasize agility (regardless of the process model that is chosen), adaptation, and the need to develop only those work products that have value to others.

The principles that guide practice focus on problem solving. It might be worthwhile to reprise Polya’s work (Chapter 2) and then be certain that all terms discussed in Section 7.2 are well-understood by your students.

You can choose to postpone a discussion of the principles that guide each framework activity until the chapters associated with that activity are covered. Alternatively, each set of principle can be considered now, with an eye toward introducing more detailed discussions that follow.

**Chapter 8 — Understanding Requirements**

The intent of this chapter is to present requirements engineering tasks and basic requirements analysis concepts and principles. Students must understand that customer communication and requirements elicitation are pivot activities in the software process. Even the best technology will fail if you don’t understand what the customer wants.

Regardless of the process model used by software engineers, requirements engineering is an essential part of a successful software development project. The focus of this chapter is gathering the information needed to build the information, function, and behavioral portions of the analysis model. Creation of use-cases is described. Several UML diagrams useful to analysis modeling are discussed. Dataflow diagrams are introduced, but will be discussed in greater detail in the next chapter.

In this chapter six requirements engineering task are considered—inception, elicitation, elaboration, negotiation, specification, and validation.

A good way for you to introduce inception, elicitation, elaboration, negotiation, specification, and requirements validation is to create a scenario in which your students must address what they’d do to accomplish each for a hypothetical project you pose. For example, assume you work for a music company and want to develop a system that will distribute .mp3 files for very low prices. What things will happen during inception, elicitation, elaboration, negotiation, specification, and validation?

Students need to experience the process of working with a customer to develop the requirements for a software product before they graduate. It is hard for students to do this in a one-semester software engineering course, but they should do this as part of a senior design project. Trying to understand customers' needs and negotiating changing requirements is part of every software development project. Students also need to experience the opportunity to judge the feasibility of solutions and steering the customer toward options that can be completed with the resources available.

Spend a few moments going over the requirement validation checklist presented in the sidebar.

The term “stakeholder” (Section 8.2.1) is used throughout SEPA. It’s very important that your students understand its meaning and just who is a stakeholder. Be sure you emphasize this.

Spend some time discussing how you reconcile multiple viewpoints (and conflicting requirements). Also discuss how collaboration is best achieved on a software project.

Two elicitation techniques are discussed (the use of context free questions in structured customer interviews and collaborative requirement gathering. Having students use these techniques to work with real customers or role-play working with simulated customers is a worthwhile task. If there is not enough time to do this in a one-semester course, if should be done in the senior projects course.

The series of context-free questions listed in section 8.2.4 should help the students gain a rudimentary understanding of the type of dialog needed between developers and customers when begin a software development project.

Basic requirement elicitation is presented for *SafeHome.* It would be worthwhile to cover this in lecture.

Use cases are a pivotal topic throughout SEPA. Be certain that you emphasize and re-emphasize their importance as a requirements gathering tool.

Go though each of the questions that use cases ask and answer. Students often have difficulty understanding what an “actor” is. Be sure you explain the concept to them.

*Note:* If you’re a purist, then you’ll insist that students develop use cases by following the template suggested in this section. However, you can be a bit more relaxed and have students develop a more narrative and informal “preliminary use case.” The idea is to have them start thinking in scenarios—in our opinion, the form the scenario takes is secondary.

Spend a bit of time discussing the syntax and semantics of UML use case diagrams.

Your students may need some help in thinking about people and devices as actors. Object-oriented analysis does not always come easily (even if your students are good object-oriented programmers). Functional decomposition may seem more natural to them in the beginning.

The elements of the analysis model (scenario-based, class-based, behavioral, and flow-oriented) are introduced. It may be helpful to show your students additional examples of use-case, activity, class, and state diagrams.

The analysis model is considered in detail in Chapters 9 to 11. At this point, you should focus on defining the four major elements of the model: scenario-based elements (use-cases), class-based elements, and behavioral elements. A few simple examples can serve to foreshadow the content of Chapters 9 to 11, but there’s no need to go into detail at this point.

Most students have very little experience in the art of negotiation. The key point is to remind students to avoid an “us versus them” mentality when they become software developers. The negotiation guidelines included in this section should be helpful to your students in this regard. If time permits, discuss how the negotiation process works and point out tricks of the trade. You’ll provide students with information that will serve them well.

**Chapter 9 — Requirements Modeling: Scenario-Based Methods**

This is the first of three chapters that provide the reader with an understanding of the mechanics of requirements modeling. In this chapter, the emphasis is on scenario-based modeling.

Be certain to emphasize the objectives of requirements analysis: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. Also, emphasize the “rules of thumb” presented in Section 9.1.2.

Domain analysis is discussed as an umbrella activity in which a set of reusable classes is defined following the examination of application area. Students may benefit from examining at a case study in which a library of reusable software components is assembled over time. Students might also benefit from an assignment in which they are required to devise a plan for building a new software application out of an existing set of reusable objects.

Spend some time discussing Figure 9.3 so that students get a big picture view of different approaches to requirements. Approaches.

The degree to which you emphasize scenario-based modeling will depend on whether you examined the subject in detail as part of your discussion of Chapter 8.

The formal use-case template presented in Chapter 8 should be revisited and then compared to the less formal use-case presentation in this body section. The formal use-case template is presented in the sidebar contained here.

Section 9.3.1 presents UML activity diagrams, a mechanism for representing procedural scenarios. It is important to emphasize that the activity diagram is not used to “write the program” at this stage. Rather it is intended to represent topic level procedural scenarios.

The key point to emphasize when discussing UML swimlane diagrams (Section 9.3.2) is that they are a variation of activity diagrams that “connect” procedural flow to major analysis classes.

Students may benefit from trying to represent the usage scenarios from familiar software applications using UML use-case, activity, and swimlane diagrams. These UML representations are valuable in both structured and object-oriented analysis modeling.

**Chapter 10 — Requirements Modeling: Class-Based Methods**

This is the second of three chapters that provide the reader with an understanding of the mechanics of requirements modeling. In this chapter, the emphasis is on class-based modeling.

The section on class-based modeling describes the process of developing an object-oriented analysis (OOA) model. The generic process described begins with guidelines for identifying potential analysis classes, suggestions for defining attributes and operations for those classes, and a discussion of the Class-Responsibility-Collaborator (CRC) model. The CRC card is used as the basis for developing a network of objects that comprise the object-relationship model.

CRC modeling has significant value and should be emphasized here. Students need to experience the process of developing a set of CRC cards for one of their own systems (preferably one that they have written usage scenarios for). They also need to experience the process of conducting walkthroughs of their usage scenarios using their CRC card sets. The process of building a class hierarchy from their debugged CRC system would also be good student assignment.

UML diagrams associated with the class-based modeling element are relatively simple, but can still pose a challenge for many students. We recommend supplementing SEPA with additional UML materials so that your students have a solid understanding of class diagrams, associations, aggregations, dependencies, multiplicity, packages, and the like.

**Chapter 11 — Requirements Modeling: Behavior, Patterns, and Web/Mobile Apps**

This is the third of three chapters that provide the reader with an understanding of the mechanics of requirements modeling. In this chapter, the emphasis is on, behavioral modeling, patterns, and analysis modeling for Web and Mobile Apps.

We have removed data-flow modeling in this edition of SEPA. Some reviewers were upset by our decision, arguing that DFDs (and related notation) are still widely used. There is some truth to their argument, but trends in requirements modeling are moving away from data flow-oriented approaches.

The section on behavioral modeling describes the use of UML state diagrams (SDs) as one means of representing a software behavioral model. Students may have seen state diagrams in their theory of computation or discrete mathematics classes. If they have not, you may need to show students some more examples. Students often tend to omit state transitions to handle erroneous inputs when building their first SD. Students will need to construct at least one SD on their own before they begin to feel comfortable with behavioral modeling.

The UML sequence diagram is another form of behavioral model that indicates how events cause transitions from object to object.

To build the object-behavior model, students will need to return to the use-cases and identify the sequences of events. Events occur whenever an actor (person, device, or system) and the OO system exchange information. Students should be encouraged to markup their own use-cases to determine the events. The events trigger transitions between system states. Sequences of events can be used to build a state diagram that represents the object-behavioral model.

The use of patterns (for both analysis and design) is an important skill for all students to understand and learn. If you haven’t already done so, spend some time introducing basic patterns concepts. These ideas are not always intuitively obvious to all students. Be sure to explain how analysis patterns and design patterns differ and consider (in some detail) the example in Section 11.4.2 or one of your own choosing.

Requirements analysis for Web and Mobile Apps encompasses three major tasks: formulation, requirements gathering, and requirements modeling. Three different requirements modeling actions are introduced. Because the elements of the requirements model (and the actions required to develop each element) are presented in subsequent sections, we would suggest that you spend relatively little time on this section.

The content model contains structural elements that provide an important view of content requirements for a WebApp or MobileApp. These structural elements encompass content objects (e.g., text, graphical images, photographs, video images, audio) that are presented as part of the App. In addition, the content model includes all analysis classes— user-visible entities that are created or manipulated as a user interacts with the App.

The approach for developing the content model is analogous to the approach presented in Chapter 10 for the derivation of analysis classes. Content objects are extracted by examining use-case descriptions, the relationships among content objects are modeled, and analysis classes are derived.

If time permits, you might expand the example provided in this section, adding a few more objects and classes.

We’ve emphasized use-cases as an analysis mechanism, so by now, your students should be quite familiar with them. You might consider sending some time further discussing the UML sequence diagram and state diagrams (as an alternative or supplement to the use-case).

As a classroom exercise, present your students with basic requirements for a WebApp or MobileApp of your choosing. Allow them to ask you (the customer) questions about these requirements. Then ask them to create a paper prototype of the interface. Compare the results to illustrate how different interpretations of the system can be.

The functional model addresses two processing elements of the App, each representing a different levels of procedural abstraction: (1) user observable functionality that is delivered by the App to end-users, and (2) the operations contained within analysis classes that implement behaviors associated with the class.

If time is at a premium, you may have to present configuration modeling topics only briefly. Depending upon their background, your students may require a digression into client/server architectures and related issues. This is normally beyond the scope of a software engineering course.

**Chapter 12 — Design Concepts**

This intent of this chapter is to provide an introduction to the design process and to describe fundamental design concepts that are essential to an understanding of any software design method. Basic concepts are introduced and a fundamental design model is discussed. The design model consists of architectural design, component-level design, interface design, patterns-based design and Web/Mobile App design.

Although many of the design concepts discussed in this chapter may have been presented in earlier courses, it is important to re-emphasize each concept so that all students have a consistent understanding of them. Students should be encouraged to use the design document template form the SEPA website as a basis for the design work products they create for their own software projects.

It is important for students to understand the mapping from analysis model to the design model. Spend some time discussing Figure 12.1 to establish the relationships.

Students should be reminded that all design work products must be traceable to software requirements and that all design work products must be reviewed for quality.

Be sure your students understand that software quality begins with solid design. The first part of this chapter addresses the elements of software quality. Be certain to emphasize these.

Make the point that software design is an iterative process that is traceable to the software requirements analysis process. Students need to be reminded that many software projects iterate through the analysis and design phases several times. Pure separation of analysis and design may not always be possible or desirable. Having your students discuss the generic design guidelines as a class may be a worthwhile activity.

Section 12.3 discusses many significant design concepts (e.g., abstraction, refinement, modularity, architecture, patterns, refactoring, functional independence, information hiding, and OO design concepts). For some of these concepts a simple definition will be sufficient. For others (e.g. functional independence) presenting additional examples may be helpful.

Students should be told that the details required to develop a complete design model appear in Chapters 13 - 18). Students should be reminded that design changes are inevitable and that delaying component level design can reduce the impact of these changes.

Cover each of the design “elements” presented in Section 12.4 to foreshadow content to be discussed in detail in later chapters.

The use of UML notation in design models is described and several examples are given. Students should be encouraged to create some of these diagrams as part of the process of building a design model for a familiar software application. A document template appears on the SEPA web site. It is important to get students in the habit of using diagrams to represent design information whenever it is feasible. Students should be reminded that design changes are inevitable and that delaying component level design can reduce the impact of these changes.

**Chapter 13 — Architectural Design**

The intent of this chapter is to provide a systematic approach for the derivation of the architectural design. Architectural design encompasses both the data architecture and the program structure layers of the design model. A general introduction to software architecture is presented. Examples are presented to illustrate the use of transform mapping and transaction mapping as means of building the architectural model using structured design approach. Students should be reminded that quality reviews need to be conducted for the architectural model work products.

The term “software architecture” is presented as a framework made up of the system structures that comprise the software components, their properties, and the relationships among these components. The goal of the architectural model is to allow the software engineer to view and evaluate the system as a whole before moving to component design.

To get a discussion of software architecture started, ask your students what the architect of a building does and why it’s important. Draw parallels to software.

Be sure that your students understand that an architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. A pattern also imposes a transformation on the design of an architecture but differs from a style as described in the introduction of this section. Patterns can be used in conjunction with an architectural style the shape the overall structure of a system.

A taxonomy of architectural styles is described in Section 13.3.1. With the exception of object-oriented architectures, the architecture modeling technique discussed in this chapter is generally applicable to each of these architectural styles. However, the derivation of the call and return architecture is emphasized. Students should be encouraged to use the questions dealing with control and data issues as basis for assessing the quality of the initial architectural framework before doing more detailed analyses of the proposed software architecture.

Architectural patterns (Section 13.3.2) are relatively obvious for a house or some other physical object, but they are more abstract for software. Spend some time discussing the patterns noted in this section.

The architectural design process begins by representing the system in context. Spend some time discussing the ACD presented in Figures 13.5 and 13.6.

Be sure that your students understand the meaning of an “archetype” (Section 13.6.2) and how it fits into the notion of the architectural design.

The derivation of components that populate the architecture sometime seems like a slight of hand. Be sure you provide examples of how this is done. Both good examples and bad examples might help their understanding.

It’s important that you emphasize the need to consider architectural alternatives. Section 13.7 provides students with an iterative method for performing trade-off analyses of alternative architectural designs for the same software system. Students should find enough information in this section to be able to apply this technique to their own architectural designs. The second approach presented in this section applies pseudo quantitative techniques as a means of assessing the quality of an architectural design. Students might appreciate seeing an example of this approach applied to real system.

Refining the architectural model involves writing a fair amount of documentation (e.g. process narratives, interface descriptions, limitations, etc.) and reviewing work products for quality. Student projects need to be subjected to the refinement process before detailed design work is undertaken. It may be worthwhile to have the students look over the design document template from the SEPA website to see where they need to go next with their design work.

**Chapter 14 — Component-Level Design**

This chapter discusses the portion of the software development process where the design is elaborated. Individual data elements and operations are designed in detail. First, different views of a “component” are introduced. Guidelines for the design of object-oriented and traditional (conventional) program components are presented. This chapter also describes *component-based software engineering* (CBSE) as a process that emphasizes the design and construction of systems with reusable software components. CBSE has two parallel engineering activities, *domain engineering* (discussed earlier in the text) and *component-based software development*. The important point to emphasize to students is that custom software components are only created when existing components cannot be modified for reuse in the system under development.

*Note:* This is as close as SEPA gets to a discussion of coding. If you want to have your students implement software so that it is executable, the discussion of component-level design can be extended to show how the design is implemented in a programming language.

The chapter begins by defining the term *component* and discussing the differences between object-oriented, traditional, and process related views of component-level design.

UML defines a component as “… a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.” You should parse this definition for your students, discussion each important phrase within the context of component-level design.

Be certain that your students understand the rather fundamental difference between the object-oriented and conventional (traditional) view of a software component. In addition, the idea of different categories of components (e.g., control, problem domain, infrastructure) should be discussed.

Finally, the process (CBSE) view of a component should be presented. Middleware components should be discussed, but there is no need to emphasize them if time is short.

Spend some time on the basic design principles presented in Section 14.2.1. It would be a good idea to provide additional examples to help your students understand each.

The principles presented in Section 14.2.1 are worthy of classroom discussion. Again additional examples will enhance understanding. The *SafeHome* sidebar accompanying the section might be considered during your discussion.

The concepts of cohesion and coupling are *very* important. Be certain that your students (1) understand what they are in this context and (2) know how to create component designs that achieve them. The sidebars presented can be discussed in class to help cement student understanding.

The steps outlined for conducting component-level design provide a reasonable task set for designing a component. You should emphasize that (1) design classes in the problem domain are usually custom-designed, however, if an organization has encouraged design for reuse, there may be an existing component that fits the bill; (2) design classes corresponding to the infrastructure domain can sometimes be often from existing class libraries; (3) a UML collaboration diagram provides an indication of message passing between components.

The printing system example presented in this section can be elaborated to provide your students with a more in-depth view of the component design process.

The quote by Blaise Pascal deserves mention in the classroom. Too often, there is a tendency to develop complex designs for relatively simple functions. Stress simplicity!

Discuss refactoring as a design tactic and provide a few simple examples using source code or PDL.

Section 14.6 reviews the elements of the structured programming and a number of representation techniques for procedural design. Students may need to see examples of representing algorithms using flowcharts, if they are not familiar with them from earlier course work. Be sure to note that UML activity diagrams can be used in lieu of flowcharts.

Decision tables are discussed as an example of a tabular tool for representing the connections between conditions and actions. Students will benefit from the experience of developing decision tables for their own projects.

Program design language (PDL) is discussed in this section. The hardest part about getting students to express their designs using a PDL is getting them to stop trying to write directly in their favorite programming language (C++, Java, etc.). Students often fail to distinguish between low-level design and implementation. Requiring students to include pseudocode representations of their algorithms as part of the module design documentation seems to help them understand some of the differences between the two.

It is important to have students understand the differences between CBSE and object-oriented software engineering. The biggest difference is that in CBSE, after the architectural design is established, the software team examines the requirements to see which can be satisfied by reusing existing components rather than constructing everything from scratch. In object-oriented software engineering, developers begin detailed design immediately after establishing the architectural design. Students need to understand the software activities that take place once reusable components are identified (component qualification, component adaptation, component composition, and component update). These activities will be described in more detail later in the chapter. It may be worthwhile for students to be given a set of requirements and an indexed collection of reusable components and try to determine which requirements can be satisfied by the reusable component and which cannot.

Be sure students understand the interrelationships between the domain engineering and component-based engineering activities. The activities are discussed in more detail in the subsequent chapter sections.

Students should make sure they understand the three major domain engineering activities (analysis, construction, and dissemination). It is important for students to remember is that the purpose of conducting domain analysis is to identify reusable software components. Structural modeling is an important pattern-based domain engineering approach. Students may benefit from trying to conduct their own domain analysis. Alternatively, they may benefit from discussing a real world case study that includes domain analysis.

A useful exercise is to have students identify domain classes (or functions) for a domain such as “retailing” or “health care.”

Component-based software development activities are discussed in detail in this section. If students have access to a library of reusable components (or COTS components) they should be encouraged to use the composition techniques presented to assemble a new software product. Students may benefit from using one of the free component libraries like JavaBeans. Another good exercise might be to have students try to design one of their own software components so that is can be added to an existing software reuse library.

The issues associated with indexing and retrieving software components from a reuse library should be addressed because students often (mistakenly) believe this is a simple problem. The material is presented at a fairly general level. If your students are familiar with multimedia databases and client-server computing, you might explore some of the implementation concerns that need to be addressed to construct a reuse repository by examining a real life example of one.

**Chapter 15 — User Interface Design**

Virtually all modern computer-based systems (and many legacy systems that are reengineered) have some type of interactive user interface, and most require reasonably sophisticated interface designs. It is easy for programmers to focus on splashy new technologies and ignore the fact that functionality and usability (not innovation) is what users are most concerned about.

This chapter outlines the design processes for software user interfaces. One key point to get across to students is that understanding the user’s task goals and preferred methods of reaching them is essential to good interface design. A second important point is that the user interface should be designed early in the software development process (and not as an afterthought). Having students formally evaluate each other’s interfaces using the usability checklist from the SEPA web site is a worthwhile course activity.

The Golden Rules discuss three principles of user interface design that students should be encouraged to follow as they build their own software projects. The first is to place the user in control (which means have the computer interface support the user’s understanding of a task and do not force the user to follow the computer's way of doing things). The second (reduce the user’s memory load) means place all necessary information in the screen at the same time. The third is consistency of form and behavior. It is sometimes good to bring in commercial software and try to see how well the interface designers seem to have followed these guidelines.

The notion that user interface design is really the process of reconciling different models (the user model, a design model, a mental model, and an implementation model) of the user interface is important to emphasize. The point to get across to students is that the overall goal of the interface design process is to understand the user and the user’s task so well that the implementation model will match the user’s perception of the of the system. This will make the system easy to learn and easy to use. If your course is more than one semester in length, having students build user interfaces for real customers can be a very instructive activity for them.

In order to understand the user’s perception of the interface, a variety of different information sources are noted. Each of these should be discussed so that the student understand how each is used and how they differ from one another. The questions noted in this section should be asked of every user type that will interact with the software via the interface.

The questions noted for task analysis and modeling are answered as part of this task. This is an important section and you should dedicate lecture time to each of the techniques noted here—use-cases, task elaboration, object elaboration, workflow analysis, and hierarchical representation.

Use-cases provide worthwhile input for task analysis. If time permits, have your students write a set of use-cases that describe how they would like to interact with a futuristic car that drives itself. Don’t get hung up on the technology that accomplishes this—assume it exists. Just address how to interact with the car when it is standing still and moving.

Elaboration is pivotal for many software engineering activities. Discuss task and object elaboration for a user interface and elaborate on the tasks for the futuristic car example. Do the same for the workflow.

Regardless of the technique used to create the user task model, a model describing the users' actions using the operators (keys pressed, mouse operations, menu selections, icon manipulations, commands typed) available in the proposed computer system should be built for each usage scenario. Students will find it very instructive to story board prototypes for task scenarios using the operators from familiar user interfaces. These paper prototypes can be “tested” quickly and inexpensively.

You should dedicate some lecture time to the design issues noted in Section 15.4.3. In many cases features like help facilities, accessibility, internationalization, and error handling are design afterthoughts. Be sure your students understand that these features can be as important as other UI functionality.

Observing users interacting with paper or electronic prototypes is a common way of identifying defects and refining user interfaces. Many database systems (e.g. Oracle Easy SQL or MS Access) come with interface construction tools that can be used build testable user interface prototypes. Multimedia scripting tools (e.g. Director or even MS Powerpoint) or web page editors (e.g. Dreamweaver or MS Frontpage) can also be used to build working interface prototypes quickly. Having a reasonable interface development tool kit allows student designers low risk means of testing the sequencing and layout of the key screens for proposed user interfaces.

Spend some time on Dix’s questions, emphasizing that the interface design must actively help the user answer them. Tognozzi’s principles and Nielsen and Wagner’s design guidelines should be considered at some depth. Try to provide examples that illustrate each.

The interface design workflow is worth covering in class. Be sure that your students understand the intent of each step. In many cases, reference is made to a specific UML diagram. It may be necessary to review these.

**Chapter 16 — Pattern-Based Design**

The intent of this chapter is to provide students with an introduction to design patterns—a topic worthy of serious classroom coverage. However, it’s almost more important that your student understand the intent of patterns and their benefits as opposed to the details of a specific design pattern.

As an introduction, emphasize the pattern as a “a three-part rule which expresses a relation between a certain context, a problem, and a solution.” Discuss Coplien characterization at some length. Have your students use the pattern template to describe a design pattern for some product other than software, say a car or a consumer electronic device. This will help them get a feel for just what a pattern is.

Be sure to explain the difference between a design pattern and a framework, indicated where each is used. Also spend some time discussion the sidebar “Creational, Structural and Behavioral patterns” indicating how each might be used in a familiar application.

Use Figure 16.1 as an introduction to pattern-based design with particular emphasis extracting the problem, context and forces from the requirements model. Go through Shalloway and Trotts approach to “thinking in patterns” and if time permits, expand on the brief SafeHome example that follows it.

The design tasks presented in Section 16.2.3 should be considered within the context of an example. It might be worth discussing the pattern organizing table as a way to structure requirements into a series of problems that are amenable to pattern-based design. Also, be sure to cover the common design mistakes noted in Section 16.2.5.

The design patterns repositories noted in the sidebar offer a wide variety of examples and should be explored by your students.

It’s important to emphasize that design patterns are developed at many different levels of abstraction. Be sure you discuss design focus (Section 16.6.1) and design granularity.

**Chapter 17 — WebApp Design**

When most people think of “Web design,” they envision the aesthetic layout of a WebApp. The intent of this chapter is to introduce students to the other important components of WebApp design.

Because students are so familiar with the Web, you should have little trouble getting a discussion of design started. Virtually everyone has an opinion on what constitutes a “good design” but few people understand the process through which a good design is achieved.

The quality requirements shown in Figure 17.1 are a good catalyst for a discussion of WebApp design quality. Each of the five attributes presented in the figure should be discussed at some length. Poll your students for their interpretation of each attribute and ask for examples of actual WebApp that exhibit (or don’t exhibit) each attribute. Spend a bit of time on the quality checklist presented in the sidebar.

The design goals should be discussed at some length. Again, examples of WebApps that meet (and don’t meet) these goals will help to solidify understanding.

WebE design encompasses six design activities, and each of these is introduced in a discussion of Figure 17.2.

**WebApp Interface Design**

If you haven’t already done so, have your students review Chapter 15. All of the information presented there is applicable to this discussion.

The interface design workflow is worth covering in class. Be sure that your students understand the intent of each step. In many cases, reference is made to a specific UML diagram. It may be necessary to review these.

**Aesthetic Design**

Although this is an important topic, you may choose to cover it only peripherally if time is short. For major WebApps, aesthetic design is performed by graphic arts professional, not Web engineers. However, for smaller WebApps, the Web engineer may have to do it all. I recommend spending time on this subject to sensitize your students to the importance of aesthetics. At a minimum, have them visit one or more of the Web sites noted in the sidebar entitled “Well-Designed Web Sites.”

**Content Design**

Although the notion of a content object (Section 17.6.1) should be relatively easy for your students to grasp, be certain that you provide additional examples to solidify the concept.

**Architecture Design**

If you haven’t already done so, have your students review Chapter 13. Much of the information presented there is applicable to this discussion.

The content structures presented in Section 17.7.1 should be covered during lecture. Use actual WebApps to illustrate these structures.

The MVC architecture (Section 17.7.2) has been discussed widely in the literature and should be emphasized during your discussion of architectural design. Use Figure 17.8 as a point of departure for your comments.

**Navigation Design**

Navigation design is pivotal to the success of a WebApp and yet, navigation for many Web sites sort of “just happens.” Emphasize to your students that navigation design should be explicit when complex navigation pathways are present.

Students will be quite familiar with navigation syntax and will undoubtedly have strong opinions on the best design approach in this area. Be certain to tie syntax to semantics, indicating how syntax can aid or hurt a user’s understanding of navigation.

If you haven’t already done so, have your students review Chapter 14. Much of the information presented there is applicable to component design for WebApps..

**Object-Oriented Hypermedia Design Method (OOHDM)**

OOHDM is a reasonably comprehensive design method that is well-worth covering if time permits. I If you intend to present this topic in some detail, you’ll need supplementary materials and reading. See the SEPA website for OOHDM resources.

**Chapter 18 — MobileApp Design**

Students will not appreciate all the nuances of designing a MobileApp until they have built one of their own. Many mobile development tools are available as free downloads for educational use (see the web links in this chapter. Many courses will not have enough time to teach students how to use more than one development environment and may not have access to physical devices (needing to rely on simulators instead).

MobileApp design encompasses technical and nontechnical activities that include: establishing the look and feel of the mobile application, creating the aesthetic layout of the user interface, establishing the rhythm of user interaction, defining the overall architectural structure, developing the content and functionality that reside within the architecture, and planning the navigation that occurs within the MobileApp. Having students create paper prototypes of a mobile app user interface can be a good a good exercise. Special attention needs to be given to the elements that add context awareness to the MobileApp.

MobileApp design is similar to WebApp design in that it encompasses six major steps that are driven by information obtained during requirements modeling. Content design is critically important and takes the screen and other limitations of mobile devices into account. Aesthetic design, also called graphic design, describes the “look and feel” of the MobileApp and includes color schemes; graphic layout; the use of graphics; and related aesthetic decisions. Aesthetic design must also take device limitations into account

Architecture design identifies the overall hypermedia structure for the MobileApp and encompasses both content architecture and MobileApp architecture. It is critical to determine how much of the MobileApp functionality will reside on the mobile device and how much will be provided by Web or cloud services.

Interface design describes the structure and organization of the user interface and includes a representation of screen layout, a definition of the modes of interaction, and a description of navigation mechanisms. In addition, the interface for a good MobileApp will promote the brand signature and focus on its targeted device platform(s). A set of core user stories is used to trim unnecessary features from the app to manage its resources requirements.

Context-aware devices make use of discoverable services to help personalize the user experience. Navigation design represents the navigational flow between content objects and for all MobileApp functions. Navigation syntax is defined by the widgets available on the targeted mobile device(s) and the semantics are often determined by the mobile platform. Content chunking must take intermittent service interruptions into account and user demands for fast performance.

Component design develops the detailed processing logic required to implement the components that are used to build a complete MobileApp function. The design techniques described in Chapter 14 may be applicable for the engineering of MobileApp components.

Students might be encouraged to use the MobileApp quality checklist in Section 18.2.1 to evaluate an assigned MobileApp. Similarly students might be challenged to look for examples illustrating the MobileApp design mistakes listed in Section 18.2.4 and suggest ways to correct them.

**Part 3—Quality Management**

**Chapter 19 — Quality Concepts**

Part 3 of SEPA focuses on a broad spectrum of quality management activities. In one way or another each of the chapters in this Part focus on techniques for achieving and/or ensuring high quality software. The intent of this chapter is to define software quality from a number of different perspectives. The operative definition used in Section 19.2 is: An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it. It would be worth parsing this statement in some detail, examining each phrase and what it means within the context of software engineering work.

An important concept is that controlling variation among products is what quality assurance work is all about. Software engineers are concerned with controlling the variation in their processes, resource expenditures, and the quality attributes of the end products. The definitions of many quality concepts appear in this section. Students need to be familiar with these definitions, since their use in software quality work does not always match their use in casual conversation. Students also need to be made aware that customer and user satisfaction is every bit as important to modern quality work as is quality of design and quality of conformance.

Spend time discussing the indirect cost of quality. That is, the costs associated with customer dissatisfaction, increased support and reduction in internal morale.

If time permits, you might have your student read excepts from Crosby’s Quality is Free or Persig’s Zen and the Art of Motorcycle Maintenance. Each contains many useful insights on quality.

The next subsections provide a variety of quality dimensions and factors. In addition to discussing some subset of these, it’s also worth noting that software quality factors are timeless—those that were applicable 30 years ago are still applicable today, even though technologies have changed dramatically.

The discussion of “good enough” software is vitally important in world where new versions are released regularly. Be sure to emphasis the pros and cons of this approach and poll your students to determine their feeling about “good enough.” Also be sure to address the business realities associated with the philosophy. Use a major software vendor as an example.

A discussion of the cost of quality is a natural follow-on to “good enough.” It might be worth developing a quantitative example so that students better understand that “quality” translates into dollar and cents issues and that unexpected costs often accrue. Be sure to discuss the relative cost of correcting errors (Figure 19.2). Also, be certain to discuss the liability issues.

Section 19.4 can serve as an introduction to the chapters that follow in Part 3 and also part 4 of SEPA.

**Chapter 20 — Review Techniques**

The intent of this chapter is to provide students with an indication of why reviews are an important quality assurance mechanism and how reviews can be conducted as part of a software project. If your course requires a team project, this chapter is essential and should be presented early in the course.

The defect amplification model can serve as a interesting team project. The model itself can be forms driven, with added sophistication and input, to allow users to model the effectiveness, cost and impact of reviews on the software process. The review metrics discussed in Section 20.3 can be integrated into the model and output similar to Figure 20.4 might be appropriate.

The sidebar discussing bugs, errors, and defects is worth discussing in class. Our approach to these terms is somewhat different that the mainstream (which treats them all synonymously. At the risk of hubris, we really do believe that the distinctions I make are important and that they should be considered.

The reviews reference model (Figure 20.5) should be used to establish a mode of comparison for various review types. Have your students add to the review checklist presented in the text and suggest that they create a set of review guidelines for their team project.

In addition to the review checklists contained within the SEPA Web site, we have also included a small sampler in the special section that follows.

**Review Checklists**

Formal technical reviews can be conducted during each step in the software engineering process. In this section, we present a brief checklist that can be used to assess products that are derived as part of software development. The checklists are not intended to be comprehensive, but rather to provide a point of departure for each review.

**System Engineering**. The system specification allocates function and performance to many system elements. Therefore, the system review involves many constituencies that may each focus on their own area of concern. Software engineering and hardware engineering groups focus on software and hardware allocation, respectively. Quality assurance assesses system level validation requirements and field service examines the requirements for diagnostics. Once all reviews are conducted, a larger review meeting, with representatives from each constituency, is conducted to ensure early communication of concerns. The following checklist covers some of the more important areas of concern.

1. Are major functions defined in a bounded and unambiguous fashion?
2. Are interfaces between system elements defined?
3. Have performance bounds been established for the system as a whole and for each element?
4. Are design constraints established for each element?
5. Has the best alternative been selected?
6. Is the solution technologically feasible?
7. Has a mechanism for system validation and verification been established?
8. Is there consistency among all system elements?

**Software Project Planning**. Software project planning develops estimates for resources, cost and schedule based on the software allocation established as part of the system engineering activity. Like any estimation process, software project planning is inherently risky. The review of the Software Project Plan establishes the degree of risk. The following checklist is applicable.

1. Is software scope unambiguously defined and bounded?
2. Is terminology clear?
3. Are resources adequate for scope?
4. Are resources readily available?
5. Have risks in all important categories been defined.
6. Is a risk management plan in place?
7. Are tasks properly defined and sequenced? Is parallelism reasonable given available resources?
8. Is the basis for cost estimation reasonable? Has the cost estimate been developed using two independent methods?
9. Have historical productivity and quality data been used?
10. Have differences in estimates been reconciled?
11. Are pre-established budgets and deadlines realistic?
12. Is the schedule consistent?

**Software Requirements Analysis**. Reviews for software requirements analysis focus on traceability to system requirements and consistency and correctness of the analysis model. A number of FTRs are conducted for the requirements of a large system and may be augmented by reviews and evaluation of prototypes as well as customer meetings. The following topics are considered during FTRs for analysis:

1. Is information domain analysis complete, consistent and accurate?

2. Is problem partitioning complete?

3. Are external and internal interfaces properly defined?

4. Does the data model properly reflect data objects, their attributes and relationships.

5. Are all requirements traceable to system level?

6. Has prototyping been conducted for the user/customer?

7. Is performance achievable within the constraints imposed by other system elements?

8. Are requirements consistent with schedule, resources and budget?

9. Are validation criteria complete?

**Software Design**. Reviews for software design focus on data design, architectural design and procedural design. In general, two types of design reviews are conducted. The preliminary design review assesses the translation of requirements to the design of data and architecture. The second review, often called a design walkthrough, concentrates on the procedural correctness of algorithms as they are implemented within program modules. The following checklists are useful for each review:

**Preliminary design review**

1. Are software requirements reflected in the software architecture?

2. Is effective modularity achieved? Are modules functionally independent?

3. Is the program architecture factored?

4. Are interfaces defined for modules and external system elements?

5. Is the data structure consistent with information domain?

6. Is data structure consistent with software requirements?

7. Has maintainability considered?

8. Have quality factors (section 17.1.1) been explicitly assessed?

**Design walkthrough**

1. Does the algorithm accomplishes desired function?

2. Is the algorithm logically correct?

3. Is the interface consistent with architectural design?

4. Is the logical complexity reasonable?

5. Have error handling and "anti-bugging" been specified?

6. Are local data structures properly defined?

7. Are structured programming constructs used throughout?

8. Is design detail amenable to implementation language?

9. Which are used: operating system or language dependent features?

10. Is compound or inverse logic used?

11. Has maintainability considered?

**Coding**. Although coding is a mechanistic outgrowth of procedural design, errors can be introduced as the design is translated into a programming language. This is particularly true if the programming language does not directly support data and control structures represented in the design. A code walkthrough can be an effective means for uncovering these translation errors. The checklist that follows assumes that a design walkthrough has been conducted and that algorithm correctness has been established as part of the design FTR.

1. Has the design properly been translated into code? [The results of the procedural design should be available during this review.]

2. Are there misspellings and typos?

3. Has proper use of language conventions been made?

4. Is there compliance with coding standards for language style, comments, module prologue?

5. Are there incorrect or ambiguous comments?

6. Are data types and data declaration proper?

7. Are physical constants correct?

8. Have all items on the design walkthrough checklist been re-applied (as required)?

**Software Testing**. Software testing is a quality assurance activity in it own right. Therefore, it may seem odd to discuss reviews for testing. However, the completeness and effectiveness of testing can be dramatically improved by critically assessing any test plans and procedures that have been created. In the next two Chapters, test case design techniques and testing strategies are discussed in detail.

**Test plan**

1. Have major test phases properly been identified and sequenced?

2. Has traceability to validation criteria/requirements been established as part of software requirements analysis?

3. Are major functions demonstrated early?

4. Is the test plan consistent with overall project plan?

5. Has a test schedule been explicitly defined?

6. Are test resources and tools identified and available?

7. Has a test record keeping mechanism been established?

8. Have test drivers and stubs been identified and has work to develop them been scheduled?

9. Has stress testing for software been specified?

**Test procedure**

1. Have both white and black box tests been specified?

2. Have all independent logic paths been tested?

3. Have test cases been identified and listed with expected results?

4. Is error-handling to be tested?

5. Are boundary values to be tested?

6. Are timing and performance to be tested?

7. Has acceptable variation from expected results been specified?

In addition to the formal technical reviews and review checklists noted above, reviews (with corresponding checklists) can be conducted to assess the readiness of field service mechanisms for product software; to evaluate the completeness and effectiveness of training; to assess the quality of user and technical documentation, and to investigate the applicability and availability of software tools.

**Maintenance**. The review checklists for software development are equally valid for the software maintenance phase. In addition to all of the questions posed in the checklists, the following special considerations should be kept in mind:

1. Have side effects associated with change been considered?

2. Has the request for change been documented, evaluated and approved?

3. Has the change, once made, been documented and reported to interested parties?

4. Have appropriate FTRs been conducted?

5. Has a final acceptance review been conducted to ensure that all software has been properly updated, tested and replaced?

**Chapter 21 — Software Quality Assurance**

This chapter provides an introduction to software quality management and software quality assurance (SQA). It is important to have the students understand that software quality work begins before the testing phase and continues after the software is delivered. The role of metrics in software management is reinforced in this chapter.

The elements of SQA presented in Section 21.2 should be discussed in order to define the broad spectrum of issues and activities that contribute to quality management. You should encourage your students (a homework assignment might be the best method of “encouragement” to visit the Quality Management Resources noted in the sidebar.

Most students will be unfamiliar with the organizational placement and role of SQA and its relationship with the software team. Be certain to spend some time discussing alternatives and the pros and cons of each (e.g., the software team does its own SQA).

The concept of formal methods in software engineering will be new to most students. More comprehensive discussions of formal specification techniques and formal verification of software appear Chapters 28.

A comprehensive discussion of statistical quality assurance is beyond the scope of a software engineering course. However, a high level description of the process and gives examples of metrics that might be used in this type of work is worth presenting in class. The key points to emphasize to students are that each defect needs to be traced to its cause and that defect causes having the greatest impact on the success of the project must be addressed first. Because six-sigma is widely used in industry, you might spend some lecture time on it.

Software reliability is discussed in Section 21.7. It is important to have the students distinguish between software consistency (repeatability of results) and reliability (probability of failure free operation for a specified time period). Students should be made aware of the arguments for and against applying hardware reliability theory to software (e.g. a key point is that, unlike hardware, software does not wear out so that failures are likely to be caused by design defects). It is also important for students to be able to make a distinction between software safety (identifying and assessing the impact of potential hazards) and software reliability.

Software safety and hazard analysis become increasingly important as software based systems become pervasive. The information presented in this chapter is rudimentary, at best. If time permits, introduce supplementary source material.

The ISO 9000 quality standard is discussed in Section 21.8 as an example of quality model that is based on the assessment of quality of the individual processes used in the enterprise as a whole. ISO 9001:2000 is described as the quality assurance standard that contains 20 requirements that must be present in any software quality assurance system.

The major sections of a SQA plan are described in Section 21.9. It would advisable to have students write a SQA plan for one of their own projects sometime during the course. This will be a difficult task for them. It may be advisable to assign this task after student complete all of Part 3.

**Chapter 22 — Software Testing Strategies**

This chapter discusses a strategic approach to software testing that is applicable to most software development projects. The recommended process begins unit testing, proceeds to integration testing, then validation testing, and finally system testing. You should emphasize the spiral—we believe it is a useful metaphor for the software engineering process and the relationship of testing steps to earlier definition and development activities.

The majority of testing that most students have done has been *ad hoc.* Therefore, the key concept for students to grasp is that testing must planned and assessed for quality like any other software engineering process. Students should use the *Test Specification* template from the SEPA website as part of their term project.

Section 22.1 describes testing as a generic process that is essential to developing high quality software economically. It is important for students to understand the distinction between verification (building a product correctly) and validation (building the right product). It is also important for students to be aware that testing cannot replace attention to quality during early software engineering activities.

The role of software testing groups in egoless software development is another important point to stress with students.

Section 22.1.4 discusses the issue of how to determine when testing has been completed. Which is an important issue to consider, if students buy into the argument that it is impossible to remove all bugs from a given program. This issue provides an opportunity to reconsider the role of metrics in project planning and software development.

Several testing issues are introduced in Section 22.2. Planning is described as being an important part of testing. Students may need assistance in learning to write testing objectives that cover all portions of their software projects. Formal technical reviews of test plans and testing results are discussed as means of providing oversight control to the testing process. Students should be encouraged to review each other’s testing work products some time during the semester.

Testing for conventional software begins “in the small” and moves toward “testing “in the large.” Your students should understand the “big picture” reasons for this strategy. You might also discuss the “daily build and smoke test” strategy that is used by many software product builders and is an of encountered in agile process models.

Both black-box and white-box testing techniques have roles in testing individual software modules. It is important to emphasize that the white-box techniques to be introduced in Chapter 23 are most advantageous during this testing step. Students need to be aware that testing module interfaces is also a part of unit testing. Students need to consider of the overhead incurred in writing drivers and stubs required by unit testing. This effort must be taken into account during the creation of the project schedule. This section also contains lists of common software errors. Students should be encouraged to keep these errors in mind when they design their test cases.

**S**ection 22.3.2 focuses on integration testing issues. Integration testing often forms the heart of the test specification document. Don't be dogmatic about a "pure" top down or bottom up strategy. Rather, emphasize the need for an approach that is tied to a series of tests that (hopefully) uncover module interfacing problems. Be sure to discuss the importance of software drivers and stubs (as well as simulators and other test software), indicating that development of this "overhead" software takes time and can be partially avoided with a well thought out integration strategy. Regression testing is an essential part of the integration testing process. It is very easy to introduce new module interaction errors when adding new modules to a software product. It may be wise for students to review the role of coupling and cohesion in the development of high quality software. Don't gloss over the need for thorough test planning during this step, even if your students won't have time to complete any test documentation as part of their term projects.

Section 22.4 clarifies the differences between OOT and conventional testing with regard to unit testing and integration testing. The key point to unit testing in an OO context is that the lowest testable unit should be the encapsulated class or object (not isolated operations) and all test cases should be written with this goal in mind. Given the absence of a hierarchical control structure in OO systems integration testing of adding operators to classes is not appropriate. Students should try writing an integration test plan for an OOD based on one of the three strategies described in this section (thread-based testing, use-based testing, and cluster testing). Similarly, students should try to write a plan for validating an OOD based on the use-case scenarios defined in the OOA model.

Be sure to introduce the 10 steps associated with WebApp testing now to foreshadow a more detailed discussion in Chapter 25.

Section 22.7 describes validation testing as the last chance to catch program errors before delivery to the customer. Since the focus is on testing requirements that are apparent to the end-users, students should regard successful validation testing as very important to system delivery. If the users are not happy with what they see, the developers often do not get paid. It is sometimes worthwhile to have students test each other’s software for conformance to the explicitly stated software requirements. The key point to emphasize is traceability to requirements. In addition, the importance of alpha and beta testing (in product environments) should be stressed.

System testing is described as involving people outside the software engineering group (since hardware engineers, system engineers or network engineers are often involved). Several systems tests are mentioned in this section (recovery, security, stress, and performance). Students should be familiar with each of them.

A thorough discussion of the problems associated with "finger pointing," possibly with excerpts from Tracy Kidder's classic book, *The Soul of a New Machine,* will provide your students with important insight.

The “art of debugging” reviews the process of debugging a piece of software. Students may have seen this material in their programming courses. The debugging approaches might be illustrated by using each to track down bugs in real software as part of a class demonstration or laboratory exercise. Students need to get in the habit of examining the questions at the end of this section each time they remove a bug from their own programs.

To emphasize how luck, intuition, and some innate aptitude contribute to successful debugging, conduct the following class experiment:

1. Handout a 30 -50 line module with one or more semantic errors purposely embedded in it.

2. Explain the function of the module and the symptom that the error produces.

3. Conduct a "race" to determine:

a) error discovery time

b) proposed correction time

4. Collect timing results for the class; have each student submit his or her proposed correction and the clock time that was required to achieve it.

5. Develop a histogram with response distribution.

It is extremely likely that you find wide variation in the students' ability to debug the problem.

**Chapter 23 —Testing Conventional Applications**

This intent of this chapter is to introduce a variety of black-box and white-box testing methods that can be used for both conventional software. The vast majority of students will be unaware of even the most simple test case design method, viewing testing as an afterthought—something that has to be done after coding. Students should be encouraged to design and build test cases for their course projects, using several of the testing techniques presented here.

Students need to be encouraged to look at testing as an essential part of the quality assurance work and a normal part of modern software engineering. Technical reviews by themselves cannot locate all software errors. Testing occurs late in the software development process and is the last chance to catch bugs prior to customer release. This section contains a software testability checklist that students should keep in mind while writing software and designing test cases to test software. The toughest part of testing for students is understanding the necessity of being thorough, and yet recognizing that testing can never prove that a program is bug free.

**C**onvince students that exhaustive testing is not possible for most real applications (too many logic paths and two many input data combinations). This means that the number of test cases processed is less important than the quality of the test cases used in software testing. Sections that follow discuss strategies that will help students to design test cases that will make both white-box and black-box testing feasible for large software systems.

Section 23.3 makes the case that white-box testing is important, since there are many program defects (e.g. logic errors) that black-box testing can not uncover. Students should be reminded that the goal of white-box testing is to exercise all program logic paths, check all loop execution constraints, and internal data structure boundaries.

Basis path testing is presented as an example of a white-box testing technique. Basis path testing is easiest for students to use if they will construct a program flow graph first. However, students should understand that cyclomatic complexity could be computed from the PDL representation of the program (or from source code itself). Be sure your students understand what an “independent path” is and why there are a limited number of them (as opposed to an extremely large number of program paths).

Students should be encouraged to use the basis path example as a model and construct a set of test cases for one of their own programs. The term "graph matrix" is introduced in this section; students might have studied these as adjacency matrices in a discrete mathematics or data structures unit on graph theory. If your students are unfamiliar with graph theory, you may need to show them more examples of how to construct adjacency matrices with various types of graph edge weights.

Basis path testing is one form of control structure testing. Section 23.5 introduces three others (condition testing, data flow testing, loop testing). The argument given for using these techniques is that they broaden the test coverage from that which is possible using basis path testing alone. Showing students how to build truth tables may be beneficial to ensure thorough coverage by the test cases used in condition testing. Students may need to see an example of building test cases for data flow testing using a complete algorithm implemented in a familiar programming language. Similarly students may benefit from seeing examples of building test cases for each of the loop types listed in Section 23.5.3. Students should be required to build a set of test cases to do control structure testing of one of their own programs sometime during the semester.

The purpose of black-box testing is to devise a set of data inputs that fully exercise all functional requirements for a program. Students should be reminded that black-box testing is complementary to white-box testing. Both are necessary to test a program thoroughly. Several black-box testing techniques are introduced in this section (graph-based testing, equivalence partitioning, boundary value analysis, comparison testing, orthogonal array testing).

It is important to emphasize that in black-box testing the test designer has no knowledge of algorithm implementation. The test cases are designed from the requirement statements directly, supplemented by the test designer's knowledge of defects that are likely to be present in modules of the type being tested. It may be desirable to show students the process of building test cases from an actual program's requirements using several of these techniques. A worthwhile activity for students is devising test cases for another student's program from the software specification document without seeing the program source code.

**Chapter 24 — Testing Object-Oriented Applications**

The intent of this chapter is to complement and extend the discussion of conventional software testing to the object-oriented domain. Test case design for OO software is directed more toward identifying collaboration and communication errors between objects, than toward finding processing errors involving input or data like conventional software testing.

Students should spend some time discussing the differences between testing the surface structure (end-user view) and deep structure (implementation view) of an OO system.

Be sure to emphasize the unique characteristics of unit and integration testing when OO systems are considered. The discussion of methods for evaluating the consistency of OO models (Section 24.2.2) will prove useful and should be addressed in class.

Testing Methods Applicable at the Class Level discusses the process of testing at the individual class level. Students should be reminded of the haphazard nature of random testing and be urged to consider using the three operation partitioning techniques (state-based, attribute-based, category-based) to improve the efficiency of their testing efforts.

Interclass Test Case Design discusses the task of interclass test case design. Two techniques for conducting interclass testing are described (multiple class testing and tests derived from behavioral models). Students might be encouraged to develop a set of test cases for a real system using each technique and see which they prefer using.

**Chapter 25 — Testing Web Applications**

Of all Web engineering tasks, testing is arguably the most important. At the same time, many practitioners have a relatively weak understanding of WebApp testing and as a result, WebApp testing is conducted poorly. That is a cause for concern.

The intent of this chapter is to introduce the important elements of testing for WebApps. Both testing strategy and tactics are considered in this chapter.

There has been much discussion of quality attributes and concepts throughout SEPA. I think it’s a very important topic. However, students may begin to roll their eyes when the subject is introduced yet again in this section. If time permits, it would be worthwhile to tie the “dimensions of quality” presented in Section 25.1.1 with other quality discussions presented in Chapters 12, 19, and 21. Where are the differences? Where are there similarities?

Section 25.1.2 discusses the characteristics of WebApp errors. Emphasize to your students that they must understand the nature of WebApp errors before they can hope to design tests to uncover those errors.

Be certain to emphasize the testing strategy discussed in Section 25.1.3 and the need to develop a test plan for any major WebApp.

Use Figure 25.1 as a point of departure for your discussion of the WebApp testing process. This section serves as a TOC for sections that follow.

**Content Testing**

Spend some time discussing the questions posed in Section 25.3.1. Ask your students what types of “tests” they would design to answer each of these questions.

Database testing (Section 25.3.2) is an advanced topic and may be too specialized for inclusion in an introductory course. However, if time permits, an overview of the key issues is recommended. Use Figure 25.2 as a point of departure for your discussion.

**User Interface Testing**

The interface testing strategy presented in Section 25.4.1 should be emphasized during lecture. It is important to note that it is sometimes difficult to make a clear distinction between interface testing, usability testing, and even navigation testing. Interface testing attempts to find errors in the syntax or semantics of user interaction.

Interface mechanics (syntax) is tested by examining the interface mechanisms discussed in Section 25.4.2. Interface semantics (Section 25.4.3) examine how the interface achieved required user functionality and features. Usability tests address issues presented in Section 25.4.4. Use Figure 25.3 as a trigger for this discussion.

**Component-Level Testing**

Component level testing uses techniques presented in Chapters 23 and 24. You may want to revisit black-box and white-box techniques as part of this discussion.

**Navigation Testing**

Like interface testing, navigation testing attempts to find errors in the syntax or semantics of navigation. Discuss each of the navigation mechanisms noted in Section 25.6.1 and ask your student how they might test each in a generic sense.

The NSU is the driver for testing navigation syntax, a topic discussed in Section 25.6.2. Review the questions posed in this section.

**Configuration Testing**

It might be best to begin this discussion by considering the vagaries of an Internet-based client/server environment. The discussion presented in this section is fairly rudimentary, focusing on configuration compatibility issues on both the client and server sides.

If your students have a solid background in client/server architectures, you might expand this discussion (as time permits) to cover more advanced topics.

**Security Testing**

We only present an overview of this topic in SEPA. It’s likely that your students will find this fascinating and if time permits, you might want to extend coverage a bit.

If you intend to spend some time here, you’ll need to supplement SEPA content with outside sources. See the SEPA Web site and SEPA Chapter 27 for resource recommendations.

**Performance Testing**

Performance testing addresses the questions posed in the introduction to Section 25.9.1. Specific testing methods—load testing and stress testing—are conducted to answer these questions. Be certain that your student understand the subtle difference between load and stress testing and the intent of each.

If time permits, have your students investigate one or more testing tools suggested in the sidebar.

**Chapter 26 — Testing Mobileapps**

The goal of MobileApp testing is to exercise each quality dimension of a MobileApp with the intent of finding errors or uncovering issues that may lead to failures. Testing incorporates reviews and usability assessments that occur as the MobileApp is designed as well as tests that are conducted once the MobileApp has been deployed on actual device. A MobileApp test plan is developed and identifies testing steps, work products (e.g., test cases), and mechanisms for the evaluation of test results.

The MobileApp testing strategy begins by examining “units” of content, functionality, or navigation. Once individual units have been validated, the focus shifts to tests that exercise the MobileApp as a whole. To accomplish this, many tests are derived from the user’s perspective and are driven by information contained in use cases similar to the object-oriented testing.

Students might be encouraged to take the user stories created for the Three-Ring described in the Chapter 18 problems and create test cases for them. You might also suggest that they create a paper prototype to facilitate testing before programming the app. It may be worthwhile to discuss the benefits and limitations of paper prototype testing for mobile apps.

Content testing (and reviews) focus on various categories of content. The intent is to examine errors that affect the presentation of the content to the end user. The content needs to be examined for performance issues imposed by the mobile device constraints. Interface testing exercises the interaction mechanisms that define the user experience provided by the MobileApp. The intent is to uncover errors that result when the MobileApp does not take device, user, or location context into account.

You might have students create a generic content and usability check lists for mobile apps and then use their check list to assess an app they are familiar with. Students might be asked to use some of the automated test tools suggested in this chapter and see if an existing web site is mobile app ready. Ask them to create a list of defects and suggest corrections.

Navigation testing is based on use cases and derived as part of the modeling activity. The test cases are designed to exercise each usage scenario against the navigation design within architectural framework used to deploy the MobileApp. Component testing exercises content and functional units within the MobileApp.

Configuration testing attempts to uncover errors and/or compatibility problems that are specific to a particular device or network environment. Tests are then conducted to uncover errors associated with each possible configuration. This is complicated by the large number or mobile devices and network service providers. This is why MobileApps designed for global markets need to be tested in the wild.

Have students create a test plan based on the certification requirements for a specific mobile app store. It may be useful to have a class discussion where the certification requirements for two different app stores are compared with thought as to how to devise test plans for the same app developed for each of the devices.

Security testing incorporates a series of tests designed to exploit vulnerabilities in the MobileApp or its environment. The intent is to find security holes in either the device operating environment or the web services being accessed. Security testing is discussed in greater detail in Chapter 27.

Performance testing encompasses a series of tests that are designed to assess MobileApp response time and reliability as demands on server-side resource capacity increase. Finally, MobileApp testing should address performance issues as power usage, processing speed, memory limitations, ability to recover from failures, and connectivity issues. The role of software testing tools and and device emulators should be emphasized.

Having students devise a set of test cases and the procedures necessary to examine performance issues without adding instrumentation to the mobile device is a good exercise. They might compare the results from conducting the same tests using a mobile device and an emulator running on a PC.

**Chapter 27 — Security Engineering**

Software security engineering is concerned with developing software that protects the assets that it manages from threats. Threats may involve attacks that exploit system vulnerabilities to compromise the confidentiality, integrity, or availability of system services or data.

The material in this chapter may benefit from having the students look at the material on Risk Management discussed in Chapter 35. Risk analysis is similar to process used security threat modeling. Security risk management is concerned with assessing the impact of possible threats and deriving security requirements to minimize critical loses.

Have students identify attack patterns that are common for a specific mobile device. Ask them to consider design process elements that could be used to build in defenses to these attacks.

The critical point to get across to students is that it is impossible to anticipate every possible security threat. It is important to make the efforts made to defend against a threat proportional to the cost of losing or replacing the system assets affected.

Have students list the assets at risk to the user of a mobile banking app and determine threats to these assets. Have students try to write abuser stories detailing each threat. You might ask students to model the abuser stories using finite state machines and write representative test cases to test one of the abuser stories.

Design for security involves creating a system architecture that minimizes the introduction of known vulnerabilities. Software engineers should make use of techniques to prevent attacks, to repel attacks, and to recover from attacks as a means to mitigating the effects of losses.

After looking at the material in Chapter 35. Have students try to develop a threat model for at least security concern for the YourCornerPharmacy app discussed in the Chapter 26 problems. Use the process presented in Section 27.5 and devise a threat mitigation strategy to preserve high value assets.

Inspiring trust among stakeholders requires that developers regard security assurance as an umbrella activity that is present at the beginning throughout the software process. The development of security metrics is still in its infancy. Building a security case for a system involves evidence collected using security testing, security-focused formal technical reviews, and inspections to ensure security guidelines and mitigation practices are being followed. You may wish to use a case study which illustrates the security assurance process and how a security risk analysis might be performed,

**Chapter 28 — Formal Modeling and Verification**

The late Harlan Mills (one of the true giants of the first half century of computing) suggested that software could be constructed in a way that eliminated all (or at least most) errors *before* delivery to a customer. He argued that proper specification, correctness proofs, and formal review mechanisms could replace haphazard testing, and as a consequence, very high quality computer software could be built. The intent of this chapter is to present two approaches that take a more formal approach to program verification cleanroom and formal methods.

The cleanroom software engineering strategy introduces a radically different paradigm for software work. It emphasizes a special specification approach, formal design, correctness verification, “statistical” testing, and certification as the set of salient activities for software engineering. The chapter begins by introducing box structure specification and a more rigorous approach for representing the analysis model. Next define refinement of the box structure specification is presented, followed by the correctness proofs that can be applied to the specification to verify that it is correct. The cleanroom approach to testing is radically different that more conventional software engineering paradigms. The culmination of this chapter is to emphasize the cleanroom testing approach.

The key concepts for students to understand are boxes, formal verification, probability distributions, and usage based testing. The mathematics and statistical background needed to read the chapter is not overwhelming. However, if you want to have your students do some cleanroom software development, you may need to do some additional teaching on program verification and statistical sampling.

**Functional Specification**

Functional specification using boxes is the focus of this section. It is important for students to understand the differences between black boxes (specifications), state boxes (architectural designs), and clear boxes (component designs). Even if students have weak understanding of program verification techniques, they should be able to write box specifications for their own projects using the notations shown in this section.

**Cleanroom Design**

If you plan to have your students verify their box specifications formally, you may need to show them some examples of the formal methods techniques used later in this chapter. The key to making verification accessible to students at this level is to have them write procedural designs using only structured programming constructs in their designs. This will reduce considerably the complexity of the logic required to complete the proof. It is important for students to have a chance to consider the advantages offered by formal verification over exhaustive unit testing to try to identify defects after the fact.

**Cleanroom Testing**

This section provides and overview of statistical use testing and increment certification. It is important for students to understand that some type of empirical data needs to be collected to determine the probability distribution for the software usage pattern. The set of test cases created should reflect this probability distribution and then random samples of these test cases may be used as part of the testing process. Some additional review of probability and sampling may be required. Students would benefit from seeing the process of developing usage test cases for a real software product. Developing usage test cases for their own projects will be difficult, unless they have some means of acquiring projected usage pattern data. Certification is an important concept. Students should understand the differences among the certification models presented in this section as well.

**Formal Methods**

This section discusses the benefits of using formal specification techniques and the weaknesses of informal specification techniques. Many of the concepts of formal specification are introduced (without mathematics) through the presentation of three examples showing how formal specifications would be written using natural language. It may be worthwhile to revisit these examples after students have completed the chapter and have them write these specifications using mathematical notation or a specification language (like OCL or Z).

This section uses mathematical notation to refine the block handler specification from Section 28.6. It may be desirable to refine the other two specification examples from If your students are comfortable with mathematical proofs, you may wish to present an informal correctness proof for these three specifications. Having students write specifications for some of their own functions, using notation similar to that used in this section may be desirable.

**Chapter 29 — Software Configuration Management**

This chapter presents an introduction to the issues associated with change management and SCM. Version control and change control are emphasized. It will not be easy to convince students of the importance of these topics, unless your course happens to include a term long project that involves interaction with a real customer. Most introductory software engineering courses are not set up this way. One way to reinforce the importance of configuration management (without a real customer) would be to change the project requirements sometime after students have begun its implementation. However, to help students maintain their sanity (these courses do tend to be a lot of work!), We can’t recommend this in good conscience.

**Software Configuration Management**

Section 29.1.1 presents a useful SCM scenario that will help students to understand the elements of change management. Be certain to discuss it in class.

The elements of a configuration management system are discussed in Section 29.1.2. If time permits, you might consider expanding the discussion to provide detailed examples of each element.

Baselines and software configuration items are presented in Sections 29.1.3 and 29.1.4, respectively. Students need to be aware that changes can occur to any SCI during the course of a project and the degree of rigor applied to change control is tied to whether or not an SCI has been baselined.

The impact of changes can have a negative ripple effect throughout a project, if they are not controlled properly. The concept of a baseline configuration item as being something that has very tightly defined change procedures is important for students to understand.

**The SCM Repository**

The characteristics of the SCM repository are discussed in this section. Key points to emphasize are: (1) that the repository is a database; (2) that it must be integrated with tools and with the SCM process; (3) that it should possess or be integrated with tools to provide the features noted in Section 29.2.2.

**The SCM process**

It is important for students to be able to distinguish between the terms version and variant. It is also important for students to get in the habit of incorporating the version number in the naming scheme used for their SCI's.

Section 29.3.1 advocates the use of an object-oriented approach to managing the software configuration items once they are identified. Students who are unfamiliar with the concepts of object-oriented programming and/or the use of ERDs may struggle with the concepts introduced here. It may be desirable to review some of the concepts from Chapters 9/10 and/or Appendix II while covering this section.

Version control (Section 29.3.2) is arguably the most important SCM function for software product builders. Be certain to discuss the mechanics of version control. If time permits, have your students research one of the tools noted in the sidebar in this section and present their findings to the class.

Change control (Section 29.3.3) is the SCM task that helps to avoid chaos within a software project. The details of processing and controlling a change request for an SCI are discussed. Students need to be familiar with each of the major steps. Having students write change requests and engineering change orders (ECO) are worthwhile assignments. The process of item check-in and check-out from the project data base might be illustrated by having students use a Unix tool like CVS12 or an MS Windows tool like Visual SourceSafe. It is important for students to understand the distinction between informal change control and the formal change control required for a baseline SCI.

It’s important to note that most CM functions remain unchanged for Web or Mobile Apps (Section 29.4). However, the issues noted in Section 29.4.1 are emphasized and should be discussed. Content management is introduced here.

**SCM Standards**

A wide array of military and ANSI/IEEE software configuration management standards are listed in a side bar late in this chapter. It may be worthwhile to have students do some web research to locate the details for one of them.

**Chapter 30 — Product Metrics**

This chapter discusses the use of software measurement and metrics as a means of helping to assess the quality of software engineering work products. Most of the metrics discussed in this chapter are not difficult to compute. Students can be encouraged to compute several of them for them own work products. What will be difficult for students is trying to interpret their meaning, since they will not have any historic data to use in their analyses of the metrics. Discussing case studies based on commercial products may help students to understand the use of metrics in improving the quality of software engineering work products.

**A Framework for Technical Software Metrics**

General principles for selecting product measures and metrics are discussed in this section. The generic measurement process activities parallel the scientific method taught in natural science classes (formulation, collection, analysis, interpretation, feedback). A key point to get across to students is that effective measures need to be easy for developers to collect. If the measurement process is too time consuming, no data will ever be collected during the development process. Metrics should be easy to compute or developers will not take the time to compute them. The tricky part is that in addition to being easy compute, the metrics need to be perceived as being important to predicting whether product quality can be improved or not.

The GQM paradigm (Section 30.1.4) should be used whenever someone wants to develop a software metric. To help your students better understand its intent, have them use the GQM approach to create a metric for a product other than software.

**Metrics for the Requirements Model**

This section discusses metrics that may be useful in assessing the quality of the requirements model. Students need to understand that it is important to begin assessing product quality long before the first line of code is written. Most managers want the quality effort to be rewarded in the current project, not the next one. The metrics presented in this section are presented in sufficient detail to allow students to compute them for their own projects. It may be interesting to have students compute the same metrics for their term projects before and after implementation. If similar student projects are assigned each semester, it may be worthwhile to collect historic product measures and allow future student development teams to make use of this data to interpret their own metrics.

**Metrics for the Design Model**

Representative design model metrics are discussed for architectural design, component-level design, and interface design. Students may benefit from seeing examples of each computed from real design work products. The computations are not too difficult and after seeing some examples students should be able to compute them for their own work products. Design is where measurement is a ‘must.’ Be sure to emphasize that a software engineer should use these metrics as he/she is doing design work.

If time permits, a discussion of Architectural design metrics is worthwhile. Be sure to emphasize that some of the measures noted have not been validated widely and are intended only to give a “feel” for design quality.

Section 30.3.2 describes nine measurable characteristics of an OO design. The details of the data that need to be collected and the metrics that can be computed are discussed later in the chapter. Students may benefit from seeing examples of actual measures and metrics computed for each of the nine characteristics.

A variety of different class-oriented, component level, and operation-oriented metrics are discussed in Sections 30.3.3 through 30.3.5. Enough detail is presented to allow student to compute most of these metrics for their own software projects. Students would benefit from seeing how to compute and interpret these metrics for a real OO design (which requires access to some historical project information). A potential project activity might be the development of one or more "tiny tools" to automatically (or semi-automatically) compute the metrics presented in this section.

**Metrics for Web and Mobile Apps**

The intent of this section is to emphasize that measurement is important in WebE. The design metrics presented are representative of many that have been proposed in the literature. It is important to note that many of these metrics have not as yet been validated and should be used judiciously.

**Metrics for Source Code**

The only source code metrics mentioned in this section are associated with Halstead’s software science. It may be a good idea to show students how to compute them for a real program and then have them compute these for one of their own programs. Again students may benefit from seeing examples of how to use these metrics to improve a developing software product.

**Metrics for Testing**

This section discusses the role of metrics in assessing the quality of the tests themselves. The majority of testing metrics focus on the quality of the testing process, not on the test work products. In general, testers rely on analysis, design, and coding metrics to guide the construction and execution of test cases. The point to get across to students is that good test metrics need to either predict the number of tests required at various testing levels or focus on test coverage for particular software components.

Six new metrics that may be useful in determining the testability of OO systems are described in Section 30.6.2. Students may benefit from seeing how the metrics may be used in measuring the testability of an actual system (in terms of threads, scenarios, and clusters). Ultimately, students will want to find some relationship between these metrics and the number of test cases that need to be generated to test a system.

**Metrics for Maintenance**

Students should be reminded that all the metrics discussed in this chapter can be used for both the development and maintenance of existing software. The IEEE software maturity index (SMI) is specifically designed to assist developers in assessing the stability of a product undergoing changes. An example of how to compute SMI for a real product might be helpful to students.

**Part 4 — Managing Software Projects**

**Chapter 31 — Project Management Concepts**

This chapter discusses project management at a fairly general level. The important point to get across is that all software engineers are responsible for managing some portion of the projects they work on. Modern software development is a very complex undertaking that involves many people working over long periods of time. The key to successful project management is to focus on the four P's (people, product, process, and project). Effective communication among all stakeholders is essential. The software process selected must be appropriate for the people and the product. The project must be planned if the goal is to deliver high quality software, on time and under budget.

**The Management Spectrum**

This section provides an overview of the four P's of project management. The point to emphasize is that each the P's is important and it is the synergy of all four working together that yields the successful management of software products. This also the time to remind students that it is customer and the end-user for whom the product is being developed. Process framework activities are populated with tasks, milestones, work products, and quality assurance checkpoints regardless of the project size. To avoid project failure developers need react to warning signs and focus their attention on practices that are associated with good project management.

**People**

Companies that manage their people wisely prosper in the long run. To be effective the project team must be organized in a way that maximizes each person's skills and abilities. Effective managers focus on problem solving and insist on high product quality. Software teams may be organized in many different ways. Two keys factors in selecting a team organizational model are desired level of communication among its members and difficulty level of the problems to be solved. Hierarchically organized teams can develop routine software applications without much communication among the team members. Teams having a more democratic style organization often develop novel applications more efficiently. It is important for students to understand that the larger the team, the greater the effort required to ensure effective communication and coordination of team member efforts.

If you’re teaching a graduate course or an undegraduate course populated by industry professionals, we recommend assigning reading from DeMarco and Lister’s classic text, *Peopleware* and from Weinberg’s *On Becoming a Technical Leader.* Another worthwhile book that discusses the patterns of project behavior is *Adrenaline Junkies and Template Zombies* by Tom DeMarco and his colleagues.

**The Product**

The first project management activity is the determination of software scope. This is essential to ensure the product developed is the product requested by the customer. It is sometimes helpful to remind students that unless developers and customers agree on the scope of the project there is no way to determine when it ends (or when they will get paid). Regardless of the process model followed, a problem should be decomposed along functional lines into smaller, more easily managed subproblems.

**The Process**

Once a process model is chosen, it needs to be populated with the minimum set of work tasks and work products. Avoid process overkill. It is important to remind students that framework activities are applied on every project, no matter how small. Work tasks may vary, but not the common process framework. Process decomposition can occur simultaneously with product decomposition as the project plan evolves.

If time permits, you might review some of the content of Chapters 3 to 5, emphasizing the process model (s) that are most appropriate to the work done by your students.

**The Project**

The text lists warning signs that indicate when a software project is failing. Software engineers need to be on the watch for them and take corrective action before failure occurs. Most failures can be avoided by doing things right the first time and avoiding the temptation to cut corners to try to shorten the development cycle. Skipping process steps often has the effect of lengthening the development time since the amount of work usually increases. Taking time to reflect on how things went once a project is over, is a good habit to develop in students (who should be striving to avoid repeating their past mistakes on future projects).

**The W5HH Principle**

Boehm's W5HH principle is a simple organizing tool that can help both novice and experienced software engineers focus on what is really important to include in a project management plan. Boehm's questions are applicable to all software projects, regardless of their size or complexity.

As a classroom exercise, select a well know software product and have students answer each of Boehm’s W5HH questions.

**Critical Practices**

The Airlie Council list of project integrity critical practices provides a good baseline for assessing how well a project team understands its practices. Most of the items in this list will be discussed in later chapters of the text. It may be helpful to have students begin thinking about this list in the context of developing their own project management plans. While this is difficult undertaking for students early in the course, it does get them thinking about the big picture without worrying about the details of software implementation.

**Chapter 32 — Process and Project Metrics**

This chapter provides an introduction to the use of metrics as a mechanism for improving the software development process and managing software projects. The concepts discussed in this chapter will be difficult for the students to relate to prior to working on a large software project. It is important to expose them to the reasons for using metrics and so that they can appreciate their potential in monitoring development costs and schedules on future projects.

**Metrics in the Process and Project Domains**

Measurement is not used in software engineering work as often as it is in other branches of engineering. Software engineers have trouble agreeing on what to measure and have trouble evaluating the measures that are collected. The point to get across to the students is that the only rational way to improve a process is to make strategic decisions based on metrics and indicators developed from measurements of process attributes. Students also need to understand the differences between process metrics and project metrics. Process metrics are used to make strategic decisions about how to complete the common process framework activities. Project metrics are used to monitor progress during software development and to control product quality.

**Software Measurement**

In this section the metrics that can help in the management, tracking and control of software projects are introduced. Size-oriented metrics are derived by normalizing quality or productivity measures over the product size (typically LOC or KLOC). Students need to appreciate some weaknesses of LOC as a measure (like language dependency). Some discussion about what to count in LOC (e.g. executable statements) and what not to count (e.g. comments) might be wise here.

Function points are presented as an example of a method of indirectly measuring functionality using other direct measures. Function points can be used to normalize software. Function point values (FP) are easier for students to compute (prior to implementation) than LOC for their projects (see Chapter 30 for details).

The table presented in Section 32.2.3 should be discussed during class. The points to make are: (1) the data are empirical, not absolute; (2) the data are based on a database of 6,000+ projects; (3) most important, the data provide an indication of which languages provide the most functionality per LOC.

The OO metrics presented in Section 32.2.4 provide a means for assessing the “size” of an OO project. If time permits, it would be worthwhile to present an example based on Lorenz and Kidd’s metrics.

Stress that metrics for Web engineering projects (Section 32.2.6) are in their infancy but they can provide an indication of overall project “size.”

If time permits, it’s worth having your students examine one or more of the tools suggested in the sidebar in this section.

**Metrics for Software Quality**

This discussion should be considered within the context of information presented in Chapter 19 and general quality management information presented in Chapter 21.

We believe the most important topic to emphasize here is DRE (Section 32.3.2)—a measure that provides substantial benefit at both the project and process levels.

**Integrating Metrics within the Software Engineering Process**

The fact that many software developers resist the use of measurement to guide their work will make it hard to convince students of its importance. However, the fact remains if developers do not measure they have no means of determining whether they are improving or not. Students need to understand that many current practitioners are still self-trained and may not be following the best development practices. Current thinking among experienced software developers is that process improvement is essential to remain competitive economically. This cannot happen without means of repeating past successes and avoiding inefficient development practices.

**Metrics for Small Organizations**

The important point in this section is that small projects and small organizations can also benefit economically from the intelligent use of software metrics. The key is to select metrics to compute carefully and to ensure that the data collection process is not to burdensome for the software developers.

**Establishing a Software Metrics Program**

This section discusses the steps needed to establish a goal-driven software metrics program. The important points are to choose your business goals and to determine what you expect to learn from the metrics program. The measures and derived indicators used will need to answer questions related to the attainment of these goals. It is also important to keep in mind that the modern view of software quality assurance includes customer satisfaction goals as well as product quality and process improvement goals.

**Chapter 33 — Estimation for Software Projects**

This chapter provides students with basic information on estimating for software projects. The estimating task involves estimating how much time, effort, and resources are required to build a software product. In most cases, there is enough information provided in the text to allow students to estimate their own projects and write their own planning documents. Students should be assigned the task of using the planning document template on the SEPA web site to write a planning document as part of their coursework early in the semester.

**Observations on Estimating**

The point to get across in this section is that project estimation is a difficult task to do without historical data and a fair amount of experience with the proposed application. Nonetheless, it is a task that is required for virtually all projects. Project complexity, project size, and structural uncertainties affect the reliability of the estimate.

**The Project Planning Process**

The objective of software project planning is providing a framework that allows managers to make reasonable estimates of the resources and time required to build a software product. It is important to point out to the students that the more information an estimator has, the better his or her estimates will be. This is an important reason to update all estimates, as the actual project costs and schedule become known as the project unfolds.

The task set presented in this section should be discussed during lecture. Students may be unfamiliar with many steps, if they do not have actual project experience.

**Software Scope and Feasibility**

Determining the scope of a software project is the first project planning activity. Students need to understand that until the developer and customer agree on the scope of the project it is impossible to determine what the project will cost and when the project will end. The best software practices call for the customer and developer to work together to identify the problem areas to be addressed and to negotiate different approaches to their solutions. Once the project scope is established, feasibility is the next issue to address. It is sometimes hard for young software developers to recognize that having the resources and capabilities needed to build a system, does not always justify building it. The best interests of the customer must come first, even if it means advising against the creation of a new software product.

**Resources**

For software project work the resources used involve people, reusable software components, the development environment (hardware and software). The number of people required for software projects can only be determined after an estimate of development (e.g. person months) effort is computed. Students may have a tough time relating to software reuse. Student are either anxious to build their own software or naively believe that all they need to do is browse the Internet for some code to download. A more detailed discussion of component-based software design and software reengineering appears later in the text. In modern software development, people and hardware may be shared among several projects. Time windows for resource availability must be prescribed and planned for.

We would suggest using Figure 33.1 for your talking points as you discuss this topic during lecture.

**Software Project Estimation**

Software is now the most costly element of virtually every computer-based system. Cost and effort estimates may determine whether an organization can realistically undertake the development of software product or not. Software estimating can never be an exact science, but even students can be taught the steps needed to make estimates having acceptable risks associated with them. It is important to get students used to the idea or using 2 or more methods for making an estimate and then using the results to cross check one another. Students should be encouraged to reconcile the differences between multiple estimates to improve their confidence in the values computed.

**Decomposition Techniques**

This section compares two methods of performing software sizing (directly by estimating LOC or indirectly using FP). The function point method seems to be a little easier for students to work with during the planning phase of their projects. The text suggests using the expected value (3 point) method of adjusting their software size estimates (either LOC or FP). It will be easier for students to develop meaningful LOC or FP estimates if they attempt to decompose their projects along functional lines and then estimate the size of each subfunction individually. This approach is called problem-based estimation. Process-based estimation is also discussed in this section. Students often prefer process-based estimation since they are estimating the amount of time they plan spend on the tasks that make up each phase of their process model after they have determined the work products for each phase (several low cost PC scheduling tools support this method, like MS Project). It may be wise to have the students reconcile the results obtained from a problem-based method like FP with their process-based estimates. It is important to point out that without some historical data to give these estimates a context LOC and FP values may not be very useful for estimating cost or effort.

Sections 33.6.7 and 33.6.8 discuss estimation using use-cases. It’s important to emphasize that this approach is still in its formative stages and is likely to result in somewhat less reliable estimates.

**Empirical Estimation Models**

This section describes the general process of creating and using empirical cost estimation models. It may be wise to work through an example showing how a simple linear regression model is created from raw data and used to predict the value of dependent variables from new data points. Most students have not seen linear regression prior to this course and may not appreciate how these models are built. The equations in these models still require inputs like LOC or FP but users do not need local project data to compute their estimates. Model users only need to be confident that their project is similar to those used to create the model in the first place. The complete details of COCOMO II are not given in text and will need to be found on the COCOMO II Web site. Similarly, the details of the Software Equation will need to be located on the Web.

**Estimation for Object-Oriented Projects**

You should examine the similarities and differences between the model presented for conventional and OO project estimation. Lines of code (LOC) and function point (FP) techniques are not always relevant to estimating object-oriented projects. Students should be encouraged to use the OO estimating and scheduling approach presented in this section on one of their own projects. Examining predicted versus actual time and project size for a completed project (using the text values) may be a worthwhile activity.

**Specialized Estimation Techniques**

Estimation for agile software development and Web engineering projects is presented in this section. The steps outlines for agile projects make heavy use of scenarios (a mini-use-case) as the independent estimation variable. You should note that the “modified” FP measure suggested for WebE projects must be calibrated locally before it can be used at the project level.

**The Make-Buy Decision**

The make-buy decision is an important concern these days. Many customers will not have a good feel for when an application may be bought off the shelf and when it needs to be developed. The software engineer needs to perform a cost benefit analysis in order to give the customer a realistic picture of the true costs of the proposed development options. The use of a decision tree is a reasonable way to organize this information. Outsourcing is a popular idea for many companies these days. The decision is usually motivated by the promise of reducing costs. This promise may or may not prove to be true, if the outside contractor handles the project management.

**Chapter 34 — Project Scheduling**

This chapter describes many of the issues associated with building and monitoring schedules for software projects. Students will need to be shown the process of building a schedule for a case study to really understand how it's done. They should be required to build a schedule for one of their own projects early in the semester. Ideally, all scheduling work should be performed using a scheduling tool.

**Basic Concepts**

This section is intended to motivate the student's interest in project scheduling by describing several reasons why software projects are not completed on time. There is also a description of a proactive way to deal with unrealistic customer deadlines (based on detailed estimates and use of incremental development to deliver critical functionality on time). Scheduling is no longer a seat of the pants activity. There are many excellent tools that can be used to make the process easier. The basic idea to get across to the students is to break the software project into well-defined tasks, determine the interdependencies among the tasks, determine the time duration for each task, and assign the tasks to project team members. Each task must have defined outcomes and be associated a meaningful project milestone.

**Project Scheduling**

Section 34.2.1 introduces a set of basic principles that guide software project scheduling. Each should be discussed during lecture.

The most important point to get across in Section 34.2.2 is that adding people to a project in an arbitrary manner does not reduce the project completion time (and may in fact lengthen the completion time). There are times when a project schedule has slipped so badly that adding people cannot save it and the only option a manager has is to renegotiate the completion date with the customer. The effort distribution model presented in this section is a good guideline for students to follow when they build their first project schedules.

Be certain that your students understand that the 40-20-40 rule is a rule of thumb, not a concrete “rule” that must be applied for every project.

**Defining a Task Set for the Software Project**

A "task set" is a collection of engineering tasks, milestones, and deliverables. The software process model selected for project provides much guidance in determining the task set. Task set is also dependent on the project type and degree of rigor desired. Students should be familiar with the five project types described in this section. The method used for determining degree of rigor should be demonstrated for case study.

Scheduling involves taking the software engineering task set and distributing it on the project time line. The details of how to do this will depend on whether the software process model is linear, iterative, or evolutionary. The example discussed in this section describes the major tasks for a concept development project. It may be worthwhile to show students the task sets from the adaptable process model available through the SEPA Web site.

Section 34.3.2 contains an example of refining a major scheduling task (concept scoping) into the smaller activities needed to create a detailed project schedule. Students may need to see additional examples of task refinement.

**Defining a Task Network**

Building a task graph or activity network is the key to building a feasible schedule. The task graph represents inter-task dependencies very clearly. This allows managers to determine which tasks may be done in parallel and which tasks need to be done first.

**Scheduling**

This section recommends the use of project scheduling tools for any non-trivial project. The PERT (program evaluation and review technique) and CPM (critical path method) are mentioned in the section, but no examples are given. It may be a good idea to show students a simple project activity graph and then show them how to use CPM to determine the critical path and compute minimum project completion time. Timeline (Gantt) charts are fairly easy for students to understand and are often available as output from scheduling tools like Microsoft Project. The time-boxing procedure described at the end of this section is a time management strategy that students should made aware of.

**Earned Value Analysis**

Earned value analysis is an example of a quantitative technique for monitoring project completion to date. If students are able to estimate total project completion time they should be able to compute the percentage of the total project time associated with each project task. The progress indicators discussed in this section are fairly easy for students to compute and interpret.

**Chapter 35 — Risk Management**

This chapter defines the process of risk management and explains why it is an important part of the planning process for any software project. The text contains practical advice on how to perform risk analysis and how to build risk mitigation, monitoring, and management (RMMM) plans. Students will have a hard time relating to this material without seeing lots of examples of software risks and techniques for managing them. Having students write RMMM plans or risk information (RSI) sheets for projects of their own design is an important part of their learning process.

**Reactive vs. Proactive Risk Strategies**

This section distinguishes between reactive and proactive risk strategies. It is important for students to understand that reactive strategies are usually not successful as a means of avoiding serious project problem. Select a serious system failure that made the news recently and use it to discuss reactive and proactive risk strategies.

**Software Risks**

Risks involve areas of uncertainty in the software development process that have the potential to result in nontrivial losses to the project. Most computing students will need help in recognizing that software risks go beyond technical concerns and also include the economic uncertainties that come with marketing a piece of software. Students also need to be aware that while most software risks can be identified prior to beginning a project, some cannot. The fact remains that even if it is impossible to manage all risks any planning is better than no planning.

**Risk Identification**

This section discusses the differences between identifying generic risks and product-specific risks. Generic risks can be listed on a checklist to examine for every software product. Examining the project plan and the software statement of scope identifies product-specific risks. Students may need to be shown examples of software project risk checklists. The risk assessment table shown in this section provides students with a good to begin quantifying the impact of many types of risk.

**Risk Projection**

Risk projection (estimation) attempts to associate with each risk the likelihood (probability) of its occurrence and the consequences of the resulting problems if the risk should occur. The students should go through the process of creating risk tables for projects of their own. Determining the probabilities and quantitative impact measures will be very hard for them. It may be wise to give them some heuristics for converting qualitative statements into measures. If it is easier for them to estimate costs to fix problems, then Halstead's risk exposure (RE) might be helpful to use.

**Risk Refinement**

Risk refinement is the process of decomposing risks into more detailed risks that will be easier to manage. Using the CTC (condition-transition-consequence) format may be helpful to students as they refine their own risks.

**Risk Mitigation, Monitoring and Management**

When teaching this section, it will be helpful to give students examples of several types of risks and have the class discuss detailed strategies for mitigation, monitoring, and management. Contingency plans often benefit from brain storming activities. This section also provides a chance to pitch the importance of metrics as a source of indicators that can assist managers in risk monitoring.

It is important to note that risk monitoring also includes tracing problems back to their points of origin (to do a better job of mitigating this risk in the future).

**Safety Risks and Hazards**

Engineers are often entrusted with ensuring the safety of the public and software engineers are not exempted from this obligation. There are lots of stories in the media that provide examples of critical software systems that failed with dire consequences to people or businesses.

If time permits, you might use Bill Joy’s now famous article “The Future Doesn’t Need Us” (available on the Web) as a wonderful discussion topic about risks and hazards of *Terminator* proportions. As a counterpoint, use Kurweil’s book, *The Singularity is Near.* We guarantee your students will love the discussion and will want to spend more time than you’ll have.

**The RMMM Plan**

The template RMMM plan appears on the SEPA website. Students should be encouraged to examine its headings.

**Chapter 36 — Maintenance and Reengineering**

As many types of software evolve into a Web-based entities (with all of the concomitant characteristics of WebApps) we hear a bit less talk about software maintenance. Yet, maintenance and reengineering continue to be major challenges for the software industry and continue to consume vast amounts of resources.

The intent of the chapter is to discuss software maintenance, software supportability, and reengineering. It’s important to note that virtually all of the principles, concepts, techniques, methods, and tools presented in SEPA contribute to maintainable software and lead to systems that can be more easily reengineered.

**Software Maintenance**

Lehman’s laws, although a bit academic, are worth discussing because they provide insight into how software-based systems evolve.

Some students will have difficulty appreciating the challenges associated with maintenance because the only programs they’ve modified have been their own. Spend some time discussing the reality of maintaining a 15 year old program.

**Software Supportability**

Some students will perceive maintainability and supportability as the same thing, but there are subtle and important differences. Be sure you explain them, emphasizing the need to build software and supporting infrastructure that assist maintainers and others to support the operational use of the system, not only maintain it when it “breaks.”

**Reengineering**

A discussion of software reengineering should begin with the “maintenance iceberg.” Even after almost 40 years, this metaphor rings true, and yet, many students have virtually no appreciation of the burden that maintenance places on the software community. The steps of the software reengineering process model are considered in some detail in the sections that follow.

Here, we introduce business process reengineering (BPR) and discuss the management and technical aspects of software reengineering. The basic principles of BPR are introduced and an overall model for reengineering the business is discussed briefly. In reality BPR is a topic that is beyond the scope of most courses that use SEPA. It is introduced in this chapter because it is the driving force behind most software reengineering activities.

*Critical Points:* BPR extends beyond the scope of software to the entire business. Yet the results of BPR can have a profound impact on information systems that service a business. Software maintenance is a significant burden for all software organizations. The software reengineering process encompasses inventory analysis, restructuring, reverse engineering, and forward engineering.

**Software Reengineering**

The basic activities that are performed when software is to be reengineered are discussed in Section 36.5.1 and 36.5.2. The discussion begins with software maintenance and then continues into an overview of the software reengineering process model. Each task performed as part of the model is discussed briefly.

**Reverse Engineering**

This section considers reverse engineering activities and identifies key concepts that must be understood as information is extracted from an existing program. Reverse engineering techniques for processing, data, and user interfaces are all discussed. If time permits, distribute two undocumented pieces of code, one well structured and designed, the other a kludge. Have your students attempt to reverse engineer each and then draw conclusions from their experience.

**Restructuring**

This section presents a brief overview of restructuring techniques for code and data. If time permits have students research one or more restructuring tools (see the sidebar in this section) and then discuss how they work on existing code.

**Forward Engineering**

This section presents an overview of forward engineering approaches for client/server systems, OO systems, and user interfaces.

**The Economics of Reengineering**

This section presents a simple cost-benefit model for reengineering. Not every application should be reengineered. The model presented in this section enables your students to compute the projected cost benefit of a reengineering activity.

**Chapter 37 — Software Process Improvement**

The vast majority of undergraduate students will not have the background or level of experience to appreciate many of the topics discussed in this chapter. However, we have included it in this edition of SEPA because SPI is a very important topic and one that continues to challenge event the best software companies and organizations. If you choose to cover it, be sure you present a balanced treatment, covering the benefits, but also the challenges that face any organization that ventures into SPI.

Be sure your students understand the meaning of a SPI framework. Most will be unfamiliar with the CMMI or any other framework and may view these as overkill in the extreme. Be sure to spend time discussing Figure 37.1. We think it does a reasonably good job of describing the key elements of a SPI framework.

**The SPI Process**

You may choose to delay a detailed discussion of maturity models until Section 37.3, but you should introduce the topic at this point so that students understand that they are important parts of a framework.

The CSFs discussed in Section 37.2 may ring hollow for students who have not had industry experience. If you present them in class be sure to explain why each is important in real world terms.

**The CMMI and the People CMM**

If time permits and your student population is appropriate, you may want to supplement the information presented in Section 37.3 with Web-based source materials on the CMM or CMMI. For most courses, however, you intent should not be to present a complete discussion, but rather to provide your students with a feel for what the CMMI is and why it has been developed. The same comments apply to the People CMM discussed in Section 37.4.

**Other SPI Frameworks**

It’s unlikely that you’ll have the time to present any of the frameworks presented in Section 37.5 in any detail. If you choose to do so, you’ll need to supplement the discussion in SEPA will other Web-based materials.

**SPI ROI**

Relatively little has been written of the ROI for SPI work. Section 37.6 is based on David Rico’s book on the subject and may be worth considering in some detail if SEPA is being used for an industry course with management attendance. For most university course, a brief overview of the subject is all that is warranted.

A good source of SPI information is IISP (Intl. Institute for Software Process). They have created a SPIBOK that can be acquired at **http://www.spinstitute.org/spibok.htm**

**Chapter 38 — Emerging Trends in Software Engineering**

The intent of this chapter is to provide a peek into the future of software engineering practice. Like any attempt at crystal ball gazing, my projections may be off the mark, but the general trends outlined in this chapter are likely to emerge as in the years ahead.

More important, your students should understand the underlying structure of technology trends and adoption. Section 38.1 and 38.2 discusses “technology evolution” and depicts a few “life cycles” that represent differing views on how it occurs. Spend some time discussing Figures 38.1 and 38.2. The soft trends discussed in Section 38.3 will likely have more to do with technology evolution than geeky topics. Be sure to address this in class.

By the time you use this chapter, there may be new software trends (as well as hard technology trends) that are dominating the discussion. Adapt your presentation as required.

Broy’s discussion of the “grand challenge” (Section 38.5.2) makes an excellent discussion topic.

It’s likely that you’ll only have time for a few of the topics presented in Sections 38.4 to 38.6. I would suggest choosing those topics that are most closely related to the subject you emphasized during the body of the course. If this is a graduate course (and you have time) you might consider assigning readings from Kurzweil’s *The Singularity is Near* and his more recent book, *How to Create a Mind.* We can almost guarantee your students will find the subject matter fascinating.

**Chapter 39 — Concluding Comments**

To be honest, most adopters tell us they do not cover this chapter in lecture (with the exception of the Section of Ethics). However, a few comments are still appropriate.

**The Importance of Software—Revisited**

This section revisits the importance of computer software. The key aspect of this discussion is "software as differentiator." It is interesting to have your students come up with new examples of software as a differentiator for products and services.

**People and the Way They Build Systems**

People and cultures change very slowly. In this section, We make the argument that ad evolving software engineering environment may have as much or more to do with people issues (in software engineering) than the people themselves. As tools, interaction mechanisms, and methodology mature, the culture for building software may change accordingly.

**New Modes of Representing Information**

Students spend much time thinking about data and program architectures, algorithms and the like. They spend very little time considering the intent of the data that is processed. This section considers the relationship between data, information, knowledge and wisdom.

You might relate some of this discussion to data mining in general and specific applications that span multiple databases.

**The Long View**

What will it be like 20 or 30 years into the future? Will progress be linear or exponential? Will it be “all good” or will there be unintended consequences that yield disastrous results? These are but a few of the questions that can be discussed as part of this section.

**The Software Engineer’s Responsibility**

Software engineers should abide by a code of ethics that guides the work that they do and the products that they produce. The *Software Engineering Code of Ethics and Professional Practices* is well worth discussing with your students. To make the discussion more meaningful, you should pose specific business or personal situations and have your students indicate how they would react to them.

AFTERTHOUGHT

When you write a reasonably successful book and are asked to begin a yet another edition, the first thought that crosses your mind is, "Good, now we can fix all the shortcomings of the preceding edition, respond to the many, many suggestions proposed by adopters, and re-establish SEPA as the most widely used textbook on software engineering—we'll make this book a lot better!"

Once all the research, writing, editing, and iteration are complete, we look back on what we've done and hope that the result is "a lot better." But we're never completely satisfied. There are additional topics we wish were included, expanded discussion that could provide benefit, last minute insights that occur while reading page proofs, and a multitude of additional sins.

The above comments not withstanding, we sincerely hope that *Software Engineering: A Practitioner's Approach* provides you with an effective foundation for teaching software engineering.

As always, we welcome your comments, your corrections, and your criticism, and wish you the best of luck in applying our book.

*RSP*

*BRM*

APPENDIX I

AN ANNOTATED PAPER DESCRIBING A TYPICAL SOFTWARE ENGINEERING COURSE



In 1981, Elaine Kant (*ACM Software Engineering Notes,* August, 1981, vol. 6, no. 4, pp. 52-76) of Carnegie Mellon University published a detailed discussion of software engineering course design. Professor Kant gave me permission to include her paper in the *Instructor's Manual* for the first edition of SEPA and it has included it in every subsequent edition. Kant's paper is pragmatic and based on actual experience. Although it is now almost 30 years old, it remains one of the best discussions of software engineering course design that we have encountered. Undoubtedly, some of the concepts Kant notes must be updated for current technology and modern students, but the overall recommendations are as valid today as they were when the paper was originally written.

The Kant paper[[6]](#footnote-6) is reproduced with permission and is presented in the following manner:

1. Reduced type font (i.e., reduced font) is used for all text reproduced directly from the Kant paper.

2. **Bold sans serif type** font is used to include my comments and supplementary notes.

You should note that parts of the Kant paper are direct extracts of student handouts and are therefore addressed directly to the student. Notes to the instructor (which do not appear in the actual handouts) are denoted with **boldface** font.

**4.1 A SEMESTER COURSE IN SOFTWARE ENGINEERING**

**Elaine Kant, Carnegie-Mellon University**

**1. Introduction**

Software engineering is a difficult subject to teach. The field is not a rigorous one, and there are few good textbooks [RSP note: hopefully, SEPA remedies this problem] or model courses to follow. To add to the confusion, the course is taught at many different levels to students with widely varying backgrounds. This paper discusses a project-oriented software engineering course aimed primarily at sophomores and juniors with at least two previous computer courses. Readings, suggested projects, and assignment descriptions are included.… The rest of this paper is a modification (what it should have been, rather than exactly what it was) of the handout given to the students on the most recent version of the class. It introduces the schedule of lectures and assignments for the project documentation. Comments for instructors are noted in **boldface** type.

**RSP: Although Professor Kant indicates that the course has been designed for and taught to an undergraduate audience, there appears to be no reason why the same presentation (with added work and a bit more rigor) cannot be offered in a graduate curriculum.**

**2. Course Organization**

This handout describes the course Software Engineering Methods. It explains the organization of the course, outlines the lecture topics and project assignment due dates, and gives some hints about working in groups and writing up the project documents. Finally, some suggested projects are given, and all of the assignments related to the project are described in detail.

**2.1 Course Overview**

Software engineering is concerned with long-term, large scale programming projects. This course introduces the topic through lectures and by giving you a chance to help design, manage, and implement a medium-sized project. The lectures and the group project will cover topics in software engineering management, problem specification and analysis, system design techniques, documentation, system testing and performance evaluation, system maintenance, reliable software, current programming and run-time environments, and possibilities for the future. The course will be difficult and time consuming, so you must have satisfied the prerequisites listed below.

**2.2 Prerequisites**

**The prerequisite courses at CMU use covered basic data structures (including a number of implementations for lists and sets), recursion, data abstraction, formal specification and verification, and finite state machine models. Prerequisites should probably also include a data structures course and an algorithms course and previous experience in writing small packages. I strongly recommend giving a pretest during the first week in class to make sure students have sufficient programming abilities. I gave out a questionnaire on programming background to help us spot potential strengths and weaknesses, but I had to take some of the answers with a grain of salt. For example, some students claimed to have written 106 lines of code!**

**Homework**

Most homework will be related to the course project described below. There will also be reading assignments and occasional other exercises. There is an assignment writeup for each step of the project that includes a reading assignment. The readings should be done before you do the assignment. You are encouraged to read ahead as much as possible. Another course requirement is a weekly log of how much time you have spent on different activities related to the course. A sample log sheet is attached. You should turn in your log sheet each week at your section meeting.

**The students generally do not like keeping logs. Although the logs gave me valuable feedback the first time I taught the course and although I think the students who kept them carefully learned from the experience, I'm not sure I believe they are critical. Perhaps they could be optional for extra credit.**

**Group Project**

The class will be divided into groups of people. Students are strongly encouraged to form their own groups of compatible people. Projects normally will be selected from the list of suggested topics and be written in Pascal, but other projects can be negotiated. Each group will have regular weekly meetings with a "consultant" ( a graduate student or member of the research staff) from the Computer Science Department. The projects will take most of the semester with major write-ups due at approximately two week intervals. There will also be some oral presentations, including a final demonstration of the project. Sample documents from the last course are on reserve in the library.

**Five-person groups were chosen partly because of the large size of the class and partly to force students to face issues of project management. Smaller groups would probably cause fewer problems but would teach fewer lessons about programming in groups. The last time I taught the class I scheduled two hours of lectures and made the section meeting the third class hour. It is important to impress on the students that these meetings are required and that they are if anything more important than the large lectures. Of course the success of this approach depends on having good teaching assistants, which I have been very fortunate to have.**

**RSP: Although allowing students to make-up their own teams ("groups") has definite advantages from the compatibility standpoint, it might be worthwhile to define teams based on the overall performance of individual students. If you take no part in defining the make-up of the teams, it is likely that you'll get one team of superstars and other teams with less capable performers. Distributing the superstars among various teams benefits everyone.**

**Tests and Individual Projects**

There will be an in-class mid-term exam. Rather than a final, individual programming projects will be required during the last several weeks of class. The project will involve a modification or extension to a group project on a topic in software engineering.

**I would suggest giving a final or second in-class test to make students read the second half of the textbook. There usually isn't much time for individual projects; if you want students to take them seriously you must make it clear to them that they are important.**

**Grades**

About 50% of your grade will be based on the group project documents and demonstration. You are graded on the quality of the work you produce, not on how many hours a week you spend. Use your energy wisely! The rest of the grade will depend on individual tests, projects, and contributions to the group projects. It is very important that project assignments be turned in promptly, both to allow you time to complete the entire project on schedule and to allow us time to evaluate your work and make suggestions.

**I strongly recommend insisting that the new students stick to a strict schedule. Groups that get more than a few days behind usually stay behind and never catch up. If a group is just lazy, insist that they turn things in on time and lower their grade if they don't; if their project is harder than it appeared at first, give them an extra week to figure out how to simplify it, but then insist that they stay on schedule. Also, make it very clear from the beginning that grades are not assigned in proportion to the number of hours spent or you will get a lot of complaints. Assigning individual grades in a projects course is difficult. I suggest stating exactly how much documentation and how many lines of debugged code, not counting comments and write-in statements, you expect from each person for an A, and what the grade penalties are for lateness and poor or incomplete documents. A reasonable breakdown for the grade might be 35% on whatever the group project as a whole deserves, 35% on the student's contributions to the project, and 30% on tests and individual projects. It is important to identify people who try to do their share as early as possible and convince them to shape up or drop the course, but this is difficult in practice.**

**RSP: I have found that the following method works well on team projects: A team leader is elected by the team within one week of the start of the project. (Kant discusses the leadership question later and I will make further comments at that time.) At monthly intervals, each member of the team submits a confidential rating sheet for all other members to the Professor. The grades submitted by the team leader receive extra weight. Poor individual performance is relatively easy to spot and a conference with offenders can be arranged if serious deficiencies are being reported.**

**Comments from previous students . . .**

*Warn people ahead of time what to expect... a lot of work is involved and anyone that doesn't have the time or perseverance should drop the course.*

*This has been the most worthwhile course I have taken since I learned to write in first grade.....I've learned a lot about computing, time management, people management, and my ability to stay logged in for days at a time. I even liked the course. There is only one major problem. My QPA is dropping faster than a wingless airplane because I've spent my semester working on 311.*

*CHORUS: This course is too much work!!!!*

**The course really was too much work for the number of units given.**

**Textbooks**

**RSP: For undergraduate courses, the only required textbook for my course was SEPA. However, like Professor Kant, I put a good selection of related texts on reserve at the library. It would also be worthwhile for you to review the References and Further Readings and Information Sources sections at the end of each SEPA chapter. Of course, the Web provides a vast library of sources. Kant's required and recommended texts are dated and have not been reproduced. As an alternative to supplementary textbooks (which students tend not to use), emphasize the Software Engineering Resources section of the SEPA Web site and encourage students to visit the Web sites noted.**

**Keeping a Log**

Each person is to maintain a log of the time spent on course work and to break down the total time into categories. For example, categories might include time spent reading textbooks or articles, reading project problem descriptions, devising specifications, testing your program, etc.

**Some of the numbers that students give seem a bit inflated, but in general they do spend an impressive amount of time on the course. Unfortunately a lot of time is wasted when the machine is heavily loaded.**

We hope these logs will help you see how you spend your time and help you to make better predictions of the time needed by the different phases of a software project. Many companies in "the real world" actually require this sort of time-keeping.

**RSP: The idea of a log is a good one. To help the students keep the log and to establish a consistent format for data collection, it might not be a bad idea to establish a Web site in which each student can build her or his own log.**

When in doubt, use as many categories as occur to you. You may want to compute the time breakdown in several ways for later summaries; for example, total computer time, or total time spent on the testing phase. You may want to make a special category for the time you spend in the computer center when you aren't really working on your project. When you make computer runs, record the reason for the run, the changes since the last run, and the result of the run. I think you will find this interesting to look back on and see how your time was spent and how it compares with your predicted times.

Comments from Previous Students

*KEEP THE LOG SHEET. It may be a pain, but I wish I would have kept mine. It would be interesting to look back on it now and see how much time I really spent on the project in front of a terminal and how much I spent away from it.*

**3. Lecture Schedule**

**RSP: The schedule proposed by Kant assumes two lecture periods and a project session. When I was teaching, we met once a week for a 2.5 hour lecture and allowed students to schedule their own outside meetings. Frankly, I like the CMU approach better. Unfortunately, not all institutions have the resources (grad assistants, support faculty, etc.) to properly pull it off.**

**I believe that the schedule of topics proposed by Professor Kant could be improved somewhat. Alternative schedules are proposed in section 4.2 of this *Instructor's Manual* segment.**

Week 1

Lecture: course overview, background questionnaire, possible pretest

Lecture: working in groups, management structures

Section: no meeting; students should be picking groups and projects

Week 2

Lecture: software requirements analysis, functional specifications

Lecture: Paradigms, common system organizations

Section: no meeting, but project/group proposals due

Week 3

Lecture: user interfaces, error/exception handling

Lecture: technical writing

Section: draft of functional specification due

Week 4

Lecture: abstraction in design

Lecture: modularity in design

Section: functional specification due

Week 5

Lecture: more on modules and abstraction

Lecture: more on modules and abstraction; families of systems

Section: top level code, overall design draft due

Week 6

Lecture: using libraries, journals, and references on algorithms

Lecture: checking interfaces, separate compilation

Section: overall design specification due

Week 7

Lecture: reliability

Lecture: midterm exam

Section: interface definitions and stubs due

Week 8

Lecture: testing -- integration order, estimation, and scheduling

Lecture: testing -- individual module testing, test drivers

Section: detailed design specifications due, compiled module drivers due

Week 9

Lecture: discuss midterm, group interactions

Lecture: programming tools and environments for Pascal

Section: module top level code due

Week 10

Lecture: programming style, hints, and tricks

Lecture: more programming hints

Section: draft of test plan due

Week 11

Lecture: performance evaluation

Lecture: advances in programming environments (good debuggers, version control, syntactic editors

Section: test plan due, trade undebugged code

Week 12

Lecture: automated tools, program synthesis

Lecture: structure of organizations in the real world, types of jobs

Section: user manual due

Week 13

Lecture: computers and law

Lecture: computers and law

Section: practice demo

Week 14

Lecture: personal computers, networking

Lecture: project demonstrations

Week 15

Project demonstrations Final evaluation and individual projects due

**4. Working in Groups**

**The students seem to feel that the groups won't be a problem in most cases, or don't want to face the issue early enough even if they have suspicions that there are problems in their group. I strongly recommend being very aware of group dynamics in the first few weeks of class. I will let the student experiences speak for themselves for the most part.**

Most students prefer a democratic group organization. In a minority of groups, those in which all students have a reasonably equal commitment to the course, this can in fact work, as the following comments indicate:

*. . . As a group, we did manage to work well without any strict leadership structure, partly because everyone was kept pretty well informed as to the work that was due. Also, everyone participated in the writing of all papers, no one person was dumped on. We each were given a section that was later critiqued and made a part of the total paper. I think that most groups would probably need some sort of structured management, but because of our group members' personalities, we worked well without any ....*

*. . . Our group worked together very well with no formal division of management. As in any society, there are leaders and there are followers. It seems that we had a good mix between leaders and followers.*

**However, most groups find they need some sort of leader; either because they all try to do everything and waste a lot of time arguing over details or because some people don't know what they should do unless someone gives them fairly explicit assignments. Some groups may want to have a design leader who is responsible for final technical decisions and a manager who is responsible for making sure everyone comes to meeting and does their share of the work. You may wish to have a permanent leader or to rotate leaders at different stages of the project (such as letting the editor for a document be the manager for that part of the project) Additional student comments follow:**

*. . . Unfortunately none of us had worked on a software team before. We were foolish and thought we could use the 5 - pointed star managerless technique where everyone talked to everyone and each did his equal share of each assignment. We realized pretty late in the game that this would not work ... any project group must have a leader or manager. The sooner the group realizes this the better off they'll be ... As few people as possible should write a particular document. One person (definitely not the author) should edit the entire document.*

*. . . The design process should, in my opinion, have a single person in charge of both the design and the paper. He should have final say in major disputes that seem to be unresolvable by any other course ... As many of the design decisions should be delegated as can be done reasonably... he must take care to limit the discussion of relatively minor points in order to keep the meetings from getting extremely long. Finally, let me say something about communication. In a word, you can't have enough.*

While it is to be expected that some members of your group will be better writers, some better programmers, and so on, you are not to divide the labor on these grounds alone. This class is to be a learning experience, and each of you should get a substantial amount of practice in all of these areas, not just the ones you are already good at. Each person must write some of the documents, must edit one (see the next section), and must write code. You are not to turn one person into a full time typist and help - module writer just because you don't think their code will be perfect.

If one group member is a fantastic writer, by all means let him edit the first document and help teach the others how to write better. He might be assistant editor for some of the other documents, but he is not to write them all. If another member is a "wizard", go ahead and have her be design manager, but she is not to develop the software for the whole system, rather she should help the others improve their design and programming skills.

**One problem that the students who were nominal leaders pointed out was that they had no power to back up the responsibilities that went with their authority. I don't have any good answers to this, except possibly to point out that the professor might believe this person's opinions about how much the other group members are contributing to the project. Another alternative is to get the TA to be a manager. To some extent this happened informally anyway, but I suspect it would require much more of a time commitment for them to take a formal leadership role.**

**RSP: The team approach to course projects is, as Professor Kant indicates, a difficult one to resolve. The following scheme tends to eliminate some, but not all, of the problems associated with "no-work team members" and a need for leadership authority:**

**1. A team leader is elected by the team.**

**2. Because the leader has more responsibility, she/he should receive rewards for good technical management and "take some of the flack" for a poor showing.**

**3. Given the philosophy indicated in (2), the team leader will receive 1.10 of the overall team grade, if that grade is above 75% and no reward if the grade is below 75%.**

**4. At monthly intervals, each member of the team submits a confidential rating sheet for other members. Comments of the team leader are weighted heavily in a final assessment.**

**5. At the end of the project, a "bonus pool of points" (usually some percentage of the final grade) is distributed to team members based on the cumulative ratings.**

**This scheme tends to keep everyone actively involved in the project and provides the team leader with a "club" for those members who are not contributing.**

**5. General Comments on Documentation**

This course involves a lot of writing. You are not being graded directly on your writing style, but good writing usually conveys ideas more clearly than poor writing, and the process of writing down your thoughts often clarifies them. Thus it is to your advantage to make the effort to organize your thoughts and write well, and we will often make comments about your style as well as about the project you are describing.

At the beginning of the semester you should skim all of the assignments, then reread each several times before you begin to work on it. If you still have questions about the assignment after reading it carefully, be sure to ask your TA or the instructor.

You are to turn in two copies of each final document. One will be read by your TA and the instructor and will be returned to you with comments and a grade; the other will be kept in our files for future reference. Drafts of documents will only be read by your TA who will give comments but not a grade; you only need to turn in one copy for this.

Each document is to have one (not five!) editors, and each group member must edit one of the documents. Let us be very clear about what it means to edit a document. The editor has the final responsibility for turning in the document, sets deadlines for what the other members are to write, and may have to apply pressure to the other group members to do their part unless you have another manager for this.

To make things easier for you, we suggest you decide on a standard format (especially for the design documents). You may use the outline in the class directory as a model. Most of the assignments suggest an organization for the document. You do not need to follow it exactly if you have a better idea, but you should be sure to cover all the same material. Be sure to include acknowledgements at the end that say who wrote which sections and who the editor was.

**I made up some grade sheets as guidelines for what to look for in evaluating the documents, but mostly they just reiterated the points made in the assignments, and I don't think it is worth reproducing them here.**

**RSP: The above material presents a good set of guidelines for students. I suggest that you require your students to follow predefined document outlines (See Document/Work Product Outlines at the SEPA Web site). First, this drives home the need for standardization in software engineering and second, it makes the task of reviewing and grading the documents much simpler.**

**You should save the best work from year to year so that over time, a sample "software configuration" can be provided as a guide for new students. I have found that outlines, in and of themselves, provide the students with inadequate guidance. Sample documents do a much better job. The Case Study presented at the SEPA website can also be used to help students by example.**

**6. Project Suggestions**

**Due Dates**

Week 2: Turn in a description of the project you have chosen, one page minimum, five page maximum. Include a name for your group, a list of the names of the members, and a list of possible meeting times for your group. Include evening times; some TAs prefer them.

**General Comments**

Each group should select a project from the list given below; other project ideas may be used if approved by the teaching staff. A project should be feasible given the time constraints, but large enough that a cooperative team effort is required to complete it. Read the functional specification assignment and some of the others to get an idea of what will be expected of you. Note that you are asked to propose four increasingly difficult versions of your system. The first should be something you are sure you can do, and the last something you think you probably won't have time to finish. You should try to select a project that spans that range for your group.

**After a group proposes a project, a teaching assistant should be assigned, and the TA and the instructor should give the group a critique of the proposal. Is it too ambitious or not enough? Are the goals clear? Can it be expanded or reduced later if necessary? Will it be possible to split it into smaller modules? The instructor should decide on the dates for each group's final demonstration fairly early in the semester so the group has a definite date to work towards.**

**Suggested Projects [an abbreviated list is presented here]**

A registrar's program to manage registration and scheduling, including printing student registration status, class lists for instructors, room use sheets, and process add-drop cards.

A program to teach arithmetic to a little brother or sister (i.e., a program for computer-aided instruction). This could include a scoring system, for example, and perhaps some analysis of errors to determine what kinds of problems to concentrate on. This is not a well-defined project and the group must be careful to decide what is to be handled.

A system to perform differentiation and simplification on symbolic expressions. Warning - simplification is quite tricky. I recommend encouraging groups to use existing algorithms first and then augment their programs. One group attempted this project and did a reasonable job with a pretty printing expression display and with function definitions but got bogged down on simplification. They had a hard time believing that simplification is in the eye of the beholder and the next step of the problem solving.

A database system, perhaps to maintain SCRIBE bibliography databases. This would make it convenient to add new entries to the database, along with key words, and also allow searching of the database, either by key word or by other fields, or with a special query language. Keep the on-line database alphabetized.

A database system for maintaining voting records of Federal and/or State Senators and Representatives. The system should keep all districts and their current representative along with their party, data of election, and a set of user-determined votes. There should be a way to score votes; for example, to determine how a person rates on a set of issues about the environment or the ERA or the space program. In the congressional record, votes recorded include yes, no, did not vote, paired for/against, unknown, etc. The user should be able to specify how these correspond to 1 or 0 or whatever for computing scores. There should be various ways to print out this information (again, perhaps with a query language), deal with people winning and losing elections, resigning, dying, etc.

A prototype simulation of a small, low-cost computing engine that one might find for sale in stores to the average consumer in the foreseeable future. It could be for children or adults; it might perform a common routine task now performed manually, or it might be a game. The Texas Instruments "Speak-and-Spell" ~(a registered trademark of Texas Instruments~) game would be one example.

A modest version of a game of Adventure or Dungeons and Dragons. The computer would serve as the dungeon-master. Alternatively, a game where the computer is a competitor or the board manager for a group of people. Depending on the game, an attractive display may be an important part of the project. Parker Brothers' "Clue" is an interesting game: you have to keep track of what other players know as well as what you know. Other possibilities are mastermind, checkers, bridge, backgammon, chess, and Go. Keep in mind that writing the programs to handle these games may not be nearly as fun or easy as playing them; certain of last year's students have threatened murder if anyone mentions the Rivets game to them again. Mastermind is perhaps too simple a project, checkers is a bit more complex but still pretty straightforward if existing approaches are used, and adventure games can be made reasonably difficult.

An on-line recipe file and meal planner. This could compute the overall nutrition value of a meal, summarize the necessary ingredients, and estimate cost. It might also be able to suggest a meal given "I want to use up this ground beef" or "I'm out of eggs and lettuce."

A teacher's program to detect similarities in programs (i.e., a "cheat detector"), for programs of about the size and complexity required for week long homework assignments.

A program version comparator and updater. This is like the previous suggestion, but the emphasis is on determining differences rather than similarities. One might use this system after a day's editing, to get a summary of all changes. Suppose that two people edit the same program, adding different fancy features. This system might determine the differences between them and automatically or (more likely) interactively merge them to produce a new program with the features of both.

An automated bank teller. This would keep track of accounts, and let users perform various transactions: shuffle money between accounts, make deposits and withdrawals, pay bills (manually or automatically), include a display similar to the ones at the real money machines, etc. The system should be resistant to abuse and should be reliable (able to keep everyone's money straight in spite of system crashes). This is a simple project unless it includes features like multiple tellers and simultaneous deposits and withdrawals, bank manager programs, and good backup systems.

A string handling package. This would provide dynamic allocation of variable-length character strings, with either explicit or automatic de-allocation. In addition to managing storage, the package should include facilities for operating on strings (copying, concatenating, comparing, sorting, searching, matching, etc.). This project is straightforward but can involve a lot of work, especially if fancy pattern matching is attempted. One group did a nice job on a moderately sized string package.

A complex string-matching package, expanded into a spelling and diction checker. (The string matching might be based on regular expressions a la Aho). It could check a document for commonly misspelled words, as well as instances of such redundant fragments as "-ing behavior", "added increment", and so on.

A personal finances manager. Something to balance the checking account, keep track of bills, match receipts to credit card statements, do taxes, provide summary reports in various categories, and so on. One group did a modest version of this.

A calendar maintainer. This would schedule your time, send reminders when necessary (through the computer mail system), perhaps coordinate with other people's calendars for scheduling meetings, and so on. Printing parts of the calendar on demand is a must ("What's up for today?" or "Let's see next week at a glance." or "What in the world was I doing last Tuesday?" or "On what day of the week for the next four weeks is there an hour free starting at 10:00?").

A verification condition generator.

A tree (or graph) structured bulletin-board system, with the root allowed in any user's directory. Any user could post notices for others to read. Notices (especially replies) could be associated with other notices (hence the graph structure). A user might want to ask such questions as "which notices of mine have had answers attached that I haven't read yet?" Some branches of such a bulletin board might serve as a structured on-line documentation system. This would have to have a nice user interface; it should also lead the beginner by the hand (possibly by guiding them through a documentation tree!) A stripped-down version of the user interface for hard-copy terminals should be a part of the core system. One group tried a similar project but was overly ambitious and had trouble completing the project.

Write a program for analyzing the NFL playoff situation, which has a complex set of rules for resolving various situations that can arise. One portion of the project would be an evaluator to determine the playoff choices given the end-of-season scenario. The rules should be modularly implemented so that they can be changed (as they frequently are). The user interface could vary from something simple to a system capable of answering questions like "If Pittsburgh loses to Houston, what must occur for Pittsburgh to enter the playoffs?" (expressed in some more formal notation). Incremental improvements might include something like the capability to guess, based on the season record so far, how some upcoming games might be decided or what games might be crucial.

Other possibilities are limited only by your imagination. While projects such as display and window packages, command interpreters, and general menu handling systems would be quite feasible, some projects such as a poem or music composer or simple language translator or simple theorem prover boarder on research topics and you will have to be very careful not to waste all your time fighting about what you are going to do or which of the many possible approaches you will take. We want you to have a chance to complete a project, not just start one.

**RSP: Some of the projects noted above are somewhat frivolous, but all will serve to demonstrate software engineering methodology.**

**It is often worthwhile to poll various departments at your institution for small, but "real," projects that students might attempt. If you do this, your students will get to communicate with real customers and the results of their work may have lasting benefit. Be careful here, however, there is a tendency to bite off more than your students can chew!**

**Another project approach for a software engineering course is to have students develop a "Tiny Tool" for inclusion at the SEPA website. A wish list for Tiny Tools and instructions on how to submit such tools for evaluation is provided at the SEPA website.**

**The difficulty of each of the above projects can be tuned to the sophistication of the students and the number of people on a team. For many of these, the students will have to "learn ahead" for topics that are not discussed until late in the course.**

**First Reading Assignment**

SEPA assignment: See alternative course outline presented in section 4.2 of the *Instructor's Manual.*

**7. Functional Specification Assignment**

Due Dates

Week 3 draft of the paper described here; week 4 final version of paper, hand in two copies, one returned with comments, one for our files.

**General comments on the assignment**

**It is very important for the students to make definite commitments about what their project will involve. Insist on a draft and make them rewrite the final version if necessary. Points to stress are: coming up with realistic versions of the system for the "plain and fancy" section; writing samples of the system behavior, and making sure the functional specification is a very good first step towards the user manual.**

The main purpose of the Functional Specification is to explain what you are going to do for your project as opposed to how you will accomplish these goals. The functional specification, coupled with a management plan (which we will not write up formally) and an overall design (which will be your next assignment), describe the proposal to your client. When accepted, these documents are a contract between you and your client (the teaching staff) about what you intend to deliver. Changes can be negotiated later and should be recorded as an appendix to this document. When writing this paper, therefore, you should keep in mind that your audience will be a client (who, in general, is not an expert in computer science) and your group members, who will want to refer back to this document periodically to be reminded of what they are supposed to be doing. The fact is that in this course your clients are computer scientists and that your document will be graded might encourage you to think and write as carefully as possible.

**RSP: It is important to have each team develop an abbreviated Software Plan for the project they undertake. The plan should emphasize scheduling, in fact, you should insist that each team develop a week by week schedule and encourage them to schedule on a daily basis for 7 to 10 days in advance. It is critically important to have the students assign responsibilities within the team. If possible, it would be nice to provide the students with a scheduling tool (e.g., MS Project) that will make this effort a bit easier.**

**Cost estimation is not nearly as important in this context, but you can provide data for LOC costing and suggest the use of an empirical model such as COCOMO to help in estimating.**

Discuss what you want your system to look like and produce as many examples as possible of its behavior. In your group meetings you can take turns playing different roles. One person could be the system user, one could play the system, one could be system implementer and make sure the "system" is feasible, one could be the group facilitator.

**Contents of a Functional Specification**

*Title Page*

This assignment, as well as all others throughout the year, should be turned in with a title cover or cover page that includes the following information:

- A title that identifies the document

- The name of your group (such as "Great Hacks, Inc.")

- The people who contributed to this document

- Course name, instructor's name, TA's name, date

*Functional Summary*

In about 2 pages, you should briefly summarize the project you are working on. Give your system a name. Describe what it does and who will use it. What needs will your system satisfy? How will it help the users? Outline the most important features of your system. Describe the physical environment in which your system will be used, including any other systems with which the new system will interface. Are there any important performance goals for your system: time or space efficiency, security, or reliability?

*Details concerning system user interactions*

The main point of this section (which should be approximately ten pages) is to describe the functions that the system will perform from the point of view of the system user. You need to cover the kinds of inputs your system expects, the actions it will take on both expected and unexpected inputs, and the types of outputs that the user will see in those cases. Part of this section will eventually be developed into a user manual. First we'll consider the content, then the form, of these descriptions.

The inputs, state changes, and outputs should be described in reasonable detail. You need not stick to the exact wording of messages, but you should make definite statements. You can negotiate major changes later. Describe what legal values or ranges your system will accept for inputs, what precision or accuracy constraints you will follow, and how you will handle errors.

Your functional description should employ some of the following techniques:

Sample transcript . A very important part of the description is a sample interaction with the system in the form of a transcript of a dialogue between a user and the system. Use upper and lower case or some similar convention to distinguish between what the program types and what the user types.

**Students resist doing this, but I would insist that they produce a number of examples.**

**RSP: I had each team develop a "paper prototype" of the system that they are to build. This is particularly useful for graphics applications but has benefits for all types of projects. One way to overcome students' resistance is to convince them that the prototype will be a useful artifact if disaster strikes and they can't get their application "up and running**

FSM model . If your system can be thought of as being in a number of states, you can draw a finite state machine model (FSM). The different states might correspond to different prompts for user response that are shown in your dialogue.

BNF specification . If much of your system involves commands specified by the user, write a BNF describing the syntax of these commands.

Boxes, arrows, and tables. As appropriate, use pictures of equipment, data flow or control flow diagrams, decision tables, etc. Be sure that any diagrams are captioned and that there is a key explaining what various boxes and arrows mean.

Glossaries. Be sure that all specialized terms are defined in the body of the document or in a collection at the end. This could include computer science terms that you wouldn't expect your client to know or application terms (from accounting, geology, or whatever) that programmers working on the project might not understand.

*Feasibility*

Make sure your project has a chance of being completed in a semester. What are your preliminary thoughts on how you will break down the different features of the project? What are the major classes of functions and the relationships between them? Think a little about your main data structures and algorithms, and spend a page or so discussing possible implementations. Don't waste a lot of time fighting over details, but be sure to ask your TA or the instructor if you aren't sure how hard something will be.

*Our system, plain and fancy*

Predicting how much can really be accomplished in a finite period of time is often tricky. It is important to decide what a bare bones version of your dream would involve. Describe, in one or two pages, the composite of your skeletal system. What functions will not be included, and why? Cover yourself by making sure that reading between the lines doesn't make your customer think that more is being promised than you plan to deliver. Now sketch out which features will be added as time permits. Try to organize the bells and whistles into packages that could be added independently or in some predictable order. Also think about the order in which parts of the system can be dropped so that you can retreat gracefully if necessary. (You may wish to look ahead at the Parnas readings listed in the design document handout to find out more about subsetting.)

A good way to describe your series of systems is in the form of user reference cards. You should make a series of three or four cards outlining what is contained in (or added by):

- a kernel system with absolutely minimal features (that you are positive you can finish in 3/4 of a semester)

- a cheap but usable system (that you expect to finish)

- a standard system (that you have a good chance finishing)

- a super system (that you probably won't be able to finish but would like to; don't make this something impossible though)

*Summary*

Don't drop the reader off a cliff at the end of your paper. It's been a long time since the beginning of the paper. Restate the main points you want remembered.

*Acknowledgements*

Write down the authors of individual sections, the editor of the whole paper, outside consultants, etc. (Do this for all future papers as well.) This will help people figure out who to ask for more details.

**RSP: I try to emphasize to the students that all documentation should stand on its own—that is, they should not assume that "We'll be around to answer any questions, so we can be sloppy about ambiguity, etc."**

*Optional Sections*

A real functional specification would include a number of items that you may wish to skip at this point. Some of them require more experience to predict and may be covered later, some are not relevant to all projects, and some have already been discussed. Use your good judgment.

- Make some general statements about the performance goals for your system. What are your goals for system run time, main and disk memory use? What kind of reliability will you guarantee? Security of information? What kind of performance trade-offs have you decided to make?

- Can you say anything about compatibility with existing software or hardware? What about an installation agreement? Maintenance contract?

- What resources are to be committed to the project? Who are the people on the project? Their skills and background? What is the promised delivery date? How much computer time and space will be used to develop the project?

- What publications will be produced? Who are the intended audiences?

**Readings**

**SEPA assignment: See alternative course outline presented in section 4.2 of the *Instructor's Manual.***

**Some comments by previous students**

*. . . Writing down the functional specifications is fundamental to having a good start on a well-designed project. No matter how much you discuss something there will be loose ends and not until you write your ideas down will these unresolved parts become visible. The most important sections of the paper are those dealing with the functions of the system. You need to understand thoroughly "the transformations from the user point of view." You should not have vague points unless you absolutely cannot help it.*

*. . . In some respects the Functional Specification was the most useful of the documents. It forced the group to adopt a rigid management structure (we quickly found that five people writing the document causes tremendous amounts of bickering) and outlined the goals of the project. It also provided a first look at the more serious problem which the project would have to overcome.*

**8. Design Documents**

**Due dates**

Week 5. bring in top-level code and design draft; divide responsibilities for overall design sections

Week 6. overall design specifications due (2 copies) (preliminary grade) reapportion responsibilities, group review

Week 7. interface definitions and stubs due (preliminary grade)

Week 8. detailed design specifications due (2 copies) major grade) module drivers for kernel system due (compiled but not tested) reapportion and review remaining tasks in section

**The design process**

The design of large systems is still a bit of an art. To put a little order into the chaos, we've established a few milestones for you. During the design process you should focus on the parts of the design essential to producing a core system and merely outline extensions to a more complete system. One of your major goals is to get something working by your deadline, even if it isn't the full system. Thus, you should determine what needs to go into a skeletal system and schedule the design of those components first.

The end goal in the design process is a detailed design document that is complete enough to be a reference from which any competent [software engineer] computer scientist can produce code, test plans, or a user manual. (This document should eventually evolve into the code and system maintainers' guide.)

An intermediate goal is the overall design specification. This can be considered a draft of the detailed design document with some of the details missing. Interfaces between modules should be clearly defined.

Once the design has been written down, everyone in the group should read the entire document to make sure they understand the design. Individuals should be assigned responsibility for particular modules and should design these in more detail.

Each person should carefully review at least one other person's section (and all sections should be reviewed by someone other than the author). Pick the one(s) with which your module has the strongest interaction(s). Based on the examination on the module interactions, you should prepare a set of interface definitions.

The final step is then to prepare the detailed design document. Most of the work will be done by individuals filling in the details of their modules, but you must review the document as a whole to make sure people are using the same interface definitions and to make sure that everything has been covered and that the parts of the document hang together.

**RSP: SEPA introduces a review technique that can be applied by the students during design. You should encourage the use of walkthroughs. Be sure that students apply a formal (data flow, data structure or object-oriented) design technique. A preliminary design "review"—a 15 minute in class design presentation by each team—is recommended during the early weeks of design.**

**Writing the documents**

First of all, remember Brook's comment (The Mythical Man-Month, page 165

*Most documentation fails in giving too little overview. The trees are described, the bark and leaves are commented, but there is no map of the forest. To write a useful prose description, stand way back and come in slowly.*

Your goal should be to provide a recursive (top down) description of your program in a form something like the following. You probably won't be able to design your project completely top down, but such a description should be a good explanatory and reference tool. For each level, you should cover the following:

*Abstract*—What services does this program, module, routine, etc. provide to outsiders?

*Implementation Documentation*—Are any special instructions need for the system user; for the writer of other modules?

*Design*—What is the basic design of this module or routine? How are the design decisions reflected in the submodules or routines which make up the whole? How are the pieces combined to accomplish the main function described in the abstract?

*Exports*—What routines does this (sub) module make available to other modules? What type declarations? What constants? (Remember, types can be exported, but not parameters.)

*Imports*—What procedures defined in other modules are used? (Beware: a procedure, function, or type cannot be both an import and an export of the same module!).

*Input/output*—Make sure all necessary actions are specified.

*Subparts*—What are the names of the submodules, routines, and/or data abstractions that make up the module or submodule?

*Pre and Post conditions*—What are the pre and post conditions on the main routines in your module? What are the important invariants on your data structures?

*Error Handling*—What is the range of legal values? What happens when other values are found? Test Cases -- List all your fiendish ideas for cases that might break the system or module or routine. If you do a good job here, you will be more sure of having a good design and most of your test plan will be already written.

*Concrete Implementation*—For (sub)modules that are data abstractions, give the concrete representation (Pascal declaration) here; the access functions will be listed as subparts. For routines, this is where the code goes. Make clear what data structures are private to the module. Define any technical terms relevant to the module.

*Side Effects*—Hopefully there aren't any, but if there are, you'd better make them explicit.

*Miscellaneous*—Just in case you think of anything else (for example, an estimate of how frequently various routines might be used or how much code will have to be written), throw it in!

Obviously, not all categories are needed at all levels. For access routines, for example all that is needed maybe a good function declaration. (Also, please don't call all of your procedures modules!) Where appropriate, write your description in Pascal or in a formal high-level language.

**RSP: In the above discussion, Kant uses terminology that is specific to her course. Refer to the SEPA Web Site for a discussion of recommended design document format. I would recommend against using the above section verbatim (as a handout) in that conflicting terms may cause some confusion.**

**Comments from previous students**

*. . . The specifications were extremely helpful when designing the code. Our specifications were detailed enough that attempting to write a piece of code for a module could almost be taken directly out off of the specifications. Writing the design specifications twice helped us a great deal. Many things were pointed out to us that needed improvement, and the second set of specifications gave us time to think things out a little longer.*

*. . . Every time I violated the principles of abstract design, I suffered. Abstract data types, modular decomposition, and information hiding can really work. The design process was definitely a major factor in simplifying the debugging process. It is interesting to note that we had almost no interface errors. We attribute this to good communication within the group.*

**9. Testing Plans Assignment**

Due Dates

Week 9. top level code for main modules, tested with drives and stubs, due

Week 10. test plan document due

**Overview**

**About this time students often start to fall behind or decide that the test plan isn't important. All I can suggest is making sure they stay on time beforehand or having them write test schedules as part of the design document.**

Testing is an important part of a software project. The software team must therefore carefully plan and document the order in which modules are to be integrated and the order in which the individual modules are to be completed and tested in isolation. Testing and debugging methods must also be agreed upon, documented, and eventually carried out. The grade you get on the test plan is tentative; a final grade will be assigned after you complete the testing process.

The purpose of a test plan document is to convince the management (in this case, the teaching staff) that a feasible test plan has been designed and also to provide the project members with directions for the testing phase of the project. This assignment outlines some constraints on the test plans. Several stages of tests must be scheduled, and several testing procedures must be explored. Scheduling is required for unit tests, integration testing, functional testing, performance evaluation, and an acceptance test (the demonstration). These tests must include practice in the techniques of walkthroughs, extensive logic testing, input/output testing, and optionally verification. The following sections give details of the contents and style of the test plan document, and the final section lists some reading material that should help you design a better test plan.

**Design the test plan**

The test and evaluation plan should contain the following components: statements of objectives and success criteria, integration plan, testing and evaluation methodologies, and responsibilities and schedules. These components are described in more detail below.

1. Objectives and Success Criteria: The test plan document should contain a statement of the overall testing objectives and the objectives of the individual tests that are planned.

2. Integration Plan: An important decision to be made about testing is the order in which modules are to be combined and therefore the order in which they will be tested individually. You must plan for individual module tests, the combination of modules during integration testing, a functional test, and an acceptance test (the demo).

The test plan document should describe and defend your integration method. The "big bang" method of integration is not acceptable, but many variants or combinations of top down and bottom up integration are acceptable if convincingly defended.

3. Testing Methodologies: You should design a plan to test all of the modules of your system. Your test plan document should name at least one module to which each [testing] technique is to be applied; you may decide the testing technique for the other modules later on. Each person on the software team is to perform at least one set of tests and turn in a couple of pages describing the process at a later date.

4. Responsibilities and Schedules: The test plan should provide a schedule describing dates and responsibilities.

- First, determine an order in which to perform integration and functional tests. In scheduling, you should make use of the dependency graphs based on the external functions that the modules require.

- Next, this order should be used to determine the order in which the individual modules are to be tested. Determine the dates by which tests are to be completed and the individuals responsible for the testing. Prepare a master test plan schedule and include it in the document.

- Finally, you must define a monitoring procedure to ensure that tests are designed and carried out on schedule. Someone will have to keep a notebook and report lack of compliance to the other team members. Similarly, there must be a procedure for reporting and correcting bugs.

**Writing the document**

The test plan document is to contain an introductory section that summarizes the whole document in a page or two and discussion sections that cover the plans in more detail. A total number of pages between 15 and 25 is about right. Remember that this document is intended for several audiences. The document should be organized so that it is easy to find schedule summaries and monitoring plans and easy directions on their personal responsibilities. The test plan should not be very long; if you find that the writing of the document is taking significantly longer than the design of the test plan, consult a member of the teaching staff. Representative samples of your tests are to be turned in later (as scheduled in the test plan).

The introductory section of the test plan document, which should only be about three pages long, should include the following:

- Overall objectives and success criteria

- Summary of the integration plan -- a list of tests and dates and people responsible

- Summary of the module-to-test-technique mapping for the four required testing techniques

- Summary of the monitoring, reporting, and correcting procedures

- Proposed dates for submission of individual test reports

The discussion sections should contain:

- Defense for the integration plan (about half a page)

- Details of the tests with objective and success criteria for each test or group of tests (assuming you have proposed most of the tests in your design document, there should not be more than 20 entries in this list, and each entry should be less than one page long; if you did not have good tests in your design document however, you should include them here)

- Details of the tests with objectives and success criteria for each test or group of tests (assuming you have proposed most of the tests in your design document, there should not be more than 20 entries in this list, and each entry should be less than one page long.

- Details of monitoring, reporting, and testing procedures (a page or so should suffice)

- Details of individual team member assignments (just a page or so, this is basically a cross reference list.

**Readings**

**SEPA assignment: See alternative course outline presented in section 4.2 of the *Instructor's Manual.***

**Comments by Previous Students**

*... we proposed to implement on a few functions in various categories, but instead actually tried to implement all. As a result, it was difficult to test the integrated system. It would have been better to try running a small system which handled all modes, rather than a large one that was buggy ... design was generally adaptable to the smaller, more complete system and certainly wasn't responsible for hindering us from building it.*

**10. User Manual**

Week 11. draft due

Week 12. final version due

This document should be a self contained description of how to use your system. A user manual should be a polished, professional piece of technical prose that a software company is proud to have accompany one of their products. (And it is a handy accomplishment to show off at job interviews!)

The document should have a structure that is evident to someone who is reading it straight through and someone looking for a particular topic or fact. A table of contents is required, and the organization that it reflects should be considered carefully. An index and Appendices might also be helpful.

Remember that the document should be completely self explanatory. Do not assume the reader has your functional specification. You may of course edit the sections of prose from your previous documents. Do not discuss any implementation unless it directly affects the user's interface with the system.

**RSP: I suggest to students that any interactive system, if properly designed, should be capable of being used without repeated and heavy *User Manual* references. If you haven't done so already, this might be a good place to discuss key aspects of User Interface Design (SEPA, Chapter 12).**

**In my opinion, the single most important part of the User Manual is the Index. Even if your students do not have the tools to develop one, you should emphasize its importance. In fact, if you don't have an indexing tool, this might be an excellent course project.**

Your User Manual should be no longer than 25-30 pages. A short command summary or pocket reference card might also be useful.

Your document ought to cover the following list of topics. The exact order in which you present material and whether certain topics are combined should be dictated by your particular project and your own writing style.

*Introduction*—a concise statement of what your program does, possibly including motivation and philosophy.

*How to use your system*—an overall description of the style of user interaction, device constraints and any arbitrary design choices you made that the user ought to know about.

*Detailed system operation*—an organized list of all use commands and when they are appropriate; some examples might be helpful [RSP note: mandatory in my courses!] A section for novices and experts are also possibilities.

*Error recognition and handling*—what to expect and what to do.

*An extended example*—show exactly what the user does and how the computer responds. Use appropriate conventions to indicate user actions and computer responses.

A *list of known "features"* [actions that some would call bugs!] and deficiencies

Everyone complains about how cryptic computer manuals are. This is your chance to show us what a good manual looks like.

**RSP: In her original paper, Professor Kant goes on to say that she has mixed emotions about the *User Manual,* indicating that students rush through it and often write what they hope the system will do as opposed to what it really does. On the other hand, learning how to properly prepare this form of documentation is one of the most valuable lessons for a budding software engineer.**

**In a modern context, the ‘user manual’ is an on-line help facility, but regardless of the presentation format, I would recommend attaching a reasonably heavy grade weight to this ‘document’—thereby forcing students to focus on it.**

**11. Demonstration**

You should prepare about 15 minutes of material that exhibit the best features of your system and allow about five minutes for questions and feedback from the audience. If your system has been properly debugged, you can let the audience tell you what to type to show off your error handling and user help facilities. Because time is so short, it is important to practice what you are going to say and do. You should give one practice demo to your TA about one week beforehand.

**Twenty minutes is not nearly enough time to do the demo properly, but it's enough time to give the other students a flavor for what has been done. Many of the groups gave another demo outside class so that there was enough time to find out how their system really performed. The deadline for the demo seemed to be a good forcing function for most students; without it, I suspect that many students would never finish their project.**

**12. Final Project Evaluation**

**I didn't give these a letter grade but did insist that they turn them in. I find the feedback helpful (and a good source of quotations!) and believe it gives the students a chance to express their feelings about the course.**

The purpose of this document is to evaluate your final project. The document provides an opportunity to step back and put things in perspective and appreciate how much you have accomplished. If your group doesn't agree on the evaluation and recommendations, feel free to write individual versions.

***Code and Demo Description***

You should turn in the complete code for your system and a transcript of your demo along with the rest of the evaluation.

***Functional Performance***

Discuss how well your project satisfies your original functional specifications. What has been added or subtracted? What advice would you give next year's students about writing functional specifications? Suggest improvements to the assignment description.

Compare the actual performance of your system to the predicted performance (LOC, memory requirement, response times, calculation times, etc.). Can you come up with reasons for the discrepancies (if any)?

***The Design Process***

Were you satisfied with your design process? How helpful was the process of writing out the specifications? How would you go about the design process next time? What differences, if any, would you propose for the overall and detailed design assignments?

***Testing***

Did the order of integration that you chose seem to work out reasonably well? Did you stick to your test plan schedule? If not, why, and what would you change if you had to do it again? What method did you use for recording test results?

Was debugging easier or harder than you expected? Do you think that the design process significantly reduced the amount of time needed for coding and debugging? Discuss the types of bugs you encountered. Were they limited to one module? The result of errors in the interface specifications? How did you handle fixing bugs?

***Management***

What was the real management structure within your group? Does it bear any resemblance to the structure that you had planned? Did you have any problems getting people to do their share of the work? Do you have any suggestions on how this could have been handled better?

Compare the actual time spent designing, coding, testing, debugging, and managing the project with your predicted times. Break this down by individual modules and individual people. Have you learned anything that will help you to make better predictions next time? What would you suggest to next year's students?

***The Real World***

Discuss what you might have done differently if this were a real world project. How much additional work would be required to complete the project to the level described in the functional specification? How much would you consider selling the system for? Do you think the system is worth its cost?

***Recommendations for Next Year***

What changes would you suggest for next year? Be more specific than "less work." Try to summarize what you have learned and which aspects of the course were most helpful. Which would you suggest shortening or eliminating? What changes or scheduling would you suggest? Were the regular weekly meetings with the TAs helpful? Would you recommend easier projects, or would you rather start with more specific specifications and decomposition, or would you rather not choose your own projects? What about the number of people in the group? Any other topics that you would like to have covered (or ignored)?

**13. Alternate Course Organizations**

**The course as it is currently organized is really too much work for one semester. It should get more credits for being a lab course or be split into two semesters. If there were two semesters, the first could be spent on the readings and on writing a very small system or a module for a system to learn about abstraction. This would also allow students to read about testing beforehand so that they can plan the design and implementation schedule for easy testing.**

**Another organization, which is unlikely at most schools, is to have a full semester schedule of software engineering -- a total immersion experience. This would involve a team teaching effort with lecture on technical writing and more serious grading of the assignments for writing style, a social psychology course focusing on the interactions of small groups (or a management course), and another set of computer science lectures on algorithms and data structures (or perhaps on compilers and operating systems.**

**Concluding Comments by RSP and BRM**

The course described by Professor Kant is an ambitious undertaking. The project flavor is essential for a complete understanding of software engineering, but the time pressure and demands of other courses may blunt its impact. It is possible to conduct the course with a series of controlled "mini-projects," each addressing one key software engineering activity, and still get all important points across.

Ideally, the course should span two terms—the first emphasizing readings and lectures and the second dedicated to a full term project (with necessary oversight). Such a course would be ideal for a senior year in CS or computer engineering, providing the student with a real world project flavor before he/she graduates.

Some professors have told us that they assign all reading in SEPA (the entire book!) during the first four to six weeks of a term, enabling students to get a feel for all aspects of software engineering. Lectures are paced throughout the term and the project commences immediately after the "reading period." We've never tried this, but everyone who has reports uniformly good results. Note: We think SEPA8e is too much reading for such a short time span and I don’t recommend this approach!

Kant's paper doesn't mention a number of important topics that should be covered in every software engineering course and encouraged in all projects. Project planning, elements of system engineering and software maintenance and SCM are important for all young software engineers to understand. Course content should include them! Formal technical reviews (walkthroughs) are an essential SQA mechanism. You should encourage your students to conduct them as part of their projects by requiring that Technical Review Summary Reports be submitted as part of the project deliverables.

You should note that software engineering courses can be successfully conducted without TAs and with somewhat less ambitious project goals. The design of your course should, of course, be guided by the backgrounds, interests and capabilities of your students. However, the overall framework proposed by Kant can serve as an excellent foundation.

1. SEPA 8/e conforms well to the *ACM/IEEE Computing Curriculum* and accommodates the "knowledge unit" concept well. This will be discussed in segment 3 of this Manual. [↑](#footnote-ref-1)
2. If your primary focus is Web-based systems and applications, you might consider a complementary book that David Lowe and Roger Pressman have completed—*Web Engineering: A Practitioner’s Approach,* McGraw-Hill, 2008. [↑](#footnote-ref-2)
3. The decision to drop data flow modeling was not taken lightly. Over the years, DFD modeling has been replaced with object-oriented methods and with UML modeling notation. Although it has served our community well, we feel it is time to retire DFD representations. [↑](#footnote-ref-3)
4. The complete report can be downloaded from **http://sites.computer.org/ccse/** [↑](#footnote-ref-4)
5. In my experience, more software projects fail because of lapses in good management practice than because of technical deficiencies. Maybe it’s time that computer science/engineering programs spend a bit more time on management subjects at the undergraduate level. [↑](#footnote-ref-5)
6. Portions of the paper have been deleted for brevity. [↑](#footnote-ref-6)