Software Engineering:

A Practitioner's Approach

Chapter 1
The Product

CHAPTER OVERVIEW AND COMMENTS

The goal of this chapter is to introduce the notion of software as a product designed and built by software engineers. Software is important because it is used by a great many people in society. Software engineers have a moral and ethical responsibility to ensure that the software they design does no serious harm to any people. Software engineers tend to be concerned with the technical elegance of their software products. Customers tend to be concerned only with whether or not a software product meets their needs and is easy to use.

1.1 The Evolving Role of Software

The main point of this section is that the primary purpose of software is that of information transformer. Software is used to produce, manage, acquire, modify, display, and transmit information anywhere in the world. The days of the lone programmer are gone. Modern software is developed by teams of software specialists. Yet, the software developer's concerns have remained the same. Why does software take so long to complete? Why does it cost so much to produce? Why can't all errors be found and removed before software is delivered to the customer?

1.2 Software

Software is not like the artifacts produced in most other engineering disciplines. Software is developed it is not manufactured in the classical sense. Building a software product is more like constructing a design prototype. Opportunities for replication without customization are not very common. Software may become deteriorate, but it does not wear out. The chief reason for software deterioration is that many changes are made to a software product over its lifetime. As changes are

made, defects may be inadvertently introduced to other portions of the software that interact with the portion that was changed.

1.3 Software: A Crisis on the Horizon

Many people have written about the "software development crisis". There seems to be an inordinate fascination with the spectacular software failures and a tendency to ignore the large number of successes achieved by the software industry. Software developers are capable of producing software the functions properly most of the time. The biggest problem facing modern software engineers is trying to figure out how to produce software fast enough to meet the growing demand for more products, while also having to maintain a growing volume of existing software.

1.4 Software Myths

The myths presented in this section provide a good source of material for class discussion. Managers need to understand that buying hardware and adding people does not spontaneously solve all software development problems. Customers need to understand that computer software does not make all their other problems go away. If the customer can not define his or her needs clearly, it is not possible to build a software product to meet these needs. If a customer does not have defined business processes without computer support, building computer software will not create these processes automatically. Software engineers must be committed to the concept of doing things right the first time. Software quality does not happen on its own after a product is finished. Quality must be built into every portion of the software development process.

PROBLEMS AND POINTS TO PONDER

1.1. Classic examples include the use of "digital automobile dashboards" to impart a high tech, high quality images. Appliances that "think;" the broad array of consumer electronics; personal computers (today, differentiated more by their software function than the hardware), industrial instrumentation and machines. All e-commerce applications are differentiated by software.

1.2. The apprentice/artist culture of the 1950s and 1960s worked fine for single person, well constrained projects. Today, applications are more complex, teams work on large projects, and software outlives generations of developers. Yet, the culture established in the early days dies hard, leading more than a little resistance to more disciplined methods. In addition (see Chapter 29), the new generation of Web application developers is repeating many of the same mistakes that programmer made during the circa 1965.

1.3. This is a good problem for classroom discussion (time permitting). Rather than focusing on cliche' ridden (albeit important) issues of privacy, quality of life, etc., you might want to discuss "technofright" and how software can help to exacerbate or remedy it. Another interesting possibility is to use Neumann's "Risks" column in SEN to key discussion. You might also consider new attempts at software-based 'cash' economies, new modes of interactive entertainment, virtual reality, e-commerce, etc.

1.5. There are literally dozens of real life circumstances to choose from. For example, software errors that have caused major telephone networks to fail, failures in avionics that have contributed to plane crashes, computer viruses (e.g., Michaelangelo) that have caused significant economic losses. Attacks on major e-commerce sites.

1.6. The two broadest categories encompass risks associated with economic loss and risks to the well-being of people. You might suggest that each student select five risks (culled from the sources noted) and present them to the class. Encourage them to look for humorous as well as serious risks.

1.7. To be honest, most professors do not assign papers as part of software engineering courses that use SEPA. Instead, a term project (or a number of somewhat smaller projects) are assigned. However, you might consider discussing software engineering issues as they relate to the topics noted in this problem.

1.8 Another way to characterize this problem is to suggest that each student describe a software myth that he or she believed that has subsequently been dispelled. From the student's point of view, myths will likely be driven by programming projects that they've attempted earlier in their career. To give this problem a contemporary feel, you might consider the myths that have evolved around the development of Web-based applications.

Chapter 2
The Process

CHAPTER OVERVIEW AND COMMENTS

This chapter discusses several process models used by professional software developers to manage large-scale software projects. No software process works well for every project. However, every project needs to conform to some systematic process in order to manage software development activities that can easily get out of control. Software processes help to organize the work products that need to be produced during a software development project. Ultimately the best indicator of how well a software process has worked is the quality of the deliverables produced. A well-managed process will produce high quality products on time and under budget.

2.1 Software Engineering - A Layered Technology

Software engineering encompasses a process, the management of activities, technical methods, and use of tools to develop software products. Software is engineered by applying three distinct phases (definition, development, and support). Students need to understand that maintenance involves more than fixing bugs.

2.2 The Software Process

This section discusses the concept of a software process framework and provides a brief overview of the Software Engineering Institute Capability Maturity Model. It is important to emphasize that the Capability Maturity Model does not specify what process model needs to be used for a given project or organization. Rather, it provides a means of assessing how well an organization's processes allow them to complete and manage new software projects.

2.3 Software Process Models

The terms "software process model" and "software engineering paradigm" are used interchangeably in the literature. This chapter presents overviews of several software process models. 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. Another difficulty students have is their belief that each phase of a process is performed completely independently of the other phases. The reality is that there tends to be lots overlap among the phases.

2.4 The Linear Sequential Model

The linear sequential model is also known as the "classic life cycle" or "waterfall model". System development proceeds though the phases (analysis, design, coding, testing, support) in order. This is a good model to use when requirements are well understood. If a phase must be revisited in this model, process failure is indicated (more thorough requirements analysis is needed).

2.5 The Prototyping Model

This model is good to use when the customer has legitimate needs, but is not able to articulate the details at the start of the project. A small mock-up of a working system is developed and presented to the customer. Sometimes this first system is discarded and sometimes it is extended based on the customer's feedback.

2.6 The RAD Model

The rapid application deployment model is a high-speed adaptation of the linear sequential model. Project requirements must be well understood and the project scope tightly constrained. Developers are often able to use component-based construction techniques to build a fully functional system in a short time period.

2.7 Evolutionary Models

The two most important models in this section are the incremental model and the spiral model. The incremental model combines elements of the linear sequential model applied repetitively with the iterative philosophy of the prototyping model. Each increment produces a working version of a

software product with increasing functionality. There is no throwaway code. The spiral model also combines the iterative nature of prototyping with the systematic control found in the linear sequential model. An essential component of the spiral model is that assessment of both management and technical risks is performed as each incremental release is completed.

2.8 Component-Based Development

Object-based technologies provide the technical framework for component-based software engineering. The component-based development (CBD) model incorporates many of the iterative characteristics of the spiral model. The main difference is that in CBD the emphasis is on composing solutions from prepackaged software components or classes. This CBD emphasizes software reusability. The unified software development process is an example of CBD that has been proposed for industrial use. The unified modeling language (UML) is used to define the components and interfaces used in the unified software development process.

2.9 The Formal Methods Model

Formal methods in software development require the use of rigorous mathematical notation to specify, design, and verify computer-based systems. Mathematical proof is used to verify the correctness of a system (not empirical testing). Cleanroom software engineering is an example of this approach. While formal methods have the potential to produce defect-free software, the development of formal models is both time-consuming and expensive.

2.10 Fourth Generation Techniques

This is a broad class of techniques. The general idea is that a software tool is used to describe a system in manner understood by the customer using a specialized design language or graphical notation. In the ideal case, the tool then generates source code from this system description that can be compiled into a running system. The running system then needs to be tested and refined using other software engineering processes. There is some risk that the customer is not able to describe the system in sufficient detail or that the initial system will be deployed without sufficient testing.

Section 2.12 uses a short essay by Margaret Davis to put process and product issues into perspective. If you're teaching a graduate course, I'd recommend Phillip Howard's *The Death of Common Sense,* as outside reading on the failures of a wholly process-oriented mind set.

PROBLEMS AND POINTS TO PONDER

2.1. You might suggest that students use the *Further Readings and Information Sources* section of Chapter 8 for pointers.

2.2. The support phase is applied differently for embedded software. In most cases, embedded software is defined and developed, but once it is placed in its host environment, it is not likely to change until a new release of the product occurs.

2.3. The latest SEI information can be obtained at:

http://www.sei.cmu.edu/

2.4. In each case status quo, problem definition, technical development, and solution integration are applied at a different levels of abstraction. For example, problem definition at the product requirements level would entail many meetings with marketing and even potential end-users; "technical development" at the product requirements level would demand the creation of a product specification; "solution integration" would require detailed review against customer requirements and modifications. At a lower level of abstraction (e.g., generate code ...) the nature of these activities would change, moving away from customer related information and toward implementation specific information.

2.5. Assign this problem as is if the majority of your class is composed of industry practitioners. If your students are "green," suggest a project scenario and ask the students to identify the appropriate paradigm. The key here is to recognize that the "best" paradigm is a function of the problem, the customer, the environment, the people doing the work and many other factors. My choice — all thing being equal — is the evolutionary approach.

2.6. Software applications that are relatively easy to prototype almost always involve human-machine interaction and/or heavy computer graphics. Other applications that are sometimes amenable to prototyping are certain classes of mathematical algorithms, subset of command driven systems and other applications where results can be easily examined without real-time interaction. Applications that are difficult to prototype include control and process control functions, many classes of real-time applications and embedded software.

2.7. Any software project that has significant functionality that must be delivered in a very tight (too tight) time frame is a candidate for the incremental approach. The idea is to deliver functionality in increments. Example: a sophisticated software product that can be released to the marketplace with only partial functionality—new and improved versions to follow!

2.8 Any project in which tight time-lines and delivery dates preclude delivery of full functionality is a candidate for the incremental model. Also any software product in which partial functionality may still be saleable is a candidate.

2.9. As work moves outward on the spiral, the product moves toward a more complete state and the level of abstraction at which work is performed is reduced (i.e., implementation specific work accelerates as we move further from the origin).

2.10. I would suggest postponing this problem until Chapter 27 is assigned, unless you do not intend to assign it. In that case, it can serve to introduce the importance of reuse.

2.11. Stated simply, the concurrent process model assumes that different parts of a project will be a different stages of completeness, and therefore, different software engineering activities are all being performed concurrently. The challenge is to manage the concurrency and be able to assess the status of the project.

2.12. An SQL, a spreadsheet, a CASE code generator, graphical tools like VisualBasic, Powerbuilder, Web page construction tools

2.13. The product is more important! That’s what people use and what provides benefit to end users.

Chapter 3

Project Management Concepts

CHAPTER OVERVIEW AND COMMENTS

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 customers and developers is essential. The 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.

3.1 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 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.

3.2 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.

3.3 The Problem

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 must be decomposed along functional lines into smaller, more easily managed subproblems.

3.4 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.

3.5 The Project

The text lists 10 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).

3.6 The W^{5}HH Principle

Boehm's W^{5}HH 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.

3.7 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.

PROBLEMS AND POINTS TO PONDER

3.1. Ten commandments:

1. Thou shalt get smarter (provide education).
2. Thou shalt focus on quality.
3. Thou shalt listen to the customer.
4. Thou shalt understand the problem.
5. Thou shalt work within a repeatable process.
6. Thou shalt not agree to ridiculous schedules.
7. Thou shalt measure the product, the process and yourself.
8. Thou shalt document the work in the most effective way.
9. Thou shalt remember that others will also work on the software.
10. Thou shalt continually improve

3.2. The latest SEI information can be obtained at:

http://www.sei.cmu.edu/

3.3. Same person: (1) an engineer who must develop a program for personal use; (2) a business person creating a spreadsheet model for personal use; (3) an entrepreneur who has a new concept for a killer App. Different person: (1) an IS department servicing some business function; (2) a software development group servicing marketing needs; (3) a contractor building to customer specs.

3.4. In today's environment downsizing and outsourcing have the most immediate and significant impact. In addition, "expense reduction measures" that lead to lower product quality; unrealistic project deadlines; failure to "listen" to the customer, or

conversely, to warnings noted by the software engineers doing the work.

3.6. A controlled decentralized team is one option. Since requirements are well defined, it will be possible to partition requirements and allocation to subteams. The large size of the project also mitigates in favor of a CD team. Since there is no discussion of schedule, we assume that delivery date is reasonable. Therefore, it might be possible to use a linear sequential process model (work has been done before). However, an iterative model (e.g., spiral) is also a good possibility.

3.7. The DD team is probably the only viable option, given hazy requirements and the experimental nature of the work. A prototyping approach or an evolutionary process model should be used.

3.8. A CC team is probably best, given time pressure and familiarity with the work (however, a CD team might also work well). An incremental process model is indicated by the deadline driven nature of this work.

3.9. A CD team is probably best, given that the work is experimental, but that there is a business deadline. Another possibility is to use a DD team. An incremental process model or an evolutionary process model could be used, given the deadline driven nature of this work.

3.10. Much of the problem has to do with the structure of documents and when they are created. Documents should be kept lean and should be generated as software engineering work is being conducted, NOT after the fact. If this is done, the perceived value of documents will improve.

3.11. The GradeAnalyzer application will obtain grades for all undergraduate and graduate for-credit courses contained in the Registrar's data base of courses for a given semester. GradeAnalyzer will read all grades for each course and compute the average grade using a numerical scale in which an A = 4.0 and other grades are assigned values as indicated in grade value document UC29-1. GradeAnalyzer will print and/or display a report that lists each course, the instructor, the average grade. The report may be sorted by department, by highest to lowest average grade, by instructor or by any permutation of these characteristics. GradeAnalyzer will be implemented to run under Microsoft Windows 200x environment.

3.12. A simple decomposition:

page layout

define page parameters

allocate text regions

allocate graphical regions

define emphasis (lines, shading, etc.)

input/import text

input/import graphics

edit text

edit graphics

outpage/export page

end page layout

Chapter 4

Software Process and Project Metrics

CHAPTER OVERVIEW AND COMMENTS

This chapter provides an introduction to the use of metrics as a mechanism for improving the software development process and managing software projects. A more complete discussion of software quality assurance appears later in the text. 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.

4.1 Measures, Metrics, and Indicators

The important point to get across in this section is that measures, metrics, and indicators are distinct (though related) entities. A measure is established when a single data point is collected. A software metric relates individual measures in a meaningful way. An indicator is a metric or combination of metrics that provide insight into the software project, process, or product.

4.2 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.

4.3 Software Measurement

In this section the differences between direct measures (e.g. LOC or defects over time) and indirect measures (e.g. functionality or quality) in software engineering are discussed. 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. Function points were originally developed for information systems applications. Feature points and 3D function points are extensions of the function point measures to engineering applications involving real-time programming or process control.

4.4 Reconciling Different Metrics Approaches

This table presented in this section provides a good resource for students to use when estimating LOC for their projects from function points prior to writing the source code. Students should be cautioned against the practice of backtracking (using the table to compute function point from the LOC measure for a completed program). A key point: using LOC or FP in project estimation (discussed in Chapter 5) requires the existence of baseline historical data.

4.5 Metrics for Software Quality

A detailed discussion of software quality assurance appears later in the text. This section is intended to get students thinking about the role of measurement to assess product quality and process effectiveness during project development. Students should also be made aware of the fact that the factors that defined software quality in the 1970's (correctness, maintainability, integrity, usability) continue to define software quality today. Defect removal efficiency is an important software quality metric that can useful at both the process and project levels.

4.6 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.

4.7 Managing Variation—Statistical Quality Control

A complete discussion of statistical process control is beyond the scope of this text. However, the idea of using a control chart to identify out of control processes is understandable to students. Implicit in using this technique is the necessity of having access to baseline historic metric data to compute the standard deviations and means required to apply the zone rules.

4.8 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.

4.9 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.

PROBLEMS AND POINTS TO PONDER

4.1. For an automobile:

measures: weight, wheelbase, horsepower

metrics: max speed per quarter mile; breaking distance at 30 mph; miles between schedule engine maintenance

indicator: defects per vehicle at delivery; cost per vehicle at manufacture; parts per vehicle at manufacture

4.2. For a dealership:

measures: square feet of showroom space; number of service bays; number of sales people

metrics: units sold per month; units sold per salesperson; cars serviced per month

indicators: $ profit per sale; $ profit per service call; number of repeat customers/total number of customers

4.3. A process metric is used to assess the activities that are used to engineer and build computer software (with the intent of improving those activities on subsequent projects). A project metric is used to assess the status of a software project.

4.4. Metrics should NOT be used as rewards or punishments at an individual level. Hence, any metric that measures individual productivity or quality should be private. Group and process metrics may be public, as long as they are not used for reprisals.

4.6. Additional rules of etiquette: (1) don't look for the perfect metric ... it doesn't exist; (2) be sure to measure consistently so that apples and oranges comparisons are not made; (3) focus on quality, it's safer!

4.7. For the "missing" rib:

spine: customer forgot requirement

ribs: customer not properly queried

customer misunderstands requirements

spine: requirement inadvertently omitted

ribs: improper review techniques

analyst error or sloppiness

For the "ambiguous" rib:

spine: vague wording

ribs: improper review techniques

lack on specification training

misunderstanding of requirements

spine: misunderstood or unclear requirement

ribs: customer not properly queried

customer misunderstands requirements

analyst misunderstands requirements

For the “change” rib:

spine: legitimate business modification

spine: correcting error in specification

spine: correcting ambiguity in specification

4.8. An indirect measure is used when the characteristic to be assessed cannot be measured directly. For example, “quality” cannot be measured directly, but only by measuring other software characteristics. Some examples:

Direct metrics:

1. number of pages

2. number of figures

3. number of distinct documents

Indirect measures:

1. readability (use the FOG index)

2. completeness (count the number of "help desk" questions that you receive)

3. maintainability (time to make a documentation change)

Many metrics in software work are indirect because software is not a tangible entity where direct dimensional or physical measures (e.g., length or weight) can be applied.

4.9. The “size” or “functionality” of the software produced by both teams would have to be determined. For example, errors/FP would provide a normalized measure. In addition, a metric such as DRE would provide an indication of the efficiency of SQA within both teams’ software process.

4.10. LOC is weak because it rewards “verbose” programs. It is also difficult to use in a world where visual programming, 4GLs, code generators and other 4GTs are moving development away from 3GLs.

4.11. Using the table noted in Figure 4.5 with average values for complexity and average values for each CAF,

FP = 661.

4.12. Using the table noted in Figure 4.5 with average values for complexity, and assuming that the weight for transitions is 1 across all complexity values, we compute 905 3-D function points.

4.13. The backfiring technique can be used here. From the table in Section 4.4,

C: 128 LOC/FP

4GL: 20 LOC/FP

Therefore, 32,000/128 + 4,200/20 = 250 + 210

= 460 FP (approximate)

4.16. "Spoilage" is defined as the cost to correct defects encountered after the software has been released to its end-users. It is possible for spoilage to increase even though defects/KLOC decreases. This occurs when SQA techniques have not found the "difficult" defects. These consume large amounts of maintenance/rework time and hence cost significantly more on average that less dramatic defects.

4.17. No. The level of abstraction of most 4GLs is too high.

4.18 You might suggest that students implement the calculations required for this problem (see Section 4.7) using a spreadsheet model. Results for this case are:

Mean:	0.77			
UNPL:	1.15			
LNPL:	0.381			
Stdev.	0.127			
1	stdev	above	mean:	0.89
2	stdev	above	mean:	1.02

Data should be plotted using the approach outlined in the book.

Chapter 5

Software Project Planning

CHAPTER OVERVIEW AND COMMENTS

This chapter provides students with basic information on planning software projects. Software planning 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.

5.1 Observations on Estimating

The point to get across in this section is that project estimation is a difficult task to do without both historical data and a fair amount of experience with the proposed application to draw on. None the less, it is an activity that needs to be done for all projects in order to begin the process of quality management. Project complexity, project size, and structural uncertainties affect the reliabilities of the estimates. The topic of risk management is discussed more fully in the next chapter.

5.2 Project Planning Objectives

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.

5.3 Software Scope

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.

5.4 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.

5.5 Software Project Estimation

Software is now the most costly element of virtually every computer 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.

5.6 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 planing 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.

5.7 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 web site. Similarly, the details of the Software Equation will need to be located on the web.

5.8 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 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.

5.9 Automated Estimation Tools

This section lists some areas where automated estimation tools might be useful to project planners. It is unfortunate that the results obtained from different estimation tools show a wide variance for the same project data. It is also true that the

predicted values are often significantly different from the actual values.

PROBLEMS AND POINTS TO PONDER

5.1. This is the first appearance of the *SafeHome* system that will recur throughout the book. Even if you normally don't assign homework, you might try this one. What you're looking for is a reasonable work breakdown structure, but your students will struggle because they don't have a bounded scope. You can help them by defining the basic characteristics of *SafeHome* (see later chapters, use the SEPA index for pointers to the problem).

5.2. Sometimes, complexity arises from a poorly established interface between the customer and the software developer. Discounting that, the following technical characteristics should be considered:

real-time attributes

multiprocessing requirement

(concurrency)

nature of the algorithm

requirement for recursion

nature of input

determinacy of input

nature of output

language characteristics

. . . and knowledge/experience of staff on application.

5.3. Performance:

- real time application—raw processing of data measured by CPU time and possibly interrupt servicing efficiency
- engineering/scientific applications—numerical accuracy and for large systems, CPU time.
- commercial applications—I/O efficiency
- interactive applications—user "wait time"
- microprocessor applications—CPU time and memory requirement

5.4. The SafeHome system is analyzed in detail in Chapter 12. For our purposes here, we do a simple functional decomposition:

- user interaction (2400)
- sensor monitoring (1100)
- message display (850)

- system configuration (1200)
- system control [activation/deactivation] (400)
- password processing (500)

LOC estimates (in the C language) for each function are noted in parentheses. A total of 6450 LOC are estimated. Using the data noted in the problem:

6450 LOC / 450 LOC/pm = 14.3 pm

Cost: 14.3 pm * $7,000/pm = $100,000 (approximate)

5.5. Assuming that the SafeHome system is implemented in C, we can use backfiring for a quick FP estimate:

6450 LOC / 128 LOC/FP = 50 FP (approximate)

5.7. Using the relationship noted in equations 5.4 (note that this project is a bit small for these equations) and the discussion that follows, assuming B=0.16, PP = 3,000,

t.min = 8.14 (6450/3000)^0.43

= 8.14 * 1.38 = 11.31 months

E = 180 * 0.16 * (0.94)^3

= 24 months

5.8. The estimates are all within a reasonable range with the software equation representing the most conservative estimate. Note however, that the SafeHome project is small for software equations 5.4, therefore putting this result in doubt.

Given the values computed in the above problems, it likely that an estimate in the range 15 - 18 person-months would be appropriate for this project.

5.9. The software equation predicts "no" but we may be outside its bounds. Using the original COCOMO model (not COCOMO II),

D = 2.5 E ^ 0.35

= 2.5 * 16.92 ^ 0.35

= 6.7 person-months

It appears that 6 months is aggressive, but probably possible, given the results of COCOMO and the size/complexity of the SafeHome project.

5.12. Costs and schedule are estimated early because such information is demanded (by upper management) as early as possible. If a project is extremely complex with high technological risk and a fixed price proposal is to be submitted, costing of the project should (if possible) be delayed until after requirements analysis. Note: the cost of requirements analysis alone can be estimated early.

Chapter 6

Risk Management

CHAPTER OVERVIEW AND COMMENTS

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 plans (RMMM). 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 this material.

6.1 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 crisis management. A better strategy is to anticipate the occurrence of potential risks and to prepare plans for managing them before they become problems that cause a project to fail.

6.2 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.

6.3 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.

6.4 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 though 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.

6.5 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.

6.6 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 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.

6.7 Safety Risks and Hazards

This section provides an opportunity to introduce a computing ethics topic. 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.

6.8 The RMMM Plan

The template RMMM plan appears on the text web site. Students should be encouraged to examine its headings. An important new point in this section is 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).

PROBLEMS AND POINTS TO PONDER

6.1. Reactive risk management occur 'once the horse has left the barn.' A few examples: putting a stop sign at a dangerous corner only after a fatal accident has occurred; fixing a pothole only after the city has been sued by an angry motorist. In essence, we react instead of planning.

6.2. Known risks are those that are determine through careful evaluation of the project and technology. Predictable risks are extrapolated from past experience.

6.4. Technology risks:

- rapid changes in digital format for video data
- changing compression algorithms and format
- rapid changes in processing power and bus architectures
- rapid changes in video input modes (e.g., via internet, direct from camera, across LAN, from analog tape, from DAT)

All of the above represent technology risk for the project.

6.5. Business risks here are probably the most relevant, although your students will likely select technical and project risks. Be sure to indicate that business risks (i.e., large market share of existing WPs with significant penetration problems); industry standard WPs (e.g., MSWord) will cause reticence to buy, etc. may be significantly more important than technical or project risks.

6.6. Risk components indicate the four areas of impact that will be affected by risk. That is, risk can impact performance, cost, support, or schedule. Risk drivers are the risks that will have a focused impact on the risk components. For example, some risks will have an affect on schedule; other risks might drive the performance component.

6.7. Realistically, the scheme noted is probably sufficient. See Karolak for more detailed risk model with more sophisticated weighting.

6.8 through 6.10. A single example of RMMM is provided to illustrate the approach:

Risk: from table — "Lack of training on tools"

Mitigation: (1) Develop a training schedule for all software staff that is "just-in-time." That is, training will be provided just before the tool is required for use. (2) Have one or two experts on staff for each tool. These people will be available to mentor others in the use of the tool and answer questions.

Monitoring: (1) Log number of hours tools are being used. (2) debrief staff to determine how tools are perceived; whether frustration is setting in; (3) examine work products created using tools for quality; (4) determine time required to create work products using tools vs. manual approach—look for anomalies.

Management: Assumes that tools are <u>not</u> working. Determine reasons why. If staff are untrained, then provide one-to-one mentoring/training using staff experts and vendor trainers (budget for this contingency). If training is OK but tools don't work, then consider alternative work product generation approaches using a semi automated approach, e.g., word processors and graphical tool and a stripped down version of required models.

6.13. If a risk is high probability, high impact, but the project team cannot do anything to mitigate, monitor, or manage it, it should not appear in the risk table. For example, if the company goes out of business in the

middle of the project, the impact is catastrophic. Assume the probability of this is high, given outside business pressures. The software project team will likely have no control over the outcome.

6.14. The referent curve can be skewed to emphasize the impact of either schedule risks or costs risks. In general, schedule risks are more important. For example, if a risk would result in a schedule delay that causes a market window to be missed, the project must be cancelled (no market for the product).

6.15. An excellent source of information is Nancy Leveson's book (see SEPA Chapter 8).

6.16. Any application area in which human life can be affected by defects is a candidate: medical equipment, avionics, power plant control (especially nuclear), automobile control systems, radar. In addition, systems that are involved in national security and systems that facilitate major economic transactions are also candidates.

Chapter 7

Project Scheduling and Tracking

CHAPTER OVERVIEW AND COMMENTS

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.

7.1 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 not a seat of the pants activity any more. 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.

7.2 The Relationship Between People and Effort

The most important point to get across in this section 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 can not 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.

7.3 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.

7.4 Selecting Software Engineering Tasks

This section points out that 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 on the text web site.

7.5 Refinement of Major Tasks

This section 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.

7.6 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.

7.7 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 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 MS Schedule. The time-boxing procedure described at the end of this section is a time management strategy that students should made aware of.

7.8 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.

7.9 Error tracking

The important point to get across regarding error tracking is that metrics need to be defined so that the project manager has a way to measure progress and proactively deal with problems before they become serious. This also implies the need have historical project data to determine whether the current metric values are typical or atypical at a given point in the development process.

7.10 The Project Plan

This section merely reviews the purposes of the software project plan. Three of these purposes are the presentation of project costs, resource needs, and schedule. Students need to be aware that the project plan needs to be updated (e.g. risks, estimates, schedule, etc.) as a project proceeds and the team learns more about the project itself.

PROBLEMS AND POINTS TO PONDER

7.1. Document your reservations using quantitative arguments derived from past project metrics. Then suggest an incremental approach (see Chapter 2) that offers to deliver partial functionality in the time allotted with complete functionality following.

7.2. A macroscopic schedule defines major activities and tasks. It is the table of contents for the project and may be sufficient for small efforts. For larger projects, a detailed schedule, defining all work tasks, work products, and milestones should be developed.

7.3. In most cases, milestones are tied to the review of a work product. However, some milestones (e.g., integration testing complete) may be tied to successful completion of an integration test plan, rather than a review.

7.4. The reduction of rework is the key to your argument. If good reviews reduce rework significantly, it can be argued that the time spent on communication (during reviews) results in time savings (due to less rework) overall. The crux of this argument is as follows: Typically, it costs more than 10 times as much to find and correct an error during testing as it does to find and correct the same error during, say, design.

Therefore, if one hour of communication finds one error during design, it saves 10 hours of work later on. Big payoff!

7.5. If a project is compartmentalized and well organized; if good documentation has been done; if interfaces are clearly defined; if parallel tasks are possible; if the people who are added are generally competent; if newcomers can pick up what they need form the configuration, rather than the software engineers currently working on the project, you can add people to a project without negative impact.

7.6. The relationship can be developed by solving for t_d and varying the number of person months and duration as indicated.

Problems 7.7 through 7.11. Be sure that your students first establish the scope of the OLCRS project. Depending on requirements, this can become quite complex. An alternative is for you to provide a requirements handout so that everyone is working from the same scope.

In general, a structured, new application development project should be chosen.

The project team should have at least three people. This will force students to consider parallel activities.

A tool such as Microsoft Project will be extremely useful to illustrate important concepts.

7.13 Use the steps defined in Section 7.8 to computer earned value. First sum all planned effort through task 12. In this case,

BCWS = 126.50
BAC = 156.50
BCWP = 127.50

SPI = BCWP/BCWS = 127.5/126.5 = 1.008
SV = BCWP - BCWS = 127.5 - 126.5 = 1.0 person-day

percent complete = BCWP/BAC = 81%

Other earned-value data are computed in the same way.

Chapter 8

Software Quality Assurance

CHAPTER OVERVIEW AND COMMENTS

This chapter provides an introduction to 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.

8.1 Quality Concepts

An important concept in this section 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 satisfaction is every bit as important to modern quality work as is quality of design and quality of conformance.

8.2 The Quality Movement

Total quality management (TQM) is described in this section. TQM stresses continuous process improvement and can be applied to software development. The most important point made in the section is that until a visible, repeatable, and measurable process is created not much can be done to improve quality.

8.3 Software Quality Assurance

This section describes software quality as conformance to explicitly stated requirements and standards, as well as implicit characteristics that customers assume will be present in any professionally developed software. The SQA group must look at software from the customer's perspective, as well as assessing its technical merits. The activities performed by the SQA group involve quality planning, oversight, record keeping, analysis and reporting. SQA plans are discussed in more detail later in this chapter.

8.4 Software Reviews

This section describes the purposes of conducting software reviews. It is important to point out to students that any work product (including documents) should be reviewed. Students are usually impressed by the fact that conducting timely

reviews of all work products can often eliminate 80% of the defects before any testing is conducted. This message often needs to be carried to managers in the field, whose impatience to generate code sometimes makes them reluctant to spend time on reviews.

8.5 Formal Technical Reviews

The mechanics of conducting a formal technical review (FTR) are described in this section. Students should pay particular attention to the point that it is the work product that is being reviewed not the producer. Encouraging the students to conduct formal reviews of their development projects is a good way to make this section more meaningful. Requiring students to generate review summary reports and issues lists also helps to reinforce the importance of the review activities.

8.6 Formal Approaches to SQA

This section merely introduces the concept of formal methods in software engineering. More comprehensive discussions of formal specification techniques and formal verification of software appear Chapters 25 and 26.

8.7 Statistical Quality Assurance

Statistical quality assurance is beyond the scope of this text. However, this section does contain a high level description of the process and gives examples of metrics that might be used in this type of work. 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.

8.8 Software Reliability

Software reliability is discussed in this section. 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.

8.9 A Software Quality Assurance Approach

This section describes the use of poka-yoke devices as mechanisms that lead to the prevention of potential quality problems or the rapid detection of quality problems introduced into a work product. Examples of poka-yoke devices are given, but students will need to see others (a web reference is given in the text).

8.10 The ISO 9000 Quality Standards

The ISO 9000 quality standard is discussed in this section 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 is described as the quality assurance standard that contains 20 requirements that must be present in any software quality assurance system.

8.11 The SQA Plan

The major sections of a SQA plan are described in this section. 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 have the students look at the material in Chapters 17-19 (testing and technical metrics) before beginning this assignment.

In addition to the review checklists contained within the SEPA Web site, I 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?

PROBLEMS AND POINTS TO PONDER

8.1. We look for variation in the traceability from requirements to design and design to code. We assess the variation in the form and content of work products. We look for variation in the process — repeatable process is a goal. We look for variation in expected and actual results derived from software testing.

8.2. In reality, if we define quality as "conformance to requirements," and requirements are dynamic (keep changing), the definition of quality will be dynamic and an assessment of quality will be difficult.

8.3. Quality focuses on the software's conformance to explicit and implicit requirements. Reliability focuses on the ability of software to function correctly as a function of time or some other quantity. Safety considers the risks associated with failure of a computer-based system that is controlled by software. In most cases an assessment of quality considers many factors that are qualitative in nature. Assessment of reliability and to some extent safety is more quantitative, relying on statistical models of past events that are coupled with software characteristics in an attempt to predict future operation of a program.

8.4. Yes. It is possible for a program to conform to all explicit functional and performance requirements at a given instant, yet have errors that cause degradation that ultimately causes the program to fail.

8.5. Absolutely. Many programs have been patched together or otherwise "kludged up" so that they work, yet these program exhibit very low quality if measured by most of the criteria described in Figure 17.1.

8.6. There is often a natural "tension" that exists between these two groups. The reason is simple: if the SQA group takes on the role of the "watch dog," flagging quality problems and high-lighting shortcomings in the developed software, it is only normal that this would not be embraced with open arms by the software engineering group. As long as the tension does not degenerate into hostility, there is no problem. It is important to note, however, that a software engineering organization should work to eliminate this tension by encouraging a team approach that has development and QA people working together toward a common goal—high quality software.

8.7. Institute formal technical reviews. After these are working smoothly, any of a number of SQA activities might be implemented: change control and SCM (see Chapter 9); comprehensive testing methodology (see Chapters 16 and 17); SQA audits of documentation and related software. Also software metrics can help (Chapters 4, 18, and 23)

8.8. Any countable measure that indicates the factors noted in Chapter 18 are candidates. For example, maintainability as measured by mean-time-to-change; portability as measured by an index that indicates conformance to language standard; complexity as measured by McCabe's metric, and so on.

8.9. Typically, an unprepared reviewer will be reading the product materials while everyone else is conducting the review; will be especially quiet throughout the review; will have made no annotation on product materials; will make only very general comments; will focus solely on one part (the part he/she read) of the product materials. As a review leader, you should ask if the person has had time to prepare. If most reviewers have not prepared, the review should be cancelled and rescheduled.

8.10. Assessing style is tricky and can lead to bad feelings if a reviewer is not careful when he/she makes comments concerning style. If the producer gets the feeling that the reviewer is saying, "Do it like I do," it is likely that some resentment will arise. In general, the review should focus on correctness.

8.12. You might suggest that students create a spreadsheet model using a table like Table 8.2 and the equations discussed in Section 8.7 to do the necessary computation.

8.13 The May, 1995 and January, 1996 issues of *IEEE Software* have useful articles.

8.14. For hardware the MTBF concept is based on statistical error data that occurs due to physical wear in a product. In general, when a failure does occur in hardware, the failed part is replaced with a spare. However, when an error occurs for software, a design change is made to correct it. The change may create side effects that generate other errors. Therefore, the statistical validity of MTBF for software is suspect.

8.15. Classic examples include aircraft avionics systems, control systems for nuclear power plants, software contained in sophisticated medical instrumentation (e.g., CAT scanners or MRI devices) control systems for trains or subway systems; elevator control systems.

8.17 device: Develop a simple parsing algorithm that will detect common errors in e-mail addresses (e.g., bad characters, double dots)

detection device: recognize an e-mail message that has no content (e.g., e-mail text is null)

8.18 See the SEPA Web site for links to ISO sites.

Chapter 9

Software Configuration Management

CHAPTER OVERVIEW AND COMMENTS

This chapter presents a high level overview of the issues associated with software configuration management, version control, and change control. 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.

9.1 Software Configuration Management

This section defines software configuration items as programs, documentation, and data. Students need to be aware that changes can occur to any of these items during the course of a project. The impact of these changes can have a negative ripple affect 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.

9.2 The SCM Process

Software configuration management (SCM) is defined as having change control as its primary responsibility. SCM is also concerned with auditing the software configuration and reporting all changes applied to the configuration. The next sections cover the five SCM tasks in more detail.

9.3 Identification of Objects in the Software Configuration

This section 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 E-R diagrams may struggle with the concepts introduced here. It may be desirable to review some of the object-oriented programming concepts from Chapter 20 while covering this chapter.

9.4 Version Control

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.

9.5 Change Control

This section discusses the details of processing and controlling a change request for an SCI. 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 SCCS 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.

9.6 Configuration Audit

This section contains a list of questions that need to be answered during a configuration audit. Students might profit from the experience of doing (or observing) a configuration audit using these questions following an assignment involving a required change to one of their projects.

9.7 Status Reporting

This section suggests that software engineers develop a need to know list for every SCI and keep them up to date. This is probably the easiest and most reliable way to deal with the configuration status reporting task.

9.8 SCM Standards

Several military and ANSI/IEEE software configuration standards are listed in this section. It may be worthwhile to have students do library or web research to locate the details for one of them.

PROBLEMS AND POINTS TO PONDER

9.1. Because change is a fact of life, it is necessary to recognize that iteration occurs in all paradigms for software engineering. The biggest danger is to assume that the software engineering process will flow sequentially from start to finish with no changes. This just is not realistic!

9.2. We have to establish a point at which we "cut the chord." That is, we must define a point beyond which we will not change something without careful (even formal) evaluation and approval.

9.3. A small project might combine analysis and design modeling into a "Development Specification" that would serve as the first SCI. The source would be the second. It is unlikely that a formal Test Specification would be created, but the suite of test cases would be the third

SCI. A User's manual would likely be the fourth SCI and executable versions of the software would be the fifth.

Any change to these baseline SCIs might generate the following questions:

1. What are the effort and cost required?

2. How complex is the change and what is the technological risk?

3. Is the software that is to be changed heavily coupled to other components of the system?

4. Are the modules to be changed cohesive?

5. What is likely to happen if the change is not implemented properly?

6. What is the relative importance of this change as compared to other requested changes?

7. Who will make the change?

8. How can we be sure that the change is implemented properly?

9.4. Any book on database design would provide pointers here.

9.8. Other relationships:

<mapped from>

<describes>

<derived from>

<uses>

<model of>

<created using>

<identified as>

9.12. An SCM audit focuses on compliance with software engineering standards while an FTR concentrates on uncovering errors associated with function, performance or constraints. The audit has a less technological feel.

Chapter 10

System Engineering

CHAPTER OVERVIEW AND COMMENTS

This chapter provides an introduction to the process of system engineering. The emphasis is on requirement analysis, specification, and quality management. Simple examples of system modeling are presented. More detailed discussion of system representation (e.g. data flow diagrams) appears later in the text.

10.1 Computer-Based Systems

This section introduces the systems view of engineering (all complex systems can be viewed as being composed of cooperating subsystems). The elements of computer-based systems are defined as software, hardware, people, database, documentation, and procedures. Software engineering students find it difficult to think about system elements other than software when they begin to develop their first projects. It is important to get students in the habit of taking time to understand the all the elements of the systems in which their software products will reside.

10.2 The System Engineering Hierarchy

The key to system engineering is a clear understanding of context. For software development this means creating a "world view" and progressively narrowing its focus until all technical detail is known. It is important to get students in the habit of recognizing the importance of developing alternative solutions to problems. In software engineering there is rarely one right way of doing something. Instead designers must consider the tradeoffs present in the feasible solutions and select one that seems advantageous for the current problem. This section lists several factors that need to be examined by software engineers when evaluating alternative solutions (assumptions, simplifications, limitations, constraints, and preferences). This section also cautions students that developing reactive systems without some simulation capability is a risky practice.

10.3 Business Process Engineering: An Overview

Software engineers usually do not participate in the upper levels of business process engineering (information strategy planning and business area analysis). Student projects usually begin with the assumption that this work has been done and is understood by the customer. Students need to believe that it is important for software engineers to understand the results of the system engineering that has taken place prior to beginning the definition of the requirements for a specific information system to support a particular business application.

10.4 Product Engineering: An Overview

Product engineering is described as the task of translating a customer's desires for a set of capabilities into a working product. Software engineers participate in all levels of the

product engineering process that begins with requirements engineering. The analysis step maps requirements into representations of data, function, and behavior. The design step maps the analysis model into data, architectural, interface, and software component designs. Detailed discussion of analysis and design techniques appears in the next several chapters of this text.

10.5 Requirements Engineering

This section describes the process of requirements engineering detail, from requirement elicitation through requirement management. 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. This section provides an opportunity to reinforce quality assurance and configuration management concepts discussed previously. Students should be encouraged to develop and use traceability tables on their own projects.

10.6 System Modeling

This section describes a system modeling technique that is based on developing a hierarchical system model that has a system context diagram (SCD) as its root. This model is based on the notion of an information transform using an input-processing-output template. The SDC provides a big picture of the system, even when all details are not known. The process of building a system flow diagram (SFD) is similar to building a data flow diagram (DFD). Data flow diagrams are not discussed in detail until Chapter 12, so it is probably better to hold off on a discussion of DFD's until later in the course.

Chapter 10 begins with Machiavelli's quote on the dangers associated with the introduction of a "new order". It might be worthwhile to project how computer-based systems will introduce a "new order" in the first decade of the 21st century. You might use Toeffler, Naisbitt, or Negroponte as guidelines for these discussions. Topics could

include: the Internet (personal privacy and e-commerce), intelligent agents, artificial reality, artificial intelligence, office automation, education, warfare, medicine, voice recognition, robots, vision systems, personal digital assistants, appliance computers, DVD/MP3, networks, and data warehousing issues.

PROBLEMS AND POINTS TO PONDER

10.1. Use a large dictionary to present the many definitions of the word. A thesaurus may help. Good luck!

10.2. You might suggest the following systems for the model requested:

An airport

Your university

An on-line bank

A retail store

An e-commerce site

10.3. Suggest that the student draw a hierarchy similar to those represented in Figures 10.1 through 10.3.

10.5. The data architecture refers to corporate data, its organization, and the relationships between individual corporate data elements. For example, a telephone billing system might draw on elements that include customer information, rate tables, calling logs and so forth. Application architecture defines the functional hierarchy that is required to realize the goals and objectives of a large information system.

10.6. A business objective for a software products vendor might be to "get the customer written information about our products within 4 hours of receipt of inquiry." The following goals would be stated:

- improve system that integrate telephone sales to customer response;
- make better use of the internet/e-mail and web
- make better use of automated instant response systems
- improve customer follow-up post receipt of inquiry information

10.7. There is less likelihood of miscommunication when the developer is the system engineer, but there is high likelihood of "creeping enhancements" and the possibility of building a system that doesn't meet customer's needs because perceptions of need are

incorrect. When the customer is the system engineer, the is the likelihood that technical issues may be omitted, but it is likely that customer need will be more accurately reflected. When an outside third party is the system engineering, communication bandwidth between developer and customer must be very high. It's likely that things will fall into the cracks. However, the third party may be an expert, hence better overall quality of the system model. The ideal system engineer is technically expert in system engineering, has full understanding of the customer business/product domain, and understands the development environment (and leaps tall buildings in a single bound!).

10.8. Additional questions:

- What has been done to accomplish this task previously?
- Couldn't sorting occur as the boxes were packed?
- Is the number of different parts (and boxes) likely to increase or decrease?
- Are all boxes the same size and weight?
- What is the environment in which this system will sit (e.g., corrosive, high temperature, dusty, etc.)?

Allocations: (4) Bar code reader and PC. PC prints destination on screen and a human operator does the placement into the bin; (5) Human reads box and speaks destination into voice recognition system that controls and automated shunt.

10.9. The checklist should answer one global question: "Can we do it?" An example checklist follows:

project complexity
    hardware
    software
    interfaces
relevant staff experience
projected degree of change
percent "new" technology
availability of resources
    people
    hardware
    software
support
performance requirements

In addition to the above items, other topics considered in the book should also be added.

A "feasibility number could be calculated by assigned a numeric grade (in the range 1 to 5, say) to each attribute and a weight (same range) to be used as a multiplier for the grade. The feasibility number is the sum of the products of grade x weight.

10.10. Suggest that your students obtain one or more books on business and/or project management. The following titles should be useful:

Degarmo, E.P, et al, Engineering Economy, 7th edition, MacMillan, 1984.

DeMarco, T., Controlling Software Projects, Yourdon Press, 1982.

Gilb, T., Principles of Software Engineering Management, Addison-Wesley, 1988.

Gunther, R.C., Management Methodology for Software Product Engineering, Wiley, 1978.

Harrison, F., Advanced Project Management, Wiley, 1981.

Londiex, B., Cost Estimation for Software Development, Addison-Wesley, 1987.

Ould, M.A., Strategies for Software Engineering, Wiley, 1990.

Page-Jones, M., Practical Project Management, Dorset House Publishing, New York, 1985.

10.11. Although the ACD is not used widely in industry, it is a worthwhile mechanism for getting student to understand the interaction between various components of a computer based system, If time permits, I would strongly recommend that you go through this notation in class and have your student try this problem.

10.14. You may, if you choose, allow students to derive their own System Specification for any of the systems shown without any handout. However, it is relatively easy to acquire a general article or paper that will serve as a consistent basis for student work. Alternatively, require that students research attributes of one of the systems, then develop the specification.

10.15. There are many cases where complete specification is impossible early in a project. In some of these, it is necessary to prototype before a formal specification can be developed. Examples are:

1. sophisticated analytical models that must be researched and developed—delay until requirements specification or even design!

2. sometimes, specific hardware/software interfaces—delay until requirements, but no later!

3. certain performance constraints—delay until requirements, but no later!

10.16. In many instances, a computer-based system is actually a "software only" system. That is, an existing standard computer, operating system and I/O

environment are to be used. In such cases, the System Specification can be eliminated in lieu of a Software Requirements Specification.

There are also situations in which a system prototype (either real or paper) is much more effective than a specification.

Chapter 11

Analysis Concepts and Principles

CHAPTER OVERVIEW AND COMMENTS

This chapter describes the process of software requirements analysis as a refinement of the initial work products developed during the systems engineering process. It is important that students be encouraged to complete a system specification document, before beginning to write a software specification document for their own course projects. The focus of software requirements analysis is on working with the customer to identify and refine the information, functional, and behavioral requirements for the proposed software solution to the customer's problems.

11.1 Requirements Analysis

Software requirements analysis is intended to bridge the gaps between system requirements engineering and software design. In this chapter the focus of software requirements analysis is on the process of modeling the data, function, and performance of the software to be developed. The process model used has five areas of effort: problem recognition; evaluation and synthesis; modeling; specification; and review. It is important to remind students that requirements analysis is an important part of the software configuration management plan and the software quality assurance plan.

11.2 Requirements Elicitation for Software

This section discusses the task of software requirements elicitation. Two elicitation techniques are discussed (the use of context free questions in structured customer interviews and facilitated application specification techniques). 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. Quality function deployment is described as a quality management technique that helps software engineers understand the relative importance of solution elements to the customer. Having students develop customer voice tables for their own projects and reviewing them with real or simulated customers would be a good lab activity. Encouraging students to develop use-cases as part of their requirements analysis will help prepare them for object-oriented analysis later in the text.

11.3 Analysis Principles

This section presents guidelines for conducting software requirements analysis. The perspective presented is that the goal of software requirements analysis is to build working models of the information, functional characteristics, and behavioral performance of a software product. Specific model representations are discussed in Chapters 12, 20, and 21. It may be wise to focus on the general characteristics of models when lecturing on this chapter. Presenting examples of horizontal and vertical partitioning of data, function, and/or behavior might be good to do. It is important to convince students that going straight to the implementation view of software requirements, without considering the essential view, should not be done in the real world. Going directly to the implementation view is similar to deciding on a technology and then trying to define the customer's needs based on the capabilities of the technology (a strategy that is rarely satisfactory from the customer's perspective).

11.4 Software Prototyping

The pros and cons of using software prototyping during the analysis phase of the software process should be discussed. Students should not to be allowed to believe that prototypes are used to avoid writing the formal analysis and design documents. Prototypes are particularly good for working out the appearance details and behavioral characteristics of a software user interface. However, as will be discussed in

Chapter 15, techniques exist for developing models of computer-user interaction without building throwaway prototypes. Evolutionary prototyping is usually a more cost-effective choice, if the user has a good understanding of his or her own goals and needs.

11.5 Specification

Students need to understand that is not generally possible to completely separate software requirements specification from design. Even the waterfall model contains some provisions for revisiting the specification phase if defects are uncovered during the design phase. Students should be encouraged to use the software requirements specification template on the text web site as they work on the their own course projects. Model representations and specification notations are discussed in more detail in the next chapter. It is important for students to remember that the section on validation criteria is an important part of the specification document. Part of the reason for this, is that the validation criteria along with the software models (data, functional, and behavioral) define a contract between the developer and the client. When this contract is satisfied the product is finished and the developer can be paid.

11.6 Specification Review

The purpose of the specification review is to make sure that the customer and developer agree on details of the software requirements (or prototype) before beginning the major design work. This implies that both the customer and developer need to be present during the review meeting. Students will benefit from having to present their requirements specifications to a customer or to their classmates.

PROBLEMS AND POINTS TO PONDER

11.1. Refer to the last sentence in the introduction to this chapter.

- Customer doesn't completely understand what is desired
- final result is hazy
- analyst cannot understand customer's world
- personality conflict between analyst and customer

- too many customers

11.2. One or more constituencies who will be using the new software feel "threatened" by it in some way, e.g., it may threaten jobs; provide greater management control; attack "tradition." One way to reduce (but rarely eliminate) problems is to introduce the rationale for a new system early, and educate users so that the threat of the unknown is reduced.

11.3. Although "ideal" analysts are a rare breed, the following attributes would be nice:

- conversant in both customer and developer world
- excellent communication skills
- good technical computing background

11.4. The customer for information systems is often another department in the same company—often referred to as the "user department." The customer for computer based products is typically the marketing department! The customer for computer-based systems may be an internal department, marketing or an outside entity (e.g., a government agency, another company, an end-user).

11.7. The *Preliminary User Manual* is a form of paper prototype for the software that is to be built. It provides the reader with a characterization of the software taken from the user's point of view.

11.8. The information domain for *SafeHome* should be organized hierarchically:

- user input information
- entry code
- bounding levels
- phone numbers
- delay times
- sensor information
- break-in

... etc.

You might introduce a DFD (see Chapter 7) like notation for representing flow, or delay consideration of this until later.

11.9. Partitioning will depend on the requirements established by the students. Be sure that each level of the function hierarchy represents the same level of detail. For example, a function such as read sensor input should not be at the same level (in the hierarchy) as dial telephone number. The latter is a reasonably low level function, while the former encompasses many important activities. Teach your students to think hierarchically!

11.10 - 11.13. The remainder of these problems might best be left for a term project.

11.14. It is extremely useful to compare the requirements derived by different students. The lesson learned is that even when the problem statement is identical, there is much room for interpretation. Hence, a critical need for review with your customer.

Chapter 12
Analysis Modeling

CHAPTER OVERVIEW AND COMMENTS

This chapter describes the process of analysis modeling. Structured analysis modeling is discussed in this chapter and object-oriented modeling techniques are described in Chapter 21. The primary representation techniques discussed in Chapter 12 are entity relationship diagrams to represent the information models, data flow diagrams to represent functional models, and state transition diagrams to represent behavioral models. Several alternative analysis techniques (DSSD, JSD, SADT) are presented briefly at the SEPA web site.

12.1 A Brief History

This section contains a brief history of structured analysis. The use of graphical notation to represent data and data transformation processes is an important part of many of the widely known structured analysis methodologies.

12.2 The Elements of the Analysis Model

In this chapter, the generic analysis model consists of an entity-relationship diagram (data model), a data flow diagram (functional model), and a state transition diagram (behavioral model). Descriptions of how to build each appear later in this chapter. The data dictionary is described as a repository that contains descriptions of all data objects that consumed or produced by the software product. Students should be encouraged to construct each of these as they complete the analysis portions of their own course projects.

12.3 Data Modeling

This section describes the use of entity relationship diagrams (ERD) as one means of representing the objects and their relationships in the data model for a software product. It is important for students to recognize that any entity (person, device, or software product) that produces or consumes data

needs to appear in the data model. Students who are not familiar with object-oriented programming or database definition may need to see some examples of how object hierarchies are defined and revised. Determining the details of how objects are related to one another is a skill that improves with experience. Students will need to construct at least one ERD on their own before they begin to feel comfortable with data modeling. The details of constructing ERD's are described in Section 12.6.

12.4 Functional Modeling and Information Flow

This section describes the use of data flow data diagrams as one means of representing the functional model of a software product. Students may have difficulty understanding that DFD's do not represent program logic like flowcharts do. The construction of control flow diagrams from DFD's is described as one method of modeling real-time systems. Students will need to go through the process of constructing (and refining) at least one DFD on their own to begin to feel comfortable with functional modeling. The details of constructing DFD's are described in Section 12.6.

12.5 Behavioral Modeling

This section describes the use of state transition diagrams (STD) as one means of representing a software behavioral model. Students may have seen STD's 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 STD's. Students will need to construct at least one STD on their own before they begin to feel comfortable with behavioral modeling. The details of constructing STD's are described in Section 12.6.

12.6 The Mechanics of Structured Analysis

Several models (ERD, DFD, and STD) are developed for the home security system. Guidelines for constructing each type of model are described in detail. The use of a PAT (process activation table or state transition table) as a alternative to using an STD is discussed. It might be good to show students a sample PSPEC written using a program design language (PDL), since a natural language (English) is used in the example.

12.7 The Data Dictionary

The details of data dictionary entries are described in this section. Showing students an example of a complete data dictionary for a small professionally developed project might be a good idea. Showing students a data dictionary produced using a CASE tool might also be a good thing to do. Having students build data dictionaries for their own projects should be a course requirement.

12.8 An Overview of Other Classical Analysis Methods

This section provides a very brief discussion of three classic structured analysis methodologies (Data Structured Systems Development - DSSD, Jackson System Development - JSD, Structured Analysis and Design Technique - SADT) and points the reader to the SEPA web site for further detail. It is generally not possible to cover these alternative methods in the time allotted to most software engineering courses and still cover Structured Analysis and/or OO analysis.

PROBLEMS AND POINTS TO PONDER

12.2. Be certain to emphasize the fact that all data-objects and relationships MUST be customer visible. Attributes should be reviewed to be certain that they properly reflect the requirements of the system.

12.3. Cardinality defines the range of object-to-object relationships. Modality indicates whether or not a relationship between two objects in mandatory.

12.4. Be sure that models produced indicate major input and output information and any databases that is required (Note: many people believe that the context level DFD should not represent data stores; I don't feel that this restriction is necessary.) At the undergraduate level, you might suggest systems like: course registration; dormitory assignments; the library; the work study or co-op system; financial aid, etc.

12.5. If students have background in compilers or operating systems, suggest one of these as a DFD exercise. Otherwise, propose any one of the following as a possible alternative to the systems noted on problem 12.2:

- an income tax package
- a suite of statistics functions
- a popular video game

- a flight controller for an aircraft

- any system relevant to your students

Be sure to emphasize the importance of a hierarchical approach to function partitioning when functions are defined.

12.7. Not necessarily. For example, a composite data item can be shown as input to a level 0 transform. The composite item (data object) is refined in the data dictionary into three elementary data items that are each shown separately at a refinement indicated at level 1. Even though the names of the flow items have changes and their number has changed, flow continuity is maintained through the dictionary.

12.8. Ward and Mellor use a "control transform" in the place of the CSPEC and provide detailed of the control transform in the processing narrative that accompanies the transform. The use of the STD is the same, although it is not "attached" to a CSPEC.

12.9. A control process is accommodated with Hatley-Pirbhai using the CSPEC and (possibly) a PAT.

12.10. Event flow indicates that something has occurred. In most cases the occurrence can be represented either by a true/false condition or by a limited number of discrete values.

12.13. You might suggest that your students approach the problem using the grammatical parse described in this chapter. The following data objects should be identified for PHTRS and reorganized into 1NF, 2NF, 3NF:

pot hole data

identifying number

street address

size

location in street

district *

repair priority *

work order data:

pot hole location

size

repair crew ID number

number of people in crew

equipment assigned

hours applied

hole status

amount of filler used

cost of repair *

damage data

citizen's name

citizen's address

citizens phone number

type of damage

dollar claim

Note that those objects followed by a * can be derived from other objects. Be sure that students properly define the various normal forms.

This problem is amenable to data flow and object-oriented (Chapter 20) analysis approaches. If time permits (it usually doesn't unless you're conducting a two term course) it is worthwhile to have students solve the problem using both analysis methods.

Chapter 13

Design Concepts and Principles

CHAPTER OVERVIEW AND COMMENTS

This chapter describes fundamental design concepts and principles that are essential to the study of an understanding of any software design method. Basic concepts are introduced and a fundamental design model is discussed. The design model consists of the data design, architectural design, interface design, and component-level design. Although many of the design concepts and principles discussed in this chapter may have been presented in earlier courses, it is important to re-emphasize each concept/principle so that all students have a consistent understanding of them. Students should be encouraged to use the design document template form the SEPA web site as a basis for the design documents they write for their own software projects.

13.1 Software Design and Software Engineering

It is important for students to understand the mapping from analysis model to the design model. The data design is derived from the data dictionary and the ERD. The architectural design is derived from the DFD. The interface design comes from the DFD and CFD. The component level design is derived from the PSPEC, CSPEC, and the STD. Students will need to be reminded that all design work products must be traceable to the software requirements document and that all design work products must be reviewed for quality.

13.2 The Design Process

This section makes 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.

13.3 Design Principles

This section presents several fundamental design principles. Students need to be aware of the rational behind each principle. The principles for conducting effective design reviews were presented in Chapter 8.

13.4 Design Concepts

This section discusses many significant design concepts (abstraction, refinement, modularity, software architecture, control hierarchy, structural partitioning, data structure, software procedure, information hiding). For some of these concepts a simple definition will be sufficient. For others (e.g. control hierarchy) presenting additional examples may be helpful. The issues associated with modularity are discussed in Sections 13.5 and 13.6.

13.5 Effective Modular Design

Several types of coupling and cohesion are described in this section. Students may have difficulty distinguishing the various types of coupling and cohesion without seeing examples of each. Students will have difficulty in remembering that high cohesion is good and that high coupling is bad. It may help to remind them that functional independence is the goal for each module. This is more likely when modules have single purposes (high cohesion) and rely on their own resources for data and control information (low coupling).

13.6 Design Heuristics for Effective Modularity

This section lists several fairly abstract heuristics for creating good program structure through careful module design. Discussing examples of programs that make use of these heuristics and some that violate these heuristics may be a good thing to do.

13.7 The Design Model

Students should be told that the details of creating a complete design model appear later in the text (Chapters 14, 15, 16, and 22). Students should be reminded that design changes are inevitable and that delaying component level design can reduce the impact of these changes.

13.8 Design Documentation

This section lists the major sections of a software design specification document. 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.

PROBLEMS AND POINTS TO PONDER

13.1. Yes, but the design is conducted implicitly — often in a haphazard manner. During design we develop representations of programs—not the programs themselves.

13.2. (1) The design should be characterized in a way that make it easy for other to understand. (2) The design should be independent of programming language but should consider the limitation imposed by a programming language.(3) The design emphasize the structure of data as much as it emphasizes the structure of the program.

13.3. Credit-card> validate; check-limit; charge-against

Engineering-drawing> scale; draw; revise

web-site> add-page; access; add-graphic

13.4. We create a functional hierarchy and as a result refine the problem. For example, considering the check writer, we might write:

```
Refinement 1:
   write dollar amount in words
Refinement 2:
   procedure write_amount;
      validate amount is within bounds;
      parse to determine each dollar unit;
      generate alpha representation;
   end write_amount
Refinement 3:
   procedure write_amount;
      do while checks remain to be printed
         if dollar amount > upper amount bound
             then print "amount too large error
               message;
               else set process flag true;
         endif;
         determine maximum significant digit;
         do while (process flag true and
          significant digits remain)
            set for corresponded alpha phrase;
            divide to determine whole number
            value;
```

```
                concatenate partial alpha string;
                reduce significant digit count by
                one;
            enddo
            print  alpha string;
        enddo
    end write_amount
```

13.5. There are cases in which different parts of a problem are interrelated in a manner that makes separate considerations more complex than combined considerations. Highly coupled problems exhibit this characteristic. However, continuing combination of problem parts cannot go on indefinitely because the amount of information exceeds one's ability to understand. Therefore, when (10.2) is not true, modularity may be modified, but not eliminated.

13.6. In some time critical applications, monolithic implement-ation may be required. However, design can and should occur as if the software was to be implemented modularly. Then "modules" are coded in-line.

13.7. The robot controller:

Abstraction I:

The software will incorporate control functions that will allow manipulation of a robot arm using instructions obtained form disk storage. The instructions will control x, y, z motion, pitch and roll motion, gripper (the robot hand) action and force sensing by the arm.

Abstraction II:

```
    Robot Software Tasks:
      Instruction input interface;
      Instruction processor;
      Command output interface;
      Feedback controller;
end.
```

Abstraction III:

```
    sub-task: instruction processor
      obtain an instruction;
      determine instruction type;
         assess potential conflicts;
         process instruction;
      end.
```

Abstraction IV:

```
     procedure: process instruction;
         case of <instruction-type>;
            when <instruction-type is motion> select
               motion-calculations;
            when <instruction-type is gripper> select
               gripper-control;
            when <instruction-type is sensing> select
               read sensing ports;
         end case
         test error-flags prior to instruction
execution;
         execute instruction;
       end
Abstraction V:
     procedure: motion calculations;
         case of <motion-type>;
            when <motion-type is linear> select
            begin: compute linear motion from (x,y,z)
                     start to (x,y,z);
               apply accel/decel information to path ;
               test violations of arm travel bounds;
               set  flags as required;
               end
              when  ...  end
```

13.8. Parnas' seminal paper has been reproduced in a number of anthologies of important software engineering papers (e.g., Yourdon's Classics in Software Engineering or the IEEE tutorial by Freeman and Wasserman, Software Design Techniques. Parnas uses a KWIC index system as an example. To quote from his description:

> The KWIC index system accepts an ordered set of lines, each line is an ordered set of words and each word is an ordered set of characters. A line may be "circularly shifted" by repeatedly removing the first word and appending it to the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

13.9. Information hiding can be related to both coupling and cohesion concepts. By limiting the availability of information to only those modules that have absolute need, coupling between modules is inherently reduced. In general, isolation of information predicates isolation of function; therefore, cohesion of individual modules can also be improved.

13.11. Common coupling is increased when a programming language supports internal procedures and as a consequence, information hiding is difficult to achieve (although the spirit of the concept can be maintained).

13.12. External world, compiler, and operating system coupling will affect software portability adversely. As an example, consider a program that has been designed to make use of special graphics features of an intelligent terminal. If the software is moved to a system without the terminal, major design and code modifications may be required.

13.13. Structural partitioning separates functionality with the program architecture. This means that changes will have less likelihood of propagating side effects, if other forms of coupling have been damped.

13.14. A factored program architecture is more maintainable. Changes in "worker" modules are most common and since these reside at the bottom of the architecture, there is less likelihood of defect propagation when changes are made.

13.15. Access to a data or procedural abstraction occurs on through a controlled interface. The details of the abstraction are hidden.

13.16. The effects of decisions should not exceed the direct structural coupling of the modules involved in those decisions. If it does, there is (again) more likelihood of side effects.

Chapter 14

Software Design

CHAPTER OVERVIEW AND COMMENTS

The objective 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. Architectural modeling of object-oriented systems appears later in the text. Students should be reminded that quality reviews need to be conducted for the architectural model work products.

14.1 Software Architecture

This section defines the term software architecture 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. Students should be encouraged to build architectural models of their own software development projects.

14.2 Data Design

This section describes data design as the process of creating a model of the information represented at a high level of abstraction (using the customer's view of data). Students should be reminded that careful attention to data structure design is at least as important as the effort devoted to algorithm design. Students should make an effort to learn definitions for the terms data modeling, data structure, database, and data warehouse. Students should be encouraged to use the principles of component level data design presented in this section.

14.3 Architectural Styles

A taxonomy of architectural styles (design patterns) is described in this section. 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 emphasizes. 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.

14.4 Analyzing Alternative Architectural Designs

This section 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.

14.5 Mapping Requirements into Software Architecture

This section describes the general process of mapping requirements into software architectures during the structured design process. The technique described in this chapter is based

on analysis of the system data flow diagram. The key point to get across to students is the difference between transform flow and transaction flow. Transform flow is largely sequential and has very little divergence along the action path. Transaction flow can follow many action paths and the processing of a single data item often triggers path selection.

14.6 Transform Mapping

The process of mapping a data flow diagram with transform characteristics into a specific architectural style is illustrated in this section. Students should be reminded to strive for DFD bubbles that exhibit high cohesion. Students should be told that both transform and transaction flows may occur in the same DFD. Each subflow should be classified and mapped using the appropriate technique (transform or transaction). Students may need to see more than one example of transform mapping before they attempt to do it on their own.

14.7 Transaction Mapping

The process of performing transaction mapping is presented as being similar to transform mapping, but focusing on transaction (not transform) centers. In transaction mapping, the first level factoring results in the derivation of the control hierarchy. The second level factoring distributes the low-level modules among the appropriate controller.

14.8 Refining the Architectural Design

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 web site to see where they need to go next with their design work.

PROBLEMS AND POINTS TO PONDER

14.1 The concepts of styles and patterns occur for buildings and software at both macroscopic and microscopic levels. For example, overall styles (center-hall colonial, A-frame) can be found in

a house. These represent macroscopic styles. Lower-level microscopic patterns (for a house) can be found in categories of wood molding, fireplace designs, windows, etc.

14.3 A data warehouse is a global repository for information gathered from many corporate sources.

14.5 Data centered architecture: airline reservation system; library catalog system; hotel booking system
Data flow architecture: any engineering/scientific application where computation is the major function
Call and return architecture: any I-P-O application
Object-oriented architectures: GUI-based applications; any OO application
Layered architecture: any application in which the application functions must be decoupled from the underlying OS or network detail. Client server software is often layered.

14.11 When transform mapping is used where transaction mapping should be used, two problems arise: (1) the program structure tends to become top heavy with control modules—these should be eliminated; (2) significant data coupling occurs because information associated with one of many "processing actions" must be passed to an outgoing path.

Classic examples include:

- many engineering analysis algorithms
- many process control applications
- some commercial applications

Look for a well-developed DFD and correct mapping to program structure.

Chapter 15

User Interface Design

CHAPTER OVERVIEW AND COMMENTS

User interface design is a topic that receives a lot of attention these days. 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 after thought). Having students formally evaluate each other's interfaces using the usability checklist from the SEPA web site is a very worthwhile course activity.

15.1 The Golden Rules

This section discusses 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.

15.2 User Interface Design

This section presents the notion that user interface design is really the process of reconciling four models (the design model, the user model, the user's perception of the system, and the system image). 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 well enough, that the system image 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.

15.3 Task Analysis and Modeling

Task modeling is presented as an extension of software analysis techniques described in earlier chapters. Two general approaches are suggested, observing the user complete his or her tasks in the work place or analyzing an existing software specification and determining the tasks required of the user. The object-oriented analysis techniques discussed later in this text can also be helpful during user task modeling. The primary reasons for conducting this type of analysis are to understand the user's task goals and to derive typical use case scenarios to guide the interface design process.

15.4 Interface Design Activities

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 scenario. Students will find it very instructive to build paper prototypes for task scenarios using the operators from familiar user interfaces. These paper prototypes are tested quickly and inexpensively.

15.5 Implementation Tools

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, MS Frontpage, or MS Word) 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.

15.6 Design Evaluation

Two interface design evaluation techniques are mentioned in this section, usability questionnaires and usability testing. Having the students use a usability questionnaire similar to the one on the SEPA web site to evaluate software user interfaces is fairly easy to do. The process of learning how to design good user interfaces often begins with learning to identify the weaknesses in existing products. Usability testing involves more statistics and research design knowledge than is typically found in undergraduate computing students. It is important to remind students, that beginning to evaluate user interface prototypes early in the design phase reduces the risk of building an interface not acceptable to the customer.

PROBLEMS AND POINTS TO PONDER

15.1 This should be not difficult! For those of us old enough to remember, some of the early interactive systems had abominable interfaces. In the modern context, have your studnets focus on Web-based application interfaces … many sacrifice ease-of-use for flash.

15.2 Catch potential interaction errors before they do "undoable" damage.
Allow the user to customize screen layout as well as commands.
Make use of breakaway menus so that common functions are always readily available.

15.3 If the user desires, display shortcut command sequences on the screen at all times.
Provide the facilities for "hints" when a WebApp requires password input

15.4 Use color consistently, e.g., red for warning messages, blue for informational content.
Provide keyword driven on-line help.

15.5 If you have decided to assign "tiny tools" for class projects, this might be a good place to begin. Be certain to examine some of the timny tools at the SEPA Web site to get a feel for user interface style.

15.6 If your students get bogged down in task analysis, the old I-P-O stand-by will work. Ask: What does the user input? How is it processed and how is the processing evidenced via the interface? What is produced as output?

15.11 When response time is unpredictable, the user becomes impatient and re-attempts the command requested or tries another command. In some cases, this can create queuing problems (for commands) and in extreme cases, cause loss of data or even a system failure.

Chapter 16

Component-Level Design

CHAPTER OVERVIEW AND COMMENTS

This chapter discusses the portion of the software development process where the design is elaborated and the individual data elements and operations are designed in detail. Much of the chapter discussion is focused on notations that can be used to represent low level procedural designs. Students should be encouraged to experiment with different design notations and try to find one that suits their programming style and aesthetic sense. Object-oriented methods for component level design appear later in the text.

16.1 Structured Programming

This section reviews the elements of the structured programming theorem. Two graphical techniques for representing algorithms (flowcharts and box diagrams) appear in the figures. Students may need to see examples of representing algorithms using flowcharts and box diagrams, if they are not familiar with them from earlier coursework. 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 languages (PDL) are discussed in detail 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++, Visual Basic, Cobol, 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.

16.2 Comparison of Design Notation

This section presents criteria for assessing the usefulness of particular design notations. If your course is more than one semester long, students may benefit from writing comparative evaluations of two different design notations or programming design languages. If your students have access to CASE tools, evaluating the design notations supported can be a worthwhile activity. Class discussion assessing the design notations described in this chapter may also be worthwhile.

PROBLEMS AND POINTS TO PONDER

16.1 It's not a bad idea to suggest that student "box-out" the structured constructs in an attempt to see if box boundaries cross (a violation of the structured programming philososphy.

16.2 Use examples from Java, C++ or any other localy used language.

16.3 "Chunking" is important because it leads to better pattern recognition and improved semantic information processing on the part of the reader.

16.4 - 16.11. Because the majority of your students wil have had multiple courses in "programming" it's likely that these problems will be unnecessary. It is important, however, to insist that students understand and apply the structured programming concepts.

Chapter 17

Software Testing Techniques

CHAPTER OVERVIEW AND COMMENTS

This chapter provides an overview of the software testing process. The focus of the chapter is on the differences between black-box and white-box testing. Several types of black-box and white-box testing techniques are described in detail. Several strategies for developing test cases are discussed in this chapter. The process of developing detailed test plans is described in the next chapter. Object-oriented testing is described later in the text. Students should be encouraged to try building test cases for their course projects, using several of the testing techniques presented here.

17.1 Software Testing Fundamentals

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. Formal reviews by themselves cannot locate all software defects. 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.

17.2 Test Case Design

This section discusses the differences between black-box and white-box testing. Another purpose of this section is to convince 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. The next text sections 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.

17.3 White-Box Testing

This section 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.

17.4 Basis Path Testing

This section describes basis path testing 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). 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.

17.5 Control Structure Testing

Basis path testing is one form of control structure testing. This section 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 17.5.3. 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.

17.6 Black-Box Testing

The purpose of black-box testing is to devise a set of data inputs that fully exercise all functional requirements for a program. Students will need to 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). Students will need to be reminded 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.

17.7 Testing for Specialized Environments, Architectures, and Applications

This section briefly discusses several specialized testing situations (GUI's, client/server architectures, documentation and help facilities, real-time systems). More extensive discussion of these testing situations appears elsewhere in the text or in the SEPA web site resource links. Object-oriented testing is discussed in Chapter 23. Discussion of test plans appears in Chapter 18.

PROBLEMS AND POINTS TO PONDER

17.1. See Myers [MYE79] for an extremely detailed "solution" to this problem.

17.2. You may elect to distribute the program source code to your students (embedding a few errors purposely).

17.3. In addition to objectives noted in Section 17.1:

a) a successful test demonstrates compliance with function and performance;

b) a successful test uncovers documentation errors;

5 a successful test uncovers interfacing problems;

6 a successful test demonstrates an understanding of program architecture, data structure, interface design and procedural design;

7 a successful test establishes an entry into a test case database that may later be used for regression testing.

17.4. As an alternative, you might suggest that your students apply basis path to one or more of the modules that they have created for their term project.

17.5 through 17.6. With some extensions, these problems could be assigned as a term project.

17.11 For specific input, an error occurs internally resulting in:

1) improper data placed in a global data area;

2) improper flags that will be tested in a subsequent series of tests;

3) improper hardware control that can only be uncovered during system test; yet "correct" output is produced.

17.12 No, even an exhaustive test (if it were possible) may be unable to uncover performance problems and errors in the specification of the software.

17.14 In this case both input and output "equivalence classes" are considered. For each class, the student should identify boundaries based on numeric ranges, elements of a set, system commands, etc.

17.15 This can be a paper exercise in which test cases for a GUI for some well know application are derived.

17.18 It might be a good idea to generate a set of use-cases to help "test" user documentation.

Chapter 18

Software Testing Strategies

CHAPTER OVERVIEW AND COMMENTS

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—I believe it is a useful metaphor for the software engineering process and the relationship of testing steps to earlier definition and development activities. The key concept for students to grasp is that testing must planned and assessed for quality like any other software engineering process. Students should be required to use the Test Specification template from the SEPA web site as part of their term project.

18.1 A Strategic Approach to Software Testing

This section 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 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 the early phases of software development. The role of software testing groups in egoless software development is another important point to stress with students. This section also discusses the issue of how to determine when testing is completed. Which is an important issue 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.

18.2 Strategic Issues

Several testing issues are introduced in this section. 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.

18.3 Unit Testing

This section discusses unit testing in the abstract. Both black-box and white-box testing techniques have roles in testing individual software modules. It is important to emphasize that white-box techniques introduced in Chapter 17 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.

18.4 Integration Testing

This section focused 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.

18.5 Validation Testing

In this section validation testing is described 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.

18.6 System Testing

System testing is described as involving people outside the software engineering group (since hardware and systems programming issues are often involved). Several systems tests are mentioned in this section (recovery, security, stress, and performance). Students should be familiar with each of them. Very little literature is available on this subject—therefore, it is difficult to suggest any worthwhile supplementary materials. However, a thorough discussion of the problems associated with "finger pointing," possibly with excerpts from Tracy Kidder's outstanding book, *The Soul of a New Machine,* will provide your students with important insight.

18.7 The Art of Debugging

This section 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. The art of debugging is presented in Section 18.7. 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.

PROBLEMS AND POINTS TO PONDER

18.1 Verification focuses on the correctness of a program by attempting to find errors in function or performance. Validation focuses on "conformance to requirements—a fundamental characteristic of quality.

18.2 The most common problem in the creation of an ITG is getting and keeping good people. In addition, hostility between the ITG and the software engineering group can arise if the interaction between groups is not properly managed. Finally, the ITG may get involved in a project too late—when time is short and a thorough job of test planning and execution cannot be accomplished.

An ITG and an SQA group are not necessarily the same. The ITG focuses solely on testing, while the SQA group considers all aspects of quality assurance (see Chapter 17).

18.3 It is not always possible to conduct thorough unit testing in that the complexity of a test environment to accomplish unit testing (i.e., complicated drivers and stubs) may not justify the benefit. Integration testing is complicated by the scheduled availability of unit tested modules (especially when such modules fall behind

schedule). In many cases (especially for embedded systems) validation testing for software cannot be adequately conducted outside of the target hardware configuration. Therefore, validation and system testing are combined.

18.4 If only three test case design methods could be selected during unit test they would be:

1. basis path testing—it's critically important to ensure that all statements in the program have been exercised.

2. equivalence partitioning—an effective black-box test at the module level.

3. boundary value analysis—"bugs lurk in corners and congregate at boundaries."

Naturally, good arguments can be made for other combinations of test case design techniques.

18.5 A highly coupled module interacts with other modules, data and other system elements. Therefore is function is often dependent of the operation of those coupled elements. In order to thoroughly unit test such a module, the function of the coupled elements must be simulated in some manner. This can be difficult and time consuming.

18.6 A single rule covers a multitude of situations: All data moving across software interfaces (both external and internal) should be validated (if possible).

Advantages: Errors don't "snowball."

Disadvantages: Does require extra processing time and memory (usually a small price to pay).

18.8 The availability of completed modules can affect the order and strategy for integration. Project status must be known so that integration planning can be accomplished successfully.

18.9 No. If a module has 3 or 4 subordinates that supply data essential to a meaningful evaluation of the module, it may not be possible to conduct a unit test without "clustering" all of the modules as a unit.

18.10 Developer, if customer acceptance test is planned. Both developer and customer (user) if no further tests are contemplated. An independent test group is probably the best alternative here, but it isn't one of the choices.

Chapter 19

Technical Metrics for Software

CHAPTER OVERVIEW AND COMMENTS

This chapter discusses the use of software metrics as a means of helping to assess the quality of software engineering work products. In Chapter 8, metrics were discussed in the context of assessing software process and project management. Many students may find this material difficult, if they have not studied regression in a previous course. 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 metric in improving the quality of software engineering work products.

19.1 Software Quality

This section defines software quality as conformance to the implicit and explicit requirements associated with a product. This implies the existence of a set of standards used by the developer and customer expectations that a product will work well. Conformance to implicit requirements (e.g. ease of use and reliable performance) is what sets software engineering apart from simply writing programs that work most of the time. Several sets of software quality factors are described. Be sure to emphasize the McCall triangle (Figure 19.1) during lecture. Although McCall's work was done almost 30 years ago, quality factors remain the same today. What does that say about quality issues for computer software? They remain invariant even though software technology has changed radically.

Students need to be reminded that these factors are used in the qualitative assessment of software quality. The main weakness of this approach, is that it is based on opinion obtained by having customers or developers score products using a questionnaire. Disagreements on qualitative ratings are hard to resolve. In the remainder of this chapter focuses on quantitative approaches to assessing software quality.

19.2 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. Similarly, metrics computed from the measure need to 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.

19.3 Metrics for the Analysis Model

This section discusses metrics that may be useful in assessing the quality of the requirements analysis 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.

19.4 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 begin computing 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 measure is a 'must.' Be sure to emphasize that a software engineer should use these metrics as he/she is doing design work.

19.5 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.

19.6 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.

19.7 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.

PROBLEMS AND POINTS TO PONDER

19.1. As an alternative, you might assign readings from one or more recent papers on the subject.

19.2. Quality metrics are time invariant. If you build software that meets quality factors today, it will almost certainly continue to exhibit quality in years to come.

19.3. That's like asking for a single metric to describe a human being. Should it be weight, height, eye color, IQ, EQ, foot size, ...?

Software systems are just too complex and have too many attributes, all of which are meaningful. Therefore, the search for a single number is fruitless.

19.4. The key here is to development meaningful counts for inputs, outputs, etc. For guidance, see Chapter 4 or one of the FP books noted in the *Further Readings and Other Information Sources* section of that chapter.

19.5 through 19.7. Rather than assigning these problems, you might require that the members of each project team submit a metrics report for the analysis and design models they have created for their course project

19.8. To computer DSQI, we use equation 19.5 and related definitions:

S1 = 1140

S2 = 1140 - 96 = 1044

S3 = 490

S4 = 220 + 220*3 = 880

S5 = 140

S6 = 90

S7 = 600

we assume D1 = 1

D2 = 1 - (1044/1140) = 0.08

D3 = 1 - (490/1140) = 0.57

D4 = 1 - (140/880) = 0.84

D5 = 1 - (90/880) = 0.90

D6 = 1 - (600/1140) = 0.47

Assuming wi = 0.167,

DSQI = 0.167 SUM (Di)

= 0.644

This value has no meaning in a vacuum. It must be compared to values for other applications. If lower, design may be suspect.

19.16. The software maturity index is computed using equation 19.15:

Mt = 940

Fc = 90

Fa = 40

Fd = 12

SMI = [940 - (40 + 90 + 12)] / 940

= 0.85

This value indicated that the system has a way to go before it fully stabilizes (SMI -> 1.0).

Chapter 20

Object-Oriented Concepts and Principles

CHAPTER OVERVIEW AND COMMENTS

This chapter introduces students to the object-oriented point of view with regard to software design. The next three chapters cover object-oriented analysis, object-oriented design, and object-oriented testing. The details of object-oriented programming are not covered in this text. Students who have previously defined classes in languages like C++ or Java will have an advantage over those who have not. One of the big ideas for students to come away with from this chapter is how to recognize potential objects in a problem environment. Another important point is that object-oriented software engineering has many of the same basic processes found in conventional software engineering, they may require slightly different execution.

Be certain that all key OO terms are introduced and under-stood by your students. They should feel comfortable defining class, object, attribute, operations, message, etc. They should also be aware of the criteria through which object selections are made. I would suggest giving a pop quiz during the class session immediately after this lecture. Ask for all definitions to drive their importance home.

20.1 The Object-Oriented Paradigm

This section introduces students to an evolutionary object-oriented process model resembling Boehm's spiral model (Chapter 2). It is hard to define all necessary classes in a single iteration of the OO process. Students need to understand that simply implementing their software using object-oriented programming (OOP) does not take full advantage of the benefits to be gained by using the object-oriented process model.

20.2 Object-Oriented Concepts

A number of terms associated with the object-oriented are defined in this section. There are three key definitions students need to learn (class, instance, and inheritance). Students should understand the differences between the terms class (data type) and instance (variable) with regard to objects. Students should

be encouraged to think of objects as abstract data types with private data members (attributes) and public operators (methods). Messages are similar to function calls in imperative programming languages. If your students have not programmed in a language that supports objects, you may need to spend some time explaining why encapsulation, inheritance, and polymorphism are supported better by objects than without them. Presenting some examples showing code reuse via inheritance may be helpful. Students will struggle with the concept of polymorphism, if they have not seen it prior to this course.

20.3 Identifying the Elements of the Object Model

This section describes the process of identifying candidate objects, as well as their attributes and operations. The grammatical parse is a useful technique for identifying potential objects from the problem statement (nouns are potential objects, verbs are potential operations) and does not require any programming. Having students analyze problem statements and build class hierarchies may be a worthwhile activity for your students. Object-oriented analysis techniques are described in detail in the next chapter. Determining all operations for each class cannot be done until the objects and their interactions have been defined.

20.4 Management of Object-Oriented Projects

Students need to examine the similarities and differences between the model presented for OO project management, with the model presented in Part Two of SEPA, 5/e. This section also discusses the use of several object-oriented metrics that might be useful during project planning. 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.

PROBLEMS AND POINTS TO PONDER

20.1. You might consider assigning a few people to report on Java as an object oriented language.

20.2, 20.3. A good source of information on the "real" answer to this question can be found in Meyer's book (see *Further Readings and Other Information Sources*).

20.4. A class is a categorization of some collection of those things noted in Figure 20.9. Inheritance is like a template for attributes and operations. The template is overlaid on an object definition, enabling the object to use all attributes and operations defined for its class. Encapsulation is a packaging mechanism—it allows the analysis to combine data and process and refer to them with the same name. Polymorphism allows us to use the same method name to refer to operations of different objects. For example, we might use DRAW as a method for drawing different instances of the class SHAPE: circle, rectangle, triangle, etc.

20.6. The classes can be derived by visually examining a typical GUI. Windows, Icons, menu bars, pull-down menus, messages, are typical classes. The manner in which we interact with these objects defines both the attributes that they contain and the operations (methods) that are applied to them.

20.7. A composite object: window. It would contain a scroll bar (another object) and a variety of content (which almost certainly would itself contain other objects).

20.8. The class document would have the following attributes and operations:

```
attributes:
      name
      body-text
      margin-left
      margin-right
      header (object, document)
      footer (object, document)
      ... etc.

methods:
      save
      open
      enter/edit text
      set margin
      create header/footer
      ... etc.
```

20.9. Regardless of the programming language, the message must specify the object to which it is being sent, the method to be invoked, and the set of arguments (if any) that are required for the method to do its work (and return appropriate information, if required).

20.10. Restructuring of a class hierarchy often occurs when an existing class hierarchy has been established and it becomes apparent that classes in the middle of the hierarchy should be added to better meet the needs of the problem.

20.11. Consider a GUI environment in which a class, screen-object has attributes that define the geometric properties of a window; the class text contains the text properties that are contained in windows; the class tree contain the structure and methods than enable nested windows to be created. The class window could be represented in a manner that inherits from all three of the classes noted above.

20.13 Classes are defined and then refined recursively. For example, the first cut at a class may only define a subset of attributes or a set of attributes that includes one or two objects that also need to be refined. The full definition of the class is therefore iterative (for the entire class) and recursive (for attributes within the class). Because many different classes are required for a system, definition occurs in parallel.

Chapter 21
Object-Oriented Analysis

CHAPTER OVERVIEW AND COMMENTS

This chapter describes the process of developing an object-oriented analysis (OOA) model. The generic process described

begins with the development of use-cases (usage scenarios) and a Class-Responsibility-Collaborator (CRC) card model. The CRC card is used as the basis for developing a network of objects that comprise the object-relationship model. The event sequences implied by the use-cases provide the basis for the state transition diagram that is used to define the object-behavior model. Students should work thought the process of building a set of CRC cards and use these cards to build a complete OOA model for their own projects.

UML notation is used to illustrate some important concepts, but this chapter is *not* intended to teach OOA using UML. If you want to emphasize UML for OOA, use one or more of the references noted at the end of this chapter.

21.1 Object-Oriented Analysis

This section introduces object-oriented analysis. It is important to emphasize to students that the OOA process begins by working to build an model of the end-users' understanding of the problem space. OOA does not attempt to separate data from functions (a la structured analysis/design). OOA does not rely on functional decomposition like structured analysis/design either. Students without much experience with object-oriented programming may struggle a little bit OOA. Many students have trouble thinking of humans as system objects. Students should focus their attention on the generic OOA steps listed in Section 21.1.2. Students should also be sure that they understand conceptual differences among the five views that make up the Unified Modeling Language.

21.2 Domain Analysis

Domain analysis is discussed in this section as a process in which a set of reusable classes is defined following the examination of an application area. Students may benefit from examining 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.

21.3 Generic Components of the OO Analysis Model

This section describes the generic components that make up all object-oriented analysis models. It is important that students try to understand the static components do not change as the

application is executing. The dynamic components are influenced by the sequence of events and their timing. Some of this material will be hard for students to comprehend fully, until after they have followed the process model described in the next section.

21.4 The OOA Process

A good OOA process begins with an understanding of how people and systems will interact with one another when using the final software product. Students should be given the task of developing usage scenarios for one of their software projects. Students may need to see some additional examples of use-case diagrams for usage scenarios. 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 CRC card sets. The process of building a class hierarchy from a debugged CRC system would also be good student assignment.

21.5 The Object-Relationship Model

Once the CRC card set is complete, it is a fairly easy task to build a network of collaborating classes. This network of collaborating cards is examined for missed information and a complete object-relationship model can be derived. Students should make sure they understand how the message paths are defined by object-relationship model.

21.6 The Object-Behavior Model

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 (as shown in the text) to determine the events. The events trigger transitions between system states. Sequences of events can be used to build a state transition diagram that represents the object-behavioral model. Students should be encouraged to construct an object-behavior model for one of their own projects (preferably one that they have written use-cases for).

PROBLEMS AND POINTS TO PONDER

21.1. Realistically, this is probably too much for a homework problem. If you are not conducting a project course, you might assign this as a term paper.

21.3. An informal approach to domain analysis follows a use case scenario. Write a series of use cases that provide a generic description of the business functions and/or product functions that are relevant to the domain. This can lead to the isolation of "items" as noted in Section 21.2.2. Categorization, the definition of classes, attributes, methods, and relationships/behavior follows.

21.4. A static view is used to define objects that persist throughout the operational "life" of the application. The dynamic view considers the events that cause objects to behave in specific ways. To some extend the behavior is defined by the use of the software in real time.

21.5. The primary actors for the use case for setting a security zone are the user and the sensors involved in a particular security zone. Secondary actors include the keypad and display that enables the user to set the zone; messages that guide the user, and alarms that may be used when zones are tested.

The user defines a security zone by pressing # followed by "8" on the key pad. A message appears on the display asking for a zone definition number between 1 and 10. The user presses the appropriate number on the keypad followed by the * key. A message appears on the display asking for the sensor numbers to be included in the security zone. The users keys in the appropriate sensor number. The system validates that the sensor exists and is properly connected to the system and responds with a single beep and by listing the sensor number next to a message that reads:

Security zone # *n*
Sensors: *q, r, s*

where *n, q, r,* and *s* are sensor numbers. This procedure is repeated for all sensors to be included in the security zone. If a sensor number is incorrect or no such sensor exists, a double beep and a warning message appears on the display. When the user has completed defining all sensors in the security zone, the user presses # twice and the activity is completed. To redefine the sensors in a zone, the process is repeated from the beginning.

21.6 through 21.17. If your course emphasizes project work, you will not have time to assign these problems. However, you should require that your student develop use cases and CRC cards for their project (unless you've opted to teach a different OOA method). The use of subsystems, object-behavior and object-relationship models should also be emphasized.

21.13. An OO application achieves its function when many classes work in collaboration with one another. This occurs when the objects instantiated from one class send messages to objects instantiated from another class. Therefore, to determine the collaborators for a specific class, the list of responsibility (on a CRC card) is evaluated and the following question is asked: "How can this responsibility be achieved and what classes are involved in achieving it?" To a large extent, the methods listed for the class will provide key information on collaboration. What messages does a method generate? What classes receive the message?

21.14. The answer is stated in SEPA. When a set of classes all collaborate to achieve some responsibility, it is likely that they form a subsystem—that is, they achieve some aspect of the system to be constructed.

21.15. Cardinality plays the same role in the OOA model as it does in a standard data model. In addition, cardinality provides an indication of whether a message originating from an object is passed to one or many objects and how these objects relate to one another.

21.16. The passive state represents the values or status of an object's attributes at a given point in time. The active state indicates the status of the object based on a trigger (event) that has caused the object to make a transition from another active state.

Chapter 22
Object-Oriented Design

CHAPTER OUTLINE AND COMMENTS

This chapter describes the process of developing an object-oriented design (OOD) from an object-oriented analysis model. The generic process model discussed for object-oriented design has two major activities, object-oriented system design and object design. Object-oriented system design defines the product architecture by focusing on the specification of three components (user interface, data management functions, and task management facilities). Object design focuses on the individual class details and the messaging scheme. Students need to be aware that an OOD does not have to be implemented using an object-oriented programming language (although using an OO language is recommended). Students should be reminded that the design document template on the SEPA web site is applicable to OOA as well as structured design.

22.1 Design for OO Systems

This section describes an object-oriented version of the design pyramid presented in Chapter 13. Conventional and object-

oriented design approaches are discussed briefly. It may be worthwhile to have students examine an object-oriented design and a structured design for the same system (if SD is part of your course). The issues listed for object-oriented designs should be discussed in class and illustrated for students using a real design document. Component modularity, product quality, and potential for reuse are ultimately the real reason for using object-oriented design methodologies. Several specific OOD methods are discussed briefly in this section, students should be encouraged to focus the attention on the generic process model. The UML (unified modeling language) approach is organized around two design activities: system design and object design.

22.2 The System Design Process

Nine activities are described as part of the generic system design process model discussed in this section. The subsections give some detail on each activity. Students would benefit from studying a complete object-oriented system design model, prior to developing one for their own projects. The currency issues may be difficult for students to understand if they have not studied parallel algorithms in a previous course. Having students build their own subsystem collaboration tables would be helpful and should be easy for them if they have already built a CRC model for some system.

22.3 The Object Design Process

Object design is the part of OOD where students who have not used object-oriented programming languages may struggle (as they will be less familiar with the process of object reuse through inheritance). Junior level software engineering students will be fairly comfortable with the process of refining algorithms and building reasonable data structures. Defining protocols is not too different from defining the interfaces for abstract data types implemented in any programming language. Students should be encouraged to design their objects so as to take full advantage of any information hiding and encapsulation facilities provided by their target implementation language. If objects cannot be defined in their target language, then hopefully objects can be implemented as functions and data structures in something like an Ada package. Students would benefit from reviewing the object design used in a real world application. Students should be encouraged to perform object design for one of their own projects.

22.4 Design Patterns

This section discusses the use of two object-oriented design patterns (inheritance and composition). Students should have an opportunity to compare the advantages of the two design patterns. Perhaps they might assigned the task of building a new subsystem out of an existing class hierarchy using each design pattern.

22.5 Object-Oriented Programming

This section reinforces the importance of using OOA and OOD even if object-oriented programming will not be used for implementation. Students whose only programming experience is with object-oriented languages may benefit from seeing the implementation of an OOD in an imperative programming language.

PROBLEMS AND POINTS TO PONDER

22.1. The design pyramid for conventional software represents difference areas of design, that is, it focuses of the design of data, program architecture, interfaces and algorithms. The pyramid for OOD takes a different view (since the aforementioned design areas are bundled to some extent in the OO context). It defines layers that represent different levels of abstract of the design, beginning with the subsystem, moving to the classes that must exist to describe the subsystem and then toward messages and responsibilities.

12-2. Structured design mapped data flow into a hierarchical (factored) representation of program architecture. Data design occurs but it is decoupled from the design of the program (at least to some extent). Modules are described procedurally. OOD focuses on the design of subsystems and objects. Data and procedural design occur at the same time. There is no hierarchical modular structure for OO systems.

22.3. Subsystem design achieves decomposability. Object design, if properly done, should create reusable components and therefore achieve composability. Object design that emphasizes effective encapsulation will decouple the design and lead to better understandability and continuity. Translation of information about collaborations and responsibilities into effective message design will lead to protection.

22.4 and 22.5. These are useful assignments for graduate courses that emphasize OO design.

22.6. The use represents a scenario of usage. As such, its primary contribution is in the description of user requirements for the system. However, the definition of actors and the procedural actions that actors initiate will help in understanding the behavior of the system and in translating basic class definition (done during analysis) into OOD representations. Use cases can be categorized into collections of scenarios that provide guidance on which subsystems make up the OO system.

22.7. You might suggest that your student acquire a copy of one of the many textbook that address "windows programming."

22.10. Your students may have to review the discussion of real-time techniques presented at the SEPA Web site, if they have not already done so before addressing this problem. Although there are many keys to the definition of concurrent tasks, one of the most important indicators is the representation of classes within subsystems. Timing of subsystem functions and the location of different processors provide an indication of which tasks will be concurrent.

22.11 through 22.15. If you are conducting a projects course, you will not have time to assign these problems. However, if you do, I would suggest assigning them piecewise over a number of weeks. First have students do an OOA on the problem, then have them do a subsystem design; after this is complete, object design and other more detailed design activities can commence. The idea here is to be sure that they're headed in the right direction before they move to lower levels of abstraction (where they'll undoubtedly feel more comfortable).

Chapter 23
Object-Oriented Testing

CHAPTER OUTLINE AND COMMENTS

This chapter discusses object-oriented testing (OOT) by comparing it with conventional software testing (Chapters 17, 18). The most important point to get across to students is that the majority of OOT focuses on the OO class, rather than on individual operations within a class. This means that the lowest level of testing in OOT resembles the first level of integration testing in conventional software testing. White box testing of the operator algorithms is not very meaningful. It is also important for students to understand that OOT really begins during the review of OOA and OOD models. Most black-box testing techniques are appropriate for OOT. Students should be encouraged to use the test plan specification document from the SEPA webs site to develop an OOT test plan for one of their semester projects.

23.1 Broadening the View of Testing

This section tries to broaden the student's view of testing. Because of the evolutionary nature of OO software development, testing is an ongoing concern and begins as soon as the OOA model components are completed (since the same semantic elements appear in the analysis, design, and code levels). Students might need to be reminded that the earlier a defect is removed in the software development cycle, the less costly it is to remove. This is especially true for evolutionary software process models.

23.2 Testing OOA and OOD Models

Two dimensions (correctness and consistency) of testing OOA and OOD models are discussed in this section. Syntactic correctness is judged by reviewing the notation and modeling conventions used in the OOA and OOD models. Semantic correctness of the models is judged based on their conformance to the real world problem domain (based on the judgement of problem domain experts). Reviewing collaborations defined in the CRC model can be used to assess the consistency of the OOA model. The OO system design is reviewed by examining the object-behavior model, subsystem allocations to processors, and the user interface design as specified in the use-case scenarios. The object design is reviewed by examining the lower level details of the object model. More detailed discussion of the mechanics of OOD reviews appear in the subsequent sections of this chapter.

23.3 Object-Oriented Testing Strategies

This section clarifies the differences between OOT and conventional testing with regard to unit testing, integration testing, and validation 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.

23.4 Test Case Design for OO Software

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. Fault-based testing and scenario-based testing are complementary testing techniques that seem particularly well-suited to OO test case design. White-box test case construction techniques are not well suited to OOT. Students should be encouraged to develop a set of test cases for an OO system of their own design. Students need to be reminded of the fact that the process of inheritance

does excuse them from having to test operators obtained from superclasses (their context has changed). Similarly, operators redefined in subclasses will need to be tested in scenarios involving run-time resolution of the operator calls (polymorphism). 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.

23.5 Testing Methods Applicable at the Class Level

This section 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.

23.6 Interclass Test Case Design

This section 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.

PROBLEMS AND POINTS TO PONDER

23.1. The class encapsulates data and the operations that manipulate the data. Since these are packaged as a unit, testing just the methods one-by-one would be sterile and would not uncover errors associated with messaging, responsibilities and collaborations.

23.2. Because each subclass of a given class is applied within the context of private attributes and operations (and the fact that these private attributes and operations) can add complexity), subclasses must be retested in their operational context. Test cases can be reused, but it is important to extend them to address the private attributes and operations of the subclass.

23.3. If the analysis and design models are incorrect or erroneous, it follows that the resultant program will be flawed. The costs associated with correcting the flawed implementation will be much higher than the costs associated with "testing" the analysis and design models.

23.5. Thread-based testing is used to integrate a set of classes that are required to respond to a single program input or a single event. Use based testing is an integration testing strategy that begins by integrating the classes that do not collaborate with "server" classes (these are called independent classes). Then, classes that collaborate with the independent classes are tested in a layered fashion.

23.6 through 23.12. If you are conducting a projects course, you will not have time to assign these problems. However, if you do, I would suggest supplementing lecture discussions on testing with outside reading from Binder. His text is the most thorough treatment of OOT published to date.

Chapter 24
Technical Metrics for Object-Oriented Systems

CHAPTER OUTLINE AND COMMENTS

This chapter describes several metrics that may be useful in assessing the quality of object-oriented systems. Students may need to be reminded of the importance of using quantitative measures to provide objective guidance in improving the quality of a work or process. Analogies from other areas of engineering and construction are useful.

The basic process of using metrics is the same for both object-oriented and conventional software, but a change of focus is required. In object design the smallest design unit is the encapsulated class and this is generally the target of OO metrics.

It will be easy for students with weak statistics backgrounds to be overwhelmed by the number of equations listed in this chapter. However, none of the metrics presented in this chapter is difficult for students to compute. Students should be encouraged to compute several representative metrics for one of their own object-oriented course projects. The analysis and interpretation of the metrics computed may not be possible unless students have access to some sort of historical project data.

24.1 The Intent of OO Metrics

This section states that the goals for using OO metrics are the same as the goals for using metrics for conventional software development projects. There are, of course, differences in the types of measures that need to be collected, since the primary unit of focus in OO systems is class or object and not individual algorithms.

24.2 The Distinguishing Characteristics of OO Metrics

Students should become familiar with the five characteristics of OO software described in this section (localization, encapsulation, information hiding, inheritance, and abstraction). These are the characteristics to keep in mind when considering candidate measures to use in computing technical metrics for OO software. These characteristics can also be found in conventional software designs. What has changed is the size of the package. Each of the five characteristics must be considered from the perspective of the class or object as the atomic design component.

24.3 Metrics for the OO Design Model

This section 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.

24.4 Class-Oriented Metrics

Three different suites of class-oriented metrics are discussed in this section. 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.

24.5 Operation-Oriented Metrics

Three operation-oriented metrics are presented in this section. Students will need to refer to earlier chapters on metrics to have enough information to compute these metrics. It may be beneficial to compute and interpret these metrics for the design model used for a real OO system.

24.6 Metrics for OO Testing

Six new metrics (and three class-oriented metrics from section 24.4) that may be useful in determining the testability of OO systems are described in this section. 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.

24.7 Metrics for OO Projects

Three metrics useful in OO project management work are described in this section. The metrics are presented in enough detail to allow students to compute them for their own projects. These metrics can be used along with the OO design metrics to derive productivity metrics for OO projects. Students may benefit from having access to historical project data and then using these metrics to estimate effort, duration, or staffing requirements for an OO project plan.

PROBLEMS AND POINTS TO PONDER

24.1. The intent of both types of metrics is the same—to assess the quality of the work product. It can be argued that conventional technical metrics are tied more closely to the mode of representation (e.g., DFD or ERD) while OO metrics are more closely aligned with characteristics that are independent of the mode of representation.

24.2. Because conventional software is functionally localized, technical metrics tend to focus on a functional unit, the model or on a data unit, a specific data structure. The localized unit for Oo software is the class. Therefore, technical metrics tend to focus on class level issues.

24.3. The class is the thing to asses for quality and since it encapsulates both data and operations, they should be considered as a package. hence, these is a deemphasis on procedurally oriented metrics.

24.4. Among the metrics that provide a direct or indirect indication of information hiding are: CBO, RFC, LCOM, NP_{avg}, and PAD.

24.6. WMC is simply the sum of each of the individual complexities divided by the average complexity:

WMC = 1.25 + 1 + 0.75 + 0.75 + 1.5 + 2 + 0.5 +0.5 +1.25 + 1.25 + 1 + 1 = 12.75

24.7. DIT for the trees in Figure 20.8 is 4 and 4 respectively. NOC for X2 is 2 in both figures.

24.9. NOA for both classes is 1. That is, each of the subclasses adds 1 private attribute.

24.11. The value for NSS should correspond closely to the number of use cases generated for each of the projects presented in the Table. One approach to this problem would be to compute averages for all five projects. The other would be to use regression analysis and derive a relationship between NSS and NKC and effort (linear curve fitting could also be used. Once a relationship is established, the new value of NSS = 95 could be used to project estimated values for NKC and effort.

Part V

Chapter 25
Formal Methods

CHAPTER OVERVIEW AND COMMENTS

This chapter presents an introduction to the use of formal methods in software engineering. The focus of the discussion is on why formal methods allow software engineers to write better specifications than can be done using natural language. Students without precious exposure to set theory, logic, and proof of correctness (found in a discrete mathematics course) will need more instruction on these topics than is contained in this chapter. The chapter contains several examples of specifications that are written using various levels of rigor. If your students are familiar with a specification language (like Z) they should be encouraged to use it to write formal specifications for some of their own functions. If they are not familiar with a specification language, it may be worthwhile to teach them one (if your course is two semesters long). Having students write correctness proofs for their function specifications will be difficult and you may need to review proof techniques and heuristics if you wish students to write proofs.

25.1 Basic Concepts

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 Z or VDM).

25.2 Mathematical Preliminaries

A review of the mathematics needed for the remainder of the chapter appears in this section. If your students have completed a good course in discrete mathematics, this review should be adequate. If they have not, some additional (and more concrete) examples will need to be presented to your students. Constructive set specification writing is a very important concept for your students to understand, as is work with predicate calculus and quantified logic. Formal proofs of set theory axioms and logic expressions is not necessary, unless you plan to have your students do correctness proofs for their specifications. Work with sequences may be less familiar to your students, if they have not worked with files and lists at an abstract level.

25.3 Applying Mathematical Notation for Formal Specification

This section uses mathematical notation to refine the block handler specification from Section 25.1.3. It may be desirable to refine the other two specification examples from Section 25.1 using similar notation. If your students are comfortable with mathematical proof, you may wish to present 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.

25.4 Formal Specification Languages

This section discusses the properties of formal specification languages from a theoretical perspective. The next section, uses the Z specification language to rewrite the block handler specification more formally. You might have students try writing the specifications for their own functions using a pseudocode type notation embellished with comments describing semantic information.

25.5 Using Z to Represent and Example Software Component

The Z specification language is used to rewrite the block handler specification from Sections 25.1.3 and 25.3. If your students are familiar with some other specification language you may want to present the block handler specification using that language. If your students are not familiar with the Z specification language, it may be beneficial to use the Z

language to write a formal specification for some function that they understand well. Showing them a correctness proof using the Z language notation may also be a worthwhile activity.

25.6 The Ten Commandments of Formal Methods

Class discussion of the 10 commandments listed in this section would be beneficial to students. They need to be able to articulate in their own words, why these commandments are important when using formal methods in software engineering.

25.7 Formal Methods—The Road Ahead

This section mentions several obstacles to using formal methods on large real world several development projects. Students should acknowledge that formal methods require discipline and hard work. They should also recognize that the there are times when the risks associated with using informal specification techniques make the use of formal methods necessary.

A Detailed Example of the Z Language

To illustrate the practical use of a specification language, Spivey [1] considers a real-time operating system kernel and represents some of its basic characteristics using the Z specification language [SPI88]. The remainder of this section has been adapted from his paper (with permission of the IEEE).

****

Embedded systems are commonly built around a small operating-system kernel that provides process-scheduling and interrupt-handling facilities. This article reports on a case study made using Z notation, a mathematical specification language, to specify the kernel for a diagnostic X-ray machine.

Beginning with the documentation and source code of an existing implementation, a mathematical model, expressed in Z, was constructed of the states that the kernel could occupy and the events that could take it from one state to another. The goal was a precise specification that could be used as a basis for a new implementation on different hardware.

This case study in specification had a surprising by-product. In studying one of the kernel's operations, the potential for deadlock was discovered: the kernel would disable interrupts and enter a tight loop, vainly searching for a process ready to run.

---

1 Spivey, J.M., "Specifying a Real-Time Kernel," *IEEE Software,* September, 1990, pp. 21 - 28.

This flaw in the kernel's design was reflected directly in a mathematical property of its specification, demonstrating how formal techniques can help avoid design errors. This help should be especially welcome in embedded systems, which are notoriously difficult to test effectively.

A conversion with the kernel designer later revealed that, for two reasons, the design error did not in fact endanger patients using the X-ray machine. Nevertheless, the error seriously affected the X-ray machine's robustness and reliability because later enhancements to the controlling software might reveal the problem with deadlock that had been hidden before.

The specification presented in this article has been simplified by making less use of the schema calculus, a way of structuring Z specifications. This has made the specification a little longer and more repetitive, but perhaps a little easier to follow without knowledge of Z.

About the Kernel

The kernel supports both background processes and interrupt handlers. There may be several background processes, and one may be marked as current. This process runs whenever no interrupts are active, and it remains current until it explicitly releases the processor, the kernel may then select another process to be current. Each background process has a ready flag, and the kernel chooses the new current process from among those with a ready flag set to true.

When interrupts are active, the kernel chooses the most urgent according to a numerical priority, and the interrupt handler for that priority runs. An interrupt may become active if it has a higher priority than those already active and it becomes inactive again when its handler signals that it has finished. A background process may become an interrupt handler by registering itself itself as the handler for a certain priority.

Documentation

Figures 9.15 and 9.16 are diagrams from the existing kernel documentation, typical of the ones used to describe kernels like this. Figure 9.15 shows the kernel data structures. Figure 9.16 shows the states that a single process may occupy and the possible transitions between them, caused either by a kernel call from the process itself or by some other event.

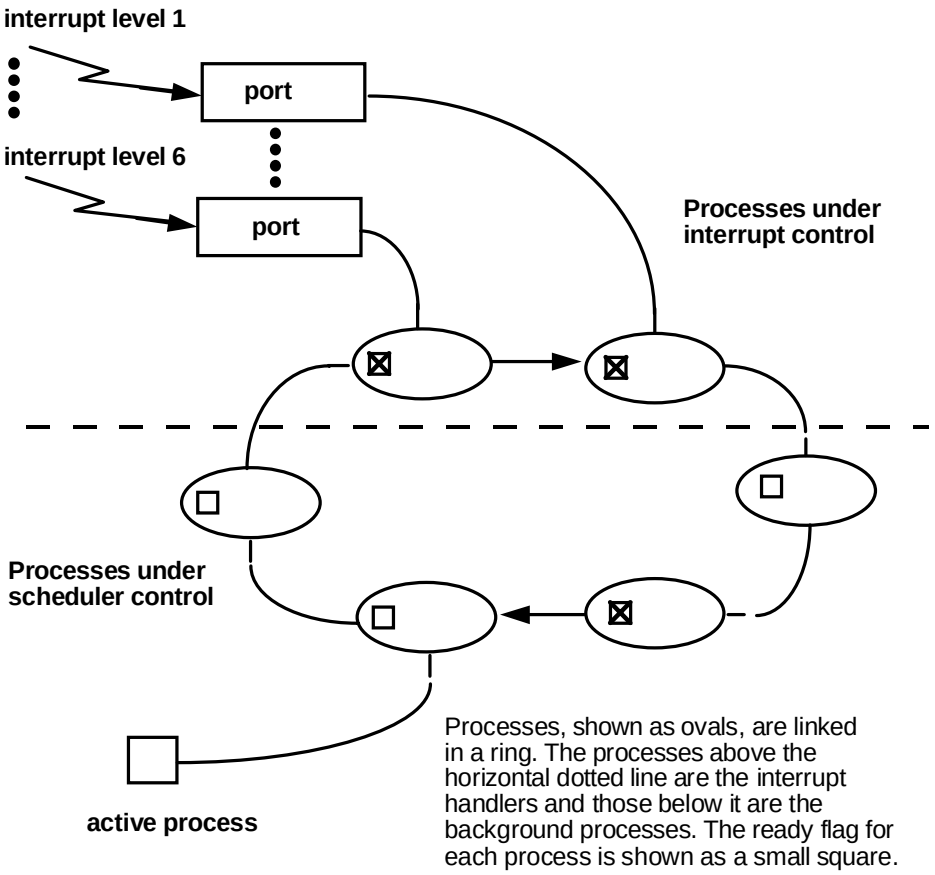

Figure 9.15 Data structures for O.S. kernel [SPI90]

In a way, Figure 9.16 is a partial specification of the kernel as a set of finite-state machines, one for each process. However, it gives no explicit information about the interactions between processes—the very thing the kernel is required to manage. Also, it fails to show several possible states of a process. For example , the current background process may not be ready if it has set its own ready flag to false, but the state "current but not ready" is not shown in the diagram. Correcting this defect would require adding two more states and seven more transitions. This highlights another deficiency of state diagrams like this: their size tends to grow exponentially as a system complexity increases.

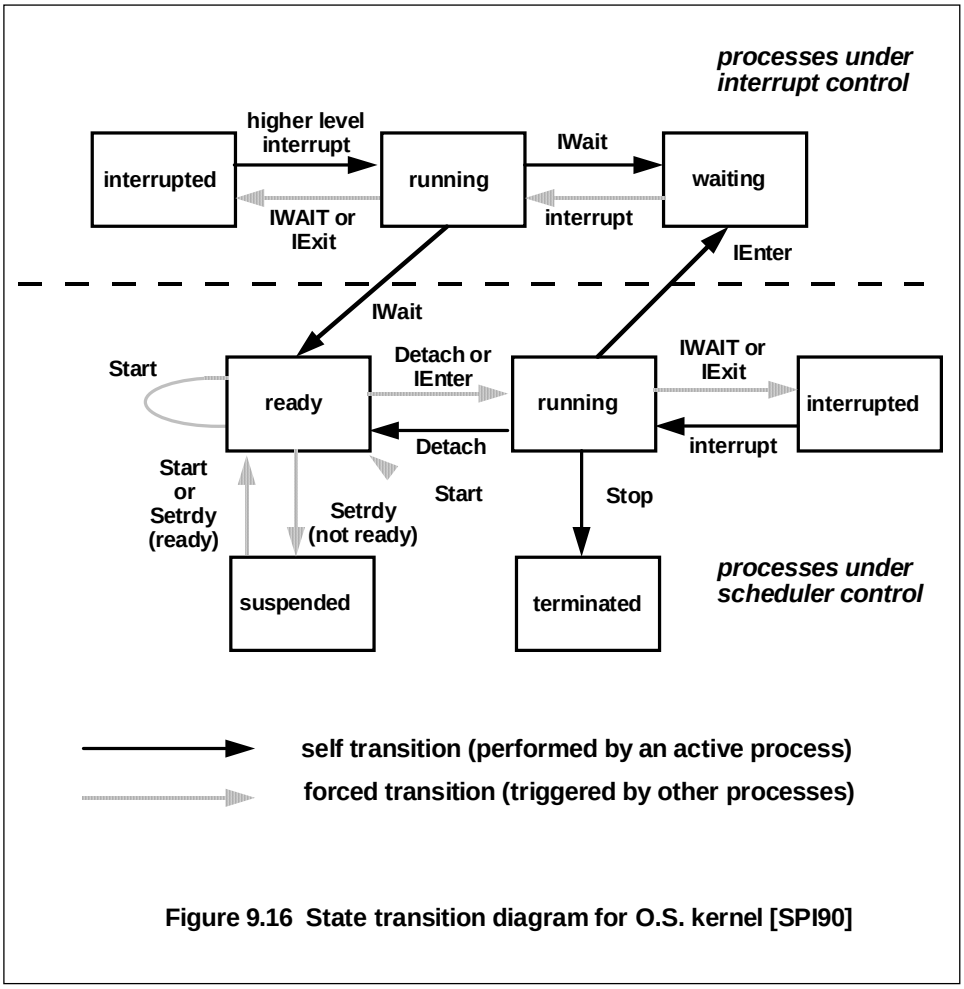

Figure 9.16 State transition diagram for O.S. kernel [SPI90]

Kernel States

Like most Z specifications, the kernel model begins with a description of the *state space,* the collection of variables that determine what state the kernel is in and the invariant relationships that always hold between these variables' values.

In this article, the kernel state space is described in several parts, corresponding to the background processes, the interrupt handlers, and the state of the processor on which the kernel runs. Each piece is described by a *schema,* the basic unit of specification in Z. Table 9.1 {Table 25.2 in SEPA, 4/e] describes the Z notation used in this case study.

The state space of the whole kernel is obtained by putting together these pieces and adding more invariants, among them the static policy for allocating the processor to a background process or interrupt handler. Processes are named in the kernel by *process identifiers*. In the implemented kernel, these are the addresses of process-control blocks, but this detail is irrelevant to a programmer using the kernel, so they are introduced as a basic type PID:

[PID]

This declaration introduces PID as the name of a set, without giving any information about its members. From the specification's point of view, the members are simply atomic objects.

For convenience, a fictitious process identifier, *none,* was introduces. *none* is not the name of any genuine process. When the processor is idle, the current process is *none.* The set PID_1 contains all process identifiers except none:

$none$: PID
PID_1: $\mathbb{P}$ PID
———————————
$PID_1 = PID \setminus \{none\}$

The part of the kernel state concerned with background processes is described by this schema:

————Scheduler————————————
background: $\mathbb{P}$ PID_1
ready: $\mathbb{P}$ PID_1
current: PID
———————————————————
ready $\subseteq$ *background*
current $\in$ *background* $\cup$ *{none}*
———————————————————

Like all schema, this one declares some typed variables and states a relationship between them. Above the horizontal dividing line, in the *declaration* part, the three variables are declared:

- *background* is the set of processes under control of the scheduler;
- *ready* is the set of processes that may be selected for execution when the processor becomes free;
- *current* is the process selected for execution in the background.

Below the line, in the *predicate* part, the schema states two relationships that always hold. The set *ready* is always a subset of *background,* and *current* is either a member of *background* or the fictitious process *none.* This schema lets the current process be not *ready,* because no relationship between *current* and *ready* is specified.

This schema does not reflect the full significance of the set *ready,* but its real meaning will be shown later in the *Select* operation, where it forms the pool from which a new current process is chosen.

Interrupts are identified by their priority levels, which are small positive integers. The finite set *ILEVEL* includes all the priorities:

ILEVEL: $\mathbb{F}$ N
———————————

$0 \notin \text{ILEVEL}$

Zero is not one of the interrupt priority levels, but it is the priority associated with background processes.

The state space for the interrupt-handling part of the kernel is described like this:

———— *IntHandler* ————

$handler : ILEVEL \rightarrowtail PID_1$

$enabled, active : \mathbf{P}\, ILEVEL$

———————————

$enabled \cup active \subseteq \mathbf{dom}\, handler$

———————————

This schema declares the three variables:

- *handler* is a function that associates certain priority levels with processes, the interrupt handlers for those priorities;
- *enabled* is the set of priority levels that are enabled, so an interrupt can happen at that priority;
- *active* is the set of priority levels for which an interrupt is being handled.

The predicate part of this schema says that each priority level that is either enabled or active must be associated with a handler. An interrupt may be active without being enabled if, for example, it has been disabled by the handler itself since becoming active. The declaration

$handler : ILEVEL \rightarrowtail PID_1$

declares handler to be a partial injection. It is partial in that not every priority level need be associated with an interrupt handler and it is an injection in that no two distinct priority levels may share the same handler.

Information like this — that interrupts may be active without being enabled and that different priority levels may not share a handler— is vital to understanding how the kernel works. It is especially valuable because it is static information about the states the kernel may occupy, rather than about what happens as the system moves from state to state. Such static information is often absent from the text of the program, which consists mostly of dynamic, executable code.

The state space of the whole kernel combines the background and interrupt parts:

———— *Kernel* ————

Scheduler

IntHandler

———————————

$\text{background} \cap \mathbf{ran}\, \text{handler} = \emptyset$

The declarations *Scheduler* and *IntHandler* in this schema implicitly include above the line all the declarations from those schemas and implicitly include below the line all their invariants. So the schema *Kernel* has six variables: *background, ready,* and *current* from *Scheduler,* and *handler, enabled,* and *active* for *IntHandler.*

The additional invariant has been added so that no process can be both background process and an interrupt handler at the same time.

The main job of the kernel is to control which process the processor runs and at what priority. Therefore, the *running* process and the processor *priority* have been made part of the state of the system. Here is a schema that declares them:

CPU

$$running : PID$$
$$priority : ILEVEL \cup \{0\}$$

This schema has an empty predicate part, so it places no restriction on the values of its variables, except that each must be a member of its type. The variable *running* takes the value *none* when no process is running.

Of course, there are many other parts of the processor state, including the contents of registers and memory, condition codes, and so on, but they are irrelevant in this context.

Because the kernel always uses the same scheduling policy to select the running process and the CPU priority, this policy is another invariant of the system. It is stated in the schema *State,* which combines the kernel and CPU parts of the system state:

State

$$Kernel$$
$$CPU$$

$$priority = max(active \cup \{0\})$$
$$priority = 0 \Rightarrow running = current$$
$$priority > 0 \Rightarrow running = handler(priority)$$

If any interrupts are active, the processor priority is the highest priority of an active interrupt, and the processor runs the interrupt handler for that priority. Otherwise, the processor runs the current background process at priority zero.

The invariant part of this schema uniquely determines *priority* and *running* in terms of *active, current,* and *handler,* three variables of the schema *Kernel.* This fact will be exploited when the events that change the system state are described. With the description of the kernel's and processor's state space complete, the next step is to look at the operations and events that can change the state.

Background Processing

Some kernel operations affect only the background processing part of the state space. They start background processes, set and clear their ready flags, let them release the processor temporarily or permanently, and select a new process to run when the processor is idle.

A process enters the control of the scheduler through the operation *Start,* described by this schema:

$$
\begin{array}{l}
\hline
\textit{Start} \\
\Delta \textit{State} \\
p? : \textit{PID}_1 \\
\hline
p? \notin \textbf{ran}\ \textit{handler} \\
\textit{background}' = \textit{background} \cup \{p?\} \\
\textit{ready}' = \textit{ready} \cup \{p?\} \\
\textit{current}' = \textit{current} \\
\theta \textit{IntHandler}' = \theta \textit{IntHandler} \\
\hline
\end{array}
$$

LIke all schemas describing operations, this one includes the declaration *_State,* which implicitly declares two copies of each variable in the state space *State,* one with a prime (') and one without. Variables like *background* and *ready* without a prime refer to the system state before the operation has happened, and variables like *background'* and *ready'* refer to the state afterward.

The declaration *_State* also implicitly constrains these variables to obey the invariant relationships that were documented in defining the schema *State*—including the scheduling policy—so they hold both before and after the operation.

In addition to theses state variables, the *Start* operation has an input *p?,* the identifier of the process to be started, By convention, inputs to operations are given names that end in a ?.

The predicate part of an operation schema lists the precondition that must be true when the operation is invoked and postcondition that must be true afterward. In this case, the precondition is explicitly stated: that the process *p?* being started must not be an interrupt handler (because that would violate the invariant that background processes are disjoint from interrupt handlers).

In general, an operation's precondition is that a final state exists that satisfies the predicates written in the schema. Part of the precondition may be implicit in the predicates that relate the initial and final states. If an operation is specified be the schema *Op,* its precondition can be calculated as

$$\exists \textit{State} \bullet \textit{Op}$$

If the precondition is true, the specification requires that the operation should terminate in a state satisfying the postcondition. On the other hand, if the precondition is false when the operation is invoked,

the specification says nothing about what happens. The operation may fail to terminate, or the kernel may stop working completely.

For the *Start* operation, the postcondition says that the new process is added to the set of background processes and marked as ready to run. The new process does not start to run immediately, because *current* is unchanged; instead, the processor continues to run the same process as before.

The final equation in the postcondition

$$\theta Inthandler' = \theta Inthandler$$

means that the part of the state described by the schema *IntHandler* is the same after the operation as before it.

The equation in this schema determines the final values of the six variables in the kernel state space in terms of their initial values and the inputs *p?,* but they say nothing about the final values of the CPU variables *running* and *priority.* These are determined by the requirements, implicit in the declaration *_State,* that the scheduling policy be obeyed after the operation has finished. Because the values of *active, handler,* and *current* do not change in the operation, neither does the CPU state.

The current background process may release the processor by calling the *Detach* operation, specified like this:

```
------Detach---------
  _State
----------------------
  running ∈ background
  background' = background
  ready' = ready
  current' = none
  θIntHandler' = θ IntHandler
----------------------
```

Again, this operation is described using *_State* variables before and after the operation has happened. The precondition is that the process is running a background process. The only change specified in the postcondition is that the current process changes to none, meaning that the processor is now idle. The next event will be either an interrupt or the selection of a new background process or run.

After a call to *Detach*—and after other operations described later—*current* has values *none,* indicating that no background process has been selected for execution. If no interrupts are active, the processor is idle, and the *Select* operation may happen spontaneously. It is specified like this:

```
------Select---------
  _State
----------------------
  running =none
```

$background' = background$
$ready' = ready$
$current' \in ready$
$\theta IntHandler' = \theta IntHandler$

Rather than a part of the interface between the kernel and an application, *Select* is an internal operation of the kernel that can happen whenever its precondition is true. The precondition is

$running = none \land ready \neq \emptyset$

The processor must be idle, and at least one background process must be ready to run. The first part of this precondition is stated explicitly, and the second part is implicit in the predicate

$current' \in ready$

The new value of *current* is selected from *ready,* but the specification does not say how the choice is made—it is nondeterministic. This nondeterminism lets the specification say exactly what programmers may rely on the kernel to do: there is no guarantee that processes will be scheduled in a particular order.

In fact, the nondeterminism is a natural consequence of the abstract view taken in the specification. Although the program that implements this specification is deterministic—if started with the ring of processes in a certain state, it will always select the same process—it appears to be nondeterministic if the *set* of processes that are ready are considered, as has been done in the specification.

However, the kernel selects the new current process, the specification says that it starts to run, because of the static scheduling policy, which determines that after the operation, running is current and priority is zero.

A background process may terminate itself using the *Stop* operation:

Stop

$\Xi State$

$running \in background$
$background' = background \setminus \{current\}$
$ready' = ready \setminus \{current\}$
$current' = none$
$\theta IntHandler' = \theta IntHandler$

For this operation to be permissible, the processor must be running a background process. This process is removed from *background* and *ready,*

and the current process becomes *none,* so the next action will be to select another process.

A final operation, *SetReady,* sets or clears a process's ready flag. It has two inputs, the process identifier and a flag , which takes one of the values *set* or *clear:*

FLAG :: = *set* \ *clear*

The SetReady operation is:

$$
\begin{array}{l}
\text{------}\ SetReady\ \text{-------} \\
\quad \_State \\
\quad p?: PID \\
\quad flag?: FLAG \\
\text{-----------------} \\
\quad p? \in background \\
\quad flag? = set \Rightarrow ready' = ready \cup \{p?\} \\
\quad flag? = clear \Rightarrow ready' = ready \setminus \{p?\} \\
\quad background' = background \\
\quad current' = current \\
\quad \theta IntHandler' = \theta IntHandler \\
\text{-----------------}
\end{array}
$$

The precondition is that *p?* is a background process, according to the value of *flag?,* it is either inserted in *ready* or removed from it. The scheduling parameters do not change, so there is no change in the running process.

Interrupt handling

Other operations affect the kernel's interrupt-handling part. A background process may register itself as the handler for a certain priority level by calling the operation *IEnter:*

$$
\begin{array}{l}
\text{------}\ IEnter\ \text{-------} \\
\quad \_State \\
\quad i?: ILEVEL \\
\text{-----------------} \\
\quad running \in background \\
\quad background' = background \setminus \{current\} \\
\quad ready' = ready \setminus \{current\} \\
\quad current' = none \\
\quad handler' = handler \oplus \{i? \mapsto current\} \\
\quad enabled' = enabled \cup \{i?\} \\
\quad active' = active \\
\text{-----------------}
\end{array}
$$

This operation may be called only by a background process. The operation removes the calling process from *background* and *ready,* and the new value of current is *none,* just as in the Stop operation. Also, the calling process becomes an interrupt handler for the priority level *i*?, given as an input to the operation, and that priority level becomes enabled. The expression

$$handler \oplus \{i? \mapsto current\}$$

denotes a function identical to handler, except that *i*? is mapped to *current,* This function is an injection, because *current,* a background process, cannot already be the handler for any other priority level. The new handler supersedes any existing handler for priority *i*?, which can never run again unless restarted with the *Start* operation.

Once a process has registered itself as an interrupt handler, the scheduler chooses a new process to run in the background, and the new interrupt handler waits for an interrupt to happen:

$$
\begin{array}{l}
\text{------}\ Interrupt\ \text{-------} \\
\quad \_State \\
\quad i?: ILEVEL \\
\text{----------------} \\
\quad i? \in enabled \ \land\ i? > priority \\
\quad \theta Scheduler' = \theta Scheduler \\
\quad handler' = handler \\
\quad enabled' = enabled \\
\quad active' = active \cup \{i?\} \\
\text{----------------}
\end{array}
$$

The process hardware ensures that interrupts happen only when they are enabled and have priority greater than the processor priority. If these conditions are satisfied, the interrupt can happen and the kernel then adds the interrupt can happen and the kernel then adds the interrupt to *active.*

The scheduling policy ensures that the associated interrupt handler starts to run. In this calculation of the processor priority, each step is justified by the comment in brackets:

$$
\begin{array}{ll}
 & Priority' \\
= & \text{[scheduling policy]} \\
 & max(active' \cup \{0\}) \\
= & \text{[postcondition]} \\
 & max((active \cup \{i?\}) \cup \{0\}) \\
= & [\ \cup \text{ assoc. and comm.}] \\
 & max((active \cup \{0\}) \cup \{i?\}) \\
= & \text{max dist. over } \cup\ ] \\
 & max\{max(active \cup \{0\}), i?\} \\
= & \text{[scheduling policy]} \\
 & max\{priority, i?\} \\
= & [\ i? > \text{priority}]
\end{array}
$$

i?

So priority' = *i*? > 0 and the other part of the scheduling policy ensures that *running*' equals handler (*i*?).

After the interrupt handler has finished the processing associated with the interrupt, it calls the kernel operation *IWait* and suspends itself until another interrupt arrives. *IWait* is specified as

```
――――――IWait――――――――
  _State
――――――――――――――――――
  priority > 0
  θ Scheduler' = θ Scheduler
  handler' = handler
  enabled' = enabled
  active' = active \ {priority}
――――――――――――――――――
```

The precondition *priority* > 0 means that the processor must be running an interrupt handler. The current priority level is removed from active, and as for the *Interrupt* operation, the scheduling policy determines what happens next. If any other interrupts are active, the processor returns to the interrupt handler with the next highest priority. Otherwise, it returns to the current background process.

Another kernel operation, *IExit,* lets an interrupt handler cancel its registration:

```
――――――IExit――――――――
  _State
――――――――――――――――――
  priority > 0
  background' = background ⌣ {handler (priority)}
  ready' = ready ⌣ {handler (priority)}
  current' = current
  handler' = {priority} ⩤ handler
  enabled' = enabled \ {priority}
  active' = active \ {priority}
――――――――――――――――――
```

Again, the processor must be running an interrupt handler. This handler leaves the interrupt-handling part of the kernel and becomes a background process again, marked as ready to run. As with *IWait,* the processor returns to the interrupt handler with the next highest priority or to the current background process. The process that called *IWait* is suspended until the scheduler selects it for execution. In this schema, the expression

{*priority*} ⩤ *handler*

denotes a function identical to handler except that priority has been removed from its domain; it is an injection provided that handler is one.

Two more kernel operations, *Mask* and *Unmask,* let interrupt priorities be selectively disabled and enabled. Their specifications are like *SetReady,* so have been omitted. The kernel specification is now complete.

PROBLEMS AND POINTS TO PONDER

25.1. Vagueness occurs is actual specification when the customer is unsure of requirements, For example, the use of words like "many, sometimes, usually" are flags that indicate a vague specification. Bounding (hopefully quantitative) is essential. Incompleteness is often encountered when lists are specified, ending with "etc." For example, the statement, SafeHome software uses a number of sensors that include fire, smoke, water, etc." indicates a potentially incomplete specification. Mixed levels of abstraction often occur when technologists specify a system. The specifier moves too quickly to implementation detail. For example, a statement like: "The customer swipes his bank card through the local ATM card reader ... the bank card numbers to be processed by the ATM server will be help in a FIFO queue ..." mixes customer visible abstraction with low level technical detail (a FIFO queue).

25.2. Mathematics requires additional learning. It may be more difficult for others to understand the specification. See *IEEE Trans.Software Engineering,* February, 1996, pp. 158ff for a discussion of this.

25.3. Data invariant: the phone book will contain no more than maxnames names; there will be no duplicate names in the phone book. The state is the data that the software access, therefore, in this case, the state is the phone book itself. Operations are similar to the operations defined for objects (Chapter 19) in that they are capable of changing the state. Operations for this problem include: add, delete, edit, obtain.

25.4. The data invariant is the memory block list that contains the start and end address of the unused memory, along with appropriate assignment information; any given block can only be assigned to one program at a time. The state is the memory block table. Operations for this problem include: add, delete, assign, lookup.

25.5. $\{x,y,z:\mathbf{N} \mid x + y = z \bullet (x,y,z^2)\}$

25.6. The operator $\in$ and/or the operator $\subset$ are candidates. A particular piece of hardware and/or software is compared to the set of required hardware and software. Alternatively, a list of required hardware and software are stored in appropriate sets

and then compared to the set of hardware and software that a particular system has.

Chapter 26
Cleanroom Software Engineering

CHAPTER OVERVIEW AND COMMENTS

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. His approach, called *cleanroom software engineering,* is the focus of this chapter.

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 intent of this chapter is to introduce the student to each of these activities.

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.

26.1 The Cleanroom Approach

This section introduces the key concepts of cleanroom software engineering and discusses its strengths and weaknesses. An outline of the basic cleanroom strategy is presented. Students will need some additional information on the use of box specifications and probability distributions before they can apply this strategy for their own projects.

26.2 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.

26.3 Cleanroom Design

If you plan to have your students verify their box specifications formally, you may need to show them some examples of the techniques used (if you did not already do so when covering Chapter 25). 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.

26.4 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.

PROBLEMS AND POINTS TO PONDER

25.1. Proof of correctness applied as software is specified is probably the most significant departure from other approaches.

25.2. Each increment in the cleanroom process may be planned to produce one or more certified components. As the increments are completed, the software is integrated and because the components are certified, integration can occur with relatively little concern for error.

25.3. To some extent the black box specification is similar to a level 0 data flow diagram in that inputs and outputs are specified for the box. Therefore, the SafeHome black box would be similar to the level 0 DFD presented in Chapter 12. The state box is analogous to the state transition diagram. The clear box representation is analogous to procedural design and can be represented using a PDL.

25.6 and 25.7. A detailed solution to this problem and many other common algorithms can be found in [LIN79].

Chapter 27
Component-Based Software Engineering

CHAPTER OVERVIEW AND COMMENTS

The software engineering community must keep pace with the growing demand for computer software, and it can be argued that current technology falls short. In addition, the user community's demands higher quality and shorter delivery

timelines. There is really only one hope—program construction using certified reusable components.

This chapter describes component-based software engineering (CBSE) as a process that emphasizes the design and construction of systems out of highly 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. It is also important to remind students that formal technical reviews and testing are still used to ensure the quality of the final product. The advantage, of course, is that less time is spent designing and coding, so software can be produced less expensively. Students would benefit from having access to a library of commercial off-the-shelf components or at least an locally developed component library and being assigned an exercise involving CBSE.

27.1 Engineering of Component Based Systems

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.

27.2 The CBSE Process

This is a short section that contains activity diagrams for the two CBSE engineering activities (domain engineering and component-based engineering). The most important things to show students in this diagram are the interrelationships

between the domain engineering and component-based engineering activities. The activities are discussed in more detail in the subsequent chapter sections.

27.3 Domain Engineering

This section reviews the material on domain engineering presented in Chapter 21 and expands on it considerably. 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.

27.4 Component-Based Development

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.

27.5 Classifying and Retrieving Components

This section discusses the issues associated with indexing and retrieving software components from a reuse library. It also describes the necessary features for a component reuse environment. 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.

27.6 Economics of Reuse

This section discusses the economics of software reuse by examining its impact on software quality, programmer productivity, and system development cost. The calculations are not hard to follow. Students might appreciate seeing the benefits gained from reuse by examining project data from a real world example. Some discussion of structure points during project estimation appears in this section. Students might appreciate seeing a complete example that uses structure points as part of the cost estimation process. Some reuse metrics are defined. Students might be encouraged to compute the reuse metrics for their own software projects.

PROBLEMS AND POINTS TO PONDER

27.1. The best incentive is to reward the creation of reusable components. One way to do this is to (literally) pay developers for components that meet rigid reuse and quality standards and which can be placed in an organization domain library. On the use side, metrics on the amount of reuse achieved by a project team should be maintained. The team is rewarded when a certain percentage of overall delivered source code is reused.

27.2. Many projects are similar. It is possible to use "planning templates" that would enable a project manager to hunt for past similar projects and then use the functional decomposition (with modifications as required), size or function estimates, risks management plans, and accompanying resource estimates, schedule, and SQA/SCM plans. It is important to note that these MUST be modified to meet the needs of the current project, but much of the plan can and should be reused.

27.3. The following task outline describes activities required for the domain analysis of a business application

```
proc domain analysis
     do while further refinement is necessary
     for a particular business domain:
        identify major business objects;
        define major attributes for each object;
        define relationships between objects;
        identify business functions that manipulate
objects;
        for each business function:
           identify process behavior;
              define events/trigger that cause changes
in            behavior;
              define states (modes of behavior);
        endfor;
     identify object/relationship/function tuples
        for each tuple:
           identify characterization functions;
           using characterization functions:
```

```
          identify repeated patterns throughout the
domain;
          translate pattern into basic component
       specification;
       endfor;
    endfor;
    enddo;
endproc;
```

27.4. The intent of both is to provide a quasi-quantitative means for identifying reusable patterns within a system. However, characterization functions are used to identify characteristics of an application domain or product at a customer-visible level. Classification schemes identify characteristics of a software component at a lower level of abstraction.

27.5. Top-level domain characteristics might include: (1) used for identifying student; (2) used for student grades; (3) used for student housing; (3) used for academic scheduling; (4) used for financial aid; (5) used for tuition payments/university fees. Objects would then be characterized for use within these characteristics.

27.8. A structure point is a software architectural pattern (e.g., the architecture of a human interface) that can be found in all applications within a specific domain. Structure points interact with one another and represent architectural building blocks for systems. For example, the transaction structure (discussed in Chapter 14) represented an architectural building block from which many different types of applications can be built.

27.9. Rather than the standard itself, you might assign one of the many Web sites containing relevant information or one of the books noted in *Further Readings and Information Sources.*

27.13. Reuse leverage is defined as:

$R_{lev} = OBJ_{reused}/OBJ_{built}$
where

OBJ_{reused} is the number of object reused in a system
OBJ_{built} is the number of objects build for a system

Therefore, for the stated problem, $R_{lev} = 190/320 = 0.59$

$R_b(S) = [C_{noreuse} - C_{reuse}] / C_{noreuse}$
where

$C_{noreuse}$ is the cost of developing S with no reuse, and
C_{reuse} is the cost of developing S with reuse.

Therefore, $R_b(S) = [1000 - 400]/1000 = 0.60$

Overall cost is cost of building an object from scratch, the cost of adapting an existing object, and the cost of integrating an object with little or no adaptation (note adaptation costs are assumed to include integration cost). Assuming that half of the

```
objects extracted from the library must be adapted,
the overall cost of the system is:

cost  = 130*1000 + 95*600 + 95*400
      = 130,000 + 57,000 + 38,000 = $225,000
```

Chapter 28
Client/Server Software Engineering

CHAPTER OVERVIEW AND COMMENTS

The client/server architecture has dominated computer-based for almost two decades.. From ATM networks to the Internet, from medical diagnostic systems to weapons control, C/S architectures distribute computing capability and add flexibility and power for networked solutions. However, C/S systems also present new challenges for software engineers.

The intent of this chapter is to introduce some of the issues associated with software engineering for C/S systems. The chapter begins with an overview of the structure of C/S systems, describing both the hardware architecture and the software components that can be used during overall system design. The concepts of "middleware" and object request broker architectures are also introduced.

Special considerations for the engineering of C/S software absorb the remainder of the chapter. In the main, the concepts,

principles, and methods presented throughout SEPA are applicable to C/S. But there is often a new 'spin,' and the discussion here tries to relate what it is. A brief discussion of analysis, design, and coding issues is presented. Testing is given a bit more emphasis. Both strategy and tactics are considered. Knowledge of client/server programming techniques is not necessary for students to understand the concepts presented in this chapter.

28.1 The Structure of Client/Server Systems

This section gives examples of several client/server systems that students are likely to have encountered already. The focus of the discussion is the division of functionality between the client and server in each system. The properties of several different software configurations are described. The software components present in all C/S systems are described. Examples of middleware software (CORBA, COM, JavaBeans) were described in the chapter on component-based software design. Some guidelines for deciding how to allocate applications subsystem functions between client and server are also presented.

28.2 Software Engineering for C/S Systems

Many of the process models discussed earlier in the text can be used in C/S software system design. Two approaches are commonly used, an evolutionary approach based on event driven or object-oriented software engineering and component-based software engineering that draws heavily on reusable components. You may need to review the material from Chapter 27 if you have not covered it with your students.

28.3 Analysis Modeling Issues

The analysis modeling methods described earlier in the text can be applied to the modeling of C/S systems. Students should be cautioned that evolutionary development with an eye toward making use of reusable components is the most commonly used implementation method for C/S systems. Students might appreciate seeing the requirements model for a real world C/S system,

28.4 Design for C/S Systems

When software is designed for implementation using client/server architecture the design approach used needs to accommodate several issues specific to C/S design. Data and architectural design will dominate the process and user interface design becomes more important. Behavioral modeling and OO design approaches are often effective as well. The steps for a design approach for C/S systems is presented with fairly complete detail. This section discusses the importance of the design repository and its implementation using relational database technology. Students with weak database backgrounds might benefit from seeing how the design repository was designed and implemented for a real C/S application. The concept of business objects is new and may require supplementary lecture coverage.

28.5 Testing Issues

A general testing strategy for C/S systems is discussed in this section. Some specific tests for C/S systems are suggested. The general strategy is similar that suggested earlier in the text for systems testing. The process should begin by testing individual components and then focus on integration and compatibility testing. Students might be encouraged to develop a test plan for a C/S system.

PROBLEMS AND POINTS TO PONDER

28.1. The best sources of current information are industry periodical such as *Application Development Trends* and/or *Software Development.* A more academic view can be found in refereed journals.

28.2. Any application which requires an extremely powerful compute engine or must manage terabyte-size data structures mitigates toward fat server. Animation systems, weather prediction systems are examples.

28.3. Any application system in which the majority of processing can be accommodated at the user location is a candidate for fat client.

28.4. Visit the OMG web site for additional information.

28.7. Objects for this system would include (one way to describe candidate objects is to use the

grammatical parse approach discussed earlier in the book applied to a statement of scope):

user interaction/presentation:
order form
product description

database:
product inventory
customer mailing list
order log

application:
customer order
customer query
web site
web pages
web forms

28.8. Business rules for a credit card order:

1. Has type of card been specified?
2. Is card number valid?
3. Is expiration date valid?
4. Is name on card same as person to whom shipment will be made?
5. Has a telephone number been provided?
6. Has amount of order been transmitted to the credit card company?
7. Has an authorization number been obtained from the credit card company?

If all answers above questions are "yes" then ship
else contact customer for additional information.

28.9. The key events for the order entry clerk are:

order request—>order entry screen appears
payment method specified—>credit card screen appears
order information completed—>confirmation number appears (i.e., all items ordered are in stock and credit card data has been approved)

Chapter 29
Web Engineering

CHAPTER OVERVIEW AND COMMENTS

This chapter discusses the process of building web applications or WebApps. Many of the principles students have studied for component-based design and object-oriented software engineering are easily adapted to Web engineering (WebE). It is important to impress upon students that WebApps should not just be hacked together or built on the fly. Like all software, the principles of proper analysis, design, quality assurance, and testing apply to WebApps. The nature of the WebE requires some special considerations during some activities. Students often have access to several Web development tools and many students may have their own personal homepages. Having students design a commercial WebApp (or redesign their personal web pages) may be a good activity for them.

29.1 The Attributes of Web-Based Applications

This section discusses the attributes of WebApps and compares them to conventional software applications. A big obstacle to overcome is getting students to see that aesthetics are *not* more important than functionality or usability when designing WebApps. It is important for students to understand that most WebApps are content driven and this is where the greatest effort needs to be applied during their development. Students should study the quality requirements tree (Figure 29.1) and discuss the reasons for including the items that are listed there. It might also be worthwhile to have students expand the tree by

adding new branches or expanding horizontally to provide more detail.

29.2 The WebE Process

This section suggests that WebE needs to be done using an evolutionary, incremental process model. Because of the emphasis on both content and aesthetics, teams having both technical and non-technical members perform WebE.

29.3 A Framework for WebE

A WebE process model is introduced in the section. Students should be given the opportunity to discuss the differences between this model and the evolutionary models discussed in Chapter 2. Note that the activities introduced in this section are expanded in the sections that follow.

29.4 Formulating/Analyzing Web-Based Systems

This section discusses the formulation and analysis of Web-based systems. It might be worthwhile to examine a case study analysis of a real WebApp. If students have some familiarity with Web applications, it may be useful to have them perform content, interaction, functional, and configuration analysis for a WebApp of their own. One way to do this is to assign one group of students to play the role of the "customer" and another group to play the role of WebApp developers for a WebApp that you characterize.

29.5 Design for Web-Based Applications

If students have access to some web development tools (e.g. MS *Frontpage* or Adobe *GoLive*) they might benefit from designing and implementing their own WebApp. Students should focus on understanding the difference between the Web structures used in architectural design. Understanding the attributes of the Web design patterns discussed in this chapter is also important. Even if students do not have access to Web development tools

(other than text editors) it may be worthwhile to have then conduct navigational design and user interface design for a WebApp of their own. Analyzing the strengths and weakness of an existing WebApp is a good way to introduce the importance of attention to user interaction in web design. See the software engineering resource pages at the SEPA, 5/e Web site for additional guidance in this area.

29.6 Testing Web-Based Applications

This section discusses several testing issues that should be considering during WebE. In general, the testing strategies used for component-based and object-oriented software engineering are applicable to Web engineering. Students need to be sensitive to the fact that the volatile nature of WebApps means that testing is an ongoing activity.

29.7 Management Issues

There two major differences between project management for WebE and project management for other types of software. One difference is that WebApps don't often have a long history of measurement to assist in estimation and quantitative quality assessment. And second, there is a tendency by many companies to outsource WebApp development. Students should have an opportunity to discuss the implications of both these situations. Examining the project management plan for a real WebApp might be worthwhile for students. Trying to write a project plan for a WebApp of their own (under typically tight time constraints) is a challenging and worthwhile activity for students.

PROBLEMS AND POINTS TO PONDER

29.1 In addition to the attributes noted at the beginning of this chapter, WebApps have the following specialized characteristics:

Customizable - each user is often given the opportunity of customizing the information presented to his/her won needs

Dependent (interoperable) – the content of the WebApp may be dependent of the content of other WebApps (e.g., hot links to other Web sites)

that are not under the control of the WebApp developer

Responsive – users can "comment" on the WebApp in real-time and expect a timely response to their query/comments

29.2 Obviously, this list is subjective and will vary from person to person. Be absolutely sure, however, that every list includes some of the attributes noted in the quality tree shown in Figure 29.1.

29.4 The key here is to recognize that "content" spans a wide array of media including text, graphics, animation, audio, video. Even within a specific media type (e.g., video), we encounter many different formats and presentation modes.

29.5 For the SEPA, 5/e web site, as an example:

What is the main motivation? The site was developed to assist all those who use SEPA, 5/e in their effort to teach, learn and apply software engineering practices.

Why is the WebApp needed? It helps to supplment content already provided in SEPA, 5/e and assists in the learning process.

Who will use the WebApp? Instructors, students and professionals who use SEPA, 5/e (and some who don't) and want to learn more about software engineering.

Statement of scope: The SEPA, 5/e Web site is designed to provide supplementary information for readers of SEPA, 5/e.

29.6 SafeHome user profiles:

Homeowner – non-technical end-user of the SafeHome system.

Technician – technical user who will install and configure the system for the homeowner

29.7 In addition to the goals noted on p. 777, we might add:

Informational:

The site will provide a sales argument for why a security system is necessary.

The site will provide general home security information/articles that may be of use to the homeowner.

The site will provide guidelines for comparing and selecting home security systems.

Applicative:

The site will acquire relevant information (e.g., address, phone number)about each visitor.

The site will provide user guided tutorials on SafeHome Installation

29.9 Content analysis examines the specific information that will be presented as part of the WebApp. Interaction and functional analysis address the manner in which users interact with the WebApp and the functions to be performed at/by the WebApp as the user navigated the software.

29.11Golden rules:

1. The architecture should always be designed to accommodate the content to be presented and the navigation that will likely occur.

2. The user should never become lost while navigating the site.

3. Interaction should always be intuitive and should rely on a metaphor that is appropriate for the goals of the WebApp.

29.12A template is the implementation realization of a design pattern. That is, the template provides the skeleton that an implementer would use to construct the pattern within a WebApp.

29.18Most of the basic principles of project management remain the same. However, because of the immediacy of WebApps, project monitoring, tracking and control must be conducted at a higher level of granularity. For example, the project schedule must be defined in half-day increments; milestones must be closely spaced; and progress must be assessed much more frequently.

Chapter 30
Reengineering

CHAPTER OVERVIEW AND COMMENTS

Reengineering became 'corporate chic' during the 1990s. At the corporate level, entire business processes were reengineered to make them more efficient and more competitive (at least in theory!). These changes had a trickle down effect on information systems. As business processes changed, the information technology that supported them also had to change.

The intent of this chapter is to 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.

The software reengineering process involves inventory analysis, document restructuring, reverse engineering, program restructuring, data restructuring, and forward engineering. Students need to be reminded that standard SQA and testing practices are applied through out the software reengineering process. Having students reengineer the user interface or perform data restructuring for an existing program may be a worthwhile project assignment.

30.1 Business Process Reengineering

This section discusses business process reengineering to provide students with some background on why working processes may need to be revised. Software reengineering takes place at the lowest levels of BPR. The BPR principles listed in Section 30.1.2 have general applicability to SQA and software project management work. Many students will not have the experience to appreciate the subtleties implied by BPR. Therefore, they should have the opportunity to discuss BPR in class and examine their relationship to software development. Students

should examine the BPR model described in Section 30.1.3 and discuss how it may be used as part of workflow analysis.

30.2 Software Reengineering

Often, students have no appreciation of the overwhelming burden that software maintenance places on many IT organizations. This should be discussed at length. Students should work to understand the differences between software maintenance and software reengineering. Class discussion of case studies involving reengineering may help students to understand when reengineering is justified and when partial reengineering may need to be used instead. A software reengineering process model is presented. Students should look at the inventory analysis checklist on the SEPA web site. Showing students examples of code restructuring and data restructuring may be helpful. Forward engineering will be discussed in more detail later in the chapter.

30.3 Reverse Engineering

This section discusses reverse engineering of process, data, and user interfaces. It may be worthwhile to have students try to create an analysis or design model from a program's source code. A good way to illustrate the difficulty of reverse engineering (and by implication, software maintenance in general) is to provide students with two small programs of equal complexity. The first program should be implemented without documentation and in a sloppy manner. The second should be well documented, structured, and have meaningful variable names. Ask the students to reverse engineer each, and then ask (1) which took less time to understand, (2) which was easier to reverse engineer, and (3) which assessment they have more confidence in.

30.4 Restructuring

This section provides a brief discussion of code restructuring and data restructuring. Students may benefit from seeing case

studies in which restructuring code or data was performed to improve software quality. It is important for students to see that data and architecture must be restructured before any benefit is achieved. If time permits (and you are so inclined, an old book by J.D. Warnier *(Logical Construction of Programs,* Van Nostrand-Reinhold, 1974, out of print) has an interesting restructuring algorithm that may be presented or used as a case study.

30.5 Forward Engineering

Forward engineering is a tough concept for students to understand. Forward engineering is a little bit like preventative maintenance with the addition of the use of tools to automate parts of the process. Getting students to appreciate the arguments for reengineering a working applications is a major goal for this section. Many of the arguments in favor of reengineering are based on reducing the costs maintaining software. Part of the motivation for reengineering working applications comes for the growing popularity of new technologies like client/server or OO systems.

30.6 The Economics of Reengineering

This section presents a short discussion on a model that may be used to describe the cost benefits realized from reengineering. The mathematics is not complicated and students would benefit from seeing the benefits reaped by reengineering a real application.

PROBLEMS AND POINTS TO PONDER

30.1. This is a useful problem. Begin by having your students create a quite work flow for the existing job. A simple outline of tasks and work products should suffice. Next, have them isolate where in the work flow efficiencies can be obtained. Then have them recommend changes to the process that will achieve these efficiencies. If computing technologies will provide benefit, have them specify where they are most applicable.

30.2. The positive arguments for BPR are contains in books such as [HAM93]. On the negative side, many

companies give only half-hearted support to BPR activities, resulting in upheaval, but little benefit. Some management teams reorganize people, but don't change process. This rarely achieves significant benefits and almost always creates upheaval that lowers productivity. The Internet contains much information on this subject.

30.3. I strongly recommend that you assign this exercise during every software engineering course. It will teach more about maintenance (and the problems associated with doing it) than any amount of lecture.

30.4. There are any number of ways to do this. The best approach is to create a spreadsheet model with a column for each of the criteria noted in the checklist. Each of the quantitative criteria can then be normalized by dividing it by the average value for the characteristic for all programs. The normalized value can then be multiplied by a weighting factor (0.0 to 1.0) based on the relative impact of the characteristic on the reengineering decision (e.g., overall maintainability may have a high impact, while the annual cost of operation may have a somewhat lower impact on the reengineering decision). The resultant grade provides a guide for selecting programs for reengineering.

30.5. One example, capturing the developer's thoughts on video and embedding quick-time movies directly into the source listing.

30.7. As the abstraction level goes up, the description of the software moves further and further away from implementation details. Therefore, it becomes hard to trace from the model or document being assessed back to code level specification (unless there is a clear and repeatable mapping).

30.8. A human can make the "leaps" necessary to bridge the gap between higher levels of abstraction and implementation detail. Therefore, high interactivity is essential if completeness is to increase.

30.10. Restructuring tends to make only minor design changes. Reengineering completely rebuilds the application, making major design modifications. For example, a reengineering activity might implement an OO solution when the original program was developed using a conventional paradigm.

30.12. The best way is to examine applications that have already been reengineered and collect metrics that will provide average values for P4 to P7. If such data do not exist, the values must be estimated based on projected percent improvements.

Chapter 31
Computer-Aided Software Engineering

CHAPTER OVERVIEW AND COMMENTS

The intent of this chapter is to provide a generic overview of CASE. Because the software tools domain is changing so rapidly, I have attempted to write this chapter in a way that

will cause it to remain meaningful, even after this generation of CASE tools is obsolete.

The chapter begins with a brief discussion of the history of software tools and the overall philosophy of CASE, and a presentation of the overall CASE building blocks. A CASE tools taxonomy is presented, and an overview of CASE environments is discussed. Finally, the role and structure of the CASE repository (referenced in Chapter 9 and elsewhere in SEPA) is presented. If you plan to use CASE tools as part of your course, this chapter should be covered early and you will probably need two semesters to complete the entire text. It may be worthwhile to have students locate CASE tools and evaluate their capabilities in light of the features of integrated CASE (I-CASE) environments described in this chapter.

31.1 What is CASE?

This section provides a brief discussion of CASE and its overall objectives. It might be a good idea to discuss how CASE has failed to achieve its promise by noting the hype during the 1980s and early 90s followed by the disappointments of the later 1990s. This discussion, however, should emphasize that the role of automation in software engineering is a key to progress in the discipline.

31.2 Building Blocks for CASE

This section describes the CASE environment. The bottom layers (environment, hardware, operating system) would be found in most application support models. The integration framework and portability layers are particularly important to I-CASE and students should be sure that they understand their importance to CASE tool use. Presenting the architecture of a real CASE tool may be beneficial. Tools vendors can be contacted via the Web. See the Software Engineering resources section of the SEPA, 5/e Web site for hot links.

31.3 A Taxonomy of CASE Tools

Several categories of CASE tools are described in this section. CASE tools do not have to be part of an integrated environment to be useful to software engineers, but their impact on product quality will greater if they are. Students might be encouraged to locate representative tools for each category listed (many were discussed earlier in the text). Examining the functionality present in existing CASE tools may be worthwhile (no CASE tool supports every function listed in this section).

31.4 Integrated CASE Environments

The benefits derived from I-CASE are discussed briefly in this section. The additional challenges caused by seeking to integrate all tools, work products, and design information are also described.

31.5 The Integration Architecture

An I-CASE integration architecture model is presented in this section. It is important that students become familiar with the role of each model layer. Of particular importance are the object management layer (OML) and the shared repository layer. The OML is used to support the configuration management function of the I-CASE environment. The shared repository layer consists of the CASE database and the access functions used by OML to interact with it. It may be worthwhile to compare the integration architecture model presented in this section with one found in a real I-CASE environment.

31.6 The CASE Repository

This section discusses the CASE repository. The similarities and differences between the CASE repository and an ordinary database management system are also discussed. The contents of the CASE repository for a real I-CASE system might be discussed with students.

PROBLEMS AND POINTS TO PONDER

31.1. Even though a student list may be short, you should insist that students consider the tool box that they apply to "write programs." For industry practitioners: the list can number dozens of tools — some used globally and some used local to one group. Try to categorize.

31.2. The key here is information hiding and decoupling. That is, the portability services should be designed in a way that hides operation system details from the CASE environment and decouples and CASE related tools from the OS.

31.6. There are many situations in which dynamic analysis tools for software testing are "the only way to go." For example, if an embedded application is to be build and performance of various software functions is crucially important, a dynamic tool can provide an important window into the system operation. Another situation: an interactive system is to build and heavy testing of the interface is mandatory. In addition, it is likely that many changes to the system will occur over the first year of operation, demanding heavy regression testing with each change. A capture/playback tool can reduce the effort required to conduct regression tests by a substantial percentage.

31.7. An example that jumps to mind is medical instrumentation and information management. By collecting, collating, relating and reporting the results from many tests on the same patient, a doctor has a better chance of deriving an accurate diagnosis. Another example is automated manufacturing. Machines and manufacturing cells are integrated so that materials flow more smoothly among them.

31.8. In essence, data-tool integration encapsulates (in an object-oriented sense) configuration items and the tools (functions) that operate on them.

31.10 Metamodel and metadata are terms that describe a model or data that is used to describe a model or data. For example, metadata is used to describe the attributes of a configuration item such as a design specification. The generic contents are described with metadata. The information contained within the specification is data.

Chapter 32
The Road Ahead

CHAPTER OVERVIEW AND COMMENTS

The intent of this chapter is to provide a peek into the future of software engineering practice and to summarize some of the significant themes that have been discussed earlier in the text. Like any attempt at crystal ball gazing, my projections may be off the mark, but the general trends outlined in this final chapter of SEPA are highly likely to emerge as we move into the 21st century.

32.1 The Importance of Software—Revisited

This section reviews the comments on the changing of role software that were first discussed in Chapter 1. Software is part of many daily activities and many objects used in every day life. If time permits, a discussion of the student's view of the impact of software may be worthwhile. What are the dangers? What are the opportunities? Young people have not known a world without pervasive use of software. They have an interesting perspective on its importance.

32.2 The Scope of Change

A general perspective on change is presented in this section. The most controversial point, is the suggestion that revolutionary advances in computing will be driven more by the soft sciences, rather than by the hard sciences. Students might be encouraged to discuss whether or not this seems to be true.

32.3 People and the Way They Build Systems

This section discusses the trends that seem to be present in the way systems are built. They are bigger, more complex, and involve more people in their development than we have seen in the past. Maintaining current productivity levels will be difficult if more effective means of communication are not used to share information between developers. This may involve increased reliance on CASE, groupware, and video

conferencing, or simply greater reliance on e-mail and Web technologies. Artificial intelligence advances may give rise to more intelligent software tools or more efficient knowledge acquisition. Students should be cautioned that AI has extremely long delivery times, so a revolution may not be right around the corner.

32.4 The "New" Software Engineering Process

The trend toward evolutionary process models and OO or component-based design may seem quite reasonable to your students. As long as development time lines remain short and the economics of reusing existing software seems reasonable, these processes are likely to dominate the behavior of many software engineers. The issue to consider it how to maintain quality and avoid cutting critical corners to try to meet very tighter project deadlines. Guidelines like the SEI Capability Maturity Model have the promise of holding organizations accountable to stringent quality guidelines, but to be frank, they are simply unrealistic in some development environments. The World Wide Web has changed the ways in which many software applications are developed and are deployed. Students might be challenged to look for other trends (e.g. government mandated software policies).

32.5 New Modes for Representing Information

This section talks about the differences between data, information, knowledge, and wisdom. Students should have an opportunity to discuss the differences between each. Knowledge bases are beginning to replace databases and information systems in some application domains. If your students have not studied artificial intelligence, it may be worthwhile to look at a system containing an embedded neural network or expert (rule-based) system. Likewise, some discussion of knowledge discovery techniques in data warehouses may be worthwhile.

32.6 Technology as a Driver

There is some evidence that suggests that hardware technology will proceed along two parallel paths, rapid advance of traditional VonNeuman architectures and the maturing of non-traditional architectures (e.g. massively parallel systems). Developing software for massively parallel systems is a very difficult task and one in which not many software engineers have much experience. If reuse and component-based technologies continue to mature, software development technologies may begin to look like today's hardware development. Vendors may be building and selling discrete reusable software components, rather than complete systems. Your students should discuss the obstacles to this goal.

PROBLEMS AND POINTS TO PONDER

30.1. *The Wall Street Journal* (try also the interactive WSJ at **http://www.wsj.com**) is an excellent source of articles of this type. Your student should have no trouble finding dozens of stories.

30.5. The Internet has captured the imagination of most educated people worldwide. I believe an excellent case study would be to show how raw data contained at a Web site can be synthesized into useful information and then to knowledge about some topic. Try to develop an Internet example in which wisdom can be extracted from the Web.