ANNAMACHARYA UNIVERSITY, RAJAMPET

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016
RAJAMPET, Annamayya District, AP, INDIA

Course : Software Engineering

Course Code : **24FMCA23T**

Branch : **MCA**

Prepared by : P. Kavitha

Designation : **Assistant Professor**

Department : **MCA**

Title of the Course : Software Engineering
Category : PC
Course Code : 24FMCA23T
Branch : MCA
Semester : II Semester

Lecture Hours	Tutorial Hours	Practice Hours	Credits
3	0	0	3

COURSE OBJECTIVES:

- To Apply software engineering principles to real world scenarios and projects.
- To Enable students to understand and document both functional and non-functional requirements various system modeling techniques effectively.
- To Develop skills in making critical architectural decisions that influence the overall system structure.
- To Apply verification, validation, and testing concepts to real-world software projects through hands-on exercises and projects.
- To Apply project management, planning, scheduling, risk management, and cost estimation concepts to real-world software projects.

- **COURSE OUTCOMES:**
- **The Student will be able to**
- 1. Comprehend the basic terminologies associated with software engineering.
- 2. Describe various process models for developing software.
- 3. Apply fundamental design concepts to create software solutions that are modular, scalable, and maintainable.
- 4. Apply various testing strategies for software quality assurance.
- 5. Apply project management and software cost estimation skills to real-world scenarios, considering industry best practices.

UNIT I **10**

SOFTWARE, SOFTWARE ENGINEERING, AND PROCESS: The nature of Software, The unique nature of WebApps, Software engineering- A layered technology, General principles of software engineering practice, Software myths, Agile development: What is an Agile Process?, Capability Maturity Model Integration (CMMI).

UNIT II **12**

PROCESS MODELS: A Generic process model (framework), Process assessment and improvement, Prescriptive process models: The waterfall model, Incremental process models, Evolutionary process models, The Unified process, **SOFTWARE REQUIREMENTS**: Functional and Non-functional requirements; User requirements, The software requirements document, **Requirements Engineering Processes**: Requirements elicitation and analysis, Requirements validation Requirements management, **System Modeling**: Context models, Behavioral models, Data models, Object models, Structured Methods.

UNIT III **8**

DESIGN CONCEPTS: Design Concepts, **ARCHITECTURAL DESIGN**: Architectural design decisions, System organization, Modular decomposition styles.

OBJECT-ORIENTED DESIGN: Objects and Object Classes, An Object-Oriented design process, Design Evolution.

UNIT IV **10**

VERIFICATION AND VALIDATION: Planning verification and validation, Software inspections, Automated static analysis, Verification and formal methods.

SOFTWARE TESTING: System testing, Component testing, Test case design, Test automation, **Quality management:** Software Quality Assurance.

UNIT V **8**

PROJECT MANAGEMENT: Management activities, Project planning, Project scheduling, Risk management. **SOFTWARE COST ESTIMATION:** Software Productivity, Estimation techniques- The COCOMO II Model, Project duration and staffing.

PRESCRIBED TEXTBOOKS:

1. Roger S. Pressman. Software Engineering - A Practitioners approach, McGraw-Hill, 9th Edition 2020.
2. Ian Sommerville, Software Engineering, Pearson Education Publications, 10th Edition 2017.

REFERENCE BOOKS:

1. Shari Lawrence Pfleeger, Joanne M. Atlee, Software Engineering Theory and Practice, Pearson Education, 4th Edition 2010.
2. Waman S Jawadekar. Software Engineering Principles and Practice, Tata McGraw Hill, 2012.

CO-PO MAPPING:

Course Outcomes	Foundation Knowledge	Problem Analysis	Development of Solutions	Modern Tool Usage	Individual and Teamwork	Project Management and Finance	Ethics	Life-long Learning
24FMCA23T.1	2	2	1	-	-	-	-	-
24FMCA23T.2	2	2	1	-	-	-	-	-
24FMCA23T.3	3	2	1	-	-	-	-	-
24FMCA23T.4	3	2	1	-	-	-	-	-
24FMCA23T.5	3	2	1	-	-	-	-	-

UNIT-I

SOFTWARE, SOFTWARE ENGINEERING, AND PROCESS: The nature of Software, The unique nature of WebApps, Software engineering- A layered technology, General principles of software engineering practice, Software myths, Agile development: What is an Agile Process?, Capability Maturity Model Integration (CMMI).

1.1 The Nature of Software

- Today, software takes on a dual role. It is a product, and at the same time, the vehicle for delivering a product .Whether it resides within a mobile phone or operates inside a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources.

- As the vehicle is used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments). Software delivers the most important product of our time—*information.*
- The role of computer software has undergone significant change over the last half-century. The questions that were asked of the lone programmer are the same questions that are asked when modern computer-based systems are built:

• Why does it take so long to get software finished?

• Why are development costs so high?

• Why can't we find all errors before we give the software to our customers?

• Why do we spend so much time and effort maintaining existing programs?

• Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

- **1.1.1 Defining Software**
- **1.1.2 Software Application Domains**
- **1.1.3 Legacy Software**

1.1.1 Defining Software

- ***Software is:***

  (1) instructions (computer programs) that when executed provide desired features, function, and performance;

  (2) data structures that enable the programs to adequately manipulate information, and

  (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

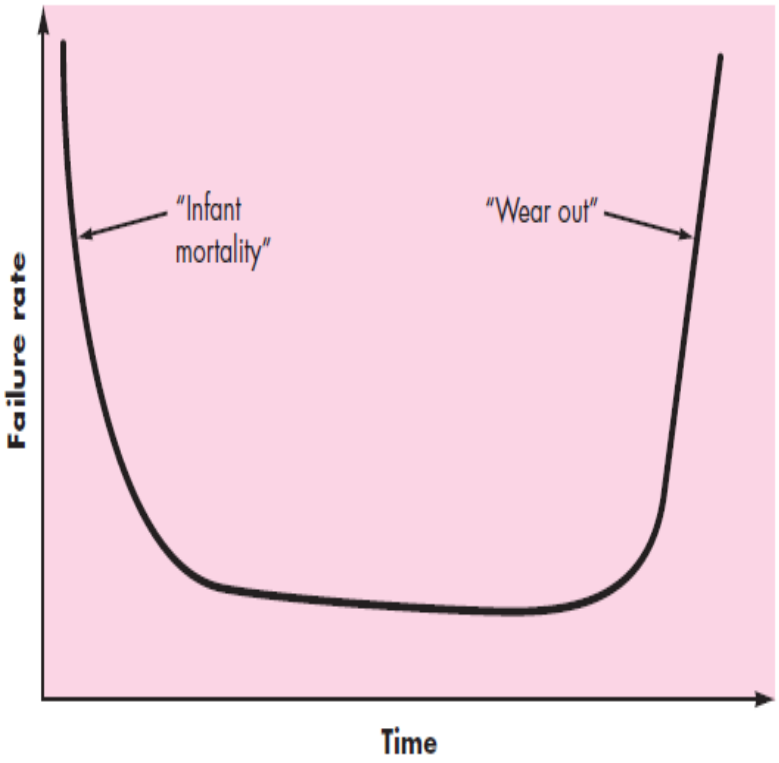
  - It's important to examine the characteristics of software that make it different from other things that human beings build.
- Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1. ***Software is developed or engineered; it is not manufactured in the classical sense.***

- Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different.
- In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are non-existent (or easily corrected) for software.
- Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different
- Both activities require the construction of a "product," but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

FIGURE 1.1

Failure curve for hardware



2. ***Software doesn't "wear out."***

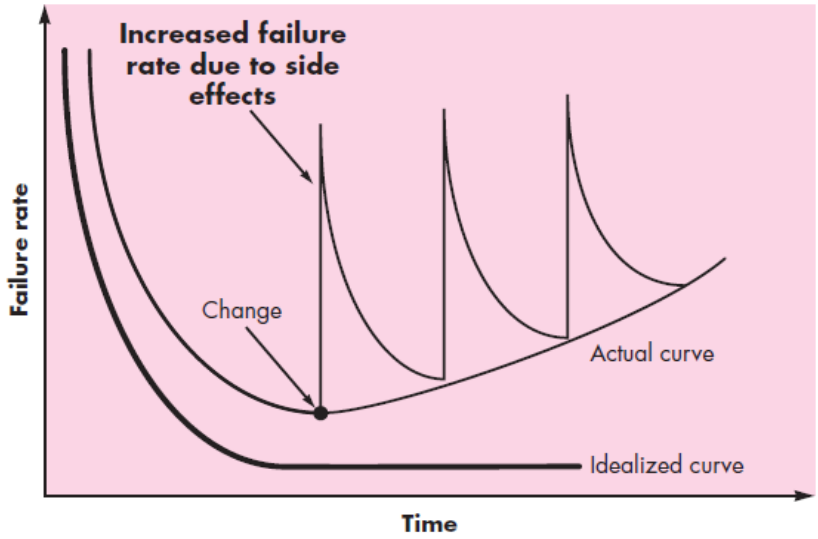
- Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life ,defects are corrected and the failure rate drops to a steady-state level for some period of time.

- As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to ***wear out.***

- Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of ***the "idealized curve"*** shown in Figure 1.2.

- Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of ***the "idealized curve"*** shown in Figure 1.2.

FIGURE 1.2
Failure curves for software



- However, these are corrected and the curve flattens as shown. The idealized curve is a gross over-simplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does ***deteriorate!***
- This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its life, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve" (Figure 1.2).
- Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.
- Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts.
- Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

2. ***Although the industry is moving toward component-based construction, most software continues to be custom built.***

- As an engineering discipline evolves, a collection of standard design components is created.
- The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new.
- In the ***hardware world,*** component reuse is a natural part of the engineering process. In the ***software world***, it is something that has only begun to be achieved on a broad scale.
- A software component should be designed and implemented so that it can be reused in many different programs.
- Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts.

1.1.2 Software Application Domains

Today, seven broad categories of computer software present continuing challenges for software engineers:

- **System Software**
- **Application Software**
- **Engineering/Scientific Software**
- **Embedded Software**
- **Product-line Software**
- **Web Applications**
- **Artificial Intelligence Software**
- **Open-world Computing**
- **NetSourcing**
- **Open Source**
- **System Software**—a collection of programs written to service other programs.

- Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate, information structures.
- Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.
- **Application software**—stand-alone programs that solve a specific business need.
- Applications in this area process business or technical data in a way that facilitates business operations or management/ technical decision making.
- **Engineering/Scientific Software**—has been characterized by "number crunching" algorithms.
- Applications range from astronomy (study of space) to volcanology(study of geology, geophysics, geochemistry), from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.
- Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.
- **Embedded software**—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself.
- Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant function and control capability.
- **Product-line Software**—
- designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

- **Web Applications**—called "WebApps," this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics.
- However, as Web 2.0 emerges, WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.
- **Artificial Intelligence Software**—makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis.
- Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.
- **Open-World Computing**—
- The rapid growth of wireless networking may soon lead to true pervasive, distributed computing.
- The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.
- **NetSourcing**—
- The World Wide Web is rapidly becoming a computing engine as well as a content provider.
- The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.
- **Open Source**—
- A growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many people can contribute to its development.

- The challenge for software engineers is to build source code that is self-descriptive, but more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

1.1.3 Legacy Software

- Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection.
- Some of these are state of- the-art software—just released to individuals, industry, and government.
- But other programs are older, in some cases *much* older. These older programs—often referred to as ***legacy software***—have been the focus of continuous attention and concern since the 1960s.
- Dayani-Fard and his colleagues describe legacy software in the following way:
- ***Legacy software systems*** . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms.
- Unfortunately, there is sometimes one additional characteristic that is present in legacy software—***poor quality*.**
- The only reasonable answer may be: ***Do nothing***, at least until the legacy system must undergo some significant change.
- If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons:

• The software must be adapted to meet the needs of new computing environments or technology.

• The software must be enhanced to implement new business requirements.

• The software must be extended to make it interoperable with other more modern systems or databases.

• The software must be re-architected to make it viable within a network environment. When

these modes of evolution occur, a legacy system must be re-engineered so that it remains viable into the future.

1.2 The Unique Nature of WebApps

- Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.
- WebApps are one of a number of distinct software categories. And yet, it can be argued that WebApps are different.
- Powell [Pow98] suggests that Web-based systems and applications *"involve a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology."*
- The following attributes are encountered in the

- **Network intensiveness.** A WebApp resides on a network and must serve the needs of a diverse community of clients.
  - The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).
- **Concurrency.** A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.
- **Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.
- **Performance.** If a WebApp user must wait too long (for access, for server-side processing, for client-side formatting and display), he or she may decide to go elsewhere.
- **Availability.** Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis. Users in Australia or Asia might demand access during times when traditional domestic software applications in North America might be taken off-line for maintenance.

- **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

- **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.

    - **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously.

    - It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

- **Immediacy.** Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.7

- **Security.** Because WebApps are available via network access, it is difficult,if not impossible, to limit the population of end users who may access the application.

  In order to protect sensitive content and provide secure modesof data transmission, strong security measures must be implemented throughout the infrastructure that supports a WebApp and within the application itself.
- **Aesthetics.** An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

    - It can be argued that other application categories discussed in Section 1.1.2 can exhibit some of the attributes noted. However, WebApps almost always exhibit all of them.

1.3 Software Engineering

• Software has become deeply embedded in virtually every aspect of our lives***, It follows that a concerted effort should be made to understand the problem before a software solution is developed.***

• The information technology requirements demanded by individuals, businesses, and governments grow increasing complex with each passing year. Large teams of people now create computer programs that were once built by a single individual. ***It follows that design becomes a pivotal activity.***

• Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures. ***It follows that software should exhibit high quality.***

- • As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow. As its user base and time-in-use increase, demands for adaptation and enhancement will also grow. *It follows that software should be maintainable.*

- These simple realities lead to one conclusion: *software in all of its forms and across all of its application domains should be engineered.* And that leads us to the topic of this book—*software engineering.*

- The IEEE (Institute of Electrical and Electronics Engineers,) has developed a more comprehensive definition when it states:

- **Software Engineering: (**1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

- And yet, a "systematic, disciplined, and quantifiable" approach applied by one software team may be burdensome to another.

- Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality.

- Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus.

- The foundation for software engineering is the ***process*** layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software

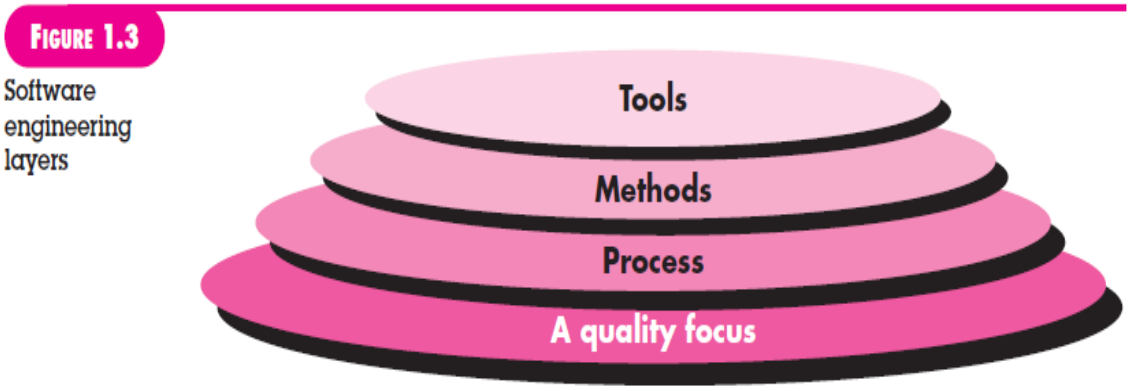


FIGURE 1.3 Software engineering layers

Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

- Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support.

- Software engineering *tools* provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering*, is established.

1.4 **The Software Process**

- A ***process*** is a collection of activities, actions, and tasks that are performed when some work product is to be created.

- An ***activity*** strives to achieve a broad objective

(e.g., communication with stakeholders) and is applied regardless of the application domain, size

of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.

An ***action*** (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).

A ***task*** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

- In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

- A ***process framework*** establishes the foundation for a complete software engineering process by identifying a small number of *framework activities* that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of *umbrella activities* that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

- **Communication.**

- **Planning**

- **Modeling.**

- **Construction**

- **Deployment**

- These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems.

- For many software projects, framework activities are applied iteratively as a project progresses. That is, **communication, planning, modeling, construction,**and **deployment** are applied repeatedly through a number of project iterations.

- Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete. Software engineering process framework activities are complemented by a number of *umbrella activities.*

- **Typical umbrella activities include:**

- **Software project tracking and control**—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

- **Risk management**—assesses risks that may affect the outcome of the project or the quality of the product.

- **Software quality assurance**—defines and conducts the activities required to ensure software quality.

- **Technical reviews**—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

- **Measurement**—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

- **Software configuration management**—manages the effects of change throughout the software process.

- **Reusability management**—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

- **Work product preparation and production**—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

- ***Prescriptive process models***

Stress detailed definition, identification, and application of process activities and tasks. Their intent is to improve system quality, make projects more manageable, make delivery dates and costs more predictable, and guide teams of software engineers as they perform the work

required to build a system.

- ***Agile process models*** emphasize project "agility" and follow a set of principles that lead to a more informal (but, proponents argue, no less effective) approach to software process. These process models are generally characterized as "agile" because they emphasize maneuverability and adaptability.

1.5 **Software Engineering Practice**

1.5.1 The Essence of Practice

In a classic book, *How to Solve It,* written before modern computers existed, George Polya [Pol45] outlined the essence of problem solving, and consequently, the essence of software engineering practice:

1. *Understand the problem* (communication and analysis).

2. *Plan a solution* (modeling and software design).

3. *Carry out the plan* (code generation).

4. *Examine the result for accuracy* (testing and quality assurance).

- Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

• *Who has a stake in the solution to the problem?* That is, who are the stakeholders?

• *What are the unknowns?* What data, functions, and features are required to properly solve the problem?

• *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?

• *Can the problem be represented graphically?* Can an analysis model be created?

- **Plan the solution.** Now you understand the problem (or so you think) and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

• *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?

• *Has a similar problem been solved?* If so, are elements of the solution reusable?

• *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?

• *Can you represent a solution in a manner that leads to effective implementation?*

- Can a design model be created?

- **Carry out the plan.** The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

• *Does the solution conform to the plan?* Is source code traceable to the design model?
• *Is each component part of the solution provably correct?* Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

- **Examine the result.** You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

• *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
• *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

- The dictionary defines the word *principle* as "an important underlying law or assumption required in a system of thought." Regardless of their level of focus, principles help you establish a mind-set for solid software engineering practice. They are important for that reason.

- **David Hooker** has proposed seven principles that focus on software engineering practice as a whole

- **The First Principle: *The Reason It All Exists***

- A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is "no," don't do it. All other principles support this one.

- **The Second Principle: *KISS (Keep It Simple, Stupid!)***

There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. Simple also does not mean "quick and dirty." In fact, it often takes a lot

of thought and work over multiple iterations to simplify.

- **The Third Principle: *Maintain the Vision***

- *A clear vision is essential to the success of a software project*. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself.

- Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

- **The Fourth Principle: *What You Produce, Others Will Consume***

- In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system.

- So, *always specify, design, and implement knowing someone else will have to understand what you are doing*. Design, keeping the implementers in mind.

- **The Fifth Principle: *Be Open to the Future***

- A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years.

  - *Never design yourself into a corner*. Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific

  - **The Sixth Principle: *Plan Ahead for Reuse***

  - Reuse saves time and effort. There are many techniques to realize reuse at every level of the system development process. . . . *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated*.

The Seventh principle: *Think!*

This last principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results*. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of

thinking is learning to recognize when you don't know something, at which point you can research the answer.

1.6 **Software Myths (Common Beliefs)**

- Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious (**something unpleasant or dangerous**.)

  - Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike.

  - **Management myths.**

  - **Customer myths.**

  - **Practitioner's myths.**

  - **Management myths**

- **Myth:** *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

  Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

- **Myth:** *If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).*

  Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive

development effort. People can be added but only in a planned and well coordinated manner.

- **Myth:** *If I decide to outsource the software project to a third party, I can just relax and let that firm build it.*

- **Reality:** If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

- **Customer myths**

- **Myth:** *A general statement of objectives is sufficient to begin writing*

programs—we can fill in the details later.

- **Reality:** Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and continuous communication between customer and developer.

Myth: *Software requirements continually change, but change can be easily accommodated because software is flexible.*

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small.16 However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

- **Practitioner's myths**

- **Myth:** *Once we write the program and get it to work, our job is done.*

Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

- **Myth:** *Until I get the program "running" I have no way of assessing its quality.*

Reality: One of the most effective software quality assurance mechanisms can be applied from

the inception of a project—*the technical review.* Software reviews (described in Chapter 15) are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

- **Myth:** *The only deliverable work product for a successful project is the working program.*

Reality: A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Myth: *Software engineering will make us create voluminous (capacious,spacious) and unnecessary documentation and will invariably slow us down.*

Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times. Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

1.7 What is Agility?

1.7.1 Agility in context of software engineering

- Agility means effective (rapid and adaptive) response to change, effective communication among all stockholder.
- Drawing the customer onto team and organizing a team so that it is in control of work performed. -The Agile process, light-weight methods are People-based rather than plan-based methods.
- The agile process forces the development team to focus on software itself rather than design and documentation.
- The agile process believes in iterative method.
- The aim of agile process is to deliver the working model of software quickly to the customer For example: Extreme programming is the best known of agile process.

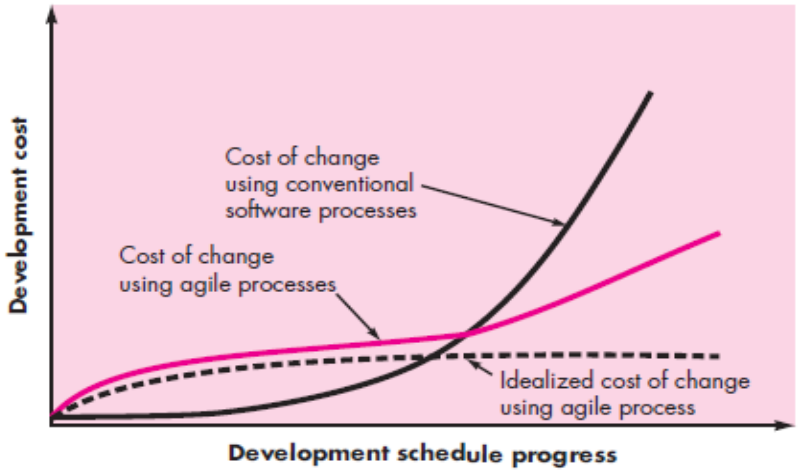
An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project. Agility can be applied to any software process

1.7.2 Agility And The Cost Of Change

- The conventional wisdom in software development (supported by decades of experience) is that the cost of change increases nonlinearly as a project progresses

- (Figure 3.1, solid black curve). It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project). A usage scenario might have to be modified, a list of functions may be extended, or a written specification can be edited. The costs of doing this work are minimal, and the time required will not adversely affect the outcome of the project.

FIGURE 3.1
Change costs as a function of time in development



uote:
"Agility is dynamic, content specific, aggressively change embracing

- Proponents of agility argue that a well-designed agile process "flattens" the cost of change curve (Figure 3.1, shaded, solid curve), allowing a software team to accommodate changes late in a software project without dramatic cost and time impact.

1.7.3 What is Agile Process?

- Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects:

- **1.** It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
- **2.** For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
- **3.** Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Given these three assumptions, an important question arises: How do we create a process that can manage *unpredictability?* The answer, as I have already noted, lies in process adaptability (to rapidly changing project and technical conditions). An agile process, therefore, must be *adaptable.*

- But continual adaptation without forward progress accomplishes little. Therefore an agile software process must adapt *incrementally.* To accomplish incremental adaptation, an agile team requires customer feedback . Hence, an *incremental development strategy* should be instituted. *Software increments* must be delivered in short time periods so that adaptation keeps pace with change (unpredictability).
- 1.7.3.1 **Agility Principles**
- The Agile Alliance defines 12 agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they

need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7. Working software is the primary measure of progress.

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9. Continuous attention to technical excellence and good design enhances agility.

10. Simplicity—the art of maximizing the amount of work not done—is essential.

11. The best architectures, requirements, and designs emerge from self– organizing teams.

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

- **1.7.3.2 The Politics of Agile Development**

- There is considerable debate (sometimes strident) about the benefits and applicability of agile software development as opposed to more conventional software engineering processes. Jim Highsmith [Hig02a] (facetiously) states the extremes when he characterizes the feeling of the pro-agility camp ("agilists").

- No one is against agility. The real question is: What is the best way to achieve it? As important, how do you build software that meets customers' needs today and exhibits the quality characteristics that will enable it to be extended and scaled to meet customers' needs over the long term?

There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed process models (Section 3.4), each with a subtly different approach to the agility problem. Within each model there is a set of "ideas" (agilists are loath to call them "work tasks") that represent a significant departure from traditional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts. Bottom line: there is much that can be gained by considering the best of both schools and virtually nothing to be gained by denigrating either approach.

- **1.7.3.3 Human Factors**

- Proponents of agile software development take great pains to emphasize the importance of "people factors." As Cockburn and Highsmith state, "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that *the process molds to the needs of the people and team,* not the other way around

- **A number of key traits(** distinguishing quality or characteristics)

- **1)Competence.** In an agile development (as well as software engineering) context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply.

- **2) Common focus.** Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.
- **3)Collaboration.** Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.
- **4)Decision-making ability.** Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.
- **5)Fuzzy problem-solving ability.** Software managers must recognize that the agile team will continually have to deal with ambiguity (inexactness) and will continually be buffeted (strike repeatedly)by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However,

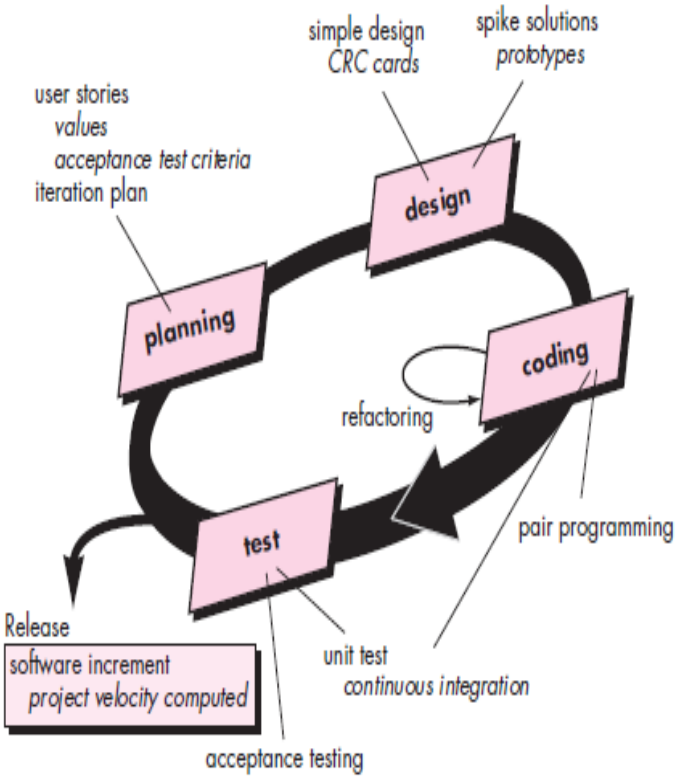
lessons learned from any problem-solving activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

- **6)Mutual trust and respect.** The agile team must become what DeMarco and Lister [DeM98] call a "jelled" team (Chapter 24). A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts." [DeM98]
- **7)Self-organization.** In the context of agile development, self-organization implies three things:
- the agile team organizes itself for the work to be done,
- the team organizes the process to best accommodate its local environment,
- the team organizes the work schedule to best achieve delivery of the software increment.
- Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale. In essence, the team serves as its own management. Ken Schwaber addresses these issues when he writes: "The team selects how much work it believes it can perform within the iteration, and the team commits to the work.
- **1.7.3.4 EXTREME PROGRAMMING (XP)**
- *Extreme Programming* (XP), the most widely used approach to agile software development.
- More recently, a variant of XP, called *Industrial XP* (IXP) has been proposed . IXP refines XP and targets the agile process specifically for use within large organizations.
- ***1.7.3.4.1 XP Values***
- Beck defines a set of five *values* that establish a foundation for all work performed as part of **XP—communication, simplicity, feedback, courage, and respect**. Each of these values is used as a driver for specific XP activities, actions, and tasks.
- In order to achieve effective ***communication*** between software engineers and other stakeholders (e.g., to establish required features and functions for the software), XP emphasizes close, yet informal (verbal) collaboration between customers and developers.
- To achieve ***simplicity***, XP restricts developers to design only for immediate needs, rather than consider future needs.
- ***Feedback*** is derived from three sources: the implemented software itself, the customer, and other software team members

- Beck argues that strict adherence to certain XP practices demands *courage.* A better word might be *discipline.* Most software teams succumb, arguing that "designing for tomorrow" will save time and effort in the long run.
- The agile team inculcates *respect* among it
- members, between other stakeholders and team members, and indirectly, for the software itself.
- **1.7.3.4.2 The XP Process**
- Extreme Programming uses an object-oriented approach (Appendix 2) as its preferred development paradigm and encompasses a set of rules and practices that
- occur within the context of four framework activities: planning, design, coding, and testing. Figure 3.2 illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity.

FIGURE 3.2

The Extreme Programming process



- **Planning.** The planning activity (also called *the planning game*) begins with *listening*—a requirements gathering activity that enables the technical members of the XP team to

understand the business context for the software and to get a broad feel for required output and major features and functionality.

- **Design.** XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less,
- nothing more. The design of extra functionality (because the developer assumes it will be required later) is discouraged. XP encourages the use of (Class-responsibility-collaboration) as an effective mechanism for

  thinking about the software in an object-oriented context.
- XP encourages *refactoring*—a construction technique that is also a method for design optimization.
- A central notion in XP is that design occurs both before *and after* coding commences
- **Coding**
- A key concept during the coding activity (and one of the most talked about aspects of XP) is *pair programming*. XP recommends that two people work together at one computer workstation to create code for a story.
- This provides a mechanism for real-time problem solving (two heads are often better than one) and real-time quality assurance (the code is reviewed as it is created). It also keeps the developers focused on the problem at hand. As pair programmers complete their work, the code they develop is integrated with the work of others.
- **Testing** The unit tests that are created should be implemented using a framework that enables them to be automated. XP *acceptance tests*, also called *customer tests*, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer.
- **1.7.3.4.3 Industrial XP**
- Joshua Kerievsky describes *Industrial Extreme Programming* (IXP) in the following manner: "IXP is an organic evolution of XP. It is imbued with XP's minimalist, customer-

centric, test-driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices." IXP incorporates **six new practices** that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

1) Readiness assessment

- . The assessment ascertains whether
- (1) an appropriate development environment exists to support IXP, (2) the team will be populated by the proper set of stakeholders,
- (3) the organization has a distinct quality program and supports continuous improvement,
- (4) the organizational culture will support the new values of an agile team, and
- (5) the broader project community will be populated appropriately.
- **Project community**
- A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders (e.g., legal staff, quality auditors, manufacturing or sales types) who "are often at the periphery of an IXP project yet they may play important roles on the project".
- **Project chartering.** Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.
- **Test-driven management.** Test-driven management establishes a series of measurable "destinations" and then defines mechanisms for determining whether or not these destinations have been reached.
- **Retrospectives.** An IXP team conducts a specialized technical review after a software increment is delivered. Called a *retrospective*, the review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release.

- **Continuous learning.** Because learning is a vital part of continuous process improvement, members of the XP team are encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher quality product.
- In addition to the six new practices discussed, IXP modifies a number of existing XP practices.
- ***Story-driven development*** **(SDD)** insists that stories for acceptance tests be written before a single line of code is generated.
- ***Domain-driven design*** **(DDD**) is an improvement on the "system metaphor" concept used in XP.
- ***Pairing*** extends the XP pair programming concept to include managers and other stakeholders. The intent is to improve knowledge sharing among XP team members who may not be directly involved in technical development.
- ***Iterative usability*** discourages front-loaded interface design in favor of usability design that evolves as software increments are delivered and users' interaction with the software is studied.
- **1.7.3.4.4 The XP Debate**
- Proponents counter that XP is continuously evolving and that many of the issues raised by critics have been addressed as XP practice matures. Among the issues that continue to trouble some critics of XP are:
- *Requirements volatility* (liability to change rapidly and unpredictably)
- *Conflicting customer needs*.
- *Requirements are expressed informally*.
- *Lack of formal design*.

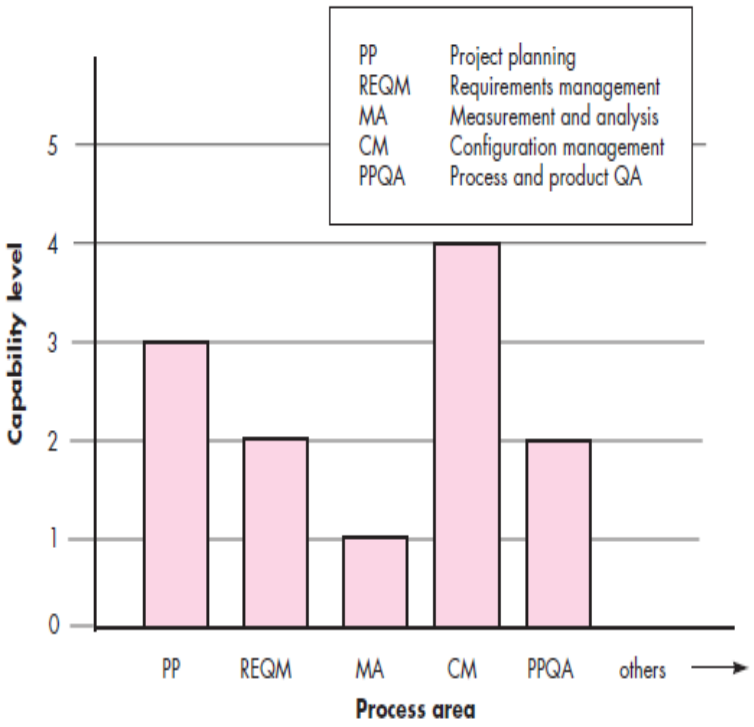
1.8 **CMMI** (***Capability Maturity Model Integration)***

- The original CMM was developed and upgraded by the Software Engineering Institute throughout the 1990s as a complete SPI framework. Today, it has evolved into the *Capability*

Maturity Model Integration (CMMI) , a comprehensive process meta-model that is predicated on a set of system and software engineering capabilities that should be present as organizations reach different levels of process capability and maturity.

- The CMMI represents a process meta-model in two different ways: (1) as a "continuous" model and (2) as a "staged" model. The continuous CMMI meta-model describes a process in two dimensions as illustrated in Figure 30.2. Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels:

FIGURE 30.2
CMMI process area capability profile.
Source: [Phi02].



- **Level 0:** *Incomplete*—the process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.
- **Level 1:** *Performed*—all of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.

- **Level 2:** *Managed*—all capability level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description".

- **Level 3:** *Defined*—all capability level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets".

- **Level 4:** *Quantitatively managed*—all capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process" .

- **Level 5:** *Optimized*—all capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

- The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. *Specific goals* establish the characteristics that must exist if the activities implied by a process area are to be effective. *Specific practices* refine a goal into a set of process-related activities.

- For example, **project planning** is one of eight process areas defined by the CMMI for "project management" category.The specific goals (SG) and the associated specific practices (SP) defined for **project planning** are :

- **SG 1 Establish Estimates**
- SP 1.1-1 Estimate the Scope of the Project
- SP 1.2-1 Establish Estimates of Work Product and Task Attributes
- SP 1.3-1 Define Project Life Cycle
- SP 1.4-1 Determine Estimates of Effort and Cost
- **SG 2 Develop a Project Plan**

SP 2.1-1 Establish the Budget and Schedule
SP 2.2-1 Identify Project Risks
SP 2.3-1 Plan for Data Management
SP 2.4-1 Plan for Project Resources
SP 2.5-1 Plan for Needed Knowledge and Skills
SP 2.6-1 Plan Stakeholder Involvement
SP 2.7-1 Establish the Project Plan

- **SG 3 Obtain Commitment to the Plan**

SP 3.1-1 Review Plans That Affect the Project
SP 3.2-1 Reconcile Work and Resource Levels
SP 3.3-1 Obtain Plan Commitment

In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. Hence, to achieve a particular capability level, the generic goal for that level and the generic practices that correspond to that goal must be achieved. To illustrate, the generic goals (GG) and practices (GP) for the **project planning** process area are :

- **GG 1 Achieve Specific Goals**

GP 1.1 Perform Base Practices

- **GG 2 Institutionalize a Managed Process**

GP 2.1 Establish an Organizational Policy

GP 2.2 Plan the Process

GP 2.3 Provide Resources

GP 2.4 Assign Responsibility

GP 2.5 Train People

GP 2.6 Manage Configurations

GP 2.7 Identify and Involve Relevant Stakeholders

GP 2.8 Monitor and Control the Process

GP 2.9 Objectively Evaluate Adherence

GP 2.10 Review Status with Higher-Level Management

GG 3 Institutionalize a Defined Process

GP 3.1 Establish a Defined Process

GP 3.2 Collect Improvement Information

- **GG 4 Institutionalize a Quantitatively Managed Process**

GP 4.1 Establish Quantitative Objectives for the Process

GP 4.2 Stabilize Sub-process Performance

- **GG 5 Institutionalize an Optimizing Process**

GP 5.1 Ensure Continuous Process Improvement

GP 5.2 Correct Root Causes of Problems

The staged CMMI model defines the same process areas, goals, and practices as the continuous model. The primary difference is that the staged model defines five maturity levels, rather than five capability levels. To achieve a maturity level, the specific goals and practices associated with a set of process areas must be achieved. The relationship between maturity levels and process areas is shown in Figure 30.3.

FIGURE 30.3
Process areas required to achieve a maturity level.
Source: [Phi02].

Level	Focus	Process Areas
Optimizing	*Continuous process improvement*	Organizational innovation and deployment Causal analysis and resolution
Quantitatively managed	*Quantitative management*	Organizational process performance Quantitative project management
Defined	*Process standardization*	Requirements development Technical solution Product integration Verification Validation Organizational process focus Organizational process definition Organizational training Integrated project management Integrated supplier management Risk management Decision analysis and resolution Organizational environment for integration Integrated teaming
Managed	*Basic project management*	Requirements management Project planning Project monitoring and control Supplier agreement management Measurement and analysis Process and product quality assurance Configuration management
Performed		

The People CMM (*Capability Maturity Model)*

- The *People Capability Maturity Model* "is a roadmap for implementing workforce practices that continuously improve the capability of an organization's workforce".
- The goal of the People CMM is to encourage continuous improvement of generic workforce knowledge (called "core competencies"), specific software engineering and project management skills (called "workforce competencies"), and process-related abilities.
- Like the CMM, CMMI, and related SPI frameworks, the People CMM defines a set of five organizational maturity levels that provide an indication of the relative sophistication of workforce practices and processes. These maturity levels are tied to the existence (within an organization) of a set of key process areas (KPAs). An overview of organizational levels and related KPAs is shown in Figure 30.4

FIGURE 30.4

Process areas for the People CMM

Level	Focus	Process Areas
Optimized	*Continuous improvement*	Continuous workforce innovation Organizational performance alignment Continuous capability improvement
Predictable	*Quantifies and manages knowledge, skills, and abilities*	Mentoring Organizational capability management Quantitative performance management Competency-based assets Empowered workgroups Competency integration
Defined	*Identifies and develops knowledge, skills, and abilities*	Participatory culture Workgroup development Competency-based practices Career development Competency development Workforce planning Competency analysis
Managed	*Repeatable, basic people management practices*	Compensation Training and development Performance management Work environment Communication and co-ordination Staffing
Initial	*Inconsistent practices*	

UNIT-II

PROCESS MODELS: A Generic process model (framework), Process assessment and improvement, Prescriptive process models: The waterfall model, Incremental process models, Evolutionary process models, The Unified process, **SOFTWARE REQUIREMENTS**: Functional and Non-functional requirements; User requirements, The software requirements document, **Requirements Engineering Processes**: Requirements elicitation and analysis, Requirements validation Requirements management, **System Modeling**: Context models, Behavioral models, Data models, Object models, Structured Methods.

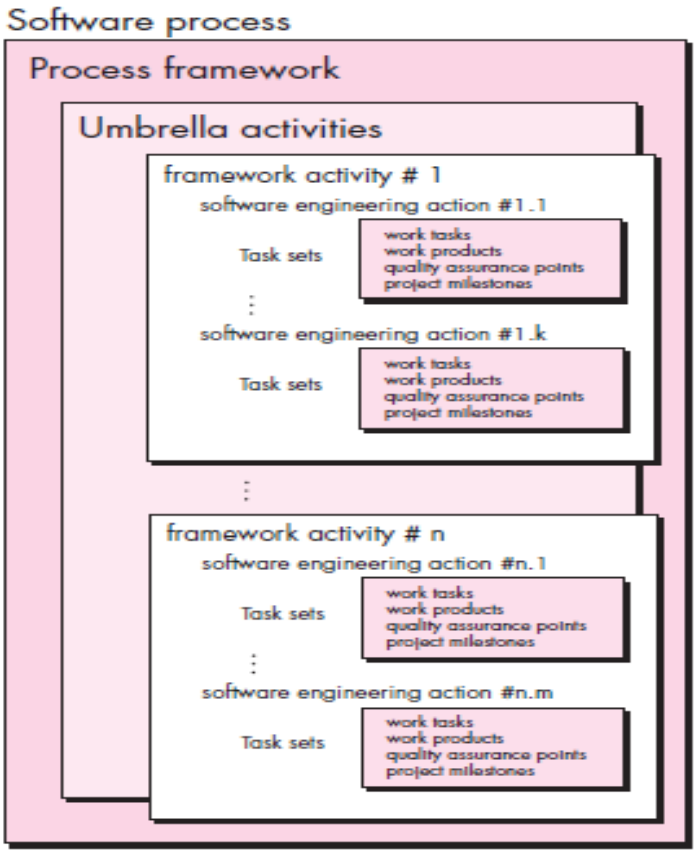
2.1.1 A Generic Process Model

A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.

The software process is represented schematically in Figure 2.1. Referring to the figure, each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

A generic process framework for software engineering defines five framework activities—**communication, planning, modeling, construction,** and **deployment.** In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

FIGURE 2.1
A software process framework



- One important aspect of the software process called *process flow*—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in Figure 2.2.
- A *linear process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 2.2a). An *iterative process flow* repeats one or more of the activities before proceeding to the next (Figure 2.2b).
- An *evolutionary process flow* executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software (Figure 2.2c).
- A *parallel process flow* (Figure 2.2d) executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

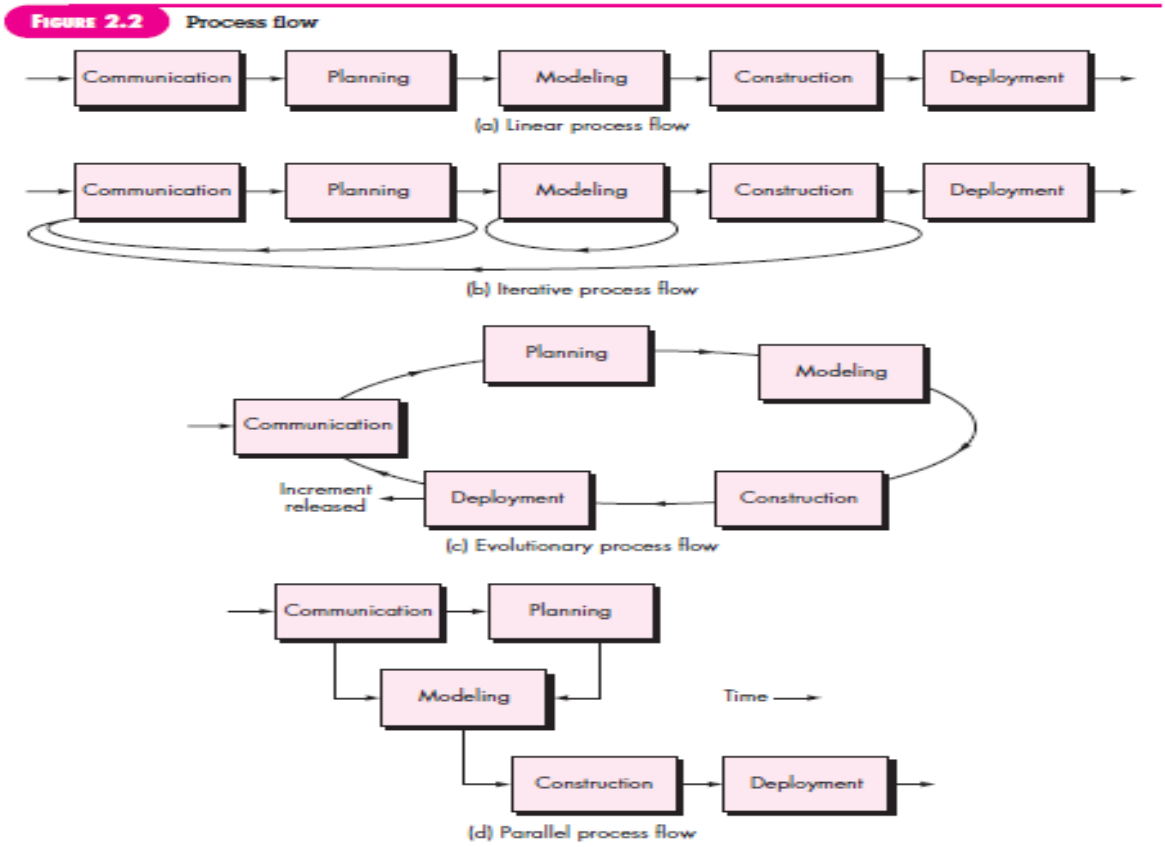


FIGURE 2.2 Process flow

- **2.1.1 Defining a Framework Activity**

Although five framework activities and provided a basic definition , a software team would need significantly more information before it could properly execute any one of these activities as part of the software process. Therefore, you are faced with a key question:

What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the communication activity might encompass little more than a phone call with the appropriate stakeholder. Therefore, the only necessary action is *phone conversation*, and the work tasks (the *task set*) that this action encompasses are:

- 1. Make contact with stakeholder via telephone.
- **2.** Discuss requirements and take notes.
- **3.** Organize notes into a brief written statement of requirements.

- **4.** E-mail to stakeholder for review and approval.

If the project was considerably more complex with many stakeholders, each with a different set of (sometime conflicting) requirements, the communication activity might have six distinct actions : ***inception, elicitation, elaboration ,negotiation, specification,*** **and** ***validation.*** Each of these software engineering actions would have many work tasks and a number of distinct work products.

2.1.2 Identifying a Task Set

Referring again to Figure 2.1, each software engineering action (e.g., *elicitation*, an action associated with the communication activity) can be represented by a number of different *task sets*—each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones. You should choose a task set that best accommodates the needs of the project and the characteristics of your team. This implies that a software engineering action can be adapted to the specific needs of the software project and the characteristics of the project team.

Task Set

A task set defines the actual work to be done to accomplish the objectives of a software engineering action. For example, *elicitation* (more commonly called "requirements gathering") is an important software engineering action that occurs during the communication activity. The goal of requirements gathering is to understand what various stakeholders want from the software that is to be built.

For a small, relatively simple project, the task set for requirements gathering might look like this:

1. Make a list of stakeholders for the project.
2. Invite all stakeholders to an informal meeting.
3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a final list.
5. Prioritize requirements.
6. Note areas of uncertainty.

For a larger, more complex software project, a different task set would be required. It might encompass the following work tasks:

1. Make a list of stakeholders for the project.
2. Interview each stakeholder separately to determine overall wants and needs.
3. Build a preliminary list of functions and features based on stakeholder input.
4. Schedule a series of facilitated application specification meetings.
5. Conduct meetings.
6. Produce informal user scenarios as part of each meeting.
7. Refine user scenarios based on stakeholder feedback.
8. Build a revised list of stakeholder requirements.
9. Use quality function deployment techniques to prioritize requirements.
10. Package requirements so that they can be delivered incrementally.
11. Note constraints and restrictions that will be placed on the system.
12. Discuss methods for validating the system.

Both of these task sets achieve "requirements gathering," but they are quite different in their depth and formality. The software team chooses the task set that will allow it to achieve the goal of each action and still maintain quality and agility.

- **2.1.3 Process Patterns**
- A *process pattern* describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template -a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.
- Patterns can be defined at any level of abstraction.2 In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).
- Ambler has proposed a template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

Forces. The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

- **Type.** The pattern type is specified. Ambler suggests three types:

1. *Stage pattern*—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of a stage pattern might be **Establishing Communication.** This pattern would incorporate the task pattern **Requirements Gathering** and others.

2. *Task pattern*—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **Requirements Gathering** is a task pattern).

3. *Phase pattern*—define the sequence of framework activities that occurs within the process, even

when the overall flow of activities is iterative in nature. An example of a phase pattern might be **Spiral Model** or **Prototyping.**

- **Initial context.** Describes the conditions under which the pattern applies. Prior to the initiation of the pattern: (1) What organizational or team-related activities have already occurred? (2) What is the entry state for the process?

(3) What software engineering information or project information already exists?
For example, the **Planning** pattern (a stage pattern) requires that (1) customers and software engineers have established a collaborative communication;
(2) successful completion of a number of task patterns [specified] for the **Communication** pattern has occurred; and (3) the project scope, basic business requirements, and project constraints are known.

- **Problem.** The specific problem to be solved by the pattern.

Solution. Describes how to implement the pattern successfully. This section describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern. It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

Resulting Context. Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern:
(1) What organizational or team-related activities must have occurred?
(2) What is the exit state for the process? (3) What software engineering information or project information has been developed?
Related Patterns. Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form.

2.1.2 **Process Assessment and Improvement**

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics . Process patterns must be coupled with solid software engineering practice. In addition, the process itself can be assessed to ensure that it meets a set of basic process

criteria that have been shown to be essential for a successful software engineering.

Standard CMMI Assessment Method for Process Improvement

- **(SCAMPI)**—provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment . **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**—provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment.

- **SPICE (ISO/IEC15504)**—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process .

- **ISO 9001:2000 for Software**—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies .

2.1.3 **Perspective Process Models**

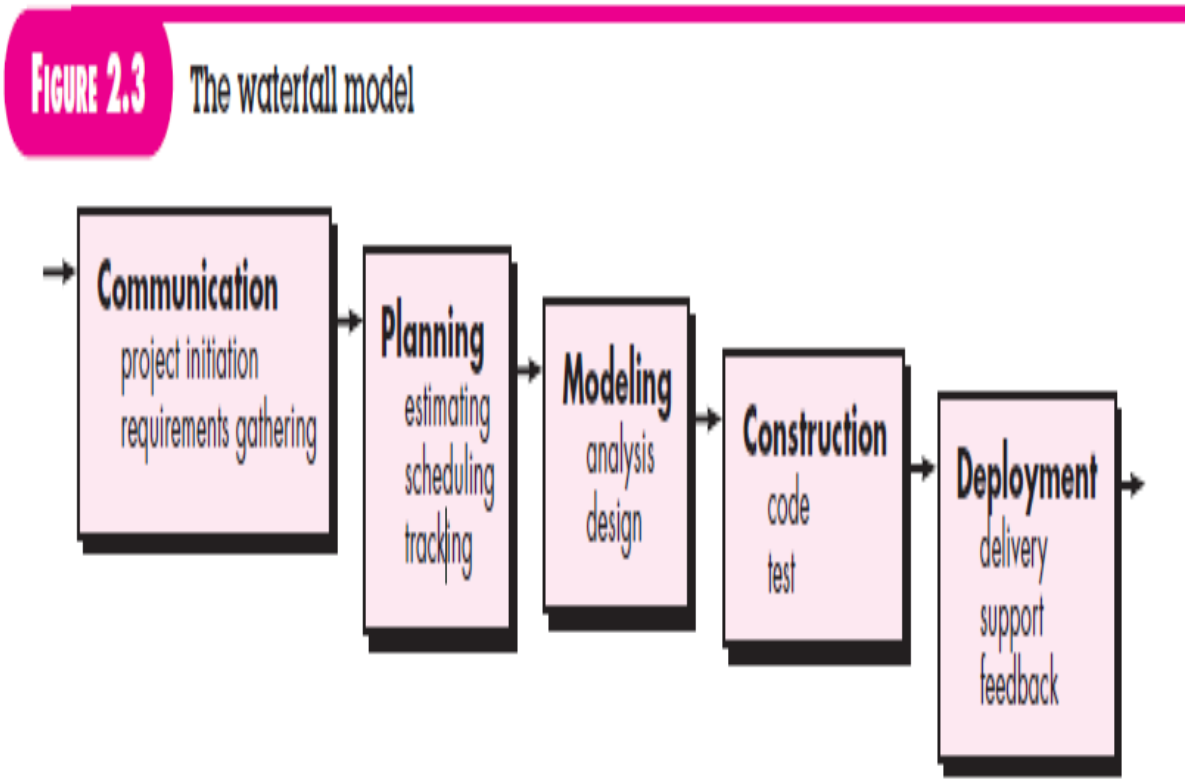
- Prescriptive process models were originally proposed to bring order to the chaos of software development. History has indicated that these traditional models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams.

- Called “prescriptive” because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.

- All software process models can accommodate the generic framework activities described , but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner.

2.3.1 The Waterfall Model

- There are times when the requirements for a problem are well understood—when work flows from **communication** through **deployment** in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made .
- It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable.
- The ***waterfall model,*** sometimes called the ***classic life cycle***, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software (Figure 2.3).

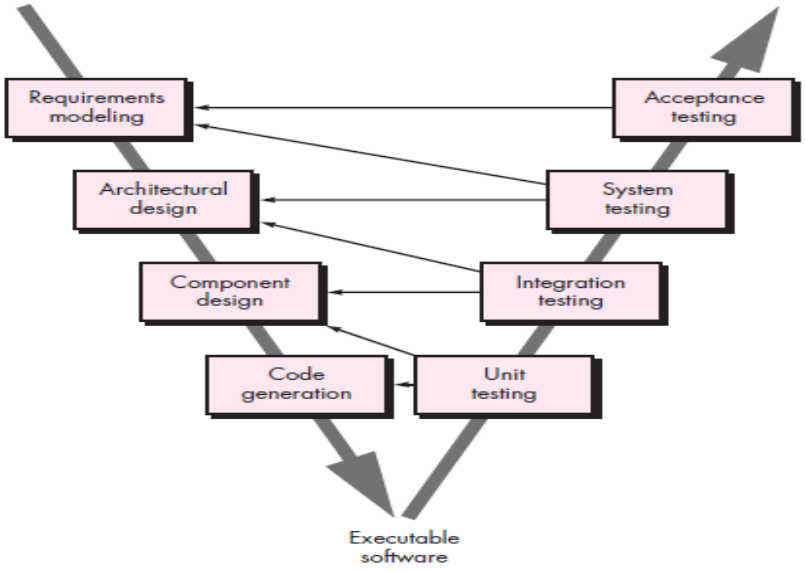


- A variation in the representation of the waterfall model is called the *V-model*. Represented in Figure 2.4, the V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software

team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.

- Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.
- In reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

FIGURE 2.4
The V-model



- The waterfall model is the oldest paradigm for software engineering. However, over the past three decades, criticism of this process model has caused even ardent supporters to question its efficacy . Among the problems that are sometimes encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes.

- Although the linear model can accommodate iteration, it does so indirectly.
- As a result, changes can cause confusion as the project team proceeds

2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

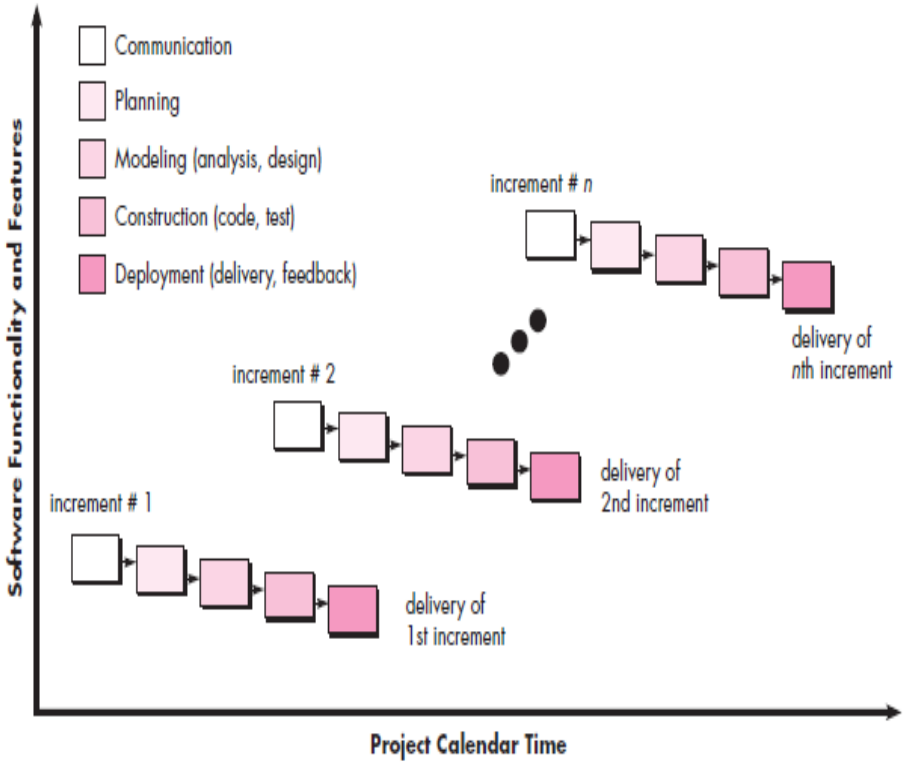
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

- **2.3.2 Incremental Process Models**

- There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

FIGURE 2.5

The incremental model



- The *incremental* model combines elements of linear and parallel process flows discussed . Referring to Figure 2.5, the incremental model applies linear sequences in a staggered fashion as calendar time progresses.

- Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow .

- For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

- When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a plan is developed for the next increment.

2.3.3 Evolutionary Process Models

- Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. Evolutionary models are iterative. They are characterized in a manner that

enables you to develop increasingly more complete versions of the software

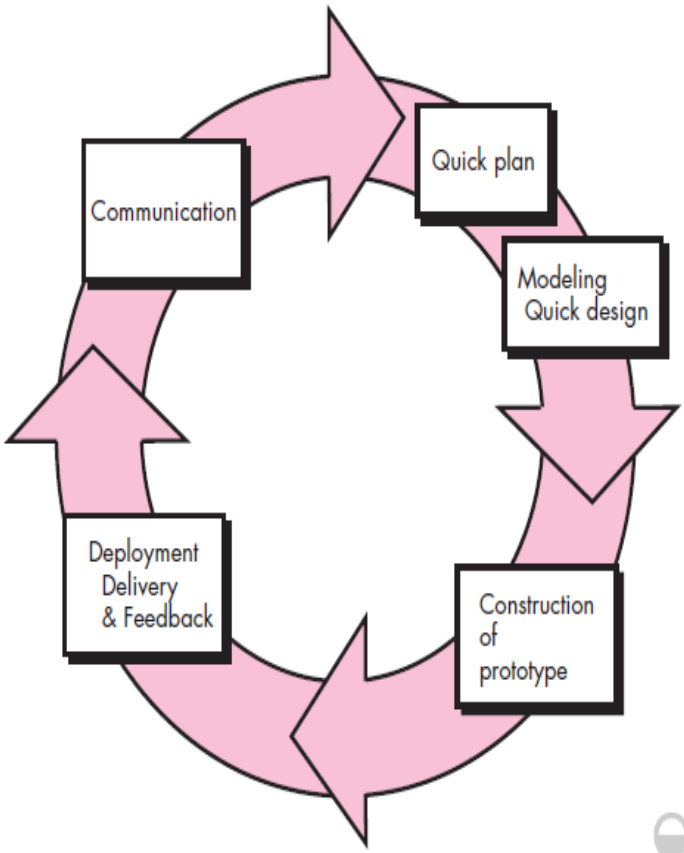
- **Prototyping.** Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach.

- Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models understand

- The prototyping paradigm (Figure 2.6) begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs.

FIGURE 2.6

The prototyping paradigm



- Yet, prototyping can be problematic for the following reasons:

1. Stakeholders see what appears to be a working version of the software, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working Product.

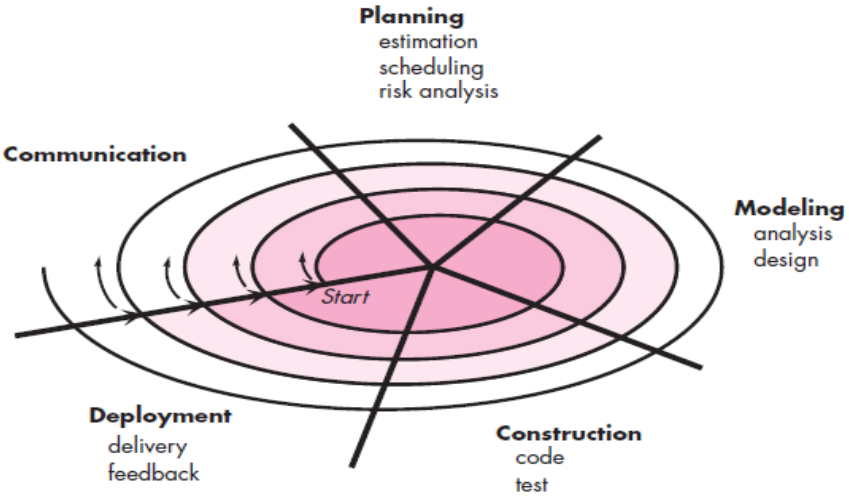
2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all

the reasons why they were inappropriate.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part), and the actual software is engineered with an eye toward quality.

- **The Spiral Model.** Originally proposed by Barry Boehm , the *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- It provides the potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner:
- The spiral development model is a *risk*-driven *process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems.
- **It has two main distinguishing features.**
- One is a *cyclic* approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk.
- The other is a set of *anchor point milestones* for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.
- During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

FIGURE 2.7
A typical spiral model



- A spiral model is divided into a set of framework activities defined by the software engineering team. Each of the framework activities represent one segment of the spiral path illustrated in Figure 2.7As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center.
- ***Anchor point milestones*—a** combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.
- The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
- Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery.
- In addition, the project manager adjusts the planned number of iterations required to complete the software. Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software.
- The first circuit around the spiral might represent a "concept development project" that starts at the core of the spiral and continues for multiple iterations until concept development is complete.
- If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "new product development project" commences.
- The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "product enhancement project."
- The spiral model is a realistic approach to the development of large-scale systems and software.
- The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic

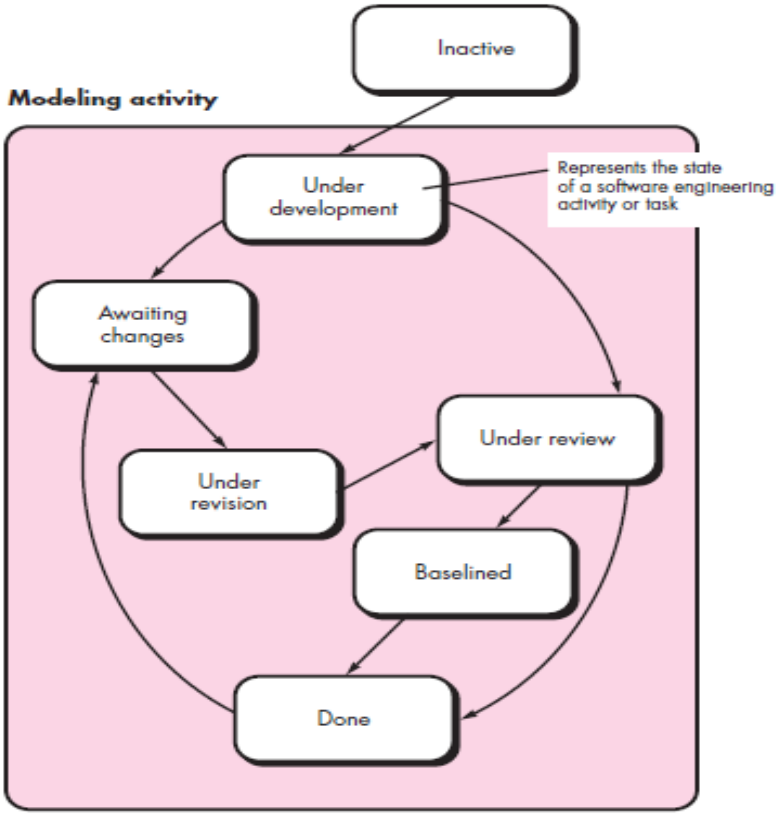
2.3.4 Concurrent Models

The *concurrent development model,* sometimes called *concurrent engineering,* allows a software team to represent iterative and concurrent elements of any of the process models described .

- Figure 2.8 provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach.

- The activity—**modeling**—may be in any one of the states12 noted at any given time. Similarly, other activities, actions, or tasks (e.g., **communication** or **construction**) can be represented in an analogous manner. All software engineering activities exist concurrently but reside in different states.

FIGURE 2.8
One element of the concurrent process model



2.1.4 Unified Process Model

- The *Unified Process* was developed by Jacobsen, Booch, and Rumbaugh, who were already some of the biggest names in OOA&D before they decided to collaborate on a unified version of their previously distinctive approaches.
- A process model that was created 1997 to give a framework for Object-oriented Software Engineering
- Iterative, incremental model to adapt to specific project needs
- Risk driven development Combining spiral and evolutionary models.

In some ways the Unified Process is an attempt to draw on the best features and

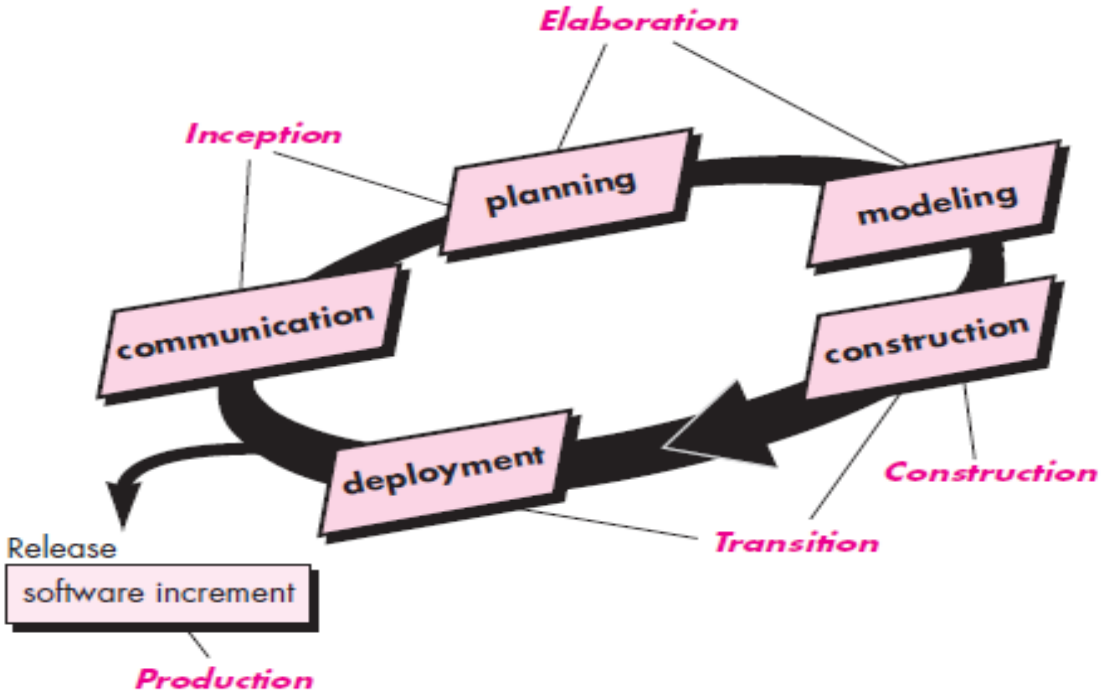
- characteristics of traditional software process models, but characterize them in a way that implements many of the best principles of agile software development. The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system.

- **A Brief History**

- During the early 1990s James Rumbaugh , Grady Booch , and Ivar Jacobson began working on a "unified method" that would combine the best features of each of their individual object-oriented analysis and design methods and adopt additional features proposed by other experts in object-oriented modeling. The result was UML—a *unified modeling language* that contains a robust notation for the modeling and development of object-oriented systems.

- Jacobson, Rumbaugh, and Booch developed the *Unified Process*, a framework for object-oriented software engineering using UML. Today, the Unified Process (UP) and UML are widely used on object-oriented projects of all kinds. The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

- **Phases of the Unified Process Fig : 8.9**



-

- **Advantages of UP**
- Rapid feedback from users and developers
    - Then adapt to changes in the next iteration (adaptive development)
- Visible progress
- Start with high risk
- Manage complexity by dividing the problem into smaller ones

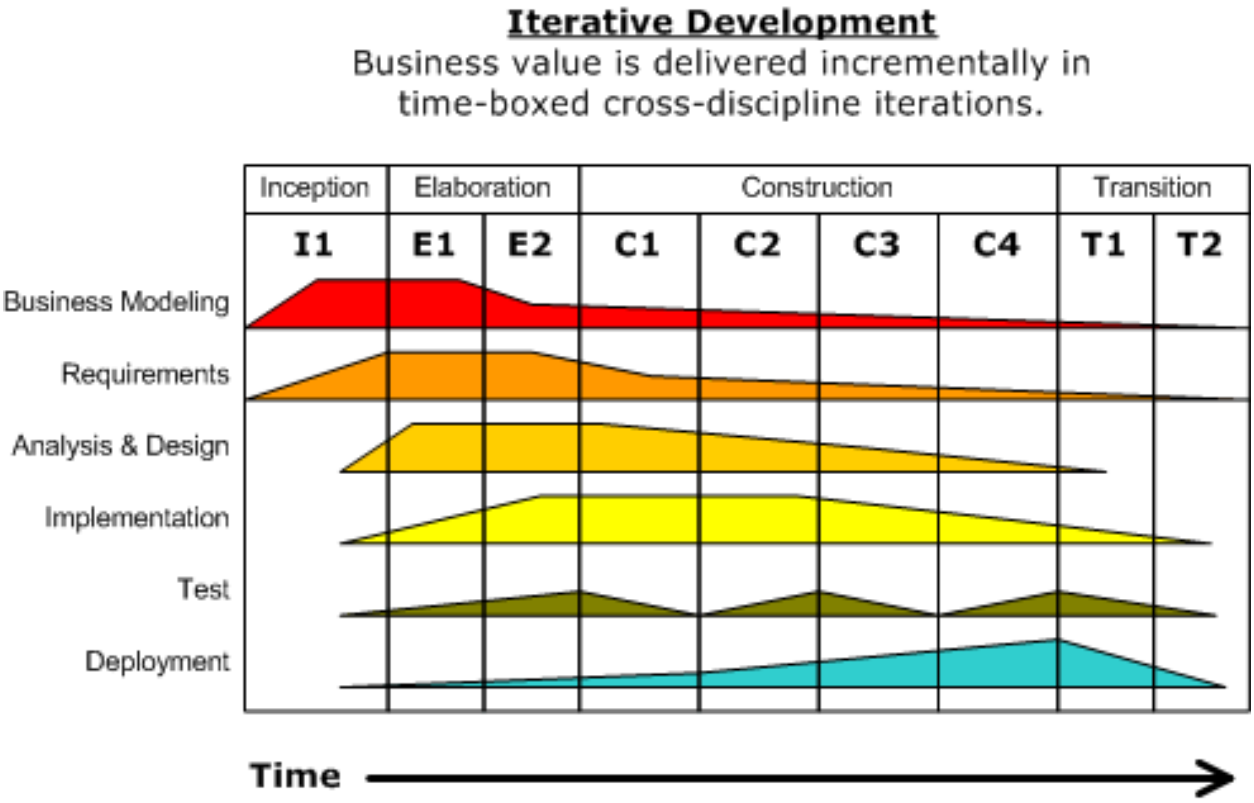
Stages of the UP

- Inception: Customer communication, project vision and planning activities (feasibility study)
- Elaboration: multiple iterations that refines the requirements and models of the system
- Construction: develop software code
- Transition: user testing and installation
- Production: operation
- **Inception Stage**
- The *inception phase* of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed.
- Perform feasibility study …..
- Identify the project vision (vision document)
- Identify general business requirements
- Identify project and business risks
- Produce initial use-case model (10-20%)
- Plan the elaboration stages
- Rough architecture of the software (subsystems)
- **Elaboration Stage**
- An iterative process where refinements are made on system requirements, system design, develop part of the code and test it.
- Products from these iterations:
- Refinements on use-case model

- Software architecture description
- Executable prototypes
- Initial design model
- Refinement on project risks and plan
- The *elaboration phase* encompasses the communication and modeling activities of the generic process model (Figure 2.9). Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—
- the use case model, the requirements model, the design model, the implementation model, and the deployment model.
- **Construction Stage**
- The *construction phase* of the UP is identical to the construction activity defined for the generic software process.
- Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users.
- To accomplish this, requirements and design models that were started during the elaboration phase are completed to reflect the final version of the software increment
- Translate the design into software components
- Products of this stage are:
- Design model
- Integrated software components
- Test plan and test cases
- User documentation
- **Transition Stage**
- The *transition phase* of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software is given to end users for beta testing and user feedback reports both defects and necessary changes.
- Deliver the software and documentation
- Get user feedback from Beta tests
- ***Production Stage***

- The *production phase* of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.
- A software engineering workflow is distributed across all UP phases. In the context of UP, a *workflow* is analogous to a task set . That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks.

Changes in activities according to stages

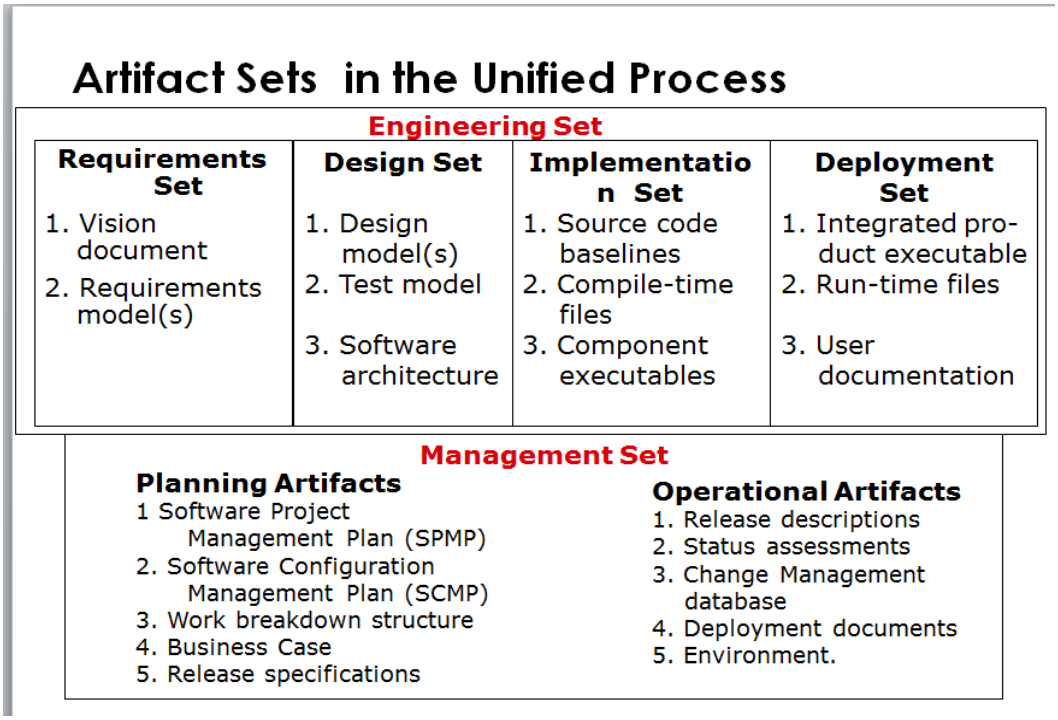


Artifact Sets in the Unified Process

- **Artifact set:**
  - A set of artifacts developed and reviewed as a single entity

- **The Unified Process distinguishes five artifact sets**
  - Management set
  - Requirements set
  - Design set
  - Implementation set
  - Deployment set

  Also called the engineering set.



2.2 System requirements

Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. I distinguish between them by using the term ‘user requirements’ to mean the high-level abstract requirements and ‘system requirements’ to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.

2. System requirements are more detailed descriptions of the software system’s functions, services, and operational constraints. The system requirements document (sometimes called a functional

specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

- Different levels of requirements are useful because they communicate information about the system to different types of reader. Figure 4.1 illustrates the distinction between user and system requirements. This example from a mental health care patient management system (MHC-PMS) shows how a user requirement may be expanded into several system requirements. You can see from Figure 4.1 that the user requirement is quite general. The system requirements provide more specific information about the services and functions of the system that is to be implemented.

User Requirement Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

Figure 4.1 User and system requirements

- **Functional and Non-Functional requirements**
- Software system requirements are often classified as functional requirements or nonfunctional requirements:
- 1. ***Functional requirements*** These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular

situations. In some cases, the functional requirements may also explicitly state what the system should not do.

- 2. ***Non-functional requirements*** These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services.

- 3. **Domain Requirements** These are requirements that come from the application domain of the system and that reflect characteristics and constraints of that domain. They may be functional or non-functional requirements.

- Requirements are not independent and that one requirement often generates or constrains other requirements.

- The system requirements therefore do not just specify the services or the features of the system that are required; they also specify the necessary functionality to ensure that these services/features are delivered properly.

- You need to write requirements at different levels of detail because different readers use them in different ways. Figure 4.2 shows possible readers of the user and system requirements. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation

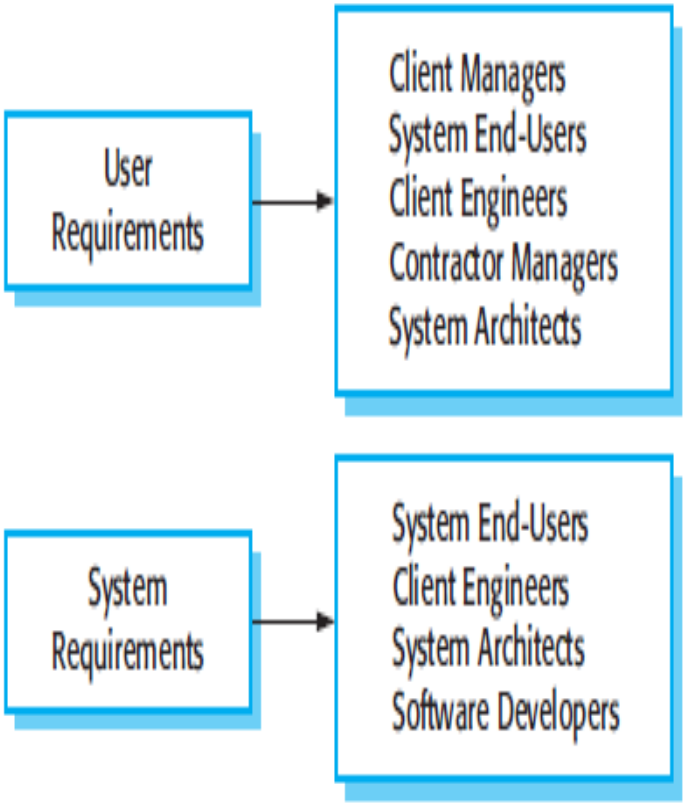


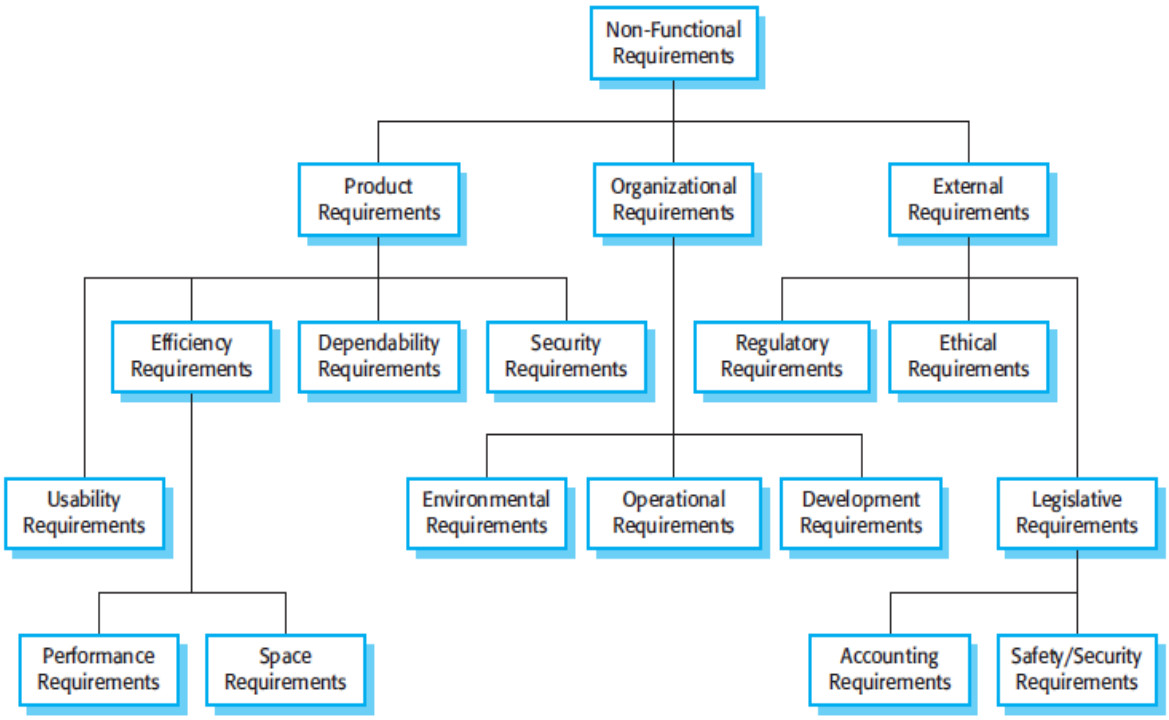
Figure 4.2 Readers of different types of requirements specification

- **1. Functional requirements**
- The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users.
- However, more specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail.

- Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems.
- For example, here are examples of functional requirements for the MHC-PMS system, used to maintain information about patients receiving treatment for mental health problems:
- 1. A user shall be able to search the appointments lists for all clinics.
- 2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
- 3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number. These functional user requirements define specific facilities to be provided by the system.
- For example, the first example requirement for the MHC-PMS states that a user shall be able to search the appointments lists for all clinics. The rationale for this requirement is that patients with mental health problems are sometimes confused. They may have an appointment at one clinic but actually go to a different clinic. If they have an appointment, they will be recorded as having attended, irrespective of the clinic.
- The medical staff member specifying this may expect 'search' to mean that, given a patient name, the system looks for that name in all appointments at all clinics.
- However, this is not explicit in the requirement. System developers may interpret the requirement in a different way and may implement a search so that the user has to choose a clinic then carry out the search. This obviously will involve more user input and so take longer.
- In principle, the functional requirements specification of a system should be both **complete** and **consistent.**
- **Completeness** means that all services required by the user should be defined.
- **Consistency** means that requirements should not have contradictory definitions

- In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness.
- One reason for this is that it is easy to make mistakes and omissions when writing specifications for complex systems.
- Another reason is that there are many stakeholders in a large system. A stakeholder is a person or role that is affected by the system in some way.
- **2. Non-functional requirements**
- Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. They may relate to emergent system properties such as reliability, response time, and store occupancy. Alternatively, they may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems.
- Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole. Non-functional requirements are often more critical than individual functional requirements
- Although it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), it is often more difficult to relate components to non-functional requirements. The implementation of these requirements may be diffused throughout the system. There are two reasons for this:
- 1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance
- requirements are met, you may have to organize the system to minimize communications between components.
- 2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required.

In addition, it may also generate requirements that restrict existing requirements. Fig :4.3 is shown in next page



- Figure 4.3 is a classification of non-functional requirements. You can see from this diagram that the non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements), or from external sources:
- 1. *Product requirements* These requirements specify or constrain the behavior of the software. Examples include performance requirements on how fast the system must execute and how much memory it requires, reliability requirements that set out the acceptable failure rate, security requirements, and usability requirements.
- 2. *Organizational requirements* These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organization.

- 3. *External requirements* This broad heading covers all requirements that are derived from factors external to the system and its development process.
- A common problem with non-functional requirements is that users or customers often propose these requirements as general goals, such as ease of use, the ability of the system to recover from failure, or rapid user response.
- Figure 4.5 shows metrics that you can use to specify non-functional system properties.

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

- **3. Domain requirements**
- Domain requirements are derived from the application domain of the system rather than from the specific needs of system users. They may be new functional requirements in their own right, constrain existing functional requirements, or set out how particular computations must be carried out.
- The problem with domain requirements is that software engineers may not understand the characteristics of the domain in which the system operates. They often cannot tell whether or not a domain requirement has been missed out or conflicts with other requirements
- Domain requirements are derived from the application domain of the system rather than from the specific needs of system users. They usually include specialized domain terminology or reference to domain concepts.

- They may be new functional requirements in their own right, constrain existing functional requirements or set out how particular computations must be carried out. Because these requirements are specialized, software engineers often find it difficult to understand how they are related to other system requirements.
- Domain requirements are important because they often reflect fundamentals of the application domain. If these requirements are not satisfied, it may be impossible to make the system work satisfactorily.
- The LIBSYS system includes a number of domain requirements:
- I. There shall be a standard user interface to all databases that shall be based on the Z39.50 standard.
- 2. Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user s requirements, these documents will either be printed locally on the system server for manual forwarding to the user or routed to a network printer.
- The first requirement is a design constraint. It specifies that the user interface to the database must be implemented according to a specific library standard.
- The second requirement has been introduced because of copyright laws that apply to material used in libraries. It specifies that the system must include an automatic delete-on-print facility for some classes of document.

2.2.2 **User Requirements**

- The user requirements for a system should describe the functional and non-functional requirements so that they are understandable by the system users without detailed technical knowledge.
- Consequently, if you are writing user requirements, you should not use software jargon, structured notations or formal notations, or describe the requirement by describing the system implementation.
- You should write user requirements in simple language, with simple tables and forms and intuitive diagrams.

- Various problems can arise when requirements are written in natural language sentences in a text document.
- 1) **Lack of clarity:** It is sometimes difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read.
- 2) **Requirements confusion:** Functional requirements, non-functional requirements, system goals and design information may not be clearly distinguished.
- 3) **Requirements amalgamation:** Several different requirements may be expressed together as a single requirement.
- As an illustration of some of these problems, consider one of the requirements for the library shown in Figure 6.8.
- This requirement includes both conceptual and detailed information. It expresses the concept that there should be an accounting system as an inherent part of LIBSYS.
- However, it also includes the detail that the accounting system should support discounts for regular LIBSYS users. This detail would have been better left to the system requirements specification
- **Fig 6.8 -** *LIBSYS shall provide a financial accounting system that maintains records of all payments made by users of the system. System managers may configure this system so that regular users may receive discounted rates.*
- 2.2.3 **What is Software Requirement Specification - [SRS]?**
- A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.
- **Qualities of SRS:**
- Correct
- Unambiguous

- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- Traceable
- **Types of Requirements:**
- The below diagram depicts the various types of requirements that are captured during SRS.

- **Software Requirements Specification document**
- A Software Requirements Specification (SRS) is a document that describes the nature of a project, software or application. In simple words, SRS document is a manual of a project provided it is prepared before you kick-start a project/application. This document is also known by the names SRS report, software document. A software document is primarily prepared for a project, software or any kind of application.

- There are a set of guidelines to be followed while preparing the software requirement specification document. This includes the purpose, scope, functional and nonfunctional requirements, software and hardware requirements of the project. In addition to this, it also contains the information about environmental conditions required, safety and security requirements, software quality attributes of the project etc.

- **What is a Software Requirements Specification document?**
- A Software requirements specification document describes the intended purpose, requirements and nature of a software to be developed. It also includes the yield and cost of the software.
- In this document, flight management project is used as an example to explain few points.

Table of Contents for a SRS Document

-

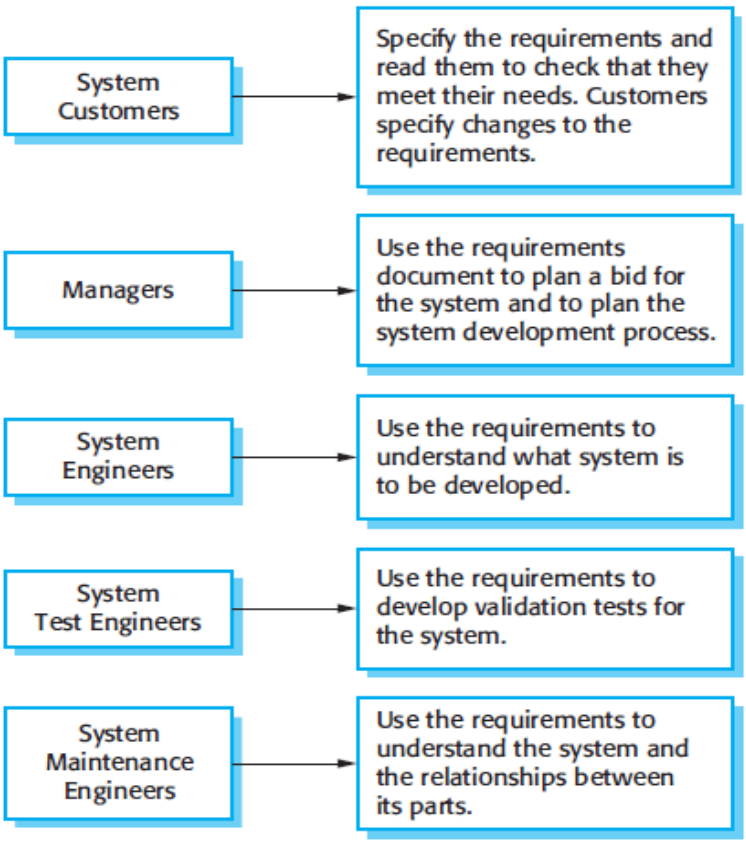


Figure 4.6 Users of a requirements document

- **Fig 4.7 The structure of a requirements Document**

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

- 2.3.1 **Requirements Engineering Process**

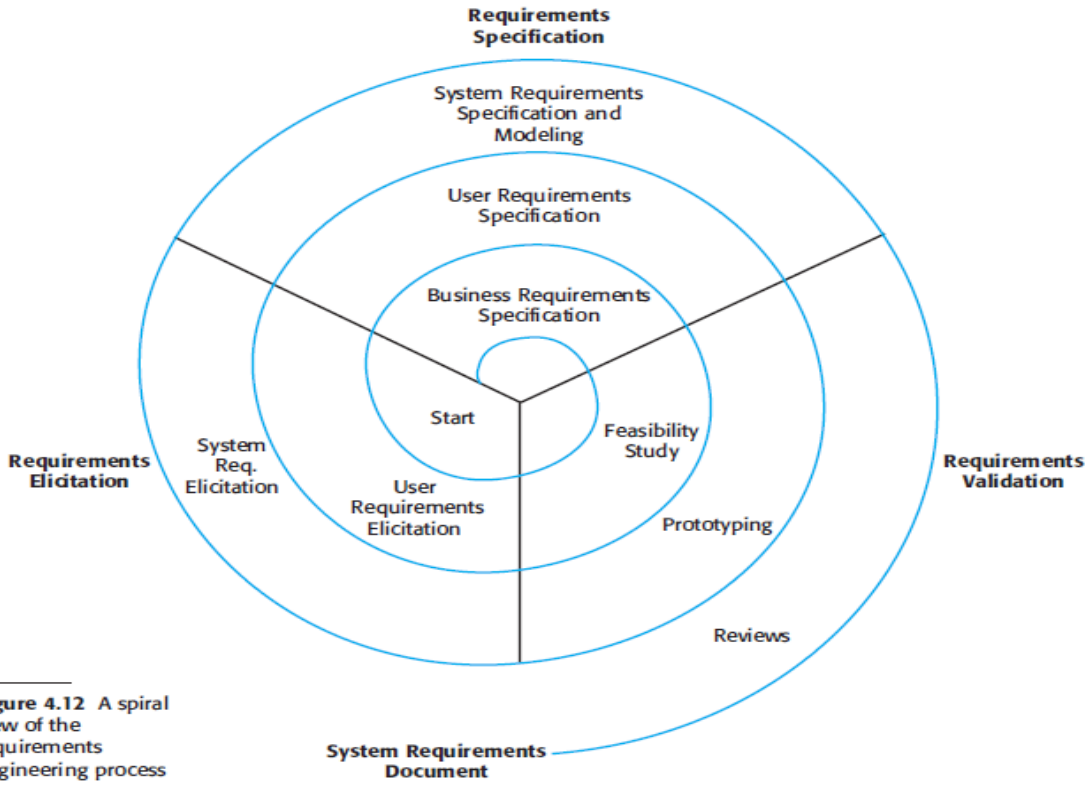


Figure 4.12 A spiral view of the requirements engineering process

- **2.3.2 Requirements elicitation and analysis**

- After an initial feasibility study, the next stage of the requirements engineering process is requirements elicitation and analysis.

- Requirements elicitation and analysis may involve a variety of different kinds of people in an organization. A system stakeholder is anyone who should have some direct or indirect influence on the system requirements.

- A process model of the elicitation and analysis process is shown in Figure 4.13. Each organization will have its own version or instantiation of this general model depending on local factors such as the expertise of the staff, the type of system being developed, the standards used, etc.

- Figure 4.13 shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities.

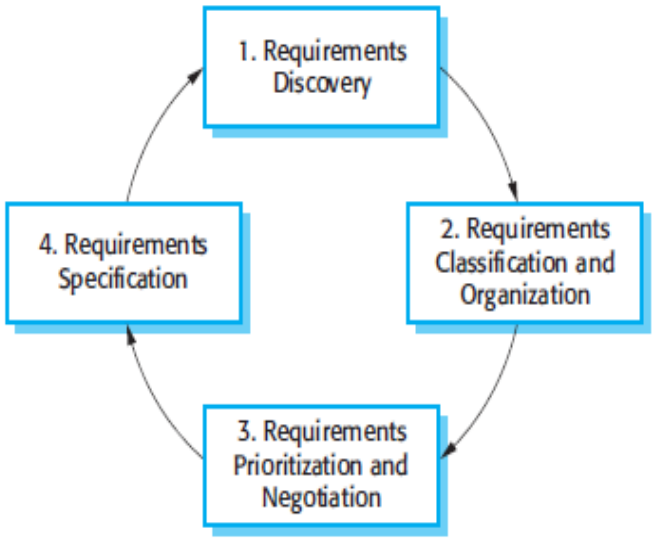


Figure 4.13 The requirements elicitation and analysis process

- 1. ***Requirements discovery*** This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.
- **2. *Requirements classification and organization*** This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system. In practice, requirements engineering and architectural design cannot be completely separate activities.
- **3. *Requirements prioritization and negotiation*** Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.
- **4. *Requirements specification*** The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced.
- **Eliciting and understanding requirements from system stakeholders is a difficult**

- **process for several reasons:**
- 1. Stakeholders often don’t know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don’t know what is and isn’t feasible.
- 2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer’s domain, may not understand these requirements.
- 3. Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
- 4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
- 5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

4.5.1 Requirements discovery

- Requirements discovery (sometime called requirements elicitation) is the process of gathering information about the required system and existing systems. Sources of information during the requirements discovery phase include documentation, system stakeholders, and specifications of similar systems. Stakeholders range from system end-users through managers and external stakeholders such as regulators who certify the acceptability of the system.

For example, system stakeholders for a bank ATM include:

I. Current bank customers who receive services from the system

2. Representatives from other banks who have reciprocal agreements that allow each other's ATMs to be used

3. *Managers of bank branches* who obtain management information from the system

4. Counter staff at bank branches who are involved in the day-to-day running of the system

5. Database administrators who are responsible for integrating the system with the bank's customer database

6. Bank security managers who must ensure that the system will not pose a security hazard

7. The bank's marketing department who are likely be interested in using the system as a means of marketing the bank

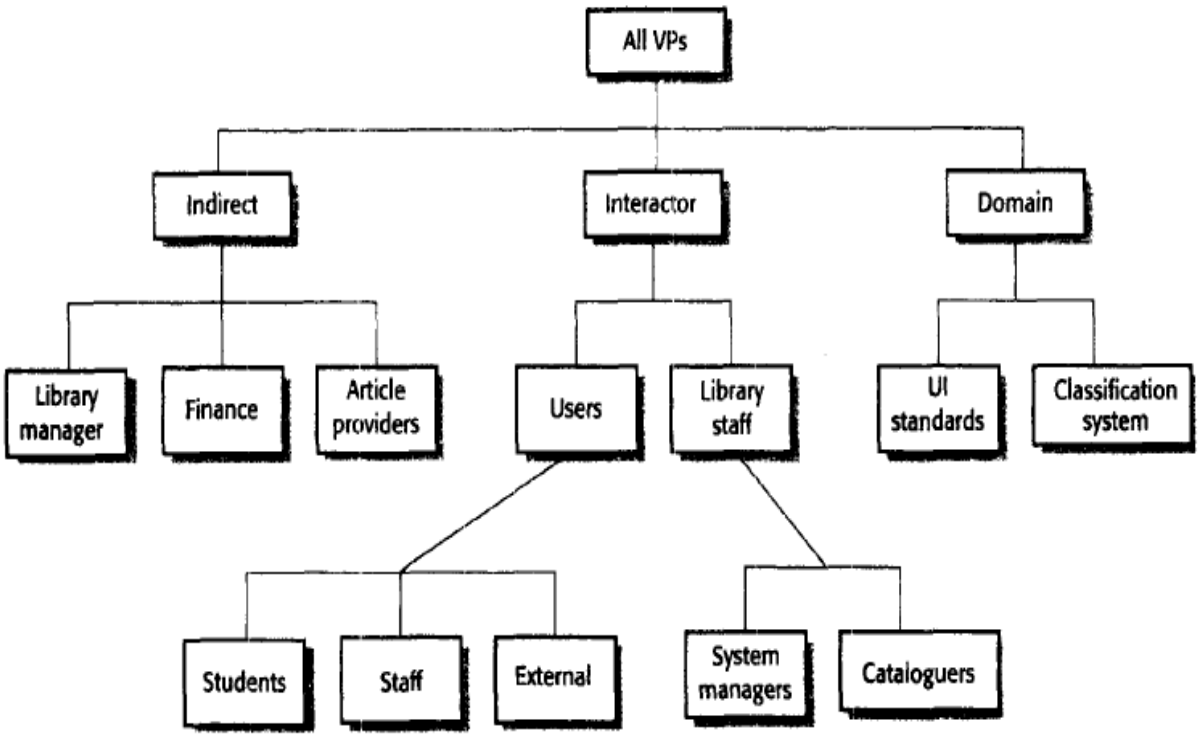
8. Hardware and software maintenance engineers who are responsible for maintaining and upgrading the hardware and software

9. National banking regulators who are responsible for ensuring that the system conforms to banking regulations

- These requirements sources (stakeholders, domain, systems) can all be represented as system viewpoints, where each viewpoint presents a sub-set of the requirements for the system. Each viewpoint provides a fresh perspective on the system, but these
- perspectives are not completely independent--they usually overlap so that they have common requirements.
- **Viewpoints**
- Viewpoint-oriented approaches to requirements engineering organize both the elicitation process and the requirements themselves using viewpoints. A key strength of viewpoint-oriented analysis is that it recognizes multiple perspectives and provides a framework for discovering conflicts in the requirements proposed by different stakeholders.
- Viewpoints can be used as a way of classifying stakeholders and other sources of requirements. There are three generic types of viewpoint:
- 1. *Interactor viewpoints* represent people or other systems that interact directly with the system. In the bank ATM system, examples of interactor viewpoints are the bank's customers and the bank's account database.

- *2. Indirect viewpoints* represent stakeholders who do not use the system themselves but who influence the requirements in some way. In the bank ATM system, examples of indirect viewpoints are the management of the bank and the bank security staff.
- *3. Domain viewpoints* represent domain characteristics and constraints that influence the system requirements. In the bank ATM system, an example of a domain viewpoint would be the standards that have been developed for interbank communications.
- The initial identification of viewpoints that are relevant to a system can sometimes be difficult. To help with this process, you should try to identify more specific viewpoint types:
- l. Providers of services to the system and receivers of system services
- 2. Systems that should interface directly with the system being specified
- 3. Regulations and standards that apply to the system
- 4. The sources of system business and non-functional requirements
- 5. Engineering viewpoints reflecting the requirements of people who have to develop, manage and maintain the system
- 6. Marketing and other viewpoints that generate requirements on the product features expected by customers and how the system should reflect the external image of the organization.

- **Figure 7.4 Viewpoints in LIBSYS**



4.5.2 Interviewing

- Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions.

- **Interviews may be of two types:**

1. Closed interviews, where the stakeholder answers a pre-defined set of questions.
2. Open interviews, in which there is no pre-defined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develop a better understanding of their needs.

- **Effective interviewers have two characteristics:**

1. They are open-minded, avoid pre-conceived ideas about the requirements, and are willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then they are willing to change their mind about the system.

2. They prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system. Saying to people 'tell me what you want' is unlikely to result in useful information.

Interviews are good for getting an overall understanding of what stakeholders do, how they might interact with the system and the difficulties that they face with current systems. People like talking about their work and are usually happy to get involved in interviews. However, interviews are not so good for understanding the requirements from the application domain.

It is hard to elicit domain knowledge during interviews for two reasons:

1. All application specialists use terminology and jargon that is specific to a domain. It is impossible for them to discuss domain requirements without using this terminology. They normally use terminology in a precise and subtle way that is easy for requirements engineers to misunderstand.

2. Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning. For example, for a librarian, it goes without saying that all acquisitions are catalogued before they are added to the library. However, this may not be obvious to the interviewer so it isn't taken into account in the requirements.

- Interviews are not an effective technique for eliciting knowledge about organizational requirements and constraints because there are subtle power and influence relationships between the stakeholders in the organization. Published organizational structures rarely match the reality of decision making in an organization, but interviewees may not wish to reveal the actual rather than the theoretical structure to a stranger. In general, most people are reluctant to discuss political and organizational issues that may affect the requirements.

4.5.3 Scenarios

- Scenarios can be particularly useful for adding detail to an outline requirements description. They are descriptions of example interaction sessions. Each scenario usually covers one or a small number of possible interactions. Different forms of scenarios are developed and they provide different types of information at different levels of detail about the system.

- A scenario starts with an outline of the interaction. During the elicitation process, details are added to this to create a complete description of that interaction. At its most general, a scenario may include:

1. A description of what the system and users expects when the scenario starts.
2. A description of the normal flow of events in the scenario.
3. A description of what can go wrong and how this is handled.
4. Information about other activities that might be going on at the same time.
5. A description of the system state when the scenario finishes.

4.5.4 Use cases

- Use cases are a requirements discovery technique that were first introduced in the Objectory method (Jacobson et al., 1993). They have now become a fundamental feature of the unified modeling language. In their simplest form, a use case identifies the actors involved in an interaction and names the type of interaction. This is then supplemented by additional information describing the interaction with the system.

- Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements.

- Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse.

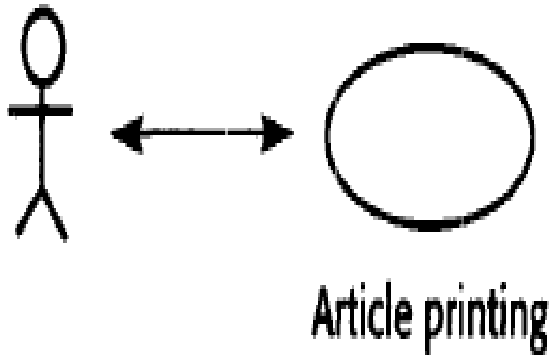
- Lines link the actors with the interaction.

- Scenarios and use cases are effective techniques for eliciting requirements from stakeholders who interact directly with the system.

- Each type of interaction can be represented as a use case. However, because they focus on interactions with the system, they are not as effective for eliciting constraints or high-level business and nonfunctional requirements or for discovering domain requirements.

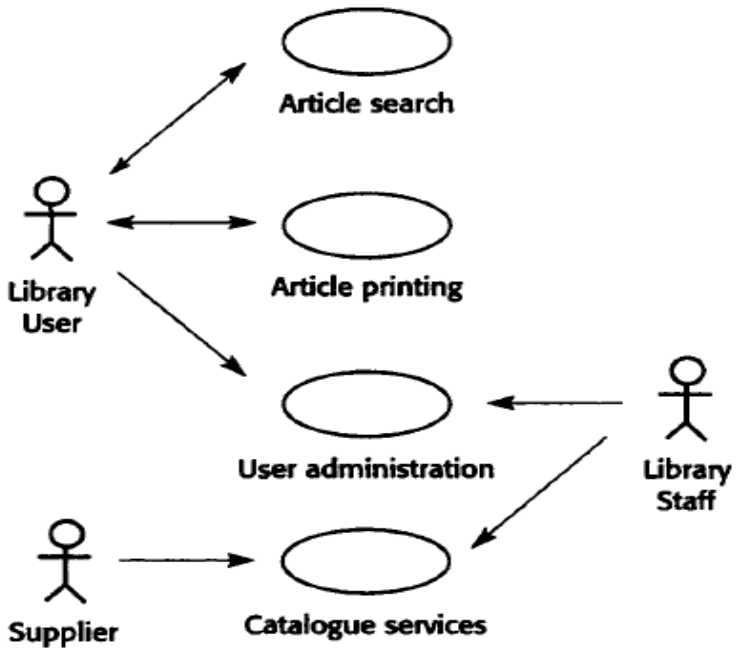
- Figure 7.6 illustrates the essentials of the use-case notation. Actors in the process are represented as stick figures, and each class of interaction is represented as a named ellipse. The set of use-cases represents all of the possible interactions to be represented in the system requirements.

Figure 7.6 A simple use-case for article printing



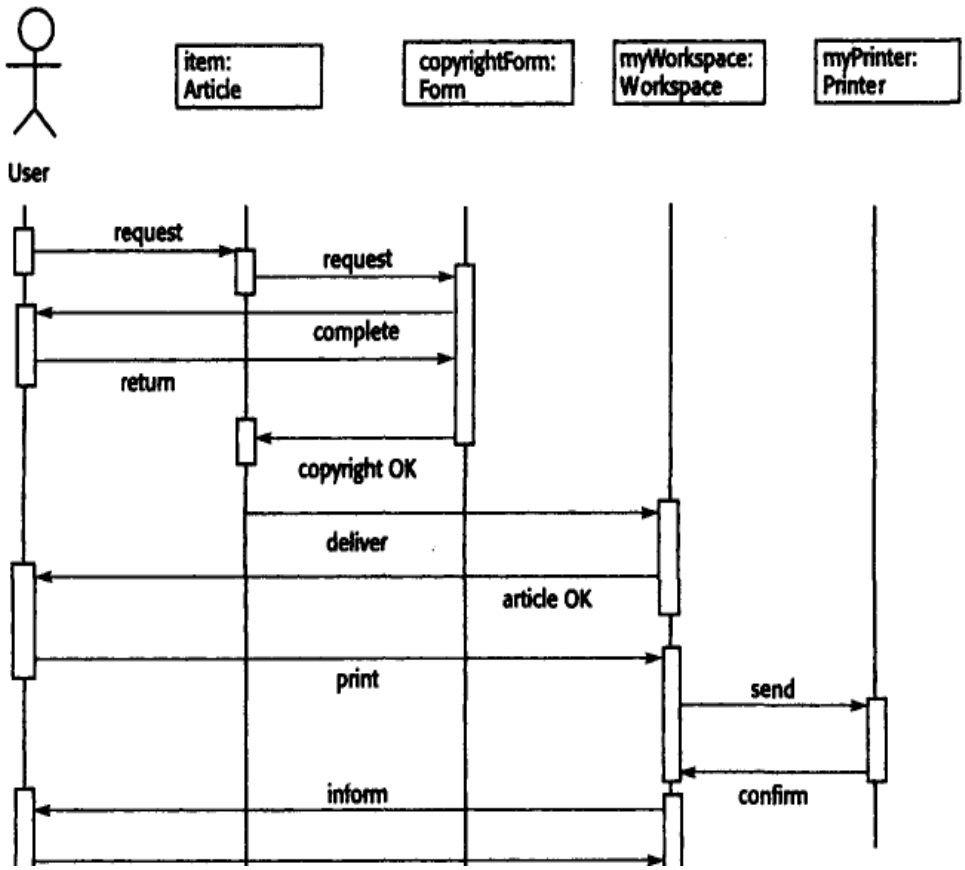
- Figure 7.7 develops the LIBSYS example and shows other use-cases in that environment.

Figure 7.7 Use cases for the library system



-

Figure 7.8 System interactions for article printing



- **4.5.5 Ethnography**

- Ethnography is an observational technique that can be used to understand operational processes and help derive support requirements for these processes. The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

- Suchman (1987) pioneered the use of ethnography to study office work. She found that the actual work practices were far richer, more complex, and more dynamic than the simple models assumed by office automation systems. The difference between the assumed and the actual work was the most important reason why these office systems had no significant effect on productivity

- Ethnography is particularly effective for discovering two types of requirements:

- 1. Requirements that are derived from the way in which people actually work, rather than the way in which process definitions say they ought to work.

- 2. Requirements that are derive d from cooperation and awareness of other people's activities.
- Ethnography can be combined with prototyping (Figure 4.16). The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer

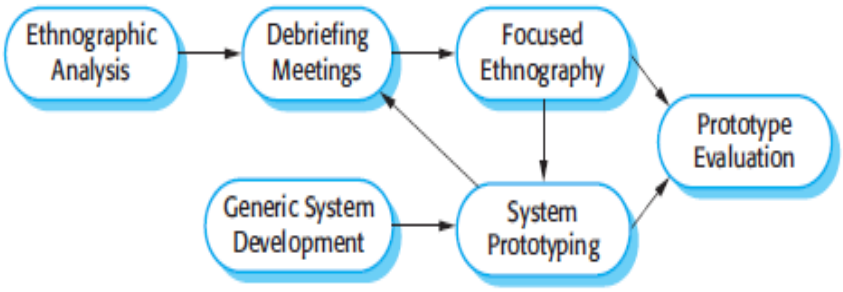


Figure 4.16 Ethnography and prototyping for requirements analysis

- 2.3.3 **Requirements validation**
- Requirements validation is the process of checking that requirements actually define the system that the customer really wants. Requirements validation is important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.
- During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

1. ***Validity checks*** A user may think that a system is needed to perform certain functions.

- 2. ***Consistency checks*** Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.
- 3. ***Completeness checks*** The requirements document should include requirements that define all functions and the constraints intended by the system user.

- 4. ***Realism checks*** Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.
- 5***. Verifiability*** To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable.
- There are a number of requirements validation techniques that can be used individually or in conjunction with one another:
- 1. ***Requirements reviews*** The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
- 2. ***Prototyping*** In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.
- 3***. Test-case generation*** Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered.
- **Requirements reviews**
- A requirements review is a manual process that involves people from both client and contractor organizations. They check the requirements document for anomalies and omissions. The review process may be managed in the same way as program Inspections.
- Requirements reviews can be informal or formal. 1)Informal reviews simply involve contractors discussing requirements with as many system stakeholders as possible.
- 2) In a formal requirements review, the development team should 'walk' the client through the system requirements, explaining the implications of each requirement.
- Reviewers may also check for:
- *1.* ***Verifiability*** Is the requirement as stated realistically testable?

- *2.* ***Comprehensibility*** Do the procurers or end-users of the system properly understand the requirement?
- *3.* ***Traceability*** Is the origin of the requirement clearly stated? You may have to go back to the source of the requirement to assess the impact of a change
- Traceability is important as it allows the impact of change on the rest of the system to be assessed. I discuss it in more detail in the following section.
- *4.* ***Adaptability*** Is the requirement adaptable? That is, can the requirement be changed without large-scale effects on other system requirements?
- 2.3.4 **Requirements management**
- The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address 'wicked' problems—problems that cannot be completely defined. Because the problem cannot be fully defined, the software requirements are bound to be incomplete. During the software process, the stakeholders' understanding of the problem is constantly changing (Figure 4.17). The system requirements must then also evolve to reflect this changed problem view

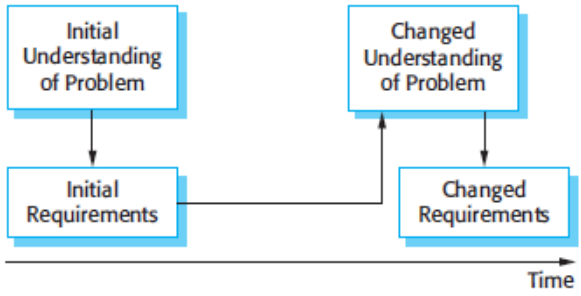


Figure 4.17 Requirements evolution

- **There are several reasons why change is inevitable:**
- 1. The business and technical environment of the system always changes after installation. New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.

- 2. The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.
- 3. Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory. The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.
- Requirements management is the process of understanding and controlling changes to system requirements.
- You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes.
- **a) Enduring and volatile requirements**
- Requirements evolution during the RE process and after a system has gone into service is inevitable. Developing software requirements focuses attention on software capabilities, business objectives and other business systems. As the requirements definition is developed, you normally develop a better understanding of users needs.
- This feeds information back to the user, who may then propose a change to the
- requirements (Figure 4.17). Furthermore, it may take several years to specify and
- develop a large system.
- **From an evolution perspective, requirements fall into two classes:**
- *I.* ***Enduring requirements*** These are relatively stable requirements that derive from the core activity of the organization and which relate directly to the domain of the system. For example, in a hospital, there will always be requirements concerned with patients, doctors, nurses and treatments. These requirements may be derived from domain models that show the entities and relations that characterize an application domain.

- *2.* ***Volatile requirements*** These are requirements that are likely to change during the system development process or after the system has been become operational. An example would be requirements resulting from government healthcare policies

- **Harker and others have suggested that volatile requirements fall into the classes.**

Figure 7.11 Classification of volatile requirements

Requirement Type	Description
Mutable requirements	Requirements which change because of changes to the environment in which the organisation is operating. For example, in hospital systems, the funding of patient care may change and thus require different treatment information to be collected.
Emergent requirements	Requirements which emerge as the customer's understanding of the system develops during the system development. The design process may reveal new emergent requirements.
Consequential requirements	Requirements which result from the introduction of the computer system. Introducing the computer system may change the organisation's processes and open up new ways of working which generate new system requirements.
Compatibility requirements	Requirements which depend on the particular systems or business processes within an organisation. As these change, the compatibility requirements on the commissioned or delivered system may also have to evolve.

-

- **b) Requirements management planning**

- Planning is an essential first stage in the requirements management process. The planning stage establishes the level of requirements management detail that is required. During the requirements management stage, you have to decide on:

- 1. ***Requirements identification*** Each requirement must be uniquely identified so

- that it can be cross-referenced with other requirements and used in traceability

- assessments.

- 2. ***A change management process*** This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.

- 3. ***Traceability policies*** These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.
- 4***. Tool support*** **Requirements** management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.
- There are three types of traceability information that may be maintained:
- 1. ***Source traceability*** information links the requirements to the stakeholders who proposed the requirements and to the rationale for these requirements. When a change is proposed, you use this information to find and consult the stakeholders about the change.
- *2.* ***Requirements traceability*** information links dependent requirements within the requirements document. You use this information to assess how many requirements are likely to be affected by a proposed change and the extent of consequential requirements changes that may be necessary.
- *3.* ***Design traceability*** information links the requirements to the design modules where these requirements are implemented. You use this information to assess the impact of proposed requirements changes on the system design and implementation.
- Traceability information is often represented using traceability matrices, which relate requirements to stakeholders, each other or design modules. In a requirements traceability matrix, each requirement is entered in a row and in a column in the matrix.
- Figure 7.12 shows a simple traceability matrix that records the dependencies between requirements. A 'D' in the row/column intersection illustrates that the requirement in the row depends on the requirement named in the column; an 'R' means that there is some other, weaker relationship between the requirements.
- Traceability matrices may be used when a small number of requirements have to be managed, but they become unwieldy and expensive to maintain for large systems with many requirements.

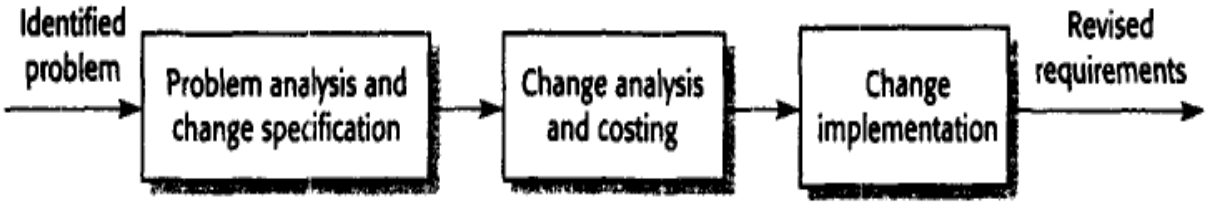
Figure 7.12
A traceability matrix

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		D	R					
1.2			D			R		D
1.3	R			R				
2.1			R		D			D
2.2								D
2.3		R		D				
3.1								R
3.2							R	

- Requirements management needs automated support and the software tools for this should be chosen during the planning phase. You need tool support for:
- 1. *Requirements storage* The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
- 2. *Change management* The process of change management (Figure 7.13) is simplified if active tool support is available.
- 3. *Traceability management* As discussed above, tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements. For small systems, it may not be necessary to use specialized requirements management tools.
- **4.7.2 Requirements change management**
- Requirements change management (Figure 7.13) should be applied to all proposed changes to a system's requirements after the requirements document has been approved. Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation. The advantage of using a formal

process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way. There are three principal stages to a change management process:

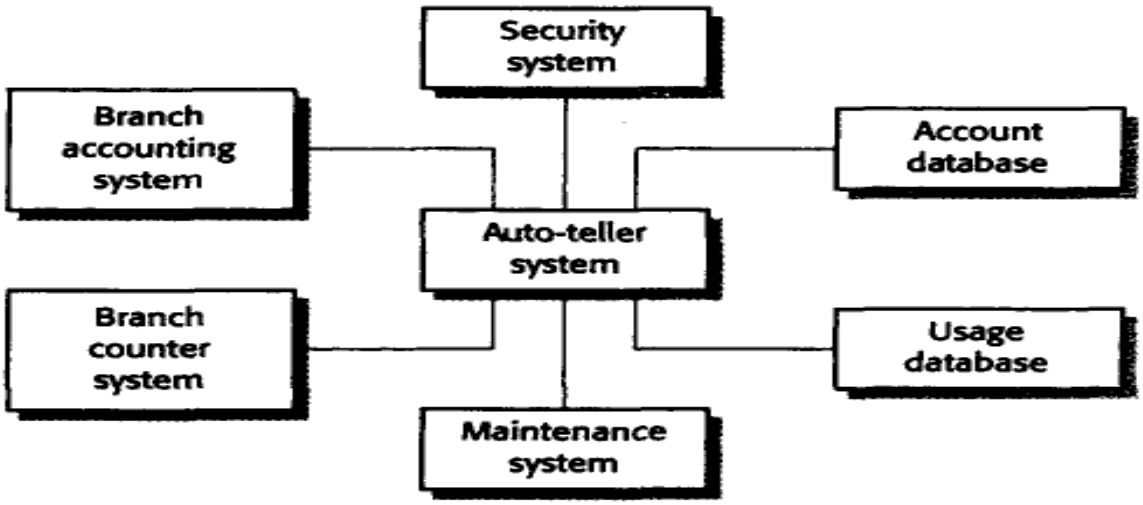
- 1. ***Problem analysis and change specification*** The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
- 2. ***Change analysis and costing*** The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated both in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
- 3. ***Change implementation*** The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization
- Figure 4.18 Requirements Change Management



-
- 2.4.1 **System Models**
- User requirements should be written in natural language because they have to be understood by people who are not technical experts. However, more detailed system requirements may be expressed in a more technical way.

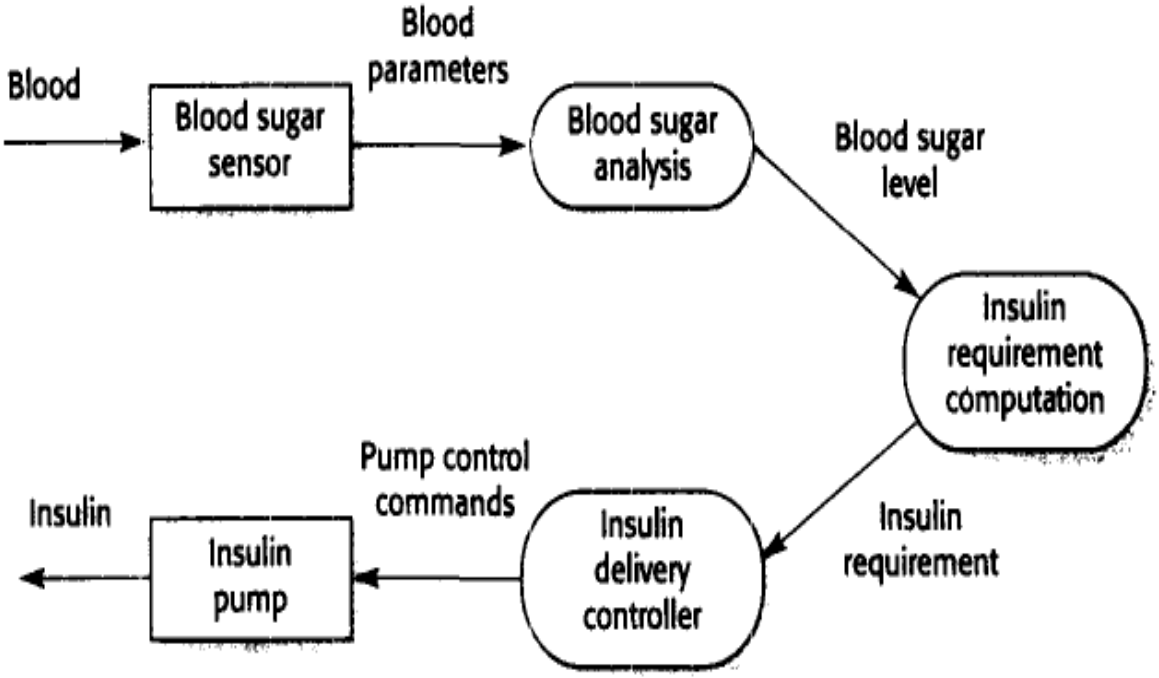
- One widely used technique is to document the system specification as a set of system models. These models are graphical representations that describe business processes, the problem to be solved and the system that is to be developed.
- You can use models in the analysis process to develop an understanding of the
- existing system that is to be replaced or improved or to specify the new system that is required. You may develop different models to represent the system from different perspectives. For example:
- 1. An external perspective, where the context or environment of the system is modeled
- 2. A behavioral perspective, where the behavior of the system is modeled
- 3. A structural perspective, where the architecture of the system or the structure of the data processed by the system is modeled
- A system model is an abstraction of the system being studied rather than an alternative representation of that system. Ideally, a *representation* of a system should maintain all the information about the entity being represented.
- An *abstraction* deliberately simplifies and picks out the most salient characteristics.
- Different types of system models are based on different approaches to abstraction.
- A data-flow model (for example) concentrates on the flow of data and the functional transformations on that data.
- Examples of the types of system models that you might create during the analysis process are:
- 1. *A data- flow model* Data-flow models show how data is processed at different stages in the system.
- *2. A composition model* A composition or *aggregation* model shows how entities in the system are composed of other entities.

- *3. An architectural model* Architectural models show the principal sub-systems that make up a system.
- *4. A classification model* Object class/inheritance diagrams show how entities have common characteristics.
- *5. A stimulus-response model* A stimulus-response model, or *state transition diagram,* shows how the system reacts to internal and external events.
- 2.4.2 (1) **Context models**
- At an early stage in the requirements elicitation and analysis process you should decide the boundaries of the system. This involves working with system stakeholders to distinguish what is the system and what is the system's environment.
- You should make these decisions early in the process to limit the system costs and the time needed for analysis.
- In some cases, the boundary between a system and its environment is relatively clear.
- For example, where an automated system is replacing an existing manual or computerized system, the environment of the new system is usually the same as the existing system's environment. In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.
- Figure 8.1 is an architectural model that illustrates the structure of the information system that includes a bank auto-teller network. High-level architectural models are usually expressed as simple block diagrams where each sub-system is represented by a named rectangle, and lines indicate associations between sub-systems. From Figure 8.1, we see that each ATM is connected to an account database, a local branch accounting system, a security system and a system to support machine maintenance. The system is also connected to a usage database that monitors how the network of ATMs is used and to a local branch counter system. This counter system provides services such as backup and printing. These, therefore, need not be included in the ATM system itself.



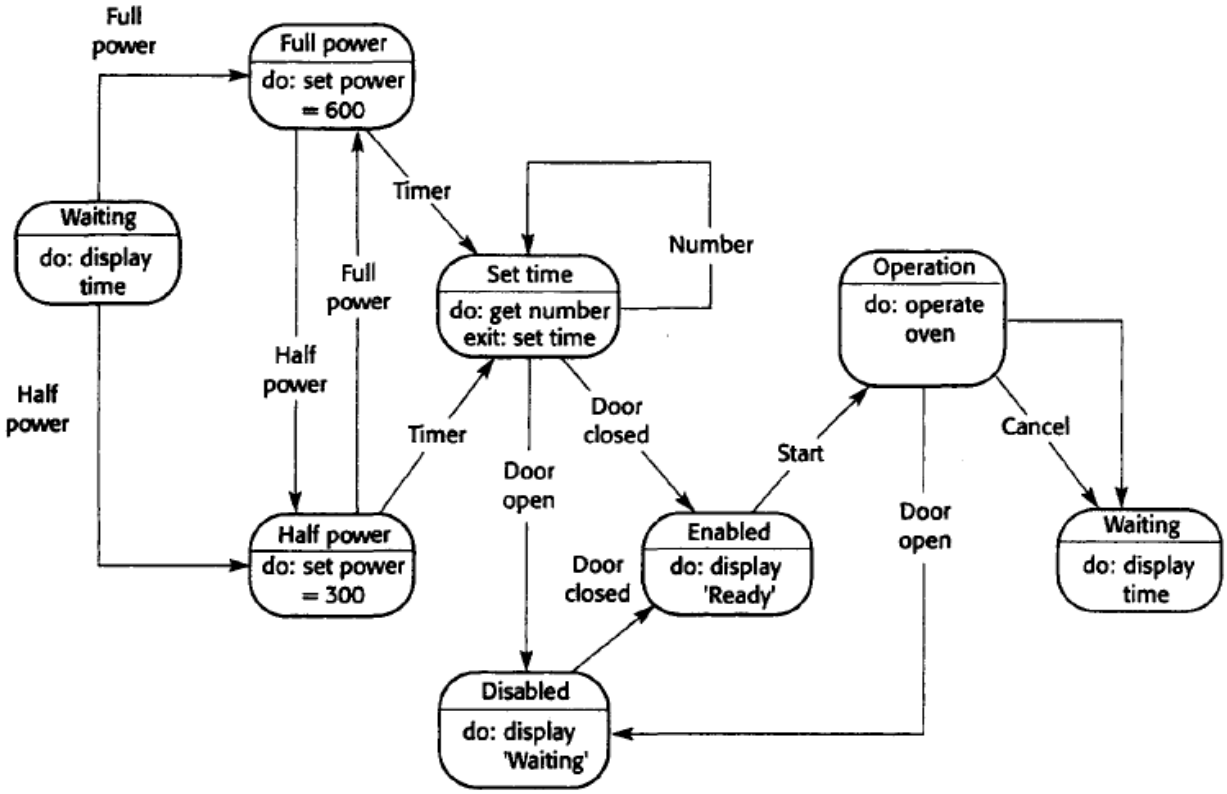
- 2.4.3 (2) **Behavioral models**
- Behavioral models are used to describe the overall behavior of the system.
- Two types of behavioral model here*:* ***data-flow models,*** which model the data processing in the system, and ***state machine models,*** which model how the system reacts to events. These models may be used separately or together, depending on the type of system that is being developed.
- A ***dataflow model*** may be all that is needed to represent the behavior of these systems.
- A ***state machine model*** is the most effective way to represent their behavior.
- 2.1 **Data flow models**
- Data-flow models are an intuitive way of showing how data is processed by a system. At the analysis level, they should be used to model the way in which data is processed in the existing system.
- The use of data-flow models for analysis became widespread after the publication of DeMarco's book on structured systems analysis. They are an intrinsic part of structured methods that have been developed from this work.
- The notation used in these models represents functional processing (rounded rectangles), data stores (rectangles) and data movements between functions (labelled arrows).

- Data-flow models are used to show how data flows through a sequence of processing steps.
- Data-flow models show a functional perspective where each transformation represents a single function or process. They are particularly useful during the analysis of requirements as they can be used to show end-to- end processing in a system.
- That is, they show the entire sequence of actions that take place from an input being processed to the corresponding output that is the system's response.
- Figure 8.4 illustrates this use of data flow diagrams. It is a diagram of the processing that takes place in the insulin pump system.
- Data-Model diagram of an insulin pump.Fig 8.4



- **2.2 State machine models**
- A state machine model describes how a system responds to internal or external events.
- The state machine model shows system states and events that cause transitions from one state to another. It does not show the flow of data within the system.

- This type of model is often used for modeling real-time systems because these systems are often driven by stimuli from the system's environment.
- This approach to system modeling is illustrated in Figure 8.5. This diagram shows a state machine model of a simple microwave oven equipped with buttons to set the power and the timer and to start the system.
- Real microwave ovens are actually much more complex than the system described here. However, this model includes the essential features of the system.
- **Fig: 8.5 This diagram shows a state machine model of a simple microwave oven equipped with buttons to set the power and the timer and to start the system.**



- To simplify the model, assume that the
- sequence of actions in using the microwave is:
- 1. Select the power level (either half-power or full-power).
- 2. Input the cooking time.

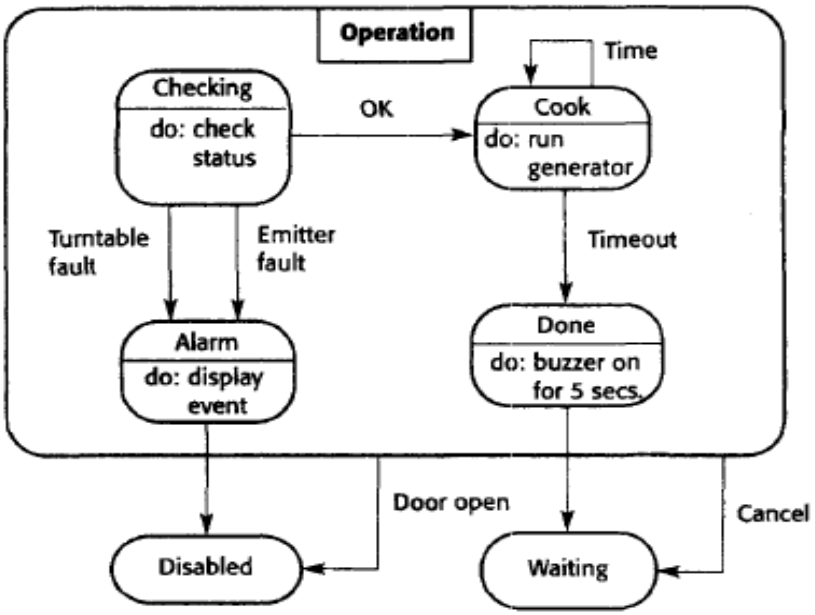
- 3. Press Start, and the food is cooked for the given time.
- For safety reasons, the oven should not operate when the door is open and, on completion of cooking, a buzzer is sounded. The oven has a very simple alphanumeric display that is used to display various alerts and warning messages
- The UML notation describe state machine models is designed for modeling the behavior of objects. However, it is a general-purpose notation that can be used for any type of state machine modeling. The rounded rectangles in a model represent system states.
- Therefore, from Figure 8.5, we can see that the system responds initially to either the full-power or the half-power button.
- Users can change their mind after selecting one of these and press the other button. The time is set and, if the door is closed, the Start button is enabled. Pushing this button starts the oven operation and cooking takes place for the specified time.
- In a detailed system specification, you have to provide more detail about both the stimuli and the system states (Figure 8.6).
- Figure 8.6

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power .
Full power	The oven power is set to 600 watts. The display shows 'Full power .
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

Stimulus	Description
Half power	The user has pressed the half power button.
Full power	The user has pressed the full power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the start button.
Cancel	The user has pressed the cancel button.

- The problem with the state machine approach is that the number of possible states increases rapidly. For large system models, therefore, some structuring of these state models is necessary. One way to do this is by using the notion of a super state that encapsulates a number of separate states. This super state looks like a single state on a high-level model but is then expanded in more detail on a separate diagram.
- To illustrate this concept, consider the Operation state in Figure 8.5. This is a super state that can be expanded, as illustrated in Figure: 8.7.
- The Operation state includes a number of sub-states. It shows that operation starts with a status check, and that if any problems are discovered, an alarm is indicated and operation is disabled. Cooking involves running the microwave generator for the specified time; on completion, a buzzer is sounded. If the door is opened during operation, the system moves to the disabled state, as shown in Figure 8.5.

Figure 8.7 Microwave oven operation



- **2.4.2 (3) Data Models**
- An important point of systems modeling is defining the logical form of the data processed by the system. These are sometimes called ***semantic data models.***
- The most widely used data modeling technique is Entity-Relation-Attribute modeling (ERA modeling), which shows the data entities, their associated attributes and the relations

between these entities. This approach to modeling was first proposed in the mid-1970s by Chen.

- Entity-relationship models have been widely used in database design. The relational database schemas derived from these models are naturally in third normal form, which is a desirable characteristic.
- The UML does not include a specific notation for this database modeling, as it assumes an object-oriented development process and models data using objects and their relationships. However, you can use the UML to represent a semantic data model.
- Figure 8.8 is an example of a data model that is part of the library system LIBSYS. Figure shows that an Article has attributes representing the title, the authors, the name of the PDF file of the article and the fee payable. This is linked to the Source, where the article was published, and to the Copyright Agency for the country of publication. Both Copyright Agency and Source are linked to Country. The country of publication is important because copyright laws vary by country. The diagram also shows that Buyers place Orders for Articles.

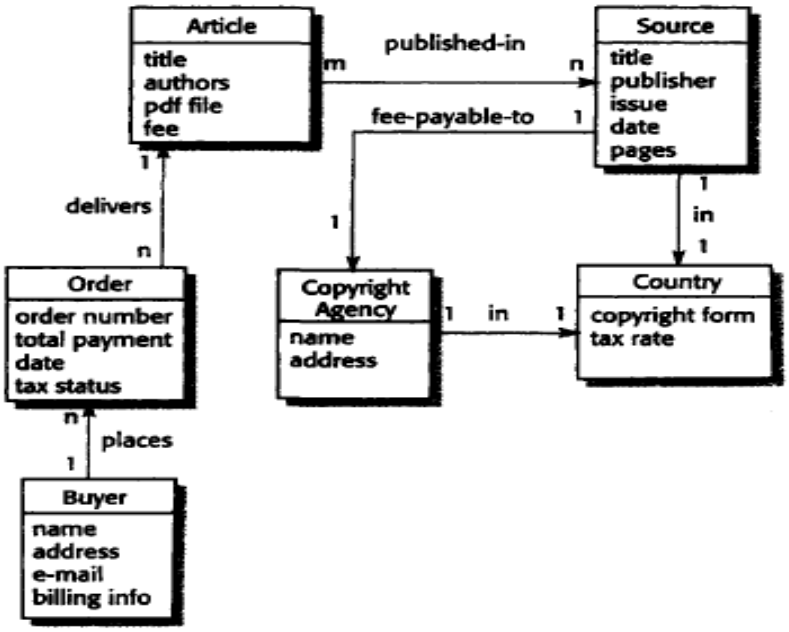


Figure 8.8 Semantic data model for the LIBSYS system

- A data dictionary is. simplistically, an alphabetic list of the names included in the system models. As well as the name, the dictionary should include an associated description of the named entity and, if the name represents a composite object, a description of the

composition. Other information such as the date of creation. the creator and the representation of the entity may also be included depending on the type of model being developed.

The advantages of using a data dictionary are:

- 1. ***It is a mechanism for name management.*** Many people may have to invent names for entities and relationships when developing a large system model. These names should be used consistently and should not clash. The data dictionary software can check for name uniqueness where necessary and warn requirements analysts of name duplications.

- *2.* ***It serves as a store of organizational information.*** As the system is developed, information that can link analysis, design, implementation and evolution is added to the data dictionary, so that all information about an entity is in one place.

- The data dictionary entries shown in Figure 8.9 define the names in the semantic data model for LIBSYS.

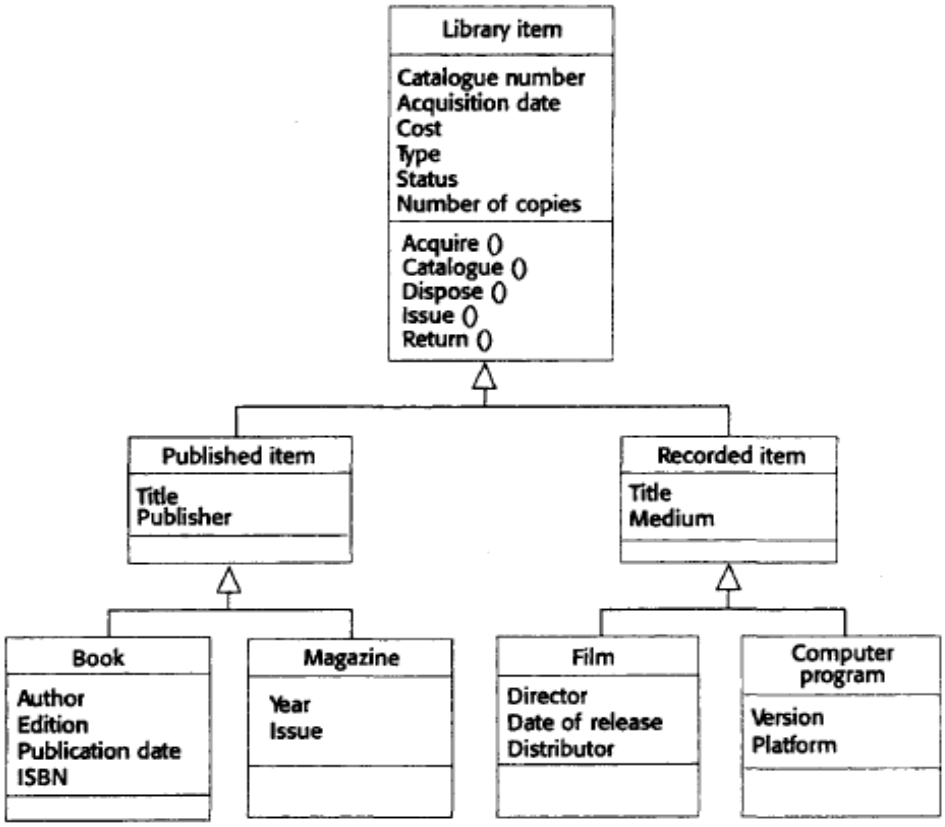
- All system names, whether they are names of entities, relations, attributes or services. should be entered in the dictionary. Software is normally used to create, maintain and interrogate the dictionary.

Figure 8.9 Examples of data dictionary entries

Name	Description	Type	Date
Article	Details of the published article that may be ordered by people using LIBSYS.	Entity	30.12.2002
authors	The names of the authors of the article who may be due a share of the fee.	Attribute	30.12.2002
Buyer	The person or organisation that orders a copy of the article.	Entity	30.12.2002
fee-payable-to	A 1:1 relationship between Article and the Copyright Agency who should be paid the copyright fee.	Relation	29.12.2002
Address (Buyer)	The address of the buyer. This is used to any paper billing information that is required.	Attribute	31.12.2002

- 2.4.5 (**4) Object models**
- An object-oriented approach to the whole software development process is now commonly used, particularly for interactive systems development. This means expressing the systems requirements using an object model, designing using objects and developing the system in an object-oriented programming language such as Java or C++.
- Object models that you develop during requirements analysis may be used to represent both system data and its processing.
- Developing object models during requirements analysis usually simplifies the transition to object-oriented design and programming.
- An object class is an abstraction over a set of objects that identifies common attributes (as in a semantic data model) and the services or operations that are provided by each object.
- Objects are executable entities with the attributes and services of the object class.
- Objects are instantiations of the object class, and many objects may be created from a class.
- Generally, the models developed using analysis focus on object classes and their relationships.
- An object class in UML, as illustrated in the examples in Figure 8.10, is represented as a vertically oriented rectangle with three sections:
- I. The name of the object class is in the top section.
- 2. The class attributes are in the middle section.
- 3. The operations associated with the object class are in the lower section of the rectangle.

Figure 8.10 Part of a class hierarchy for a library



a) Inheritance Models

- Figure 8.10 illustrates part of a simplified class hierarchy for a model of a library. This hierarchy gives information about the items held in the library. The library holds various items, such as books, music, recordings of films, magazines and newspapers.

- Figure 8.11 is an example of another inheritance hierarchy that might be part of the library model. In this case, the users of a library are shown. There are two classes of user:

a) those who are allowed to borrow books, and

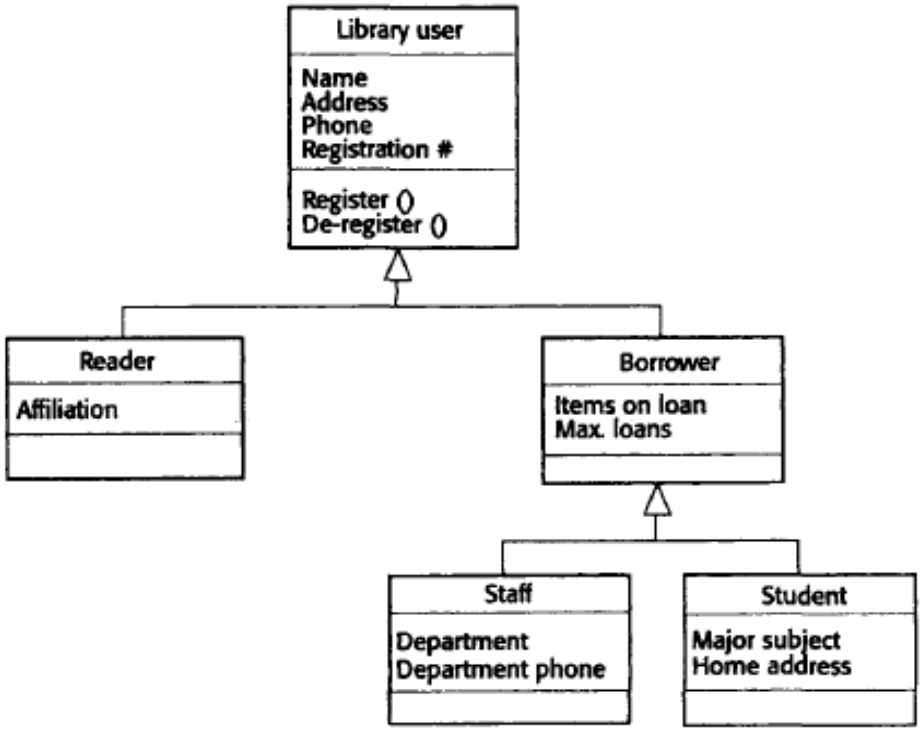
b)those who may only read books in the library without taking them away.

- In the UML notation, inheritance is shown upwards' rather than 'downwards‘ as it is in some other object-oriented notations or in languages such as Java, where sub-classes inherit from super-classes.

- That is, the arrowhead (shown as a triangle) points from the classes that inherit attributes and operations to the super-class. Rather than use the term *inheritance*, UML refers to the *generalization relationship.*

- **Figure 8.10 and Figure 8.11** show class inheritance hierarchies where every object class inherits its attributes and operations from a single parent class. Multiple inheritance models may also be constructed where a class has several parents. Its inherited attributes and services are a conjunction of those inherited from each super-class.

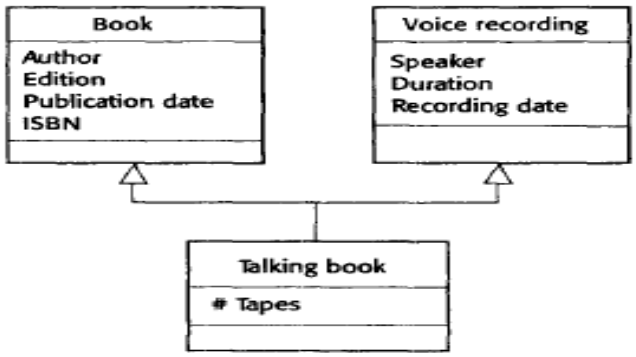
Figure 8.11 User class hierarchy



- Figure 8.12 shows an example of a multiple inheritance model that may also be part of the library model. The main problem with multiple inheritance is designing an inheritance graph where objects do not inherit unnecessary attributes.

- Other problems include the difficulty of reorganizing the inheritance graph when changes are required and resolving name clashes where attributes of two or more super-classes have the same name but different meanings.

Figure 8.12 Multiple inheritance

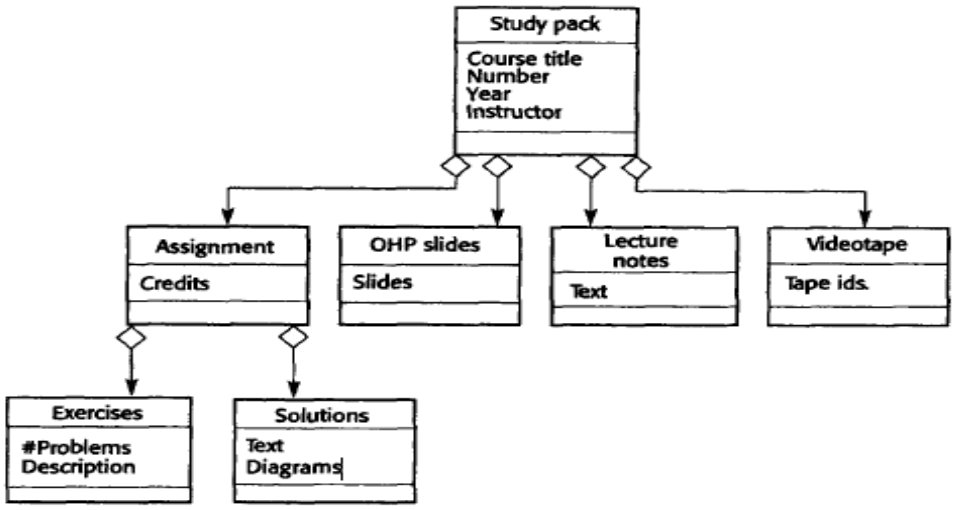


-
-

b) Object aggregation

Acquiring attributes and services through an inheritance relationship with other objects, some objects are groupings of other objects. That is, an object is an *aggregate* of a set of other objects. The classes representing these objects may be modeled using an object aggregation model, as shown in Figure 8.13.

Figure 8.13 Aggregate object representing a course



- Figure 8.13 could be maintained electronically and downloaded to the student's computer. In a sequence diagram (8.14), objects and actors are aligned along the top of the diagram. Labeled arrows indicate operations; the sequence of operations is from top to bot tom. In I:his scenario, the library user accesses the catalogue to see whether the item required is available electronically; if it is, the user requests the electronic issue of that item.

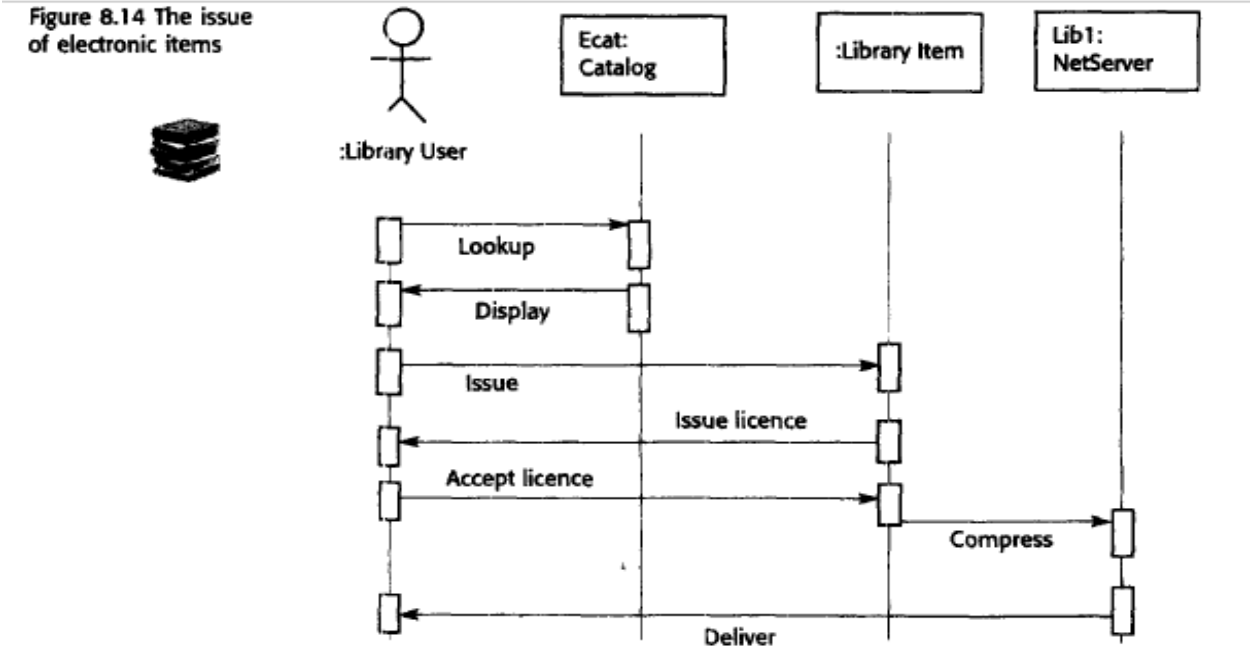


Figure 8.14 The issue of electronic items

2.4.6 (**5) Structured Methods**

- A structured method is a systematic way of producing models of an existing system or of a system that is to be built.
- They were first developed in the 1970s to support software analysis and design(Constantine and Yourdon, 1979; Gane and Sarson, 1979; Jackson, 1983) and evolved in the 1980s and 1990s to support object-oriented development.
- Structured methods provide a framework for detailed system modeling as part of requirements elicitation and analysis. Most structured methods have their own preferred set of system models.
- They usually define a process that may be used to derive these models and a set of rules and guidelines that apply to the models. Standard documentation is produced for the system. CASE tools are usually available for method support.
- These tools support model editing and code and report generation, and provide some model-checking capabilities.
- Structured methods have been applied successfully in many large projects.
- However, structured methods suffer from a number of weaknesses:

I. They do not provide effective support for understanding or modeling nonfunctional system requirements.

2. They are indiscriminate in that they do not usually include guidelines to help users decide whether a method is appropriate for a particular problem. Nor do they normally include advice on how they may be adapted for use in a particular environment.

3. They often produce too much documentation. The essence of the system requirements may be hidden by the mass of detail that is included.

4. The models that are produced are very detailed, and users often find them difficult to understand. These users therefore cannot check the realism of these models.

- Analysis and design CASE tools support the creation, editing and analysis of the graphical notations used in structured methods. Figure 8.15 shows the components that may be included method support environment.

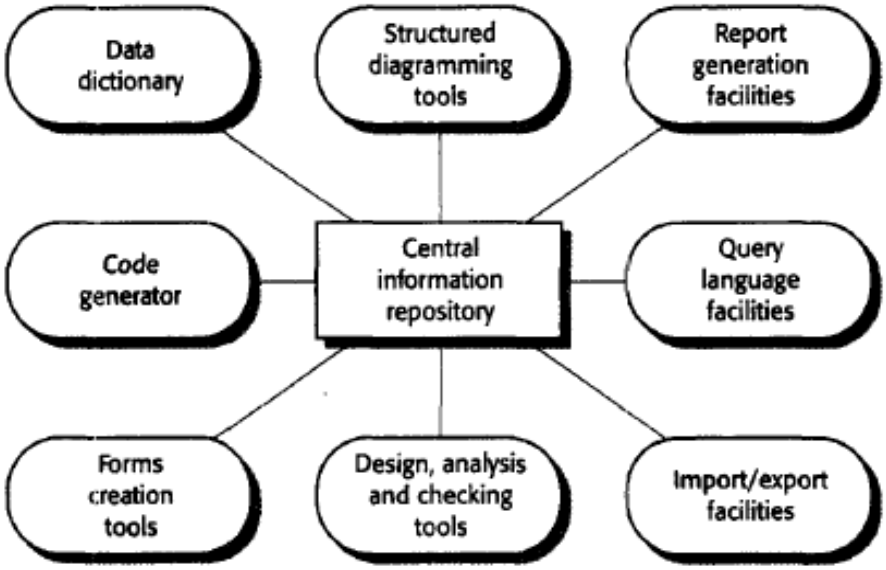


Figure 8.15 The components of a CASE tool for structured method support

- **Comprehensive method support tools, as illustrated in Figure 8.15, normally include**:

1. ***Diagram editors*** used to create object models, data models, behavioral models, and so on. These editors are not just drawing tools but are aware of the types of entities in the diagram. They capture information about these entities and save this information in the central repository.

2. ***Design analysis and checking tools*** that process the design and report on error and anomalies. These may be integrated with the editing system so that user errors are trapped at an early stage in the process.

3. ***Repository query languages*** that allow the designer to find designs and associated design information in the repository.

4. A ***data dictionary*** that maintains information about the entities used in a system design.

5. ***Report definition and generation tools*** that take information from the central store and automatically generate system documentation.

6. ***Forms definition tools*** that allow screen and document formats to be specified.

7. ***Import/export facilities*** that allow the interchange of information from the central repository with other development tools.

8. ***Code generators*** that generate code or code skeletons automatically from the design captured in the central store.

UNIT – III

Design concepts: Design Concepts, **Architectural Design**: Architectural design decisions, System organization, Modular decomposition styles.

Object-Oriented design: Objects and Object Classes, An Object-Oriented design process, Design Evolution.

3.1 (1) Software process designing concepts

Introduction to design process

- The main aim of design engineering is to generate a model which shows firmness, delight and commodity.
- Software design is an iterative process through which requirements are translated into the blueprint for building the software.

Software quality guidelines

- A design is generated using the recognizable architectural styles and compose a good design characteristic of components and it is implemented in evolutionary manner for testing.
- A design of the software must be modular i.e the software must be logically partitioned into elements. In design, the representation of data , architecture, interface and components should be distinct.
- A design must carry appropriate data structure and recognizable data patterns. Design components must show the independent functional characteristic.
- A design creates an interface that reduce the complexity of connections between the components. A design must be derived using the repeatable method. The notations should be use in design which can effectively communicates its meaning.

Quality attributes

The attributes of design name as 'FURPS' are as follows:

1)Functionality: It evaluates the feature set and capabilities of the program.

2)Usability: It is accessed by considering the factors such as human factor, overall aesthetics, consistency and documentation.

3)Reliability: It is evaluated by measuring parameters like frequency and security of failure, output result accuracy, the mean-time-to-failure(MTTF), recovery from failure and the program predictability.

4)Performance: It is measured by considering processing speed, response time, resource consumption, throughput and efficiency.

5)Supportability: It combines the ability to extend the program, adaptability, serviceability. These three term defines the maintainability.

- Testability, compatibility and configurability are the terms using which a system can be easily installed and found the problem easily.
- Supportability also consists of more attributes such as compatibility, extensibility, fault tolerance, modularity, reusability, robustness, security, portability, scalability.

3.1(2) Design concepts

The set of fundamental software design concepts are as follows:

1. Abstraction

- A solution is stated in large terms using the language of the problem environment at the highest level abstraction.
- The lower level of abstraction provides a more detail description of the solution.
- A sequence of instruction that contain a specific and limited function refers in a procedural abstraction.
- A collection of data that describes a data object is a data abstraction.

2. Architecture

- The complete structure of the software is known as software architecture.
- Structure provides conceptual integrity for a system in a number of ways.
- The architecture is the structure of program modules where they interact with each other in a specialized way.
- The components use the structure of data.
- The aim of the software design is to obtain an architectural framework of a system.
- The more detailed design activities are conducted from the framework.

Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:

- **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those

components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

- **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- **Families of related systems**. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

3. Patterns

A design pattern describes a design structure and that structure solves a particular design problem in a specified content.

4. Separation of Concerns

Separation of Concerns is a design concept [Dij82] that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.

- A concern is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

5. Modularity

- A software is separately divided into name and addressable components. Sometime they are called as modules which integrate to satisfy the problem requirements.
- Modularity is the single attribute of a software that permits a program to be managed easily.

6. Information hiding

Modules must be specified and designed so that the information like algorithm and data presented in a module is not accessible for other modules not requiring that information.

7. Functional independence

- The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.
- The functional independence is accessed using two criteria i.e Cohesion and coupling.

a)Cohesion

- Cohesion is an extension of the information hiding concept.
- A cohesive module performs a single task and it requires a small interaction with the other components in other parts of the program.

b)Coupling

Coupling is an indication of interconnection between modules in a structure of software.

8. Refinement

- Refinement is a top-down design approach.
- It is a process of elaboration.
- A program is established for refining levels of procedural details.
- A hierarchy is established by decomposing a statement of function in a stepwise manner till the programming language statement are reached.

9.Aspect

- An aspect is a representation of a crosscutting concern.

10. Refactoring

- It is a reorganization technique which simplifies the design of components without changing its function behaviour.
- Refactoring is the process of changing the software system in a way that it does not change the external behaviour of the code still improves its internal structure.

11.Object-Oriented Design Concepts The object-oriented (OO) paradigm is widely used in modern software engineering. Appendix 2 has been provided for those readers who may be unfamiliar with OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.

12. Design classes

- The model of software is defined as a set of design classes.
- Every class describes the elements of problem domain and that focus on features of the problem which are user visible.
- **Five different types of design classes**, each representing a different layer of the design architecture, can be developed :

1) ***User interface classes*** define all abstractions that are necessary for human computer interaction (HCI). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.

•2)**Business domain classes** are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.

3) **Process classes** implement lower-level business abstractions required to fully manage the business domain classes.

•4)**Persistent classes** represent data stores (e.g., a database) that will persist beyond the execution of the software.

•5)**System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Arlow and Neustadt suggest that each design class be reviewed to ensure that it is "well-formed." They define four characteristics of a well-formed design class:

1) Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class. For example, the class Scene defined for video-editing software is complete only if it contains all attributes and methods that can reasonably be associated with the creation of a video scene. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

2) Primitiveness Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing. For example, the class VideoClip for video-editing software might have attributes start-point and end-point to indicate the start and end points of the clip (note that the raw video loaded into the system may be longer than the clip that is used). The methods, setStartPoint() and setEndPoint(), provide the only means for establishing start and end points for the clip.

3)High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class VideoClip might contain a set of methods for editing the video clip. As long as each method focuses

solely on attributes associated with the video clip, cohesion is maintained.

4) **Low coupling.** Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the Law of Demeter [Lie03], suggests that a method should only send messages to methods in neighboring classes.

3.2 (1) Architectural design

Large systems are always decomposed into sub-systems that provide some related set of services. The initial design process of identifying these sub-systems and establishing a framework for sub-system control and communication is called architectural design. The output of this design process is a description of the software architecture.

In the model presented, architectural design is the first stage in the design process and represents a critical link between the design and requirements engineering processes. The architectural design process is concerned with establishing a basic structural framework that identifies the major components of a system and the communications between these components.

- Bass et al. (Bass, et al., 2(03) discuss three advantages of explicitly designing

and documenting a software architecture:

I. ***Stakeholder communication*** The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.

2. ***System analysis*** Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether the system can meet critical requirements such as performance, reliability and maintainability.

3. ***Large-scale reuse*** A system architecture model is a compact, manageable description of how a system is organized and how the components interoperate.

- The system architecture is often the same for systems with similar requirements

and so can support large-scale software reuse. It may be possible to develop product-line architectures where the same architecture is used across a range of related systems.

- The system architecture affects the performance, robustness, distributability and maintainability of a system (Bosch). The particular style and structure chosen for an application may therefore depend on the non-functional system requirements:

1. *Performance* If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of subsystems, with as little communication as possible between these sub-systems. This may mean using relatively large-grain rather than fine-grain components to reduce component

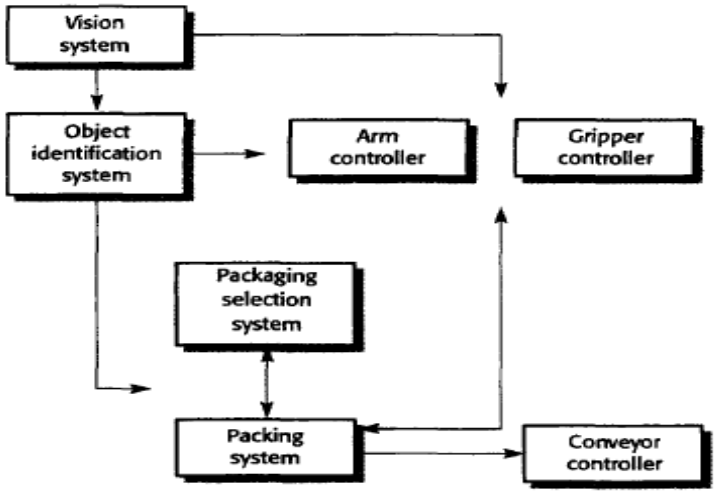
2. ***Security*** If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers and with a high level of security validation applied to these layers.

3. ***Safety*** If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single sub-system or in a small number of sub-systems. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems.

4. Availability If availability is a critical requirement, the architecture should be designed to include redundant components and so that it is possible to replace and update components without stopping the system. Fault-tolerant system architectures for high-availability systems are covered .

5. Maintainability If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed. Producers of data should be separated from consumers and shared data structures should be avoided.

- For example, Figure 11.1 is an abstract model of the architecture for a packing robot system that shows the sub-systems that have to be developed. This robotic system can pack different kinds of object. It uses a vision sub-system to pick out objects on a conveyor, identify the type of object and select the right kind of packaging. The system then moves objects from the delivery conveyor to be packaged. It places packaged objects on another conveyor.



3.2(2)Architectural design decisions

- Architectural design is a creative process where you try to establish a system organization that will satisfy the functional and non-functional system requirements.
- Because it is a creative process, the activities within the process differ radically depending on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system.
- It is therefore more useful to think of the architectural design process from a decision perspective rather than from an activity perspective. During the architectural design process, system architects have to make a number of fundamental decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to answer the following fundamental questions:

1**. Is there a generic application architecture that can act as a template for the system that is being designed?**

2. How will the system be distributed across a number of processors?

3. What architectural style or styles are appropriate for the system?

4. What will be the fundamental approach used to structure the system?

5. How will the structural units in the system be decomposed into modules?

6. What strategy will be used to control the operation of the units in the system?

7. How will the architectural design be evaluated?

8. How should the architecture of the system be documented?

- Evaluating an architectural design is difficult because the true test of an architecture is in how well it meets its functional and non-functional requirements after it has been deployed
- However, in some cases, you can do some evaluation by comparing your design against reference or generic architectural models.
- The graphical models of the system present different perspectives on the architecture.

Architectural models that may be developed may include:

- I. A ***static structural model*** that shows the sub-systems or components that are to be developed as separate units.
- 2. A ***dynamic process model*** that shows how the system is organised into processes at run-time. This may be different from the static model.
- 3. An ***interface model*** that defines the services offered by each sub-system through its public interface.
- *4.* ***Relationship models*** that shows relationships, such as data flow, between the sub-systems.
- 5. A ***distribution model*** that shows how sub-systems may be distributed across computers.

3.2(3) System Organization

- The organization of a system reflects the basic strategy that is used to structure a system. The system organization may be directly reflected in the sub-system structure.
- However, it is often the case that the sub-system model includes more detail than the organizational model, and there is not always a simple mapping from sub-systems to organizational structure

1) The repository model

- Sub-systems making up a system must exchange information so that they can work together effectively. There are two fundamental ways in which this can be done.

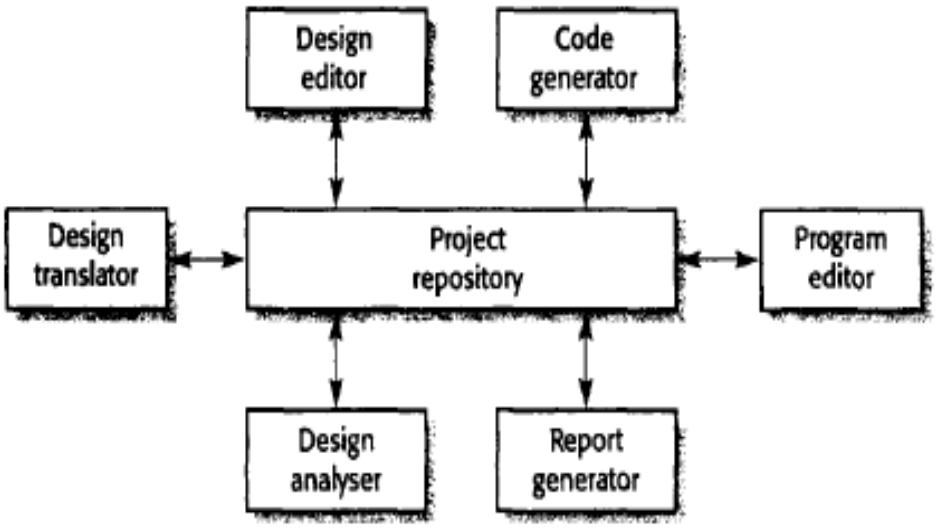
1. All shared data is held in a central database that can be accessed by all subsystems.

- A system model based on a shared database is sometimes called a *repository model.*

2. Each sub-system maintains its own database. Data is interchanged with other sub-systems by passing messages to them.

- The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications where data is generated by one sub-system and used by another.
- Examples of this type of system include command and control systems, management information systems, CAD systems and CASE toolsets.
- Figure 11.2 is an example of a CASE toolset architecture based on a shared repository.

Figure 11.2 The architecture of an integrated CASE toolset



The advantages and disadvantages of a shared repository are as follows:

1. It is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one sub-system to another.

2. However, sub-systems must agree on the repository data model. Inevitably, this is a compromise between the specific needs of each tool. Performance may be adversely affected by this compromise. It may be difficult or impossible to integrate new sub-systems if their data models do not fit the agreed schema.

3. Sub-systems that produce data need not be concerned with how that data is used by other sub-systems.

4. However, evolution may be difficult as a large volume of information is generated according to an agreed data model. Translating this to a new model will certainly be expensive; it may be difficult or even impossible.

5. Activities such as backup, security, access control and recovery from error are centralized. They are the responsibility of the repository manager. Tools can focus on their principal function rather

than be concerned with these issues.

6. However, different sub-systems may have different requirements for security, recovery and backup policies. The repository model forces the same policy on all sub-systems.

7. The model of sharing is visible through the repository schema. It is straightforward to integrate new tools given that they are compatible with the agreed data model.

8. However, it may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, there may be problems with data redundancy and inconsistency.

2. **The client-server mode**l

- The client-server architectural model is a system model where the system is organized as set of services and associated servers and clients that access and use the services. The major components of this model are:

I. A set of servers that offer services to other sub-systems. Examples of servers are print servers that offer printing services, file servers that offer file management services and a compile server, which offers programming language compilation services.

2. A set of clients that call on the services offered by servers. These are normally sub- systems In their own right. There may be several instances of a client program executing concurrently.

3. A network that allows the clients to access these services. This is not strictly necessary as both the clients and the servers could run on a single machine. In practice, however, most client-server systems are implemented as distributed systems.

Clients may have to know the names of the available servers and the services that they provide. However, servers need not know either the identity of clients or how many clients there are.

Figure 11.3 shows ,m example of a system that is based on the client-server model.

This is a multi-user, web-based system to provide a film and photograph library. In this system, several servers manage and display the different types of media.

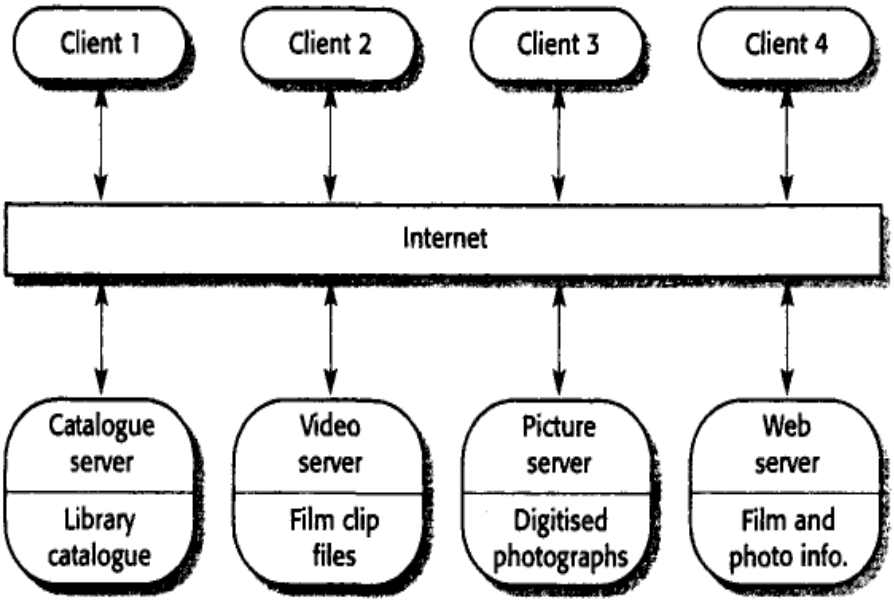


Figure 11.3 The architecture of a film and picture library system

- The catalogue must be able to deal with a variety of queries and provide links into the web information system that includes data about the film and video clip, and an e-commerce system that supports the sale of film and video clips.
- The most important advantage of the client-server model is that it is a distributed architecture. Effective use can be made of networked systems with many distributed processors. It is easy to add a new server and integrate it with the rest of the system or to upgrade servers transparently without affecting other parts of the system.
- However, changes to existing clients and servers may be required to gain the full benefits of integrating a new server. There may be no shared data model across servers and sub-systems may organize their data in different ways. This means that specific data models may be established on each server to allow its performance to be optimized.

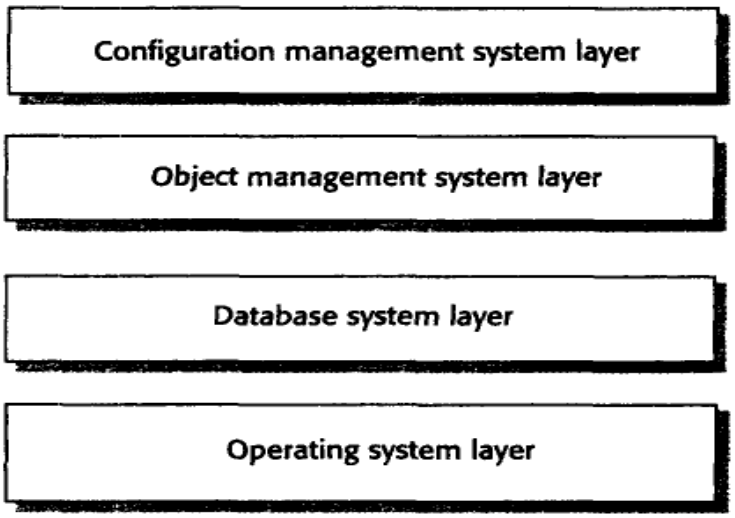
3. The layered model

- The layered model of an architecture (sometimes called an abstract machine model)
- organizes a system into layers, each of which provide a set of services. Each layer can be thought of as an abstract machine whose machine language is defined by the services provided by the layer. This 'language' is used to implement the next level of abstract machine. For example, a common way to implement a language is to define an ideal 'language machine' and compile the language into code for this machine.
- An example of a layered model is the OSI reference model of network protocols

- Figure 11.4 reflects the APSE structure and shows how a configuration management system might be integrated using this abstract machine approach.
- The configuration management system manages versions of objects and provides general configuration management facilities, To support these configuration management facilities, it uses an object management system that provides information storage and management services for configuration items or objects. This system is built on top of a database system to provide basic data storage and services such as transaction management, rollback and recovery,

  and access control. The database management uses the underlying operating system facilities and file store in its implementation.
- The layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users. This architecture is also changeable and portable. So long as its interface is unchanged, a layer can be replaced by another, equivalent layer

Figure 11.4 Layered model of a version management system



- As layered systems localize machine dependencies in inner layers, this makes it easier to provide multi-platform implementations of an application system. Only the inner, machine-dependent layers need be re-implemented to take account of the facilities of a different operating system or Database
- A disadvantage of the layered approach is that structuring systems in this way can be difficult. Inner layers may provide basic facilities, such as file management, that are required at all levels.

- Services required by a user of the top level may therefore have to 'punch through' adjacent layers to get access to services that are provided several levels beneath it.
- Performance can also be a problem because of the multiple levels of command interpretation that are sometimes required. If there are many layers, a service request from a top layer may have to be interpreted several times in different layers before it is processed.
- To avoid these problems, applications may have to communicate directly with inner layers rather than use the services provided by the adjacent layer.

3.2(4) Modular decomposition styles

- You need to make a decision on the approach to be used in decomposing sub-systems into modules. There is not a rigid distinction between system organization and modular decomposition.

There is no clear distinction between sub-systems and modules, but I find it useful to think of them as follows:

1. A sub-system is a system in its own right whose operation does not depend on the services provided by other sub-systems. Sub-systems are composed of modules and have defined interfaces, which are used for communication with other sub-systems.

2. A module is normally a system component that provides one or more services to other modules. It makes use of services provided by other modules. It is not normally considered to be an independent system. Modules are usually composed from a number of other simpler system components.

- There are two main strategies that you can use when decomposing a sub-system into modules:
- I. *Object-oriented decomposition* where you decompose a system into a set of communicating objects.
- *2.Function-oriented pipelining* where you decompose a system into functional modules that accept input data and transform it into output data.
- In the object-oriented approach, modules are objects with private state and defined operations on that state.
- In the pipelining model, modules are functional transformations. In both cases, modules may be implemented as sequential components or as processes.

1. Object-oriented decomposition

- An object-oriented, architectural model structures the system into a set of loosely coupled objects with well-defined interfaces. Objects call on the services offered by other objects".
- Figure 11.5 is an example of an object-oriented architectural model of an invoice processing system. This system can issue invoices to customers, receive payments, and issue receipts for these payments and reminders for unpaid invoices.
- An object-oriented decomposition is concerned with object classes, their attributes and their operations. When implemented, objects are created from these classes and some control model is used to coordinate object operations.
- In this particular example, the Invoice class has various associated operations that implement the system functionality. This class makes use of other classes representing customers, payments and receipts.

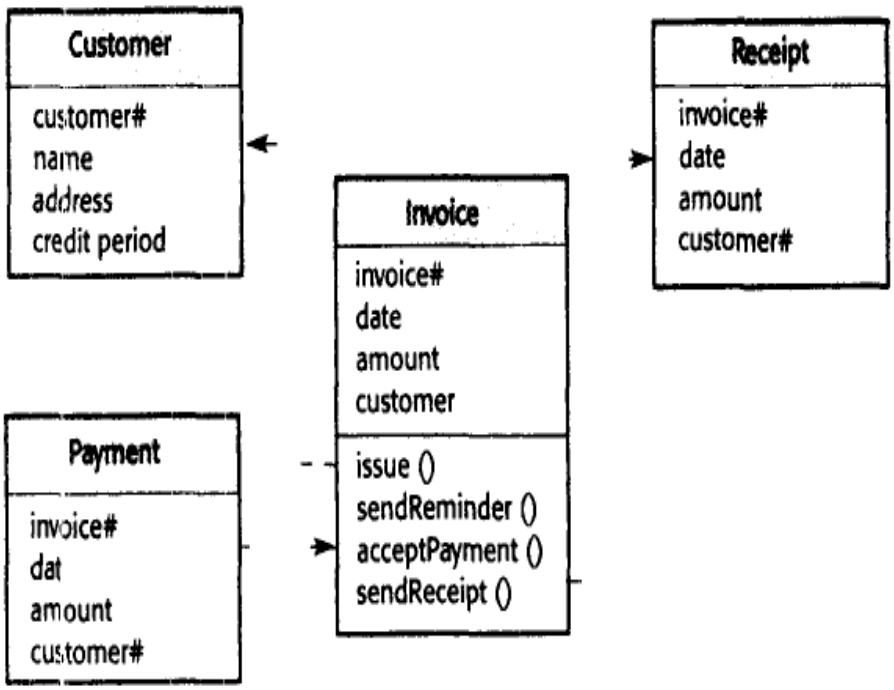


Figure 11.5 An object model of an invoice processing system

The advantages of the object-oriented approach are :

- Are loosely coupled, the implementation of objects can be modified without affecting other objects.
- Objects are often representations of real-world entities so the structure of the system is readily understandable.
- Because these real-world entities are used in different systems, objects can be reused.
- Object-oriented programming languages have been developed that provide direct implementations of architectural components.

The object-oriented approach does have disadvantages.

- To use services, objects must explicitly reference the name and the interface of other objects.
- If an interface change is required to satisfy proposed system changes, the effect of that change on all users of the changed object must be evaluated.
- While objects may map cleanly to small-scale real-world entities, more complex entities are sometimes difficult to represent as objects.

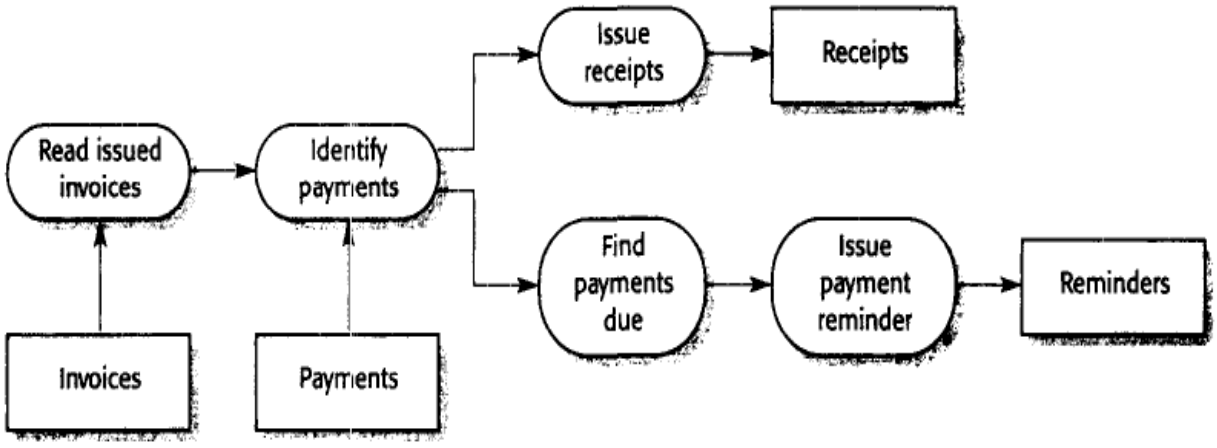
2.Function-oriented pipelining

- In a function-oriented pipeline or data-flow model, functional transformations process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence.
- Each processing step is implemented as a transform. Input data flows through these transforms until converted to output.
- The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.
- When the transformations are represented as separate processes, this model is sometimes called the pipe and filter style after the terminology used in the Unix system.
- The Unix system provides pipes that act as data conduits and a set of commands that are functional transformations.
- The term *filter* is used because a transformation 'filters out the data it can process from its input data stream.

- Variants of this pipelining model have been in use since computers were first used for automatic data processing. When transformations are sequential with data processed in batches, this architectural model is a batch sequential model.
- An example of this type of system architecture is shown in Figure 11.6.
- An organization has issued invoices to customers. Once a week, payments that have been made are reconciled with the invoices.
- For those invoices that have been paid, a receipt is issued. For those invoices that have not been paid within the allowed payment time, a reminder is issued.
- This is a model of only part of the invoice processing system; alternative transformations would be used for the issue of invoices. Notice the difference between this and its object-oriented equivalent discussed in the previous section.
- The object model is more abstract as it does not include information about the sequence of operations.
- **Fig : 11.6 A Pipeline model of an invoice processing system.**

The advantages of this architecture are:

- I. It supports the reuse of transformations.
- 2. It is intuitive in that many people think of their work in terms of input and output processing.
- 3. Evolving the system by adding new transformations is usually straightforward.
- 4. It is simple to implement either as a concurrent or a sequential system.



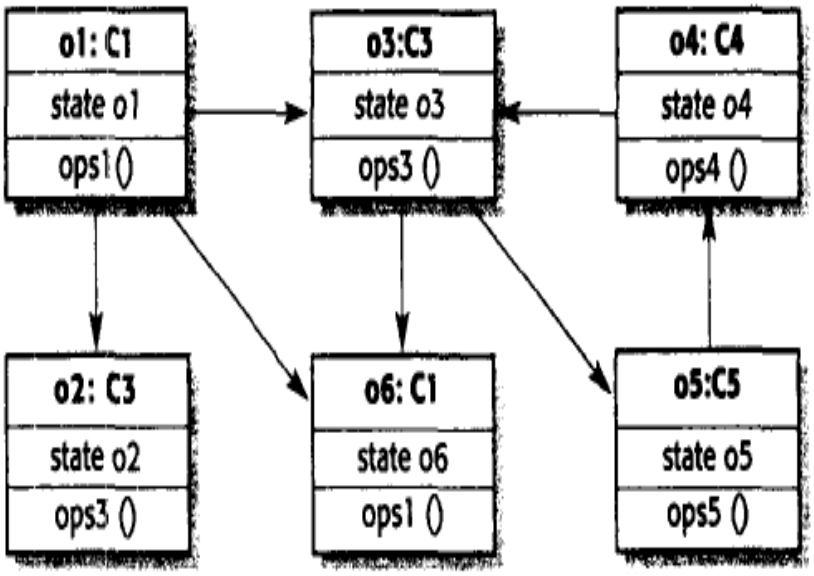
- The principal problem with this style is that there has to be a common format for data transfer that can be recognized by all transformations.

- Each transformation must either agree with its communicating transformations on the format of the data that will be processed or with a standard format for all data communicated must be imposed.
- Interactive systems are difficult to write using the pipelining model because of the need for a stream of data to be processed.

3.3(1) Object Oriented Design

- An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state (Figure 14.1).
- The representation of the state is private and cannot be accessed directly from outside the object.
- Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions.
- When the design is realized as an executing program, the objects are created dynamically from these class definitions.

Figure 14.1 A system made up of interacting objects



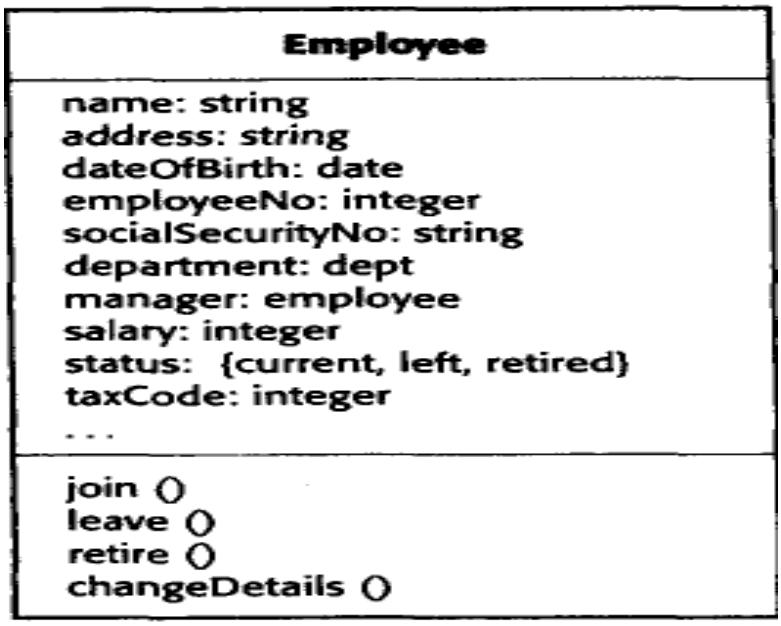
- Object-oriented design is part of object-oriented development where an object-oriented strategy is used throughout the development process:
- ***Object-oriented analysis*** is concerned with developing an object-oriented model of the application domain. The objects in that model reflect the entities and operations associated with the problem to be solved.
- ***Object-oriented design*** is concerned with developing an object-oriented model of a software system to implement the identified requirements. The objects in an object-oriented design are related to the solution to the problem. There may be close relationships between some problem objects and some solution objects, but the designer inevitably has to add new objects and to transform problem objects to implement the solution.
- ***Object-oriented programming*** is concerned with realizing a software design using an object-oriented programming language, such as Java. An object-oriented programming language provides constructs to define object classes and a run-time system to create objects from these classes.
- Object-oriented systems are easier to change than systems developed using other approaches because the objects are independent. They may be understood and modified as standalone entities. Changing the implementation of an object or adding services should not affect other system object
- Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system. This improves the understandability and hence the maintainability of the design.
- Objects are, potentially, reusable components because they are independent encapsulations of state and operations. Designs can be developed using objects that have been created in previous designs. This reduces design, programming and validation costs.
- Several object-oriented design methods have been proposed .
- The UML is a unification of the notations used in these methods.

3.3(2) 1. Objects and object classes

- The terms *object* and *object-oriented* are applied to different types of entity, design methods, systems and programming languages. There is a general acceptance that an object is an

encapsulation of information, and this is reflected in my definition of an object and an object class:

- *An object is an entity that has a state and a defined set of operations that operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) that request these services when some computation is required.*
- *Objects are created according to an object class definition. An object class definition is both a type specification and a template for creating objects. It includes declarations of all the attributes and operations that should be associated with an object of that class.*
- Figure 14.2 (An Employee Object) illustrates this notation using an object class that models an employee in an organization.
- The UML uses the term *operation* to mean the specification of an action; the term *method* is used to refer to the implementation of an operation.

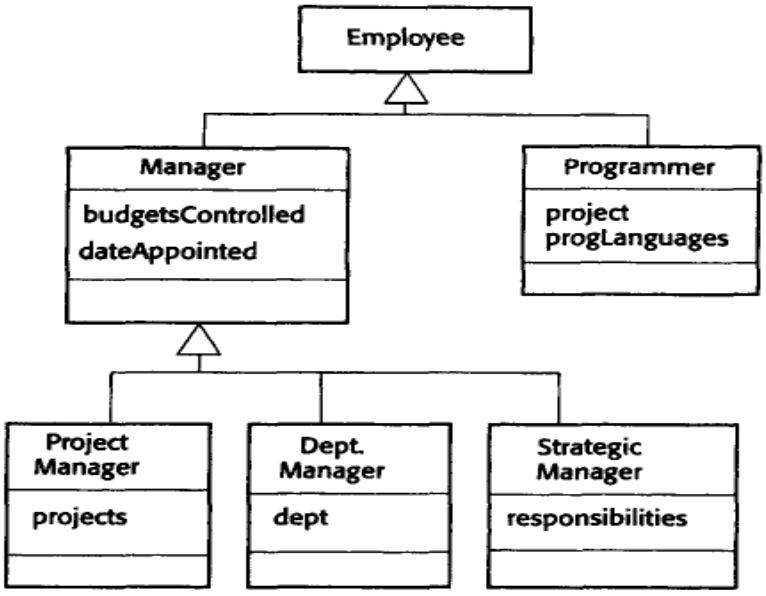


- The class Employee defines a number of attributes that hold information about employees including their name and address, social security number, tax code, and so on.
- The ellipsis (...) indicates that there are more attributes associated with the class than are shown.
- Operations associated with the object are join (called when an employee joins the organization), leave (called when an employee leaves the organization), retire (called when

the employee becomes a pensioner of the organization)and changeDetails (called when some employee information needs to be modified).

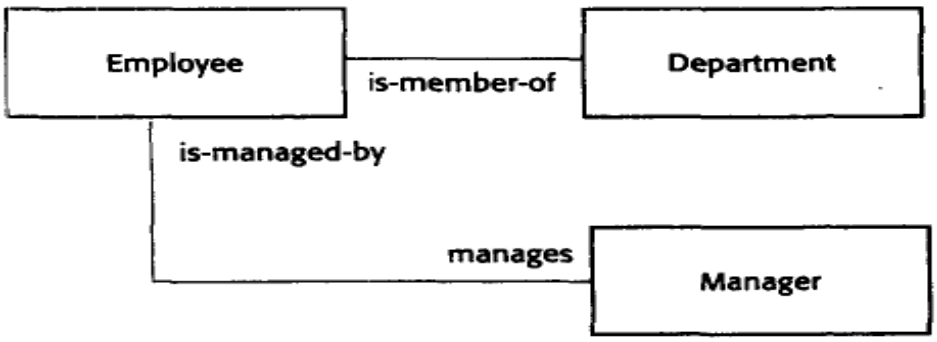
- Objects communicate by requesting services (calling methods) from other objects and, if necessary, by exchanging the information required for service provision. The copies of information needed to execute the service and the results of service execution are passed as parameters.
- In service-based systems, object communications are implemented directly as XML text messages that objects exchange.
- When service requests are implemented in this way, communication between objects
- is synchronous. That is, the calling object waits for the service request to be completed.
- However, if objects are implemented as concurrent processes or threads, the object (communication may be asynchronous. The calling object may continue in operation while the requested service is executing.

- **Figure 14.3**



- **Figure 14.3** shows an example of an object class hierarchy where different classes of employee are shown.

- Classes lower down the hierarchy have the same attributes and operations as their parent classes but may add new attributes and operations or modify some of those from their parent classes.
- This means that there is one-way interchangability. If the name of a parent class is used in a model, the object in the system may either be defined as of that class or of any of its descendants.
- The class Manager in Figure 14.3 has all of the attributes and operations of the class Employee but has, in addition, two new attributes that record the budgets controlled by the manager and the date that the manager was appointed to a particular management role.
- Similarly, the class Programmer adds new attributes that define
- the project that the programmer is working on and the programming language skills that he or she has. Objects of class Manager or Programmer may therefore be used anywhere an object of class Employee is required.
- Objects that are members of an object class participate in relationships with other objects. These relationships may be modeled by describing the associations between the object classes.
- In the UML, associations are denoted by a line between the object classes that may optionally be annotated with information about the association.
- This is illustrated in Figure 14.4, which shows the association between objects of class Employee and objects of class Department, and between objects of class Employee and objects of class Manager.
- Association is a very general relationship and is often used in the UML to indicate that either an attribute of an object is an associated object or the implementation of an object method relies on the associated object.
- **Fig: 14.4 An Association Model**
- One of the most common associations is aggregation, which illustrates how objects may be composed of other objects.



- 1.1 Concurrent Objects
- All object requests a service from another object by sending a service request' message to that object. There is no requirement for serial execution where one object waits for completion of a requested service.
- Consequently, the general model of object interaction allows objects to execute concurrently as parallel processes. These objects may execute on the same computer or as distributed objects on different machines.
- In practice, most object-oriented programming languages have as their default a serial execution model where requests for object services are implemented in the same way as function calls. Therefore, when an object called theList is created from a normal object class, you write in Java:

  theList.append (17)
- This calls the append method associated with theList object to add the element 17 to theList, and execution of the calling object is suspended until the append operation has been completed. However, Java includes a very simple mechanism
- (threads) that lets you create objects that execute concurrently.
- Threads are created in Java by using the built-in Thread class as a parent class in a class declaration.
- Threads must include a method called run, which is started by the Java run-time system when objects that are defined as threads are created.
- There are two kinds of concurrent object implementation:
- *1. Servers* where the object is realised as a parallel process with methods corresponding to the defined object operations. Methods start up in response to an external message and may

execute in parallel with methods associated with other objects. When they have completed their operation, the object suspends itself and waits for further requests for service.

- *2. Active objects* where the state of the object may be changed by internal operations executing within the object itself. The process representing the object continually executes these operations so never suspends itself.
- Servers are most useful in a distributed environment where the calling and the called object may execute on different computers. The response time for the service that is requested is unpredictable, so, wherever possible, you should design the system so that the object that has requested a service does not have to wait for that service to be completed. may request the service.
- Active objects are used when an object needs to update its own state at specified intervals. This is common in real-time systems where objects are associated with hardware devices that collect information about the system's environment.
- The object's methods allow other objects access to the state information.
- Figure 14.5 shows how an active object may be defined and implemented in Java.
- The object class represents a transponder on an aircraft. The transponder keeps track of the aircraft's position using a satellite navigation system. It can respond to messages from air traffic control computers.
- It provides the current aircraft position in response to a request to the givePosition method.

```
class Transponder extends Thread {

        Position currentPosition ;
        Coords c1, c2 ;
        Satellite sat1, sat2 ;
        Navigator theNavigator ;

        public Position givePosition ()
        {
            return currentPosition ;
        }

        public void run ()
        {
            while (true)
            {
                c1 = sat1.position () ;
                c2 = sat2.position () ;
                currentPosition = theNavigator.compute (c1, c2) ;
            }

        }

} //Transponder
```

3.3(3) 2.An object-oriented design process

- There are several methods of object-oriented design with no definitive 'best' method or design process.
- The process here is a general one that incorporates activities common to most OOD processes.
- The general process that I use here for object-oriented design has a number of stages:
  1. Understand and define the context and the modes of use of the system.
  2. Design the system architecture.
  3. Identify the principal objects in the system.
  4. Develop design models.
  5. Specify object interfaces.
- In fact all of the above activities are interleaved and so influence each other.

- Objects are identified and the interfaces fully or partially specified as the architecture of the system is defined.
- As object models are produced these individual object definitions may be refined which leads to changes to the system architecture.
- We can illustrate the process activities by developing an example of an object-oriented design.
- This example is part of a system for creating weather maps using automatically collected meteorological data. The detailed requirements for such a weather mapping system would take up many pages.
- *A weather mapping system is required to generate weather maps on a regular basis using data collected from remote, unattended weather stations and other data sources such as weather observers, balloons and satellites. Weather stations transmit their data to the area computer in response to a request from that machine.*

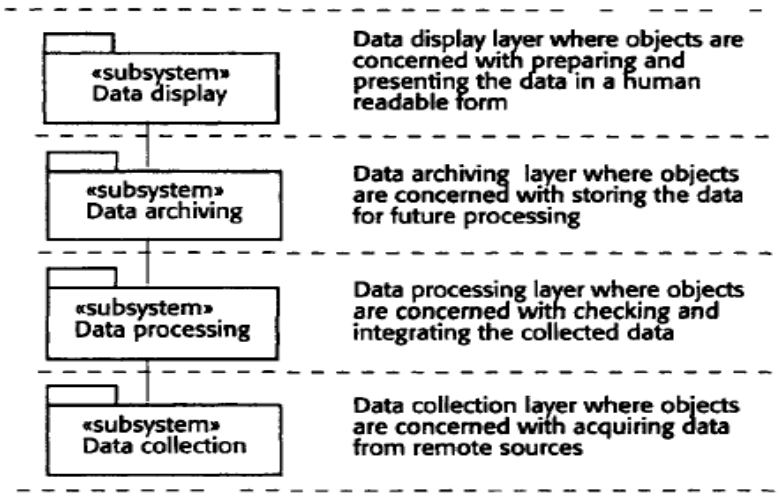


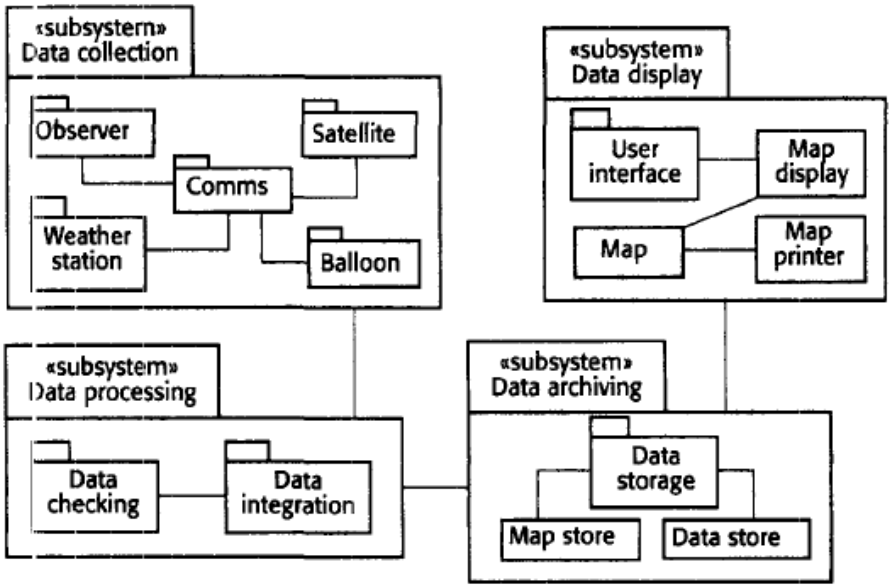
Figure 14.6 Layered architecture for weather mapping system

- *The area computer system validates the collected data and integrates the data from different sources. The integrated data is archived and, using data from this archive and a digitized map database, a set of local weather maps is created. Maps may be printed for distribution on a special-purpose map printer or may be displayed in a number of different formats.*
- This description shows that part of the overall system is concerned with collecting data, part with integrating the data from different sources, part with archiving that data and part with creating weather maps.
- This is a layered architecture that reflects the stages of processing in the system, namely data collection, data integration, data archiving and map generation. A layered architecture is

appropriate in this case because each stage relies only on the processing of the previous stage for its operation.

- The layers and have included the layer name in a UML package symbol that has been denoted as a sub-system.
- In Figure 14.7, we have expanded on this abstract architectural model by showing the components of the sub-systems. These are still abstract and have been derived from the information in the description of the system.

Figure 14.7 Subsystems in the weather mapping system



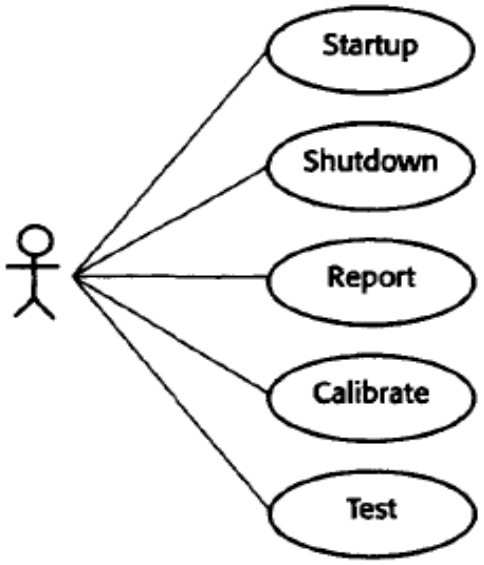
2.1. System context and models of use

- The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment.
- The system context and the model of system use represent two complementary models of the relationships between a system and its environment:
- 1. The system context is a static model that describes the other systems in that environment.
- 2. The model of the system use is a dynamic model that describes how the system actually interacts with its environment.
- The context model of a system may be represented using associations (see Figure 14.4) where a simple block diagram of the overall system architecture is produced.
- You then develop this by deriving a sub-system model using UML packages as shown in Figure 14.7. This model shows that the context of the weather station system is within a sub-

system concerned with data collection. It also shows other sub-systems that make up the weather mapping system.

- When you model the interactions of a system with its environment you should use an abstract approach that does not include too much detail.
- The use-case model for the weather station is shown in Figure 14.8. This shows that weather station interacts with external entities for startup and shutdown, for reporting the weather data that has been collected, and for instrument testing and calibration.
- Each of these use-cases can be described in structured natural language. This helps designers identify objects in the system and gives them an understanding of what the system is intended to do.

Figure 14.8 Use-cases for the weather station



- The use-case description helps to identify objects and operations in the system.
- From the description of the Report use-case, it is obvious that objects representing the instruments that collect weather data will be required, as will an object representing the summary of the weather data. Operations to request weather data and to send weather data are required.

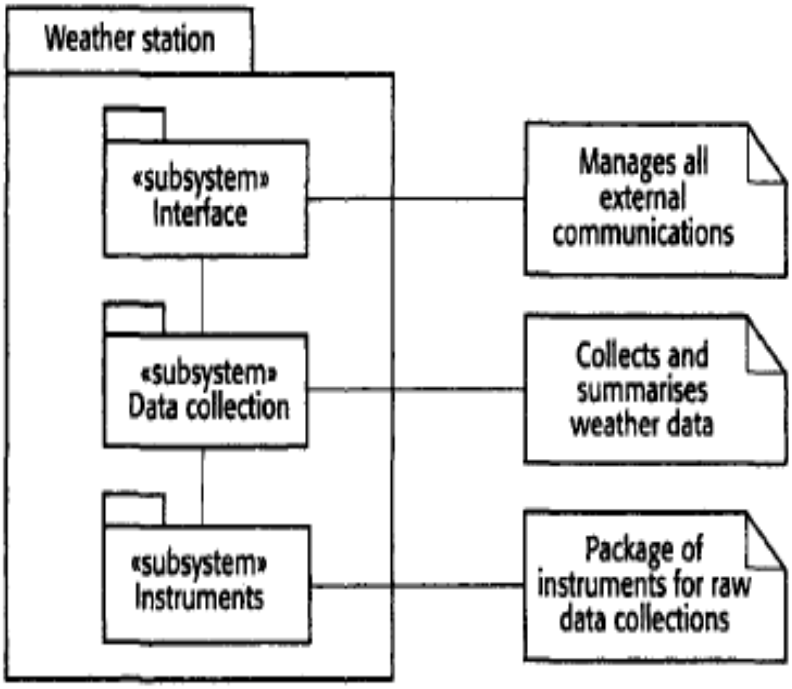
Figure 14.9 Report use-case description

System	Weather station
Use-case	Report
Actors	Weather data collection system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather data collection system. The data sent are the maximum, minimum and average ground and air temperatures, the maximum, minimum and average air pressures, the maximum, minimum and average wind speeds, the total rainfall, and the wind direction as sampled at five-minute intervals.
Stimulus	The weather data collection system establishes a modem link with the weather station and requests transmission of the data.
Response	The summarised data is sent to the weather data collection system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in future.

2.2Architectural design

- Once the interactions between the software system that is being designed and the system, environment have been defined, you can use this information as a basis for designing the system architecture. The automated weather station is a relatively simple system, and its architecture can again be represented as a layered model.
- illustrated this in **Figure 14.10** as three UML packages within the more general Weather station package. Notice how I have used UML annotations (text in boxes with a folded comer) to provide additional information here.
- The three layers in the weather station software are:
- 1. The **interface *layer*** that is concerned with all communications with other parts of the system and with providing the external interfaces of the system;
- 2. The ***data collection layer*** that is concerned with managing the collection of data from the instruments and with summarizing the weather data before transmission to the mapping system;
- 3. The ***instruments layer*** that is an encapsulation of all of the instruments used to collect raw data about the weather conditions.

Figure 14.10 The weather station architecture



2.3 Object identification

In practice this process is actually concerned with identifying object classes. The design is described in terms of these classes. Inevitably, you have to refine the object classes that you initially identify and revisit this stage of the process as you develop a deeper understanding of the design.

There have been various proposals made about how to identify object classes:

- 1. Use a grammatical analysis of a natural language description of a system. Objects and attributes are nouns; operations or services are verbs. This approach has been embodied in the HOOD method for object-oriented design that was widely used in the European aerospace industry.
- 2. Use tangible entities (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings locations such as offices, organizational units such as companies, and so on Support this by identifying storage structures (abstract data structures) in the solution domain that might be required to support these objects.
- 3. Use a behavioral approach where the designer first understands the overall behavior of the system. The various behaviors are assigned to different parts of the system and an

understanding is derived of who initiates and participates in these behaviors. Participants who play significant roles are recognized as

objects .

- 4. Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes and operations. A method of analysis called CRC cards where analysts and designers take on the role of objects is effective in supporting this scenario-based approach.
- These approaches help you get started with object identification. In practice, you may have to use several knowledge sources to discover object classes. Object classes, attributes and operations that are initially identified from the informal system description can be a starting point for the design.
- A hybrid approach here to identify the weather station objects to describe all the objects, but shown five object classes in Figure 14.11. Ground thermometer, Anemometer and Barometer represent application domain objects, and WeatherStation and WeatherData have been identified from the system description and the scenario (use-case) description.

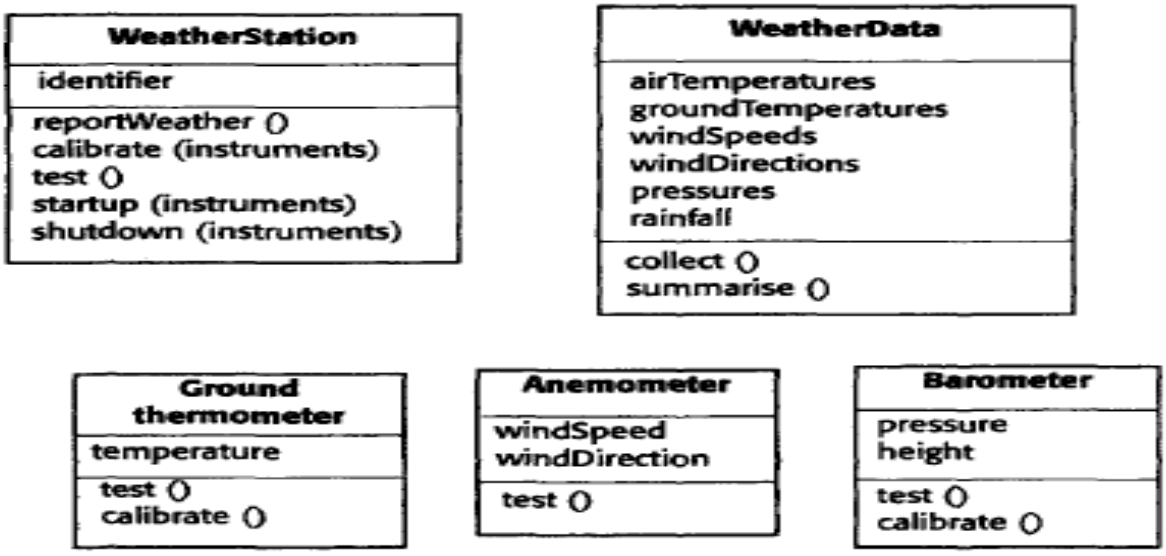
  These objects are related to the levels in the system architecture.

  1. The WeatherStation object class provides the basic interface of the weather station with its environment. Its operations therefore reflect the interactions shown in Figure 14.8. In this case, I use a single object class to encapsulate all of these interactions, but in other designs you may chose to design the system Interface as several different classes.

  2. The WeatherData object class encapsulates the summarized data from the instruments in the weather station. Its associated operations are concerned with collecting and summarizing the data that is required.

  3. The Ground thermometer, Anemometer and Barometer object classes are directly related to instruments in the system. They reflect tangible hardware entities in the system and the operations are concerned with controlling that hardware.

- **Fig: 14.11 Examples of object classes in the weather station system.**



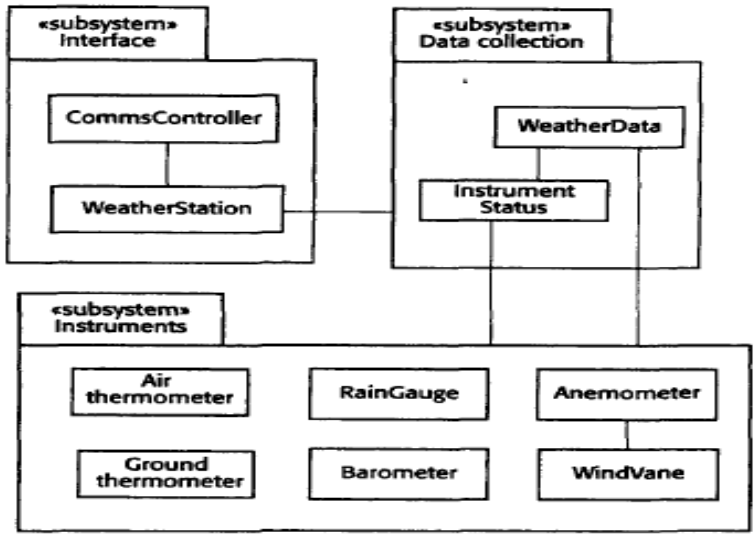
- The objects associated with each instrument should not be active objects. The collect operation in WeatherData calls on instrument objects to make readings when required. Active objects include their own control and, in this case, it would mean that each instrument would decide when to make readings.
- The disadvantage of this is that, if a decision was made to change the timing of the data collection or if different weather stations collected data differently, then new object classes would have to be introduced. By making the instrument objects make readings on request, any changes to collection strategy can be easily implemented without changing the objects associated with the instruments.

2.4 Design models

- Design models show the objects or object classes in a system and, where appropriate, the relationships between these entities. Design models essentially are the design. They are the bridge between the requirements for the system and the system implementation.
- An important step in the design process, therefore, is to decide which design models that you need and the level of detail of these models. This depends on the type of system that is being developed. A sequential data processing system will be designed in a different way from an embedded real-time system, and different design models will therefore be used

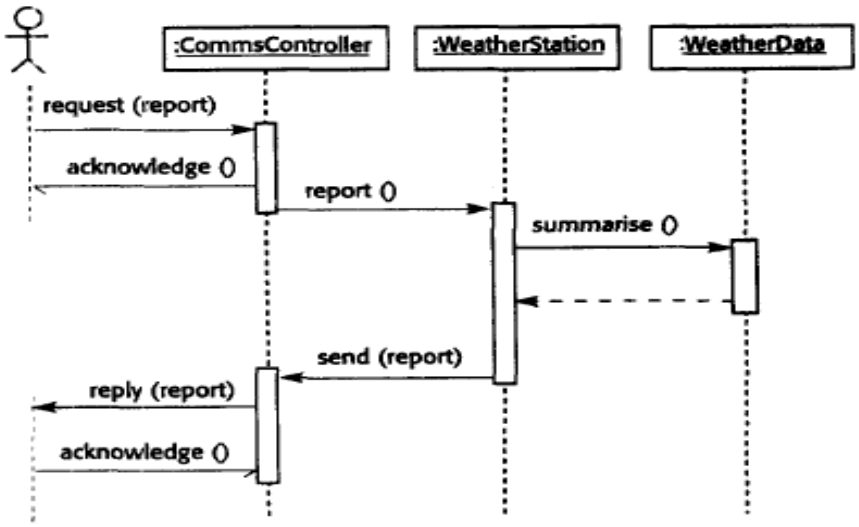
- There are two types of design models that should normally be produced to describe an objen-oriented design:
- *1.* ***Static models*** describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalization relationships, uses/used-by relationships and composition relationships.
- *2.* ***Dynamic models*** describe the dynamic structure of the system and show the interactions between the system objects (not the object classes). Interactions that may be documented include the sequence of service requests made by objects and the way in which the state of the system is related to these object interactions.
- The UML provides for 12 different static and dynamic models that may be produced to document a design. The models that we discuss in this section are:
- ***1. Subsystem models*** that show logical groupings of objects into coherent sub-systems. These are represented using a form of class diagram where each sub-system is shown as a package. Subsystem models are static models.
- *2.* ***Sequence models*** that show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic: models.
- *3.* ***State machine models*** that show how individual objects change their state in response to ,events. These are represented in the UML using statechart diagrams. State machine models are dynamic models.
- Figure 14.12 shows the objects in the sub-systems in the weather station. I also show some associations in this model. For example, the CommsController object is associated with the WeatherStation object, and the WeatherStation object is associated with the Data collection package. This means that this object is associated with one or more objects in this package. A package model plus an object class model should describe the logical groupings in the system.

Figure 14.12 Weather station packages



- A sub-system model is a useful static model as it shows how the design may be organized into logically related groups of objects. *Sequence models* are dynamic models that document, for each mode of interaction, the sequence of object interactions that take place. **Figure 14.13** is an example of a sequence model that shows the operations involved in collecting the data from a weather station.

Figure 14.13 Sequence of operations—data collection



In a sequence model:

1. The objects involved in the interaction are arranged horizontally with a vertical line linked to each object.

2. Time is represented vertically so that time progresses down the dashed vertical lines.

Therefore, the sequence of operations can be read easily from the mode1.

3. Labelled arrows linking the vertical lines represent interactions between objects. These are *not* data flows but represent messages or events that are fundamental to the interaction.

4. The thin rectangle on the object lifeline represents the time when the object is the controlling object in the system. An object takes over control at the top of the rectangle and relinquishes control to another object at the bottom of the rectangle. If there is a hierarchy of calls, control is not relinquished until the lase return to the initial method call has been completed.

You read sequence diagrams from top to bottom:

1. An, object that is an instance of CommsController (:CommsController) receives a request from its environment to send a weather report. It acknowledges receipt of this request. The half-arrowhead on the acknowledge message indicates that the message sender does not expect a reply.

2. This object sends a message to an object that is an instance of WeatherStation to create a weather report. The instance of CommsController then suspends itself (its control box ends). The style of arrowhead used indicates that the CommsController object instance and the WeatherStation object instance are objects that may execute concurrently.

3. The object that is an instance of WeatherStation sends a message to a WeatherData object to summarize the weather data. In this case, the squared off style of arrowhead indicates that the instance of WeatherStation waits for a reply.

4. This summary is computed and control returns to the WeatherStation object. The dotted arrow indicates a return of control.

5. This object sends a message to CommsController requesting it to transfer the data to the remote system. The WeatherStation object then suspends itself.

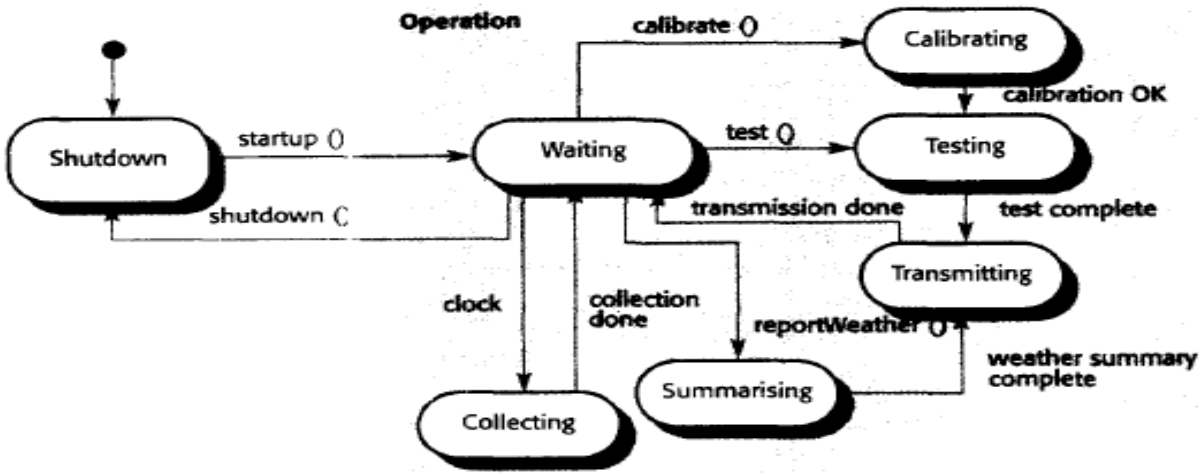
6. The CommsController object sends the summarized data to the remote system, receives an acknowledgement, and then suspends itself waiting for the next request.

- Figure 14.14 is a state-chart for the WeatherStation object that shows how it responds to requests for various services.
- You can read this diagram as follows:

- 1. If the object state is Shutdown then it can only respond to a startup ( )message. It then moves into a state where it is waiting for further messages. The un-labelled arrow with the black blob indicates that the Shutdown state is the initial state.
- 2. In the Waiting state, the system expects further messages. If a shutdown( ) message is received, the object returns to the shutdown state.
- 3. If a reportWeather( ) message is received, the system moves to the Summarizing state. When the summary is complete, the system moves to a Transmitting state where the information is transmitted through the

  CommsController. It then returns to the Waiting state.
- 4. If a calibrate( ) message is received, the system moves to the Calibrating state, then the Testing state, and then the Transmitting state, before returning to the Waiting state If atest( ) message is received, the system moves directly to the Testing state.
- 5. If a signal from the clock is received, the system moves to the Collecting state, where it is collecting data from the instruments. Each instrument is instructed in turn collect its data.

Figure 14.14



2.5 Object interface Specification

- An important part of any design process is the specification of the interfaces between the components in the design. You need to specify interfaces so that objects and sub-systems can be designed in parallel.
- There is not necessarily a simple I: I relationship between objects and interfaces.

- The same object may have several interfaces, each of which is a viewpoint on the methods that it provides. This is supported directly in Java, where interfaces are declared separately from objects, and objects 'implement' interfaces.
- **Figure 14.15,** which shows the interface specification in Java of the weather station. As interfaces become more complex, this approach becomes more effective because the syntax-checking facilities in the compiler may be used to discover errors and inconsistencies in the interface description. The

Figure 14.15 Java description of weather station interface

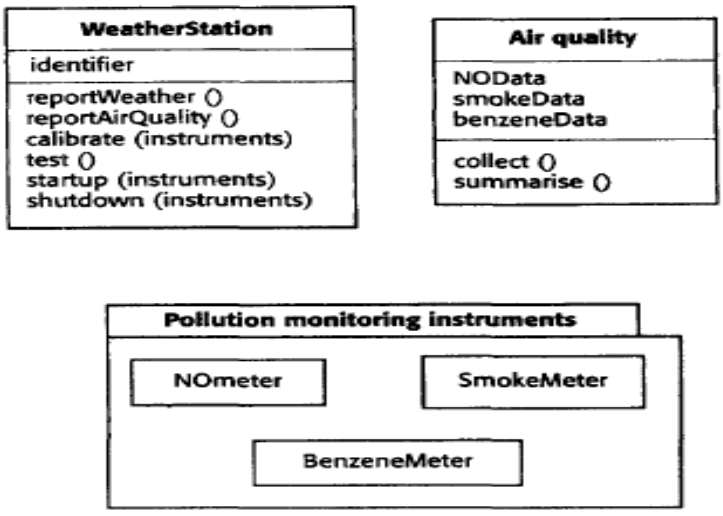
```
interface WeatherStation {

    public void WeatherStation () ;

    public void startup () ;
    public void startup (Instrument i) ;

    public void shutdown () ;
    public void shutdown (Instrument i) ;

    public void reportWeather ( ) ;

    public void test () ;
    public void test ( Instrument i ) ;

    public void calibrate ( Instrument i) ;

    public int getID () ;

} //WeatherStation
```

3.3(4) Design evolution

- After a decision has been made to develop a system such as a weather data collection system, it is inevitable that proposals for system changes will be made.
- An important advantage of an object-oriented approach to design is that it simplifies the problem of making changes to the design. The reason for this is that object state representation does not influence the design.
- To show how an object-oriented approach to design makes change easier, assume that pollution-monitoring capabilities are to be added to each weather station.

- This involves adding an air quality meter to compute the amount of various pollutants in the atmosphere. The pollution readings are transmitted at the same time as the weather data.
- To modify the design, the following changes must be made:
- 1. An object class called Air quality should be introduced as part of WeatherStation at the same level as WeatherData.
- 2. An operation reportAirQuality should be added to WeatherStation to send the pollution information to the central computer. The weather station control software must be modified so that pollution readings are automatically collected when requested by the top-level WeatherStation object.
- 3. Objects representing the types of pollution monitoring instruments should be added. In this case, levels of nitrous oxide, smoke and benzene can be measured.
- The pollution monitoring objects are encapsulated in a separate package called Pollution monitoring instruments.
- This has associations with Air quality and WeatherStation but not with any of the objects used to collect weather data.
- **Figure 14.16** shows WeatherStation and the new objects added to the system. Apart from at the highest level of the system (WeatherStation), no software changes are required in the original objects in the weather station.

Figure 14.16 New objects to support pollution monitoring



UNIT – IV

VERIFICATION AND VALIDATION: Planning verification and validation, Software inspections, Automated static analysis, Verification and formal methods.

SOFTWARE TESTING: System testing, Component testing, Test case design, Test automation,

Quality management: Software Quality Assurance.

4.1(1) Verification and Validation

- Testing a program is the most common way of checking that it meets its specification and does what the customer wants. However, testing is only one of a range of verification and validation techniques.

Verification and validation **(V &V)**

is the name given to these checking and analysis processes. Verification and validation activities take place at each stage of the software process. V & V starts with requirements reviews and continues through design reviews and code inspections to product testing.

Verification and validation are not the same thing, although they are often confused. Boehm (1979) succinctly expressed the difference between them:

'**Validation:** Are we building the right product?'

'**Verification:** Are we building the product right?'

- These definitions tell us that the role of **verification** involves checking that the software conforms to its specification.

You should check that it meets its specified functional and non-functional requirements.

- **Validation,** however, is a more general process. The aim of validation is to ensure that the software system meets the customer s expectations. **It** goes beyond checking that the system conforms to its specification to showing that the software does what the customer expects it to do.

- The ultimate goal of the ***verification*** and ***validation*** process is to establish confidence that the software system is 'fit for purpose'

- This means that the system must be good enough for its intended use.

- **The level of required confidence depends**

On the system's purpose, the expectations of the system users and the current marketing environment for the system:

1. Software function The level of confidence required depends on how critical the software is to an organization.

2. ***User expectations*** It is a sad reflection on the software industry that many users have low expectations of their software and are not surprised when it fails during use. They are willing to accept these system failures when the benefits of use outweigh the disadvantages

3. ***Marketing environment*** When a system is marketed, the sellers of the system must take into account competing programs, the price those customers are willing to pay for a system and the required schedule for delivering that system. Where a company has few competitors, it may decide to release a program before it has been fully tested and debugged because they want to be the first into the **mar**ket.

Where customers are not willing to pay high prices for software, they may be willing to tolerate more software faults

- Within the V & V process, there are two complementary approaches to system checking and analysis:

- 1. ***Software inspections or peer reviews*** analyze and check system representations such as the requirements document, design diagrams and the program source code. You can use inspections at all stages of the process. Inspections may be supplemented by some automatic analysis of the source text of a system or associated documents. Software inspections and automated analyses are static V & V techniques, as you don't need to run the software on a computer.

- *2.* ***Software testing*** involves running an implementation of the software with test data. Testing is a dynamic technique of verification and validation.

- Figure 22.1 shows that software inspections and testing play complementary roles in the software process. The arrows indicate the stages in the process where the techniques may be

used. Therefore, you can use software inspections at all stages of the software process. Starting with the requirements, any readable representations of the software can be inspected.

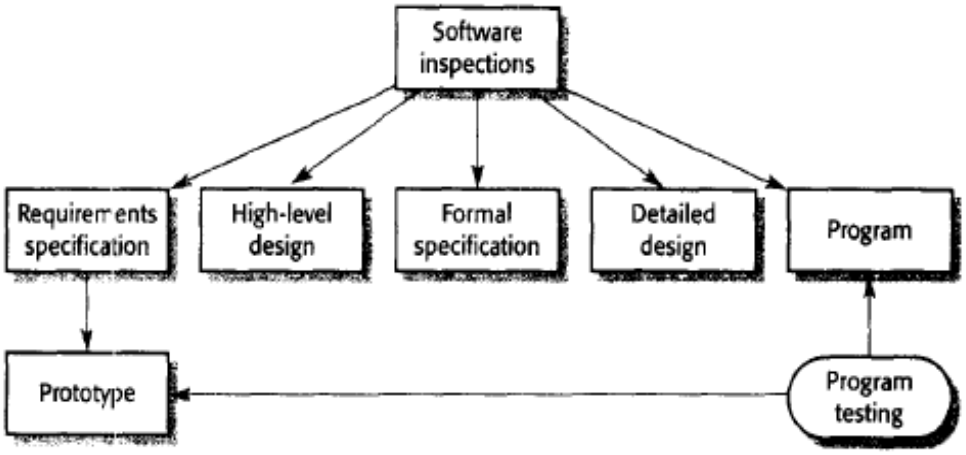
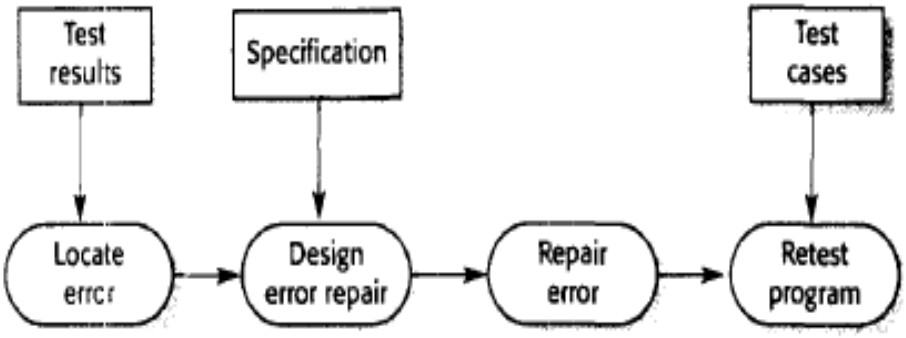


Figure 22.1 Static and dynamic verification and validation

- Inspection techniques include program inspections, automated source code analysis and formal verification. However, static techniques can only check the correspondence between a program and its specification (verification); they cannot demonstrate that the software is operationally useful.
- Although software inspections are now widely used, program testing will always be the main software verification and validation technique. Testing involves exercising the program using data like the real data processed by the program.
- There are two distinct types of testing that may be used at different stages in the software process:
- 1. ***Validation testing*** is intended to show that the software is what the customer wants-that it meets its requirements. As part of validation testing, you may use statistical testing to test the program s performance and reliability, and to check how it works under operational conditions. I discuss statistical testing and reliability estimation.
- *2.* ***Defect testing*** is intended to reveal defects in the system rather than to simulate its operational use. The goal of defect testing is to [md inconsistencies between a program and its specification.

- The processes of V & V and debugging are normally interleaved. As you discover faults in the program that you are testing, you have to change the program to correct these faults. However, testing (or, more generally verification and validation) and debugging have different goals:
- 1. Verification and validation processes are intended to establish the existence of defects in a software system.
- 2. Debugging is a process (Figure 22.2) that locates and corrects these defects.

Figure 22.2 The debugging process

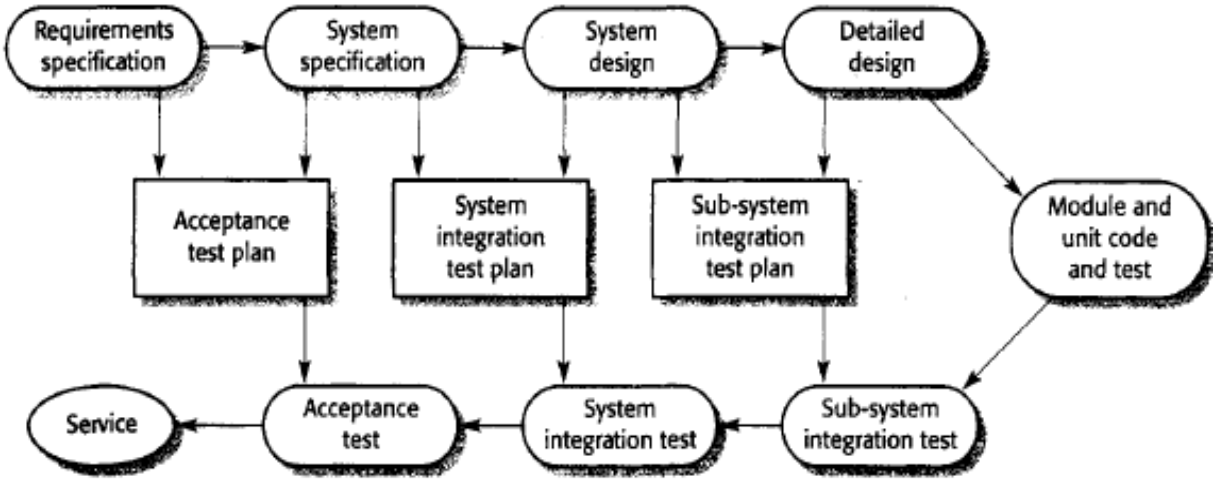


- There is no simple method for program debugging. Skilled debuggers look for patterns in the test output where the defect is exhibited and use their knowledge of the type of defect, the output pattern, the programming language and the programming process to locate the defect. When you are debugging, you can use your knowledge of common programmer errors (such as failing to increment a counter) and match these against the observed patterns.
- Interactive debugging tools are generally part of a set of language support tools that are integrated with a compilation system. They provide a specialized run-time environment for the program that allows access to the compiler symbol table and, from there, to the values of program variables

4.1(2) Planning verification and validation

Verification and validation is an expensive process. For some systems, such as real-time systems with complex non-functional constraints, more than half the system development budget may be spent on V & V. Careful planning is needed to get the most out of inspections and testing and to control the costs of the verification and validation process.

Figure 22.3 Test plans as a link between development and testing



- The software development process model shown in Figure 22.3 is sometimes called the V-model. It is an instantiation of the generic waterfall model.
- This model also breaks down system V & V into a number of stages. Each stage is driven by tests that have been defined to check the conformance of the program with its design and specification.
- As part of the V & V planning process, you should decide on the balance between static and dynamic approaches to verification and validation, draw up standards and procedures for software inspections and testing, establish checklists to drive program inspections and define the software test plan.
- Test planning is concerned with establishing standards for the testing process, not just with describing product tests.
- The major components of a test plan for a large and complex system are shown in Figure 22.4. As well as setting out the testing schedule and procedures, the test plan defines the hardware and software resources that are required.
- This is useful for system managers who are responsible for ensuring that these resources are available to the testing team.

- Test plans should normally include significant amounts of contingency so that slippages in design and implementation can be accommodated and staff redeployed to other activities.

Figure 22.4 The structure of a software test plan

The testing process
A description of the major phases of the testing process. These might be as described earlier in this chapter.

Requirements traceability
Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

Tested items
The products of the software process that are to be tested should be specified.

Testing schedule
An overall testing schedule and resource allocation for this schedule is, obviously, linked to the more general project development schedule.

Test recording procedures
It is not enough simply to run tests; the results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it has been carried out correctly.

Hardware and software requirements
This section should set out the software tools required and estimated hardware utilisation.

Constraints
Constraints affecting the testing process such as staff shortages should be anticipated in this section.

4.1.(3) Software inspections

- Software inspection is a static V & V process in which a software system is reviewed to find errors. omissions and anomalies. Generally, inspections focus on source code. but any readable representation of the software such as its requirements or a design model can be inspected. When you inspect a system, you use knowledge of the system, its application domain and the programming language or design model to discover errors.

- **There are three major advantages of inspection over testing:**

1. During testing, errors can mask (hide) other errors. Once one error is discovered, you can never be sure if other output anomalies are due to a new error or are side effects of the original error. Because inspection is a static process, you don't have to be concerned with interactions between errors. Consequently, a single inspection session can discover many errors in a system.
2. Incomplete versions of a system can be inspected without additional costs. If a program is

incomplete, then you need to develop specialized test harnesses to test the parts that are available. This obviously adds to the system development costs.

3. As well as searching for program defects, an inspection can also consider broader quality attributes of a program such as compliance with standards, portability and maintainability. You can look for inefficiencies, inappropriate algorithms and poor programming style that could make the system difficult to maintain and update.

- Inspections are an old idea. There have been several studies and experiments that have demonstrated that inspections are more effective for defect discovery than program testing.
- **Fagan** (Fagan, 1986) reported that more than 60% of the errors in a program can be detected using informal program inspections.
- **Mills et al**(1987) suggest that a more formal approach to inspection based on correctness arguments can detect more than 90% of the errors in a program. This technique is used in the Cleanroom process.
- Reviews and testing each have advantages and disadvantages and should be used together in the verification and validation process.
- **Gilb** and **Graham** suggest that one of the most effective uses of reviews is to review the test cases for a system. Reviews can discover problems with these tests and can help design more effective ways to test the system.
- In spite of the success of inspections, it has proven to be difficult to introduce formal inspections into many software development organizations.
- There is no doubt that inspections 'front··load' software V & V costs and result in cost savings only after the development teams become experienced in their use.

a) **The program inspection process**

- Program inspections are reviews whose objective is program defect detection.
- The notion of a formalized inspection process was first developed at IBM in the 1970s (Fagan. 1976; Fagan, 1986).

- It is now a fairly widely used method of program verification, especially in critical systems engineering.
- From Fagan's original method, a number of alternative approaches to inspection have been developed (Gilb and Graham, 1993).
- The key difference between program inspections and other types of quality review is that the specific goal of inspections is to find program defects rather than to consider broader design issues.
- Defects may be logical errors, anomalies in the code that might indicate an erroneous condition or noncompliance with organizational or project standards.
- By contrast, other types of review may be more concerned with schedule costs., progress against defined milestones or assessing whether the software is likely to meet organizational goals.
- The program Inspection is a formal process that is carried out by a team of at least four people. Team members systematically analyze the code and point out possible defects.
- Grady and Van Slack (Grady and Van Slack, 1994) suggest six roles, as shown in Fig 22.5.

Figure 22.5 Roles in the inspection process

Role	Description
Author or owner	The programmer or designer responsible for producing the program or document. Responsible for fixing defects discovered during the inspection process.
Inspector	Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues that are outside the scope of the inspection team.
Reader	Presents the code or document at an inspection meeting.
Scribe	Records the results of the inspection meeting.
Chairman or moderator	Manages the process and facilitates the inspection. Reports process results to the chief moderator.
Chief moderator	Responsible for inspection process improvements, checklist updating, standards development, etc.

- The activities in the inspection process are shown in **Figure 22.6**. Before a program Inspection process begins, it is essential that:

1. You have a precise specification of the code to be inspected. It is impossible to inspect a

component at the level of detail required to detect defects without a complete specification.

2. The inspection team members are familiar with the organizational standards.

3. An up-to-date, compilable version of the code has been distributed to all team members. There is no point in inspecting code that is 'almost complete' even if a delay causes schedule disruption.

- The inspection team moderator is responsible for inspection planning. This involves selecting an inspection team, organizing a meeting room and ensuring that the material to be inspected and its specifications are complete. The program to be inspected is presented to the inspection team during the overview stage when the author of the code describes what the program is intended to do. This is followed by a period of individual preparation. Each inspection team member studies the specification and the program and looks for defects in the code.

- The inspection itself should be fairly short (no more than two hours) and should focus on defect detection, standards conformance and poor-quality programming.

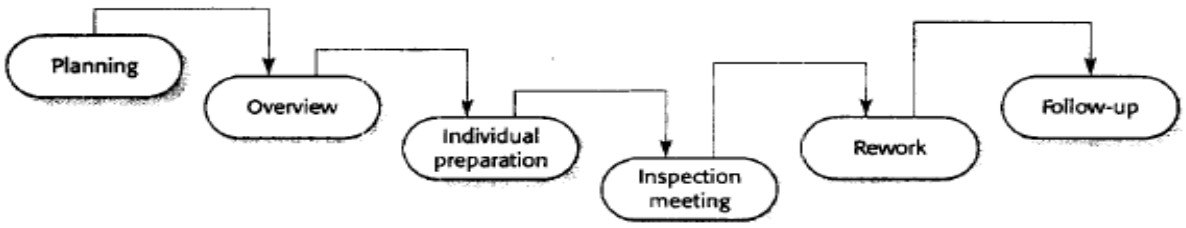
- Following the inspection, the program s author should make changes to it to correct the identified problems. In the follow-up stage, the moderator should decide whether a re-inspection of the code is required.

- During an inspection, a checklist of common programmer errors is often used

to focus the discussion. This checklist can be based on checklist examples from books or from knowledge of defects that are common in a particular application domain. You need different checklists for different programming languages because each language has its own characteristic errors.

- This checklist varies according to programming language because of the different

levels of checking provided by the language compiler. For example, a Java compiler checks that functions have the correct number of parameters, a C compiler. **Figure 22.6**.



- Possible checks that might be made during the inspection process are shown
- in Figure 22.7. Gilb and Graham (Gilb and Graham, 1993) emphasize that each organization should develop its own inspection checklist based on local standards and practices. Checklists should be regularly updated as new types of defects are found.
- **Figure 22.7**

Fault class	Inspection check
Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a delimiter explicitly assigned? Is there any possibility of buffer overflow?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for? If a break is required after each case in case statements, has it been included?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output? Can unexpected inputs cause corruption?
Interface faults	Do all function and method calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

- The time needed for an inspection and the amount of code that can be covered depends on the experience of the inspection team, the programming language and the application domain.
- Both Fagan at IBM and Barnard and Price (Barnard and Price, 1994), who assessed the inspection process for telecommunications software, came to similar conclusions:

1. About 500 source code statements per hour can be presented during the overview stage.
2. During individual preparation, about 125 source code statements per hour can be examined.
3. From 90 to 125 statements per hour can be inspected during the inspection meeting.

- Some organizations (Gilb and Graham, 1993) have now abandoned component testing in favor of inspections. They have found that program inspections are so effective at finding errors that the costs of component testing are not justifiable.
- The introduction of inspections has implications for project management.
- Sensitive management is important if inspections are to be accepted by software development teams. Program inspection is a public process of error detection compared with the more private component testing process.

4.1(4) Automated static analysis

- Inspections are one form of static analysis--you examine the program without executing it.
- Inspections are often driven by checklists of errors and heuristics that identify common errors in different programming languages.
- For some errors and heuristics, it is possible to automate the process of checking programs against this list, which has resulted in the development of automated static analyzers for different programming languages.
- Static analyzers are software tools that scan the source text of a program and detect possible faults and anomalies.
- They parse the program text and thus recognize the types of statements in the program.
- They can then detect whether statements are well-formed. make inferences about the control flow in the program and, in many cases, compute the sell of all possible values for program data.
- They complement the error-detection facilities provided by the language compiler. They can be used as part of the inspection process or as a separate V & V process activity.
- The Intention of automatic static analysis is to draw an inspector's attention to anomalies in the program, such as variables that are used without initialization, variables that are unused

or data whose value could go out of range. Some of the checks that can Je detected by static analysis are shown in Figure 22.8.

- Anomalies are often a result of programming errors or omissions, so they highlight things that could go wrong when the program is executed.

- The stages involved in static analysis include:

1. ***Control flow analysis*** This stage identifies and highlights loops with multiple exit or entry points and unreachable code. Unreachable code is code that is surrounded by unconditional go to statements or that is in a branch of a conditional statement 'where the guarding condition can never be true.

2. Data use analysis This stage highlights how variables in the program are used. It defects variables that are used without previous initialization, variables that are written twice without an

intervening assignment and variables that are declared but never used. Data use analysis also discovers ineffective tests where the test condition is redundant. Redundant conditions are conditions that are either always true or always false.

3. Interface analysis This analysis checks the consistency of routine and procedure declarations and their use. It is unnecessary if a strongly typed language such as Java is used for implementation as the compiler carries out these checks. Interface analysis can detect type errors in weakly typed languages like FORTRAN and C. Interface analysis can also detect functions and procedures that are declared and never called or function results that are never used.

4. Information flow analysis This phase of the analysis identifies the dependencies between input and output variables. While it does not detect anomalies, it shows how the value of each program variable is derived from other variable values. With this information, a code inspection should be able to find values that have been wrongly computed. Information flow analysis can also show the conditions that affect a variable's value.

5. Path analysis This phase of semantic analysis identifies all possible paths through the program and sets out the statements executed in that path. It essentially unravels the program s control and allows each possible predicate to be analysed individually.

Figure 22 8
Automated static analysis checks

Fault class	Static analysis check
Data faults	Variables used before initialisation Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic

- Static analyzers are particularly valuable when a programming language such as C is used. C does not have strict type rules, and the checking that the C compiler can do is limited.
- There is no doubt that, for languages such as C, static analysis is an effective technique for discovering program errors. It compensates for weaknesses in the programming language design.
- All variables must be initialized there are no go to statements so unreachable code is less likely to be created accidentally, and storage management is automatic.
- To illustrate static analysis I use a small C program rather than a Java
- program. Unix and Linux systems include a static analyzer called LINT for C programs.
- LINT provides static checking, which is equivalent to that provided by the compiler in a strongly typed language such as Java. An example of the output produced by LINT is shown in Figure 22.9.

- In this transcript of a Unix terminal session, commands are shown in italics. The first command (line 138) lists the (nonsensical) program. It defines a function with one parameter, called print array, and then calls this function with three parameters. Variables i and c are declared but are never assigned values. The value returned by the function is never used.

Figure 22.9 LINT static analysis

```
138% more lint_ex.c

#include <stdio.h>
printarray (Anarray)
int Anarray;
{
printf("%d",Anarray);
}
main ()
{
int Anarray[5]; int i; char c;
printarray (Anarray, i, c);
printarray (Anarray) ;
}

139% cc lint_ex.c
140% lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

- The line numbered 139 shows the C compilation of this program with no errors reported by the C compiler. This is followed by a call of the LINT static analyzer, which detects and reports program errors.
- The static analyzer shows that the variables;: and i have been used but not initialized and that print array has been called with a different number of arguments than are declared. It also identifies the inconsistent use of the first argument in print array and the fact that the function value is never used. d reports program errors.
- Tool-based analysis cannot replace inspections, as there are some types of error that static analyzers cannot detect.
- To address some of these problems, static analyzers such as LCLint (Orcero, *2000;* Evans anc Larochelle, 2002) support the use of annotations where users define constraints as stylized comments in the program.

4.1(5) Verification and formal methods

- Formal methods of software development are based on mathematical representations of the software, usually as a formal specification.
- These formal methods are mainly concerned with a mathematical analysis of the specification; with transforming the specification to a more detailed, semantically equivalent representation; or with formally verifying that one representation of the system is semantically equivalent to another representation.
- You can think of the use of formal methods as the ultimate static verification technique. They require very detailed analyses of the system specification and the program, and their use is often time consuming and expensive.
- **Formal methods may be** . used at different stages in the V & V process:

1. A formal specification of the system may be developed and mathematically analyzed for inconsistency. This technique is effective in discovering specification errors and omissions,
2. You can formally verify, using mathematical arguments, that the code of a software system is consistent with its specification. This requires a formal specification and is effective in discovering programming and some design errors. A transformational development process where a formal specification is transformed through a series of more detailed representations or a Cleanroom

process may be used to support the formal verification process.

- The argument for the use of formal specification and associated program verification is that formal specification forces a detailed analysis of the specification.
- The argument against the use of formal specification is that it requires specialized notations. These can only be used by specially trained staff and cannot be understood by domain experts. Hence, problems with the system requirements can be concealed by formality.
- Verifying a nontrivial software system takes a great deal of time and requires specialized tools such as theorem provers and mathematical expertise. It is therefore an extremely expensive process and, as the system size increases, the costs of formal verification increase disproportionately. Many people therefore think that formal verification is not cost-effective.

- It is sometimes claimed that the use of formal methods for system development leads to more reliable and safer systems. There is no doubt that a formal system specification is less likely to contain anomalies that must be resolved by the system designer.
- However, formal specification and proof do not guarantee that the software will be reliable in practical use.
- **The reasons for this are:**
- 1. *The specification may not reflect the real requirements of system users.* Lutz (Lutz, 1993) discovered that many failures experienced by users were a consequence of specification errors and omissions that could not be detected by formal system specification. Furthermore, system users rarely understand formal notations so they cannot read the formal specification directly to find errors and omissions.
- *2. The proof may contain errors.* Program proofs are large and complex, so, like large and complex programs, they usually contain errors.
- *3. The proof may assume a usage pattern which is incorrect.* If the system is not used as anticipated, the proof may be invalid.
- **Cleanroom software development**
- A model of the Cleanroom process is shown **in Figure 22.10.** The objective of this approach to software development is zero-defect software.
- The name 'Cleanroom was derived by analogy with semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere.
- Cleanroom development is particularly relevant to this chapter because it has replaced the unit testing of system components by inspections to check the consistency of these components with their specifications.
- **The Cleanroom approach to software development is based on five key strategies:**

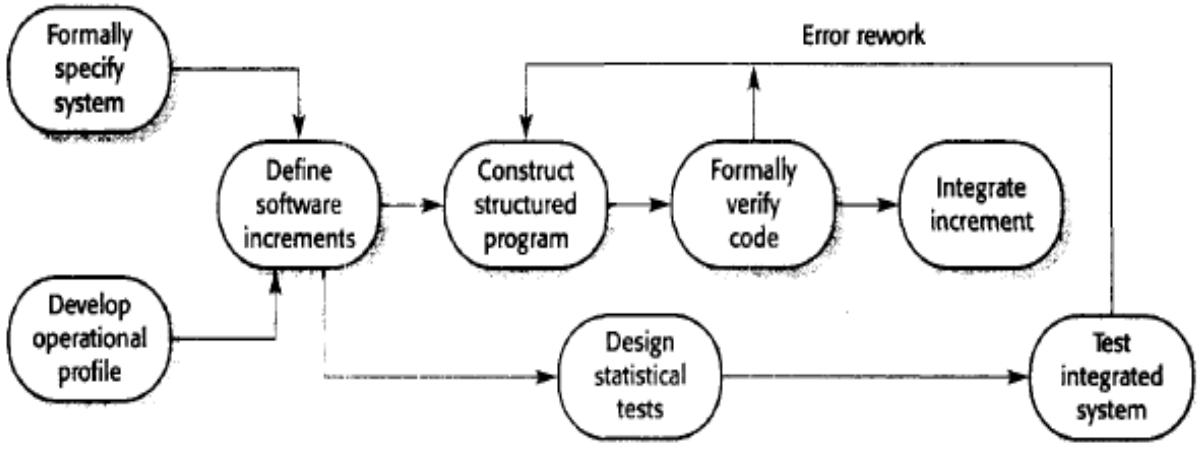
1. *Formal specification* The software to be developed is formally specified. A state transition model that shows system responses to stimuli is used to express the specification.

2. Incremental development The software is partitioned into increments that are developed and validated separately using the Cleanroom process. These increments are specified, with customer input, at an early stage in the process.

3. Structured programming Only a limited number of control and data abstraction constructs are used. The program development process is a process of stepwise refinement of the specification. A limited number of constructs are used and the aim is to systematically transform the specification to create the program code.

4.*Static verification* The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.

5. Statistical testing of the system The integrated software increment is tested statistically, as discussed , to determine its reliability. These statistical tests are based on an operational profile, which is developed in parallel with the system specification as shown in **Figure 22.10**.



- There are three teams involved when the Cleanroom process is used for large system development:
- *1.* ***The specification team*** This group is responsible for developing and maintaining the system specification. This team produces customer-oriented specifications (the user requirements definition) and mathematical specifications for verification. In some cases, when the specification is complete, the specification team also takes responsibility for development.
- *2.* ***The development team*** This team has the responsibility of developing and verifying the software. The software is not executed during the development process. A structured, formal

approach to verification based on inspection of code supplemented with correctness arguments is used.

- *3.* ***The certification team*** This team is responsible for developing a set of statistical tests to exercise the software after it has been developed. These tests are based on the formal specification. Test case development is carried out in parallel with software development. The test cases are used to certify the software reliability. Reliability growth models may be used to decide when to stop testing.
- Use of the Cleanroom approach has generally led to software with very few errors.
- **Cobb and Mills** discuss several successful Cleanroom development projects that had a uniformly low failure rate in delivered systems.
- The approach to incremental development in the Cleanroom process is to deliver critical customer functionality in early increments. Less important system functions are included in later increments.
- Rigorous program inspection is a fundamental part of the Cleanroom process. A state model of the system is produced as a system specification. The approach used for development is based on well-defined transformations that attempt to preserve the correctness at each transformation to a more detailed representation.
- At each stage, the new representation is inspected, and mathematically rigorous arguments are developed that demonstrate that the output of the transformation is consistent with its input.
- Inspection and formal analysis has been found to be very effective in the

Cleanroom process. The vast majority of defects are discovered before execution and are not introduced into the developed software

4.2 (1) Software Testing

- A general testing process that started with the testing of individual program units such as functions or objects. These were then integrated into sub-systems and systems, and the interactions of these units were tested.

- This model of the testing process is appropriate for large system development but for smaller systems, or for systems that are developed through scripting or reuse, there are often fewer distinct stages in the process. A more abstract view of software testing is shown in Figure 23.1.

- The two fundamental testing activities are ***component testing***-testing the parts of the system-and ***system testing***-testing the system as a whole.

- The aim of the ***component testing*** stage is to discover defects by testing individual program components.

- During ***system testing***, these components are integrated to form sub-systems or the complete system. At this stage, system testing should focus on establishing that the system meets its functional and non-functional requirements, and does not behave in unexpected ways.

- The software testing process has two distinct goals:

1. To demonstrate to the developer and the customer that the software meets its requirements. For custom software, this means that there should be at least one test for every requirement in the user and system requirements documents.

For generic software products, it means that there should be tests for all of the system features that will be incorporated in the product release. As discussed some systems may have an explicit acceptance testing phase where the customer formally checks that the delivered system conforms to its specification.

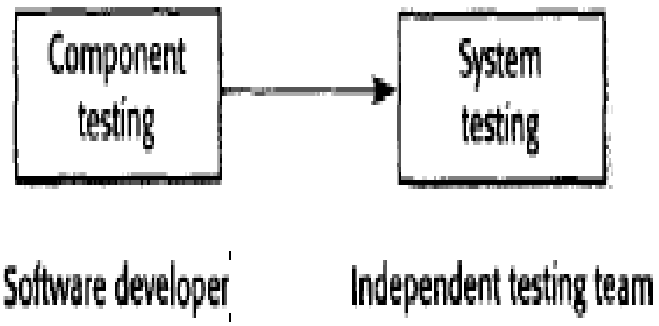
2. To discover faults or defects in the software where the behavior of the software is *incorrect, undesirable or does not conform to its specification.* Defect testing is concerned with rooting out all kinds of undesirable system behavior, such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

- The first goal leads to validation testing, where you expect the system to perform correctly using a given set of test cases that reflect the systems expected use.

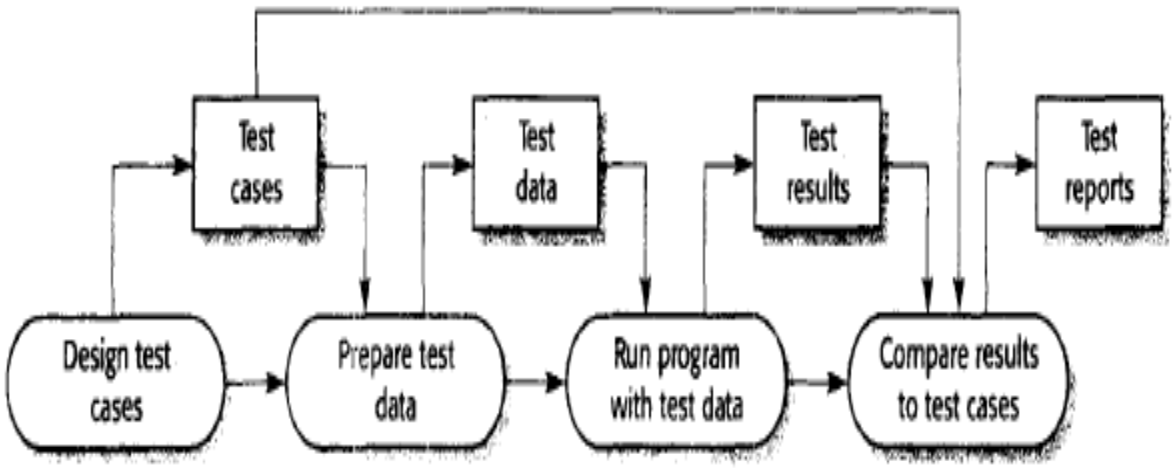
- The second goal leads to defect testing, where the test cases are designed to expose defects. The test cases can be deliberately obscure and need not reflect how the system is normally used.

- For validation testing, a successful test is one where the system performs correctly. For defect testing, a successful test is one that exposes a defect that causes the system to perform incorrectly.

Figure 23.1 Testing phases



- A general model of the testing process is shown in **Figure 23.2**. Test cases are specifications of the inputs to the test and the expected output from the system plus a statement of what is being tested. Test data are the inputs that have been devised to test the system.



- ***Exhaustive testing,*** where every possible program execution sequence is tested, is impossible. Testing, therefore, has to be based on a subset of possible test cases. Ideally, software companies should have policies for choosing this subset rather than leave this to the development team.

- These policies might be based on general testing policies, such as a policy that all program statements should be executed at least once.

- Alternatively. the testing policies may be based on experience of system usage and may focus 011 testing the features of the operational system.

- **For example:**

1 . All system functions that are accessed through menus should be tested.

2. Combinations of functions (e.g., text formatting) that are accessed through the same menu must be tested.

3. Where user input is provided, all functions must be tested with both correct and incorrect input.

- It is clear from experience with major software products such as word processors or spreadsheets that comparable guidelines are normally used during product testing. When features of the software are used in isolation, they normally work.

- Problems arise, as Whittaker explains , when combinations of features have not been tested together

4.2 (2) System testing

- System testing involves integrating two or more components that implement system functions or features and then testing this integrated system.

- In an iterative development process, system testing is concerned with testing an increment to be delivered to the customer; in a waterfall process, system testing is concerned with testing the entire system.

For most complex systems, there are two distinct phases to system testing:

1. ***Integration testing.*** where the test team have access to the source code of the system. When a problem is discovered, the integration team tries to find the source of the problem and identify the components that have to be debugged. Integration testing is mostly concerned with finding defects in the system.

2. Release testing, where a version of the system that could be released to users is tested. Here, the test team is concerned with validating that the system meets its requirements and with ensuring that

the system is dependable.

Release testing is usually 'black-box' testing where the test team is simply concerned with demonstrating that the system does or does not work properly. Problems are reported to the development team whose job is to debug the program. Where customers are involved in release testing, this is sometimes called ***acceptance testing.***

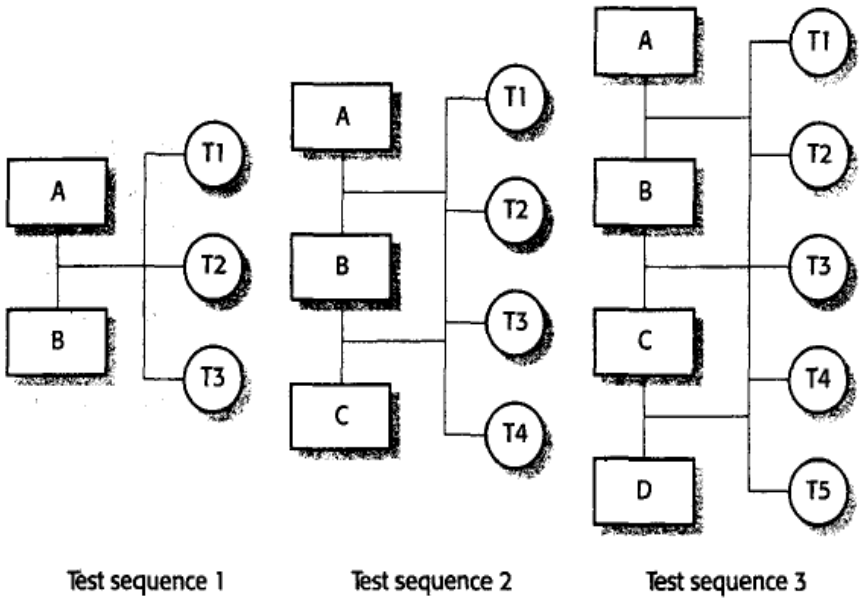
- Fundamentally, you can think of integration testing as the testing of incomplete systems composed of clusters or groupings of system components. Release testing is concerned with testing the system release that is intended for delivery to customers.

1) Integration Testing

- The process of system integration involves building a system from its components and testing the resultant system for problems that arise from component interactions.
- The components that are integrated may be off-the-shelf components, reusable components that have been adapted for a particular system or newly developed components.
- For many large systems, all three types of components are likely to be used.
- Integration testing checks that these components actually work together, are called correctly and transfer the right data at the right time across their interfaces.
- System integration involves identifying clusters of components that deliver some system functionality and integrating these by adding code that makes them work together.
- Sometimes, the overall skeleton of the system is developed first, and components are added to it. This is called ***top-down integration.***
- Alternatively, you may first integrate infrastructure components that provide common services, such as network and database access, then add the functional components. This is ***bottom-up integration.***
- A major problem that arises during integration testing is localizing errors. There are complex interactions between the system components and, when an anomalous output is discovered, you may find it hard to identify where the error occurred.

- To make it easier to locate errors, you should always use an incremental approach to system integration and testing.
- Initially, you should integrate a minimal system configuration and test this system. You then add components to this minimal configuration and test after each added increment.
- In the example shown in Figure 23.3, A, B, C and D are components and n to T5 are related sets of tests of the features incorporated in the system. n, T2 and T3 are first run on a system composed of component A and component B (the minimal system). If these reveal defects, they are corrected.

Figure 23.3
Incremental
integration testing



- Component C is integrated and n, T2 and T3 are repeated to ensure that there have not been unexpected interactions with A and B.
- If problems arise in these tests, this probably means that they are due to interactions with the new component.
- The source of the problem is localized, thus simplifying defect location and repair. Test set T4 is also run on the system.
- Finally, component D is integrated and tested using existing and new tests (T5).
- When planning integration, you have to decide the order of integration of components.

- In a process such as XP, the customer is involved in the development process and decides which functionality should be included in each system increment. Therefore, system integration is driven by customer priorities.
- In such cases, a good rule of thumb is to integrate the components that implement the most frequently used functionality first.
- This means that the components that are most used receive the most testing.
- For example, in the library system, LIBSYS, you should start by integrating the search facility so that, in a minimal system,

users can search for documents that they need. You should then add the functionality to allow users to download a document, then progressively add the components that implement other system features.

- The testing may reveal errors in the interactions between these individual components and other parts of the system.
- Repairing errors may be difficult because a group of components that implement the system feature may have to be changed
- These problems mean that when a new increment is integrated, it is important to rerun the tests for previous increments as well as the new tests that are required

to verify the new system functionality.

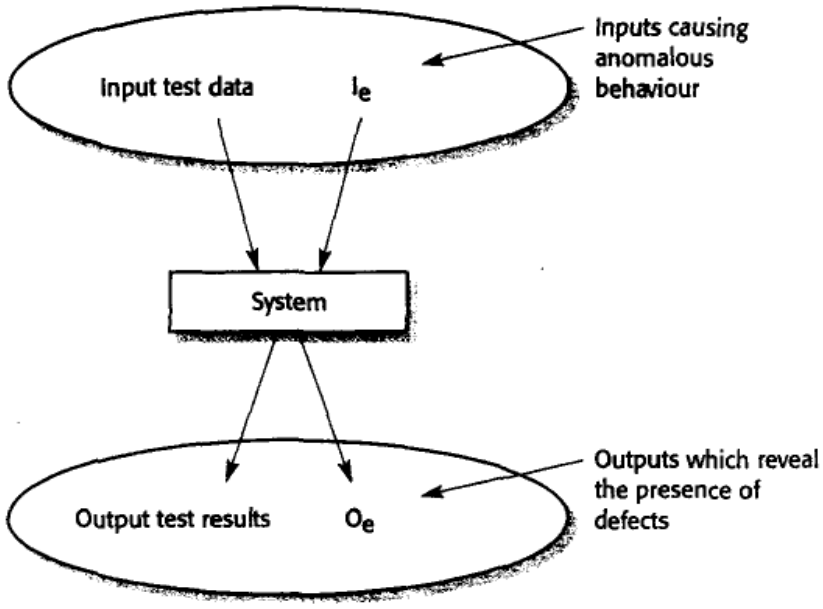
- Rerunning an existing set of tests is called ***regression testing.*** If regression testing exposes problems, then you have to check whether these are problems in the previous increment that the new increment has exposed or whether these are due to the added.
- Regression testing is clearly an expensive process and is impractical without some automated support.

2. Release testing

- Release testing is the process of testing a release of the system that will be distributed to customers. The primary goal of this process is to increase the supplier's confidence that the system meets its requirements.

- If so, it can be released as a product or delivered to the customer. To demonstrate that the system meets its requirements, you have to show that it delivers the specified functionality, performance and dependability, and that it does not fail during normal use.
- Release testing is usually a black-box testing process where the tests are derived from the system specification. The system is treated as a black box whose behavior can only be determined by studying its inputs and the related outputs.
- Another name for this is ***functional testing*** because the tester is only concerned with the functionality and not the implementation of the software
- Figure 23.4 illustrates the model of a system that is assumed in black-box testing. The tester presents inputs to the component or the system and examines the corresponding outputs. If the outputs are not those predicted (i.e., if the outputs are in set 0,) then the test has detected a problem with the software.

Figure 23.4 Black box testing



- When testing system releases, you should try to 'break' the software by choosing test cases that are in the set in Figure 23.4. That is, your aim should be to select inputs that have a high probability of generating system failures (outputs in set Oe)' You use previous experience of what are likely to be successful defect tests and testing guidelines to help you make your choice.

- Authors such as Whittaker have encapsulated their testing experience in a set of guidelines that increase the probability that the defect tests will be successful.

Some examples of these guidelines are:

1. Choose inputs that force the system to generate all error messages.
2. Design inputs that cause input buffers to overflow.
3. Repeat the same input or series of inputs numerous times.
4. Force invalid outputs to be generated.
5. Force computation results to be too large or too small.

- To validate that the system meets its requirements, the best approach to use is scenario-based testing, where you devise a number of scenarios and develop test cases from these scenarios. For example, the following scenario might describe how the library system LIBSYS.

- ***A student in Scotland studying American history has been asked to write a paper on 'Frontier mentality in the American West from 1840 to 1880'. To do this, she needs to find sources from a range of libraries. She logs on to the LIBSYS system and uses the search facility to discover whether she can access original documents from that time. She discovers sources in various US university libraries and downloads copies of some of these. However, for one document, she needs to have confirmation from her university that she is a genuine student and that use is for non- commercial purposes. The student then uses the facility in LIBSYS that can request such permission and registers her request. If granted, the document will be downloaded to the registered library's server and printed for her. She receives a message from LIBS YS telling her that she will receive an e-mail message when the printed document is available for collection.***

- From this scenario, it is possible to device a number of tests that can be applied to the proposed release of LIBSYS:

1. Test the login mechanism using correct and incorrect logins to check that valid users are accepted and invalid users are rejected.
2. Test the search facility using queries against known sources to check that the search mechanism is actually finding documents.
3. Test the system presentation facility to check that information about documents is displayed

properly.

4. Test the mechanism to request permission for downloading.

5. Test the e-mail response indicating that the downloaded document is available.

- For each of these tests, you should design a set of tests that include valid and invalid inputs and that generate valid and invalid outputs.
- Figure 23.5 shows the sequence of operations in the weather station when it responds to a request to collect data for the mapping system. You can use this diagram to identify operations that will be tested and to help design the test cases to execute the test~. Therefore issuing a request for a report will result in the execution of the following thread of methods:

CommsController:request --> WeatherStation:report --> WeatherData:summarise

- The sequence diagram can also be used to identify inputs and outputs that have

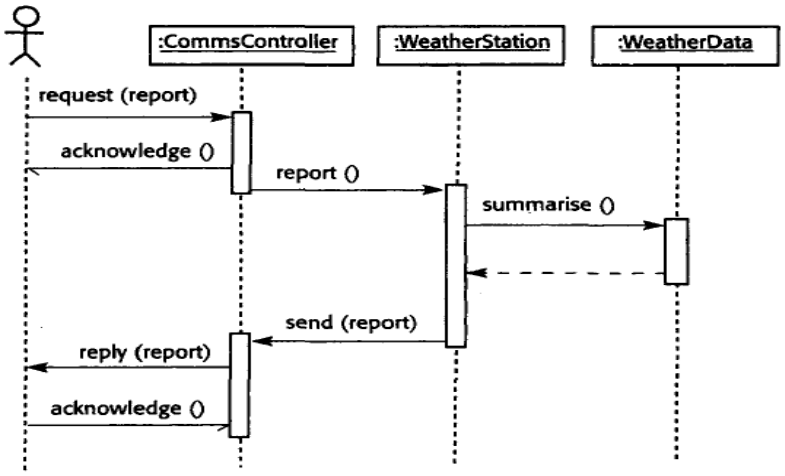
to be created for the test:

1. An input of a request for a report should have an associated acknowledgement and a report should ultimately be returned from the request. During the testing. you should create summarized data that can be used to check that the report is correctly organized.

2. An input request for a report to WeatherStation results in a summarized report being generated. You can test this in isolation by creating raw data corresponding to the summary that you have prepared for the test of CommsController and checking that the WeatherStation object correctly produces this summary.

3. This raw data is also used to test the WeatherData object.

Figure 23.5 Collect weather data sequence chart



3) Performance testing

- Performance tests have to be designed to ensure that the system can process its intended load. This usually involves planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- As with other types of testing, performance testing is concerned both with demonstrating that the system meets its requirements and discovering problems and defects in the system.
- To test whether performance requirements are being achieved, you may have to construct an operational profile. An operational profile is a set of tests that reflect the actual mix of work that will be handled by the system.
- Therefore, if 90% of the transactions in a system are of type A, 5% of type B and the remainder of types C, D, and E, then you have to design the operational profile so that the vast majority of tests are of type A. Otherwise, you will not get an accurate test of the operational performance of the system.
- This approach, of course, is not necessarily the best approach for defect testing. In performance testing, this means stressing the system (hence the name *stress testing)* by making demands that are outside the design limits of the software.
- For example, a transaction processing system may be designed to process up to 300 transactions per second; an operating system may be designed to handle up to 1,000 separate terminals. Stress testing continues these tests beyond the maximum design load of the system until the system fails. This type of testing has two functions:

I. It tests the failure behavior of the system. Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. In these circumstances, it is important that system failure should not cause data corruption or unexpected loss of user services. Stress testing checks that overloading the system causes it to 'fail-soft‘ rather than collapse under its load.

2. It stresses the system and may cause defects to come to light that would not normally be discovered. Although it can be argued that these defects are unlikely to cause system failures in normal usage, there may be unusual combinations of normal circumstances that the stress testing

replicates.

4.2 (3) Component Testing

Component Testing (sometimes called ***unit testing)*** is the process of testing individual components in the system. This is a defect testing process so its goal is to expose faults in these components, for most systems, the developers of components are responsible for component testing. There are different types of component that may be tested at this stage:

l. Individual functions or methods within an object

2. Object classes that have several attributes and methods

3. Composite components made up of several different objects or functions. These composite components have a defined interface that is used to access their functionality.

Individual functions or methods are the simplest type of component and your tests are a set of calls to these routines with different input parameters. You can use the approaches to test case design, discussed in the next section, to design the function or method tests.

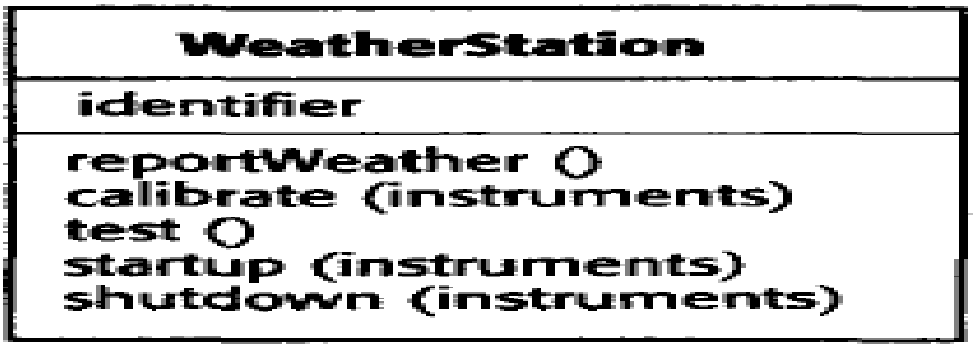
- When you are testing object classes, you should design your tests to provide coverage of all of the features of the object. Therefore, object class testing should include:

1. The testing in isolation of all operations associated with the object

2. The setting and interrogation of all attributes associated with the object

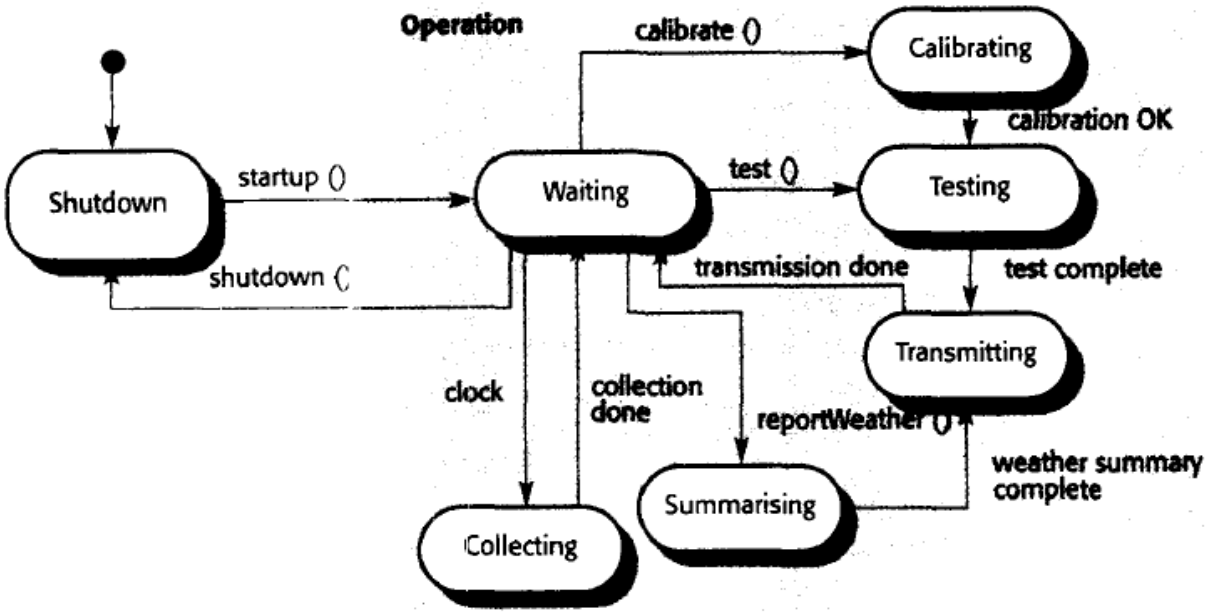
3. The exercise of the object in all possible states. This means that all events that cause a state change in the object should be simulated.

In Figure 23.6. It has only a single attribute, which is its identifier. This is a constant that is set when the weather station is installed. You therefore only need a test that checks whether it has been set up. You need to define test cases for reportWeather, calibrate, test, startup and shutdown. Ideally, you should test methods in isolation but, in some cases, some test sequences are necessary. For example, to test shutdown you need to have executed the startup method.



- To test the states of the weather station, you use a state model as shown in Figure 14.14. Using this model, you can identify sequences of state transitions that have to be tested and define event sequences to force these transitions. In principle, you should test every possible state transition sequence, although in practice this may be too expensive. Examples of state sequences that Should be tested in the weather station include:

Shutdown -> Waiting -> Shutdown
Waiting -> Calibrating -> Testing -> Transmitting -> Waiting
Waiting -> Collecting -> Waiting -> Summarizing --> Transmitting -> Waiting



- If you use: inheritance, this makes it more difficult to design object class tests. Where a superclass provides operations that are inherited by a number of subclasses, all of these subclasses should be tested with all inherited operations.

1. **Interface Testing**

- Many components in a system are not simple functions or objects but are composite components that are made up of several interacting objects.

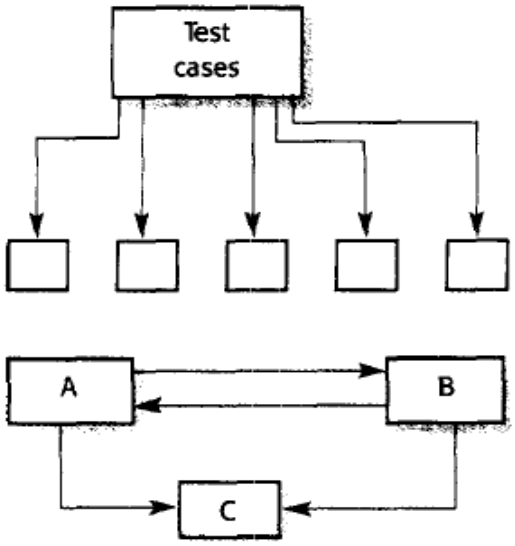
- Testing these composite components then is primarily concerned with testing that the component interface behaves according to its specification.

- Figure 23.7 illustrates this process of interface testing. Assume that components A, Band C have been integrated to create a larger component or sub-system. The test cases are not

applied to the individual components but to the interface of the comp051te component created by combining these components.

- Interface testing is particularly important for object-oriented and component-based development. Objects and components are defined by their interfaces and may be reused in combination with other components in different systems. Interface errors in the composite component cannot be detected by testing the individual objects or components. Errors in the composite component may arise because of interactions between its parts.

Figure 23.7 Interface testing



- There are different types of interfaces between program components and, consequently, different types of interface errors that can occur:

  1. ***Parameter interfaces*** These are interfaces where data or sometimes function references are passed from one component to another.

  2. ***Shared memory interfaces*** These are interfaces where a block of memory is shared between components. Data is placed in the memory by one sub-system and retrieved from there by other sub-systems.

  3. ***Procedural interfaces*** These are interfaces where one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.

  4. Message passing interfaces These are interfaces where one component requests a service from another component by passing a message to it. A return message includes the results of

executing the service. Some object-oriented systems have this form of interface, as do client-server systems.

- Interface errors are one of the most common forms of error in complex systems. These errors fall into three classes:

    1. ***Interface misuse*** A calling component calls some other component and makes an error in the use of its interface. This type of error is particularly common with parameter interfaces where parameters may be of the wrong type, may be passed in the wrong order or the wrong number of parameters may be passed.

    2. ***Interface misunderstanding*** A calling component misunderstands the specification of the interface of the called component and makes assumptions about the behavior of the called component. The called component does not behave as expected and this causes unexpected behavior in the calling component. For example, a binary search routine may be called with an unordered array to be searched. The search would then fail.

    3. ***Timing errors*** These occur in real-time systems that use a shared memory or a message-passing interface. The producer of data and the consumer of data may operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-date information because the producer of the information has not updated the shared interface information.

- **Some general guidelines for interface testing are:**

    I. Examine the code to be tested and explicitly list each call to an external component. Design a set of tests where the values of the parameters to the external components are at the extreme ends of their ranges. These extreme values are most likely to reveal interface inconsistencies.

    2. Where pointers are passed across an interface, always test the interface with null pointer parameters.

    3. Where a component is called through a procedural interface, design tests that should cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.

    4. Use stress testing, as discussed in the previous section, in message-passing system;.

Design tests that generate many more messages than are likely to occur in practice. Timing problems may be revealed in this way.

5. Where several components interact through shared memory, design tests that Val) the order in which these components: are activated. These tests may reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed.

4.2 (4)Test case design

- Test case design is a part of system and component testing where you design the test cases (inputs and predicted outputs) that test the system.
- The goal of the test case design process is to create a set of test cases that are effective in discovering program defects and showing that the system meets its requirements.
- To design a test case, you select a feature of the system or component that you are testing.
- You then select a set of inputs that execute that feature, document the expected outputs or output ranges and, where possible, design an automated check that tests that the actual and expected outputs are the same.
- There are various approaches that you can take to test case design:

1. ***Requirements-based testing*** where test cases are designed to test the system requirements. This is mostly used at the system-testing stage as system requirements are usually implemented by several components. For each requirement, you identify test cases that can demonstrate that the system meets that requirement.

2. ***Partition testing*** where you identify input and output partitions and design tests so that the system executes inputs from all partitions and generates outputs in all partitions. Partitions are groups of data that have common characteristics such as all negative numbers. all names less than 30 characters, all events arising from choosing items on a menu, and so on.

3. ***Structural testing*** where you use knowledge of the program's structure to design tests that exercise all parts of the program. Essentially, when testing a program, you should try to execute each statement at least once. Structural testing helps identify test cases that can make this possible.

4. ***Path Testing*** Path testing is a structural testing strategy whose objective is to exercise

every independent execution path through a component or program. If every independent path is executed, then all statements in the component must have been executed at least once.

1. **Requirements-based testing**

- A general principle of requirements engineering, is that requirements should be testable. That is, the requirement should be written in such a way that a test can be designed so that an observer can check that the requirement has been satisfied.

- Requirements-based testing, therefore, is a systematic approach to test case design where you consider each requirement and derive a set of tests for it.

- Requirements-based testing is validation rather than defect testing-you are trying to demonstrate that the system has properly implemented its requirements.

- For example, consider the requirements for the LIBSYS system .

  1. The user shall be able to search either all of the initial set of databases or select a subset from it.
  2. The system shall provide appropriate viewers for the user to read documents in the document store.
  3. Every order shall be allocated a unique identifier (ORDER_ID) that the user shall be able to copy to the account's permanent storage area.

- Possible tests for the first of these requirements, assuming that a search function has been tested, are:

  a)Initiate user searches for items that are known to be present and known not to be present., where the set of databases includes one database.
  b)Initiate user searches for items that are known to be present and known not to be present., where the set of databases includes two databases.
  c)Initiate user searches for items that are known to be present and known not to be present where the set of databases includes more than two databases.
  d)Select one database from the set of databases and initiate user searches for items that are known to be present and known not to be present.
  e)Select more than one database from the set of databases and initiate searches for items that

are known to be present and known not to be present.

2. Partition Testing

- The input data and output results of a program usually fall into a number of different classes :that have common characteristics such as positive numbers, negative numbers and menu selections.

- Programs normally behave in a comparable way for all members of a class. That is, if you test a program that does some computation

  and requires two positive numbers, then you would expect the program to behave in the same way for all positive numbers.

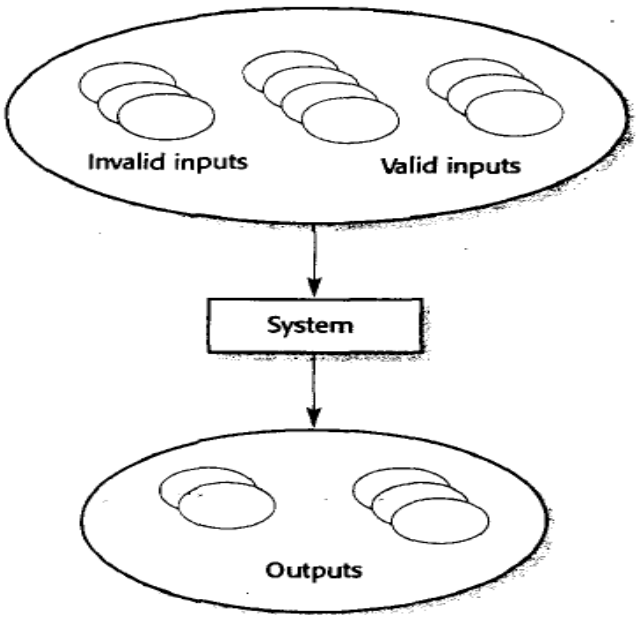
- Because of this equivalent behavior, these classes are sometimes called *equivalence partitions* or *domains* .

- One systematic approach to test case design is based on identifying all partitions for a system or component.

- Test cases are designed so that the inputs or outputs lie within these partitions. Partition testing can be used to design test cases for both systems and components.

- In Figure 2:3.8, each equivalence partition is shown as an ellipse. Input equivalence partitions are sets of data where all of the set members should be processed in an equivalent way. Output equivalence partitions are program outputs that have common characteristics, so they can be considered as a distinct class. Valid and invalid inputs also form equivalence partitions.

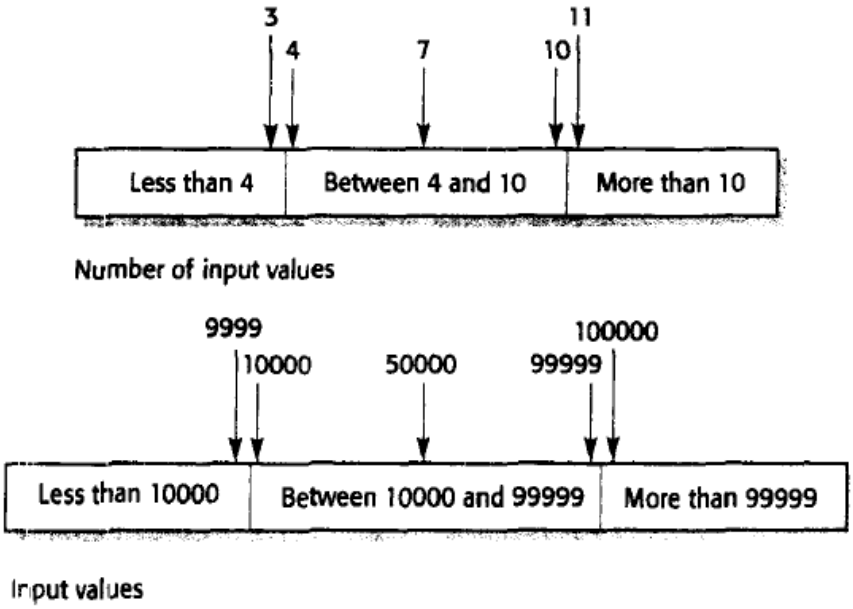
Figure 23.8
Equivalence
partitioning



- Once you have identified a set of partitions, you can chose test cases from each of these partitions.
- A good rule of thumb for test case selection is to choose test cases on the boundaries of the partitions plus cases close to the mid-point of the partition.
- You identify partitions by using the program specification or user documentation and, from experience, where you predict the classes of input value that are likely to detect errors.
- For example, say a program specification states that the program accepts 4 to 8 inputs that are five-digit integers greater than 10,000. Figure 23.9 shows the partitions for this situation and possible test input values.
- From this specification, you can see two equivalence partitions:

  1. Inputs where the key element is a member of the sequence
     (Found =true)
  2. Inputs where the key element is not a sequence member (Found =false).

Figure 23.9 Equivalence partitions



- To illustrate the derivation of test cases, we can use the specification of a search component, shown in Figure 23.10.
- This component searches a sequence of elements for a given element (the key). It returns the position of that element in the sequence.

Figure 23.10 The specification of a search routine

```
procedure Search (Key : ELEM ; T: SEQ of ELEM ;
Found : in out BOOLEAN; L: in out ELEM_INDEX) ;

Pre-condition
    – the sequence has at least one element
    T'FIRST <= T'LAST
Post-condition
    – the element is found and is referenced by L
    ( Found and T (L) = Key)
or
    – the element is not in the sequence
    ( not Found and
    not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key ))
```

- When you are testing programs with sequences, arrays or lists, there are a number of guidelines that are often useful in designing test cases:

1. Test software with sequences that have only a single value. Programmers naturally think of sequences as made up of several values, and sometimes they embed this assumption in their programs. Consequently, the program may not work properly when presented with a single-value sequence.

2. Use different sequences of different sizes in different tests. This decreases the chances that a program with defects will accidentally produce a correct output because of some accidental characteristics of the input.

3. Derive tests so that the first, middle and last elements of the sequence are accessed. This approach reveals problems at partition boundaries.

- **From these guidelines, two more equivalence partitions can be identified:**

1. The input sequence has a single value.
2. The number of elements in the input sequence is greater than 1.

- You then identify further partitions by combining these partitions-for example, the partition where the number of elements in the sequence is greater than 1 and the element is not in the sequence. Figure 23.11 shows the partitions that have identified to test the search component.

- A set of possible test cases based on these partitions is also shown in Figure 23.11. If the key element is not in the sequence, the value of L is undefined *('n').* The guideline that different sequences of different sizes should be used has been applied in these test cases

- The set of input values used to test the search routine is not exhaustive. The routine may fail if the input sequence happens to include the elements 1, 2, 3 and 4.

Figure 23.11 Equivalence partitions for search routine

Sequence	Element
Single value	In sequ nce
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

3. Structural Testing

- Structural testing is an approach to test case design where the tests are derived from knowledge of the software's structure and implementation.
- This approach is sometimes called 'white-box', 'glass-box' testing, or "clear-box' testing to distinguish it from black-box testing.
- Understanding the algorithm used in a component can help you identify further partitions and test cases.
- To illustrate this, the search routine specification (Figure 23.10) as a binary search routine (Figure 23.14) is implemented.
- Of course, this has stricter pre-conditions. The sequence is implemented as an array that array must be ordered and the value of the lower bound of the array must be less than

the value of the upper bound.

- By examining the code of the search routine, you can see that binary searching involves splitting the search space into three parts. Each of these parts makes up an equivalence partition (Figure 23.13). You then design test cases where the key lies at the boundaries of each of these partitions.

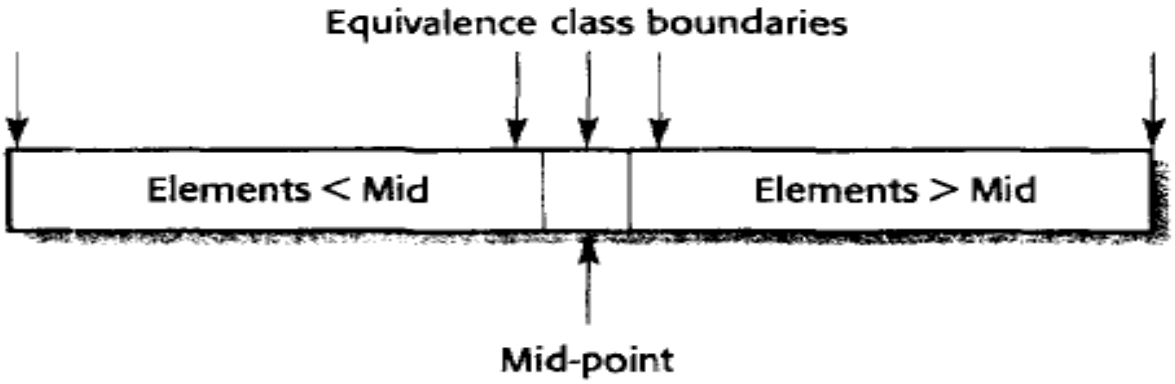


Figure 23.14 Java implementation of a binary search routine

```
class BinSearch {

// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types
by
// reference to a function and so return two values
// the key is -1 if the element is not found

      public static void search ( int key, int [] elemArray, Result r )
      {
1.        int bottom = 0 ;
2.        int top = elemArray.length - 1 ;
          int mid ;
3.        r.found = false ;
4.        r.index = -1 ;
5.        while ( bottom <= top )
          {
6             mid = (top + bottom) / 2 ;
7             if (elemArray [mid] == key)
              {
8                   r.index = mid ;
9                   r.found = true ;
10                   return ;
              } // if part
              else
              {
11                  if (elemArray [mid] < key)
12                      bottom = mid + 1 ;
                    else
13                      top = mid - 1 ;
               }
          } //while loop
14.   } // search
} //BinSearch
```

- This leads to a revised set of test cases for the search routine, as shown in Figure 23.15. Notice that , modified the input array so that it is arranged in ascending order and have added further tests where the key element is adjacent to the midpoint of the array.

Figure 23.15 Test cases for search routine

Input array (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

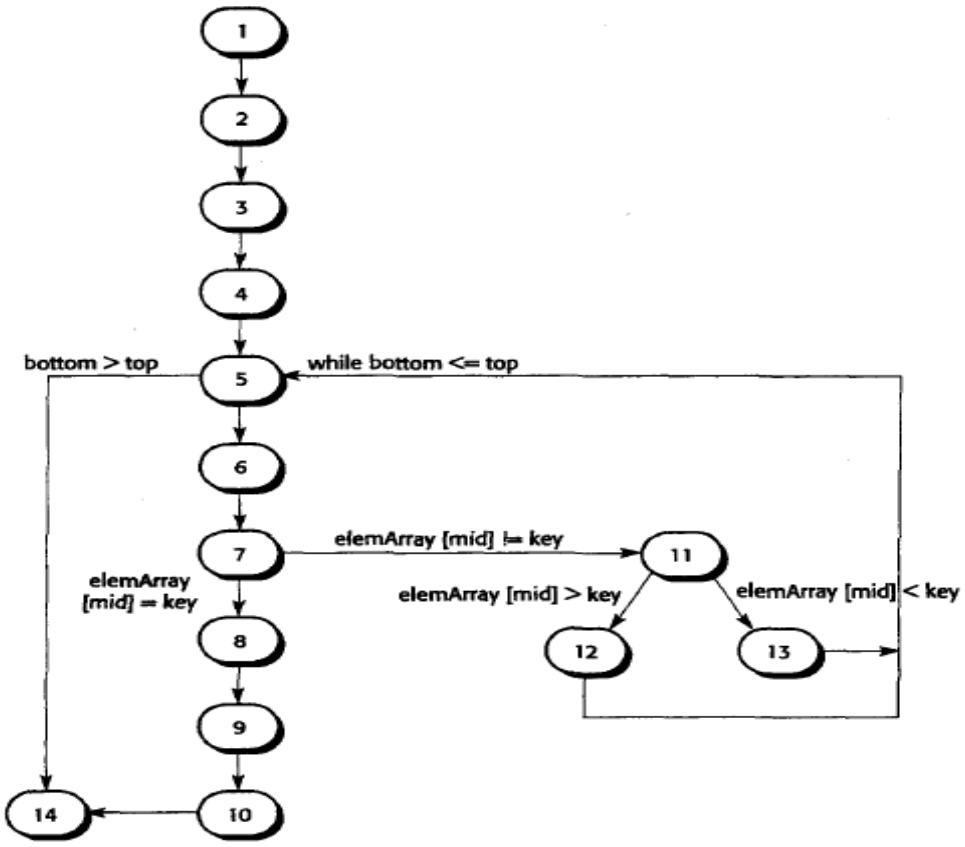
4.

Path Testing

- Path testing is a structural testing strategy whose objective is to exercise every independent execution path through a component or program. If every independent path is executed, then all statements in the component must have been executed at least once.
- The number of paths through a program is usually proportional to its size. Path testing techniques are therefore mostly used during component testing.
- Path testing does not test all possible combinations of all paths through the program.
- For any components apart from very trivial ones without loops, this is an impossible objective. There are an infinite number of possible path combinations in programs with loops.
- Even when all program statements have been executed at least once, program defects may still show up when particular paths are combined.
- The starting point for path testing is a program flow graph. This is a skeletal model of all paths through the program.

- A flow graph consists of nodes representing decisions and edges showing flow of control. The flow graph is constructed by replacing program control statements by equivalent diagrams.
- If there are no goto statements in a program, it is a simple process to derive its flow graph. Each branch in a conditional statement (if-then-else or case) is shown as a separate path.
- An arrow looping back to the condition node denotes a loop. I have drawn the flow graph for the binary search method in Figure 23.16.
- To make the correspondence between this and the program in Figure 23.14 more obvious, it is shown that each statement as a separate node where the node number corresponds to the line number in the program.
- The objective of path testing is to ensure that each independent path through the program is executed at least once.

Figure 23.16 Flow graph for a binary search routine



- The flow graph for the binary search procedure is shown in Figure 23.16 where each node represents a line in the program with an executable statement.

- By tracing the flow, therefore, you can see that the paths through the binary search flow graph are:

  1,2,3,4,5,6,7,8,9, 10, 14
  I, 2, 3, 4, 5, 14
  1,2,3,4,5,6,7, 11, 12,5, ..
  1,2,3,4,6,7,2,11,13,5, ..
- If all of these paths are executed, we can be sure that every statement in the method has been executed at least once and that every branch has been exercised

  for true and false conditions.
- You can find the number of independent paths in a program by computing the *cyclomatic complexity* of the program flow graph.
- A simple condition is logical expression without 'and' or 'or connectors. If the program includes compound conditions, which are logical expressions including 'and' or 'or' connectors, then you count the number of simple conditions in the compound conditions when calculating the cyclomatic complexity.
- Therefore, if there are six if-statements and a while loop and all conditional expressions are simple, the cyclomatic complexity is 8. If one conditional expression is a compound expression such as 'if A and B or C', then you count this as three simple conditions. The cyclomatic complexity is therefore 10. The cyclomatic complexity of the binary search algorithm (Figure 23.14) is 4 because there are three simple conditions at lines 5, 7 and 11.

4.2 (5)Test Automation

- Testing is an expensive and laborious phase of the software process. As a result, testing tools were among the first software tools to be developed.
- These tools now offer a rouge of facilities and their use can significantly reduce the costs of testing.
- One approach to test automation (Mosley and Posey, 2002) where a testing framework such as JUnit (Massol and Husted, 2003)

- is used for regression testing.
- JUnit is a set of Java classes that the user extends to create an automated testing environment. Each individual test is implemented as an object and a test runner runs all of the tests.
- The tests themselves should be written in such a way that they indicate whether the tested system has behaved as expected.
- A software testing workbench is an integrated set of tools to support the testing process. In addition to testing frameworks that support automated test execution, a workbench may include tools to simulate other parts of the system and to generate system test data.

  Figure 23.17 shows some of the tools that might be included in such a testing workbench:

  1. ***Test manager*** Manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested. Test automation frameworks such as JUnit are examples of test managers.

  2. ***Test data generator*** Generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.

  3. ***Oracle*** **Generates** predictions of expected test results. Oracles may either be previous program versions or prototype systems. Back-ta-back testing involves running the oracle and the program to be tested in parallel. Differences in their outputs are highlighted.

  4. ***File comparator*** Compares the results of program tests with previous test results and reports differences between them. Comparators are used in regression testing where the results of executing different versions are compared. Where automated tests are used, this may be called from within the tests themselves.
- A software testing workbench is an integrated set of tools to support the testing process. In addition to testing frameworks that support automated test execution, a workbench may include tools to simulate other parts of the system and to generate system test data.

  Figure 23.17 shows some of the tools that might be included in such a testing workbench:

1. ***Test manager*** Manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested. Test automation frameworks such as JUnit are examples of test managers.

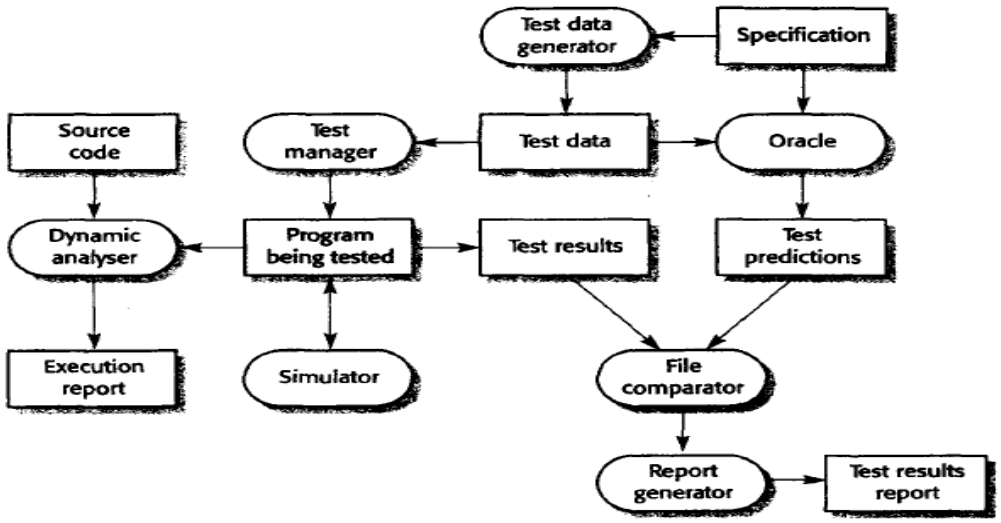
2. ***Test data generator*** Generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.

3. ***Oracle*** **Generates** predictions of expected test results. Oracles may either be previous program versions or prototype systems. Back-ta-back testing involves running the oracle and the program to be tested in parallel.

Differences in their outputs are highlighted.

4. ***File comparator*** Compares the results of program tests with previous test results and reports differences between them. Comparators are used in regression testing where the results of executing different versions are compared. Where automated tests are used, this may be called from within the tests themselves.

- **Figure 23.17**



4.3 (1) Software Quality Assurance

- Quality control and assurance are essential activities for any business that produces products to be used by others.

- The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world.

- During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement as key elements of quality management.
- Today, every company has mechanisms to ensure quality in its products.
- The history of quality assurance in software development parallels the history of quality in hardware manufacturing.
- Software quality assurance is a "planned and systematic pattern of actions" that are required to ensure high quality in software.
- The scope of quality assurance responsibility might best be characterized by paraphrasing a once-popular automobile commercial: "Quality Is Job #1."
- The implication for software is that many different constituencies have software quality assurance responsibility—software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group.
- The SQA group serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view.
- Does the software adequately meet the quality factors noted in Chapter 14?
- Has software development been conducted according to pre-established standards?
- Have technical disciplines properly performed their roles as part of the SQA activity?
- The SQA group attempts to answer these and other questions to ensure that software quality is maintained.

4.3 (2) Elements of Software Quality Assurance

- Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality. These can be summarized in the following manner:

1) Standards. The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents. Standards may be adopted voluntarily by a software engineering organization or imposed by the customer or other

stakeholders. The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them.

2) Reviews and audits. Technical reviews are a quality control activity performed by software engineers for software engineers (Chapter 15). Their intent is to uncover errors. Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work. For example, an audit of the review process might be conducted to ensure that reviews are being performed in a manner that will lead to the highest likelihood of uncovering errors.

3) Testing. Software testing is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of achieving its primary goal.

4) Error/defect collection and analysis. The only way to improve is to measure how you're doing. SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.

5) Change management. Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. SQA ensures that adequate change management practices have been instituted.

4.3 (3) SQA Tasks, Goals and Metrics

- Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.
- Software engineers address quality (and perform quality control activities) by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

SQA Tasks

The Software Engineering Institute recommends a set of SQA

actions that address quality assurance planning, oversight, record keeping, analysis, and reporting. These actions are performed (or facilitated) by an independent SQA group that:

1) Prepares an SQA plan for a project.

- The plan is developed as part of project planning and is reviewed by all stakeholders. Quality assurance actions performed by the software engineering team and the SQA group are governed by the plan.

- The plan identifies evaluations to be performed, audits and reviews to be conducted, standards that are applicable to the project, procedures for error reporting and tracking, work products that are produced by the SQA group, and feedback that will be provided to the software team.

2) Participates in the development of the project's software process description.

- The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

3)Reviews software engineering activities to verify compliance with the defined software process.

- The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

4) Audits designated software work products to verify compliance with those defined as part of the software process.

- The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

5) Ensures that deviations in software work and work products are documented and handled according to a documented procedure.

- Deviations may be encountered in the project plan, process description,

applicable standards, or software engineering work products.

6) Records any noncompliance and reports to senior management.

- Noncompliance items are tracked until they are resolved.

Goals, Attributes, and Metrics

The SQA actions described in the preceding section are performed to achieve a set of pragmatic goals:

- **Requirements quality.** The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow. SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.

- **Design quality.** Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements. SQA looks for attributes of the design that are indicators of quality.

- **Code quality.** Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability. SQA should isolate those attributes that allow a reasonable analysis of the quality of code.

- **Quality control effectiveness.** A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result. SQA analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

- Figure 16.1 identifies the attributes that are indicators for the existence of quality for each of the goals discussed. Metrics that can be used to indicate the relative strength of an attribute are also shown.

Goal	Attribute	Metric
Requirement quality	Ambiguity	Number of ambiguous modifiers (e.g., many, large, human-friendly)
	Completeness	Number of TBA, TBD
	Understandability	Number of sections/subsections
	Volatility	Number of changes per requirement
		Time (by activity) when change is requested
	Traceability	Number of requirements not traceable to design/code
	Model clarity	Number of UML models
		Number of descriptive pages per model
		Number of UML errors
Design quality	Architectural integrity	Existence of architectural model
	Component completeness	Number of components that trace to architectural model
		Complexity of procedural design
	Interface complexity	Average number of pick to get to a typical function or content
		Layout appropriateness
	Patterns	Number of patterns used
Code quality	Complexity	Cyclomatic complexity
	Maintainability	Design factors (Chapter 8)
	Understandability	Percent internal comments
		Variable naming conventions
	Reusability	Percent reused components
	Documentation	Readability index
QC effectiveness	Resource allocation	Staff hour percentage per activity
	Completion rate	Actual vs. budgeted completion time
	Review effectiveness	See review metrics (Chapter 14)
	Testing effectiveness	Number of errors found and criticality
		Effort required to correct an error
		Origin of error

4.3 (4) Formal Approaches to SQA

- Software quality is everyone's job and that it can be achieved through competent software engineering practice as well as through the application of technical reviews, a multi-tiered testing strategy, better control of software work products and the changes made to them, and the application of accepted software engineering standards.
- In addition, quality can be defined in terms of a broad array of quality attributes and measured (indirectly) using a variety of indices and metrics.
- Over the past three decades, a small, but vocal, segment of the software engineering community has argued that a more formal approach to software quality assurance is required. It can be argued that a computer program is a mathematical object.
- A rigorous syntax and semantics can be defined for every programming language, and a rigorous approach to the specification of software requirements is available.
- If the requirements model (specification) and the programming language can be represented in a rigorous manner, it should be possible to apply mathematic proof.

4.3 (5) Statistical Software Quality Assurance

- Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

1. Information about software errors and defects is collected and categorized.

2. An attempt is made to trace each error and defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).

3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the *vital few*).

4. Once the vital few causes have been identified, move to correct the problems that have caused the errors and defects.

1) Generic Example

- To illustrate the use of statistical methods for software engineering work, assume that a software engineering organization collects information on errors and defects for a period of one year. Some of the errors are uncovered as software is being developed.

- Others (defects) are encountered after the software has been released to its end users. Although hundreds of different problems are uncovered, all can be tracked to one (or more) of the following causes:

- Incomplete or erroneous specifications (IES)
- Misinterpretation of customer communication (MCC)
- Intentional deviation from specifications (IDS)

  • Violation of programming standards (VPS)
  • Error in data representation (EDR)
  • Inconsistent component interface (ICI)
  • Error in design logic (EDL)
  • Incomplete or erroneous testing (IET)
  • Inaccurate or incomplete documentation (IID)

• Error in programming language translation of design (PLT)

• Ambiguous or inconsistent human/computer interface (HCI)

• Miscellaneous (MIS)

- To apply statistical SQA, the table in Figure 16.2 is built. The table indicates that IES, MCC, and EDR are the vital few causes that account for 53 percent of all errors.
- It should be noted, however, that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once the vital few causes are determined, the software engineering organization can begin corrective action.
- For example, to correct MCC, you might implement requirements gathering techniques to improve the quality of customer communication and specifications.
- To improve EDR, you might acquire tools for data modeling and perform more stringent data design reviews.
- It is important to note that corrective action focuses primarily on the vital few. As the vital few causes are corrected, new candidates pop to the top of the stack.
- Statistical quality assurance techniques for software have been shown to provide substantial quality improvement .
- In some cases, software organizations have achieved a 50 percent reduction per year in defects after applying these techniques.
- The application of the statistical SQA and the Pareto principle can be summarized

in a single sentence: ***Spend your time focusing on things that really matter, but first be sure that you understand what really matters!***

2) Six Sigma for Software Engineering

- *Six Sigma* is the most widely used strategy for statistical quality assurance in industry today.
- Originally popularized by Motorola in the 1980s, the Six Sigma strategy "is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects' in manufacturing and service-related processes" . The term Six Sigma is derived from six standard deviations.

The Six Sigma methodology defines three core steps:

• ***Define*** customer requirements and deliverables and project goals via well-defined methods of customer communication.

• ***Measure*** the existing process and its output to determine current quality performance (collect defect metrics).

• ***Analyze*** defect metrics and determine the vital few causes.

- If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

  • ***Improve*** the process by eliminating the root causes of defects.

  • ***Control*** the process to ensure that future work does not reintroduce the causes of defects.

- These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

- If an organization is developing a software process (rather than improving an existing process), the core steps are augmented as follows:

  • ***Design*** the process to (1) avoid the root causes of defects and (2) to meet customer requirements.

  • ***Verify*** that the process model will, in fact, avoid defects and meet customer requirements.

  This variation is sometimes called the DMADV (define, measure, analyze, design, and verify) method.

4.3 (6) Software Reliability

- There is no doubt that the ***reliability*** of a computer program is an important element of its overall quality. If a program repeatedly and frequently fails to perform, it matters little whether other software quality factors are acceptable.

- Software reliability, unlike many other quality factors, can be measured directly

  and estimated using historical and developmental data.

- ***Software reliability*** is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time". To illustrate, program *X* is estimated to have a reliability of 0.999 over eight elapsed processing hours.

- In other words, if program *X* were to be executed 1000 times and require a total of eight hours of elapsed processing time (execution time), it is likely to operate correctly (without failure) 999 times.
- Whenever software reliability is discussed, a pivotal question arises: What is meant by the term *failure*? In the context of any discussion of software quality and reliability, failure is nonconformance to software requirements. Yet, even within this definition, there are gradations.
- Failures can be only annoying or catastrophic. One failure can be corrected within seconds, while another requires weeks or even months to correct. Complicating the issue even further, the correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures.

1) **Measures of Reliability and Availability**

- **Software reliability** attempted to extrapolate the mathematics of hardware

  reliability theory to the prediction of software reliability. Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects.
- In hardware, failures due to physical wear (e.g., the effects of temperature, corrosion, shock) are more likely than a design-related failure. Unfortunately, the opposite is true for software. In fact, all software failures can be traced to design or implementation problems; wear (see Chapter 1) does not enter into the picture.
- There has been an ongoing debate over the relationship between key concepts in hardware reliability and their applicability to software. Although an irrefutable link has yet to be established, it is worthwhile to consider a few simple concepts that apply to both system elements.
- If we consider a computer-based system, a simple measure of reliability is *meantime-between-failure* (MTBF):

MTBF MTTF MTTR

where the acronyms MTTF and MTTR are *mean-time-to-failure* and *mean-time-to repair*, respectively.

- Many researchers argue that MTBF is a far more useful measure than other quality-related software metrics Stated simply, an end user is concerned with failures, not with the total defect count. Because each defect contained within a program does not have the same failure rate, the total defect count provides little indication of the reliability of a system.
- For example, consider a program that has been in operation for 3000 processor hours without failure. Many defects in this program may remain undetected for tens of thousand of hours before they are discovered.
- The MTBF of such obscure errors might be 30,000 or even 60,000 processor hours. Other defects, as yet undiscovered, might have a failure rate of 4000 or 5000 hours. Even if every one of the first category of errors (those with long MTBF) is removed, the impact on software reliability is negligible.
- However, MTBF can be problematic for two reasons: (1) it projects a time span between failures, but does not provide us with a projected failure rate, and (2) MTBF can be misinterpreted to mean average life span even though this is *not* what it implies.
- An alternative measure of reliability is *failures-in-time* (FIT)—a statistical measure of how many failures a component will have over one billion hours of operation.
- Therefore, 1 FIT is equivalent to one failure in every billion hours of operation.
- In addition to a reliability measure, you should also develop a measure of availability.
- *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times 100\%$$

- The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software.
- **2) Software Safety**
- ***Software safety*** is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.
- A modeling and analysis process is conducted as part of software safety. Initially, hazards are identified and categorized by criticality and risk.
- For example, some of the hazards associated with a computer-based cruise control for an automobile might be: (1) causes uncontrolled acceleration that cannot be stopped, (2) does not respond to depression of brake pedal (by turning off), (3) does not engage when switch is activated, and (4) slowly loses or gains speed. Once these system-level hazards are identified, analysis techniques are used to assign severity and probability of occurrence.
- To be effective, software must be analyzed in the context of the entire system.
- Once hazards are identified and analyzed, safety-related requirements can be specified for the software. That is, the specification can contain a list of undesirable events and the desired system responses to these events.
- The role of software in managing undesirable events is then indicated. Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them.
- Software reliability uses statistical analysis to determine the likelihood that a software failure will occur. However, the occurrence of a failure does not necessarily result in a hazard or mishap.

- Software safety examines the ways in which failures result in conditions that can lead to a mishap. That is, failures are not considered in a vacuum, but are evaluated in the context of an entire computer-based system and its environment.

4.3 (7) The ISO 9000 Quality Standards

- A ***quality assurance system*** may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management.
-
- Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations by meeting their specifications.
- ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.
- To become registered to one of the quality assurance system models contained in ISO 9000, a company's quality system and operations are scrutinized by third-party auditors for compliance to the standard and for effective operation.
- Upon successful registration, a company is issued a certificate from a registration body represented by the auditors.
- The requirements delineated by ISO 9001:2000 address topics such as management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques.
- In order for a software organization to become registered to ISO 9001:2000, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed. If you desire further information on ISO 9001:2000.

- **The SQA Plan**

- The *SQA Plan* provides a road map for instituting software quality assurance. Developed by the SQA group (or by the software team if an SQA group does not exist), the plan serves as a template for SQA activities that are instituted for each software project.

- A standard for SQA plans has been published by the IEEE. The standard recommends a structure that identifies:

  (1) the purpose and scope of the plan,
  (2) a description of all software engineering work products (e.g., models, documents, source code) that fall within the purview of SQA,
  (3) all applicable standards and practices that are applied during the software process,
  (4) SQA actions and tasks(including reviews and audits) and their placement throughout the software process,
  (5) the tools and methods that support SQA actions and tasks,
  (6) software configuration management procedures,
  (7) methods for assembling, safeguarding, and maintaining all SQA-related records, and
  (8) organizational roles and responsibilities relative to product quality.

UNIT V

Project Management: Management activities, Project planning, Project scheduling, Risk management. **Software Cost Estimation:** Software Productivity, Estimation techniques- The COCOMO II Model, Project duration and staffing.

5.1 (1) Project Management

- Software project management is an essential part of software engineering. Good management cannot guarantee project success.
- However, bad management usually results in project failure: The software is delivered late, costs more than originally estimated and fails to meet its requirements.
- Software managers are responsible for planning and scheduling project development. They supervise the work to ensure that it is carried out to the required standards and monitor progress to check that the development is on time and within budget.
- We need software project management because professional software engineering is always subject to organizational budget and schedule constraints.
- The software project manager's job is to ensure that the software project meets these constraints and delivers software that contributes to the goals of the company developing the software.
- Software managers do the same kind of job as other engineering project managers. However, software engineering is different from other types of engineering in a number of ways.
- These distinctions make software management particularly difficult. Some of the differences are:
- 1. ***The product* is *intangible*** The manager of a shipbuilding project or of a civil engineering project can see the product being developed. If a schedule slips, the 'effect on the product is visible-parts of the structure are obviously unfinished. Software is intangible. It cannot be

seen or touched. Software project managers cannot see progress. They rely on others to produce the documentation needed to review progress.

- *2.* ***There are no standard software processes*** In engineering disciplines with a long history, the process is tried and tested. The engineering process for some types of system, such as bridges and buildings is well understood. However, software processes vary dramatically from one organization to another. Although our understanding of these processes has developed significantly in the past few years, we still cannot reliably predict when a particular software process is likely to cause development problems.. This is especially true when the software project is part of a wider systems engineering project.

- ***3. Large software projects are often one-off projects*** Large software projects are usually different in some ways from previous projects. Therefore, even managers who have a large body of previous experience may find it difficult to anticipate problems. Furthermore, rapid technological changes in computers and communications can make a manager s experience obsolete. Lessons learned from previous projects may not be transferable to new projects.

- Because of these problems, it is not surprising that some software projects are late, over budget and behind schedule. Software systems are often new and technically innovative. Engineering projects (such as new transport systems) that are innovative often also have schedule problems.

5.1 (2) Management activities

- It is impossible to write a standard job description for a software manager. The job varies tremendously depending on the organization and the software product being developed. However, most managers take responsibility at some stage for some or all of the following activities:

1) Proposal writing

2) Project planning and scheduling

3) Project cost

4) Project monitoring and reviews

5) **Personnel selection and evaluation**

6) **Report writing and presentations**

1) **Proposal writing**

- The first stage in a software project may involve writing a proposal to win a contract to carry out the work.
- The proposal describes the objectives of the project and how it will be carried out. It usually includes cost and schedule estimates, and justifies why the project contract should be awarded to a particular organization or team.
- Proposal writing is a critical task as the existence of many software organizations depends on having enough proposals accepted and contracts awarded.

  There can be no set guidelines for this task; proposal writing is a skill that you acquire through practice and experience.

2) **Project planning and scheduling**

Project planning is concerned with identifying the activities. milestones and deliverables produced by a project. A plan is drawn up to guide the development towards the project goals.

3) **Project cost**

Cost estimation is a related activity that is concerned with estimating the resources required to accomplish the project plan.

4) **Project monitoring and reviews**

- Project monitoring is a continuing project activity. The manager must keep track of the progress of the project and compare actual and planned progress and costs.
- Although most organizations have formal mechanisms for monitoring, a skilled manager can often form a clear picture of what is going on through informal discussions with project staff.

- Informal monitoring can often predict potential project problems by revealing difficulties as they occur.
- For example, daily discussions with project staff might reveal a particular problem in finding some software fault. Rather than waiting for a schedule slippage to be reported, the software manager might assign some expert to the problem or might decide that it should be programmed around.
- During a project, it is normal to have a number of formal project management reviews. They are concerned with reviewing overall progress and technical development of the project and checking whether the project and the goals of the organization paying for the software are still aligned.
- The outcome of a review may be a decision to cancel a project. The development time for a large software project may be several years. During that time, organizational objectives are almost certain to change. These changes may mean that the software is no longer required or that the original project requirements are inappropriate.
- Management may decide to stop software development or to change the project to accommodate the changes to the organizational objectives.
- Project managers usually have to select people to work on their project. Ideally, skilled staff with appropriate experience will be available to work on the project.
- However, in most cases, managers have to settle for a less-than-ideal project team. The reasons for this are:

1. The project budget may not cover the use of highly paid staff. Less experienced, less well-paid staff may have to be used.
2. Staff with the appropriate experience may not be available either within an organization or externally. It may be impossible to recruit new staff to the project. Within the organization, the best people may already be allocated to other projects.
3. The organization may wish to develop the skills of its employees. Inexperienced staff may be assigned to a project to learn and to gain experience.

- The software manager has to work within these constraints when selecting project staff However, problems are likely unless at least one project member has some experience with the type of system being developed. Without this experience, many simple mistakes are likely to be made.

- **5.1 (3) Project planning**

- Effective management of a software project depends on thoroughly planning the progress of the project.

- Managers must anticipate problems that might arise and prepare tentative solutions to those problems.

- A plan, drawn up at the start of a project, should be used as the driver for the project. This initial plan should be the best

  possible plan given the available information.

- It evolves as the project progresses and better information becomes available.

- As well as a project plan, managers may also have to draw up other types of plans. These are briefly described in Figure 5.1

Figure 5.1 Types of plan

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project. See Chapter 24.
Validation plan	Describes the approach, resources and schedule used for system validation. See Chapter 19.
Configuration management plan	Describes the configuration management procedures and structures to be used. See Chapter 29.
Maintenance plan	Predicts the maintenance requirements of the system, maintenance costs and effort required. See Chapter 27.
Staff development plan	Describes how the skills and experience of the project team members will be developed. See Chapter 22.

- The pseudo-code shown in Figure 5.2 sets out a project planning process for software development.
- It shows that planning is an iterative process, which is only complete when the project itself is complete.
- As project information becomes available during the project, the plan should be regularly revised.
- The goals of the business are an important factor that must be considered when formulating the project plan.
- As these change, the project's goals also change so changes to the project plan are necessary

Figure 5.2 Project planning

```
Establish the project constraints
Make initial assessments of the project parameters
Define project milestones and deliverables
while project has not been completed or cancelled loop
    Draw up project schedule
    Initiate activities according to schedule
    Wait ( for a while )
    Review project progress
    Revise estimates of project parameters
    Update the project schedule
    Renegotiate project constraints and deliverables
    if ( problems arise ) then
        Initiate technical review and possible revision
    end if
end loop
```

- At the beginning of a planning process, you should assess the constraints (required delivery date, staff available, overall budget, etc.) affecting the. project.
- In conjunction with this, you should estimate project parameters such as its structure, size, and distribution of functions. You next define the progress milestones and deliverables.
- The process then enters a loop. You draw up an estimated schedule for the project and the activities defined in the schedule are started or given permission to continue.
- After some time (usually about two to three weeks), you should review progress and note discrepancies from the planned schedule.

- Because initial estimates of project parameters are tentative, you will always have to modify the original plan.
- As more information becomes available. you revise your original assumptions about the project and the project schedule.
- If the project is delayed, you may have to renegotiate the project constraints, and deliverables with the customer.
- If this renegotiation is unsuccessful and the schedule cannot be met, a project technical review may be held.

  The objective of this review is to find an alternative approach that falls within the project constraints and meets the schedule.

1) ***The Project Plan***

- The project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work.
- In some organizations, the project plan is a single document that includes the different types of plan (Figure 5.1).
- In other cases, the project plan is solely concerned with the development process.
- References to other plans are included but the plans themselves are separate.
- However, most plans should include the following sections:

1. ***Introduction*** This briefly describes the objectives of the project and sets out the constraints (e.g., budget, time, etc.) that affect the project management.

2. Project organization This describes the way in which the development team is organized, the people involved and their roles in the team.

3. Risk analysis This describes possible project risks, the likelihood of these risks arising and the risk reduction strategies that are proposed.

4. Hardware and software resource requirements This specifies the hardware and the support software required to carry out the development. If hardware has to be bought, estimates of the prices and the delivery schedule may be included.

5. Work breakdown This sets out the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity.

6. Project schedule This shows the dependencies between activities, the estimated time required to reach each milestone and the allocation of people to activities.

7. Monitoring and reporting mechanisms This defines the management reports that should be produced, when these should be produced and the project monitoring mechanisms used.

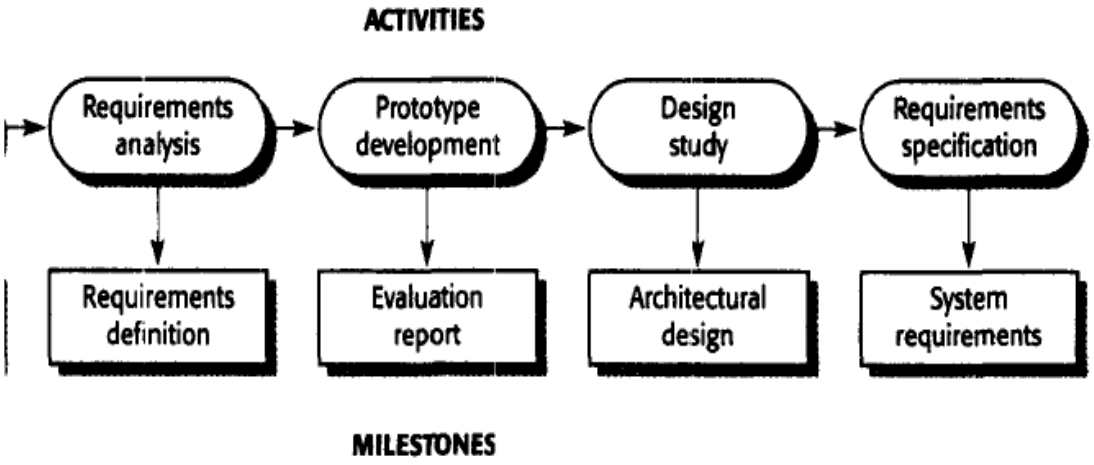
2) Milestones and deliverables

- When planning a project, you should establish a series of *milestones.* where a milestone is a recognizable end-point of a software process activity.
- At each milestone, there should be a formal output, such as a report, that can be presented to management. Milestone reports need not be large documents.
- They may simply be a short report of what has been completed. Milestones should represent the end of a distinct, logical stage in the project.
- Indefinite milestones such as 'Coding 80% complete' that can't be checked are useless for project management.

You can't check whether this state has been achieved because the amount of code that still has to be developed is uncertain.

- A deliverable is a project result that is delivered to the customer. It is usually delivered at the end of some major project phase such as ***specification or design.***
- *Deliverables are usually milestones, but milestones need not be deliverables.*

- Milestones may be internal project results that are used by the project manager to check project progress but which are not delivered to the customer.
- To establish milestones, the software process must be broken down into basic activities with associated outputs.
- For example***, Figure 5.3*** shows possible activities involved in requirements specification when prototyping is used to help validate requirements.
- The milestones in this case are the completion of the outputs for each activity.
- The project deliverables, which are delivered to the customer, are the requirements definition and the requirements specification.



5.1 (4) Project scheduling

- Project scheduling is one of the most difficult jobs for a project manager. Managers estimate the time and resources required to complete activities and organize them into a coherent sequence.
- Unless the project being scheduled is similar to a previous project, previous estimates are an uncertain basis for new project scheduling.
- Schedule estimation is further complicated by the fact that different projects may use different design methods and implementation languages.

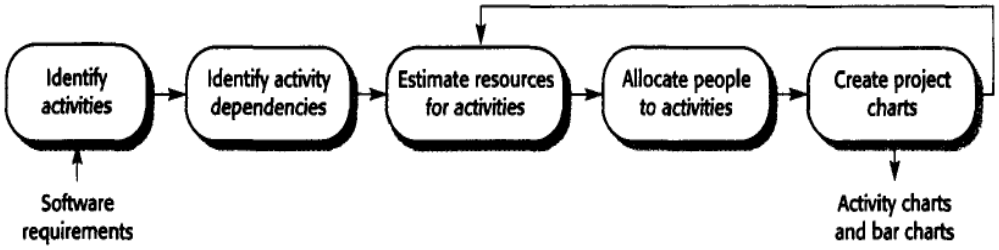
- Project scheduling (Figure 5.4) involves separating the total work involved in a project into separate activities and judging the time required to complete these activities. Usually, some of these activities are carried out in parallel.

Figure 5.4 The Project Scheduling Process

- Project activities should normally last at least a week. Finer subdivision means that

  a disproportionate amount of time must be spent on estimating and chart revision.

  - It is also useful to set a maximum amount of time for any activity of about 8 to 10 weeks. If it takes longer than this, it should be subdivided for project planning and scheduling



- When you are estimating schedules, you should not assume that every stage of the project will be problem free.
- People working on a project may fall ill or may leave, hardware may break down, and essential support software or hardware may be delivered late.
- If the project is new and technically advanced, certain parts of it may turn out to be more difficult and take longer than originally anticipated.
- As well as calendar time, you also have to estimate the resources needed to complete each task. The principal resource is the human effort required.
- A good rule of thumb is to estimate as if nothing will go wrong, then increase your estimate to cover anticipated problems.

- A further contingency factor to cover unanticipated problems may also be added to the estimate.
- This extra contingency factor depends on the type of project, the process parameters (deadline, standards, etc.) and the quality and experience of the software engineers working on the project.
- Project schedules are usually represented as a set of charts showing the work breakdown, activities dependencies and staff allocations.

1) ***Bar charts and activity networks***

- Bar charts and activity networks are graphical notations that are used to illustrate the project schedule.
- Bar charts show who is responsible for each activity and when the activity is scheduled to begin and end.
- Activity networks show the dependencies between the different activities making up a project.
- Bar charts and activity charts can be generated automatically from a database of project information using a project management tool.
- To illustrate how these charts are used, a hypothetical set of activities as shown in Figure 5.5.

Figure 5.5 Task durations and dependencies

Task	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

- This table shows activities, their duration, and activity interdependencies.
- From Figure 5.5, you can see that Activity T3 is dependent on Activity Ti. This means that T1 must be completed before T3 starts.
- For example, T1 might be the preparation of a component design and n, the implementation of that design.
- Before implementation starts, the design should be complete.
- Given the dependencies and estimated duration of activities, an activity chart that shows activity sequences may be generated

(Figure 5.6).

- This shows which activities can be carried out in parallel and which must be executed in sequence because of a dependency on an earlier activity.
- Activities are represented as rectangles; milestones and project deliverables are shown with rounded comers.
- Dates in this diagram show the start elate of the activity and are written in British style, where the day precedes the month.
- You should read the chart from left to right and from top to bottom.
- In the project management tool used to produce this chart, all activities must end in milestones.
- An activity may start when its preceding milestone (which may depend

on several activities) has been reached.

- Therefore, in the third column in Figure 5.5 shows the corresponding milestone (e.g., M5) that is reached when the tasks finish

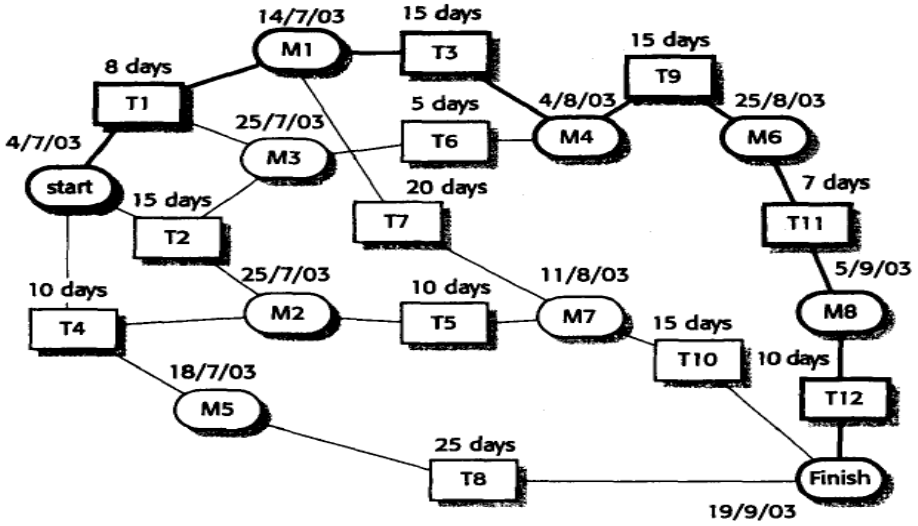
(see Figure 5.6).

- Before progress can be made from one milestone to another, all paths leading to it must be complete.

- For example, when activities n and T6 are finished, then activity

T9, shown in Figure 5.6, can start.

Figure 5.6 An activity network



- The minimum time required to finish the project can be estimated by considering the longest path in the activity graph (the critical path). In this case, it is 11 weeks of elapsed time or 55 working days.
- In Figure 5.6, the critical path is shown as a sequence of emboldened boxes. The critical path is the sequence of dependent activities that defines the time required to complete the project.
- The overall schedule of the project depends on the critical path. Any slippage in the completion in any critical activity causes project delays because the following activities cannot start until the delayed activity has been completed.
- For example, if T8 is delayed by two weeks, it will not affect the final completion date of the project because it does not lie on the critical path.
- Most project management tools compute the allowed delays, as shown in the project bar chart.

- Managers also use activity charts when allocating project work. They can provide insights into activity dependencies that are not intuitively obvious. It may be possible to modify the system design so that the critical path is shortened.

- Figure 5.7 is a complementary way of representing project schedule information. It is a bar chart showing a project calendar and the start and finish dates of activities.

- Sometimes these are called ***Gantt charts***, after their inventor. Reading from left to right, the bar chart clearly shows when activities start and end.

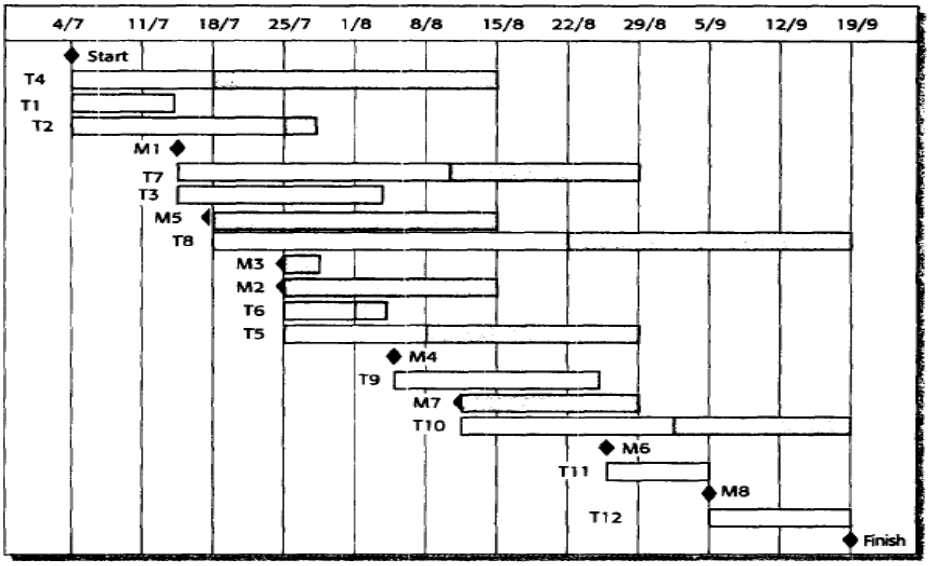
- Some of the activities shown in l:he bar chart in Figure 5.7 are followed by a shaded bar whose length is computed by the scheduling tool.

- This highlights the flexibility in the completion date of these activities. If an activity does not complete on time, the critical path will not be affected until the end of the period marked by the shaded bar.

- Activities that lie on the critical path have no margin of error

and can be identified because they have no associated shaded bar

Figure 5.7 Activity bar chart



- Large organizations usually employ a number of specialists who work on a project when needed.

- In ***Figure 5.8***, you can see that Mary and Jim are specialists who work on only a single task in the project. This can cause scheduling problems.
- If one project is delayed while a specialist is working on it, this may have a knock on effect on other projects.
- They may also be delayed because the specialist is not available.

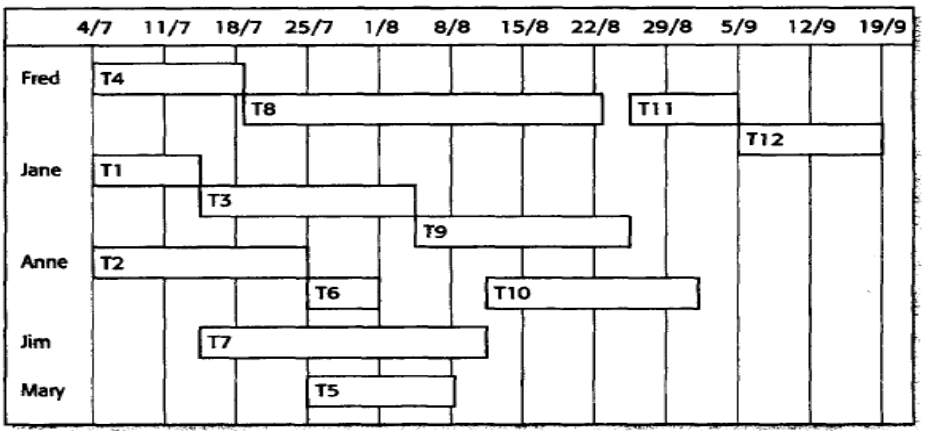


Figure 5.8 Staff allocation vs. time chart

5.1(5) Risk Management

- Risk management is increasingly seen as one of the main jobs of project managers.
- It involves anticipating risks that might affect the project schedule or the quality of the software being developed and taking action to avoid these risks.
- The results of the risk analysis should be documented in the project plan along with an analysis of the consequences of a risk occurring.
- Effective risk management makes it easier to cope with problems and to ensure that these do not lead to unacceptable budget or schedule slippage.
- Simplistically, you can think of a risk as something that you'd prefer not to have happen. Risks may threaten the project, the software that is being developed or the organization.
- There are, therefore, three related categories of risk:

1. ***Project risks*** are risks that affect the project schedule or resources. An example might be the loss of an experienced designer.

2. ***Product risks*** are risks that affect the quality or performance of the software being developed. An example might be the failure of a purchased component
to perform as expected.

3. **Business *risks*** are risks that affect the organization developing or procuring the software. For example, a competitor introducing Ii new product is a business risk.

- Of course, these risk types overlap. If an experienced programmer leaves a project, this can be a project risk because the delivery of the system may be delayed.
- It can also be a product risk because a replacement may not be as experienced and so may make programming errors.
- Finally, it can be a business risk because the programmer's experience is not available for bidding for future business.
- The risks that may affect a project depend on the project and the organizational environment where the software is being developed.
- However, many risks are universal- some of the most common risks are shown in Figure 5.9.

Figure 5.9 Possible software risks

Risk	Risk type	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organisational management with different priorities.
Hardware unavailability	Project	Hardware which is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool under-performance	Product	CASE tools which support the project do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

- The process of risk management is illustrated in Figure 5.10. It involves several stages:

1. ***Risk identification*** Possible project, product and business risks are identified.

2. ***Risk analysis*** The likelihood and consequences of these risks are assessed.

3. ***Risk planning*** Plans to address the risk either by avoiding it or minimizing its effects on the project are drawn up.

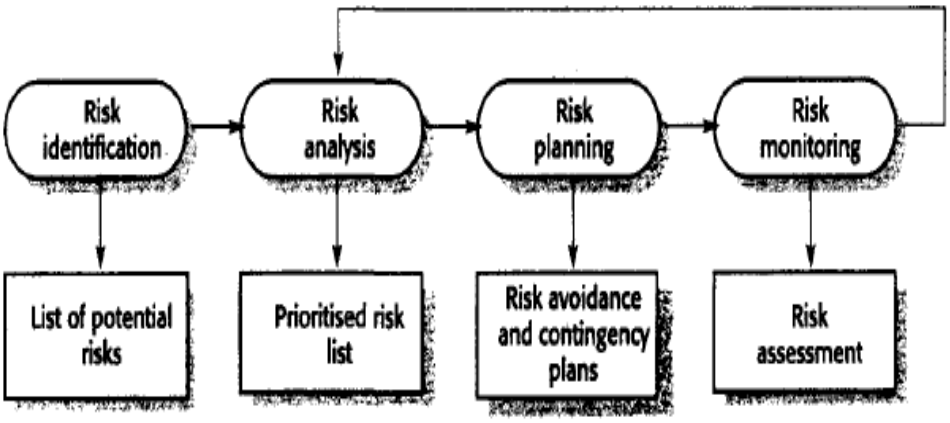
4. ***Risk monitoring*** The risk is constantly assessed and plans for risk mitigation are revised as more information about the risk becomes available.

- The risk management process, like all other project planning, is an iterative

process which continues throughout the project. Once an initial set of plans are drawn up, the situation is monitored. As more information about the risks becomes available, the risks have to be reanalyzed and new priorities established.

- The risk avoidance and contingency plans may be modified as new risk information emerges.

Figure 5.10 The risk management process



1) ***Risk Identification***

- Risk identification is the first stage of risk management. It is concerned with discovering possible risks to the project
- There are at least six types of risk that can arise:

1. ***Technology risks*** Risks that derive from the software or hardware technologies that are used to develop the system.

2. ***People risks*** Risks that are associated with the people in the development team.

3. ***Organizational risks*** Risks that derive from the organizational environment where the software is being developed.

4. ***Tools risks*** Risks that derive from the CASE tools and other support software used to develop the system.

5. ***Requirements risks*** Risks that derive from changes to the customer requirements and the process of managing the requirements change.

6. ***Estimation risks*** Risks that derive from the management estimates of the system characteristics and the resources required to build the system.

- Figure: 5.11 gives some examples of possible risks in each of these categories.

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. Software components which should be reused contain defects which limit their functionality.
People	It is impossible to recruit staff with the skills required. Key staff are ill and unavailable at critical times. Required training for staff is not available.
Organisational	The organisation is restructured so that different management are responsible for the project. Organisational financial problems force reductions in the project budget.
Tools	The code generated by CASE tools is inefficient. CASE tools cannot be integrated.
Requirements	Changes to requirements which require major design rework are proposed. Customers fail to understand the impact of requirements changes.
Estimation	The time required to develop the software is underestimated. The rate of defect repair is underestimated The size of the software is underestimated.

2) ***Risk Analysis***

- During the risk analysis process, you have to consider each identified risk and make a judgement about the probability and the seriousness of it.

- There is no easy way to do this-you must rely on your own judgement and experience, which is why experienced project managers are generally the best people to help with risk management.

- These risk estimates should not generally be precise numeric assessments but should be based around a number of bands:

• The probability of the risk might be assessed as very low (<10%), low (l0-25%), moderate (25-50%), high *(50-75%)* or very high (>75%).

- The effects of the risk might be assessed as catastrophic, serious, tolerable or insignificant.

Figure 5.12 illustrates this for the risks identified in Figure 5.11.

Figure 5.12 Risk analysis

Risk	Probability	Effects
Organisational financial problems force reductions in the project budget.	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project.	High	Catastrophic
Key staff are ill at critical times in the project.	Moderate	Serious
Software components which should be reused contain defects which limit their functionality.	Moderate	Serious
Changes to requirements which require major design rework are proposed.	Moderate	Serious
The organisation is restructured so that different management are responsible for the project.	High	Serious
The database used in the system cannot process as many transactions per second as expected.	Moderate	Serious
The time required to develop the software is underestimated.	High	Serious
CASE tools cannot be integrated	High	Tolerable
Customers fail to understand the impact of requirements changes.	Moderate	Tolerable
Required training for staff is not available	Moderate	Tolerable
The rate of defect repair is underestimated	Moderate	Tolerable
The size of the software is underestimated	High	Tolerable
The code generated by CASE tools is inefficient.	Moderate	Insignificant

- Once the risks have been analyzed and ranked, you should assess which are most significant. In general, catastrophic risks should always be considered, as should all serious risks that have more than a moderate probability of occurrence.
- Boehm (1988) recommends identify and monitoring the 'top 10' risks, but I think that this figure is rather arbitrary. The right number of risks to monitor must depend on the project. It might be 5 or it might be 15.
- However, the number of risks chosen for monitoring should be manageable. A very large number of risks would simply require too much information to be collected.
- From the risks identified in Figure 5.12, it is appropriate to consider all 8 risks that have catastrophic or serious consequences.

3) Risk Planning

- The risk planning process considers each of the key risks that have been identified and identifies strategies to manage the risk. Again. there is no simple process that can be followed to establish risk management plans. It relies on the judgement and experience of the project manager. Figure 5.13 shows possible strategies that have been identified for the key risks from Figure 5.12. These strategies fall into three categories:

1. ***Avoidance strategies*** Following these strategies means that the probability that the risk will arise will be reduced. An example of a risk avoidance strategy is the strategy for dealing with defective components shown in Figure 5.13.

2. ***Minimization strategies*** Following these strategies means that the impact of the risk will be reduced. An example of a risk minimization strategy is that for staff illness shown in Figure 5.13.

3. ***Contingency plans*** Following these strategies means that you are prepared for the worst and have a strategy in place to deal with it An example of a contingency strategy is the strategy for organizational financial problems in Figure 5.13

Figure 5.13 Risk management strategies

Risk	Strategy
Organisational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Recruitment problems	Alert customer of potential difficulties and the possibility of delays, investigate buying-in components.
Staff illness	Reorganise team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact, maximise information hiding in the design.
Organisational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	I vestigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components, investigate the use of a program generator.

4) Risk Monitoring

- Risk monitoring involves regularly assessing each of the identified risks to decide whether or not that risk is becoming more or less probable and whether the effects of the risk have changed.
- Of course, this cannot usually be observed directly, so you have to look at other factors that give you clues about the risk probability and its effects.
- These factors are obviously dependent on the types of risk. Figure 5.14 gives some examples of factors that may be helpful in assessing these risk types.
- Risk monitoring should be a continuous process, and, at every management progress review, you should consider and discuss each of the key risks separately.

Figure 5.14 Risk factors

Risk type	Potential indicators
Technology	Late delivery of hardware or support software, many reported technology problems
People	Poor staff morale, poor relationships amongst team members, job availability
Organisational	Organisational gossip, lack of action by senior management
Tools	Reluctance by team members to use tools, complaints about CASE tools, demands for higher-powered workstations
Requirements	Many requirements change requests, customer complaints
Estimation	Failure to meet agreed schedule, failure to clear reported defects

5.2 (1) Software Cost Estimation

- Estimation involves answering the following questions:

1. How much effort is required to complete each activity?
2. How much calendar time is needed to complete each activity?
3. What is the total cost of each activity?

- Project cost estimation and project scheduling are normally carried out together. The costs of development are primarily the costs of the effort involved, so the effort computation is used in both the cost and the schedule estimate.

- There are three parameters involved in computing the total cost of a software development project:

1) Hardware and software costs including maintenance
2) Travel and training costs
3) Effort costs (the costs of paying software engineers).

- For most projects, the dominant cost is the effort cost. Effort costs are not just the salaries of the software engineers who are involved in the project.

- Organizations compute effort costs in terms of overhead costs where they take the total cost of running the organization and divide this by the number of productive staff. Therefore, the following costs are all part of the total effort cost:

1. Costs of providing, heating and lighting office space
2. Costs of support staff such as accountants, administrators, system managers, cleaners and technicians
3. Costs of networking and communications.
4. Costs of central facilities such as a library or recreational facilities
5. Costs of Social Security and employee benefits such as pensions and health insurance.

- Once a project is underway, project managers should regularly update their cost and schedule estimates.

- This helps with the planning process and the effective use of resources. If actual expenditure is significantly greater than the estimates, then the project manager must take some action.

- Software costing should be carried out objectively with the aim of accurately predicting the cost of developing the software.

- Software pricing must take into account broader organizational, economic, political and business considerations, such as those shown in Figure 26.1.

- Therefore, there may not be a simple relationship between the price to the customer for the software and the development costs.

Figure 26.1 Factors affecting software pricing

Factor	Description
Market opportunity	A development organisation may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organisation the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Cost estimate uncertainty	If an organisation is unsure of its cost estimate, it may increase its price by some contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organisation may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business.

5.2 (2) Software Productivity

- You can measure productivity in a manufacturing system by counting the number of units that are produced and dividing this by the number of person-hours required to produce them.
- However, for any software problem, there are many different solutions, each of which has different attributes.
- One solution may execute more efficiently while another may be more readable and easier to maintain.
- When solutions with different attributes are produced, comparing their production rates is not really meaningful.
- Nevertheless, as a project manager, you may be faced with the problem of estimating the productivity of software engineers.

- You may need these productivity estimates to help define the project cost or schedule, to inform investment decisions or to assess, whether process or technology improvements are effective.
- Productivity estimates are usually based on measuring attributes of the software and dividing this by the total effort required for development. There are two types of metric that have been used:

1. ***Size-related metrics*** These are related to the size of some output from an activity. The most commonly used size-related metric is lines of delivered source code.

- Other metrics that may be used are the number of delivered object code instructions or the number of pages of system documentation.

2. ***Function-related metrics*** These are related to the overall functionality of the delivered software.

- Productivity is expressed in terms of the amount of useful functionality produced in some given time. Function points and object points are the best-known metrics of this type.
- Lines of source code per programmer-month (LOC/pm) is a widely used software productivity metric.
- You can compute LOC/pm by counting the total number of lines of source code that are delivered, then divide the count by the total time in programmer-months required to complete the project.
- This time therefore includes the time required for all other activities (requirements, design, coding, testing and documentation) involved in software development.
- This approach was first developed when most programming was in FORTRAN, assembly language or COBOL.
- Then, programs were typed on cards, with one statement on each card. The number of lines of code was easy to count: It corresponded to the number of cards in the program deck.
- However, programs in languages such as Java or C++ consist of declarations, executable statements and commentary.

- They may include macro instructions that expand to several lines of code.
- For example, consider an embedded real-time system that might be coded in 5,000 lines of assembly code or 1,500 lines of C. The development time for the various phases is shown in Figure 26.2.
- The assembler programmer has a productivity of 714 lines/month and the high-level language programmer less than half of this 300 lines/month.
- Productivity is expressed as the number of function points that are implemented per person-month. A function point is not a single characteristic but is computed by combining several different measurements or estimates.
- You compute the total number of function points in a program by measuring or estimating the following program features:
- ***external inputs and outputs;***
- ***user interactions;***
- ***external interfaces;***
- ***file:; used by the system***.

Figure 26.2 System development times

	Analysis	Design	Coding	Testing	Documentation
Assembly code	3 weeks	5 weeks	8 weeks	10 weeks	2 weeks
High-level language	3 weeks	5 weeks	4 weeks	6 weeks	2 weeks

	Size	Effort	Productivity
Assembly code	5000 lines	28 weeks	714 lines/month
High-level language	1500 lines	20 weeks	300 lines/month

- Obviously, some inputs and outputs, interactions. and so on are more complex than others and take longer to implement.

- The function-point metric takes this into account by multiplying the initial function-point estimate by a complexity-weighting factor.
- You should assess each of these features for complexity and then assign the weighting factor that vanes from 3 (for simple external inputs) to 15 for complex internal files .
- Either the weighting values proposed by Albrecht or values based on local experience may be used.
- You can then compute the so-called unadjusted function-point count (UFC) by multiplying each initial count by the estimated weight and summing all values.

UFC = Σ(number of elements of given type) x (weight)

- You then modify this unadjusted function-point count by additional complexity factors that are related to the complexity of the system as a whole.
- This takes into account the degree of distributed processing, the amount of reuse, the performance, and so or.
- The unadjusted function-point count is multiplied by these project complexity factors to produce a final function-point count for the overall system.
- Object points (Banker, et al., 1994) are an alternative to function points. They can be used with languages such as database programming languages or scripting languages.
- Object points are not object classes that may be produced when an object-oriented approach is taken to software development. Rather, the number of object points in a program is a weighted estimate of:

1. ***The number of separate screens that are displayed*** Simple screens count as 1 object point, moderately complex screens count as 2, and very complex screens count as 3 object points.
2. ***The number of reports that are produced*** For simple reports, count 2 object points, for moderately complex reports, count 5, and for reports that are likely to be difficult to produce, count 8 object points.

3. ***The number of modules in imperative programming languages such as Java or C++ that must be developed to supplement the database programming code*** Each of these modules counts as 10 object points.

- Object points are used in the COCOMO II estimation model (where they are called application points) The advantage of object points over function points is that they are easier to estimate from a high-level software specification.
- Object points are only concerned with screens, reports and modules in conventional programming languages. They are not concerned with implementation details, and the complexity factor estimation is much simpler.
- If function points or object points are used, they can be estimated at an early stage in the development process before decisions that affect the program size have been made.
- Function-point and object-point counts can be used in conjunction with lines of code-estimation models. The final code size is calculated from the number of function points.
- Using historical data analysis, the average number of lines of code, AVC, in a particular language required to implement a function point can be estimated.
- Values of AVC vary from 200 to 300 LOCIFP in assembly language to 2 to 40 LOCIFP for a database programming language such as SQL.
- The estimated code size for a new application is then computed as follows:

Code size = AVC x Number of function points

- The programming productivity of individuals working in an organization is affected by a number of factors. Some of the most important of these are summarized in

Figure 26.3. However, individual differences in ability are usually more significant than any of these factors.

Figure 26.3 Factors affecting software engineering productivity

Factor	Description
Application domain experience	Knowledge of the application domain is essential for effective software development. Engineers who already understand a domain are likely to be the most productive.
Process quality	The development process used can have a significant effect on productivity. This is covered in Chapter 28.
Project size	The larger a project, the more time required for team communications. Less time is available for development so individual productivity is reduced.
Technology support	Good support technology such as CASE tools and configuration management systems can improve productivity.
Working environment	As I discussed in Chapter 25, a quiet working environment with private work areas contributes to improved productivity.

- Software development productivity varies dramatically across application domains and organizations.
- For large, complex, embedded systems, productivity has been estimated to be as low as 30 LOC/pm.
- For straightforward, well-understood application systems, written in a language such as Java, it may be as high as 900 LOC/pm.
- The problem with measures that rely on the amount produced in a given time period is that they take no account of quality characteristics such as reliability and maintainability.
- These measures also do not take into account the possibility of reusing the software produced, using code generators and other tools that help create the software.

5.2 (3) Estimation Techniques

- There is no simple way to make an accurate estimate of the effort required to develop a software system.
- You may have to make initial estimates on the basis of a high-level user requirements definition.

- Project cost estimates are often self-fulfilling.
- The estimate is used to define the project budget, and the product is adjusted so that the budget figure is realized.
- Nevertheless, organizations need to make software effort and cost estimates. To do so, one or more of the techniques described in Figure 26.4 may be used

Figure 26.4 Cost-estimation techniques

Technique	Description
Algorithmic cost modelling	A model is developed using historical cost information that relates some software metric (usually its size) to the project cost. An estimate is made of that metric and the model predicts the effort required.
Expert judgement	Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. These estimates are compared and discussed. The estimation process iterates until an agreed estimate is reached.
Estimation by analogy	This technique is applicable when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects. Myers (Myers, 1989) gives a very clear description of this approach.
Parkinson's Law	Parkinson's Law states that work expands to fill the time available. The cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months.
Pricing to win	The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customer's budget and not on the software functionality.

- However, there may be important differences between past and future projects. Many new development methods and techniques have been introduced in the last 10 years.
- Some examples of the changes that may affect estimates based on experience include:

1. Distributed object systems rather than mainframe-based systems
2. Use of web services
3. Use of ERP or database-centered systems
4. Use of off-the-shelf software rather than original system development
5. Development for and with reuse rather than new development of all parts of a system

6. Development using scripting languages such as TCL or Perl (Ousterhout, 1998)
7. The use of CASE tools and program generators rather than unsupported software development

- If project managers have not worked with these techniques, their previous experience may not help them estimate software project costs.
- You can tackle the approaches to cost estimation shown in Figure 26.4 using either a top-down or a bottom-up approach. A top-down approach starts at the system level.
- The bottom-up approach, by contrast, starts at the component level. The system is decomposed into components, and you estimate the effort required to develop each of these components.
- The disadvantages of the top-down approach are the advantages of the bottom-up approach and vice versa. Top-down estimation can underestimate the costs of solving difficult technical problems associated with specific components such as interfaces to nonstandard hardware.
- By contrast, bottom-up estimation produces such a justification and considers each component.
- However, this approach is more likely to underestimate the costs of system activities such as integration.
- Bottom-up estimation is also more expensive. Each estimation technique has its own strengths and weaknesses.
- Each uses different information about the project and the development team, so if you use a single model and this information is not accurate, your final estimate will be wrong.
- For large projects, therefore, you should use several cost estimation techniques and compare their results.
- These estimation techniques are applicable where a requirements document for the system has been produced. This should define all users and system requirements.
- However, in many cases, the costs of many projects must be estimated using only incomplete user requirements for the system.

- This means that the estimators have very little information with which to work.
- Requirements analysis and specification is expensive, and the managers in a company may need an initial cost estimate for the system before they can have a budget approved to develop more detailed requirements or a system prototype.
- Under these circumstances, 'pricing to win" is a commonly used strategy. The notion of pricing to win may seem unethical and un-business like.
- However, it does have some advantages. A project cost is agreed on the basis of an outline proposal. Negotiations then take place between client and customer to establish the detailed project specification.

5.2 (4) Algorithmic cost modeling

- Algorithmic cost modeling uses a mathematical formula to predict project costs based on estimates of the project size, the number of software engineers, and other process and product factors.
- An algorithmic cost model can be built by analyzing the costs and attributes of completed projects and finding the closest fit formula to actual experience.
- Algorithmic cost models are primarily used to make estimates of software development costs, but Boehm (2000) discusses a range of other uses for algorithmic cost estimates, including estimates for investors in software companies, estimates of alternative strategies to help assess risks, and estimates to inform decisions about reuse, redevelopment or outsourcing.
- In its most general form, an algorithmic cost estimate for software cost can be expressed as:

  $Effort = A \times Size^{B} \times M$

- A is a constant factor that depends on local organizational practices and the type of software that is developed.
- Size may be either an assessment of the code size of the software or a functionality estimate expressed in function or object points.

- The value of exponent B usually lies between 1 and 1.5.
- M is a multiplier made by combining process, product and development attributes, such as the dependability requirements for the software and the experience of the development team.
- Most algorithmic estimation models have an exponential component (B in the above equation) that is associated with the size estimate. This reflects the fact that costs do not normally increase linearly with project size.
- Unfortunately, all algorithmic models suffer from the same fundamental difficulties

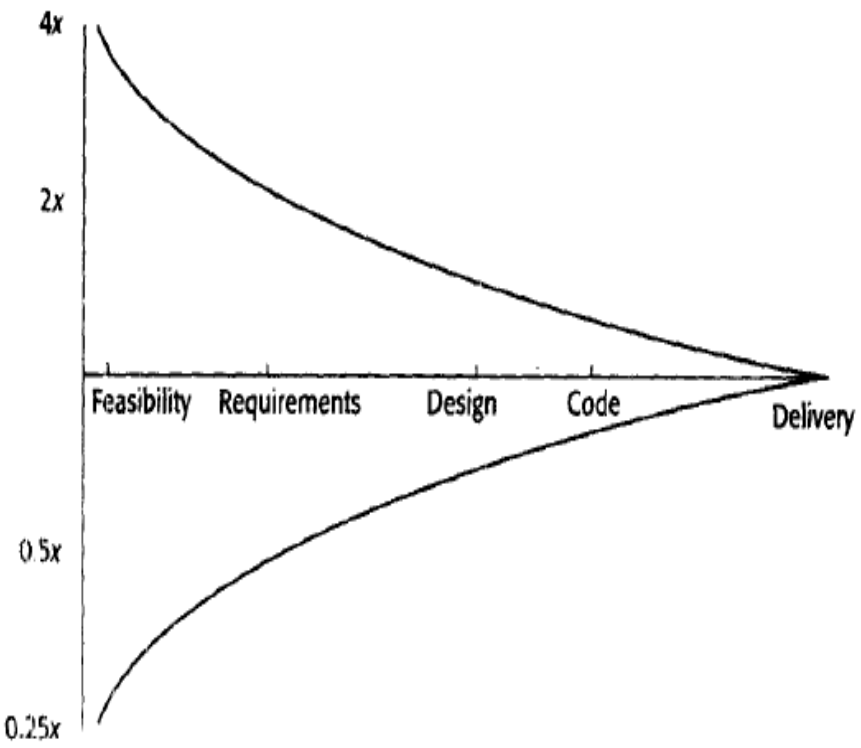
1. ***It is often difficult to estimate Size at an early stage in a project when only a specfication is available.*** Function-point and object-point estimates are easier to produce than estimates of code size but are often still inaccurate.

2. ***The estimates of the factors contributing to*** **B** ***and*** **M** ***are subjective***. Estimates vary from one person to another, depending on their background and experience with the type of system that is being developed.

- The number of lines of source code in the delivered system is the basic metric used in many algorithmic cost models.
- Size estimation may involve estimation by analogy with other projects, estimation by converting function or object points to code size, estimation by ranking the sizes of system components and using a known reference component to estimate the component size, or it may simply be a question of engineering judgement
- Accurate code size estimation is difficult at an early stage in a project because the code size is affected by design decisions that have not yet been made.
- For example, an application that requires complex data management may either use a commercial database or implement its own data-management system.
- If a commercial database is used, the code size will be smaller but additional effort may be needed to overcome the performance limitations of the commercial product.

- The programming language used for system development also affects the number of lines of code to be developed.
- A language such as Java might mean that more lines of code are necessary than if C (say) were used. However, this extra code allows more compile-time checking so validation costs are likely to be reduced.
- If you use an algorithmic cost estimation model, you should develop a range of estimates (worst, expected and best) rather than a single estimate and apply the costing formula to all of them.
- The accuracy of the estimates produced by an algorithmic model depends on the system information that is available.
- As the software process proceeds, more information becomes available so estimates become more and more accurate.
- If the initial estimate of effort required is *x* months of effort, this range may be from *O.25x* to *4x* when the system is first proposed.
- This narrows during the development process, as shown in Figure 26.5. This figure, adapted from Boehm's paper.

Figure 26.5 Estimate uncertainty



1) The COCOMO model

- A number of algorithmic models have been proposed as the basis for estimating the effort, schedule and costs of a software project. These are conceptually similar but use different parameter values.
- The model that is discussed here is the COCOMO model.
- The COCOMO model is an empirical model that was derived by collecting data from a large number of software projects. These data were analyzed to discover formulae that were the best fit to the observations. These formulae link the size of the system and product, project and team factors to the effort to develop the system.
-
- Use the COCOMO model for several reasons:

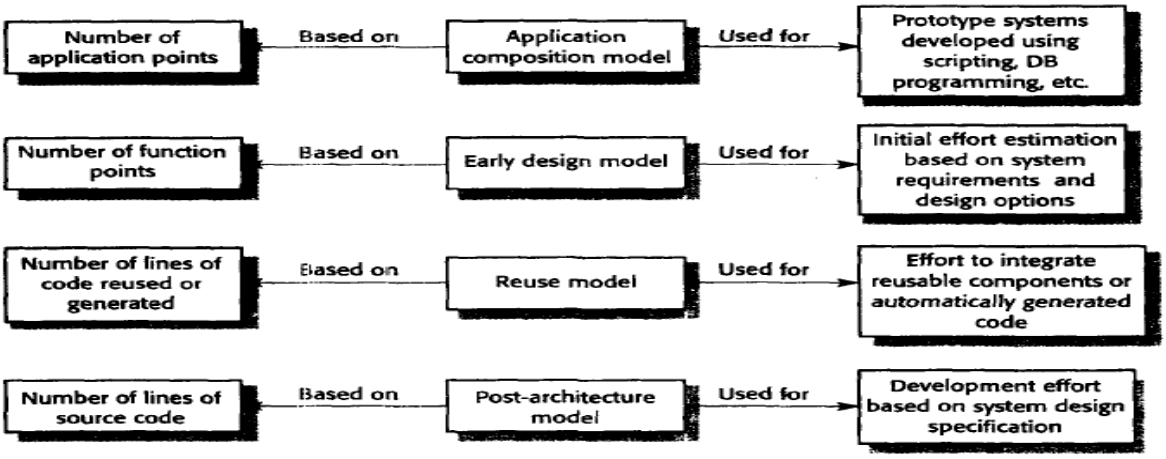
1. It is well documented, available in the public domain and supported by public domain and commercial tools.
2. It has been widely used and evaluated in a range of organizations.

3. It has a long pedigree from its first instantiation in 1981 (Boehm, 1981), through a refinement tailored to Ada software development (Boehm and Royce, 1989). to its most recent version, COCOMO II, published in 2000 (Boehm,et al. 2000).

- The COCOMO models are comprehensive. with a large number of parameters that can each take a range of values. They are so complex
- The first version of the COCOMO model (COCOMO 81) was a three-level model where the: levels corresponded to the detail of the analysis of the cost estimate.
- The first level (basic) provided an initial rough estimate; the second level modified this using a number of project and process multipliers; and the most detailed level produced estimates for different phases of the project.
- Figure 26.6 shows the basic COCOMO formula for different types of projects. The multiplier M reflects product, project and team characteristics.
- COCOMO 81 assumed that the software would be developed according to a waterfall process using standard imperative programming languages such as C or FORTRAN.
- Prototyping and incremental development are commonly used process models. Software is now often developed by assembling reusable components with off-the-shelf systems and 'gluing' them together with scripting language.

Project complexity	Formula	Description
Simple	$PM = 2.4\ (KDSI)^{1.05} \times M$	Well-understood applications developed by small teams
Moderate	$PM = 3.0\ (KDSI)^{1.12} \times M$	More complex projects where team members may have limited experience of related systems
Embedded	$PM = 3.6\ (KDSI)^{1.20} \times M$	Complex projects where the software is part of a strongly coupled complex of hardware, software, regulations and operational procedures

- Existing software is re-engineered to create new software. CASE tool support for most software process activities is now available.
- To take these changes into account, the COCOMO II model recognizes different approaches to software development such as prototyping, development by component composition and use of database programming.
- COCOMO II supports a spiral model of development and embeds several sub-models that produce increasingly detailed estimates.
- These can be used in successive rounds of the development spiral. Figure 26.7 shows COCOMO II sub-models and where they are used.
- ***Figure: 26.7 The COCOMO II Models***



- The sub-models that are part of the COCOMO II model are:

1. ***An application-composition model***

- This assumes that systems are created from reusable components, scripting or database programming. It is 'designed to make estimates of prototype development.
- Software size estimates are based on application points, and a simple size/productivity formula is used to estimate the effort required. Application points are the same as object points but the name was changed to avoid confusion with objects in object-oriented development.

*2. **An early design model***

- This model is used during early stages of the system design after the requirements have been established.

Estimates are based on function points, which are then converted to number of lines of source code. The formula follows the standard form discussed above with a simplified set of seven multipliers.

*3. **A reuse model***

- This model is used to compute the effort required to integrate reusable components and/or program code that is automatically generated by design or program translation tools. It is usually used in conjunction with the post-architecture model.

*4. **A post-architecture model***

- Once the system architecture has been designed, a more accurate estimate of the software size can be made. Again this model uses the standard formula for cost estimation discussed above.

- However, it includes a more extensive set of 17 multipliers reflecting personnel capability and product and project characteristics.

1) *The application-composition model*

- The application-composition model was introduced into COCOMO II to support the estimation of effort required for prototyping projects and for projects where the software is developed by composing existing components.

- It is based on an estimate of weighted application points (object points) divided by a standard estimate of application-point productivity.

- Figure 26.8 shows the levels of object-point productivity suggested by the model developers.

Figure 26.8 Object-point productivity

Developer's experience and capability	Very low	L w	Nominal	High	Very high
CASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NOP/month)	4	7	13	25	50

- Application composition usually involves significant software reuse, and some of the total number of application points in the system may be implemented with reusable components .
- Consequently, you have to adjust the estimate based on the total number of application points to take into account the percentage of reuse expected.
- Therefore, the final formula for effort computation for system prototypes is:

PM = (NAP x (1 - %reuse/100) / PROD

- PM is the effort estimate in person-months. NAP is the total number of application points in the delivered system.
- %reuse is an estimate of the amount of reused code in the development.
- PROD is the object-point productivity

2) The early design model

- This model is used once user requirements have been agreed and initial stages of the system design process are underway. However, you don't need a detailed architectural design to make these initial estimates.
- Your goal at this stage should be to make an approximate estimate without undue effort. Early design estimates are most useful for option exploration where you need to compare different ways of implementing the user requirements.
- The estimates produced at this stage are based on the standard formula for algorithmic models, namely:

Effort = A x Size^{B}x M

- Based on his own large data set, Boehm proposes that the coefficient A should be 2.94.
- The size of the system is expressed in KSLOC, which is the number of thousands of lines of source code.
- You calculate KSLOC by estimating the number of function points in the software.
- You then use standard tables that relate software size to function points for different programming languages to compute an initial estimate of the system size in KSLOC.
- The exponent B reflects the increased effort required as the size of the project increases.
- This is not fixed for different types of systems, as in COCOMO 81, but can vary from 1.1 to 1.24 depending on the novelty of the project, the development flexibility, the risk resolution processes used, the cohesion of the development team

and the process maturity level.

- The multiplier M in COCOMO II is based on a simplified set of seven project and process characteristics that influence the estimate. These can increase or decrease the effort required.
- These characteristics used in the early design model are product reliability and complexity (RCPX), reuse required (RUSE), platform difficulty (PDIF), personnel capability (PERS), personnel experience (PREX), schedule (SCED) and support facilities(FCIL).
- You estimate values for these attributes using a six-point scale where 1 corresponds to very low values for these multipliers and 6 corresponds to very high values.
- This results in an effort computation as follows:

$$PM = 2.94 \times Size^{B} \times M$$

- where,

$$M = PERS \times RCPX \times RUSE \times PDIF \times PREX \times FCIL \times SCED$$

3) The reuse model

- Software reuse is now common, and most large systems include a significant percentage of code that is reused from previous developments.

- The reuse model is used to estimate the effort required to integrate reusable or generated code.
- COCOMO II considers reused code to be of two types. Black-box code is code that can be reused without understanding the code or making changes to it.
- The development effort for black-box code is taken to be zero. Code that has to be adapted to integrate it with new code or other reused components is called white-box code.
- Some development effort is required to reuse this because it has to be understood and modified before it can work correctly in the system.
- The COCOMO II reuse model includes a separate model to
- estimate the costs associated with this generated code.
- For cede that IS, automatically generated, the model estimates the number of person months required to integrate this code. The formula for effort estimation is:
- PM $_{Auto=}$(ASLOC x AT/100) / ATPROD / / Estimate for generated code
- AT is the percentage of adapted code that is automatically generated and ATPROD is the productivity of engineers m integrating such code.
- Boehm (2000) have measured ATPROD to be about 2,400 source statements per month. Therefore. if there is a total of 20,000 lines of white-box reused code in a system and 30% of this is automatically generated, then the effort required to integrate this generated code is:
- (20,000 x 30/100) / 2400 =2.5 person months //Generated code example
- The other component of the reuse model is used when a system includes some new code and some reused white-box components that have to be integrated.
- In this case, the reuse model does not compute the effort -directly.
- Rather, based on the number of lines of code that are reused, it calculates a figure that represents the equivalent number of lines of new code.

- Therefore, if 30,000 lines of code are to be reused, the new equivalent size estimate might be 6,000.
- Essentially, reusing 30,000 lines of code is taken to be equivalent to writing 6,000 lines of new code.
- This calculated figure is added to the number of lines of new code to be developed in the COCOMO II post-architecture model.
- The estimates in this reuse model are:
- ***ASLOC***-the number of lines of code in the components that have to be adapted;
- ***ESLOC***-the equivalent number of lines of new source code
- The formula used to compute ESLOC takes into account the effort required for software understanding, for making changes to the reused code and for making changes to the system to integrate that code.
- It also takes into account the amount of code that is automatically generated where the development effort is calculated, as explained earlier in this section.
- The following formula is used to calculate the number of equivalent lines of source code:

  ESLOC=ASLOC x (1-AT/100) x AAM
- ***ASLOC*** is reduced according to the percentage of automatically generated code.
- ***AAM*** is the Adaptation Adjustment Multiplier, which takes into account the effort required to reuse code
- ***Simplistically, AAM is the sum of three components***:

1. ***An adaptation component*** (referred to as AAF) that represents the costs of making changes to the reused code. This includes components that take into account design, code and integration changes.
2. ***An understanding component*** (referred to as SU) that represents the costs of understanding the code to be reused and the familiarity of the engineer with the code. SU ranges from 50 for complex unstructured code to 10 for well-written,

object-oriented code.

3. ***An assessment factor*** (referred to as AA) that represents the costs of reuse decision making. That is, some analysis is always required to decide whether code can be reused, and this is included in the cost as AA. AA varies from 0 to 8depending
on the amount of analysis effort required.

The reuse model is a nonlinear model.

4) The post-architecture level

- The post-architecture model is the most detailed of the COCOMO II models. It is used one e an initial architectural design for the system is available so the sub-system structure is known.
- The estimates produced at the post-architecture level are based on the same basic formula ***(PM == A X Size^{B} X M)*** used in the early design estimates.
- However, the size estimate for the software should be more accurate by this stage in the estimation process.
- In addition, a much more extensive set of product, process and organizational attributes (17 rather than 7) are used to refine the initial effort computation.
- It is possible to use more attributes at this stage because you have more information about the software to be developed and the development process.
- The estimate of the code size in the post-architecture model is computed using three components:
- I. An estimate of the total number of lines of new code to be developed
- 2. An estimate of the equivalent number of source lines of code (ESLOC) calculated using the reuse model
- 3. An estimate of the number of lines of code that have to be modified because of changes to the requirements.

- These three estimates are added to give the total code size in KSLOC that you use in the effort computation formula. The final component in the estimate-the number of lines of modified code-reflects the fact that software requirements always change.
- The exponent term (B) in the effort computation formula had three possible values in CDCOMO I. These were related to the levels of project complexity.
- As projects become more complex, the effects of increasing system size become more significant.
- However, good organizational practices and procedures can control this 'diseconomy of scale'. This is recognized in COCOMO II, where the range of values for the exponent B is continuous rather than discrete.
- The exponent is based on five scale factors, as shown in Figure 26.9. These factors are rated on a six-point scale from Very low to Extra high (5 to 0).
- You should then add the ratings, divide them by 1100 and add the result to 1.01 to get the exponent that should be used.

Figure 26.9 Scale factors used in the COCOMO II exponent computation

Scale factor	Explanation
Precedentedness	Reflects the previous experience of the organisation with this type of project. Very low means no previous experience; Extra high means that the organisation is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; Extra high means that the client sets only general goals.
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis; Extra high means a complete and thorough risk analysis.
Team cohesion	Reflects how well the development team know each other and work together. Very low means very difficult interactions; Extra high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organisation. The computation of this value depends on the CMM Maturity Questionnaire, but an estimate can be achieved by subtracting the CMM process maturity level from 5.

- The organization has recently put a process improvement programe in place and has been rated as a Level 2 organization according to the CMM model.
- Possible values for the ratings used in exponent calculation are:
- *Precedentedness* This is a new project for the organization-rated Low (4)

- *Development flexibility* No client involvement-rated Very high (1)
- *Architecture/risk resolution* No risk analysis carried out-rated Very low (5)
- *Team cohesion* New team so no information-rated Nominal (3)
- *Process maturity* Some process control in place-rated Nominal (3)
- The sum of these values is 16, so you calculate the exponent by adding 0.16 to 1.01, getting a value of 1.17.
- The attributes ***(Figure 26.10)*** that are used to adjust the initial estimates and create multiplier M in the post-architecture model fall into four classes:

1. Product attributes are concerned with required characteristics of the software product being developed.
2. Computer attributes are constraints imposed on the software by the hardware platform.
3. Personnel attributes are multipliers that take the experience and capabilities of the people working on the project into account.
4. Project attributes are concerned with the particular characteristics of the software development project.

Attribute	Type	Description
RELY	Product	Required system reliability
CPLX	Product	Complexity of system modules
DOCU	Product	Extent of documentation required
DATA	Product	Size of database used
RUSE	Product	Required percentage of reusable components
TIME	Computer	Execution time constraint
PVOL	Computer	Volatility of development platform
STOR	Computer	Memory constraints
ACAP	Personnel	Capability of project analysts
PCON	Personnel	Personnel continuity
PCAP	Personnel	Programmer capability
PEXP	Personnel	Programmer experience in project domain
AEXP	Personnel	Analyst experience in project domain
LTEX	Personnel	Language and tool experience
TOOL	Project	Use of software tools
SCED	Project	Development schedule compression
SITE	Project	Extent of multisite working and quality of inter-site communications

- Figure 26.11 shows how these cost drivers influence effort estimates.
- In Figure 26.11, I have assigned maximum and minimum values to the key cost drivers to show how they influence the effort estimate.
- The values taken are those from the COCOMO II reference manual (Boehm, 1997).
- You can see that high values for the cost drivers lead to an effort estimate that is more than three times the initial estimate, whereas low values reduce the estimate to about one third of the original.
- This formulae proposed by the developers of the COCOMO II model reflects their experience and data, but it is an extremely complex model to understand and use.
- There are many attributes and considerable scope for uncertainty in estimating their values.
- Very large organizations may have the resources to employ a cost-modeling expert to adapt and use the COCOMO II models.

Figure 26.11 The effect of cost drivers on effort estimates

Exponent value	1.17
System si e (including factors for reuse and requirements volatility)	128,000 DSI
Initial COCOMO estimate without cost drivers	**730 person-months**
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	**2306 person-months**
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	**295 person-months**

2) Algorithmic cost models in project planning

- One of the most valuable uses of algorithmic cost modeling is to compare different ways of investing money to reduce project costs.
- The algorithmic code model helps you assess the risks of each option. Applying the cost model reveals the financial exposure that is associated with different management decisions.
- Consider an embedded system to control an experiment that is to be launched into space. Space-borne experiments have to be very reliable and are subject to stringent weight limits.
- The number of chips on a circuit board may have to be minimized.
- In terms of the COCOMO model, the multipliers based on computer constraints and reliability are greater than 1.
- There are three components to be taken into account in costing this project:

1. The cost of the target hardware to execute the system
2. The cost of the platform (computer plus software) to develop the system
3. The cost of the effort required to develop the software.

- Figure 26.13 shows some possible options for this project. These include spending more on target hardware to reduce software costs or investing in better development tools.
- Additional hardware costs may be acceptable because the system is a specialized system that does not have to be mass-produced.
- Figure 26.13 shows the hardware, software and total costs for the options A-F shown in Figure 26.12.
- Applying the COCOMO II model without cost drivers predicts an effort of 45 person-months to develop an embedded software system for this application. The average cost for one person-month of effort is $15,000.
- The relevant multipliers are based on storage and execution time constraints (TIME and STOR), the availability of tool support (cross-compilers, etc.) for the development system (TOOL), and development team s experience platform experience (LTEX).
- In all options, the reliability multiplier (RELY) is 1.39, indicating that significant extra effort is needed to develop a reliable system.
- The software cost (SC) is computed as follows:

SC=Effort estimate X RELY X TIME X STOR X TOOL X EXP X$15,000

- Option A represents the cost of building the system with existing support and staff. It represents a baseline for comparison.
- All other options involve either more hardware expenditure or the recruitment (with associated costs and risks) of new staff.
- Option B shows that upgrading hardware does not necessarily reduce costs.
- The staff lack experience with the new hardware so the increase in the experience multiplier negates the reduction in the STOR and TIME multipliers.
- It is actually more cost-effective to upgrade memory rather than the whole computer configuration.

Figure 26.12
Management options

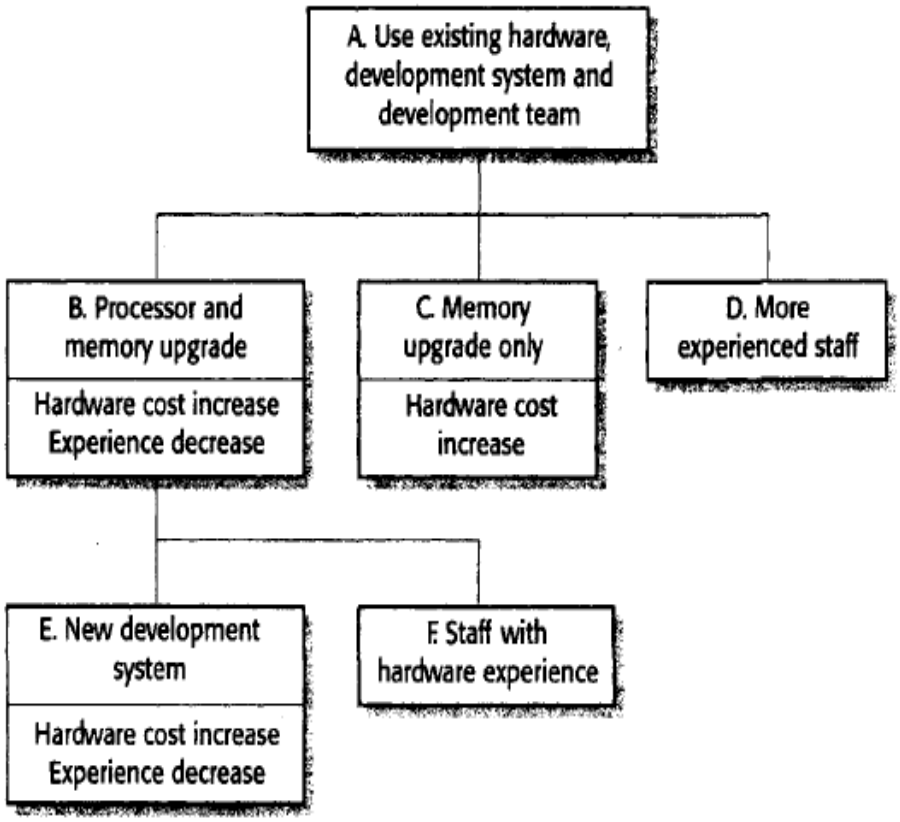


Figure 26.13 Cost of Management options

Option	RELY	STOR	TIME	TOOLS	LTEX	Total effort	Software cost	Hardware cost	Total cost
A	1.39	1.06	1.11	0.86	1	63	949393	100000	1049393
B	1.39	1	1	1.12	1.22	88	1313550	120000	1402025
C	1.39	1	1.11	0.86	1	60	895653	105000	1000653
D	**1.39**	**1.06**	**1.11**	**0.86**	**0.84**	**51**	**769008**	**100000**	**897490**
EX	1.39	1	1	0.72	1.22	56	844425	220000	1044159
F	1.39	1	1	1.12	0.84	57	851180	120000	1002706

- Option D appears to offer the lowest costs for all basic estimates. No additional hardware expenditure is involved but new staff must be recruited onto the project.
- If these are already available in the company, this is probably the best option to choose.
- If not, they must be recruited externally, which involves significant costs and risks.

- These may mean that the cost advantages of this option are much less significant than suggested by Figure 26.13.
- Option C offers a saving of almost $50,000 with virtually no associated risk. Conservative project managers would probably select this option rather than the riskier Option D.

5.2 (5) Project Duration and Staffing

- To develop a software system and the overall project costs., project managers must also estimate how long the software will take to develop and when staff will be needed to work on the project.
- The development Line for the project is called the project schedule The relationship between the number of staff working on a project, the total effort required and the development time is not linear.
- As the number of staff increases, more effort may be needed. The reason for this is that people spend more time communicating and defining interfaces between the parts of the system developed by other people.
- Doubling the number of staff (for example) therefore does not mean that the duration of the project will be halved.
- The COCOMO model includes a formula to estimate the calendar time (TDEV) required to complete a project.
- The time computation formula is the same for all COCOMO levels:
- $TDEV = 3 \times (PM)^{(0.33+0.2*(B-1.01))}$
- PM is 1.he effort computation and 8 is the exponent computed, as discussed above (8 is 1for the early prototyping model). This computation predicts the nominal schedule for the project.
- However the predicted project schedule and the schedule required by the project plan are not necessarily the same thing.
- The planned schedule may be shorter or longer than the nominal predicted schedule.

- However, there is obviously a limit to the extent of schedule changes, and the COCOMO II model predicts this:
- ***TDEV= 3 X (PM)*** $^{(0.33+0.2*(B-1.01))}$ ***X SCED Percentage / 100***
- SCED Percentage is the percentage increase or decrease in the nominal schedule.
- If the predicted figure then differs significantly from the planned schedule, it suggests that there is a high risk of problems delivering the software as planned.
- To illustrate the COCOMO development schedule computation, assume that 60 months 01 effort are estimated to develop a software system (Option C in Figure 26.12).
- Assume that the value of exponent B is 1.17. From the schedule equation, the time required to complete the project is:
- ***TDEV= 3 X (60) 0.36= 13 months***
- In this case, there is no schedule compression or expansion, so the last term in the formula has no effect on the computation.
- An interesting implication of the COCOMO model is that the time required to complete the project is a function of the total effort required for the project.
- It does not depend on the number of software engineers working on the project.