

SOFTWARE PROJECT MANAGEMENT

DEPARTMENT OF CSE

SYLLABUS (THEORY)

Sub. Code : MG6088

Branch / Year / Sem : CSE / IV / VIII

Sub.Name : SOFTWARE PROJECT MANAGEMENT

Staff Name : R.VENKATESAN

UNIT I SOFTWARE PROCESS AND PROJECT MANAGEMENT 9

Introduction to Software Engineering, Software Process, Perspective and Specialized Process Models – Software Project Management: Estimation – LOC and FP Based Estimation, COCOMO Model – Project Scheduling – Scheduling, Earned Value Analysis - Risk Management.

UNIT II REQUIREMENTS ANALYSIS AND SPECIFICATION 9

Software Requirements: Functional and Non-Functional, User requirements, System requirements, Software Requirements Document – Requirement Engineering Process: Feasibility Studies, Requirements elicitation and analysis, requirements validation, requirements management-Classical analysis: Structured system Analysis, Petri Nets- Data Dictionary.

UNIT III SOFTWARE DESIGN 9

Design process – Design Concepts-Design Model– Design Heuristic – Architectural Design – Architectural styles, Architectural Design, Architectural Mapping using Data Flow- User Interface Design: Interface analysis, Interface Design –Component level Design: Designing Class based components, traditional Components.

UNIT IV TESTING AND IMPLEMENTATION 9

Software testing fundamentals-Internal and external views of Testing-white box testing- basis path testing-control structure testing-black box testing- Regression Testing – Unit Testing – Integration Testing – Validation Testing – System Testing And Debugging – Software Implementation Techniques: Coding practices-Refactoring.

UNIT V PROJECT MANAGEMENT 9

Estimation – FP Based, LOC Based, Make/Buy Decision, COCOMO II - Planning – Project Plan, Planning Process, RFP Risk Management – Identification, Projection, RMMM - Scheduling and Tracking –Relationship between people and effort, Task Set & Network, Scheduling, EVA – Process and Project Metrics.

TOTAL: 45 PERIODS

UNIT I

SOFTWARE PROCESS AND PROJECT MANAGEMENT

1.1 INTRODUCTION TO SOFTWARE ENGINEERING

Software engineering paradigm:

- The framework activities will always be applied on every project ... BUT the tasks (and degree of rigor) for each activity will vary based on:
 - the type of project
 - characteristics of the project
 - common sense judgment; concurrence of the project team

The software process:

- A structured set of activities required to develop a software system
 - Specification;
 - Design;
 - Validation;
 - Evolution.
- A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

1.2 SOFTWARE PROCESS

Introduction

Software engineering is the establishment and sound engineering principles applied to obtain reliable and efficient software in an economical manner

Software engineering includes process, management techniques, technical methods, and the use of tool. While building any software, the software process provides the interaction between user and developer. In this chapter we will understand the basic concept of process and process models. We will discuss how to use various life cycle models for building qualitative software in an economic manner. Finally we will focus on system engineering.

Layered Technology

- Software engineering is a layered technology. Any software can be developed using these .layered approaches. Various layers on which the technology is based are quality focus layer, process layer, methods layer, tools layer.
- A disciplined quality management is a backbone of software engineering technology.
- Process layer is a foundation of software engineering. Basically, process defines the framework for timely delivery of software.
- In method layer the actual method of implementation is carried out with the help of

requirement analysis, designing, coding using desired programming constructs and testing.

- Software tools are used to bring automation in software development process.

Thus software engineering is a combination of process, methods, and tools for development of quality software.

Software Process

Software process can be defined as the structured set of activities that are required to develop the software system.

The fundamental activities are

- Specification
- Design and implementation
- Validation
- Evolution

A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

1.2.1 COMMON PROCESS FRAMEWORK

The process framework is required for representing the common process activities. It is as shown below.

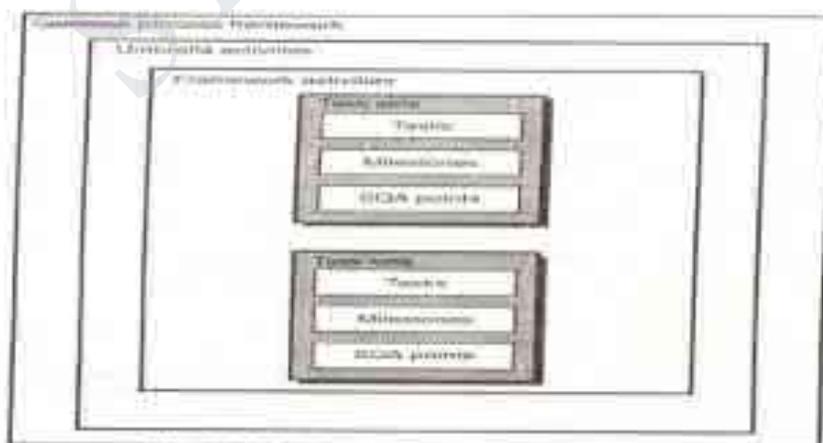


Fig. 2-2 Software process framework

As shown in figure the software process is characterized by process framework activities, task sets and umbrella activities.

Process Framework Activities

- Communication
 - By communicating customer requirement gathering is done.
- Planning - Establishes engineering work plan, describes technical risks, lists resource requirements, work products produced, and defines work schedule.
- Modeling - The software model is prepared by:
 - Analysis of requirements
 - Design
- Construction ~ The software design is mapped into a code by:
 - Code generation.
 - Testing
- Deployment - The software delivered for customer evaluation and feedback is obtained.

Task Sets - The task set defines the actual work done in order to achieve the software objective. The task set is used to adopt the framework activities and project team requirements using

Collection of software engineering work tasks

Project milestones

Software quality assurance points

Umbrella activities - The umbrella activities occur throughout the process. They focus on project management, tracking and control. The umbrella activities are

1. Software project tracking and control - This is an activity in which software team can assess progress and take corrective action to maintain schedule.
2. Risk management- The risks that may affect project outcomes or quality can be analyzed.
3. Software quality assurance - These are activities required to maintain software quality.
4. Formal technical reviews - It is required to assess engineering work products to uncover and remove errors before they propagate to next activity.

5. Software configuration management - Managing of configuration process when any change in the software occurs.
6. Work product preparation and production - The activities to create models, documents, logs, forms, and lists are carried out.
7. Reusability management - It defines criteria for work product reuse.
8. Measurement - In this activity, the process can be defined and collected. Also project and product measures are used to assist the software team in delivering the required software.

1.2.2 CAPABILITY MATURITY MODEL (CMM):

The Software Engineering Institute (SEI) has developed a comprehensive process meta-model emphasizing process maturity. It is predicated on a set of system and software capabilities that should be present when organizations reach different levels of process capability and maturity .

' The Capability Maturity Model (CMM) is used in assessing how well an organization's processes allow to complete and manage new software projects.

Various process maturity levels are

Level 1 : Initial - Few processes are defined and individual efforts are taken.

Level 2 : Repeatable - To track cost schedule and functionality basic project management processes are established .Depending on earlier successes of projects with similar applications necessary process discipline can be repeated.

Level 3 : Defined - The process is standardized, documented and followed. All the projects use documented and approved version of software process which is useful in developing and supporting software.

Level 4 : Managed - Both the software process and product are quantitatively understood and controlled using detailed measures.

Level 5 : Optimizing - Establish mechanisms to plan and implement change. Innovative ideas and technologies can be tested.

Thus CMM is used for improving the software project.

1.2.3 SOFTWARE ENGINEERING PARADIGM :

Software engineering paradigm or process model IS an abstract representation of a process.

The process model is chosen based on nature of software project and application and then to obtain deliverable product method and tools are applied.

Using problem solving loop the software development can be done. The problem solving loop includes.

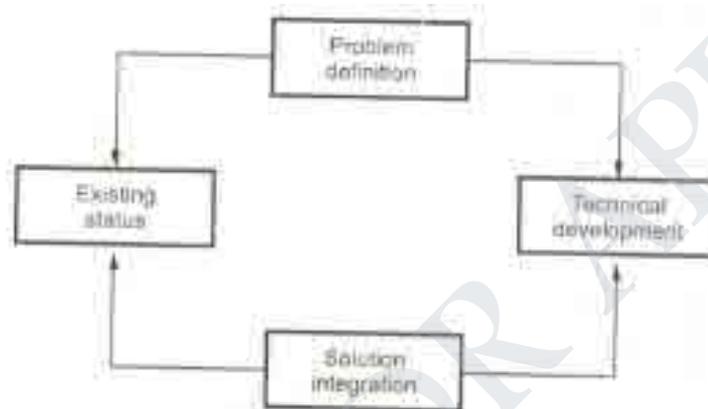


Fig. 2.3 Problem solving loop

The **existing status** that represents current state of affairs.

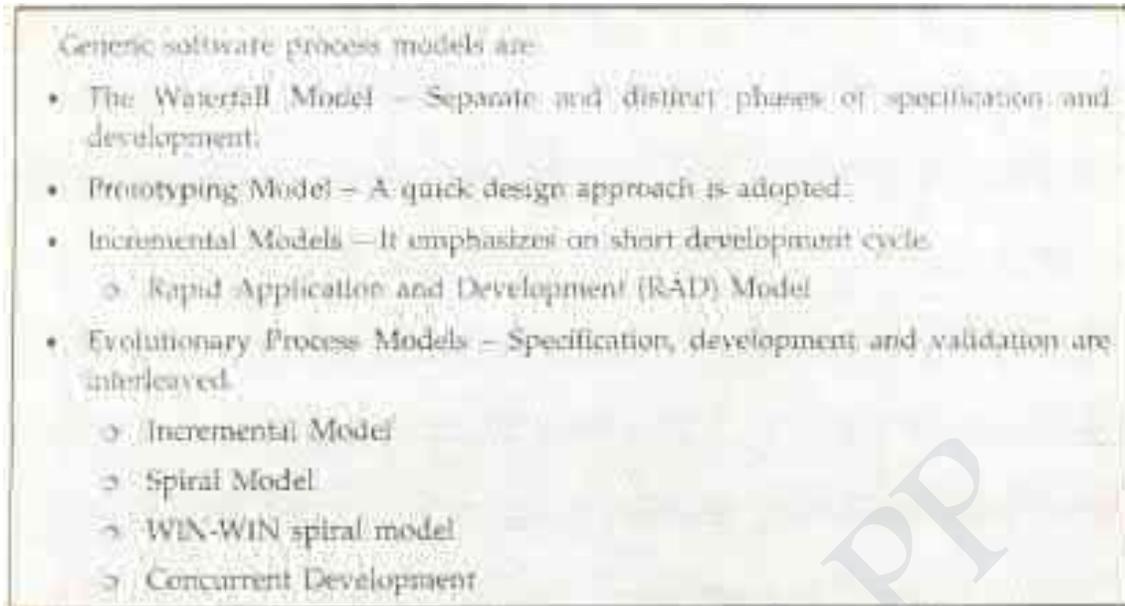
In the **problem identification phase** particular problem is identified.

The **technical development stage** is for solving the identified problem using an appropriate technology.

Finally **solution integration** is responsible for delivering the results.

But applying such problem solving loop in software development process is very difficult because we can not strictly categorize the development in these phases.

There may be a requirement of cross talk within and across stages. Hence some software process models are suggested depending upon nature of software. Such models are called generic software models.



1.3 PERSPECTIVE AND SPECIALIZED PROCESS MODELS:

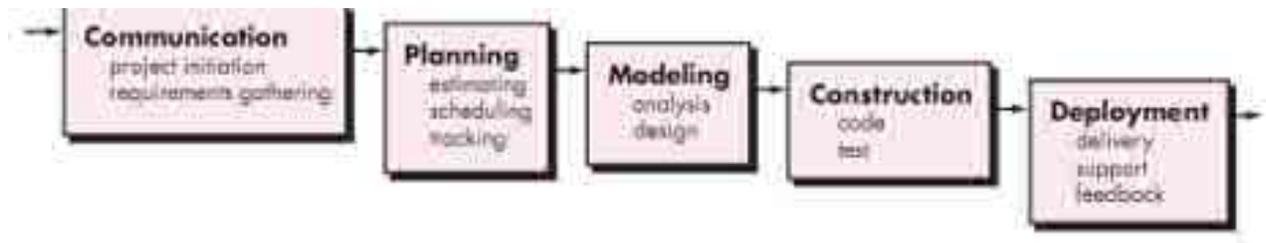
1.3.1 PRESCRIPTIVE PROCESS MODEL

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. However, software engineering work and the product that it produces remain on “the edge of chaos.”

Here are no easy answers to these questions, but there are alternatives available to software engineers. In the sections that follow, I examine the prescriptive process approach in which order and project consistency are dominant issues. I call them “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.

1.3.1.1 Waterfall model/Linear Sequential Model/classic life cycle :



- **Systems Engineering**
 - Software as **part of larger system, determine requirements for all system elements, allocate requirements to software.**
- **Software Requirements Analysis**
 - Develop understanding of problem domain, user needs, function, performance, interfaces, ...
 - **Software Design**
 - **Multi-step process to determine architecture, interfaces, data structures, functional detail. Produces (high-level) form that can be checked for quality, conformance before coding.**
- **Coding**
 - Produce machine readable and executable form, match HW, OS and design needs.
- **Testing**
 - Confirm that components, subsystems and complete products meet requirements, specifications and quality, find and fix defects.
- **Maintenance**
 - Incrementally, **evolve software** to fix defects, add features, adapt to new condition. Often 80% of effort spent here!

Waterfall model phases:

- Requirements analysis and definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance
- The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. One phase has to be complete before moving onto the next phase.
- Each phase terminates only when the documents are complete and approved by the SQA group.
- Maintenance begins when the client reports an error after having accepted the product. It could also begin due to a change in requirements after the client has accepted the product

Waterfall model: Advantages:

- Disciplined approach
- Careful checking by the Software Quality Assurance Group at the end of each phase.
- Testing in each phase.

- Documentation available at the end of each phase.

Waterfall model problems:

- It is difficult to respond to changing customer requirements.
- Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
- Few business systems have stable requirements.
- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
- The customer must have patience. A working version of the program will not be available until late in the project time-span
- Feedback from one phase to another might be too late and hence expensive.

1.3.1.2 The Prototyping Models:

- Often, a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements.
- 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 this case prototyping paradigm may offer the best approach
- Requirements gathering
- Quick design
- Prototype building
- Prototype evaluation by customers
- Prototype may be refined
- Prototype thrown away and software developed using formal process{ it is used to define the requirement } Prototyping

Strengths:

- Requirements can be set earlier and more reliably
- Customer sees results very quickly.
- Customer is educated in what is possible helping to refine requirements.
- Requirements can be communicated more clearly and completely
- Between developers and clients Requirements and design options can be investigated quickly and Cheaply

Weaknesses:

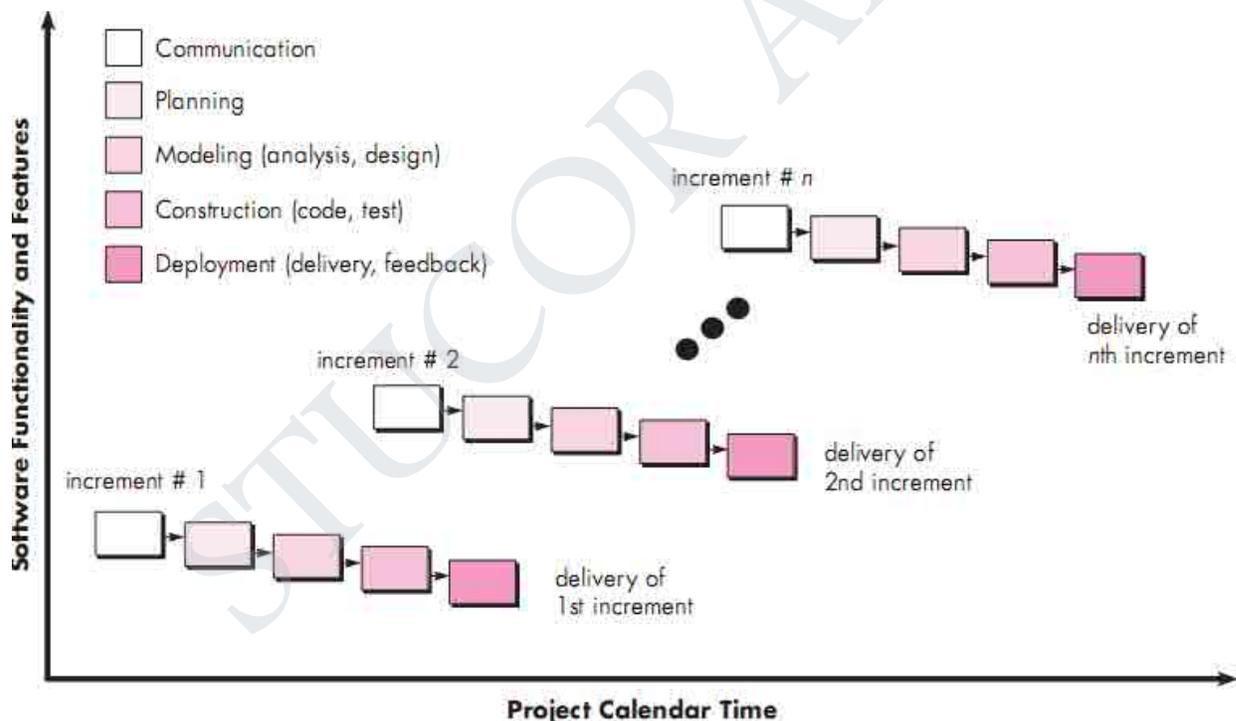
- Requires a rapid prototyping tool and expertise in using it—a cost for the development organisation
- Smoke and mirrors - looks like a working version, but it is not.

1.3.1.3 Incremental Model

- The incremental model has same phases that are in waterfall model. But it is iterative in nature. The incremental model has following phases.

1. Analysis
2. Design
3. Code
4. Test

- The incremental model delivers series of releases to the customer. These releases are called increments. More and more functionality is associated with each increment.
- The first increment is called core product. In this release the basic requirements are implemented and then in subsequent increments new requirements are added.
- The word processing software package can be considered as an example of incremental model. In the first increment only the document processing facilities are available. In the second increment, more sophisticated document producing and processing facilities, file management functionalities are given. In the next increment spelling and grammar checking facilities can be given. Thus in incremental model progressive functionalities are obtained with each release.



When to choose it?

1. When requirements are reasonably well-defined.
2. When overall scope of the development effort suggests a purely linear effort.
3. When limited set of software functionality needed quickly.

Merits of incremental model

1. The incremental model can be adopted when there are less number of people involved in the project.
2. Technical risks can be managed with each increment.
3. For a very small time span, at least core product can be delivered to the customer.

1.3.1.4 Rapid Application Development (RAD) Model

- The rapid application development model is type of incremental software process model in which there is extremely short development cycle.
- This model is similar to waterfall model which achieves the high speed development using compoilent based construction.
- To develop the fully functional system within short time period using this model it is necessary to understand the requirements fully and to have a restricted project scope.

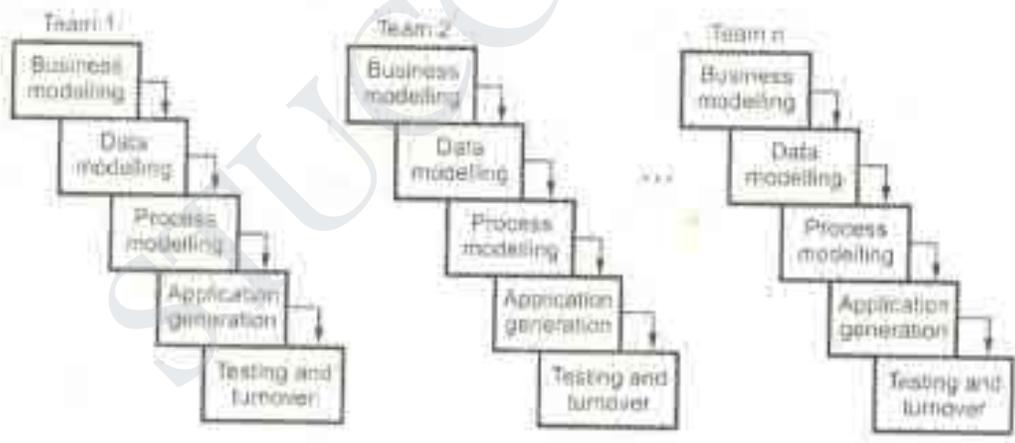


Fig. 2.6 Rapid application development model

- Various phases of RAD model are
- 1) **Business modeling** - In business modeling, the information flow is modeled into various business functions. These business functions collect following information.
 - Information that drives the business process.

- The type of information being generated.
 - The generator of information.
 - The information flow.
 - The processor of information.
- 2) **Data modeling** - In this phase the information obtained in business model' classified into data objects. The characteristics of data objects (attributes) are identified. The relationship among various data objects is defined.
 - 3) **Process modeling** - In this phase the data objects are transformed into processes. These processes are to extract the information from data objects and are responsible for implementing business functions.
 - 4) **Application generation** - For creating software various automation tools can be used. RAD also makes use of reusable components or creates reusable components to have rapid development of software.
 - 5) **Testing and turnover** - As RAD uses reusable components the testing efforts are reduced. But if new components are added in software develop men process then such components need to be tested. It is equally important to test all the interfaces.

1.3.1.5 Spiral Model:

- This model possesses the iterative nature of prototyping model and controlled and systematic approaches of the linear sequential model.
- This model gives efficient development of incremental versions of software. In this model, the software is developed in series of increments.
- The spiral model is divided into a number of framework activities. These framework activities are denoted by task regions.
- Usually there are six tasks regions. The spiral model is as shown in Fig.

In the initial pass, product specification is built and in subsequent passes around the spiral the prototype gets developed and then more improved versions of software gets developed.

- During planning phase, the cost and schedule of software can be planned and adjusted

carried out in this task region.

- vi. Customer evaluation - Customer's feedback is obtained and based on customer evaluation required tasks are performed and implemented at installation stage.

STUCOR APP

In each region, numbers of work tasks are carried out depending upon the characteristics of project. For a small project relatively small number of work tasks is adopted but for a complex project large number of work tasks can be carried out.

- In spiral model, the software engineering team moves around the spiral in a clockwise direction beginning at the core.

Drawbacks of spiral model

- It is based on customer communication. If the communication is not proper then the software product that gets developed will not be the up to the mark.
- It demands considerable risk assessment. If the risk assessment is done properly then only the successful product can be obtained.

Prototyping

- In prototyping model initially the requirement gathering is done.
- Developer & customer define overall objectives; identify areas needing more requirement gathering.
- Then a quick design is prepared. This design represents what will be visible to user- in input and output format.
- From the quick design a prototype is prepared. Customer or user evaluates the prototype in order to refine the requirements. Iteratively prototype is tuned for satisfying customer requirements. Thus prototype is important to identify the software requirements.
- When working prototype is built, developer use existing program fragments or program generators", to throwaway the prototype and rebuild the system to high quality
- Certain classes of mathematical algorithms, subset of command driven systems and other applications where results can be easily examined without real time interaction can be developed using prototyping paradigm.

When to choose it?

- Software applications that are relatively easy to prototype almost always involve human-machine interaction (He!) the prototyping model is suggested.
- A general objective of software is defined but not detailed input, processing or output requirements. Then in such a case prototyping model is useful.
- When the developer is unsure of the efficiency of an algorithm or the adaptability of an

operating system then prototype serves as a better choice.

Drawbacks of Prototyping

1. In the first version itself, customer often wants "few fixes" rather than rebuilding of the system. Whereas rebuilding of new system maintains high level of quality.
2. The first version may have some compromises.
3. Sometimes developer may make implementation compromises to get prototype working quickly. Later on developer may become comfortable with compromises and forget why they are inappropriate.

1.3.2 SPECIALIZED PROCESS MODEL

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.

Component-Based Development

- Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built.
- The *component-based development model* incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components.
- Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture.

The Formal Methods Model

The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called clean room software engineering is currently applied by some software development organizations.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected.

Although not a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

- The development of formal models is currently quite time consuming and expensive.
- Because few software developers have the necessary background to apply formal methods, extensive training is required.
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, the formal methods approach has gained adherents among software developers who must build safety-critical software (e.g., developers of aircraft avionics and medical devices) and among developers that would suffer severe economic hardship should software errors occur.

Aspect-Oriented Software Development

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., objectoriented classes) and then constructed within the context of a system architecture. As modern computer-based systems

become more sophisticated (and complex), certain *concerns*—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).

When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns*. *Aspectual requirements* define those crosscutting concerns that have an impact across the software architecture.

Aspect-oriented software development (AOSD), often referred to as *aspect-oriented programming* (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing *aspects*—“mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern”.

A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both evolutionary and concurrent process models. The evolutionary model is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components. Hence, it is essential to instantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components.

1.4 SOFTWARE PROJECT MANAGEMENT: ESTIMATION

1.4.1 SOFTWARE PROJECT MANAGEMENT

Effective software project management focuses on the four P's: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project.

The People

The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the “people factor” is so important that the Software Engineering Institute has

developed a *People Capability Maturity Model* (People-CMM), in recognition of the fact that “every organization needs to continually improve its ability to attract, develop, motivate, organize, and retain the workforce needed to accomplish its strategic business objectives” .

The people capability maturity model defines the following key practice areas for software people: staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, team/culture development, and others. Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices.

The Product

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress. As a software developer, you and other stakeholders must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements engineering (Chapter 5). Objectives identify the overall goals for the product (from the stakeholders’ points of view) without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner.

The Process

A software process provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large

software projects between 1998 and 2004, Capers Jones [Jon04] found that “about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below 35 percent, while about 175 experienced major delays and overruns, or were terminated without completion.” Although the success rate for present-day software projects may have improved somewhat, our project failure rate remains much higher than it should be.

To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project. Each of these issues is discussed in Section 24.5 and in the chapters that follow.

1.4.2 ESTIMATION

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, you should decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems.

Software Sizing

The accuracy of a software project estimate is predicated on a number of things: (1) the degree to which you have properly estimated the size of the product to be built; (2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects); (3) the degree to which the project plan reflects the abilities of the software team; and (4) the stability of product requirements and the environment that supports the software engineering effort.

In this section, I consider the *software sizing* problem. Because a project estimate is only as good as the estimate of the size of the work to be accomplished, sizing represents your first major challenge as a planner. In the context of project planning, size refers to a quantifiable outcome of the software project. If a direct approach is taken, size can be measured in lines of code (LOC). If an indirect approach is chosen, size is represented as function points (FP).

Putnam and Myers suggest four different approaches to the sizing problem:

1. *“Fuzzy logic” sizing.* This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.
2. *Function point sizing.* The planner develops estimates of the information domain characteristics discussed.

3. Standard component sizing. Software is composed of a number of different “standard components” that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to estimate the delivered size per standard component.

4. Change sizing. This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished.

1.5 LOC AND FP BASED ESTIMATION:

LOC-based Estimation

- The problems of lines of code (LOC)

Different languages lead to different lengths of code

It is not clear how to count lines of code

A report, screen, or GUI generator can generate thousands of lines of code in minutes

Depending on the application, the complexity of code is different.

Function Point Analysis

The Five Components of Function Points

- Internal Logical Files
- External Interface Files

Transactional Functions

- External Inputs
- External Outputs
- External Inquiries

Internal Logical Files - The first data function allows users to utilize data they are responsible for maintaining. For example, a pilot may enter navigational data through a display in the cockpit prior to departure. The data is stored in a file for use and can be modified during the mission. Therefore the pilot is responsible for maintaining the file that contains the navigational information. Logical groupings of data in a system, maintained by an end user, are referred to as Internal Logical Files (ILF).

External Interface Files - The second Data Function a system provides an end user is also related to logical groupings of data. In this case the user is not responsible for maintaining the data. The data resides in another system and is maintained by another user or system. The user of the system

being counted requires this data for reference purposes only. For example, it may be necessary for a pilot to reference position data from a satellite or ground-based facility during flight. The pilot does not have the responsibility for updating data at these sites but must reference it during the flight. Groupings of data from another system that are used only for reference purposes are defined as External Interface Files (EIF).

External Input - The first Transactional Function allows a user to maintain Internal Logical Files (ILFs) through the ability to add, change and delete the data. For example, a pilot can add, change and delete navigational information prior to and during the mission. In this case the pilot is utilizing a transaction referred to as an External Input (EI). An External Input gives the user the capability to maintain the data in ILF's through adding, changing and deleting its contents.

External Output - The next Transactional Function gives the user the ability to produce outputs. For example a pilot has the ability to separately display ground speed, true air speed and calibrated air speed. The results displayed are derived using data that is maintained and data that is referenced. In function point terminology the resulting display is called an External Output (EO).

External Inquiries - The final capability provided to users through a computerized system addresses their requirement to select and display specific data from files. To accomplish this a user inputs selection information that is used to retrieve data that meets the specific criteria. In this situation there is no manipulation of the data. It is a direct retrieval of information contained on the files. For example if a pilot displays terrain clearance data that was previously set, the resulting output is the direct retrieval of stored information. These transactions are referred to as External Inquiries (EQ).

Functional Complexity - The first adjustment factor considers the Functional Complexity for each unique function. Functional Complexity is determined based on the combination of data groupings and data elements of a particular function. The number of data elements and unique groupings are counted and compared to a complexity matrix that will rate the function as low, average or high complexity. Each of the five functional components (ILF, EIF, EI, EO and EQ) has its own unique complexity matrix. The following is the complexity matrix for External Outputs.

	1-5 DETs	6 - 19 DETs	20+ DETs
0 or 1 FTRs	L	L	A
2 or 3 FTRs	L	A	H
4+ FTRs	A	H	H

Complexity	UFP
L (Low)	4
A (Average)	5
H (High)	7

Using the examples given above and their appropriate complexity matrices, the function point count for these functions would be:

Function Name	Function Type	Record Element Type	Data Element Type	File Types Referenced	Unadjusted FPs
Navigational data	ILF	3	36	n/a	10
Positional data	EIF	1	3	n/a	5
Navigational data - add	EI	n/a	36	1	4
Navigational data - change	EI	n/a	36	1	4
Navigational data - delete	EI	n/a	3	1	3
Ground speed	EO	n/a	20	3	7

display					
Air speed display	EO	n/a	20	3	7
Calibrated air speed display	EO	n/a	20	3	7
Terrain clearance display	EQ	n/a	1	1	3
Total unadjusted count					50 UFPs

1.6 COCOMO MODEL:

The COCOMO cost estimation model is used by thousands of software project managers, and is based on a study of hundreds of software projects. Unlike other cost estimation models, COCOMO is an open model, so all of the details are published, including:

- The underlying cost estimation equations
- Every assumption made in the model (e.g. "the project will enjoy good management")
- Every definition (e.g. the precise definition of the Product Design phase of a project)
- The costs included in an estimate are explicitly stated (e.g. project managers are included, secretaries aren't)
- Because COCOMO is well defined, and because it doesn't rely upon proprietary estimation algorithms, Costar offers these advantages to its users:
- COCOMO estimates are more objective and repeatable than estimates made by methods relying on proprietary models
- COCOMO can be calibrated to reflect your software development environment, and to produce more accurate estimates

Costar is a faithful implementation of the COCOMO model that is easy to use on small projects, and yet powerful enough to plan and control large projects.

Typically, you'll start with only a rough description of the software system that you'll be developing, and you'll use Costar to give you early estimates about the proper schedule and staffing levels. As you refine your knowledge of the problem, and as you design more of the system, you can use Costar to produce more and more refined estimates.

Source Lines of Code

The COCOMO calculations are based on your estimates of a project's size in Source Lines of Code (SLOC). SLOC is defined such that:

The original COCOMO 81 model was defined in terms of Delivered Source Instructions, which are very similar to SLOC. The major difference between DSI and SLOC is that a single Source Line of Code may be several physical lines. For example, an "if-then-else" statement would be counted as one SLOC, but might be counted as several DSI.

The Scale Drivers

In the COCOMO II model, some of the most important factors contributing to a project's duration and cost are the Scale Drivers. You set each Scale Driver to describe your project; these Scale Drivers determine the exponent used in the Effort Equation.

The 5 Scale Drivers are:

- Precedentedness
- Development Flexibility
- Architecture / Risk Resolution
- Team Cohesion
- Process Maturity

1.7 PROJECT SCHEDULING – SCHEDULING:

Project scheduling is a mechanism to communicate what tasks need to get done and which organizational resources will be allocated to complete those tasks in what timeframe. A project schedule is a document collecting all the work needed to deliver the project on time.

- But when it comes to creating a project schedule, well, that's something few have deep experience with.
- What and who is being scheduled, and for what purposes, and where is this scheduling taking place, anyway?

A project is made up of many tasks, and each task is given a start and end (or due date), so it can be completed on time.

- Likewise, people have different schedules, and their availability and vacation or leave dates need to be documented in order to successfully plan those tasks.
- Whereas people in the past might have printed calendars on a shared wall in the water-cooler room, or shared spreadsheets via email, today most teams use online project scheduling tools.
- Typically, project scheduling is just one feature within a larger project management software solution, and there are many different places in the software where scheduling takes place.
- For example, most tools have task lists, which enable the manager to schedule multiple tasks, their due dates, sometimes the planned effort against that task, and then assign that task to a person.
- The software might also have resource scheduling, basically the ability to schedule the team's availability, but also the availability of non-human resources like machines or buildings or meeting rooms.

Because projects have so many moving parts, and are frequently changing, project scheduling software automatically updates tasks that are dependent on one another, when one scheduled task is not completed on time. It also generates automated email alerts, so team members know when their scheduled tasks are due or overdue, and to let the manager know when someone's availability has changed.

1.7.1 SCHEDULING TASKS:

What you want when scheduling tasks is not a glorified to-do list, but a smart software that gives you the flexibility to handle the variety of responsibilities attached to each tasks in your project.

An interactive Gantt chart is crucial. You can add tasks and dates into your Gantt chart to have a visual representation of each task's duration. Better still, as dates change—as they inevitably do—you can simply drag and drop those changes and the whole Gantt chart is updated instantly.

There's also automating processes to help with efficiencies. Email notifications are a great way to know immediately when a team member has completed a task.

- When they update, you know because your software is online and responding in real-time.
- Continuing with automation, it's one way to scheduling tasks more efficiently. If there are recurring tasks on a project, they can be scheduled in your PM tool so that once set you don't have to worry about scheduling the same task over and over again.

1.8 EARNED VALUE ANALYSIS:

Earned value analysis is the project management tool that is used to measure project progress. It compares the actual work completed at any time to the original budget and schedule. It forecasts the final budget and schedule and analyzes the path to get there. It gives you the essential early warning signal that things are going awry.

There are two variables which the earned value method focuses on.

- Schedule (time)
- Cost

There are 8 steps to performing earned value analysis effectively. It may seem like a lot at first glance, but for small projects this takes five minutes once you learn how to do it:

1. Determine the percent complete of each task.
2. Determine Planned Value (PV).
3. Determine Earned Value (EV).
4. Obtain Actual Cost (AC).
5. Calculate Schedule Variance (SV).
6. Calculate Cost Variance (CV).
7. Calculate Other Status Indicators (SPI, CPI, EAC, ETC, and TCPI)
8. Compile Results

The first four steps represent an information gathering phase. The remaining steps are calculations which give the project manager a glimpse into the current status of the project from a budget and schedule perspective.

Before you get started, it is important to define appropriate project status points in which this calculation is performed. Weekly status meetings work very well for any size project, but whatever time frame is used the important thing is to make sure these calculations are performed at that time.

Determine Percent Complete

To start the process, the percentage complete of each task needs to be determined.

Small tasks (80 hours or less) are often best done on a 0, 50, or 100% complete basis (not started, in progress, or complete). This brings the workload down to reasonable levels and prevents abuse when project team members exaggerate, for example they might tell you a task is 80% complete when it is really 50% complete.

Determine Planned Value (PV)

Planned Value, also known as **Budgeted Cost of Work Scheduled (BCWS)**, is defined as the amount of the task that is supposed to have been completed. It is in monetary terms as a portion of the task budget. For example let's say that:

- The task budget is \$5,000,
- The task start date is January 1, and
- The task finish date is January 10.

If it's January 6 today, the task is supposed to be 60% complete. Therefore, $PV = \$5,000 \times 60\% = \$3,000$.

Determine Earned Value (EV)

Earned Value, also known as **Budgeted Cost of Work Performed (BCWP)**, is the amount of the task that is actually complete. It is, again, in monetary terms as a portion of the task budget. For example, let's use the same example task.

- The task budget is \$5,000, (same as above)

- The task start date is January 1, and (same as above)
- The task finish date is January 10. (same as above)

Let's say the actual percent complete of the task (step 1) is 40%. Therefore, $EV = \$5,000 \times 40\% = \$2,000$.

1.9 RISK MANAGEMENT:

Software development is activity that uses a variety of technological advancements and requires high levels of knowledge. Because of these and other factors, every software development project contains elements of uncertainty. This is known as project risk. The success of a software development project depends quite heavily on the amount of risk that corresponds to each project activity. As a project manager, it's not enough to merely be aware of the risks. To achieve a successful outcome, project leadership must identify, assess, prioritize, and manage all of the major risks.

The goal of most software development and software engineering projects is to be distinctive—often through new features, more efficiency, or exploiting advancements in software engineering. Any software project executive will agree that the pursuit of such opportunities cannot move forward without risk.

Because risks are painfully real and quite prevalent on all software projects, it's critically necessary that stakeholders work hard to identify, understand, and mitigate any risks that might threaten the success of a project. For projects that have time and cost constraints, our experience shows most clearly that successful software development efforts are those in which risk mitigation is a central management activity.

1.9.1 WHAT IS RISK IN SOFTWARE ENGINEERING?

Very simply, a risk is a *potential* problem. It's an activity or event that may compromise the success of a software development project. Risk is the possibility of suffering loss, and total risk exposure to a specific project will account for both the *probability* and the *size* of the potential loss.

Guesswork and crisis-management are never effective. Identifying and aggregating risks is the only predictive method for capturing the probability that a software development project will experience unplanned or inadmissible events. These include terminations, discontinuities, schedule delays, cost underestimation, and overrun of project resources

What Is Risk Management In Software Engineering?

- Risk management means risk containment and mitigation. First, you've got to identify and plan. Then be ready to act when a risk arises, drawing upon the experience and knowledge of the entire team to minimize the impact to the project.

Risk management includes the following tasks:

Identify risks and their triggers

- **Classify** and prioritize all risks
- Craft a **plan** that links each risk to a mitigation
- **Monitor** for risk triggers during the project
- Implement the **mitigating action** if any risk materializes
- **Communicate** risk status throughout project

Identify and Classify Risks

The importance here is not the elegance or range of classification, but rather to precisely *identify and describe all of the real threats to project success*. A simple but effective classification scheme is to arrange risks according to the areas of impact.

Five Types of Risk In Software Project Management

For most software development projects, we can define five main risk impact areas:

- New, unproven technologies
- User and functional requirements & Application and system architecture
- Performance & Organizational

UNIT II

REQUIREMENTS ANALYSIS AND SPECIFICATION

2.1 SOFTWARE REQUIREMENTS

The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed

- Requirements may be functional or non-functional
- Functional requirements describe system services or functions
- Non-functional requirements is a constraint on the system or on the development process

Types of requirements

- User requirements
 - Statements in natural language (NL) plus diagrams of the services the system provides and its operational constraints. Written for customers
- System requirements
 - A structured document setting out detailed descriptions of the system services. Written as a contract between client and contractor
- Software specification
 - A detailed software description which can serve as a basis for a design or implementation. Written for developers

2.2 FUNCTIONAL AND NON-FUNCTIONAL

2.2.1 Functional requirements

- Functionality or services that the system is expected to provide.
- Functional requirements may also explicitly state what the system shouldn't do.
- Functional requirements specification should be:
 - Complete: All services required by the user should be defined
 - Consistent: should not have contradictory definition (also avoid ambiguity □ don't leave room for different interpretations)

Examples of functional requirements

- The LIBSYS system
- A library system that provides a single interface to a number of databases of articles in different libraries.
- Users can search for, download and print these articles for personal study.
- The user shall be able to search either all of the initial set of databases or select a subset from it.
- The system shall provide appropriate viewers for the user to read documents in the document store.
- Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account's permanent storage area.

2.2.2 Non-Functional requirements

- Requirements that are not directly concerned with the specific functions delivered by the system
- Typically relate to the system as a whole rather than the individual system features
- Often could be deciding factor on the survival of the system.

2.3 USER REQUIREMENTS

Domain requirements

- Domain requirements are derived from the application domain of the system rather than from the specific needs of the system users.
- May be new functional requirements, constrain existing requirements or set out how particular computation must take place.
- Example: tolerance level of landing gear on an aircraft (different on dirt, asphalt, water), or what happens to fiber optics line in case of sever weather during winter Olympics (Only domain-area experts know)

Product requirements

- Specify the desired characteristics that a system or subsystem must possess.
- Most NFRs are concerned with specifying constraints on the behaviour of the executing system.

Specifying product requirements

- Some product requirements can be formulated precisely, and thus easily quantified
 - Performance
 - Capacity
- Others are more difficult to quantify and, consequently, are often stated informally
 - Usability

System requirements

2.4 PROCESS REQUIREMENTS

- Process requirements are constraints placed upon the development process of the system
- Process requirements include:
 - Requirements on development standards and methods which must be followed
 - CASE tools which should be used
 - The management reports which must be provided

Examples of process requirements

- The development process to be used must be explicitly defined and must be conformant with ISO 9000 standards
- The system must be developed using the XYZ suite of CASE tools
- Management reports setting out the effort expended on each identified system component must be produced every two weeks
- A disaster recovery plan for the system development must be specified

2.4.1 External requirements

- May be placed on both the product and the process
- Derived from the environment in which the system is developed
- External requirements are based on:
 - application domain information
 - organisational considerations
 - the need for the system to work with other systems
 - health and safety or data protection regulations
 - or even basic natural laws such as the laws of physics

Examples of external requirements

- Medical data system The organisation's data protection officer must certify that all data is maintained according to data protection legislation before the system is put into operation.

2.5 SOFTWARE REQUIREMENTS DOCUMENT

- Should provide for communication among team members
- Should act as an information repository to be used by maintenance engineers
- Should provide enough information to management to allow them to perform all program management related activities
- Should describe to users how to operate and administer the system
- Specify external system behaviour
- Specify implementation constraints
- Easy to change
- Serve as reference tool for maintenance
- Record forethought about the life cycle of the system i.e. predict changes
- Characterise responses to unexpected events.

2.5.1 PROCESS DOCUMENTATION:

- Used to record and track the development process
 - Planning documentation
 - Cost, Schedule, Funding tracking
 - Schedules
 - Standards
- This documentation is created to allow for successful management of a software product
- Has a relatively short lifespan
 - Only important to internal development process
 - Except in cases where the customer requires a view into this data
- Some items, such as papers that describe design decisions should be extracted and moved into the *product* documentation category when they become implemented
 - Product Documentation
- Describes the delivered product
- Must evolve with the development of the software product
- Two main categories:
 - System Documentation
 - User Documentation

Product Documentation

- System Documentation
 - Describes how the system works, but not how to operate it
- Examples:
 - Requirements Spec
 - Architectural Design
 - Detailed Design
 - Commented Source Code
 - Including output such as JavaDoc
 - Test Plans
 - Including test cases
 - V&V plan and results
 - List of Known Bugs
- User Documentation has two main types
 - End User
 - System Administrator
 - In some cases these are the same people
 - The target audience must be well understood!

- There are five important areas that should be documented for a formal release of a software application
 - These do not necessarily each have to have their own document, but the topics should be covered thoroughly
- Functional Description of the Software
- Installation Instructions
- Introductory Manual
- Reference Manual
- System Administrator's Guide

Document Quality

- Providing thorough and professional documentation is important for any size product development team
 - The problem is that many software professionals lack the writing skills to create professional level documents

Document Structure

- All documents for a given product should have a similar structure
 - A good reason for product standards
- The IEEE Standard for User Documentation lists such a structure
 - It is a superset of what most documents need
- The authors —best practices—are:
 - Put a cover page on all documents
 - Divide documents into chapters with sections and subsections
 - Add an index if there is lots of reference information
 - Add a glossary to define ambiguous terms

2.5.2 STANDARDS:

- Standards play an important role in the development, maintenance and usefulness of documentation
- Standards can act as a basis for quality documentation
 - But are not good enough on their own
Usually define high level content and organization
- There are three types of documentation standards

1.Process Standards

- Define the approach that is to be used when creating the documentation
- Don't actually define any of the content of the documents

2. Product Standards

- Goal is to have all documents created for a specific product attain a consistent structure and appearance
 - Can be based on organizational or contractually required standards
- Four main types:
 - Documentation Identification Standards
 - Document Structure Standards
 - Document Presentation Standards
 - Document Update Standards
- One caveat:
 - Documentation that will be viewed by end users should be created in a way that is best consumed and is most attractive to them

- Internal development documentation generally does not meet this need

3. Interchange Standards

- Deals with the creation of documents in a format that allows others to effectively use
 - PDF may be good for end users who don't need to edit
 - Word may be good for text editing
- Specialized CASE tools need to be considered
- This is usually not a problem within a single organization, but when sharing data between organizations it can occur
 - This same problem is faced all the time during software integration

Other Standards

- IEEE
 - Has a published standard for user documentation
 - Provides a structure and superset of content areas
 - Many organizations probably won't create documents that completely match the standard
- Writing Style
 - Ten —best practices when writing are provided
 - Author proposes that group edits of important documents should occur in a similar fashion to software walkthroughs

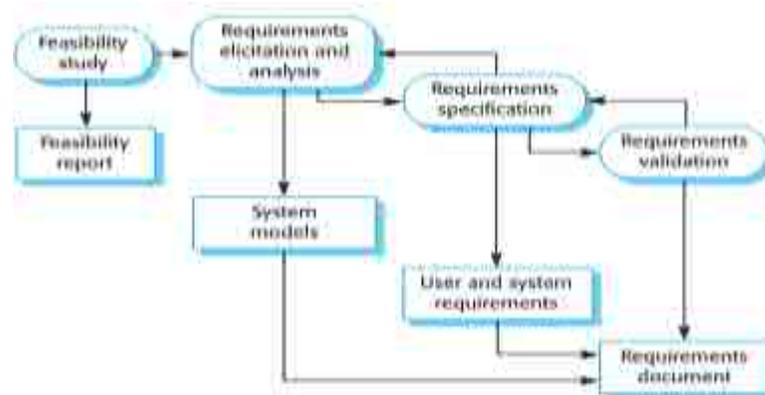
2.6 REQUIREMENT ENGINEERING PROCESS

The processes used for RE vary widely depending on the application domain, the people involved and the organization developing the requirements.

However, there are a number of generic activities common to all processes

- Requirements elicitation
- Requirements analysis
- Requirements validation
- Requirements management . .

The requirements engineering process includes a feasibility study, requirements elicitation and analysis, requirements specification and requirements management



2.7 FEASIBILITY STUDIES

- A feasibility study decides whether or not the proposed system is worthwhile
- A short focused study that checks
 - If the system contributes to organisational objectives
 - If the system can be engineered using current technology and within budget
 - If the system can be integrated with other systems that are used
- Based on information assessment (what is required), information collection and report writing
- Questions for people in the organisation
- What if the system wasn't implemented?
- What are current process problems?
- How will the proposed system help?
- What will be the integration problems?
- Is new technology needed? What skills?
- What facilities must be supported by the proposed system?

2.8 REQUIREMENT ELICITATION AND ANALYSIS:

- Sometimes called requirements elicitation or requirements discovery
- Involves technical staff working with customers to find out about
 - the application domain
 - the services that the system should provide
 - the system's operational constraints
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc.
 - These are called stakeholders

2.8.1 PROBLEMS OF REQUIREMENTS ANALYSIS:

- Stakeholders don't know what they really want
 - Stakeholders express requirements in their own terms
 - Different stakeholders may have conflicting requirements
 - Organisational and political factors may influence the system requirements
 - The requirements change during the analysis process
- New stakeholders may emerge and the business environment change

System models

- Different models may be produced during the requirements analysis activity
- Requirements analysis may involve three structuring activities which result in these different models
 - Partitioning – Identifies the structural (part-of) relationships between entities
 - Abstraction – Identifies generalities among entities
 - Projection – Identifies different ways of looking at a problem
- System models will be covered on January 30

Scenarios

- Scenarios are descriptions of how a system is used in practice
- They are helpful in requirements elicitation as people can relate to these more readily than abstract statement of what they require from a system
- Scenarios are particularly useful for adding detail to an outline requirements description

Ethnography

- A social scientists spends a considerable time observing and analysing how people actually

- People do not have to explain or articulate their work
- Social and organisational factors of importance may be observed
- Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models

2.9 REQUIREMENTS VALIDATION, REQUIREMENTS MANAGEMENT

2.9.1 REQUIREMENTS VALIDATION

- Concerned with demonstrating that the requirements define the system that the customer really wants
- Requirements error costs are high so validation is very important
 - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error
- Requirements checking
 - Validity
 - Consistency
 - Completeness
 - Realism
 - Verifiability

Requirements validation techniques

- Reviews
 - Systematic manual analysis of the requirements
- Prototyping
 - Using an executable model of the system to check requirements.
- Test-case generation
 - Developing tests for requirements to check testability
- Automated consistency analysis
 - Checking the consistency of a structured requirements description

2.9.2 REQUIREMENTS MANAGEMENT

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development
- Requirements are inevitably incomplete and inconsistent
 - New requirements emerge during the process as business needs change and a better understanding of the system is developed
 - Different viewpoints have different requirements and these are often contradictory

Software prototyping

Incomplete versions of the software program being developed. Prototyping can also be used by end users to describe and prove requirements that developers have not considered

Benefits:

The software designer and implementer can obtain feedback from the users early in the project. The client and the contractor can compare if the software made matches the software specification, according to which the software program is built.

It also allows the software engineer some insight into the accuracy of initial project estimates and whether the deadlines and milestones proposed can be successfully met.

Process of prototyping

1. Identify basic requirements

Determine basic requirements including the input and output information desired.

Details, such as security, can typically be ignored.

2. Develop Initial Prototype

The initial prototype is developed that includes only user interfaces. (See Horizontal Prototype, below)

3. Review

The customers, including end-users, examine the prototype and provide feedback on additions or changes.

4. Revise and Enhance the Prototype

Using the feedback both the specifications and the prototype can be improved. Negotiation about what is within the scope of the contract/product may be necessary. If changes are introduced then a repeat of steps #3 and #4 may be needed.

Dimensions of prototypes

1. Horizontal Prototype

It provides a broad view of an entire system or subsystem, focusing on user interaction more than low-level system functionality, such as database access. Horizontal prototypes are useful for:

- Confirmation of user interface requirements and system scope
- Develop preliminary estimates of development time, cost and effort.

2. Vertical Prototypes

A vertical prototype is a more complete elaboration of a single subsystem or function. It is useful for obtaining detailed requirements for a given function, with the following benefits:

- Refinement database design
- Obtain information on data volumes and system interface needs, for network sizing and performance engineering

Types of prototyping

Software prototyping has many variants. However, all the methods are in some way based on two major types of prototyping: Throwaway Prototyping and Evolutionary Prototyping.

1. Throwaway prototyping

Also called close ended prototyping. Throwaway refers to the creation of a model

that will eventually be discarded rather than becoming part of the final delivered software. After preliminary requirements gathering is accomplished, a simple working model of the system is constructed to visually show the users what their requirements may look like when they are implemented into a finished system.

The most obvious reason for using Throwaway Prototyping is that it can be done quickly. If the users can get quick feedback on their requirements, they may be able to refine them early in the development of the software. Making changes early in the development lifecycle is extremely cost effective since there is nothing at that point to redo. If a project is changed after a considerable work has been done then small changes could require large efforts to implement since software systems have many dependencies. Speed is crucial in implementing a throwaway prototype, since with a limited budget of time and money little can be expended on a prototype that will be discarded.

Strength of Throwaway Prototyping is its ability to construct interfaces that the users can test. The user interface is what the user sees as the system, and by seeing it in front of them, it is much easier to grasp how the system will work.

2. Evolutionary prototyping

Evolutionary Prototyping (also known as breadboard prototyping) is quite different from Throwaway Prototyping. The main goal when using Evolutionary Prototyping is to build a very

robust prototype in a structured manner and constantly refine it. "The reason for this is that the Evolutionary prototype, when built, forms the heart of the new system, and the improvements and further requirements will be built.

Evolutionary Prototypes have an advantage over Throwaway Prototypes in that they are functional systems. Although they may not have all the features the users have planned, they may be used on a temporary basis until the final system is delivered.

In Evolutionary Prototyping, developers can focus themselves to develop parts of the system that they understand instead of working on developing a whole system. To minimize risk, the developer does not implement poorly understood features. The partial system is sent to customer sites. As users work with the system, they detect opportunities for new features and give requests for these features to developers. Developers then take these enhancement requests along with their own and use sound configuration-management practices to change the software-requirements specification, update the design, recode and retest.

3. Incremental prototyping

The final product is built as separate prototypes. At the end the separate prototypes are merged in an overall design.

4. Extreme prototyping

Extreme Prototyping as a development process is used especially for developing web applications. Basically, it breaks down web development into three phases, each one based on the preceding one. The first phase is a static prototype that consists mainly of HTML pages. In the second phase, the screens are programmed and fully functional using a simulated services layer. In the third phase the services are implemented. The process is called Extreme Prototyping to draw attention to the second phase of the process, where a fully-functional UI is developed with very little regard to the services other than their contract.

Advantages of prototyping

1. Reduced time and costs: Prototyping can improve the quality of requirements and specifications provided to developers. Because changes cost exponentially more to implement as they are detected later in development, the early determination of what the user really wants can result in faster and less expensive software.

2. Improved and increased user involvement: Prototyping requires user involvement and allows them to see and interact with a prototype allowing them to provide better and more complete feedback and specifications. The presence of the prototype being examined by the user prevents many misunderstandings and miscommunications that occur when each side believe the other understands what they said. Since users know the problem domain better than anyone on the development team does, increased interaction can result in final product that has greater tangible and intangible quality. The final product is more likely to satisfy the users' desire for look, feel and performance.

Disadvantages of prototyping

1. Insufficient analysis: The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into poorly engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable,

which may not be noticed if developers are too focused on building a prototype as a model.

2. User confusion of prototype and finished system: Users can begin to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished. (They are, for example, often unaware of the effort needed to add error - checking and security features which a prototype may not have.) This can lead them to expect the prototype to accurately model the performance of the final system when this is not the intent of the developers. Users can also become attached to features that were included in a prototype for consideration and then removed from the specification for a final system. If users are able to require all proposed features be included in the final system this can lead to conflict.

3. Developer misunderstanding of user objectives: Developers may assume that users share their objectives (e.g. to deliver core functionality on time and within budget), without understanding wider commercial issues. For example, user representatives attending Enterprise software (e.g. PeopleSoft) events may have seen demonstrations of "transaction auditing" (where

changes are logged and displayed in a difference grid view) without being told that this feature demands additional coding and often requires more hardware to handle extra database accesses. Users might believe they can demand auditing on every field, whereas developers might think this is feature creep because they have made assumptions about the extent of user requirements. If the developer has committed delivery before the user requirements were reviewed, developers are between a rock and a hard place, particularly if user management derives some advantage from their failure to implement requirements.

4. Developer attachment to prototype: Developers can also become attached to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture. (This may suggest that throwaway prototyping, rather than evolutionary prototyping, should be used.)

5. Excessive development time of the prototype: A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they very well may try to develop a prototype that is too complex. When the prototype is thrown away the precisely developed requirements that it provides may not yield a sufficient increase in productivity to make up for the time spent developing the prototype. Users can become stuck in debates over details of the prototype, holding up the development team and delaying the final product.

6. Expense of implementing prototyping: the start up costs for building a development team focused on prototyping may be high. Many companies have development methodologies in place, and changing them can mean retraining, retooling, or both. Many companies tend to just jump into the prototyping without bothering to retrain their workers as much as they should.

A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort behind the learning curve. In addition to training for the use of a

prototyping technique, there is an often overlooked need for developing corporate and project

specific underlying structure to support the technology. When this underlying structure is omitted, lower productivity can often result.

Best projects to use prototyping

It has been found that prototyping is very effective in the analysis and design of on-line systems, especially for transaction processing, where the use of screen dialogs is much more in evidence. The greater the interaction between the computer and the user, the greater the benefit is that can be obtained from building a quick system and letting the user play with it.

Systems with little user interaction, such as batch processing or systems that mostly

do calculations, benefit little from prototyping. Sometimes, the coding needed to perform the system functions may be too intensive and the potential gains that prototyping could provide are too small.

Prototyping is especially good for designing good human-computer interfaces. "One of the most productive uses of rapid prototyping to date has been as a tool for iterative user requirements engineering and human-computer interface design.

Methods

There are few formal prototyping methodologies even though most Agile Methods rely heavily upon prototyping techniques.

1. Dynamic systems development method

Dynamic Systems Development Method (DSDM) is a framework for delivering business solutions that relies heavily upon prototyping as a core technique, and is itself ISO 9001 approved. It expands upon most understood definitions of a prototype. According to DSDM the prototype may be a diagram, a business process, or even a system placed into production. DSDM prototypes are intended to be incremental, evolving from simple forms into more comprehensive ones.

DSDM prototypes may be throwaway or evolutionary. Evolutionary prototypes may be evolved horizontally (breadth then depth) or vertically (each section is built in detail with additional iterations detailing subsequent sections). Evolutionary prototypes can eventually evolve into final systems.

The four categories of prototypes as recommended by DSDM are:

- **Business prototypes** – used to design and demonstrate the business processes being automated.
- **Usability prototypes** – used to define, refine, and demonstrate user interface design usability, accessibility, look and feel.
- **Performance and capacity prototypes** - used to define, demonstrate, and predict how systems will perform under peak loads as well as to demonstrate and evaluate other non- functional aspects of the system (transaction rates, data storage volume, response time)
- **Capability/technique prototypes** – used to develop, demonstrate, and evaluate a design approach or concept.

The DSDM lifecycle of a prototype is to:

1. Identify prototype
2. Agree to a plan
3. Create the prototype
4. Review the prototype

2. Operational prototyping

Operational Prototyping was proposed by Alan Davis as a way to integrate throwaway and evolutionary prototyping with conventional system development. "[It] offers the best of both the quick-and-dirty and conventional-development worlds in a sensible manner. Designers develop only well-understood features in building the evolutionary baseline, while using throwaway prototyping to experiment with the poorly understood features." Davis' belief is that to try to "retrofit quality onto a rapid prototype" is not the correct approach when trying to combine the two approaches. His idea is to engage in an evolutionary prototyping methodology and rapidly prototype the features of the system after each evolution.

The specific methodology follows these steps:

- An evolutionary prototype is constructed and made into a baseline using conventional development strategies, specifying and implementing only the requirements that are well understood.
- Copies of the baseline are sent to multiple customer sites along with a trained prototyper.
 - At each site, the prototyper watches the user at the system.
 - Whenever the user encounters a problem or thinks of a new feature or requirement, the prototyper logs it. This frees the user from having to record the problem, and allows them to continue working.
 - After the user session is over, the prototyper constructs a throwaway prototype on top of the baseline system.
 - The user now uses the new system and evaluates. If the new changes aren't effective, the prototyper removes them.
 - If the user likes the changes, the prototyper writes feature-enhancement requests and forwards them to the development team.
 - The development team, with the change requests in hand from all the sites, then produce a new evolutionary prototype using conventional methods.

Obviously, a key to this method is to have well trained prototypers available to go to the user sites. The Operational Prototyping methodology has many benefits in systems that are complex and have few known requirements in advance.

3. Evolutionary systems development

Evolutionary Systems Development is a class of methodologies that attempt to formally implement Evolutionary Prototyping. One particular type, called Systems craft is described by John Crinnion in his book: Evolutionary Systems Development.

Systemscraft was designed as a 'prototype' methodology that should be modified and adapted to fit the specific environment in which it was implemented. Systemscraft was not designed as a rigid 'cookbook' approach to the development process. It is now generally recognised[sic] that a good methodology should be flexible enough to be adjustable to suit all kinds of environment and situation... The basis of Systemscraft, not unlike Evolutionary Prototyping, is to create a working system from the initial requirements and build upon it in a series of revisions. Systemscraft places heavy emphasis on traditional analysis being used throughout the development of the system.

4. Evolutionary rapid development

Evolutionary Rapid Development (ERD) was developed by the Software Productivity Consortium, a technology development and integration agent for the Information Technology Office of the Defense Advanced Research Projects Agency (DARPA).

Fundamental to ERD is the concept of composing software systems based on the reuse of components, the use of software templates and on an architectural template. Continuous evolution of system capabilities in rapid response to changing user needs and technology is highlighted by the evolvable architecture, representing a class of solutions. The process focuses on the use of small artisan-based teams integrating software and systems engineering disciplines working multiple, often parallel short-duration timeboxes with frequent customer interaction.

Key to the success of the ERD-based projects is parallel exploratory analysis and development of features, infrastructures, and components with and adoption of leading edge technologies enabling the quick reaction to changes in technologies, the marketplace, or customer requirements.

To elicit customer/user input, frequent scheduled and ad hoc/impromptu meetings with the demonstrations of system capabilities are held to solicit feedback

before design/implementation decisions are solidified. Frequent releases (e.g., betas) are made available for use to provide insight into how the system could better support user and customer needs. This assures that the system evolves to satisfy existing user needs.

The design framework for the system is based on using existing published or de facto standards. The system is organized to allow for evolving a set of capabilities that includes considerations for performance, capacities, and functionality. The architecture is defined in terms of abstract interfaces that encapsulate the services and their implementation (e.g., COTS applications). The architecture serves as a template to be used for guiding development of more than a single instance of the system. It allows for multiple application components to be used to implement the services. A core set of functionality not likely to change is also identified and established.

The ERD process is structured to use demonstrated functionality rather than paper products as a way for stakeholders to communicate their needs and expectations. Central to this goal of rapid delivery is the use of the "time box" method. Timeboxes are fixed periods of time in which specific tasks (e.g., developing a set of functionality) must be performed. Rather than allowing time to expand to satisfy some vague set of goals, the time is fixed (both in terms of calendar weeks and person-hours) and a set of goals is defined that realistically can be achieved within these constraints. To keep development from degenerating into a "random walk," long-range plans are defined to guide the iterations. These plans provide a vision for the overall system and set boundaries (e.g., constraints) for the project. Each iteration within the process is conducted in the context of these long-range plans.

Once architecture is established, software is integrated and tested on a daily basis. This allows the team to assess progress objectively and identify potential problems quickly. Since small amounts of the system are integrated at one time, diagnosing and removing the defect is rapid. User demonstrations can be held at short notice since the system is generally ready to exercise at all times.

5. Scrum

Scrum is an agile method for project management. The approach was first described by Takeuchi and Nonaka in "The New New Product Development Game" (Harvard Business Review, Jan-Feb 1986).

Tools

Efficiently using prototyping requires that an organization have proper tools and a staff trained to use those tools. Tools used in prototyping can vary from individual tools like 4th generation programming languages used for rapid prototyping to complex integrated CASE tools. 4th generation programming languages like Visual Basic and ColdFusion are frequently used since they are cheap, well known and relatively easy and fast to use. CASE tools are often developed or selected by the military or large organizations. Users may prototype elements of an application themselves in a spreadsheet.

1. Screen generators, design tools & Software Factories

Commonly used screen generating programs that enable prototypers to show users systems that don't function, but show what the screens may look like. Developing Human Computer Interfaces can sometimes be the critical part of the development effort, since to the users the interface essentially is the system.

Software Factories are Code Generators that allow you to model the domain model and then drag and drop the UI. Also they enable you to run the prototype and use basic database functionality. This approach allows you to explore the domain model and make sure it is in sync with the GUI prototype.

2. Application definition or simulation software

It enables users to rapidly build lightweight, animated simulations of another computer program, without writing code. Application simulation software allows both technical and non-technical users to experience, test, collaborate and validate the simulated program, and provides reports such as annotations, screenshot and schematics. To simulate applications one can also use software which simulate real-world software programs for computer based training, demonstration, and customer support, such as screen casting software as those areas are closely related.

3. Sketchflow

Sketch Flow, a feature of Microsoft Expression Studio Ultimate, gives the ability to quickly and effectively map out and iterate the flow of an application UI, the layout of individual screens and transition from one application state to another.

- Interactive Visual Tool
- Easy to learn
- Dynamic
- Provides environment to collect feedback

4. Visual Basic

One of the most popular tools for Rapid Prototyping is Visual Basic (VB). Microsoft Access, which includes a Visual Basic extensibility module, is also a widely accepted prototyping tool that is used by many non-technical business analysts. Although VB is a programming language it has many features that facilitate using it to create prototypes, including:

- An interactive/visual user interface design tool.
- Easy connection of user interface components to underlying functional behavior.
- Modifications to the resulting software are easy to perform.

5. Requirements Engineering Environment

It provides an integrated toolset for rapidly representing, building, and executing models of critical aspects of complex systems.

It is currently used by the Air Force to develop systems. It is: an integrated set of tools that allows systems analysts to rapidly build functional, user interface, and performance prototype models of system components. These modeling activities are performed to gain a greater understanding of complex systems and lessen the impact that inaccurate requirement specifications have on cost and scheduling during the system development process.

REE is composed of three parts. The first, called proto is a CASE tool specifically designed to support rapid prototyping. The second part is called the Rapid Interface Prototyping System or RIP, which is a collection of tools that facilitate the creation of user interfaces. The third part of REE is a user interface to RIP and proto that is graphical and intended to be easy to use.

Rome Laboratory, the developer of REE, intended that to support their internal requirements gathering methodology. Their method has three main parts:

- Elicitation from various sources which means u loose (users, interfaces to other systems), specification, and consistency checking
- Analysis that the needs of diverse users taken together do not conflict and are

- technically and economically feasible
- Validation that requirements so derived are an accurate reflection of user needs.

6.LYM

B

LYMB is an object-oriented development environment aimed at developing applications that require combining graphics-based user interfaces, visualization, and rapid prototyping.

7. Non-relational environments

Non-relational definition of data (e.g. using Cache or associative models can help make end-user prototyping more productive by delaying or avoiding the need to normalize data at every iteration of a simulation. This may yield earlier/greater clarity of business requirements, though it does not specifically confirm that requirements are technically and economically feasible in the target production system.

8.

PSDL

PSDL is a prototype description language to describe real-time software.

2.10 CLASSICAL ANALYSIS: STRUCTURED SYSTEM ANALYSIS:

2.10.1 CLASSICAL ANALYSIS:

Data Model

- Used to describe the logical structure of data processed by the system
- Entity-relation-attribute model sets out the entities in the system, the relationships between these entities and the entity attributes
- Widely used in database design. Can readily be implemented using relational databases
- No specific notation provided in the UML but objects and associations can be used

Behavioural Model

- Behavioural models are used to describe the overall behaviour of a system
- Two types of behavioural model are shown here
 - Data processing models that show how data is processed as it moves through the system
 - State machine models that show the systems response to events
- Both of these models are required for a description of the system's behaviour

1. Data-processing models

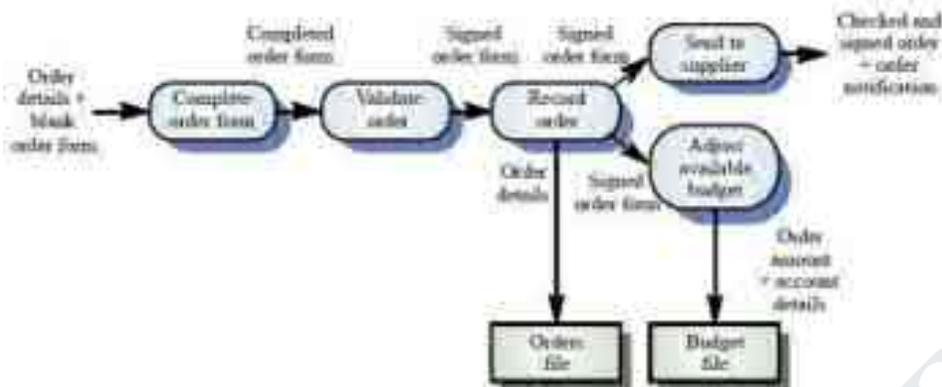
- Data flow diagrams are used to model the system's data processing
- These show the processing steps as data flows through a system
- Intrinsic part of many analysis methods
- Simple and intuitive notation that customers can understand
- Show end-to-end processing of data

Data flow diagrams

- DFDs model the system from a functional perspective
- Tracking and documenting how the data associated with a process is helpful to develop an overall understanding of the system

- Data flow diagrams may also be used in showing the data exchange between a system and other systems in its environment

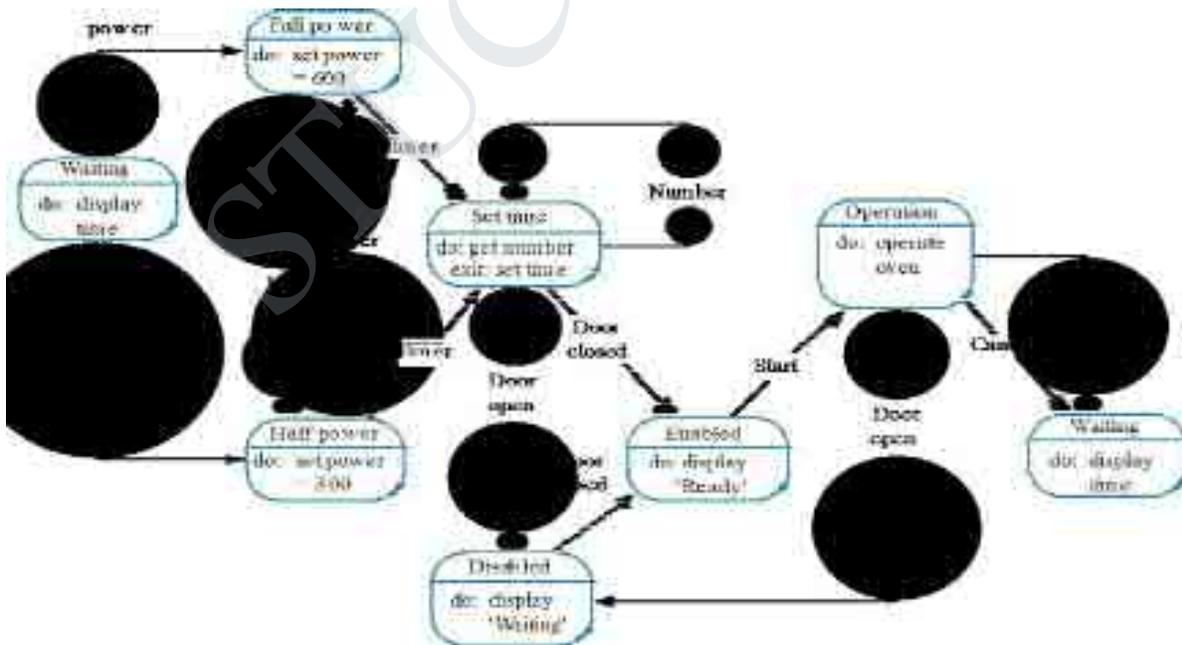
Order processing DFD



2. State machine models

- These model the behaviour of the system in response to external and internal events
- They show the system's responses to stimuli so are often used for modelling real-time systems
- State machine models show system states as nodes and events as arcs between these nodes.
- When an event occurs, the system moves from one state to another
- Statecharts are an integral part of the UML

Microwave oven model



Statecharts

- Allow the decomposition of a model into submodels
- A brief description of the actions is included following the `_do` in each state
- Can be complemented by tables describing the states and the stimuli

2.10.2 STRUCTURED SYSTEM ANALYSIS:

- The data-flow approach is typified by the Structured Analysis method (SA)
- Two major strategies dominate structured analysis
 - `_Old` method popularised by DeMarco
 - `_Modern` approach by Yourdon

2.10.2.1 DeMarco

- A top-down approach
 - The analyst maps the current physical system onto the current logical data-flow model
- The approach can be summarised in four steps:
 - Analysis of current physical system
 - Derivation of logical model
 - Derivation of proposed logical model
 - Implementation of new physical system

2.10.2.2 Modern structured analysis

- Distinguishes between user's real needs and those requirements that represent the external behaviour satisfying those needs
- Includes real-time extensions
- Other structured analysis approaches include:
 - Structured Analysis and Design Technique (SADT)
 - Structured Systems Analysis and Design Methodology (SSADM)

Method weaknesses

- They do not model non-functional system requirements.
- They do not usually include information about whether a method is appropriate for a given problem.
- They may produce too much documentation.
- The system models are sometimes too detailed and difficult for users to understand.

CASE workbenches

- A coherent set of tools that is designed to support related software process activities such as analysis, design or testing.
- Analysis and design workbenches support system modelling during both requirements engineering and system design.
- These workbenches may support a specific design method or may provide support for a creating several different types of system model.
 - Diagram editors
 - Model analysis and checking tools
 - Repository and associated query language

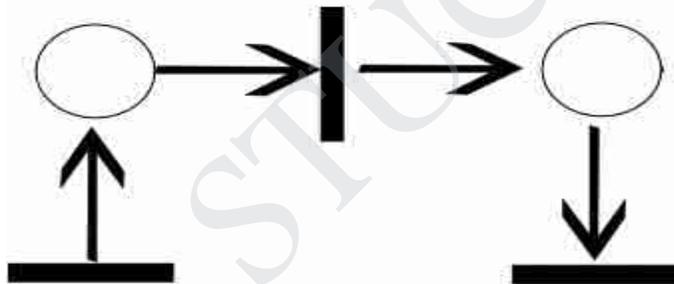
- Data dictionary
- Report definition and generation tools
- Forms definition tools
- Import/export translators
- Code generation tools

2.11 PETRI NETS- DATA DICTIONARY:

2.11.1 PETRI NETS:

A Petri-net-based formal specification method for distributed systems is accommodated to software process modeling. The authors introduce domain-specific concepts, stressing the distributedness and dynamic nature of software processes. Development states are viewed as distributed entities. Development activities are characterized by their effects on software objects, pertinent information exchange with human or technical carriers of such activities, and local changes to development states. These dynamic aspects of software processes are visualized by labeled Petri nets. Sketched are structuring mechanisms that support hierarchical decomposition and systematic combinations of separate views of a software engineering process.

A **Petri net** (also known as a **place/transition net** or **P/T net**) is one of several mathematical modeling languages for the description of distributed systems. A Petri net is a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur, signified by bars) and places (i.e. conditions, signified by circles). The directed arcs describe which places are pre- and/or postconditions for which transitions (signified by arrows).



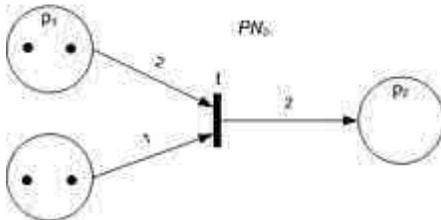
Formal definition and basic terminology

Petri nets are [state-transition systems](#) that extend a class of nets called elementary nets.^[2]

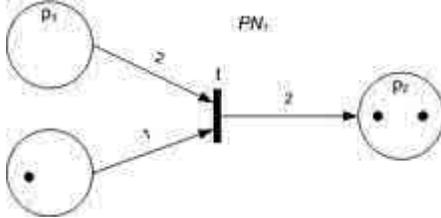
Definition 1. A net is a triple $N = (P, T, F)$ where:

1. P and T are *disjoint* finite sets of *places* and *transitions*, respectively.
2. $F \subset (P \times T) \cup (T \times P)$ is a set of *arcs* (or flow relations).

Definition 2. Given a net $N = (P, T, F)$, a *configuration* is a set C so that $C \subseteq P$.



A Petri net with an enabled transition.



The Petri net that follows after the transition fires (Initial Petri net in the figure above).

Definition 3. An *elementary net* is a net of the form $EN = (N, C)$ where:

1. $N = (P, T, F)$ is a net.
2. C is such that $C \subseteq P$ is a *configuration*.

Definition 4. A *Petri net* is a net of the form $PN = (N, M, W)$, which extends the elementary net so that:

1. $N = (P, T, F)$ is a net.
2. $M: P \rightarrow Z$ is a place [multiset](#), where Z is a countable set. M extends the concept of *configuration* and is commonly described with reference to Petri net diagrams as a *marking*.
3. $W: F \rightarrow Z$ is an arc [multiset](#), so that the count (or weight) for each arc is a measure of the arc *multiplicity*.

2.11.2 DATA DICTIONARY:

- Data dictionaries are lists of all of the names used in the system models. Descriptions of the entities, relationships and attributes are also included
- Advantages
 - Support name management and avoid duplication
 - Store of organisational knowledge linking analysis, design and implementation
- Many CASE workbenches support data dictionaries

2.11.2.1 Data dictionary entries

Name	Description	Type	Date
has-labels	1:N relation between entities of type Node or Link and entities of type Label.	Relation	5.10.1998
Label	Holds structured or unstructured information about nodes or links. Labels are represented by an icon (which can be a transparent box) and associated text.	Entity	8.12.1998
Link	A 1:1 relation between design entities represented as nodes. Links are typed and may be named.	Relation	8.12.1998
name (label)	Each label has a name which identifies the type of label. The name must be unique within the set of label types used in a design.	Attribute	8.12.1998
name (node)	Each node has a name which must be unique within a design. The name may be up to 64 characters long.	Attribute	15.11.1998

—
—

UNIT III

SOFTWARE DESIGN

3.1 DESIGN PROCESS – DESIGN

CONCEPTS:

3.1.1 DESIGN CONCEPTS AND PRINCIPLES:

- Map the information from the analysis model to the design representations - data design, architectural design, interface design, procedural design.

Software Engineering:

MODULAR DESIGN:

Modular Design Methodology Evaluation – 1:

Modularity

- the degree to which software can be understood by examining its components independently of one another
- Modular decomposability
 - provides systematic means for breaking problem into sub problems
- Modular compos ability
 - supports reuse of existing modules in new systems
- Modular understandability
 - module can be understood as a stand-alone unit

Modular Design Methodology Evaluation – 2:

- Modular continuity
 - module change side-effects minimized
- Modular protection
 - processing error side-effects minimized

Effective Modular Design:

- Functional independence
 - modules have high cohesion and low coupling
- Cohesion
 - qualitative indication of the degree to which a module focuses on just one thing
- Coupling
 - qualitative indication of the degree to which a module is connected to other modules and to the outside world

3.2 DESIGN MODEL– DESIGN HEURISTIC:

Design Models – 1 :

- **Data Design**
 - created by transforming the data dictionary and ERD into implementation data structures
 - requires as much attention as algorithm design
- **Architectural Design**
 - derived from the analysis model and the subsystem interactions defined in the DFD
- **Interface Design**
 - derived from DFD and CFD
 - describes software elements communication with
 - other software elements
 - other systems
 - human users

Design Models – 2 :

- Procedure-level design
 - created by transforming the structural elements defined by the software architecture into procedural descriptions of software components
 - Derived from information in the PSPEC, CSPEC, and STD

Design Principles – 1:

- Process should not suffer from tunnel vision – consider alternative approaches
- Design should be traceable to analysis model
- Do not try to reinvent the wheel
- use design patterns ie reusable components
- Design should exhibit both uniformity and integration
- Should be structured to accommodate changes

Design Principles – 2 :

- Design is not coding and coding is not design
- Should be structured to degrade gently, when bad data, events, or operating conditions are encountered
- Needs to be assessed for quality as it is being created
- Needs to be reviewed to minimize conceptual (semantic) errors

Design Concepts -1 :

- Abstraction
 - allows designers to focus on solving a problem without being concerned about irrelevant lower level details

Procedural abstraction is a named sequence of instructions that has a specific and limited function

e.g open a door

Open implies a long sequence of procedural steps

data abstraction is collection of data that describes a data object e.g door type, opening mech, weight, dimen

Design Concepts -2 :

- Design Patterns
 - description of a design structure that solves a particular design problem within a specific context and its impact when applied

Design Concepts -3

- Software Architecture
 - overall structure of the software components and the ways in which that structure
 - provides conceptual integrity for a system

Design Concepts -4 :

- Information Hiding
 - information (data and procedure) contained within a module is inaccessible to modules that have no need for such information
- Functional Independence
 - achieved by developing modules with single-minded purpose and an aversion to excessive interaction with other models

Refactoring – Design concepts :

- Fowler [FOW99] defines refactoring in the following manner:
 - "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.¶

- When software is refectories, the existing design is examined for
 - redundancy
 - unused design elements
 - inefficient or unnecessary algorithms
 - poorly constructed or inappropriate data structures
 - or any other design failure that can be corrected to yield a better design.

Design Concepts – 4 :

- Objects
 - encapsulate both data and data manipulation procedures needed to describe the content and behavior of a real world entity
- Class
 - generalized description (template or pattern) that describes a collection of similar objects
- Inheritance
 - provides a means for allowing subclasses to reuse existing superclass data and procedures; also provides mechanism for propagating changes

Design Concepts – 5:

- Messages
 - the means by which objects exchange information with one another
- Polymorphism
 - a mechanism that allows several objects in a class hierarchy to have different methods with the same name
 - instances of each subclass will be free to respond to messages by calling

their own version of the method

3.3 ARCHITECTURAL DESIGN:

Why Architecture?

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

- (1) analyze the effectiveness of the design in meeting its stated requirements,
- (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and
- (3) reduce the risks associated with the construction of the software.

Importance :

- Software architecture representations enable communications among stakeholders
- Architecture highlights early design decisions that will have a profound impact on the ultimate success of the system as an operational entity
- The architecture constitutes an intellectually graspable model of how the system is structured and how its components work together

3.4 ARCHITECTURAL STYLES: ARCHITECTURAL DESIGN:

3.4.1 ARCHITECTURAL STYLES:

Architectural Styles – 1:

- Data centered
 - file or database lies at the center of this architecture and is accessed frequently by other components that modify data

Architectural Styles – 2:

- Data flow
 - input data is transformed by a series of computational components into output data

- Pipe and filter pattern has a set of components called filters, connected by pipes that transmit data from one component to the next.
- If the data flow degenerates into a single line of transforms, it is termed batch sequential
- Object-oriented
 - components of system encapsulate data and operations, communication between components is by message passing
- Layered
 - several layers are defined
 - each layer performs operations that become closer to the machine instruction set in the lower layer.

Architectural Styles – 3:

Call and return

- program structure decomposes function into control hierarchy with main program invoking several subprograms

Software Architecture Design – 1:

- Software to be developed must be put into context
 - model external entities and define interfaces
- Identify architectural archetypes
 - collection of abstractions that must be modeled if the system is to be constructed

Object oriented Architecture :

- The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing

Software Architecture Design – 2:

- Specify structure of the system
 - define and refine the software components needed to implement each archet ype
- Continue the process iteratively until a complete architectural structure has been derived

Layered Architecture:

- Number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set
- At the outer layer –components service user interface operations.
- At the inner layer – components perform operating system interfacing.
- Intermediate layers provide utility services and application software function

Architecture Tradeoff Analysis – 1:

1. Collect scenarios
2. Elicit requirements, constraints, and environmental description
3. Describe architectural styles/patterns chosen to address scenarios and requirements
 - module view
 - process view
 - data flow view

Architecture Tradeoff Analysis – 2:

4. Evaluate quality attributes independently (e.g. reliability, performance, security, maintainability, flexibility, testability, portability, reusability, interoperability)
5. Identify sensitivity points for architecture
 - any attributes significantly affected by changing in the architecture

Refining Architectural Design:

- Processing narrative developed for each module

- Interface description provided for each module
- Local and global data structures are defined
- Design restrictions/limitations noted
- Design reviews conducted
- Refinement considered if required and justified

3.4.2 ARCHITECTURAL DESIGN

- An early stage of the system design process.
- Represents the link between specification and design processes.
- Often carried out in parallel with some specification activities.
- It involves identifying major system components and their communications.

Advantages of explicit architecture

- Stakeholder communication
 - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
 - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
 - The architecture may be reusable across a range of systems.

Architecture and system characteristics

- Performance
 - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- Security
 - Use a layered architecture with critical assets in the inner layers.
- Safety
 - Localise safety-critical features in a small number of sub-systems.
- Availability
 - Include redundant components and mechanisms for fault tolerance.
- Maintainability
 - Use fine-grain, replaceable components.

Architectural conflicts

- Using large-grain components improves performance but reduces maintainability.
- Introducing redundant data improves availability but makes security more difficult.
- Localising safety-related features usually means more communication so degraded performance.

System structuring

- Concerned with decomposing the system into interacting sub-systems.
- The architectural design is normally expressed as a block diagram presenting an overview of the system structure.
- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed.

3.5 ARCHITECTURAL MAPPING USING DATA FLOW:

3.5.1 PACKING ROBOT CONTROL SYSTEM:

Box and line diagrams

- Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.

- However, useful for communication with stakeholders and for project planning.

Architectural design decisions

- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes.
- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- What control strategy should be used?
- How will the architectural design be evaluated?
- How should the architecture be documented?

Architecture reuse

- Systems in the same domain often have similar architectures that reflect domain concepts.
- Application product lines are built around a core architecture with variants that satisfy

particular customer requirements.

Architectural styles

- The architectural model of a system may conform to a generic architectural model or style.
- An awareness of these styles can simplify the problem of defining system architectures.
- However, most large systems are heterogeneous and do not follow a single architectural style.

Architectural models

- Used to document an architectural design.
- Static structural model that shows the major system components.
- Dynamic process model that shows the process structure of the system.
- Interface model that defines sub-system interfaces.
- Relationships model such as a data-flow model that shows sub-system relationships.
- Distribution model that shows how sub-systems are distributed across computers.

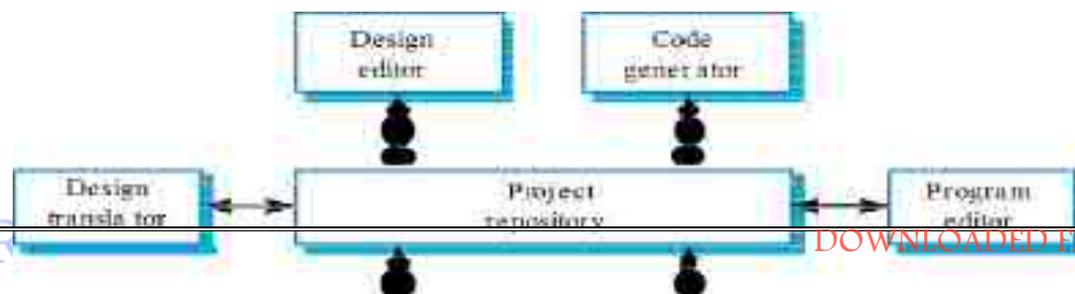
System organisation

- Reflects the basic strategy that is used to structure a system.
- Three organisational styles are widely used:
 - A shared data repository style;
 - A shared services and servers style;
 - An abstract machine or layered style.

3.5.2 THE REPOSITORY MODEL:

- Sub-systems must exchange data. This may be done in two ways:
 - Shared data is held in a central database or repository and may be accessed by all sub-systems;
- Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used.

CASE toolset architecture



Repository model characteristics**Advantages**

- Efficient way to share large amounts of data;
- Sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc.
- Sharing model is published as the repository schema.

Disadvantages:

Sub-systems must agree on a repository data model. Inevitably a compromise;

- Data evolution is difficult and expensive;
- No scope for specific management policies;
- Difficult to distribute efficiently.

3.5.3 CLIENT-SERVER MODEL:

- Distributed system model which shows how data and processing is distributed across a range of components.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

Client-server characteristics**Advantages**

- Distribution of data is straightforward;
- Makes effective use of networked systems. May require cheaper hardware;
- Easy to add new servers or upgrade existing

servers. Disadvantages

- No shared data model so sub-systems use different data organisation. Data interchange may be inefficient;
- Redundant management in each server;
- No central register of names and services - it may be hard to find out what servers and services are available.

Abstract machine (layered) model

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.

- However, often artificial to structure systems in this way.

Modular decomposition styles

- Styles of decomposing sub-systems into modules.
- No rigid distinction between system organisation and modular decomposition.

Sub-systems and modules

- A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
- A module is a system component that provides services to other components but would not normally be considered as a separate system.
- Modular decomposition
- Another structural level where sub-systems are decomposed into modules.
- Two modular decomposition models covered
 - An object model where the system is decomposed into interacting object;
 - A pipeline or data-flow model where the system is decomposed into functional modules which transform inputs to outputs.
- If possible, decisions about concurrency should be delayed until modules are implemented.

Object models

- Structure the system into a set of loosely coupled objects with well-defined interfaces.
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations.
- When implemented, objects are created from these classes and some control model used to coordinate object operations.

Invoice processing system

Object model advantages

- Objects are loosely coupled so their implementation can be modified without affecting other objects.
- The objects may reflect real-world entities.
- OO implementation languages are widely used.
- However, object interface changes may cause problems and complex entities may be hard to represent as objects.

Function-oriented pipelining

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

3.6 USER INTERFACE DESIGN:

- Designing effective interfaces for software systems
- System users often judge a system by its interface rather than its functionality
- A poorly designed interface can cause a user to make catastrophic errors
- Poor user interface design is the reason why so many software systems are never used
- Most users of business systems interact with these systems through graphical user interfaces (GUIs)
- In some cases, legacy text-based interfaces are still used

User interface design process

consistency

- The system should display an appropriate level of consistency

- Commands and menus should have the same format, command punctuation should be similar, etc.
- Minimal surprise
 - If a command operates in a known way, the user should be able to predict the operation of comparable commands
- Recoverability
 - The system should provide some interface to user errors and allow the user to recover from errors
- User guidance
 - Some user guidance such as help systems, on-line manuals, etc. should be supplied
- User diversity
 - Interaction facilities for different types of user should be supported
 - E.g., some users have seeing difficulties and so larger text should be available

User-system interaction

- Two problems must be addressed in interactive systems design
 - How should information from the user be provided to the computer system?
 - How should information from the computer system be presented to the user?

Interaction styles

- Direct manipulation
 - Easiest to grasp with immediate feedback
 - Difficult to program
- Menu selection
 - User effort and errors minimized
 - Large numbers and combinations of choices a problem
- Form fill-in
 - Ease of use, simple data entry
 - Tedious, takes a lot of screen space
- Natural language
 - Great for casual users
 - Tedious for expert users

3.7 INTERFACE ANALYSIS:

Information presentation

- Information presentation is concerned with presenting system information to system users
- The information may be presented directly or may be transformed in some way for presentation
- The Model-View-Controller approach is a way of supporting multiple presentations of data
- Visualisation can reveal relationships between entities and trends in the data
- Possible data visualisations are:
 - Weather information
 - State of a telephone network
 - Chemical plant pressures and temperatures
 - A model of a molecule

Colour displays

- Colour adds an extra dimension to an interface and can help the user understand complex information structures
- Can be used to highlight exceptional events

- The use of colour to communicate meaning

Error messages

- Error message design is critically important. Poor error messages can mean that a user rejects rather than accepts a system
- Messages should be polite, concise, consistent and constructive
- The background and experience of users should be the determining factor in message design

3.8 INTERFACE DESIGN:

User interface evaluation

- Some evaluation of a user interface design should be carried out to assess its suitability
- Full scale evaluation is very expensive and impractical for most systems
- Ideally, an interface should be evaluated against req
- However, it is rare for such specifications to be produced

Real Time Software Design

- Systems which monitor and control their environment
- Inevitably associated with hardware devices
 - Sensors: Collect data from the system environment
 - Actuators: Change (in some way) the system's environment
- Time is critical. Real-time systems MUST respond within specified times
- A real-time system is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced
- A ‘soft’ real-time system is a system whose operation is degraded if results are not produced according to the specified timing requirements
- A ‘hard’ real-time system is a system whose operation is incorrect if results are not produced according to the timing specification

Stimulus/Response Systems

- Given a stimulus, the system must produce a response within a specified time
- 2 classes
- Periodic stimuli. Stimuli which occur at predictable time intervals
 - For example, a temperature sensor may be polled 10 times per second
- Aperiodic stimuli. Stimuli which occur at unpredictable times
 - For example, a system power failure may trigger an interrupt which must be processed by the system

Architectural considerations

- Because of the need to respond to timing demands made by different stimuli / responses, the system architecture must allow for fast switching between stimulus handlers
- Timing demands of different stimuli are different so a simple sequential loop is not usually adequate

Real –Time Software Design:

- Designing embedded software systems whose behaviour is subject to timing constraints
- To explain the concept of a real-time system and why these systems are usually implemented as concurrent processes
- To describe a design process for real-time systems

- To explain the role of a real-time executive
- To introduce generic architectures for monitoring and control and data acquisition systems

Real-time systems:

- Systems which monitor and control their environment
- Inevitably associated with hardware devices
 - Sensors: Collect data from the system environment
 - Actuators: Change (in some way) the system's environment
- Time is critical. Real-time systems MUST respond within specified times

Definition:

- A real-time system is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced
- A soft real-time system is a system whose operation is degraded if results are not produced according to the specified timing requirements
- A hard real-time system is a system whose operation is incorrect if results are not produced according to the timing specification

Stimulus/Response Systems:

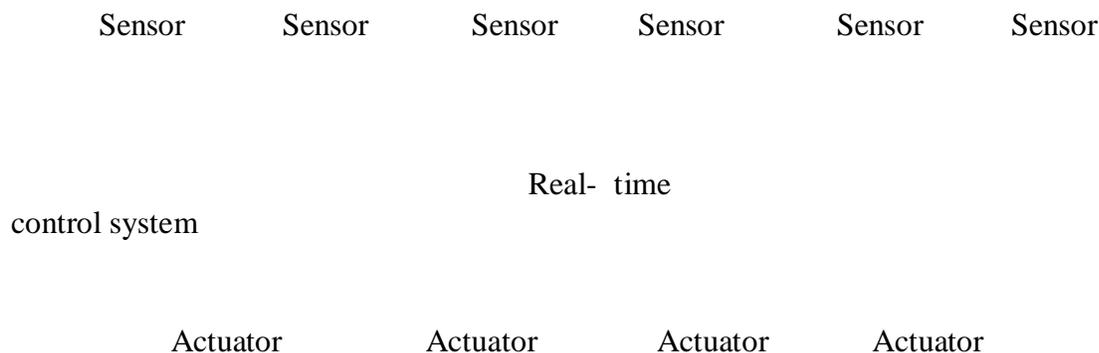
- Given a stimulus, the system must produce a response within a specified time
- Periodic stimuli. Stimuli which occur at predictable time intervals
 - For example, a temperature sensor may be polled 10 times per second
- Aperiodic stimuli. Stimuli which occur at unpredictable times
 - For example, a system power failure may trigger an interrupt which must be processed by the system

Architectural considerations:

- Because of the need to respond to timing demands made by different stimuli/responses, the system architecture must allow for fast switching between stimulus handlers
- Timing demands of different stimuli are different so a simple sequential loop is not usually adequate
- Real-time systems are usually designed as cooperating processes with a real-time executive controlling these processes

3.9 COMPONENT LEVEL DESIGN: DESIGNING CLASS BASED COMPONENTS :

3.9.1 A real-time system model:



System elements:

- Sensors control processes
 - Collect information from sensors. May buffer information collected in response to a sensor stimulus
- Data processor
 - Carries out processing of collected information and computes the system response
- Actuator control
 - Generates control signals for the actuator

R-T systems design process:

- Identify the stimuli to be processed and the required responses to these stimuli
- For each stimulus and response, identify the timing constraints
- Aggregate the stimulus and response processing into concurrent processes. A process may be associated with each class of stimulus and response
- Design algorithms to process each class of stimulus and response. These must meet the given timing requirements
- Design a scheduling system which will ensure that processes are started in time to meet their deadlines
- Integrate using a real-time executive or operating system

Timing constraints:

- May require extensive simulation and experiment to ensure that these are met by the system
- May mean that certain design strategies such as object-oriented design cannot be used because of the additional overhead involved
- May mean that low-level programming language features have to be used for performance reasons

Real-time programming:

- Hard-real time systems may have to be programmed in assembly language to ensure that deadlines are met
- Languages such as C allow efficient programs to be written but do not have constructs to support concurrency or shared resource management
- Ada as a language designed to support real-time systems design so includes a general purpose concurrency mechanism

Non-stop system components:

- Configuration manager
 - Responsible for the dynamic reconfiguration of the system software and hardware. Hardware modules may be replaced and software upgraded without stopping the systems
- Fault manager
 - Responsible for detecting software and hardware faults and taking appropriate actions (e.g. switching to backup disks) to ensure that the system continues in operation

Burglar alarm system e.g

- A system is required to monitor sensors on doors and windows to detect the presence of intruders in a building
- When a sensor indicates a break-in, the system switches on lights around the area and calls police automatically

- The system should include provision for operation without a mains power supply
- Sensors
 - Movement detectors, window sensors, door sensors.
 - 50 window sensors, 30 door sensors and 200 movement detectors
 - Voltage drop sensor
- Actions
 - When an intruder is detected, police are called automatically.
 - Lights are switched on in rooms with active sensors.
 - An audible alarm is switched on.
 - The system switches automatically to backup power when a voltage drop is detected.

The R-T system design process:

- Identify stimuli and associated responses
- Define the timing constraints associated with each stimulus and response
- Allocate system functions to concurrent processes
- Design algorithms for stimulus processing and response generation
- Design a scheduling system which ensures that processes will always be scheduled to meet their deadlines

Control systems:

- A burglar alarm system is primarily a monitoring system. It collects data from sensors but no real-time actuator control
- Control systems are similar but, in response to sensor values, the system sends control signals to actuators
- An example of a monitoring and control system is a system which monitors temperature and switches heaters on and off

Data acquisition systems:

- Collect data from sensors for subsequent processing and analysis.
- Data collection processes and processing processes may have different periods and deadlines.
- Data collection may be faster than processing e.g. collecting information about an explosion.
- Circular or ring buffers are a mechanism for smoothing speed differences.
- Producer and consumer processes must be mutually excluded from accessing the same element.

The buffer must stop producer processes adding information to a full buffer and consumer processes trying to take information from an empty buffer

System Design

- Design both the hardware and the software associated with system. Partition functions to either hardware or software
- Design decisions should be made on the basis on non-functional system requirements
- Hardware delivers better performance but potentially longer development and less scope for change

System elements

- Sensors control processes
 - Collect information from sensors. May buffer information collected in response to a sensor stimulus
- Data processor
 - Carries out processing of collected information and computes the system response

- Actuator control
 - Generates control signals for the actuator

R-T systems design process

- Identify the stimuli to be processed and the required responses to these stimuli
- For each stimulus and response, identify the timing constraints
- Aggregate the stimulus and response processing into concurrent processes. A process may be associated with each class of stimulus and response
- Design algorithms to process each class of stimulus and response. These must meet the given timing requirements
- Design a scheduling system which will ensure that processes are started in time to meet their deadlines
- Integrate using a real-time executive or operating system

Timing constraints

- For aperiodic stimuli, designers make assumptions about probability of occurrence of stimuli.
- May mean that certain design strategies such as object-oriented design cannot be used because of the additional overhead involved

State machine modelling

- The effect of a stimulus in a real-time system may trigger a transition from one state to another.
- Finite state machines can be used for modelling real-time systems.
- However, FSM models lack structure. Even simple systems can have a complex model.
- The UML includes notations for defining state machine models

Real-time programming

- Hard-real time systems may have to be programmed in assembly language to ensure that deadlines are met
- Languages such as C allow efficient programs to be written but do not have constructs to support concurrency or shared resource management
- Ada as a language designed to support real-time systems design so includes a general purpose concurrency mechanism

Java as a real-time language

- Java supports lightweight concurrency (threads and synchronized methods) and can be used for some soft real-time systems
- Java 2.0 is not suitable for hard RT programming or programming where precise control of timing is required
 - Not possible to specify thread execution time
 - Uncontrollable garbage collection
 - Not possible to discover queue sizes for shared resources
 - Variable virtual machine implementation
 - Not possible to do space or timing analysis

Real Time Executives

- Real-time executives are specialised operating systems which manage processes in the RTS
- Responsible for process management and resource (processor and memory) allocation
- Storage management, fault management.

- Components depend on complexity of system

Executive components

- Real-time clock
 - Provides information for process scheduling.
- Interrupt handler
 - Manages aperiodic requests for service.
- Scheduler
 - Chooses the next process to be run.
- Resource manager
 - Allocates memory and processor resources.
- Dispatchers
 - Starts process execution.

Non-stop system components

- Configuration manager
 - Responsible

3.10 TRADITIONAL COMPONENTS:

3.10.1 PROCESS PRIORITY:

- The processing of some types of stimuli must sometimes take priority
- Interrupt level priority. Highest priority which is allocated to processes requiring a very fast response
- Clock level priority. Allocated to periodic processes
- Within these, further levels of priority may be assigned

Interrupt servicing

- Control is transferred automatically to a pre-determined memory location
- This location contains an instruction to jump to an interrupt service routine
- Further interrupts are disabled, the interrupt serviced and control returned to the interrupted process
- Interrupt service routines **MUST** be short, simple and fast

Periodic process servicing

- In most real-time systems, there will be several classes of periodic process, each with different periods (the time between executions), execution times and deadlines (the time by which processing must be completed)
- The real-time clock ticks periodically and each tick causes an interrupt which schedules the process manager for periodic processes
- The process manager selects a process which is ready for execution

Process management

- Concerned with managing the set of concurrent processes
- Periodic processes are executed at pre-specified time intervals
- The executive uses the real-time clock to determine when to execute a process
- Process period - time between executions
- Process deadline - the time by which processing must be complete

Process switching

- The scheduler chooses the next process to be executed by the processor. This depends

on a scheduling strategy which may take the process priority into account

- The resource manager allocates memory and a processor for the process to be executed
- The dispatcher takes the process from ready list, loads it onto a processor and starts execution

Scheduling strategies

- Non pre-emptive scheduling
 - Once a process has been scheduled for execution, it runs to completion or until it is blocked for some reason (e.g. waiting for I/O)
- Pre-emptive scheduling
 - The execution of an executing processes may be stopped if a higher priority process requires service
- Scheduling algorithms
 - Round-robin
 - Shortest deadline first

Data Acquisition System

- Collect data from sensors for subsequent processing and analysis.
- Data collection processes and processing processes may have different periods and deadlines.
- Data collection may be faster than processing
e.g. collecting information about an explosion, scientific experiments
- Circular or ring buffers are a mechanism for smoothing speed differences.

Reactor data collection

- A system collects data from a set of sensors monitoring the neutron flux from a nuclear reactor.
- Flux data is placed in a ring buffer for later processing.
- The ring buffer is itself implemented as a concurrent process so that the collection and processing processes may be synchronized.

Mutual exclusion

- Producer processes collect data and add it to the buffer. Consumer processes take data from the buffer and make elements available.
- Producer and consumer processes must be mutually excluded from accessing the same element.
- The buffer must stop producer processes adding information to a full buffer and consumer processes trying to take information from an empty buffer.

Monitoring and Control System

- Important class of real-time systems
- Continuously check sensors and take actions depending on sensor values
- Monitoring systems examine sensors and report their results
- Control systems take sensor values and control hardware actuators
- Burglar alarm system e.g
- A system is required to monitor sensors on doors and windows to detect the presence of intruders in a building

- When a sensor indicates a break-in, the system switches on lights around the area and calls police automatically
- The system should include provision for operation without a mains power supply

3.10.2 ALARM SYSTEM:

- Sensors
 - Movement detectors, window sensors, door sensors.
 - 50 window sensors, 30 door sensors and 200 movement detectors
 - Voltage drop sensor
- Actions
 - When an intruder is detected, police are called automatically.
 - Lights are switched on in rooms with active sensors.
 - An audible alarm is switched on.
 - The system switches automatically to backup power when a voltage drop is detected.

The R-T system design process

- Identify stimuli and associated responses
- Define the timing constraints associated with each stimulus and response
- Allocate system functions to concurrent processes
- Design algorithms for stimulus processing and response generation
- Design a scheduling system which ensures that processes will always be scheduled to meet their deadlines
- Stimuli to be processed
- Power failure
 - Generated by a circuit monitor. When received, the system must switch to backup power within 50 ms
- Intruder alarm
 - Stimulus generated by system sensors. Response is to call the police, switch on building lights and the audible alarm

Timing requirements

Stimulus/Response	Timing requirements
Power fail interrupt	The switch to backup power must be completed within a deadline of 50 ms.
Door alarm	Each door alarm should be polled twice per second.
Window alarm	Each window alarm should be polled twice per second.
Movement detector	Each movement detector should be polled twice per second.
Audible alarm	The audible alarm should be switched on within 1/2 second of an alarm being raised by a sensor.
Lights	switch The lights should be switched on
within 1/2 second	of an alarm being raised by a sensor.
Communications	The call to the police should be started within 2

Voice synthesiser

4

seconds of an alarm being raised by a sensor.
A synthesised message should be available within

seconds of an alarm being raised by a sensor

Building monitor process

```
class BuildingMonitor extends Thread {

    BuildingSensor win, door, move ;

    Siren siren = new Siren () ;
    Lights lights = new Lights ()
    ;
    Synthesizer synthesizer = new Synthesizer () ;
    DoorSensors doors = new DoorSensors (30) ;
    WindowSensors windows = new
        WindowSensors (50) ;
    MovementSensors movements = new MovementSensors
    (200) ; PowerMonitor pm = new PowerMonitor () ;

    BuildingMonitor()
    {
        // initialise all the sensors and start the
        processes siren.start () ; lights.start () ;
        synthesizer.start () ; windows.start () ;
        doors.start () ; movements.start () ; pm.start
        () ;
    }

    public void run ()
    {
        int room = 0
        ; while (true)
        {
            // poll the movement sensors at least twice per second
            (400 Hz) move = movements.getVal () ;
            // poll the window sensors at least twice/second (100 Hz)
            win = windows.getVal () ;
            // poll the door sensors at least twice per second
            (60 Hz) door = doors.getVal () ;
            if (move.sensorVal == 1 | door.sensorVal == 1 | win.sensorVal == 1)
            {
                // a sensor has indicated an intruder
```

```
        movements.shutdown () ;  
  
    } //  
    run  
} //BuildingMonitor
```

- A burglar alarm system is primarily a monitoring system. It collects data from sensors but no real-time actuator control
- Control systems are similar but, in response to sensor values, the system sends control signals to actuators
- An example of a monitoring and control system is a system which monitors temperature and switches heaters on and off

STUCOR APP

UNIT IV

TESTING

4.1 SOFTWARE TESTING FUNDAMENTALS

Taxonomy of Software Testing

Classified by purpose, software testing can be divided into: correctness testing, performance testing, and reliability testing and security testing.

Classified by life-cycle phase, software testing can be classified into the following categories: requirements phase testing, design phase testing, program phase testing, evaluating test results, installation phase testing, acceptance testing and maintenance testing.

By scope, software testing can be categorized as follows: unit testing, component testing, integration testing, and system testing.

Correctness testing

Correctness is the minimum requirement of software, the essential purpose of testing. It is used to tell the right behavior from the wrong one. The tester may or may not know the inside details of the software module under test, e.g. control flow, data flow, etc. Therefore, either a white-box point of view or black-box point of view can be taken in testing software. We must note that the black-box and white-box ideas are not limited in correctness testing only.

Black-box testing
White-box testing

Performance testing

Not all software systems have specifications on performance explicitly. But every system will have implicit performance requirements. The software should not take infinite time or infinite resource to execute. "Performance bugs" sometimes are used to refer to those design problems in software that cause the system performance to degrade.

Performance has always been a great concern and a driving force of computer evolution. Performance evaluation of a software system usually includes: resource usage, throughput, stimulus-response time and queue lengths detailing the average or maximum number of tasks waiting to be serviced by selected resources. Typical resources that need to be considered include network bandwidth requirements, CPU cycles, disk space, disk access operations, and memory

usage. The goal of performance testing can be performance bottleneck identification, performance comparison and evaluation, etc.

Reliability testing

Software reliability refers to the probability of failure-free operation of a system. It is related to many aspects of software, including the testing process. Directly estimating software reliability by quantifying its related factors can be difficult. Testing is an effective sampling method to measure software reliability. Guided by the operational profile, software testing (usually black- box testing) can be used to obtain failure data, and an estimation model can be further used to analyze the data to estimate the present reliability and predict future reliability. Therefore, based on the estimation, the developers can decide whether to release the software, and the users can decide whether to adopt and use the software.

Security testing

Software quality, reliability and security are tightly coupled. Flaws in software can be exploited by intruders to open security holes. With the development of the Internet, software security problems are becoming even more severe.

Many critical software applications and services have integrated security measures against malicious attacks. The purpose of security testing of these systems include identifying and removing software flaws that may potentially lead to security violations, and validating the effectiveness of security measures. Simulated security attacks can be performed to find vulnerabilities.

Types of S/W Test

Acceptance testing

Testing to verify a product meets customer specified requirements. A customer usually does this type of testing on a product that is developed externally.

Compatibility testing

This is used to ensure compatibility of an application or Web site with different browsers, OSs, and hardware platforms. Compatibility testing can be performed manually or can be driven by an automated functional or regression test suite.

Conformance testing

This is used to verify implementation conformance to industry standards. Producing tests for the behavior of an implementation to be sure it provides the portability, interoperability, and/or compatibility a standard defines.

Integration testing

Modules are typically code modules, individual applications, client and server applications on a network, etc. Integration Testing follows unit testing and precedes system testing.

Load testing

Load testing is a generic term covering Performance Testing and Stress Testing.

Performance testing

This testing can be applied to understand your application or WWW site's scalability, or to benchmark the performance in an environment of third party products such as servers and middleware for potential purchase. This sort of testing is particularly useful to identify performance bottlenecks in high use applications. Performance testing generally involves an automated test suite as this allows easy simulation of a variety of normal, peak, and exceptional load conditions.

Regression testing

Similar in scope to a functional test, a regression test allows a consistent, repeatable validation of each new release of a product or Web site. Such testing ensures reported product defects have been corrected for each new release and that no new quality problems were introduced in the maintenance process. Though regression testing can be performed manually an automated test suite is often used to reduce the time and resources needed to perform the required testing.

System testing

Entire system is tested as per the requirements. Black-box type testing that is based on overall requirements specifications, covers all combined parts of a system.

End-to-end testing

Similar to system testing, involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.

Sanity testing

Testing is to determine if a new software version is performing well enough to accept it for a major testing effort. If application is crashing for initial use then system is not stable enough for further testing and build or application is assigned to fix.

Alpha testing

In house virtual user environment can be created for this type of testing. Testing is done at the end of development. Still minor design changes may be made as a result of such testing.

Beta testing

Testing is typically done by end -users or others. This is the final testing before releasing the application to commercial purpose.

4.2 INTERNAL AND EXTERNAL VIEWS OF TESTING

Software Testing Techniques

Software Testing:

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

Testing Objectives:

Testing is the process of executing a program with the intent of finding errors.
A good test case is one with a high probability of finding an as-yet undiscovered error.

A successful test is one that discovers an as-yet-undiscovered error.

Testing Principles:

All tests should be traceable to customer requirements.
Tests should be planned before testing begins.

80% of all errors are in 20% of the code.
Testing should begin in the small and progress to the large.
Exhaustive testing is not possible.

Specification may be wrong.

Specification may be a physical impossibility.
Faulty program design.

Program may be incorrect.

Types of Errors:

Algorithmic error.
Computation & precision error.
Documentation error.
Capacity error or boundary error.
Timing and coordination error.

Throughput or performance error.
Recovery error.

Hardware & system software error.
Standards & procedure errors.

Operability

- if it works better it can be tested more efficiently

Observability

- what you see is what you test

Controllability

- if software can be controlled better the it is more that testing can be automated and optimized

Decomposability

- controlling the scope of testing allows problems to be isolated quickly and retested intelligently

Stability

- the fewer the changes, the fewer the disruptions to testing

Understandability

- the more information that is known, the smarter the testing can be done

A good test has a high probability of finding an error.
A good test is not redundant.
A good test should be best of breed.
A good test should not be too simple or too complex.

Test Strategies:

Black-box or behavioral testing,

- knowing the specified function a product is to perform and demonstrating correct operation based solely on its specification without regard for its internal logic

White-box or glass-box testing

- knowing the internal workings of a product, tests are performed to check the workings of all possible logic paths

STUCOR APP

4.3 WHITE BOX TESTING - BASIS PATH TESTING:

White-box technique usually based on the program flow graph,

The cyclomatic complexity of the program computed from its flow graph using the formula $V(G) = E - N + 2$ or by counting the conditional statements in the PDL representation and adding 1

Determine the basis set of linearly independent paths (the cardinality of this set is the program cyclomatic complexity)

Prepare test cases that will force the execution of each path in the basis set.

White-box testing (also known as **clear box testing**, **glass box testing**, **transparent box testing**, and **structural testing**) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the expected outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT). White-box testing can be applied at the unit, integration and system levels of the software testing process. Although traditional testers tended to think of white-box testing as being done at the unit level, it is used for integration and system testing more frequently today. It can test paths within a unit, paths between units during integration, and between subsystems during a system-level test. Though this method of test design can uncover many errors or problems, it has the potential to miss unimplemented parts of the specification or missing requirements.

White-box test design techniques include the following code coverage criteria:

- Control flow testing
- Data flow testing
- Branch testing
- Statement coverage
- Decision coverage
- Modified condition/decision coverage
- Prime path testing
- Path testing

Basic Procedures:

White-box testing's basic procedures require the tester to have an in-depth knowledge of the source code being tested. The programmer must have a deep understanding of the application to know what kinds of test cases to create so that every visible path is exercised for testing. Once the source code is understood then the source code can be analyzed for test cases to be created. The following are the three basic steps that white-box testing takes in order to create test cases:

1. Input involves different types of requirements, functional specifications, detailed designing of documents, proper source code and security specifications. This is the preparation stage of white-box testing to lay out all of the basic information.
2. Processing involves performing risk analysis to guide whole testing process, proper test plan, execute test cases and communicate results. This is the phase of building test cases to make sure they thoroughly test the application the given results are recorded accordingly.
3. Output involves preparing final report that encompasses all of the above preparations and results.

Advantages:

White-box testing is one of the two biggest testing methodologies used today. It has several major advantages:

1. Side effects of having the knowledge of the source code is beneficial to thorough testing.
2. Optimization of code becomes easy as inconspicuous bottlenecks are exposed.
3. Gives the programmer introspection because developers carefully describe any new implementation.
4. Provides traceability of tests from the source, thereby allowing future changes to the source to be easily captured in the newly added or modified tests.
5. Easy to automate.
6. Provides clear, engineering-based rules for when to stop testing.

Disadvantages:

1. White-box testing brings complexity to testing because the tester must have knowledge of the program, including being a programmer. White-box testing requires a programmer with a high level of knowledge due to the complexity of the level of testing that needs to be done.
2. On some occasions, it is not realistic to be able to test every single existing condition of the application and some conditions will be untested.
3. The tests focus on the software as it exists, and missing functionality may not be discovered.
4. The resulting test can be fragile because they are tightly coupled to the specific implementation of the thing being tested. The code under test could be rewritten to implement the same functionality in a different way that invalidates the assumptions baked into the test. This could result in tests that fail unnecessarily or, in the worst case, tests that now give false positives and mask errors in the code.

4.4 CONTROL STRUCTURE TESTING-BLACK BOX TESTING:

Control Structure Testing – 1:

White-box techniques focusing on control structures present in the software

Condition testing (e.g. branch testing)

- focuses on testing each decision statement in a software module
- it is important to ensure coverage of all logical combinations of data that may be processed by the module (a truth table may be helpful)

Data flow testing

- selects test paths based according to the locations of variable definitions and uses in the program (e.g. definition use chains)

Loop testing

- focuses on the validity of the program loop constructs (i.e. while, for, go to)
- involves checking to ensure loops start and stop when they are supposed to (unstructured loops should be redesigned whenever possible)

Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

If the loops are independent of one another

then treat each as a simple loop

else* treat as nested loops

end if*

for example, the final loop counter value of loop 1 is used to initialize loop 2.

Black-Box Testing:

Graph-Based Testing – 1:

Black-box methods based on the nature of the relationships (links) among the program objects (nodes), test cases are designed to traverse the entire graph

Transaction flow testing

- nodes represent steps in some transaction and links represent logical connections between steps that need to be validated

Finite state modeling

- nodes represent user observable states of the software and links represent state transitions

Data flow modeling

- nodes are data objects and links are transformations of one data object to another data object

Timing modeling

- nodes are program objects and links are sequential connections between these objects

- link weights are required execution times

Black-box technique that divides the input domain into classes of data from which test cases can be derived

An ideal test case uncovers a class of errors that might require many arbitrary test cases to be executed before a general error is observed

If an input condition specifies a range, one valid and two invalid equivalence classes are defined

If an input condition requires a specific value, one valid and two invalid equivalence classes are defined

If an input condition specifies a member of a set, one valid and one invalid equivalence class is defined

If an input condition is Boolean, one valid and one invalid equivalence class is defined

Boundary Value Analysis - 1
Black-box technique

- focuses on the boundaries of the input domain rather than its center

Guidelines:

-
- If input condition specifies a range bounded by values a and b, test cases should include a and b, values just above and just below a and b
 - If an input condition specifies and number of values, test cases should be exercise the minimum and maximum numbers, as well as values just above and just below the minimum and maximum values

Apply guidelines 1 and 2 to output conditions, test cases should be designed to produce the minimum and maximum output reports

If internal program data structures have boundaries (e.g. size limitations), be certain to test the boundaries

Black-box testing for safety critical systems in which independently developed implementations of redundant systems are tested for conformance to specifications

Often equivalence class partitioning is used to develop a common set of test cases for each implementation

Black-box technique that enables the design of a reasonably small set of test cases that provide maximum test coverage

Focus is on categories of faulty logic likely to be present in the software component (without examining the code)

Priorities for assessing tests using an orthogonal array

- Detect and isolate all single mode faults
- Detect all double mode faults
- Multimode faults

Testing begins at the component level and works outward toward the integration of the entire computer-based system.

Different testing techniques are appropriate at different points in time.

The developer of the software conducts testing and may be assisted by independent test groups for large projects.

The role of the independent tester is to remove the conflict of interest inherent when the builder is testing his or her own product.

Testing and debugging are different activities.
Debugging must be accommodated in any testing strategy.

Need to consider verification issues
– are we building the product right?

Need to Consider validation issues are we building the right product?

Verification vs validation:

Verification:

"Are we building the product right" The software should conform to its specification

Validation:

"Are we building the right product" The software should do what the user really requires

The V & V process:

As a whole life-cycle process - V & V must be applied at each stage in the software process.

Has two principal objectives

- The discovery of defects in a system
- The assessment of whether or not the system is usable in an operational situation.

Strategic Testing Issues - 1 Specify product requirements in a quantifiable manner before testing starts.

Specify testing objectives explicitly.

Identify the user classes of the software and develop a profile for each.

Develop a test plan that emphasizes rapid cycle testing.

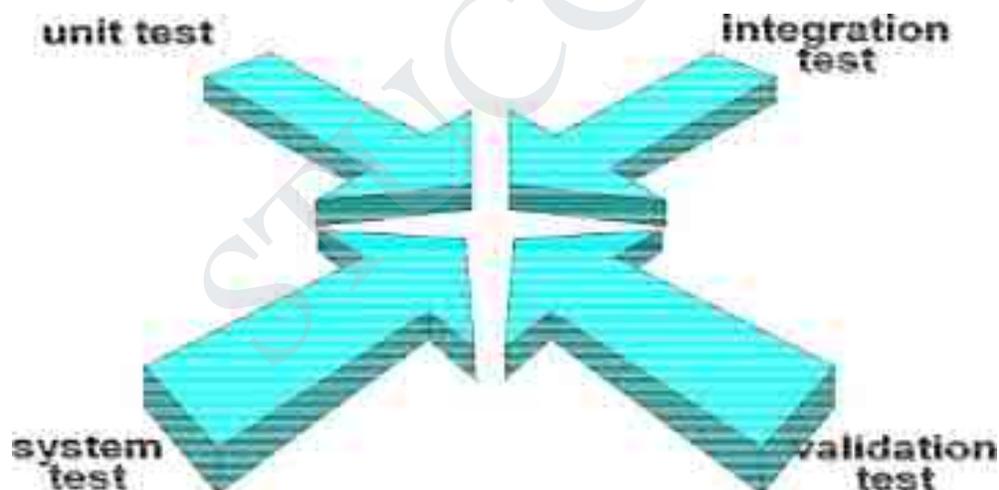
Strategic Testing Issues – 2:

Build robust software that is designed to test itself (e.g. use anti-bugging).

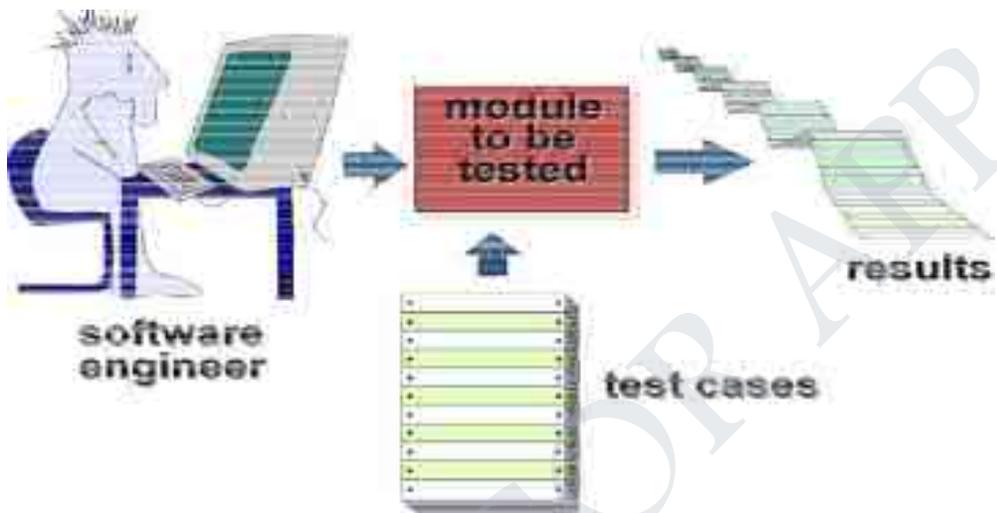
Use effective formal reviews as a filter prior to testing.

Conduct formal technical reviews to assess the test strategy and test cases.

Testing Strategy:



Unit Testing:



Program reviews.
Formal verification.

Testing the program itself.
– black box and white box testing.

Black Box or White Box?:

Maximum # of logic paths - determine if white box testing is possible.
Nature of input data.
Amount of computation involved.

Complexity of algorithms.

Interfaces tested for proper information flow.
Local data are examined to ensure that integrity is maintained.
Boundary conditions are tested.
Basis path testing should be used.

All error handling paths should be tested.
Drivers and/or stubs need to be developed to test incomplete software.

4.5 REGRESSION TESTING:

The selective retesting of a software system that has been modified to ensure that any bugs have been fixed and that no other previously working functions have failed as a result of the reparations and that newly added features have not created problems with previous versions of the software. Also referred to as verification testing, regression testing is initiated after a programmer has attempted to fix a recognized problem or has added source code to a program that may have inadvertently introduced errors. It is a quality control measure to ensure that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the maintenance activity.

Regression Testing:

Regression test suit contains 3 different classes of test cases

- Representative sample of existing test cases is used to exercise all software functions.

- Additional test cases focusing software functions likely to be affected by the change.

- Tests cases that focus on the changed software components.

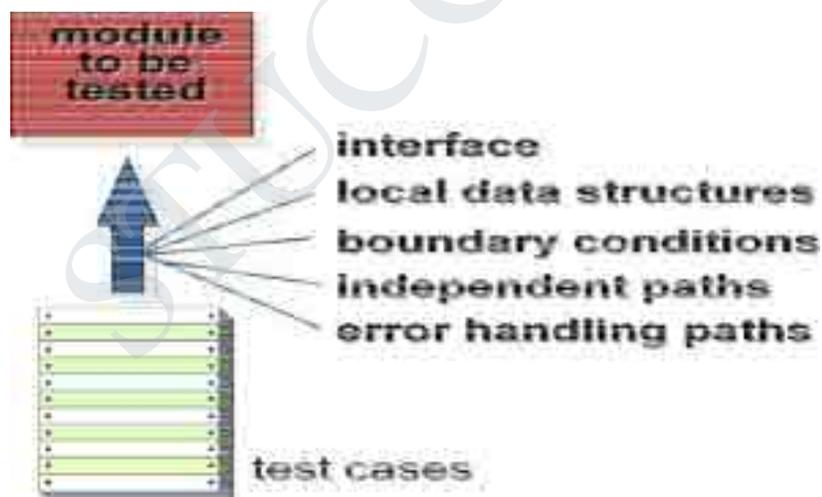
Software components already translated into code are integrated into a build.

A series of tests designed to expose errors that will keep the build from performing its functions are created.

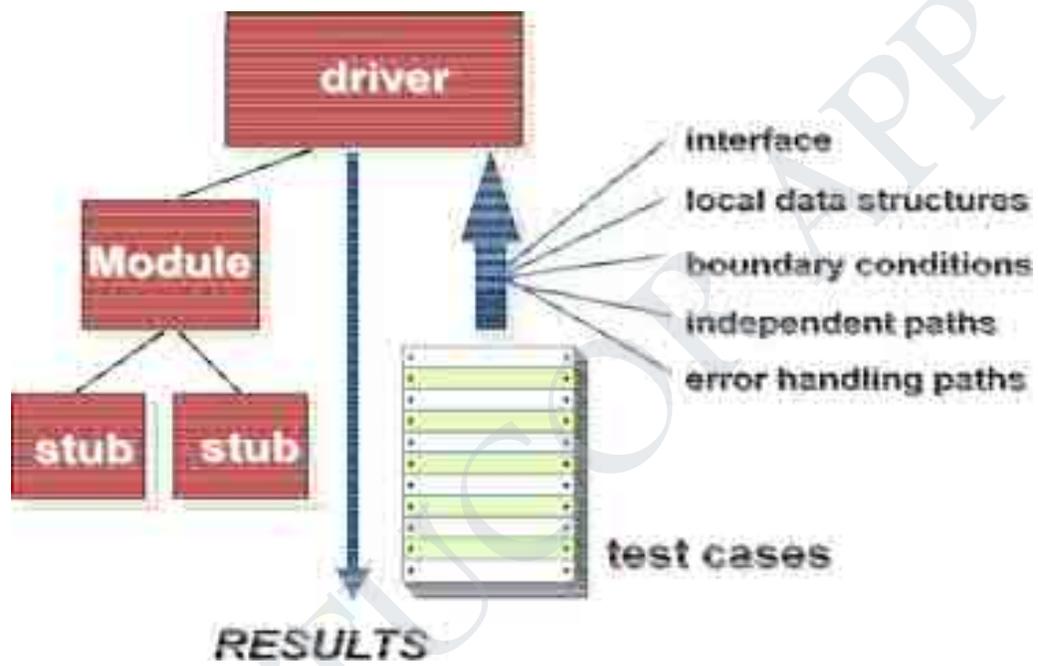
The build is integrated with the other builds and the entire product is smoke tested daily using either top-down or bottom integration.

4.6 UNIT TESTING – INTEGRATION TESTING:

Unit Testing:



Unit Test Environment:



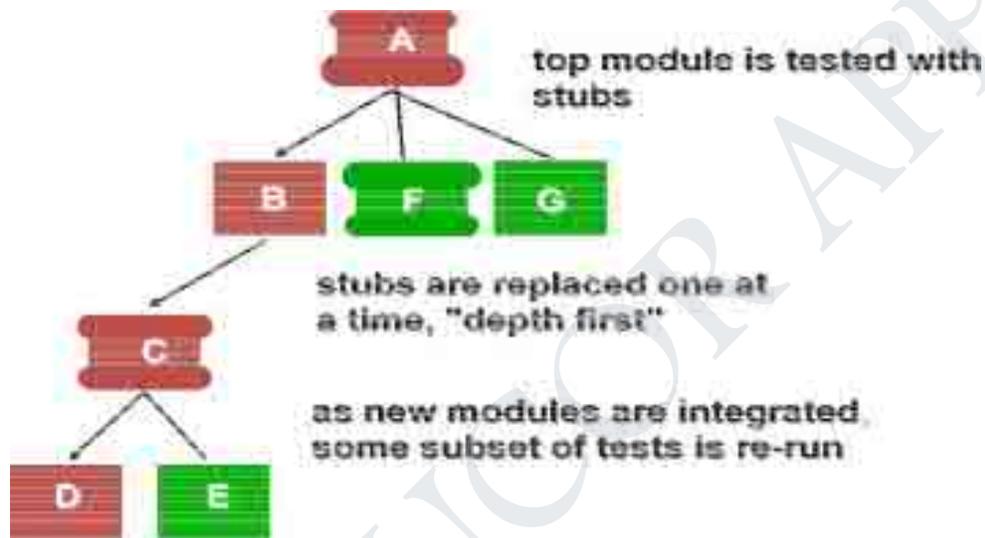
Integration Testing:

Bottom - up testing (test harness).

Top - down testing (stubs).

Regression Testing.

Smoke Testing

Top Down Integration:**Top-Down Integration Testing:**

Main program used as a test driver and stubs are substitutes for components directly subordinate to it.

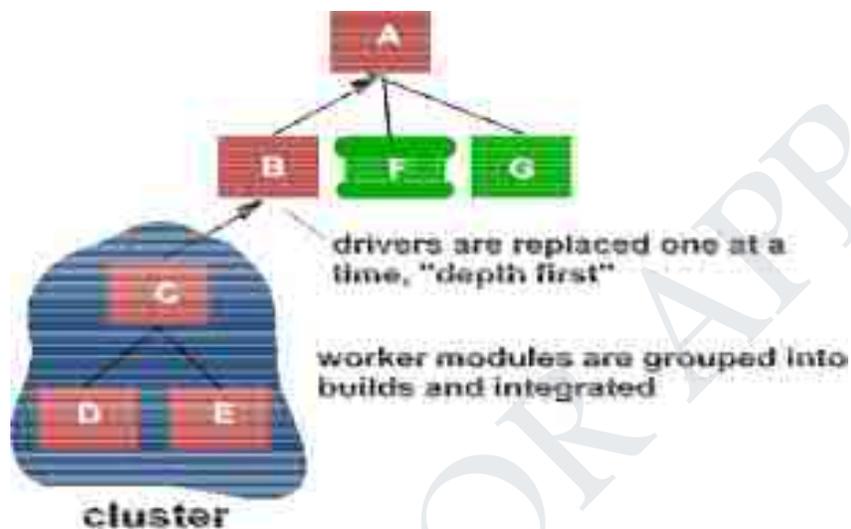
Subordinate stubs are replaced one at a time with real components (following the depth-first or breadth-first approach).

Tests are conducted as each component is integrated.

On completion of each set of tests and other stub is replaced with a real component.

Regression testing may be used to ensure that new errors not introduced.

Bottom-Up Integration:



4.7 VALIDATION TESTING:

Ensure that each function or performance characteristic conforms to its specification. Deviations (deficiencies) must be negotiated with the customer to establish a means for resolving the errors.

Configuration review or audit is used to ensure that all elements of the software configuration have been properly developed, cataloged, and documented to allow its support during its maintenance phase.

Making sure the software works correctly for intended user in his or her normal work environment.

Alpha test

- version of the complete software is tested by customer under the supervision of the developer at the developer's site

Beta test

- version of the complete software is tested by customer at his or her own site without the developer being present

Recovery testing

- checks system's ability to recover from failures

Security testing

- verifies that system protection mechanism prevents improper penetration or data alteration

Stress testing

- program is checked to see how well it deals with abnormal resource demands

Performance testing

- tests the run-time performance of software

Stress test.

Volume test.

Configuration test (hardware & software).

Compatibility.
Regression tests.

Security tests.
Timing tests.

Environmental tests.
Quality tests.

Recovery tests.
Maintenance tests.

Documentation tests.
Human factors tests.

4.8 SYSTEM TESTING AND DEBUGGING:

Establish test objectives.

Design criteria (review criteria).

– Correct.

– Feasible.

Writing test cases.
Testing test cases.
Execute test cases.

Evaluate test results.

Testing Tools:

Simulators.
Monitors.

Analyzers.
Test data generators.

Document Each Test Case:

Requirement tested.
Facet / feature / path tested.
Person & date.

Tools & code needed.
Test data & instructions.

Expected results.
Actual test results & analysis
Correction, schedule, and signoff.

Debugging:

Debugging (removal of a defect) occurs as a consequence of successful testing.
Some people better at debugging than others.

Is the cause of the bug reproduced in another part of the program?

What —next bugll might be introduced by the fix that is being proposed?

What could have been done to prevent this bug in the first place?

4.9 SOFTWARE IMPLEMENTATION TECHNIQUES: CODING PRACTICES

Software Implementation techniques

Implementation techniques include imperative languages (object-oriented or procedural), functional languages, and logic languages.

Software Implementation Techniques include process and thread scheduling, synchronization and concurrency primitives, file management, memory management, performance, networking facilities, and user interfaces. Software Implementation Techniques is designed to facilitate determining what is required to implement a specific operating system function.

Procedural programming

Procedural programming can sometimes be used as a synonym for imperative programming (specifying the steps the program must take to reach the desired state), but can also refer (as in this article) to a programming paradigm, derived from structured programming, based upon the concept of the procedure call. Procedures, also known as routines, subroutines, methods, or functions (not to be confused with mathematical functions, but similar to those used in functional programming) simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

STUCOR APP

Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm that uses "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance. Many modern programming languages now support OOP.

An object-oriented program may thus be viewed as a collection of interacting *objects*, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent 'machine' with a distinct role or responsibility. The actions (or "methods") on these objects are closely associated with the object. For example, OOP data structures tend to 'carry their own operators around with them' (or at least "inherit" them from a similar object or class). In the conventional model, the data and operations on the data don't have a tight, formal association.

functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state. Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus.

In practice, the difference between a mathematical function and the notion of a "function" used in imperative programming is that imperative functions can have side effects, changing the value of already calculated computations. Because of this they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program. Conversely, in functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result $f(x)$ both times. Eliminating side effects can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming. JavaScript, one of the most widely employed languages today, incorporates functional programming capabilities.

Logic programming is, in its broadest sense, the use of mathematical logic for computer programming. In this view of logic programming, which can be traced at least as far back as John McCarthy's [1958] advice-taker proposal, logic is used as a purely declarative representation language, and a theorem-prover or model-generator is used as the problem-solver. The problem-solving task is split between the programmer, who is responsible only for ensuring the truth of programs expressed in logical form, and the theorem-prover or model-generator, which is responsible for solving problems efficiently.

Oracle's Application Implementation Method

AIM provides with an integrated set of templates, procedures, PowerPoint presentations, spreadsheets, and project plans for implementing the applications. AIM was such a success, Oracle created a subset of the templates, called it AIM Advantage, and made it available as a product to customers and other consulting firms. Since its initial release, AIM has been revised and improved several times with new templates and methods.

STUCOR APP

AIM Is a Six-Phase Method

Because the Oracle ERP Applications are software modules buy from a vendor, different implementation methods are used than the techniques used for custom developed systems. AIM has six major phases:

Definition phase: During this phase, you plan the project, determine business objectives, and verify the feasibility of the project for given time, resource, and budget limits.

Operations Analysis phase: Includes documents business requirements, gaps in the software (which can lead to customizations), and system architecture requirements. Results of the analysis should provide a proposal for future business processes, a technical architecture model, an application architecture model, workarounds for application gaps, performance testing models, and a transition strategy to migrate to the new systems. Another task that can begin in this phase is mapping of legacy data to Oracle Application APIs or open interfaces—data conversion.

Solution Design phase—Used to create designs for solutions that meet future business requirements and processes. The design of your future organization comes alive during this phase as customizations and module configurations are finalized.

Build phase—During this phase of AIM, coding and testing of customizations, enhancements, interfaces, and data conversions happens. In addition, one or more conference room pilots test the integrated enterprise system. The results of the build phase should be a working, tested business system solution.

Transition phase—During this phase, the project team delivers the finished solution to the enterprise. End-user training and support, management of change, and data conversions are major activities of this phase.

Production phase—Starts when the system goes live. Technical people work to stabilize and maintain the system under full transaction loads. Users and the implementation team begin a series of refinements to minimize unfavorable impacts and realize the business objectives identified in the definition phase.

Rapid Implementations

In the late 1990s as Y2K approached, customers demanded and consulting firms discovered faster ways to implement packaged software applications. The rapid implementation became

possible for certain types of customers. The events that converged in the late 1990s to provide faster implementations include the following:

Many smaller companies couldn't afford the big ERP project. If the software vendors and consulting firms were going to sell to the —middle marketll companies, they had to develop more efficient methods.

Many dotcoms needed a financial infrastructure; ERP applications filled the need, and rapid implementation methods provided the way.

The functionality of the software improved a lot, many gaps were eliminated, and more companies could implement with fewer customizations.

After the big, complex companies implemented their ERP systems, the typical implementation became less difficult.

The number of skilled consultants and project managers increased significantly.

Other software vendors started packaging preprogrammed integration points to the Oracle ERP modules.

Rapid implementations focus on delivering a predefined set of functionality. A key set of business processes is installed in a standard way to accelerate the implementation schedule. These projects benefit from the use of preconfigured modules and predefined business processes. You get to reuse the analysis and integration testing from other implementations, and you agree to ignore all gaps by modifying your business to fit the software. Typically, the enterprise will be allowed some control over key decisions such as the structure of the chart of accounts. Fixed budgets are set for training, production support, and data conversions (a limited amount of data).

Phased Implementations

Phased implementations seek to break up the work of an ERP implementation project. This technique can make the system more manageable and reduce risks, and costs in some cases, to the enterprise. In the mid-1990s, 4 or 5 was about the maximum number of application modules that could be launched into production at one time. If you bought 12 or 13 applications, there would be a financial phase that would be followed by phases for the distribution and manufacturing applications. As implementation techniques improved and Y2K pressures grew in the late 1990s, more and more companies started launching most of their applications at the same time. This method became known as the big-bang approach. Now, each company selects a phased or big-bang approach based on its individual requirements.

Another approach to phasing can be employed by companies with business units at multiple sites. With this technique, one business unit is used as a template, and all applications are completely implemented in an initial phase lasting 10–14 months. Then, other sites implement the applications in cookie-cutter fashion. The cookie-cutter phases are focused on end-user training and the differences that a site has from the prototype site. The cookie-cutter phase can be as short as 9–12 weeks, and these phases can be conducted at several sites simultaneously. For your reference, we participated in an efficient project where 13 applications were implemented big bang–style in July at the Chicago site after about 8 months work. A site in Malaysia went live in October. The Ireland site started up in November. After a holiday break, the Atlanta business unit went live in February, and the final site in China started using the applications in April. Implementing thirteen application modules at five sites in four countries in sixteen months was pretty impressive.

Case Studies Illustrating Implementation Techniques

Some practical examples from the real world might help to illustrate some of the principles and techniques of various software implementation methods. These case studies are composites from about 60 implementation projects we have observed during the past 9 years.

Big companies often have a horrible time resolving issues and deciding on configuration parameters because there is so much money involved and each of many sites might want to control decisions about what it considers its critical success factors. For example, we once saw a

large company argue for over two months about the chart of accounts structure, while eight consultants from two consulting firms tried to referee among the feuding operating units. Another large company labored for more than six months to unify a master customer list for a centralized receivables and decentralized order entry system.

Transition activities at large companies need special attention. Training end users can be a logistical challenge and can require considerable planning. For example, if you have 800 users to train and each user needs an average of three classes of two hours each and you have one month, how many classrooms and instructors do you need? Another example is that loading data

STUCOR APP

from a legacy system can be a problem. If you have one million customers to load into Oracle receivables at the rate of 5,000/hour and the database administrator allows you to load 20 hours per day, you have a 10-day task.

Because they spend huge amounts of money on their ERP systems, many big companies try to optimize the systems and capture specific returns on the investment. However, sometimes companies can be incredibly insensitive and uncoordinated as they try to make money from their ERP software. For example, one business announced at the beginning of a project that the accounts payable department would be cut from 50–17 employees as soon as the system went live. Another company decided to centralize about 30 accounting sites into one shared service center and advised about 60 accountants that they would lose their jobs in about a year. Several of the 60 employees were offered positions on the ERP implementation team.

Small companies have other problems when creating an implementation team. Occasionally, the small company tries to put clerical employees on the team and they have problems with issue resolution or some of the ERP concepts. In another case, one small company didn't create the position of project manager. Each department worked on its own modules and ignored the integration points, testing, and requirements of other users. When Y2K deadlines forced the system startup, results were disastrous with a cost impact that doubled the cost of the entire project.

Project team members at small companies sometimes have a hard time relating to the cost of the implementation. We once worked with a company where the project manager (who was also the database administrator) advised me within the first hour of our meeting that he thought consulting charges of \$3/minute were outrageous, and he couldn't rationalize how we could possibly make such a contribution. We agreed a consultant could not contribute \$3 in value each and every minute to his project. However, when I told him we would be able to save him \$10,000/week and make the difference between success and failure, he realized we should get to work.

Because the small company might be relatively simple to implement and the technical staff might be inexperienced with the database and software, it is possible that the technical staff will be on the critical path of the project. If the database administrator can't learn how to handle the production database by the time the users are ready to go live, you might need to hire some temporary help to enable the users to keep to the schedule. In addition, we often see small companies with just a single database administrator who might be working 60 or more hours per week. They feel they can afford to have more DBAs as employees, but they don't know how to establish the right ratio of support staff to user requirements. These companies can burn out a DBA quickly and then have to deal with the problem of replacing an important skill.

4.10 REFACTORING:

Refactoring is usually motivated by noticing a code smell.^[3] For example the method at hand may be very long, or it may be a near duplicate of another nearby method. Once recognized, such problems can be addressed by *refactoring* the source code, or transforming it into a new form that behaves the same as before but that no longer "smells". For a long routine, extract one or more smaller subroutines. Or for duplicate routines, remove the duplication and utilize one shared function in their place. Failure to perform refactoring can result in accumulating technical debt.

There are two general categories of benefits to the activity of refactoring.

1. **Maintainability.** It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp. This might be achieved by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods. It might be achieved by moving a method to a more appropriate class, or by removing misleading comments.
2. **Extensibility.** It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed.^[2]

Before refactoring a section of code, a solid set of automatic unit tests is needed. The tests should demonstrate in a few seconds that the behavior of the module is correct. The process is then an iterative cycle of making a small program transformation, testing it to ensure correctness, and making another small transformation. If at any point a test fails, you undo your last small change and try again in a different way. Through many small steps the program moves from where it was to where you want it to be. Proponents of extreme programming and other agile methodologies describe this activity as an integral part of the software development cycle.

Here are some examples of code refactorings;

some of these may only apply to certain languages or language types.

- Techniques that allow for more abstraction
 - Encapsulate Field – force code to access the field with getter and setter methods
 - Generalize Type – create more general types to allow for more code sharing
 - Replace type-checking code with State/Strategy
 - Replace conditional with polymorphism
- Techniques for breaking code apart into more logical pieces
 - Extract Method, to turn part of a larger method into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions.
 - Extract Class moves part of the code from an existing class into a new class.
- Techniques for improving names and location of code

- Move Method or Move Field – move to a more appropriate Class or source file
- Rename Method or Rename Field – changing the name into a new one that better reveals its purpose
- Pull Up – in OOP, move to a superclass
- Push Down – in OOP, move to a subclass

While the term *refactoring* originally referred exclusively to refactoring of software code, in recent years code written in hardware description languages (HDLs) has also been refactored. The term *hardware refactoring* is used as a shorthand term for refactoring of code in hardware description languages. Since HDLs are not considered to be programming languages by most hardware engineers, hardware refactoring is to be considered a separate field from traditional code refactoring.

Automated refactoring of analog hardware descriptions (in VHDL-AMS) has been proposed by Zeng and Huss. In their approach, refactoring preserves the simulated behavior of a hardware design. The non-functional measurement that improves is that refactored code can be processed by standard synthesis tools, while the original code cannot. Refactoring of digital HDLs, albeit manual refactoring, has also been investigated by Synopsys fellow Mike Keating. His target is to make complex systems easier to understand, which increases the designers' productivity.

UNIT V

SOFTWARE PROJECT MANAGEMENT

5.1 ESTIMATION – FP BASED, LOC BASED:

Measures and Measurements

Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.

This allows for objective comparisons between techniques and processes.

Although some companies have introduced measurement programmes, most organisations still don't make systematic use of software measurement.

There are few established standards in this area.

Software metric

Any type of measurement which relates to a software system, process or related documentation

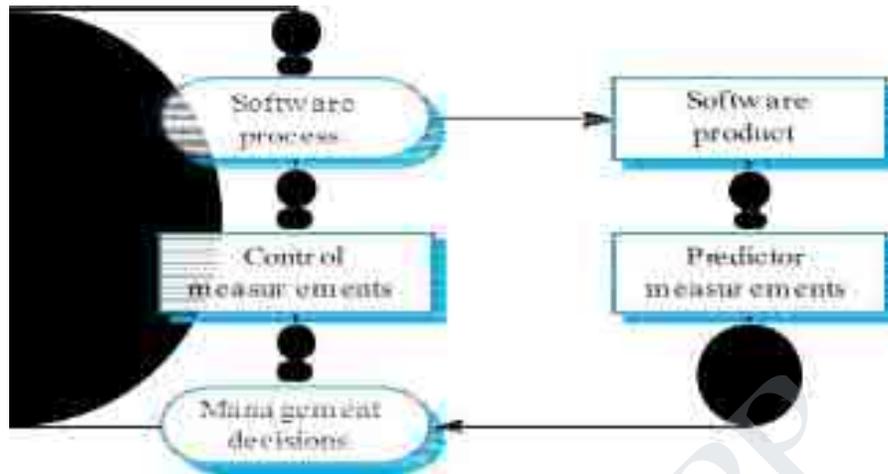
Lines of code in a program, the Fog index, number of person-days required to develop a component.

Allow the software and the software process to be quantified.

May be used to predict product attributes or to control the software process.

Product metrics can be used for general predictions or to identify anomalous components.

Predictor and control metrics



Metrics assumptions

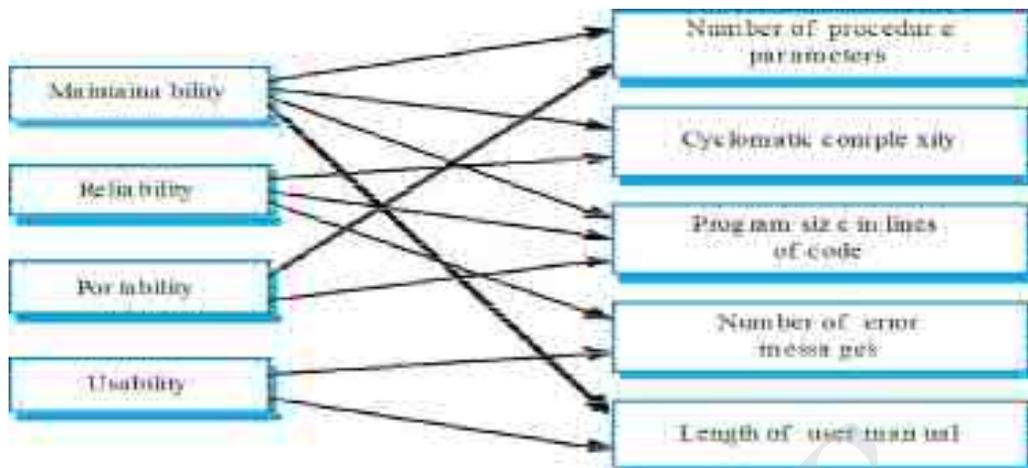
A software property can be measured.

The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.

This relationship has been formalised and validated.

It may be difficult to relate what can be measured to desirable external quality attributes.

Internal and external attributes



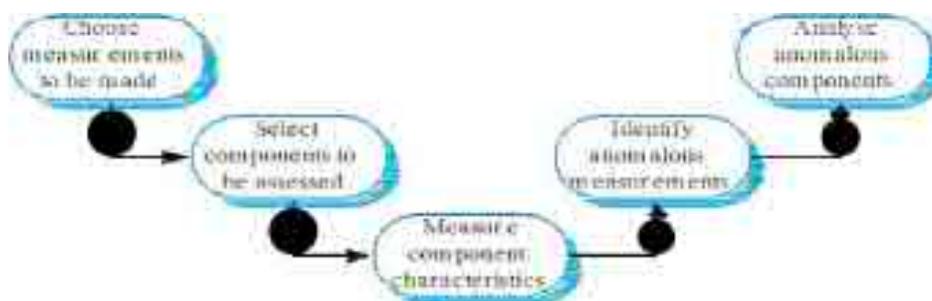
The measurement process

A software measurement process may be part of a quality control process.

Data collected during this process should be maintained as an organisational resource.

Once a measurement database has been established, comparisons across projects become possible.

Product measurement process



Data collection

A metrics programme should be based on a set of product and process data.

Data should be collected immediately (not in retrospect) and, if possible, automatically.

Three types of automatic data collection

Static product analysis;

Dynamic product analysis;

Process data collation.

Data accuracy

Don't collect unnecessary data

The questions to be answered should be decided in advance and the required data identified.

Tell people why the data is being collected.

It should not be part of personnel evaluation.

Don't rely on memory

Collect data when it is generated not after a project has finished.

Product metrics

A quality metric should be a predictor of product quality.

Classes of product metric

Dynamic metrics which are collected by measurements made of a program in execution;

Static metrics which are collected by measurements made of the system representations;

Dynamic metrics help assess efficiency and reliability; static metrics help assess complexity, understand ability and maintainability.

Dynamic and static metrics

Dynamic metrics are closely related to software quality attributes

It is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).

Static metrics have an indirect relationship with quality attributes

You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

Software product metrics

Software metric	Description
Fan in/Fan-out	Fan-in is a measure of the number of functions or methods that call some other function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to

	coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss how to compute cyclomatic complexity in Chapter 22.
Length of identifiers	This is a measure of the average length of distinct identifiers in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if statements are hard to understand and are potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document is to understand.

Object-oriented metrics

Object-oriented metric	Description
Depth of inheritance tree	This represents the number of discrete levels in the inheritance tree where sub-classes inherit attributes and operations (methods) from super-classes. The deeper the inheritance tree, the more complex the design. Many different object classes may have to be understood to understand the object classes at the leaves of the tree.
Method fan-in/fan-out	This is directly related to fan-in and fan-out as described above and means essentially the same thing. However, it may be appropriate to make a distinction between calls from other methods within the object and calls from external methods.
Weighted methods per class	This is the number of methods that are included in a class weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1 and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be more difficult to understand. They may not be logically cohesive so cannot be reused effectively as super-classes in an inheritance tree.
Number of overriding operations	This is the number of operations in a super-class that are overridden in a sub-class. A high value for this metric indicates that the super-class used may not be an appropriate parent for the sub-class.

Measurement analysis

It is not always obvious what data means
Analysing collected data is very difficult.

Professional statisticians should be consulted if available.

Data analysis must take local circumstances into account.

Measurement surprises

Reducing the number of faults in a program leads to an increased number of help desk calls
The program is now thought of as more reliable and so has a wider more diverse market. The percentage of users who call the help desk may have decreased but the total may increase;

A more reliable system is used in a different way from a system where users work around the faults. This leads to more help desk calls.

ZIPF's Law

Zipf's Law as "the observation that frequency of occurrence of some event (P), as a function of the rank (i) when the rank is determined by the above frequency of occurrence, is a power-law function $P_i \sim 1/i^a$ with the exponent a close to unity (1)."

Let P (a random variable) represented the frequency of occurrence of a keyword in a program listing.

It applies to computer programs written in any modern computer language.

Without empirical proof because it's an obvious finding, that any computer program written in any programming language has a power law distribution, i.e., some keywords are used more than others.

Frequency of occurrence of events is inversely proportional to the rank in this frequency of occurrence.

When both are plotted on a log scale, the graph is a straight line.

we create entities that don't exist except in computer memory at run time; we create logic nodes that will never be tested because it's impossible to test every logic branch; we create information flows in quantities that are humanly impossible to analyze with a glance; Software application is the combination of keywords within the context of a solution and not their quantity used in a program; context is not a trivial task because the context of an application is attached to the problem being solved and every problem to solve is different and must have a specific program to solve it.

Although a program could be syntactically correct, it doesn't mean that the algorithms implemented solve the problem at hand. What's more, a correct program can solve the wrong problem. Let's say we have the simple requirement of printing "Hello, World!" A syntactically correct solution in Java looks as follows:

```
Public class SayHello {  
public static void main(String[] args) {  
System.out.println("John Sena!");  
    }  
}
```

This solution is obviously wrong because it doesn't solve the original requirement. This means that the context of the solution within the problem being solved needs to be determined to ensure its quality. In other words, we need to verify that the output matches the original requirement.

Zip's Law can't even say too much about larger systems.

Software Cost Estimation

Software cost components

Hardware and software costs.

Travel and training costs.

Effort costs (the dominant factor in most projects)

The salaries of engineers involved in the project;

Social and insurance costs.

Effort costs must take overheads into account

Costs of building, heating, lighting.

Costs of networking and communications.

Costs of shared facilities (e.g library, staff restaurant, etc.).

Costing and pricing

Estimates are made to discover the cost, to the developer, of producing a software system.

There is not a simple relationship between the development cost and the price charged to the customer.

Broader organisational, economic, political and business considerations influence the price charged.

Software productivity

A measure of the rate at which individual engineers involved in software development produce software and associated documentation.

Not quality-oriented although quality assurance is a factor in productivity assessment.

Essentially, we want to measure useful functionality produced per time unit.

Productivity measures

Size related measures based on some output from the software process. This may be lines of delivered source code, object code instructions, etc.

Function-related measures based on an estimate of the functionality of the delivered software. Function-points are the best known of this type of measure.

Measurement problems

Estimating the size of the measure (e.g. how many function points).

Estimating the total number of programmer months that have elapsed.

Estimating contractor productivity (e.g. documentation team) and incorporating this estimate in overall estimate.

Lines of code

The measure was first proposed when programs were typed on cards with one line per card;

How does this correspond to statements as in Java which can span several lines or where there can be several statements on one line.

Productivity comparisons

The lower level the language, the more productive the programmer

The same functionality takes more code to implement in a lower-level language than in a high-level language.

The more verbose the programmer, the higher the productivity

Measures of productivity based on lines of code suggest that programmers who write verbose code are more productive than programmers who write compact code.

Function Point model

Function points

Based on a combination of program characteristics

external inputs and outputs;

user interactions;

external interfaces;

files used by the system.

A weight is associated with each of these and the function point count is computed by multiplying each raw count by the weight and summing all values.

The function point count is modified by complexity of the project

FPs can be used to estimate LOC depending on the average number of LOC per FP for a given language

$$\text{LOC} = \text{AVC} * \text{number of function points};$$

AVC is a language-dependent factor varying from 200-300 for assemble language to 2-40 for a 4GL;

FPs are very subjective. They depend on the estimator

Automatic function-point counting is impossible.

5.2 MAKE/BUY DECISION, COCOMO II:

COCOMO model

An empirical model based on project experience.

Well-documented, ‘_independent’ model which is not tied to a specific software vendor.

Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.

COCOMO 2 takes into account different approaches to software development, reuse, etc.

COCOMO 81

Project complexity	Formula	Description
Simple	$PM = 2.4 (\text{KDSI})^{1.00} \times M$	Well-understood applications developed by small teams.
Moderate	$PM = 3.0 (\text{KDSI})^{1.12} \times M$	More complex projects where team members may have limited experience of related systems.
Embedded	$PM = 3.6 (\text{KDSI})^{1.20} \times M$	Complex projects where the software is part of a strongly coupled complex of hardware, software, regulations and operational

		procedures.
--	--	-------------

COCOMO 2

COCOMO 81 was developed with the assumption that a waterfall process would be used and that all software would be developed from scratch.

Since its formulation, there have been many changes in practice and COCOMO 2 is designed to accommodate different approaches to software development.

COCOMO 2 models

COCOMO 2 incorporates a range of sub-models that produce increasingly detailed software estimates.

The sub-models in COCOMO 2 are:

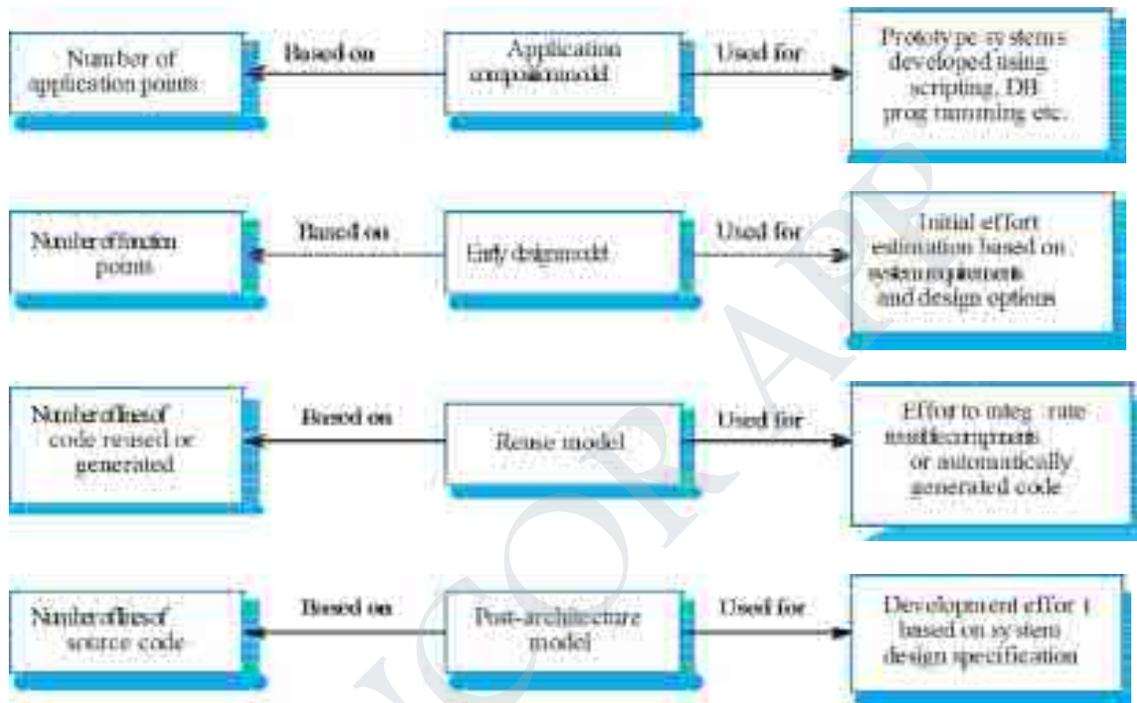
Application composition model. Used when software is composed from existing parts.

Early design model. Used when requirements are available but design has not yet started.

Reuse model. Used to compute the effort of integrating reusable components.

Post-architecture model. Used once the system architecture has been designed and more information about the system is available.

Use of COCOMO 2 models



Application composition model

Supports prototyping projects and projects where there is extensive reuse.

Based on standard estimates of developer productivity in application (object) points/month.

Takes CASE tool use into account.

Formula is

$$PM = (NAP (1 - \%reuse/100)) / PROD$$

PM is the effort in person-months, NAP is the number of application points and

PROD is the productivity.

Early design model

Estimates can be made after the requirements have been agreed.

Based on a standard formula for algorithmic models

$$PM = A \cdot Size^B \cdot M \text{ where}$$

$$M = PERS \cdot RCPX \cdot RUSE \cdot PDIF \cdot PREX \cdot FCIL \cdot SCED;$$

A = 2.94 in initial calibration, Size in KLOC, B varies from 1.1 to 1.24 depending on novelty of the project, development flexibility, risk management approaches and the process maturity.

Multipliers

Multipliers reflect the capability of the developers, the non-functional requirements, the familiarity with the development platform, etc.

5.3 PLANNING – PROJECT PLAN, PLANNING PROCESS:**Types of project plan**

Plan	Description
------	-------------

Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources and schedule used for system validation.
Configuration management Plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements of the system, maintenance costs and effort required.
Development plan.	Describes how the skills and experience of the project team members will be developed.

Project planning process

Establish the project constraints(delivery date, staff, budget)

Make initial assessments of the project parameters (structure, size)

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

Re-negotiate project constraints and deliverables

if (problems arise) then

 Initiate technical review and possible revision

end if

end loop

STUCOR APP

project plan

The project plan sets out:

resources available to the project

work breakdown
schedule for the work.

Project plan structure

Introduction – objective, budget, time

Project organisation. – roles of people

Risk analysis. – arising, reduction

Hardware and software resource requirements.

Work breakdown. – break project to activity, milestone

Project schedule. – time, allocation of people

Monitoring and reporting mechanisms.

Milestones and deliverables

Milestones are the end-point of a process activity.- report presented to management

Deliverables are project results delivered to customers.

milestones need not be deliverables. May be used by project managers. – not to customers

The waterfall process allows for the straight forward definition of progress milestones.

Milestones in requirement process



5.4 RFP:

A request for proposal (RFP) is a document that an organization posts to elicit bids from potential vendors for a desired IT solution. The RFP specifies what the customer is looking for and establishes evaluation criteria for assessing proposals. An RFP generally includes background on the issuing organization and its lines of business, a set of specifications that describe the sought-after solution, and evaluation criteria that disclose how proposals will be graded. RFPs may also include a [statement of work](#), which describes the tasks to be performed by the winning bidder and a timeline for providing deliverables.

An RFP may be issued for a number of reasons. In some cases, the complexity of an IT project calls for a formal RFP. An organization can benefit from multiple bidders and perspectives when seeking an integrated solution calling for a mix of technologies, vendors and potential configurations. A business moving from a paper-based system to a computer-based system, for example, might request proposals for all the hardware, software, and user training required to establish and integrate the new system into the organization. A simple hardware upgrade, in contrast, may only involve issuing a [request for quotation](#) to a single vendor.

Some entities such as government agencies may be required to issue RFPs to provide full and open competition. An organization may also release an RFP to boost competition to drive down the cost of a solution. That said, a proposal accepted on the basis of being the most responsive to an RFP's specifications may not always be the lowest-priced bid.

5.5 RISK MANAGEMENT – IDENTIFICATION:

Risk management

Risk management - identifying risks and drawing up plans to minimise their effect on a project.

A risk is a probability that some adverse circumstance will occur

Project risks : affect schedule or resources. eg: loss of experienced designer.

Product risks: affect the quality or performance of the software being developed.
eg: failure of purchased component.

Risk management process

Risk identification

Identify project, product and business risks;

Risk analysis

Assess the likelihood and consequences of these risks;

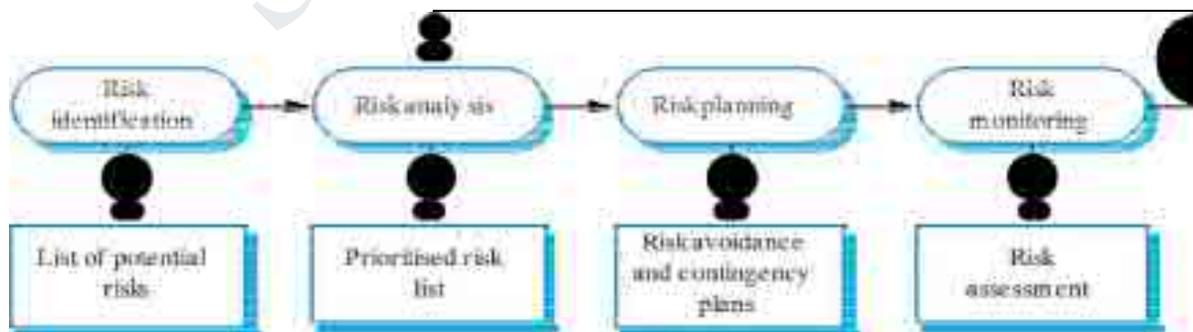
Risk planning

Draw up plans to avoid or minimise the effects of the risk;

Risk monitoring

Constantly monitor risks & plans for risk mitigation.

Risk management process



Risk identification

- Discovering possible risk
- Technology risks.
- People risks.
- Organisational risks.
- Tool risk.
- Requirements risks.
- Estimation risks.

Risks and risk types

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. Software components that should be reused contain defects that 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 that 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.

Risk analysis

Make judgement about probability and seriousness of each identified risk.

Made by experienced project managers

Probability may be very low(<10%), low(10-25%), moderate(25-50%), high(50-75%) or very high(>75%). not precise value. Only range.

Risk effects might be catastrophic, serious, tolerable or insignificant.

Risk planning

Consider each identified risk and develop a **strategy** to manage that risk.

Avoidance strategies

The probability that the risk will arise is reduced;

Minimisation strategies

The impact of the risk on the project will be reduced;

Contingency plans

If the risk arises, contingency plans are plans to deal with that risk. eg: financial problems

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	Investigate the possibility of buying a higher-

	performance database.
Underestimated development time	Investigate buying in components, investigate use of a program generator

Risk monitoring

Assess each identified risks regularly to decide whether or not it is becoming less or more probable.

Also assess whether the effects of the risk have changed.

Cannot be observed directly. Factors affecting will give clues.

Each key risk should be discussed at management progress meetings & review.

Risk indicators

Risk type	Potential indicators
Technology	Late delivery of hardware or support software, many reported technology problems
People	Poor staff morale, poor relationships amongst team member, 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

5.6 PROJECTION, RMMM:

Projection

Risk Projection (aka Risk Estimation)

Attempts to rate each risk in two ways

- · The probability that the risk is real
- · The consequences of the problems associated with the risk, should it occur.

Project planner, along with other managers and technical staff, performs four risk projection activities:

- (1) establish a measure that reflects the perceived likelihood of a risk
- (2) delineate the consequences of the risk

(3) estimate the impact of the risk on the project and the product

(4) note the overall accuracy of the risk projection so that there will be no misunderstandings..

Risk Mitigation, Monitoring, and Management

Effective strategy must consider three issues:

- risk avoidance
- risk monitoring
- risk management and contingency planning
- · Proactive approach to risk - avoidance strategy.
- · Develop *risk mitigation* plan.
 - · e.g. assume high staff turnover is noted as a project risk, r_1 .
 - · Based on past history
 - o o the likelihood, l_1 , of high turnover is estimated to be 0.70 o o the impact, x_1 , is projected at level 2.
 - o o So... high turnover will have a critical impact on project cost and schedule.
- · Develop a strategy to mitigate this risk for reducing turnover. · · Possible steps to be taken
 - o Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
 - o Mitigate those causes that are under our control before the project starts.

o Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.

o Organize project teams so that information about each development activity is widely dispersed. o Define documentation standards and establish mechanisms to be sure that documents are

developed in a timely manner.

o Conduct peer reviews of all work (so that more than one person is "up to speed"). o Assign a backup staff member for every critical technologist.

- · Project manager monitors for likelihood of risk

- · For high staff turnover, the following factors can be monitored: o General attitude of team members based on project pressures. o The degree to which the team has jelled.

o Interpersonal relationships among team members. o Potential problems with compensation and benefits.

o The availability of jobs within the company and outside it.

- · Project manager should monitor the effectiveness of risk mitigation steps.

- · *Risk management and contingency planning* assumes that mitigation efforts have failed and that the risk has become a reality.

e.g., the project is underway and a number of people announce that they will be leaving.

- · Mitigation strategy makes sure:

- § § backup is available

§ § information is documented

§ § knowledge has been dispersed across the team.

- RMMM steps incur additional project cost

e.g. spending time to "backup" every critical technologist costs money.

- Large project - 30 or 40 risks.
- 80 percent of the overall project risk (i.e., 80 percent of the potential for project failure) can be accounted for by only 20 percent of the identified risks.
- Work performed during earlier risk analysis steps will help the planner to determine which of the risks reside in that 20 percent (e.g., risks that lead to the highest risk exposure).

The RMMM Plan

- ***Risk Mitigation, Monitoring and Management Plan*** (RMMM) - documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.
- Alternative to RMMM - ***risk information sheet*** (RIS)
 1. Risk monitoring is a project tracking activity
 2. Three primary objectives:

1. assess whether predicted risks do, in fact, occur
 2. ensure that risk aversion steps defined for the risk are being properly applied
 3. collect information that can be used for future risk analysis.
- Problems that occur during a project can be traced to more than one risk.
- o Another job of risk monitoring is to attempt to allocate *origin* (what risk(s) caused which problems throughout the project).

5.7 SCHEDULING AND TRACKING –RELATIONSHIP BETWEEN PEOPLE AND EFFORT:

Project scheduling

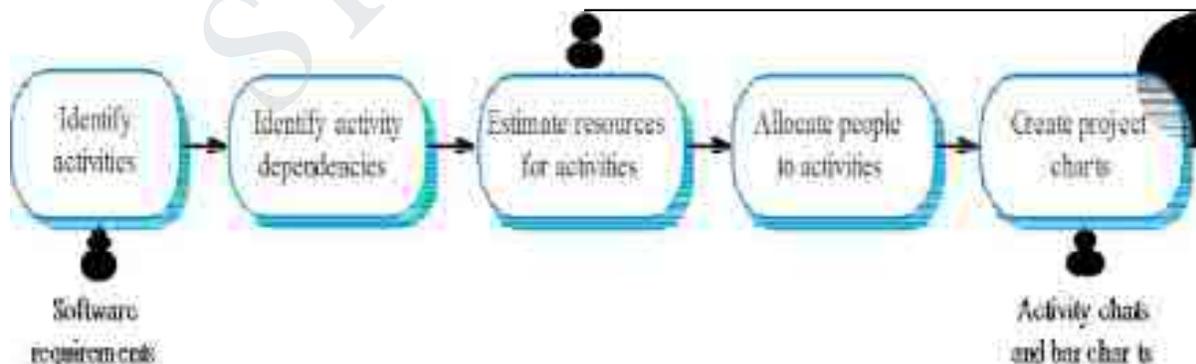
Split project into tasks and estimate time and resources required to complete each task.

Organize tasks concurrently to make optimal use of workforce.

Minimize task dependencies to avoid delays caused by one task waiting for another to complete.

Dependent on project managers intuition and experience.

The project scheduling process

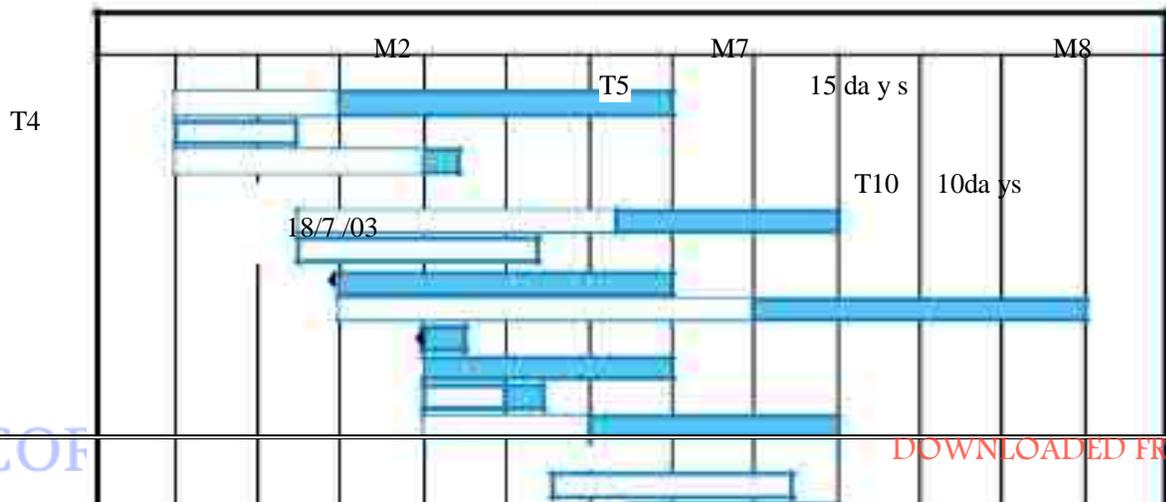
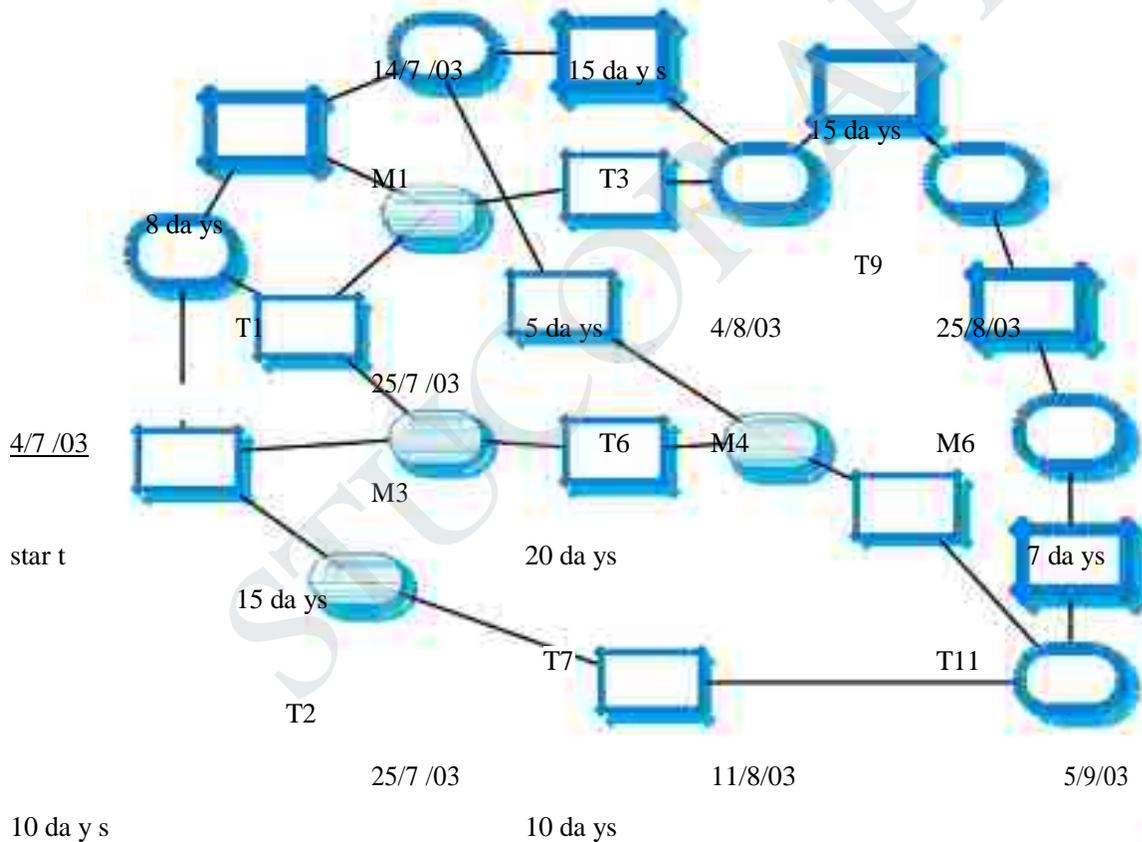


5.8 TASK SET & NETWORK

Defining Task Sets

determine type of project concept development, new application development, application enhancement, application maintenance, and reengineering projects

Activity network



T12

M5

25 da ys

T8

Finish

Activity timeline

4/7 11/7 18/7 25/7 1/8 8/8 15/8 22/8 29/8 5/9 12/9 19/9

T11

Start

T4

T1

T2

M1

T7

T3

M5

T8

M3

M2

T6

T5

M4

T9

M7

T10



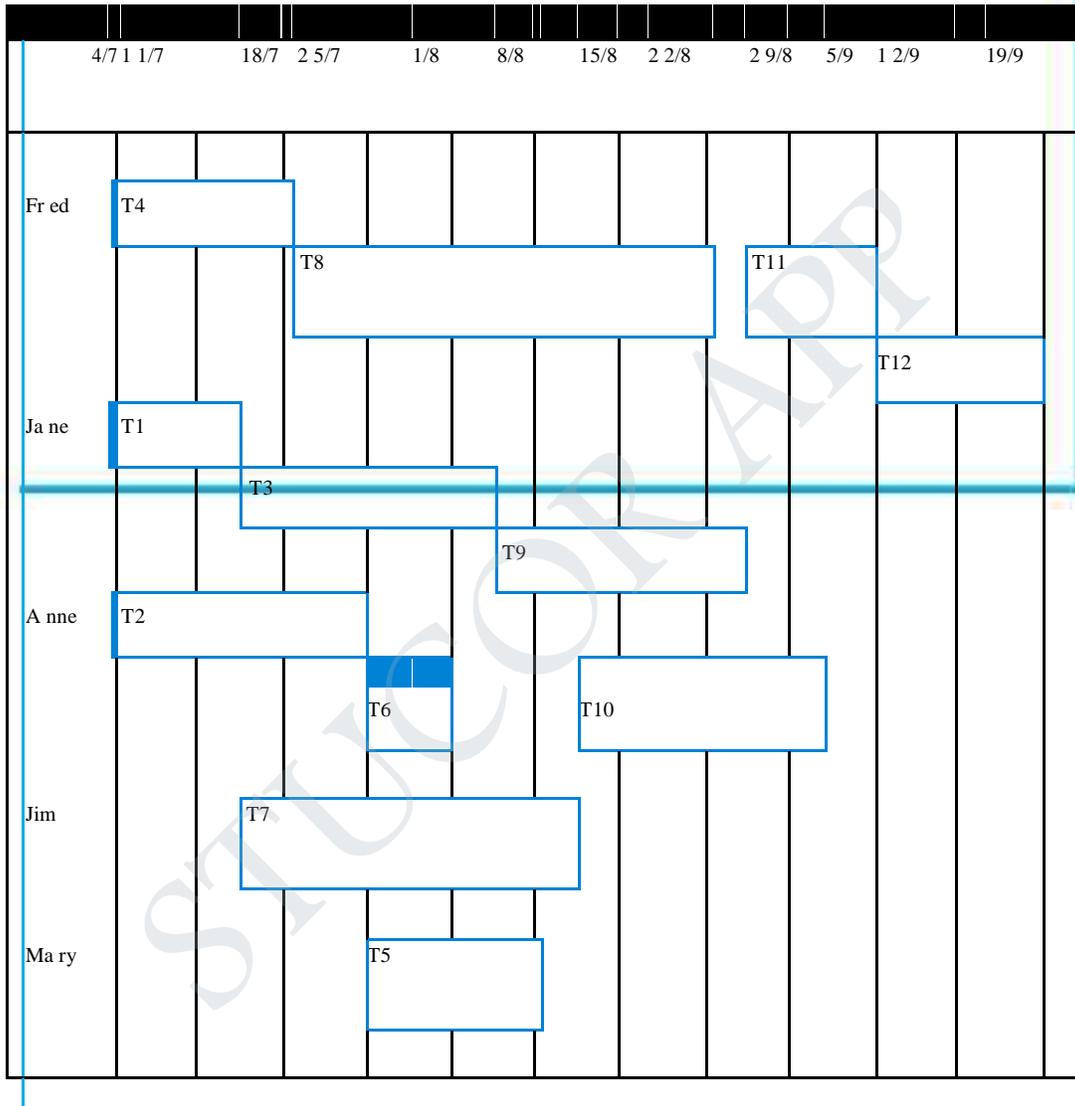


M6

M8

STUCOR APP

Staff allocation



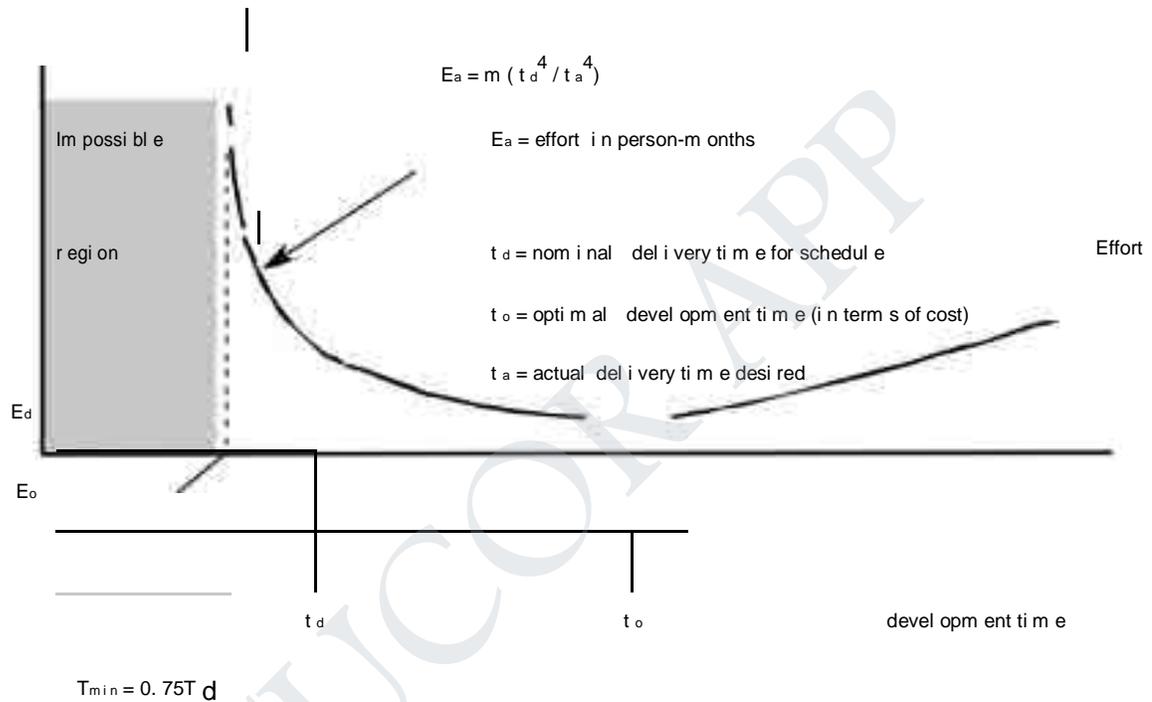
5.9 SCHEDULING, EVA

Scheduling

Scheduling Principles

- compartmentalization—define distinct tasks
- interdependency—indicate task interrelationship
- effort validation—be sure resources are available
- defined responsibilities—people must be assigned
- defined outcomes—each task must have an output
- defined milestones—review for quality

Effort and Delivery Time



Empirical Relationship: P vs E

Given Putnam's Software Equation (5-3),

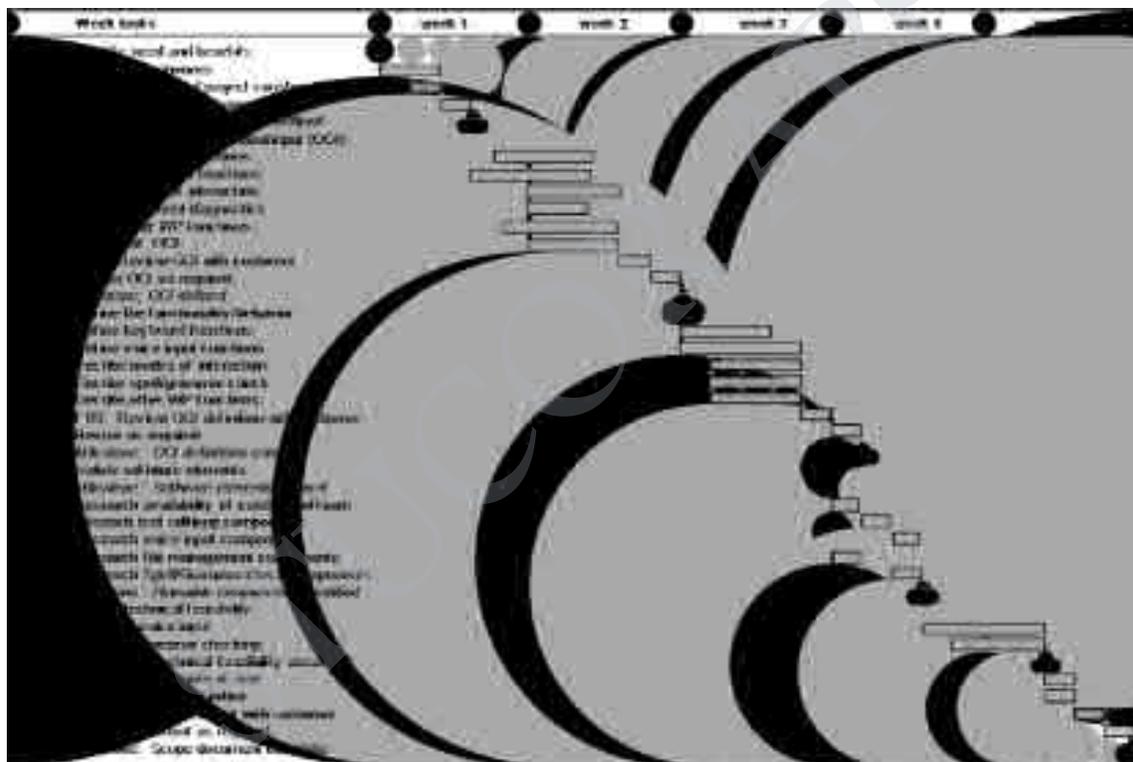
$$E = L^3 / (P^3 t^4)$$

Consider a project estimated at 33 KLOC, 12 person-years of effort, with a P of 10K, the completion time would be 1.3 years

If deadline can be extended to 1.75 years,

$$E = L^3 / (P^3 t^4) \approx 3.8 \text{ p-years vs } 12 \text{ p-years}$$

Timeline Charts



Effort Allocation

- front end activities
 - customer communication
 - analysis
 - design
- review and modification
- construction activities
 - coding or code generation
- testing and installation
 - unit, integration
 - white-box, black box
- regression

Earned Value Analysis

Earned value

- is a measure of progress
- enables us to assess the —percent of completeness of a project using quantitative analysis rather than rely on a gut feeling
- provides accurate and reliable readings of performance from as early as 15 percent into the project.

Computing Earned Value

Budgeted cost of work scheduled (BCWS)

The *budgeted cost of work scheduled* (BCWS) is determined for each work task represented in the schedule.

$BCWS_i$ is the effort planned for work task i .

To determine progress at a given point along the project schedule, the value of BCWS is the sum of the $BCWS_i$ values for all work tasks that should have been completed by that point in time on the project schedule.

The BCWS values for all work tasks are summed to derive the *budget at completion*, BAC. Hence,

$$BAC = \sum (BCWS_k) \text{ for all tasks } k$$

Budgeted cost of work performed (BCWP)

Next, the value for *budgeted cost of work performed* (BCWP) is computed.

The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule. —the distinction between the BCWS and the BCWP is that the former represents the budget of the activities that were planned to be completed and the latter represents the budget of the activities that actually were completed. ||

Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

Schedule performance index, $SPI = BCWP/BCWS$

Schedule variance, $SV = BCWP - BCWS$

SPI is an indication of the efficiency with which the project is utilizing scheduled resources.

Actual cost of work performed, ACWP

Percent scheduled for completion = $BCWS/BAC$

provides an indication of the percentage of work that should have been completed by time t .

Percent complete = $BCWP/BAC$

provides a quantitative indication of the percent of completeness of the project at a given point in time, t .

Actual cost of work performed, ACWP, is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule. It is then possible to compute

Cost performance index, $CPI = BCWP/ACWP$

Cost variance, $CV = BCWP - ACWP$

Problem

Assume you are a software project manager and that you've been asked to computer earned value statistics for a small software project. The project has 56 planned work tasks that are estimated to require 582 person-days to complete. At the time that you've been asked to do the earned value analysis, 12 tasks have been completed. However, the project schedule indicates that 15 tasks should have been completed. The following scheduling data (in person-days) are available:

• Task	Planned Effort	Actual Effort
• 1	12	12.5
• 2	15	11
• 3	13	17
• 4	8	9.5
• 5	9.5	9.0
• 6	18	19
• 7	10	10
• 8	4	4.5
• 9	12	10
• 10	6	6.5
• 11	5	4
• 12	14	14.5
• 13	16	
• 14	6	
• 15	8	

5.10 PROCESS AND PROJECT METRICS

Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficiency of the software process and the projects conducted using the process framework. In software project management, we are primarily concerned with productivity and

quality metrics. There are four reasons for measuring software processes, products, and resources (to characterize, to evaluate, to predict, and to improve).

Process and Project Metrics

- Metrics should be collected so that process and product indicators can be ascertained
- *Process metrics* used to provide indicators that lead to long term process improvement
- *Project metrics* enable project manager to
 - Assess status of ongoing project
 - Track potential risks
 - Uncover problem are before they go critical
 - Adjust work flow or tasks
 - Evaluate the project team's ability to control quality of software wrok products

Process Metrics

- Private process metrics (e.g. defect rates by individual or module) are only known to by the individual or team concerned.
- Public process metrics enable organizations to make strategic changes to improve the software process.
- Metrics should not be used to evaluate the performance of individuals.
- Statistical software process improvement helps and organization to discover where they are strong and where are week.

Statistical Process Control

1. Errors are categorized by their origin
2. Record cost to correct each error and defect
3. Count number of errors and defects in each category
4. Overall cost of errors and defects computed for each category
5. Identify category with greatest cost to organization
6. Develop plans to eliminate the most costly class of errors and defects or at least reduce their frequency

- A software team can use software project metrics to adapt project workflow and technical activities.
- Project metrics are used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis.
- Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables).

Software Measurement

- *Direct process measures* include cost and effort.
- *Direct process measures* include lines of code (LOC), execution speed, memory size, defects reported over some time period.
- *Indirect product measures* examine the quality of the software product itself (e.g. functionality, complexity, efficiency, reliability, maintainability).

Size-Oriented Metrics

- Derived by normalizing (dividing) any direct measure (e.g. defects or human effort) associated with the product or project by LOC.
- Size oriented metrics are widely used but their validity and applicability is widely debated.

Function-Oriented Metrics

- Function points are computed from direct measures of the information domain of a business software application and assessment of its complexity.
- Once computed function points are used like LOC to normalize measures for software productivity, quality, and other attributes.
- The relationship of LOC and function points depends on the language used to implement the software.

Reconciling LOC and FP Metrics

- The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design
- Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost
- Using LOC and FP for estimation a historical baseline of information must be established.

Object-Oriented Metrics

- Number of scenario scripts (NSS)
- Number of key classes (NKC)
- Number of support classes (e.g. UI classes, database access classes, computations classes, etc.)
- Average number of support classes per key class
- Number of subsystems (NSUB)

Use Case-Oriented Metrics

- Describe (indirectly) user-visible functions and features in language independent manner
- Number of use case is directly proportional to LOC size of application and number of test cases needed
- However use cases do not come in standard sizes and use as a normalization measure is suspect
- Use case points have been suggested as a mechanism for estimating effort

WebApp Project Metrics

- Number of static Web pages (N_{sp})
- Number of dynamic Web pages (N_{dp})
- Customization index: $C = N_{sp} / (N_{dp} + N_{sp})$
- Number of internal page links
- Number of persistent data objects
- Number of external systems interfaced
- Number of static content objects
- Number of dynamic content objects
- Number of executable functions

Software Quality Metrics

- Factors assessing software quality come from three distinct points of view (product operation, product revision, product modification).
- Software quality factors requiring measures include
 - correctness (defects per KLOC)
 - maintainability (mean time to change)
 - integrity (threat and security)
 - usability (easy to learn, easy to use, productivity increase, user attitude)
- Defect removal efficiency (DRE) is a measure of the filtering ability of the quality assurance and control activities as they are applied through out the process framework
$$DRE = E / (E + D)$$

E = number of errors found before delivery of work product

D = number of defects found after work product delivery

Integrating Metrics with Software Process

- Many software developers do not collect measures.
- Without measurement it is impossible to determine whether a process is improving or not.
- Baseline metrics data should be collected from a large, representative sampling of past software projects.

Getting this historic project data is very difficult, if the previous developers did not collect data in an on-going manner.