SOFTWARE ENGINEERING

Subject Code: 10IS51 **I.A. Marks : 25**
Hours/Week : 04 **Exam Hours: 03**
Total Hours : 52 **Exam Marks: 100**

PART – A

UNIT – 1 **6 Hours**

Overview: Introduction: FAQ's about software engineering, Professional and ethical responsibility.

Socio-Technical systems: Emergent system properties; Systems engineering; Organizations, people and computer systems; Legacy systems.

UNIT – 2 **6 Hours**

Critical Systems, Software Processes: Critical Systems: A simple safety critical system; System dependability; Availability and reliability.

Software Processes: Models, Process iteration, Process activities; The Rational Unified Process; Computer Aided Software Engineering.

UNIT – 3 **7 Hours**

Requirements: Software Requirements: Functional and Non-functional requirements; User requirements; System requirements; Interface specification; The software requirements document.

Requirements Engineering Processes: Feasibility studies; Requirements elicitation and analysis; Requirements validation; Requirements management.

UNIT – 4 **7 Hours**

System models, Project Management: System Models: Context models; Behavioral models; Data models; Object models; Structured methods.

Project Management: Management activities; Project planning; Project scheduling; Risk management

PART – B

UNIT – 5 **7 Hours**

Software Design: Architectural Design: Architectural design decisions; System organization; Modular decomposition styles; Control styles. 33

Object-Oriented design: Objects and Object Classes; An Object-Oriented design process; Design evolution.

UNIT – 6 **6 Hours**

Development: Rapid Software Development: Agile methods;Extreme programming; Rapid application development.

Software Evolution: Program evolution dynamics; Software maintenance; Evolution processes; Legacy system evolution.

UNIT – 7 **7 Hours**

Verification and Validation: Verification and Validation: Planning; Software inspections; Automated static analysis; Verification and formal methods.

Software testing: System testing; Component testing; Test case design; Test automation.

UNIT – 8 **6 Hours**

Management: Managing People: Selecting staff; Motivating people; Managing people; The People Capability Maturity Model.

Software Cost Estimation: Productivity; Estimation techniques; Algorithmic cost modeling, Project duration and staffing.

Text Book:

1. Ian Sommerville: Software Engineering, 8th Edition, Pearson Education, 2007. (Chapters-: 1, 2, 3, 4, 5, 6, 7, 8, 11, 14, 17, 21, 22, 23, 25, 26)

Reference Books:

1. Roger.S.Pressman: Software Engineering-A Practitioners approach, 7th Edition, Tata McGraw Hill, 2007.
2. Pankaj Jalote: An Integrated Approach to Software Engineering, Wiley India, 2009.

TABLE OF CONTENTS

UNIT -1
OVERVIEW

The economies of ALL developed nations are dependent on software. More and more systems are software controlled.

Software engineering is concerned with theories, methods and tools for professional software development.

FAQs About software engineering:

What is software?
Software is set of Computer programs associated with documentation & configuration data that is needed to make these programs operate correctly. A software system consists of a number of programs, configuration files (used to set up programs), system documentation (describes the structure of the system) and user documentation (explains how to use system).
Software products may be developed for a particular customer or may be developed for a general market.
Software products may be
• Generic - developed to be sold to a range of different customers
• Bespoke (custom) - developed for a single customer according to their specification

What is software engineering?

- Software engineering is an engineering discipline which is concerned with all aspects of software production.
- Software engineers should adopt a systematic and organized approach to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available.

What is the difference between software engineering and computer science?

- Computer science is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software
- Computer science theories are currently insufficient to act as a complete underpinning for software engineering

What is the difference between software engineering and system engineering?

- System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this process
- System engineers are involved in system specification, architectural design, integration and deployment

What is a software process?
A set of activities whose goal is the development or evolution of software
Generic activities in all software processes are:
• Specification - what the system should do and its development constraints
• Development - production of the software system

• Validation - checking that the software is what the customer wants
• Evolution - changing the software in response to changing demands

What is a software process model?
A simplified representation of a software process, presented from a specific perspective
Examples of process perspectives are
• Workflow perspective - sequence of activities
• Data-flow perspective - information flow
• Role/action perspective - who does what

Generic process models
• Waterfall
• Evolutionary development
• Formal transformation
• Integration from reusable components

Socio-Technical Systems:

- A system is a purposeful collection of inter-related components working together towards some common objective.

- A system may include software, mechanical, electrical and electronic hardware and be operated by people.

- System components are dependent on other system components
  The properties and behavior of system components are inextricably inter-mingled

Problems of systems engineering

- Large systems are usually designed to solve 'wicked' problems
- Systems engineering requires a great deal of co-ordination across disciplines
  - Almost infinite possibilities for design trade-offs across components
  - Mutual distrust and lack of understanding across engineering disciplines
- Systems must be designed to last many years in a changing environment

Software and systems engineering

The proportion of software in systems is increasing. Software-driven general purpose electronics is replacing special-purpose systems
Problems of systems engineering are similar to problems of software engineering

Software is seen as a problem in systems engineering. Many large system projects have been delayed because of software problems.

Emergent properties

- Properties of the system as a whole rather than properties that can be derived from the
  properties of components of a system

- Emergent properties are a consequence of the relationships between system components. They can therefore only be assessed and measured once the components have been integrated into a system.

Examples of emergent properties

1. *The overall weight of the system*
• This is an example of an emergent property that can be computed from individual component properties.
2. *The reliability of the system*
• This depends on the reliability of system components and the relationships between the components.
3. *The usability of a system*
• This is a complex property which is not simply dependent on the system hardware and software but also depends on the system operators and the environment where it is used.

Types of emergent property

1. Functional properties
• These appear when all the parts of a system work together to achieve some objective. For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.

2. Non-functional emergent properties
• Examples are reliability, performance, safety, and security.
These relate to the behaviour of the system in its operational environment. They are often critical for computer-based systems as failure to achieve some minimal defined level in
these properties may make the system unusable.

Because of component inter-dependencies, faults can be propagated through the system
System failures often occur because of unforeseen inter-relationships between Components It is probably impossible to anticipate all possible component relationships
Software reliability measures may give a false picture of the system reliability

System reliability engineering

1. *Hardware reliability*
• What is the probability of a hardware component failing and how long does it take to repair that component?
2. *Software reliability*
• How likely is it that a software component will produce an incorrect output.
Software failure is usually distinct from hardware failure in that software does not wear out.
3. *Operator reliability*
• How likely is it that the operator of a system will make an error?
Influences on reliability

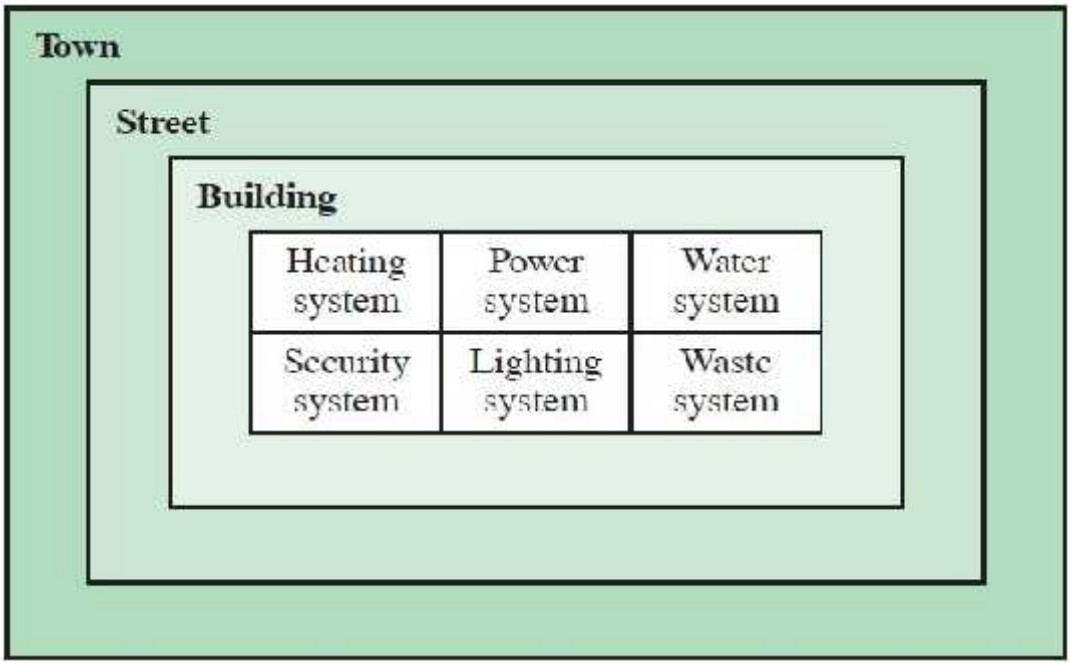
Reliability relationships

1. Hardware failure can generate spurious signals that are outside the range of inputs expected by the software
2. Software errors can cause alarms to be activated which cause operator stress and lead to operator errors

3. The environment in which a system is installed can affect its reliability

Systems and their environment

Systems are not independent but exist in an environment
System's function may be to change its environment. Environment affects the functioning of the system e.g. system may require electrical supply from its environment
The organizational as well as the physical environment may be important



Human and organisational factors

Process changes
• Does the system require changes to the work processes in the environment?
Job changes
• Does the system de-skill the users in an environment or cause them to change the way they work?
Organisational changes
• Does the system change the political power structure in an organisation?

System architecture modelling

An architectural model presents an abstract view of the sub-systems making up a system

may include major information flows between sub-systems

- Usually presented as a block diagram
- May identify different types of functional component in the model

System evolution

Large systems have a long lifetime. They must evolve to meet changing requirements

Evolution is inherently costly

- Changes must be analysed from a technical and business perspective
- Sub-systems interact so unanticipated problems can arise
- There is rarely a rationale for original design decisions
- System structure is corrupted as changes are made to it

Existing systems which must be maintained are sometimes called legacy systems

The system engineering process

Usually follows a 'waterfall' model because of the need for parallel development of different parts of the system

- Little scope for iteration between phases because hardware

changes are very expensive. Software may have to compensate for hardware problems

Inevitably involves engineers from different disciplines who must work together

- Much scope for misunderstanding here. Different disciplines use a

different vocabulary and much negotiation is required. Engineers may have personal agendas to fulfill.

UNIT-2
CRITICAL SYSTEMS, SOFTWARE PROCESSES

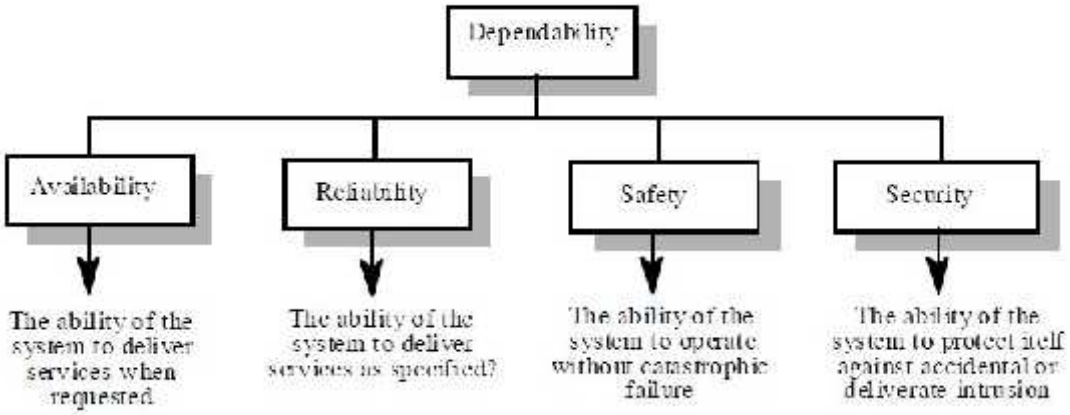
Critical Systems

- For critical systems, it is usually the case that the most important system property is the dependability of the system.

- The dependability of a system reflects the user's degree of trust in that system. It reflects the extent of the user's confidence that it will operate as users expect and that it will not 'fail' in normal use.

- Usefulness and trustworthiness are not the same thing. A system does not have to be trusted to be useful

Dimensions of dependability



The software process

A software process is a structured set of activities required to develop a software system

It involves the following phases:

• 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.

Software process models

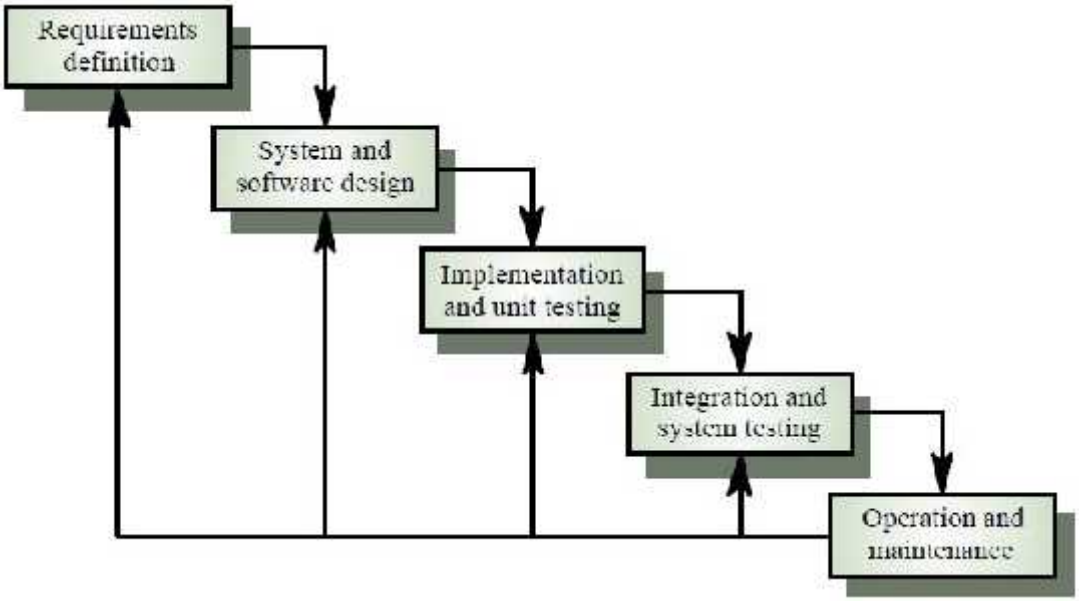
1. The waterfall model
• Separate and distinct phases of specification and development

2. Evolutionary development
• Specification and development are interleaved

3. Formal systems development
• A mathematical system model is formally transformed to an Implementation

4. Reuse-based development
• The system is assembled from existing components

Waterfall model



The different phases in waterfall model are :

- Requirements analysis and definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance

The drawback of the waterfall model is the difficulty of accommodating change after the process is underway.

Waterfall model problems

- Inflexible partitioning of the project into distinct stages
- This makes it difficult to respond to changing customer requirements

This model is only appropriate when the requirements are well-understood.

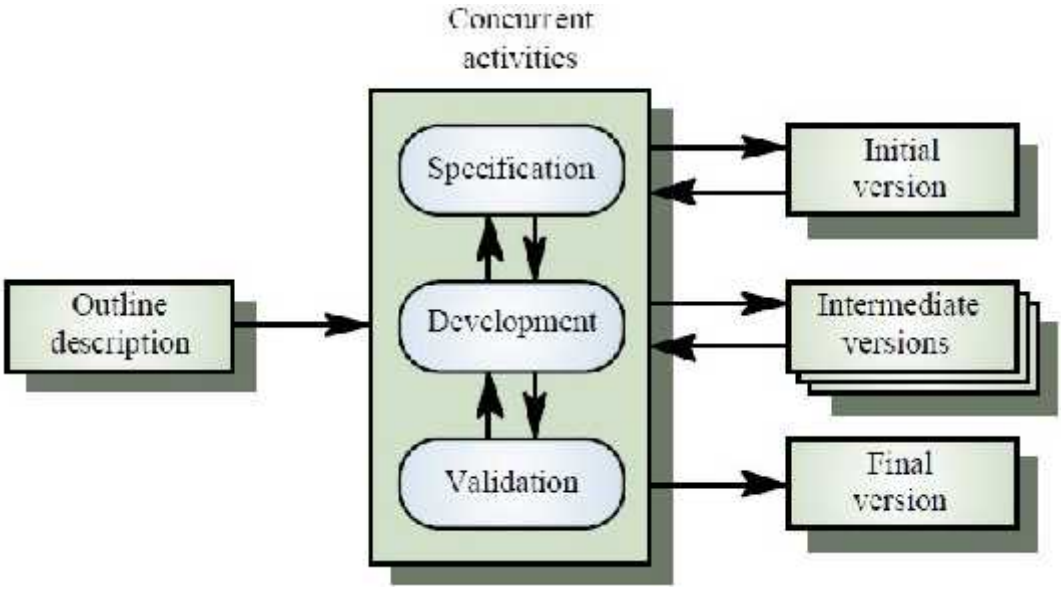
Evolutionary development

There are 2 types :

1. Exploratory development

• Objective is to work with customers and to evolve a final system from an initial outline
specification. Should start with well-understood requirements

2.Throw-away prototyping
• Objective is to understand the system requirements. Should start
with poorly understood requirements



Problems

- Lack of process visibility
- Systems are often poorly structured
- Special skills (e.g. in languages for rapid prototyping) may be required

Applicability

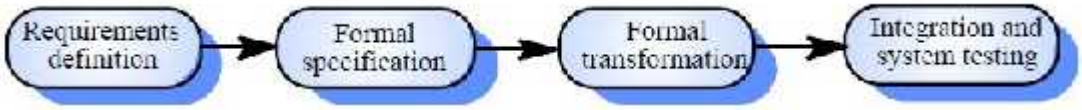
• For small or medium-size interactive systems
• For parts of large systems (e.g. the user interface)
• For short-lifetime systems

Formal systems development

It is based on the transformation of a mathematical specification through different representations to an executable program.

Transformations are 'correctness-preserving' so it is straightforward to show that the program conforms to its specification.

It is embodied in the 'Cleanroom' approach to software development.



Problems

- Need for specialised skills and training to apply the technique
- Difficult to formally specify some aspects of the system such as the user interface

Applicability

- Critical systems especially those where a safety or security case must be made before the system is put into operation

Reuse-oriented development

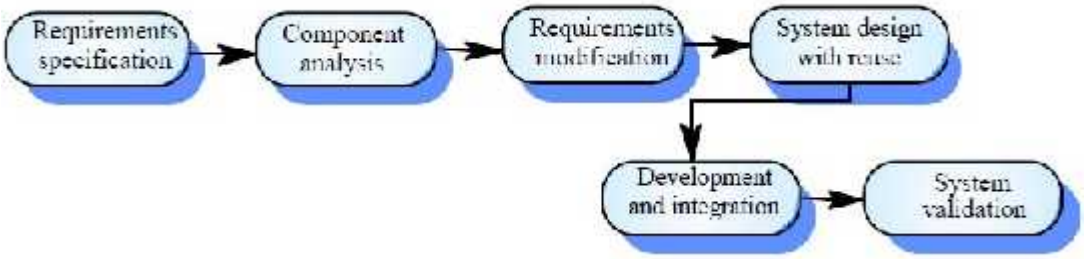
It is based on systematic reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems

L

Process stages

- Component analysis
- Requirements modification
- System design with reuse
- Development and integration

This approach is becoming more important but still limited experience with it



Process iteration

System requirements ALWAYS evolve in the course of a project so process iteration where earlier stages are reworked is always part of the process for large systems
Iteration can be applied to any of the generic process models

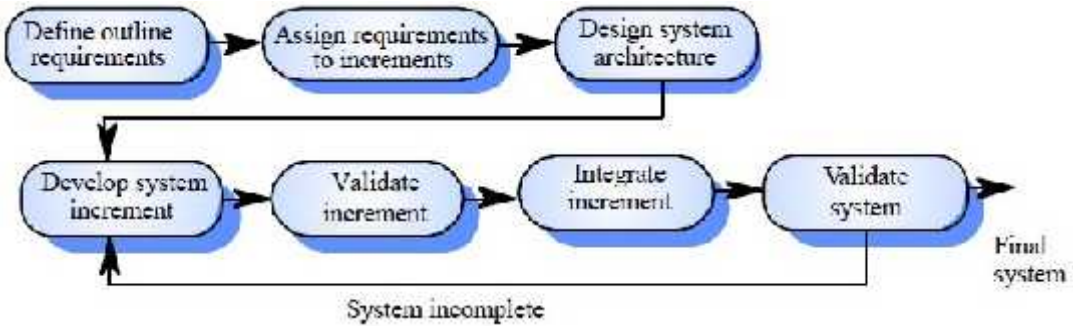
Two (related) approaches

- Incremental development
- Spiral development

Incremental development

Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.

User requirements are prioritised and the highest priority requirements are included in early increments.
Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.



Advantages

- Customer value can be delivered with each increment so system functionality is available earlier
- Early increments act as a prototype to help elicit requirements for later increments
- Lower risk of overall project failure
- The highest priority system services tend to receive the most testing

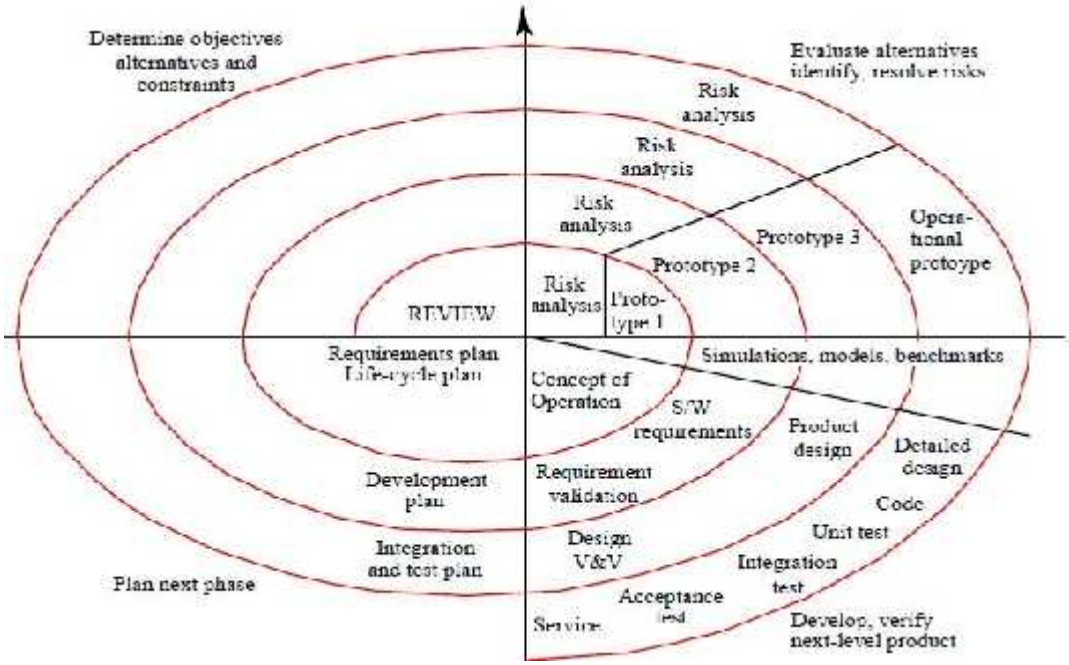
Spiral development

Process is represented as a spiral rather than as a sequence of activities with backtracking

Each loop in the spiral represents a phase in the process.

No fixed phases such as specification or design -loops in the spiral are chosen depending on what is required.

Risks are explicitly assessed and resolved throughout the process.



CASE

Computer-aided software engineering (CASE) is software to support software development and evolution processes.

Activity automation
• Graphical editors for system model development
• Data dictionary to manage design entities
• Graphical UI builder for user interface construction
• Debuggers to support program fault finding
• Automated translators to generate new versions of a program

Case technology

Case technology has led to significant improvements in the software process though not
the order of magnitude improvements that were once predicted

• Software engineering requires creative thought - this is not
readily automatable
• Software engineering is a team activity and, for large projects,
much time is spent in team interactions. CASE technology does
not really support these

CASE classification

Classification helps us understand the different types of CASE tools and their support for process activities.

1. Functional perspective
• Tools are classified according to their specific function

2. Process perspective
• Tools are classified according to process activities that are supported

3. Integration perspective
• Tools are classified according to their organisation into integrated units

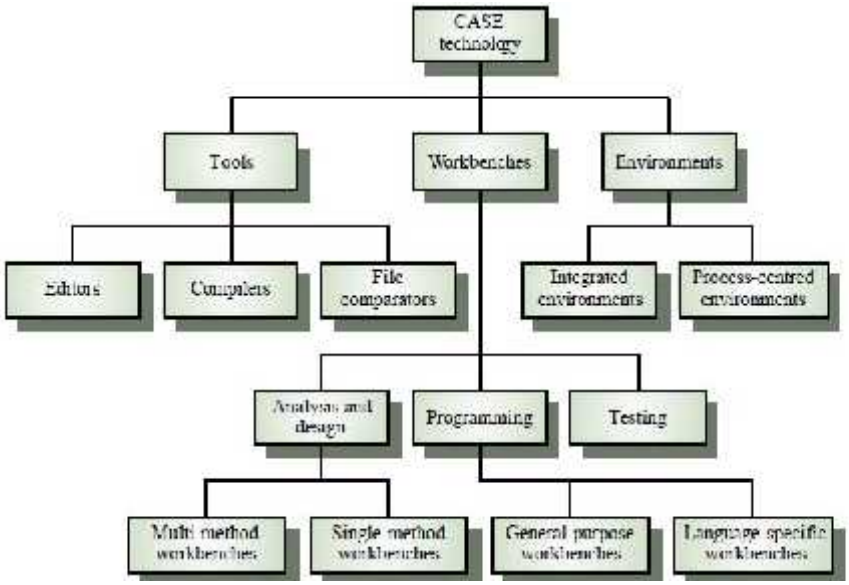
CASE integration

Tools
• Support individual process tasks such as design consistency checking, text editing, etc.

Workbenches
• Support a process phase such as specification or design, Normally include a number of integrated tools

Environments
• Support all or a substantial part of an entire software process.
Normally include several integrated workbenches

Tools, workbenches, environments



UNIT -3
REQUIREMENTS

Requirements

Requirement - Descriptions and specifications of a system.

Requirements engineering
- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.

Requirement : A requirement may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.

Requirements serve a dual function :
• May be the basis for a bid for a contract - therefore must be open to interpretation
• May be the basis for the contract itself - therefore must be defined in detail
• Both these statements may be called requirements

Functional and non-functional requirements

Definitions

Functional requirements : Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.

Non-functional requirements : Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.

Domain requirements : Requirements that come from the application domain of the system and that reflect characteristics of that domain.

Detailed descriptions

Functional requirements

- They Describe functionality or system services
- Depend on the type of software, expected users and the type of system where the software is used
- Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail

Examples (The requirements can be defined as follows)

- 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.

Non-functional requirements

- They define system properties and constraints like reliability, response time and storage requirements.
- Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method

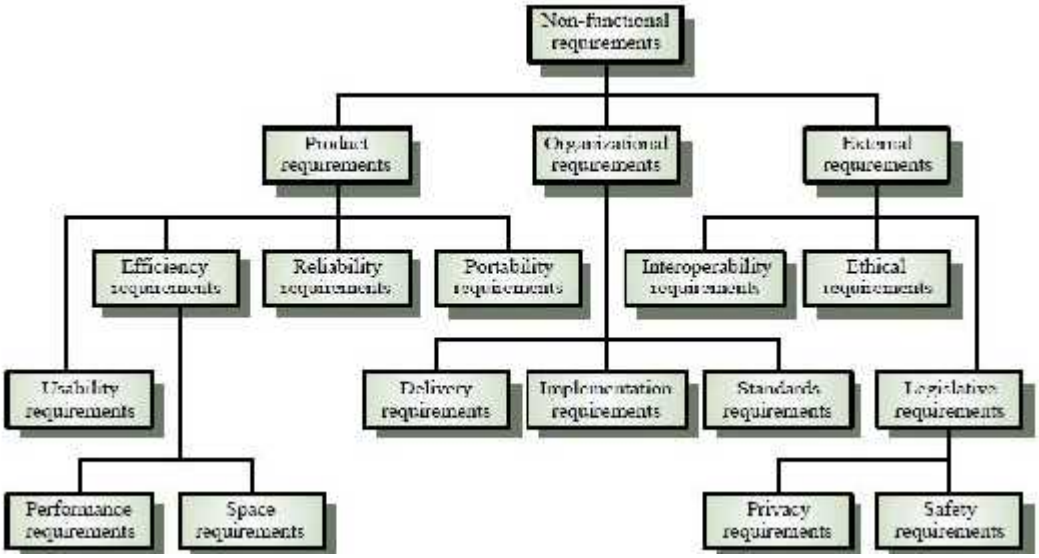
Non-functional requirements may be more critical than functional requirements. If these are not met, the system becomes useless.

Non-functional classifications

1. Product requirements
These requirements specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

2. Organizational requirements
• Requirements which are a consequence of organizational policies and procedures e.g.
process standards used, implementation requirements, etc.

3. External requirements
• Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.



Types of requirement

1. User requirements
• Statements in natural language plus diagrams of the services the system provides and its
operational constraints. Written for customers
2. System requirements
• A structured document setting out detailed descriptions of the system services. Written
as a contract between client and contractor
3. Software specification
• A detailed software description which can serve as a basis for a design
or implementation. These set of requirements are written for developers

User requirements
- Should describe functional and non-functional requirements so that they are understandable by system users who don't have detailed technical knowledge
- User requirements are defined using natural language, tables and diagrams

Some of the problems with natural language
1. Lack of clarity
   • Precision is difficult without making the document difficult to read
2. Requirements confusion
   • Functional and non-functional requirements tend to be mixed-up
3. Requirements amalgamation
   • Several different requirements may be expressed together

System requirements
- More detailed specifications of user requirements
- Serve as a basis for designing the system
- May be used as part of the system contract
- System requirements may be expressed using system models

Interface specification

Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements

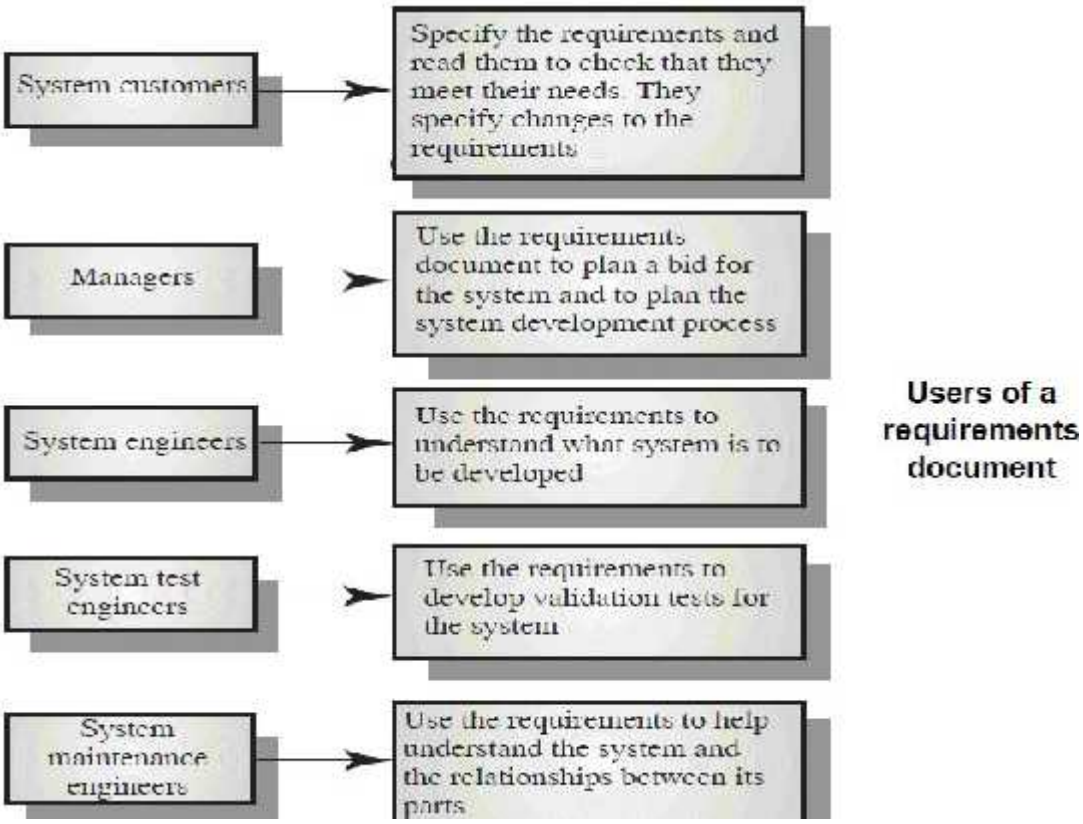
Three types of interface may have to be defined
- Procedural interfaces
- Data structures that are exchanged
- Data representations

Formal notations are an effective technique for interface specification

The requirements document

The requirements document is the official statement of what is required of the system developers. It should include both a definition and a specification of requirements

The requirements document is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.



Requirements document requirements – The requirement doc should have the following :

- 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

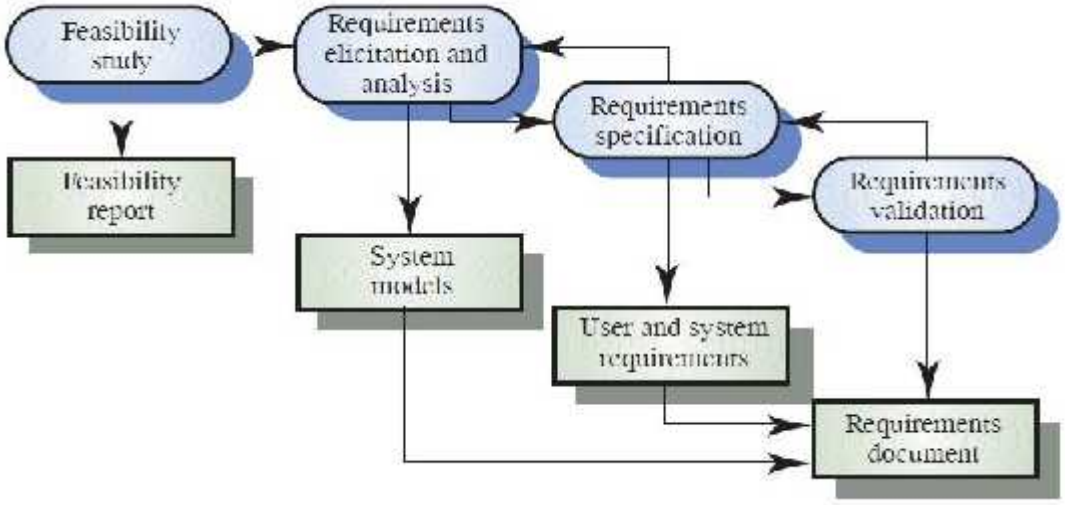
Requirements document structure - These are the various contents that the req doc should possess :

- Introduction
- Glossary
- User requirements definition
- System architecture
- System requirements specification
- System models
- System evolution
- Appendices
- Index

Requirements engineering processes

The processes used for RE vary widely depending on the application domain, the people
involved and the organization developing the requirements
These are some of the generic activities common to all processes

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



Feasibility studies

A feasibility study decides whether or not the proposed system is worthwhile
It is 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

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 and the system's operational constraints

- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*

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 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

Requirements change

- The priority of requirements from different viewpoints changes during the development process
- System customers may specify requirements from a business perspective that conflict with end-user requirements
- The business and technical environment of the system changes during its development

UNIT 4

SYSTEM MODELS, PROJECT MANAGEMENT

System models

System modeling : System modeling helps the analyst to understand the functionality of the system and models are used to communicate with customers

Different models present the system from different perspectives
• External perspective showing the system's context or environment
• Behavioral perspective showing the behavior of the system
• Structural perspective showing the system or data architecture

Structured methods
- Structured methods incorporate system modeling as an inherent part of the method
- Methods define a set of models, a process for deriving these models and rules and guidelines that should apply to the models
- CASE tools support system modeling as part of a structured method

Context models
- Context models are used to illustrate the boundaries of a system
- Social and organizational concerns may affect the decision on where to position system boundaries
- Architectural models show the a system and its relationship with other systems

Process models
- Process models show the overall process and the processes that are supported by the system
- Data flow models may be used to show the processes and the flow of information from one process to another

Behavioural models
- 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

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

Object models
- Object models describe the system in terms of object classes

- An object class is an abstraction over a set of objects with common attributes and the services (operations) provided by each object
- Various object models may be produced

• Inheritance models
• Aggregation models
• Interaction models

Object models

- Natural ways of reflecting the real-world entities manipulated by the system
- More abstract entities are more difficult to model using this approach
- Object class identification is recognised as a difficult process requiring a deep understanding of the application domain
- Object classes reflecting domain entities are reusable across systems

The Unified Modeling Language

- Devised by the developers of widely used objectoriented analysis and design methods
- Has become an effective standard for objectoriented modelling
- Notation

• Object classes are rectangles with the name at the top, attributes In he middle section
and operations in the bottom section
• Relationships between object classes (known as associations) are shown as lines linking objects

• Inheritance is referred to as generalisation and is shown‘upwards’ rather than ‘downwards’ in a hierarchy

Project Management

It is concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organisations developing
and procuring the software

Project management is needed because software development is always subject to budget and schedule constraints that are set by the organisation developing the software

Software management distinctions

- The product is intangible
- The product is uniquely flexible
- Software engineering is not recognized as an engineering discipline with the same status as mechanical, electrical engineering, etc.
- The software development process is not standardised
- Many software projects are 'one-off' projects

Management activities

- Proposal writing includes Feasibility, Project costing, Overall requirements (Internal and External), terms and conditions
- Resource requirements also include Personnel selection

- Project planning and scheduling
- Project monitoring and reviews also including Personnel and Process evaluation
- Report writing and presentations

Project staffing involves the following

- May not be possible to appoint the ideal people to work on a project
- Project budget may not allow for the use of highlypaid staff
- Staff with the appropriate experience may not be available
- An organization may wish to develop employee skills on a software project

Managers have to work within these constraints especially when (as is currently the case) there is an international shortage of skilled IT staff

Project planning

- Probably the most time-consuming project management activity
- Continuous activity from initial concept through to system delivery. Plans must be regularly revised as new information becomes available
- Various different types of plan may be developed to support the main software project plan that is concerned with schedule and budget

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.
Staff development plan.	Describes how the skills and experience of the project team members will be developed.

Project plan structure – It should include the following:

- Introduction
- Project organization
- Risk analysis
- Hardware and software resource requirements
- Work breakdown
- Project schedule
- Monitoring and reporting mechanisms

Activity organization

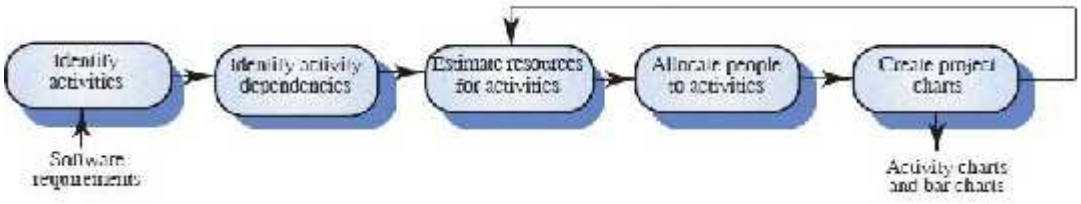
- Activities in a project should be organized to produce tangible outputs for management to judge progress

- *Milestones* are the end-point of a process activity
- *Deliverables* are project results delivered to customers at the end of some major project phase such as specification or design
- The waterfall process allows for the straightforward definition of progress milestones

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

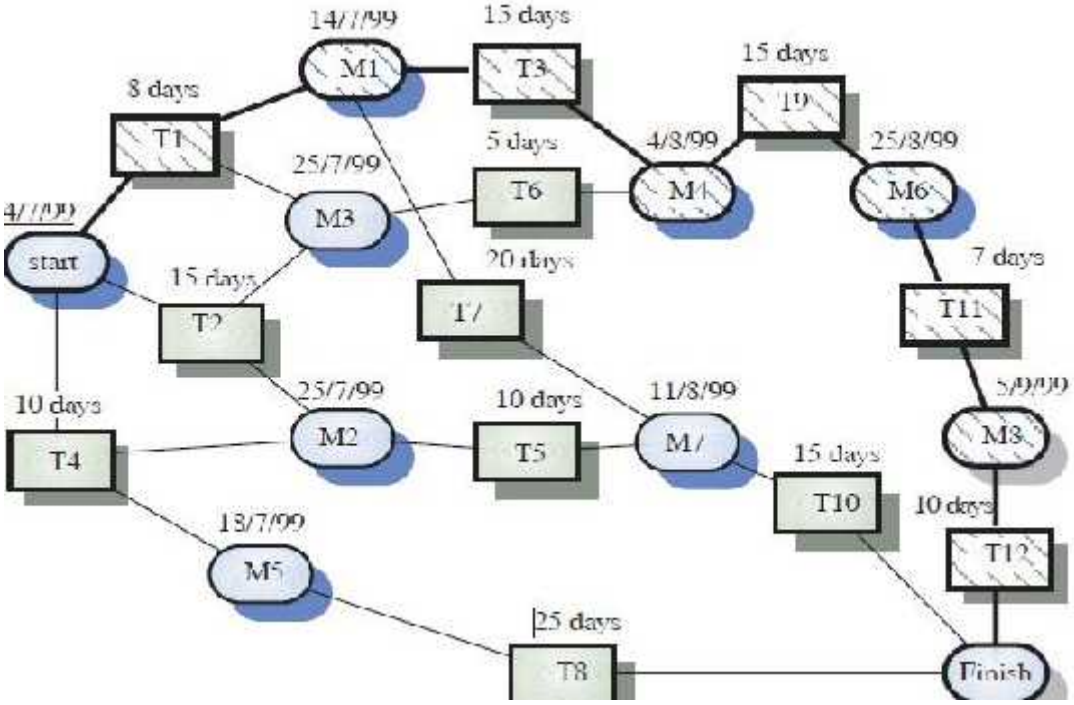
The project scheduling process



Scheduling problems

- Estimating the difficulty of problems and hence the cost of developing a solution is hard
- Productivity is not proportional to the number of people working on a task
- Adding people to a late project makes it later because of communication overheads
- The unexpected always happens. Always allow contingency in planning

Activity network



Risk management

- Risk management is concerned with identifying risks and drawing up plans to minimize their effect on a project.
- A risk is a probability that some adverse circumstance will occur.
  - Project risks affect schedule or resources
  - Product risks affect the quality or performance of the software being developed
  - Business risks affect the organization developing or procuring the software

The 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
  - Monitor the risks throughout the project

Risk identification

- Technology risks
- People risks
- Organizational risks
- Requirements risks
- Estimation risks

Risk analysis

- Assess probability and seriousness of each risk
- Probability may be very low, low, moderate, high or very high

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

Risk planning

- Consider each risk and develop a strategy to manage that risk
- Avoidance strategies
  - The probability that the risk will arise is reduced
- Minimization strategies
  - The impact of the risk on the project or product will be reduced
- Contingency plans
  - If the risk arises, contingency plans are plans to deal with that risk

UNIT –5
SOFTWARE DESIGN

Software Design

Architectural Design

- Establishing the overall Structure of a software system.

Objectives

- To introduce software engineering and to explain its importance
- To set out the answers to key questions about software engineering
- To introduce ethical and professional issues and to explain why they are of concern to software engineers

Software architecture

- The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is the architectural design
- The output of this design process is a description of the software architecture

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 or not.
- Large-scale reuse: The architecture may be reusable across a range of systems

Architectural Design Decisions

Architectural design process

System structuring: The system is decomposed into several principal sub-systems and communications between these sub-systems are identified.

Control modeling: A model of the control relationships between the different parts of the system is established.

Modular decomposition: The **identified sub-systems are** decomposed into modules

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

Architectural models

- Different architectural models may be produced during the design process
- Each model presents different perspectives on them architecture
- 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

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

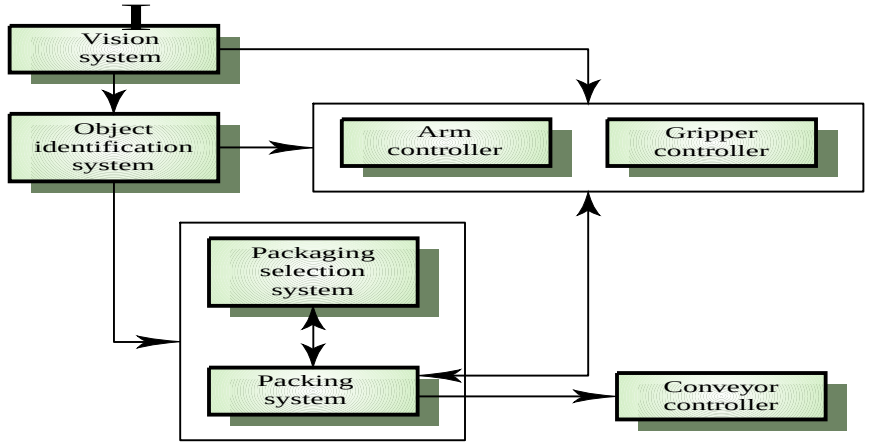
Architecture attributes

- Performance: Localize operations to minimize sub-system communication
- Security: Use a layered architecture with critical assets in inner layers
- Safety: Isolate safety-critical components
- Availability: Include redundant components in the architecture
- Maintainability: Use fine-grain, self-contained components

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

Packing robot control system

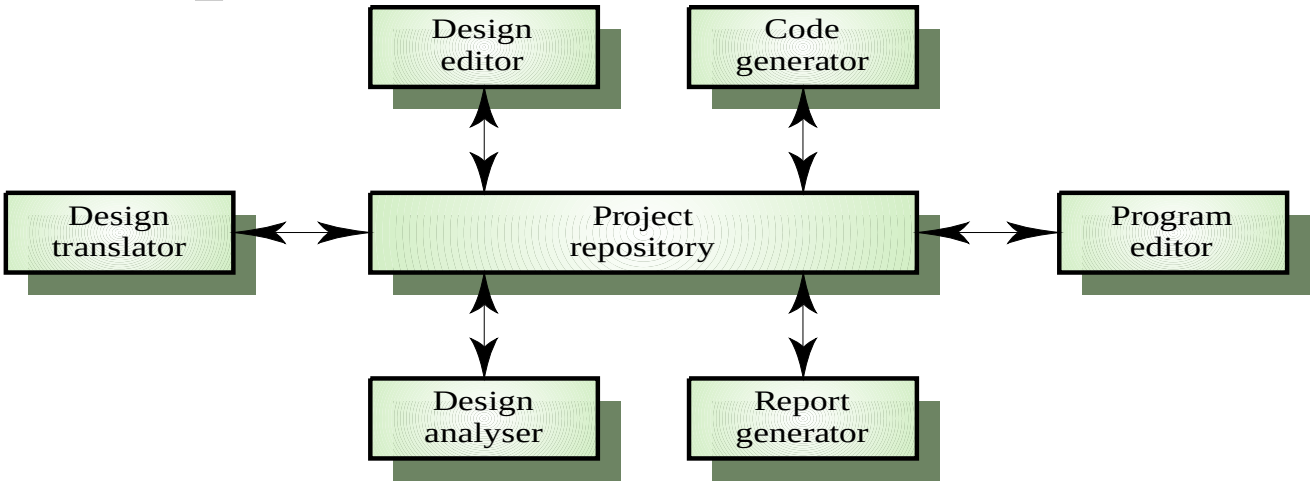


System Organization

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
- Centralized management e.g. backup, security, etc.
- Sharing model is published as the repository schema

Disadvantages

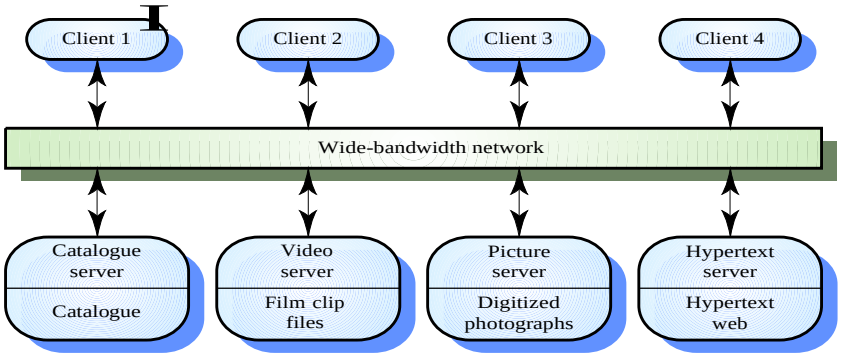
- Sub-systems must agree on a repository data model. Inevitably a compromise
- Data evolution is difficult and expensive
- No scope for specific management policies
- Difficult to distribute efficiently

Client-server architecture

- 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

Film and picture library



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

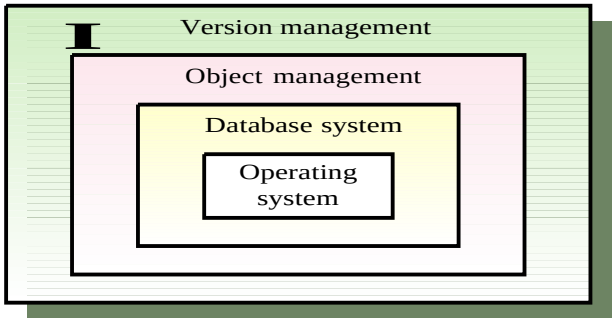
Disadvantages

- No shared data model so sub-systems use different data organization. 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 model

- Used to model the interfacing of sub-systems
- Organizes 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 difficult to structure systems in this way

Version management system



Control Styles

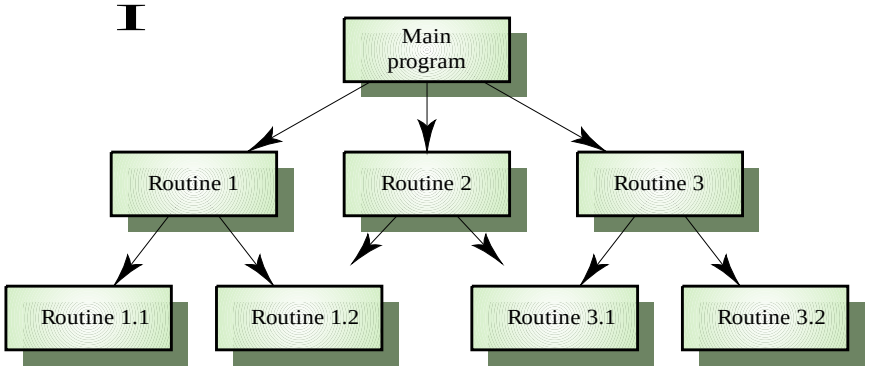
Control models: Are concerned with the control flow between sub-systems. Distinct from the system decomposition model

- Centralized control: One sub-system has overall responsibility for control and starts and stops other sub-systems
- Event-based control: Each sub-system can respond to externally generated events from other sub-systems or the system's environment

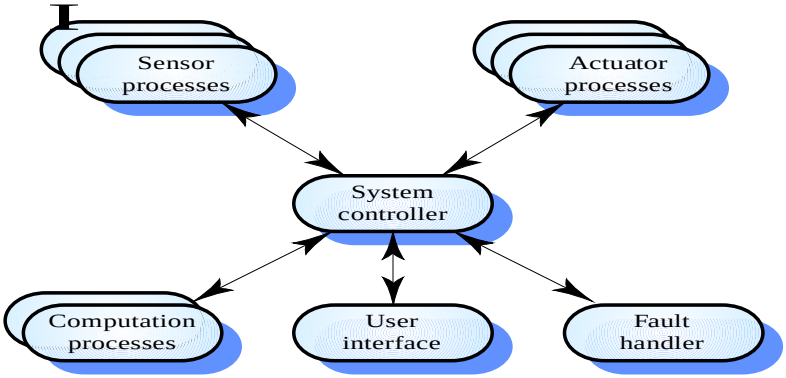
Centralized control

- A control sub-system takes responsibility for managing the execution of other sub-systems
- Call-return model: Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards. Applicable to sequential systems
- Manager model: Applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. Can be implemented in sequential systems as a case statement

Call-return model



Real-time system control



Event-driven systems:

Driven by externally generated events where the timing of the event is outwit the control of the sub-systems which process the event.

Two principal event-driven models

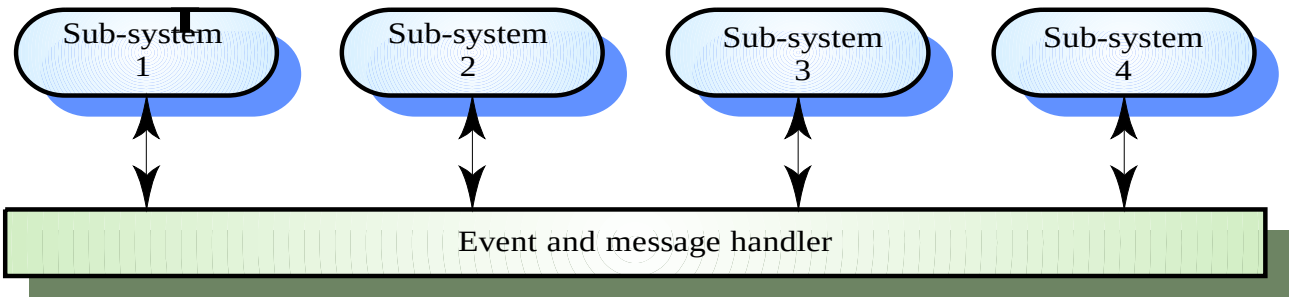
- Broadcast models. An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so.
- Interrupt-driven models. Used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing

Other event driven models include spreadsheets and production systems

Broadcast model

- Effective in integrating sub-systems on different computers in a network
- Sub-systems register an interest in specific events. When these occur, control is transferred to the sub-system which can handle the event
- Control policy is not embedded in the event and message handler. Sub-systems decide on events of interest to them
- However, sub-systems don't know if or when an event will be handled

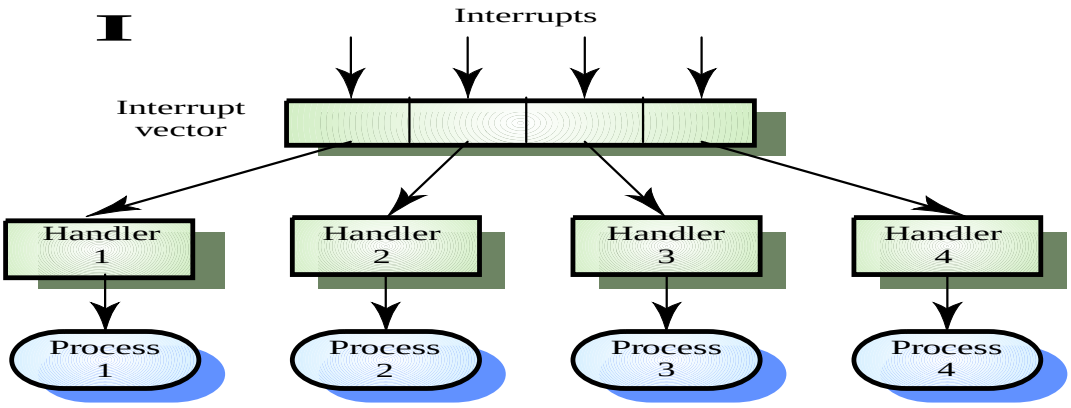
Selective broadcasting



Interrupt-driven systems

- Used in real-time systems where fast response to an event is essential
- There are known interrupt types with a handler defined for each type
- Each type is associated with a memory location and a hardware switch causes transfer to its handler
- Allows fast response but complex to program and difficult to validate

Interrupt-driven control



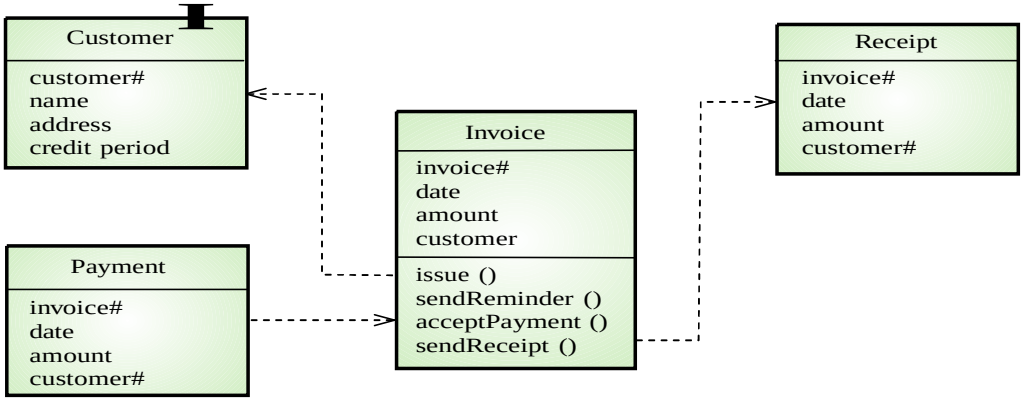
Modular decomposition Styles

- 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 objects
- A data-flow model where the system is decomposed into functional modules which transform inputs to outputs. Also known as the pipeline model
- 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



Data-flow models

- 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

Domain-specific architectures

- Architectural models which are specific to some application domain
- Two types of domain-specific model
- Generic models which are abstractions from a number of real systems and which encapsulate the principal characteristics of these systems
- Reference models which are more abstract, idealized model. Provide a means of information about that class of system and of comparing different architectures
- Generic models are usually bottom-up models; Reference models are top-down models

Object-oriented Design

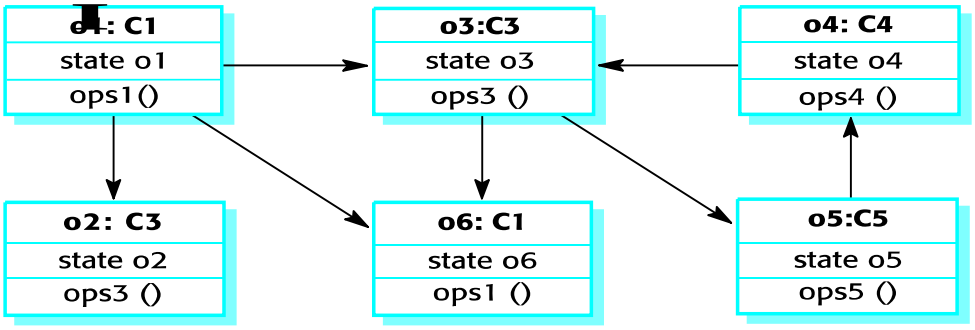
- Designing systems using self- contained objects and object classes

Characteristics of OOD

- Objects are abstractions of real-world or system entities and manage themselves
- Objects are independent and encapsulate state and representation information.
- System functionality is expressed in terms of object services

- Shared data areas are eliminated. Objects communicate by message passing
- Objects may be distributed and may execute sequentially or in parallel.

Interacting objects



Advantages of OOD

- Easier maintenance. Objects may be understood as stand-alone entities
- Objects are appropriate reusable components.
- For some systems, there may be an obvious mapping from real world entities to system objects

Object-oriented development

- Object-oriented analysis, design and programming are related but distinct
- OOA is concerned with developing an object model of the application domain
- OOD is concerned with developing an object-oriented system model to implement requirements
- OOP is concerned with realizing an OOD using an OO programming language such as Java or C++

Objects and object classes

- Objects are entities in a software system which represent instances of real-world and system entities
- Object classes are templates for objects. They may be used to createobjects
- Object classes may inherit attributes and services from other object classes

Objects

- An Object is an entity which has a state and a defined set of operations which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) which request these services when some computation is required.

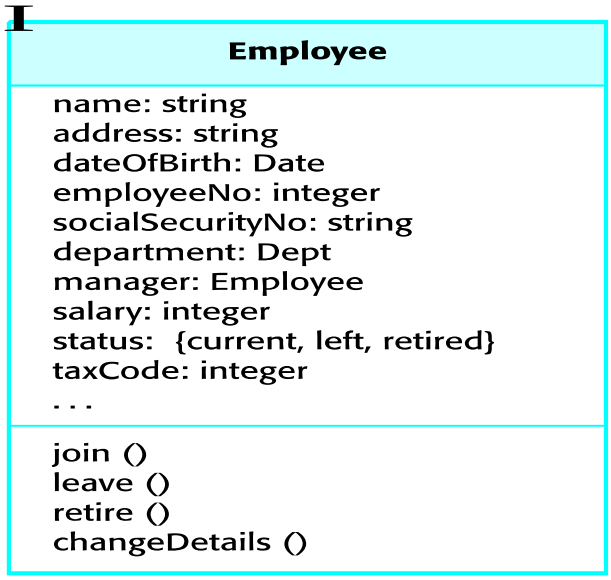
- Objects are created according to some object class definition. An object class definition serves as a template for objects. It includes declarations of

all the attributes and services which should be associated with an object of that class.

The Unified Modeling Language

- Several different notations for describing object-oriented designs were proposed in the 1980s and 1990s
- The Unified Modeling Language is an integration of these notations
- It describes notations for a number of different models that may be produced during OO analysis and design
- It is now a de facto standard for OO modeling

Employee object class (UML)



Object communication

- Conceptually, objects communicate by message passing.
- Messages
- The name of the service requested by the calling object.
- Copies of the information required to execute the service and the name of a holder for the result of the service.
- In practice, messages are often implemented by procedure calls
- Name = procedure name.
- Information = parameter list.

Message examples

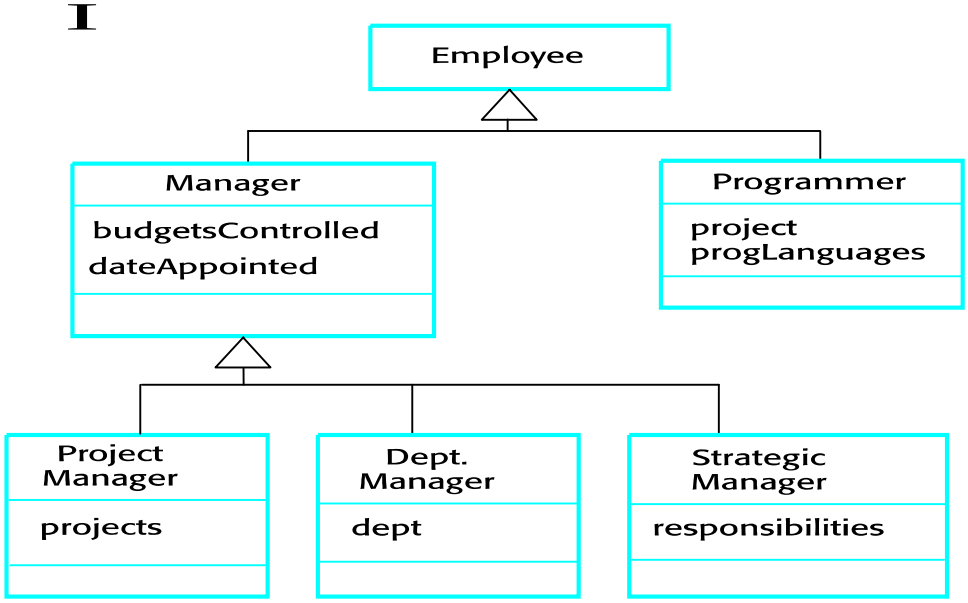
```
// Call a method associated with a buffer
// object that returns the next value
//   in the buffer
v = circularBuffer.Get ();
// Call the method associated with a
// thermostat object that sets the
```

```
// temperature to be maintained
thermostat.setTemp (20);
```

Generalization and inheritance

- Objects are members of classes which define attribute types and operations
- Classes may be arranged in a class hierarchy where one class (a super-class) is a generalization of one or more other classes (sub-classes)
- A sub-class inherits the attributes and operations from its super class and may add new methods or attributes of its own
- Generalization in the UML is implemented as inheritance in OO programming languages

A generalization hierarchy



Advantages of inheritance

- It is an abstraction mechanism which may be used to classify entities
- It is a reuse mechanism at both the design and the programming level
- The inheritance graph is a source of organizational knowledge about domains and systems

Problems with inheritance

- Object classes are not self-contained. they cannot be understood without reference to their super-classes
- Designers have a tendency to reuse the inheritance graph created during analysis. Can lead to significant inefficiency
- The inheritance graphs of analysis, design and implementation have different functions and should be separately maintained

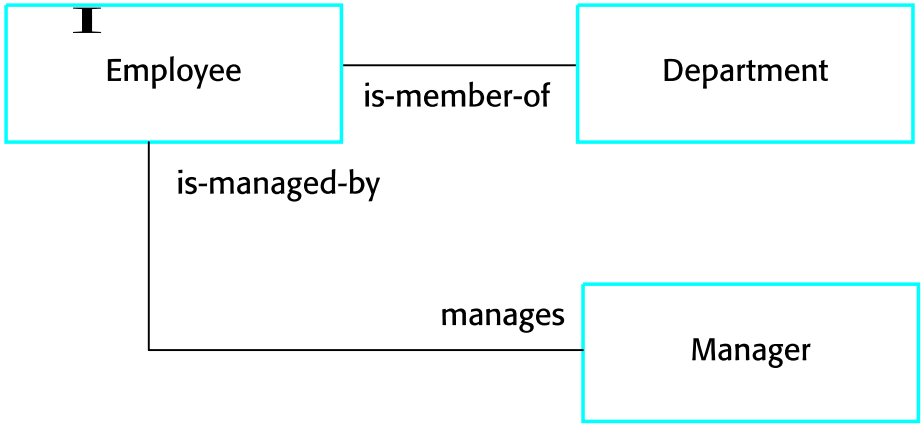
Inheritance and OOD

- There are differing views as to whether inheritance is fundamental to OOD.
- View 1. Identifying the inheritance hierarchy or network is a fundamental part of object-oriented design. Obviously this can only be implemented using an OOPL.
- View 2. Inheritance is a useful implementation concept which allows reuse of attribute and operation definitions. Identifying an inheritance hierarchy at the design stage places unnecessary restrictions on the implementation
- Inheritance introduces complexity and this is undesirable, especially in critical systems

UML associations

- Objects and object classes participate in relationships with other objects and object classes
- In the UML, a generalized relationship is indicated by an association
- Associations may be annotated with information that describes the association Associations are general but may indicate that an attribute of an object is an associated object or that a method relies on an associated object

An association model




```
Coords c1, c2 ;
Satellite sat1, sat2 ;
Navigator theNavigator ;
public Position givePosition ()
{
return currentPosition ;
}
```

Concurrent objects

- The nature of objects as self-contained entities make them suitable for concurrent implementation
- The message-passing model of object communication can be implemented directly if objects are running on separate processors in a distributed system

Servers and active objects

- Servers: The object is implemented as a parallel process (server) with entry points corresponding to object operations. If no calls are made to it, the object suspends itself and waits for further requests for service
- Active objects: Objects are implemented as parallel processes and the internal object state may be changed by the object itself and not simply by external calls.

Object- Oriented design process

Active transponder object

- Active objects may have their attributes modified by operations but may also update them autonomously using internal operations
- Transponder object broadcasts an aircraft's position. The position may be updated using a satellite positioning system. The object periodically update the position by triangulation from satellites.

An active transponder object

```
class Transponder extends Thread {
Position currentPositi on ;

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

Java threads

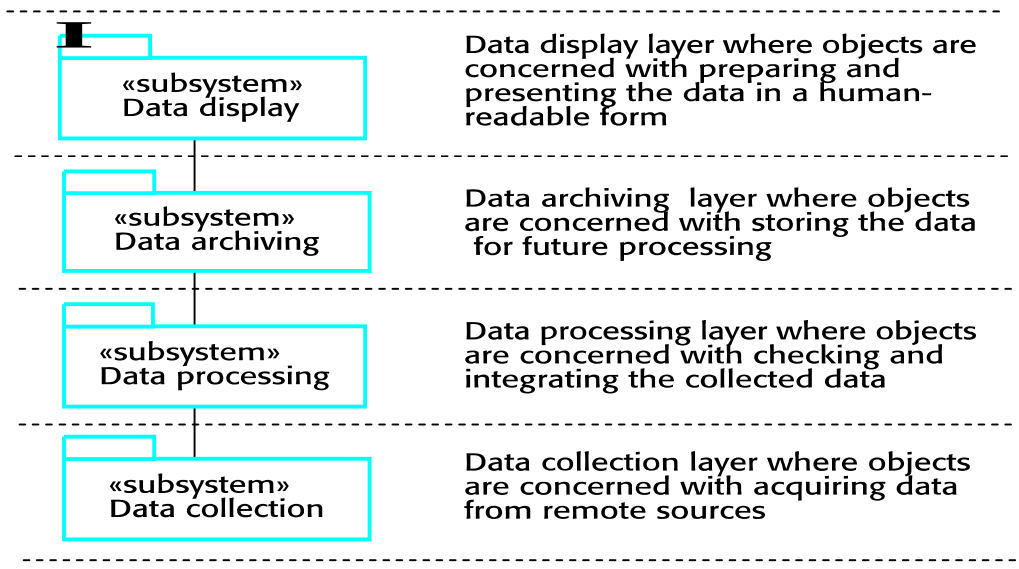
- Threads in Java are a simple construct for implementing concurrent objects
- Threads must include a method called run() and this is started up by the Java run-time system

- Active objects typically include an infinite loop so that they are always carrying out the computation

An object-oriented design process

- Define the context and modes of use of the system
- Design the system architecture
- Identify the principal system objects
- Develop design models
- Specify object interfaces

Layered architecture

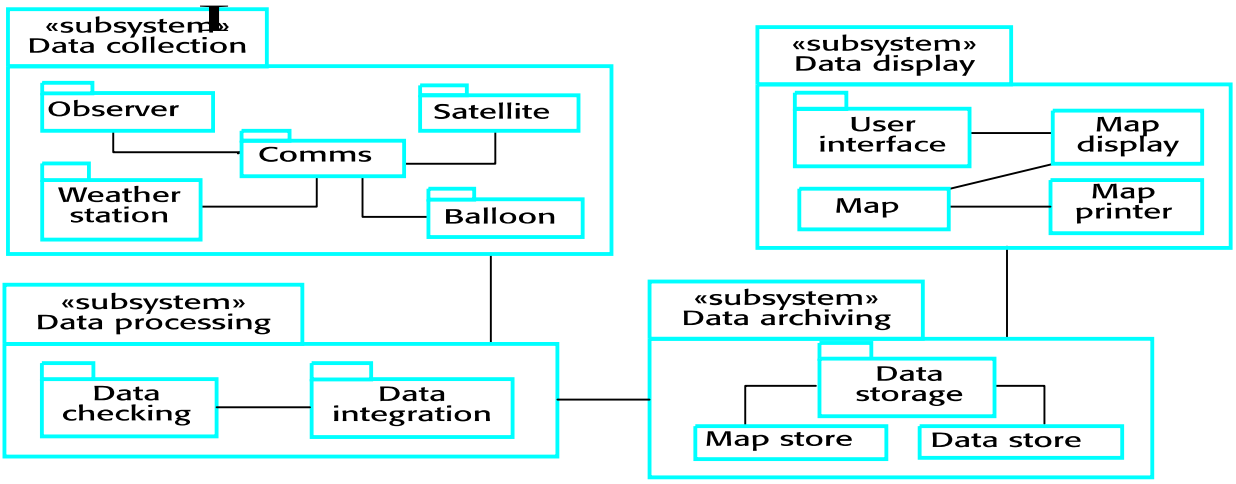


System context and models of use

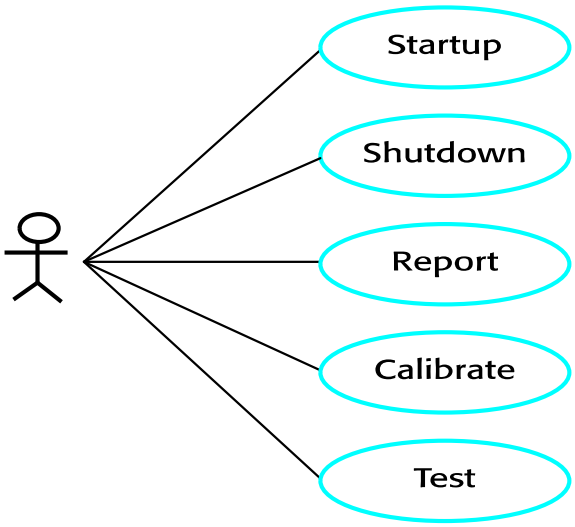
Develop an understanding of the relationships between the software being designed and its external environment

- System context: A static model that describes other systems in the environment. Use a subsystem model to show other systems. Following slide shows the systems around the weather station system.
- Model of system use: A dynamic model that describes how the system interacts with its environment. Use use-cases to show interactions

Subsystems in the weather mapping system



Use-cases for the weather station



Use-case description

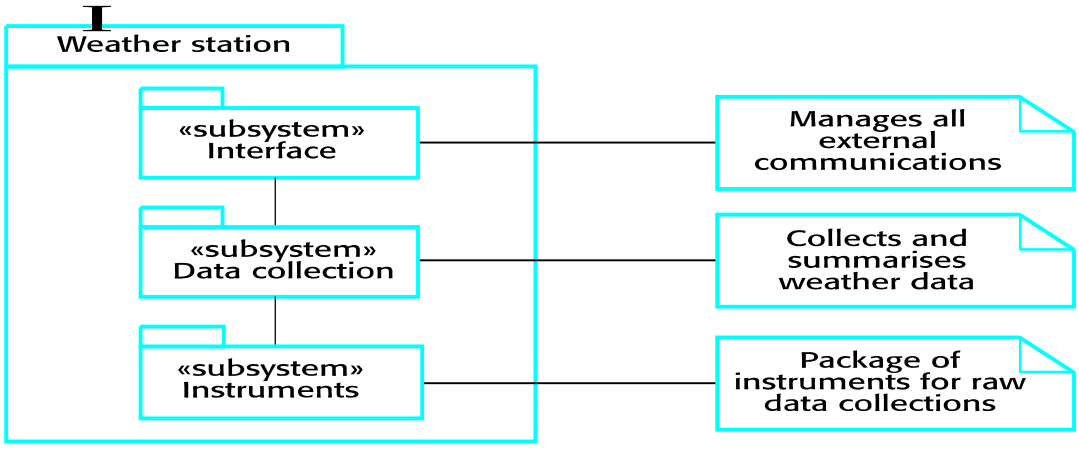
System	Weather station
Use-case	Report
Actors	Weather data collection system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather data **collection system. The** data sent are the maximum minimum and average ground and air temperatures, the maximum, minimum and average air pressures, the maximum, minimum and average wind

	speeds, the total rainfall and the wind direction as sampled at 5 minute s intervals.
Stimulus	The weather data collection system establishes a modem link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather data collection system
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to the other and may be modified in future

Architectural design

- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture
- Layered architecture is appropriate for the weather station
- Interface layer for handling communications
- Data collection layer for managing instruments
- Instruments layer for collecting data
- There should be no more than 7 entities in an architectural model

Weather station architecture



Object identification

- Identifying objects (or object classes) is the most difficult part of object oriented design
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers
- Object identification is an iterative process. You are unlikely to get it right first time

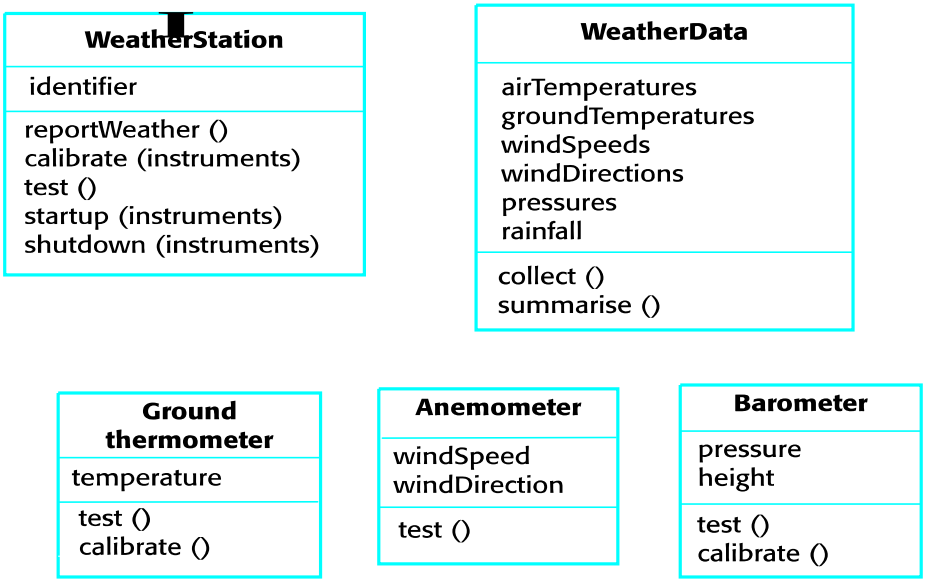
Approaches to identification

- Use a grammatical approach based on a natural language description of the system (used in Hood method)
- Base the identification on tangible things in the application domain
- Use a behavioral approach and identify objects based on what participates in what behavior
- Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified

Weather station object classes

- Ground thermometer, Anemometer, Barometer: Application domain objects that are 'hardware' objects related to the instruments in the system.
- Weather station: The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model
- Weather data: Encapsulates the summarized data from the instruments

Weather station object classes



Further objects and object refinement

- Use domain knowledge to identify more objects and operations
- Weather stations should have a unique identifier
- Weather stations are remotely situated so instrument failures have to be reported automatically. Therefore attributes and operations for self-checking are required
- Active or passive objects: In this case, objects are passive and collect data on request rather than autonomously. This introduces flexibility at the expense of controller processing time

Design models

- Design models show the objects and object classes and relationships between these entities
- Static models describe the static structure of the system in terms of object classes and relationships
- Dynamic models describe the dynamic interactions between objects.

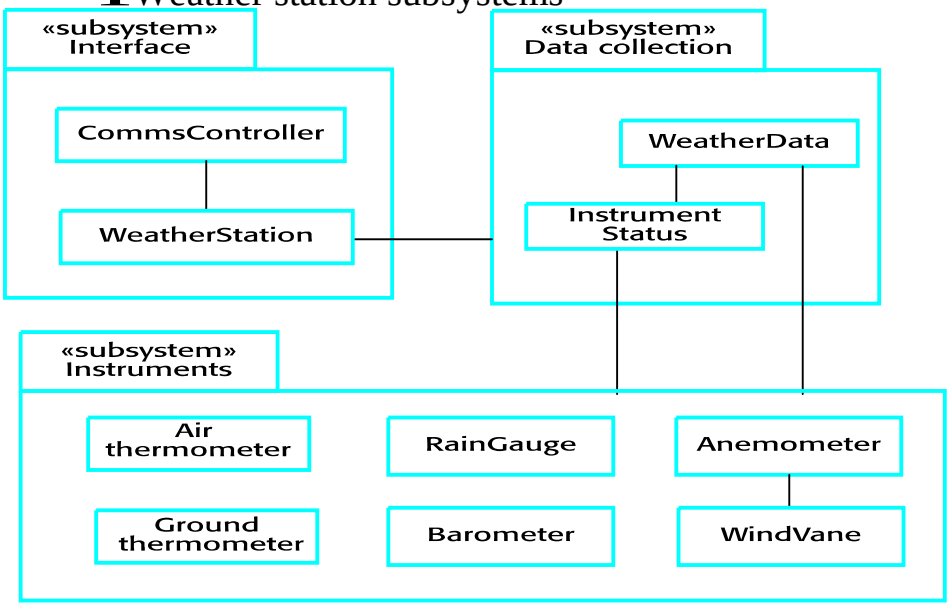
Examples of design models

- Sub-system models that show logical groupings of objects into coherent subsystems
- Sequence models that show the sequence of object interactions
- State machine models that show how individual objects change their state in response to events
- Other models include use-case models, aggregation models, generalization models, etc.

Subsystem models

- Shows how the design is organized into logically related groups of objects
- In the UML, these are shown using packages – an encapsulation construct. This is a logical model. The actual organization of objects in the system may be different.

Weather station subsystems

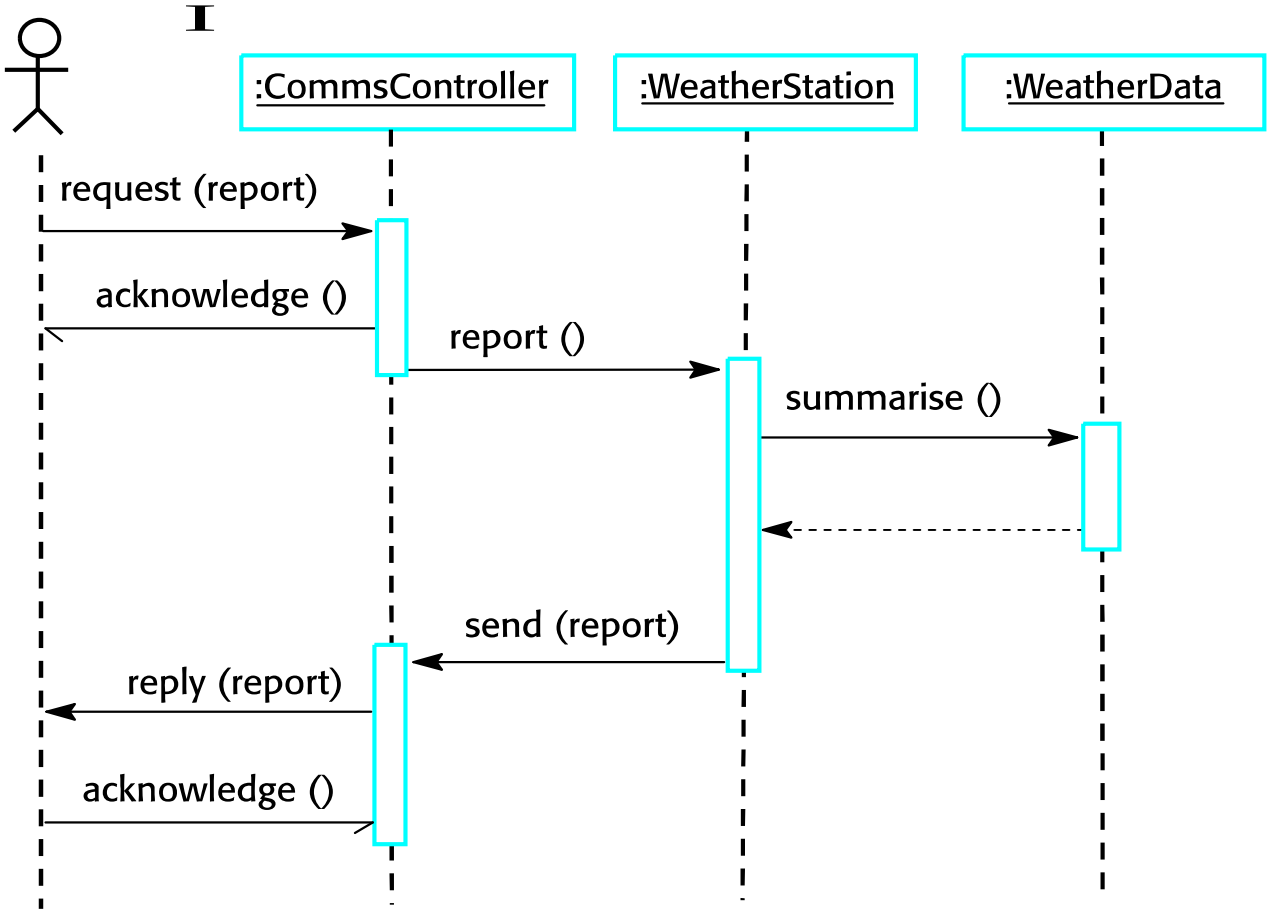


Sequence models

- Sequence models show the sequence of object interactions that take place
- Objects are arranged horizontally across the top
- Time is represented vertically so models are read top to bottom
- Interactions are represented by labeled arrows, Different styles of arrow represent different types of interaction

- A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system

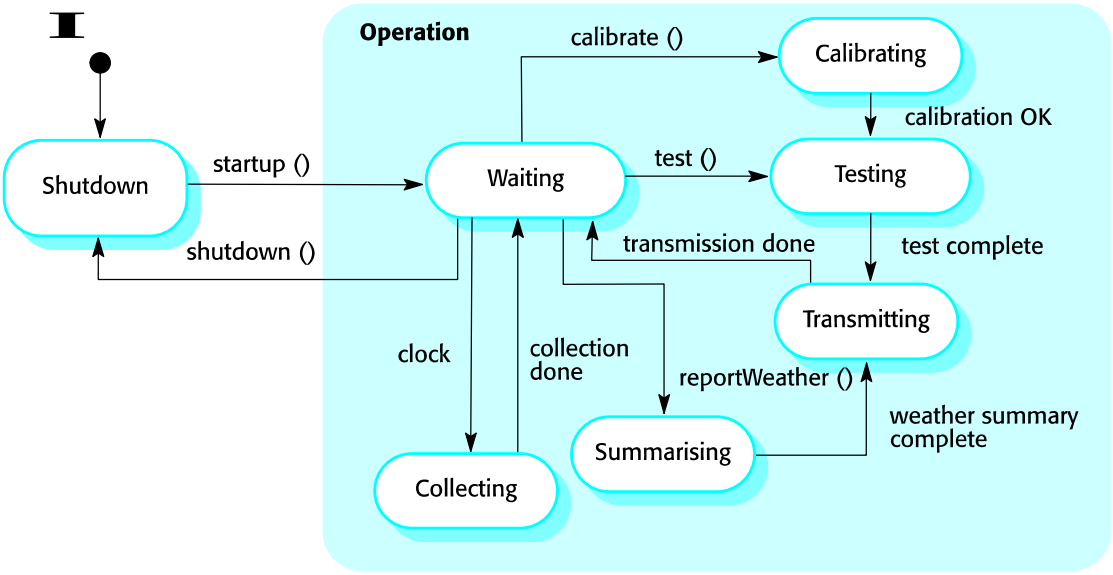
Data collection sequence



Statecharts

- Show how objects respond to different service requests and the state transitions triggered by these requests
- If object state is Shutdown then it responds to a Startup() message In the waiting state the objec t is waiting for further messages
- If reportWeather () then system moves to summarizing state
- If calibrate () the system moves to a calibrating state
- A collecting state is entered when a clock signal is received

Weather station state diagram



Object interface specification

- Object interfaces have to be specified so that the objects and other components can be designed in parallel
- Designers should avoid designing the interface representation but should hide this in the object itself
- Objects may have several interfaces which are viewpoints on the methods provided
- The UML uses class diagrams for interface specification but Java may also be used

Weather station interface

```
interface WeatherStation {
public void WeatherStation () ;
public void startup () ;
public void startup (Instrument i) ;
public void shutdown () ;
public void shutdown (Instrument i) ;
public void reportWeather ( ) ;
public void test () ;
public void test ( Instrument i ) ;
public void calibrate ( Instrument i) ;
public int getID () ;
} //WeatherStation
```

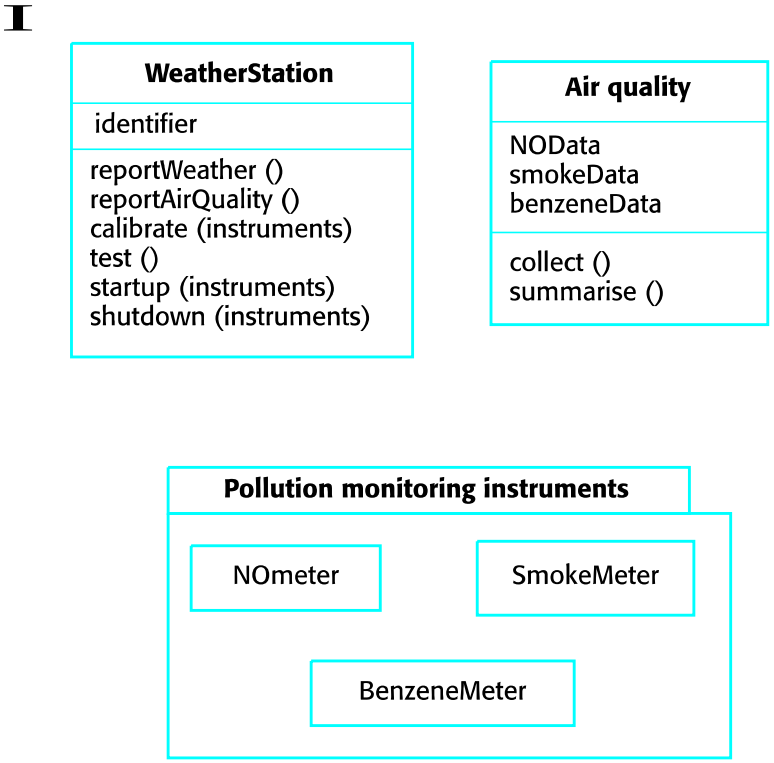
Design evolution

- Hiding information inside objects means that changes made to an object do not affect other objects in an unpredictable way
- Assume pollution monitoring facilities are to be added to weather stations. These sample the air and compute the amount of different pollutants in the atmosphere
- Pollution readings are transmitted with weather data

Changes required

- Add an object class called 'Air quality' as part of WeatherStation
- Add an operation reportAirQuality to WeatherStation. Modify the control software to collect pollution readings
- Add objects representing pollution monitoring instruments

Pollution monitoring



- OOD is an approach to design so that design components have their own private state and operations
- Objects should have constructor and inspection operations. They provide services to other objects

- Objects may be implemented sequentially or concurrently
- The Unified Modeling Language provides different notations for defining different object models
- A range of different models may be produced during an object-oriented design process. These include static and dynamic system models
- Object interfaces should be defined precisely using e.g. a programming language like Java.
- Object-oriented design simplifies system evolution

UNIT –6
Development

Rapid Software Development

Because of rapidly changing business environments, businesses have to respond to new opportunities and competition. This requires software and rapid development and delivery is not often the most critical requirement for software systems. Businesses may be willing to accept lower quality software if rapid delivery of essential functionality is possible.

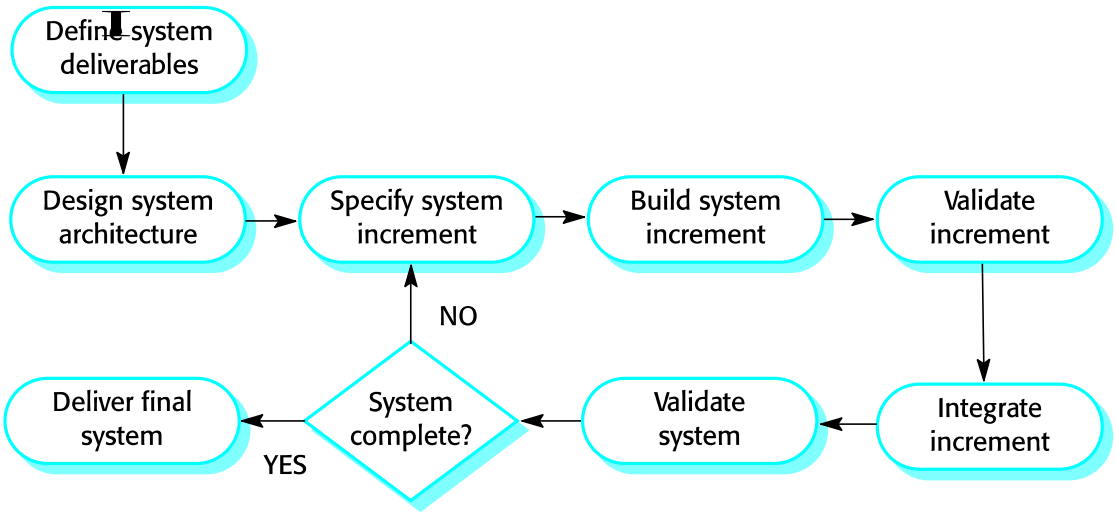
Requirements

Because of the changing environment, it is often impossible to arrive at a stable, consistent set of system requirements. Therefore a waterfall model of development is impractical and an approach to development based on iterative specification and delivery is the only way to deliver software quickly.

Characteristics of RAD processes

- The processes of specification, design and implementation are concurrent. There is no detailed specification and design documentation is minimized.
- The system is developed in a series of increments. End users evaluate each increment and make proposals for later increments.
- System user interfaces are usually developed using an interactive development system.

An iterative development process



Advantages of incremental development

- *Accelerated delivery of customer services*. Each increment delivers the highest priority functionality to the customer.

- *User engagement with the system*. Users have to be involved in the development which means the system is more likely to meet their requirements and the users are more committed to the system.

Problems with incremental development

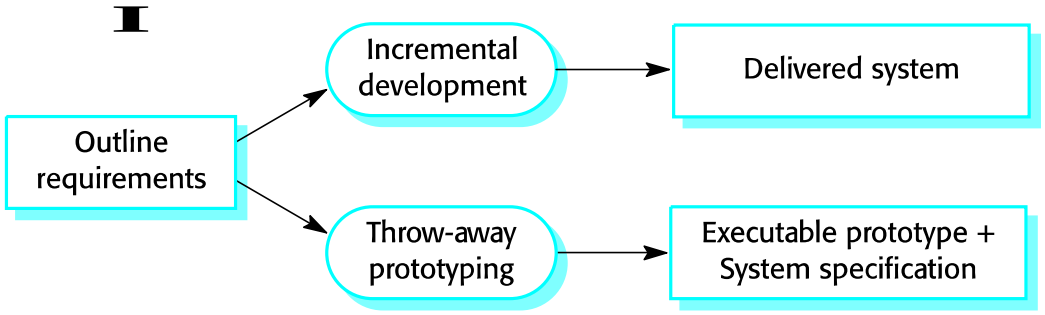
- Management problems: Progress can be hard to judge and problems hard to find because there is no documentation to demonstrate what has been done.
- Contractual problems: The normal contract may include a specification; without a specification, different forms of contract have to be used.
- Validation problems: Without a specification, what is the system being tested against?
- Maintenance problems: Continual change tends to corrupt software structure making it more expensive to change and evolve to meet new requirements.

Prototyping

For some large systems, incremental iterative development and delivery may be impractical; this is especially true when multiple teams are working on different sites.

Prototyping, where an experimental system is developed as a basis for formulating the requirements may be used. This system is thrown away when the system specification has been agreed.

Incremental development and prototyping



Conflicting objectives

The objective of incremental development is to deliver a working system to end-users. The development starts with those requirements which are best understood.

The objective of throw-away prototyping is to validate or derive the system requirements. The prototyping process starts with those requirements which are poorly understood.

Agile methods

- Dissatisfaction with the overheads involved in design methods led to the creation of agile methods. These methods:
- Focus on the code rather than the design;
- Are based on an iterative approach to software development;

- Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- Agile methods are probably best suited to small/medium-sized business systems or PC products.

Principles of agile methods

Principle	Description
Customer involvement	The customer should be closely involved throughout the development process. Their role is provide and prioritise new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognised and exploited. The team should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and design the system so that it can accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process used. Wherever possible, actively work to eliminate complexity from the system.

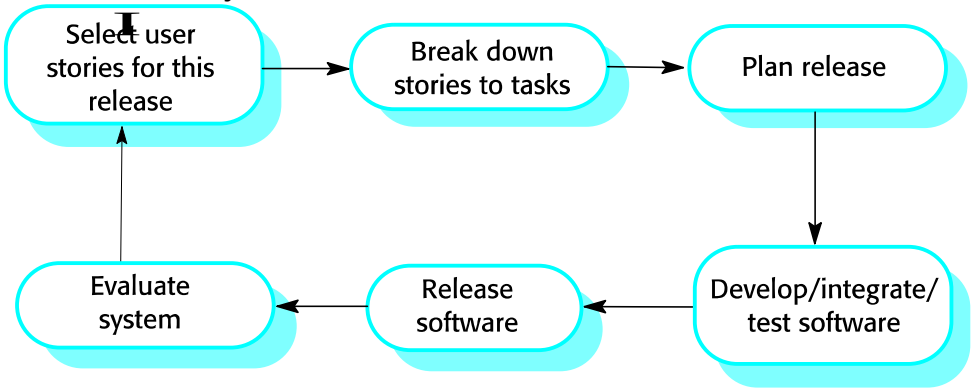
Problems with agile methods

- It can be difficult to keep the interest of customers who are involved in the process.
- Team members may be unsuited to the intense involvement that characterizes agile methods.
- Prioritizing changes can be difficult where there are multiple stakeholders.
- Maintaining simplicity requires extra work.
- Contracts may be a problem as with other approaches to iterative development.

Extreme programming

- Perhaps the best-known and most widely used agile method.
- Extreme Programming (XP) takes an ‘extreme’ approach to iterative development.
- New versions may be built several times per day;
- Increments are delivered to customers every 2 weeks;
- All tests must be run for every build and the build is only accepted if tests run successfully.

The XP release cycle



Extreme programming practices 1

Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development ÕTasksÕ
Small Releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple Design	Enough design is carried out to meet the current requirements and no more.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

Extreme programming practices 2

Pair Programming	Developers work in pairs, checking each otherÕs work and providing the support to always do a good job.
Collective Ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers own all the code. Anyone can change anything.
Continuous Integration	As soon as work on a task is complete it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of over-time are not considered acceptable as the net effect is often to reduce code qua lity and medium term productivity
On-site Customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

XP and agile principles

- Incremental development is supported through small, frequent system releases.
- Customer involvement means full-time customer engagement with the team.
- People not process through pair programming, collective ownership and a process that avoids long working hours.
- Change supported through regular system releases.
- Maintaining simplicity through constant refactoring of code.

Requirements scenarios

- In XP, user requirements are expressed as scenarios or user stories.
- These are written on cards and the development team breaks them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

Story card for document downloading

Downloading and printing an article
First, you select the article that you want from a displayed list. You then have to tell the system how you will pay for it - this can either be through a subscription, through a company account or by credit card. After this, you get a copyright form from the system to fill in and, when you have submitted this, the article you want is downloaded onto your computer. You then choose a printer and a copy of the article is printed. You tell the system if printing has been successful. If the article is a print-only article, you can't keep the PDF version so it is automatically deleted from your computer.

XP and change

- Conventional wisdom in software engineering is to design for change. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
- XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated.
- Rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented

Testing in XP

- Test-first development. Incremental test development from scenarios.
- User involvement in test development and validation.
- Automated test harnesses are used to run all component tests each time that a new release is built.

Test case description

Test 4: Test credit card validity
Input: A string representing the credit card number and two integers representing the month and year when the card expires **Tests:** Check that all bytes in the string are digits Check that the month lies between 1 and 12 and the year is greater than or equal to the current year. Using the first 4 digits of the credit card number, check that the card issuer is valid by looking up the card issuer table. Check credit card validity by submitting the card number and expiry date information to the card issuer **Output:** OK or error message indicating that the card is invalid

Test-first development

- Writing tests before code clarifies the requirements to be implemented.
- Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
- All previous and new tests are automatically run when new functionality is added. Thus checking that the new functionality has not introduced errors.

Pair programming

In XP, programmers work in pairs, sitting together to develop code. This helps develop common ownership of code and spreads knowledge across the team. It serves as an informal review process as each line of code is looked at by more than 1 person. It encourages refactoring as the whole team can benefit from this. Measurements suggest that development productivity with pair programming is similar to that of two people working independently.

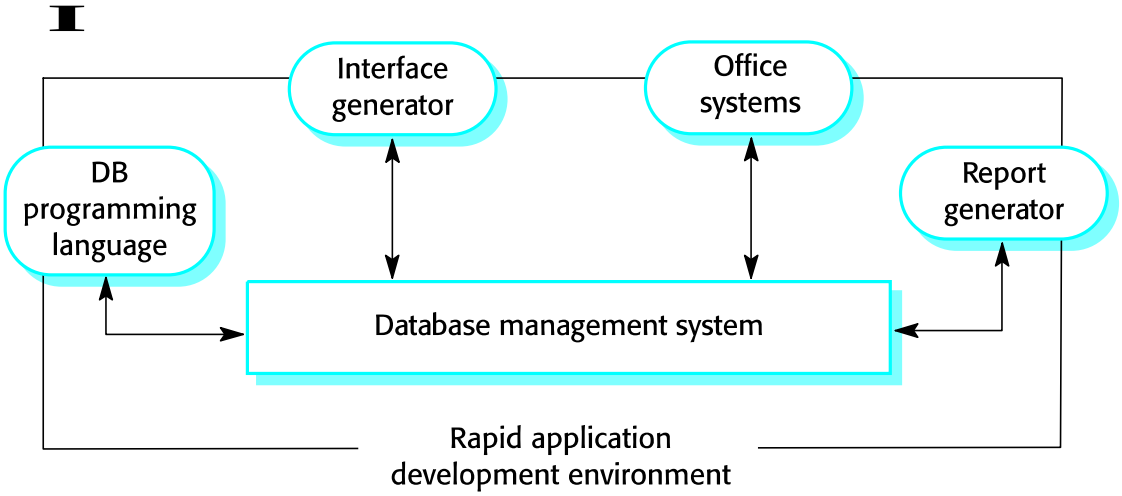
Rapid application development

Agile methods have received a lot of attention but other approaches to rapid application development have been used for many years.
These are designed to develop data-intensive business applications and rely on programming and presenting information from a database.

RAD environment tools

- Database programming language
- Interface generator
- Links to office applications
- Report generators

A RAD environment



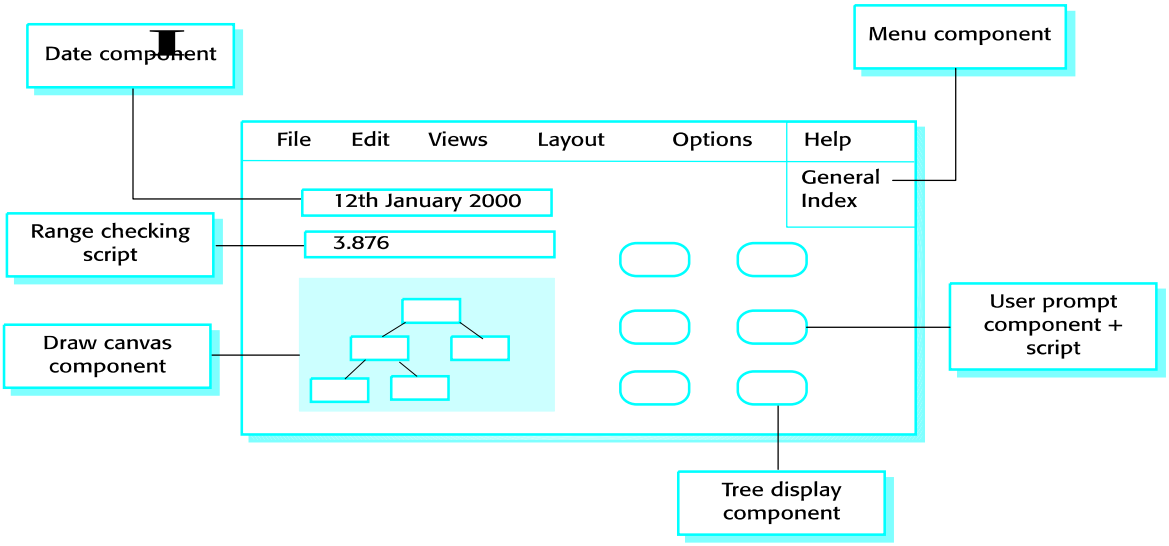
Interface generation

- Many applications are based around complex forms and developing these forms manually is a time-consuming activity.
- RAD environments include support for screen generation including:
- Interactive form definition using drag and drop techniques;
- Form linking where the sequence of forms to be presented is specified;
- Form verification where allowed ranges in form fields are defined.

Visual programming

- Scripting languages such as Visual Basic support visual programming where the prototype is developed by creating a user interface from standard items and associating components with these items
- A large library of components exists to support this type of development
- These may be tailored to suit the specific application requirements.

Visual programming with reuse



Problems with visual development

Difficult to coordinate team-based development. No explicit system architecture. Complex dependencies between parts of the program can cause maintainability problems.

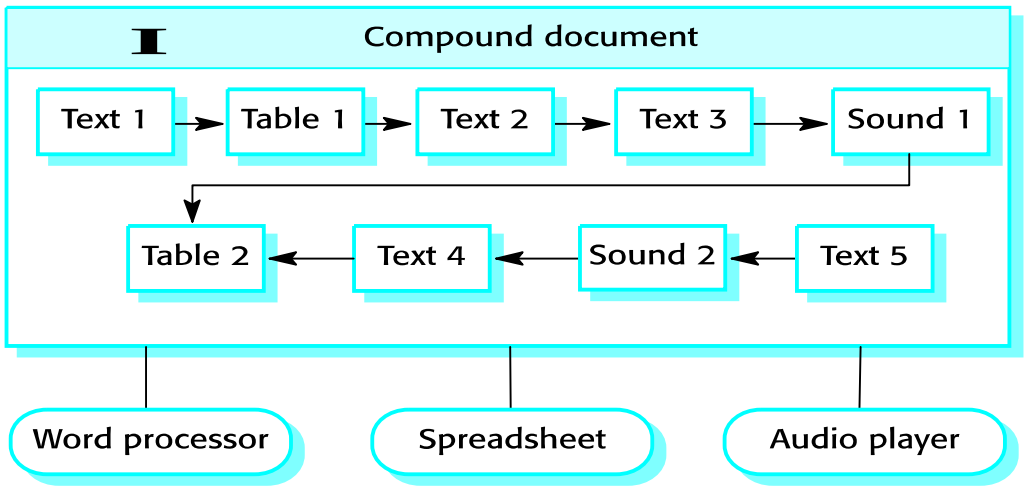
COTS reuse

- An effective approach to rapid development is to configure and link existing off the shelf systems. For example, a requirements management system could be built by using:
- A database to store requirements;
- A word processor to capture requirements and format reports;
- A spreadsheet for traceability management;

Compound documents

For some applications, a prototype can be created by developing a compound document. This is a document with active elements (such as a spreadsheet) that allow user computations. Each active element has an associated application which is invoked when that element is selected. The document itself is the integrator for the different applications.

Application linking



Software prototyping

- A prototype is an initial version of a system used to demonstrate concepts and try out design options.

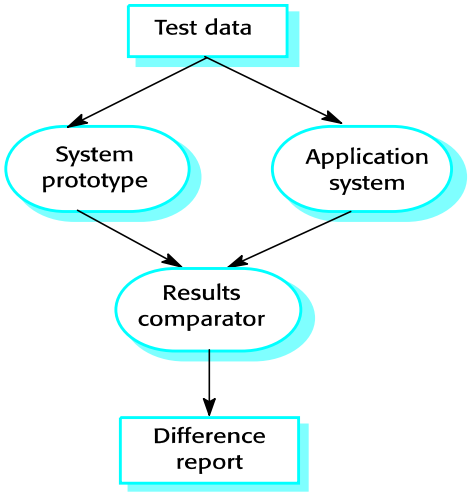
A prototype can be used in:

- The requirements engineering process to help with requirements elicitation and validation;
- In design processes to explore options and develop a UI design;
- In the testing process to run back-to-back tests.

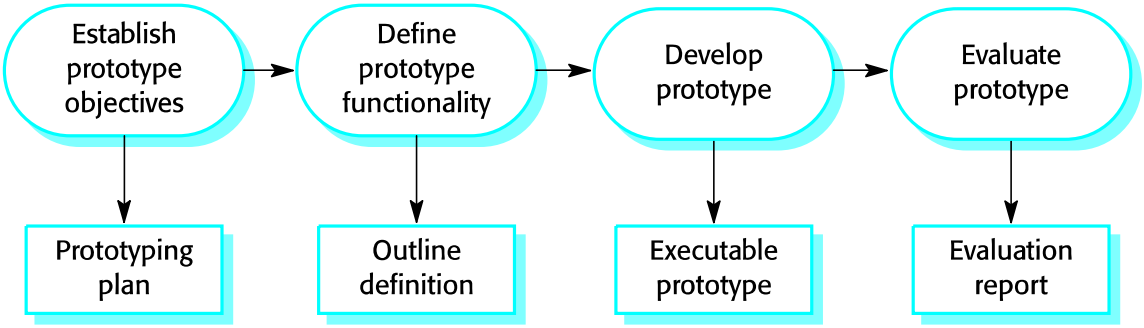
Benefits of prototyping

- Improved system usability.
- A closer match to users' real needs.
- Improved design quality.
- Improved maintainability.
- Reduced development effort.

Back to back testing



The prototyping process



Throw-away prototypes

- Prototypes should be discarded after development as they are not a good basis for a production system:
- It may be impossible to tune the system to meet non-functional requirements;
- Prototypes are normally undocumented;
- The prototype structure is usually degraded through rapid change;
- The prototype probably will not meet normal organizational quality standards.

Software evolution

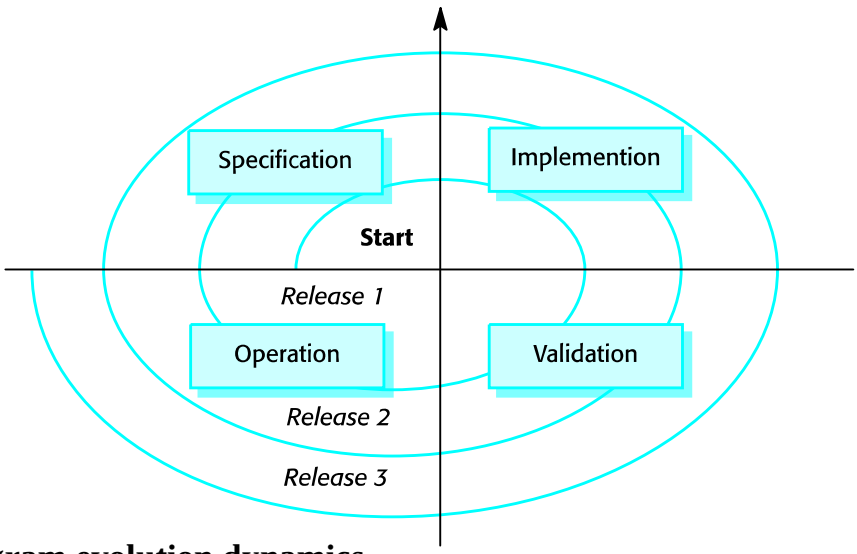
Software change

- Software change is inevitable
- New requirements emerge when the software is used;
- The business environment changes;
- Errors must be repaired;
- New computers and equipment is added to the system;
- The performance or reliability of the system may have to be improved.
- A key problem for organisations is implementing and managing change to their existing software systems

Importance of evolution

Organizations have huge investments in their software systems - they are critical business assets. To maintain the value of these assets to the business, they must be changed and updated. The majority of the software budget in large companies is devoted to evolving existing software rather than developing new software.

Spiral model of evolution



Program evolution dynamics

Program evolution dynamics is the study of the processes of system change. After major empirical studies, Lehman and Belady proposed that there were a number of 'laws' which applied to all systems as they evolved. There are sensible observations rather than laws. They are applicable to large systems developed by large organisations. Perhaps less applicable in other cases.

Lehman's laws (important)

Law	Description
Continuing change	A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.
Organisational stability	Over a programÕs lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.
Feedback system	Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

Applicability of Lehman's laws

Lehman's laws seem to be generally applicable to large, tailored systems developed by large organisations.

- Confirmed in more recent work by Lehman on the FEAST project (see further reading on book website).
- It is not clear how they should be modified for
- Shrink-wrapped software products;
- Systems that incorporate a significant number of COTS components;
- Small organisations;
- Medium sized systems.

Software maintenance

Modifying a program after it has been put into use. Maintenance does not normally involve major changes to the system's architecture. Changes are implemented by modifying existing components and adding new components to the system

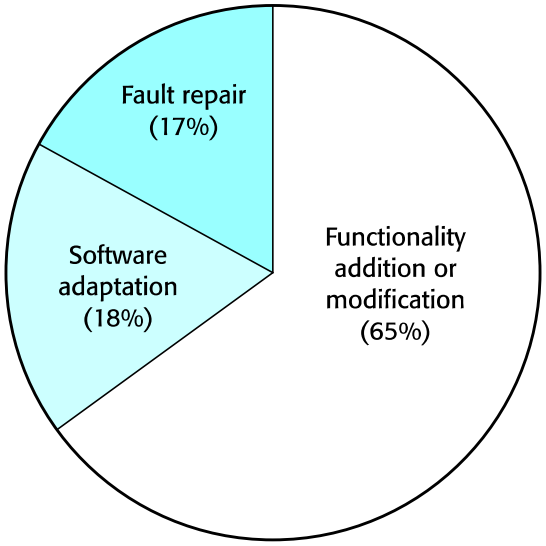
Maintenance is inevitable

- The system requirements are likely to change while the system is being developed because the environment is changing. Therefore a delivered system won't meet its requirements!
- Systems are tightly coupled with their environment. When a system is installed in an environment it changes that environment and therefore changes the system requirements.
- Systems MUST be maintained therefore if they are to remain useful in an environment.

Types of maintenance

- Maintenance to repair software faults
  - Changing a system to correct deficiencies in the way meets its requirements.
- Maintenance to adapt software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- Maintenance to add to or modify the system's functionality
  - Modifying the system to satisfy new requirements.

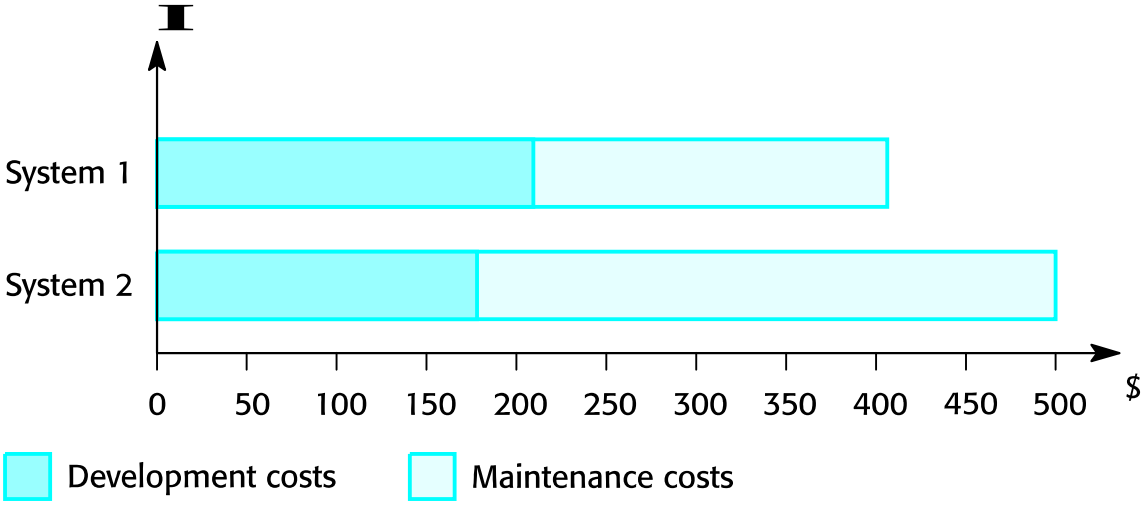
Distribution of maintenance effort



Maintenance costs

- Usually greater than development costs (2* to 100* depending on the application).
- Affected by both technical and non-technical factors.
- Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.
- Ageing software can have high support costs (e.g. old languages, compilers etc.).

Development/maintenance costs



Maintenance cost factors

- Team stability
- Maintenance costs are reduced if the same staffs are involved with them for some time.
- Contractual responsibility
- The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.
- Staff skills
- Maintenance staffs are often inexperienced and have limited domain knowledge.
- Program age and structure
- As programs age, their structure is degraded and they become harder to understand and change.

Maintenance prediction

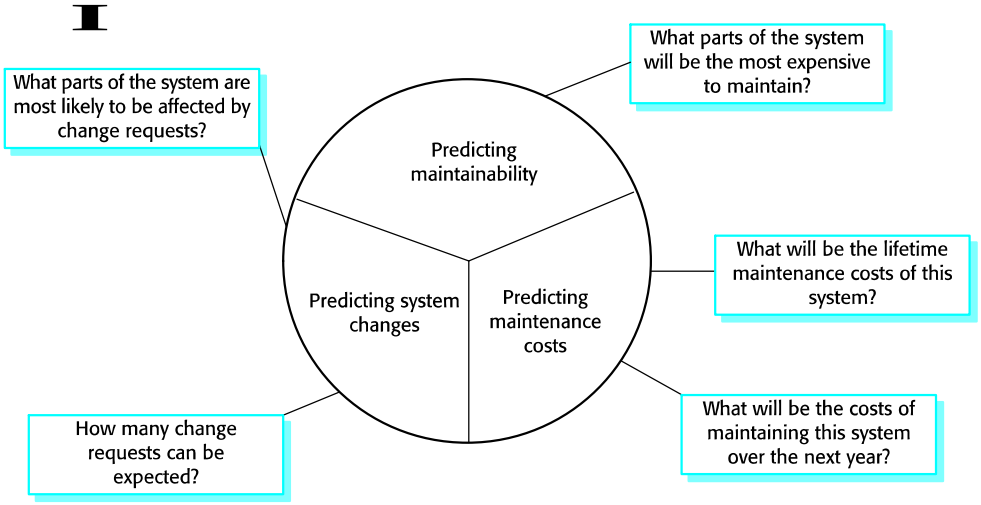
Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs

Change acceptance depends on the maintainability of the components affected by the change;

Implementing changes degrades the system and reduces its maintainability;

Maintenance costs depend on the number of changes and costs of change depend on maintainability.

Maintenance prediction



Change prediction

- Predicting the number of changes requires and understanding of the relationships between a system and its environment.
- Tightly coupled systems require changes whenever the environment is changed.
- Factors influencing this relationship are
- Number and complexity of system interfaces;
- Number of inherently volatile system requirements;
- The business processes where the system is used.

Complexity metrics

- Predictions of maintainability can be made by assessing the complexity of system components.
- Studies have shown that most maintenance effort is spent on a relatively small number of system components.
- Complexity depends on
- Complexity of control structures;
- Complexity of data structures;
- Object, method (procedure) and module size.

Process metrics

Process measurements may be used to assess maintainability

- Number of requests for corrective maintenance;
- Average time required for impact analysis;
- Average time taken to implement a change request;
- Number of outstanding change requests.

If any or all of these is increasing, this may indicate a decline in maintainability.

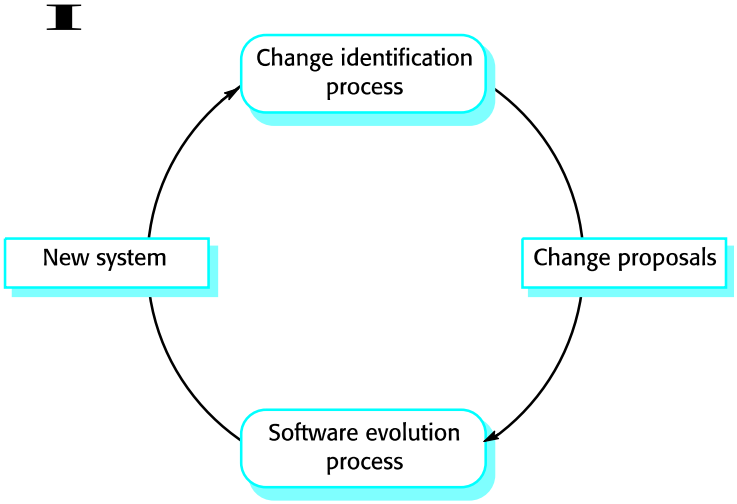
Evolution processes

Evolution processes depend on

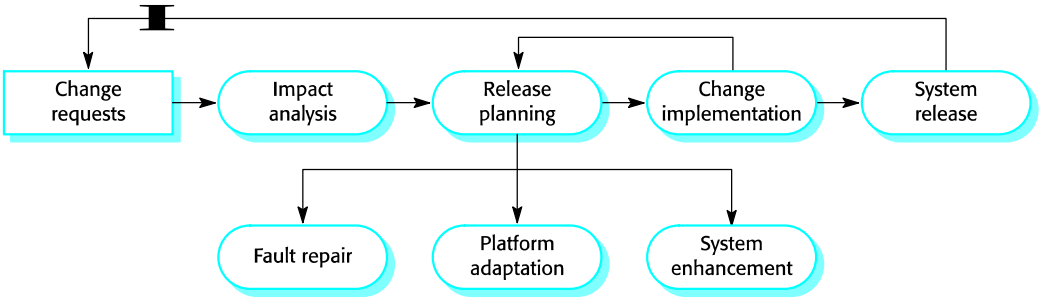
·The type of software being maintained;

·The development processes used;
·The skills and experience of the people involved.
Proposals for change are the driver for system evolution. Change identification and evolution continue throughout the system lifetime.

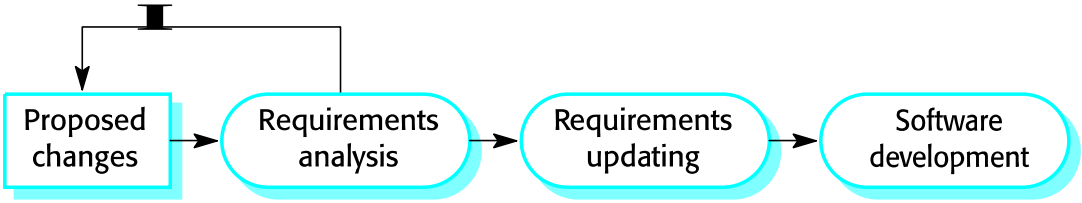
Change identification and evolution



The system evolution process



Change implementation

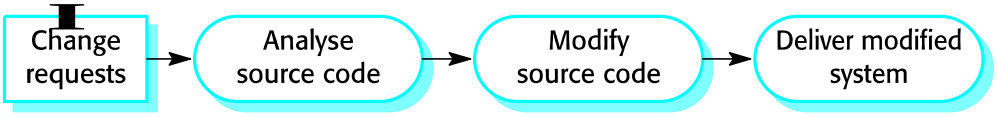


Urgent change requests

Urgent changes may have to be implemented without going through all stages of the software engineering process

·If a serious system fault has to be repaired;
·If changes to the system's environment (e.g. an OS upgrade) have unexpected effects;
·If there are business changes that require a very rapid response (e.g. the release of a competing product).

Emergency repair



System re-engineering

- Re-structuring or re-writing part or all of a legacy system without changing its functionality.
- Applicable where some but not all sub-systems of a larger system require frequent maintenance.
- Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.

Advantages of reengineering

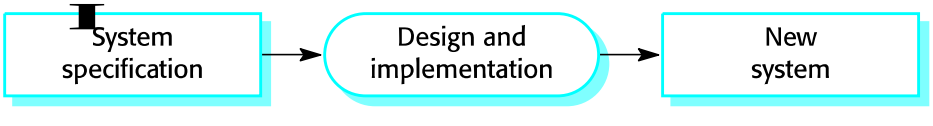
Reduced risk

·There is a high risk in new software development. There may be development problems, staffing problems and specification problems.

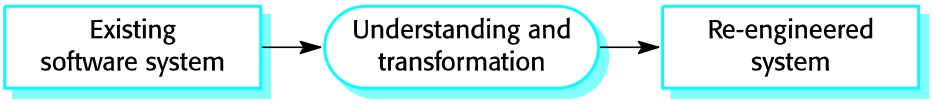
Reduced cost

·The cost of re-engineering is often significantly less than the costs of developing new software.

Forward and re-engineering

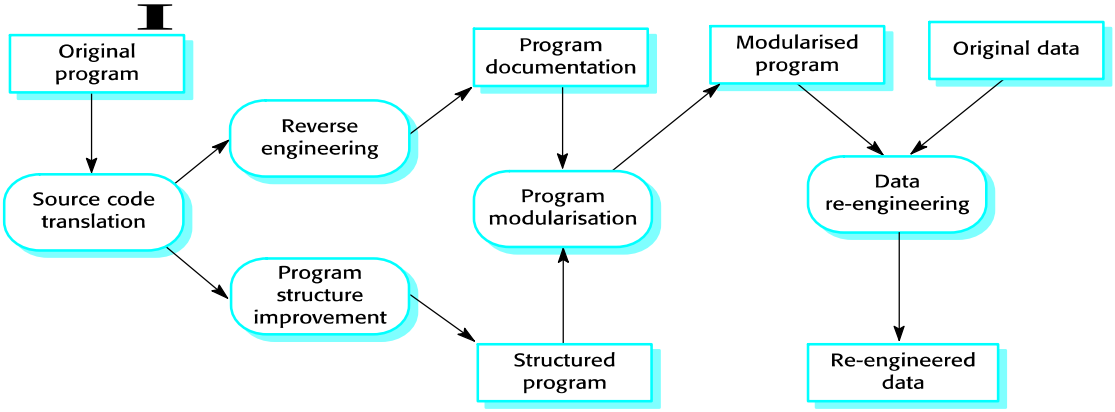


Forward engineering



Software re-engineering

The re-engineering process



Reengineering process activities

Source code translation

- Convert code to a new language.

Reverse engineering

- Analyze the program to understand it;

Program structure improvement

- Restructure automatically for understandability;

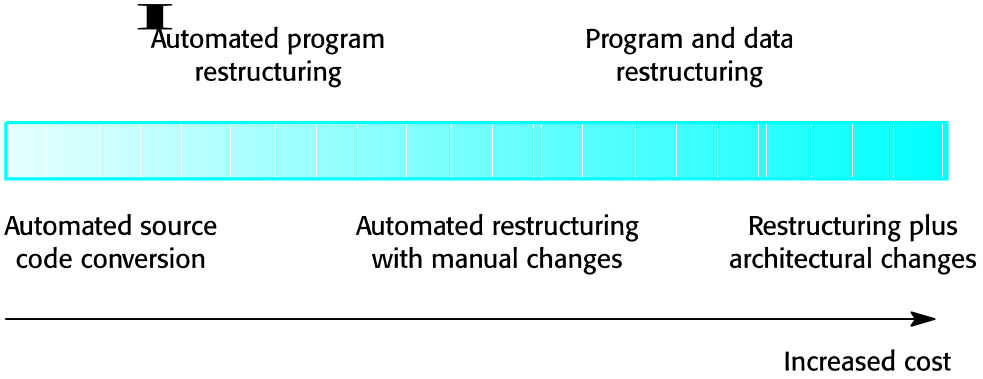
Program modularization

- Reorganize the program structure;

Data reengineering

- Clean-up and restructure system data.

Reengineering cost factors



Re-engineering approaches

- The quality of the software to be reengineered.
- The tool support available for reengineering.
- The extent of the data conversion which is required.
- The availability of expert staff for reengineering.
- This can be a problem with old systems based on technology that is no longer widely used.

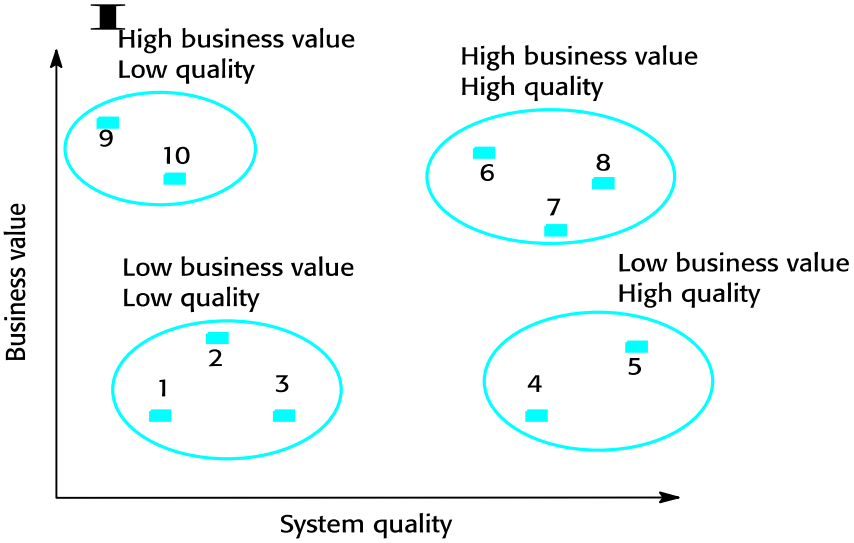
Legacy system evolution

Organisations that rely on legacy systems must choose a strategy for evolving these systems

- Scrap the system completely and modify business processes so that it is no longer required;
- Continue maintaining the system;
- Transform the system by re-engineering to improve its maintainability;
- Replace the system with a new system.

The strategy chosen should depend on the system quality and its business value.

System quality and business value



Legacy system categories

Low quality, low business value

- These systems should be scrapped.

Low-quality, high-business value

- These make an important business contribution but are expensive to maintain. Should be re-engineered or replaced if a suitable system is available.

High-quality, low-business value

- Replace with COTS, scrap completely or maintain.

High-quality, high business value

- Continue in operation using normal system maintenance.

Business value assessment

Assessment should take different viewpoints into account

- System end-users;
- Business customers;
- Line managers;
- IT managers;
- Senior managers.

Interview different stakeholders and collate results.

System quality assessment

Business process assessment
- How well does the business process support the current goals of the business?

Environment assessment
- How effective is the system's environment and how expensive is it to maintain?

Application assessment
- What is the quality of the application software system?

Business process assessment

Use a viewpoint-oriented approach and seek answers from system stakeholders

- Is there a defined process model and is it followed?
- Do different parts of the organisation use different processes for the same function?
- How has the process been adapted?
- What are the relationships with other business processes and are these necessary?
- Is the process effectively supported by the legacy application software?

Example - a travel ordering system may have a low business value because of the widespread use of web-based ordering.

Environment assessment 1

Factor	**Questions**
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent and up-to-date?
Data	Is there an explicit data model for the system? To what extent is data duplicated in different files? Is the data used by the system up-to-date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?

System measurement

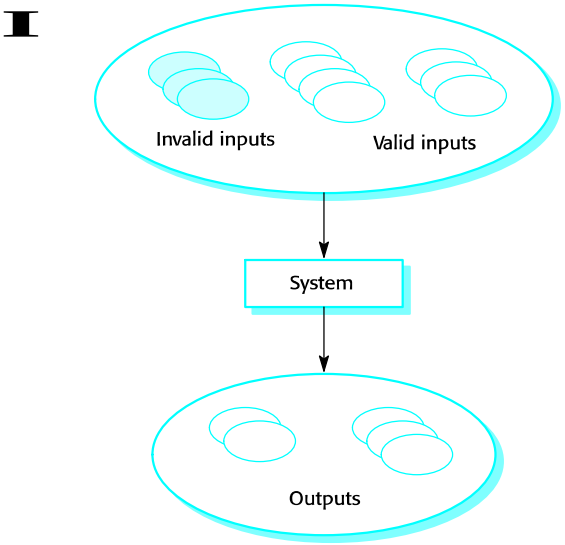
You may collect quantitative data to make an assessment of the quality of the application system

·The number of system change requests;
·The number of different user interfaces used by the system;
·The volume of data used by the system.

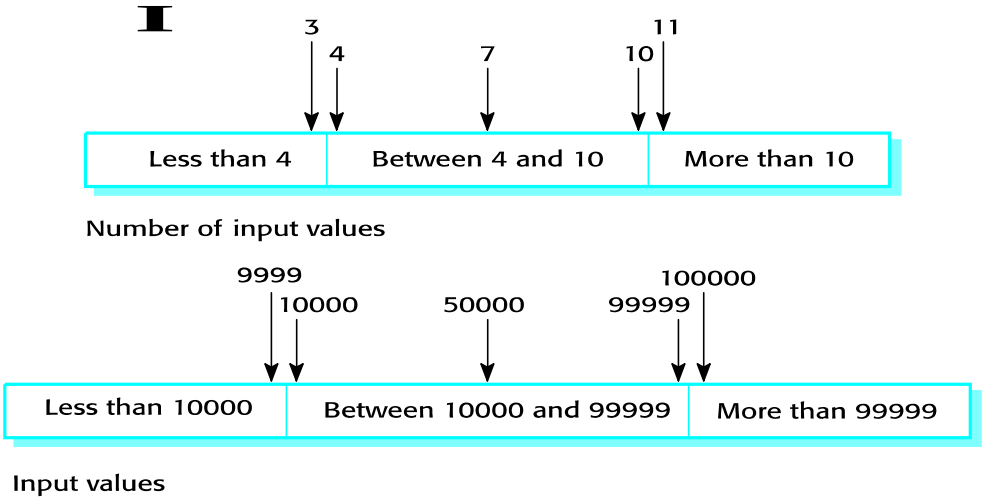
Black-box testing

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.

Equivalence partitioning



Equivalence partitions



UNIT 7
VERIFICATION AND VALIDATION

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

- Is 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 useful and useable in an operational situation

V& V goals

Verification and validation should establish confidence that the software is fit for purpose. This does NOT mean completely free of defects. Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed.

V & V confidence

Depends on system's purpose, user expectations and marketing environment

Software function

·The level of confidence depends on how critical the software is to an organisation.

User expectations

·Users may have low expectations of certain kinds of software.

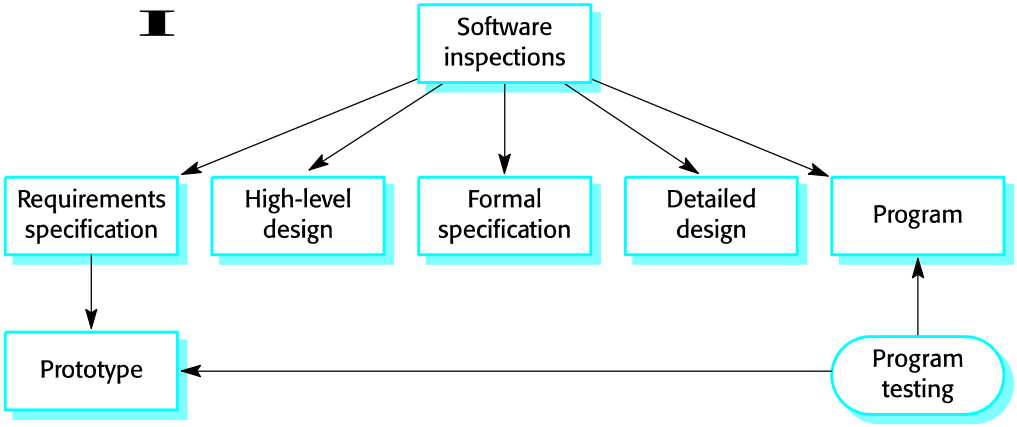
Marketing environment

·Getting a product to market early may be more important than finding defects in the program.

Static and dynamic verification

- Software inspections. Concerned with analysis of the static system representation to discover problems *(*static verification)
- May be supplement by tool-based document and code analysis
- Software testing. Concerned with exercising and observing product behaviour (dynamic verification)
- The system is executed with test data and its operational behaviour is observed

Static and dynamic V&V



Program testing

Can reveal the presence of errors NOT their absence. The only validation technique for non-functional requirements as the software has to be executed to see how it behaves. Should be used in conjunction with static verification to provide full V&V coverage.

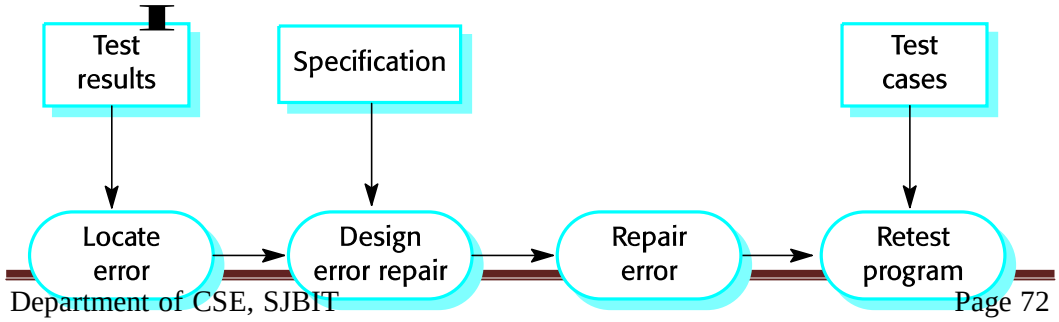
Types of testing

- Defect testing: Tests designed to discover system defects. A successful defect test is one which reveals the presence of defects in a system. Covered in Chapter 23
- Validation testing: Intended to show that the software meets its requirements. A successful test is one that shows that a requirement has been properly implemented.

Testing and debugging

- Defect testing and debugging are distinct processes. Verification and validation is concerned with establishing the existence of defects in a program. Debugging is concerned with locating and repairing these errors. Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error.

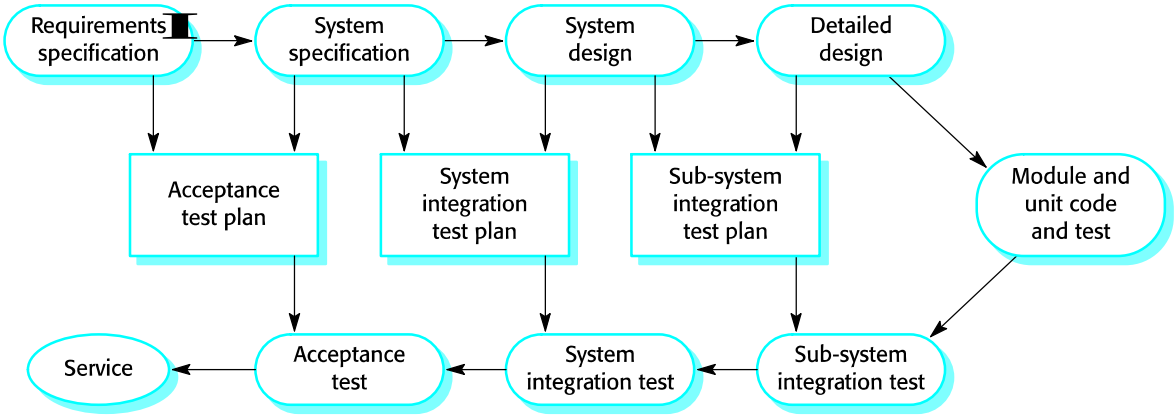
The debugging process



V & V planning

Careful planning is required to get the most out of testing and inspection processes. Planning should start early in the development process. The plan should identify the balance between static verification and testing. Test planning is about defining standards for the testing process rather than describing product tests.

The V-model of development



The structure of a software test plan

- The testing process.
- Requirements traceability.
- Tested items.
- Testing schedule.
- Test recording procedures.
- Hardware and software requirements.
- Constraints.

The software test plan

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

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

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

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

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

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

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

Software inspections

These involve people examining the source representation with the aim of discovering anomalies and defects.
Inspections not require execution of a system so may be used before implementation.
They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
They have been shown to be an effective technique for discovering program errors.

Inspection success

Many different defects may be discovered in a single inspection. In testing, one defect, may mask another so several executions are required. The reuse domain and programming knowledge so reviewers are likely to have seen the types of error that commonly arise.

Inspections and testing

Inspections and testing are complementary and not opposing verification techniques. Both should be used during the V & V process. Inspections can check conformance with a specification but not conformance with the customer's real requirements. Inspections

cannot check non-functional characteristics such as performance, usability, etc.

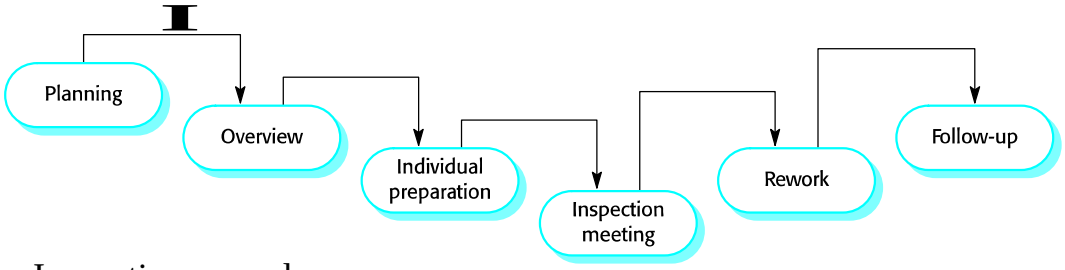
Program inspections

Formalised approach to document reviews. Intended explicitly for defect detection (not correction). Defects may be logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an uninitialised variable) or non-compliance with standards.

Inspection pre-conditions

- A precise specification must be available.
- Team members must be familiar with the organisation standards.
- Syntactically correct code or other system representations must be available.
- An error checklist should be prepared.
- Management must accept that inspection will increase costs early in the software process.
- Management should not use inspections for staff appraisal i.e. finding out who makes mistakes.

The inspection process



Inspection procedure

- System overview presented to inspection team.
- Code and associated documents are distributed to inspection team in advance.
- Inspection takes place and discovered errors are noted.
- Modifications are made to repair discovered errors.
- Re-inspection may or may not be required.

Inspection roles

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

Inspection checklists

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- In general, the 'weaker' the type checking, the larger the checklist.
- Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

Inspection check

Data faults	are all program variables initialized before their values are used? Have all constants been named? Should the lower bound of arrays be 0, 1, or something else? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a delimiter explicitly assigned?
Control faults	for each conditional statement, is the condition correct? Is each loop certain to terminate?

	Are compound statements correctly bracketed? In case statements, are all possible cases accounted for? Input/output faults are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	do all function and procedure calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Segment faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer Inspection checks required?
Exception Management	Have all possible error conditions been taken into account?

Inspection rate

500 statements/hour during overview. 125 source statement/hour during individual
preparation. 90-125 statements/hour can be inspected. Inspection is therefore an expensive process. Inspecting 500 lines costs about 40 man/hours effort - about £2800 at UK rates.

Automated static analysis

- Static analysers are software tools for source text processing.
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

Stages of static analysis

- Control flow analysis. Checks for loops with multiple exit or entry points, finds unreachable code, etc.

- Data use analysis. Detects uninitialised variables, variables written twice without an intervening assignment, variables which are declared but never used, etc.
- Interface analysis. Checks the consistency of routine and procedure declarations and their use.
- Information flow analysis. Identifies the dependencies of output variables. Does not detect anomalies itself but highlights information for code inspection or review
- Path analysis. Identifies paths through the program and sets out the statements executed in that path. Again, potentially useful in the review process
- Both these stages generate vast amounts of information. They must be used with care.

LINT static analysis

```
138% more lint_ex.c
#include <stdio.h>
printarray (Anarray)
int Anarray;
{
printf("%d",Anarray);
}
main ()
{
int Anarray[5]; int i; char c;
printarray (Anarray, i, c);
printarray (Anarray) ;
}
139% cc lint_ex.c
140% lint lint_ex.c
lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) ::
lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) ::
LINT static analysis
lint_ex.c(11)
printf returns value which is always ignored.
```

Use of static analysis

Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler, Less cost-effective for languages like Java that have strong type checking and can therefore detect many errors during compilation.

Verification and formal methods

- Formal methods can be used when a mathematical specification of the system is produced.
- They are the ultimate static verification technique.
- They involve detailed mathematical analysis of the specification and may develop formal arguments that a program conforms to its mathematical specification.

Arguments for formal methods

Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors. They can detect implementation errors before testing when the program is analyzed alongside the specification.

Arguments against formal methods

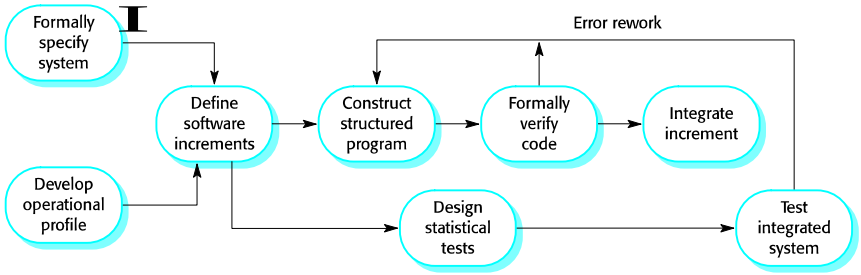
- Require specialized notations that cannot be understood by domain experts.
- Very expensive to develop a specification and even more expensive to show that a program meets that specification.
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.

Cleanroom software development

The name is derived from the 'Cleanroom' process in semiconductor fabrication. The philosophy is defect avoidance rather than defect removal. This software development process is based on:

- Incremental development;
- Formal specification;
- Static verification using correctness arguments;
- Statistical testing to determine program reliability.

The Cleanroom process



Cleanroom process characteristics

- Formal specification using a state transition model.

- Incremental development where the customer prioritises increments.
- Structured programming - limited control and abstraction constructs are used in the program.
- Static verification using rigorous inspections.
- Statistical testing of the system

Formal specification and inspections

- The state based model is a system specification and the inspection process checks the program against this mode. l
- The programming approach is defined so that the correspondence between the model and the system is clear.
- Mathematical arguments (not proofs) are used to increase confidence in the inspection process.

Cleanroom process teams

Specification team*:* Responsible for developing and maintaining the system specification.

Development team*:* Responsible for developing and verifying the software. The software is NOT executed or even compiled during this process.

Certification team*:* Responsible for developing a set of statistical tests to exercise the software after development. Reliability growth models used to determine when reliability is acceptable.

Cleanroom process evaluation

- The results of using the Cleanroom process have been very impressive with few discovered faults in delivered systems.
- Independent assessment shows that the process is no more expensive than other approaches.
- There were fewer errors than in a 'traditional' development process.
- However, the process is not widely used. It is not clear how this approach can be transferred to an environment with less skilled or less motivated software engineers.

The testing process

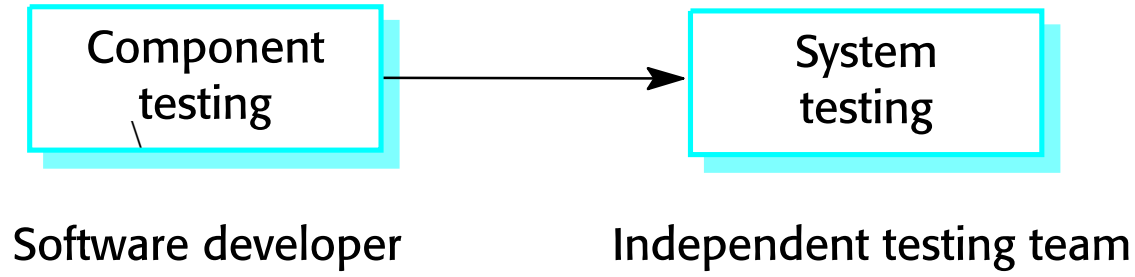
Component testing

- Testing of individual program components;
- Usually the responsibility of the component developer (except sometimes for critical systems);
- Tests are derived from the developer’s experience.

System testing

- Testing of groups of components integrated to create a system or sub-system;
- The responsibility of an independent testing team;
- Tests are based on a system specification.

Testing phases



Defect testing

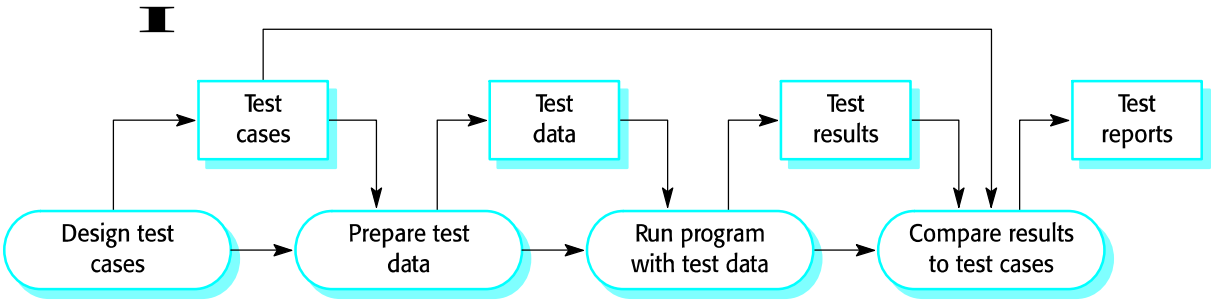
- The goal of defect testing is to discover defects in programs
- A *successful* defect test is a test which causes a program to behave in an anomalous way
- Tests show the presence not the absence of defects

Testing process goals

Validation testing: To demonstrate to the developer and the system customer that the software meets its requirements; a successful test shows that the system operates as intended.

Defect testing: To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification; a successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

The software testing process



Testing policies

Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible,

Testing policies define the approach to be used in selecting system tests:

·All functions accessed through menus should be tested;

·Combinations of functions accessed through the same menu should be tested;

·Where user input is required, all functions must be tested with correct and incorrect input.

System testing

- Involves integrating components to create a system or sub-system.
- May involve testing an increment to be delivered to the customer.
- Two phases:

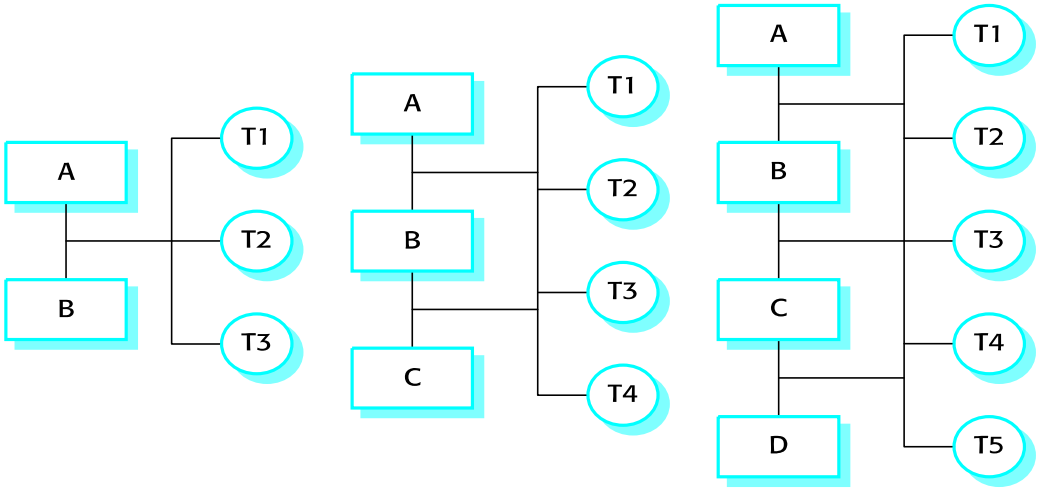
Integration testing - the test team have access to the system source code. The system is tested as components are integrated.
Release testing - the test team test the complete system to be delivered as a black-box.
Integration testing

- Involves building a system from its components and testing it for problems that arise from component interactions.

Top-down integration: Develop the skeleton of the system and populate it with components.
Bottom-up integration: Integrate infrastructure components then add functional components.

- To simplify error localisation, systems should be incrementally integrated.

Incremental integration testing



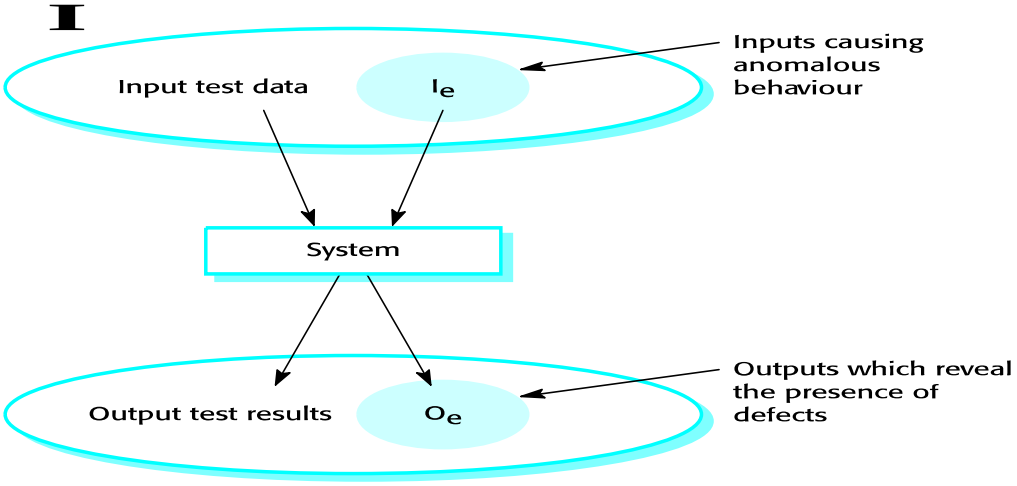
Testing approaches

- Architectural validation: Top-down integration testing is better at discovering errors in the system architecture.
- System demonstration: Top-down integration testing allows a limited demonstration at an early stage in the development.
- Test implementation: Often easier with bottom-up integration testing.
- Test observation: Problems with both approaches. Extra code may be required to observe tests.

Release testing

- The process of testing a release of a system that will be distributed to customers.
- Primary goal is to increase the supplier's confidence that the system meets its requirements.
- Release testing is usually black-box or functional testing
- Based on the system specification only;
- Testers do not have knowledge of the system implementation.

Black-box testing



Testing guidelines

Testing guidelines are hints for the testing team to help them choose tests that will reveal defects in the system

- Choose inputs that force the system to generate all error messages;
- Design inputs that cause buffers to overflow;
- Repeat the same input or input series several times;
- Force invalid outputs to be generated;
- Force computation results to be too large or too small.

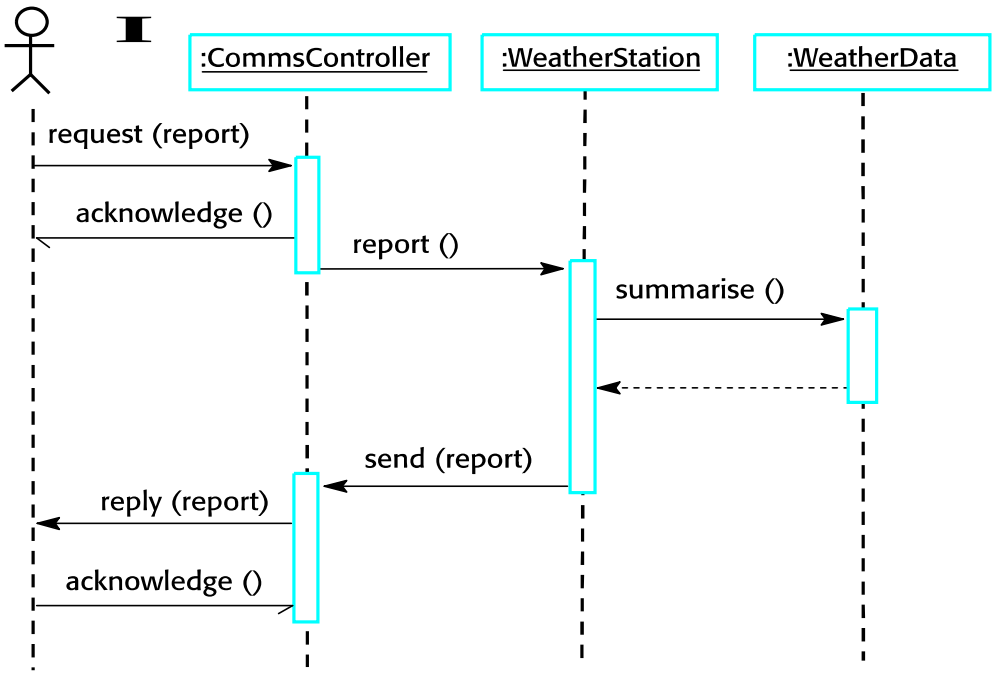
System tests

1. Test the login mechanism using correct and incorrect logins to check that valid users are accepted and invalid users are rejected.
2. Test the search facility using different queries against known sources to check that the search mechanism is actually finding documents.
3. Test the system presentation facility to check that information about documents is displayed properly.
4. Test the mechanism to request permission for downloading.
5. Test the e-mail response indicating that the downloaded document is available.

Use cases

Use cases can be a basis for deriving the tests for a system. They help identify operations to be tested and help design the required test cases. From an associated sequence diagram, the inputs and outputs to be created for the tests can be identified.

Collect weather data sequence chart



Performance testing

Part of release testing may involve testing the emergent properties of a system, such as performance and reliability. Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

Stress testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light.
- Stressing the system test failure behaviour.. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data.
- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as a network becomes overloaded.

Component testing

- Component or unit testing is the process of testing individual components in isolation.

- It is a defect testing process.
- Components may be:
- Individual functions or methods within an object;
- Object classes with several attributes and methods;
- Composite components with defined interfaces used to access their functionality.

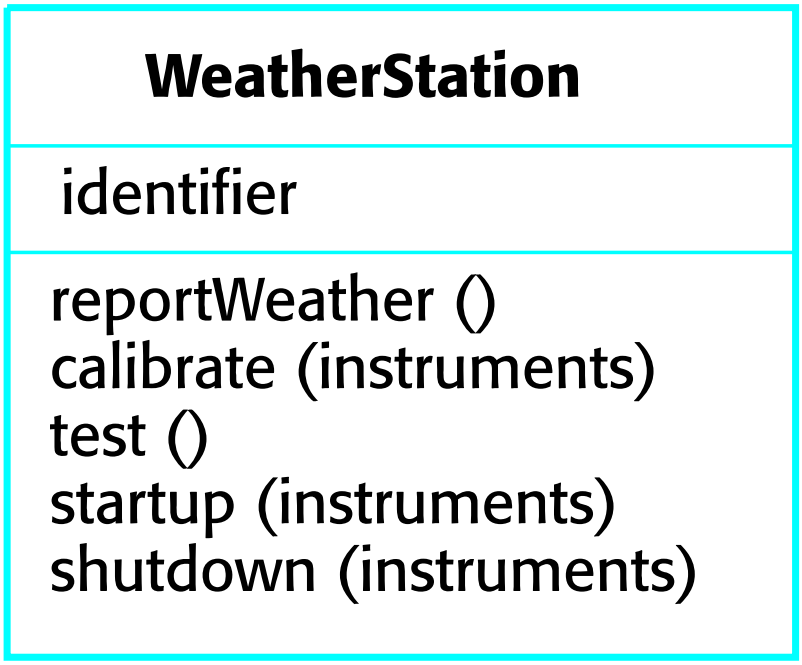
Object class testing

Complete test coverage of a class involves

- Testing all operations associated with an object;
- Setting and interrogating all object attributes;
- Exercising the object in all possible states.

Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.

Weather station object interface



Weather station testing

·Need to define test cases for reportWeather, calibrate, test, startup and shutdown.

·Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions

For example:

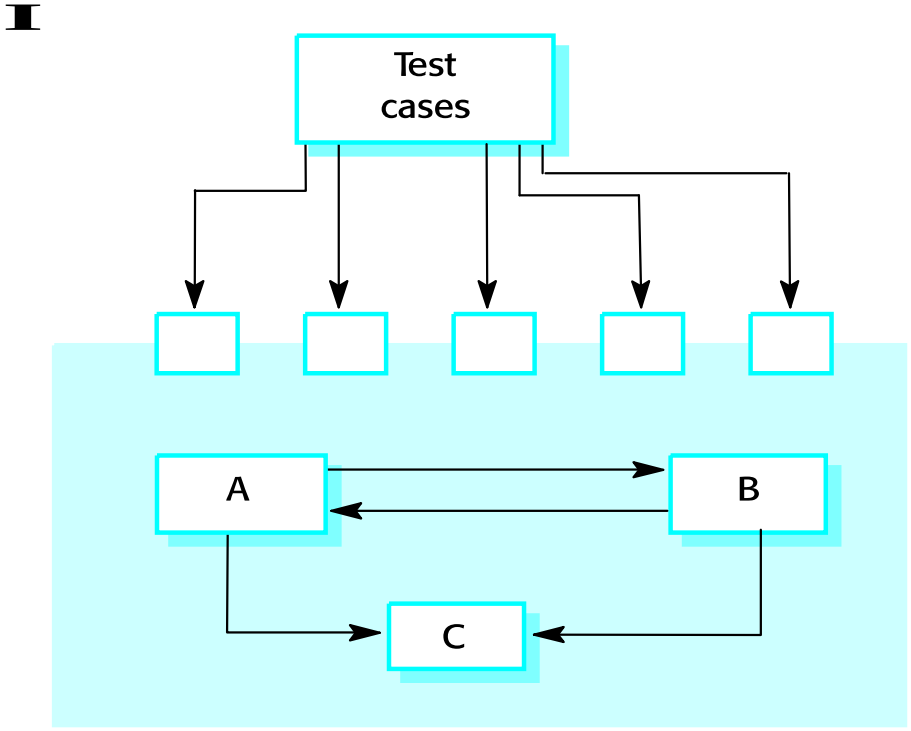
·Waiting -> Calibrating -> Testing -> Transmitting -> Waiting.

Interface testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.

- Particularly important for object-oriented development as objects are defined by their interfaces.

Interface testing



Interface types

- Parameter interfaces: Data passed from one procedure to another.
- Shared memory interfaces: Block of memory is shared between procedures or functions.
- Procedural interfaces: Sub-system encapsulates a set of procedures to be called by other sub-systems.
- Message passing interfaces: Sub-systems request services from other sub-systems

Interface errors

- Interface misuse: A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
- Interface misunderstanding: A calling component embeds assumptions about the behaviour of the called component which are incorrect.

- Timing errors: The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Use stress testing in message passing systems.
- In shared memory systems, vary the order in which components are activated.

Test case design

- Involves designing the test cases (inputs and outputs) used to test the system.
- The goal of test case design is to create a set of tests that are effective in validation and defect testing.
- Design approaches:
- Requirements-based testing;
- Partition testing;
- Structural testing.

Requirements based testing

A general principle of requirements engineering is that requirements should be testable. Requirements-based testing is a validation testing technique where you consider each requirement and derive a set of tests for that requirement.

LIBSYS requirementsLIBSYS tests

The user shall be able to search either all of the initial set of databases or select a subset from it.

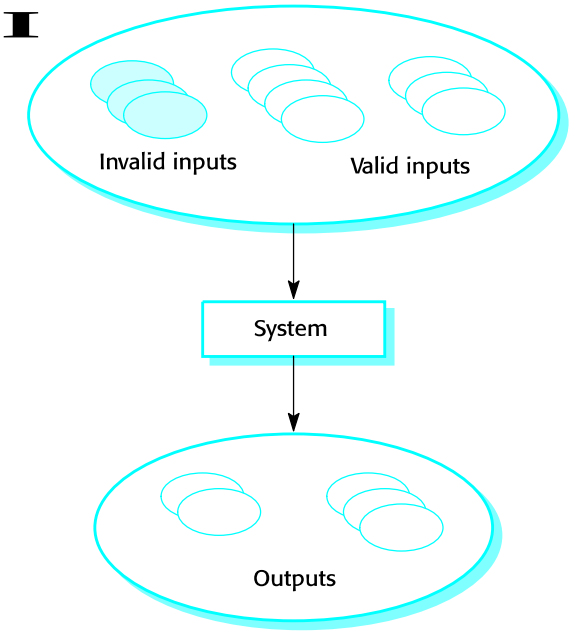
The system shall provide appropriate viewers for the user to read documents in the document store.

Every order shall be allocated a unique identifier (ORDER_ID) that the user shall be able to copy to the accountõ permanent storage area.

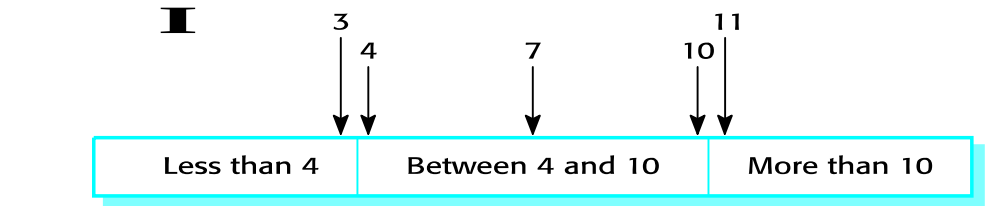
Partition testing

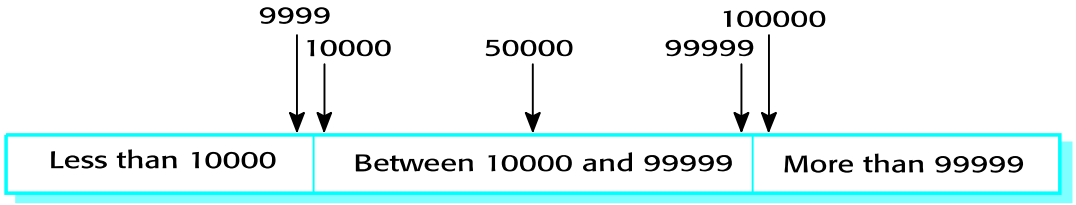
- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
  - Initiate user search for searches for items that are known to be present and known not to be present, where the set of databases includes 1 database.
  - Initiate user searches for items that are known to be present and known not to be present, where the set of databases includes 2 databases
  - Initiate user searches for items that are known to be present and known not to be present where the set of databases includes more than 2 databases.
  - Select one database from the set of databases and initiate user searches for items that are known to be present and known not to be present.
  - Select more than one database from the set of databases and initiate searches for items that are known to be present and known not to be present.
- Test cases should be chosen from each partition.

Equivalence partitioning



Equivalence partitions





Search routine specification

procedure Search (Key : ELEM ; T: SEQ of ELEM;
Found : in out BOOLEAN; L: in out ELEM_INDEX) ;

Pre-condition
-- the sequence has at least one element
T'FIRST <= T'LAST
Post-condition
-- the element is found and is referenced by L
(Found and T (L) = Key)
or
-- the element is not in the array
(not Found and
not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key))

Search routine - input partitions

- Inputs which conform to the pre-conditions.
- Inputs where a pre-condition does not hold.
- Inputs where the key element is a member of the array.
- Inputs where the key element is not a member of the array.

Testing guidelines (sequences)

- Test software with sequences which have only a single value.
- Use sequences of different sizes in different tests.
- Derive tests so that the first, middle and last elements of the sequence are accessed.
- Test with sequences of zero length.

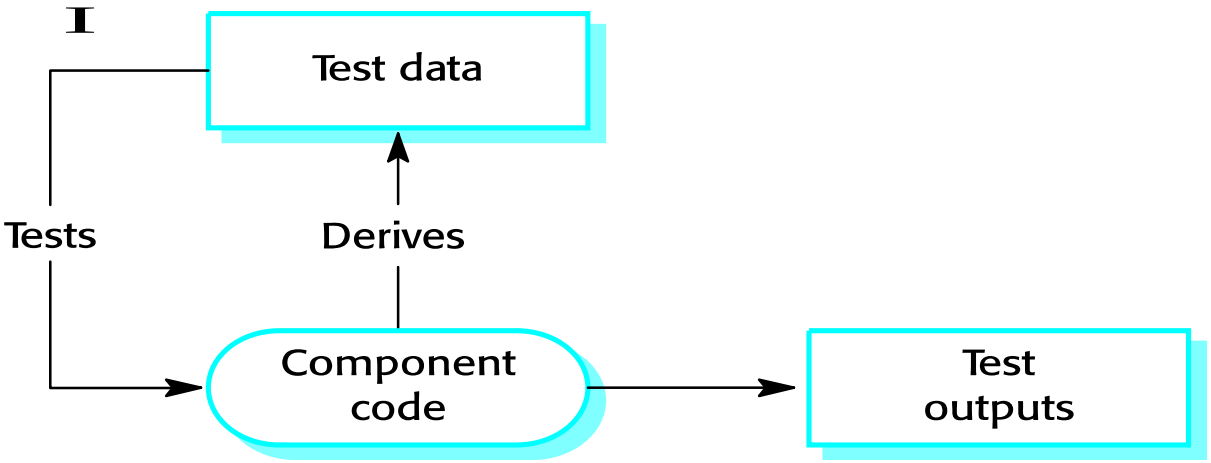
Search routine - input partitions

Sequence	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Structural testing

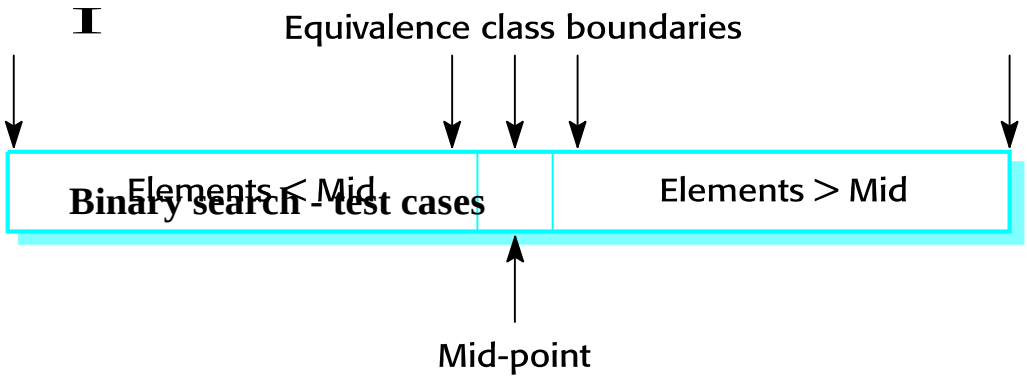
Sometime called white-box testing. Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases. Objective is to exercise all program statements (not all path combinations)



Binary search - equiv. partitions

- Pre-conditions satisfied, key element in array.
- Pre-conditions satisfied, key element not in array.
- Pre-conditions unsatisfied, key element in array.
- Pre-conditions unsatisfied, key element not in array.
- Input array has a single value.
- Input array has an even number of values.

- Input array has an odd number of values



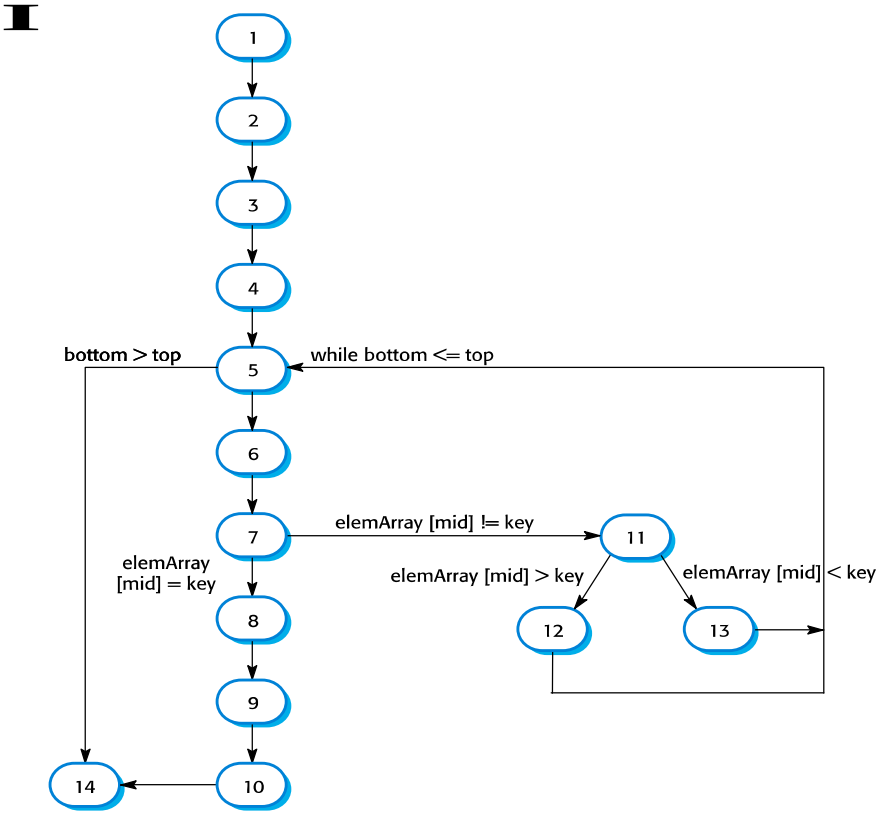
Binary search - test cases

Input array (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Path testing

- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once.
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control.
- Statements with conditions are therefore nodes in the flow graph.

Binary search flow graph



Independent paths

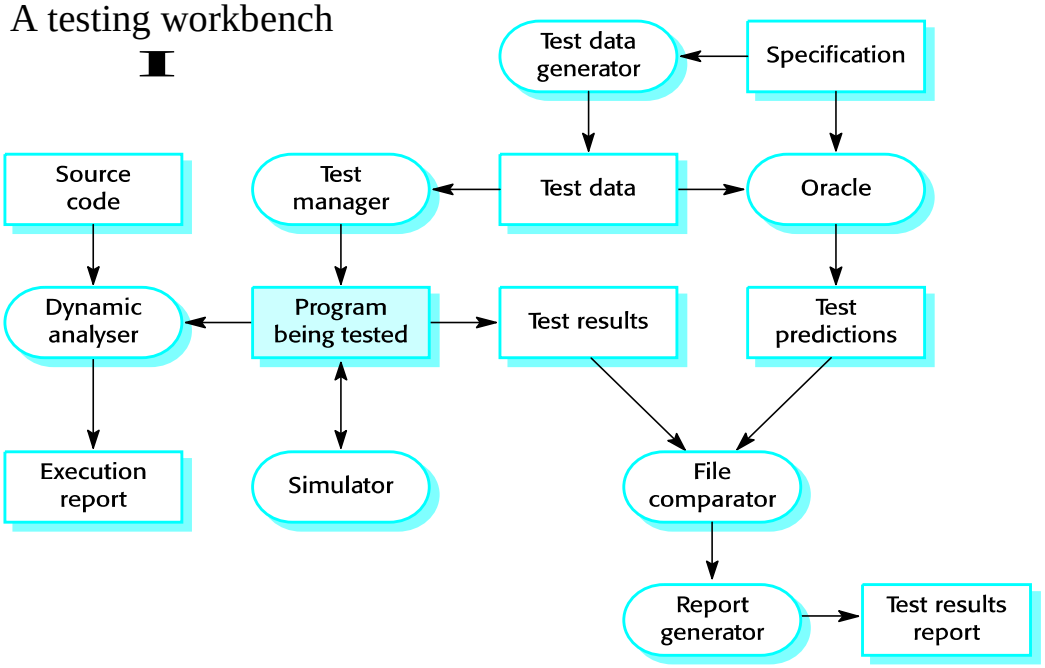
- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
- 1, 2, 3, 4, 5, 14
- 1, 2, 3, 4, 5, 6, 7, 11, 12, 5, …
- 1, 2, 3, 4, 6, 7, 2, 11, 13, 5, …

- Test cases should be derived so that all of these paths are executed
- A dynamic program analyser may be used to check that paths have been executed

Test automation

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs.
- Systems such as Junit support the automatic execution of tests.
- Most testing workbenches are open systems because testing needs are organisation-specific.
- They are sometimes difficult to integrate with closed design and analysis workbenches.

A testing workbench



Testing workbench adaptation

- Scripts may be developed for user interface simulators and patterns for test data generators.
- Test outputs may have to be prepared manually for comparison.
- Special-purpose file comparators may be developed.

UNIT – 8
MANAGEMENT

Managing people

- Managing people working as individuals and in groups
- To explain some of the issues involved in selecting and retaining staff
- To describe factors that influence individual motivation
- To discuss key issues of team working including composition, cohesiveness and communications
- To introduce the people capability maturity model (P-CMM) - a framework for enhancing the capabilities of people in an organization.

People in the process

People are an organization's most important assets. The tasks of a manager are essentially people-oriented. Unless there is some understanding of people, management will be unsuccessful. Poor people management is an important contributor to project failure.

People management factors

Consistency: Team members should all be treated in a comparable way without favorites or discrimination.
Respect: Different team members have different skills and these differences should be respected.
Inclusion: Involve all team members and make sure that people's views are considered.
Honesty: You should always be honest about what is going well and what is going badly in a project.

Selecting staff

An important project management task is team selection.
Information on selection comes from:

- Information provided by the candidates.
- Information gained by interviewing and talking with candidates.
- Recommendations and comments from other people who know or who have worked with the candidates.

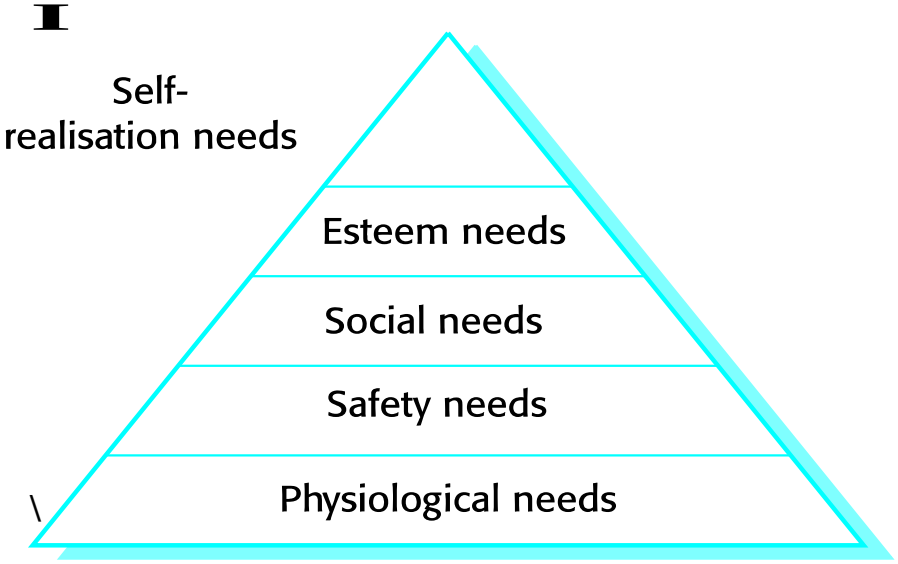
Motivating people

An important role of a manager is to motivate the people working on a project.
Motivation is a complex issue but it appears that there are different types of motivation based on:

- Basic needs (e.g. food, sleep, etc.);

- Personal needs (e.g. respect, self-esteem);
- Social needs (e.g. to be accepted as part of a group).

Human needs hierarchy



Need satisfaction

Social

- Provide communal facilities;
- Allow informal communications.

Esteem

- Recognition of achievements;
- Appropriate rewards.
- Self-realizationTraining - people want to learn more;Responsibility

Personality types

The needs hierarchy is almost certainly an over-simplification of motivation in practice.

Motivation should also take into account different personality types:

- Task-oriented;
- Self-oriented;
- Interaction-oriented.

Task-oriented.

- The motivation for doing the work is the work itself;

Self-oriented.

- The work is a means to an end which is the achievement of individual goals - e.g. to get rich, to play tennis, to travel etc.; Interaction-oriented
- The principal motivation is the presence and actions of co-workers. People go to work because they like to go to work.

Motivation balance

Individual motivations are made up of elements of each class. The balance can change depending on personal circumstances and external events. However, people are not just motivated by personal factors but also by being part of a group and culture. People go to work because they are motivated by the people that they work with.

Managing groups

Most software engineering is a group activity

- The development schedule for most non-trivial software projects is such that they cannot be completed by one person working alone.

Group interaction is a key determinant of group performance. Flexibility in group composition is limited

- Managers must do the best they can with available people.

Factors influencing group working

- Group composition.
- Group cohesiveness.
- Group communications.
- Group organization.

Group composition

Group composed of members who share the same motivation can be problematic

- Task-oriented - everyone wants to do their own thing;
- Self-oriented - everyone wants to be the boss;
- Interaction-oriented - too much chatting, not enough work.

An effective group has a balance of all types. This can be difficult to achieve software engineers are often task-oriented. Interaction-oriented people are very important as they can detect and defuse tensions that arise.

Group composition

In creating a group for assistive technology development, Alice is aware of the importance of selecting members with complementary personalities. When interviewing people, she tried to assess whether they were task oriented, self-oriented and interaction oriented. She felt that she was primarily a self-oriented type as she felt that this project was a way in which she would be noticed by senior management and promoted. She therefore looked for 1 or perhaps 2 interaction-oriented personalities with the remainder task oriented. The final assessment that she arrived at was:

Alice Š self-oriented
Brian Š task-oriented
Bob Š task-oriented
Carol Š interaction-oriented
Dorothy Š self-oriented
Ed Š interaction-oriented
Fred Š task-oriented

Group leadership

Leadership depends on respect not titular status. There may be both a technical and an administrative leader. Democratic leadership is more effective that autocratic leadership.

Group cohesiveness

In a cohesive group, members consider the group to be more important than any individual in it.

The advantages of a cohesive group are:

- Group quality standards can be developed;
- Group members work closely together so inhibitions caused by ignorance are reduced;
- Team members learn from each other and get to know each other's work;
- Egoless programming where members strive to improve each other's programs can be practised.

Team spirit

Developing cohesiveness

Cohesiveness is influenced by factors such as the organizational culture and the personalities in the group. Cohesiveness can be encouraged through

- Social events;
- Developing a group identity and territory;
- Explicit team-building activities.

Openness with information is a simple way of ensuring all group members feel part of the group.

Group loyalties

- Group members tend to be loyal to cohesive groups.
- 'Groupthink' is preservation of group irrespective of technical or organizational considerations.
- Management should act positively to avoid groupthink by forcing external involvement with each group.

Group communications

Good communications are essential for effective group working. Information must be exchanged on the status of work, design decisions and changes to previous decisions. Good communications also strengthens group cohesion as it promotes understanding.

Group size: The larger the group, the harder it is for people to communicate with other group members.
Group structure: Communication is better in informally structured groups than in hierarchically structured groups.
Group composition: Communication is better when there are different personality types in a group and when groups are mixed rather than single sex.
The physical work environment: Good workplace organization can help encourage communications.

Group organization

Small software engineering groups are usually organized informally without a rigid structure. For large projects, there may be a hierarchical structure where different groups are responsible for different sub-projects.

Informal groups

- The group acts as a whole and comes to a consensus on decisions affecting the system.
- The group leader serves as the external interface of the group but does not allocate specific work items.
- Rather, work is discussed by the group as a whole and tasks are allocated according to ability and experience.
- This approach is successful for groups where all members are experienced and competent.

Extreme programming groups

- Extreme programming groups are variants of an informal, democratic organization.
- In extreme programming groups, some ‘management’ decisions are devolved to group members.
- Programmers work in pairs and take a collective responsibility for code that is developed.

Chief programmer teams

Consist of a kernel of specialists helped by others added to the project as required. The motivation behind their development is the wide difference in ability in different programmers. Chief programmer teams provide a supporting environment for very able programmers to be responsible for most of the system development.

Problems

This chief programmer approach, in different forms, has been successful in some settings.

However, it suffers from a number of problems

- Talented designers and programmers are hard to find. Without exceptional people in these roles, the approach will fail;
- Other group members may resent the chief programmer taking the credit for success so may deliberately undermine his/her role;
- There is a high project risk as the project will fail if both the chief and deputy programmer are unavailable.
- The organizational structures and grades in a company may be unable to accommodate this type of group.

Working environments

The physical workplace provision has an important effect on individual productivity and satisfaction

- Comfort;
- Privacy;
- Facilities.

Health and safety considerations must be taken into account

- Lighting;
- Heating;
- Furniture.

Environmental factors

Privacy - each engineer requires an area for uninterrupted work.

Outside awareness - people prefer to work in natural light.

Personalization - individuals adopt different working practices and like to organize their environment in different ways.

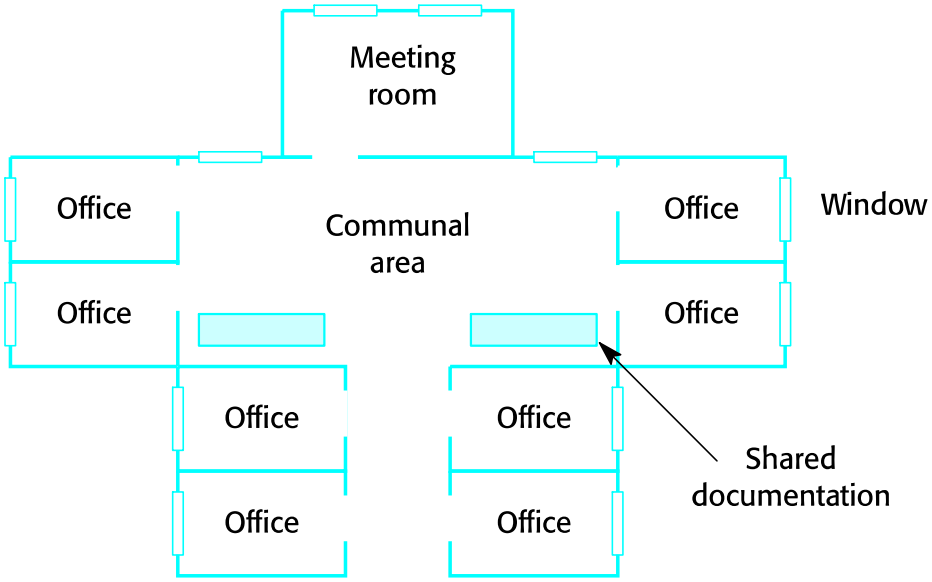
Workspace organization

Workspaces should provide private spaces where people can work without interruption

- Providing individual offices for staff has been shown to increase productivity.

However, teams working together also require spaces where formal and informal meetings can be held.

Office layout



The People Capability Maturity Model

- Intended as a framework for managing the development of people involved in software development.

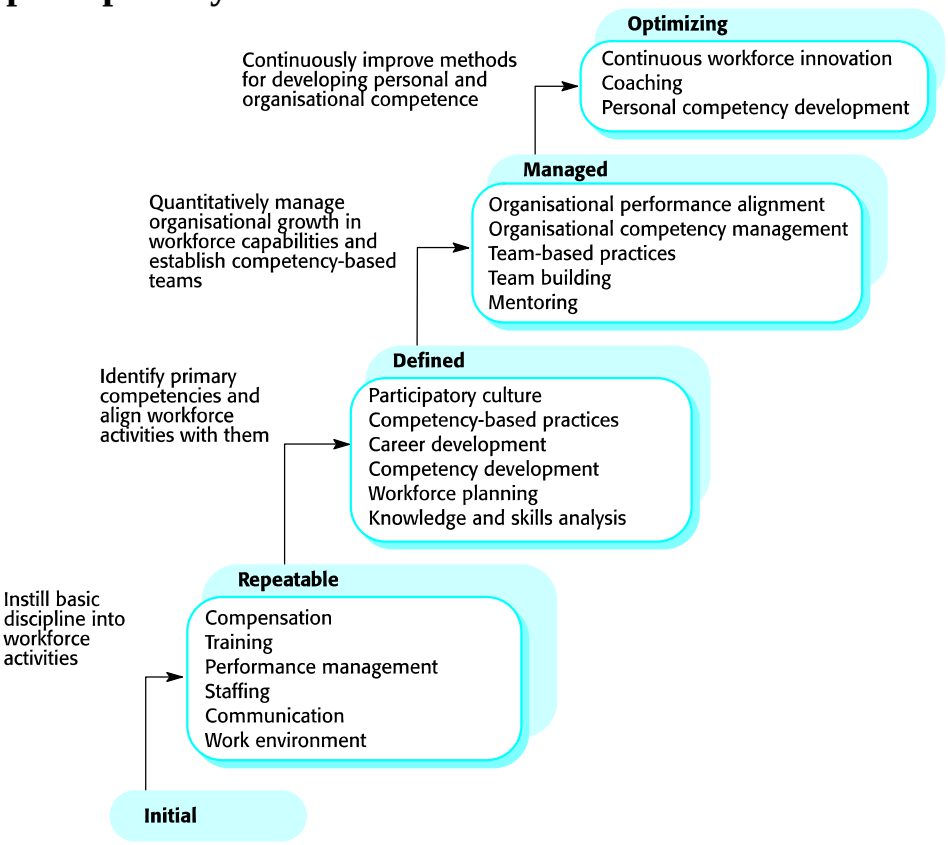
P-CMM Objectives

- To improve organizational capability by improving workforce capability.
- To ensure that software development capability is not reliant on a small number of individuals.
- To align the motivation of individuals with that of the organization.
- To help retain people with critical knowledge and skills.

P-CMM levels

Five stage model
Initial. Ad-hoc people management
Repeatable. Policies developed for capability improvement
Defined. Standardized people management across the organization
Managed. Quantitative goals for people management in place
Optimizing. Continuous focus on improving individual competence and workforce motivation

The people capability model



Software cost estimation

Fundamental estimation questions

- How much effort is required to complete an activity?
- How much calendar time is needed to complete an activity?
- What is the total cost of an activity?
- Project estimation and scheduling are interleaved management activities?

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 organizational, economic, political and business considerations influence the price charged.

Software pricing factors

Market opportunity	A development organisation may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the opportunity of more profit later. The experience gained may allow new products to be developed.
Cost estimate uncertainty	If an organisation is unsure of its cost estimate, it may increase its price by some contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organisation may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business.

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

What's a line 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?

What programs should be counted as part of the system?
This model assumes that there is a linear relationship between system size and volume of documentation.

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.

System development times

	Analysis	Design	Coding	Testing	Documentation
Assembly code	3 weeks	5 weeks	8 weeks	10 weeks	2 weeks
High-level language	3 weeks	5 weeks	4 weeks	6 weeks	2 weeks

	Size	Effort	Productivity
Assembly code	5000 lines	28 weeks	714 lines/month
High-level language	1500 lines	20 weeks	300 lines/month

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.

UPC=∑ (no. of elements of given type) X (weight).
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

- LOC = AVC * number of function points;
- AVC is a language-dependent factor varying from 200-300 for assemble language to 2-40 for a 4GL;

FPs is very subjective. They depend on the estimator

- Automatic function-point counting is impossible.

Object points

Object points (alternatively named application points) are an alternative function-related measure to function points when 4Gls or similar languages are used for development.
Object points are NOT the same as object classes.

- The number of object points in a program is a weighted estimate of
- The number of separate screens that are displayed;
- The number of reports that are produced by the system;
- The number of program modules that must be developed to supplement the database code;

Object point estimation

Object points are easier to estimate from a specification than function points as they are simply concerned with screens, reports and programming language modules. They can therefore be estimated at a fairly early point in the development process. At this stage, it is very difficult to estimate the number of lines of code in a system.

Productivity estimates

- Real-time embedded systems, 40-160 LOC/P-month.
- Systems programs, 150-400 LOC/P-month.
- Commercial applications, 200-900 LOC/P-month.
- In object points, productivity has been measured between 4 and 50 object points/month depending on tool support and developer capability.

Factors affecting productivity

Application domain experience	Knowledge of the application domain is essential for effective software development. Engineers who already understand a domain are likely to be the most productive.
Process quality	The development process used can have a significant effect on productivity. This is covered in Chapter 28.
Project size	The larger a project, the more time required for team communications. Less time is available for development so individual productivity is reduced.
Technology support	Good support technology such as CASE tools, configuration management systems, etc. can improve productivity.
Working environment	As I discussed in Chapter 25, a quiet working environment with private work areas contributes to improved productivity.

Quality and productivity

All metrics based on volume/unit time are flawed because they do not take quality into account. Productivity may generally be increased at the cost of quality. It is not clear how productivity/quality metrics are related. If requirements are constantly changing then an approach based on counting lines of code is not meaningful as the program itself is not static;

Estimation techniques

There is no simple way to make an accurate estimate of the effort required to develop a software system

- Initial estimates are based on inadequate information in a user requirements definition;
- The software may run on unfamiliar computers or use new technology;
- The people in the project may be unknown.

Project cost estimates may be self-fulfilling

- The estimate defines the budget and the product is adjusted to meet the budget.

Changing technologies

Changing technologies may mean that previous estimating experience does not carry over to new systems

- Distributed object systems rather than mainframe systems;
- Use of web services;
- Use of ERP or database-centered systems;
- Use of off-the-shelf software;
- Development for and with reuse;

- Development using scripting languages;
- The use of CASE tools and program generators.

Estimation techniques

- Algorithmic cost modeling.
- Expert judgment.
- Estimation by analogy.
- Parkinson's Law.
- Pricing to win.

Estimation techniques

Algorithmic cost modelling	A model based on historical cost information that relates some software metric (usually its size) to the project cost is used. An estimate is made of that metric and the model predicts the effort required.
Expert judgement	Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. These estimates are compared and discussed. The estimation process iterates until an agreed estimate is reached.
Estimation by analogy	This technique is applicable when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects. Myers (Myers 1989) gives a very clear description of this approach.
ParkinsonÕs Law	ParkinsonÕsLaw states that work expands to fill the time available. The cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months.
Pricing to win	The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customerÕs budget and not on the software functionality.

Pricing to win

The project costs whatever the customer has to spend on it.
Advantages:
You get the contract.
Disadvantages:
The probability that the customer gets the system he or she wants is small. Costs do not accurately reflect the work required.

Top-down and bottom-up estimation

Any of these approaches may be used top-down or bottom-up.
Top-down
Start at the system level and assess the overall system functionality and how this is delivered through sub-systems.
Bottom-up
Start at the component level and estimate the effort required for each component. Add these efforts to reach a final estimate.

Top-down estimation

- Usable without knowledge of the system architecture and the components that might be part of the system.
- Takes into account costs such as integration, configuration management and documentation.
- Can underestimate the cost of solving difficult low-level technical problems.

Bottom-up estimation

- Usable when the architecture of the system is known and components identified.
- This can be an accurate method if the system has been designed in detail.
- It may underestimate the costs of system level activities such as integration and documentation.

Estimation methods

Each method has strengths and weaknesses.
Estimation should be based on several methods.
If these do not return approximately the same result, then you have insufficient information available to make an estimate.
Some action should be taken to find out more in order to make more accurate estimates.
Pricing to win is sometimes the only applicable method.

Pricing to win

This approach may seem unethical and un-businesslike.
However, when detailed information is lacking it may be the only appropriate strategy.
The project cost is agreed on the basis of an outline proposal and the development is constrained by that cost.
A detailed specification may be negotiated or an evolutionary approach used for system development.

Algorithmic cost modeling

Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers:

- Effort = A ´ SizeB ´ M
- A is an organisation-dependent constant, B reflects the disproportionate effort for large projects and M is a multiplier reflecting product, process and people attributes.

The most commonly used product attribute for cost estimation is code size. Most models are similar but they use different values for A, B and M.

Estimation accuracy

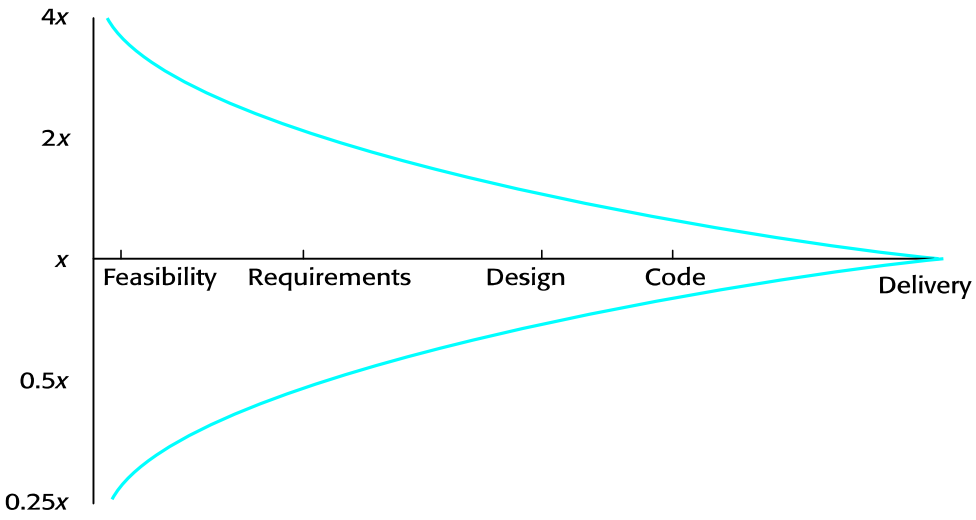
The size of a software system can only be known accurately when it is finished.

Several factors influence the final size

- Use of COTS and components;
- Programming language;
- Distribution of system.

As the development process progresses then the size estimate becomes more accurate.

Estimate uncertainty



The 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	$PM = 2.4 (KDSI)^{1.05} \times M$	Well-understood applications developed by small teams.
Moderate	$PM = 3.0 (KDSI)^{1.12} \times M$	More complex projects where team members may have limited experience of related systems.
Embedded	$PM = 3.6 (KDSI)^{1.20} \times M$	Complex projects where the software is part of a strongly coupled complex of hardware, software, regulations and operational procedures.

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 software engineering practice and COCOMO 2 is designed to accommodate different approaches to software development.

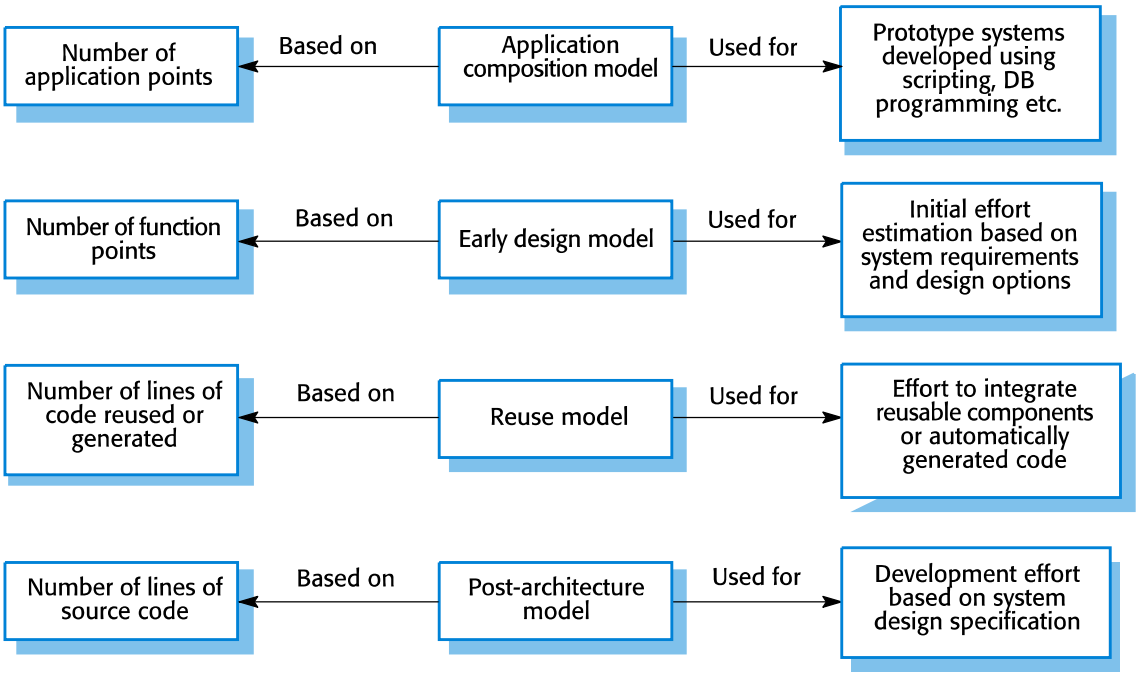
COCOMO 2 models

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.

Object point productivity

DeveloperÕs experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NOP/month)	4	7	13	25	50

Early design model

Estimates can be made after the requirements have been agreed.
Based on a standard formula for algorithmic models

- PM = A ´ SizeB ´ M where
- M = PERS ´ RCPX ´ RUSE ´ PDIF ´ PREX ´ FCIL ´ 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.

- RCPX - product reliability and complexity;
- RUSE - the reuse required;
- PDIF - platform difficulty;
- PREX - personnel experience;
- PERS - personnel capability;
- SCED - required schedule;
- FCIL - the team support facilities.

The reuse model

- Takes into account black-box code that is reused without change and code that has to be adapted to integrate it with new code.

There are two versions:

- Black-box reuse where code is not modified. An effort estimate (PM) is computed.
- White-box reuse where code is modified. A size estimate equivalent to the number of lines of new source code is computed.

This then adjusts the size estimate for new code.

Reuse model estimates 1

For generated code:
PM = (ASLOC * AT/100)/ATPROD
ASLOC is the number of lines of generated code
AT is the percentage of code automatically generated.
ATPROD is the productivity of engineers in integrating this code.

Reuse model estimates 2

When code has to be understood and integrated:
ESLOC = ASLOC * (1-AT/100) * AAM.
ASLOC and AT as before.
AAM is the adaptation adjustment multiplier computed from the costs of changing the reused code, the costs of understanding how to integrate the code and the costs of reuse decision making.

The exponent term

This depends on 5 scale factors (see next slide). Their sum/100 is added to 1.01

A company takes on a project in a new domain. The client has not defined the process to be used and has not allowed time for risk analysis. The company has a CMM level 2 rating.

- Precedent ness - new project (4)
- Development flexibility - no client involvement - Very high (1)
- Architecture/risk resolution - No risk analysis - V. Low .(5)
- Team cohesion - new team - nominal (3)
- Process maturity - some control - nominal (3)

Scale factor is therefore 1.17.

Exponent scale factors

Precedentedness	Reflects the previous experience of the organisation with this type of project. Very low means no previous experience, Extra high means that the organisation is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; Extra high means that the client only sets general goals.
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis, Extra high means a complete a thorough risk analysis.
Team cohesion	Reflects how well the development team know each other and work together. Very low means very difficult interactions, Extra high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organisation. The computation of this value depends on the CMM Maturity Questionnaire but an estimate can be achieved by subtracting the CMM process maturity level from 5.

Multipliers

Product attributes : Concerned with required characteristics of the software product being developed.

Computer attributes: Constraints imposed on the software by the hardware platform.

Personnel attributes: Multipliers that take the experience and capabilities of the people working on the project into account.

Project attributes: Concerned with the particular characteristics of the software development project.

Project planning

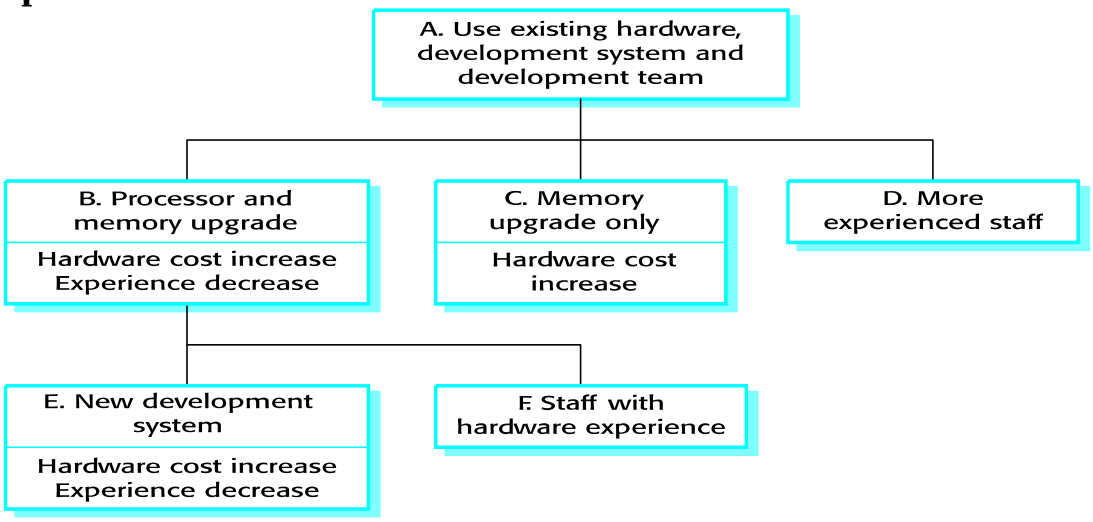
Algorithmic cost models provide a basis for project planning as they allow alternative strategies to be compared.

- Embedded spacecraft system Must be reliable;
- Must minimize weight (number of chips);
- Multipliers on reliability and computer constraints > 1.

Cost components

Target hardware;Development platform;Development effort

Management options



Management option costs

Option choice

Option	RELY	STOR	TIME	TOOLS	LTEX	Total effort	Software cost	Hardware cost	Total cost
A	1.39	1.06	1.11	0.86	1	63	949393	100000	1049393
B	1.39	1	1	1.12	1.22	88	1313550	120000	1402025
C	1.39	1	1.11	0.86	1	60	895653	105000	1000653
D	*1.39*	*1.06*	*1.11*	*0.86*	*0.84*	*51*	*769008*	*100000*	*897490*
E	1.39	1	1	0.72	1.22	56	844425	220000	1044159
F	1.39	1	1	1.12	0.84	57	851180	120000	1002706

Option D (use more experienced staff) appears to be the best alternative

- However, it has a high associated risk as experienced staff may be difficult to find.

Option C (upgrade memory) has a lower cost saving but very low risk.

Overall, the model reveals the importance of staff experience in software development.

Project duration and staffing

As well as effort estimation, managers must estimate the calendar time required completing a project and when staff will be required. Calendar time can be estimated using a COCOMO 2 formula

- TDEV = 3 ´ (PM)(0.33+0.2*(B-1.01))
- PM is the effort computation and B is the exponent computed as discussed above (B is 1 for the early prototyping model). This computation predicts the nominal schedule for the project.

The time required is independent of the number of people working on the project.

Staffing requirements

Staff required can't be computed by diving the development time by the required schedule. The number of people working on a project varies depending on the phase of the project. The more people who work on the project, the more total effort is usually required. A very rapid build-up of people often correlates with schedule slippage

Key points

- There is not a simple relationship between the price charged for a system and its development costs.
- Factors affecting productivity include individual aptitude, domain experience, the development project, the project size, tool support and the working environment.
- Software may be priced to gain a contract and the functionality adjusted to the price.
- Different techniques of cost estimation should be used when estimating costs.
- The COCOMO model takes project, product, personnel and hardware attributes into account when predicting effort required.
- Algorithmic cost models support quantitative option analysis as they allow the costs of different options to be compared.
- The time to complete a project is not proportional to the number of people working on the project.