

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering
[Subject Code: CT 601]

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

Overall Course Outline:

SOFTWARE ENGINEERING

CT 601

Lecture : 3

Year : III

Tutorial : 1

Part : I

Practical : 1.5

Course Objectives:

This course provides a systematic approach towards planning, development, implementation and maintenance of system, also help developing software projects.

1. Software Process and requirements (12 hours)

1.1. Software crisis

1.2. Software characteristics

1.3. Software quality attributes

1.4. Software process model

1.5. Process iteration

1.6. process activities

1.7. Computer-aided software engineering

1.8. Functional and non –functional requirements

1.9. User requirements

1.10. System requirement

1.11. Interface specification

1.12. The software requirements documents

1.13. Feasibility study

1.14. Requirements elicitation and analysis

1.15. Requirements validation and management

2. System models (3 hours)

2.1. Context models

2.2. Behavioural models

2.3. Data and object models

3. Architectural design (6 hours)

3.1. Architectural design decisions

3.2. System organization

3.3. Modular decomposition styles

3.4. Control styles

3.5. Reference architectures

3.6. Multiprocessor architecture

3.7. Client –server architectures

3.8. Distributed object architectures

3.9. Inter-organizational distributed computing

4. Real –time software design (3 hours)

4.1. System design

4.2. Real-time operating systems

4.3. Monitoring and control systems

4.4. Data acquisition systems

5. Software Reuse (3 hours)

5.1. The reuse landscape

5.2. Design patterns

5.3. Generator –based reuse

5.4. Application frameworks

5.5. Application system reuse

6. Component-based software engineering (2 hours)

6.1. Components and components models

6.2. The CBSE process

6.3. Component composition

7. Verification and validation (3 hours)

7.1. Planning verification and validation

7.2. Software inspections

7.3. Verification and formal methods

7.4. Critical System verification and validation

8. Software Testing and cost Estimation (4 hours)

8.1. System testing

8.2. Component testing

8.3. Test case design

8.4. Test automation

8.5. Metrics for testing

8.6. Software productivity

8.7. Estimation techniques

8.8. Algorithmic cost modeling

8.9. Project duration and staffing

9. Quality management (5 hours)

9.1. Quality concepts

9.2. Software quality assurance

9.3. Software reviews

9.4. Formal technical reviews

9.5. Formal approaches to SQA

9.6. Statistical software quality assurance

9.7. Software reliability

9.8. A framework for software metrics

9.9. Matrices for analysis and design model

9.10. ISO standards

9.11. CMMI

9.12. SQA plan

9.13. Software certification

10. Configuration Management (2 hours)

10.1. Configuration management planning

10.2. Change management

10.3. Version and release management

10.4. System building

10.5. CASE tools for configuration management

Practical

The laboratory exercises shall include projects on requirements, analysis and designing of software system. Choice of project depend upon teacher and student, case studies shall be included too. Guest lecture from software industry in the practical session.

References:

1. Ian Sommerville, Software Engineering , Latest edition
2. Roger S. Pressman, Software Engineering –A Practitioner's Approach, Latest edition
3. Pankaj Jalote, Software Engineering-A precise approach, Latest edition
4. Rajib Mall, Fundamental of Software Engineering, Latest edition

Evaluation Scheme:

The questions will cover all the chapters in syllabus. The evaluation scheme will be as indicated in the table below:

Chapters	Hours	Marks distribution*
1	12	20
2	3	5
3	6	10
4	3	5
5	3	5
6	2	3
7	5	10
8	4	8
9	5	10
10	2	4
Total	45	80

*There may be minor deviation in marks distribution

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering
Chapter One

Software Process and requirements

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

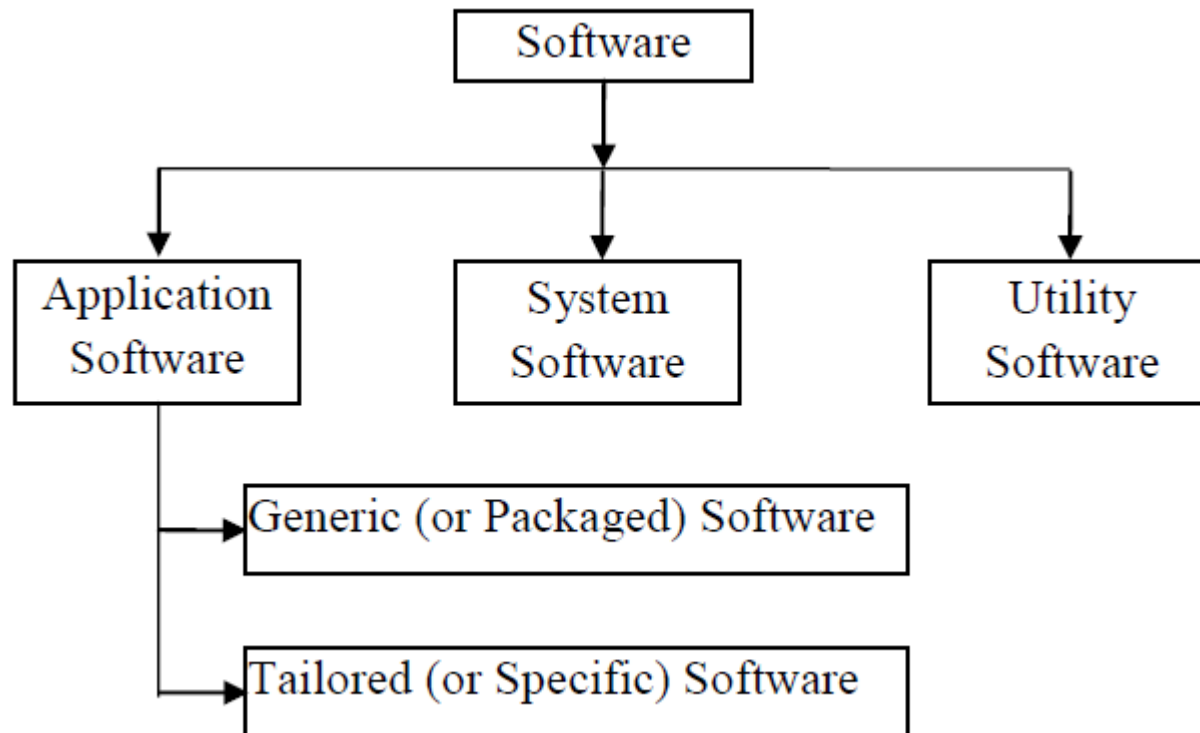
Chapter One: **Software Process and requirements**

Course Outline:

12 hours, 20 Marks

- 1.1. Software crisis
- 1.2. Software characteristics
- 1.3. Software quality attributes
- 1.4. Software process model
- 1.5. Process iteration
- 1.6. process activities
- 1.7. Computer-aided software engineering
- 1.8. Functional and non –functional requirements
- 1.9. User requirements
- 1.10. System requirement
- 1.11. Interface specification
- 1.12. The software requirements documents
- 1.13. Feasibility study
- 1.14. Requirements elicitation and analysis
- 1.15. Requirements validation and management

What is Software? Computer software is the product that software professionals build and then support over the long term.





1. Application Software:-

- Application software is that software which is designed and developed to perform some particular application.
- It can be divided into following two types:-
 - a. **Generic (or Packaged) Software:-**
 - The application software which is designed to fulfill the needs of large group of users is known as generic or packaged software.
 - Example: MS-Word, Adobe Reader, MS-Excel.



b. Tailored (or Specific) Software:-

- The application software which is designed to fulfill the needs of a particular user/company/organization is known as tailored or specific software.

Ex: Software used in department stores, hospitals, schools etc.

2. System Software:-

- The software which can directly control the hardware of the computer are known as system software. Ex: Video driver, audio driver.



3. Utility Software:-

Small software that usually performs some useful tasks is known as utility software.

Ex: Win Zip, JPEG Compressor, PDF Merger, PDF to Word Converter etc.



1.2. Software Characteristics

- Software is developed or engineered, not manufactured in the classical sense
- Software doesn't “wear out”
- Software is custom-built, rather than being assembled from existing components



Continued...

1. Software is developed or engineered, not manufactured in the classical sense

- Although some similarities exists between software development and hardware manufacturing, the two activities are fundamentally different.
- **Similarities**
 - ✓ High quality needs to be achieved
 - ✓ Both depend on people &
 - ✓ Requires construction of product



Continued...

- Software is a design of strategies, instruction which finally perform the designed, instructed tasks. And a design can only be developed, not manufactured.
- Software is virtual. That is, software can be used using proper hardware. we can only use it, but we cannot touch and see hardware. Thus software never gets manufactured, they are developed.

2. Software doesn't “wear out”

(H/W failure)

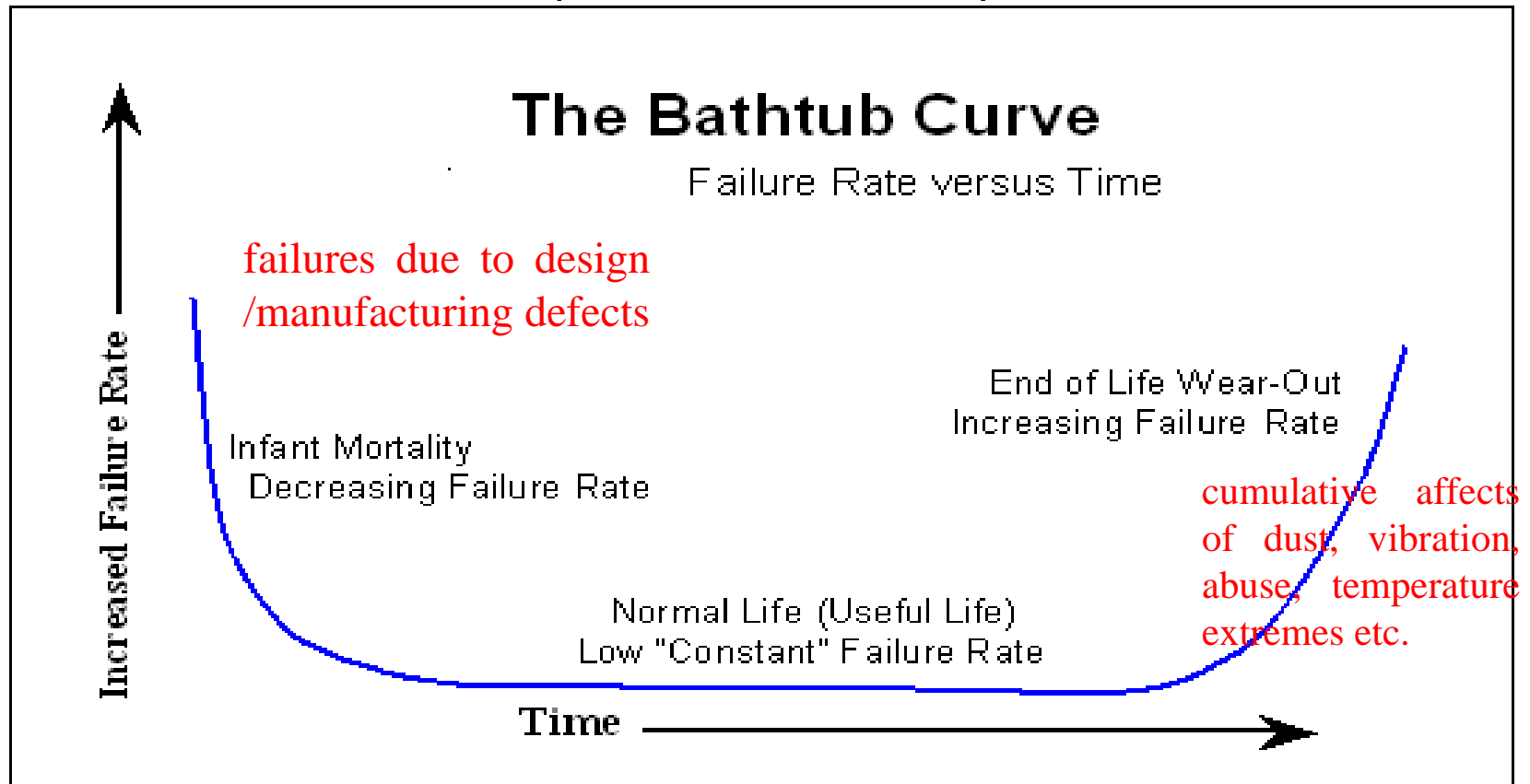


Figure 1 : Bathtub Curve



Figure 1 depicts failure rate vs time for hardware called bath tub curve

- The relationship, often called the “bathtub curve”
- It indicates that hardware exhibits relatively high failure rates early in its life (failures due to design /manufacturing defects);
- defects are corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time.

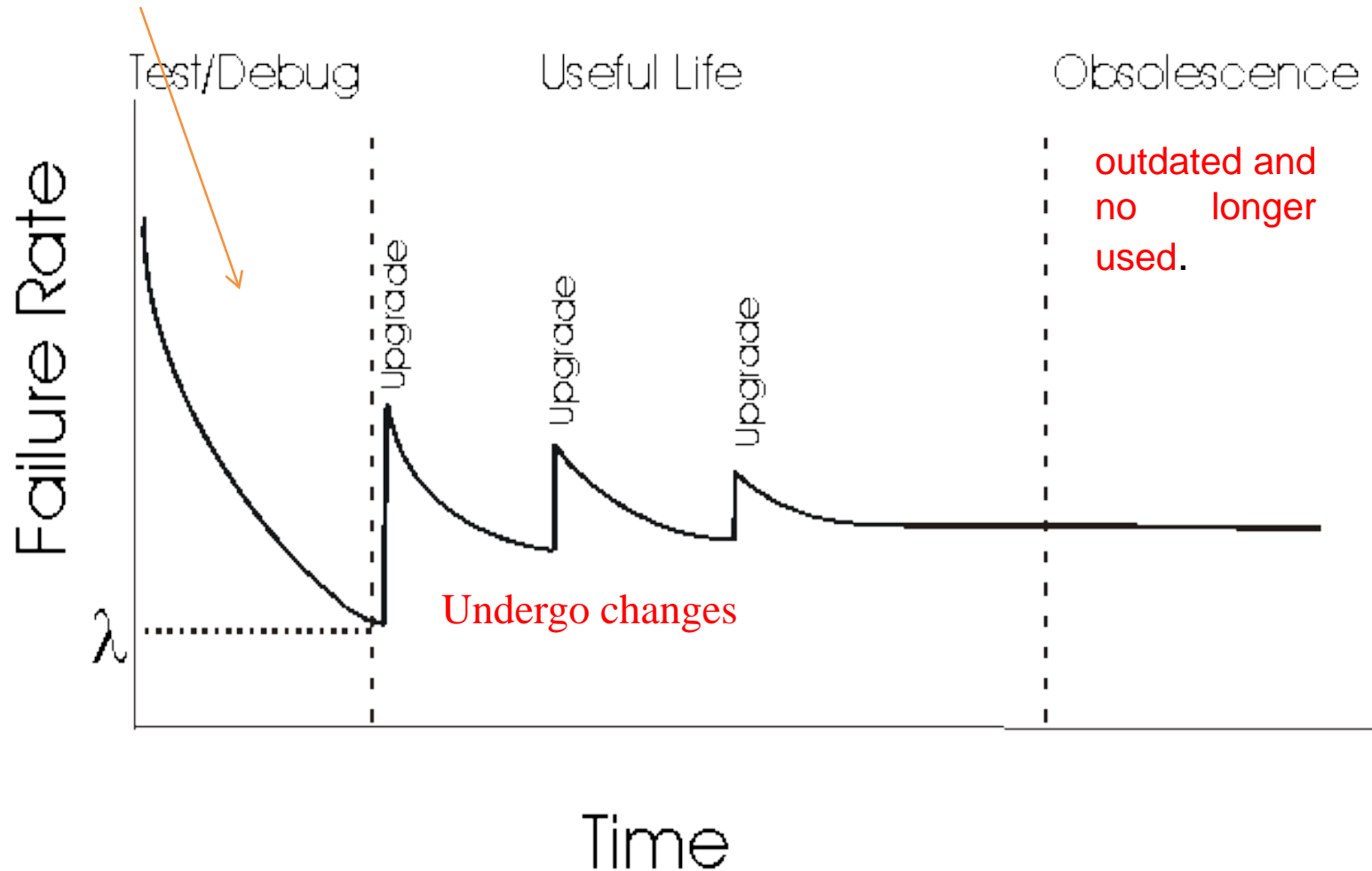


Continued...

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

high failure rates in the beginning due to Undiscovered defects.

S/W Failure





Continued...

- Software is not susceptible to environmental maladies
- In theory, s/w should take the form of “idealized curve”
- However, Undiscovered defects in the beginning will cause high failure rates
- These are corrected (ideally, without introducing other errors) and the curve flattens as shown
- During the life time it undergo changes



Continued...

- it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown
- Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again
- Slowly, the minimum failure rate level begins to rise- due to change.



Continued...

- Another aspect of wear illustrates the difference between hardware and software.
- When a hardware component wears out, it is replaced by a spare part.
- There are no software spare parts.
- Every software failure indicates an error in design or in the process through which design was translated into machine executable code.
- Therefore, software maintenance involves considerably more complexity than hardware maintenance.



3. Software is custom-built, rather than being assembled from existing components

- Consider the manner in which the control hardware for a computer-based product is designed and built.
- The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and



Continued...

- then goes to the shelf where catalogs of digital components exist.
- Each integrated circuit (called an *IC* or a *chip*) has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines.
- After each component is selected, it can be ordered off the shelf.



Continued...

- Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems.
- In the hardware world, component assemble and reuse is a natural part of the engineering process.
- In the software world, it is something that has only begun to be achieved on a broad scale.



1.3. Software quality attributes

■ **Functionality**

- ✓ All the features & their functionality should work as expected.
- ✓ There should not be any deviation in the actual result and expected result.

■ **Reliability**

- ✓ An s/w is said to be reliable if it delivers all features without any failure & that it is error free.



e.g.: an application of saving student records without any error and should not fail after entering 100 records.

- **Correctness:** A software product is correct, if different requirements as specified in the software requirements specification(SRS) document have been correctly implemented.
- **Usability**
 - ✓ An s/w product is said to be usable if it is easy to use without any specific training.
 - ✓ An s/w must be user friendly (i.e. easy to use).



- **Reusability**

- ✓ An s/w product has good reusability if different modules of the product can be reused to develop new product.

- **Efficiency**

- ✓ A product should not waste resource.

- **Portability**

- ✓ An s/w product is said to be portable if it can be easily made to work in different operating system.



■ **Maintainability**

An s/w product is said to be maintainable if

- ✓ Errors can be corrected easily.
- ✓ New functions can be added easily.
- ✓ Functionality can be modified easily

■ **Durable**

- ✓ An s/w product is said to be durable if it can be in use for long period of time.

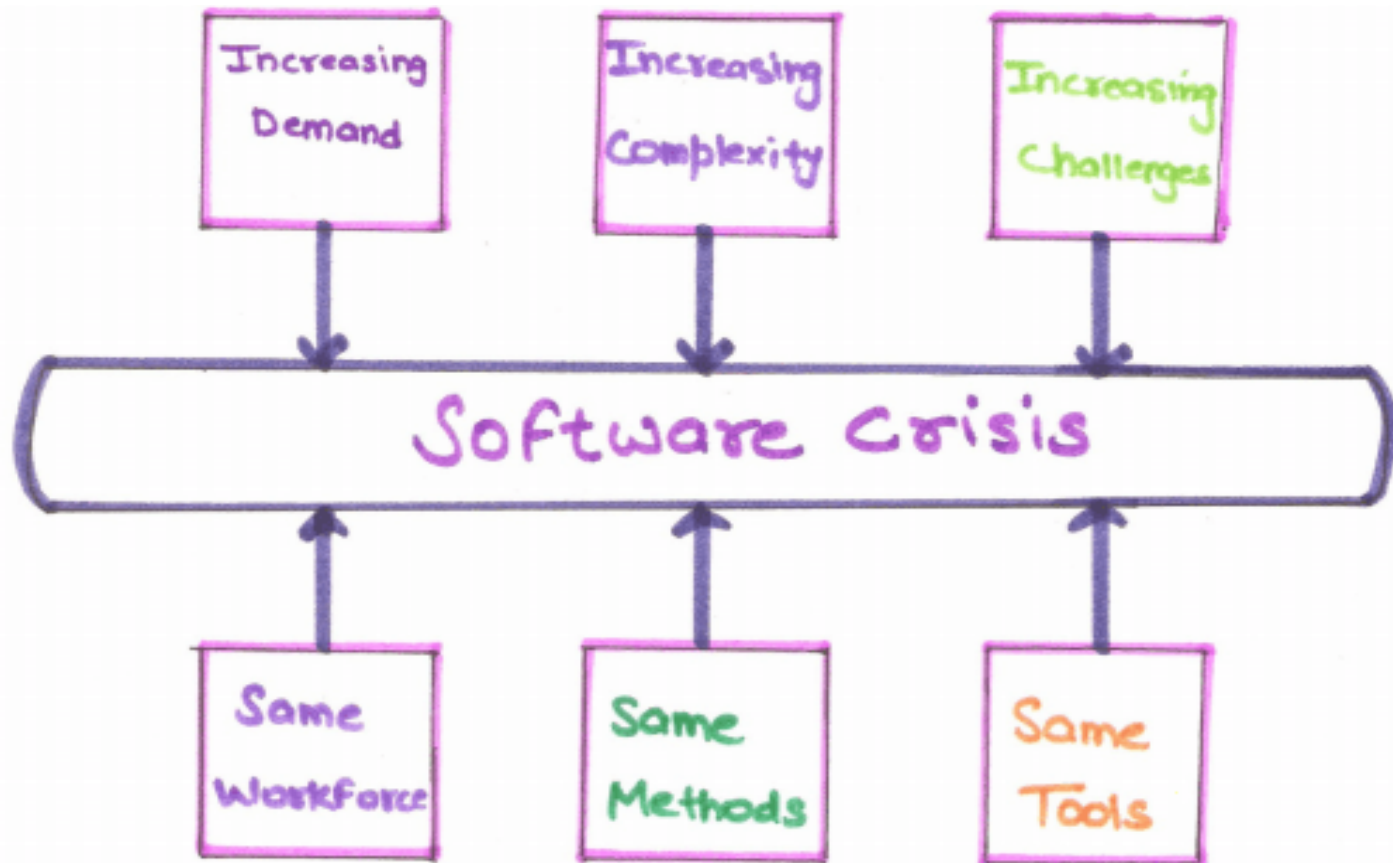


Software Crisis (assignment 1)

The difficulty of writing the code for a computer program which is correct and understandable is referred to as software crisis.

or

Software crisis is also referred to as inability to hire enough qualified programmers.





- Software market today has a turnover of more than millions of rupees.
- Out of this, approximately 30% of software is used for personal computers and the remaining software is developed for specific users or organizations.
- Application areas, such as the banking sector are completely dependent on software application for their working. Software failures in these technology-oriented areas have led to considerable loss in terms of time, money, and even human lives.



History has seen many such failures. Some of these are listed below:

1)1991 during Gulf War: The USA use patriot missiles as a defense against Iraqi scud missile. However, patriot failed to hit the scud many times which cost life of 28 USA soldiers. **In an inquiry it is found that a small bug had resulted in miscalculation of missile path.**



2) Arian- 5 Space Rocket: In 1996, developed at cost of **\$7000 Million Dollars** over a period of 10 years was destroyed within less than 1 minutes after its launch. As there was software bugs in rocket guidance system.

3) "Dollar 924 lakhs": In 1996, US bank credit accounts of nearly 800 customer with dollar 924 lakhs. This problem was due to main programming bug in the banking system.



- 4) The North East blackout in 2003-** has been major power system failures in the history of north which involves **100 power plants, 50 million customer faced problem, \$ 6 million dollar financial loss.**
- 5) In June 1980,** the North American Aerospace Defense Command (NORAD) reported that the US was under missile attack. The report was traced to **a faulty computer circuit** that generated incorrect signals.



Continue...

If the developers of the software responsible for processing these signals had taken into account the possibility that the circuit could fail, the false alert might not have occurred



1.3. Software Process Model

- Since the **prime objective** of **software engineering** is to develop methods for large systems, which produce **high quality** software at **low cost** and in **reasonable time**.
- So it is essential to perform software development **in phases**. This **phased development of software** is often referred to as software development life cycle (**SDLC**) or software life cycle.
- And the models used to achieve these goals are termed as **Software Process Models**.

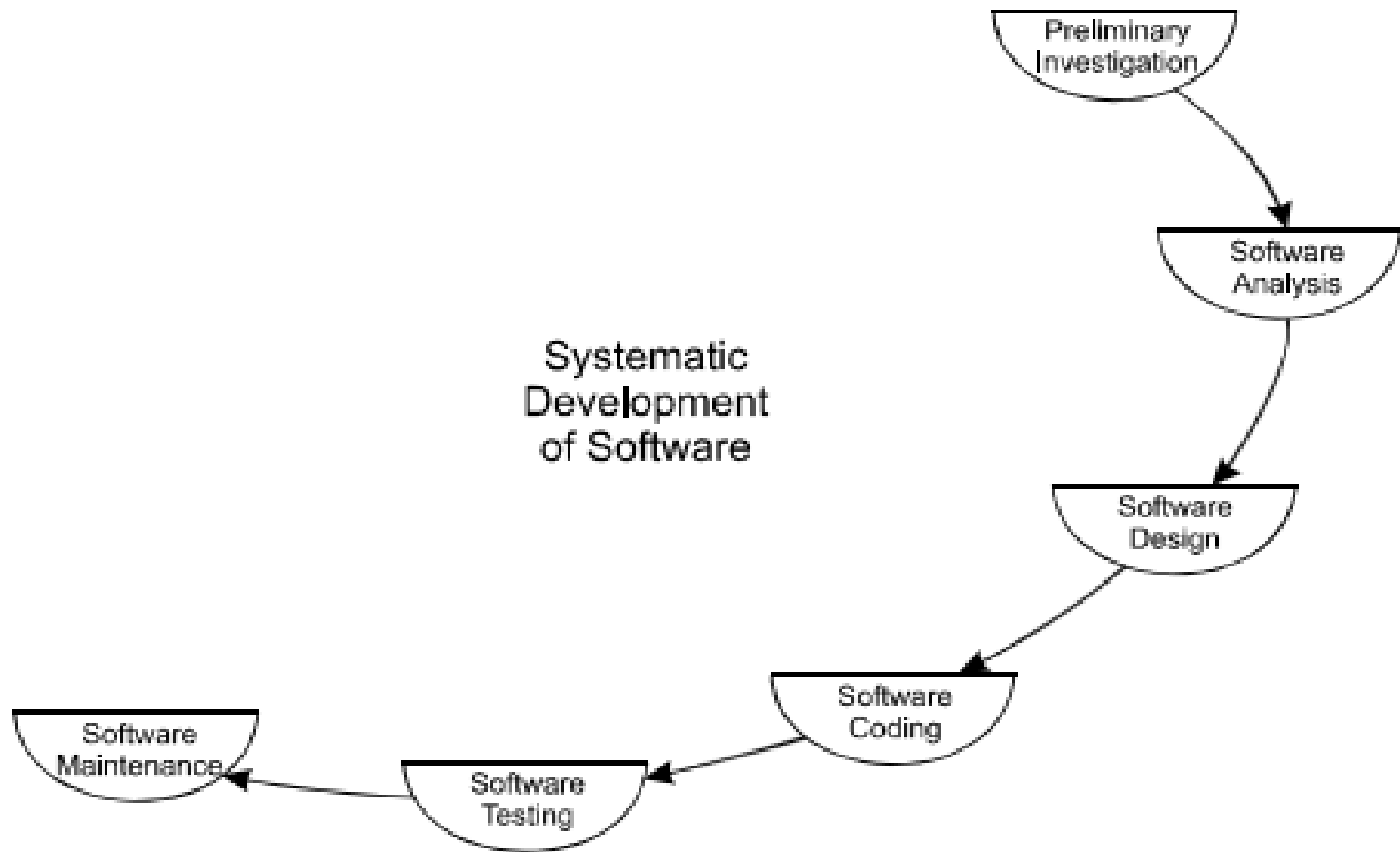


Fig 1 software development process



In fig 1,

- These phases **work in top to bottom** approach
- The phases take inputs from the previous phases, add features, and then produce outputs



1. Preliminary investigation/ feasibility study:

- Feasibility study decides whether the new system should be developed or not.
- There are three constraints, which decides the go or no-go decision.

a. Technical:

- determines whether technology needed for proposed system is available or not.
- determines whether the existing system can be upgraded to use new technology
- whether the organization has the expertise to use it or not.



b. Time:

- determines the time needed to complete a project.
- Time is an important issue as cost increases with an increase in the time period of a project.

c. Budget:

- This evaluation looks at the financial aspect of the project.
- determines whether the investment needed to implement the system will be recovered at later stages or not.



2. Software Analysis/Requirement Analysis:

- studies the problem or requirements of software in detail.
- After analyzing and elicitation of the requirements of the user, a requirement statement known as **software requirement specification (SRS)** is developed.



3. Software Design:

- **most crucial phase** in the development of a system. The SDLC process continues to move from the **what** questions of the analysis phase to the **how**.
- **logical design** is turned into a **physical design**.
- Based on the user requirements and the detailed analysis the system must be designed.



- Input, output, databases, forms, processing specifications etc. are drawn up in detail.
- Tools and techniques used for describing the system design are: Flowchart, ERD, Data flow diagram (DFD), UML diagrams like Use case, Activity, Sequence etc.



4. Software Coding:

- **Physical design into software code.**
- Writing a software code requires a prior knowledge of programming language and its tools. Therefore, it is important to choose the appropriate programming language according to the user requirements.
- A program code is efficient if it makes optimal use of resources and contains minimum errors.



5. Software Testing:

- **Software testing is performed to ensure that software produces the correct outputs i.e. free from errors.** This implies that outputs produced should be according to user requirements.
- Efficient testing improves the quality of software.
- Test plan is created to test software in a planned and systematic manner.



6. Software Maintenance:

- This phase comprises of a set of software engineering activities that occur after software is delivered to the user.
- After the software is developed and delivered, it may require changes. Sometimes, changes are made in software system when user requirements are not completely met.
- To make changes in software system, software maintenance process evaluates, controls, and implements changes.



Class Work

Q2.

Mention different phases of Software Development life cycle(SDLC), if you are under the project of **Library Management system**.

Thank You!!!

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering
Chapter One

Software Process and requirements

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.



Software Process Model (Continue...)



What is Software process?

- When you work to build a product or system, it's important to go through a series of predictable steps—a **road map** that helps you create a timely, high-quality result. The road map that you follow is called a “software process.”
- **Who does it?** **Software engineers** and their **managers** adapt the process to their needs and then follow it. In addition, the **people** who have requested the software have in the process of defining, building, and testing it.



Why is it important?

Because it provides path, stability, control over your project.

What are the steps?

At a detailed level, the process that you adopt depends on the software that you're building. One process might be appropriate for creating software for an aircraft avionics system, while an entirely different process would be indicated for the creation of a website.

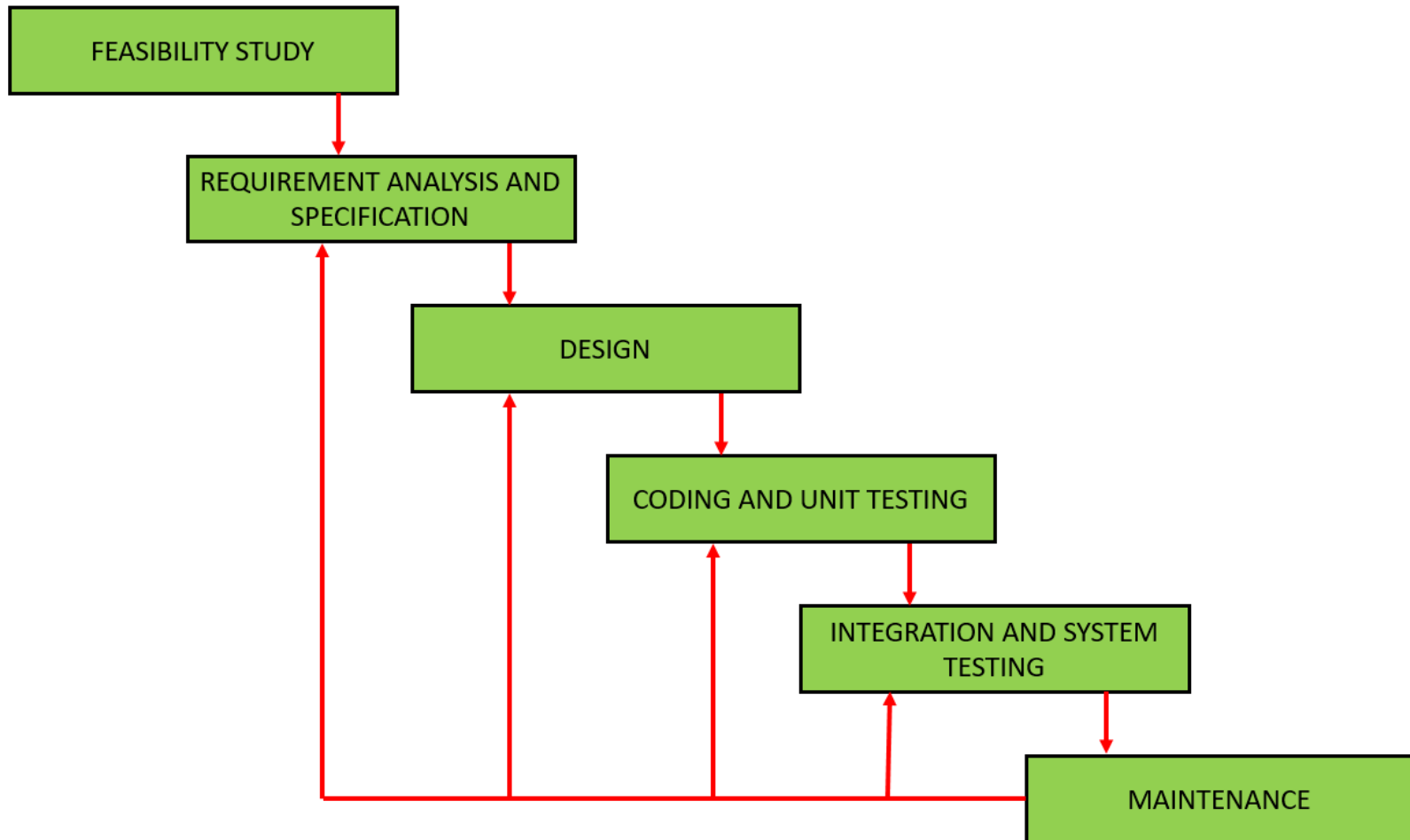


from a technical point of view:

A software process is a **framework** for the activities, actions, and tasks that are required to build high-quality software- *Roger S. Pressman*



1.3.1 Waterfall Model





a. Waterfall model

i. Feasibility study

- Financial
- Technical
- Time etc.

ii. Requirement specification: To specify the requirements' users specification should be clearly understood and the requirements should be analyzed. This phase involves **interaction between user and software engineer**, and produces a document known as software requirement specification (SRS).



a. Waterfall model

iii. Design: Determines the detailed process of developing software after the requirements are analyzed. It utilizes software requirements defined by the user and translates them into a software representation. In this phase, the emphasis is on finding a solution to the problems defined in the requirement analysis phase. The software engineer, in this phase is mainly concerned with the data structure, algorithmic detail, and interface representations.



a. Waterfall model

- iv. **Coding:** Emphasizes on translation of design into a programming language using the coding style and guidelines. The programs created should be easy to read and understand. All the programs written are documented according to the specification.
- v. **Testing:** Ensures that the product is developed according to the requirements of the user. Testing is performed to verify that the product is functioning efficiently with minimum errors. It focuses on the internal logics and external functions of the software



a. Waterfall model

vi. Implementation and maintenance: Delivers fully functioning operational software to the user. Once the software is accepted and deployed at the user's end, various changes occur due to changes in external environment (these include upgrading new operating system or addition of a new peripheral device). The changes also occur due to changing requirements of the user and the changes occurring in the field of technology. This phase focuses on modifying software, correcting errors, and improving the performance of the software.



a. Waterfall model

Input to the Phase	Process/Phase	Output of the Phase
Requirements defined through communication	Requirements analysis	Software requirements specification document
Software requirements specification document	Design	Design specification document
Design specification document	Coding	Executable software modules
Executable software modules	Testing	Integrated product
Integrated product	Implementation	Delivered software
Delivered software	Maintenance	Changed requirements



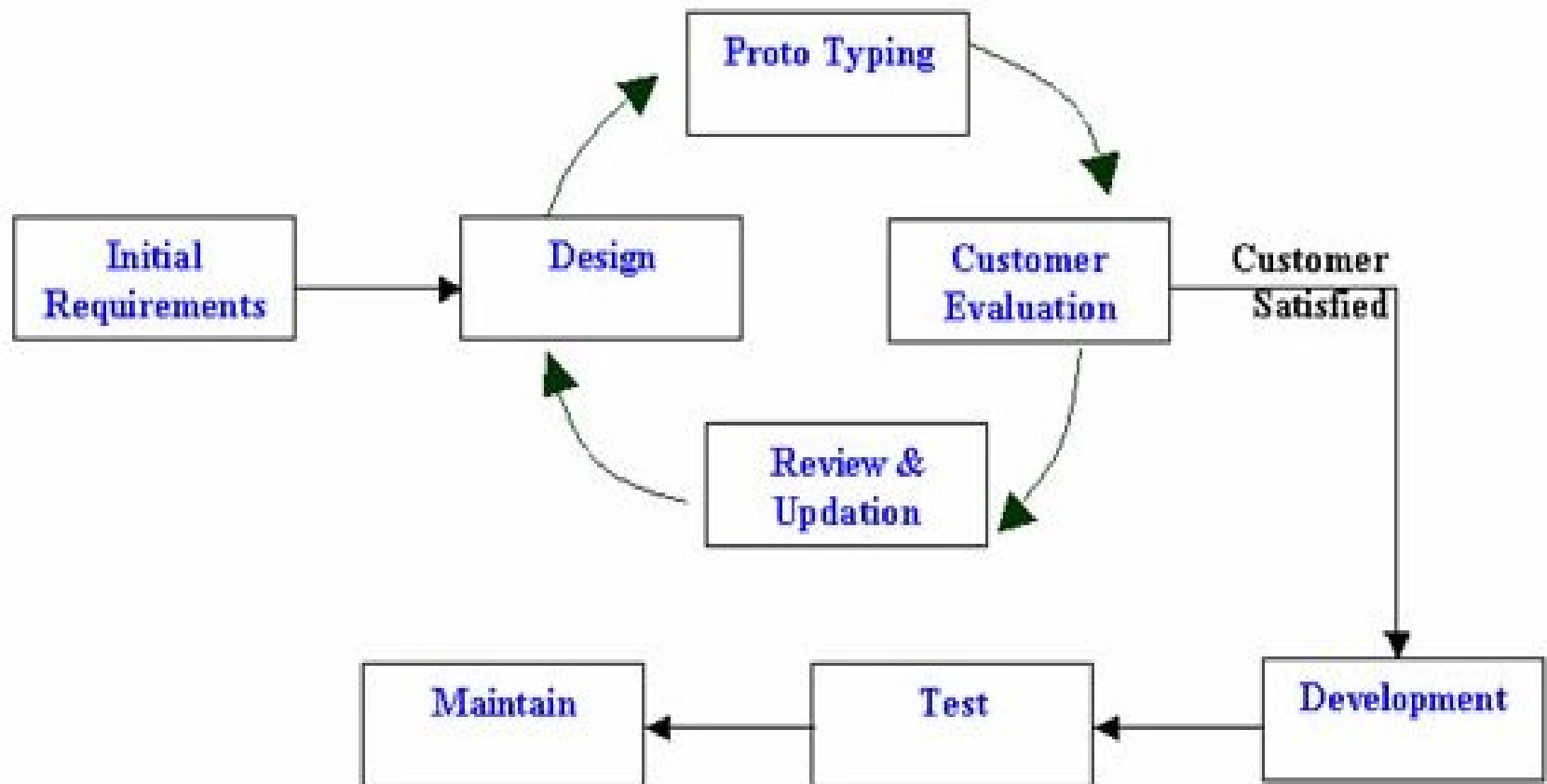
a. Waterfall model

Advantages	Disadvantages
<ul style="list-style-type: none">▪ Relatively simple to understand.▪ Each phase of development proceeds sequentially.▪ Allows managerial control where a schedule with deadlines is set for each stage of development.▪ Helps in controlling schedules, budgets, and documentation.	<ul style="list-style-type: none">▪ Requirements need to be specified before the development proceeds.▪ The changes of requirements in later phases of the waterfall model cannot be done. This implies that once an application is in the testing phase, it is difficult to incorporate changes at such a late phase.▪ No user involvement and working version of the software is available when the software is developed.▪ Does not involve risk management.▪ Assumes that requirements are stable and are frozen across the project span.



b. prototype model

- The prototyping model is applied when there is an absence of detailed information regarding input and output requirements in the software.
- Used if the requirements are not precisely specified.
- Prototyping model increases flexibility of the development process by allowing the user to interact and experiment with a working representation of the product known as **prototype**. A prototype gives the user an actual feel of the system.



Proto Type Model



i.Requirements gathering and analysis: Prototyping model begins with requirements analysis, and the requirements of the system are defined in detail. The **user is interviewed to know the requirements** of the system.

ii.Quick design: When requirements are known, a preliminary design or a quick design for the system is created. It is **not a detailed design**, however, it includes the important aspects of the system, which gives an idea of the system to the user. Quick design helps in developing the prototype.



iii. Build prototype: Information gathered from quick design is modified to form a prototype. The first prototype of the required system is developed from quick design. It represents a 'rough' design of the required system.

iv. User evaluation: Next, the proposed system is presented to the user for consideration as a part of development process. The users thoroughly evaluate the prototype and recognize its strengths and weaknesses, such as what is to be added or removed. Comments and suggestions are collected from the users and are provided to the developer.



v. Prototype refinement: Once the user evaluates the prototype, it is refined according to the requirements. The developer revises the prototype to make it more effective and efficient according to the user requirement. **If the user is not satisfied with the developed prototype,** a new prototype is developed with the additional information provided by the user. **The new prototype is evaluated in the same manner,** as the previous prototype, process continues until all the requirements specified by the user are met. **Once the user is satisfied a final system is developed.**



vi. Engineer product: Once the requirements are completely known, user accepts the final prototype. The final system is thoroughly evaluated and tested followed by routine maintenance on continuing basis to prevent large-scale failures and to minimize downtime.

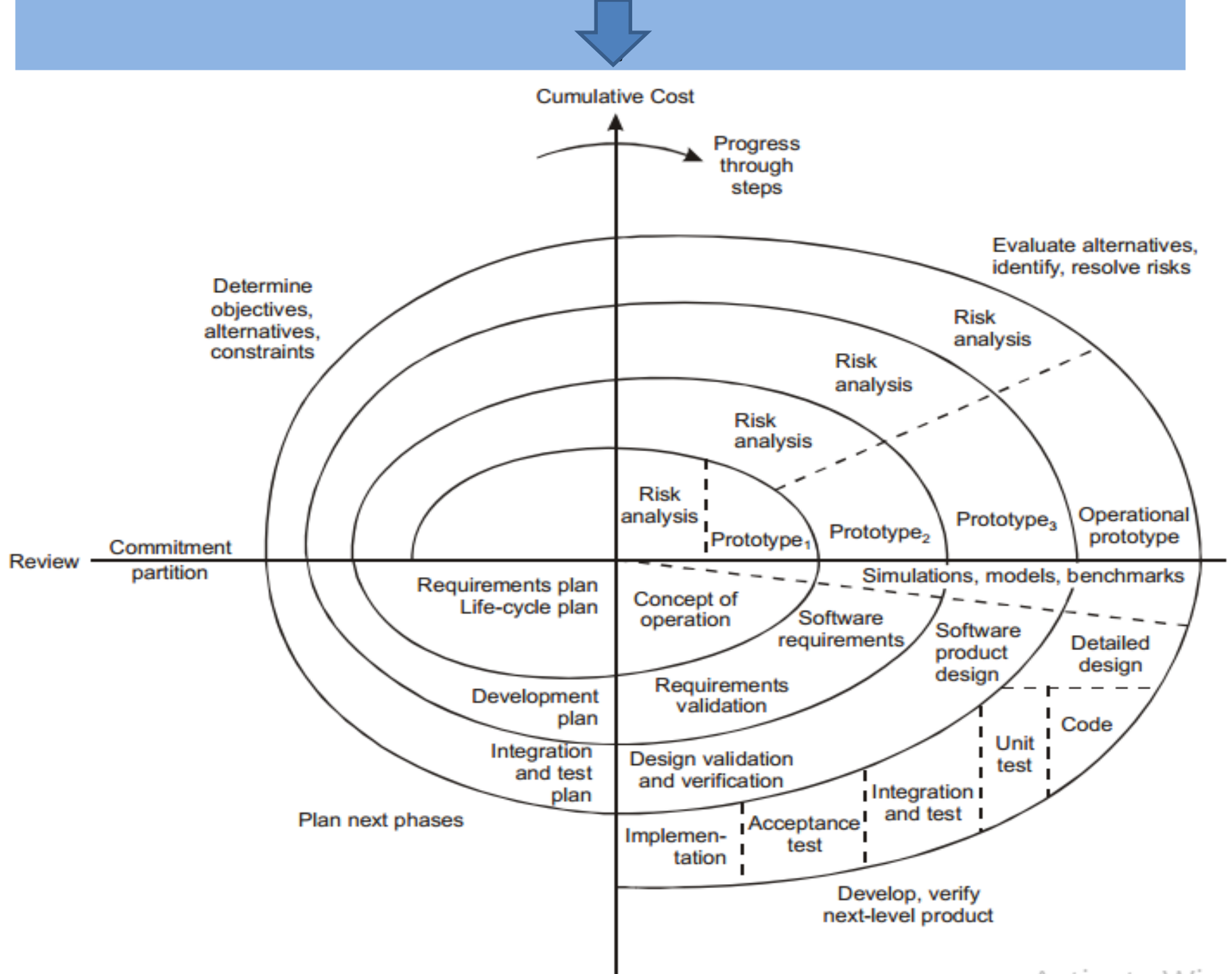


Advantages	Disadvantages
<ul style="list-style-type: none">▪ Provides a working model to the user early in the process, enabling early assessment and increasing user confidence.▪ Developer gains experience and insight by developing a prototype, thereby resulting in better implementation of requirements.▪ Prototyping model serves to clarify requirements, which are not clear, hence reducing ambiguity and improving communication between developer and user.▪ There is a great involvement of users in software development. Hence, the requirements of the users are met to the greatest extent.▪ Helps in reducing risks associated with the project.	<ul style="list-style-type: none">▪ If the user is not satisfied by the developed prototype, then a new prototype is developed. This process goes on until a perfect prototype is developed. Thus, this model is time consuming and expensive.▪ Developer loses focus of the real purpose of prototype and compromise with the quality of the product. For example, they apply some of the inefficient algorithms or inappropriate programming languages used in developing the prototype.▪ Prototyping can lead to false expectations. It often creates a situation where user believes that the development of the system is finished when it is not.▪ The primary goal of prototyping is rapid development, thus, the design of system can suffer as it is built in a series of layers without considering integration of all the other components.



c. Spiral Model

In 1980's Boehm introduced a process model known as spiral model. The spiral model comprises of activities organized in a spiral, which has many cycles. This model combines the features of prototyping model and waterfall model and is advantageous for large, complex and expensive projects which involves high risk.





1. Each cycle of the **first quadrant** commences with identifying the goals for that cycle. In addition, it **determines other alternatives, which are possible in accomplishing those goals.**
2. Next step in the cycle evaluates alternatives based on objectives and constraints. This process **identifies the project risks.** Risk signifies that there is a possibility that the objectives of the project cannot be accomplished. If so the formulation of a cost effective **strategy for resolving risks is followed. the strategy, which includes prototyping, simulation, benchmarking.**



3. The development of the software depends on remaining risks. The third quadrant **develops the final software while considering the risks that can occur**. Risk management considers the time and effort to be devoted to each project activity, such as planning, configuration management, quality assurance, verification, and testing.
4. The last quadrant plans the next step, and includes planning for the next prototype and thus, comprises of requirements plan, development plan, integration plan, and test plan

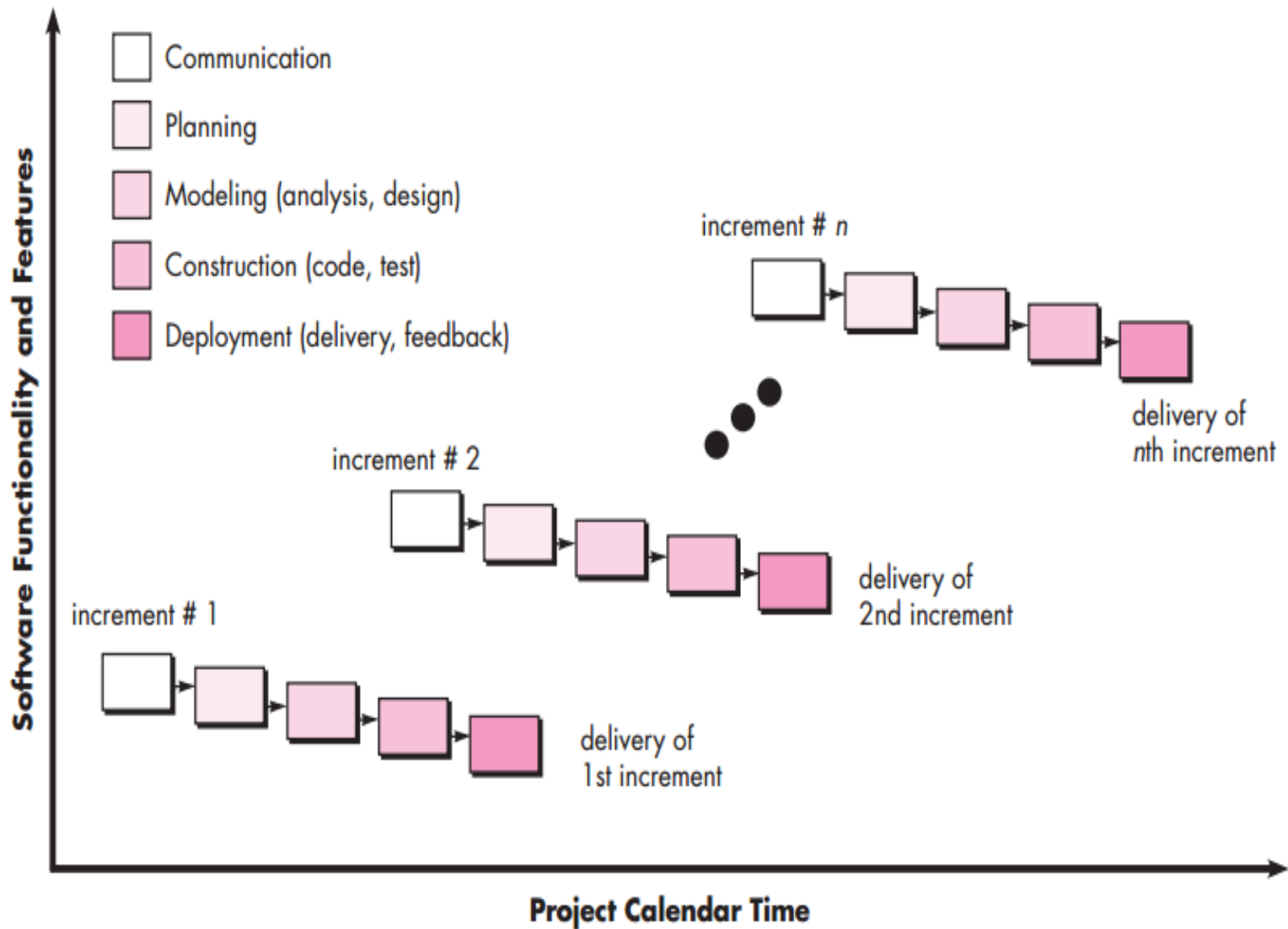


Advantages	Disadvantages
<ul style="list-style-type: none">▪ Avoids the problems resulting in risk-driven approach in the software.▪ Specifies a mechanism for software quality assurance activities.▪ Spiral model is utilised by complex and dynamic projects.▪ Re-evaluation after each step allows changes in user perspectives, technology advances or financial perspectives.▪ Estimation of budget and schedule gets realistic as the work progresses.	<ul style="list-style-type: none">▪ Assessment of project risks and its resolution is not an easy task.▪ Difficult to estimate budget and schedule in the beginning, as some of the analysis is not done until the design of the software is developed.



d. Evolutionary model

- An Evolutionary model **breaks up an overall solution into increments** of functionality and develops each increment individually.
- The evolution model **divides** the development cycle into **smaller, "Incremental Waterfall Model"** in which users are able to get access to the product at the end of each cycle.
- The users provide feedback on the product for planning stage of the next cycle and the development team responds, often by changing the product, plans or process.





Evolutionary Model

- **Advantages**

- The user of an evolutionary model gets a chance to experiment with a partially developed system much before the actual fully developed version is released .
- Thus this model facilitates to elicit the exact requirements of the user for incorporating into the fully developed system.
- Also the core modules get tested thoroughly ,thereby reducing chances of errors in the final product

- **Disadvantages**

- In most practical problems ,it is difficult to subdivide the problem into several functional units that can be incrementally implemented and delivered.
- Therefore ,this model is useful only for large problems, where it is easier to identify modules for incremental implementation.

RAD model

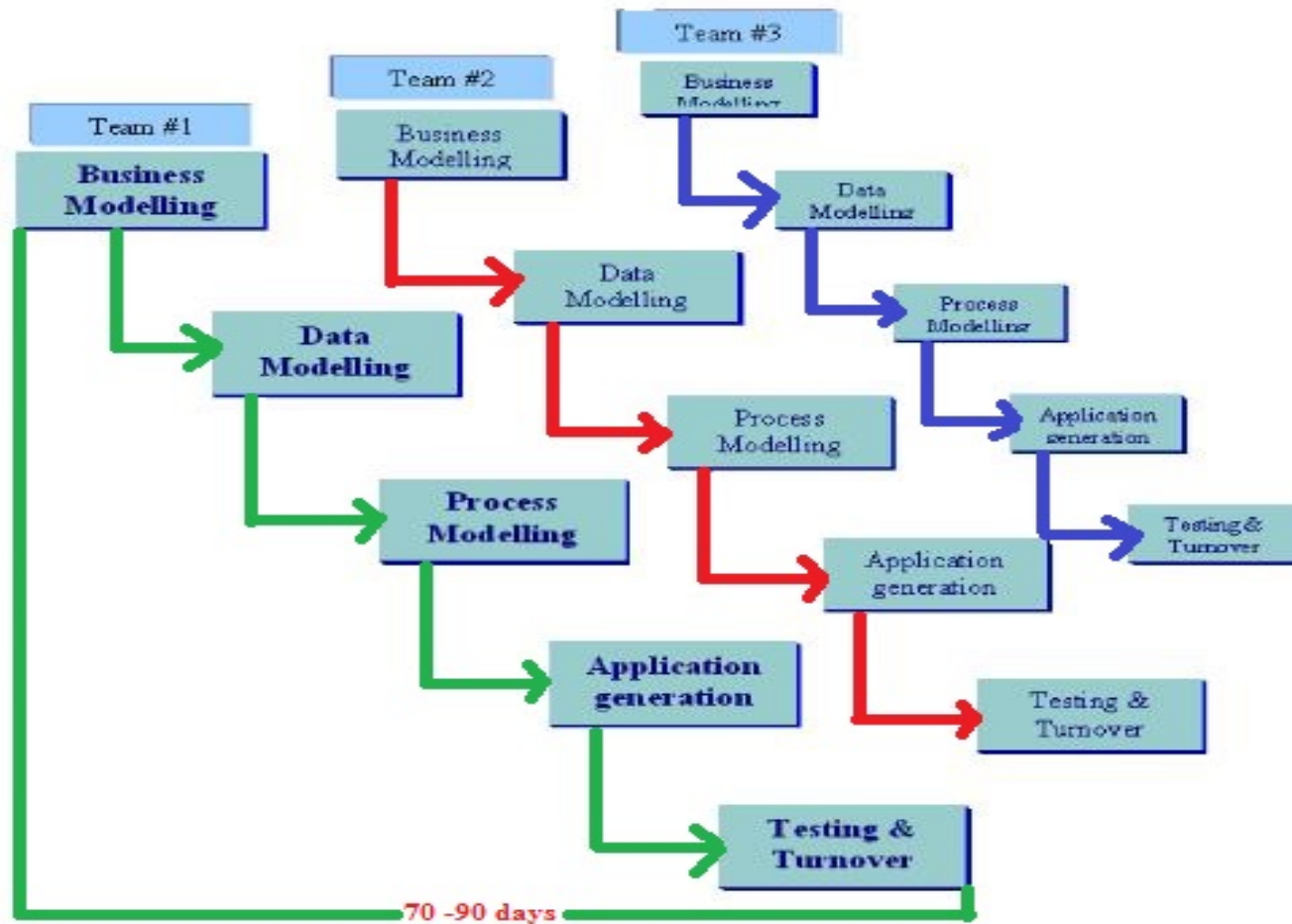
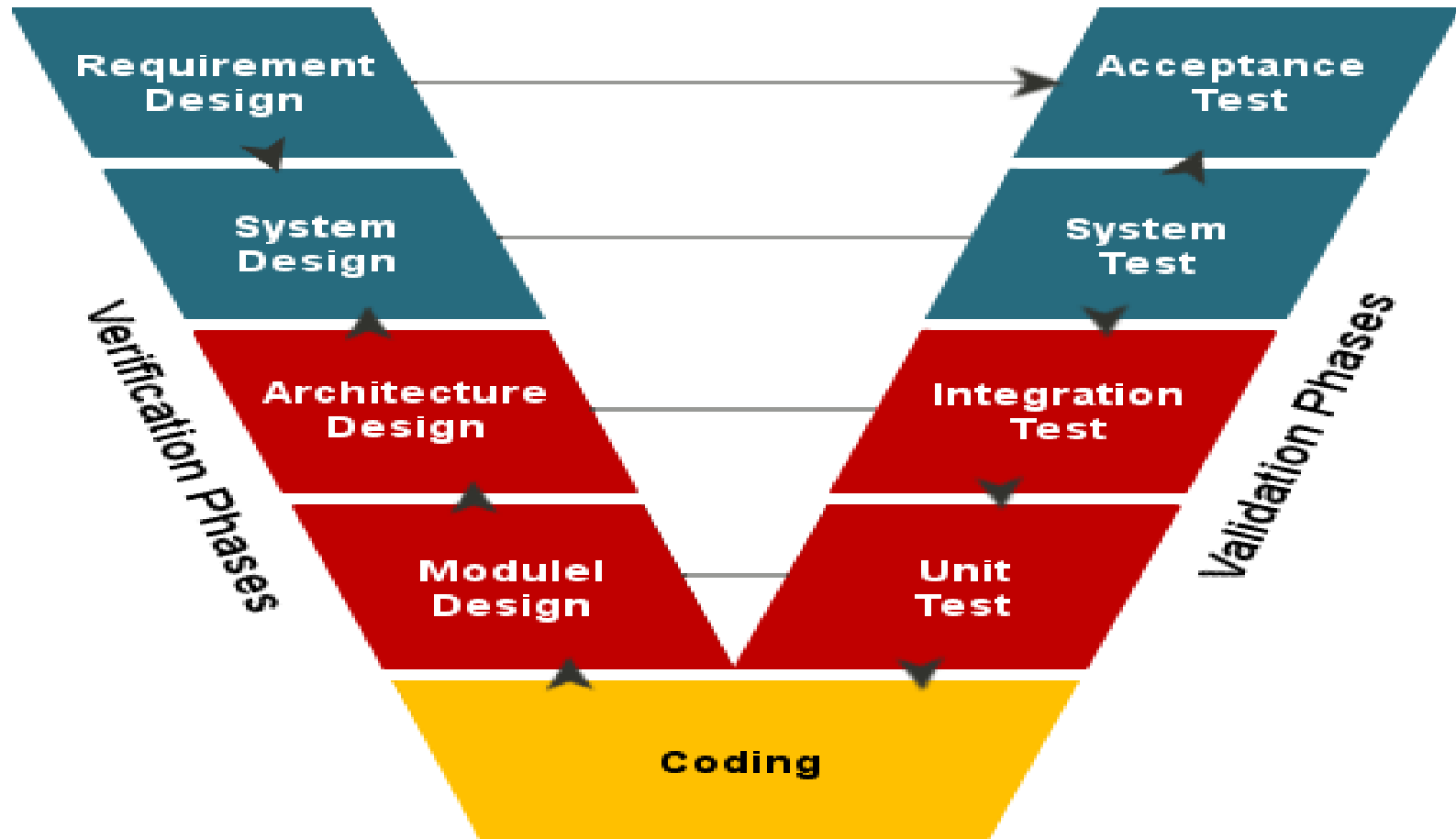


Fig:- RAD (Rapid Application Development) Model

V model



Thank You!!!

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering
Chapter One

Software Process and requirements

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

Chapter One: **Software Process and requirements**

Course Outline:

12 hours, 20 Marks

- 1.1. Software crisis
- 1.2. Software characteristics
- 1.3. Software quality attributes
- 1.4. Software process model
- 1.5. Process iteration
- 1.6. process activities
- 1.7. Computer-aided software engineering
- 1.8. Functional and non –functional requirements
- 1.9. User requirements
- 1.10. System requirement
- 1.11. Interface specification
- 1.12. The software requirements documents
- 1.13. Feasibility study
- 1.14. Requirements elicitation and analysis
- 1.15. Requirements validation and management



What is requirement?

- Requirements describe how a system should act, appear, or perform.
- For this, when users request for software, they possess an approximation of **what the new system should be capable of doing**.
- Requirements differ from one user to another user and from one business process to another business process.



What is software requirement?

IEEE defines requirement as

“(1) A condition or **capability needed by a user to solve a problem** or achieve an objective. (2) A condition or **capability that must be met or possessed by a system** or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A **documented representation of a condition or capability** as in (1) or (2)”

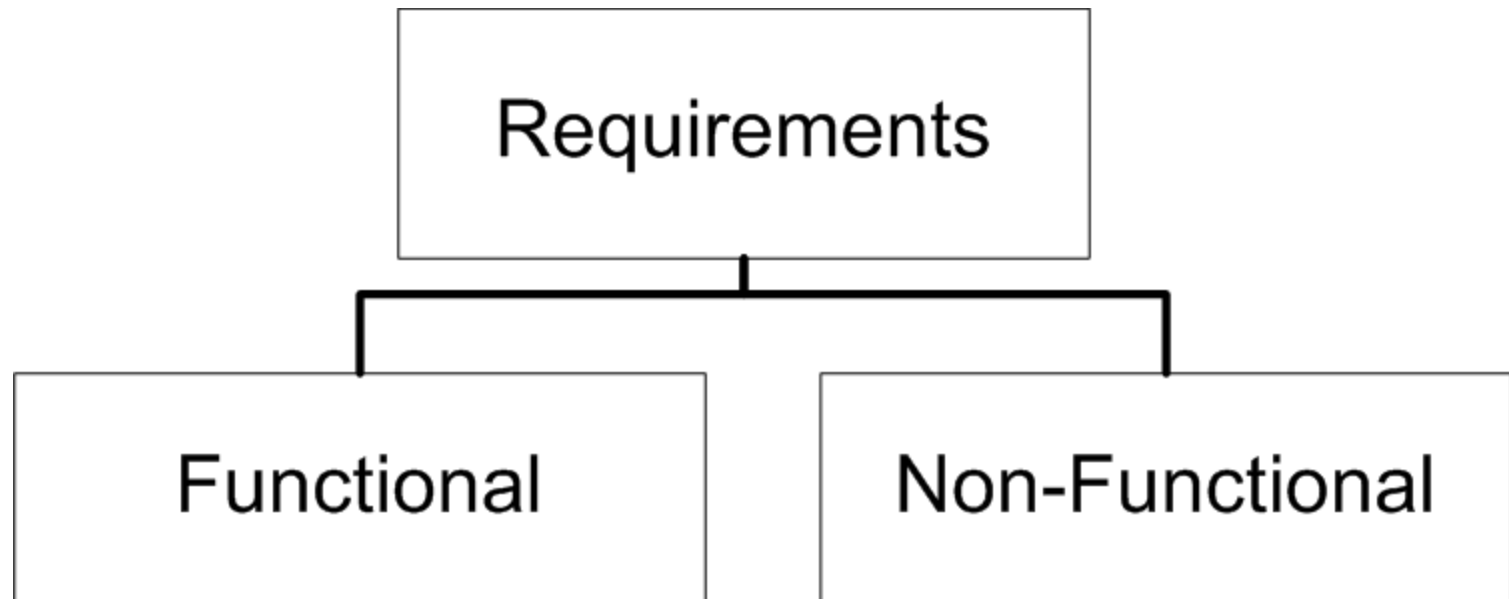


Guidelines for Expressing Requirements

- Sentences and paragraphs should be short and written in active voice. Also, proper grammar, spelling, and punctuation should be used.
- **Conjunctions**, such as ‘and’ and ‘or’ **should be avoided** as they indicate the combination of several requirements in one requirement.
- Each requirement should be stated only once so that it does not create redundancy in the requirements specification document



Types of Requirements





- A functional requirement describes *what* a software system should do, while **non-functional** requirements place constraints on *how* the system **will do so**.
- Functional requirements **specifies a function that a system** or system component must be able to perform. Whereas, non-functional requirements (also known as **quality requirements**) **relate to system attributes, such as reliability, response time etc.**



Functional requirement

- A banking system must send perform requested transaction, whenever a certain condition is met (i.e. account no, password, etc).

Non-functional requirement

- Those transaction should be completed with a latency of no greater than 6 hours from such an activity.
- **Note:** *Example of functional and non-functional is on the “Case study example”.*



Requirements Engineering Process

- The requirements engineering (RE) process is a series of activities that are performed in requirements phase in order to express requirements in **software requirements specification (SRS) document**.
- These **steps include** feasibility study, requirements elicitation, requirements analysis, requirements specification, requirements validation, and requirement management

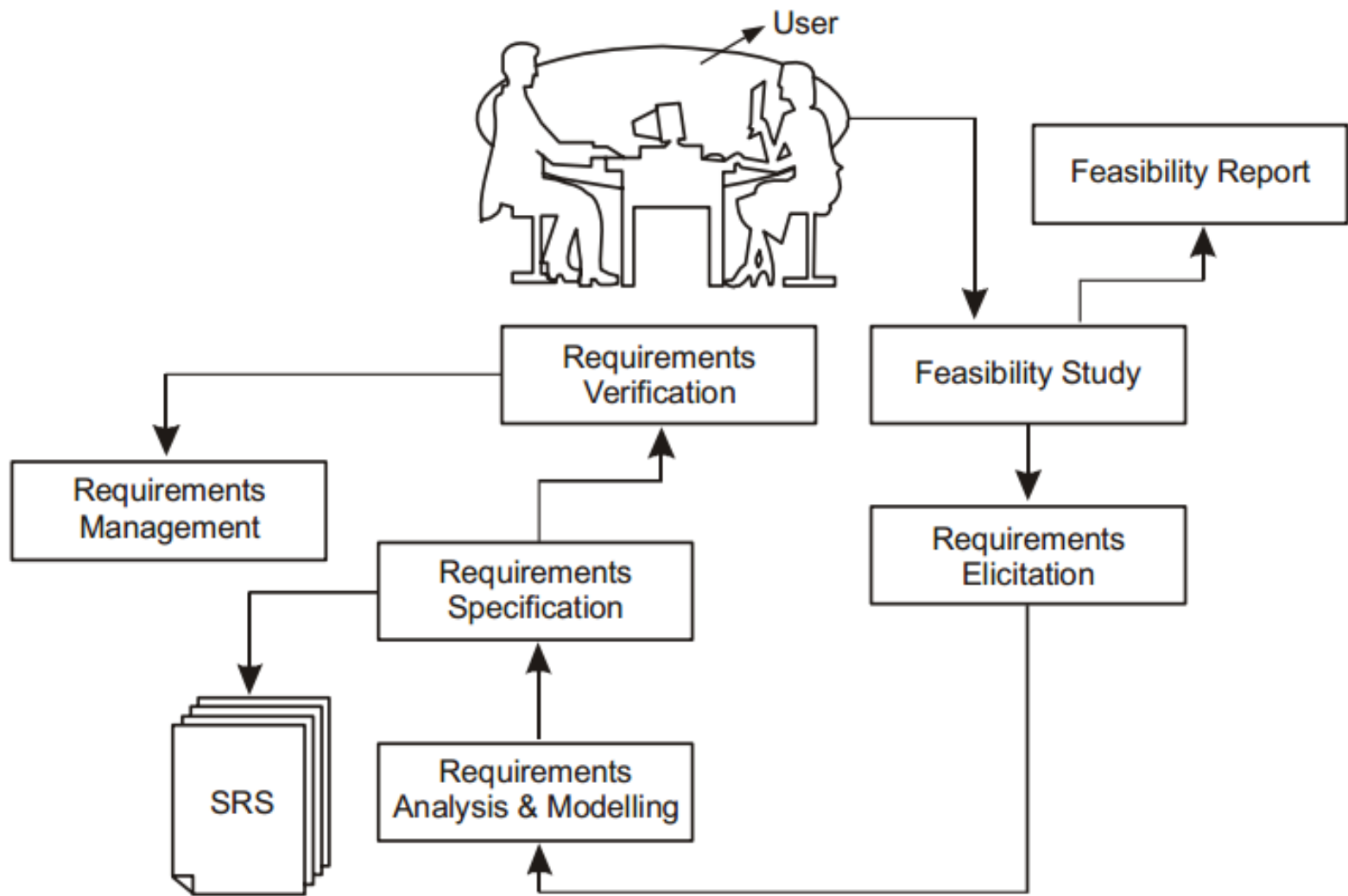


Fig: Requirement engineering process



STEP 1: FEASIBILITY STUDY

Objectives of feasibility study:

- To determine whether the software can be implemented using current technology and within the specified budget and schedule or not.
- To determine whether the software can be integrated with other existing software or not.
- To minimizes project failure.



Types of feasibility study:

Technical

- ✓ technical skills and capabilities of development team.
- ✓ Assure that the technology chosen, has large number of users so that they can be consulted when problems arise.

Operational

- ✓ solution suggested by software development team is acceptable or not.
- ✓ whether users will adapt to new software or not.



Types of feasibility study:

Economic feasibility/ Budget

- ✓ whether the required software is capable of generating financial gains for an organization or not.
- ✓ cost incurred on software development team
- ✓ estimated cost of hardware and software.
- ✓ cost of performing feasibility study.

Time

- ✓ Whether the project will be completed on pre-specified time or not.



Feasibility Study Process

1. Information assessment:

- verifies that the system can be implemented using new technology and within the budget.

2. Information collection:

- **Specifies the sources** from where information about software can be obtained.
- **Sources:**
 - ✓ users (who will operate the software)
 - ✓ organization (where the software will be used).
 - ✓ software development team (who understands user requirements and knows how to fulfill them in software).

3. Report writing:

- Information about changes in software scope, budget, schedule, and suggestion of any requirements in the system.



STEP2:REQUIREMENTS ELICITATION

- **Process of collecting information** about software requirements from different stakeholders (users, developer, project manager etc.)
- Various **issues**:

1. Problems of understanding:

- Users are not certain about their requirements and thus are unable to express what they require in software and which requirements are feasible.
- This problem also arises when users have no or little knowledge of the problem domain and are unable to understand the limitations of computing environment of software.



2. Problems of volatility:

- This problem arises when **requirements change over time.**

Elicitation Techniques

The commonly followed elicitation techniques are listed below:

1. Interviews:

- Ways for eliciting requirements, it helps software engineer, users, & development team to understand the problem and suggest solution for the problem.
- **An effective interview should have characteristics listed below:**
 - ✓ Individuals involved in interviews should be able to **accept new ideas**, focus on **listening to the views of stakeholders & avoid biased views.** □
 - ✓ Interviews **should be conducted in defined context to requirements rather than in general terms.** E.g. a set of a questions or **a *requirements proposal*.**



2.Scenarios:

- Helps to determine possible future outcome before implementation.
- In Generally, a scenario comprises of:
 - ✓ Description of **what** users expect when scenario starts.
 - ✓ Description of **how** to handle the situation when software is not operating correctly.
 - ✓ Description of the state of software **when** scenario ends.

3.Prototypes:

- helps to clarify **unclear requirements**.
- helps users to **understand the information they need to provide** to software development team.

4.Quality function deployment (QFD):

-Assignment 3



STEP3: REQUIREMENT ANALYSIS

- *It is the process of studying and refining requirements*

Tasks performed in requirements analysis are:

- Understand the problem for which software is to be developed.
- Develop analysis model to analyze the requirements in the software.
- Detect and resolve conflicts that arise due to unclear and unstated requirements.
- Determine operational characteristics of software and how it interacts with its environment.



STEP4: REQUIREMENTS SPECIFICATION

- **Development of SRS document** (software requirement specification document).

Characteristics of SRS

1. Correct:

SRS is correct when

- all user requirements are stated in the requirements document.
- The stated requirements should be according to the desired system.

2. Unambiguous:

- **SRS is unambiguous when every stated requirement has only one interpretation** i.e. each requirement is uniquely interpreted.



3. Complete:

- SRS is **complete** when the requirements clearly define what the software is required to do.

4. Modifiable:

- The requirements of the user can change, hence, requirements document should be created in such a manner where those **changes can be modified easily.**

5. Ranked for importance and stability:

- All requirements are not equally important.



6. Verifiable:


- SRS is verifiable when the specified requirements **can be verified with a cost-effective process** to check whether the final software meets those requirements or not.

7. Consistent:

- SRS is consistent **when the individual requirements defined does not conflict with each other.**
- **e.g.**, a requirement states that an event ‘a’ is to occur before another event ‘b’. But then another set of requirements states that event ‘b’ should occur before event ‘a’.

8. Traceable:

- SRS is traceable **when the source of each requirement is clear and it facilitates the reference of each requirement in future.**



1.0	Introduction
1.1	Purposes
1.2	Scope
1.3	Definitions, Acronyms, and Abbreviations
1.4	References
1.5	Overview
2.0	The Overall Description
2.1	Product Perspective
2.1.1	System Interface
2.1.2	Interface
2.1.3	Hardware Interface
2.1.4	Software Interface
2.1.5	Communications Interface
2.1.6	Memory Constraints
2.1.7	Operations
2.1.8	Site Adaptation Requirements
2.2	Product Functions
2.3	User Characteristics
2.4	Constraints
2.5	Assumptions and Dependency
2.6	Apportioning of Requirements
3.0	Specific Requirements
3.1	External Interface
3.2	Functions
3.3	Performance Requirements
3.4	Logical Database of Requirement
3.5	Design Constraints
3.5.1	Standards Compliance
3.6	Software System Attributes
3.6.1	Reliability
3.6.2	Availability
3.6.3	Security
3.6.4	Maintainability
3.6.5	Portability
3.7	Organizing the Specific Requirements
3.7.1	System Mode
3.7.2	User Class
3.7.3	Objects
3.7.4	Feature
3.7.5	Stimulus
3.7.6	Response
3.7.7	Functional Hierarchy
3.8	Additional Comments
4.0	Change Management Process
5.0	Document Approvals
6.0	Supporting Information

Fig : SRS Document template



STEP 5 : REQUIREMENTS VALIDATION

WHY VALIDATION ?

- Errors present in the SRS will adversely affect the cost if they are detected later in the development process or when the software is delivered to the user.

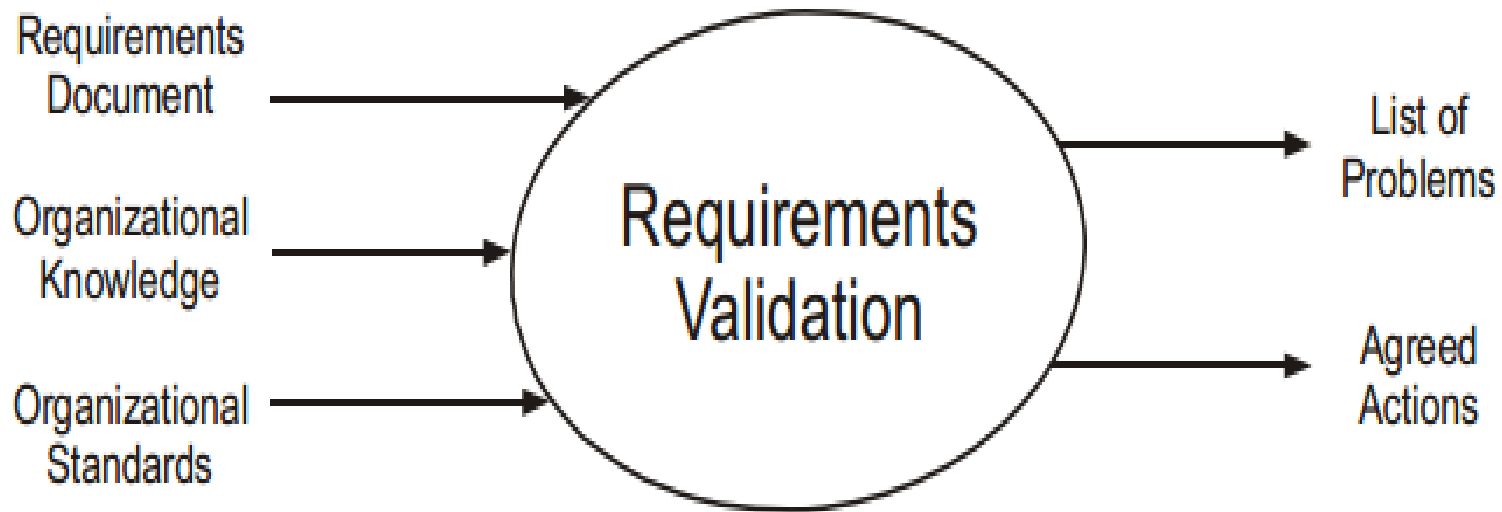


Fig: Requirement Validation



STEP 6: REQUIREMENTS MANAGEMENT

WHY ??

- To understand and control changes to system requirements.

Advantages of requirements management:

Better control of complex projects:

- Provides overview to development team with a **clear understanding of what, when and why** software is to be delivered.

Improves software quality:

- **Ensures that the software performs according to requirements to enhance software quality.**



Reduced project costs and delays:

- **Minimizes errors early in the development cycle**, as it is expensive to 'fix' errors at the later stages of the development cycle. As a result, the project costs also reduced.

Improved team communications:

- **Facilitates early involvement of users** to ensure that their needs are achieved.



Requirements Management Process

- Requirements management starts with **planning**,
- Then, **each requirement is assigned a unique 'identifier'** so that it can be crosschecked by other requirements. Once requirements are identified, requirements **tracing** is performed.
- The **objective** of requirement **tracing** is to ensure that all the requirements are well understood and are included in test plans and test cases.
- Traceability information is stored in a **traceability matrix**, which relates requirements to stakeholders or design module. **Traceability matrix** refers to a **table that correlates requirements**.



Req. ID	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		U	R					
1.2			U			R		U
1.3	R			R				
2.1			R		U			U
2.2								U
2.3		R		U				
3.1								R
3.2							R	

U->dependency

R-> weaker Relationship

Requirements change management

- It is used when there is a request or proposal for a change to the requirements.



Fig: Required change management

Thank You!!!

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering

Chapter 2

System Model

by

Santosh Giri

Lecturer, IOE, Pulchowk Campus.

Chapter Two: System models

Course Outline: 3 hours, 5-7 Marks

2. System models (3 hours)

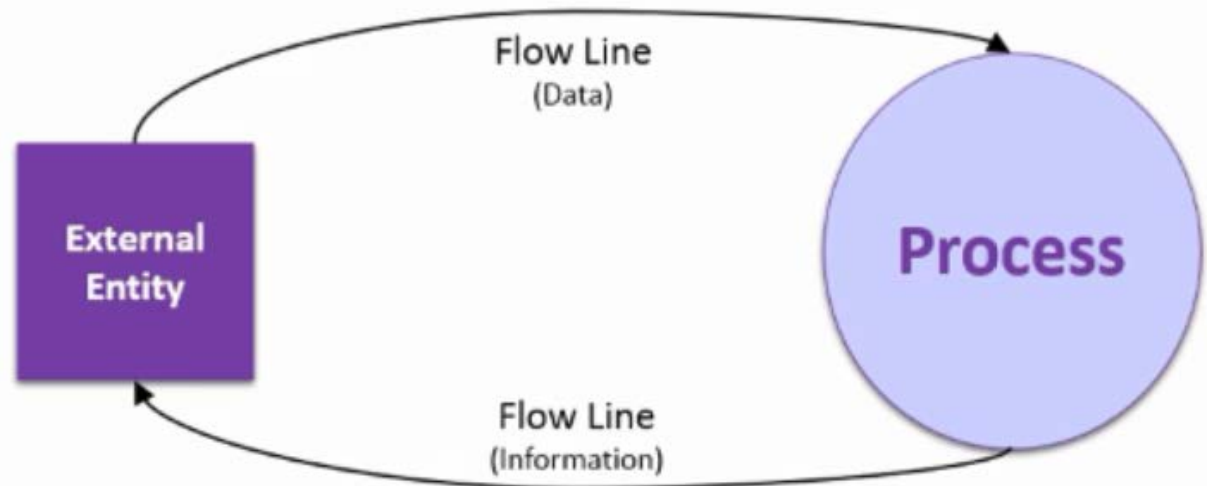
2.1. Context models

2.2. Behavioral models

2.3. Data and object models

2.1. Context Models

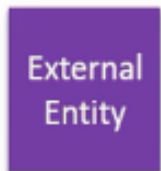
- Contents model show what lies outside the system boundaries.
- There exist only one circle or process that represents the whole system.
- **Purpose:** to show expected inputs and outputs to and from the system.





Context Models

Shapes used in Context Diagrams



External Entity

An element that inputs data into an information system and / or retrieves data from the information system.



Process

When an action takes place on data, turning it into Information. In the case of a Context Diagram there is **only 1 Process** that represents the **entire System**.



Flow Line

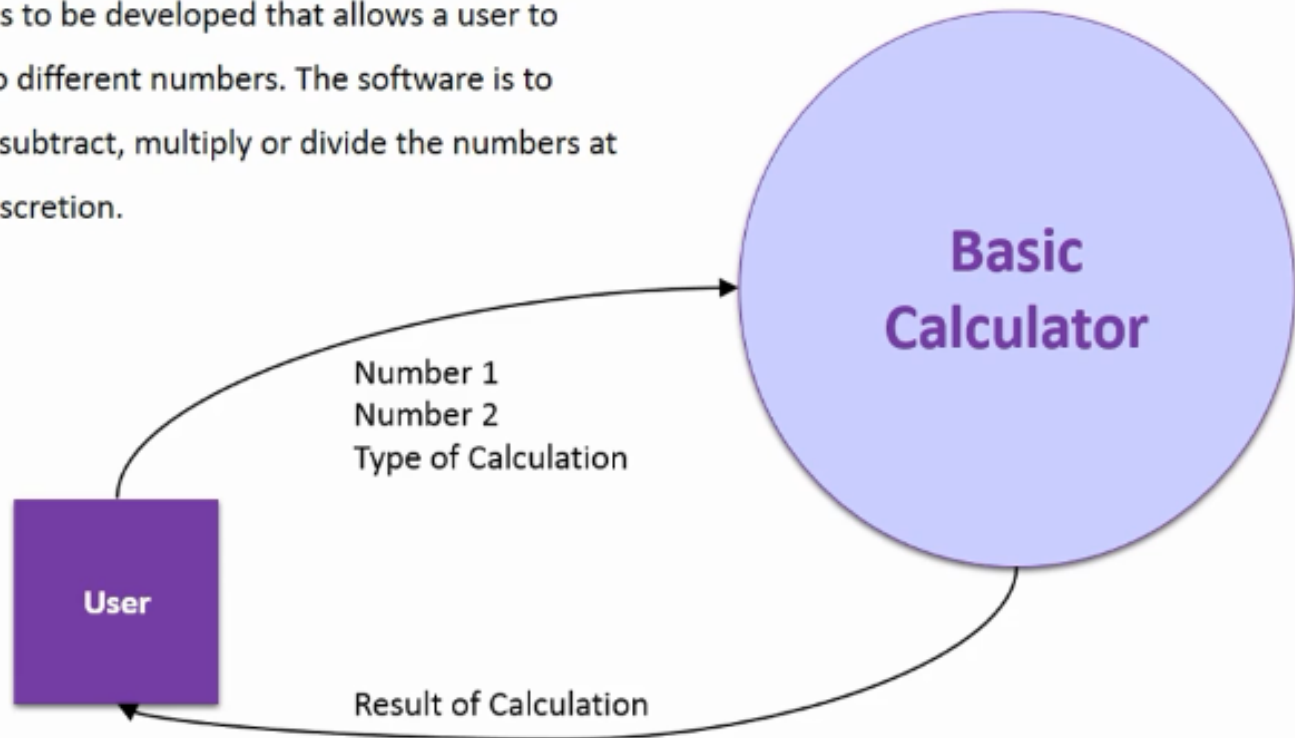
Illustrates the movement of data from one entity / process to another. A Data Flow line is supported by text stating what data is being sent / retrieved

Context Models

Context Diagrams

Example: Basic Calculator

A program is to be developed that allows a user to enter in two different numbers. The software is to either add, subtract, multiply or divide the numbers at the users discretion.





2.2. Behavioral Model

Use case diagrams:

- shows the interactions between a system and its environment (actors).

Activity diagrams:

- shows the activities involved in a process or in data processing.

Sequence diagrams:

- shows interactions between objects within the system.

Start chart diagrams:

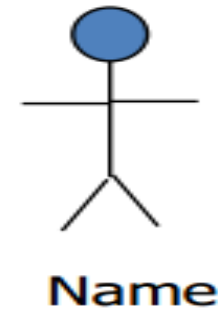
- shows how the system reacts to internal and external events.



Elements of use case diagram:

a. Actor:

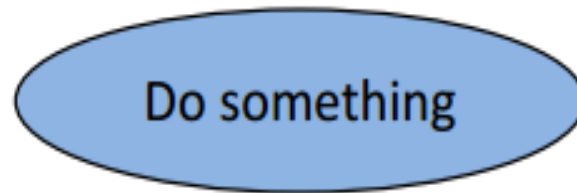
- Actor is someone interacting with use case (system function).
- Actor has responsibility toward the system (inputs), and Actor have expectations from the system (outputs)
- Actor triggers use case.



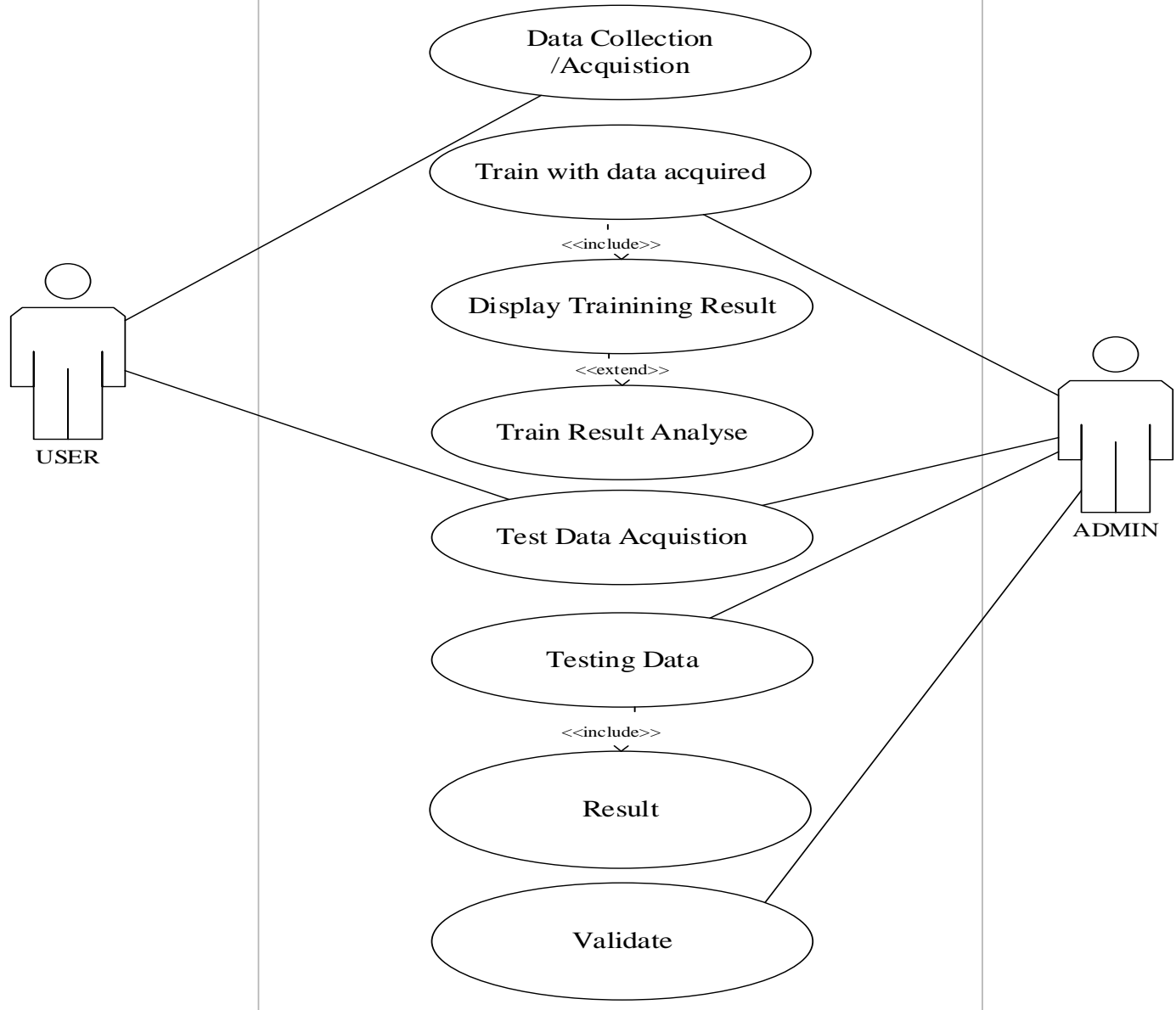


b. Use case

- System function (process—automated or manual).
- Each Actor must be linked to a use case, while some use cases may not be linked to actors.



Signature Verification System





2.4. ER Model

- An Entity Relationship (ER) Diagram is a **type of flowchart that illustrates how “entities”** such as people, objects or concepts **relate to each other** within a system.
- ER Diagrams are most often **used to design or debug relational databases** in the fields of software engineering, business information systems, education and research.
- They **use a defined set of symbols** such as rectangles, diamonds, ovals and connecting lines to depict the interconnectedness of entities, relationships and their attributes.



Components of ER Model

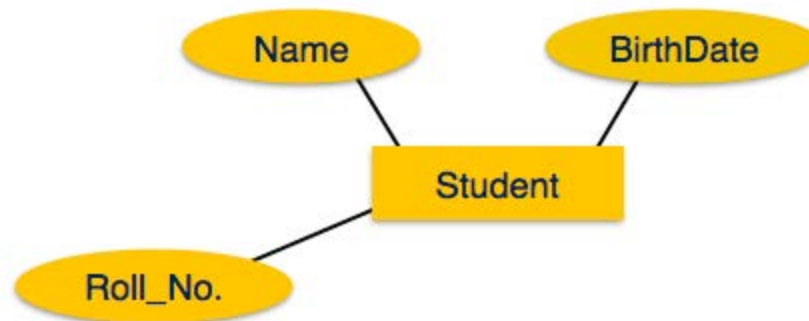
■ Entity

A definable thing—such as **a person, object, concept or event**. Examples: a customer, student, car or product. Typically shown as a rectangle.



■ Attributes

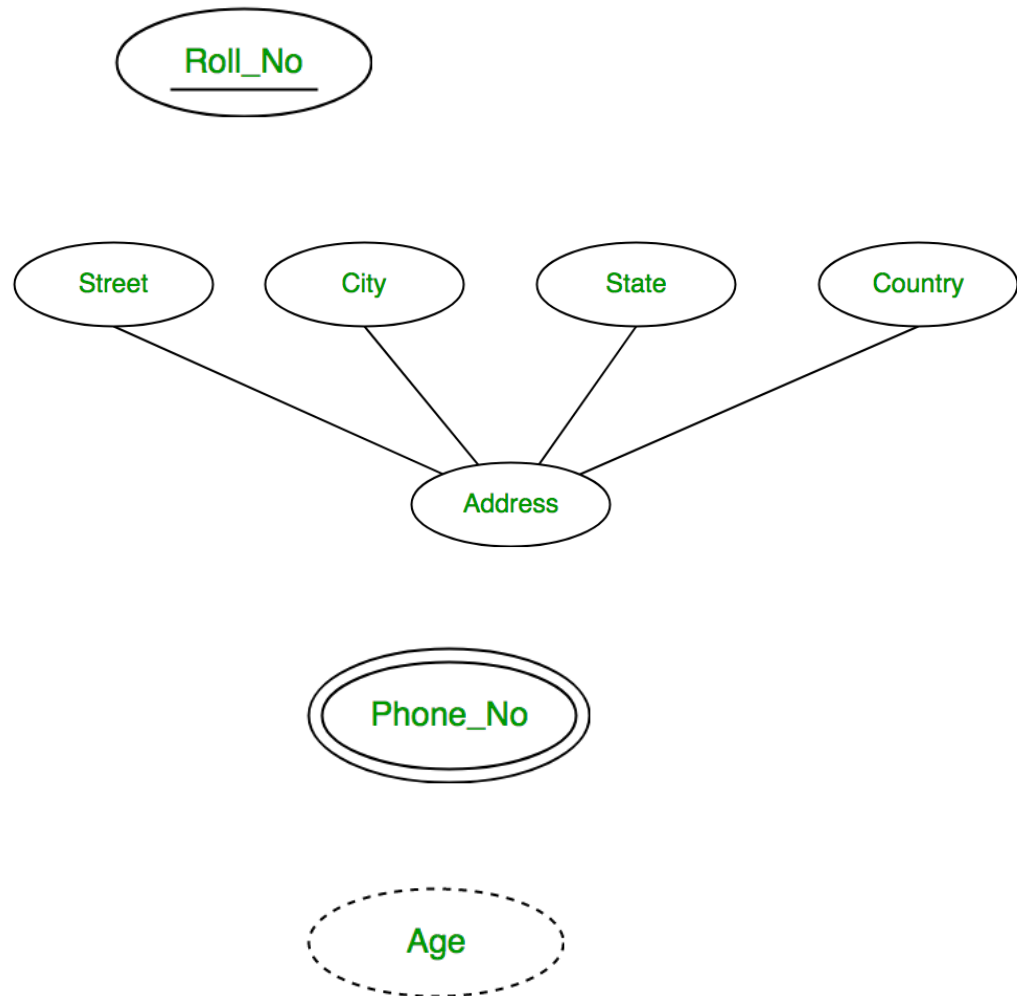
A property or characteristic of an entity. Often shown as an oval or circle.



Components of ER Model

Attributes Types:

- Key
- Composite
- Multivalued
- Derived



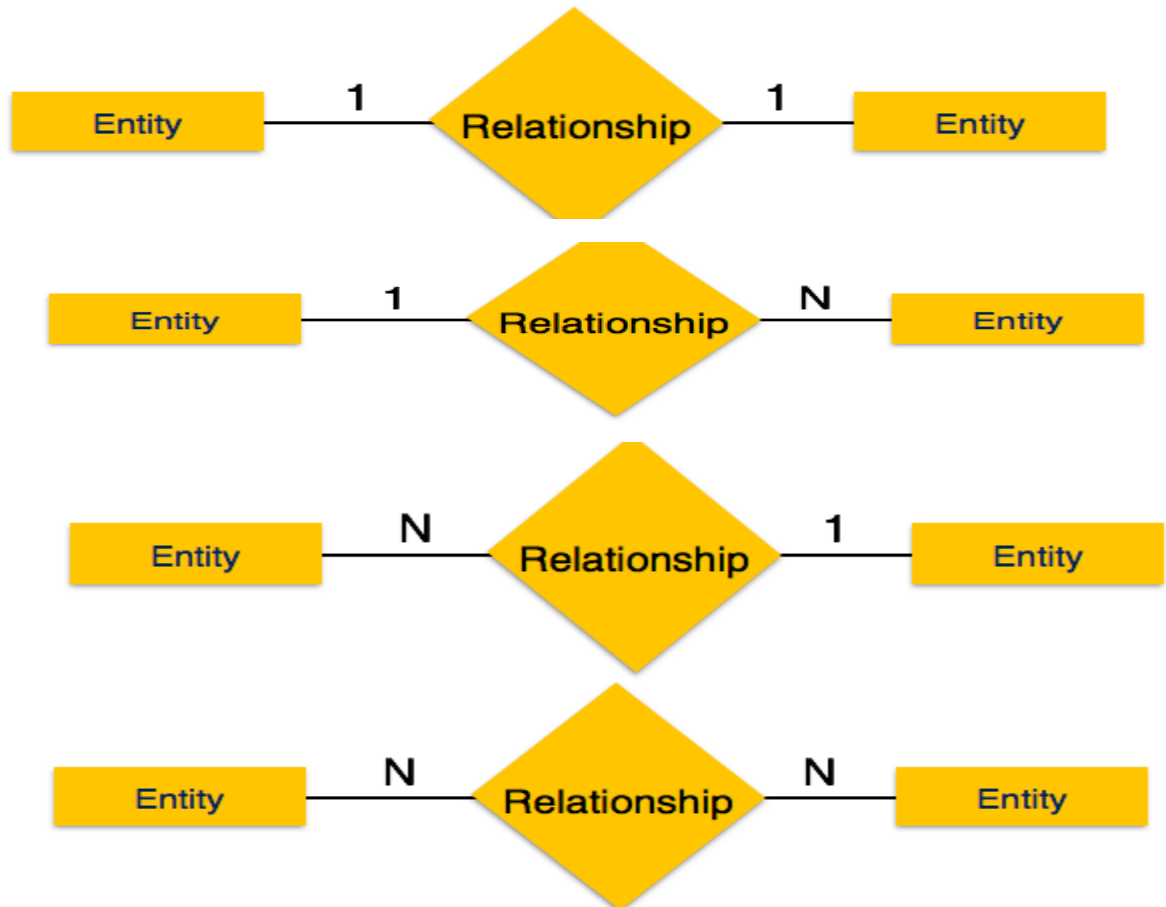


Components of ER Model

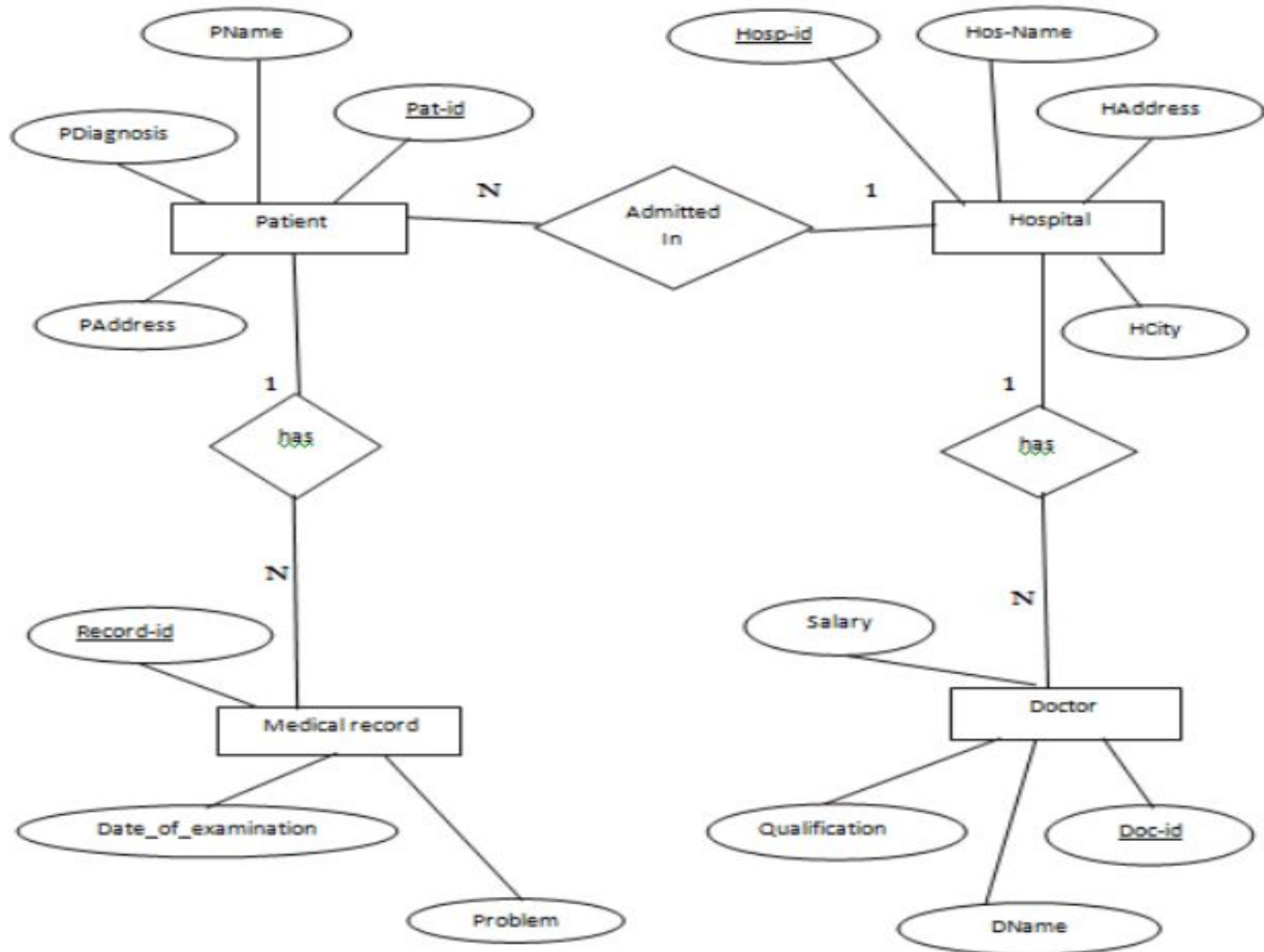
Relationship

How entities act upon each other or are associated with each other.

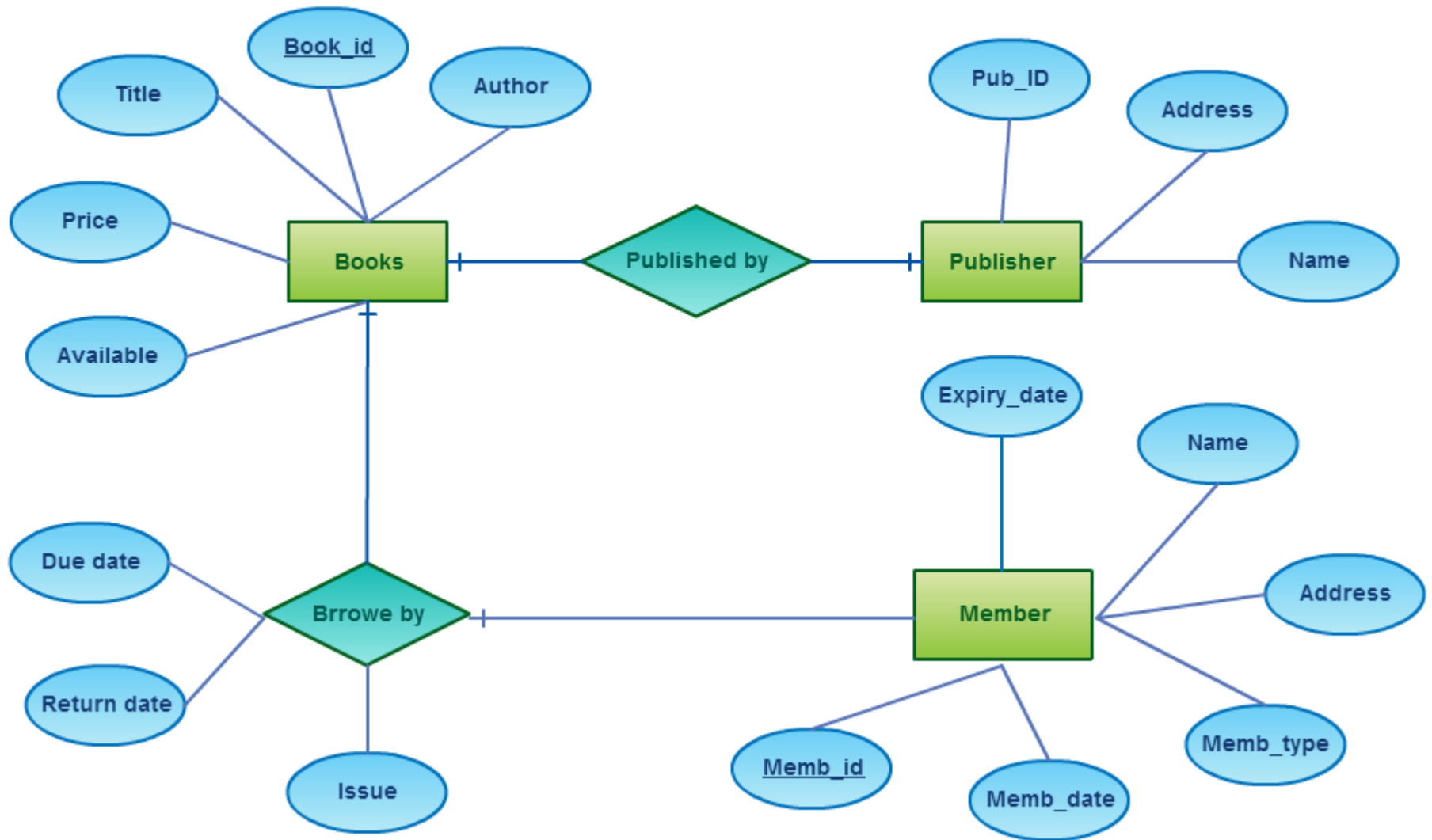
- One to one
- One to many
- Many to one
- Many to many



ER Model Example[HMS]



E-R Diagram of Library Management System





2.4. Data Flow Diagram

- A data flow diagram (DFD) **maps out the flow of information** for any process or system. **It uses defined symbols like rectangles, circles and arrows and short text labels**, to show data inputs, outputs, storage points and the routes between each destination.
- They can be **used to analyze an existing system** or model a **new one**.
- components of data flow diagrams:

External entity:

- Entities are **source and destination** of information.
- Entities are **represented by a rectangles** with their respective names.

Process:




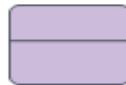




- **Activities, operation and action** taken on the data.
- Represented by **Circle or Round-edged** rectangles.



2.4. Data Flow Diagram

- **Data store:**
Files or repositories that hold information for later use, such as a database table or a membership form.
- **Data Flow:**
The route that data takes between the external entities, processes and data stores.

- **DFD Notations:**

Notation	Yourdon and Coad	Gane and Sarson
External Entity		
Process		
Data Store		
Data Flow		

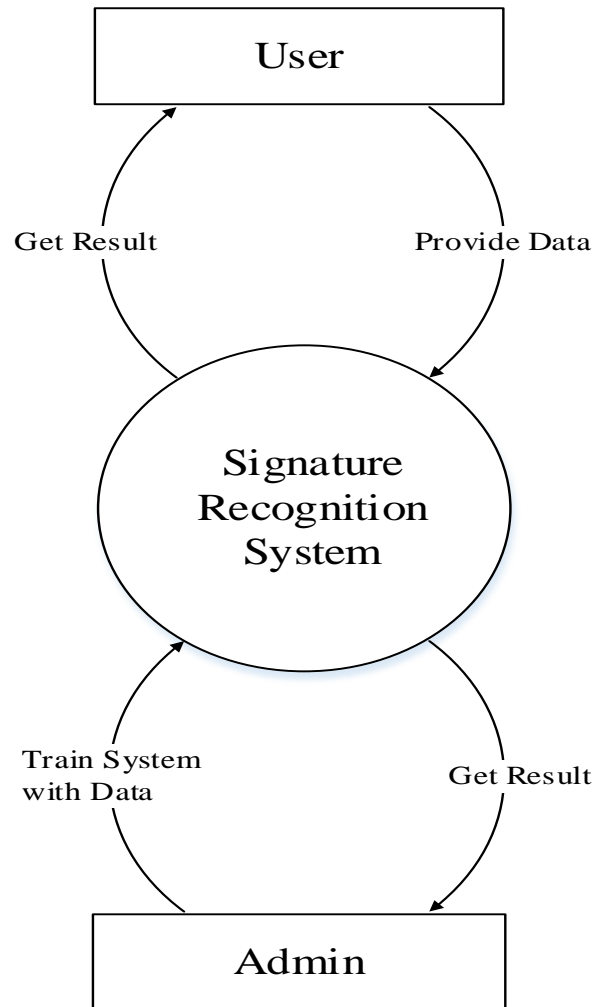


2.4. Data Flow Diagram

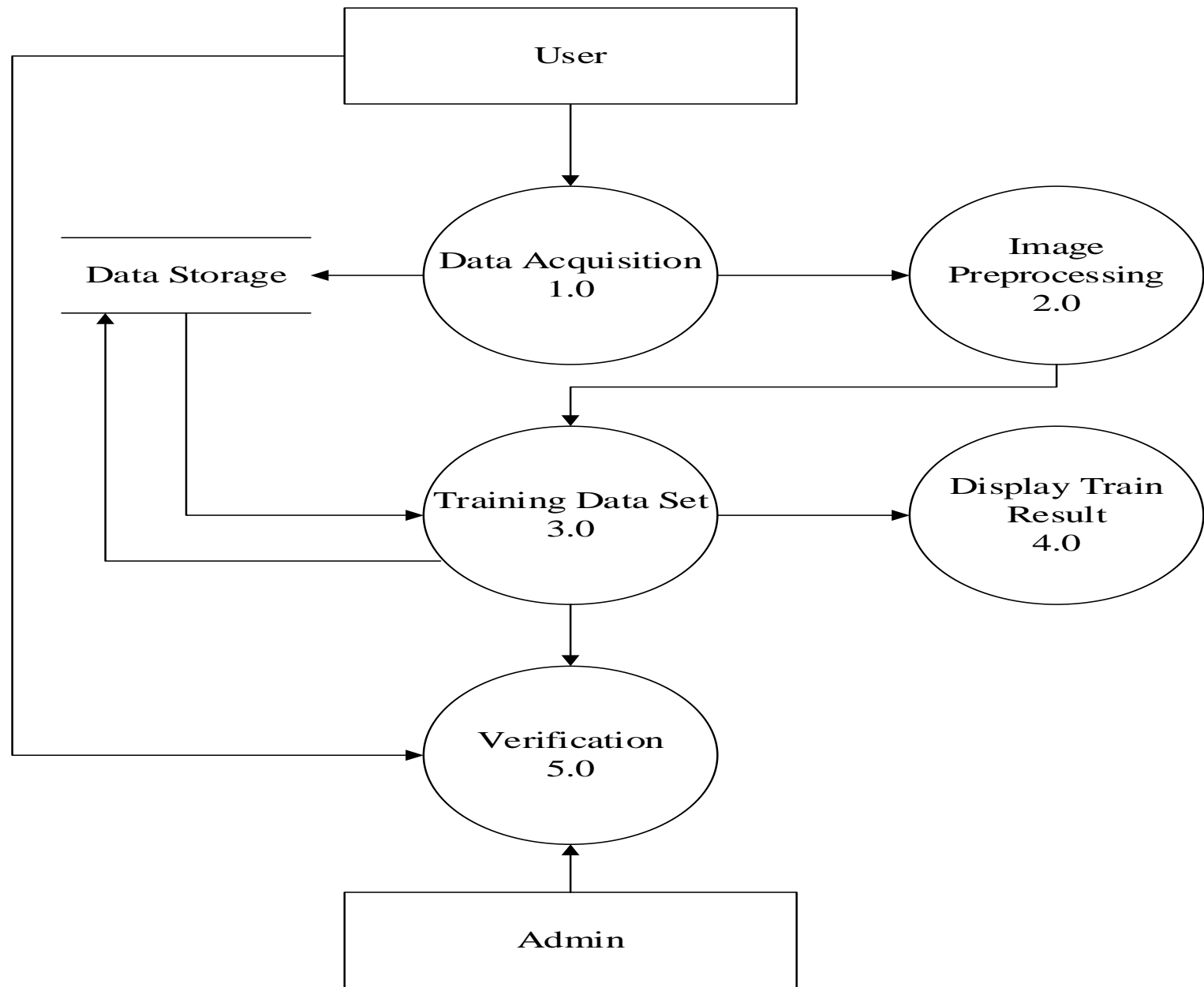
Levels of DFD:

- **Level 0** - **Highest abstraction level** DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are **also known as context level DFDs**.
- **Level 1** - The **Level 0 DFD is broken down into more specific, Level 1 DFD**. Level 1 DFD depicts basic modules in the system and **flow of data** among various modules.
- **Level 2** - At this level, DFD **shows how data flows inside the modules** mentioned in Level 1.

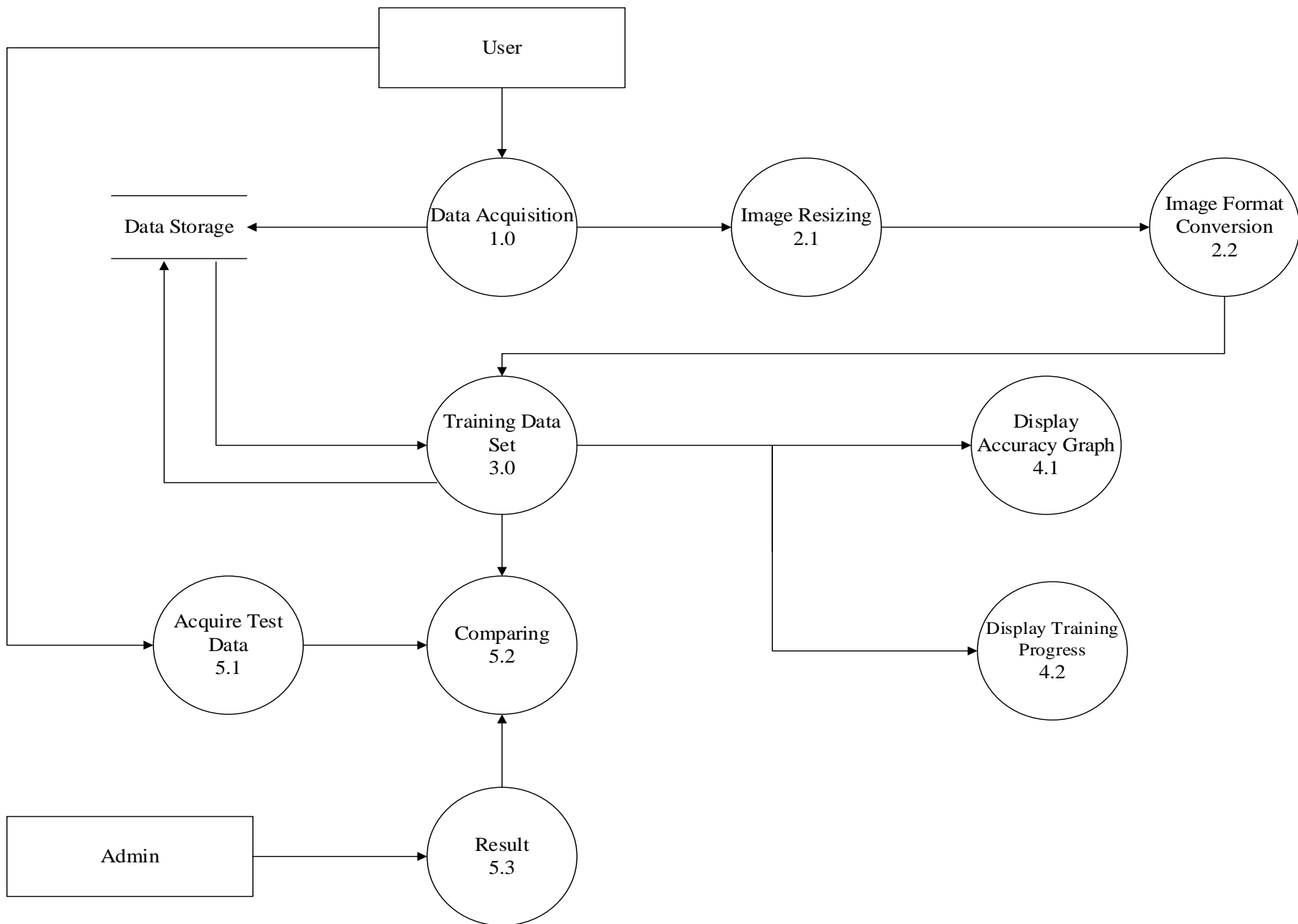
2.4. Data Flow Diagram **Example (SVS)**



Context free DFD [level 0]



Signature Verification system DFD [level 1]



Signature Verification system DFD [level 2]

Workout examples

→ On Case study Examples

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering
Chapter 3
Architectural Design

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

Chapter Three: **Architectural Design**

Course Outline: **6 hours**

1. Architectural design decisions
2. System organization
3. Decomposition styles
4. Control styles
5. Reference architectures



What is Architectural Design?

- It is the design process for identifying the subsystems for making a system and the framework for sub-system control and communication.
- The output of this design process is a description of the software architecture.
- Architectural design is an early stage of the system design process. It represents the link between specification and design processes and is often carried out in parallel with some specification activities.
- It involves identifying major system components and their communications.



Architectural Design

- IEEE defines architectural design as:
- “The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.”
- The software system needs the architectural design to represents the design of software.



Architectural Design

- The software that is built for computer-based systems can exhibit one of many architectural styles. Each style will describe a system category that consists of :
 - ✓ A set of components (e.g.: a database, computational modules) that will perform a function required by the system.
 - ✓ A set of connectors will help in coordination, communication, and cooperation between the components.
 - ✓ Conditions that how components can be integrated to form the system.
 - ✓ Semantic models (logical models) that help the designer to understand the overall properties of the system.
- The use of architectural styles is to establish a structure for all the components of the system.



Architectural Design

Software architectures can be designed at **two levels of abstraction**:

- **Architecture in the small**

It is concerned with the **architecture of individual programs**. At this level, we are concerned with the way that an **individual program is decomposed into components**.

- **Architecture in the large**

It is **concerned with the architecture of complex enterprise systems** that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.



Architectural Design

Three **advantages** of explicitly designing and documenting software architecture:

- **Stakeholder communication:**

Architecture may be used as a focus of discussion by system stakeholders.

- **System analysis:**

Well-documented architecture enables the analysis of whether the system can meet its non-functional requirements.

- **Large-scale reuse:**

The architecture may be reusable across a range of systems or entire lines of products.



Uses of architectural models:

- **As a way of** facilitating discussion **about the system design:**

A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.

- **As a way of** documenting an architecture **that has been designed:**

The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.



Architectural Design decisions

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



Architectural Design decisions

The particular architectural style should depend on the **non-functional system requirements**:

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



Architectural Conflicts

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



Architectural models

- **static models:** which shows the major system components.
- **dynamic models:** which show the organization of the system when it is executing.
- **Interface model:** that defines sub-system interfaces.
- **Relationships model:** such as a data-flow model that shows sub-system relationships.
- **Distribution model:** that shows how sub-systems are distributed across computers.



System Organization

Reflects the basic strategy that is used to structure a system.

Three types:

- The Repository model
- Client-Server model
- Abstract Machine (Layered) model



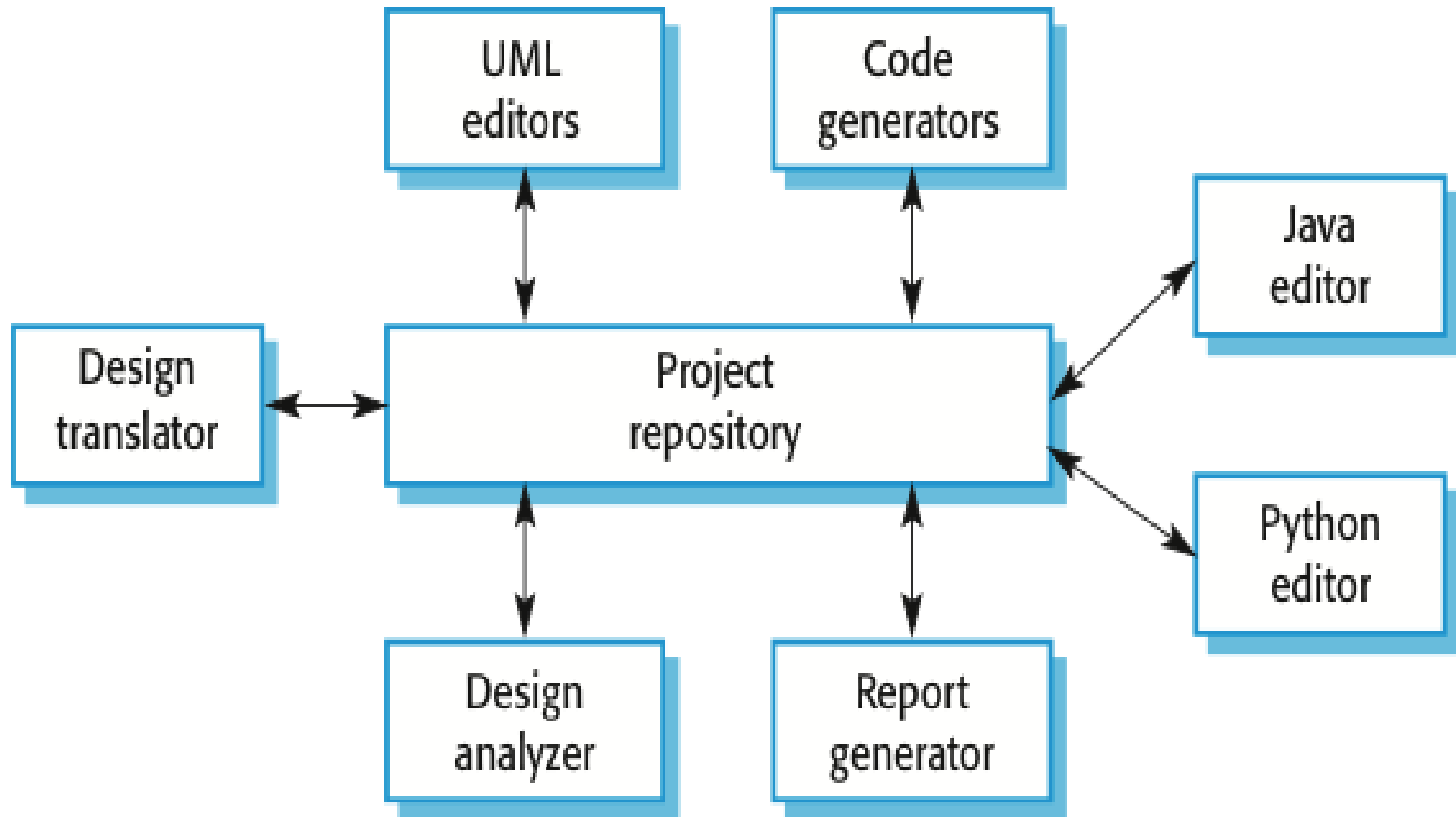
System Organization

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 as this is an efficient data sharing mechanism

System Organization

The Repository model Architecture:





System Organization

The Repository model:

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent--they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.



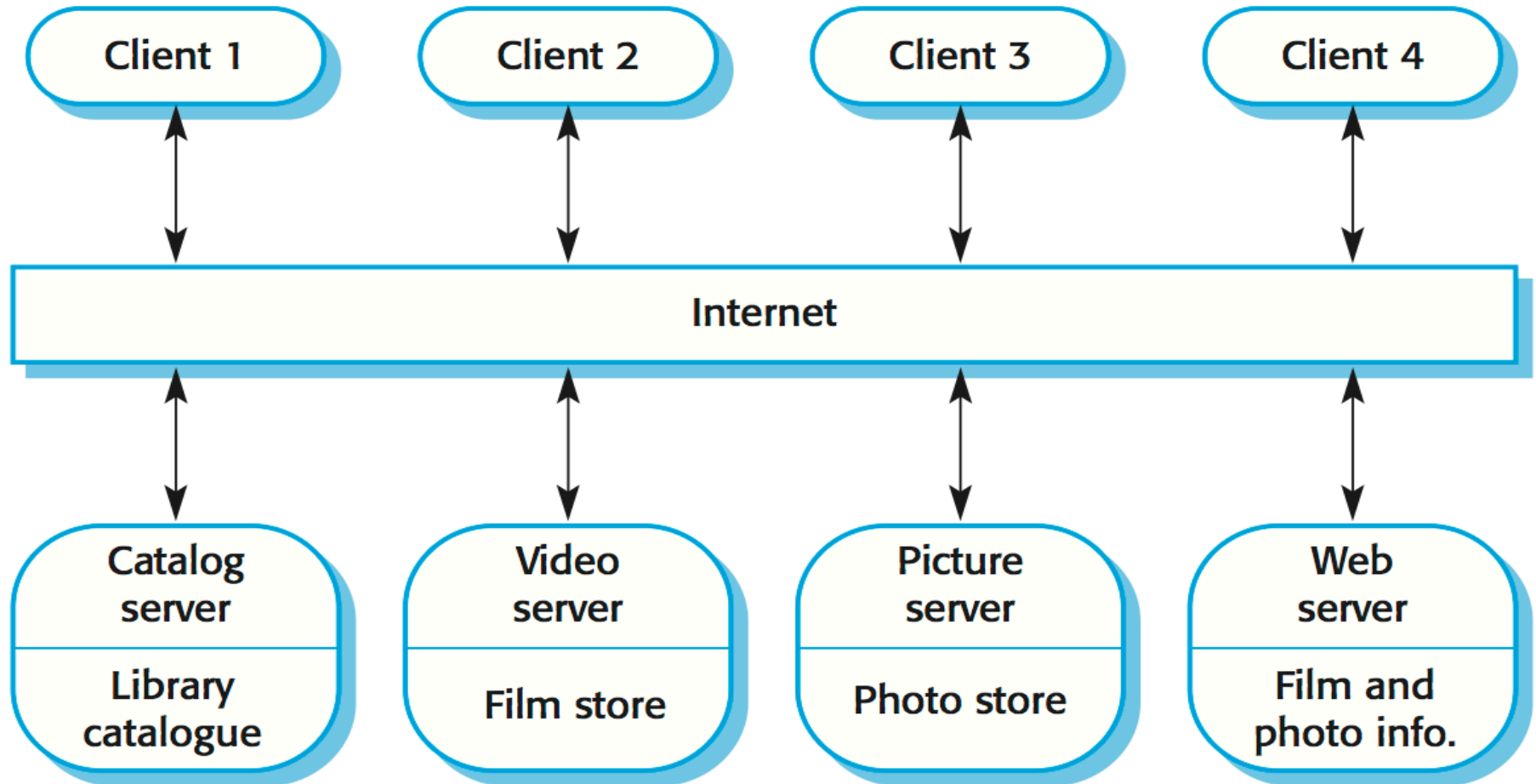
System Organization

The Client-Server model:

- **Distributed system model** which shows how data and processing is distributed across a range of components, but can also be implemented on a single computer.
- **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.

System Organization

The Client-Server Architecture:





System Organization

The Client-Server Architecture:

Name	Client-server
Description	In a client-server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.



System Organization

The Layered 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.



System Organization

The Layered Architecture:

User interface

User interface management
Authentication and authorization

Core business logic/application functionality
System utilities

System support (OS, database etc.)



System Organization

The Layered Architecture:

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Thank You!!!

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering
Chapter 3
Architectural Design

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

Chapter Three: **Architectural Design**

Course Outline: **6 hours**

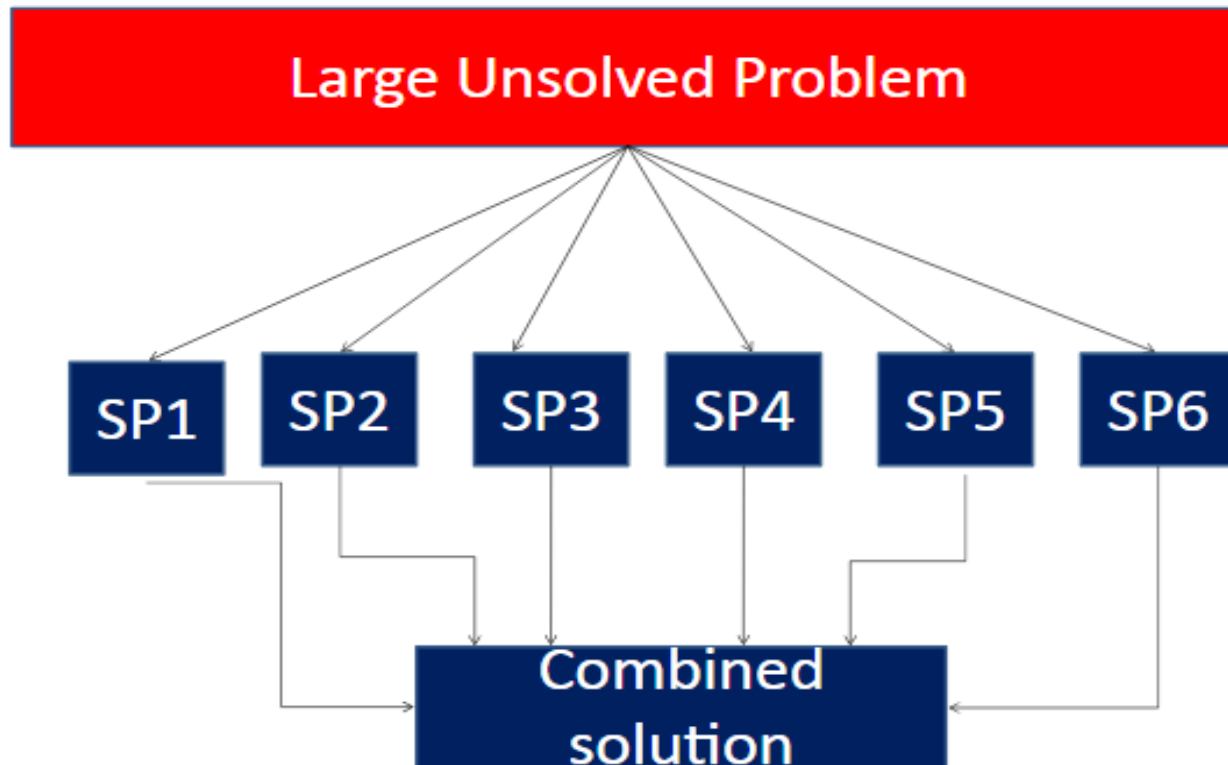
1. Architectural design decisions
2. System organization
3. **Decomposition styles**
4. **Control styles**
5. Reference architectures



Modular Decomposition styles?

Styles of decomposing sub-systems into modules.

Decomposition Example





Modular Decomposition styles?

Sub system and Components

- 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.
- To make it short :
 - ✓ a subsystem can exist without its parent system.
 - ✓ a component cannot be used alone and must be part of a system to exist.



Modular Decomposition styles?

Sub system and Components

To take an analogy :

- ✓ a car is a sub-system of travel infrastructure.
- ✓ a wheel is a component of the car.



Modular Decomposition styles?

- structural level where sub-systems are decomposed into modules.
 - Two modular decomposition models
- ✓ *Object Oriented decomposition:*
- An object model where the system is decomposed into interacting object.*
- ✓ *Function oriented decomposition :*
- A pipeline or data-flow model where the system is decomposed into functional modules which transform inputs to outputs.*



Modular Decomposition?

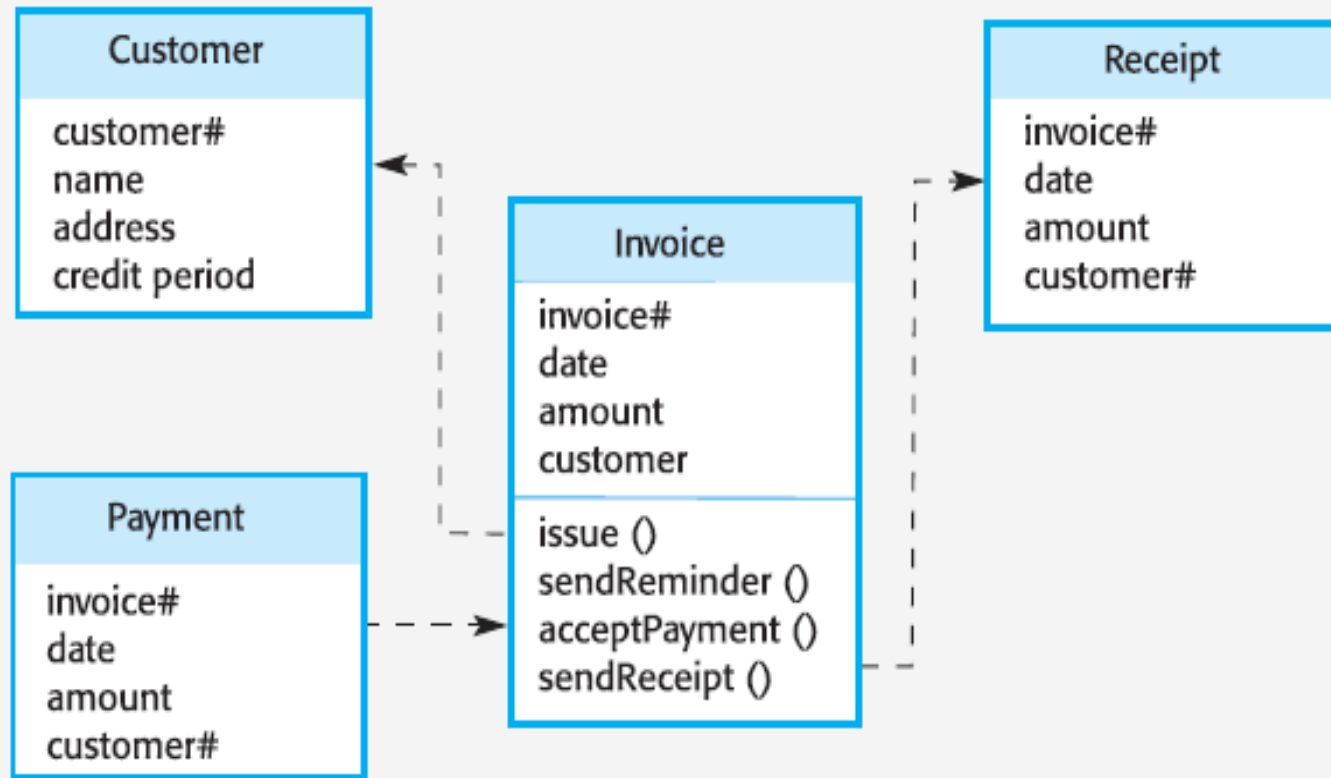
Object models

- 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.



Modular Decomposition?

Object models (Invoice processing system)





Modular Decomposition?

Object models (advantages)

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



Modular Decomposition?

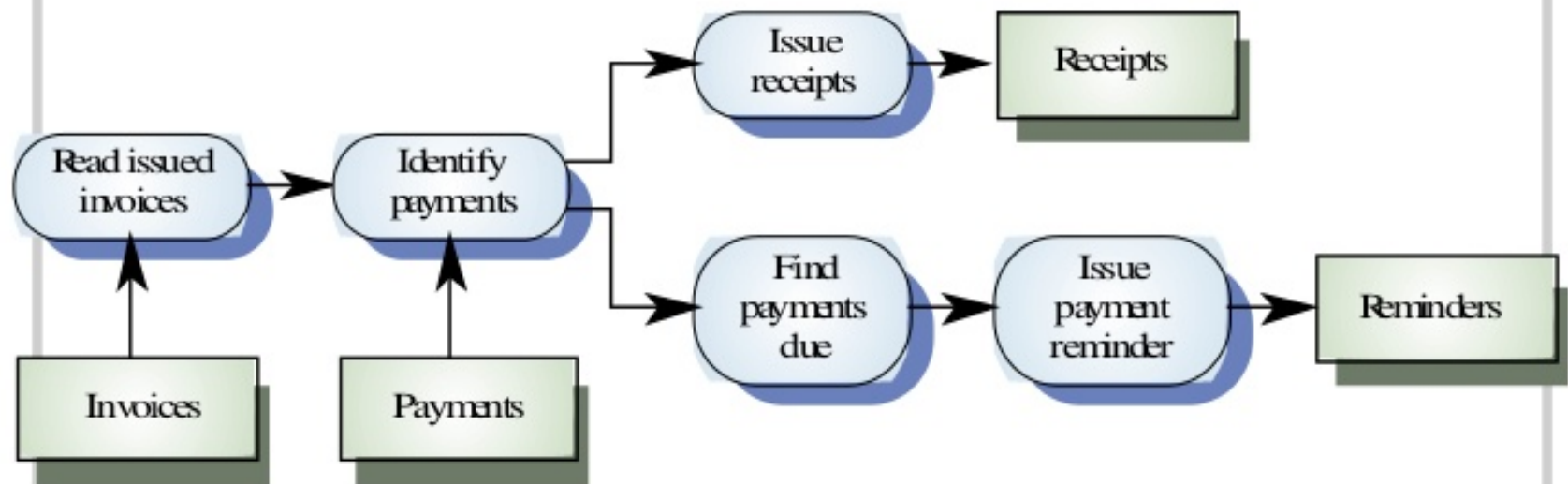
Functional models

- In function-oriented design, the system is divided into many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.
- This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing (providing the means for data hiding) the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.

Modular Decomposition?

Functional models

Invoice processing system





Modular Decomposition?

Functional model design process:

- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions changes data and state of entire system.
- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.



Modular Decomposition?

Functional model Advantages

Assignment 2

Refer: Ian Sommerville's book from Library



Control styles/models

Are concerned with the control flow between sub-systems.

Two generic control styles

- ✓ **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.



Control styles/models

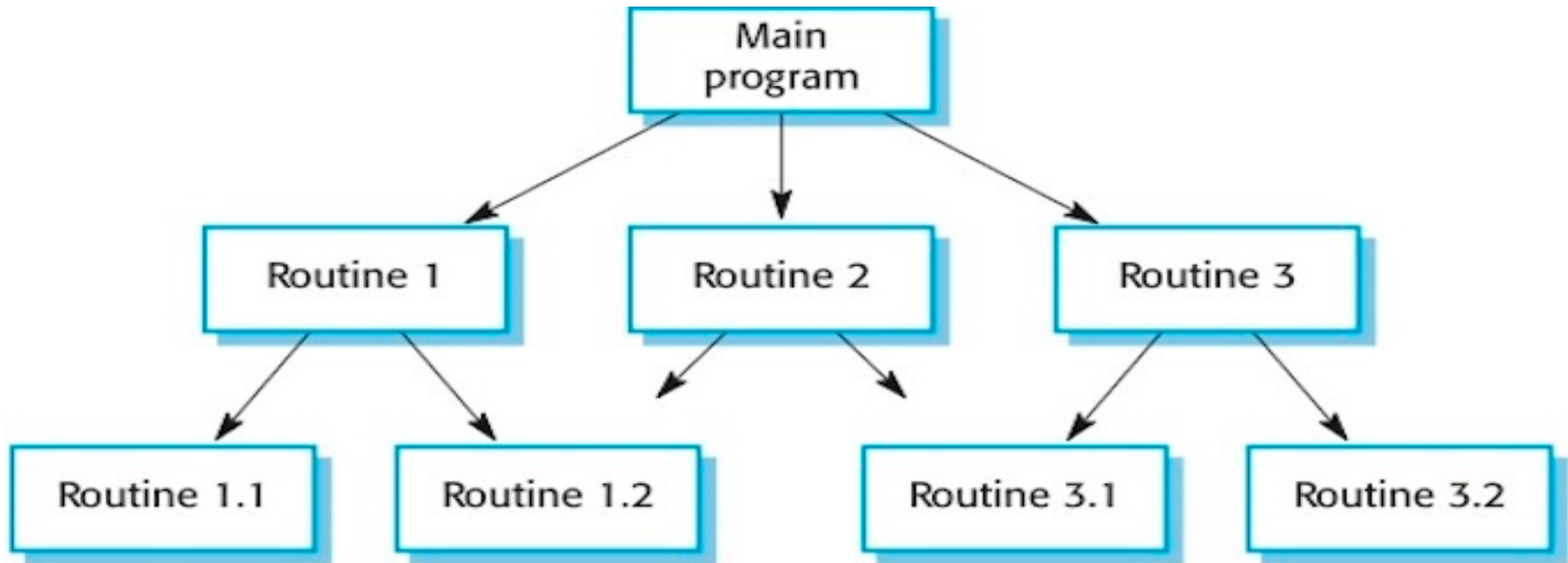
Centralized control

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.

Centralized Control styles/models

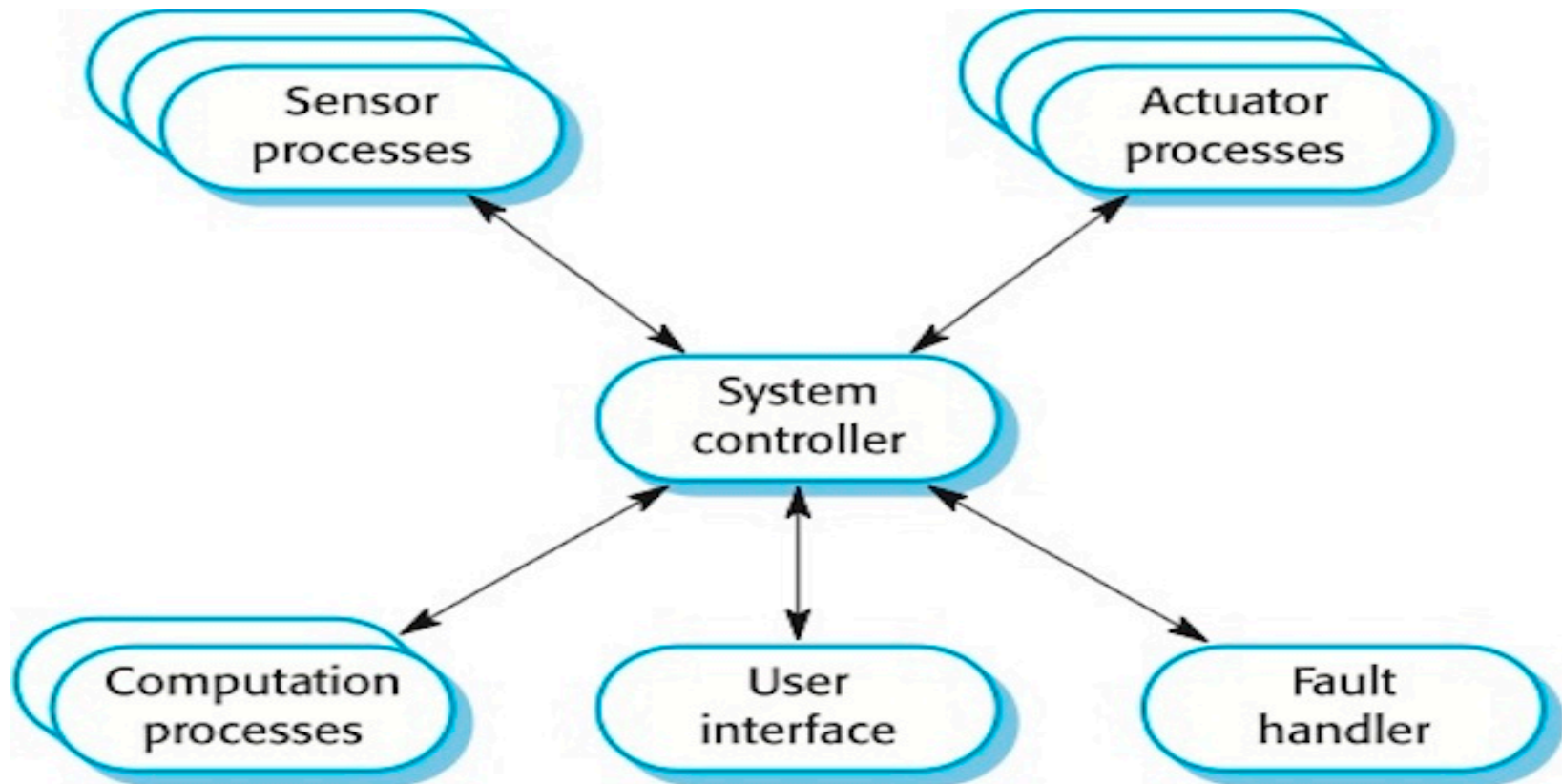
Call-return model:



- ✓ The main program can call Routines 1, 2 and 3; Routine 1 can call Routines 1.1 or 1.2; Routine 3 can call Routines 3.1 or 3.2; and so on.

Centralized Control styles/models

Manager model (real time system control):





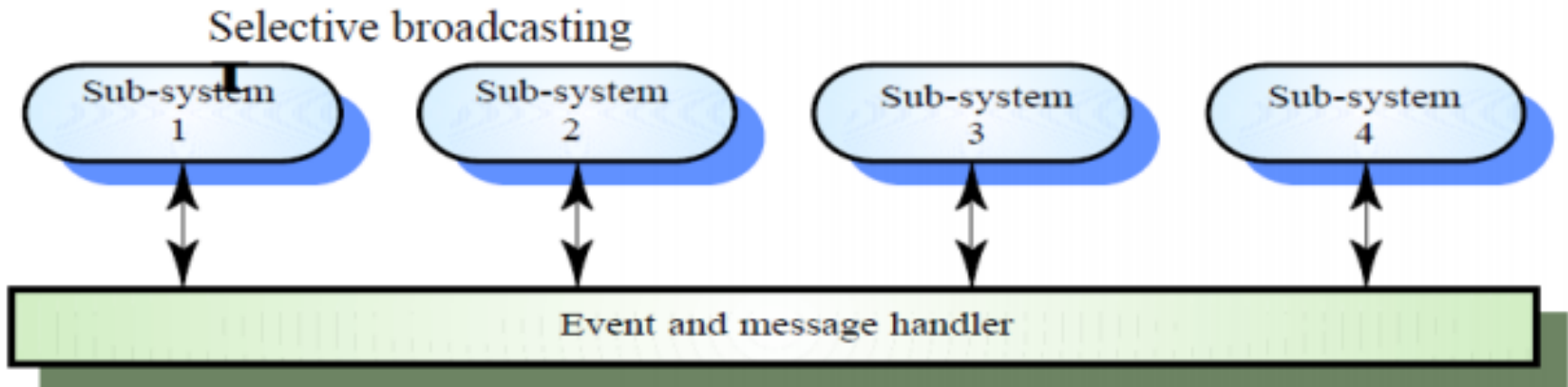
Control styles/models

Event-driven systems:

- Two principal event-driven models
 - ✓ **Broadcast models.**
 - ✓ **Interrupt-driven models.**
- ✓ **Broadcast model**
 - 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.

Event Driven control styles/models

✓ Broadcast model



- components register an interest in specific events. When these events occur, control is transferred to the component that can handle the event.



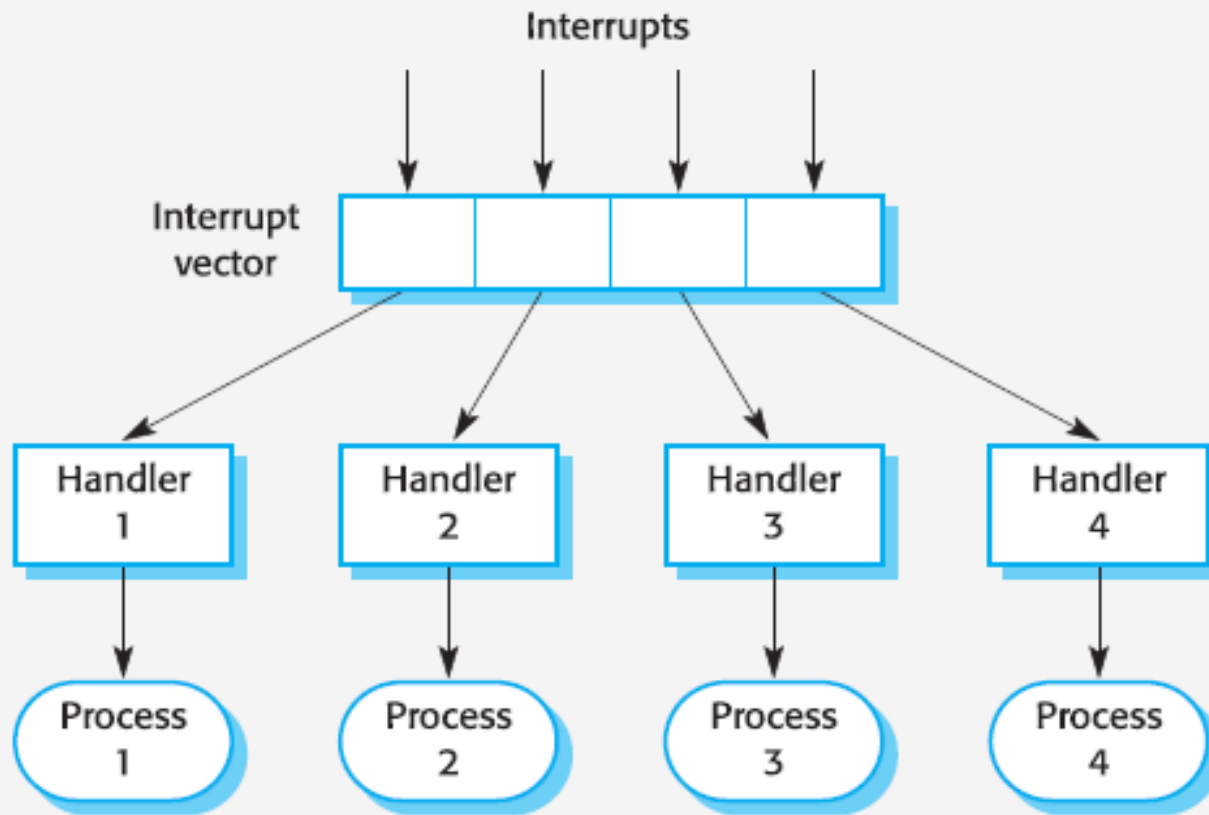
Event-driven Control styles/models

Interrupt-driven models.

- 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.
- Allows fast response but complex to program and difficult to validate.

Event-driven Control styles/models

✓ Interrupt-driven models.



Thank You!!!

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering
Chapter 3
Architectural Design

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

Chapter Three: **Architectural Design**

Course Outline: **6 hours**

1. Architectural design decisions
2. System organization
3. Decomposition styles
4. Control styles
5. **Reference architectures**

Objectives

- To explain the advantages and disadvantages of different distributed systems architectures
- To discuss two principal models of distributed systems architecture -client-server and distributed object architectures
- To understand the concept of object request brokers and the principles underlying the CORBA standards
- To introduce peer-to-peer and service-oriented architectures as new models of distributed computing.

Topics covered

- Multiprocessor architectures
- Client-server architectures
- Distributed object architectures
- Inter-organisational computing

Distributed systems

- Virtually all large computer-based systems are now distributed systems.
- Here, Information processing is distributed over several computers rather than confined to a single machine.
- Distributed software engineering is therefore very important for enterprise computing systems.

System types

- Personal systems that are not distributed and that are designed to run on a personal computer or workstation.
- Embedded systems that run on a single processor or on an integrated group of processors.
- Distributed systems where the system software runs on a loosely integrated group of cooperating processors linked by a network.

Distributed system characteristics(Advantages)

- **Resource sharing**

- A distributed system allows sharing of hardware and software resources such as disks, printers, files & compilers.

- **Concurrency**

- Here, several processes may operate at the same time on separate computers on the network called concurrent processing.
 - Concurrent processing to enhance performance.

- **Scalability**

- It is the capability of the system that can be increased by adding new resources to cope with new demands in the system.
 - Increased throughput [how many units of information a system can process in a given amount of time] by adding new resources.

- **Fault tolerance**

- The availability of several computers & potential for replicating information means the distributed system can be tolerant of some hardware and software failures i.e.
 - The ability to continue in operation after a fault has occurred.

Distributed system disadvantages

■ Complexity

- Typically, distributed systems are more complex than centralised systems; makes it more difficult to understand their emergent properties & to test these systems.
- Example- rather than the performance of the system being dependent on execution speed of one processor, it depends
 - on the network bandwidth and
 - speed of the processor on the network

■ Security

- The system may be accessed from several different computers, & the traffic on the network may subject to eavesdropping.
- This makes it more difficult to ensure that the integrity of the data in the system is maintained &
- The services are not degraded by the denial of attack . i.e.
- They are more susceptible to external attack.

Distributed system disadvantages

- **Manageability**

- The computers in a system may be different types and run on different versions of operating system.
- Fault in one machine may propagate to other machines with unexpected consequences.
- Means more effort required for system management.

- **Unpredictability**

- All users of the WWW know, distributed systems are unpredictable in their response.
- Unpredictable responses depending on the system organisation and network load.
- As all these may vary over a short period, the time taken to a user request may vary dramatically from one request to another.

Distributed systems architectures

- Client-server architectures
 - Distributed **services** which are **called on by clients**.
 - **Servers** that **provide services** are treated differently from clients that use services.
- Distributed object architectures
 - **No distinction** between clients and servers.
 - The server may be thought of as a set of interactive objects whose location is irrelevant.
 - **Any object on the system may provide and use services** from other objects.

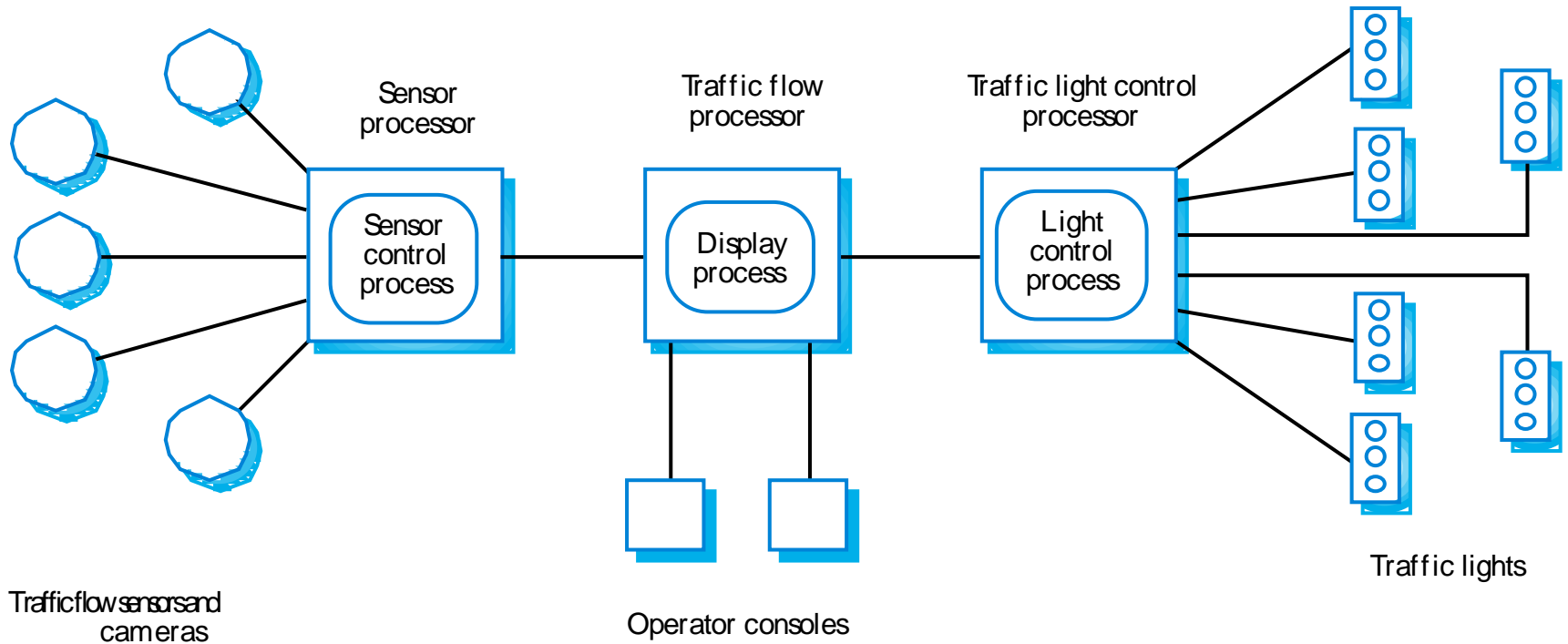
Middleware

- The **components of the distributed system** may be implemented on different programming language & may execute on the completely different
 - ✓ types of processors
 - ✓ Models of data
 - ✓ Information representation &
 - ✓ Protocols
- Thus, it requires to manage these diverse parts, ensure that they communicate and exchange data.
- **Middleware refers to software that manages and supports the different components of a distributed system.** In essence, it sits in the *middle* of the system.
- Middleware is usually off-the-shelf [E.g.: MS package] rather than specially written software.
- Examples
 - ✓ Transaction processing monitors;
 - ✓ Data converters;
 - ✓ Communication controllers.

3.6 Multiprocessor architectures

- Simplest distributed system model where **System composed of multiple processes** which may (but need not) **execute on different processors**.
- This process is common in large real-time systems. These systems:
 - ✓ Collect information
 - ✓ Make decision using information &
 - ✓ Send signals to actuator [A mechanism that causes a device to be turned on or off, adjusted or moved] to modify the system's environment.
- Distribution of process to processor may be pre-determined or may be under the control of a dispatcher [Software that determines what pending tasks should be done next and assigns the available resources to accomplish it].

Example - A multiprocessor traffic control system



Example - A multiprocessor traffic control system

- In fig. – a simplified model of the traffic control system is shown.
- A set of distributed sensors collects information on the traffic flow & processes locally.
- Operators make decisions using this information & give instruction to a separate traffic light control process
- Here, there are **separate logical processes** for managing sensors, control room & traffic light which **run on separate processors**.

3.7 Client-server architectures

- The application is modelled as a set of servers that provide services and a set of clients that use these services.
- Clients know of servers but servers need not know of clients.
- Clients and servers are separate logical processes as shown in fig below (Fig.1)
- Several server processes can run on a single server processor so there is not necessarily 1:1 mapping between processors & processes.

Example - A client-server system

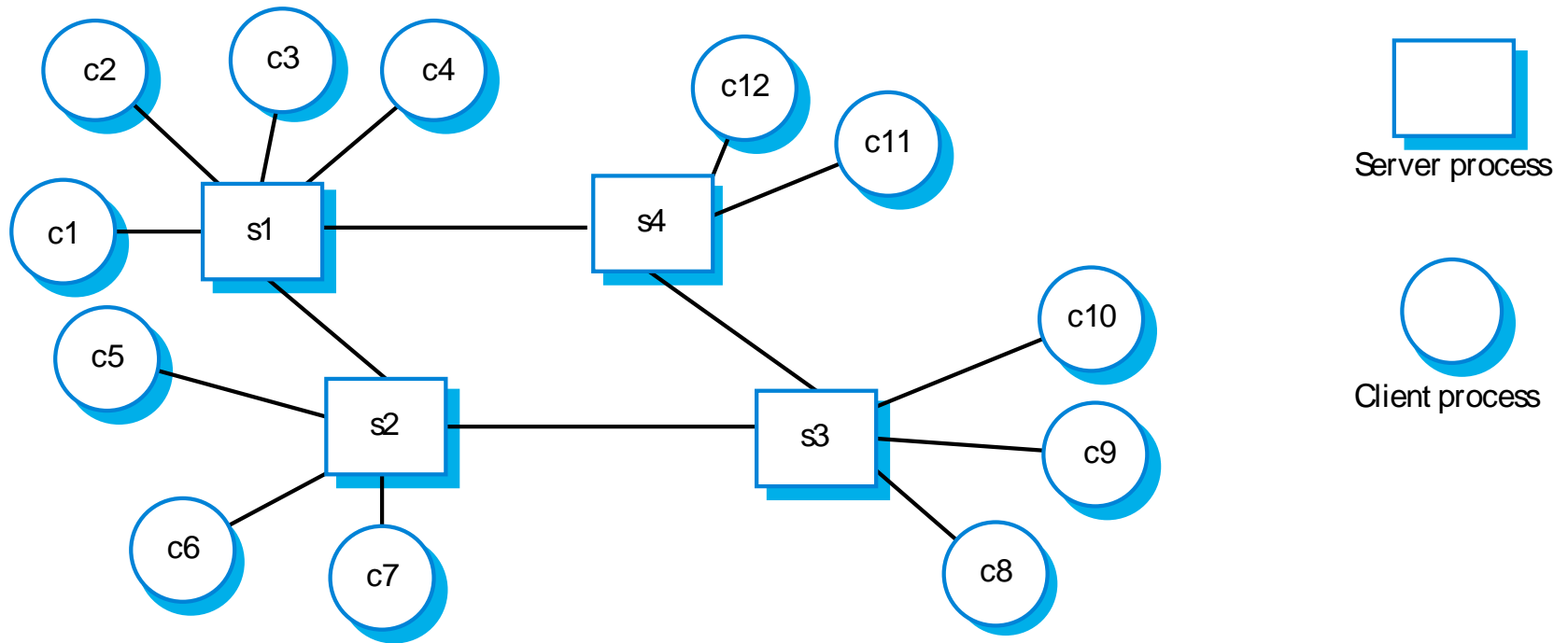


Fig. 1

Example - Computers in a C/S network

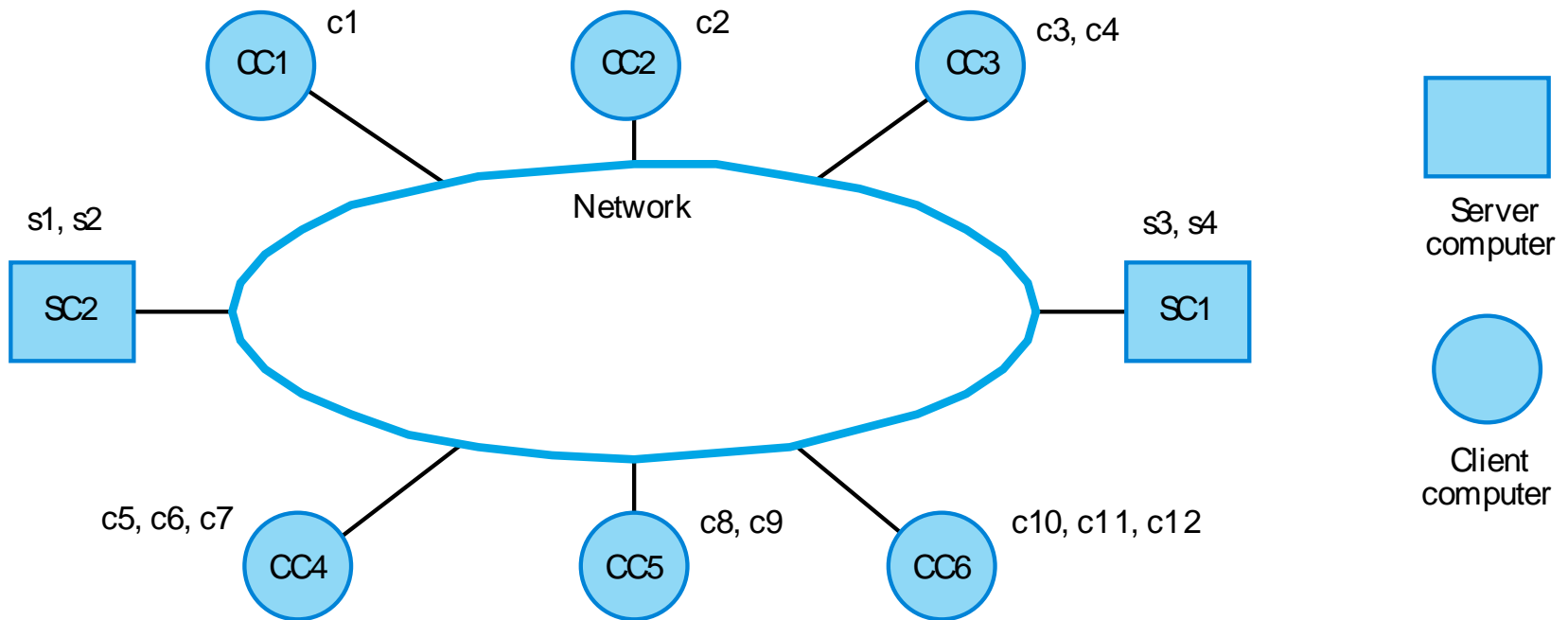


Fig.2

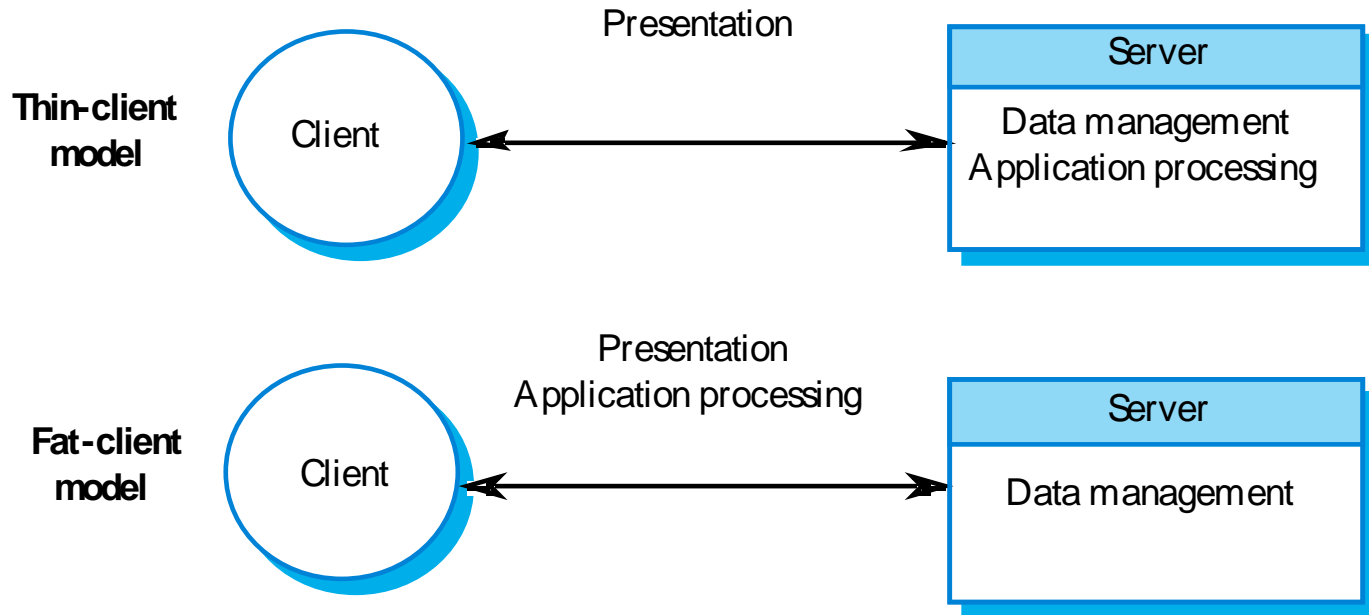
Example - Computers in a C/S network

- Fig. 2 shows the physical architecture of the system with six client & two server computers.
- These can run the client & server processes as shown in Fig. 1

Thin and fat clients

- The simplest client server architecture is called two tier client server architecture, where an application is organized as
 - ✓ a server(or multiple identical servers) &
 - ✓ a set of clients
- The two tier client server architecture can take two forms:
- **Thin-client model**
 - ✓ In a thin-client model, all of the application processing and data management is carried out on the server.
 - ✓ The client is simply responsible for running the presentation software.
- **Fat-client model**
 - ✓ In this model, the server is only responsible for data management.
 - ✓ The software on the client implements the application logic and the interactions with the system user.

Thin and fat clients



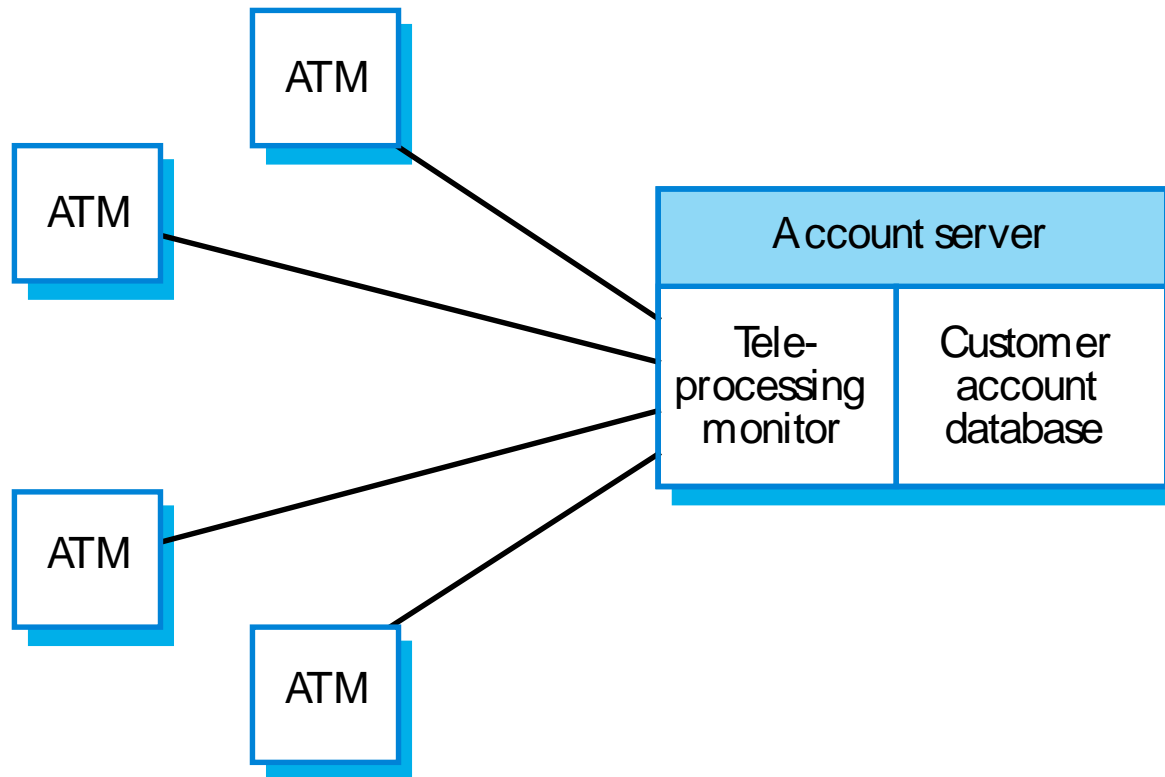
Thin client model

- Used when legacy systems [software that has been around a long time and still fulfills a business need, e.g. **voicemail** system] **are** migrated to client server architectures.
- The legacy system acts as a server in its own right with a graphical interface implemented on a client.
- A major disadvantage is that it **places a heavy processing load on both the server and the network.**

Fat client model

- More processing is delegated to the client as the application processing is locally executed.
- The server is essentially a transaction server that manages all database transactions.
- Example- Banking ATM system, where ATM is the client & the server is a mainframe running the customer account database.
- The hardware in the teller machine carries out a lot of customer related processing associated with a transaction.
- More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.

A client-server ATM system



A client-server ATM system

- Fig. above is the ATM distributed system
- The ATMs are not connected directly to the customer database but to a teleprocessing monitor.
- It is a middleware system that organizes communication with remote clients & serializes the client transaction processing by the database.
- Using serial transaction means that the system can recover from faults without corrupting system data.

Disadvantages-Fat client model

- The fat-client model distributes processing more effectively than thin client model but the system management is more complex
- Application functionality is spread over many computers
- When the application software is to be changed, reinstallation is needed on every computer
- This can be a major cost if there are hundreds of clients in the system.

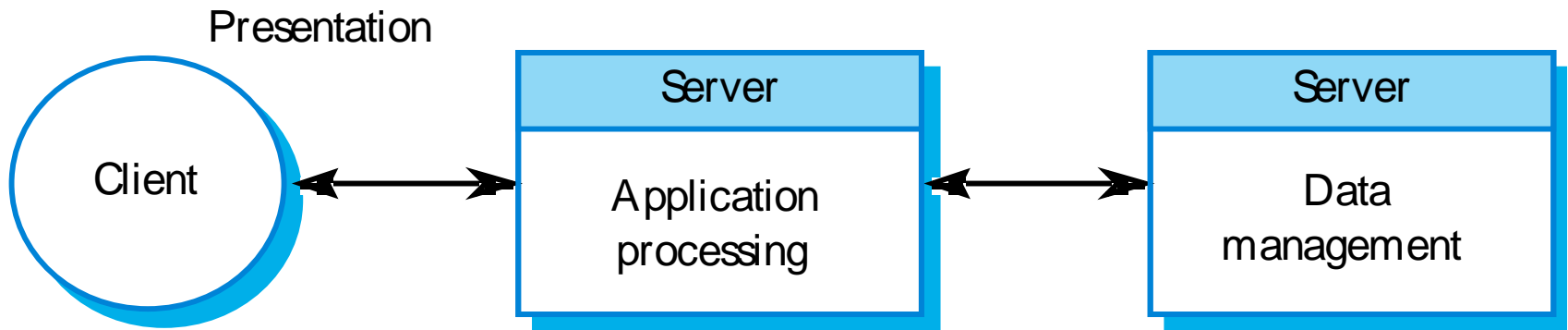
Disadvantages- Two-tier architecture

- The three logical layers-presentation, application processing & data management must be mapped onto two computer systems-the client & the server.
- This may either be problems with scalability & performance if the thin client model is chosen, or the problems of system management if the fat client model is used
- To avoid these issues, a three-tier client server architecture is used.

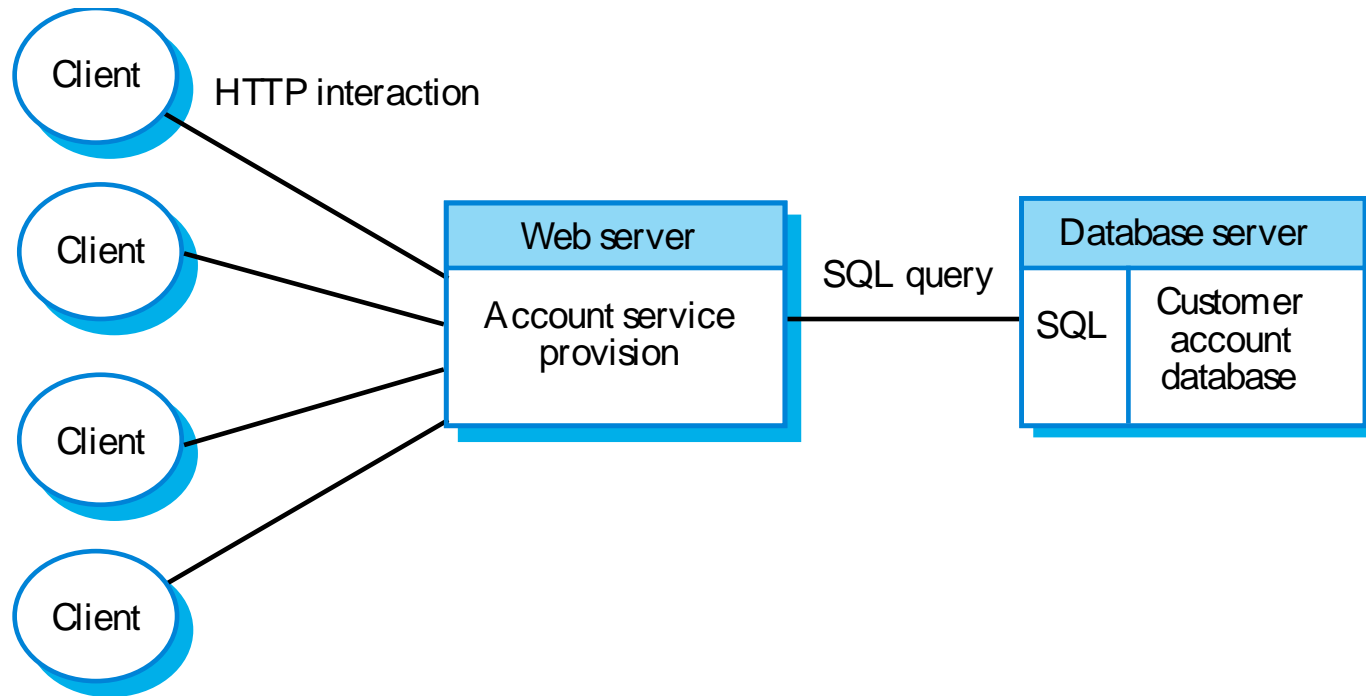
Three-tier architectures

- In a three-tier architecture,
 - ✓ the presentation,
 - ✓ the application processing &
 - ✓ the data management are logically separate processes that execute on a separate processor.
- Allows for better performance than a thin-client approach and is simpler to manage than a fat-client approach.
- A more scalable architecture - as demands increase, extra servers can be added.

A 3-tier C/S architecture



Example- An internet banking system



Example- An internet banking system

- Here, the
 - ✓ Bank's customer database (usually hosted on a mainframe computer) provides data management services;
 - ✓ a web server provides application services such as transferring of cash, generate statements, pay bills etc. &
 - ✓ The user's own computer with an internet browser is the client.
- This system is scalable because it is relatively easy to add new web servers as the number of customers increase.

Advantages-A 3-tier C/S architecture

- The use of three-tier architecture in this case allows the information transfer between the web server and the database server to be optimized.
- Network traffic is reduced.
- More rapid response to clients.
- Efficient middleware that supports database queries in SQL is used to handle information retrieval from the database.

Multi-tier Architecture

- At times it is appropriate to extend three tier architecture to multi-tier where applications need to access and use the data from different databases
- In such case , an integration server is positioned between application servers and database servers.
- The integration server collects the distributed data & presents it to the application as if it were from single database.

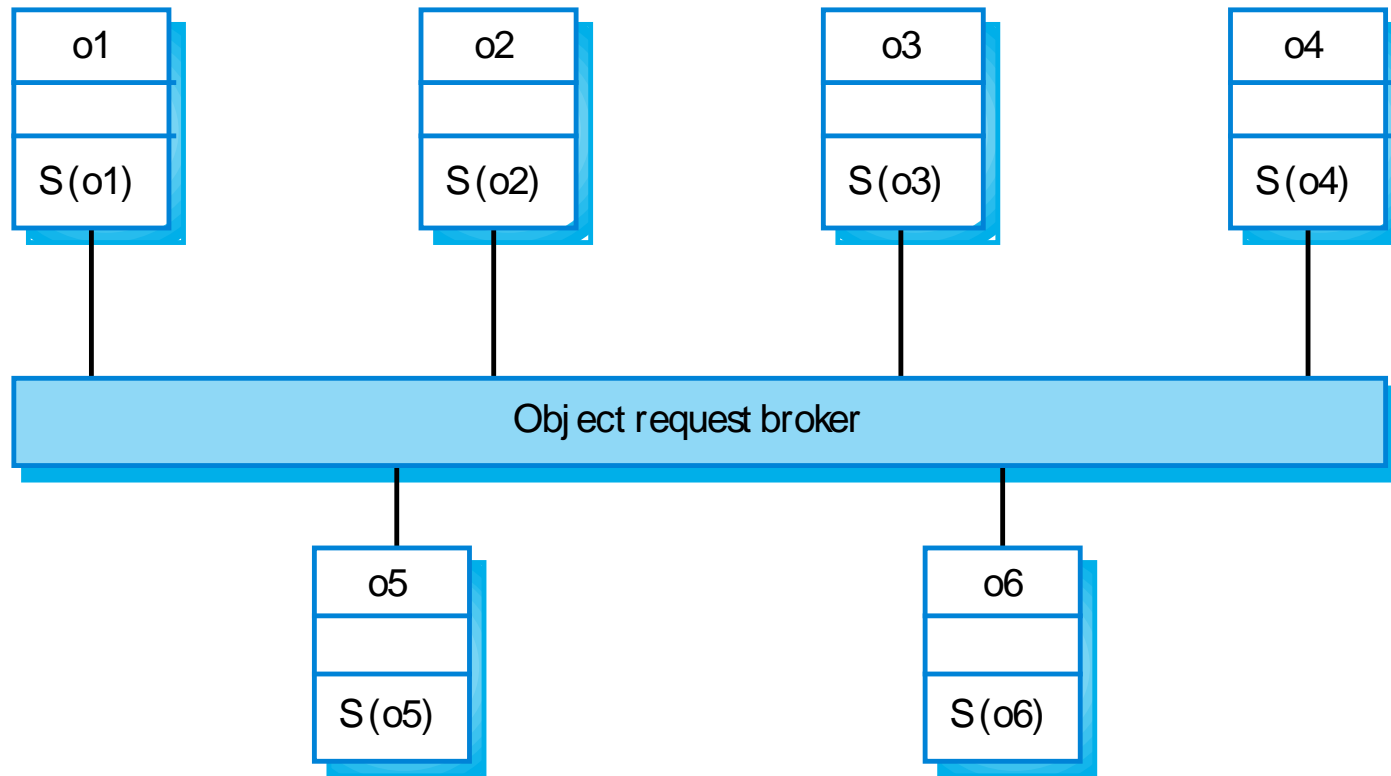
Use of C/S architectures

Architecture	Applications
Two-tier C/S architecture with thin clients	<p>Legacy system applications where separating application processing and data management is impractical.</p> <p>Computationally-intensive applications such as compilers with little or no data management.</p> <p>Data-intensive applications (browsing and querying) with little or no application processing.</p>
Two-tier C/S architecture with fat clients	<p>Applications where application processing is provided by off-the-shelf software (e.g. Microsoft Excel) on the client.</p> <p>Applications where computationally-intensive processing of data (e.g. data visualisation) is required.</p> <p>Applications with relatively stable end-user functionality used in an environment with well-established system management.</p>
Three-tier or multi-tier C/S architecture	<p>Large scale applications with hundreds or thousands of clients</p> <p>Applications where both the data and the application are volatile.</p> <p>Applications where data from multiple sources are integrated.</p>

3.8 Distributed object architectures

- There is no distinction in a distributed object architectures between clients and servers.
- Each distributable entity is an object that provides services to other objects and receives services from other objects.
- Objects may be distributed across a number of computers on a network.
- Object communication is through a middleware system called an object request broker.
- Its role is to provide a seamless interface between objects.
- It provides set of services that allows objects to communicate & to be added to & removed from the system
- However, distributed object architectures are more complex to design than C/S systems.

Distributed object architecture



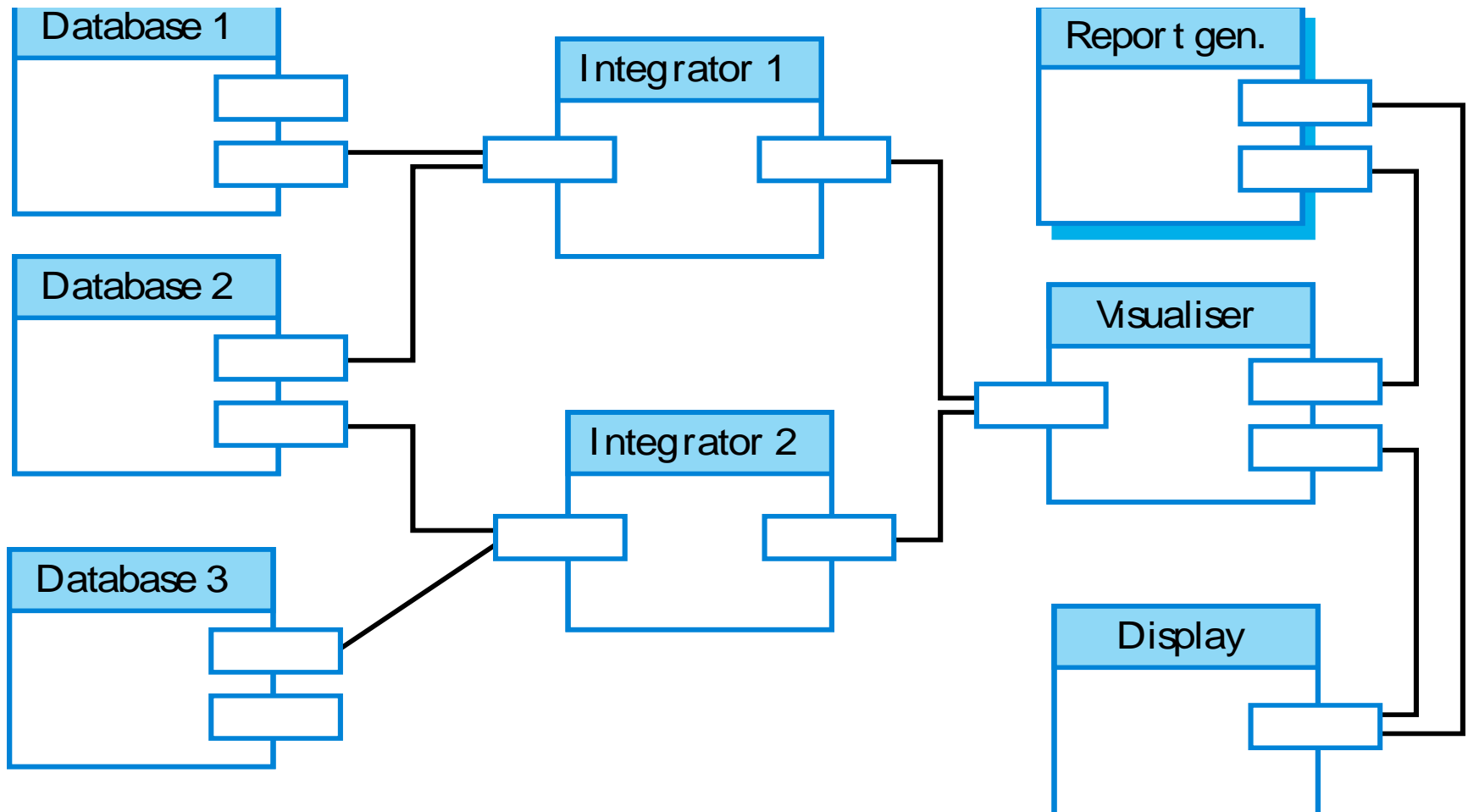
Advantages of distributed object architecture

- It allows the system designer to delay decisions on where and how services should be provided.
 - ✓ Service providing objects may execute on any node of the network.
 - ✓ There is no need to decide in advance where application logic objects are located.
- It is a very open system architecture that allows new resources to be added to it as required.
- The system is flexible and scalable.
- It is possible to reconfigure the system dynamically with objects migrating across the network as required.

Uses of distributed object architecture

- As a logical model that allows to structure and organise the system. In this case, one thinks about how to provide application functionality solely in terms of services and combinations of services.
- As a flexible approach to the implementation of client-server systems. The logical model of the system is a client-server model but both clients and servers are realised as distributed objects communicating through a common communication framework.

A data mining system



Data mining system

- Here, each database can be encapsulated as a distributed object with an interface that provides read only access to its data.
- Integrator objects are each concerned with specific types of relationships, & they collect from all the databases to try to deduce the relationships.
- There might be integrator object that is concerned with seasonal variations in goods sold & another that is concerned with relationships between different types of goods.

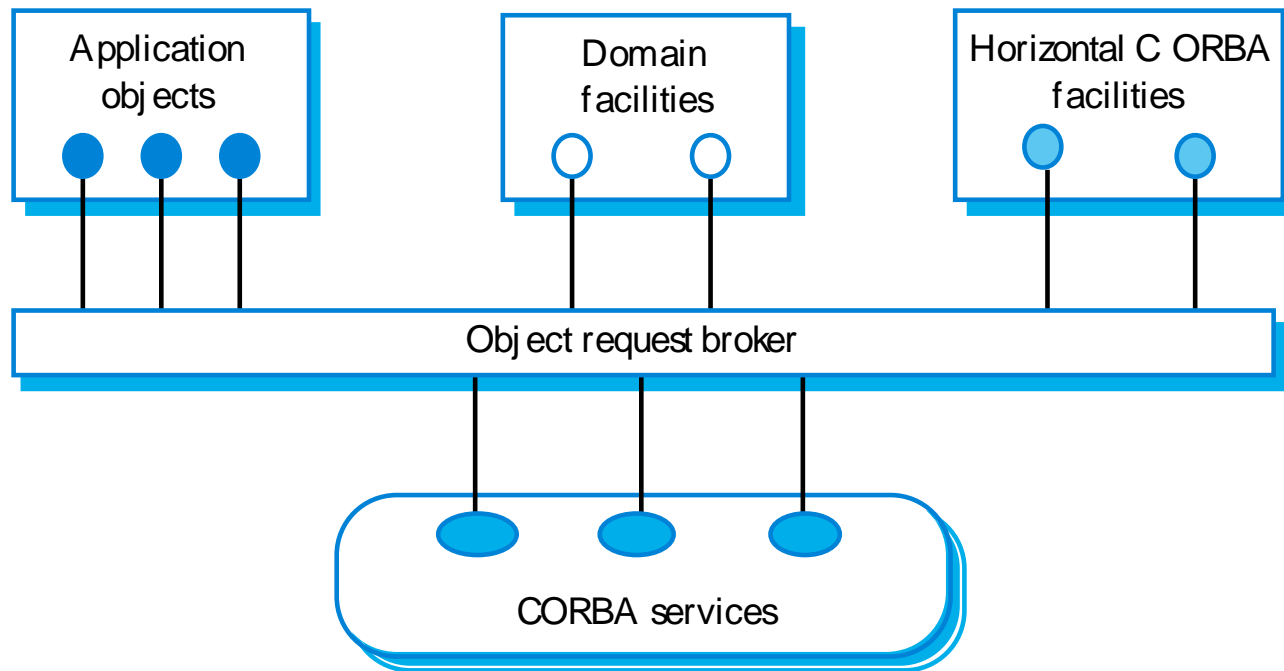
Data mining system

- The logical model of the system is not one of service provision where there are distinguished data management services.
- It allows the number of databases that are accessed to be increased without disrupting the system.
- It allows new types of relationship to be mined by adding new integrator objects.

CORBA

- CORBA is an international standard for an Object Request Broker - middleware to manage communications between distributed objects.
- Middleware for distributed computing is required at 2 levels:
 - At the logical communication level, the middleware allows objects on different computers to exchange data and control information;
 - At the component level, the middleware provides a basis for developing compatible components. CORBA component standards have been defined.

CORBA application structure- Object Management architecture(Siegal,1998)



Application structure

- This architecture above proposes distribution application should be made up of number of components
- Application objects -that are designed & implemented for the application.
- Standard objects -that are defined by the OMG(Object Management Group) for a specific domain which cover Insurance, health care etc.
- Fundamental CORBA services that provides basic distributed computing service such as directories and security management.
- Horizontal CORBA facilities such as user interface facilities.
 - ✓ The term horizontal facilities suggests that these facilities are common to many application domains

CORBA standards

- CORBA standards cover all aspect of the above vision.
- There are four major elements to these standards.
- An object model for application objects

A CORBA object is an encapsulation of state with a well-defined, language-neutral interface defined in an IDL (interface definition language).

- An object request broker that manages requests for object services.
- A set of general object services likely to be required by many distributed applications (e.g. Directory service)
- A set of common components built on top of these basic services.

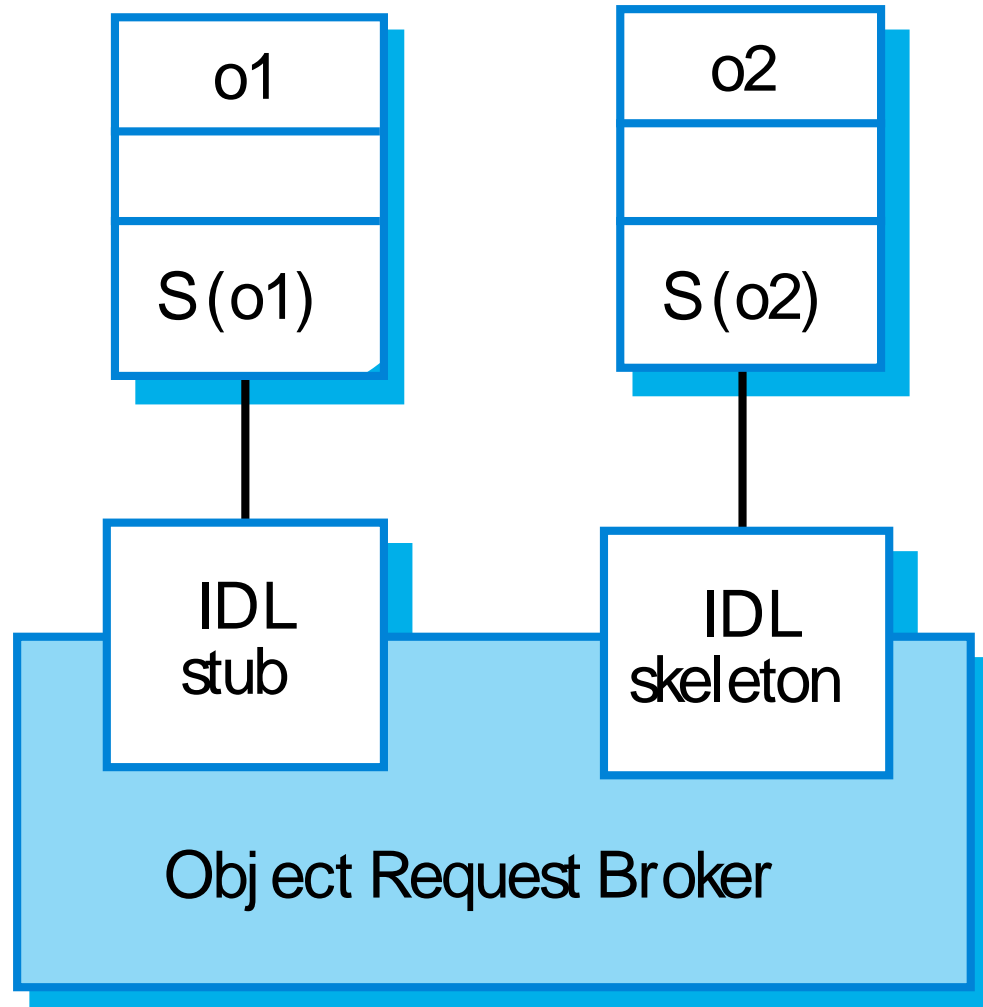
CORBA objects

- CORBA objects are comparable, in principle, to objects in C++ and Java.
- They MUST have a separate interface definition that is expressed using a common language (IDL) similar to C++.
- There is a mapping from this IDL to programming languages (C++, Java, etc.).
- Therefore, objects written in different languages can communicate with each other.

Object request broker (ORB)

- The ORB handles object communications. It knows of all objects in the system and their interfaces.
- Using an ORB, the calling object binds an IDL stub that defines the interface of the called object.
- Calling this stub results in calls to the ORB which then calls the required object through a published IDL skeleton that links the interface to the service implementation.

ORB-based object communications



Inter-ORB communications

- ORBs are not usually separate programs but are a set of objects in a library that are linked with an application when it is developed.
- ORBs handle communications between objects executing on the same machine.
- Several ORBs may be available and each computer in a distributed system will have its own ORB.
- Inter-ORB communications are used for distributed object calls.

CORBA services

- Naming and trading services
 - These allow objects to discover and refer to other objects on the network.
- Notification services
 - These allow objects to notify other objects that an event has occurred.
- Transaction services
 - These support atomic transactions and rollback on failure.
 - Transactions are fault-tolerance facility that supports recovery from errors during an update operation.
 - If an object update operation fails, then the object state can be rolled back to its state before the update was started.

3.9 Inter-organizational computing

- For security and inter-operability reasons, most distributed computing has been implemented at the organizational level.
- An organization has a number of servers & spreads its computation load across these.
- Because these all located within the same organization, local standards, management and operational processes apply.
- Newer models of distributed computing have been designed to support inter-organizational computing rather than intra-organization distributed computing where different nodes are located in different organizations.
- Two of these approaches are discussed here:
 1. Peer to Peer architectures
 2. Service oriented architectures

Peer-to-peer architectures

- Peer to peer (p2p) systems are decentralized systems where computations may be carried out by any node in the network.
- The overall system is designed to take advantage of the computational power and storage of a large number of networked computers.
- Most p2p systems have been personal systems but there is increasing business use of this technology.

P2p architectural models

One can look at architecture of P2P applications from two perspectives:

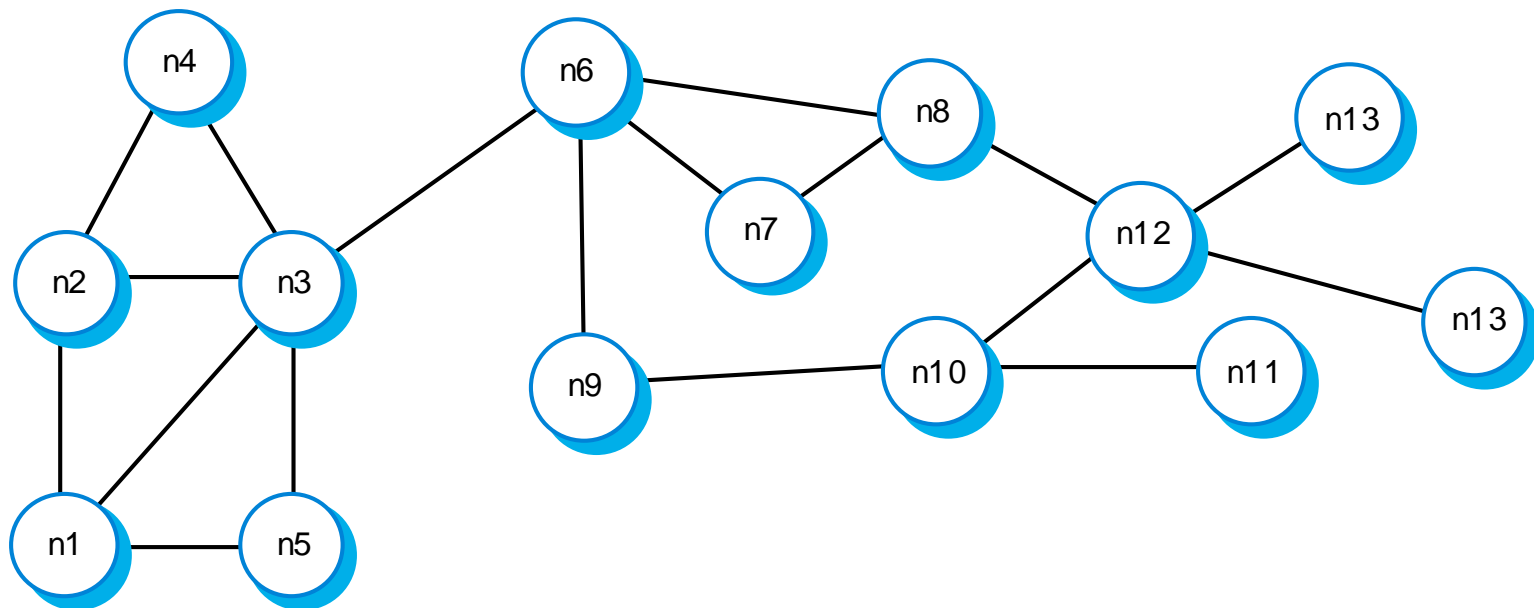
1. The logical network architecture

- Decentralized architectures;
- Semi-centralized architectures.

2. Application architecture

- The generic organization of components in each architecture type; making up a p2p application.
- Here, focused on network architectures.

Decentralized p2p architecture



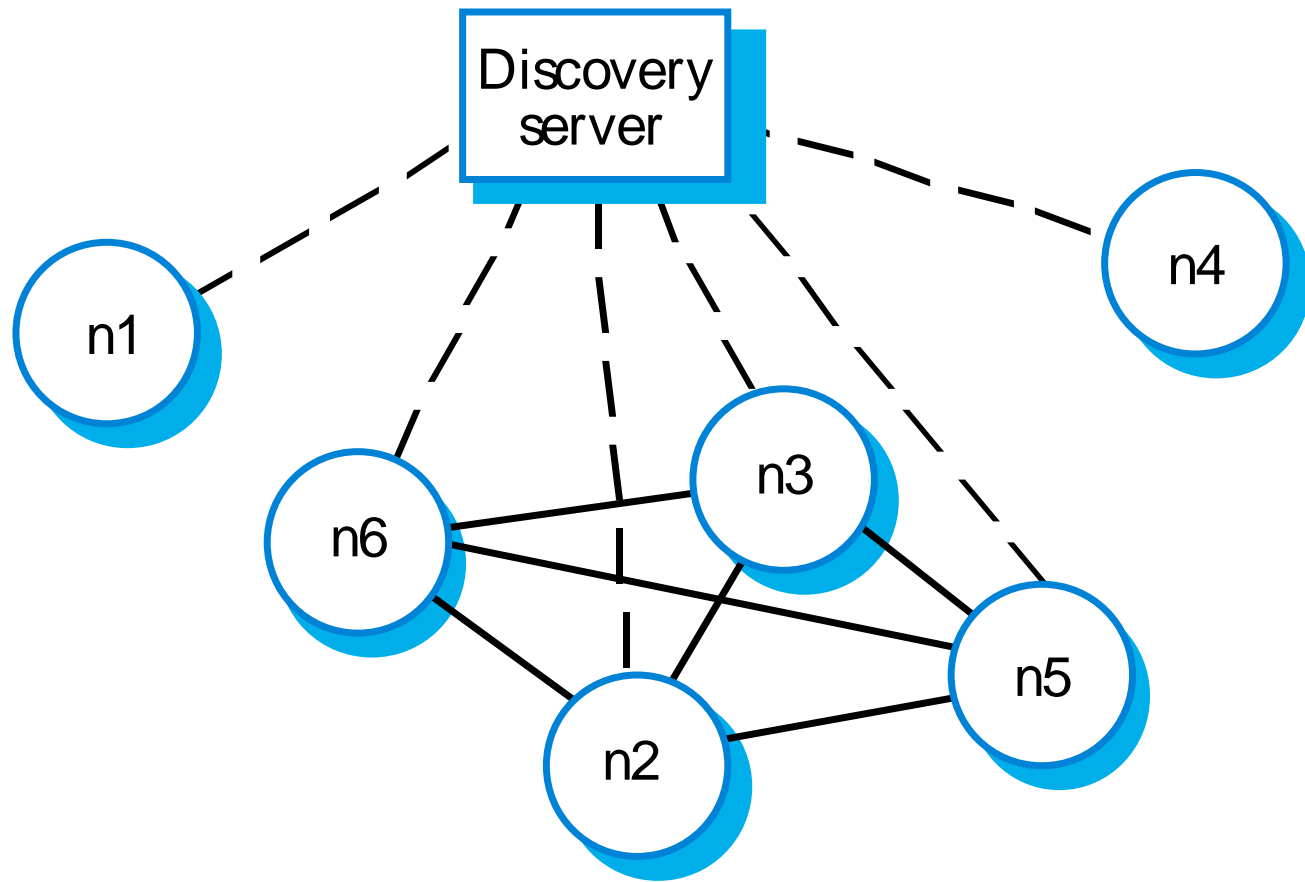
Decentralized p2p architecture

- Here, the nodes in the network are not simply functional elements but are also communication switches that can route data & control signals from one node to another
- Fig above represents a decentralized document management system-used by the consortium of researchers to share documents.
- Each member maintains own document store

Decentralized p2p architecture

- However, when the document is retrieved, the node retrieving that document makes it available to other nodes
- Someone who needs the document issues a search command that is sent to nodes in that locality .
- These nodes check whether they have the document &
- If so return to the requestor
- If they do not have route search to another node
- When the document is finally discovered , the node can route the document back to the original requestor

Semi-centralized p2p architecture



Semi-centralized p2p architecture

- With the sue of decentralized architecture the are obvious overheads in the system in that
 - ✓ same search may be process by many different nodes &
 - ✓ there is significant overhead in replicated peer communication
- Alternative- semi centralized where, within the network one or more nodes act as servers to facilitate node communications.

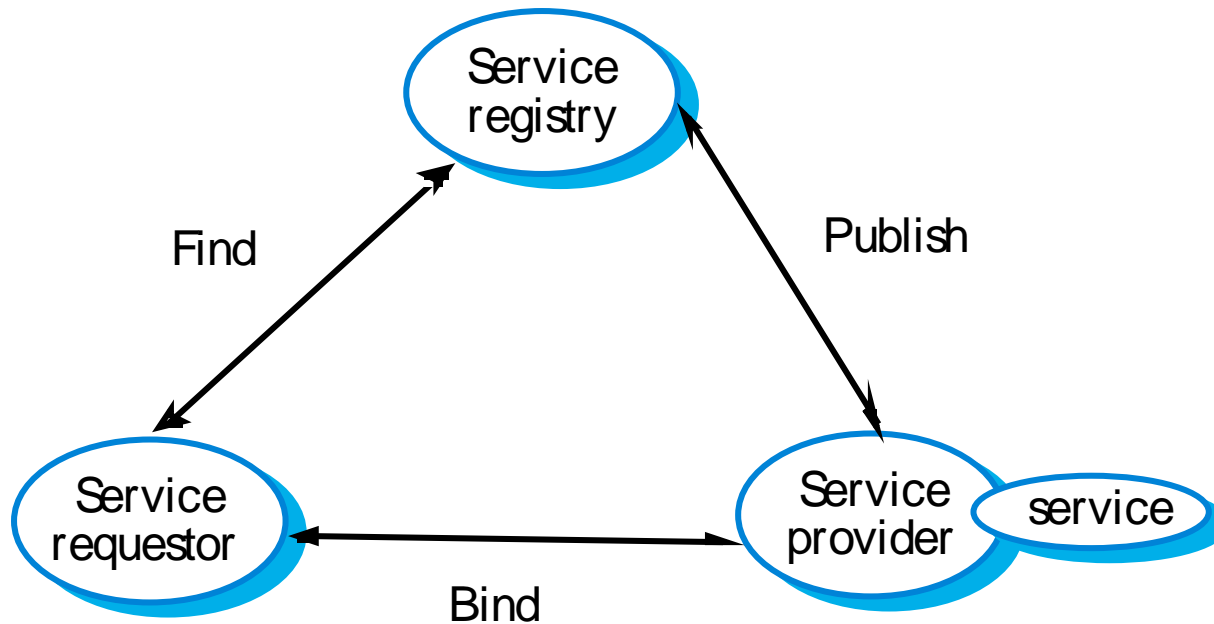
Service-oriented architectures

- Based around the notion of externally provided services (web services).
- A web service is a standard representation for some computational or information resource that are accessible across the web
 - A tax filing service could provide support for users to fill in their tax forms and submit these to the tax authorities.

Web service

- *An act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production.*
- Service provision is therefore independent of the application using the service.

Web services



Services and distributed objects

- Provider independence.
- Public advertising of service availability.
- Potentially, run-time service binding.
- Opportunistic construction of new services through composition.
- Pay for use of services.
- Smaller, more compact applications.
- Reactive and adaptive applications.

Services standards

- Services are based on agreed, XML-based standards so can be provided on any platform and written in any programming language.

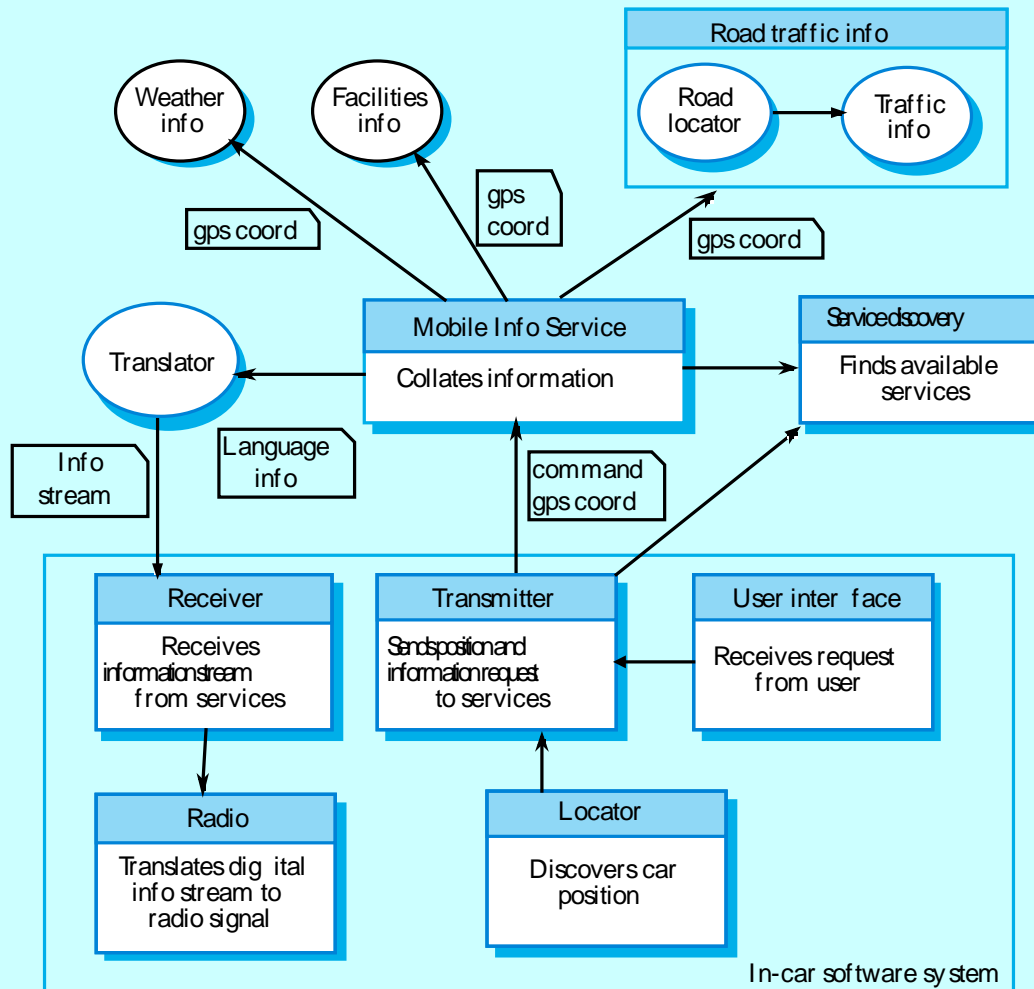
Key standards

- ✓ SOAP - Simple Object Access Protocol;
- ✓ WSDL - Web Services Description Language;
- ✓ UDDI - Universal Description, Discovery and Integration.

Services scenario

- An in-car information system provides drivers with information on weather, road traffic conditions, local information etc.
- This is linked to car radio so that information is delivered as a signal on a specific radio channel.
- The car is equipped with GPS receiver to discover its position and,
- based on that position, the system accesses a range of information services.
- Information may be delivered in the driver's specified language.

Automotive system



Layered application architecture

- Presentation layer

Concerned with presenting information(results of a computation) to the user with all the user interaction.

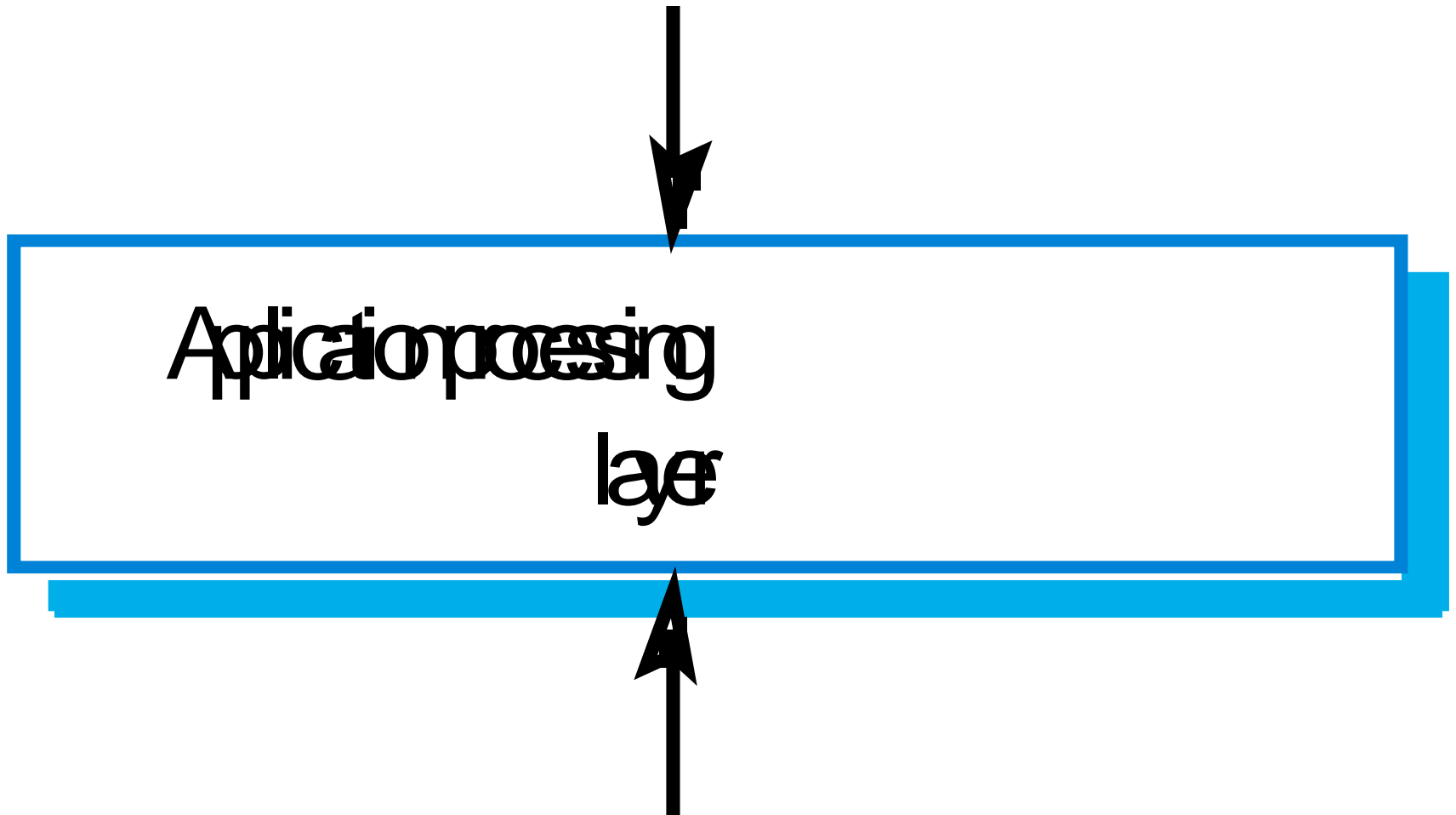
- Application processing layer

Concerned with implementing the logic of the application.

- Data management layer

Concerned with managing all the database operations.

Application layers



Thank You!!!

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering

Chapter 4

Real Time System

by

Santosh Giri

Lecturer, IOE, Pulchowk Campus.

Overview

Real Time System

Real Time Operating System

Monitoring and Control System

Data Acquisition System

Real Time System

- System where the **correct functioning of the system** depends on the **results produced by the system** and the **time** at which these results are produced.
- Systems which monitor and control their environment.
- Associated with hardware devices
- ✓ **Sensors:** Collect data from the system environment;
- ✓ **Actuators:** Change (in some way) the system's environment;
- **Time is critical** i.e. Real-time systems must respond within specified times.

Real Time System

- **Soft real-time system**

Operation is degraded, if results are not produced according to the specified timing requirements.

- **Hard real-time system:**

Operation is incorrect, if results are not produced according to the timing specification

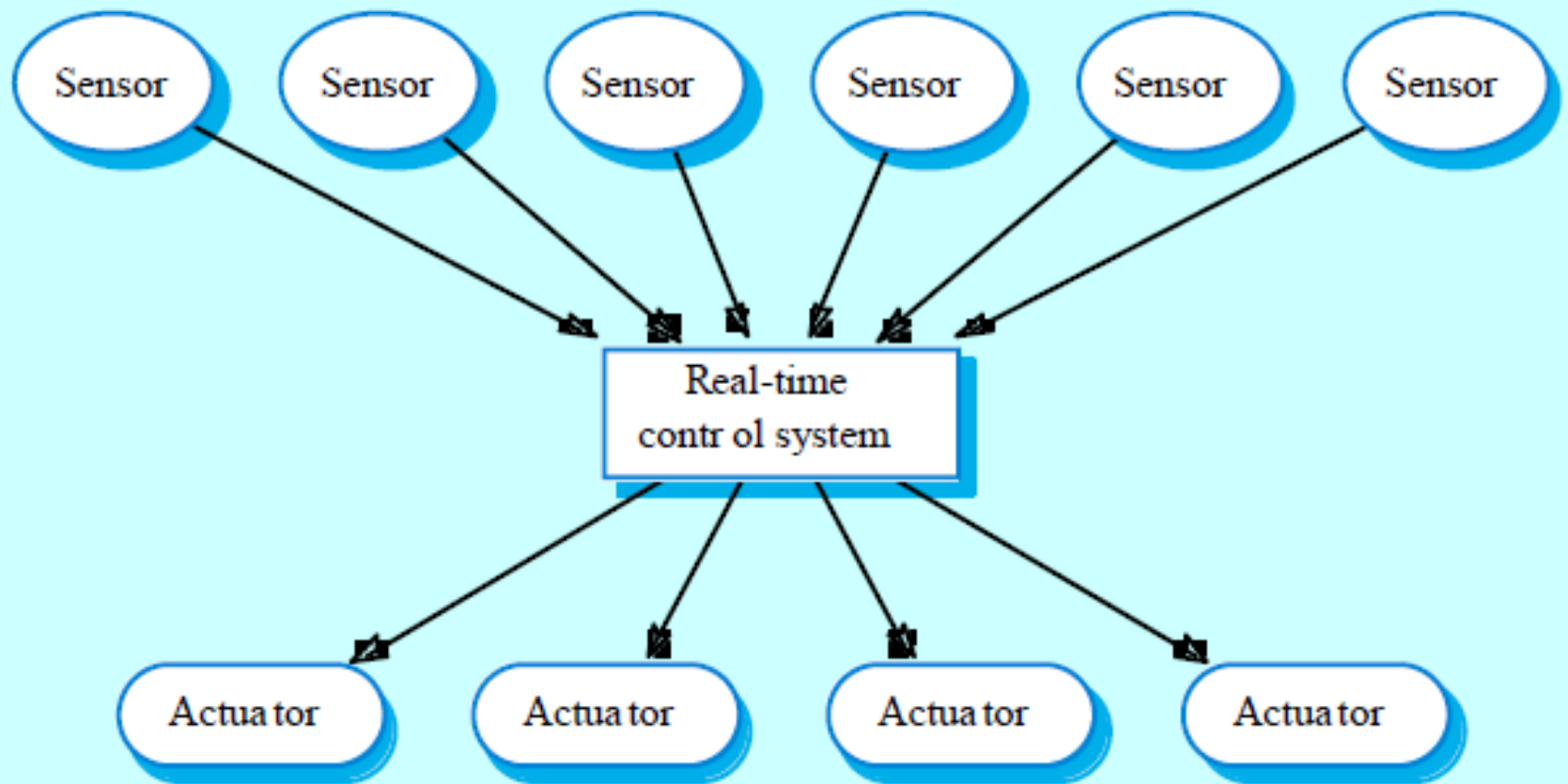
Stimulus/Response Systems

- Given a stimulus [event that evokes a specific function] , the system **must produce a response** within a specified time.
- ✓ **Periodic stimuli:** Stimuli which occur at predictable time intervals. **E.g.:** a temperature sensor may be polled 10 times per second.
- ✓ **Aperiodic stimuli:** Stimuli which occur at unpredictable times. **E.g.:** a system power failure may trigger an interrupt which must be processed by the system.

Architectural Considerations

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

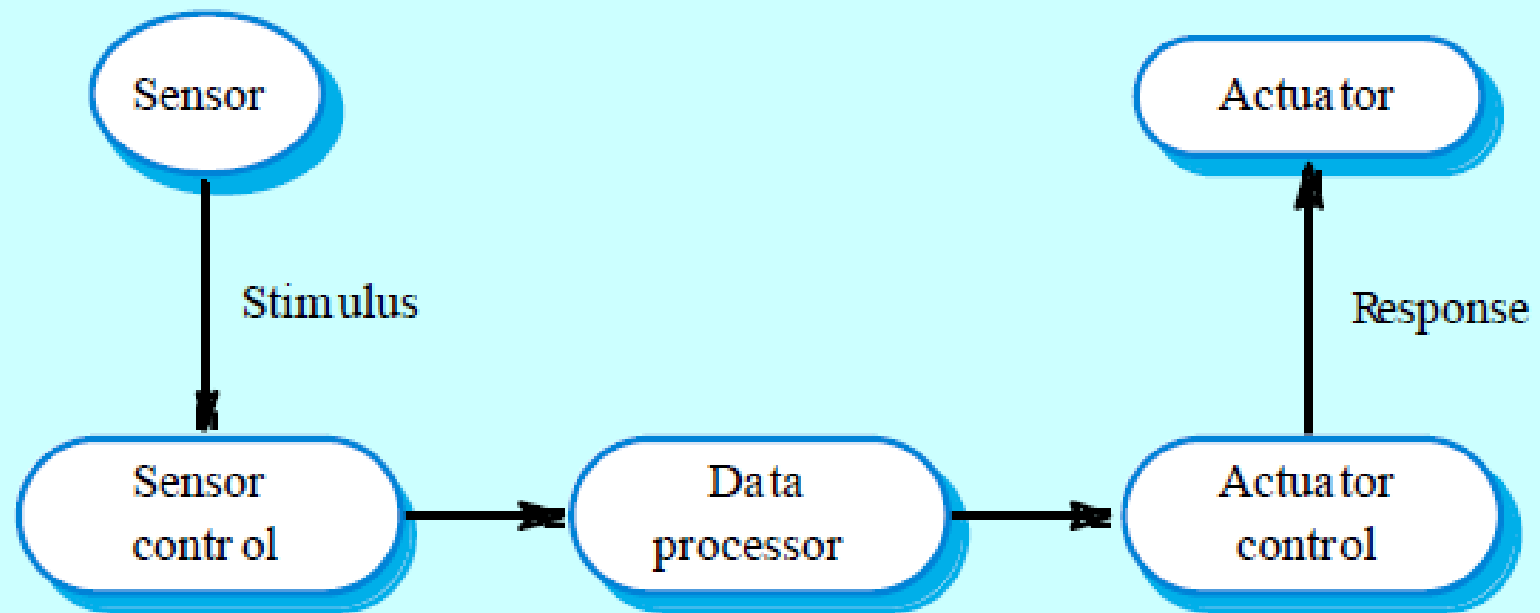
A real-time system model



Sensor/actuator processes

Microphone,
thermistor

Loudspeaker,
LED



System elements

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

Real-time operating systems

- ▶ Real-time operating systems are specialised operating systems which manage the processes in the RTS.
- ▶ Responsible for process management and resource (processor and memory) allocation.
- ▶ May be based on a standard kernel which is used unchanged or modified for a particular application.
- ▶ Do not normally include facilities such as file management.

Examples:

- VxWorks
 - QNX
 - eCos
 - RTLinux
-
- Especially VxWorks has a long history in critical applications, for example: in cars and various NASA space platforms.

Operating system components

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

1.3. Monitoring and control systems

- Important class of real-time systems.
- **Monitoring systems** examine sensors and report their results.
- **Control systems** take sensor values and control hardware actuators.

RTS design process

- Identify **stimuli** and **associated responses**.
- Define the **timing constraints** associated with each stimulus and response.
- Design **algorithms** for stimulus processing and response generation.

Monitoring System → burglar alarm systems

■ Sensors

- ✓ Movement detectors, window sensors, door sensors;
- ✓ 50 window sensors, 30 door sensors and 200 movement detectors;
- ✓ Voltage drop sensor.

■ Actions

- ✓ When an intruder is detected, police are called automatically;
- ✓ Lights are switched on in rooms with active sensors;
- ✓ An audible alarm is switched on;
- ✓ The system switches automatically to backup power when a voltage drop is detected.

Stimuli to be processed

- **Power failure**

- ✓ Generated aperiodically by a circuit monitor.
- ✓ When received, the system must switch to backup power within 50 ms.

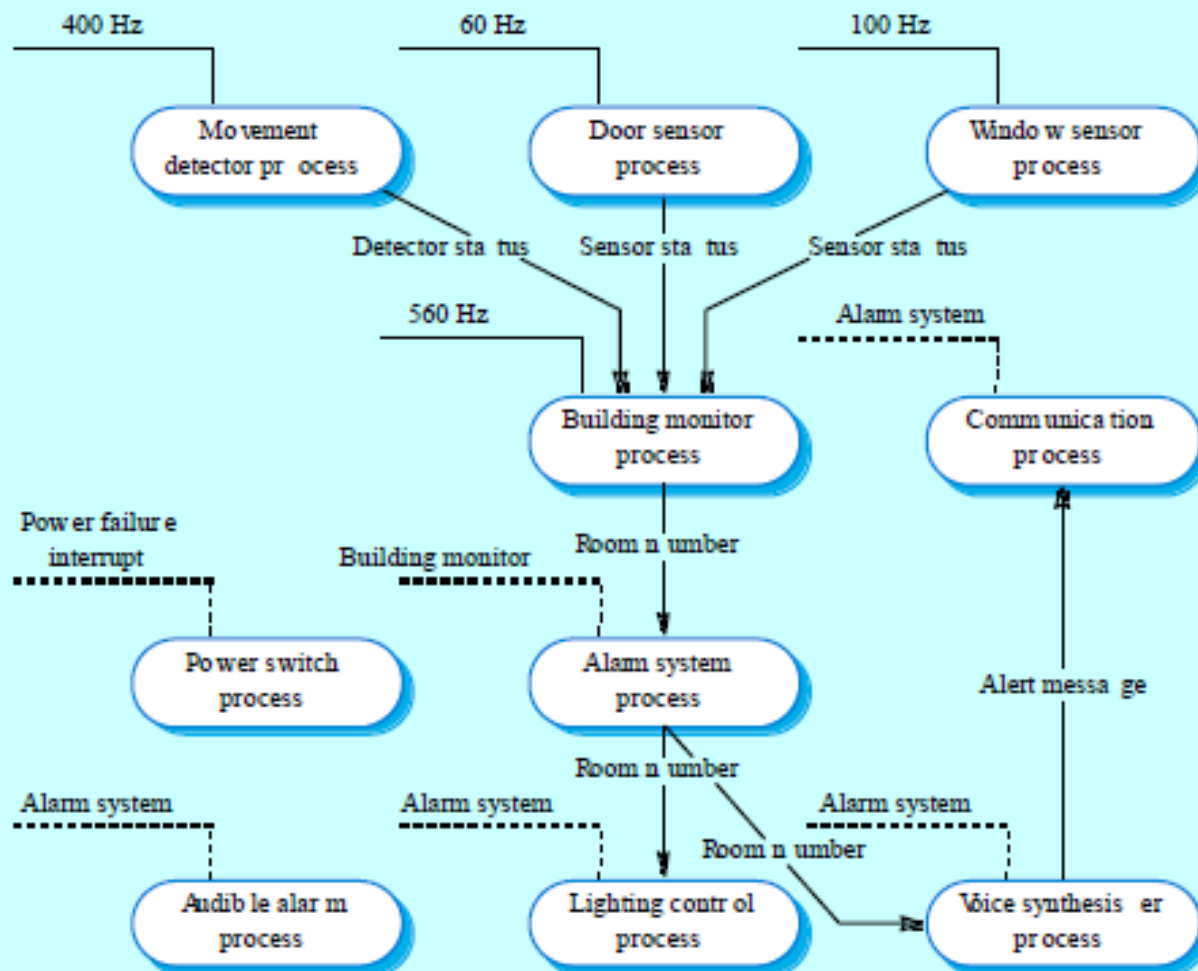
- **Intruder alarm**

- ✓ Stimulus generated by system sensors.
- ✓ Response is to call the police, switch on building lights and the audible alarm.

Timing requirements

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

Burglar alarm system processes



Control systems

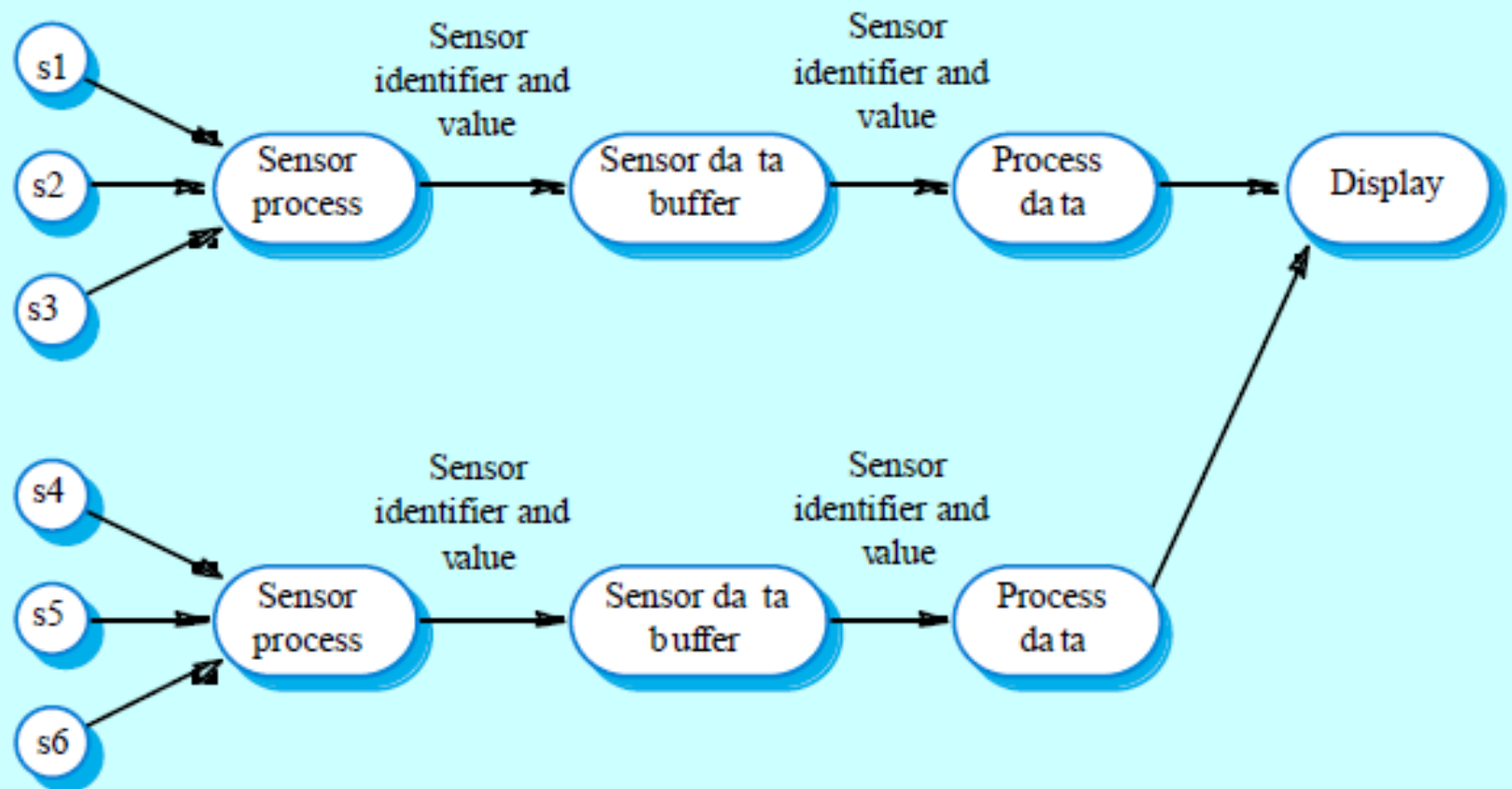
- A burglar alarm system is primarily a **monitoring system**. It **collects data from sensors** but **no real-time actuator control**.
- Control systems are **similar** but, in response to sensor values, the system **sends control signals to actuators**.
- An example of a monitoring and control system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

Data acquisition system

- Collect data from sensors for subsequent processing and analysis.
- Data collection processes and processing processes may have different periods and deadlines.
- Data collection may be faster than processing e.g. collecting information about an explosion

Data acquisition architecture

Sensors (each data flow is a sensor value)



Thank You!!!

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering

Chapter 5

Software Reuse

by

Santosh Giri

Lecturer, IOE, Pulchowk Campus.

Chapter Three: **Software reuse**

Course Outline: **3 hours**

1. The Software Reuse
2. Design patterns
3. Application framework
4. MVC patterns
5. Application system reuse



Software reuse

- Software engineering has been more focused on original development but it is now recognised that **to achieve better software, more quickly and at lower cost**, we need to adopt a design process that is based on *systematic software reuse*.



Reuse-based software engineering

■ **Application system reuse**

The whole of an application system may be reused either

- ✓ by incorporating it without change into other systems (COTS reuse) or
- ✓ by developing application families that have common architecture

■ **Component reuse**

- ✓ Components of an application from sub-systems to single objects may be reused

■ **Object and function reuse**

- ✓ Software components that implement a single well-defined object or function may be reused



Benefits of Software Reuse

being trustworthy and reliable.

Benefit	Explanation
Increased dependability	Reused software, which has been tried and tested in working systems, should be more dependable than new software. Its design and implementation faults should have been found and fixed.
Reduced process risk	The cost of existing software is already known, whereas the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as subsystems are reused.
Effective use of specialists	Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
Standards compliance	Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.
Accelerated development	Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.

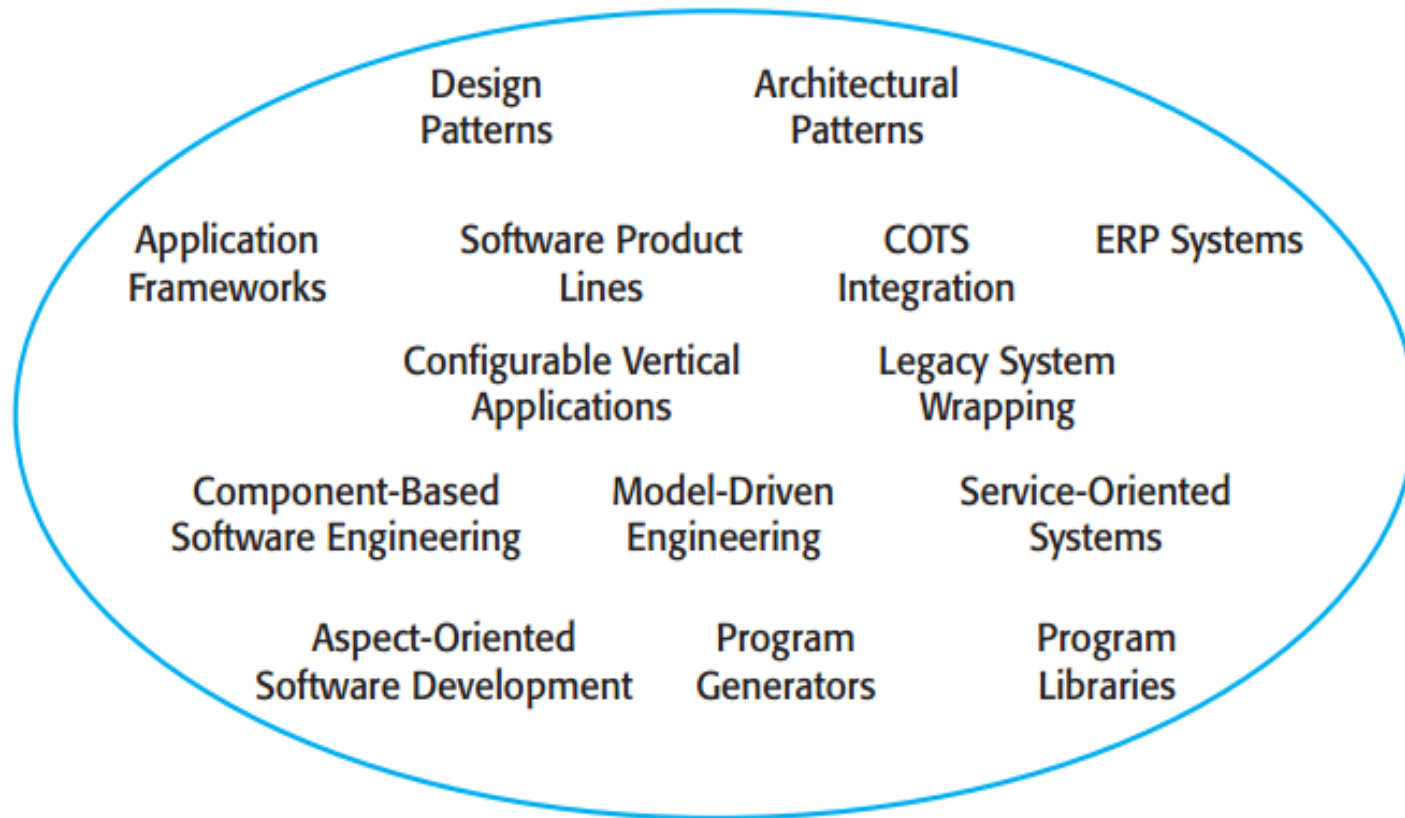


Problems of Software Reuse

Problem	Explanation
Increased maintenance costs	If the source code of a reused software system or component is not available, then maintenance costs may be higher because the reused elements of the system may become increasingly incompatible with system changes.
Lack of tool support	Some software tools do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account. This is particularly true for tools that support embedded systems engineering, less so for object-oriented development tools.
Not-invented-here syndrome	Some software engineers prefer to rewrite components because they believe they can improve on them. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.
Creating, maintaining, and using a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
Finding, understanding, and adapting reusable components	Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include a component search as part of their normal development process.



5.1. Reuse landscape





Reuse planning factors

- The development of **schedule** for the software
- The **expected software lifetime**
- The **background, skills and experience** of the development **team**
- The **criticality** of the software and its **non-functional requirements**
- The **application domain**
- The **execution platform** for the software



5.2. Design Patterns

- In software engineering, a **design pattern** is a general **repeatable solution to a commonly occurring problem** in software design.
- A design pattern **isn't a finished design** that can be transformed directly into code.
- It is a **description or template** for how to solve a problem that can be used in many different situations.



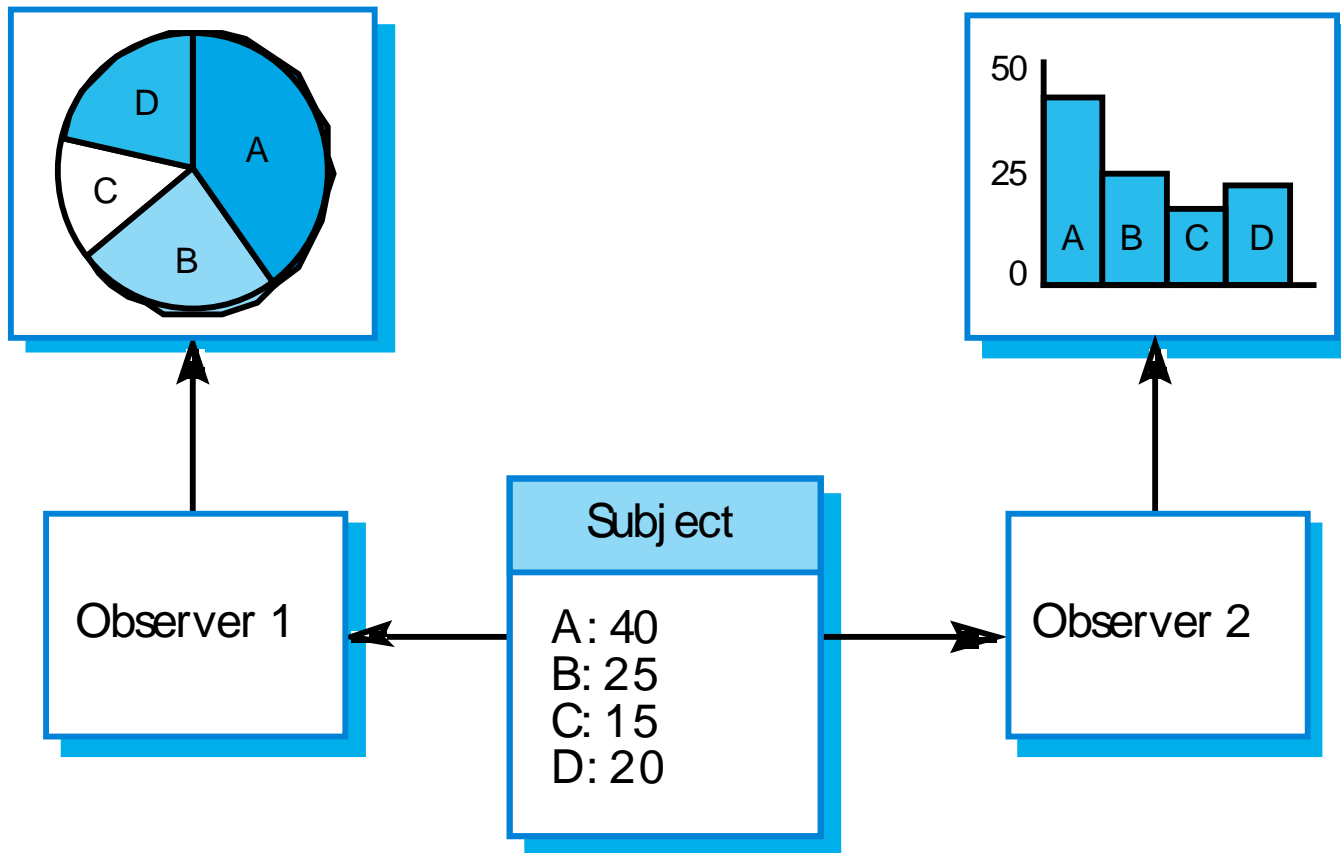
Pattern elements

There are four essential elements of the design patterns:

- **Name**
 - ✓ A name that is a meaningful reference to the pattern
- **Problem description.**
 - ✓ Description of the problem & explains what patterns may be applied.
- **Solution description.**
 - ✓ Not a concrete design but a template for a design solution that can be instantiated in different ways.
- **Consequences**
 - ✓ The results and trade-offs of applying the pattern
 - ✓ Helps the designer to understand whether a pattern can be effectively applied in particular situation

Pattern elements

Two graphical representations of same data.





5.3. Framework

- A framework is something that **gives programmers most of the basic building blocks they need to make an app.**
- Imagine you're cooking feast for 20 people. You're going to need an oven, a stove, a fridge, a sink, probably hundreds of ingredients, utensils, plates – etc.
- A framework is **like a fully stocked kitchen. It has all of these things ready for you to cook** and you just need to work out what to make with it all!
- **But**, there are a few downsides to having a ready made kitchen. Maybe the **oven isn't quite the right size**, or there **aren't quite enough plates**, or you're **lacking some ingredients**, **but** for the most part, **everything you want is in there** where you can find it and you can make it work.



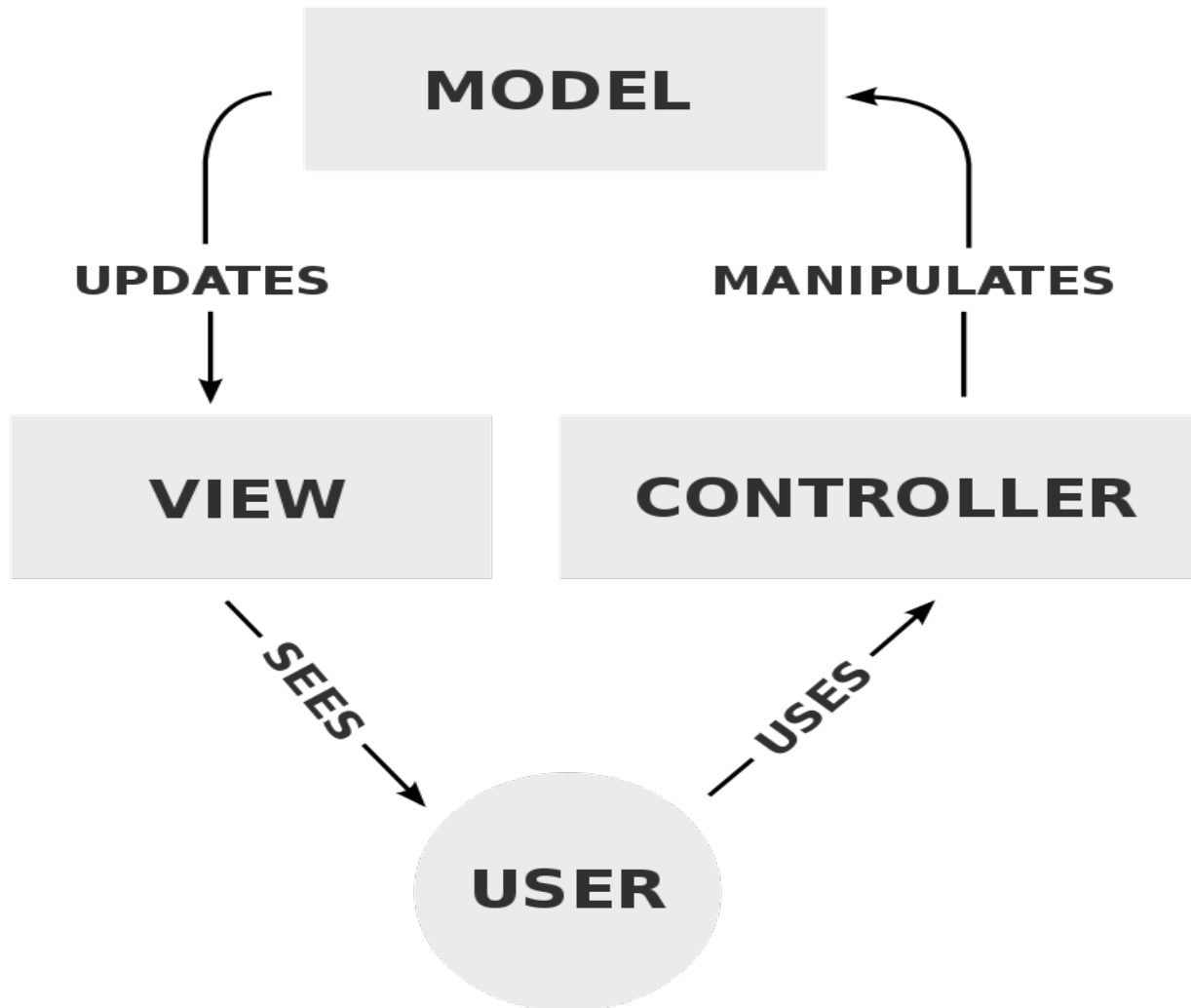
Framework

- Programming without a framework is like trying to build the perfect kitchen from scratch before preparing the meal.
- First you need to decide what you're going to make. If it needs an oven, you can decide to either buy the perfect oven, or build your own makeshift one.
- If your ingredients need refrigeration, you can work out some way to keep them cold.
- Maybe you like certain brands of ingredients? Well, you've got the freedom to buy just those brands, instead of what a pre-stocked kitchen might give you work.



- A software framework is an all inclusive, **reusable programming environment** that gives specific usefulness as a major aspect of a bigger programming stage to encourage advancement of programming applications, items and arrangements.
- Software frameworks may **incorporate bolster programs, compilers, code libraries, device sets, and application programming interfaces (APIs)** that unite all the diverse segments to empower advancement of an undertaking or arrangement.

MVC PATTERN





MVC PATTERN

The **model** manages fundamental **behaviors and data** of the application. It can **respond to requests** for information, **respond to instructions to change** the state of its information, and even to notify observers in event-driven systems when information changes. This could be a database, or any number of data structures or storage systems. In short, it is the data and data-management of the application.



Now let's say we have an **Online Banking System**, from where the user needs to check his account balance.

View:

- The **UI form** which the end user sees and sends the request from. Typically in this case it could either be the online web browsers or the mobile UI, from where the end user sends the request to check his balance.



Controller

- Now what if the user desires to do an **online fund transfer** from one account to another. In this case you would be needing a whole lot of business logic, that
 - 1.*accepts the user request,*
 - 2.*checks his balance in Account 1,*
 - 3.*deducts the funds,*
 4. transfers to Account 2,*and updates the balance in both cases.*
- What the **Controller part** here essentially does is **accept the request from user to transfer funds**, and **redirect it to the necessary components** that would do the job of transfer.



Model

- Here **it responds to requests from users** to just **read the data**(handled from the view) or **do an update of the data**(handled by the controller).
- In this case the Model, would be the part of the application that **interacts with the database here either to read or write the data.**
- So the user makes a request from the browser to check his balance amount, **the Model would be the part of the application, that receives it either from view, processes the request, and sends the data back.**

Thank You!!!

Chapter 6: CBSE

Component Based Software Engineering

Component-based software engineering (CBSE) is an approach to software development that relies on software reuse. It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific. Components are more abstract than object classes and can be considered to be stand-alone service providers.

SC's are parts of a system or application. Components are a means of breaking the complexity of software into manageable parts. Each component hides the complexity of its implementation behind an interface. Components can be swapped in and out like the interchangeable parts of a machine. This reduces the complexity of software development, maintenance, operations and support and allows the same code to be reused in many places. The following are illustrative examples of a component.

Views

User interface components for different requests, views and scenarios. For example, difficult components can be used to display the same information in a web page and mobile app.

Models

Components that handle requests or events including business rules and data processing. For example, a model might handle a bill payment request for an internet banking website.

Controllers

A controller is a component that decides what components to call for a particular request or event. For example, a controller might dynamically load different views for a bill payment based on factors such as language, transaction status or channel.

APIs

A component that can be reused across multiple systems and applications can be packaged and distributed as an API. For example, an open source API to connect to a particular database.

CBSE essentials

- **Independent components** specified by their interfaces.
- **Component standards** to facilitate component integration.
- **Middleware** that provides support for component inter-operability.
- **A development process** that is geared to reuse.

CBSE Design principles

Apart from the benefits of reuse, CBSE is based on sound software engineering design principles:

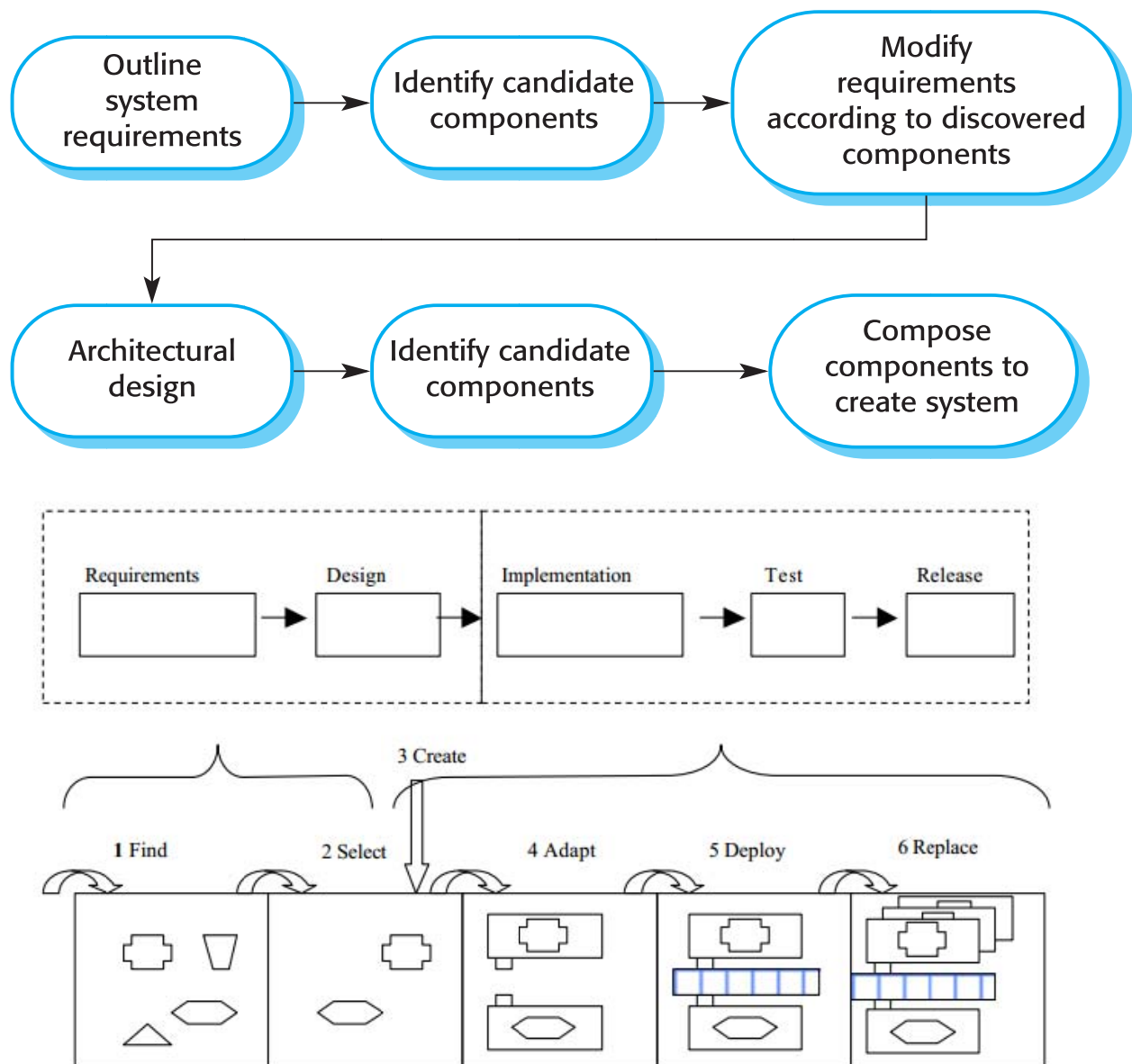
- Components are independent so do not interfere with each other;
- Component implementations are hidden;
- Communication is through well-defined interfaces;

- Component platforms are shared and reduce development costs.

CBSE Problems

- Component trustworthiness - how can a component with no available source code be trusted?
- Component certification - who will certify the quality of components?
- Emergent property prediction - how can the emergent properties of component compositions be predicted?
- Requirements trade-offs - how do we do trade-off analysis between the features of one component and another?

6.1.The CBSE Process:



The different steps in the component development process are:

1. Finding components that may be used in the product. Here all possible components are listed for further investigation.
2. Select the components that fit the requirements of the product.
3. Create a proprietary component that will be used in the product. We do not have to find these types of components since we develop them ourselves.
4. Adapt the selected components so that they suit the existing component model or requirement specification. Some component needs more wrapping than others.
5. Compose or deploy the product. This is done with a framework or infrastructure for components.
6. Replace old versions of the product with new ones. This is also called maintaining the product. There might be bugs that have been fixed or new functionality added.

Advantages:

- faster development,
- lower costs of the development,
- better usability,
- to reduce the time to market,
- To meet rapidly emerging consumer demands. Etc

Disadvantages:

- when you buy a component you do not know exactly its behavior,
- you do not have control over its maintenance,
- the implementation is quite hard,
- Process of improving reuse has been long and laborious etc.
- Security is another major concern for the developers who reuse the components available over the Internet. There may be a virus inside that component and may pass all the information of the business organization to attacker.

6.2. Components

Components provide a service without regard to where the component is executing or its programming language

- A component is an independent executable entity that can be made up of one or more executable objects;
- The component interface is published and all interactions are through the published interface;

A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. - Councill and Heinmann:

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.- Szyperski

Component as a service provider

- The component is an independent, executable entity.
- It does not have to be compiled before it is used with other components.
- The services offered by a component are made available through an interface and all component interactions take place through that interface.

Component Characteristics:

Standardised	Component standardisation means that a component that is used in a CBSE process has to conform to some standardised component model. This model may define component interfaces, component meta-data, documentation, composition and deployment.
Independent	A component should be independent – it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a ‘requires’ interface specification.
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself such as its methods and attributes.
<hr/>	
Deployable	To be deployable, a component has to be self-contained and must be able to operate as a stand-alone entity on some component platform that implements the component model. This usually means that the component is a binary component that does not have to be compiled before it is deployed.
Documented	Components have to be fully documented so that potential users of the component can decide whether or not they meet their needs. The syntax and, ideally, the semantics of all component interfaces have to be specified.

Components and objects

A component model is a definition of standards for component implementation, documentation and deployment.

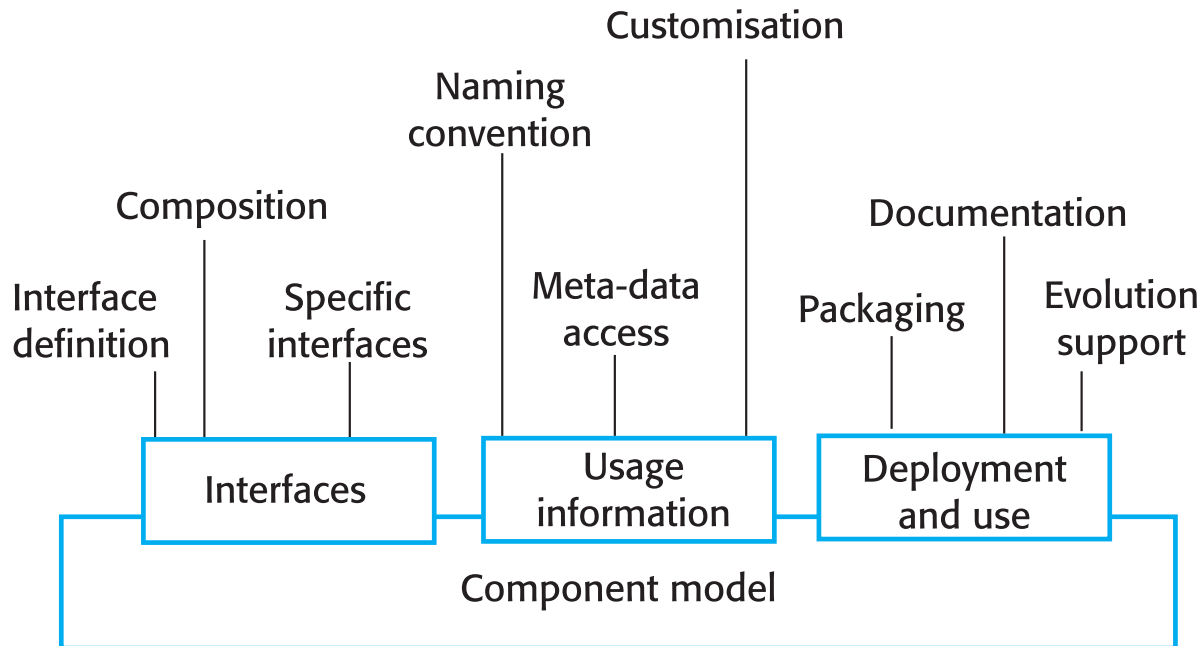
Examples of component models are

- EJB model (Enterprise Java Beans)
- COM+ model (.NET model)

- Corba Component Model

The component model specifies how interfaces should be defined and the elements that should be included in an interface definition

Elements of components model

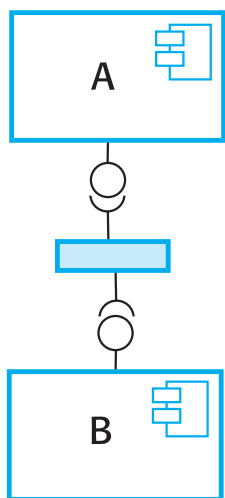


6.3. Component Composition:

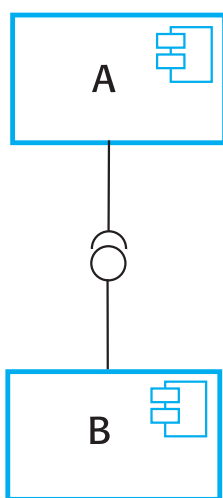
- The process of assembling components to create a system.
- Composition involves integrating components with each other and with the component infrastructure.
- Normally you have to write 'glue code' to integrate components.

Types:

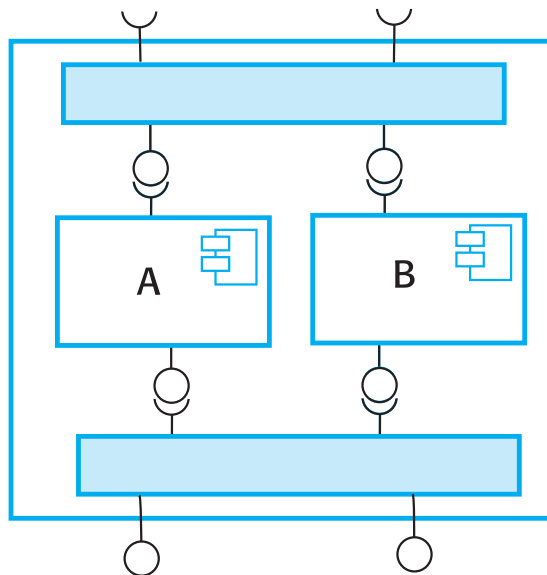
- **Sequential composition:** where the composed components are executed in sequence. This involves composing provides interfaces of each component.
- **Hierarchical composition:** where one component calls on the services of another. The provides interface of one component is composed with the requires interface of another.
- **Additive composition:** where the interfaces of two components are put together to create a new component.



(a)



(b)



(c)

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering
Chapter Seven
Verification and Validation

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

V vs V

Verification	Validation
Are we building the system right?	Are we building the right system?
Verification is the process of evaluating products of a development phase to find out whether they meet the specified requirements.	Validation is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
The objective of Verification is to make sure that the product being develop is as per design specifications .	The objective of Validation is to make sure that the product actually meet up the user's requirements .
Following activities are involved in Verification: Reviews, Meetings and Inspections .	Following activities are involved in Validation: Testing like black box testing, white box testing etc .
Verification process checks whether the outputs are according to inputs or not .	Validation process checks whether the software is accepted by the user or not .
Verification comes before the Validation	Validation comes after the Verification .



V and V process

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

It 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 and V Goal

- 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.



Software Inspections

Software Inspections refers to **peer review** of any work **product by trained individuals** who look for defects using a well defined process- Wikipedia

What are software inspections (reviews)?

Meetings during which designs and code are reviewed by people other than the original developer.



Software Inspections

- It is usually **manual and a static** technique that is applied in the **early development cycle**.
- Software inspection is regarded as the most **formal type of review**.
- It is **led by the trained moderators** and **involves peers** to examine the product.
- The **defects** found during this process are **documented in a issue log (checklist)**.



Software Inspections

Note:

Static Testing:

code is not executed. Rather it manually checks the code, requirement documents, and design documents to find errors. Hence, the name "static".

The main objective of this testing is to improve the quality of software products by finding errors in the early stages of the development cycle. This testing is also called a Non-execution technique or verification testing.

Static testing involves manual or automated reviews of the documents. This review is done during an initial phase of testing to catch Defect early in STLC. It examines work documents and provides review comments

Dynamic Testing:

A code is executed. It checks for functional behavior of software system, memory/cpu usage and overall performance of the system. Hence the name "Dynamic"

The main objective of this testing is to confirm that the software product works in conformance with the business requirements. This testing is also called an Execution technique or validation testing.

Dynamic testing executes the software and validates the output with the expected outcome. Dynamic testing is performed at all levels of testing and it can be either black or white box testing.



Inspections preconditions 1

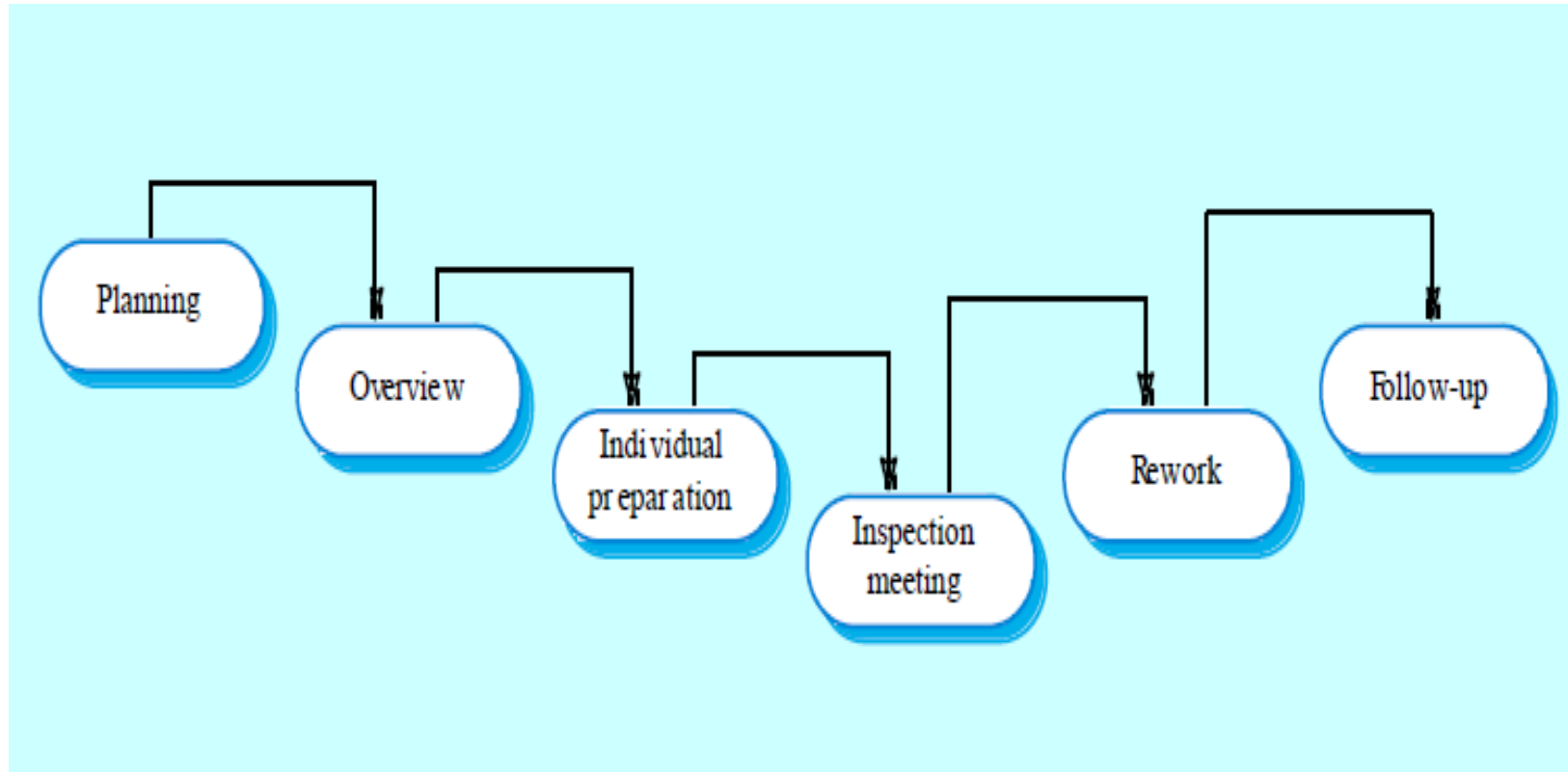
- A group of **participants** is nominated. □
- **Participants must be familiar** with inspections procedures.
- **Each participant has a well defined role,**
i.e. participant may be a moderator, an author, an inspector, a reader or a recorder.
- A **precise specification** must be available.
- Team members **must be familiar with the organization standards.**



Inspections preconditions 2

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

Inspections process





Inspections process

- 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.



Inspections 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 checklist

- 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.
- **Check-list examples:**
 - ✓ Initialisation,
 - ✓ ConstantNaming,
 - ✓ loopTermination
 - ✓ ArrayBounds, etc.

Inspection checks 1

Data faults	<p>Are all program variables initialised before their values are used?</p> <p>Have all constants been named?</p> <p>Should the upper bound of arrays be equal to the size of the array or Size -1?</p> <p>If character strings are used, is a delimiter explicitly assigned?</p> <p>Is there any possibility of buffer overflow?</p>
Control faults	<p>For each conditional statement, is the condition correct?</p> <p>Is each loop certain to terminate?</p> <p>Are compound statements correctly bracketed?</p> <p>In case statements, are all possible cases accounted for?</p> <p>If a break is required after each case in case statements, has it been included?</p>
Input/output faults	<p>Are all input variables used?</p> <p>Are all output variables assigned a value before they are output?</p> <p>Can unexpected inputs cause corruption?</p>

Inspection checks 2

Interface faults	<p>Do all function and method calls have the correct number of parameters?</p> <p>Do formal and actual parameter types match?</p> <p>Are the parameters in the right order?</p> <p>If components access shared memory, do they have the same model of the shared memory structure?</p>
Storage management faults	<p>If a linked structure is modified, have all links been correctly reassigned?</p> <p>If dynamic storage is used, has space been allocated correctly?</p> <p>Is space explicitly de-allocated after it is no longer required?</p>
Exception management faults	<p>Have all possible error conditions been taken into account?</p>



Inspection Rate

- 500 statements/hour during overview.
- 125 source statement/hour during individual preparation.
- 90-125 statements/hour can be inspected in 'I' meeting.
- Inspection is therefore an expensive process.
- Inspecting 500 lines of code with 40 man/hours effort - cost around \$3200.



Formal Methods

- Formal methods are **a particular kind of mathematically-based techniques** for the **specification, development and verification** of s/w and h/w systems.
- Major goal of software engineers:
- To develop reliable system but how?

Formal Methods:

- ✓ Mathematical languages, techniques and tools.
- ✓ Used to specify and verify systems.
- ✓ **Goal:**

Help engineers construct more reliable systems.



Formal Methods

- Formal methods can be applied at various points through the development process

✓ Specification

✓ Verification

Specification:

- give a **description** of the system to be developed and **its properties**.

Verification:

prove or disprove the **correctness of a system** with respect to the **formal specification or property**.

- The use of formal methods can **contribute to the reliability and robustness** of a design.



Formal Methods

- However the **high cost of using formal methods means that they are usually only used in** the development of **high-integrity systems**, where safety or security is very important.

Example of high-integrity systems:

Transport, communications, health and energy are all representative example of critical system where errors is not permitted.



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

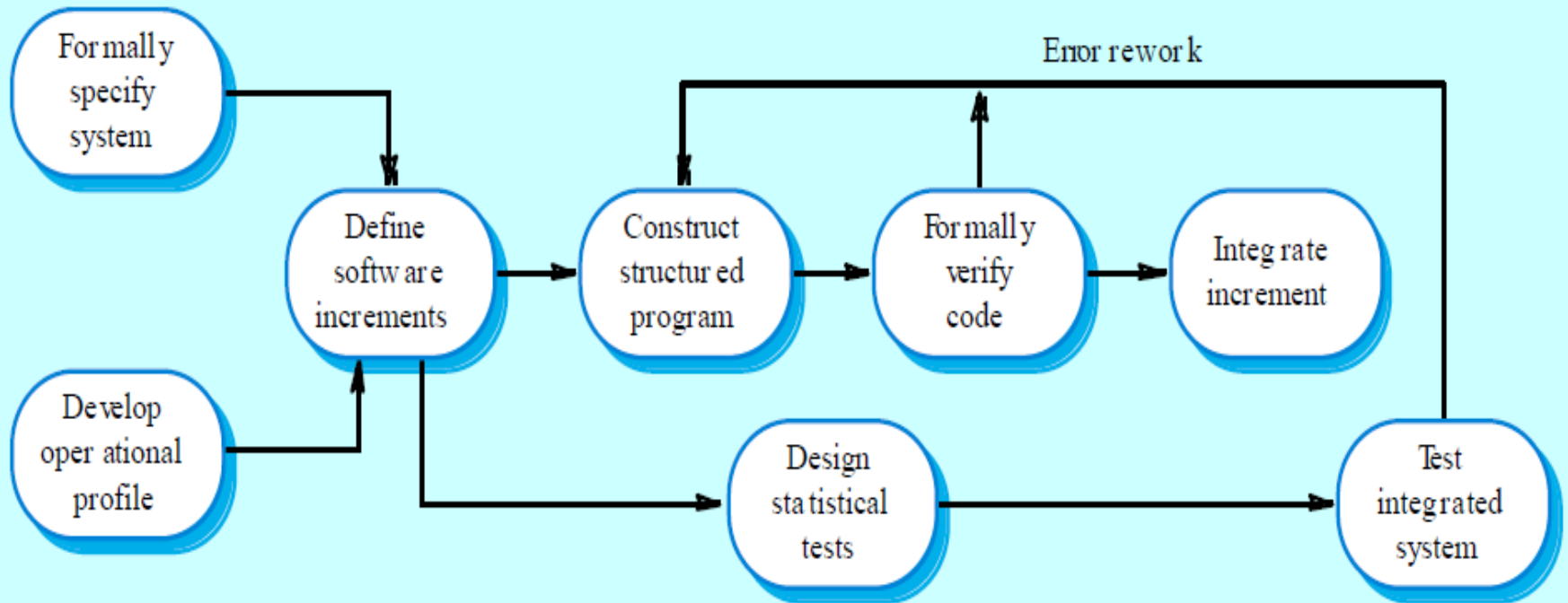
- **Require specialized notations** that cannot be understood by domain experts.
- **Very expensive** to develop a specification and even more expensive to show that a program meets that specification.
- It may be possible to reach the same level of confidence in a program **more cheaply using other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



Cleanroom Software Engineering

- The name is derived from the 'Cleanroom' process in semiconductor fabrication. The philosophy is → “defect prevention rather than defect removal”.
- Way of s/w development in which defect are avoided by using formal methods of development and rigorous (strict) inspection process.
- Objective of this approach is “zero defect s/w”.

Cleanroom Software Engineering





Cleanroom Software Engineering

Process is based on five strategic activities:

Formal specification:

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

Incremental development:

- The software is **partitioned into increments** which are **developed and validated separately** using the Cleanroom process. These increments are **specified, with customer input**, at an early stage in the process.



Structured programming:

- Only a **limited number of control and data abstraction constructs are used**. The program development process is a process of stepwise refinement of the specification.

Static verification:

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

Statistical testing of the system:

- The **integrated software increment is tested statistically, to determine its reliability**. These statistical tests are based on an operational profile which is developed in parallel with the system specification.

Thank You!!!

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering

Chapter 8

Software Testing and cost estimation

by

Santosh Giri

Lecturer, IOE, Pulchowk Campus.

Types of Testing (Exam)

1. Unit Testing

- While coding, the programmer performs some tests on that unit of program to know if it is error free.
- Testing is performed under white-box testing approach.
- Unit testing helps developers to decide that individual units of the program are working as per requirement and are error free.

2. Integration Testing

- Even if the units of software are working fine individually, there is a need to find out if the units is integrated together will also work without errors.

3. System Testing

- software is **compiled as product and then it is tested as a whole.**
- software is tested such that it works fine for different operating system.
- Performed by developers and testers.

4. Acceptance Testing:

- **Tested for user-interaction and response.** This is important because even if the software **matches all user requirements** and but user does not like the way it appears or works, it may be rejected.
- Is **performed by** independent set of testers as well as stakeholders, clients.

4. Acceptance Testing types:

4.1. Alpha Testing

- **team of developer** themselves perform testing by using the system, **as if it is being used in work environment**.
- This is performed to assess the Product in the development/testing environment by a specialized testers team usually called alpha testers. Here, **the testers feedback, suggestions help to improve the Product usage and also to fix certain bugs.**

4. Acceptance Testing:

4.2. Beta Testing

- After the software is tested internally, it is handed over to the users to use it under their production environment only for testing purpose. product is not as yet the delivered product
- Continuous feedback from the users is collected and the issues are fixed

Black box vs. white box testing (Exam)

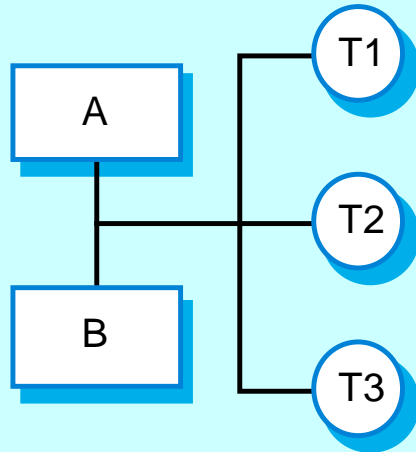
Black box testing	White box testing
Testing techniques without any knowledge of internal working of application.	Testing techniques in which tester must have knowledge of internal working of application.
Knowledge of programming is not necessary.	Knowledge of programming & internal logic of code is necessary.
Less time consuming.	More time consuming
Usually testing performed by end-users and also by testers & developers.	Normally testing is performed by testers & developers.
Focus on what is performed.	Focus on how it is performed.
Tester is unaware of internal architecture.	Tester is aware of internal architecture.

Integration testing

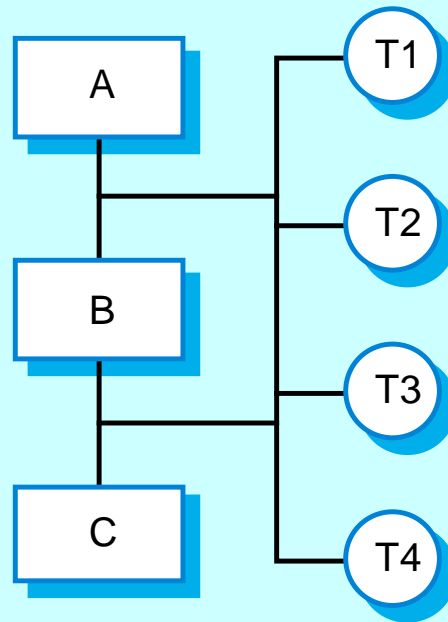
- Involves **building a system from its components and testing for problems** that arise from component interactions.
- **Top-down integration testing**
 - ✓ In this approach **testing is conducted from main module to sub module**, if the **sub module is not developed**, a temporary program called **STUB** is used to simulate the sub module.
- **Bottom-up integration testing**
 - ✓ In this approach testing is conducted **from sub module to main module**, if the **main module is not developed**, a **temporary program called DRIVERS** is used to simulate the main module.

Incremental integration testing

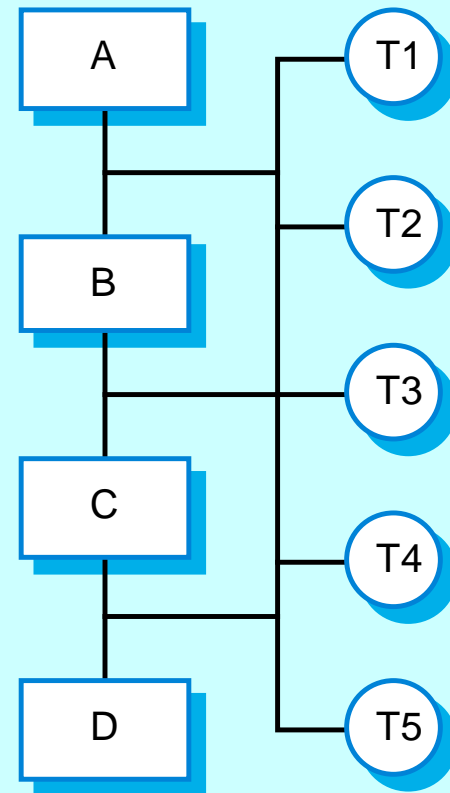
A,B,C,D-Components
T1 to T5 – test sets



Testsequence1



Testsequence2



Testsequence3

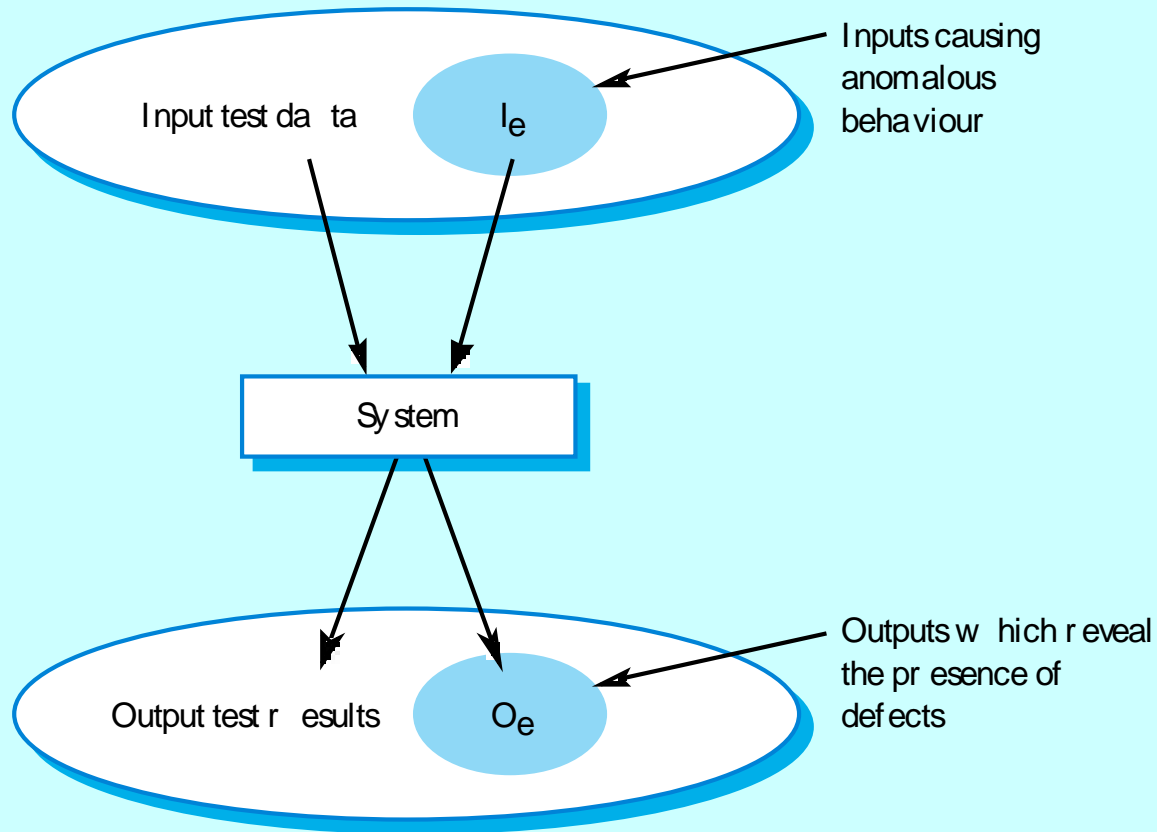
Incremental integration testing

- In the above fig. A,B,C,D are the components & T1 to T5 are the related sets of tests .
- T1,T2,T3 are first run on the system composed of component A & component B(the minimal system), If these reveal defects, they are corrected.
- Component C is integrated & T1,T2,T3 is repeated to ensure that there have not been unexpected interactions with A & B.
- Test set T4 is also run to the system & so on for D with addition of T5.

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



Black-box testing

- Fig illustrates the model of the system that is assumed in black box testing.
- Tester presents inputs to the component or the system & examines the corresponding outputs.
- If the outputs are not those predicted (i.e. if outputs are in set O_e), then the test has detected a problem with the software.
- When testing system releases, one should try to break the software, choosing test case that are in the set I_e . i.e. aim should be to select inputs that have a high probability of generating system failures.

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.

Case study for testingscenario1

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

System tests for scenario1

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 of request permission for downloading.
5. Test the e-mail response indicating that the downloaded document is available.

8.2 Component testing

- Component testing is a method where **testing of each component in an application is done separately.**
- There are different types of components may be tested at this stage:
 - **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 object class involves:

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

8.3 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.**
- Example in **Case Study example**

Thank

You!!!

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering

Chapter 8

Software Testing and cost estimation

by

Santosh Giri

Lecturer, IOE, Pulchowk Campus.



Project Planning:

- Project planning is an organized and integrated management process, which focuses on activities required for successful completion of the project.
- It helps in better utilization of resources and optimal usage of the allotted time for a project.

Objectives of project planning

- Define roles and responsibilities of the project management team members.
- Ensure that project management team works according to business objectives
- Check feasibility of schedule and user requirements.



Project Scheduling

- Project scheduling is concerned with **determining the time limit** required to complete the project.
- **An appropriate project schedule aims to complete the project on time**, and also helps in avoiding additional cost that is incurred when software is not developed on time.

Various factors that delay project schedule

Unrealistic deadline

- ✓ Project schedule is affected when the **time allocated for completing a project is impractical** and not according to the effort required for it.

Under-estimation of resources

- ✓ Under-estimation of resources leads to delay in performing tasks of the project.



Changing user requirements:

- ✓ Sometimes, project schedule is affected when **user requirements are changed after the project has started**. This affects the project schedule, and thus more time is consumed both in revision of project plan and implementation of new user requirements.

Difficulties of team members

- ✓ Software project can also be delayed due to unforeseen difficulties of the team members. For example, **some of the team members may require leave for personal reasons**.

Lack of action by project management team

- ✓ Project **management team does not recognize that the project is getting delayed**. Thus they do not take necessary action to speed up the software development process and complete it on time.



BASICS OF COST ESTIMATION

- Cost estimation is the process of approximating the costs involved in the software project.
- Cost estimation **should be done before software development** is initiated since it helps the project manager to know about resources required and the feasibility of the project.
- There are many parameters or factors, such as complexity, time availability, and reliability, which are considered during cost estimation process. However, **software size is considered as important parameters for cost estimation.**



Software Sizing:

- Before estimating cost, it is necessary to estimate the accurate size of software.
- This is a difficult task as many software are of large size. Therefore, software is divided into smaller components to estimate size.
- This is because it is easier to calculate size of smaller components, as the complexity involved in them is less than the larger components.
- There are mainly two approaches followed for estimating size:
 - ✓ **Direct approach** → size can be measured in terms of **lines of code (LOC)**
 - ✓ **Indirect approach** → size can be measured in terms of **functional point (FP)**.



1. Lines Of Code(LOC)

- LOC can be defined as the **number of delivered lines of code** in software excluding the comments and blank lines.

If comments and blank lines are excluded from the software sizing, then Why to include them?

Blank lines → Included to improve readability of code.

Comments → Included to help in code understanding as well as during maintenance.

But, these blank lines and comments do not contribute to any kind of the functionality so not considered in LOC for size estimation.



Advantages

- Very easy to count and calculate from the developer code.

Disadvantages

- LOC is language depended.
 - ✓ Same computation in python may have smaller code than C++.
- Varies from one organization of code to another organization of code. **Example:**

```
for( int i=0;i<5;i++)  
    printf(“%d”,i);
```

→ Here, lines of code =2

```
for( int i=0;i<5;i++)  
{  
    printf(“%d”,i);  
}
```

Here, lines of code =4

Same operation but differ in size of the code.



2. Function Point(FP)

- Function point metric is **used to measure effort in a project.**

- Some **features FP considered to compute the size are:**
 - ✓ **Number of external inputs (EI)**
 - **Users and other applications** act as a source of external inputs and provide distinct application oriented data or information.

 - ✓ **Number of external outputs (EO)**
 - **Each external output provided by the application**
 - External outputs refer to reports, screens, error message, and so on.



✓ **Number of external inquires (EQ):**

Used to sends data or control information outside the application.

✓ **Number of internal logical files (ILF):**

Logical grouping of data that resides within the application boundary, such as **database, Storage file, Intermediate buffer.**

These files are maintained through external inputs.

✓ **Number of external interface files (EIF):**

Logical grouping of data that resides external to the application, such as **data files on tape or disk.**



Steps in function point analysis:

- Count the number of functions of each proposed type.
- Compute the Unadjusted Function Points(UFP) as
$$\text{UFP} = \sum \{F * \text{weight}\}$$
- Compute Complexity/Value Adjustment Factor(CAF) as
$$\text{CAF} = 0.01 * \sum (F_i) + 0.65$$
- Find the Function Point Count(FPC) as
$$\text{FP} = \text{U.F.P} * \text{CAF}$$



Step1:Compute the Unadjusted Function Points(UFP):

- Categorize each of the five function types as low, average or high based on their complexity. Multiply count of each function type with its weighting factor and find the weighted sum.

$$\text{i.e. UFP} = \sum \{F * \text{weight}\}$$

- The weighting factors for each type based on their complexity are as follows:

Function Type	Weight or Factor		
	Low	Average	High
External Inputs	3	4	6
External Output	4	5	7
External Inquiries	3	4	6
Internal Logical Files	7	10	15
External Interface Files	5	7	10



Step 2 : Calculate Final FP as

$FP = U.F.P \text{ (unadjusted functional point)} * CAF(\text{complexity adjustment factor}).$

where $CAF = 0.01 * \sum(F_i) + 0.65$

$i=1$ to 14 i.e. total number of questions where each question have answer with scale value 0 to 5 .

$\sum F_i \rightarrow$ sum of all scale value $[0-5]$ from 1 to 14 .



The value adjustment factors are based on the response to these 14 questions, which are listed below:

1. Is reliable backup and recovery required by the system?
2. Is data communication required to transfer the information?
3. Do distributed processing functions exist?
4. Is performance vital?
5. Does the system run under immensely utilised operational environment?
6. Is on-line data entry required by system?
7. Is it possible for the on-line data entry (that requires the input transaction) to be built over multiple screens or operations?
8. Is updation of internal logical files allowed on-line?
9. Are the inputs, outputs, files, or inquires complex?
10. Is the internal processing complex?
11. Is the code reusable?
12. Does design include conversion and installation?
13. Does system design allow multiple installations in different organizations?
14. Is the application easy to use and does it facilitate changes?



Example:

Given the following parameters, compute FP. Complexity adjustment factors and weighting factors are average.

user i/p =50

user o/p=40

user enquiries =35

user files =6

external interface =4

$$\text{Unadjusted FP} = 50 * 4 + 40 * 5 + 35 * 4 + 6 * 10 + 4 * 7 = 628$$

$$\text{Complexity AF} = 0.01 * (14 * \text{Average} \rightarrow 3) + 0.65 = 1.07$$

$$\text{Function of Point} = \text{UFP} * \text{CAF} = 628 * 1.07 = 671.96$$



Advantages

- Independent of Language and technical tool
- Directly estimated from requirements before design and coding.

Disadvantages

- More complex calculation than LOC.



Cost Estimation Models

Algorithmic models:

- Estimation in these models is performed with the help of **mathematical equations**, which are based on historical data or theory.
- In order to estimate cost accurately, various **inputs are provided to these algorithmic models**.
- These **inputs include software size and other parameters**.
- The various algorithmic models used are COCOMO, COCOMO II, and software equation.



Non-algorithmic models:

- Estimation in these models depends on the prior experience and domain knowledge of project managers.
- Note that these models do not use mathematical equations to estimate cost of software project.
- The various non-algorithmic cost estimation models are expert judgment, estimation by analogy, and price to win etc.



Constructive Cost Model(COCOMO)

- COCOMO is one of the most widely used software estimation models in the world.
- In this model, size is measured in terms of **thousand of delivered lines of code (KDLOC)**.
- In order to estimate effort accurately, COCOMO model divides projects into three categories :

1. Organic projects:

- ✓ These projects are **small in size (not more than 50 KDLOC)**.
- ✓ Example of organic project are, business system, inventory management system, payroll management system, and library management system.



2. Semi-detached projects:

- ✓ The size of semi-detached project is **not more than 300 KDLOC**.
- ✓ Examples of semi-detached projects include **operating system, compiler design, and database design**.

3. Embedded projects:

- ✓ These projects are **complex in nature (size is more than 300 KDLOC)**.
- ✓ Example of embedded projects are software system used in **avionics and military hardware**.



Constructive cost model is based on the hierarchy of three models, **basic model, intermediate model, and advance model.**

1. Basic Model:

- In basic model, only the size of project is considered while calculating effort.
- To calculate effort, use the following equation (known as effort equation):

$$E = A \times (\text{size})^B \dots\dots\dots(i)$$

where E is the effort in person-months and size is measured in terms of KDLOC.



The values of constants 'A' and 'B' depend on the type of the software project. In this model, values of constants ('A' and 'B') for three different types of projects are listed in Table.

Project Type	A	B
Organic project	3.2	1.05
Semi-detached project	3.0	1.12
Embedded project	2.8	1.20

Example:

if the project is an organic project having a size of 30 KDLOC, then effort is calculated using equation,

$$E = 3.2 \times (30)^{1.05}$$

$$E = 114 \text{ Person-Month}$$



2. Intermediate Model:

- In intermediate model, parameters like **software reliability and software complexity** are also considered along with the size, while estimating effort.
- To estimate total effort in this model, a number of steps are followed, which are listed below:
 - ✓ Calculate an **initial estimate** of development effort by considering the **size in terms of KDLOC**.
 - ✓ **Identify a set of 15 parameters**, which are derived from attributes of the current project. All these parameters are rated against a numeric value, called **multiplying factor**.
 - ✓ Effort adjustment factor (EAF) is derived by multiplying all the multiplying factors with each other.



The COCOMO II Effort Equation:

$$\text{Effort(Person-Month)} = 2.94(\text{Initial calibration}) * \text{EAF} * (\text{KDLOC})^E$$

Where,

EAF: Effort Adjustment Factor derived from the 15 Cost Drivers or multiplying factors (make assumption if value are not given in exam)

E: Exponent derived from the five Scale Drivers (make Assumption if value not given)

Example:

A project with all **Nominal Cost Drivers** and **Scale Drivers** would have an “**EAF**” of **1.00** and **exponent “E”** of **1.0997**. Assuming that the project is projected to consist of **8,000** source lines of code.

Then by Using COCOMO II estimation

$$\text{Effort} = 2.94 * (1.0) * (8)^{1.0997} = 28.9 \text{ Person-Months}$$



In the same example if effort multipliers are given then

If your project is rated **Very High for Complexity** (effort multiplier of **1.34**), and **Low for Language & Tools Experience** (effort multiplier of **1.09**), and **all of the other cost drivers are rated to be Nominal** (effort multiplier of **1.00**) then,

$$\begin{aligned} \text{Effort Adjustment Factor (EAF)} &= 1.34 * 1.09 * 1 = 1.46 \\ \text{Effort} &= 2.94 * (1.46) * (8)^{1.0997} = 42.3 \text{ Person-Months} \end{aligned}$$



COCOMO II Schedule Equation:

The COCOMO II schedule equation predicts the number of months required to complete your software project. It is predicted as:

$$\text{Duration} = 3.67(\text{Initial calibration}) * (\text{Effort})^{\text{SE}}$$

Where,

Effort: Effort from the COCOMO II effort equation.

SE: Schedule equation exponent derived from the five Scale Drivers

Example:

Continuing previous example and assuming the schedule equation exponent of 0.3179 that is calculated from the five scale drivers.

$$\text{Duration} = 3.67 * (42.3)^{0.3179} = 12.1 \text{ months}$$

$$\text{Average staffing} = \text{Effort} / \text{Duration}$$

$$= (42.3 \text{ Person-Months}) / (12.1 \text{ Months}) = 3.5 \text{ people}$$



Advantages	Disadvantages
<ul style="list-style-type: none">▪ Easy to verify the working involved in it.▪ Cost drivers are useful in effort estimation as they help in understanding impact of different parameters involved in cost estimation.▪ Efficient and good for sensitivity analysis.▪ Can be easily adjusted according to the organization needs and environment.	<ul style="list-style-type: none">▪ Difficult to accurately estimate size, in the early phases of the project.▪ Vulnerable to misclassification of the project type.▪ Success depends on calibration of the model according to the needs of the organization. This is done using historic data, which is not always available.▪ Excludes overhead cost, travel cost and other incidental cost.

Thank

You!!!

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering

Chapter 9

Software Quality Management

by

Santosh Giri

Lecturer, IOE, Pulchowk Campus.

Software Quality?

- ° It is the degree of conformance to **explicit** or **implicit requirements** and **expectations**.

Explicit: clearly defined and documented.

Implicit: not clearly defined and documented but indirectly suggested.

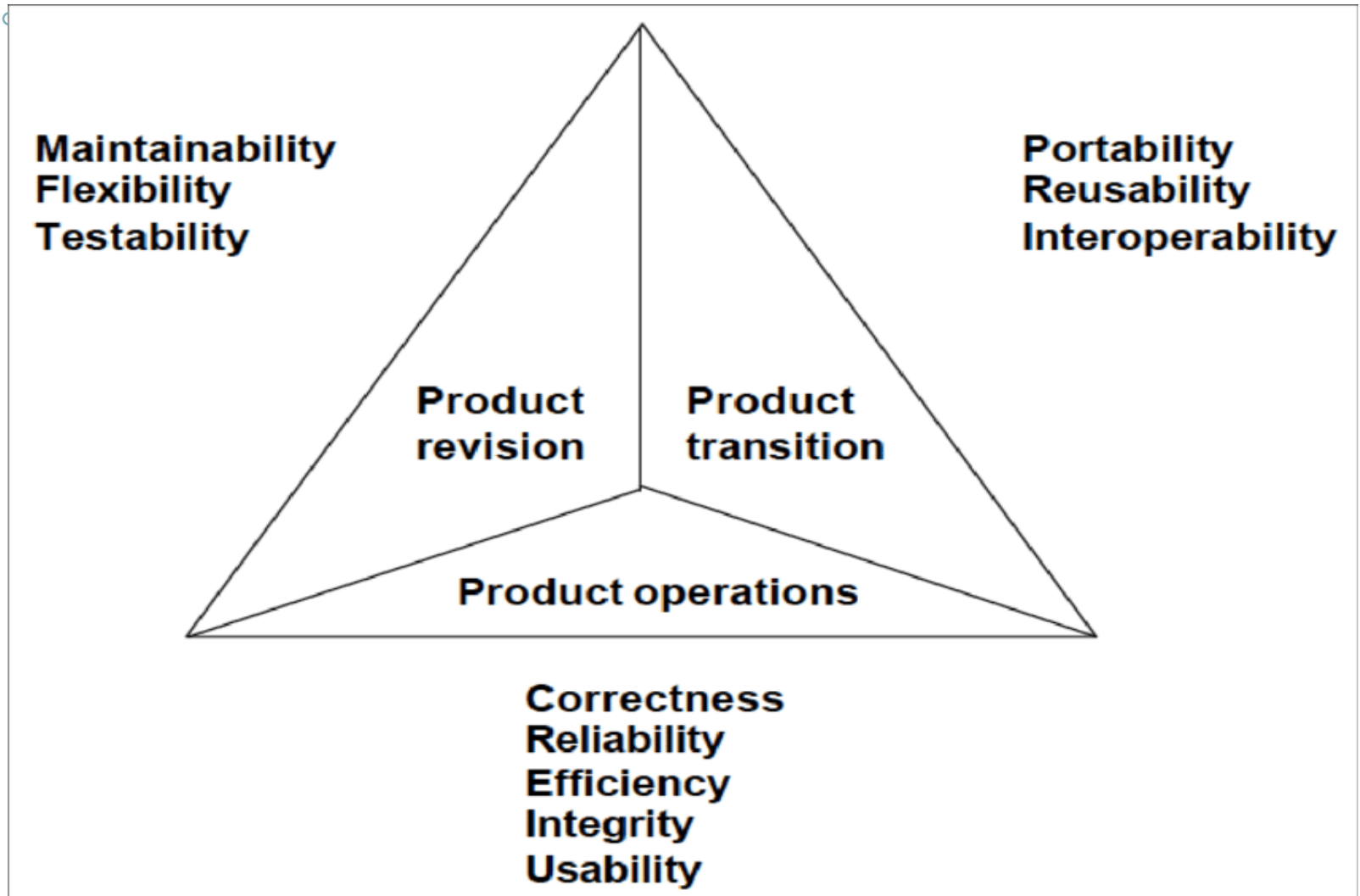
Requirements: business/product/software requirements.

Expectations: mainly end-user expectations.

Short and simple:

- ✓ The lack of bugs
- ✓ Low defect rate
- ✓ High reliability (no. of failures per hours)

McCall's Quality Factors → already discussed



Software Quality Assurance?

- ▪ Software Quality Assurance is a **planned and systematic** way of creating an environment **to assure** that the **software** product being developed **meets the quality requirements**.
- Refers to the **implementation of well-defined standards and methods** (such as ISO-9000 or CMMI model).
- This process is **controlled and determined at managerial level**.

SQA Activities & Tasks

1. Prepare a SQA plan for a product

- ✓ The plan is developed during project planning and reviewed by all interested parties.
- ✓ SQA activities, performed by the s/w engineering team & SQA team are governed by the plan.

2. Participate in the development of the project's s/w process descriptions

- ✓ The s/w engineering team select a process for work to be performed.
- ✓ The SQA groups reviews the process descriptions for the agreements with the organizations policy, standards etc.

SQA Activities & Tasks

° 3. Review s/w engineering activities to verify compliance (agreement) with the defined s/w process

✓The SQA group identifies, documents and tracks deviations from the process and verify that and corrections have been made.

4. Audits designed s/w work product to verify compliance with those defined as part of the s/w process

✓The SQA groups reviews selected work products, identifies documents, tracks deviations and verify that correctness have been made and periodically reports the results of its work to the project manager.

SQA Activities & Tasks

° **5.Ensure that deviation in s/w work and work product are documented and handled according to the documented procedure**

✓ Deviation occurred in s/w product development process has been documented according to documentation procedure.

6.Periodically reports any non-compliance(agreement) and reports to the senior management

✓ Non agreement items are tracked and reports to the senior management.

Formal Technical Reviews

- FTR is the software quality control activities performed by software engineers.
- It is conducted as a meeting and will be useful only if it is properly planned, controlled & attended.

Features of FTR

1. The review meeting

- ✓ Advance preparations should be made.
- ✓ Between 3 to 5 people are involved.
- ✓ Duration should be less than 2 hrs.

The review meeting will be generally attended by review leader, producers & reviewers and decides whether to:

- ✓ Accept the project.
- ✓ Reject the project due to errors.
- ✓ Accept the product conditionally.

FTR Features continue...

2. Review reporting & record keeping

- ✓ During FTR a reviewer actively records all issues that have been raised during FTR.
- ✓ At the end of the meeting the review issue list is produced.

It answers:

- ✓ What was reviewed?
- ✓ Who reviewed it?
- ✓ What were the finding & conclusions?

3. Review guidelines

- ✓ Guidelines for the conduct of FTR is established in advance and distributed to all the members of FTR groups.

Reviews Guidelines

- ✓ Review the product, not the producer.
- ✓ Set an agenda and maintain it.
- ✓ Limit debate and rebuttal {disproval}.
- ✓ identify problem areas, but don't attempt to solve every problem noted.
- ✓ Take written notes.
- ✓ Limit the number of participants and insist upon advance preparation.
- ✓ Develop a checklist for each product that is likely to be reviewed.
- ✓ Allocate resources and schedule time for FTRs.
- ✓ Conduct meaningful training for all reviewers.
- ✓ Review your early reviews.

Formal approach to SQA

- ▪ Over past two decades, a small but the vocal segment of the software engineering community has argued that a more formal approaches to SQA is required.
- Assuming that computer program is a mathematical object. Rigorous syntax & semantics can be defined for every language & A rigorous{strict} approach to the specification of the software is available.
- And if the requirement model{specification} & the programming language can be represented in a rigorous manner,
- So it should be possible to apply mathematical proof of correctness to demonstrate that the program confirms exactly to its specifications.

Statistical Quality Assurance

- ▪ Assuming that computer program is a mathematical object, So it should be possible to **apply mathematical proof of correctness** to demonstrate that the program confirms exactly to its specifications.

Statistical quality assurance steps:

- ✓ Information about software defects are collected & categorized.
- ✓ An attempt is made to trace each defect to its underlying cause.
- ✓ Use **pareto principle** to trace defect:
 - “80% of software problems are caused by 20% of bugs”
i.e. most of the problems are caused by a handful of serious bugs.
 - “80% of the defects can be traced to 20% of all possible causes”.
i.e. Isolate 20% (“The vital few”)
- ✓ Once vital few identified, move to correct the problem.

Application of statistical SQA & pareto principle can be summarized as:

“Spend your time focusing on things that really matters but at first, be sure that you understand what really matters.”

Statistical Quality Assurance

- Although hundreds of errors are uncovered, all can be tracked to one of the following **causes**:
 - ✓ Incomplete or erroneous specification (IES)
 - ✓ Misinterpretation of customer communication (MCC)
 - ✓ Intentional deviation from specification (IDS)
 - ✓ Violation of programming standards (VPS)
 - ✓ Error in data representation (EDR)
 - ✓ Inconsistent module interface (IMI)
 - ✓ Error in design logic (EDL)
 - ✓ Incomplete or erroneous testing (IET)
 - ✓ Inaccurate or incomplete documentation (IID)
 - ✓ Error in programming language translation of design (PLT)
 - ✓ Ambiguous or inconsistent human-computer interface (HCI)
 - ✓ Miscellaneous (MIS)

Statistical Quality Assurance

- ° ✓ To apply SQA following table is build:

Error	Total		Serious		Moderate		Minor	
	NO.	%	NO	%	NO	%	NO	%
IES	205	20	34	27	68	18	103	24
MCC	156	17	19	9	68	18	76	17
IDS	48	5	1	1	24	6	23	5
VPS	25	3	0	0	15	4	10	2

Software Measurement and metrics

° Software measurement:

- ✓ concerned with **deriving a numeric value** for an attribute of a software.

Software metric

- ✓ Any type of **measurement** which relates to a software system, process or related documentation.
 - Lines of code in a program, the Fog index-**readability test**, number of person-days required to develop a component.
- ✓ Allow the software and the software process to be quantified.
- ✓ May be used to predict product attributes or to control the software process.
- ✓ Product metrics can be **used** for **general predictions** or to **identify anomalous components**.

Software product metrics

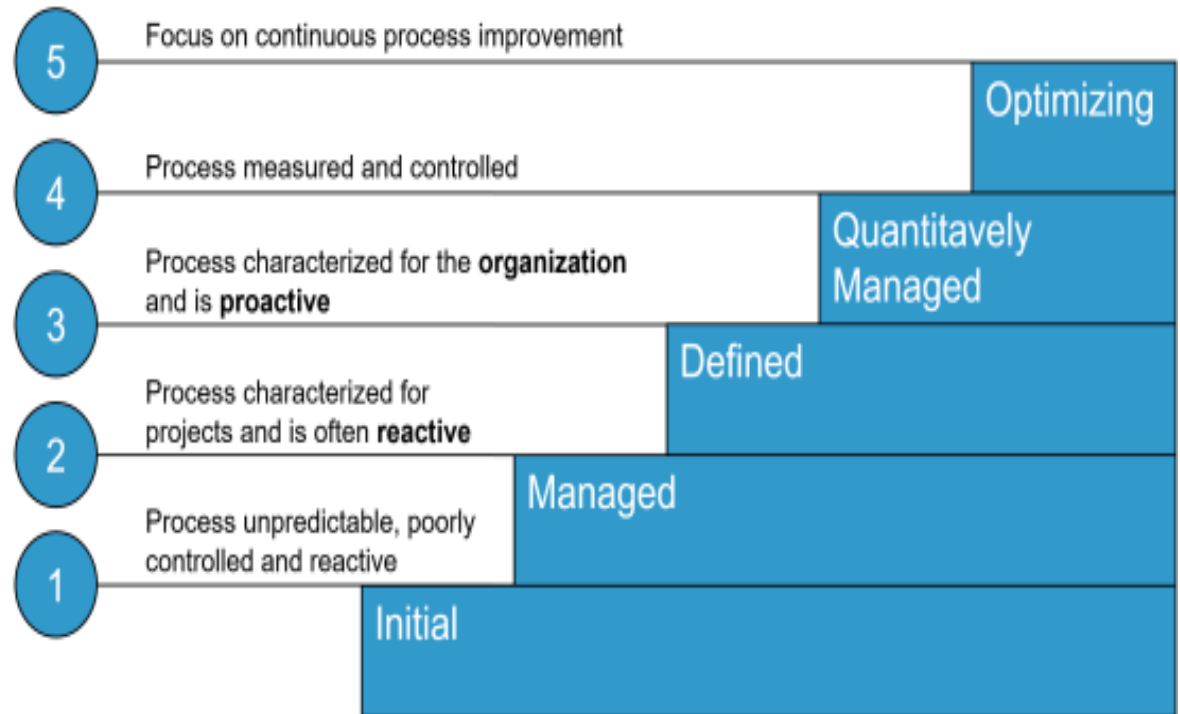
Software Metrics		Description
Fan-in/ Fan-out		Fan-in is a measure of the number of functions or methods that call some other function or method (say X). Fan-out is the number of functions that are called by function X . A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of Code		This is a measure of the size of a program . Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error proneness in components.
Length of identifiers	of	This is a measure of the average length of distinct identifiers in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional Nesting	of	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if statements are hard to understand and are potentially error-prone.
Fog index		This is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document is to understand.

Object oriented metrics

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

Capability Maturity Model Integration (CMMI)

It is a process improvement model whose goal is to help organizations improve their performance. CMMI can be used to guide process improvement across a project, a division, or an entire organization. Currently supported version is CMMI Version 1.3.



Maturity level I- Initial

- ▪Processes are usually ad hoc and chaotic. The organization usually does not provide a stable environment.
- Success in these organizations depends on the competence and heroics of the people in the organization and not on the use of proven processes.
- Maturity level 1 organizations often produce products and services that work; however, they frequently exceed the budget and schedule of their projects.
- Maturity level 1 organizations are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes.



Maturity level 2- Managed

- **At maturity level 2, an organization has achieved all the specific and generic goals of the maturity level 1 process areas.**
- **At Maturity Level 2, an organization's development processes are repeatable and produce consistent results.**
- **At this stage, all business projects are managed so that processes are “planned, performed, measured and controlled”**

Maturity level 3- Defined

- At this stage, organizations are **more proactive** {proactive development you solve matters before they become an issue} **than reactive**.
- Businesses understand their shortcomings, how to address them and what the goal is for improvement.
- Organization has well characterized and understood processes that are described in standards, procedures, tools and methods,



Maturity level 4- Quantitatively managed

- Achieve All the specific and generic goals at level 3 and more measured and controlled than level 3.
- Processes have reached a stage where they can be measured using defined metrics that demonstrate how the process is beneficial to business operations.
- Processes have been repeatedly tested, refined and adapted in multiple conditions across the organization.
- At this level, your process should easily adapt to suit other projects in the organization and to stand as a template for future process development.



Maturity level 5 - Optimizing

- Final level of CMMI, achieve all the specific and generic goal at level 4.
- processes are continually monitored and improved as needed.
- At this level, organizations processes should always remain flexible enough to accommodate new technologies and innovation in the organization.

Software Reliability

Definition:

- ✓ In statistical term, it is the **probability of failure free operation of a computer program in a specified environment for a specified time.**
- ✓ **Reliability is the probability of not failing in a specified length of time.**

Example:

Program X is executed to have reliability of 0.96 over eight elapsed processing hours **i.e.**

if program x were to be executed 100 times & requires eight hours of elapsed processing time, It is likely to operate correctly 96 out of 100.

Software Reliability and Failure

- Mathematical representation of Software failure:

$$F(n) = 1 - R(n) \quad \text{where,}$$

F(n) = probability of failing in a specified length of time.

R(n) = probability of reliability (i.e. not failing)

n = no. of time units,

Note: If time unit is assumed in days then probability of not failing in 1 day is $R(1)$

Measure of Reliability and Availability

Measure of Reliability

A simple measure of reliability for such a system is *mean-time-between-failure (MTBF)*

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

✓ MTTF = Mean-time-to-failure

✓ MTTR = Mean-time-to-repair

Measure of Availability:

It is the probability that a program is operating according to the requirements at a given point in time & is defined as:

$$\text{Availability} = [\text{MTTF} / (\text{MTTF} + \text{MTTR})] * 100\%$$



Software Safety

- Focuses on identification and assessment of potential hazards to software operation.
- Is different from software reliability
- Software reliability uses statistical analysis to determine the likelihood that a software failure will occur; however, the failure may not necessarily result in a hazard or mishap.
- Software safety examines the ways in which failures result in conditions that can lead to a hazard or mishap; it identifies faults that may lead to failures.
- Software failures are evaluated in the context of an entire computer-based system and its environment through the process of fault tree analysis or hazard analysis.

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

Software Engineering
Chapter 10
Configuration Management

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

Software Configuration Management

- It is the **task of tracking and controlling changes** in the software.
- If something goes wrong, SCM can determine what was changed and who changed it. If a configuration is working well, SCM can determine how to replicate it across many hosts.- **Wikipedia**
- New versions of software systems are created as they change, **but why:**
 - ✓ For different machines/OS.
 - ✓ Offering different functionality.
 - ✓ Tailored for particular user requirements.

Software Configuration Management Planning

All products of the software process may have to be managed:

- ✓ Specifications.
- ✓ Designs.
- ✓ Programs.
- ✓ Test data.
- ✓ User manuals.

Thousands of separate documents may be generated for a large, complex software system.



Configuration Management Plan

- Defines the types of documents to be managed and a document naming scheme.
- Defines who takes responsibility for the CM procedures and creation of baselines.
- Defines policies for change control and version management.
- Defines the CM records which must be maintained.
- Describes the tools which should be used to assist the CM process and any limitations on their use.
- Defines the process of tool use.
- Defines the CM database used to record configuration information.



Change Management

- Software systems are subject to continual change requests:
 - ✓ From users
 - ✓ From developers
 - ✓ From market forces
- Change management is concerned with keeping track of these changes and ensuring that they are implemented in the most cost-effective way.

Change Management Process

Request change by completing a change request form.

Analyze change request

if change is valid **then**

- ✓ Assess how change might be implemented
- ✓ Assess change cost
- ✓ Submit request to change control board

if change is accepted **then**

repeat

- ✓ make changes to software
- ✓ submit changed software for quality approval

until software quality is adequate/acceptable
create new system version

else

reject change request

else

reject change request



Version and Release Management

- Invent an identification scheme for system versions.
- Plan when a new system version is to be produced.
- Ensure that version management procedures and tools are properly applied.
- Plan and distribute new system releases.

▪ Version vs Variant vs Release

- **Version:** An instance of a system which is functionally distinct in some way from other system instances.
- **Variant:** An instance of a system which is functionally identical but non-functionally distinct from other instances of a system.
- **Release:** An instance of a system which is distributed to users outside of the development team.

Version Identification

Procedures for version identification should define an unambiguous way of identifying component versions.

Some of basic techniques for component identification

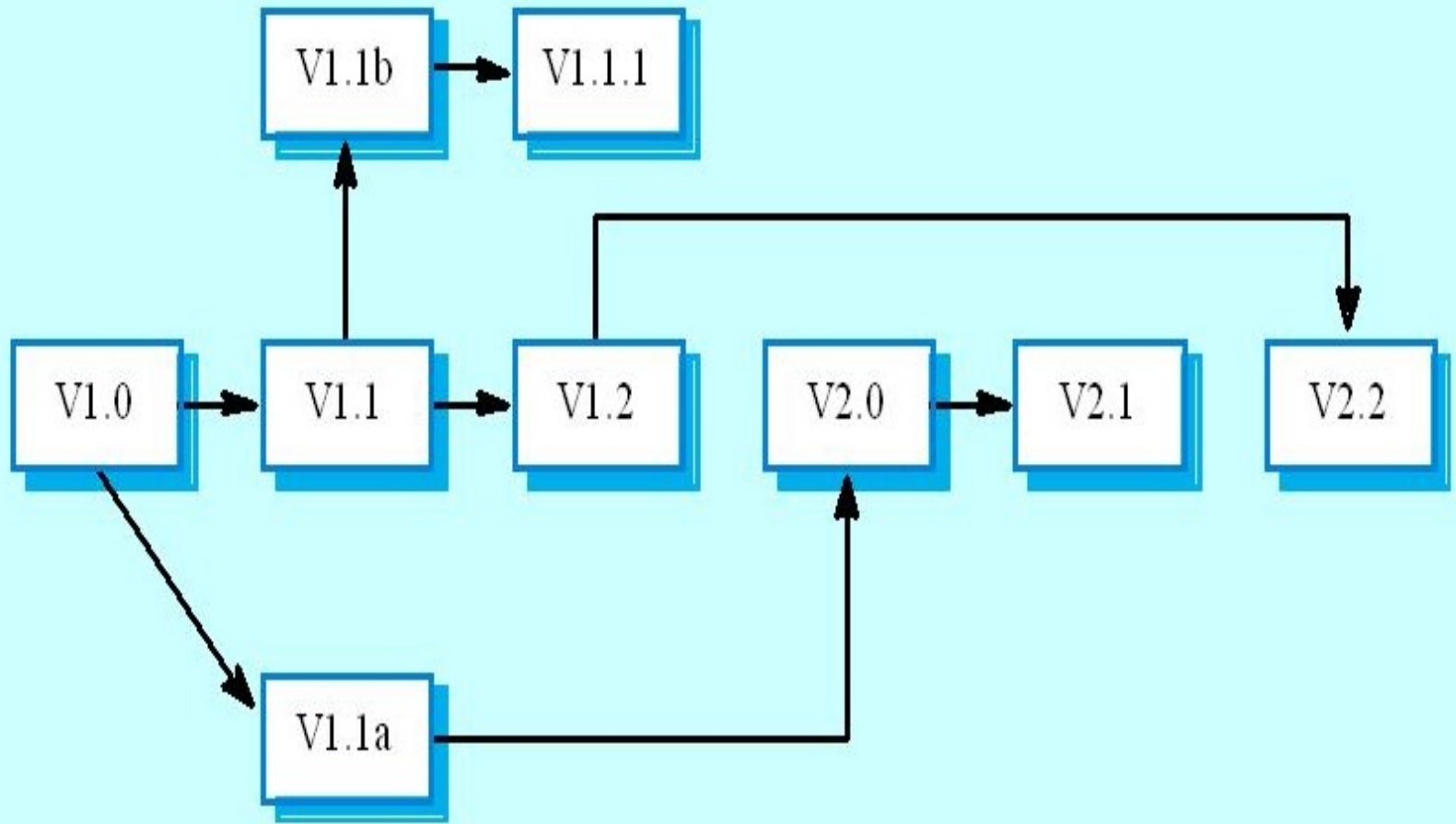
1. Version numbering
2. Attribute-based identification etc.

1. Version Numbering

✓ Simple naming scheme uses a linear derivation such as
→ V1, V1.1, V1.2, V2.1, V2.2 etc.

✓ The actual derivation structure is a tree or a network rather than a sequence.

Version Numbering



Version Identification

2. Attribute-based identification

- Attributes can be associated with a version with the combination of attributes identifying that version
- Examples of attributes are Date, Creator, Programming Language, Customer, Status etc.
- In practice, a version also needs an associated name for easy reference.

Release Management

- Releases must incorporate changes forced on the system by users due to errors discovered, hardware changes etc.
- They must also incorporate new system functionality.
- Release planning is concerned with when to issue a system version as a release.

System Release

- Not just a set of executable programs. It may also include:
 - ✓ Configuration files defining how the release is configured for a particular installation.
 - ✓ Data files needed for system operation.
 - ✓ An installation program or shell script to install the system on target hardware.
 - ✓ Electronic and paper documentation,
 - ✓ Packaging etc.
- Systems are now normally released on optical disks (CD or DVD) or as downloadable installation files from the web.



Release Problem

- Customer may not want a new release of the system
 - ✓ They may be happy with their current system as the new version may provide unwanted functionality.
- Release management should not assume that all previous releases have been accepted. So, all files required for a release should be re-created when a new release is installed.

Case Tools for CM {see by yourself}

- CM processes are standardized and involve applying pre-defined procedures.
 - ✓ Large amounts of data must be managed.
 - ✓ CASE tool support for CM is therefore essential.
 - ✓ Mature CASE tools to support configuration management are available ranging from stand-alone tools to integrated CM workbenches.

▪ CM workbenches

Open workbenches

Tools for each stage in the CM process are integrated through organizational procedures and scripts. Gives flexibility in tool selection.

Integrated workbenches

Provide whole-process, integrated support for configuration management. More tightly integrated tools so easier to use. However, the cost is less flexibility in the tools used.

▪ System Building

Building a large system is computationally expensive and may take several hours.

Hundreds of files may be involved.

System building tools may provide

- ✓ A dependency specification language and interpreter
- ✓ Tool selection and instantiation support
- ✓ Distributed compilation
- ✓ Derived object management.

Thank

You!!!