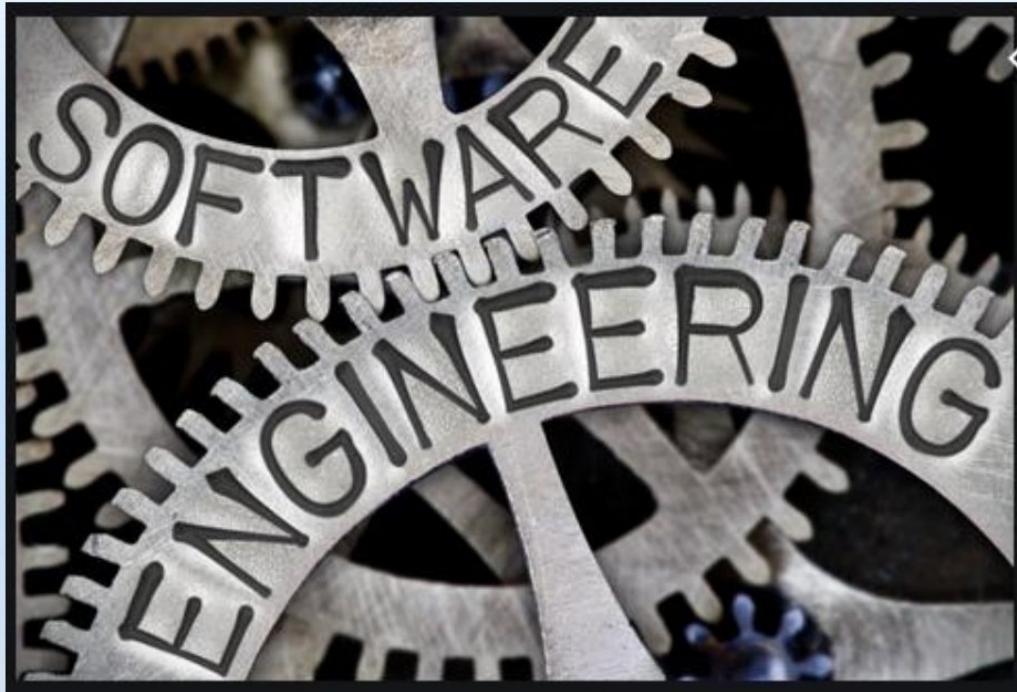


Software Engineering

Module-1



Faculty :
Dr. Suchismita Rout
Associate Professor

Module-1 Contents

- **Software Engineering:**
 - Introduction & Evolving role of software, Process framework, **CMM**
- **Software Life Cycle Models:**
 - **Waterfall model, Iterative Waterfall model**
 - **V-Process model**
 - **Incremental Process model**
 - **Evolutionary Process models**
 - Prototyping & Spiral model
 - **6. Agile & RAD models**
 - Extreme Programming, Scrum, Crystal models,
 - **Unified Process**

Introduction to S/W Engineering

- What is **Software Engineering**?
- Problem complexity reduction using:
 - **Abstraction & Decomposition**
- **S/W Crisis**
- Diff between **S/W Programs** vs. **S/W Products**
- Diff between **Systems Engg** vs **S/W Engg**
- **Evolution of Software Engineering**
- Introduction to **Life Cycle Models**

What is S/W Engg.

➤ What is **Engineering**?



➤ What is **Software Engineering**?



➤ Is writing program straightaway in computer is S/W Engg ?



Introduction to S/W Engineering

- What is **Software Engineering**?
- Problem complexity reduction using:
 - **Abstraction & Decomposition**
- **S/W Crisis**
- Diff between **S/W Programs** vs. **S/W Products**
- Diff between **Systems Engg** vs **S/W Engg**
- **Evolution of Software Engineering**
- Introduction to **Life Cycle Models**

SDLC Models



- **1. Waterfall model**
- **2. Iterative Waterfall model**
- **3. V-Process model**
- **4. Incremental Process model**
- **5. Evolutionary Process models**
 - **5.1. Prototyping model**
 - **5.2. Spiral model**
- **6. Agile & RAD models**
 - **6.1. Extreme Programming**
 - **6.2. Scrum**
 - **6.3. Crystal**
- **Unified Process**

S/W Engineering

➤ S/W Engineering

- Is Methodical approach to software development
- Makes use of past experience
- Systematic use of techniques, methodologies & guidelines



➤ Purpose

- To achieve quality s/w which is cost effective

- Two important techniques used to reduce problem complexity

- Abstraction and Decomposition



S/W Development Myths

Myth-1:

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

Myth:-2:

- If we get behind schedule, we can add more programmers & catch up (“Mongolian horde” concept).

Myth-3:

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



Legacy software - Maintenance

- Legacy software systems were developed decades ago & have been continually modified to meet changes in business requirements & computing platforms
- The maintenance of such systems is causing Difficulties for large organizations who find them costly to maintain & risky to evolve.

Abstraction



- **Abstraction** is a powerful way of reducing complexity of problem
- Simplify a problem by :
 - **Considering** relevant aspects & **suppressing** irrelevant aspects
- Top managers are not interested in tech. details of each program
- **Ex:** Inputs (loan amount, duration) & outputs (EMI) of “Loan EMI Calc” system
- The omitted details are considered in next level abstraction

Decomposition



- Another approach to tackle problem complexity
- A complex problem is divided into **smaller problems**
- The smaller problems are then **solved one by one**
- But, random decompositions does not reduce complexity
- Problem is decomposed such that each component can be **solved independently**
- Then component solutions are **combined** to get the full solution

★ Software Crisis



- Occurs when **S/w Products:**
 - **Fail** to meet **user requirements**
 - Frequently **crash** due to **bad design**
 - Difficult to **debug** & **alter**
 - **Delivered late**
 - **Over-budget**
 - Skill Shortage
 - Low productivity
 - **Lack of adequate training in s/w engg**



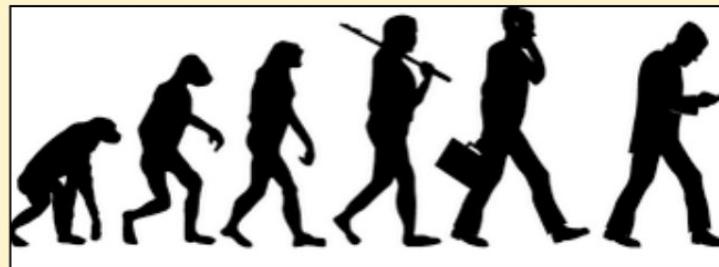
Programs Vs S/W Products



- Usually small in size
 - Author may be the sole user
 - Single developer
 - Lacks proper user interface
 - Lacks proper documentation
 - Ad hoc development
 - Large size
 - Large number of users
 - Team of developers
 - Well-designed interface
 - Well documented & user-manual prepared
 - Systematic development



Evolution & Emergence of Software Engineering



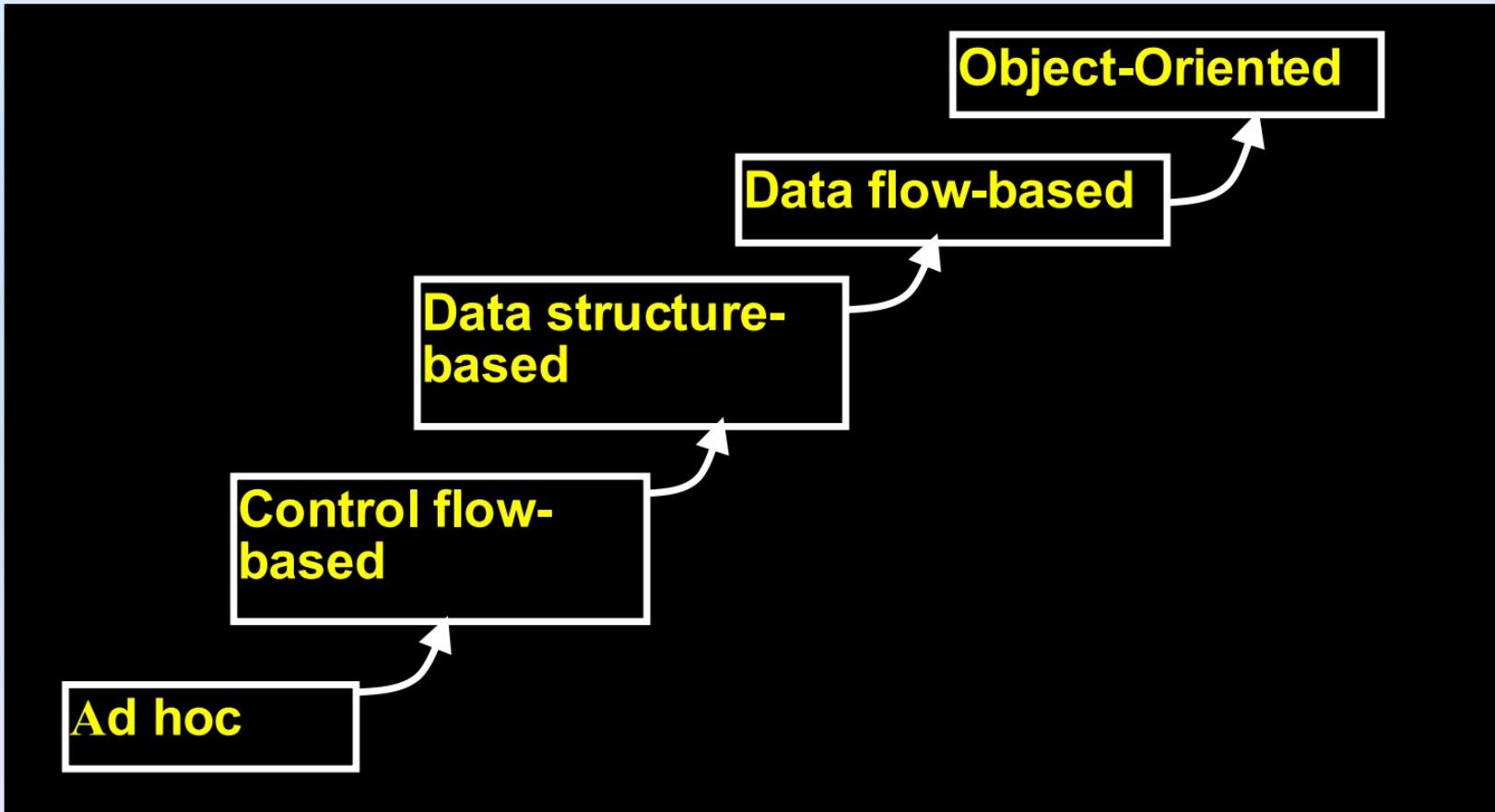


Evolution of Software Engineering

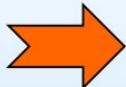
- Early Programming Style (*Exploratory/ Adhoc, Assembly Language prog.*)
- High Level Programming (*Exploratory/ Adhoc, **HLL**- Fortran, Cobol...*)
- Control flow based Programming (*Design using **Flow-charts***)
- Structured Programming (*Decompose into set of modules*)
- Data Structure Oriented Programming (*Design data structure – Then design program structure*)
- Data Flow Oriented Design (*Input-Process-Output : **DFD***)
- Object Oriented Design (*Design objects & relationships between them*)
- Modern S/W Engg Techniques (*SDLC, CASE tools*)



Evolution of Design Techniques



★1. Early Computer Programming (1950s)



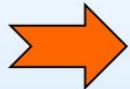
- Programs were written in assembly language



- Every programmer uses his own style
(Called exploratory programming)
- Difficult to learn & understand



2. High-Level Language Programming (Early 60s)



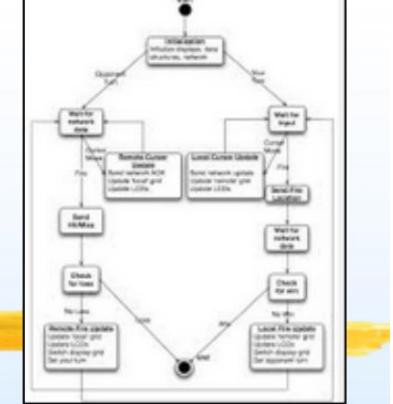
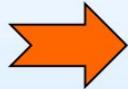
There are many high level languages

Some Examples:

COBOL	Business applications
FORTRAN	Engineering & Scientific Applications
PASCAL	General use and as a teaching tool
C & C++	General Purpose - currently most popular.
PROLOG	Artificial Intelligence
JAVA	General all purpose programming
.NET	General or web applications.

- High-level languages such as **FORTRAN, ALGOL & COBOL** were introduced
Easy to learn & understand
- This reduced s/w development efforts
- S/w development style was still exploratory

★ 3. Control Flow-Based Design (late 60s)



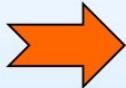
- Size & complexity of programs increased
- Exploratory programming style was insufficient
- Difficulty faced to write complex programs
- Difficulty faced to understand & maintain other's code
- Focus on **Control flow (Flow Chart)** based design
- **Flow charting technique** was developed

★ Disadv.s of Control Flow-Based Design

- Messy flow charts are difficult to understand & debug
- GO TO statements makes a program messy
- Alter the flow of control arbitrarily
- Need to restrict use of GO TO statements



4. Structured Programming



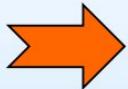
Chapter 5: Structured Programming

- In this chapter you will learn about:
- Sequential structure
 - Selection structure
 - if
 - if...else
 - switch
 - Repetition Structure
 - while
 - do...while
 - for
 - Continue and break statements

- It was proved that a program needs only below 3 types of statements (GO-TO not needed)
- A program is called structured, if it uses below types of constructs :
 - **Sequence** (eg: `a=0;b=5;`)
 - **Selection** (eg: `if(c=true) k=5 else m=5;`)
 - **Iteration** (eg: `while(k>0) k=j-k;`)

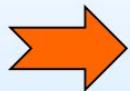


Structured programming



- Unstructured control flows are avoided
- Consist of a clean set of modules
- Use **single-entry, single-exit** program constructs
- **Structured programs** are :
 - Easier to read & understand & maintain
 - Require **less effort** & **time** to develop

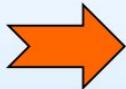
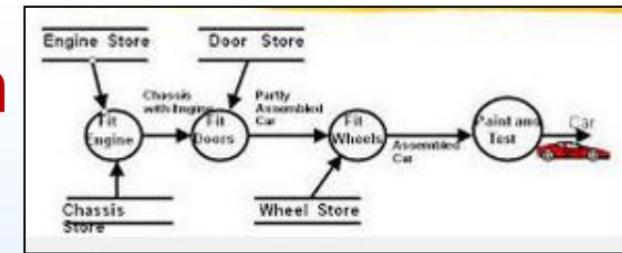
★ 5. Data Structure Oriented Design (Early 70s)



- In this methodology:
 - Program's data structures are first designed
 - From that program structure is derived



6. Data Flow-Oriented Design (Late 70s)



- In Data flow-oriented technique:
 - First Identify input data items of the system
 - Then Processing required to produce the outputs is determined
 - **Ex:** Payroll System, Credit card approval system

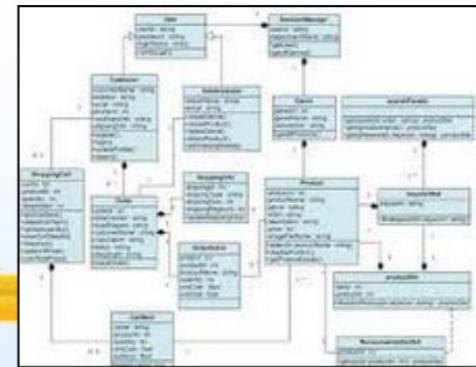
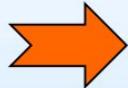


Data Flow-Oriented Design (Late 70s)

- Data flow technique is a generic technique:
- Can be used to model the working of any system not just s/w systems
- A major advantage of the data flow technique is its simplicity



7. Object-Oriented Design (80s)



- In Object-oriented technique:
 - Objects (such as **employees**, **pay-roll-register**, etc.) occurring in a problem are first **identified**
 - Relationships among objects are determined
 - Each **object** essentially acts as a **data hiding** entity



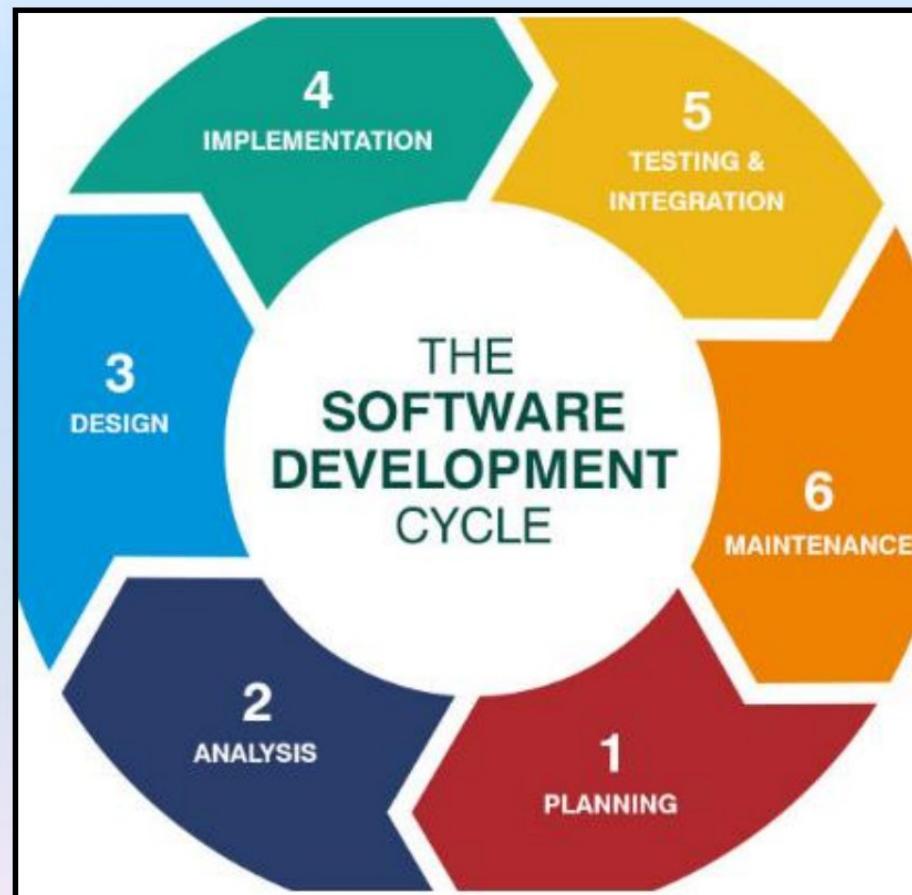
Advantages of Object-Oriented Design

- OO Techniques are very popular & well-accepted due to:
 - Simplicity
 - Reusability
 - Lower development time & cost
 - More robust code
 - Easy maintenance

Changes in S/W Engg Techniques

- Development of below Software Engg techniques :
 - Software Development Life cycle models (**SDLCs**)
 - Specification techniques (formal techs like State Transition diagrams)
 - Project management techniques (Chief-prog, Democratic)
 - Testing techniques (Automated, Manual)
 - Debugging techniques (Brute-force, Backtracking, Prog slicing)
 - Quality assurance techniques (ISO 9001, Six Sigma)
 - Software measurement techniques (LOC, FP)
 - CASE tools (Rational Rose/Architect) etc

Software Life Cycle Models



➤ Topics:

- **Exploratory Vs Modern S/W Engg techniques**
- **Software Life Cycle**
- **Software Life Cycle Models**

Exploratory style VS modern s/w development practices

- Use of Life Cycle Models
- S/W is developed through well-defined stages:
 - Feasibility study
 - Requirements analysis & specification
 - Design
 - Coding
 - Testing
 - Maintenance
- Emphasis has shifted
 - From error correction to error prevention



Exploratory style Vs modern s/w development practices

- In exploratory style, errors are detected only during testing
 - Now, focus is on detecting errors in **each phase of s/w development**
- In exploratory style, main focus was on coding
 - Now, coding is considered only a small part of s/w dev.
- A lot of effort and attention is on requirements specification

Exploratory style Vs modern s/w development practices

- Creation of good quality documents
 - In the past, very little attention was being given to producing good quality documents
- ✓ **Several metrics are being used:**
 - ✓ To help in s/w project mgmt, quality assurance etc.

CASE tools are being used

Process Framework

- **Software Process** - *A process is a collection of activities, actions & tasks that are performed when some work product is to be created*
- **The Process Framework** - *A process framework provides the foundation for a complete s/w engg. process by identifying a no. of framework activities*
- Process framework specifies a set of *umbrella activities* that are applicable across the entire s/w process
- A generic process framework for s/w engg contains **five activities**:

1. Communication

2. Planning

3. Modeling

4. Construction

5. Deployment

Process Framework

1. Communication

- Communication with customer & stakeholders is very important in order to understand the **project objectives** & to **gather requirements** that help define s/w features & functions.

2. Planning

- Creation of a map, called a **s/w project plan** to **define** the s/w engg work by describing the **tech. tasks** to be conducted, the likely risks, the **resources required**, the work products & the work schedule.

3. Modeling

- A s/w engineer needs to **create models of the system** to understand the **requirements** & the **design** that will achieve the requirements.

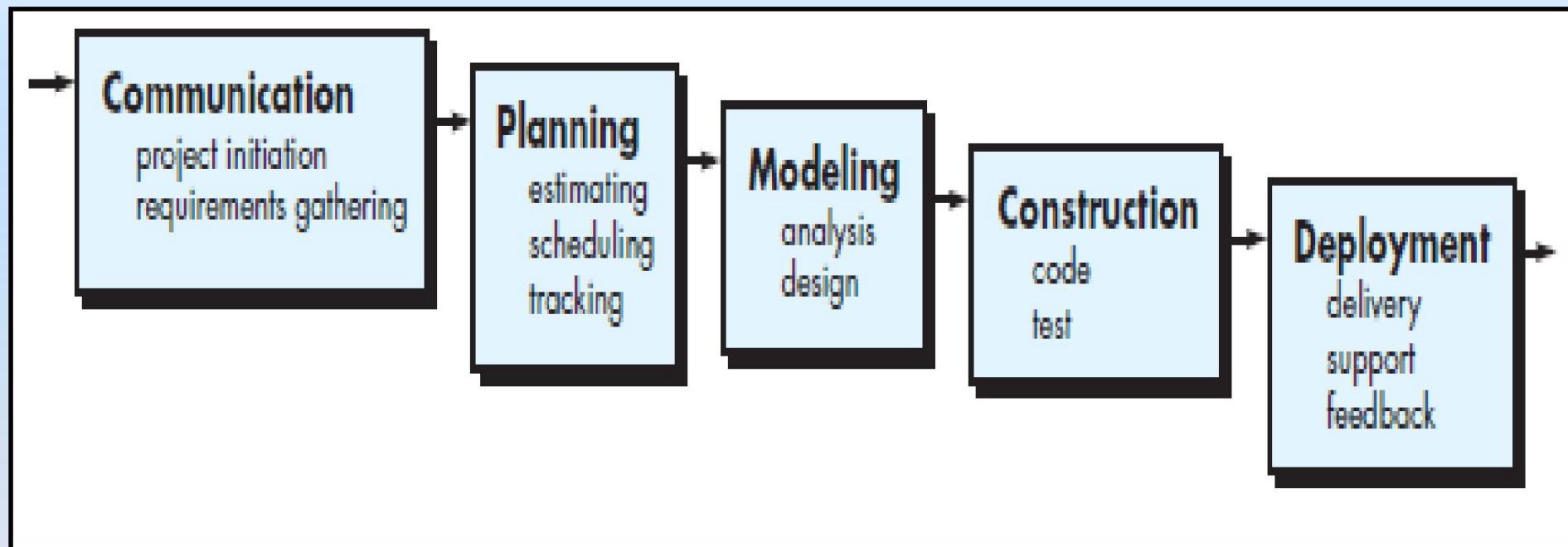
4. Construction

- Building what was designed. It is a combination of code generation & testing to uncover errors in the code.

5. Deployment

- The process of **delivering the completed product** to the customer.

Process Framework





Capability Maturity Model (CMM)

by SEI

★ Process Capability Models : SEI Capability Maturity Model (**CMM**)

- **SEI-CMM** is a widely-accepted **quality certification** offered by **SEI** mainly to s/w organizations
- **SEI** - Software Engineering Institute (SEI), Mellon University, USA
- **Aim:** To improve **quality** of *s/w products* through **5 stages**



★ Process Capability Models : SEI Capability Maturity Model (CMM)



- SEI CMM helps organizations:
 - To improve quality of s/w developed
- Very popular & adopted by many organizations

★ Process Capability Models : SEI Capability Maturity Model (CMM)

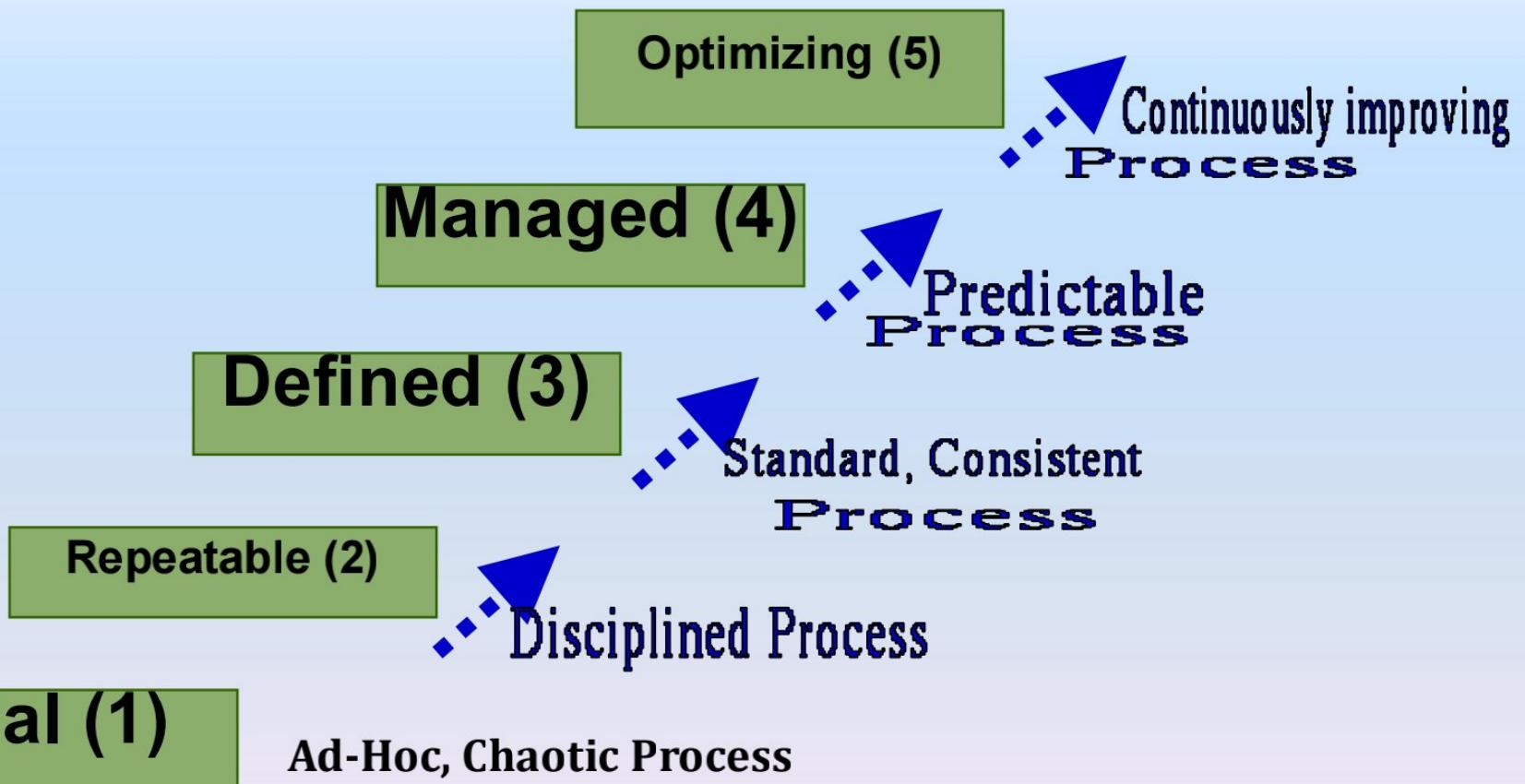
- **CMM** is a model for evaluating the s/w process maturity of an Org. into one of five different levels



- Can be used in two ways:
 - For Capability evaluation of an Org.
 - Software process assessment



SEI -CMM





Level 1: (Initial)

- Org. operates **without** any **formalized process** or **project plan**
- Characterized by **ad hoc** and often **chaotic activities**
- **Different engineers** follow their **own process**
- The **success** of projects depend on **individual efforts** & **heroics**



Level 2: (Repeatable)

- Basic project management practices like
 - Tracking of *cost, schedule* & *functionality* are followed
- Size and cost estimation techniques like
 - *Function point analysis, COCOMO..* are used
- Development process is still ad hoc (*neither formally defined & nor documented*)
- Process may vary between different projects
- Earlier success on projects can be repeated

★ Level 3: (Defined)

- Management & development activities are :
Defined & Documented
- Common org-wide standards of activities, roles & responsibilities exist
- The processes are **defined**
- But, process & product qualities are **not measured**

★ Level 4: (Managed)

- **Quantitative quality goals** for products are **set**
 - **Ex:** Defects per Kloc, MTBF
- **Software processes** & **product qualities** are **measured**
- The **measured values** are **used to improve the product quality**
- But, these are **not** used to **improve the processes**



Level 5: (Optimizing)

- Statistics collected from process/product measurements are analyzed:
- **Continuous process improvement** is done based on the measurements
- Known types of defects are **prevented** from recurring
- Lessons learned from projects incorporated into the process
- **Best software engineering** practices, methods & innovations are identified & promoted throughout the org.

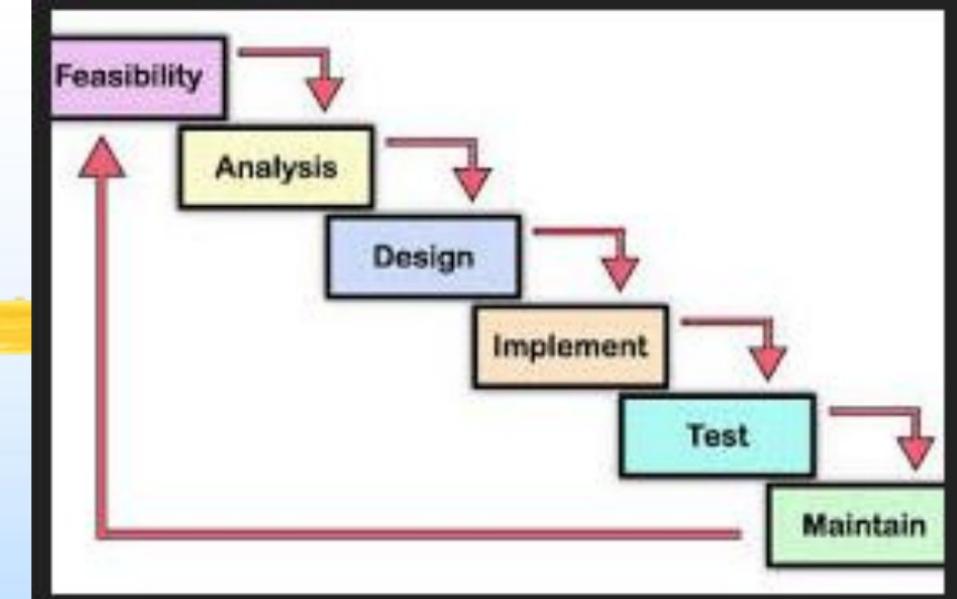
Software Life Cycle

Objectives :

- What is a **life cycle model**
- What **problems** would occur if life cycle model is not used
- Different **software life cycle models**
- Different **phases** of the **software lifecycle**
- **Activities** undertaken in each **phase**
- **Shortcomings** of the each life cycle model



Life Cycle Model



- A software life cycle model (or process model):
 - Is a descriptive & diagrammatic representation of software life cycle
 - It shows all the activities required for s/w product development
 - Establishes a precedence ordering among the activities

Software Life Cycle Model



- **Software life cycle** (or software process):
 - Series of stages/phases that a software product undergoes during its life time are:
 - Feasibility study
 - Requirements analysis and specification
 - Design
 - Coding
 - Testing
 - Maintenance
 - Helps dev. of s/w in a systematic & disciplined manner

SDLC Models



- **1. Waterfall model**
- **2. Iterative Waterfall model**
- **3. V-Process model**
- **4. Incremental Process model**
- **5. Evolutionary Process models**
 - **5.1. Prototyping model**
 - **5.2. Spiral model**
- **6. Agile & RAD Models**
 - **6.1. Extreme Programming**
 - **6.2. Scrum**
 - **6.3. Crystal**
- **Unified Process**

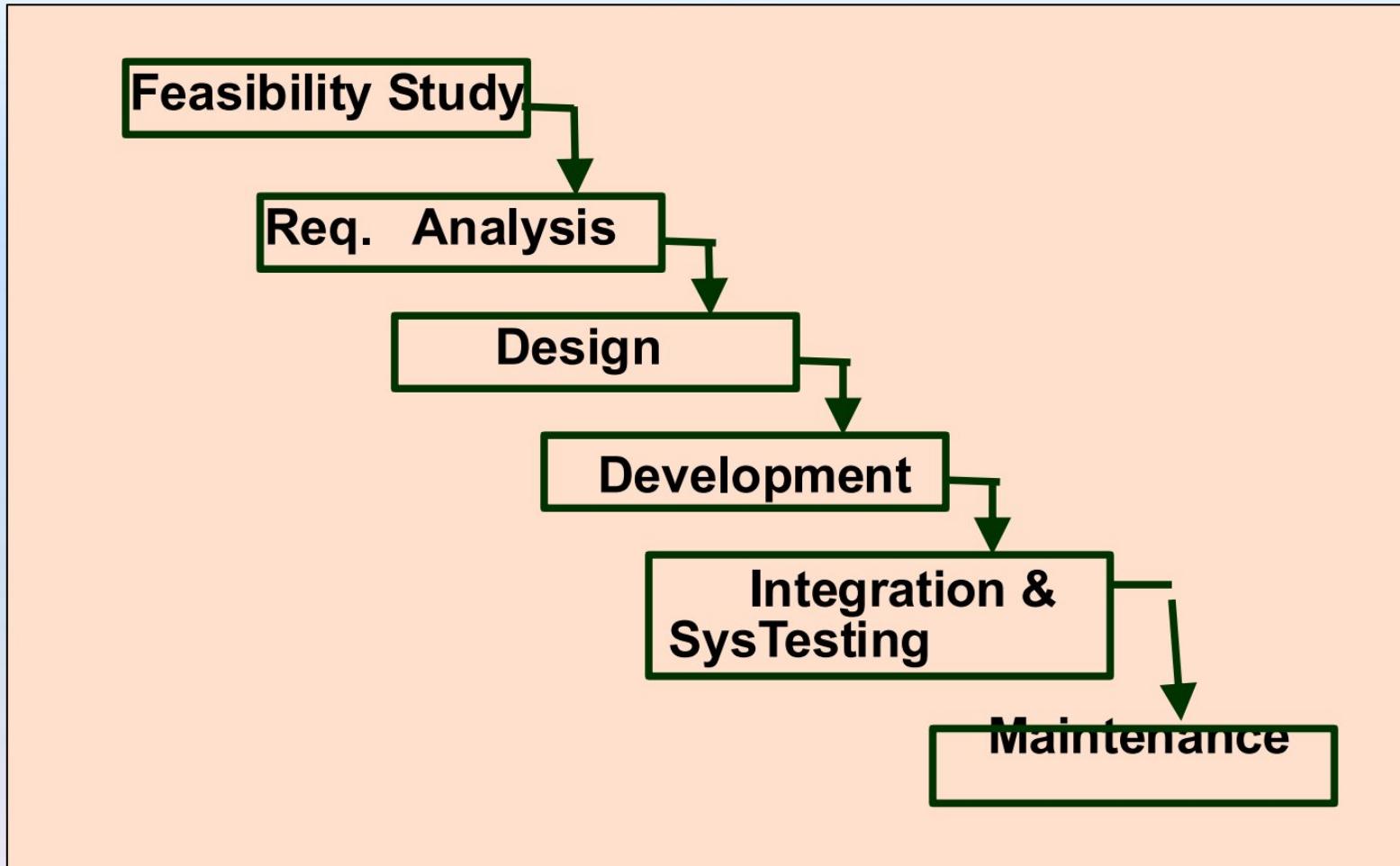


1. Classical Waterfall Model

- Classical waterfall model divides a project's life cycle into below phases:
 - *Feasibility study*
 - *Requirements analysis & specification*
 - *Design*
 - *Coding & Unit testing*
 - *Integration & system testing*
 - *Maintenance*



Waterfall model stages



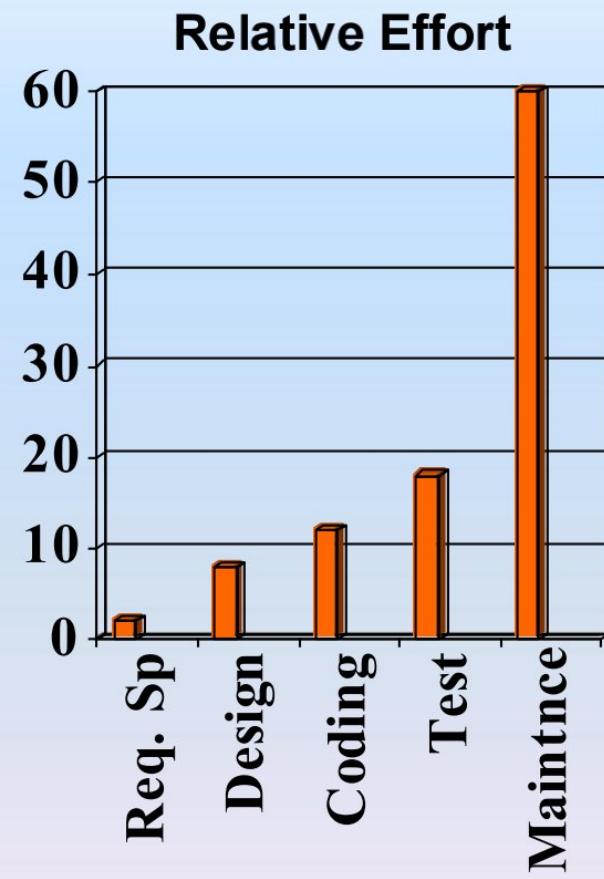
Quiz: Which stage takes most effort ??

52



Relative Effort for Phases

- The phases between feasibility study & testing are
 - Known as **development phases**
- Among all life cycle phases
 - **Maintenance phase consumes maximum effort**
- Among development phases,
 - **Testing phase consumes the maximum effort**



Phase-1: Feasibility Study



- Main aim of feasibility study: is to determine whether developing the software product is :
 - Financially worthwhile
 - Technically feasible
- Work out an overall understanding of the problem :
 - Data inputs
 - Processing needed
 - Output data
 - Various constraints (timeline, resource, performance..)

Activities during Feasibility Study

- Formulate different solution strategies (Alternatives)
- Examine each strategy
- Based on that Decide whether the project is feasible

Phase-2: Requirements Analysis & Specification

- The aim of this phase is:
 - To understand the **requirements** of the customer accurately
 - Then **Document** them properly
- Consists of two distinct activities:
 - Requirements gathering and analysis
 - Writing Requirements specification

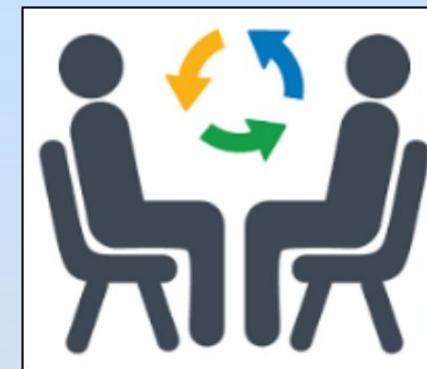


2.1. Requirements Gathering



➤ Gathering Requirements:

- Requirements are usually collected from the end-users through
 - Interviews
 - Discussions
 - Survey/ Questionnaire
 - Workshops
 - Group or one-to-one meetings etc..



2.2. Requirements Analysis



- The data you **initially collected** from the users:
 - May contain several **contradictions** & **ambiguities**
- These **ambiguities & contradictions**:
 - Must be **identified**
 - **Resolved** by **discussions** with the customers
- Then requirements are **organized**:
 - Into **Software Requirements Specification (SRS)** document

Stage-3: Design

- Design phase **transforms** the “*Requirements specification*”
 - into a form suitable for **implementation** in some programming language
- Two design approaches are there:
 - **Function oriented approach**
 - **Object oriented approach**



Function Oriented Design Approach

➤ Consists of two activities:

➤ **Structured analysis**

➤ Output is "Data flow Diagram"

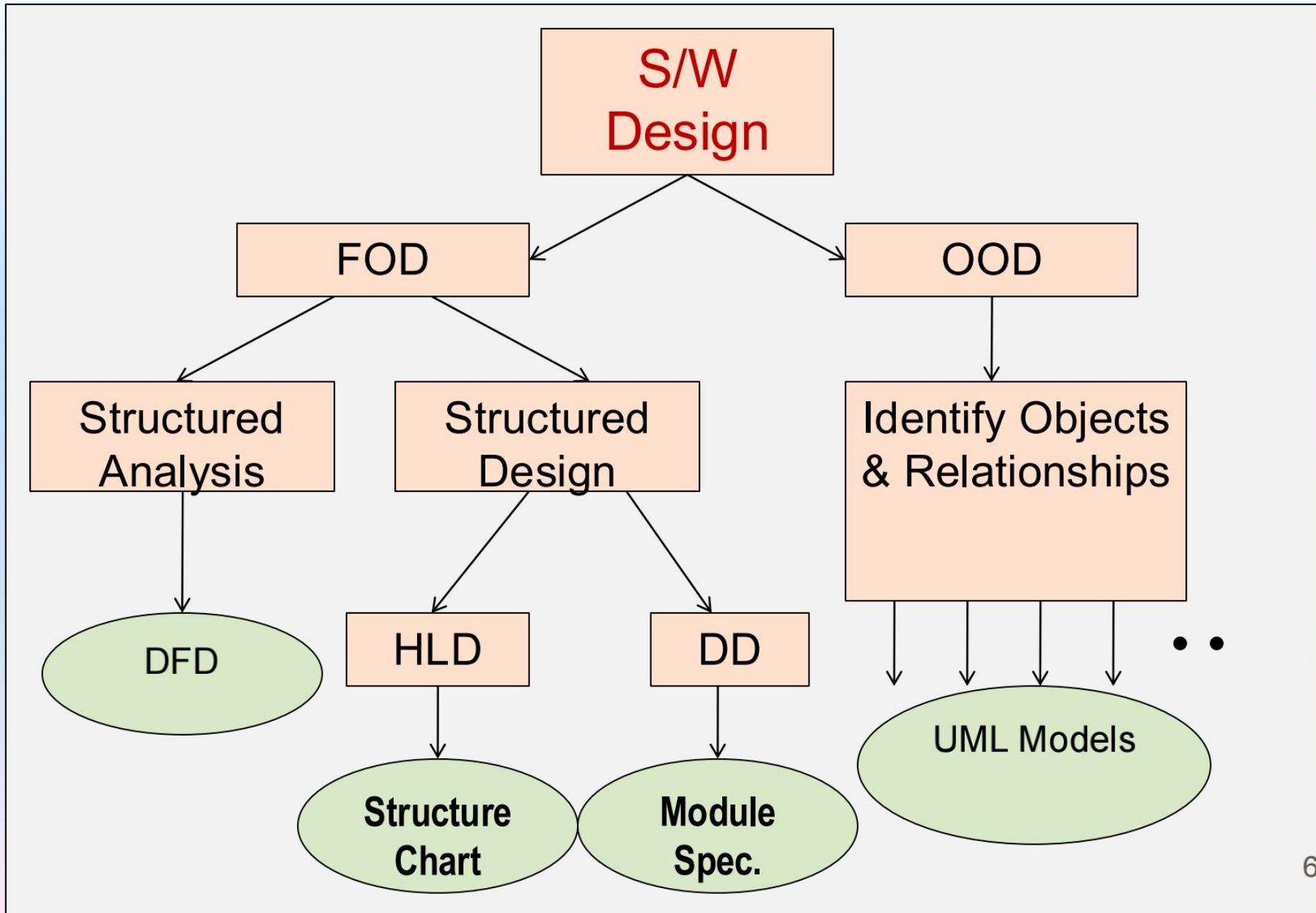
➤ **Structured design**

➤ Outputs :

1. High Level Design - "Structure Chart"

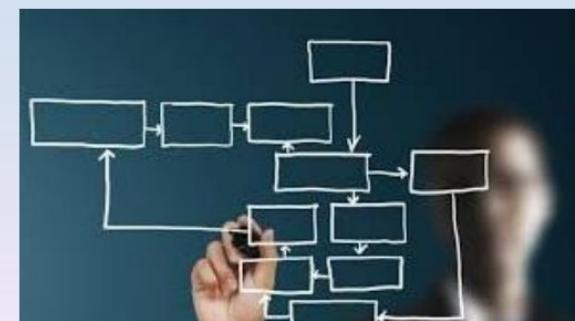
2. Detail Design – "Module Specification"

S/W Design Process

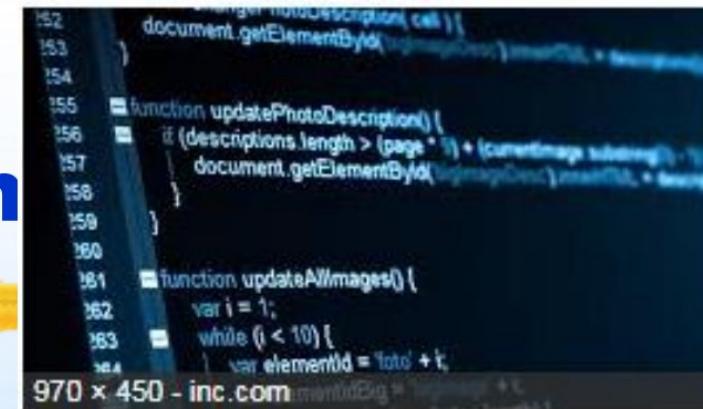


Object Oriented Design Approach

- First **identify various objects** (real world or conceptual entities) in the problem & **relationships** among them
- **Ex:** The objects in a pay-roll software may be:
 - Employees
 - Managers
 - Payroll register
 - Departments etc.



Stage-4: Implementation

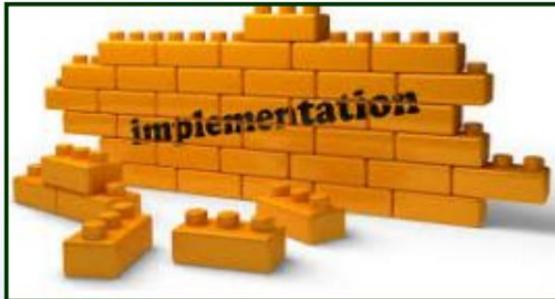


```
152 153 154 155 156 157 158 159 160 161 162 163 164
function updatePhotoDescription() {
  if (descriptions.length > (page * 1) + currentImage + 1) {
    document.getElementById('imageDesc').innerHTML = descriptions[page * 1 + currentImage];
  }
}

function updateAllImages() {
  var i = 1;
  while (i < 10) {
    var elementId = 'foto' + i;
    document.getElementById(elementId).innerHTML = descriptions[i];
    i++;
  }
}

970 x 450 - inc.com
```

- Purpose of **implementation** (**coding + unit testing**) phase is to :
 - **Translate software design into source code**



- During the **implementation** phase:
 - Each designed module is coded & unit tested *independently* for correctness
- The end product of **implementation** phase:
 - Program modules that have been tested individually

Stage-5: Integration & System Testing



Integration Testing :

- The **modules** are integrated in a **planned manner**:
 - Integrated in a number of steps
 - During each integration step
 - The **partially integrated system** is tested

Integration Testing



System Testing



- **System testing** is carried out after Integration testing

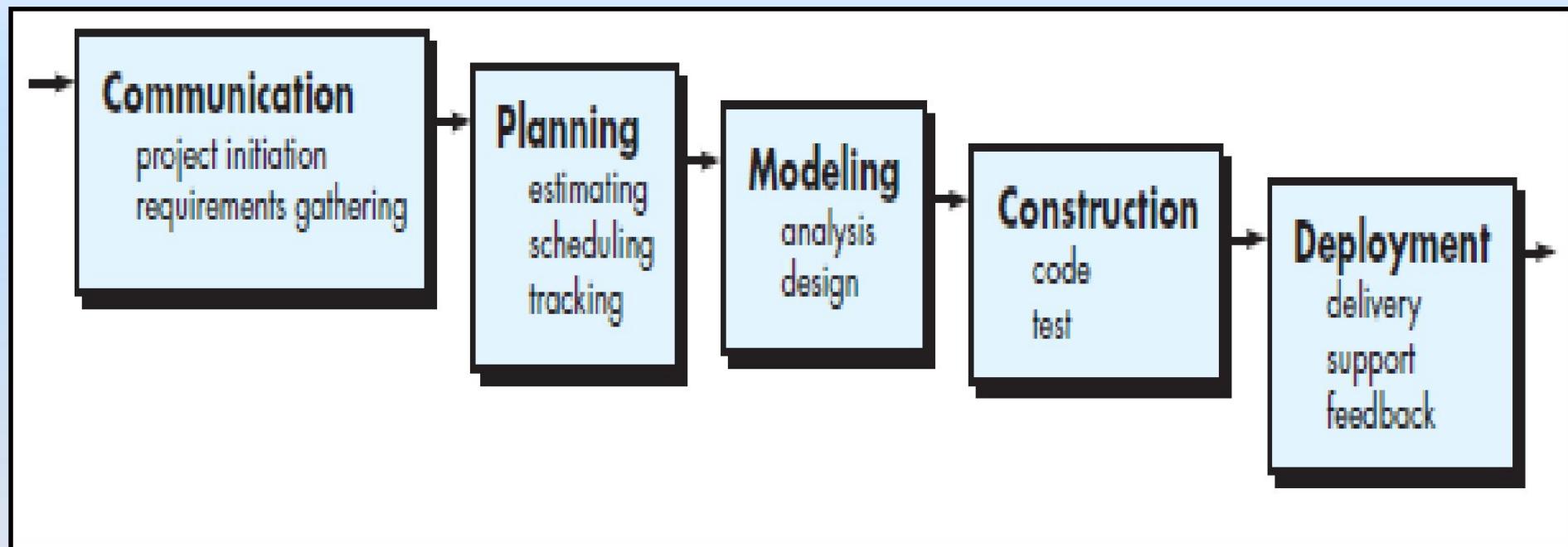
- **Goal of system testing:**
 - Ensure that the “**Developed system**” works according to “**Requirements**” specified in the SRS document

Stage-6: Maintenance



- **Maintenance** of any software product:
- Involves any **change** done to the product after it is delivered
- Requires **more effort** than **development**
 - 40:60

Waterfall model – an alternate diagram



WF model - Description of each stage

1. Communication

- Communication with customer & stakeholders is very important in order to understand the **project objectives** & to **gather requirements** that help define s/w features & functions.

2. Planning

- Creation of a map, called a **s/w project plan** to **define the s/w engg work** by describing the **tech. tasks** to be conducted, the **likely risks**, the **resources required**, the **work products** & the **work schedule**.

3. Modeling

- A **s/w engineer** needs to **create models of the system** to understand the **requirements** & the **design** that will achieve the requirements.

4. Construction

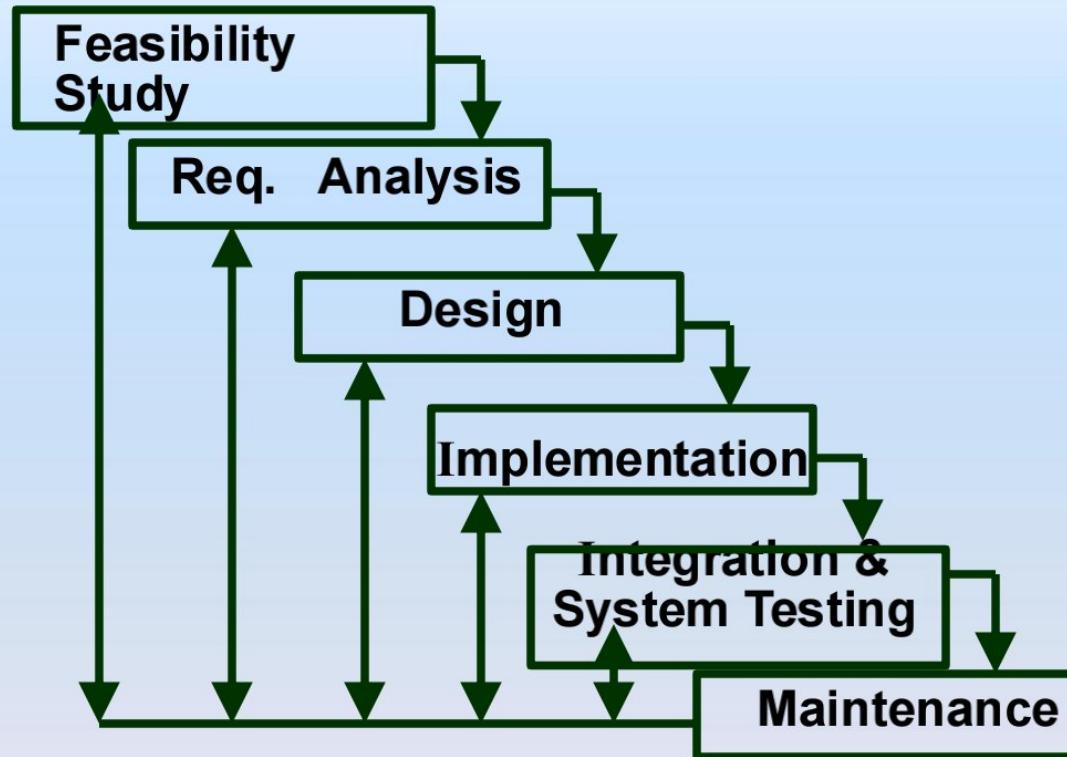
- Building what was designed. It is a combination of **code generation** & **testing** to uncover errors in the code.

5. Deployment

- The process of **delivering the completed product** to the customer.



(2) Iterative Waterfall Model





Iterative Waterfall Model

- Classical waterfall model is **idealistic** & **rigid**:
- It assumes, **no defect is found** in **earlier phases**, after we have moved to the **next phase**
- In practice: ***Defects*** are discovered in **earlier phases**
- Ex: A **design defect** might go unnoticed till the **coding** or **testing** phase



Iterative Waterfall Model

- Once a **defect is detected**:
 - We need to **go back to the phase** where it was **introduced**
 - **Redo** some work in **that phase & subsequent phases**
- Therefore we need **feedback paths** in ***waterfall model***
- **Iterative waterfall model** was the **most widely used model**

Phase containment of errors

- Principle of S/W Engg. recommends:

Detection of errors as close to its point of introduction as possible

- This is known as ***phase containment of errors***

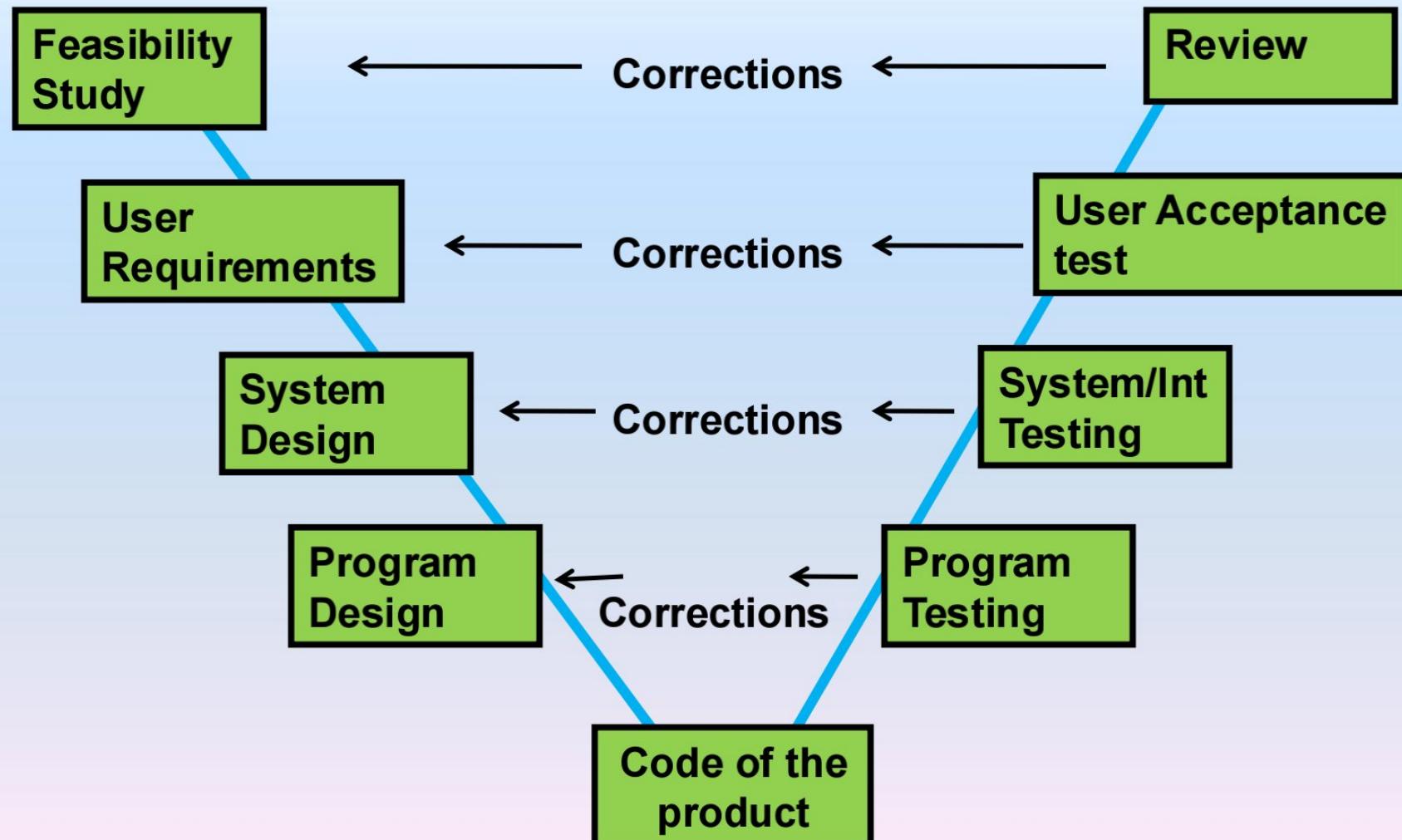


3. V-Process Model

- **Elaboration** of “Waterfall model” with **stress on testing/validation of each phase**
- **Expands** the Test/validation activities of Waterfall model
- Each Phase has a “**matching validation**” process
- If defects found, then “**loop back**” to corresponding development stage



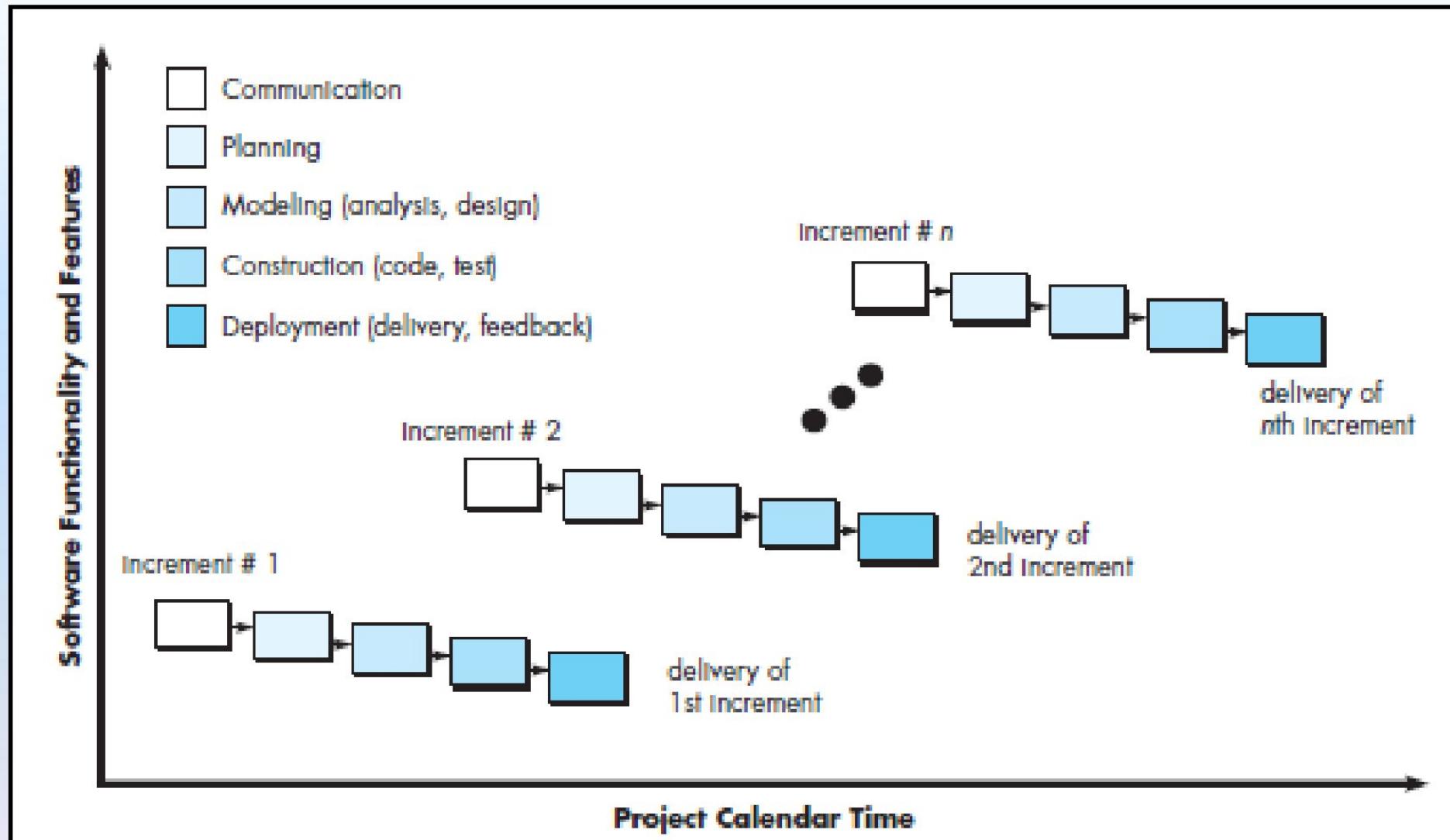
V-Process Model





(4) Incremental Process Model

- This model is suitable when:
 - There is a compelling need to provide a limited set of s/w functionality to users quickly & then expand that functionality in later s/w releases
- This model that is designed to **produce the s/w in increments**
- It combines both **linear** & **parallel** process flow
- It applies **linear sequence** in a **staggered fashion** as **time progresses**
- Each **linear sequence** produces deliverable “**increment**” of the s/w
- The **1st** increment normally contains the “**Core**” part of the product
- Additional features are delivered in **subsequent increments** until **complete product** is produced



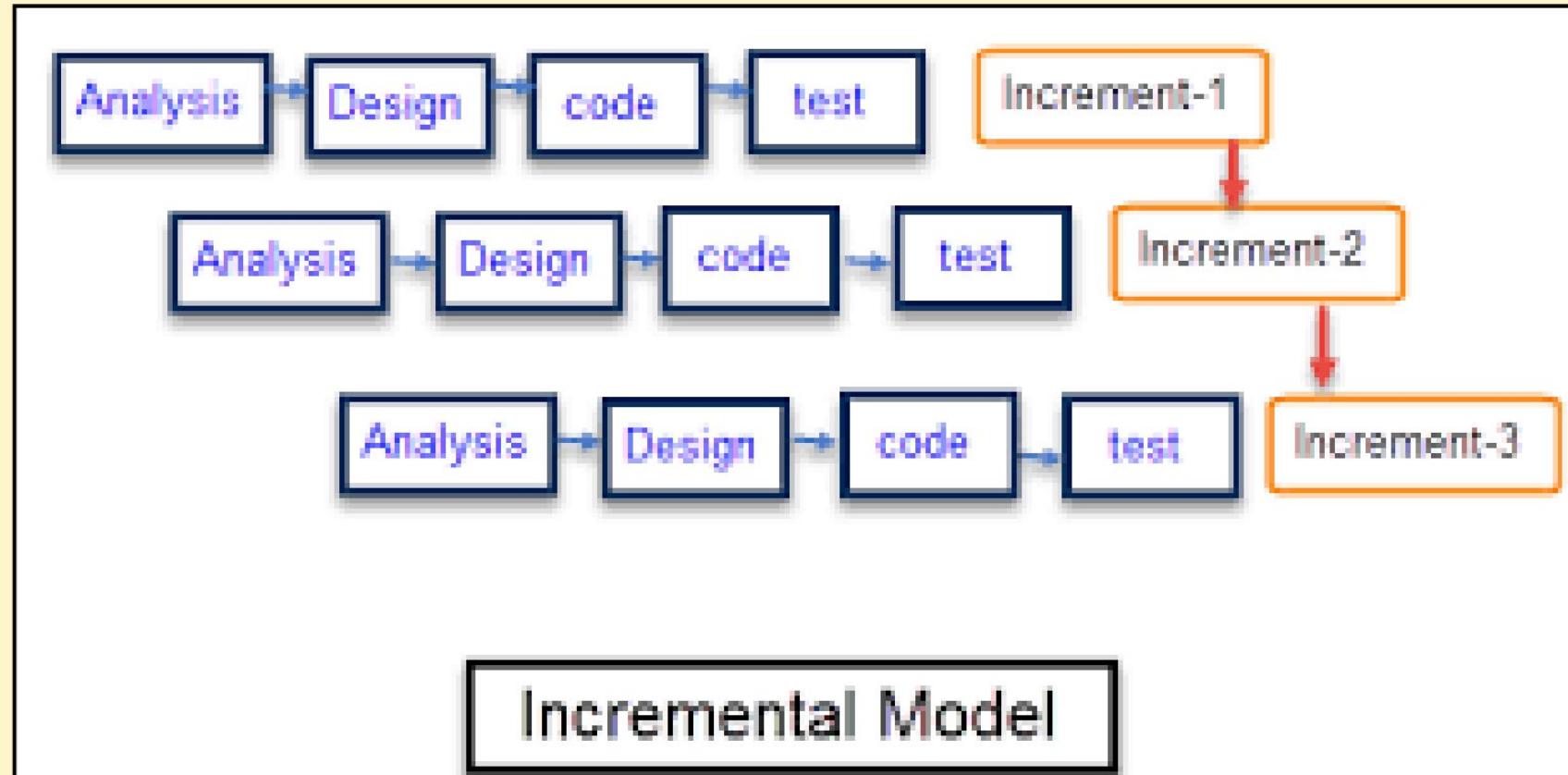
SDLC Models



- **1. Waterfall model**
- **2. Iterative Waterfall model**
- **3. V-Process model**
- **4. Incremental Process model**
- **5. Evolutionary Process models**
 - 5.1. Prototyping model
 - 5.2. Spiral model
- **6. Agile & RAD Models**
 - 6.1. Extreme Programming
 - 6.2. Scrum
 - 6.3. Crystal
- **Unified Process**



Incremental Model





(5) Evolutionary Process Models

- Software usually evolves over a period of time
- Business & product reqts often change as development progresses
- If the situation requires to release a limited version of the product to meet competitive or business pressure
 - or
- The core system requirements are well understood, but additional features are yet to be defined
- Evolutionary SDLC model is suitable
 1. Evolutionary models are iterative in nature
 2. They facilitate development of increasingly more complete versions of the s/w

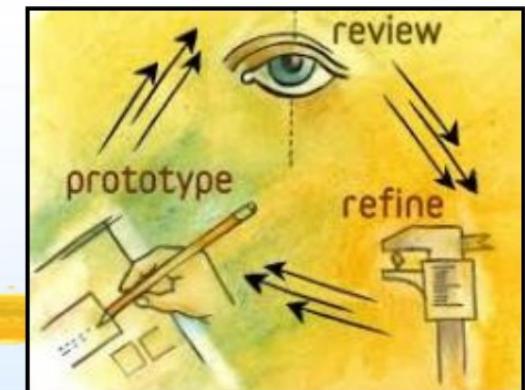


- 
- Two common evolutionary models are:

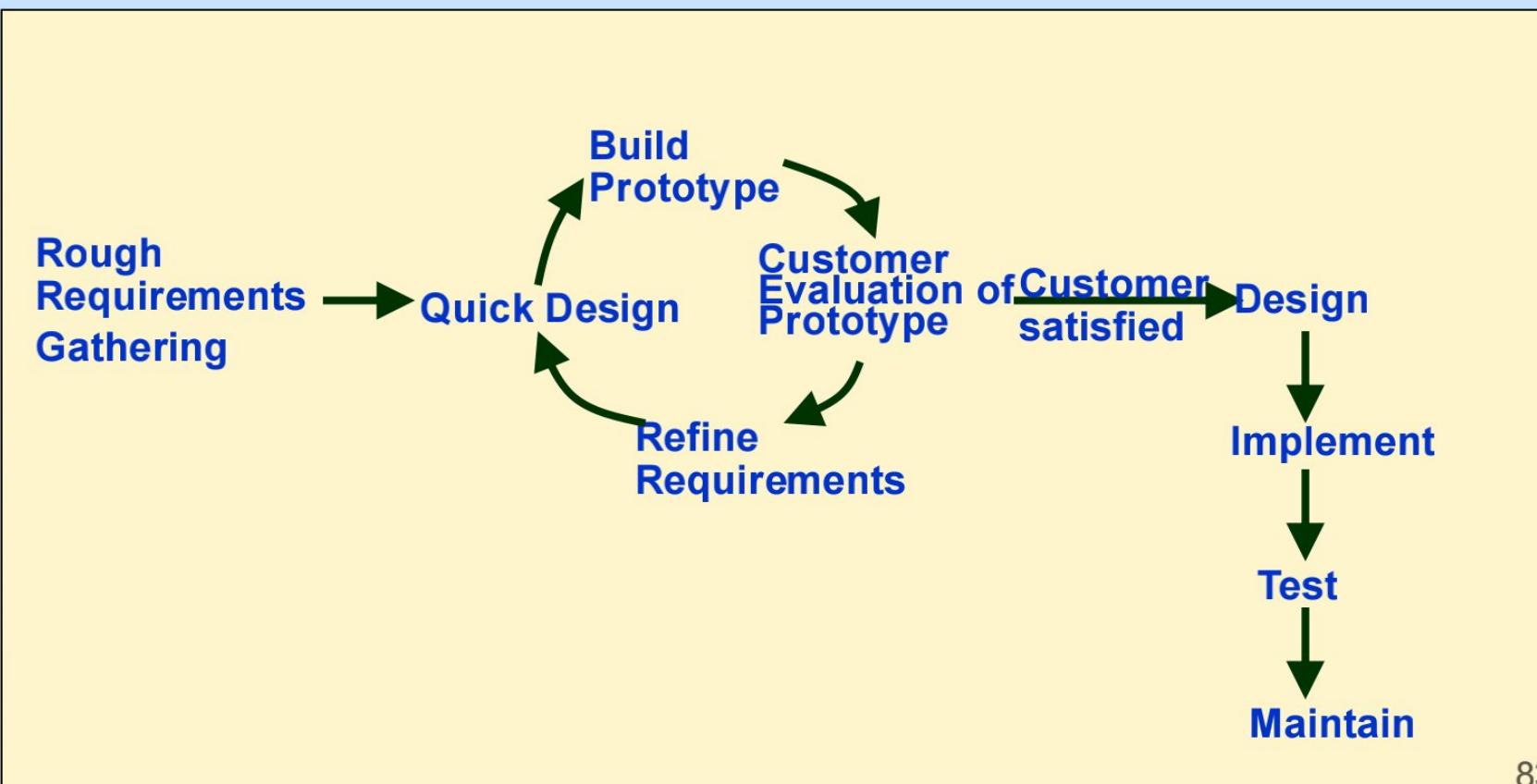
5.1. Prototyping model

5.2. Spiral model

(5.1) Prototyping Model



- Used for systems with :
 - Unclear user requirements
 - Unresolved technical issues



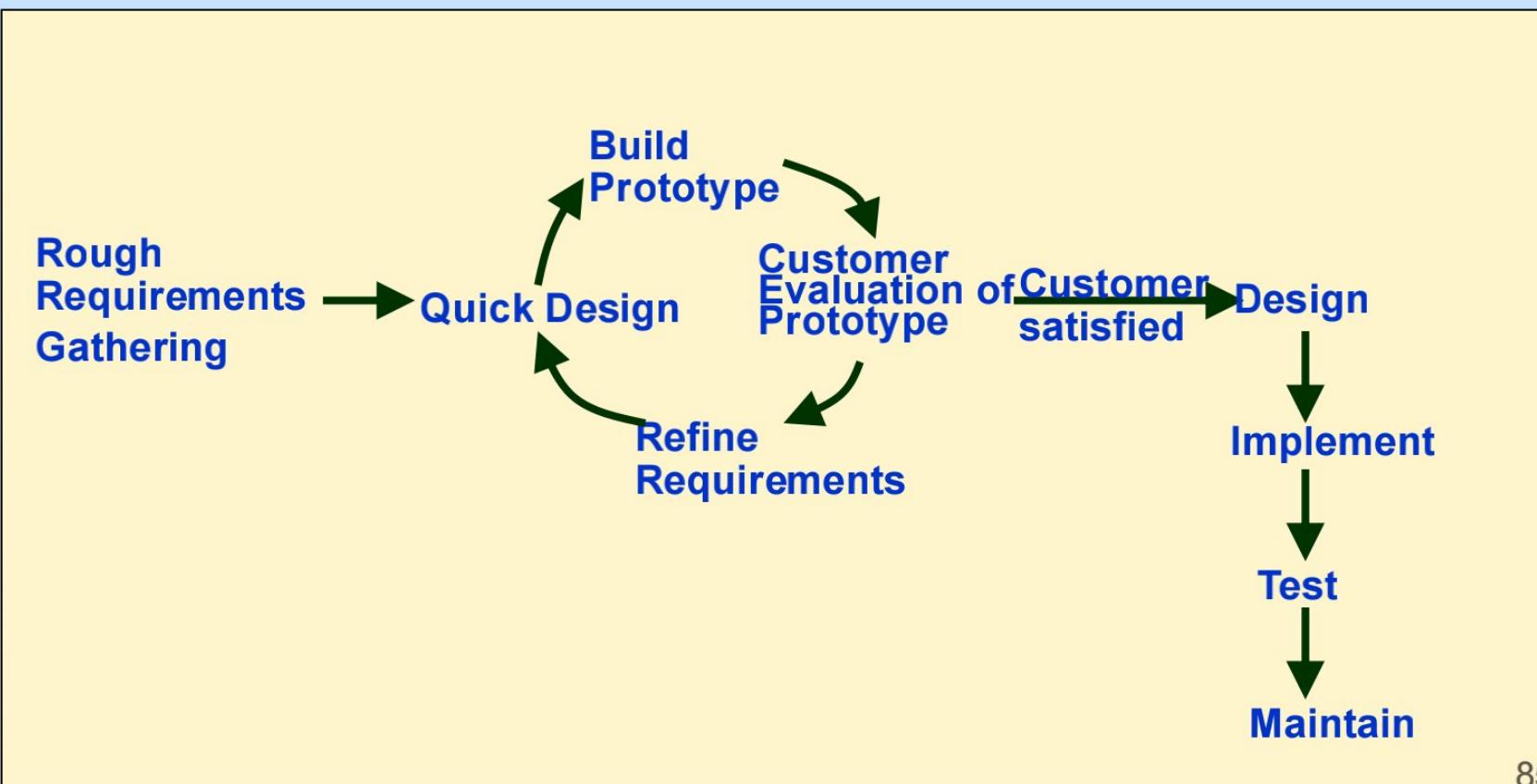
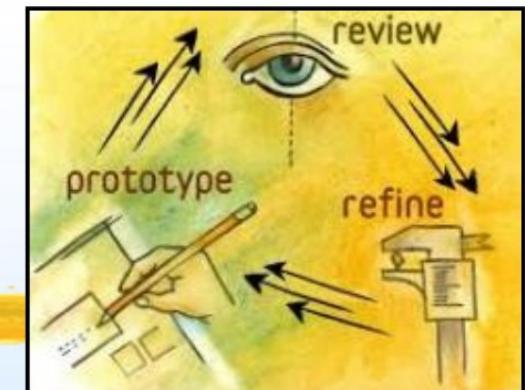


Prototyping Model



- Water fall model assumes that **requirements are completely known before development starts**
 - It's not the case always
- In such cases, before starting actual development,
 - A "working prototype" of the system should first be built
- A prototype is a toy implementation of a system with:
 - limited functional capabilities (dummy functions) , low reliability & inefficient performance
- It Illustrates the **system** to the **customer**
 - For providing **complete requirements**

(5.1) Prototyping Model





Prototyping Model Phases



- Start with **approximate or rough requirements**
- Carry out a **quick design**
- The **prototype** is submitted to **customer** for **evaluation**
 - Based on the user feedback, **requirements are refined**
 - This **cycle** continues until the user **approves** the prototype
- The actual system is developed using the **classical waterfall** approach

Prototyping Model Summary



- **Final working prototype** (with all user feedbacks) serves as an **animated requirements specification**
- The **prototype** is usually **thrown away**:
 - But, the experience gained **helps** with developing the actual product



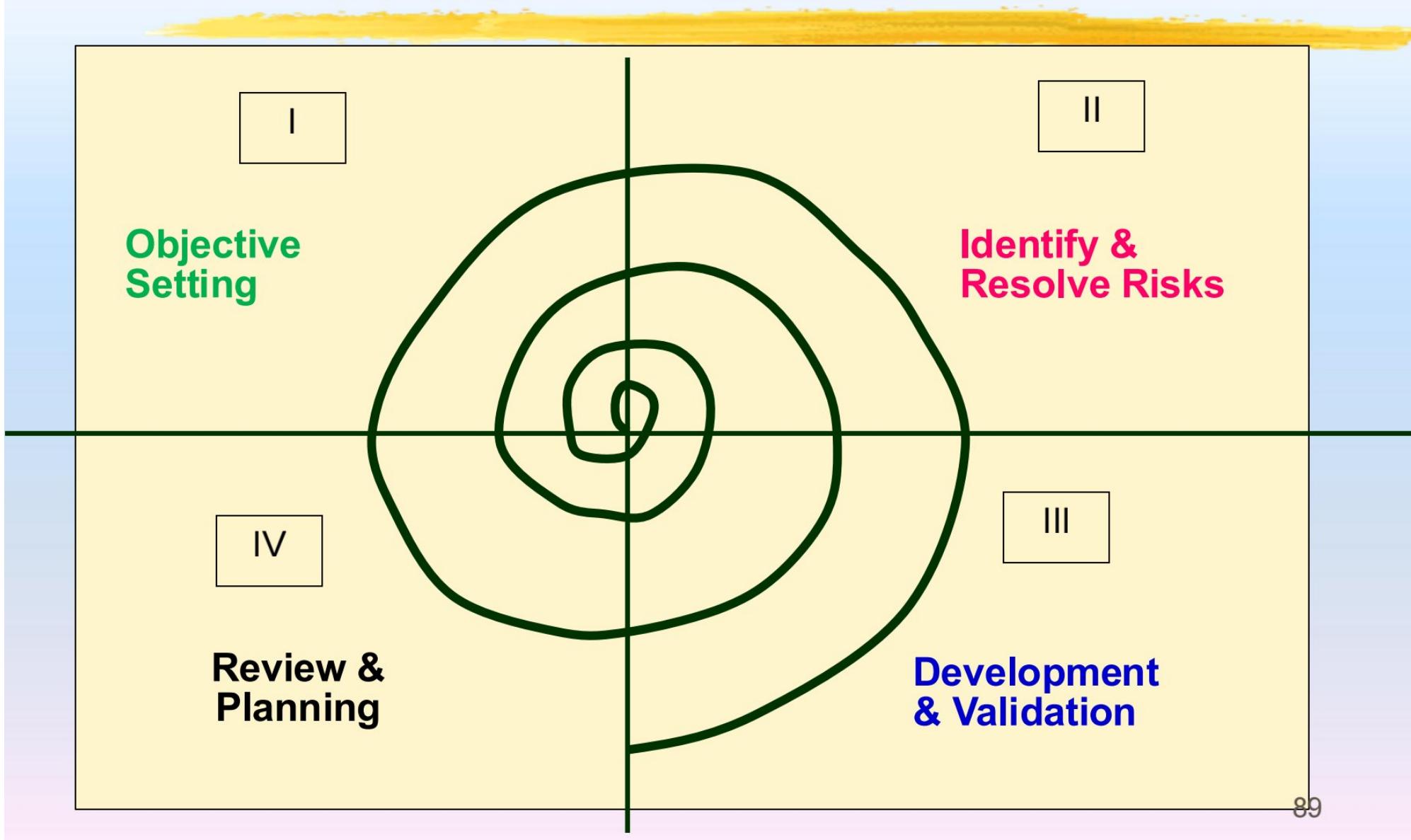
5.2. Spiral Model



- Proposed by **Barry Boehm** in **1988**
- This model appears like a **spiral with many loops**
- Each **loop** represents a **phase of the s/w dev. process**
 - The innermost loop may be feasibility study phase
 - Next loop: requirements Analysis phase
 - Next one: system design, and so on
- There are **no fixed number of loops** or **phases**
- The team decides: How to structure the project into phases

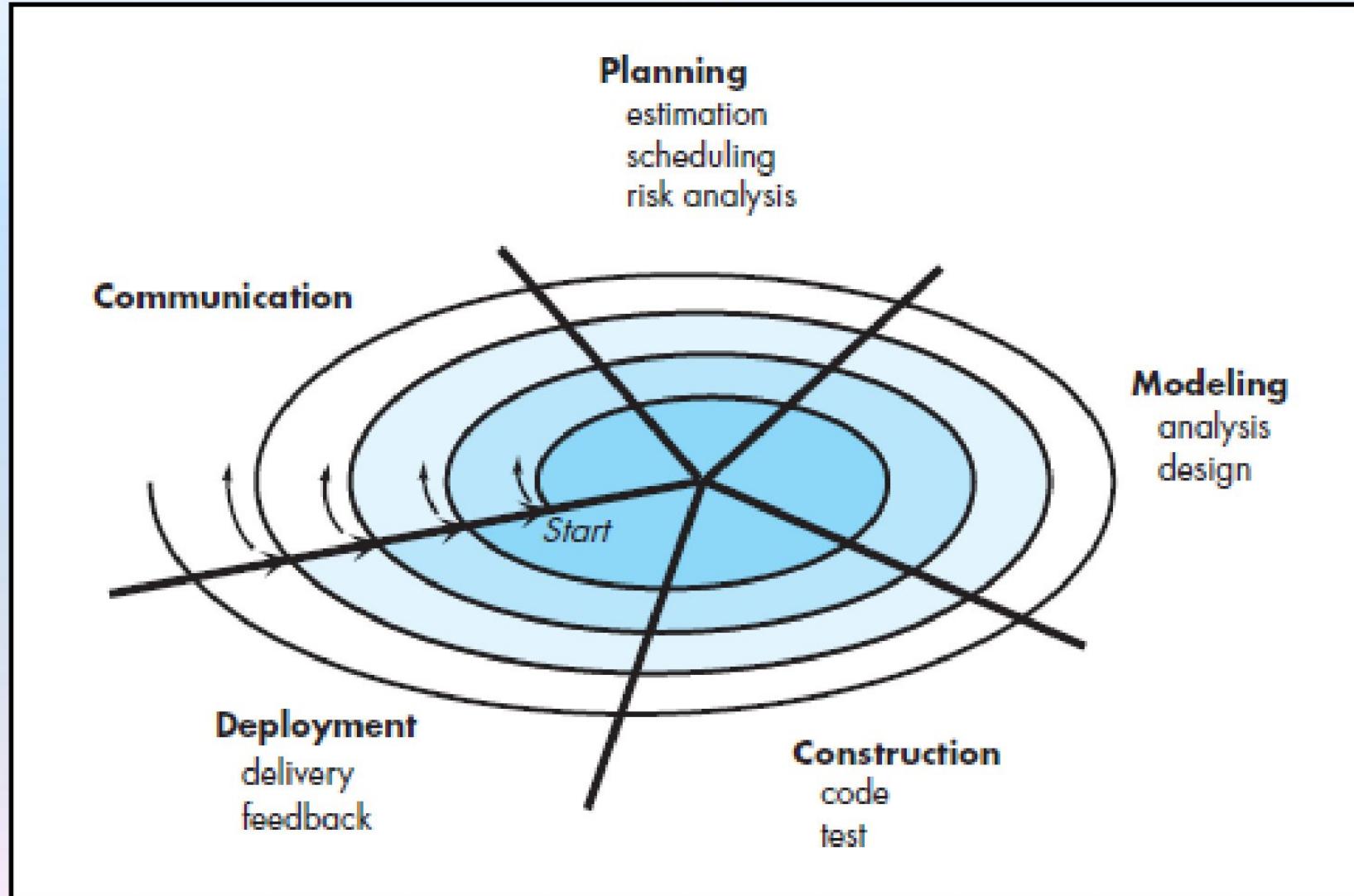


Spiral Model Graphical Representation





Spiral Model – alternate diagram





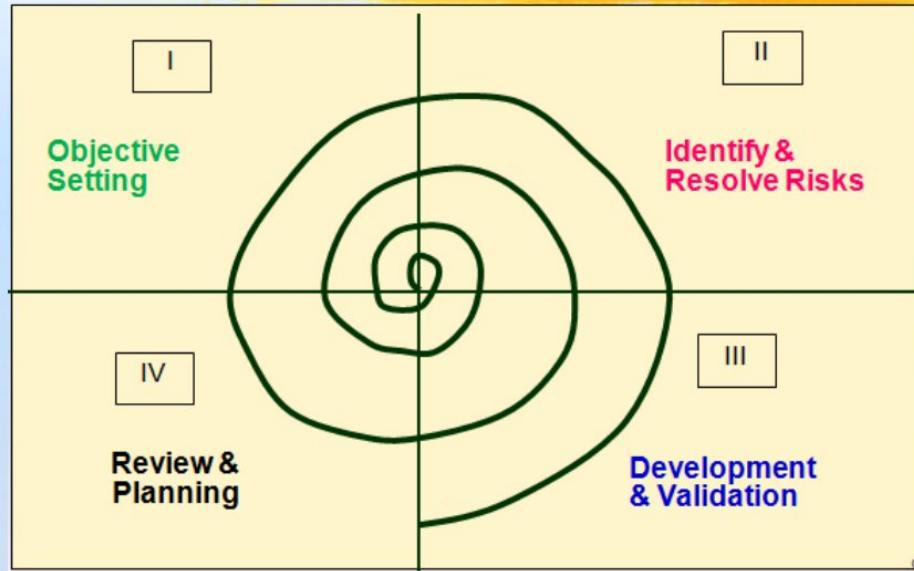
Activities Of Each Phase



- Each loop in the spiral is **split into 4 quadrants**
- The following activities are carried out in **each phase** :
 - **Objective Setting** (1st Quadrant)
 - **Identify & Resolve Risks** (2nd Quadrant)
 - **Development & Validation** (3rd Quadrant)
 - **Review & Planning** (4th Quadrant)



1. Objective Setting (First Quadrant)



- Identify **objectives of the phase**
- Identify **deliverables** for the phase (*SRS, Design docs ..*)

2. Identify & Resolve Risks (2nd Quadrant)



- Identify the **risks** associated with the objectives
 - **Risk:** Any adverse circumstance that might impact successful completion of a project
- **Analyze** each identified project risk
- Take steps to **resolve** or **reduce** the risk



Spiral Model



3. Development & Validation (Third quadrant):

- **Develop & validate** the product

4. Review & Planning (Fourth quadrant):

- **Review the results** with the **customer** & **plan the next iteration** in the spiral
- **With each iteration in the spiral:**
- **More complete version** of the s/w gets built



Adv.s & Disadv.s of Spiral Model

➤ ADV.s:

- Suitable for projects with many unknown risks
- More powerful than Prototyping model

➤ DISADV.s:

- Complex model to follow unless have experienced staff
- Not suitable for outsourced projects (as continuous risk assessment is needed)

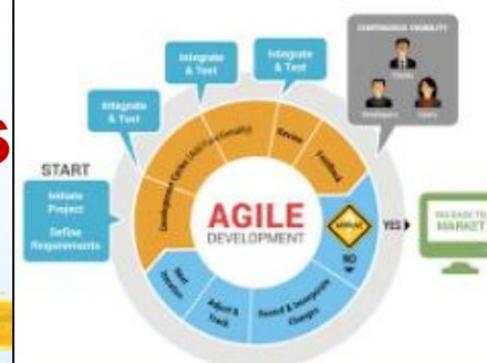


Spiral Model as a meta model

- Encompasses all discussed models:
 - A single loop of spiral represents a waterfall model phase
 - Uses an evolutionary approach
 - Iterations through the spiral are evolutionary levels
 - Uses Prototyping as a risk reduction mechanism

Write more details

6. Agile Development Models (Mid 90's)



- **Unsuitability** of Water Fall & Iterative models due to:
 - Late changes to requirements discouraged
 - No customer interaction allowed after Req. Stage
 - For customized s/w needing component reuse
- **Agile Models** help :
 - Avoid above shortcomings
 - to adapt to change requests quickly
 - Quick project completion by removing unnecessary activities

12 Agile Principles

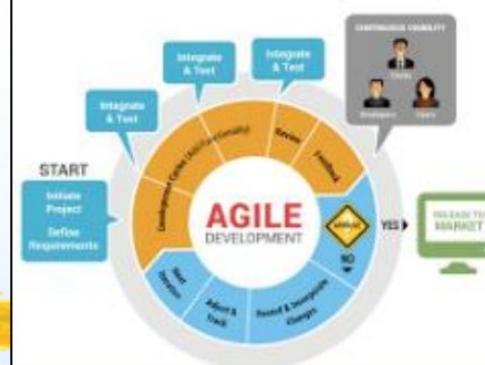
1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.

12 Agile Principles



8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile Development Models



➤ Features of Agile Models

- Reqs are divided into small parts for incremental development
- An iteration : approx. 2 weeks (time box)
- Delivery date is fixed
- Customer encouraged to give change requests
- Competent team
- Continuous customer interaction
- Pair programming

Adv.s & Disadv.s of Agile Methods

➤ ADV:

- Agility to requirement changes
- Highly effective if team members are competent
- Reduced development time & cost
- Elimination of overheads like formal docs & reviews

➤ DISADV:

- Lack of formal docs lead to confusion & maintenance issues
- Lack of review by external experts

Agile Development Models

➤ Popular Agile Models :

- Crystal
- Atern
- Scrum
- Extreme Programming (XP)
- Lean Development
- Unified process





6.1. Extreme Programming (XP)

(Kent Beck – 1999)

- Takes below best practices to extreme levels
 - **Code Review:** Pair programming for continuous review
 - **Testing:** Test driven dev. (TDD) – write code & execute TCs
 - **Incremental dev.** to Implement customer feedbacks
 - **Simplicity:** Start with the simplest approach, enhance later
 - **Design:** Continuously improve design
 - **Int. Testing:** Continuous integration (no. of times a day) 103

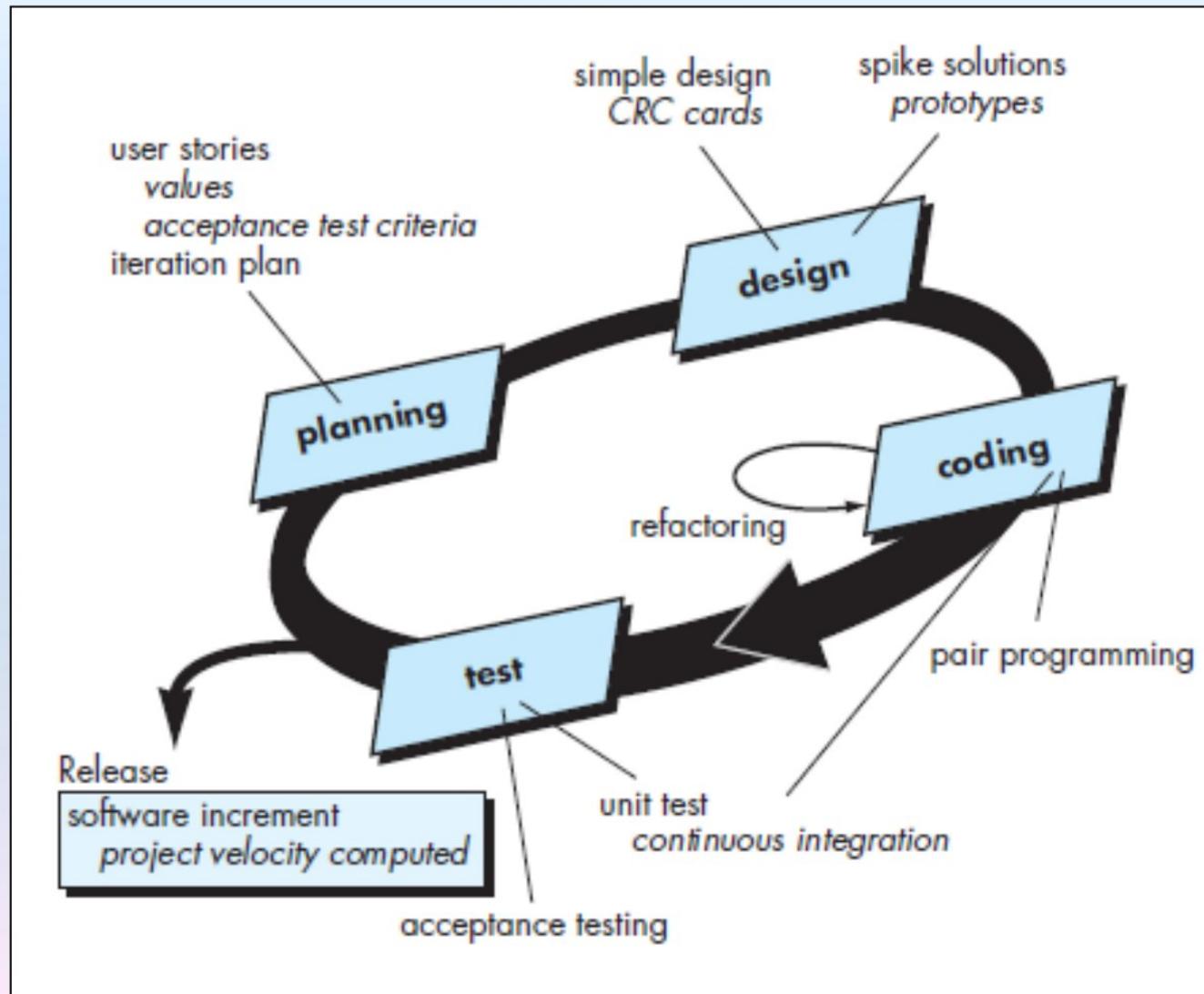
★ Extreme Programming (XP)

➤ Features of XP

- **Frequent releases** (Iterations): implement “user stories”
 - **User stories** : *Informal description of a feature by user*
 - **Ex:** *A Library member can issue a book if available*
- **Metaphors:** How the system should work
 - Spike : A simple solution
- **Coding** – Is the most important activity
- **Testing** – High importance given
- **Designing** – Effective simple design to be followed
- **Feedback** – Frequent feedback obtained from customer
- **Simplicity** – Build something simple that will work today



Extreme Programming (XP)



★ Extreme Programming (XP)

➤ **Applicability of XP**

- Projects with new tech.(research projects)
- Small projects

➤ **Unsuitability of XP**

- Projects with stable requirements
- Mission critical projects (need high reliability)



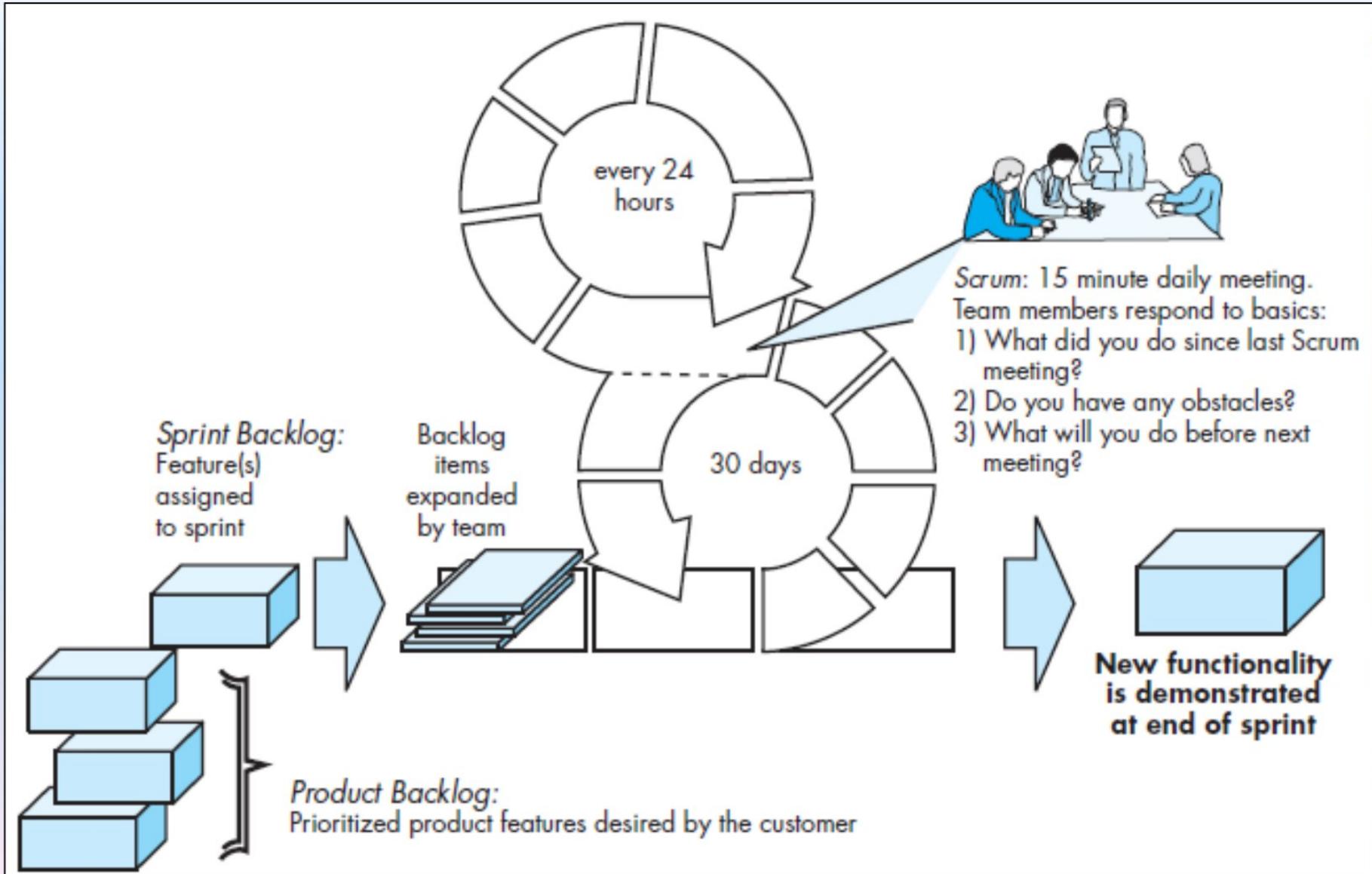
6.2. SCRUM Models

➤ Features of Scrum model

- Project divided into small parts – developed incrementally
- Time period of an increment – **Sprint** (*couple of weeks*)
 - Each Sprint will contain **multiple tasks** as decided by the **stakeholders**
- Team & stake holders **meet** after sprint to discuss progress
- Team members have **3 roles** :
 - **S/w owner** – communicates customer vision to team
 - **Scrum master** – liaison between s/w owner & team members
 - **Team member**



SCRUM Models



6.3. Crystal Methodology

- Developed by **Alistair Cockburn** in the **mid-1990s**
- Described as “**lightweight methodologies**”
- **Crystal methods are focused on:**
 - **People**
 - **Interaction**
 - **Community**
 - **Skills**
 - **Talents**
 - **Communications**

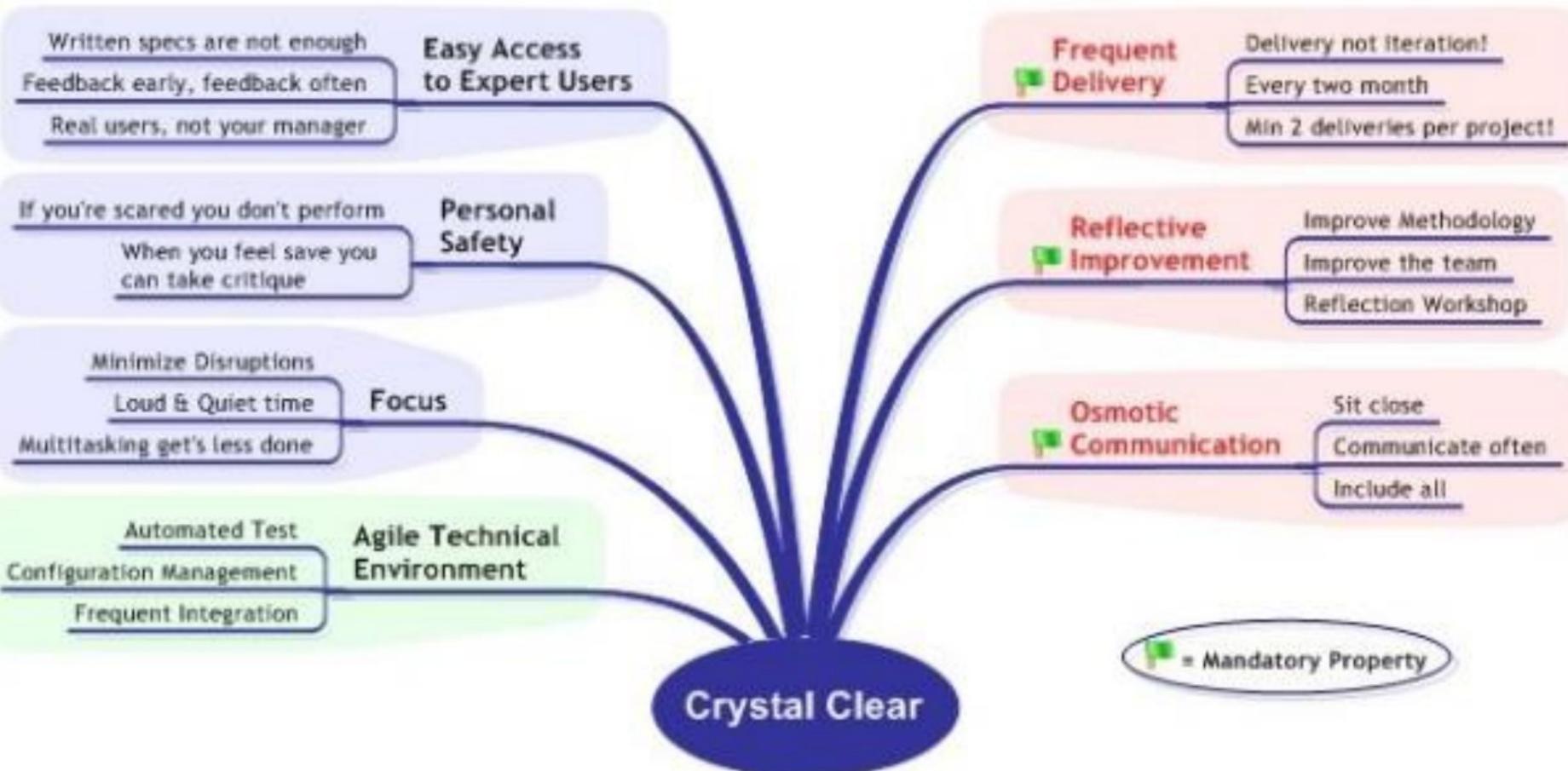
7 Properties of Crystal

➤ The seven properties are:

- Frequent delivery
- Reflective improvement
- Close or osmotic communication
- Personal safety
- Focus
- Easy access to expert users
- Technical env with automated tests, config mgmt & frequent integration

Crystal Model

The 7 Properties of Crystal Clear



Functionality of Crystal

1. **Crystal Clear-** The team consists of only 1-6 members that is suitable for short-term projects where members work out in a single workspace.
2. **Crystal Yellow-** It has a small team size of 7-20 members, where feedback is taken from Real Users. This variant involves automated testing which resolves bugs faster and reduces the use of too much documentation.
3. **Crystal Orange-** It has a team size of 21-40 members, where the team is split according to their functional skills. Here the project generally lasts for 1-2 years and the release is required every 3 to 4 months.
4. **Crystal Orange Web-** It has also a team size of 21-40 members were the projects that have a continually evolving code base that is being used by the public. It is also similar to Crystal Orange but here they do not deal with a single project but a series of initiatives that required programming.
5. **Crystal Red-** The software development is led by 40-80 members where the teams can be formed and divided according to requirements.
6. **Crystal Maroon-** It involves large-sized projects where the team size is 80-200 members and where methods are different and as per the requirement of the software.
7. **Crystal Diamond & Sapphire-** This variant is used in large projects where there is a potential risk to human life.

- **Benefits of using the Crystal Agile Framework :**
 - Facilitate and enhance team communication and accountability.
 - The adaptive approach lets the team respond well to the demanding requirements.
 - Allows team to work with the one they see as the most effective.
 - Teams talk directly with each other, which reduces management overhead.
- **Drawbacks of using the Crystal Agile Framework :**
 - A lack of pre-defined plans may lead to confusion and loss of focus.
 - Lack of structure may slow down inexperienced teams.
 - Not clear on how a remote team can share knowledge informally.

★ 6.4. RAD Model (Agile, XP & Scrum) (early 90's)

➤ Goals of RAD model

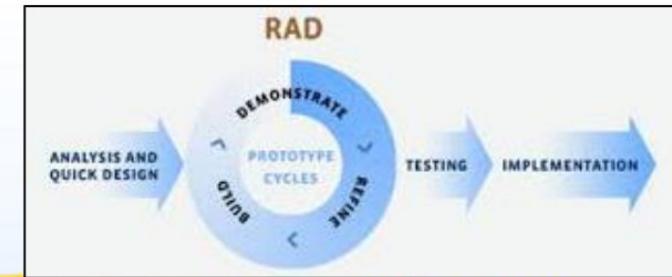
1. Decrease time/cost of building s/w systems
2. Reduce communication gap between customer & developers

➤ **RAD** (Rapid Application Development) **Model :**

- Combines features from Prototyping & Evolutionary Models
- Prototypes are first constructed
 - But not thrown away, used in system construction later
- Features are developed incrementally & delivered to customer



RAD & Agile concepts

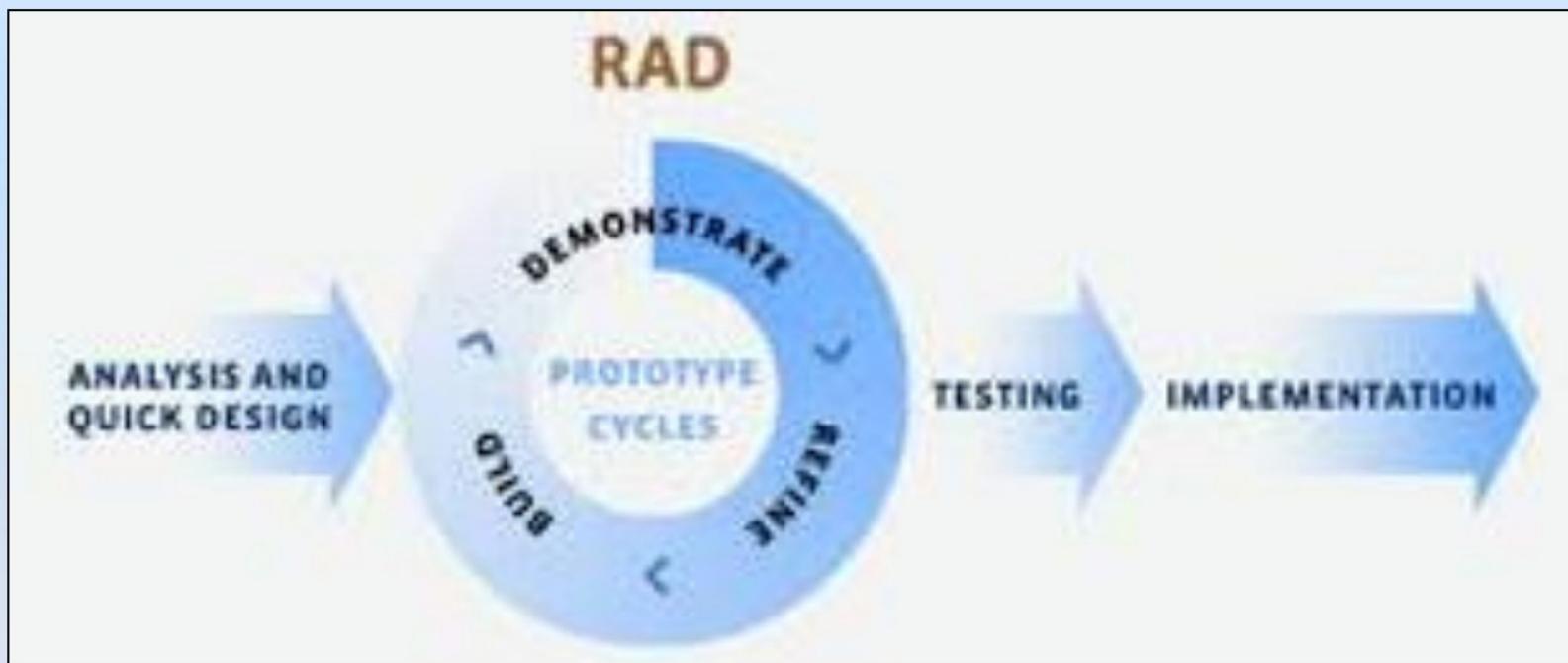


Agile models and **RAD** has many features in common.

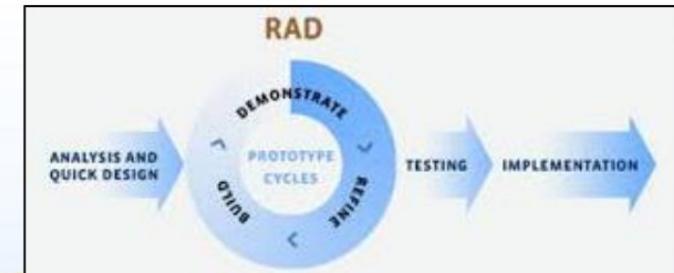
Both models:

- Break the product into small increments (time box) & deliver iteratively
- Focus on adaptability to customer requirement changes
- Aim at reducing time & cost by rapid delivery of working s/w by doing :
 - Minimal planning
 - Heavy reuse of code by rapid prototyping

RAD Model



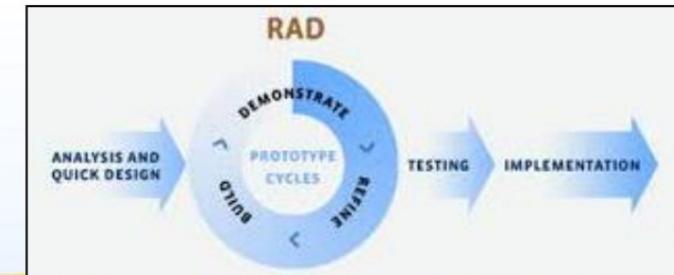
RAD Model



➤ Working of RAD Model :

- Development takes place in short iterations (**Time box**)
- **During each time box :**
 - Quick prototype is made & refined based on customer feedbacks
 - Dev. Team contains customer rep.s to bridge communication gaps
- Customer is encouraged to give change requests

RAD Model



➤ RAD Model is applicable to:

- Customized products
- Non-critical & large s/w
- Product with tight schedule

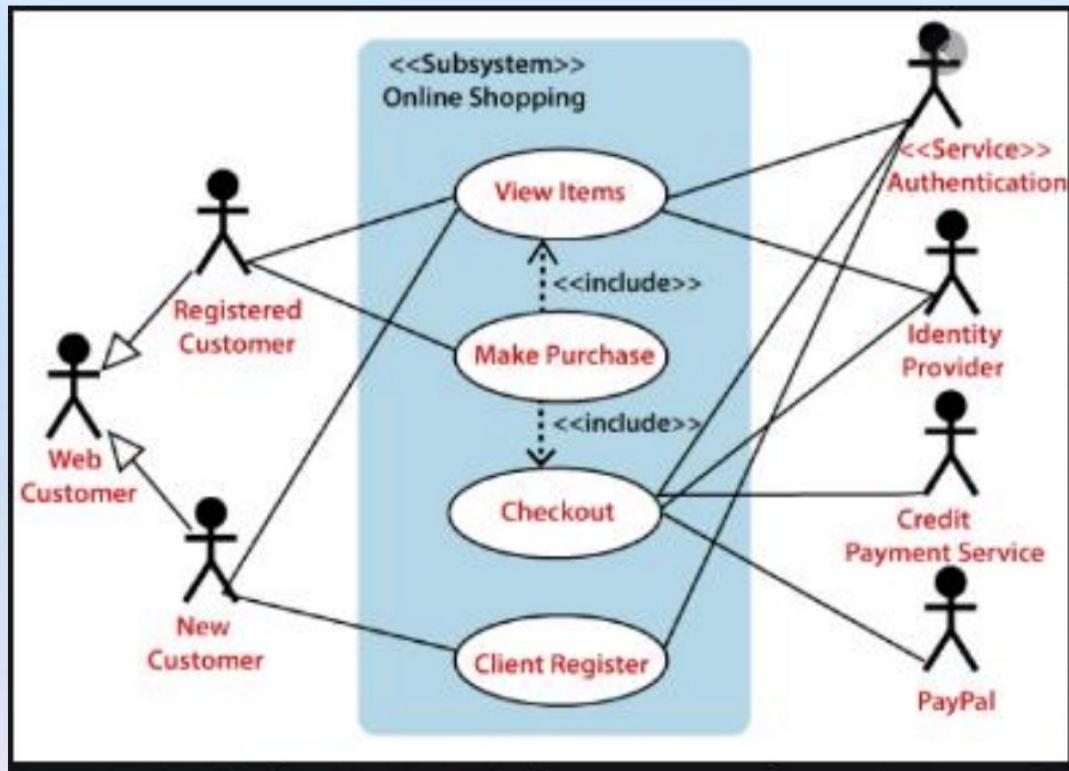
➤ RAD Model is unsuitable for :

- Generic products (wide distribution)
- Products needing reliability
- Monolithic s/w (small – difficult to divide)

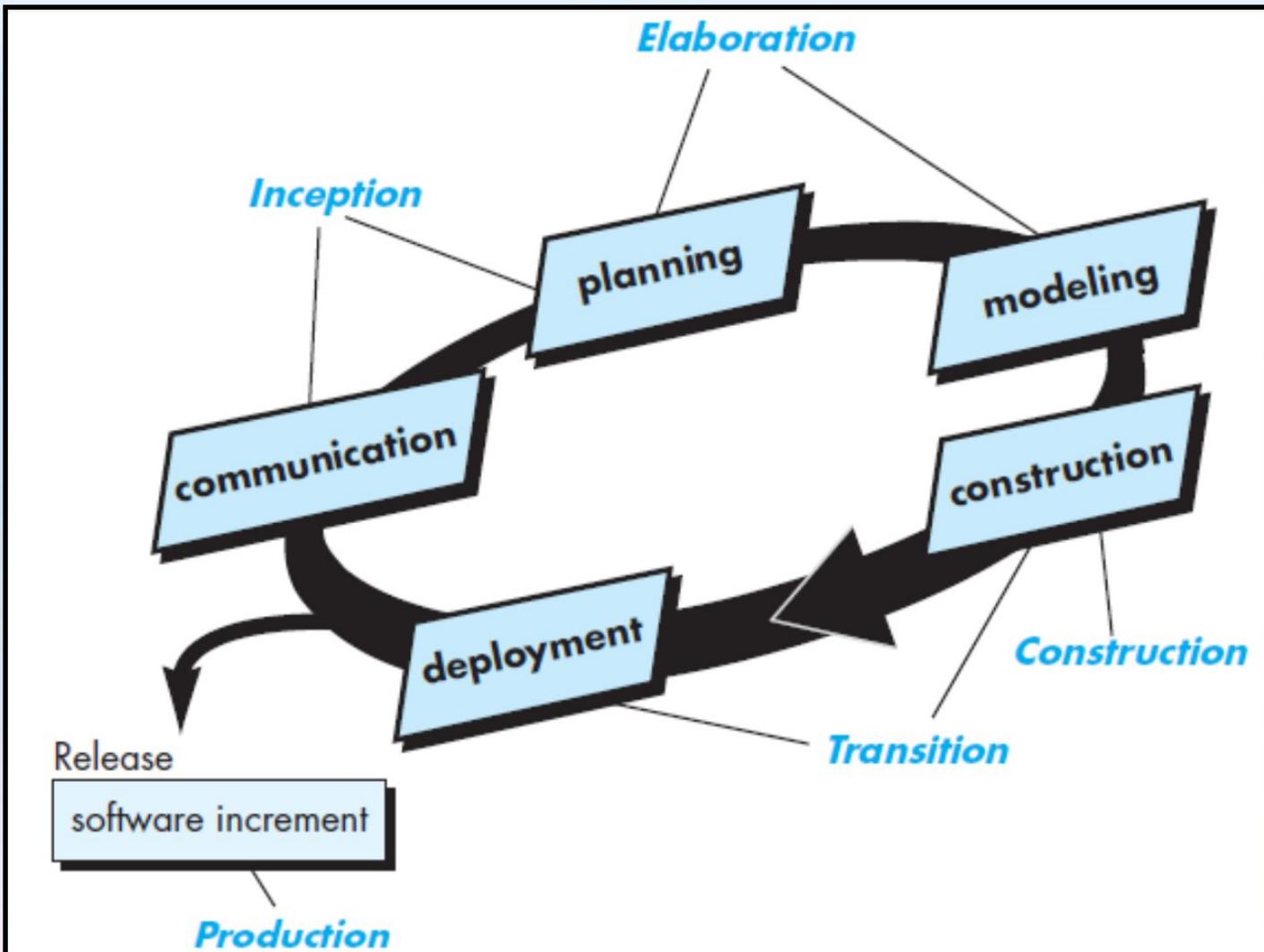
7. Unified Process

- Proposed by Rumbaugh, Booch & Jacobson in 1990's
- The **Unified Process** derives the best features of traditional s/w process models & implements many principles of agile s/w development
- It gives importance to customer communication & describing the customer's view of a system (the **use case**)
- Emphasizes the importance of **s/w architecture**

Use case diagram



Unified Process - diagram



Phases of Unified Process

1. Inception phase -

- Consists of customer communication & planning activities
- Business requirements are identified
- A rough architecture for the system proposed
- Business requirements are described using a set of use cases

2. Elaboration phase -

- Consists of the communication & modelling activities
- Refines & expands the preliminary use cases developed earlier
- Architectural representation to includes 5 different views of the s/w
 - Use case, Analysis, Design, Implementation & Deployment models*

Phases of Unified Process

3. Construction phase –

- The s/w components are developed using architectural model as input
- Use cases are made operational
- Features for the s/w increments are then implemented in source code
- Integration & Acceptance testing done

4. Transition phase -

- Contains latter stages of construction & 1st part of the deployment
- Software is given to end users for beta testing & defects are fixed
- Support information is created

5. Production phase -

- Ongoing use of the software is monitored
- Support for the operating env is provided
- Defect reports & requests for changes are submitted & evaluated

★ Comparison of Different Life Cycle Models

- **Iterative waterfall model**
 - most widely used model
 - But, suitable only for **well-understood problems**
- **Prototype model is suitable for **projects not well understood from below aspects :****
 - user requirements
 - technical aspects

★Comparison of Different Life Cycle Models

- **Evolutionary model is suitable for large problems:**
 - That can be decomposed into a set of modules that can be incrementally implemented
 - incremental delivery is acceptable to the customer
- **The spiral model:**
 - Suitable for development of technically challenging s/w products that are subject to several kinds of risks

★Comparison of Different Life Cycle Models

- **RAD model is suitable for** customized, non-critical & large s/w products with tight schedule & **unsuitable** for small, generic, high reliability products
- Unlike **Prototyping** model, the prototypes are not thrown away
- Unlike **Iterative WF** model, functionalities are developed incrementally not together with heavy code/design reuse
- Unlike **Evolutionary** model, each increment is shorter & builds a prototype not systematic development

★Comparison of Different Life Cycle Models

- Agile models are suitable for customized, large s/w products that can be released in increments & unsuitable for stable requirement, mission-critical products
- Unlike RAD model, developing prototypes are not recommended
- In Agile model, only completed work is demonstrated to customer
- Unlike Iterative WF model, if the project is cancelled mid-way, still some functionality is implemented
- Progress is measured in terms of functionalities delivered



Entry and Exit criteria for a particular phase of software life-cycle

- For the **testing phase** of “Library Information System – Renew Book” module:
- **Entry criteria:**
 - All the sub-modules/programs for the “Renew Book” module are coded & unit tested successfully
 - Test suite containing all test cases for Black-Box & White-Box testing are written
- **Exit criteria:**
 - All the sub-modules/programs for the “Renew Book” module are tested by executing all test cases of the test suite
 - All defects discovered are fixed



Example systems & model suitability (justify)

- Nuclear Plant Cooling System
- Medical Diagnostic System
- Student Info Mgmt System
- Animated computer game system
- ERP System
- Payroll System



Example systems & model suitability (justify)

- Nuclear Plant Cooling System ([V-Process model](#))
- Medical Diagnostic System ([V-Process model](#))
- Student Info Mgmt System ([Prototyping model](#))
- Animated computer game system ([Prototyping model](#))
- ERP System ([Agile model](#))
- Payroll System ([Iterative waterfall model](#))





Possible questions

Q: Suggest a suitable life-cycle model for development of a **ECG** (*Electrocardiography*) Monitoring System. Justify your choice.

A: V-process model will be suitable. Because, this system needs to be highly reliable and fault-proof. It is important to validate and test all the components thoroughly in all stages of software development.

Justify this in detail

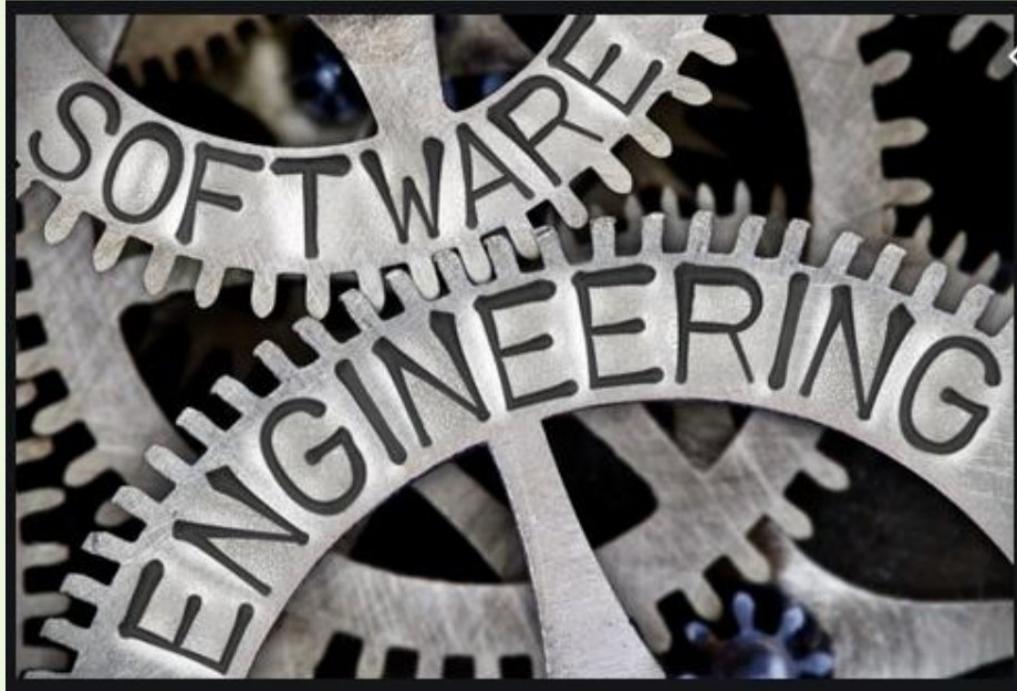
Q: Suggest a suitable life-cycle model for development of a **gaming application**. Justify your choice.

A: Prototyping model will be suitable as it is important to get all requirements clearly from customer before building the actual system.

Justify this in detail

Software Engineering

Module-2



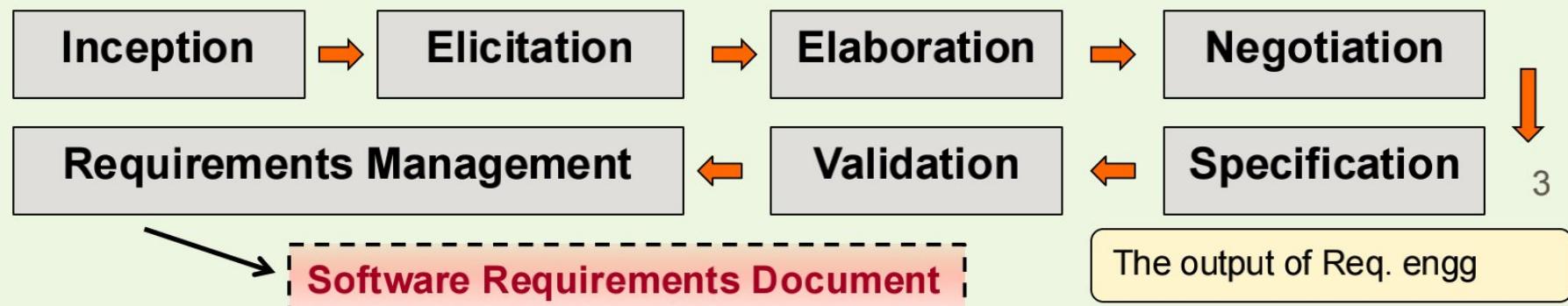
Faculty :
Dr. Suchismita Rout
Associate Professor

Module2 Contents

- **Requirements Engineering:**
 - Types of Requirements, Functional and non-functional requirements
 - The software requirements document
 - Requirements - specification, engineering processes, elicitation & analysis, validation, and management
- **Decision Trees & Decision Tables**
- **Formal Specification**

Requirements Engineering

- The broad spectrum of **tasks & techniques** used to understand the **system requirements** is called **requirements engineering**
- Begins during the **communication** activity & Continues into the **modeling** activity
- Must **adapt** to the needs of the **process**, the **project**, the **product** & the **people** involved
- **Requirements engg** builds a **bridge** to design & construction
- **Stages of Req. Engg.** - Inception, Elicitation, Elaboration, Negotiation, Specification, Validation & Requirements Management



Stages of Requirements Engineering

1. Inception

- Most **projects begin** when a business need is identified
- **Stakeholders** from the business community define a business case
- **Rough feasibility analysis** is done
- Working description of the **project's scope** is created

2. Elicitation

- **Business goals** are established
- **Stakeholders** share their goals honestly
- **Prioritization mechanism** for goals is established
- Potential **architecture** to meet stakeholder goals is created
- **Scope problems** occur when system boundary is ill-defined
- The **requirements-gathering** is started in an organized manner 4

Stages of Requirements Engineering

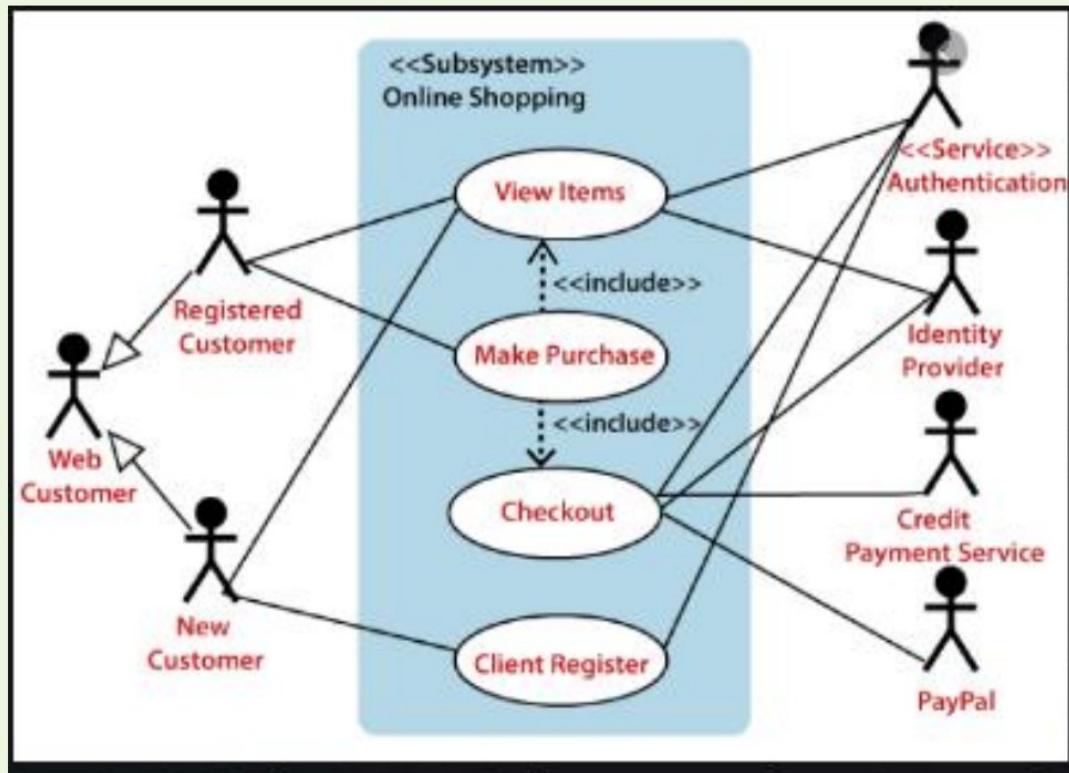
3. Elaboration

- Expansion & refinement of requirements model including various aspects of s/w function based on info obtained from customer
- Creation & refinement of user scenarios describing how the end user (actors) will interact with the system (use cases)
- Identification of relationships between classes & creation of a variety of supplementary diagrams (*class diagrams, activity diagrams etc..*)

4. Negotiation

- Customers sometimes ask for more than that can be achieved & propose conflicting requirements
- Reconcile these conflicts through a process of **negotiation**
- Discuss "conflicts in priority" & resolve them
- Achieve as much stakeholder satisfaction as possible

Use case diagram



Stages of Requirements Engineering

5. Specification

- Is a written document / a set of graphical models / a formal model / a collection of usage scenarios / a prototype / a combination of these
- A “Standard template” or a flexible format can be used for a specification depending upon the need
- For large systems, a written document, combining natural language descriptions & graphical models may be used

6. Validation

- Requirements validation ensures that all s/w requirements are stated unambiguously
- Inconsistencies, omissions & errors have been detected & corrected
- A review team (including s/w engineers, customers, users & other stakeholders) does a “tech. review” of the requirements

Stages of Requirements Engineering



7. Requirements Management

- Requirements for computer-based systems change throughout the life of the system
- Requirements mgmt is a set of activities that help the project team identify, control & track requirements & changes to requirements as the project proceeds
- Stakeholders & s/w engineers work together on this like part of the same team



Requirements Analysis & Specification

Contents

- Introduction
- Requirements gathering & analysis
- Requirements specification
- SRS document
 - Different Sections in SRS
 - Good & Bad Properties of SRS
- Formal Specification

Requirements Analysis & Specification

- ✓ Many projects fail:
 - ✓ Because they start implementing the system:
 - ✓ Without clearly understanding **what the customer exactly wants**
- ✓ That's why it is important to learn:
 - ✓ Requirements gathering, analysis & specification techniques thoroughly

Requirements Analysis & Specification

- activities

- **Consists of 3 main activities:**

1. Requirement Gathering

- Collection of all the data regarding the system to be developed

2. Requirement Analysis

- Remove all inconsistencies & anomalies from the requirements

3. Requirement Specification

- Systematically organize requirements into a Software Requirements Specification (SRS) document

1. Requirements Gathering

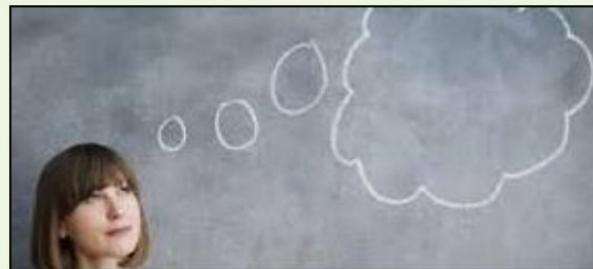


- If the project is to automate an existing system
 - **Ex:** *Automating existing manual payroll activities*
- The task of the system analyst is a little easier
- Analyst can immediately obtain:
 - Input & output formats
 - The operational procedures

Requirements Gathering (CONT.)



- In the **absence** of a working system
 - Lot of imagination & creativity are required to gather requirements from the scratch



- Interacting with the customer to gather relevant data through interviews, meetings, workshops, email/phone communications, surveys, questionnaires..

2. Analysis of Gathered Requirements



- After gathering all the requirements, analyze it to:
 - Clearly understand the user requirements
 - Detect Inconsistencies, anomalies & incompleteness
 - Resolve through further discussions with the customer



Anomaly

- An **anomaly** is an **ambiguity** in the requirement:
- **Examples:** In case of a "**temperature control system**"
 - Customer says turn off heater when temperature is **very high**



Inconsistent requirement

- Inconsistent requirement means some part of the requirement:
 - Contradicts with other parts
- **Example:** In case of a “temperature control system”
 - One customer says turn off heater and open water shower when temperature > 100 C
 - Another customer says turn off heater and turn ON cooler when temperature > 100 C



Incomplete requirement

- Some important parts of the requirements have been **omitted** :
 - due to oversight or
 - due to lack of clarity
- **Example:**
- In case of a "**temperature control system**"
 - The analyst has **not recorded**:
when temperature **falls below 90 C**:
 - Heater should be **turned ON**
 - Water shower turned OFF

3. Software Requirements Specification



- Main aim of requirement specification:
 - Systematically organize the requirements
 - Document requirements properly

Software Requirements Specification



- The SRS document is useful in various contexts:
- It serves as:
 - Statement of user needs
 - Contract document
 - Reference document
 - Definition for implementation



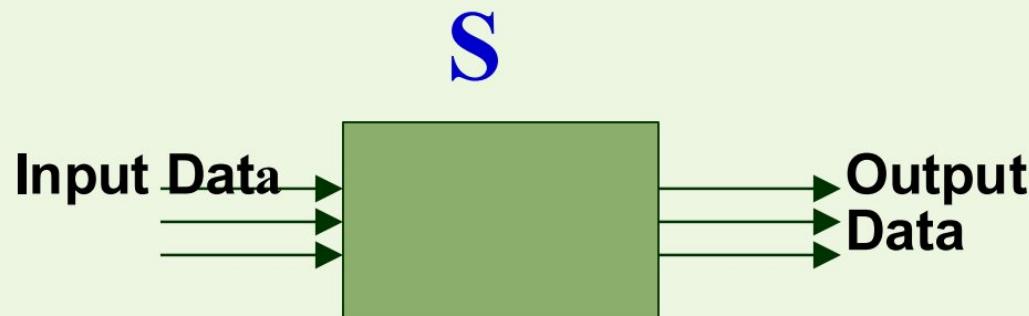
Software Requirements Specification: A Contract Document

- SRS document is not only a ***reference*** document :
- It is also a **contract** between the **development team** and the **customer**
 - Once the SRS document is **approved by the customer**,
 - Any subsequent **controversies are settled** by ***referring*** the **SRS document**



SRS Document (CONT.)

- The SRS document is known as black-box specification because :
 - It's internal details are not documented
 - Only its visible external behaviour (i.e. input/output) is documented



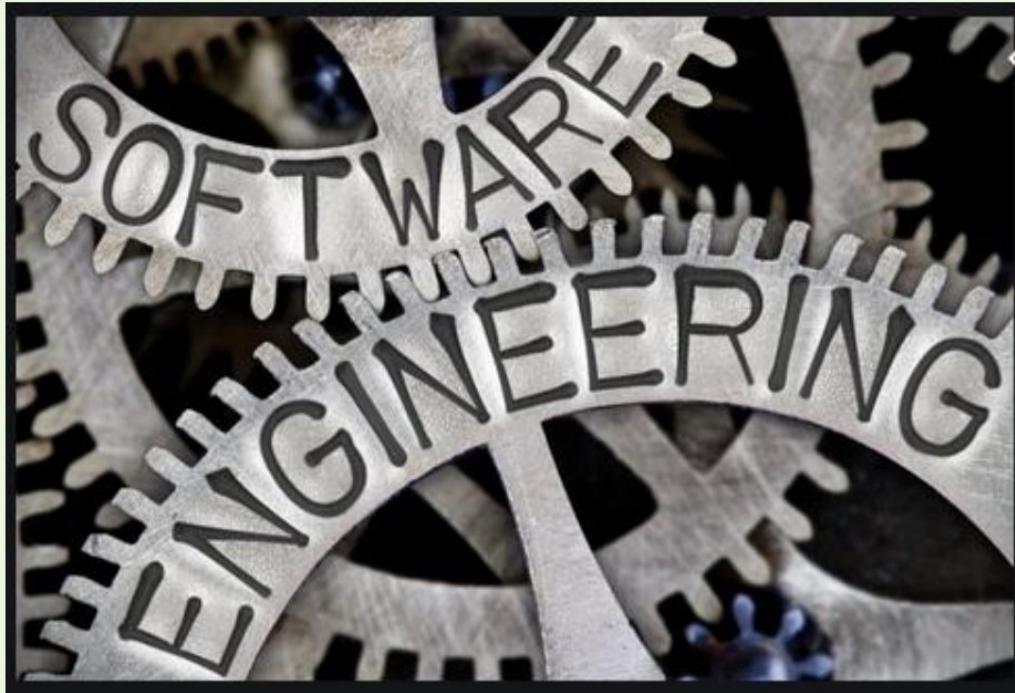
SRS Document (CONT.)



- SRS document concentrates on:
 - What needs to be done
 - Carefully avoids the solution ("how to do") aspects
- The requirements are documented:
 - Using end-user terminology

Software Engineering

Module-2



Faculty :
Dr. Suchismita Rout
Associate Professor

Module2 Contents

- **Requirements Engineering:**
 - Types of Requirements, Functional and non-functional requirements
 - The software requirements document
 - Requirements - specification, engineering processes, elicitation & analysis, validation, and management
- **Decision Trees & Decision Tables**
- **Formal Specification**



Properties of a good SRS doc

- It should be concise
 - At the same time should not be ambiguous
- It should specify what the system must do
 - Not say how to do it
- It should be well-structured & Easy to change
- It should be consistent & complete



- 
- It should be traceable
 - One should be able to trace which part of the specification corresponds to which part of the design and code and vice versa
 - It should be verifiable
 - **Ex.** “system should be user friendly” is not verifiable



Properties of Bad SRS Documents

➤ Unstructured Specifications:

- Narrative essay one of the worst types of specification document:
 - Difficult to change, be precise, be unambiguous,
 - Has scope for contradictions



➤ Noise:

- Presence of text containing information **irrelevant** to the problem

➤ Silence:

- aspects important to proper solution of the problem are omitted

➤ Overspecification:

- Addressing “how to” aspects
- Over specification restricts the **solution space**

➤ Contradictions:

- Contradictions might arise
 - if the same thing described at several places in different ways



➤ Ambiguity:

- Unquantifiable aspects, e.g. "good user interface"

➤ Forward References:

- References used earlier but defined only later on in the text
- *Ex: using abbreviated terms like GUI & MIS etc. in the earlier parts of SRS but defining them later.*

➤ Wishful Thinking:

- Descriptions of aspects for which realistic solutions will be hard to find
- *Ex: The complex logic should be implemented in most simple manner or the user Interface should be the best.*

★ Main Components of SRS Document

- SRS document, mainly **contains** :
 - Introduction (*Problem statement in summarized form*)
 - Goals Of Implementation (*Describes the benefits offered to the stakeholders*)
 - Functional requirements (*Detailed description of each functional element of the system including inputs, outputs & processing*)
 - Non-functional requirements (*performance, interface, reusability, security, usability, maintainability etc..*)
 - Constraints on the system (*H/W, S/W, OS to be used, Standards compliance etc..*)



Functional Requirements

- Functional requirements describe:
 - A set of high-level requirements
 - Each high-level requirement:
 - Takes some input data from the user
 - Outputs some data to the user
 - Each high-level requirement:
 - Might consist of a set of identifiable functions

Functional Requirements

- For each high-level requirement:
 - Every **function** is described in terms of
 - **input** data set
 - **output** data set
 - **processing** required to get the output data set

Example Functional Requirements

➤ List all functional requirements with proper numbering

➤ **Req. 1: SEARCH BOOK**

➤ User selects the “search” option,
➤ he is asked to enter the key words

➤ The system should output details of all books
➤ whose title or author name matches any of the key words entered.
➤ Details include: Title, Author Name, Publisher name, Year of Publication, ISBN Number, Catalogue Number, Location in the Library.



Example Functional Requirements

➤ Req. 2: **RENEW BOOK**

- When the “**renew**” option is selected,
 - the user is asked to enter his *membership number and password*
- After password validation,
 - the list of the books borrowed by him are *displayed*
- The user can renew any of the books:
 - by clicking in the corresponding *renew box*



Non-functional Requirements

- Non-functional Requirements are those characteristics of the system which can not be expressed as functions:
- Some examples of NF requirements are
 - Performance
 - Portability
 - Security
 - Usability
 - Reusability
 - Reliability
 - Interface
 - Maintainability etc.



Possible questions

Q: Classify the following as functional or non-functional requirement.

- Response Time of a Web Page (**Non- Functional**)
- Renew Book (**Functional**)
- Login (**Functional**)
- Authentication (**Non- Functional**)

Q: Using examples differentiate between functional and non-functional requirements.

- **Examples of F.R –** *in Library Info. System - Renew book, Create & cancel membership parts*
- **Examples of NF.R –** *in Railway Reservation System – Info. Inquiry & Ticket booking response time, authentication parts*

Constraints

- Constraints describe things that the *system should or should not do*
- Ex
 - H/W, S/W, OS to be used
 - Standards compliance

Examples of constraints

- Hardware to be used
- Operating system or DBMS to be used
- Capabilities of I/O devices
- Standards compliance

Decision Trees & Decision Tables

★ Decision tree & Decision table - Techniques for Representing Complex Logic

- When the **Requirements** of the system are **complex** containing
 - Many different **scenarios**
 - **Condition – Decision** rules
- **Textual description** using natural languages may not be appropriate
- In such situations, a **decision tree** or a **decision table** can be used to represent the logic & the processing involved

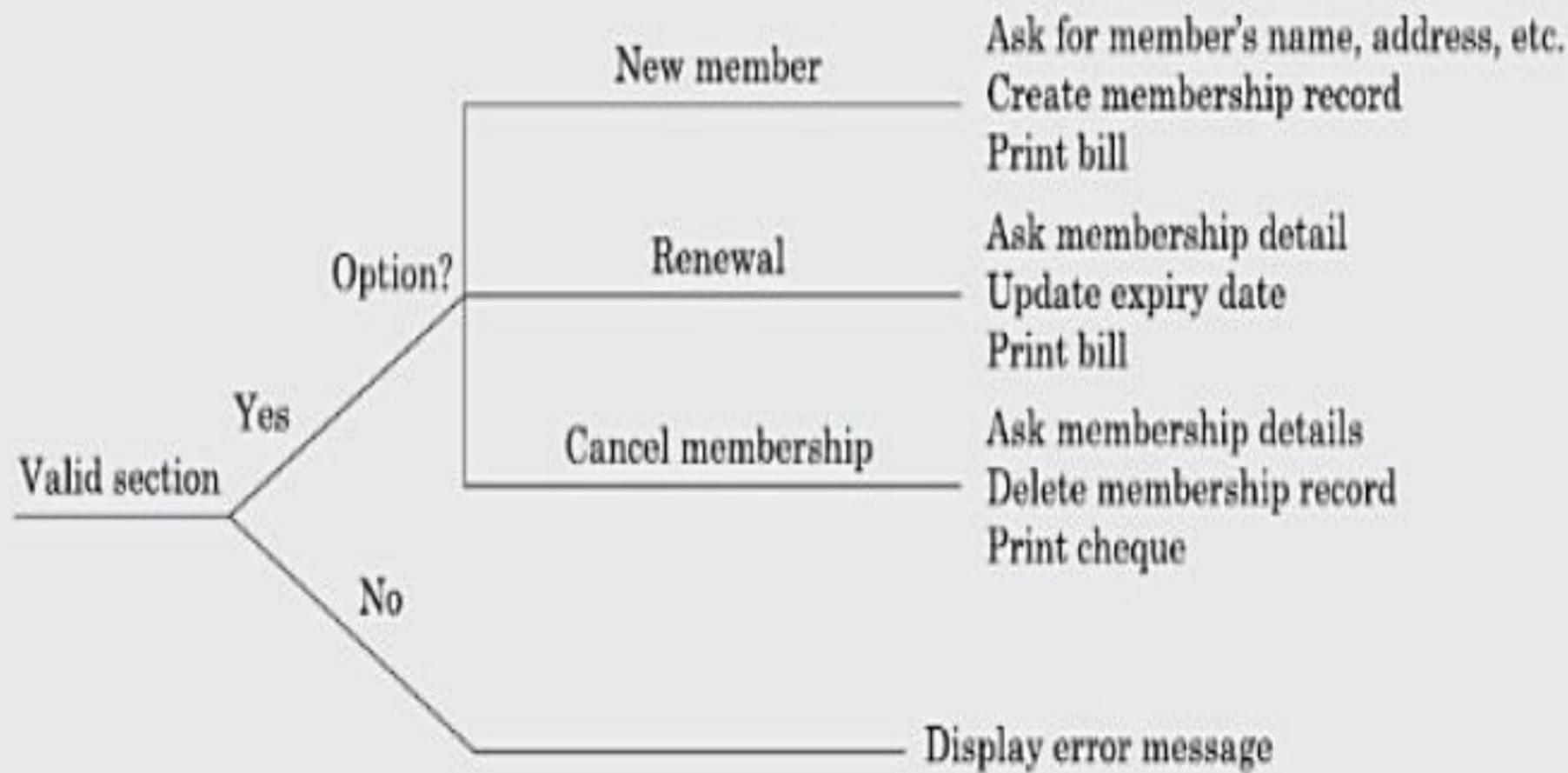


Decision Tree

- **Decision Tree** gives a **graphic view** of :
- The **processing logic** involved in **decision making** &
- The corresponding **actions** to be taken
- The **edges** of a **Decision Tree** represent **conditions**
- The **leaf nodes** represent the **actions** to be performed
- ***Example of DT for library membership management software (LMS) is given in the next page***



Decision Tree for LMS system



Decision Table

- Shows the **decision making logic** & the **corresponding actions** in a **tabular form**
- The **upper rows** of the table specify the **variables** or **conditions** to be evaluated
- The **lower rows** specify the **actions** to be taken
- A **column** in the table is called a ***rule***
- Example of DT for library membership management software (LMS) is given in the next page***

Decision Table for LMS system

Table 4.1: Decision Table for the LMS Problem

Conditions

Valid selection	NO	YES	YES	YES
New member	-	YES	NO	NO
Renewal	-	NO	YES	NO
Cancellation	-	NO	NO	YES

Actions

Display error message	x
Ask member's name, etc.	x
Build customer record	x
Generate bill	x x
Ask membership details	x x
Update expiry date	x
Print cheque	x
Delete record	x

Ex2: Decision Table for Salary Increment

Decision Table Question					
	<u>Grade</u>	<u>exp</u>	<u>Increment</u>	<u>Bonus</u>	
• If	A	—	> 10 yrs	30 %	5 %
• If	A	—	< 10 yrs	25 %	—
• If	B	—	> 10 yrs	20 %	—
• If	B	—	< 10 yrs	15 %	—
• If	C	—	Any	10 %	—
• If	D	—	Any	0 %	5 %

Conditions	Rules →						
	NO	YES	YES	YES	YES	YES	YES
valid selection							
Grade	-	A	A	B	B	C	D
Experience in yrs	-	>10	<10	>10	<10	-	-
Actions							
Increment of 30%		✓					
Increment of 25%			✓				
Increment of 20%				✓			
Increment of 15%					✓		
Increment of 10%						✓	
Increment of 5%							✓
Bonus of 5%	✓	✓					
Error message	✓						



Possible Question

Q. Construct the Decision Table for the following business rules (L)

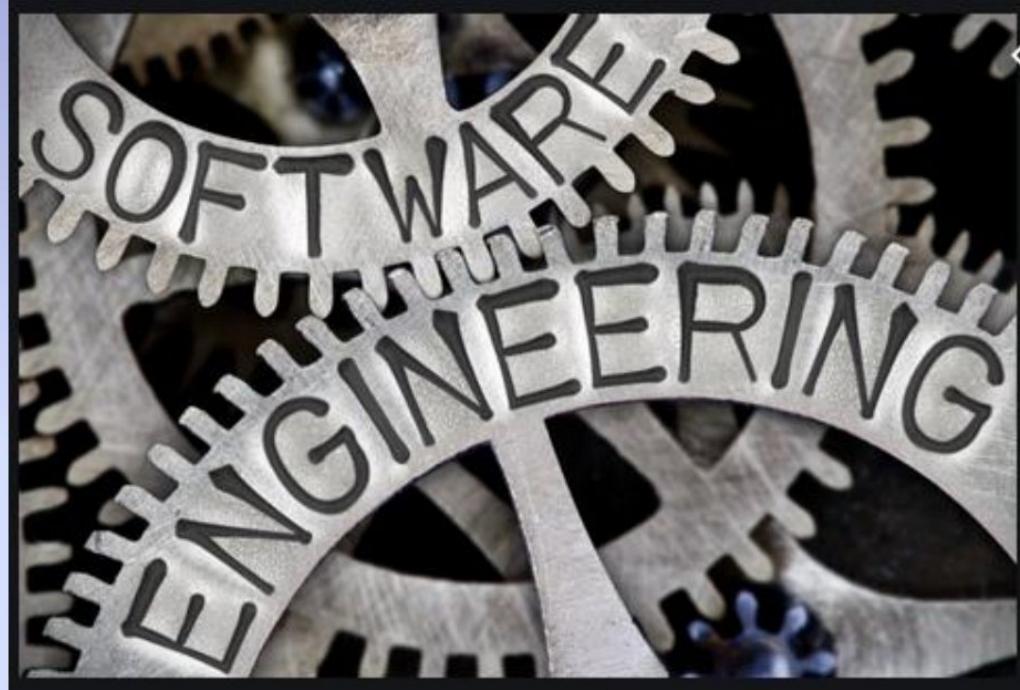
- The number of vacation days depends on age & years of service
- Every employee receives at least 22 days.
- Additional days are provided according to the following criteria:
- Only (employees < 18 yrs or at least 60 yrs, or employees with >= 30 years of service) will receive **5 extra days**
- (Employees with >= 30 years of service & employees of age >= 60 yrs) will receive **3 extra days**, on top of additional days already given.
- If (employee has >= 15 & < 30 years of service, **2 extra days** are given. These 2 days are also provided for employees of age 45 or more. These 2 extra days can not be combined with the extra 5 days.



Thanks!!!

Software Engineering & UML

Module-2 A



Software Design



Module-2 Contents

- **Software Design:**

- Overview of the Design Process
- Cohesion and Coupling
- Layered Arrangement of Modules
- Approaches to Software Design

- **FOD:**

- SA/SD Methodology
- DFD
- Structured Design
- Detailed Design

Introduction

- ✓ Design phase transforms **SRS doc** to **Design docs:**
 - ✓ **Design documents** are easily implementable using any *programming language*



Items Designed in Design Phase

Items designed during **design phases** are :

1. Design of all the **Modules**
- HLD** 2. **Relationship** (or dependencies) among modules
3. **Interface** between modules
i.e. Data exchanged between modules
- DD** 4. **Data structures** of individual modules
5. **Algorithms** for individual modules

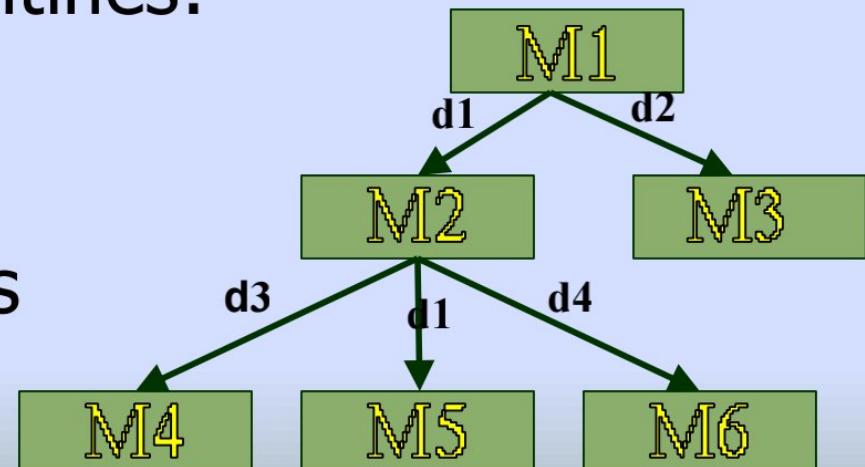
Design activities

Design activities are usually classified into two stages:

1. **High-level design (HLD)**
2. **Detailed design (DD)**

1. High-level design Identifies:

- ✓ Modules
- ✓ Relationships among modules
- ✓ interfaces among modules



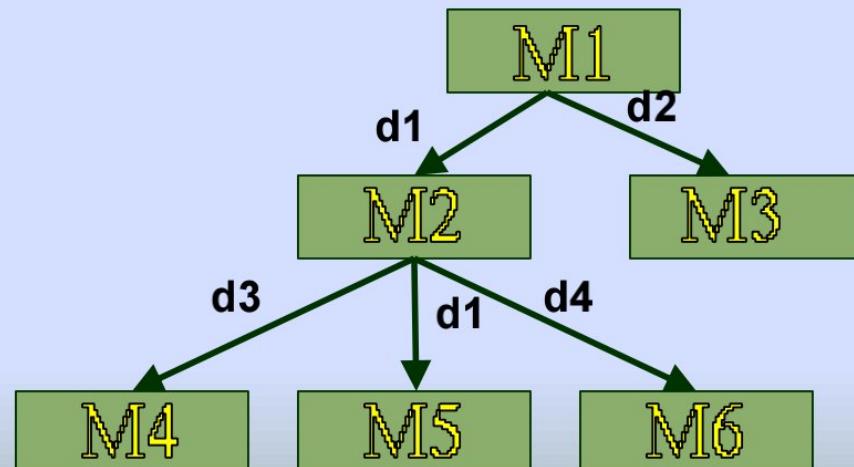
1. High-level design

- ✓ The **output** of high-level design:

✓ **Software architecture document**

- ✓ Also called

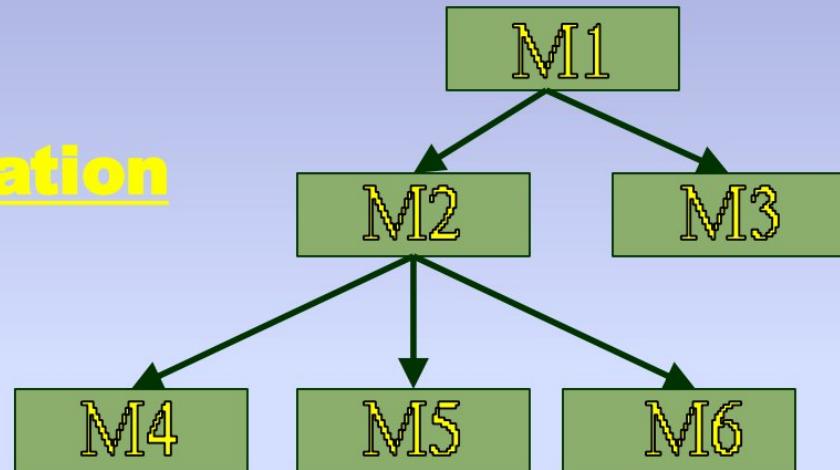
✓ **structure chart**



2. Detailed design

✓ **Output of detailed design :**

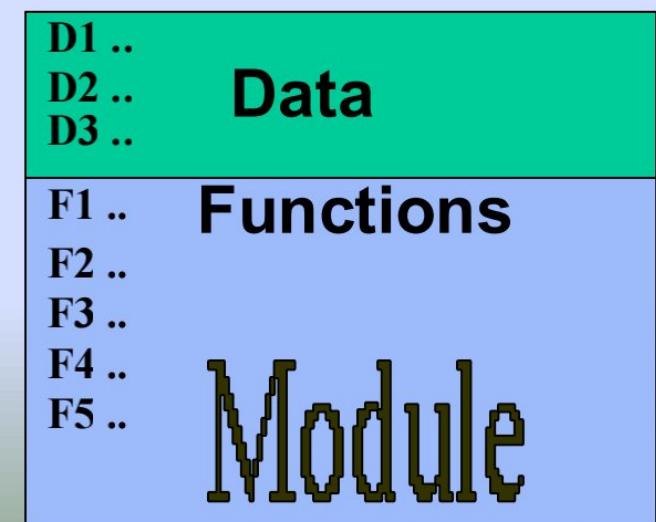
Module specification



✓ **For each module, design :**

Data structure &

Algorithms (or functions)



Characteristics of a good design

- 1. Correctness**
- 2. Understandability**
- 3. Maintainability**
- 4. Efficiency**
- 5. Modularity**
- 6. Clean decomposition**
- 7. Neat Arrangement**
- 8. High cohesion & Low coupling**
- 9. High Fan-In & Low Fan-Out**

Characteristics of a good design

1. Correctness & 4. Efficiency

Design should accurately translate the **requirements** :

For correct & efficient **Implementation** of the system

2. Understandability & 3.Maintainability

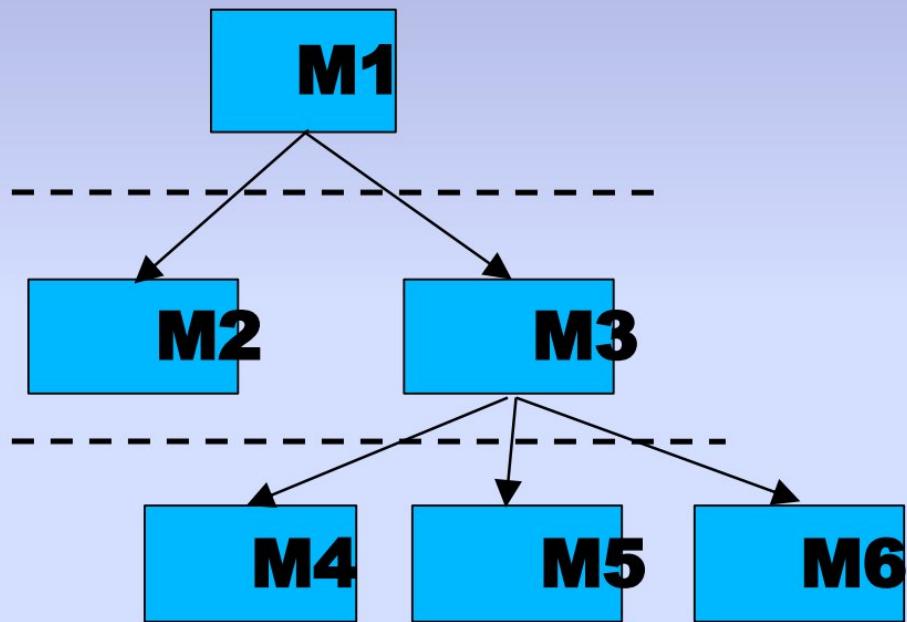
- ✓ The design should be easy to understand
- ✓ Use consistent & meaningful names for design components
- ✓ This will make the system easy to maintain & change

Characteristics of a good design

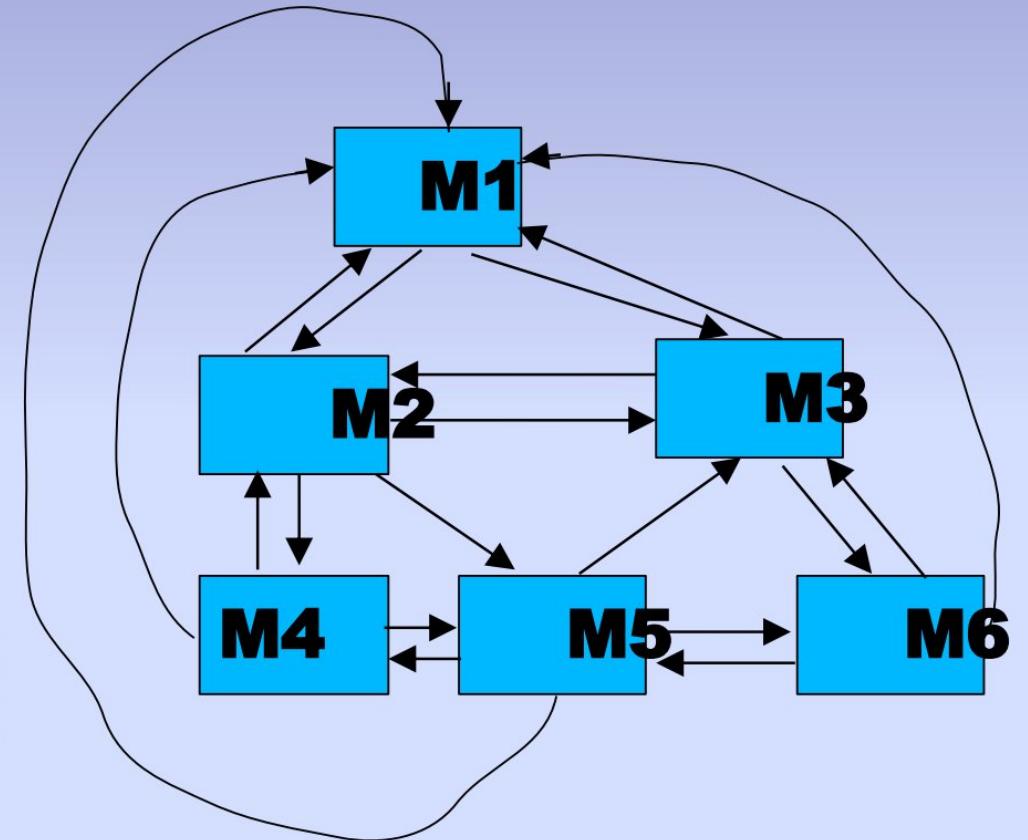
5. Modularity & 6. Clean Decomposition

- ✓ **Decomposition** of a problem **cleanly** into **modules** using **divide & conquer principle**
- ✓ **Modules** must be **almost independent** of each other
- ✓ If **modules** are **independent**: they can be solved separately
 - ✓ This **reduces the complexity**

Design



**Modular
Design**



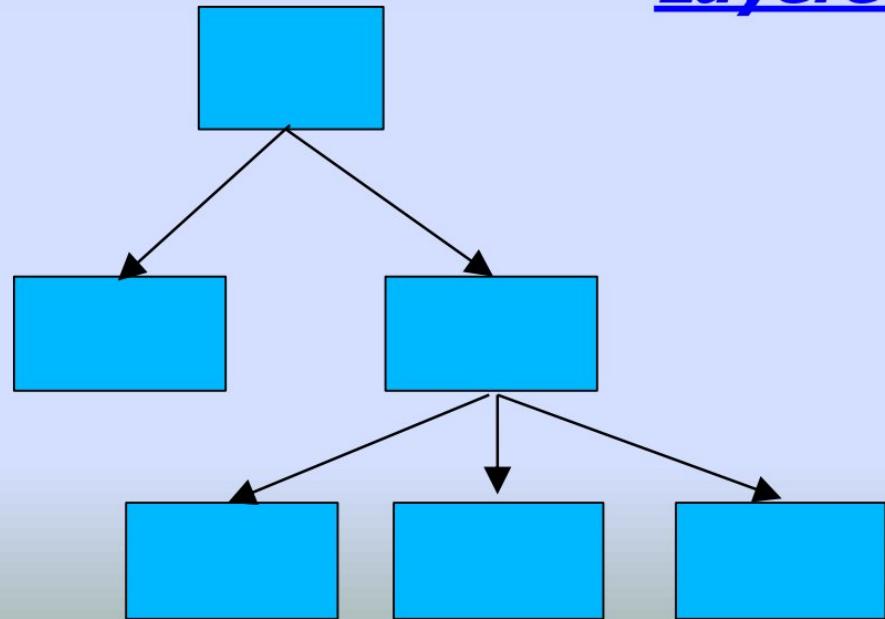
**Design
exhibiting poor
modularity**

Characteristics of a good design

7. Neat arrangement

Neat arrangement of modules in a hierarchy means:

Layered solution & abstraction



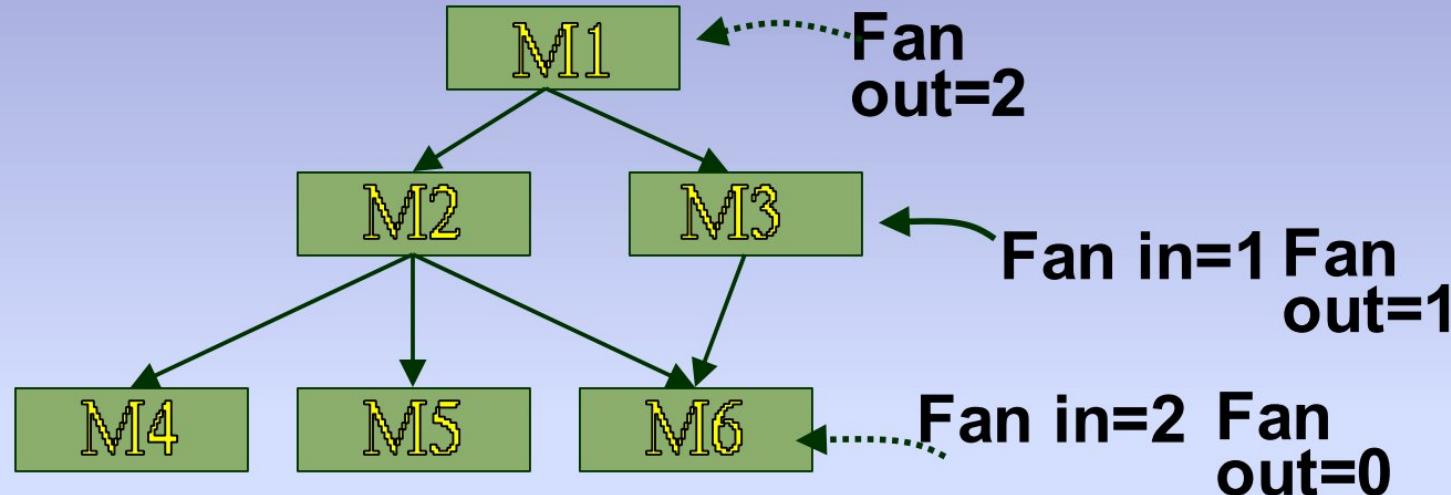
Characteristics of a good design

9. High Fan-In & Low Fan-Out

Characteristics of Structure Chart

- ✓ **Depth**: no. of levels of control
- ✓ **Width**: overall span of control
- ✓ **Fan-out**: no. of modules directly called by a module
=> Fan-out => **dependency (Low is good / high is bad)**
- ✓ **Fan-in**: how many modules call/invoke a given module
=> Fan-in => **re-usability (High is good / low is bad)**

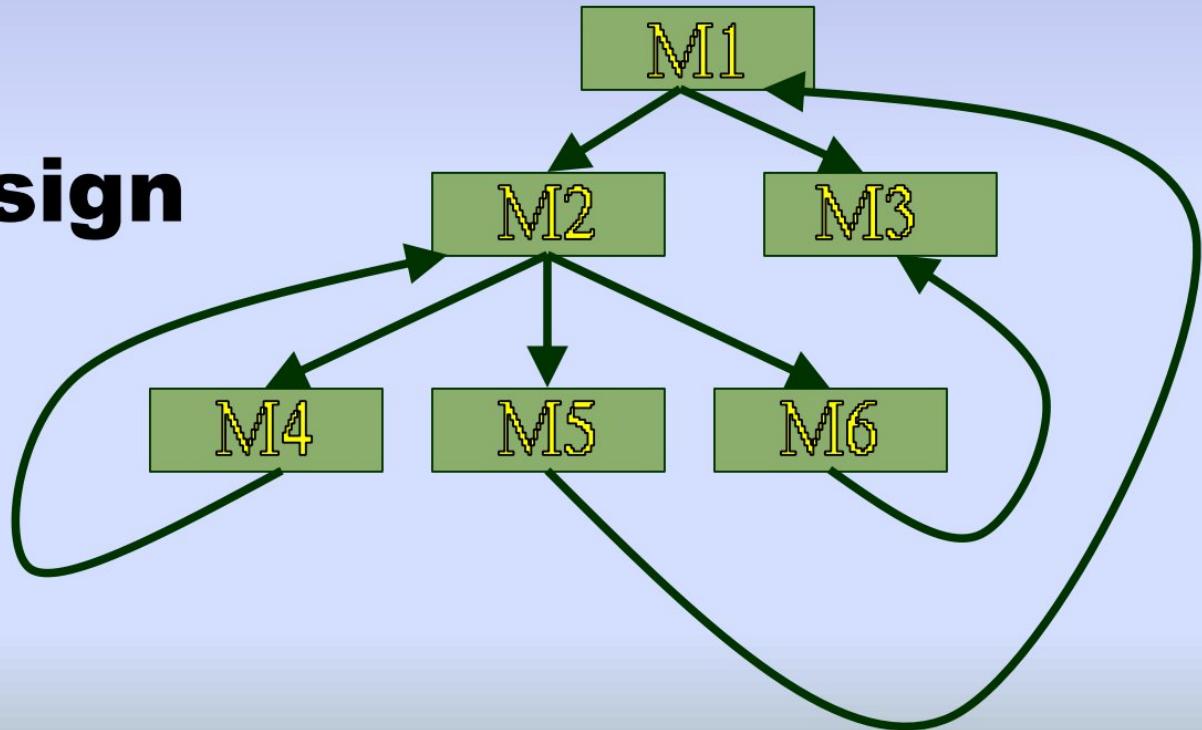
Module Structure



- ✓ **Fan-out:** no. of modules directly called by a module
=> **dependency (Low is good / high is bad)**
- ✓ **Fan-in:** how many modules call a given module
=> **re-usability (High is good / low is bad)**

✓ Having high Fan-In & low Fan-out is a good design

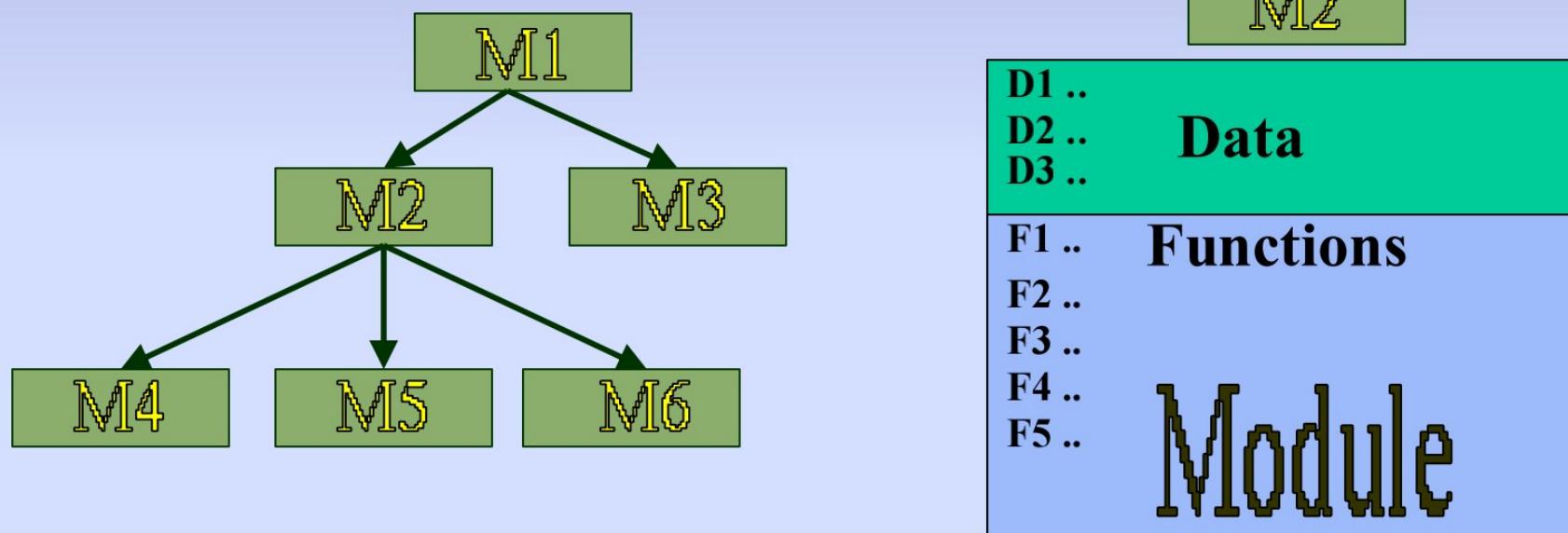
Bad Design



8. Cohesion & Coupling

- ✓ **Cohesion** is a measure of:
 - ✓ functional strength of a module
 - ✓ How cohesive/closely knit the functions of the module are
- ✓ **Coupling** between two modules:
 - ✓ a measure of the degree of interdependence or interaction between modules
- ✓ A module having high cohesion & low coupling is
 - ✓ functionally independent of other modules

Structure Chart

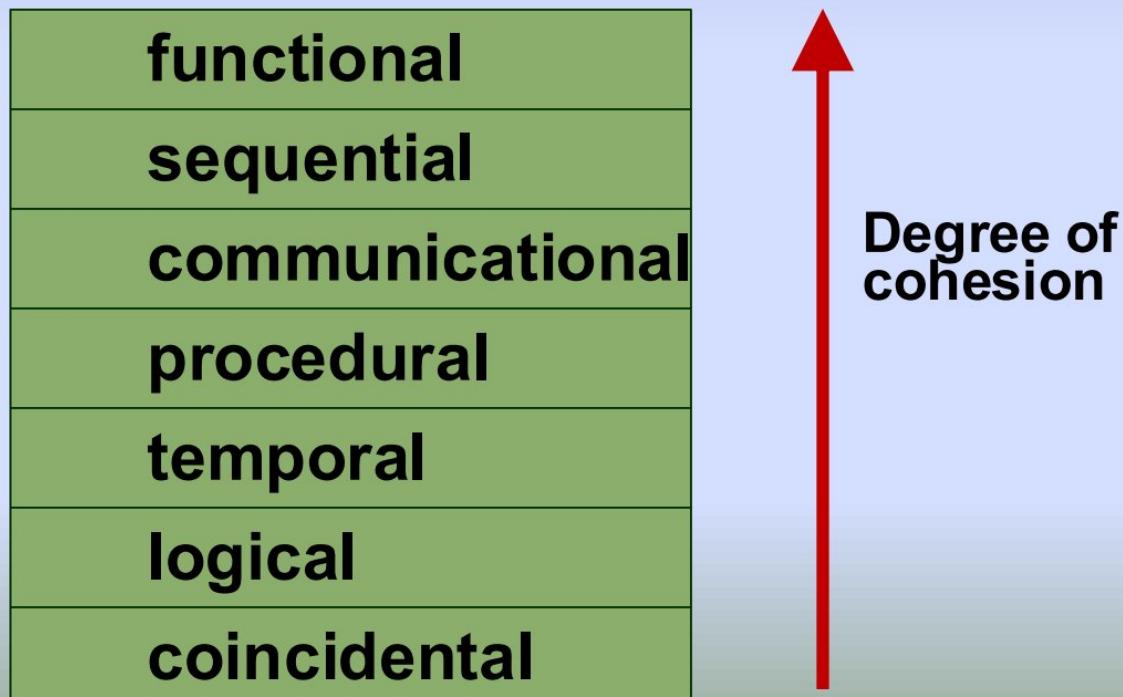


Adv.s of Functional Independence

- ✓ **Complexity** of design is reduced
 - ✓ Modules are easily understood
- ✓ Reduces error propagation
 - ✓ An error in one module does not affect other modules
- ✓ Reuse of modules is better
 - ✓ if it is functionally independent

Classification of Cohesiveness

- ✓ **Classification** gives us idea about cohesiveness of a module
i.e. whether it displays high or low cohesion
- ✓ **7 types of cohesion are there**



1. Coincidental cohesion

- ✓ The module performs a set of tasks
- ✓ But these are not related
- ✓ Tasks are **random collection of functions**
- ✓ The functions are put in the module
 - ✓ without any thought or design

2. Logical cohesion



- ✓ All the functions of the module

- ✓ Perform **similar operations**

Ex. Error handling
data input
data output etc.

3. Temporal cohesion

- ✓ The tasks are related by **time**
- ✓ All the tasks are to be executed in the same time span
- ✓ Ex
 - ✓ Initialization of a process
 - ✓ Start-up process
 - ✓ Shut-down process etc.

4. Procedural cohesion

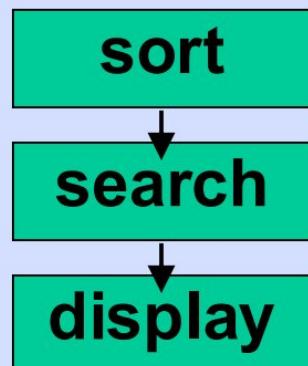
- ✓ All the functions of the module:
 - ✓ Are part of a procedure (algorithm)
 - ✓ Contains sequence of steps to be carried out
 - ✓ in certain order for achieving an objective
 - ✓ **Ex:** Payroll run to generate salary of employees

5. Communicational cohesion

- ✓ All functions of the module:
 - ✓ Reference or update the **same data structure**
 - ✓ **Ex:** The set of functions defined for a stack or queue

6. Sequential cohesion

- ✓ All tasks of the module form different parts of a **sequence**
- ✓ **Output** from one task of the sequence is **input** to the next



7. Functional cohesion

- ✓ All the functions of the module cooperate to achieve a single goal
- ✓ **Ex:** Renew book in LIS, Ticket booking in RRS

Coupling

✓ **Coupling** indicates:

- ✓ **Interdependent** between the modules of a system
- ✓ **Classifications of coupling** will help us estimate the degree of coupling between two modules
- ✓ **Five types** of coupling are there

data
stamp
control
common
content



Degree of
coupling

1. Data Coupling & 2. Stamp coupling

1. Two modules are **data coupled**,

- ✓ If they communicate via an elementary data item
- ✓ **Ex:** an integer, a float, a character etc.

2. Two modules are **stamp coupled**,

- ✓ If they communicate via a composite data item
- ✓ such as a record in PASCAL or a structure in C

4. Common & 5. Content Coupling

4. Two modules are **common coupled**,

✓ If they share some **global data**

✓ Ex: *Room-Number[100]* in a Hotel Management system

5. Two modules are **content coupled**,

✓ if they **share code**

✓ Ex: Branching from one module into another module

The degree of coupling increases from data coupling to content coupling

3. Control Coupling

3. Two modules are **control coupled**,

✓ If **data** from one module is used to direct **order of instruction execution** in another

✓ **Ex:**

A **flag** set in **Module-1** and **tested** in **Module-2**

If (Book Availability_Status = True)
 Then Renew_Book
 Else Display Regret message

Design Approaches

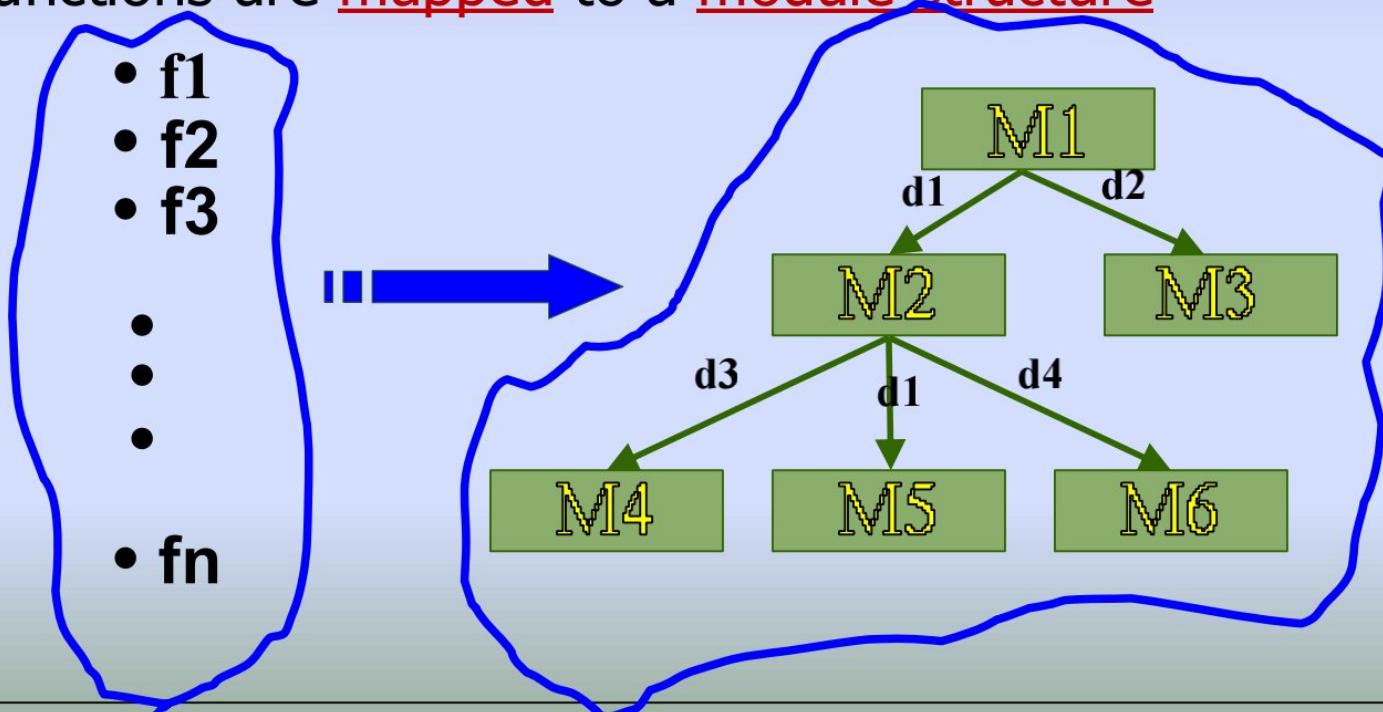
✓ **Two** fundamentally different software design approaches:

1. Function-oriented design

2. Object-oriented design

1. Function-Oriented Design

- ✓ In FOD, A system is seen as a set or collection of functions
- ✓ Starting at this high-level view of the system:
 - ✓ Each function is successively refined into more detailed functions
 - ✓ Functions are mapped to a module structure



✓ **EX:**

- ✓ The function `create-new-library-member`:
- ✓ Creates the record for a new member
- ✓ Assigns a unique membership number
- ✓ Prints a bill towards the membership

2. Object-Oriented Design

- ✓ System is viewed as a **collection of objects** (i.e. entities)
- ✓ System state is **decentralized** among the **objects**
 - ✓ Each **object** manages its own information
- ✓ **Library Information System:**
 - ✓ Each **library member** is a separate **object**
 - ✓ With its own data and **functions**
 - ✓ **Functions** defined for one object:
 - ✓ Cannot directly refer to or change data of other objects

Object-Oriented Design

- ✓ **Objects** have their own internal data
- ✓ **Similar objects** constitute a **class**
 - ✓ Each **object** is a member of some class
- ✓ Classes may inherit features from a super class
- ✓ Objects **communicate by message passing**
- ✓ Unlike **FOD**,
 - ✓ in **OOD**, the basic component is not functions such as "sort", "display", "track", etc.
 - ✓ but **real-world entities** such as "employee", "machine", "Account", etc.

OOD

VS

FOD

1. As per Grady Booch : Identify **verbs** if you are doing FOD & Identify **nouns** if you are doing OOD"
2. In FOD: We design functions, but in OOD, we design Objects
3. In FOD: the system state is **centralized**, In OOD, it is **decentralized**
4. Objects communicate by message passing, Functions communicate by data passing
5. In **FOD**, we follow Top-down approach & in **OOD**, we follow Bottom-Up approach

Function-Oriented Software Design

39



Scanned with OKEN Scanner

Topics

- **Introduction to function-oriented design**
- **High level Design**
 - **SA/SD Methodology**
 - **Data flow diagrams (DFDs)**
 - Introduction
 - Symbols
 - Level-0 DFD (Context Diagram)
 - Level-1 DFD (Example)
 - Level-2 DFD (Example)
 - Rules of DFD
 - Examples
- **Detailed Design**

40



Scanned with OKEN Scanner

FOD - Introduction

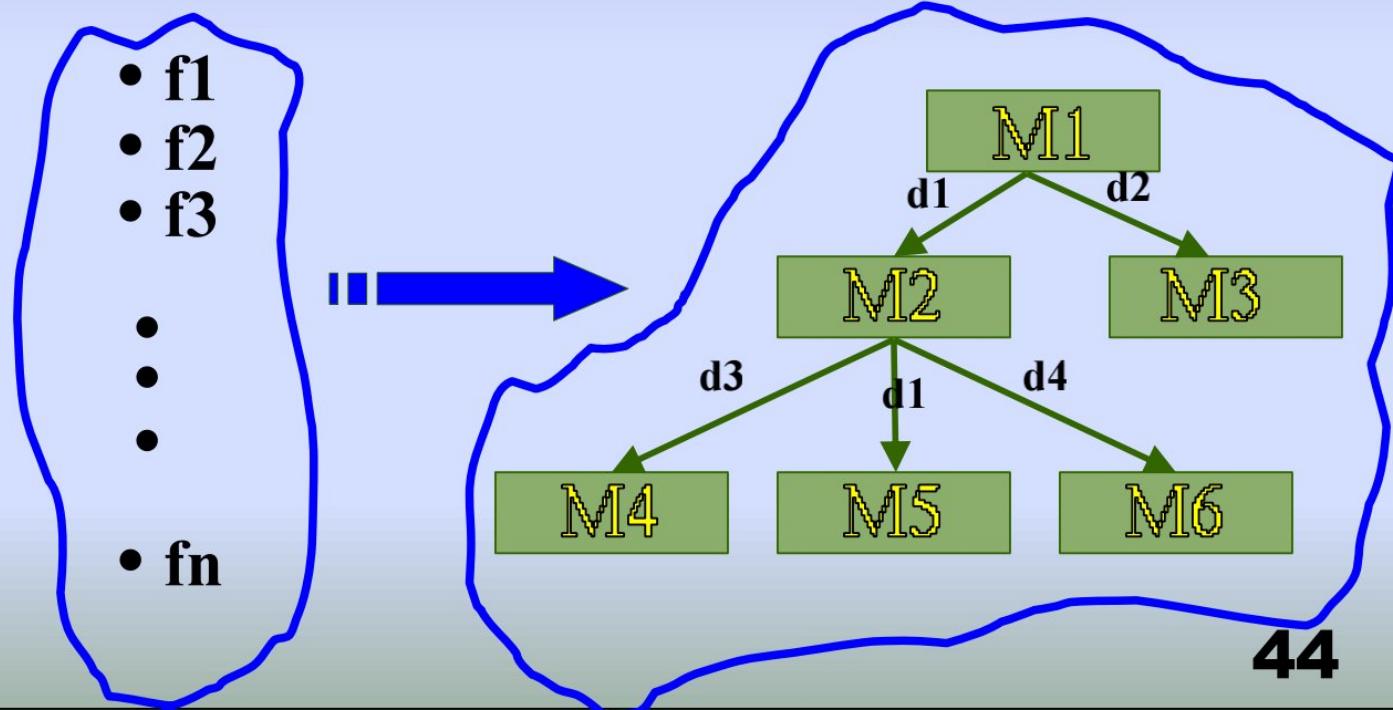
- **Function-oriented design techniques** are very popular
 - Heavily used in many s/w dev. organizations
- **During the FOD process:**
 - **Structured Analysis** is performed – produces **DFDs**
 - **Structured Design** is performed
 - **High Level Design** – produces “**Structure Chart**”
 - **Detail Design** – produces “**Module Specification**”

Structured Analysis

- Each functionality specified in the **SRS** is Analyzed
- Then Decomposed into more detailed functions
- High-level data is decomposed into more detailed data
- **SA transforms** a textual problem description in the SRS to a graphic model called DFD
- **DFDs** graphically represent the results of structured analysis

Structured Design - HLD

- All the functions represented in the **DFD** are mapped to the **module structure**
- The module structure also called as the : **Software architecture (Structure chart)** – the result of **HLD**



Structured Design -Detail Design

➤ **S/W architecture produced in HLD**

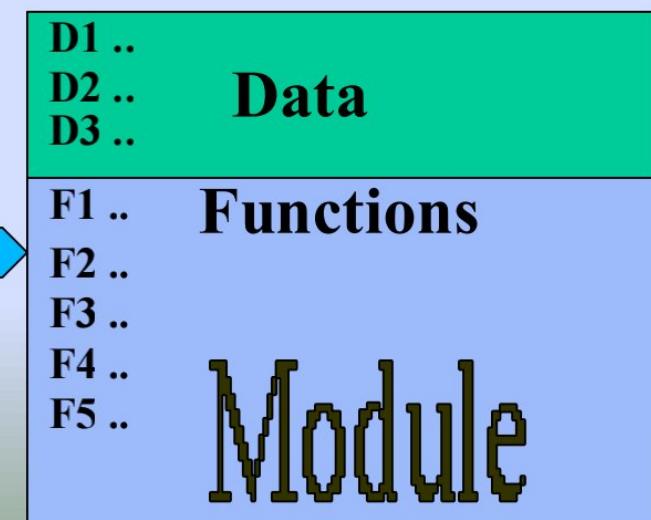
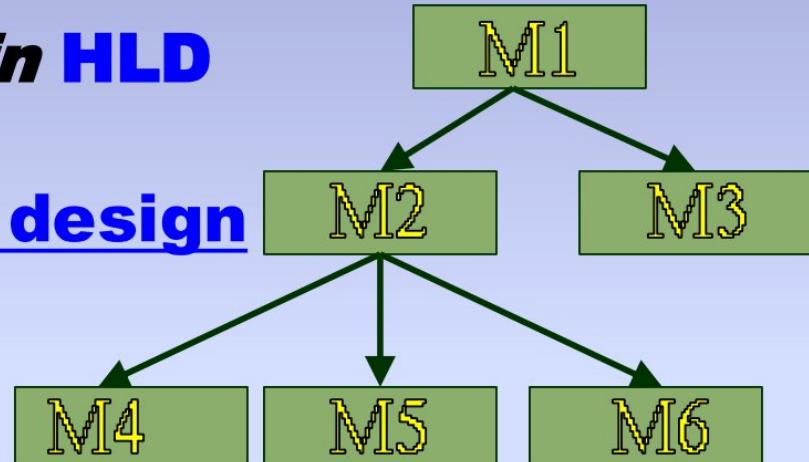
➤ is **Refined** during **detailed design**

✓ **Output of detailed design :**

Module specification

✓ **For each module, design :**

Data structure & Algorithms



Structured Design -Detail Design

- **Detailed design** can be directly implemented:
 - Using any programming language

Data Flow Diagrams

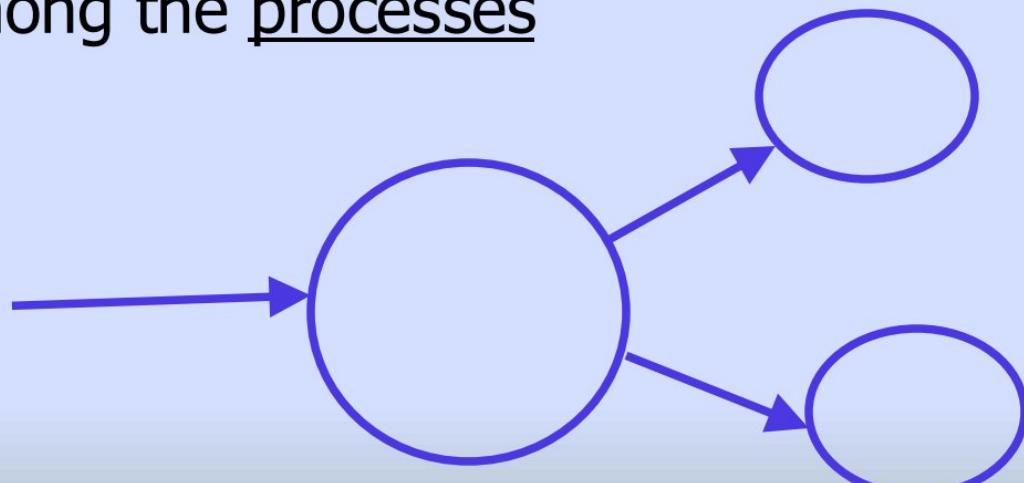
(DFDs)

Data flow diagrams

- **DFD** is a popular modelling technique:
- Used to **represent the results of structured analysis**
- DFD technique is very popular because:
 - It is simple to understand and use

Data flow diagram

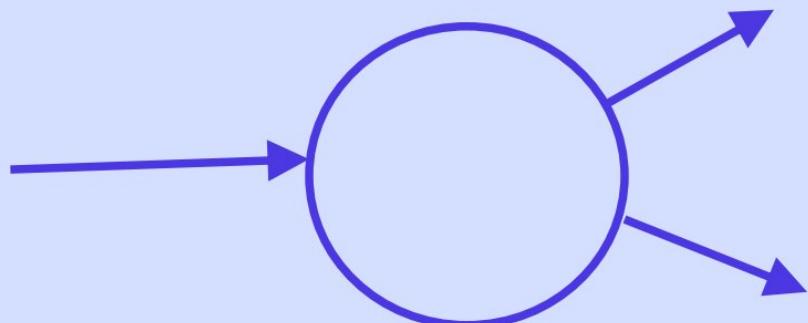
- **DFD is a graphical model:**
 - Shows different **processes** of the system &
 - **Data interchange** among the **processes**



DFD Concepts

➤ Function or Process:

- Each function *accepts some input data*
 &
➤ Produces some *output data*
- A DFD model uses:
 - Limited types of **symbols**
 &
 - Simple **set of rules**

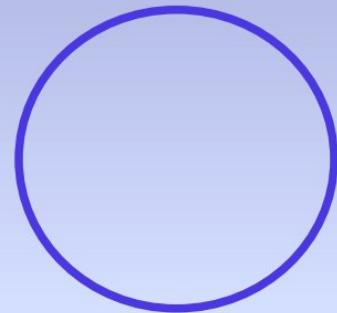


Data Flow Diagrams (DFDs)

➤ Primitive Symbols Used for Constructing DFDs:



External Entity



Process



Data Flow



Output



Data Store

(1) External Entity

- Is a **real physical entity**

- These entities input data to the system

- Receive output data produced by the system

- These are also called terminator, source, or sink

- Represented by a **rectangle**

Librarian

Student

Customer

(2) Process Symbol

- A process or function such as "search-book" / "Renew-book" is represented using a **circle**

➤ This symbol is called a **process or bubble or transform**



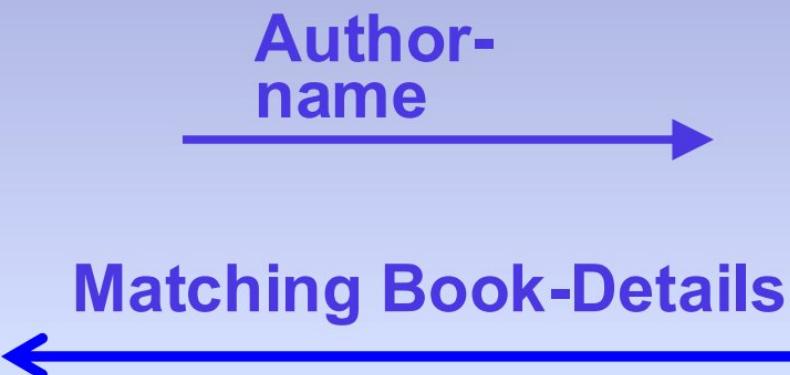
➤ Generally processes transform data values

➤ Process represent **some activity**
➤ Process names should be **verbs**



(3) Data Flow Symbol

- A Directed arc



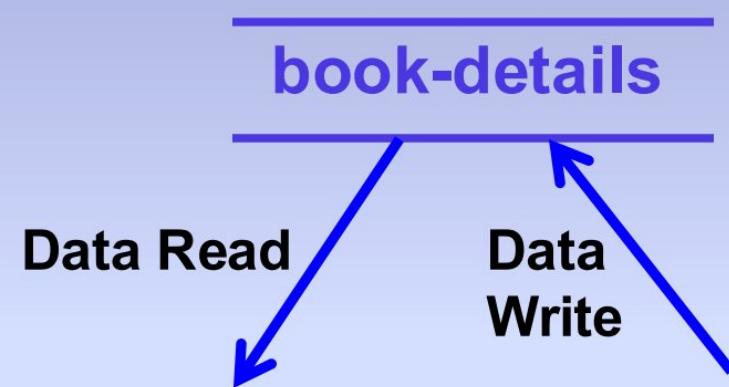
- Represents data flow in the direction of the arrow
- Data flow symbols are labelled with names of data they carry

(4) Data Store Symbol

✓ Represents a logical file:

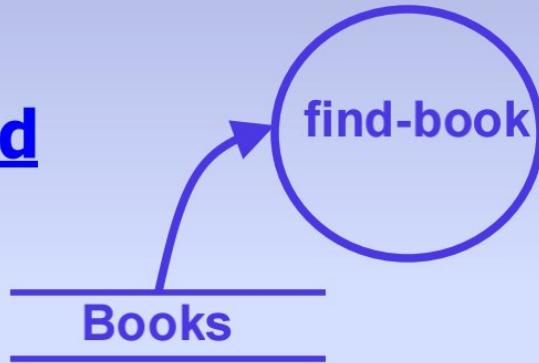
✓ A logical file can be:
✓ a data structure
✓ a physical file
✓ Table(s)

✓ Each data store is connected to a process by means of a data flow symbol



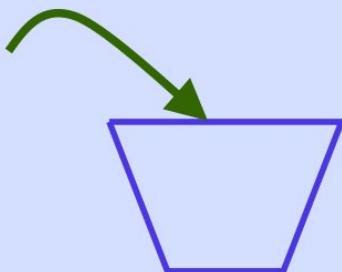
Data Store Symbol

- Direction of data flow arrow:
 - shows whether data is being read from or written to
- An arrow into or out of a **DS**
 - implicitly represents the **entire data** of the data store
 - arrows connecting to a data store **need not be labelled**



(5) Output Symbol

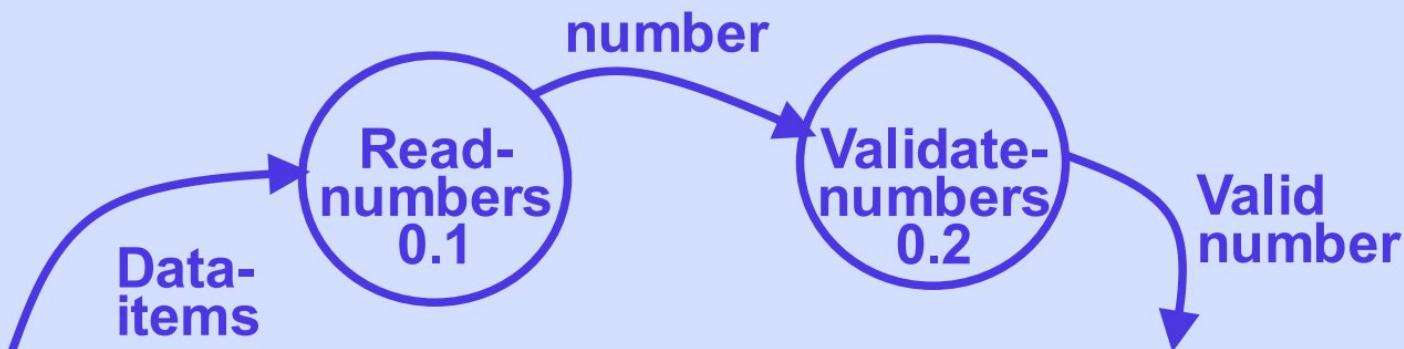
- **Output** produced by the system



- **Ex:** Reports, Physical docs (Ticket, Admit card..), Files..

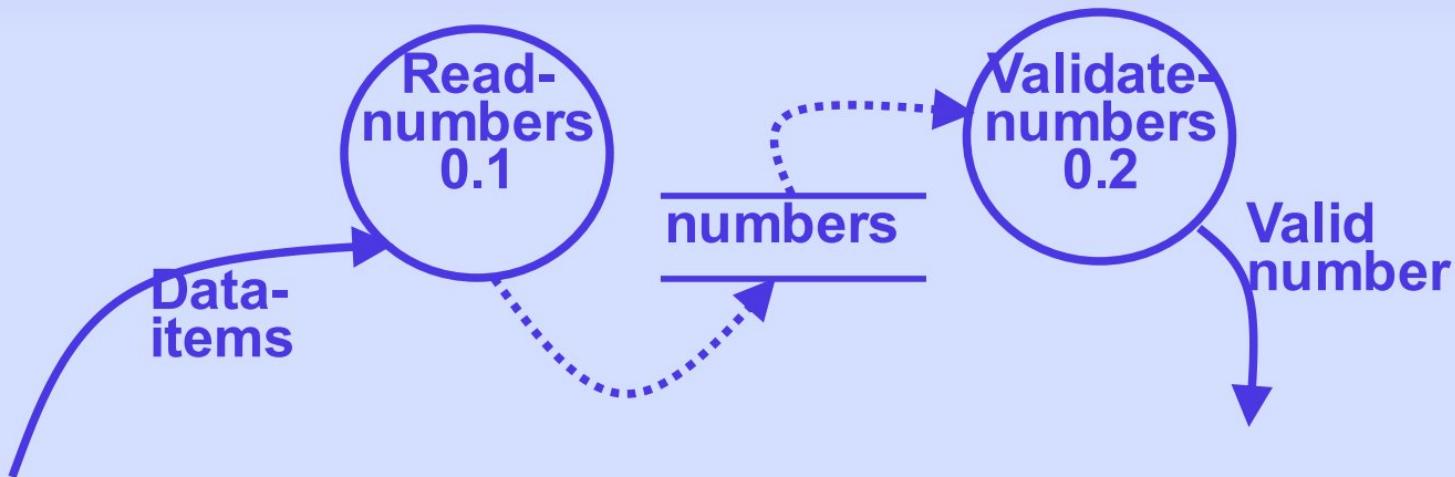
Synchronous operation

- If two bubbles are directly connected by a data flow arrow:
 - They are synchronous



Asynchronous operation

- If two bubbles are **connected via a data store:**
 - they are **not synchronous**



How is Structured Analysis Performed / Developing DFD model of a system

- Initially represent the System at the most abstract level called the Context diagram
- The entire system is represented as a single Bubble(Or Process)
- This bubble is labelled according to the main function of the system (*or Title of the system : Indian Railway Reservation System*)

Context Diagram

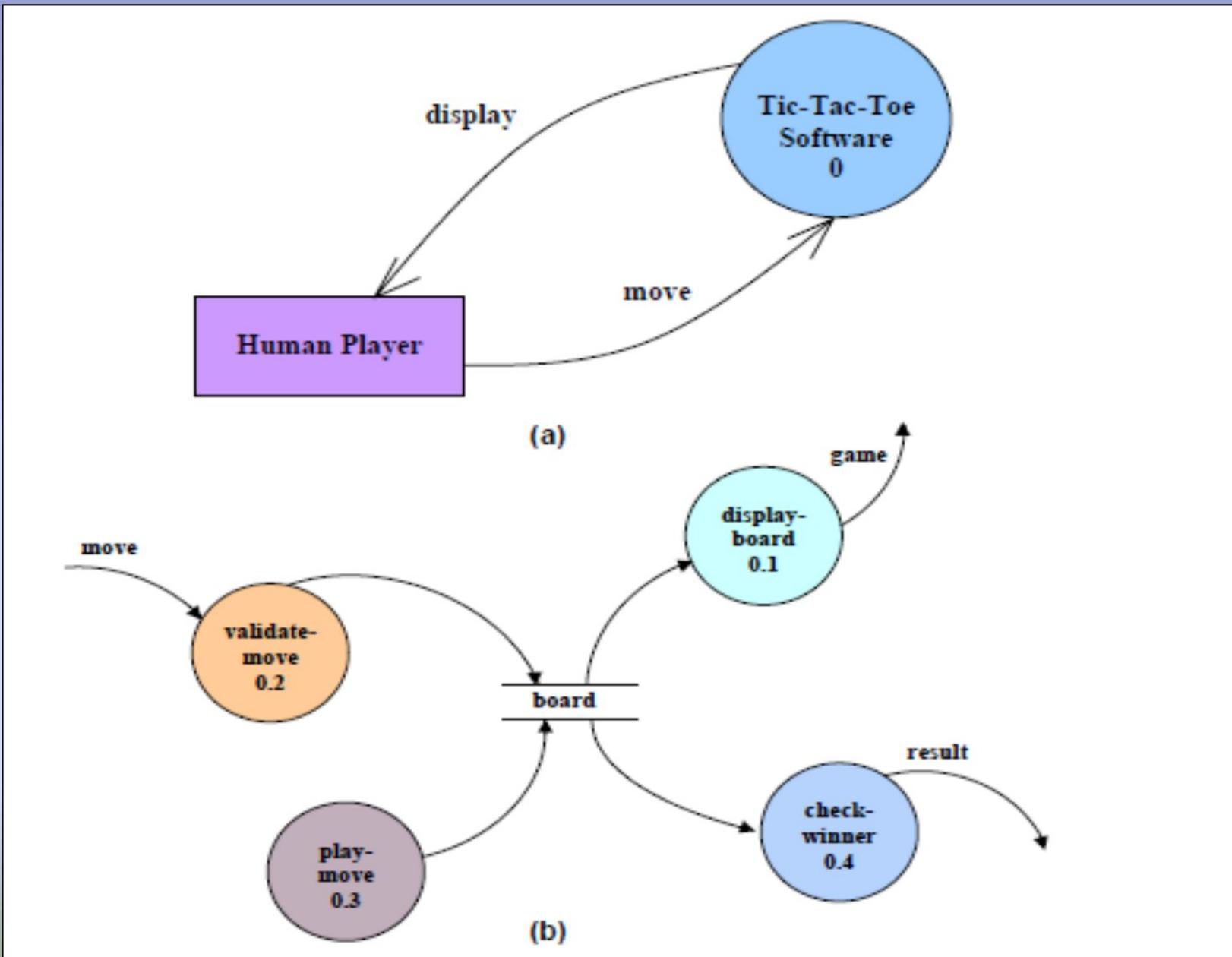
- A **context diagram** shows:
 1. **Data input** to the system
 2. **Output data** generated by the system
 3. **External entities**
- **Context diagram** captures:
 - Various **entities external** to the system and interacting with it (*Student, Admin, Bank Accountant etc.*)
 - **Data flow** occurring between the system and the external entities (*User credentials, Authentication status, Start & End Stations, Train details*)
- The **context diagram** is also called as the **level 0 DFD**

Example 1: Tic-Tac-Toe Computer Game

Example 1: Tic-Tac-Toe Computer Game

Tic-tac-toe is a computer game in which a human player and the computer make alternative moves on a 3×3 square. A move consists of marking previously unmarked square. The player who first places three consecutive marks along a straight line on the square (i.e. along a row, column, or diagonal) wins the game. As soon as either the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, but all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

DFD- Level 0 (Context diagram) & level 1 DFD



Data Dictionary

- A DFD is always accompanied by a **data dictionary**
- A data dictionary defines **all simple & composite data items** (with component items) appearing in a DFD, along with
 - Their Name & definition
 - Their Purpose
- **DD** provides all project members with a standard terminology for all data to avoid **confusion**
- **EX:** See next page

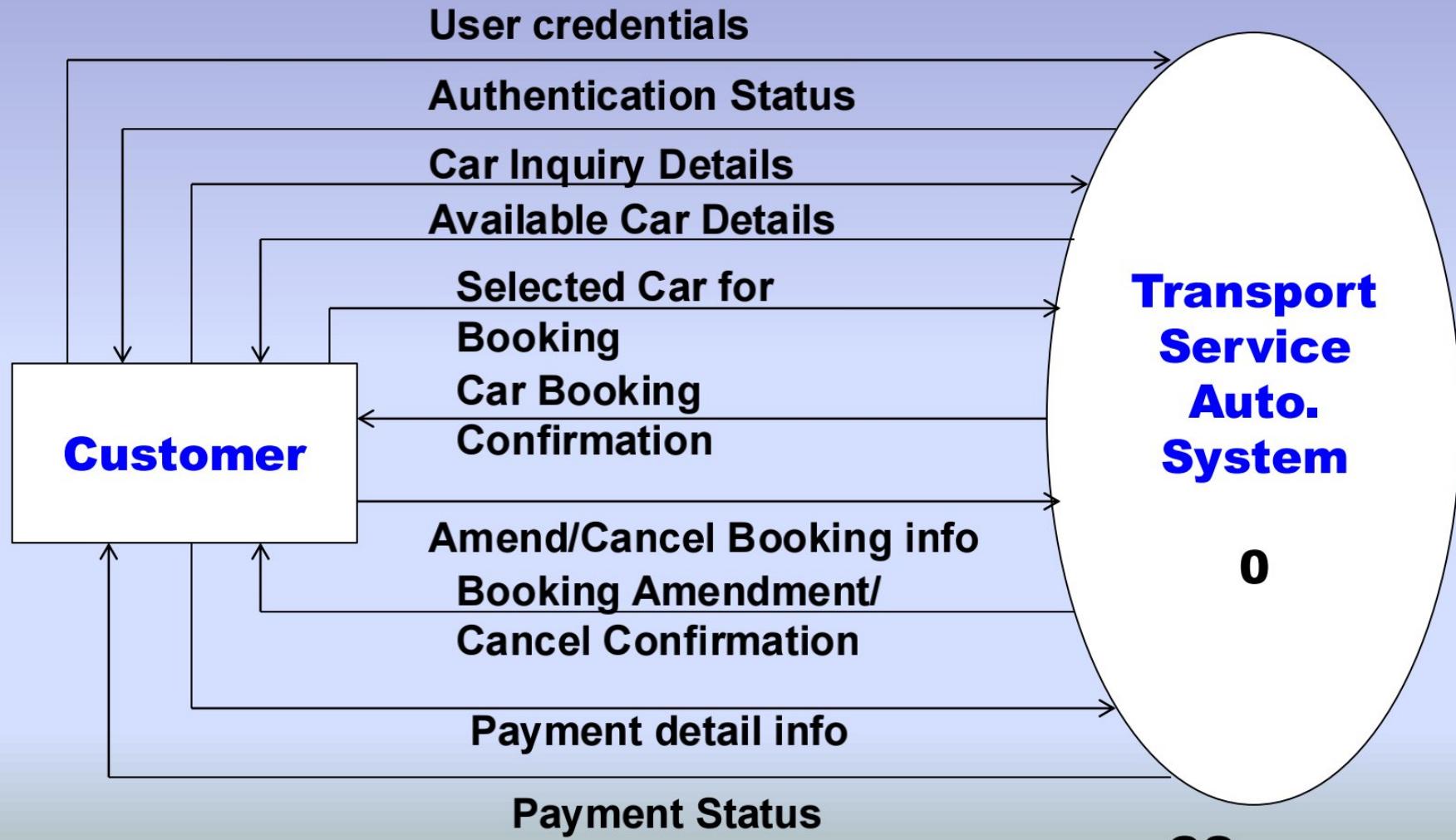
Data Definition in DD

- **Composite data** are defined in terms of primitive data items using following **operators**:
 - **+**: Denotes composition of data items
 - **Ex:** 'a+b' represents data a and b
 - **=**: Represents equivalence,
 - **Ex:** 'a=b+c' means that a represents b and c
 - **[,,,]:** Represents selection
 - Any one of the data items listed inside the square bracket can occur
 - **Ex:** [a,b,c] represents either a or b or c occurs

Data Definition

- **()**: Contents inside the bracket represent optional data (data may or may not appear)
 - **Ex:** 'a+(b)' represents either '**a' or 'a+b' occurs**
- **{ }**: Represents, iterative data definition
 - **Ex:** '{name}5' represents '**five instances of name data**'
 - **{name}*:** represents , zero or more instances of name
- *** *:**
 - Anything appearing within * * is considered as comment

Ex1: Transport Service Automation System



Level 1 DFD

- Examine the **SRS document**:
 - Represent **each high-level function** as **a bubble**
 - **Ex:** *Train Information Enquiry, Ticket Booking, Ticket Cancellation*
 - Identify **data inputs & outputs** to every **high-level function**
 - **Ex:** *Travel dates, start & end stations, Passenger info(name, age ..)*
 - Represent **data store** for I/P & O/P of data to the system
 - **Ex:** *Train Info database, Train Availability File ..*

Higher level DFDs



- Each **high-level function** is then **decomposed** into **sub-functions**:
 - **Identify** the **sub-functions**
 - **Identify** the **data input** & **output** to each sub-function
- These are represented as **DFDs**

Decomposition



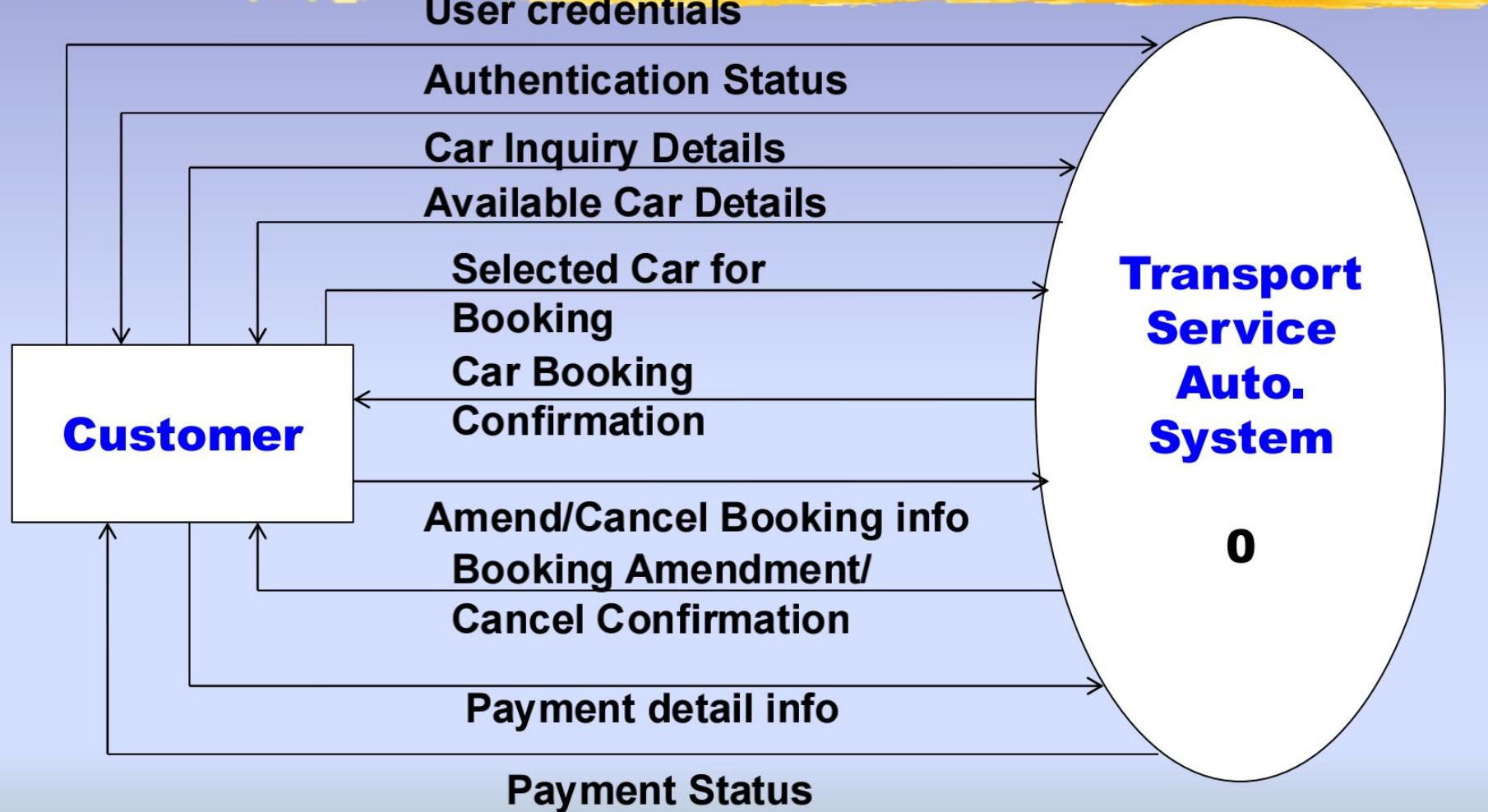
- **Decomposition of a bubble:**
 - Also called **factoring** or **exploding**
- Each bubble is decomposed to **between 2 to 7 bubbles**
- **Too few bubbles:** make decomposition **meaningless**
- **Too many bubbles:** **more than 7 bubbles** make the DFD **hard to understand**

Decompose how long?

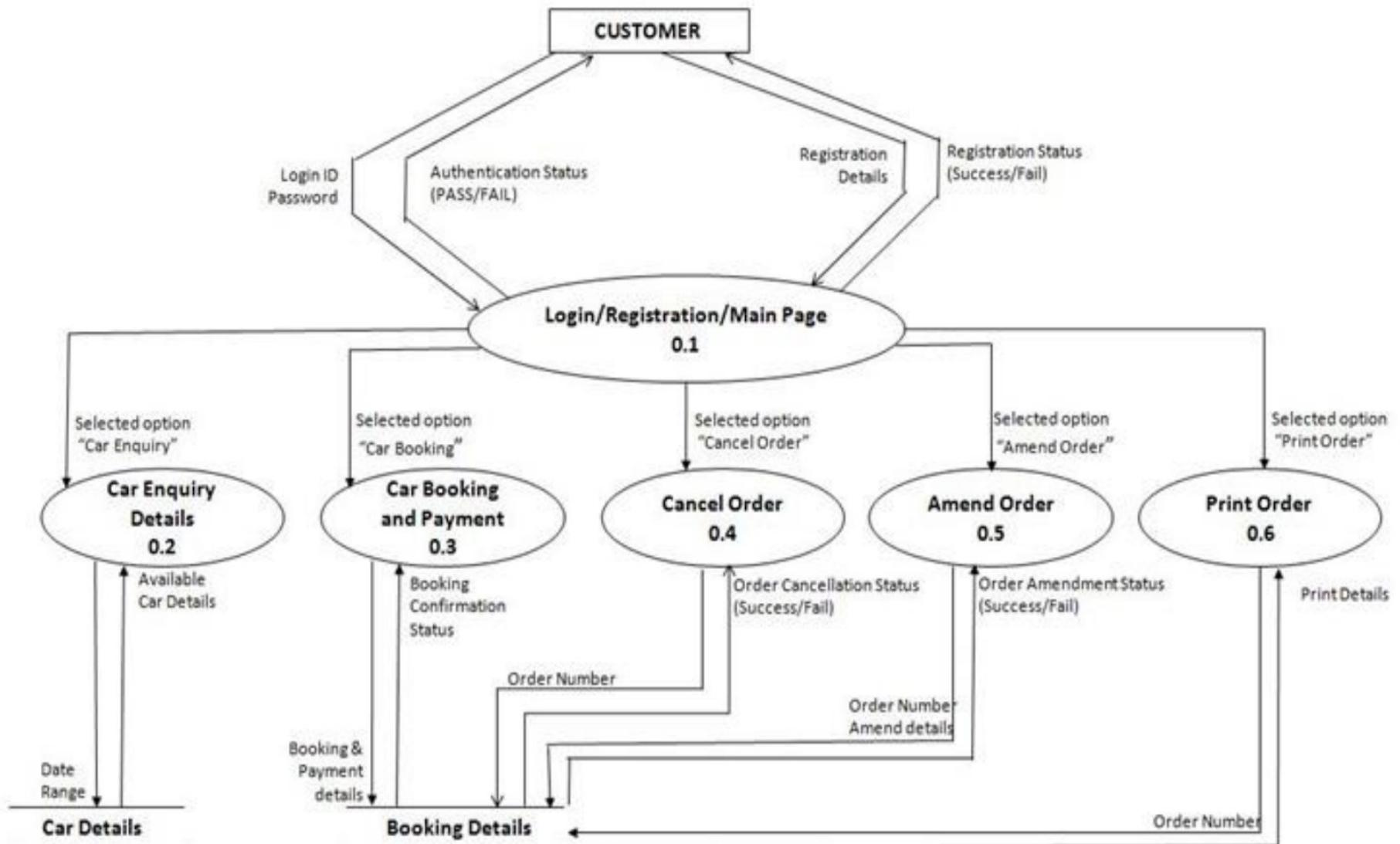


- ✓ **Decomposition of a bubble should be carried on until:**
 - ✓ The function of the bubble can be described using
 - ✓ A simple algorithm

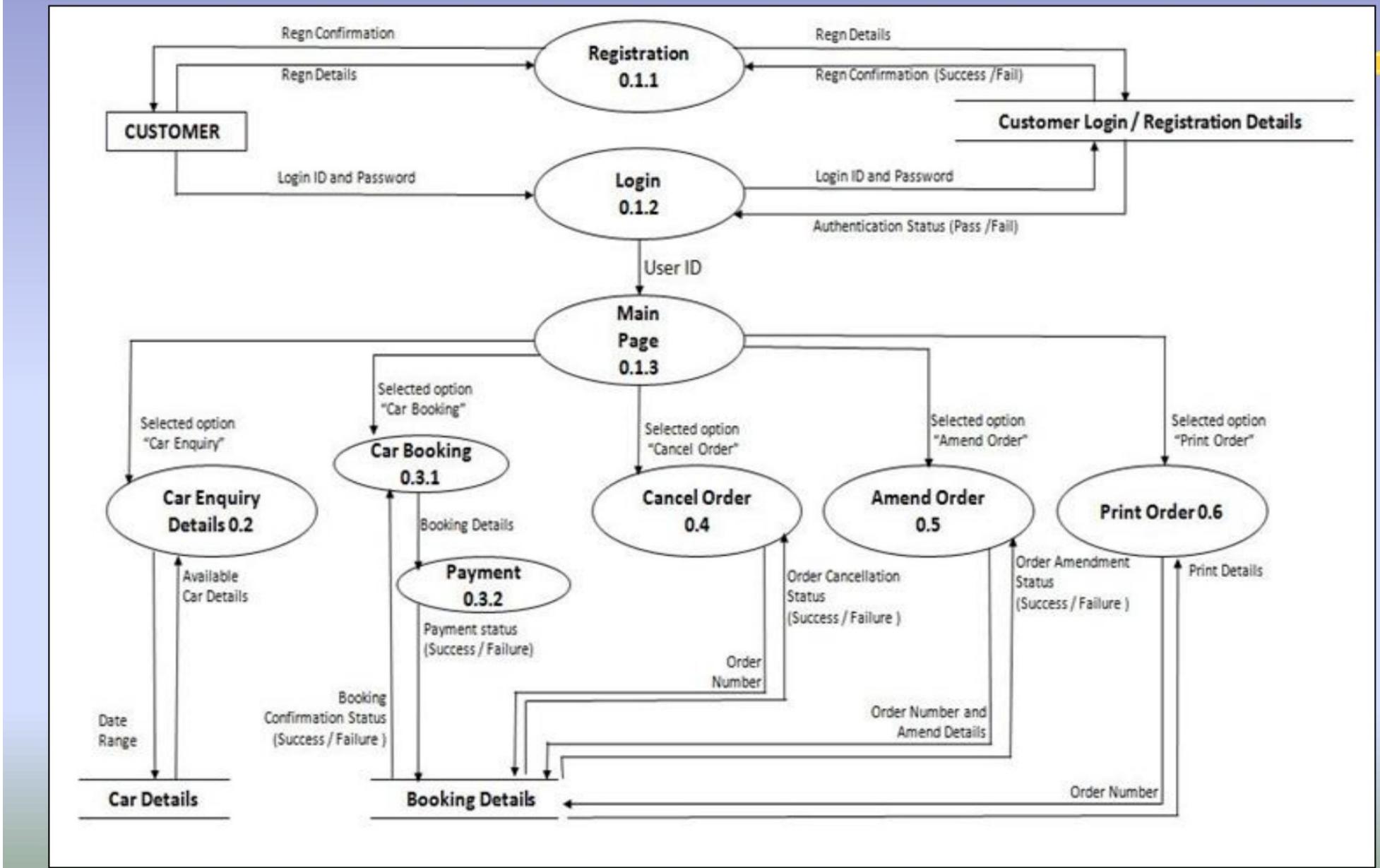
Ex1: Transport Service Automation System



DFD-1: Transport Service Auto. System



DFD-2: Transport Service Auto. System



- 
- Decomposition is stopped after reaching “simple set instruction” level:
 - A bubble is not decomposed any further:
 - If it can be represented by a simple set of instructions

Rule-1: Balancing a DFD

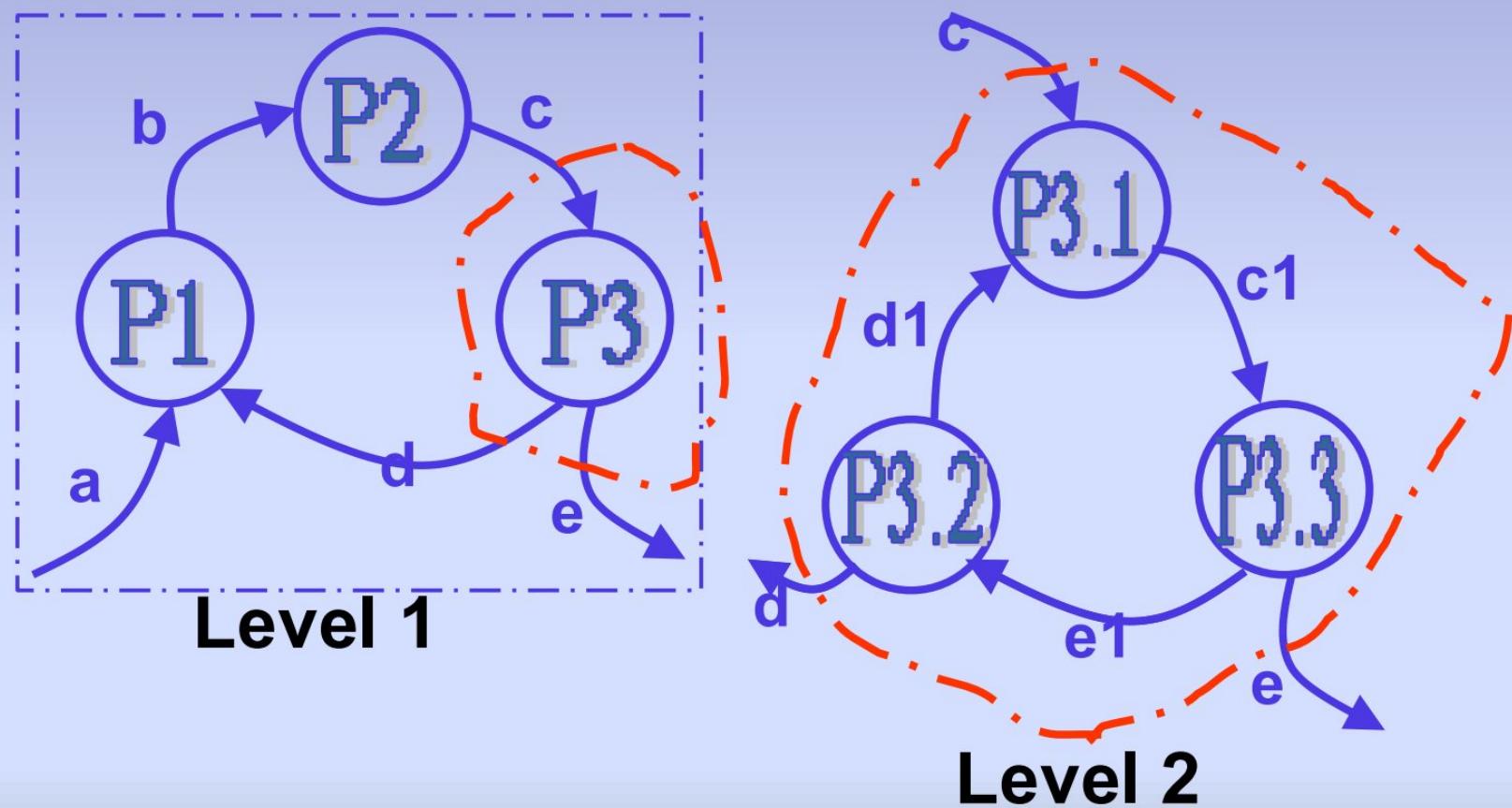
➤ Data flowing into or out of a bubble:

- Must match the data flows at the next level of DFD
- This is known as Balancing a DFD

➤ Ex: In the level 1 of the DFD of RMS

- Data item c flows into the bubble P3 and the data items d & e flow out
- In the next level, bubble P3 is decomposed
 - The decomposition is balanced as data item c flows into the level 2 diagram and d and e flow out

Balancing a DFD



Rule-2: Numbering of Bubbles

- **Number the bubbles in a DFD:**
 - Numbers help in uniquely identifying any bubble
- Bubble at **context level**, assigned **number 0**
- Bubbles at **level 1**: assigned numbers **0.1, 0.2, 0.3...**
- When a Bubble **numbered x is decomposed**,
 - its children bubble are numbered **x.1, x.2, x.3...**

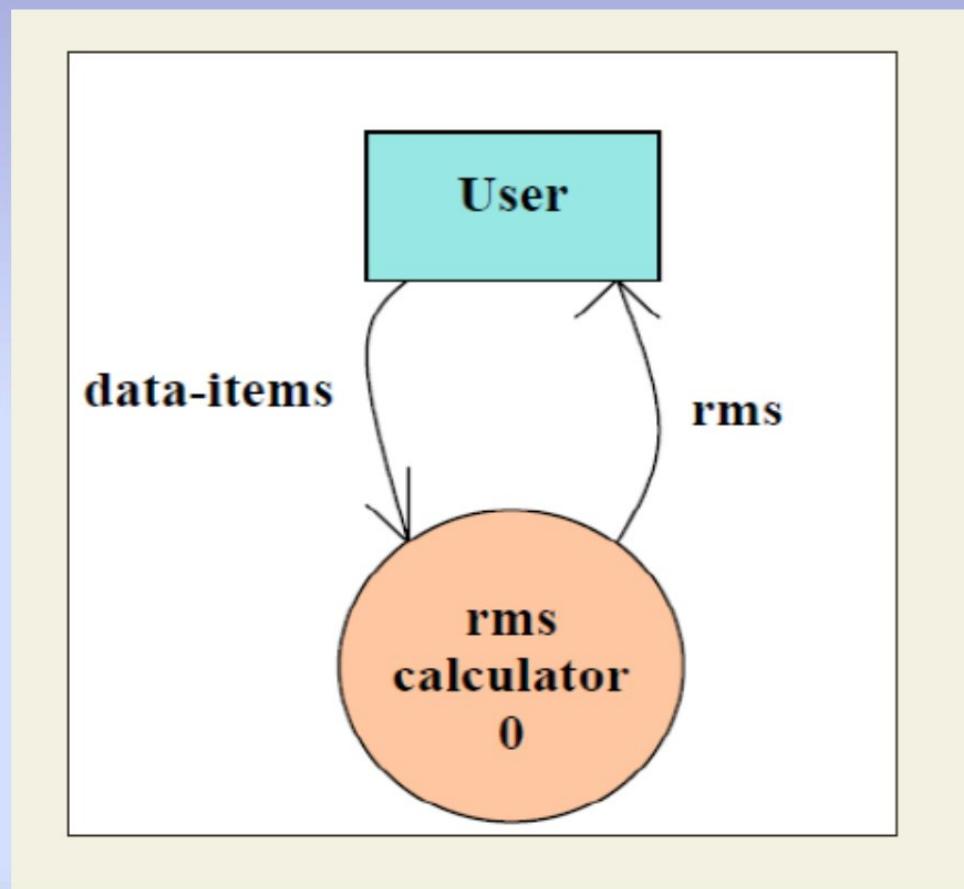
Ex2 : RMS Calculating Software

A software system called RMS calculating software would -

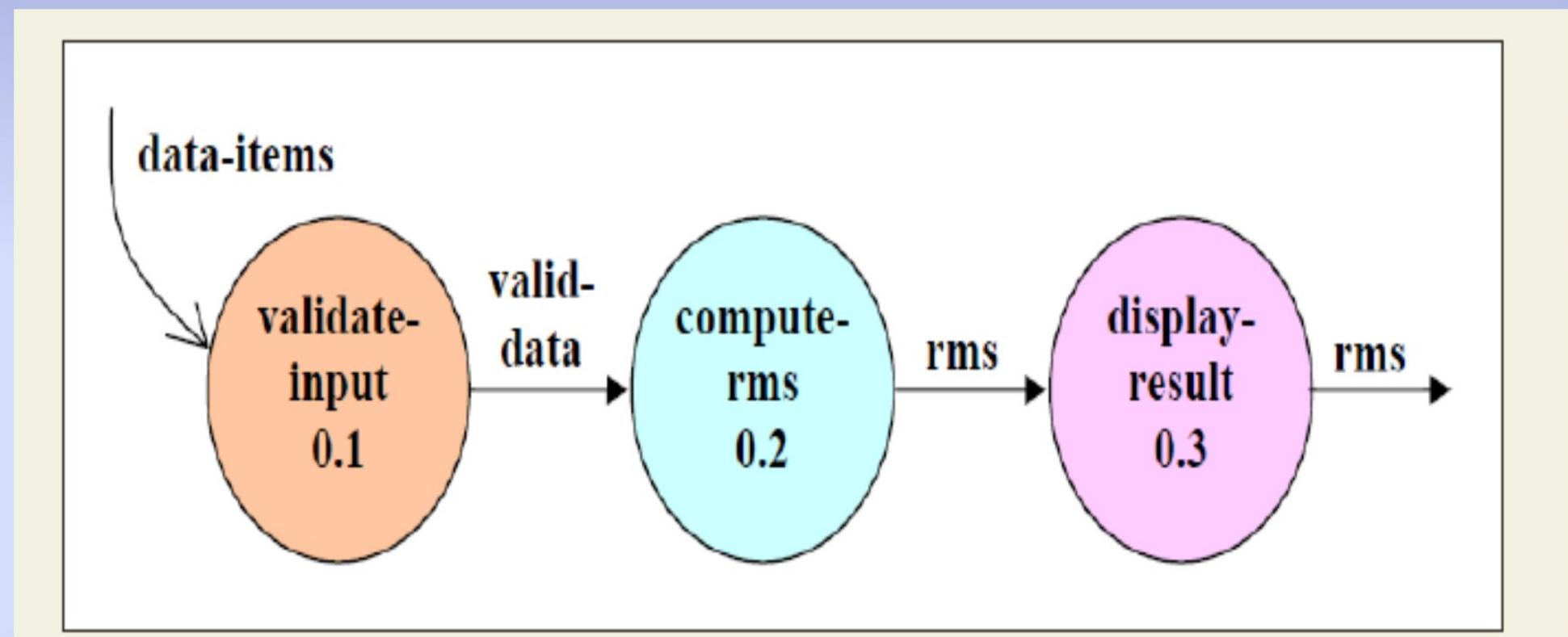
Read three integer numbers from the user in the range of -1000 and +1000 and then determine the root mean square (rms) of the three input numbers and display it.

The system accepts three integers from the user and returns the result to him.

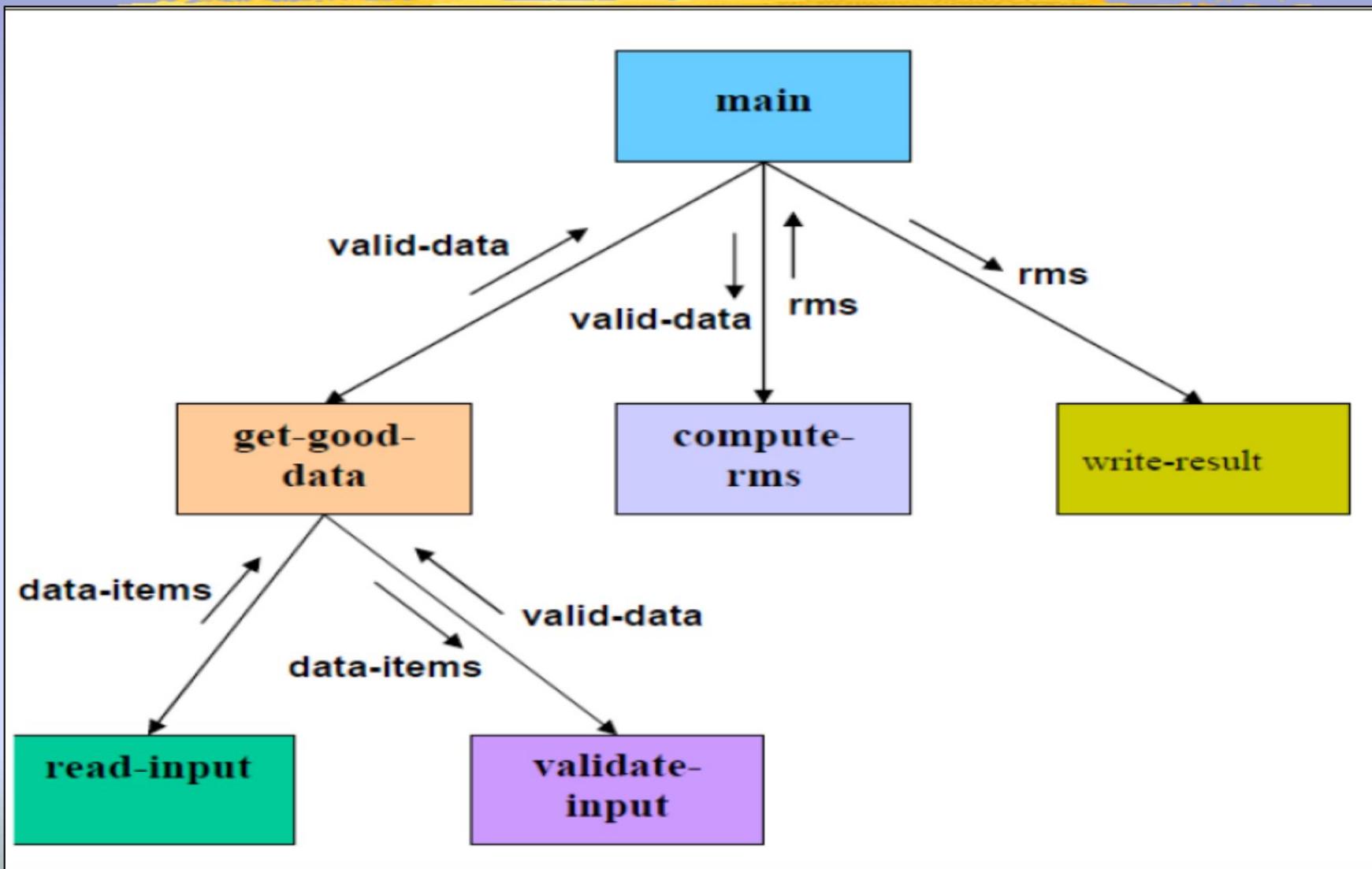
Context Diagram



Level 1



Structure chart



Data dictionary for RMS S/W

- **numbers=valid-numbers=a+b+c** * Composite i/p data *
- **a:integer** * input number *
- **b:integer** * input number *
- **c:integer** * input number *
- **asq:integer**
- **bsq:integer**
- **csq:integer**
- **squared-sum: integer**
- **Result=[RMS,error]**
- **RMS: integer** * root mean square value*
- **error:string** * error message*

Ex3 – Supermarket prize scheme

A supermarket needs to develop the following software to encourage regular customers.

For this, the customer needs to supply his/her residence address, telephone number, and the driving license number.

Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer.

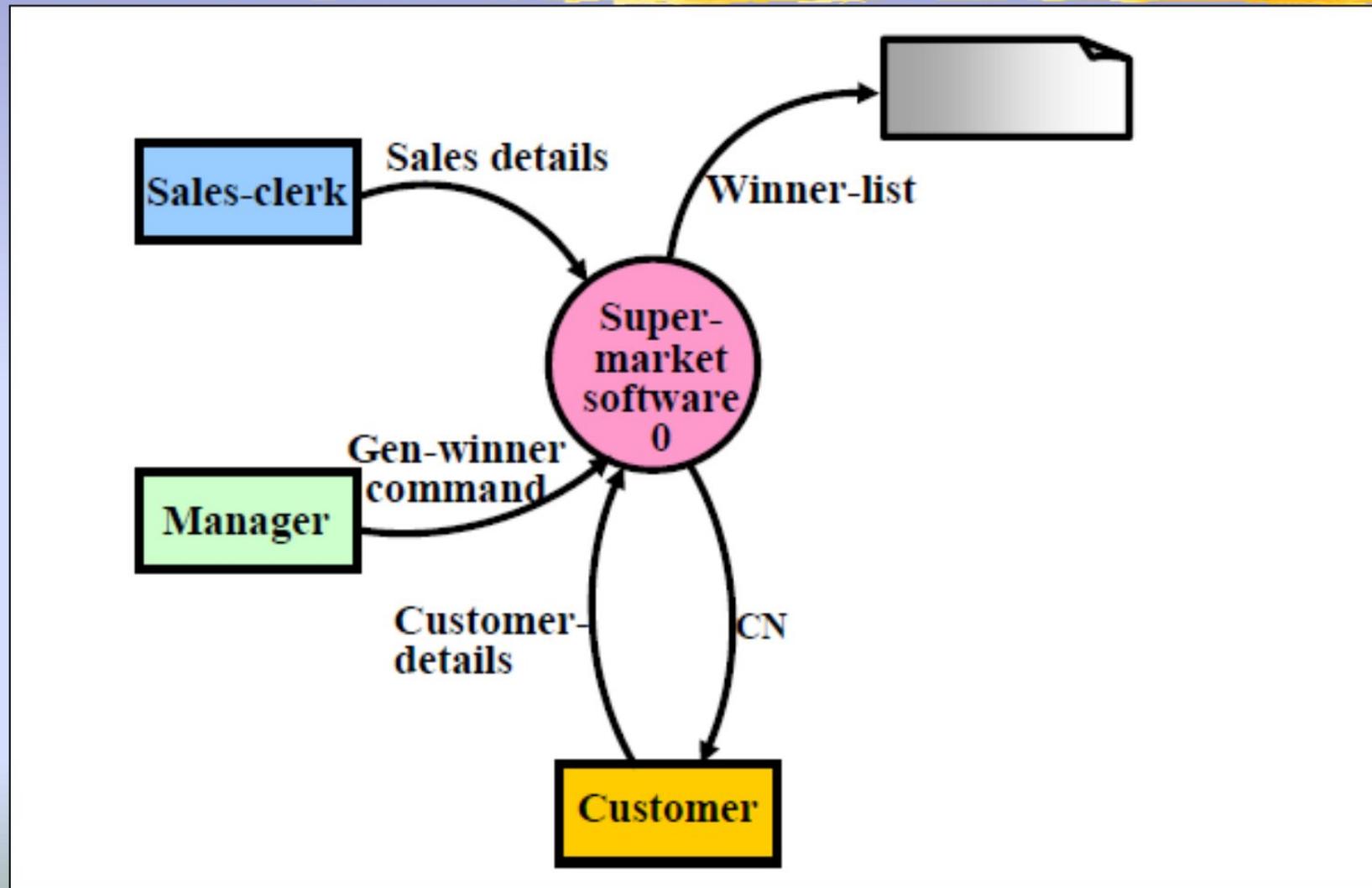
A customer can present his CN to the check out staff when he makes any purchase.

In this case, the value of his purchase is credited against his CN.

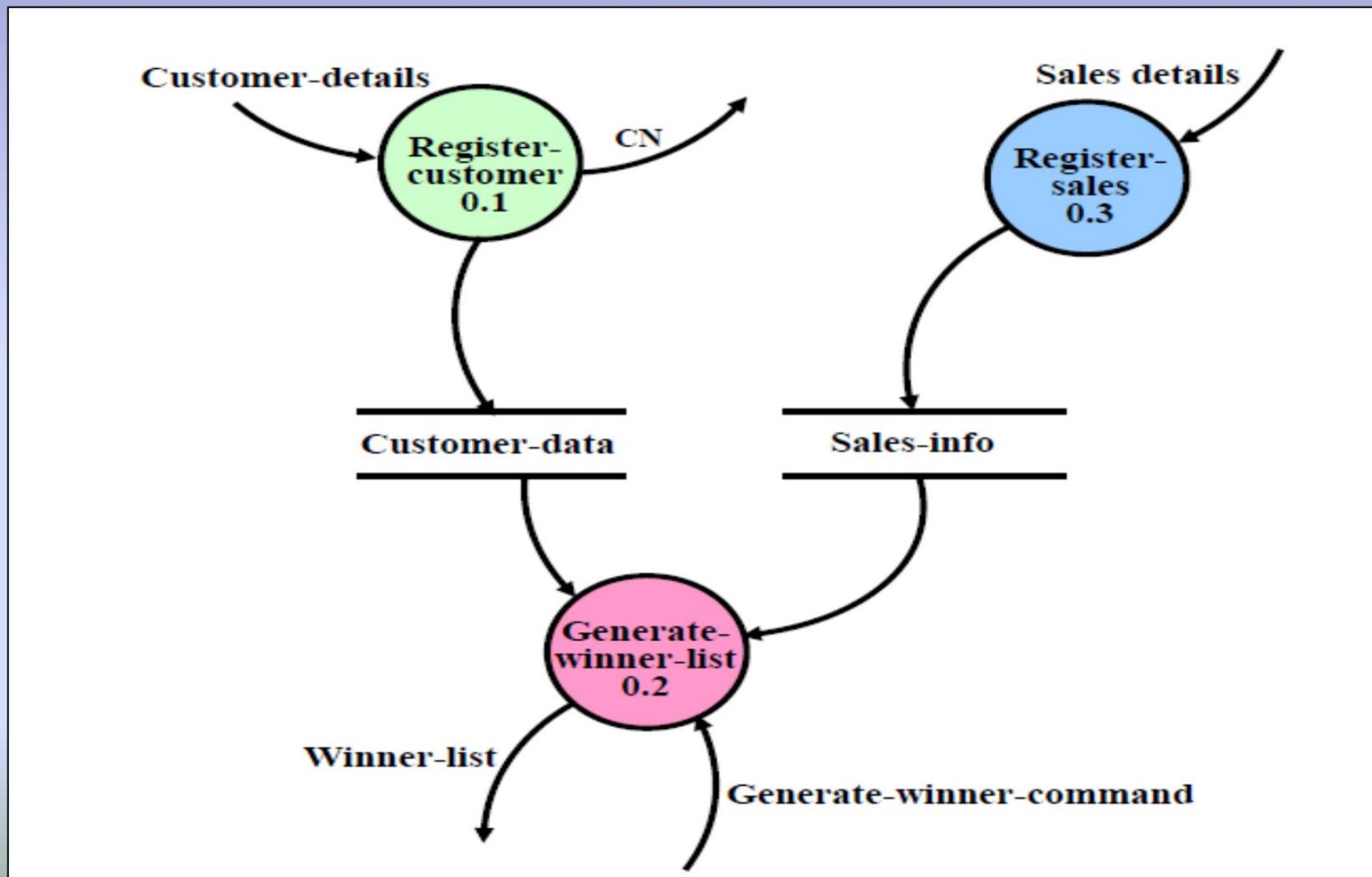
At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year.

Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs.10,000. The entries against the CN are reset on the day of every year after the prize winners' lists are generated

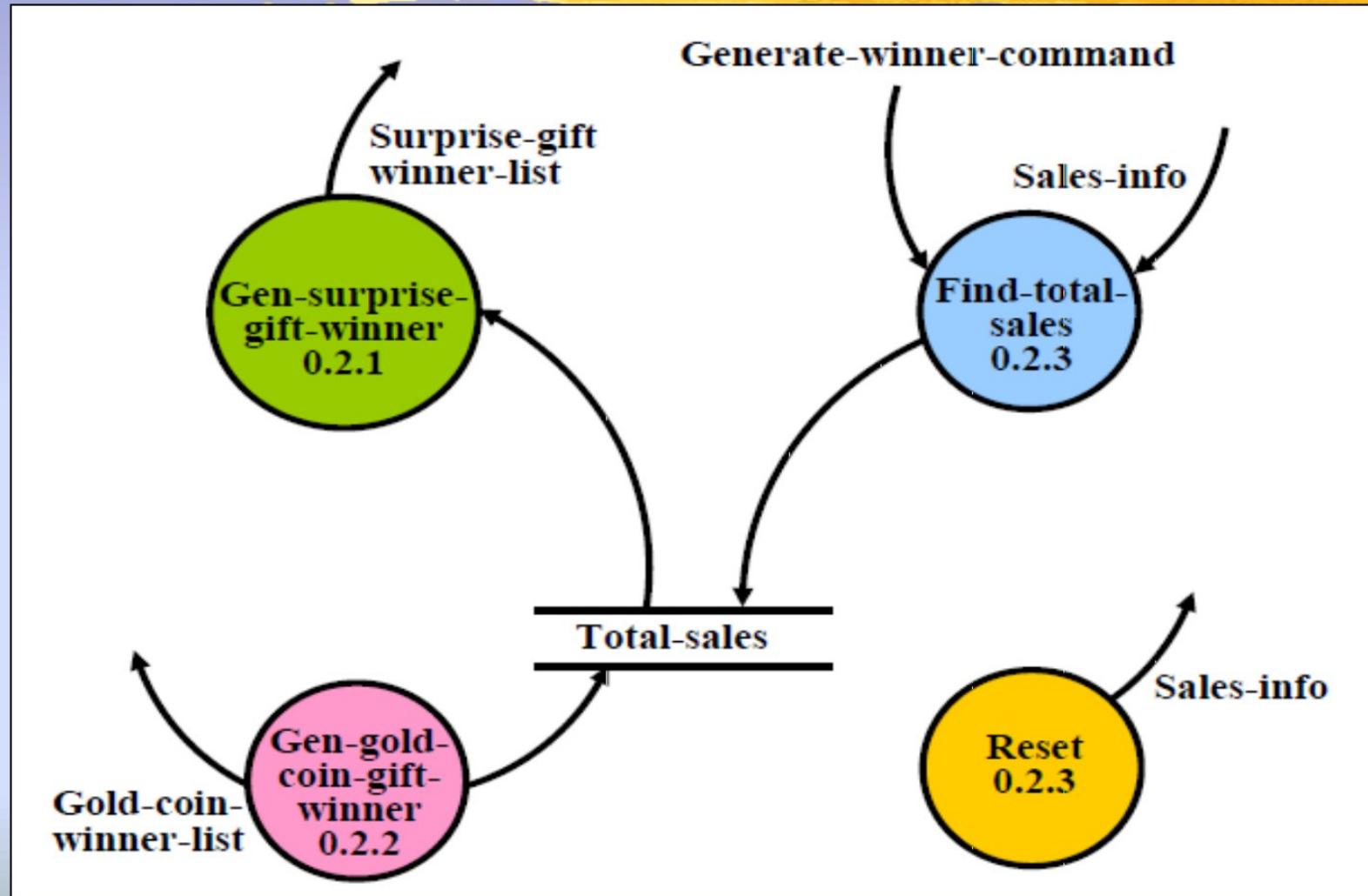
Context diagram



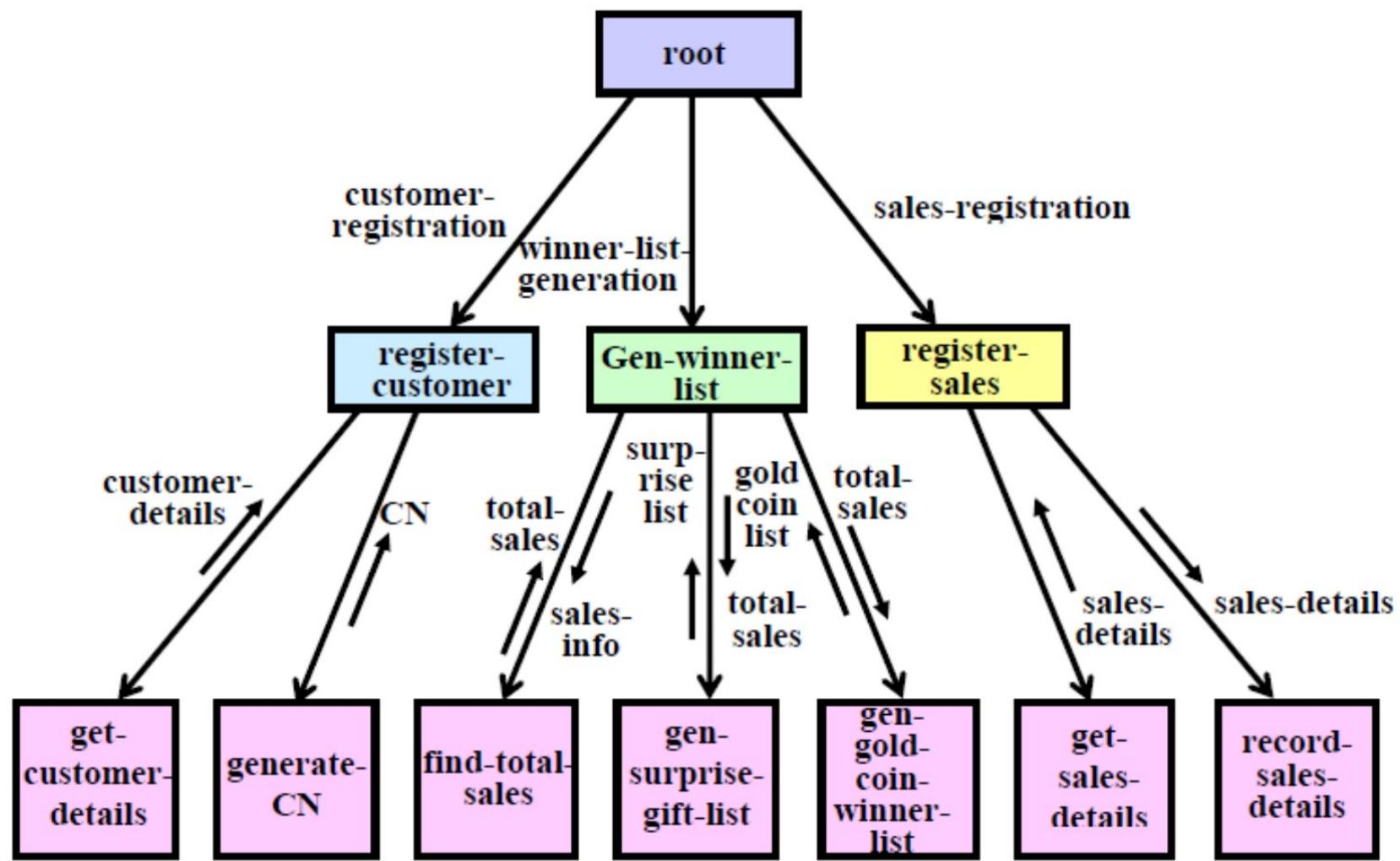
Level 1



Level 2



Structure chart



Data Dictionary - Supermarket prize scheme

Data Dictionary for the DFD Model

address: name + house# + street# + city + pin

sales-details: {item + amount}* + CN

CN: integer

customer-data: {address + CN}*

sales-info: {sales-details}*

winner-list: surprise-gift-winner-list + gold-coin-winner-list

surprise-gift-winner-list: {address + CN}*

gold-coin-winner-list: {address + CN}*

gen-winner-command: command

total-sales: {CN + integer}*

Commonly made errors

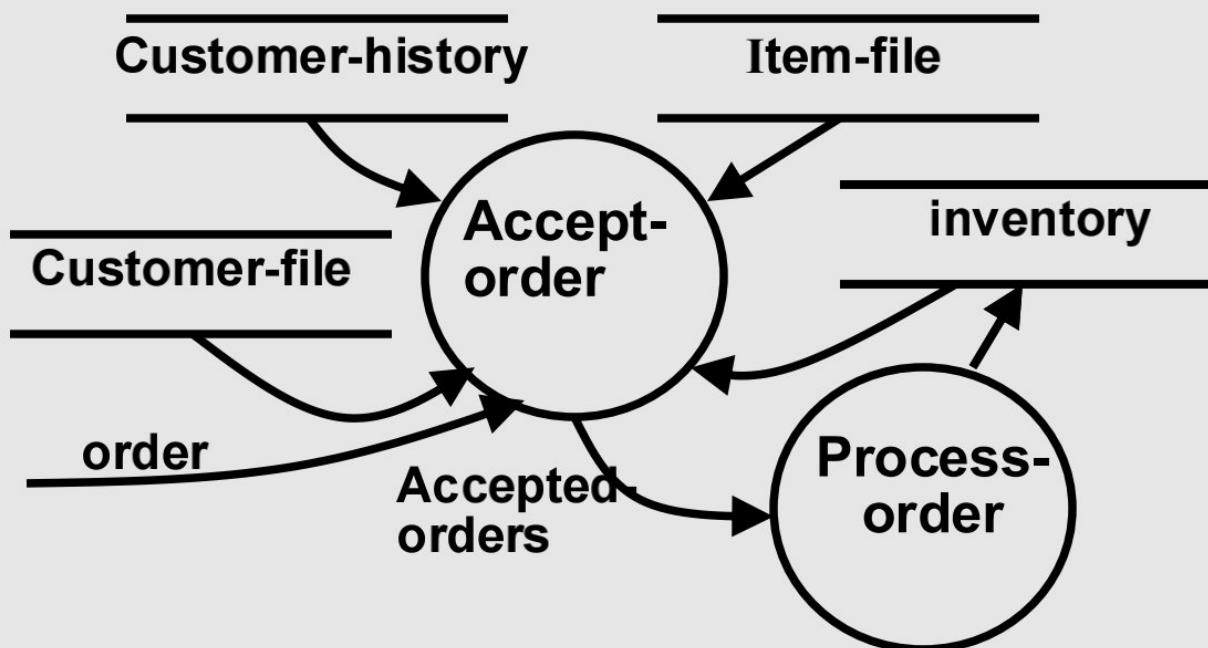
- Unbalanced DFDs
- Forgetting to mention the names of the data flows
- Unrepresented functions or data
- External entities appearing at higher level DFDs
- Trying to represent control aspects
- Context diagram having more than one bubble
- A bubble decomposed into too many bubbles in the next level (should be 3 to 7)
- Terminating decomposition too early
- Nouns used in naming bubbles

Shortcomings of the DFD Model

- DFD models suffer from several shortcomings:
- DFDs leave ample scope to be imprecise.
 - In a DFD model, we infer about the function performed by a bubble from its label
 - A label may not capture all the functionality of a bubble

Shortcomings of the DFD Model

- Control information is not represented:
 - For instance, order in which inputs are consumed and outputs are produced is not specified.

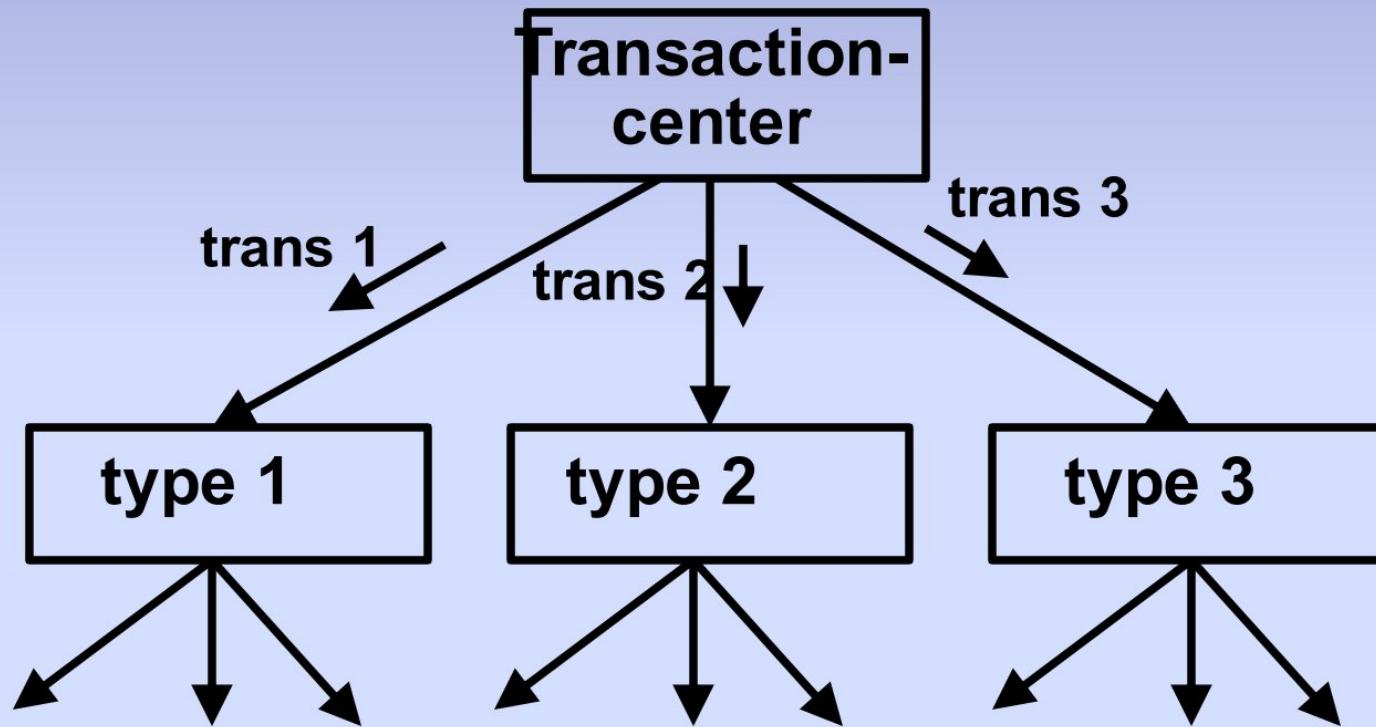


Shortcomings of the DFD Model

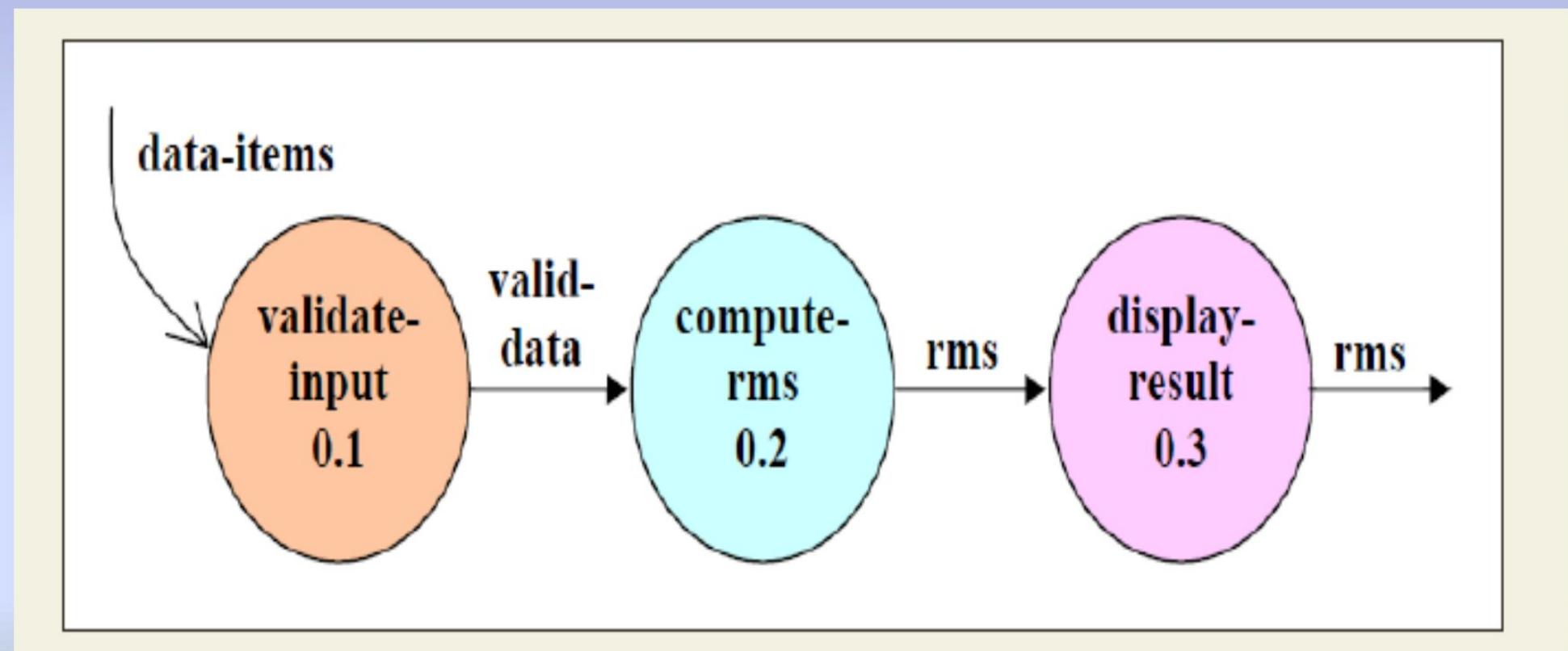


- A DFD does not specify synchronization aspects:
 - For instance, the DFD in TAS example does not specify:
 - whether **process-order** may wait until the **accept-order** produces data
 - whether **accept-order** and **handle-order** may proceed simultaneously with some buffering mechanism between them.

Transaction analysis



Transform Analysis





THE END

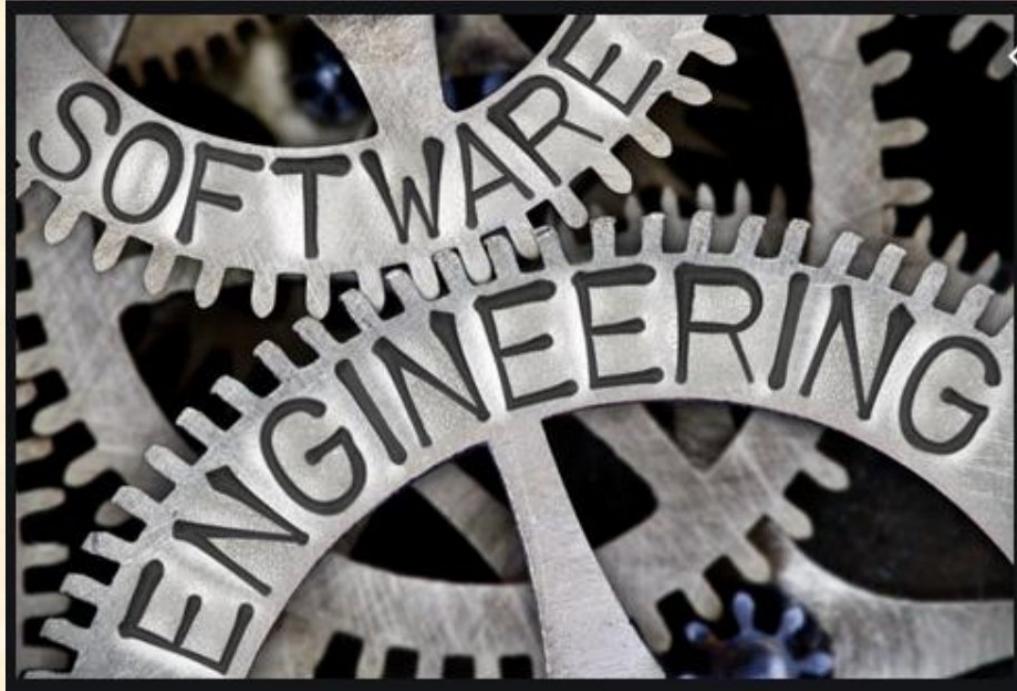
100



Scanned with OKEN Scanner

Software Engineering & UML

Module-4



Faculty :
Dr. Suchismita Rout
Associate Professor

Coding Phase

- **During coding phase:**
 - Every module identified in the design document is
 - Coded &
 - Unit tested
- **Unit testing:**
 - Testing of different modules/units of a system
 - in isolation

Unit Testing

- Test each module in **isolation** first
- Later integrate the modules and test the set of modules
 - This makes **debugging** easier
- If an error is detected when several modules are being tested together
 - It would be **difficult** to determine which module has the error

Coding

- The **Inputs** to the **Coding phase**:
 - The **Design documents**
 - **Structure chart of the system**
 - **Module Specification** (data structures & algorithms)
 - **Modules** in the **design document** are **coded**
 - As per **module spec**

Coding

- **Objective** of coding phase:
 - *Transform design into code*
 - Unit test the code

Coding Standards

- **Good software development organizations** require their programmers to:
 - Adhere to some standard style of coding called **coding standards**
 - Some formulate their *own coding standards* & follow them rigorously
- **Advantages:**
 - Gives an uniform appearance to the **codes** written by different engineers
 - Enhances code understandability
 - Encourages good programming practices

Coding Standards & guidelines

- Sets **standard ways** of:
 - *variables naming & Initializations*
 - *commenting (header, blocks)*
 - *code layout*
 - *do's & don'ts*
 - *max source lines allowed per function* etc.

Example Coding Standards

- **Rules** for *limiting* the **use of global variables**
- **Naming conventions** for
 - Global variables, local variables and constants
- **Content & format of headers** for different modules
 - *Name of the module*
 - *date created*
 - *author's name*
 - *modification history*
 - *Description*
 - *input/output parameters etc..*

Example Coding Standards

- **Error / Exception handling** mechanisms
- **Code** should be easy to understand & maintainable
- Avoid unpredictable side effects
- **Do not use** an identifier for multiple purposes
 - Leads to confusion and annoyance for anybody trying to understand the code
 - Also makes maintenance difficult

Coding Guidelines

- Code should be **well-documented**
- **Rules of thumb:**
 - On average there must be at least
 - ***One comment line for every three source lines***
- **Do not use goto statements.** make a program unstructured & difficult to understand

Code inspection vs code walk throughs

- After completion of coding of the module :
 - **Code inspection** & **code walk throughs** are carried out
- **Objectives**
 - To ensure **coding standards** are followed
 - To **detect** as many **errors** as possible **before testing**
 - Detected errors **require less effort** for **correction** at this stage
 - Much **higher effort** needed if errors were to be detected during **integration or system testing**

(1) Code Walk Through

- Is an **informal** *code analysis technique*
- Some members (3-7) of project team select important test cases & simulate execution of the code **manually**
- This technique is based on personal experience, common sense etc..
- Discussion should focus on discovery of errors & **not on how to fix them**
- Avoid the developer to feel evaluated

(2) Code Inspection

- Aims mainly at **discovery** of **commonly made errors**
- The code is examined for the *presence of certain kinds of errors* (**Ex:** GOTO statements, memory deallocation ...)
- Good software development companies:
 - Collect **statistics** of *errors committed* by their engineers
 - Identify the types of **errors most frequently committed**
- A **list of common errors**:
 - Can be used during code inspection to look out for possible errors

Commonly made errors

- Adherence to coding standards
- Use of uninitialized variables
- Non terminating loops
- Array indices out of bounds
- Incompatible assignments
- Improper storage allocation and de-allocation
- Actual and formal parameter mismatch in proc calls
- Jumps into loops

Testing



Contents

- ✓ **Testing**

- ✓ What is testing
- ✓ Verification **Vs** Validation
- ✓ **Strategies to Design test cases**
- ✓ Black box testing
- ✓ White box testing

- ✓ **Levels of testing a software product undergoes**

- ✓ Unit testing
- ✓ Integration testing
- ✓ System testing

- ✓ **Debugging**

What is testing ?

Testing a program:

- Providing the program with a **set of test inputs**
 - Observing if the program behaves as expected
- If the program fails,
 - Then the conditions are noted for later debugging & correction

Commonly used terms

➤ **Test case:**

This is the Triplet [I,S,O], where I is the data input to the system, S is the state of the system, and O is the expected output of the system

➤ **Test suite:**

This is the **set of all test cases** with which a given software product is to be tested

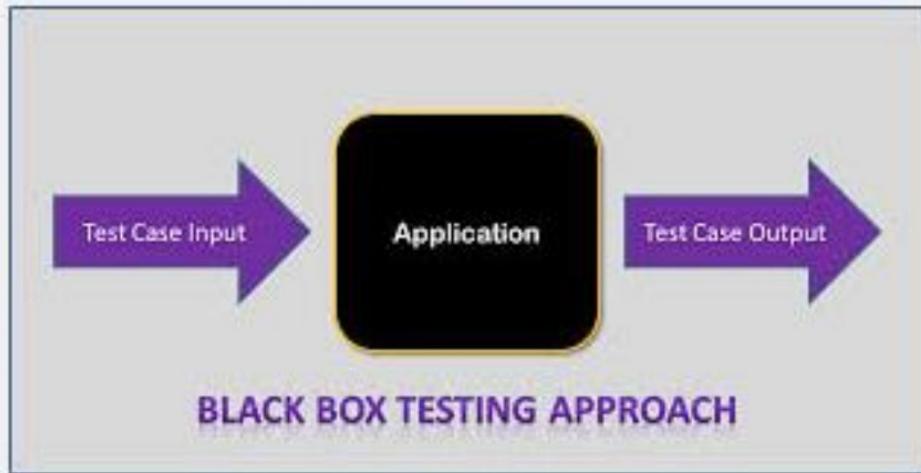
Verification vs Validation

- ✓ **Verification** is the process of determining:
 - ✓ Whether **output of one phase** of development **conforms** to its **previous phase**
Ex: Code -> Design Docs -> SRS Doc
 - ✓ Concerned with **phase containment of error**
- ✓ **Validation** is the process of determining
 - ✓ Whether a **fully developed system** **conforms** to its **SRS document**
 - ✓ Concerned with **product** to be **error free**

Design of Test Cases

- ✓ **Exhaustive testing** of any system is **impractical**:
 - ✓ As input data domain is extremely large
- ✓ **Goal:** Design an **optimal test suite**:
 - ✓ Of reasonable size &
 - ✓ Uncovers as many errors as possible
- ✓ Systematic approaches are required to design an **optimal test suite**:
 - ✓ ***Each test case in the suite should detect different errors***
- ✓ **Two main approaches of designing test cases are:**
 1. **Black Box testing approach**
 2. **White box testing approach**

Black-box Testing



- Test cases are designed without any knowledge of :
- The internal structure of the module or software

White-box Testing

- ✓ Designing white-box test cases:
 - ✓ Requires **knowledge** about the **internal structure** of software or module
 - ✓ White-box testing is also called structural testing
- ✓ Ways of **designing** white-box test cases:
 - ✓ Statement Coverage
 - ✓ Branch coverage
 - ✓ Condition coverage
 - ✓ Path coverage
 - ✓ Control flow graph (CFG)

Black-box Test case preparation approach

- ✓ There are essentially **two main approaches** to design black box test cases:

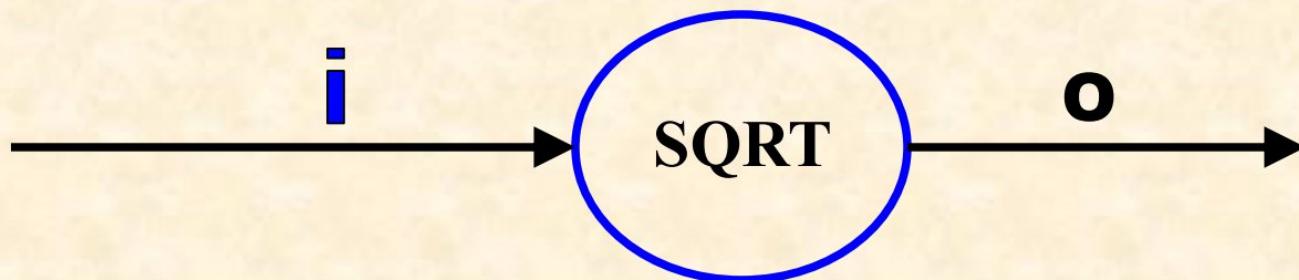
1. **Equivalence class partitioning**

- ✓ Input values are *partitioned* into equivalence classes.
- ✓ Partitioning is done such that: program behaves in similar ways to each input value in an equivalence class

2. **Boundary value analysis**

Example

- ✓ A program reads an input value in the range of 1 and 5000:
 - ✓ Computes the square root of the input number



- ✓ There are **three** equivalence classes:
 - ✓ The set of negative integers,
 - ✓ Set of integers in the range of 1 and 5000,
 - ✓ integers larger than 5000.



Example (cont.)

- ✓ The **test suite** must include:
 - ✓ Representatives from each of the three equivalence classes:
 - ✓ A possible test suite can be: {-5,500,6000}.

Boundary Value Analysis

- ✓ Some typical programming errors occur at **boundaries** of equivalence classes as Programmers often **fail to see**

- ✓ Special processing is required at the boundaries

- ✓ **Ex:** Programmers may improperly use < instead of <=

Boundary value analysis:

- ✓ Selects test cases at the boundaries of different equivalence classes.

- ✓ **Ex:** For a function that computes the square root of an integer in the range of 1 and 5000:

- ✓ Test cases must include the values: **{0,1,5000,5001}**

Testing

- ✓ **Aim:** to identify *all defects* in a s/w product
- ✓ But, even after thorough testing, one cannot guarantee that the s/w is error-free
- ✓ Vary large input data domain makes it impractical to test for each input value
- ✓ But testing exposes & reduces many defects in the system
- ✓ Increases the users' confidence
- ✓ Testing is very important: Needs most time/effort of all phases
- ✓ As much challenging as other phases & needs creativity

Unit testing

- ✓ Software products are **tested at three levels:**
 - **Unit testing, Integration testing & System testing**

Unit testing

- ✓ Modules are **tested in isolation**
 - ✓ If all modules are tested together, it is difficult to find which module has the error
- ✓ **Unit testing reduces debugging effort** several folds
 - ✓ Programmers do unit testing **immediately after coding** of a module

Integration testing

- ✓ After all modules of a system have been coded and unit tested:
- ✓ Modules are **integrated in steps** according to an integration plan
- ✓ Partially integrated system is tested at each integration step

System Testing

- ✓ **System testing** involves:
 - ✓ **Validating a fully developed system against its requirements (SRS Doc)**

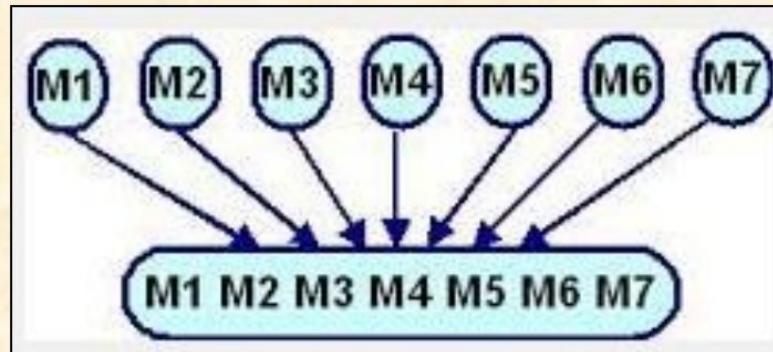
Integration Testing

- ✓ Develop *integration plan* by examining the structure chart
 - 1. **Big bang approach**
 - 2. **Top-down approach**
 - 3. **Bottom-up approach**
 - 4. **Mixed approach**

(1) Big bang Integration Testing

✓ **Simplest** integration testing approach:

- ✓ All the modules are put together at once & tested
- ✓ This technique is used for small systems

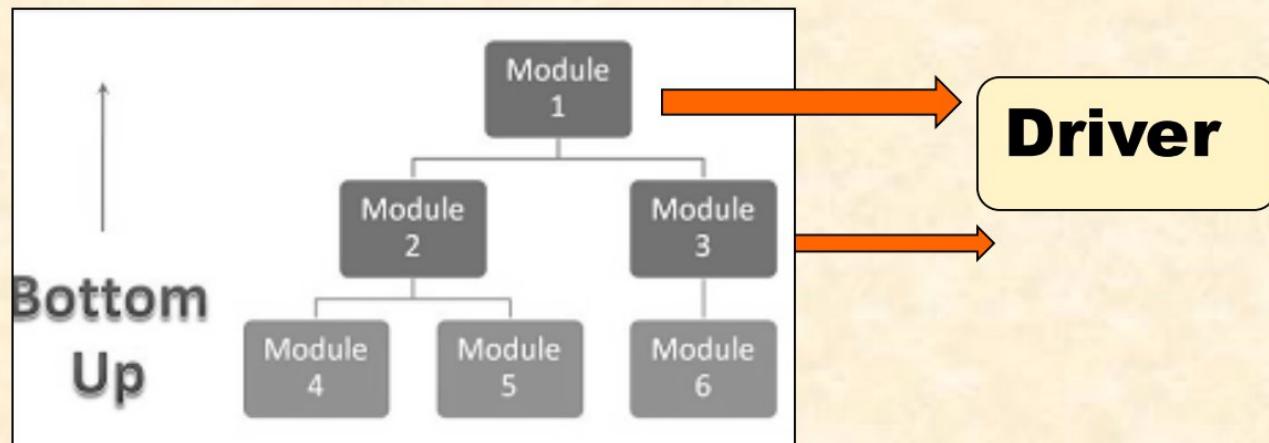


✓ **Problems:**

- ✓ Very difficult to **localize the error** (the error may belong to any of the modules)
- ✓ The errors found during big bang integration testing are very **expensive to fix**

(2) Bottom-up Int. Testing

- ✓ Integrate & test the **bottom level modules first**



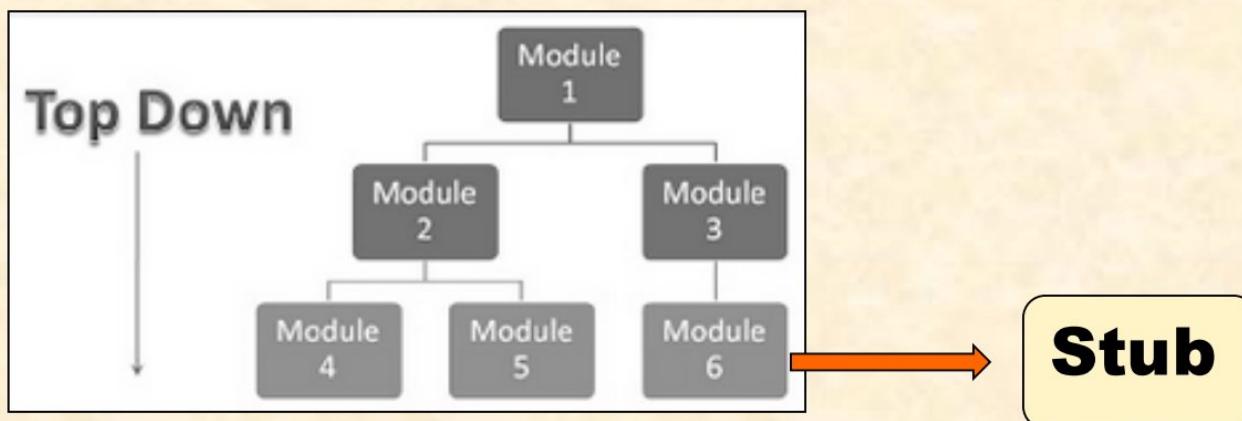
Driver modules are dummy modules used to replace top level modules if they are not ready

- ✓ **Disadv:**

- ✓ Difficult when the system has a large no. of small subsystems

(3) Top-down integration testing

Starts testing with **main** & **top-level** modules in the system



Stub modules are dummy modules used to replace lower level modules if they are not ready

- ✓ Next, immediate lower level modules are combined & tested

(4) Mixed integration testing

- ✓ **Mixed I.T** (most common approach) uses both top-down & bottom-up testing approaches
- ✓ **In top-down approach:**
 - ✓ Testing **waits** till all top-level modules are coded & unit tested
- ✓ **In bottom-up approach:**
 - ✓ Testing can start only after bottom level modules are ready

System Testing

- ✓ There are **three main kinds** of system testing:
- ✓ **Alpha Testing:**
 - ✓ Performed by **the test team** within the *developing organization*
- ✓ **Beta Testing**
 - ✓ Performed by a **select group of friendly customers**
- ✓ **Acceptance Testing**
 - ✓ Performed by the **customer himself** to determine
 - ✓ whether the system should be **accepted or rejected**

White-Box Testing - Methods

- 1. Statement coverage**
- 2. Branch coverage**
- 3. Condition coverage**
- 4. Path Coverage**
- 5. CFG – Cyclomatic complexity**

(1) Statement Coverage

- ✓ Design test cases so that
 - ✓ Every statement in a program is executed at least once
 - ✓ Unless a statement is executed, we have no way of knowing if an error exists in that statement

(2) Branch Coverage

- Test cases are designed such that:
 - All branch conditions are given true and false values in turn
 - **If (condition) Then**
 - -----
 - -----
 - **Else**
 - -----
 - -----

(3) Condition Coverage

- ✓ Test cases are designed such that:
 - ✓ Each component of a composite conditional expression is
 - ✓ Given both true & false values

EX:

- Consider the conditional expression
 - **((c1.and.c2).or.c3):**
- Each of c1, c2, and c3 are given true and false values

(4) Path Coverage

- ✓ Design test cases such that:
 - ✓ All **linearly independent paths** in the program are executed *at least once*
- ✓ **linearly independent paths** are determined from **Control flow graph (CFG)** of a program
 - The sequence in which, the different instructions of a program get executed
 - The way control flows through the program

Cyclomatic Complexity

- Set of **independent paths** through the program flow graph
- A quantitative measure of **testing difficulty**

Formulae :

1. $V(G) = E - N + 2$

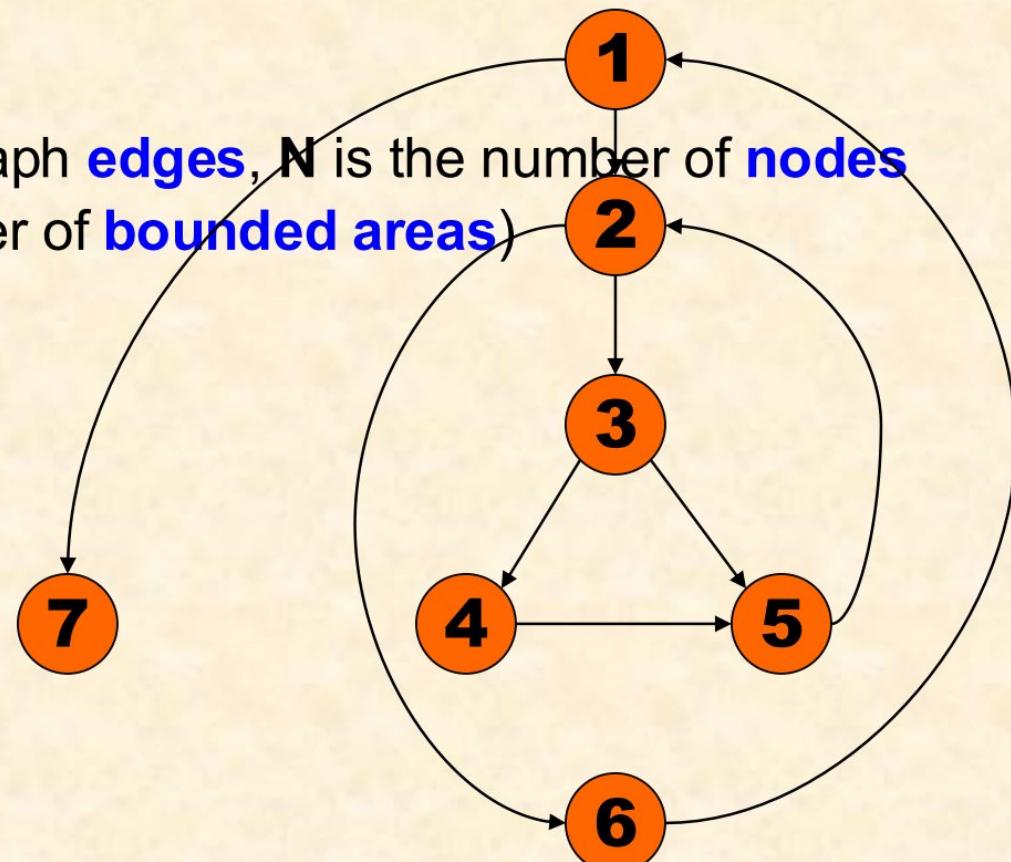
➤ **E** is the number of flow graph **edges**, **N** is the number of **nodes**

2. $V(G) = P + 1$ (**P** is the number of **bounded areas**)

3. **No. of independent paths**

Example

```
while (i<n-1) do
    j = i + 1;
    while (j<n) do
        if A[i]<A[j] then
            swap(A[i], A[j]);
        end do;
        i=i+1;
    end do;
```



Computing $V(G)$:

➤ 1. $V(G) = 9 - 7 + 2 = 4$

➤ 2. $V(G) = 3 + 1 = 4$

➤ 3. Basis Set (No. of Independent paths)

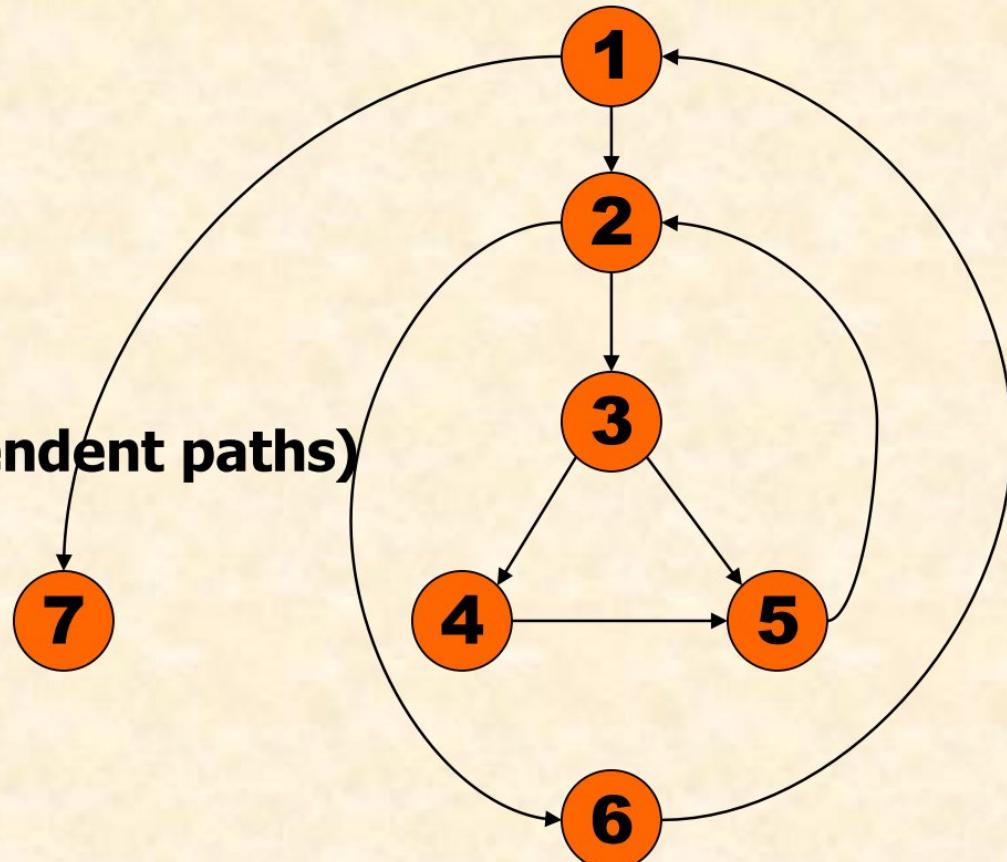
➤ 1, 7

➤ 1, 2, 6, 1, 7

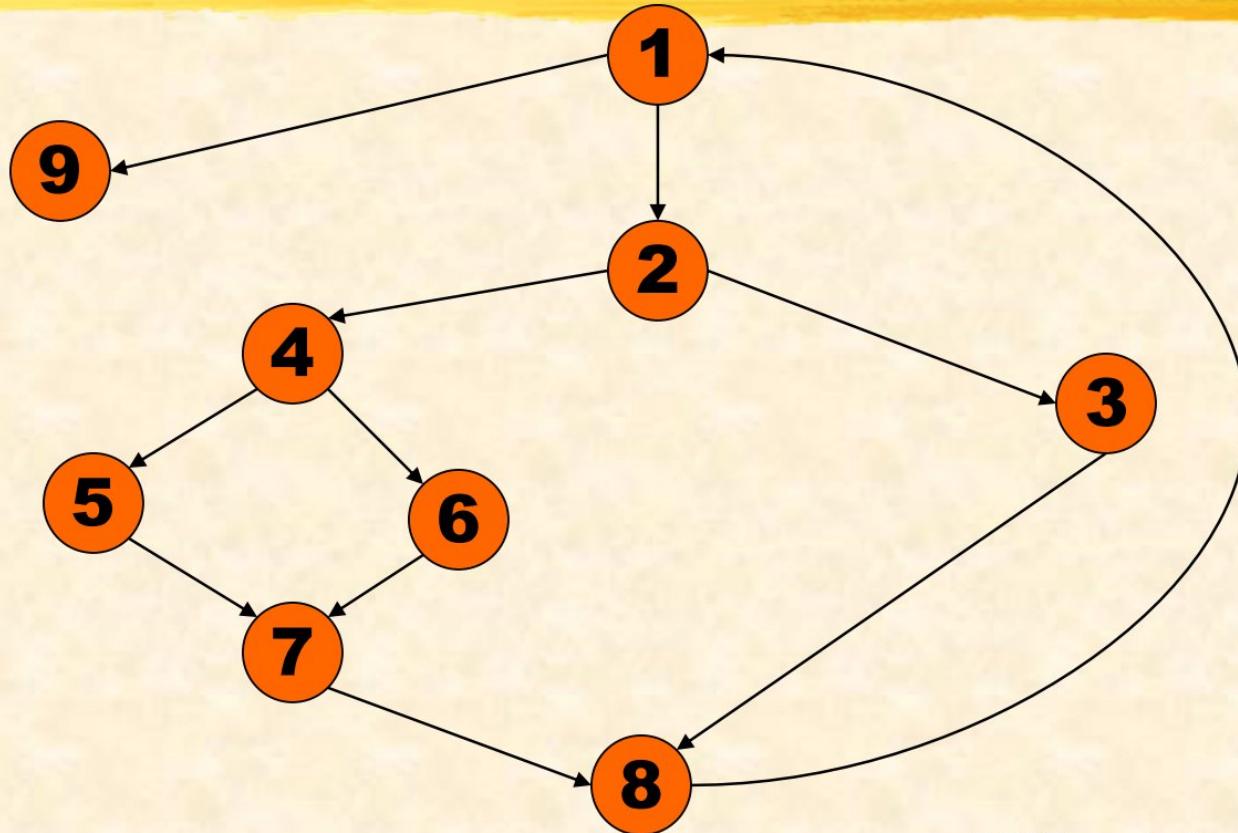
➤ 1, 2, 3, 4, 5, 2, 6, 1, 7

➤ 1, 2, 3, 5, 2, 6, 1, 7

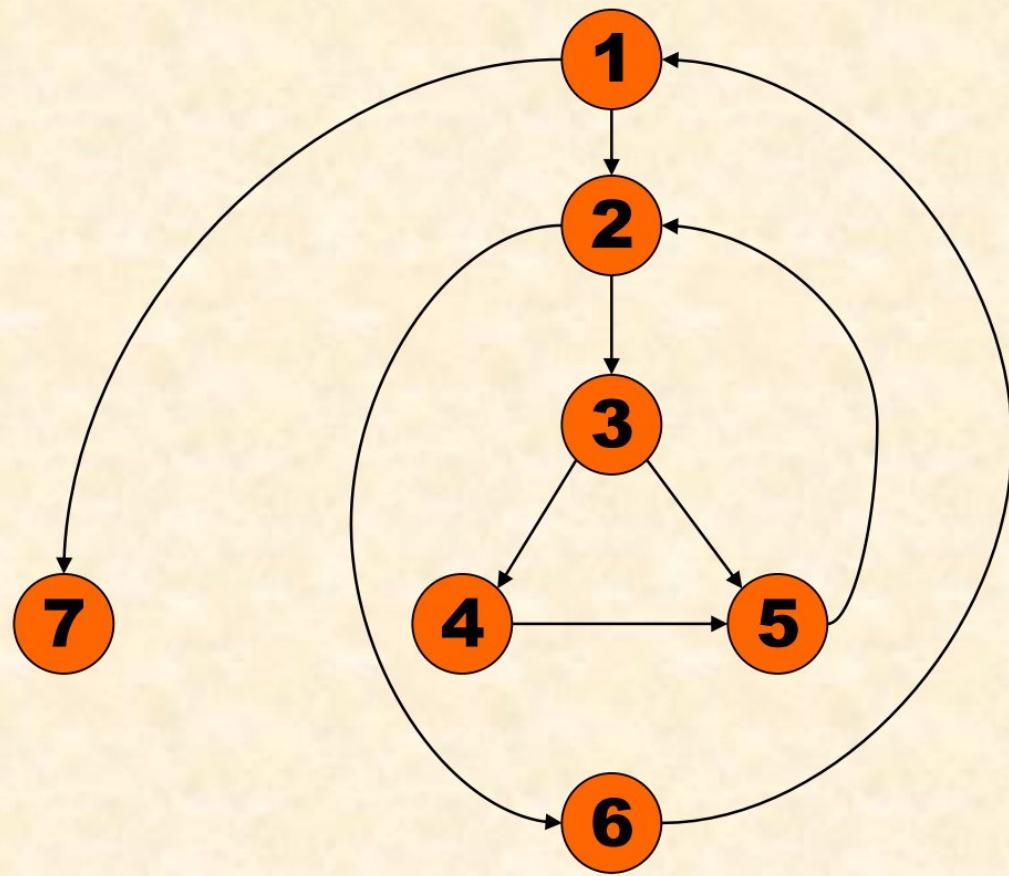
➤ If value of $V(G)$ is more than 10 - testing is very difficult



Another Example



What is $V(G)$?

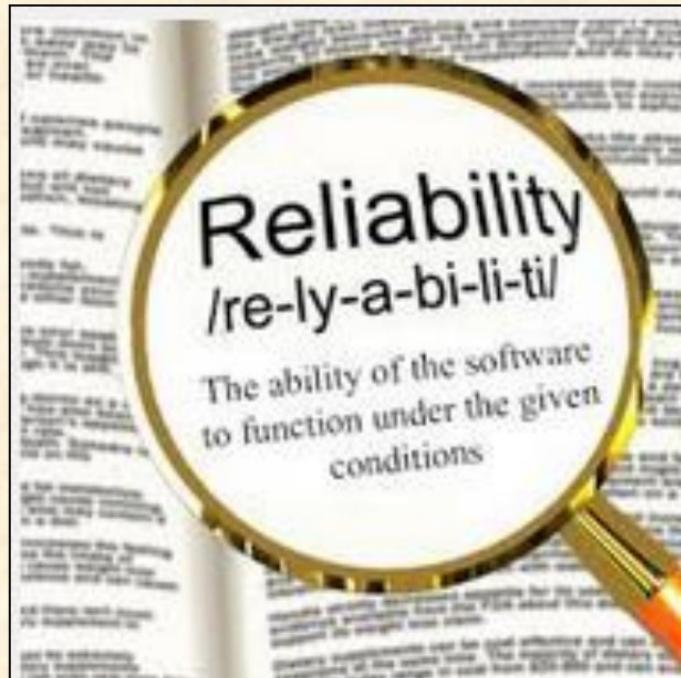
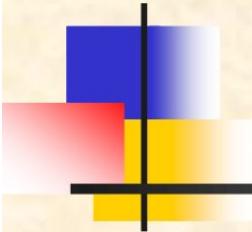


Contents



- **Regression testing**
- **Mutation testing**

Software Reliability





CONTENTS!

- Introduction
- Reliability metrics
- Reliability growth modelling
- Statistical testing

Introduction

- “**Reliability**” of a software product:
- Working **consistently** and **correctly** over a period of time
- Product’s **trustworthiness** or **dependability**
- An **imp. attribute** determining the quality of the product
- A Product having a large number of **defects** is **unreliable**

Introduction

- **Users** always want **highly reliable products**
 - Want **quantitative estimation** of **reliability** before making ***buying decision***
 - **Accurate measurement** of **software reliability**:
 - A very **difficult** & depend upon **several factors**

Software Reliability

- Different users use a software product in different ways
 - Defects which show up for one user, may not show up for another
- Reliability of a software product:
 - Is observer-dependent & cannot be determined absolutely
- Software reliability changes through out the product life
 - Each time an error is detected and corrected

Major Software Reliability Metrics

Rate of occurrence of failure (ROCOF):

- **ROCOF** measures:
 - Frequency of occurrence of failures
 - Observe the behaviour of a software product:
 - Over a specified time interval & calculate the total number of failures during the interval
 - Appropriate for s/w products

Mean Time To Failure **(MTTF)**

- **Average time** between two successive failures:
 - Observed over a large number of failures
- **MTTF** is more appropriate for **hardware**:
- We can record failure data for **n** failures:
 - Between times t_i & t_{i+1}
 - the average value is **MTTF = $(t_{i+1}-t_i)/(n-1)$**

Mean Time to Repair (MTTR)

Mean Time Between Failures (MTBF)

- Once failure occurs: **additional time** is required **to fix faults**
- **MTTR**: measures average time it takes to **fix faults**
- We can **combine MTTF and MTTR**:
 - to get an **availability** metric: **$MTBF=MTTF+MTTR$**
- **MTBF** of 100 hours would indicate
 - Once a failure occurs, the next failure is expected after 100 hours

Probability of Failure on Demand **(POFOD)**

- Measures the **likelihood of the system failing**:
- when a **service request** is made
- **POFOD** of 0.001 means:
 - **1 out of 1000** service requests may result in a failure

Availability

- Measures **how likely the system will be available** for use over a period of time:
 - considers the **number of failures** during a time interval
 - takes into account the **down time of a system**
- This metric is **important** for
 - **Telecommunication systems & operating systems**
 - where **repair and restart time are significant** and loss of service during that time is important



S/W Quality

Introduction

- **S/W Quality** & **Reliability** are ***closely inter-related***
 - Consistently producing accurate results, speed, user friendly
- **S/W Quality** can be – **Vague & Intangible**
- But needs to be **Defined** & **Measured** (using Quality Metrics)
- **Quality Management** is an essential part of **Proj. Mgmt**
- **Quality requirements** need to be ***clearly defined*** for a s/w
 - In ***SRS document***
- Needs to be **Followed** & **Monitored** throughout the s/w lifecycle

Product vs Process Metrics

Product metrics – Measuring quality of the completed product

Process metric – Measuring quality of the process used to create the product

- **Users** assess product quality by external attributes (like Size, Effort..)
 - Measured by **Product Metrics**
 - **Ex:** *LOC, Function Points* - for **size**, *Person Months* - for effort
- **Developers** assess product quality by internal attributes
 - Measured by **Process Metrics**
 - **Ex:** Review effectiveness, Avg. no of defects per hrs of inspection, Avg. defect correction time, Productivity ...

Process Capability Models : SEI Capability Maturity Model (CMM)

- **SEI-CMM** is a widely-accepted **quality certification** offered by **SEI** mainly to s/w organizations
- **SEI** - Software Engineering Institute (SEI), Mellon University, USA
- **Aim:** To improve quality of *s/w products* through **5 stages**

Process Capability Models : SEI Capability Maturity Model (CMM)

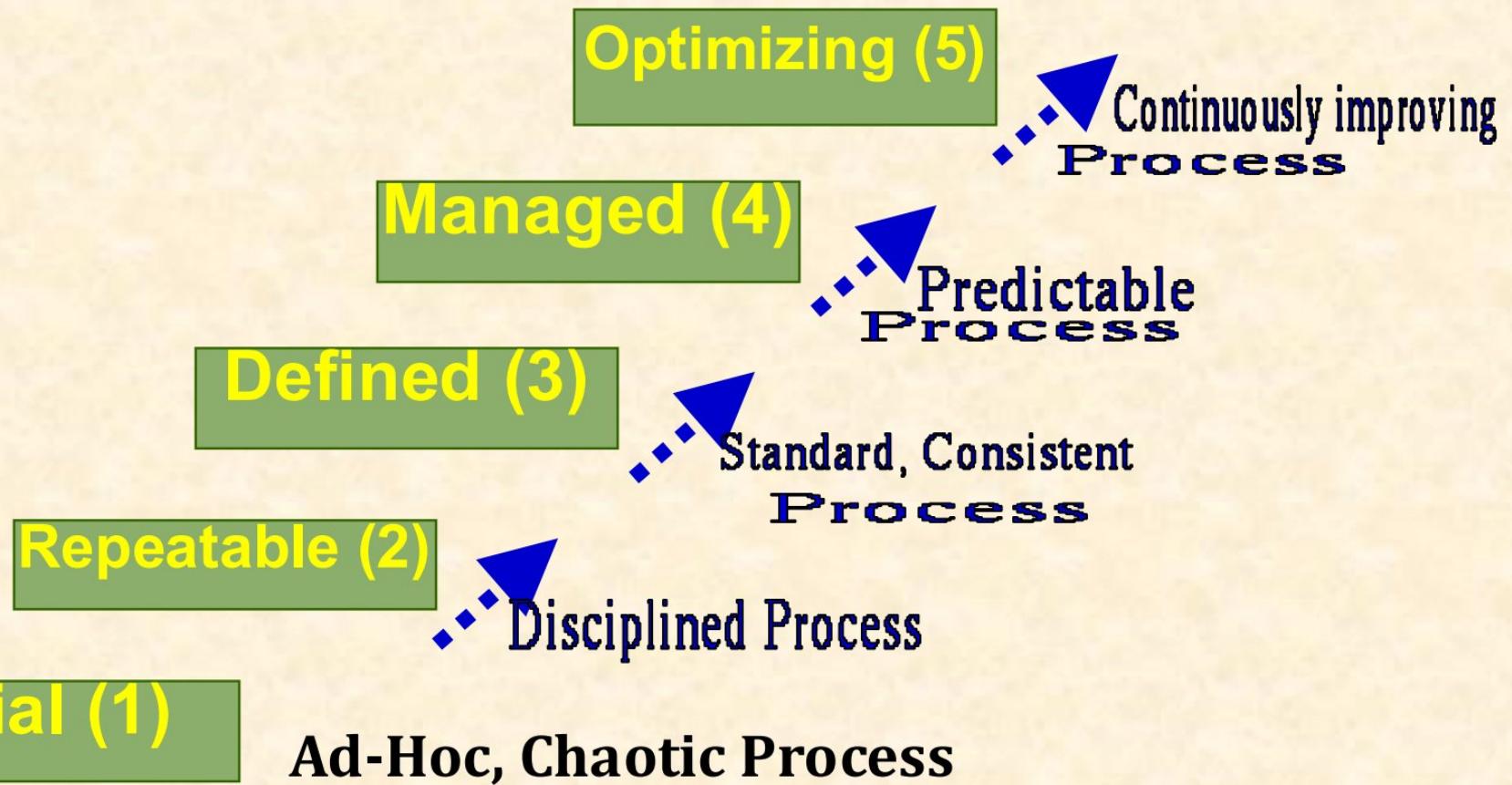
- SEI CMM helps organizations:
 - **To improve quality of s/w developed**
- Very popular & adopted by many organizations

Process Capability Models : SEI Capability Maturity Model (CMM)

- **CMM** is a model for evaluating the s/w process maturity of an Org. into one of five different levels

- Can be used in two ways:
 - For Capability evaluation of an Org.
 - Software process assessment

SEI -CMM



Level 1: (Initial)

- Org. operates **without** any **formalized process** or **project plan**
- Characterized by **ad hoc** and often **chaotic activities**
- **Different engineers** follow their **own process**
- The **success** of projects depend on **individual efforts & heroics**

Level 2: (Repeatable)

- Basic project management practices like
 - Tracking of *cost, schedule & functionality* are followed
- Size and cost estimation techniques like
 - *Function point analysis, COCOMO..* are used
- Development process is still ad hoc (neither formally defined & nor documented)
- Process may vary between different projects
- **Earlier success** on projects can be **repeated**
- Opportunity to repeat process exist

Level 3: (Defined)

- Management and development activities are :
Defined & documented
- Common org-wide understanding of activities, roles & responsibilities exist
- The processes are **defined**
- But, process and product qualities are **not measured**

Level 4: (Managed)

- **Quantitative quality goals** for products are **set**
 - Ex: Defects per Kloc, MTBF
- **Software processes** & **product qualities** are **measured**
- The **measured values** are **used to control the product quality**
- But, these are **not** used to **improve processes**

Level 5: (Optimizing)

- Statistics collected from process/product measurements are analyzed:
- **Continuous process improvement** is done based on the measurements
- Known types of defects are **prevented** from recurring
- Lessons learned from projects incorporated into the process
- **Best software engineering** practices, tools, methods & innovations are identified & promoted throughout the organization

Six Sigma

- Is a very **popular Quality standard**
- To achieve **SIX SIGMA**
 - A process must not produce more than **3.4 defects per million opportunities**
- **5 Sigma** -> A process must not produce more than **230 defects** per million opportunities
- **4 Sigma** -> A process must not produce more than **6210 defects** per million opportunities



Six Sigma

➤ Six sigma methodologies

- **DMAIC** (*Define, Measure, Analyze, Improve, Control*)
- **DMADV**: (*Define, Measure, Analyze, Design, Verify*)
- The methodologies are implemented by Green belt & Black belt workers supervised by Master black belt worker

Software Maintenance

Introduction

- **Software maintenance:**
 - Any modifications to a software product after it has been delivered to the customer
- Software products need **three types** of maintenance
 - 1. Corrective**
 - 2. Adaptive**
 - 3. Perfective**

➤ **Corrective:**

- To correct bugs in the system when reported by users
- Done to make the product defect free

➤ **Adaptive:**

- To port the product to new platforms, new OS
- To interface with new hardware or software
 - Ex: from Unix to Windows, from SQL Server to Oracle, from HP to Sun platform

➤ **Perfective:** (Enhancements)

- To support new features required by users
- To change some functionality due to customer demands

Major problems in maint.

1. **Unstructured code**
2. **Insufficient knowledge** of *Maintenance programmers*
 - Mostly *maint. team* is **different** from *dev. team*
3. **Documentation absent**, out of date or insufficient

Maint Nightmares

- Use of gos & lengthy procedures
- Poor and inconsistent variable naming
- Poor module structure
- Low cohesion & high coupling
- Deeply nested conditional statements
- Functions having side effects

How to do better maintenance?

- Clear Program understanding
- Reverse engineering
- Design recovery
- Reengineering
- Applying Maintenance process models

Software “Reverse Engineering”



- Important maintenance technique
- By analyzing a program code, recover from it -
 - the design and the requirements specification
- First carry out cosmetic changes to the code to improve:
 - Readability, structure, understandability of the code

Software “Reverse Engineering”

- **To extract the design:**
 - Need to understand the code fully
- **Automatic tools** can be used to derive:
 - Data flow & control flow diagrams from the code
- **Extract structure chart:**
 - module invocation sequence & data interchange among modules
- Extract **requirements specification** after extracting design

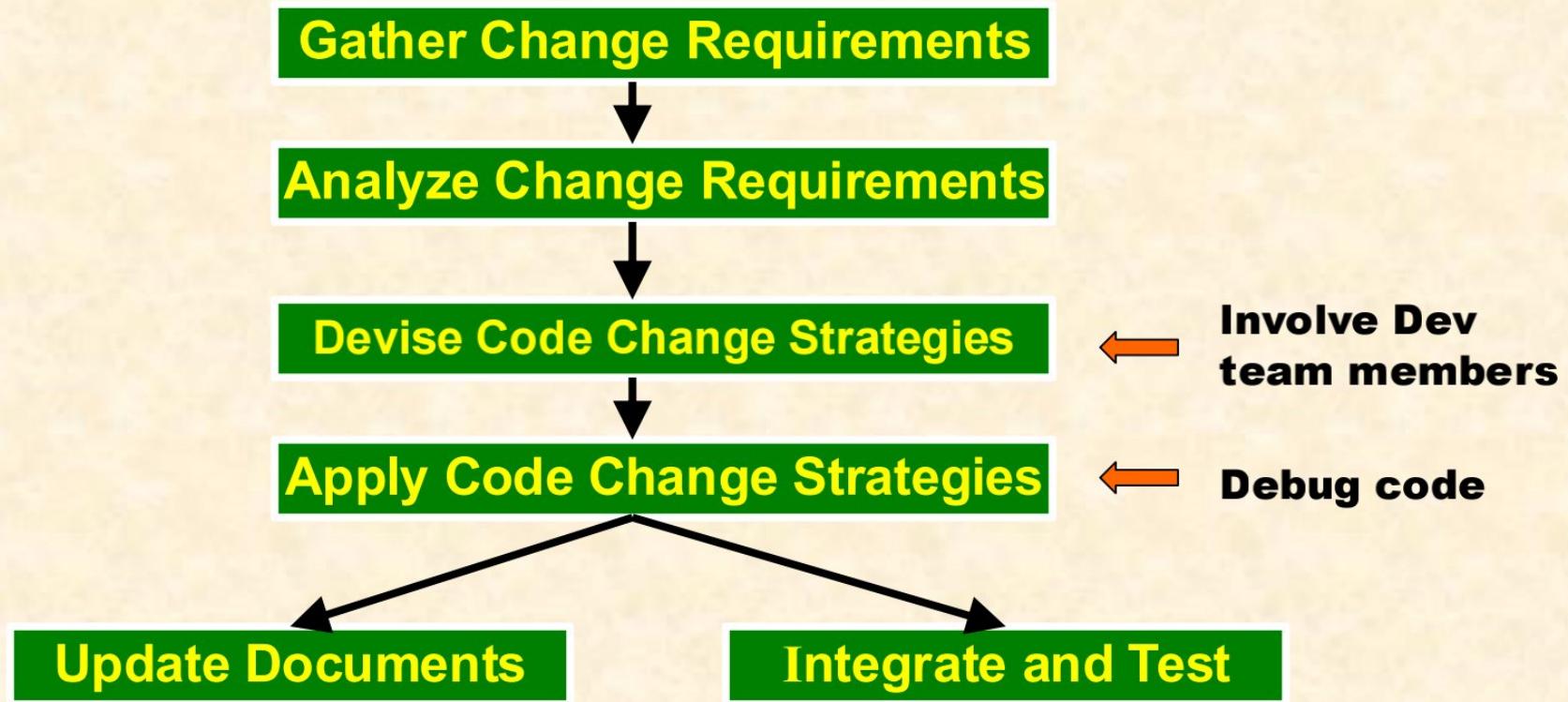
Software Maintenance “Process Models”

- **Maintenance activities** depend on :
- The extent of modifications required on the product
- Condition of the product:
 - how structured it is
 - how well documented it is

S/W Maintenance “Process Model - 1”

- Applied when the required changes are **small** and **simple**:
- The code can be **directly modified**
- Changes done in **all relevant documents**
- **Process 1** is preferable when amount of rework is **no more than 15%**

S/W Maintenance “Process Model - 1”



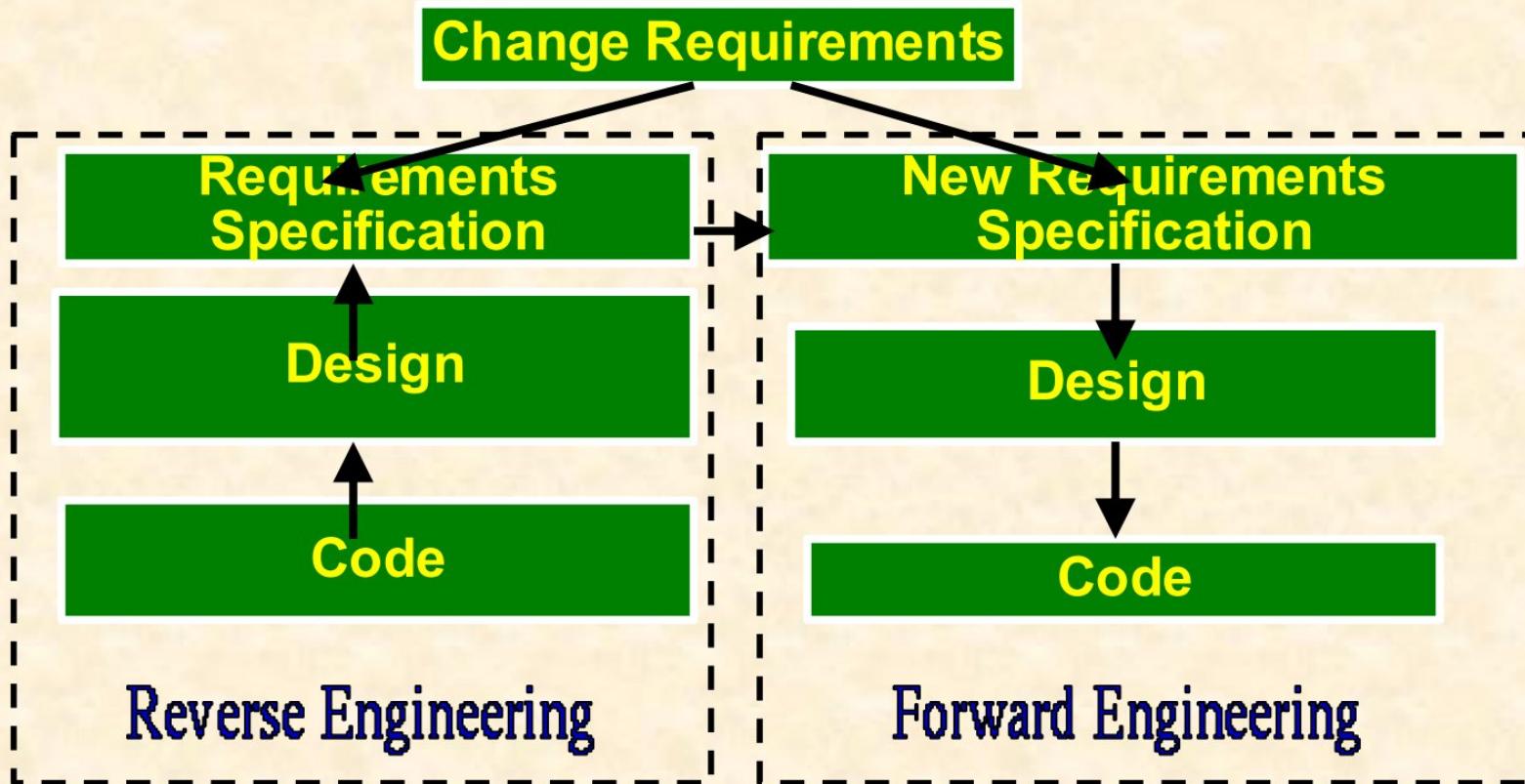
S/W Maintenance “Process Model - 2”

- Preferable when:
 - Project is *Complex*
 - Amount of rework is *significant*
 - Software has *poor structure*
- **Software re-engineering** needed:
 - a reverse engineering cycle followed by a forward engineering cycle
 - with as much reuse of existing code and documents

Software Re-engineering

1. Required for **aging** software products
2. During the **reverse engineering**, Extract the module spec from old code
3. Analyze module spec to produce the design
4. Analyze design to produce the original requirements spec
5. Apply change requests to the requirements spec:
 - Arrive at the new requirements spec
6. Then, **Forward engineering** is done to produce **new code**
7. During design, module spec and coding: Reuse is made

Process model for S/W reengineering



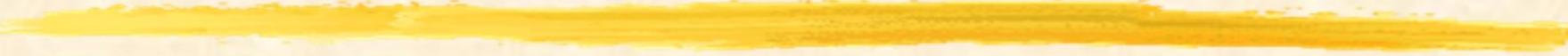
Software reengineering

Adv:

- Produces better design & required docs
- Is more efficient than the original product

Disadv:

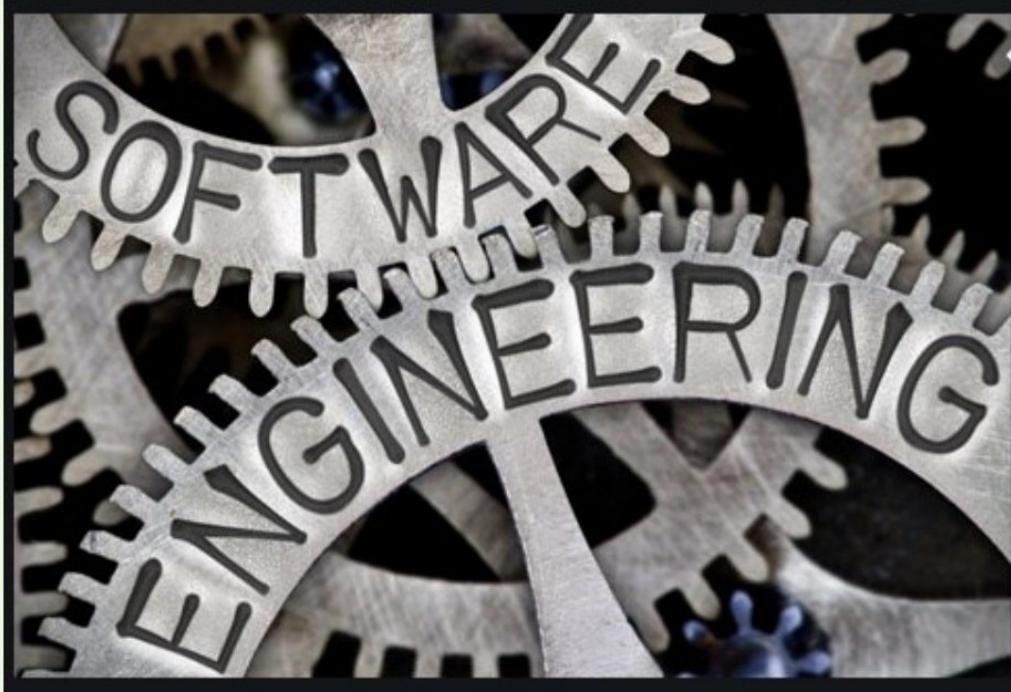
- More costly than the first approach
- Reengineering is preferable when:
 - Amount of rework is high
 - Product exhibits high failure rate
 - Product is difficult to understand



THE END

Software Engineering – Project Management

Module-2



Faculty :
Dr. Suchismita Rout
Associate Professor

Module2 Contents

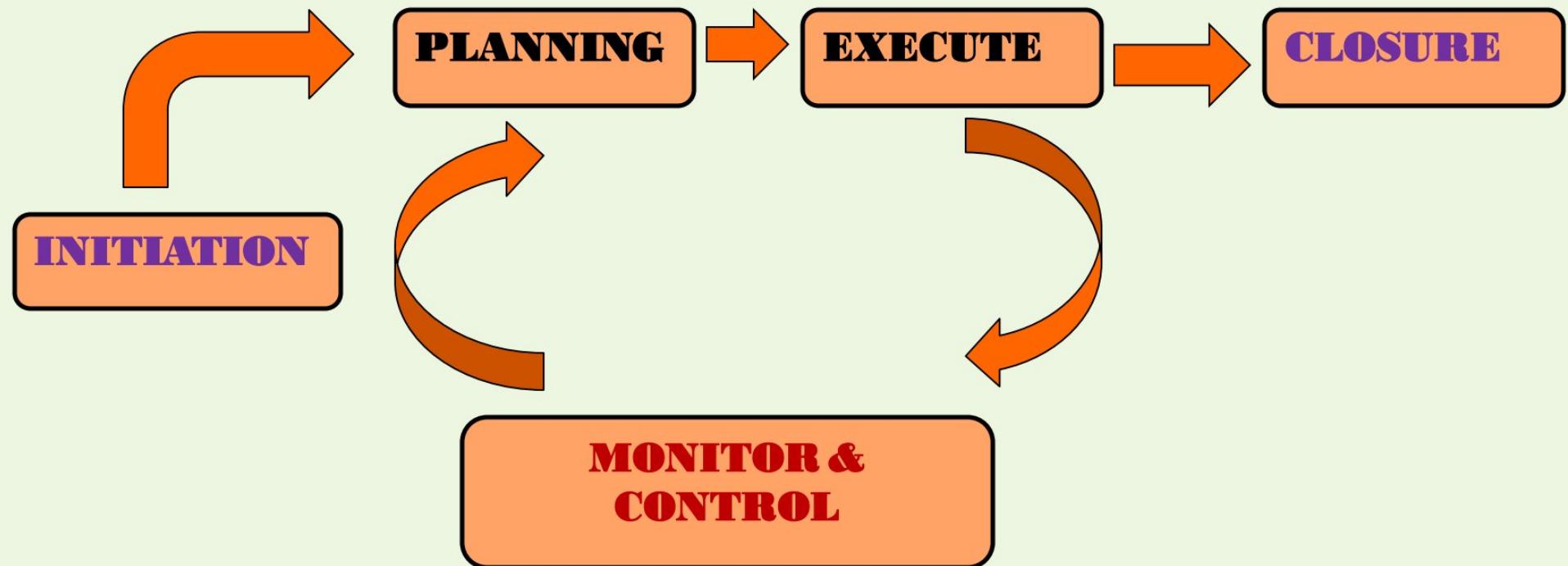
- **Software Project Management - topics**

- **Project Planning**
- **Metrics for Project Size Estimation**
- **Scheduling, Staffing**
- **Project Estimation Techniques**
- **COCOMO model**
- **Halstead's Software Science**
- **Risk Management**

Software Project Management



Project Management life cycle



- **Monitoring** : Compare Planned Progress with Actual Progress
- **Controlling** : Taking corrective action if required

Project planning



Project Planning in 10 Steps

0. Select Project
1. Identify project scope & objectives
2. Identify project infrastructure
3. Analyze project characteristics & Select SDLC model
4. Identify project deliverables & activities
5. Estimate effort for each activity
6. Identify activity risks
7. Allocate Resources
8. Review/Publish plan
9. & 10. Execute Plan & Lower level planning

Step wise planning activities

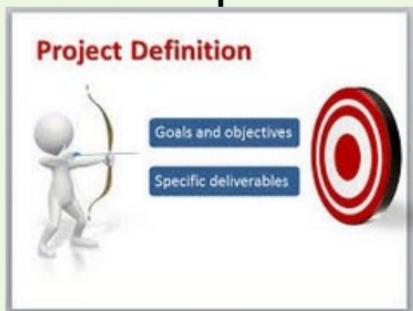


Step	Activities
0	Select Project (based on strategic evaluation)
1	Identify project scope & objectives (Have all parties agree to the objectives & commit to the success) Id. <u>Objectives</u> of the project Establish <u>project authority</u> (Project Manager) Id. all <u>stakeholders</u> (List stakeholders & their needs) <u>Modify objectives</u> based on <u>stakeholder analysis</u> Establish <u>communication methods</u> (Whom & How)
2	Identify project infrastructure (Existing inf. h/w, s/w, logistics) Establish <u>relationship</u> between <u>project</u> & <u>strategic planning</u> (Decide priority of the project) Id. <u>Installation standards & procedures</u> Id. <u>Project team organization</u>

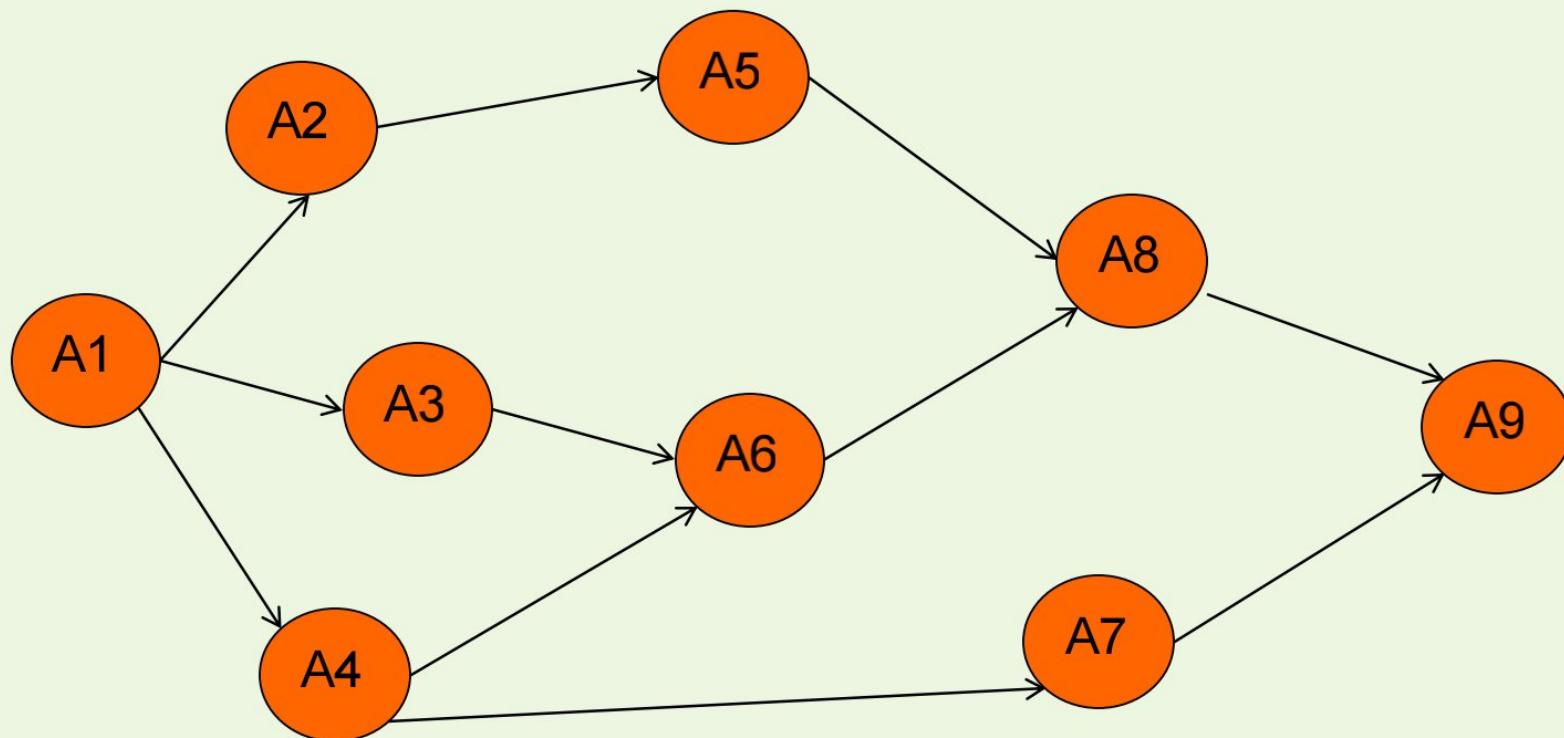
Step wise planning activities



Step	Activities
3	<p>Analyze project characteristics (for selecting appropriate methods)</p> <p>Distinguish the project (<u>Objective driven</u> or <u>Product driven</u>)</p> <p>Analyze <u>other characteristics</u> & . <u>High level risks</u></p> <p>Select <u>lifecycle approach</u></p> <p>Review <u>overall resource estimates</u></p>
4	<p>Identify project deliverables & activities</p> <p>Id. <u>deliverables</u> (SRS, Set of components, test plans/results)</p> <p>Document generic <u>product flows</u> (SRS → Design Module → Develop Module → Test module, WBS)</p> <p>Recognize <u>product components</u></p> <p>Produce <u>activity network</u></p> <p>Take into account(modify) <u>stages</u> & <u>check points</u></p>



Activity Network



Step wise planning activities



Step	Activities
5	<p><u>Estimate effort for each activity</u></p> <p>Carry out <u>bottom-up</u> estimates</p> <p>Revise Plan to create <u>controllable activities</u> (long activities to be <u>broken down</u> into smaller subtasks)</p> 
6	<p><u>Identify activity risks</u></p> <p><u>Id. & quantify</u> activity based risks (Id. risks, estimate - <u>probability & impact</u>)</p> <p>Plan for <u>countering the risks</u></p> <p>Adjust overall plan to account for risks</p> 
7	<p><u>Allocate resources</u></p> <p><u>Id & allocate</u> <u>resources</u></p> <p>Revise Plan/estimates based on <u>resource constraints</u></p> 

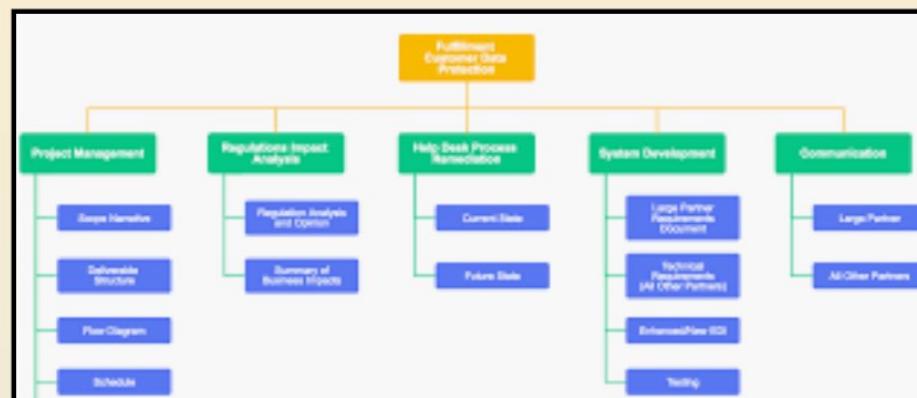
Step wise planning activities

Step	Activities
8	<p><u>Review/Publish plan</u></p>  <p>Review <u>quality aspects</u> of the Project Plan <u>Document Plans & obtain agreement</u></p>
9 & 10	<p><u>Execute Plan & Lower level planning</u></p> <p><u>Implement</u> the documented plan <u>Plan in detail</u> as you get more info.</p> 

- ✓ Project Management is a Iterative process & Project Plan is updated many times during a project

S/W Project Management Plan (SPMP)

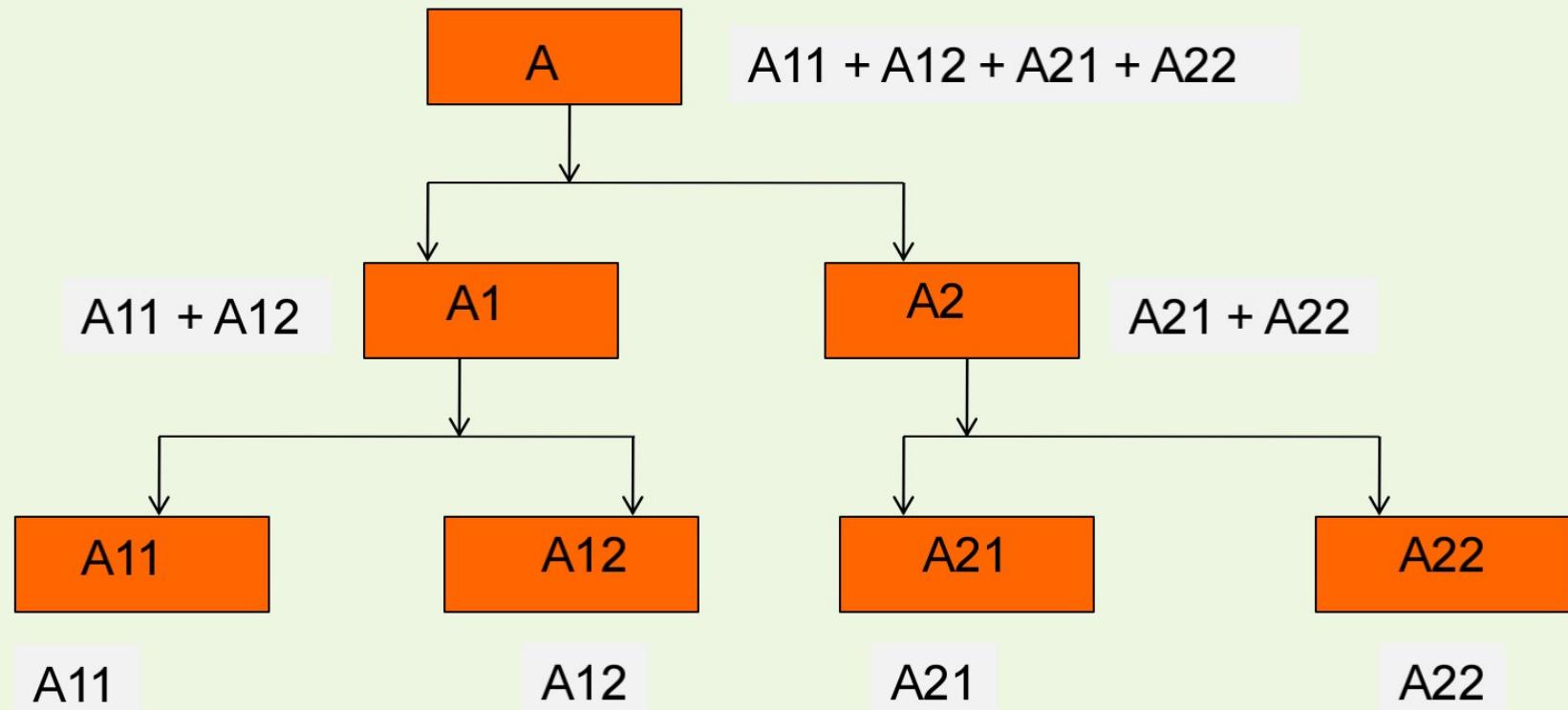
- **WBS (Work Breakdown Structure)**
 - Decomposition of project into smaller manageable parts – WBS Units



- **WBS Units**
 - Each WBS unit must have a *Clear Outcome, Accountability, is Trackable, has Well defined interface*

- “**WBS**” is created:
 - To Divide & Conquer & to estimate accurately
- **How WBS is done**
 - Id. Logical components & minimize dependency between WBS Units
- This is followed by “**Bottom-Up**” estimation approach

Bottom-up Estimate



■ SPMP includes:

- Size estimates (*KLOC, FPs...*)
- Effort estimates
- Cost estimates
- Internal & External Milestones
- Resource & Work allocation
- Project schedule
- Communication & Status Reporting plan
- Risk management plan

Project Schedule Estimation

- “Calender Time” to deliver
- Provides answer to “When can we deliver what”
- **Schedule estimation** is done taking into account :

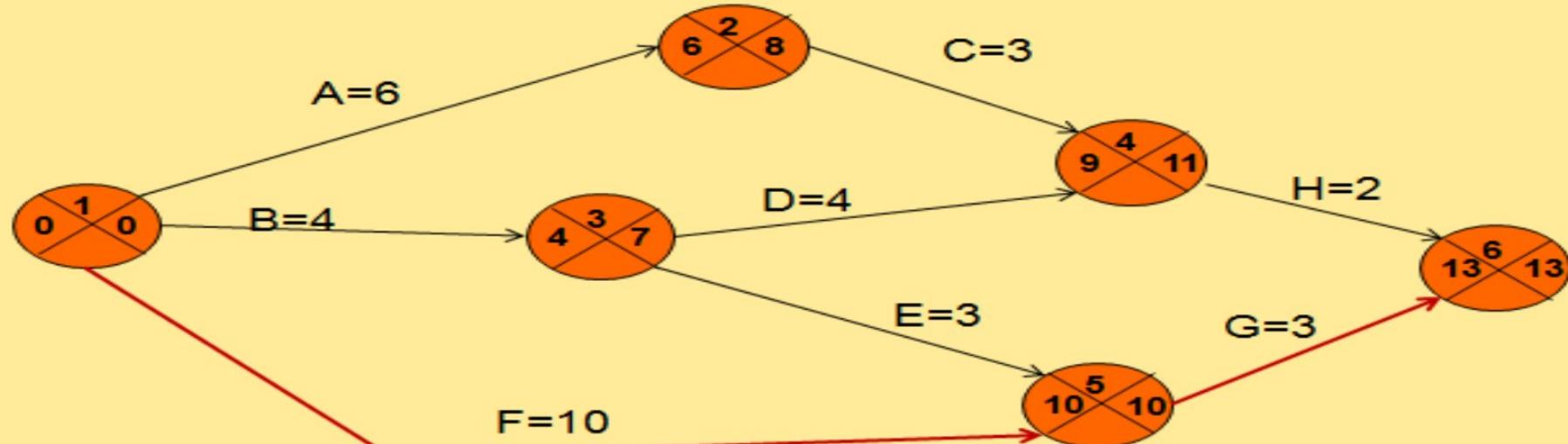
- Deadline limits
- Resources available
- H/W Constraints
- Internal & External dependencies

Vaccine	Birth	1 mo	2 mo	4 mo	6 mo	9 mo	12 mo	15 mo	18 mo	19-23	2-3 yrs	4-6 yrs	7-10 yrs	11-12 yrs	13-15 yrs	16 yrs	17-18 yrs
Hepatitis B (HepB)	1 st dose	2 nd dose					2 nd dose										
Rotavirus (RV) RVT (2-dose series) RVS (3-dose series)			1 st dose	2 nd dose	See Notes												
Diphtheria, tetanus, &acellular pertussis (DtaP; >7 yrs)			1 st dose	2 nd dose	3 rd dose				4 th dose			5 th dose					
Meningococcal influenza type b (MenB)			1 st dose	2 nd dose	See Notes				4 th dose	5 th dose	6 th dose						
Pneumococcal conjugate (PCV13)			1 st dose	2 nd dose	3 rd dose				4 th dose								
Inactivated polio virus (IPV; <18 yrs)			1 st dose	2 nd dose		3 rd dose					4 th dose						
Influenza (IV)												Annual vaccination 1 or 2 doses					
Influenza (LAIV)													Annual vaccination 1 dose only				
Measles, mumps, rubella (MMR)						See Notes			1 st dose			2 nd dose					
Varicella (VAR)									1 st dose			2 nd dose					
Hepatitis A (HepA)						See Notes				2-dose series, See Notes							

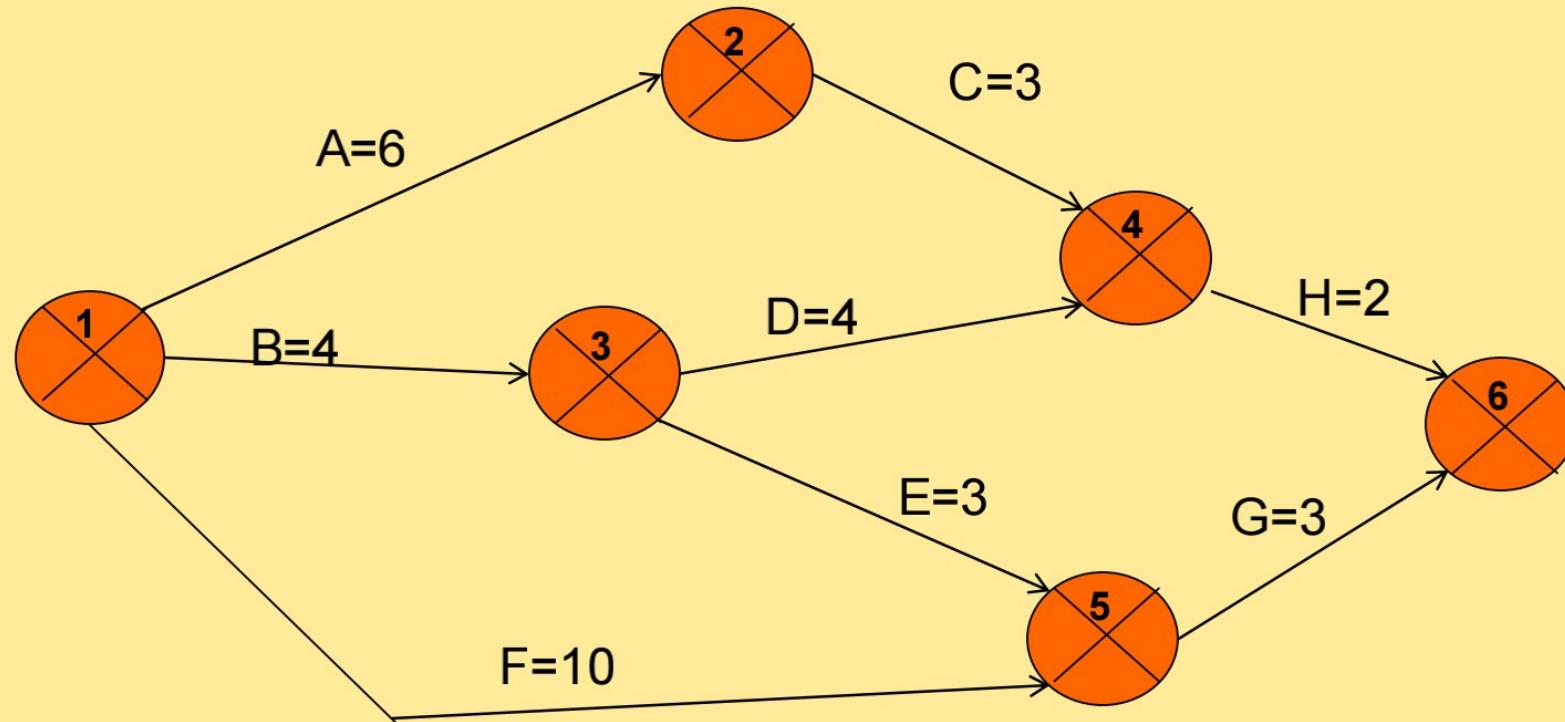
- Done by constructing “Activity Network model” & using **CPM (Critical Path Method)**

Activity network model & CPM (critical Path method)

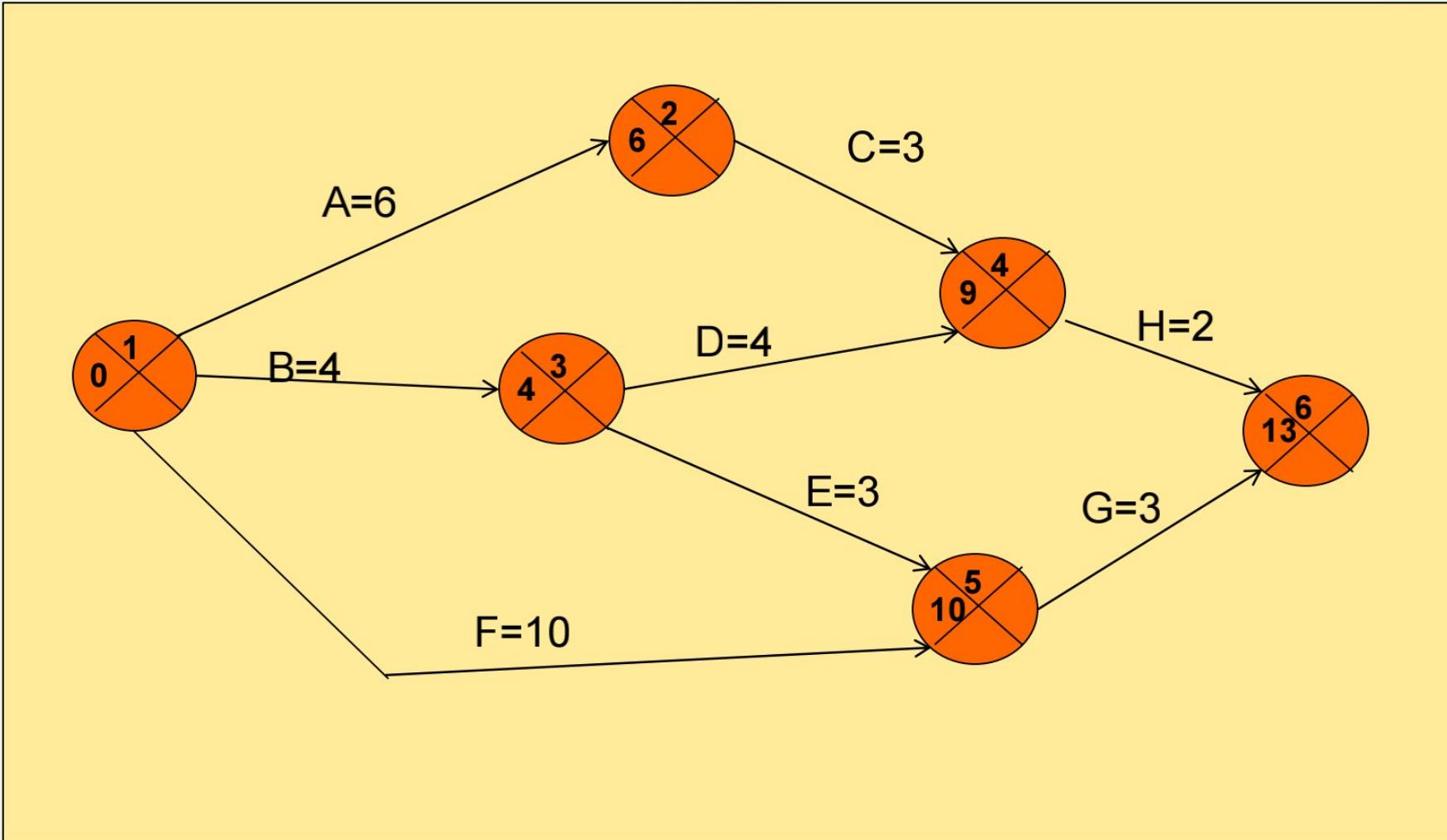
- Activities are “nodes”
- Durations are “edges”
- Direction of edges show “precedence / dependency” between activities



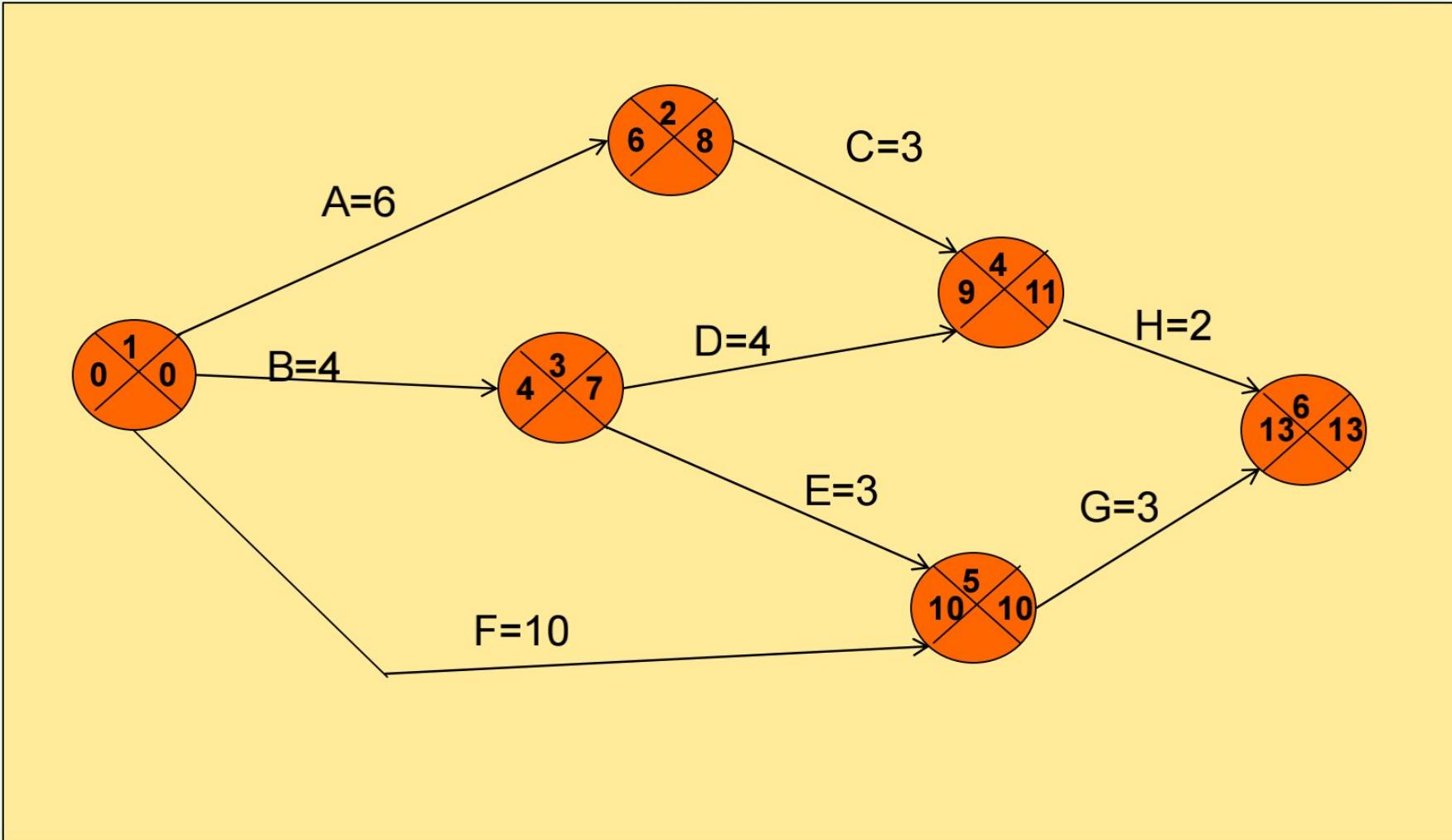
Scheduling of Activities : Activity network model (CPM method)



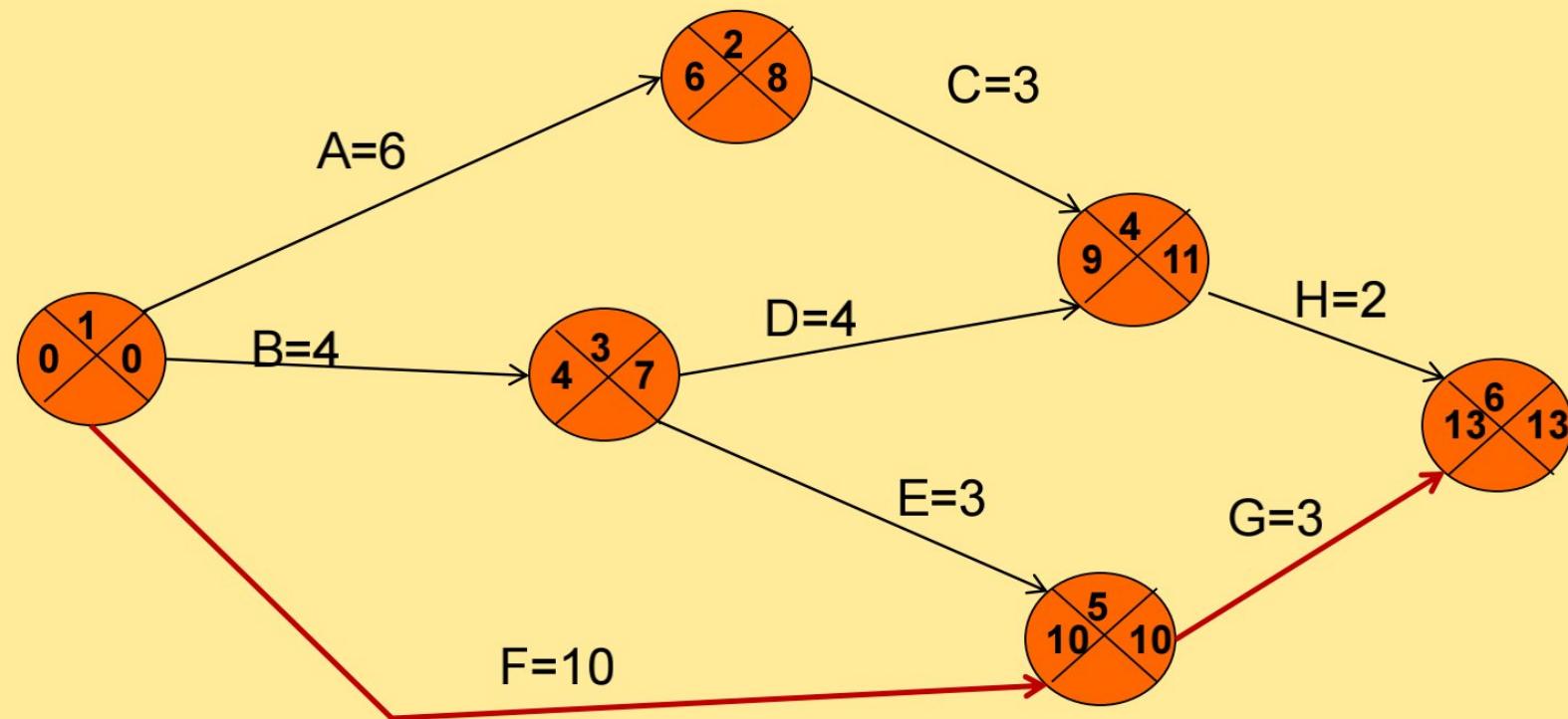
CPM n/w after forward pass



CPM n/w after backward pass



Critical Path



Critical Path

- **Difference** (also called Slack or Float)

latest start/completion date – earliest start/completion date

- **Events/Tasks** with 0 slack are Critical events

- The path joining critical events is called “Critical Path”

Critical Path

- Need to pay more attention to events on critical path
- To Avoid **delays**
- To shorten a project, the critical path must be shortened
- By adding more resources/overtime

Project Estimation & cocomo



Objectives

- To understand different methods of "Estimating" projects
- COCOMO approach

Introduction

A “Successful project” is the one that is

- Delivered **on-time**
- Within **budget**
- Is of **required quality**

Realistic estimates are **critical** to **achieve the above objectives**

Estimating s/w projects

Estimation is done during below stages :

- **Strategic Planning** (to prioritize the project)
- **Feasibility Study**
- **Requirement Specification**
- **Evaluation of supplier proposal**
- **Project Planning**

Basis of s/w estimating

Estimation is done based on below factors:

- Based on Historical data
 - Env. factors, S/w & Tools used, staff experience have to considered
- Based on Measure of work
 - Based on work content (like SLOC, KLOC)
 - Difficult to apply for app. building tools
- Based on Complexity
 - Two programs with same SLOC may take different times based on complexity (complexity needs to be taken into account)

S/W Effort estimating techniques

➤ Algorithmic methods

- Uses characteristics of target system (Ex: size of system) & the implementation environment to predict effort

➤ Empirical estimation techniques

- Expert Judgment (Take expert advise to estimate)
- Delphi Technique
- Analogy (Estimate based on similar completed projects)
- Parkinson (Estimate Based on staff effort available)
- Price to win (Estimate to win the contract)
- Top-Down (Overall effort is distributed to tasks)
 - Ex: Building a house based on sq. ft rate
- Bottom-up (Effort of individual tasks are summed up)

Bottom-up VS Top-down estimation

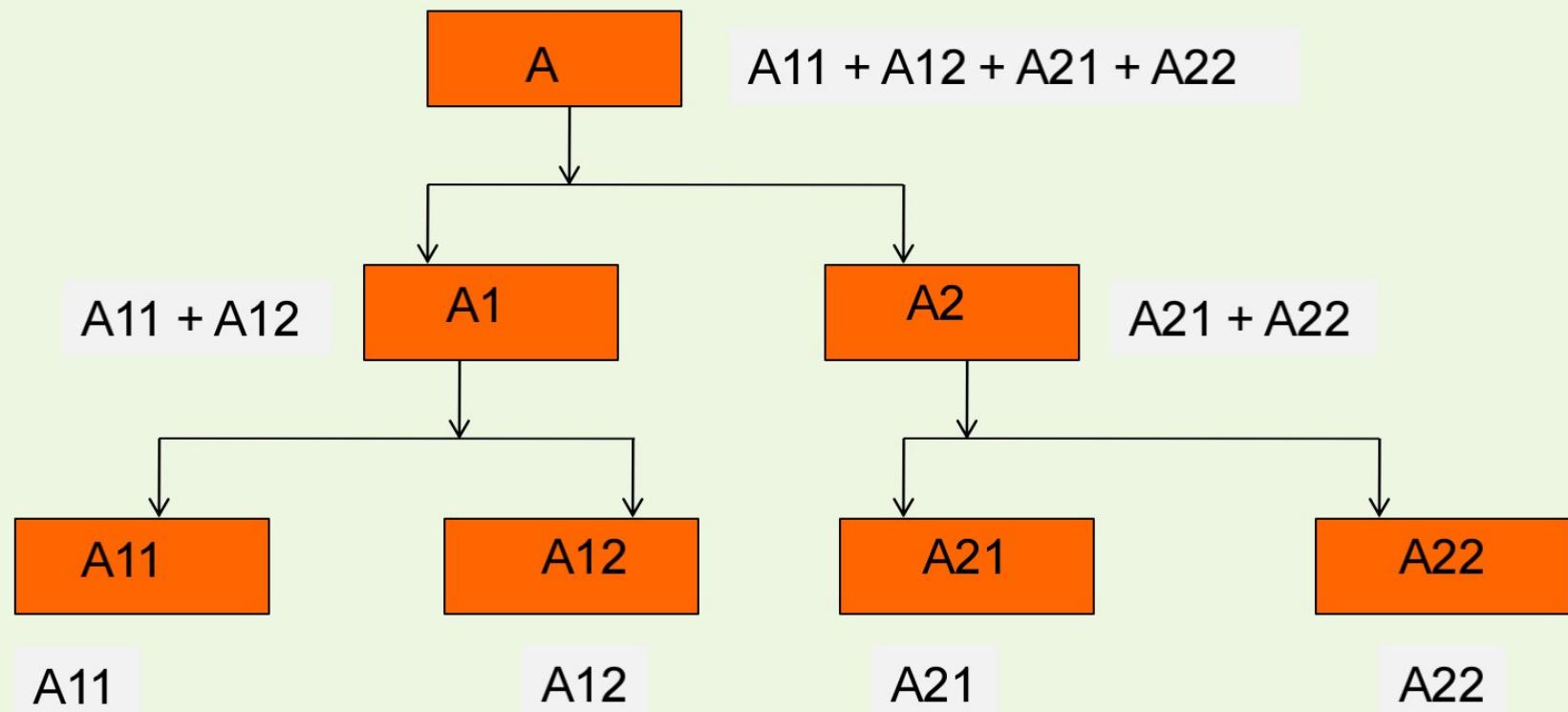
Bottom-up estimation

- Breaks the project up to tasks level
- Called WBS (*work breakdown structure*)
- Tasks can be executed by a single resource
- Estimate effort for each task
- Sum up to find total estimate

Top-down estimation

- Parametric model used to estimate the effort based on characteristics of the system & distribute to task level
- Building a house (estimation based on sq. feet of area)
 - Estimation based on a characteristics of the system
 - Effort = System size X Productivity
Ex: 50 (Kloc) X 40 (man-days per Kloc) = 2000 man-days

Bottom-up Estimate



Expert Judgement & Estimation by Analogy

✓ Expert Judgment

- People very knowledgeable about the s/w & type of project can provide the estimate based on experience & expertise

✓ Estimation by Analogy

- Case-based reasoning
- Estimation based on similar completed projects
- Adjustments made based on the differences

Delphi Method of estimation

- The Delphi method is a process used to arrive at a group consensus (agreement) on estimation given a panel of experts
- Experts respond to questionnaires
- Their responses are aggregated and shared
 - In case of differences, further rounds of discussions take place to arrive at an agreement

COCOMO Model

- ✓ **cocomo** (**CO**nstructive **CO**st **MO**del) proposed by Boehm
- ✓ Divides **Software Projects / Products** into **Three classes** based on **development complexities**:
 1. **Organic**
 2. **Semidetached**
 3. **Embedded**

Elaboration of Product classes

1. Organic:

- ✓ Relatively small project
- ✓ Team Experienced with similar projects
- ✓ Well-understood applications

2. Semidetached:

- ✓ **Project team** consists of a mixture of experienced & inexperienced staff
- ✓ Limited experience on similar systems

3. Embedded:

- ✓ **Complex** system
- ✓ The software is strongly **coupled to complex hardware**
- ✓ Real-time systems

COCOMO Model (Cont.)

- ✓ For each of the three product categories:
 - ✓ Boehm provides equations to predict:
 - ✓ Effort in person-months
 - ✓ Project duration in months
 - ✓ One **PM** is included in the effort
 - ✓ Takes into account productivity loss due to holidays, weekly offs...

COCOMO Model (cont.)

- ✓ According to Boehm, Software cost estimation is done through three stages:

- 1. Basic COCOMO**
- 2. Intermediate COCOMO**
- 3. Complete COCOMO**

(1) Basic COCOMO Model

- ✓ Gives only an approximate estimation:

- ✓ **Effort = $a_1 * (KLOC)^{a_2}$ PM**
 - ✓ **Tdev = $b_1 * (Effort)^{b_2}$ Months**

- ✓ **KLOC** – Estimated code in kloc
 - ✓ **a₁,a₂,b₁,b₂** - Constants for different categories of software products (Organic, Semi-detached, Emb.)
 - ✓ **Tdev** - Estimated time duration to develop the software in months

Development Effort & Time Estimation

➤ Values of a₁,a₂ & b₁,b₂

✓ Organic :

- ✓ Effort = $2.4 * (\text{KLOC})^{1.05}$ PM
- ✓ $T_{\text{dev}} = 2.5 * (\text{Effort})^{0.38}$ Months

✓ Semi-detached:

- ✓ Effort = $3.0 * (\text{KLOC})^{1.12}$ PM
- ✓ $T_{\text{dev}} = 2.5 * (\text{Effort})^{0.35}$ Months

✓ Embedded:

- ✓ Effort = $3.6 * (\text{KLOC})^{1.20}$ PM
- ✓ $T_{\text{dev}} = 2.5 * (\text{Effort})^{0.32}$ Months

(2) Intermediate COCOMO

✓ **Basic COCOMO** model assumes

- ✓ Effort and development time depend on **product size alone**

✓ **Several other parameters** affect effort and development time:

EX:

- ✓ **Reliability of requirements**
- ✓ **Availability modern tools**
- ✓ **Size of data to be handled ...**

Intermediate COCOMO

✓ Intermediate COCOMO model:

- ✓ Refines the initial estimate obtained by the basic COCOMO
- ✓ by using a set of 15 cost drivers (multipliers)
- ✓ Multiply cost driver values with the estimate obtained using the basic COCOMO

✓ **Effort = [Effort Basic_cocomo] * [CD₁ * CD₂ ... CD₁₅] PM**

Cost Drivers

Cost Drivers	Ratings				
	Very Low	Low	Nominal	High	Very High
Product attributes					
Required software reliability	0.75	0.88	1.00	1.15	1.40
Size of application database		0.94	1.00	1.08	1.16
Complexity of the product	0.70	0.85	1.00	1.15	1.30
Hardware attributes					
Run-time performance constraints			1.00	1.11	1.30
Memory constraints			1.00	1.06	1.21
Volatility of the virtual machine environment		0.87	1.00	1.15	1.30
Required turnabout time		0.87	1.00	1.07	1.15
Personnel attributes					
Analyst capability	1.46	1.19	1.00	0.86	0.71
Applications experience	1.29	1.13	1.00	0.91	0.82
Software engineer capability	1.42	1.17	1.00	0.86	0.70
Virtual machine experience	1.21	1.10	1.00	0.90	
Programming language experience	1.14	1.07	1.00	0.95	
Project attributes					
Application of software engineering methods	1.24	1.10	1.00	0.91	0.82
Use of software tools	1.24	1.10	1.00	0.91	0.83
Required development schedule	1.23	1.08	1.00	1.04	1.10

Shortcoming of basic and intermediate COCOMO models

In Both models:

- ✓ Consider a software product as a **single homogeneous entity**
- ✓ However, most large systems are made up of several smaller sub-systems
 - ✓ Some sub-systems may be organic type, some may be semi-detached or embedded

(3) Complete COCOMO

- ✓ Each **sub-system** may be of **different types – organic, semi-detached or embedded**
- ✓ **Cost of each sub-system** is **estimated separately**
- ✓ Costs of the sub-systems are **added** to obtain **total cost**
- ✓ This **Reduces** the **margin of error** in the final estimate

Example

- ✓ A **Management Information System (MIS)** for an organization with many branches has:
 1. **Database part** (semi-detached)
 2. **Graphical User Interface (GUI) part** (organic)
 3. **Communication part** (embedded)
- ✓ Costs of the components are estimated separately:
 - ✓ Summed up to give the overall cost of the system

Halstead's Software Science

- **Halstead complexity measures** are software metrics introduced by *M.H Halstead* in *1977*
- Consists of a number of metrics to measure complexity of code based on it's algorithm
- These metrics are computed statistically from the **code**
- Halstead's goal was to identify measurable properties of software, and the relations between them

Metrics Calculation steps

For a given problem, Let:

- η_1 = the number of distinct operators
- η_2 = the number of distinct operands
- N_1 = the total number of operators
- N_2 = the total number of operands

From these numbers, several measures can be calculated:

- Program vocabulary: $\eta = \eta_1 + \eta_2$
- Program length: $N = N_1 + N_2$
- Calculated estimated program length: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
- Volume: $V = N \times \log_2 \eta$
- Difficulty : $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
- Effort: $E = D \times V$

Metrics Calculation steps

The difficulty measure is related to the difficulty of the program to write or understand

The effort measure translates into actual coding time using the following relation,

- Time required to program: $T = \frac{E}{18}$ seconds

Halstead's delivered bugs (B) is an estimate for the number of errors in the implementation.

- Number of delivered bugs : $B = \frac{E^{\frac{2}{3}}}{3000}$ or, more recently, $B = \frac{V}{3000}$ is accepted

Example

```
main()
{
    int a, b, c, avg;
    scanf("%d %d %d", &a, &b, &c);
    avg = (a + b + c) / 3;
    printf("avg = %d", avg);
}
```

The unique operators are: main, (), {}, int, scanf, &, =, +, /, printf, , , ;

The unique operands are: a, b, c, avg, "%d %d %d", 3, "avg = %d"

- $\eta_1 = 12, \eta_2 = 7, \eta = 19$
- $N_1 = 27, N_2 = 15, N = 42$
- Calculated Estimated Program Length: $\hat{N} = 12 \times \log_2 12 + 7 \times \log_2 7 = 62.67$
- Volume: $V = 42 \times \log_2 19 = 178.4$
- Difficulty: $D = \frac{12}{2} \times \frac{15}{7} = 12.85$
- Effort: $E = 12.85 \times 178.4 = 2292.44$
- Time required to program: $T = \frac{2292.44}{18} = 127.357$ seconds
- Number of delivered bugs: $B = \frac{2292.44^{\frac{2}{3}}}{3000} = 0.05$

Risk Management



Identify Risks



Objective:

- To identify the **Risks** that may cause a project to complete late or go **over budget**

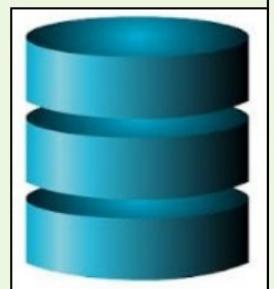
Identify 3 types of risks :

1. Risks caused by **Estimation difficulties**
2. Risks caused by **Assumptions** made during **planning**
3. Risks caused by **unforeseen events**

1. Estimation Difficulties



- ✓ Some tasks are harder to estimate than others:
 - Due to lack of experience of the estimator OR
 - Due to nature of the task
- **Ex:**
 - Debugging is harder to estimate than creating user manuals
- ✓ Estimation can be improved by analyzing historical data



2. Assumptions

- If assumptions made during planning are invalid,
 - Project will be at risk

Ex: Wrong selection of life cycle model or design approach

3. Unforeseen Events

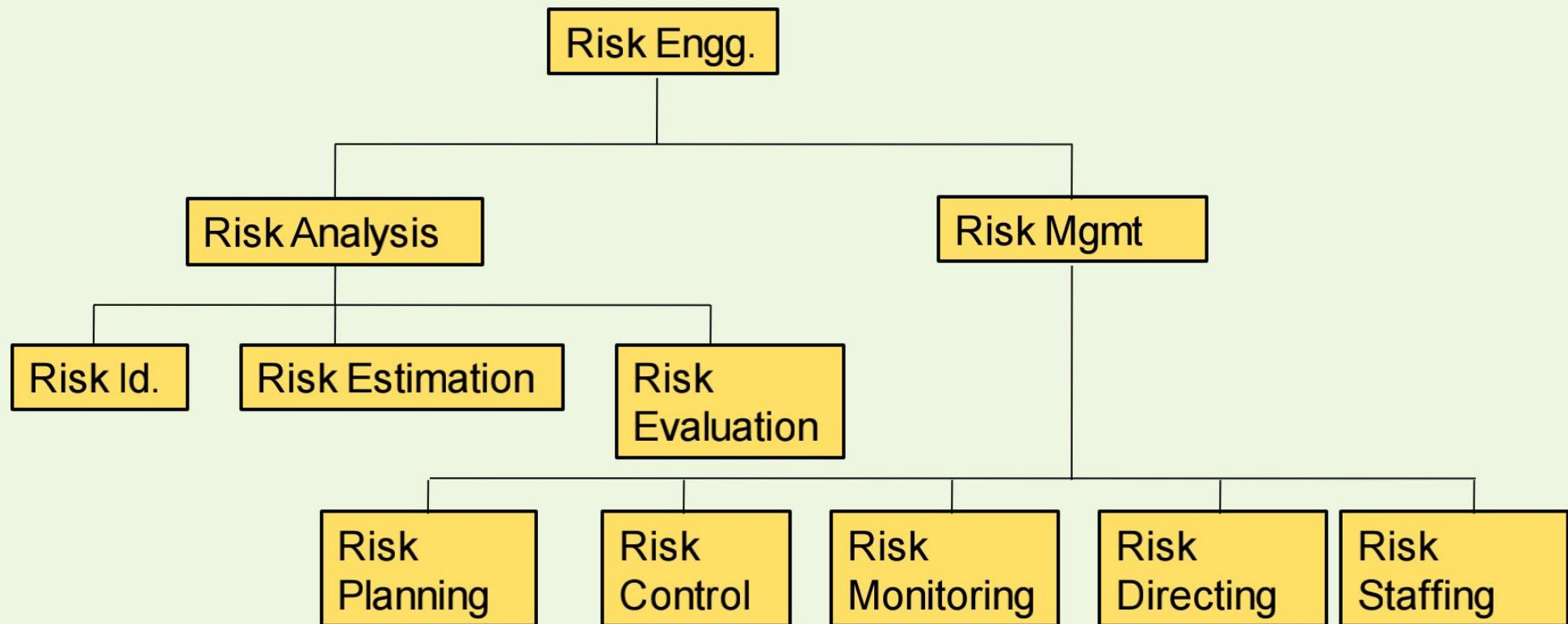
- There may be some rare unforeseen event happening
 - **Ex:** Tech. used becomes unstable / unsupported,
Drastic rule changes

Risk Engineering structure



Objectives:

- To avoid or minimize effects of project risks



Managing Risks



Risk Analysis:

- ✓ **Risk Identification** : List all the risks affecting the project
- ✓ **Risk Estimation** : Assessing the likelihood & impact of each risk
- ✓ **Risk Evaluation** : Ranking the risks

Risk Management:

- ✓ **Risk Planning** : Making 'contingency plans' & adding tasks to project.
May need a risk manager
- ✓ **Risk Control** : Minimizing & Reacting to risks throughout the project
- ✓ **Risk Monitoring** : On-going task. Risk likelihoods & impacts may change
- ✓ **Risk Staffing** : Allocate Staff to manage risks in day-to-day basis

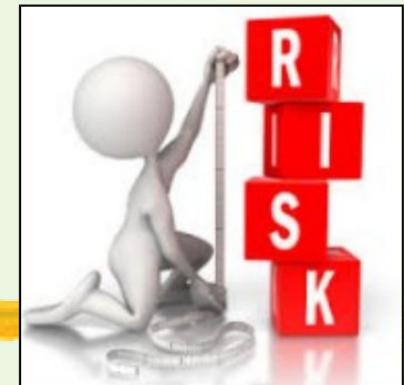
Risks Identification



- ✓ If a risk affects activities on 'critical path' it will cause delay
- ✓ Use checklist to Identify Risks (2 types)
 - ✓ Generic risks
 - ✓ **Ex:** *Misunderstanding of requirements, Key person ill*
 - ✓ Specific risks (for a given project) – More difficult to id.
 - ✓ **Ex:** *Unknown techs used in the project, Safety-critical*

1. Risk Analysis:

1.2. Risks Estimation



- Determine “**Likelihood**” & “**impact**” (in monetary terms) of risks
- Calculate ‘**Risk Value**’ (Expected cost)

Risk Exposure = Risk likelihood X Risk Impact

- Use “**Risk Exposure**” to **prioritize** or **rank** risks
- Risk scoring can be expressed as **(H, M, L)** or in scale of **1-10**

Risk Matrix



A **risk matrix** is a matrix that is used during risk assessment to define the level of risk by considering :

- The “**category of probability or likelihood**” against the “**category of consequence severity or impact**”
- This is a simple mechanism to increase visibility of risks & assist management decision making

			Impact		
		Low	Medium	High	
		Low	Medium	High	
Probability	High	Low	Medium	High	
	Medium	Low	Medium	Medium	
	Low	Low	Low	Low	

Strategies to Reduce Risks



5 Strategies are there :

- **Risk Prevention** (*Scheduling critical meetings early*)
- **Likelihood Reduction** (*Prototyping reduces chance of freq. reqt. changes*)
- **Risk Avoidance** (*Reduce functionality, Increase duration*)
- **Risk Transfer** (*Contracting, Insurance*)
- **Contingency Planning** (*Plan to minimize impact of remaining risks*)

1. Risk Analysis:

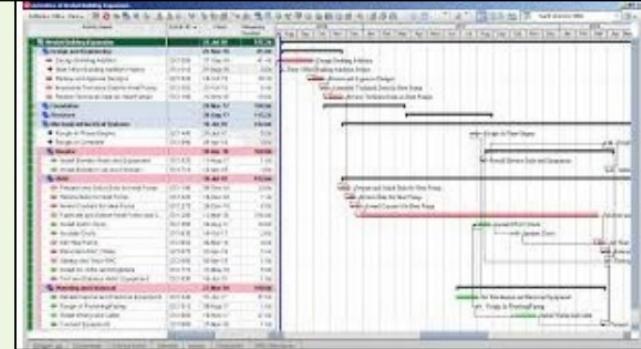
1.3. Risks Evaluation – on schedule

- All risks can not be eliminated
- Some will delay the project
- Id. such risks & their impact on duration
- Can use 2 methods:

1. PERT

- Uses 3 estimates : Most likely (m), Optimistic (a) & Pessimistic time (b)
- PERT combines these 3 estimates into a single estimate

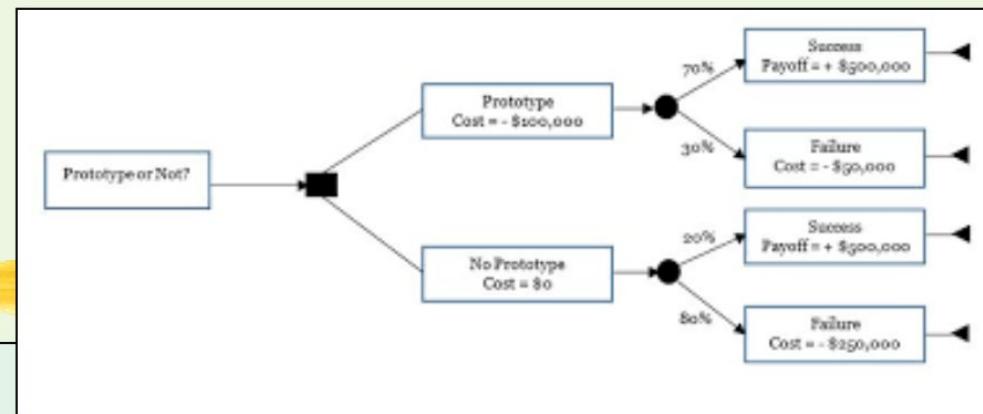
$$t_e = \frac{(a + 4m + b)}{6}$$



2. Using expected durations

- The event dates are not the 'earliest possible dates'
- But 'expected dates'

Risk Evaluation



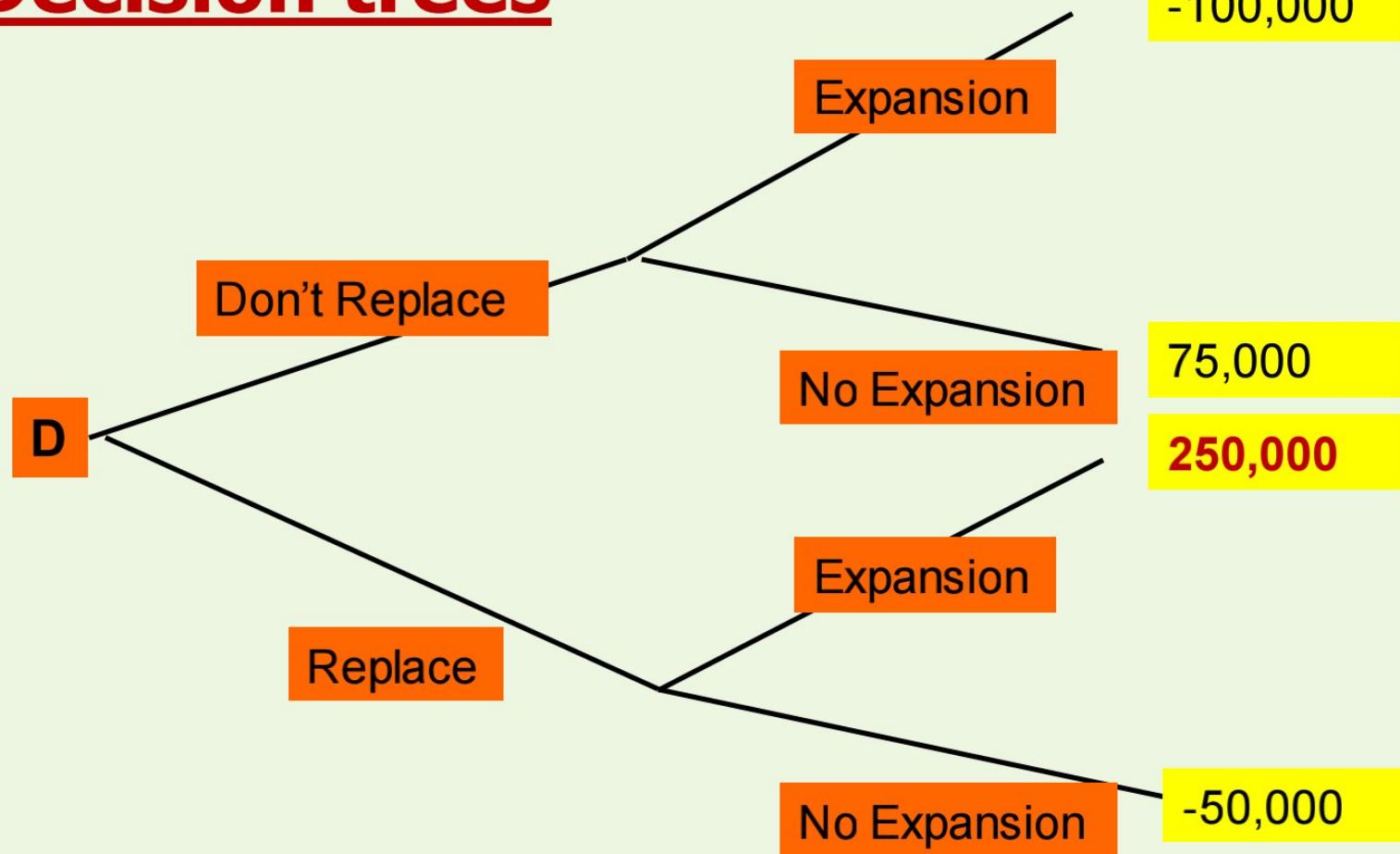
(2) Using “Decision Tree Analysis”

- It is important to see how a **decision** will affect the **future profitability** of a project
- **Ex:**
 - Not replacing old systems with new ones could result in revenue loss
 - Replacing now will result in more spending
- **D.T analysis** consists of evaluating expected benefits of taking **each path** from a **decision point**

Risk Evaluation

Expected Benefits

Decision trees





Thank You !!!