

Software Engineering



Why Software Engineering ?

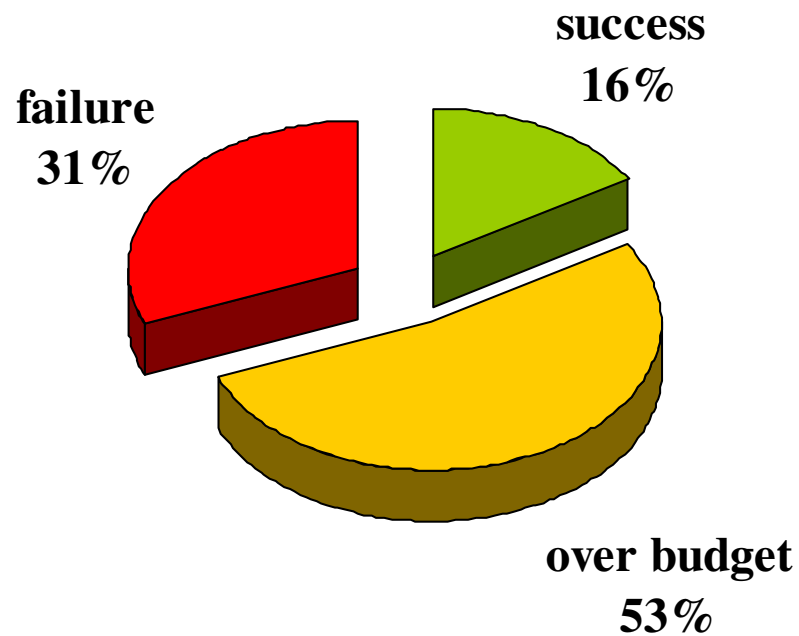
- ❖ Change in nature & complexity of software
- ❖ Concept of one “guru” is over
- ❖ We all want improvement



Ready for change

The Evolving Role of Software

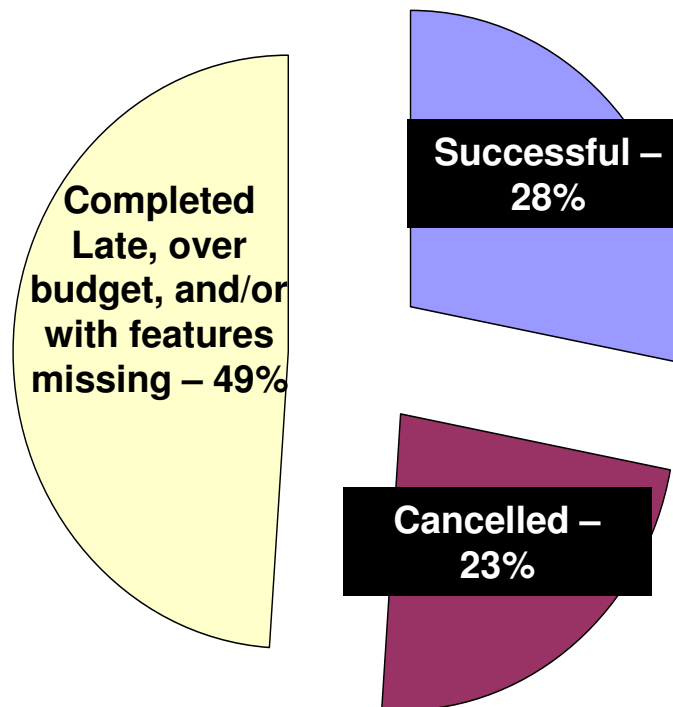
❖ Software industry is in Crisis!



Source: The Standish Group International, Inc. (CHAOS research)

The Evolving Role of Software

This is the
SORRY state
of Software
Engineering
Today!

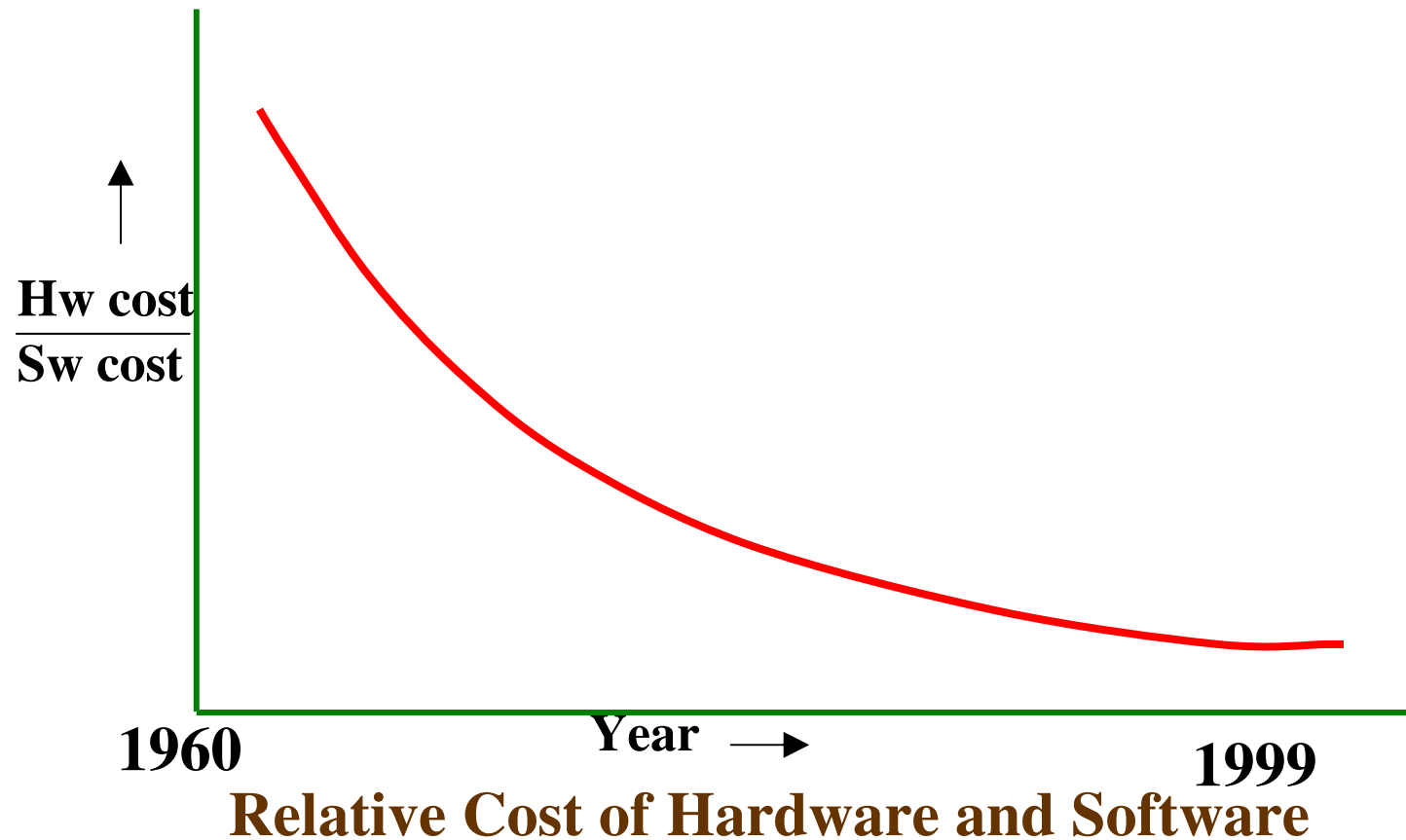


- **Data on 28,000 projects completed in 2000**

The Evolving Role of Software

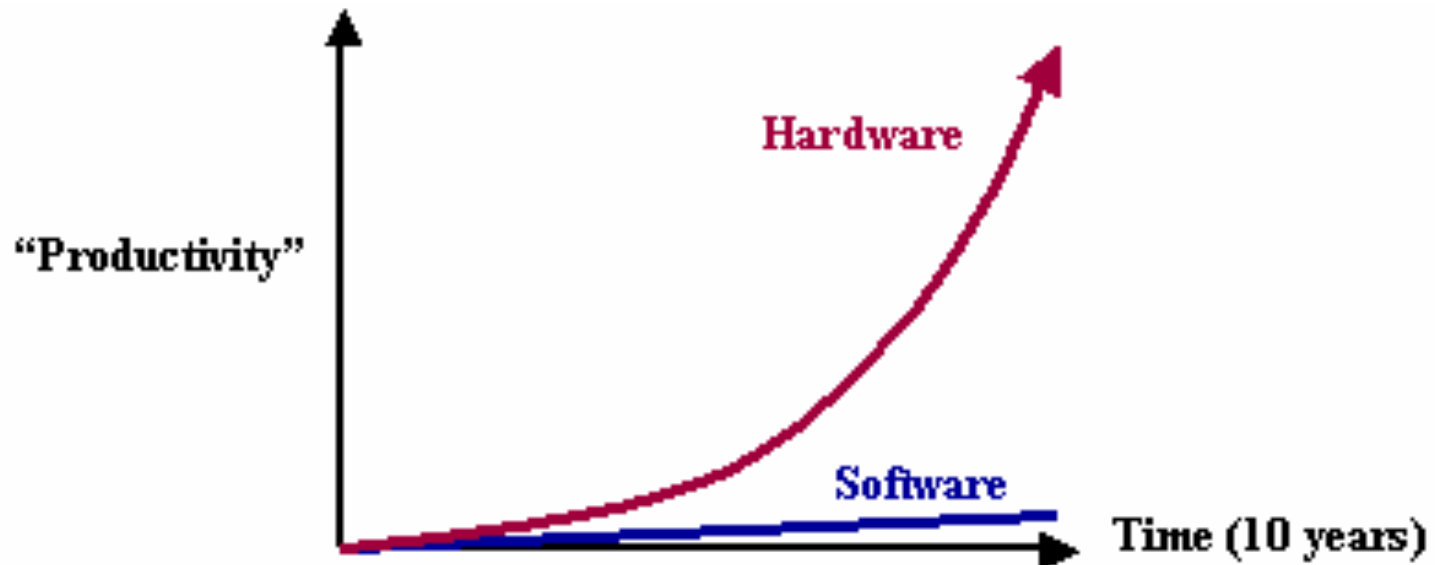
As per the IBM report, “31% of the project get cancelled before they are completed, 53% over-run their cost estimates by an average of 189% and for every 100 projects, there are 94 restarts”.

The Evolving Role of Software



The Evolving Role of Software

- Unlike Hardware
 - Moore's law: processor speed/memory capacity doubles every two years



The Evolving Role of Software

Managers and Technical Persons are asked:

- ✓ Why does it take so long to get the program finished?
- ✓ Why are costs so high?
- ✓ Why can not we find all errors before release?
- ✓ Why do we have difficulty in measuring progress of software development?

Factors Contributing to the Software Crisis

- Larger problems,
- Lack of adequate training in software engineering,
- Increasing skill shortage,
- Low productivity improvements.

Some Software failures

Ariane 5

It took the European Space Agency **10 years and \$7 billion** to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe overwhelming supremacy in the commercial space business.

The rocket was destroyed after 39 seconds of its launch, at an altitude of two and a half miles along with its payload of four expensive and uninsured scientific satellites.



Some Software failures

When the guidance system's own computer tried to convert one piece of data the sideways velocity of the rocket from a 64 bit format to a 16 bit format; the number was too big, and an overflow error resulted after 36.7 seconds. When the guidance system shutdown, it passed control to an identical, redundant unit, which was there to provide backup in case of just such a failure. Unfortunately, the second unit, which had failed in the identical manner a few milliseconds before.

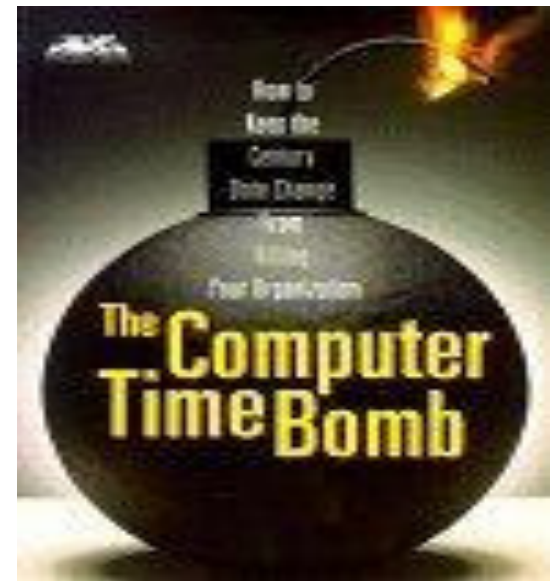


Some Software failures

Y2K problem:

It was simply the ignorance about the adequacy or otherwise of using only last two digits of the year.

The 4-digit date format, like 1964, was shortened to 2-digit format, like 64.



Some Software failures

The Patriot Missile

- o First time used in Gulf war
- o Used as a defense from Iraqi Scud missiles
- o Failed several times including one that killed 28 US soldiers in Dhahran, Saudi Arabia

Reasons:

A small timing error in the system's clock accumulated to the point that after 14 hours, the tracking system was no longer accurate. In the Dhahran attack, the system had been operating for more than 100 hours.



Some Software failures

The Space Shuttle

Part of an abort scenario for the Shuttle requires fuel dumps to lighten the spacecraft. It was during the second of these dumps that a (software) crash occurred.

...the fuel management module, which had performed one dump and successfully exited, restarted when recalled for the second fuel dump...



Some Software failures

A simple fix took care of the problem...but the programmers decided to see if they could come up with a systematic way to eliminate these generic sorts of bugs in the future. A random group of programmers applied this system to the fuel dump module and other modules.

Seventeen additional, previously unknown problems surfaced!

Some Software failures

Financial Software

Many companies have experienced failures in their accounting system due to faults in the software itself. The failures range from producing the wrong information to the whole system crashing.

Some Software failures

Windows XP

- o Microsoft released Windows XP on October 25, 2001.
- o On the same day company posted 18 MB of compatibility patches on the website for bug fixes, compatibility updates, and enhancements.
- o Two patches fixed important security holes.

This is Software Engineering.

“No Silver Bullet”

The hardware cost continues to decline drastically.

However, there are desperate cries for a silver bullet something to make software costs drop as rapidly as computer hardware costs do.

But as we look to the horizon of a decade, we see no silver bullet. There is no single development, either in technology or in management technique, that by itself promises even one order of magnitude improvement in productivity, in reliability and in simplicity.



“No Silver Bullet”

The hard part of building software is the specification, design and testing of this conceptual construct, not the labour of representing it and testing the correctness of representation.

We still make syntax errors, to be sure, but they are trivial as compared to the conceptual errors (logic errors) in most systems. That is why, building software is always hard and there is inherently no silver bullet.

While there is no royal road, there is a path forward.

Is reusability (and open source) the new silver bullet?

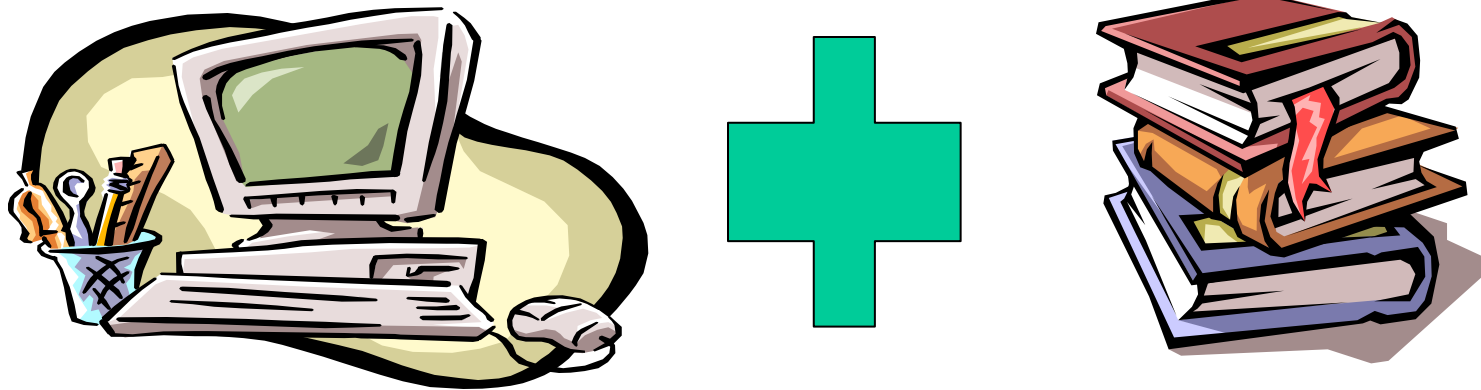
“No Silver Bullet”

The blame for software bugs belongs to:

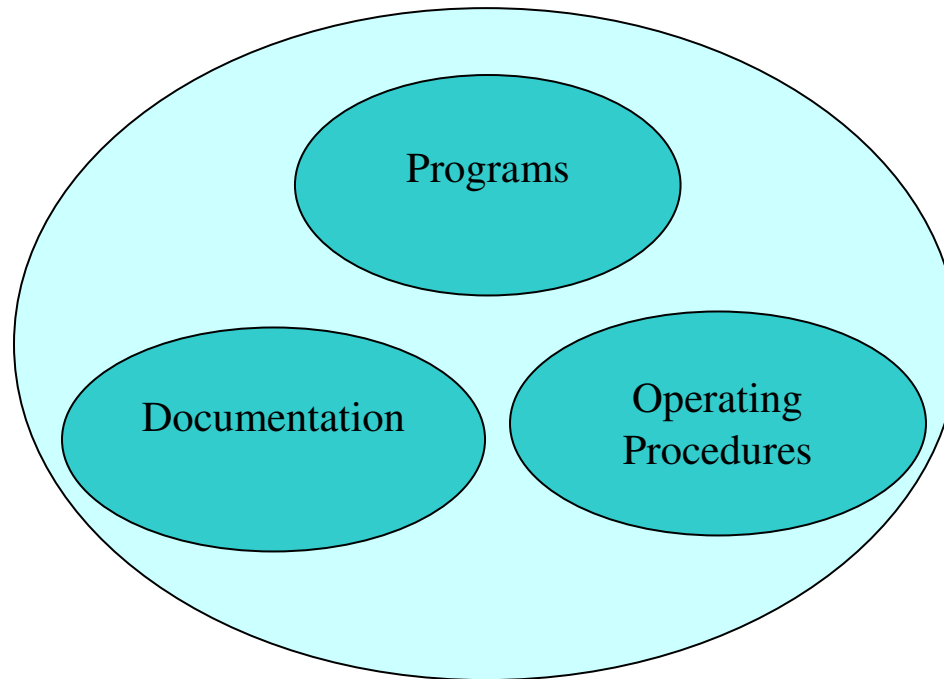
- Software companies
- Software developers
- Legal system
- Universities

What is software?

- **Computer programs** and **associated documentation**



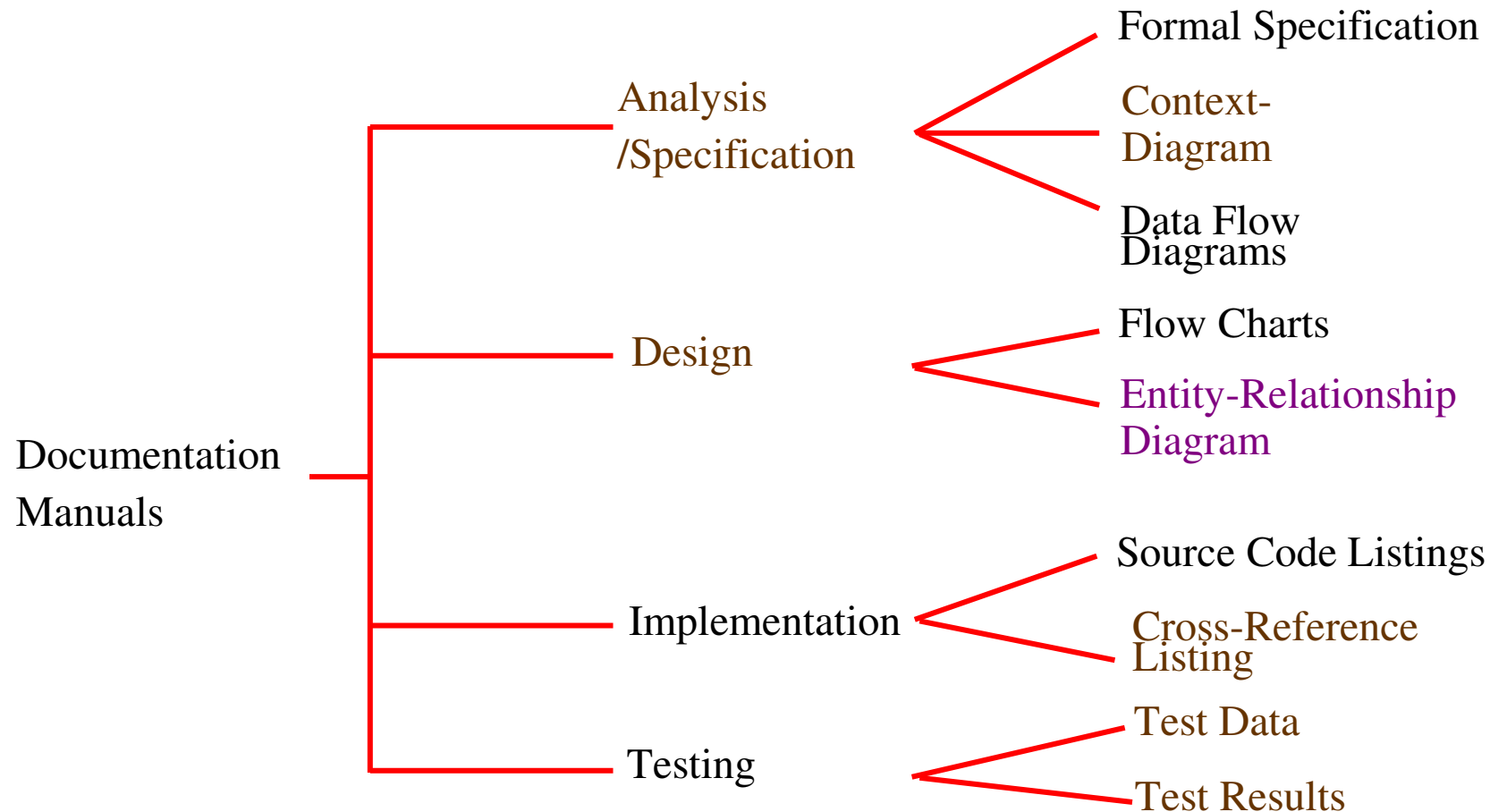
What is software?



Software=Program+Documentation+Operating Procedures

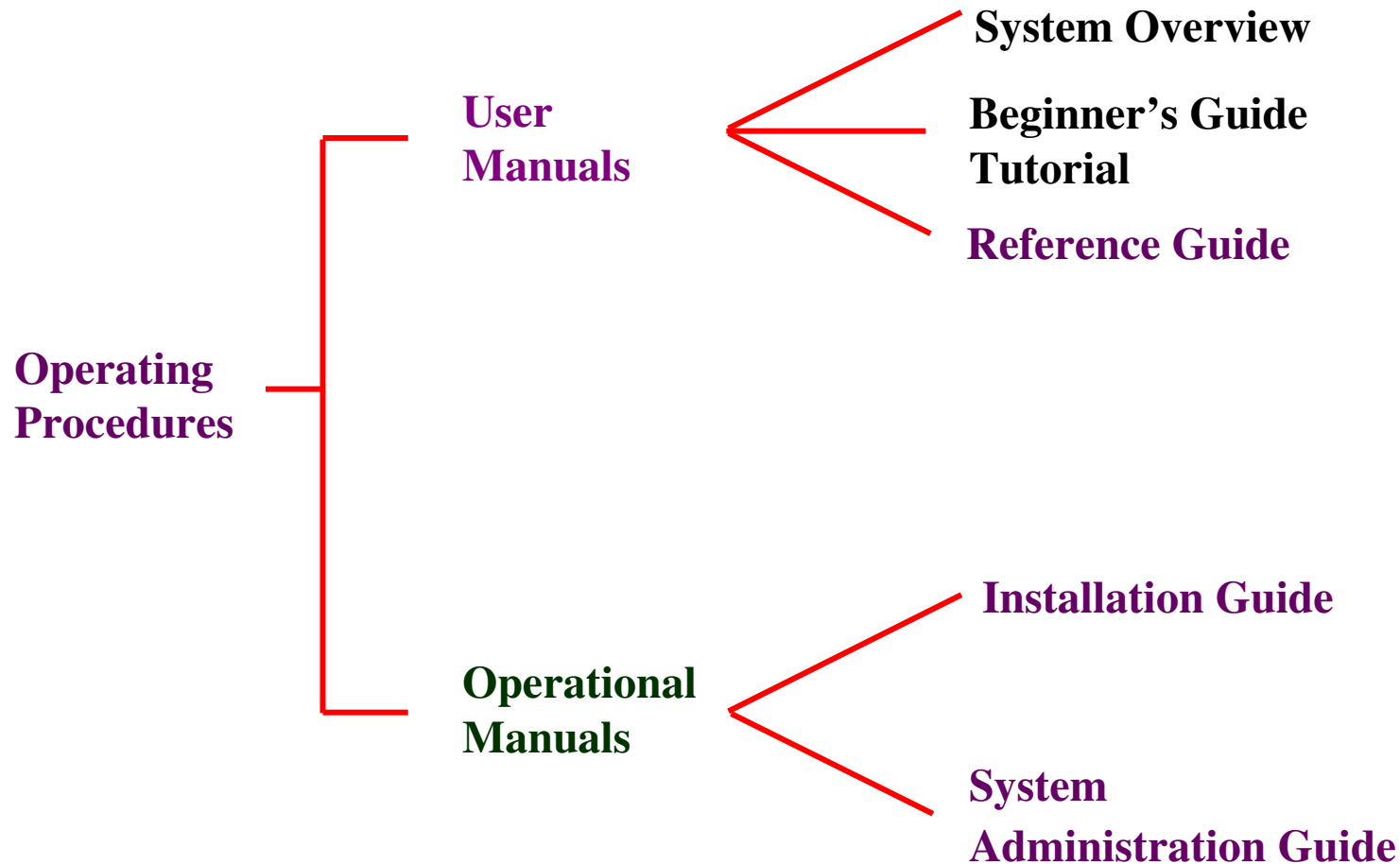
Components of software

Documentation consists of different types of manuals are



List of documentation manuals

Documentation consists of different types of manuals are



List of operating procedure manuals.

Software Product

- **Software products** may be developed for a particular customer or may be developed for a general market
- **Software products** may be
 - **Generic** - developed to be sold to a range of different customers
 - **Bespoke** (custom) - developed for a single customer according to their specification

Software Product

Software product is a product designated for delivery to the user



What is software engineering?

Software engineering is an engineering discipline which is concerned with all aspects of software production

Software engineers should

- adopt a systematic and organised approach to their work
- use appropriate tools and techniques depending on
 - the problem to be solved,
 - the development constraints and
- use the resources available



What is software engineering?

At the first conference on software engineering in 1968, Fritz Bauer defined software engineering as “The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines”.

Stephen Schach defined the same as “A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements”.

Both the definitions are popular and acceptable to majority. However, due to increase in cost of maintaining software, objective is now shifting to produce quality software that is maintainable, delivered on time, within budget, and also satisfies its requirements.

Software Process

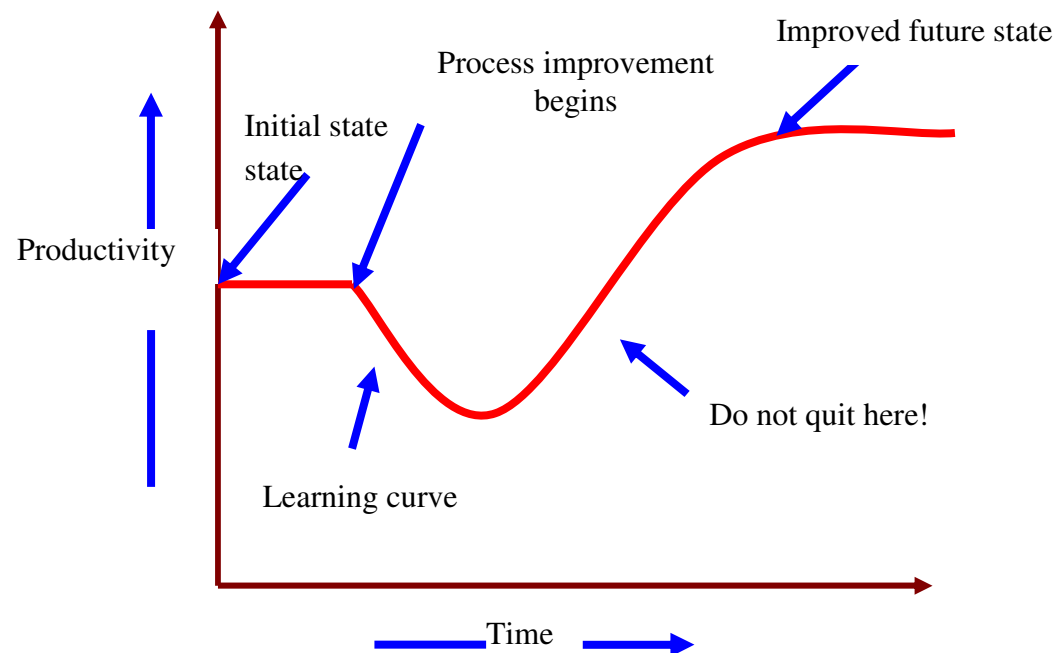
The software process is the way in which we produce software.

Why is it difficult to improve software process ?

- Not enough time
- Lack of knowledge

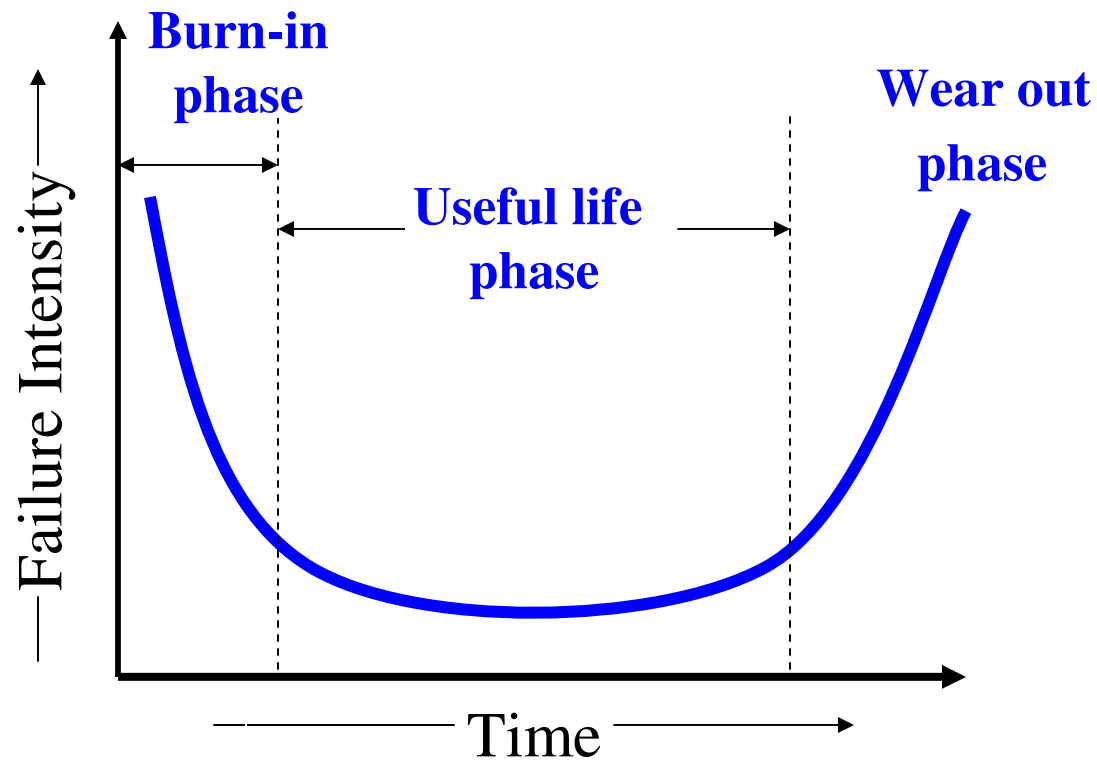
Software Process

- Wrong motivations
- Insufficient commitment



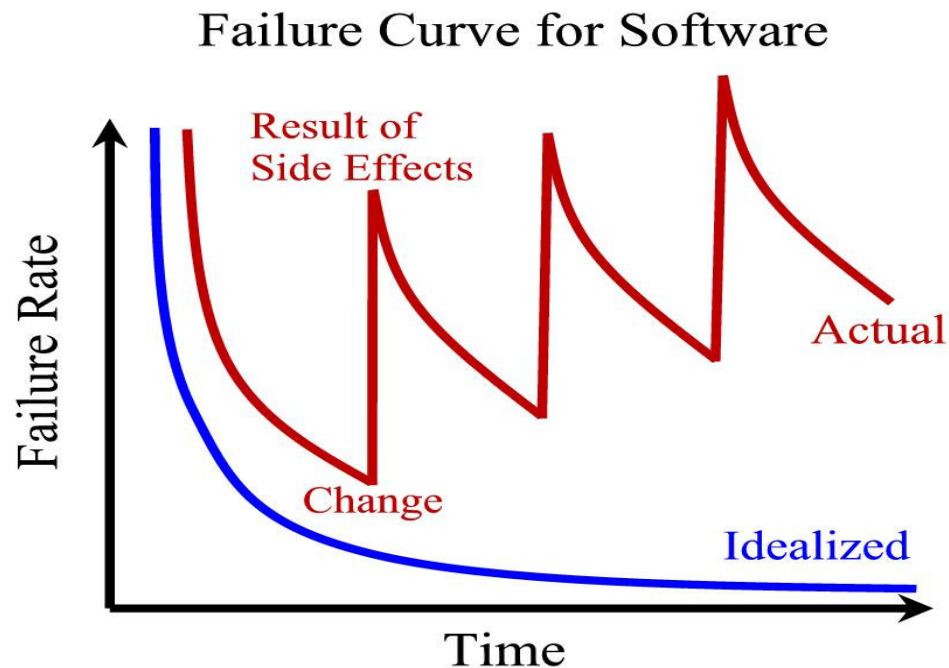
Software Characteristics:

✓ Software does not wear out.



Software Characteristics:

- ✓ Software is not manufactured
- ✓ Reusability of components
- ✓ Software is flexible

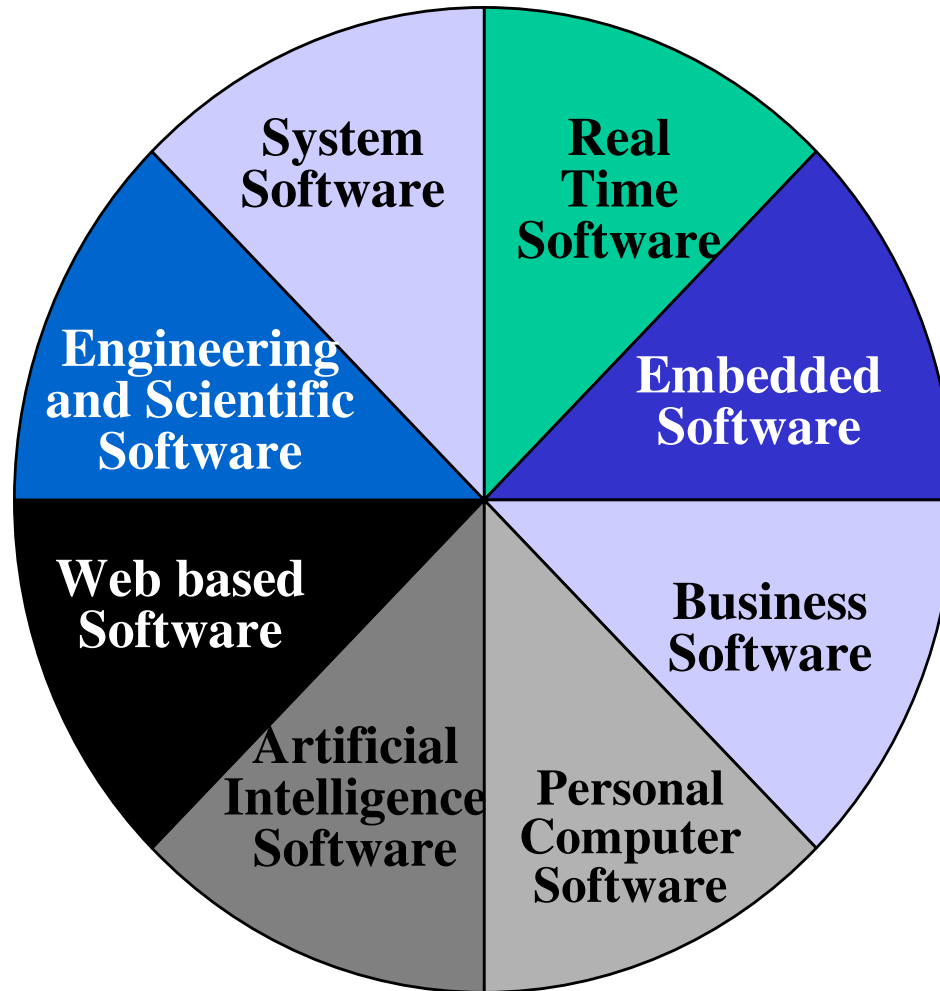


Software Characteristics:

Comparison of constructing a bridge vis-à-vis writing a program.

Sr. No	Constructing a bridge	Writing a program
1.	The problem is well understood	Only some parts of the problem are understood, others are not
2.	There are many existing bridges	Every program is different and designed for special applications.
3.	The requirement for a bridge typically do not change much during construction	Requirements typically change during all phases of development.
4.	The strength and stability of a bridge can be calculated with reasonable precision	Not possible to calculate correctness of a program with existing methods.
5.	When a bridge collapses, there is a detailed investigation and report	When a program fails, the reasons are often unavailable or even deliberately concealed.
6.	Engineers have been constructing bridges for thousands of years	Developers have been writing programs for 50 years or so.
7.	Materials (wood, stone, iron, steel) and techniques (making joints in wood, carving stone, casting iron) change slowly.	Hardware and software changes rapidly.

The Changing Nature of Software



The Changing Nature of Software

Trend has emerged to provide source code to the customer and organizations.

Software where source codes are available are known as open source software.

Examples

Open source software: LINUX, MySQL, PHP, Open office, Apache webserver etc.

Some Terminologies

➤ Deliverables and Milestones

Different deliverables are generated during software development. The examples are source code, user manuals, operating procedure manuals etc.

The milestones are the events that are used to ascertain the status of the project. Finalization of specification is a milestone. Completion of design documentation is another milestone. The milestones are essential for project planning and management.

Some Terminologies

➤ Product and Process

Product: What is delivered to the customer, is called a product. It may include source code, specification document, manuals, documentation etc. Basically, it is nothing but a set of deliverables only.

Process: Process is the way in which we produce software. It is the collection of activities that leads to (a part of) a product. An efficient process is required to produce good quality products.

If the process is weak, the end product will undoubtedly suffer, but an obsessive over reliance on process is also dangerous.

Some Terminologies

➤ Measures, Metrics and Measurement

A measure provides a quantitative indication of the extent, dimension, size, capacity, efficiency, productivity or reliability of some attributes of a product or process.

Measurement is the act of evaluating a measure.

A metric is a quantitative measure of the degree to which a system, component or process possesses a given attribute.

Some Terminologies

➤ Software Process and Product Metrics

Process metrics quantify the attributes of software development process and environment;

whereas product metrics are measures for the software product.

Examples

Process metrics: Productivity, Quality, Efficiency etc.

Product metrics: Size, Reliability, Complexity etc.

Some Terminologies

➤ Productivity and Effort

Productivity is defined as the rate of output, or production per unit of effort, i.e. the output achieved with regard to the time taken but irrespective of the cost incurred.

Hence most appropriate unit of effort is Person Months (PMs), meaning thereby number of persons involved for specified months. So, productivity may be measured as LOC/PM (lines of code produced/person month)

Some Terminologies

➤ Module and Software Components

There are many definitions of the term module. They range from “a module is a FORTRAN subroutine” to “a module is an Ada Package”, to “Procedures and functions of PASCAL and C”, to “C++ Java classes” to “Java packages” to “a module is a work assignment for an individual developer”. All these definition are correct. The term subprogram is also used sometimes in place of module.

Some Terminologies

“An independently deliverable piece of functionality providing access to its services through interfaces”.

“A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces”.

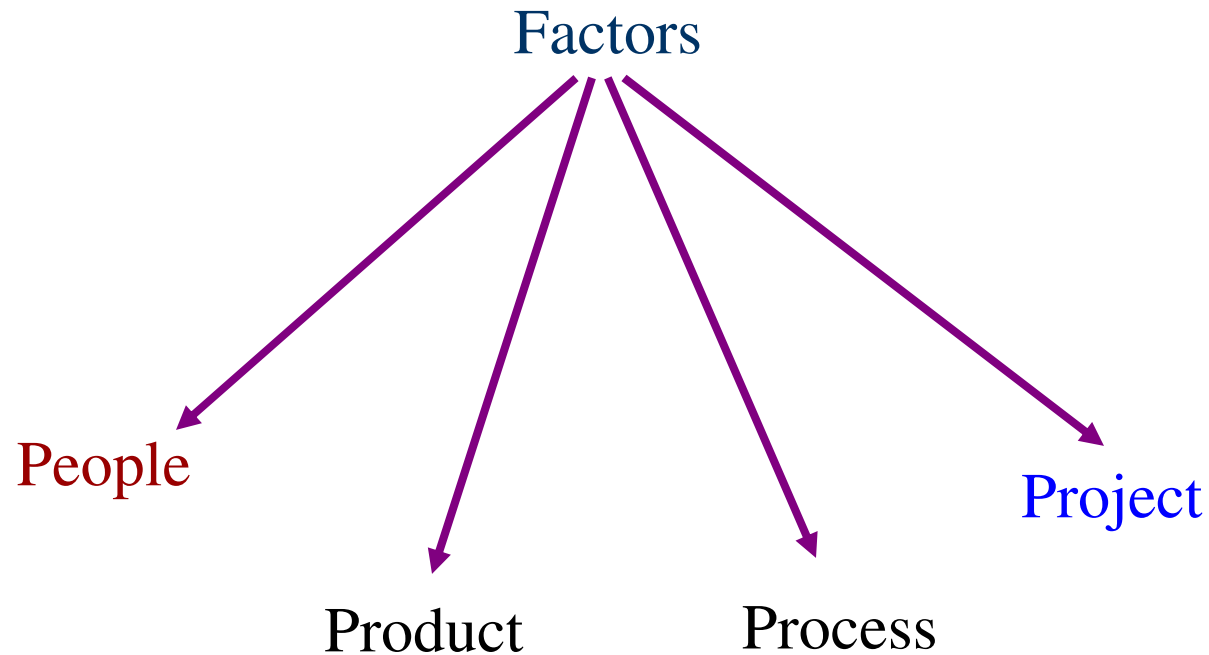
Some Terminologies

➤ Generic and Customized Software Products

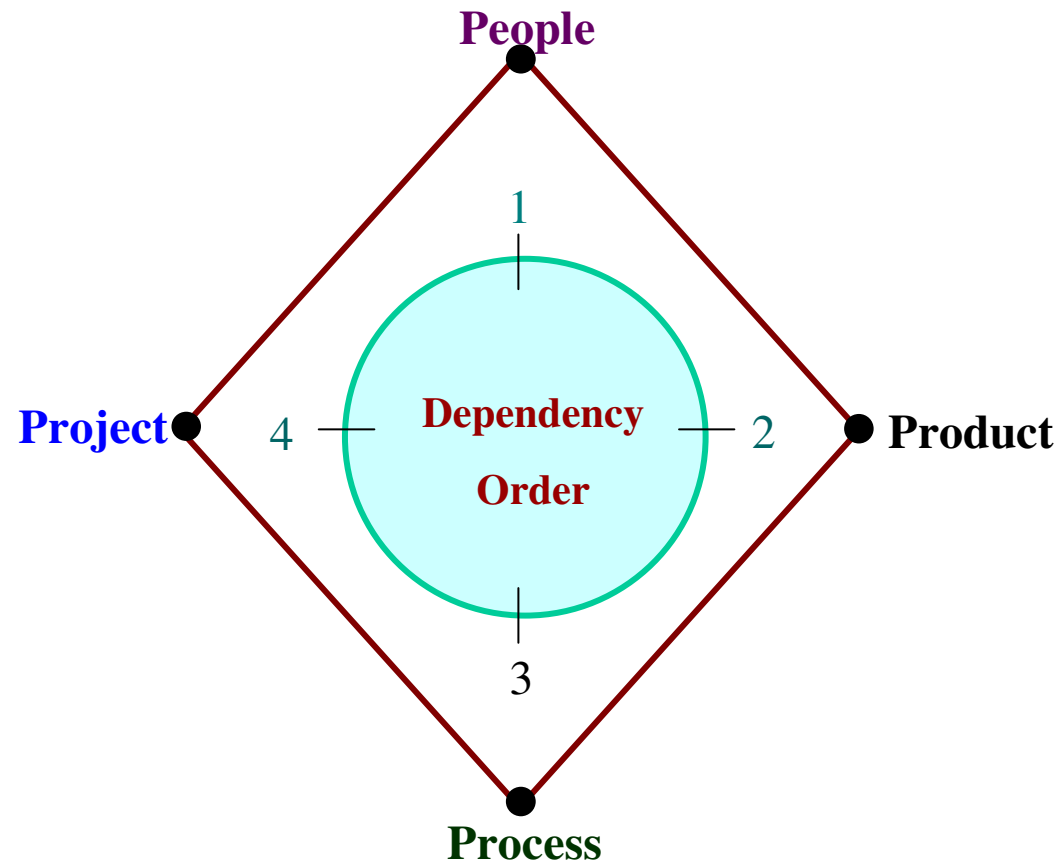
Generic products are developed for anonymous customers. The target is generally the entire world and many copies are expected to be sold. Infrastructure software like operating system, compilers, analyzers, word processors, CASE tools etc. are covered in this category.

The customized products are developed for particular customers. The specific product is designed and developed as per customer requirements. Most of the development projects (say about 80%) come under this category.

Role of Management in Software Development



Role of Management in Software Development





Software Life Cycle Models

Software Life Cycle Models

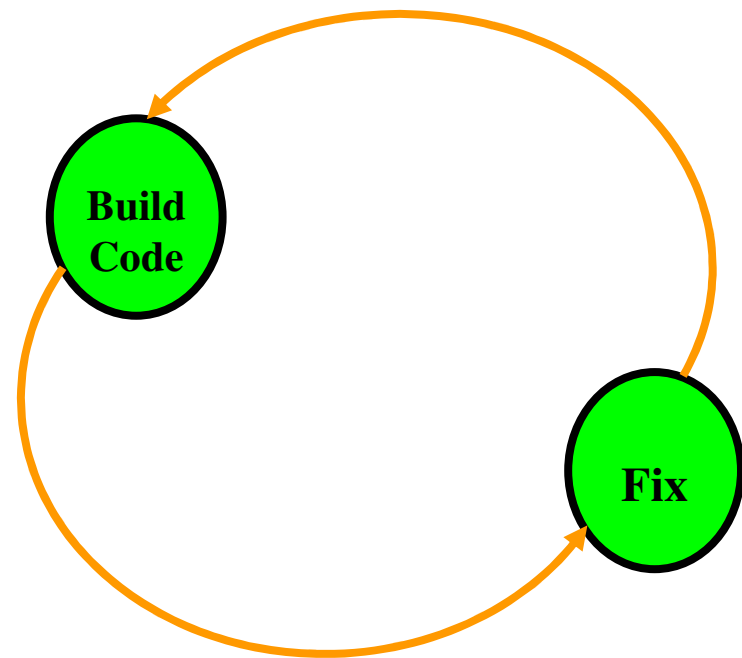
The goal of Software Engineering is to provide models and processes that lead to the production of well-documented maintainable software in a manner that is predictable.

Software Life Cycle Models

“The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle typically includes a requirement phase, design phase, implementation phase, test phase, installation and check out phase, operation and maintenance phase, and sometimes retirement phase”.

Build & Fix Model

- ❖ Product is constructed without specifications or any attempt at design
- ❖ Adhoc approach and not well defined
- ❖ Simple two phase model

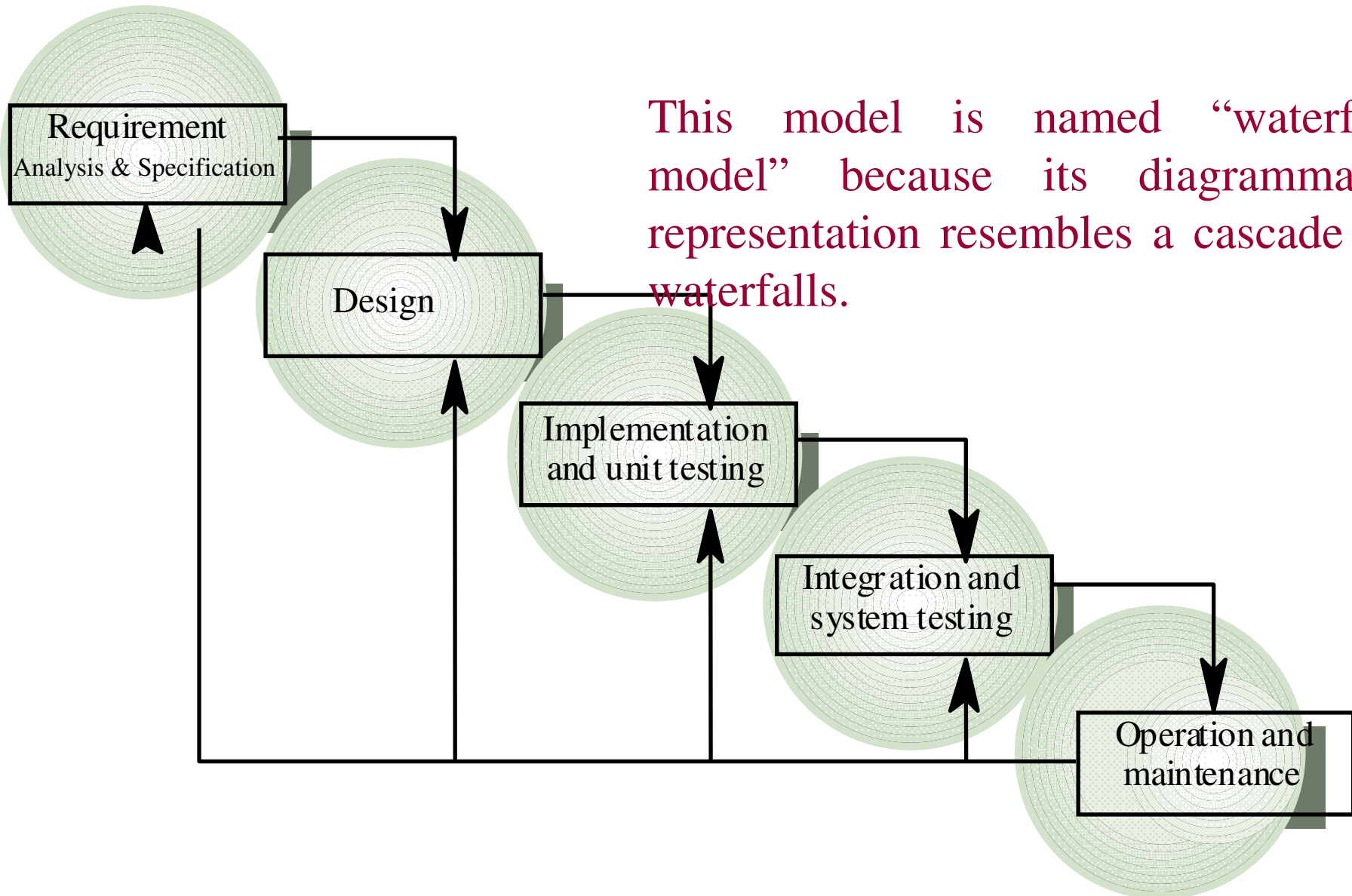


Build & Fix Model

- ❖ Suitable for small programming exercises of 100 or 200 lines
- ❖ Unsatisfactory for software for any reasonable size
- ❖ Code soon becomes unfixable & unenhanceable
- ❖ No room for structured design
- ❖ Maintenance is practically not possible

Waterfall Model

This model is named “waterfall model” because its diagrammatic representation resembles a cascade of waterfalls.



Waterfall Model

This model is easy to understand and reinforces the notion of “define before design” and “design before code”.

The model expects complete & accurate requirements early in the process, which is unrealistic

Waterfall Model

Problems of waterfall model

- i. It is difficult to define all requirements at the beginning of a project
- ii. This model is not suitable for accommodating any change
- iii. A working version of the system is not seen until late in the project's life
- iv. It does not scale up well to large projects.
- v. Real projects are rarely sequential.

Incremental Process Models

They are effective in the situations where requirements are defined precisely and there is no confusion about the functionality of the final product.

After every cycle a useable product is given to the customer.

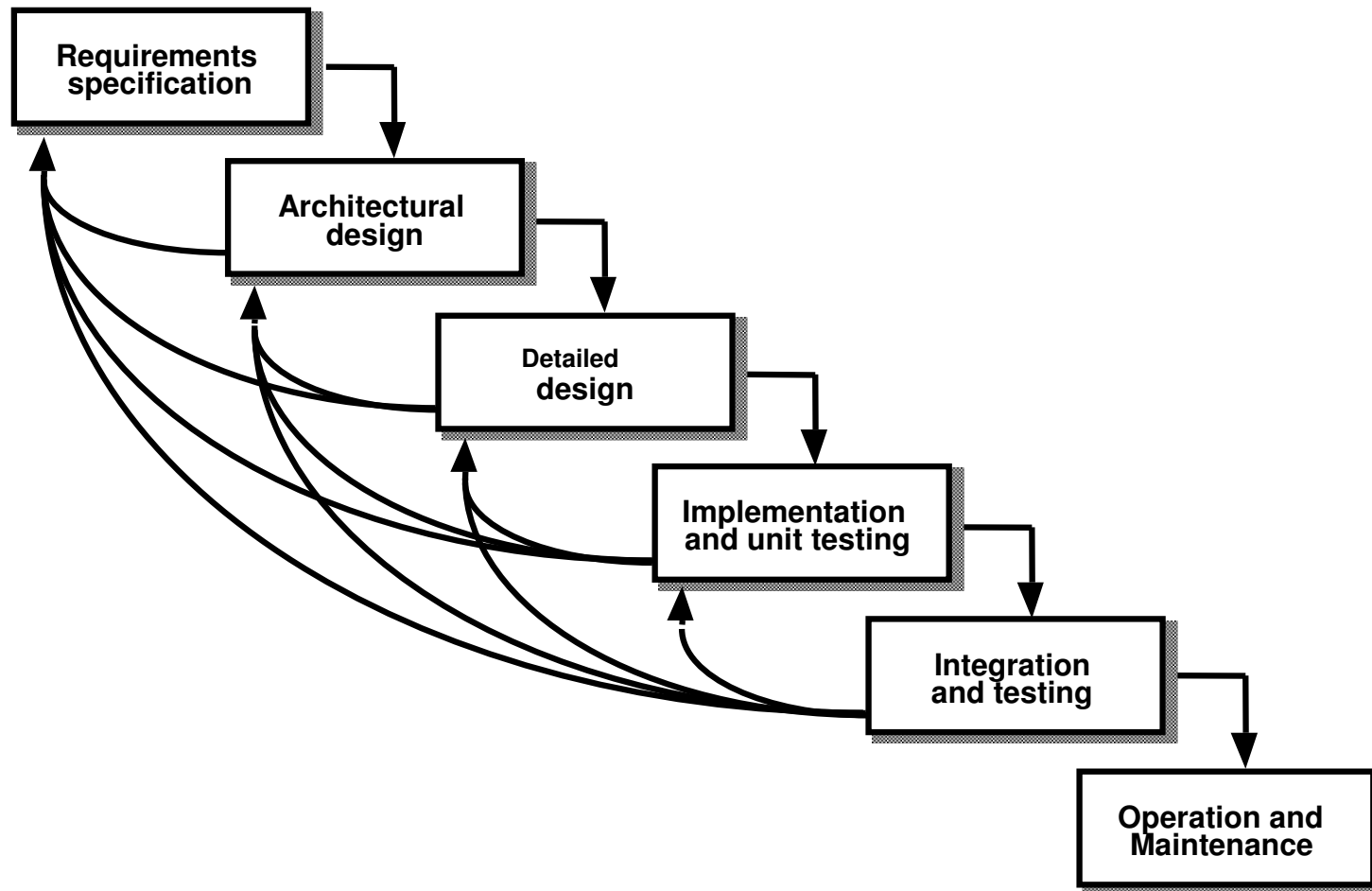
Popular particularly when we have to quickly deliver a limited functionality system.

Iterative Enhancement Model

This model has the same phases as the waterfall model, but with fewer restrictions. Generally the phases occur in the same order as in the waterfall model, but they may be conducted in several cycles. Useable product is released at the end of the each cycle, with each release providing additional functionality.

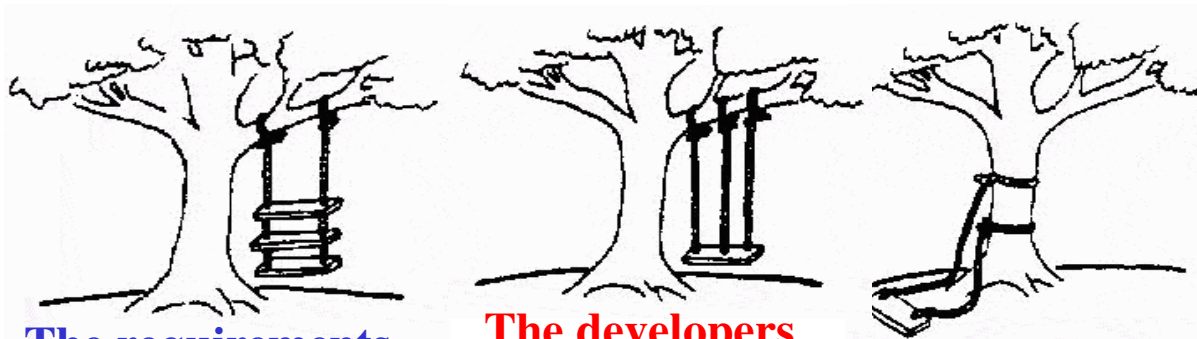
- ✓ Customers and developers specify as many requirements as possible and prepare a SRS document.
- ✓ Developers and customers then prioritize these requirements
- ✓ Developers implement the specified requirements in one or more cycles of design, implementation and test based on the defined priorities.

Iterative Enhancement Model



The Rapid Application Development (RAD) Model

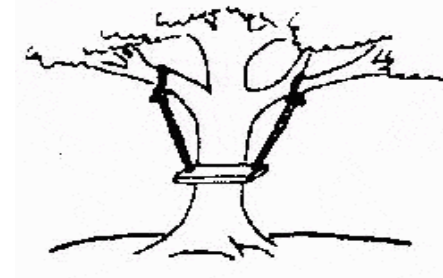
- o Developed by IBM in 1980
- o User participation is essential



The requirements specification was defined like this

The developers understood it in that way

This is how the problem was solved before.



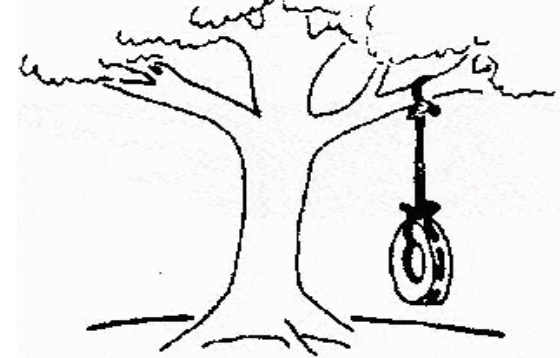
This is how the problem is solved now



That is the program after debugging



This is how the program is described by marketing department

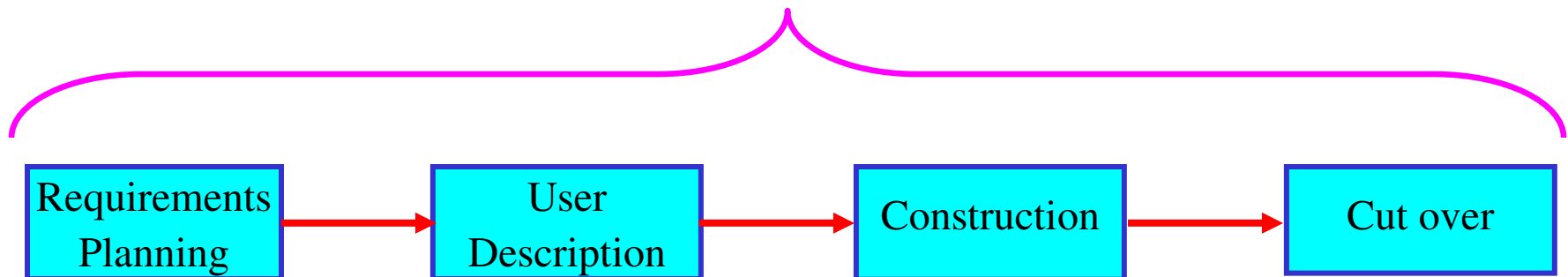


This, in fact, is what the customer wanted ...

The Rapid Application Development (RAD) Model

- o Build a rapid prototype
- o Give it to user for evaluation & obtain feedback
- o Prototype is refined

With active participation of users



The Rapid Application Development (RAD) Model

Not an appropriate model in the absence of user participation.

Reusable components are required to reduce development time.

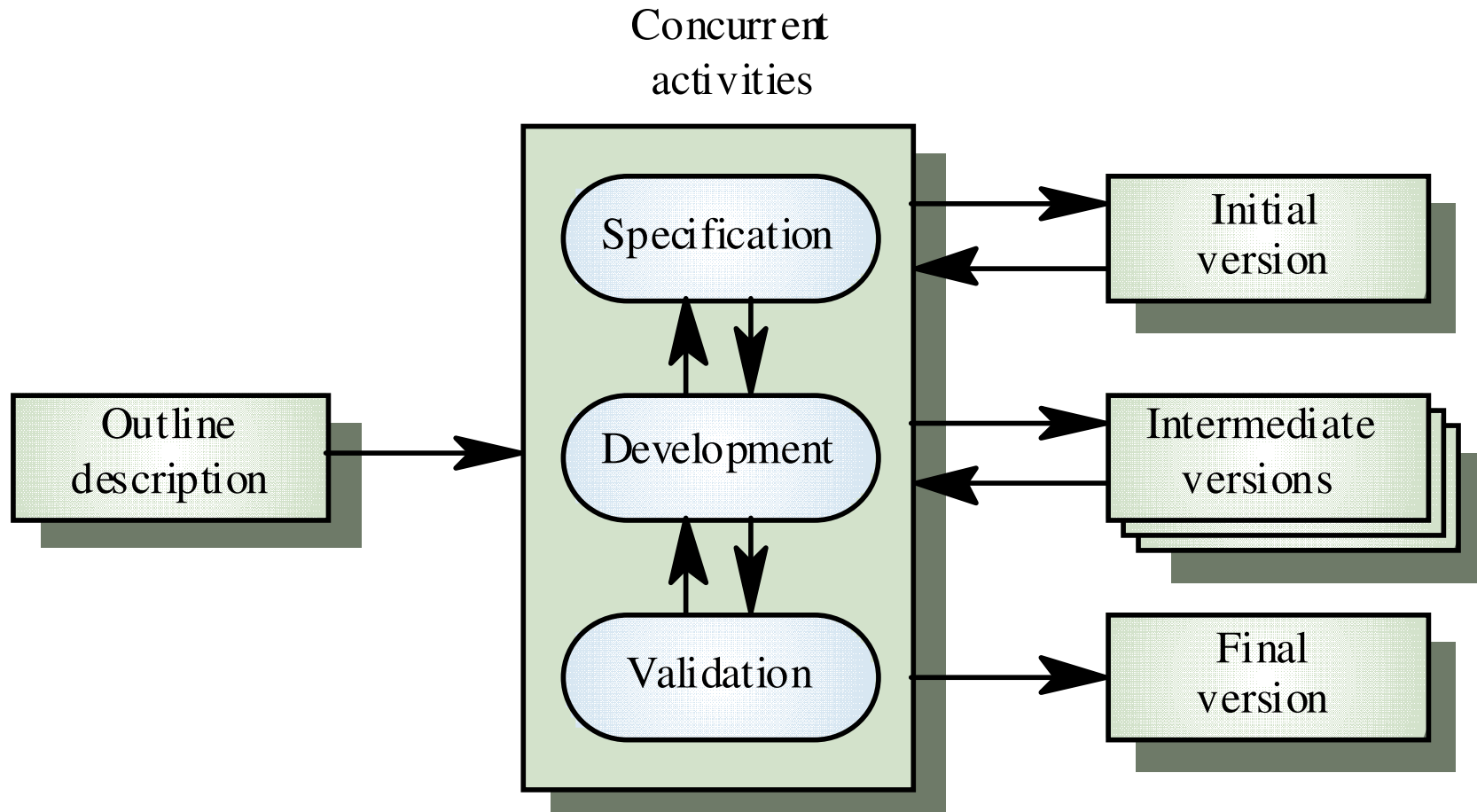
Highly specialized & skilled developers are required and such developers are not easily available.

Evolutionary Process Models

Evolutionary process model resembles iterative enhancement model. The same phases as defined for the waterfall model occur here in a cyclical fashion. This model differs from iterative enhancement model in the sense that this does not require a useable product at the end of each cycle. In evolutionary development, requirements are implemented by category rather than by priority.

This model is useful for projects using new technology that is not well understood. This is also used for complex projects where all functionality must be delivered at one time, but the requirements are unstable or not well understood at the beginning.

Evolutionary Process Model

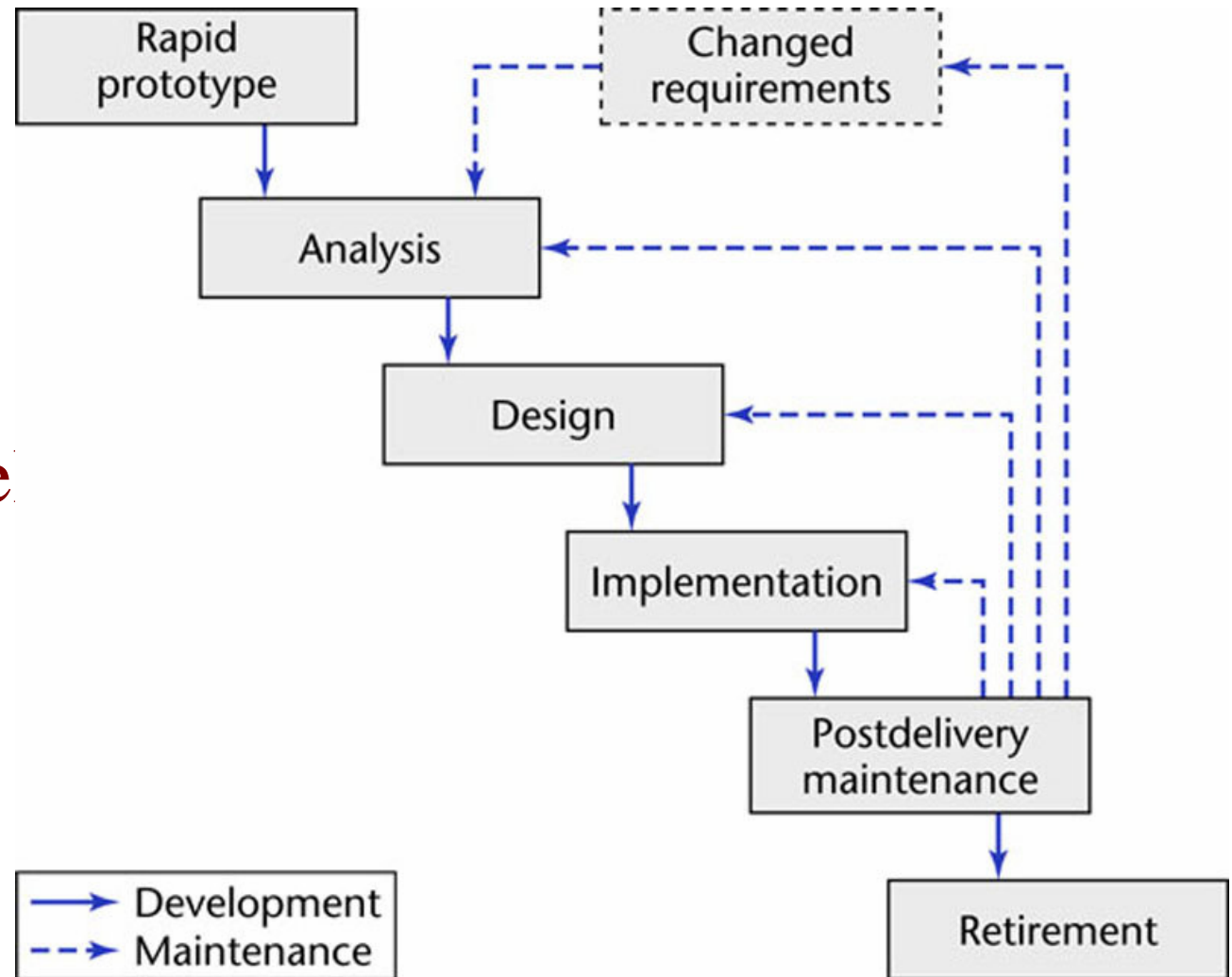


Prototyping Model

- The prototype may be a usable program but is not suitable as the final software product.
- The code for the prototype is thrown away. However experience gathered helps in developing the actual system.
- The development of a prototype might involve extra cost, but overall cost might turnout to be lower than that of an equivalent system developed using the waterfall model.

Prototyping Model

- Linear mode
- “Rapid”



Spiral Model

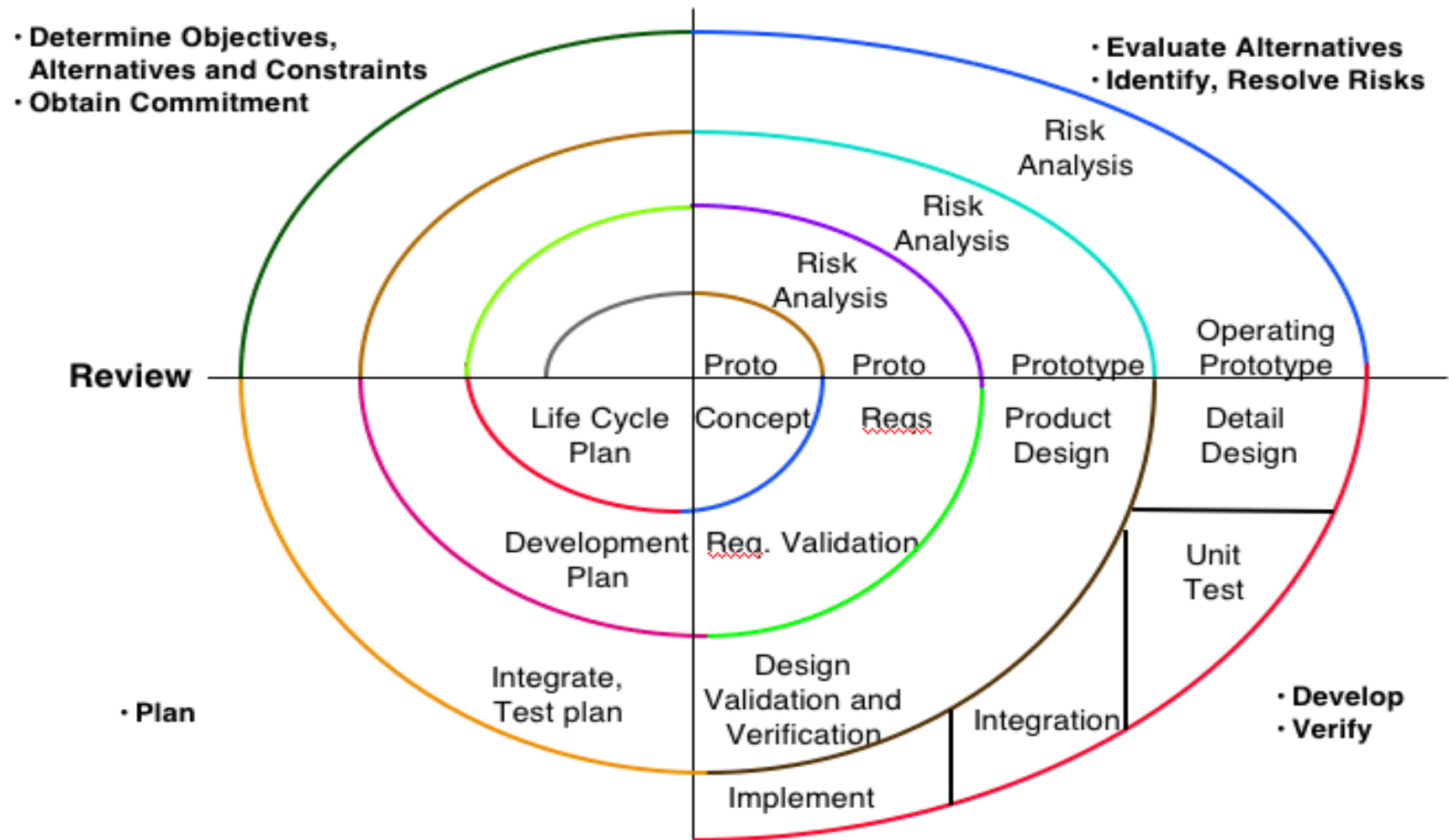
Models do not deal with uncertainty which is inherent to software projects.

Important software projects have failed because project risks were neglected & nobody was prepared when something unforeseen happened.

Barry Boehm recognized this and tried to incorporate the “project risk” factor into a life cycle model.

The result is the spiral model, which was presented in 1986.

Spiral Model



Spiral Model

The radial dimension of the model represents the cumulative costs. Each path around the spiral is indicative of increased costs. The angular dimension represents the progress made in completing each cycle. Each loop of the spiral from X-axis clockwise through 360° represents one phase. One phase is split roughly into four sectors of major activities.

- **Planning:** Determination of objectives, alternatives & constraints.
- **Risk Analysis:** Analyze alternatives and attempts to identify and resolve the risks involved.
- **Development:** Product development and testing product.
- **Assessment:** Customer evaluation

Spiral Model

- An important feature of the spiral model is that each phase is completed with a review by the people concerned with the project (designers and programmers)
- The advantage of this model is the wide range of options to accommodate the good features of other life cycle models.
- It becomes equivalent to another life cycle model in appropriate situations.

The spiral model has some difficulties that need to be resolved before it can be a universally applied life cycle model. These difficulties include lack of explicit process guidance in determining objectives, constraints, alternatives; relying on risk assessment expertise; and provides more flexibility than required for many applications.

Based On Characteristics Of Requirements

Requirements	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
Are requirements easily understandable and defined?	Yes	No	No	No	No	Yes
Do we change requirements quite often?	No	Yes	No	No	Yes	No
Can we define requirements early in the cycle?	Yes	No	Yes	Yes	No	Yes
Requirements are indicating a complex system to be built	No	Yes	Yes	Yes	Yes	No

Based On Status Of Development Team

Development team	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
Less experience on similar projects?	No	Yes	No	No	Yes	No
Less domain knowledge (new to the technology)	Yes	No	Yes	Yes	Yes	No
Less experience on tools to be used	Yes	No	No	No	Yes	No
Availability of training if required	No	No	Yes	Yes	No	Yes

Based On User's Participation

Involvement of Users	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
User involvement in all phases	No	Yes	No	No	No	Yes
Limited user participation	Yes	No	Yes	Yes	Yes	No
User have no previous experience of participation in similar projects	No	Yes	Yes	Yes	Yes	No
Users are experts of problem domain	No	Yes	Yes	Yes	No	Yes

Based On Type Of Project With Associated Risk

Project type and risk	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
Project is the enhancement of the existing system	No	No	Yes	Yes	No	Yes
Funding is stable for the project	Yes	Yes	No	No	No	Yes
High reliability requirements	No	No	Yes	Yes	Yes	No
Tight project schedule	No	Yes	Yes	Yes	Yes	Yes
Use of reusable components	No	Yes	No	No	Yes	Yes
Are resources (time, money, people etc.) scare?	No	Yes	No	No	Yes	No



Software Requirements Analysis and specification


Requirement Engineering

Requirements describe

What not How

Produces one large document written in natural language contains a description of what the system will do without describing how it will do it.

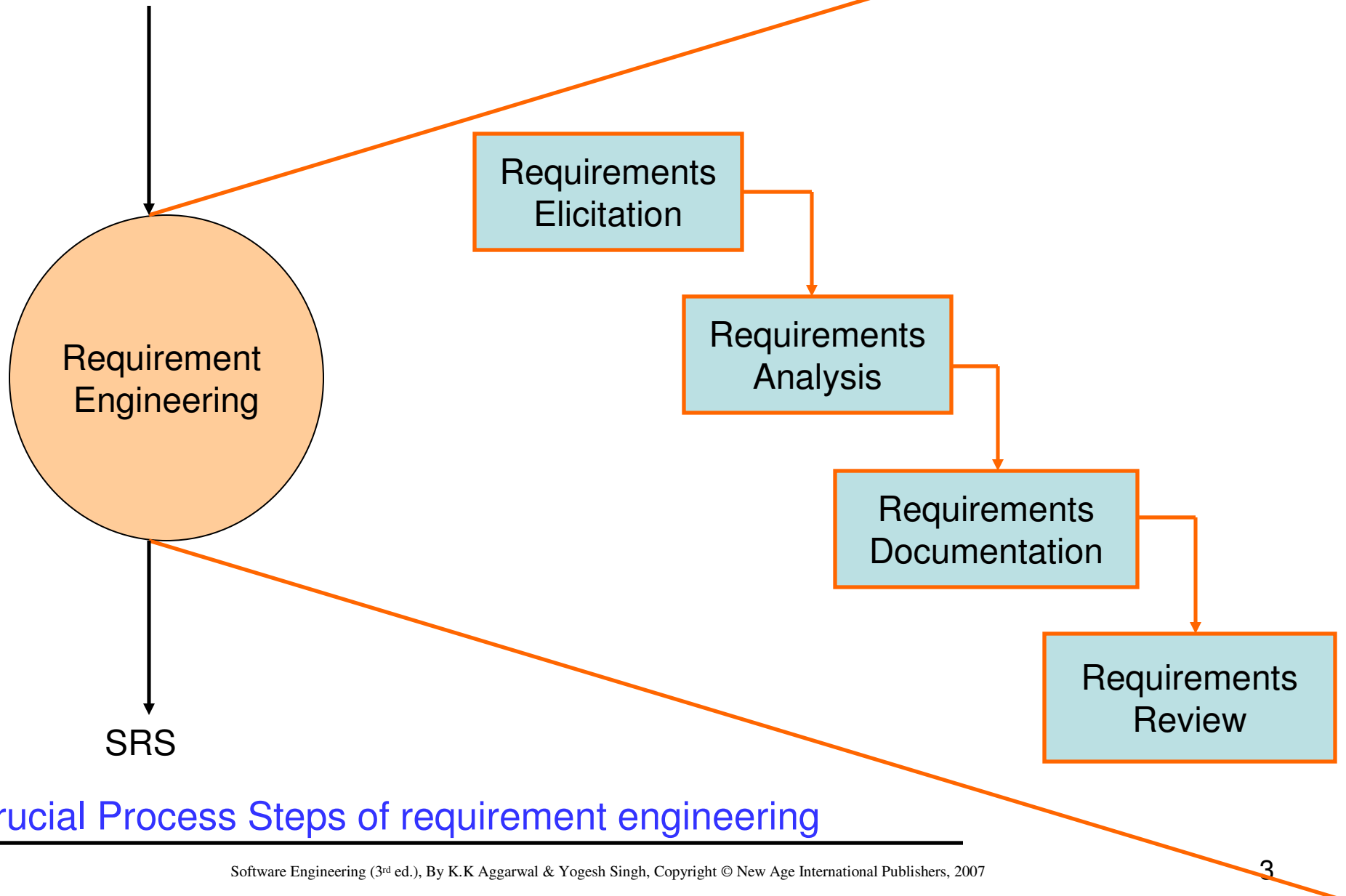
Crucial process steps

Quality of product  Process that creates it

Without well written document

- Developers do not know what to build
- Customers do not know what to expect
- What to validate

Problem Statement



Crucial Process Steps of requirement engineering

Requirement Engineering

Requirement Engineering is the disciplined application of proven principles, methods, tools, and notations to describe a proposed system's intended behavior and its associated constraints.

SRS may act as a contract between developer and customer.

State of practice

Requirements are difficult to uncover

- Requirements change
- Over reliance on CASE Tools
- Tight project Schedule
- Communication barriers
- Market driven software development
- Lack of resources

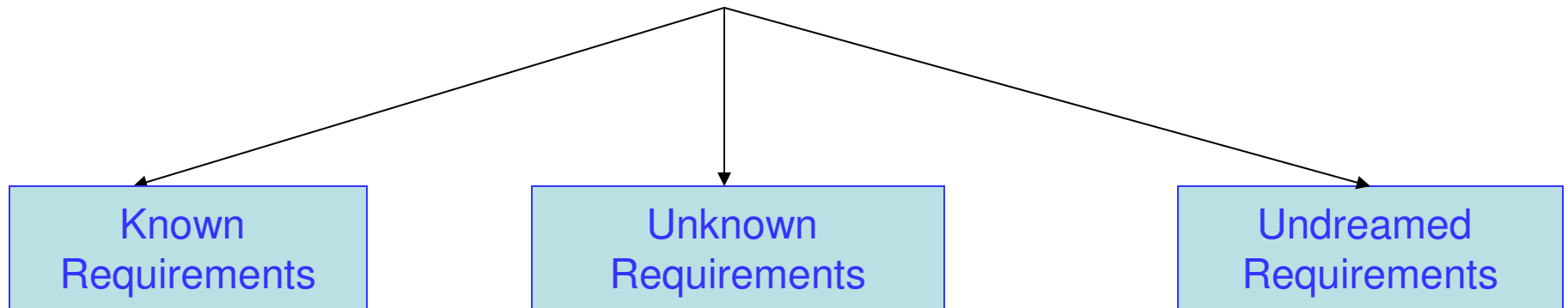
Requirement Engineering

Example

A University wish to develop a software system for the student result management of its M.Tech. Programme. A problem statement is to be prepared for the software development company. The problem statement may give an overview of the existing system and broad expectations from the new software system.

Types of Requirements

Types of Requirements

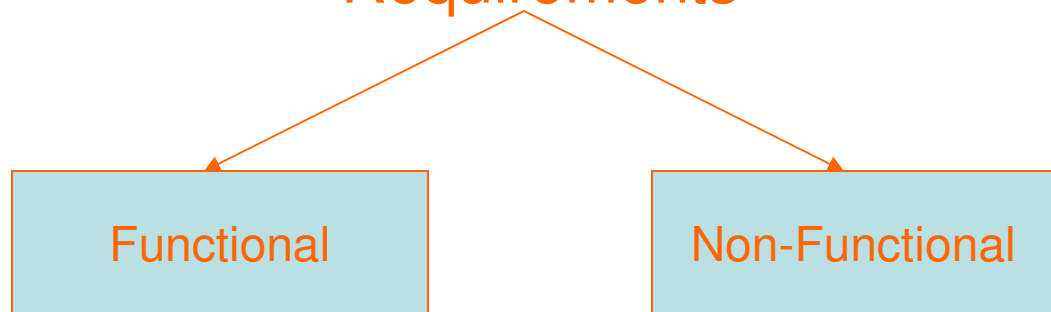


Stakeholder: Anyone who should have some direct or indirect influence on the system requirements.

--- User

--- Affected persons

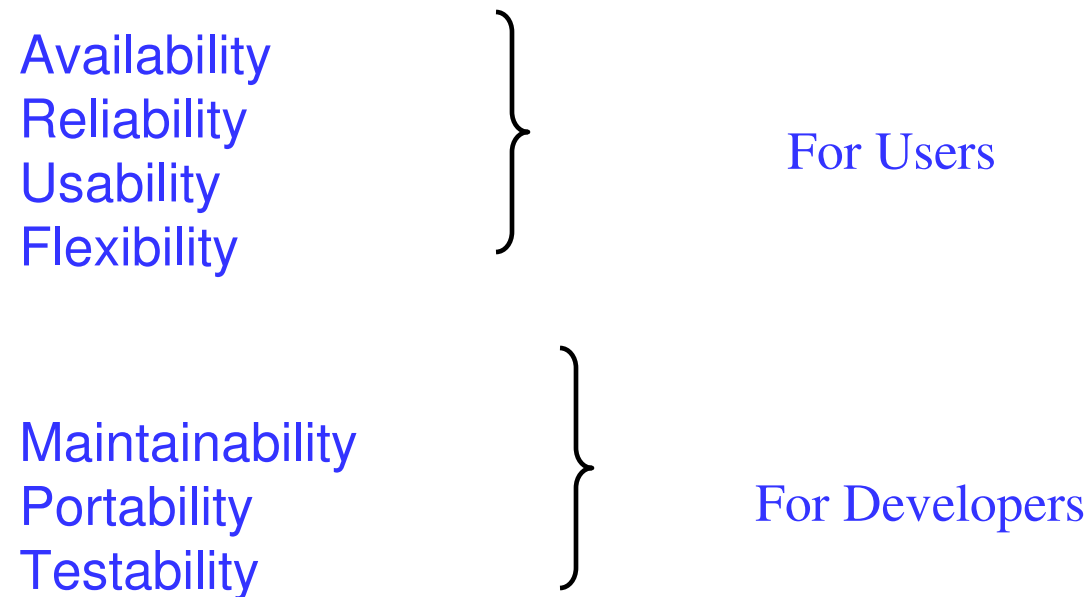
Requirements



Types of Requirements

Functional requirements describe what the software has to do. They are often called product features.

Non Functional requirements are mostly quality requirements. That stipulate how well the software does, what it has to do.



Types of Requirements

User and system requirements

- User requirement are written for the users and include functional and non functional requirement.
- System requirement are derived from user requirement.
- The user system requirements are the parts of software requirement and specification (SRS) document.

Types of Requirements

Interface Specification

- Important for the customers.

TYPES OF INTERFACES

- Procedural interfaces (also called Application Programming Interfaces (APIs)).
- Data structures
- Representation of data.

Feasibility Study

Is cancellation of a project a bad news?

As per IBM report, “31% projects get cancelled before they are completed, 53% over-run their cost estimates by an average of 189% & for every 100 projects, there are 94 restarts.

How do we cancel a project with the least work?

 **CONDUCT A FEASIBILITY STUDY**

Feasibility Study

Technical feasibility

- Is it technically feasible to provide direct communication connectivity through space from one location of globe to another location?
- Is it technically feasible to design a programming language using “Sanskrit”?

Feasibility Study

Feasibility depends upon non technical Issues like:

- Are the project's cost and schedule assumption realistic?
- Does the business model realistic?
- Is there any market for the product?

Feasibility Study

Purpose of feasibility study

“evaluation or analysis of the potential impact of a proposed project or program.”

Focus of feasibility studies

- Is the product concept viable?
- Will it be possible to develop a product that matches the project's vision statement?
- What are the current estimated cost and schedule for the project?

Feasibility Study

Focus of feasibility studies

- How big is the gap between the original cost & schedule targets & current estimates?
- Is the business model for software justified when the current cost & schedule estimate are considered?
- Have the major risks to the project been identified & can they be surmounted?
- Is the specifications complete & stable enough to support remaining development work?

Feasibility Study

Focus of feasibility studies

- Have users & developers been able to agree on a detailed user interface prototype? If not, are the requirements really stable?
- Is the software development plan complete & adequate to support further development work?

Requirements Elicitation

Perhaps

- Most difficult
- Most critical
- Most error prone
- Most communication intensive

Succeed

 effective customer developer partnership

Selection of any method

1. It is the only method that we know
2. It is our favorite method for all situations
3. We understand intuitively that the method is effective in the present circumstances.

Normally we rely on first two reasons.

Requirements Elicitation

1. Interviews

Both parties have a common goal



--- open ended

--- structured



Interview

Success of the project

Selection of stakeholder

1. Entry level personnel
2. Middle level stakeholder
3. Managers
4. Users of the software (Most important)

Requirements Elicitation

Types of questions.

- Any problems with existing system
- Any Calculation errors
- Possible reasons for malfunctioning
- No. of Student Enrolled

Requirements Elicitation

- 5. Possible benefits
- 6. Satisfied with current policies
- 7. How are you maintaining the records of previous students?
- 8. Any requirement of data from other system
- 9. Any specific problems
- 10. Any additional functionality
- 11. Most important goal of the proposed development

At the end, we may have wide variety of expectations from the proposed software.

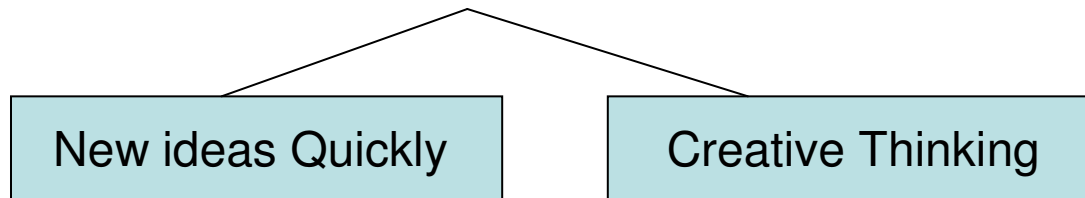
Requirements Elicitation

2. Brainstorming Sessions

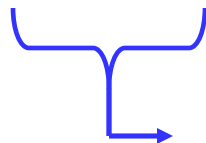
It is a group technique



group discussions



Prepare long list of requirements



Categorized
Prioritized
Pruned

*Idea is to generate views ,not to vet them.

Groups

1. Users 2. Middle Level managers 3. Total Stakeholders

Requirements Elicitation

A Facilitator may handle group bias, conflicts carefully.

- Facilitator may follow a published agenda
- Every idea will be documented in a way that everyone can see it.
- A detailed report is prepared.

3. Facilitated Application specification Techniques (FAST)

- Similar to brainstorming sessions.
- Team oriented approach
- Creation of joint team of customers and developers.

Requirements Elicitation

Guidelines

1. Arrange a meeting at a neutral site.
2. Establish rules for participation.
3. Informal agenda to encourage free flow of ideas.
4. Appoint a facilitator.
5. Prepare definition mechanism board, worksheets, wall stickier.
6. Participants should not criticize or debate.

FAST session Preparations

Each attendee is asked to make a list of objects that are:

Requirements Elicitation

1. Part of environment that surrounds the system.
2. Produced by the system.
3. Used by the system.
 - A. List of constraints
 - B. Functions
 - C. Performance criteria

Activities of FAST session

1. Every participant presents his/her list
2. Combine list for each topic
3. Discussion
4. Consensus list
5. Sub teams for mini specifications
6. Presentations of mini-specifications
7. Validation criteria
8. A sub team to draft specifications

Requirements Elicitation

4. Quality Function Deployment

-- Incorporate voice of the customer

Technical requirements.

Documented

Prime concern is customer satisfaction

What is important for customer?

- Normal requirements
- Expected requirements
- Exciting requirements

Requirements Elicitation

Steps

1. Identify stakeholders
2. List out requirements
3. Degree of importance to each requirement.

Requirements Elicitation

- 5 Points : V. Important
- 4 Points : Important
- 3 Points : Not Important but nice to have
- 2 Points : Not important
- 1 Points : Unrealistic, required further exploration

Requirement Engineer may categorize like:

- (i) It is possible to achieve
- (ii) It should be deferred & Why
- (iii) It is impossible and should be dropped from consideration

First Category requirements will be implemented as per priority assigned with every requirement.

Requirements Elicitation

5. The Use Case Approach

Ivar Jacobson & others introduced Use Case approach for elicitation & modeling.

Use Case – give functional view

The terms

Use Case	}	Often Interchanged But they are different
Use Case Scenario		
Use Case Diagram		

Use Cases are structured outline or template for the description of user requirements modeled in a structured language like English.

Requirements Elicitation

Use case Scenarios are unstructured descriptions of user requirements.

Use case diagrams are graphical representations that may be decomposed into further levels of abstraction.

Components of Use Case approach

Actor:

An actor or external agent, lies outside the system model, but interacts with it in some way.

Actor → Person, machine, information System

Requirements Elicitation

- Cockburn distinguishes between Primary and secondary actors.
- A Primary actor is one having a goal requiring the assistance of the system.
- A Secondary actor is one from which System needs assistance.

Use Cases

A use case is initiated by a user with a particular goal in mind, and completes successfully when that goal is satisfied.

Requirements Elicitation

- * It describes the sequence of interactions between actors and the system necessary to deliver the services that satisfies the goal.
- * Alternate sequence
- * System is treated as black box.

Thus

Use Case captures who (actor) does what (interaction) with the system, for what purpose (goal), without dealing with system internals.

Requirements Elicitation

*defines all behavior required of the system, bounding the scope of the system.

Jacobson & others proposed a template for writing Use cases as shown below:

1. Introduction

Describe a quick background of the use case.

2. Actors

List the actors that interact and participate in the use cases.

3. Pre Conditions

Pre conditions that need to be satisfied for the use case to perform.

4. Post Conditions

Define the different states in which we expect the system to be in, after the use case executes.

Requirements Elicitation

5. Flow of events

5.1 Basic Flow

List the primary events that will occur when this use case is executed.

5.2 Alternative Flows

Any Subsidiary events that can occur in the use case should be separately listed. List each such event as an alternative flow.

A use case can have many alternative flows as required.

6.Special Requirements

Business rules should be listed for basic & information flows as special requirements in the use case narration .These rules will also be used for writing test cases. Both success and failures scenarios should be described.

7.Use Case relationships

For Complex systems it is recommended to document the relationships between use cases. Listing the relationships between use cases also provides a mechanism for traceability

Use Case Template.

Requirements Elicitation

Use Case Guidelines

1. Identify all users
2. Create a user profile for each category of users including all roles of the users play that are relevant to the system.
3. Create a use case for each goal, following the use case template maintain the same level of abstraction throughout the use case. Steps in higher level use cases may be treated as goals for lower level (i.e. more detailed), sub-use cases.
4. Structure the use case
5. Review and validate with users.

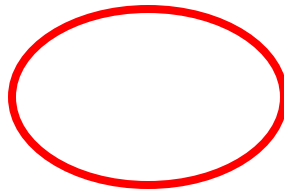
Requirements Elicitation

Use case Diagrams

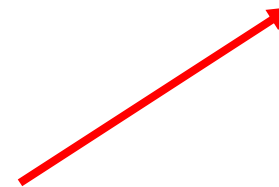
- represents what happens when actor interacts with a system.
- captures functional aspect of the system.



Actor



Use Case



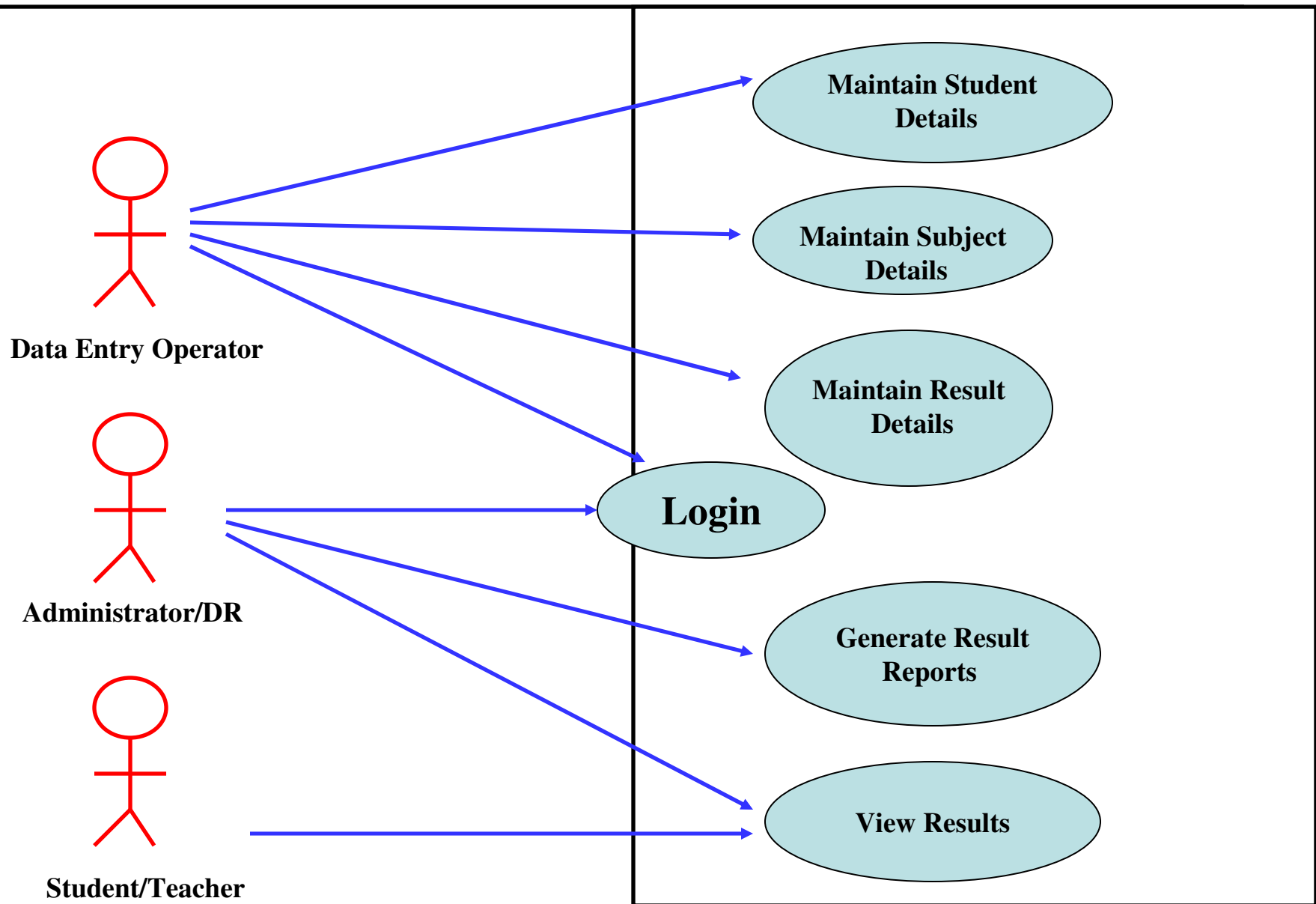
Relationship between actors and use case and/or between the use cases.

- Actors appear outside the rectangle.
- Use cases within rectangle providing functionality.
- Relationship association is a solid line between actor & use cases.

Requirements Elicitation

- *Use cases should not be used to capture all the details of the system.
- *Only significant aspects of the required functionality
- *No design issues
- *Use Cases are for “what” the system is , not “how” the system will be designed
- * Free of design characteristics

Use case diagram for Result Management System



Requirements Elicitation

1. Maintain student Details

Add/Modify/update students details like name, address.

2. Maintain subject Details

Add/Modify/Update Subject information semester wise

3. Maintain Result Details

Include entry of marks and assignment of credit points for each paper.

4. Login

Use to Provide way to enter through user id & password.

5. Generate Result Report

Use to print various reports

6. View Result

- (i) According to course code
- (ii) According to Enrollment number/roll number

Requirements Elicitation (Use Case)

Login

1.1 Introduction : This use case describes how a user logs into the Result Management System.

1.2 Actors :
(i) Data Entry Operator
(ii) Administrator/Deputy Registrar

1.3 Pre Conditions : None

1.4 Post Conditions : If the use case is successful, the actor is logged into the system. If not, the system state is unchanged.

Requirements Elicitation (Use Case)

1.5 Basic Flow : This use case starts when the actor wishes to login to the Result Management system.

- (i) System requests that the actor enter his/her name and password.
- (ii) The actor enters his/her name & password.
- (iii) System validates name & password, and if finds correct allow the actor to logs into the system.

Use Cases

1.6 Alternate Flows

1.6.1 Invalid name & password

If in the basic flow, the actor enters an invalid name and/or password, the system displays an error message. The actor can choose to either return to the beginning of the basic flow or cancel the login, at that point, the use case ends.

1.7 Special Requirements:

None

1.8 Use case Relationships:

None

Use Cases

2.Maintain student details

2.1 Introduction : Allow DEO to maintain student details. This includes adding, changing and deleting student information

2.2 Actors : DEO

2.3 Pre-Conditions: DEO must be logged onto the system before this use case begins.

Use Cases

2.4 Post-conditions : If use case is successful, student information is added/updated/deleted from the system. Otherwise, the system state is unchanged.

2.5 Basic Flow : Starts when DEO wishes to add/modify/update/delete Student information.

- (i) The system requests the DEO to specify the function, he/she would like to perform (Add/update/delete)
- (ii) One of the sub flow will execute after getting the information.

Use Cases

- ❑ If DEO selects "Add a student", "Add a student" sub flow will be executed.
- ❑ If DEO selects "update a student", "update a student" sub flow will be executed.
- ❑ If DEO selects "delete a student", "delete a student" sub flow will be executed.

2.5.1 Add a student

(i) The system requests the DEO to enter:

Name

Address

Roll No

Phone No

Date of admission

(ii) System generates unique id

2.5.2 Update a student

- (i) System requires the DEO to enter student-id.
- (ii) DEO enters the student_id. The system retrieves and display the student information.
- (iii) DEO makes the desired changes to the student information.
- (iv) After changes, the system updates the student record with changed information.

2.5.3 Delete a student

- (i) The system requests the DEO to specify the student-id.
- (ii) DEO enters the student-id. The system retrieves and displays the student information.
- (iii) The system prompts the DEO to confirm the deletion of the student.
- (iv) The DEO confirms the deletion.
- (v) The system marks the student record for deletion.

2.6 Alternative flows

2.6.1 Student not found

If in the update a student or delete a student sub flows, a student with specified_id does not exist, the system displays an error message .The DEO may enter a different id or cancel the operation. At this point ,Use case ends.

2.6.2 Update Cancelled

If in the update a student sub-flow, the data entry operator decides not to update the student information, the update is cancelled and the basic flow is restarted at the begin.

Use Cases

2.6.3 Delete cancelled

If in the delete a student sub flows, DEO decides not to delete student record ,the delete is cancelled and the basic flow is re-started at the beginning.

2.7 Special requirements

None

2.8 Use case relationships

None

Use Cases

3. Maintain Subject Details

3.1 Introduction

The DEO to maintain subject information. This includes adding, changing, deleting subject information from the system

3.2 Actors : DEO

3.3 Preconditions : DEO must be logged onto the system before the use case begins.

Use Cases

3.4 Post conditions :

If the use case is successful, the subject information is added, updated, or deleted from the system, otherwise the system state is unchanged.

3.5 Basic flows :

The use case starts when DEO wishes to add, change, and/or delete subject information from the system.

(i) The system requests DEO to specify the function he/she would like to perform i.e.

- Add a subject
- Update a subject
- Delete a subject.

Use Cases

(ii) Once the DEO provides the required information, one of the sub flows is executed.

- ☐ If DEO selected “Add a subject” the “Add-a subject sub flow is executed.
- ☐ If DEO selected “Update-a subject” the “update-a- subject” sub flow is executed
- ☐ If DEO selected “Delete- a- subject”, the “Delete-a-subject” sub flow is executed.

3.5.1 Add a Subject

- (i) The System requests the DEO to enter the subject information. This includes :
 - * Name of the subject

Use Cases

- * Subject Code
- * Semester
- * Credit points

(ii) Once DEO provides the requested information, the system generates and assigns a unique subject-id to the subject. The subject is added to the system.

(iii) The system provides the DEO with new subject-id.

3.5.2 Update a Subject

- (i) The system requests the DEO to enter subject_id.
- (ii) DEO enters the subject_id. The system retrieves and displays the subject information.
- (iii) DEO makes the changes.
- (iv) Record is updated.

3.5.3 Delete a Subject

- (i) Entry of subject_id.
- (ii) After this, system retrieves & displays subject information.
 - * System prompts the DEO to confirm the deletion.
 - * DEO verifies the deletion.
 - * The system marks the subject record for deletion.

3.6 Alternative Flow

3.6.1 Subject not found

If in any sub flows, subject-id not found, error message is displayed. The DEO may enter a different id or cancel the case ends here.

3.6.2 Update Cancelled

If in the update a subject sub-flow, the data entry operator decides not to update the subject information, the update is cancelled and the basic flow is restarted at the begin.

3.6.3 Delete Cancellation

If in delete-a-subject sub flow, the DEO decides not to delete subject, the delete is cancelled, and the basic flow is restarted from the beginning.

3.7 Special Requirements:

None

3.8 Use Case-relationships

None

Use Cases

4. Maintain Result Details

4.1 Introduction

This use case allows the DEO to maintain subject & marks information of each student. This includes adding and/or deleting subject and marks information from the system.

4.2 Actor

DEO

4.3 Pre Conditions

DEO must be logged onto the system.

4.4 Post Conditions

If use case is successful ,marks information is added or deleted from the system. Otherwise, the system state is unchanged.

4.5 Basic Flow

This use case starts, when the DEO wishes to add, update and/or delete marks from the system.

- (i) DEO to specify the function
- (ii) Once DEO provides the information one of the subflow is executed.
 - * If DEO selected “Add Marks”, the Add marks subflow is executed.
 - * If DEO selected “Update Marks”, the update marks subflow is executed.
 - * If DEO selected “Delete Marks”, the delete marks subflow is executed.

4.5.1 Add Marks Records

Add marks information .This includes:

- a. Selecting a subject code.
- b. Selecting the student enrollment number.
- c. Entering internal/external marks for that subject code & enrollment number.

Use Cases

(ii) If DEO tries to enter marks for the same combination of subject and enrollment number, the system gives a message that the marks have already been entered.

(iii) Each record is assigned a unique result_id.

4.5.2 Delete Marks records

1. DEO makes the following entries:

- a. Selecting subject for which marks have to be deleted.
- b. Selecting student enrollment number.
- c. Displays the record with id number.
- d. Verify the deletion.
- e. Delete the record.

4.5.2 Update Marks records

1. The System requests DEO to enter the record_id.
2. DEO enters record_id. The system retrieves & displays the information.
3. DEO makes changes.
4. Record is updated.

Use Cases

4.5.3 Compute Result

(i) Once the marks are entered, result is computed for each student.

(ii) If a student has scored more than 50% in a subject, the associated credit points are allotted to that student.

(iii) The result is displayed with subject-code, marks & credit points.

4.6 Alternative Flow

4.6.1 Record not found

If in update or delete marks sub flows, marks with specified id number do not exist, the system displays an error message. DEO can enter another id or cancel the operation.

4.6.2 Delete Cancelled

If in Delete Marks, DEO decides not to delete marks, the delete is cancelled and basic flow is re-started at the beginning.

4.7 Special Requirements

None

4.8 Use case relationships

None

5 View/Display result

5.1 Introduction

This use case allows the student/Teacher or anyone to view the result. The result can be viewed on the basis of course code and/or enrollment number.

5.2 Actors

Administrator/DR, Teacher/Student

5.3 Pre Conditions

None

5.4 Post Conditions

If use case is successful, the marks information is displayed by the system. Otherwise, state is unchanged.

5.5 Basic Flow

Use case begins when student, teacher or any other person wish to view the result.

Two ways

- Enrollment no.
- Course code

Use Cases

(ii) After selection, one of the sub flow is executed.

Course code → Sub flow is executed

Enrollment no. → Sub flow is executed

5.5.1 View result enrollment number wise

(i) User to enter enrollment number

(ii) System retrieves the marks of all subjects with credit points

(iii) Result is displayed.

Use Cases

5.6 Alternative Flow

5.6.1 Record not found
Error message should be displayed.

5.7 Special Requirements

None

5.8 Use Case relationships

None

Use Cases

6. Generate Report

6.1 Introduction

This use case allows the DR to generate result reports. Options are

- a. Course code wise
- b. Semester wise
- c. Enrollment Number wise

6.2 Actors

DR

6.3 Pre-Conditions

DR must logged on to the system

Use Cases

6.4 Post conditions

If use case is successful, desired report is generated. Otherwise, the system state is unchanged.

6.5 Basic Flow

The use case starts, when DR wish to generate reports.

- (i) DR selects option.
- (ii) System retrieves the information displays.
- (iii) DR takes printed reports.

Use Cases

6.6 Alternative Flows

6.6.1 Record not found

If not found, system generates appropriate message. The DR can select another option or cancel the operation. At this point, the use case ends.

6.7 Special Requirements

None

6.8 Use case relationships

None

Use Cases

7. Maintain User Accounts

7.1 Introduction

This use case allows the administrator to maintain user account. This includes adding, changing and deleting user account information from the system.

7.2 Actors

Administrator

7.3 Pre-Conditions

The administrator must be logged on to the system before the use case begins.

Use Cases

7.4 Post-Conditions

If the use case was successful, the user account information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

7.5 Basic Flow

This use case starts when the Administrator wishes to add, change, and/or delete use account information from the system.

- (i) The system requests that the Administrator specify the function he/she would like to perform (either Add a User Account, Update a User Account, or Delete a User Account).
- (ii) Once the Administrator provides the requested information, one of the sub-flows is executed

Use Cases

- * If the Administrator selected “Add a User Account”, the **Add a User Account** sub flow is executed.
- * If the Administrator selected “Update a User Account”, the **Update a User Account** sub-flow is executed.
- * If the Administrator selected “Delete a User Account”, the **Delete a User Account** sub-flow is executed.22

7.5.1 Add a User Account

1. The system requests that the Administrator enters the user information. This includes:

- (a) User Name
- (b) User ID-should be unique for each user account
- (c) Password
- (d) Role

Use Cases

2. Once the Administrator provides the requested information, the user account information is added.

7.5.2 Update a User Account

1. The system requests that the Administrator enters the User ID.
2. The Administrator enters the User ID. The system retrieves and displays the user account information.
3. The Administrator makes the desired changes to the user account information. This includes any of the information specified in the **Add a User Account** sub-flow.
4. Once the Administrator updates the necessary information, the system updates the user account record with the updated information.

7.5.3 Delete a User Account

1. The system requests that the Administrator enters the User ID.
2. The Administrator enters the User ID. The system retrieves and displays the user account information.
3. The system prompts the Administrator to confirm the deletion of the user account.
4. The Administrator confirms the deletion.
5. The system deletes the user account record.

Use Cases

7.6 Alternative Flows

7.6.1 User Not Found

If in the **Update a User Account** or **Delete a User Account** sub-flows, a user account with the specified User ID does not exist, the system displays an error message. The Administrator can then enter a different User ID or cancel the operation, at which point the use case ends.

7.6.2 Update Cancelled

If in the **Update a User Account** sub-flow, the Administrator decides not to update the user account information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

7.6.3 Delete Cancelled

If in the **Delete a User Account** sub-flow, the Administrator decides not to delete the user account information, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

7.7 **Special Requirements**
None

7.8 **Use case relationships**
None

Use Cases

8. Reset System

8.1 Introduction

This use case allows the administrator to reset the system by deleting all existing information from the system .

8.2 Actors

Administrator

8.3 Pre-Conditions

The administrator must be logged on to the system before the use case begins.

Use Cases

8.4 Post-Conditions

If the use case was successful, all the existing information is deleted from the backend database of the system. Otherwise, the system state is unchanged.

8.5 Basic Flow

This use case starts when the Administrator wishes to reset the system.

- i. The system requests the Administrator to confirm if he/she wants to delete all the existing information from the system.
- ii. Once the Administrator provides confirmation, the system deletes all the existing information from the backend database and displays an appropriate message.

Use Cases

8.6 Alternative Flows

8.6.1 Reset Cancelled

If in the Basic Flow, the Administrator decides not to delete the entire existing information, the reset is cancelled and the use case ends.

8.7 Special Requirements

None

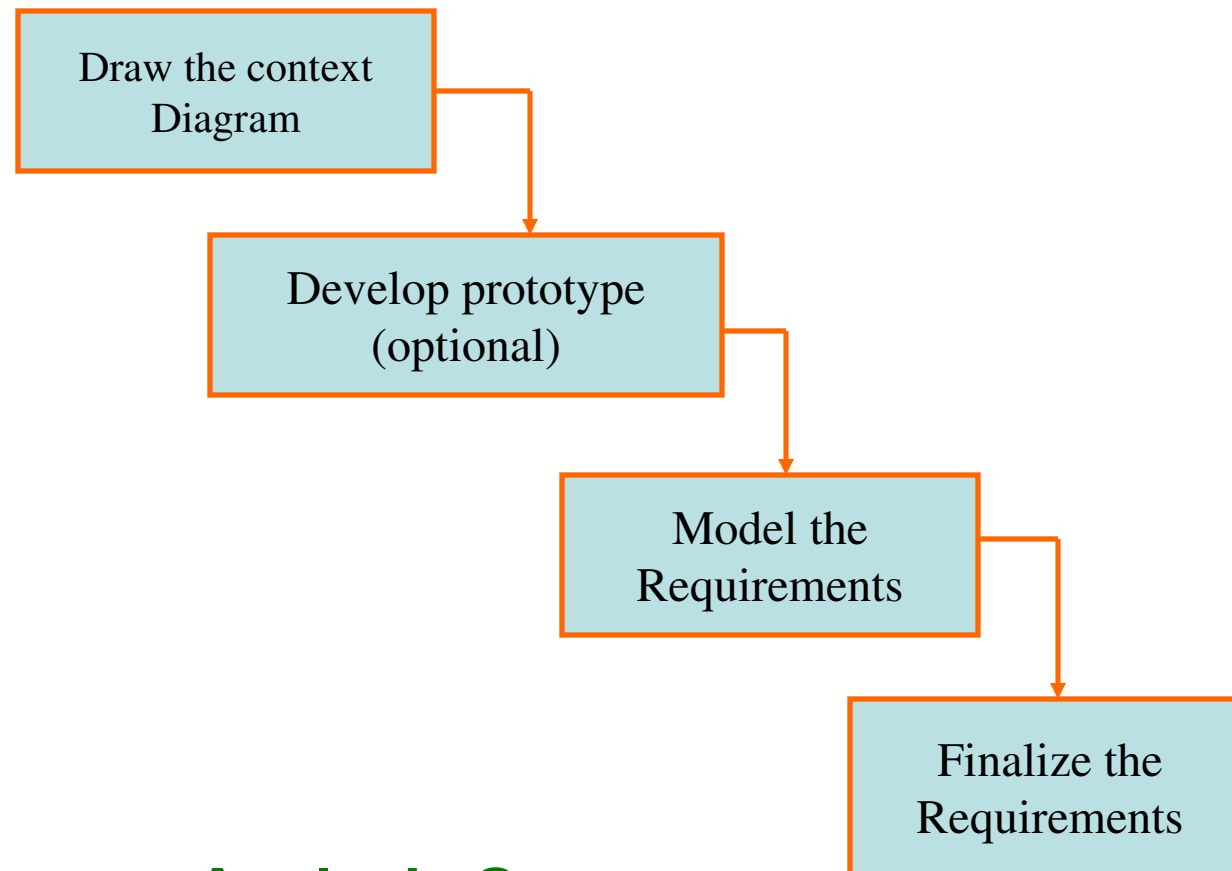
8.8 Use case relationships

None

Requirements Analysis

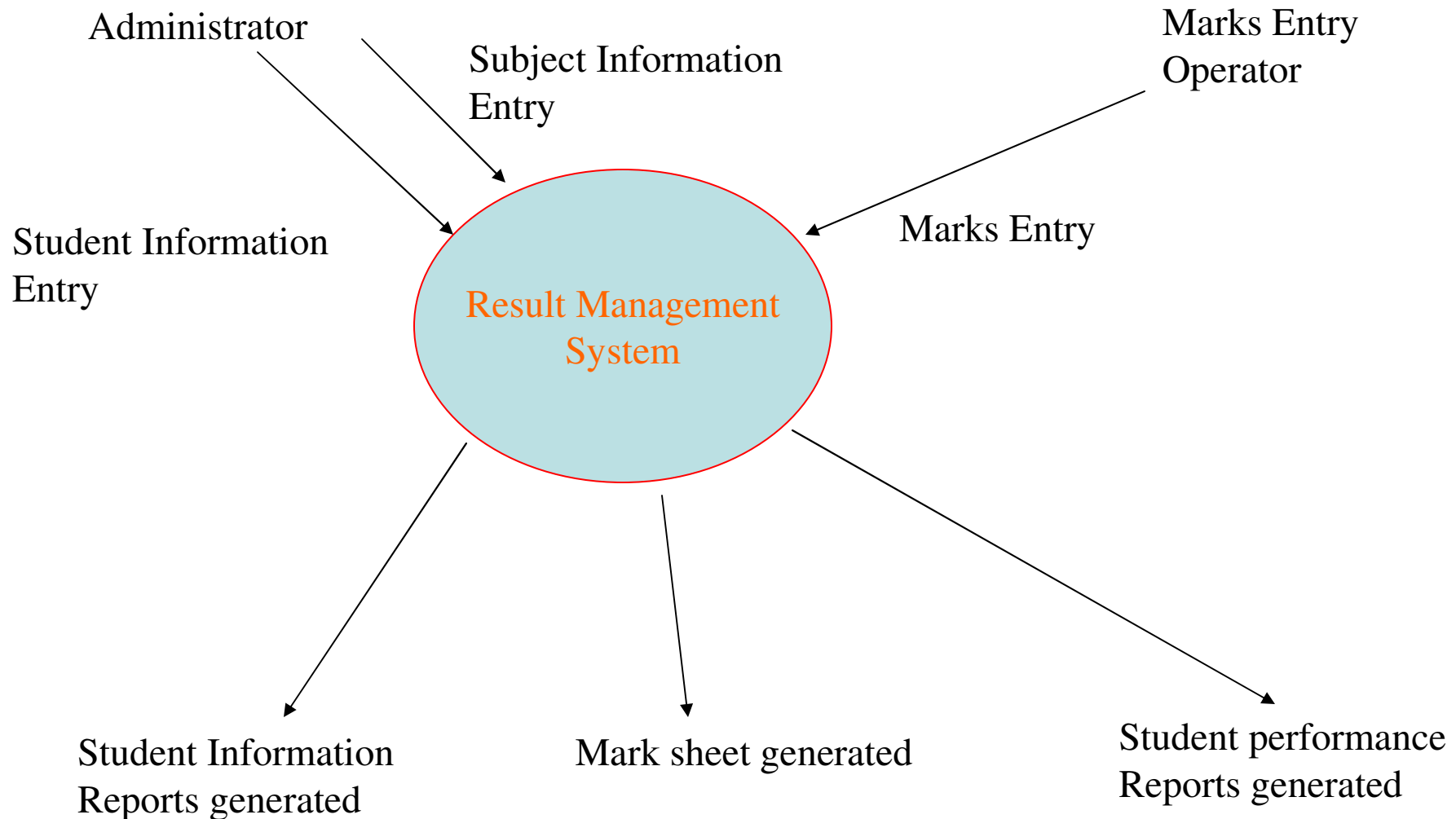
We analyze, refine and scrutinize requirements to make consistent & unambiguous requirements.

Steps



Requirements Analysis Steps

Requirements Analysis

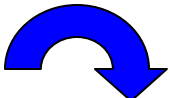
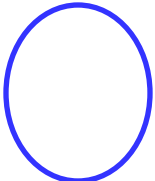


Requirements Analysis


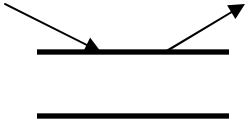
Data Flow Diagrams

DFD show the flow of data through the system.

- All names should be unique
- It is not a flow chart
- Suppress logical decisions
- Defer error conditions & handling until the end of the analysis

Symbol	Name	Function
	Data Flow	Connect process
	Process	Perform some transformation of its input data to yield output data.

Requirements Analysis

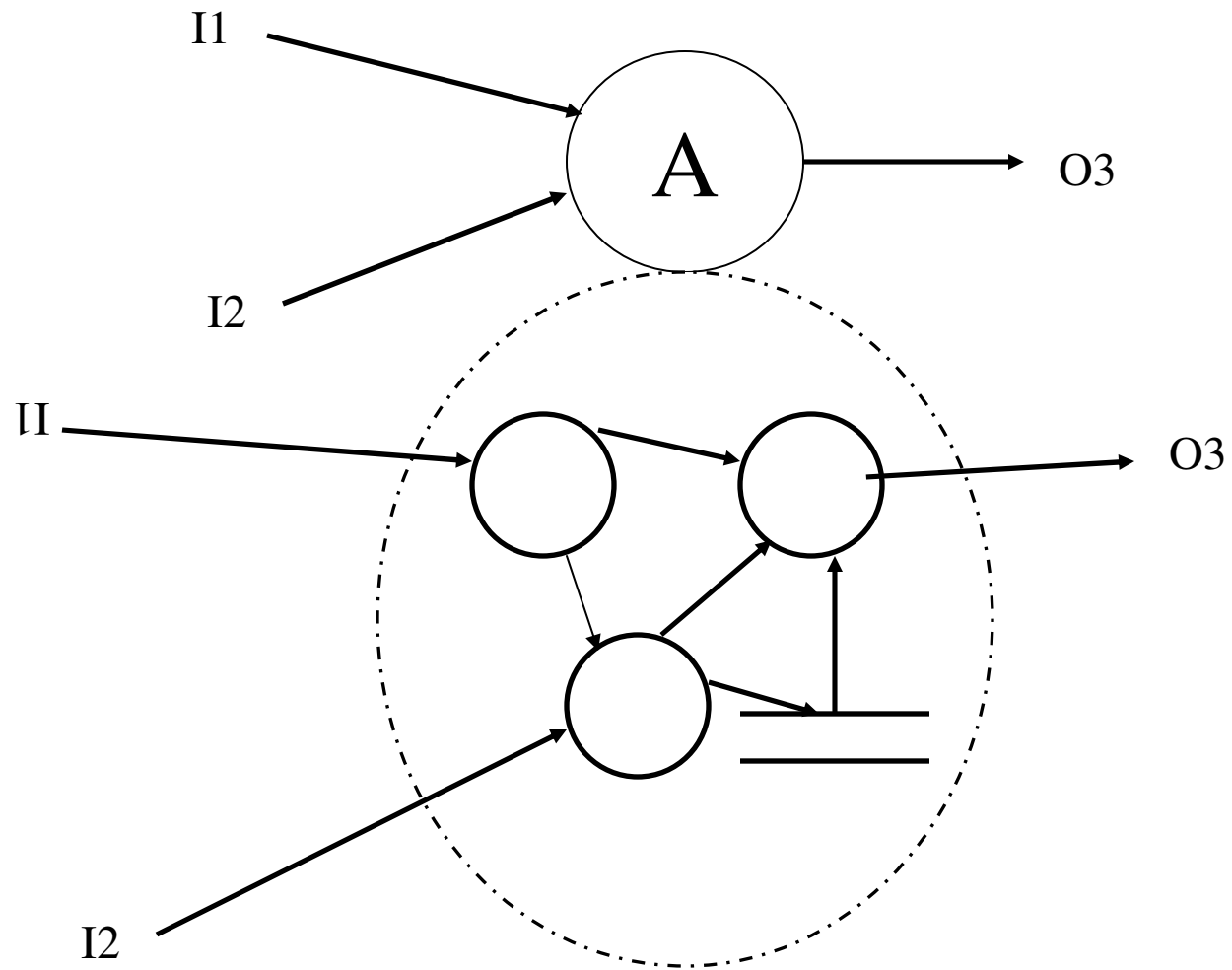
Symbol	Name	Function
	Source or sink	A source of system inputs or sink of system outputs
	Data Store	A repository of data, the arrowhead indicate net input and net outputs to store

Leveling

DFD represent a system or software at any level of abstraction.

A level 0 DFD is called fundamental system model or context model represents entire software element as a single bubble with input and output data indicating by incoming & outgoing arrows.

Requirements Analysis



Data Dictionaries

DFD \longrightarrow DD

Data Dictionaries are simply repositories to store information about all data items defined in DFD.

Includes :

- Name of data item
- Aliases (other names for items)
- Description/Purpose
- Related data items
- Range of values
- Data flows
- Data structure definition

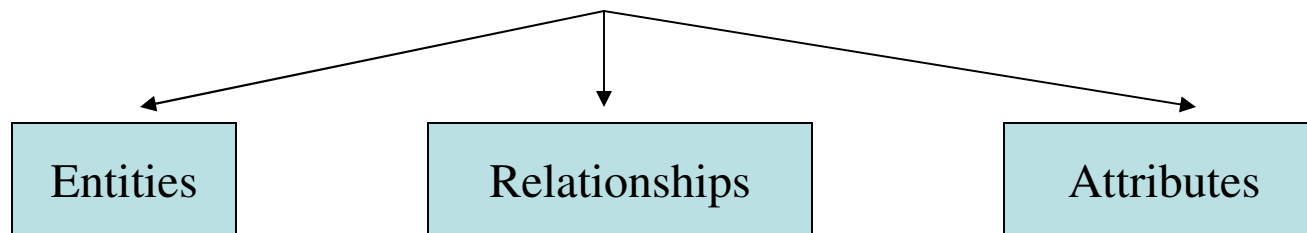
Data Dictionaries

Notation	Meaning
$x = a + b$	x consists of data element a & b
$x = \{a/b\}$	x consists of either a or b
$x = (a)$	x consists of an optional data element a
$x = y\{a\}$	x consists of y or more occurrences
$x = \{a\}z$	x consists of z or fewer occurrences of a
$x = y\{a\}z$	x consists of between y & z occurrences of a

Entity-Relationship Diagrams

Entity-Relationship Diagrams

It is a detailed logical representation of data for an organization and uses three main constructs.



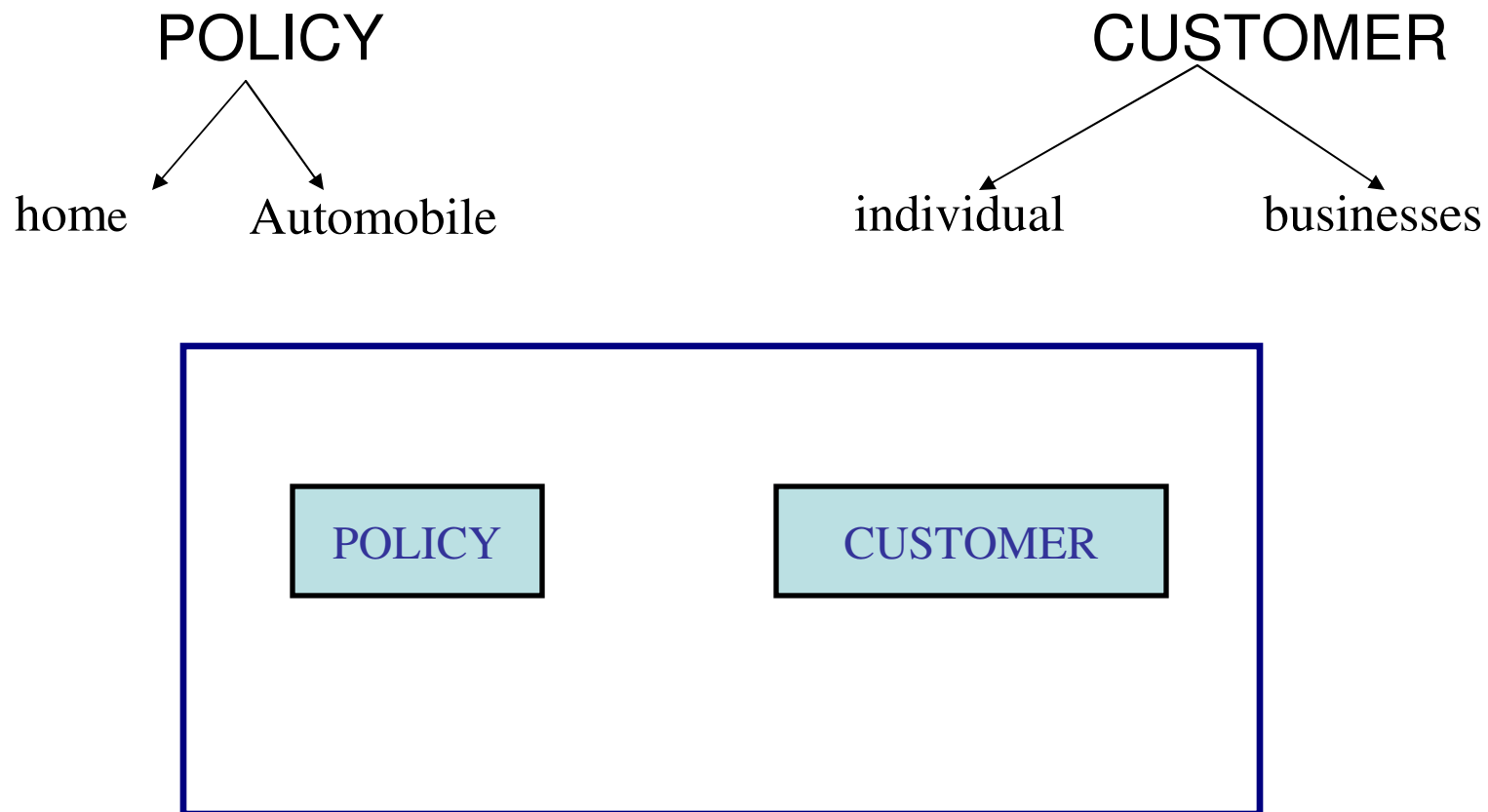
Entities

Fundamental thing about which data may be maintained. Each entity has its own identity.

Entity Type is the description of all entities to which a common definition and common relationships and attributes apply.

Entity-Relationship Diagrams

Consider an insurance company that offers both home and automobile insurance policies .These policies are offered to individuals and businesses.



Entity-Relationship Diagrams

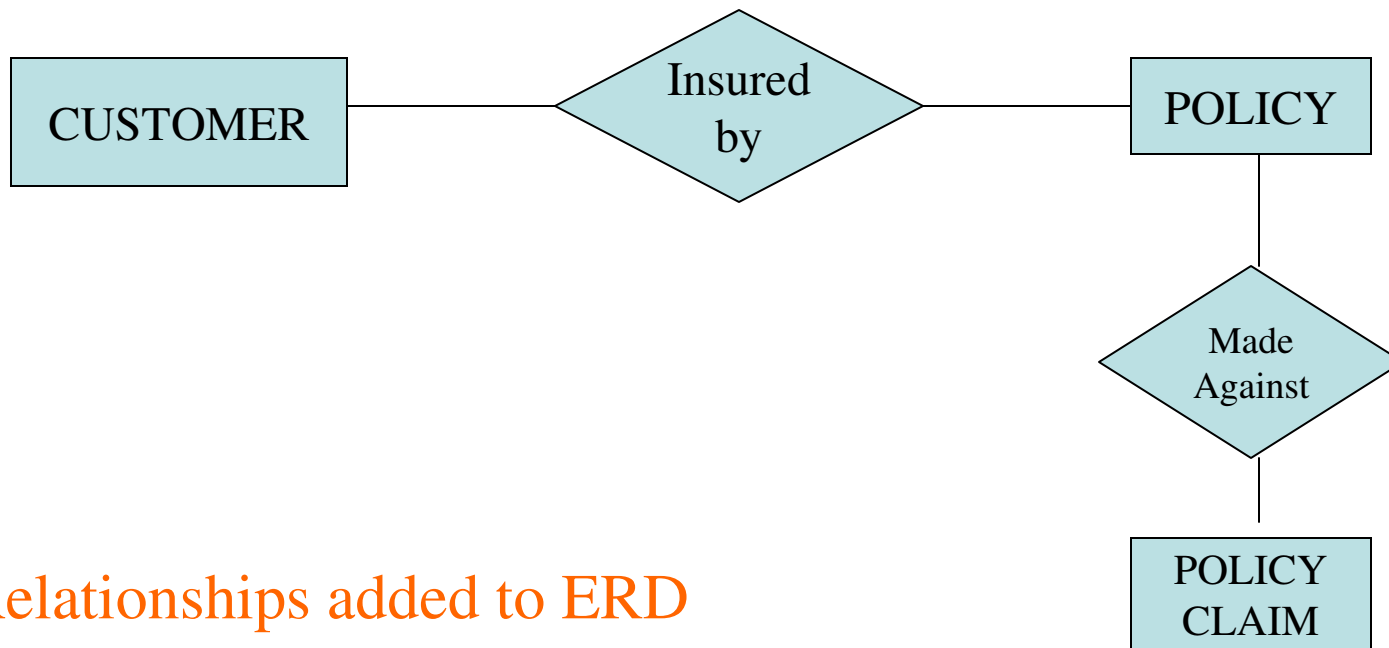
Relationships

A relationship is a reason for associating two entity types.

Binary relationships —————> involve two entity types

A CUSTOMER is insured by a POLICY. A POLICY CLAIM is made against a POLICY.

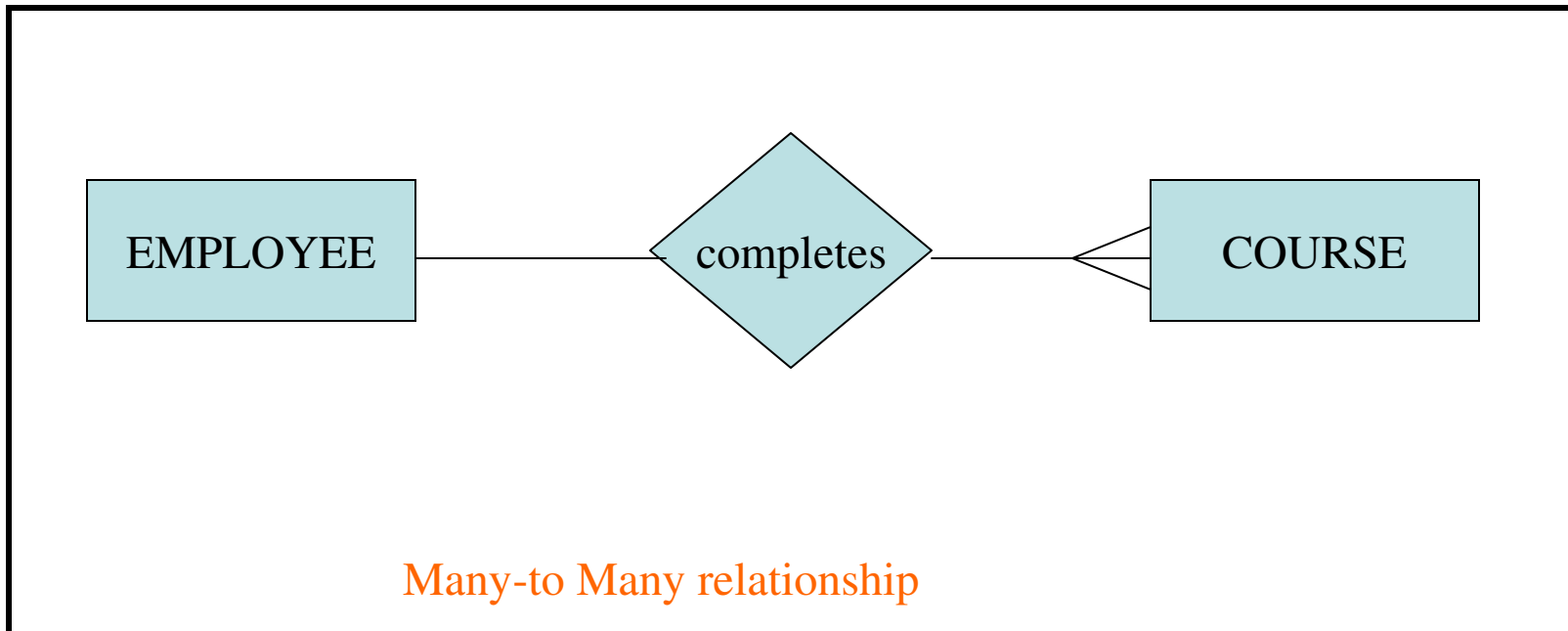
Relationships are represented by diamond notation in a ER diagram.



Relationships added to ERD

Entity-Relationship Diagrams

A training department is interested in tracking which training courses each of its employee has completed.

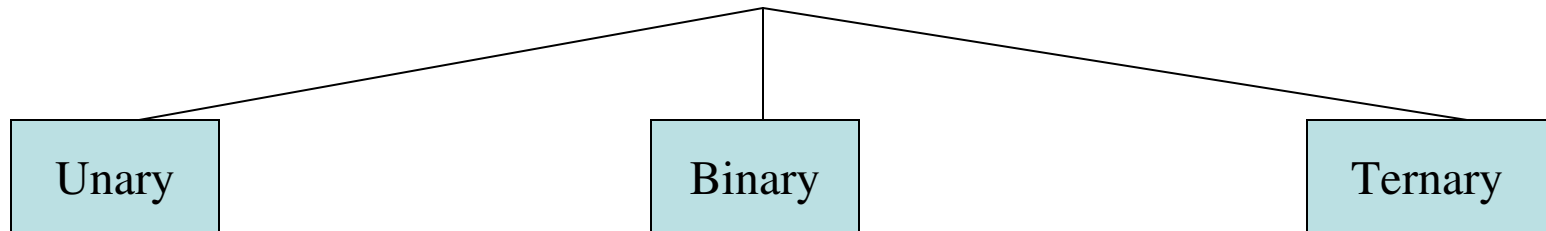


Each employee may complete more than one course, and each course may be completed by more than one employee.

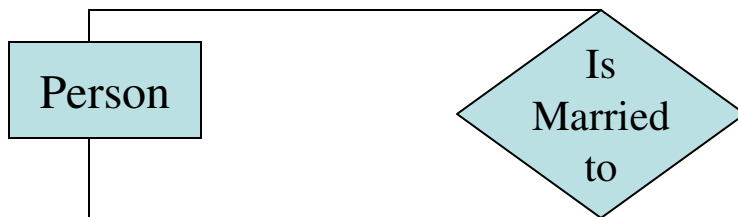
Entity-Relationship Diagrams

Degree of relationship

It is the number of entity types that participates in that relationship.



Unary relationship



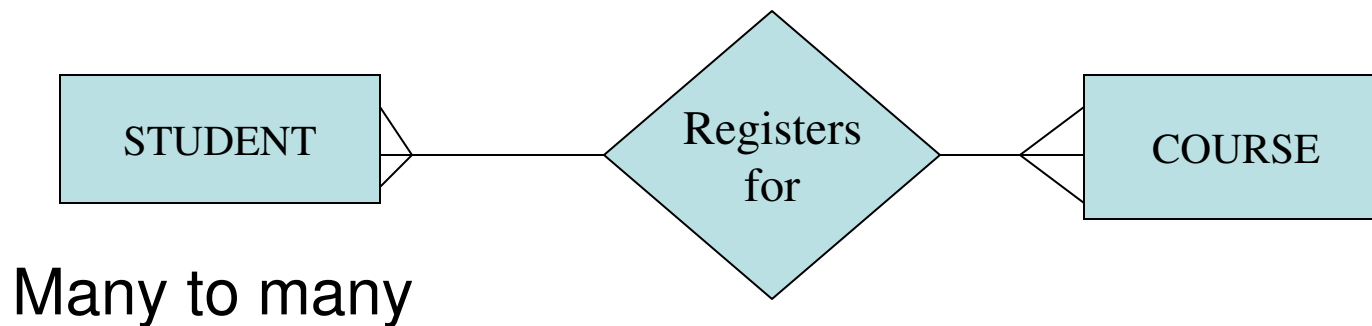
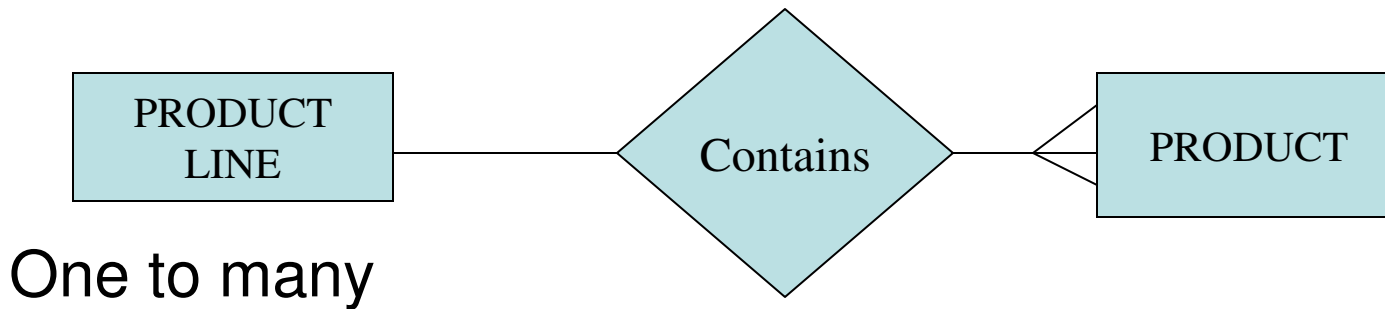
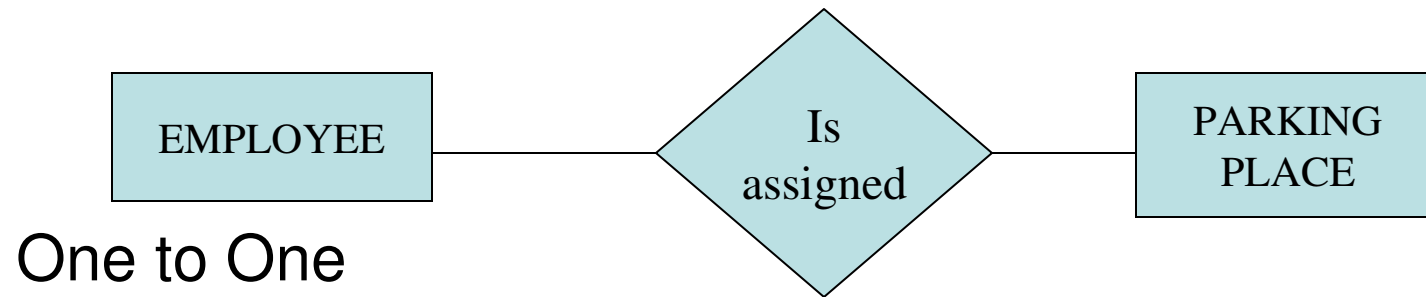
One to One



One to many

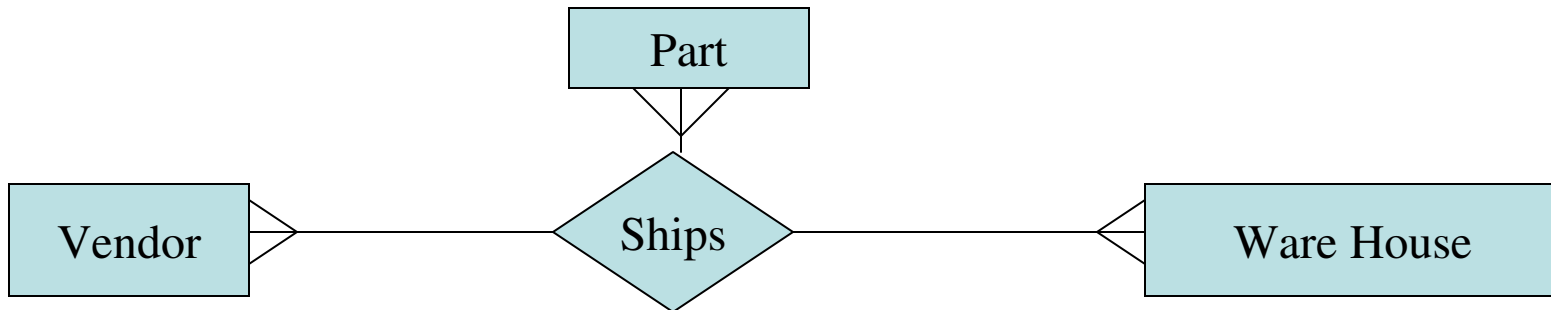
Entity-Relationship Diagrams

Binary Relationship



Entity-Relationship Diagrams

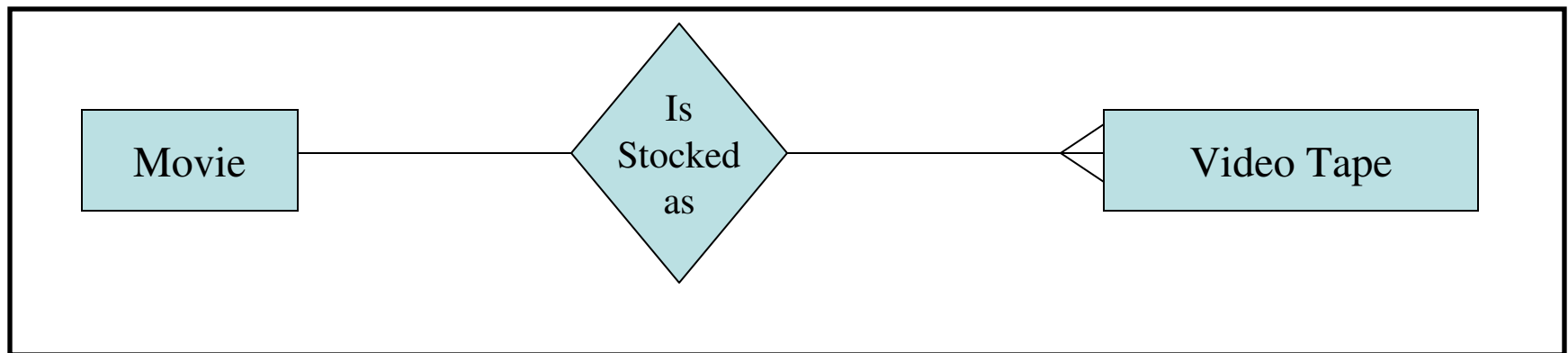
Ternary relationship



Cardinalities and optionality

Two entity types A,B, connected by a relationship.

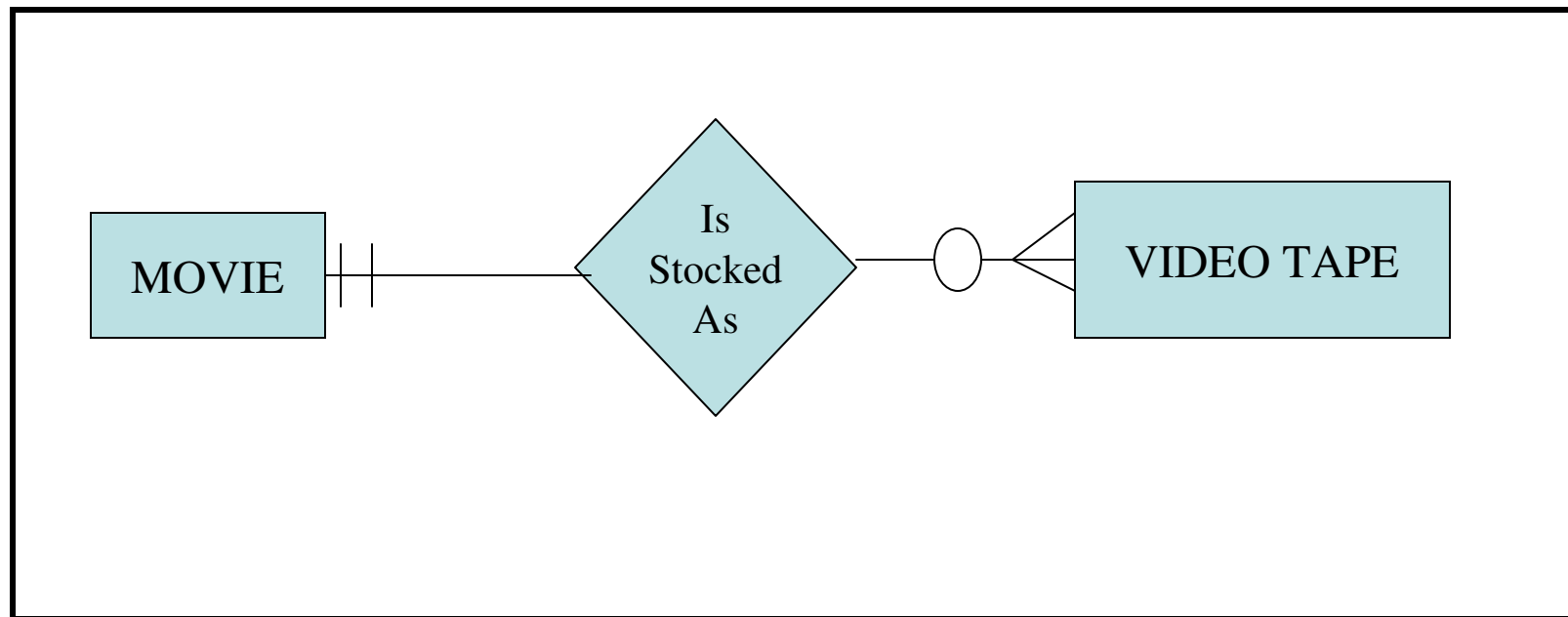
The cardinality of a relationship is the number of instances of entity B that can be associated with each instance of entity A



Entity-Relationship Diagrams

Minimum cardinality is the minimum number of instances of entity B that may be associated with each instance of entity A.

Minimum no. of tapes available for a movie is zero. We say VIDEO TAPE is an optional participant in the is-stocked-as relationship.

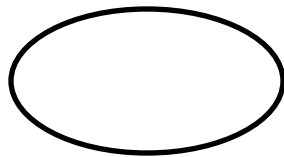


Entity-Relationship Diagrams

Attributes

Each entity type has a set of attributes associated with it.

An attribute is a property or characteristic of an entity that is of interest to organization.



Attribute

Entity-Relationship Diagrams

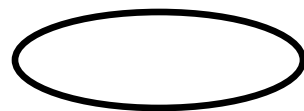
A candidate key is an attribute or combination of attributes that uniquely identifies each instance of an entity type.

Student_ID \longrightarrow Candidate Key

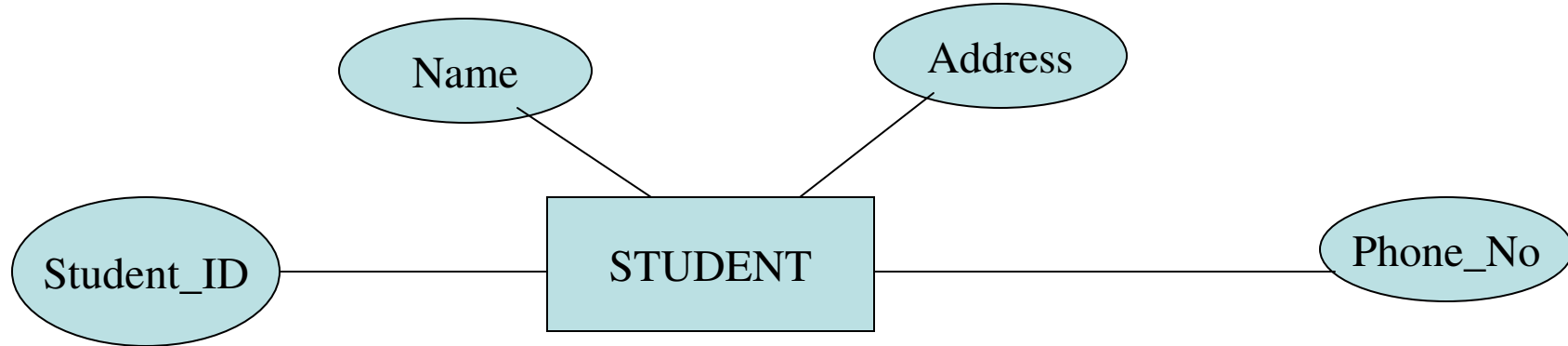
If there are more candidate keys, one of the key may be chosen as the Identifier.

It is used as unique characteristic for an entity type.

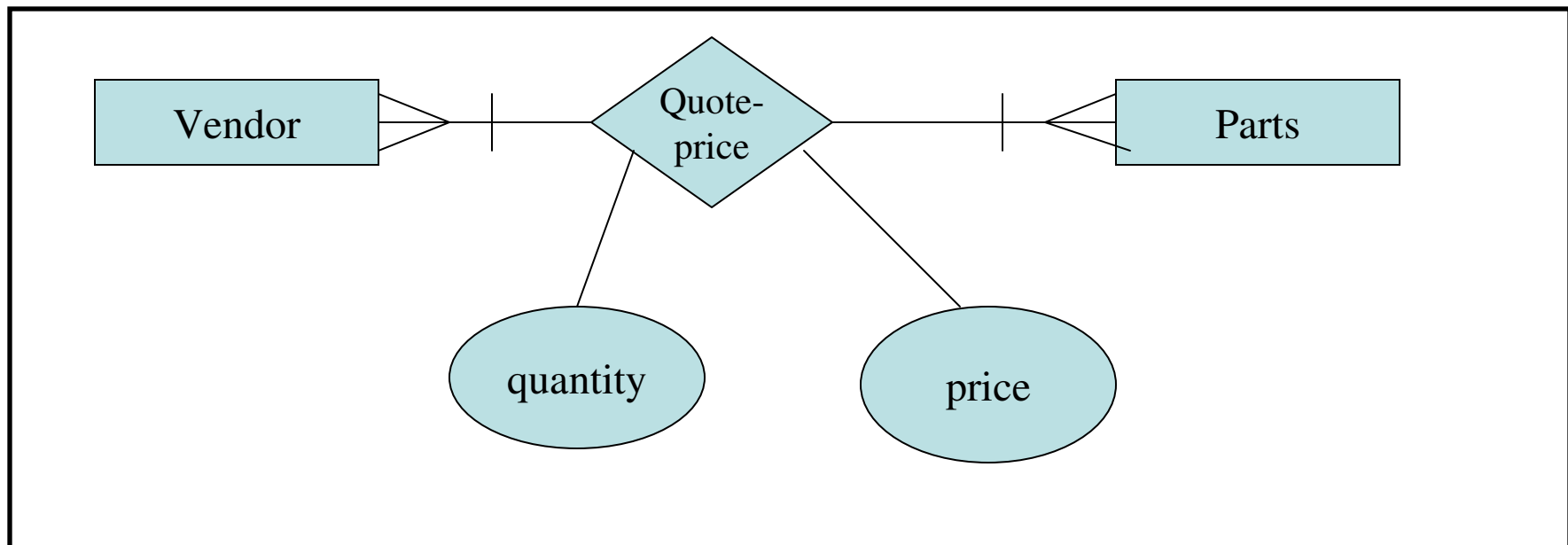
Identifier



Entity-Relationship Diagrams



Vendors quote prices for several parts along with quantity of parts.
Draw an E-R diagram.



Approaches to problem analysis

1. List all inputs, outputs and functions.
2. List all functions and then list all inputs and outputs associated with each function.

Structured requirements definition (SRD)

Step1

Define a user level DFD. Record the inputs and outputs for each individual in a DFD.

Step2

Define a combined user level DFD.

Step3

Define application level DFD.

Step4

Define application level functions.

Requirements Documentation

This is the way of representing requirements in a consistent format

SRS serves many purpose depending upon who is writing it.

- written by customer
 - written by developer
- 

Serves as contract between customer & developer.

Requirements Documentation

Nature of SRS

Basic Issues

- Functionality
- External Interfaces
- Performance
- Attributes
- Design constraints imposed on an Implementation

Requirements Documentation

SRS Should

- Correctly define all requirements
- not describe any design details
- not impose any additional constraints

Characteristics of a good SRS

An SRS Should be

- ✓ **Correct**
- ✓ **Unambiguous**
- ✓ **Complete**
- ✓ **Consistent**

Requirements Documentation

- ✓ Ranked for important and/or stability
- ✓ Verifiable
- ✓ Modifiable
- ✓ Traceable

Requirements Documentation

Correct

An SRS is correct if and only if every requirement stated therein is one that the software shall meet.

Unambiguous

An SRS is unambiguous if and only if, every requirement stated therein has only one interpretation.

Complete

An SRS is complete if and only if, it includes the following elements

- (i) All significant requirements, whether related to functionality, performance, design constraints, attributes or external interfaces.

Requirements Documentation

- (ii) Responses to both valid & invalid inputs.
- (iii) Full Label and references to all figures, tables and diagrams in the SRS and definition of all terms and units of measure.

Consistent

An SRS is consistent if and only if, no subset of individual requirements described in it conflict.

Ranked for importance and/or Stability

If an identifier is attached to every requirement to indicate either the importance or stability of that particular requirement.

Requirements Documentation

Verifiable

An SRS is verifiable, if and only if, every requirement stated therein is verifiable.

Modifiable

An SRS is modifiable, if and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining structure and style.

Traceable

An SRS is traceable, if the origin of each of the requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.

Requirements Documentation

Organization of the SRS

IEEE has published guidelines and standards to organize an SRS.

First two sections are same. The specific tailoring occurs in section-3.

1. Introduction

- 1.1 Purpose
- 1.2 Scope
- 1.3 Definition, Acronyms and abbreviations
- 1.4 References
- 1.5 Overview

Requirements Documentation

2. The Overall Description

2.1 Product Perspective

2.1.1 System Interfaces

2.1.2 Interfaces

2.1.3 Hardware Interfaces

2.1.4 Software Interfaces

2.1.5 Communication Interfaces

2.1.6 Memory Constraints

2.1.7 Operations

2.1.8 Site Adaptation Requirements

Requirements Documentation

- 2.2 Product Functions
- 2.3 User Characteristics
- 2.4 Constraints
- 2.5 Assumptions for dependencies
- 2.6 Apportioning of requirements

3. Specific Requirements

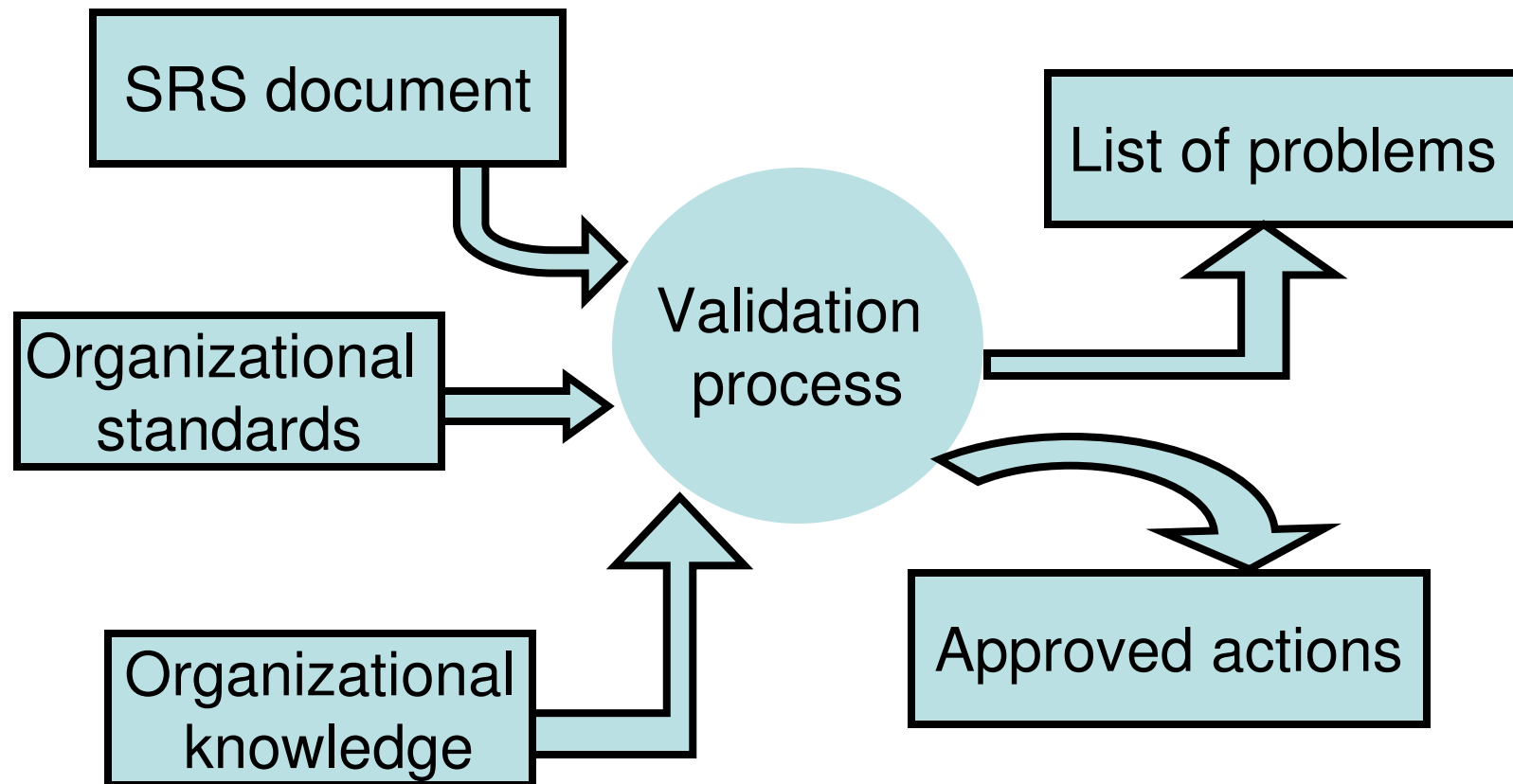
- 3.1 External Interfaces
- 3.2 Functions
- 3.3 Performance requirements
- 3.4 Logical database requirements
- 3.5 Design Constraints
- 3.6 Software System attributes
- 3.7 Organization of specific requirements
- 3.8 Additional Comments.

Requirements Validation

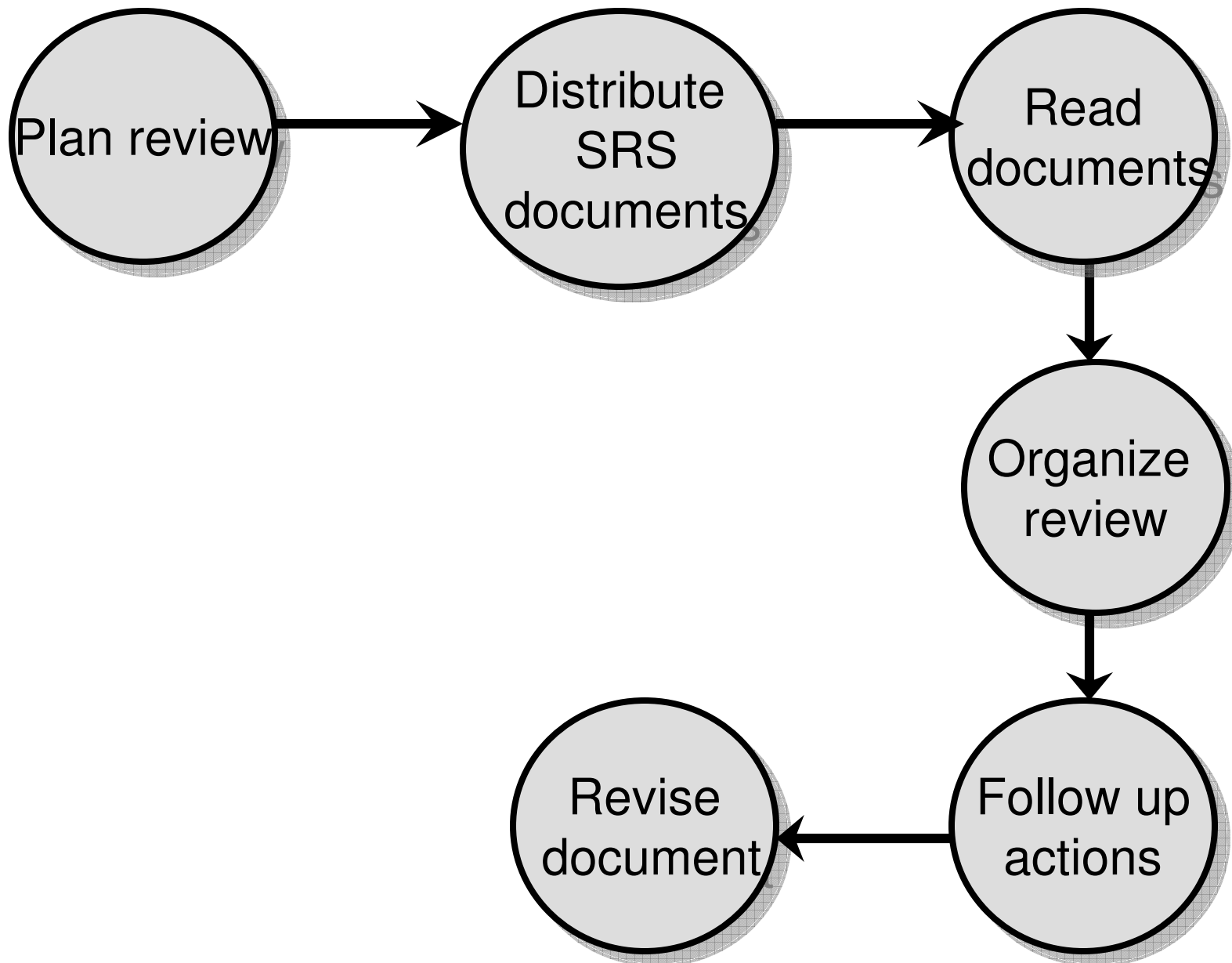
Check the document for:

- ✓ Completeness & consistency
- ✓ Conformance to standards
- ✓ Requirements conflicts
- ✓ Technical errors
- ✓ Ambiguous requirements

Requirements Validation



Requirements Review Process



Requirements Validation

Problem actions

- Requirements clarification
- Missing information
 - find this information from stakeholders
- Requirements conflicts
 - Stakeholders must negotiate to resolve this conflict
- Unrealistic requirements
 - Stakeholders must be consulted
- Security issues
 - Review the system in accordance to security standards

Review Checklists

- ✓ Redundancy
- ✓ Completeness
- ✓ Ambiguity
- ✓ Consistency
- ✓ Organization
- ✓ Conformance
- ✓ Traceability

Prototyping

Validation prototype should be reasonably complete & efficient & should be used as the required system.

Requirements Management

- Process of understanding and controlling changes to system requirements.

ENDURING & VOLATILE REQUIREMENTS

- o Enduring requirements: They are core requirements & are related to main activity of the organization.

Example: issue/return of a book, cataloging etc.

- o Volatile requirements: likely to change during software development life cycle or after delivery of the product

Requirements Management Planning

- Very critical.
- Important for the success of any project.

Requirements Change Management

- Allocating adequate resources
- Analysis of requirements
- Documenting requirements
- Requirements traceability
- Establishing team communication
- Establishment of baseline

Download from

Q:\IRM\PRIVATE\INITIATIAT\QA\QAPLAN\SRSPLAN.DOC

Software Project Planning



Software Project Planning

After the finalization of SRS, we would like to estimate size, cost and development time of the project. Also, in many cases, customer may like to know the cost and development time even prior to finalization of the SRS.

Software Project Planning

In order to conduct a successful software project, we must understand:

- Scope of work to be done
- The risk to be incurred
- The resources required
- The task to be accomplished
- The cost to be expended
- The schedule to be followed

Software Project Planning

Software planning begins before technical work starts, continues as the software evolves from concept to reality, and culminates only when the software is retired.

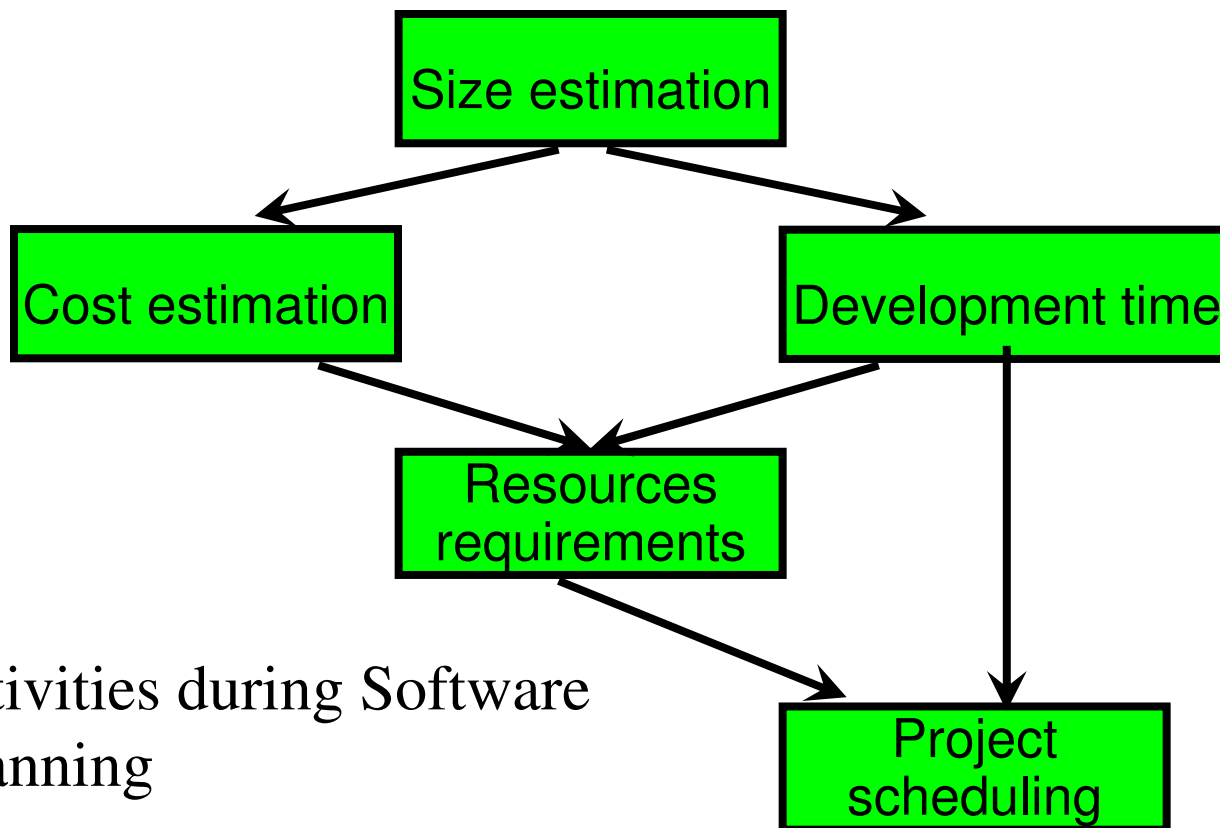


Fig. 1: Activities during Software Project Planning

Software Project Planning

Size Estimation

Lines of Code (LOC)

If LOC is simply a count of the number of lines then figure shown below contains 18 LOC .

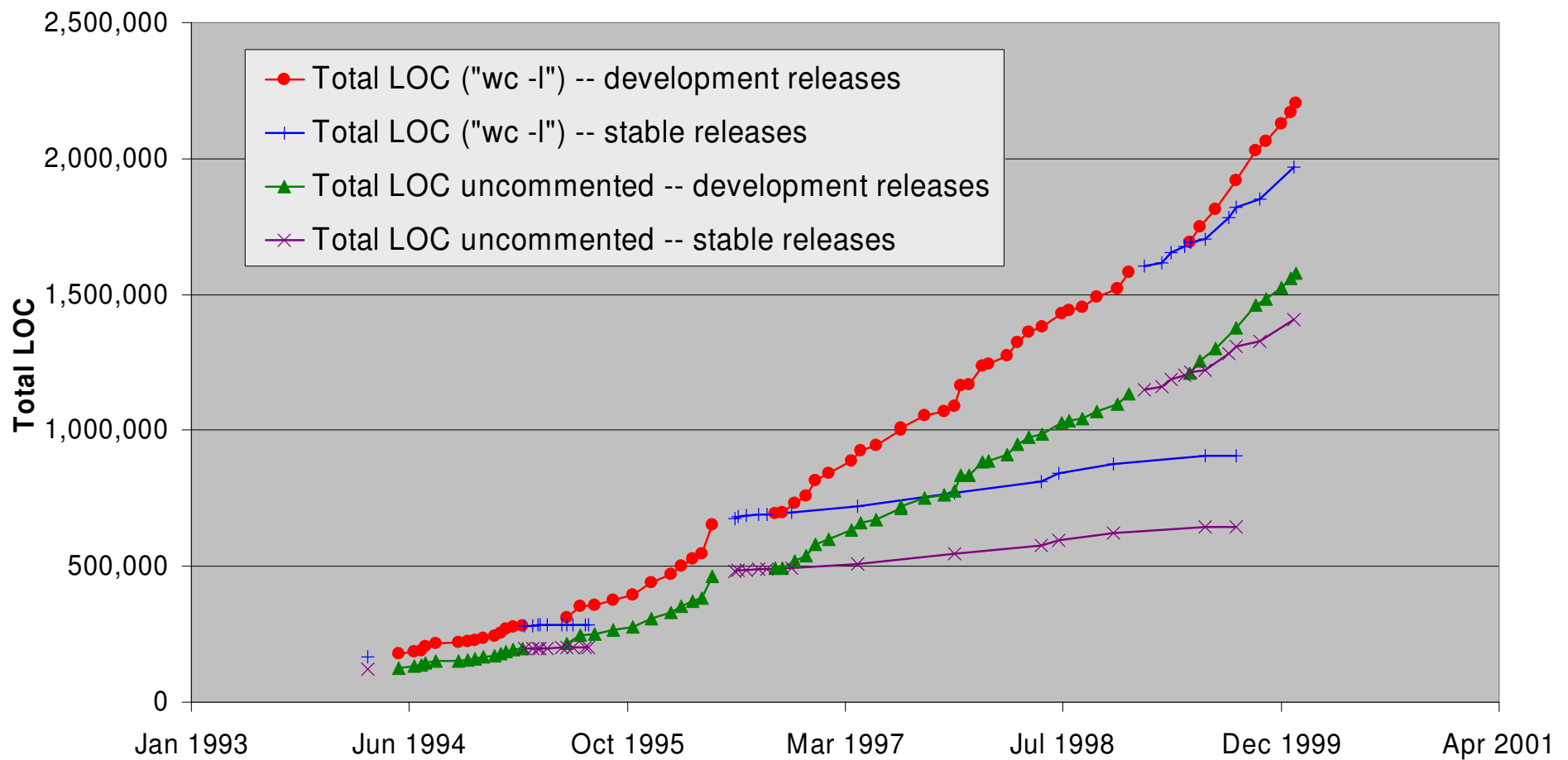
When comments and blank lines are ignored, the program in figure 2 shown below contains 17 LOC.

Fig. 2: Function for sorting an array

1.	int. sort (int x[], int n)
2.	{
3.	int i, j, save, im1;
4.	/*This function sorts array x in ascending order */
5.	If (n<2) return 1;
6.	for (i=2; i<=n; i++)
7.	{
8.	im1=i-1;
9.	for (j=1; j<=im; j++)
10.	if (x[i] < x[j])
11.	{
12.	Save = x[i];
13.	x[i] = x[j];
14.	x[j] = save;
15.	}
16.	}
17.	return 0;
18.	}

Software Project Planning

Growth of Lines of Code (LOC)



Software Project Planning

Furthermore, if the main interest is the size of the program for specific functionality, it may be reasonable to include executable statements. The only executable statements in figure shown above are in lines 5-17 leading to a count of 13. The differences in the counts are 18 to 17 to 13. One can easily see the potential for major discrepancies for large programs with many comments or programs written in language that allow a large number of descriptive but non-executable statement. Conte has defined lines of code as:

Software Project Planning

“A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program header, declaration, and executable and non-executable statements”.

This is the predominant definition for lines of code used by researchers. By this definition, figure shown above has 17 LOC.

Software Project Planning

Function Count

Alan Albrecht while working for IBM, recognized the problem in size measurement in the 1970s, and developed a technique (which he called Function Point Analysis), which appeared to be a solution to the size measurement problem.

Software Project Planning

The principle of Albrecht's function point analysis (FPA) is that a system is decomposed into functional units.

- Inputs : information entering the system
- Outputs : information leaving the system
- Enquiries : requests for instant access to information
- Internal logical files : information held within the system
- External interface files : information held by other system that is used by the system being analyzed.

Software Project Planning

The FPA functional units are shown in figure given below:

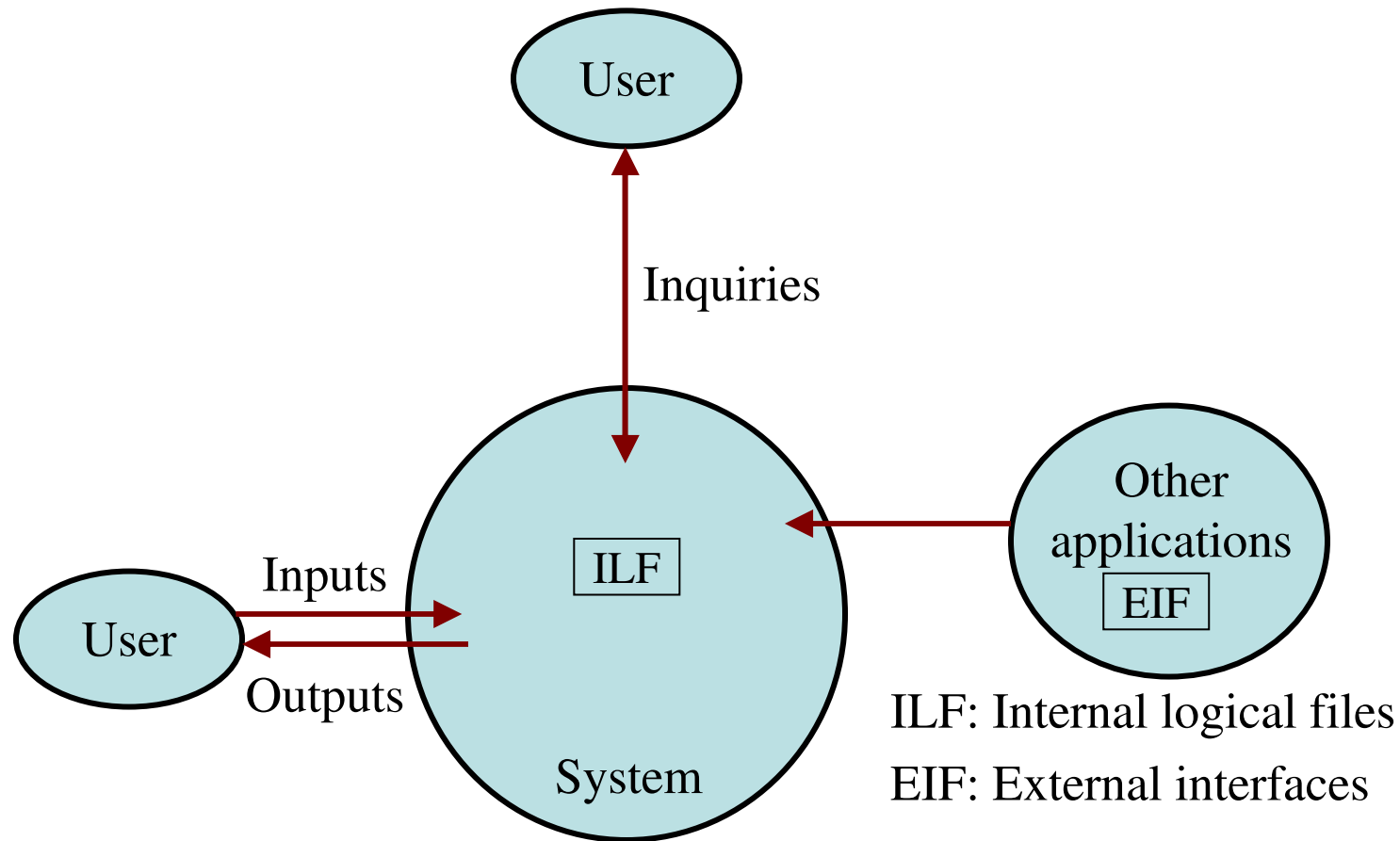


Fig. 3: FPAs functional units System

Software Project Planning

The five functional units are divided in two categories:

(i) Data function types

- **Internal Logical Files (ILF):** A user identifiable group of logical related data or control information maintained within the system.
- **External Interface files (EIF):** A user identifiable group of logically related data or control information referenced by the system, but maintained within another system. This means that EIF counted for one system, may be an ILF in another system.

Software Project Planning

(ii) Transactional function types

- External Input (EI): An EI processes data or control information that comes from outside the system. The EI is an elementary process, which is the smallest unit of activity that is meaningful to the end user in the business.
- External Output (EO): An EO is an elementary process that generate data or control information to be sent outside the system.
- External Inquiry (EQ): An EQ is an elementary process that is made up to an input-output combination that results in data retrieval.

Software Project Planning

Special features

- Function point approach is independent of the language, tools, or methodologies used for implementation; i.e. they do not take into consideration programming languages, data base management systems, processing hardware or any other data base technology.
- Function points can be estimated from requirement specification or design specification, thus making it possible to estimate development efforts in early phases of development.

Software Project Planning

- Function points are directly linked to the statement of requirements; any change of requirements can easily be followed by a re-estimate.
- Function points are based on the system user's external view of the system, non-technical users of the software system have a better understanding of what function points are measuring.

Software Project Planning

Counting function points

Functional Units	Weighting factors		
	Low	Average	High
External Inputs (EI)	3	4	6
External Output (EO)	4	5	7
External Inquiries (EQ)	3	4	6
External logical files (ILF)	7	10	15
External Interface files (EIF)	5	7	10

Table 1 : Functional units with weighting factors

Software Project Planning

Table 2: UFP calculation table

Functional Units	Count	Complexity	Complexity Totals	Functional Unit Totals
External Inputs (EIs)	<input type="text"/>	Low x 3	= <input type="text"/>	<input type="text"/>
	<input type="text"/>	Average x 4	= <input type="text"/>	
	<input type="text"/>	High x 6	= <input type="text"/>	
External Outputs (EOs)	<input type="text"/>	Low x 4	= <input type="text"/>	<input type="text"/>
	<input type="text"/>	Average x 5	= <input type="text"/>	
	<input type="text"/>	High x 7	= <input type="text"/>	
External Inquiries (EQs)	<input type="text"/>	Low x 3	= <input type="text"/>	<input type="text"/>
	<input type="text"/>	Average x 4	= <input type="text"/>	
	<input type="text"/>	High x 6	= <input type="text"/>	
External logical Files (ILFs)	<input type="text"/>	Low x 7	= <input type="text"/>	<input type="text"/>
	<input type="text"/>	Average x 10	= <input type="text"/>	
	<input type="text"/>	High x 15	= <input type="text"/>	
External Interface Files (EIFs)	<input type="text"/>	Low x 5	= <input type="text"/>	<input type="text"/>
	<input type="text"/>	Average x 7	= <input type="text"/>	
	<input type="text"/>	High x 10	= <input type="text"/>	
Total Unadjusted Function Point Count				<input type="text"/>

Software Project Planning

The weighting factors are identified for all functional units and multiplied with the functional units accordingly. The procedure for the calculation of Unadjusted Function Point (UFP) is given in table shown above.

Software Project Planning

The procedure for the calculation of UFP in mathematical form is given below:

$$UFP = \sum_{i=1}^5 \sum_{J=1}^3 Z_{ij} w_{ij}$$

Where i indicate the row and j indicates the column of Table 1

w_{ij} : It is the entry of the i^{th} row and j^{th} column of the table 1

Z_{ij} : It is the count of the number of functional units of Type i that have been classified as having the complexity corresponding to column j .

Software Project Planning

Organizations that use function point methods develop a criterion for determining whether a particular entry is Low, Average or High. Nonetheless, the determination of complexity is somewhat subjective.

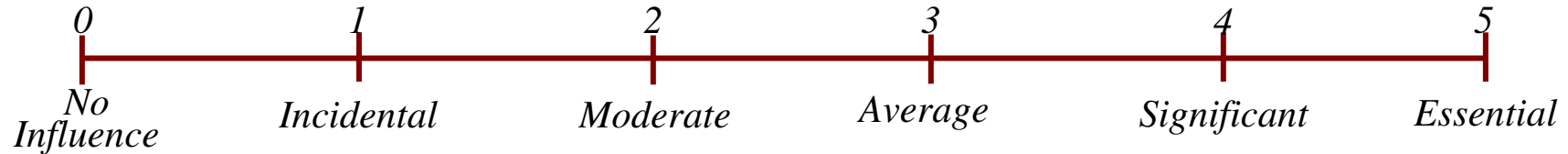
$$FP = UFP * CAF$$

Where CAF is complexity adjustment factor and is equal to $[0.65 + 0.01 \times \sum F_i]$. The F_i ($i=1$ to 14) are the degree of influence and are based on responses to questions noted in table 3.

Software Project Planning

Table 3 : Computing function points.

Rate each factor on a scale of 0 to 5.



Number of factors considered (F_i)

1. Does the system require reliable backup and recovery ?
2. Is data communication required ?
3. Are there distributed processing functions ?
4. Is performance critical ?
5. Will the system run in an existing heavily utilized operational environment ?
6. Does the system require on line data entry ?
7. Does the on line data entry require the input transaction to be built over multiple screens or operations ?
8. Are the master files updated on line ?
9. Is the inputs, outputs, files, or inquiries complex ?
10. Is the internal processing complex ?
11. Is the code designed to be reusable ?
12. Are conversion and installation included in the design ?
13. Is the system designed for multiple installations in different organizations ?
14. Is the application designed to facilitate change and ease of use by the user ?

Software Project Planning

Functions points may compute the following important metrics:

Productivity = FP / persons-months

Quality = Defects / FP

Cost = Rupees / FP

Documentation = Pages of documentation per FP

These metrics are controversial and are not universally acceptable. There are standards issued by the International Functions Point User Group (IFPUG, covering the Albrecht method) and the United Kingdom Function Point User Group (UFGU, covering the MK11 method). An ISO standard for function point method is also being developed.

Software Project Planning

Example: 4.1

Consider a project with the following functional units:

Number of user inputs = 50

Number of user outputs = 40

Number of user enquiries = 35

Number of user files = 06

Number of external interfaces = 04

Assume all complexity adjustment factors and weighting factors are average. Compute the function points for the project.

Software Project Planning

Solution

We know

$$UFP = \sum_{i=1}^5 \sum_{j=1}^3 Z_{ij} w_{ij}$$

$$\begin{aligned} UFP &= 50 \times 4 + 40 \times 5 + 35 \times 4 + 6 \times 10 + 4 \times 7 \\ &= 200 + 200 + 140 + 60 + 28 = 628 \end{aligned}$$

$$\begin{aligned} CAF &= (0.65 + 0.01 \sum F_i) \\ &= (0.65 + 0.01 (14 \times 3)) = 0.65 + 0.42 = 1.07 \end{aligned}$$

$$\begin{aligned} FP &= UFP \times CAF \\ &= 628 \times 1.07 = 672 \end{aligned}$$

Software Project Planning

Example:4.2

An application has the following:

10 low external inputs, 12 high external outputs, 20 low internal logical files, 15 high external interface files, 12 average external inquiries, and a value of complexity adjustment factor of 1.10.

What are the unadjusted and adjusted function point counts ?

Software Project Planning

Solution

Unadjusted function point counts may be calculated using as:

$$\begin{aligned} UFP &= \sum_{i=1}^5 \sum_{J=1}^3 Z_{ij} w_{ij} \\ &= 10 \times 3 + 12 \times 7 + 20 \times 7 + 15 + 10 + 12 \times 4 \\ &= 30 + 84 + 140 + 150 + 48 \\ &= 452 \\ \text{FP} &= \text{UFP} \times \text{CAF} \\ &= 452 \times 1.10 = 497.2. \end{aligned}$$

Software Project Planning

Example: 4.3

Consider a project with the following parameters.

- (i) External Inputs:
 - (a) 10 with low complexity
 - (b) 15 with average complexity
 - (c) 17 with high complexity
- (ii) External Outputs:
 - (a) 6 with low complexity
 - (b) 13 with high complexity
- (iii) External Inquiries:
 - (a) 3 with low complexity
 - (b) 4 with average complexity
 - (c) 2 high complexity

Software Project Planning

- (iv) Internal logical files:
 - (a) 2 with average complexity
 - (b) 1 with high complexity
- (v) External Interface files:
 - (a) 9 with low complexity

In addition to above, system requires

- i. Significant data communication
- ii. Performance is very critical
- iii. Designed code may be moderately reusable
- iv. System is not designed for multiple installation in different organizations.

Other complexity adjustment factors are treated as average. Compute the function points for the project.

Software Project Planning

Solution: Unadjusted function points may be counted using table 2

Functional Units	Count	Complexity		Complexity Totals	Functional Unit Totals
External Inputs (EIs)	10	Low x 3	=	30	192
	15	Average x 4	=	60	
	17	High x 6	=	102	
External Outputs (EOs)	6	Low x 4	=	24	115
	0	Average x 5	=	0	
	13	High x 7	=	91	
External Inquiries (EQs)	3	Low x 3	=	9	37
	4	Average x 4	=	16	
	2	High x 6	=	12	
External logical Files (ILFs)	0	Low x 7	=	0	35
	2	Average x 10	=	20	
	1	High x 15	=	15	
External Interface Files (EIFs)	9	Low x 5	=	45	45
	0	Average x 7	=	0	
	0	High x 10	=	0	
Total Unadjusted Function Point Count					424

Software Project Planning

$$\sum_{i=1}^{14} F_i = 3+4+3+5+3+3+3+3+3+3+2+3+0+3=41$$

$$\begin{aligned}\text{CAF} &= (0.65 + 0.01 \times \sum F_i) \\ &= (0.65 + 0.01 \times 41) \\ &= 1.06\end{aligned}$$

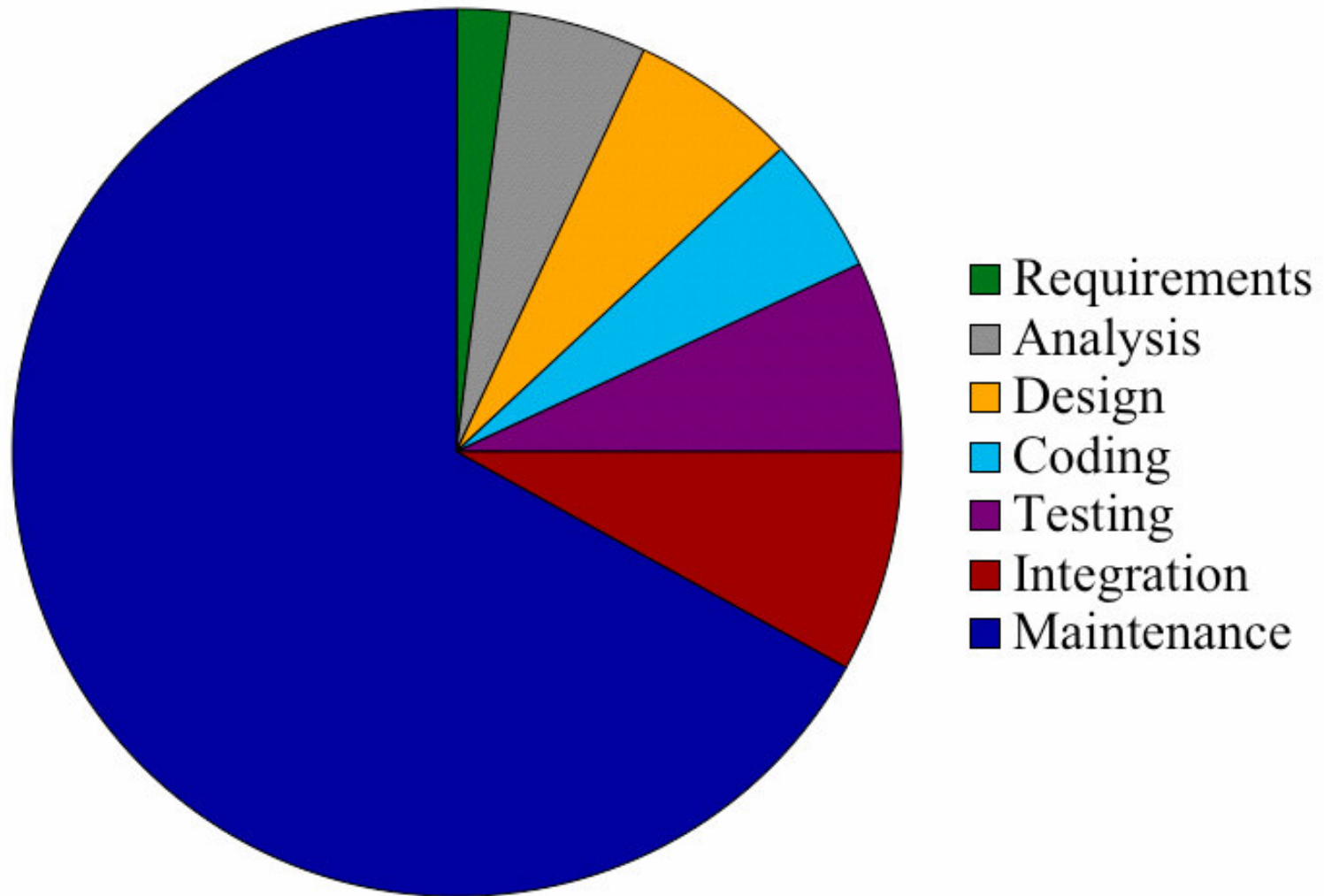
$$\begin{aligned}\text{FP} &= \text{UFP} \times \text{CAF} \\ &= 424 \times 1.06 \\ &= 449.44\end{aligned}$$

Hence

$\text{FP} = 449$

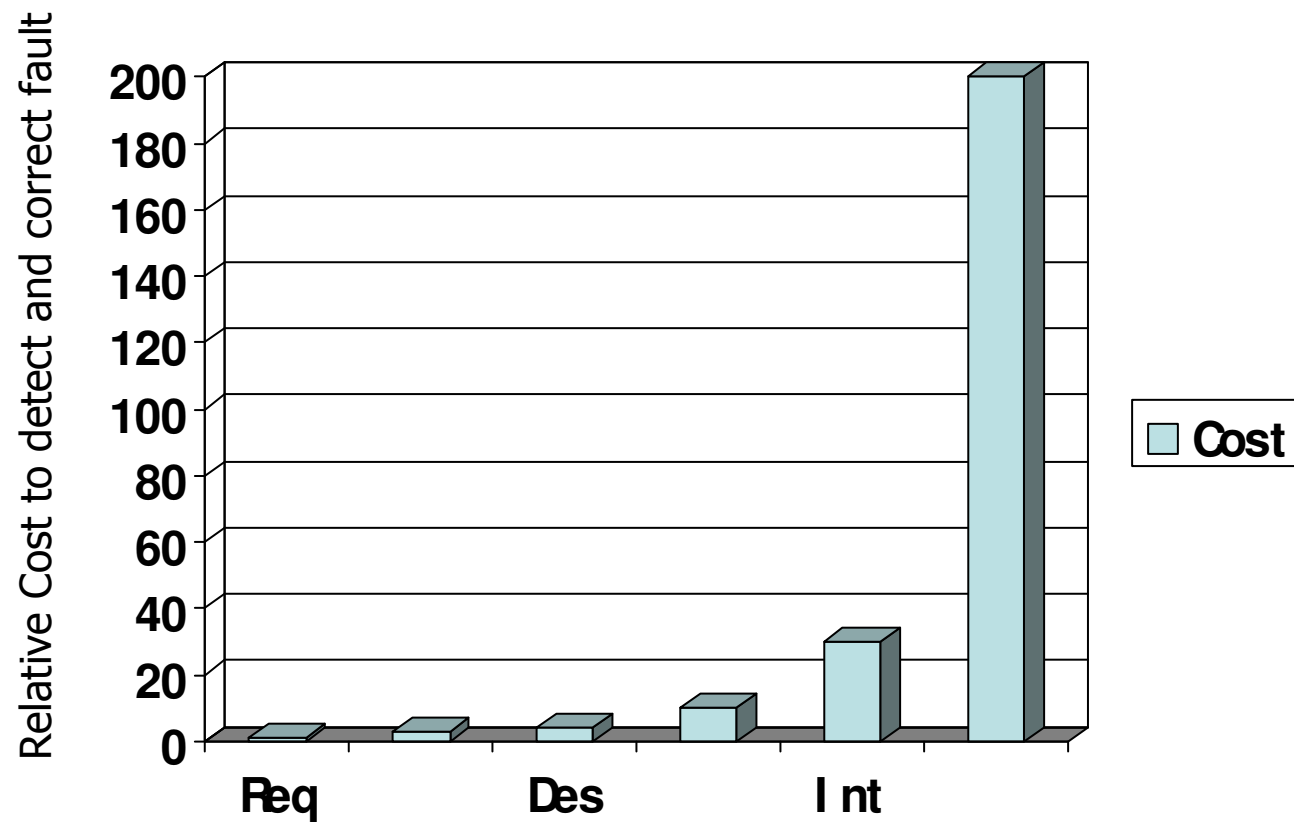
Software Project Planning

Relative Cost of Software Phases



Software Project Planning

Cost to Detect and Fix Faults



Software Project Planning

Cost Estimation

A number of estimation techniques have been developed and are having following attributes in common :

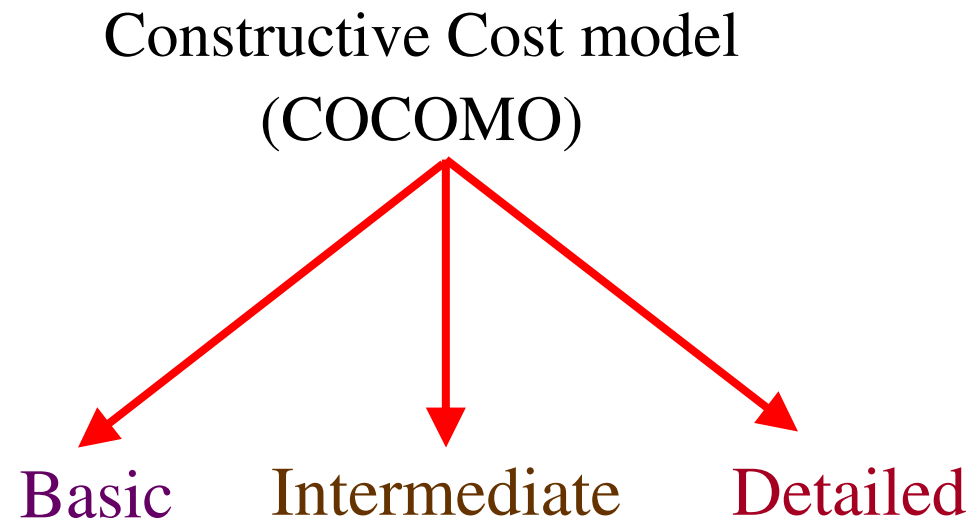
- Project scope must be established in advance
- Software metrics are used as a basis from which estimates are made
- The project is broken into small pieces which are estimated individually

To achieve reliable cost and schedule estimates, a number of options arise:

- Delay estimation until late in project
- Use simple decomposition techniques to generate project cost and schedule estimates
- Develop empirical models for estimation
- Acquire one or more automated estimation tools

Software Project Planning

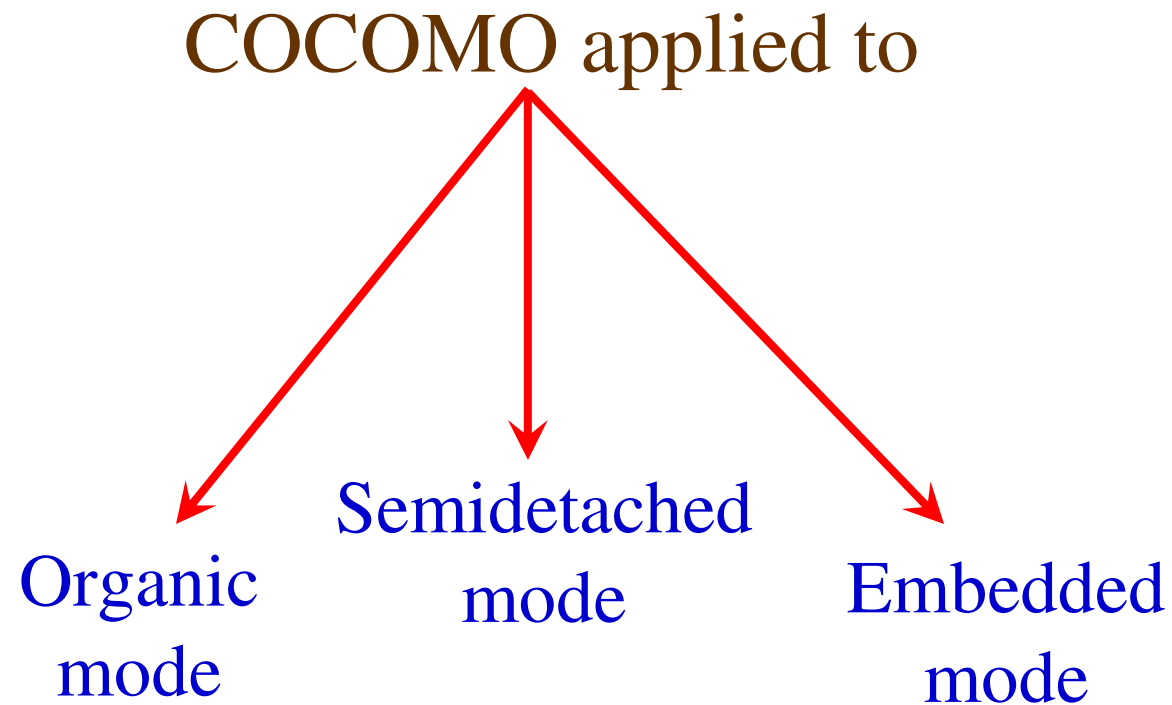
The Constructive Cost Model (COCOMO)



Model proposed by
B. W. Boehm's
through his book

Software Engineering Economics in 1981

Software Project Planning



Software Project Planning

Mode	Project size	Nature of Project	Innovation	Deadline of the project	Development Environment
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	Familiar & In house
Semi detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar project. For example: Utility systems like compilers, database systems, editors etc.	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

Table 4: The comparison of three COCOMO modes

Software Project Planning

Basic Model

Basic COCOMO model takes the form

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (E)^{d_b}$$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a_b , b_b , c_b and d_b are given in table 4 (a).

Software Project Planning

Software Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 4(a): Basic COCOMO coefficients

Software Project Planning

When effort and development time are known, the average staff size to complete the project may be calculated as:

$$\text{Average staff size } (SS) = \frac{E}{D} \text{ Persons}$$

When project size is known, the productivity level may be calculated as:

$$\text{Productivity } (P) = \frac{KLOC}{E} \text{ KLOC / PM}$$

Software Project Planning

Example: 4.5

Suppose that a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three modes i.e., organic, semidetached and embedded.

Software Project Planning

Solution

The basic COCOMO equation take the form:

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (KLOC)^{d_b}$$

Estimated size of the project = 400 KLOC

(i) Organic mode

$$E = 2.4(400)^{1.05} = 1295.31 \text{ PM}$$

$$D = 2.5(1295.31)^{0.38} = 38.07 \text{ PM}$$

Software Project Planning

(ii) Semidetached mode

$$E = 3.0(400)^{1.12} = 2462.79 \text{ PM}$$

$$D = 2.5(2462.79)^{0.35} = 38.45 \text{ PM}$$

(iii) Embedded mode

$$E = 3.6(400)^{1.20} = 4772.81 \text{ PM}$$

$$D = 2.5(4772.8)^{0.32} = 38 \text{ PM}$$

Software Project Planning

Example: 4.6

A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the effort, development time, average staff size and productivity of the project.

Software Project Planning

Solution

The semi-detached mode is the most appropriate mode; keeping in view the size, schedule and experience of the development team.

Hence $E = 3.0(200)^{1.12} = 1133.12 \text{ PM}$

$$D = 2.5(1133.12)^{0.35} = 29.3 \text{ PM}$$

$$\text{Average staff size } (SS) = \frac{E}{D} \text{ Persons}$$

$$= \frac{1133.12}{29.3} = 38.67 \text{ Persons}$$

Software Project Planning

$$\text{Productivity} = \frac{KLOC}{E} = \frac{200}{1133.12} = 0.1765 \text{ KLOC / PM}$$

$$P = 176 \text{ LOC / PM}$$

Software Project Planning

Intermediate Model

Cost drivers

(i) Product Attributes

- Required s/w reliability
- Size of application database
- Complexity of the product

(ii) Hardware Attributes

- Run time performance constraints
- Memory constraints
- Virtual machine volatility
- Turnaround time

Software Project Planning

(iii) Personal Attributes

- Analyst capability
- Programmer capability
- Application experience
- Virtual m/c experience
- Programming language experience

(iv) Project Attributes

- Modern programming practices
- Use of software tools
- Required development Schedule

Software Project Planning

Multipliers of different cost drivers

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Product Attributes						
RELY	0.75	0.88	1.00	1.15	1.40	--
DATA	--	0.94	1.00	1.08	1.16	--
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME	--	--	1.00	1.11	1.30	1.66
STOR	--	--	1.00	1.06	1.21	1.56
VIRT	--	0.87	1.00	1.15	1.30	--
TURN	--	0.87	1.00	1.07	1.15	--

Software Project Planning

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Personnel Attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	--
AEXP	1.29	1.13	1.00	0.91	0.82	--
PCAP	1.42	1.17	1.00	0.86	0.70	--
VEXP	1.21	1.10	1.00	0.90	--	--
LEXP	1.14	1.07	1.00	0.95	--	--
Project Attributes						
MODP	1.24	1.10	1.00	0.91	0.82	--
TOOL	1.24	1.10	1.00	0.91	0.83	--
SCED	1.23	1.08	1.00	1.04	1.10	--

Table 5: Multiplier values for effort calculations

Software Project Planning

Intermediate COCOMO equations

$$E = a_i (KLOC)^{b_i} * EAF$$

$$D = c_i (E)^{d_i}$$

Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Table 6: Coefficients for intermediate COCOMO

Software Project Planning

Example: 4.7

A new project with estimated 400 KLOC embedded system has to be developed. Project manager has a choice of hiring from two pools of developers: Very highly capable with very little experience in the programming language being used

Or

Developers of low quality but a lot of experience with the programming language. What is the impact of hiring all developers from one or the other pool ?

Software Project Planning

Solution

This is the case of embedded mode and model is intermediate COCOMO.

$$\begin{aligned}\text{Hence } E &= a_i (KLOC)^{d_i} \\ &= 2.8 (400)^{1.20} = 3712 \text{ PM}\end{aligned}$$

Case I: Developers are very highly capable with very little experience in the programming being used.

$$EAF = 0.82 \times 1.14 = 0.9348$$

$$E = 3712 \times .9348 = 3470 \text{ PM}$$

$$D = 2.5 (3470)^{0.32} = 33.9 \text{ M}$$

Software Project Planning

Case II: Developers are of low quality but lot of experience with the programming language being used.

$$\text{EAF} = 1.29 \times 0.95 = 1.22$$

$$\text{E} = 3712 \times 1.22 = 4528 \text{ PM}$$

$$\text{D} = 2.5 (4528)^{0.32} = 36.9 \text{ M}$$

Case II requires more effort and time. Hence, low quality developers with lot of programming language experience could not match with the performance of very highly capable developers with very little experience.

Software Project Planning

Example: 4.8

Consider a project to develop a full screen editor. The major components identified are:

- I. Screen edit
- II. Command Language Interpreter
- III. File Input & Output
- IV. Cursor Movement
- V. Screen Movement

The size of these are estimated to be 4k, 2k, 1k, 2k and 3k delivered source code lines. Use COCOMO to determine

1. Overall cost and schedule estimates (assume values for different cost drivers, with at least three of them being different from 1.0)
2. Cost & Schedule estimates for different phases.

Software Project Planning

Solution

Size of five modules are:

Screen edit = 4 KLOC

Command language interpreter = 2 KLOC

File input and output = 1 KLOC

Cursor movement = 2 KLOC

Screen movement = 3 KLOC

Total = 12 KLOC

Software Project Planning

Let us assume that significant cost drivers are

- i. Required software reliability is high, i.e., 1.15
- ii. Product complexity is high, i.e., 1.15
- iii. Analyst capability is high, i.e., 0.86
- iv. Programming language experience is low, i.e., 1.07
- v. All other drivers are nominal

$$\text{EAF} = 1.15 \times 1.15 \times 0.86 \times 1.07 = 1.2169$$

Software Project Planning

- (a) The initial effort estimate for the project is obtained from the following equation

$$\begin{aligned} E &= a_i (\text{KLOC})^{b_i} \times \text{EAF} \\ &= 3.2(12)^{1.05} \times 1.2169 = 52.91 \text{ PM} \end{aligned}$$

Development time

$$\begin{aligned} D &= C_i (E)^{d_i} \\ &= 2.5(52.91)^{0.38} = 11.29 \text{ M} \end{aligned}$$

- (b) Using the following equations and referring Table 7, phase wise cost and schedule estimates can be calculated.

$$E_p = \mu_p E$$

$$D_p = \tau_p D$$

Software Project Planning

Since size is only 12 KLOC, it is an organic small model. Phase wise effort distribution is given below:

System Design	$= 0.16 \times 52.91 = 8.465 \text{ PM}$
Detailed Design	$= 0.26 \times 52.91 = 13.756 \text{ PM}$
Module Code & Test	$= 0.42 \times 52.91 = 22.222 \text{ PM}$
Integration & Test	$= 0.16 \times 52.91 = 8.465 \text{ Pm}$

Now Phase wise development time duration is

System Design	$= 0.19 \times 11.29 = 2.145 \text{ M}$
Detailed Design	$= 0.24 \times 11.29 = 2.709 \text{ M}$
Module Code & Test	$= 0.39 \times 11.29 = 4.403 \text{ M}$
Integration & Test	$= 0.18 \times 11.29 = 2.032 \text{ M}$

Software Design

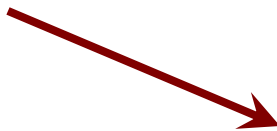


Software Design

- ❖ More creative than analysis
- ❖ Problem solving activity

WHAT IS DESIGN

‘HOW’



Software design document (SDD)

Software Design

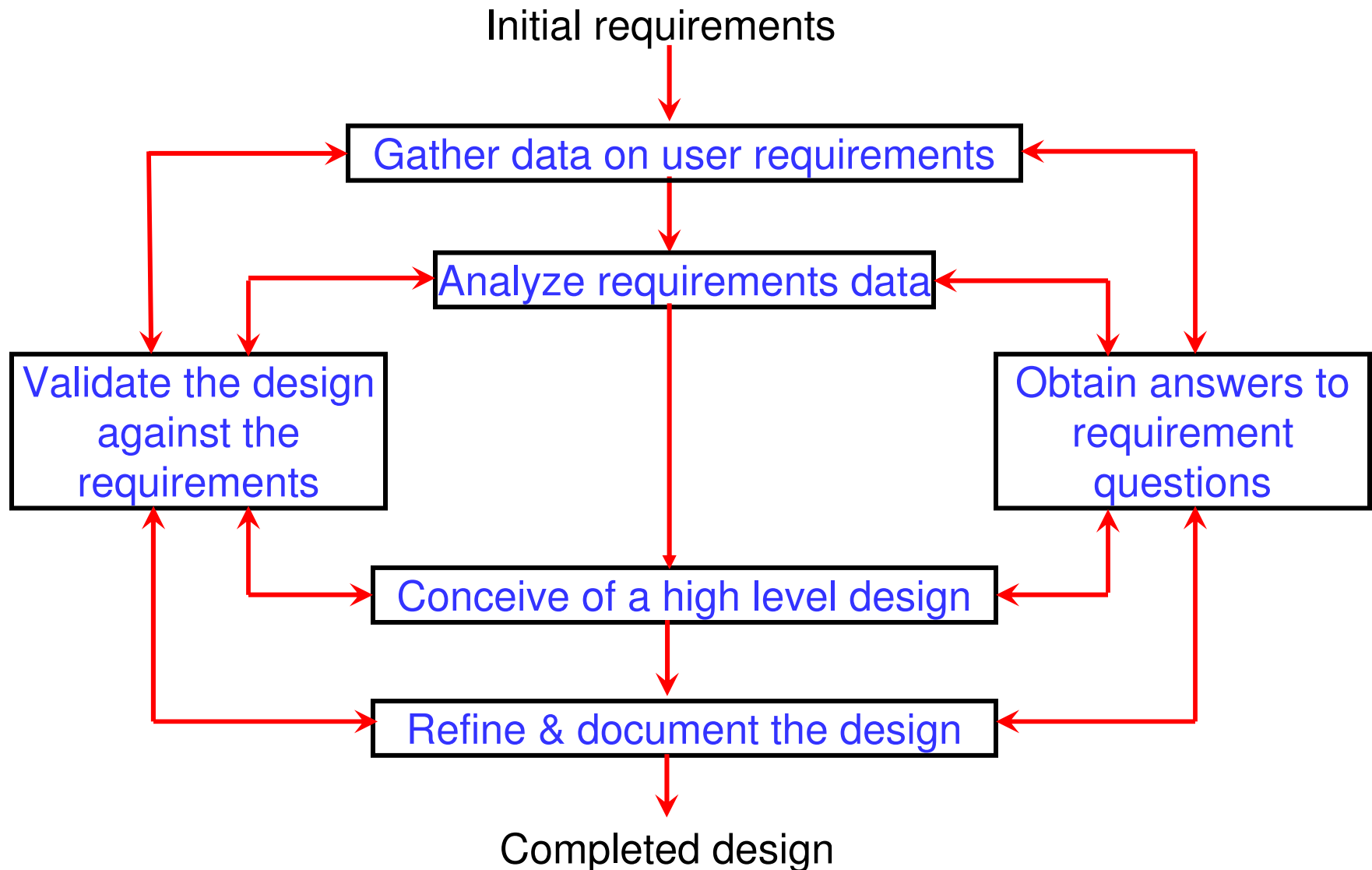
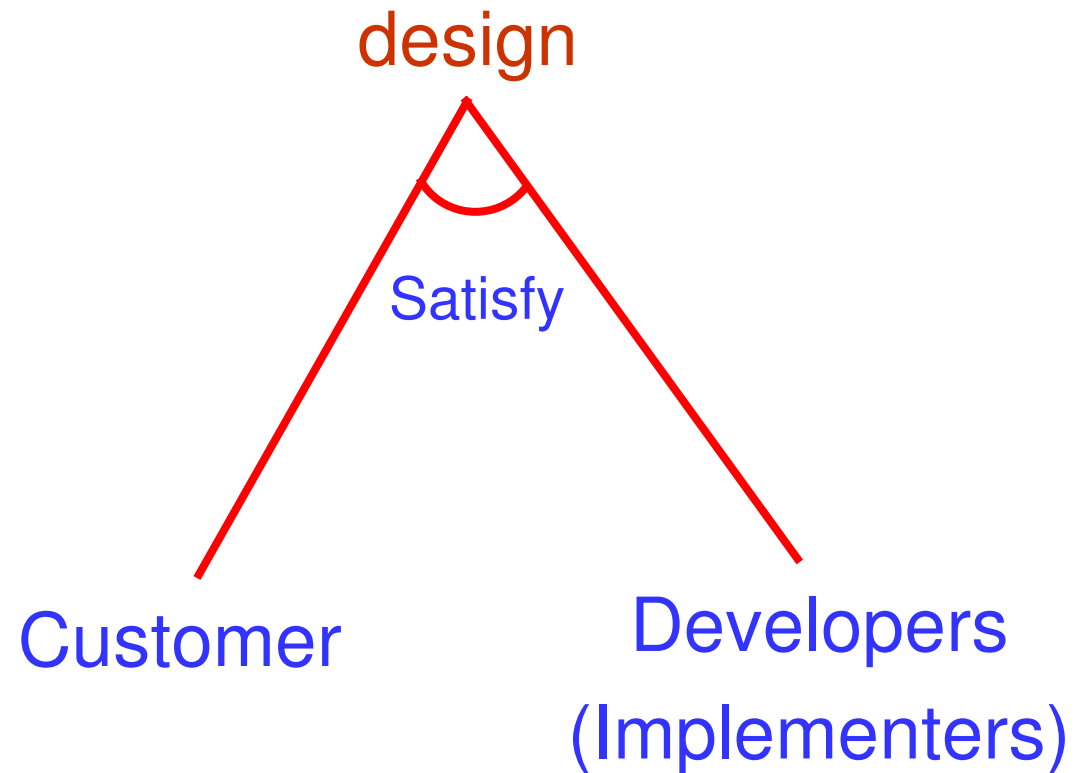


Fig. 1 : Design framework

Software Design



Software Design

Conceptual Design and Technical Design

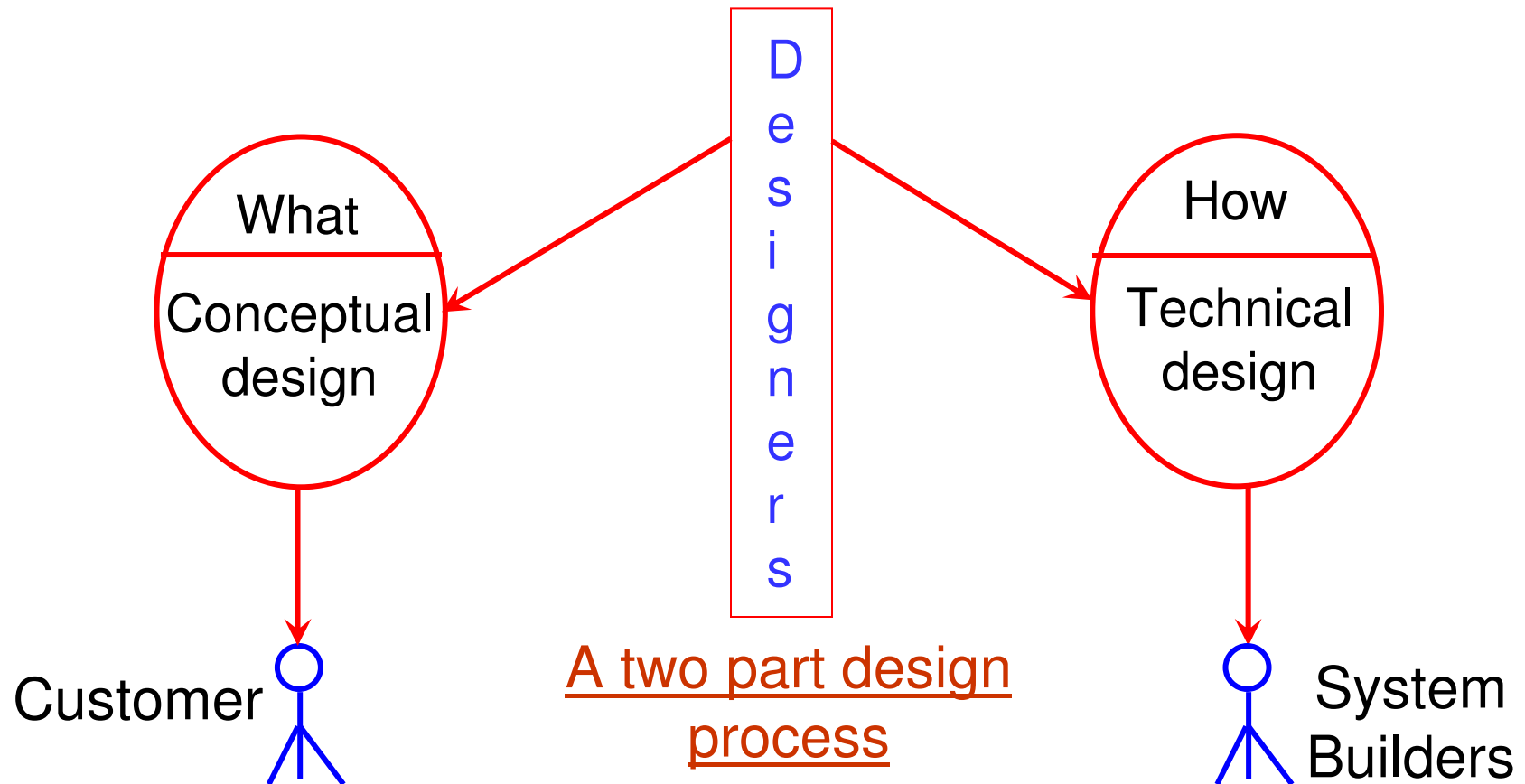


Fig. 2 : A two part design process

Software Design

Conceptual design answers :

- ✓ Where will the data come from ?
- ✓ What will happen to data in the system?
- ✓ How will the system look to users?
- ✓ What choices will be offered to users?
- ✓ What is the timings of events?
- ✓ How will the reports & screens look like?

Software Design

Technical design describes :

- ❖ Hardware configuration
- ❖ Software needs
- ❖ Communication interfaces
- ❖ I/O of the system
- ❖ Software architecture
- ❖ Network architecture
- ❖ Any other thing that translates the requirements in to a solution to the customer's problem.

Software Design

The design needs to be

- Correct & complete
- Understandable
- At the right level
- Maintainable

Software Design

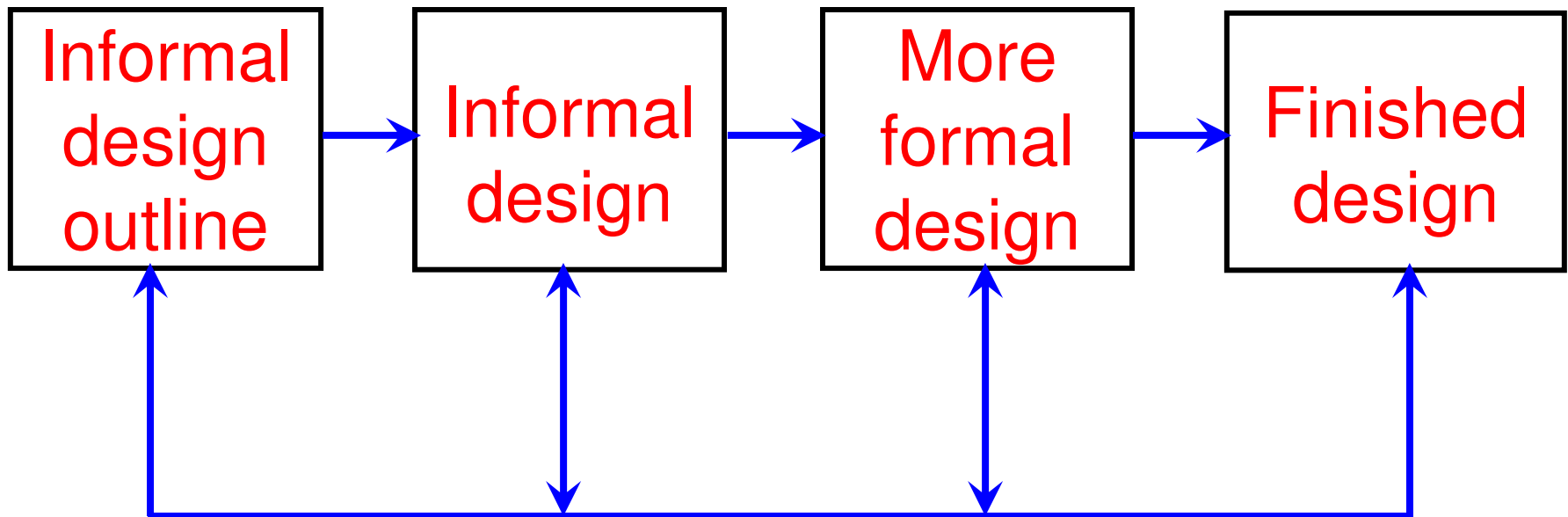


Fig. 3 : The transformation of an informal design to a detailed design.

Software Design

MODULARITY

There are many definitions of the term module. Range is from :

- i. Fortran subroutine
- ii. Ada package
- iii. Procedures & functions of PASCAL & C
- iv. C++ / Java classes
- v. Java packages
- vi. Work assignment for an individual programmer

Software Design

All these definitions are correct. A modular system consist of well defined manageable units with well defined interfaces among the units.

Software Design

Properties :

- i. Well defined subsystem
- ii. Well defined purpose
- iii. Can be separately compiled and stored in a library.
- iv. Module can use other modules
- v. Module should be easier to use than to build
- vi. Simpler from outside than from the inside.

Software Design

Modularity is the single attribute of software that allows a program to be intellectually manageable.

It enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of software product.

Software Design

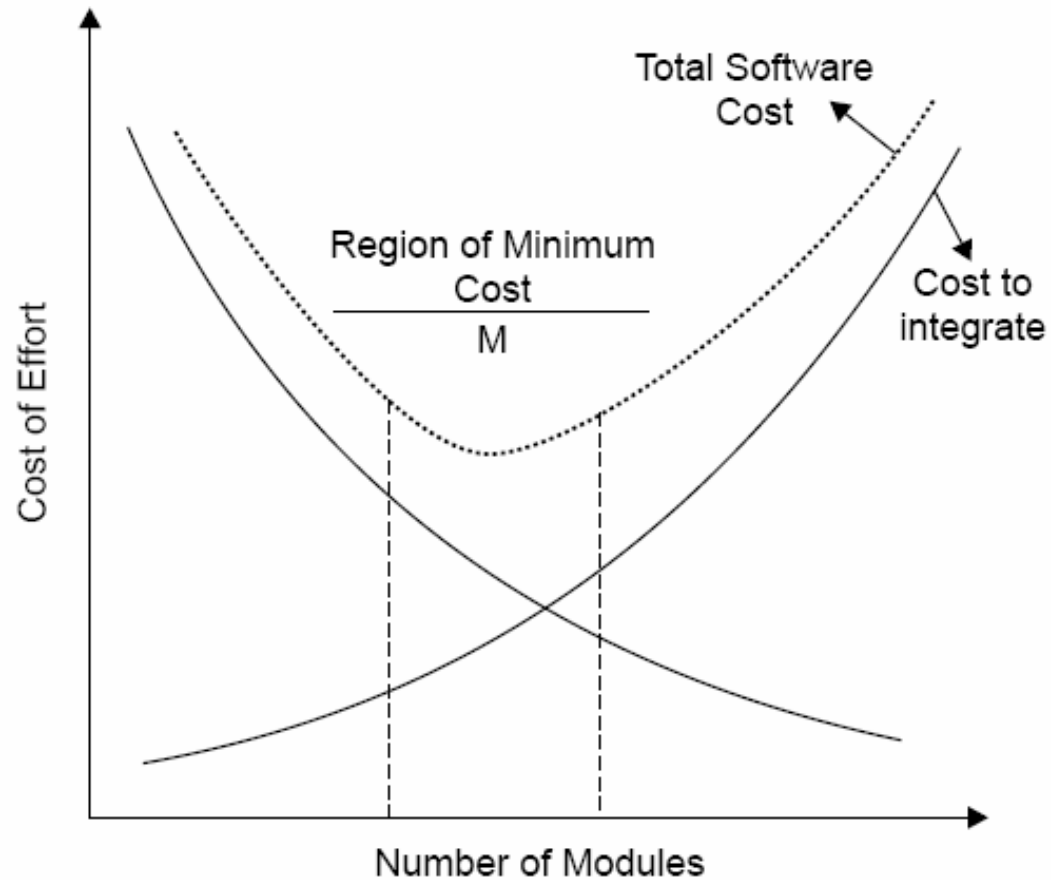
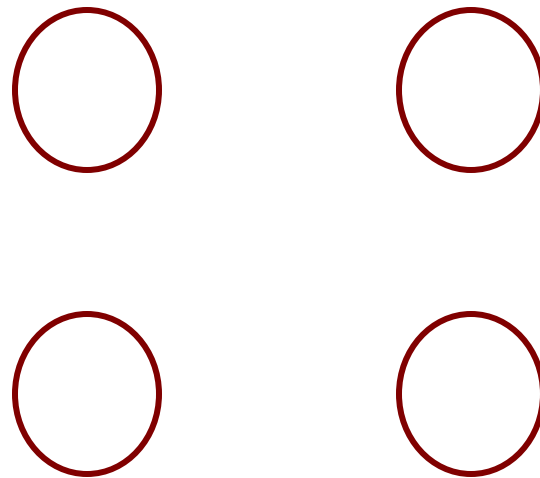


Fig. 4 : Modularity and software cost

Software Design

Module Coupling

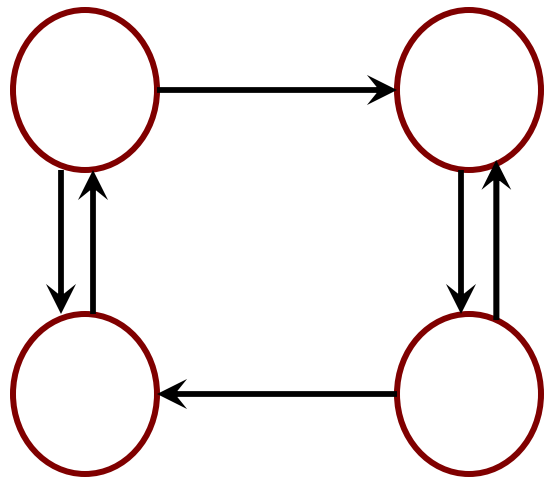
Coupling is the measure of the degree of interdependence between modules.



(Uncoupled : no dependencies)

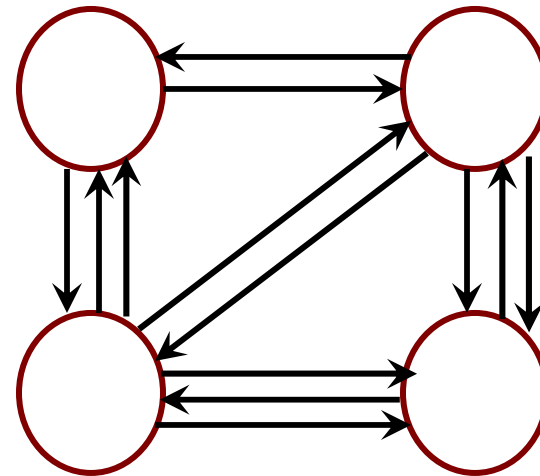
(a)

Software Design



Loosely coupled:
some dependencies

(B)



Highly coupled:
many dependencies

(C)

Fig. 5 : Module coupling

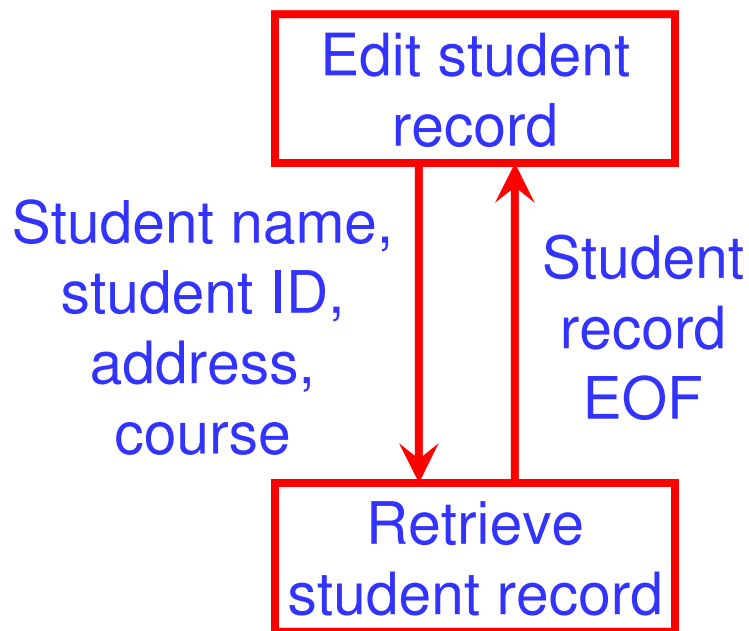
Software Design

This can be achieved as:

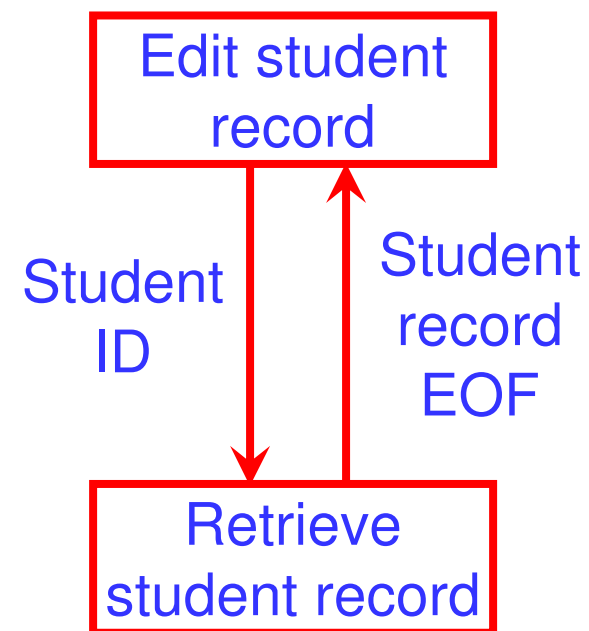
- ❑ Controlling the number of parameters passed amongst modules.
- ❑ Avoid passing undesired data to calling module.
- ❑ Maintain parent / child relationship between calling & called modules.
- ❑ Pass data, not the control information.

Software Design

Consider the example of editing a student record in a 'student information system'.



Poor design: Tight Coupling



Good design: Loose Coupling

Fig. 6 : Example of coupling

Software Design


Data coupling	Best
Stamp coupling	
Control coupling	
External coupling	
Common coupling	
Content coupling	Worst

Fig. 7 : The types of module coupling

Given two procedures A & B, we can identify number of ways in which they can be coupled.

Software Design

Data coupling

The dependency between module A and B is said to be data coupled if their dependency is based on the fact they communicate by only passing of data. Other than communicating through data, the two modules are independent.

Stamp coupling

Stamp coupling occurs between module A and B when complete data structure is passed from one module to another.

Software Design

Control coupling

Module A and B are said to be control coupled if they communicate by passing of control information. This is usually accomplished by means of flags that are set by one module and reacted upon by the dependent module.

Common coupling

With common coupling, module A and module B have shared data. Global data areas are commonly found in programming languages. Making a change to the common data means tracing back to all the modules which access that data to evaluate the effect of changes.

Software Design

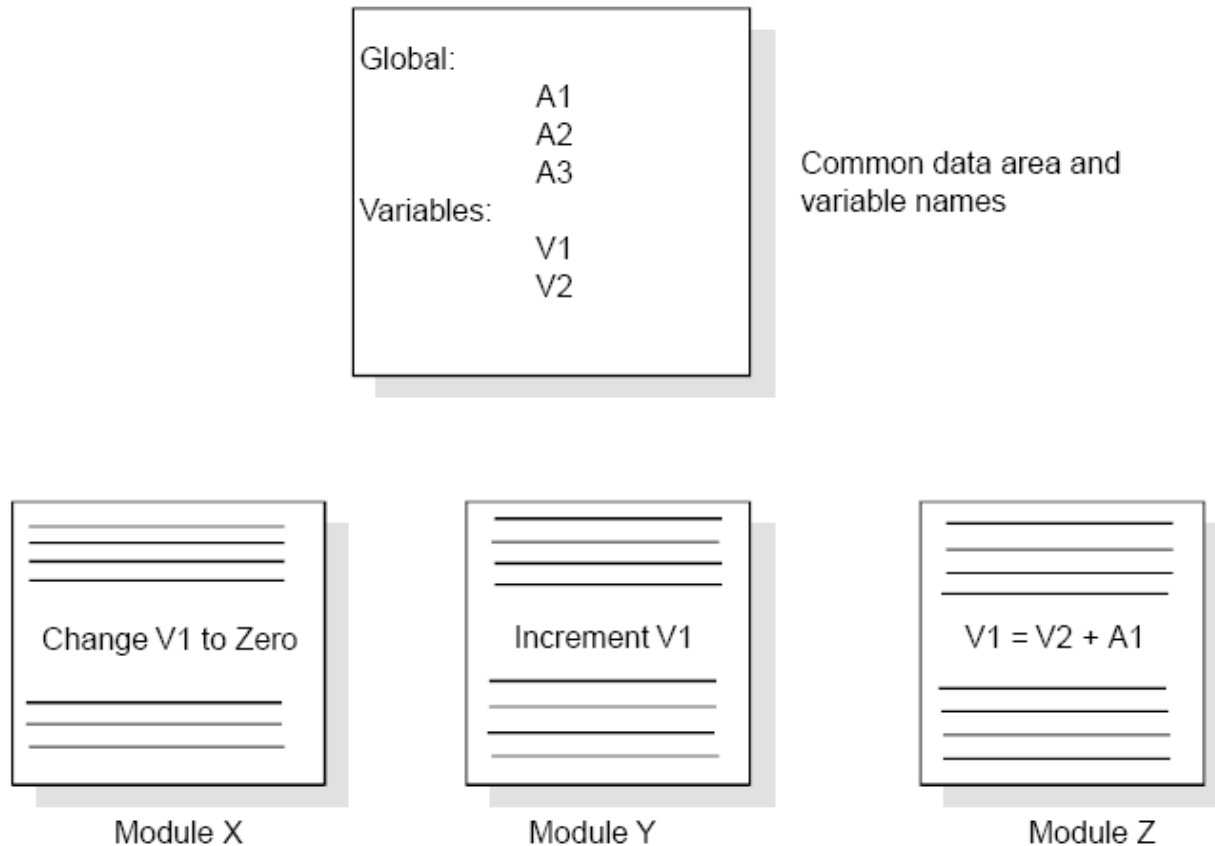


Fig. 8 : Example of common coupling

Software Design

Content coupling

Content coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another. In Fig. 9, module B branches into D, even though D is supposed to be under the control of C.

Software Design

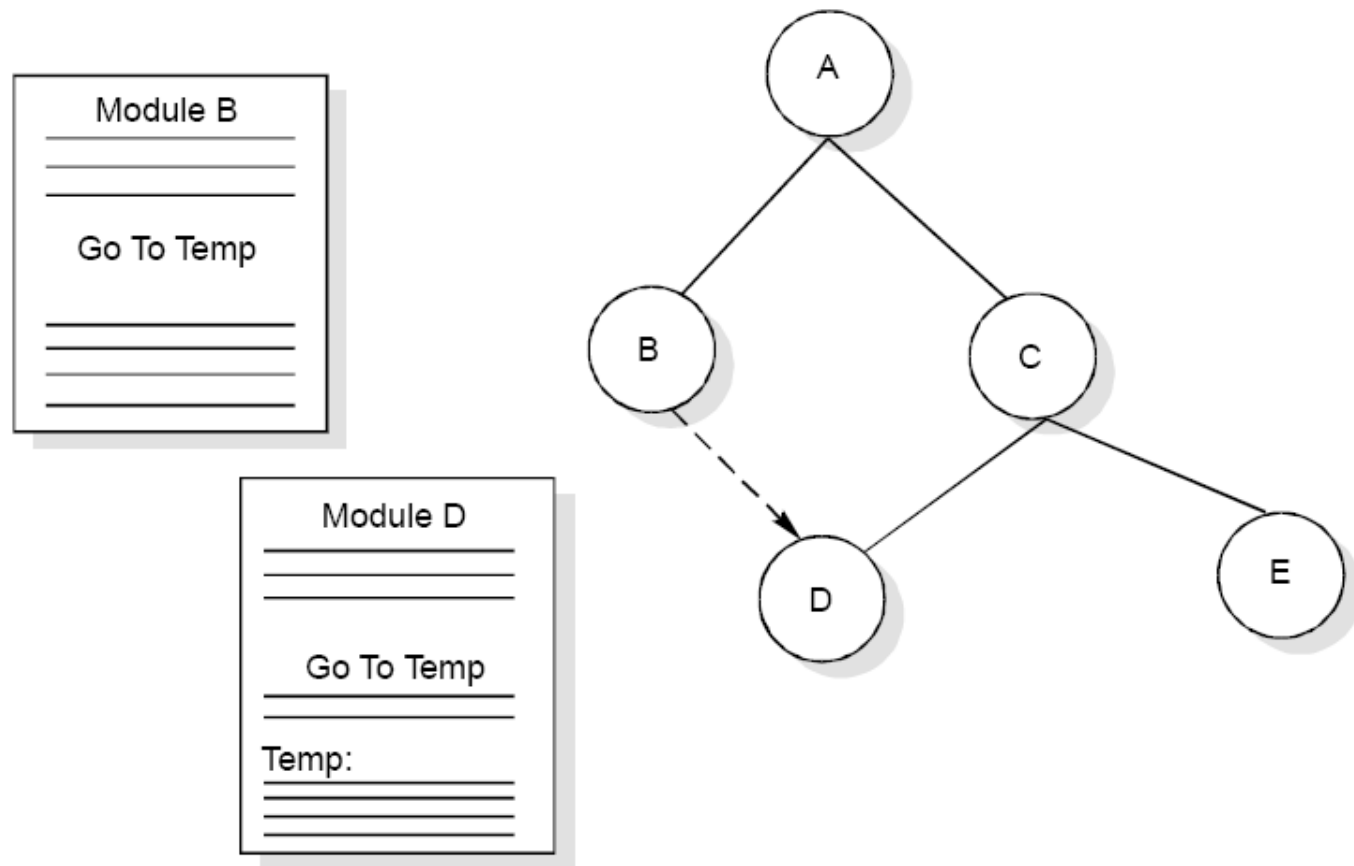


Fig. 9 : Example of content coupling

Software Design

Module Cohesion

Cohesion is a measure of the degree to which the elements of a module are functionally related.

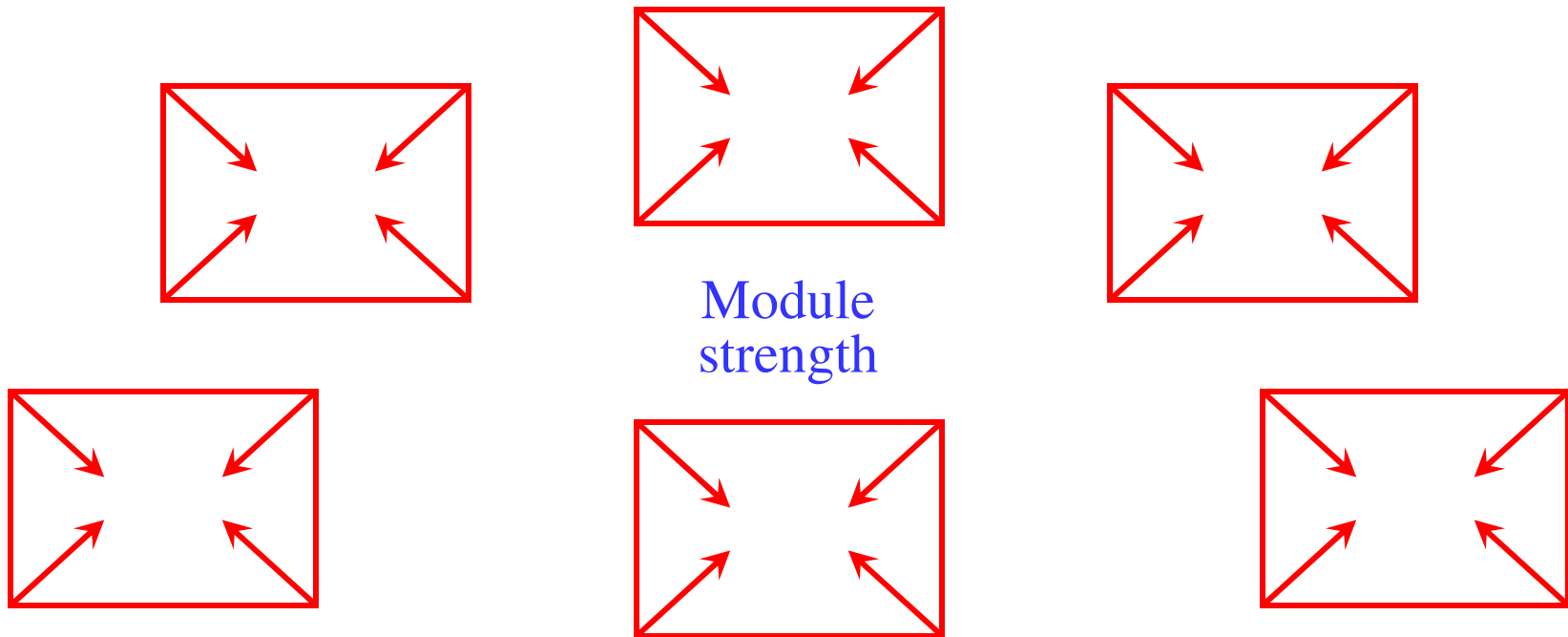


Fig. 10 : Cohesion=Strength of relations within modules

Software Design

Types of cohesion

- Functional cohesion
- Sequential cohesion
- Procedural cohesion
- Temporal cohesion
- Logical cohesion
- Coincident cohesion

Software Design


Functional Cohesion	Best (high)
Sequential Cohesion	
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	Worst (low)

Fig. 11 : Types of module cohesion

Software Design

Functional Cohesion

- A and B are part of a single functional task. This is very good reason for them to be contained in the same procedure.

Sequential Cohesion

- Module A outputs some data which forms the input to B. This is the reason for them to be contained in the same procedure.

Software Design

Procedural Cohesion

- Procedural Cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed.

Temporal Cohesion

- Module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time-span.

Software Design

Logical Cohesion

- Logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions.

Coincidental Cohesion

- Coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another.

Software Design

Relationship between Cohesion & Coupling

If the software is not properly modularized, a host of seemingly trivial enhancement or changes will result into death of the project. Therefore, a software engineer must design the modules with goal of high cohesion and low coupling.

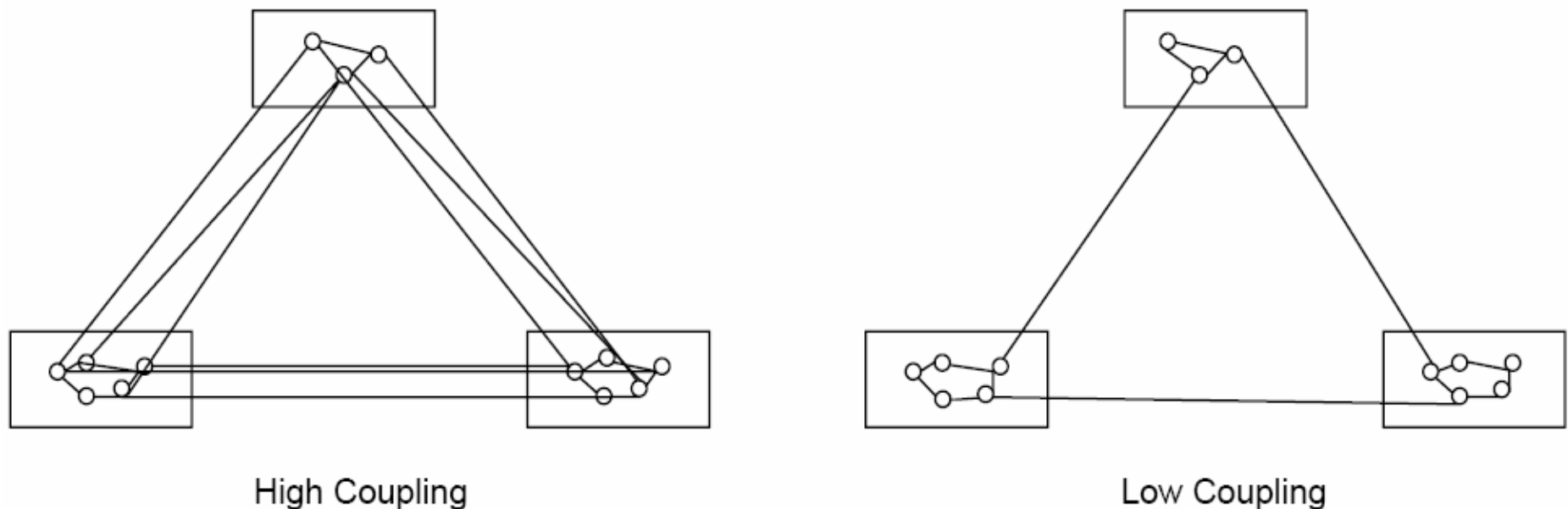


Fig. 12 : View of cohesion and coupling

Software Design

STRATEGY OF DESIGN

A good system design strategy is to organize the program modules in such a way that are easy to develop and latter to, change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program. It is important for two reasons:

- First, even pre-existing code, if any, needs to be understood, organized and pieced together.
- Second, it is still common for the project team to have to write some code and produce original programs that support the application logic of the system.

Software Design

Bottom-Up Design

These modules are collected together in the form of a “library”.

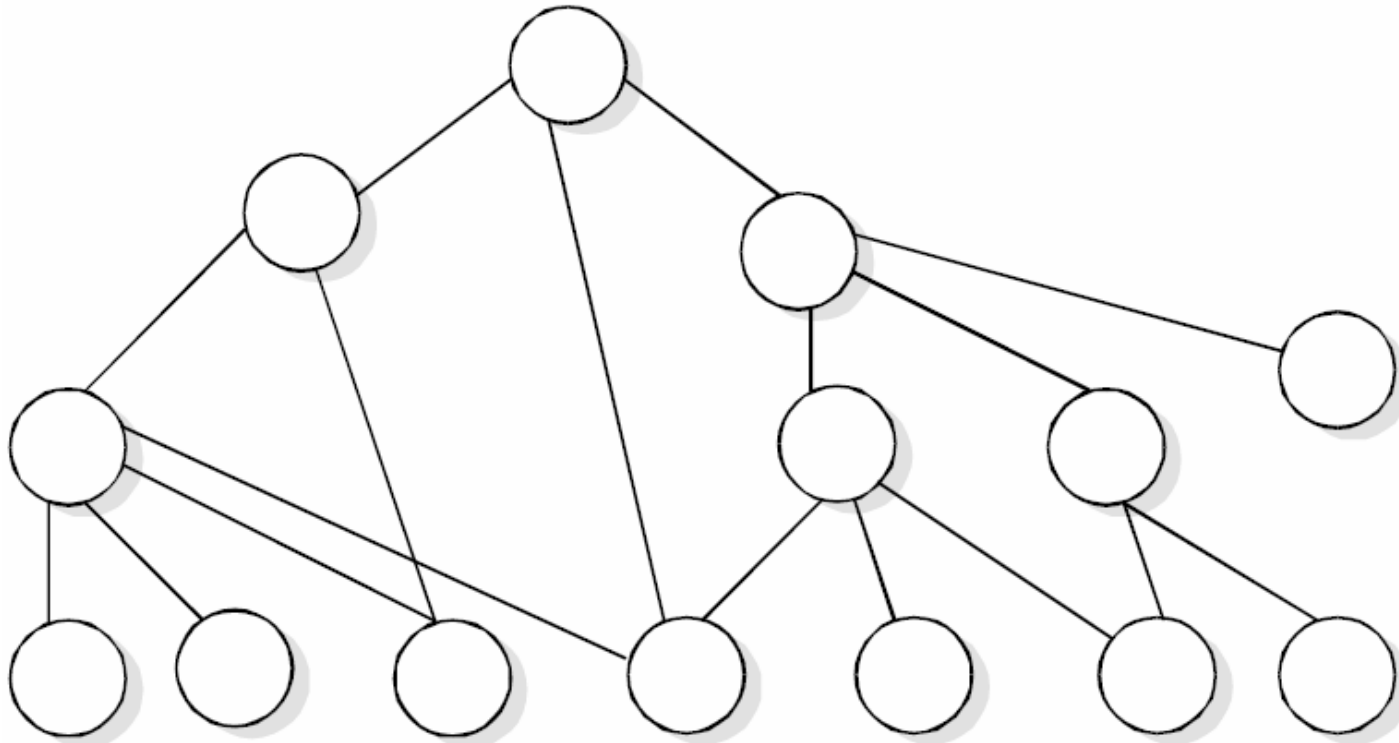


Fig. 13 : Bottom-up tree structure

Software Design

Top-Down Design

A top down design approach starts by identifying the major modules of the system, decomposing them into their lower level modules and iterating until the desired level of detail is achieved. This is stepwise refinement; starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly.

Software Design

Hybrid Design

For top-down approach to be effective, some bottom-up approach is essential for the following reasons:

- To permit common sub modules.
- Near the bottom of the hierarchy, where the intuition is simpler, and the need for bottom-up testing is greater, because there are more number of modules at low levels than high levels.
- In the use of pre-written library modules, in particular, reuse of modules.

Software Design

FUNCTION ORIENTED DESIGN

Function Oriented design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function. Thus, system is designed from a functional viewpoint.

Software Design

Consider the example of scheme interpreter. Top-level function may look like:

While (not finished)

```
{  
    Read an expression from the terminal;  
    Evaluate the expression;  
    Print the value;  
}
```

We thus get a fairly natural division of our interpreter into a “read” module, an “evaluate” module and a “print” module. Now we consider the “print” module and is given below:

Print (expression exp)

```
{  
    Switch (exp → type)  
    Case integer: /*print an integer*/  
    Case real:   /*print a real*/  
    Case list:   /*print a list*/  
    ...  
}
```

Software Design

We continue the refinement of each module until we reach the statement level of our programming language. At that point, we can describe the structure of our program as a tree of refinement as in design top-down structure as shown in fig. 14.

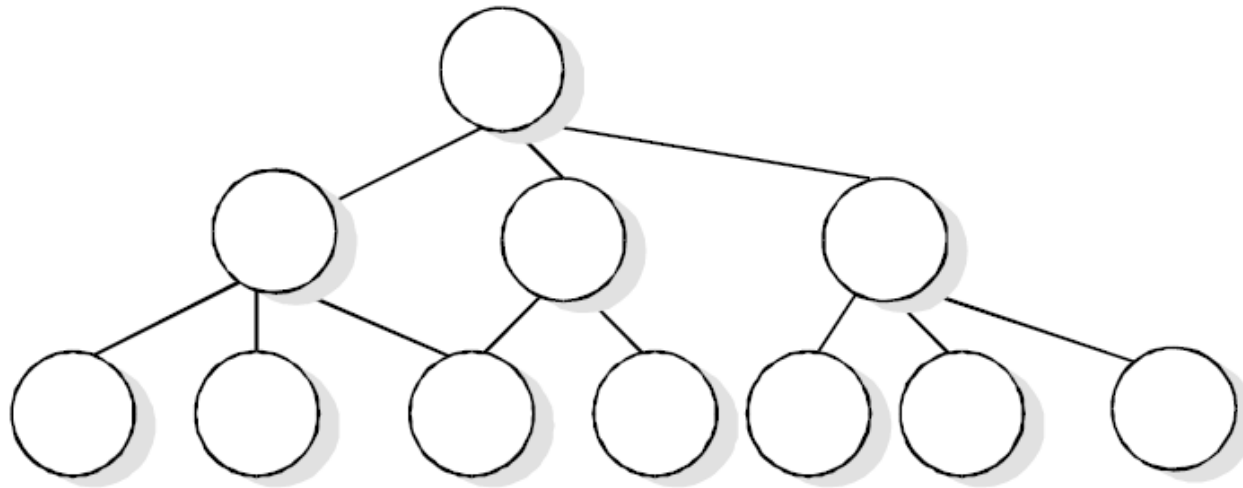


Fig. 14 : Top-down structure

Software Design

If a program is created top-down, the modules become very specialized. As one can easily see in top down design structure, each module is used by at most one other module, its parent. For a module, however, we must require that several other modules as in design reusable structure as shown in fig. 15.

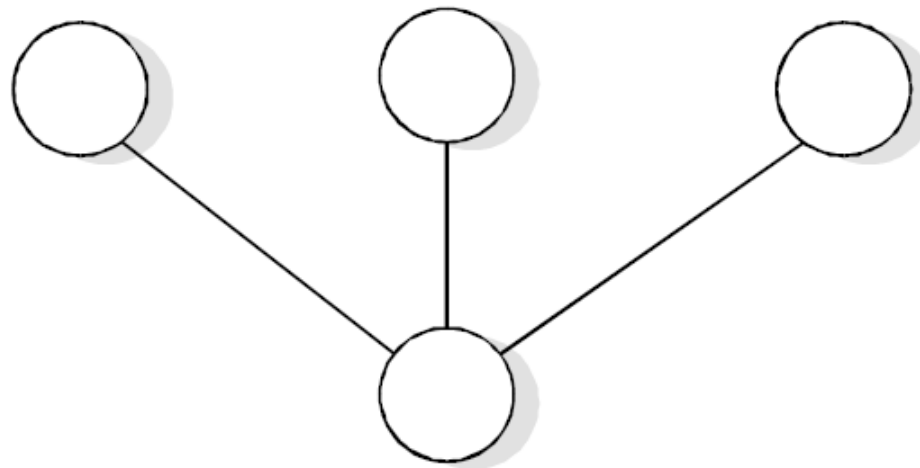


Fig. 15 : Design reusable structure

Software Design

Design Notations

Design notations are largely meant to be used during the process of design and are used to represent design or design decisions. For a function oriented design, the design can be represented graphically or mathematically by the following:

- Data flow diagrams
- Data Dictionaries
- Structure Charts
- Pseudocode

Software Design

Structure Chart

It partition a system into block boxes. A black box means that functionality is known to the user without the knowledge of internal design.

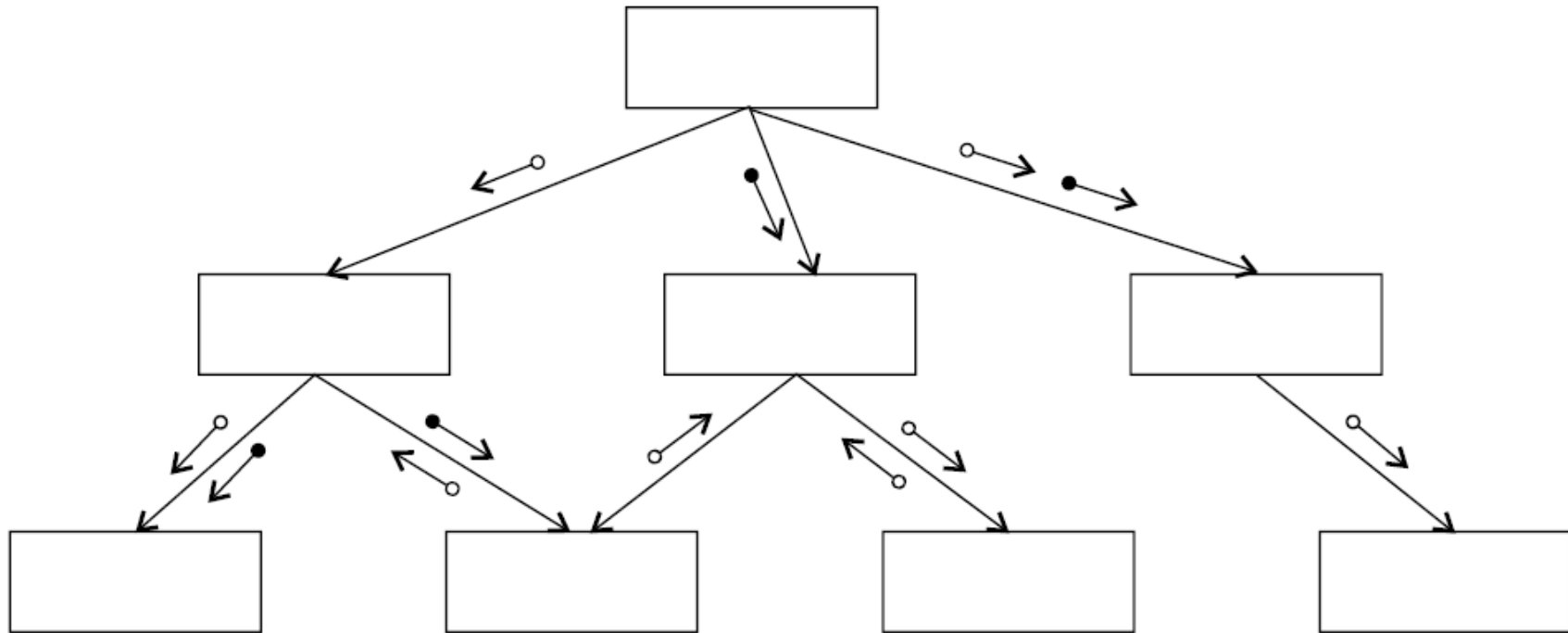


Fig. 16 : Hierarchical format of a structure chart

Software Engineering (3rd ed.), By K.K Aggarwal & Yogesh Singh, Copyright © New Age International Publishers, 2007

Software Design

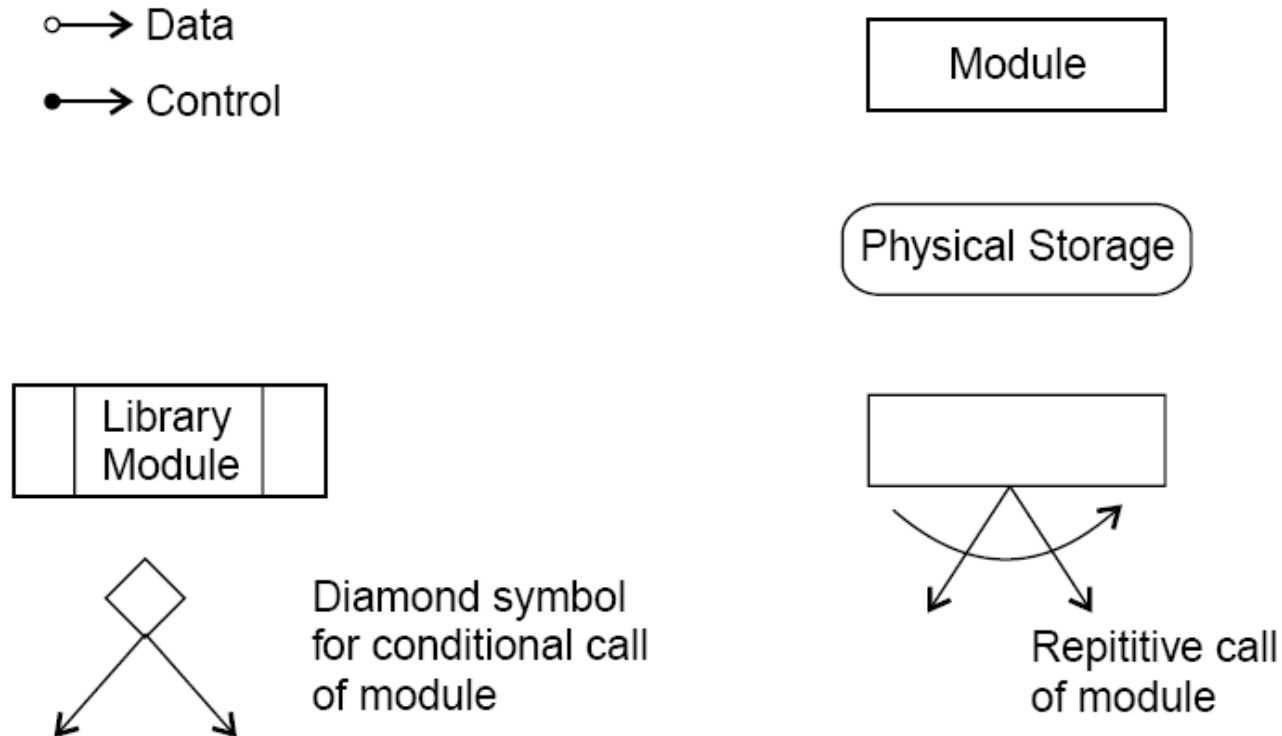


Fig. 17 : Structure chart notations

Software Design

A structure chart for “update file” is given in fig. 18.

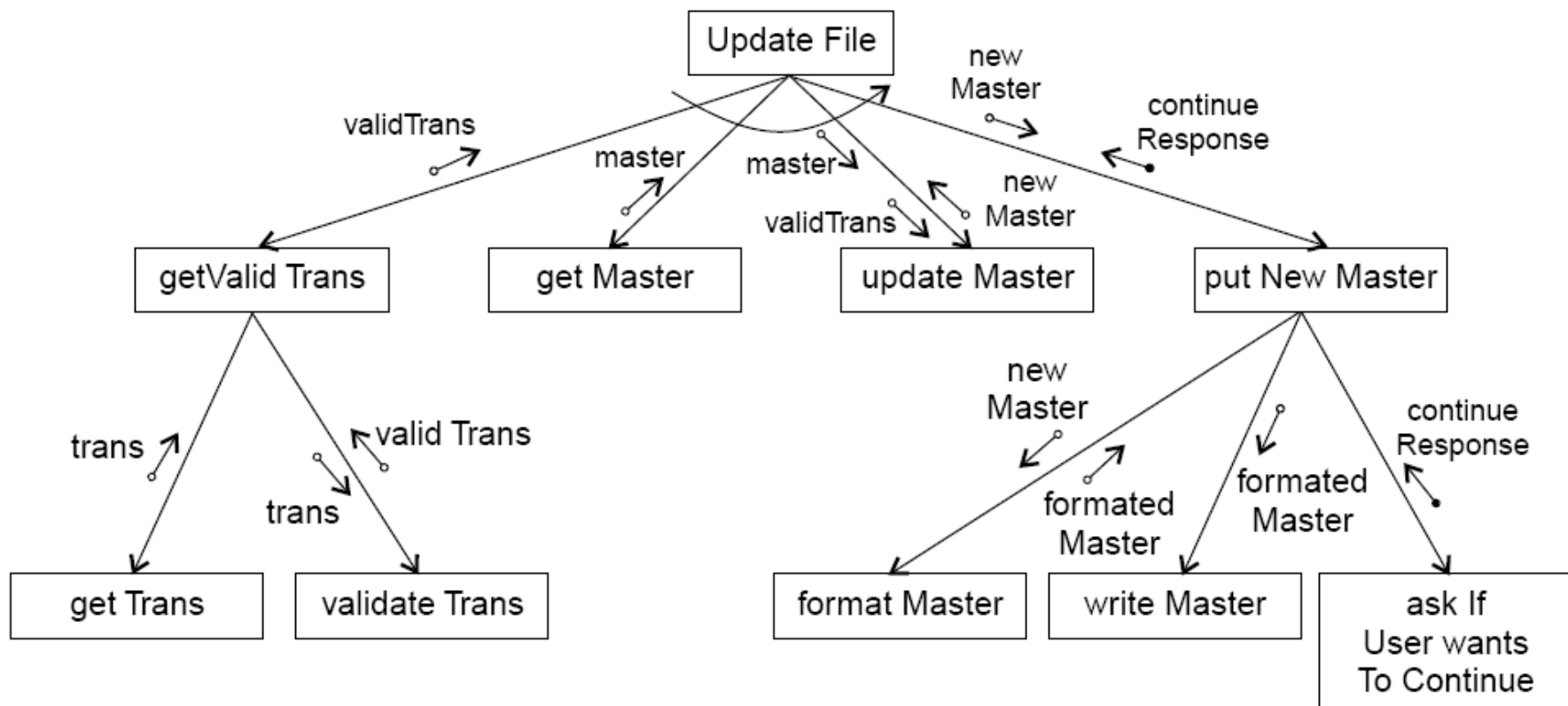


Fig. 18 : Update file

Software Design

A transaction centered structure describes a system that processes a number of different types of transactions. It is illustrated in Fig.19.

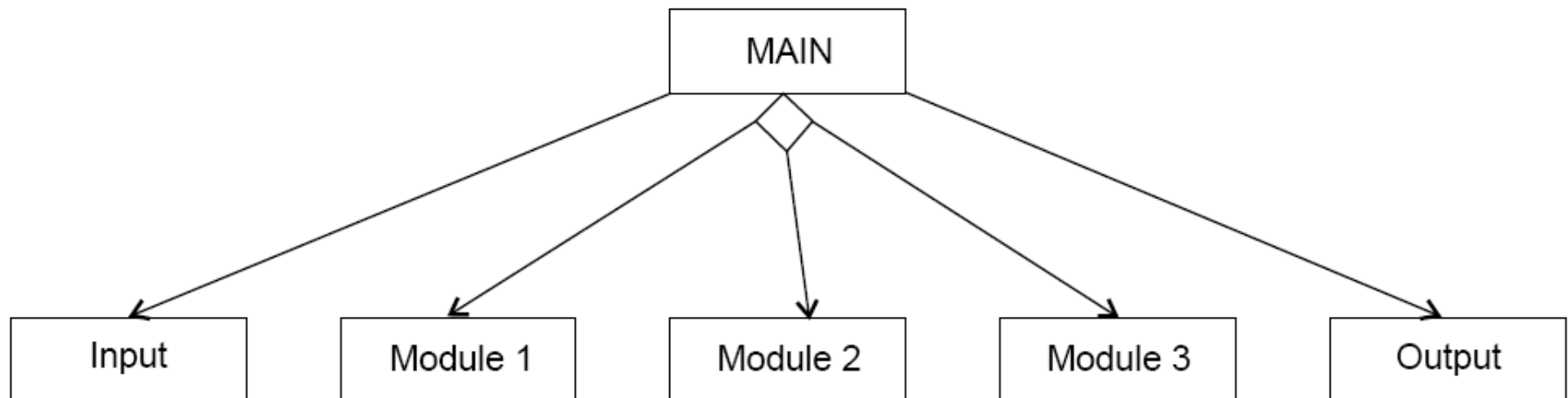


Fig. 19 : Transaction-centered structure

Software Design

In the above figure the MAIN module controls the system operation its functions is to:

- invoke the INPUT module to read a transaction;
- determine the kind of transaction and select one of a number of transaction modules to process that transaction, and
- output the results of the processing by calling OUTPUT module.

Software Design

Pseudocode

Pseudocode notation can be used in both the preliminary and detailed design phases.

Using pseudocode, the designer describes system characteristics using short, concise, English language phrases that are structured by key words such as It-Then-Else, While-Do, and End.

Software Design

Functional Procedure Layers

- Function are built in layers, Additional notation is used to specify details.
- Level 0
 - Function or procedure name
 - Relationship to other system components (e.g., part of which system, called by which routines, etc.)
 - Brief description of the function purpose.
 - Author, date

Software Design

➤ Level 1

- Function Parameters (problem variables, types, purpose, etc.)
- Global variables (problem variable, type, purpose, sharing information)
- Routines called by the function
- Side effects
- Input/Output Assertions

Software Design

➤ Level 2

- Local data structures (variable etc.)
- Timing constraints
- Exception handling (conditions, responses, events)
- Any other limitations

➤ Level 3

- Body (structured chart, English pseudo code, decision tables, flow charts, etc.)

Software Design

IEEE Recommended practice for software design descriptions (IEEE STD 1016-1998)

➤ Scope

An SDD is a representation of a software system that is used as a medium for communicating software design information.

➤ References

- i. IEEE std 830-1998, IEEE recommended practice for software requirements specifications.
- ii. IEEE std 610.12-1990, IEEE glossary of software engineering terminology.

Software Design

➤ Definitions

- i. **Design entity.** An element (Component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.
- ii. **Design View.** A subset of design entity attribute information that is specifically suited to the needs of a software project activity.
- iii. **Entity attributes.** A named property or characteristics of a design entity. It provides a statement of fact about the entity.
- iv. **Software design description (SDD).** A representation of a software system created to facilitate analysis, planning, implementation and decision making.

Software Design

➤ Purpose of an SDD

The SDD shows how the software system will be structured to satisfy the requirements identified in the SRS. It is basically the translation of requirements into a description of the software structure, software components, interfaces, and data necessary for the implementation phase. Hence, SDD becomes the blue print for the implementation activity.

➤ Design Description Information Content

- Introduction
- Design entities
- Design entity attributes

Software Design

The attributes and associated information items are defined in the following subsections:

- | | |
|-------------------|-----------------|
| a) Identification | f) Dependencies |
| b) Type | g) Interface |
| c) Purpose | h) Resources |
| d) Function | i) Processing |
| e) Subordinates | j) Data |

Software Design

➤ Design Description Organization

Each design description writer may have a different view of what are considered the essential aspects of a software design. The organization of SDD is given in table 1. This is one of the possible ways to organize and format the SDD.

A recommended organization of the SDD into separate design views to facilitate information access and assimilation is given in table 2.

Software Design

1. Introduction
 - 1.1 Purpose
 - 1.2 Scope
 - 1.3 Definitions and acronyms
2. References
3. Decomposition description
 - 3.1 Module decomposition
 - 3.1.1 Module 1 description
 - 3.1.2 Module 2 description
 - 3.2 Concurrent Process decomposition
 - 3.2.1 Process 1 description
 - 3.2.2 Process 2 description
 - 3.3 Data decomposition
 - 3.3.1 Data entity 1 description
 - 3.3.2 Data entity 2 description

Cont...

Software Design

- 4. Dependency description
 - 4.1 Intermodule dependencies
 - 4.2 Interprocess dependencies
 - 4.3 Data dependencies
- 5. Interface description
 - 5.1 Module Interface
 - 5.1.1 Module 1 description
 - 5.1.2 Module 2 description
 - 5.2 Process interface
 - 5.2.1 Process 1 description
 - 5.2.2 Process 2 description
- 6. Detailed design
 - 6.1 Module detailed design
 - 6.1.1 Module 1 detail
 - 6.1.2 Module 2 detail
 - 6.2 Data detailed design
 - 6.2.1 Data entry 1 detail
 - 6.2.2 Data entry 2 detail

Table 1:
Organization of
SDD

Software Design

Design View	Scope	Entity attribute	Example representation
Decomposition description	Partition of the system into design entities	Identification, type purpose, function, subordinate	Hierarchical decomposition diagram, natural language
Dependency description	Description of relationships among entities of system resources	Identification, type, purpose, dependencies, resources	Structure chart, data flow diagrams, transaction diagrams
Interface description	List of everything a designer, developer, tester needs to know to use design entities that make up the system	Identification, function, interfaces	Interface files, parameter tables
Detail description	Description of the internal design details of an entity	Identification, processing, data	Flow charts, PDL etc.

Table 2: Design views

Software Design

Object Oriented Design

Object oriented design is the result of focusing attention not on the function performed by the program, but instead on the data that are to do manipulated by the program. Thus, it is orthogonal to function oriented design.

Object Oriented Design begins with an examination of the real world “things” that are part of the problem to be solved. These things (which we will call objects) are characterized individually in terms of their attributes and behavior.

Software Design

➤ Basic Concepts

Object Oriented Design is not dependent on any specific implementation language. Problems are modeled using objects. Objects have:

- Behavior (they do things)
- State (which changes when they do things)

Software Design

The various terms related to object design are:

i. Objects

The word “Object” is used very frequently and conveys different meaning in different circumstances. Here, meaning is an entity able to save a state (information) and which offers a number of operations (behavior) to either examine or affect this state. An object is characterized by number of operations and a state which remembers the effect of these operations.

Software Design

ii. Messages

Objects communicate by message passing. Messages consist of the identity of the target object, the name of the requested operation and any other operation needed to perform the function. Message are often implemented as procedure or function calls.

iii. Abstraction

In object oriented design, complexity is managed using abstraction. Abstraction is the elimination of the irrelevant and the amplification of the essentials.

Software Design

iv. Class

In any system, there shall be number of objects. Some of the objects may have common characteristics and we can group the objects according to these characteristics. This type of grouping is known as a class. Hence, a class is a set of objects that share a common structure and a common behavior.

We may define a class “car” and each object that represent a car becomes an instance of this class. In this class “car”, Indica, Santro, Maruti, Indigo are instances of this class as shown in fig. 20.

Classes are useful because they act as a blueprint for objects. If we want a new square we may use the square class and simply fill in the particular details (i.e. colour and position) fig. 21 shows how can we represent the square class.

Software Design

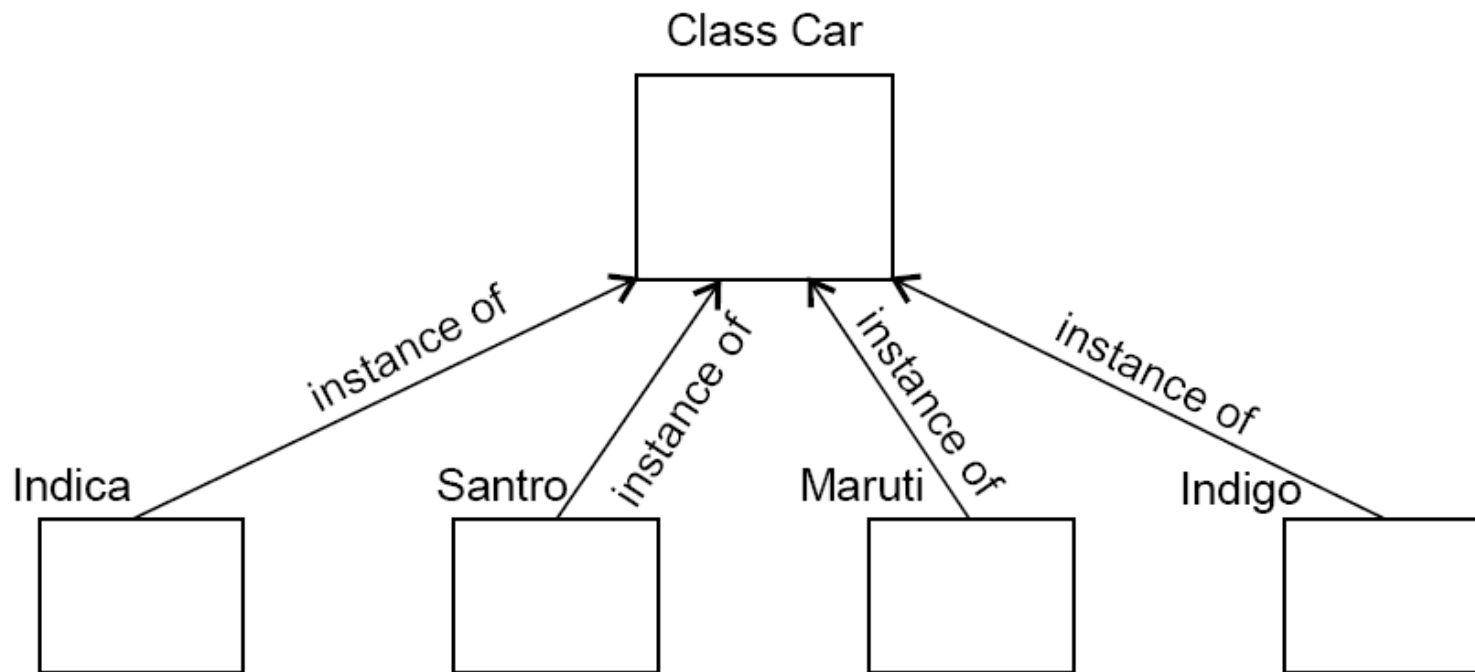


Fig.20: Indica, Santro, Maruti, Indigo are all instances of the class “car”

Software Design

Class Square

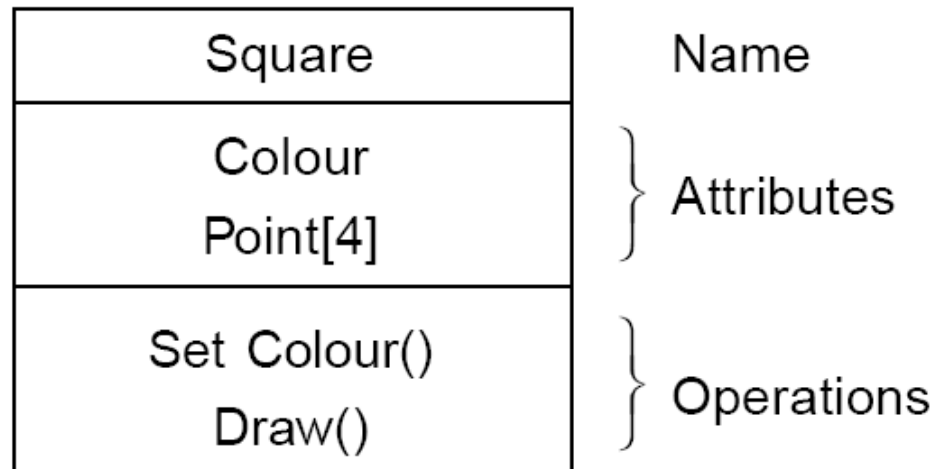


Fig. 21: The square class

Software Design

v. Attributes

An attributes is a data value held by the objects in a class. The square class has two attributes: a colour and array of points. Each attributes has a value for each object instance. The attributes are shown as second part of the class as shown in fig. 21.

vi. Operations

An operation is a function or transformation that may be applied to or by objects in a class. In the square class, we have two operations: set colour() and draw(). All objects in a class share the same operations. An object “knows” its class, and hence the right implementation of the operation. Operation are shown in the third part of the class as indicated in fig. 21.

Software Design

vii. Inheritance

Imagine that, as well as squares, we have triangle class. Fig. 22 shows the class for a triangle.

Class Triangle

Triangle
Colour Point[3]
Set Colour() Draw()

Fig. 22: The triangle class

Software Design

Now, comparing fig. 21 and 22, we can see that there is some difference between triangle and squares classes.

For example, at a high level of abstraction, we might want to think of a picture as made up of shapes and to draw the picture, we draw each shape in turn. We want to eliminate the irrelevant details: we do not care that one shape is a square and the other is a triangle as long as both can draw themselves.

To do this, we consider the important parts out of these classes in to a new class called Shape. Fig. 23 shows the results.

Software Design

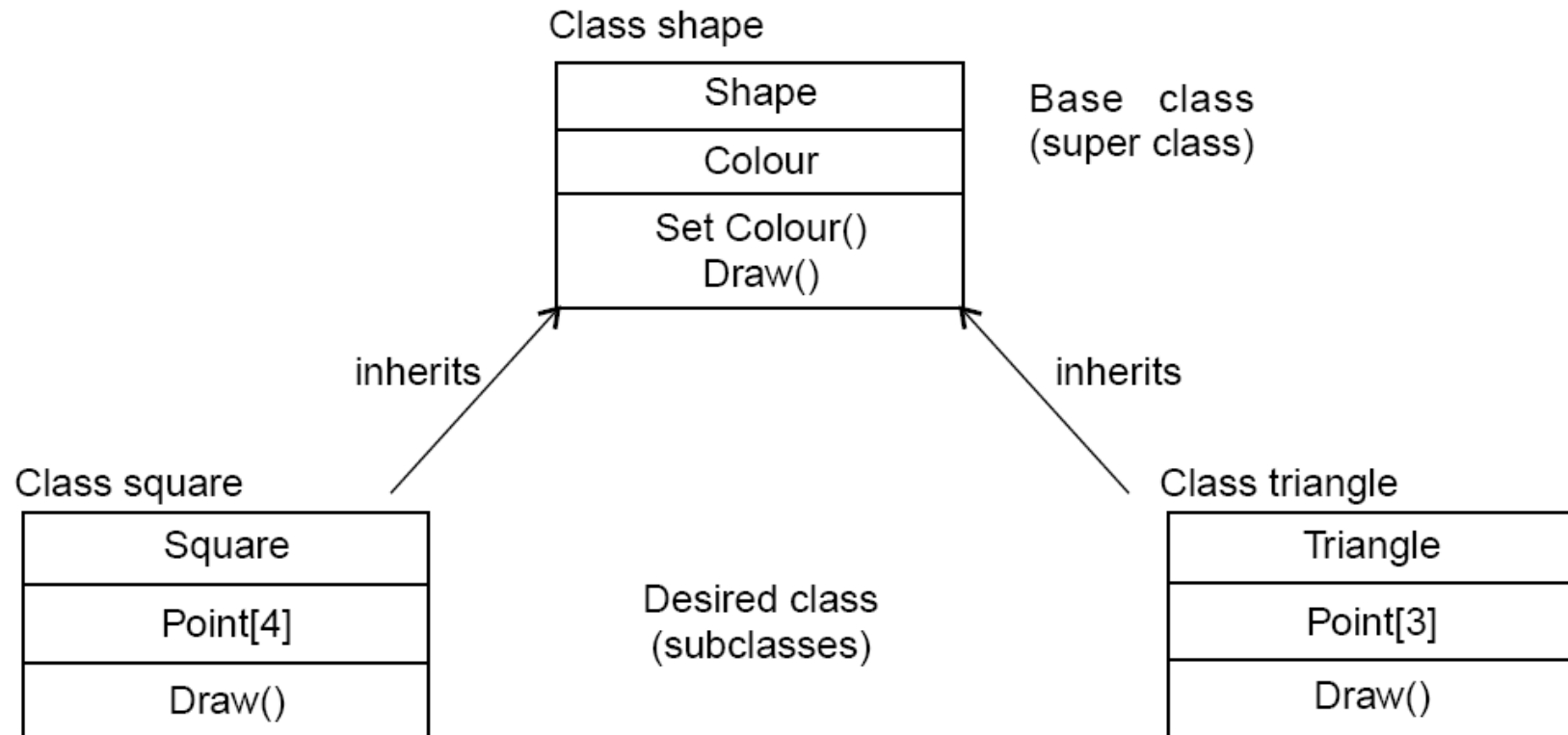


Fig. 23: Abstracting common features in a new class

This sort of abstraction is called inheritance. The low level classes (known as subclasses or derived classes) inherit state and behavior from this high level class (known as a super class or base class).

Software Design

viii. Polymorphism

When we abstract just the interface of an operation and leave the implementation to subclasses it is called a polymorphic operation and process is called polymorphism.

ix. Encapsulation (Information Hiding)

Encapsulation is also commonly referred to as “Information Hiding”. It consists of the separation of the external aspects of an object from the internal implementation details of the object.

x. Hierarchy

Hierarchy involves organizing something according to some particular order or rank. It is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way.

Software Design

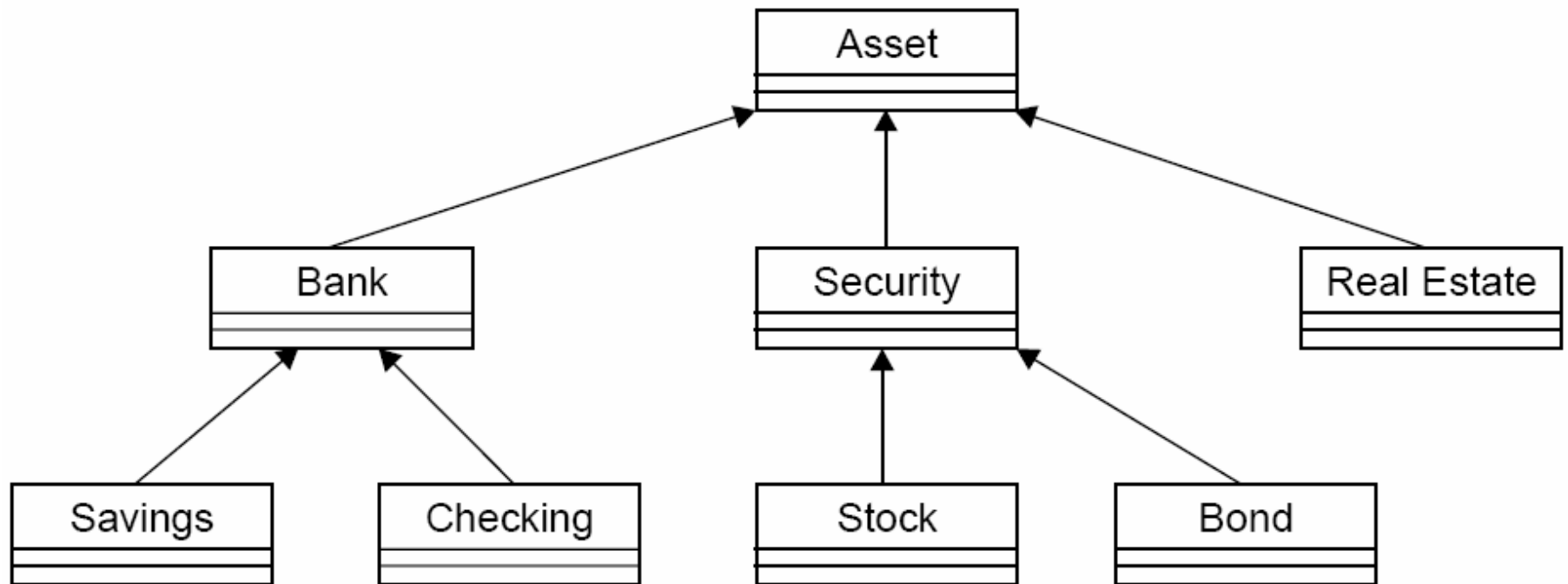


Fig. 24: Hierarchy

Software Design

➤ Steps to Analyze and Design Object Oriented System

There are various steps in the analysis and design of an object oriented system and are given in fig. 25

Software Design

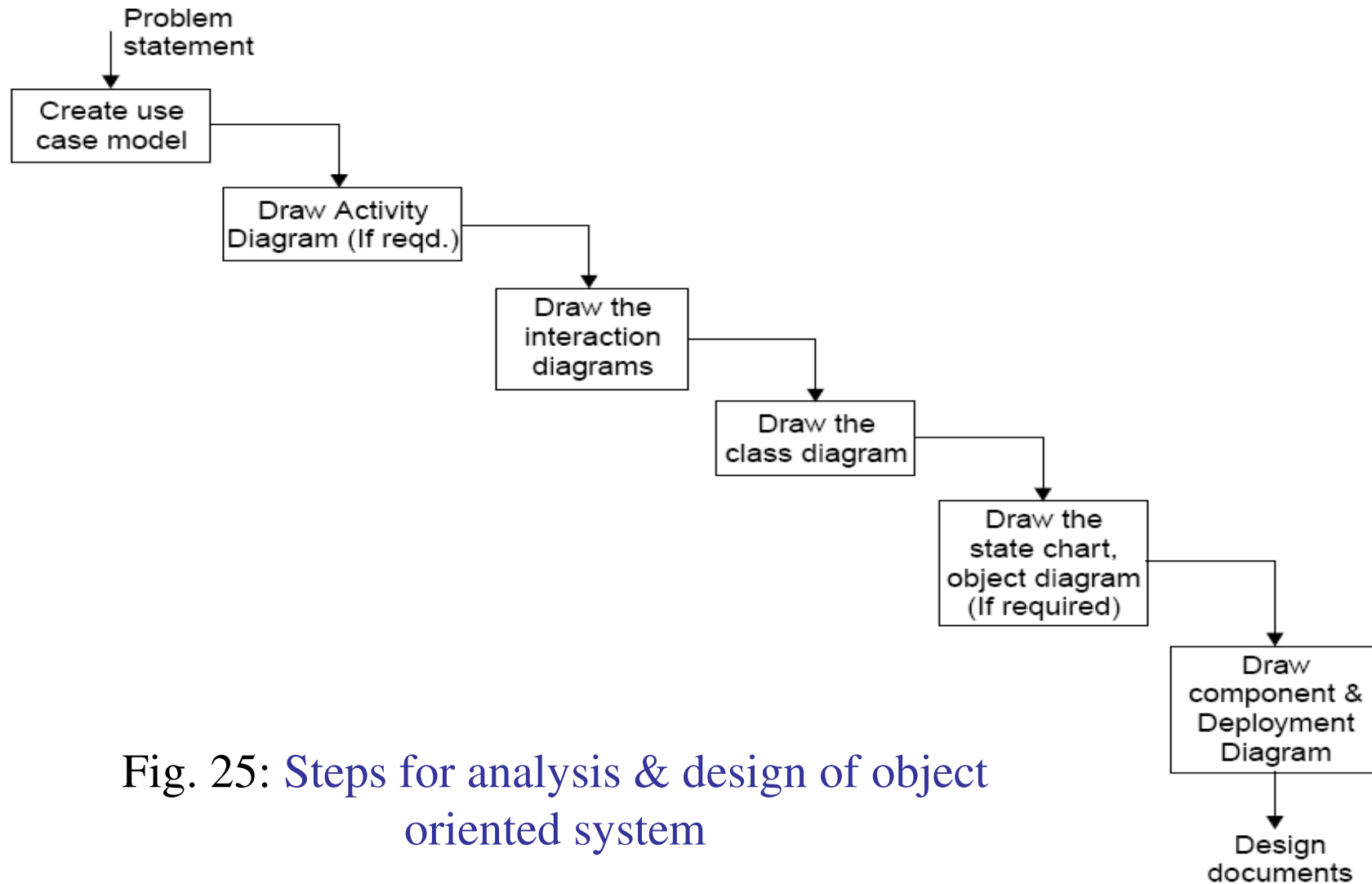


Fig. 25: Steps for analysis & design of object oriented system