

NARASIMHAREDDYENGINEERINGCOLLEGE

FACULTY:Dr.VenkateswaruluNaik

CS3102PC:SOFTWAREENGINEERING

III YEARB.TECH. CSEI -SEM (R21)



UNIT-I

INTRODUCTION TO SOFTWARE ENGINEERING

Software:

- ▶ collection of integrated programs.
- ▶ carefully-organized instructions that provide desired features, function, and performance, when executed.
- ▶ Computer program and related documentations such as requirements, design models and user manuals.

Characteristic of Software:

- ▶ Software is developed or engineered; it is not manufactured in the classical sense.
- ▶ Software does not “wear out” (not susceptible to environment effects).
- ▶ Reusability of components.

Engineering:

- ▶ application of scientific and practical knowledge to invent, design, build, maintain, and improve frameworks, processes, etc.

Software Engineering:

- ▶ The systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.
- ▶ includes activities to manage the project, develop tools, methods and theories that support the software production.
- ▶ provides a standard procedure to design and develop software.

EVOLVING ROLE OF SOFTWARE:

- ▶ Software takes dual role.
- ▶ **product:**
 - ▶ It delivers the computing potential embodied by computer hardware or by a network of computers.
- ▶ **vehicle:**
 - ▶ It provides system functionality (e.g., payroll system)
 - ▶ It controls other software (e.g., an operating system)
 - ▶ It helps build other software (e.g., software tools)



Evolution:

- ▶ 1970sand1980s
- ▶ 1990sbegan
- ▶ Mid-1990s
- ▶ Later1990s
- ▶ 2000sprogressed
- ▶ Todayahugesoftwareindustryhasbecomeadominantfactorintheeconomiesoftheindustrialized world.

THECHANGINGNATURE OFSOFTWARE:

- ▶ Systemsoftware
- ▶ Applicationsoftware
- ▶ Engineering/scientificsoftware
- ▶ Embeddedsoftware
- ▶ Product-linesoftware
- ▶ Web-applications
- ▶ Artificialintelligencesoftware.



Systemsoftware:

- ▶ collection of programs written to service other programs.
- ▶ *E.g.* compilers, editors and file management utilities.

Applicationsoftware:

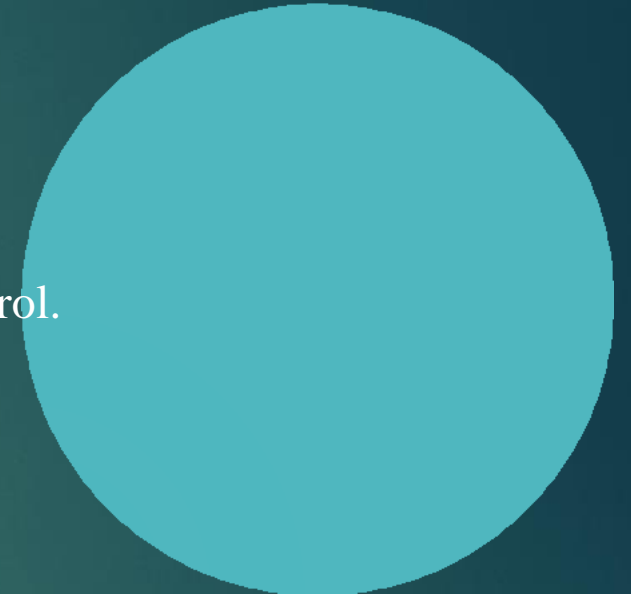
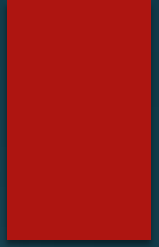
- ▶ standalone programs that solve a specific business need.
- ▶ *E.g.* point-of-sale transaction processing, real-time manufacturing process control.

Engineering/Scientificsoftware:

- ▶ from astronomy to volcanology
- ▶ *E.g.* computer aided design, systems simulation and other interactive applications.

Embedded software:

- ▶ resides within a product or system
- ▶ *E.g.* Digital functions in automobile, dashboard displays, braking system etc.



Product-line software:

- ▶ provide a specific capability for use by many different customers
- ▶ *E.g.* Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications

Web-applications:

- ▶ evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

Artificial intelligence software:

- ▶ make use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis.
- ▶ *E.g.* robotics, expert systems, pattern recognition, artificial neural networks, theorem proving, and game playing.

New challenges on the horizon:

1. Ubiquitous computing

2. Netsourcing



your roots to success

SOFTWARE MYTHS

Management myths:

- ▶ Manages with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

Customer myths:

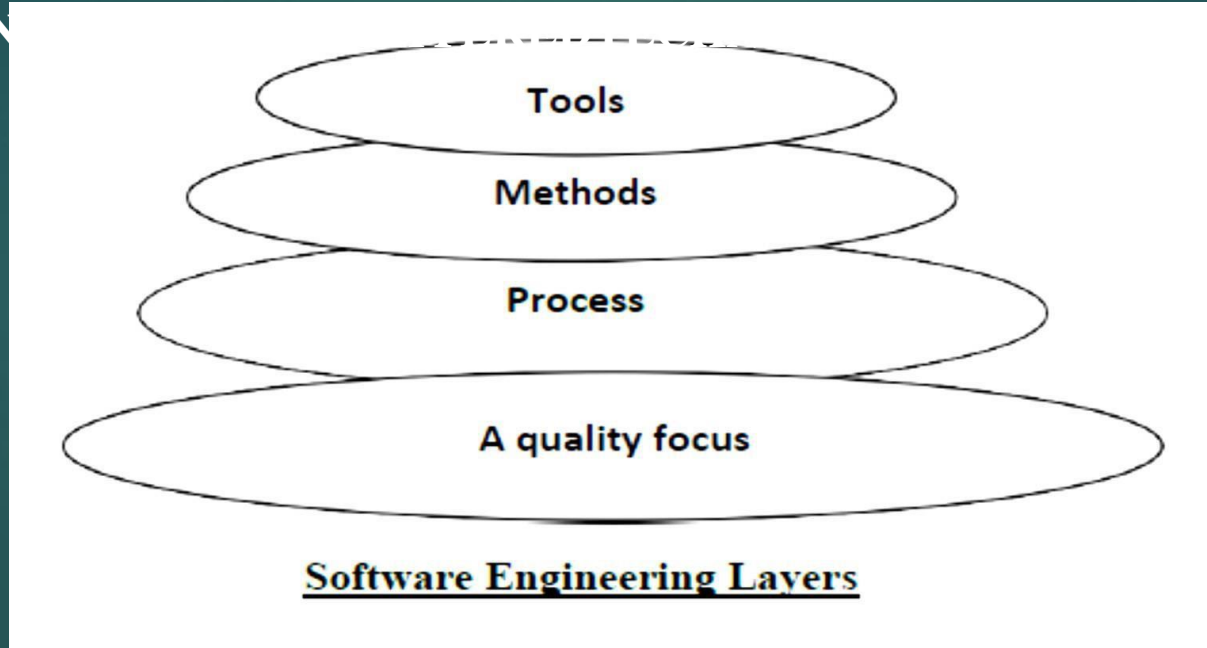
- ▶ The customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations and ultimately, dissatisfaction with the developer.

Practitioner's myths:

- ▶ Myths that are still believed by software practitioners: during the early days of software, programming was viewed as an art from old ways and attitudes die hard.

A GENERIC VIEW OF PROCESS

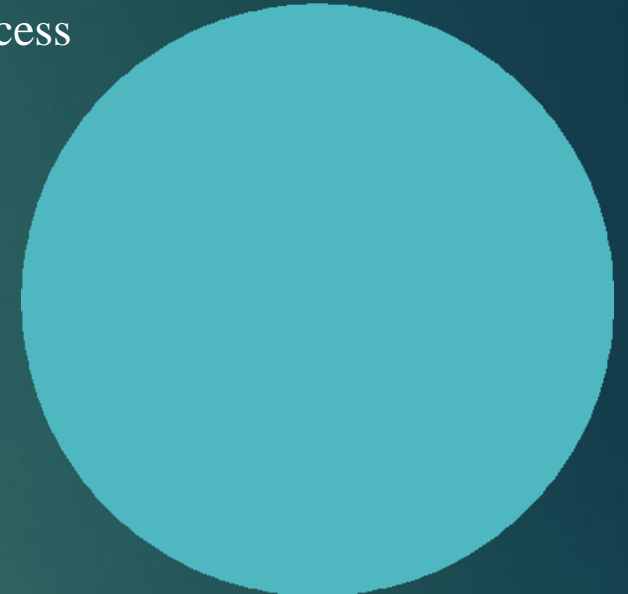
SOFTWARE ENGINEERING



- ▶ The bedrock that supports software engineering is a quality focus.
- ▶ The foundation for software engineering is the process layer.
- ▶ Software engineering methods rely on a set of basic principles that govern a area of the technology and include modeling activities.
- ▶ Software engineering tools provide automated or semi-automated support for the process and the methods.

A PROCESS FRAMEWORK:

- ▶ A process defines who is doing what, when, and how to reach a certain goal.
- ▶ A Process Framework establishes the foundation for a complete software process
- ▶ identifies a small number of **framework activities**
- ▶ also, set of **umbrella activities**
- ▶ Each framework activity has set of s/w engineering actions.
- ▶ Each s/w engineering action (e.g., design) has
 - ▶ collection of related tasks (called task sets):
 - ▶ work tasks
 - ▶ work products (deliverables)
 - ▶ quality assurance points
 - ▶ project milestones.



Process framework

Umbrella activities

Framework activity #1

Software engineering action

Task sets

- Work tasks
- Work products
- Quality assurance points
- Project milestones

Software engineering action

- Work tasks
- Work products
- Quality assurance points
- Project milestones

Framework activity #n

Software engineering action

Task sets

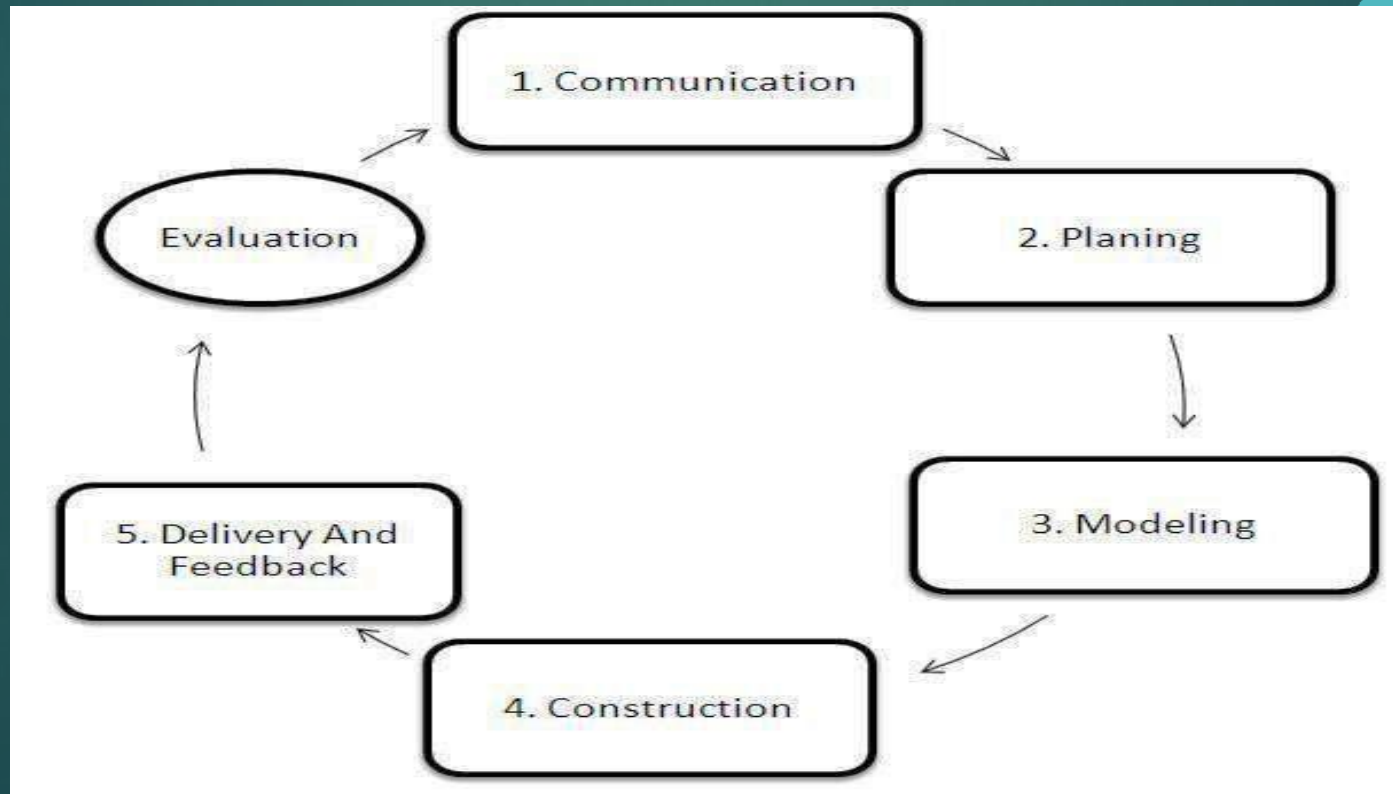
- Work tasks
- Work products
- Quality assurance points
- Project milestones

Software engineering action

- Work tasks
- Work products
- Quality assurance points
- Project milestones

Generic Process Framework:

- ▶ It is applicable to the vast majority of software projects.
- ▶ These 5 generic framework activities can be used during the development of small programs, the creation of large web applications, and for the engineering of large, complex computer-based systems.



Communication:

- ▶ involves communication and collaboration with the customer and encompasses requirements gathering and other related activities.

Planning:

- ▶ establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling:

- ▶ encompasses the creation of models that allow the developer and customer to better understand software requirements
- ▶ The modeling activity is composed of 2 software engineering actions - analysis and design.
 - ▶ Analysis encompasses a set of work tasks.
 - ▶ Design encompasses work tasks that create a design model.

Construction:

- ▶ combines code generation and the testing that is required to uncover the errors in the code.

Deployment:

- ▶ The software is delivered to the customer who evaluates the delivered product and provides feedback



your roots to success

The following are
these set of **Umbrella Activities**. **Software project tracking and control**

- ▶ assess progress against the project plan and take necessary action to maintain schedule.

Risk Management

- ▶ assess risks that may affect the outcome of the project or the quality of the product.

Software Quality Assurance

- ▶ define and conduct the activities required to ensure software quality.

Formal Technical Reviews

- ▶ assess software engineering work products in an effort to uncover and remove errors.

Measurement

- ▶ define and collect process, project and product measures that assist the team in delivering software that meets customer's needs

Software configuration management

- ▶ manage the effects of change throughout the software process.

Reusability management

- ▶ define criteria for work product reuse and establish mechanisms to achieve reusable components.

Work Product preparation and production



your roots to success

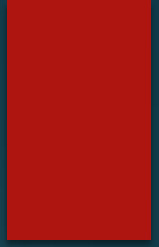
THE CAPABILITY MATURITY MODEL INTEGRATION (CMMI):

- ▶ The CMMI represents a process meta-model in two different ways:
 - ▶ As a continuous model
 - ▶ As a staged model.

Each process area is formally assessed and is rated according to the following capability levels.

- ▶ **Level 0: Incomplete**
- ▶ **Level 1: Performed**
- ▶ **Level 2: Managed**
- ▶ **Level 3: Defined**
- ▶ **Level 4: Quantitatively managed**
- ▶ **Level 5: Optimized**

The CMMI defines each process area in terms of “specific goals” and the “specific practices” required to achieve these goals.



The specific goals(SG)andtheassociatedspecificpractices(SP)definedforprojectplanning are

SG1Establishestimates

- ▶ SP1.1Estimatethescopeoftheproject
- ▶ SP1.2 Establish estimates ofwork productandtaskattributes
- ▶ SP1.3Defineprojectlifecycle
- ▶ SP1.4 Determineestimatesof effortandcost

SG2Develop aProjectPlan

- ▶ SP2.1Establishthebudgetandschedule
- ▶ SP2.2Identifyprojectrisks
- ▶ SP2.3Planfordatamanagement
- ▶ SP2.4Planforneeded knowledgeandskills
- ▶ SP2.5Planstakeholderinvolvement
- ▶ SP2.6Establishtheprojectplan

SG3Obtaincommitmenttothe plan

- ▶ SP3.1Reviewplansthataffecttheproject
- ▶ SP3.2Reconcileworkandresourcelevels
- ▶ SP3.3Obtainplancommitment



- ▶ In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area.
- ▶ Each of the five generic goals corresponds to one of the five capability levels.

GG 1 Achieve specific goals

- ▶ GP1.1 Perform base practices

GG2 Institutionalize a managed process

- ▶ GP2.1 Establish an organizational policy
- ▶ GP2.2 Plan the process
- ▶ GP2.3 Provide resources
- ▶ GP2.4 Assign responsibility
- ▶ GP2.5 Train people
- ▶ GP2.6 Manage configurations
- ▶ GP2.7 Identify and involve relevant stakeholders
- ▶ GP2.8 Monitor and control the process
- ▶ GP2.9 Objectively evaluate adherence
- ▶ GP2.10 Review status with higher level management



GG3 Institutionalize a defined process

- ▶ GP3.1 Establish a defined process
- ▶ GP3.2 Collect improvement information

GG 4 Institutionalize a quantitatively managed process

- ▶ GP4.1 Establish quantitative objectives for the process
- ▶ GP4.2 Stabilize sub process performance

GG5 Institutionalize and optimizing process

- ▶ GP5.1 Ensure continuous process improvement
- ▶ GP5.2 Correct root causes of problems

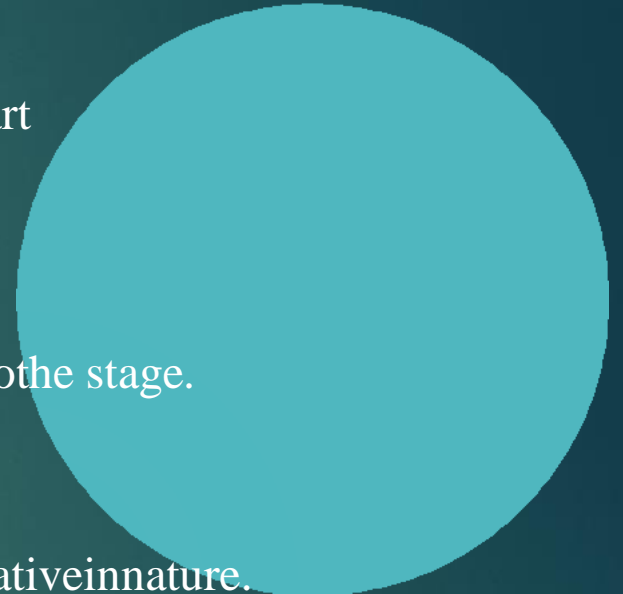
PROCESS PATTERNS:

- ▶ The software process can be defined as a collection of patterns that define a set of activities, actions, work tasks, work products and/or related behaviors required to develop computer software.
- ▶ provides us with a template - a consistent method for describing an important characteristic of the software process.



A pattern might be used to describe a complete process and a task within a framework activity.

- ▶ PatternName
- ▶ Intent
- ▶ Type: 3 types
 - ▶ **Task patterns** define a software engineering action or work task that is part of the process and relevant to successful software engineering practice.
 - ▶ *Example:* Requirement Gathering
 - ▶ **Stage Patterns** define a framework activity for the process. This pattern incorporates multiple task patterns that are relevant to the stage.
 - ▶ *Example:* Communication
 - ▶ **Phase patterns** define the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature.
 - ▶ *Example:* Spiral model or prototyping.
- ▶ InitialContext
- ▶ Problem
- ▶ Solution
- ▶ ResultingContext

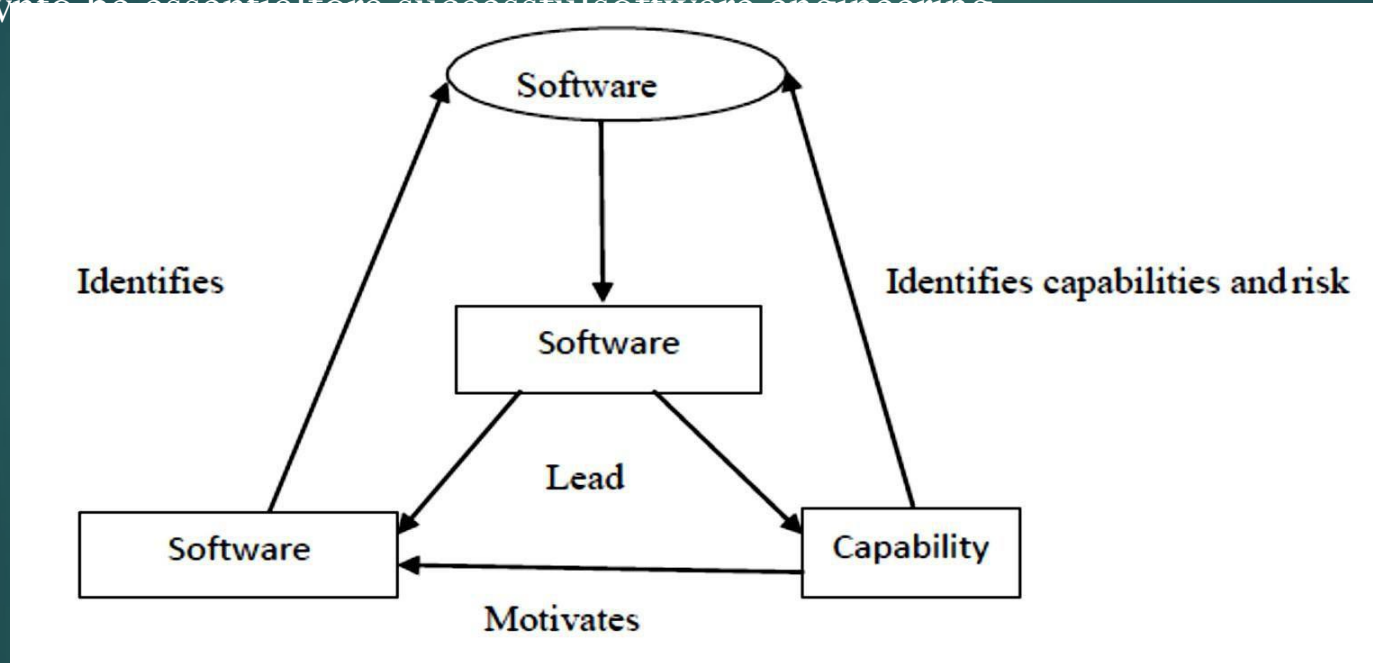




your roots to success

PROCESSASSESSMENT

- ▶ the process itself should be assessed to be essential to ensure that it meets a set of basic process criteria that have been shown to be essential for successful software engineering



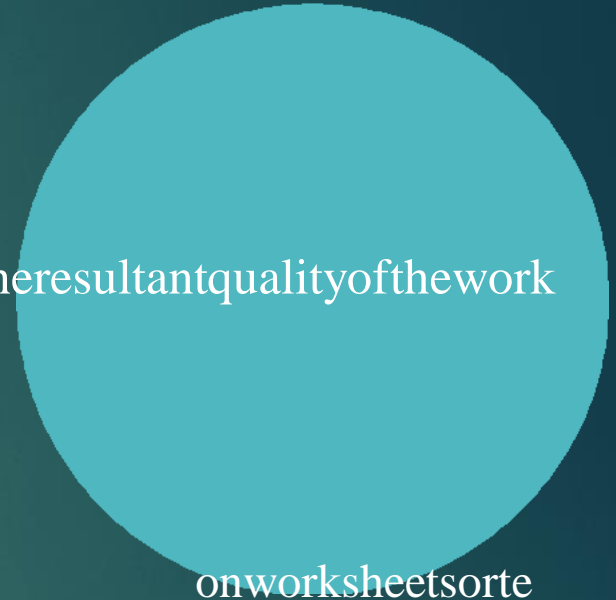
- ▶ Number of different approaches to software process assessment have been proposed,
 - ▶ Standards CMMI Assessment Method for Process Improvement (SCAMPI)
 - ▶ CMM Based Appraisal for Internal Process Improvement (CBAIPI)
 - ▶ SPICE (ISO/IEC 15504)
 - ▶ ISO 9001:2000 for Software.

PERSONAL AND TEAM PROCESS MODELS:

- ▶ Each software engineer would create a process that best fits his or her needs, and at the same time meets the broader needs of the team and the organization.

Personal software process (PSP)

- ▶ emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product.
- ▶ The PSP process model defines five framework activities
 - ▶ **Planning:** This activity isolates requirements and all the metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.
 - ▶ **High level design:** External specifications for each component to be constructed are developed and a component design is created. All issues are recorded and tracked.
 - ▶ **High level design review:** Formal verification methods are applied to uncover errors in the design.
 - ▶ **Development:** The component level design is refined and reviewed. Code is generated, reviewed, compiled,





...te



your roots to success

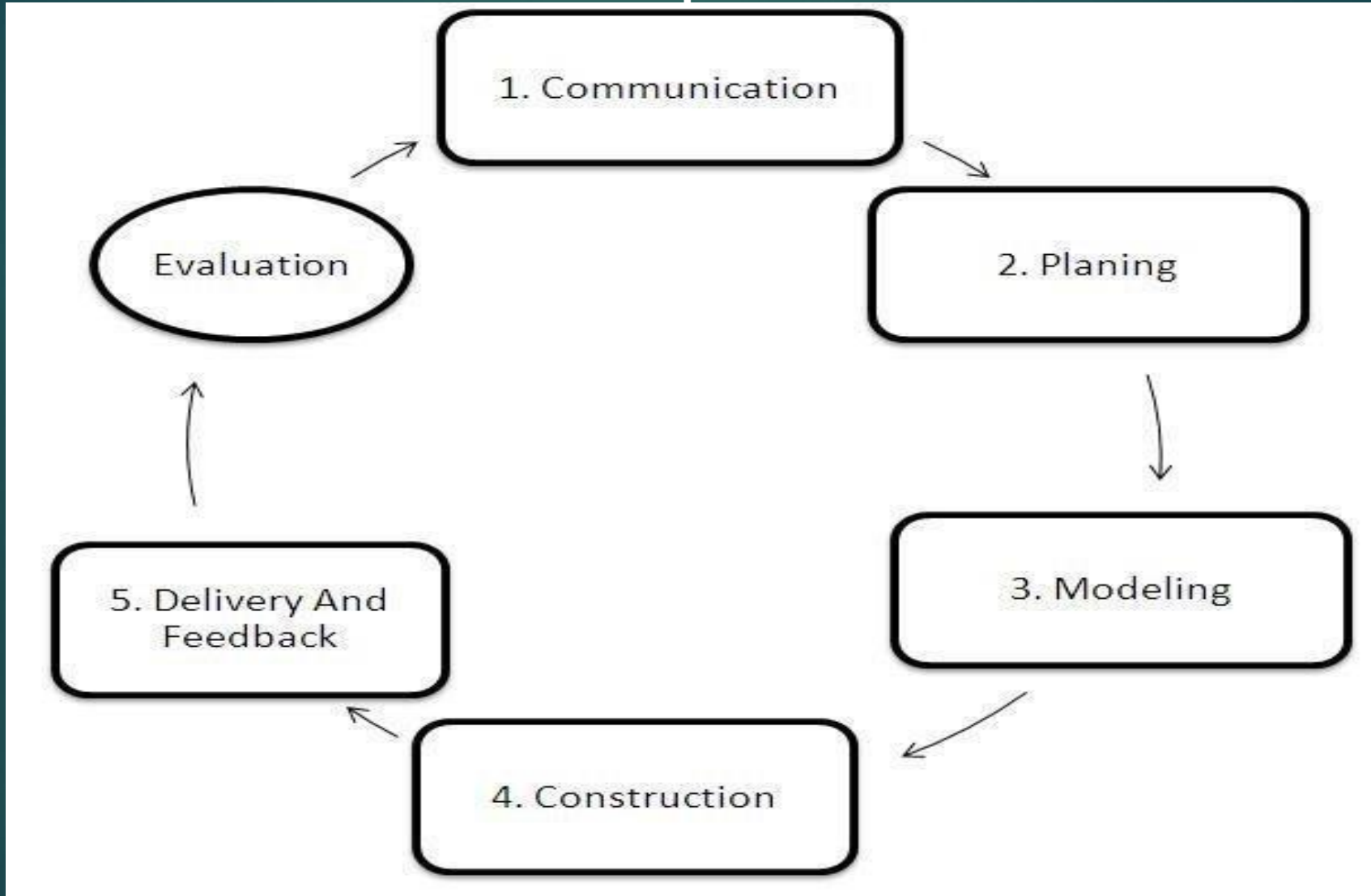
Team software process (TSP)

- ▶ The goal of TSP is to build a “self-directed project team that organizes itself to produce high-quality software.
- ▶ TSP defines the following framework activities:
 - ▶ **launch**
 - ▶ **high-level design**
 - ▶ **implementation**
 - ▶ **integration and test**
 - ▶ **postmortem.**

PROCESS MODELS:

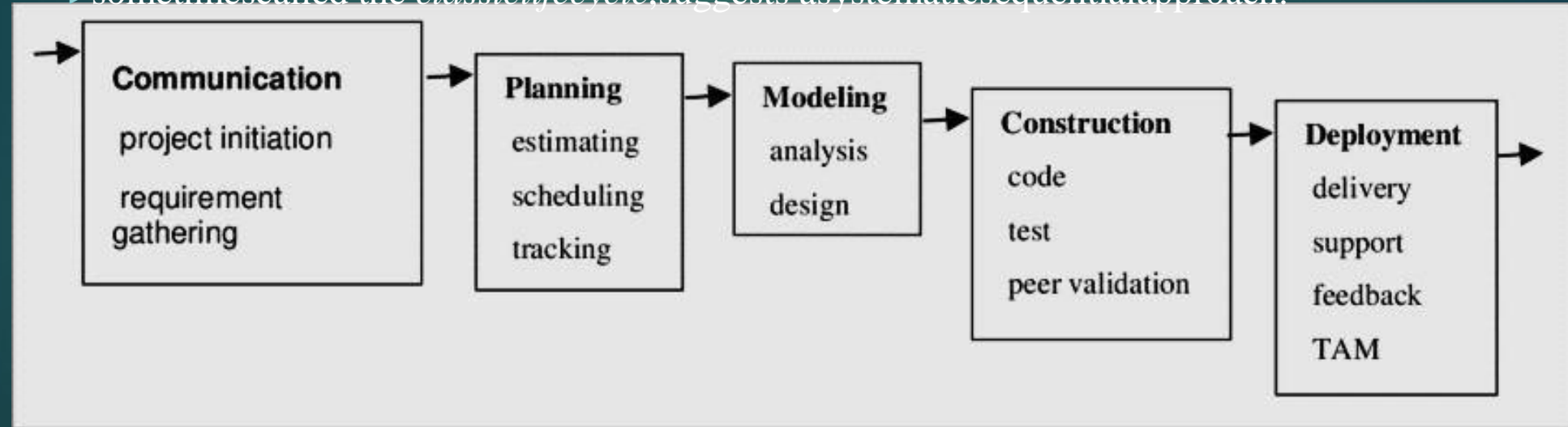
- ▶ **Prescriptive process models** define a set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software.
- ▶ **Software Development Life Cycle (SDLC)** is a process used by the software industry to design, develop and test high quality software.

SDLCphases



THE WATERFALL MODEL:

- ▶ sometimes called the *classic lifecycle*, suggests a systematic sequential approach.



Adv:

- ▶ useful process model where requirements are fixed and work is to proceed to complete in a linear manner.

Disadv:

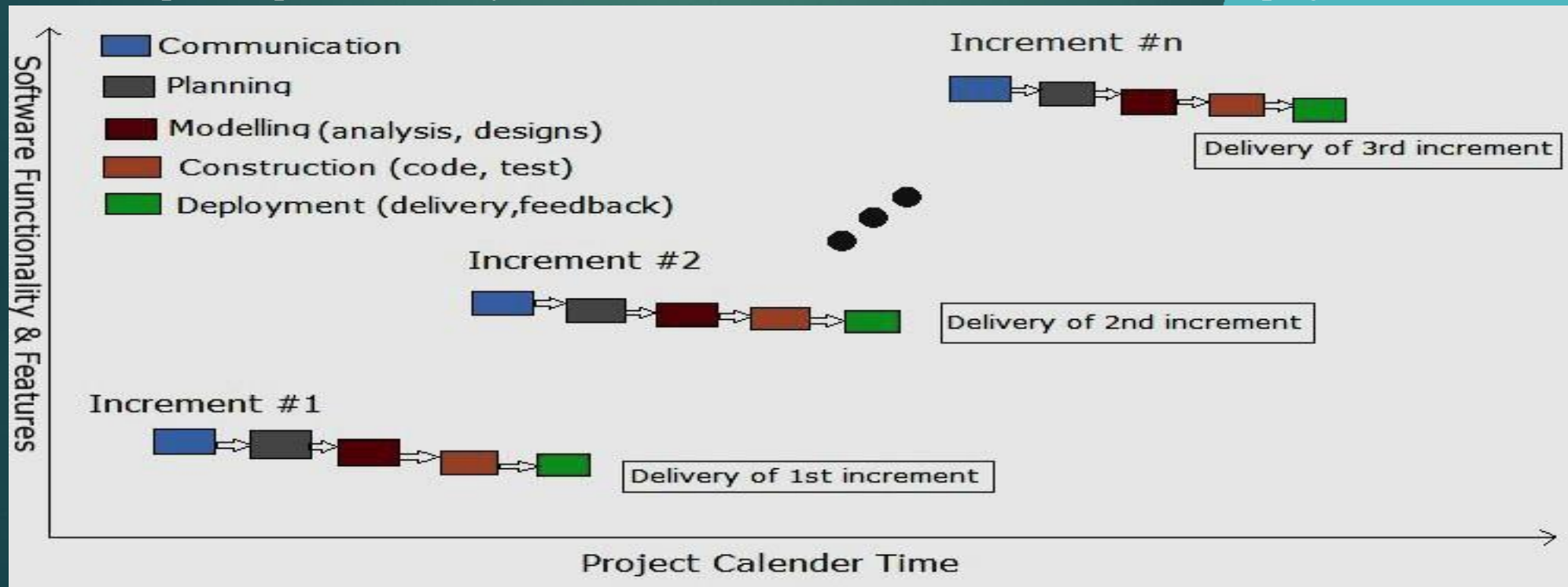
- ▶ changes can cause confusion as the project team proceeds.
- ▶ It is often difficult for the customer to state all requirements explicitly.
- ▶ A working version of the program will not be available until late in the project time-span.

INCREMENTAL PROCESS MODELS:

- ▶ The incremental model
- ▶ The RAD model

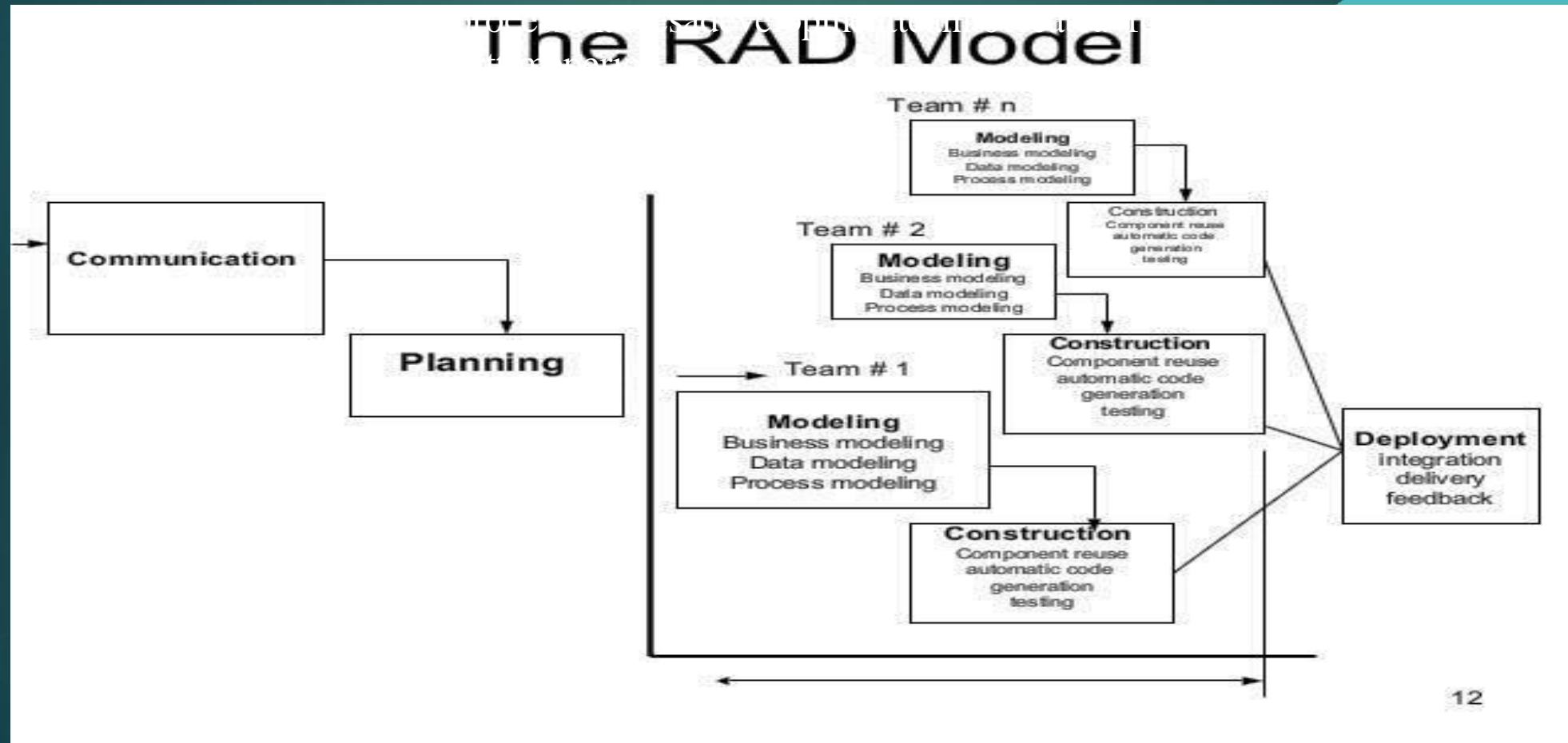
THE INCREMENTAL MODEL:

- ▶ Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.



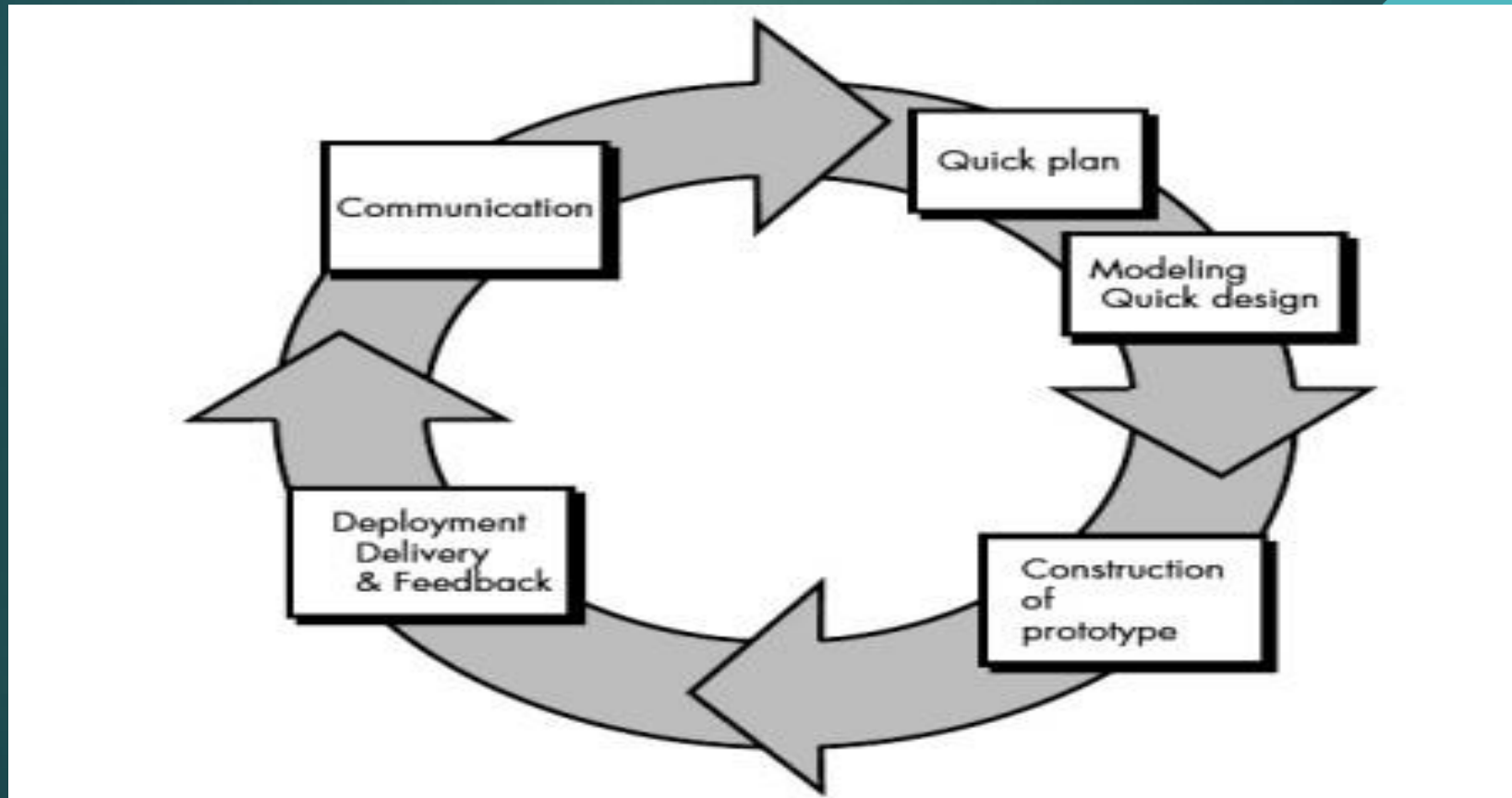
THE RAD MODEL:

- ▶ a “high-speed” adaption of the waterfall model, in which rapid development is achieved by using a component base construction approach.
- ▶ If requirements are well understood and project scope



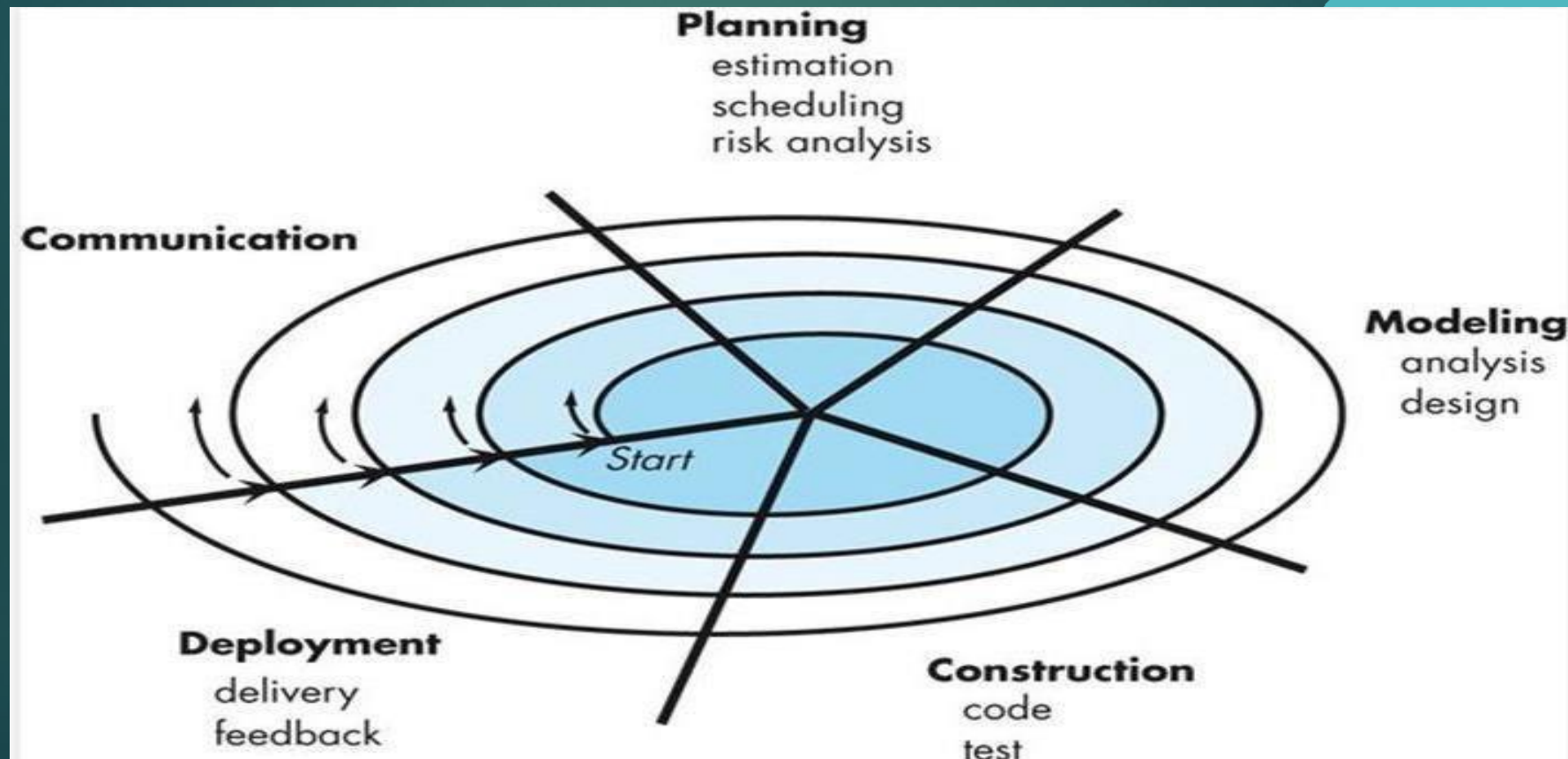
EVOLUTIONARY PROCESS MODELS: PROTOTYPING:

- ▶ can be implemented within the context of any one of the process model.
- ▶ Prototyping iteration is planned quickly and modeling occurs. The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer/user.



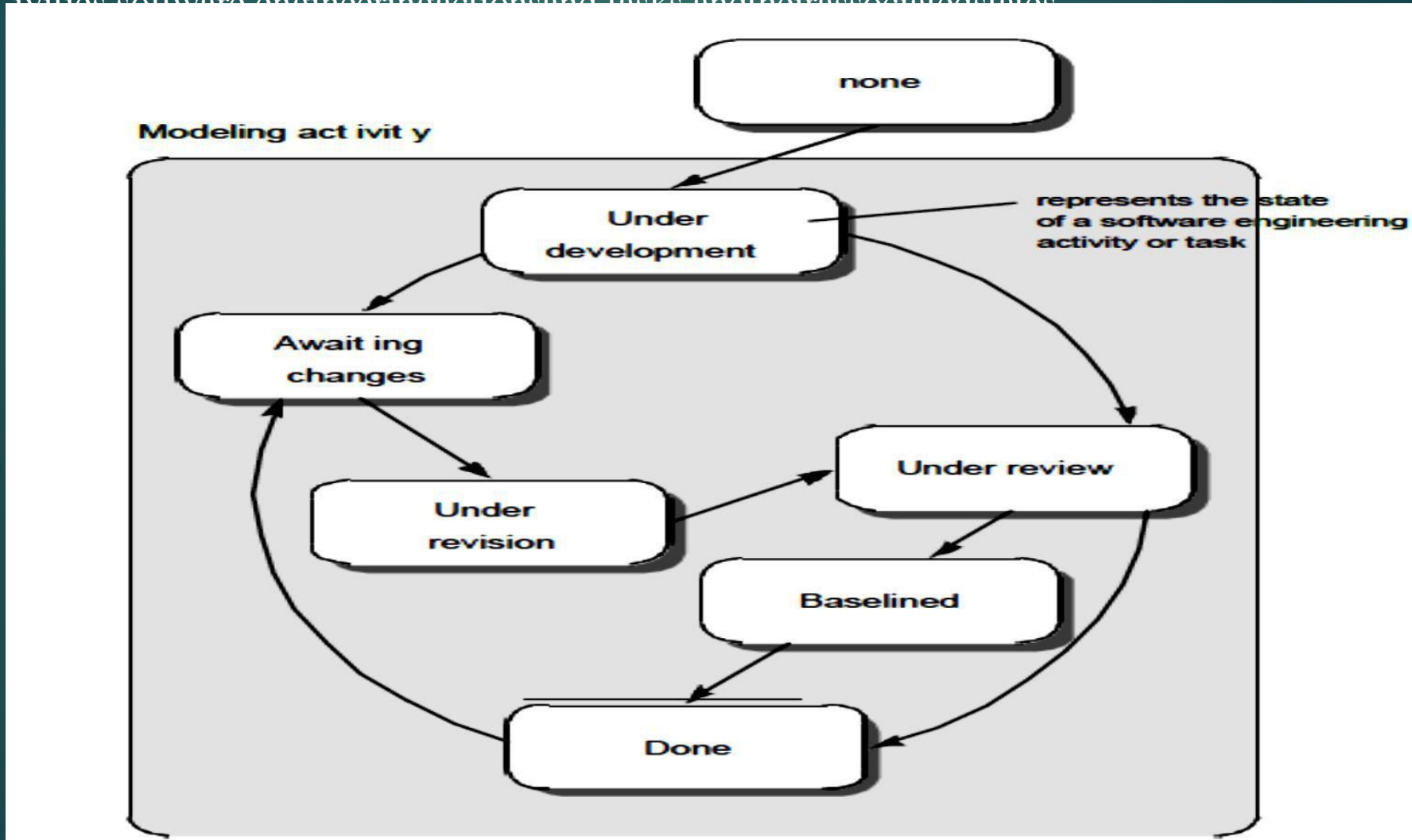
THE SPIRAL MODEL

- ▶ couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- ▶ adapted to apply throughout the entire lifecycle of an application, from concept development to maintenance.



THE CONCURRENT DEVELOPMENT MODEL:

- ▶ sometimes called *concurrent engineering*, can be represented schematically as a series of framework activities, software engineering activities and tasks, and their associated states.

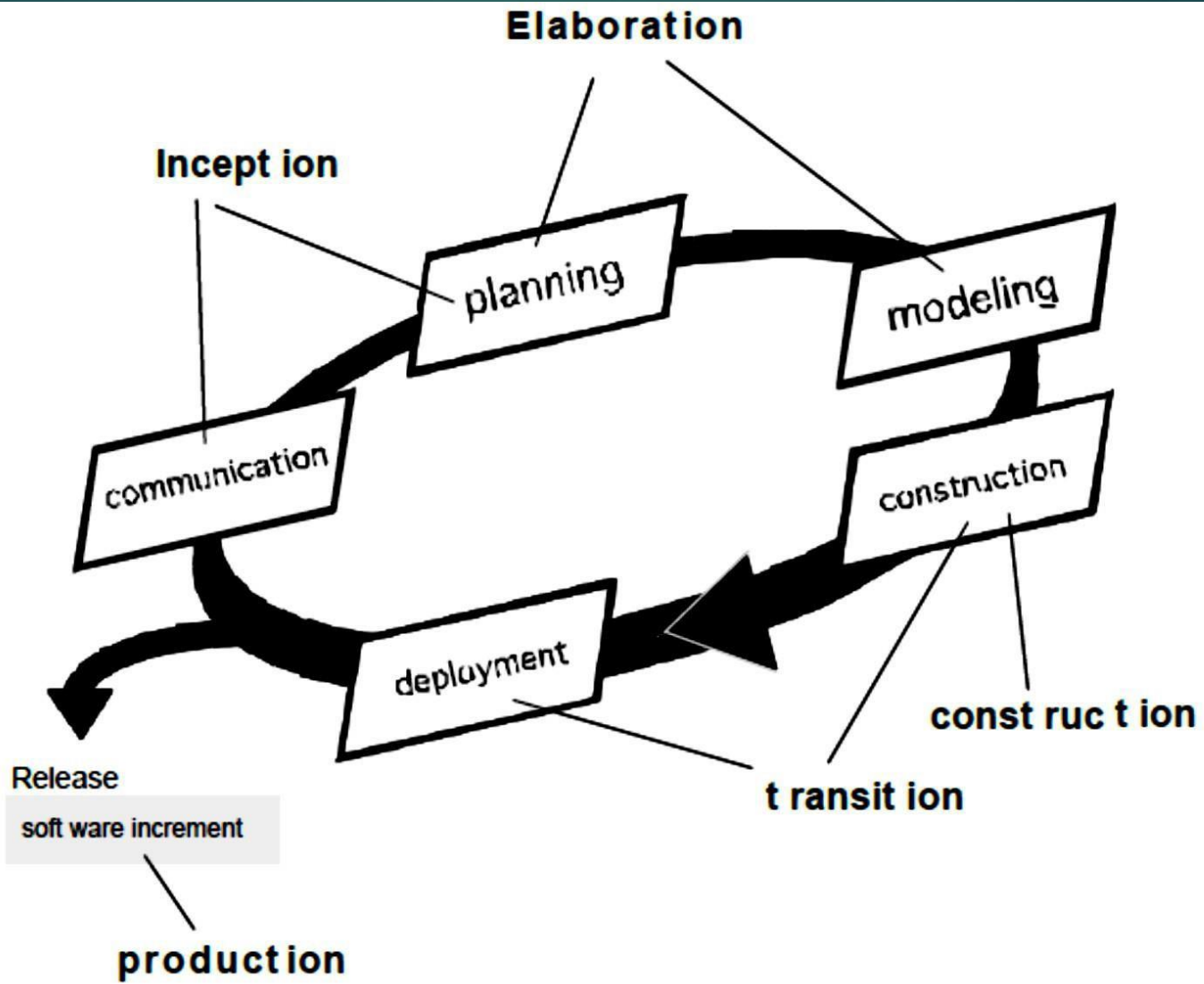


THE UNIFIED PROCESS:

- ▶ implements many of the best principles of agile software development.
- ▶ recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system.

PHASES OF THE UNIFIED PROCESS:

- ▶ The *inception* phase -encompasses both customer communication and planning activities.
- ▶ The *elaboration* phase-encompasses the customer communication and modeling activities of the generic process model.
- ▶ The *construction* phase-identical to the construction activity defined for the generic software process.
- ▶ The *transition* phase-
encompasses the latter stages of the generic construction activity and the first part of the generic deployment activity.
- ▶ The *production* phase -coincides with the deployment activity of the generic process.




NARASIMHAREDDYENGINEERINGCOLLEGE

FACULTY:Dr.VenkateswaruluNaik

CS3102PC:SOFTWAREENGINEERING

III YEARB.TECH. CSE I-SEM (R20)





▶ **Software Requirements:** Functional and non-functional requirements, user requirements, system requirements, interfaces specification, the software requirements document.

▶ **Requirements engineering process:** Feasibility studies, requirements elicitation and analysis, requirements validation, requirements management.

▶ **System models:** Context models, behavioral models, data models, object models, structured methods.

SOFTWARE REQUIREMENTS

Software requirements are necessary

- ▶ To introduce the concepts of user and system requirements
- ▶ To describe functional and non-functional requirements
- ▶ To explain how software requirements may be organized

What is a requirement?

- ▶ the description of the services provided by the system and its operational constraints
- ▶ It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification

Requirements engineering:

- ▶ The process of finding out, analysing, documenting and checking these services and constraints.
- ▶ The requirements themselves are the description of the system services and constraints that are generated during the requirements engineering process

Types of requirement:

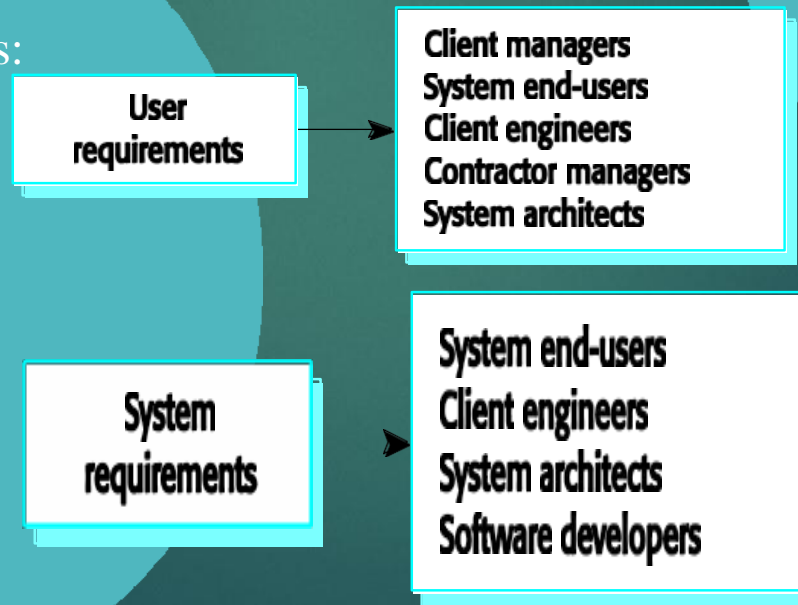
▶ User requirements

- ▶ Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.

▶ System requirements

- ▶ A structured document setting out detailed descriptions of the system's functions, services and operational constraints.
- ▶ Defines what should be implemented so may be part of a contract between client and contractor.

Requirements readers:



Functional and non-functional requirements:

▶ **Functional requirements**

- ▶ Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.

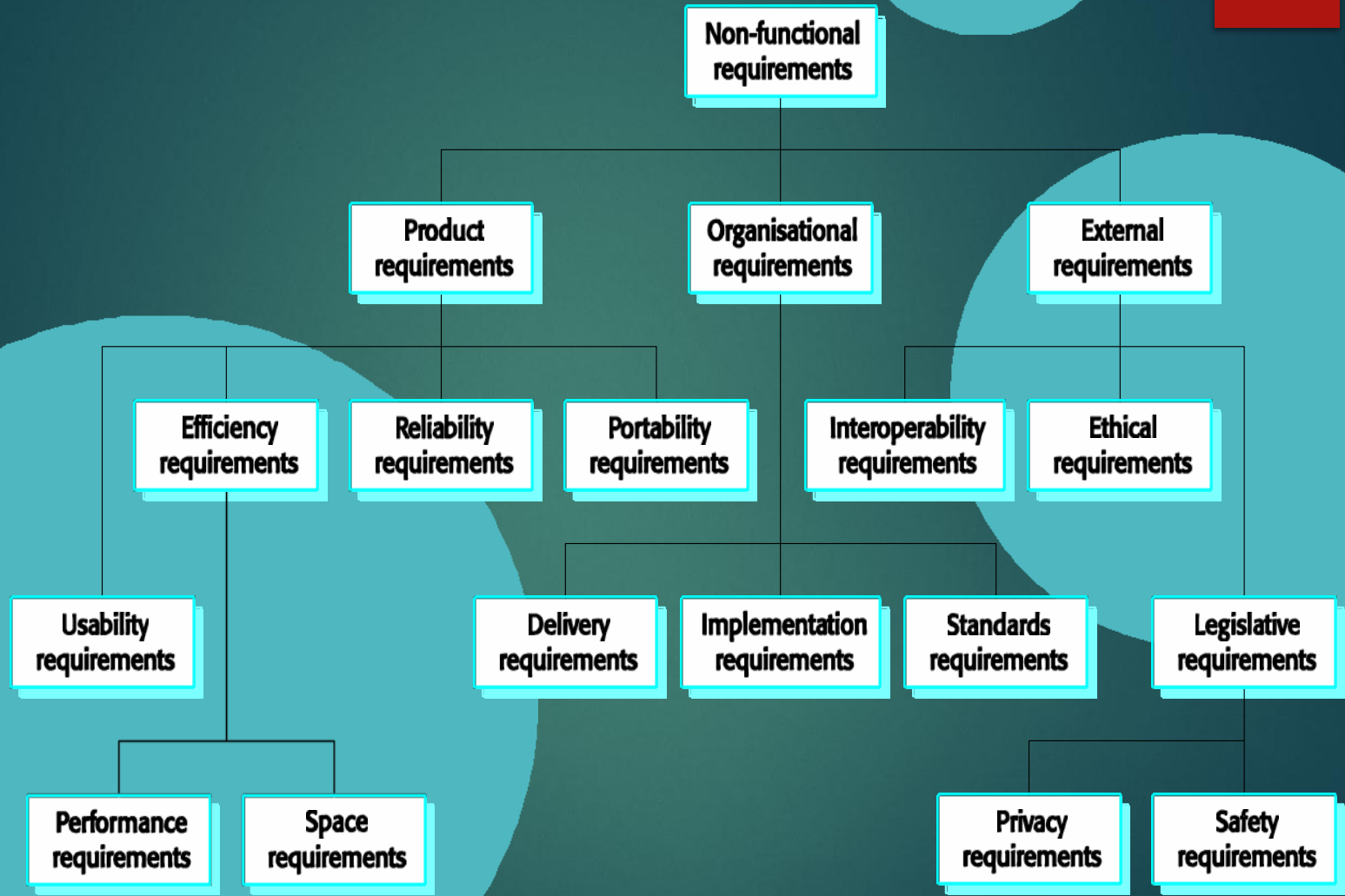
Examples of functional requirements

- ▶ The users shall be able to search either all of the initial set of databases or select a subset from it.
- ▶ The system shall provide appropriate viewers for the user to read documents in the document store.
- ▶ Every order shall be allocated a unique identifier (ORDER_ID) which the users shall be able to copy to the account's permanent storage area.

▶ **Non-functional requirements**

- ▶ Constraints on the services or functions offered by the systems such as timing constraints, constraints on the development process, standards, etc.
- ▶ Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non-functional requirement types:



Requirements measures:

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	MBytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

▶ Domain requirements

- ▶ Requirements that come from the application domain of the system and that reflect characteristics of that domain.

USER REQUIREMENTS

User requirements are defined using natural language, tables and diagrams as they can be understood by all users.

Problems with natural language

- ▶ Lack of clarity
 - ▶ Precision is difficult without making the document difficult to read.
- ▶ Requirements confusion
 - ▶ Functional and non-functional requirements tend to be mixed-up.
- ▶ Requirements amalgamation
 - ▶ Several different requirements may be expressed together.

▶ Guidelines for writing requirements

- ▶ Invent a standard format and use it for all requirements.
- ▶ Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- ▶ Use text highlighting to identify key parts of the requirement.
- ▶ Avoid the use of computer jargon.

SYSTEM REQUIREMENTS

- ▶ More detailed specifications of system functions, services and constraints than user requirements.
- ▶ They are intended to be a basis for designing the system.
- ▶ They may be incorporated into the system contract.
- ▶ System requirements may be defined or illustrated using system models

Problems with NL (natural language) specification

- ▶ Ambiguity
 - ▶ The readers and writers of the requirement must interpret the same words in the same way. NL is naturally ambiguous so this is very difficult.
- ▶ Over-flexibility
 - ▶ The same thing may be said in a number of different ways in the specification.
- ▶ Lack of modularization.
 - ▶ NL structures are inadequate to structure system requirements.

Alternatives to NL specification:

▶ **Structured language specifications**

- ▶ All requirements are written in a standard way.
- ▶ The terminology used in the description may be limited.
- ▶ The advantage is that the most of the expressiveness of natural language is maintained but a degree of uniformity is imposed on the specification.

▶ **Form-based specifications**

- ▶ Definition of the function or entity.
- ▶ Description of inputs and where they come from.
- ▶ Description of outputs and where they go to.
- ▶ Indication of other entities required.
- ▶ Pre and post conditions (if appropriate).
- ▶ The side effects (if any) of the function.



- ▶ **Tabular specification**

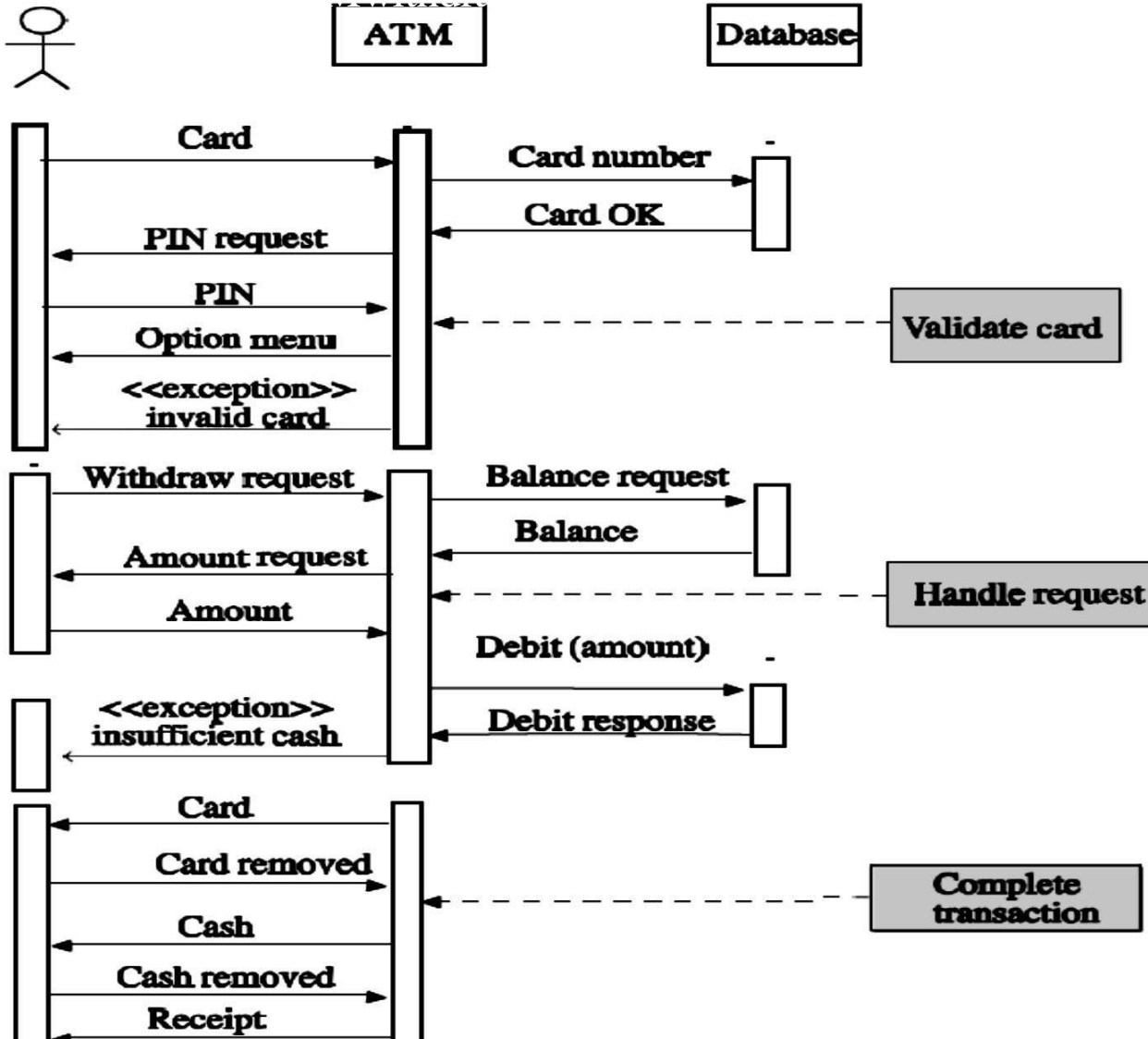
- ▶ Used to supplement natural language.
- ▶ Particularly useful when you have to define a number of possible alternative courses of action.

- ▶ **Graphical models**

- ▶ Graphical models are most useful when you need to show how state changes or where you need to describe a sequence of actions.

- ▶ **Sequenced diagrams**

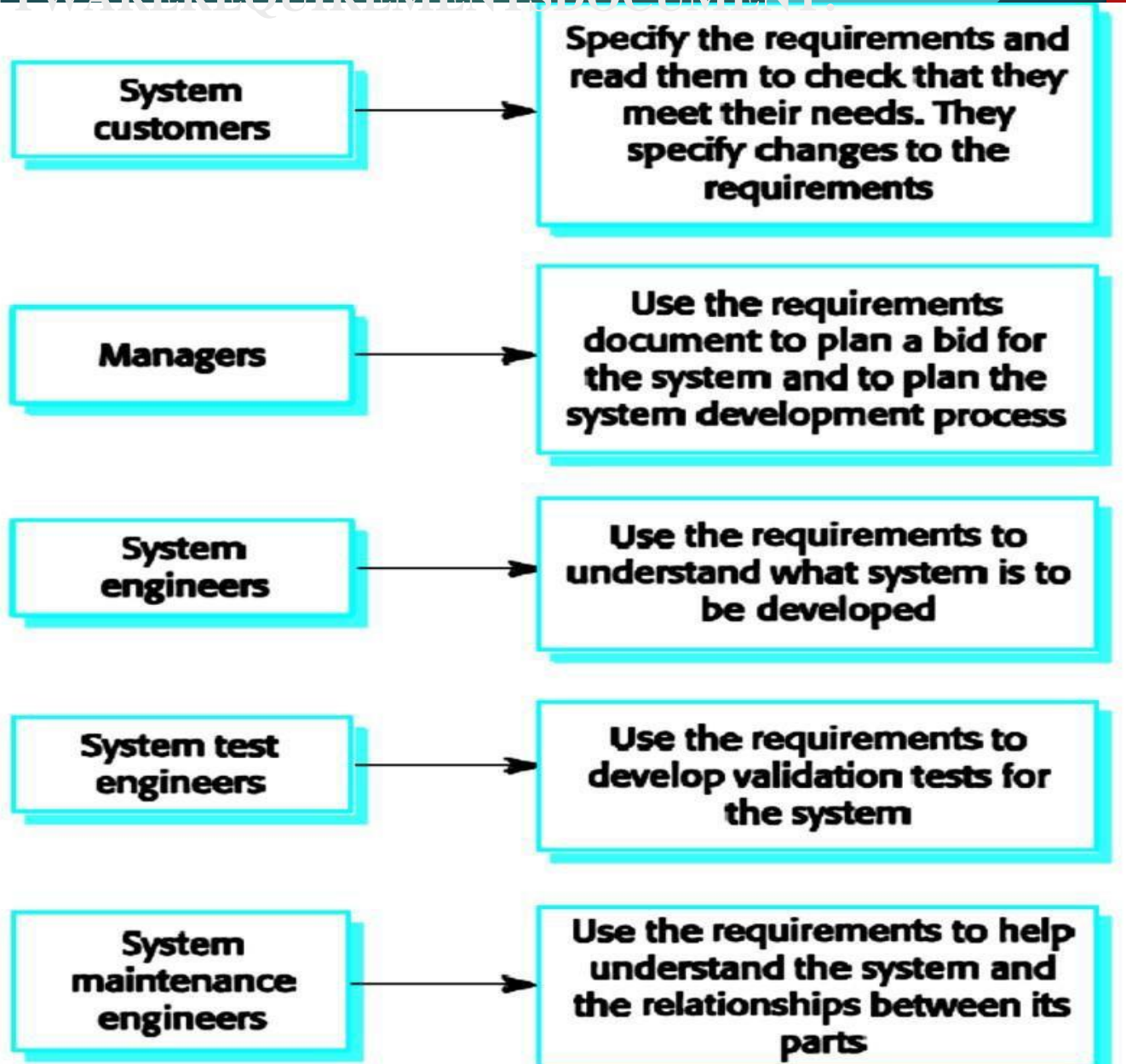
- ▶ These show the sequence of events that take place during some user interaction with a system.
- ▶ You read them from top to bottom to see the order of the actions that take place.
- ▶ Cash withdrawal from an ATM
 - ▶ Validate card;
 - ▶ Handle request;
 - ▶ Complete transaction.



INTERFACE SPECIFICATION

- ▶ Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements.
- ▶ Three types of interface may have to be defined
 - ▶ **Procedural interfaces** where existing programs or sub-systems offer a range of services that are accessed by calling interface procedures. These interfaces are sometimes called Application Programming Interfaces (APIs)
 - ▶ **Data structures that are exchanged** that are passed from one sub-system to another. Graphical data models are the best notations for this type of description
 - ▶ **Data representations** that have been established for an existing sub-system
- ▶ Formal notations are an effective technique for interface specification.

THE SOFTWARE REQUIREMENTS DOCUMENT:



▶ **IEEE requirements standard** defines a generic structure for a requirements document that must be instantiated for each specific system.

1. Introduction.

- i) Purpose of the requirements document
- ii) Scope of the project
- iii) Definitions, acronyms and abbreviations
- iv) References
- v) Overview of the remainder of the document

2. General description.

- i) Product perspective
- ii) Product functions
- iii) User characteristics
- iv) General constraints
- v) Assumptions and dependencies

3. Specific requirements cover functional, non-functional and interface requirements. The requirements may document external interfaces, describe system functionality and performance, specify logical database requirements, design constraints, emergent system properties and quality characteristics.

4. Appendices.

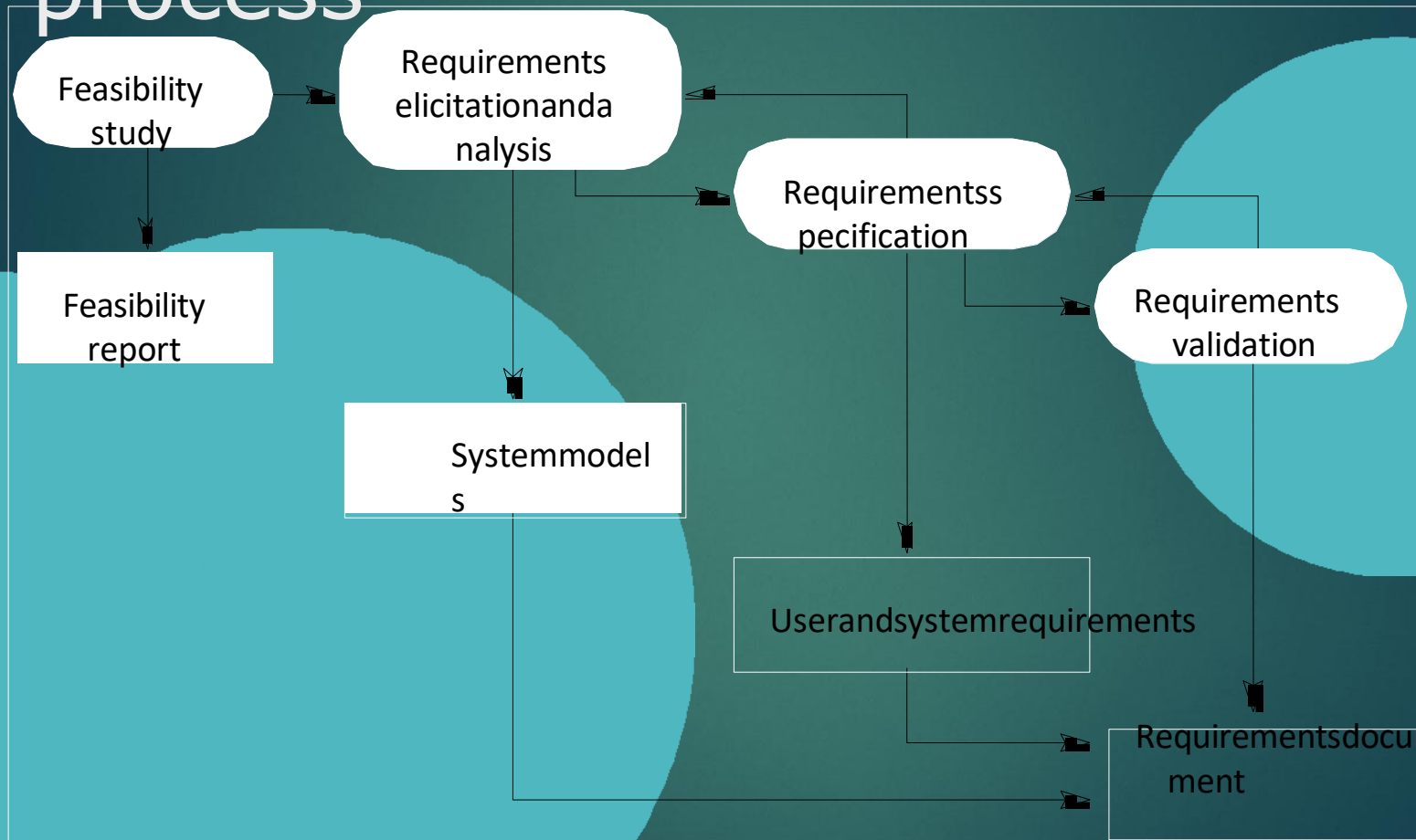


your roots to success.

REQUIREMENTS ENGINEERING PROCESS

- ▶ To create and maintain a system requirement document
- ▶ The overall process includes four high level requirements engineering sub-processes:
 - ▶ 1. Feasibility study
 - Concerned with assessing whether the system is useful to the business
 - ▶ 2. Elicitation and analysis
 - Discovering requirements
 - ▶ 3. Specifications
 - Converting the requirements into a standard form
 - ▶ 4. Validation
 - Checking that the requirements actually define the system that the customer wants

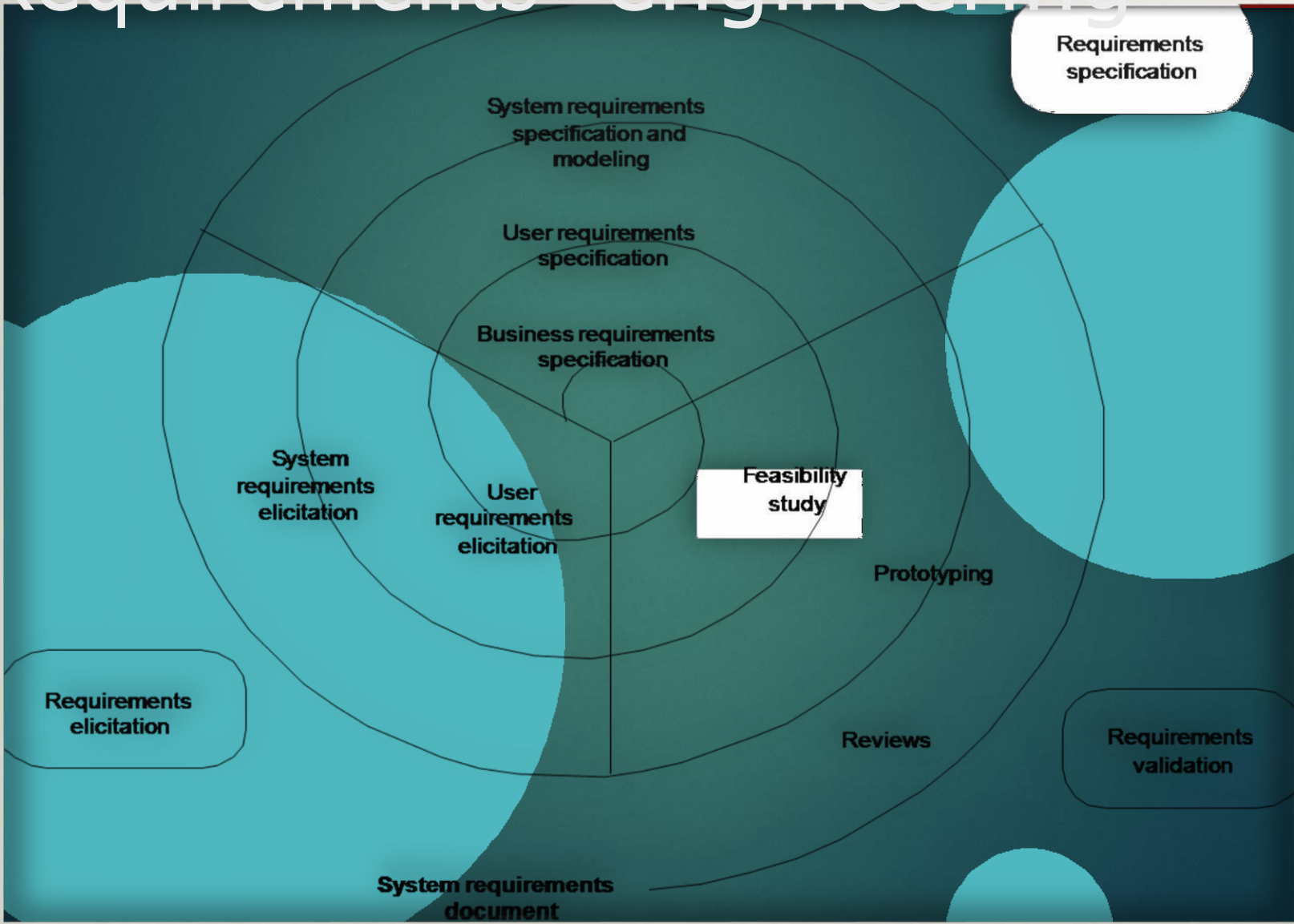
The requirements engineering process



SPIRAL REPRESENTATION OF REQUIREMENTS ENGINEERING PROCESS

- ▶ Process represented as three stage activity
- ▶ Activities are organized as an iterative process around a spiral.
- ▶ Early in the process, most effort will be spent on understanding high-level business and the user requirement.
- ▶ Later in the outer rings, more effort will be devoted to system requirements engineering and system modeling
- ▶ Three level process consist of:
 - ▶ 1. Requirements elicitation
 - ▶ 2. Requirements specification
 - ▶ 3. Requirements validation

Requirements engineering



FEASIBILITY STUDIES

- ▶ Starting point of the requirements engineering process
- ▶ Input: Set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes
- ▶ Output: Feasibility report that recommends whether or not it is worth carrying out further
 - ▶ Feasibility report answers a number of questions:
 - ▶ 1. Does the system contribute to the overall objective
 - ▶ 2. Can the system be implemented using the current technology and within given cost and schedule
 - ▶ 2. Can the system be integrated with other systems which are already in place.

REQUIREMENTS ELICITATION ANALYSIS

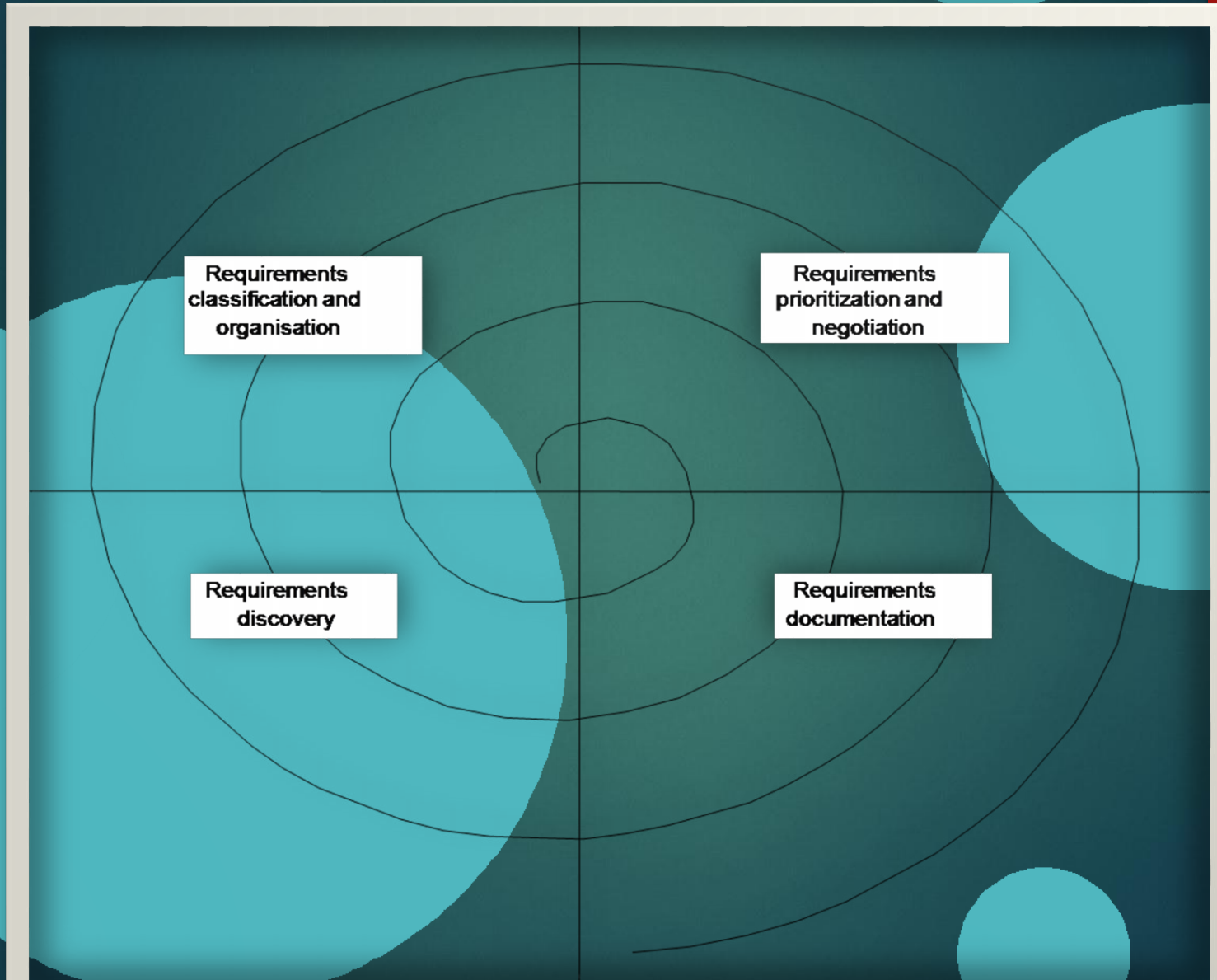
- ▶ Involves a number of people in an organization
- ▶ Stakeholder definition
Refer to any person or group who will be affected by the system directly or indirectly i.e. End users, Engineers, business managers, domain experts.
- ▶ Reasons why eliciting is difficult
 - ▶ 1. Stakeholder often don't know what they want from the computer system.
 - ▶ 2. Stakeholder expression of requirements in natural language is sometimes difficult to understand.
 - ▶ 3. Different stakeholder express requirements differently
 - ▶ 4. Influences of political factors
- ▶ Change in requirements due to dynamic environments.

REQUIREMENTS ELICITATION PROCESS

Process activities

- ▶ 1. Requirement Discovery
 - ▶ Interaction with stakeholder to collect their requirements including domain and documentation
- ▶ 2. Requirements classification and organization
 - ▶ Coherent clustering of requirements from unstructured collection of requirements
- ▶ 3. Requirements prioritization and negotiation
 - ▶ Assigning priority to requirements
 - ▶ Resolves conflicting requirements through negotiation
- ▶ 4. Requirements documentation
 - ▶ Requirements be documented and placed in the next round of spiral

The spiral representation of Requirements Engineering



REQUIREMENTS DISCOVERY TECHNIQUES

1. Viewpoints

- Based on the viewpoint expressed by the stakeholder

-

Recognizes multiple perspectives and provides a framework for discovering conflicts in the requirements proposed by different stakeholders

Three Generic types of viewpoints

1. Interactor viewpoint

- Represents people or other system that interact directly with the system

2. Indirect viewpoint

- Stakeholders who influence the requirements, but don't use the system

3. Domain viewpoint

- Requirements domain characteristics and constraints that influence the requirements.

REQUIREMENTS DISCOVERY TECHNIQUES

2. Interviewing

--Puts questions to stakeholders about the system stem to be developed.

that they use and the system

--Requirements are derived from the answers

❖ Two types of interview

- ▶ Closed interviews where the stakeholders answer a pre-defined set of questions.
- ▶ Open interviews discuss a range of issues with the stakeholders for better understanding their needs.

❖ Effective interviewers

- a) Open-minded: no pre-conceived ideas
- b) Prompter: prompt the interviewee to start discussion with a question or a proposal

REQUIREMENTS DISCOVERY TECHNIQUES

3. Scenarios

--Easiertorelatetoreallifeexamplesthantoabstractdescription

--
Startswithanoutlineoftheinteractionandduringelicitation,detailsareaddedtocreateacompletedescriptionofthatinteraction

--Scenarioincludes:

1. Descriptionatthestartofthesenario
2. Descriptionofnormalflowoftheevent
3. Descriptionofwhatcangowrongandhowthisishandled
4. Informationaboutotheractivitiesparalleltothescenario
5. Descriptionofthesystemstatewhenthescenariofinishes

LIBSYS scenario

- ▶ **Initial assumption:** The user has logged onto the LIBSYS system and has located the journal containing the copy of the article.
- ▶ **Normal:** The user selects the article to be copied. He or she is then prompted by the system to either provide subscriber information for the journal or to indicate how they will pay for the article. Alternative payment methods are by credit card or by quoting an organisational account number.
- ▶ The user is then asked to fill in a copyright form that maintains details of the transaction and they then submit this to the LIBSYS system.
- ▶ The copyright form is checked and, if OK, the PDF version of the article is downloaded to the LIBSYS working area on the user's computer and the user is informed that it is available. The user is asked to select a printer and a copy of the article is printed.

LIBSYS scenario

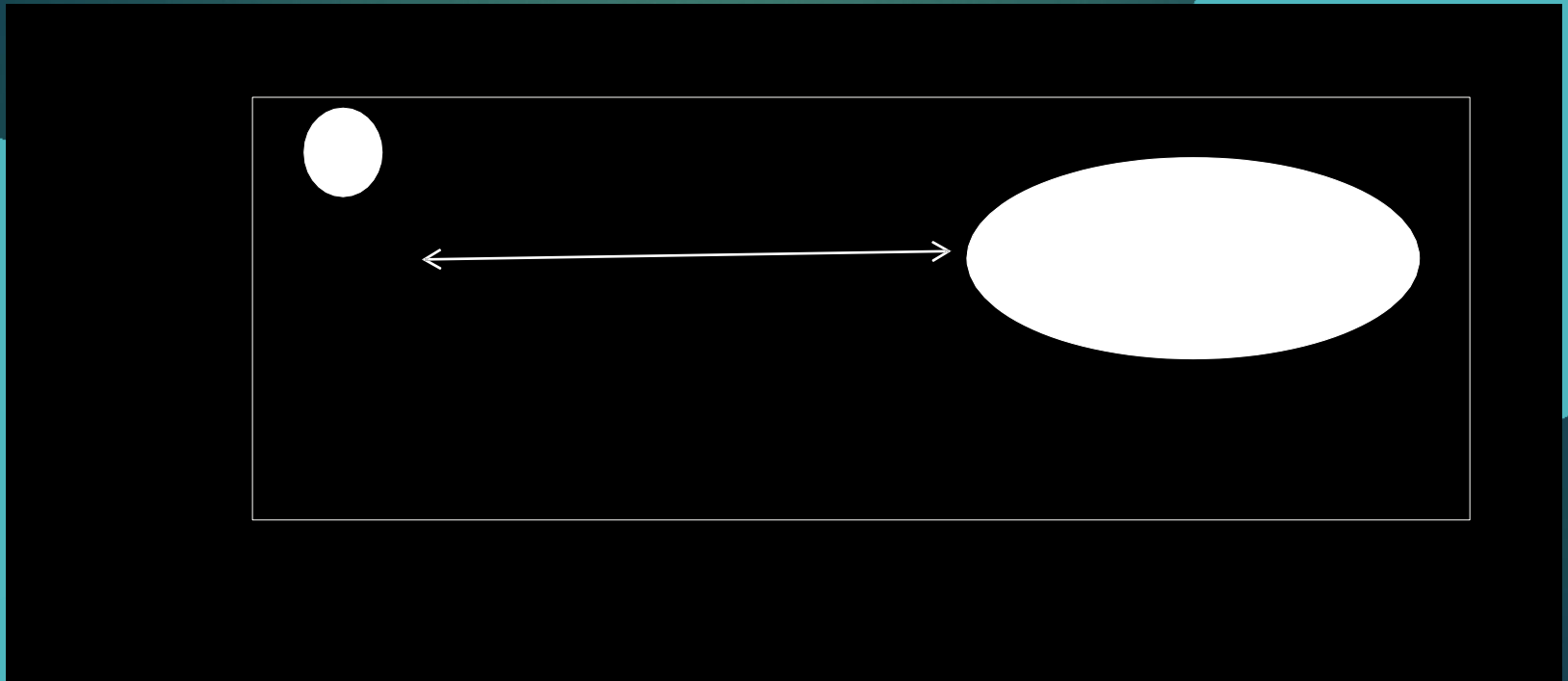
- ▶ **What can go wrong:** The user may fail to fill in the copyright form correctly. In this case, the form should be re-presented to the user for correction. If the resubmitted form is still incorrect then the user's request for the article is rejected.
- ▶ The payment may be rejected by the system. The user's request for the article is rejected.
- ▶ The article download may fail. Retry until successful or the user terminates the session..
- ▶ **Other activities:** Simultaneous downloads of other articles.
- ▶ **System state on completion:** User is logged on. The downloaded article has been deleted from LIBSYS workspace if it has been flagged as print-only.

REQUIREMENTS DISCOVERY TECHNIQUES

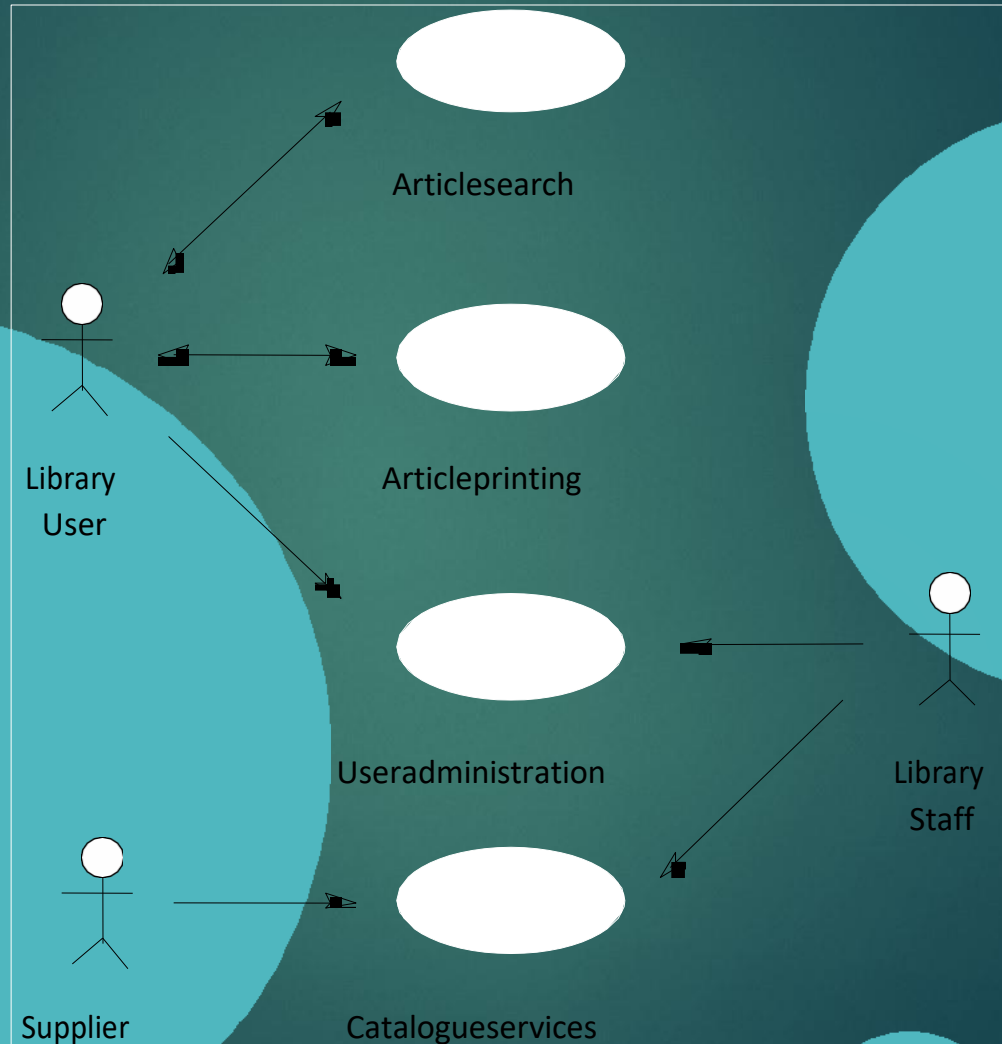
4. Usecases

- scenario based technique for requirement elicitation
- A fundamental feature of UML, notation for describing object-oriented system models
- Identifies a type of interaction and the actors involved
- Sequence diagrams are used to add information to a Use case

Articleprintinguse-case



LIBSYS use cases



REQUIREMENTS VALIDATION

- ▶ Concerned with showing that the requirements define the system that the customer wants.
- ▶ Important because errors in requirements can lead to extensive rework cost
- ▶ Validation checks
 1. Validity checks
 - Verification that the system performs the intended function by the user
 2. Consistency check
 - Requirements should not conflict
 3. Completeness checks
 - Includes requirements which define all functions and constraints intended by the system user
 4. Realism checks
 - Ensure that the requirements can be actually implemented
 5. Verifiability
 - Testable to avoid disputes between customer and developer.

VALIDATION TECHNIQUES

1. REQUIREMENTS REVIEWS

--Reviewers check the following:

- (a) Verifiability: Testable
- (b) Comprehensibility
- (c) Traceability
- (d) Adaptability

2. PROTOTYPING

3. TEST-CASE GENERATION

Requirements management

Requirements are likely to change for large software systems and as such requirements management process is required to handle changes.

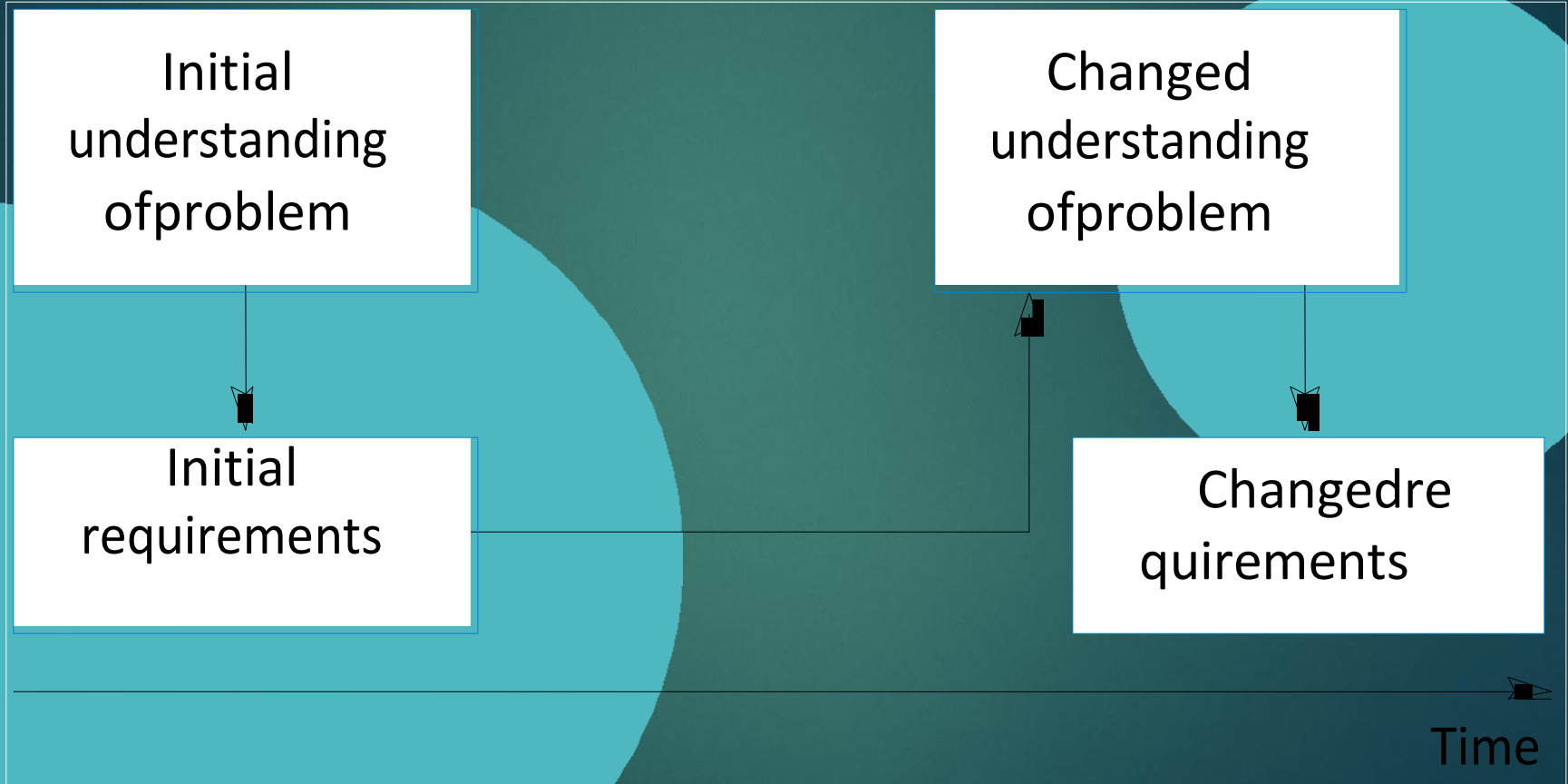
Reasons for requirements changes

- (a) Diverse Users community where users have different requirements and priorities
- (b) System customers and end users are different
- (c) Change in the business and technical environment after installation

Two classes of requirements

- (a) Enduring requirements: Relatively stable requirements
- (b) Volatile requirements: Likely to change during system development process or during operation

Requirementsevolution



Requirements management planning

An essential first stage in requirement management process Planning process consist of the following

1. Requirements identification

--

Each requirement must have unique tag for cross reference and traceability

2. Change management process

--

Set of activities that assess the impact and cost of changes

3. Traceability policy

--

A matrix showing links between requirements and other elements of software development

4. CASE tools support

--

Automatic tool to improve efficiency of change management process. Automated tools are required for requirements storage, change management and tr



your roots to success

Traceability

Maintainsthreetypesoftraceabilityinformation.

1. Source traceability

-- Linkstherequirementsto the stakeholders

2. Requirementstraceability

--

Linksdependentrequirementswithintherequirements document

3. Designtraceability

-- Linksfromtherequirementstothe designmodule

Requirements change management

Consists of three principal stages:

1. Problem analysis and change specification

--Process starts with a specific change

to verify that it

proposal and analysed
is valid

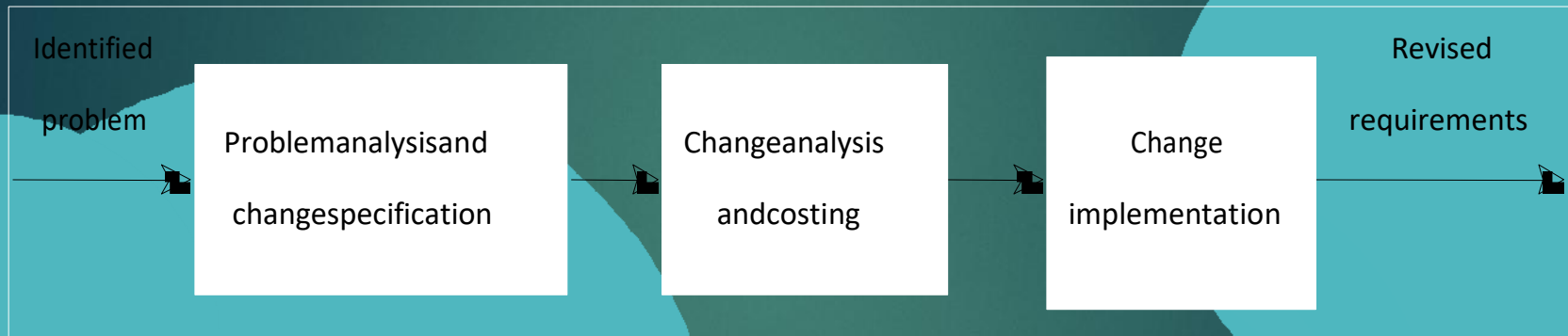
2. Change analysis and costing

--Impact analysis in terms of cost, time and risks

3. Change implementation

--Carrying out the changes in requirements document, system design and its implementation

Change management



SYSTEMMODELS

- ▶ Used in analysis process to develop understanding of the existing system or new system.

- ▶ Excluded details

- ▶ An abstraction of the system

Types of system models

1. Context models

2. Behavioural models

3. Data models
4. Object models

5. Structured models

models

CONTEXT MODELS

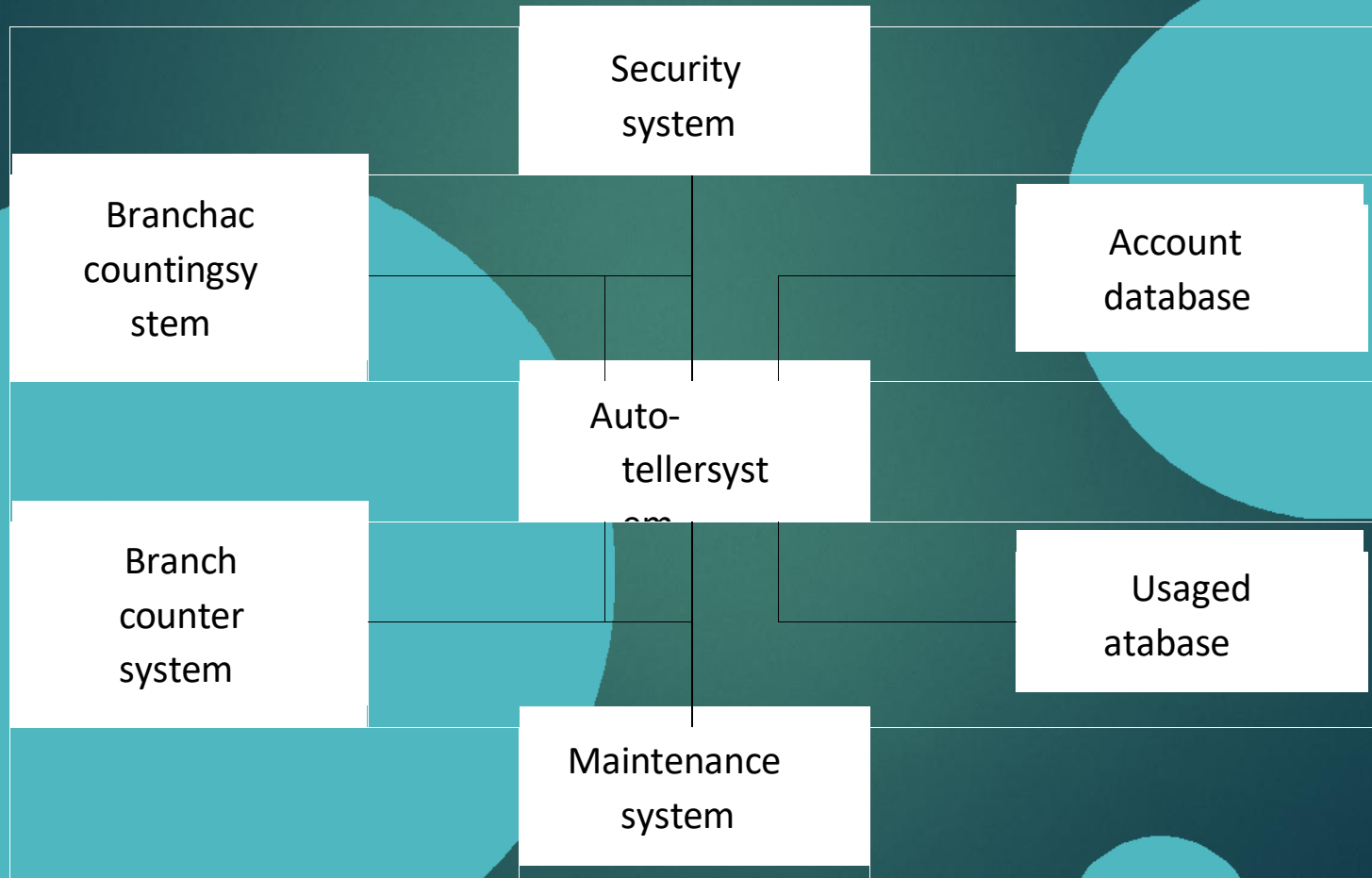
A type of architectural model

- ▶ Consists of sub-systems that make up an entire system
- ▶ First step: To identify the subsystem
- ▶ Represent the high level architectural model as simple block diagram
- ▶ Depict each subsystem as a named rectangle
- ▶ Lines between rectangles indicate associations between subsystems

Disadvantages

- ▶ Concerned with system environment only, doesn't take into account other systems, which may take data or give data to the model

The context of an ATM system



Behavioral models

Describe the overall behaviour of a system.

Two types of behavioural model

1. Data Flow models
2. State machine models

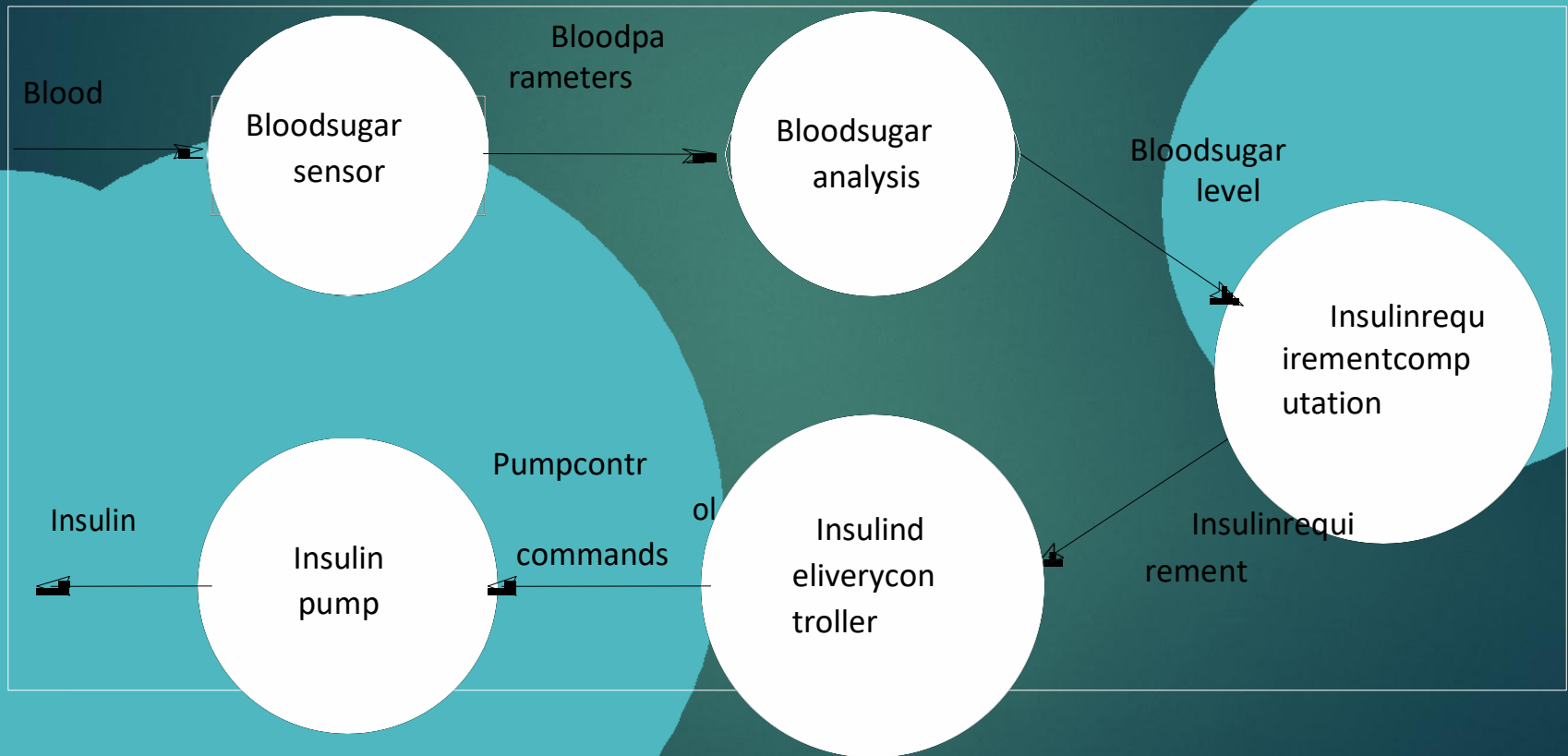
▶ Data flow models

- ▶ Concentrate on the flow of data and functional transformation on that data
- ▶ Show the processing of data and its flow through a sequence of processing steps
- ▶ Help analyst understand what is going on

Advantages

- ▶ Simple and easily understandable
- ▶ Useful during analysis of requirements

Insulin pump DFD



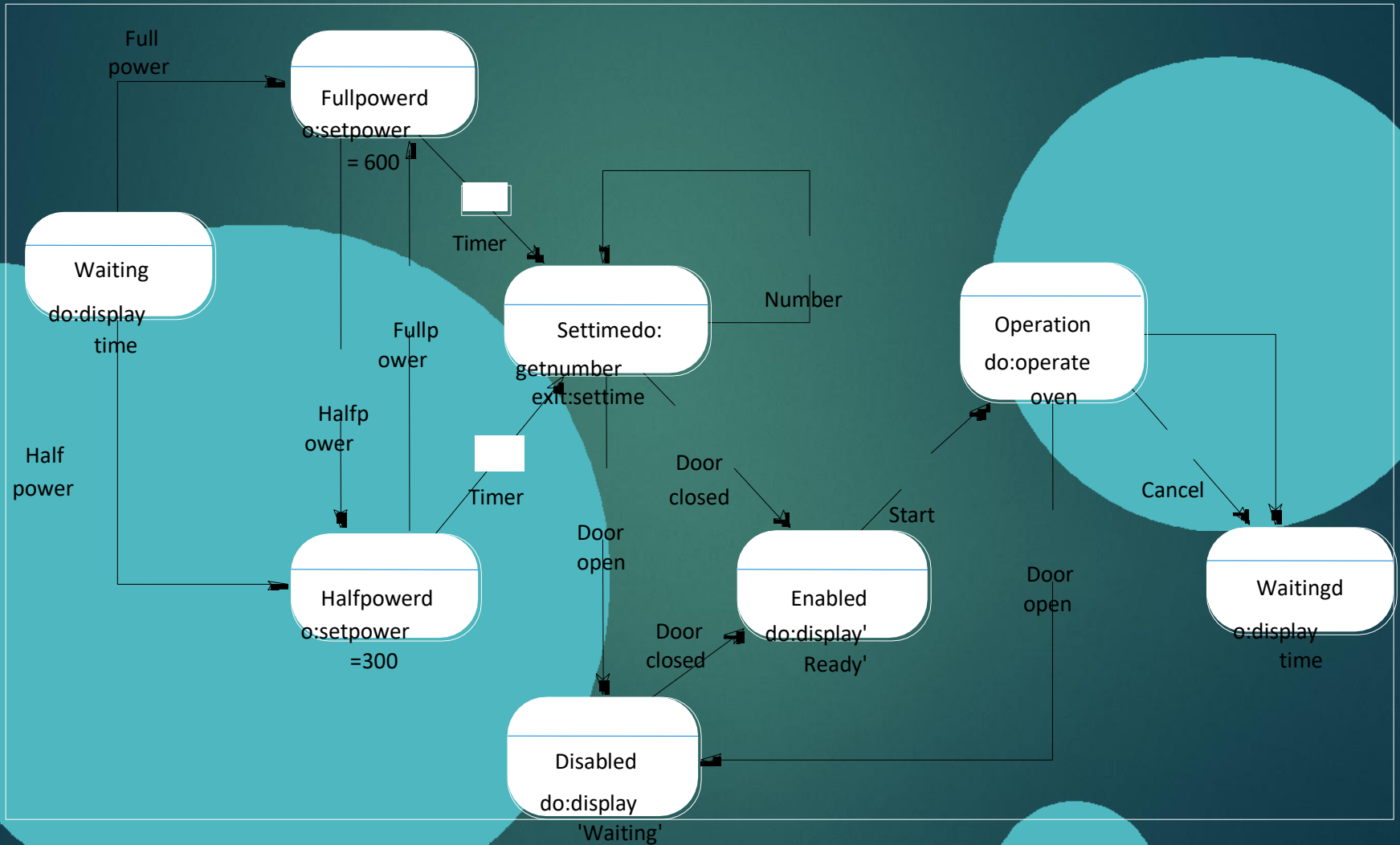
State machine models

- ▶ Describe how a system responds to internal or external events
- ▶ Show system states and events that cause transition from one state to another
- ▶ Does not show the flow of data within the system
- ▶ Used for modeling of real time systems
- ▶ Exp: Microwave oven
- ▶ Assume that at any time, the system is in one of a number of possible states
- ▶ Stimulus triggers a transition from one state to another state

Disadvantage

- ▶ Number of possible states increases rapidly for large system models

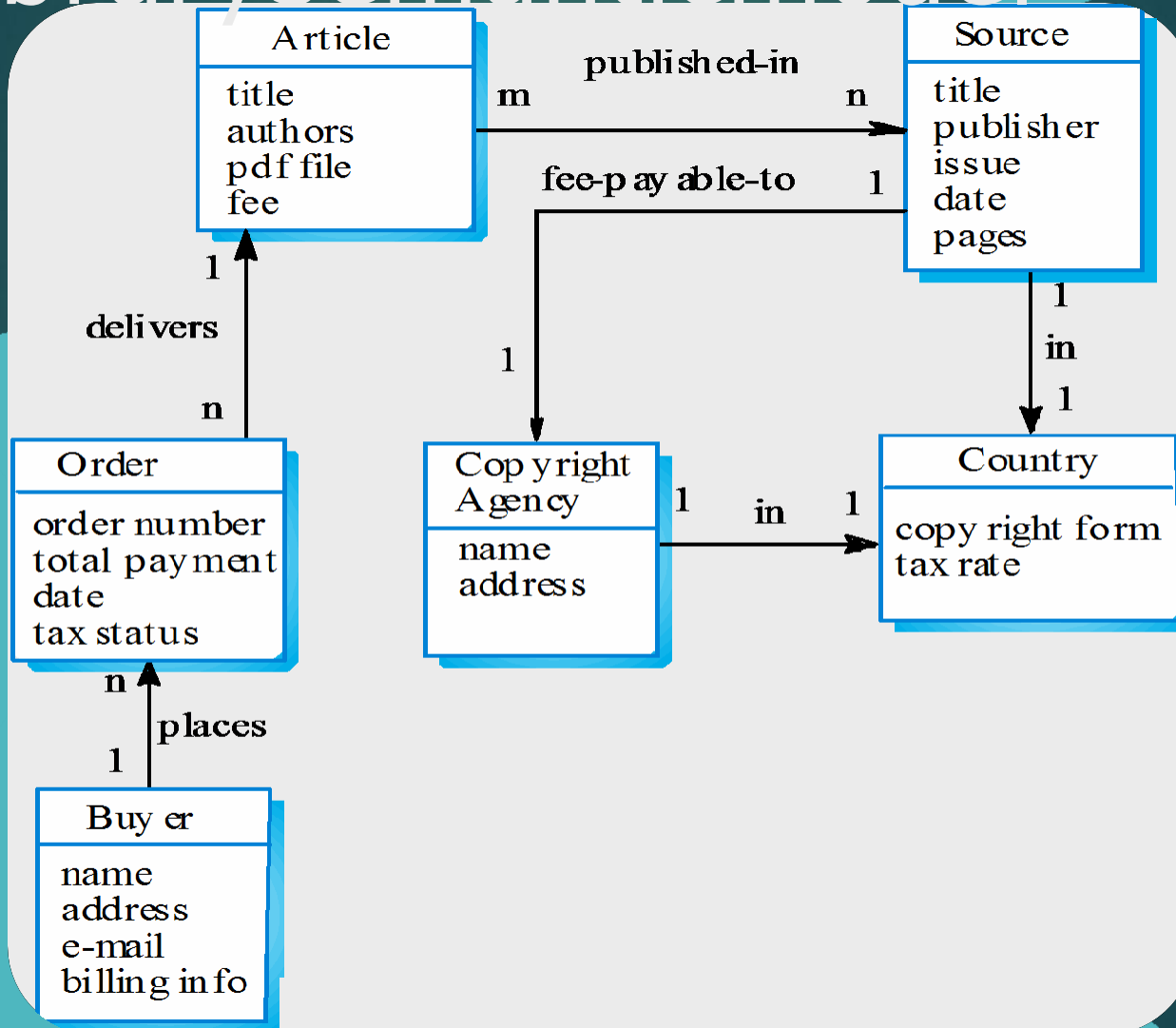
Microwave ovenmodel



DATAMODELS

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

Librarysemanticmodel



Data dictionary entries

Name	Description	Type	Date
Article	Details of the published article that may be ordered by people using LIBSYS.	Entity	30.12.2002
authors	The names of the authors of the article who may be due a share of the fee.	Attribute	30.12.2002
Buyer	The person or organisation that orders a copy of the article.	Entity	30.12.2002
fee-payable-to	A 1:1 relationship between Article and the Copyright Agency who should be paid the copyright fee.	Relation	29.12.2002
Address (Buyer)	The address of the buyer. This is used to any paper billing information that is required.	Attribute	31.12.2002

OBJECT MODELS

- ▶ An object-oriented approach is commonly used for interactive systems development
- ▶ Expresses the systems requirements using objects and developing the system in an object-oriented PL such as C++
- ▶ **Object class:** An abstraction over a set of objects that identifies common attributes
 - ▶ Objects are instances of object class
 - ▶ Many objects may be created from a single class
 - ▶ Analysis process
 - ▶ Identifies objects and object classes
 - ▶ Object class in UML
 - ▶ Represented as a vertically oriented rectangle with three sections
 - ▶ (a) The name of the object class in the top section
 - ▶ (b) The class attributes in the middle section
 - ▶ (c) The operations associated with the object class are in the lower section.

OBJECT MODELS

INHERITENCE MODELS

- ▶ A type of object-oriented model which involves in object classes attributes
- ▶ Arranges classes into an inheritance hierarchy with the most general object class at the top of hierarchy
- ▶ Specialized objects inherit their attributes and services UM

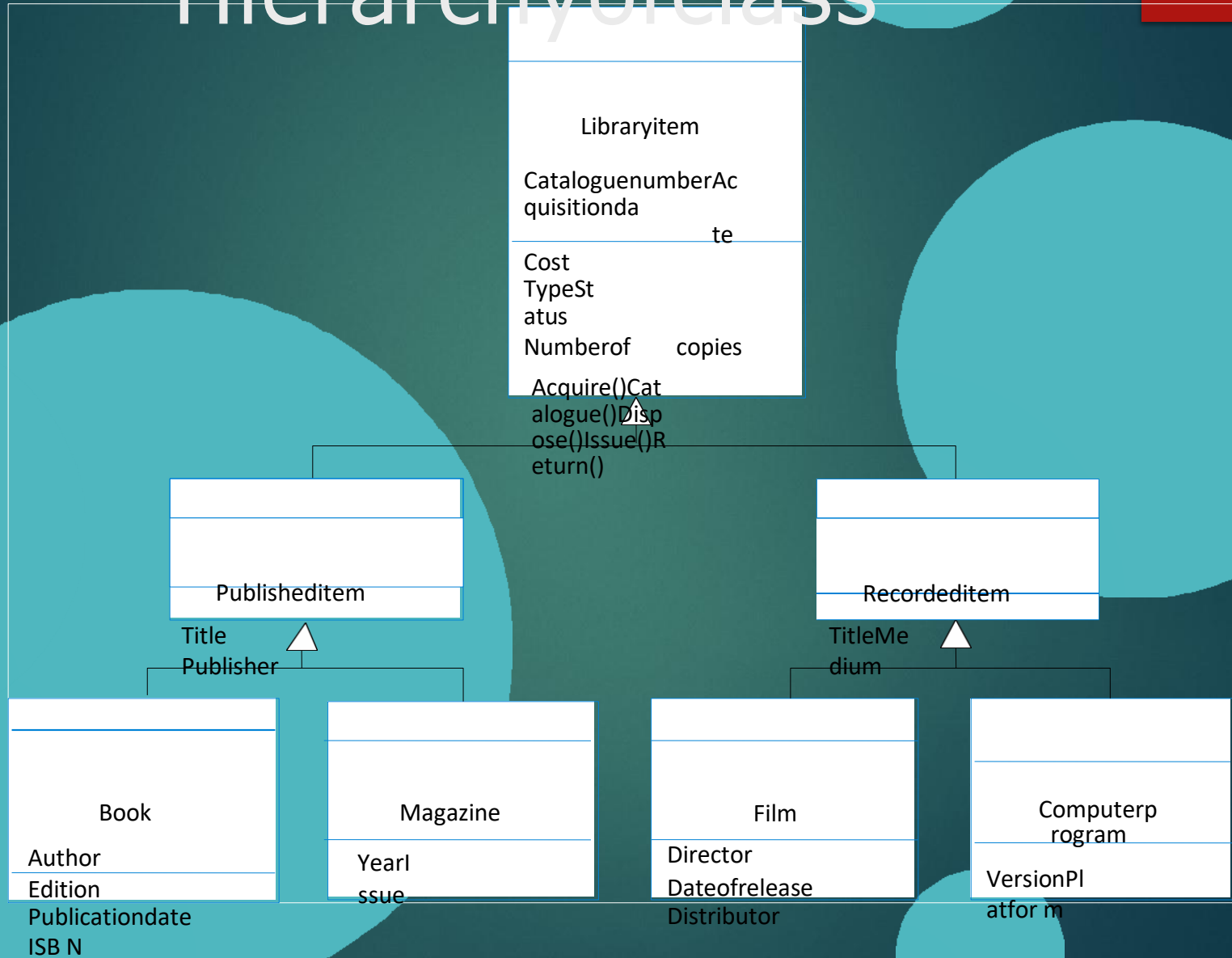
Notation

- ▶ Inheritance is shown upward rather than downward

Single Inheritance: Every object class inherits its attributes and operation from a single parent class

Multiple Inheritance: A class of several of several parents.

Hierarchy of class



OBJECT MODELS

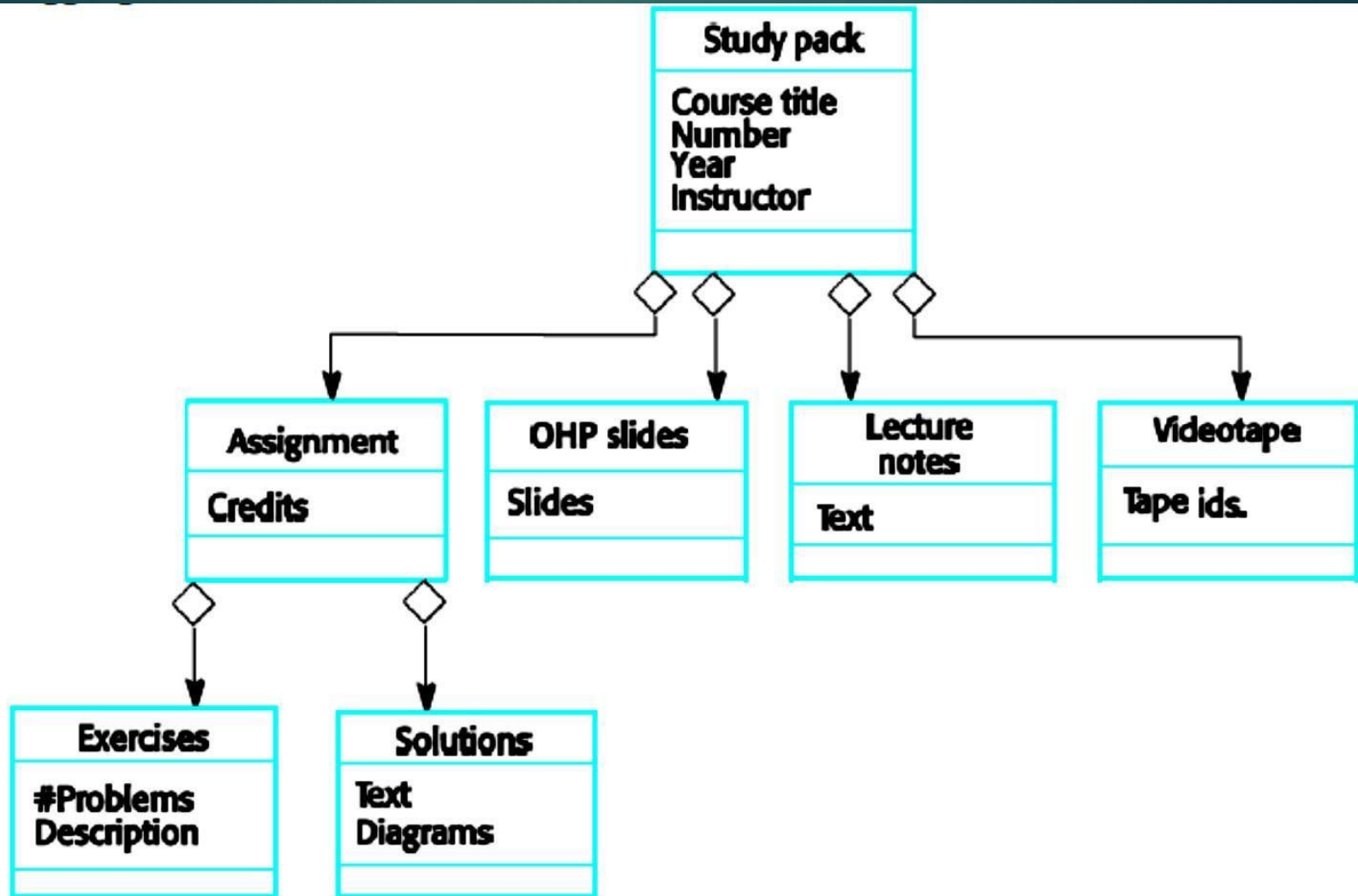
OBJECT AGGREGATION

- ▶ Some objects are grouping of other objects
- ▶ An aggregate of a set of other objects
- ▶ The classes representing these objects may be modeled using an object aggregation model
- ▶ A diamond shape on the source of the link represents the composition

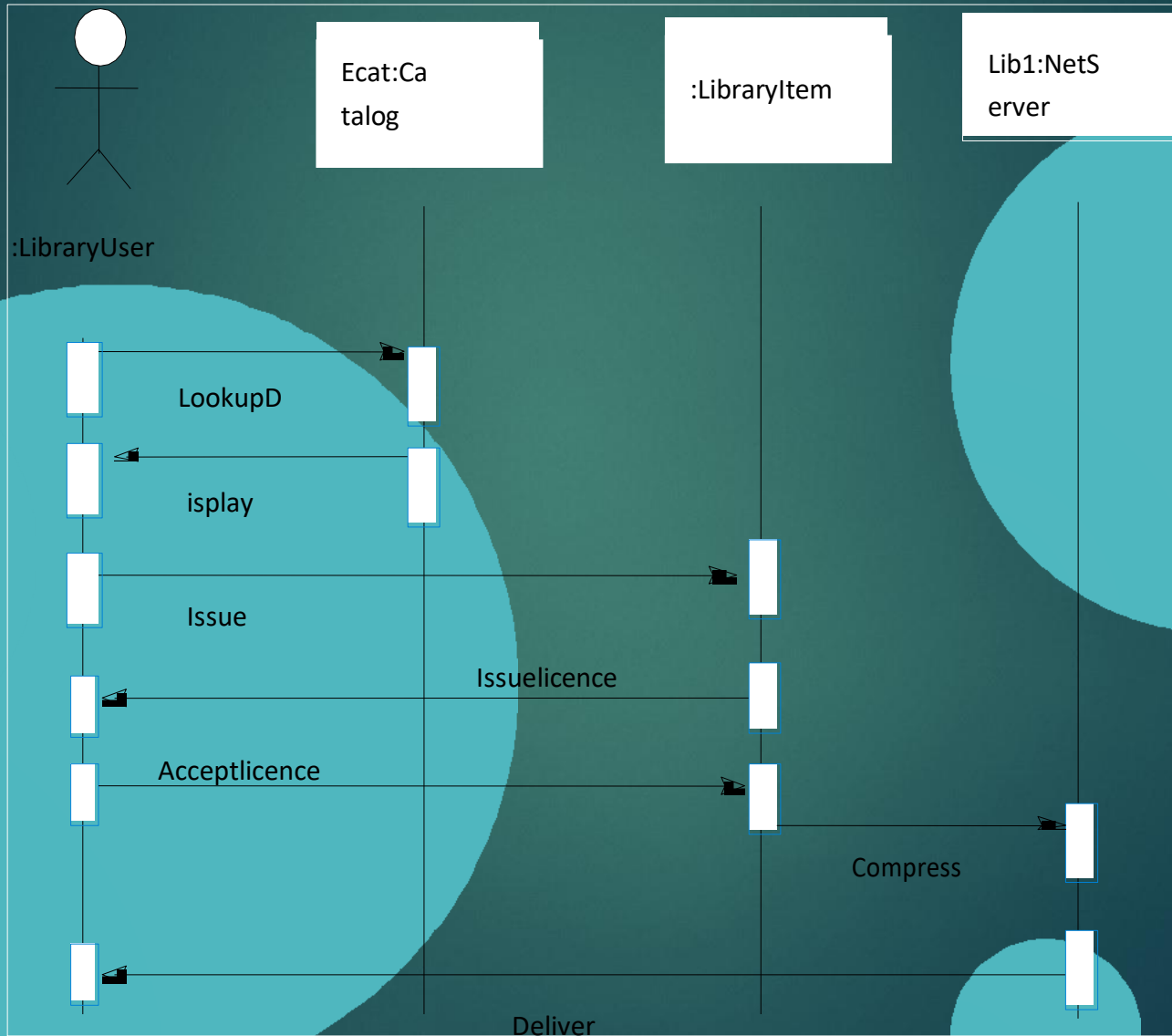
OBJECT-BEHAVIORAL MODEL

- ▶ Shows the operations provided by the objects
- ▶ Sequenced diagram of UML can be used for behavioral modeling

Object aggregation



OBJECT BEHAVIORAL MODEL



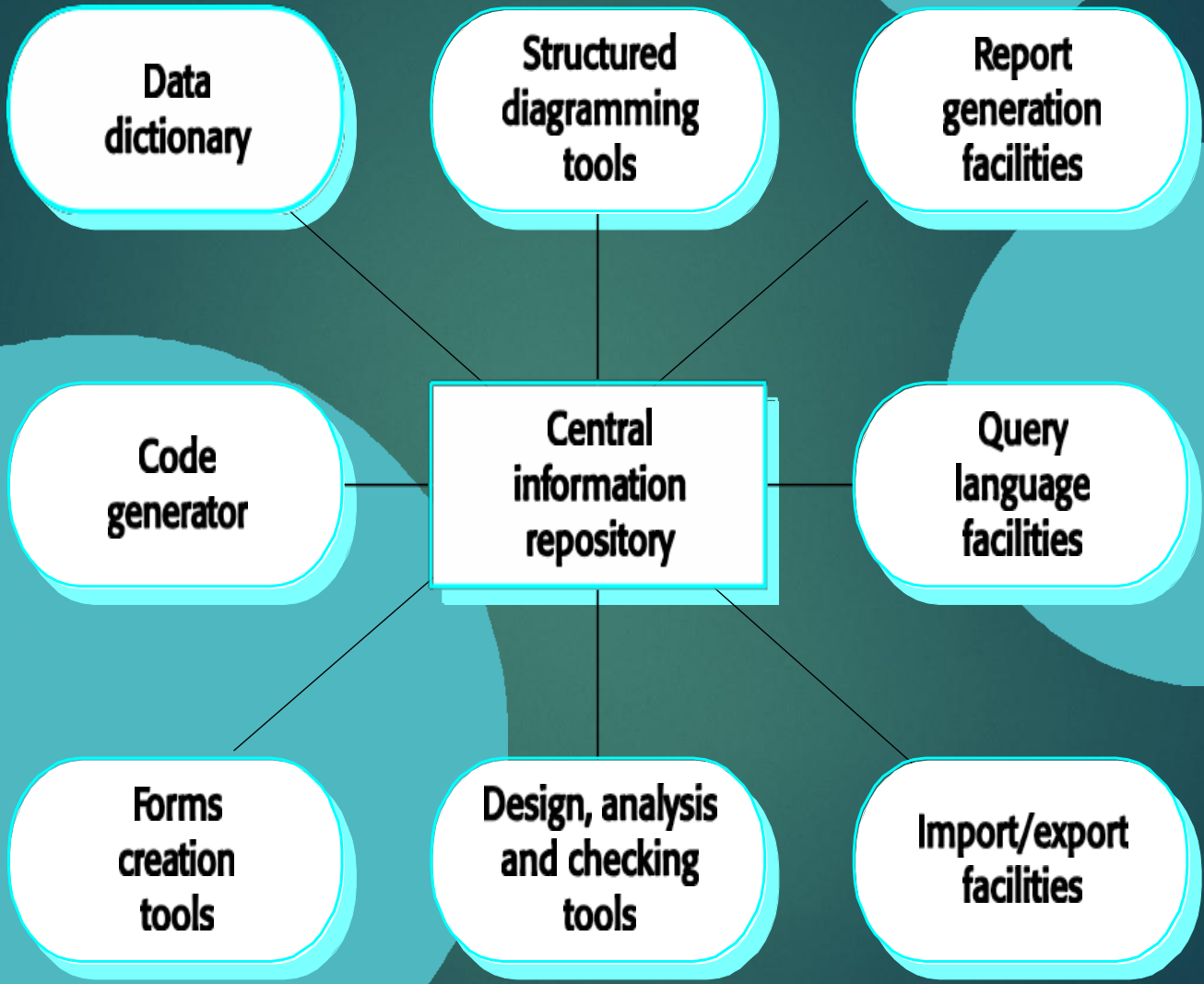
STRUCTURED METHODS:

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

Method weaknesses:

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

Analysisworkbenchcomponents:



Data dictionary

Structured diagramming tools

Report generation facilities

Code generator

Central information repository

Query language facilities

Forms creation tools

Design, analysis and checking tools

Import/export facilities

NARASIMHAREDDYENGINEERINGCOLLEGE

FACULTY:Dr.VenkateswaruluNaik

CS3102PC:SOFTWAREENGINEERING

III YEARB.TECH. CSE I-SEM (R20)

UNIT-III





▶ **Design**

Engineering: Design process and Design quality, Design concepts, the design model.

▶ **Creating an**

architectural design: Software architecture, Data design, Architectural styles and patterns, Architectural Design, conceptual model of UML, basic structural modeling, class diagrams, sequenced diagrams, collaboration diagrams, use case diagrams, component diagrams.

Design Engineering

- ▶ **Design engineering encompasses these set of principles, concepts, and practices that lead to the development of a high-quality system or product.**

What is design?

- ▶ Design is what virtually every engineer wants to do.
- ▶ It is the place where creativity rules—customer's requirements, business needs, and technical considerations all come together in the formulation of a product or a system.

Why is it important?

- ▶ Design allows a software engineer to model the system or product that is to be built.
- ▶ Design is the place where software **quality is established.**

DESIGN PROCESS AND DESIGN QUALITY:

Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software.

Goals of design:

- ▶ The design must implement all of the explicit requirements contained in the analysis model.
- ▶ The design must be readable, understandable guide.
- ▶ The design should provide a complete picture of the software

Quality guidelines:

- ▶ A design should exhibit an architecture that,
 - ▶ **a.** has been created using recognizable architectural styles or patterns
 - ▶ **b.** is composed of components that exhibit good design characteristics and
 - ▶ **c.** can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
- ▶ A design should be modular.
- ▶ A design should contain distinct representation of data, architecture, interfaces and components.

- ▶ lead to data structures that are appropriate for the classes to be implemented.
- ▶ lead to components that exhibit independent functional characteristics.
- ▶ lead to interfaces that reduce the complexity of connections between components and with the external environment.
- ▶ should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- ▶ should be represented using a notation that effectively communicates its meaning.

Quality attributes: (FURPS)

- ▶ *Functionality*
- ▶ *Usability*
- ▶ *Reliability*
- ▶ *Performance*



your roots to success

DESIGN CONCEPTS:

Fundamental software design concepts provide the necessary framework for “getting it right.”

1. Abstraction: Many levels of abstraction are there,

- ▶ At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- ▶ At lower levels of abstraction, a more detailed description of the solution is provided.
- ▶ *Procedural abstraction* refers to a sequence of instructions that have a specific and limited function.
- ▶ *Data abstraction* is a named collection of data that describes a data object.

2. Architecture:

- ▶ the overall structure of the software and the ways in which that structure provides conceptual integrity for a system
 - ▶ *Structured models* - organized collection of program components.
 - ▶ *Framework models* - increase the level of design abstraction.
 - ▶ *Dynamic models* - address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
 - ▶ *Process models* - focus on the design of the business or technical process that the system must accommodate.
 - ▶ *Functional models* - can be used to represent the functional hierarchy of a system.

3. Patterns:



a design pattern describes a design structure that solves a particular design within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.



The intent of each design pattern is to provide a description that enables a designertodetermine

- 1) Whether the pattern is capable to the current work,
- 2) Whether the pattern can be reused,
- 3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4. Modularity

- ▶ software is divided into separately named and addressable components, sometimes called *modules* that are integrated to satisfy problem requirements.

5. Information Hiding:

- ▶ information contained within a module is inaccessible to other modules that have no need for such information.

6. Functional Independence:

- ▶ direct outgrowth of modularity and the concepts of abstraction and information hiding.
- ▶ Independence is assessed using two qualitative criteria: cohesion and coupling.
 - ▶ *Cohesion* is an indication of the relative functional strength of a module.
 - ▶ *Coupling* is an indication of the relative interdependence among modules.

7. Refinement:

- ▶ A program is development by successively refining levels of procedural detail.
- ▶ Refinement is actually a process of elaboration.

8. Refactoring:

- ▶ Refactoring is a reorganization technique that simplifies the design of a component without changing its function or behavior.

9. Design classes:

- ▶ The software team must define a set of design classes that,
 - ▶ 1. Refine the analysis classes by providing design detail that will enable the classes to be implemented, and
 - ▶ 2. Create a new set of design classes that implement a software infrastructure to support the design solution.

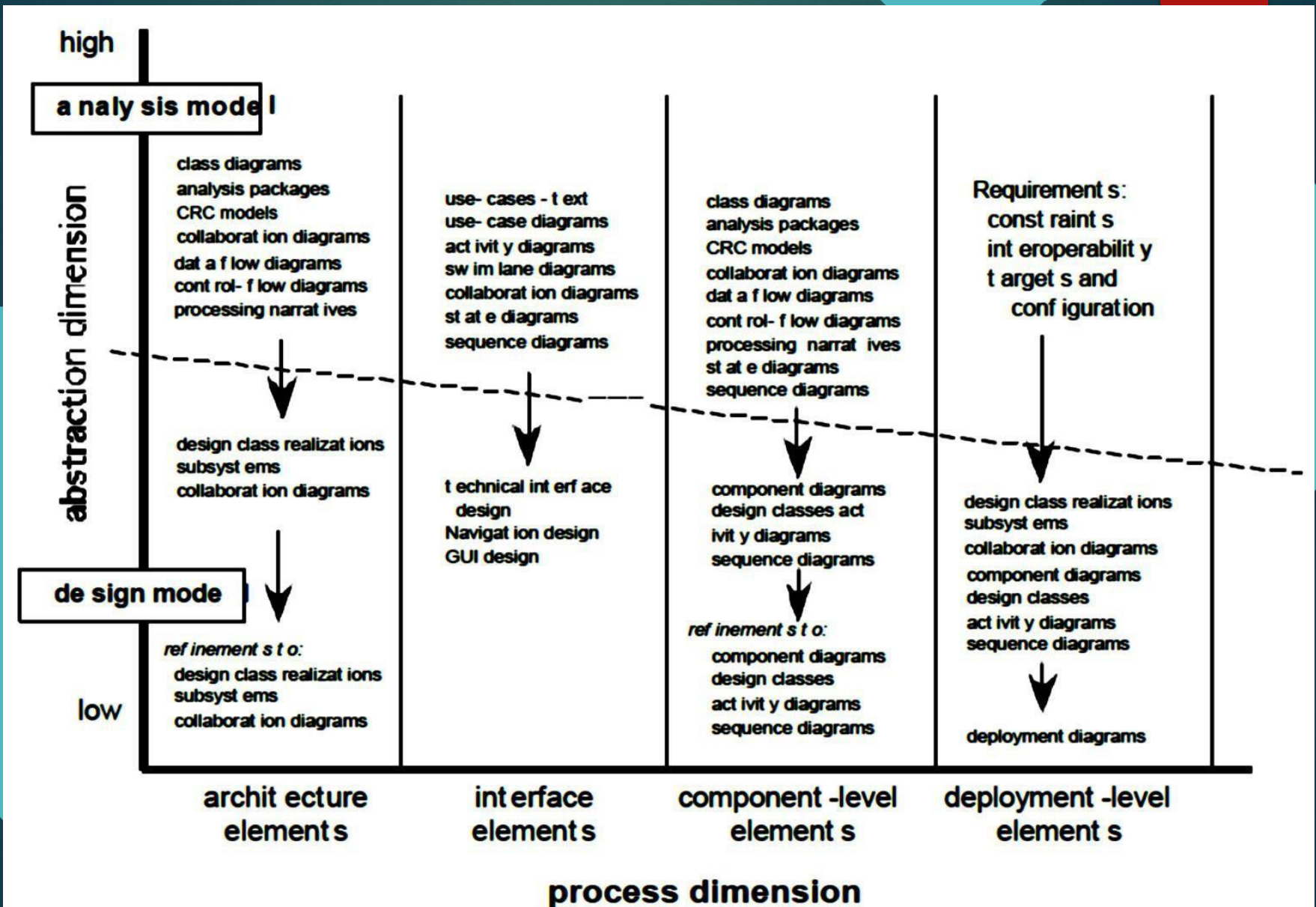
Five different types of design classes,

- ▶ **User interface classes:** define all abstractions that are necessary for human computer interaction.
- ▶ **Business domain classes:** are often refinements of the analysis classes defined earlier.
- ▶ **Process classes** implement lower – level business abstractions required to fully manage the business domain classes.
- ▶ **Persistent classes** represent data stores that will persist beyond the execution of the software.
- ▶ **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Four characteristics of a well-formed design class,

- ▶ Complete and sufficient
- ▶ Primitiveness
- ▶ High cohesion
- ▶ Low coupling

THE DESIGN MODEL:



i. Data design elements:

- ▶ creates a model of data and/or information that is represented at a high level of abstraction.
- ▶ The structure of data has always been an important part of software design.
 - ▶ **At the program component level**
 - ▶ **At the application level**
 - ▶ **At the business level**

ii. Architectural design elements:

- ▶ *The architectural design for software is the equivalent to the floor plan of a house. The architectural model is derived from three sources.*
 - ▶ a. Information about the application domain for the software to be built.
 - ▶ b. Specific analysis model elements, and
 - ▶ c. The availability of architectural patterns.

iii. **Interface design elements:**

- ▶ equivalent to a set of detailed drawings for the doors, windows, and external utilities of a house.
- ▶ There are 3 important elements of interface design:
 - ▶ a. The user interface (UI);
 - ▶ b. External interfaces to other systems, devices, networks, or other producers or consumers of information; and
 - ▶ c. Internal interfaces between various design components.

iv. **Component-level design elements:**

- ▶ It is equivalent to a set of detailed drawings.
- ▶ The component-level design for software fully describes the internal detail of each software component.

v. **Deployment-level design elements:**

- ▶ It indicates how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

ARCHITECTURAL DESIGN SOFTWARE

ARCHITECTURE:

What Is Architecture?

- ▶ Architectural design represents the structure of data and program components that are required to build a computer-based system.
- ▶ The design of software architecture considers two levels of the design pyramid
 - ▶ -data design
 - ▶ -architectural design.

Why Is Architecture Important?

- ▶ Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- ▶ The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- ▶ Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together”.

DATADesign:

- ▶ translates data objects as part of the analysis model into data structures at the software component level and, when necessary, a database architecture at the application level.

1. Data design at the Architectural Level:

- ▶ The challenge for a business has been to extract useful information from this data environment, particularly when the information desired is cross functional.

2. Data design at the Component Level:

- ▶ focuses on the representation of data structures that are directly accessed by one or more software components.

The following set of principles for data specification:

1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structures and the operations to be performed on each should be identified.
3. A data dictionary should be established and used to define both data and program design.
4. Low-level data design decisions should be deferred until late in the design process.
5. The representation of data structures should be known only to those modules that must make direct use of the data contained within the structure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

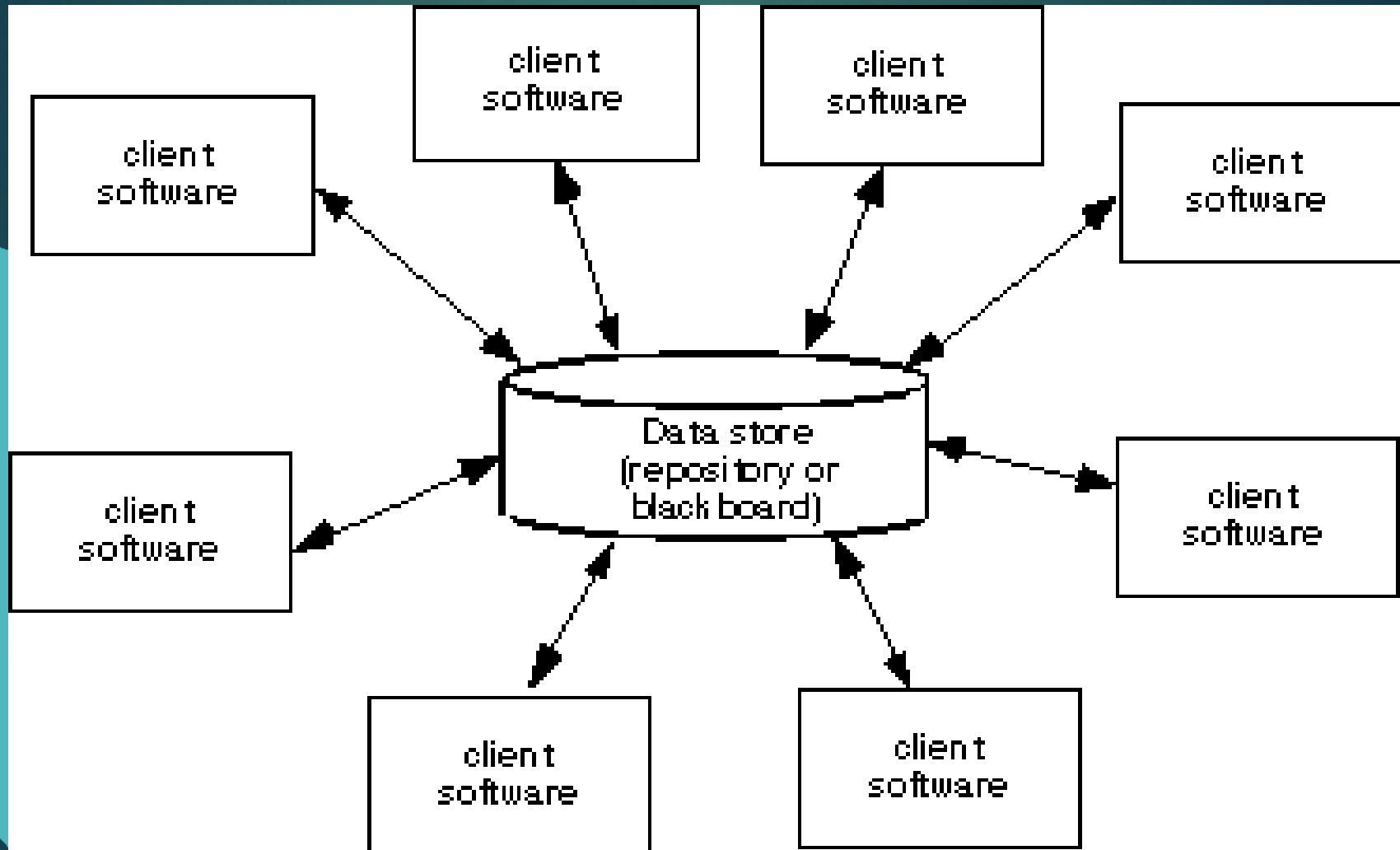
ARCHITECTURAL STYLES AND PATTERNS:

- ▶ The software that is built for computer-based systems also exhibits one of many architectural styles.
- ▶ Each style describes a system category that encompasses
 - (1) A set of *components*
 - (2) A set of *connectors*
 - (3) *Constraints*
 - (4) *Semantic models*

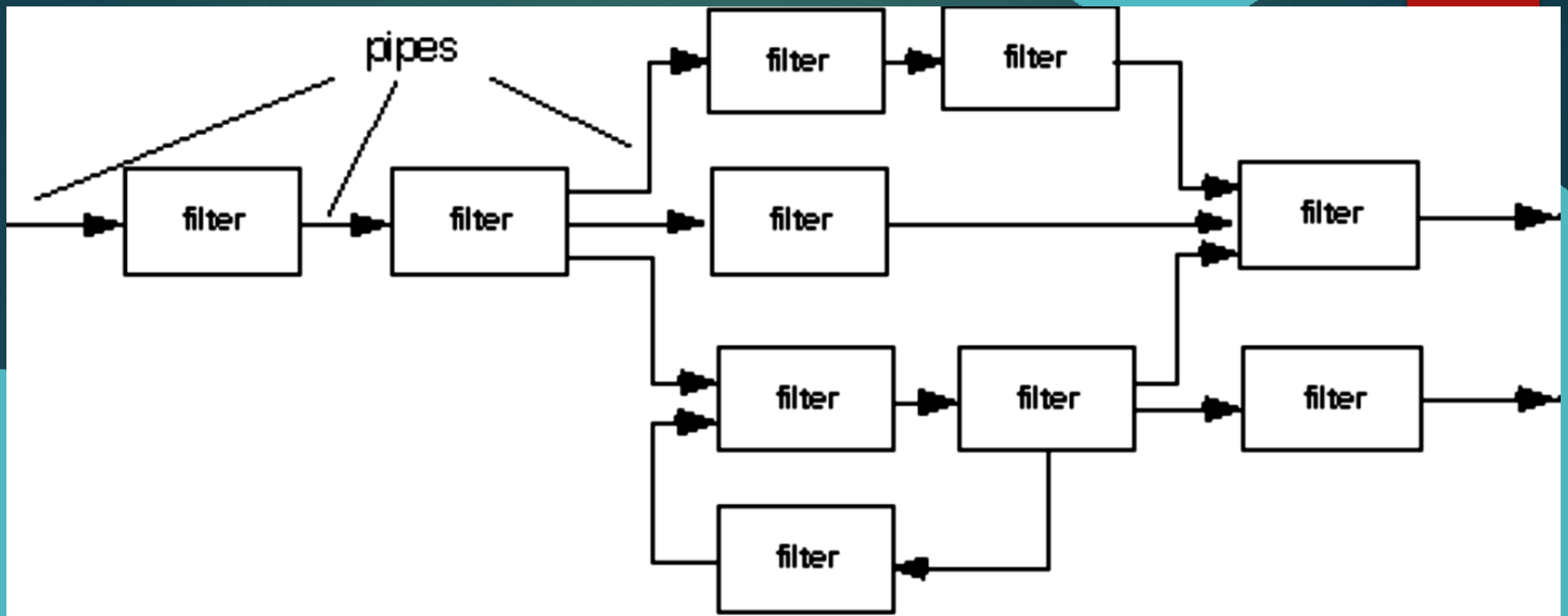
An *architectural pattern*, imposes a transformation on the design of architecture. However, a pattern differs from a style in a number of fundamental ways:

- (1) The scope of a pattern is less broad, focusing on one aspect of the architecturerather than the architecture in its entirety.
- (2) A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level.
- (3) Architectural patterns tend to address specific behavioral issues within the context of the architectural

1. A Brief Taxonomy of Styles and Patterns Data-centered architectures:



Data-flow architectures:



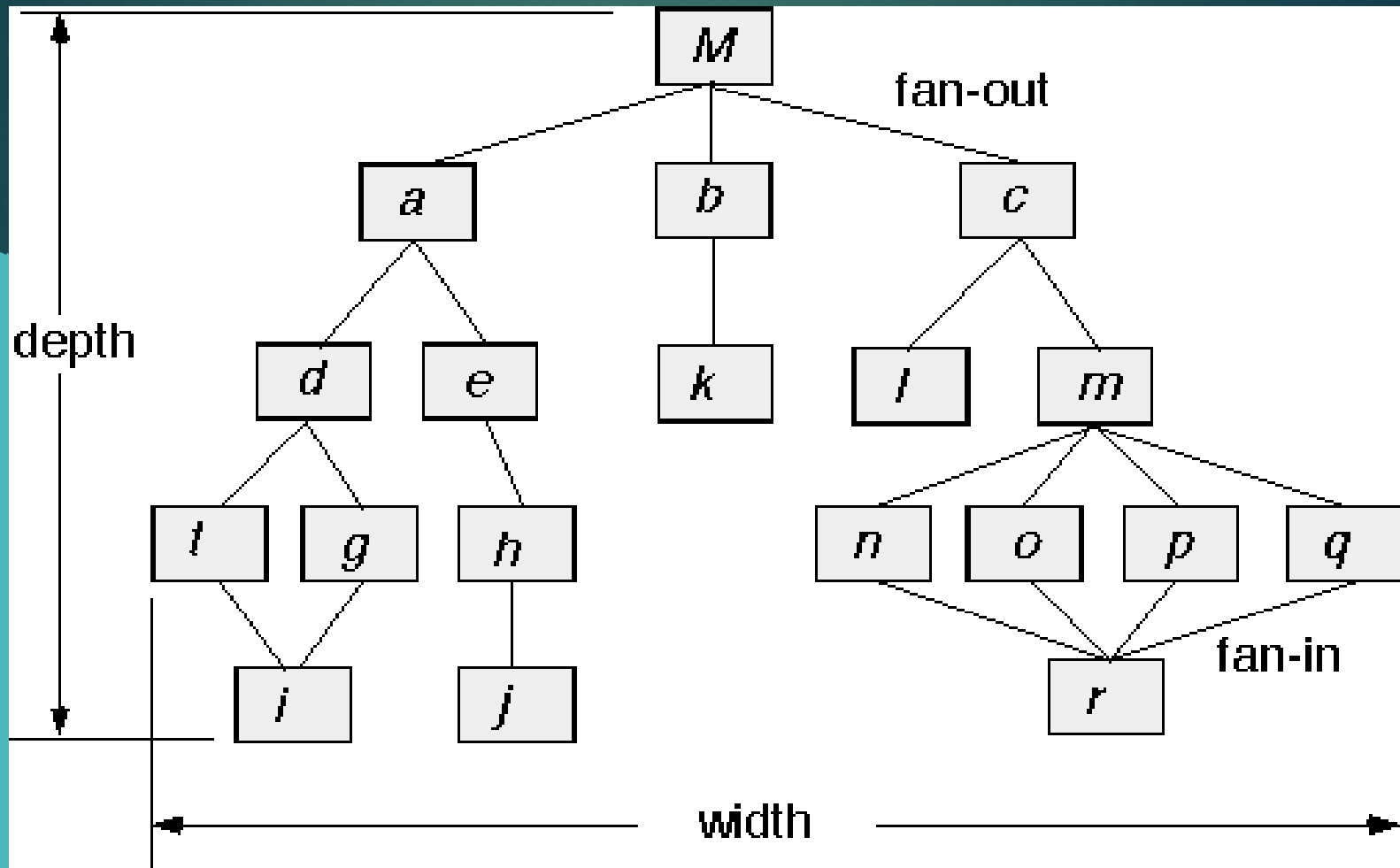
(a) pipes and filters



(b) batch sequential

Callandreturnarchitectures:

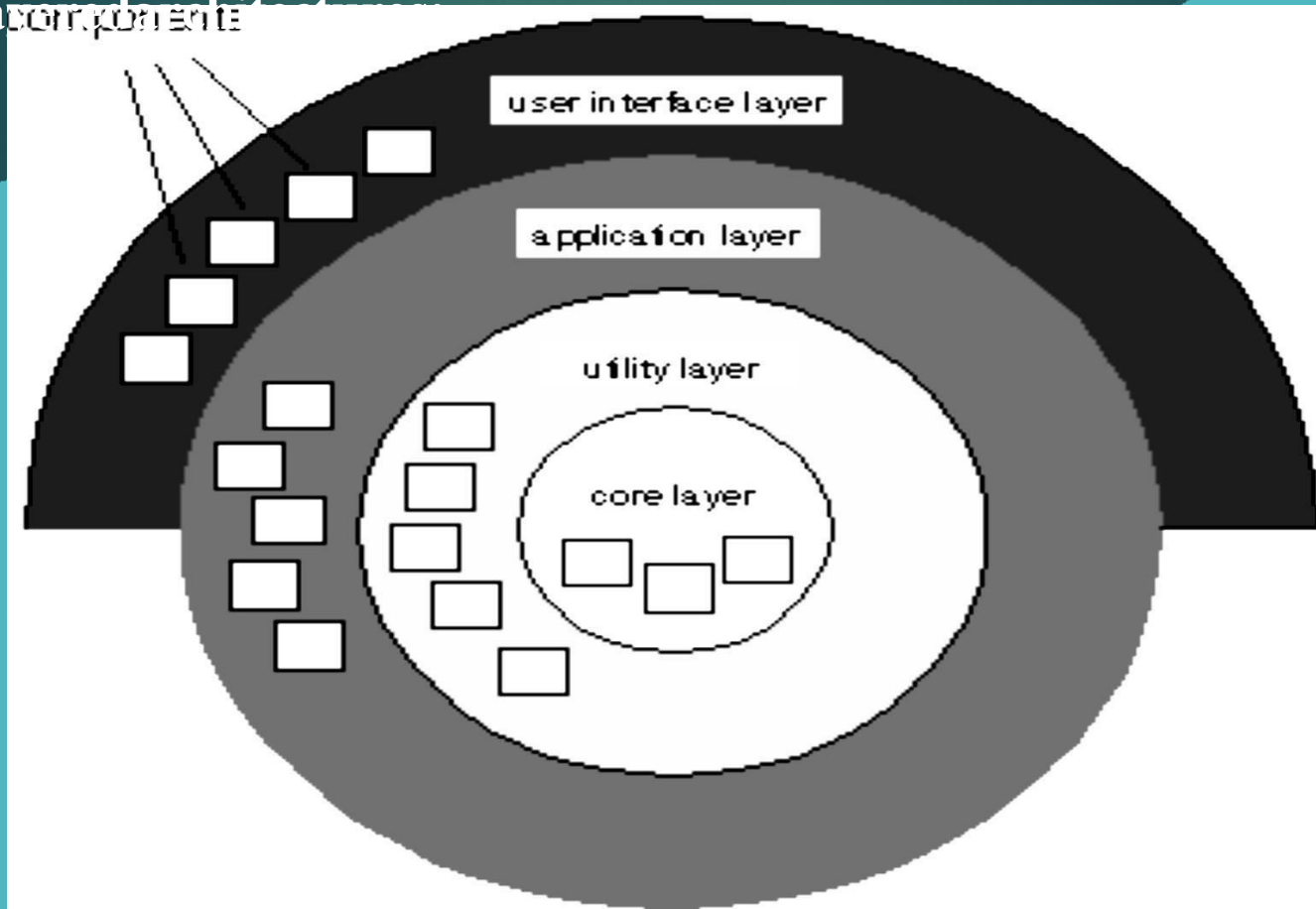
- ▶ *Mainprogram/subprogramarchitectures.*
- ▶ *Remoteprocedurecallarchitectures.*



Object-oriented architectures:

- ▶ The components of a system encapsulate data and the operations that must be applied to manipulate the data.

Layer components:

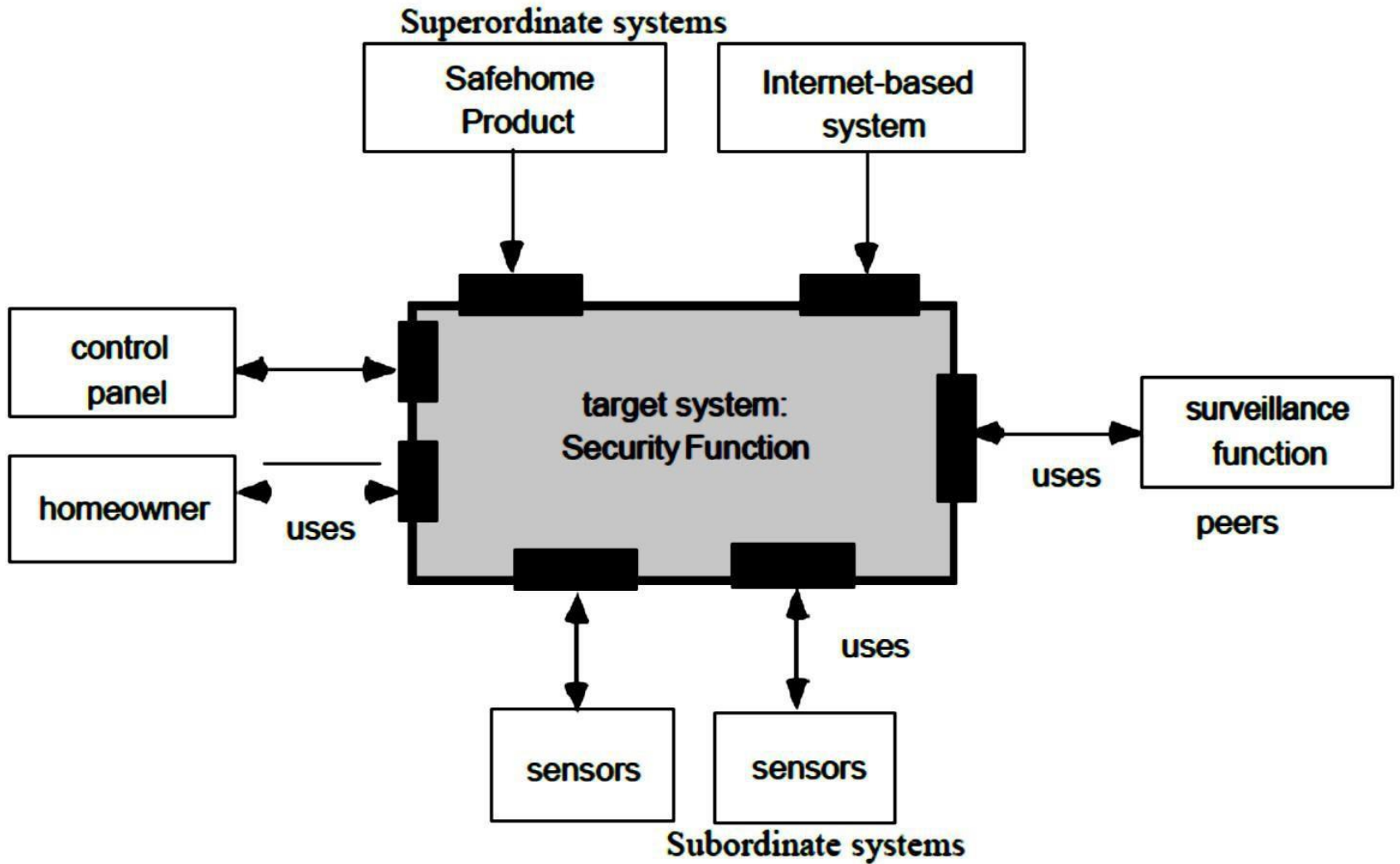


2. Architectural Patterns:

- ▶ The architectural patterns for software define a specific approach for handling some behavioral characteristics of the system,
 - ▶ **Concurrency**
 - ▶ *operating system process management pattern*
 - ▶ *task scheduler pattern*
 - ▶ **Persistence**
 - ▶ *a database management system pattern*
 - ▶ *an application level persistence pattern*
 - ▶ **Distribution**
- ▶ **Organization and Refinement:**
 - ▶ it is important to establish a set of design criteria that can be used to assess an architectural design that is derived.

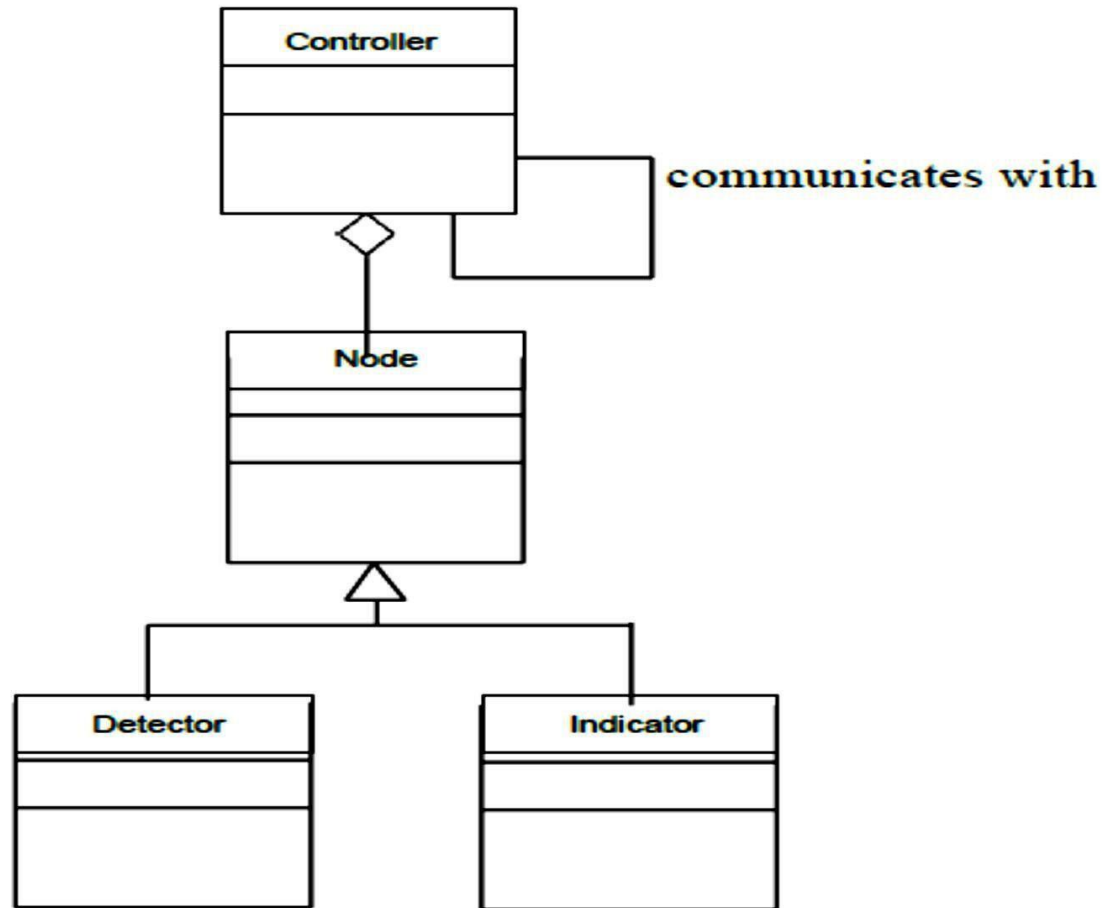
ARCHITECTURAL DESIGN:

i. Representing the System in Context:



ii. Defining Archetypes:

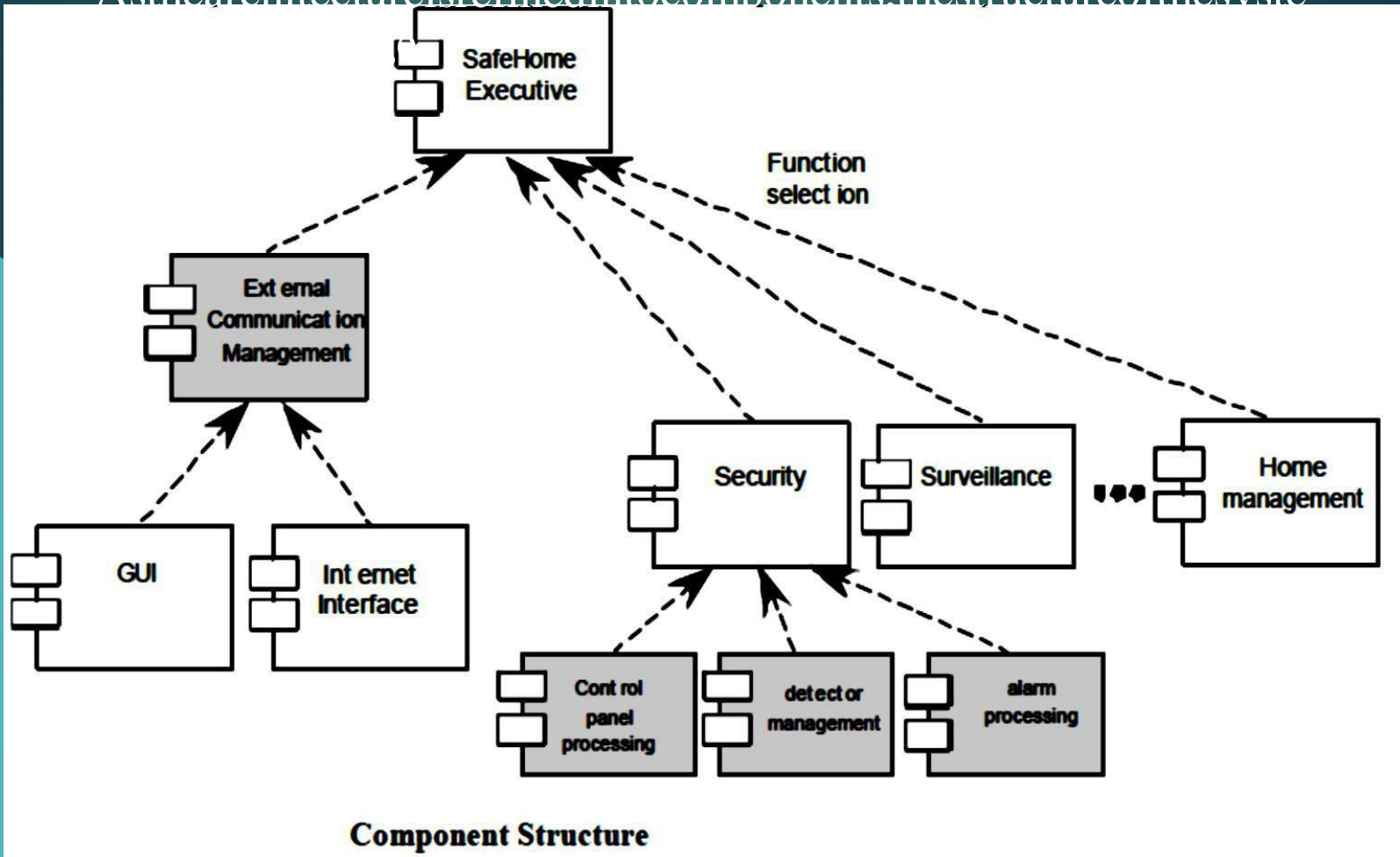
An archetype is a class or pattern that represents a core abstraction that is critical to the design of architecture for the target etc.



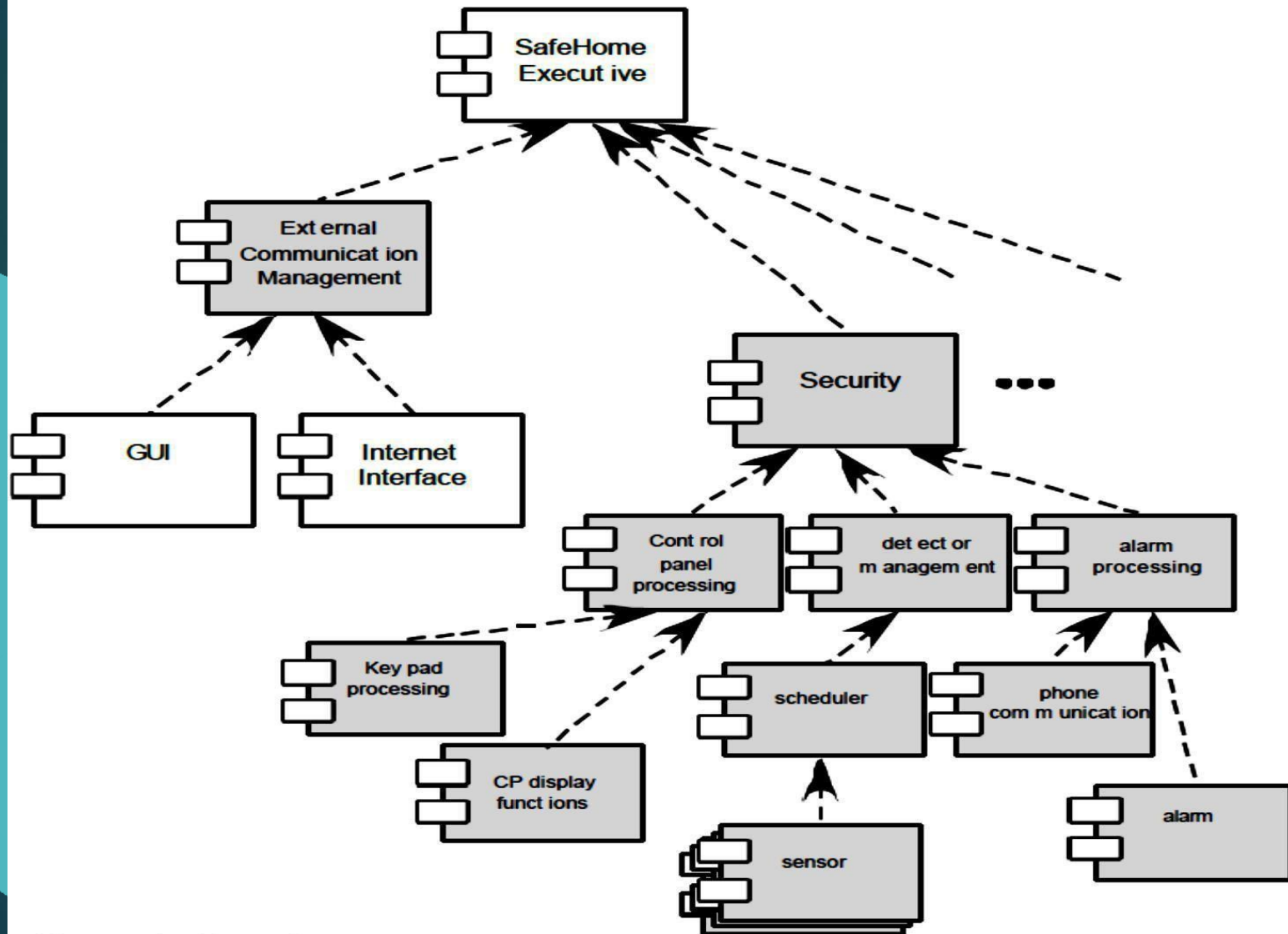
iii. Refining the Architecture into Components:



As the architecture is refined into components, the structure of the system



iv. Describing Instantiations of the System:



A Conceptual Model of UML:

- ▶ UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

Object-Oriented Concepts

- ▶ UML can be described as the successor of object-oriented (OO) analysis and design.
- ▶ Following are some fundamental concepts of the object-oriented world,
- ▶ **Objects**—Objects represent an entity and the basic building block.
- ▶ **Class**—Class is the blueprint of an object.
- ▶ **Abstraction**—Abstraction represents the behavior of a real-world entity.
- ▶ **Encapsulation**—Encapsulation is the mechanism of binding the data together and hiding them from the outside world.
- ▶ **Inheritance**—Inheritance is the mechanism of making new classes from existing ones.
- ▶ **Polymorphism**—It defines the mechanism to exist in different forms.

OO Analysis and Design:

- ▶ OO can be defined as an investigation and to be more specific, it is the investigation of objects. Design means collaboration of identified objects.
- ▶ The purpose of OO analysis and design can be described as—
 - ▶ Identifying the objects of a system.
 - ▶ Identifying their relationships.
 - ▶ Making a design, which can be converted to executable using OO languages.
- ▶ There are three basic steps where the OO concepts are applied and implemented. These steps can be defined as,

OO Analysis → OO Design → OO implementation using OO languages

Basic Structural Modeling:

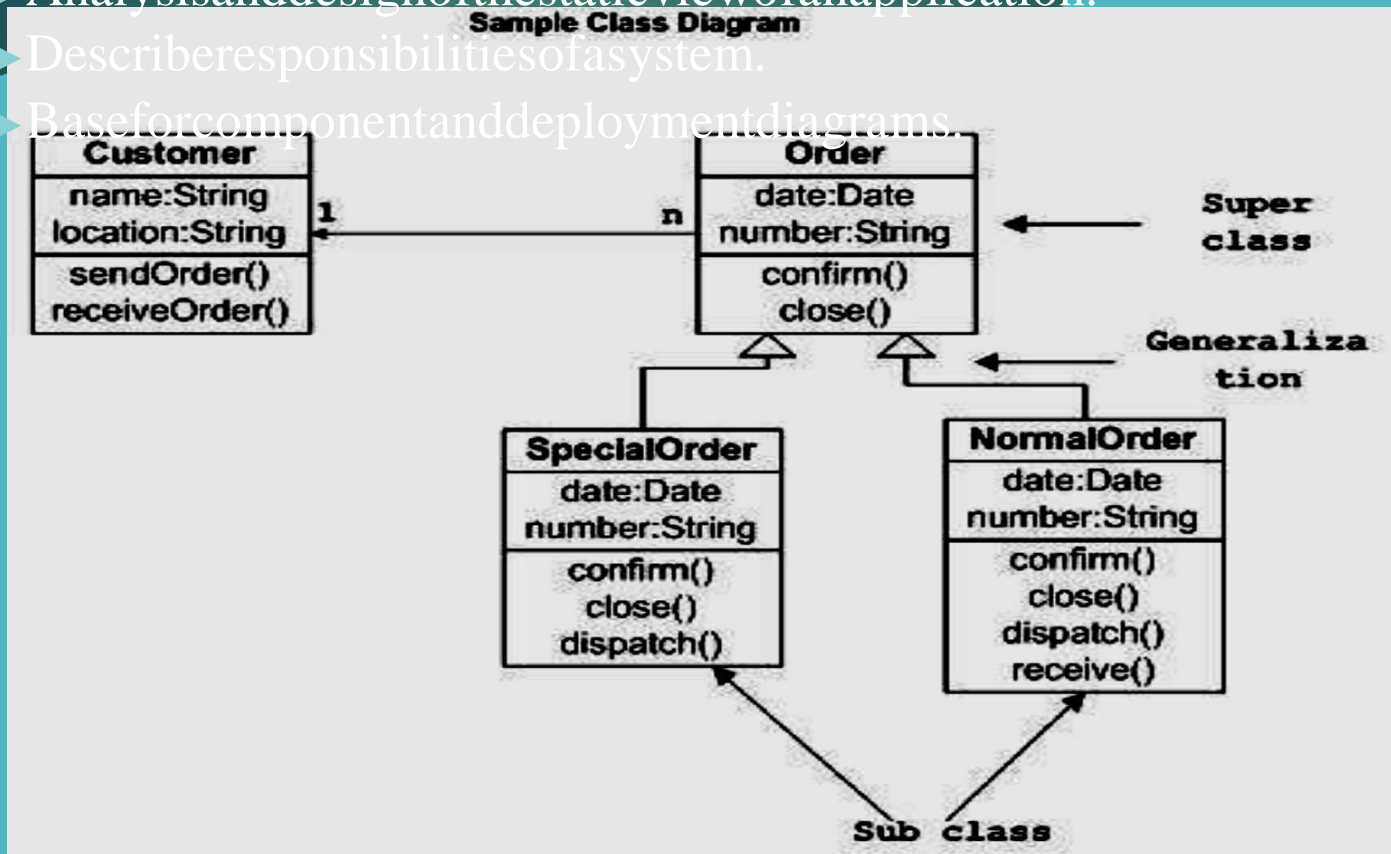
- ▶ Structural modeling captures the static features of a system. They consist of the following—
 - ▶ Classes diagrams
 - ▶ Objects diagrams
 - ▶ Deployment diagrams
 - ▶ Package diagrams
 - ▶ Composite structure diagram
 - ▶ Component diagram
- ▶ Structural model represents the framework for the system and this framework is the place where all other components exist.

ClassDiagram:

- ▶ Classdiagramisastaticdiagramthatshowsacollectionofclasses,interfaces, associations,collaborations,andconstraints.

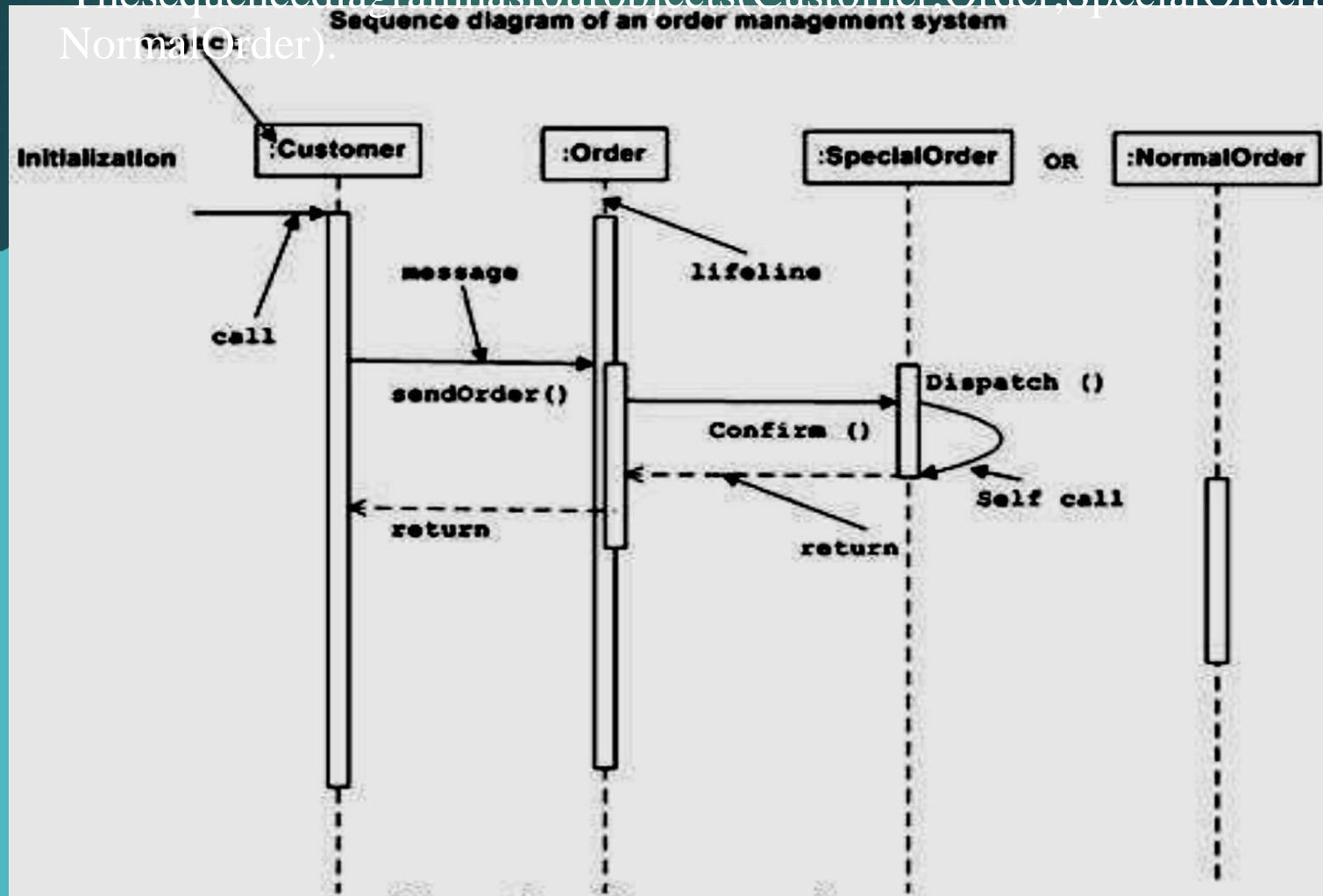
PurposeofClassDiagrams

- ▶ Analysisanddesignofthestaticviewofanapplication.
- ▶ Describeresponsibilitiesofasystem.
- ▶ Baseforcomponentanddeploymentdiagrams.



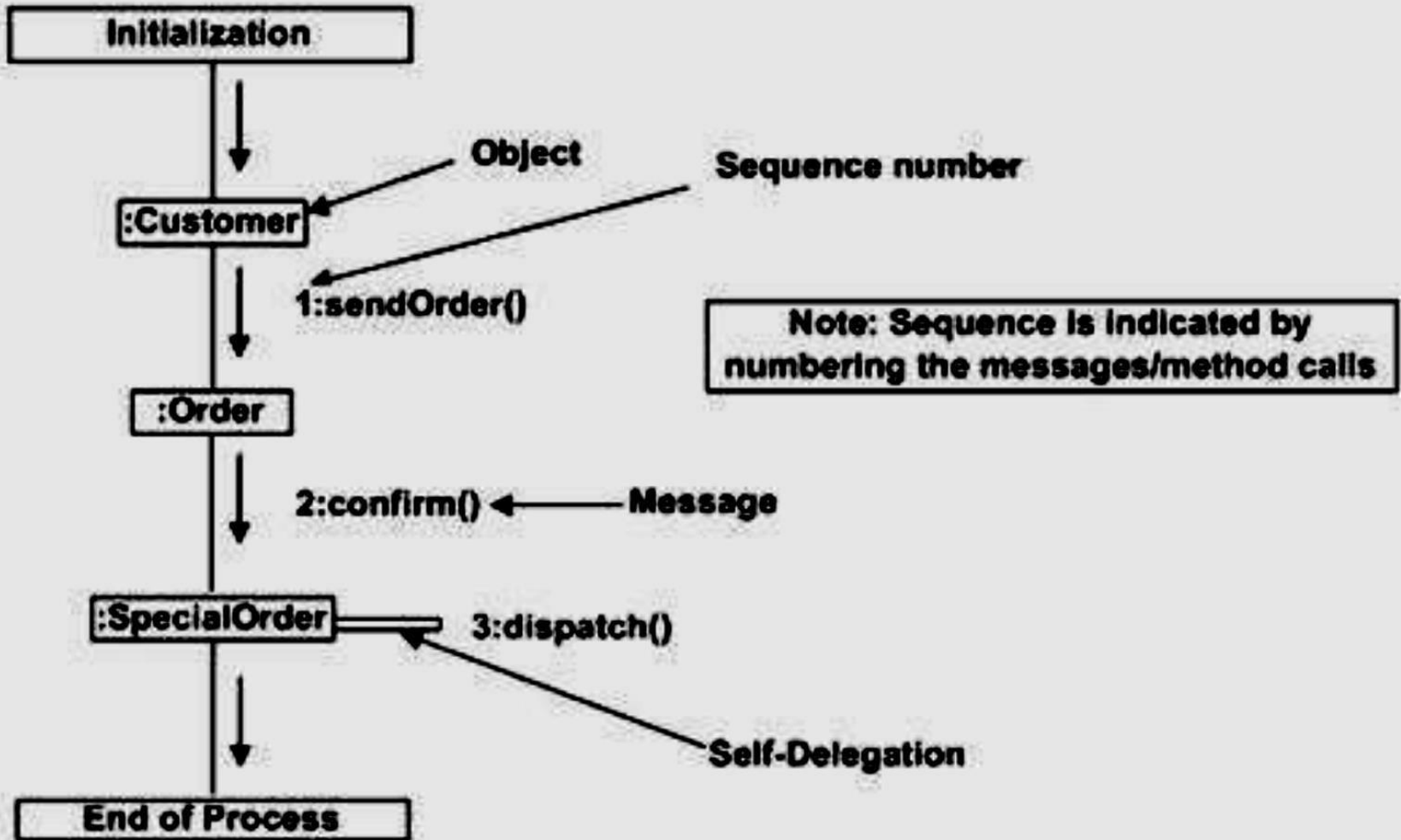
The Sequence Diagram:

These sequencediagramhasfourobjects(Customer,Order,SpecialOrderand NormalOrder).



The Collaboration Diagram:

Collaboration diagram of an order management system



Use Case Diagram:

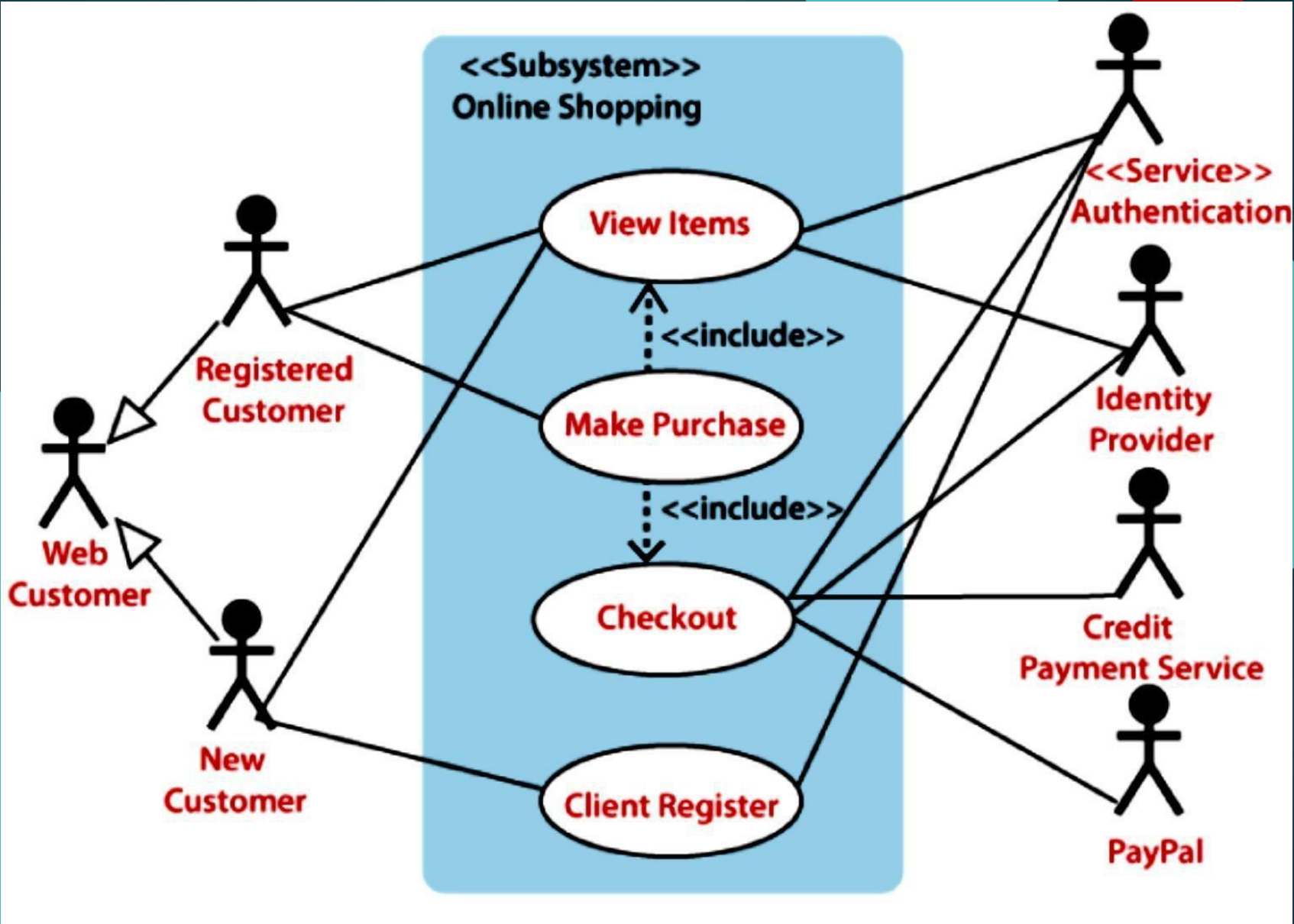
- ▶ It is used to represent the dynamic behavior of a system.
- ▶ It encapsulates the system's functionality by incorporating use cases, actors, and their relationships.

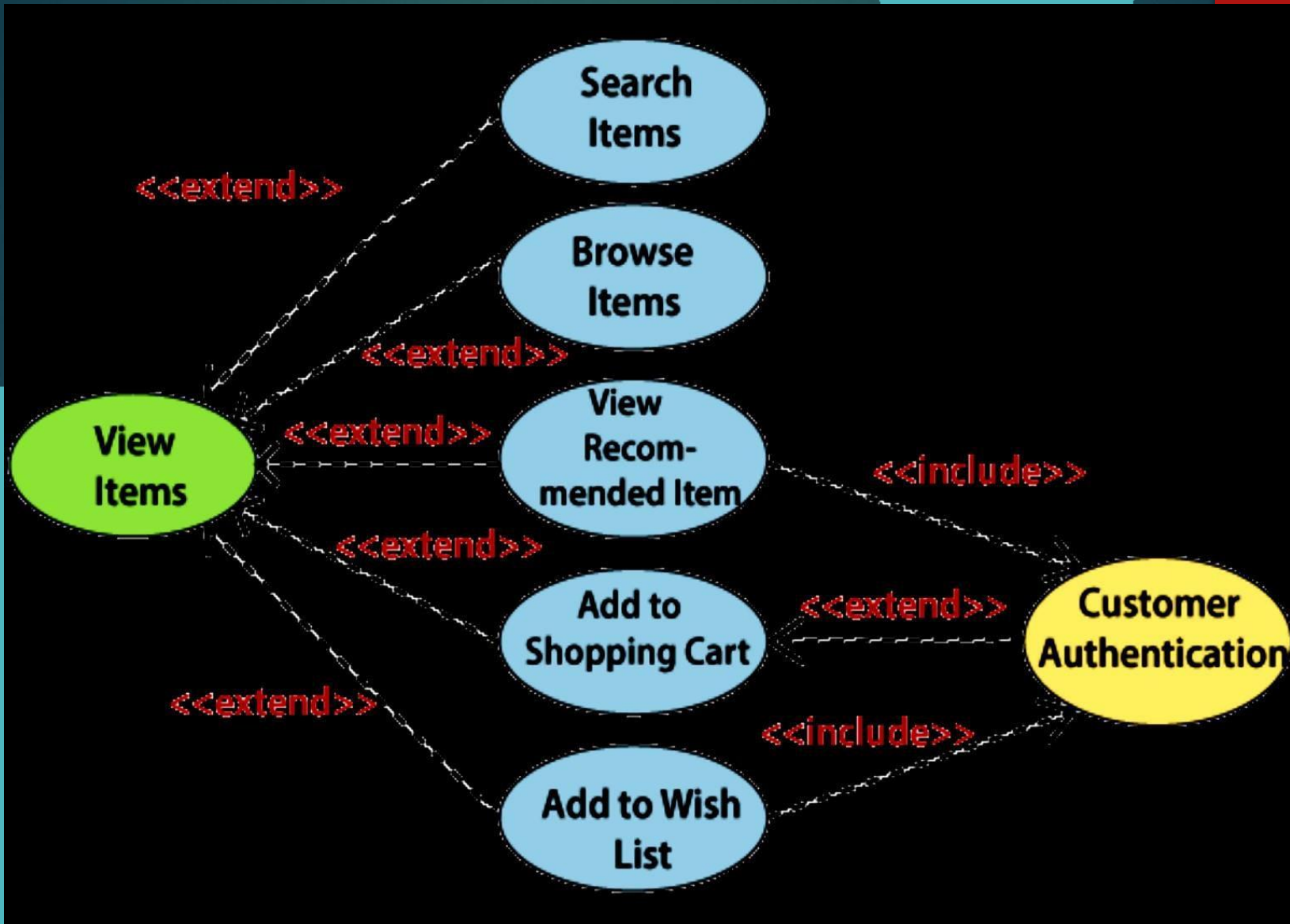
Purpose of Use Case Diagrams

1. It gathers the system's needs.
2. It depicts the external view of the system.
3. It recognizes the internal as well as external factors that influence the system.
4. It represents the interaction between the actors.

Some rules that must be followed while drawing a use case diagram:

1. A pertinent and meaningful name should be assigned to the actor or a use case of a system.
2. The communication of an actor with a use case must be defined in an understandable way.
3. Specified notation should be used as and when required.
4. The most significant interactions should be represented among the multiplicity of interactions between the use case and actors.



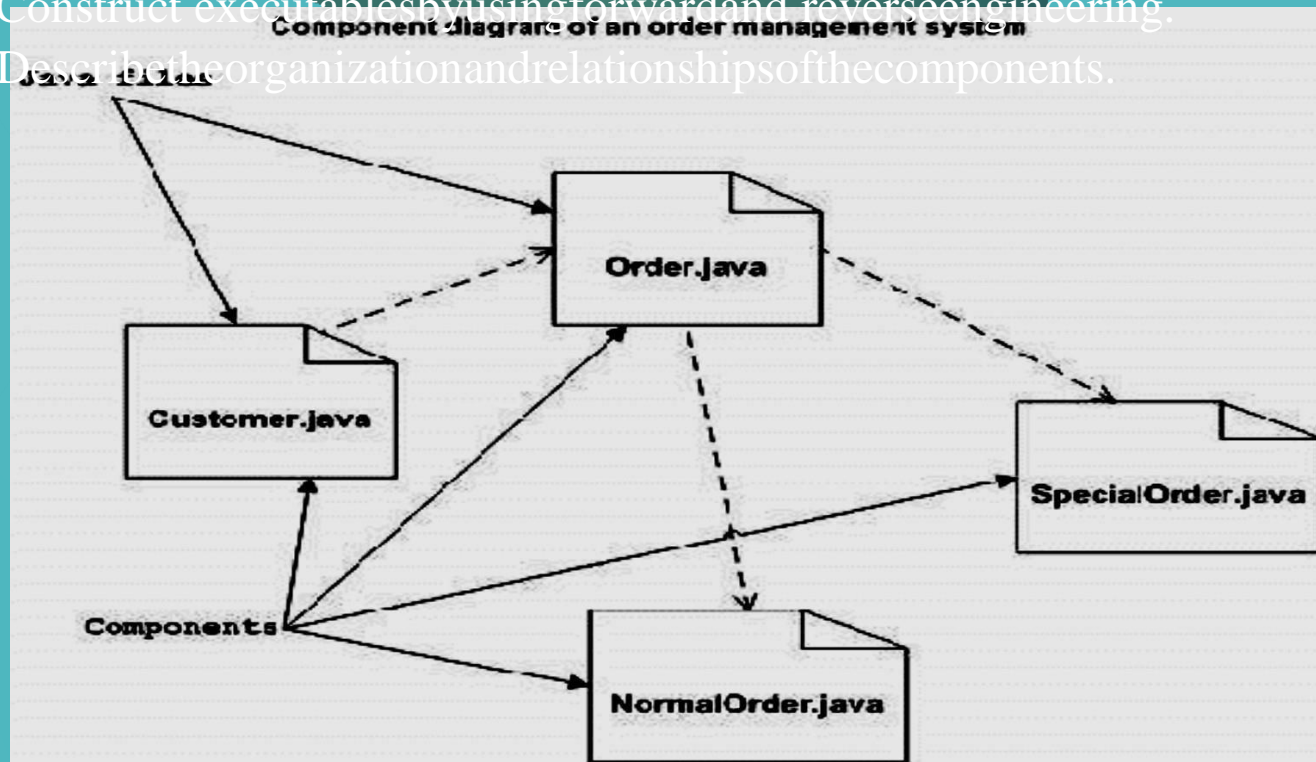


Component diagram:

- ▶ used to model the physical aspects of a system.
- ▶ Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.

Purpose of Component Diagrams

- ▶ Visualize the components of a system.
- ▶ Construct executables by using forward and reverse engineering.
- ▶ Describe the organization and relationships of the components.





NARASIMHAREDDYENGINEERINGCOLLEGE

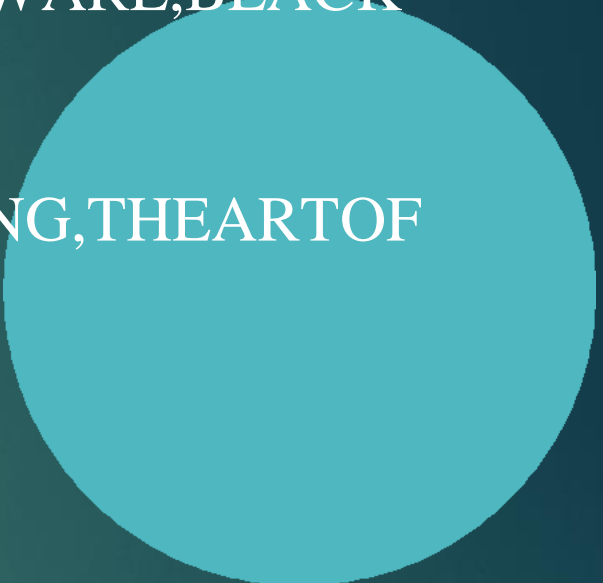

FACULTY:Dr.VenkateswaruluNaik

CS3102PC:SOFTWAREENGINEERING

III YEARB.TECH. CSE I-SEM

(R20)UNIT-IV



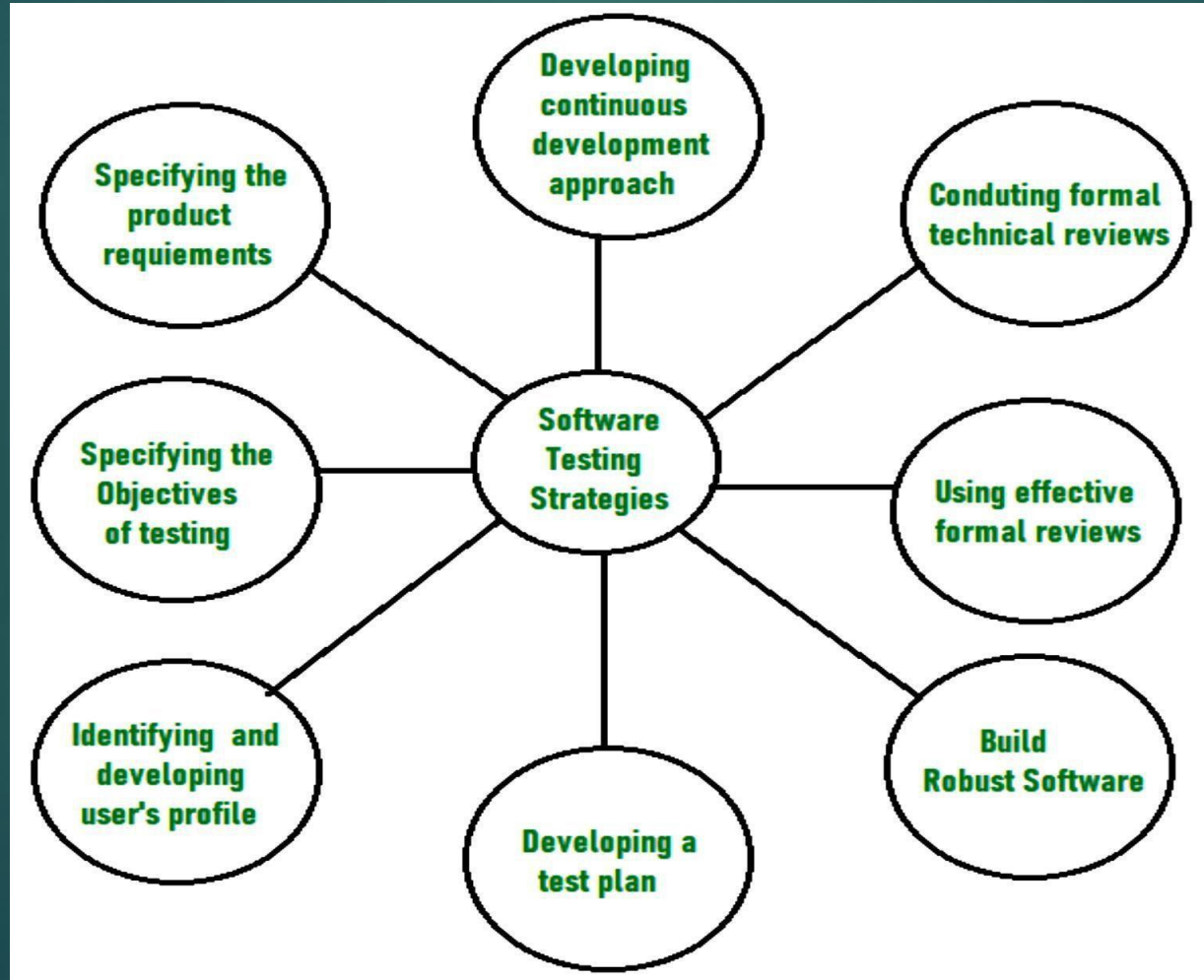


TESTING STRATEGIES: A STRATEGIC APPROACH TO SOFTWARE TESTING, TESTING STRATEGIES FOR CONVENTIONAL SOFTWARE, BLACK-BOX AND WHITE-BOX TESTING, VALIDATION TESTING, SYSTEM TESTING, THE ART OF DEBUGGING.

PRODUCT METRICS: SOFTWARE QUALITY, METRICS FOR ANALYSIS MODEL, METRICS FOR DESIGN MODEL, METRICS FOR SOURCE CODE, METRICS FOR TESTING, METRICS FOR MAINTENANCE.

Astrategic Approach for Software testing:

Software Testing is a type of investigation to find out if there is any default or error present in the software.



Testing Strategies for Conventional Software:

- ▶ you can wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors.
- ▶ you could conduct tests on a daily basis, whenever any part of the system is constructed.

Types:

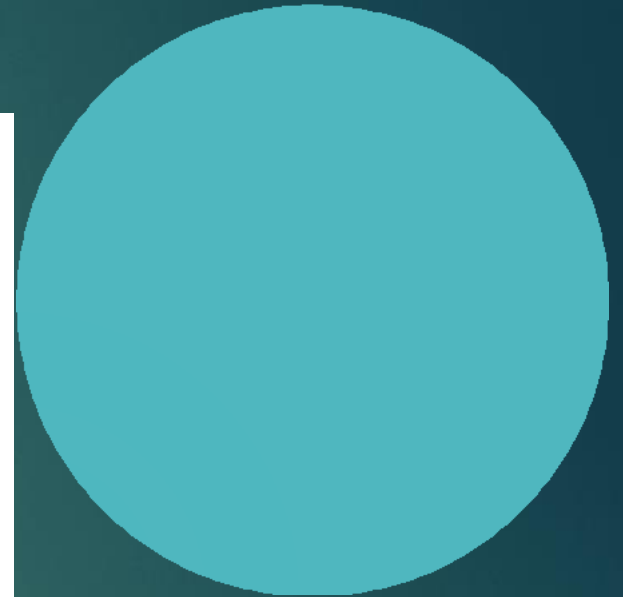
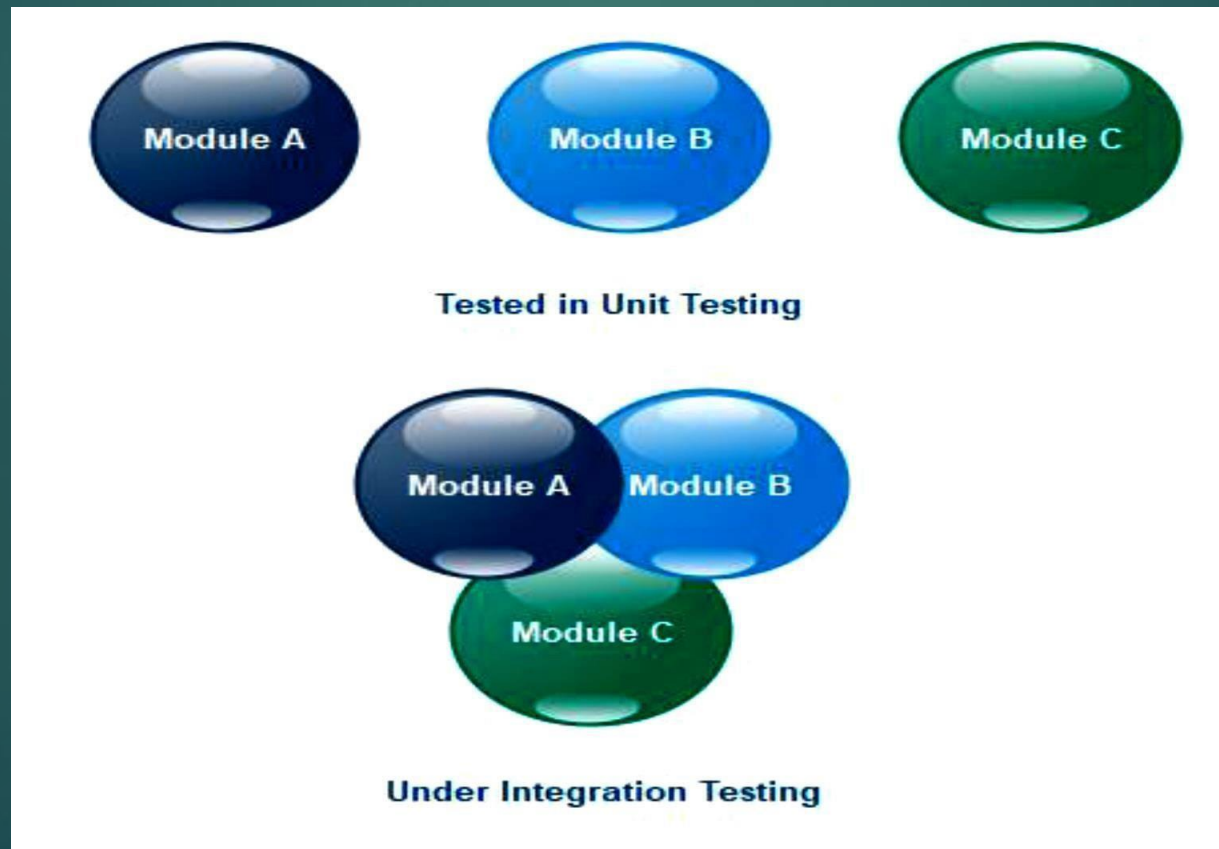
- 1) Unit Testing
- 2) Integration Testing
- 3) Validation Testing and
- 4) System Testing

1) Unit Testing:

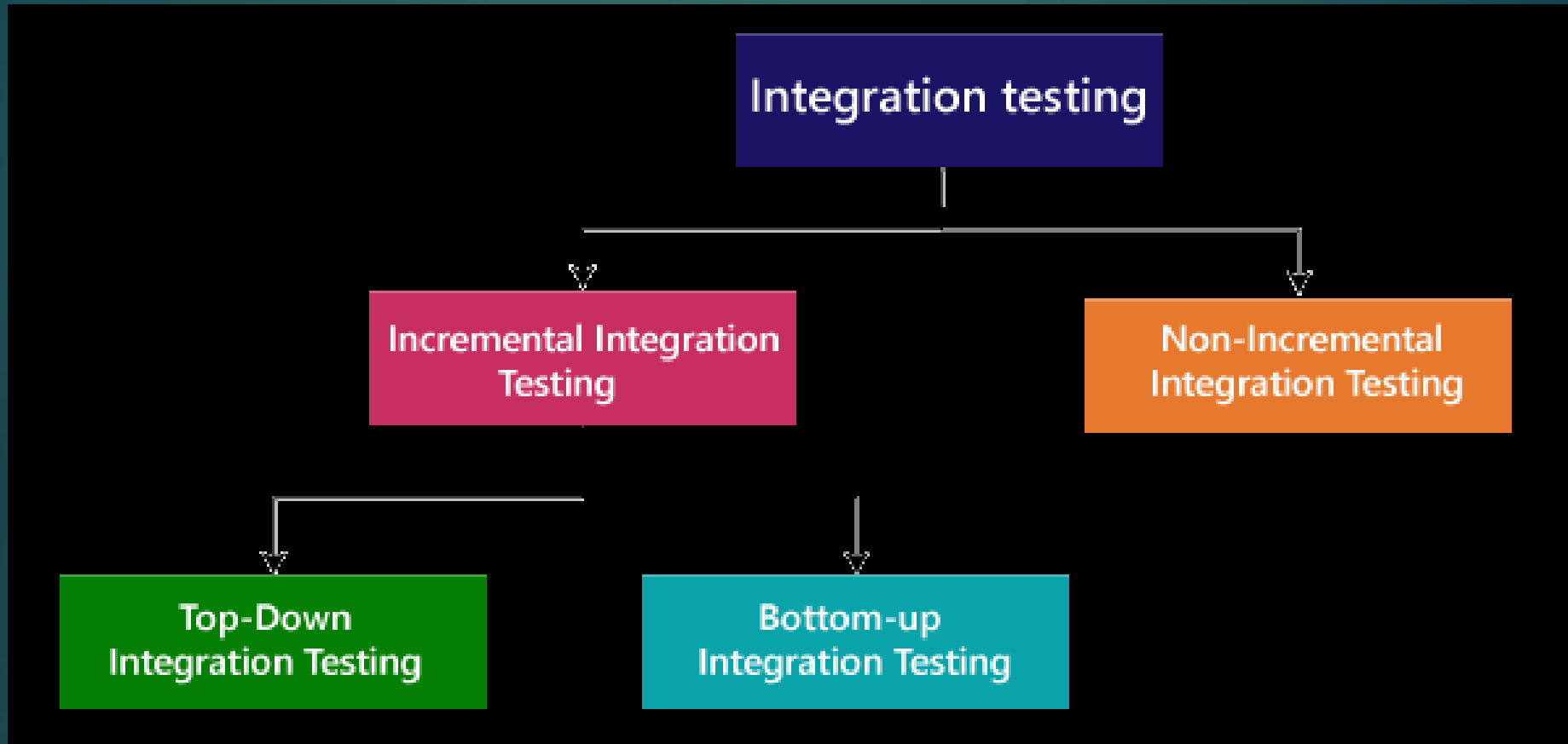
- ▶ individual units or components of a software are tested.
- ▶ done during the development (coding phase) of an application by the developers.
- ▶ concerned with functional correctness of the standalone modules

2) Integration Testing:

- ▶ second level of the software testing process comes after unit testing.
- ▶ The focus of the integration testing level is to expose defects at the time of interaction between integrated components or units.



Types of Integration Testing:

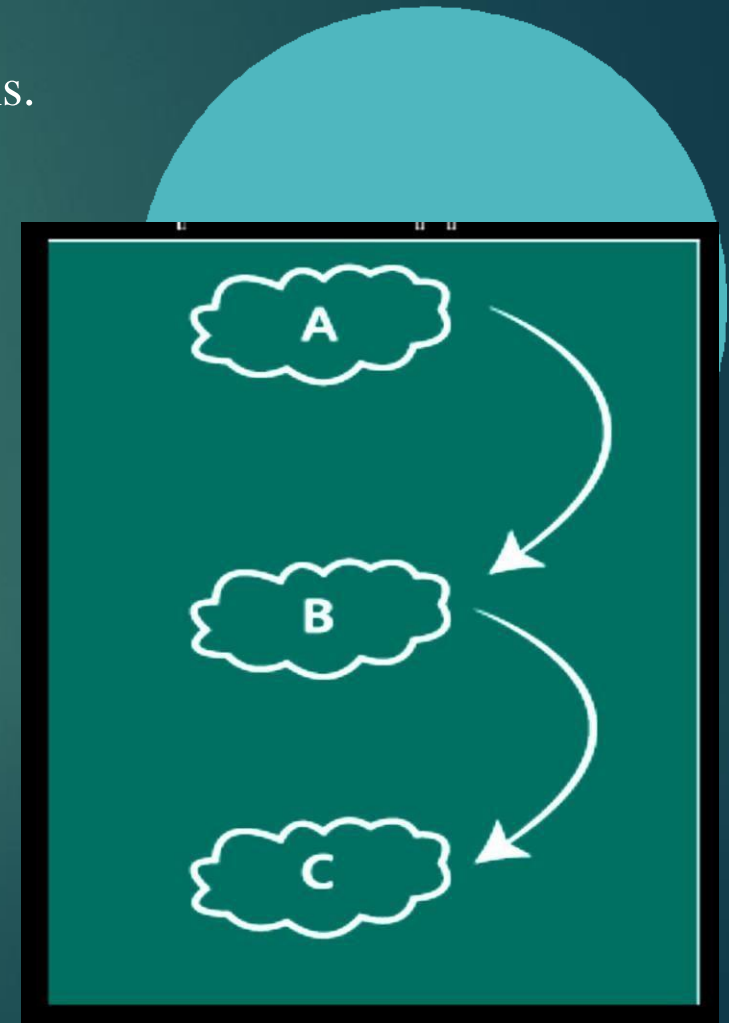


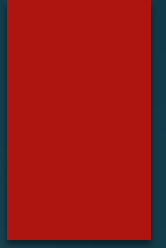
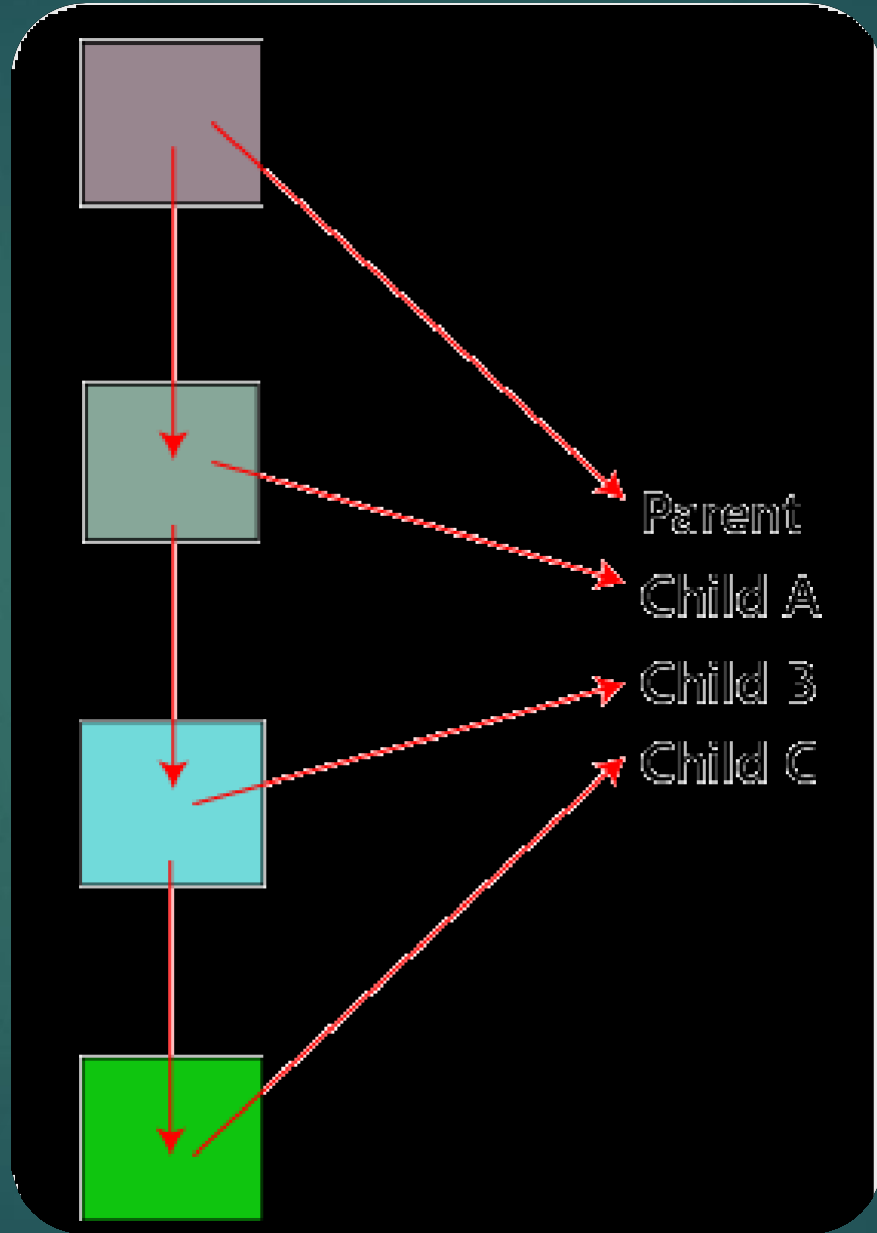
Incremental Approach:

- ▶ modules are added in ascending order one by one or according to need.
- ▶ Generally, two or more than two modules are added and tested to determine the correctness of functions.
- ▶ The process continues until the successful testing of all the modules.

Top-Down Approach:

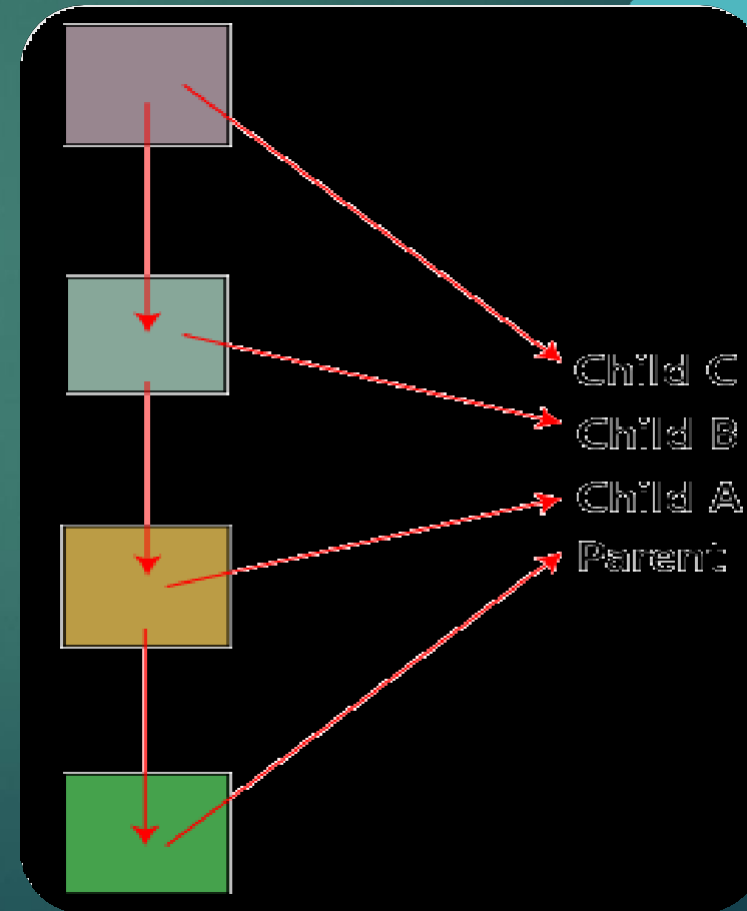
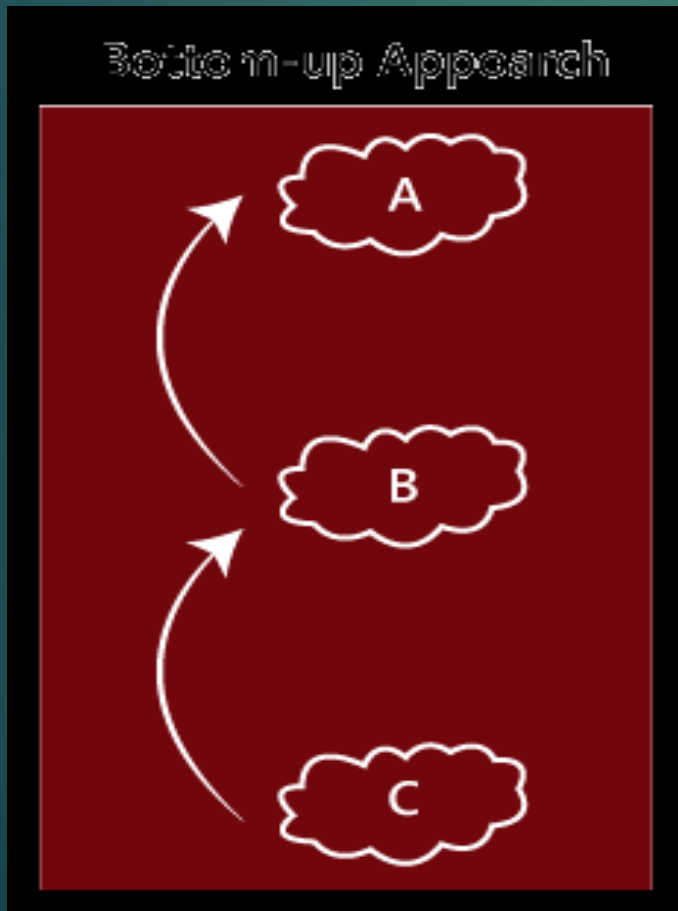
- ▶ deals with the process in which higher level modules are tested with lower level modules.
- ▶ Major design flaws can be detected and fixed early.
- ▶ the module we are adding is the **child of previous one.**





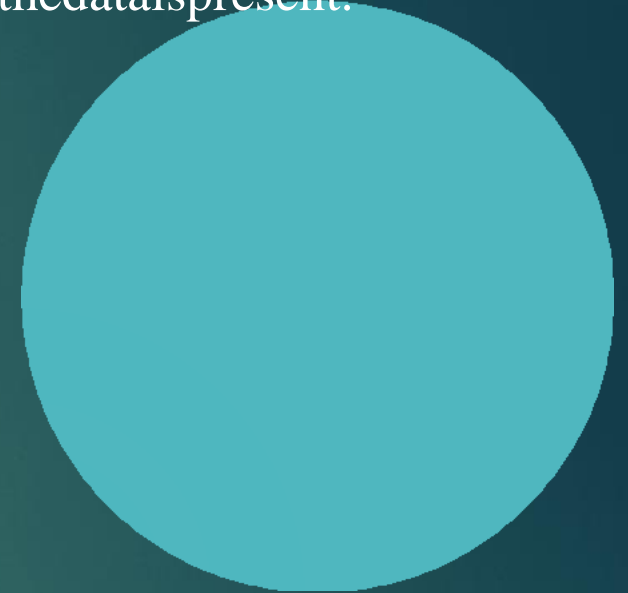
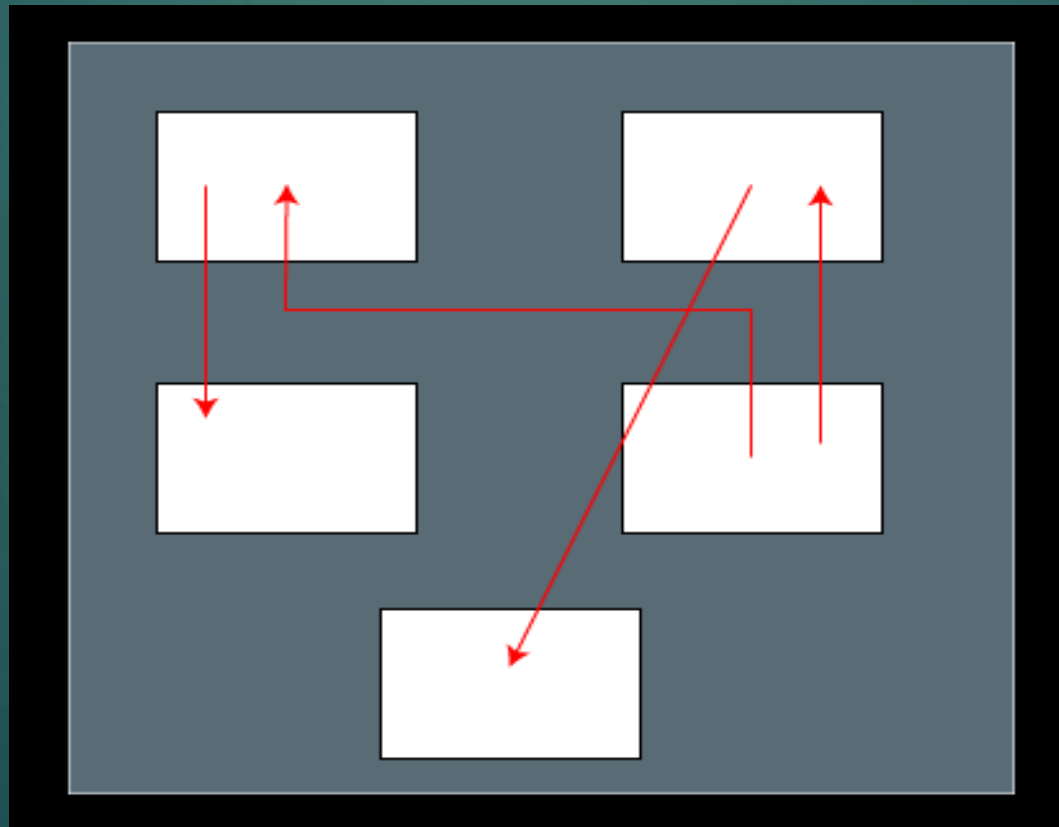
Bottom-Up Method:

- ▶ the process in which lower level modules are tested with higher level modules.
- ▶ Top level critical modules are tested at last.
- ▶ the modules we are adding are the parent of the previous one.



Non-incremental integration testing:

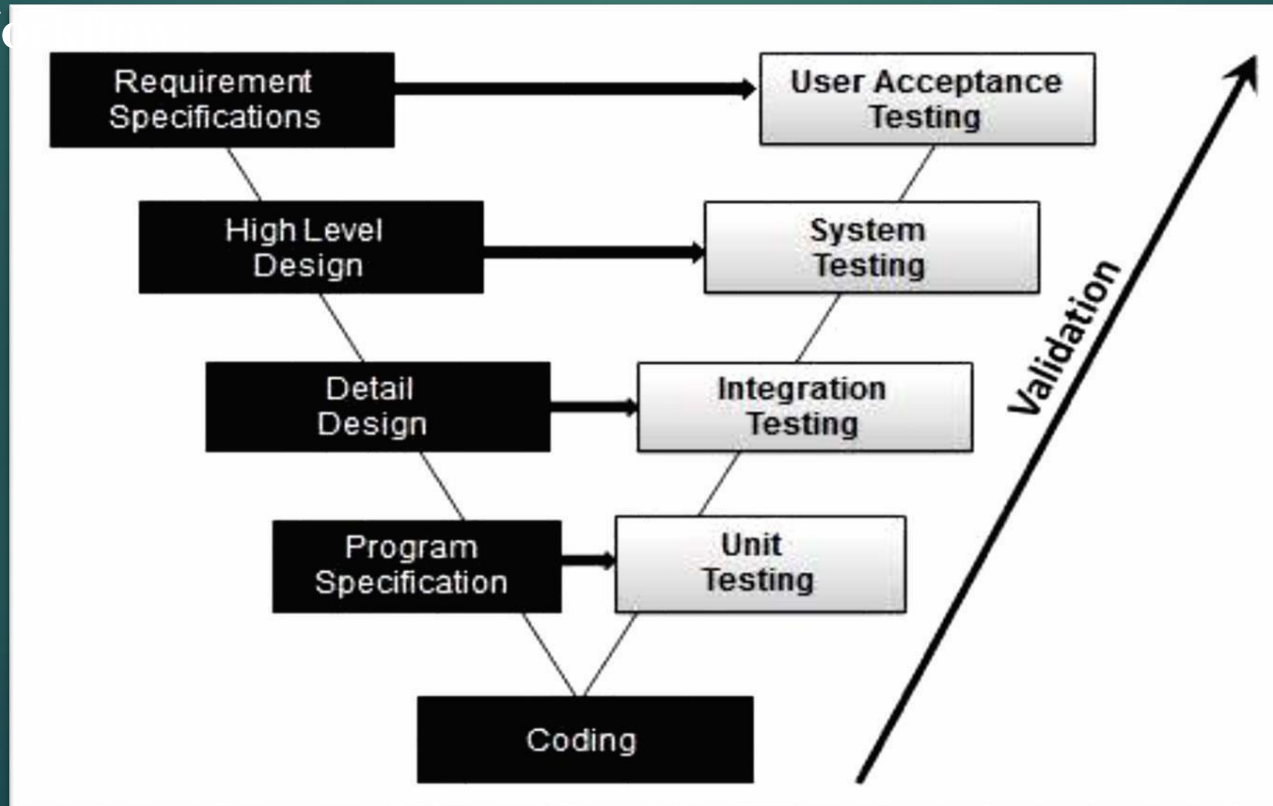
- ▶ when the data flow is very complex and when it is difficult to find who is a parent and who is a child.
- ▶ create the data in any module bang on all other existing modules and check if the data is present.
- ▶ it is also known as the **Big bang method**.



3. Validation Testing:

- ▶ Tester performs functional and non-functional testing.
- ▶ Validation testing is also known as dynamic testing.

▶ ensuring that "we have developed the product right." Validation Testing- We



4. System Testing:

- ▶ include testing of a fully integrated software system.
- ▶ To check the end-to-end flow of an application or the software as a user.

System Testing includes the following steps.

- ▶ Verification of input functions of the application to test whether it is producing the expected output or not.
- ▶ Testing of integrated software by including external peripherals to check the interaction of various components with each other.
- ▶ Testing of the whole system for End-to-End testing.
- ▶ Behavior testing of the application via a user's experience

Software Testing

Two major categories of software testing

- ▶ Blackbox testing
- ▶ Whitebox testing

Blackbox testing

- ▶ functionalities of software applications are tested.
- ▶ mainly focuses on input and output of software applications.

Types of Black Box Testing

- ▶ Functional testing
- ▶ Non-functional testing
- ▶ Regression

Black Box Testing Techniques

- ▶ Equivalence Class Partitioning
- ▶ Boundary Value Analysis
- ▶ Decision Table Testing



Equivalence Partitioning Testing:

- ▶ can be applied to all levels of software testing like unit, integration, system.
- ▶ divides the input test data of the application under test into each partition at least once of equivalent data from which test cases can be derived.

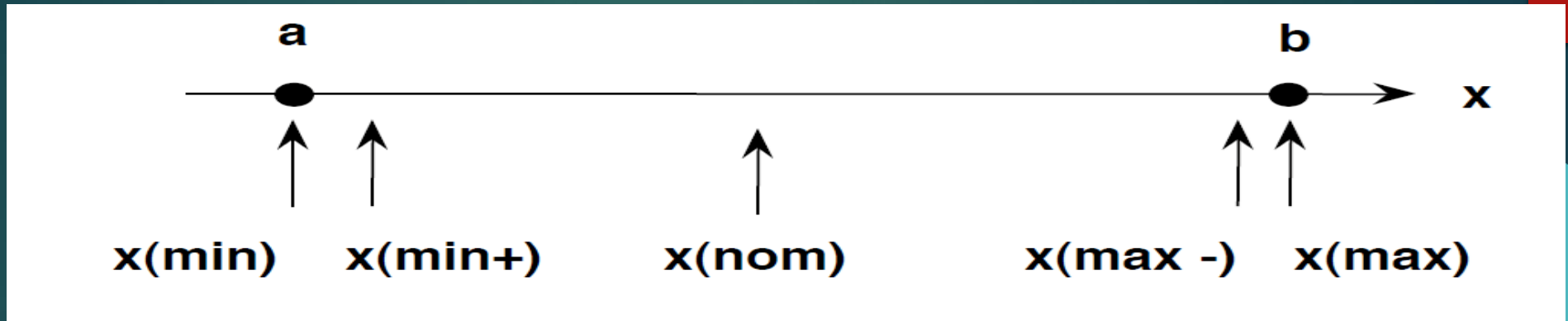
Example:

Assume that the application accepts an integer in the range 100 to 999

- ▶ Valid Equivalence Class partition: 100 to 999 inclusive.
- ▶ Non-valid Equivalence Class partitions: less than 100, more than 999, decimal numbers and alphabets/non-numeric characters.

Boundary Value Analysis:

- ▶ testing between extreme ends or boundaries between partitions of the input values.
- ▶ these extreme ends like Start-End, Lower-Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called "boundary testing".



Example:

InputBoxshouldaccepttheNumber1to10Test

ScenarioDescription

ExpectedOutcome

- | | |
|---------------------|-----------------------|
| ▶ Boundary Value=0 | SystemshouldNOTaccept |
| ▶ Boundary Value=1 | Systemshouldaccept |
| ▶ Boundary Value=2 | Systemshouldaccept |
| ▶ Boundary Value=9 | Systemshouldaccept |
| ▶ Boundary Value=10 | Systemshouldaccept |
| ▶ Boundary Value=11 | SystemshouldNOTaccept |

Equivalence and Boundary Value:

- ▶ Pizza values 1 to 10 is considered valid. A success message is shown.
- ▶ While value 11 to 99 are considered invalid for order and an error message will appear, "Only 10 Pizzas can be ordered"

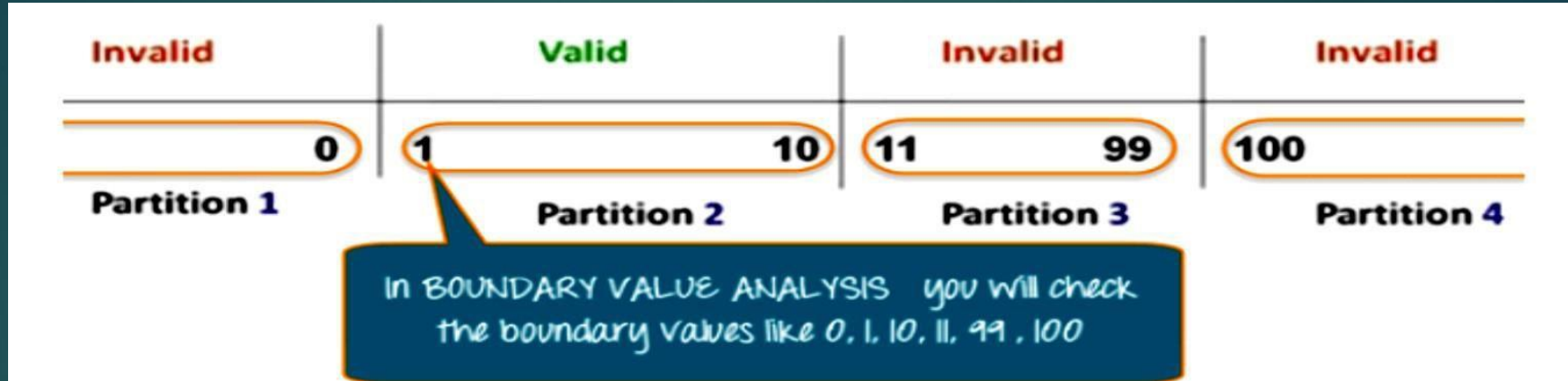
Order Pizza:

Here is the test condition

1. Any Number greater than 10 entered in the Order Pizza field (lets say 11) is considered invalid.
2. Any Number less than 1 that is 0 or below, then it is considered invalid.
3. Numbers 1 to 10 are considered valid
4. Any 3 Digit Numbers say - 100 is invalid.

Invalid	Valid	Invalid	Invalid
0	1	10	11
		99	100
Partition 1	Partition 2	Partition 3	Partition 4

Equivalence Class Partitioning



Boundaryvalueanalysis

DecisionTable:

- ▶ atabularrepresentationofinputsversusrules/cases/testconditions.
- ▶ TheconditionsareindicatedasTrue(T)andFalse(F)values.

✉

🔒

Conditions

Rule1	Rule2	Rule3	
Rule4Username(T/F)		F	
T	F	TPassword(T/F)	
F	F	T	
TOutput (E/H)		E	E
E	H		

- ▶ Case 1–Username and password both were wrong. The user is shown an error message.
- ▶ Case 2– Username was correct, but the password was wrong. The user is shown an error message.
- ▶ Case 3– Username was wrong, but the password was correct. The user is shown an error message.
- ▶ Case 4– Username and password both were correct, and the



your roots to success

WhiteBoxTesting:

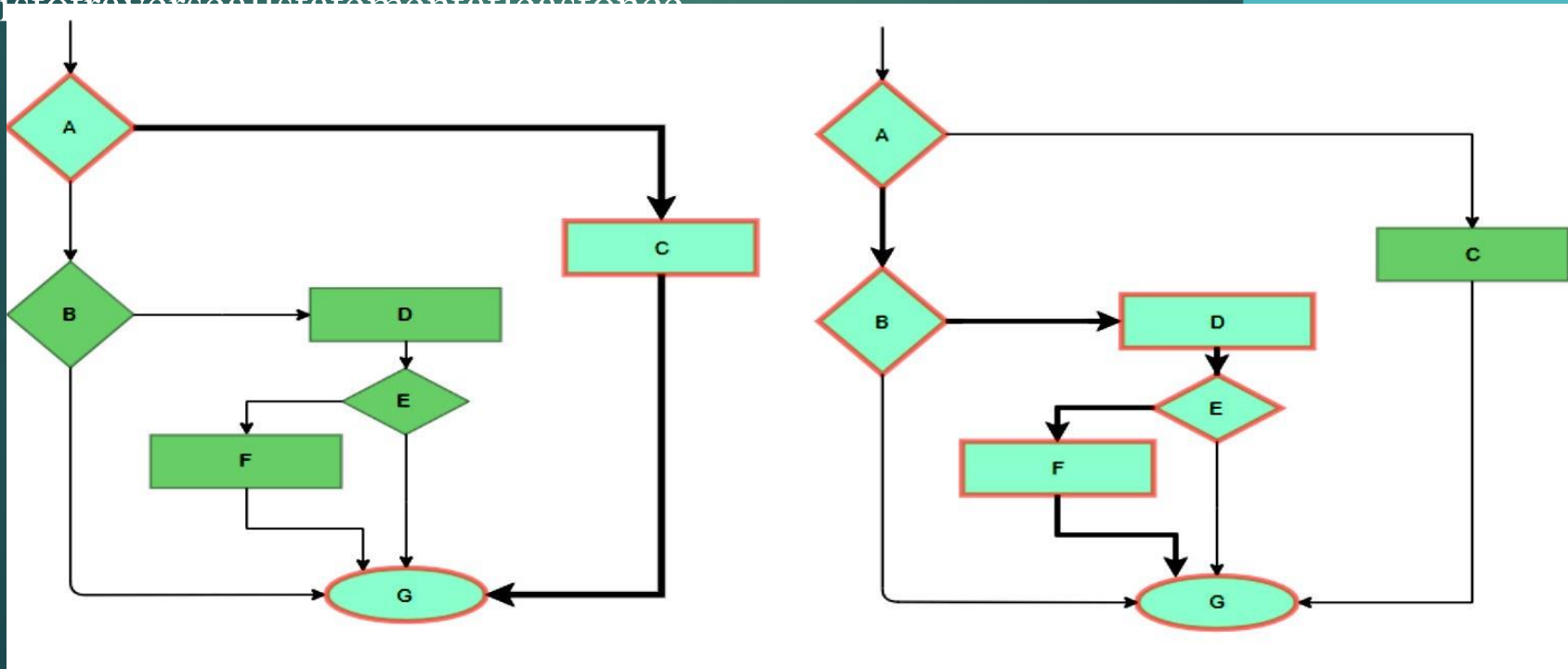
- ▶ examines the program structure and derives test data from the program logic/code.

WhiteBoxTestingTechniques:

- ▶ Statement Coverage
- ▶ Branch Coverage
- ▶ Path Coverage

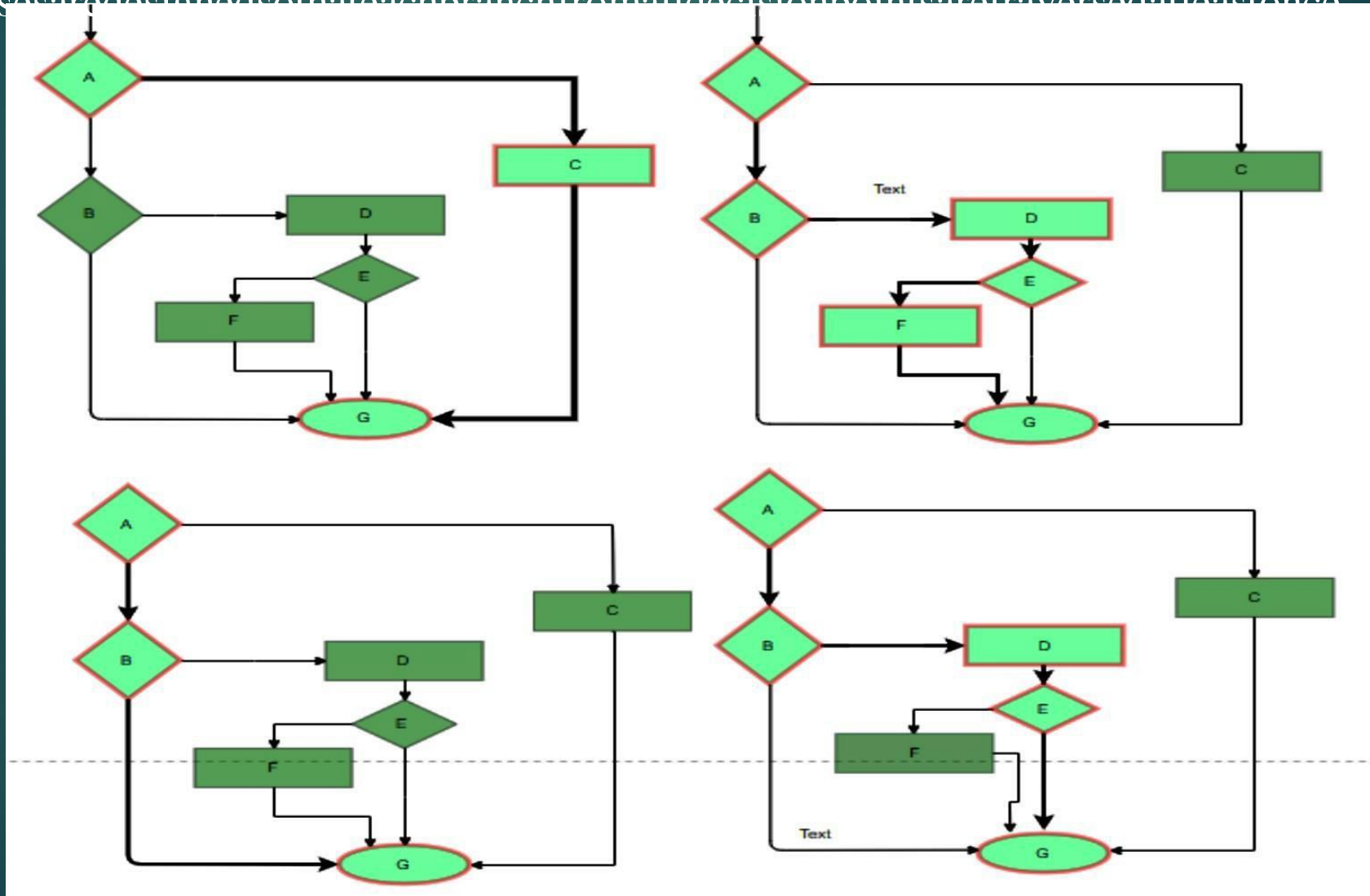
Statement coverage:

- ▶ the aim is to traverse all statements at least once



Branch Coverage:

- ▶ testcases are designed so that each branch from all decision points are traversed at least once



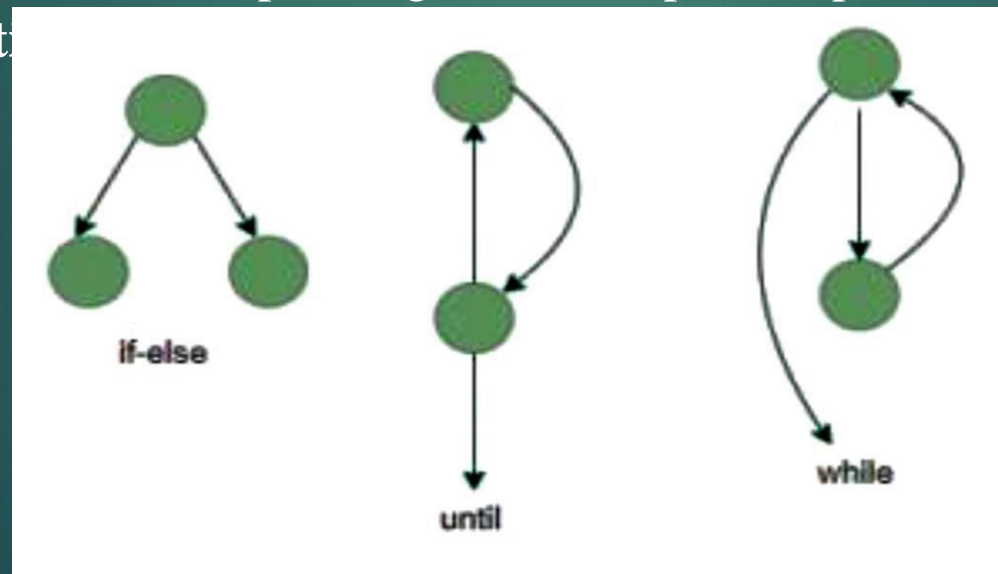
Basis Path Testing:

- ▶ control flow graphs are made from code or flowchart and then Cyclomatic complexity is calculated which defines the number of independent paths.

Steps:

1. Make the corresponding control flow graph
2. Calculate the cyclomatic complexity
3. Find the independent paths
4. Design test cases corresponding to each independent path

Flow graph notation



Software Quality:

- ▶ Software quality product is defined in terms of its fitness of purpose.
- ▶ a quality product does precisely what the users want it to do.

Several quality methods:

- ▶ Portability
- ▶ Usability
- ▶ Reusability
- ▶ Correctness
- ▶ Maintainability

Various metrics formulated for products in the development process are listed below.

- ▶ Metrics for analysis model
- ▶ Metrics for design model
- ▶ Metrics for source code
- ▶ Metrics for testing
- ▶ Metrics for maintenance

Metrics for the Analysis Model:

- ▶ used to examine the analysis model with the objective of predicting the size of the resultant system.
- ▶ Function point and lines of code are the commonly used methods for size estimation.

1. Function Point (FP) Metric:

- ▶ measure the functionality delivered by the system, estimate the effort, predict the number of errors, and estimate the number of components in the system.
- ▶ Function point is derived by using a relationship between the complexity of software and the information domain value.

2. Lines of Code (LOC):

- ▶ most widely used methods for size estimation.
- ▶ defined as the number of delivered lines of code, excluding comments and blank lines.
- ▶ highly dependent on the programming language used as code writing varies from one programming language to another.