

An Introduction to Software Engineering

University of Baghdad 2016

Dr. Afaf Al-Kaddo

Chapter One: Introduction to Software Engineering

1.1 The Computer Software Definition

1.2 Software Engineering Definition

1.3 The characteristic of software engineer

1.4 Software Characteristics

1.5 Software Applications

1.6 Software: A crisis on the horizon

1.7 The Characteristics of Well-Engineered Software

1.8 The Goals of Software Engineering

What is software?

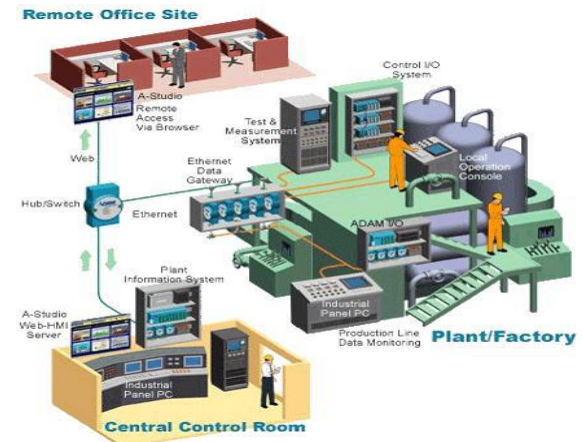
The software might take the following forms:

- ❑ **Instructions:** Computer programs, that when executed provide desired function and performance.
- ❑ **Data structured:** That enable the programs to manipulate information.
- ❑ **Documents:** That describes the operation and use of programs.

Types of Software



Generic



Custom

Generic Software Products

Developed to be sold to a range of different customers.

Examples:

- MS Office
- Photoshop
- Candy Crush Saga

Custom Software Products

Developed for a single customer according to their specification.

The user (or person paying for the software) controls the specification.

Example:

- Air traffic control systems
- Factory automation systems
- Building control systems

Computer Software Definition

It is the product that software engineers design and build. It encompasses **programs** that execute within a computer of any size and architecture, **documents** that encompass hard-copy and virtual forms, and **data** that combine numbers and text, **video**, and **audio** information.

Software Engineering Definition

- The **practical application** of **scientific knowledge** to the design and construction of *computer programs* and the *associated documentation* required to develop, operate, and maintain them.
- The **systematic approach** to the development, operation, maintenance, and retirement of software.
- The establishment and **use of engineering principles** (methods) in order to obtain economically software that is *reliable* and works on real machines.
- Multi-person construction of multi-version software.

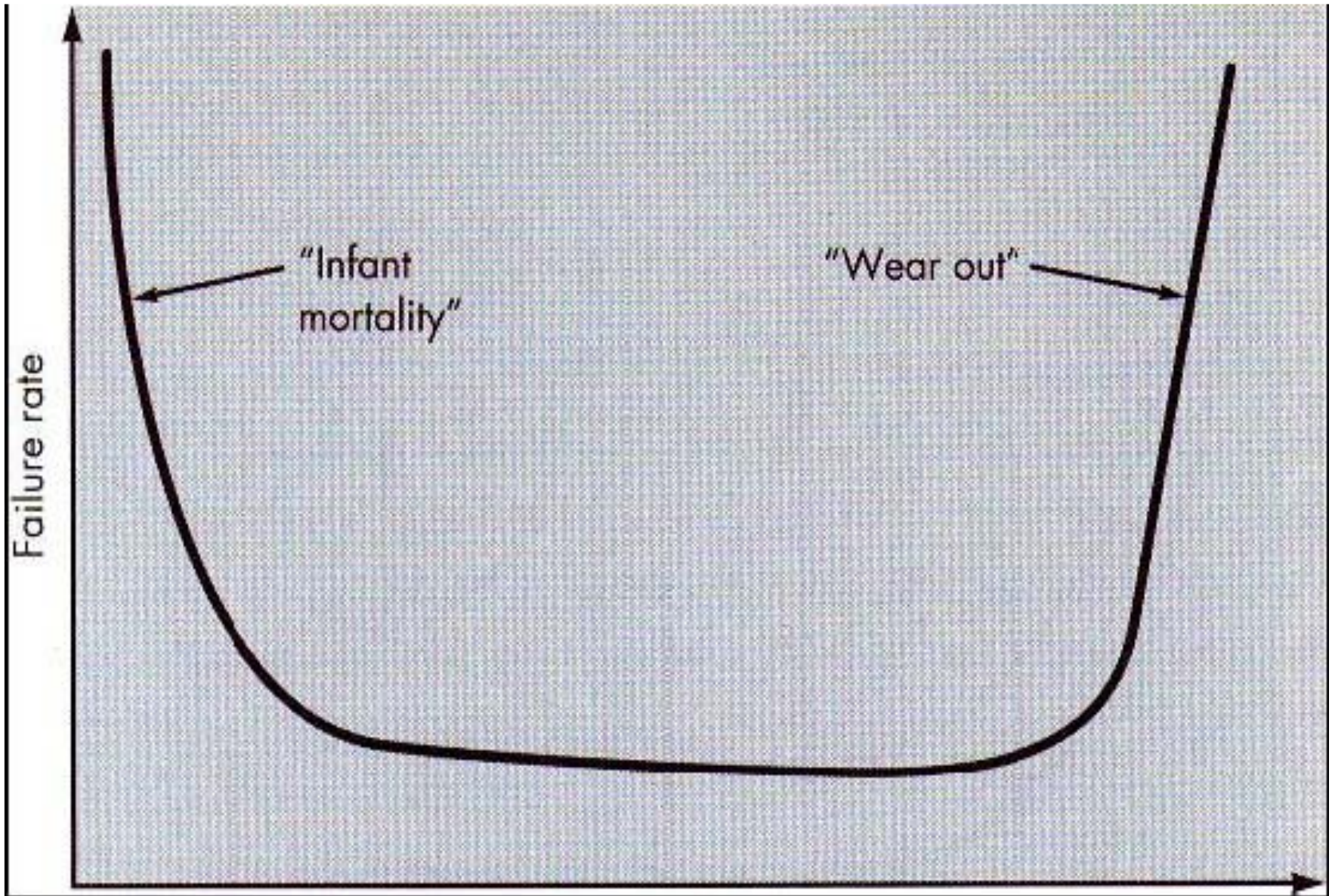
The characteristics of software engineer

- 1- Good programmer and fluent in one or more programming language.
- 2- Well versed data structure and approaches.
- 3- Familiar with several designs approaches.
- 4-Be able to translate not clear requirements and desires into precise specification (مواصفات).
- 5- Be able to converse with the user of the system in terms of application not in “computer”.
- 6- Able to a build a model.
- 7- Communication skills and interpersonal skills.

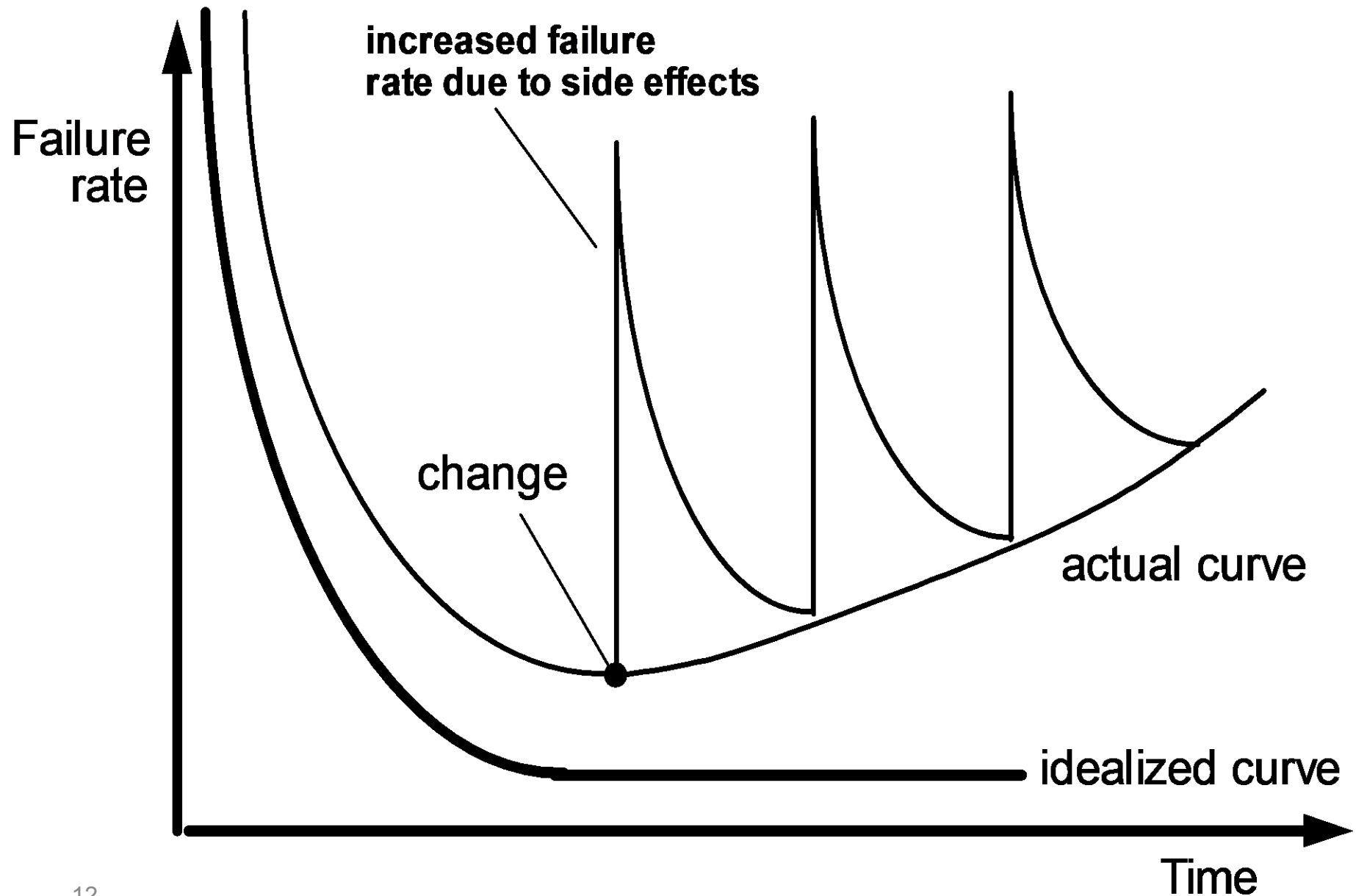
Software Characteristics

1. **Software is developed or engineered**; it is not manufactured in the classical sense. Some similarities exist between software development and hardware manufacture. In both activities, **high quality** is achieved through **good design**. Software costs are concentrated in engineering.
2. **Software doesn't "wear out"** يتآكل

Failure curve for hardware



Idealized and actual failure curves for software



Software Characteristics

3. Software continues to be custom built.

A software component should be designed and implemented so that it can be **reused** in many different programs(algorithms, data structure and encapsulation).

Software Applications

- 1. System software:** It is a collection of programs written to service other programs (compilers, editors) .
- 2. Real-time software:** Software that monitors/analyzes/controls real world events as they occur is called real time.
- 3. Business software:** Business information processing is the largest single software application area.
- 4. Engineering and scientific software:** Computer-aided design (CAD).

Software Applications

- 5. **Embedded software:** keypad control for a microwave.
- 6. **Personal computer software:** Such as (Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management).
- 7. **Web-based software:** The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., HTML, Perl, or Java).
- 8. **Artificial intelligence software:** It makes use of non-numerical algorithms to solve complex problems. Expert systems, pattern recognition.

Software: A crisis on the horizon

The term alludes to a set of problems that are encountered in the development of computer software.

Some “crisis” issues:

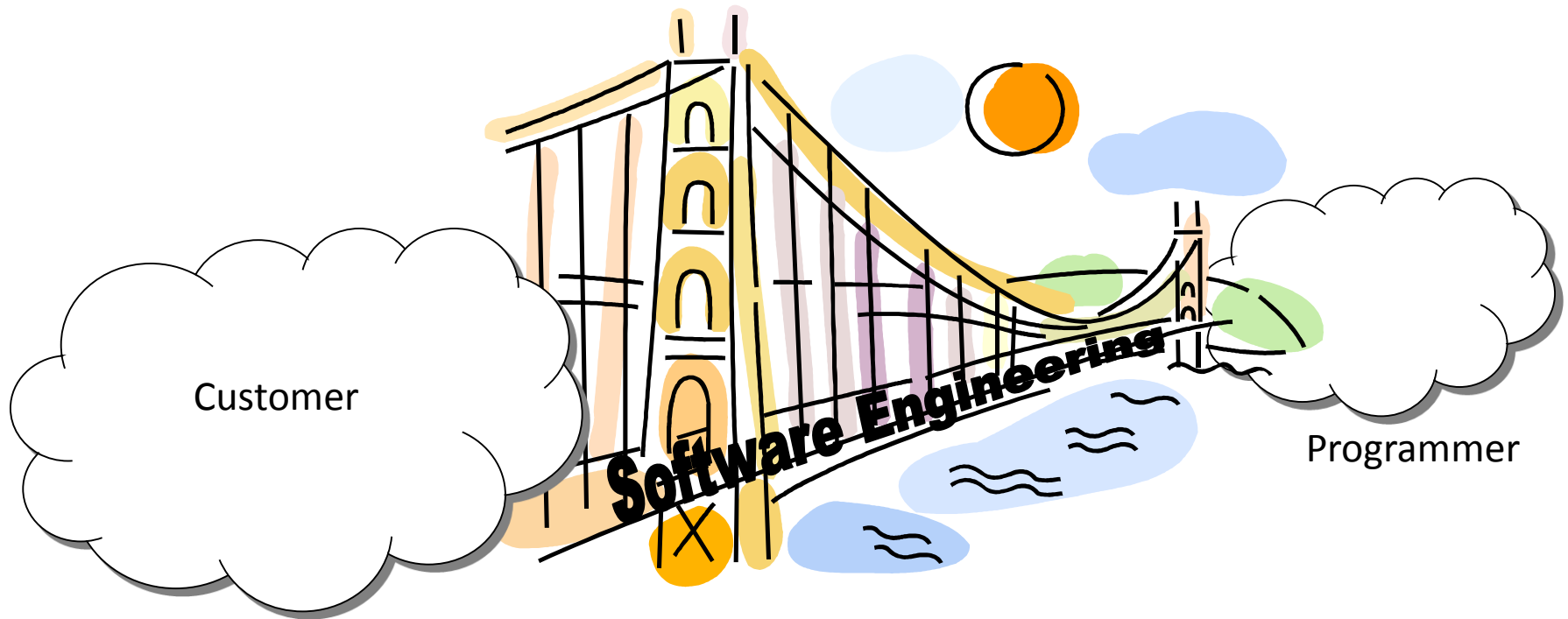
- Relative cost of hardware/software
- Low productivity
- “Wrong” products
- Poor quality
- Constant maintenance
- Technology transfer is slow

The Characteristics of Well-Engineered Software

- 1- **Maintainability:** software should be written in such a way that it may evolve to meet the changing needs of customer.
- 2- **Dependability:** software dependability has a range of characteristics, including reliability, security and safety.
- 3- **Efficiency:** software should not make wasteful use of system resources, such as memory and processor cycles.
- 4- **Usability:** software must be usable, without under effort by the type of user for whom it is designed.

The Role of Software Engg. (1)

A bridge from customer needs to programming implementation



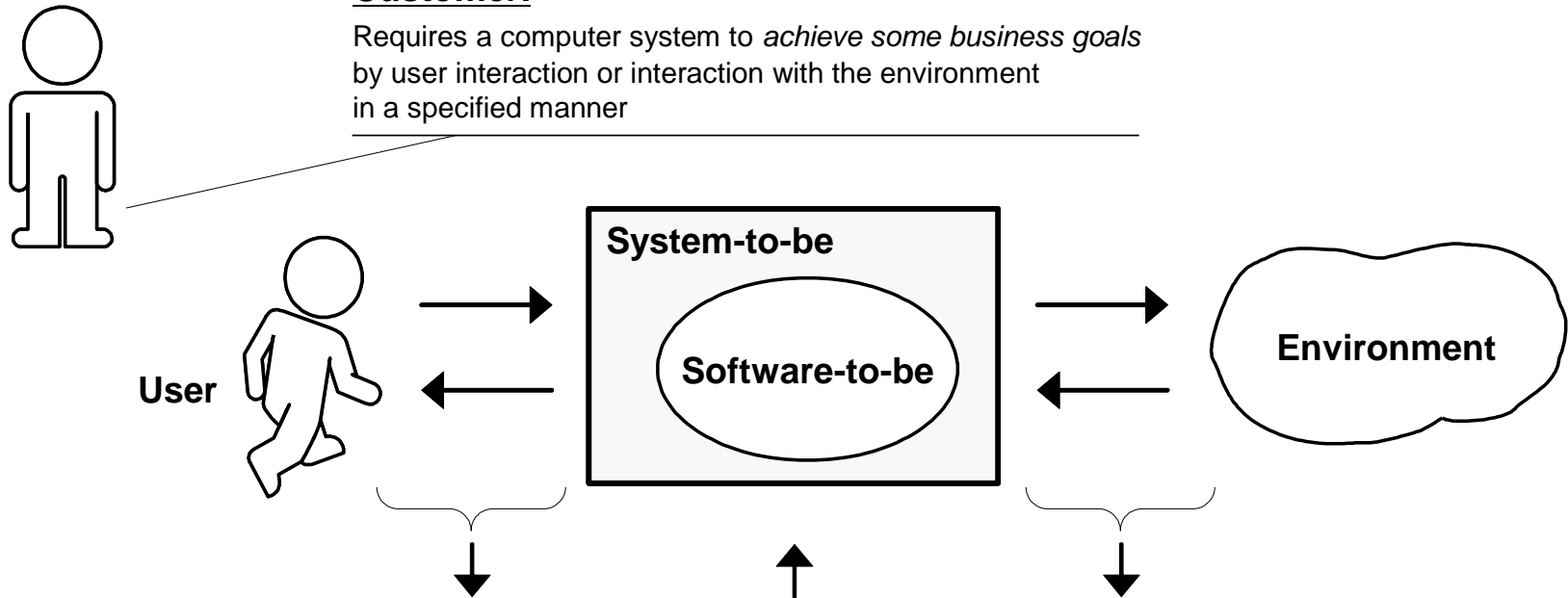
First law of software engineering

Software engineer is willing to learn the problem domain
(problem cannot be solved without understanding it first)

The Role of Software Engg. (2)

Customer:

Requires a computer system to *achieve some business goals* by user interaction or interaction with the environment in a specified manner



Software Engineer's task:

To **understand** *how* the system-to-be needs to interact with the user or the environment so that customer's requirement is met and **design** the software-to-be

May be the same person

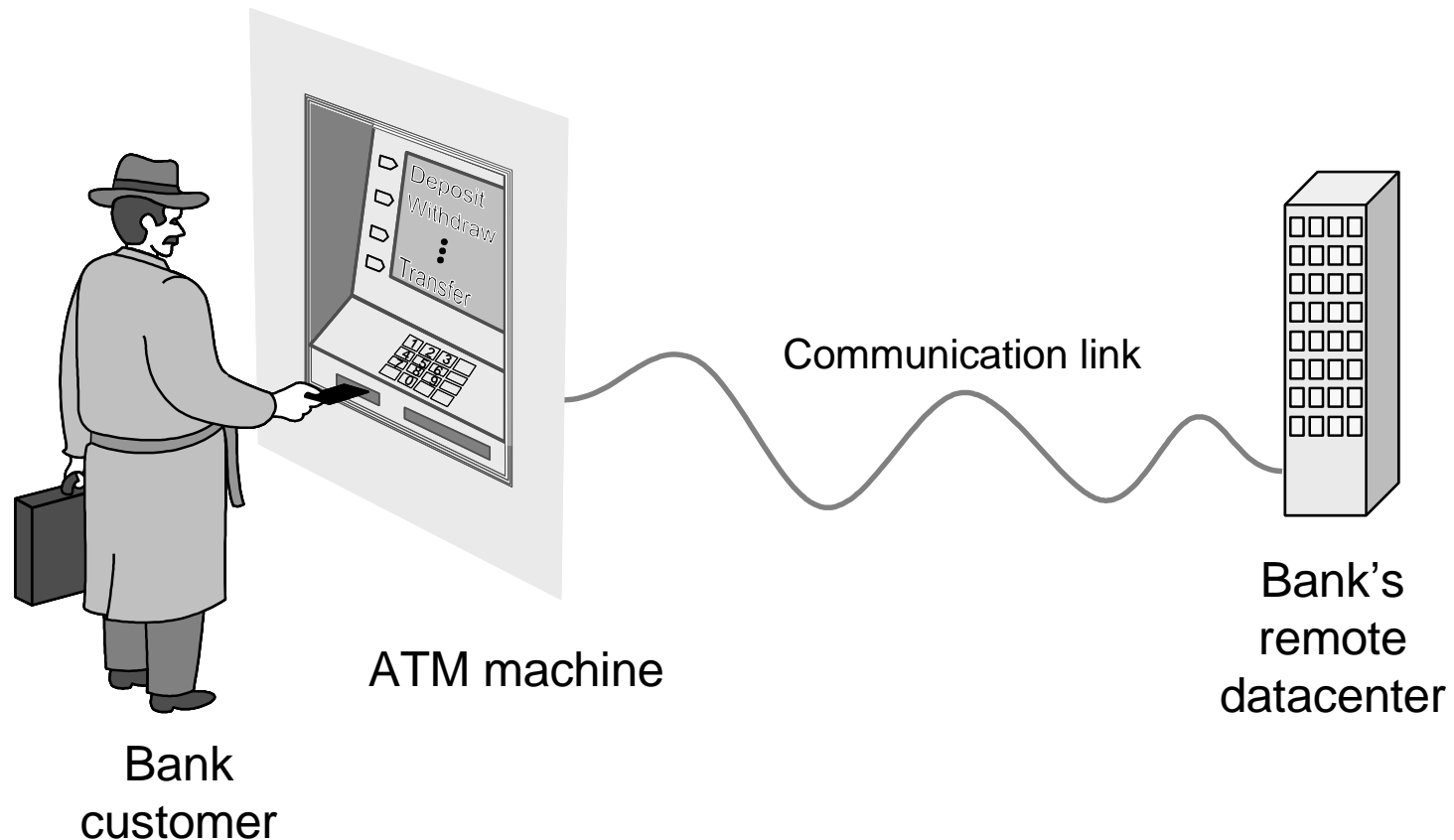


Programmer's task:

To **implement** the software-to-be designed by the software engineer

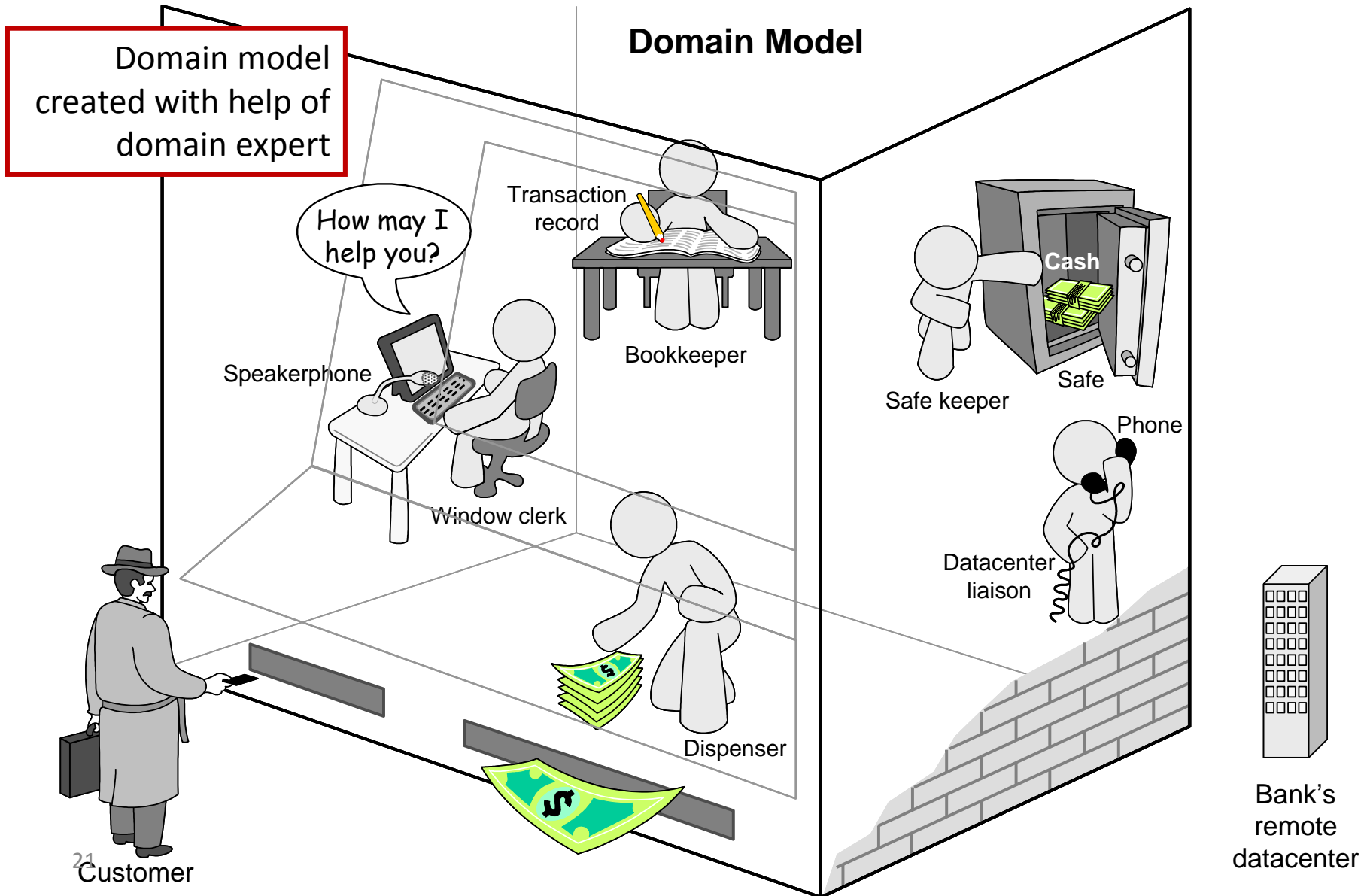
Example: ATM Machine

Understanding the money-machine problem:



How ATM Machine Might Work

In domain analysis, we consider the system as a “transparent box”



The Goals of Software Engineering

- ☐ Readability
- ☐ Correctness
- ☐ Reliability (Code must be reliable).
- ☐ Reusability
- ☐ Extensibility
- ☐ Flexibility
- ☐ Efficiency
- ☐ Need to understand requirements.
- ☐ Want software with maximum functionality.
- ☐ Cost to develop and maintain important.
- ☐ Want results as fast as possible.
- ☐ Must minimize development risks.

Chapter Two: Software Development Model

Topics Covered

2.1 Software Lifecycle

2.2 Linear Sequential Waterfall Model

2.3 Prototyping Model

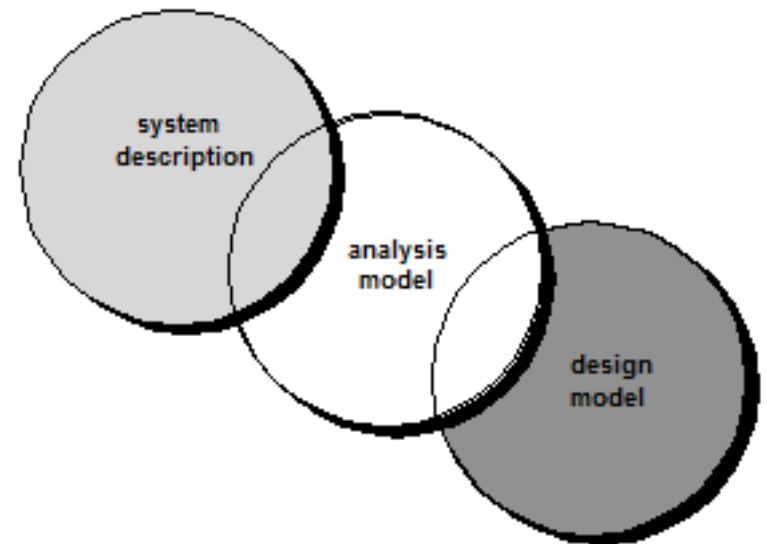
2.4 Incremental Model

2.5 Spiral Model

Software Lifecycle

A software engineering lifecycle model describes how one might **put structure** on the **software engineering activities**. **Each software product proceeds to a number of distinct stages, these are:**

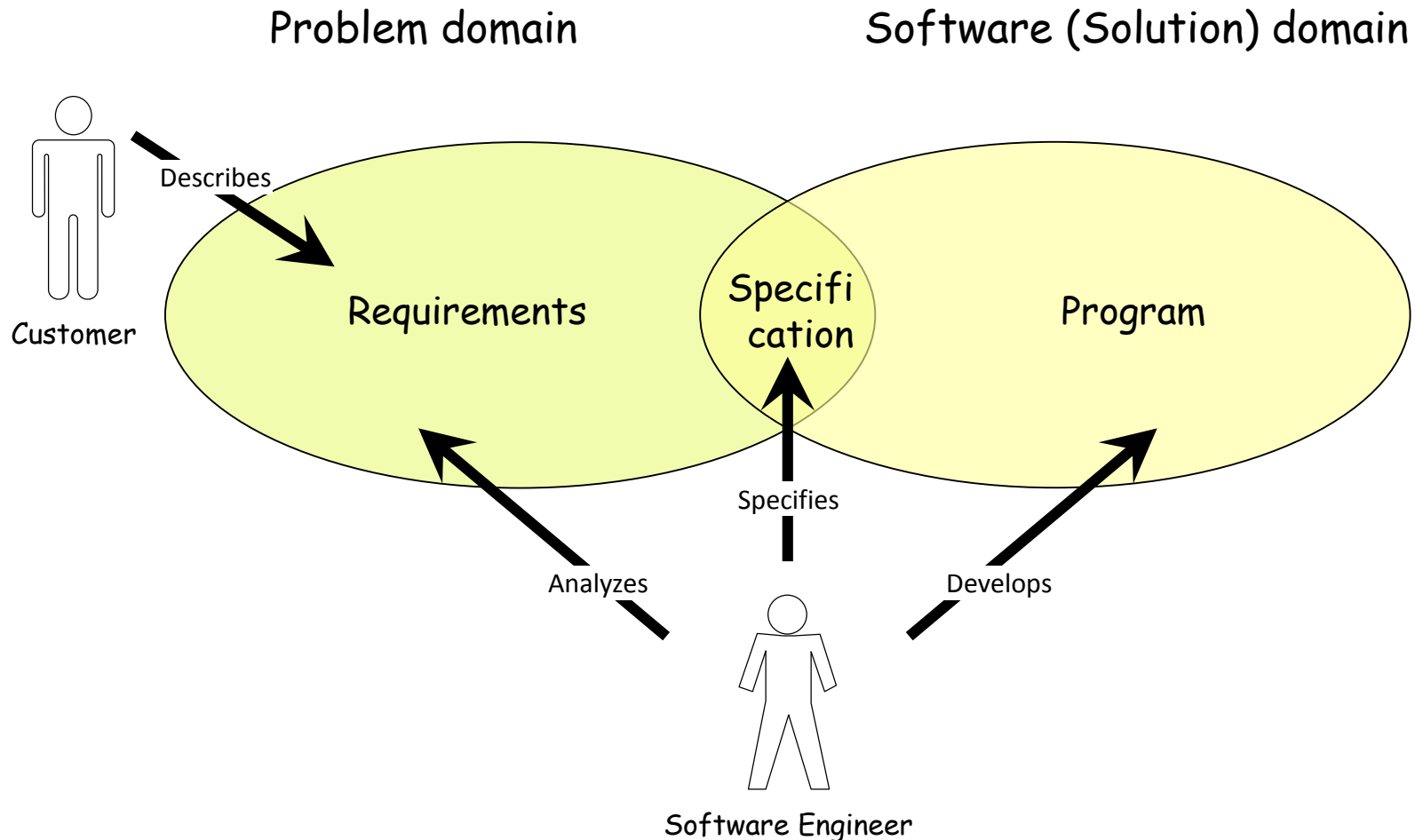
- 1- Requirements engineering
- 2- Software design
- 3- Software construction
- 4- Validation and verification
- 5- Software testing
- 6- Software deployment
- 7- Software maintenance



Requirements Engineering Components

- **Requirements gathering**
 - (“requirements elicitation”) helps the customer to define what is required: what is to be accomplished, how the system will fit into the needs of the business, and how the system will be used on a day-to-day basis
- **Requirements analysis**
 - refining and modifying the gathered requirements
- **Requirements specification**
 - documenting the system requirements in a semiformal or formal manner to ensure clarity, consistency, and completeness

Requirements and Specification



1. **Requirements Engineering:** is the interface between customers and developers on a software project.
 2. **Software Design:** the designers convert the logical software requirements from stage 1 into a software design by describe the software in such a way that programmers can write line of code that implement what the requirements specify.
 3. **Software Construction:** is concerned with implementing the software design by means of programs in one or more programming languages. This stage content several steps, these are :
 - A- **Software reuse:** (encapsulating effort in units of source code, which can be reused in other projects).
1. **Component based software engineering:** Building software systems from prefab software components .
 2. **Software product lines:** The goal of a software product line is to maintain a set of reusable core artifacts that are common to all systems in the product line.

- b. Security and reliability:** software must be dependable by making it reliable (work very well under any environments), secure and safety.
- c. Software documentation:** (User documentation, Technical documentation and Documentation generation).
- d. Coding Standards:** to ensure portability and make code maintainable by others than the original developer.

4. Validation and Verification

- a. Software Inspections:** are reviews of the code with the purpose of detecting defects.
- b. Software Testing:** testing each unit founded in this software, follow by testing software integration.

- 5. Software Deployment:** After development, software should be put to use (available to users, who can then **download, install, and activate** it).

Activities make up the SW deployment process are:

- Release
- Packaging
- Transfer
- Installation
- Configuration
- Activation
- De-activation
- Update
- Adapt
- De-installation
- De-release

6. Software Maintenance: after first release, software maintenance is needed to improve it (repair defects), and to extend it (add new functionality).

The software process

A structured set of activities required to develop a software system

- Specification
- Design
- Construction
- Validation
- Evolution

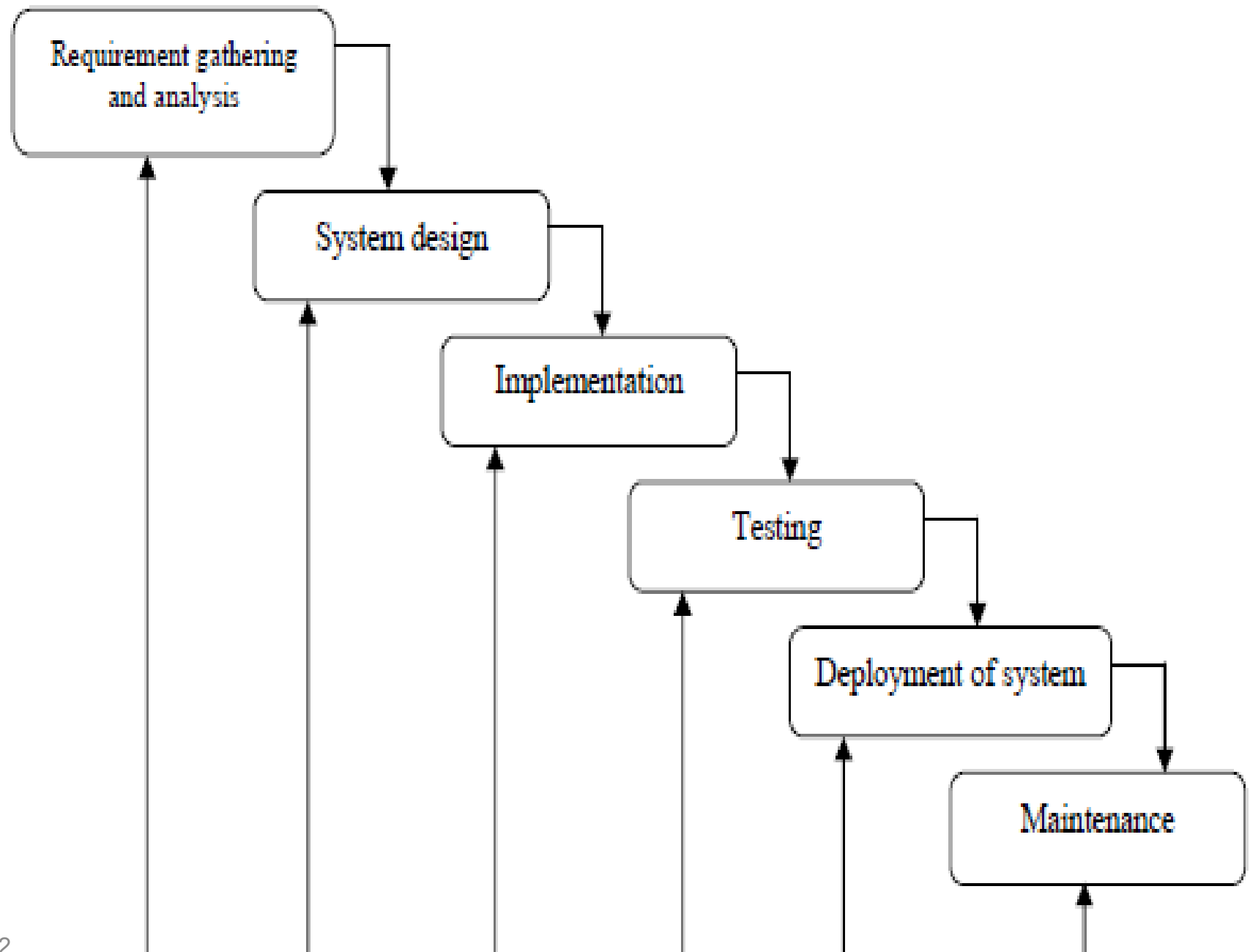
A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective

SW Process Models

Linear Sequential Waterfall Model

The whole process of software development is divided into **separate process** phases, these are:

- 1) Requirement Specifications (analysis and definition).
- 2) Software Design.
- 3) Implementation.
- 4) Testing.
- 5) Maintenance.



Waterfall Method

Requirements

Design

Implementation

Testing

Deployment &
Maintenance

***Waterfall
method***

Unidirectional, no way back
finish this step before moving to the next

Disadvantages of the Waterfall Model

1. Not all requirements are received at once.
2. The problems with one phase are never solved completely during that phase.
3. The project is not partitioned in phases in flexible way.
4. As the requirements of the customer goes on getting added to the list, not all the requirements are fulfilled, this results in development of almost unusable system.
5. difficulty of accommodating change after the process is underway

Therefore, this model is only appropriate when the requirements are well-understood

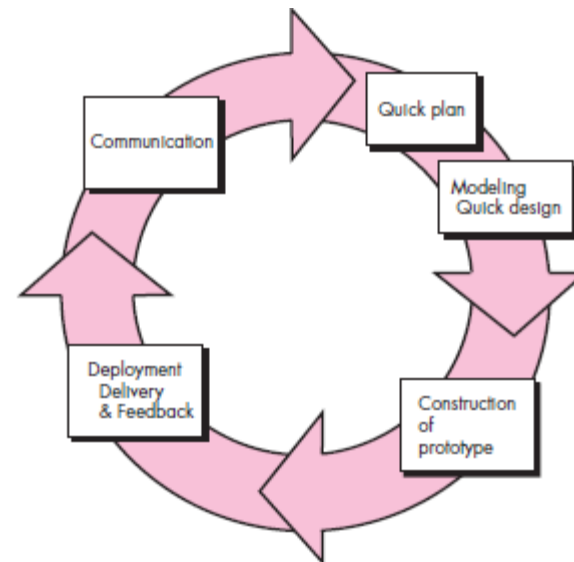
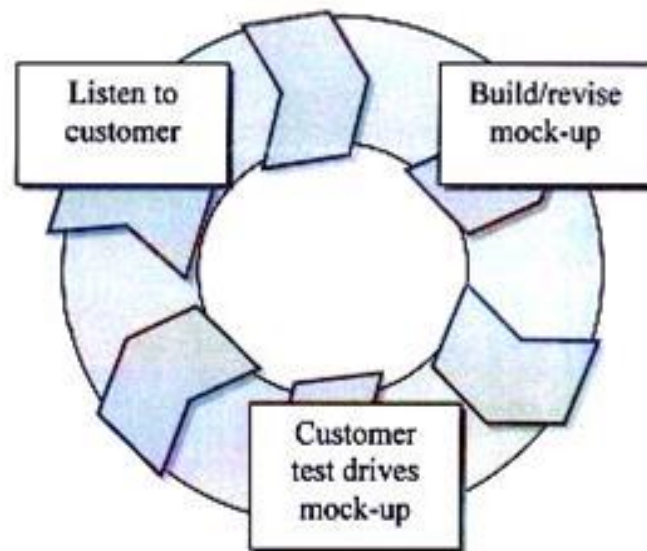
The Prototype Model

The prototype model is using for many reasons, such as:

1. A customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements.
2. The developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human/machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.

The stages of the prototyping model:

1) **Requirements gathering (listen to customer):** developer and customer meet and define the overall **objectives** for the software, **requirements**, and **outline areas**. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats).

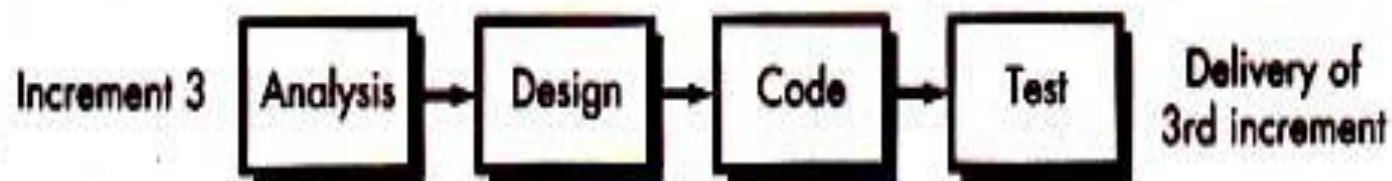
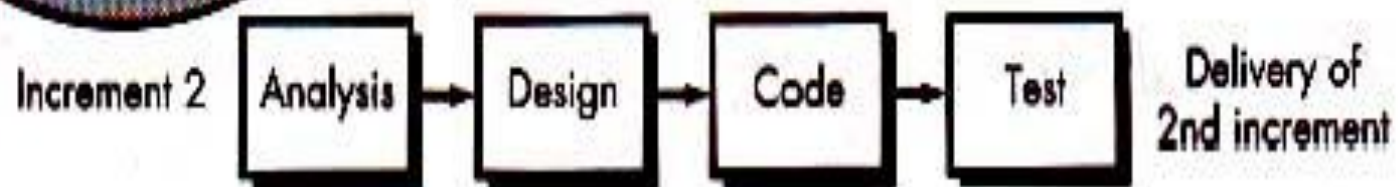
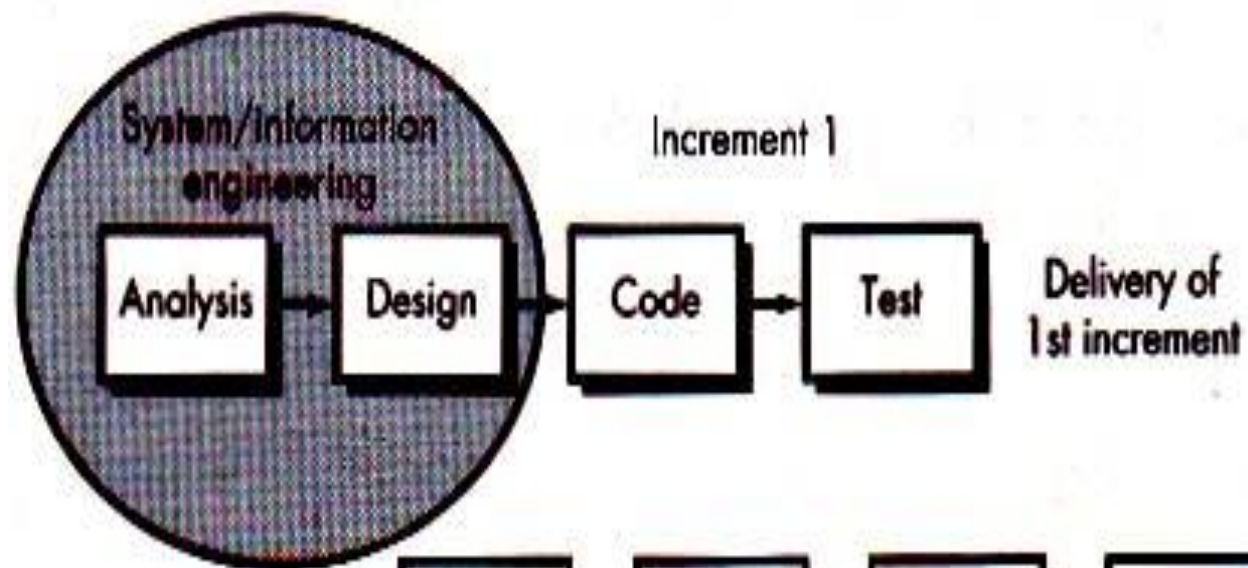


- 2) **Construction of a prototype:** writing the software code depending on the quick design information.
- 3) **Evaluation:** the prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is turned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

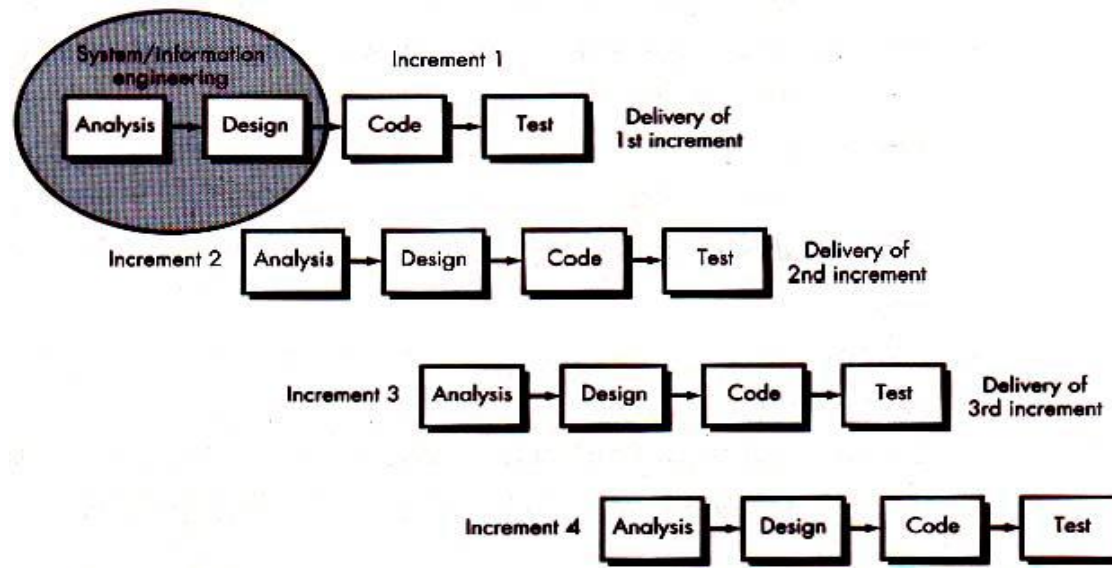
The Incremental Model

The incremental model combines elements of the (waterfall model with the iterative philosophy of prototyping). Each linear sequence produces “increment” of the software. For example, word-processing software developed using the incremental paradigm might:

- 1) Deliver basic file management, editing, and document production functions in the first increment.
- 2) More sophisticated editing and document production capabilities in the second increment.
- 3) Spelling and grammar checking in the third increment.
- 4) Advanced page layout capability in the fourth increment.

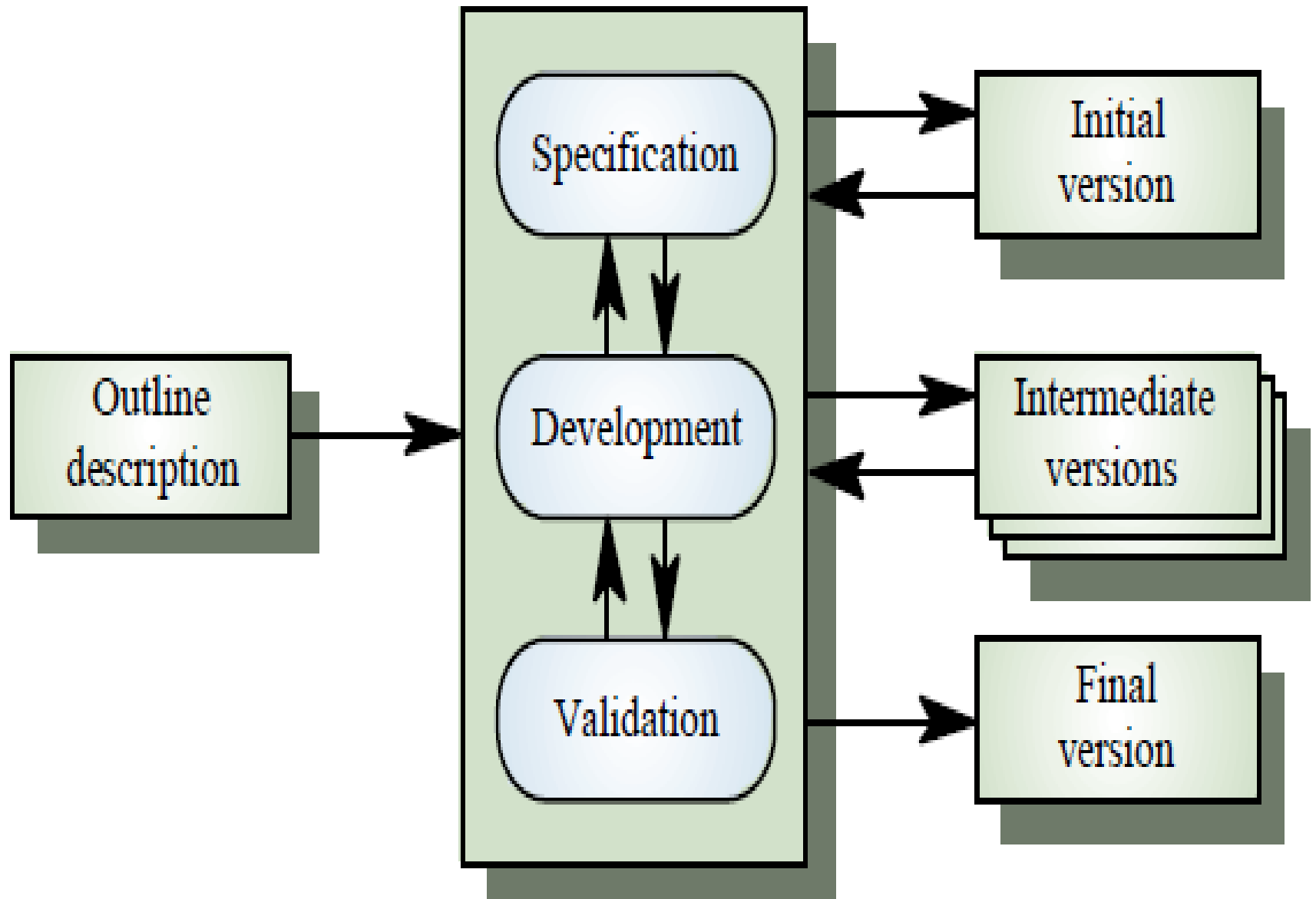


The first increment is often a *core product*. That is, basic requirements are addressed, but many supplementary features remain undelivered. The core product is used by the customer. As a result of use and/or evaluation, a plan is developed for the next increment. But unlike prototyping, the incremental model focuses on the *delivery of an operational product with each increment*.



- Rather than deliver the system as a single delivery, the **development and delivery** is broken down into **increments** with each increment delivering part of the required functionality
- User requirements are prioritised and the highest priority requirements are included in early increments
- Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve

Concurrent
activities



Incremental development advantages

- Customer value can be delivered with each increment so system functionality is available earlier
- Early increments act as a prototype to help elicit requirements for later increments
- Lower risk of overall project failure
- The highest priority system services tend to receive the most testing
- Staffing is unavailable for a complete implementation.
- Increments can be planned to manage technical risks. plan early increments in a way that avoids the use of hardware, thereby enabling partial functionality to be delivered to end-users without inordinate delay.

Problems

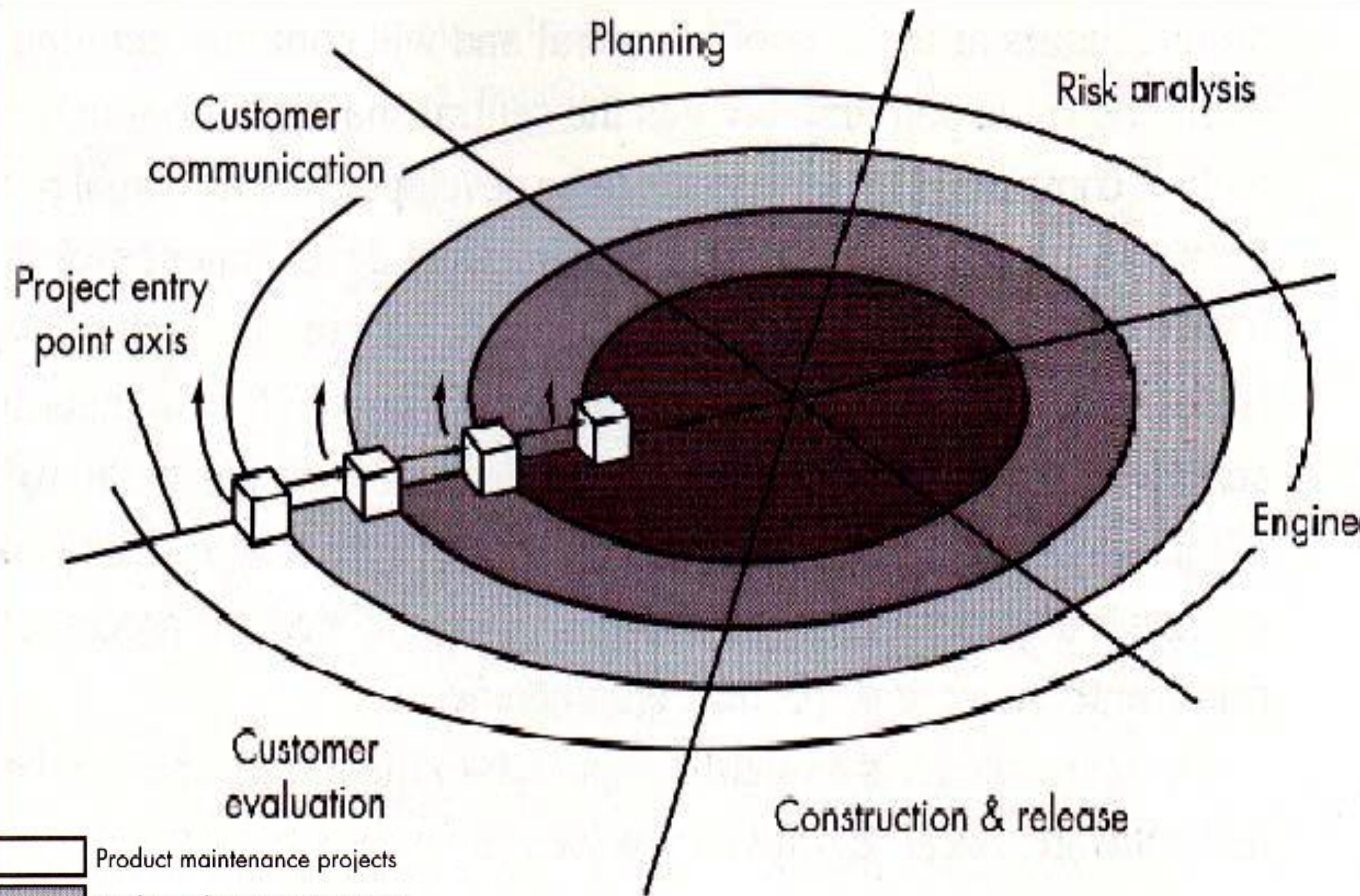
- Lack of process visibility
- Systems are often poorly structured
- Special skills (e.g. in languages for rapid prototyping) may be required

Applicability

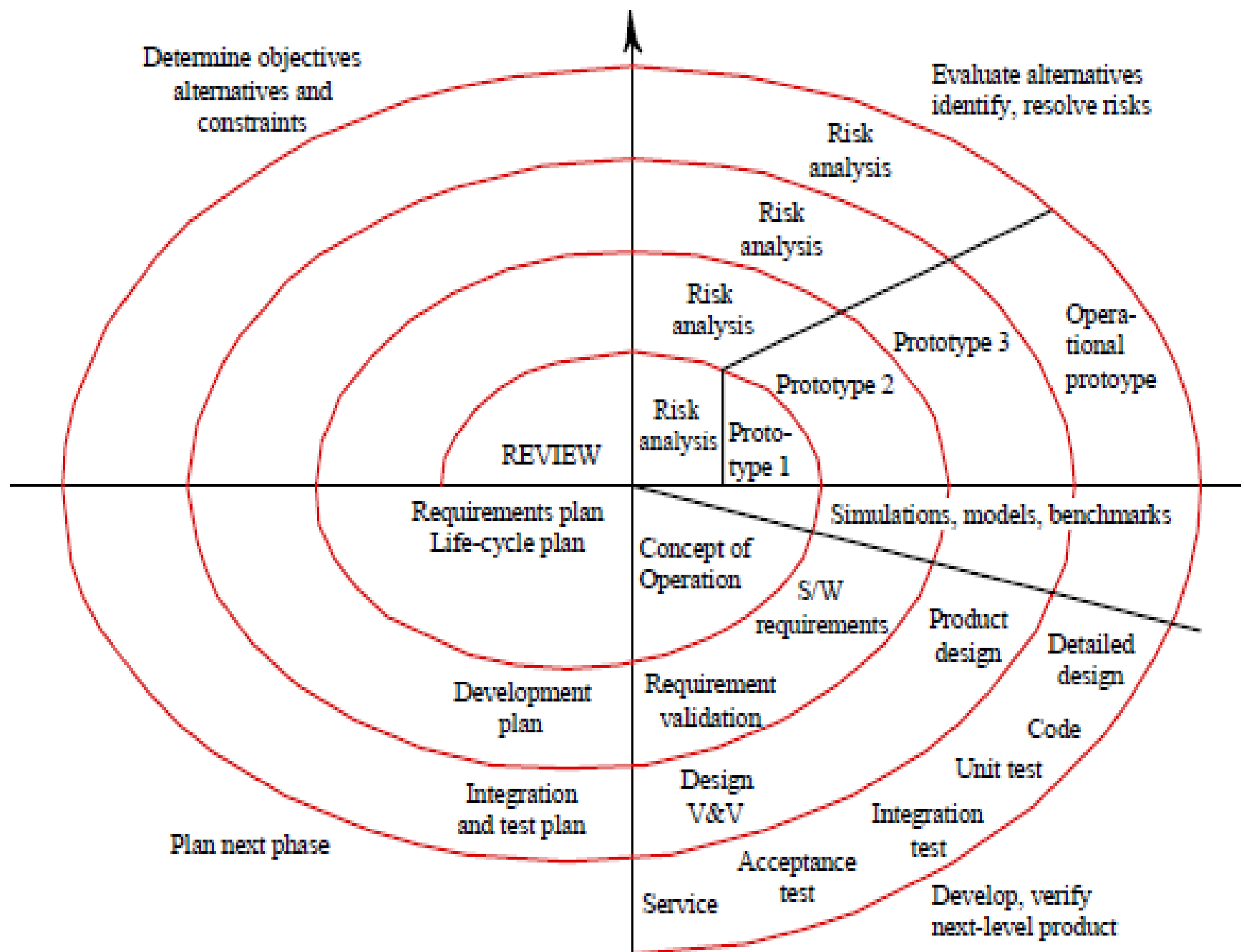
- For small or medium-size interactive systems
- For parts of large systems (e.g. the user interface)
- For short-lifetime systems

The Spiral Model

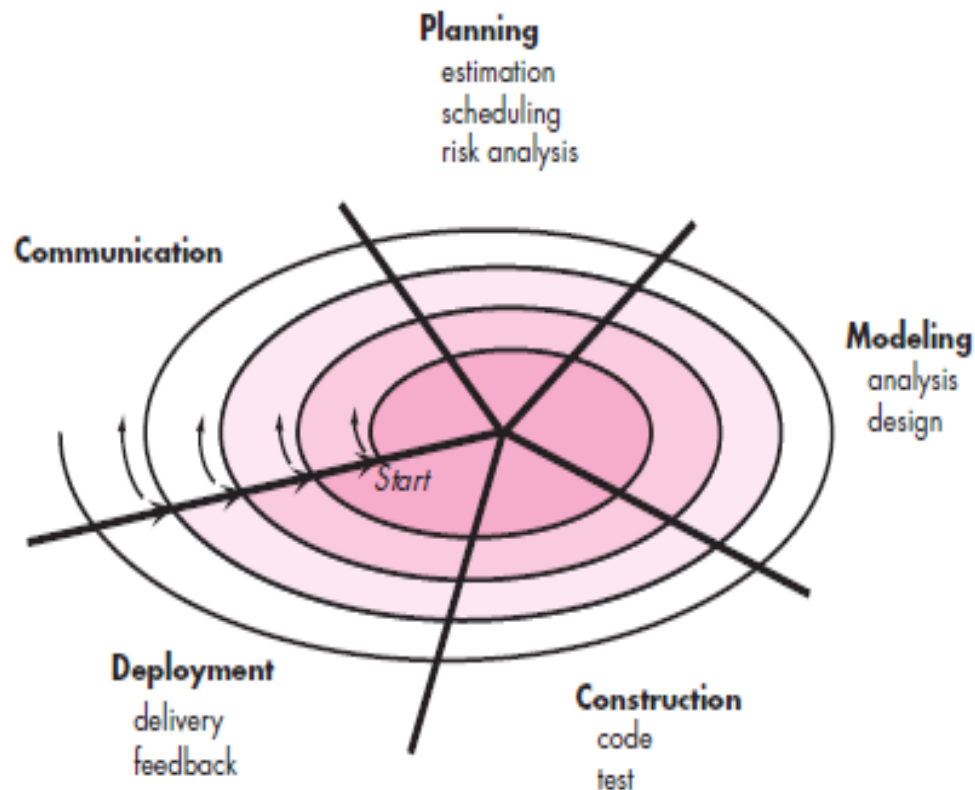
Spiral model, is an evolutionary (تطوري) software process model that **couples** the **iterative nature of prototyping** with the **controlled and systematic aspects of the linear sequential model**. Using the spiral model, software is developed in a series of incremental releases. During **early** iterations, the incremental release might be a **paper model** or prototype. During **later** iterations, increasingly **more complete** versions of the engineered system are produced.



- Product maintenance projects
- Product enhancement projects
- New product development projects
- Concept development projects

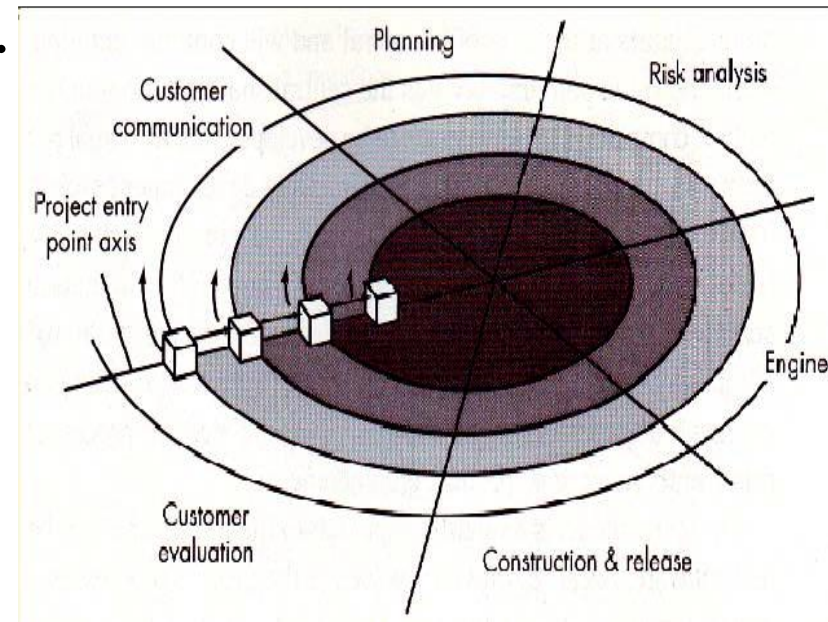
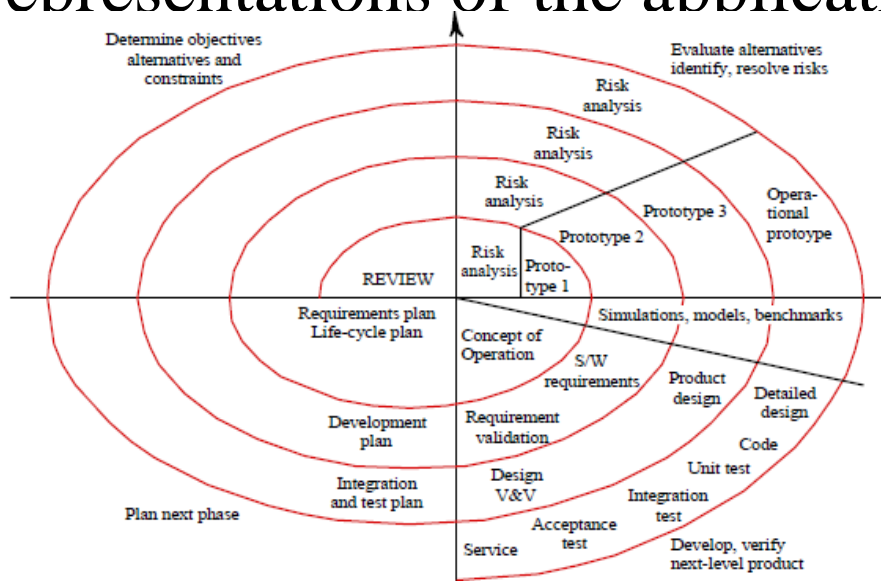


- Process is represented as a spiral rather than as a sequence of activities with backtracking
- Each loop in the spiral represents a phase in the process.
- No fixed phases such as specification or design loops in the spiral are chosen depending on what is required
- Risks are explicitly assessed and resolved throughout the process



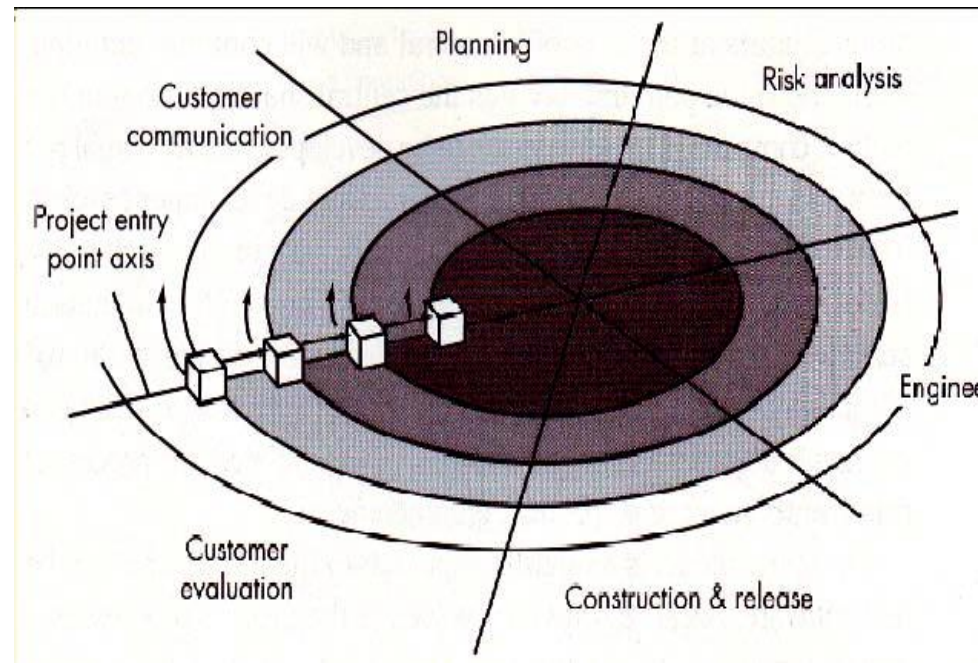
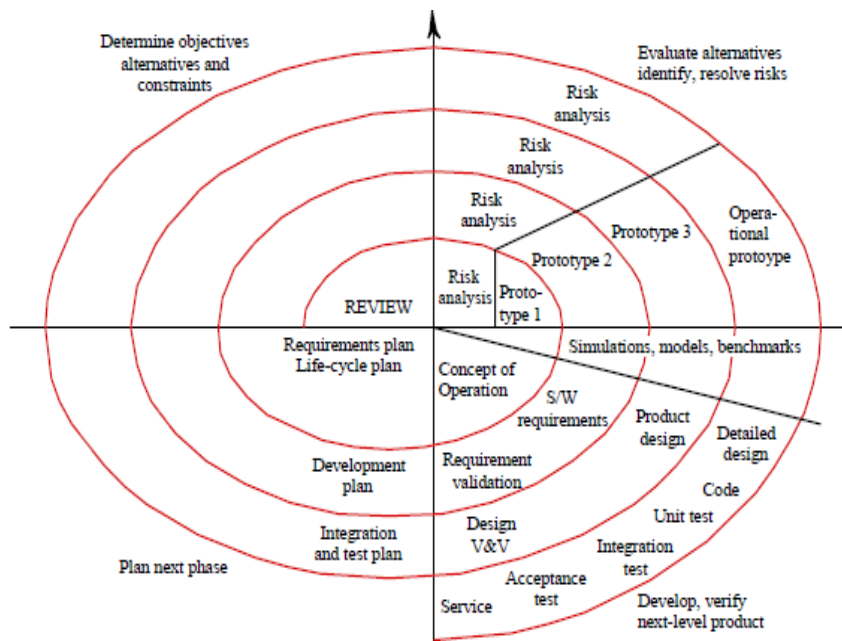
A spiral model is divided into six task regions:

1. **Customer communication-tasks** required to establish effective communication between developer and customer.
2. **Planning-tasks** required defining resources, timelines, and other project-related information.
3. **Risk analysis-tasks** required to assess both technical and management risks.
4. **Engineering-tasks** required building one or more representations of the application.

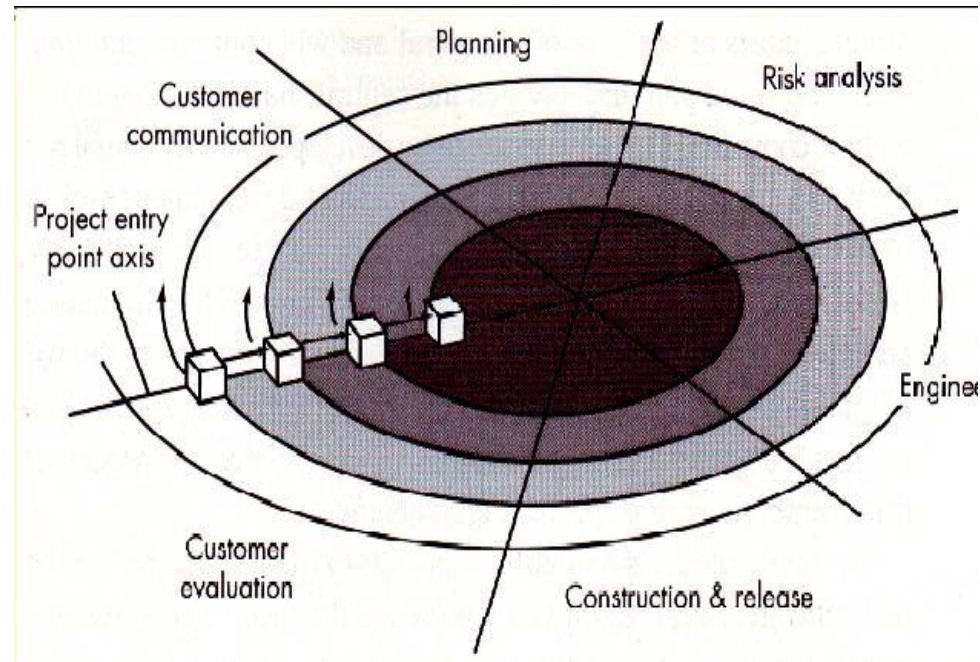
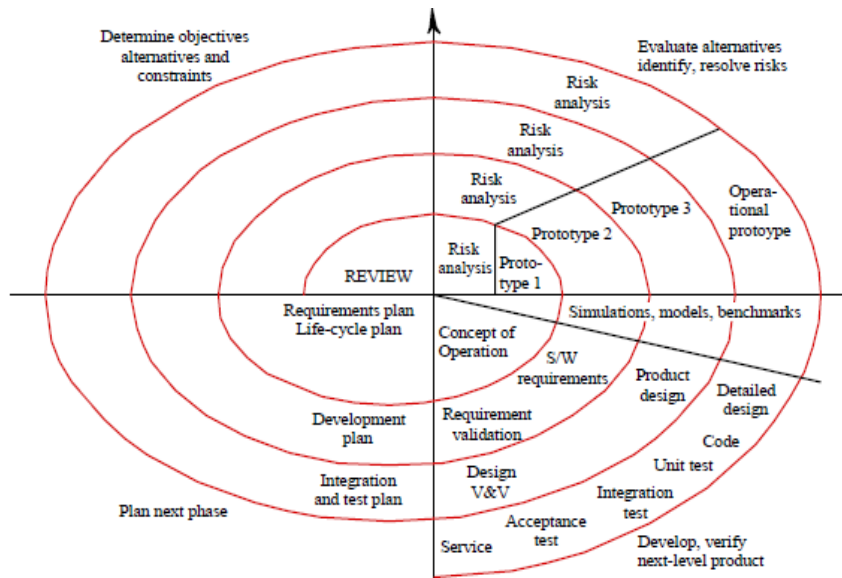


5. Construction and release-tasks required to construct, test, install, and provide user support (e.g., documentation and training).

6. Customer evaluation-tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage moves around the spiral in a clockwise direction, beginning at the center.



The **first circuit** result in the **development of a product specification**; **subsequent passes** around the spiral might be used to **develop a prototype** and then more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from customer evaluation. In addition, the project manager adjusts the planned number of iterations required to complete the software.



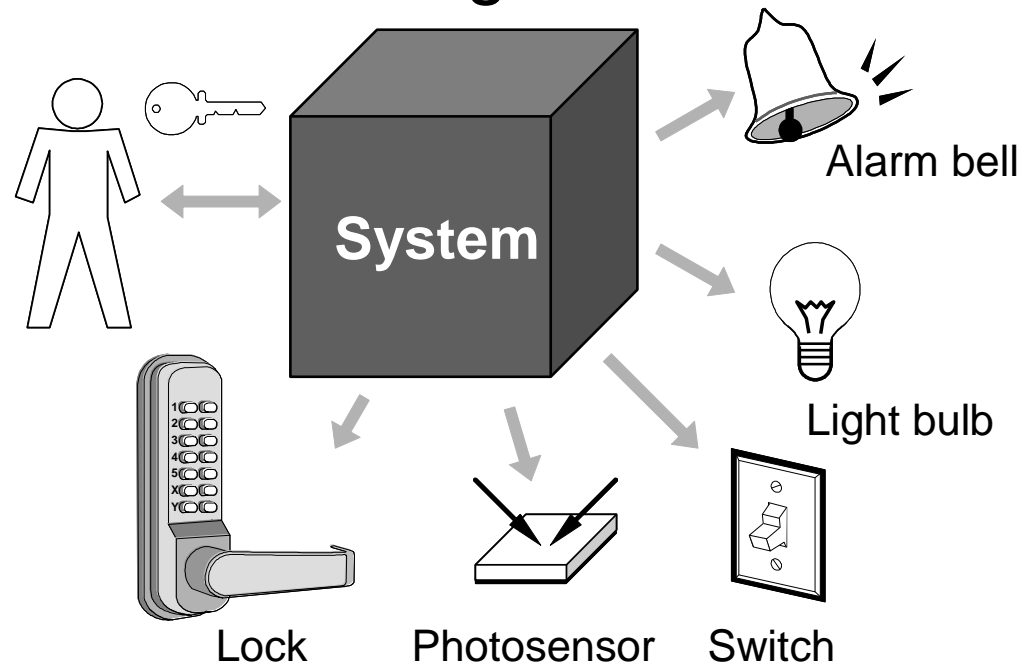
The spiral model has drawbacks for many reasons:

1. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable.
2. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.
3. The model has not been used as widely as the linear sequential or prototyping paradigms. It will take a number of years before efficacy of this important paradigm can be determined with absolute certainty.

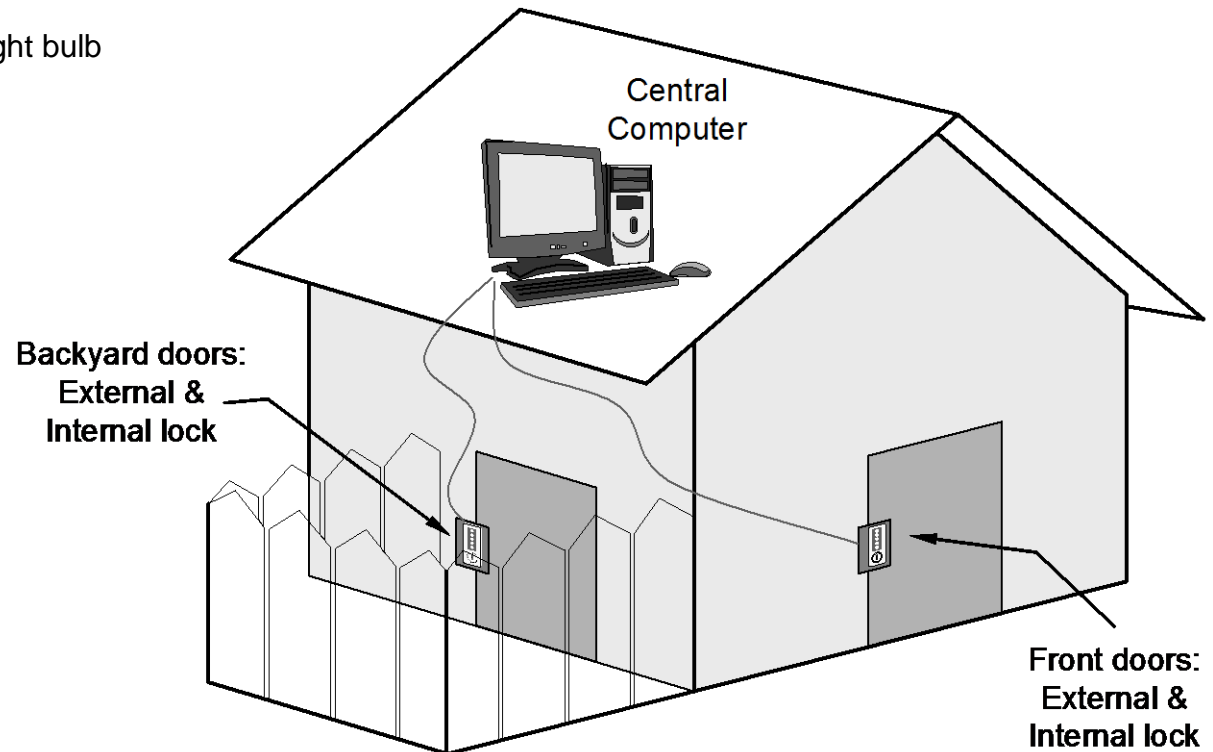
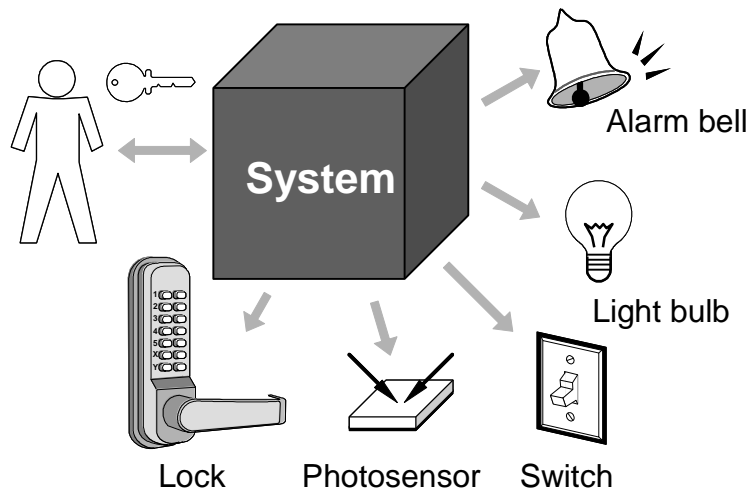
Case Study: Home Access Control

Objective: Design an electronic system for:

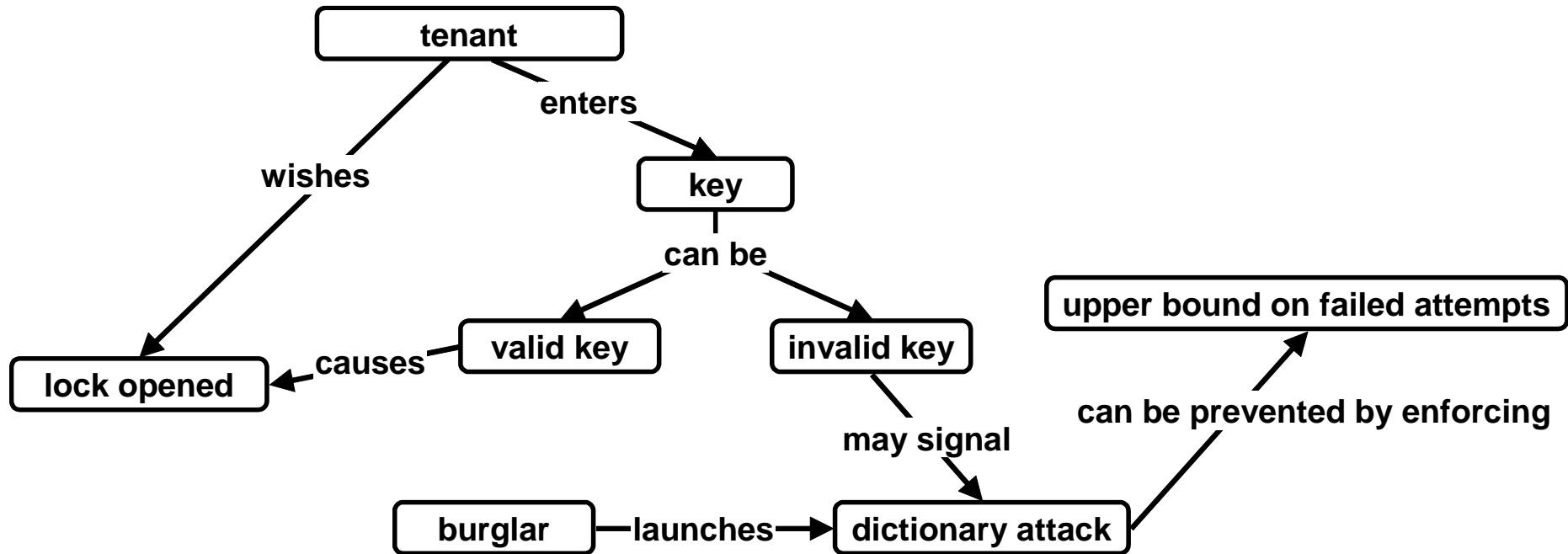
- Home access control
 - Locks and lighting operation
- Intrusion detection and warning



Case Study – More Details



Concept Map for Home Access Control



Chapter Three: Software Requirements

Topics covered

3.1 Requirements Analysis and Definition

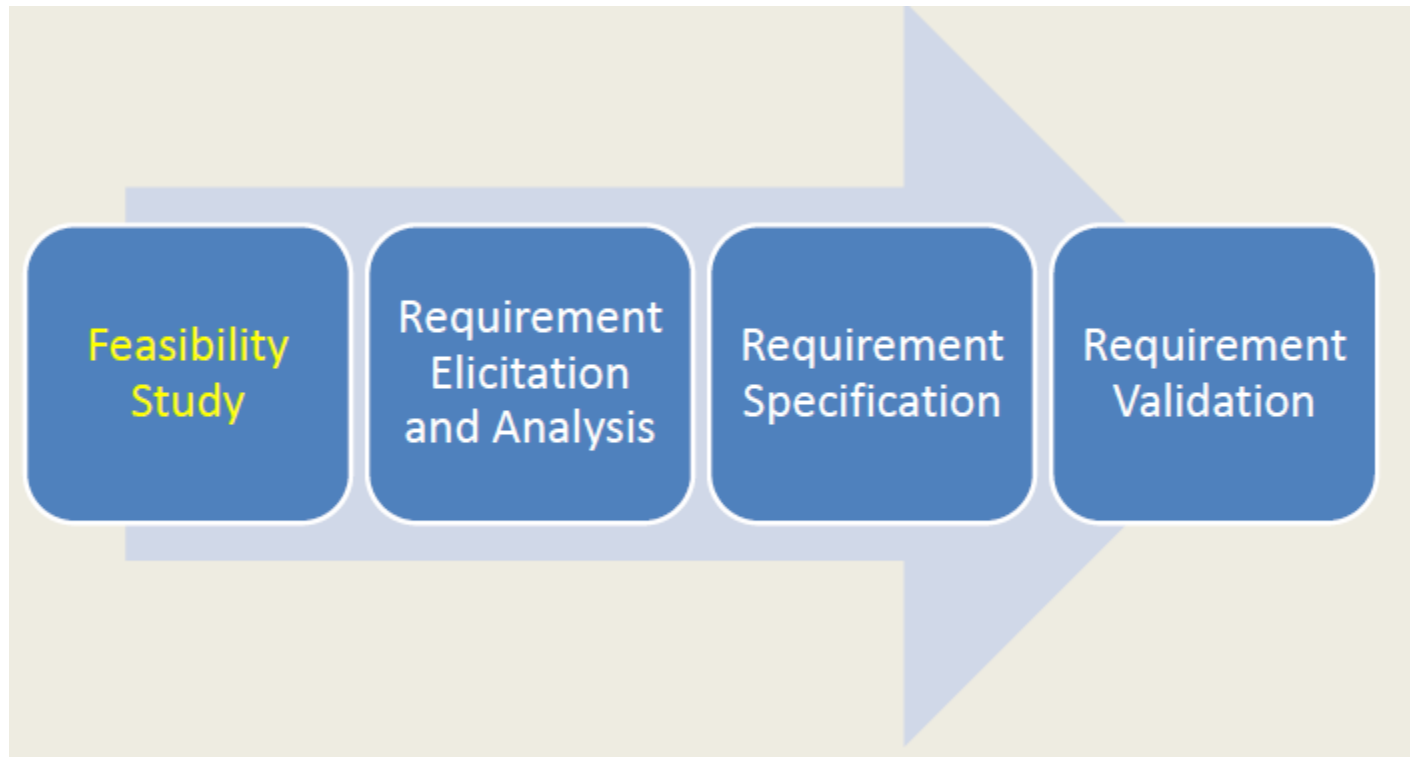
3.2 Requirements Specification

3.3 Software Requirements

3.4 Software Specification

3.5 Software Requirements Document

Requirements Engineering Process



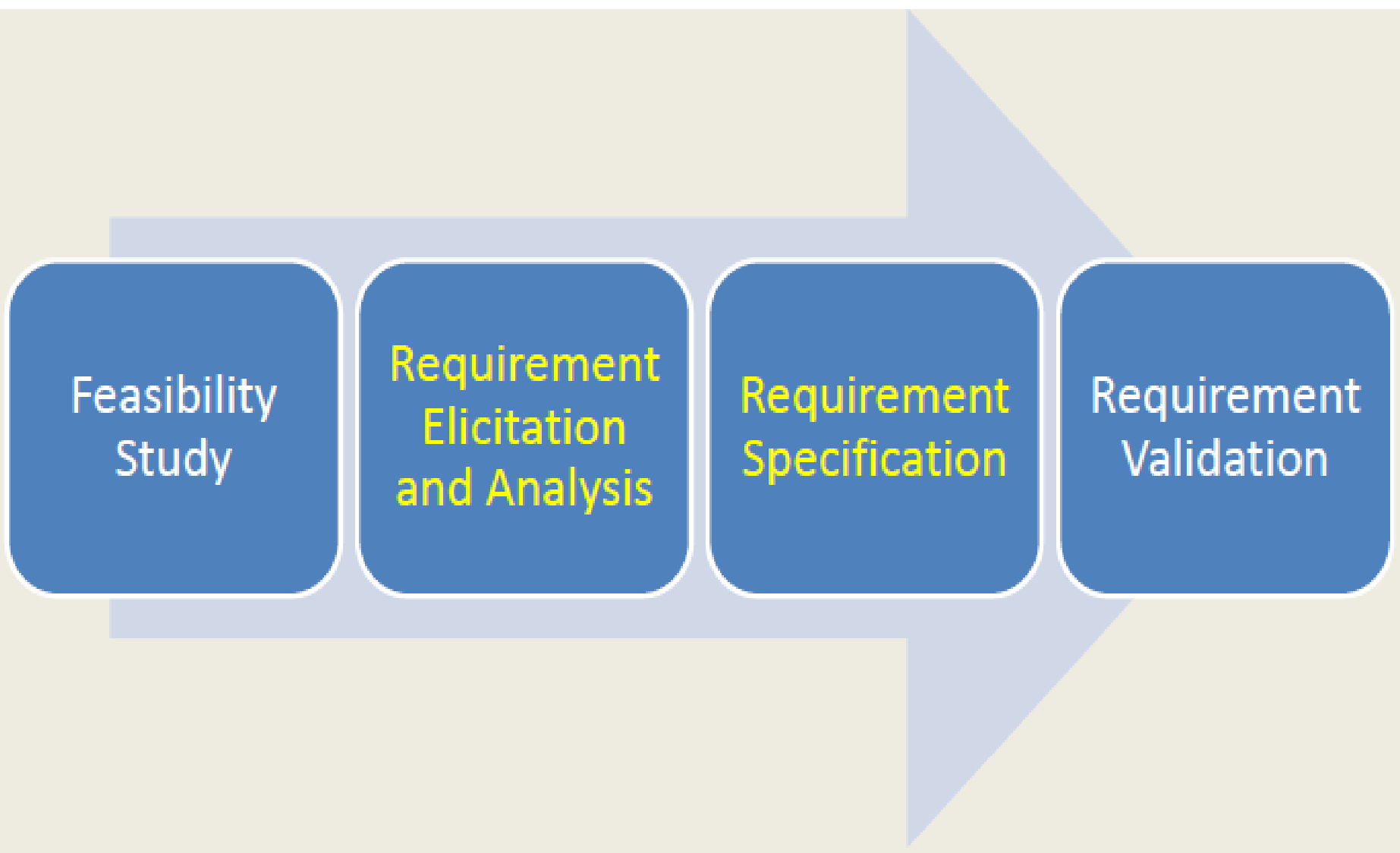
Feasibility studies

- A **feasibility study** decides whether or not the proposed system is worthwhile.
- A short (2-3 weeks) focused study that checks
 - If the system contributes to **organisational objectives**;
 - If the system can be engineered using current **technology** and within **budget**;
 - If the system can be **integrated** with other systems that are used.

Requirements

- Requirement
 - features of system or system function used to fulfill system purpose.
- Focus on **customer's needs** and problem, **not** on **solutions**:
 - Requirements **definition** document
(written for customer).
 - Requirements **specification** document
(written for programmer; technical staff).

Requirements engineering process



The diagram illustrates the Requirements Engineering process as a sequence of four steps. The steps are represented by blue rounded rectangular boxes with white borders, arranged horizontally. The first box contains 'Feasibility Study' in white text. The second box contains 'Requirement Elicitation and Analysis' in yellow text. The third box contains 'Requirement Specification' in yellow text. The fourth box contains 'Requirement Validation' in white text. A large, light blue arrow points from left to right, passing behind the boxes. The background is a light beige color with a large, light blue arrow pointing from left to right, which serves as a backdrop for the process steps.

Feasibility
Study

Requirement
Elicitation
and Analysis

Requirement
Specification

Requirement
Validation

Elicitation and analysis

- Involves **technical staff working with customers** to find out about:
 - the application **domain**,
 - the **services** that the system should provide
 - and the system's operational **constraints**.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*.

Example: ATM stakeholders

- Bank customers
- Representatives of other banks
- Bank managers
- Counter staff
- Database administrators
- Security managers
- Marketing department
- Hardware and software maintenance engineers
- Banking regulators

Problems of requirements elicitation



Stakeholders have troubles expressing requirements

Requirements Analysis and Definition

Software requirements analysis is necessary to avoid creating software product that fails to meet the customer's needs.

Software requirements are description of features and functionalities of the target system. Requirements convey the expectations of users from the software product. The requirements can be obvious or hidden, known or unknown, expected or unexpected from client's point of view.

Writing Requirements Definitions

Requirements definitions usually consist of *natural language*, supplemented by (e.g., UML: Unified Modeling Language) *diagrams and tables*.

Three types of problems can arise:

- **Lack of clarity:** It is hard to write documents that are both *precise and easy-to-read*.
- **Requirements confusion:** *Functional and non-functional requirements* tend to be intertwined.
- **Requirements amalgamation:** Several *different requirements* may be expressed together.

User requirements

- Statements in natural language **plus** diagrams of the services that the system provides **and** its operational constraints.
- Should describe functional and non-functional requirements so that they are understandable by system users who don't have detailed technical knowledge.
- Written for customers
- **Definition**

System requirements

A structured document setting out detailed descriptions of the system services.

- Written as a contract between client and contractor
- Written for developers
- **Specification**

System Req.: Functional and Non-functional Requirements

Functional requirements describe system *services or functions*

- Compute sales tax on a purchase
- Update the database on the server ...

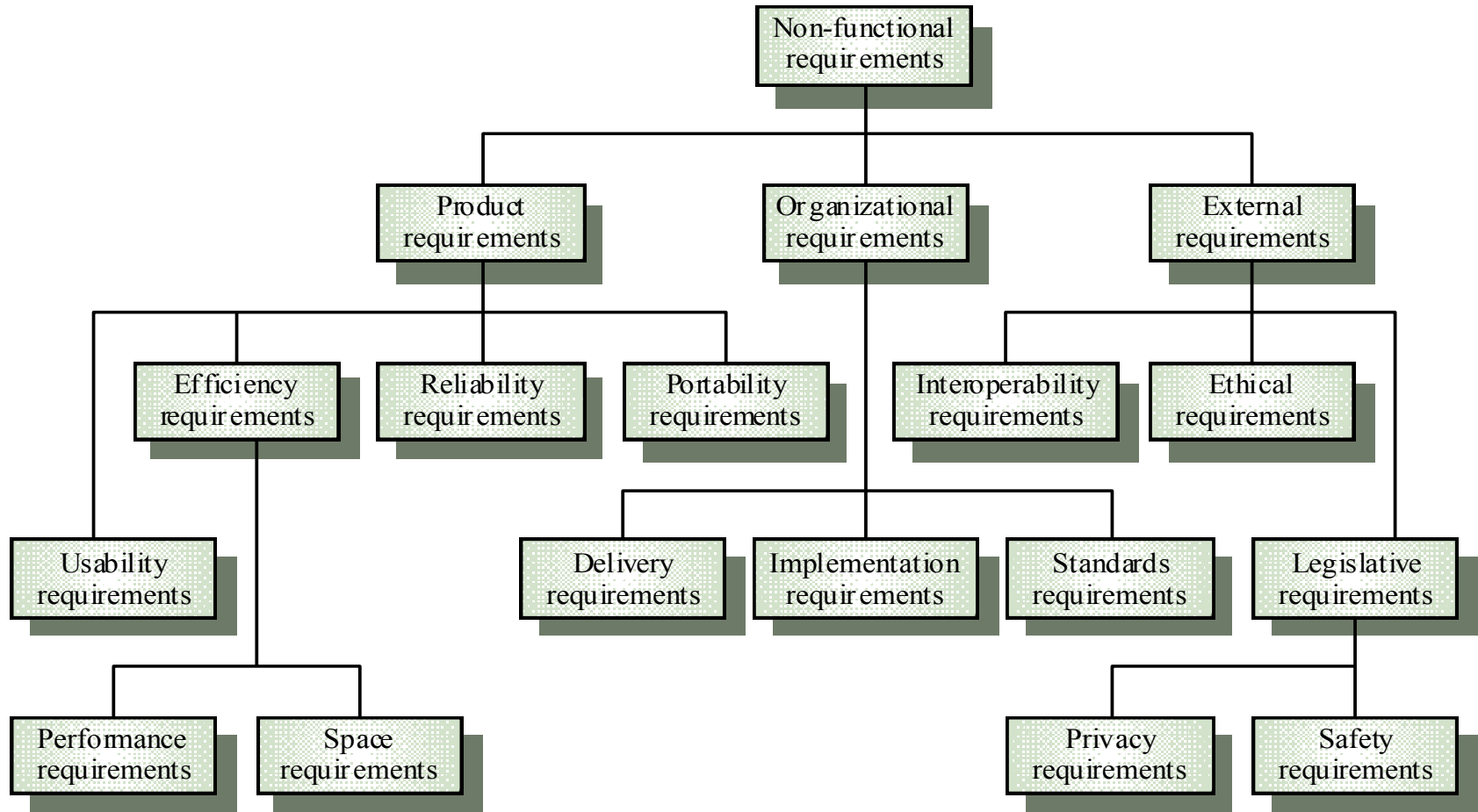
Non-functional requirements are *constraints* on the system or the development process

- Physical environment (equipment locations, multiple sites, etc.).
- Interfaces (data medium etc.).
- User & human factors (who are the users, their skill level etc.).

Non-functional requirements may be more critical than functional requirements.

If these are not met, the system is useless!

Types of Non-functional Requirements



1. Requirements Analysis

- SE bridges the gap between system level requirements engineering and software design.
- Provides software designer with a representation of system information, function, and behavior that can be translated to data, architectural, components level design.
- Expect to do a little bit of design during analysis and a little bit of analysis during design.

2. Software Requirements Analysis

- Identify the "customer" and work Together to negotiate "product –level"
- Build an analysis model
 - Focus on data
 - define function
 - represent behavior
- Prototype areas of uncertainty
- Develop a specification that will guide design
- Conduct formal technical reviews.

3. Software Requirements Analysis Phases

- Problem recognition
- Evaluation and synthesis (focus is on what not how)
- Modeling
- Specification
- Review

4. Analysis Principles

- The information domain of the problem must be represented and understood.
- The functions that the software is to perform must be defined.
- Software behavior must be represented.
- Models depicting information, function, and behavior must be partitioned in a hierarchical manner.
- The analysis process should move from the essential information toward implementation details.

Software Requirement Specification (SRS)

SRS is a document created by system analyst after the requirements are collected from various stakeholders.

SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.

SRS should come up with following features:

- **User Requirements** are expressed in natural language.
- **Technical requirements** are expressed in structured language, which is used inside the organization.
- **Design description** should be written in Pseudo code.
- **Format of Forms** and GUI screen prints.

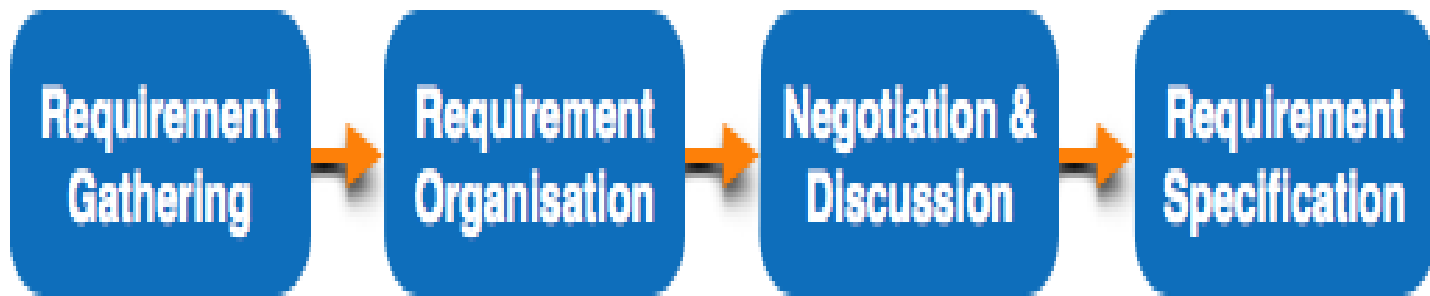
Requirement Elicitation Process

Requirements gathering

Organizing Requirements - The developers prioritize and arrange the requirements in order of importance, urgency and convenience.

Negotiation & discussion - If requirements are ambiguous or there are some conflicts, then negotiated and discussed with stakeholders (end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc.).

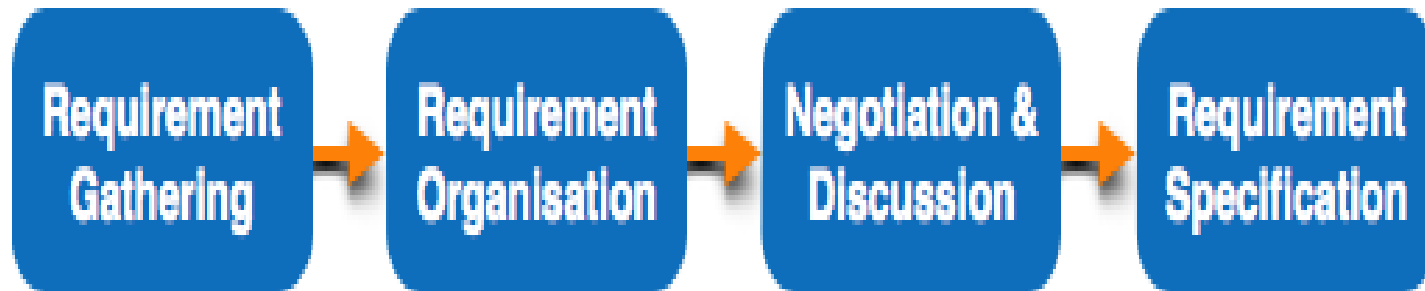
Documentation - All formal & informal, functional and non-functional requirements are documented for next phase processing.



This may take several iterations:

Business req. → User req. → System req.

(Each phase itself may also use several iterations)



Software specification

- A detailed software description which can serve as a basis for a design or implementation.
- Written for developers

The Requirements Document

- Official statement of what is required of the system developers.
- Should include both a definition and a specification of requirements
- Should:
 - specify external system behavior
 - specify implementation constraints
 - be easy to change (but changes must be managed)
 - serve as a reference tool for maintenance
 - record forethought about the life cycle of the system (i.e. predict changes)
 - characterize responses to unexpected events
- It is not a design document
 - it should state what the system should do rather than how it should do it

Requirements document structure

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

Users of Requirements documents

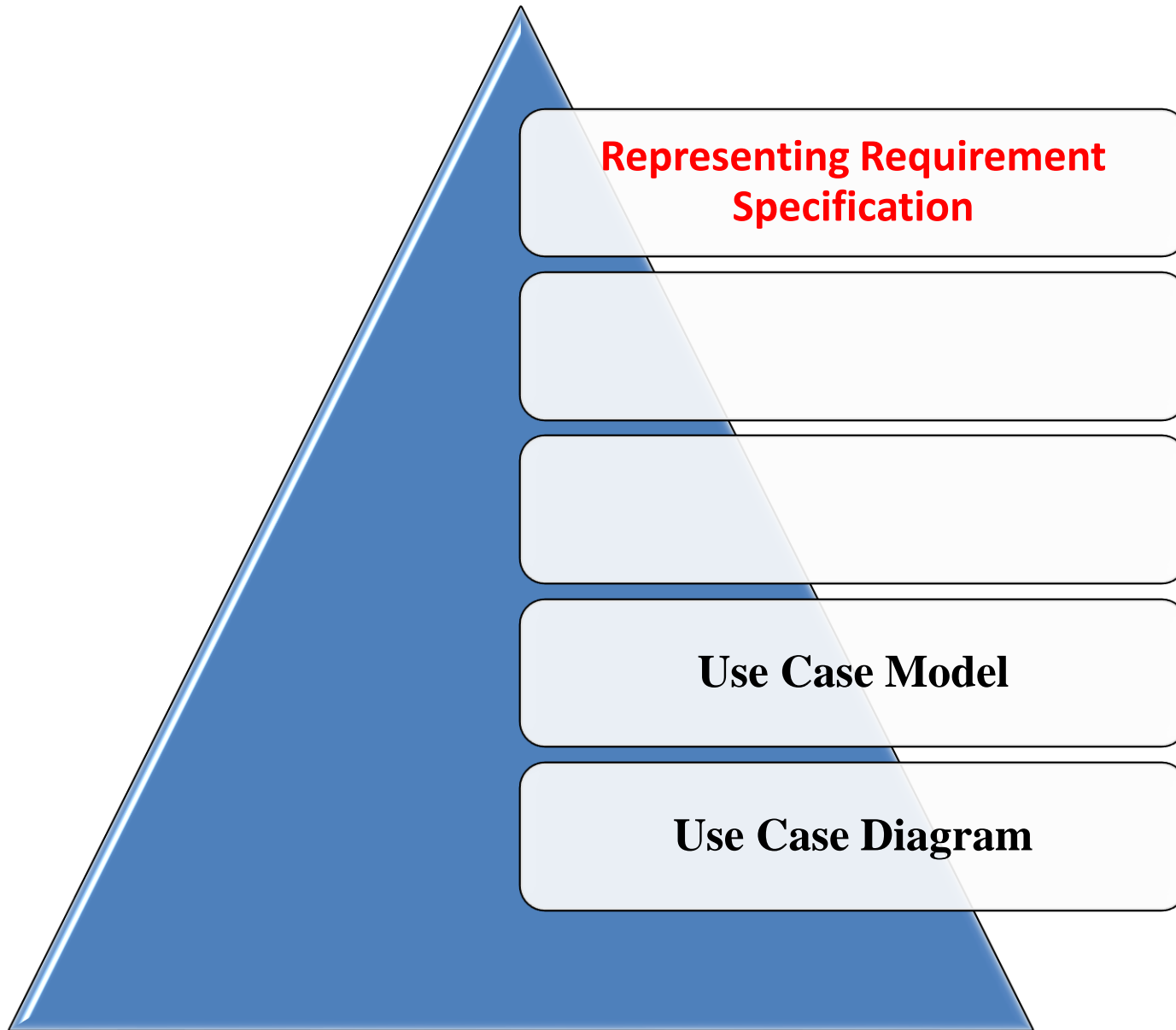
- **System customers:** Read them back to check that they meet their **needs**; specify **changes** to the requirements
- **Managers:** Use the requirements document to **plan** a bid for the system and to plan the system development process
- **System Engineers:** Use the requirements to **understand** what system is to be developed
- **Test Engineers:** Use the requirements to **develop** validation tests for the system
- **Maintenance Engineers:** Use the requirements to help **understand** the system and the relationships between its parts

Requirement Elicitation Techniques

- 1. Interviews**
- 2. Surveys**
- 3. Questionnaires**
- 4. Task analysis**
- 5. Domain Analysis**
- 6. Brainstorming**
- 7. Prototyping**
- 8. Observation**

Software Requirements Characteristics

- Clear
- Correct
- Consistent
- Coherent
- Comprehensible
- Modifiable
- Verifiable
- Prioritized
- Unambiguous
- Traceable
- Credible source



Use Cases and Scenarios

A *use case* is the *specification* of a *sequence of actions*, including *variants*, that a system (or other entity) can perform, *interacting with actors* of the system”.

- e.g., buy a DVD through the internet

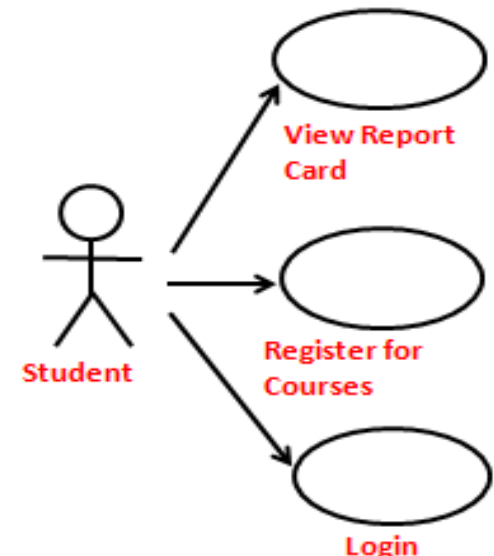
A *scenario* is a *particular trace of action occurrences*, starting from a known initial state.

- e.g., connect to myDVD.com, go to the “search” page

...

Use case model

- Use-cases are a **scenario** based technique in the **UML (Unified Modeling Language)**
 - identify the **actors** in an interaction
 - describe the **interaction** itself
- A set of use cases should describe all possible interactions with the system.
- **use case = a named collection of scenarios**

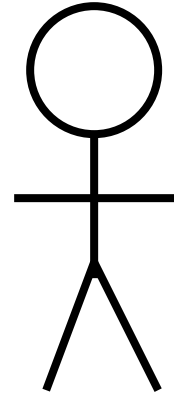


What Are the Benefits of a Use-Case Model?

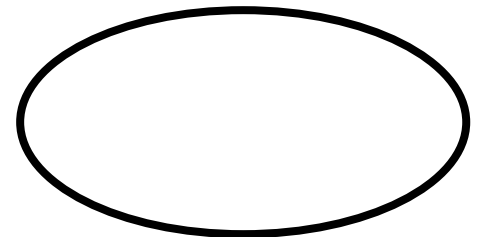
- Used to communicate with the end users and domain experts
 - Provides buy-in at an early stage of system development
 - Insures a mutual understanding of the requirements
- Used to identify
 - Who interacts with the system and what the system should do
 - The interfaces the system should have
- Used to verify
 - All requirements have been captured
 - The development team understands the requirements

Major Concepts in Use-Case Modeling

- An **actor** represents anything that interacts with the system.
- A **use case** is a sequence of actions a system performs that yields an observable result of value to a particular actor.



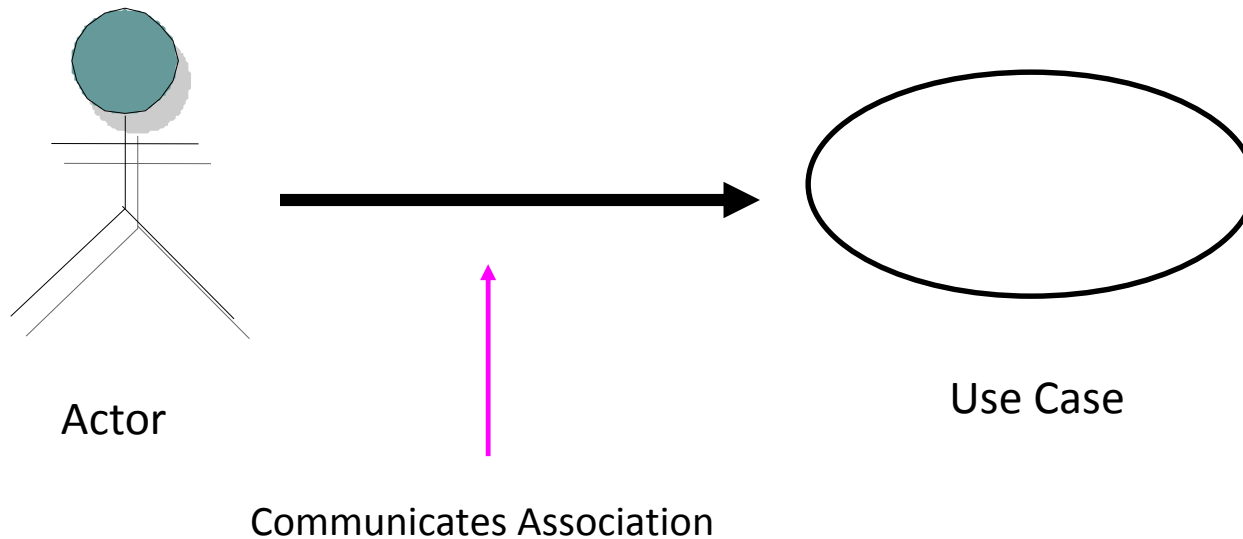
Actor



Use Case

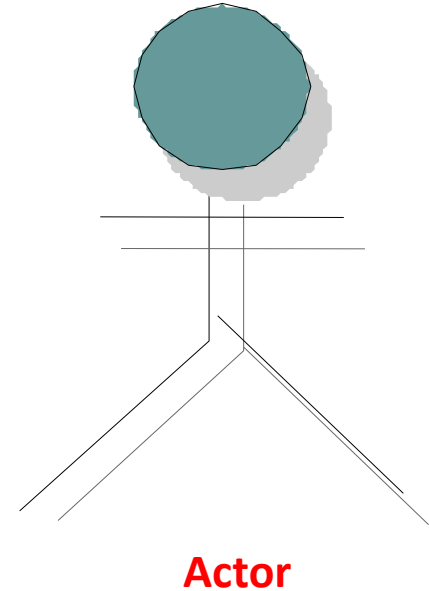
Use Cases and Actors

- A **use case models** a dialog between actors and the system.
- A **use case** is initiated by an actor to invoke a certain functionality in the system.



What Is an Actor?

- ❑ **Actors** are not part of the system.
- ❑ **Actors** represent roles a user of the system can play.
- ❑ **They** can represent a human, a machine, or another system.
- ❑ **They** can actively interchange information with the system.
- ❑ **They** can be a giver of information.
- ❑ **They** can be a passive recipient of information.
- ❑ An **actor** represents a **role** that a human, hardware device, or another system can play.



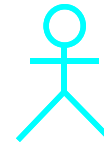
*Actors are **EXTERNAL**.*

A User May Have Different Roles



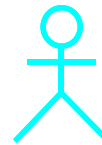
Charlie

Charlie
as student



Student

Charlie as
professor

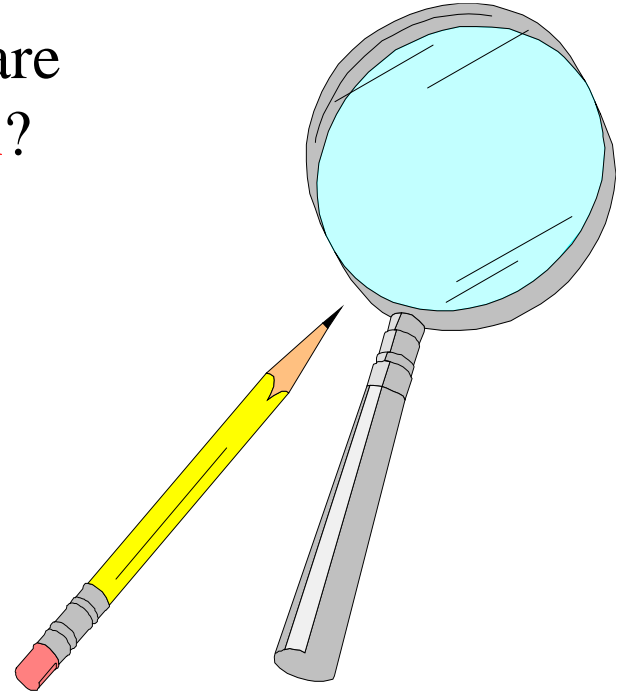


Professor

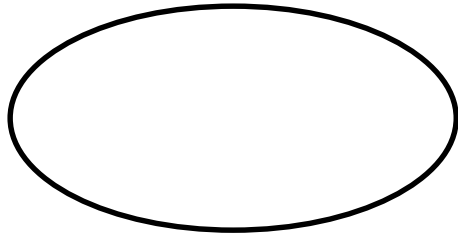
How to Find Use Cases

Answer the following questions to find use cases.

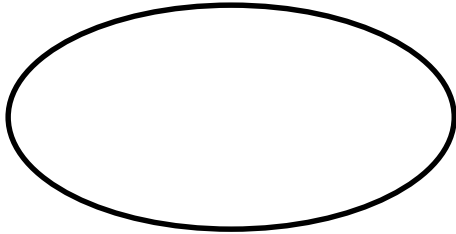
- For each actor you have identified, what are the tasks the **system** would be **involved in**?
- Does the actor need to be informed about certain occurrences in the system?
- Will the actor need to inform the system about sudden, external changes?
- What information must be modified or created in the system?



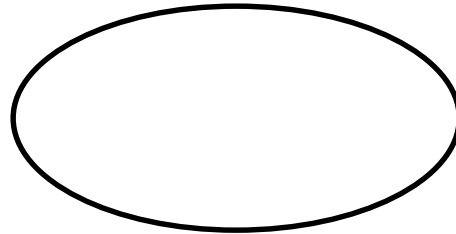
Naming the Use Case



Register for
Courses



Login



Maintain Student
Information

- The name indicates what is achieved by its interactions with the actor(s).
- The name may be several words in length.
- No two use cases should have the same name.

Use case 1: Register for courses Description:

This use case is **initiated by the student**. It provides the capability to **create, review, modify, and delete** a course schedule for a specialized semester. All required billing information is sent to the **Billing System**.

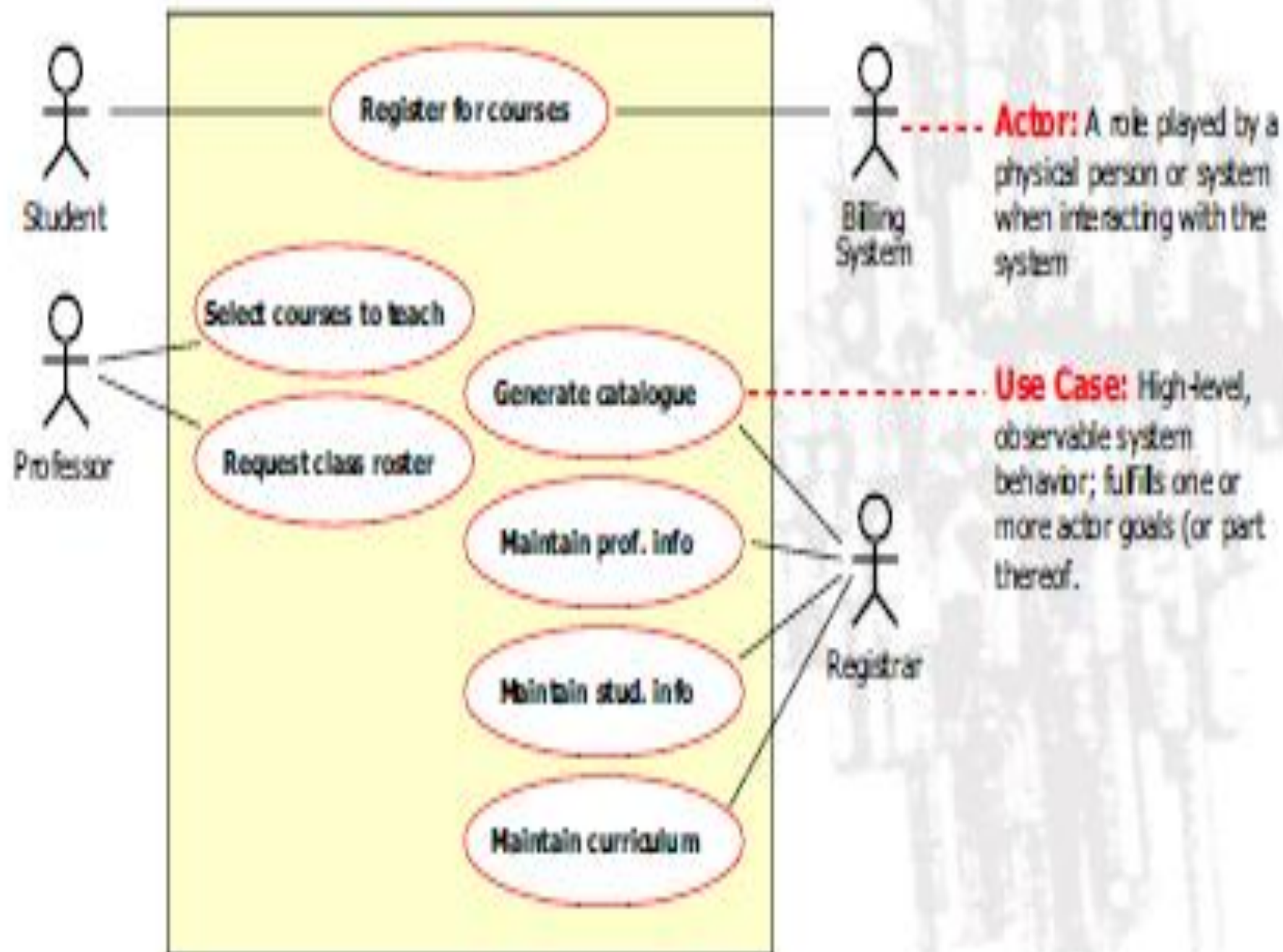
Actors: Student, Billing System.

Notes:

A student can register for at most 4 courses each semester.

Main success scenario:

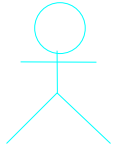
1. The student identifies himself/herself.
2. The system verifies student identity.
3. The student selects a valid semester.
4. The student creates, reviews, or changes a schedule.
5. The systems prints a notification.
6. The system sends billing information to the Billing System



Practice: Find the Actors

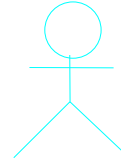
- In the Course Registration System Requirements document, read the Problem Statement for the Course Registration case study.
- As a group, identify the following
 - Actors
 - Description of the actor

Practice: Solution



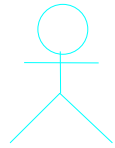
Billing System

The external system responsible for student billing



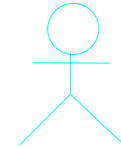
Student

A person who is registered to take courses at the University



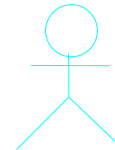
Professor

A person who is teaching classes at the University



Course Catalog

The unabridged catalog of all courses offered by the University



Registrar

The person who is responsible for the maintenance of the course registration system

Requirements Elicitation Using Use Cases

- Identifying Scenarios
- Identifying Use Cases
- Identifying Actors
- Refining Use Cases
- Identifying Relationships between Actors and Use Cases
- Identifying Nonfunctional Requirements

2- Identifying Use Cases

- Use Case
 - Specifies **all possible scenarios** for a given functionality
 - Initiated by an actor
- Motivations for use cases
 - Generalizing related scenarios help developers define the scope of the system
 - The role of each user of the system is clarified
- Use Case Descriptions
 - Entry and exit conditions
 - Flow of events
 - Quality requirements

3- Identifying Actors

- Actors
 - person** or **machine** using the system in a particular role.
- Actors usually correspond to existing roles within the client organization
- Guide Questions
 - Which user groups execute the system's **main functions**?
 - Which user groups perform **secondary functions**, such as maintenance and administration?
 - Which user groups are **supported** by the system to perform their work?
 - With what external hardware or software system will the system **interact**?

Actors vs. Stakeholders

- **Actors**

- interact with the system
- They might be humans or other systems

- **Stakeholders**

- have some interest in the system
- They include the users and many others, e.g., person who invests in a system to improve the business processes but never uses the system

Formulating Use Cases

Step 1: Name the use case

—Use case name: *ReportEmergency*

Step 2: Find the actors

—Generalize the concrete names (“Bob”) to participating actors (“Field officer”)

—Participating Actors:

- *Field Officer (Bob and Alice in the Scenario)*
- *Dispatcher(John in the Scenario)*

Step 3: Concentrate on the flow of events

—Use informal natural language

—Number them to form a flow

Step 4: Describe entry and exit conditions

Step 5: Describe exceptions

Step 6: List non-functional requirements related to the use case

Use Case Example: ReportEmergency

- ❑ **Use case name:** ReportEmergency
- ❑ **Participating Actors:**
 - Field Officer(Bob and Alice in the Scenario)
 - Dispatcher(John in the Scenario)
- ❑ **Exceptions:**
 - FRIEND notifies the FieldOfficerimmediately if the connection between her terminal and the central is lost.
 - FRIEND notifies the Dispatcher immediately if the connection between any logged in FieldOfficerand the central is lost.
- ❑ **Flow of Events:** on next slide.
- ❑ **Special Requirements (Non-functional):**
 - The system acknowledges the FieldOfficer'sreport within 30 seconds.
 - The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.

ReportEmergency:: Flow of Events

1. The FieldOfficer activates the “Report Emergency” function of her terminal.
2. FRIEND responds by presenting a form to the officer.
3. The FieldOfficer fills the form, by selecting the emergency level, type, location, and a brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. The FieldOfficer submits the form after finishing it.
4. FRIEND receives the form and notifies the Dispatcher.
5. The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncidentuse case.
6. The Dispatcher selects a response and acknowledges the emergency report.
7. FRIEND displays the acknowledgment and the selected response to the FieldOfficer.

<i>Use case name</i>	ReportEmergency
<i>Participating actors</i>	Initiated by FieldOfficer Communicates with Dispatcher
<i>Flow of events</i>	<ol style="list-style-type: none"> 1. The FieldOfficer activates the “Report Emergency” function of her terminal. 2. FRIEND responds by presenting a form to the FieldOfficer. 3. The FieldOfficer fills out the form by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form. 4. FRIEND receives the form and notifies the Dispatcher. 5. The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the report. 6. FRIEND displays the acknowledgment and the selected response to the FieldOfficer.
<i>Entry condition</i>	<ul style="list-style-type: none"> • The FieldOfficer is logged into FRIEND.
<i>Exit condition</i>	<ul style="list-style-type: none"> • The FieldOfficer has received an acknowledgment and the selected response from the Dispatcher, OR • The FieldOfficer has received an explanation indicating why the transaction could not be processed.
<i>Quality requirements</i>	<ul style="list-style-type: none"> • The FieldOfficer’s report is acknowledged within 30 seconds. • The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.

Use Case Diagram

□ Use case:

—A written description of one use of the system.

- Who will use the system?
- What will they be able to do with it?
- No standard for format and content.

□ Use case *diagram*

—A graphical representation of a use case.

—A diagram type defined in UML.

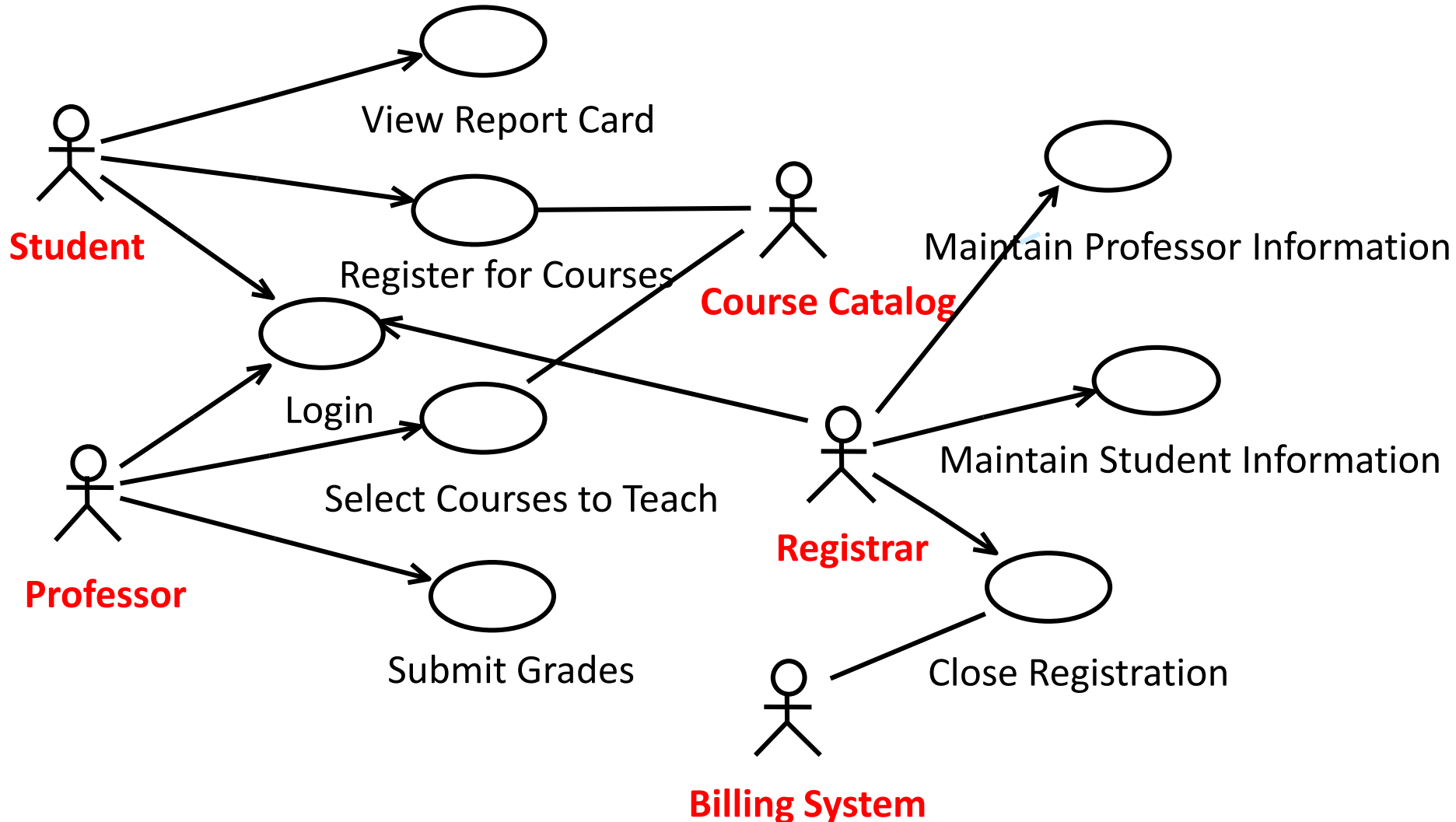
—Gives the developers and users a high-level view of the relationships among the different Use Cases and Actors of a system.

□ UML: Unified Modeling Language

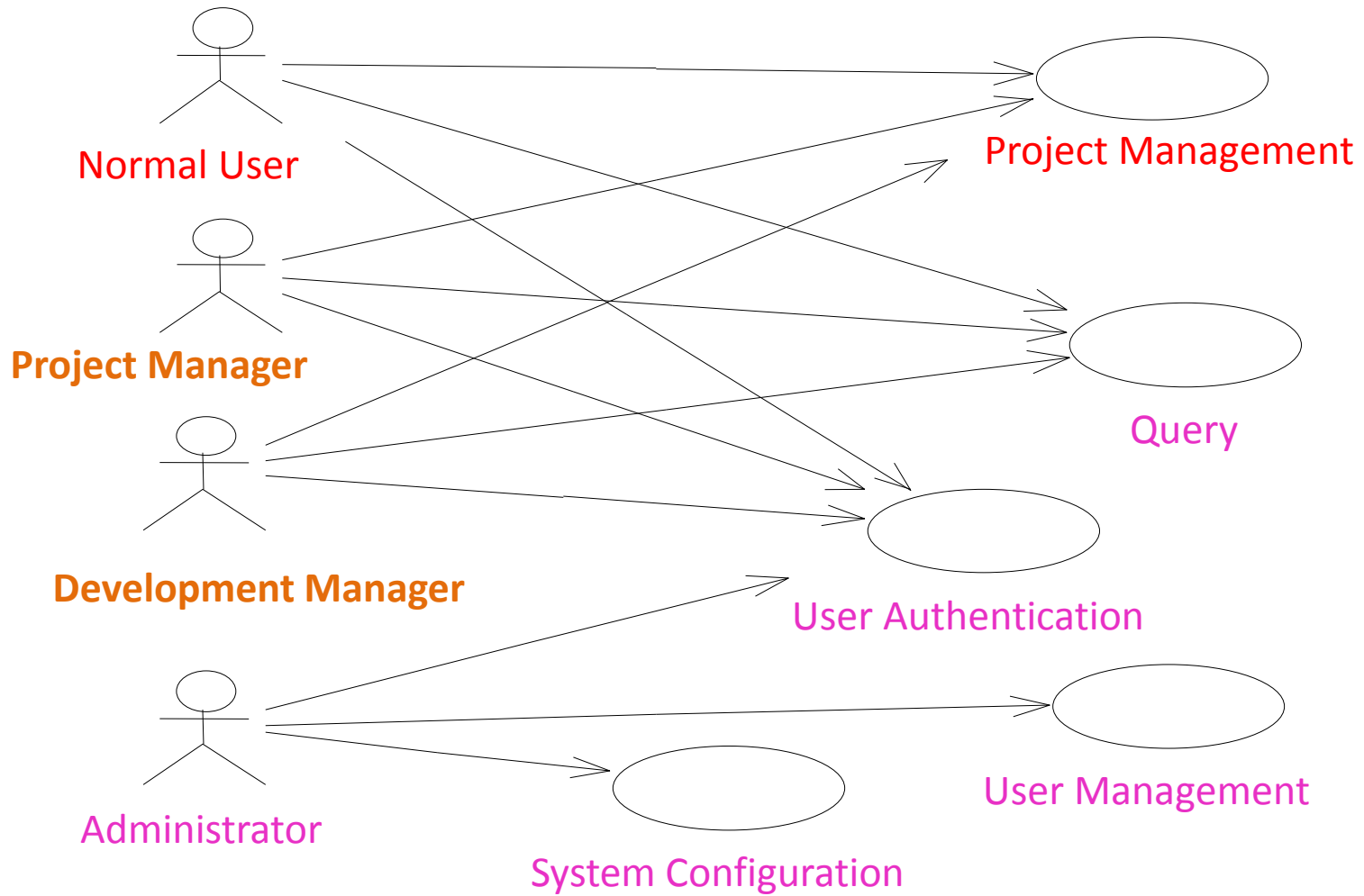
Use case diagram

- **Captures system functionality as seen by users**
- **Built in early stages of development**
- **Purpose**
 - Specify the context of a system
 - Capture the requirements of a system
 - Validate a system's architecture
 - Drive implementation and generate test cases
- **Developed by analysts and domain experts**

How Would You Read This Diagram?



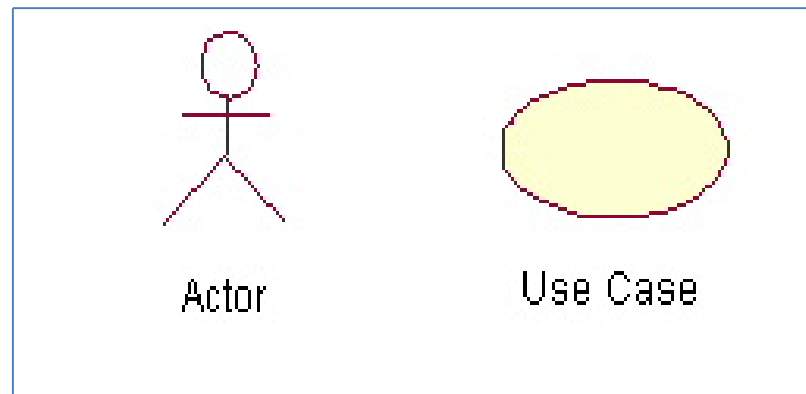
Use Case Diagram - Example



Use Case Diagram

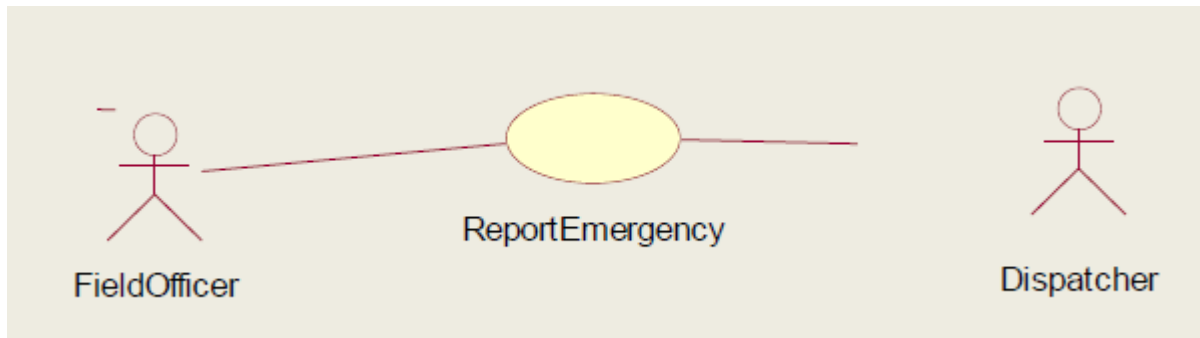
Four types of components:

- Actors: stick figure
- Use case: oval
- the system: a boundary box
- relationships: lines



Draw a Use Case Diagram

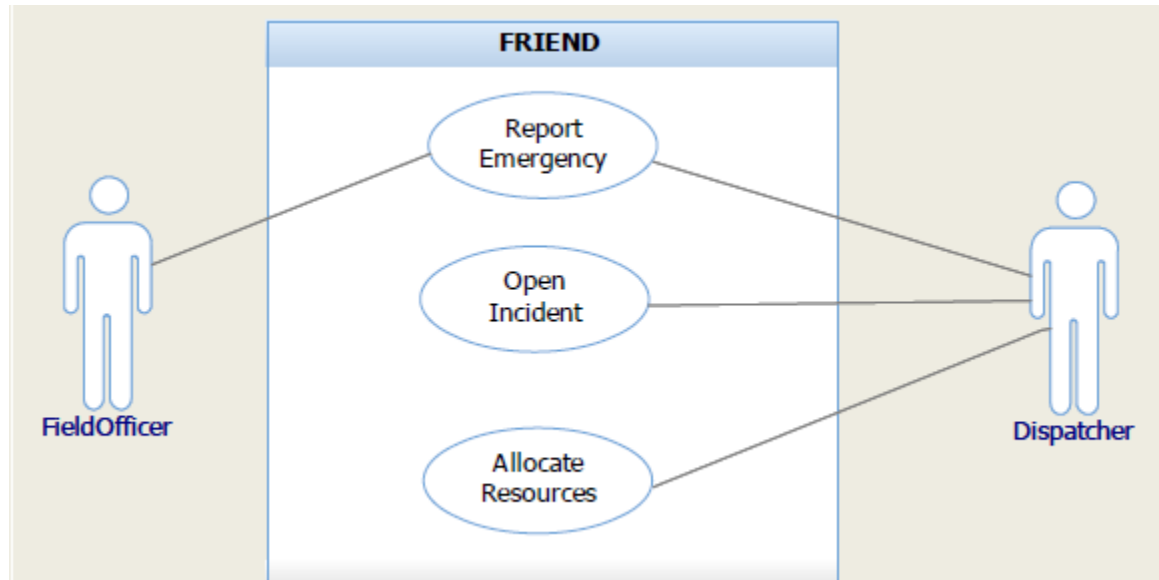
For a use case: all participants (actors) are associated with it.



Use non-directed lines to connect Actors and Use Cases.

Draw a Use Case Diagram

For a system: list all use cases and their actors



- Put all use cases inside the system boundary box, all actors outside the box. (Optional)
- Put primary actors (initiators) on the left, secondary actors (participants) on the right.
- Each use case should have a text description (as described previously).

<<Include>> and <<Extend>>

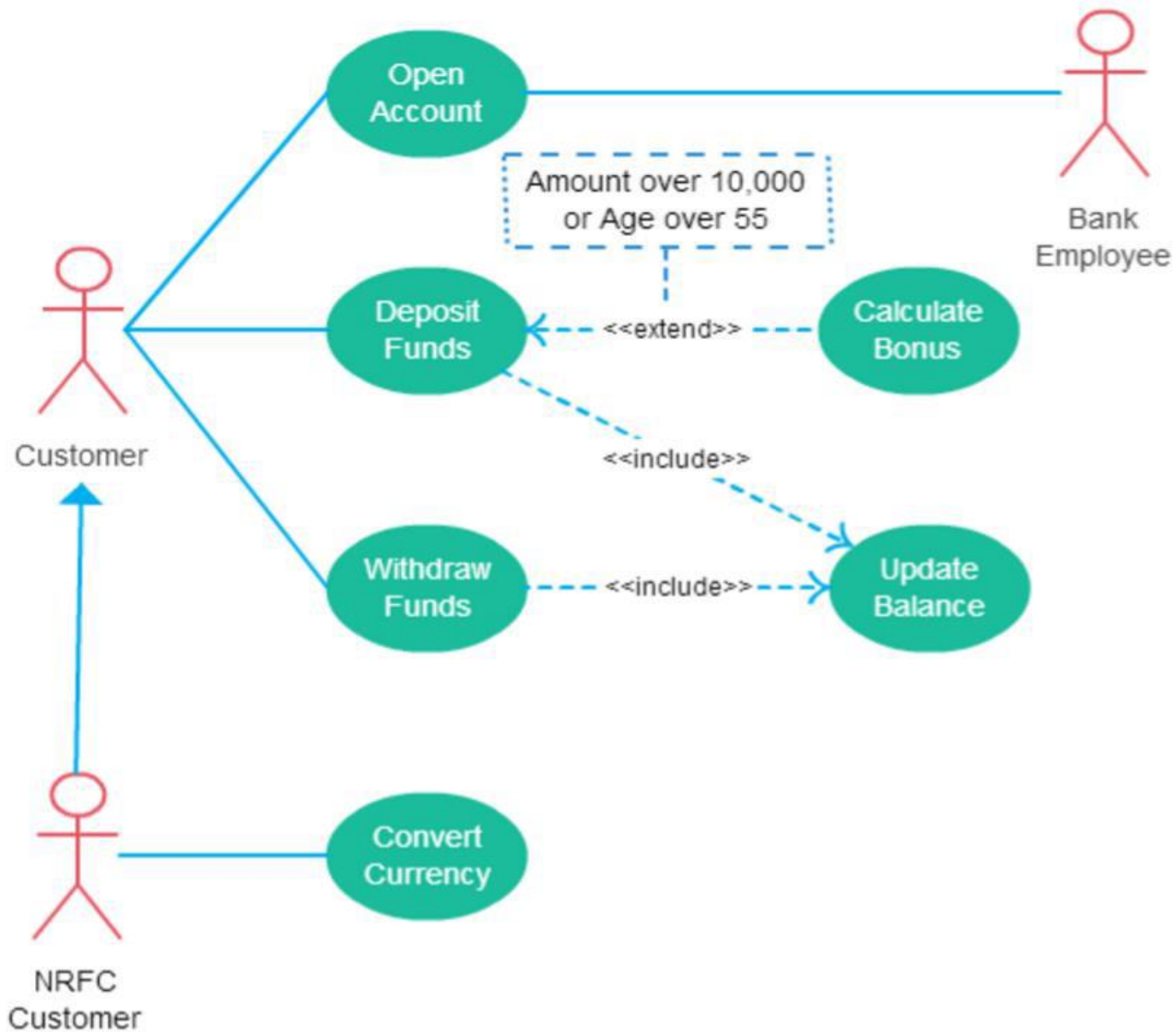
□ <<include>>:

- If multiple use cases share the same partial flow of events, make the partial flow a separate use case and include it in these multiple use cases.
- OR the inclusion use case is an important part of the base use case.
 - The base case is incomplete without the inclusion case.

□ <<extend>>:

- The extension use case consists of additional behavior that can incrementally augment the behavior of the base use case.
 - The extension use case is not meaningful on its own.

Example



Use Case Diagrams

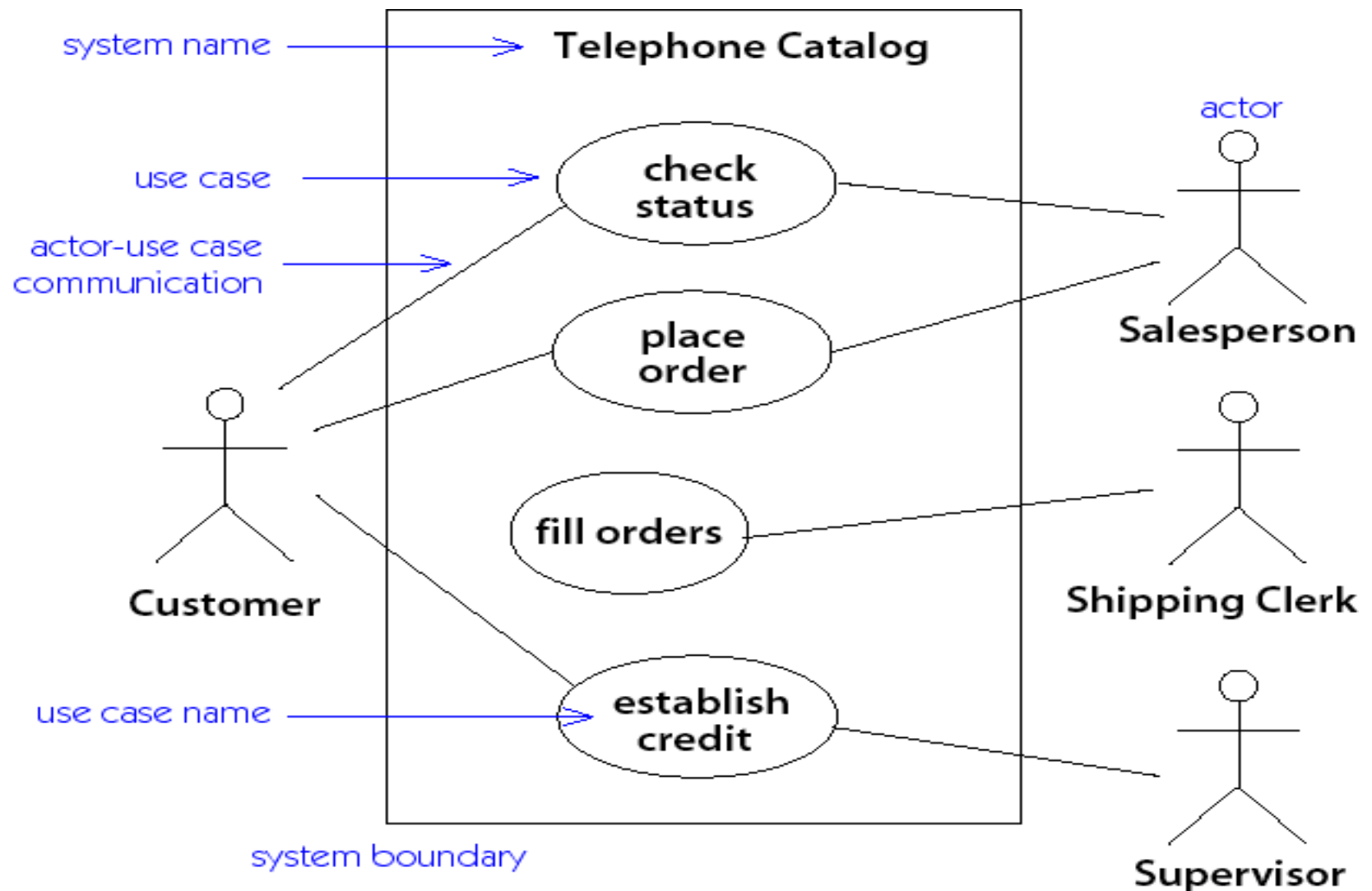


Figure 5-1. *Use case diagram*

User Profile - Example

- Full Control (Administrator)
- Read/Write/Modify All (Manager)
- Read/Write/Modify Own (Inspector)
- Read Only (General Public)

Use Case Example - 1

- **Read Only Users**
 - The read-only users will only read the database and cannot insert, delete or modify any records.
- **Read/Write/Modify Own Users**
 - This level of users will be able to insert new inspection details, facility information and generate letters. They will be also able to modify the entries they made in the past.

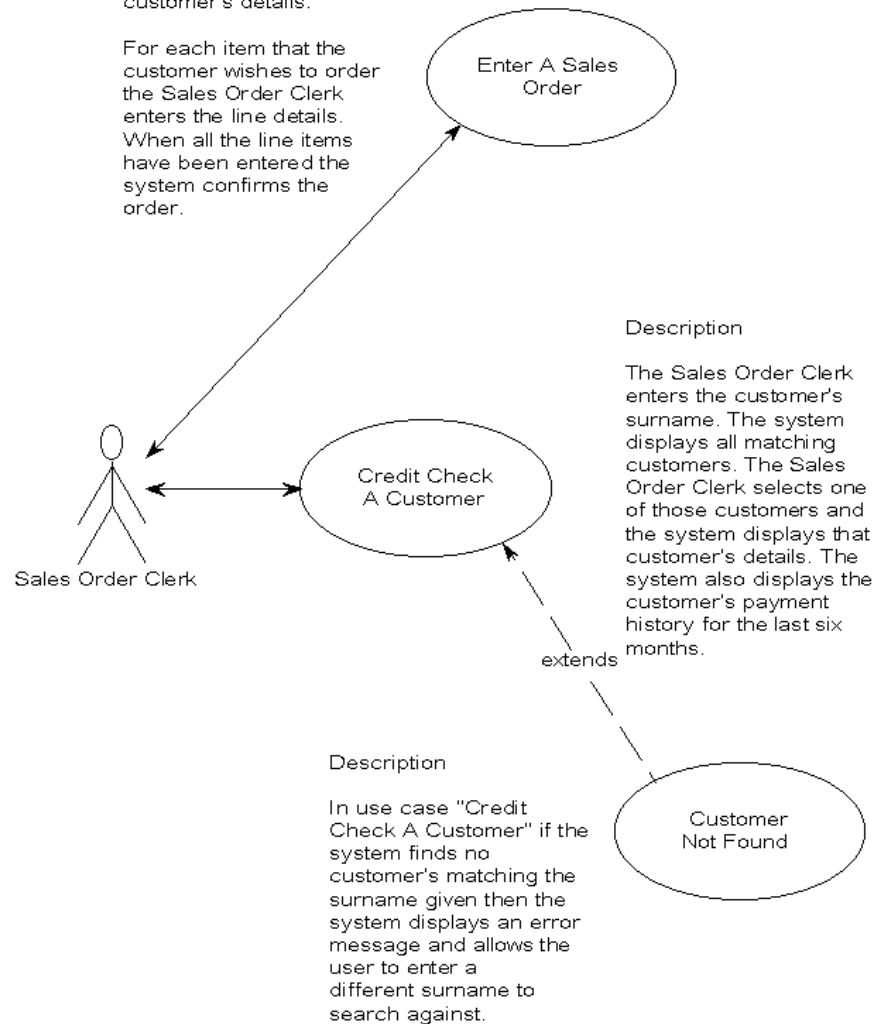
Use Case Example - 2

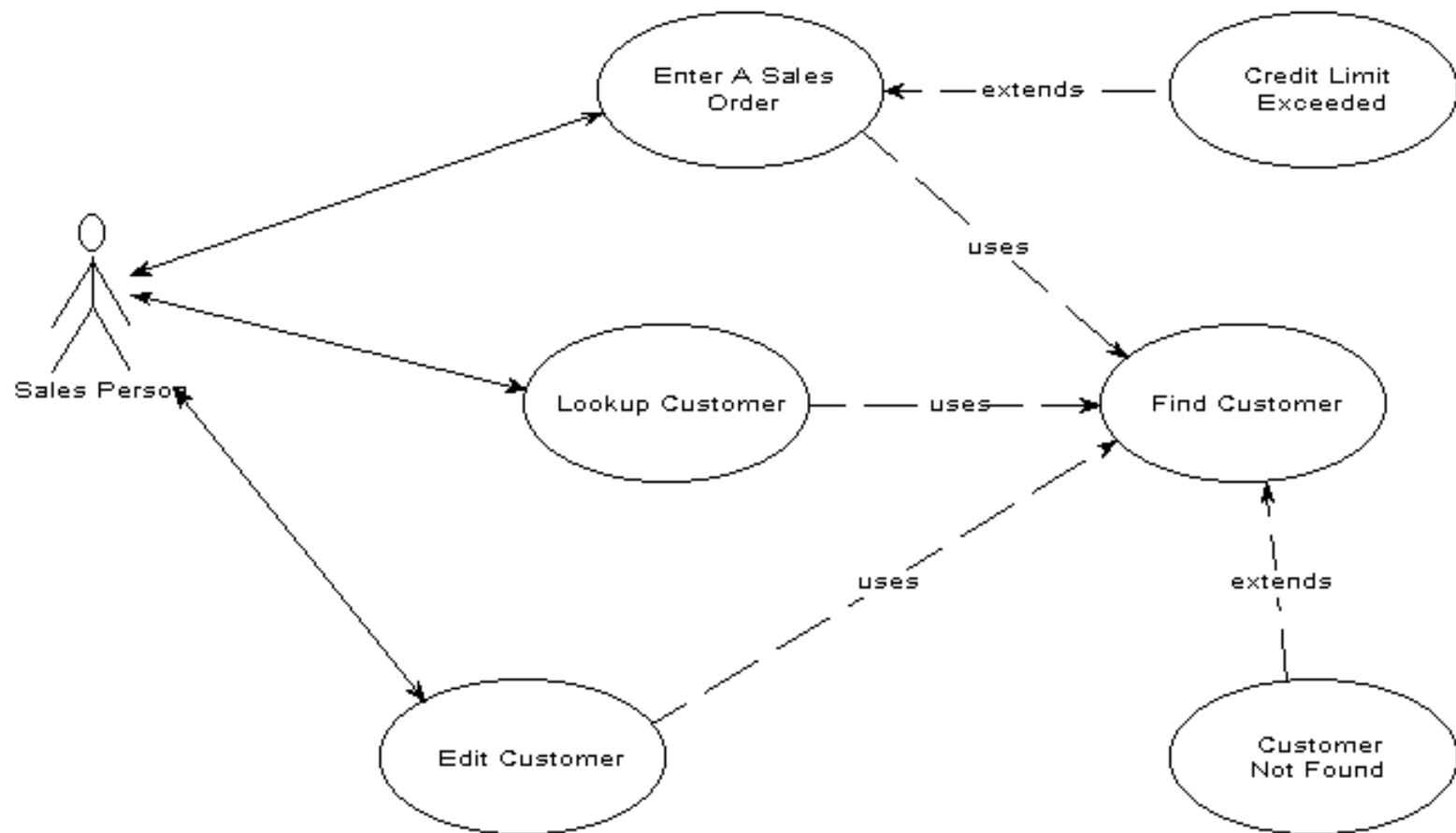
- **Read/Write/Modify All Users**
 - This level of users will be able to do all the record maintenance tasks. They will be able to modify any records created by any users.
- **Full Control Users**
 - This is the system administrative level which will be able to change any application settings, as well as maintaining user profiles.

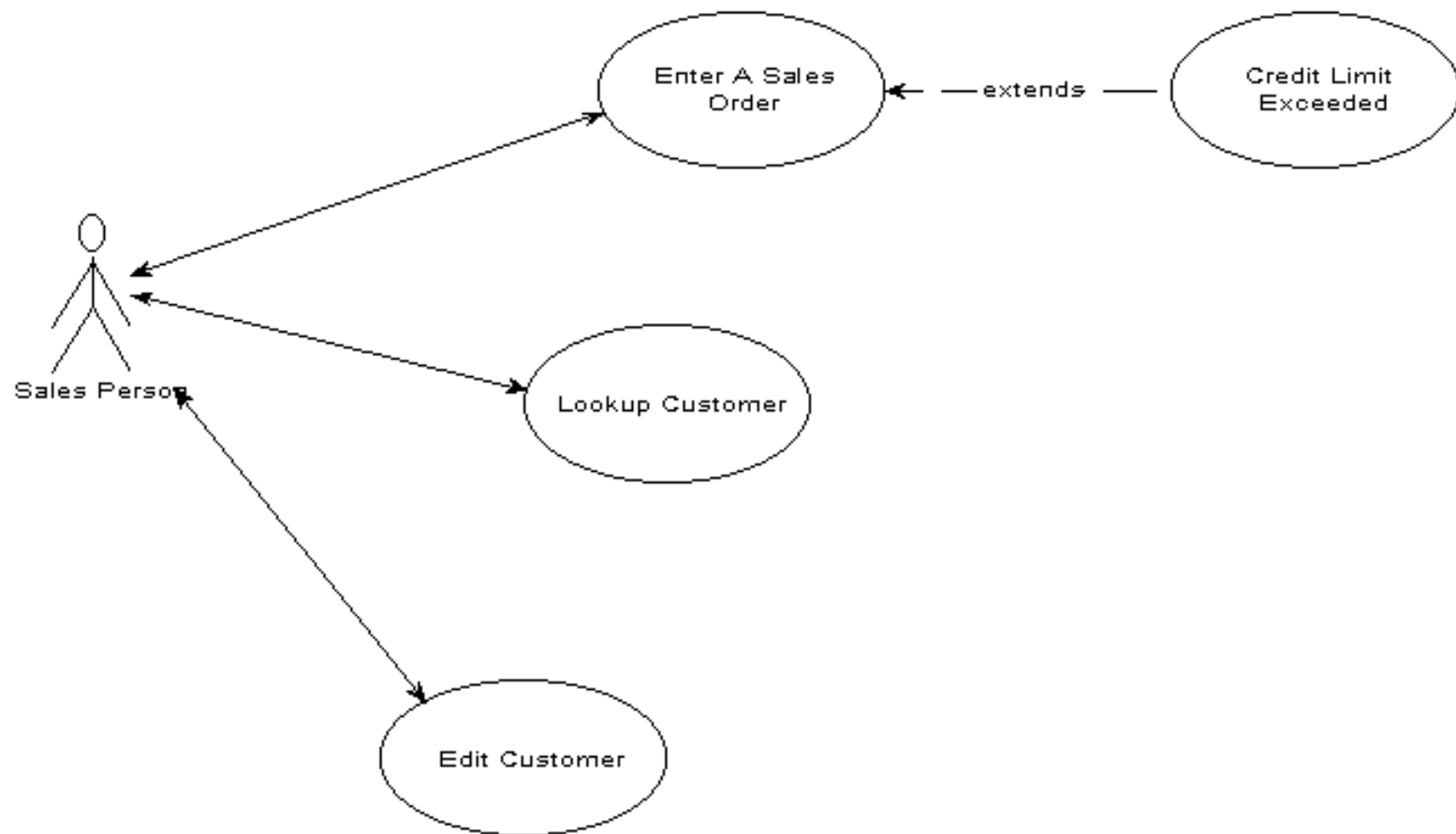
Description

The Sales Order Clerk enters the customer's surname. The system displays all matching customers. The Sales Order Clerk selects one of those customers and the system displays that customer's details.

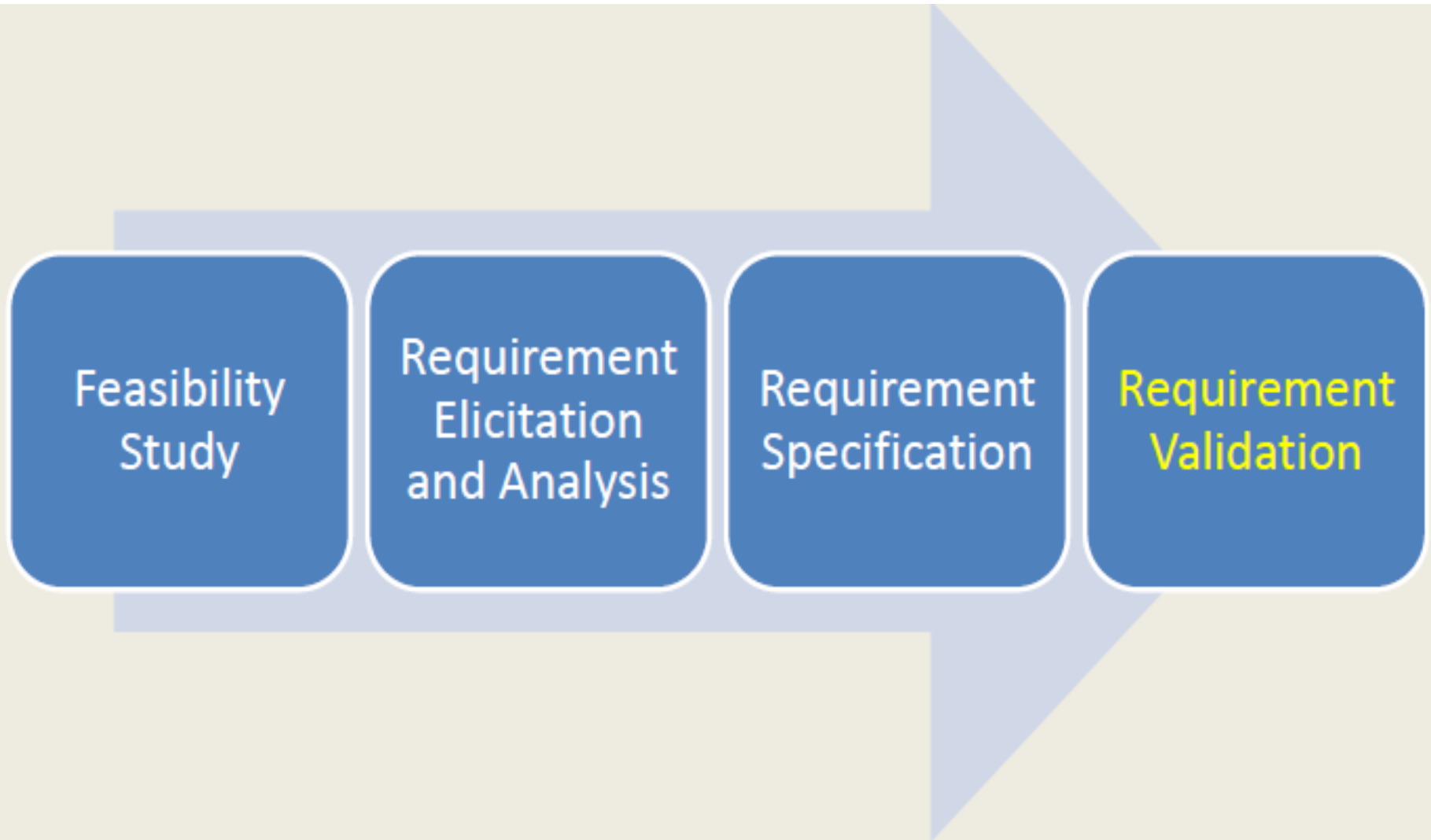
For each item that the customer wishes to order the Sales Order Clerk enters the line details. When all the line items have been entered the system confirms the order.







Requirements engineering process



Requirements validation

- Concerned with demonstrating that the requirements define the system that the customer really wants.
- Requirements error costs are high so validation is very important
 - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.
- SMART requirement!
- Focus: correctness and completeness

Requirements validation techniques

- ❖ **Review:** a manual process that involves multiple readers checking document for anomalies and omissions.
- ❑ Regular reviews should be held while the requirements definition is being formulated.
- ❑ The review team:
 - review leader
 - producer
 - recorder
 - reviewers
- ❑ Reviews may be formal (with completed documents) or informal.
- ❑ Good communications between developers, customers and users can resolve problems at an early stage.

Requirements validation techniques

❖ **Prototyping** : Using an executable model of the system to check requirements.

Goal: quickly generate something that can be tested by user

- no design, no documentation
- minimal implementation
- fake calculations
- assume input in simplistic format, etc.
- no test plan
- test with users: talk them through the form
- throw away after it is used: not delivered or reused!

Summary

- ❑ Requirements engineering process
 - Feasibility study
 - Requirement elicitation and analysis
 - *Discovery, Organization, Prioritization and Documentation
 - *Interviews
 - *Scenarios
 - Requirement specification
 - *Use case model
 - *Use case diagram
 - Requirement validation
 - *Reviews
 - *Prototyping

Chapter Four: Formal Requirements

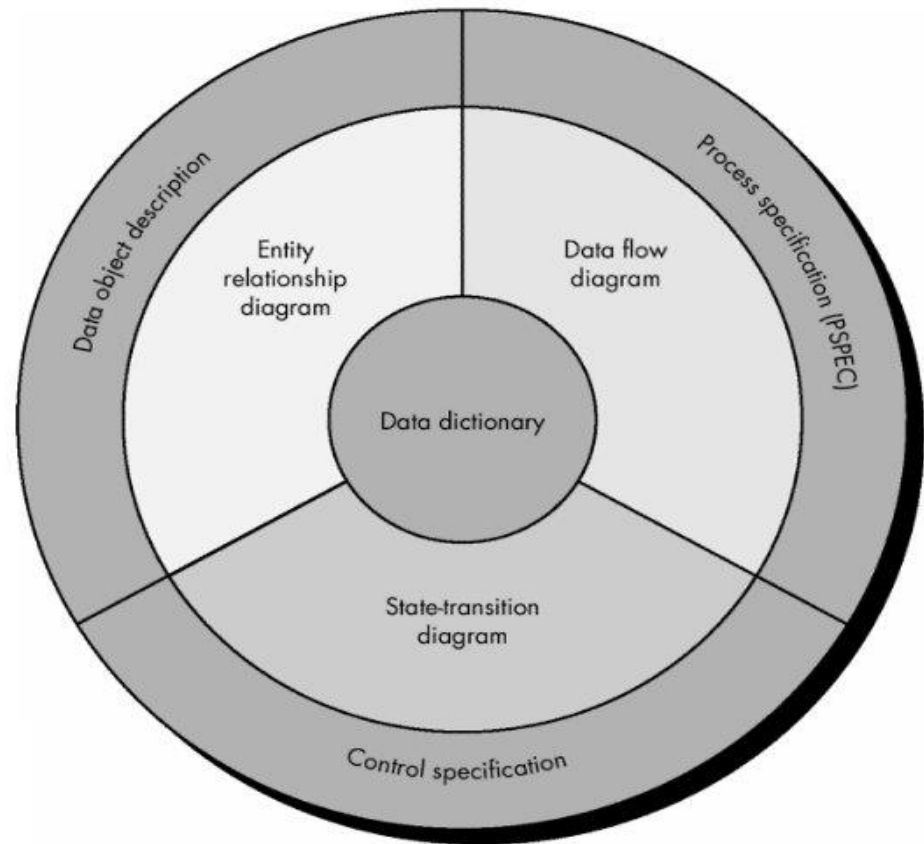
- ❑ **Structure Analysis**
- ❑ **Analysis Model Objectives**
- ❑ **The Elements of Analysis Model**
 - **Data Dictionary**
 - **Entity Relationship Diagram(ERD) and Data Object Description**
 - **Data Flow Diagram(DFD) and Process Specification**
 - **State Transition Diagram (STD) and Control Specification**
- ❑ **Data Modeling Elements**
 - **Attributes**
 - **Relationships(Cardinality and Modality)**
 - **Creation of ERD**
 - **Creation of DFD**

Structure Analysis

Structured analysis is a software engineering technique that uses **graphical diagrams** to develop and portray system **specifications** that are easily understood by users. These diagrams describe the **steps** that need to occur and the **data** required to meet the design function of particular software. This type of analysis mainly **focuses** on logical systems and functions, and **aims** to convert business requirements into computer programs and hardware specifications.

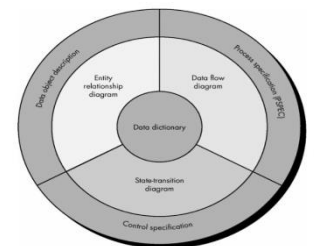
Analysis Model Objectives

- Describe what the customer requires.
- Establish a basis for the creation of a software design.
- Devise a set of requirements that can be validated once the software is built.



The Elements of Analysis Model

- 1- **Data dictionary** - contains the descriptions of all **data objects** consumed or produced by the software.
 - It stores **meaning** and **origin** of data, its **relationship** with other data, data **format** for usage etc. Data dictionary has rigorous definitions of all names in order to facilitate user and software designers.
 - It is often referenced as **meta-data** (data about data) repository. It is created along with DFD (Data Flow Diagram) model of software program and is expected to be updated whenever DFD is changed or updated.
 - The data is **referenced** via data dictionary while designing and implementing software.
 - Data dictionary removes any chances of ambiguity. It helps keeping work of programmers and designers synchronized while using same object reference everywhere in the program.
 - It provides a way of documentation for the complete database system in one place.
 - Validation of DFD is carried out using data dictionary.



A- Contents

Data dictionary should contain information about the following:

- Data Flow
- Data Structure
- Data Elements
- Data Stores
- Data Processing

=	Composed of
{ }	Repetition
()	Optional
+	And
[/]	Or

Data Flow is described by means of DFDs and represented in algebraic form as described.

Example

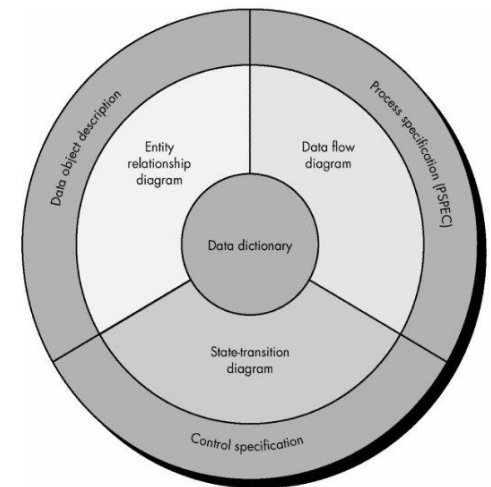
Address = House No + (Street / Area) + City + State

Course ID = Course Number + Course Name + Course
Level + Course Grades

B- Data Elements

Data elements consist of Name and descriptions of Data and Control Items, Internal or External data stores etc. with the following details:

- ✓ Primary Name
- ✓ Secondary Name (Alias)
- ✓ Use-case (How and where to use)
- ✓ Content Description (Notation etc)
- ✓ Supplementary Information (preset values, constraints etc.)



C- Data Store

It stores the information from where the data enters into the system and exists out of the system. The Data Store may include :—

- Files

- Internal to software.
- External to software but on the same machine.
- External to software and system, located on different machine.

- Tables

- Naming convention
- Indexing property

D- Data Processing

There are two types of Data Processing:

Logical: As user sees it

Physical: As software sees it

2- Entity relationship diagram (ERD):- depicts relationships between data objects.

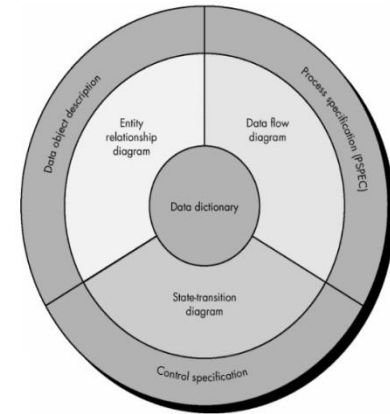
➤ **Primary components identified for the ERD:**

- data objects
- attributes
- relationships
- type indicators

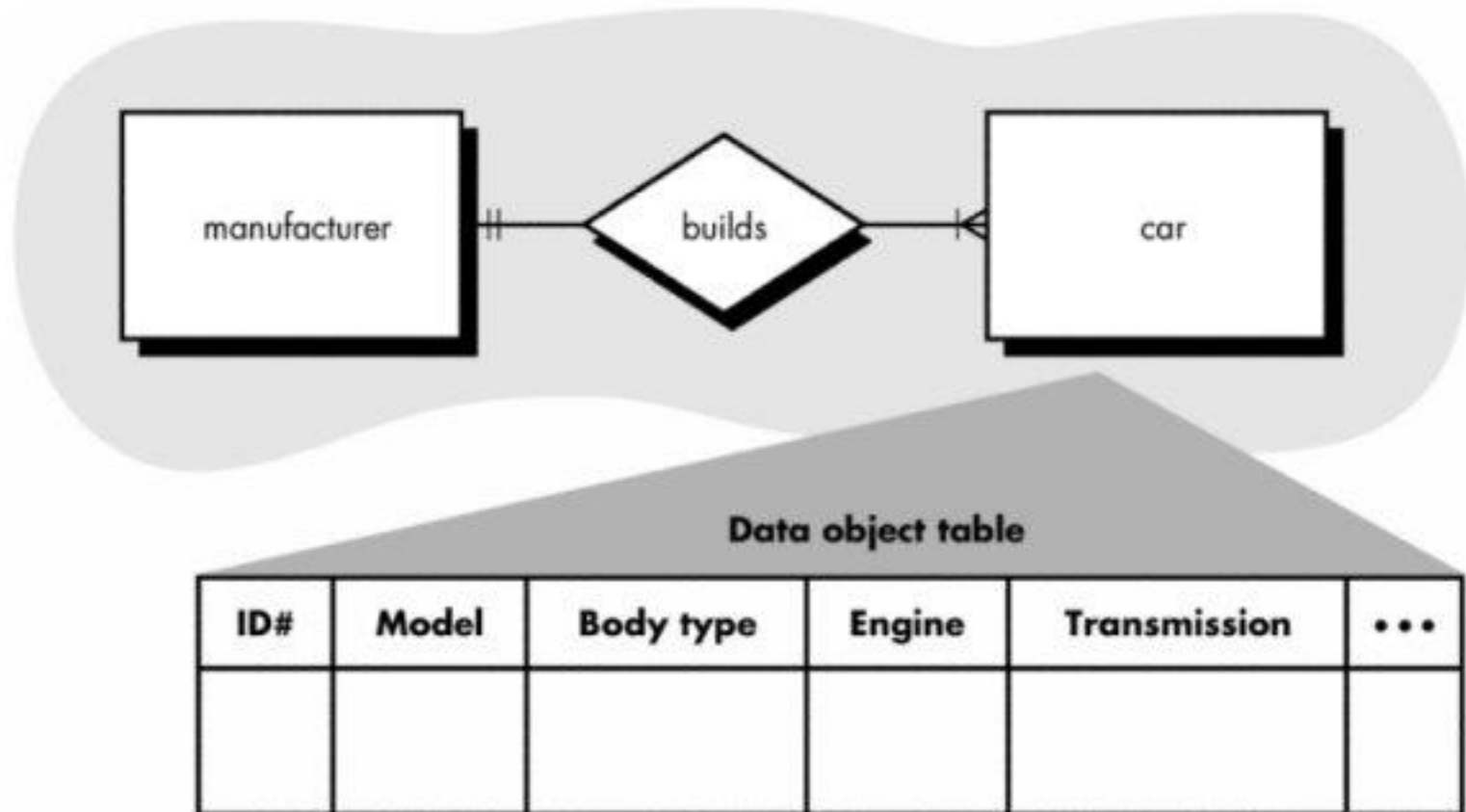
➤ **Primary purpose:** represent data objects and their relationships

➤ **Iconography:**

- **Data objects** are represented by a **labeled rectangle**
- **Relationships** are indicated with a **labeled line** connecting objects (Some variations: the connecting line contains a diamond that is labeled with the relationship)
- **Connections** between data objects and relationships are established using a variety of **special symbols** that indicate cardinality and modality



A simple ERD and data object table (Note: In this ERD the relationship *builds* is indicated by a diamond)



Creating Entity Relationship Diagrams (ERD)

1. During requirements elicitation, customers are asked to list the "things" that the application or business process addresses. These "things" evolve into a list of input and output data objects as well as external entities that produce or consume information.
2. Taking the objects one at a time, the analyst and customer define whether or not a connection (unnamed at this stage) exists between the data object and other objects.
3. Wherever a connection exists, the analyst and the customer create one or more object/relationship pairs.
4. For each object/relationship pair, cardinality and modality are explored.
5. Steps 2 through 4 are continued iteratively until all object/relationships have been defined. It is common to discover omissions as this process continues. New objects and relationships will invariably be added as the number of iterations grows.
6. The attributes of each entity are defined.
7. An entity relationship diagram is formalized and reviewed.
8. Steps 1 through 7 are repeated until data modeling is complete.

3- Data flow diagram (DFD) - provides an indication of how **data are transformed** as they move through the system; also **depicts functions** that transform the data flow (a function is represented in a DFD using a process specification or PSPEC).

DFD is graphical representation of flow of data in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data. The DFD does not mention anything about how data flows through the system.

There is a difference between DFD and Flowchart. The **flowchart** depicts flow of control in program modules. **DFDs** depict flow of data in the system at various levels. DFD does not contain any control or branch elements.

Data Flow Diagrams are either **Logical** or **Physical**.

Logical DFD - Concentrates on the system process and flow of data in the system. For example in a Banking software system, how data is moved between different entities.

Physical DFD - Shows how the data flow is actually implemented in the system. It is more specific and close to the implementation.

DFD Components

DFD can represent Source, destination, storage and flow of data using the following set of components -



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

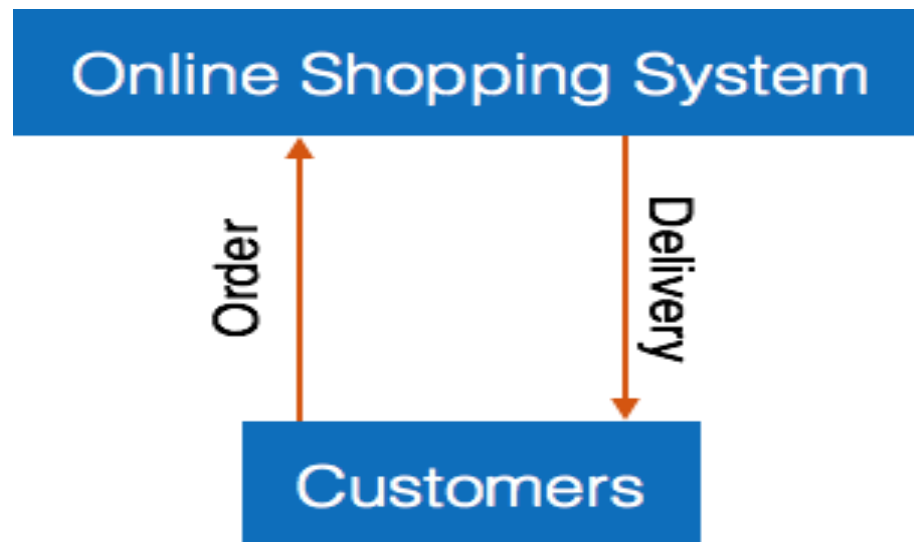
Process - Activities and action taken on the data are represented by Circle or Round-edged rectangles.

Data Storage - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.

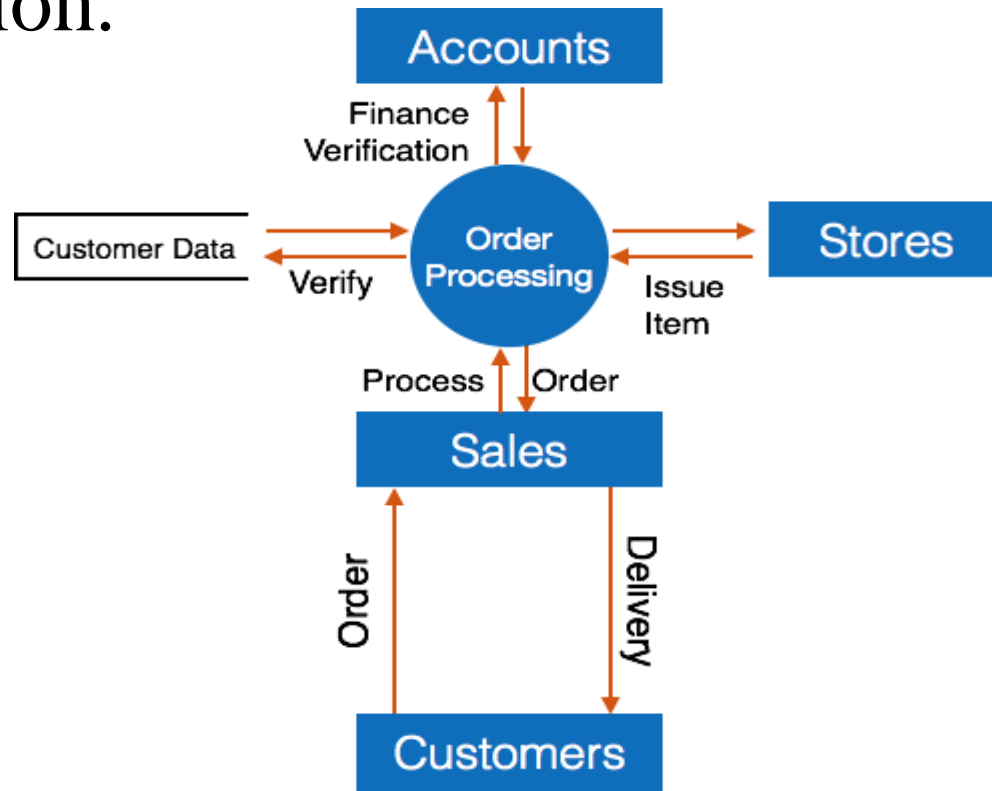
Data Flow - Movement of data is shown by pointed arrows. Data movement is shown from the base of arrow as its source towards head of the arrow as destination.

Levels of DFD

Level 0 - Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs.



Level 1 - The Level 0 DFD is broken down into more specific, Level 1 DFD. Level 1 DFD depicts basic modules in the system and flow of data among various modules. Level 1 DFD also mentions basic processes and sources of information.



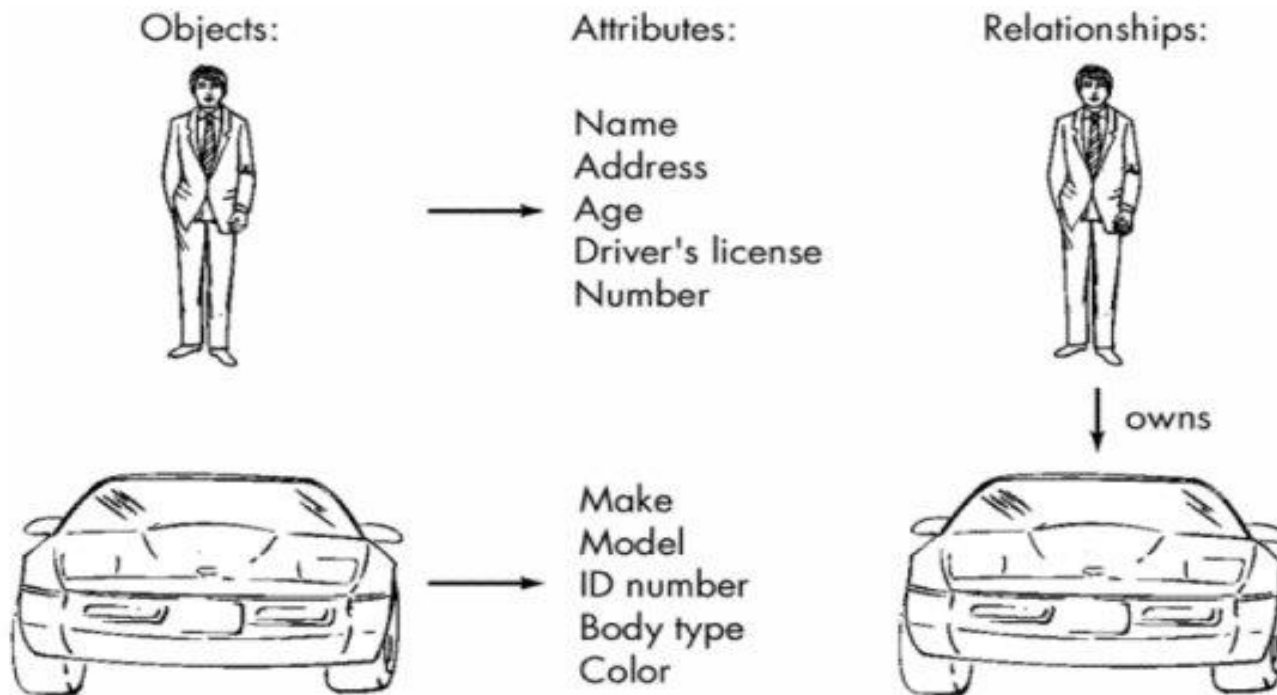
Level 2 - At this level, DFD shows how data flows inside the modules mentioned in Level 1.

Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

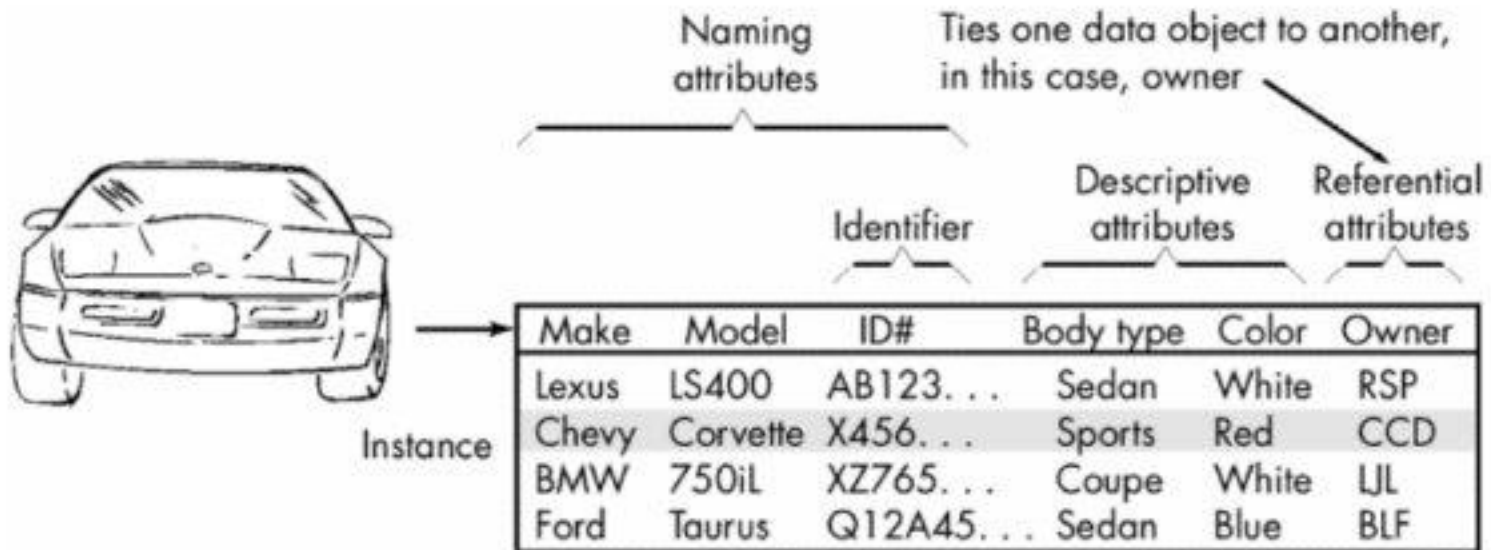
4- State transition diagram (STD) - indicates how the system behaves as a consequence of external events, states are used to represent behavior modes. Arcs are labeled with the events triggering the transitions from one state to another (control information is contained in control specification or CSPEC).

Data Modeling Elements (ERD)

Attributes - name a data object instance, describe its characteristics, or make reference to another data object



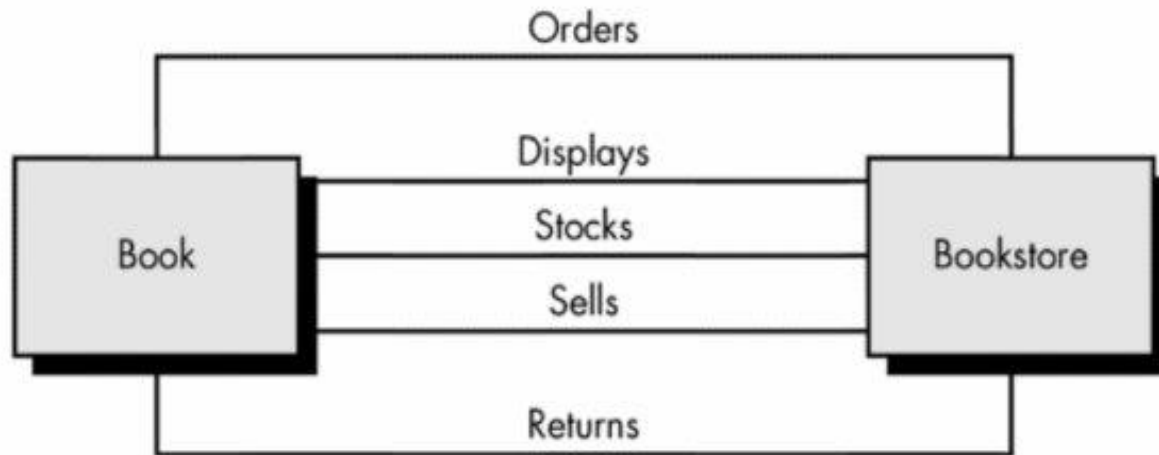
Tabular representation of data object



Relationships - indicate the manner in which data objects are connected to one another .



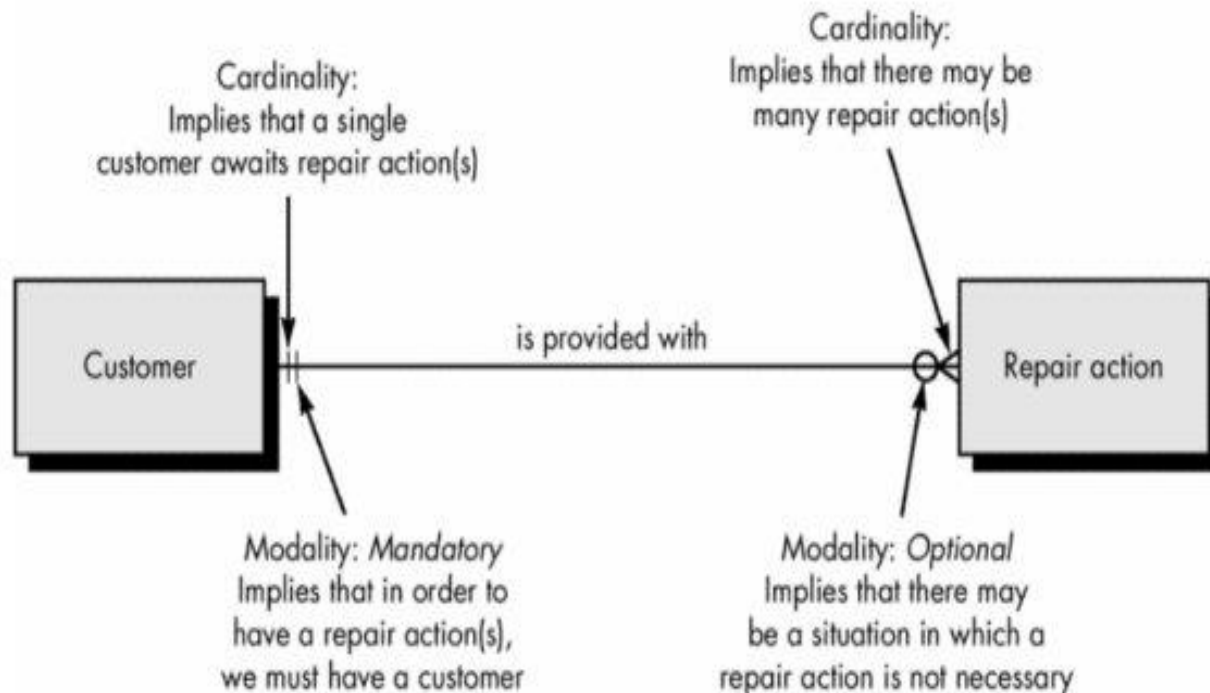
(a) A basic connection between objects



(b) Relationships between objects

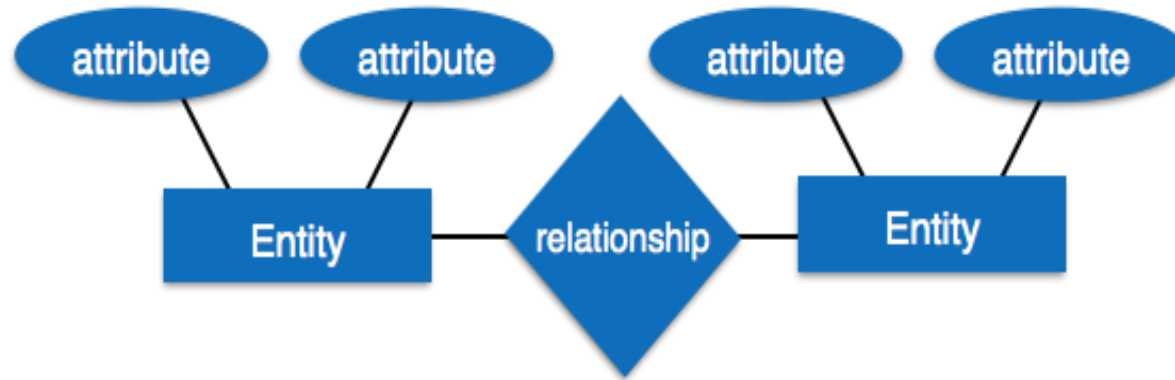
Cardinality and Modality (ERD)

- **Cardinality** - in data modeling, cardinality specifies how the number of occurrences of one object are related to the number of occurrences of another object (1:1, 1:N, M:N)
- **Modality** - zero (0) for an optional object relationship and one (1) for a mandatory relationship



Entity-Relationship Model: Entity-Relationship model is a type of database model based on the notion of real world entities and relationship among them. We can map real world scenario onto ER database model. ER Model creates a set of entities with their attributes, a set of constraints and relation among them.

ER Model is best used for the conceptual design of database. ER Model can be represented as follows:



Entity - An entity in ER Model is a real world being, which has some properties called *attributes*. Every attribute is defined by its corresponding set of values, called *domain*.

For example, Consider a **school database**. Here, a **student** is an **entity**. Student has various **attributes** like **name**, **id**, **age** and **class** etc.

Relationship - The logical association among entities is called *relationship*. Relationships are mapped with entities in various ways. Mapping cardinalities define the number of associations between two entities.

Mapping cardinalities:

- one to one
- one to many
- many to one
- many to many

Chapter Five: Software Design

- ☐ **Software Design Definition**
- ☐ **Activities of Software Design**
- ☐ **Effective Modular Design**
- ☐ **Introduction to Object Oriented Design**
- ☐ **Top Down and Bottom up Design**

Software Design Definition

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

Activities of Software Design

Software design yields three levels of results:

Architectural Design - The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.

High-level Design- The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.

Detailed Design- Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

Effective Modular Design

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Modular design unintentionally follows the rules of ‘divide and conquer’ problem-solving strategy this is because there are many other benefits attached with the modular design of software.

Advantage of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

Coupling and Cohesion

Cohesion

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

Co-incidental cohesion - It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.

Logical cohesion - When logically categorized elements are put together into a module, it is called logical cohesion.

Temporal Cohesion - When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.

Procedural cohesion - When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.

Communicational cohesion - When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.

Sequential cohesion - When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.

Functional cohesion - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

Coupling

Is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

There are five levels of coupling, namely –

Content coupling - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.

Common coupling- When multiple modules have read and write access to some global data, it is called common or global coupling.

Control coupling- Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.

Stamp coupling- When multiple modules share common data structure and work on different part of it, it is called stamp coupling.

Data coupling- Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Cohesion - grouping of all functionally related elements.

Coupling - communication between different modules.

Ideally, no coupling is considered to be the best.

Object Oriented Design

Objects - All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.

Classes - A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

Encapsulation - In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.

Inheritance - OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.

Polymorphism - OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Software Design Approaches

1. Top Down Design

We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their own set of sub-system and components and creates hierarchical structure in the system.

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.

Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

2. Bottom-up Design

The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

Software Testing is evaluation of the software against requirements gathered from users and system specifications. Testing is conducted at the phase level in software development life cycle or at module level in program code. Software testing comprises of Validation and Verification.

Chapter Six: Software Validation and Verification

Why Testing?

Two objectives – Verification and Validation (V&V):

To uncover errors (or bugs) in the software before delivery to the client. This is called **Verification** -Verify that the program is working. “Are you building the product right? Right?”

To ascertain that the software meet its requirement specification. This is called **Validation** – Validate that the software meets its requirements. “Are you building the right product? “

Software Validation

Validation is process of examining whether or not the software satisfies the user requirements. It is carried out at the end of the SDLC. If the software matches requirements for which it was made, it is validated.

- Validation ensures the product under development is as per the user requirements.
- Validation answers the question – "Are we developing the product which attempts all that user needs from this software?"
- Validation emphasizes on user requirements.

Software Verification

Verification is the process of confirming if the software is meeting the business requirements, and is developed adhering to the proper specifications and methodologies.

- Verification ensures the product being developed is according to design specifications.
- Verification answers the question– "Are we developing this product by firmly following all design specifications?"
- Verifications concentrate on the design and system specifications.

Target of the test are –

- **Errors** - These are actual coding mistakes made by developers. In addition, there is a difference in output of software and desired output, is considered as an error.
- **Fault** - When error exists fault occurs. A fault, also known as a bug, is a result of an error which can cause system to fail.
- **Failure** - failure is said to be the inability of the system to perform the desired task. Failure occurs when fault exists in the system.

What is Testing?

- ❑ At a lower level, testing involves designing a series of “TEST CASES” (or “TEST SUITE”) to uncover errors and validate conformance to requirements.
- ❑ At a higher level, testing involves formulating a test plan and test strategy for the execution of the testing process.

Example on Testing

Requirement:

Write a program to assign an alphabetic grade to raw marks as follows:

Marks	Grade
0 - 49	'F'
50 - 59	'E'
60 - 69	'D'
70 - 79	'C'
80 - 89	'B'
90 - 100	'A'

Program without Testing

```
if (marks >= 90 && marks < 100) return 'A';  
else if (marks >= 80 && marks < 90) return 'B';  
else if (marks >= 70 && marks < 80) return 'C';  
else if (marks >= 60 && marks < 70) return 'D';  
else if (marks >= 50 && marks < 60) return 'E';  
else return 'F';
```

This code can compile and run! (Compiler only catches syntax errors, NOT semantics errors or logical errors.)

- Is the program working? (May be!) (Verification)
- Is the program correct? Does the program meet its specification? (NO!) (Validation)
- Is the program efficient? (This is an issue on Software Quality Assurance (SQA), not testing—There are 10 comparisons in the code, many of them are redundant!)

Test Cases

- To verify and validate the program, we design “a series of test cases”. Each test case contains a specific input and the expected output. For examples,

Test Case No	Input	Expected Output	Purpose of Test
1	100	'A'	Grade 'A' upper limit
2	49	'F'	Grade 'F' upper limit
3	1	'F'	Grade 'F'
4	2	'F'	Grade 'F'
5

How many test cases is “necessary and sufficient”? (101? How about numbers like 200, or–155? Countable Infinity!)

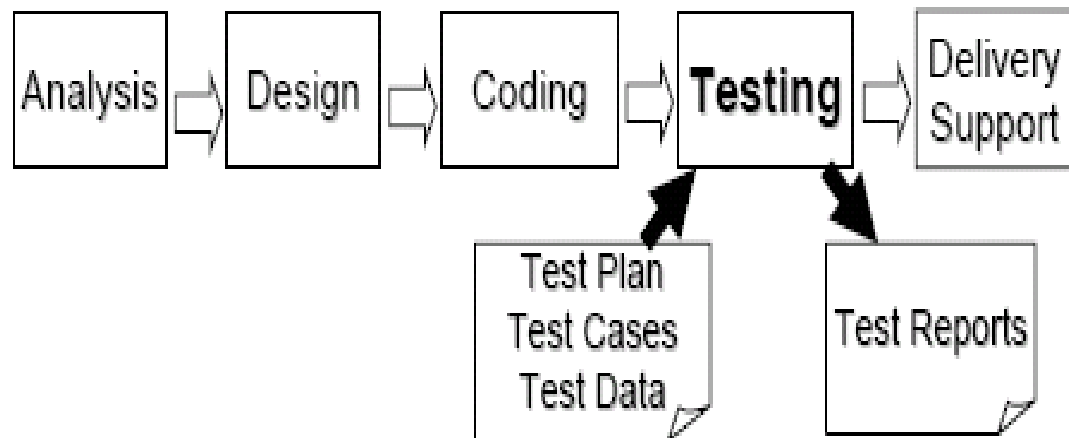
- This series of lectures on “testing techniques” teaches you how to design good test cases “applying sound engineering principles”.

What Testing Shows?

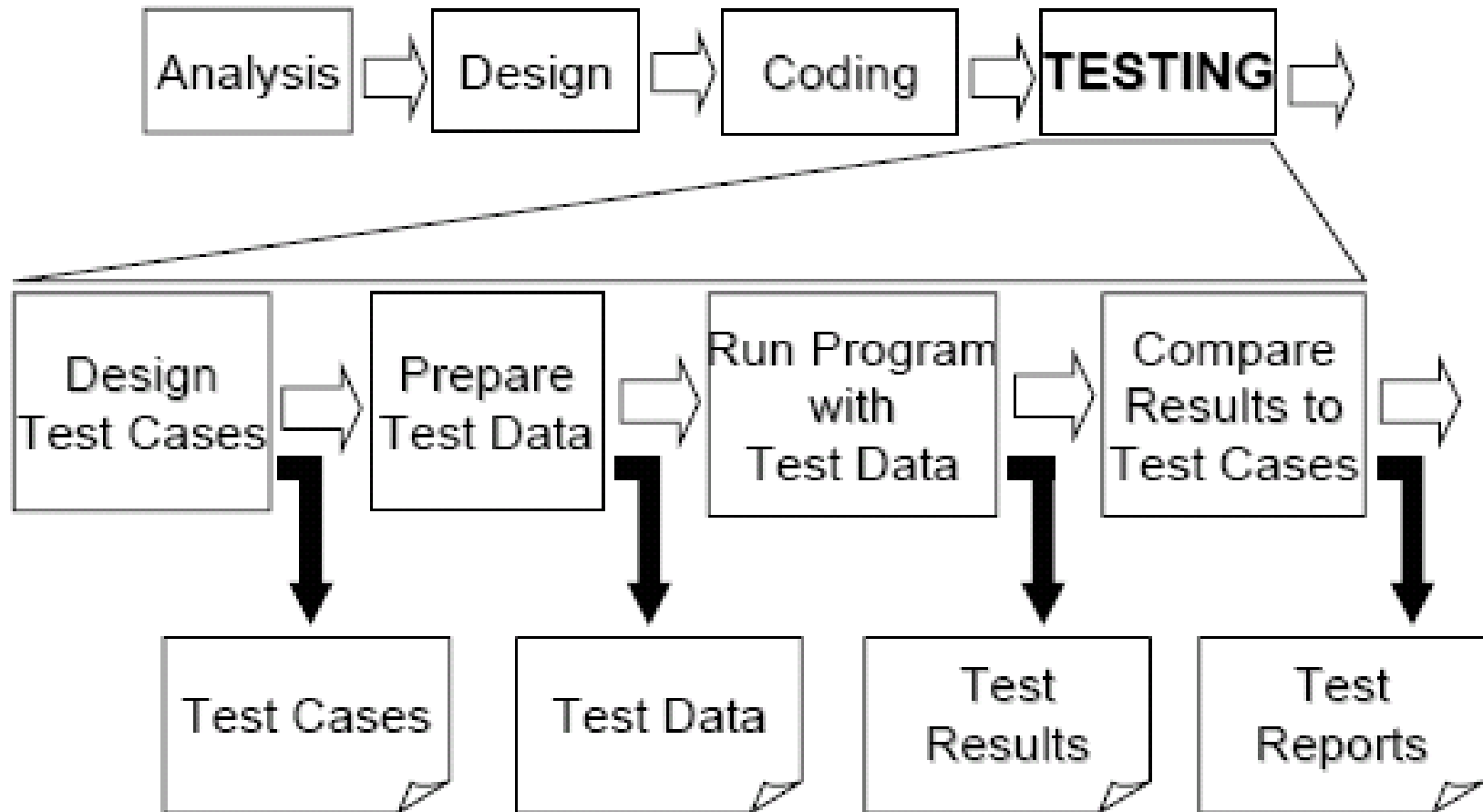


Testing Stage of the Software Process

- The goal of testing is to “design a series of test cases” that has “a high likelihood of finding errors”.
- How? Design test cases systematically by “applying sound engineering principles and methods”.
- The work product of the testing stage is a “test report” that documents all the test cases run, i.e., the test input, the expected output, the actual output, the purpose of the test and etc.



Testing Stage Details



Who tests the system?

- During the early stages of testing, the developer performs the tests. As the testing progresses, independent test specialist may involved.
- “Open–source” software like Linux, Java, Apache are known to be more secure and less buggy because many independent parties have “tested” the source code.

Chapter Seven: Software Testing

Testing Approaches

Tests can be conducted based on two approaches: –

- Functionality testing
- Implementation testing

When **functionality** is being tested without taking the actual implementation in concern it is known as **black-box** testing. The other side is known as **white-box** testing where not only functionality is tested but the way it is implemented is also analyzed.

Exhaustive tests are the best-desired method for a perfect testing. Every single possible value in the range of the input and output values is tested.

Black-box testing

It is carried out to test functionality of the program. It is also called ‘Behavioral’ testing. The tester in this case, has a set of input values and respective desired results. On providing input, if the output matches with the desired results, the program is tested ‘ok’ and problematic otherwise.



In this testing method, the **design** and **structure** of the code are **not known** to the tester, and testing engineers and end users conduct this test on the software.

Black-box testing techniques:

Equivalence class - The input is divided into similar classes. If one element of a class passes the test, it is assumed that all the class is passed.

Boundary values - The input is divided into higher and lower end values. If these values pass the test, it is assumed that all values in between may pass too.

Cause-effect graphing - In both previous methods, only one input value at a time is tested. Cause (input) – Effect (output) is a testing technique where combinations of input values are tested in a systematic way.

Pair-wise Testing - The behavior of software depends on multiple parameters. In pairwise testing, the multiple parameters are tested pair-wise for their different values.

State-based testing - The system changes state on provision of input. These systems are tested based on their states and input.

White-box testing

It is conducted to test program and its implementation, in order to improve code efficiency or structure. It is also known as ‘Structural’ testing. The design and structure of the code are known to the tester. Programmers conduct this test on the code.

The below are some White-box testing techniques:

Control-flow testing - The purpose of it is to set up test cases which cover all statements and branch conditions. The **branch** conditions are tested for both being true and false, so that all statements can be covered.

Data-flow testing - Emphasis to cover all the data variables included in the program. It tests where the variables were declared and defined and where they were used or changed.



Testing Levels

Unit Testing

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

Integration Testing

There is a need to find out if the units if integrated together would also work without errors. For example, argument passes and data updating etc.

System Testing

The software is compiled as product and then it is tested as a whole. This can be accomplished using one or more of the following tests:

Functionality testing - Tests all functionalities of the software against the requirement.

Performance testing - This test proves how efficient the software is. It tests the effectiveness and average time taken by the software to do desired task.

Security & Portability - These tests are done when the software is meant to work on various platforms and accessed by number of persons.

Acceptance Testing

Alpha testing - The team of developer themselves perform alpha testing by using the system as if it is being used in work environment. They try to find out how user would react to some action in software and how the system should respond to inputs.

Beta testing - After the software is tested internally, it is handed over to the users to use it under their production environment only for testing purpose. This is not as yet the delivered product. Developers expect that users at this stage will bring minute problems, which were skipped to attend.