22CSC51 – AGILE METHODOLOGIES

- Aravindhraj Natarajan, AP/CSE, KEC

Introduction to Software

Software

Software is a product

- **Transforms information** - produces, manages, acquires, modifies, displays, or transmits information
- **Delivers computing potential** of hardware and networks

Software is a vehicle for delivering a product

- **Controls other programs** (operating system)
- **Effects communications** (networking software)
- **Helps build other software** (software tools & environments)

What is Software ?

Software can define as:

- **Instructions** – executed provide desired features, function & performance.
- **Data structure** – *enable the programs* to adequately manipulate operation.
- **Documentation** – operation and use of the program.

Software products may be developed for a particular customer or may be developed for a general market.

- Software products may be
  - **Generic** - developed to be sold to a range of different customers e.g. PC software such as Excel or Word.
  - **Bespoke** (**custom**) - developed for a single customer according to their specification.

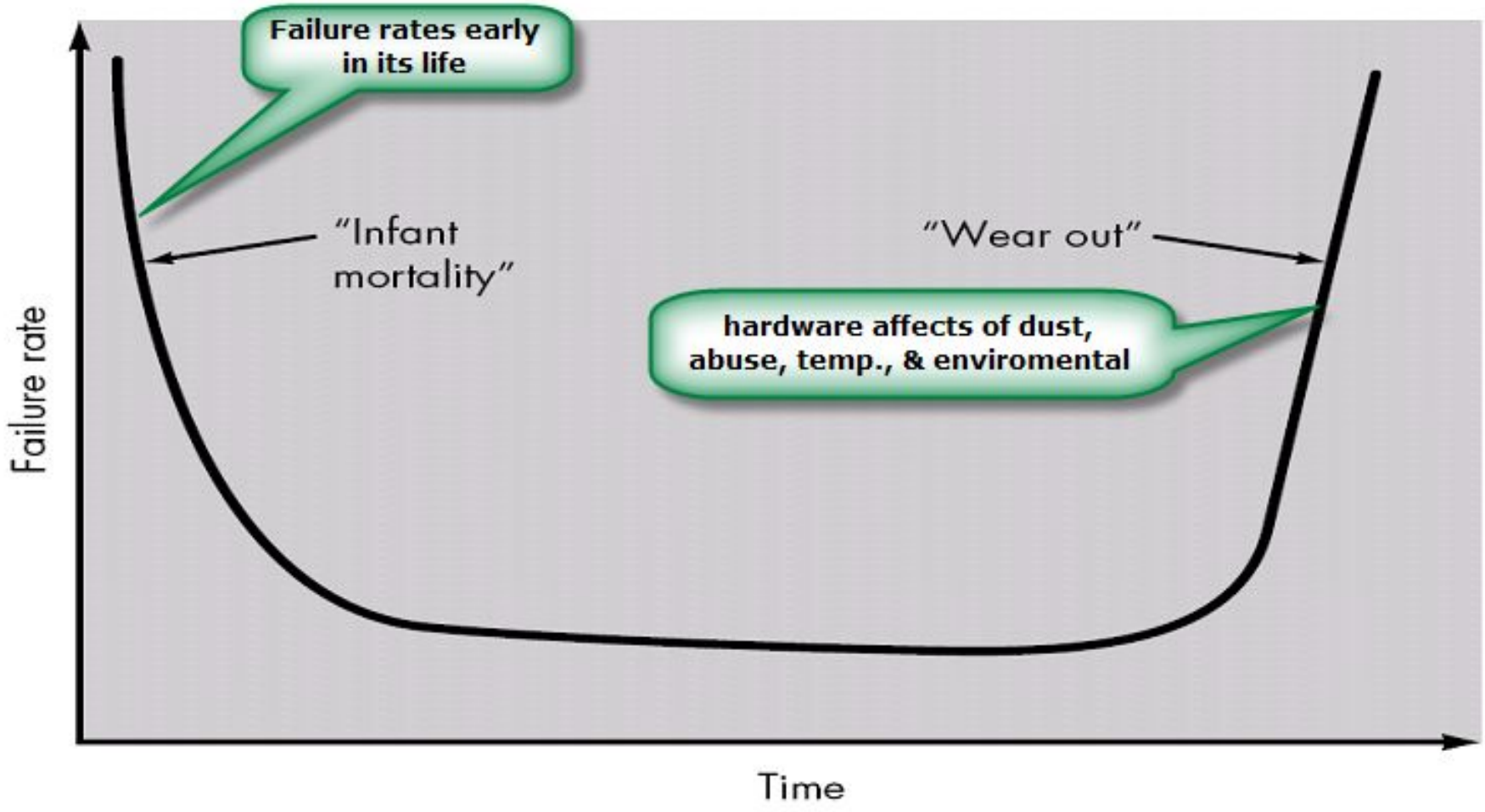
Hardware vs. Software

Hardware	Software
☐ Manufactured ☐ wear out ☐ Built using components ☐ Relatively simple	☐ Developed/ engineered ☐ deteriorate ☐ Custom built ☐ Complex

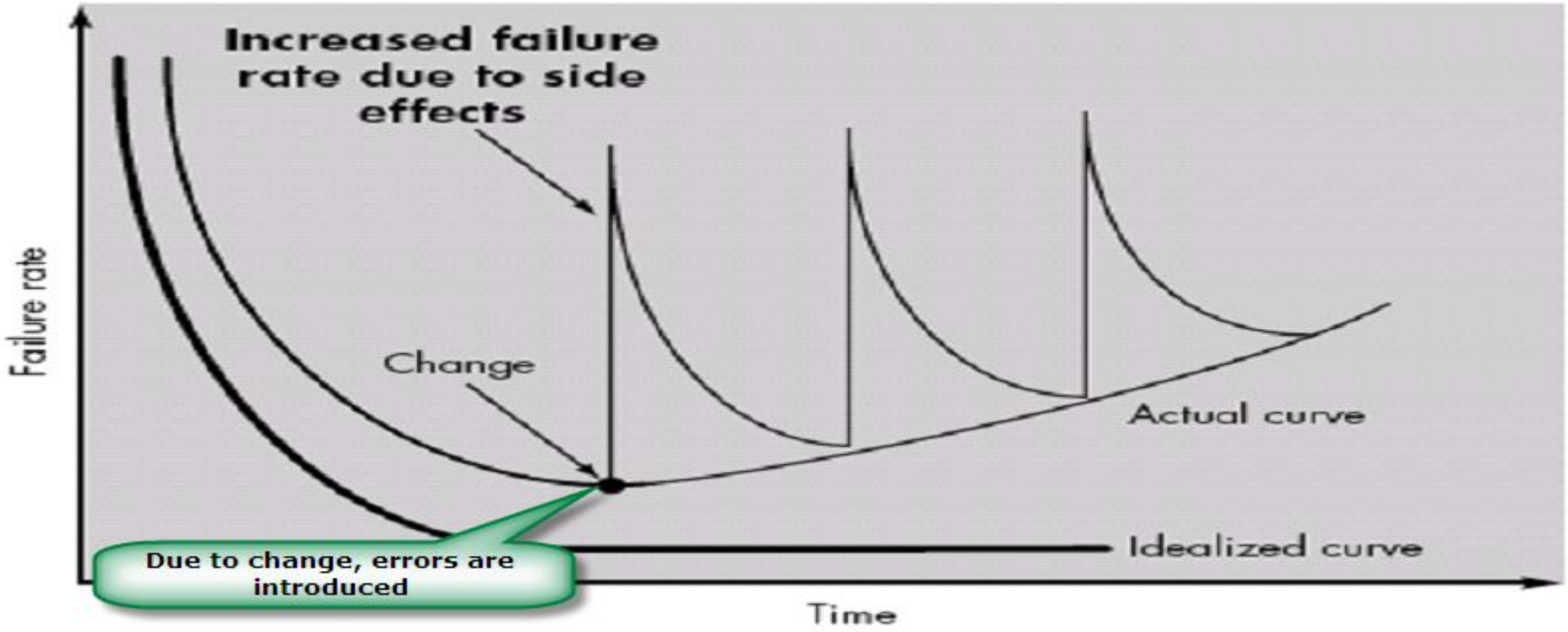
Manufacturing vs. Development

- Once a hardware product has been manufactured, it is difficult or impossible to modify. In contrast, software products are routinely modified and upgraded.
- In hardware, hiring more people allows you to accomplish more work, but the same does not necessarily hold true in software engineering.
- Unlike hardware, software costs are concentrated in design rather than production.

Failure curve for Hardware



Failure curve for Software



When a hardware component wears out, it is replaced by a spare part. There are **no software spare parts**. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably **more complexity.**

Component Based vs. Custom Built

- Hardware products typically employ many standardized design components.
- Most software continues to be custom built.
- The software industry does seem to be moving (slowly) toward component-based construction.

Software characteristics

- Software is developed or engineered; it is not manufactured.
- Software does not “wear out” but it does deteriorate.
- Software continues to be custom built, as industry is moving toward component based construction.

Software Application Domains

- System software
- Application software
- Engineering/scientific software
- Embedded software
- Product line software
- Web applications
- Artificial intelligence software

System Software:

- System software is a collection of programs written to service other programs.
- It is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

Ex. Compilers, operating system, drivers etc.

Application Software :

- Application software consists of standalone programs that solve a specific business need.
- Application software is used to control the business function in real-time.

Engineering /Scientific software:

- Characterized by "number crunching" algorithms.
- Applications range from astronomy to volcano logy, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.

Ex. Computer Aided Design (CAD), system stimulation etc.

Embedded Software:

- It resides in read-only memory and is used to control products and systems
- Embedded software can perform limited and esoteric functions.

Ex. keypad control for a microwave oven.

Product line software:

- Designed to provide a specific capability for use by many different customers, product line software can focus on a limited and esoteric marketplace.

Ex. Word processing, spreadsheet, CG, multimedia, etc.

Web Applications:

- Web apps can be little more than a set of linked hypertext files.
- It evolving into sophisticated computing environments that not only provide standalone features, functions but also integrated with corporate database and business applications.

Artificial Intelligence software

- AI software makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis

Ex. Robotics, expert system, game playing, etc.

Thank you….

Software Process and Process Model

Aravindhraj Natarajan, AP/CSE, KEC

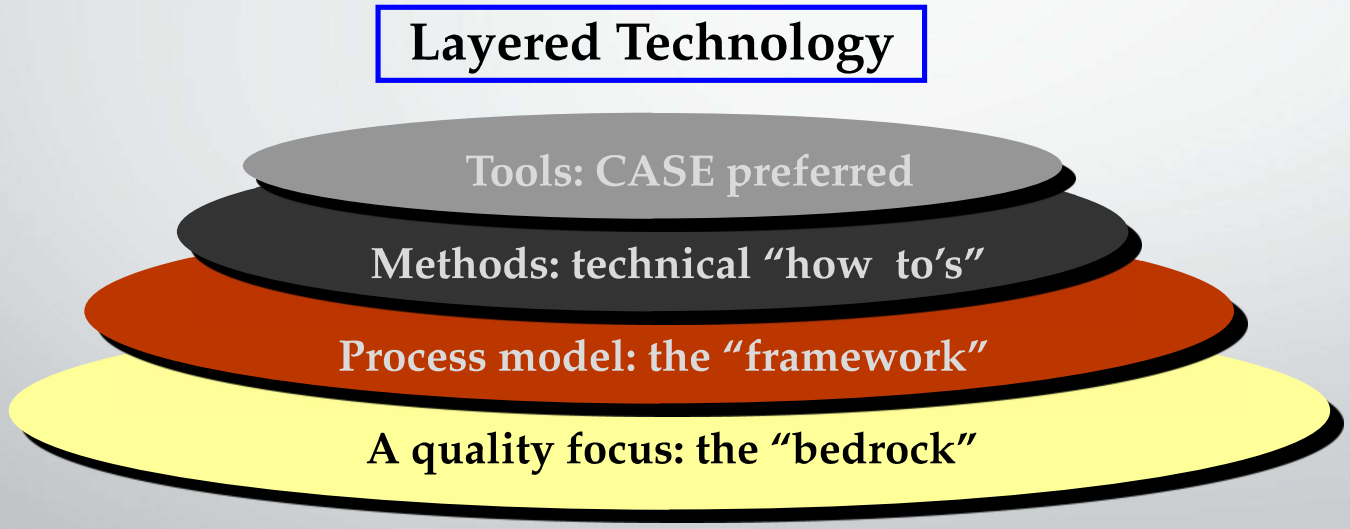
What is software engineering?

- **Definition** :
  - (1) The application of systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.(2) The study of approaches as in (1) above
- Its a discipline that is concerned with all aspects of software production.
- Software engineers should adopt
  - Systematic and organized approach to their work
  - Use appropriate tools and techniques depending on the problem to be solved
  - The development constraints and the resources available
- Apply Engineering Concepts to develop Software
- Challenge for Software Engineers is to produce high quality software with finite amount of resources & within a predicted schedule

Software Process

- **What? A software process** – as a framework for the tasks that are required to build high-quality software.
- **Who?** Managers, software engineers, and customers.
- **Why?** Provides stability, control, and organization to an otherwise chaotic activity.
- **Steps?** A handful of activities are common to all software processes, details vary.
- **Work product?** Programs, documents, and data.

Software Engineering – Layered Technology



Layered Technology

A quality Focus

- Every organization is rest on its commitment to quality.
- Total quality management, Six Sigma, or similar continuous improvement culture and it is this culture ultimately leads to development of increasingly more effective approaches to software engineering.
- The bedrock that supports software engineering is a quality focus.

Process:

- It's a foundation layer for software engineering.
- It's defined **framework** for a set of *key process areas* (KRA) to effectively manage and deliver quality software in a cost effective manner
- The processes define the tasks to be performed and the order in which they are to be performed

Layered Technology

Methods:

- It provide the technical **how-to's** for building software.
- Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and support.
- There could be more than one technique to perform a task and different techniques could be used in different situations.

Tools:

- Provide automated or semi-automated support for the process, methods and quality control.
- When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called ***computer-aided software engineering (CASE)***

Software process model

- Process models prescribe a distinct set of activities, actions, tasks, milestones, and work products required to engineer high quality software.
- Process models are not perfect, but provide roadmap for software engineering work.
- Software models provide stability, control, and organization to a process that if not managed can easily get out of control
- Software process models are adapted to meet the needs of software engineers and managers for a specific project.

Process framework

Why process :

A process defines who is doing what, when and how to reach a certain goal.

- To build complete software process.
- Identified a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.
- It encompasses a set of umbrella activities that are applicable across the entire software process.

Generic Process Framework Activities

- Communication:
  - Heavy communication with customers, stakeholders, team
  - Encompasses requirements gathering and related activities
- Planning:
  - Workflow that is to follow
  - Describe technical task, likely risk, resources will require, work products to be produced and a work schedule.
- Modeling:
  - Help developer and customer to understand requirements (Analysis of requirements) & Design of software
- Construction
  - Code generation: either manual or automated or both
  - Testing – to uncover error in the code.
- Deployment:
  - Delivery to the customer for evaluation
  - Customer provide feedback

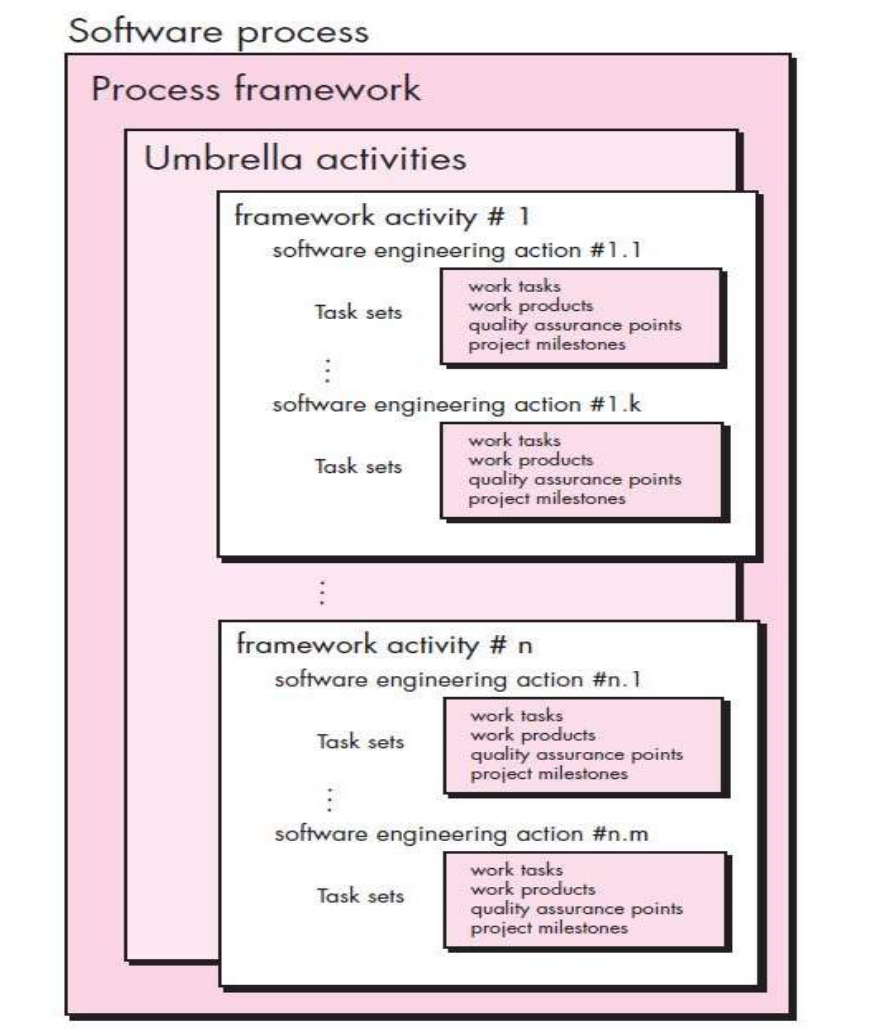
Umbrella Activities – Typical Specific Activities

- Software project tracking and control
  - Assessing progress against the project plan.
  - Take adequate action to maintain schedule.
- Formal technical reviews
  - Assessing software work products in an effort to uncover and remove errors before goes into next action or activity.
- Software quality assurance
  - Define and conducts the activities required to ensure software quality.
- Software configuration management
  - Manages the effects of change.
- Document preparation and production
  - Help to create work products such as models, documents, logs, form and list.

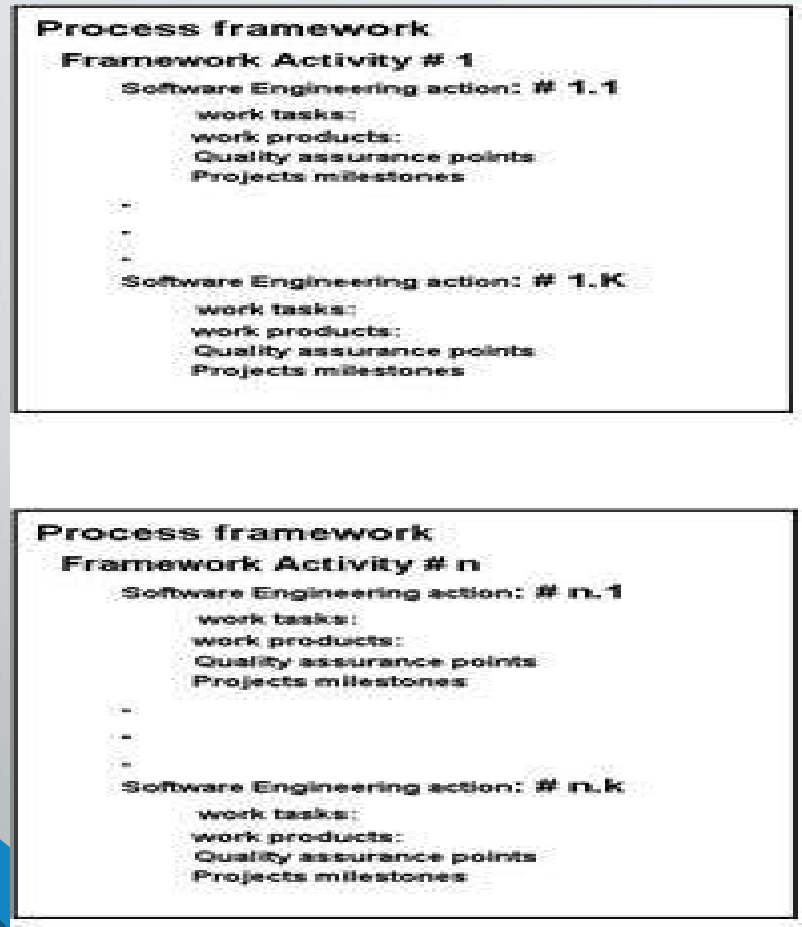
Umbrella Activities – Typical Specific Activities

- Reusability management
  - Define criteria for work product reuse
  - Mechanisms to achieve reusable components.
- Measurement
  - Define and collects process, project, and product measures
  - Assist the team in delivering software that meets customer's needs.
- Risk management
  - Assesses risks that may effect that outcome of project or quality of product (i.e. software)

Process Framework



Process Framework

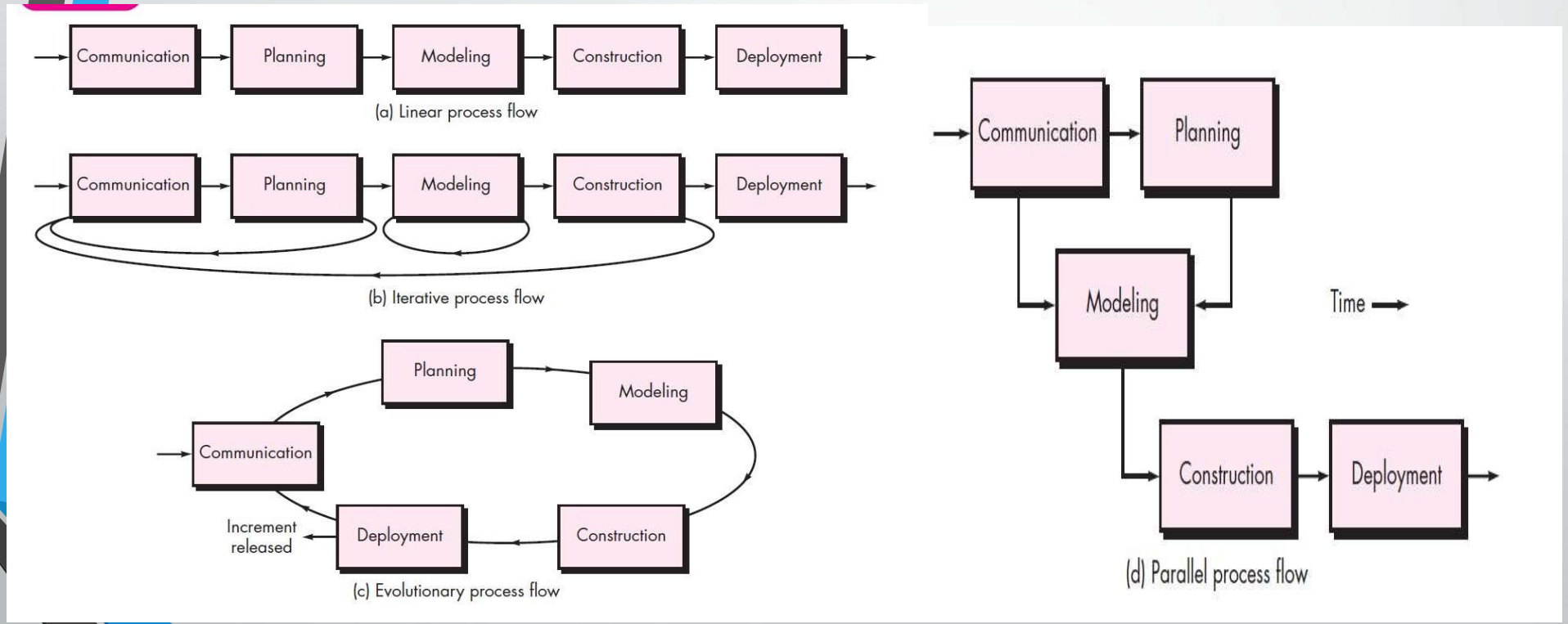


- Each framework activities is populated by a set for *software engineering actions* – a collection of related tasks.
- Each action has individual *work task*.

The Process Model: Adaptability

- The framework activities will always be applied on every project … BUT
- The tasks for each activity will vary based on:
  - The type of project (an "entry point" to the model)
  - Characteristics of the project
  - Common sense judgment; concurrence of the project team

Process Flow



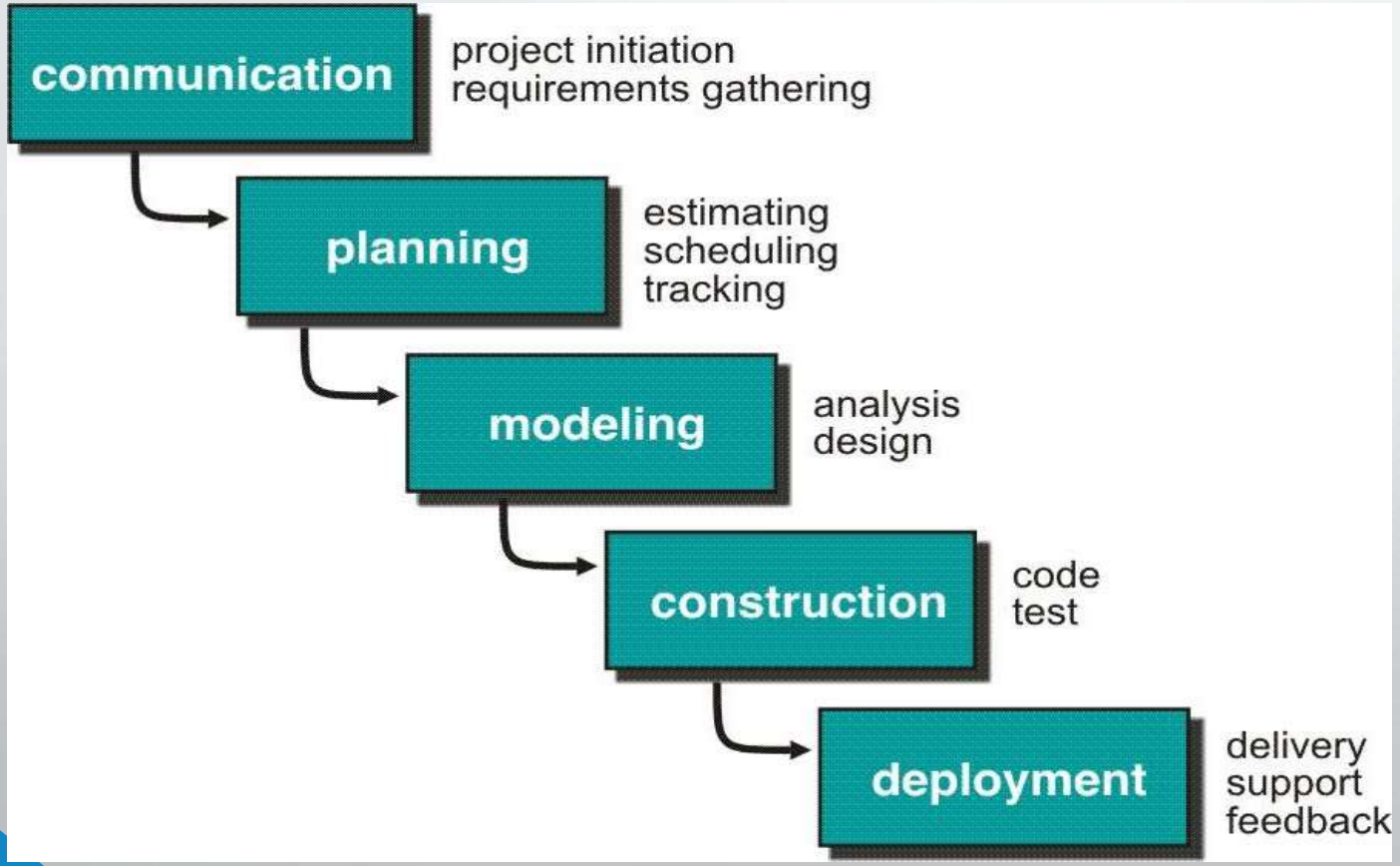
Prescriptive Model

- Prescriptive process models advocate an orderly approach to software engineering
  - Organize framework activities in a certain order
- Process framework activity with set of software engineering actions.
- Each action in terms of a task set that identifies the work to be accomplished to meet the goals.
- The resultant process model should be adapted to accommodate the nature of the specific project, people doing the work, and the work environment.
- Software engineer choose process framework that includes activities like;
  - Communication
  - Planning
  - Modeling
  - Construction
  - Deployment

Prescriptive Model

- Calling this model as "Prescribe" because it recommend a set of process elements, activities, action task, work product & quality.
- Each elements are inter related to one another (called workflow).

Waterfall Model or Classic Life Cycle



Waterfall Model or Classic Life Cycle

- Requirement Analysis and Definition: What - **The systems services, constraints and goals** are defined by customers with system users.
- Scheduling tracking -
  - **Assessing progress** against the project plan.
  - Require **action** to maintain schedule.
- System and Software Design: How –It establishes and **overall system architecture**. Software design involves fundamental system abstractions and their relationships.
- Integration and system testing: The individual program unit or **programs are integrated and tested** as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
- Operation and Maintenance: Normally this is the **longest phase of the software life cycle**. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life-cycle.

Limitations of the waterfall model

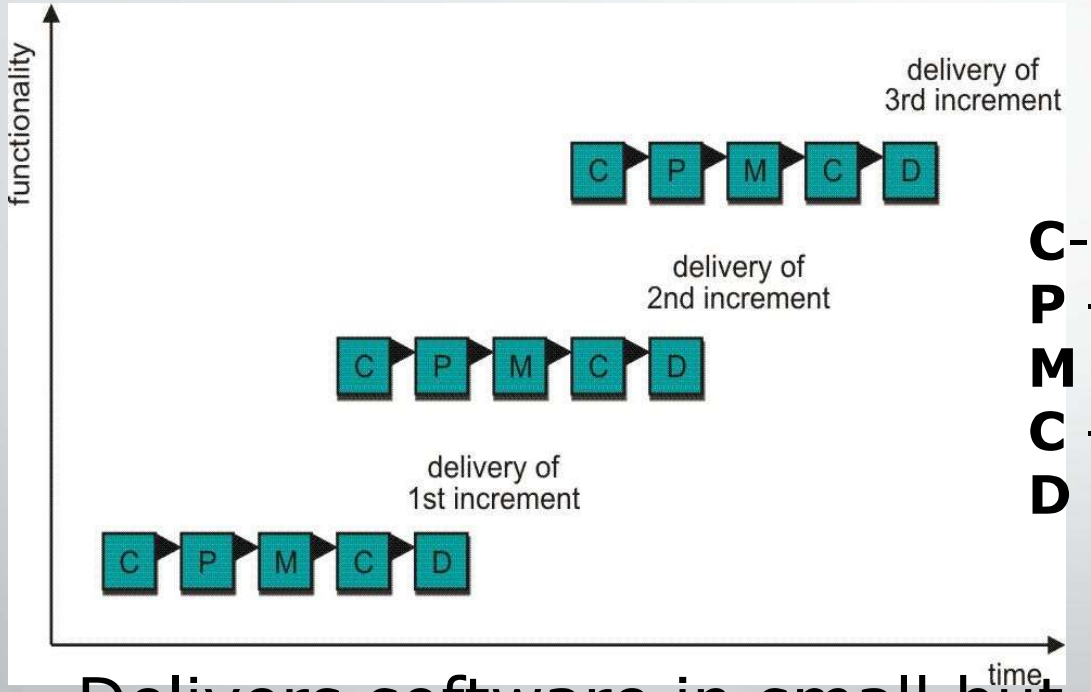
- The nature of the **requirements will not change very much** During development; during evolution
- The model implies that you should attempt to **complete a given stage before moving on to the next stage**
  - Does not account for the fact that **requirements constantly change.**
  - It also means that customers **can not use anything** until the entire system is complete.
- The model implies that once the product is finished, everything else is maintenance.
- **Surprises at the end are very expensive**
- **Some teams sit ideal** for other teams to finish
- Therefore, this model is only **appropriate when the requirements are well-understood and changes will be fairly limited** during the design process.

Limitations of the waterfall model

Problems:

1. Real projects are rarely follow the sequential model.
2. Difficult for the customer to state all the requirement explicitly.
3. Assumes patience from customer - working version of program will not available until programs not getting change fully.

Incremental Process Model



C- Communication
P - Planning
M – Modeling
C - Construction
D - Deployment

Delivers software in small but usable pieces, each piece builds on pieces already delivered

The Incremental Model

- 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.
- First Increment is often core product
  - Includes basic requirement
  - Many supplementary features (known & unknown) remain undelivered
- A plan of next increment is prepared
  - Modifications of the first increment
  - Additional features of the first increment
- It is particularly useful when enough staffing is not available for the whole project
- Increment can be planned to manage technical risks.
- Incremental model focus more on delivery of operation product with each increment.

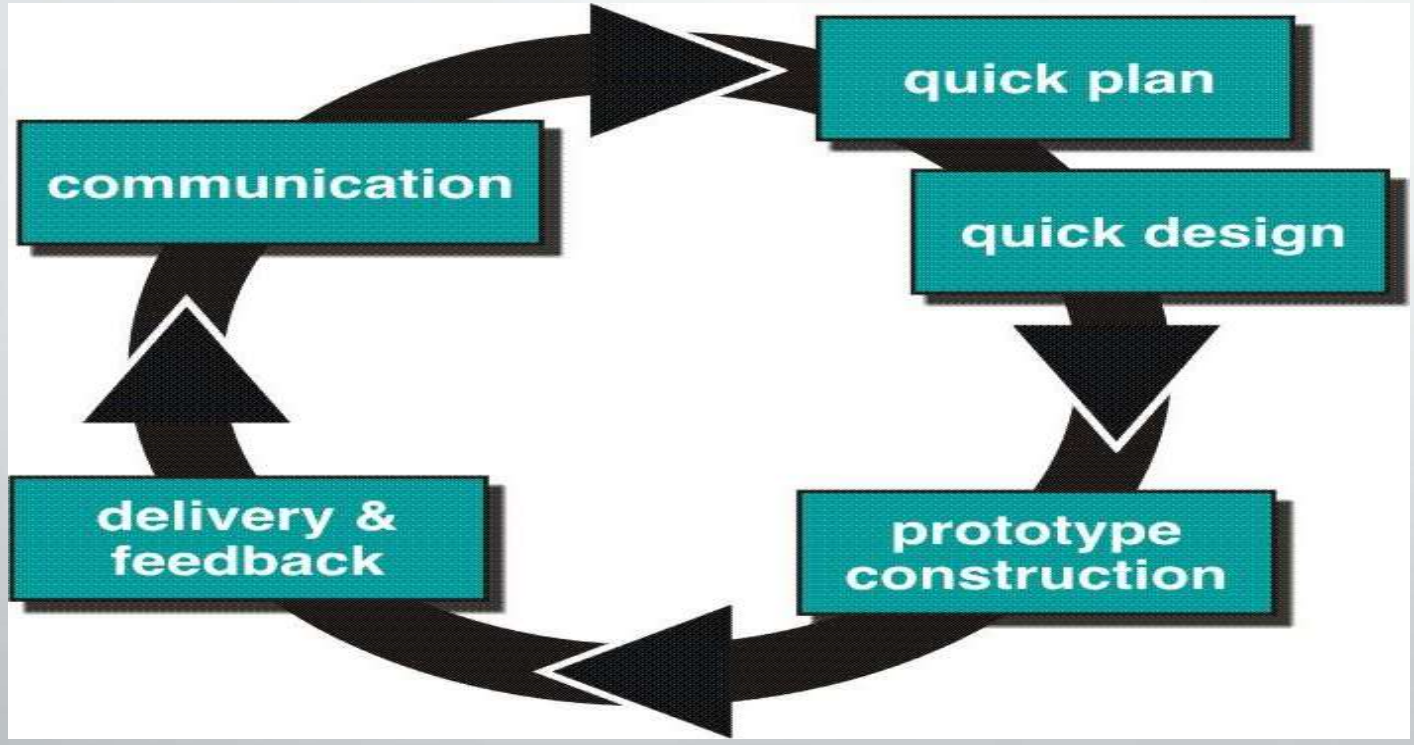
The Incremental Model

- 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.
- 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 more testing.

Evolutionary Process Model

- Produce an increasingly more complete version of the software with each iteration.
- Evolutionary Models are iterative.
- Evolutionary models are:
  - Prototyping
  - Spiral Model
  - Concurrent Development Model

Evolutionary Process Models : Prototyping



Software prototyping

- A prototype is an initial version of a system used to demonstrate concepts and try out design options.
- A prototype can be used in:
  - The requirements engineering process to help with requirements elicitation and validation;
  - In design processes to explore options and develop a UI design;
  - In the testing process to run back-to-back tests.

Benefits of prototyping

- Improved system usability.
- A closer match to users' real needs.
- Improved design quality.
- Improved maintainability.
- Reduced development effort.

Prototype development

- May be based on rapid prototyping languages or tools
- May involve leaving out functionality
  - Prototype should focus on areas of the product that are not well-understood;
  - Error checking and recovery may not be included in the prototype;
  - Focus on functional rather than non-functional requirements such as reliability and security

Throw-away prototypes

- Prototypes should be discarded after development as they are not a good basis for a production system:
  - It may be impossible to tune the system to meet non-functional requirements;
  - Prototypes are normally undocumented;
  - The prototype structure is usually degraded through rapid change;
  - The prototype probably will not meet normal organisational quality standards.

Prototyping cohesive

- **Best approach when**:
  - **Objectives defined by customer are general** but does not have details like input, processing, or output requirement.
  - Developer may be **unsure of the efficiency** of an algorithm, O.S., or the form that human machine interaction should take.
- It can be used as standalone process model.
- Model assist software engineer and customer to better understand what is to be built when requirement are fuzzy.
- Prototyping start with communication, between a customer and software engineer to define overall objective, identify requirements and make a boundary.
- Going ahead, planned quickly and modeling (software layout visible to the customers/end-user) occurs.
- Quick design leads to prototype construction.
- Prototype is deployed and evaluated by the customer/user.
- Feedback from customer/end user will refine requirement and that is how iteration occurs during prototype to satisfy the needs of the customer.

Prototyping (cont..)

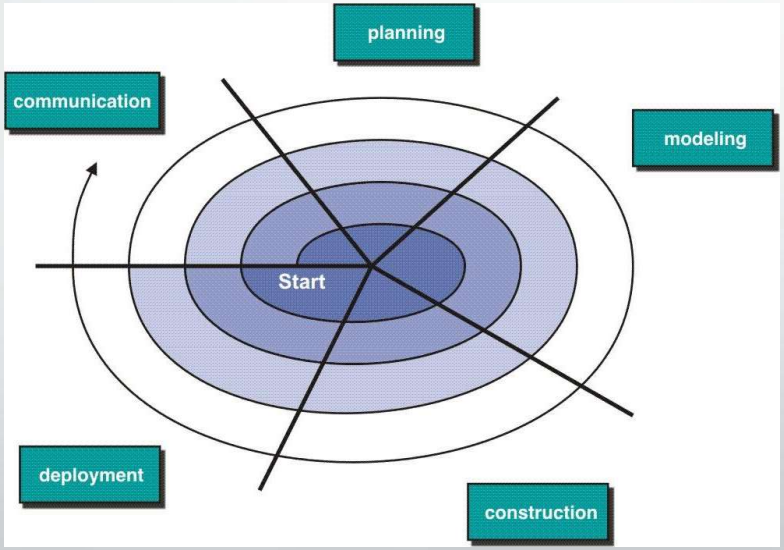
- Prototype can be serve as "the first system".
- Both customers and developers like the prototyping paradigm.
  - Customer/End user gets a feel for the actual system
  - Developer get to build something immediately.

Problem Areas:

- Customer cries foul and demand that "a few fixes" be applied to make the prototype a working product, due to that **software quality suffers** as a result.
- Developer often makes implementation in order to get a prototype working quickly without considering other factors in mind like OS, Programming language, etc.

Customer and developer both must be agree that the **prototype is built to serve as a mechanism for defining requirement**.

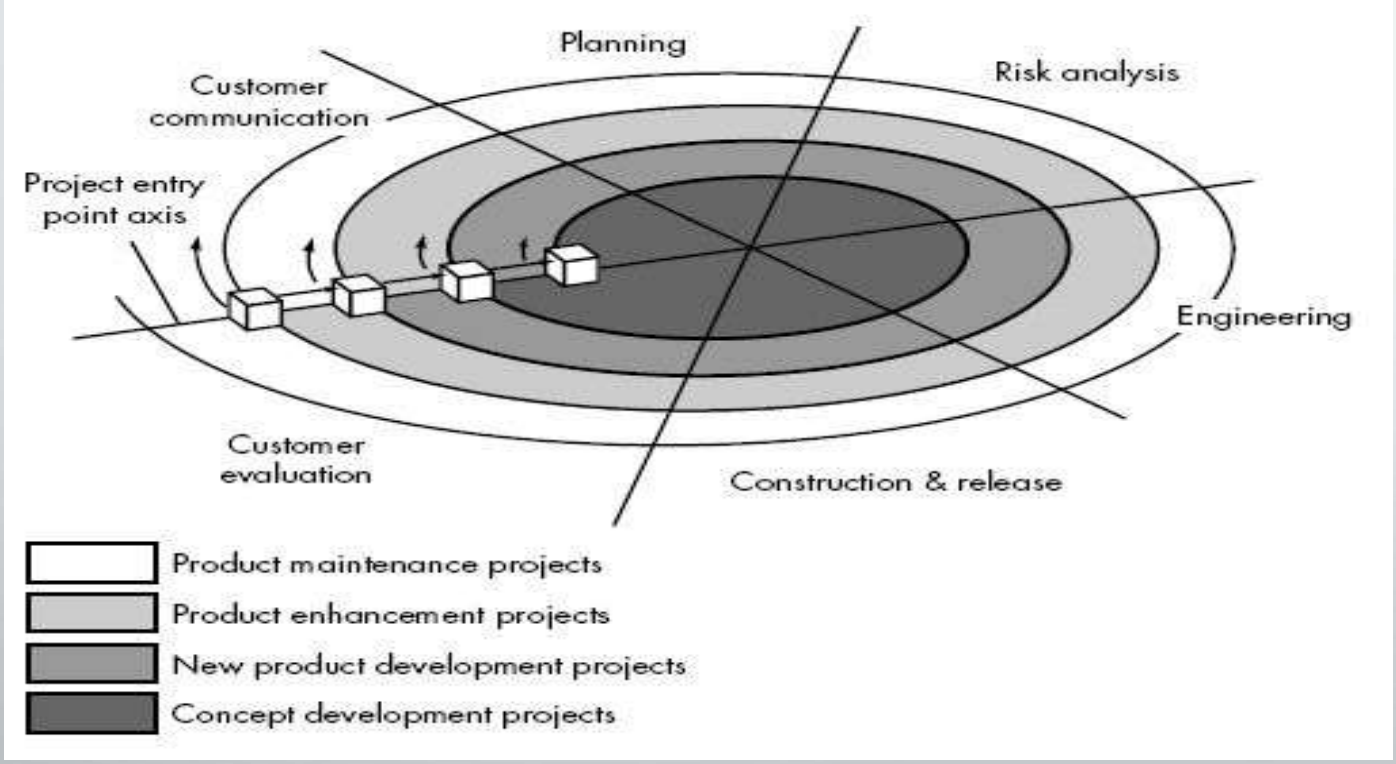
Evolutionary Model: Spiral Model



Spiral Model

- Couples iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model
  - It provide potential for rapid development of increasingly more complete version of the software.
  - Using spiral, software developed in as series of evolutionary release.
    - Early iteration, release might be on paper or prototype.
    - Later iteration, more complete version of software.
  - Divided into framework activities (C,P,M,C,D). Each activity represent one segment.
  - Evolutionary process begins in a clockwise direction, beginning at the center risk.
  - First circuit around the spiral might result in development of a product specification. Subsequently, develop a prototype and then progressively more sophisticated version of software.
  - Unlike other process models that end when software is delivered.

Spiral Model



Spiral Model (cont.)

Concept Development Project:

- Start at the core and continues for multiple iterations until it is complete.
- If concept is developed into an actual product, the process proceeds outward on the spiral.

New Product Development Project:

- New product will evolve through a number of iterations around the spiral.
- Later, a circuit around spiral might be used to represent a "Product Enhancement Project"

Product Enhancement Project:

- There are times when process is dormant or software team not developing new things but change is initiated, process start at appropriate entry point.

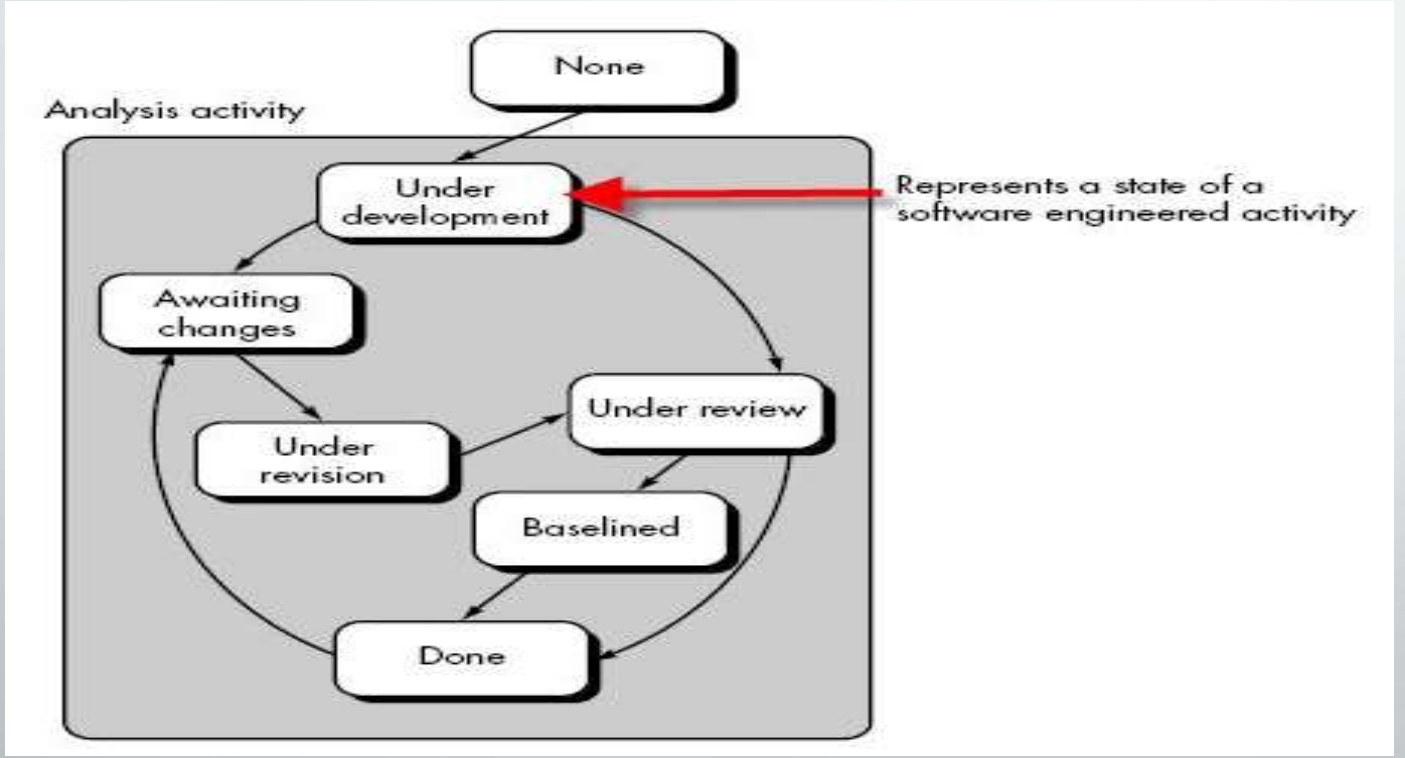
- Spiral models uses prototyping as a risk reduction mechanism but, more important, enables the developer to apply the prototyping approach at each stage in the evolution of the product.

- It maintains the systematic stepwise approach suggested by the classic life cycle but also incorporates it into an iterative framework activity.
- If risks cannot be resolved, project is immediately terminated

Problem Area:

- It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable.
- If a major risk is not uncovered and managed, problems will undoubtedly occur.

Concurrent Development Model



Concurrent Development Model

- It represented schematically as series of major technical activities, tasks, and their associated states.
- It is often more appropriate for system engineering projects where different engineering teams are involved.
- The activity-modeling may be in any one of the states for a given time.
- All activities exist concurrently but reside in different states.

Concurrent Development Model

E.g.

- The *analysis* activity (existed in the **none** state while initial customer communication was completed) now makes a transition into the **under development** state.
- *Analysis* activity moves from the **under development** state into the **awaiting changes** state only if customer indicates changes in requirements.
- Series of event will trigger transition from state to state.

E.g. During initial stage there was inconsistency in design which was uncovered. This will triggers the analysis action from the **Done** state into **Awaiting Changes** state.

Concurrent Development (Cont.)

- Visibility of current state of project
- It define network of activities
- Each activities, actions and tasks on the network exists simultaneously with other activities ,actions and tasks.
- Events generated at one point in the process network trigger transitions among the states.

Thank You….

22CSC51 - AGILE METHODOLOGIES

Prepared By,

Mr.N.Aravindhraj,

Assistant Professor

Department.of CSE

Kongu Engineering College

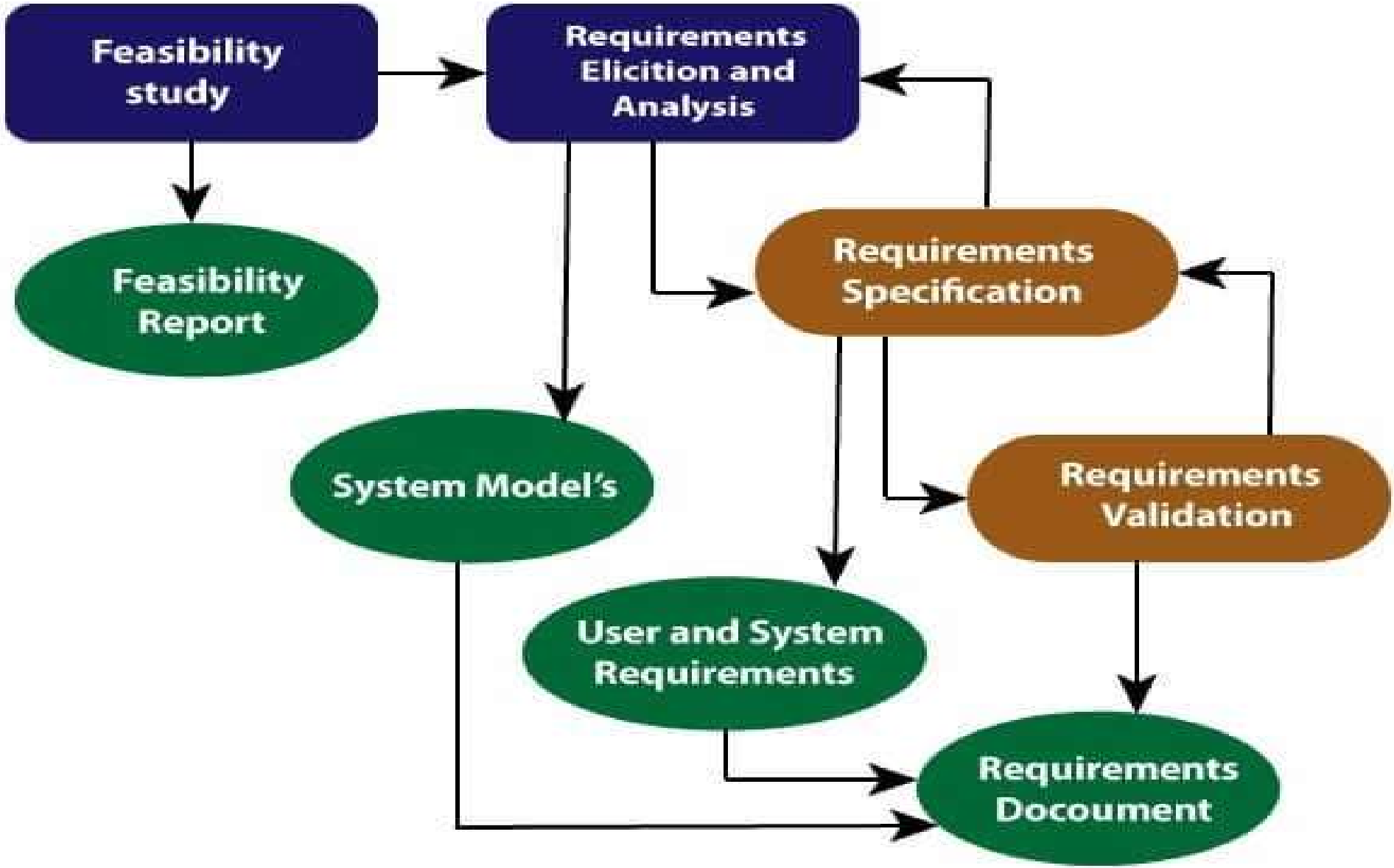
Requirement Engineering

- Requirements Engineering (RE) refers to the process of **defining, documenting, and maintaining requirements** in the engineering design process.

Requirement Engineering Process

It is a four-step process, which includes -

- Feasibility Study
- Requirement Elicitation and Analysis
- Software Requirement Specification
- Software Requirement Validation



Requirement Engineering Process

Feasibility Study

- The objective behind the feasibility study is to **create the reasons for developing the software** that is acceptable to users, flexible to change and conformable to established standards.

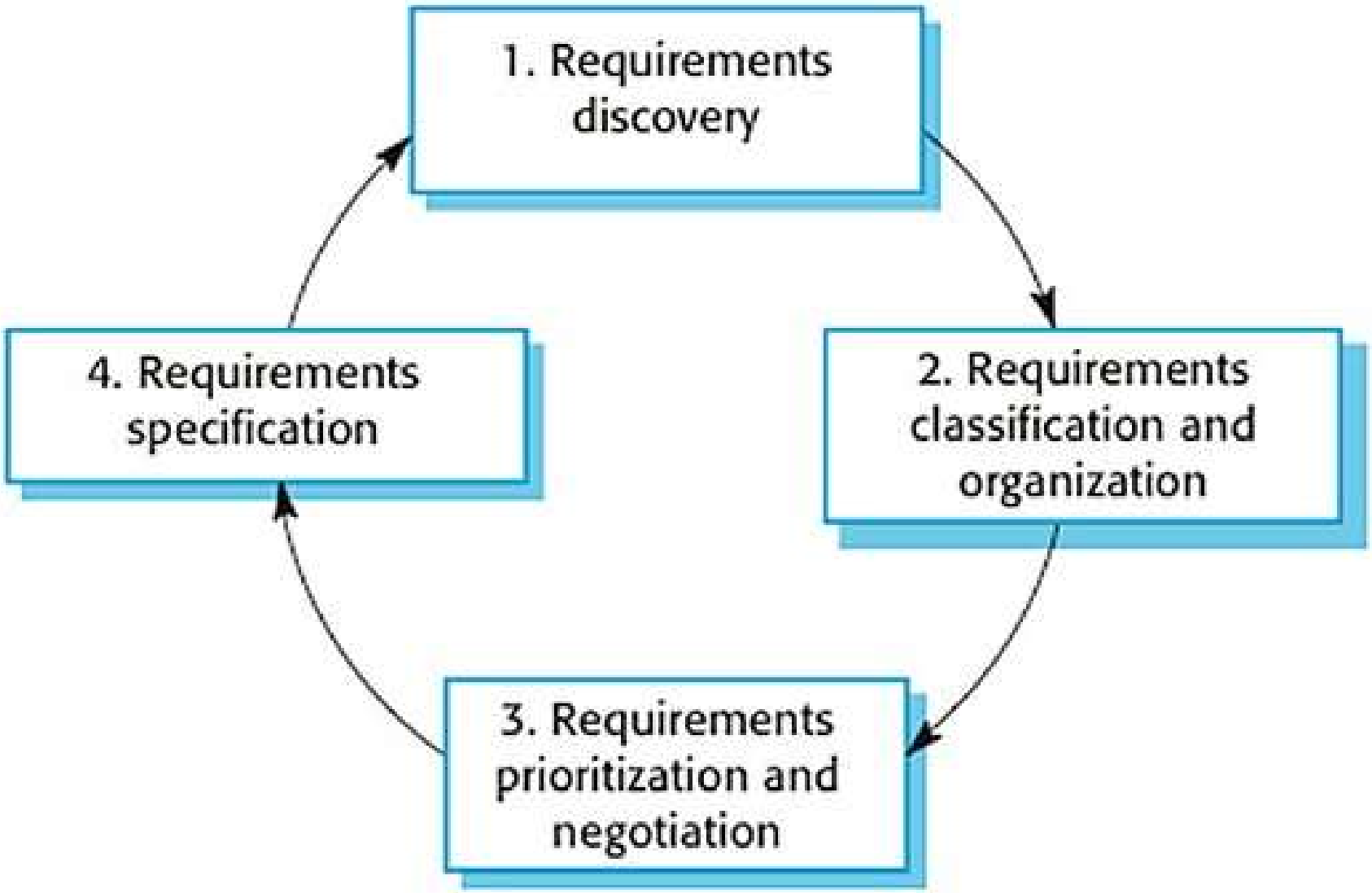
Types of Feasibility:

- **Technical Feasibility:** evaluates the **current technologies,** which are needed to accomplish customer requirements within the time and budget.
- **Operational Feasibility:** assesses the range in which the required software performs a series of levels to **solve business problems and customer requirements**.
- **Economic Feasibility:** Economic feasibility decides whether the necessary software can generate **financial profits for an organization.**

Requirement Elicitation and Analysis

- This is also known as the **gathering of requirements.** Here, requirements are identified with the help of **customers and existing systems processes,** if available.
- Analysis of requirements starts with requirement elicitation.
- The requirements are analyzed to identify **inconsistencies, defects, omission, etc.**

Requirement Elicitation and Analysis



Software Requirements

Broadly software requirements should be categorized in two categories:

- Functional Requirements
- Non Functional Requirements

Functional Requirements

Requirements, which are related to functional aspect of software fall into this category.

They define **functions and functionality** within and from the software system.

Examples -

- Search option given to **user to search from various invoices**.
- User should be able to **mail any report** to management.
- Users can be divided into groups and groups can be given separate rights.
- Should comply business rules and administrative functions.
- Software is developed keeping **downward compatibility** intact.

Non-Functional Requirements

Requirements, which are not related to functional aspect of software, fall into this category. They are **implicit or expected characteristics of software,** which users make assumption of.

Non-functional requirements include -

- **Security**
- **Logging**
- **Storage**
- **Configuration**
- **Performance**
- **Cost**
- **Interoperability**
- **Flexibility**
- **Disaster recovery**
- **Accessibility**

Requirement Elicitation and Analysis

- Requirement Elicitation Techniques
- Interviews
- Surveys
- Questionnaires
- Task analysis
- Domain Analysis
- Brainstorming
- Prototyping
- Observation

Requirements Analysis

- Requirements analysis or requirements engineering is a process used to **determine the needs and expectations of a new product**.
- It involves frequent **communication** with the **stakeholders and end-users** of the product to define expectations, resolve conflicts, and document all the key requirements.

Requirements Analysis Process

- A requirements analysis process involves the following steps:
  - Identify Key Stakeholders and End-Users
  - Capture Requirements
  - Categorize Requirements
  - Interpret and Record Requirements
  - Sign off

Requirements Analysis Process

Identify Key Stakeholders and End-Users:

- ✔ The first step of the requirements analysis process is to **identify key stakeholders** who are the main sponsors of the project.
- ✔ They will have the final say on what should be included in the scope of the project.
- ✔ Next, identify the **end-users of the product**. Since the product is intended to satisfy their needs, their inputs are equally important.

Requirements Analysis Process

Capture Requirements:

Ask each of the stakeholders and end-users their requirements for the new product. some of the requirements analysis techniques are,

1. **Hold One-on-One Interviews**
2. **Use Focus Groups**
3. **Utilize Use Cases**
4. **Build Prototypes**

Requirements Analysis Process

Categorize Requirements:

Since requirements can be of various types, they should be grouped to avoid confusion. Requirements are usually divided into four categories:

Functional Requirements - Functions the product is required to perform.

Technical Requirements - Technical issues to be considered for the successful implementation of the product.

Transitional Requirements - Steps required to implement a new product smoothly.

Operational Requirements - Operations to be carried out in the backend for proper functioning of the product.

Requirements Analysis Process

Interpret and Record Requirements

Once the requirements are **categorized**, determine which requirements are actually achievable and document each one of them. some techniques to analyze and interpret requirements are,

Define Requirements Precisely

Prioritize Requirements

Carry Out an Impact Analysis

Resolve Conflicts

Analyze Feasibility

Requirements Analysis Process

Sign off

- Once a final decision is made on the requirements, **ensure that you get a signed agreement from the key stakeholders**.
- This is done to ensure that there are **no changes or uncontrolled growth** in the scope of the project.

Software Requirement Specification

- Software requirement specification is a kind of document which is created by a **software analyst** after the requirements collected from the various sources.
- The requirement received by the customer written in **ordinary language.**
- It is the job of the **analyst to write the requirement in technical language** so that they can be understood and beneficial by the development team.

Software Requirement Specification

The models used at this stage include

- **Data Flow Diagrams:** Data Flow Diagrams (DFDs) are used widely for **modeling the requirements.** DFD shows the flow of data through a system.
- **Data Dictionaries:** Data Dictionaries are **simply repositories** to store information about all data items defined in DFDs.
- **Entity-Relationship Diagrams:** Another tool for requirement specification is the **entity-relationship diagram, often called an "E-R diagram."**
  - It is a detailed logical representation of the data for the organization and uses three main constructs **i.e. data entities, relationships, and their associated attributes.**

Software Requirement Validation

- After requirement specifications developed, the requirements discussed in this **document are validated.**
- Requirements can be the check against the following conditions -

✔ **If they can practically implement**

✔ **If they are correct and as per the functionality and specially of software**

✔ **If there are any ambiguities**

✔ **If they are full**

✔ **If they can describe**

Software Requirement Validation

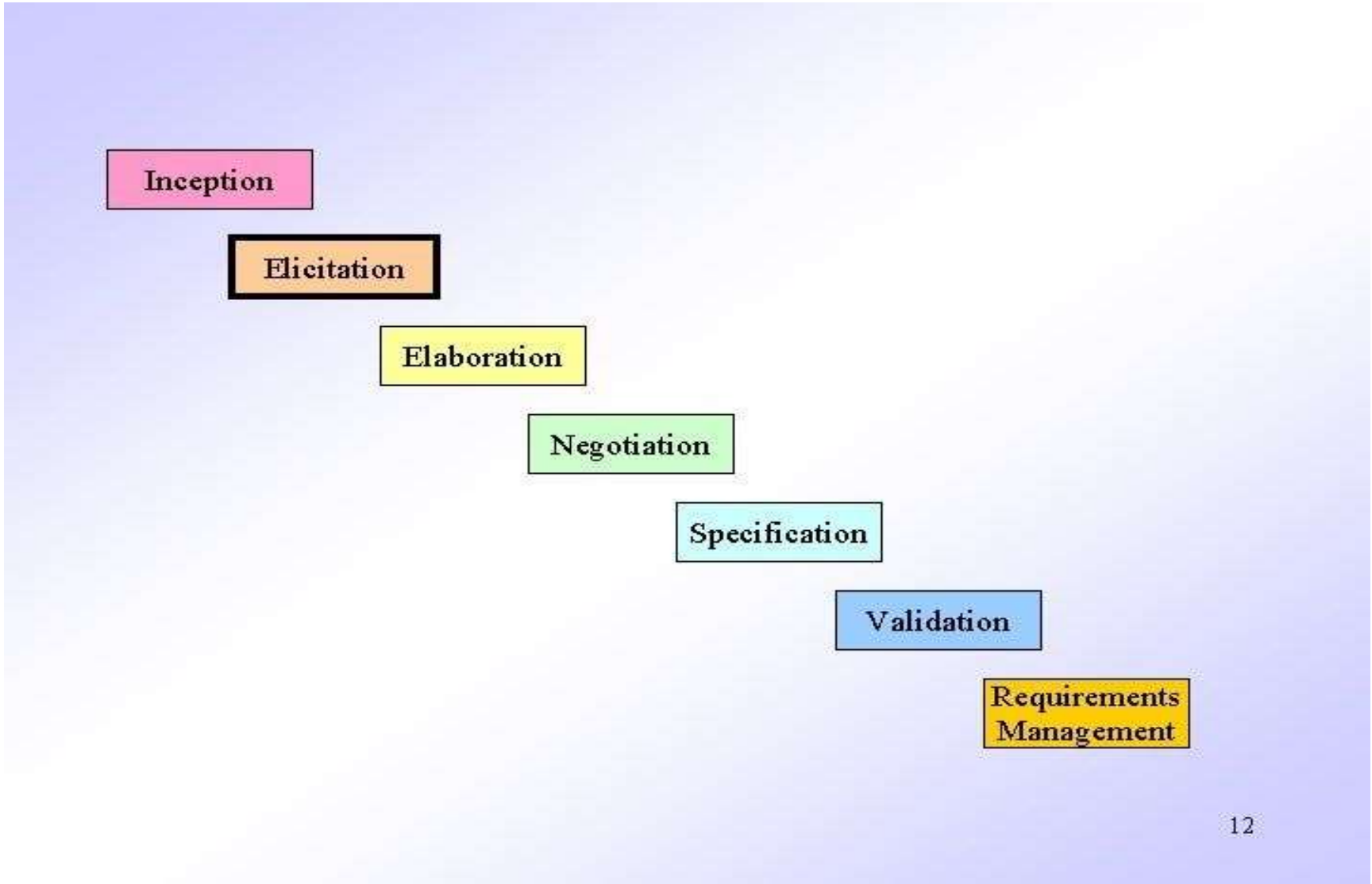
Requirements Validation Techniques

- **Requirements reviews/inspections:** systematic manual analysis of the requirements.
- **Prototyping:** Using an executable model of the system to check requirements.
- **Test-case generation:** Developing tests for requirements to check testability.
- **Automated consistency analysis:** checking for the consistency of structured requirements descriptions.

Software Requirement Management

- Requirement management is the process of **managing changing requirements** during the requirements engineering process and system development.

Requirement Engineering task



Requirement Engineering task

- ***Inception*** —Establish a basic understanding of the problem and the nature of the solution.
- ***Elicitation*** —Draw out the requirements from stakeholders.
- ***Elaboration*** —Create an analysis model that represents information, functional, and behavioral aspects of the requirements.
- ***Negotiation*** —Agree on a deliverable system that is realistic for developers and customers.
- ***Specification*** —Describe the requirements formally or informally.
- ***Validation***—*Review* the requirement specification for errors, ambiguities, omissions, and conflicts.
- ***Requirements management***—Manage changing requirements.

Inception Task

The requirement engineer ***ask a set of question*** to establish

- basic understanding of the problem
- the people who want a solution
- the nature of the solution that is desired, and
- the effectiveness of preliminary communication and collaboration
- between the customer and the developer

Through out the question , requirement engineer needs to

- **Identify the stakeholder**
- **Recognize multiple view points**
- **Work towards collaboration**
- **Break the ice and initiate the communication**

Elicitation task

Ask the customer, the users and others

- what the objectives for the system or product are,
- what is to be accomplished,
- how the system or product fits into the needs of the business.
- How the system or product to be used on day to day basis

Following are the problems that are encountered during elicitation

- **Problem of scope**
- **Problem of understanding**
- **Problems of volatility**

To overcome the above problem , we must approach the ***requirement gathering in an organized way***

Elaboration

- The information obtained from the customer during inception and elicitation is expanded and refined it
- Elaboration focuses on developing a refined technical model of software functions , features, and constraints
- It is an analysis modeling task
  - **Use cases are developed**
  - **Domain classes are identified**
  - **State machine diagrams are used**

Negotiation

- Customers and users are ask for more than can be achieved ,given limited business resources
- It is common for different customers or users to propose conflicting requirements
- Reconciling the conflict through a process of negotiation
- Customers, users and other stakeholders are asked

  To rank/prioritizes the requirement

  Assesses their cost

  Risk

  Addresses internal conflicts
- So that requirements are eliminated, combined / modified both(Developer and customer) achieve some measure of satisfaction

Specification

Specification*— *"different things to different people" can be any one (or more) of the following:

- A written document
- A set of models
- A formal mathematical model
- A collection of user scenarios (use-cases)
- A prototype

Validation

- Product produced are assessed for quality during validation
- Requirement validation examines the specification to ensure all the SW requirements stated clearly, that inconsistencies, omissions and error have been detected and corrected
- The work product conform to the standards established for the process

***Validation*—a review mechanism that looks for**

- **errors in content or interpretation**
- **areas where clarification may be required**
- **missing information**
- **inconsistencies (a major problem when large products or systems are engineered)**
- **conflicting or unrealistic (unachievable) requirements**

Requirement management

It is a **set of activities** that help the **project team identify, control and track the requirements** and changes to requirement at any time as the project proceeds

22CSC51 - AGILE METHODOLOGIES

Prepared By,

Mr.N.Aravindhraj,

Assistant Professor

Department.of CSE

Kongu Engineering College

Requirement Modeling

- It is the process of Identifying a requirement the software solution must met in order to be successful.
- It is divided into several stages
  - **Scenario Based Modeling**
  - **Flow Oriented Modeling**
  - **Class Based Modeling**
  - **Behavioral Modeling**

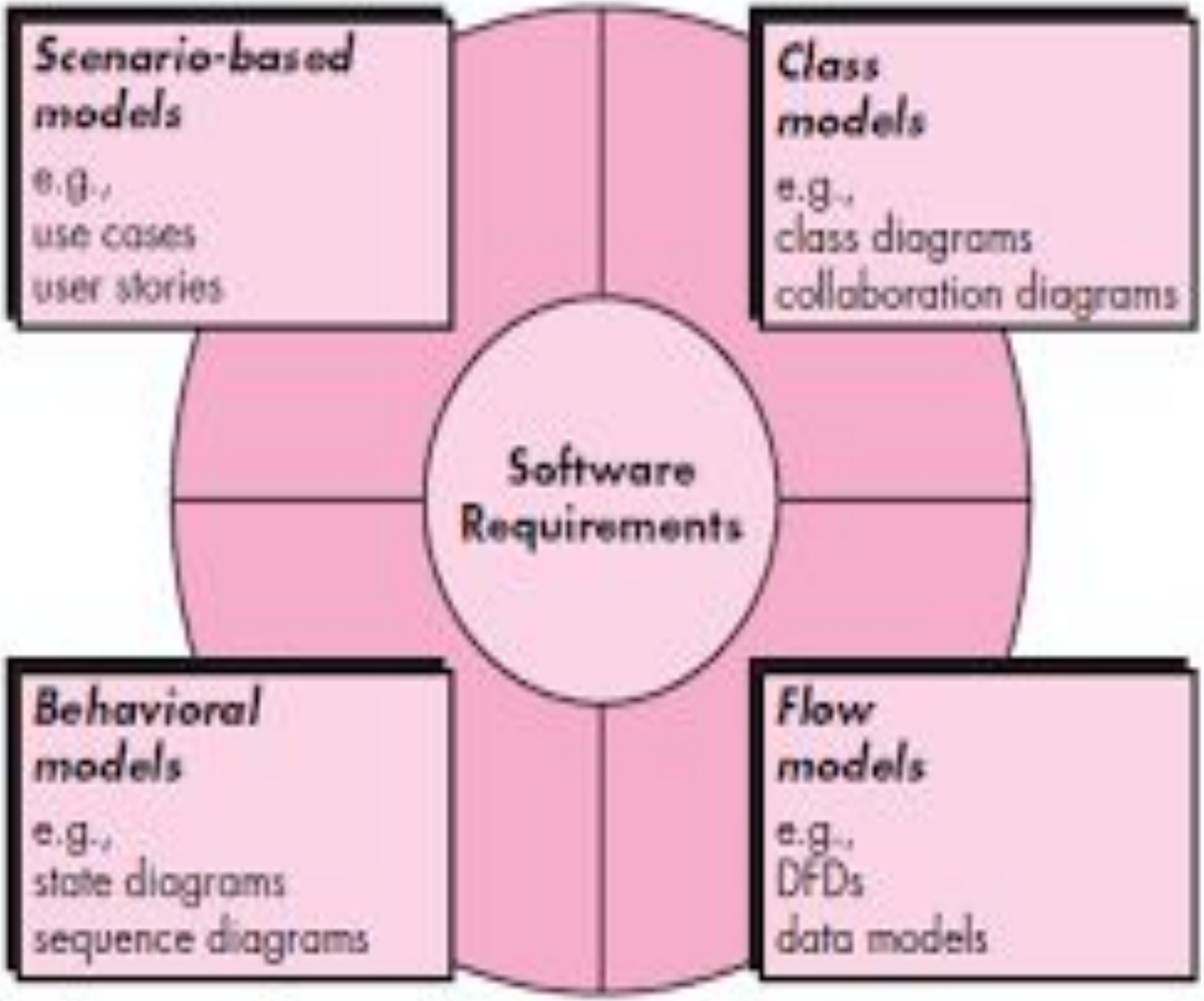


Fig 6.3: Elements of Analysis model

Scenario Based Modeling

- Invented by **Ivan Jacobson**.
- Modeling means **producing Diagrams**.
- It can be drawn by UML (Unified Modeling Language)
- Scenario based Modeling are represented in **uses or Activity diagrams.**
- **Use cases** are simply an aid to defining what exists outside the system (actors) and what should be performed by the system.
- It identifies the **possible use cases** for the system and produces the use case diagrams.
- A scenario that describes a thread of usage for a system.
- **Actor represents roles** of people or device in system functions.

Scenario Based Modeling

Use case:

Creating a Preliminary Use Case

What to write about,

How much to write about it,

How detailed to make your description and

How to organize the description?

Refining a Preliminary Use Case

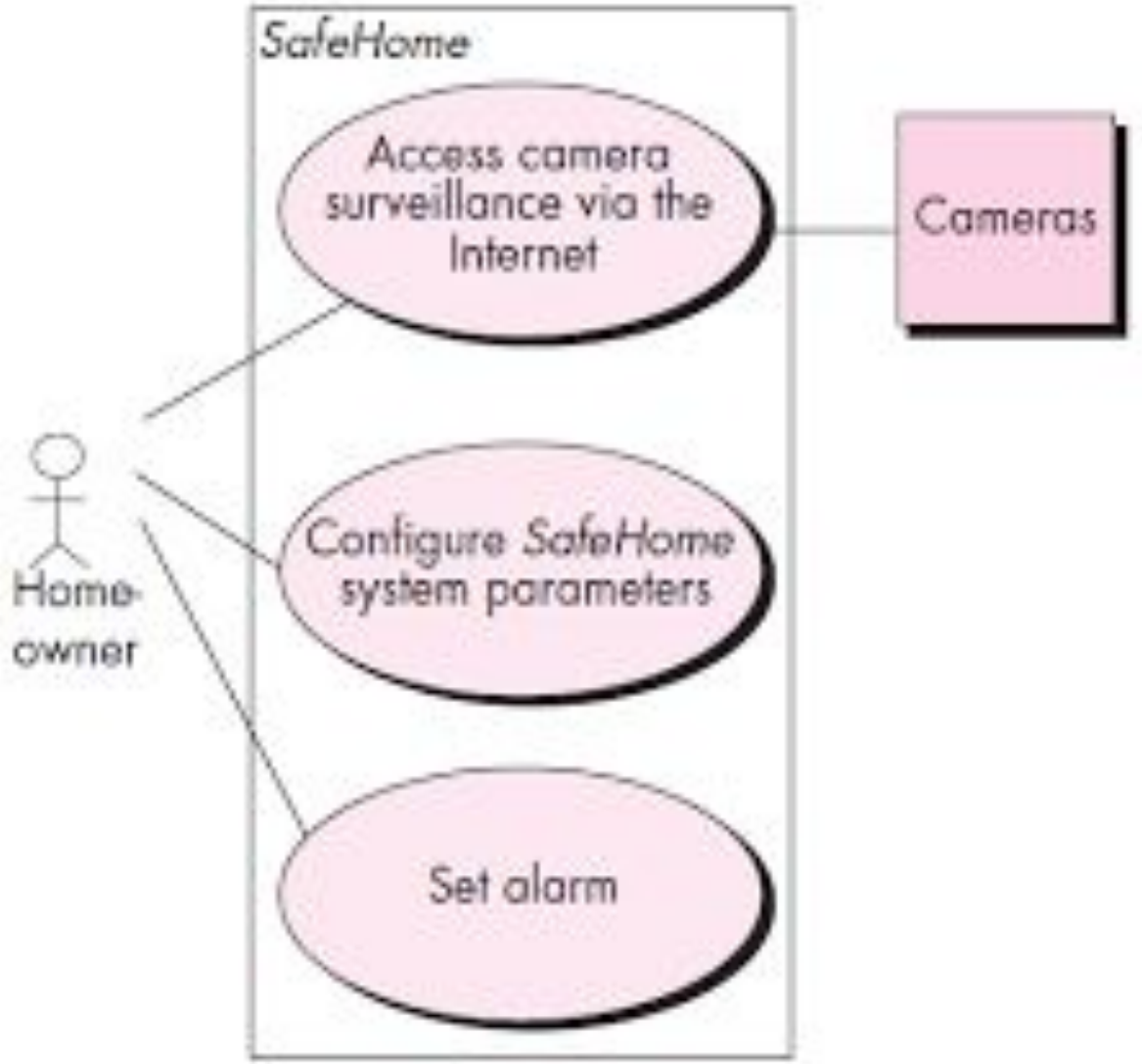
Can the actor take some other action at this point?

Is it possible that the actor will encounter an error condition at some point? If so, what?

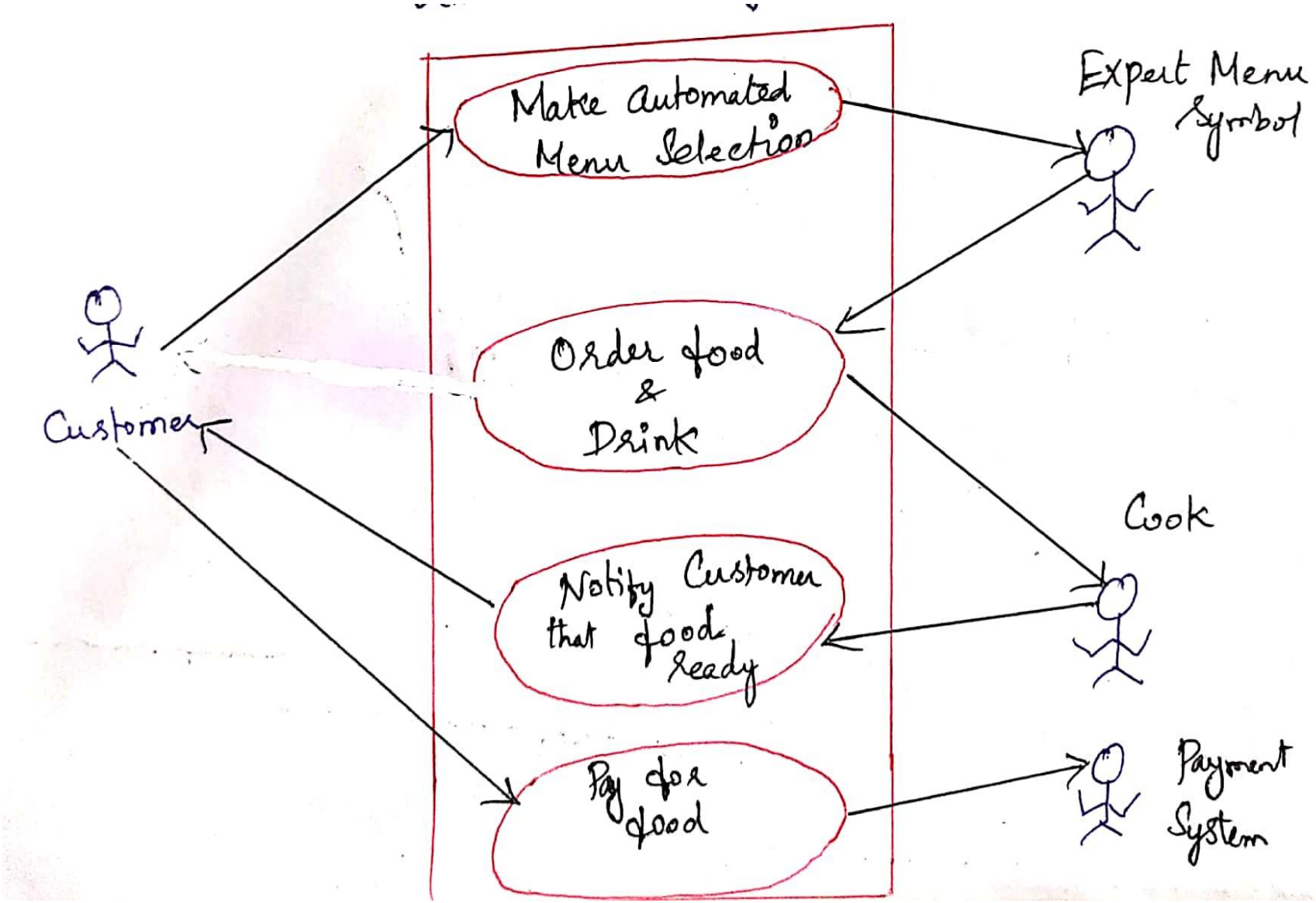
Is it possible that the actor will encounter some other behavior at some point? If so, what?

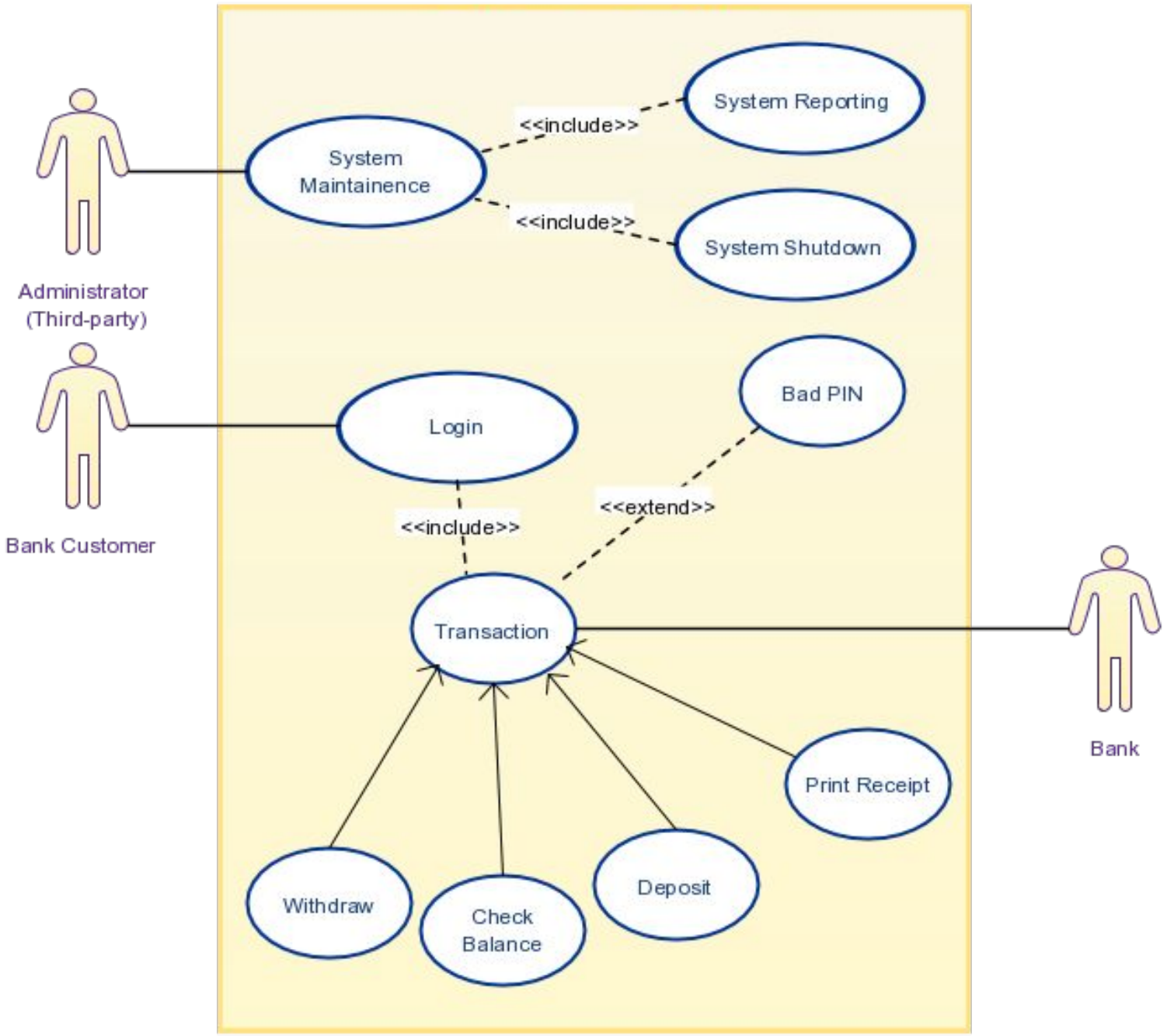
Writing a Formal Use Case

Preliminary use case diagram for safe home



Usecase Diagram Food ordering




Simple ATM Machine System
System Reporting
<<include>>
System Maintainence
<<include>>
System Shutdown
Administrator
(Third-party)
Bad PIN
Login
<<extend>>
<<include>>
Bank Customer
Transaction
Bank
Print Receipt
Withdraw
Check Balance
Deposit

Writing a Formal Use case

Use Case Template for Surveillance

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Iteration: 2, last modification: January 14 by V. Raman.

Primary actor: Homeowner.

Goal in context: To view output of camera placed throughout the house from any remote location via the Internet.

Preconditions: System must be fully configured; appropriate user ID and passwords must be obtained.

Trigger: The homeowner decides to take a look inside the house while away.

Scenario:

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Exceptions:

1. ID or passwords are incorrect or not recognized—see use case **Validate ID and passwords.**
2. Surveillance function not configured for this system—system displays appropriate error message; see use case **Configure surveillance function.**
3. Homeowner selects "View thumbnail snapshots for all camera"—see use case **View thumbnail snapshots for all cameras.**
4. A floor plan is not available or has not been configured—display appropriate error message and see use case **Configure floor plan.**
5. An alarm condition is encountered—see use case **Alarm condition encountered.**

Priority: Moderate priority, to be implemented after basic functions

When available: Third increment.

Frequency of use: Moderate frequency.

Channel to actor: Via PC-based browser and Internet connection.

Secondary actors: System administrator, cameras.

Channels to secondary actors:

1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

Open issues:

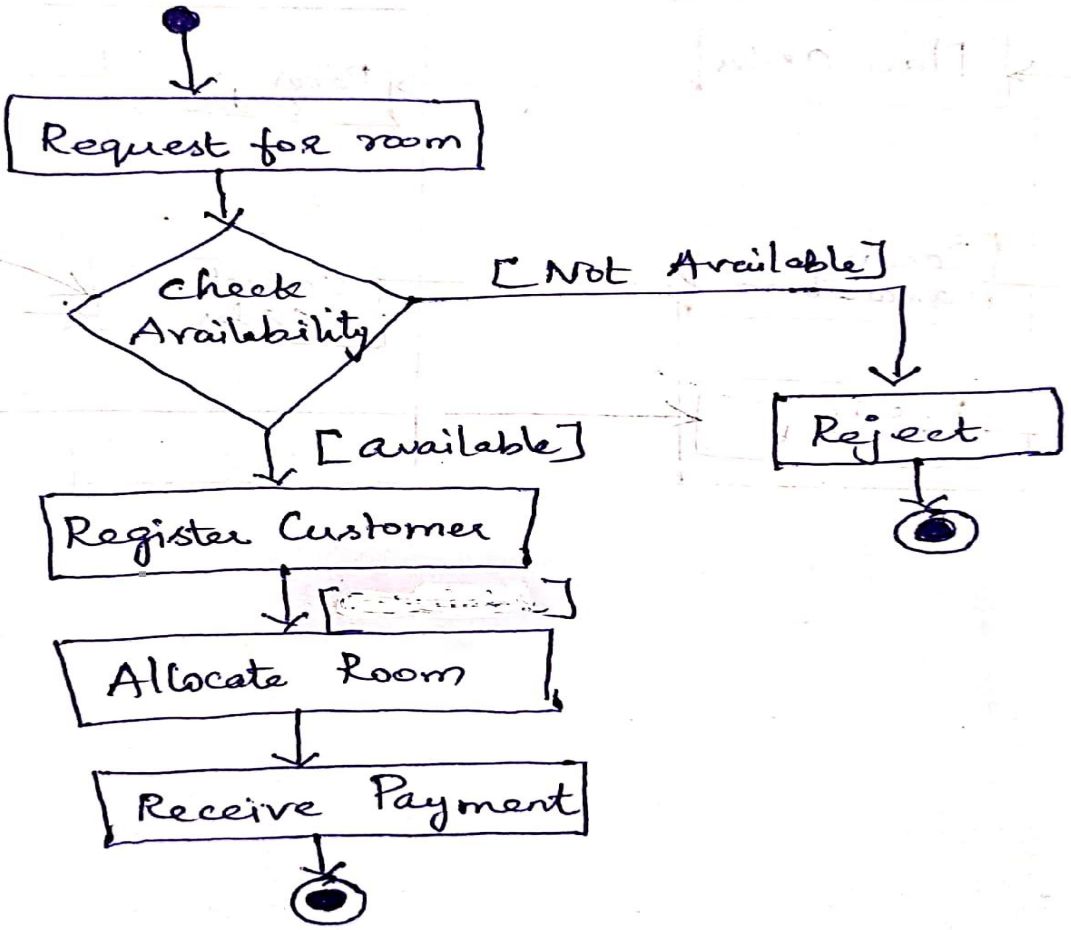
1. What mechanisms protect unauthorized use of this capability by employees of *SafeHome Products*?
2. Is security sufficient? Hacking into this feature would represent a major invasion of privacy.
3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
4. Will we develop a capability to provide video at a higher frames-per-second rate when high-bandwidth connections are available?

Scenario Based Modeling

Activity Diagram

- Graphical representation of the flow of interaction within a specific scenario
- **Rounded rectangles** to imply a specific system function,
- **Arrows** to represent flow through the system,
- **Decision diamonds** to depict a branching decision and
- **Solid horizontal** lines to indicate that parallel activities are occurring

Activity Diagram Room Booking



Activity Diagram

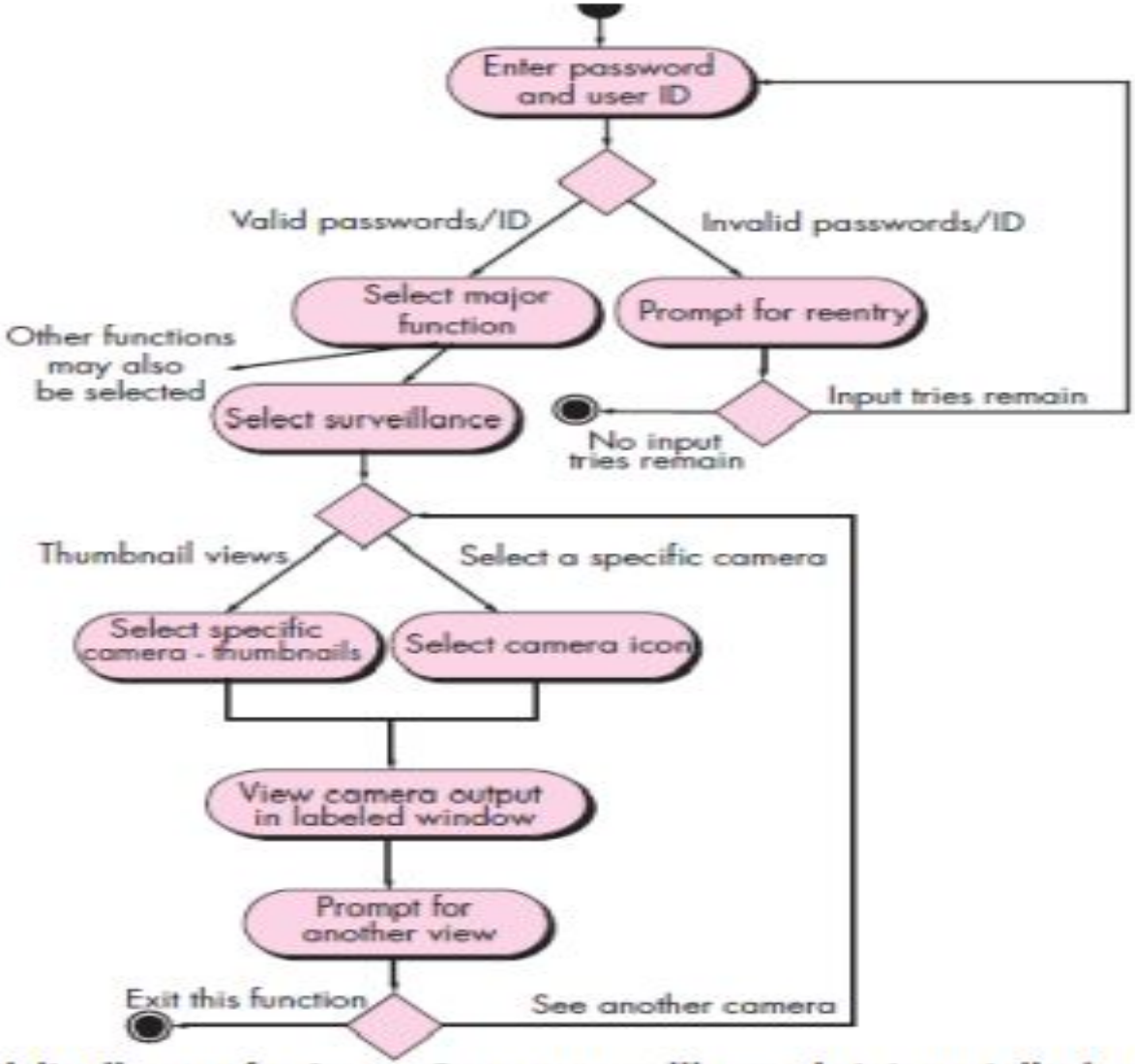


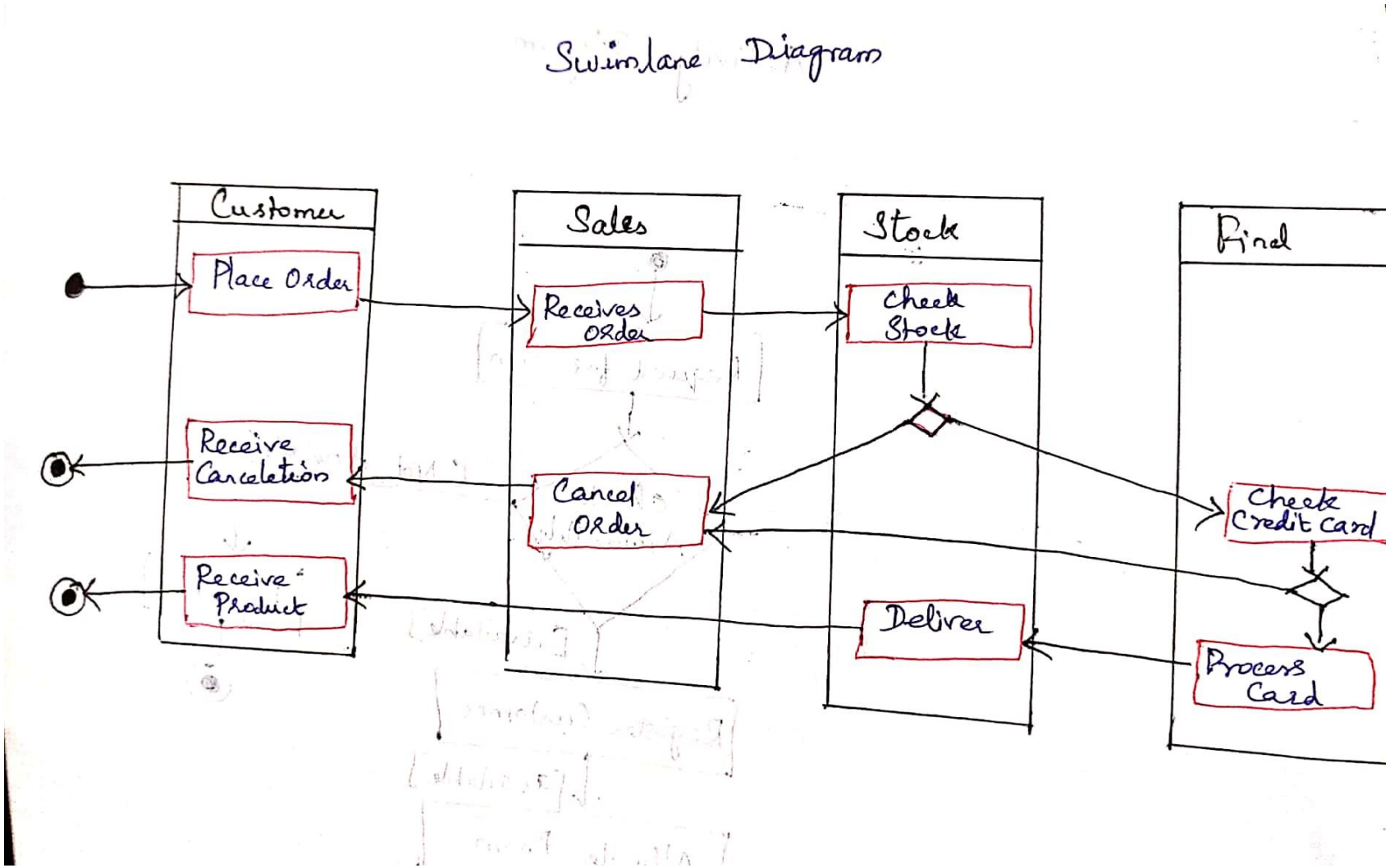
Fig 6.5: Activity diagram for Access Camera surveillance via Internet display cams

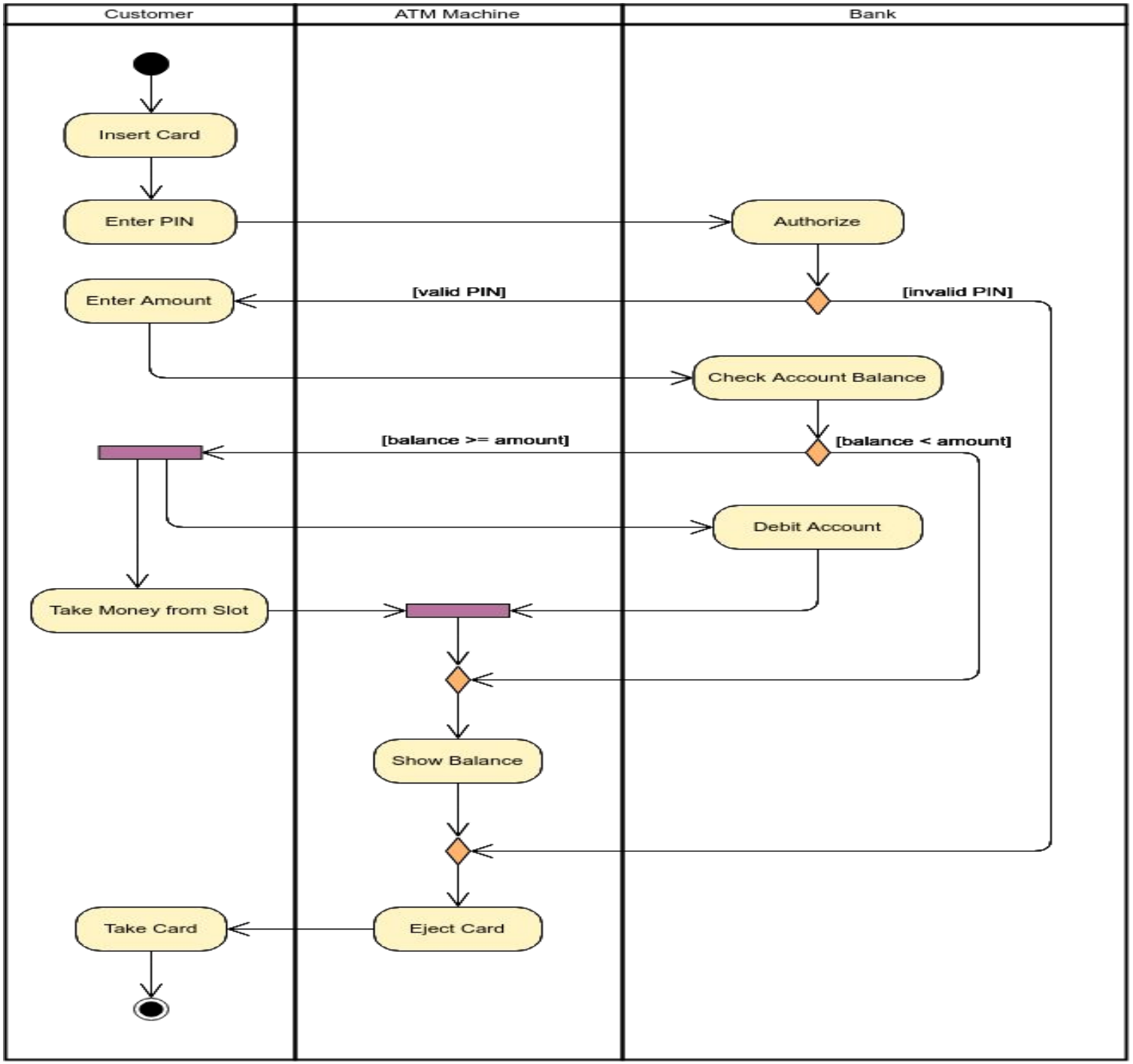
Scenario Based Modeling

Swimlane Diagrams:

- Represent the **flow of activities** described by the use case
- At the same time indicate **which actor or analysis class** has responsibility for the action described by an activity rectangle.
- **Responsibilities** are represented as parallel segments that divide the diagram **vertically**

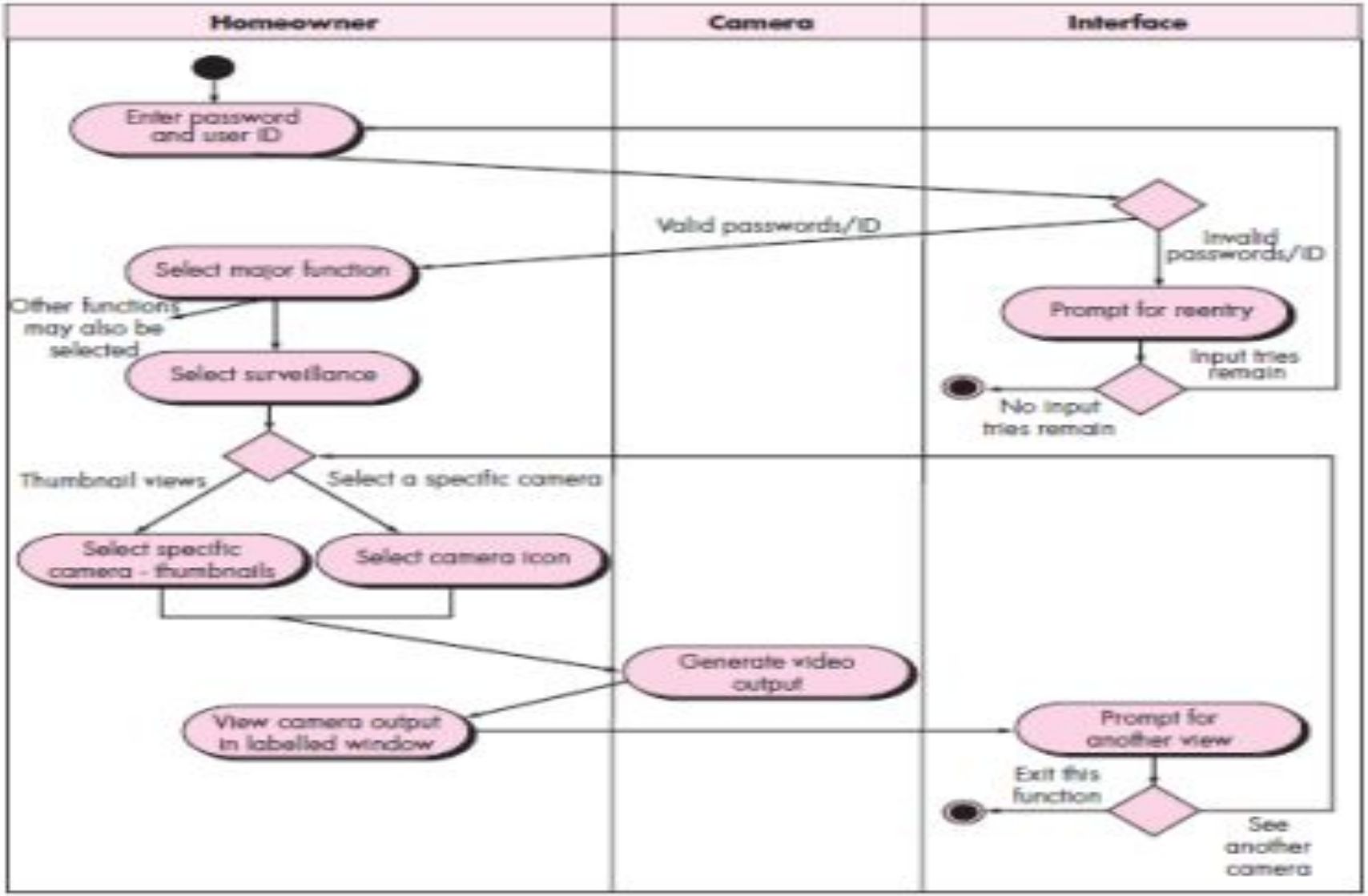
Swimlane Diagram




Customer
ATM Machine
Bank
Insert Card
Enter PIN
Authorize
Enter Amount
[valid PIN]
[invalid PIN]
Check Account Balance
[balance >= amount]
[balance < amount]
Debit Account
Take Money from Slot
Show Balance
Take Card
Eject Card

Swimlane Diagrams

Fig 6.6: Swimlane diagram for Access camera surveillance via the Internet—display camera views function



Class Based Modeling

- Class-based modeling takes the **use case and extracts from it the classes, attributes, and operations** the application will use. Like all modeling stages, the **end result of class-based modeling is most often a diagram or series of diagrams**, most frequently created using UML.
- Class-based modeling represents the **objects** that the system will manipulate, the operations (also called methods or services) that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined.
- The elements of a class-based model include **classes and objects, attributes, operations, class responsibility-collaborator (CRC) models, collaboration diagrams, and packages.**

Classes in UML Diagrams

- An abstraction which describes a collection of objects sharing some commonalties
- Syntax
  - Name: noun, singular
    - centered, bold, first letter capitalized
  - Attribute
    - left justified, lower cases
  - Operations
  - Visibility
    - \+ public
    - \- private
    - # protected

Class Based Modeling

- Identifying Analysis Classes
- Specifying Attributes
- Defining Operations
- Class Responsibility Collaborator (CRC) Modeling
- Associations and Dependencies
- Analysis Packages

Identifying Analysis Classes

- If you look around a room, there is a set of physical objects that can be easily identified, classified, and defined (in terms of attributes and operations).
- But when you **"look around"** the problem space of a software application, **the classes (and objects) may be more difficult to comprehend.**
- We can begin to identify classes by examining the **usage scenarios** developed as part of the requirements model

Identifying Analysis Classes

Analysis classes manifest themselves in one of the following ways:

- **External entities** (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- **Things** (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- **Occurrences or events** (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- **Roles** (e.g., manager, engineer, salesperson) played by people who interact with the system.
- **Organizational units** (e.g., division, group, team) that are relevant to an application.
- **Places** (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- **Structures** (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

Example.,

The **SafeHome security** function enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through the Internet, a PC, or a control panel.

Potential Class	General Classification
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

Specifying Attributes

- Attributes are the **set of Data Objects** that fully define the class within the context of the problem.
- Attributes describes the **class that has been selected for inclusion** in the requirement model.

To illustrate, we consider the **System** class defined for *SafeHome*. A homeowner can configure the security function to reflect sensor information, alarm response information, activation/deactivation information, identification information, and so forth. We can represent these composite data items in the following manner:

identification information = system ID + verification phone number + system status

alarm response information = delay time + telephone number

activation/deactivation information = master password + number of allowable tries + temporary password

Defining Operations

- Operations **define the behavior of an object**. Although many different types of operations exist, they can generally be divided into **four broad categories**:

(1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting),

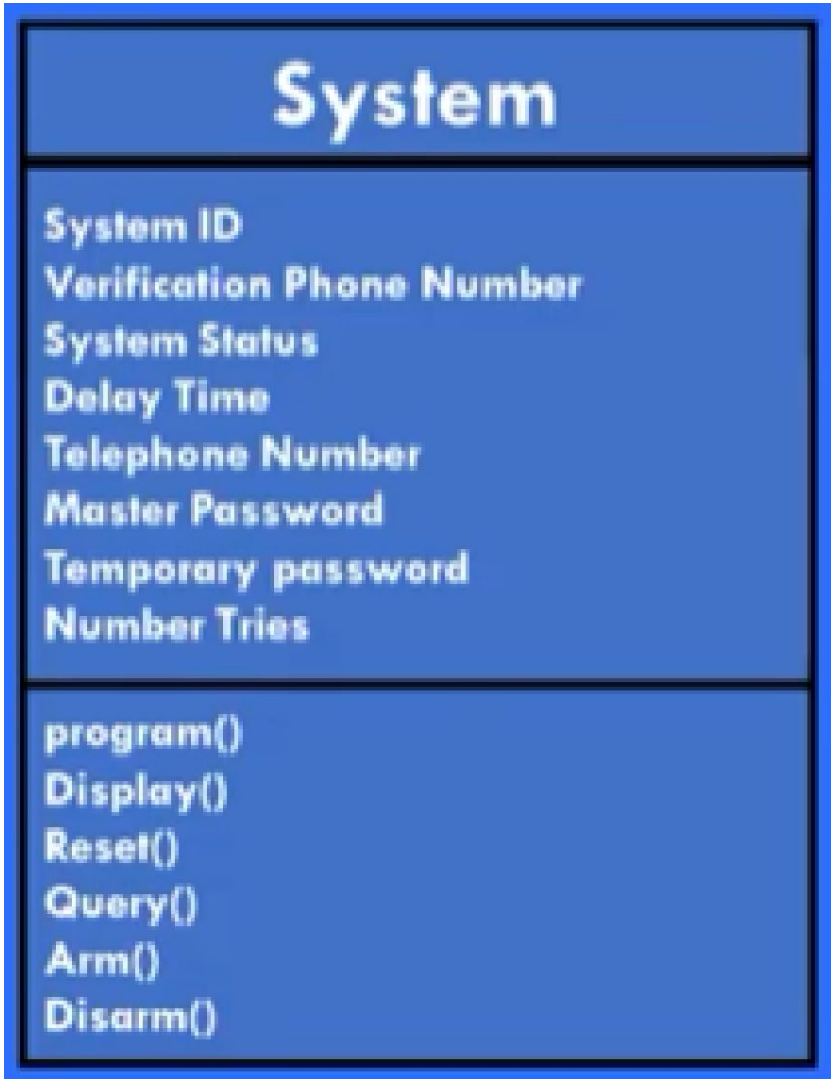
(2) operations that perform a **computation**,

(3) operations that inquire about the **state of an object**,

(4) operations that monitor an object for the occurrence of a controlling event.

- These functions are accomplished by operating on attributes and/or associations

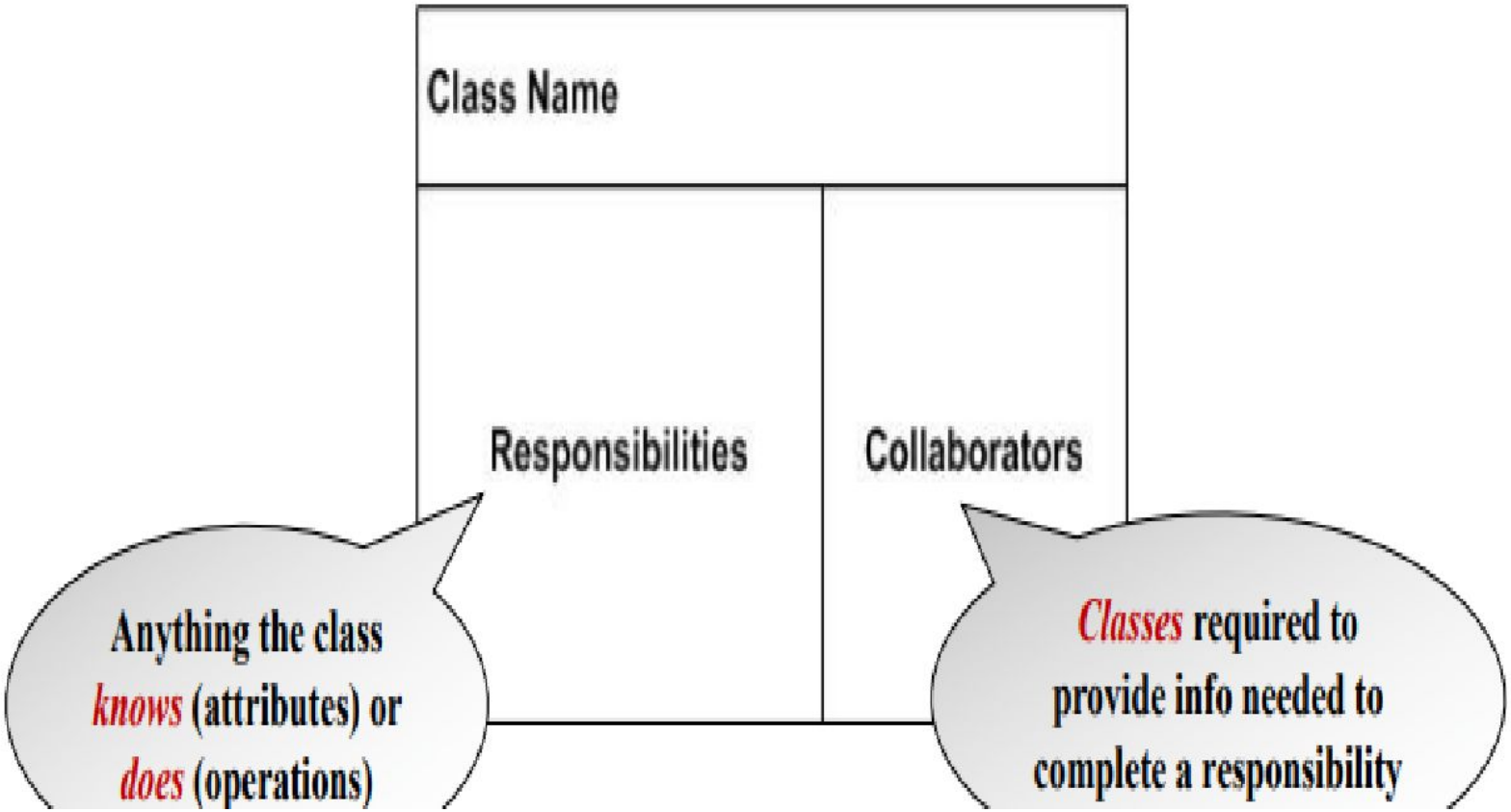
Defining Operations



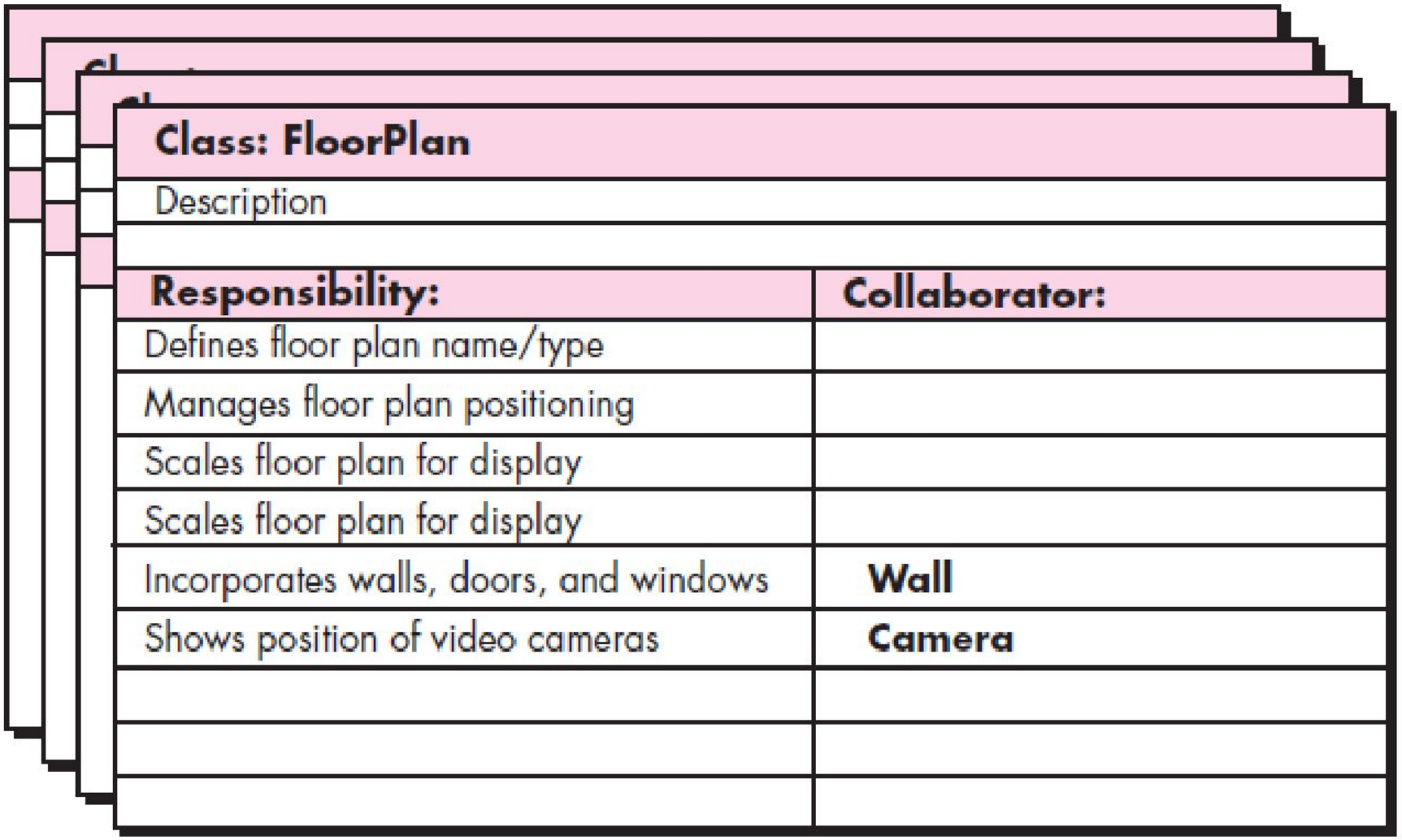
Class-Responsibility-Collaborator (CRC)

- Class-responsibility-collaborator (CRC) modeling [Wir90] provides a simple means for **identifying and organizing the classes** that are relevant to system or product requirements.
- Used to identify and organize classes
- A CRC model is really a **collection of standard index cards that represent classes.**
- The cards are divided into **three sections**. Along the **top of the card you write the name of the class**.
- In the body of the card you list the class **responsibilities on the left** and the **collaborators on the right.**

Class-Responsibility-Collaborator (CRC)



Class-Responsibility-Collaborator (CRC)



Class: FloorPlan	
Description	
Responsibility:	**Collaborator:**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**

Class-Responsibility-Collaborator (CRC)

Classes

- Entity classes
  - Extracted directly from the statement of the problem
  - Represent things to be stored or persist throughout the development
- Boundary classes
  - Create/display interface
  - Mange how to represent entity objects to users
- Controller classes
  - Create/update entity objects
  - Initiate boundary objects
  - Control communications
  - Validate data exchanged

Class-Responsibility-Collaborator (CRC)

Responsibilities:

Five guidelines for allocating responsibilities to classes.

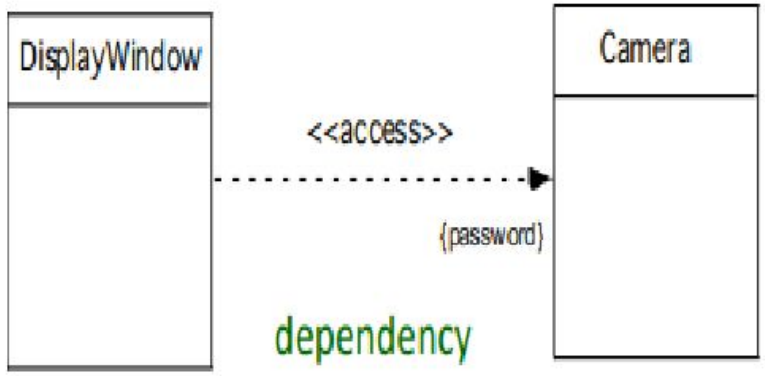
1. System intelligence should be **distributed across classes** to best address the needs of the problem.
2. Each responsibility should be **stated as generally as possible.**
3. **Information and the behavior** related to it should reside **within the same class**.
4. Information about **one thing should be localized** with a single class, not distributed across multiple classes.
5. Responsibilities **should be shared among related classes**, when appropriate.

Class-Responsibility-Collaborator (CRC)

Collaborations

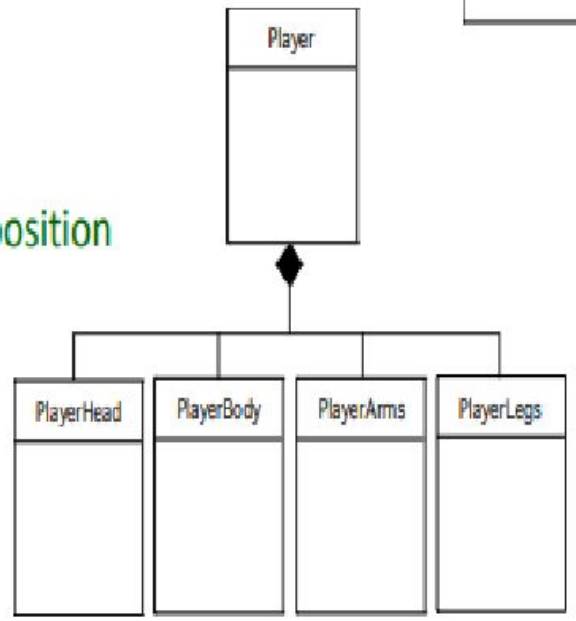
- A class may collaborate with other classes to fulfill responsibilities
  - If a class cannot fulfill every single responsibility itself, it must interact with another class
- Collaboration refers to identifying relationships between classes
  - is-part-of relationship
    - Aggregation
  - has-knowledge-of relationship: one class must acquire information from another class
    - Association
  - depends-upon relationship: dependency other than the above two

Relationships between Classes

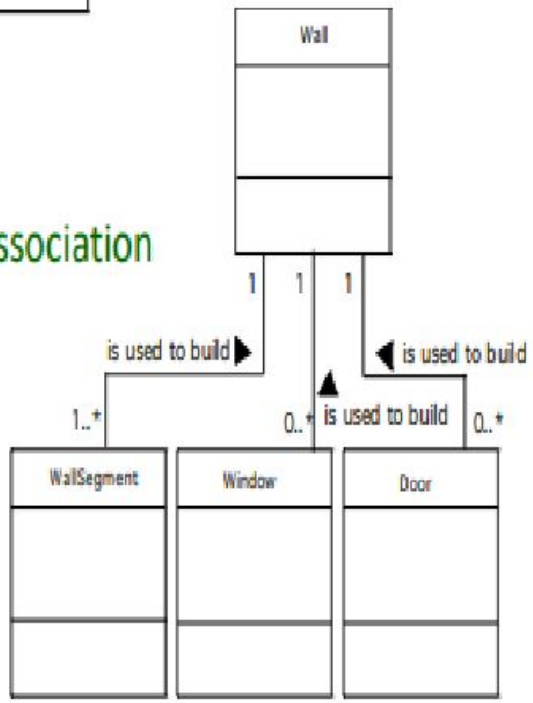


dependency

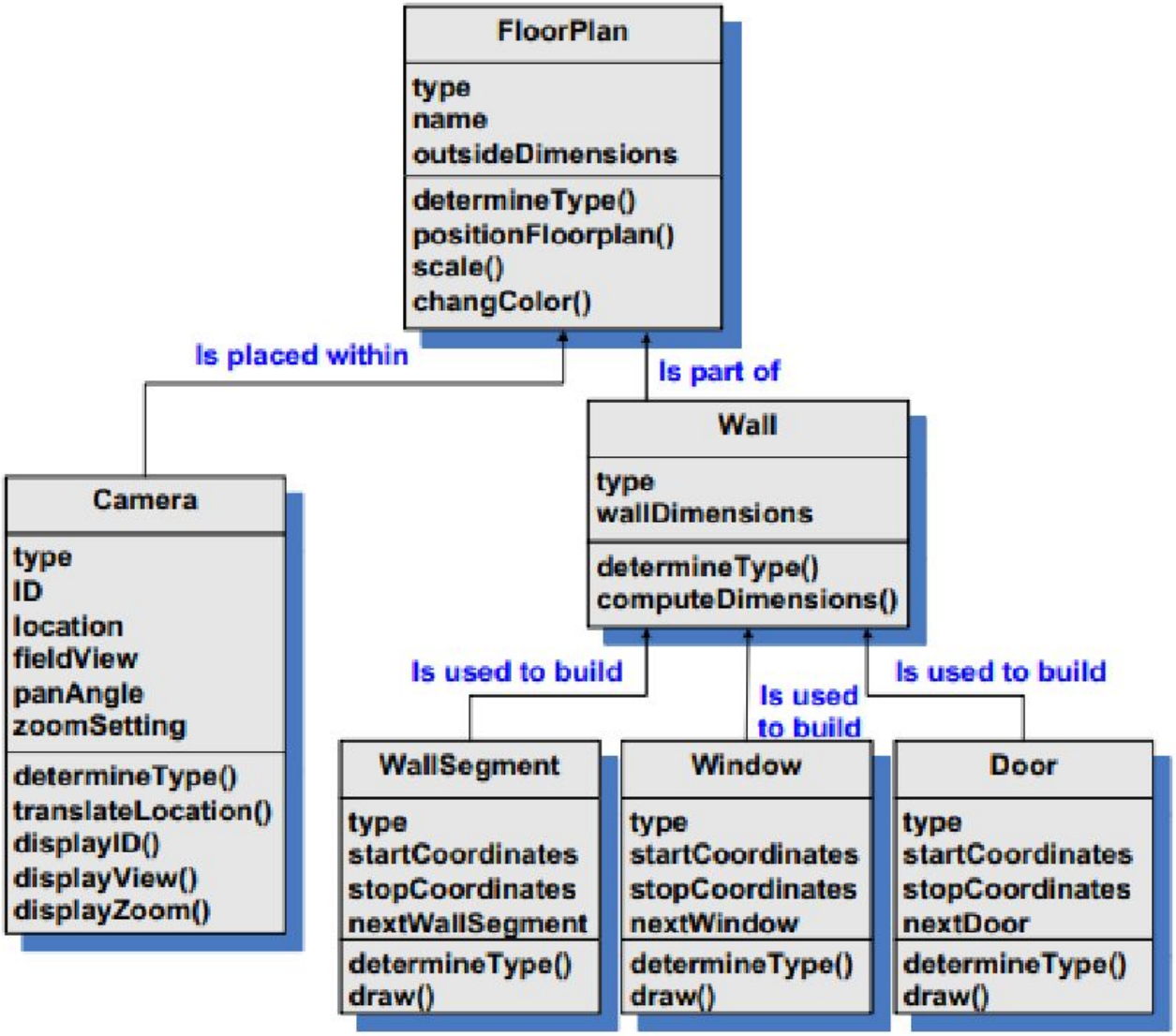
composition



association

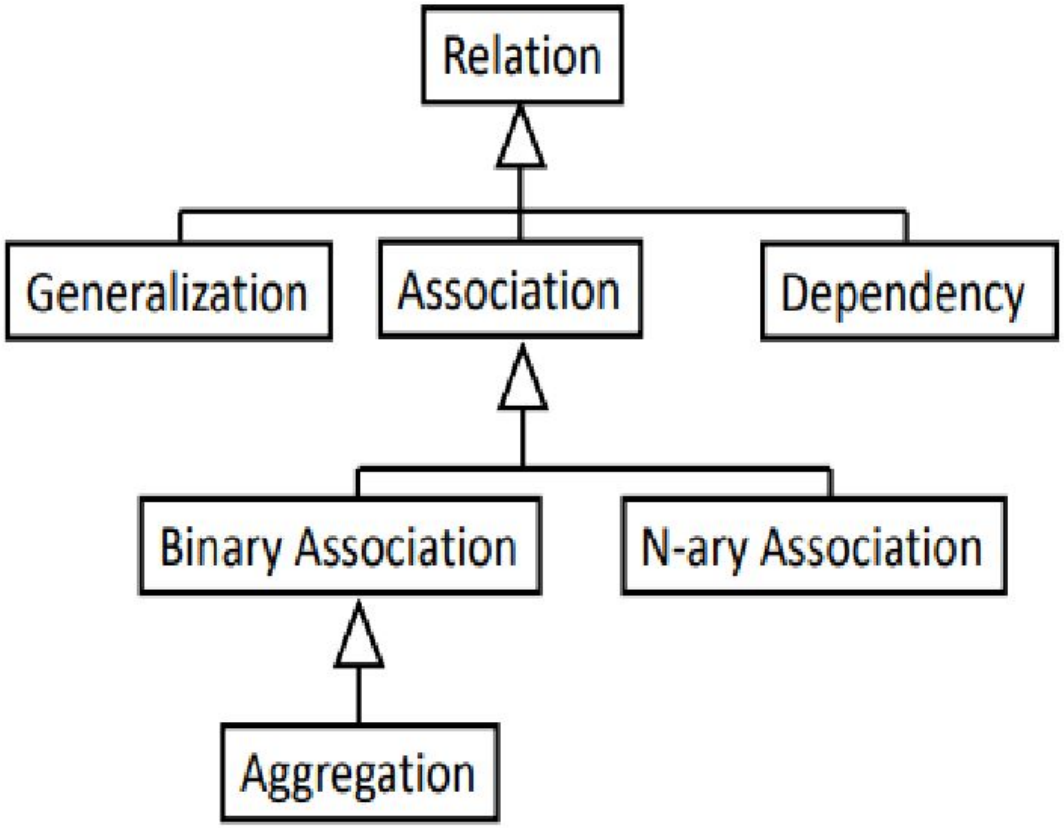


Class Diagram



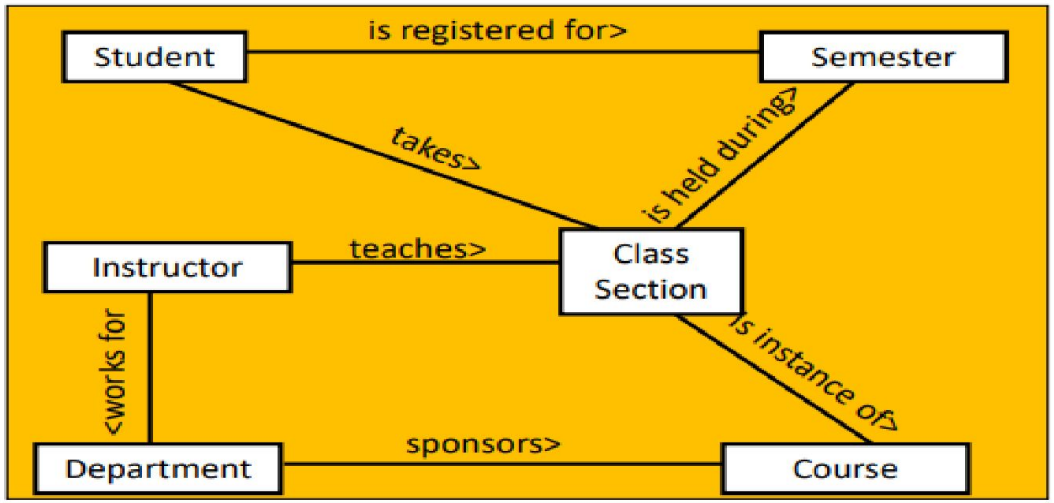
Type of Relationships in Class Diagrams

- Class diagrams show relationships between classes.



Associations

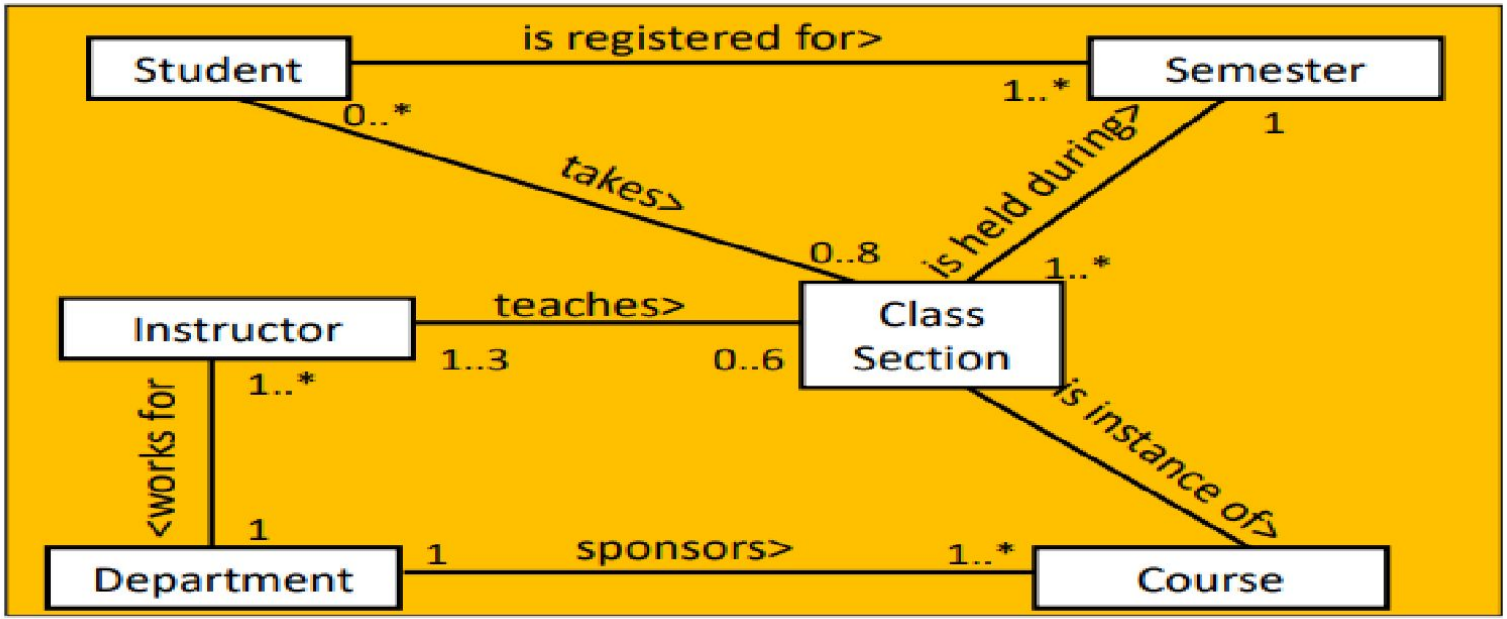
- An association is a *structural relationship* that specifies a connection between classes
- Classes A and B are associated if:
  - An object of class A sends a message to an object of B
  - An object of class A creates an instance of class B
  - An object of class A has an attribute of type B or collections of objects of type B
  - An object of class A receives a message with an argument that is an instance of B (maybe…)



Multiplicity

- How many objects from two classes are linked?
  - An exact number: indicated by the number
  - A range: two dots between a pair of numbers
  - An arbitrary number: indicated by * symbol
  - (Rare) A comma-separated list of ranges

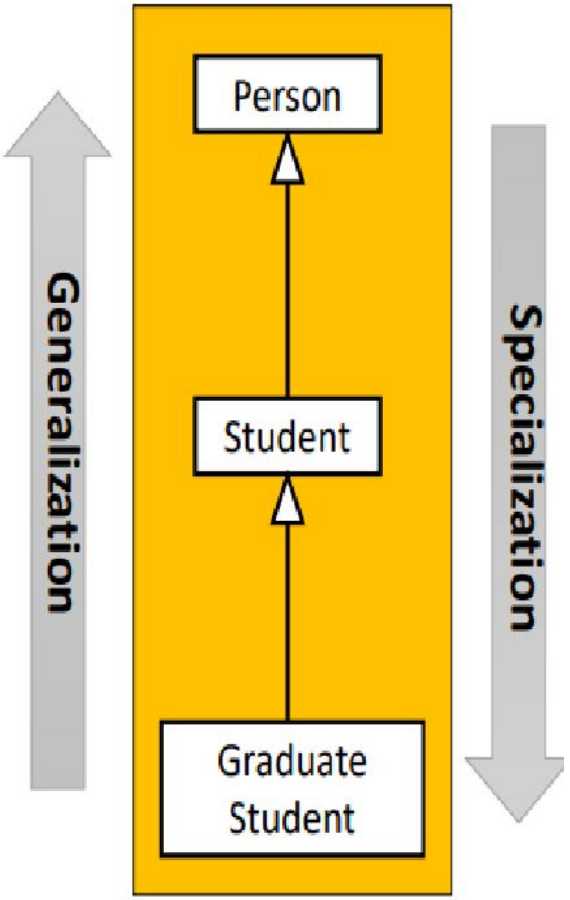
  - e.g., 1 1..2 0..* 1..* * (same as 0..*)
  - Implementing associations depends on multiplicity

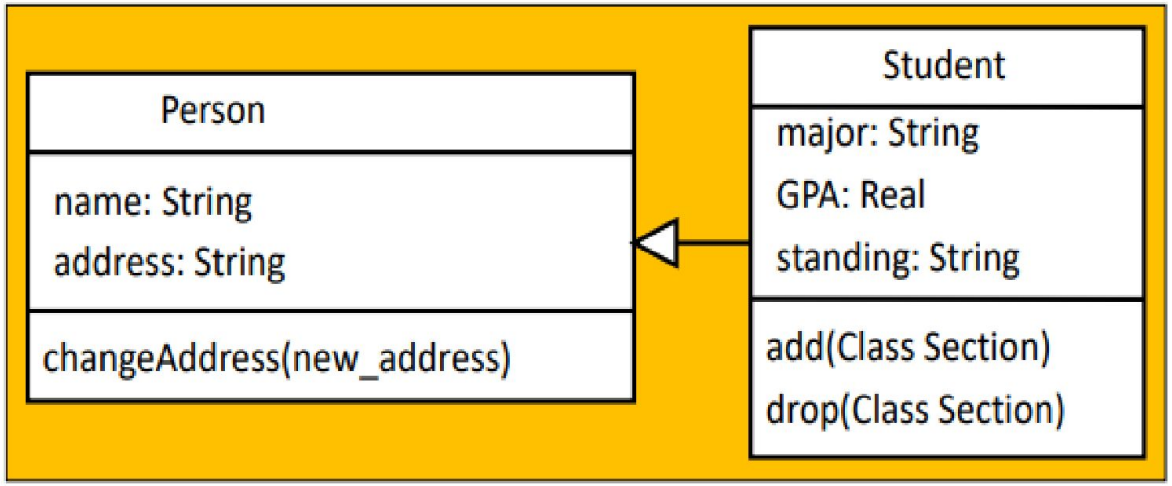


Generalization

- Generalization is an association between classes
  - A subclass is connected to a superclass by an arrow with a solid line with a hollow arrowhead.

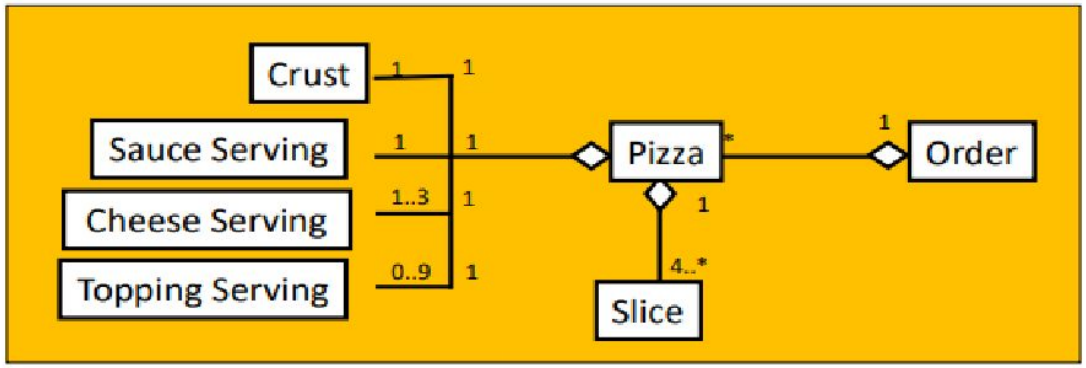
- From an analysis perspective, it represents generalization/specialization:
  - Specialization is a subset of the generalization





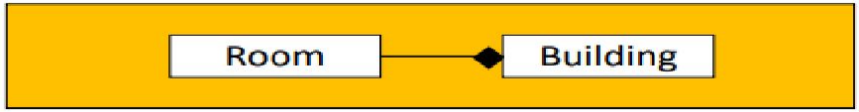
Aggregation

- Aggregation: is a special kind of association that means “part of”
- Aggregations should focus on a single type of composition (physical, organization, etc.)



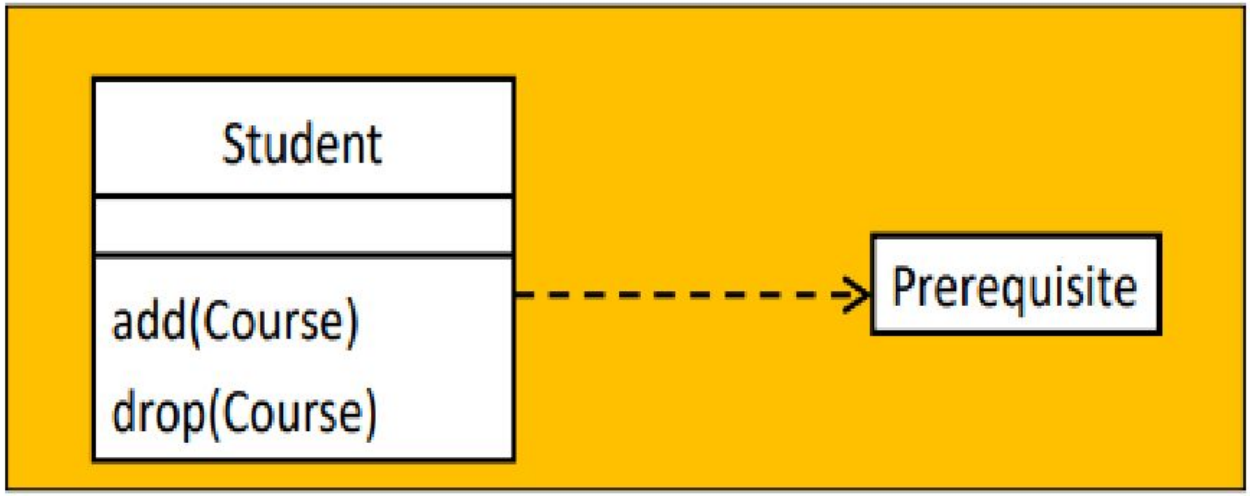
Composition

- Very similar to aggregation:
  - Think of composition as a stronger form of aggregation
  - **Composition means something is a part of the whole, but cannot survive on it’s own**



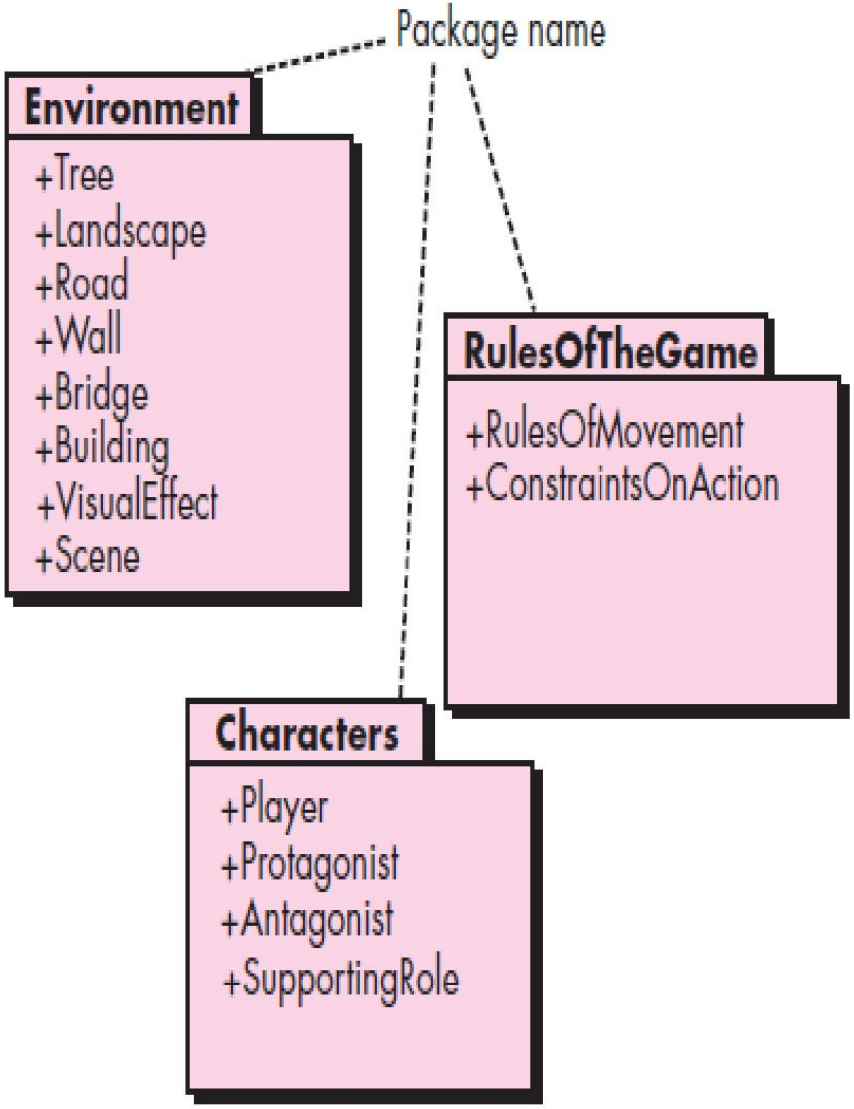
Dependencies

- A using relationship
  - A change in the specification of one class may affect the other
  - But not necessarily the reverse

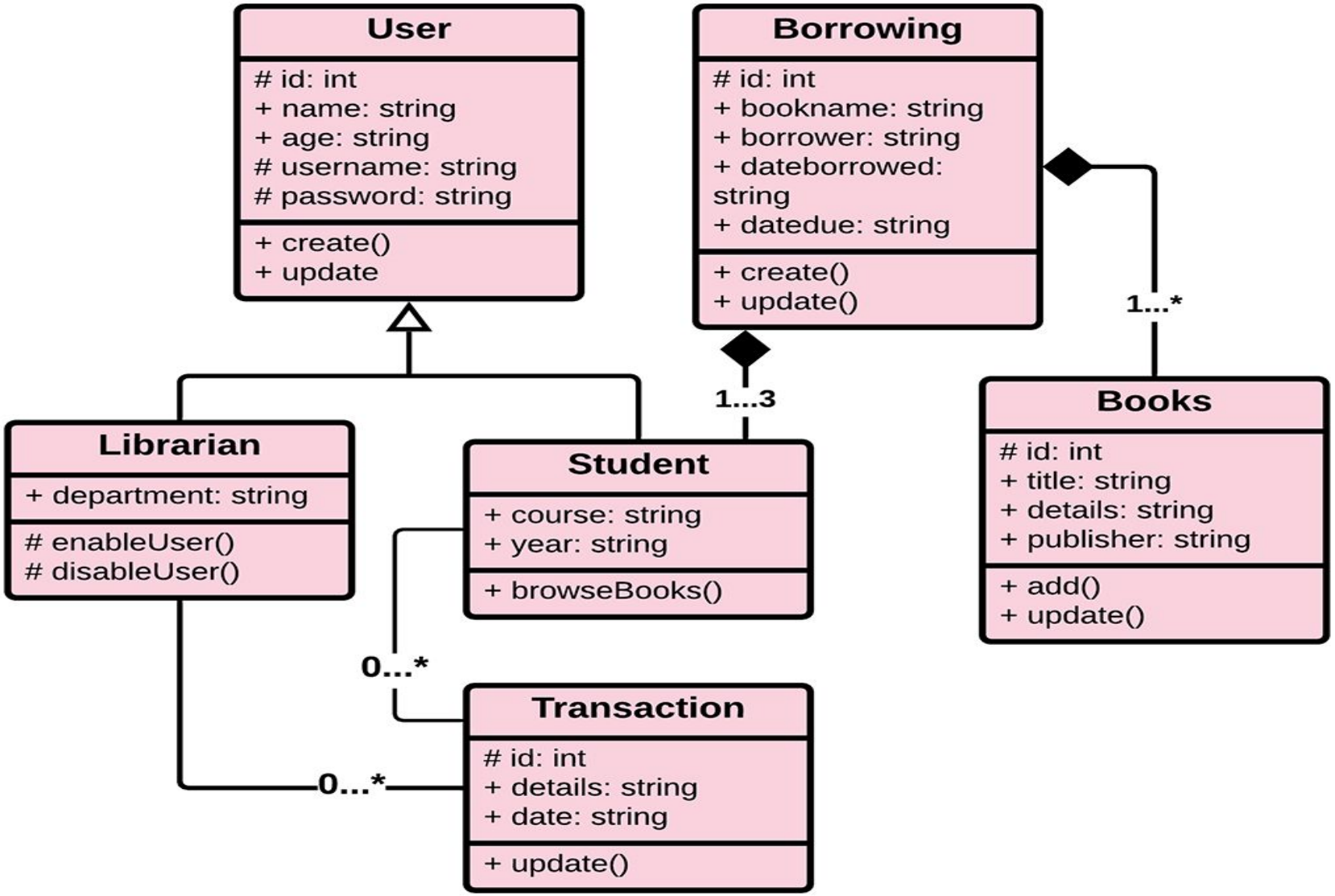


Analysis Packages

An important part of analysis modeling is **categorization**. That is, various elements of the analysis model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—**called an analysis package**—that is given a representative name.

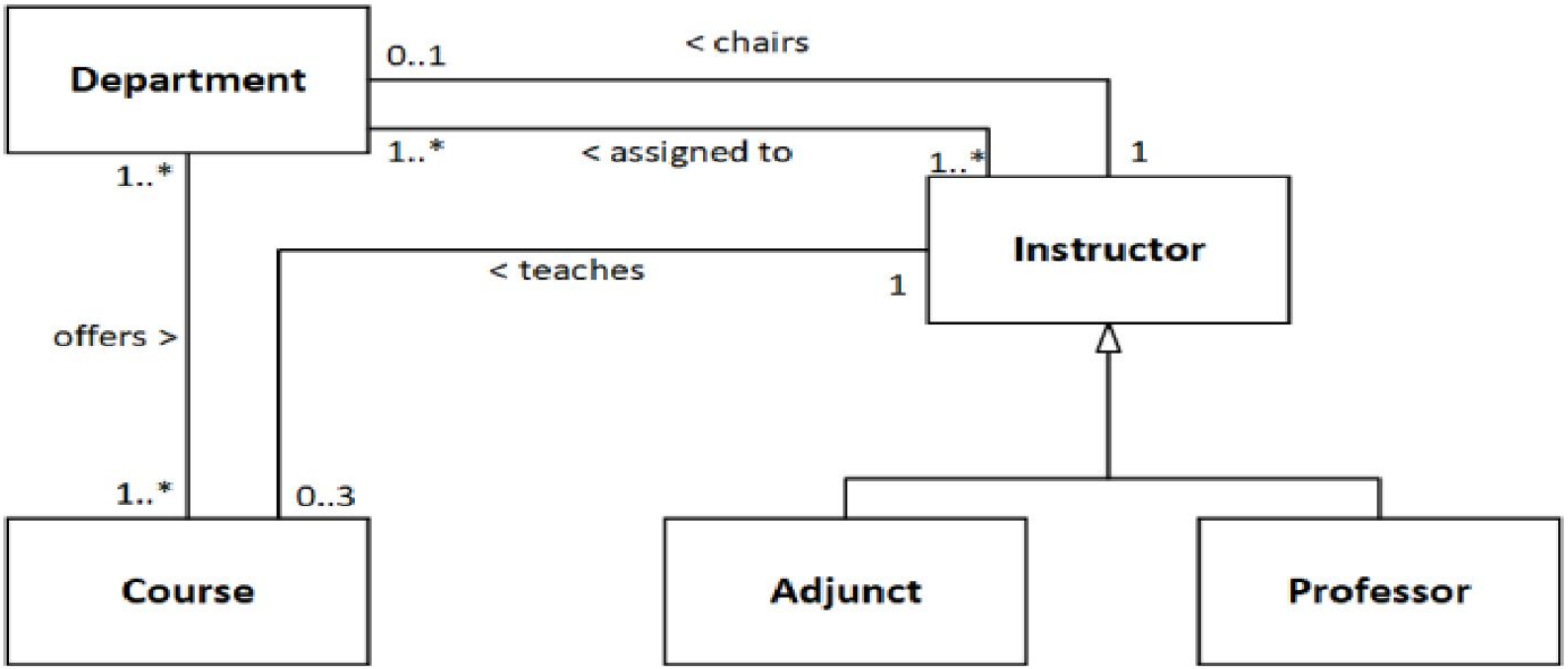


Library Management System



Example

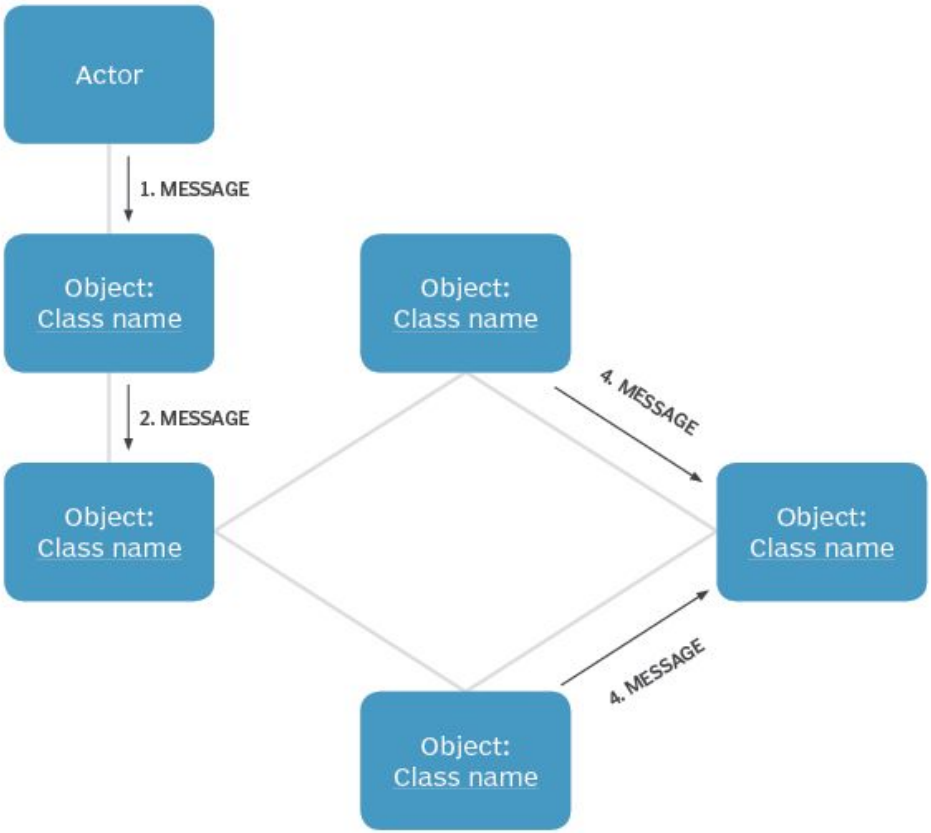
- University Courses
  - Some instructors are professors, while others have job title adjunct
  - Departments offer many courses, but a course may be offered by more than 1 department
  - Courses are taught by instructors, who may teach up to three courses
  - Instructors are assigned to one (or more) departments
  - One instructor also serves a department chair



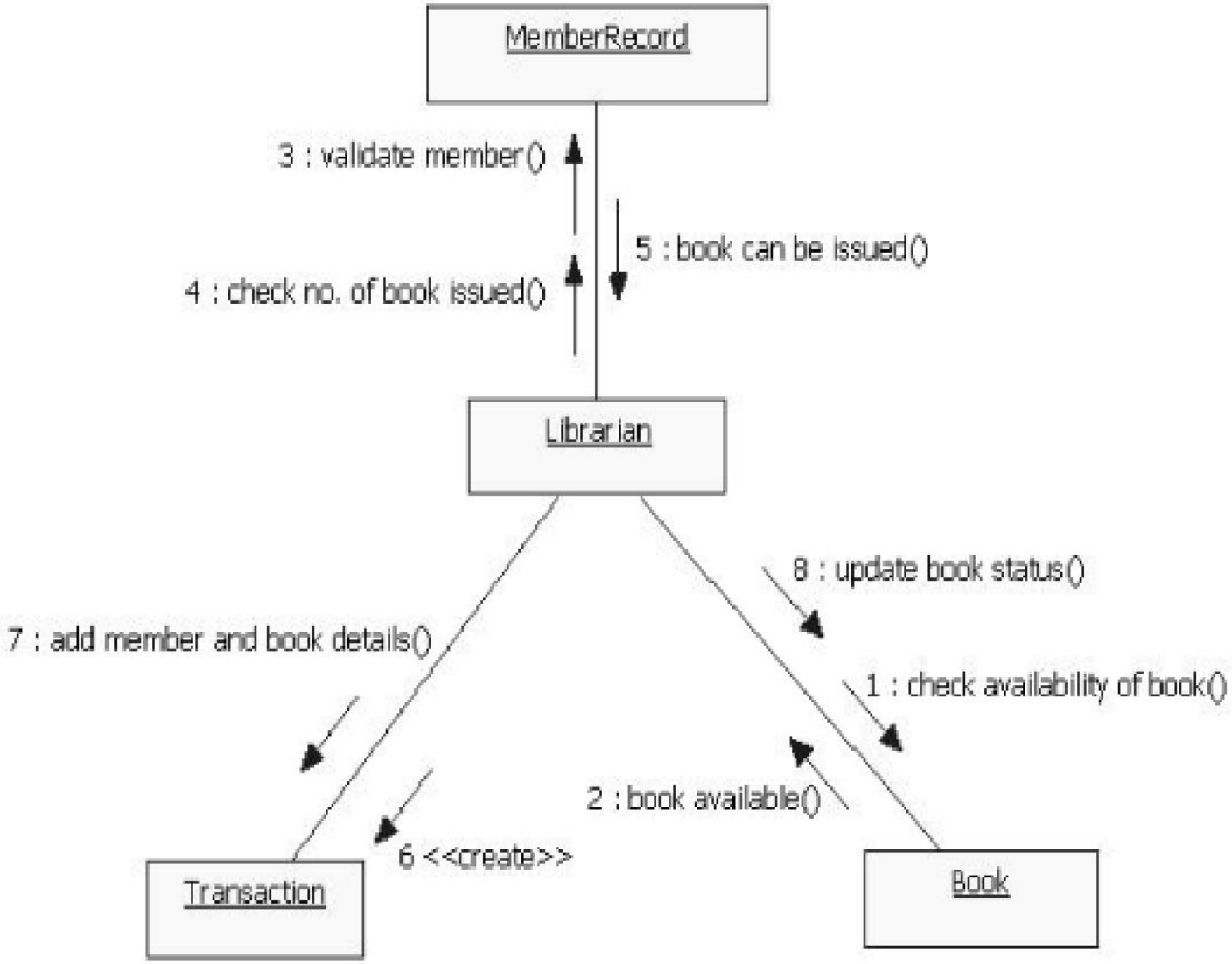
Collaboration Diagram

- A collaboration diagram, also known as a **communication diagram**, is an illustration of the relationships and interactions among software objects in the **Unified Modeling Language**(UML).
- The four major components of a collaboration diagram are:
  - **Objects-** Objects are shown as **rectangles** with naming labels inside.
  - **Actors-** Actors are instances that invoke the interaction in the diagram.
  - **Links-** Links connect **objects with actors** and are depicted using a solid line between two elements.
  - **Messages-** Messages between objects are shown as a labeled arrow placed near a link.

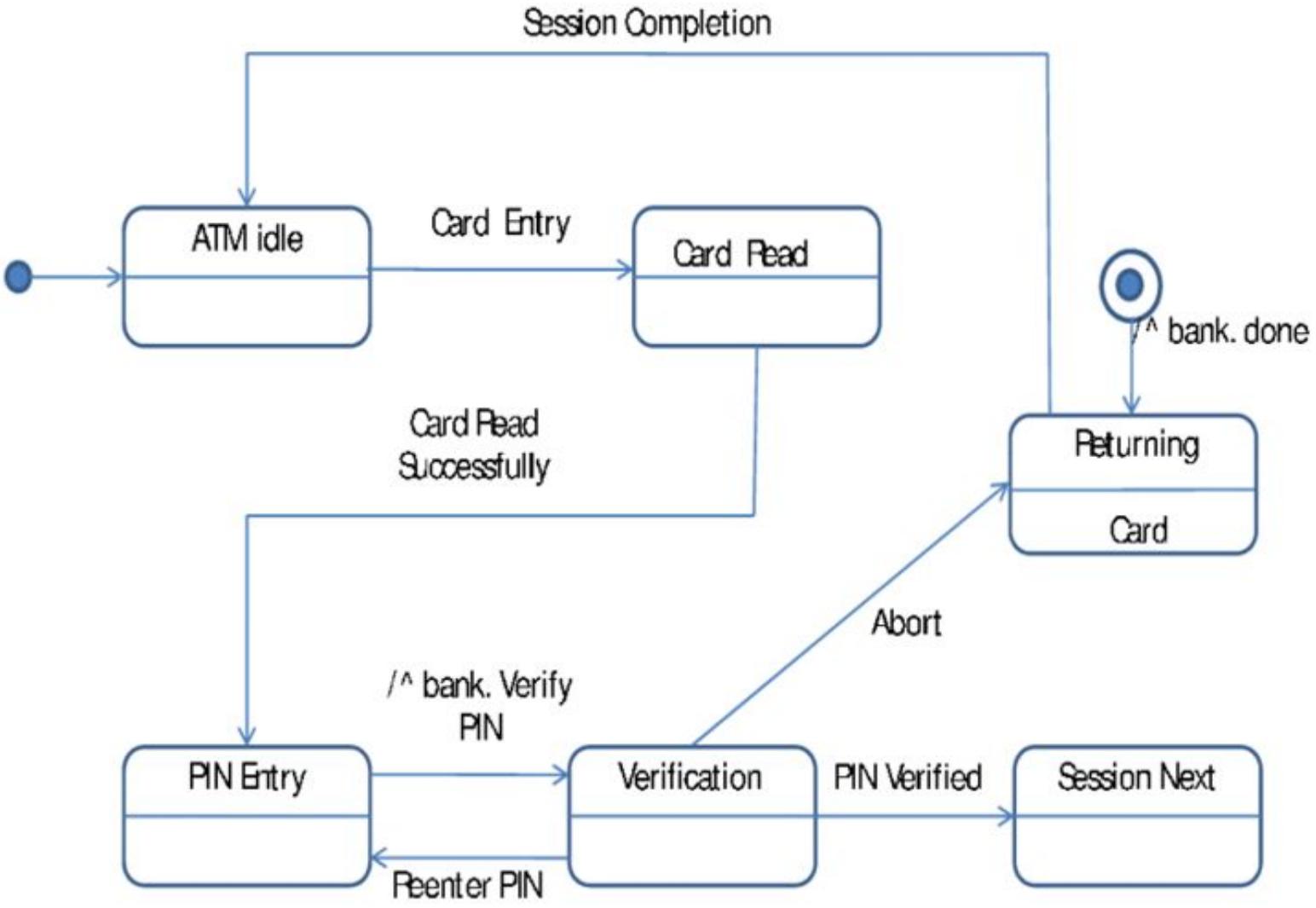
Components of a collaboration diagram



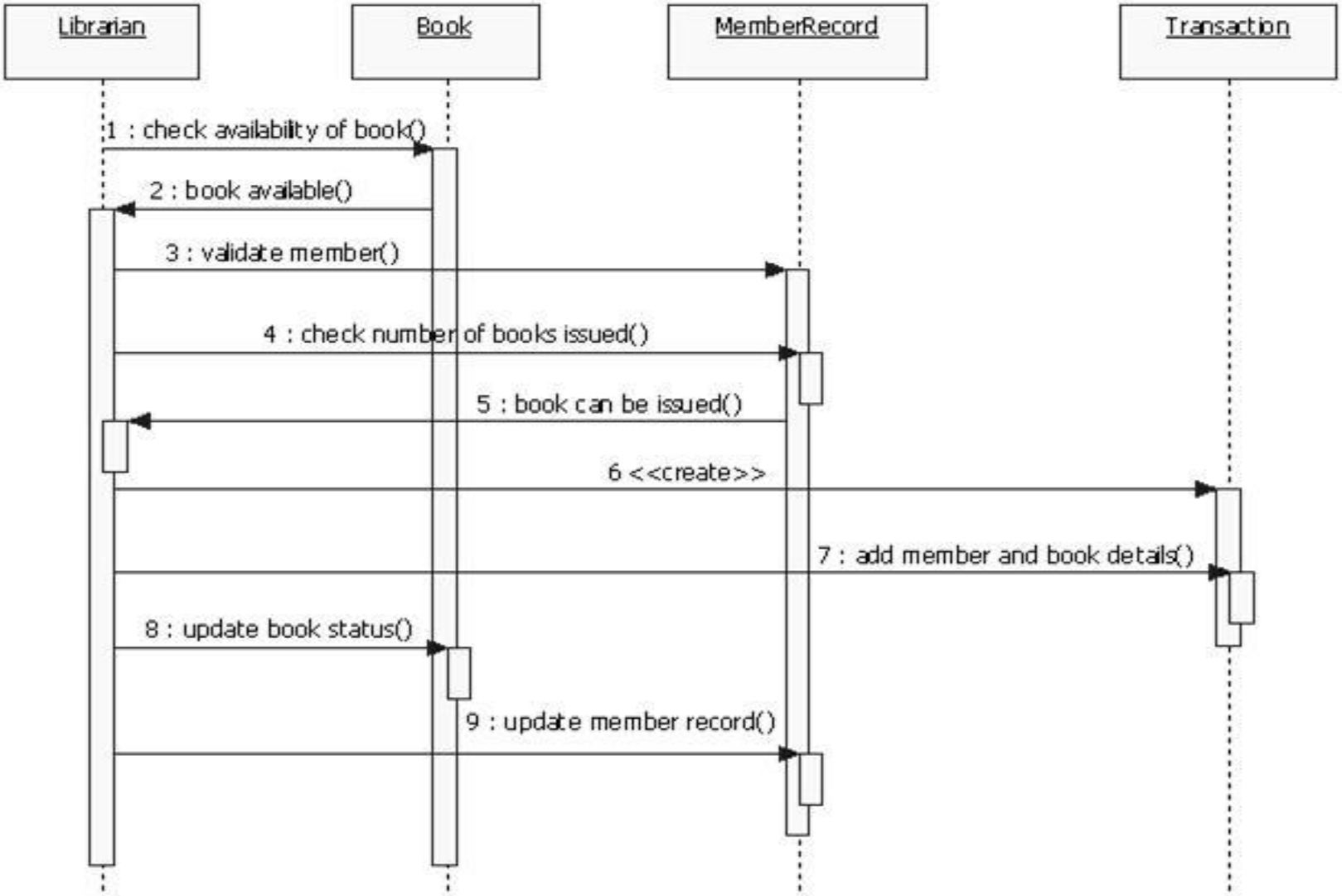
Collaboration Diagram for Issuing Book



State Chart Diagram Example



Sequence Diagram Example



22CSC51 - AGILE METHODOLOGIES

Prepared By,

Mr.N.Aravindhraj,

Assistant Professor

Department.of CSE

Kongu Engineering College

Quality attributes

The attributes of design name as **'FURPS'** are as follows:

Functionality:
It evaluates the **feature set and capabilities** of the program.

Usability:
It is accessed by considering the factors such as **human factor, overall aesthetics, consistency and documentation**.

Reliability:
It is evaluated by measuring parameters like frequency and **security of failure, output result accuracy, the mean-time-to-failure**(MTTF), recovery from failure and the program predictability.

Performance:
It is measured by considering processing **speed, response time, resource consumption, throughput and efficiency**.

Supportability:

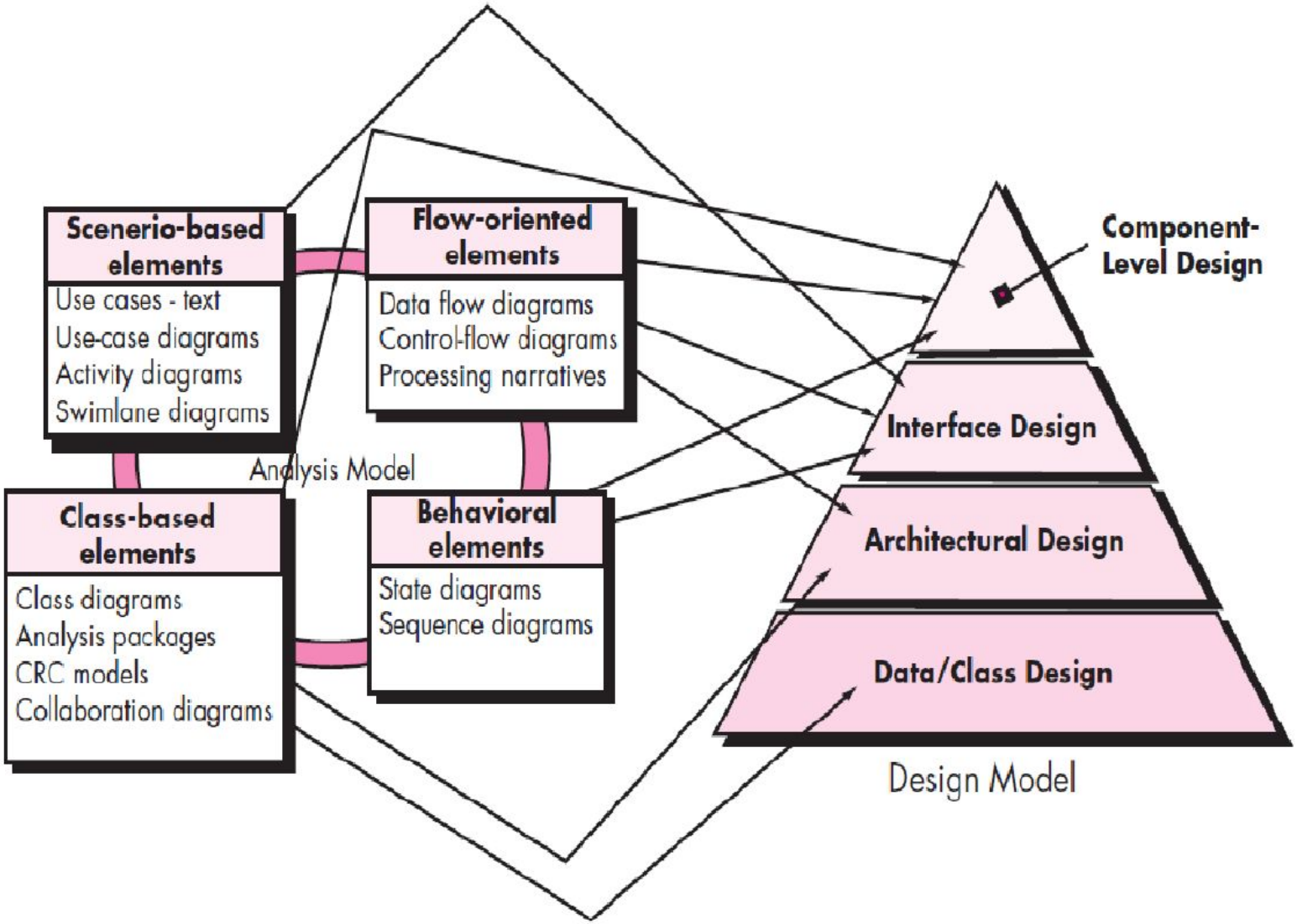
- It combines the ability to extend the **program, adaptability, serviceability**. These three term defines the maintainability.
- **Testability, compatibility and configurability** are the terms using which a system can be easily installed and found the problem easily.
- Supportability also consists of more **attributes such as compatibility, extensibility, fault tolerance, modularity, reusability, robustness, security, portability, scalability**.

Design Concepts

The goal of the Good software design is to produce a model or representation that exhibits

- **Firmness:** A program should not have any bugs that inhibit its function.
- **Commodity:** A program should be suitable for the purposes for which it was intended.
- **Delight:** The experience of using the program should be pleasurable one.

Design Model



Design Concepts

A set of **fundamental software design concepts** has evolved over the history of Software Engineering.

Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you answer the following questions:

- **What criteria can be used to partition software into individual components?**
- **How is function or data structure detail separated from a conceptual representation of the software?**
- **What uniform criteria define the technical quality of a software design?**

Design Concepts

The set of fundamental software design concepts are as follows:

- **Abstraction**
- **Architecture**
- **Patterns**
- **Separation of Concerns**
- **Modularity**
- **Information hiding**
- **Functional independence**
- **Refinement**
- **Aspects**
- **Refactoring**
- **OO Design Concepts**
- **Design classes**

Design Concepts

- Abstraction—data, procedure, control
- Architecture—the overall structure of the software
- Patterns—”conveys the essence” of a proven design solution
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- Refinement—elaboration of detail for all abstractions
- Aspects—a mechanism for understanding how global requirements affect design
- Refactoring—a reorganization technique that simplifies the design
- OO design concepts
- Design Classes—provide design detail that will enable analysis classes to be implemented

Abstraction

- Abstraction is the **act of representing essential features** without **including the background** details or explanations.
- The abstraction is used to **reduce complexity** and allow efficient design and implementation of complex software systems.
- Many levels of **abstraction can be posed**.
- 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.

Abstraction

Procedural Abstraction

- A **procedural abstraction** refers to a **sequence of instructions** that have a specific and limited function. The name of a procedural abstraction implies the functions, but specific details are suppressed.
- An example of a procedural abstraction would be the word **open for a door**. Open implies a long sequence of procedural steps.
- e.g., **walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.)**.

Abstraction

Data Abstraction

- A data abstraction is a **named collection of data** that describes a data object.
- In the context of the procedural abstraction open, we can define a **data abstraction called door**.
- Like any data object, the data abstraction for door would **encompass a set of attributes** that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).
- It follows that the **procedural abstraction** open would make use of information contained in the **attributes of the data abstraction door**.

Architecture

- Software architecture alludes to **"the overall structure of the software and the ways in which that structure provides conceptual integrity for a system"**.
- In its simplest form, architecture is the **structure or organization of program components** (modules), the manner in which these components interact, and the structure of data that are used by the components.
- set of **properties** that should be specified as part of an architectural design:
  - **Structural properties**
  - **Extra-functional properties**
  - **Families of related systems**

Architecture

Structural properties:

- This aspect of the architectural design representation defines the **components of a system** (e.g., modules, objects, filters) and the manner in which those components are **packaged and interact** with one another.

Extra-functional properties:

- The architectural design description should address

how the design architecture achieves requirements for **performance, capacity, reliability, security, adaptability,** and other system characteristics.

Families of related systems:

- The architectural design should draw upon **repeatable patterns that are commonly encountered** in the design of families of similar systems.
- In essence, the design should have the **ability to reuse architectural building blocks.**

Architecture

- The architectural design can be represented using **one or more** of a number of different models.
- **Structural models**: Represent architecture as an **organized collection** of program components.
- **Framework models**: **Increase the level of design abstraction** by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.
- **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.
- A number of different architectural description languages (ADLs) have been developed to represent these models.

Patterns

- **Brad Appleton** defines a design pattern in the following manner: “A pattern is a named nugget of insight which **conveys the essence of a proven solution** to a recurring problem within a certain context amidst competing concerns”.
- Design pattern describes a **design structure** that solves a particular design problem within a specific context and amid **“forces”** that may have an impact

on the manner in which the pattern is **applied and used**.

Patterns

- The intent of each design pattern is to provide a description that enables a designer to determine:

(1) whether the **pattern is applicable** to the current work,

(2) whether the **pattern can be reused** (hence, saving design time), and

(3) whether the **pattern can serve as a guide** for developing a similar, but functionally or structurally different pattern.

Separation of Concerns

- Separation of concerns is a **design concept** that suggests that any **complex problem** can be more easily handled if it is **subdivided into pieces** that can each be solved and/or optimized independently.
- A concern is a **feature or behavior** that is specified as part of the requirements model for the software.
- By separating concerns into smaller, and therefore more manageable pieces, **a problem takes less effort and time to solve.**
- For two problems, **p1 and p2**, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the **effort required to solve p1 is greater** than the effort required to solve p2. As a general case, this result is intuitively obvious.

Separation of Concerns

- It does not **take more time to solve** a difficult problem.
- Separation of concerns is manifested in other related design concepts:

  Modularity,

  Aspects,

  Functional Independence,

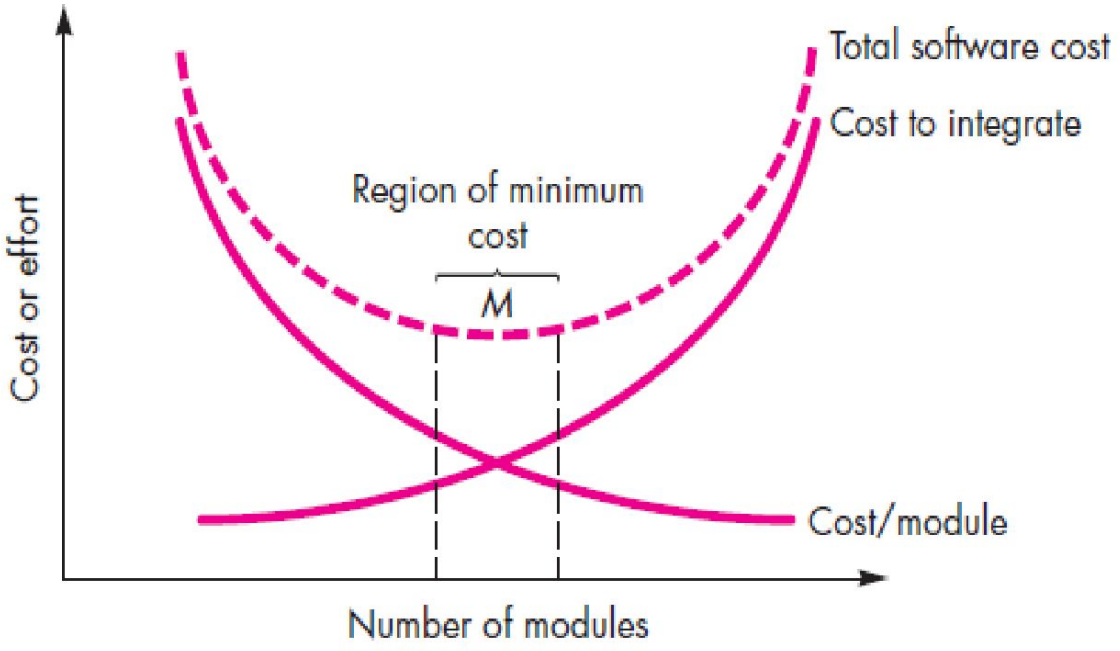
  Refinement.

Modularity

- Modularity is the most **common manifestation** of separation of concerns.
- Software is **divided into separately named and addressable components**, sometimes called modules, that are integrated to satisfy problem requirements.
- It has been stated that **“modularity is the single attribute of software that allows a program to be intellectually manageable”**

Modularity

- However, as the number of **modules grows**, the effort (cost) associated with integrating the **modules also grows**.



Information Hiding

- The principle of information hiding suggests that modules be **"characterized by design decisions that hides from all others."**
- In other words, modules should be specified and designed so that information contained within a module is **inaccessible to other modules** that have no need for such information.
- The use of **information hiding** as a design criterion for modular systems provides the greatest benefits when **modifications are required during testing** and later during software maintenance.

Functional Independence

- The concept of **functional independence** is a direct outgrowth of separation of concerns, modularity, and the concepts of **abstraction and information hiding**.
- Functional independence is achieved by developing modules with **"single minded" function and an "aversion"** to excessive interaction with other modules.
- Independence is assessed using two qualitative criteria:
  - **Cohesion**
  - **Coupling**

Cohesion

- Cohesion is an **indication of the relative functional** strength of a module.
- Cohesion is a natural extension of the **information-hiding** concept
- A cohesive module performs a **single task**, requiring little interaction with other components in other parts of a program.
- Stated simply, a **cohesive module** should (ideally) do just one thing.
- Although you should always strive for **high cohesion** (i.e., single-mindedness)

Coupling

- Coupling is an indication of the **relative interdependence** among modules.
- Coupling is an indication of interconnection among modules in a software structure.
- Coupling depends on the **interface complexity between modules**, the point at which entry or reference is made to a module, and what data pass across the interface.
- In software design, you should **strive for the lowest possible** coupling.

Refinement

- Stepwise refinement is a **top-down design strategy**.
- Refinement is actually a **process of elaboration**.
- **Abstraction and refinement** are complementary concepts.
- Abstraction enables you to specify procedure and data internally but **suppress the need for "outsiders"** to have knowledge of low-level details.
- Refinement helps you to **reveal low-level details as design progresses**.

Aspects and Refactoring

Aspects

- An aspect is a **representation of a crosscutting concern**.
- A crosscutting concern is some characteristic of the system that applies across many different requirements.

Refactoring

- "Refactoring is the **process of changing a software system** in such a way that it **does not alter the external behavior** of the code [design] yet improves its internal structure."
- An important design activity suggested for many agile methods, **refactoring is a reorganization technique** that simplifies the **design (or code) of a component without changing its function or behavior**.

Object-Oriented Design Concepts

- The object-oriented (OO) paradigm is widely used in modern software engineering.
- OO design concepts such as
  - **Classes and Objects**
  - **Inheritance**
  - **Message Passing**
  - **Polymorphism**

  are widely used.

Design Classes

- A set of design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and **implement a software infrastructure** that supports the business solution.
- Five different types of **design classes**, each representing a different layer of the design architecture, can be developed:
- User interface classes define all **abstractions that are necessary for human computer interaction** (HCI).
- The design classes for the interface may be visual representations of the elements of the metaphor.

Design Classes

- **Business domain classes** are often **refinements of the analysis classes defined earlier**. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- **Process classes** implement **lower-level business abstractions** required to fully manage the business domain classes.
- **Persistent classes** represent **data stores (e.g., a database)** 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.

Design Classes

- Four characteristics of a well-formed design class:
  - **Complete and sufficient**
  - **Primitiveness**
  - **High cohesion**
  - **Low coupling**