22CSC51 - AGILE METHODOLOGIES

Prepared By,

Mr.M.Muthuraja,

Assistant Professor

Department.of CSE

Kongu Engineering College

22CSC51 - AGILE METHODOLOGIES

Course Outcomes

Unit – I Process Models, Analysis and Design

Apply the requirement engineering tasks, design concepts and analyze the various software development models for a given scenario

Unit - II Agile Principles and Scrum

Outline agile principles and apply Scrum for project development.

Unit - III XP and Incremental Design, Lean, and Kanban

Model applications using XP, Lean and Kanban practices.

22CSC51 - AGILE METHODOLOGIES

Course Outcomes

Unit - IV Software Testing Fundamentals

Make use of various software testing techniques to test the software systems.

Unit - V Software Project Management

Estimate the cost of software, risks of handling, do software planning and configuration management.

What is Software?

Software is:

(1) **instructions** *(computer programs) that when executed provide desired features, function, and performance;*

(2) **data structures** *that enable the programs to adequately manipulate information and*

(3) **documentation** *that describes the operation and use of the programs.*

Characteristics

- Software is **developed or engineered;** it is not **manufactured**.
- Software doesn't **"wear out."**
- Although the industry is moving toward component-based construction, most software continues to be custom built.

Software Application Domains

System software—a collection of programs written to service other programs.

(e.g., compilers, editors, operating system components, drivers)

Application software—stand-alone programs that solve a specific business need.

e.g., word processing, spreadsheets,

Engineering/scientific software—has been characterized by “number crunching” algorithms.

Embedded software—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself.

(e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Software Application Domains

Product-line software—designed to provide a specific capability for use by many different customers.

(e.g., inventory control products) or address mass

consumer markets)

Web applications—called “WebApps,” this network-centric software category spans a wide array of applications.

Artificial intelligence software

Applications within this area include robotics, expert systems,

pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

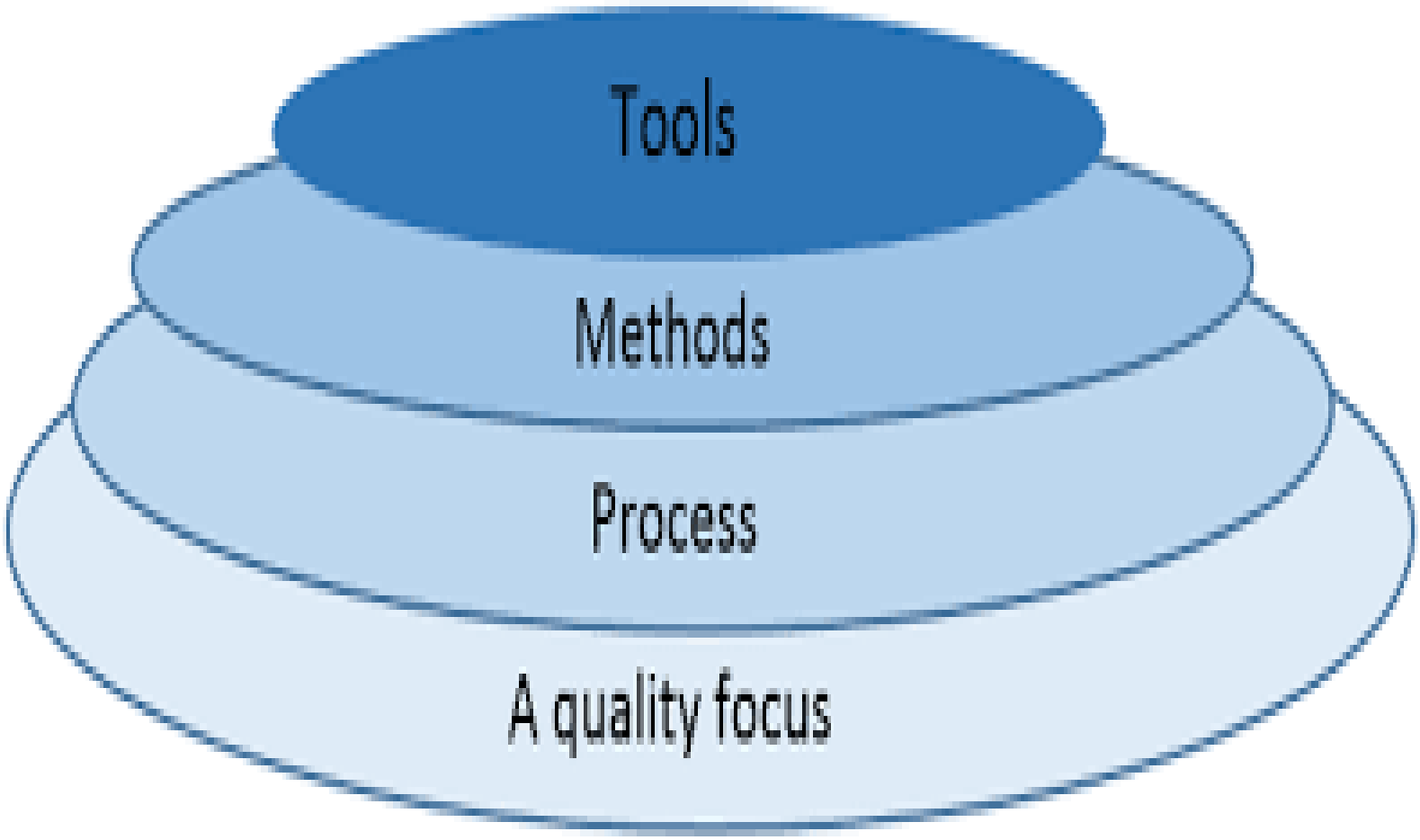
Software Engineering

Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

The IEEE Definition:

Software Engineering is defined as the application of a **systematic, disciplined, quantifiable approach** to the development, operation, and maintenance of software; that is, the **application of engineering to software**.

Software Engineering Layers



Software Process and Process Model

❖ ***Software process***

Is a **framework** for the activities, actions, and tasks that are required to build high quality software.

❖ ***A process Models***

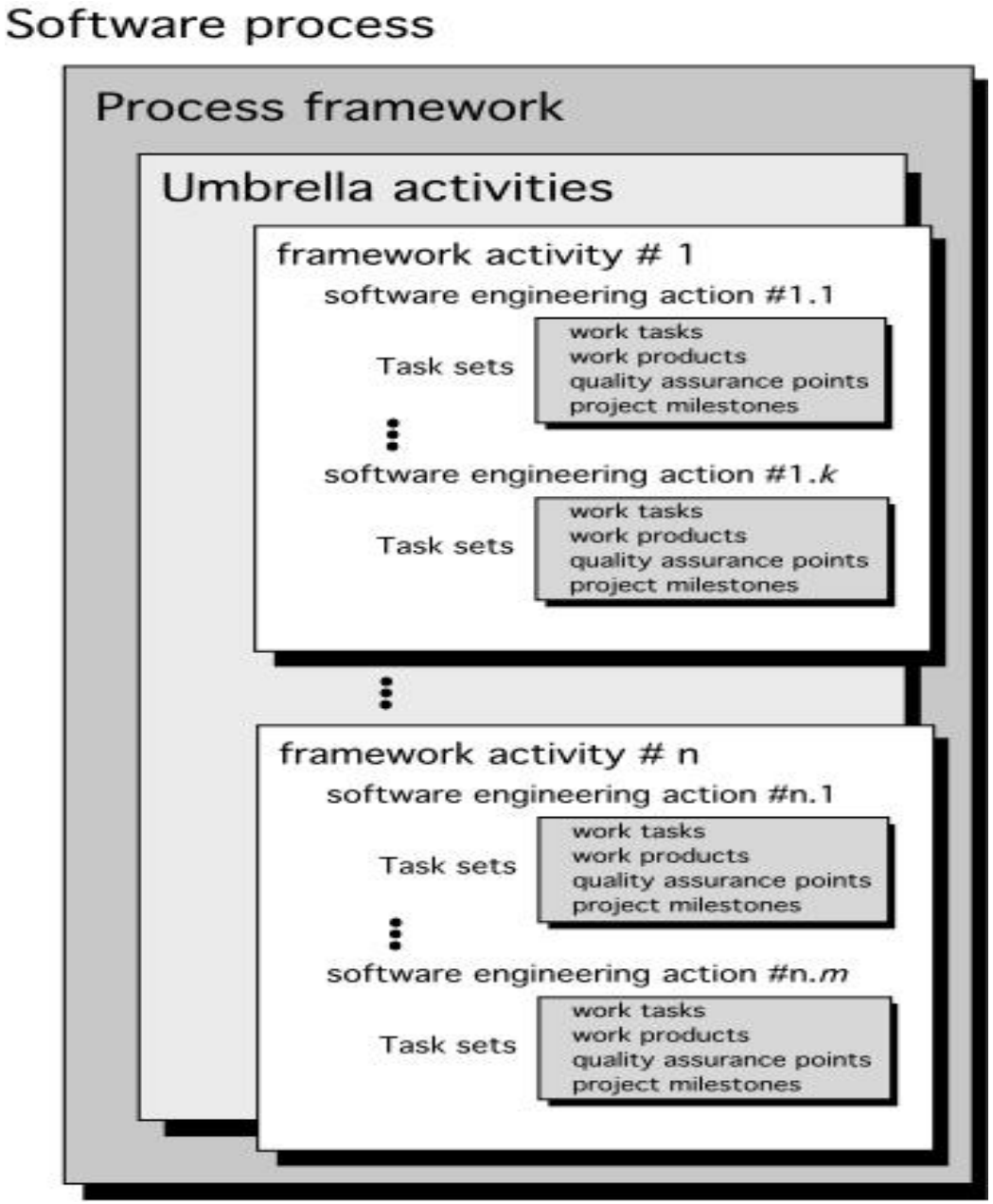
Describe the **sequence of phases** for the entire lifetime of a product. It is also known as ***Product Life Cycle.***

Generic Process Model

Generic process framework for software engineering defines

- ***Process*** was defined as a collection of work activities, actions and tasks that are performed when some product is to be created
- 5 framework activities
  - **Communication –** Gather Requirements from Customers
  - **planning –** Plan to be followed will be created. It describes the technical task to be conducted, risk ,required resources, work schedule etc.,
  - **Modeling –** A model will be created to better understand the requirements
  - **Construction –** Code generated and tested
  - **Deployment** - Complete or partial complete version of software is given to the customer to evaluate and they give feed back based on the evaluation
- a set of umbrella activities
  - Project tracking and control, risk management, quality assurance, configuration management, technical reviews, reusability, documentation and Measurement.

Generic Process Model

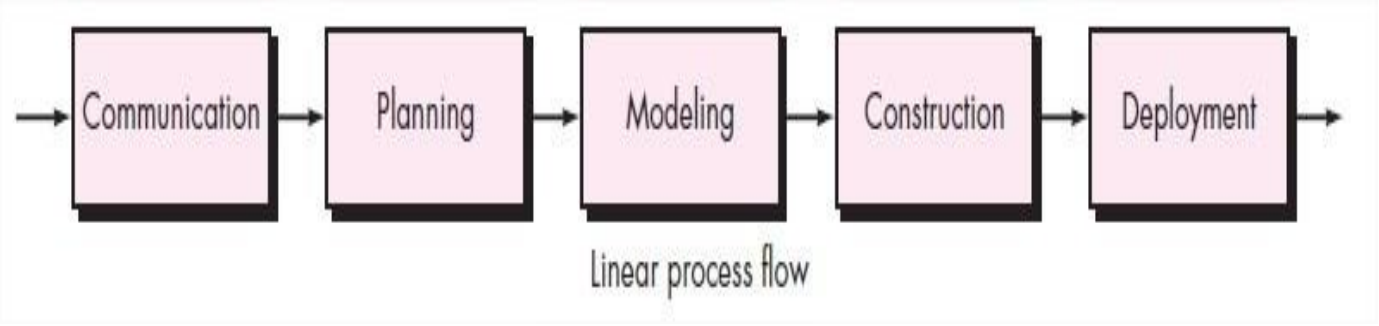


Task Set

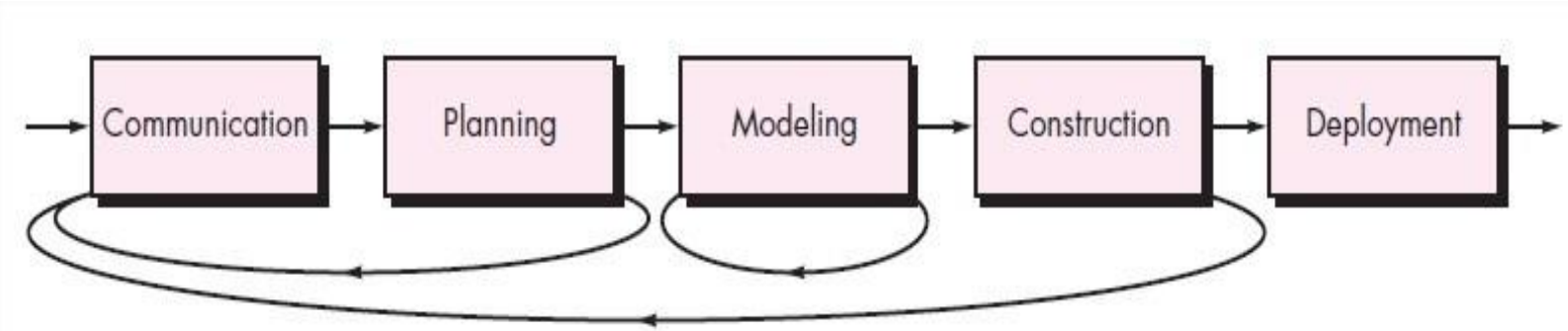
- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
  - A list of the task to be accomplished
  - A list of the work products to be produced
  - A list of the quality assurance filters to be applied

Process Flow

Describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time.

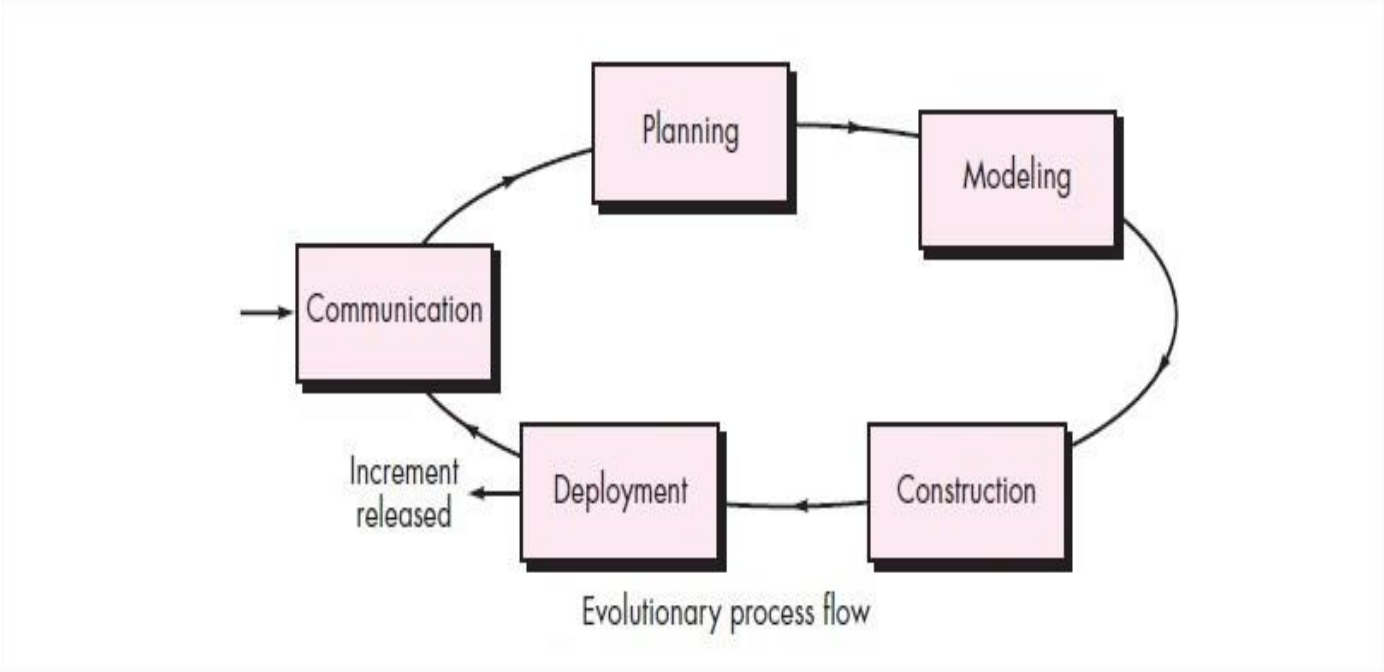


Linear process flow

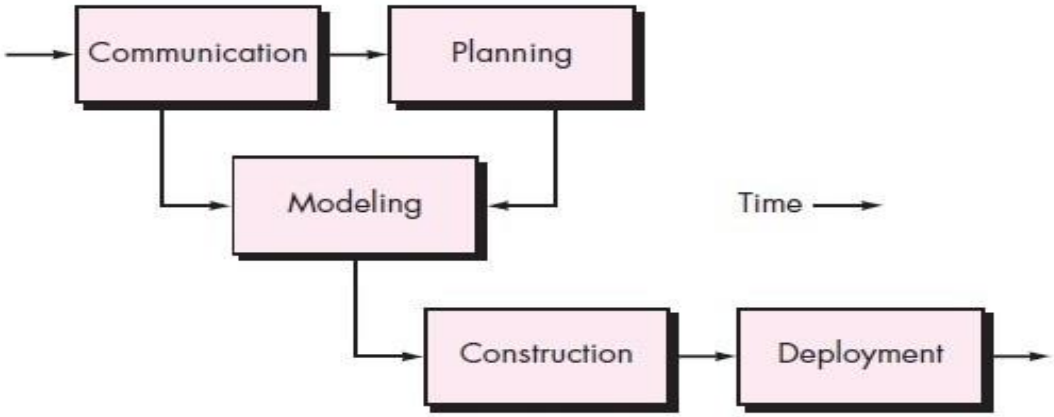


Iterative process flow

Process Flow



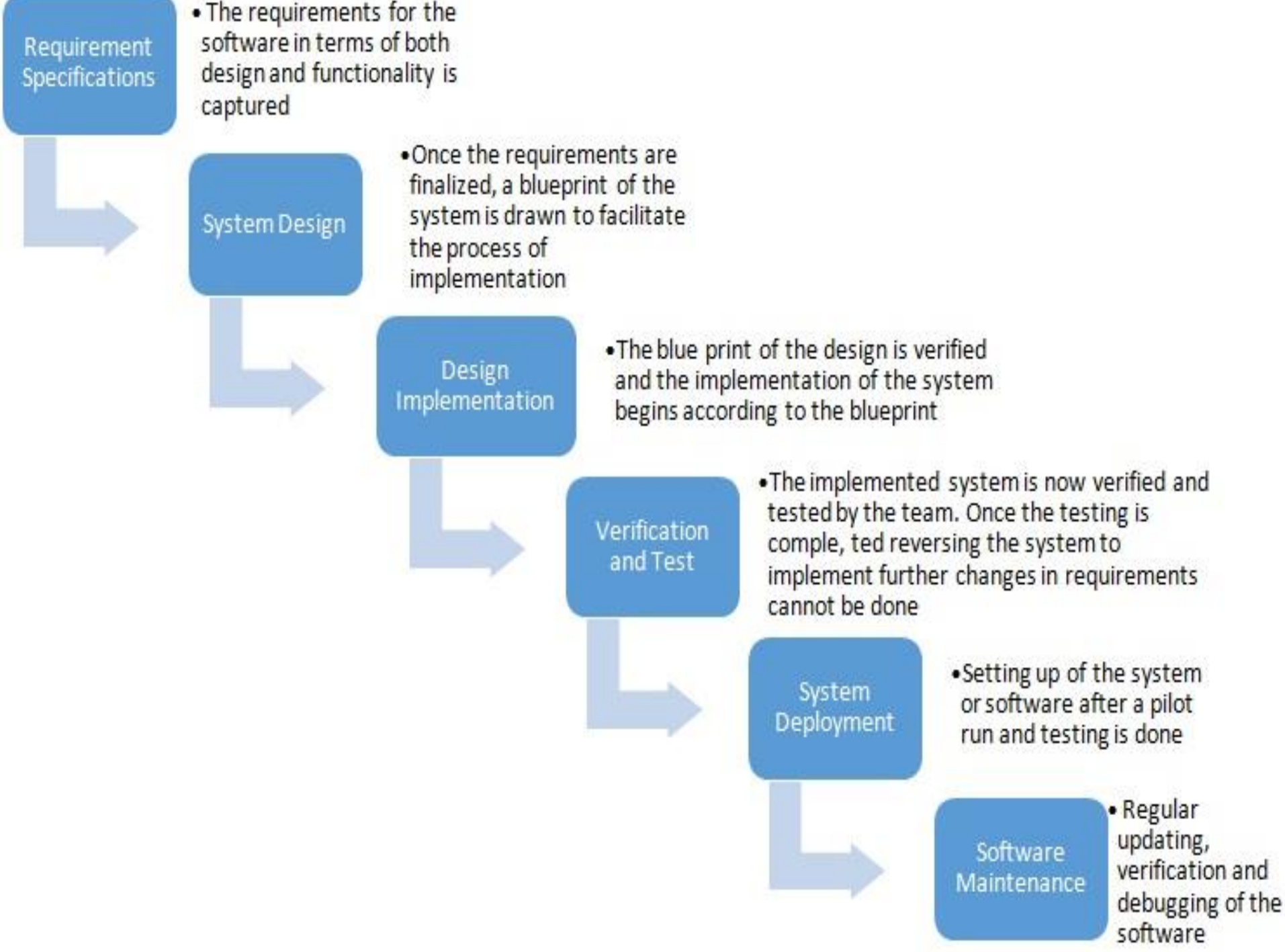
Evolutionary process flow



Parallel process flow

Waterfall Model

- The first formal description of waterfall model is cited in 1970 by **Winston W.Royce.**
- The Waterfall Model was the **first Process Model** to be introduced. It is also referred to as a **linear-sequential life cycle model.**
- It is very simple to **understand and use.**
- The Waterfall model is the earliest SDLC approach that was used for software development.
- The waterfall Model illustrates the software development process in a **linear sequential flow**.
- This means that any phase in the development process begins only if the previous phase is complete.
- In this waterfall model, **the phases do not overlap**


Requirement Specifications
• The requirements for the software in terms of both design and functionality is captured
System Design
•Once the requirements are finalized, a blueprint of the system is drawn to facilitate the process of implementation
Design Implementation
•The blue print of the design is verified and the implementation of the system begins according to the blueprint
Verification and Test
•The implemented system is now verified and tested by the team. Once the testing is comple, ted reversing the system to implement further changes in requirements cannot be done
System Deployment
•Setting up of the system or software after a pilot run and testing is done
Software Maintenance
• Regular updating, verification and debugging of the software

When to use Waterfall Model

- This model is used only when the requirements are very well known, clear and fixed.
- Product definition is stable.
- Technology is understood.
- There are no ambiguous requirements
- Ample resources with required expertise are available freely
- The project is short.

Advantage & Disadvantage of Waterfall model

Advantage:

- the waterfall model are that documentation is produced at each phase and that it fits with other engineering process models.
- Disciplined approach
- Careful checking by the Software Quality Assurance Group at the end of each phase.(or Testing in each phase.)
- Documentation available at the end of each phase
- Linear model..
- Easy to understand and implement.
- Identifies deliverables and milestones

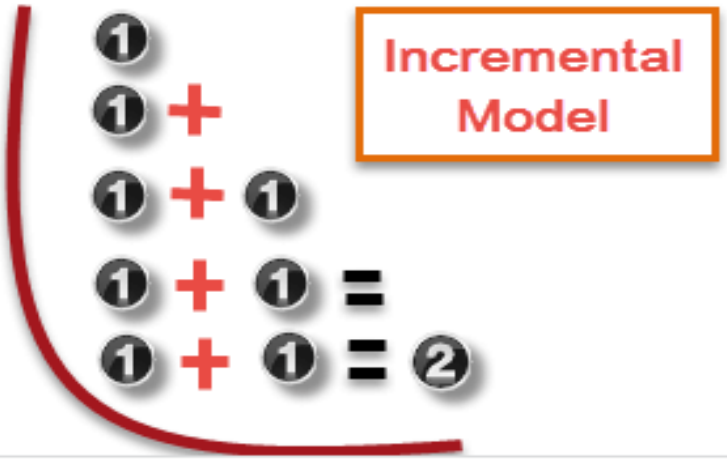
Disadvantage:

- the waterfall model is the difficulty of accommodating change after the process is underway. One phase has to be complete before moving onto the next phase.
- Unidirectional
- Unclear requirements lead to confusion.
- Client's approval is in the final stage.
- Difficult to integrate risk management
- Idealised, doesn't match reality well.
- Doesn't reflect iterative nature of exploratory development.

Verification	Validation
As per IEEE-STD-610: The process of evaluating the software to determine whether the products of a given development phase satisfy the conditions imposed at the beginning of that phase.	As per IEEE-STD-610: The process of evaluating the software during or at the end of the development process to determine whether it satisfies specified requirements.
It is the process, to ensure whether we are developing the product accordingly or not.	It is the process, to validate the product which we have developed is right or not.
Verification is verifying the documents.	Validation is to validate the actual and expected output of the software.
Activities involved here are inspections, reviews and walkthroughs.	Activities involved in this is testing the software application.
It is a low-level activity.	It is a high-level activity.
Verification is a static method of checking documents and files.	Validation is a dynamic process of testing the real product.
Am I building a product right?	Am I building a right product?

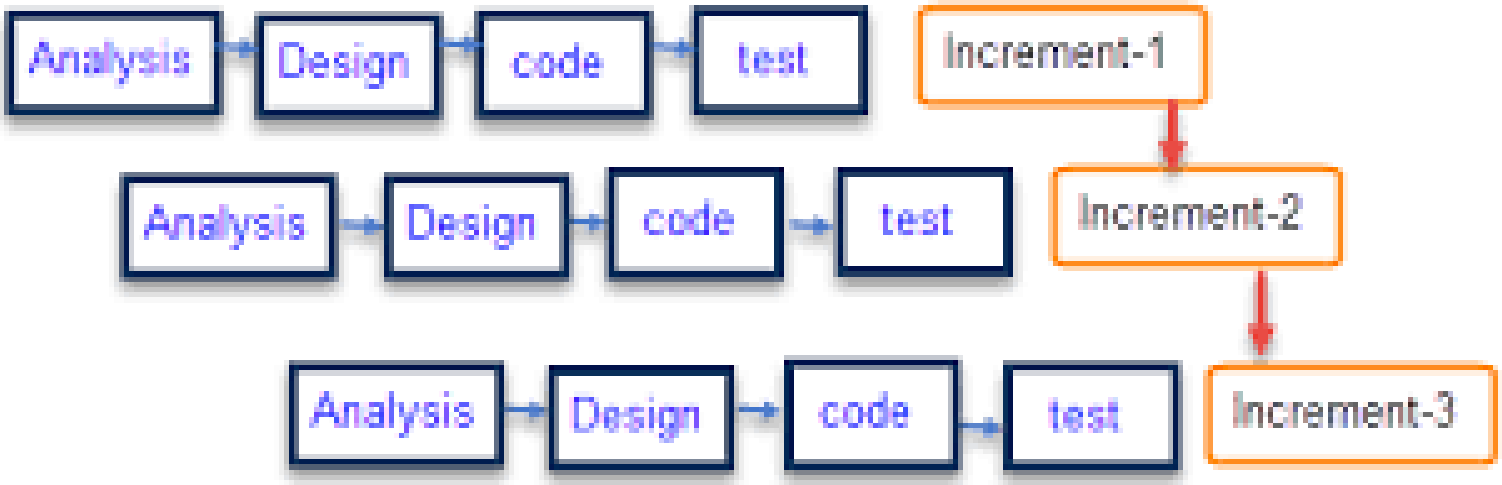
Incremental Model

- Incremental Model is a process of software development where **requirements are broken** down into multiple standalone modules of software development cycle.
- Incremental development is done in steps from **analysis, design, implementation, testing/verification, and maintenance.**



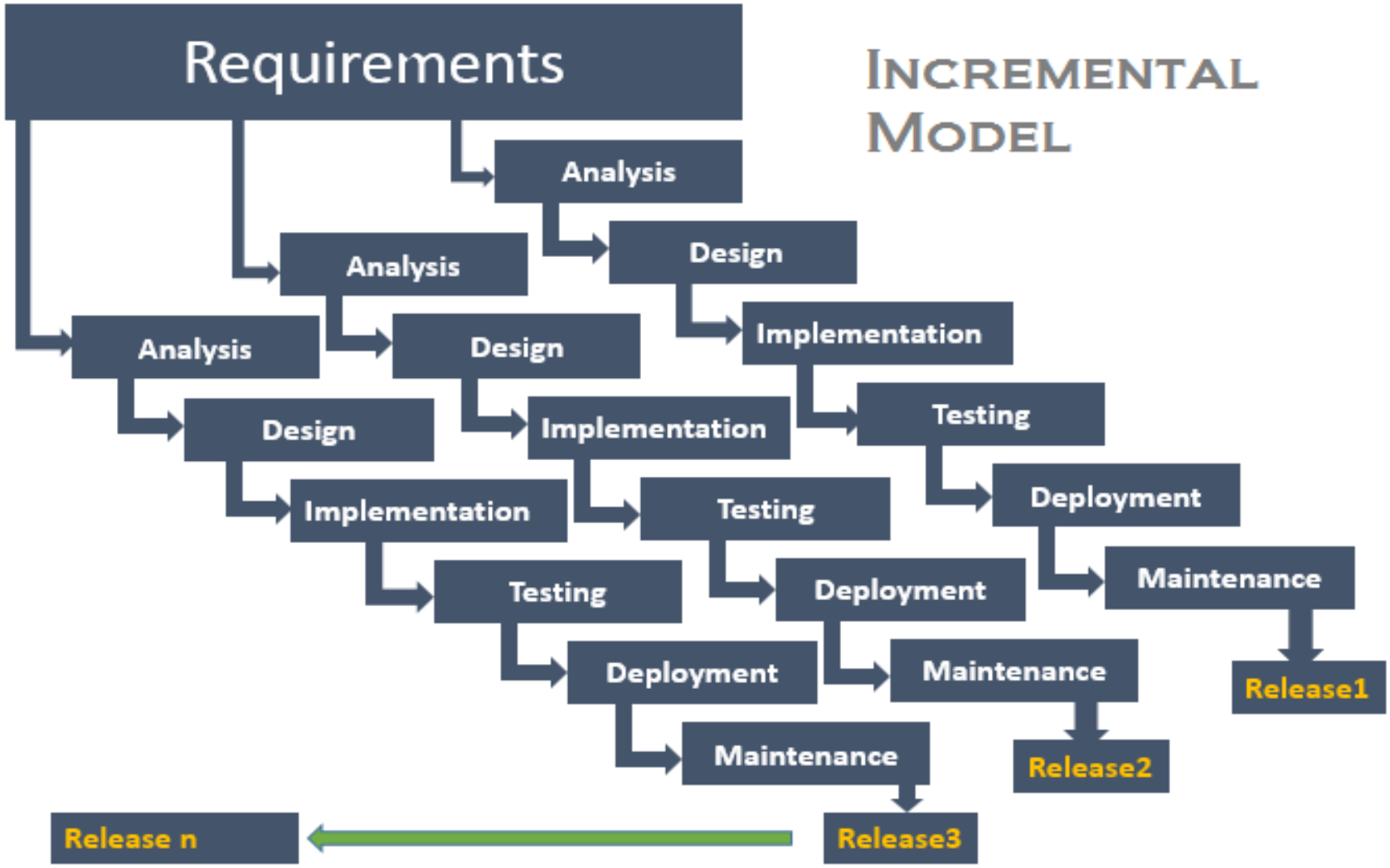
Incremental Model

- It is also Known as **Multi Water fall Model.**

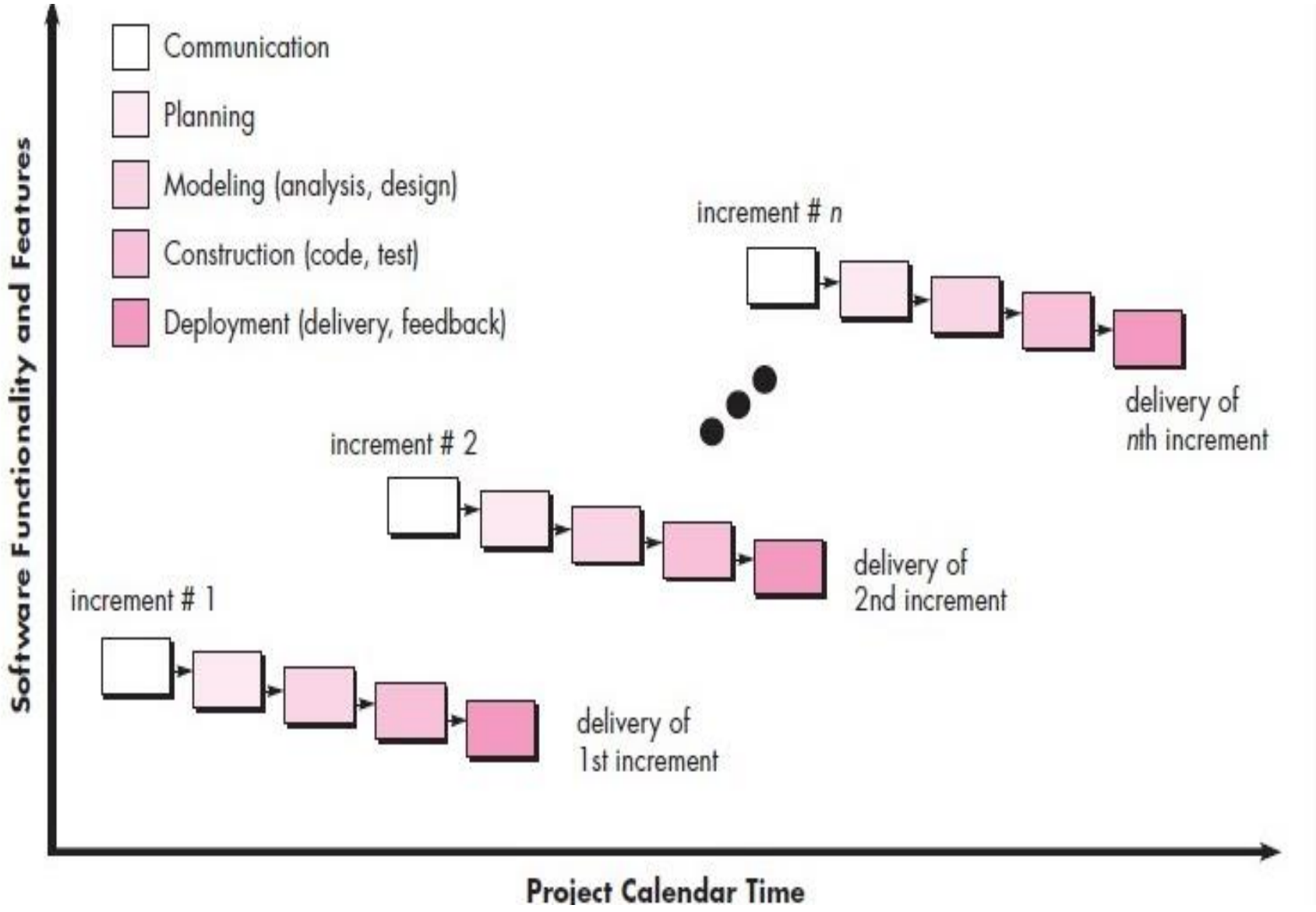


Incremental Model

Incremental Model



Incremental Model



Incremental Model

- **Incremental model** is used when **enough staffing** is **unavailable**.
- Each iteration passes through the **requirements, design, coding and testing** phases.
- Each subsequent release of the system **adds function** to the previous release **until all designed** functionality has been implemented.
- The system is put into production when the **first increment is delivered**.
- Once the core product is **analyzed by the client**, there is plan development for the next increment.

When to use Incremental models?

- Requirements of the system are **clearly understood**
- When **demand for an early release** of a product arises
- When software engineering **team are not very well skilled or trained**
- When **high-risk features** and goals are involved

Incremental model

- **Advantages of Incremental model:**
- Generates working software quickly and early during the software life cycle.
- This model is more flexible – less costly to change scope and requirements.
- It is easier to test and debug during a smaller iteration.
- In this model customer can respond to each built.
- Lowers initial delivery cost.
- Easier to manage risk because risky pieces are identified and handled during it'd iteration.

Advantage & Disadvantage of Incremental Model

Merits:

- Develop high-risk or major functions first
- Each release delivers an operational product
- Customer can respond to each build
- Lowers initial delivery cost
- Initial product delivery is faster
- Customers get important functionality early
- Risk of changing requirements is reduced
- Easier to test and debug during a smaller iteration.
- Each iteration is an easily managed milestone.

Demerits:

- Requires good planning and design
- Requires early definition of a complete and fully functional system to allow for the definition of increments
- Well-defined module interfaces are required (some will be developed long before others)
- Total cost of the complete system is not lower
- Each phase of an iteration is rigid and do not overlap each other

Evolutionary Models

- The word **Evolutionary derived** from the word **“Evolution”**.
- It is a combination of **iterative and incremental** apporach for developing a good software.
- Evolutionary process models are **Iterative** (An iterative process is a process for calculating a desired results by means of a repeated cycle of operations) and incremental type models.
- They allow to develop more complete versions of the software.

Evolutionary Process Models

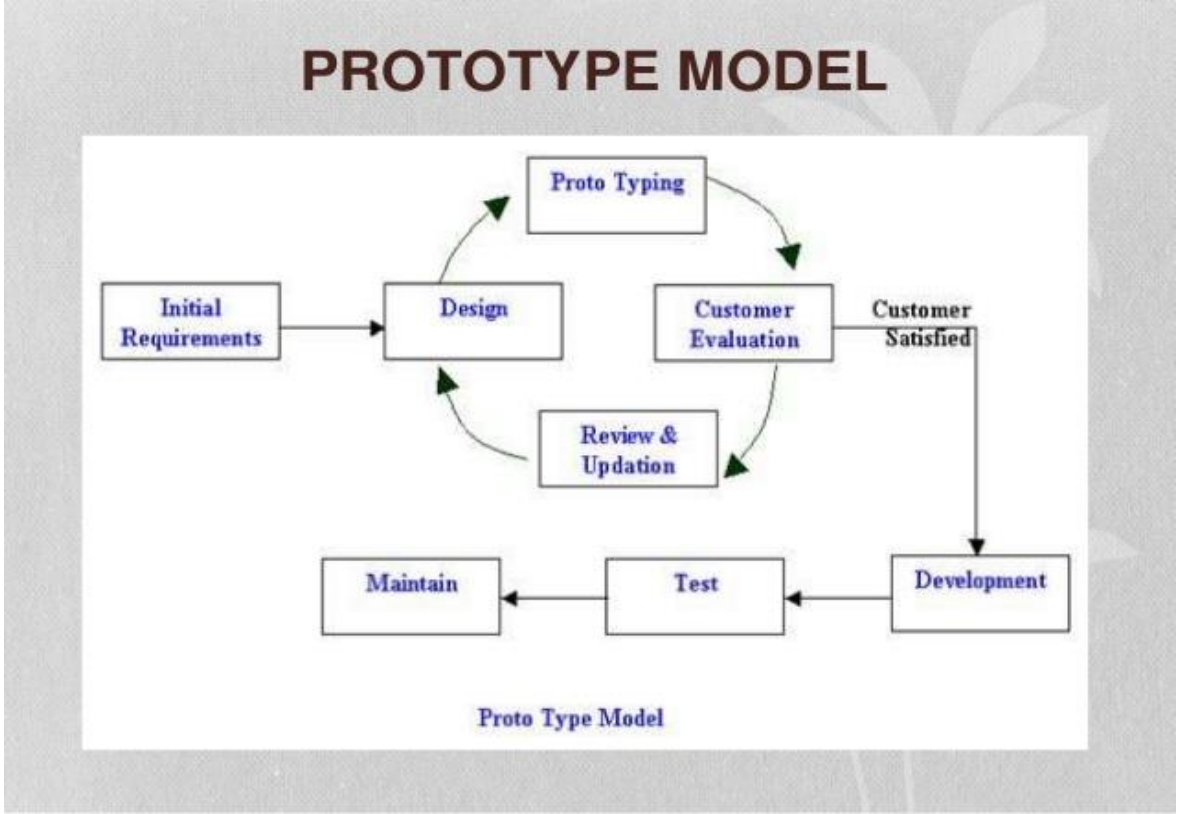
1. Prototyping Model
2. Spiral Model
3. Concurrent Development Model

Prototyping Model

- The Prototyping Model is one of the most popularly used Software Delvelopment Life Cycle Models (SDLC models)
- The Prototyping Model should be used when the **requirements of the product are not clearly understood**.
- It can also be used if **requirements are changing quickly.**
- Prototyping model also supports **risk handling**, but the risks must be identified before the start of the development work of the project.
- But in real life **project risk may occur after the development work starts**.

Prototyping Model

- This model is used when the **customers do not know** the exact project requirements.
- A prototype of the end product is **first developed, tested and refined as per the customer feedback**.



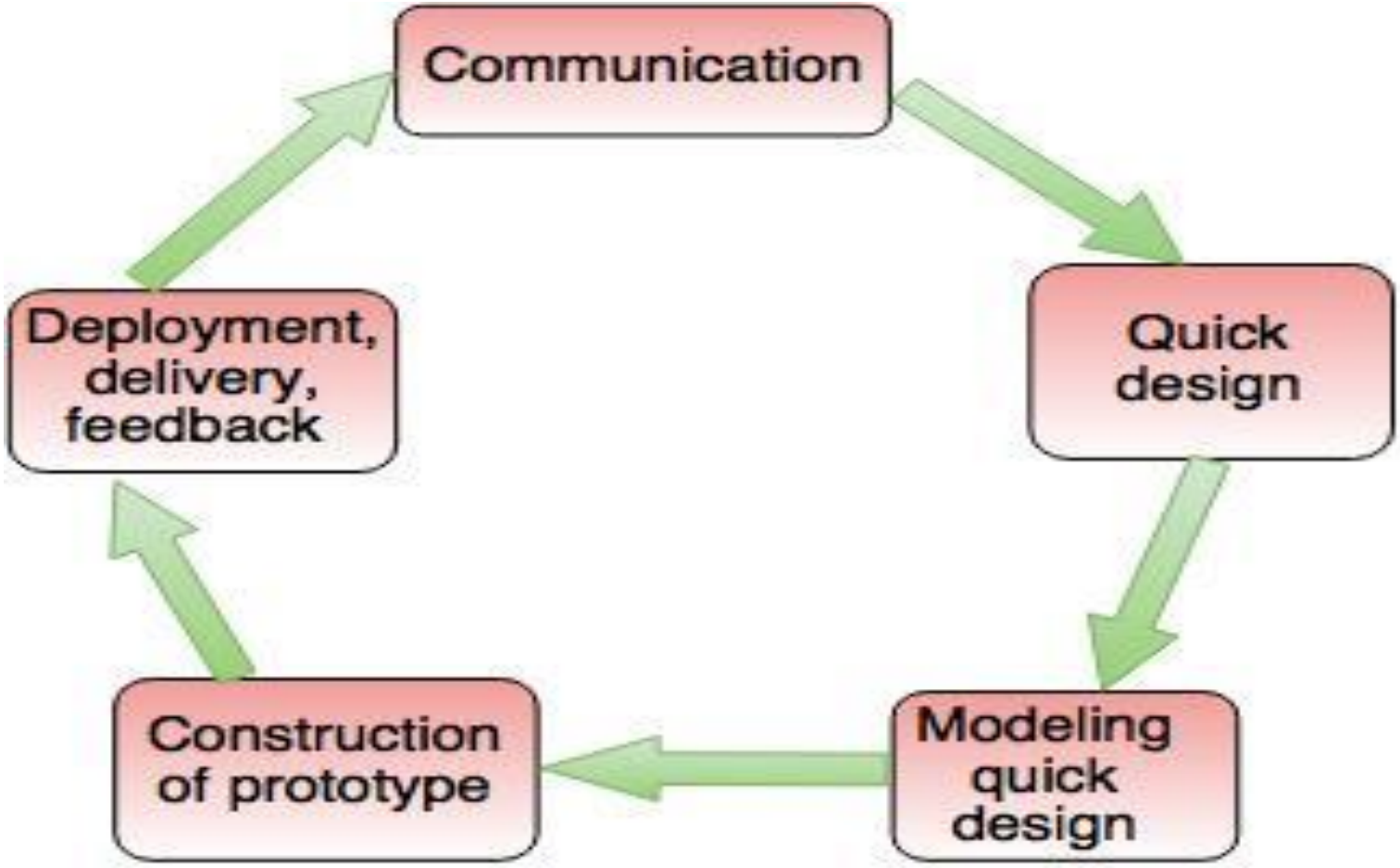


Fig. - The Prototyping Model

Prototyping Model

1. Communication

- In this phase, developer and customer meet and discuss the overall objectives of the software.

2. Quick design

- Quick design is implemented when requirements are known.
- It includes only the important aspects like **input and output format of the software.**
- It focuses on those aspects which are visible to the user rather than the detailed plan.
- It helps to **construct a prototype.**

Prototyping Model

3. Modeling quick design

- This phase gives the **clear idea about the development** of software because the software is now built.
- It allows the developer to better understand the exact requirements.

4. Construction of prototype

- The prototype is evaluated by the customer itself.

5. Deployment, delivery, feedback

- If the user is **not satisfied** with current prototype then it refines according to the requirements of the user.
- The process of refining the prototype is repeated until all the requirements of users are met.
- When the **users are satisfied** with the developed prototype then the system is developed on the basis of final prototype.

Prototyping Model

Advantages

- **Errors** are detected much earlier.
- Ensures a **greater level of customer satisfaction** and comfort.
- It identifies the **missing functionality** easily.
- It also identifies the confusing or difficult functions.

Disadvantages

- The **client involvement** is more and it is **not always considered by the developer.**
- It is a **slow process** because it takes more time for development.
- Many changes can disturb the rhythm of the development team.

Spiral Model

- **Barry Boehm.B** introduced **Sprial Model in 1986**.
- Spiral model is one of the most important Software Development Life Cycle model, which provides **support for risk handling**.
- In its **diagrammatic** representation, it looks like a **spiral with many loops**.
- Each loop of the spiral is called a phase of the software development process.
- This model is much **more flexible** compared to other SDLC models.

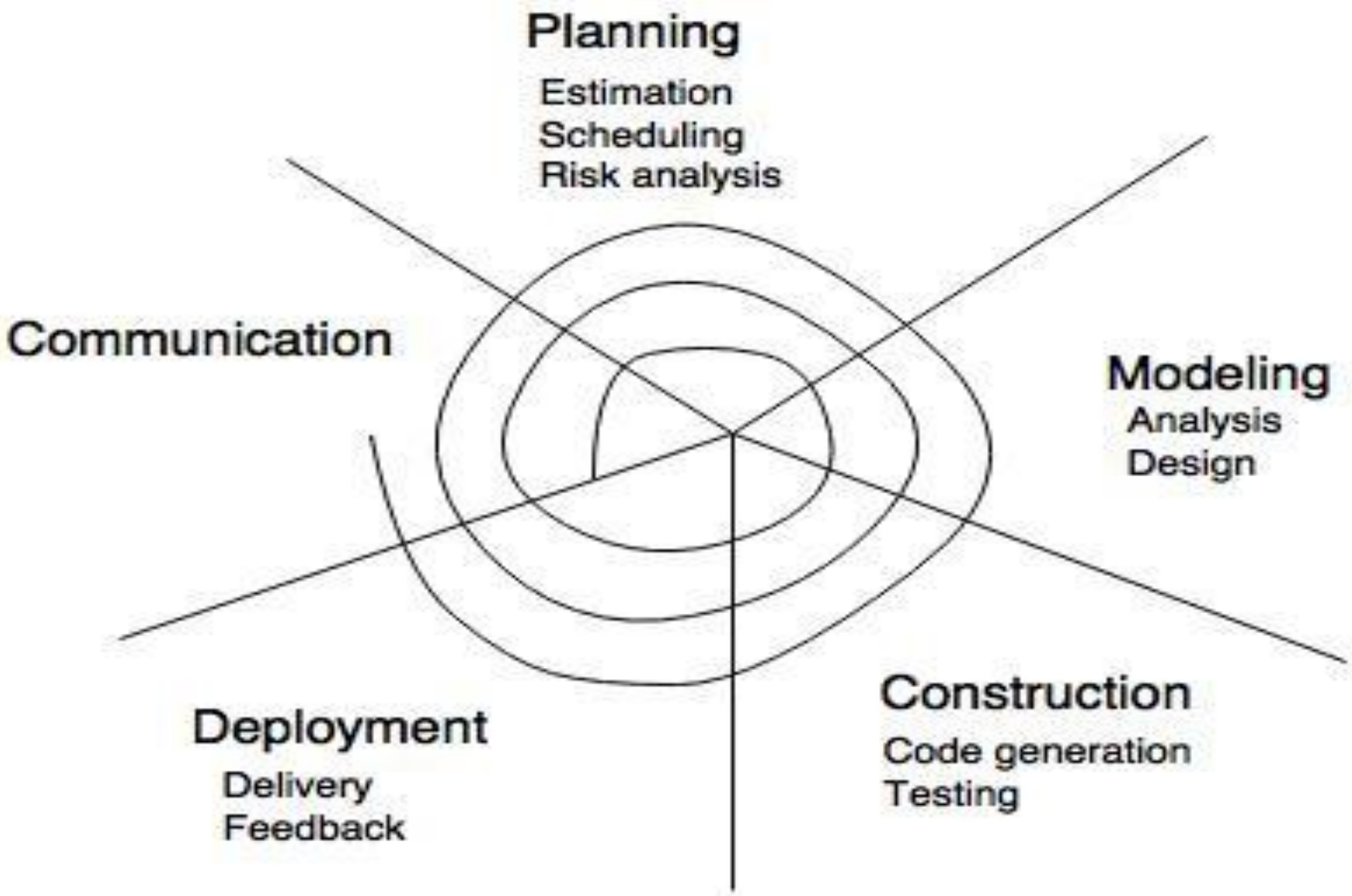


Fig. - The Spiral Model

Spiral Model

- **Communication –** Gather Requirements from Customers
- **Planning –** Plan to be followed will be created. It describes the technical task to be conducted, risk ,required resources, work schedule etc.,
- **Modeling –** A model will be created to better understand the requirements
- **Construction –** Code generated and tested
- **Deployment** - Complete or partial complete version of software is given to the customer to evaluate and they give feed back based on the evaluation

Spiral Model

Advantages:

- It **reduces high amount** of risk.
- It is **good for large and critical** projects.
- It gives strong approval and documentation control.
- In spiral model, the **software is produced** early in the life cycle process.

Disadvantages:

- It can be **costly to develop** a software model.
- It is **not used for small** projects.

Concurrent Development Model

- The concurrent development model is called as **concurrent model.**
- The communication activity has completed in the first iteration and exits in the **awaiting changes state**.
- The modeling activity completed its **initial communication** and then go to the underdevelopment state.
- If the customer specifies the change in the requirement, then the modeling activity moves from the **under development state into the awaiting change state**.
- The concurrent process model activities moving from one state to another state.

Concurrent Development Model

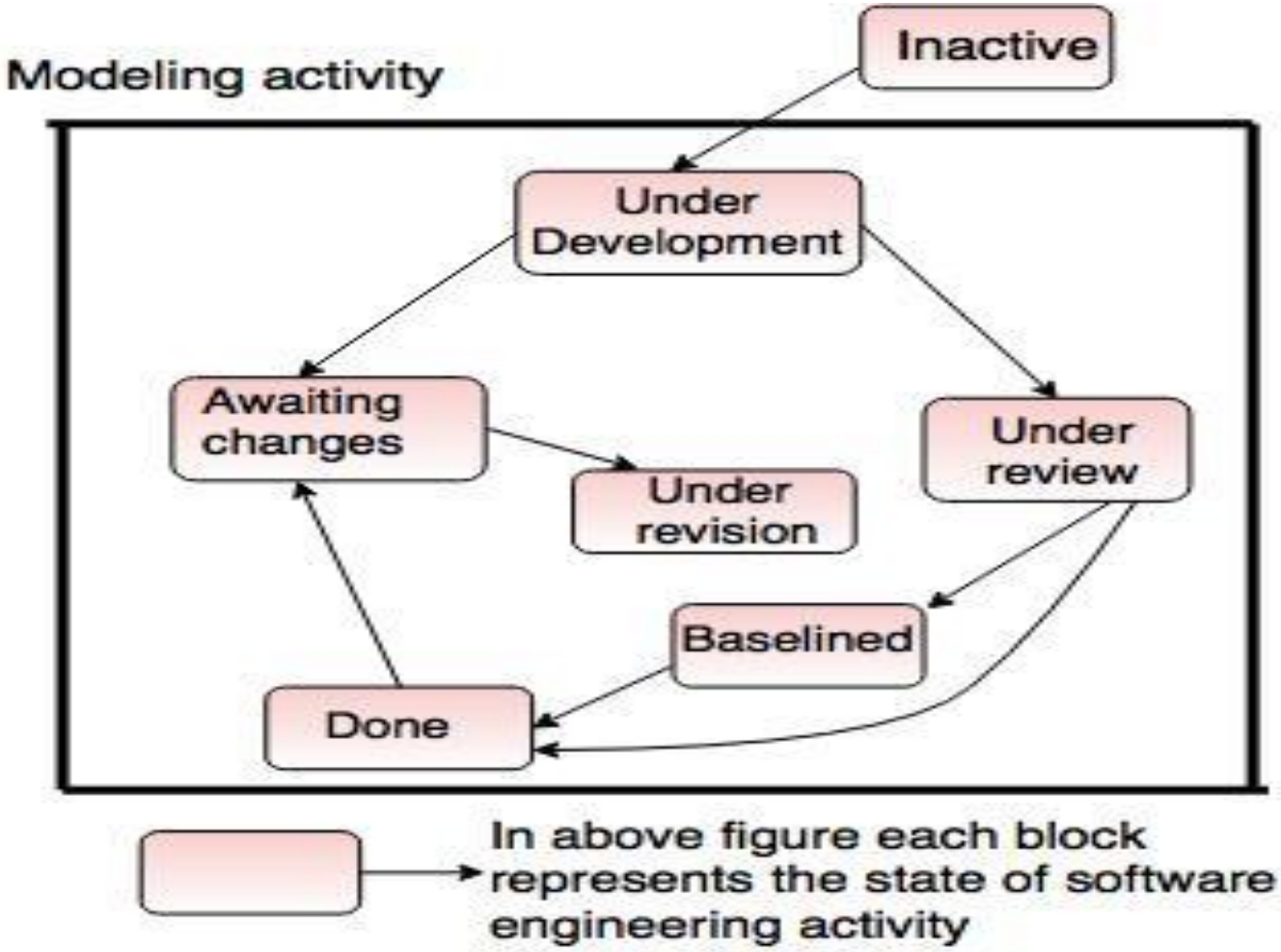


Fig. - One element of the concurrent process model

Advantages of the concurrent development model

- This model is applicable to **all types of software development processes.**
- It is **easy for understanding** and use.
- It gives **immediate feedback** from testing.
- It provides an accurate picture of the current state of a project.

Disadvantages of the concurrent development model

- It needs **better communication between the team members.** This may not be achieved all the time.
- It requires to **remember the status** of the different activities.

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

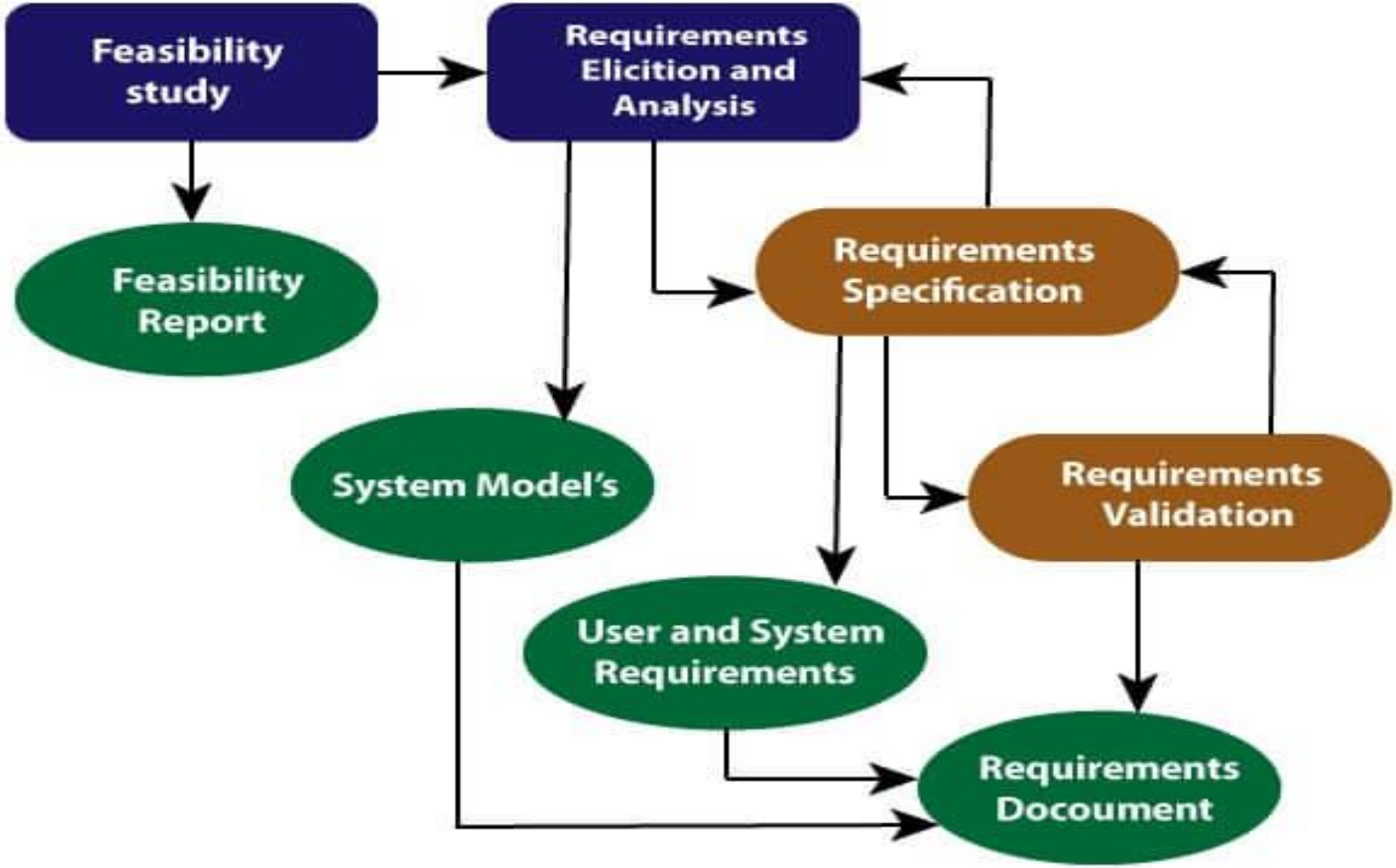
Requirement Engineering

- 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.

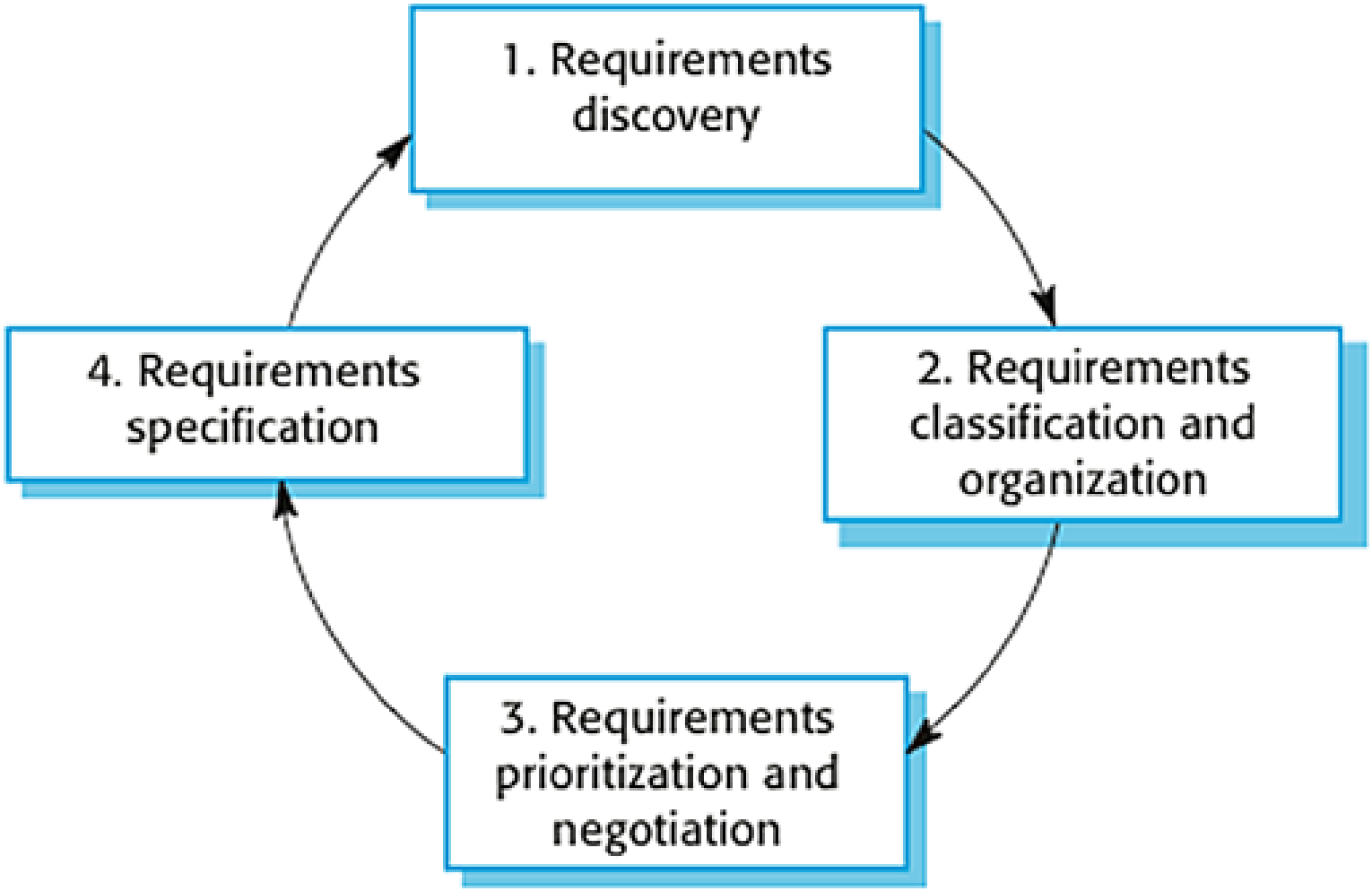
Types of Feasibility:

- **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

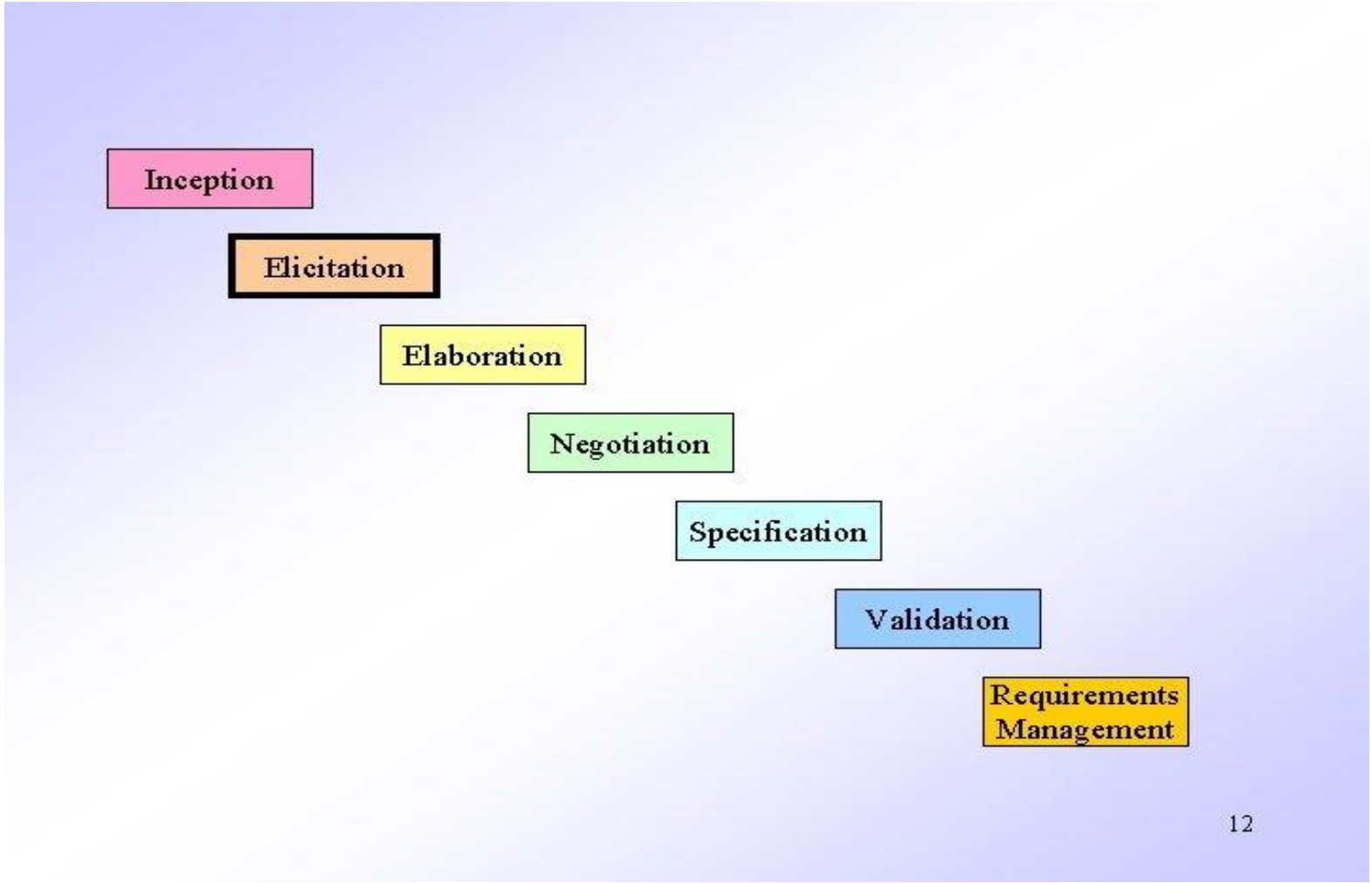
Requirements Validation Techniques

- **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

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**

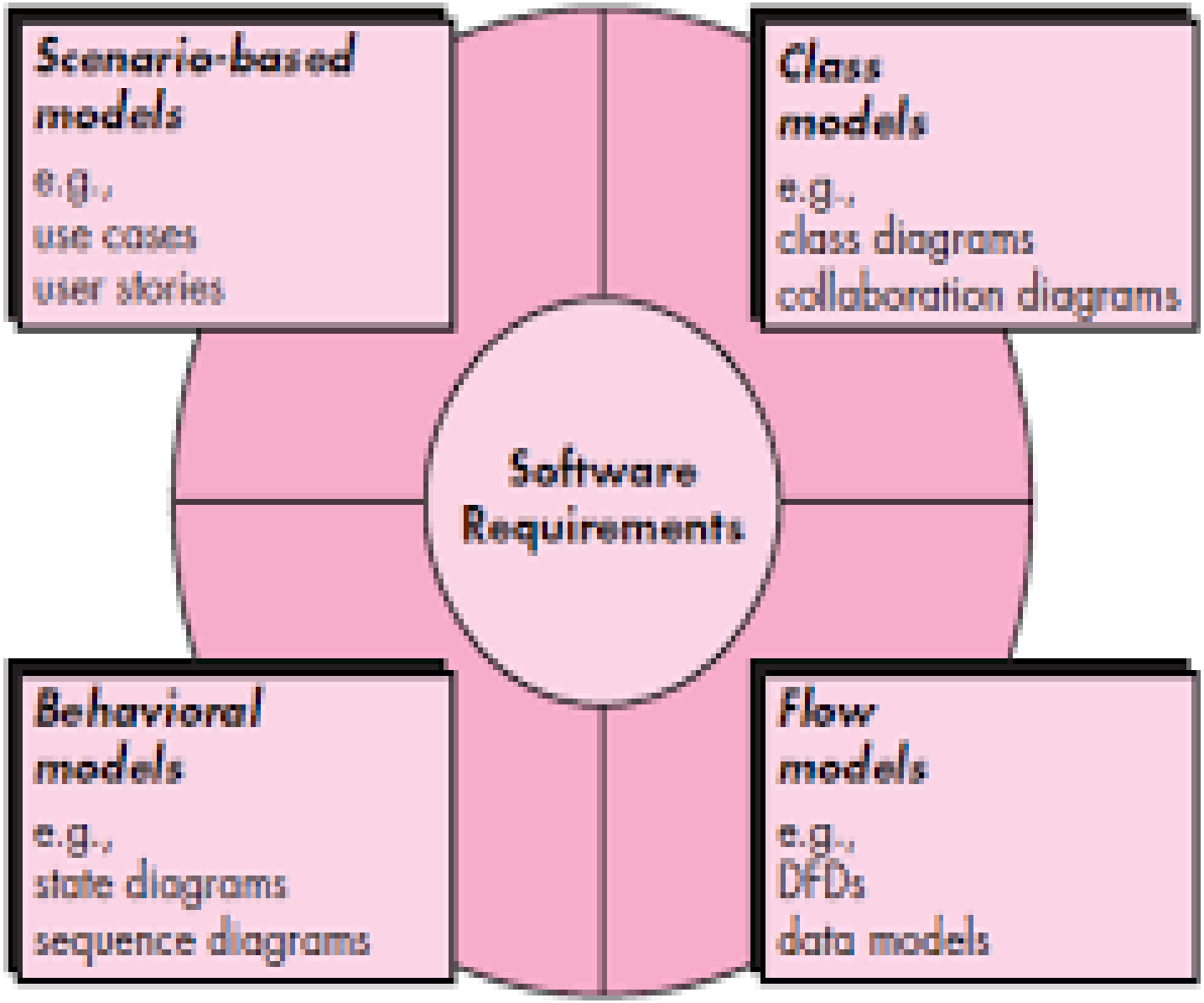


Fig 6.3: Elements of Analysis model

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

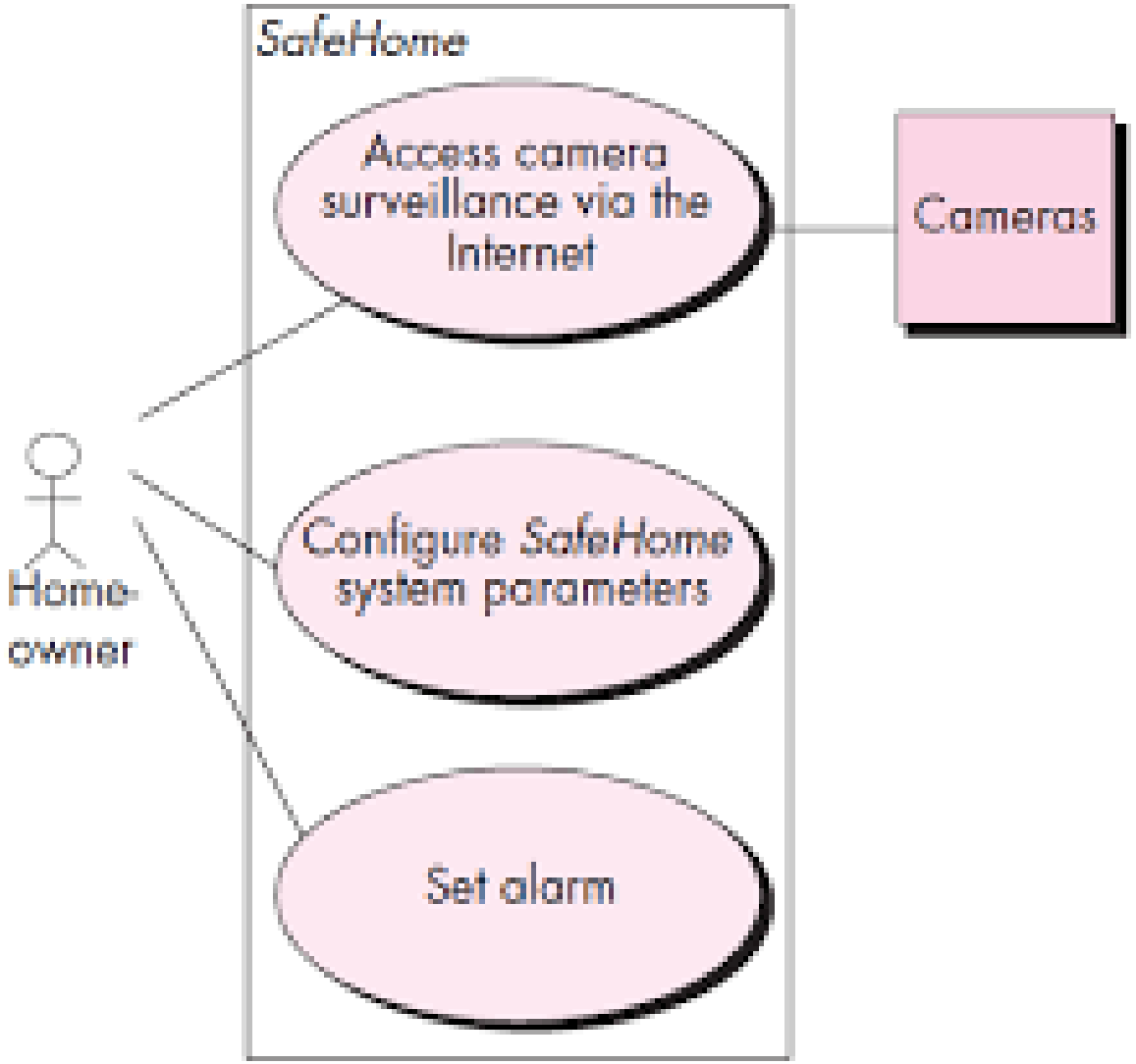
Can the actor take some other action at this point?

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

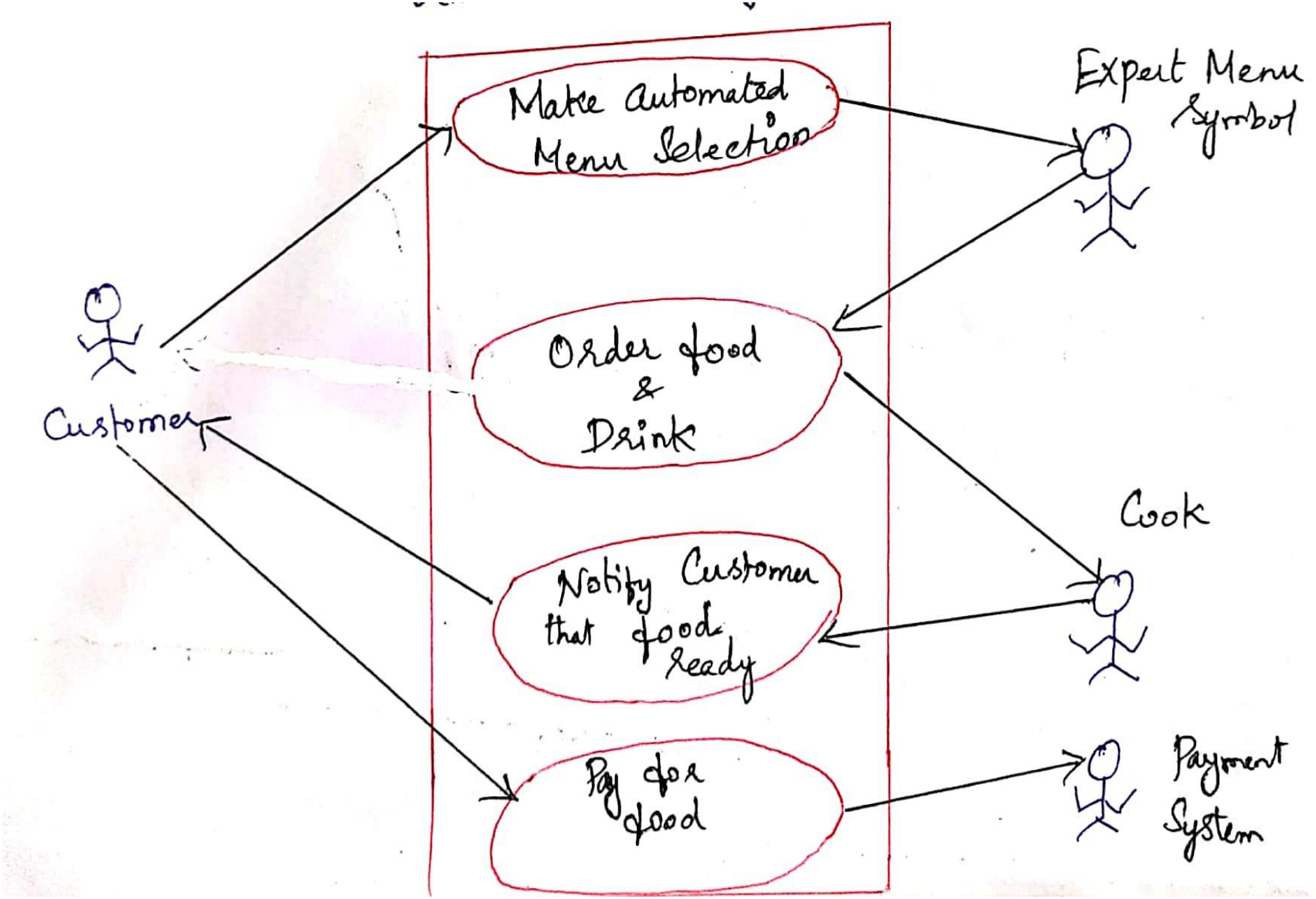
Is it possible that the actor will encounter some other behavior at some point? If so, what?

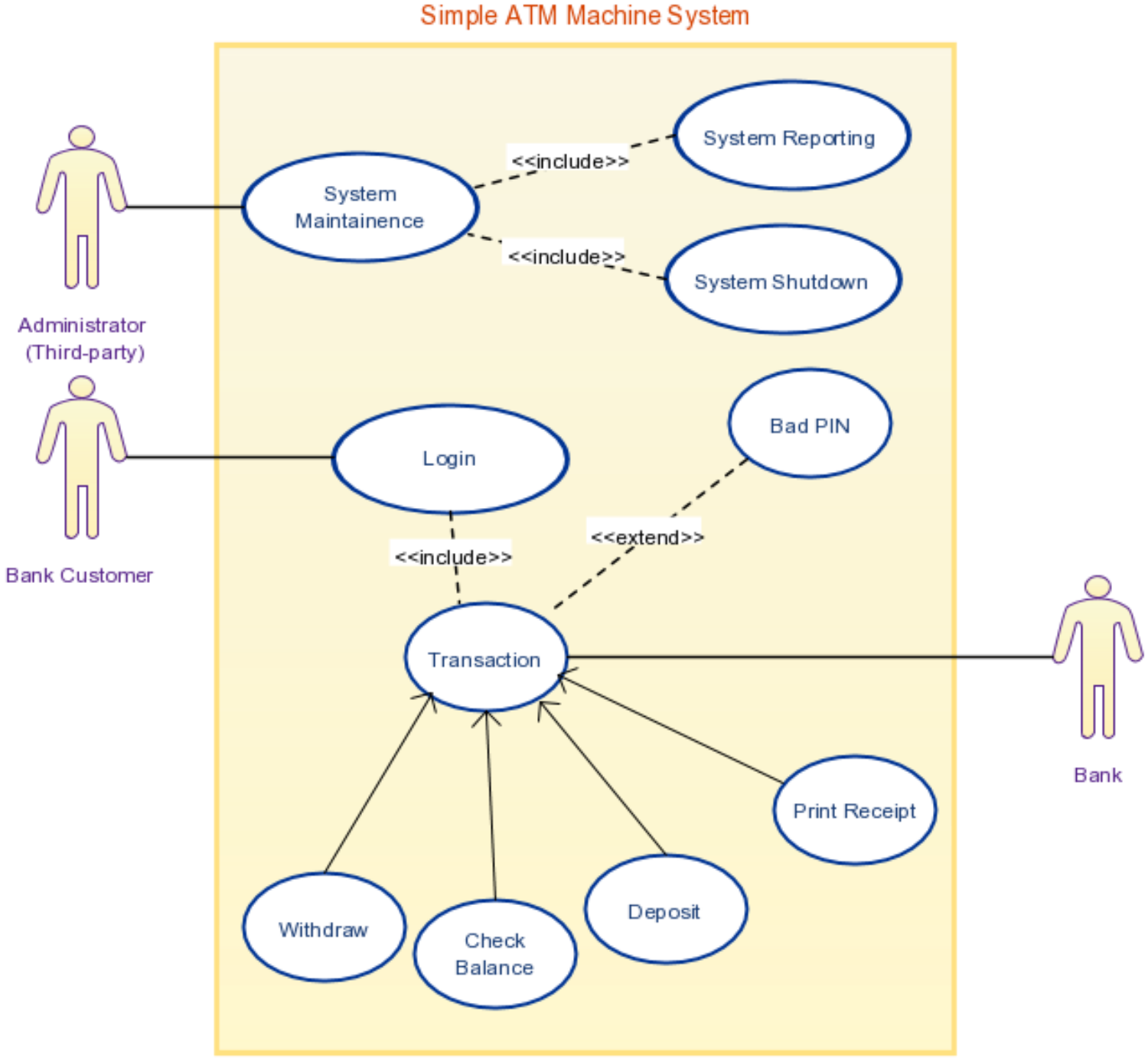
Writing a Formal Use Case

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:

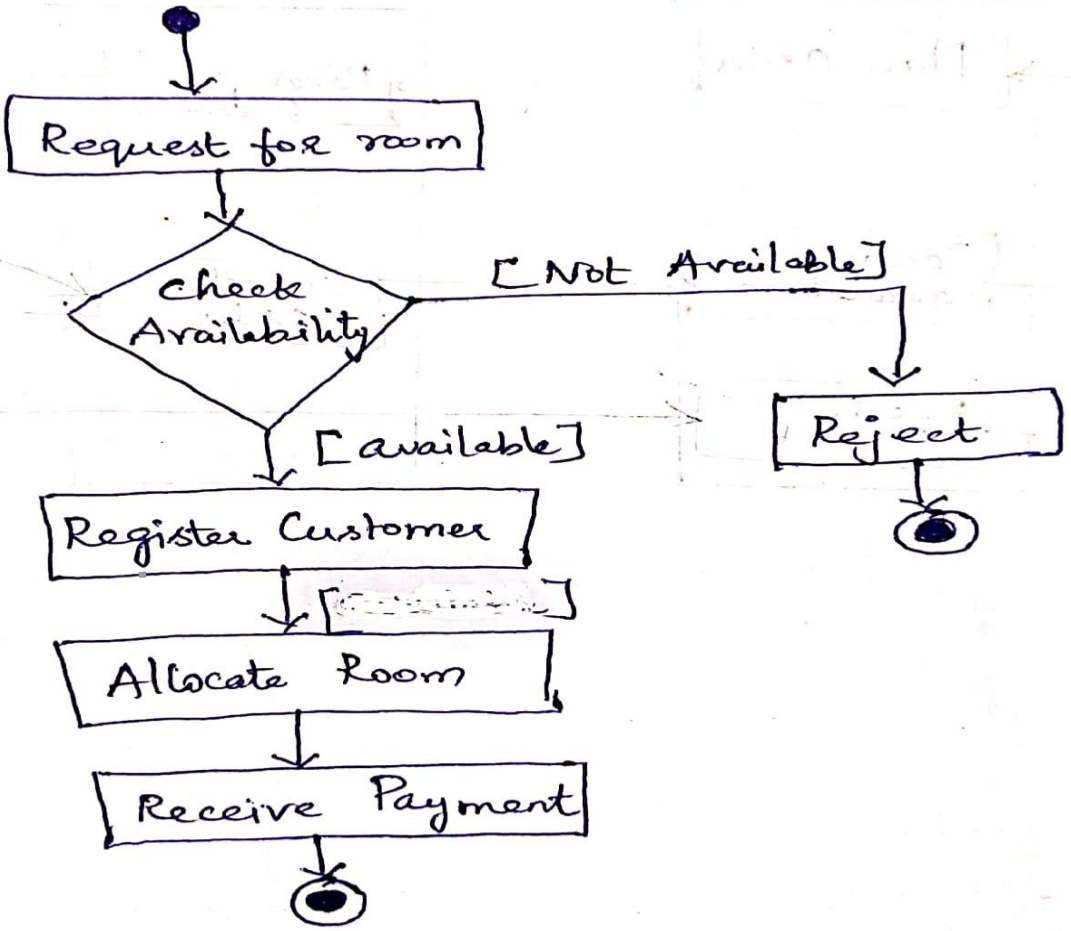
1. What mechanisms protect unauthorized use of this capability by employees of *SafeHome Products*?
2. Is security sufficient? Hacking into this feature would represent a major invasion of privacy.
3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
4. 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

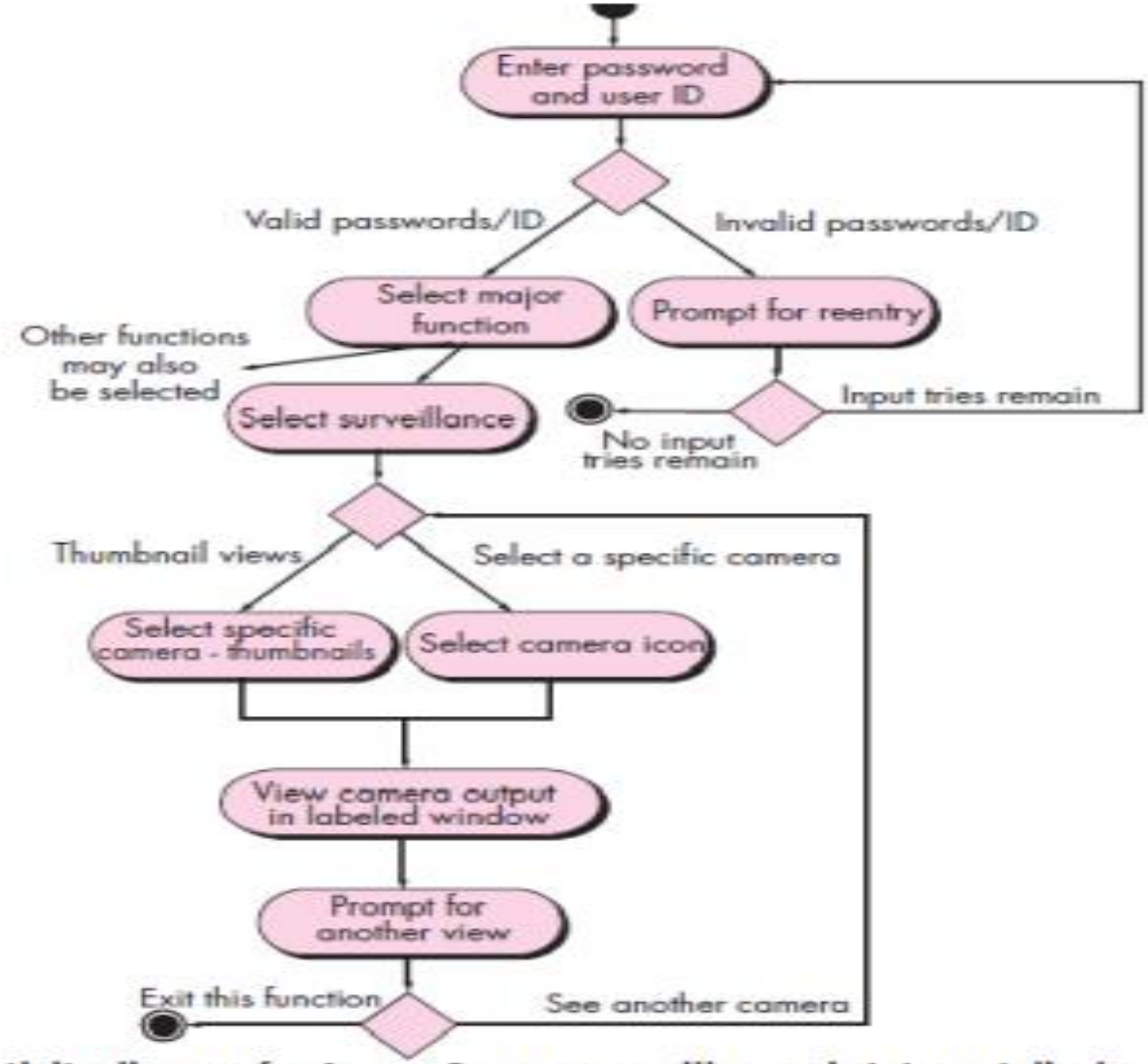


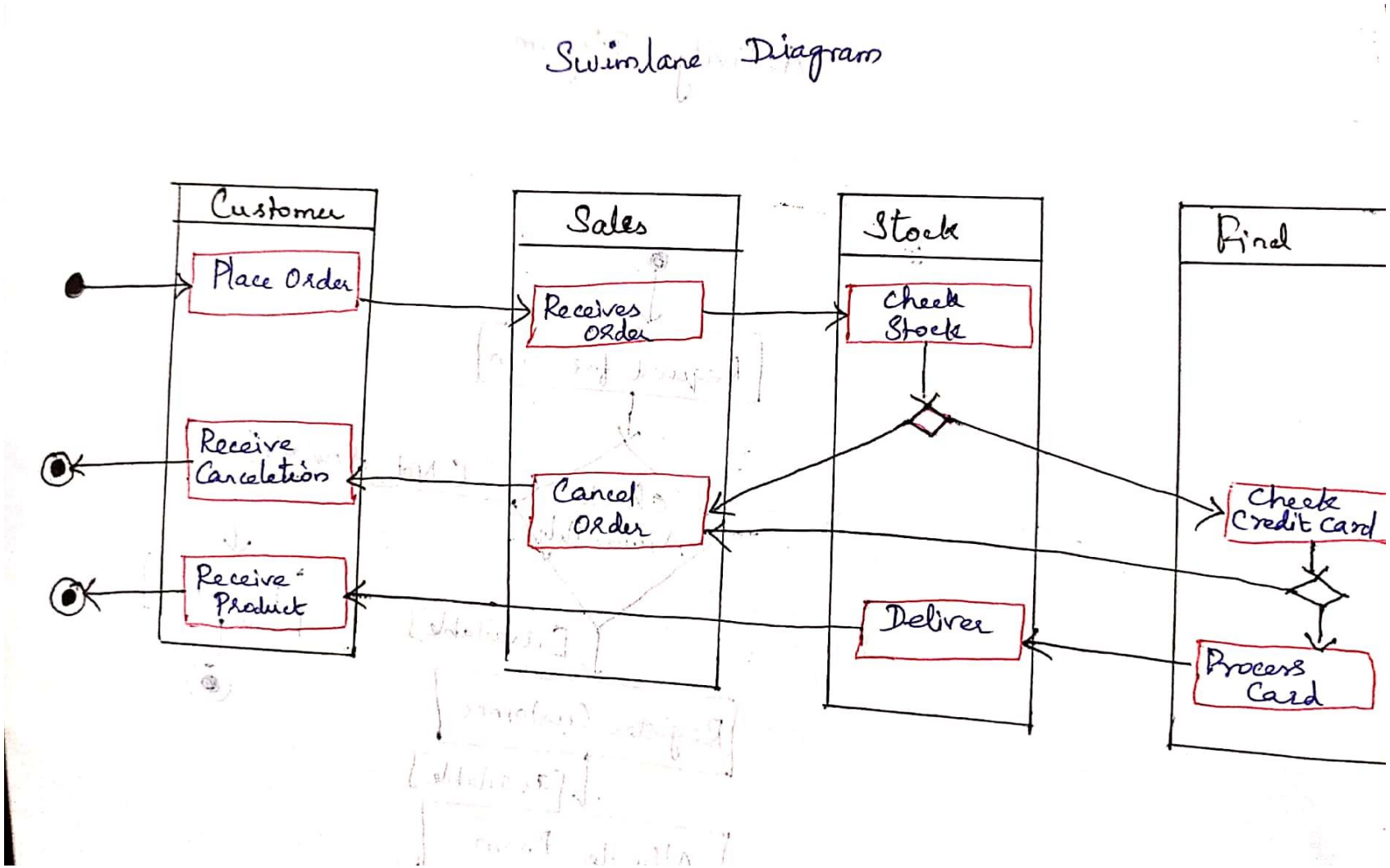
Fig 6.5: Activity diagram for Access Camera surveillance via Internet display cams

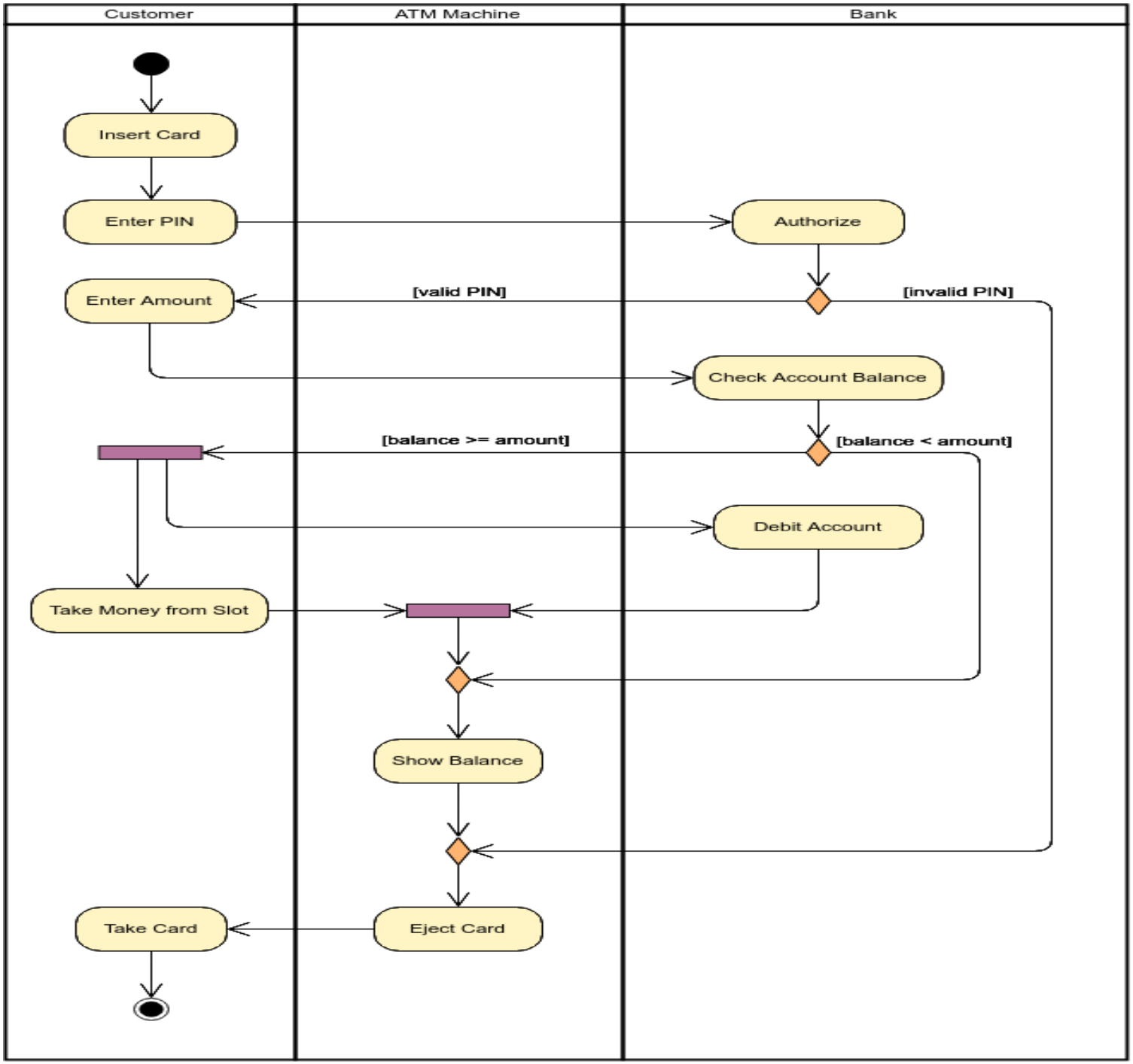
Scenario Based Modeling

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**

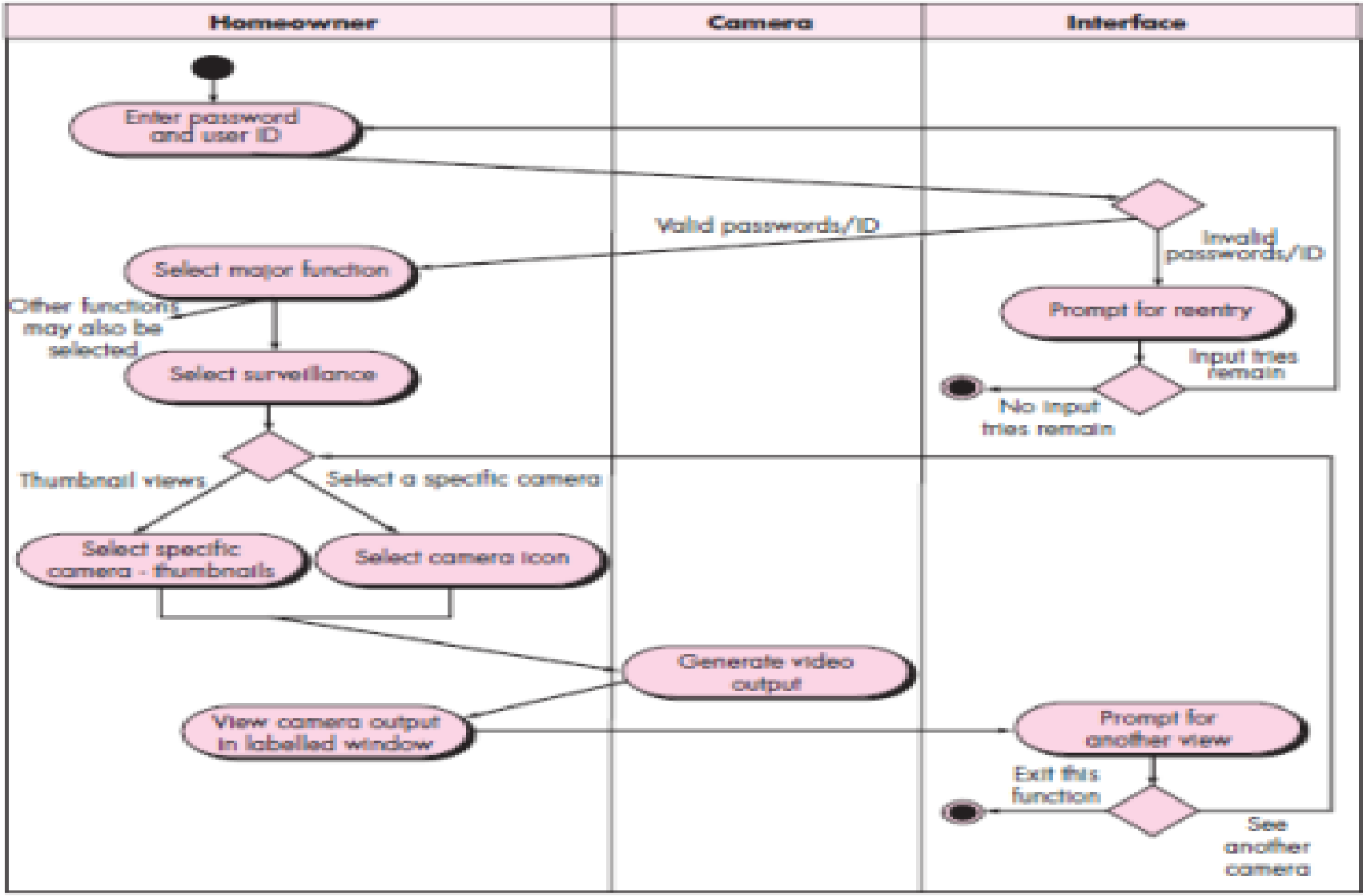
Swimlane Diagram




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.

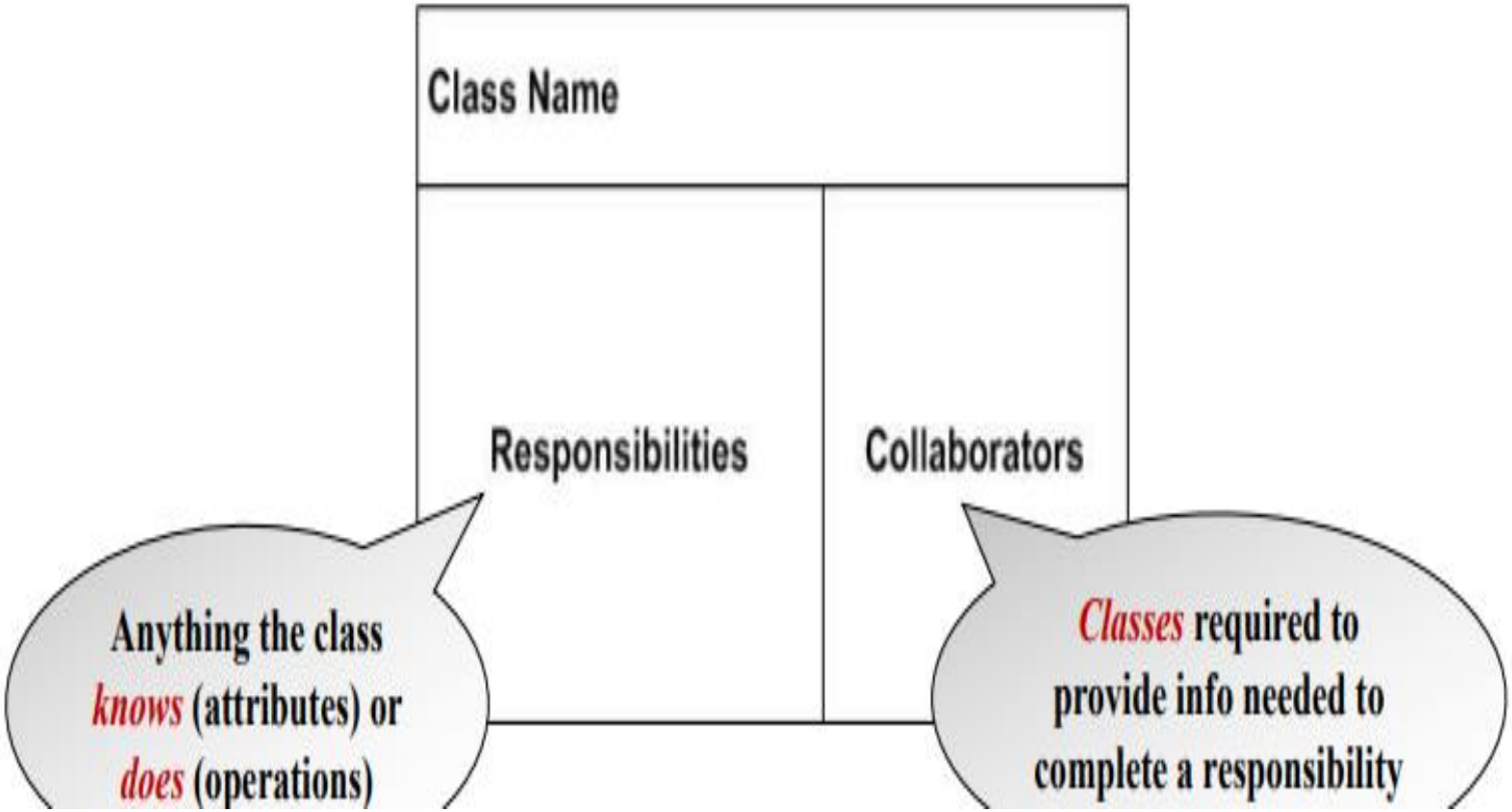
- These functions are accomplished by operating on attributes and/or associations

Defining Operations

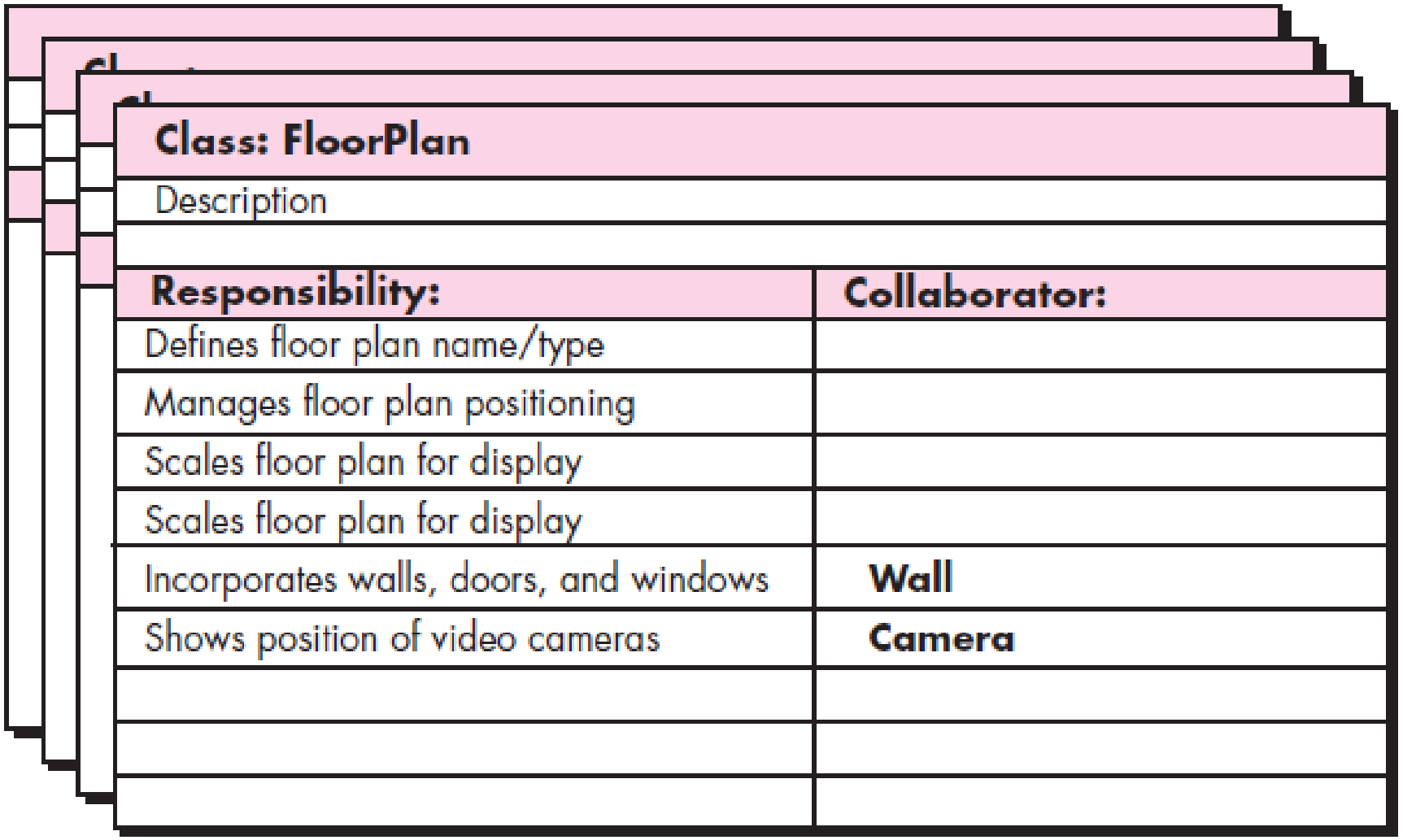
Class-Responsibility-Collaborator (CRC)

- 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.

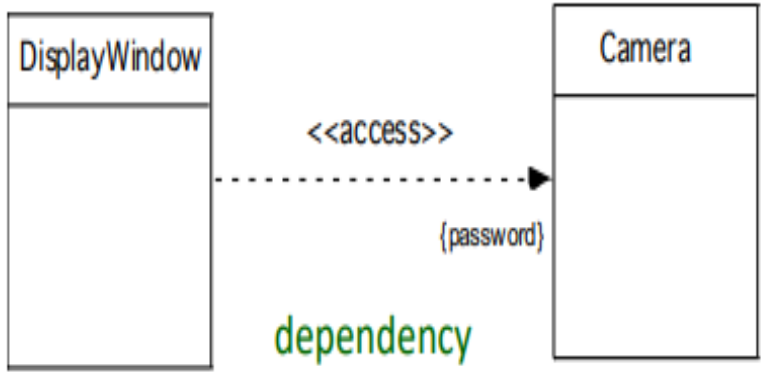
1. System intelligence should be **distributed across classes** to best address the needs of the problem.
2. Each responsibility should be **stated as generally as possible.**
3. **Information and the behavior** related to it should reside **within the same class**.
4. Information about **one thing should 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

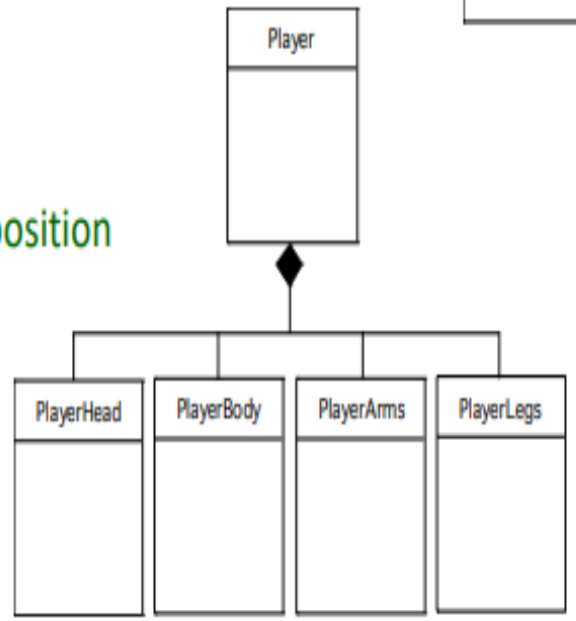
- A class may collaborate with other classes to fulfill responsibilities
  - If a class cannot fulfill every single responsibility itself, it must interact with another class
- Collaboration refers to identifying relationships between classes
  - is-part-of relationship
    - Aggregation
  - has-knowledge-of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

Relationships between Classes

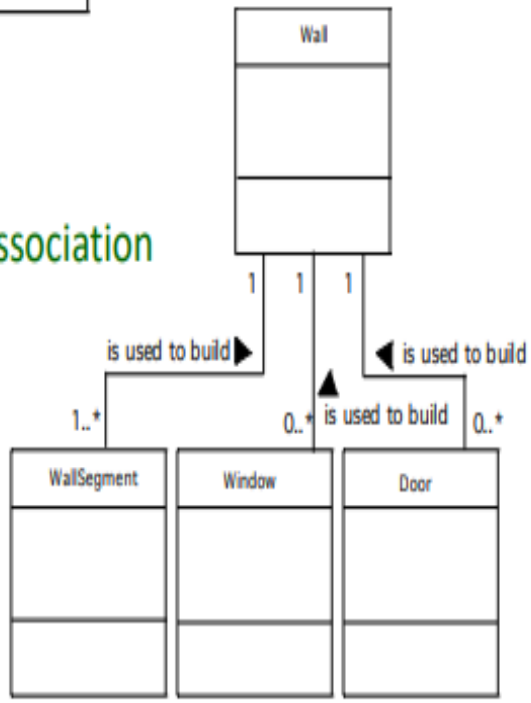


dependency

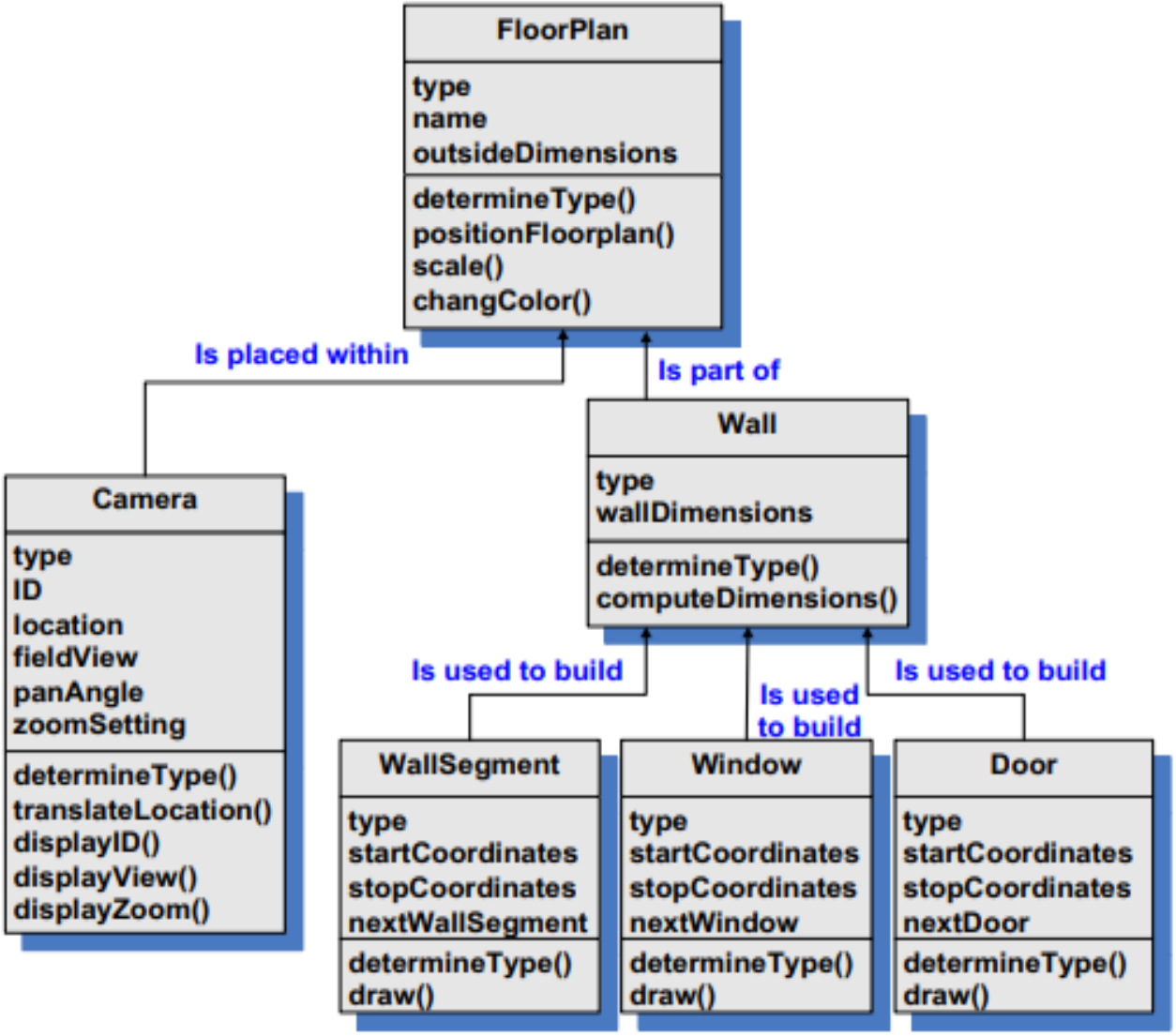
composition



association

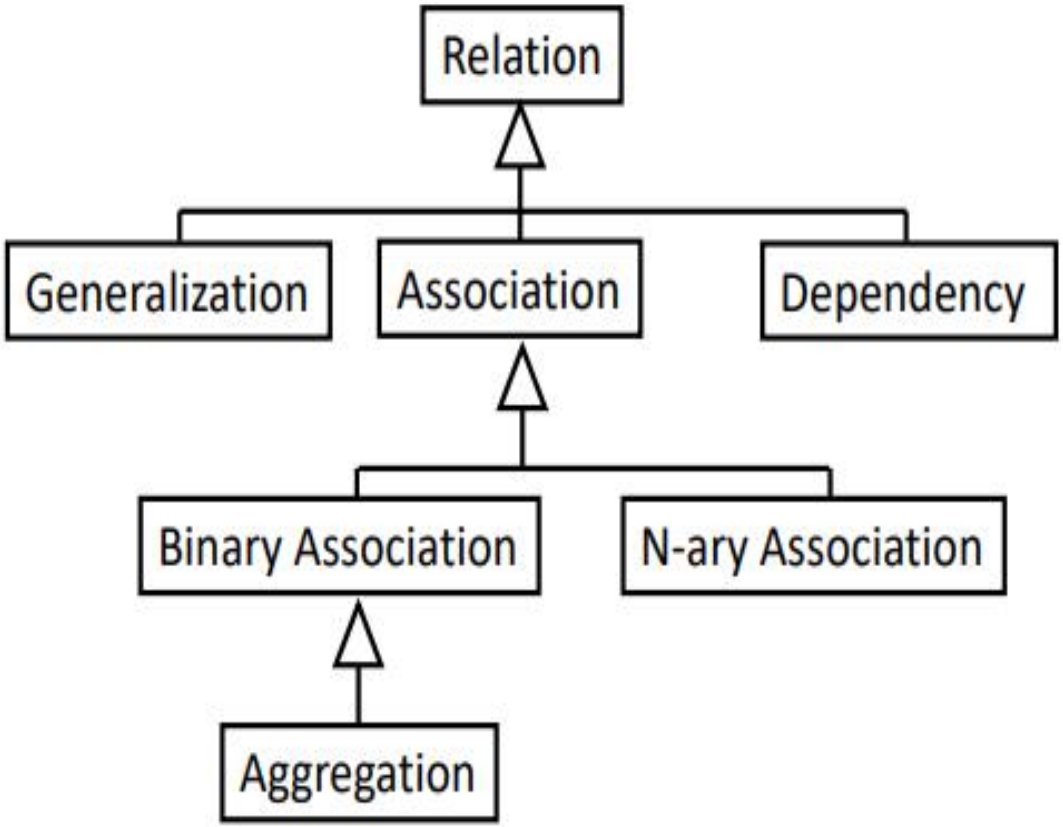


Class Diagram



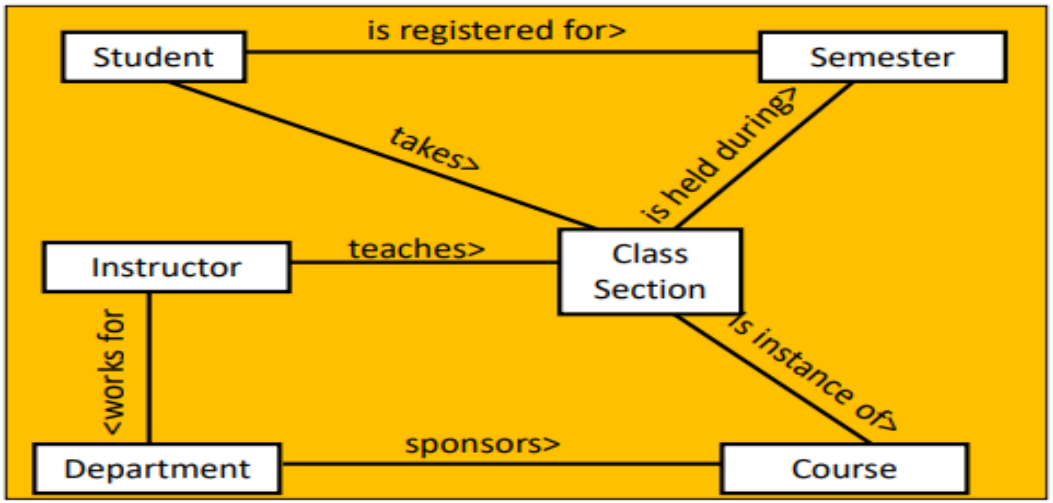
Type of Relationships in Class Diagrams

- Class diagrams show relationships between classes.



Associations

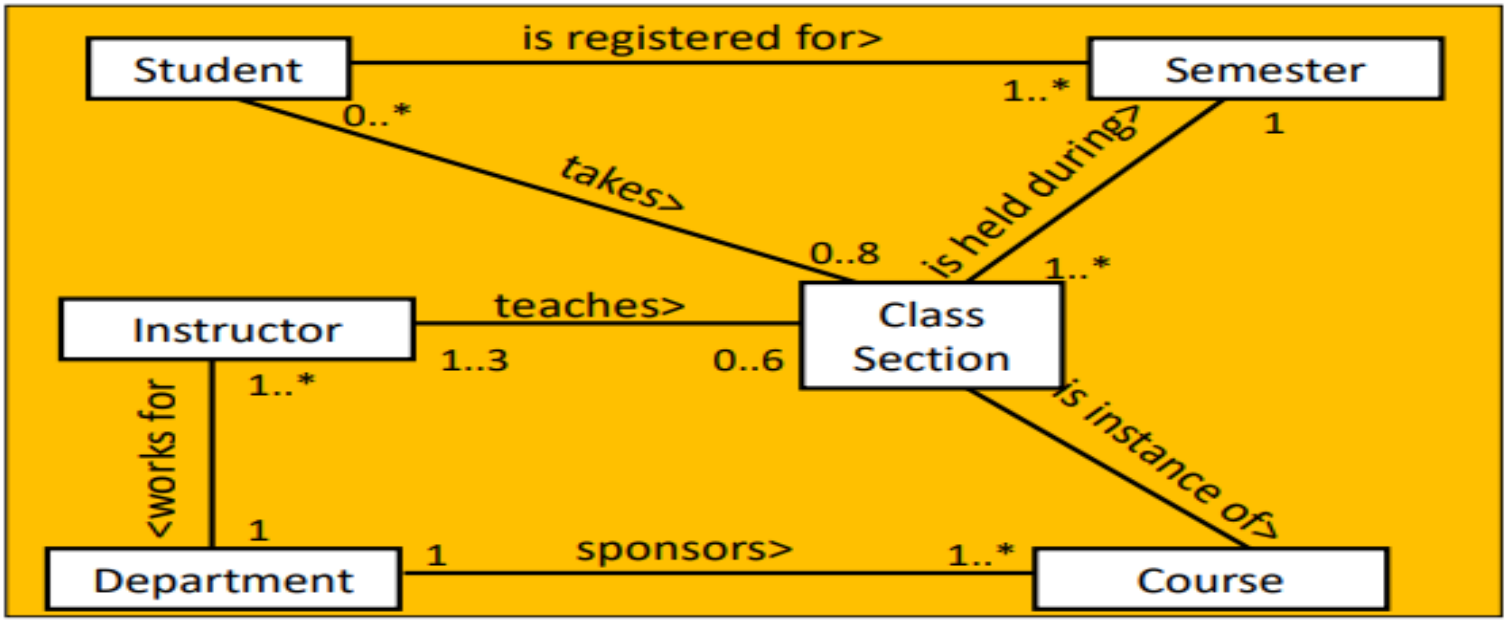
- An association is a *structural relationship* that specifies a connection between classes
- Classes A and B are associated if:
  - An object of class A sends a message to an object of B
  - An object of class A creates an instance of class B
  - An object of class A has an attribute of type B or collections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

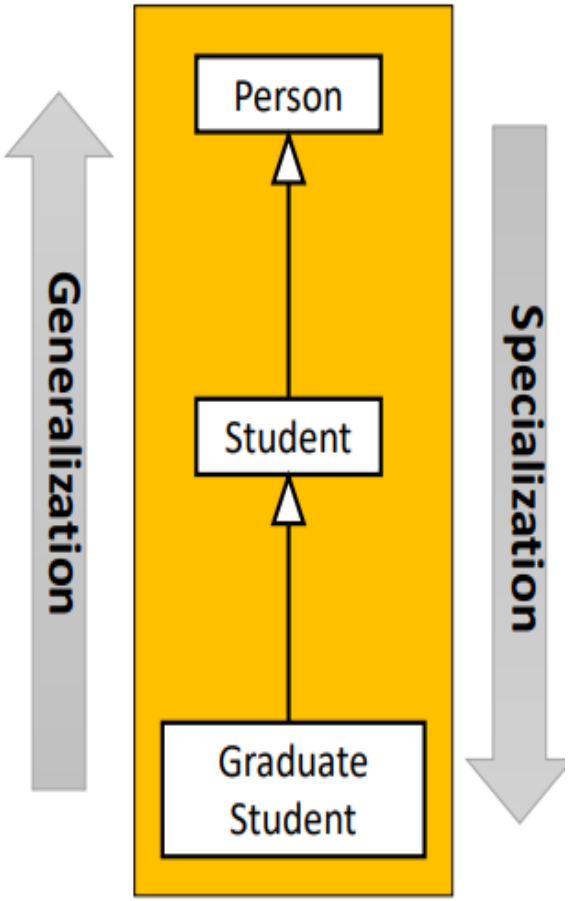
  - e.g., 1 1..2 0..* 1..* * (same as 0..*)
  - Implementing associations depends on multiplicity

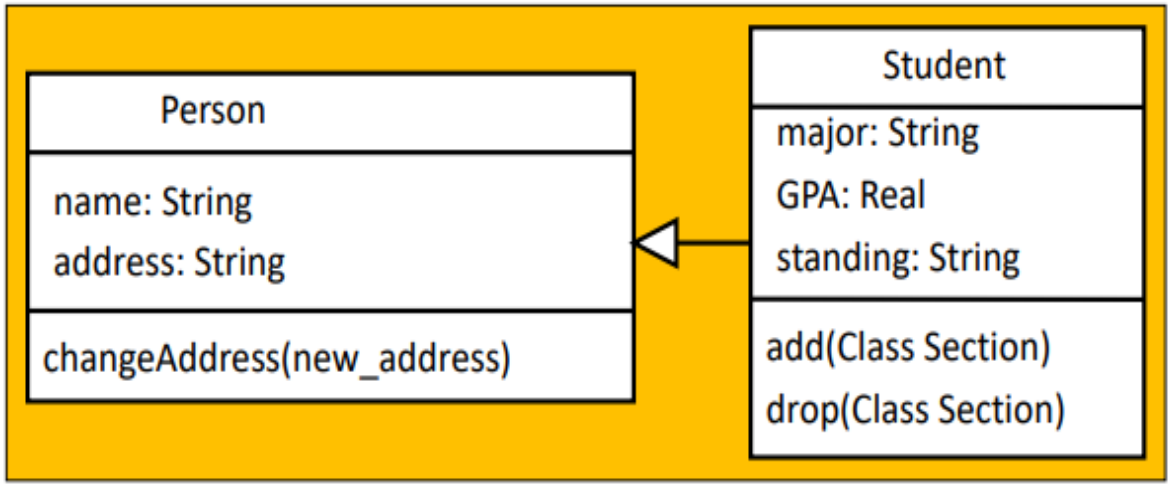


Generalization

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

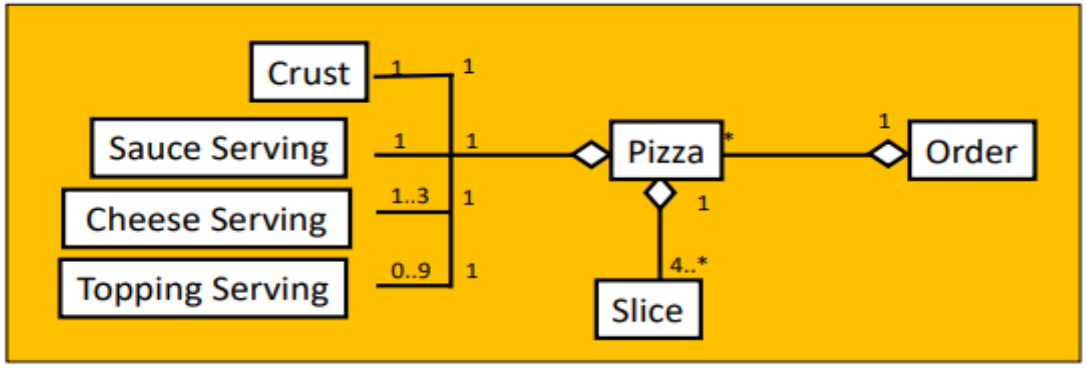
- From an analysis perspective, it represents generalization/specialization:
  - Specialization is a subset of the generalization





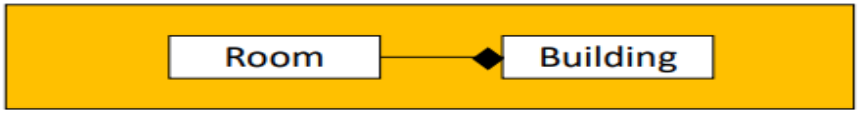
Aggregation

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



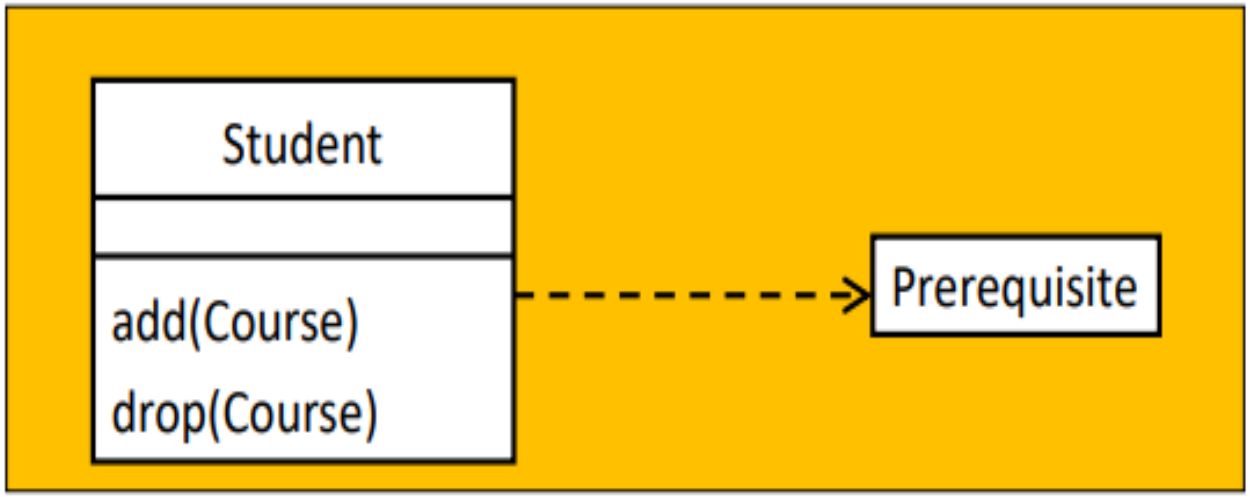
Composition

- 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**



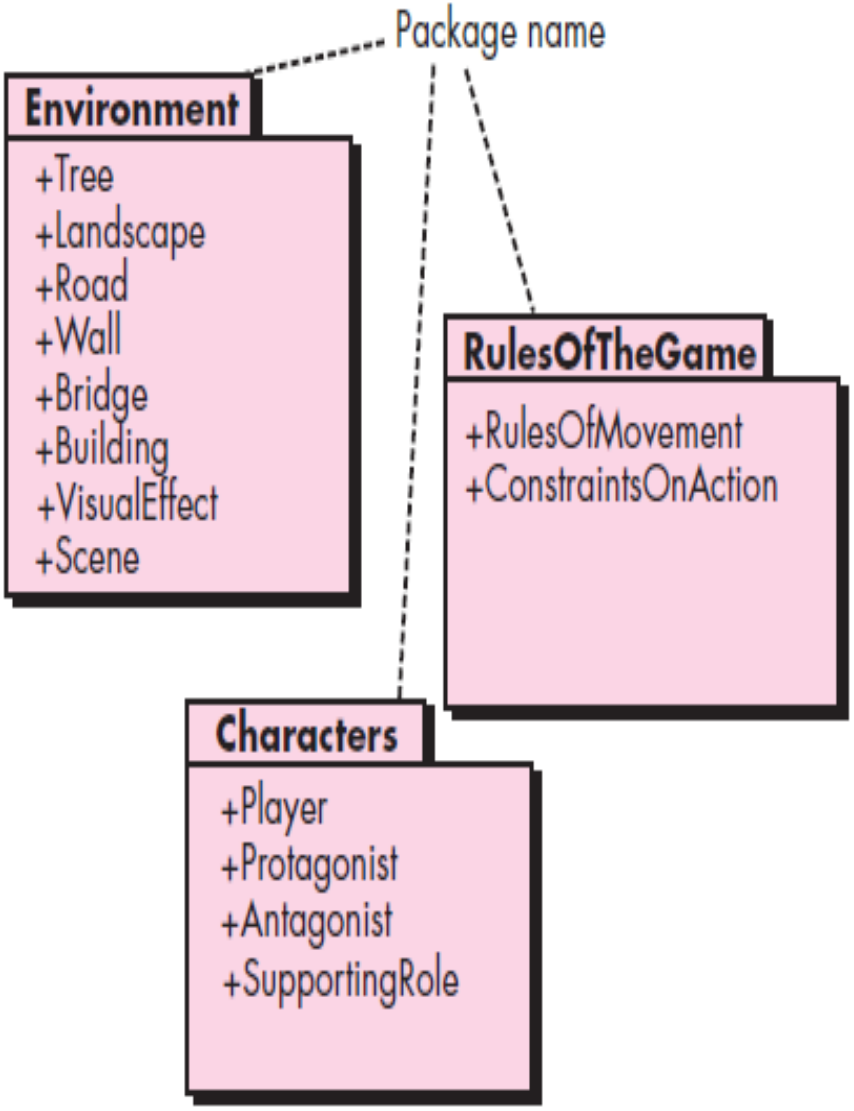
Dependencies

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

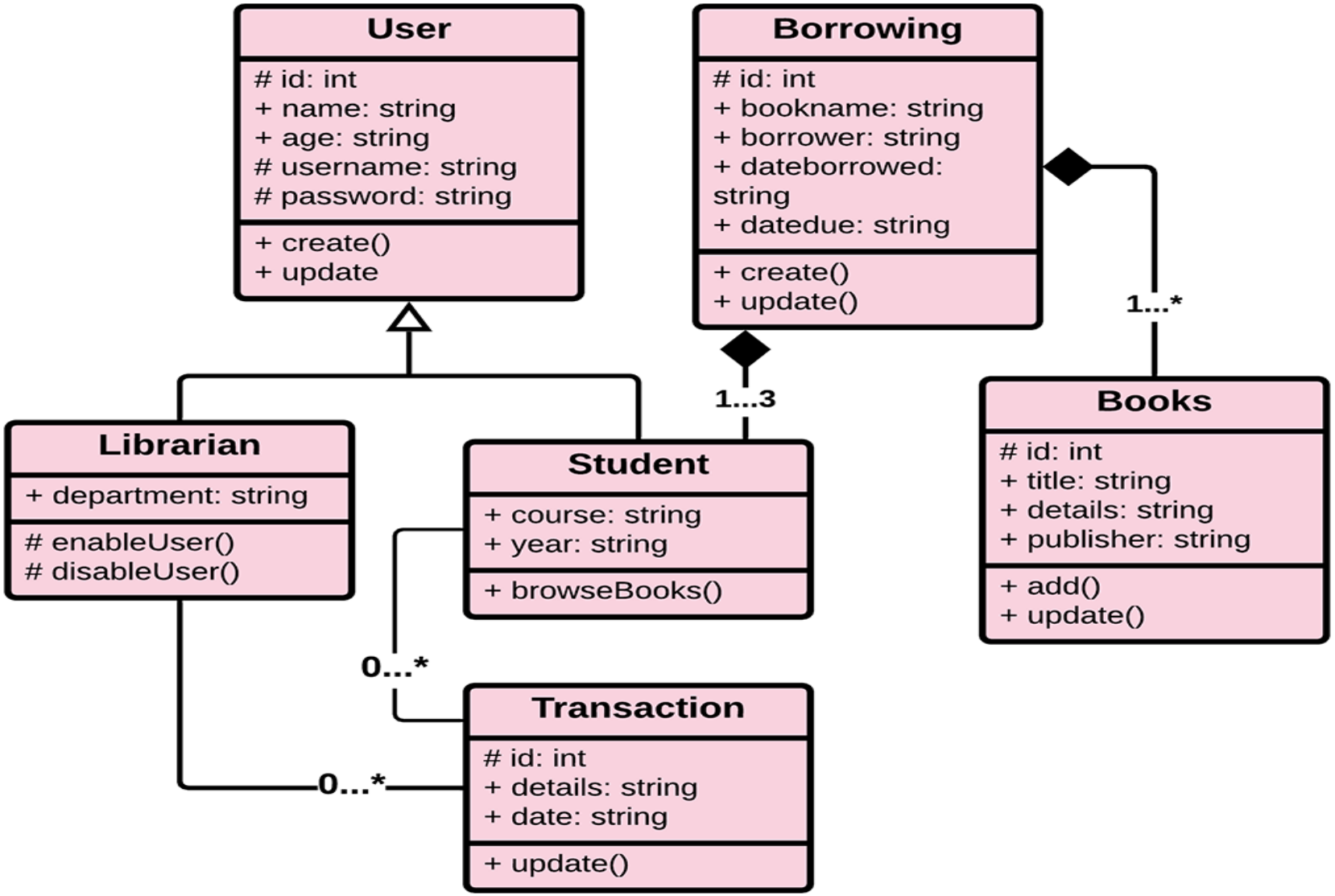


Analysis Packages

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.

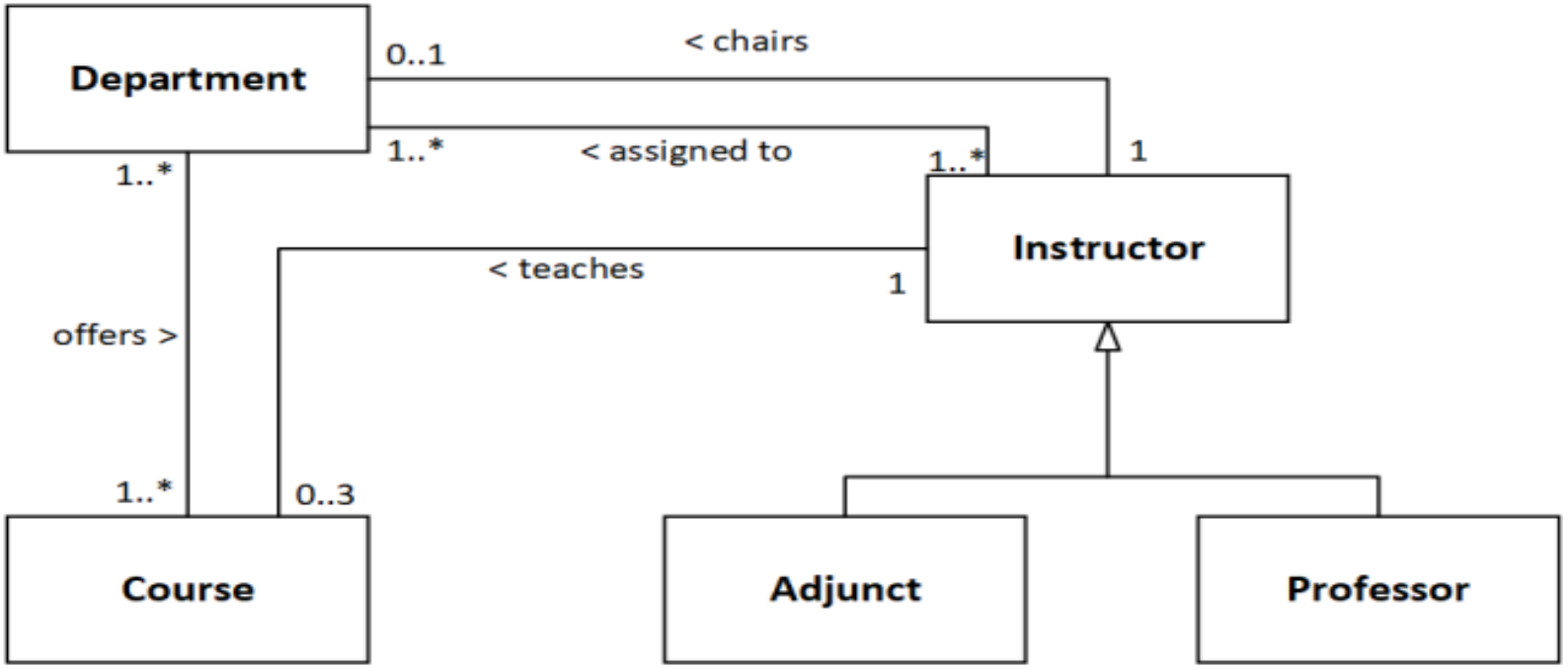


Library Management System



Example

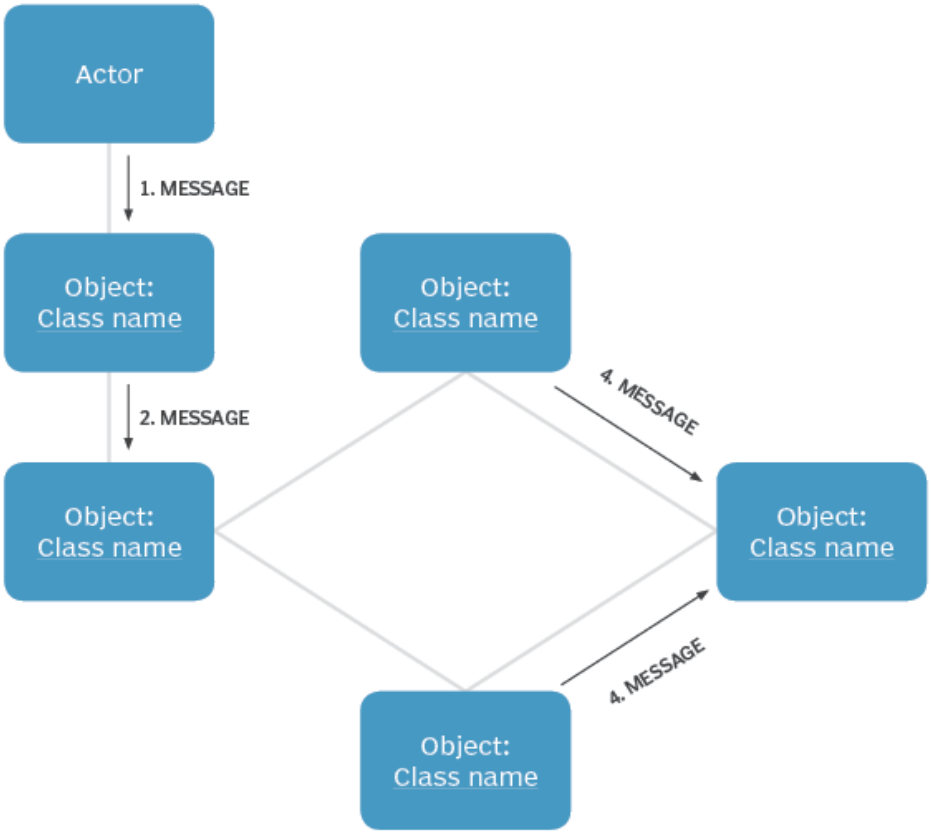
- University Courses
  - Some instructors are professors, while others have job title adjunct
  - Departments offer many courses, but a course may be offered by more than 1 department
  - Courses are taught by instructors, who may teach up to three courses
  - Instructors are assigned to one (or more)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



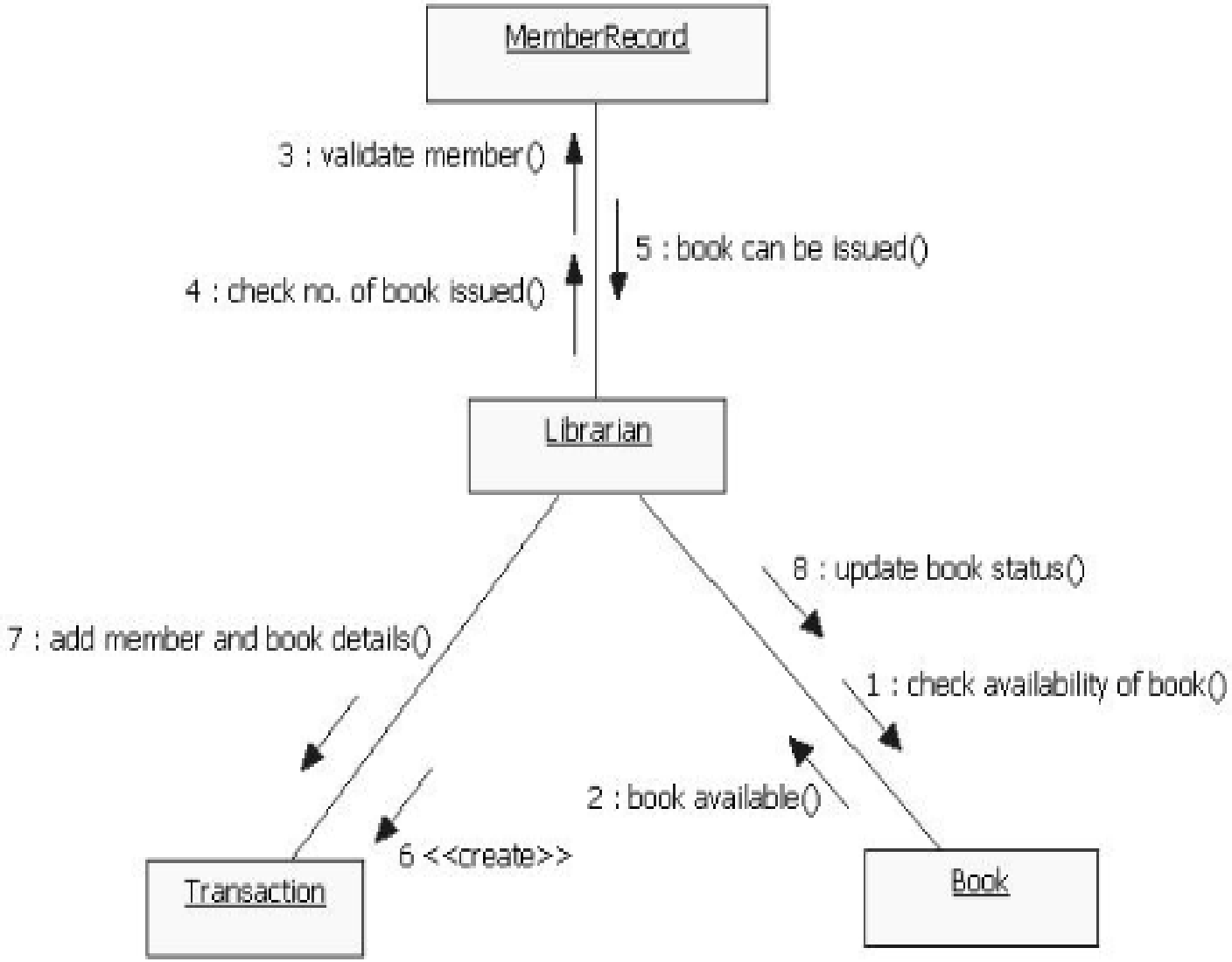
Collaboration Diagram

- 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.

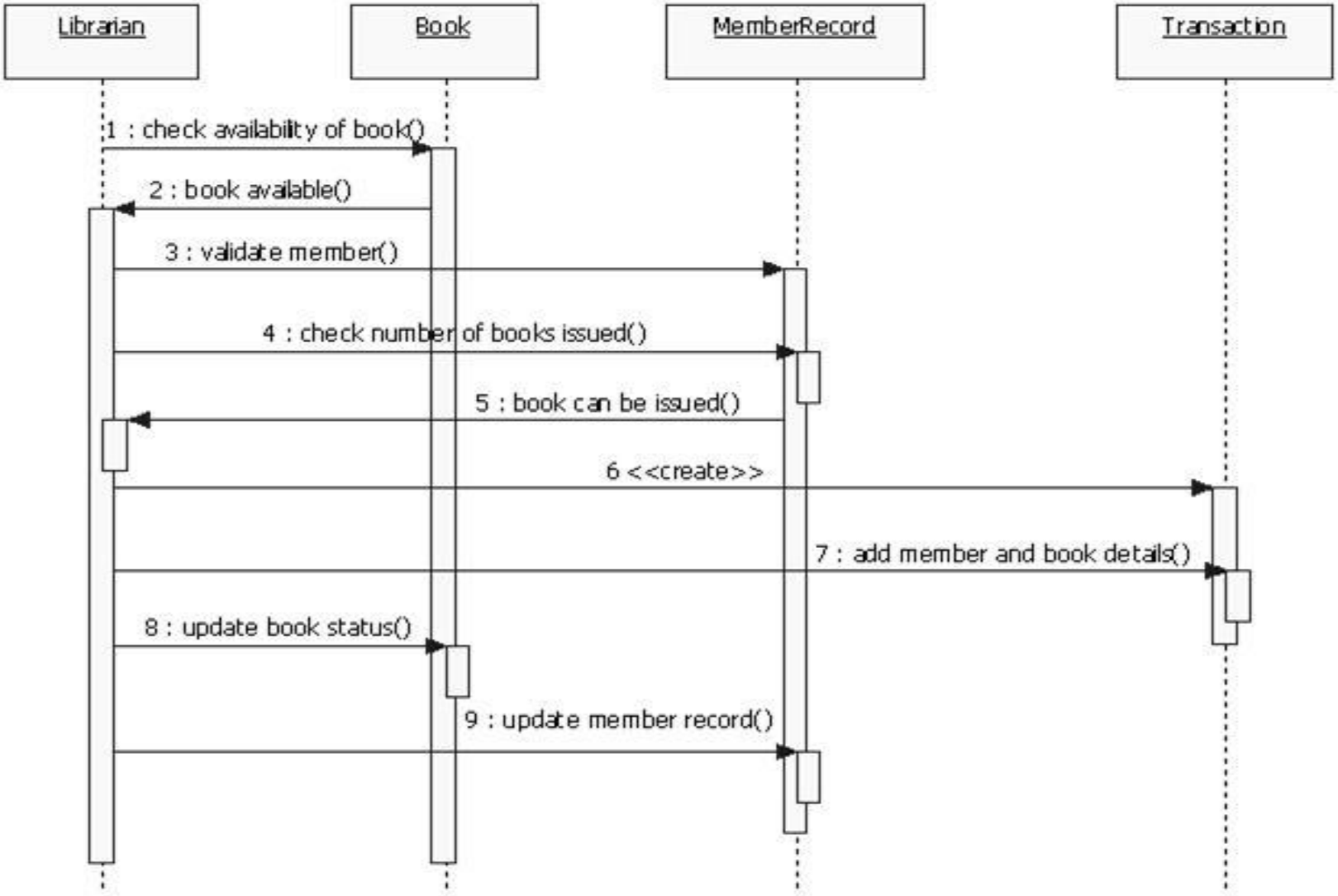
Components of a collaboration diagram



Collaboration Diagram for Issuing Book



Sequence Diagram Example



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:

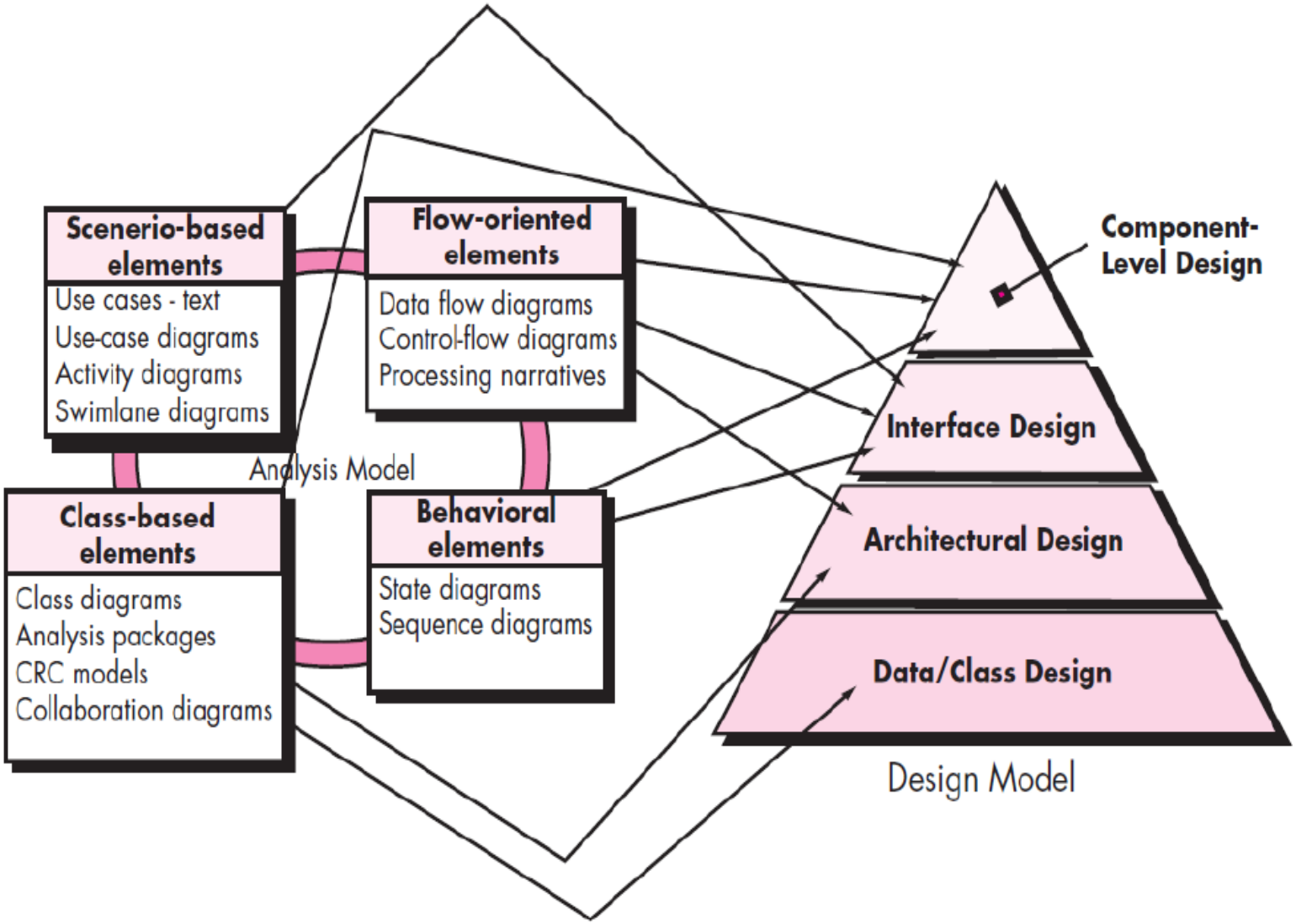
- It combines the ability to extend the **program, adaptability, serviceability**. These three term defines the maintainability.
- **Testability, compatibility and configurability** are the terms using which a system can be easily installed and found the problem easily.
- Supportability also consists 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**
- **Modularity**
- **Information hiding**
- **Functional independence**
- **Refinement**
- **Refactoring**
- **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,

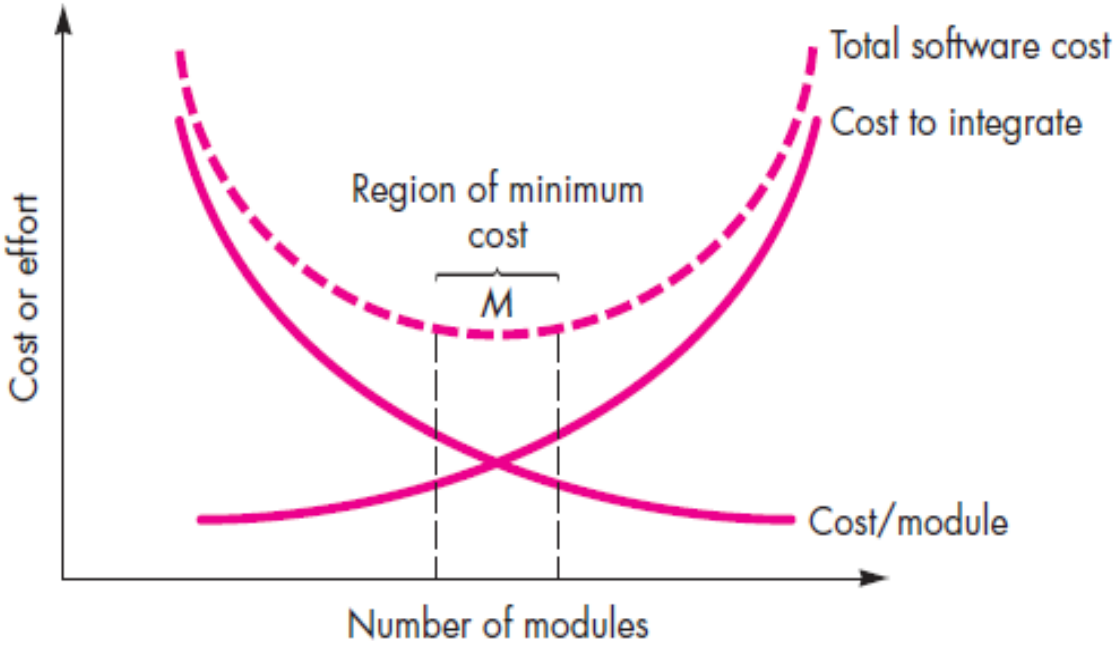
  Refinement.

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**