Software Engineering

2UCC501

August 2022- December 2022

MODULE 2: **Requirement Engineering**

2.1 Introduction to Object Oriented Methodologies :Booch, Rumbaugh and Jacobson

2.2 Requirements Engineering Tasks, Requirement Elicitation Techniques, Software Requirements: Functional, Non- Functional

2.3 Requirements Characteristics, Requirement qualities, Requirement Specification, Requirement Traceability, System Analysis Model Generation, Documentation :Use Case Diagram, Activity Diagram

2.4 Categorizing classes: entity, boundary and control, Modelling associations and collections-Class Diagram

2.5 Dynamic Analysis - Identifying Interaction – Sequence and Collaboration diagrams, State chart diagram

Object Oriented Methodologies

UML(Unified Modelling Language)

- UML is a modelling language.
- Not a system design or development methodology.
- Used to document object-oriented analysis and design results.
- Independent of any specific design methodology.

UML Based Principally on: −

OMT [Rumbaugh 1991] −
Booch's methodology [Booch 1991] −
OOSE [Jacobson 1992] −
Odell's methodology [Odell 1992] −
Shlaer and Mellor [Shlaer 1992] -

Ref: NPTEL course on software engineering, Ranbir mall

Why are UML Models Required?

- Modelling is an abstraction mechanism:
  - −Capture only important aspects and ignores the rest.
  - −Different models obtained when different aspects are ignored.
  - −An effective mechanism to handle complexity.

- UML is a graphical modelling technique
- Easy to understand and construct

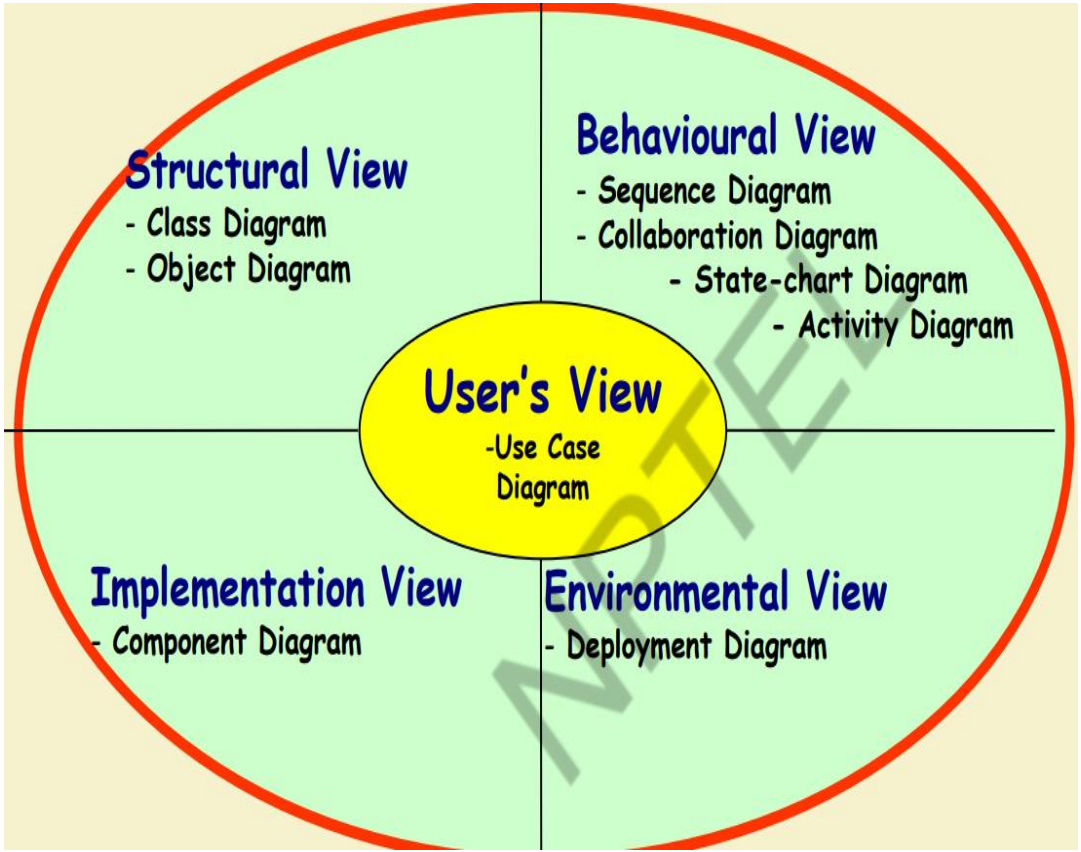
Ref: NPTEL course on software engineering, Ranbir mall

UML Model Views

Views of a system:

- User’s view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

Diagrams and views in UML



Ref: NPTEL course on software engineering, Ranbir mall

Structural Diagrams

- Class Diagram – set of classes and their relationships.
- Object Diagram – set of objects (class instances) and their relationships
- Component Diagram – logical groupings of elements and their relationships
- Deployment Diagram – set of computational resources (nodes) that host each component.

- Use Case Diagram – high-level behaviors of the system, user goals, external entities: actors
- Sequence Diagram – focus on time ordering of messages
- Collaboration Diagram – focus on structural organization of objects and messages
- State Chart Diagram – event driven state changes of system
- Activity Diagram – flow of control between activities

Use case model

- Consists of a set of “use cases”
- It is the central model:

- A Use Case • A case of use: A way in which a system can be used by the users to achieve specific goals.

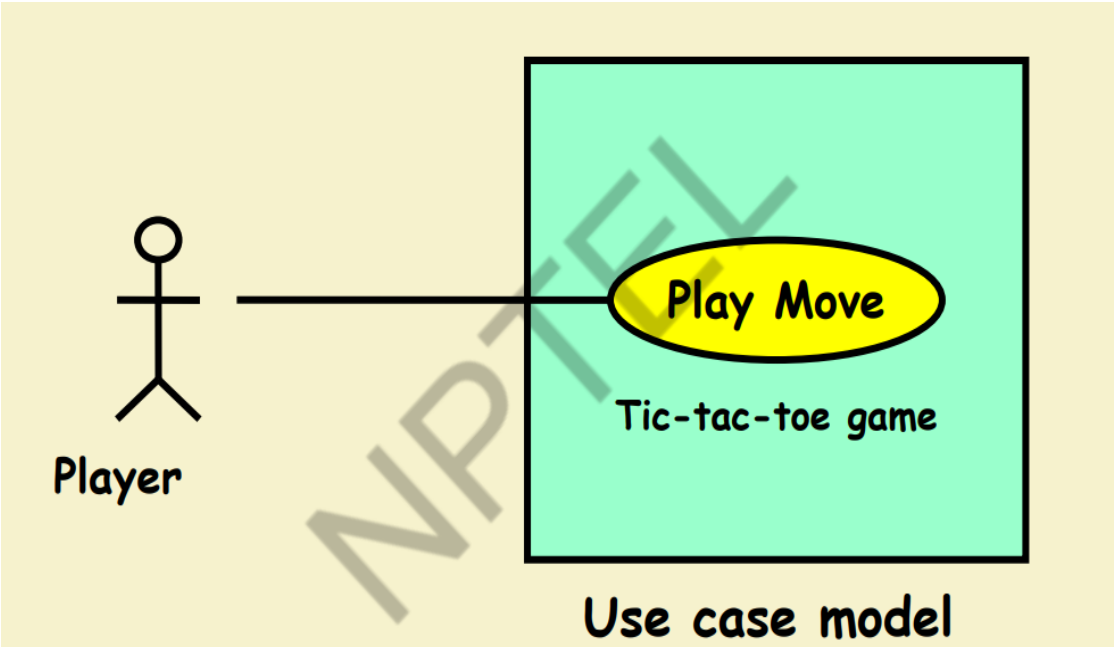
Use Case

Use cases for a Library information system

- issue-book
- query-book
- return-book
- create-member
- add-book, etc

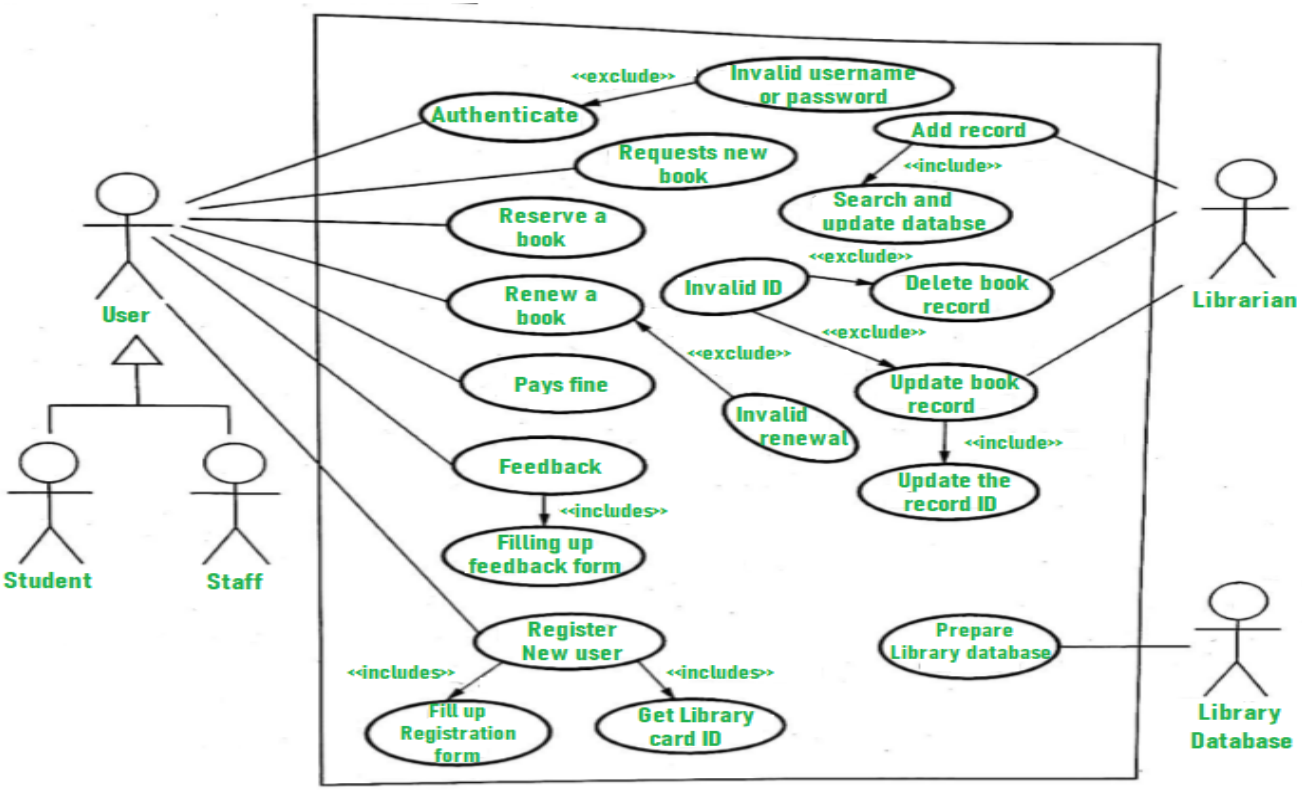
An example of use case diagram

- A use case is represented by an ellipse.
- System boundary is represented by a rectangle.
- Users are represented by stick person icons (actor).
- Communication relationship between actor and use case by a line.
- External system by adding a stereotype.



Ref: NPTEL course on software engineering, Ranbir mall

Use case diagram of LMS



Ref: https://www.geeksforgeeks.org/use-case-diagram-for-library-management-system/

Introduction to Object Oriented Methodologies :Booch, Rumbaugh and Jacobson

- Engineering is a problem-solving activity. Engineers search for an appropriate solution, often by trial and error, evaluating alternatives empirically, with limited resources and incomplete knowledge.
- Engineering method includes five steps:
  1. Formulate the problem.
  2. Analyse the problem.
  3. Search for solutions.
  4. Decide on the appropriate solution.
  5. Specify the solution.

Introduction to Object Oriented Methodologies :Booch, Rumbaugh and Jacobson

- Object-oriented software development typically includes activities:

1. Requirements elicitation
2. Analysis
3. System design
4. Object design
5. Implementation
6. Testing

Object Oriented Methodologies

1. Object oriented analysis by Code & Yourdon
2. Object oriented design by Grady Booch
3. Object oriented modeling techniques by James Rumbaugh
4. Object oriented software engineering (OOSE) by Jacobson

Object oriented design by Grady Booch

- Approaches to software improvement process
- Micro development process- identify the classes, identify the semantic of class and object, identify the relationship of class and object, interfaces and implementation.
- Macro development process- basic needs of software (conceptualization)

- Analysis
- Design of architect
- Development
- Maintenance

Object Oriented Methodologies

- Object modeling techniques (OMT) by James Rumbaugh

  It focus on analysis, design and implementation of the system.

  - Analysis- Object model (static aspects), dynamic model (behavioral aspect), functional model (functional aspect)
  - System design- high level design
  - Object design – objects in detail
  - Implementation – Coding

Object oriented software engineering (OOSE) by Jacobson

- It covers entire life cycle
- Requirement model, analysis model, design model, implementation model, test model.

OO Software Engineering

- **Requirement Elicitation & Analysis**
- Formulate the problem with the client and build the application domain model.
- Requirements elicitation and analysis correspond to steps 1 and 2 of the engineering method (formulate & analyse the problem)

System design

- Analyse the problem, break it down into smaller pieces, and select general strategies for designing the System

Object design

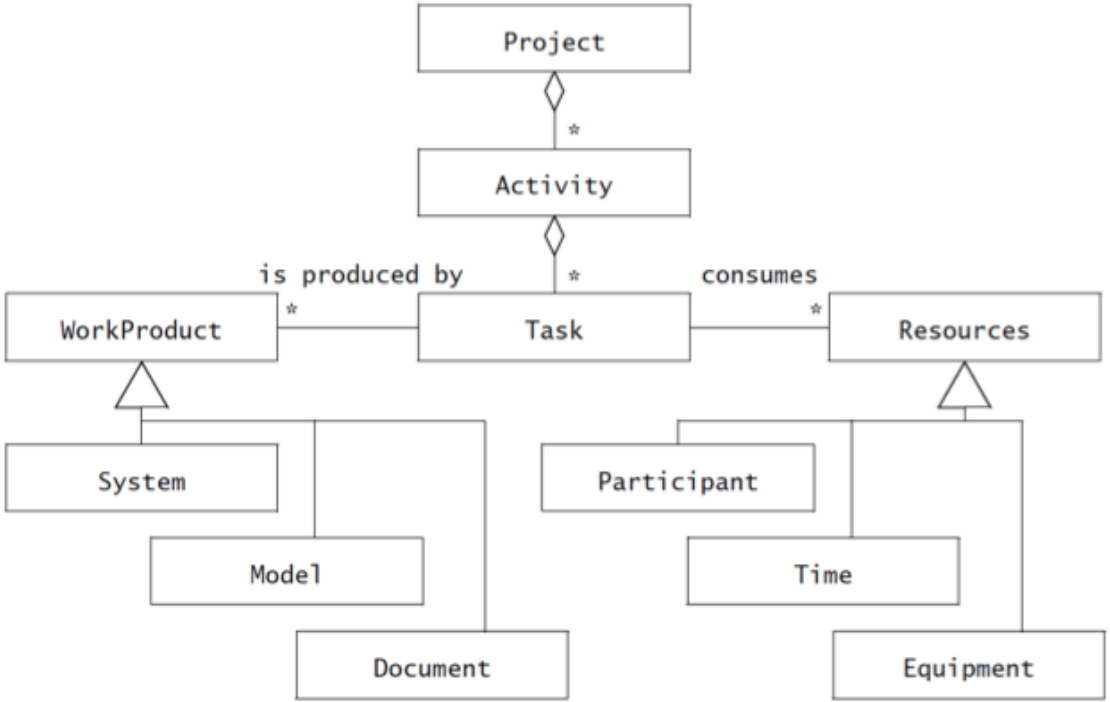
- select detail solutions for each piece and decide on the most appropriate solution.
- System design and object design result in the solution domain model.
- System and object design correspond to steps 3 and 4 of the engineering method (search for solutions & decide upon appropriate solution)

OO Software Engineering

- **Requirement Elicitation & Analysis**
- **System design**
- **Object design**
- **Implementation**
  - Realize the system by translating the solution domain model into an executable representation
  - Implementation corresponds to step 5 of the engineering method (Specify solution)

What makes software engineering different from problem solving in other sciences is that change occurs in the application and the solution domain while the problem is being solved.

OO Software Engineering



OO Software Engineering

- Each rectangle represents a **concept**.
- The lines among the rectangles represent different relationships between the concepts. For example, the **diamond shape indicates aggregation:** a Project includes a number of Activities, which includes a number of Tasks.
- The triangle shape indicates a generalization relationship; Participants, Time, and Equipment are specific kinds of Resources.
- Persons involved (client, project manager, developers, end users) in the project as **participants**
- A set of responsibilities in the project or the system as a **role**
- A role is associated with a set of **tasks** and is assigned to a participant

OO Software Engineering

Terminology

- **System** as a collection of interconnected parts
- Modelling is a way to deal with complexity by ignoring irrelevant details
- **Model** refers to any abstraction of the system
- **A work product** is an artefact that is produced during the development, such as a document or a piece of software for other developers or for the client.
  - **Internal work product:** A work product for the project's internal consumption
  - **A deliverable**: a work product that must be delivered to a client
  - Deliverables are generally defined prior to the start of the project and specified by a contract binding the developers with the client

OO Software Engineering

- **An activity/ phases** is a set of tasks that is performed toward a specific purpose
  - Requirements elicitation is an activity whose purpose is to define with the client what the system will do.
  - Delivery is an activity whose purpose is to install the system at an operational location.
  - Management is an activity whose purpose is to monitor and control the project such that it meets its goals (e.g., deadline, quality, budget). Activities can be composed of other activities.
- **A task** represents an atomic unit of work that can be managed: A manager assigns it to a developer, the developer carries it out, and the manager monitors the progress and completion of the task.
  - Tasks consume resources, result in work products, and depend on work products produced by other tasks.
- **Resources** are assets that are used to accomplish work.
  - Resources include time, equipment, and labour. When planning a project, a manager breaks down the work into tasks and assigns them to resources

TicketDistributor a case study

- TicketDistributor is a machine that distributes tickets for trains.
- Travelers have the option of selecting a ticket for a single trip or for multiple trips, or selecting a time card for a day or a week. The
- TicketDistributor computes the price of the requested ticket based on the area in which the trip will take place and whether the traveler is a child or an adult.
- The TicketDistributor must be able to handle several exceptions, such as travellers who do not complete the transaction, travellers who attempt to pay with large bills, and resource outages, such as running out of tickets, change, or power.

OO Software Engineering

Steps to be followed while designing the system requirements

- Identify roles & responsibilities of various stakeholders
- Identify the various work products
- Identify various activities, tasks & resources
- Identify Requirements

TicketDistributor a sample

- Write the different people and their roles & responsibilities in the TicketDistributor system

TicketDistributor a case study

- The different people and their roles & responsibilities in the TicketDistributor system

Role	Example	Responsibility
Client	Train company	The client is responsible for providing the high-level requirements on the system and for defining the scope of the project (delivery date, budget, quality criteria).
User	Travelers	The user is responsible for providing domain knowledge about current user tasks. Note that the client and the user are usually filled by different persons.
Human factors specialist	HCI specialist	A human factors specialist is responsible for the usability of the system.

TicketDistributor a case study

Role	Example	Responsibility
Manager	Project manager	A manager is responsible for the work organization. This includes hiring staff, assigning them tasks, monitoring their progress, providing for their training, and generally managing the resources provided by the client for a successful delivery.
Developer	Analyst, tester	A developer is responsible for the construction of the system, including specification, design, implementation, and testing. In large projects, the developer role is further specialized.
Technical writer		The technical writer is responsible for the documentation delivered to the client. A technical writer interviews developers, managers, and users to understand the system.

TicketDistributor a case study

- Some sample work products

Work product	Type	Description
Specifications	Deliverable	The specification describes the system from the user's point of view. It is used as a contractual document between the project and the client. The TicketDistributor specification describes in detail how the system should appear to the traveller.
Operation Manual	Deliverable	The operation manual for the TicketDistributor is used by the staff of the train company responsible for installing and configuring the TicketDistributor. Such a manual describes, for example, how to change the price of tickets and the structure of the network into zones.

TicketDistributor a case study

Work product	Type	Description
Status Report	Internal work product	A status report describes at a given time the tasks that have been completed and the tasks that are still in progress. The status report is produced for the manager, Alice, and is usually not seen by the train company.
Test A	Internal work product	The test plans and results are produced by the tester. These documents track the known defects in the prototype TicketDistributor and their state of repair. These documents are usually not shared with the client.

TicketDistributor a case study

- Activities, tasks, and resources for the TicketDistributor project.

Work product	Type	Description
Requirements elicitation	**Activity**	**The requirements elicitation activity includes obtaining and validating requirements and domain knowledge from the client and the users. The requirements elicitation activity produces the specification work product**
Develop “Out of Change” test case for TicketDistributor	Task	This task, assigned to the tester focuses on verifying the behaviour of the ticket distributor when it runs out of money and cannot give the correct change back to the user. This activity includes specifying the environment of the test, the sequence of inputs to be entered, and the expected outputs.

TicketDistributor a case study

Activities, tasks, and resources for the TicketDistributor project.:

Work product	Type	Description
Review "Access Online Help" use case for usability	**Task**	**This task, assigned to the human factors specialist focuses on detecting usability issues in accessing the online help features of the system. The tariff database includes an example of tariff structure with a train network plan. This example is a resource provided by the client for requirements and testing.**
Tariff Database	Resource	The tariff database includes an example of tariff structure with a train network plan. This example is a resource provided by the client for requirements and testing.

OOSE

Steps while designing the system requirements

- Identify roles & responsibilities of various stakeholders
- Identify the various work products
- Identify various activities, tasks & resources
- Identify Requirements

Steps while designing the system requirements

Identify requirements:

- Requirements specify a set of features that the system must have.
- A **functional requirement** is a specification of a function that the system **must support**
- A **nonfunctional requirement** is **a constraint** on the operation of the system that is not related directly to a function of the system.

- E.g. classify into functional & non functional requirements

1. The user must be able to purchase tickets
2. The user must be able to access tariff information
3. The user must be provided feedback in less than one second
4. The colours used in the interface should be consistent with the company

Notations, Methods & Methodology

- A **notation** is a graphical or textual set of rules for representing a model.
- A **method** is a repeatable technique that specifies the steps involved in solving a specific problem.
- A **methodology** is a collection of methods for solving a class of problems and specifies how and when each method should be used.

Different object-oriented methodologies

- Royce's methodology
- The Object Modelling Technique (OMT) Rumbaugh et al.
- The Booch methodology
- Catalysis

OOSE activities

Technical activities associated with object-oriented software engineering.

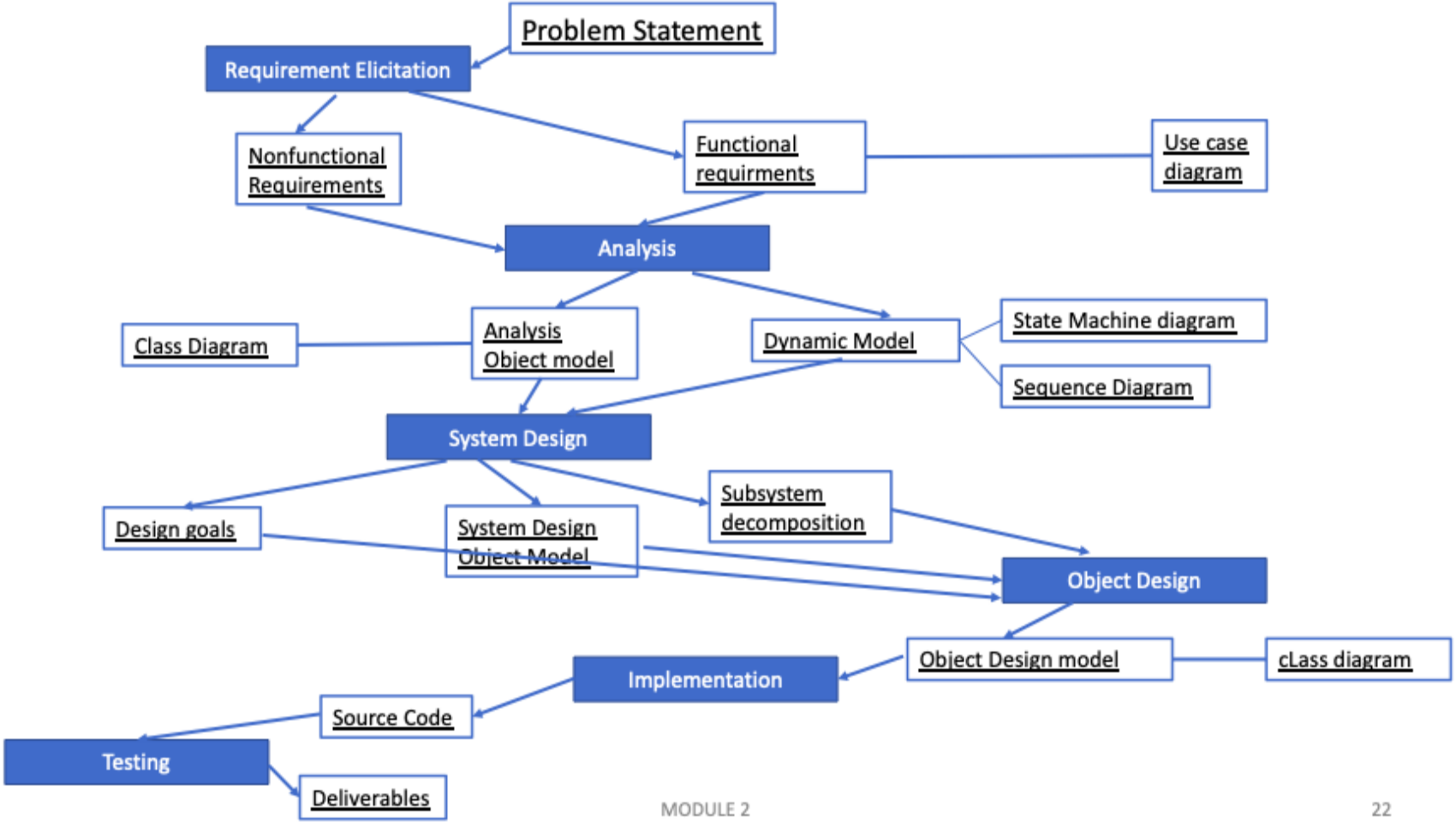
•Development activities deal with the complexity by constructing and validating models of the application domain or the system

•Development activities:

- Requirements Elicitation
- Analysis
- System Design
- Object Design
- Implementation
- Testing

you draw out(elicit) the requirements
analyse your drawing
drawing --> design? system design
desgins have objects_> design the objects
implement the desgins and software
test it

Overview of OOSE Development activities &their products



OOSE activities

- Development activities:
- **Requirements Elicitation**
  - The client and developers define the purpose of the system.
  - The result of this activity is a description of the system in terms of actors and use cases.
  - Actors represent the external entities that interact with the system. Actors include roles such as end users, other computers the system needs to deal with (e.g., a central bank computer, a network), and the environment (e.g., a chemical process).
  - Use cases are general sequences of events that describe all the possible actions between an actor and the system for a given piece of functionality.

OOSE activities

- Development activities:

- **Requirement Elicitation**

- **Analysis**
- Developers aim to produce a model of the system that is correct, complete, consistent, and unambiguous.
- Developers transform the use cases produced during requirements elicitation into an object model that completely describes the system
- Developers discover ambiguities and inconsistencies in the use case model that they resolve with the client
- The result of analysis is a system model annotated with attributes, operations, and associations.
- The system model can be described in terms of its structure and its dynamic interoperation.

OOSE activities

- Development activities:
- **Requirement Elicitation**
- **Analysis**

- **System Design**

- Developers define the design goals of the project and decompose the system into smaller subsystems that can be realized by individual teams.
- Developers also select strategies for building the system, such as the hardware/software platform on which the system will run, the persistent data management strategy, the global control flow, the access control policy, and the handling of boundary conditions.
- The result of system design is a clear description of each of these strategies, a subsystem decomposition, and a deployment diagram representing the hardware/software mapping of the system.
- Whereas both analysis and system design produce models of the system under construction, only analysis deals with entities that the client can understand.
- System design deals with a much more refined model that includes many entities that are beyond the comprehension (and interest) of the client.

OOSE activities

- Development activities:
- **Requirement Elicitation**
- **Analysis**
- **System Design**
- **Object Design**

- Developers define solution domain objects to bridge the gap between the analysis model and the hardware/software platform defined during system design.
- Precisely describing object and subsystem interfaces, selecting off-the-shelf components, restructuring the object model to attain design goals such as extensibility or understandability, and optimizing the object model for performance.
- A detailed object model annotated with constraints and precise descriptions for each element.

OOSE activities

- Development activities:
- **Requirement Elicitation**
- **Analysis**
- **System Design**
- **Object Design**
- **Implementation**

- Developers translate the solution domain model into source code.
- Includes implementing the attributes and methods of each object and integrating all the objects such that they function as a single system.
- The implementation activity spans the gap between the detailed object design model and a complete set of source code files that can be compiled.

OOSE activities

•Development activities:
•**Requirement Elicitation • Analysis • System Design • Object Design •Implementation • Testing**

•Developers find differences between the system and its models by executing the system (or parts of it) with sample input data sets.
•Developers compare the object design model with each object and subsystem.
•During integration testing, combinations of subsystems are integrated together and compared with the system design model.
•During system testing, typical and exception cases are run through the system and compared with the requirements model.
•The goal of testing is to discover as many faults as possible such that they can be repaired before the delivery of the system.
•The planning of test phases occurs in parallel to the other development activities: System tests are planned during requirements elicitation and analysis, integration tests are planned during system design, and unit tests are planned during object design.

Question

Compare Booch, Rumbaugh and Jacobson approaches

Module 2: Requirement Engineering

- 2.1 Introduction to Object Oriented Methodologies :Booch, Rumbaugh and Jacobson

2.2 Requirements Engineering Tasks, Requirement Elicitation Techniques, Software Requirements: Functional, Non- Functional

2.3 Requirements Characteristics, Requirement qualities, Requirement Specification, Requirement Traceability, System Analysis Model Generation, Documentation :Use Case Diagram, Activity Diagram

2.4 Categorizing classes: entity, boundary and control, Modelling associations and collections-Class Diagram

2.5 Dynamic Analysis - Identifying Interaction – Sequence and Collaboration diagrams, State chart diagram

Requirement Engineering Tasks

- Requirements Engineering includes two main activities:

1. **Requirements elicitation**, which results in the specification of the system that the client understands.

2. **Analysis**, which results in an analysis model that the developers can unambiguously interpret.

Requirements elicitation

- Interview

  Open ended or structured
- Brainstorming
- Delphi Technique
- Observation
- Surveys
- Prototyping
- Questionnaire

Requirement Engineering Tasks

•Requirements Engineering includes two main activities:
1.**Requirements elicitation**, which results in the specification of the system that the client understands
•Challenging of the because it requires the collaboration of several groups of participants with different backgrounds
•**The client and the users** are experts in their domain and have a general idea of what the system should do, but they often have little experience in software development
•**The developers** have experience in building systems, but often have little knowledge of the everyday environment of the users
•Scenarios and use cases provide tools for bridging this gap.
•**A scenario** describes an example of system use in terms of a series of interactions between the user and the system
•**A use case** is an abstraction that describes a class of scenarios.
•Both scenarios and use cases are written in natural language, a form that is understandable to the user.

Requirement Elicitation

Requirements elicitation: communication among developers, clients, and users to define a new system.

•Failure to communicate and understand each others' domains results in a system that is difficult to use or that simply fails to support the user's work.

•**Errors** introduced during requirements elicitation are **expensive to correct .**

•Errors include missing functionality that the system should have supported, functionality that was incorrectly specified, user interfaces that are misleading or unusable, and obsolete functionality

•

Requirements elicitation methods aim at improving communication among developers, clients, and users

•Developers construct a model of the application domain by observing users in their environment.

•Developers select a representation that is understandable by the clients and users

•Developers validate the application domain model by constructing simple prototypes of the user interface and collecting feedback from potential users.

•**A use case** is an abstraction that describes a class of scenarios.

•Both scenarios and use cases are written in natural language, a form that is understandable to the user.

Requirement Elicitation

- **Requirements elicitation** focuses on describing the purpose of the system.
- The client, the developers, and the users identify a problem area and define a system that addresses the problem.
- Thus a **requirements specification** is created and used as a contract between the client and the developers.
- The requirements specification supports the communication with the client and users
- The analysis model supports the communication among developers
- Requirements elicitation and analysis occur concurrently and iteratively

Requirement Elicitation

•Requirement elicitation Activities :

• Identifying actors : developers identify the different types of users the future system will support.
•Identifying scenarios: developers observe users and develop a set of detailed scenarios for typical functionality provided by the future system.
•Scenarios : concrete examples of the future system in use, used by the developers to communicate with the user and deepen their understanding of the application domain.
•Identifying use cases derived from the scenarios: a set of use cases that completely represent the future system which is abstractions describing all possible cases.
•Use cases are used to determine the scope of the system.
•Refining use cases : developers ensure that the requirements specification is complete by detailing each use case and describing the behaviour of the system in the presence of errors and exceptional conditions.
•Identifying relationships among use cases.
•Identify dependencies among use cases.
•Consolidate the use case model by factoring out common functionality ensuring the requirements specification is consistent.
•Identifying nonfunctional requirements. aspects that are visible to the user, but not directly related to functionality which includes constraints on the performance of the system, its documentation, the resources it consumes, its security, and its quality.

Requirement Elicitation

Sources for Requirements Developers access many different sources of information such as :

•Client-supplied documents about the application domain, manuals and technical documentation of legacy systems that the future system will replace

•users and clients themselves. Developers interact the most with users and clients during requirements elicitation.

Two methods for eliciting information, making decisions with users and clients, and managing dependencies among requirements and other artefacts:

•**Joint Application Design (JAD)** focuses on building consensus among developers, users, and clients by jointly developing the requirements specification.

•**Traceability** focuses on recording, structuring, linking, grouping, and maintaining dependencies among requirements and between requirements and other work products.

Requirement Elicitation

Functional Requirements:

- Describe the interactions between the system and its environment independent of its implementation.
- The environment includes the user and any other external system with which the system interacts.

Requirement Elicitation

Non- Functional Requirements:

•Describe aspects of the system that are not directly related to the functional behaviour of the system.

•Include a broad variety of requirements that apply to many different aspects of the system, such as

•**Usability** is the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component.

•Usability requirements include, conventions adopted by the user interface, the scope of online help, and the level of user documentation.

•Clients address usability issues by requiring the developer to follow user interface guidelines on colour schemes, logos, and fonts.

•**Reliability** is the ability of a system or component to perform its required functions under stated conditions for a specified period of time.

•Include, an acceptable mean time to failure and the ability to detect specified faults or to withstand specified security attacks.

Requirement Elicitation

Non- Functional Requirements:

- **Dependability**, which is the property of a computer system such that reliance can justifiably be placed on the service it delivers.
- Includes reliability, robustness (the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions), and safety (a measure of the absence of catastrophic consequences to the environment).

Requirement Elicitation

Non- Functional Requirements:

- **Performance requirements** are concerned with quantifiable attributes of the system, such as response time (how quickly the system reacts to a user input), throughput (how much work the system can accomplish within a specified amount of time), availability (the degree to which a system or component is operational and accessible when required for use), and accuracy.
- **Supportability** requirements are concerned with the ease of changes to the system after deployment
- Includes adaptability (the ability to change the system to deal with additional application domain concepts), maintainability (the ability to change the system to deal with new technology or to fix defects), and internationalization (the ability to change the system to deal with additional international conventions, such as languages, units, and number formats).
- **Implementation requirements** are constraints on the implementation of the system, including the use of specific tools, programming languages, or hardware platforms.

Requirement Elicitation

Non- Functional Requirements:

- **Interface requirements** are constraints imposed by external systems, including legacy systems and interchange formats.
- **Operations requirements** are constraints on the administration and management of the system in the operational setting.
- **Packaging requirements** are constraints on the actual delivery of the system (e.g., constraints on the installation media for setting up the software).
- **Legal requirements** are concerned with licensing, regulation, and certification issues. systems must be accessible to people with disabilities etc.

Module 2: Requirement Engineering

- 2.1 Introduction to Object Oriented Methodologies :Booch, Rumbaugh and Jacobson

2.2 Requirements Engineering Tasks, Requirement Elicitation Techniques, Software Requirements: Functional, Non- Functional

2.3 Requirements Characteristics, Requirement qualities, Requirement Specification, Requirement Traceability, System Analysis Model Generation, Documentation :Use Case Diagram, Activity Diagram

2.4 Categorizing classes: entity, boundary and control, Modelling associations and collections-Class Diagram

2.5 Dynamic Analysis - Identifying Interaction – Sequence and Collaboration diagrams, State chart diagram

Requirement Characteristics

Requirement Characteristics:

- **Complete:**
- All possible scenarios through the system are described
- All exceptional behaviour are mentioned (i.e., all aspects of the system are represented in the requirements model)
- **Consistent:**
- Not contradicting itself
- **Unambiguous:**
- one system is defined (i.e., it is not possible to interpret the specification two or more different ways)
- **Correct:**
- Represents accurately the system that the client needs and that the developers intend to build (i.e., everything in the requirements model accurately represents an aspect of the system to the satisfaction of both client and developer)

Requirement Characteristics

Requirement Characteristics:

•Realistic:

• Check if The system can be implemented within constraints

•Verifiable:

• Once the system is built, repeatable tests can be designed to demonstrate that the system fulfils the requirements specification

•Traceable:

•Each requirement can be traced throughout the software development to its corresponding system functions.

•Each system function can be traced back to its corresponding set of requirements.

•Enables a tester to assess the coverage of a test case, that is, to identify which requirements are tested and which are not.

•Enables the analyst and the developers to identify all components and system functions that the change would impact

Requirement Characteristics

Write down requirements of TicketDispenser

- **Complete:**
- **Consistent:**
- **Unambiguous:**
- **Correct:**

Requirements Elicitation Activities

- **Identifying Actors:**
  - Actors represent external entities that interact with the system. An actor can be human or an external system
- **Identifying Scenarios:**
  - Developers observe users and develop a set of detailed scenarios for typical functionality provided by the future system.
  - Scenarios are concrete examples of the future system in use.
  - Developers use these scenarios to communicate with the user and deepen their understanding of the application domain.
- **Identifying Use Cases:**
  - Developers derive from the scenarios a set of use cases that completely represent the future system.
  - Use cases are abstractions describing all possible cases.
  - When describing use cases, developers determine the scope of the system.

Requirements Elicitation Activities

- **Refining Use Cases:**
  - developers ensure that the requirements specification is complete by detailing each use case and describing the behaviour of the system in the presence of errors and exceptional conditions.
- **Identifying Relationships Among Actors and Use Cases:**
  - developers identify dependencies among use cases.
  - Consolidate the use case model by factoring out common functionality to ensures that the requirements specification is consistent.
- **Identifying Nonfunctional Requirements:**
  - Developers, users, and clients agree on aspects that are visible to the user, but not directly related to functionality.
  - These include constraints on the performance of the system, its documentation, the resources it consumes, its security, and its quality.

Requirements Analysis Document

1. Introduction
    - 1.1 Purpose of the system
    - 1.2 Scope of the system
    - 1.3 Objectives and success criteria of the project
    - 1.4 Definitions, acronyms, and abbreviations
    - 1.5 References
    - 1.6 Overview
2. Current system

Requirements Analysis Document

3. Proposed system

- 3.1 Overview
- 3.2 Functional requirements
- 3.3 Nonfunctional requirements
  - 3.3.1 Usability
  - 3.3.2 Reliability
  - 3.3.3 Performance
  - 3.3.4 Supportability
  - 3.3.5 Implementation
  - 3.3.6 Interface
  - 3.3.7 Packaging
  - 3.3.8 Legal
- 3.4 System models
  - 3.4.1 Scenarios
  - 3.4.2 Use case model
  - 3.4.3 Object model
  - 3.4.4 Dynamic model
  - 3.4.5 User interface—navigational paths and screen mock-ups

4. Glossary

Requirements Elicitation Activities
Actors

- **Questions for identifying actors:**
  - Which user groups are supported by the system to perform their work?
  - Which user groups execute the system's main functions?
    - Which user groups perform secondary functions, such as maintenance and administration?
  - With what external hardware or software system will the system interact?

Requirements Elicitation Activities

Scenarios

- **As-is scenarios** describe a current situation. The current system is understood by observing users and describing their actions as scenarios.
- These scenarios can then be validated for correctness and accuracy with the users.
- **Visionary scenarios** describe a future system.
- Visionary scenarios are used both as a point in the modelling space by developers as they refine their ideas of the future system and as a communication medium to elicit requirements from users.
- Visionary scenarios can be viewed as an inexpensive prototype.
- **Evaluation scenarios** describe user tasks against which the system is to be evaluated.
- The collaborative development of evaluation scenarios by users and developers also improves the definition of the functionality tested by these scenarios.
- **Training scenarios** are tutorials used for introducing new users to the system.
- These are step-by-step instructions designed to hand-hold the user through common tasks.

Requirements Elicitation Activities

Questions to identify Scenarios

- What are the tasks that the actor wants the system to perform?
- What information does the actor access? Who creates that data? Can it be modified or removed? By whom?
- Which external changes does the actor need to inform the system about? How often? When?
- Which events does the system need to inform the actor about? With what latency?

Requirements Elicitation Activities

Use Cases

- A scenario is an instance of a use case; that is, a use case specifies all possible scenarios for a given piece of functionality.
- A use case is initiated by an actor.
- After its initiation, a use case may interact with other actors, as well.
- A use case represents a complete flow of events through the system in the sense that it describes a series of related interactions that result from its initiation.

Requirements Elicitation Activities

Guidelines for Use Cases

- Use cases should be named with verb phrases. The name of the use case should indicate what the user is trying to accomplish.
- Actors should be named with noun phrases.
- The boundary of the system should be clear. Steps accomplished by the actor and steps accomplished by the system should be distinguished (system actions are indented to the right).
- Use case steps in the flow of events should be phrased in the active voice to explicitly represent who accomplished the step.
- The causal relationship between successive steps should be clear.
- A use case should describe a complete user transaction.
- Exceptions should be described separately.
- A use case should not describe the user interface of the system.
- A use case should not exceed two or three pages in length. Otherwise, use include and extend relationships to decompose it in smaller use cases

Requirements Elicitation Activities

Use Cases

Identifying Use Cases

Use cases represent what the actors want your system to do for them.

Use case represents a complete flow of events through the system.
A Use Case captures some actor-visible function.
- Achieves some discrete (business-level) goal for that actor.
- May be read, write, or read-modify-write in nature.

Requirements Elicitation Activities

Use Cases

Names begin with a verb – An use case models an action so the name should begin with a verb.

Make the name descriptive – This is to give more information for others who are looking at the diagram. For example “Print Invoice” is better than “Print”.

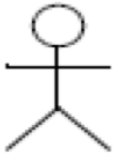
Highlight the logical order – For example if you’re analyzing a bank customer typical use cases include open account, deposit and withdraw. Showing them in the logical order makes more sense.

Place included use cases to the right of the invoking use case – This is done to improve readability and add clarity.

•**Place inheriting use case below parent use case** – Again this is done to improve the readability of the diagram.

Requirements Elicitation Activities

Elements of Use Case



Actor

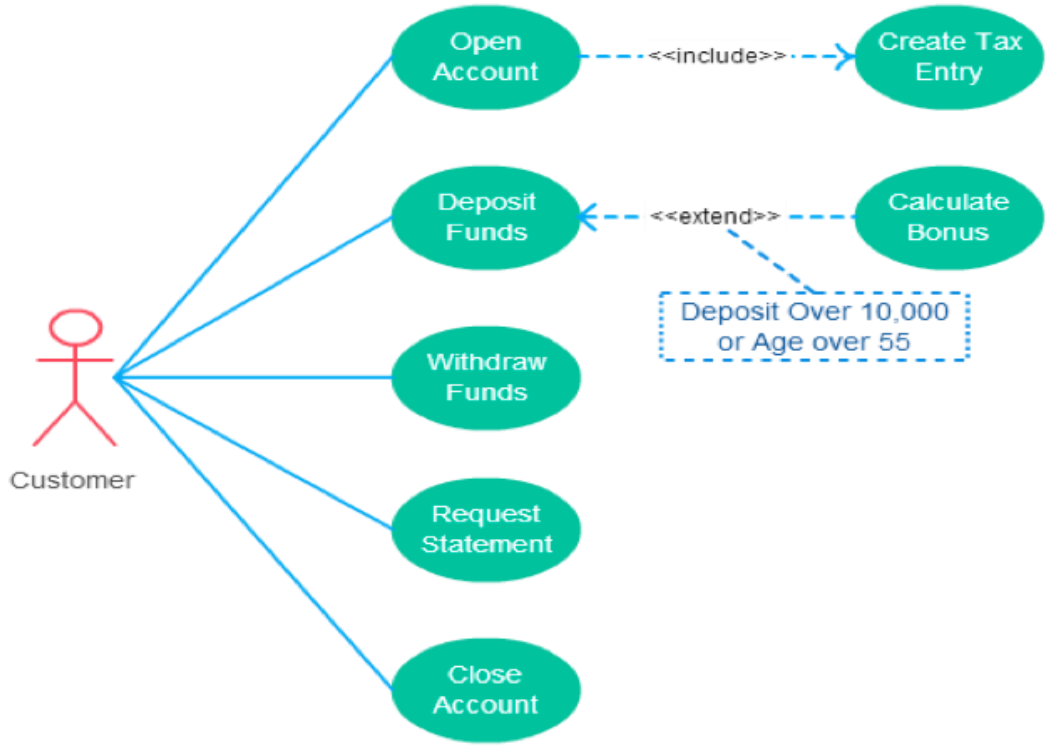
Connection between Actor and Use Case

Boundary of system

System function (process – automated or manual).

Requirements Elicitation Activities

Use Case



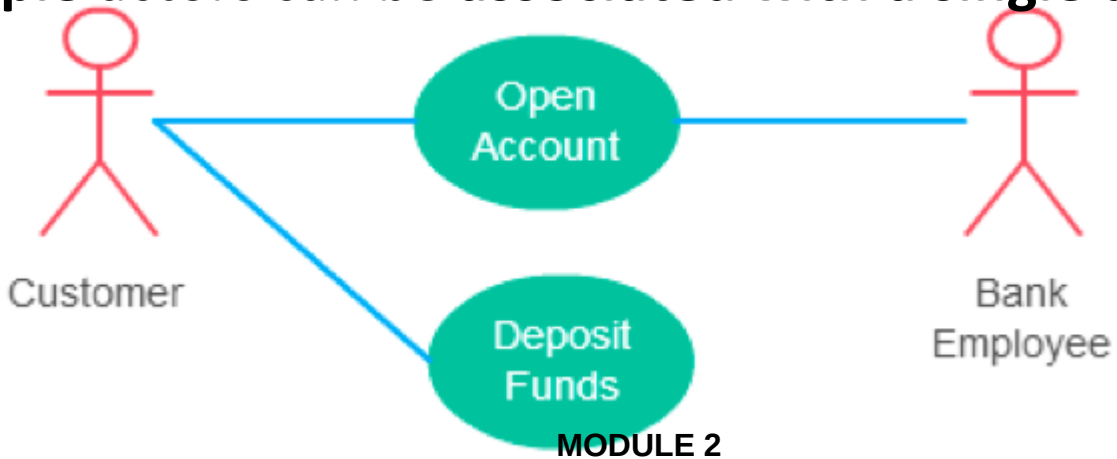
Relationship types in Use Case Diagram

1. Association between actor and use case
2. Generalization of an actor
3. Extend between two use cases
4. Include between two use cases
5. Generalization of a use case

Use Case Diagram

1. Association between actor and use case

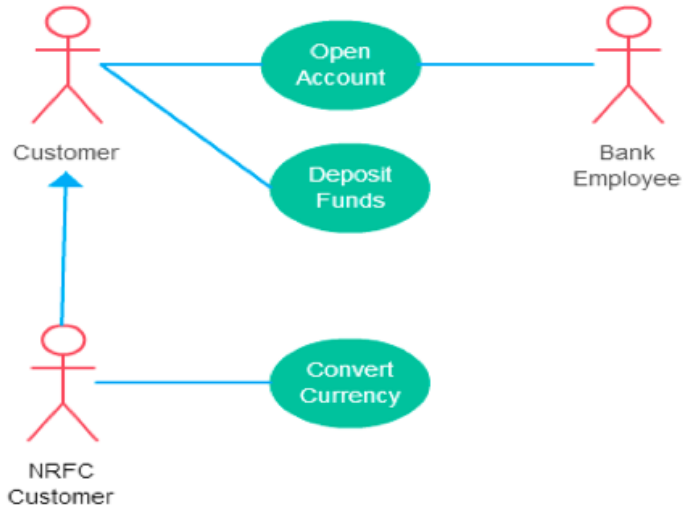
- This one is straightforward and present in every use case diagram.
- An actor must be **associated with at least one use case.**
- An actor can be **associated with multiple use cases.**
- **Multiple** actors can be **associated with a single use case.**



Use Case Diagram

2. Generalization of actor and use case

- Generalization of an actor means that one actor can inherit the role of other actor.



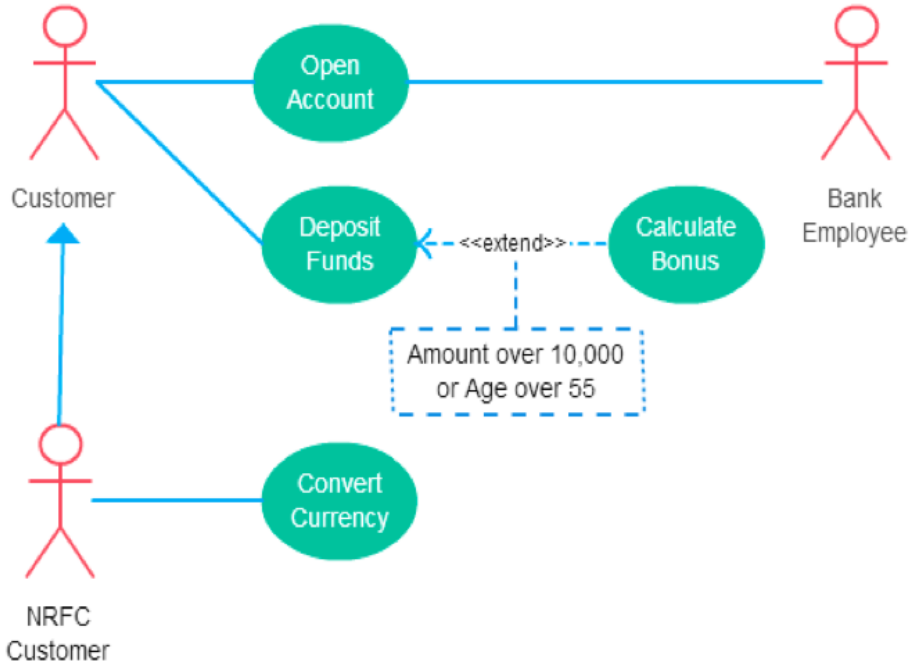
Use Case Diagram

3. Extend between two use case

- It extends the base use case and adds more functionality to the system.
- **The extending use case is dependent on the (base) use case**.
- **The extending use case is usually optional and can be triggered conditionally.**
- **The extended (base) use case must be meaningful on its own**.
- Arrow points to the base use case when using <<extend>>
- Extending use case is optional

Use Case Diagram

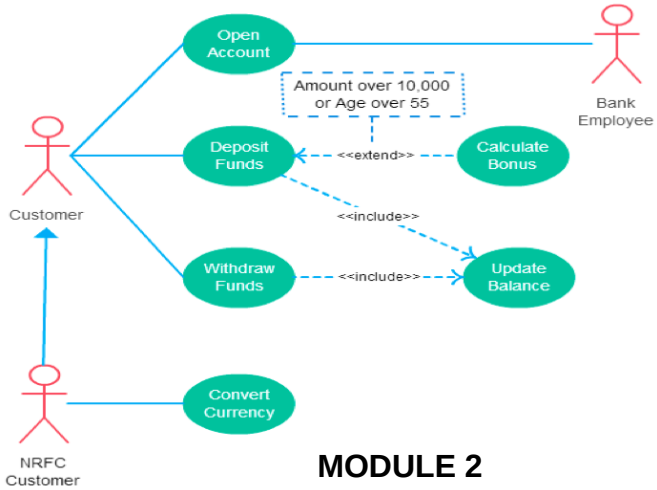
3. Extend between two use case



Use Case Diagram

4. Include between two use case

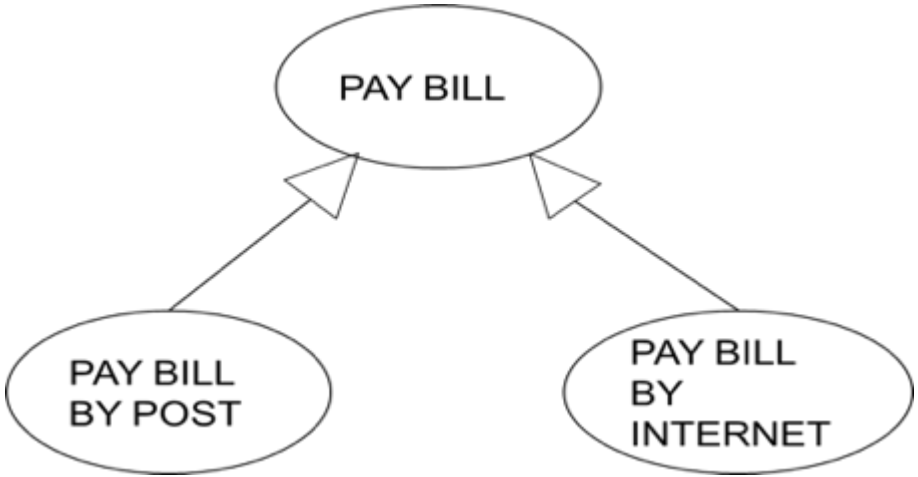
- Include relationship show that the behavior of the included use case is part of the including (base) use case.
- The base use case is incomplete without the included use case.
- The included use case is mandatory and not optional.
- Like a “use case subroutine”



Use Case Diagram

5. Generalisation of a use case

- This is similar to the generalization of an actor.
- This is used when there are common behavior between two use cases. and also specialized behavior specific to each use case.



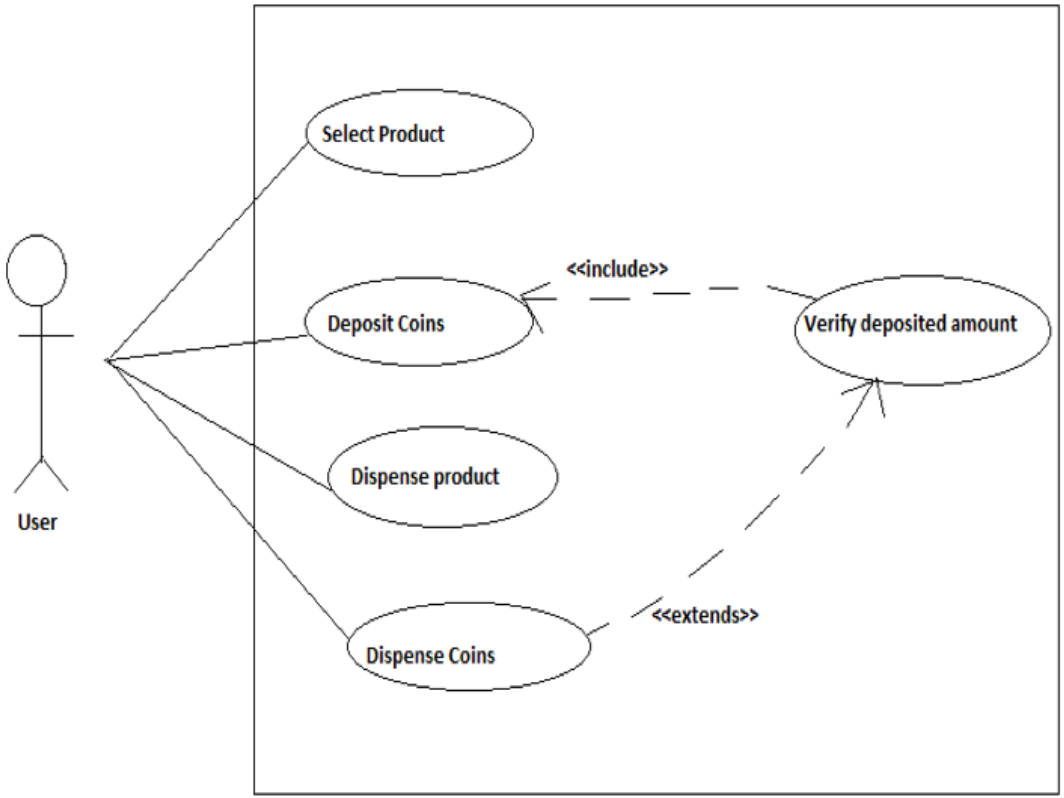
Draw use case diagram.

A vending machine accepts coins for a variety of products. The user selects the drink from products available through the selection panel. If the drink is available the price of the product is displayed. The user the deposits the coins depending on the price of the product. Coin collector collects the coins. After stipulated time, the controller will compare the deposited coins with the price, If the amount deposited is less than the price then error message will be displayed and all deposited coins will be dispensed by coin dispenser else the drink will be dispensed by the product dispenser.

Use Case Diagram for Vending Machine

- Actor- User
- Processes involved in the system
  - Select product
  - Deposit coins
  - Verify the deposited amount
  - Dispense product
  - Dispense coins

Use Case Diagram for Vending Machine



Use Cases

Template for Use Cases

Use Case name	<variable used to identify the use case>
Participating actors instances	<mention who will be a part of this even>
Flow of event	• Step 1 • Step 2 • ….
Entry Condition	
Exit Condition	
Quality Requirements	

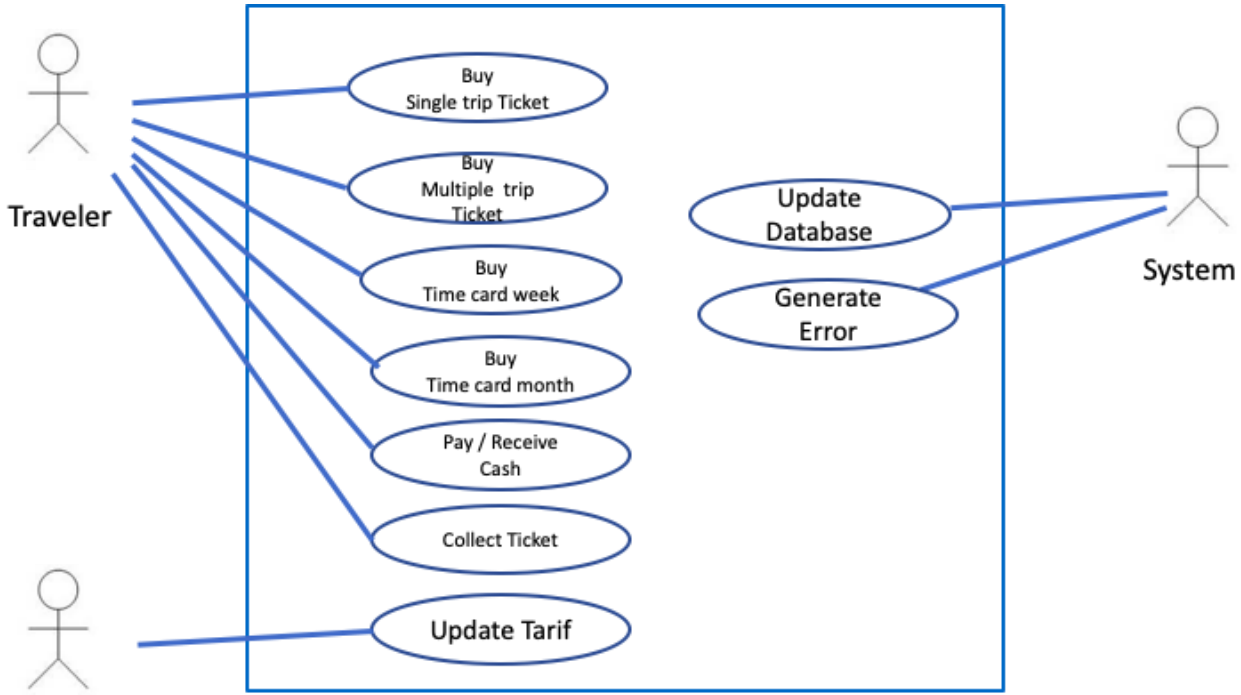
Requirements Elicitation Activities

Sample Use Case: TicketDistributor

Use case name:	*Issue of ticket*
Participating actors instances	*Initiated by: Passenger* *Communicates with computer system*
Flow of event	1. Passenger Arrives at the TicketDistributor machine and presses a button to start 2. Machine displays zones, rates, options as chicle/ adult, single multiple journey 3. Passenger selects the appropriate details 4. Machine computes and displays amount to be paid 5. Passenger pays the amount 6. Machine checks the amount if amount is sufficient 7. Machine prints the ticket and dispenses ticket 8. returns extra money (if the case is)
Entry Condition	• The passenger presses start action button
Exit Condition	• The passenger receives ticket • Collects remaining amount OR • Gets a messages , no printout possible • No change is available
Quality requirements	• The machine should display if it does not have money to return • Printing must be finished within 60 seconds

Analysis Model

Use Case Diagram



Requirements Elicitation Activities

•During this Covid-19 pandemic, government was to regulate permission to organize *Ganpati Utsav*, *Navaratri Mandals* etc. the authority is interested in setting up a computerised system. It should be an end to end solution providing a facility for various *Mandals* such as applying for permission rom municipality, electricity board, local police station, firebrigade etc. allocating/ allowing inspection by the officers. Arranging for *visarjan* of the idols.

•Write down

- •Stakeholders
- •Functional requirements
- •Nonfunctional requirements
- •Actors
- •Scenarios
- •Activities
- •Use cases

Analysis Model

Analysis Model

•Analysis results in a model of the system that aims to be correct, complete, consistent, and unambiguous.
•Developers formalize the requirements specification produced during requirements elicitation and examine in more detail boundary conditions and exceptional cases.
•Developers validate, correct and clarify the requirements specification if any errors or ambiguities are found.
•The client and the user are usually involved in this activity when the requirements specification must be changed and when additional information must be gathered.

Analysis Model

Analysis Model

•In object-oriented analysis, developers build a model describing the application domain.

•The analysis model is then extended to describe how the actors and the system interact to manipulate the application domain model.

•Developers use the analysis model, together with non-functional requirements, to prepare for the architecture of the system developed during high-level design.

Analysis Model

Analysis Activities

- Identification of objects
- Their behaviour
- Their relationships
- Their classification
- Their organization

Analysis Model

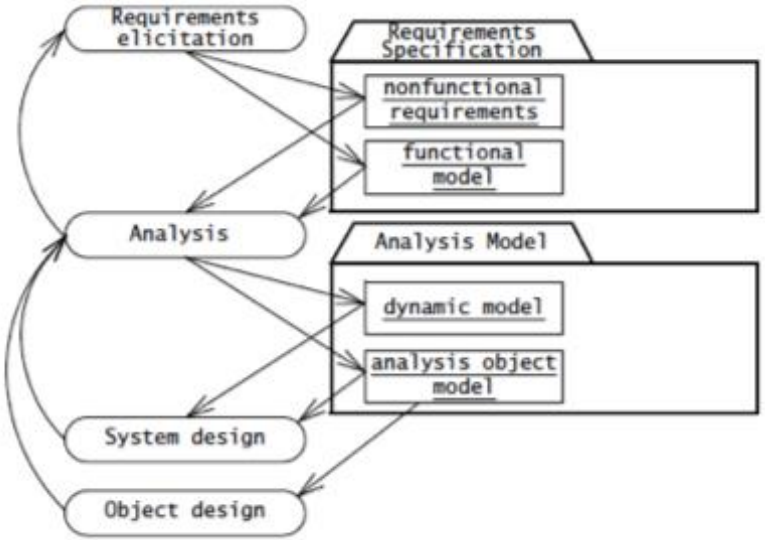
Analysis

What you See?

- Specifications may contain ambiguities caused by
  - The inaccuracies inherent to natural language
  - The assumptions of the specification authors

Analysis Model

Analysis focuses to prepare Analysis model, which is correct, complete, consistent, and verifiable.



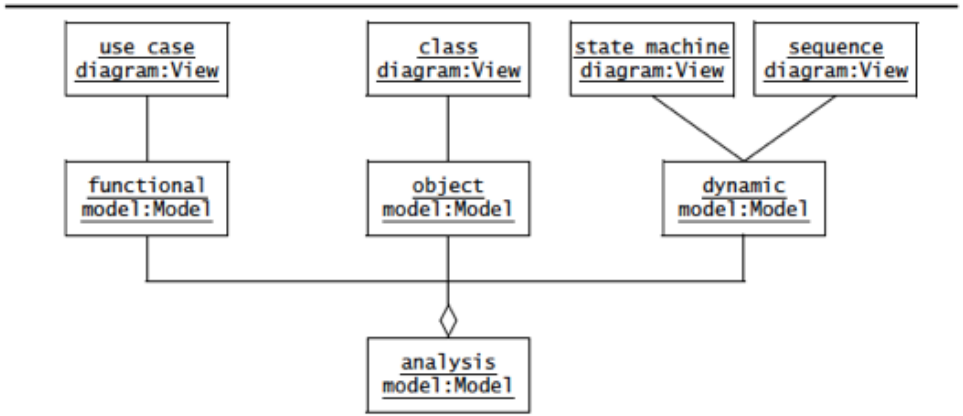
Analysis Model

Types of Analysis Models

a)The functional model :represented by use cases and scenarios

b)The analysis object model: represented by class and object diagrams

c)The dynamic model: represented by state machine and sequence diagrams



Analysis Model

These models represent user-level concepts, not actual software classes components

- **Analysis Object Model**
  - Focuses on the individual concepts that are manipulated by the system, their properties and their relationships.
  - Depicted with UML class diagrams, includes classes, attributes, and operations.
  - A visual dictionary of the main concepts visible to the user.
- **Dynamic model**
  - Focuses on the behaviour of the system.
  - Depicted with sequence diagrams and with state machines.
  - Sequence diagrams represent the interactions among a set of objects during a single use case.
  - State machines represent the behaviour of a single object (or a group of very tightly coupled objects).
  - Serves to assign responsibilities to individual classes and, in the process, to identify new classes, associations, and attributes to be added to the analysis object model.

Analysis Model

Activity Diagram

- An activity diagram describes the **behavior of a system in terms of activities.**
- **Activity diagram** is basically a **flowchart** to represent the flow from **one activity to another activity.**
- Activity diagram: **operation of a system**.
- This flow can be **sequential, branched or concurrent.**

Analysis Model

Activity Diagram Basic Notations

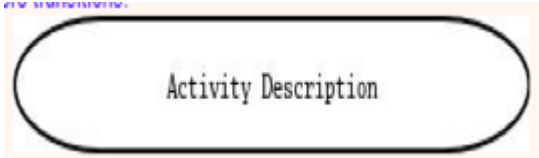
- **Initial node**.
  - The filled in circle is the starting point of the diagram.
  - It is shown as filled circle.

Analysis Model
Activity Diagram Basic Notations

- **Final node**.
  - The filled in circle with a border at the end point of the diagram.
  - It is optional

Analysis Model
Activity Diagram Basic Notations

- **Activity**. The rounded rectangles represent activities that occur



- **Flow/edge (Control Flow)**

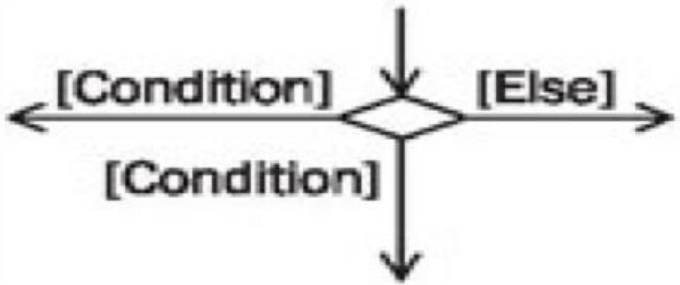
The arrows on the diagram. Within the control flow an incoming arrow starts a single step of an activity; after the step is completed the flow continues along the outgoing arrow.

Analysis Model
Activity Diagram Basic Notations

- **Decision:** are branches in the control flow.
  - They denote alternatives based on a condition of the state of an object or a set of objects.
  - Decisions are depicted by a diamond with one or more incoming open arrows and two or more outgoing arrows.
  - The outgoing edges are labelled with the conditions that select a branch in the control flow.
  - Decisions are depicted by a diamond with one or more incoming open arrows and two or more outgoing arrows.
  - The set of all outgoing edges from a decision represents the set of all possible outcomes.

Analysis Model
Activity Diagram Basic Notations

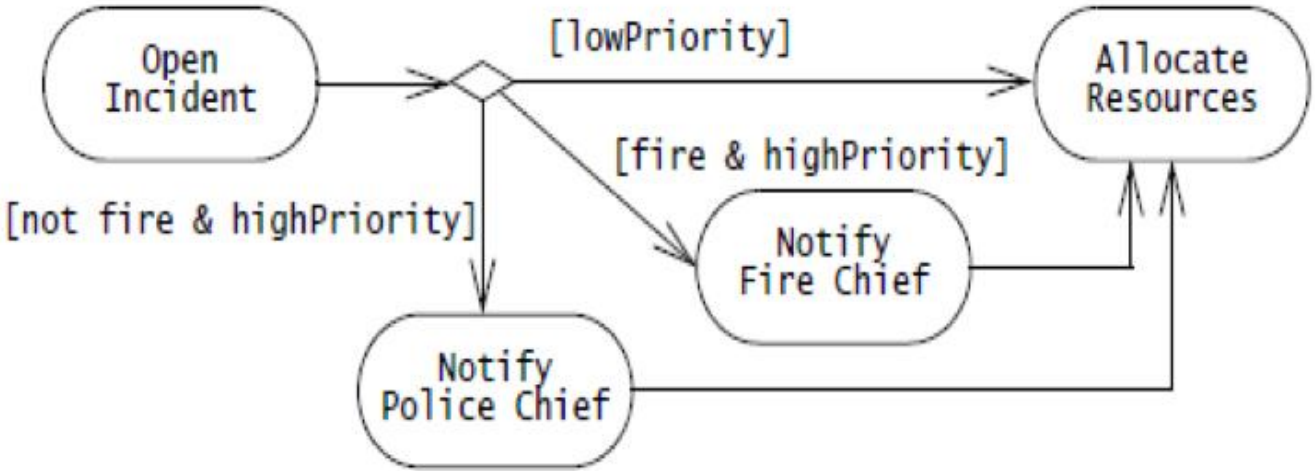
- **Decision:**



Analysis Model

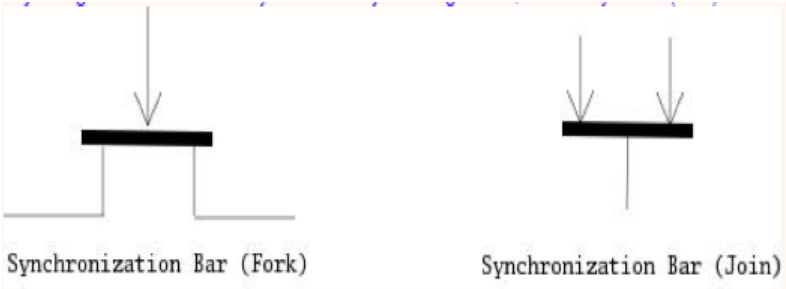
Activity Diagram Basic Notations

- **Decision:**

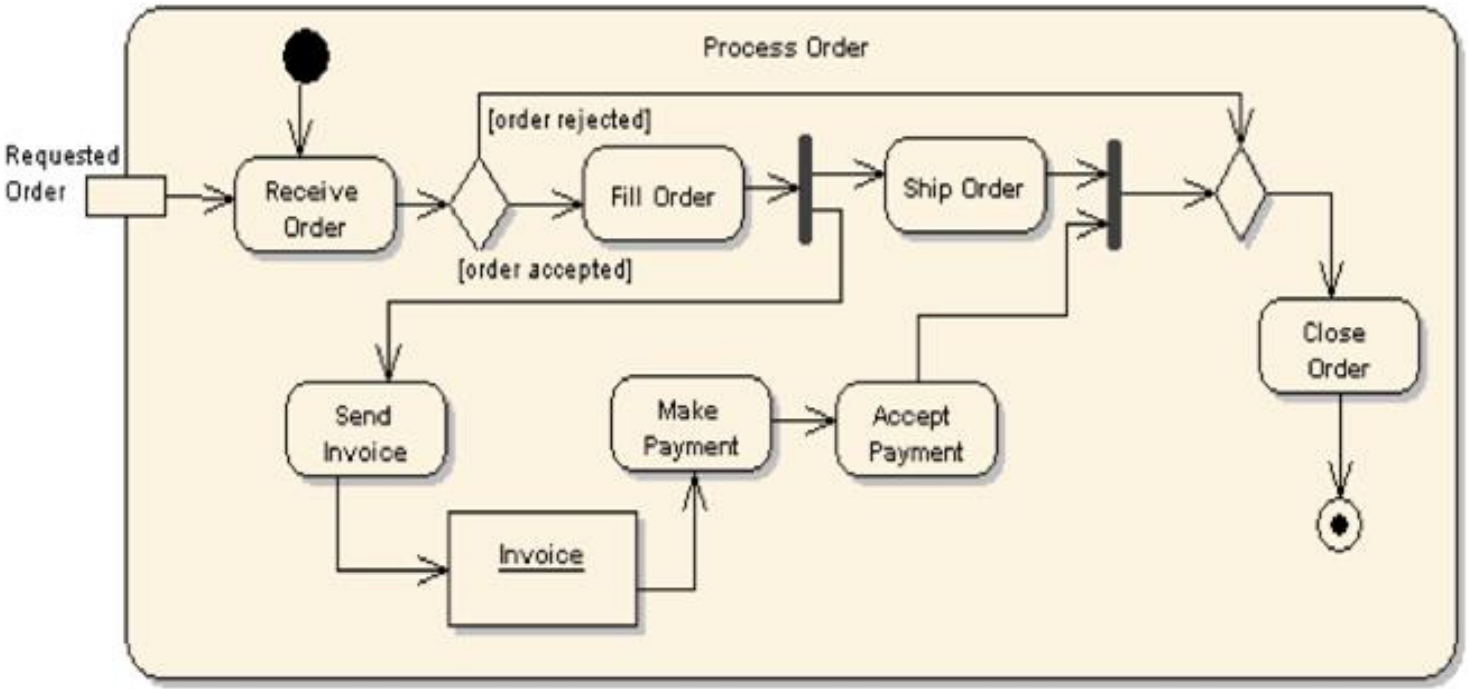


Analysis Model
Activity Diagram Basic Notations

- Some **activities** occur simultaneously or in **parallel**. Such activities are called **concurrent activities.**
- **Fork nodes** denote the **splitting of the flow** of control into multiple threads.
- Join nodes denotes the **synchronization of multiple threads** and their **merging of the flow** of control into a **single thread**.
- **Fork nodes** and **join nodes** represent concurrency.



Analysis Model

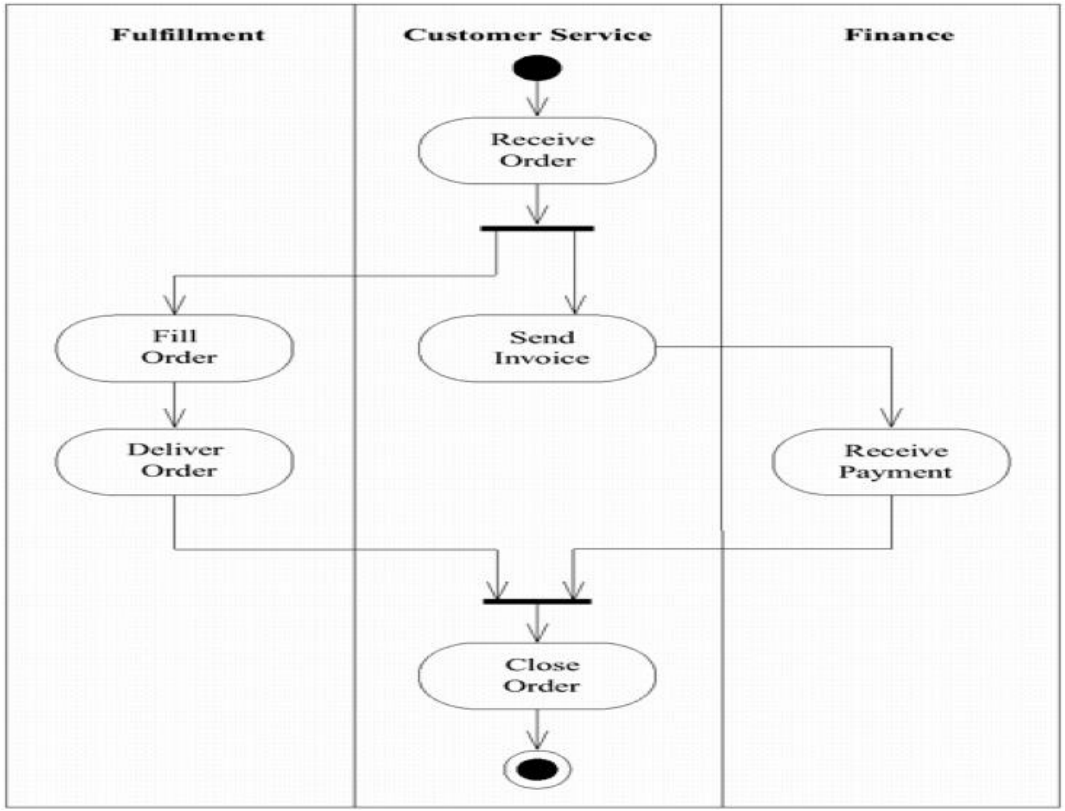


Analysis Model

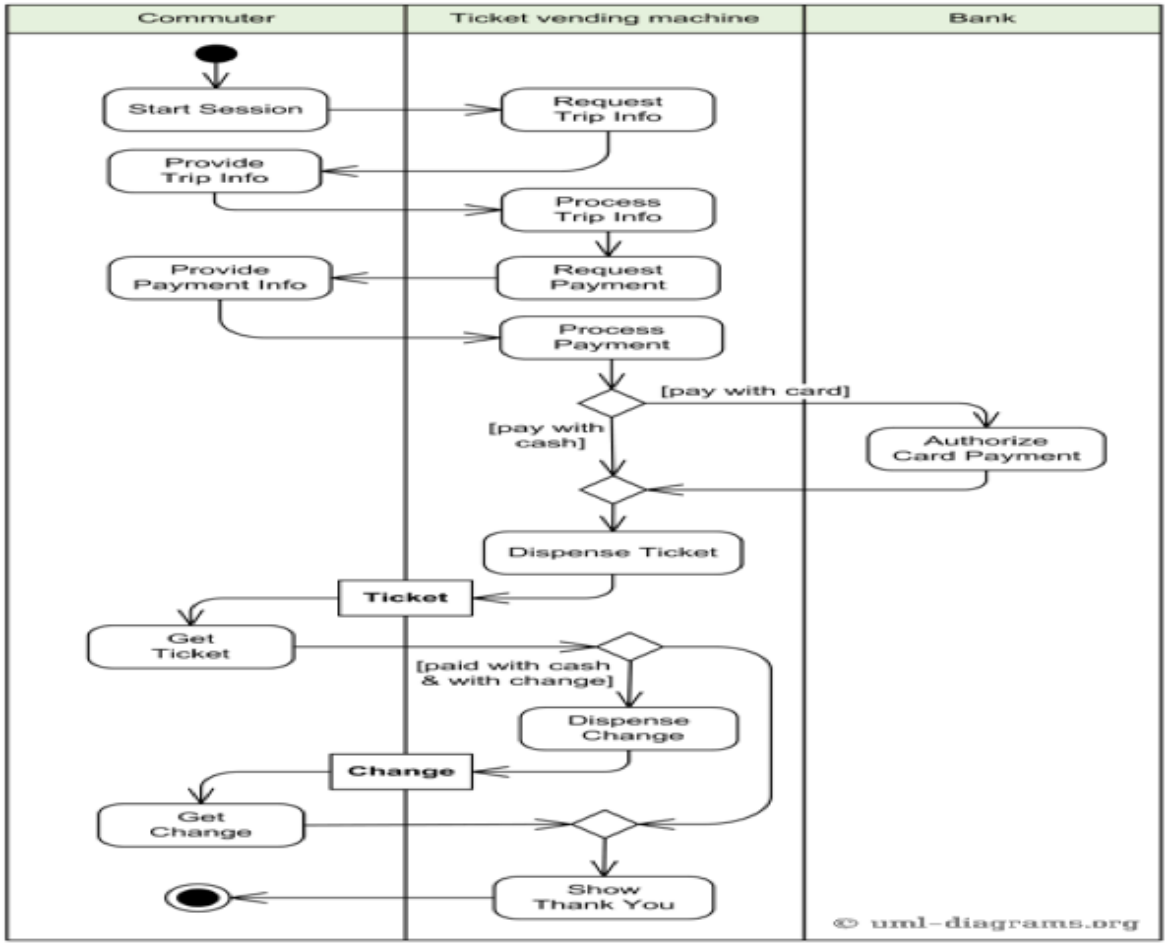
Activity Diagram : Swimlane

- The contents of an activity diagram may be organized into *partitions* (swimlanes) using solid vertical lines.
- A swim lane (or **swimlane diagram**) is a visual element used in process flow **diagrams**, or flowcharts, that visually distinguishes job sharing and responsibilities for sub-processes of a business process.
- Order Swimlanes in a Logical Manner.
- Apply Swimlanes To Linear Processes.
- Have Less Than Five Swimlanes.
- Consider Swim areas For Complex Diagrams.

Analysis Model



Analysis Model



Analysis Model

Sequence Diagram:

- The Sequence Diagram **models the collaboration of objects based on a time sequence.**
- **Sequence diagrams** represent the objects participating in the interaction horizontally and time vertically.
- Depicts sequence of actions that occur in a system
- Useful tool to **represent dynamic behavior** of a system
- 2 dimensional :
  - -Horizontal axis
  - -Vertical axis

Analysis Model

Sequence Diagram:

•The focus is **less on messages** themselves and more on the **order in which messages.**

•Sequence diagrams are good at showing **which objects communicate** with which other objects; and what messages **trigger those communications.**

Analysis Model

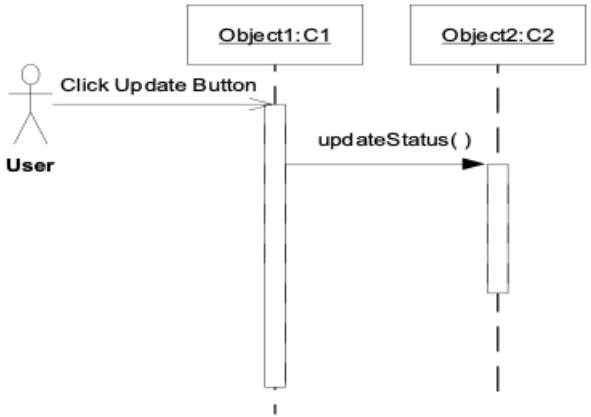
Participant in Sequence Diagram:

- A sequence diagram is made up of a collection of participants.
- Participants – the system parts that interact each other during the sequence.
- Classes or Objects – each class (object) in the interaction is represented by its named icon along the top of the diagram.

Analysis Model

Elements of Sequence Diagram:

- Actor
- Objects
- Lifelines
- Messages
- Activation: represents time an object needs to complete task



Analysis Model

Elements of Sequence Diagram:

- **Class Roles or Participants or Objects**:

Class roles describe the way an object will behave in context.

Activation or Execution Occurrence

Activation boxes represent the time an object needs to complete a task.

- When an object is busy executing a process or waiting for a reply message, use a thin gray rectangle placed vertically on its lifeline.

Analysis Model
Elements of Sequence Diagram:

- **Synchronous Message**
- A synchronous message requires a response before the interaction can continue.

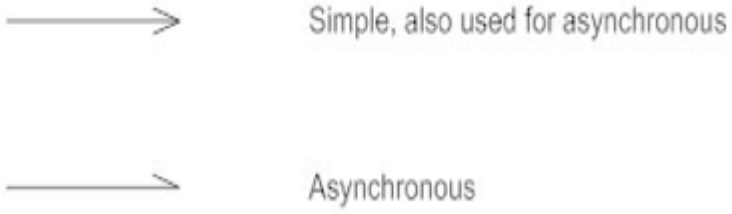
- It's usually drawn using a line with a solid arrowhead pointing from one object to another.

Analysis Model
Elements of Sequence Diagram:

Asynchronous Message

Asynchronous messages don't need a reply for interaction to continue.

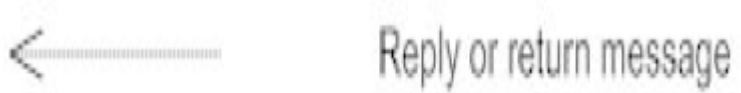
They are drawn with an arrow connecting two lifelines; however, the arrowhead is usually open and there's no return message depicted



Analysis Model
Elements of Sequence Diagram:

- **Reply or Return Message**

A reply message is drawn with a dotted line and an open arrowhead pointing back to the original lifeline

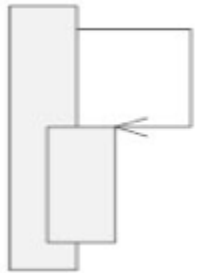


Analysis Model

Elements of Sequence Diagram:

- **Self Message**

- A message an object sends to itself, usually shown as a U shaped arrow pointing back to itself.



Self message

Analysis Model

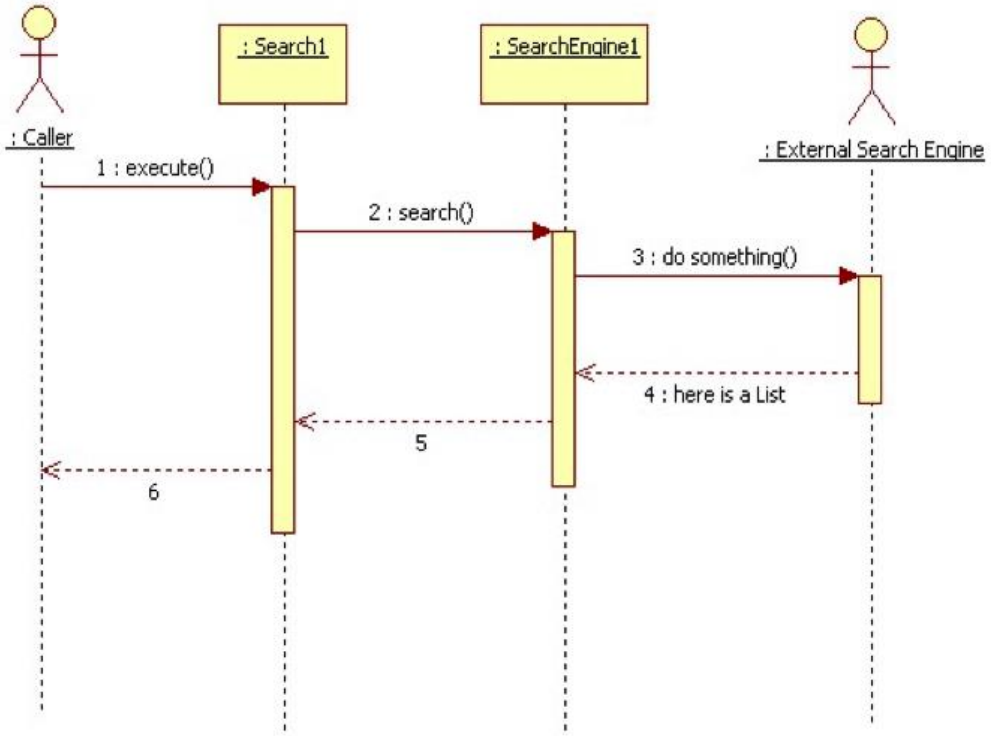
Elements of Sequence Diagram:

•Lost: A lost message occurs when the sender of the message is known but there is no reception of the message.

•Found: A found message indicates that although the receiver of the message is known in the current interaction fragment, the sender of the message is unknown.

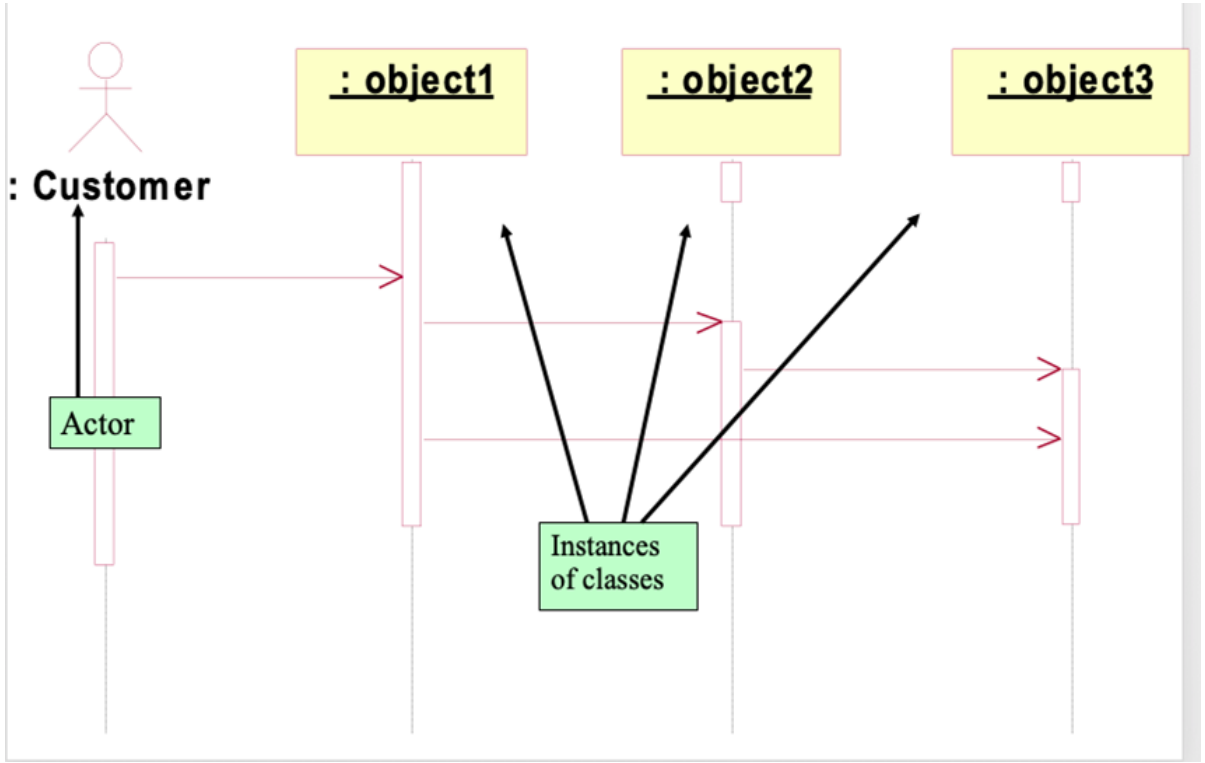
Analysis Model

Sequence Diagram



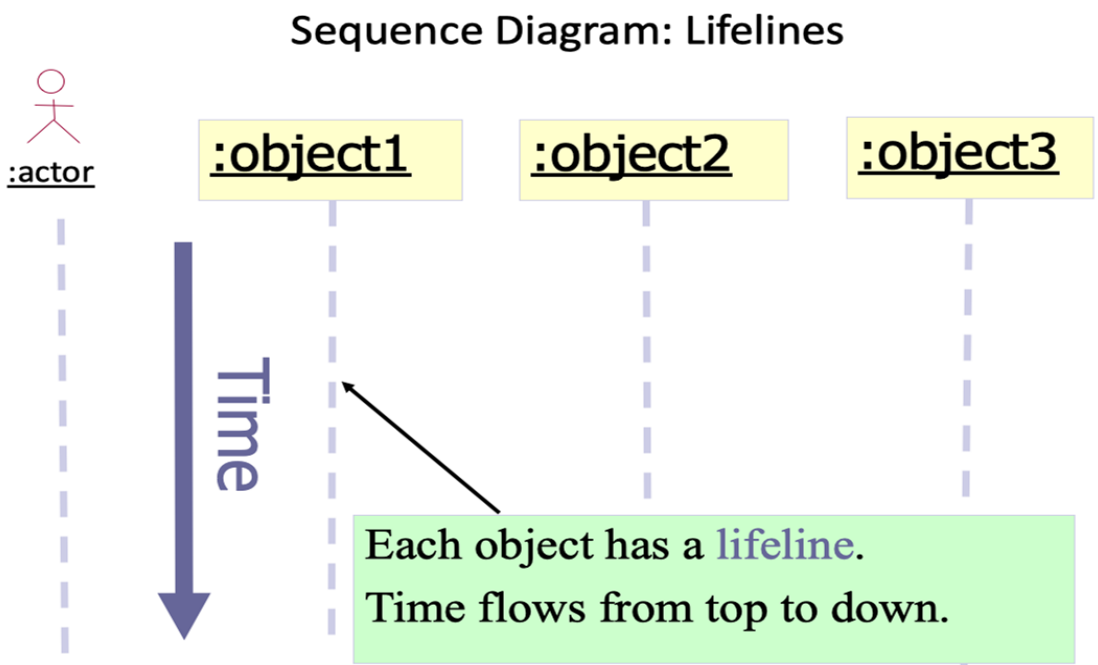
Analysis Model

Sequence Diagram



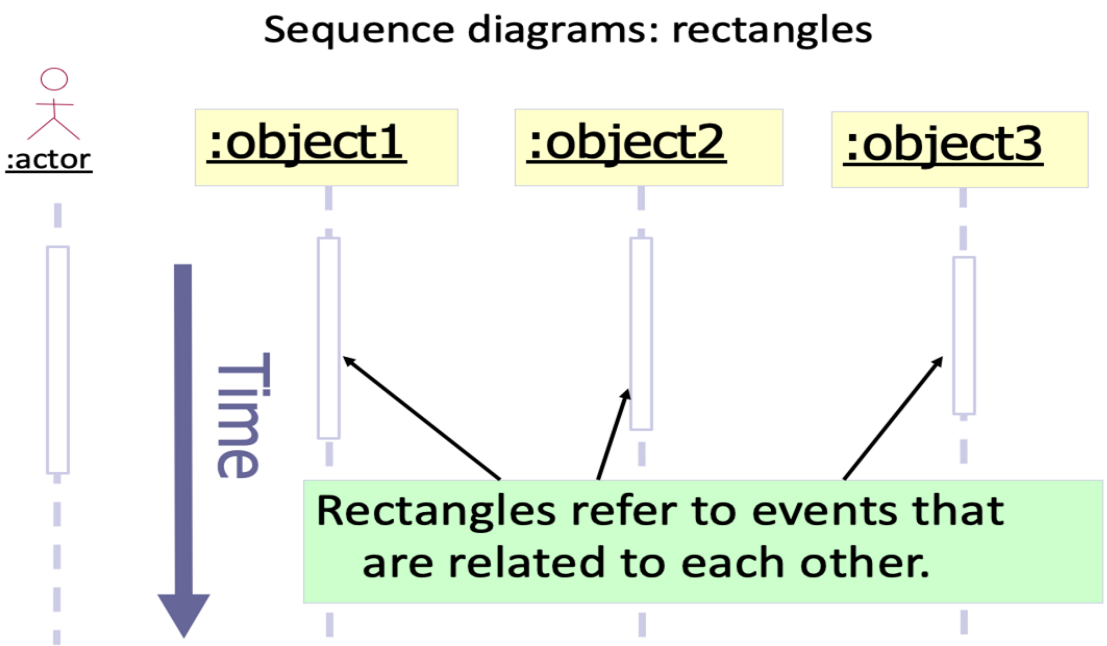
Analysis Model

Sequence Diagram



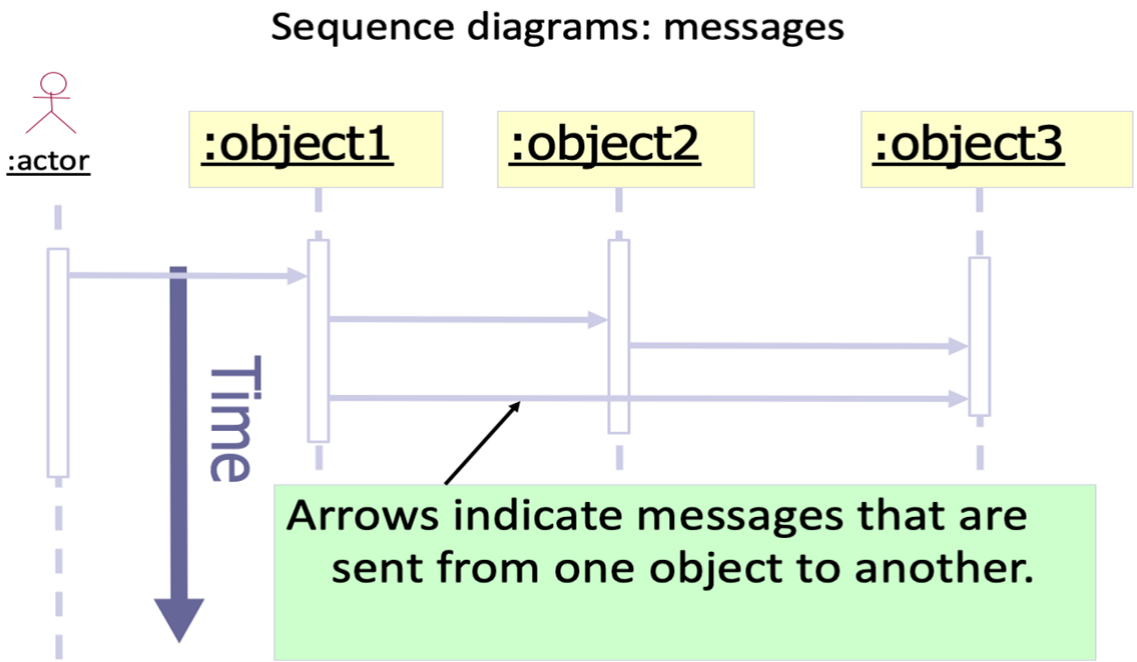
Analysis Model

Sequence Diagram



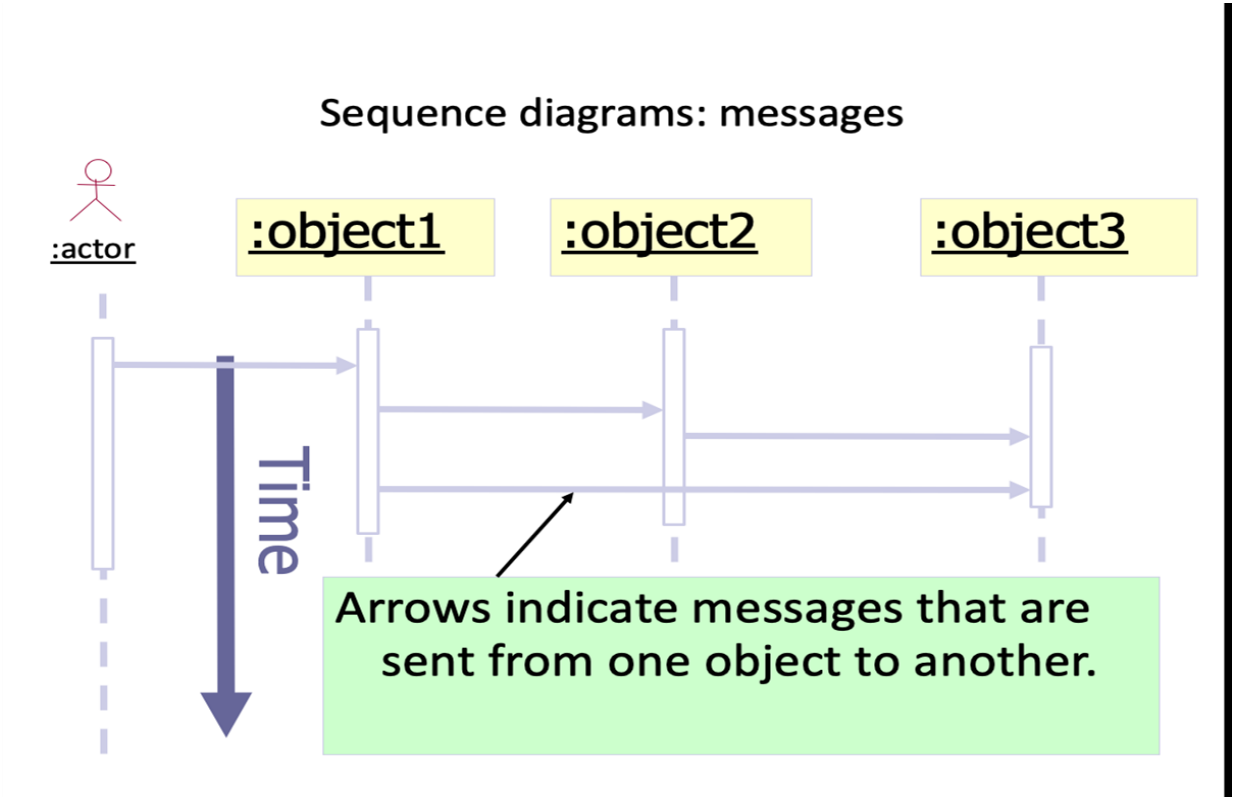
Analysis Model

Sequence Diagram



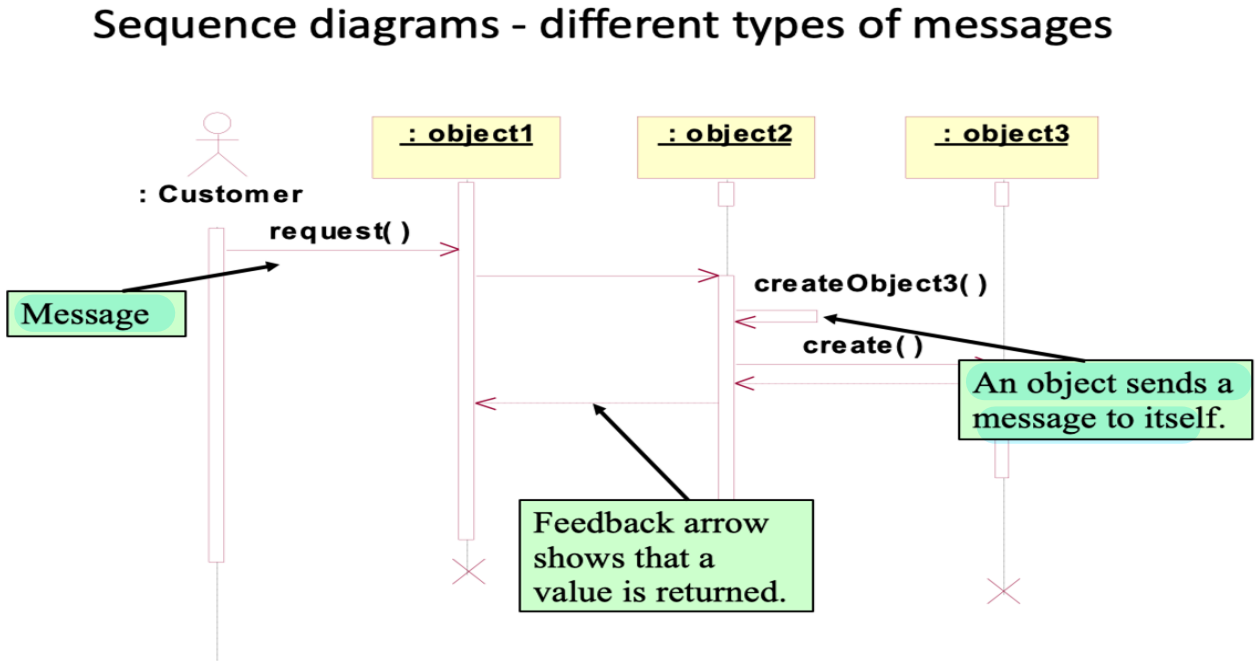
Analysis Model

Sequence Diagram



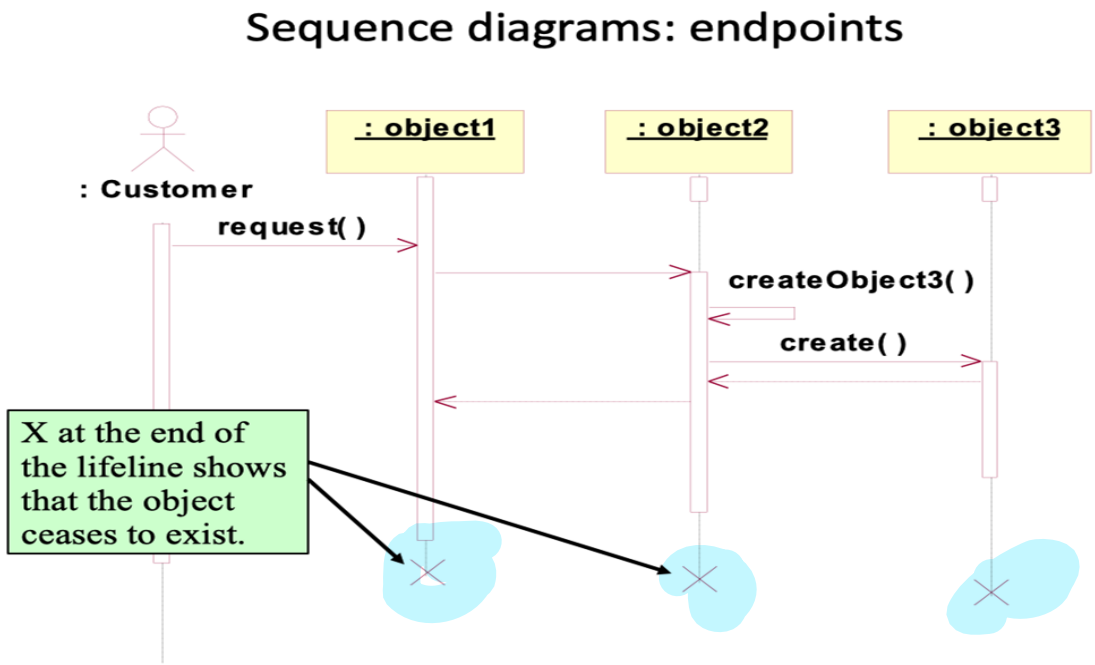
Analysis Model

Sequence Diagram



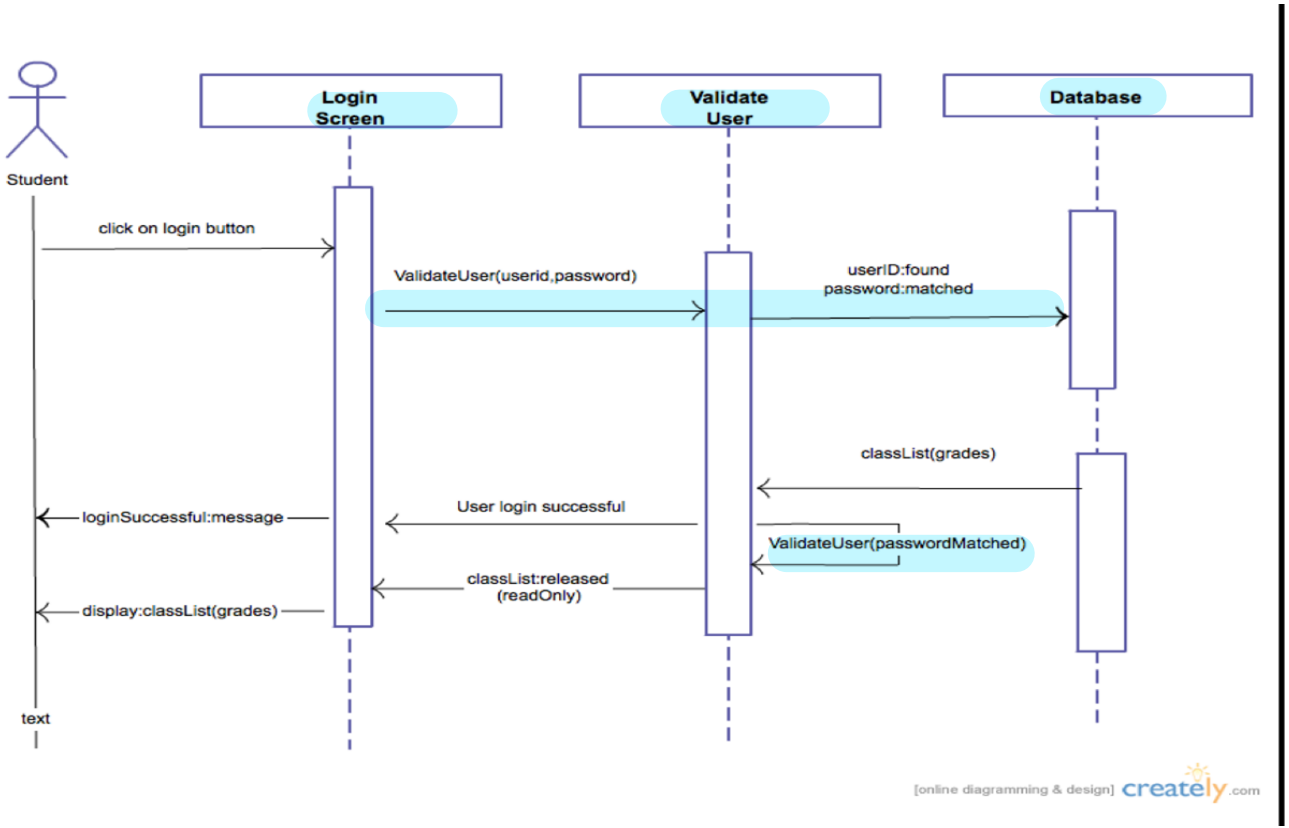
Analysis Model

Sequence Diagram

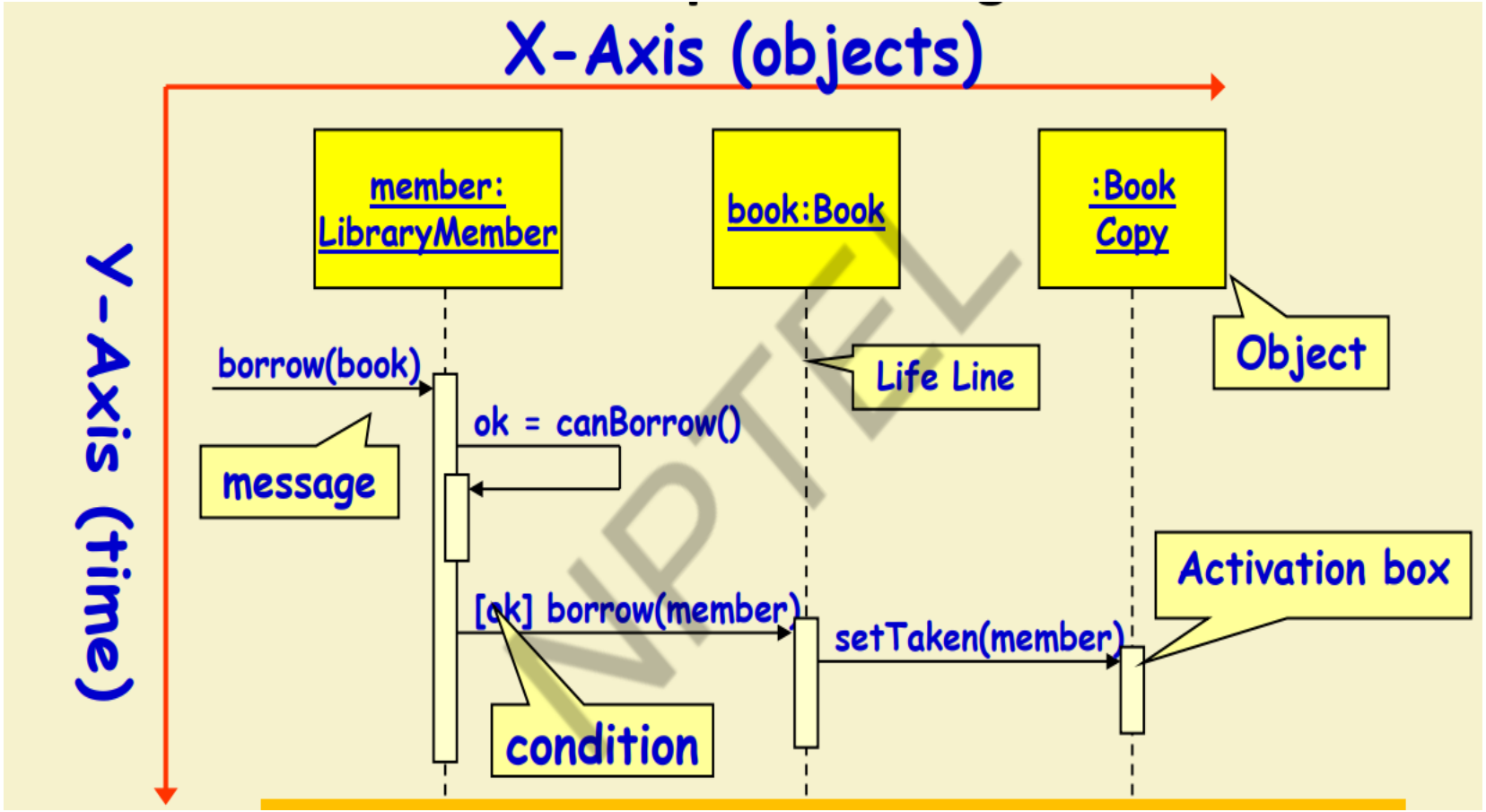


Analysis Model

Sequence Diagram diagram for login process

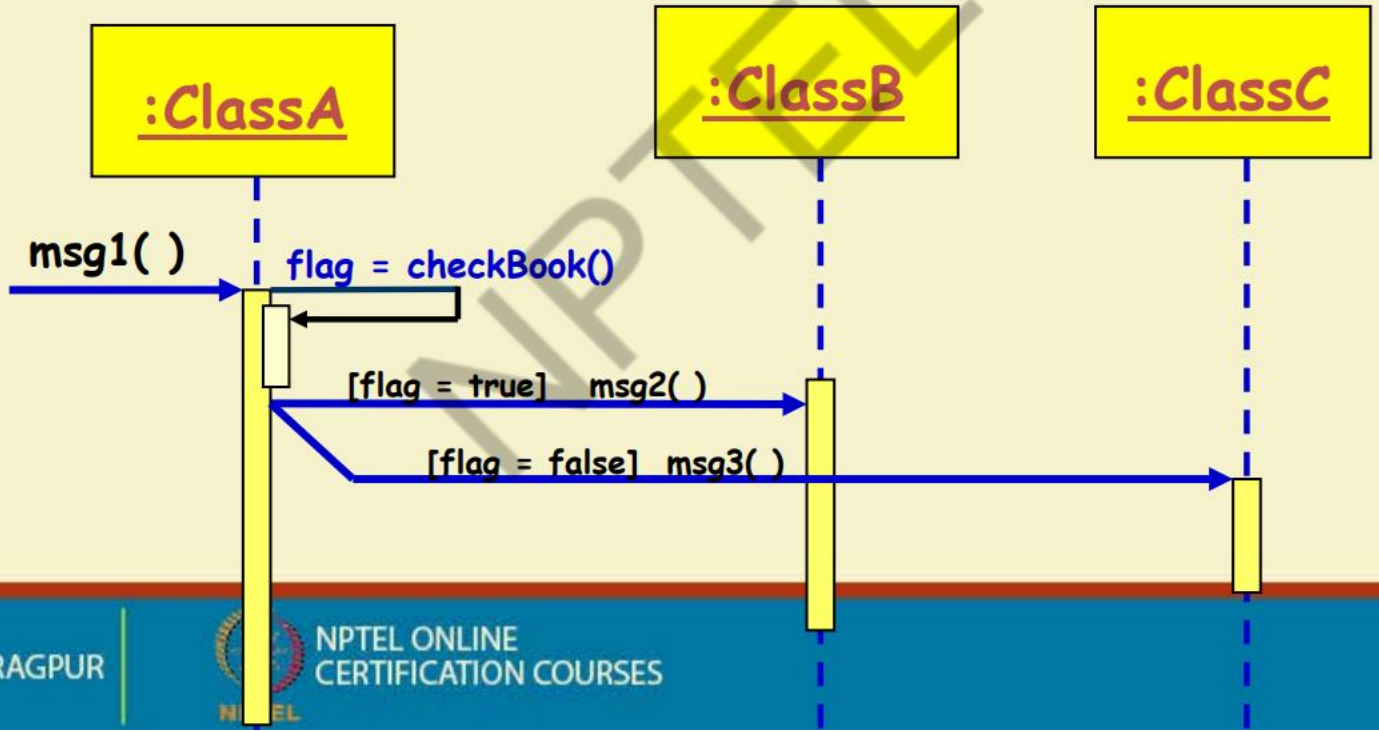


Sequence diagram to borrow book



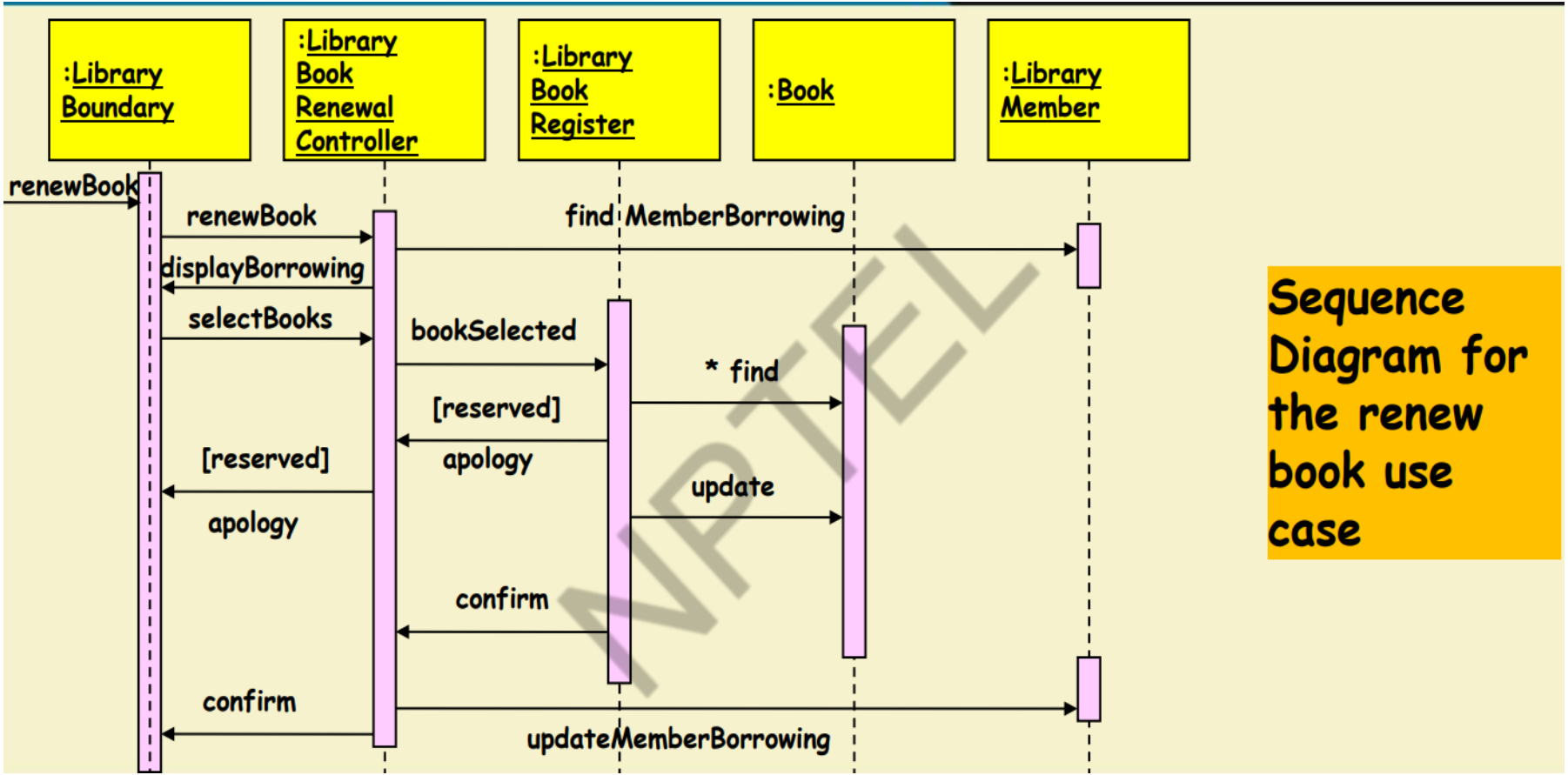
Ref: NPTEL course on software engineering, Ranbir mall

How to represent Mutually exclusive conditional invocations? If book is available, invoke msg2 on ClassB else invoke msg3 on classC,



Ref: NPTEL course on software engineering, Ranbir mall

Sequence diagram for the renew book



Sequence Diagram for the renew book use case

Ref: NPTEL course on software engineering, Ranbir mall

Analysis Model

Class Diagram: describes structure of the system

shows the system's

- Classes
- Attributes
- Operations (or methods),
- Relationships among the classes.

Analysis Model

Class Diagram

Class: A class represents the blueprint of its objects.

- Attributes
- Operations
- Relationships
  - Associations
  - Generalization
  - Realization
  - Dependencies.

Analysis Model

Class Diagram

Class: Describes a set of objects having similar:

–Attributes (status)

–Operations (behavior)

–Relationships with other classes

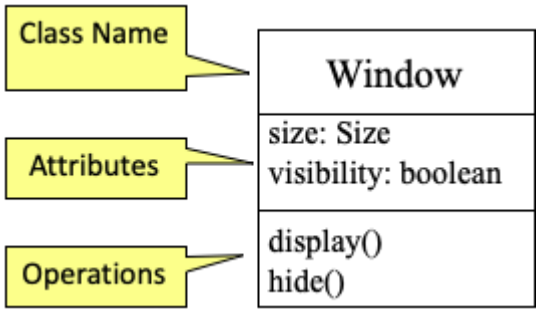
•Attributes and operations may have their visibility marked:

"+" for *public*

"#" for *protected*

"–" for *private*

"~" for *package*



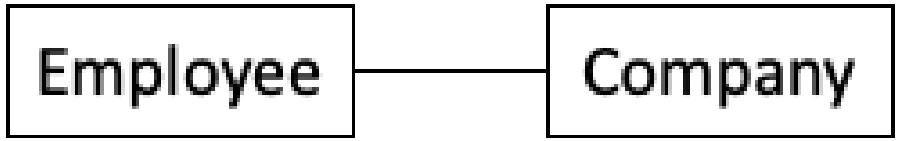
Analysis Model

Class Diagram

Class: Association Relationship

An association between two classes indicates that objects at one end of an association “recognize” objects at the other end and may send messages to them.

Example: “An Employee works for a Company”

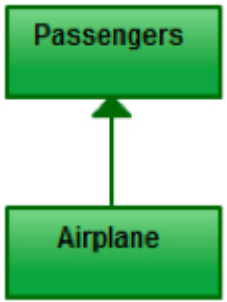


Analysis Model

Class Diagram

Class: Direct Association Relationship

Refers to a **directional relationship** represented by a line with an arrowhead. The arrowhead depicts a container-contained directional flow.

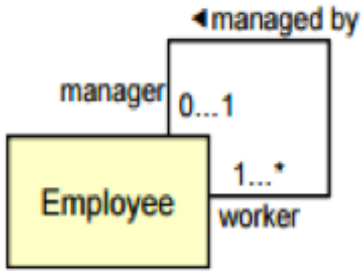


Analysis Model

Class Diagram

Class: Reflexive Association Relationship

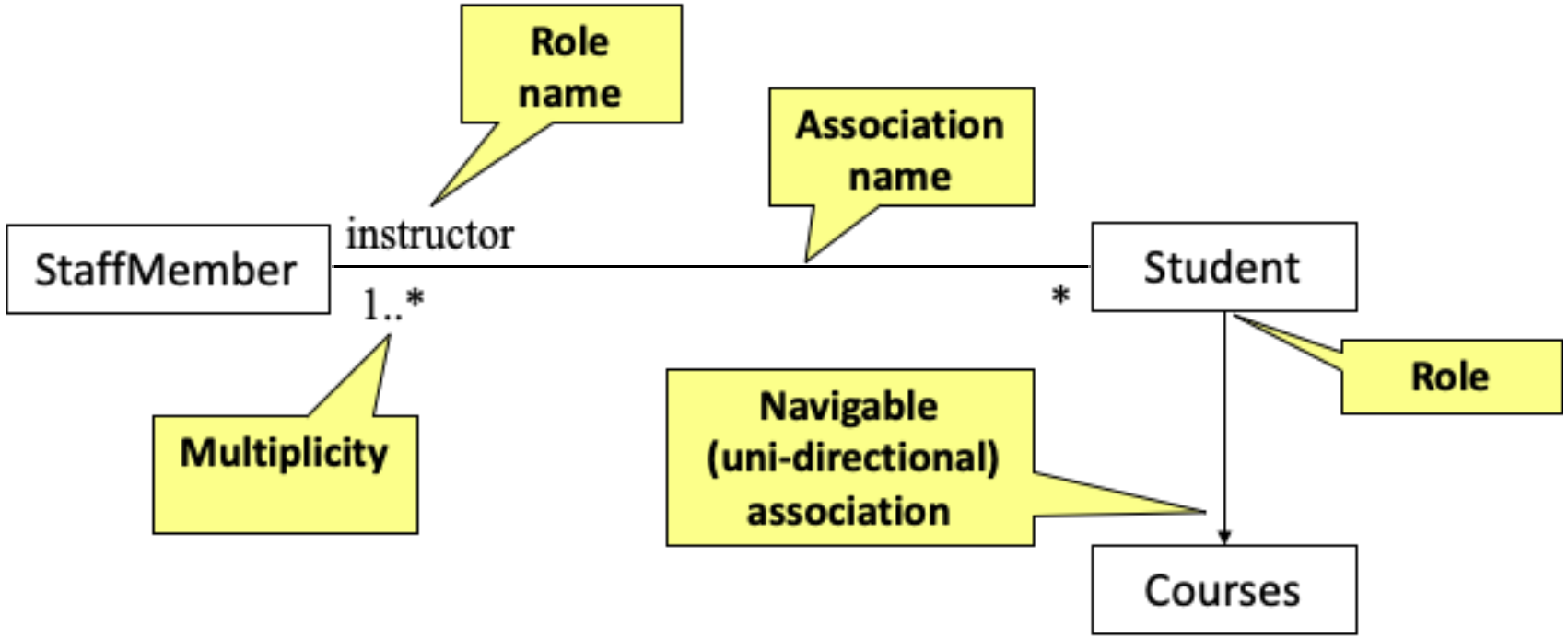
Relationship from one class back to itself



Reflexive relationship. A manager is an employee who manages other workers. Notice how the labels explain the purpose of the relationship.

Analysis Model

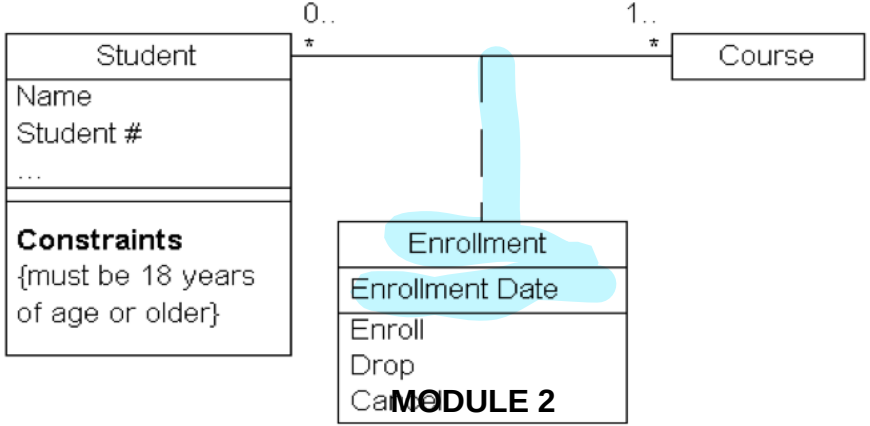
Class Diagram



Analysis Model

Class: Association Class

- **include another class to add valuable information** about the relationship.
- use an ***association class*** **to tie the primary association.**
- An association class is represented like a normal class.
- The difference is that the **association line** between the primary classes intersects a **dotted line connected** to the association class.



Analysis Model

Class Diagram

Multiplicity:

- the number of objects that participate in the association.
- **Multiplicity Indicators**

Exactly one	1
Zero or more (unlimited)	* (0..*)
One or more	1..*
Zero or one (optional association)	0..1
Specified range	2..4
Multiple, disjoint ranges	2, 4..6, 8

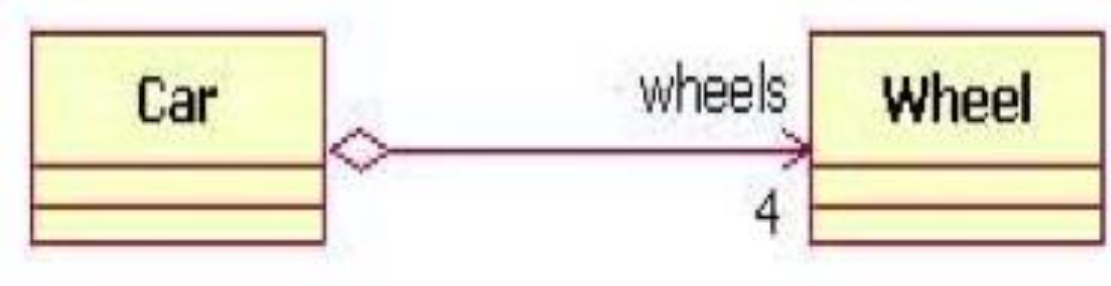
Analysis Model

Class Diagram

Aggregation

•Aggregation is a special type of association used to model a **"whole to its parts"** relationship.

•Example: Car as a whole entity and Car Wheel as part of the overall Car.

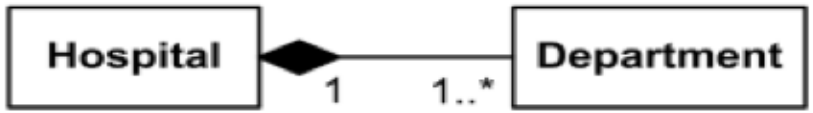


Analysis Model

Class Diagram

Composite Aggregation

- A strong form of aggregation
  - The whole is the sole owner of its part.
- The part object may belong to only one whole
  - Multiplicity on the whole side must be zero or one.
  - The life time of the part is dependent upon the whole.



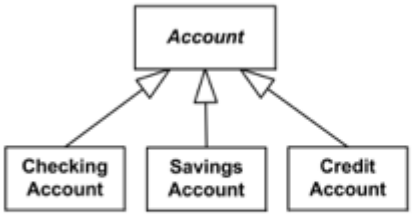
Hospital has 1 or more Departments, and each Department belongs to exactly one Hospital. If Hospital is closed, so are all of its Departments.

Analysis Model

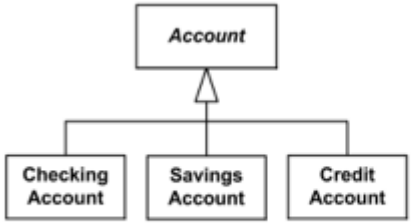
Class Diagram

Generalisation:

•The Generalization association ("**is a**") is the **relationship** between the **base class** that is named as "superclass" or "**parent**" and the specific class that is named as "**subclass**" or "**child**".



*Checking, Savings, and Credit Accounts are **generalized** by Account*



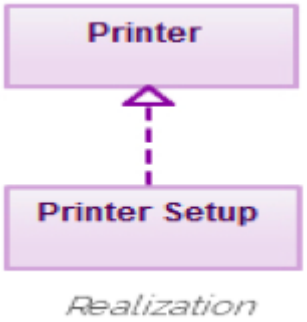
*Checking, Savings, and Credit Accounts are **generalized** by Account*

Analysis Model

Class Diagram

Realisation:

- A realization relationship indicates that **one class implements a behavior specified by another class** (an interface or protocol).
- a broken line with an unfilled solid arrowhead is drawn from the class that defines the functionality to the class that implements the function.

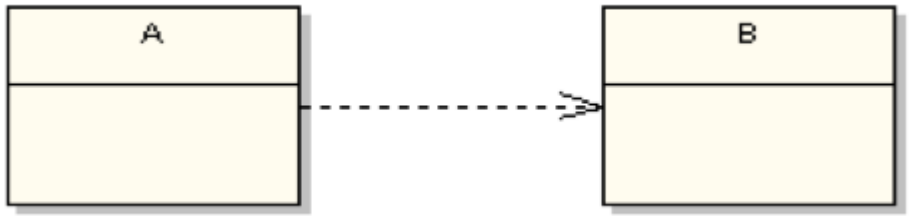


Analysis Model

Class Diagram

Dependency:

- It means that the class at the source end of the relationship has some sort of dependency on the class at the target (arrowhead) end of the relationship.



Analysis Model

Class Diagram

Dependency:

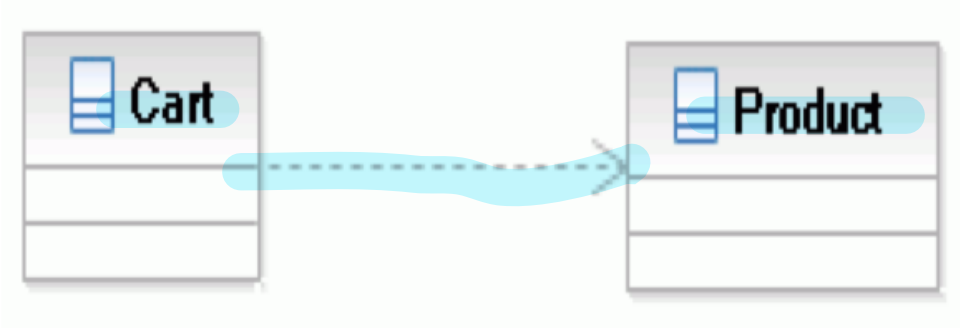
- In UML, a dependency relationship is a relationship in which one element, the client, uses or depends on another element, the supplier.
- Dependency shows that an element is dependent on another element as it fulfils its responsibilities within the system

Analysis Model

Class Diagram

Dependency:

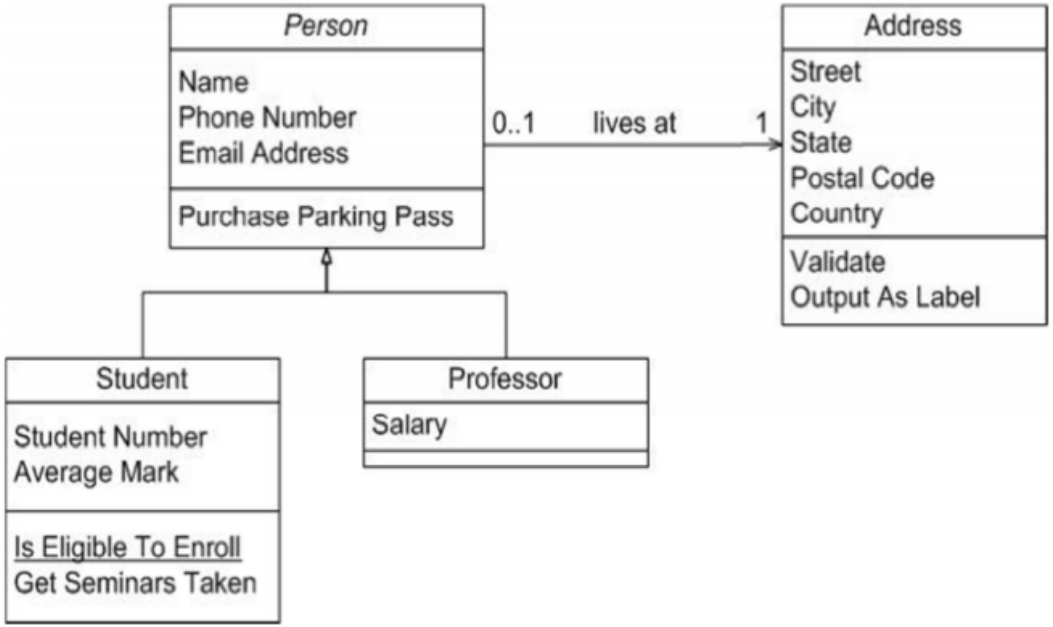
- A **Cart class depends on a Product class** because the Cart class uses the Product class as a parameter for an add operation.
- Cart class is, therefore, the client, and the Product class is the supplier.



Analysis Model

Class Diagram

UML example: people



Analysis Model

Collaboration Diagram:

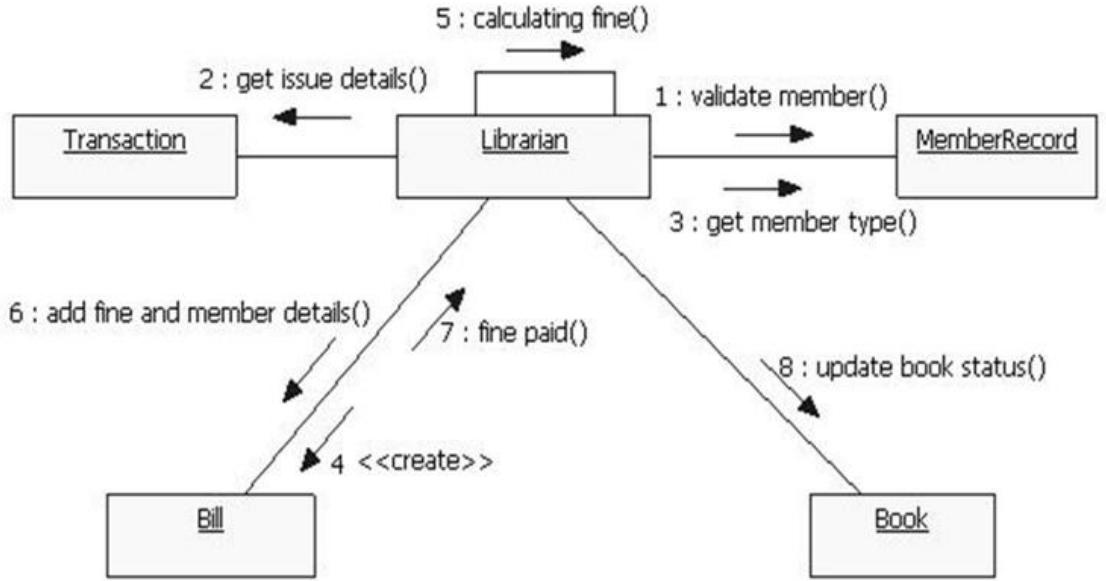
- **Emphasis on structural organization of the objects that send and receive messages**
- **Elements of Collaboration diagram:**
  - **Objects:**

  - **Relations/ associations**

  - **Messages** function()

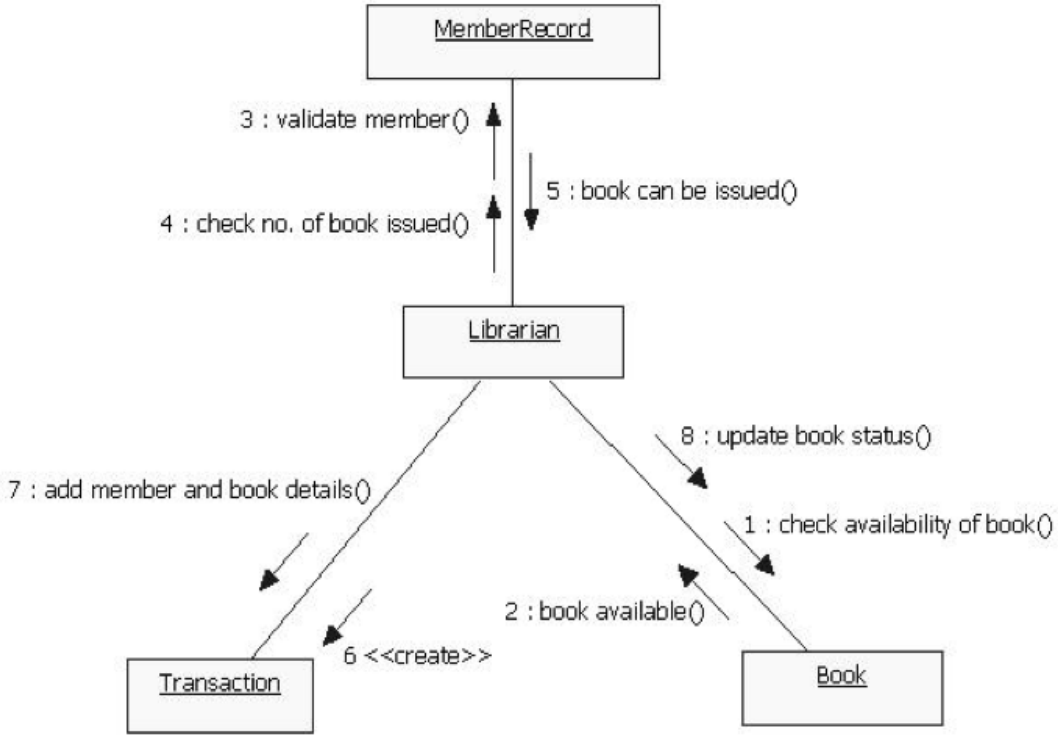
Analysis Model

Collaboration Diagram:



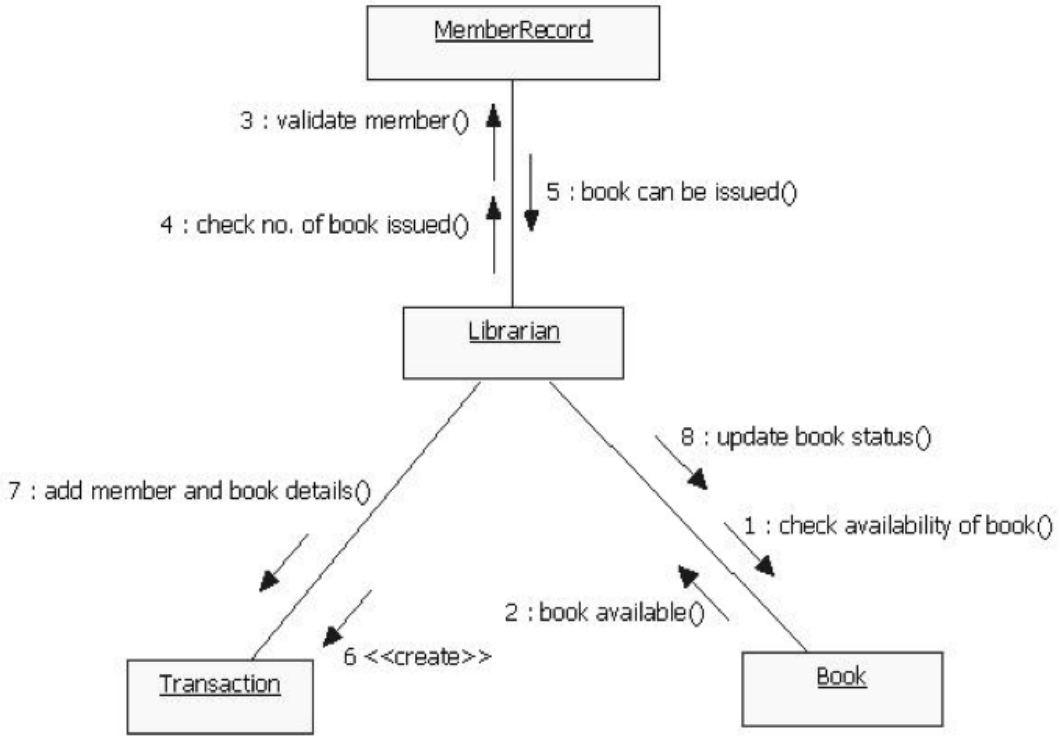
Analysis Model

Collaboration Diagram:

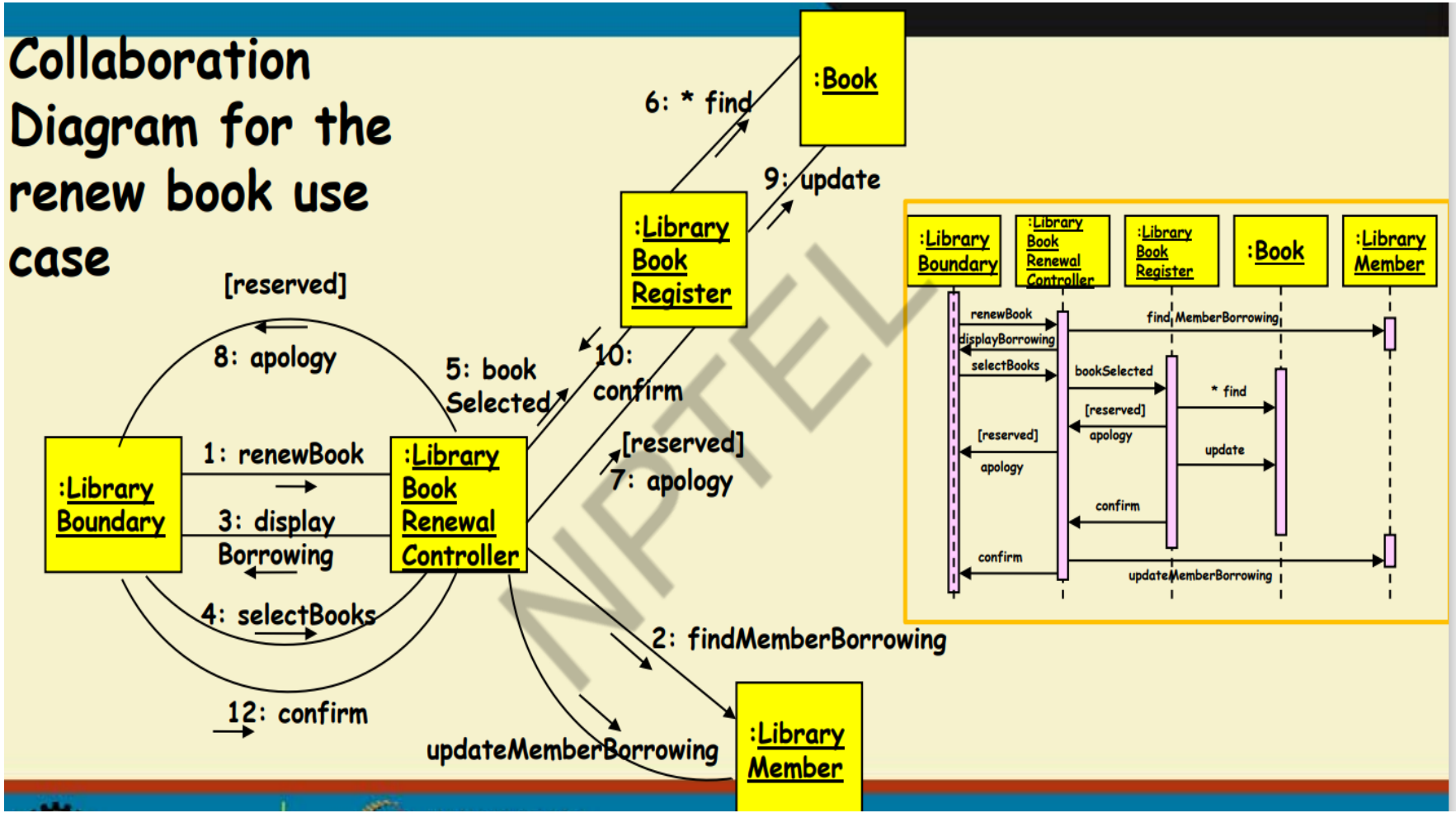


Analysis Model

Collaboration Diagram:



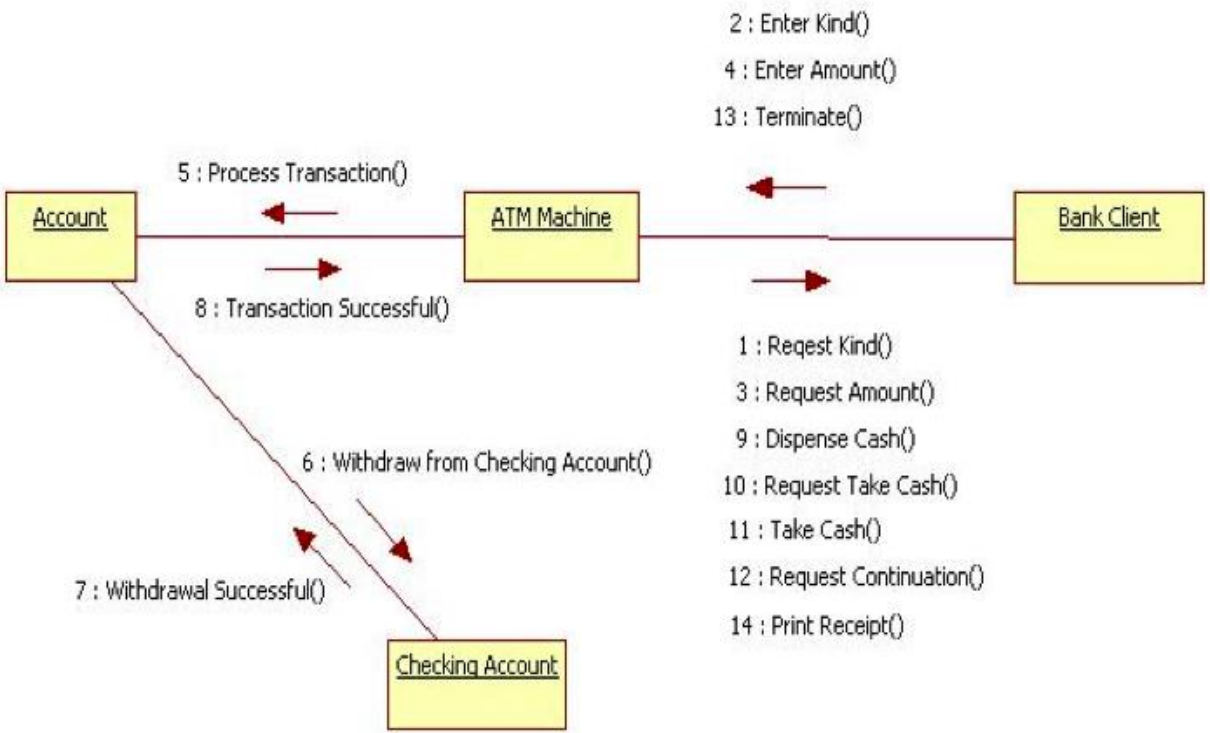
Collaboration Diagram for the renew book use case



Ref: NPTEL course on software engineering, Ranbir mall

Analysis Model

Collaboration Diagram:



Analysis Model

State Chart Diagram:

- A **state** in UML is a condition or **situation** an object (in a system) might find itself in during its life time.
- Normally used to **model how the state of an object changes** in its lifetime.
- Used to **capture** the behavior of the subject object through modeling these various states and transitions between them.
- Does not necessarily model all possible states, but rather the **critical** ones only, those that act as **stimuli** and **prompt** for **response** in the external world.

Analysis Model

State Chart Diagram:

Elements of state chart diagram

- **State:** Represented by rectangles with rounded corners.
- **Initial state:** Represented as a filled circle.
- **Final state:** Represented by a filled circle inside a larger circle.
- **Transition:** Shown as an arrow between two states. Normally, the name of the event which causes the transition is places alongside the arrow.
- **A guard condition:** Condition that has to be met in order to enable the transition to which it belongs:
  - Guard conditions can be used to document that a certain event, depending on the condition, can lead to different transitions.

Analysis Model

State Chart Diagram:

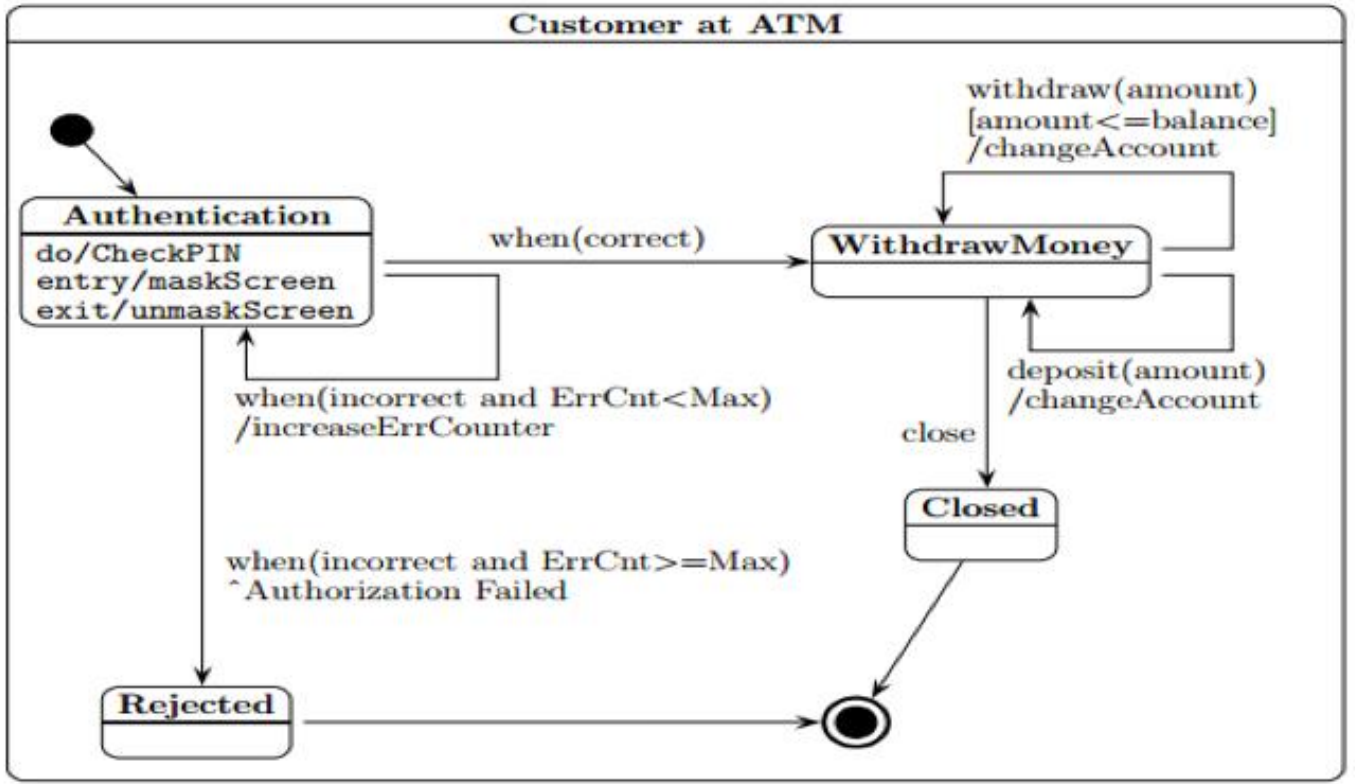
Elements of state chart diagram

- **State:** 

- **Initial state:** 
- **Final state:** 
- **Transition:** 
- **A guard condition:** Event[Guard condition/action]

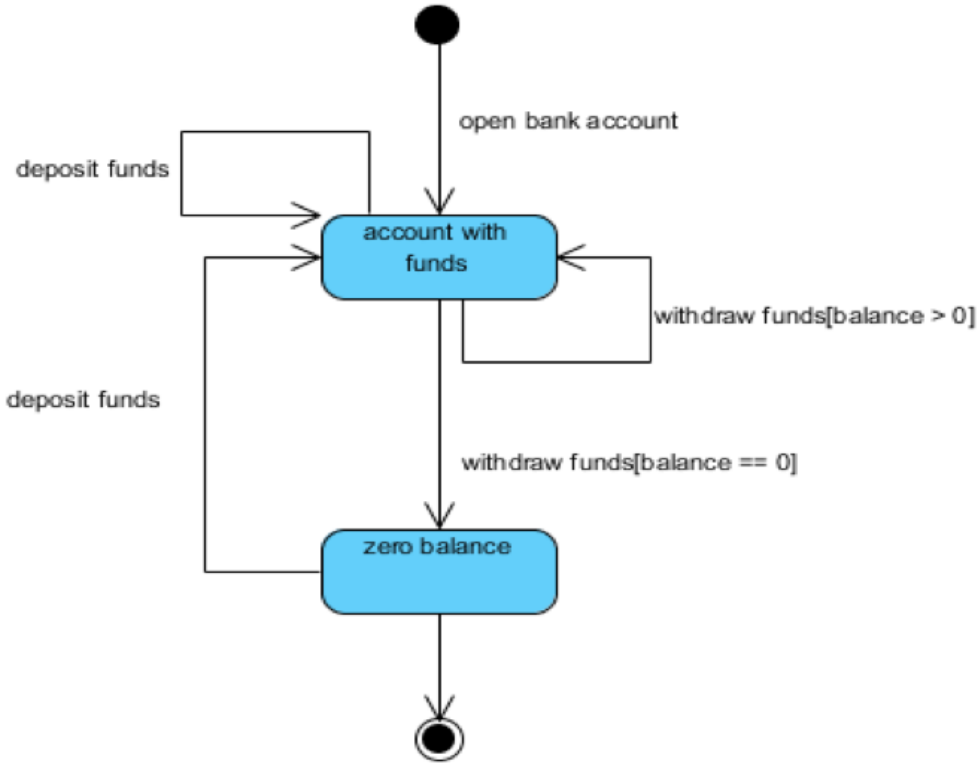
Analysis Model

State Chart Diagram:



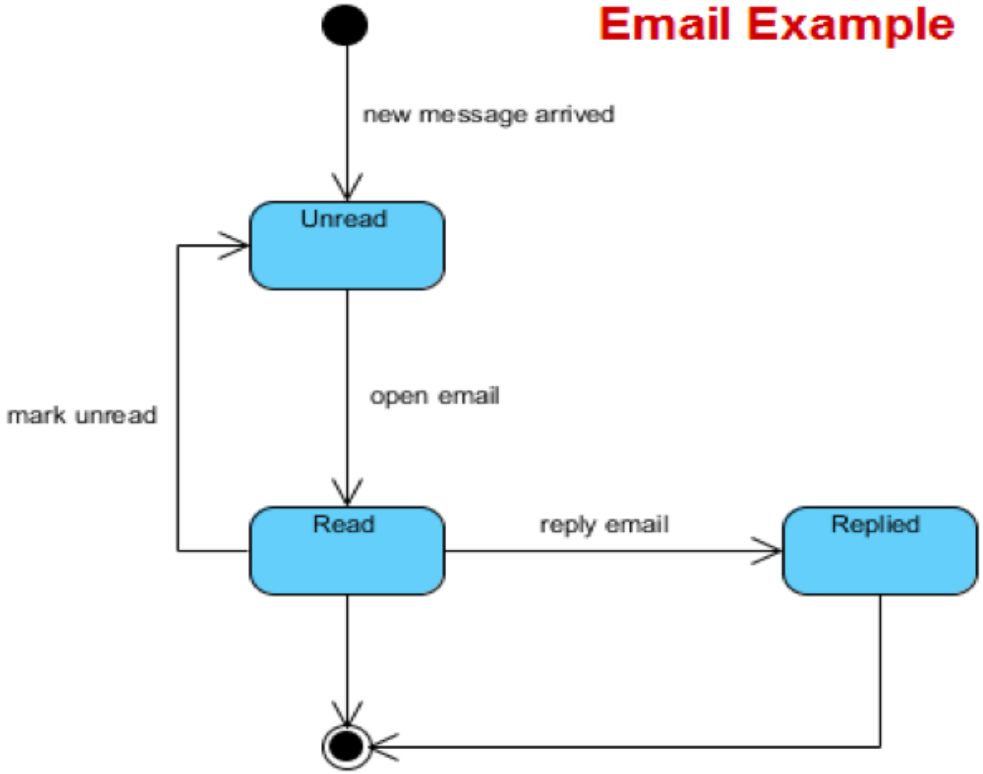
Analysis Model

State Chart Diagram:

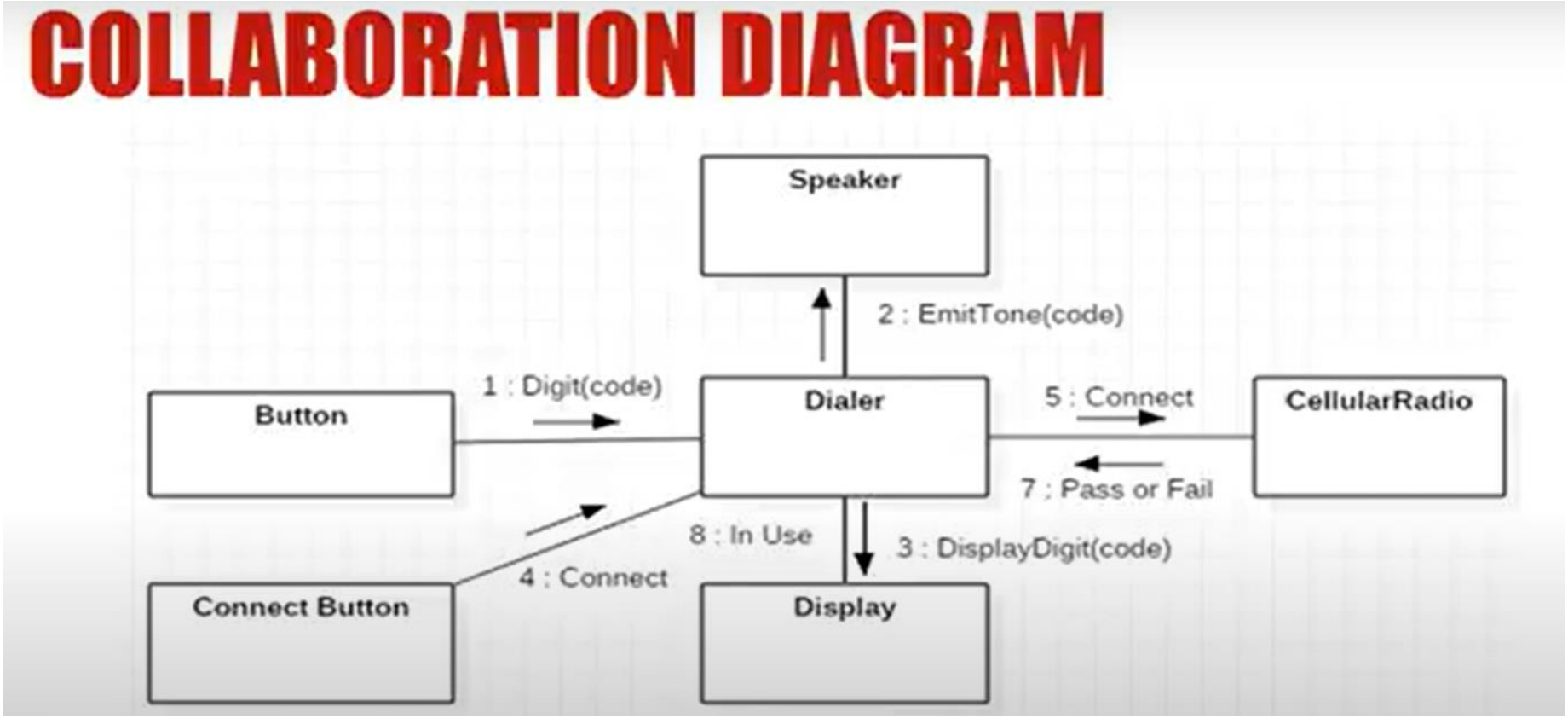


Analysis Model

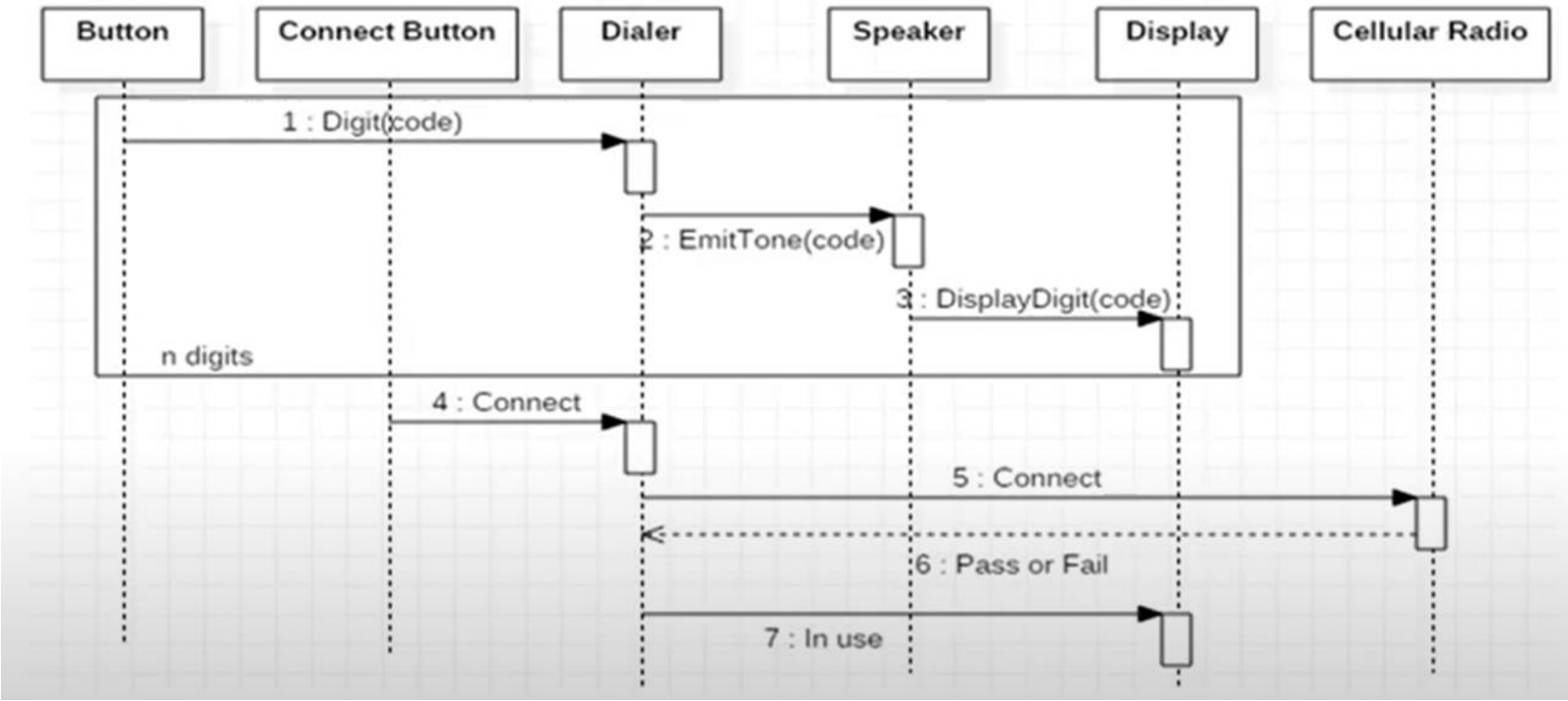
State Chart Diagram:



Collaboration diagram



Sequence diagram



Domain analysis

- Three types of classes are identified

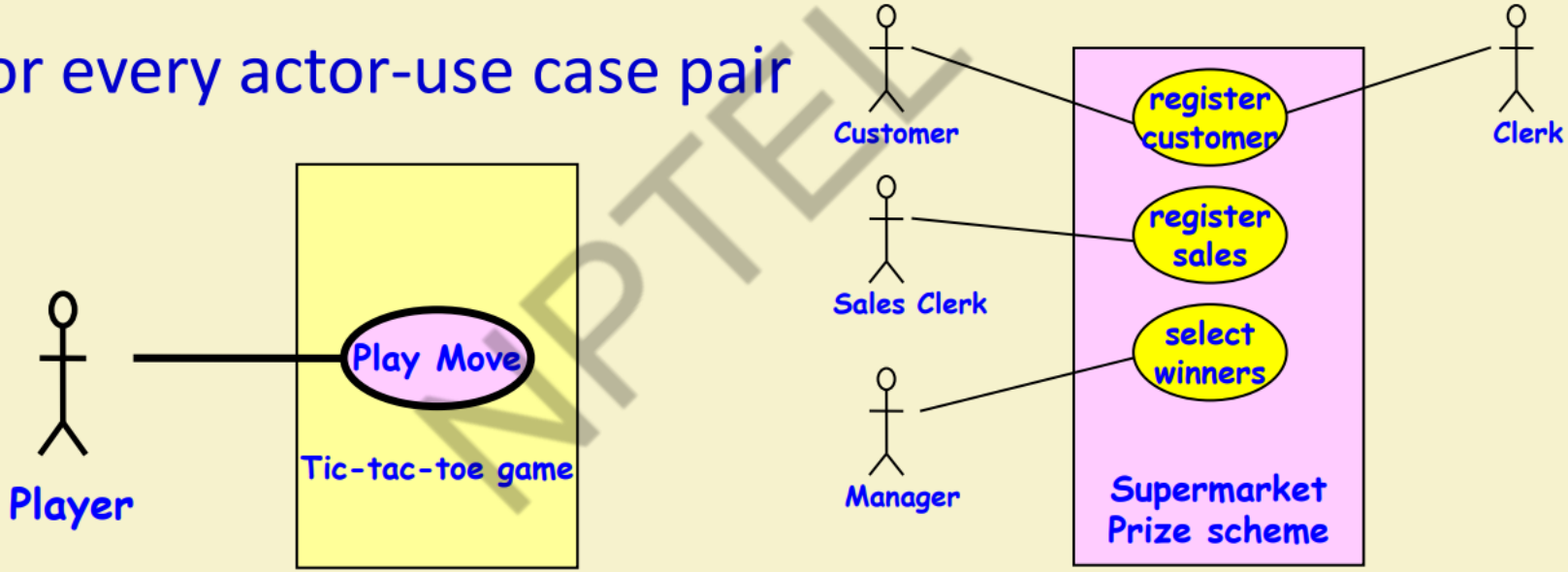
-Boundary class (Actor-use case pair)

-Controller class (One per use case)

-Entity class (Noun analysis)

Identification of Boundary Objects

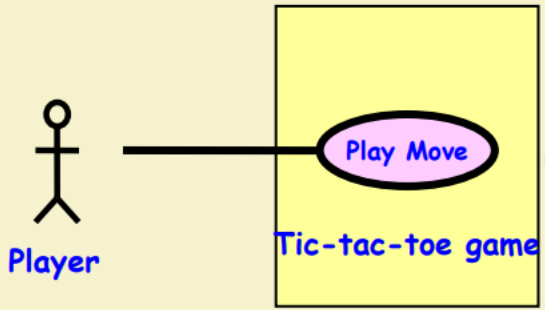
- Need one boundary object :
  - For every actor-use case pair

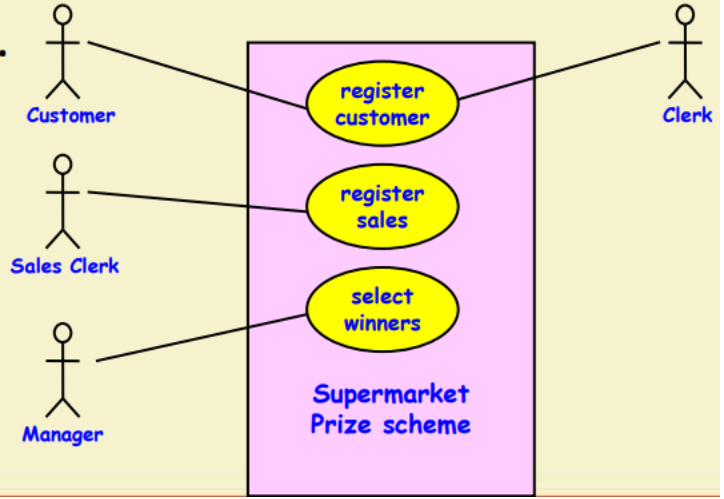


Ref: NPTEL course on software engineering, Ranbir mall

Identification of Controller Objects

- Examine the use case diagram:
  - **Add one controller class for each use case.**
  - Some controllers may need to be split into two or more controller classes if they get assigned too much responsibility.





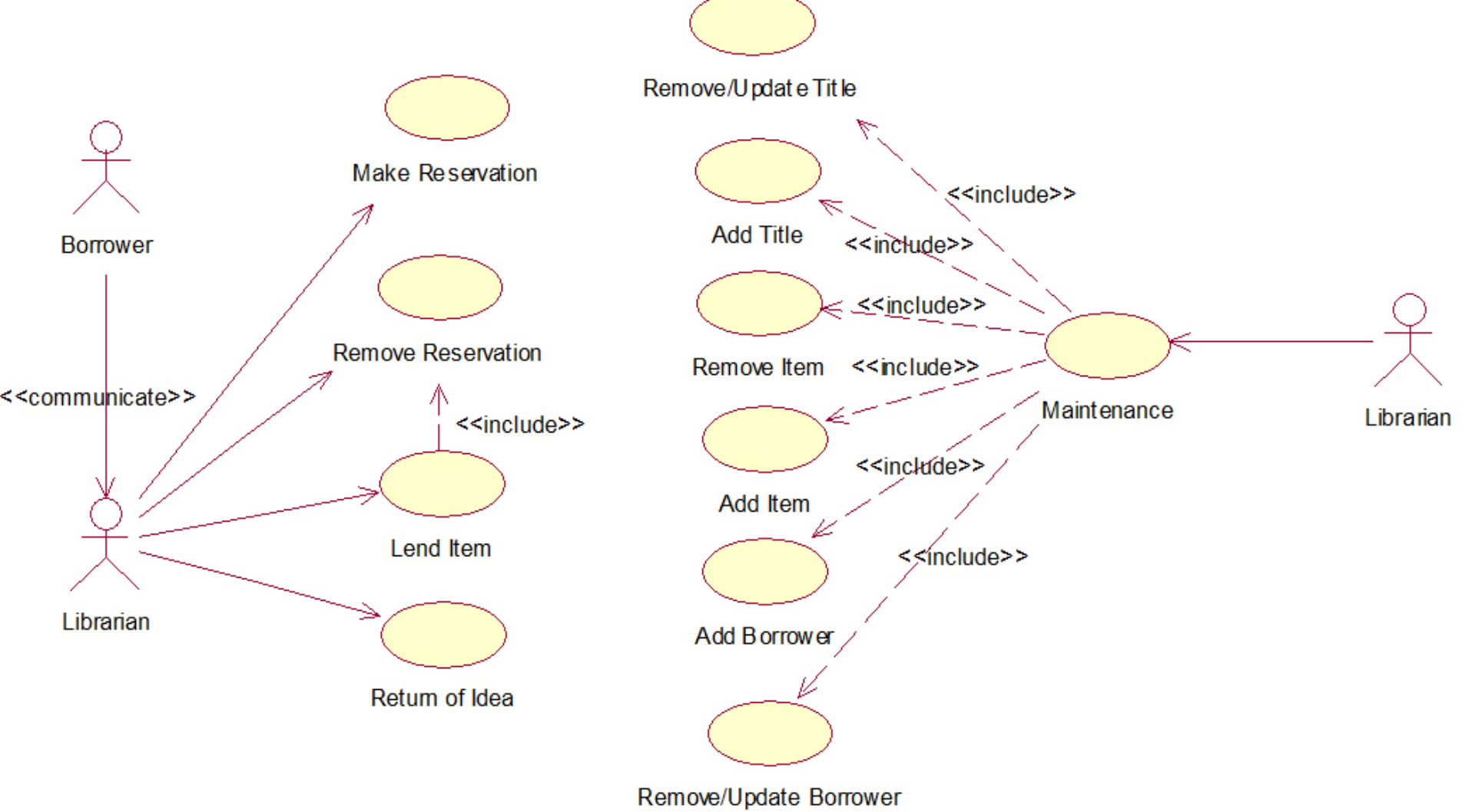
Identification of Entity objects by Noun analysis

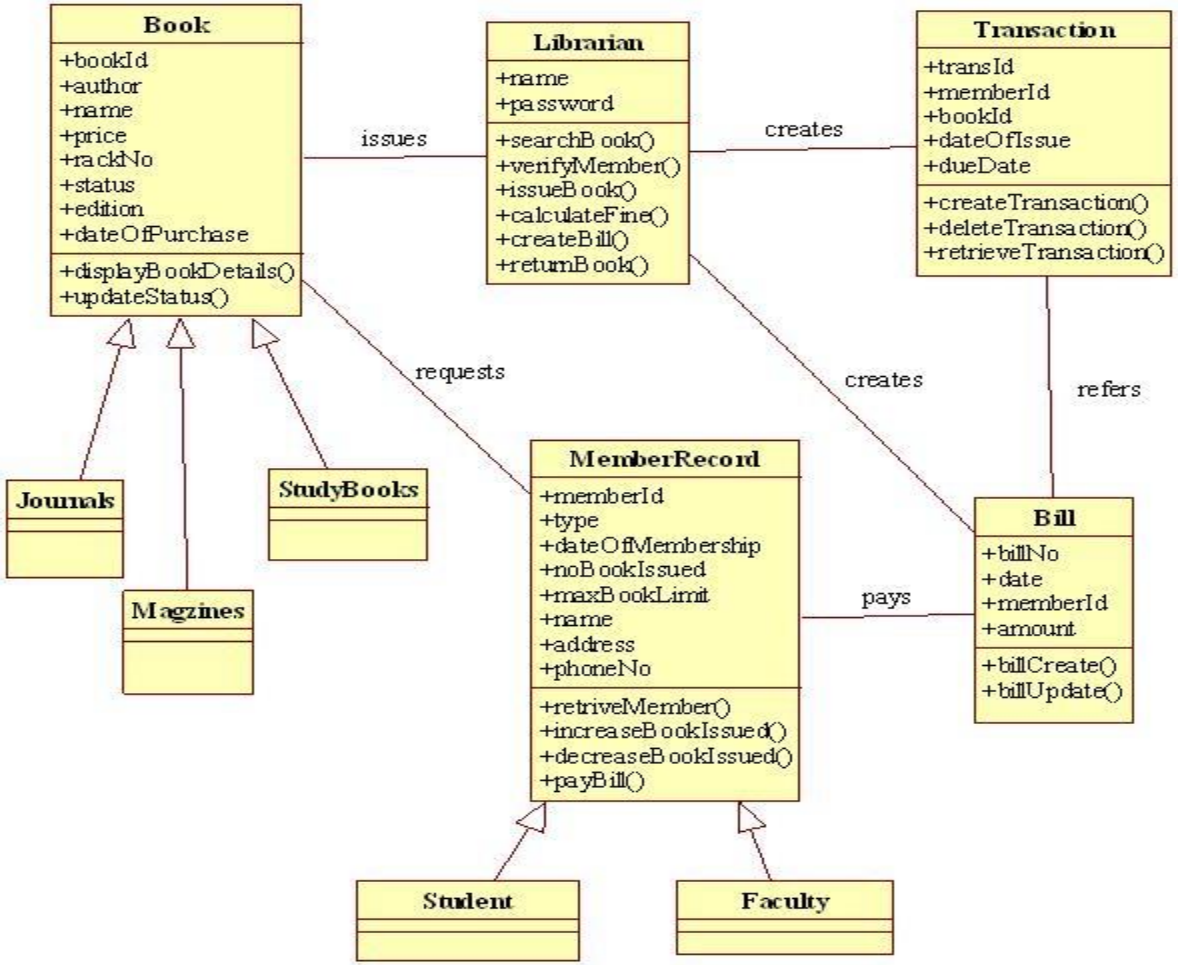
- Entity objects usually appear as nouns in the problem description. •
- From the list of nouns, need to exclude

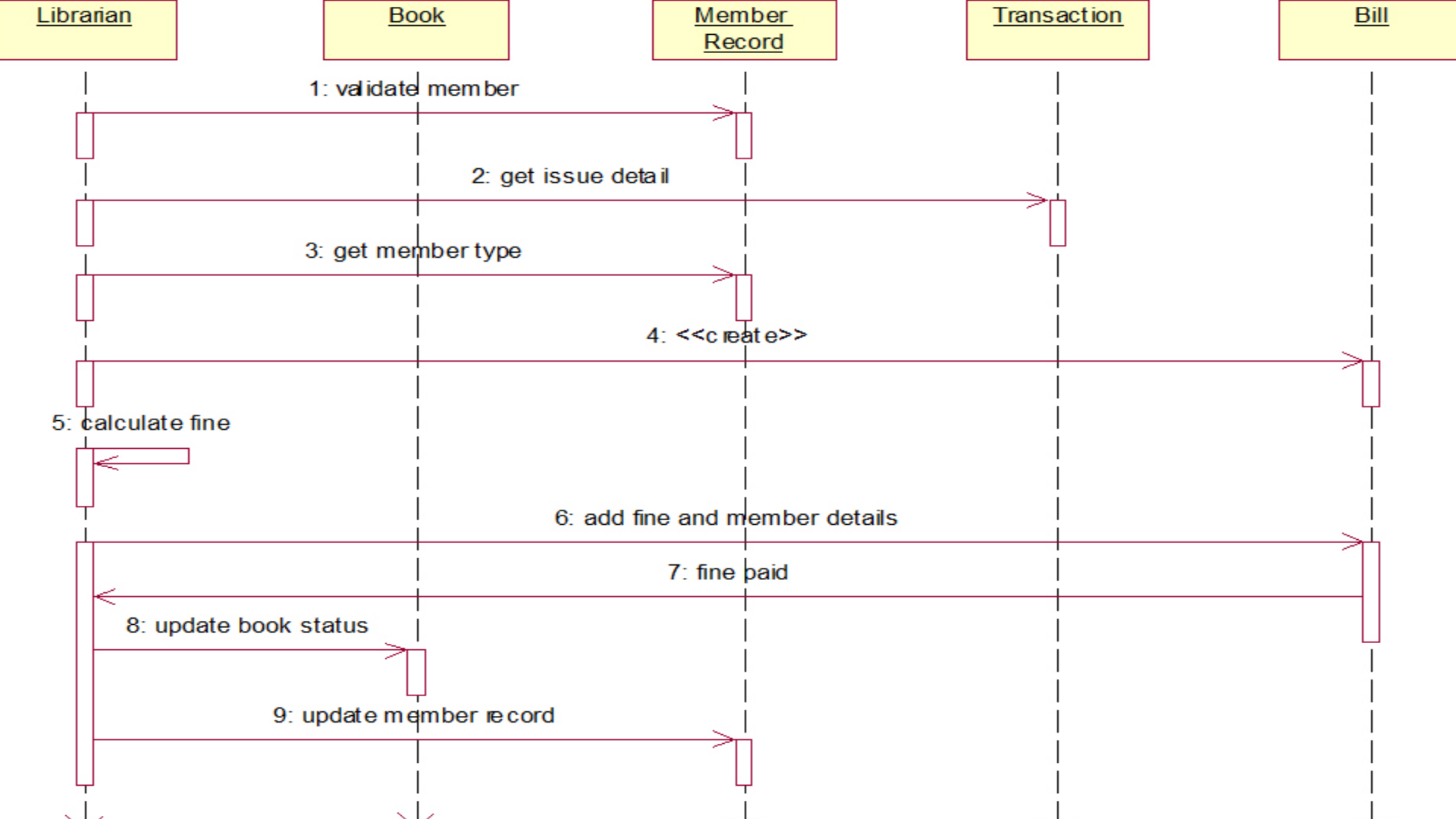
- Users (e.g. accountant, librarian, etc)
- Passive verbs (e.g. Acknowledgment) –
- Those with which you can not associate any data to store –
- Those with which you can not associate any methods

• Surrogate users may need to exist as classes:

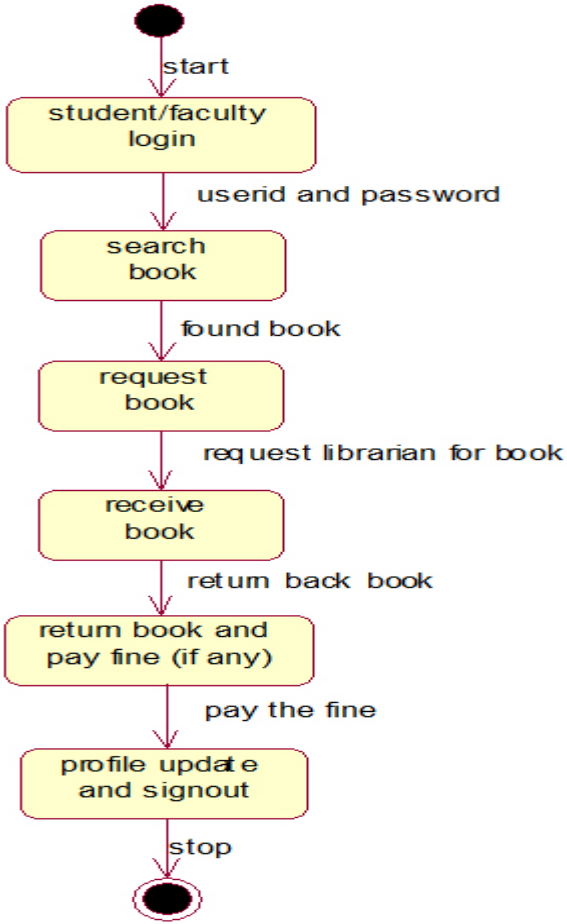
– Library member


Make Reservation
Borrower
Remove Reservation
<<communicate>>
<<include>>
Lend Item
Librarian
Return of Idea
Remove/Update Title
Add Title
<<include>>
<<include>>
Remove Item
<<include>>
Maintenance
Librarian
Add Item
<<include>>
Add Borrower
<<include>>
Remove/Update Borrower


Book
+bookId
+author
+name
+price
+rackNo
+status
+edition
+dateOfPurchase
+displayBookDetails()
+updateStatus()
Librarian
+name
+password
+searchBook()
+verifyMember()
+issueBook()
+calculateFine()
+createBill()
+returnBook()
Transaction
+transId
+memberId
+bookId
+dateOfIssue
+dueDate
+createTransaction()
+deleteTransaction()
+retrieveTransaction()
issues
creates
requests
creates
refers
Journals
Magzines
StudyBooks
MemberRecord
+memberId
+type
+dateOfMembership
+noBookIssued
+maxBookLimit
+name
+address
+phoneNo
+retriveMember()
+increaseBookIssued()
+decreaseBookIssued()
+payBill()
Bill
+billNo
+date
+memberId
+amount
+billCreate()
+billUpdate()
pays
Student
Faculty


Librarian
Book
Member Record
Transaction
Bill
1: validate member
2: get issue detail
3: get member type
4: <<create>>
5: calculate fine
6: add fine and member details
7: fine paid
8: update book status
9: update member record

State chart diagram



Activity diagram