Software Engineering

CSIT- 6th Semester

Prepared by: Er. Rupak Gyawali, 2024

Unit 5: System Modeling

Introduction to System Modeling; Context Models; Interaction Models; Structural Models; Behavioral Models; Model-Driven Architecture

Assignment:
Difference Between Structural and Behavioral Model – 5 marks
Difference Between Structural and Interaction Model – 5 marks
Difference Between Behavioral and Interaction Model – 5 marks
Difference Between Class Diagram vs Sequence Diagram -5 marks

Introduction to System Modeling:

System modeling is a process of creating simple diagrams or drawings to understand and explain how a system works. These models are often made using diagrams and symbols, usually based on UML (Unified Modeling Language). Sometimes, mathematical models are also created for detailed system specifications.

Why Do We Use Models?

- **To understand the current system:** See what works well and what needs improvement.
- **To plan the new system:** Show how the new system should work and guide its design and building.

Two Main Types of Models:

- **Existing system models:** Help us understand and fix problems in the current system.
- **New system models:** Help explain and design the new system.

- The main purpose of a system model is to focus on the essential parts by leaving out unnecessary details.
- A model is a simplified version of a system, highlighting key features rather than providing a full, detailed copy.
- For example, if this book were summarized in a newspaper, only the key points would be included—that's an abstraction. But if the book were translated into another language, it would aim to keep all the original information.

Different Ways to Model a System:

1. **External perspective**: Shows how the system interacts with its environment (Users, Other Systems).
2. **Interaction perspective**: Focuses on interactions between the system and its environment, or between parts within the system.
3. **Structural perspective**: Shows how the system is organized or how its data is structured.
4. **Behavioral perspective**: Models how the system behaves and responds to different events.

Introduction to System Modeling:

Unified Modeling Language: UML (Unified Modeling Language) is a standard way to create diagrams that model systems, especially for object-oriented design. There are many types of UML diagrams, but most users find these five types the most essential:

1. **Activity Diagrams**: Show the steps or activities in a process or how data is processed.
2. **Use Case Diagrams**: Show how the system interacts with its environment, including users and other systems.
3. **Sequence Diagrams**: Show the sequence of interactions between users (actors) and the system, or between different parts of the system.
4. **Class Diagrams**: Show the system's object classes and how they are connected.
5. **State Diagrams**: Show how the system responds to different events, both internal and external.

Context Models:

- **Context models** help define the boundaries of a system early in its development. This means, At the start of planning a system, decide what the system will do and what will be handled by other systems or processes.
- Work with stakeholders to choose which tasks to automate and which to keep manual.
- Check if similar tasks are already handled by other systems to avoid duplication. Making these decisions early helps save time and reduce costs.

Example of a Context Model:

Imagine a **library management system** being developed.

  - **System boundary:** The system will handle tasks like issuing books, returning books, and keeping track of inventory.
  - **Environment:** Tasks like maintaining the library building or organizing events will stay outside the system's scope.
- Sometimes, it's easy to define what the system will handle and what it won't. For example, if you're replacing a manual system with an automated one, the surroundings (environment) usually stay the same.
  - If we are replacing the manual library system with the automated one, the environment (user → Librarian, Students), (task performed → Borrowing and Returning), (Library Rules → Borrowing limit, fines) donot change.
- Other times, you have more choice, and you decide what the system will do and what will be left out while gathering requirements.

Context Model:

Imagine you are designing a system for mental health clinics to manage patient information and treatments. You need to decide:

1. Should the system only store details about consultations (and get personal patient info from another system)?
2. Or should it also store personal patient information?

If you rely on another system for personal data, you avoid having duplicate data. However:

- It might take longer to get information from the other system.
- If the other system isn't working, your system can't function properly.

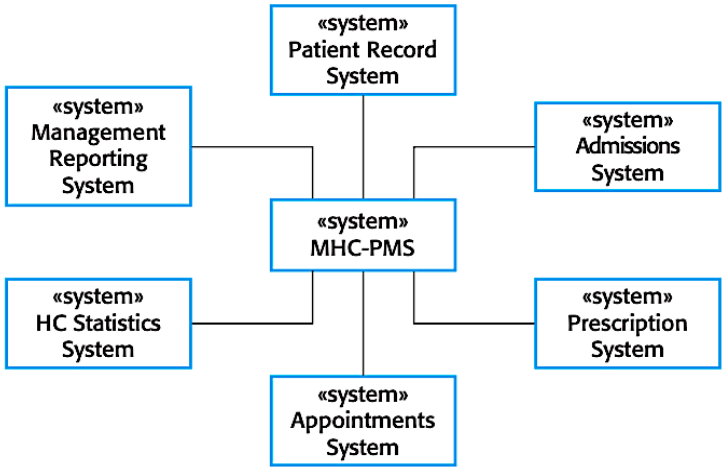


Figure 5.1 The context of the MHC-PMS

We're building the MHC-PMS (Mental Health Care Patient Management System), it interacts with other systems around it.

Figure 5.1 shows a simple diagram of how the patient information system interacts with other systems.

- The patient system connects to:
    - An appointment system for scheduling.
    - A general patient record system for sharing data.
    - Systems for management reports and hospital bed allocation.
    - A statistics system for research data.
    - A prescription system for generating medication prescriptions.
- Context models show the system you're building and the other systems around it, but don't explain how they connect or interact. These outside systems might give data to your system, take data from it, or share data with it. They might be connected by a network or not, and they may be in the same building or separate locations. These factors can impact how the system is designed and what it needs to do.
- To fully understand the system's interactions, context models are used with other models, like business process models, which show how both humans and other systems use the software.

Interaction Models:

- All systems involve some form of interaction, whether it's with users, other systems, or within the components of system itself. Understanding these interactions is important for developing a good system.
- Here are two approaches for modeling these interactions:

1. **Use case modeling**: This helps show how the system interacts with users or other systems. It highlights what users need from the system.
2. **Sequence diagrams**: These show how different parts of the system (or external systems) communicate with each other. They help us understand if the system will work as expected in terms of performance and reliability.

- Both use case models and sequence diagrams show interactions at different levels, and they can be used together.
- A use case may give an overview of an interaction, while a sequence diagram provides more detailed information about the specific steps involved.
- There are also communication diagrams, which are similar to sequence diagrams, but they show the interaction in a different format.

Interaction Models:

1. **Use case Modeling/ Diagram**
    - Capture the dynamic behavior
    - Dynamic behavior means the behavior of the system when it is running/operating
    - Viewing System through user's perspective.
    - Shows the relationship between actors and use cases.
    - Captures the functionality of the system

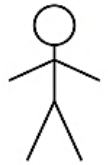
Purposes of use-case diagram:

- Gather requirements of system.
- Get outside view of system.
- Identify internal and external factors affecting the system.

Interaction Models:

1. Use Case Modeling/ Diagram:

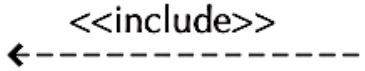
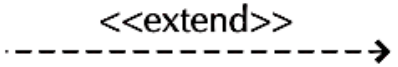
Elements/ Syntax of Use case diagram

An actor: ■ Is a person or system that derives benefit from and is external to the subject. ■ Is depicted as either a stick figure (default) or, if a nonhuman actor is involved, as a rectangle with <<actor>> in it (alternative). ■ Is labeled with its role. ■ Can be associated with other actors using a specialization/superclass association, denoted by an arrow with a hollow arrowhead. ■ Is placed outside the subject boundary.	 **Actor/Role** **<<actor>>** **Actor/Role**
A use case: ■ Represents a major piece of system functionality. ■ Can extend another use case. ■ Can include another use case. ■ Is placed inside the system boundary. ■ Is labeled with a descriptive verb–noun phrase.	**Use Case**
A subject boundary: ■ Includes the name of the subject inside or on top. ■ Represents the scope of the subject, e.g., a system or an individual business process.	**Subject**

Interaction Models:

1. Use Case Modeling/ Diagram:

Elements/ Syntax of Use case diagram

An association relationship: ■ Links an actor with the use case(s) with which it interacts.	
An include relationship: ■ Represents the inclusion of the functionality of one use case within another. ■ Has an arrow drawn from the base use case to the used use case.	
An extend relationship: ■ Represents the extension of the use case to include optional behavior. ■ Has an arrow drawn from the extension use case to the base use case.	
A generalization relationship: ■ Represents a specialized use case to a more generalized one. ■ Has an arrow drawn from the specialized use case to the base use case.	

Interaction Models:

1. Use Case Modeling/ Diagram:

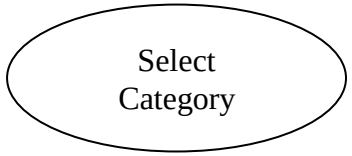
<<includes>> and <extends>> relationship in detail

<<includes>> : 

use case includes 

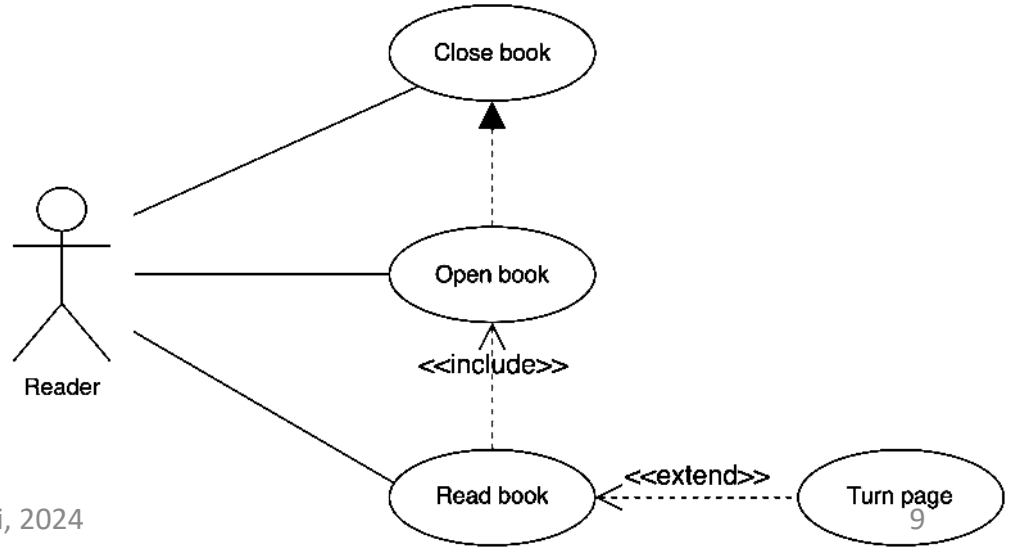
. What this mean is in order to *place an order*, *item needs to be added to the cart*. It is a mandatory case.

<<extends>> : 

extends 

. What this mean is we can still search a book without selecting the category. It is an optional case.

- "Read book" use case includes "Open book" use case. This means a reader must open book to read a book. It is mandatory.
- "Read book" use case extends "Turn page" use case. This means that while reading a book, the reader may turn pages, but it's not strictly necessary for the use case "Read Book" to be completed.



Interaction Models:

1. Use Case Modeling/ Diagram:

Rules for Naming Use Cases:

- Use a verb-noun format to describe what the use case does. This makes it clear what action is being performed and on what entity. Examples: "Place Order”, "Register User”, "Generate Report”
- Avoid vague or overly broad names. Examples: Instead of "Manage Account," use "Update Account Information" or "Deactivate Account”
- Name the use case based on the goal of the user, not the system’s action. This keeps the use case user-centric. Examples: "Submit Application" (user’s goal) instead of "Receive Application" (system’s action)
- Use language that is easily understandable by all stakeholders, including non-technical ones. Examples: Use "Upload Document" instead of "Initiate File Transfer Protocol”
- Use short and concise names. Avoid overly long names that can make the diagram cluttered and difficult to read. Examples: "Login" instead of "User Logs Into the System”
- Use present tense consistently for naming use cases. This aligns with the common practice of describing system interactions. Examples: "Print Receipt" instead of "Printing Receipt" or "Printed Receipt”

Interaction Models: 1. Use Case Modeling/ Diagram:

Identifying Major Use Cases:

a) **Review Requirements Definition: 2 types**

- **Functional Requirements:**
  - The system must generate reports based on user-specified criteria.
  - The system must support multiple payment methods, including credit card and PayPal.
- **Non-functional requirements:**
  - The system must be available 99.5% of the time.
  - The system must be able to handle 1000 concurrent users.

b) **Identify Subject's Boundaries:** This helps the analyst to identify the scope of the system. Eg: Decide to integrate Master card in payment system or just QR payment.

c) **Identify Primary Actors:** The primary actors involved with the system comes from a list of stakeholders and users.

d) **Identify Business Processes & Major Use Cases:** Rather than jumping into one use case and describing it completely at this point, we only want to identify the use cases.

e) **Review Current Set of Use Cases:** Carefully review the current set of use cases. It may be necessary to split some of them into multiple use cases or merge some of them into a single use case

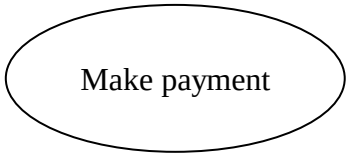
Interaction Models:

1. Use Case Diagram/ Modeling:

<u>Steps to create a Use Case Diagram</u>

a) Identify Actors
   - Actor: The person who uses the system
   - Types of Actors:
     - **Primary Actor:** Drives the use case. Uses the system to get some services. Eg: User
     - **Supporting Actor:** Provides a service to the system. Eg: Bank → Bank provides payment authorization service.
     - **Offset Actor:** Has interest in the use case but is not a primary or supporting actor. Eg: Government Tax Agency.

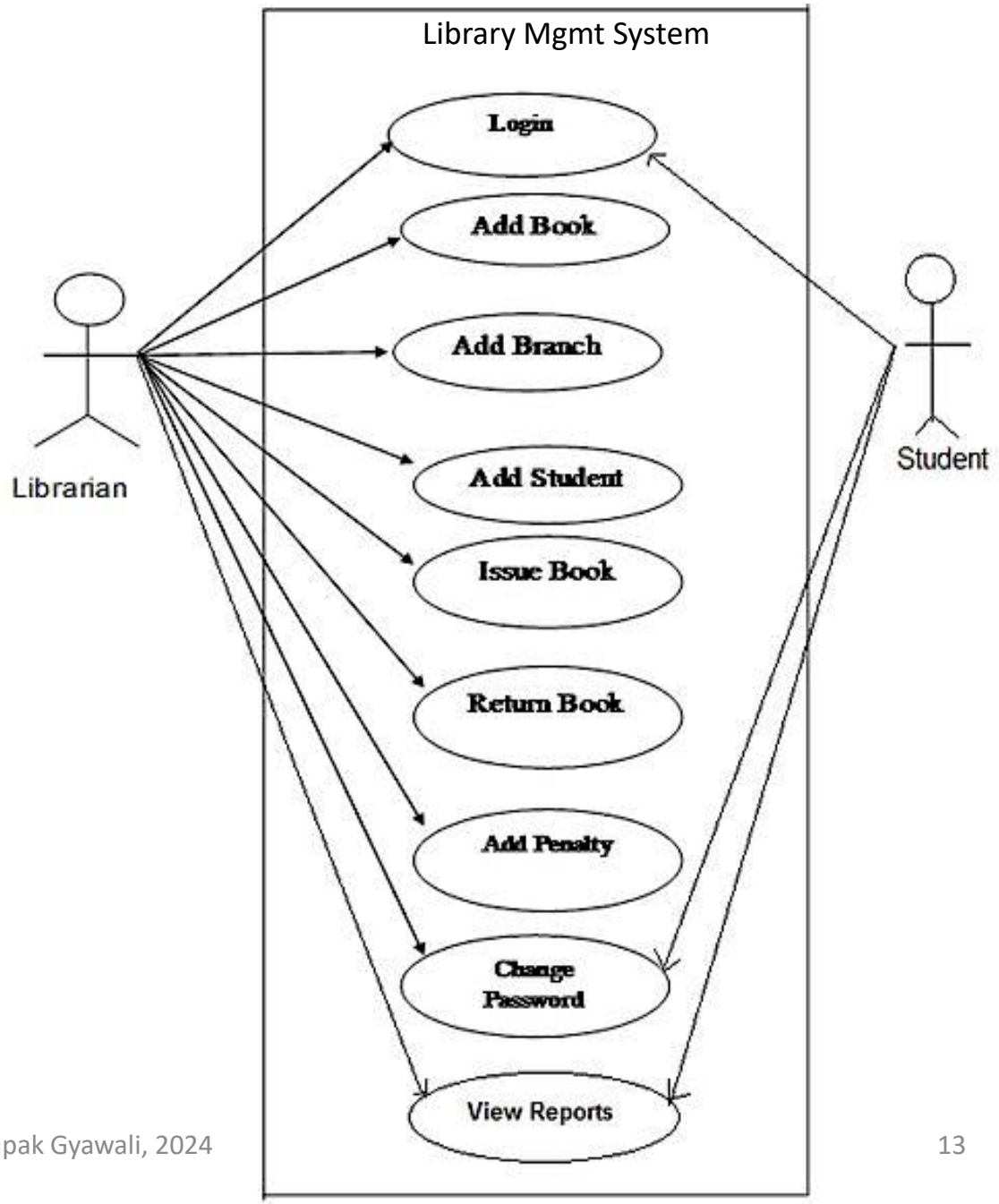
b) Identify Use cases
   - Each actor interacts with the use case that the system has.
   - Use case is the functionality provided by the system.



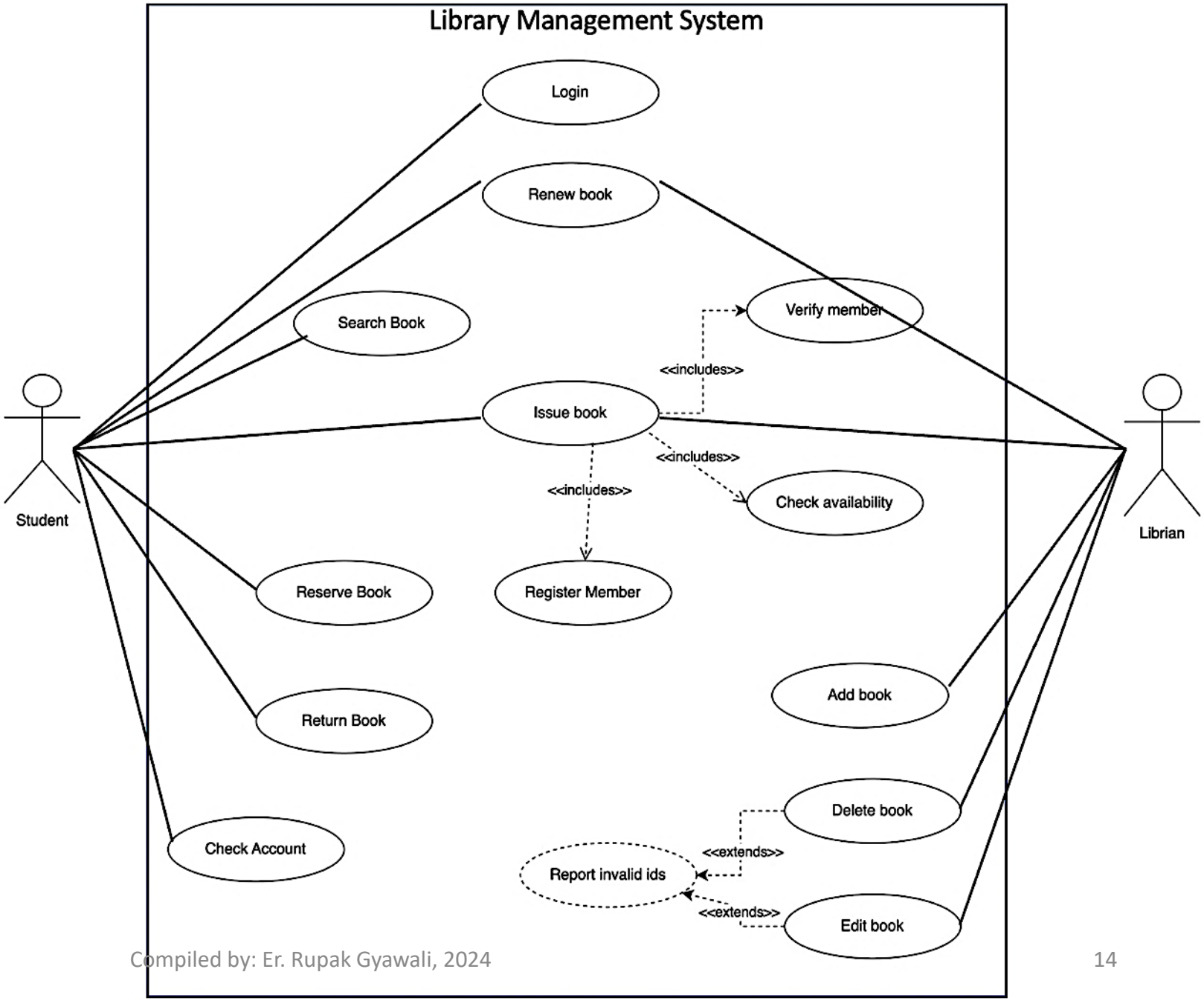
c) Draw subject boundary

d) Add Association

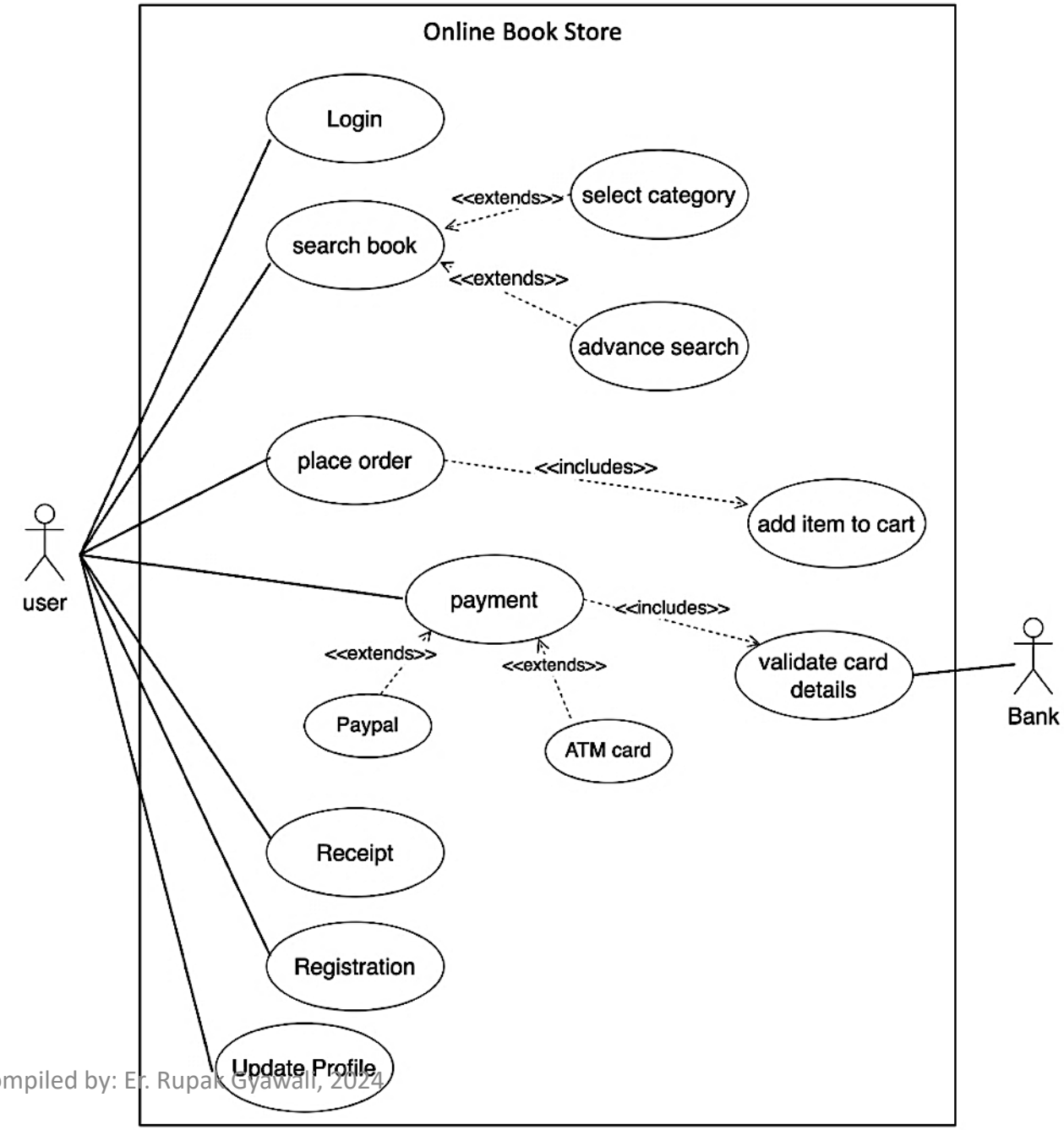
Use Case Diagram: Library Management System



Use Case Diagram: Library Management System using includes and extends



Use Case Diagram: Online Book Store



Interaction Models:

2. Sequence diagram:

- Shows the interaction between objects in sequential order.
- Sequence diagram is used to refine the use cases.

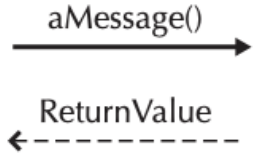
Elements/ Syntax of Sequence Diagram:

An actor: ■ Is a person or system that derives benefit from and is external to the system. ■ Participates in a sequence by sending and/or receiving messages. ■ Is placed across the top of the diagram. ■ Is depicted either as a stick figure (default) or, if a nonhuman actor is involved, as a rectangle with <<actor>> in it (alternative).	 anActor <<actor>> anActor
An object: ■ Participates in a sequence by sending and/or receiving messages. ■ Is placed across the top of the diagram.	anObject : aClass
A lifeline: ■ Denotes the life of an object during a sequence. ■ Contains an X at the point at which the class no longer interacts.	

Interaction Model:

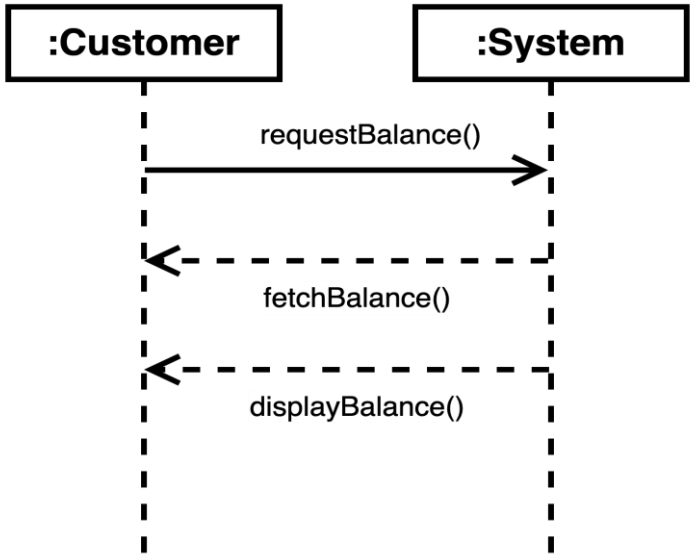
2. Sequence diagram:

Elements/ Syntax of Sequence Diagram:

An execution occurrence: ■ Is a long narrow rectangle placed atop a lifeline. ■ Denotes when an object is sending or receiving messages.	
A message: ■ Conveys information from one object to another one. ■ A operation call is labeled with the message being sent and a solid arrow, whereas a return is labeled with the value being returned and shown as a dashed arrow.	
A guard condition: ■ Represents a test that must be met for the message to be sent.	
For object destruction: ■ An X is placed at the end of an object's lifeline to show that it is going out of existence.	X

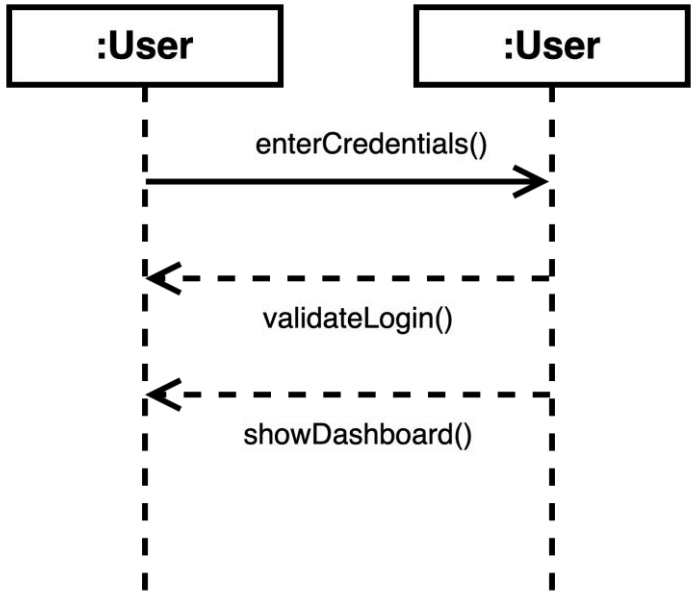
Interaction Model:

2. Sequence diagram:



Guidelines for creating Sequence Diagram:

- Strive for Left to Right Ordering of Messages
  - Example: In a banking application, a customer initiates a balance inquiry, and the system processes it in sequence.



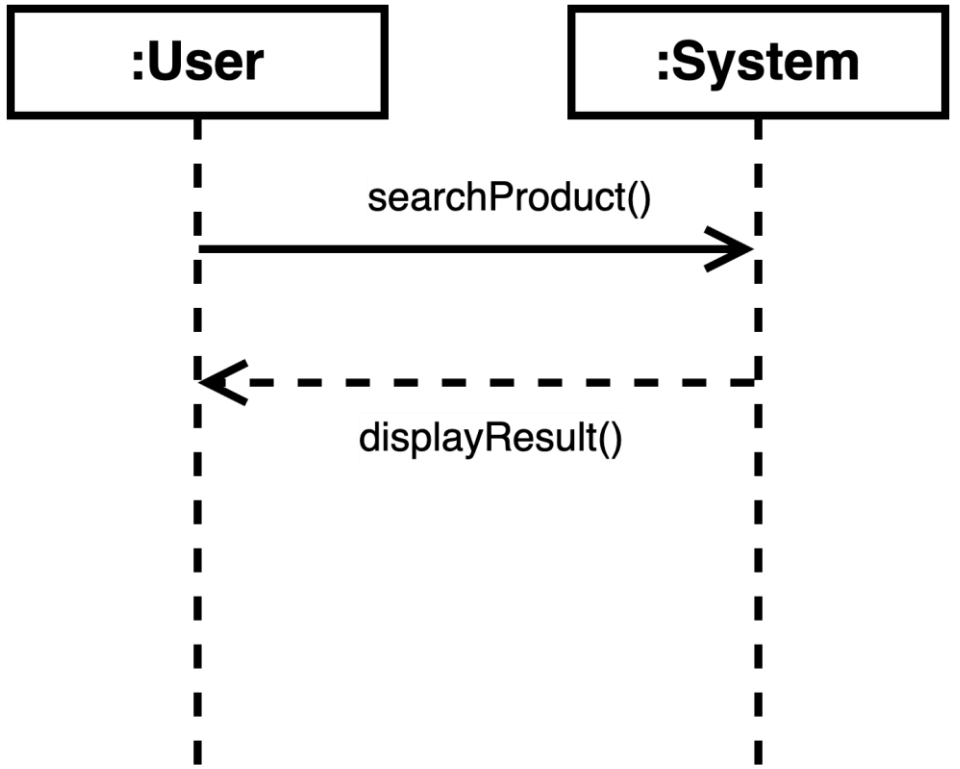
- If an Actor and Object Represent the Same Idea, Name Them the Same
  - Example: A user (actor) logging into a system. Both the actor and the object are named User.
  - Actor: User (the person trying to log in).
  - Object: User (the system's representation of the person, such as a user profile or account)

Interaction Model:

2. Sequence diagram:

<u>Guidelines for creating Sequence Diagram:</u>

- Only Show Return Values When They Are Not Clear
    - Example: A user searching for a product, where only the search result is shown.



Interaction Models:

2. Sequence diagram:

Process used to create a sequence diagram

i) **Set context:** Define the purpose and scope of the sequence diagram. Identify the specific use cases, system operation or interaction scenario you want to model.

- Eg: For library management system, the context could be “borrowing a book”

ii) **Identify the Actors and Objects:**

- Eg: For library management system, Actor → Student, Librarian. Object →Library System, Book

iii) **Setup the lifelines:** After identifying Actors and Objects, create a lifeline represented in vertical dashed lines. Lifeline shows the existence of an actor or object over time.

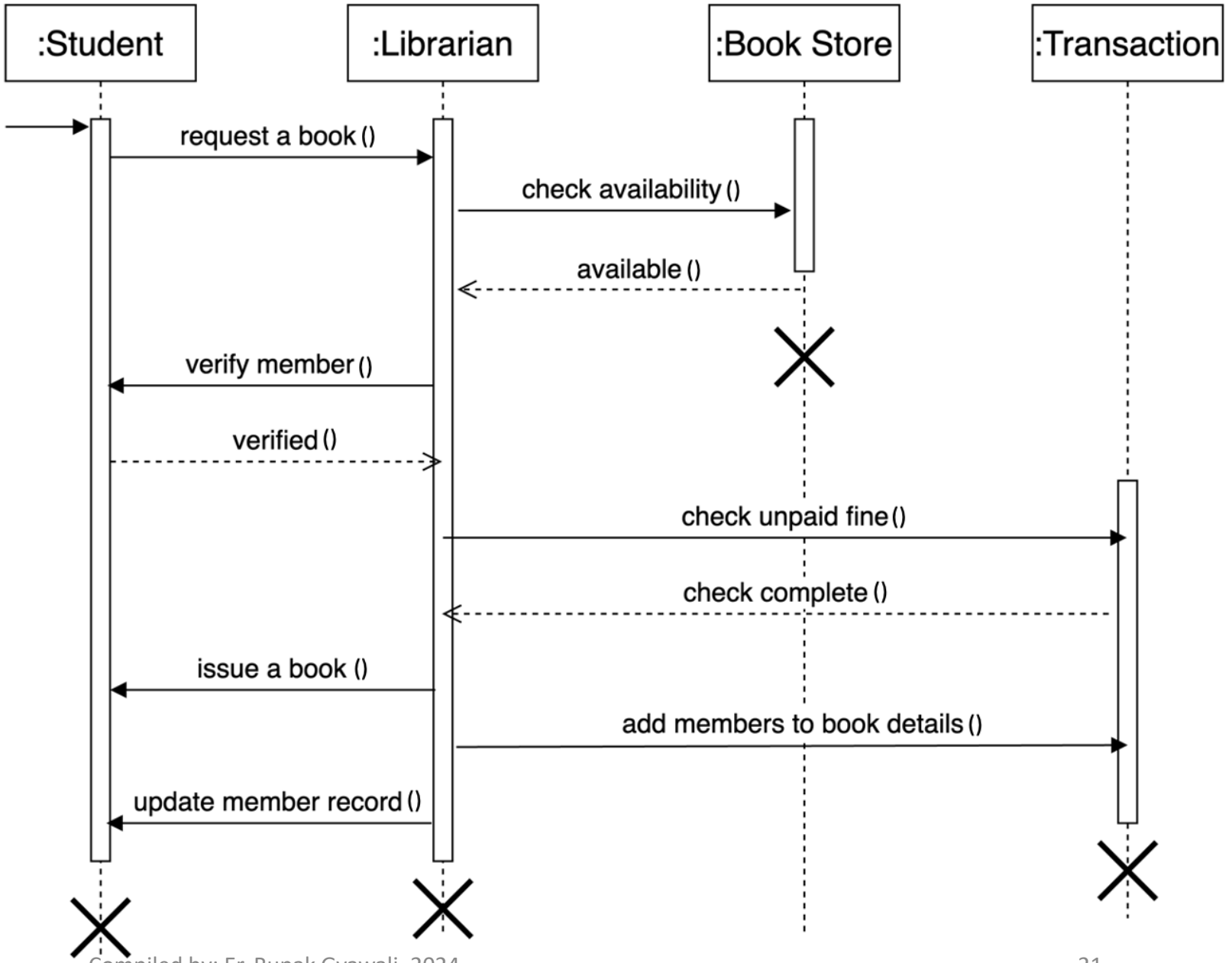
iv) **Add Messages:** Drawing arrows to represent the messages being passed from object to object, with the arrow pointing in the message’s transmission direction. Eg: requestbook(), borrowbook()

v) **Place Execution Occurrence:** Place the execution occurrence on each object’s lifeline by drawing a narrow rectangle box over the lifelines to represent when the classes are sending and receiving messages.

vi) **Validate:** Review the sequence diagram for accuracy and completeness.

Interaction Models:

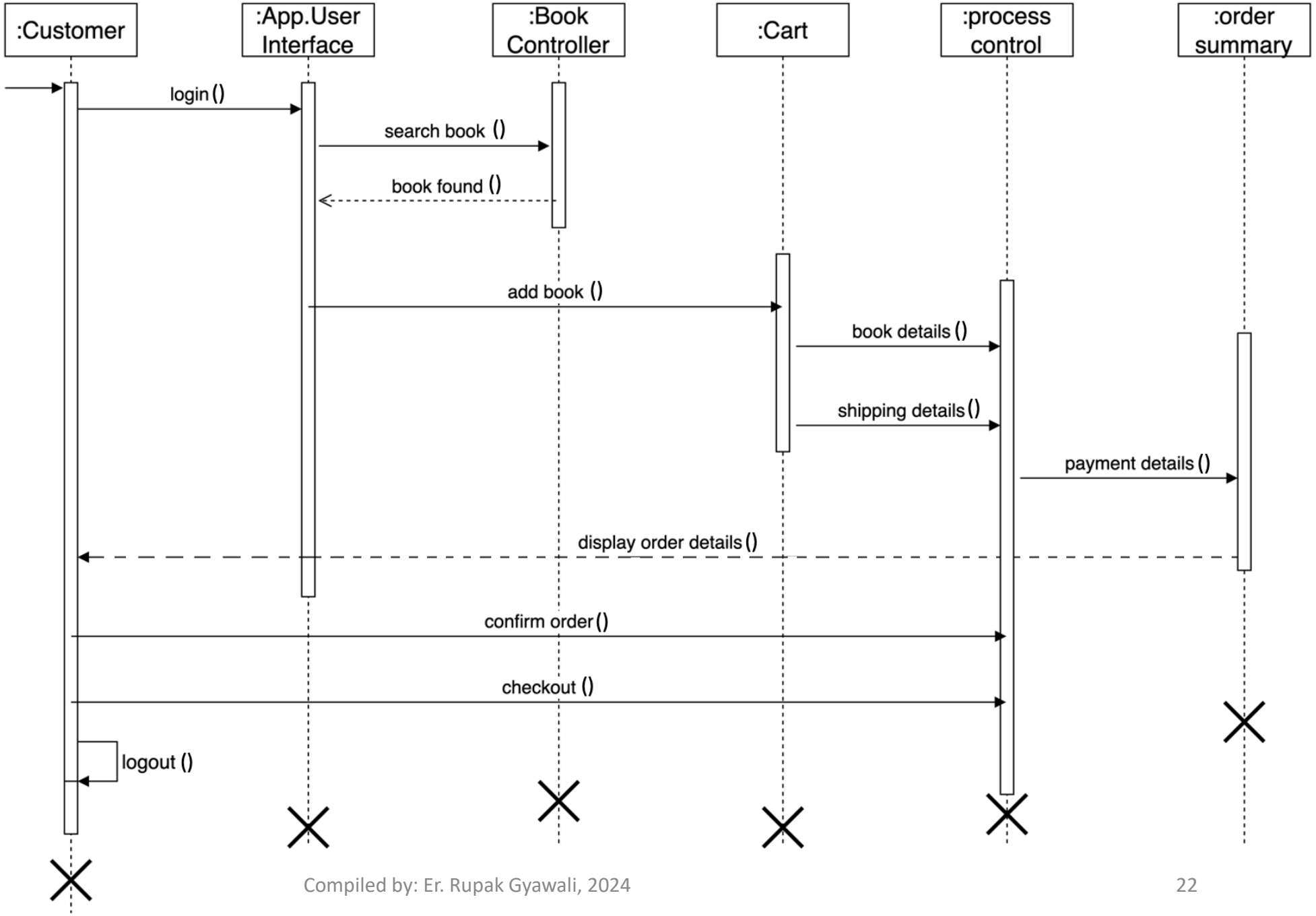
2. Sequence diagram: Library management system



Interaction Model:

2. Sequence diagram:

Online Book Store



Interaction Models:

2. Sequence diagram:

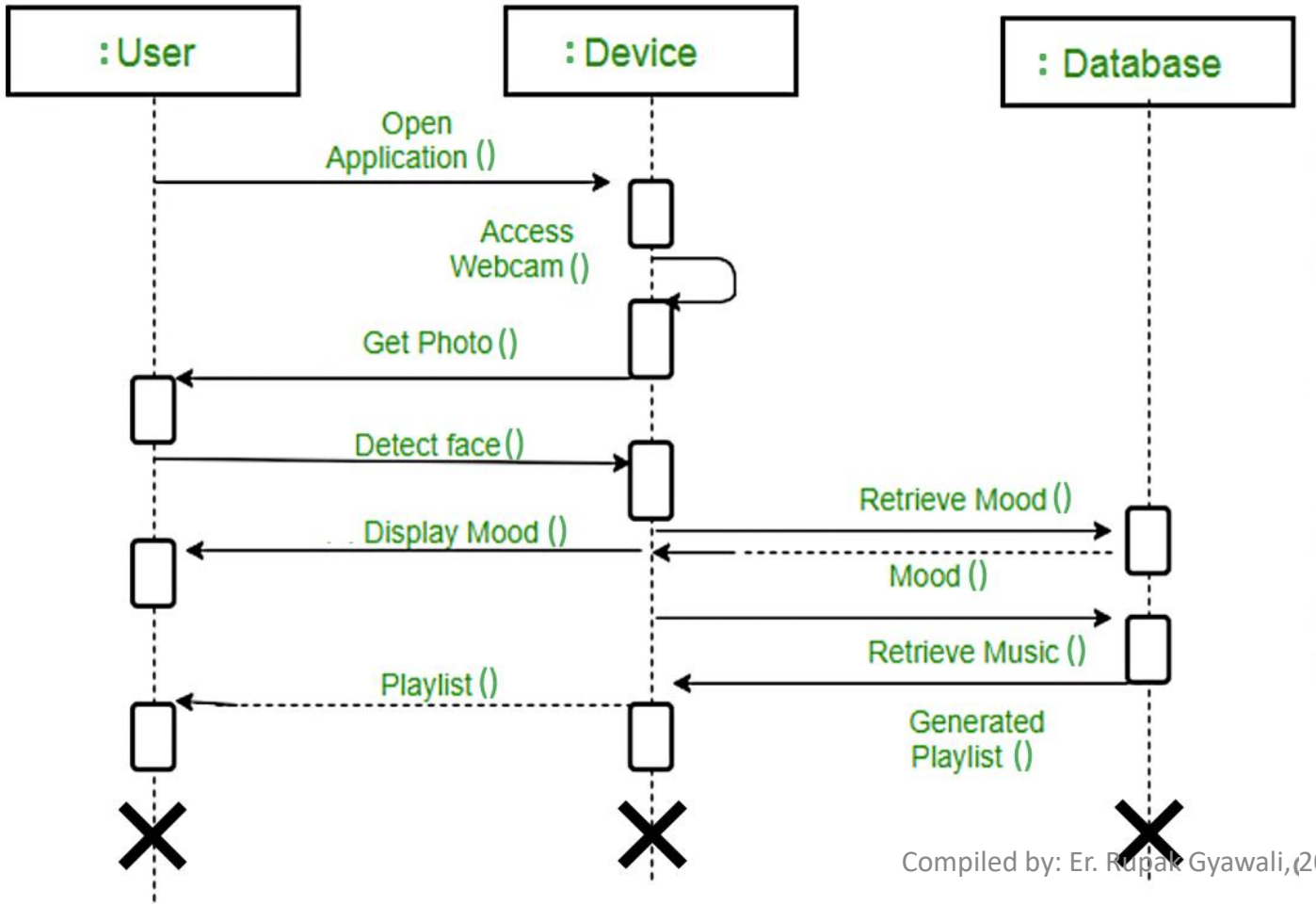
Exercise

Let us consider an emotion-based music player system which play the relevant music by detecting user's mood. When user open the application, application will use webcam and take picture of user. Then, application detect the face and predict the user's mood and mood is shown to user. Then by using the rule-based system, which is store in database, application will generate a list of songs based on user's mood and presented to system. Based on above scenario, make a sequence diagram with proper notation.

Interaction Models:

2. Sequence diagram:

Exercise Solution



1. Firstly, the application is opened by the user.
2. The device then gets access to the web cam.
3. The webcam captures the image of the user.
4. The device uses algorithms to detect the face and predict the mood.
5. It then requests database for dictionary of possible moods.
6. The mood is retrieved from the database.
7. The mood is displayed to the user.
8. The music is requested from the database.
9. The playlist is generated and finally shown to the user.

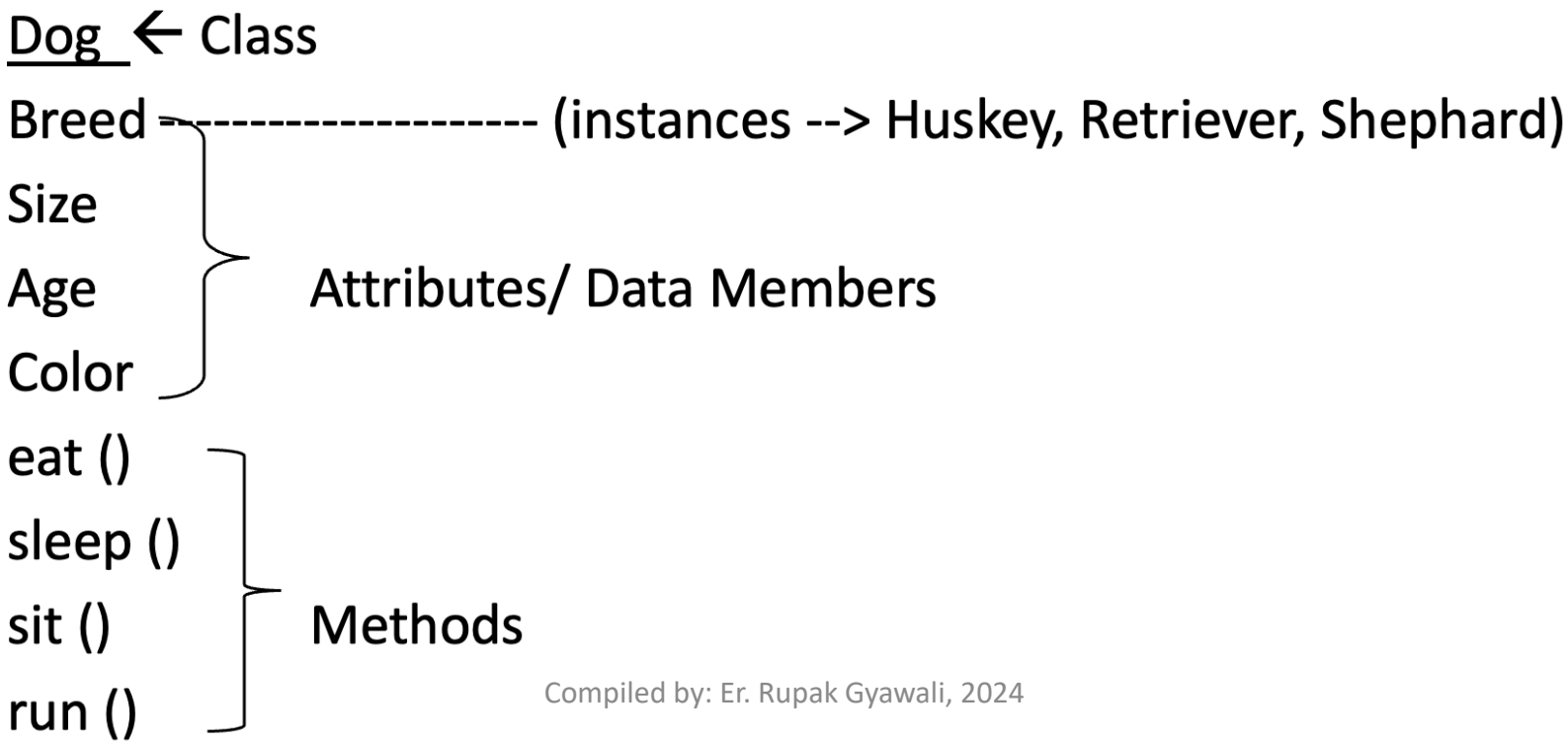
Structural Models:

- **Structural models** show how a system is organized, including its parts and how they are connected.
- **Static models**: Show the system's design and how it's structured, but not how it behaves while running.
- **Dynamic models**: Show how the system works while it's running, like how different parts interact.

Structural Models are: 1) Class Diagram 2) Object Diagram

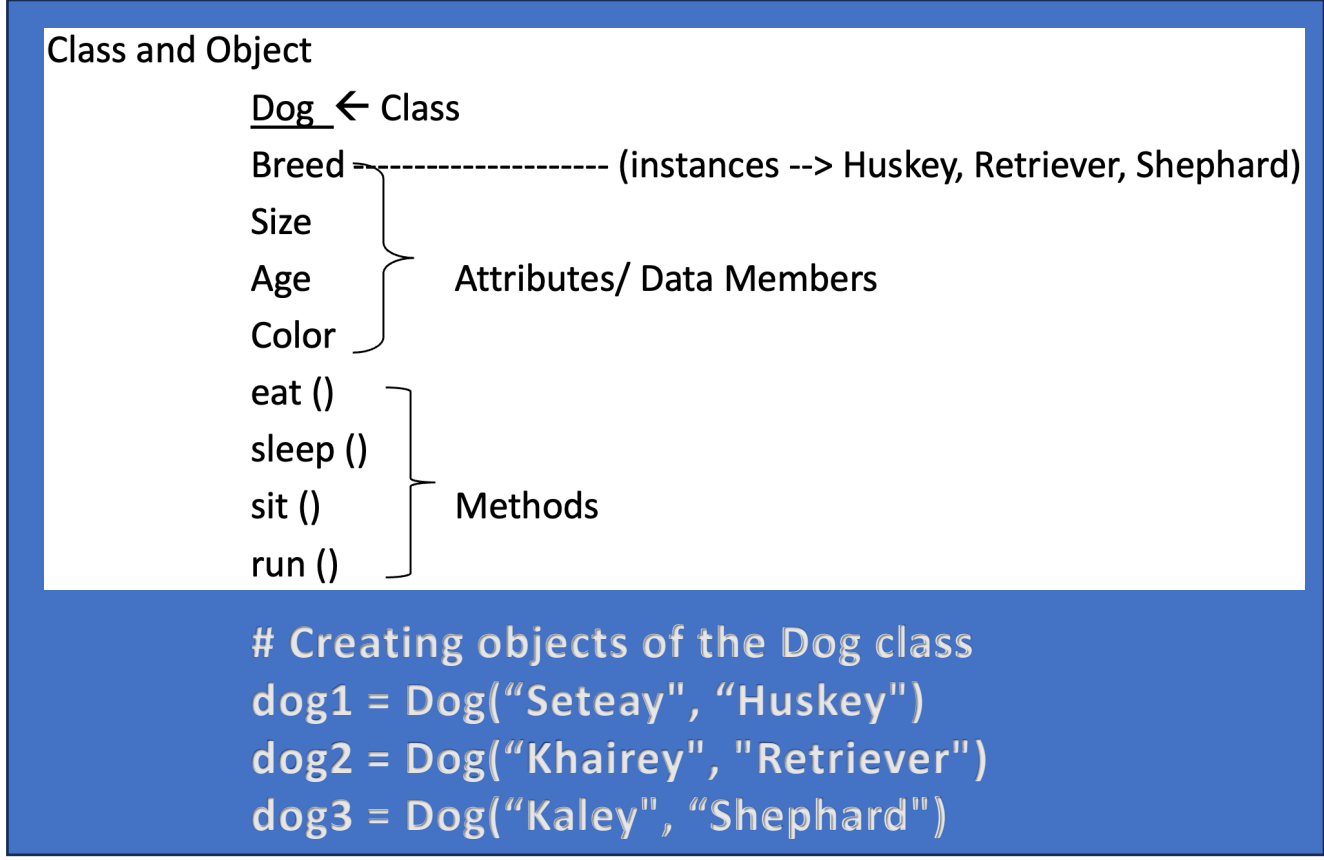
1) Class Diagram : Lets understand the concept of Class at first !

Class and Object



Structural Models:

1) Class Diagram



Class and Object contd…

Class:

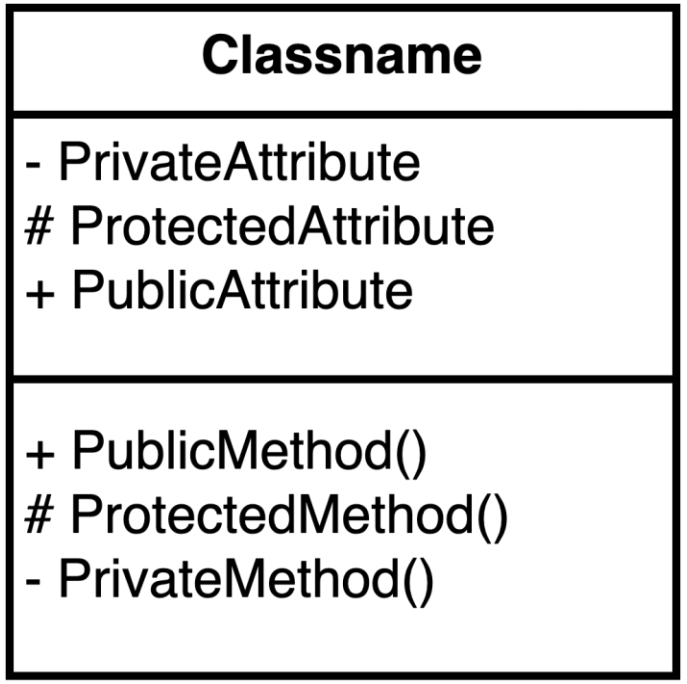
- Class is a collection of similar objects.
- Class is Conceptual.
- No memory is allocated for a class.
- Class represents real world entities eg. Dog.
- Class can exists without objects.

Object:

- An instance of a class. Eg. Huskey.
- Object is real and has instance, identity and behavior.
    - Instance: husky is an instance of the Dog class.
    - Identity: husky has a unique memory location and identity.
    - Behavior: husky can perform actions, such as barking, which are defined in the Dog class.
- Each object has its own memory.
- Objects can't exists without a class.

Structural Model:

1) Class Diagram

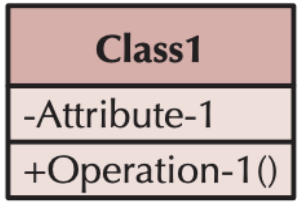


- A static diagram: A class diagram is considered static because it represents the structure of a system at a particular point in time, showing the system's classes, attributes, methods, and relationships between classes without detailing the dynamic behaviors, interactions, or state changes that occur at runtime.
- It is static means like taking a snapshot of the system's blueprint, which remains the same regardless of what the system is doing at a given moment.
- Most popular UML diagram in the coder community.
- Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.
- Class diagram describes the attributes and operations of a class and also the constraints imposed on the system.
- shows a collection of classes, interfaces, associations, collaborations, and constraints.

Structural Models:

1) Class Diagram

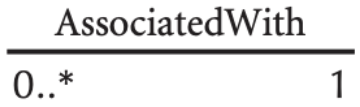
Elements/ Syntax of Class Diagram:

A class: • Represents a kind of person, place, or thing about which the system will need to capture and store information. • Has a name typed in bold and centered in its top compartment. • Has a list of attributes in its middle compartment. • Has a list of operations in its bottom compartment. • Does not explicitly show operations that are available to all classes.	
An attribute: • Represents properties that describe the state of an object. • Can be derived from other attributes, shown by placing a slash before the attribute's name.	attribute name /derived attribute name
An operation: • Represents the actions or functions that a class can perform. • Can be classified as a constructor, query, or update operation. • Includes parentheses that may contain parameters or information needed to perform the operation.	operation name ()

Structural Models:

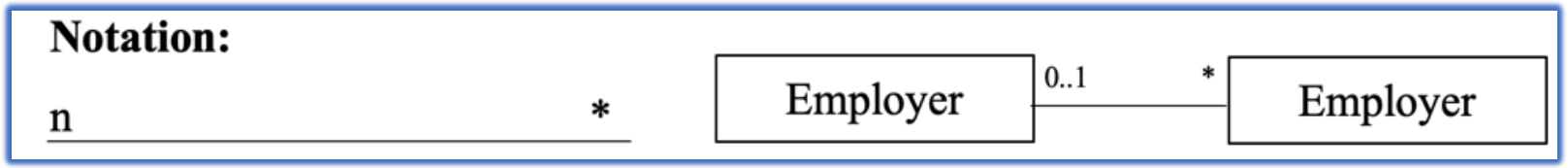
1) Class Diagram

Elements/ Syntax of Class Diagram:

An association: • Represents a relationship between multiple classes or a class and itself. • Is labeled using a verb phrase or a role name, whichever better represents the relationship. • Can exist between one or more classes. • Contains multiplicity symbols, which represent the minimum and maximum times a class instance can be associated with the related class instance.	
A generalization: • Represents a-kind-of relationship between multiple classes.	
An aggregation: • Represents a logical a-part-of relationship between multiple classes or a class and itself. • Is a special form of an association.	
A composition: • Represents a physical a-part-of relationship between multiple classes or a class and itself • Is a special form of an association.	

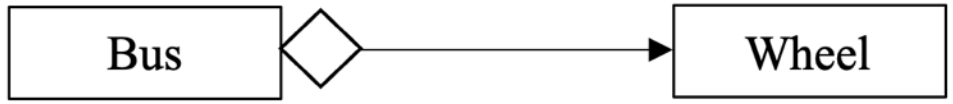
 # Association: An association represents a relationship between two or more classes. It describes how the classes are connected or related to each other. Associations can be one-to-one, one-to-many, or many-to-many.

Notation:

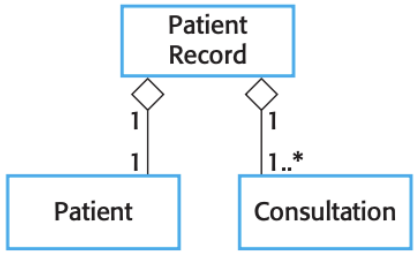


a) Aggregation: "Has a" relationship.

Notation:



Even if the container is destroyed then its content is not destroyed.



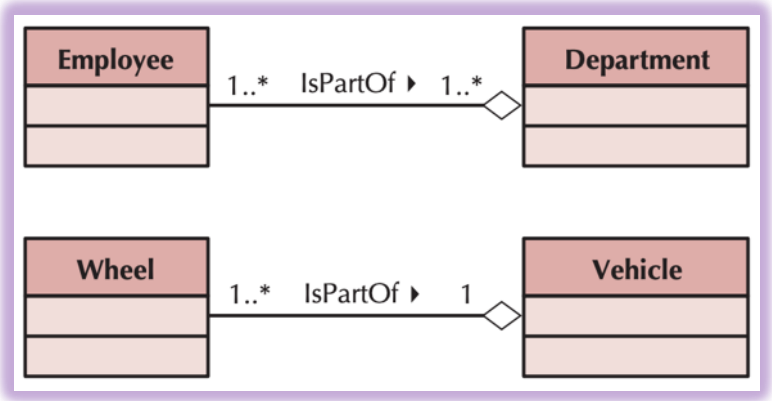
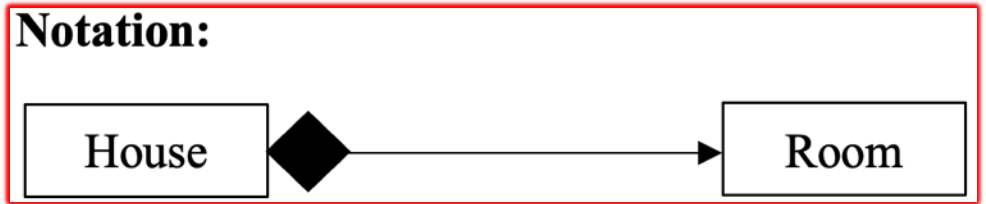


Fig: Eg. of Aggregation

b) Composition: "Owns a" relationship. A composition is a relationship between two classes where one class is made up of one or more instances of the other class.

Notation:



If container is destroyed then its content is also destroyed.

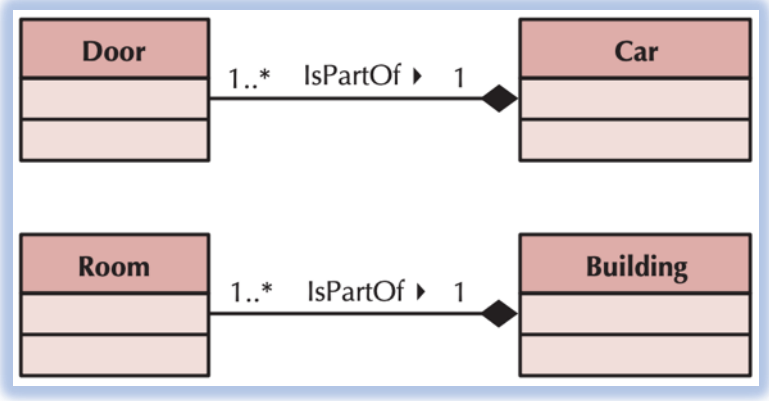


Fig: Eg. of Composition

Association:

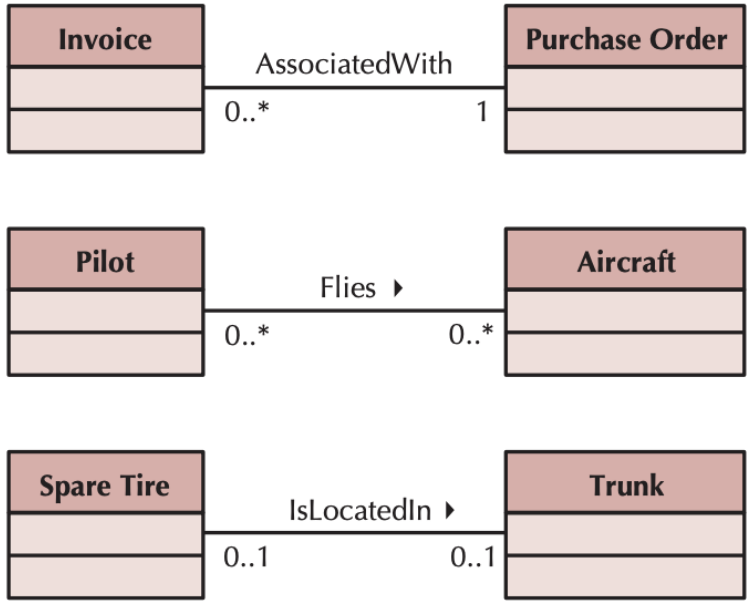


Fig: Simple Association

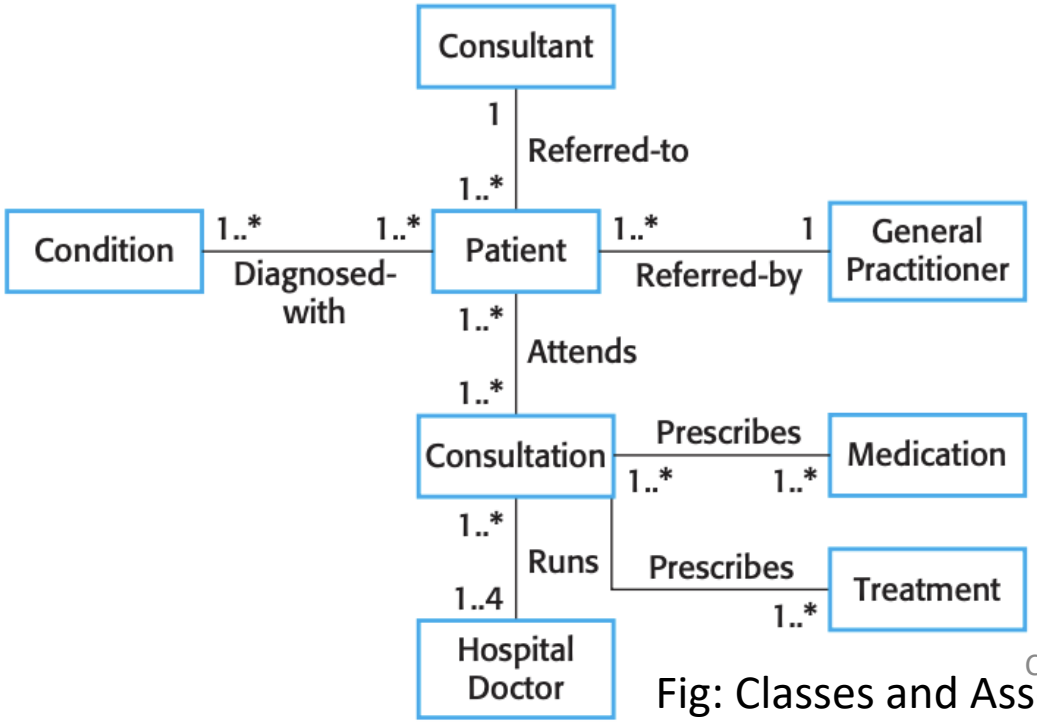


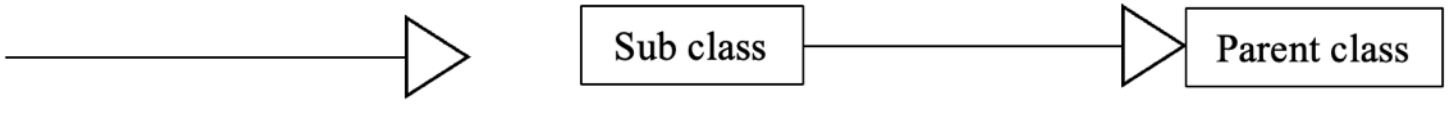
Fig: Classes and Association Relationship in MHC-PMS

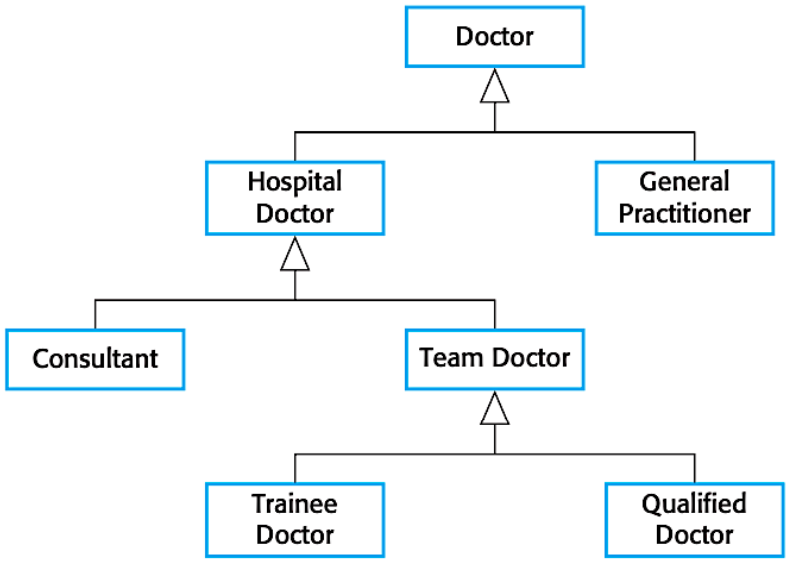
Exactly one	1	Department — 1 — Boss	A department has one and only one boss.
Zero or more	0..*	Employee — 0..* — Child	An employee has zero to many children.
One or more	1..*	Boss — 1..* — Employee	A boss is responsible for one or more employees.
Zero or one	0..1	Employee — 0..1 — Spouse	An employee can be married to zero or one spouse.
Specified range	2..4	Employee — 2..4 — Vacation	An employee can take from two to four vacations each year.
Multiple, disjoint ranges	1..3,5	Employee — 1..3,5 — Committee	An employee is a member of one to three or five committees.

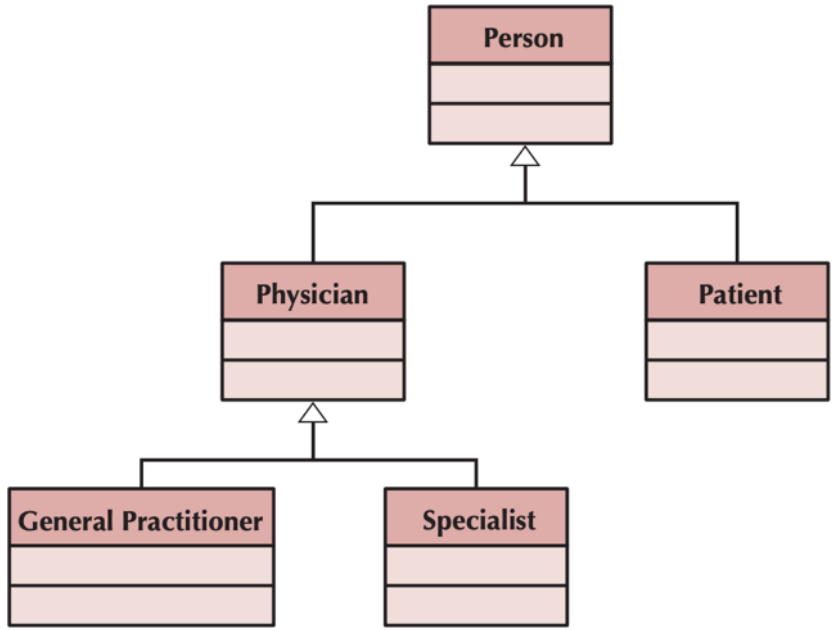
Fig: Multiplicity

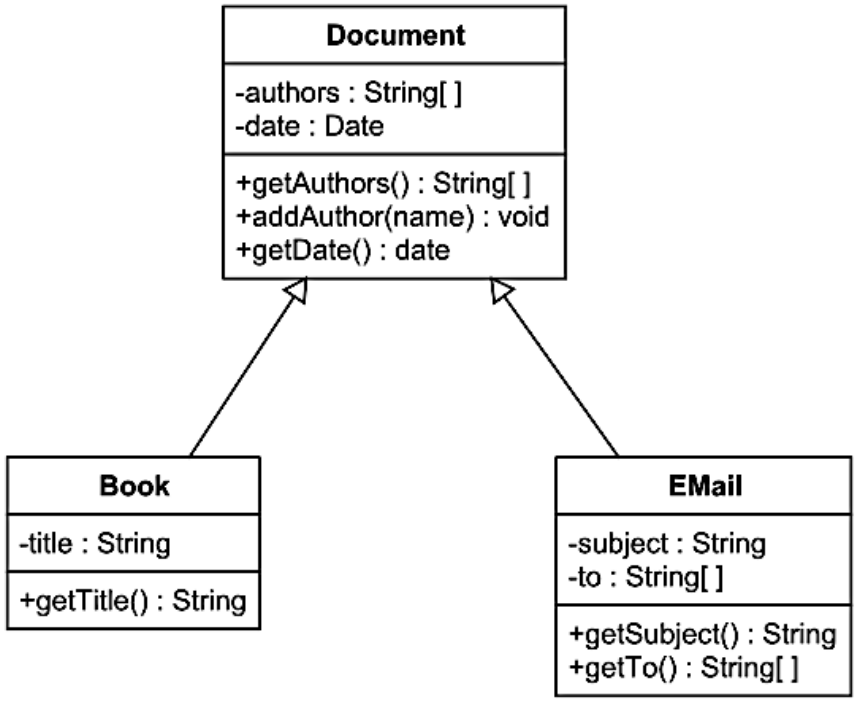
Generalization: A generalization is a relationship between two classes where one class is a specialized version of the other. The specialized class (called the subclass or derived class) inherits the attributes and behaviors of the more general class (called the superclass or base class) and may add its own unique features.

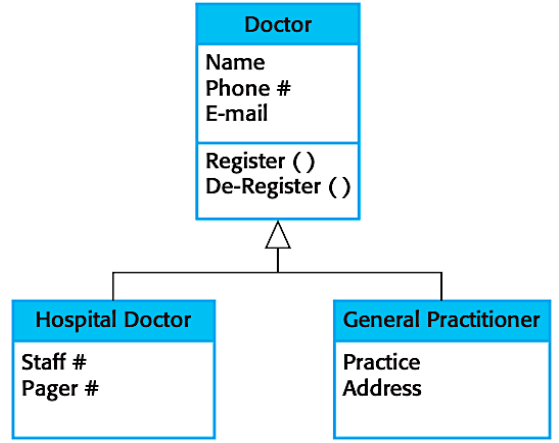
Notation:







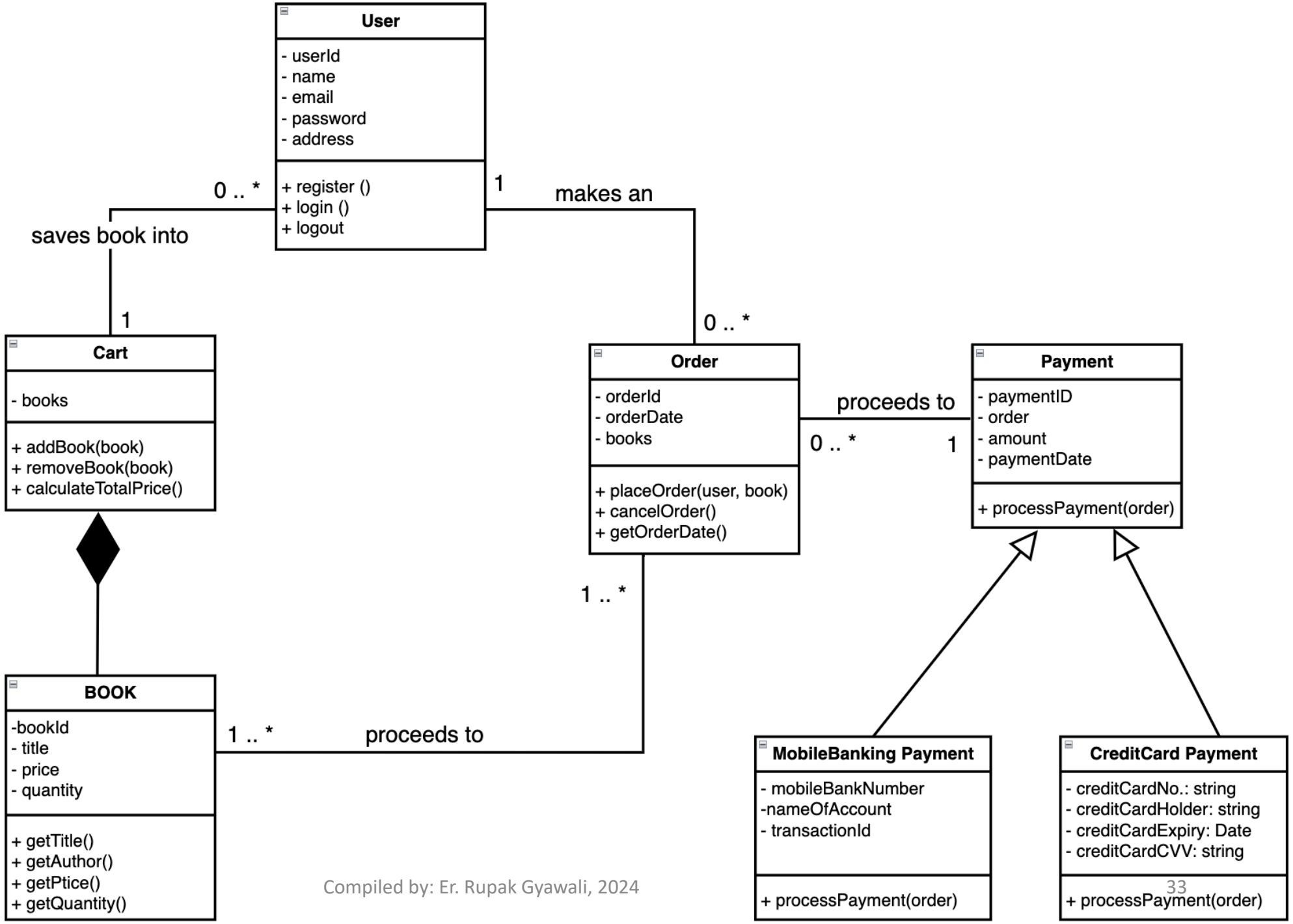




Structural Model:

1) Class Diagram

Eg: Online Book Store



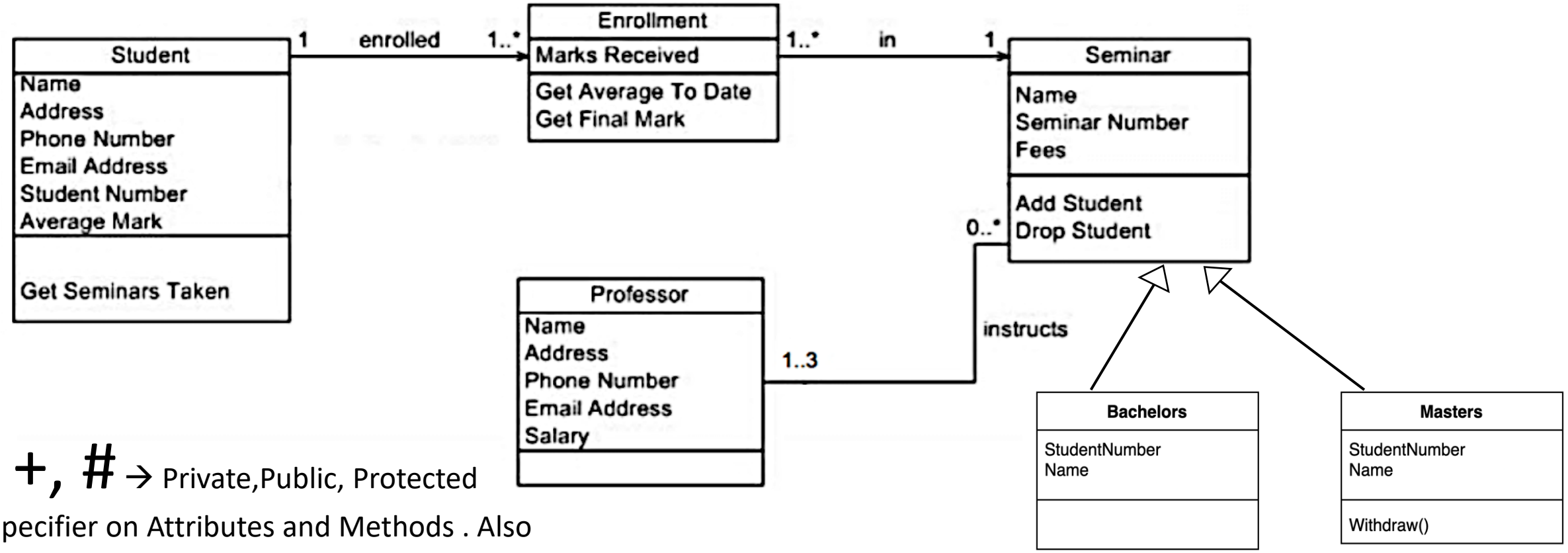
Use -, +, # →

Private,Public, Protected Access Specifier on Attributes and Methods respictevely.

Structural Models:

1) Class Diagram

Example: A professors has a name, address, phone number, email address, and salary. A student has also a name, etc., but no salary (sorry). A student, however, has an average mark (of the final marks of his or her seminars). A seminar has a name and a number. When a student is enrolled in a seminar, the marks for this enrollment are recorded and the current average as well as the final mark (if there is one) can be obtained from the enrollment. From a student one can obtain a list of seminars he or she is enrolled in. Professors teach seminars. Each seminar has at least one and at most three teachers. There are two types of seminar: bachelor and master. From a bachelor seminar students can not withdraw. From a master seminar they can.

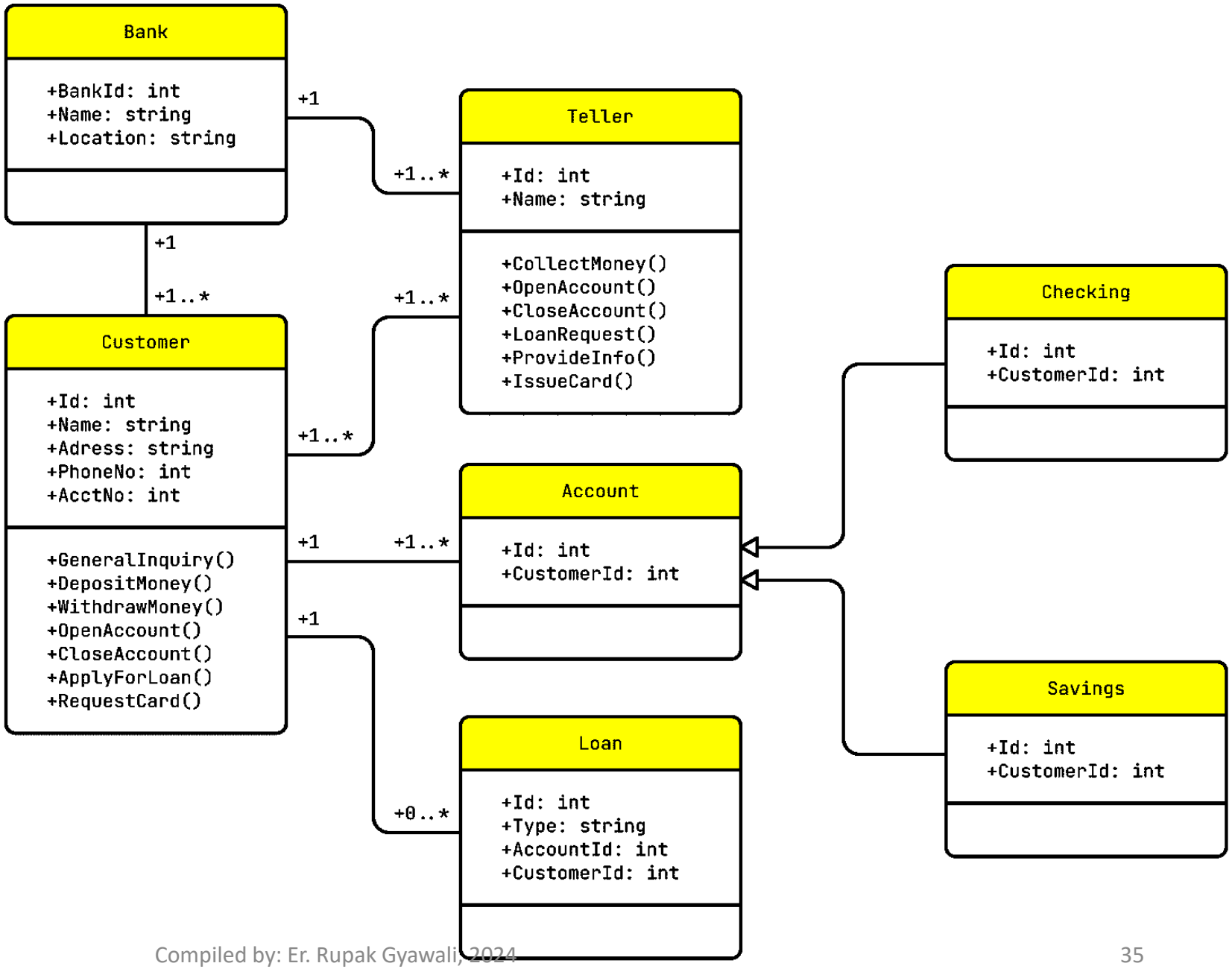


Use -, +, # → Private,Public, Protected Access Specifier on Attributes and Methods . Also use Datatypes int, string, date etc. respectevely.

Structural Models:

1) Class Diagram

Example: Banking System



Structural Models:

2) Object Diagram

- Object diagrams are derived from class diagrams.
- Represent an instance of a class diagram.
- The basic concepts are similar for class diagrams and object diagrams but object diagram does not have methods, only contains object name and attributes.
- Object diagrams also represent the static view of a system, but this static view is a snapshot of the system at a particular moment.
- Represents set of objects and their relationships.
- Syntax??

Structural Models:

2) Object Diagram

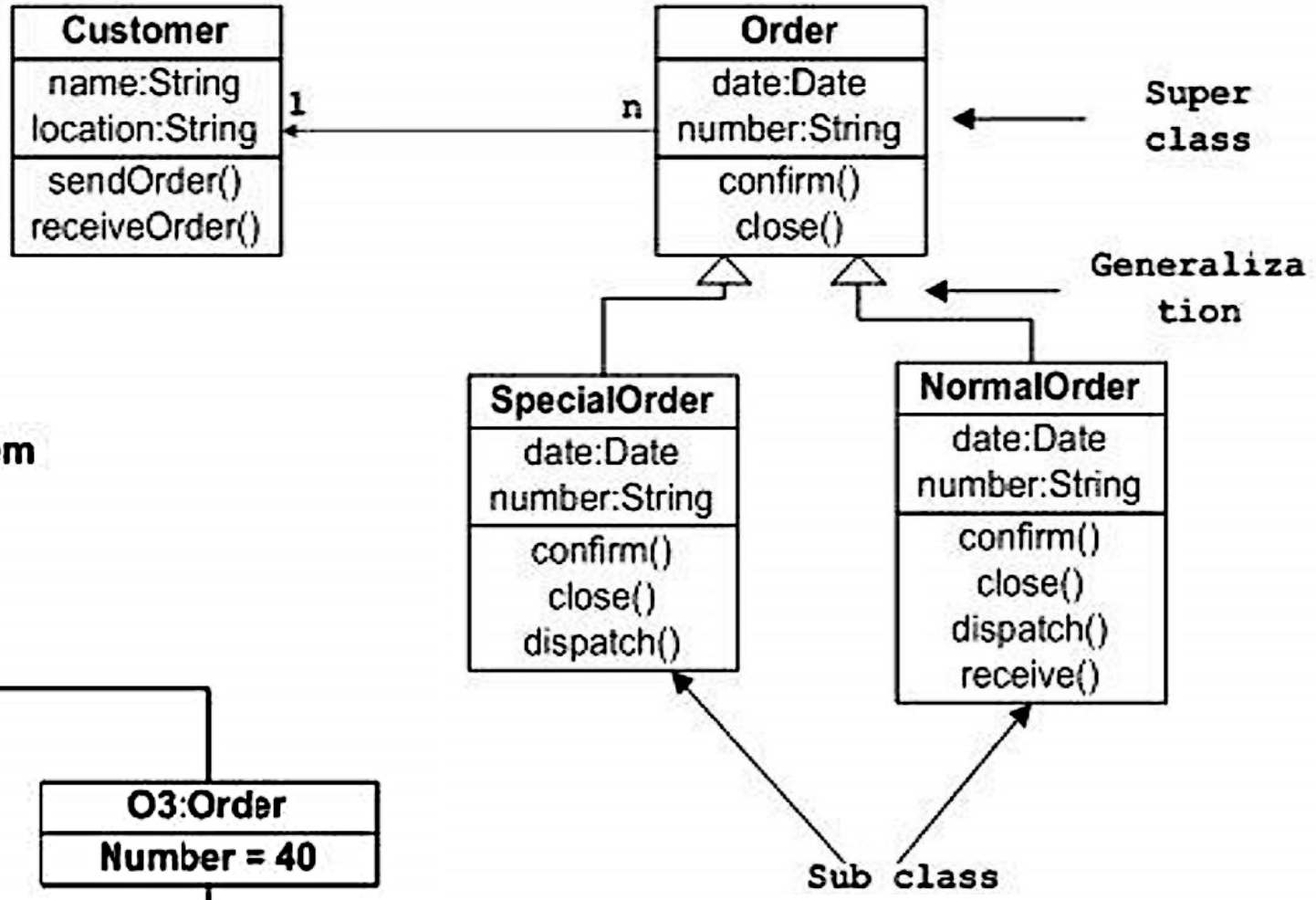
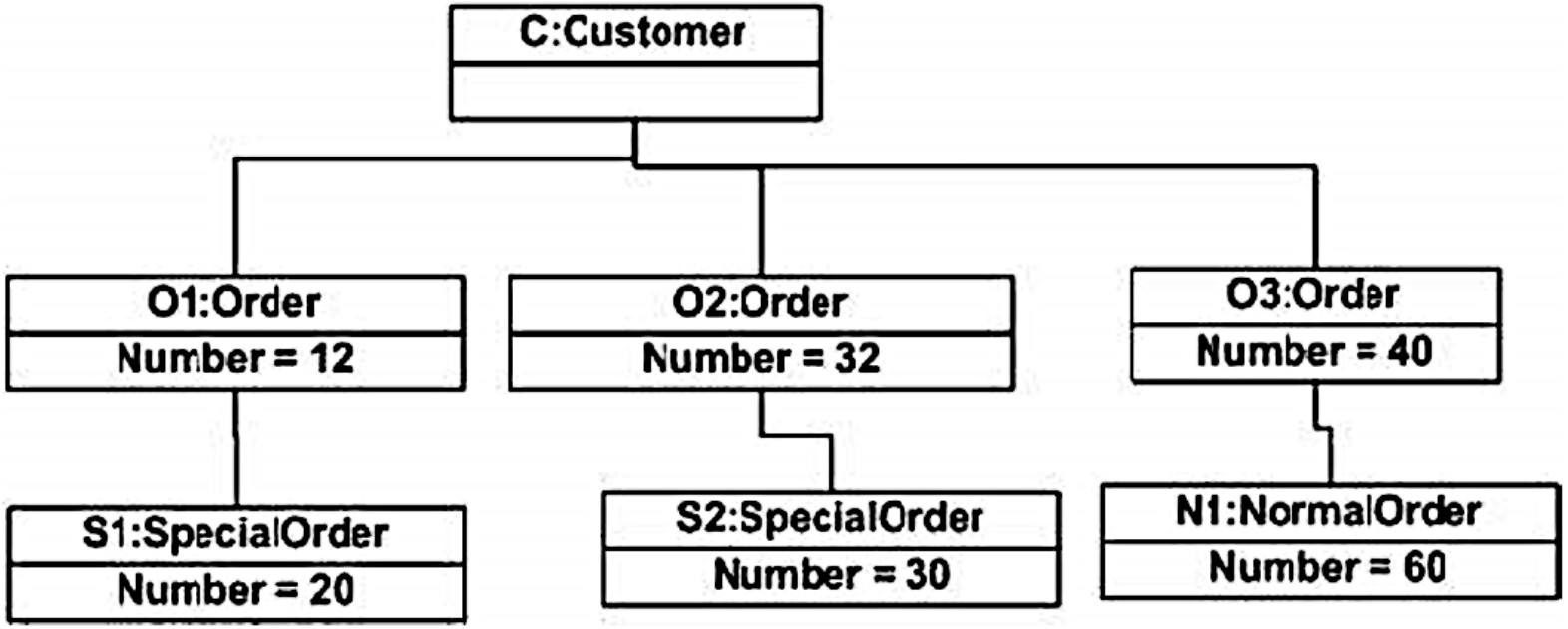


Fig: class diagram of order management system

Object diagram of an order management system



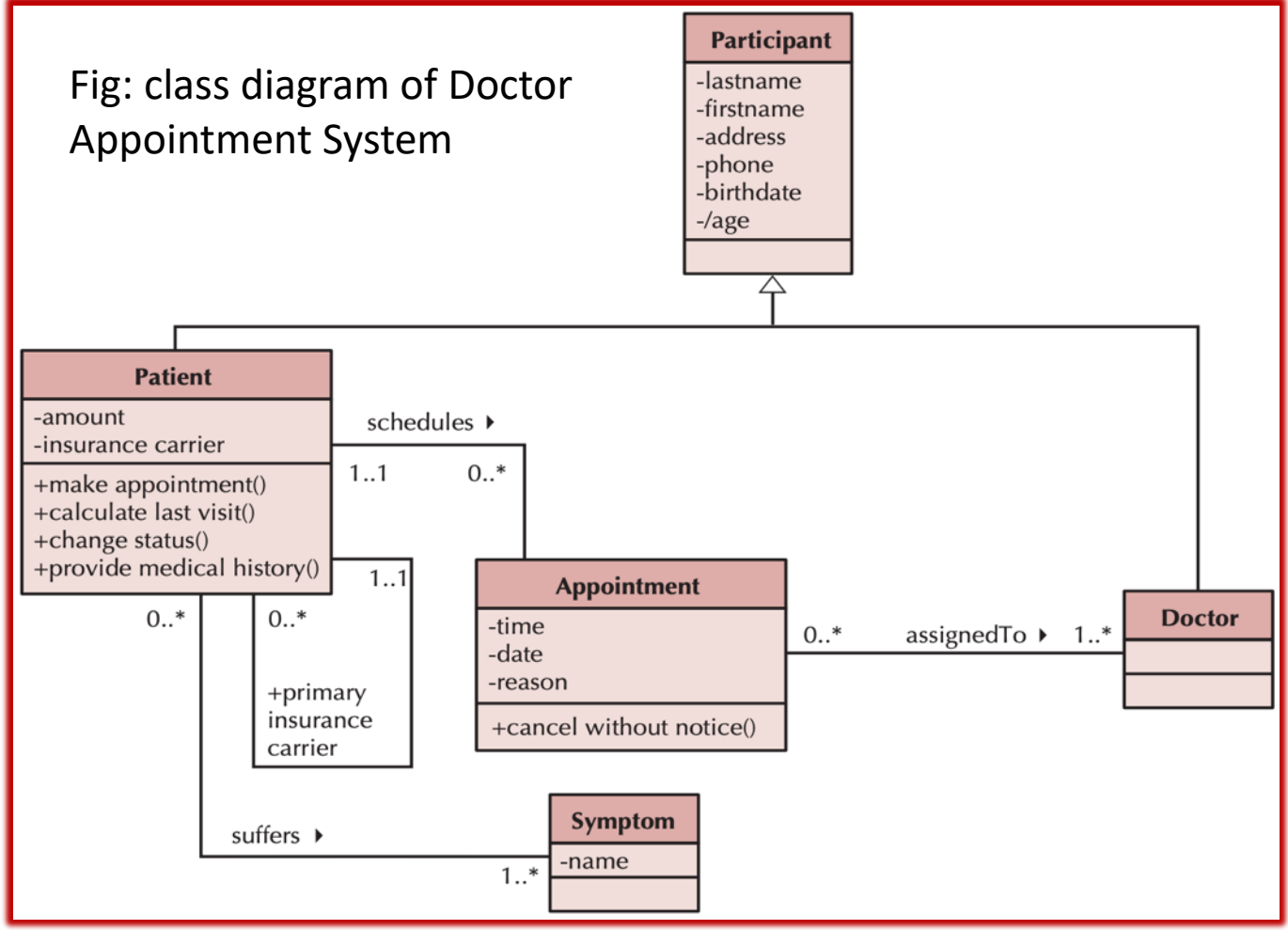
Sale	:Sale	S1:Sale	S1:Sale
class	Object	Named Object	Named actual Object

Structural Models:

2) Object Diagram

Conversion of class diagram to object diagram

Fig: class diagram of Doctor Appointment System



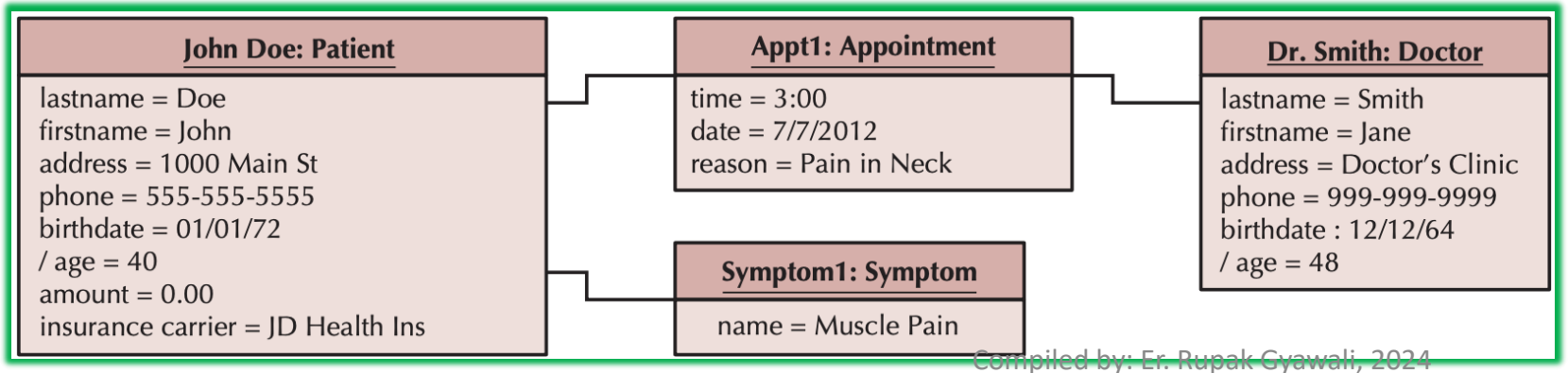


Fig: object diagram of Doctor Appointment System

Behavioral Model:

- **Behavioral models** show how a system behaves or reacts when it is running. They focus on what happens when the system gets a stimulus (a signal or input) from its surroundings. There are two main types of stimuli:
- **Data:** The system gets data that it needs to process.
- Example: A **library management system** processes book details, borrower information, and due dates to calculate and display overdue fines.
  - **Data Input:** Book borrowed, return date.
  - **Processing:** Checks if the return is late, calculates the fine.
  - **Output:** Displays the overdue fine.
- **Events:** Something happens that triggers the system to act.
- Example: An **elevator system** reacts when a floor button is pressed.
  - **Event Trigger:** Someone presses the button for floor 5.
  - **System Reaction:** The elevator moves to floor 5 and opens the doors.

Two types of Behavioral Models:

1) Data-Driven Modeling
2) Event-Driven Modeling

1) Data-Driven Models:

Data-driven modeling focuses on showing how input data is processed step by step until it produces an output. It helps understand and document the full flow of data through a system, making it especially useful when analyzing requirements.

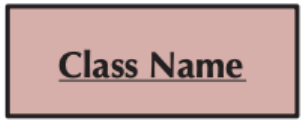
- **What it does**
  - Shows the sequence of actions from input to output.
  - Helps visualize how data moves and is processed within the system.
- **History**
  - In the 1970s, **Data-Flow Diagrams (DFDs)** were introduced to illustrate processing steps in a system.
  - DFDs are simple and easy to understand, making them useful for discussing designs with non-technical users.
- **UML and Data-Driven Models**
  - Traditional DFDs aren't part of UML because they focus on system functions, not objects.
  - Instead, **UML Activity Diagrams** and **Sequence Diagrams** are used to model data flows.
    a) **Activity Diagrams**: Show the steps in processing and the data flowing between them.
    b) **Sequence Diagrams**: If arranged so messages flow left to right, they show sequential data processing.

Behavioral Model: 1) Data-Driven Models:

a) Activity Diagram

- A flowchart to represent the flow from one activity to another activity
- Flowchart like diagrams
- Activity is a particular operation of the system
- Flow can be sequential, branched, or concurrent
- Swim lanes: Shows what activities are performed by an activity diagram. Swim lanes also clarifies which object/ class performs which activity.

Elements/ Syntax of Activity Diagram:

An activity: ■ Is used to represent a set of actions. ■ Is labeled by its name.	
An object node: ■ Is used to represent an object that is connected to a set of object flows. ■ Is labeled by its class name.	
A control flow: ■ Shows the sequence of execution.	
An object flow: ■ Shows the flow of an object from one activity (or action) to another activity (or action).	

Behavioral Model:

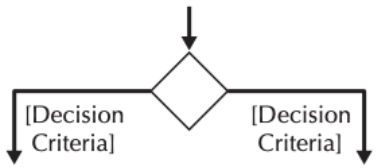
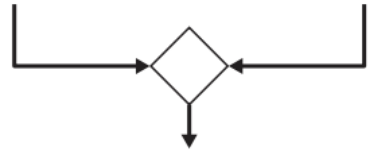
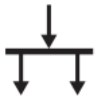
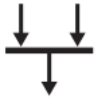
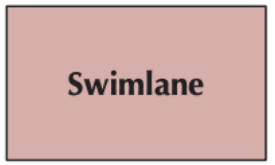
1) Data-Driven Models:

a) Activity Diagram

<u>Elements/ Syntax of Activity Diagram:</u>

Control Nodes: There are seven different types of control nodes in an activity diagram:

- initial,
- final-activity,
- final-flow,
- decision,
- merge,
- fork,
- join

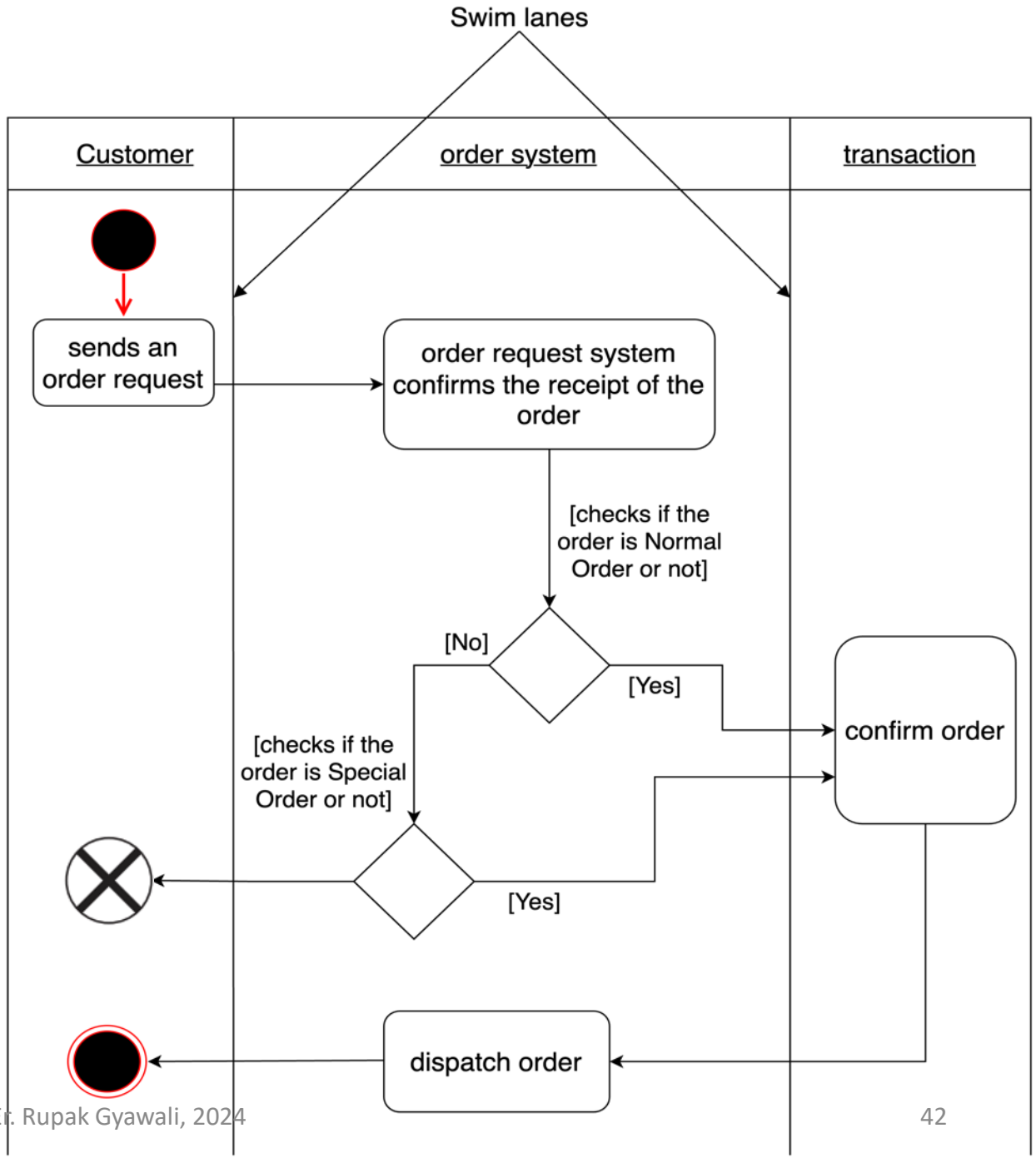
An initial node: ■ Portrays the beginning of a set of actions or activities.	
A final-activity node: ■ Is used to stop all control flows and object flows in an activity (or action).	
A final-flow node: ■ Is used to stop a specific control flow or object flow.	
A decision node: ■ Is used to represent a test condition to ensure that the control flow or object flow only goes down one path. ■ Is labeled with the decision criteria to continue down the specific path.	
A merge node: ■ Is used to bring back together different decision paths that were created using a decision node.	
A fork node: Is used to split behavior into a set of parallel or concurrent flows of activities (or actions)	
A join node: Is used to bring back together a set of parallel or concurrent flows of activities (or actions)	
A swimlane: Is used to break up an activity diagram into rows and columns to assign the individual activities (or actions) to the individuals or objects that are responsible for executing the activity (or action) Is labeled with the name of the individual or object responsible	

Behavioral Modeling:

1) Data-Driven Models:

a) Activity Diagram:

Eg: Order Management System



Behavioral Modeling:

1) Data-Driven Models:

a) Activity Diagram

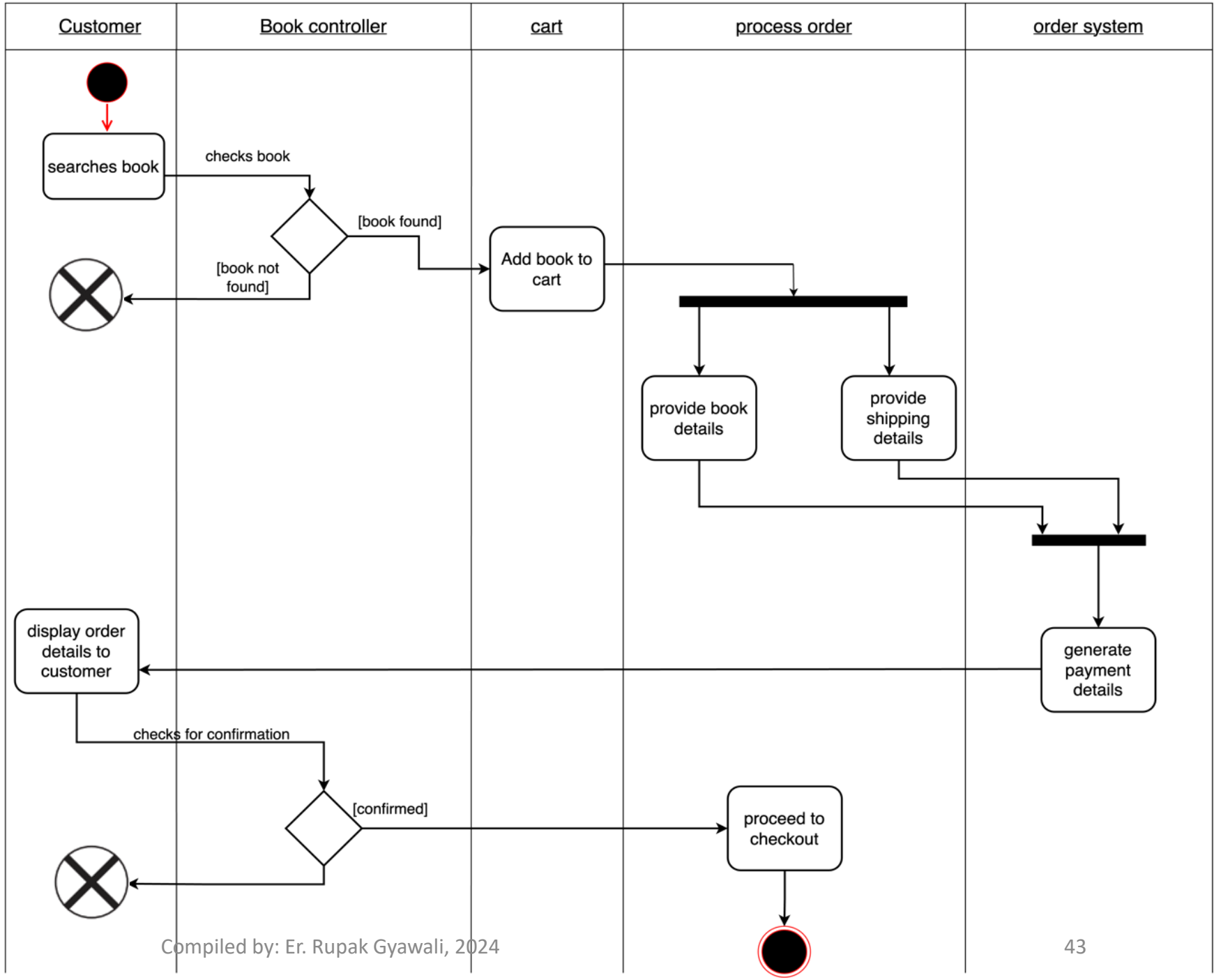
Eg: Online Book Store

Common types of Data-Driven Models are:

a) Activity Diagram

b) Sequence Diagram

c) Data Flow Diagram (DFD)



Behavioral Modeling:

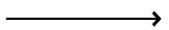
2) Event-driven modeling: It shows how a system reacts to events. It assumes the system has a limited number of states, and events (triggers) can change it from one state to another. For example, a valve system can change from "Open" to "Closed" when an operator sends a command. This approach works well for real-time systems.

- **Events**: Stimuli that cause the system to move from one state to another (e.g., pressing a button).
- Event-Driven Modeling can be done by **State Chart/ State machine Diagram**

a) State Chart / State Machine Diagram

- Activity diagram is a special kind of a State chart diagram.
- Describes the flow of control from one system state to another state.
    - **System States**: Represent the condition or mode of the system (e.g., "Waiting," "Cooking").

Elements/ Syntax of State chart Diagram

A state: ■ Is shown as a rectangle with rounded corners. ■ Has a name that represents the state of an object.	
An initial state: ■ Is shown as a small, filled-in circle. ■ Represents the point at which an object begins to exist.	
A final state: ■ Is shown as a circle surrounding a small, filled-in circle (bull's-eye). ■ Represents the completion of activity.	
An event: ■ Is a noteworthy occurrence that triggers a change in state. ■ Can be a designated condition becoming true, the receipt of an explicit signal from one object to another, or the passage of a designated period of time. ■ Is used to label a transition.	anEvent
A transition: ■ Indicates that an object in the first state will enter the second state. ■ Is triggered by the occurrence of the event labeling the transition. ■ Is shown as a solid arrow from one state to another, labeled by the event name.	

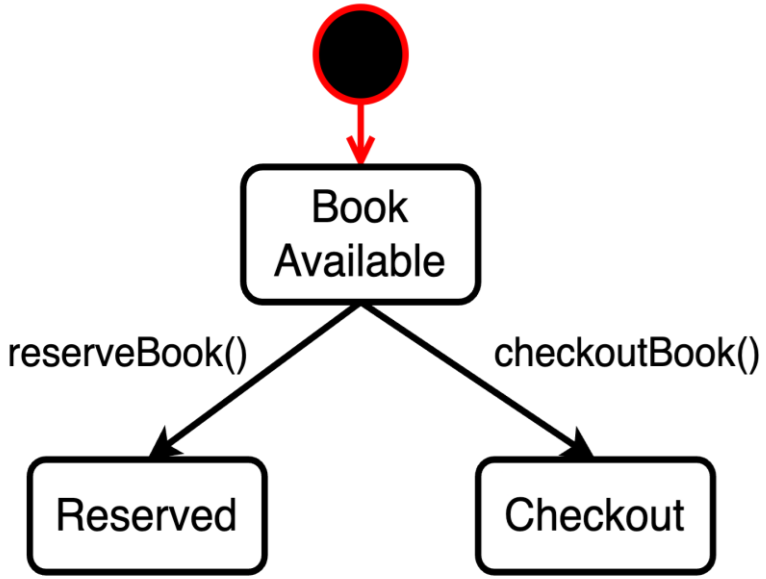
Behavioral Modeling:

2) Event-Driven Modeling:

a) State Chart / State Machine Diagram

Guidelines:

- **Only create behavioral state machines for "complex" objects.**
- **Draw the initial state in the top left corner of the diagram.**
- **Draw the final state in the bottom right corner of the diagram.**
- **Use simple, but descriptive, names for states.**
- **Make sure transitions are associated with messages and operations:** Transition from "Available" to "Reserved" is triggered by reserveBook().



For a library system,

Object: Library Book

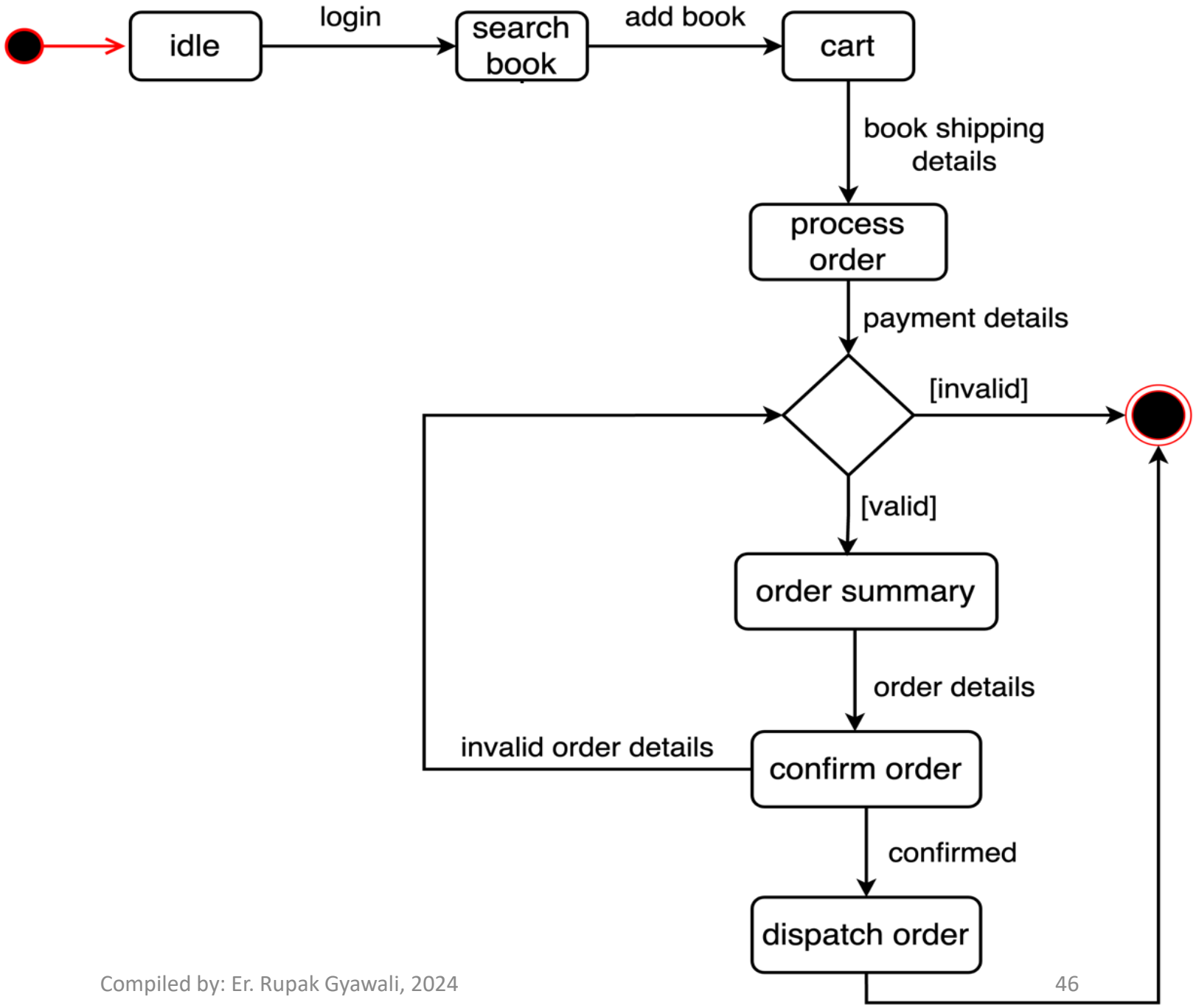
States: Available, Reserved, Checked Out, Returned

Behavioral Modeling:

2) Event-Driven Modeling:

a) State Chart / State Machine Diagram:

Eg: Online Book Store



Behavioral Modeling:

2) Event-Driven Modeling:

a) State Chart / State Machine Diagram:

Example: Simple Microwave

- A microwave system works as follows:

1. Select power (half = 300 watt or full = 600 watt).
2. Input cooking time.
3. Press Start (if the door is closed), and the food cooks.
4. After cooking, a buzzer sounds, and the system goes back to "Waiting."

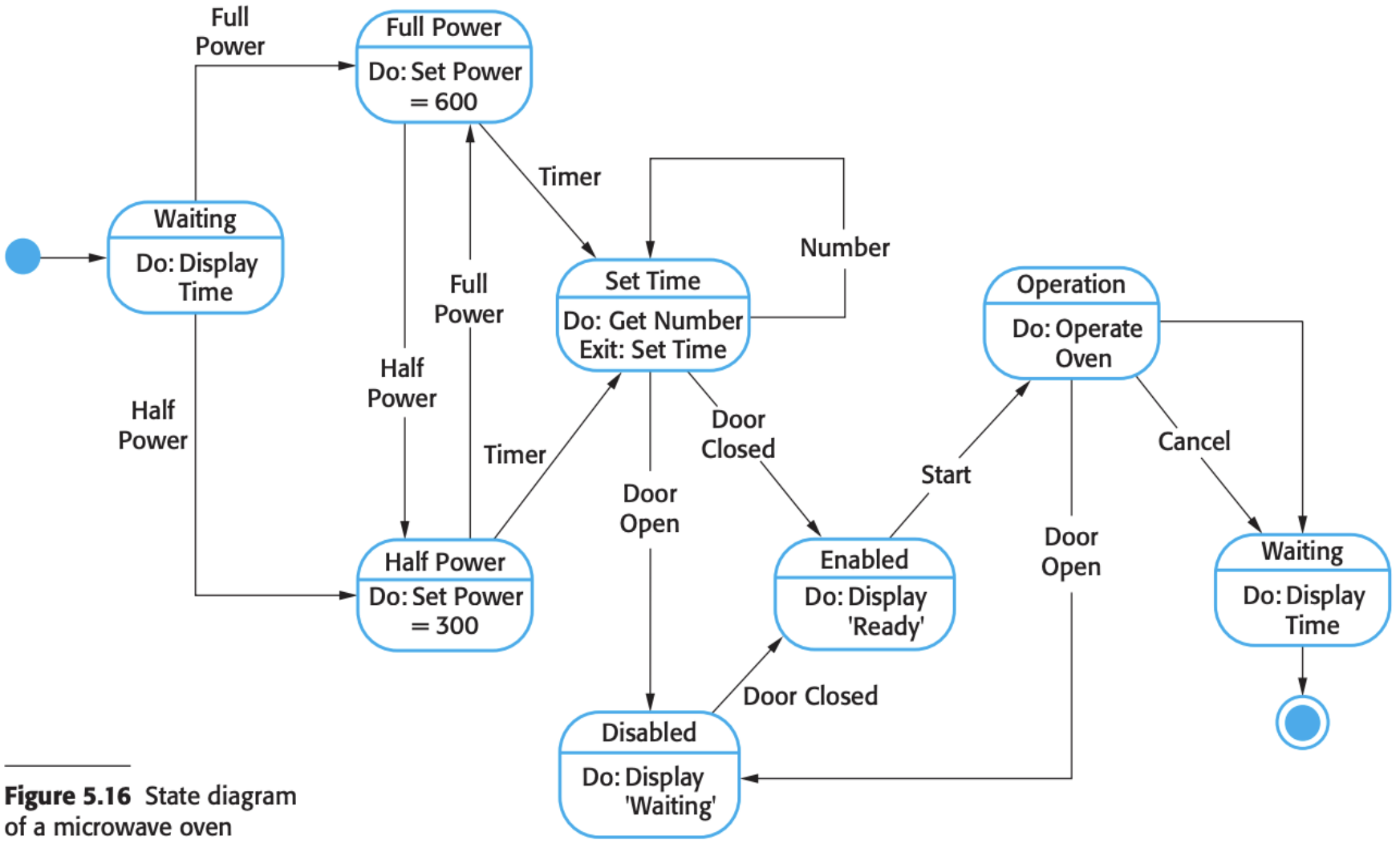


Figure 5.16 State diagram of a microwave oven

Model-driven Engineering: Model → A model could show how different parts of a program interact, how data flows, or how users will interact with the system.

Model-driven engineering (MDE) is a software development approach where **models** are the main focus, instead of writing code. Instead of directly writing programming code, developers create detailed visual or formal representations (models) of the system, such as diagrams or specifications, to design its structure and behavior. The system's actual program is then automatically generated from these models. This lets engineers work at a higher level, without worrying about programming details or hardware specifics. Specialized tools (e.g., Eclipse Modeling Framework, IBM Rational Rhapsody) generate code from Platform-Independent Models (PIM) or Platform-Specific Models (PSM) with little manual effort from developers.

Arguments For and Against MDE

Module is different parts of a software but model is a visual representation of those parts

For MDE: / Advantages

- Focuses on the overall system instead of small details.
- Reduces mistakes and speeds up development.
- Models can be reused for different platforms.
- Easy to update for new technologies with a translator.

Against MDE: / Disadvantages

- Models don't always match real-world needs.
- Sometimes, Simple designs and existing tools often work better than using MDE.
- Useful mainly for large, long-lasting systems.
- Bigger problems like security and testing are harder to solve.

a) Model-driven Architecture:

- Model-Driven Architecture (MDA) is a method of designing software that focuses on creating **models** to describe the system.
- It uses a specific set of UML diagrams to create models at different levels of detail.
- The idea is that you start with a high-level, platform-independent model, and from that, you can automatically generate a working program without needing to code it manually.

  Platform-independent model:
  - Not specifying database is MySQL or MongoDB.
  - Not specifying UI is for web, mobile, or desktop.
  - Not specifying prog. language is Python or java

a) Model-driven Architecture:

MDA involves three types of models:

- **Computation Independent Model (CIM):** Focuses on the problem and main concepts of the system (not technical details).
    - Example: In a hospital system, a CIM might show "patients" and "doctors" without worrying about how the system handles them.
- **Platform Independent Model (PIM):** Describes how the system works, but without mentioning the technology.
    - Example: It might show how users log in and manage data but doesn't mention the programming language or database.
- **Platform Specific Model (PSM):** Adjusts the platform-independent model to fit a specific technology.
    - Example: One PSM might be for a web app using MySQL, while another is for a mobile app using Firebase.

MDA Example: Online Bookstore System

- **Computation Independent Model (CIM):** Focuses on the business logic, not the system details.
    - Example: A *domain model* that shows **Customers**, **Books**, **Orders**, and their relationships.
- **Platform Independent Model (PIM):** Describes how the system works, but without mentioning the technology.
    - Example: A UML class diagram showing **Customer** with attributes like name and email, **Book** with attributes like title and price, and how an **Order** connects them.
- **Platform Specific Model (PSM):** Adds technical details for a specific platform.
    - Example:
        - For a web-based system: Specifies that the database will use MySQL, and the frontend is developed in React.
        - For a mobile app: Specifies that it will run on Android and use SQLite for data storage.

Working Process:

You start with a general model (CIM), refine it into a system model (PIM), and finally adapt it to the platform you'll use (PSM).
With MDA tools, the platform-independent model (PIM) could be automatically turned into working code, like creating database schemas or React components.

a) Model-driven Architecture:

MDA Transformations:

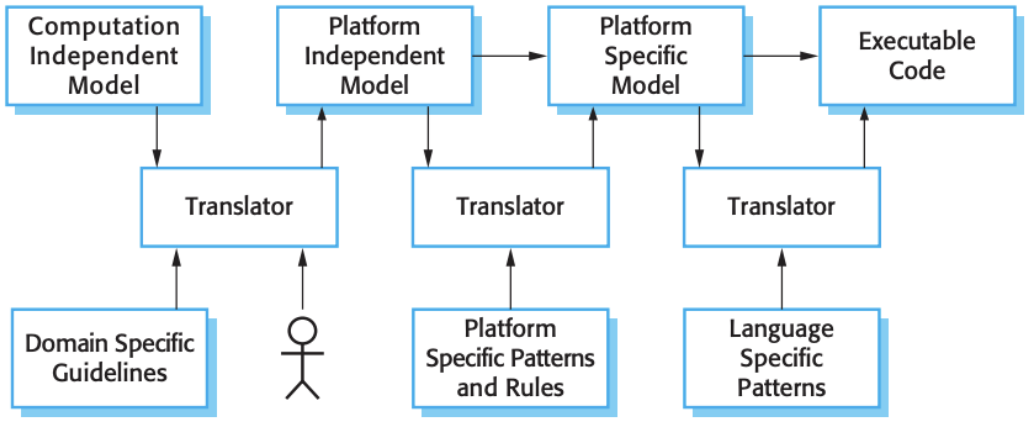


Figure 5.19 MDA transformations

- **Computation Independent Model (CIM):** Focuses on the problem and main concepts of the system (not technical details).
- **Platform Independent Model (PIM):** Describes how the system works, but without mentioning the technology.
- **Platform Specific Model (PSM):** Adjusts the platform-independent model to fit a specific technology.
- **Executable Code:** The PSM is further transformed into executable code, which is the actual software you can run on a computer or device.

The image also shows **translators** and **guidelines**:

- **Translator**: Tools or steps that convert one model into the next model (CIM to PIM, PIM to PSM).
- **Domain Specific Guidelines**: These are special instructions for creating models for a specific field or domain.
- **Platform Specific Patterns and Rules**: Guidelines for creating models specific to certain platforms (e.g., web platforms, mobile platforms).
- **Language Specific Patterns**: Patterns related to the programming languages used to write the executable code.

a) Model-driven Architecture:

Transformations of PIM into platform specific models

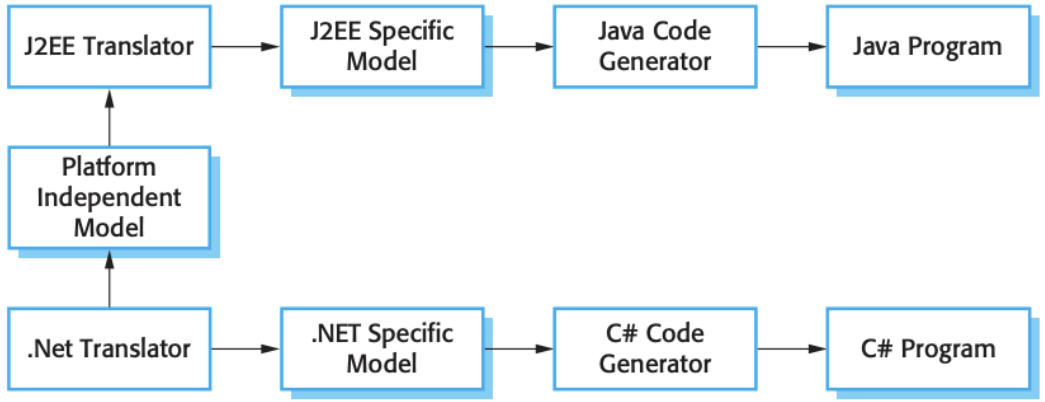


Figure 5.20 Multiple platform-specific models

This figure shows how the Platform Independent Model (PIM) is transformed into platform-specific models and then into actual code for different platforms (Java and .NET). Here's a simple breakdown of the process:

- **PIM:** The **Platform Independent Model (PIM)** is a general version of the system that doesn't depend on any specific technology or platform, like Java or .NET. It describes the system in broad terms, focusing only on what the system does, not how it will be built or run on a particular platform.

- **.NET Translator**:
  A tool or process that converts the PIM into a **.NET-specific model**. This model is tailored to work on the .NET platform.
- **J2EE Translator**:
  Similarly, this tool converts the PIM into a **J2EE-specific model** (specific to Java platforms).
- **Java Code Generator**:
  This generates **Java code** from the **J2EE specific model**, resulting in a **Java program** that can run on a Java platform.
- **C# Code Generator**:
  This generates **C# code** from the **.NET specific model**, which produces a **C# program** that can run on the .NET platform.

b) Executable UML (xUML)

- xUML is a special version of UML (Unified Modeling Language) that allows you to create models that can be automatically turned into working code. It's part of a larger idea called model-driven engineering, which aims to generate code directly from models. To make this possible, the models need to be precise and have clear rules for how things work.

- In simple terms, xUML focuses on three main types of models:
    - Domain Models: These models show the main elements of the system (like objects and how they relate to each other). They use class diagrams to describe this.
    - Class Models: These models define the classes in the system, along with their attributes (like properties) and operations (like methods).
    - State Models: These models show how each class changes over time using state diagrams. They describe the life cycle of an object.

- To describe how the system behaves, you can use either OCL (Object Constraint Language) or UML's action language. The action language is a high-level programming language that lets you specify what actions the system should perform with objects and their attributes.