SOFTWARE ENGINEERING

Module – 6 (OBJECT MODELLING USING UML)

Object-Oriented Development: An Overview

- **Object-oriented software development** has become extremely popular in recent years across both **industry** and **academia**.
- Although the **object technology** began in the 1980s, it gained strong momentum in the 1990s and is now **well-developed and widely adopted**.

🔹 Programming Language ≠ Design Skill

- Learning object-oriented programming languages like **Java or C++** is **not enough** to create good software.
- One must also learn **object-oriented design (OOD)**.
- A **good design** makes implementation easier and more efficient.
- Nowadays, **CASE tools** (Computer-Aided Software Engineering) can even **automatically generate code from the design**.

🔹 Importance of Modeling

- To build an effective object-oriented system, you must create **models**.
- Two types of models are used:
  - **Analysis Model** – represents the **problem**.
  - **Design Model** – represents the **solution** (how the software will work).

🔸 Key Difference:

Model Type	Description
Analysis Model	Represents what the problem is
Design Model	Represents how the solution will be built

🔹 Modeling Language vs. Design Process

Concept	Description
Modeling Language	A set of notations and symbols used to visually represent models (e.g., UML).
Design Process	A step-by-step approach (methodology) to convert problem description into a design.

🔷 Unified Modeling Language (UML)

- UML is the **standard modeling language** used to represent object-oriented systems.
- UML was **standardized by ISO** and has become the **dominant modeling tool** in software engineering.

🎯 Learning Objectives of This Chapter

- By the end of this chapter, you'll understand:
  - ✅ **Basic object-oriented concepts**
  - ✅ **Modeling using UML**
  - ✅ **Requirements modeling** using **Use Case Diagrams**
  - ✅ **Structure modeling** using **Class and Object Diagrams**
  - ✅ **Behavior modeling** using:
    - ➢ **Sequence Diagrams**
    - ➢ **Collaboration Diagrams**
    - ➢ **State Machine Diagrams**

6.1 Basic Object-Orientation Concepts

The principles of object-orientation have been founded on a few simple concepts. Some of these concepts are pictorially shown in Figure 6.1.

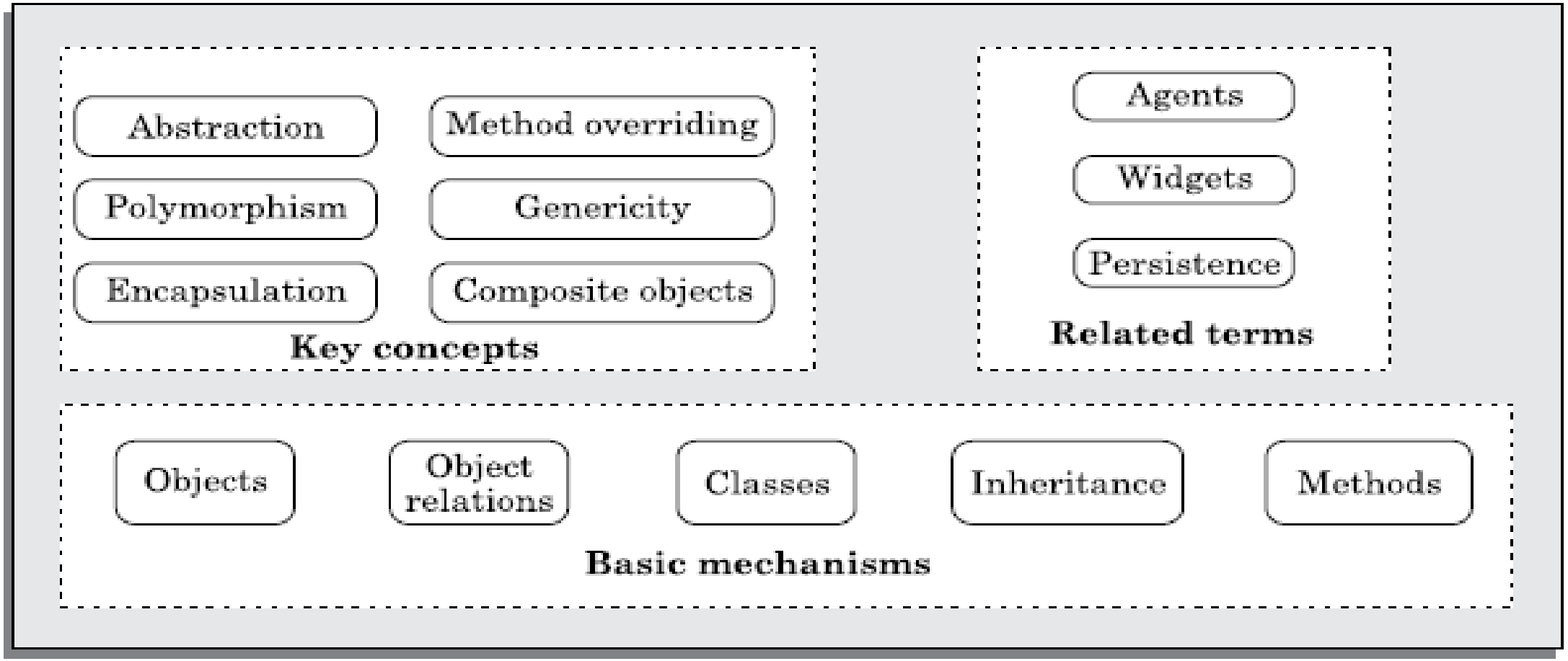


FIGURE 6.1 Important concepts used in the object-oriented approach.

Object-oriented software design revolves around **modelling real-world systems using objects**. Let's break down the core concepts:

🔷 1. Objects

- An **object** is a real-world or conceptual entity with:
  - **Attributes (data)**
  - **Methods (operations/functions)**

✅ Key Properties:

- Mimics real-world interactions (e.g., library book issue).
- Supports **data hiding** (private data accessed only through methods).
- Promotes **high cohesion** and **low coupling** – important for maintainable design.

📌 Example: libraryMember object

- **Attributes**: name, phone number, expiry date, books issued, etc.
- **Methods**: issueBook(), returnBook(), getDetails(), etc.

Each object essentially consists of some data that is private to the object and a set of functions (termed as *operations* or *methods*) that operate on those data. This aspect has pictorially been illustrated in Figure 7.2.

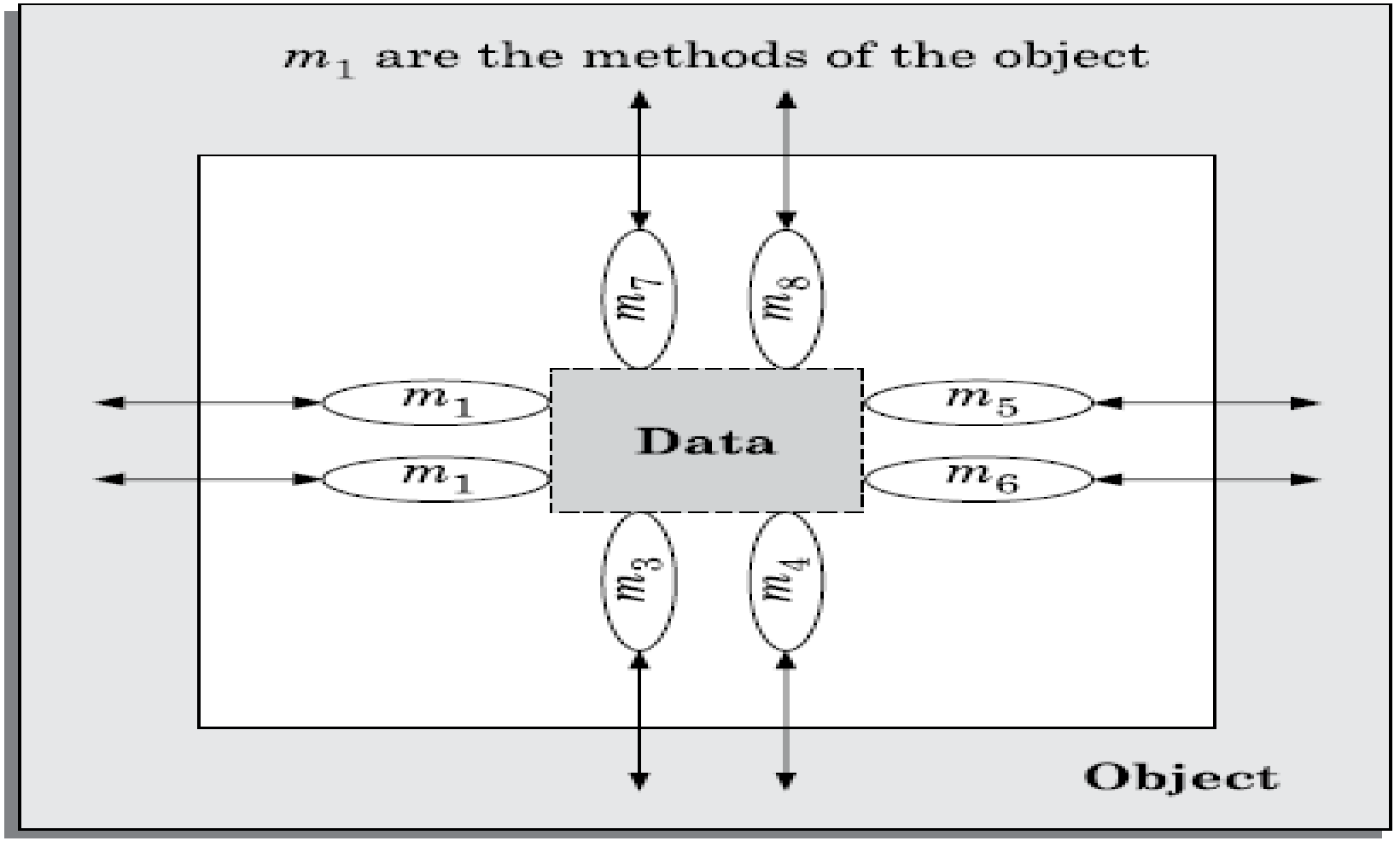


FIGURE 6.2 A model of an object

🔹 2. Classes

- A **class** is a blueprint/template for objects.
- It groups **similar objects** sharing the same attributes and methods.

✅ Example:

Class LibraryMember defines:

- Attributes: name, membership number
- Methods: issueBook(), returnBook()

🧩 Class vs Abstract Data Type (ADT):

- A class **is an ADT** because:
  - It **hides internal data** via methods (abstraction).
  - It can be **instantiated into objects** (like int x creates a variable x).

But:

Not all ADTs are classes – to be a class, it must support inheritance, encapsulation, etc.

🔹 3. Methods vs Operations

Term	Description
Operation	A responsibility or task of a class.
Method	The implementation of an operation.

❖Sometimes, one operation can be implemented in **multiple ways** – this is called:

🔄 Method Overloading

Same method name, different parameters.

create()

create(int radius)

create(float x, float y, int radius)

🔹 4. Messages vs Methods

- In **Smalltalk**, objects communicated by **sending messages** to each other.
  - This promoted **loose coupling**.
- In **C++ and Java**, the message-passing idea evolved into **method invocation** (like calling a function), which was more intuitive for traditional programmers.

🧠 Key Takeaways:

- Object-oriented design models the system as a **collection of objects**.
- Each **object encapsulates** data and behavior.
- A **class** is the definition or **template** from which objects are created.
- **Encapsulation**, **abstraction**, **modularity**, and **reusability** are built into the model.
- **UML** (Unified Modeling Language) is used to visually represent these concepts in software design.

7.1.2 Class Relationships

Classes in a program can be related to each other in the following four ways:

- Inheritance
- Association and Link
- Aggregation and Composition
- Dependency

🔶 1. Inheritance ("is-a" relationship)

- One class (**subclass**) **inherits** the attributes and methods of another class (**superclass**).
- Promotes **code reuse** and **conceptual clarity**.
- Enables a **class hierarchy**.

An example of inheritance is shown in Figure 6.3.

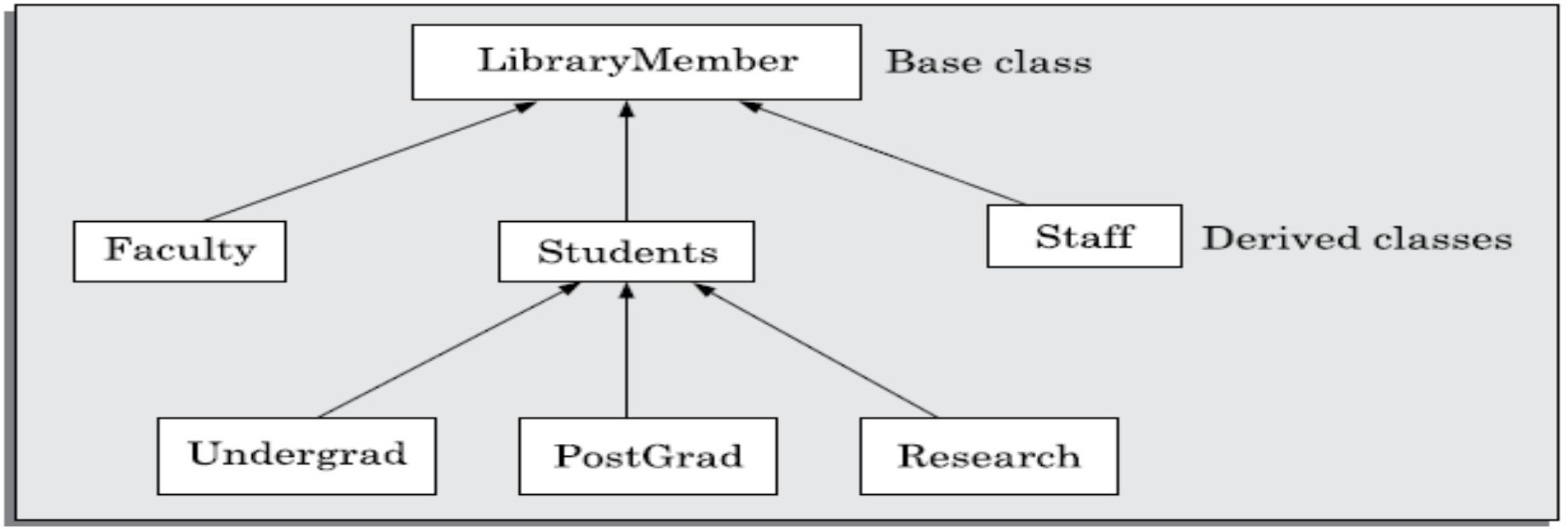


FIGURE 6.3 Library information system example.

- All subclasses inherit common features like memberID, issueBook() from LibraryMember.
- They can also **override** inherited methods (e.g., issueBook() for different durations).

🔄 **Method Overriding:**

Redefining a method from the base class in a derived class.

📌 Multiple Inheritance:

- A class inherits from **more than one superclass**.
- Example: Research inherits from both Student and Staff.

In Figure 6.4, we have shown the class Research to be derived from both the Student and Staff classes by drawing inheritance arrows to both the parent classes of the Research class.

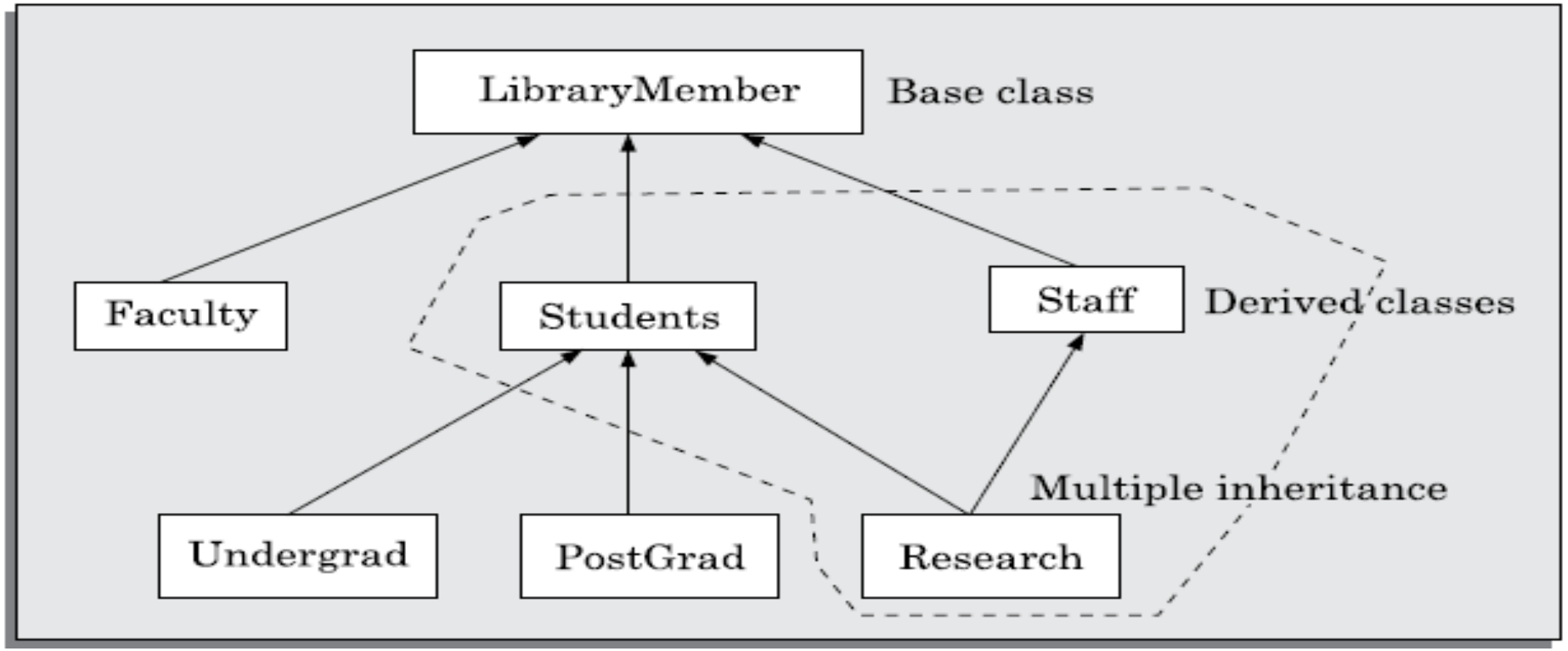


FIGURE 6.4 An example of multiple inheritance.

🔶 2. Association ("uses-a" relationship)

- Represents a **logical connection** between two or more classes.
- **Objects of one class know the identity** of objects from another class.
- Allows **method invocation** across associated classes.

📘 Example:

- A Student registers for an ElectiveSubject.
- A LibraryMember borrows a Book.

🧠 Key Points:

- **Association is static** (between classes).
- **Links** (between objects) are **dynamic** (can form and dissolve during runtime).
- Can be:
  - **Binary** (between two classes)
  - **Ternary** (among three classes: e.g., Person books Ticket for a Show)
  - **Unary** (recursive, e.g., Student has a friend who is also a Student)

We have graphically shown this association between Student class and Elective Subject in Figure 6.5(a). ternary association relationship has been represented in Figure 6.5(b). an association named friendship exists between pairs of objects of the Student class. This has been represented in Figure 6.5(c).

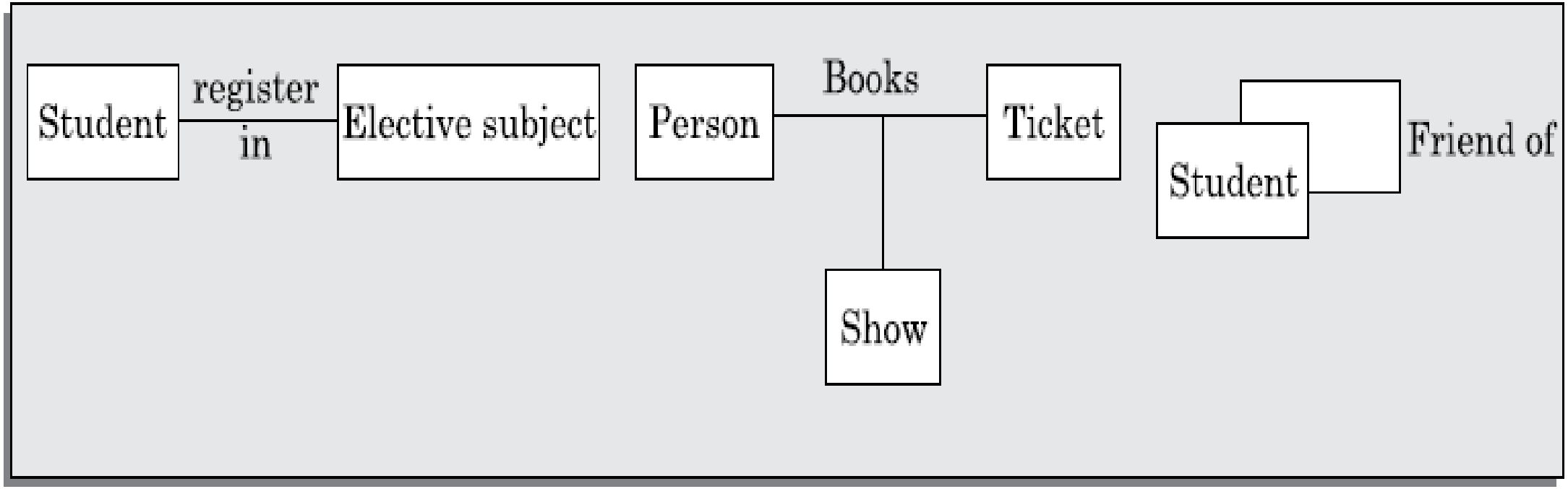


FIGURE 6.5 Example of (a) binary (b) ternary (c) unary association.

🔶 3. Aggregation and Composition ("has-a" relationship)

- Reflects **part-whole** hierarchy.
- A **composite object** contains other objects.
- Also known as the **whole-part** or **containment** relationship.

✅ Example:

- A Book has 1 to 10 Chapter objects.
- Represented using **multiplicity**: (1) Book → (1..10) Chapters

📌 Difference:

- **Aggregation**: weaker; contained object can exist independently.
- **Composition**: stronger; contained object cannot exist independently.

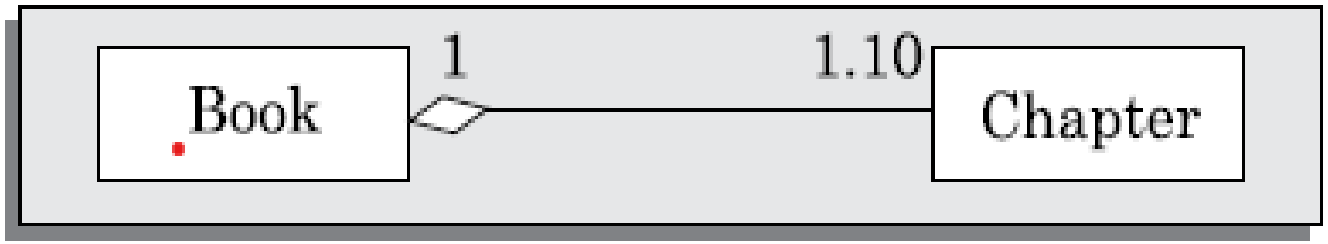


FIGURE 6.6 Example of aggregation relationship.

🔶 4. Dependency

Class **A depends on class B** if:

- ➢ It **uses** class B in its methods (e.g., as a parameter or local variable).
- ➢ It **implements** an interface that B provides.

- Dependencies among classes may arise due to various causes. Two important reasons for dependency to exist between two classes are the following:
  - A method of a class takes an object of another class as an argument. Suppose, a method of an object of class C1 takes an object of class C2 as argument. In this case, class C1 is said to be dependent on class C2, as any change to class C2 would require a corresponding change to be made to class C1.
  - A class implements an interface class (as in Java). In this case, dependency arises due to the following reason. If some properties of the interface class are changed, then a change becomes necessary to the class implementing the interface class as well.
  - A class has an object of another class as its local variable.

🔶 5. Abstract Class

- A class that **cannot be instantiated**.
- Designed to be **extended** by other classes.
- Provides a **common interface or base** for related classes.

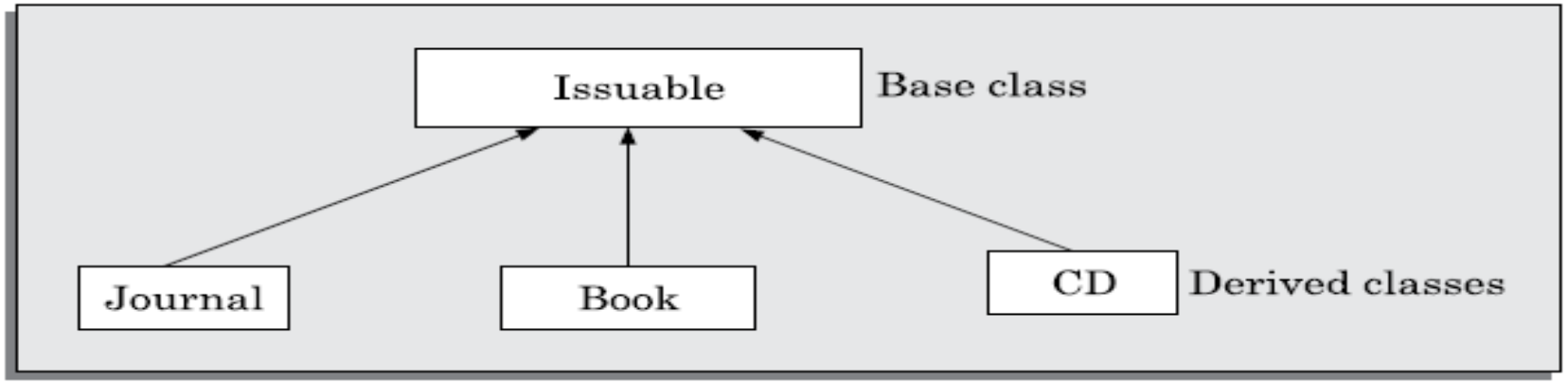


FIGURE 6.7 An example of an abstract class.

- Issuable might define a generic issue() method.
- Concrete classes override issue() to implement their own behavior.

🧠 Purpose:

- Promotes **code reuse**.
- Helps standardize method names and signatures.
- Encourages use of **polymorphism**.

Relationship	Description	Example
Inheritance	“is-a” relationship	Student is a LibraryMember
Association	“uses-a” (knows other’s ID/methods)	Student registers for Subject
Aggregation	“has-a” (part-whole, weaker)	Book has Chapters
Composition	Strong containment	Order has OrderItems
Dependency	Temporary usage/impact from changes	Method uses object of another class
Abstract Class	Base class not meant to be instantiated	Issuable base for Book, etc.

6.1.3 How to Identify Class Relationships?

When you're trying to **translate a real-world or problem description** into object-oriented code, a key step is to **identify classes and their relationships**.

🧠 General Strategy

- **Identify the classes**

  → Look for **nouns** in the problem description. These usually represent **objects or classes**.

- **Identify relationships between classes**

  → Look for **specific key phrases** or **verbs/prepositions** that indicate the **type of relationship**.

🔍 Keywords and Their Indications

🔑 Key Phrase	🔗 Indicates	📘 Example
“B is a permanent part of A”	Composition	A Car has an Engine
“A is made up of Bs”	Composition	A Book is made up of Chapters
“A is a permanent collection of Bs”	Composition	A Computer is a permanent collection of Components
“B is a part of A”	Aggregation	A Team has Players
“A contains B”	Aggregation	A Classroom contains Students
“A is a collection of Bs”	Aggregation	A Department is a collection of Professors
“A is a kind of B”	Inheritance	A Dog is a kind of Animal
“A is a specialisation of B”	Inheritance	Postgrad is a specialisation of Student
“A behaves like B”	Inheritance	ElectricCar behaves like Car
“A delegates to B”	Association	A Manager delegates to an Employee
“A needs help from B”	Association	A Doctor needs help from a Nurse
“A collaborates with B”	Association	A Customer collaborates with a Vendor
Other verbs: employs, credits, precedes…	Association (general)	A Project employs Engineers

Use this rule of thumb

- If you read:
  "**A has a B**" → Think **Aggregation** or **Composition**
  "**A is a B**" → Think **Inheritance**
  "**A uses B**" → Think **Association**

✅ **Example Breakdown**

Problem Description:

"A teacher teaches multiple students. Each student writes exams and receives grades. A student is a person. An exam has multiple questions. Each question contains options."

👉 **Identified Classes:**

- Teacher, Student, Person, Exam, Question, Option, Grade

👉 **Identified Relationships:**

Relationship Description	Relationship Type
Student is a Person	Inheritance
Teacher teaches Student	Association
Student writes Exam	Association
Exam has Questions	Aggregation
Question contains Options	Composition
Student receives Grade	Association

6.1.4 Other Key Concepts in Object-Oriented Programming

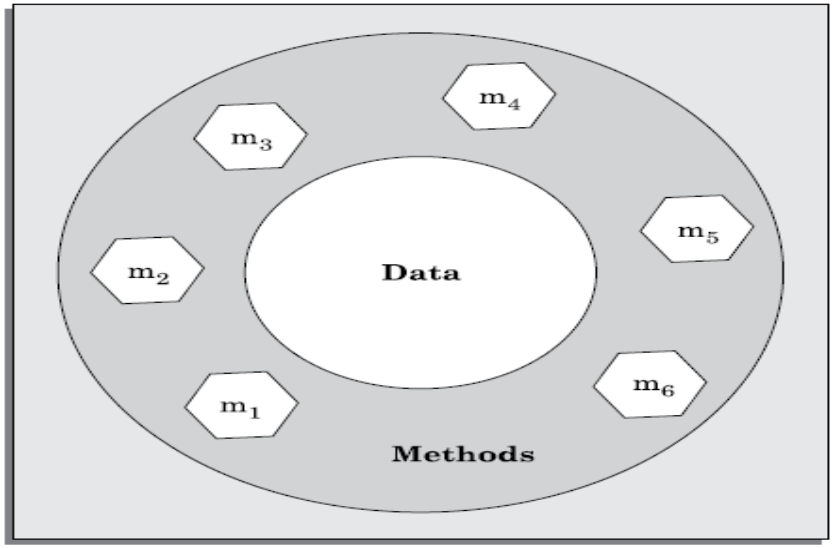
1. Abstraction

- **Definition:** The act of **selectively focusing** on relevant details and ignoring irrelevant ones.
- **Purpose:** Simplifies system design and enhances **understanding, reuse, and maintainability**.

🔹 Two Types of Abstraction:

Type	Description	Example
Feature Abstraction	A base class is a simplified version of its subclasses.	LibraryMember class abstracts common features of Student, Faculty, etc.
Data Abstraction	An object hides how it stores and manages its internal data.	A Stack object provides push() and pop(), but hides whether it uses an array or linked list.

- **2. Encapsulation**
- **Definition:** Hiding the internal data of an object and **only allowing access through methods**.
- **Illustration:**
  This concept is schematically shown in **Figure 6.8.**



- (Imagine: data inside a locked box, accessed only through method "keys")
- 🔹 **Benefits:**
- ✅ **Protection from unauthorized access**
- ✅ **Data hiding** → promotes modularity and maintainability
- ✅ **Weak coupling** → better program structure and reusability

3. Polymorphism

- **Definition:** Polymorphism means “many forms”. Just like diamond, graphite, and coal are all forms of carbon with different behavior, in OOP, polymorphism allows one interface (method) to behave differently based on the object type.

🔹 Two Types:

Type	Also Called	Binding Time	Description
Static Polymorphism	Method Overloading	Compile-time	Same method name, different parameter lists.
Dynamic Polymorphism	Method Overriding	Run-time	Base class method is redefined in derived class; actual method chosen at runtime.

1. Static Polymorphism (Compile-time / Early Binding)

- 🔸 **Also known as:**
  - **Method Overloading**
  - **Static Binding**

🔸 **How it works:**

- Multiple methods in a class share the **same name** but have **different parameter types or counts**.

🧠 **Binding time:**

- Resolved at **compile-time** based on method signature.

- ✅ **Example:**

```
class Circle {
  int create();                    // No parameters
  int create(int radius);          // One integer parameter
  int create(float x, float y, int r);  // Three parameters
};
```

When you call:

```
Circle c;
c.create();           // Calls the first method
c.create(10);         // Calls the second
c.create(1.5, 2.5, 10);  // Calls the third
```

The compiler chooses the correct method based on the parameters.

2. Dynamic Polymorphism (Run-time / Late Binding)

🔸 Also known as:

- **Method Overriding**
- **Dynamic Binding**

🔸 How it works:

➢ A **derived class overrides** a method from its **base class**.

The **actual method called is determined at runtime**, based on the object type.

🧠 Binding time:

Resolved at **runtime**, not compile-time.

we have defined a class hierarchy of different geometric shapes for a graphical drawing package as shown in Figure 6.9.

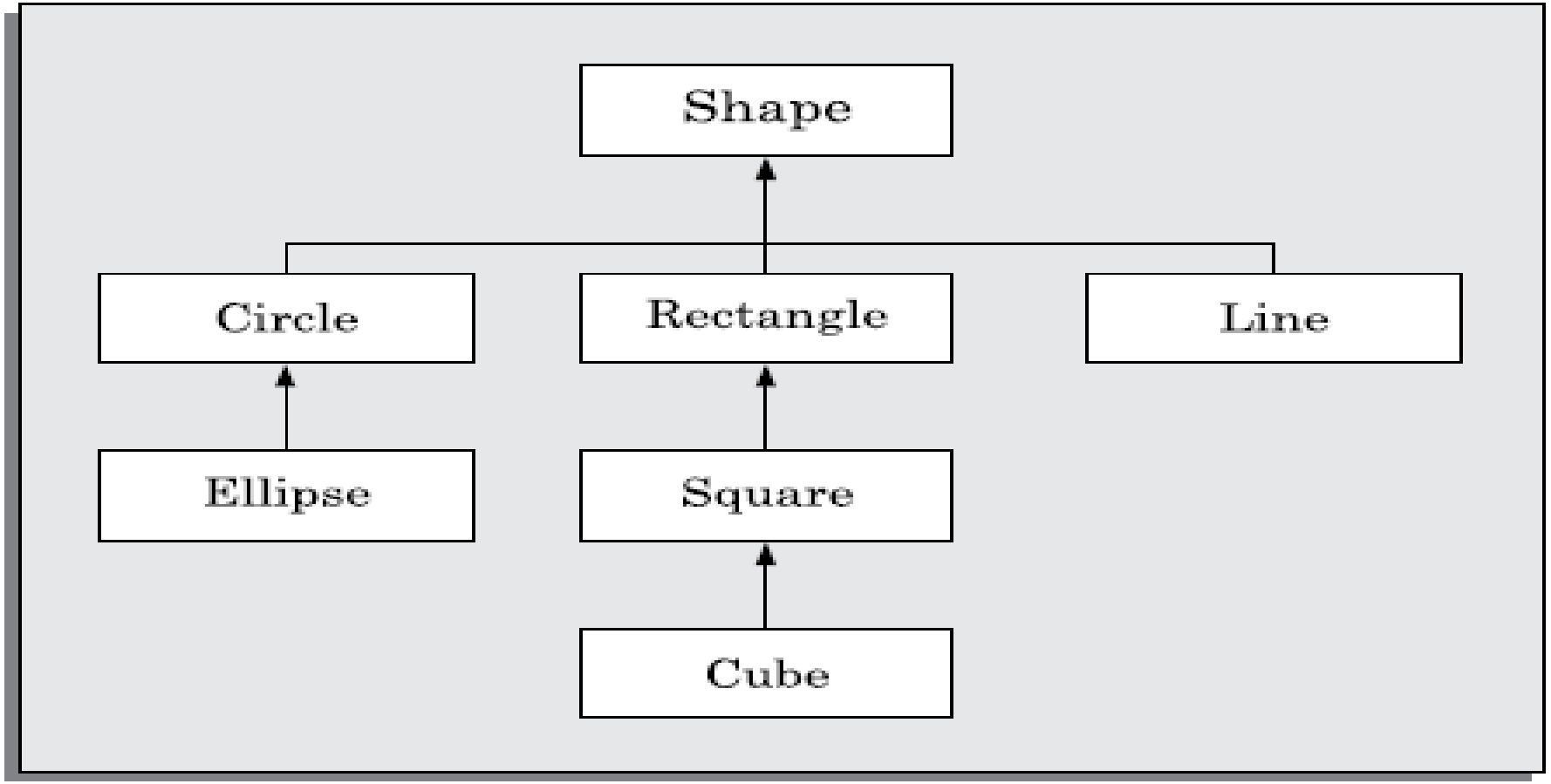


FIGURE 6.9 Class hierarchy of geometric objects.

✅ Example:

Shape Hierarchy:

```
class Shape {
public:
  virtual void draw(); // Base method
};

class Circle : public Shape {
public:
  void draw() override; // Circle's version
};

class Square : public Shape {
public:
  void draw() override; // Square's version
};
```

Polymorphic usage:

```
Shape* s;
s = new Circle();
s->draw();  // Will call Circle’s draw()

s = new Square();
s->draw();  // Will call Square’s draw()
```

💡 **Even though s is a pointer to Shape, the appropriate draw() method is called depending on the actual object type.**

🔑 Dynamic Binding Depends on:

Method Overriding

A derived class provides its own version of a method from the base class.

Liskov Substitution Principle

You can assign an object of a **derived class** to a **base class** variable (but not the reverse).

🆚 Traditional vs Object-Oriented (Using Dynamic Polymorphism)

Traditional Code	OOP with Polymorphism
Uses many if-else or switch statements	Uses common interface like draw()
Hard to maintain when adding new types	Just add a new class (e.g., Ellipse)
Less reusable and more complex	Cleaner, scalable, and reusable

Type	Binding Time	Concept	Feature
Static Polymorphism	Compile-time	Method Overloading	Same method name, different parameters
Dynamic Polymorphism	Runtime	Method Overriding	Same method name & parameters, different definitions in subclasses

4. Genericity (Generics)

- **Definition:** Writing **type-independent** classes or methods.
- **Use Case:** Creating reusable code components like a generic Stack<T> that works for any data type.

🔹 **Example:**

```
class Stack<T> {
   void push(T item) { ... }
   T pop() { ... }
}
```

You can instantiate:

```
Stack<Integer> intStack = new
Stack<>();
Stack<String> strStack = new
Stack<>();
```

Concept	Key Idea	Key Benefit
Abstraction	Focus on relevant details	Simplicity & clarity
Encapsulation	Hide internal data	Data protection & modularity
Polymorphism	One interface, many implementations	Flexible and reusable code
Genericity	Type-parameterized classes	Code reusability across types

6.1.5 Related Technical Terms in Object-Orientation

Persistence

- Normally, objects exist only during the execution of a program.
- **Persistent objects** are stored permanently (e.g., in databases or files) so they can be used across multiple runs.
- **Use case**: Saving user data or application state between sessions.

Agents

- **Passive object**: Acts only when another object calls its method.
- **Agent (Active object)**: Reacts to events **autonomously**, without being explicitly invoked.
- **Use case**: An agent watching a database for inconsistent transactions and triggering alerts automatically.

Widgets

- A **widget** is a basic **GUI (Graphical User Interface)** element.
- Examples: buttons, text boxes, sliders, etc.
- Widgets hold properties like size, color, cursor, etc., and provide methods to perform actions like resize(), iconify(), destroy(), etc.
- They form the basis of **component-based GUI development**.

🔸 6.1.6 Advantages and Disadvantages of Object-Oriented Development (OOD)

✅ Advantages of OOD

1.Code & Design Reuse

- Through **class libraries** and **inheritance**, saving development time.

2.Improved Productivity

- Easier to manage complexity using **abstraction** and **decomposition**.
- Reusing tested code components enhances development speed.

3.Easier Testing and Maintenance

- Modular design allows individual testing and easier bug fixing.

4.Better Understandability

- Especially helpful for **large systems**; objects model real-world concepts better.

5.Long-term Cost Benefits

- Although initial projects may be costly, savings appear after reusing built libraries and gaining experience.
- **Cost reduction up to 20–50%** is possible after maturity.

❌ Disadvantages of OOD

1.Run-Time Overhead

- Due to features like **inheritance**, **polymorphism**, and **data hiding**, object-oriented programs can run **slower** than procedural ones.

2.Poor Spatial Locality of Data

- In OOD, related data may be spread across memory (unlike arrays in procedural programming).
- This leads to **more cache misses** and **slower memory access**, which increases run time.

3.Overhead in Small Applications

- For **small embedded systems**, procedural approaches may be more efficient and simpler.

6.2 UNIFIED MODELLING LANGUAGE (UML)

✅ What is UML?

- **UML** (Unified Modelling Language) is **not a development method**, but a **graphical language** used to **document models** of object-oriented systems.
- It consists of **syntax** (symbols, diagrams) and **semantics** (their meanings).
- UML can be used with **any object-oriented methodology** (e.g., OMT, Booch, OOSE).
- UML provides a **common language** for analysis, design, and documentation, enabling reuse and clear communication across teams.

🔹 7.2.1 Origin of UML

In the **1980s–1990s**, multiple object-oriented methodologies existed, each with their own notations:

- **OMT** (Rumbaugh)
- **Booch's method**
- **OOSE** (Jacobson)
- **Odell**
- **Shlaer–Mellor**

➡️ These variations created **confusion** and hindered **design reuse**.

- UML was created to **standardize** all these notations and unify them into one language.
- It was developed by the **Object Management Group (OMG)** in **1997**.
- **ISO adopted UML as a standard (ISO 19805) in 2005**.

🧠 UML combines the strengths of earlier methods—mainly **OMT**, **Booch**, and **OOSE**.

The relative degrees of influence of various *object modeling techniques on UML is shown schematically in Figure 6.10.*

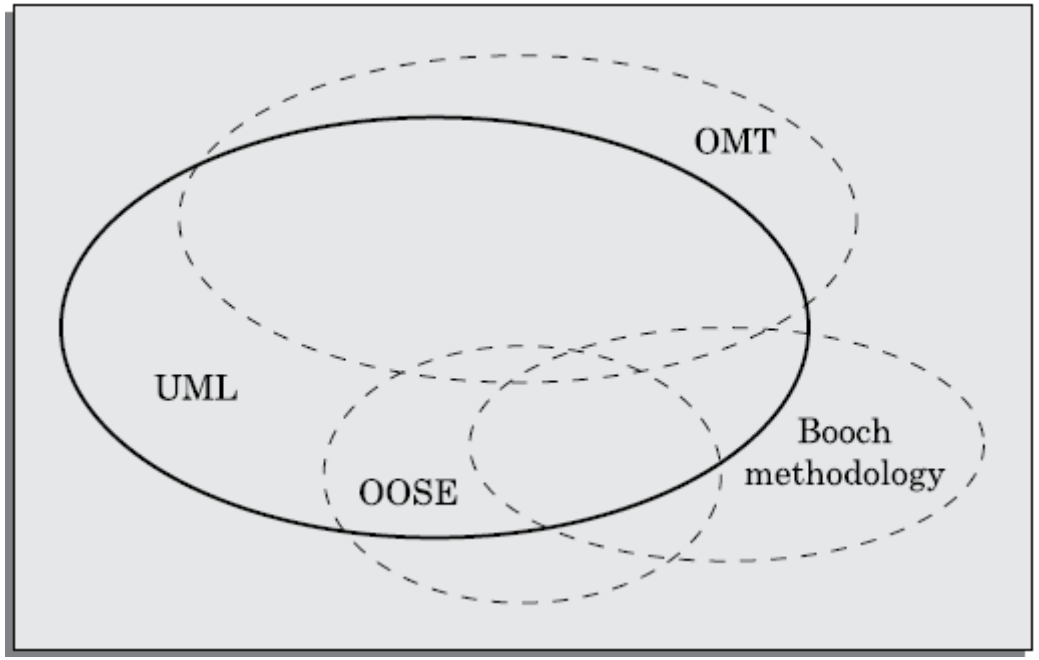


FIGURE 6.10 Schematic representation of the impact of different object modelling techniques on UML.

🔹 6.2.2 Evolution of UML

- Since **UML 1.0 (1997)**, it has evolved rapidly with feedback from academia and industry.
- A **major release**, **UML 2.0 (2007)**, included support for:
  - **Embedded** and **concurrent systems**
  - New features like **events**, **ports**, and **frames**

🌐 UML has gained wide acceptance, not only in software but also in **non-software domains** (e.g., automotive systems like "build-to-order" cars).

Almost every year several new releases (shown as UML 1.X in Figure 6.11) are being announced.

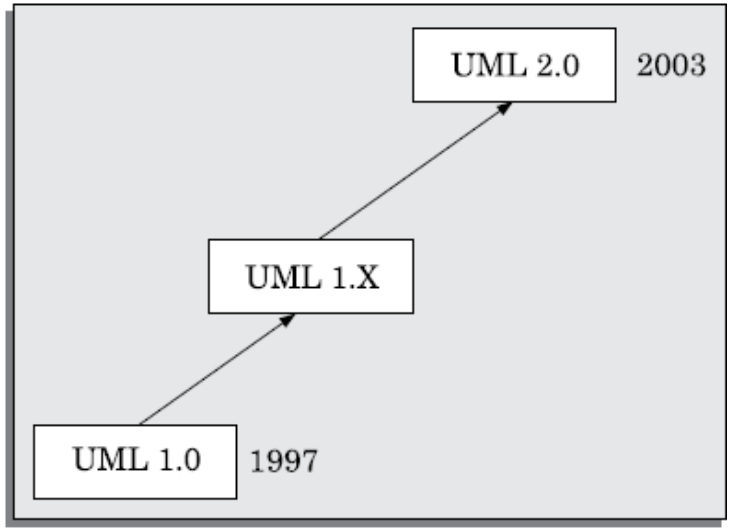


FIGURE 6.11 Evolution of UML

🔹 What is a Model?

- A **model** is a **simplified version of a real system** created by abstracting unnecessary details.
- It helps developers **manage complexity**, understand systems better, and solve problems effectively.

🔍 In software:

- **Analysis model**: Created early to understand and define system requirements.
- **Design model**: Created later by refining the analysis model with specific implementation details.

📐 **Model types depend on purpose**, like:

- **Analysis**
- **Specification**
- **Design**
- **Code generation**
- **Visualization**
- **Testing**

➡️ Always **mention the purpose** of the model being created, as different phases need different model details.

🔹 Tools for Drawing UML

✏️ UML diagrams can be hard to draw by hand, so **CASE tools** are commonly used, such as:

- ❖ **Rational Rose**
- ❖ **ArgoUML**
- ❖ **MagicDraw**
- ❖ **Draw.io**

💡 Many tools can:

- ➢ Help refine object models
- ➢ Auto-generate code templates (in Java, C++, C, etc.)

🔸 6.3 UML DIAGRAMS

✅ What Are UML Diagrams?

- UML provides **nine types of diagrams** that collectively help model **five different views** of a system.
- Just like a building can be viewed from various perspectives (electrical, plumbing, lighting), a software system too can be modeled from different perspectives.
- Modeling from **multiple views** improves **understanding** and **system design quality**.

🔹 Why Not One Diagram for Everything?

If we try to represent all perspectives in a **single diagram**, it becomes **as complex as the system itself**, defeating the purpose of modeling.

➡️ Hence, **separate views** are used to focus on specific concerns.

🔹 Five Views Supported by UML

View	Purpose
1. Users' View	Captures the functionalities provided by the system (black-box view).
2. Structural View	Models the static structure: classes, objects, and relationships.
3. Behavioural View	Captures dynamic behaviour: object interactions over time.
4. Implementation View	Shows components (GUI, database, middleware) and their connections.
5. Environmental View	Shows physical deployment of components onto hardware.

📌 The **Users' View is central**—all other views must **conform to it**, as it defines the system's purpose from a **user's perspective**.

🔍 1. Users' View

- Represents the **functional model** (i.e., the operations the system will offer).
- **Ignores** internal implementation, object structure, or runtime behaviour.
- Typically modeled using the **Use Case Diagram**.

Figure 6.12 shows the different views that the UML diagrams can help document.

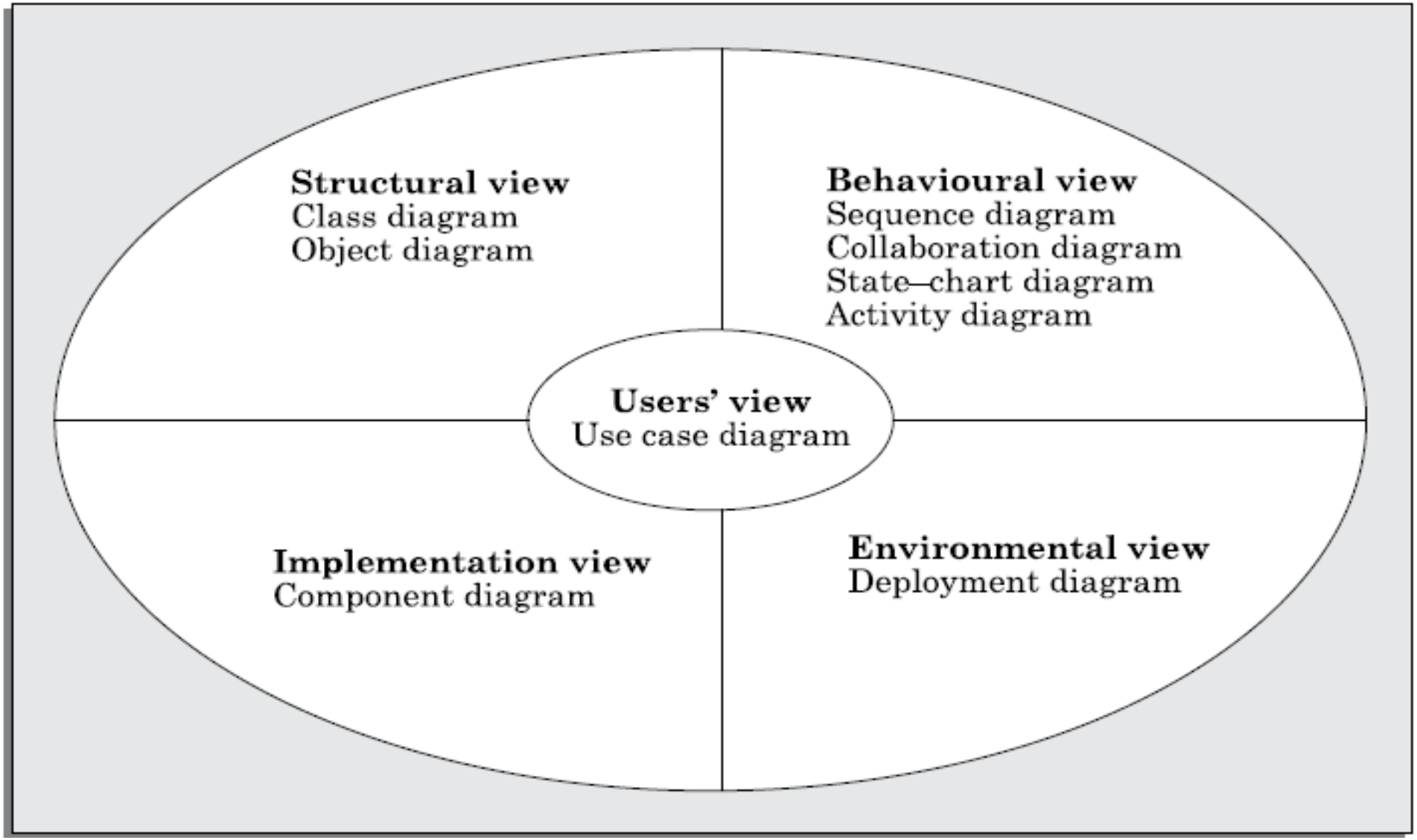


FIGURE 6.12 Different types of diagrams and views supported in UML.

🔍 2. Structural View

- Also called the **Static Model**.
  - Focuses on:
    - **Classes**
    - **Objects**
    - **Attributes**
    - **Methods**
    - **Relationships** (like association, inheritance)
- Modeled using:
    - **Class Diagrams**
    - **Object Diagrams**

🔍 3. Behavioural View

Represents how objects **interact dynamically** over time.

- Captures:
  - **Sequence of interactions**
  - **State changes**
  - **Messages passed between objects**

Modeled using:

- **Interaction Diagrams** (Sequence and Collaboration Diagrams)
- **State Chart Diagrams**
- **Activity Diagrams**

🔍 4. Implementation View

- Shows **system components** and their **dependencies**.
- Useful for understanding modularization and the structure of code.
- Modeled using:
  - **Component Diagrams**

🔍 5. Environmental View

- Describes how system components are **deployed onto hardware**.
- Important in **distributed systems** and **embedded applications**.
- Modeled using:
  - **Deployment Diagrams**

🔸 Do You Need All Diagrams for Every System?

❌ **No** — It depends on the **complexity and nature** of the system.

✳️ Example:

- A **simple web app** might only need:
  - **Use Case Diagram**
  - **Class Diagram**
  - **Sequence Diagram**
- A **real-time control system** might require:
  - **State Chart Diagrams**
  - **Deployment Diagrams**

🔸 7.4 USE CASE MODEL

✅ What is a Use Case Model?

- A **Use Case Model** consists of a set of **use cases** representing **what users can achieve** by interacting with the system.
- It answers the question:

  "What can each type of user do with the system?"

🔹 Examples of Use Cases (for a Library System):

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

Each use case represents a **user-goal-driven functional requirement** of the system.

🔹 Key Concepts:

Concept	Explanation
Use Case	A transaction or functionality from the user's perspective.
Actor	A role played by a user (e.g., librarian, member). Represented as a stick person.
Mainline Scenario	The normal or most frequent sequence of interactions in a use case.
Alternate Scenarios	Variations due to exceptional or alternative conditions.
Scenario	Any one possible flow (mainline or alternate) of user-system interaction.

✔️ **A use case is independent, but may have implementation-level dependencies on others due to shared resources or logic.**

🔹 Use Case Diagram Components:

Symbol/Term	Meaning
Ellipse	A single use case, labeled with its name
Stick Figure	An actor (user role or external system)
Rectangle	The system boundary (optional) enclosing all use cases
Line	Communication link between actor and use case

🎭 **Actor ≠ User**

An actor is a role, not necessarily a single person.

➡️ **A user may play multiple roles (e.g., librarian AND member).**

🔹 Stereotype in UML

- A **stereotype** adds **meaning or context** to a UML symbol using << >>.
  - Example: <<external system>> for an external actor.

📌 Purpose of stereotypes:

- **Avoids having too many symbols**.
- Adds **semantic meaning** using limited notation.
- Easier to **learn and use** UML with fewer base symbols.

EXAMPLE 6.2 The use case model for the Tic-tac-toe game software is shown in Figure 6.13. This software has only one use case, namely, “play move”. Note that we did not name the use case “get-user-move”, which would be inappropriate because this would represent the developer’s perspective of the use case. The use cases should be named from the users’ perspective.

🔹 Example: Tic-Tac-Toe Game Use Case

Use Case Diagram:

- Actor: Player
- Use Case: play move (not get-user-move, as use cases should be named from the **user's perspective**, not the developer’s)

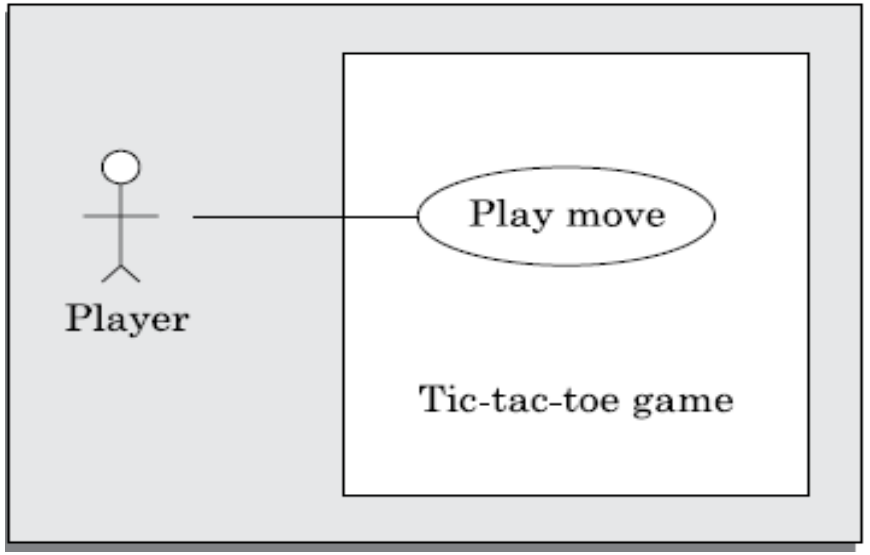


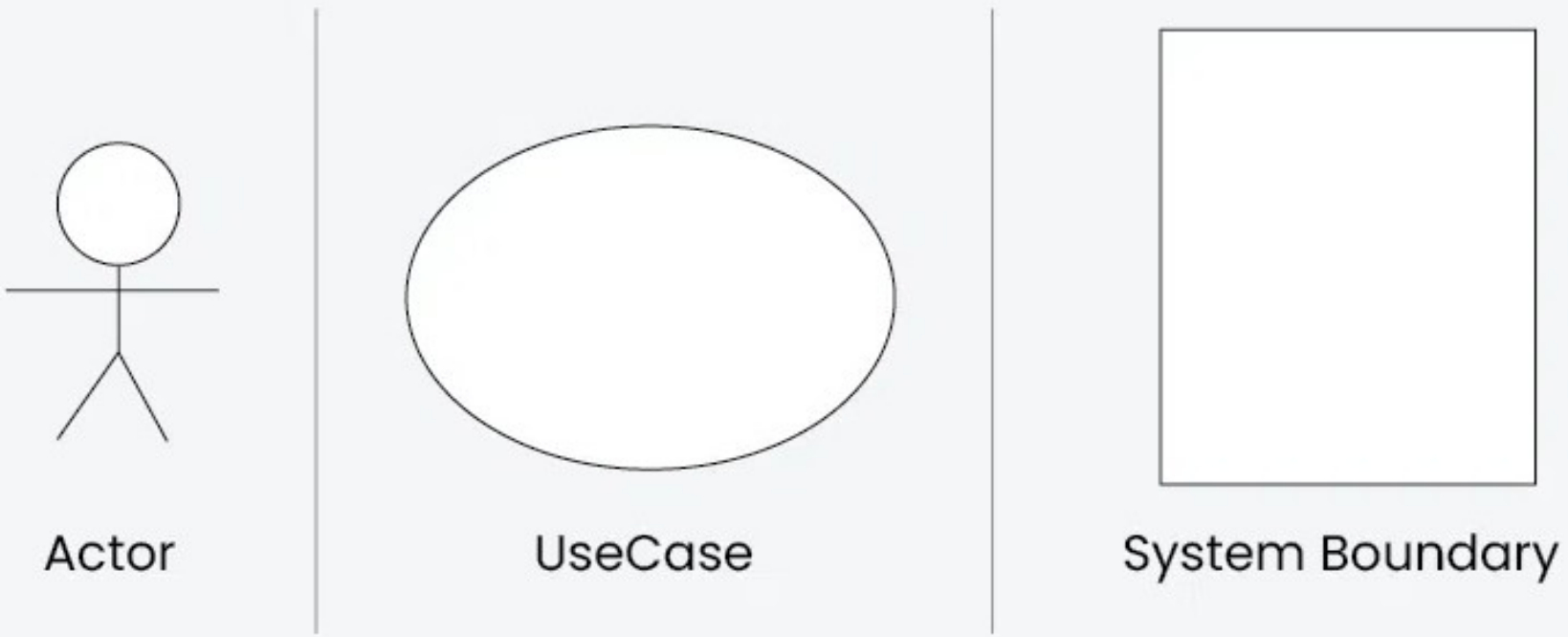
FIGURE 6.13 Use case model for Example 6.2

🔹 Textual Description of a Use Case

Since ellipses show **very limited info**, **every use case should be accompanied by text**, which includes:

Section	Description
Actors	Roles involved and their details
Pre-conditions	System state before use case starts
Post-conditions	System state after completion
Mainline Scenario	Normal interaction flow
Alternate Scenarios	Valid but different interaction flows
Exceptions	Errors and how they're handled
Non-functional requirements	Constraints like performance, platform, UI standards
Sample Dialogs	Sample user interactions
UI Requirements	Forms, screen layout, interaction styles
Document References	Any supporting documents

Use Case Diagram Notations



🔸 6.4.2 Why Develop the Use Case Diagram?

✅ Purpose and Benefits:

- **Use Case Diagrams** help represent **functional requirements** visually using *ellipses*.
- These diagrams form the **core of system modeling**, since all other UML models (like class, sequence, etc.) are derived from them.

✅ Importance of Actors (Stick Person Icons):

Actors help:

1. **Design security features** (e.g., role-based login).
2. **Design user interfaces** specific to different user roles.
3. **Prepare user-specific documentation** (like manuals).
4. **Identify use cases** by exploring user interactions.
5. **Understand system behavior** from each user's perspective.

🔸 6.4.3 How to Identify the Use Cases of a System?

✅ Two Key Techniques:

1. Based on SRS:

- Review the **Software Requirements Specification (SRS)**.
- High-level requirements typically **map directly to use cases**.

2. Actor-Based Method:

Identify all possible **actors** (user roles or systems).

Ask: **"What will each actor do with the system?"**

List all actions each actor initiates or participates in.

Example: *Library Automation System*

Actor	Related Use Cases
Member	Issue book, Return book, Renew book
Librarian	Add/Delete members, Add/Delete books
Accountant	Manage fees, Track expenses

🔸 6.4.4 Essential Use Case vs. Real Use Case

Type	Description
Essential Use Case	Defined during early analysis, independent of UI or technology. Stable and long-lasting.
Real Use Case	Created after design decisions, includes UI and platform details.

🎯 **In some organizations, the UI specs are fixed early, so the essential and real use cases may be the same.**

🔸 6.4.5 Factoring of Commonality among Use Cases

✅ Why Factor Use Cases?

1. **To simplify complex use cases**
   → Break into smaller, manageable parts.
2. **To reuse common functionality**
   → Avoid duplication of behavior (e.g., error checking).

⚠️ Don't over-factor unnecessarily. Only factor for clarity or reuse.

Use-case Relationships

- Relationship show association between actor and use case.
- They represent a generalization of actors.
- Extend between two use cases.
- Include between two use cases.
- They also represent a generalization of a use case.

🔹 UML Supports 3 Use Case Factoring Mechanisms:

1. 🧬 Generalization (Inheritance)

- Used when **one use case is a variation or extension** of another.
- The **child use case inherits** behavior from the **parent**.
- Similar to class inheritance.

📌 **Notation**: Solid line with hollow triangle pointing to the parent.

📖 Example:

- Update user profile ← Change email, Change password

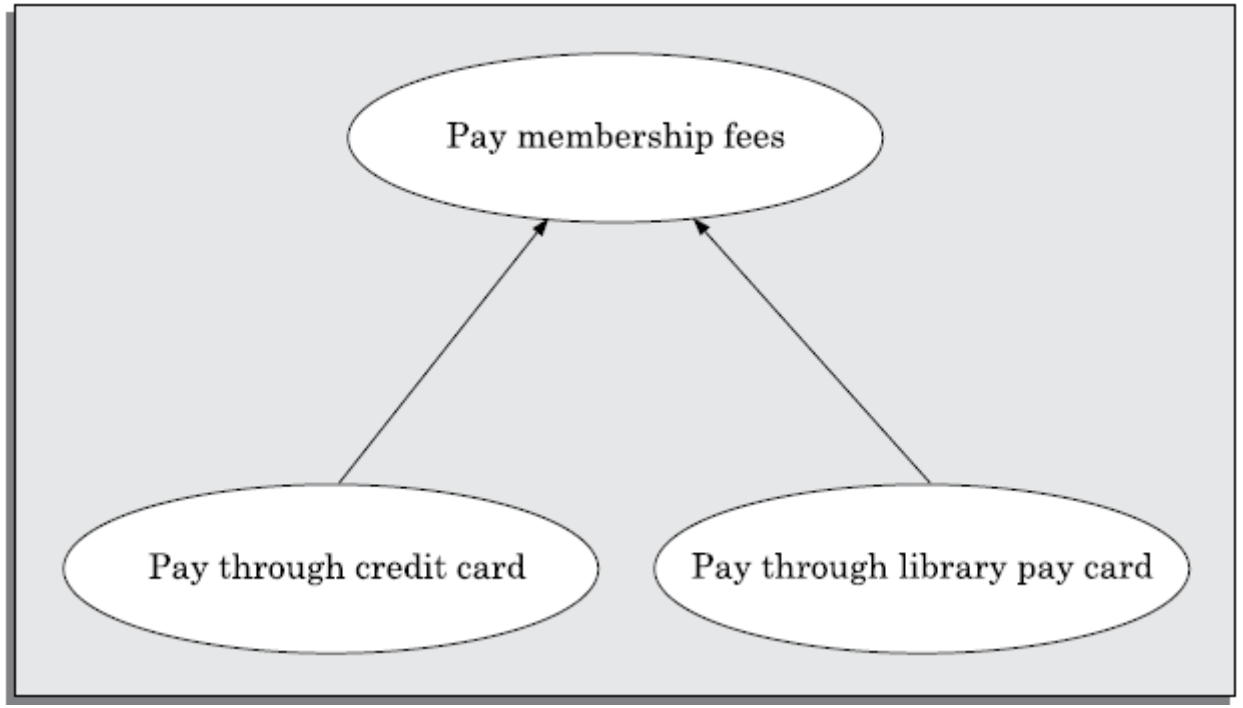


FIGURE 6.14
Representation of use case generalization.

2. 🔗 Include (Reuse Common Steps)

- Used to **factor out repeated behavior** shared across multiple use cases.
- The **base use case includes** the behavior of another.
- Always **executed** as part of the base use case.

📌 **Notation**: Dashed arrow labeled <<include>> pointing **towards** the included use case.

📖 Example:

- Issue book and Renew book both include Check reservation

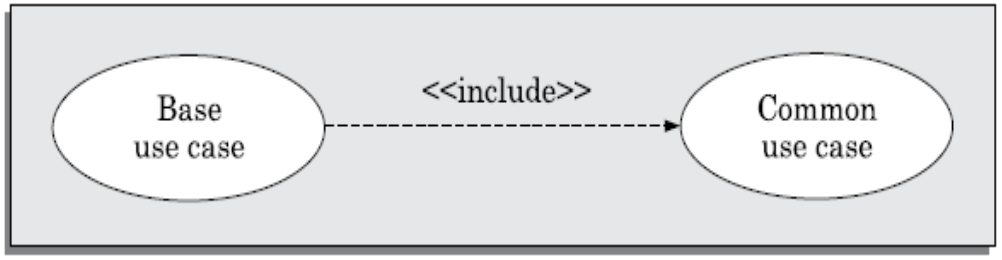


FIGURE 6.15 Representation of use case inclusion.

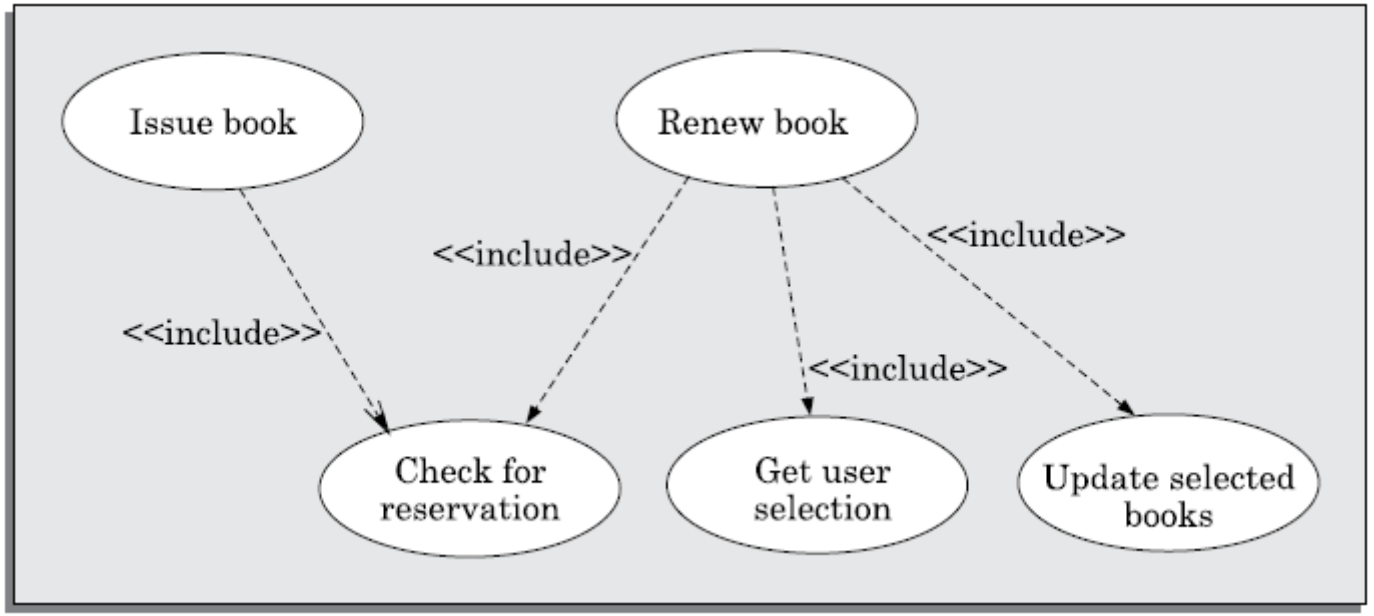


FIGURE 6.16 Example of use case inclusion.

3. 🌿 Extend (Optional Behavior)

- Used to represent **optional or conditional behavior** that extends a base use case.
- Executed **only under certain conditions**.
- Extension happens at a **defined point** in the main use case (extension point).

📌 **Notation**: Dashed arrow labeled <<extend>> pointing **away** from the base use case.

📖 Example:

- Buy ticket may be extended by Apply discount if user is eligible.

❖ **This relationship among use cases is also predefined as a stereotype as shown in Figure 6.17.**

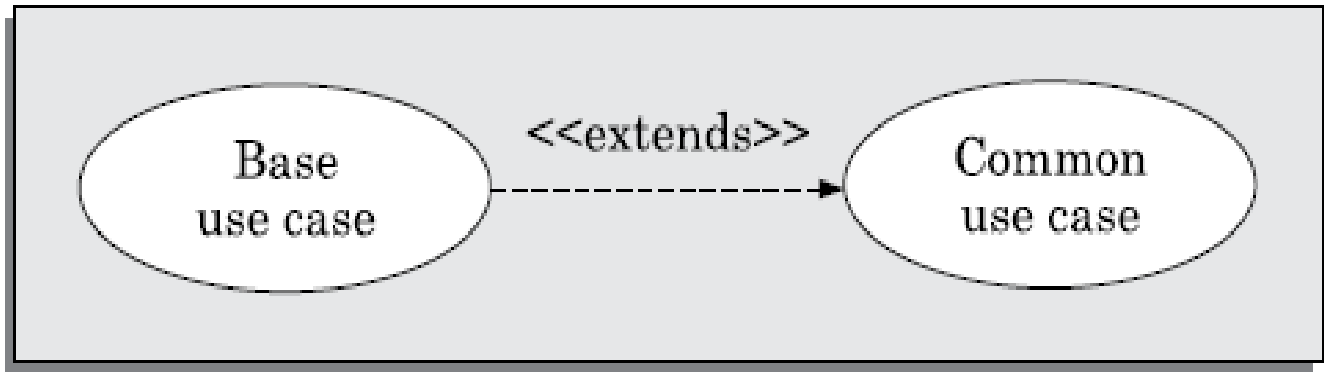


FIGURE 6.17 Example of use case extension.

🔹 Organization of Use Cases

Use cases can be **hierarchically organized**:

- **Top-level**: High-level services offered to external users.
- **Sub-level**: Smaller component use cases that combine to form the top-level ones.

🎯 This high-level diagram is often called the **Context Diagram** — shows the system boundary and only user-interacting use cases.

📌 **Only complex use cases** should be decomposed into lower-level use cases.

The high-level use cases are factored into a set of smaller use cases as shown in Figure 6.18.

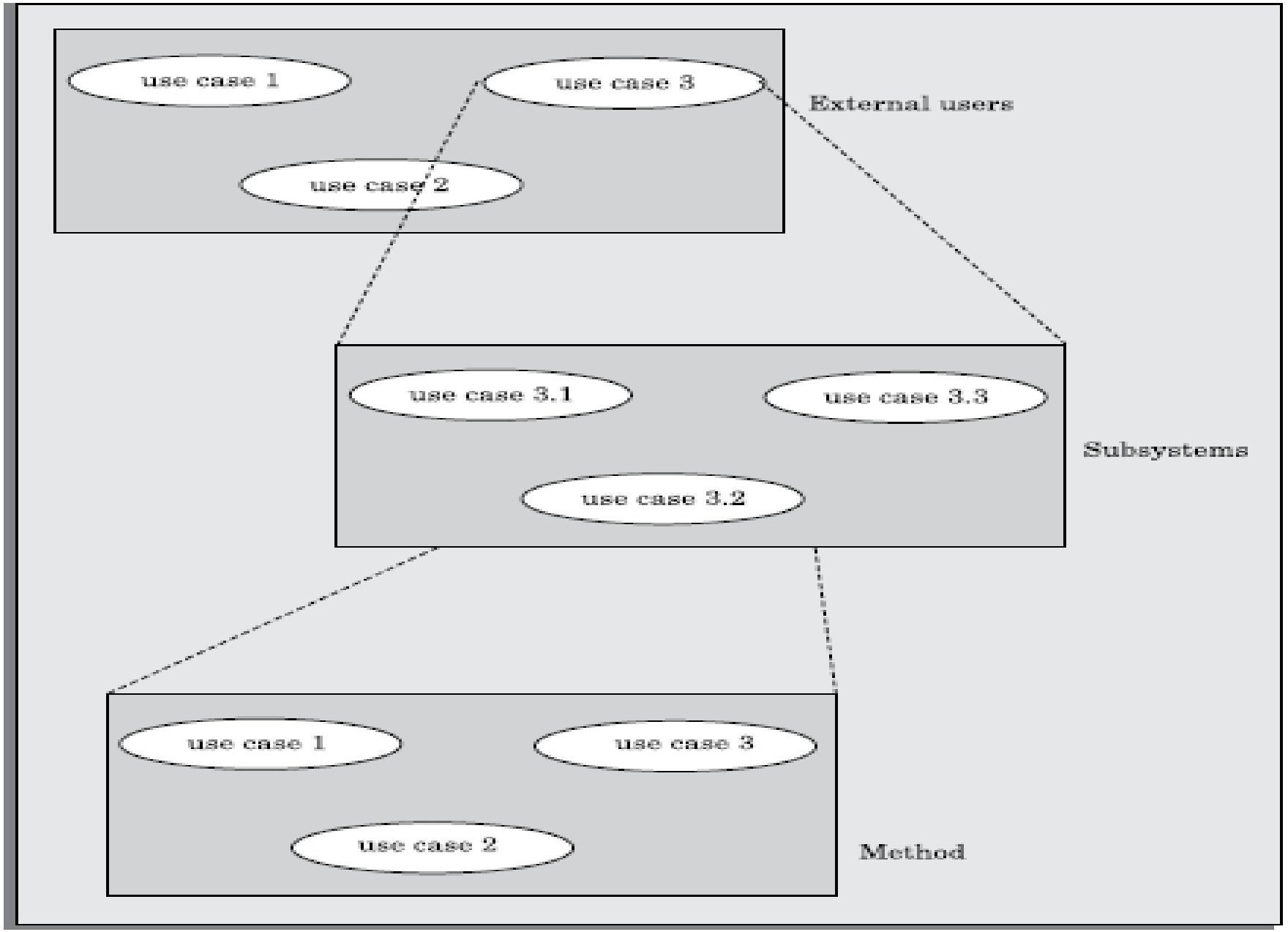


FIGURE 6.18 Hierarchical organization of

🔸 6.4.6 Use Case Packaging

✅ Why Packaging?

- **To manage complexity** when there are **too many use cases** in a system.
- Helps avoid a **cluttered top-level use case diagram**.
- Promotes **modularity, clarity**, and **reusability** in modeling.

❖ **Think of it like organizing related documents into folders on your computer.**

✅ What is a Package?

- A **package** in UML is like a **folder** 📁 that groups related elements.
- The **symbol** for a package is a **tabbed folder icon**.

🔹 A package can contain:

- Use cases
- Classes
- Components
- Even other packages (i.e., **nested packages**)

✅ Where is Packaging Used?

- **Not limited to use case diagrams**.
- Can be applied to **any UML element**:
  - Class diagrams
  - Component diagrams
  - Deployment diagrams, etc.

✅ Example Scenario

- Imagine a **University Management System**. It has use cases related to:
  - Student Accounts
  - Academics
  - Administration
- Rather than displaying 20+ use cases in a single diagram, we **group** them:

Package	Use Cases
📦 Accounts	Pay Fees, View Balance, Download Receipt
📦 Academics	Register Course, View Grades, Drop Course
📦 Admin	Create User, Assign Role, Manage System Logs

An example of packaging use cases is shown in Figure 6.19. Observe in Figure 6.19 that all accounts related use cases have been grouped into a package name Accounts. Similarly, all use cases pertaining to the academic functionalities have been put in a package named Academics, and so on.

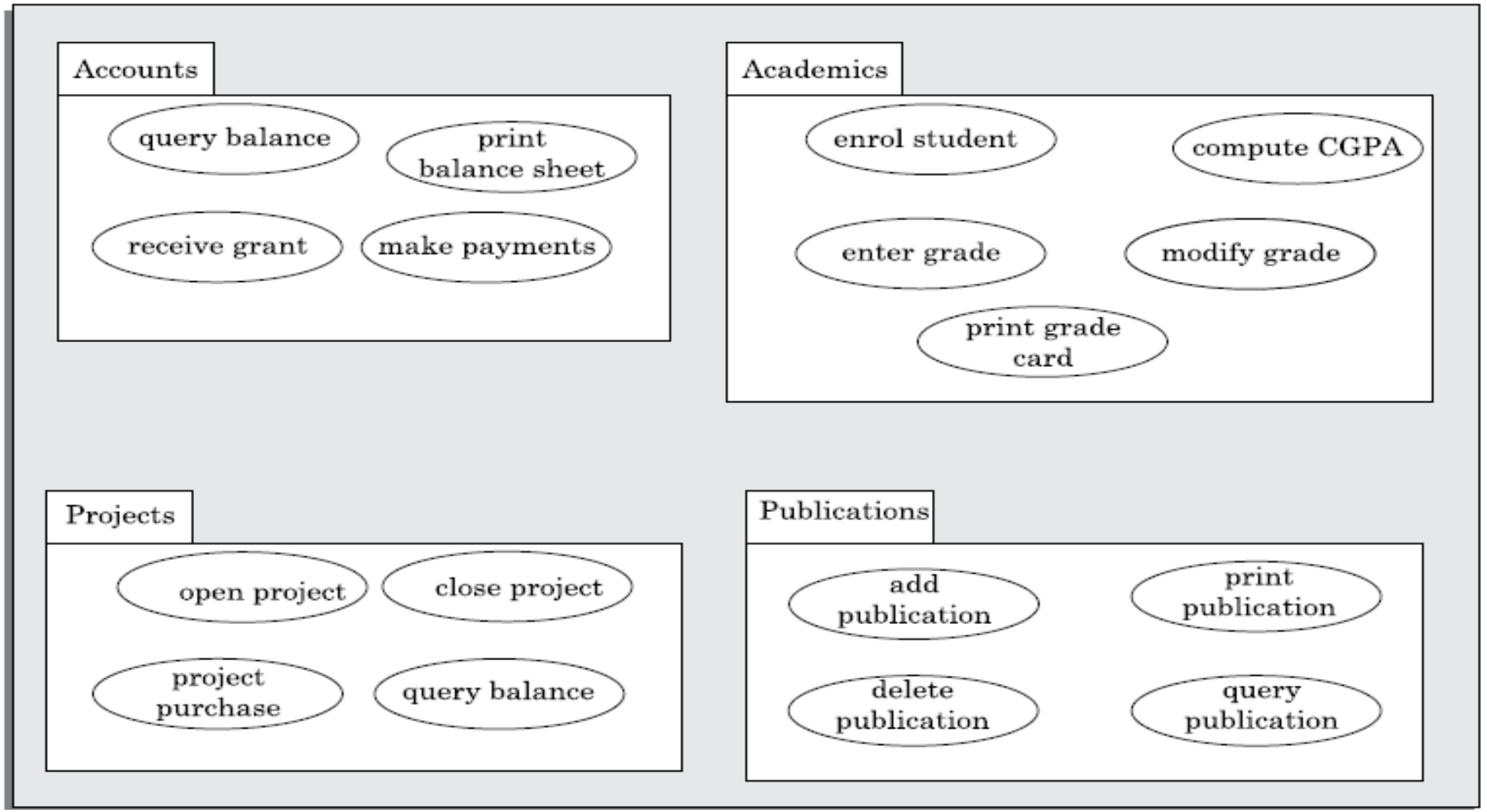


FIGURE 6.19 Use case packaging.

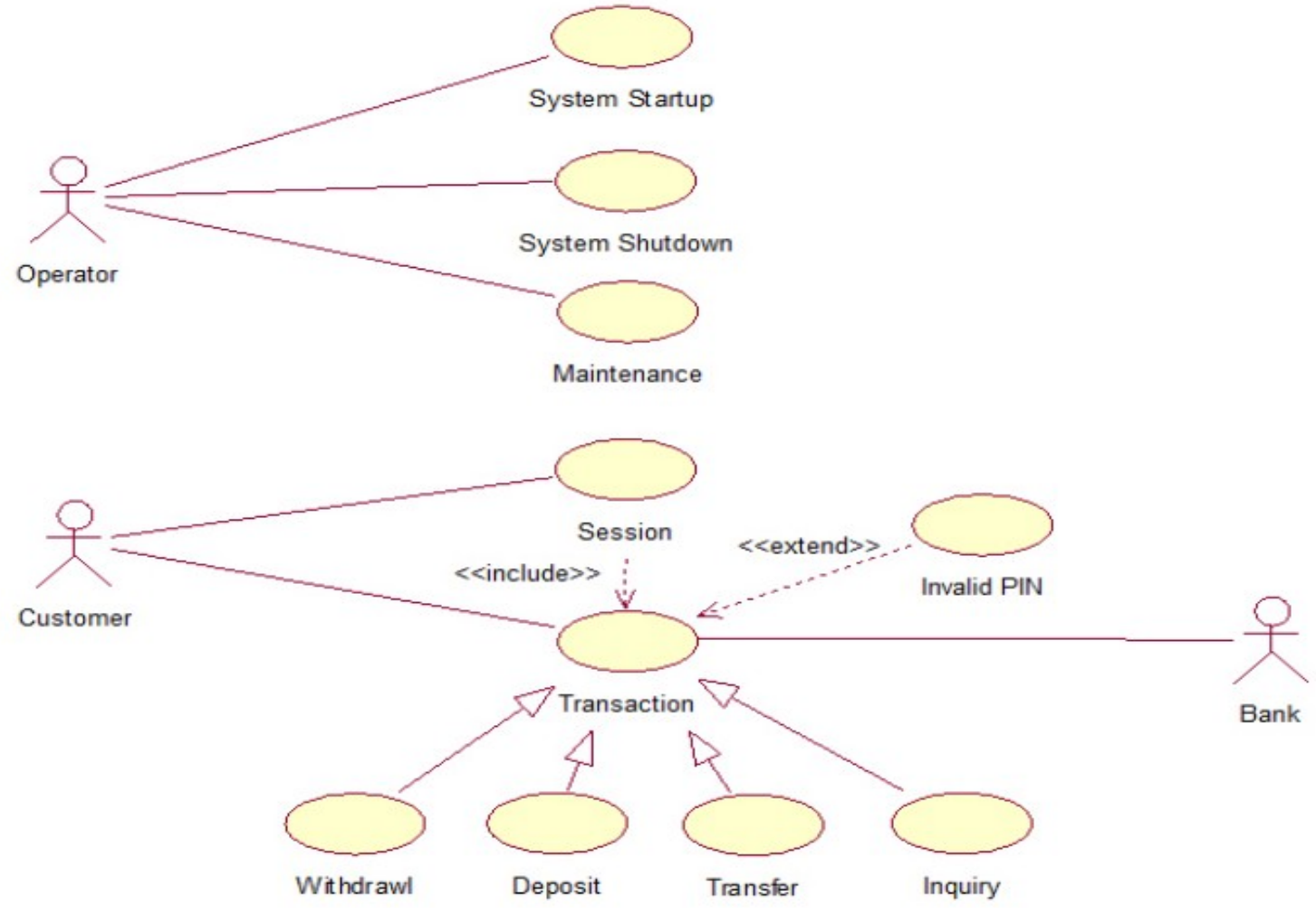
✅ Visual Representation

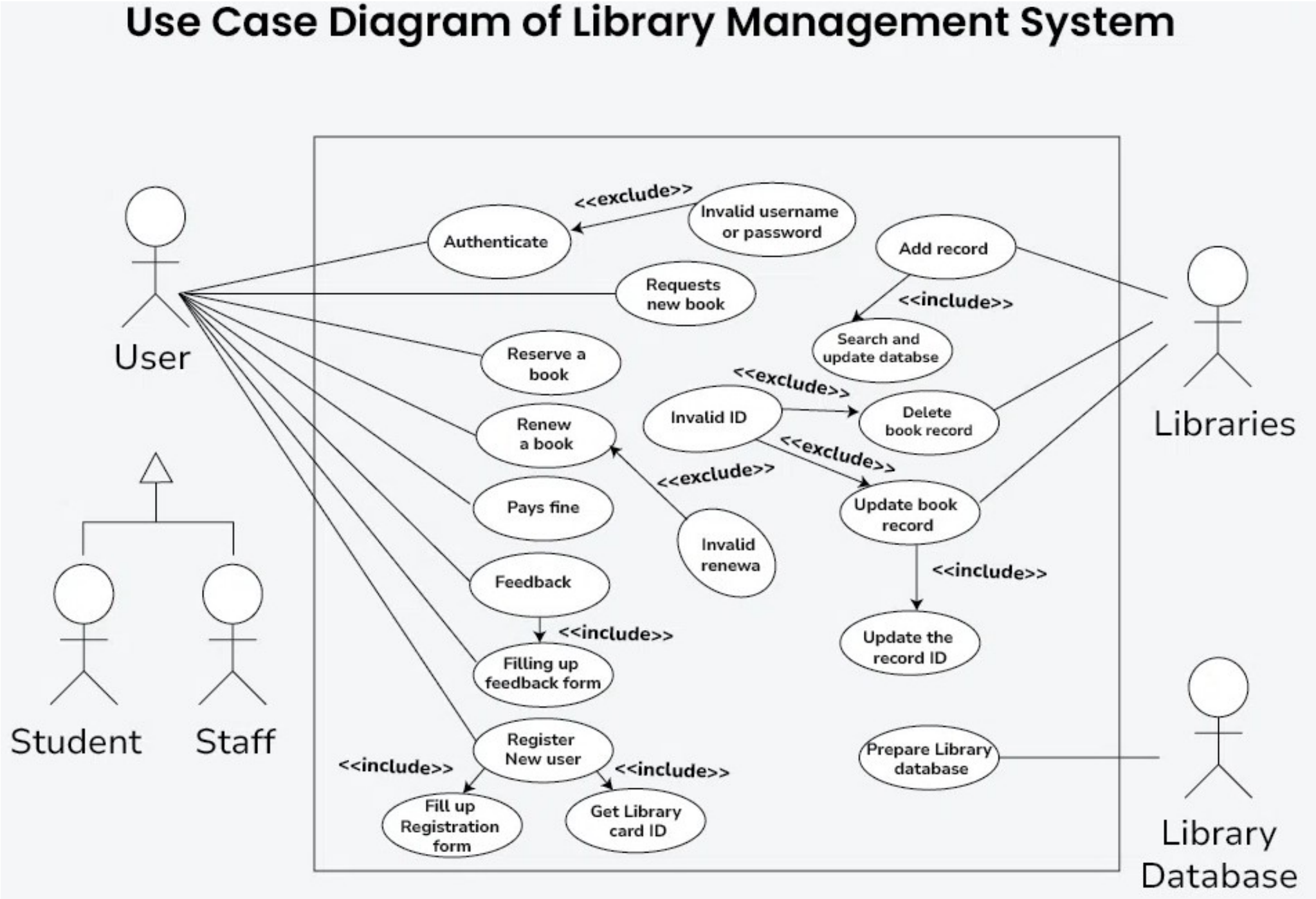
- Each package groups related use cases inside a **folder icon** with the package name at the top.
- The overall system is shown with only **6–7 top-level packages**, instead of overwhelming detail.

- **This layered structure makes it easier to navigate, design, and maintain large systems.**

Concept	Explanation
Package	A UML construct used to group related modeling elements
Symbol	A folder 📁 with a tab
Purpose	Simplifies complex diagrams by organizing related elements
Scope	Can be used across all types of UML diagrams
Nested Packages	A package can contain other packages

Sample Use – Case : ATM application




Use Case Diagram of Library Management System
User
Student
Staff
Libraries
Library Database
Authenticate
<<exclude>>
Invalid username or password
Requests new book
Reserve a book
Renew a book
Pays fine
Feedback
<<include>>
Filling up feedback form
Register New user
<<include>>
<<include>>
Fill up Registration form
Get Library card ID
Add record
<<include>>
Search and update databse
<<exclude>>
Invalid ID
Delete book record
<<exclude>>
<<exclude>>
Invalid renewa
Update book record
<<include>>
Update the record ID
Prepare Library database

6.5 CLASS DIAGRAMS (Static Structure Models)

A **Class Diagram** represents the **static structure** of a system. It shows:

- **Classes** and
- **Relationships** among them:
  - Association
  - Aggregation
  - Composition
  - Inheritance
  - Dependency

A **class** is drawn as a rectangle with three compartments:

```
+---------------------+
| ClassName           |
+---------------------+
| Attributes          |
+---------------------+
| Operations/Methods  |
+---------------------+
```

✅ 1. Class Representation

A **class** is drawn as a rectangle with three compartments:

- **Class name**: Bold, centered, TitleCase (e.g., LibraryMember)
- **Object name**: camelCase (e.g., studentMember)
- Classes can be drawn with only the name if attributes/methods are not yet known.

An example of various representations of a class supported in UML are shown in Figure 6.20. It can be seen in Figure 6.20 that classes can optionally have attributes and operations compartments. When a class appears in several diagrams, its attributes and operations are suppressed on all but one diagram.

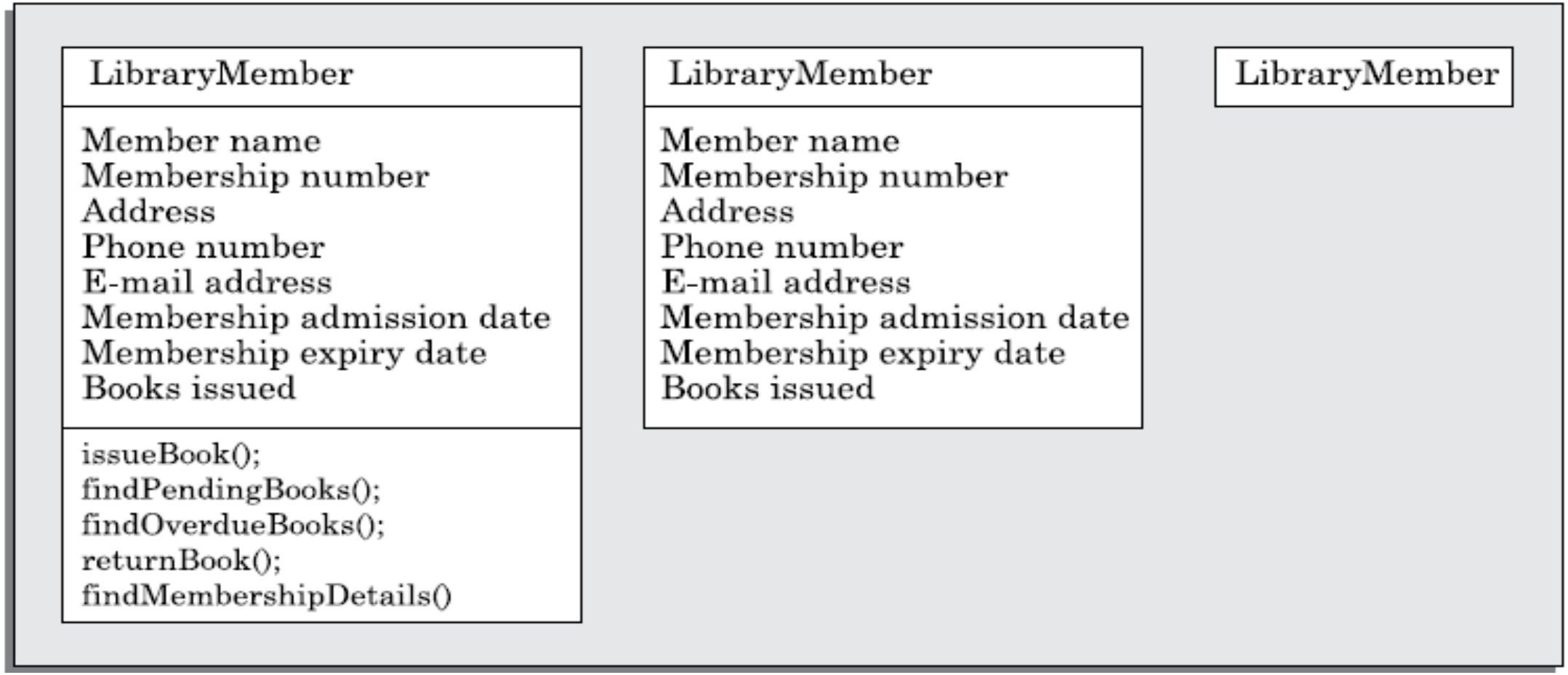


FIGURE 6.20 Different representations of the Library Member class.

✅ 2. Attributes

- Syntax: attributeName : Type = InitialValue
- Example: sensorStatus : Int = 0
- **Multiplicity**: Use square brackets. Example: sensorStatus[10] = array
- Must begin with lowercase

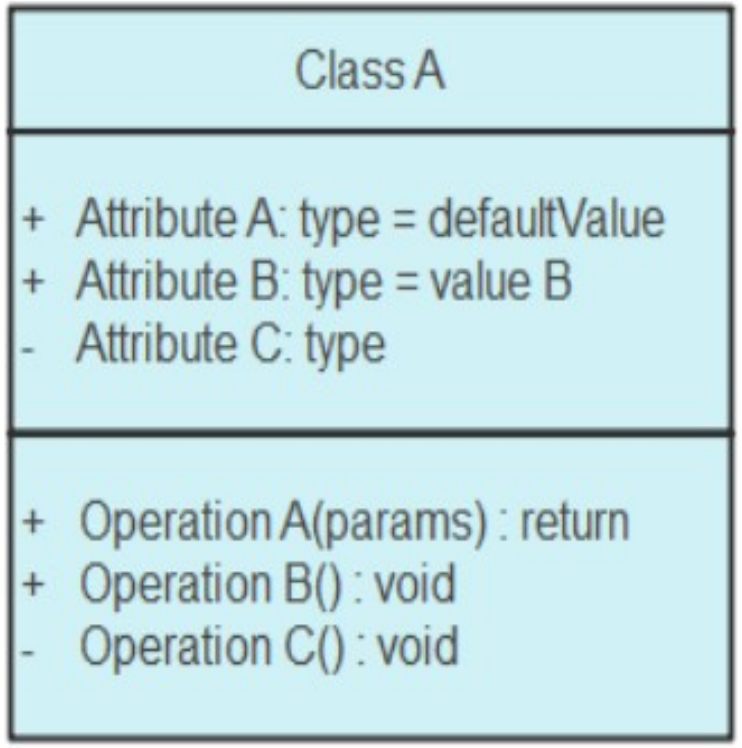
✅ 3. Operations (Methods)

- Syntax: operationName(parameters): ReturnType
- Example: issueBook(in bookName): Boolean
- Use **italics** or {abstract} for abstract operations
- Use in, out, or inout to mark parameter direction
- Underline method name for **class-scope (static)** methods

📌 Operation vs Method:

- **Operation** = what the class offers
- **Method** = how it's implemented (can have multiple methods per operation via overloading)

Class Diagram



Class Name

- The class name is important for graphical representation.
- It should be written in bold in the top compartment and start with a capital letter.
- Moreover, an abstract class should be written in italics.

Attributes

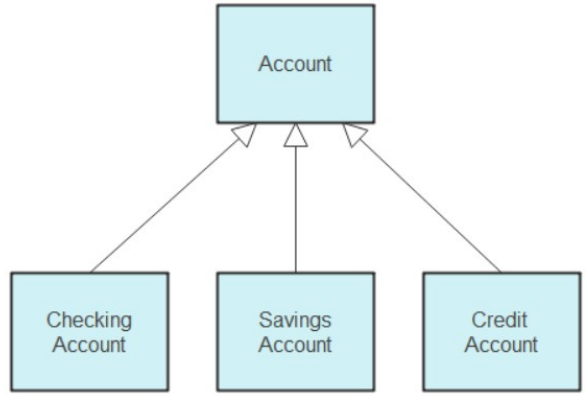
- The properties of the object being modeled.
- Attributes must be meaningful and are usually used with the visibility factor that describes the accessibility of an attribute

Operations

- Operations are processes that a class knows to carry out.
- They correspond to the methods of a class.

Class Relationships

- There are three main types of relationships
  - Generalizations
  - Associations
  - Dependencies



Generalizations :

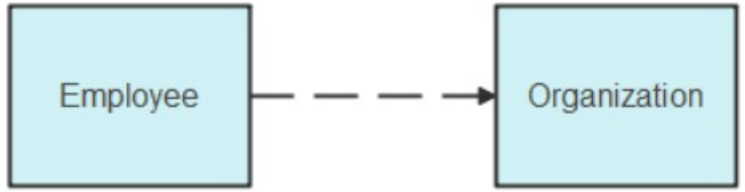
- **Generalizations** are often known as **Inheritance** because it links a subclass to its superclass.

Associations:

- Association shows a static relationship between two entities

Dependencies:

- Dependency shows that one class depends on another.



✅ 4. Association

- Represents a **link** or connection between two classes
- Line between classes with optional **association name** and **multiplicity**
- Example: LibraryMember ---- borrows ----> Book
  - LibraryMember borrows many books (*)
  - Each book borrowed by one member (1)
- **Multiplicity** examples:
  - 1 = one
  - 0..1 = optional
  - * = many (zero or more)
  - 1..5 = between 1 and 5

Figure 6.21 illustrates the graphical representation of the association relation.

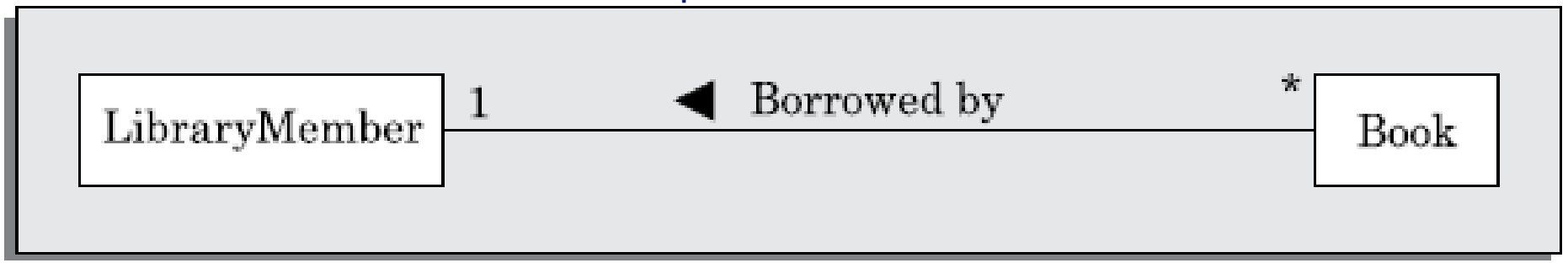


FIGURE 6.21 Association between two classes.

✅ 5. Aggregation (Has-a)

- Whole–part relationship (e.g., Register has Books)
- Represented with an **empty diamond**
- Parts can exist independently
- Example:
  - BookRegister ◇----> Book

An example of the aggregation relationship has been shown in Figure 6.22. The figure represents the fact that a document can be considered as an aggregation of paragraphs. Each paragraph can in turn be considered as an aggregation of sentences. Observe that the number 1 is annotated at the diamond end, and the symbol * is annotated at the other end. This means that one document can have many paragraphs. On the other hand, if we wanted to indicate that a document consists of exactly 10 paragraphs, then we would have written number 10 in place of the symbol *.

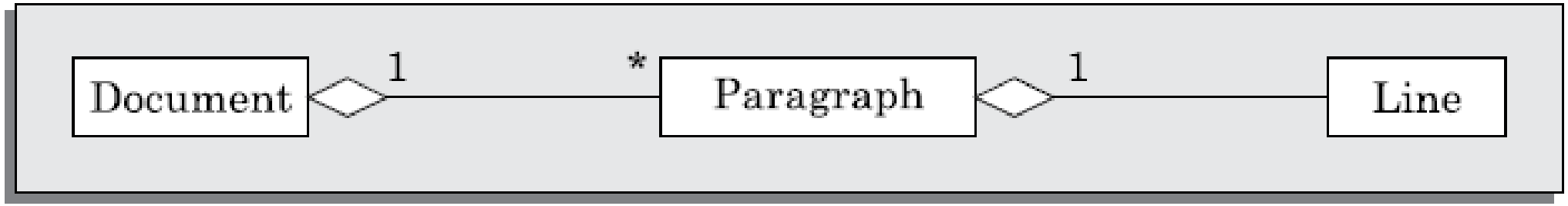


FIGURE 6.22 Representation of aggregation.

✅ 6. Composition (Strict Has-a)

- Stronger form of aggregation
- Parts are **created and destroyed** with the whole
- Represented with a **filled diamond**

•Example:

- Order ◆----> OrderItem

📌 **Key difference**:

- Aggregation: parts **can** exist independently
- Composition: parts **cannot** exist independently

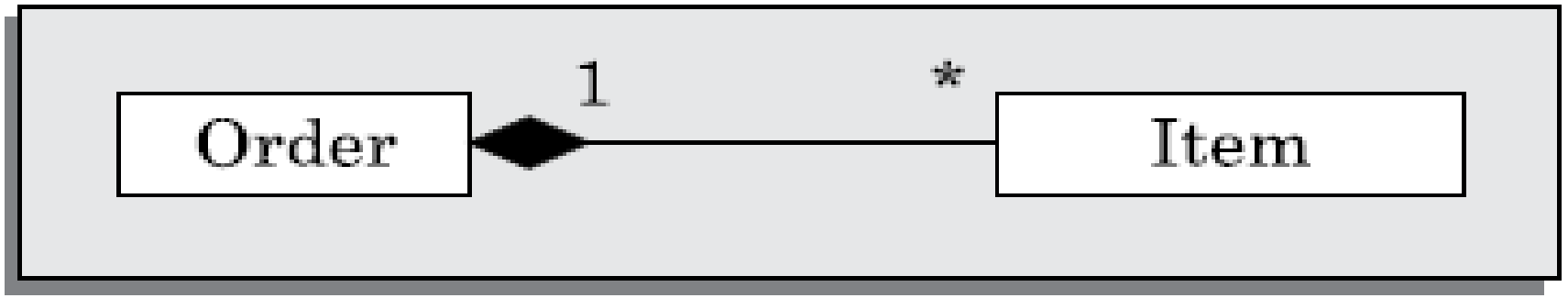


FIGURE 6.23 Representation of composition.

✅ 7. Inheritance (Is-a / Generalization)

- Denoted by a **triangle-headed arrow** from subclass to superclass
- Shared behavior is in the superclass

•**Example:**

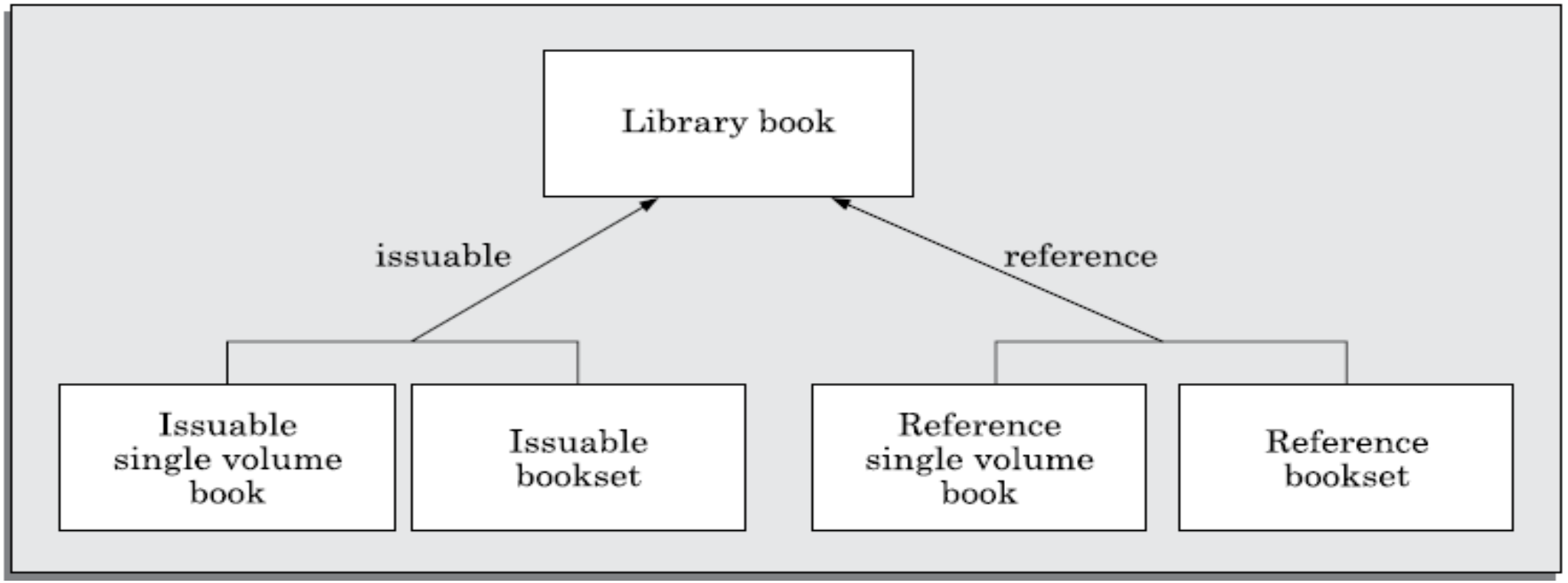


FIGURE 6.24 Representation of the inheritance relationship.

❖ **Can show discriminators (e.g., <<type>>: issuable/reference) for specialization**

✅ 8. Dependency

- Weak relationship showing one class **uses** another
- Represented by a **dotted arrow**
- Used for temporary interactions (e.g., function parameters)

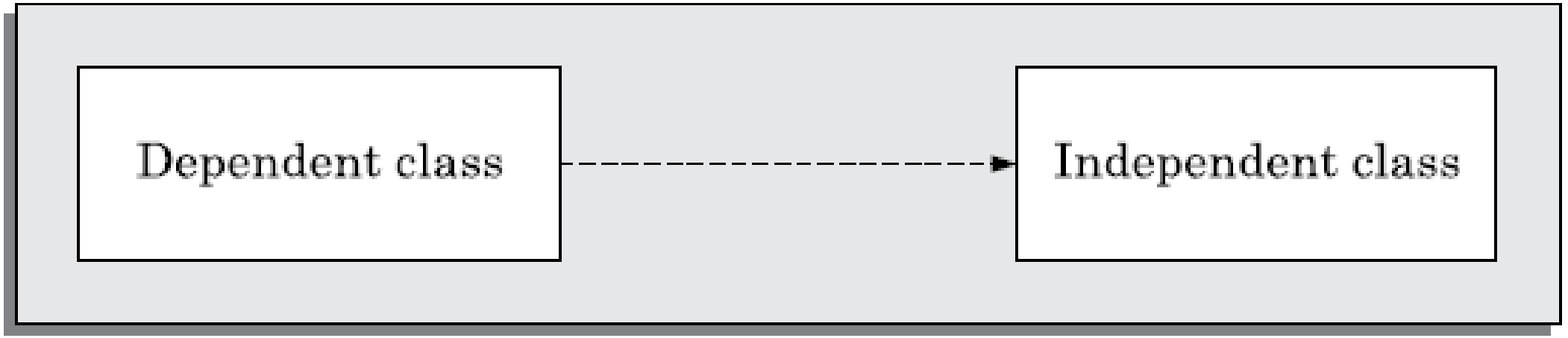


FIGURE 6.25 Representation of dependence between classes.

✅ 9. Constraints

- Written in curly braces {} to show rules or limitations
- Example: {sorted}, {unique}, {abstract}
- Can also use **Object Constraint Language (OCL)** for formal specification

✅ 10. Object Diagrams (Instance Diagrams)

- Represent a **snapshot** of the system at runtime
- Show actual **objects (instances)**, not classes
- Useful for explaining system behavior
- Syntax similar to class diagrams but uses **rounded rectangles**

•Example:

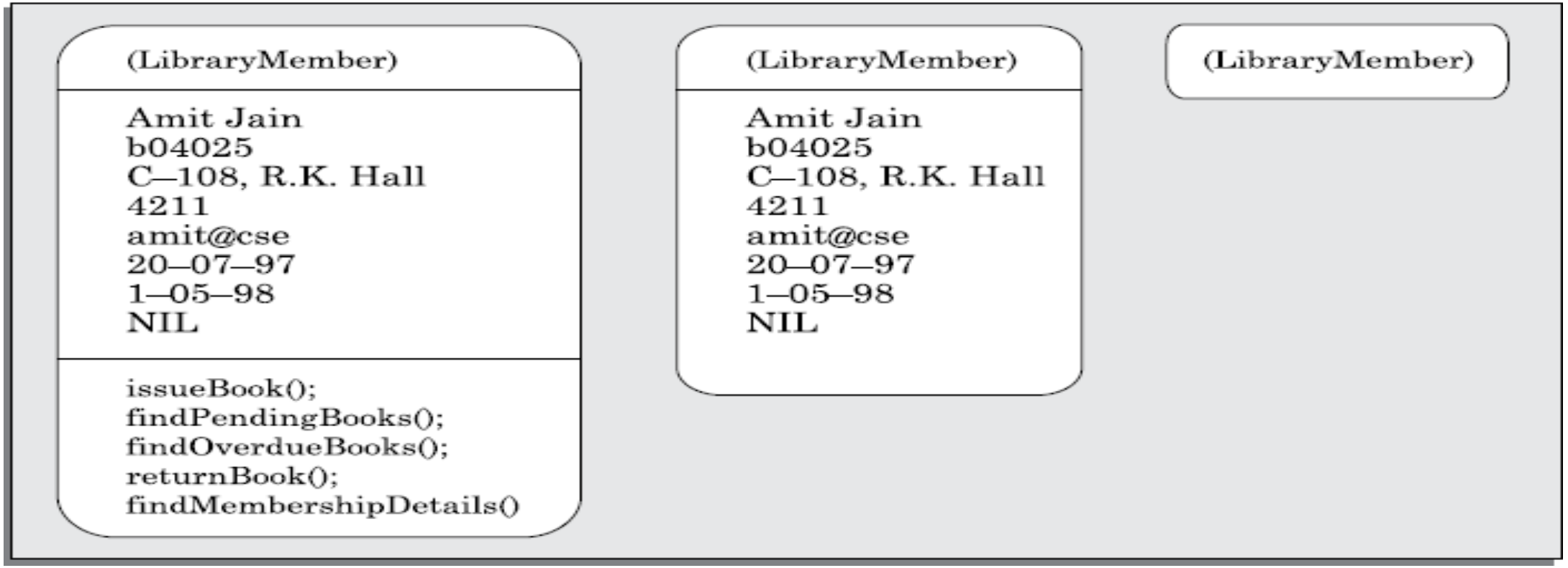
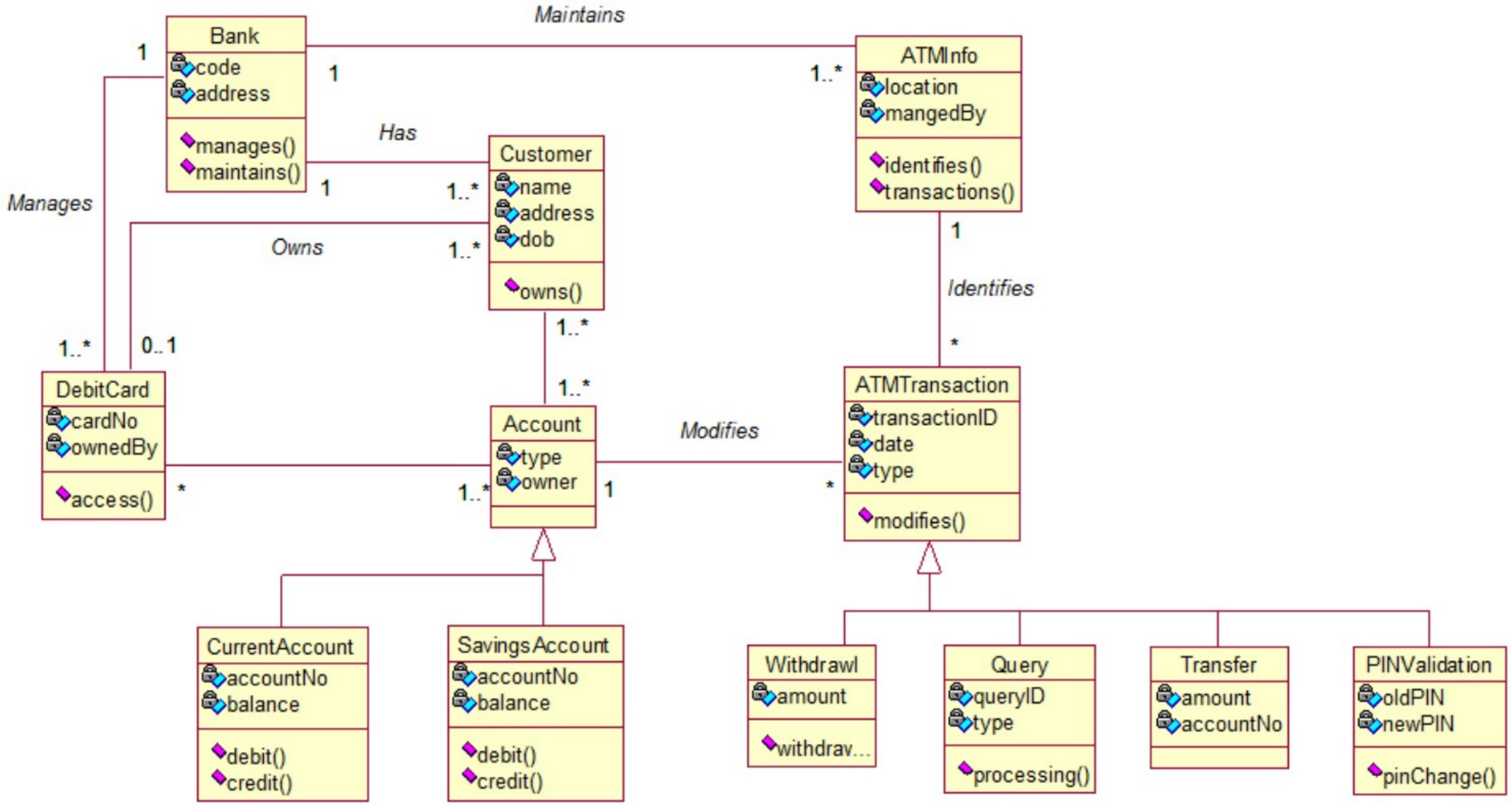


FIGURE 6.26 Different representations of a Library Member object.

Concept	UML Symbol / Notation	Notes
Class	Rectangle with compartments	Name, Attributes, Methods
Attribute	name : Type = default	Can use multiplicity
Operation	opName(params): ReturnType	Italics = abstract, underline = static
Association	Solid line, optional arrow + multiplicity	Relationship between classes
Aggregation	Empty diamond ◇	Whole–part, parts may live independently
Composition	Filled diamond ◆	Parts die with the whole
Inheritance	Triangle-headed line	Subclass to superclass
Dependency	Dotted arrow →	Temporary use
Constraint	{constraint}	Informal or OCL
Object Diagram	Rounded rectangles	Runtime snapshot

Class Diagram - **ATM application**



6.6 INTERACTION DIAGRAMS

- **Purpose**:
  Interaction diagrams show how objects **collaborate** via **message passing** to implement a **use case**.
  - **Each interaction diagram usually maps to a single use case.**

✅ Types of Interaction Diagrams

1. **Sequence Diagram**
2. **Collaboration Diagram**

Though they represent the **same behavior**, they offer **different perspectives**:

- **Sequence Diagram** → Emphasizes **time ordering**
- **Collaboration Diagram** → Emphasizes **object structure and relationships**

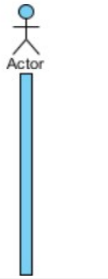
✅ 1. Sequence Diagram

Definition:

A **2D diagram** that shows the **sequence of messages** exchanged between objects **top to bottom** in **time order**.

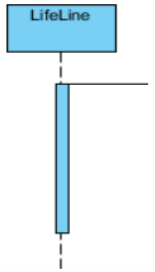
Element	Description
Objects	Shown as boxes at the top (e.g., :Book). Underlined.
Lifeline	Vertical dashed line showing object's lifetime.
Activation	A thin rectangle on lifeline showing object is "active" (executing).
Messages	Arrows between lifelines. Labeled with method name, and optional condition or iteration.
Conditions	Shown in square brackets. E.g., [invalid]
Loops/Iterations	E.g., [for each book]

Sequence Diagram Notation

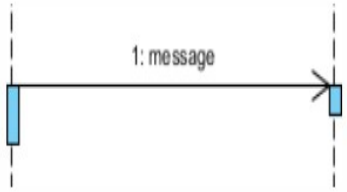


Actor - Represent roles played by human users, external hardware, or other subjects.

Lifeline - Represents an individual participant in the Interaction.

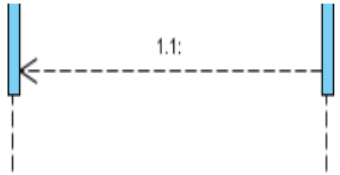


Activations - A thin rectangle on a lifeline represents the period during which an element is performing an operation.



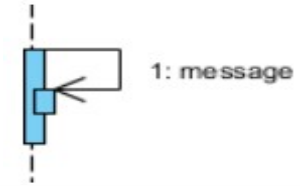
Call Message - A message defines a particular communication between the Lifelines of an Interaction.

Sequence Diagram Notation Cont'd

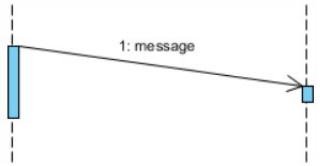


Return Message - Return message is a kind of message that represents the pass of information back to the caller of a corresponding former message..

Self Message - Self message is a kind of message that represents the invocation of message of the same lifeline.



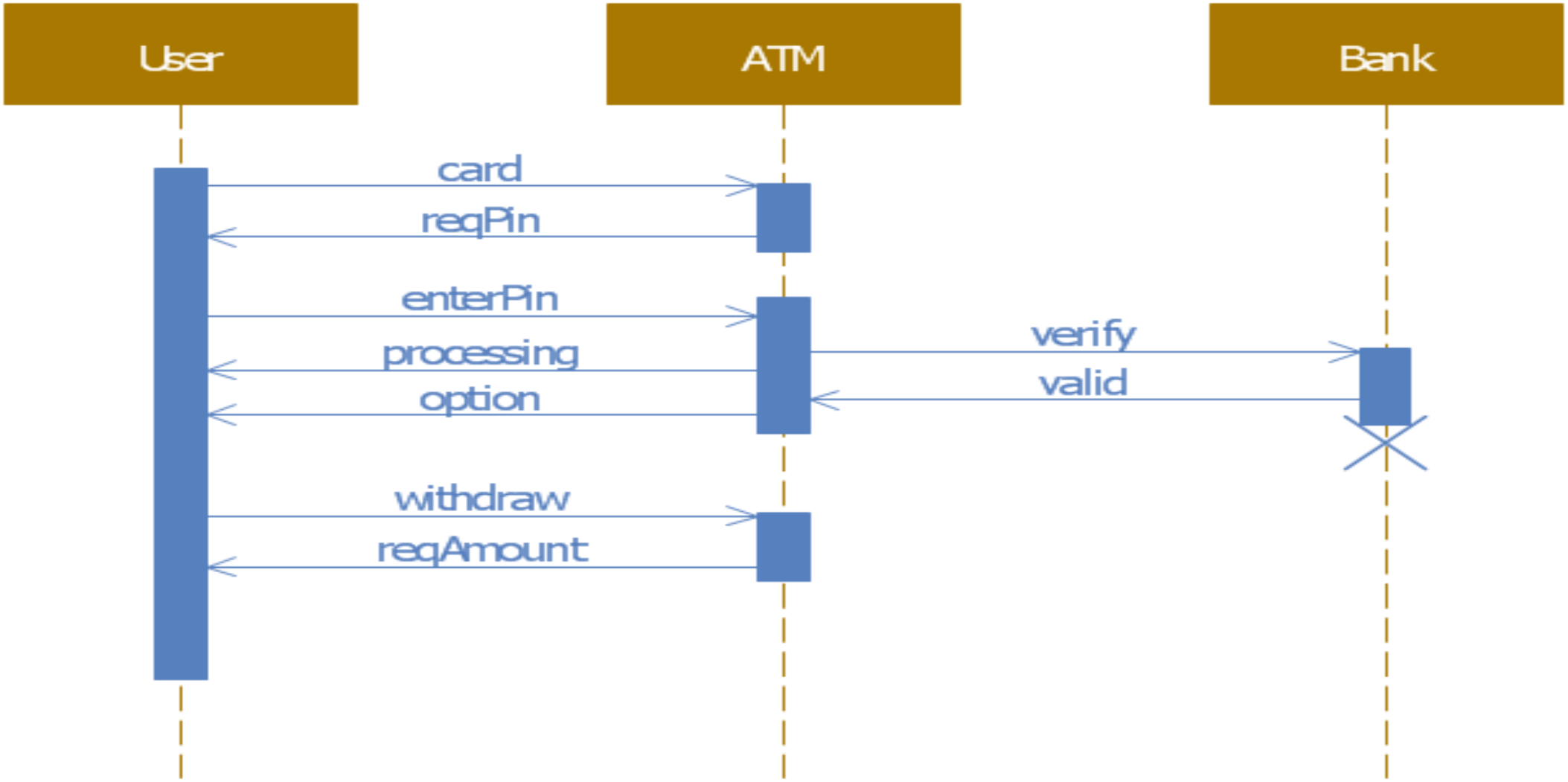
Recursive Message - Recursive message is a kind of message that represents the invocation of message of the same lifeline.



Duration Message - Duration message shows the distance between two time instants for a message invocation.

Note - A note (comment) gives the ability to attach various remarks to elements.

Sequence Diagrams for ATM



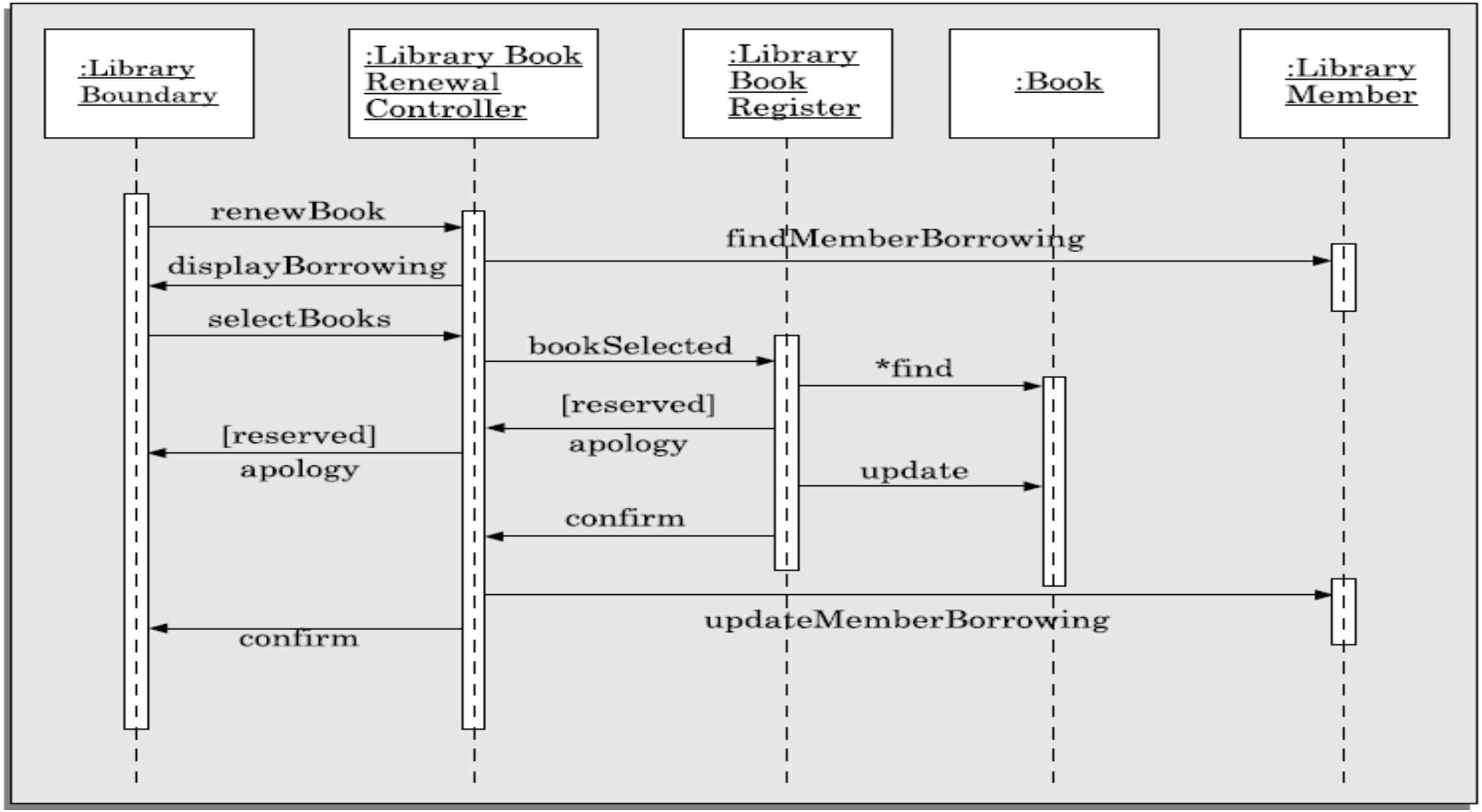


FIGURE 6.27 Sequence diagram for the renew book use case.

Purpose of Sequence Diagram

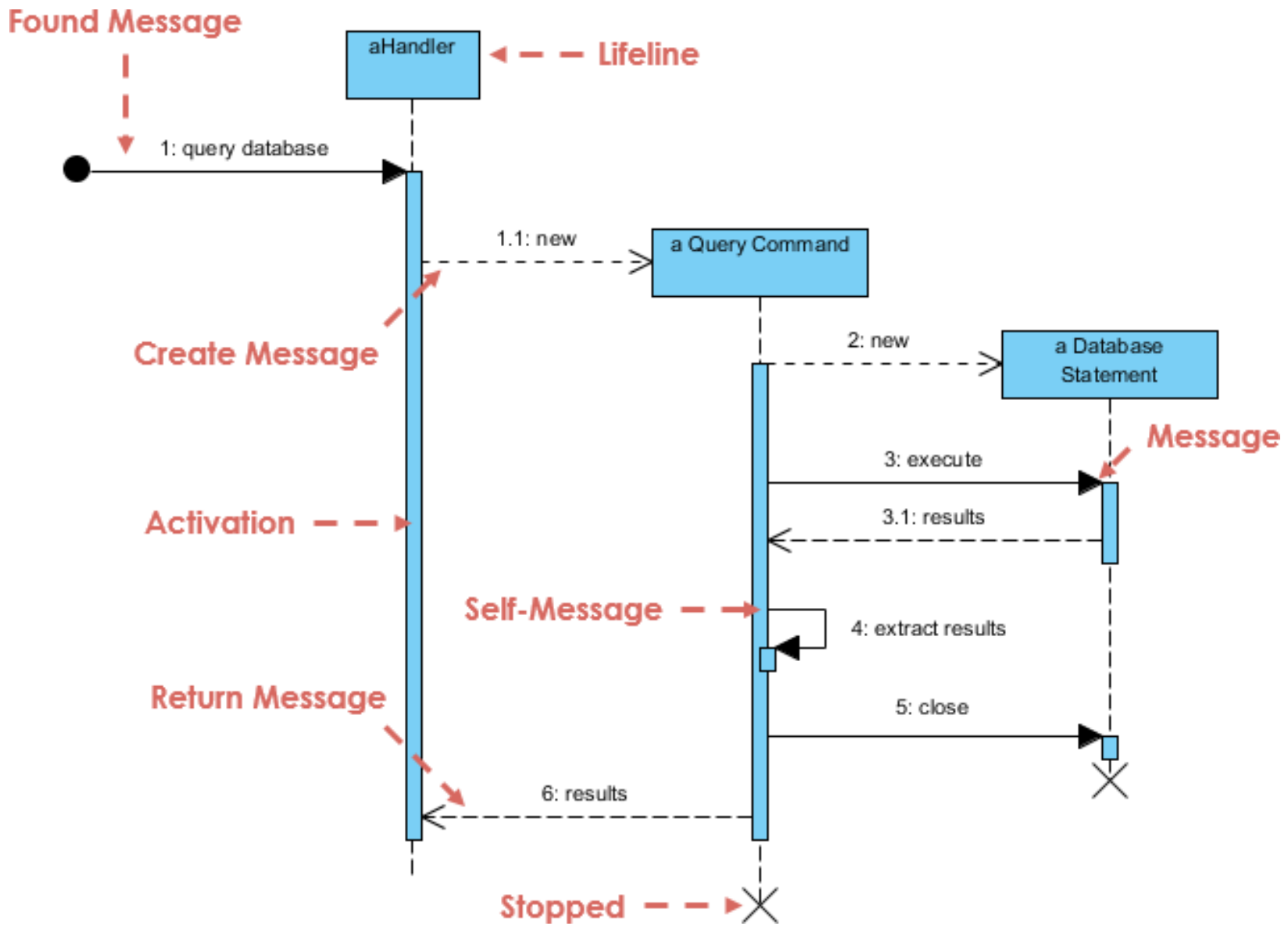
- Model high-level interaction between active objects in a system.
- Model the interaction between object instances within a collaboration that realizes a use case.
- Model the interaction between objects within a collaboration that realizes an operation
- Either model generic interactions (showing all possible paths through the interaction) or specific instances of a interaction (showing just one path through the interaction)

💡 Notes:

- **Object created during interaction** → placed at point of creation.
- **Destroyed object** → lifeline crossed out (×).

Example (From Book Renewal Use Case):

- Objects: :Member, :Book, :LibrarySystem
- Messages: verifyMember(), checkReservation(), renewBook()...
- Time flows **top to bottom**.


Found Message
aHandler
Lifeline
1: query database
1.1: new
a Query Command
Create Message
2: new
a Database Statement
Message
3: execute
Activation
3.1: results
Self-Message
4: extract results
Return Message
5: close
6: results
Stopped

✅ 2. Collaboration Diagram

Definition:

Also called **communication diagrams**, these show:

- **Structural relationships** between objects (**links**)
- **Behavioral flow** through **message numbers**

📌 Key Elements:

Element	Description
Objects (Collaborators)	Boxes representing object instances
Links	Solid lines showing associations
Messages	Arrows on links labeled with sequence numbers
Sequence	Numbers like 1, 1.1, 1.2 show message order (replacing top-down layout)

The collaboration diagram for the example of Figure 6.27 is shown in Figure 6.28. For a given sequence diagram, the collaboration diagram can be automatically generated by a CASE tool.

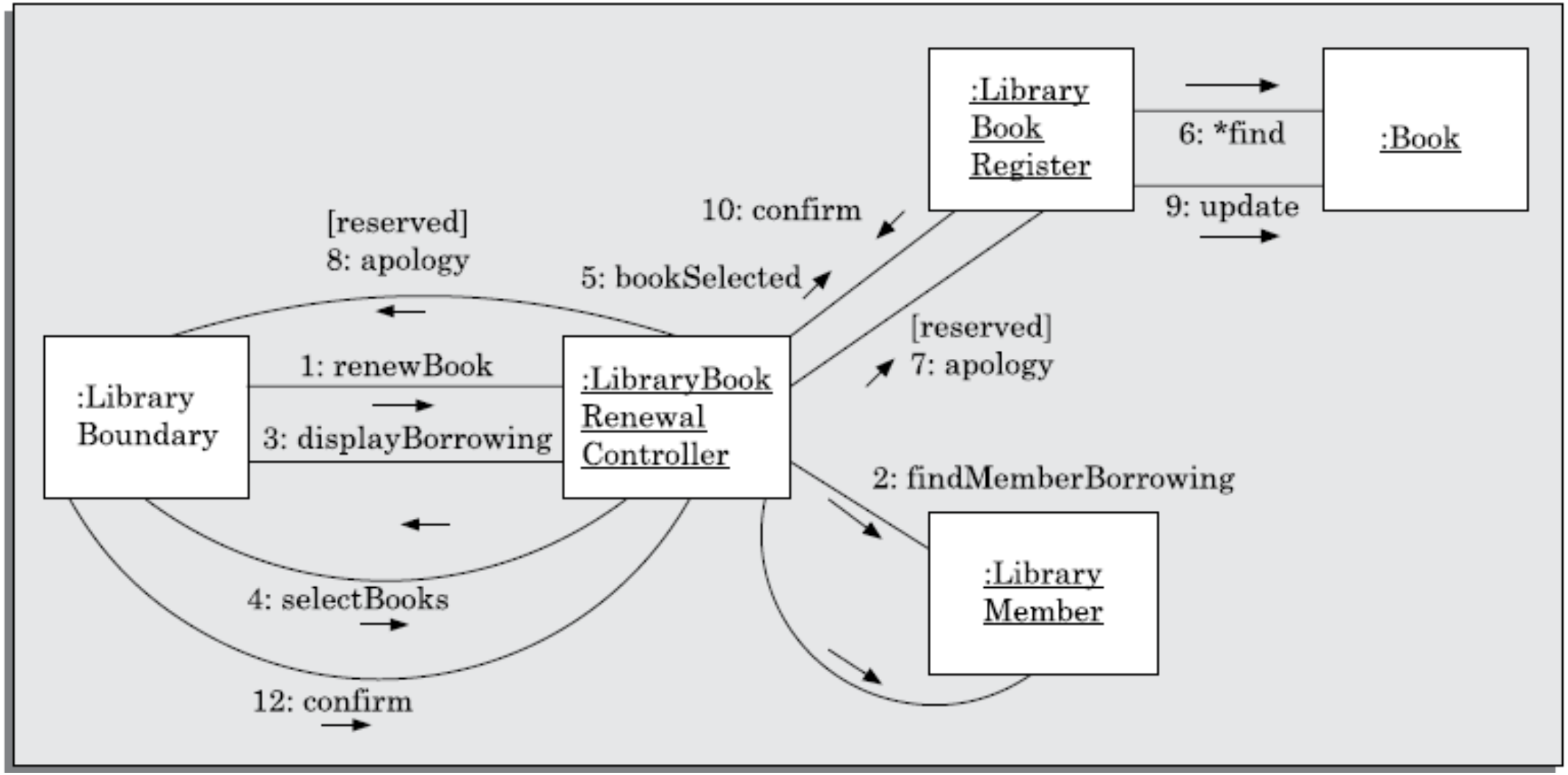
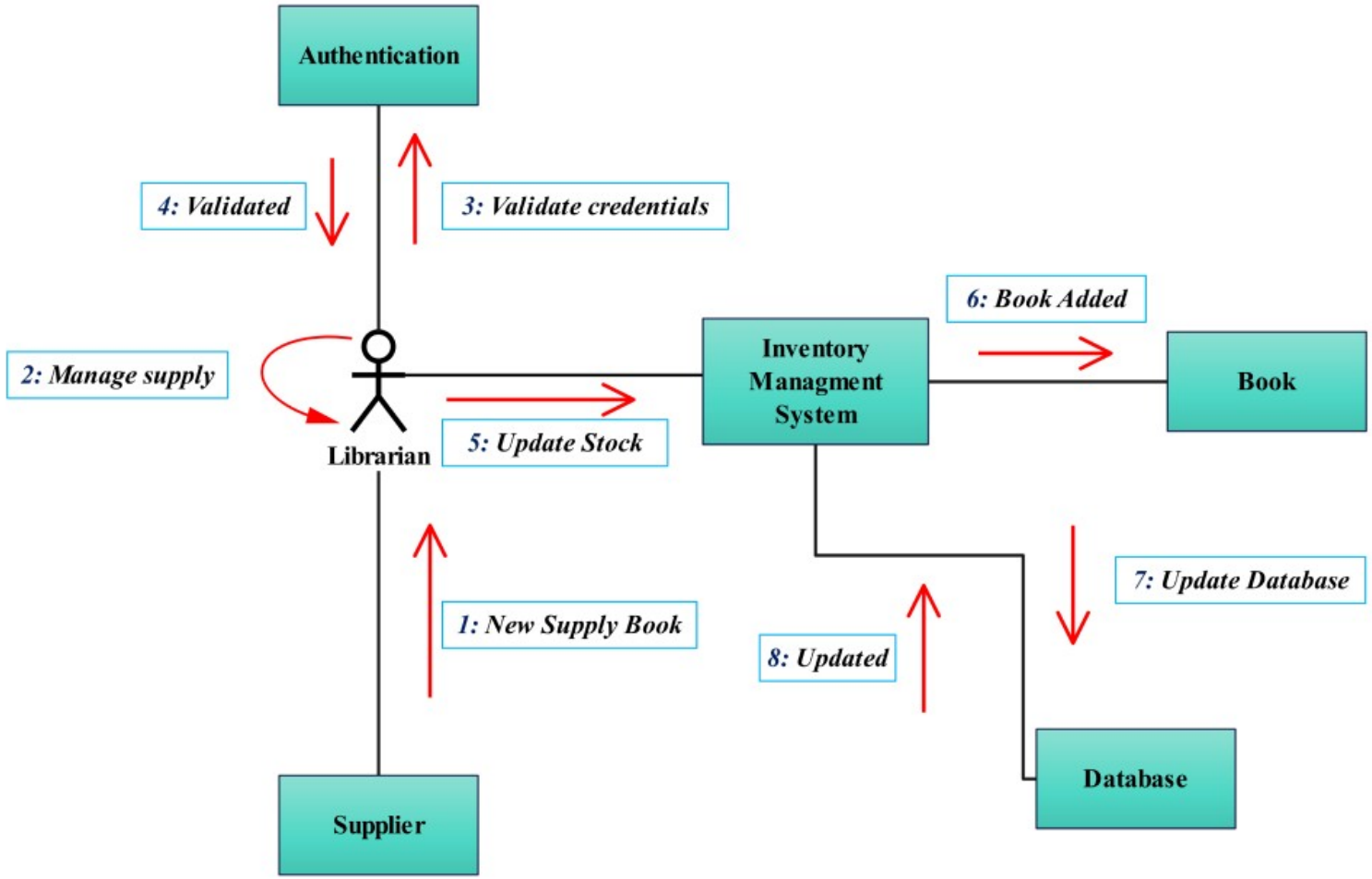


FIGURE 6.28 Collaboration diagram for the renew book use case.


Authentication
4: Validated
3: Validate credentials
6: Book Added
2: Manage supply
Inventory Managment System
Book
Librarian
5: Update Stock
1: New Supply Book
7: Update Database
8: Updated
Database
Supplier

Feature	Sequence Diagram	Collaboration Diagram
Focus	Time/order of messages	Structural links between objects
Read Direction	Top to bottom	Numbered message sequence
Emphasizes	Temporal ordering	Object relationships
Complexity	Easier for large scenarios	More compact, better for structure
Tool support	Easily convertible to/from each other using CASE tools	

- **Interaction diagrams** model **runtime behavior** of use cases.
- Use **sequence diagrams** to understand **timing and control flow**.
- Use **collaboration diagrams** to understand **who interacts with whom**.
- Both diagrams are essential tools in **OO design methodology** and help determine **class responsibilities** (i.e., what methods each class should support).

6.7 ACTIVITY DIAGRAM

✅ What Is an Activity Diagram?

- An **activity diagram** models the **flow of control or data** between **activities** during the execution of a system.
- It captures **high-level workflow** of a **business process** or **use case behavior**, **not** just method-level details.

Feature	Description
Activity	Represents a task or step. Internally contains an action and one or more outgoing flows.
Transition	Arrows connecting activities. May have guard conditions that determine which path is taken.
Decision Node	Branching logic with conditions (like if-else in flowcharts).
Merge Node	Brings multiple branches back into a single flow.
Start Node	Black filled circle (●) shows where the flow begins.
End Node	Black circle with a border (◎) shows termination.
Fork Node	A horizontal bar used to split the flow into parallel paths.
Join Node	A horizontal bar used to synchronize parallel paths.

When to Use Activity Diagram

- Activity Diagrams describe how activities are coordinated to provide a service which can be at different levels of abstraction.

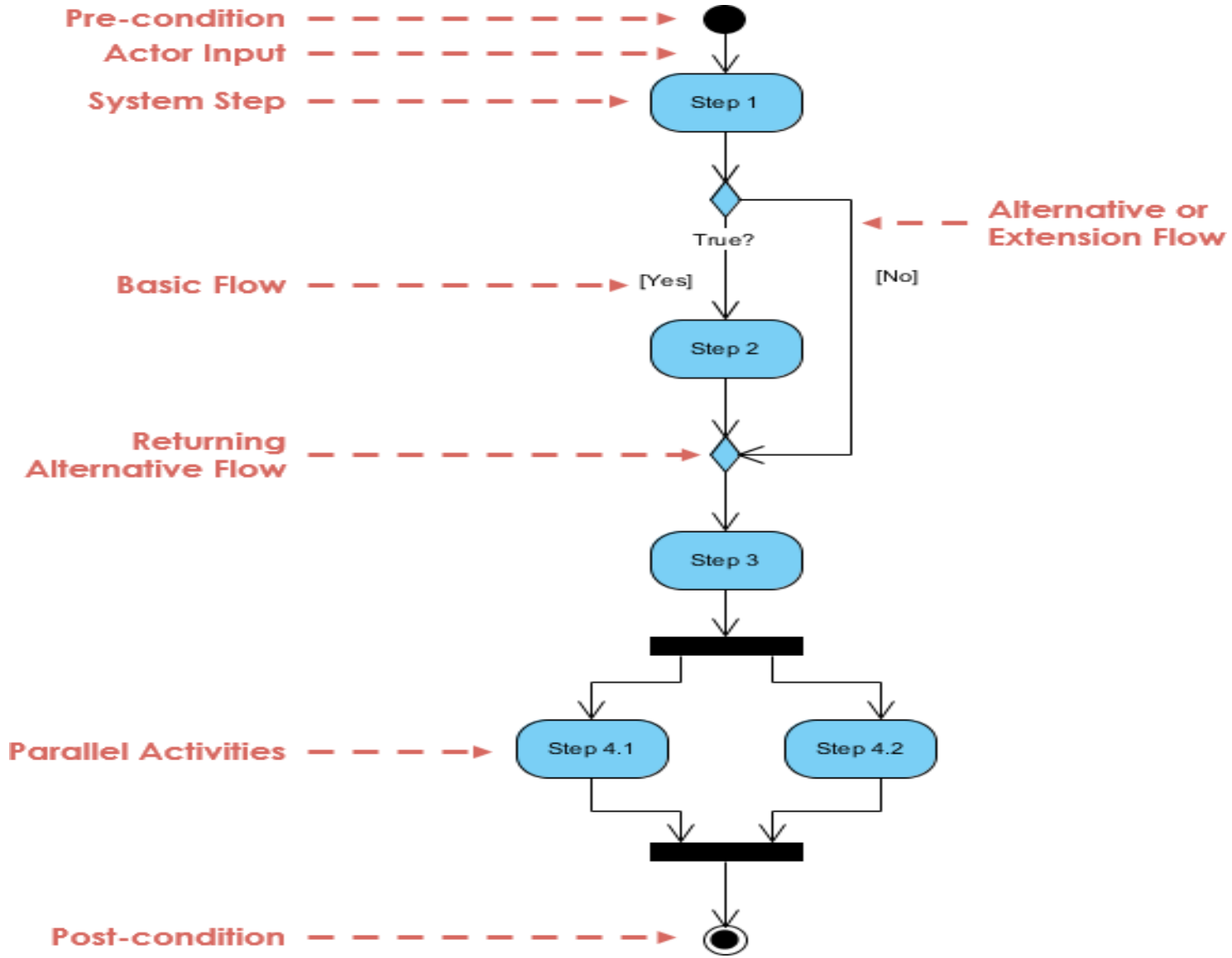
1.Identify candidate use cases, through the examination of business workflows.

2.Identify pre- and post-conditions (the context) for use cases.

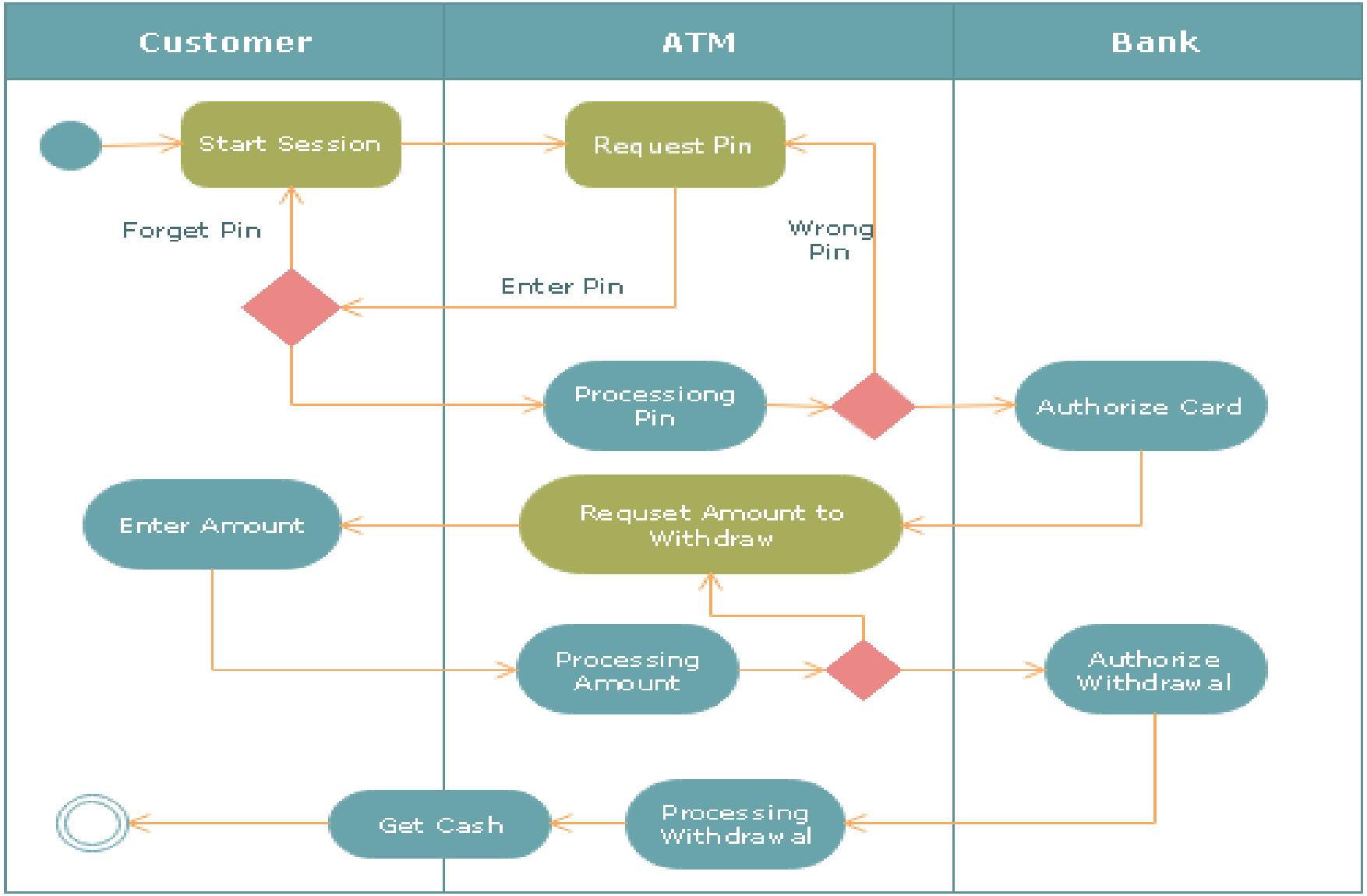
3.Model workflows between/within use cases.

4.Model complex workflows in operations on objects.

5.Model in detail complex activities in a high level activity Diagram.


Pre-condition
Actor Input
System Step
Step 1
True?
Alternative or
Extension Flow
Basic Flow
[Yes]
[No]
Step 2
Returning
Alternative Flow
Step 3
Parallel Activities
Step 4.1
Step 4.2
Post-condition

ATM Withdrawal Activity Diagram



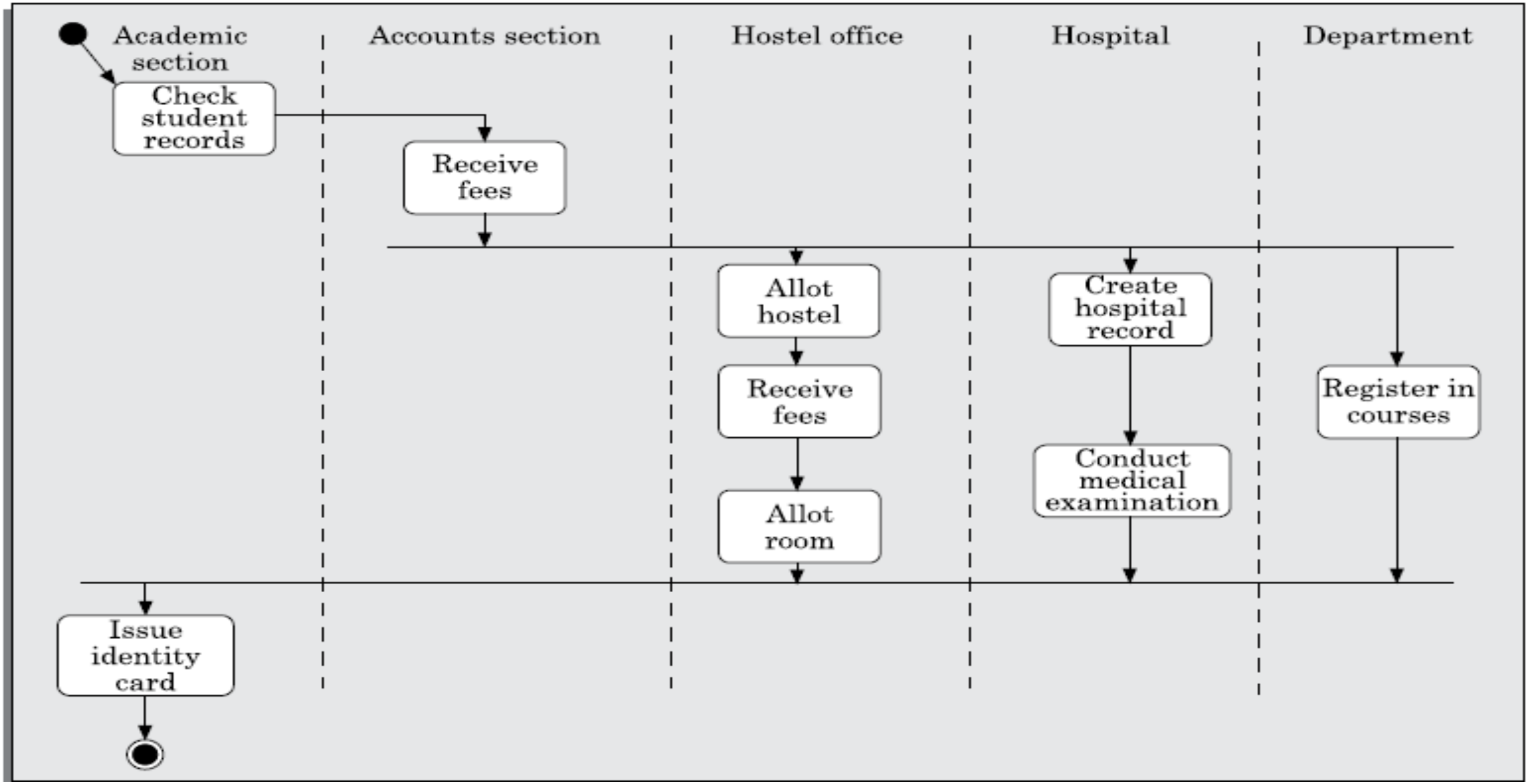


FIGURE 6.29 Activity diagram for student admission procedure at IIT.

✅ Swim Lanes 🏊

- Swim lanes divide the diagram **vertically** or **horizontally**.
- Each lane corresponds to an **entity**, **department**, or **system component** responsible for performing a set of activities.
- **Helps clarify**:
  - Who does what
  - Responsibility distribution
  - Interactions between organizational roles or software components

Example:

In a **student admission process** at IIT:

- **Accounts Section**: Collects fees
- **Hostel Office**, **Hospital**, **Department**: Carry out parallel tasks
- **Academic Section**: Issues ID card after synchronization

✅ Why Use Activity Diagrams?

Purpose	Benefit
Business Process Modelling	Helps visualize complex organizational workflows
Requirements Analysis	Aids in identifying important tasks during early system design
Design Aid	Helps in preparing sequence and collaboration diagrams later
Clarifies Parallelism	Shows simultaneous tasks using fork and join nodes
Responsibility Assignment	Swim lanes identify responsible components/entities

✅ Activity Diagram vs Flowchart

Activity Diagram	Flowchart
Supports parallelism	No support for parallel paths
Used in object-oriented design (UML)	Procedural programming focus
Includes swim lanes, fork/join	Lacks formal constructs for concurrency

6.8 STATE CHART DIAGRAM

✅ What Is a State Chart Diagram?

A **state chart diagram** models the **life cycle (state changes)** of a **single object** in response to **events** over time.

- It focuses on how **an object behaves** across **multiple use case executions**.
- Best used when **tracking the changing states** of an object.
- **Not suitable** for modeling **interactions between multiple objects** (use sequence or collaboration diagrams for that).

✅ State Chart vs FSM (Finite State Machine)

FSM	UML State Chart
Traditional formalism	Improved hierarchical model
Faces state explosion	Reduces complexity with composite (nested) states
Only events cause transitions	Allows activities within states and actions on transitions

✅ Key Components of a State Chart Diagram

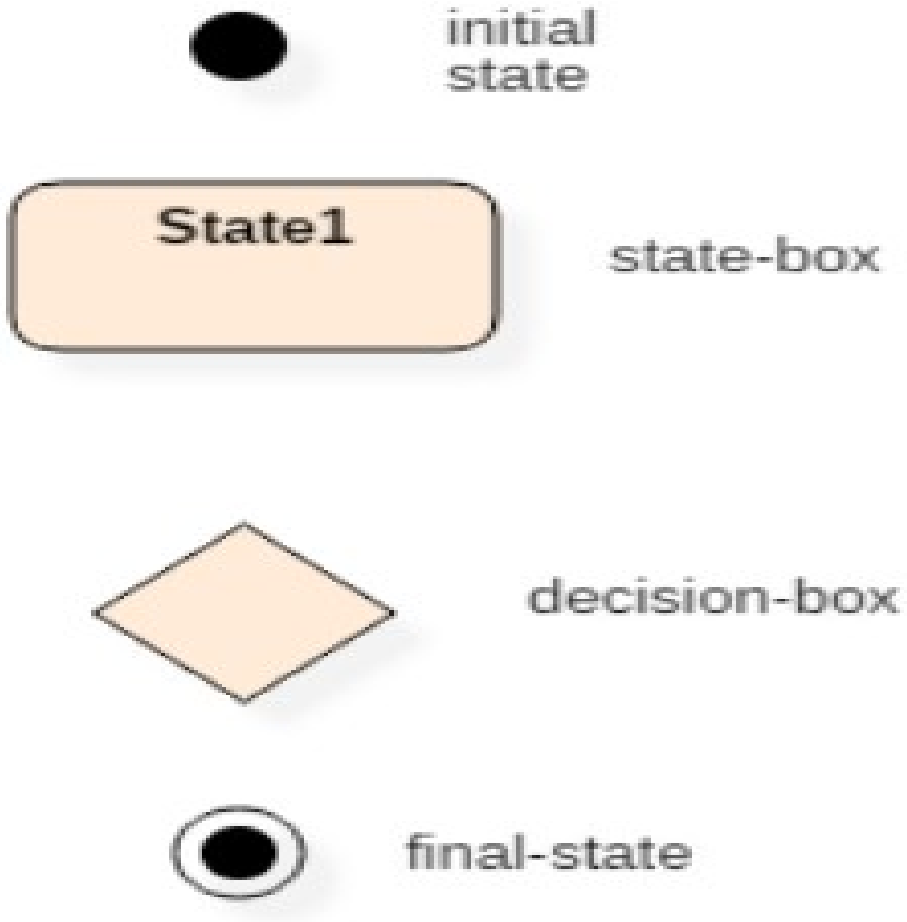
Component	Description
Initial State	Start of the state machine. Represented by a filled black circle (●).
Final State	End of the state machine. Represented by a bullseye (◎).
State	A condition during the life of an object. Shown as a rounded rectangle.
Transition	Arrow showing movement between states. Triggered by an event, may include a guard condition and an action.

✅ Transition Syntax

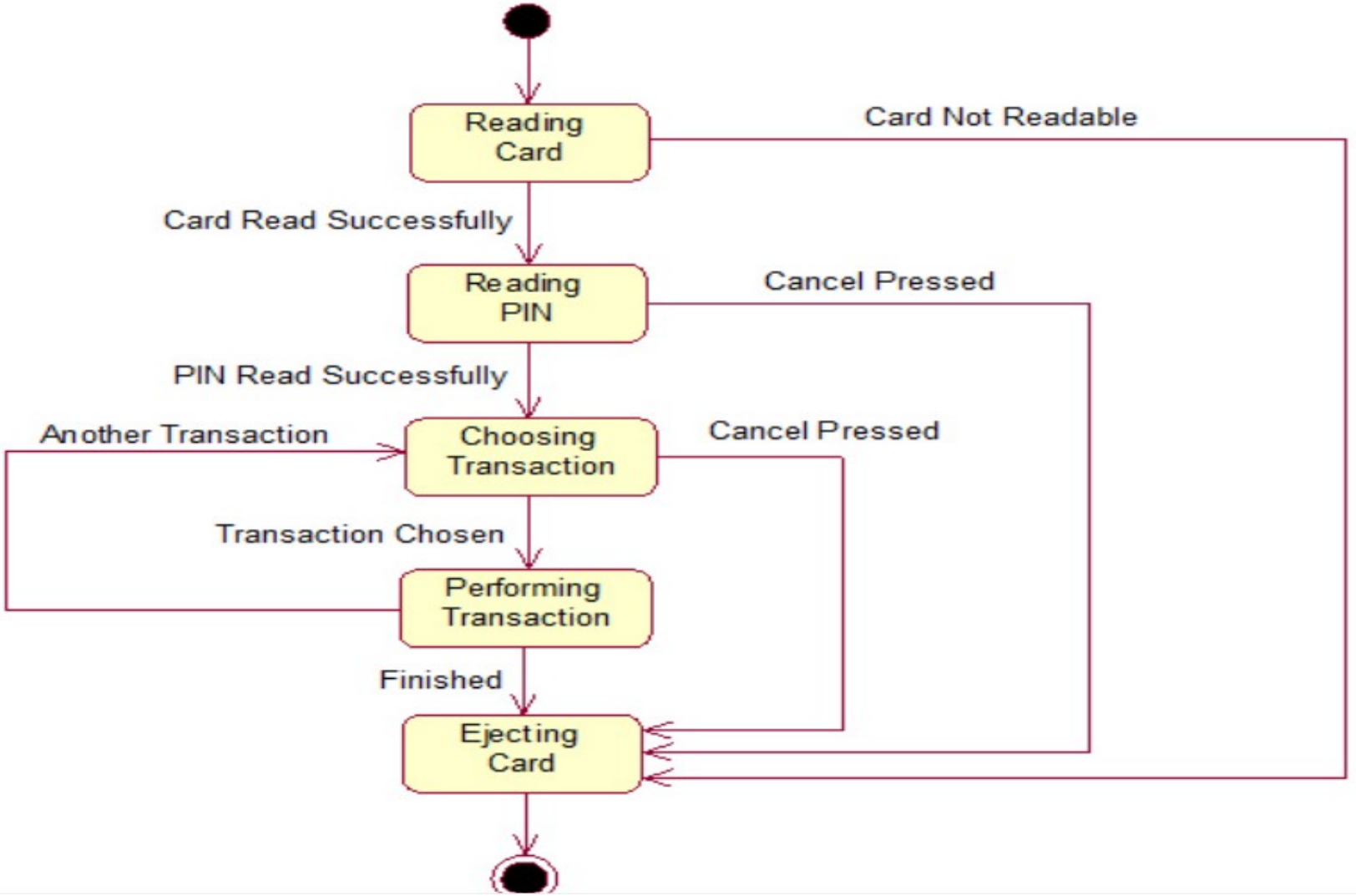
[guard] event / action

- **event** – something that triggers a transition.
- **guard** – a condition that must be true for the transition to occur.
- **action** – operation executed during the transition (assumed to be instantaneous).

Notation and Symbol



State Diagram - **ATM application**



An example state chart model for the order object of the Trade House Automation software is shown in Figure 6.30. Observe that from the Rejected order state, there is an automatic and implicit transition to the end state. Such transitions which do not have any event or guard annotated with it are called *pseudo transitions*.

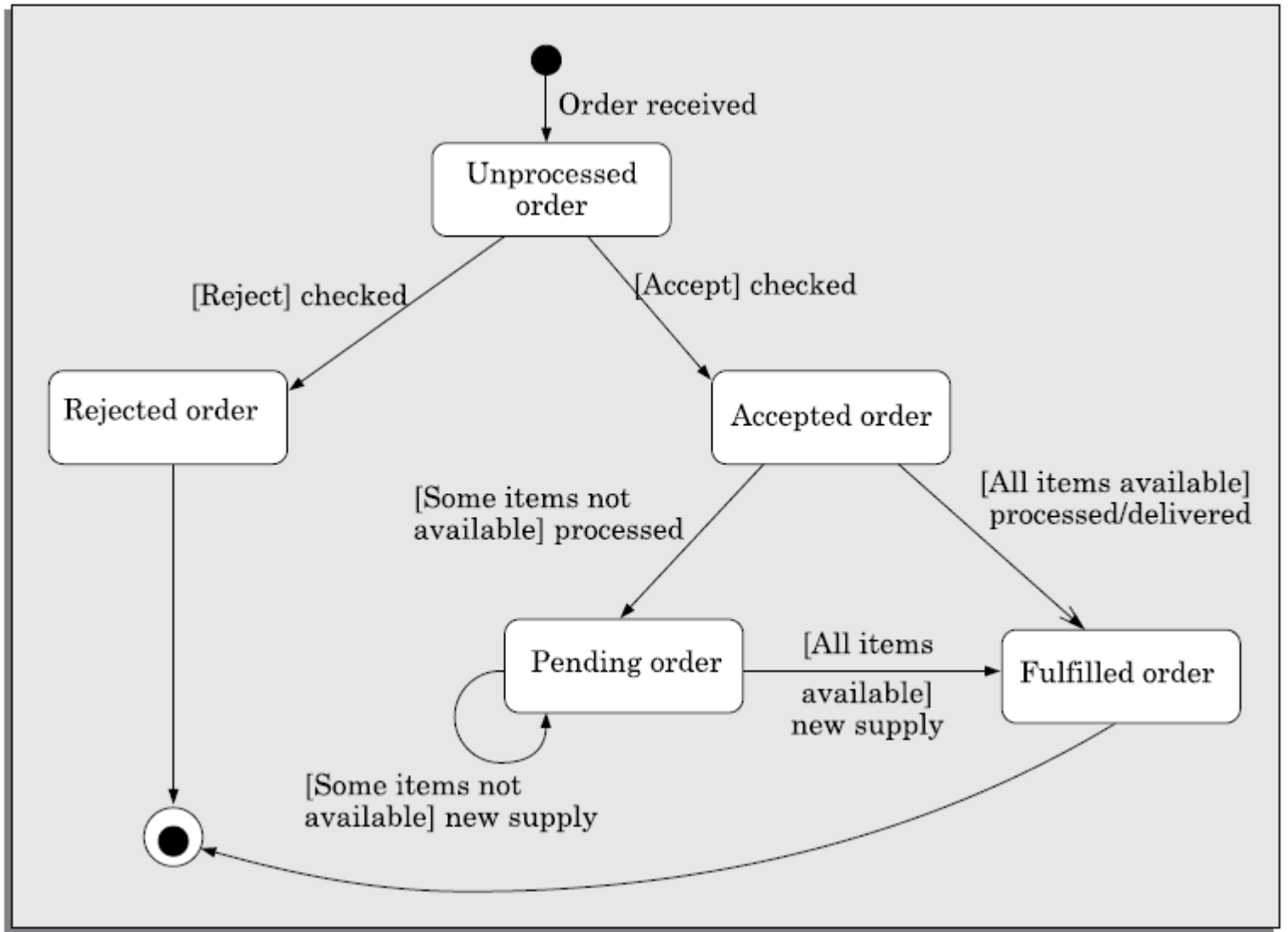


FIGURE 6.30 State chart diagram for an order object.

Difference between Activity diagram and State chart diagram

- Both activity and state chart diagrams model the dynamic behavior of the system.
- Activity diagram is essentially a flowchart showing flow of control from activity to activity.
- A state chart diagram shows a state machine emphasizing the flow of control from state to state.

✅ Activities vs

Actions Aspect	Action	Activity
Execution time	Instantaneous	Takes time
Occurrence	During transitions	During states
Interruptible?	No	Yes (by events)

✅ Special Transitions

- **Pseudo Transitions**: Transitions with **no event or guard** (e.g., automatic exit from a rejected state to final state).

✅ Example: Order Processing

In the **Trade House Automation Software**, an **Order object** may go through states such as:

New → Processing → Shipped → Delivered
↓
Rejected → (Final)

❖ If the order is **rejected**, there is an **automatic pseudo transition** to the end state—no event required.

When to Use a State Chart Diagram?

Use Case	Reason
Modeling state-dependent behavior	e.g., order status, ATM card state, document review cycle
Tracking events over time for one object	Ideal for lifespan modeling of business entities
Complex state logic with interruptions, activities, and guarded transitions	Reduces confusion with clear visuals
Alternative to FSMs when state explosion is a problem	Use hierarchical states to simplify

- State chart diagrams **focus on a single object's dynamic behavior** over time.
- They show **how an object transitions from one state to another** based on events, conditions, and actions.
- They use **guards**, **activities**, and **pseudo transitions** for accurate modeling.
- **UML state charts** are **powerful and cleaner** alternatives to traditional FSMs for real-world software design.

6.9 POSTSCRIPT

- UML has become a **widely accepted modeling language** for object-oriented design. While we focused on the **core UML elements** for moderate software design, several advanced diagrams and enhancements in **UML 2.0**.

📦 7.9.1 PACKAGE, COMPONENT, AND DEPLOYMENT DIAGRAMS

📁 Package Diagram

- **Purpose**: Group related UML elements (e.g., classes, use cases).
- **Usage**:
  - Organizes large projects into manageable modules.
  - Shows **dependencies** among different packages.
- **Structure**: Represented using folder-like symbols.
- **Note**: Packages can be **nested**.

An example of a package diagram has been shown in Figure 6.31. Note, that a package may contain further packages.

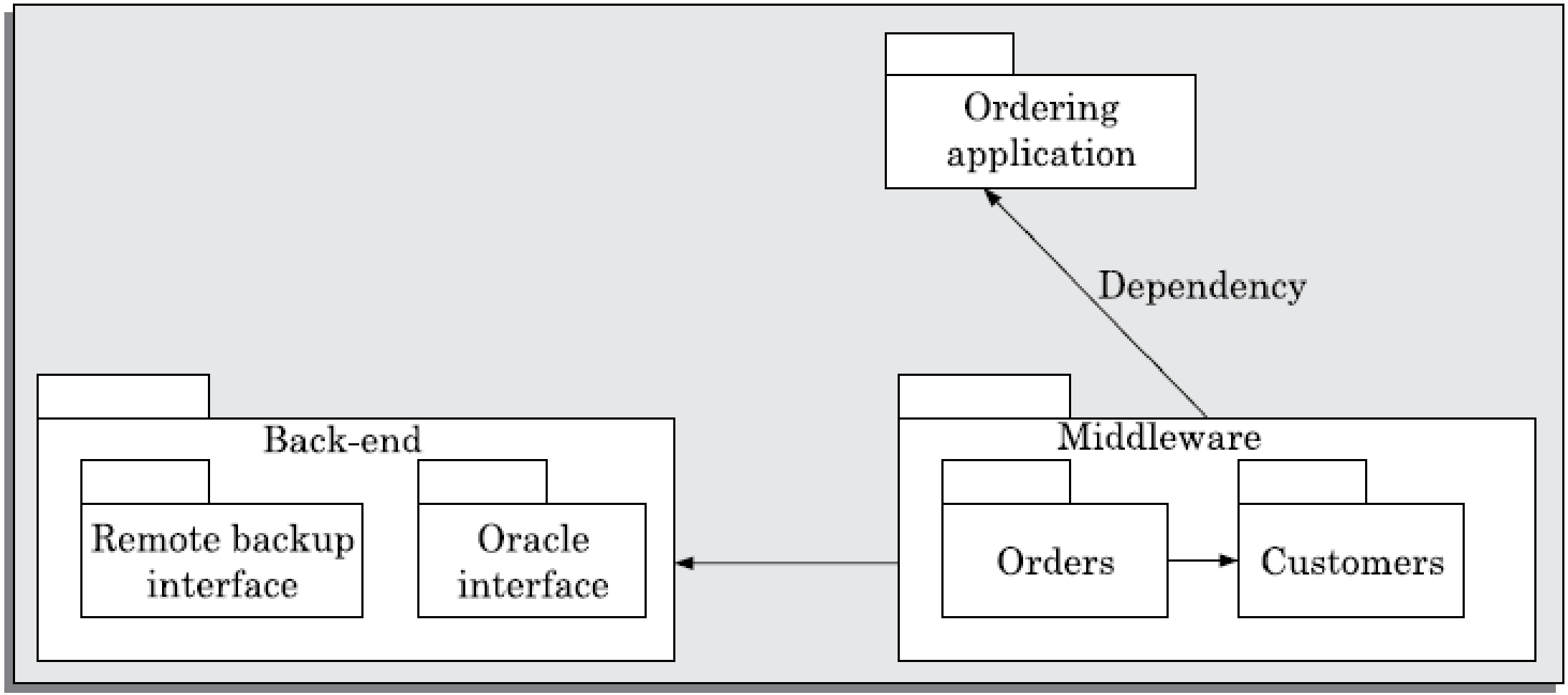


FIGURE 6.31 An example package diagram.

Component Diagram

- **Purpose**: Represents **independent software components** (modules).
- **Usage**:
  - Organize source code for **release and integration**.
  - Shows **dependencies** among components.
  - Useful in **large systems** or **plug-in architectures**.

Deployment Diagram

- **Purpose**: Shows how software is **physically deployed** on hardware.
- **Usage**:
  - Maps **components** to **nodes** (hardware elements).
  - Describes **run-time architecture** of distributed systems.
  - Essential for **operation & maintenance** teams.

6.9.2 UML 2.0

✅ Why UML 2.0?

UML 1.x was limited in:

- Modelling **concurrent/asynchronous behaviour**
- Representing **ports, events, timing**
- CASE tool interoperability using **XML Metadata Interchange (XMI)**

✅ UML 2.0 Diagram Categories

Category	Diagrams
Structure Diagrams	Class, Object, Component, Package, Deployment, Composite Structure
Behavior Diagrams	Use Case, Activity, State Machine
Interaction Diagrams	Sequence, Communication (formerly Collaboration), Timing, Interaction Overview

🔁 Combined Fragments (New in UML 2.0)

Used in **sequence diagrams** to model:

- Conditions
- Loops
- Concurrency
- Critical sections

📌 Structure:

- **Fragment**: Box containing interaction (e.g., message sequence)
- **Guard**: Boolean condition ([condition])
- **Operator**: Defines behavior of fragment

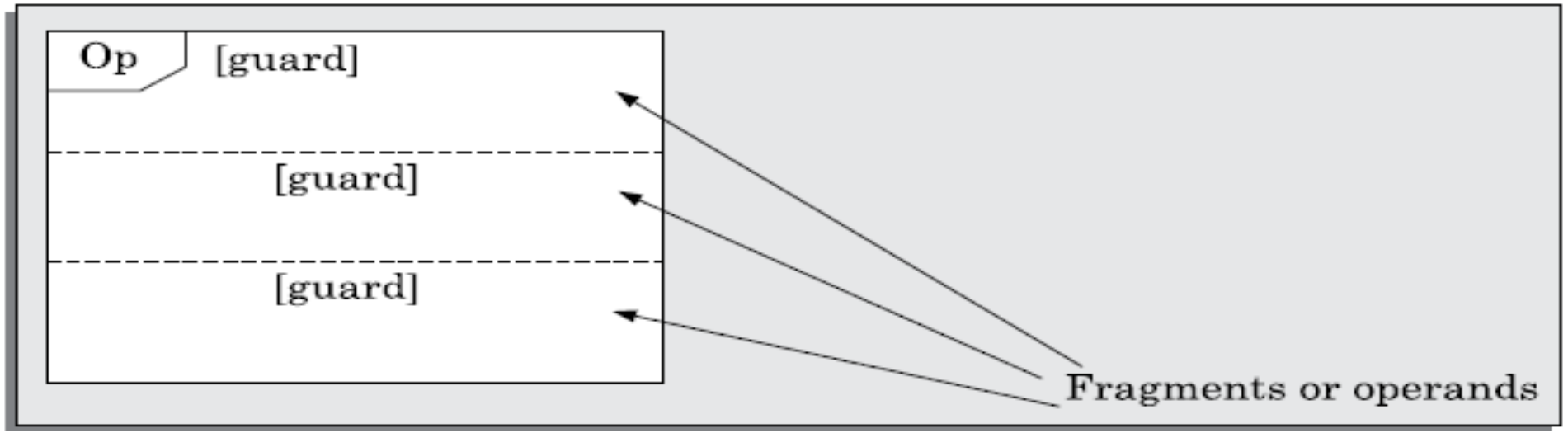


FIGURE 6.32 Anatomy of a combined fragment in UML 2.0.

Operator	Meaning
alt	If-else (alternate paths)
opt	Optional execution (if true)
par	Parallel execution
loop	Looping (iteration based on guard)
region	Critical region (single-thread access)

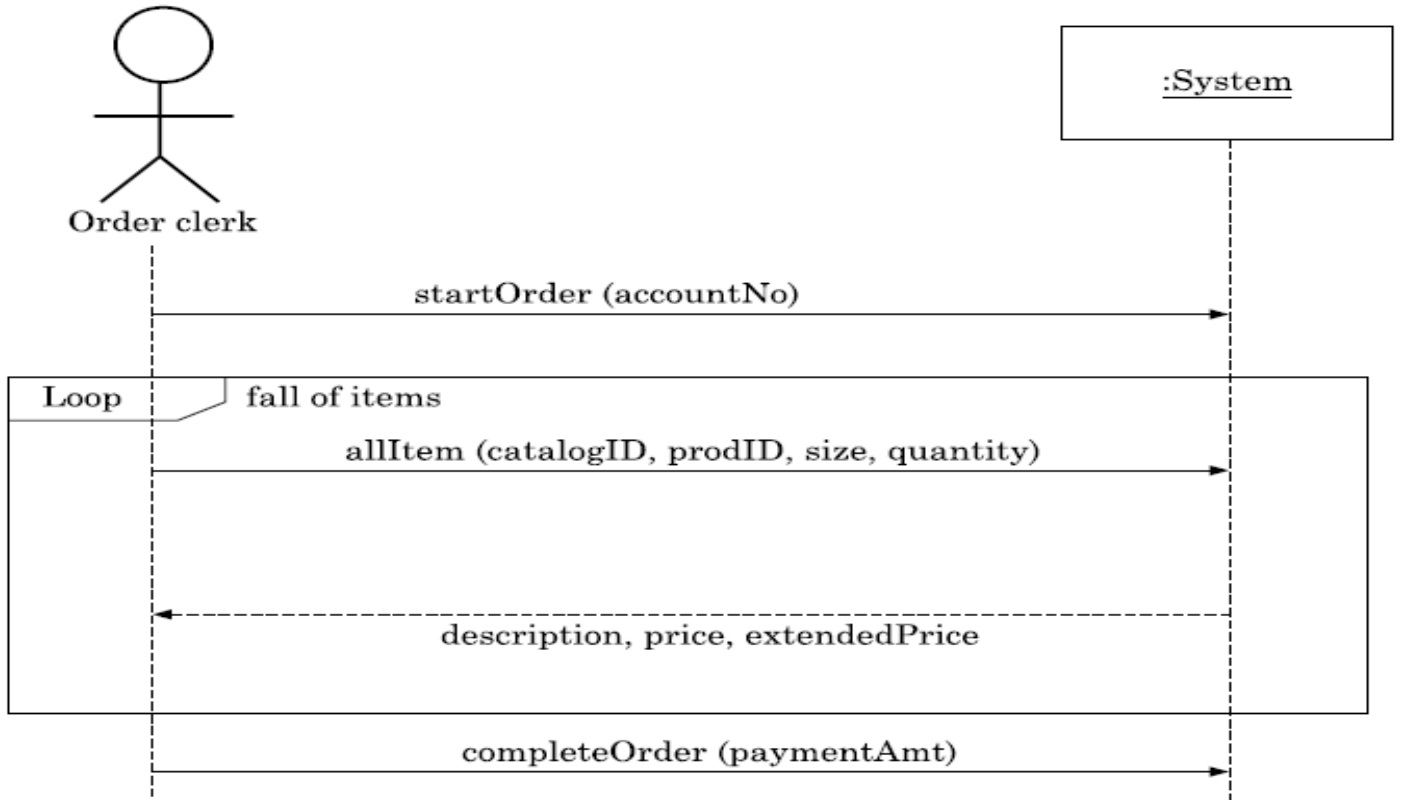


FIGURE 6.33 An example sequence diagram showing a combined fragment in UML 2.0.

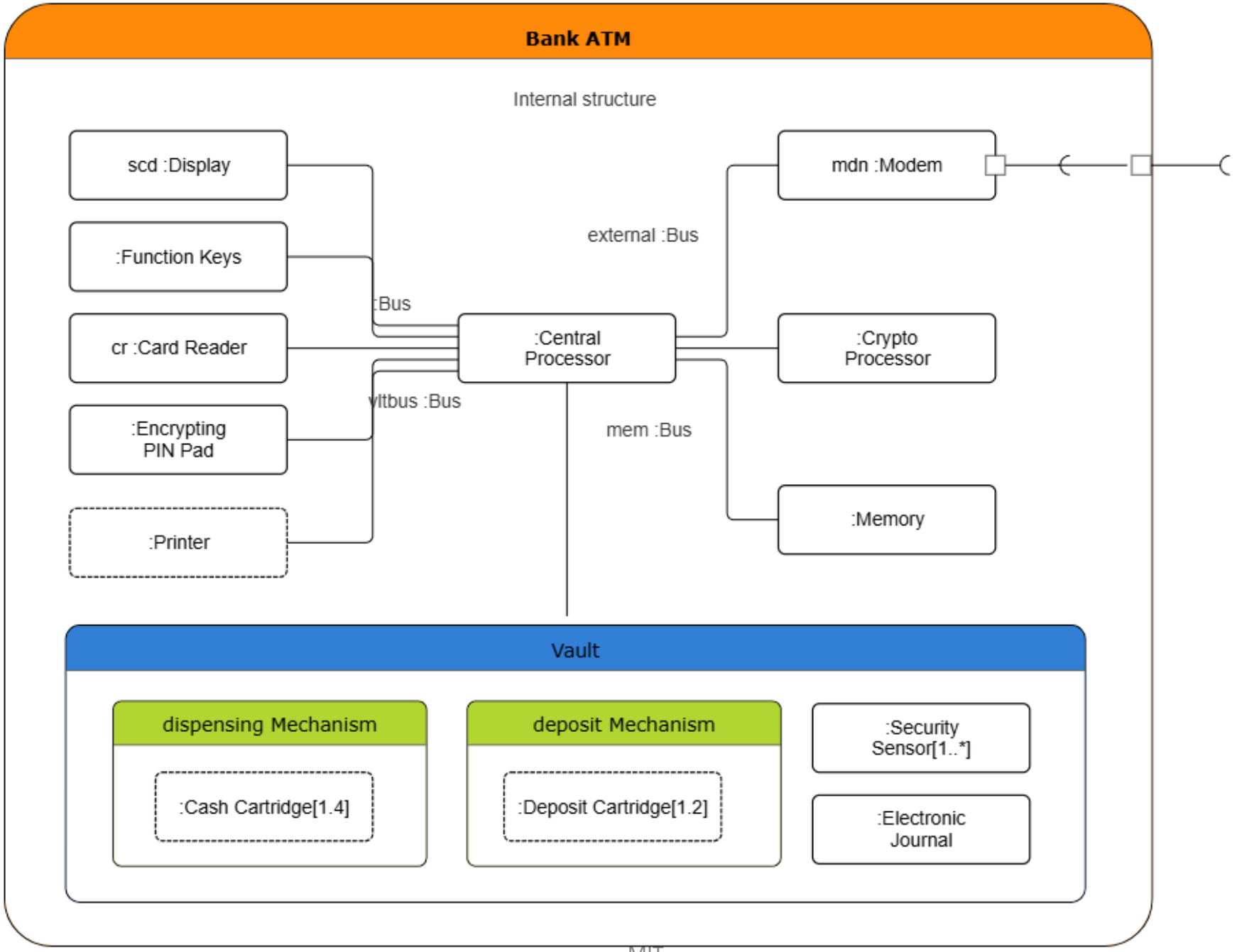
Composite Structure Diagram (New)

Models the **internal structure** of a class using:

Element	Description
Part	Represents internal components (sub-objects) of a class
Port	Communication endpoints that can send/receive signals
Connector	Represents communication links between parts/ports

Useful in **embedded systems**, **telecom**, and **real-time applications**.

- UML has become essential for object-oriented design.
- UML 2.0 introduced key features for **modern software domains** (embedded, telecom).
- New diagrams like **combined fragments** and **composite structure diagrams** enhance modeling power.
- Advanced diagrams (package, component, deployment) aid **system architecture planning**, **modular development**, and **physical deployment**.


Bank ATM
Internal structure
scd :Display
:Function Keys
cr :Card Reader
:Encrypting PIN Pad
:Printer
:Bus
vltbus :Bus
:Central Processor
external :Bus
mdn :Modem
:Crypto Processor
mem :Bus
:Memory
Vault
dispensing Mechanism
:Cash Cartridge[1.4]
deposit Mechanism
:Deposit Cartridge[1.2]
:Security Sensor[1..*]
:Electronic Journal

6.10 INTRODUCTION TO PATTERNS

📌 What Are Design Patterns?

- **Design patterns** originated from architecture and are now applied in **object-oriented analysis and design (OOAD)**.
- A **design pattern** is a commonly accepted, **reusable solution** to a frequently occurring sub problem in software design.
- Patterns help in **systematic reuse**, enhancing **design efficiency** and **reducing design iterations**.

6.10.1 Basic Pattern Concepts

- Real-world design problems often contain **repeating subproblems**.
- **Patterns document** the **problem**, **context**, and **solution**, so they can be reused and adapted across different projects.
- They **do not imply black-box reuse**; instead, solutions must be tailored per project.

🗣️ ***Christopher Alexander*: "Use a pattern solution a million times over, without ever doing it the same way twice."**

✨ Benefits of Knowing Patterns:

- Helps avoid **over-engineered**, **inflexible**, or **inefficient** solutions.
- Enables designers to **recognize problems faster** and apply proven solutions efficiently.
- Patterns help **transfer design knowledge** and establish a **common design vocabulary**.

A Pattern Typically Includes:

1. The **problem**
2. The **context** where it occurs
3. The **solution**
4. **Where** the solution **works** and **fails**

6.10.2 Types of Patterns

Architectural Patterns

- **High-level design patterns**
- Help in defining **system structure** and subsystems
- Example: **MVC (Model-View-Controller)**

Design Patterns

- Focus on **structuring classes and objects**
- Define **roles, relationships**, and **collaborations**
- Used at **medium-level design**

Idioms

- **Low-level**, **language-specific** programming patterns
- Comparable to idioms in natural language
- Improve **code quality**, **readability**, and **reusability**
- Example in C: for(i = 0; i < 100; i++)

🔁 Comparison Table:

Type	Level	Purpose
Architectural	High	Structure of large systems
Design	Medium	Class-level structure and collaboration
Idioms	Low	Language-specific coding best practices

🔄 6.10.3 More Pattern Concepts

🔍 Patterns vs. Algorithms

Aspect	Patterns	Algorithms
Focus	Maintainability, understandability	Efficiency, space/time optimization
Concerned with	Reuse in design	Reuse in computation
Usage	Subjective and structural	Objective and procedural

✅ Pros of Design Patterns

- Provide a **shared vocabulary** for designers
- Help **capture expert knowledge**
- Make designs **flexible, maintainable, efficient**
- Guide toward **good design decisions**
- **Reduce design iterations** and speed up development

❌ Cons of Design Patterns

- Do **not lead to direct code reuse**
- No strict **methodology to choose the right pattern**
- Heavily rely on **designer's intuition and experience**

Antipatterns

🔁 **Antipattern**: A common but **ineffective or counterproductive solution** that seems good initially.

Common Antipatterns:

Name	Problem
Input kludge	No input validation
Magic pushbutton	Business logic embedded in UI code
Race hazard	Failing to handle concurrent event ordering properly

Antipatterns help **recognize bad solutions** and **avoid repeating mistakes**.

- **Design patterns** provide time-tested solutions to **recurring design problems**.
- They enable **reuse**, **faster development**, and **higher-quality software**.
- Knowing patterns, **their appropriate use**, and also **antipatterns** is crucial for becoming an effective software designer.
- In this, focus is on **design patterns**, not on architectural or idiomatic ones.

6.11 OBJECT-ORIENTED ANALYSIS AND DESIGN (OOAD) METHODOLOGY

6.11.1 Unified Process

The **Unified Process (UP)** is a widely adopted **object-oriented development model**.

🔁 Characteristics:

- **Iterative and incremental** lifecycle
- **Use case-driven** (user requirements guide the process)
- All models must **conform to the use case model**

📌 Phases of Unified Process:

Phase	Description
Inception	Define project scope and initial requirements. Create prototypes if needed.
Elaboration	Analyze functional/non-functional requirements. Develop initial use cases and domain model.
Construction	Perform design, coding, and testing in short iterations. Use cases drive development.
Transition	Deploy product in the real environment and support users.

As shown in Figure 6.34, the unified process involves iterating over the following four distinct phases as follows:

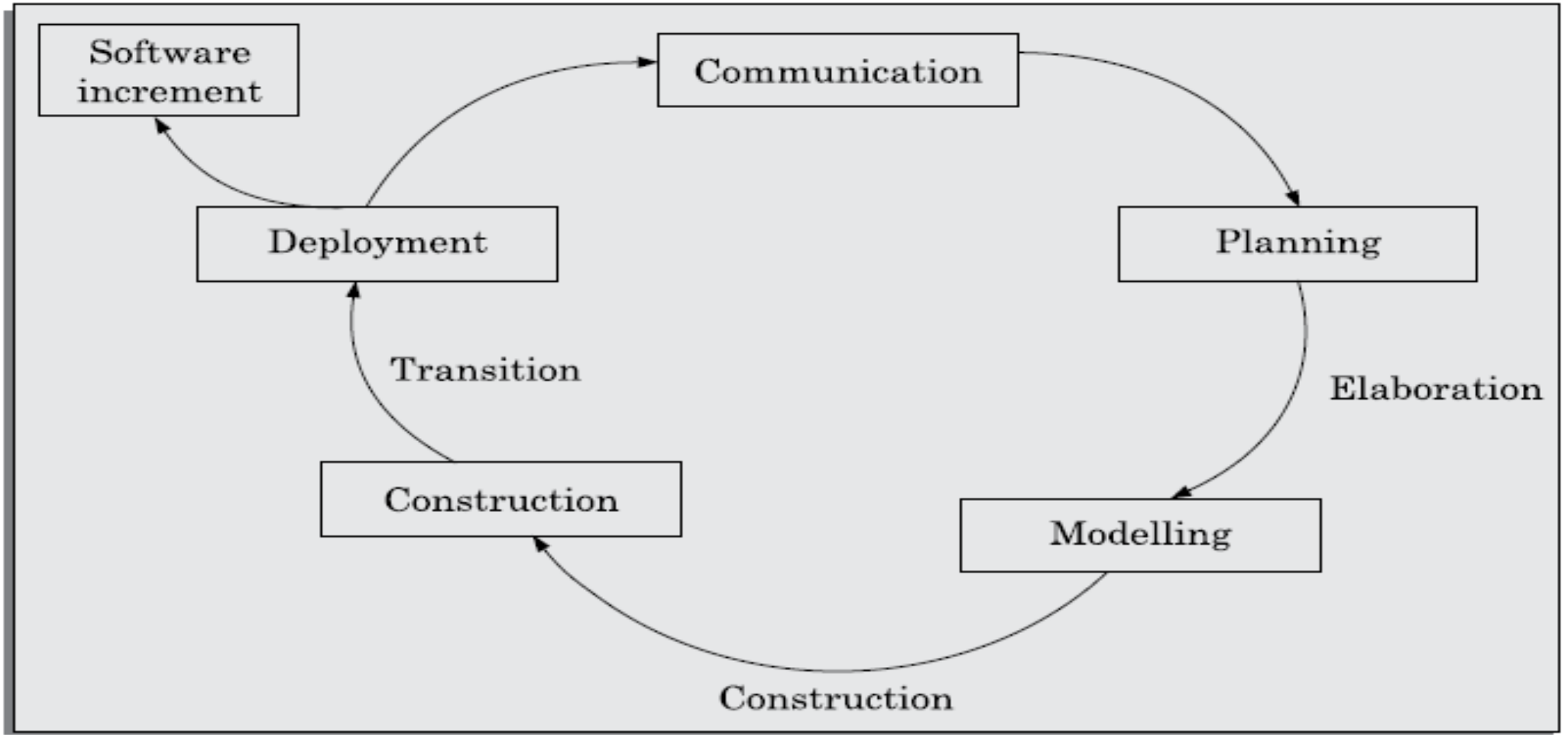


FIGURE 6.34 Unified process model.

📌 **Design methodology discussed here is applied mainly during Elaboration and Construction phases.**

6.11.2 Overview of the OOAD Process

OOAD Workflow:

1. **Use Case Model** – Captures user-visible functionalities.
2. **Domain Model** – Identifies important concepts from use cases and SRS.
3. **Class Diagram** – Iteratively refined from the domain model via interaction diagrams.
4. **Code Generation** – Many tools can auto-generate code from class diagrams.
5. **Glossary Maintenance** – Ongoing activity for clear communication across team.

The use case model plays a crucial role in the design process, and needs to be developed first. As shown in Figure 6.35, the domain model is constructed next through an analysis of the use case model and the SRS document. The domain model is refined into a class diagram through a number of iterations involving the interaction diagrams. Once the class diagram has been constructed, it can easily be translated to code. Many CASE tools support generation of code skeleton from the class diagram.

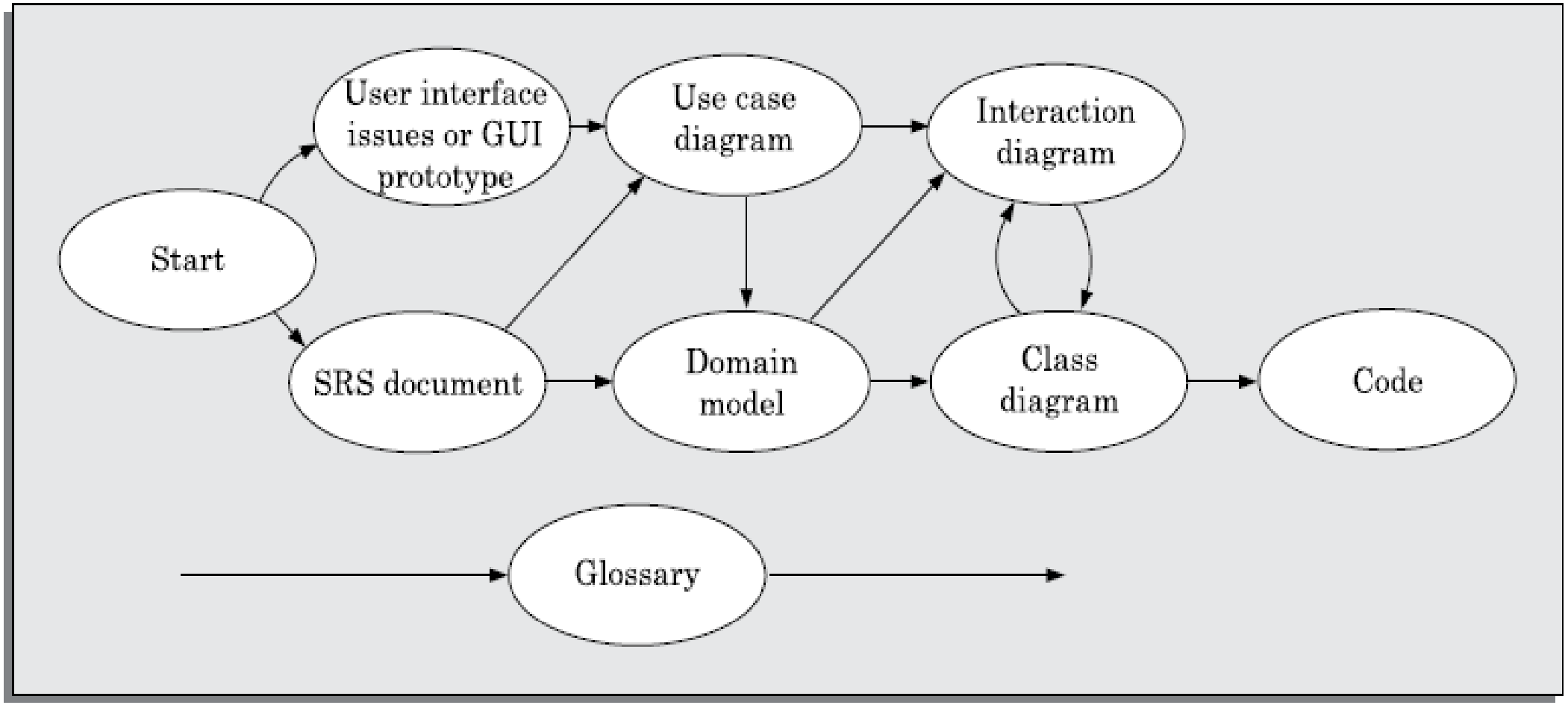


FIGURE 6.35 An object-oriented analysis and design process.

📘 Glossary

A **project glossary** is a dictionary of domain terms and concepts. It helps avoid ambiguity and improves communication.

📖 6.11.3 Use Case Model Development

📌 What Is a Use Case?

- A **use case** describes a complete interaction between a user and the system.
- Represents **user-level** functional requirements.
- Each use case should include a **textual description** with all possible **scenarios** (main and alternative).

How to Identify Use Cases:

- Use **SRS functional requirements** as source.
- Map **top-level GUI options** to **use case packages**.
- Structure use cases similar to **user manual chapters** or **GUI menus**.

Example: In a word processor:

- **File**, **Edit**, **View** menu options → **Use case packages**
- Functions like **Save**, **Print**, **Cut**, **Paste** → Individual use cases

✏️ Naming Use Cases:

- Use **present-tense verb phrases**, e.g., admit patient, print bill
- Avoid noun phrases like Patient Admission, Bill Printing

🛑 Common Mistakes in Use Case Modeling:

Mistake	Explanation
Clutter	Too many use cases in a single diagram—group them into packages
Too Detailed	Confusing steps of a use case as separate use cases (e.g., Print Receipt as a separate use case from Withdraw Cash)
No Text Description	Omitting use case descriptions makes understanding and design difficult
Missing Alternate Flows	Failure to capture alternative scenarios leads to incomplete or buggy systems

6.11.4 Domain Modelling (Conceptual Modelling)

Domain modelling involves identifying **real-world concepts** from the problem space to design a **conceptual class diagram** (first-cut class diagram).

Three Types of Objects in Domain Model:

Type	Description	Examples
Entity Objects	Represent real-world entities. Store data permanently.	Book, Library Member, Book Register
Boundary Objects	Interface between system and actor (user). Handle user inputs/outputs.	IssueBookUI, Login Screen, Menu Form
Controller Objects	Coordinate the use case behavior and system logic.	IssueBookController, RenewBookController

- Domain model shows **only class names**, not attributes/methods (those are added later).
- The **crux** of domain modeling is the **correct identification of entity objects**.
- Relationships between objects should also be identified (associations).

🎯 Boundary Objects

- Represent **screens, forms, menus, GUIs**.
- Linked to **actors** and help in reading input, validating, formatting, and displaying output.
- Earlier known as **interface classes**.
- Should be **independent of GUI technologies** (e.g., don't use terms like radio Button).

➕ Tip:

- Use **one boundary class per actor/use case pair**.
- Can later split/merge based on complexity.

Entity Objects

- Store **persistent data** across use case executions.
- Perform **basic operations**: store, search, update, delete.
- They are **stable** (don't change often) and act like **dumb servers**.
- Usually resemble **registers**, **data files**, or **data stores** in manual systems or DFDs.

Controller Objects

- **Coordinate** the flow of a **use case execution**.
- Decouple UI from data, manage **business logic**.
- Should be created **per use case**; possibly multiple controllers for complex cases.
- Help **track execution state** for each actor's use case invocation.

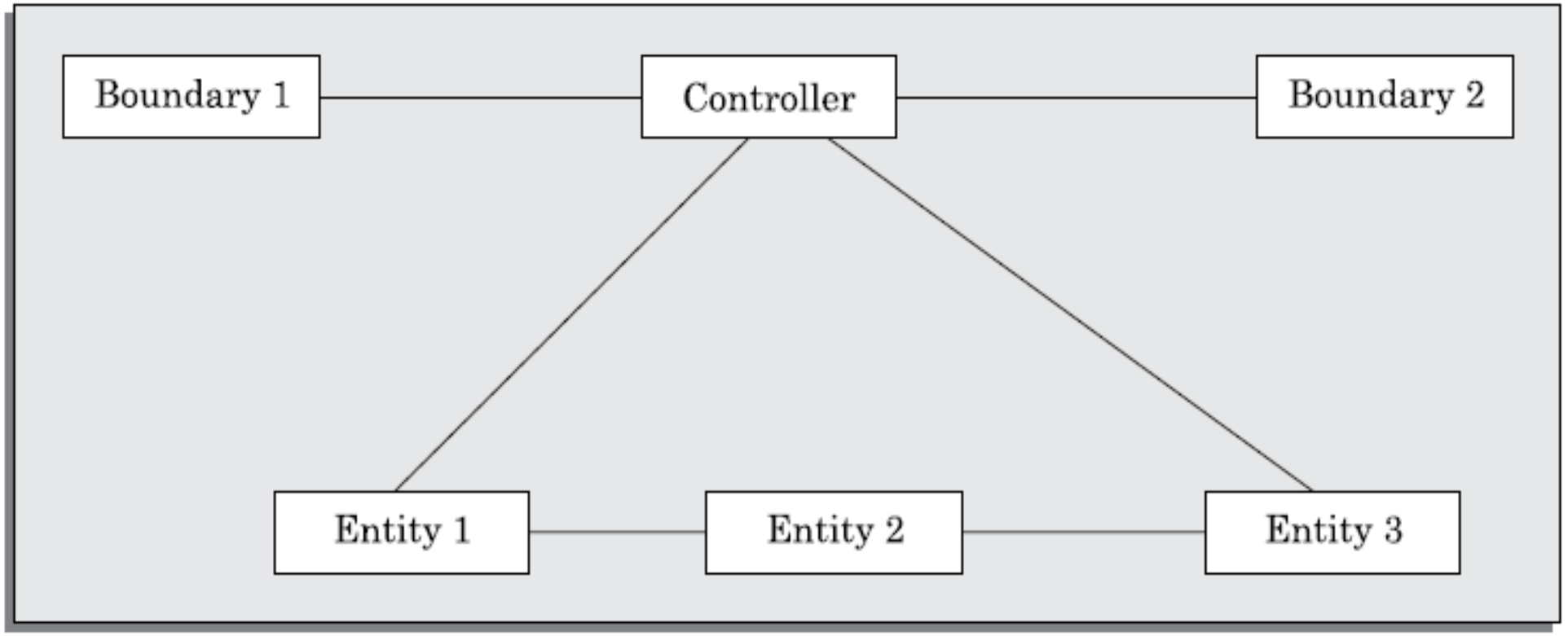
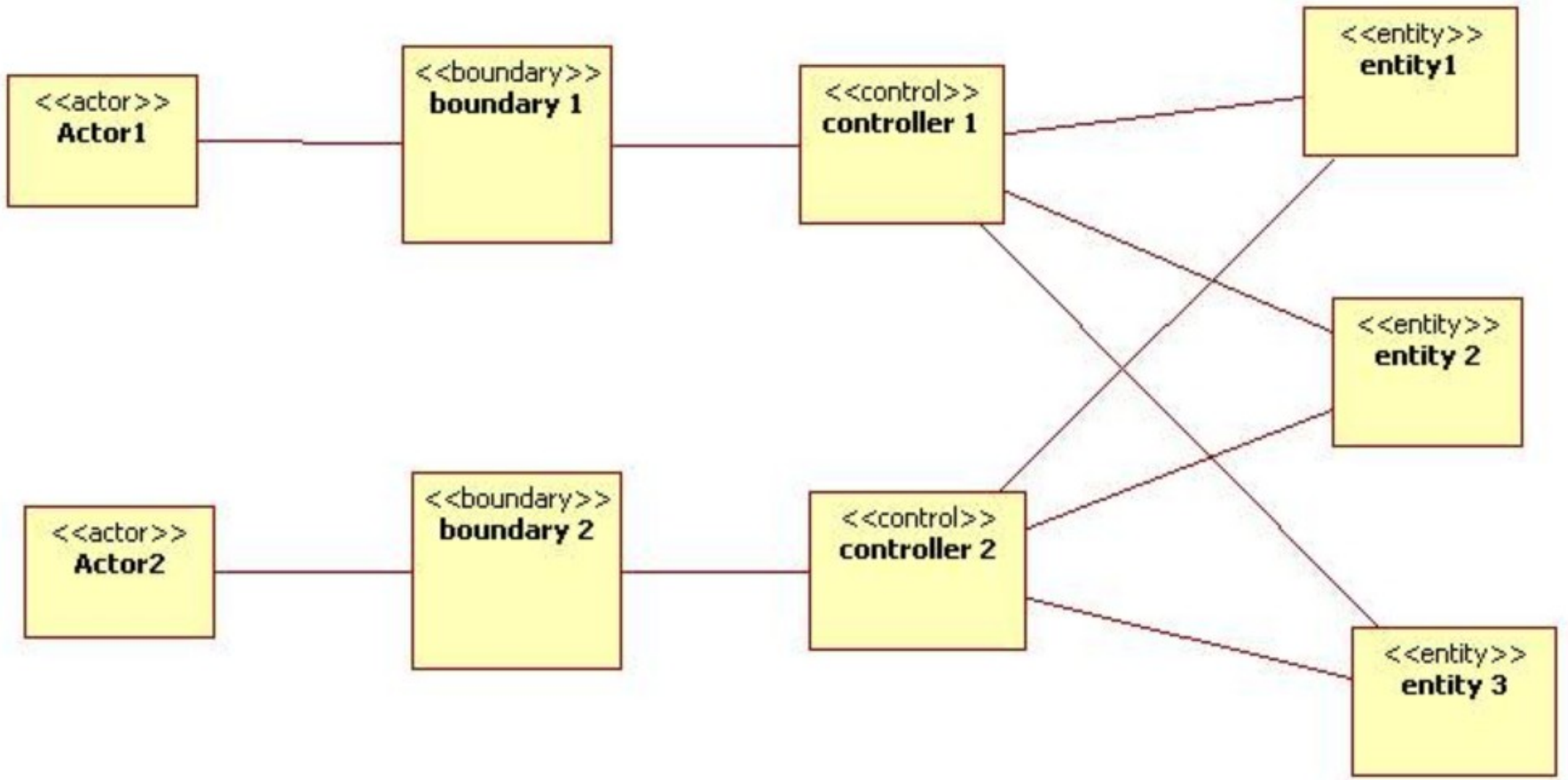


FIGURE 6.36 A typical realisation of a use case through the collaboration of boundary, controller, and entity objects.


<<actor>>
Actor1
<<boundary>>
boundary 1
<<control>>
controller 1
<<entity>>
entity1
<<entity>>
entity 2
<<actor>>
Actor2
<<boundary>>
boundary 2
<<control>>
controller 2
<<entity>>
entity 3

✅ Benefits of Controller Objects:

- Decouples boundary and entity objects.
- Helps manage business logic changes easily.
- Behaves like a **manager** directing **worker objects** (boundary/entity).

What If a Class is Missed?

- No problem: the **domain model is iterative**.
- Add missed classes later, **refine or split** overloaded ones.
- Attributes, methods, and relationships are defined **in later stages**.

6.11.5– Identification of Entity Objects

- The **quality** of the final design depends on how **well entity objects** are identified.

✅ Techniques for Identifying Entity Objects:

1. **Grammatical Analysis (Booch's Method)**
2. Derivation from **Data Flow Diagrams (DFD)**
3. Derivation from **Entity-Relationship Diagrams (ERD)**

Booch's Grammatical Analysis (1991)

Steps:

1. Write the **processing narrative** of the problem.
2. Extract all **nouns** (candidate objects) and **verbs** (candidate methods).
3. Eliminate irrelevant nouns based on the following filters:

❌ Filters to Eliminate False Nouns:

Rule	Eliminate if…	Example
Actor	It is a user, not a data object	Human Player
Synonym	It repeats an already listed noun	Computer Game ≈ Tic-Tac-Toe
Imperative Procedure Name	It is an action, not a real object	Cash Withdrawal, Move
No Attributes	Cannot store any useful data	Computer Screen
Single Attribute	Stores only one value, not meaningful on its own	Phone Number
Inconsistent Methods	Operations don't apply to all instances	Resident Student vs Non-resident Student → Use subclasses

- Look for **aggregate objects**: e.g., BookRegister, StudentRegister
- Refer to **DFD data stores**: they often map to entity classes.
- Use **manual system registers** as clues for entity classes.

Example : Tic-Tac-Toe Software

🔍 **Problem Summary:**

- Human and computer take turns marking a 3×3 board.
- Game ends when one wins or all squares are filled.

Grammatical Analysis Yields:

Tic-tac-toe, human player, move, square, mark, board, row, column, diagonal, etc.

❌ Eliminate:

- **Actor**: Human Player
- **Actions**: Move, Mark
- **No Data**: Row, Column, Square, Line

✅ Keep:

- **Entity Object**: Board

- Domain modeling is foundational for OOP design.
- Carefully identify **boundary, controller, and entity objects**.
- **Entity object identification** requires experience and intuition.
- Use **Booch's grammatical approach** as a guide, not a strict rule.

🔷 6.11.6 Interaction Modelling

📌 Purpose:

Interaction modelling shows **how objects collaborate over time** to realize the **behavior of a use case**. It helps in:

1. **Allocating responsibilities** among boundary, controller, and entity objects.
2. **Visualizing the sequence** of interactions to realize a use case.

🎯 Goals of Interaction Modelling:

- Assign **responsibilities (methods)** to the right classes.
- Capture the **temporal sequence** of messages (interactions) among objects.

📘 How it's Represented:

- Done using **UML Sequence Diagrams** (preferred).
- **Collaboration diagrams** can be generated from sequence diagrams, so often not required separately.
- **One sequence diagram per use case** is typically sufficient.

📏 Complexity Handling:

- If a use case has **multiple scenarios**, **combine them** into a single sequence diagram.
- If the diagram becomes too complex or large:
  - **Split the use case** into simpler use cases.
  - Then create **separate diagrams** for each simpler use case.

6.11.7 Class-Responsibility-Collaborator (CRC) Cards

📌 What Are CRC Cards?

CRC = **Class-Responsibility-Collaborator**

Invented by **Ward Cunningham** and **Kent Beck**.

CRC cards are **index cards** used for **team-based brainstorming** to:

- Assign **responsibilities** (functions or methods) to classes.
- Identify **collaborator classes** needed to fulfill those responsibilities.

CRC Card Layout:

Each card includes:

- **Class name** (top)
- **Responsibilities** (left column) → actions/methods
- **Collaborators** (right column) → other classes needed to perform the action

📌 **Example: CRC Card for Book Register**

Class: BookRegister

Responsibilities

Find available books

Register new issue

Update status

CRC card for the Book Register class of the Library Automation System is shown in Figure 8.11. Observe that the name of the class is written at the top of the card. The names of the methods (responsibilities to be supported) are specified in the first column. The other class methods that a method

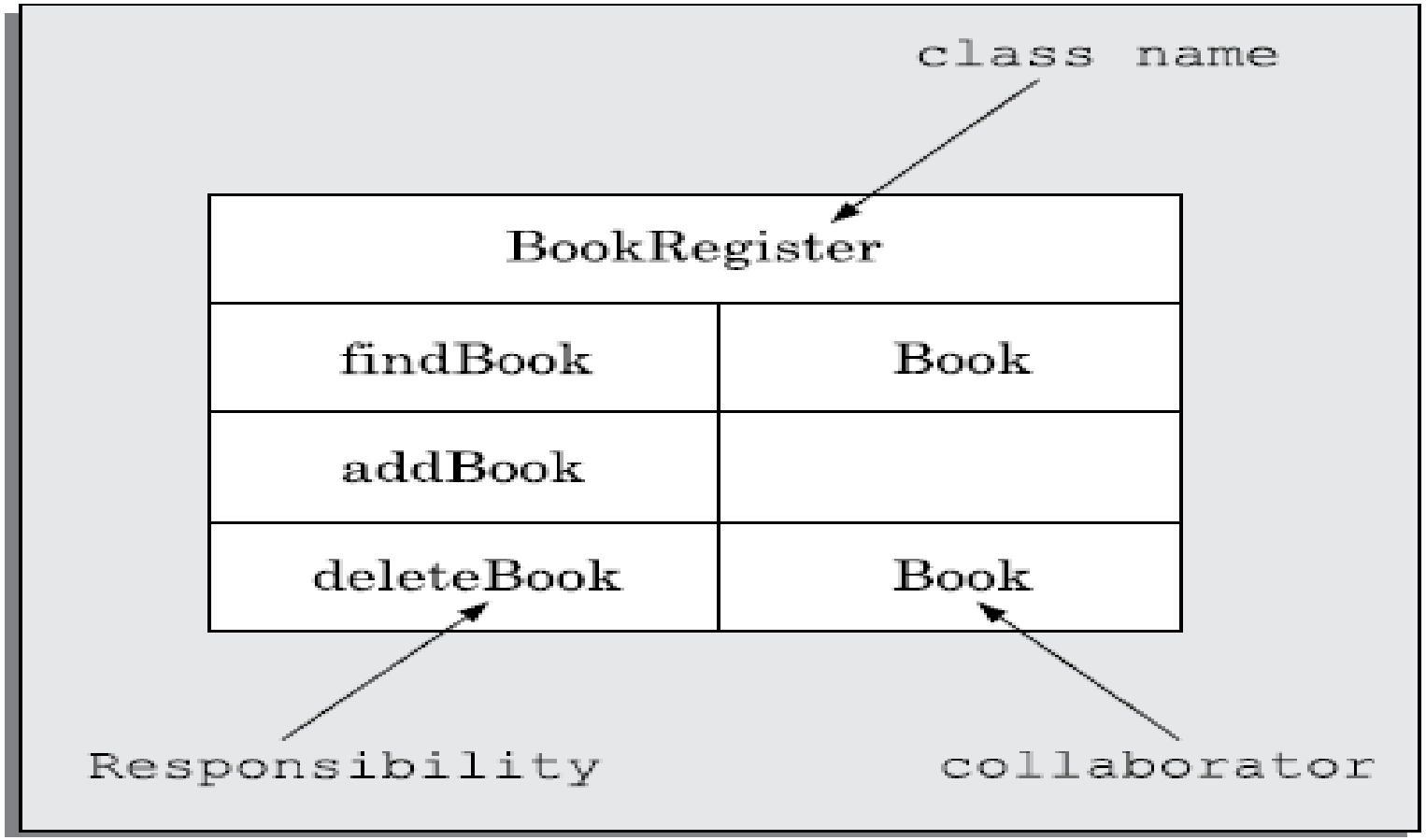


FIGURE 6.37 CRC card for the BookRegister class.

👥 How CRC Cards Are Used:

- Conducted in **small group sessions**.
- Team members **role-play** as objects/classes.
- Each holds the CRC card for their class.
- As each use case is discussed:
  - Members propose **responsibilities** their class can take on.
  - Collaborators needed are also noted.
- This simulates the **message-passing interactions** between objects.

✅ Benefits:

- **Interactive and collaborative**.
- Helps clarify the **distribution of behavior** among objects.
- Ensures every responsibility has a **clear owner**.
- Keeps class responsibilities **focused and minimal** due to card size (typically 4x6 inches).

Once CRC Cards Are Done:

- Use them to **draft interaction diagrams** (e.g., sequence diagrams).
- Cards help guide how objects **communicate** to fulfil the use case.