# Advanced Software Engineering
(CS6401)

Autumn Semester (2022-2023)

**Dr. Judhistir Mahapatro**
**Department of Computer Science and Engineering**
**National Institute of Technology Rourkela**

# Object-Oriented Software Design

# Topics covered in the previous Lecture

- Introduction to UML
- Use case diagram

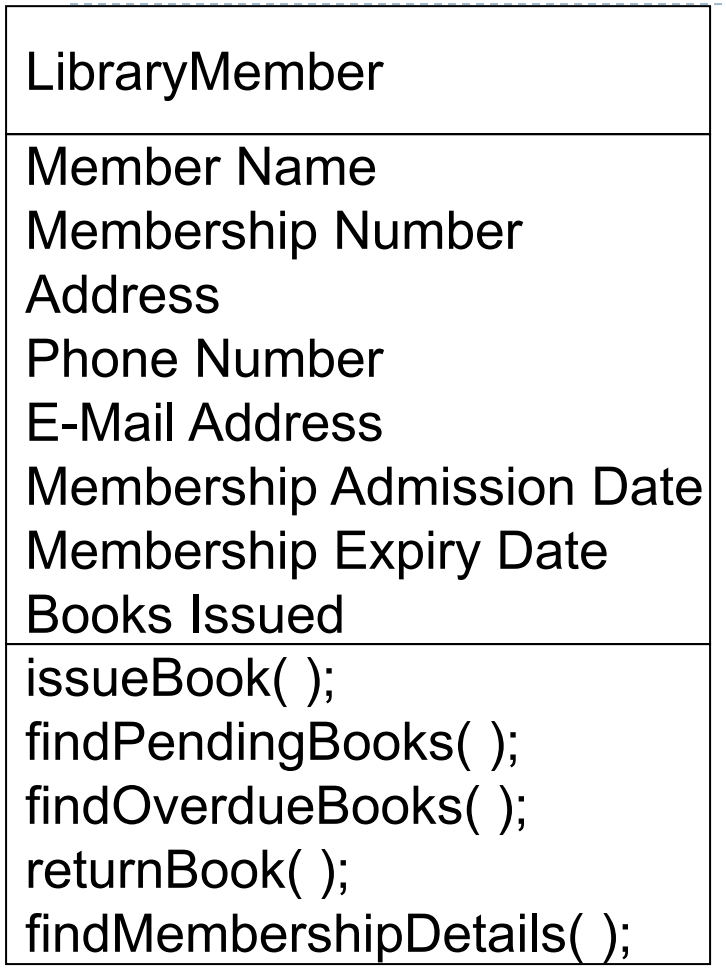
# Class diagram

- Describes static structure of a system
- Main constituents are classes and their relationships:
  - Generalization
  - Aggregation
  - Association
  - Various kinds of dependencies

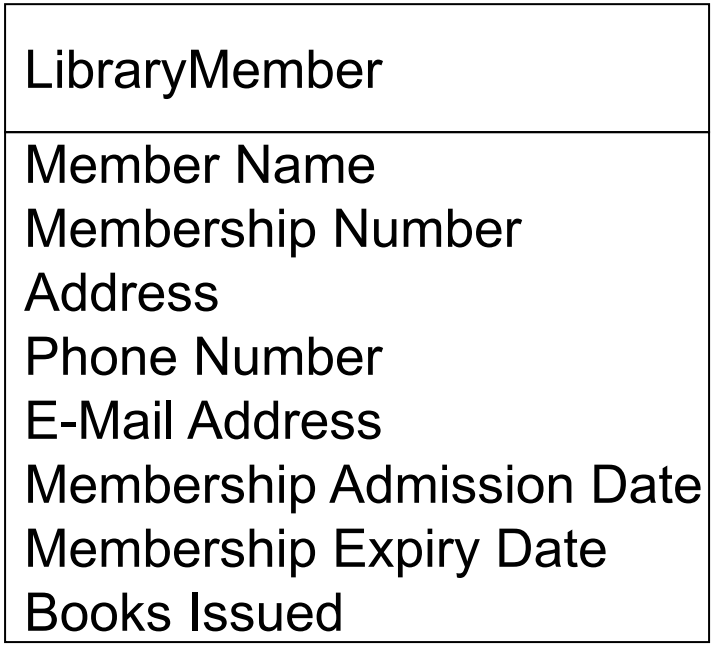
# Class diagram

- Entities with common features, i.e. attributes and operations
- Classes are represented as solid outline rectangle with compartments
- Compartments for name, attributes & operations
- Attribute and operation compartment are optional for reuse purpose

# Example of Class diagram

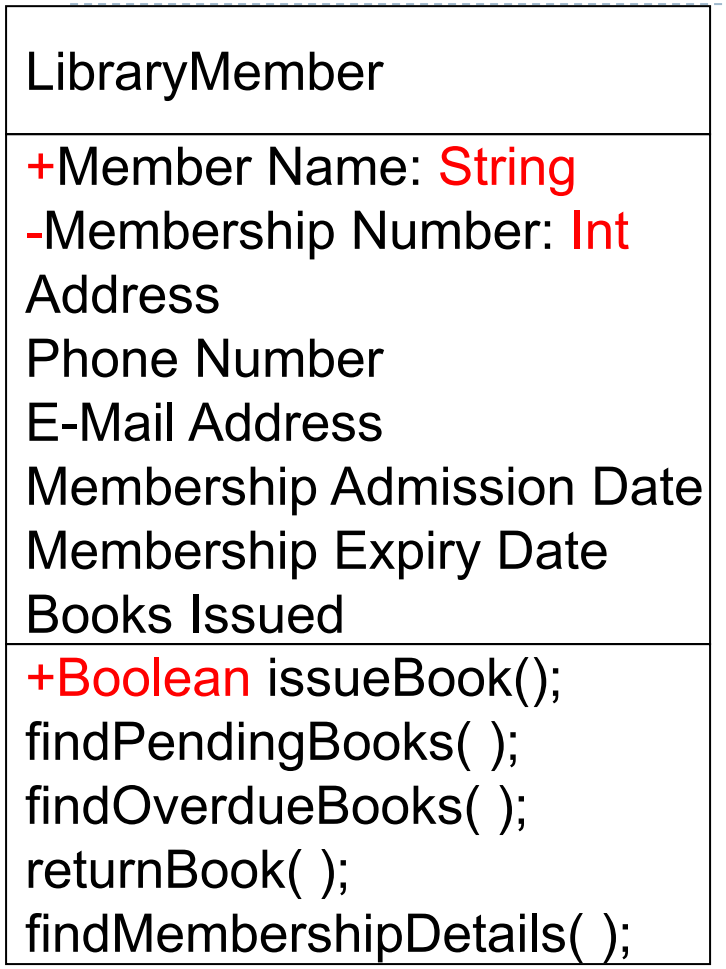




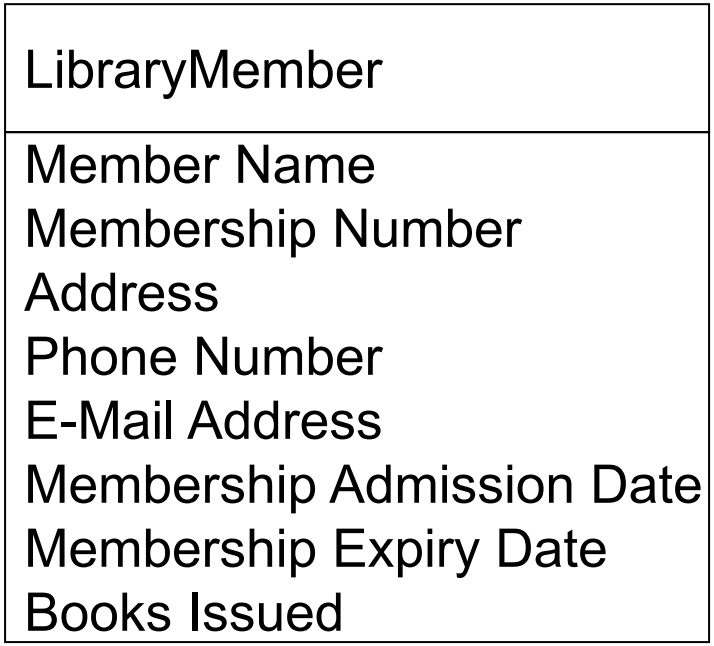



**Different representations of the LibraryMember class**

# Example of Class diagram





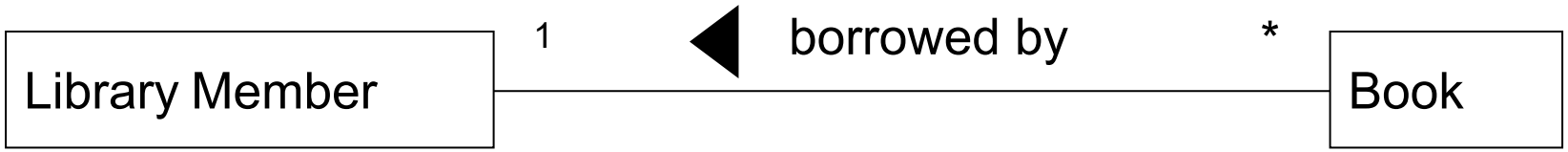



**Different representations of the LibraryMember class**

# Association Relationship

- Enable objects to communicate with each other
- Usually binary but more classes can be involved
- Class can have relationship with itself (recursive association)
- Arrowhead used along with name, indicates direction of association
- Multiplicity indicates # of instances

# Association Relationship

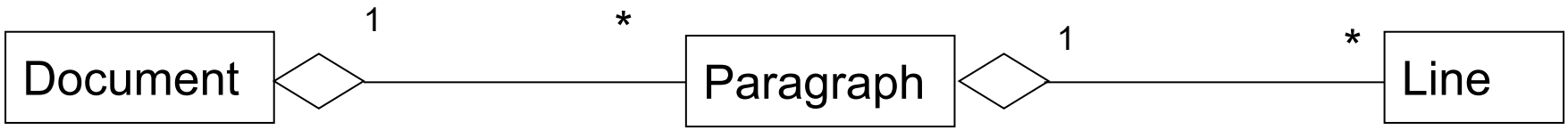



**Association between two classes**

# Aggregation Relationship

- Represent a whole-part relationship
- Represented by diamond symbol at the composite end
- Cannot be reflexive(i.e. recursive)
- Not symmetric
- It can be transitive

# Aggregation Relationship




**Representation of aggregation**

# Composition Relationship

- Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole.
- Life of a parts cannot exist outside the whole.
- Lifeline of both are identical.
- When the whole is created, the parts are created and when whole is destroyed, the parts are destroyed.

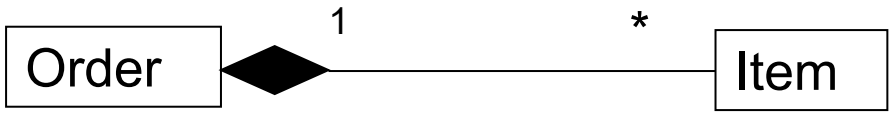
- Life of **item** is same as the **order**


**Representation of composition**

# Aggregation versus Composition

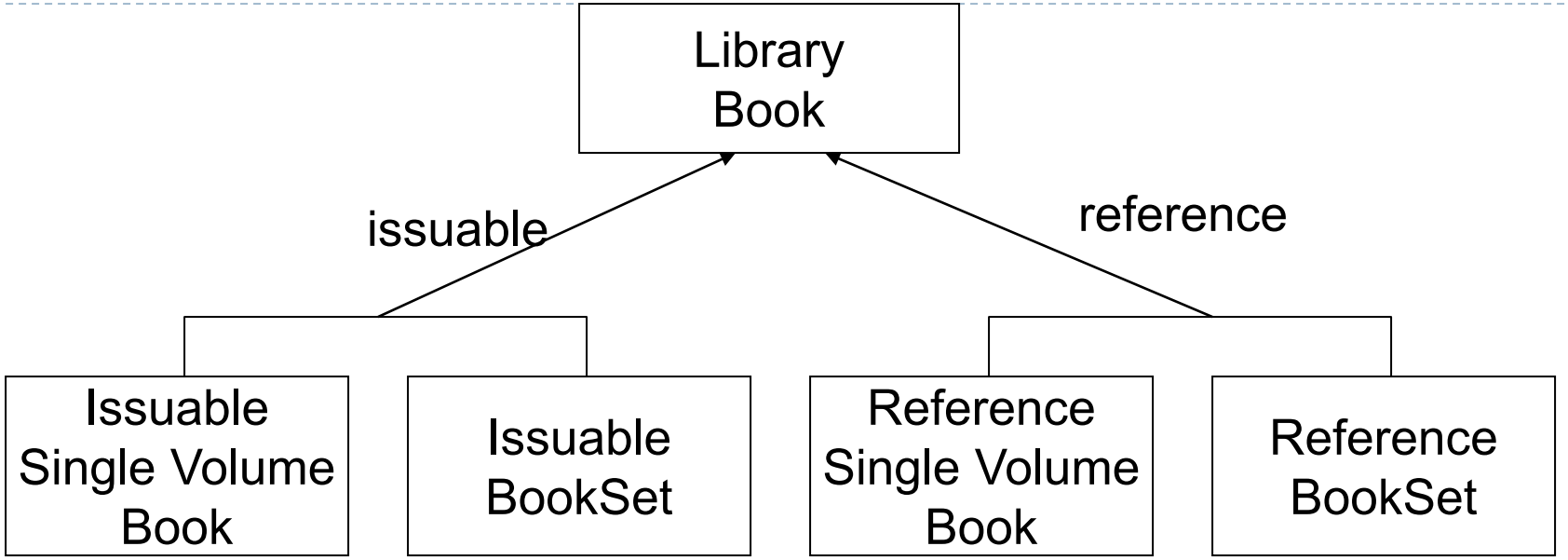
- Both represent part/whole relationships.
- When the components can dynamically be added and removed from the aggregate, then the relationship is aggregation.
- If the components can not be dynamically added or removed, then the relationship is composition.

- Life of **item** is same as the **order**




**Representation of composition**

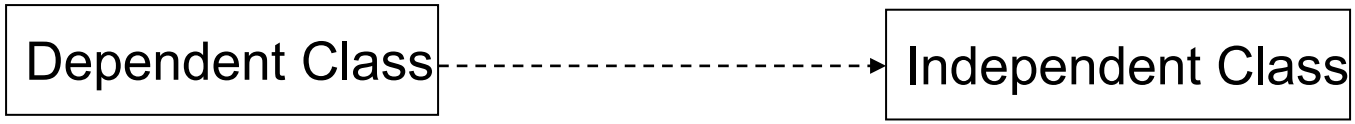
# Inheritance Relationship




**Representation of the inheritance relationship**

- *Issuable* and *reference* here are discriminators.
- The set of subclasses of a class having the same discriminator is called a partition.

# Class Dependency

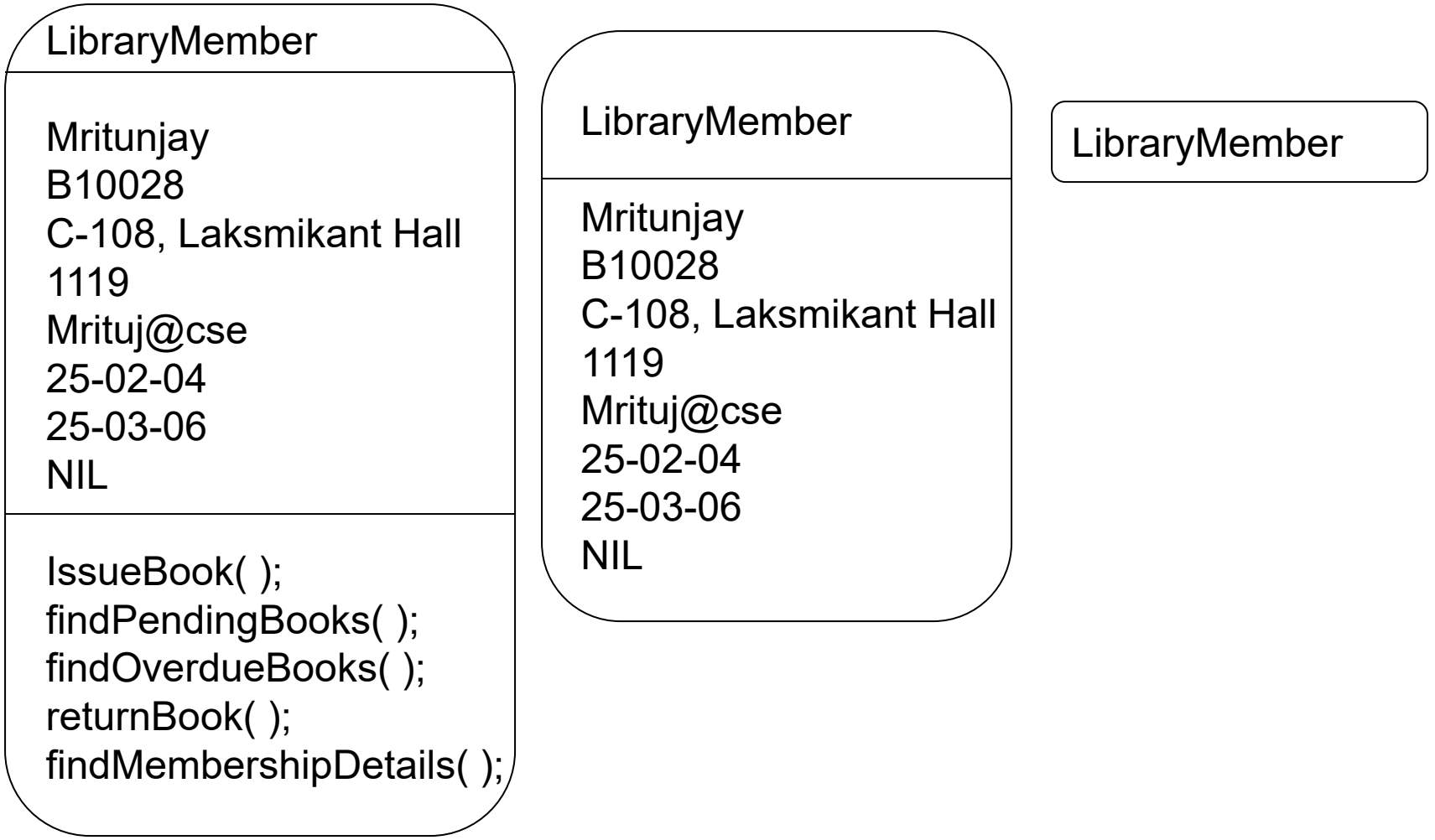



**Representation of dependence between class**

# Object diagram

- Object diagrams shows the snapshot of the objects in a system at a point in time
- It shows instances of classes, rather than the classes themselves, it is often called as an instance diagram.
- Objects are drawn using rounded rectangles.
- An object diagram may undergo continuous change as execution proceeds.
- Links may get formed between objects and get broken.

# Object diagram




**Different representations of the LibraryMember object**

# Interaction diagram

- Models show that how groups of objects collaborate to realize some behaviour
- Typically each interaction diagram realizes behaviour of a single use case
- For complex use cases, some times more than one interaction diagrams may be necessary to capture the behaviour.

# Interaction diagram

- Two kinds:
  - Sequence
  - Collaboration
- Two diagrams are equivalent but portrays different perspective
- These diagram play a very important role in the design process

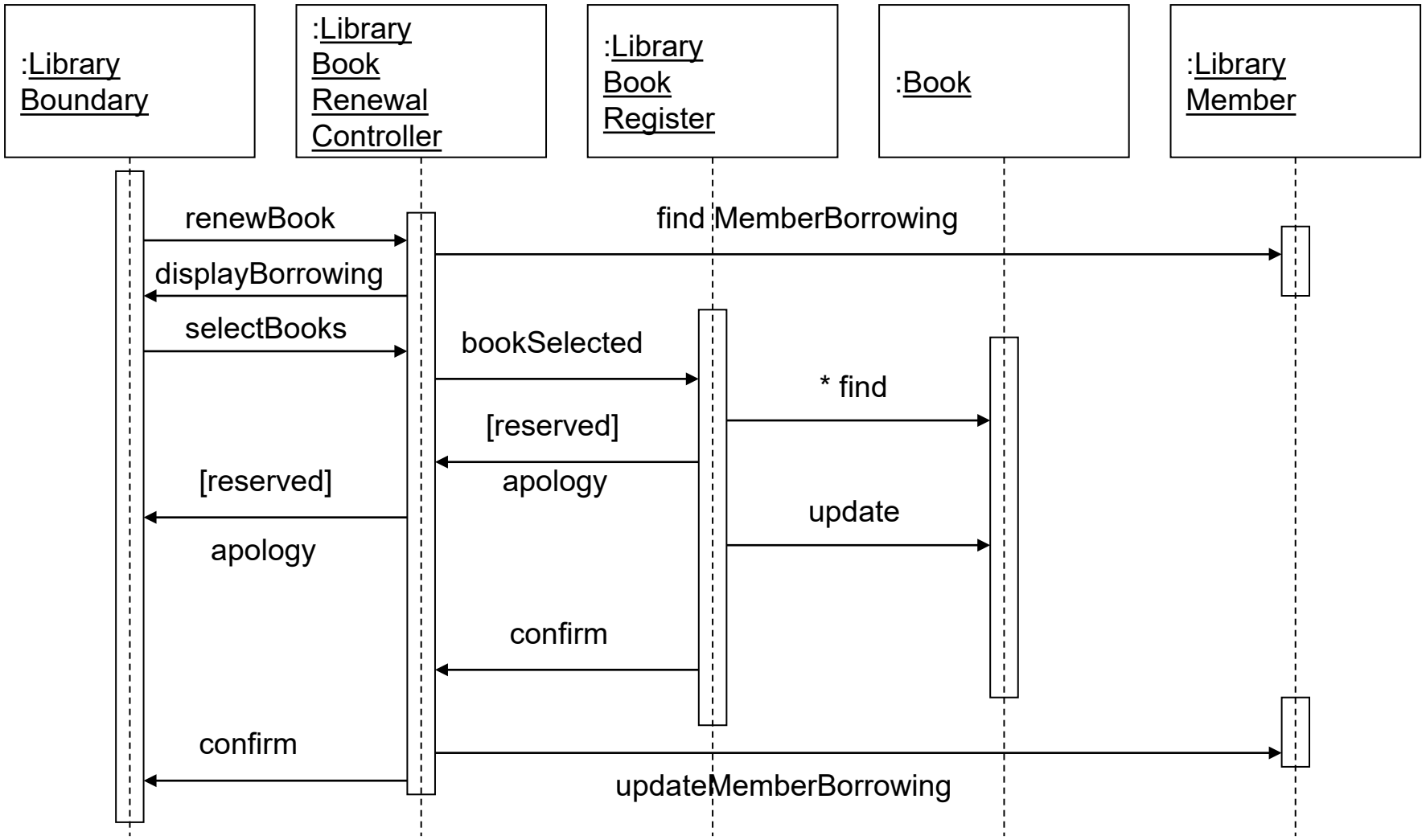
# Sequence diagram

- Shows interaction among objects as two-dimensional chart
- Objects are shown as boxes at top
- If object created during execution then shown at appropriate place
- Objects existence are shown as dashed lines (lifeline)
- Objects activeness, shown as rectangle on lifeline

# Sequence diagram

- Messages are shown as arrows
- Message labelled with message name
- Message can be labelled with control information
- Two types of control information: condition ([]) & an iteration (*)

# Example of Sequence diagram

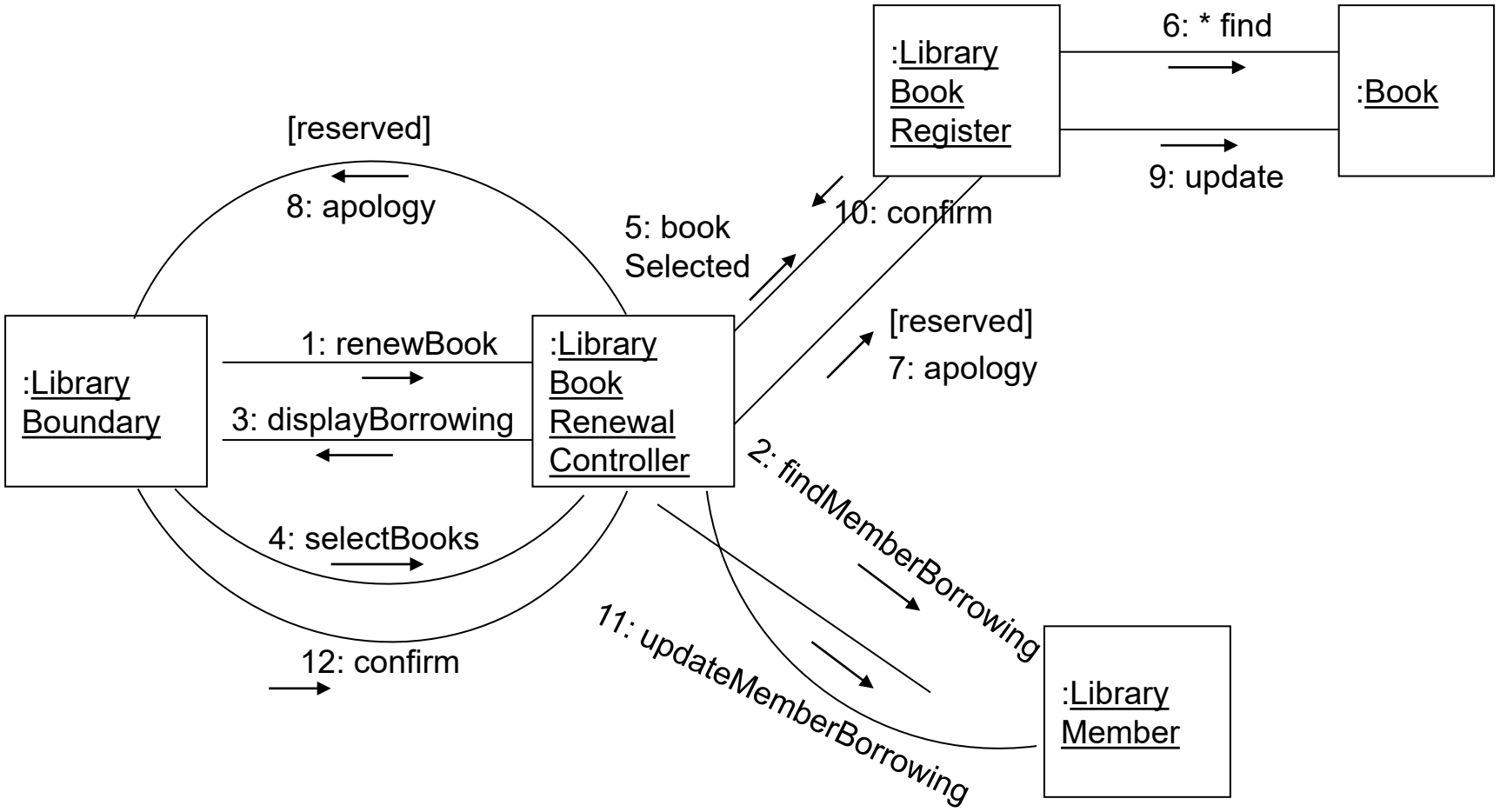



**Sequence Diagram for the renew book use case**

# Collaboration diagram

- Shows both structural and behavioural aspects
- Objects are collaborator, shown as boxes
- Messages between objects shown as a solid line
- Message is shown as a labelled arrow placed near the link
- Messages are prefixed with sequence numbers to show relative sequencing

# Example of Collaboration diagram




**Collaboration Diagram for the renew book use case**

# Activity diagram

- No such diagrams were present in Booch, Jacobson, or Rumbaugh.
- New concept, possibly based on event diagram of Odell [1992]
- Represent processing activities and their sequence of activation, may not correspond to methods

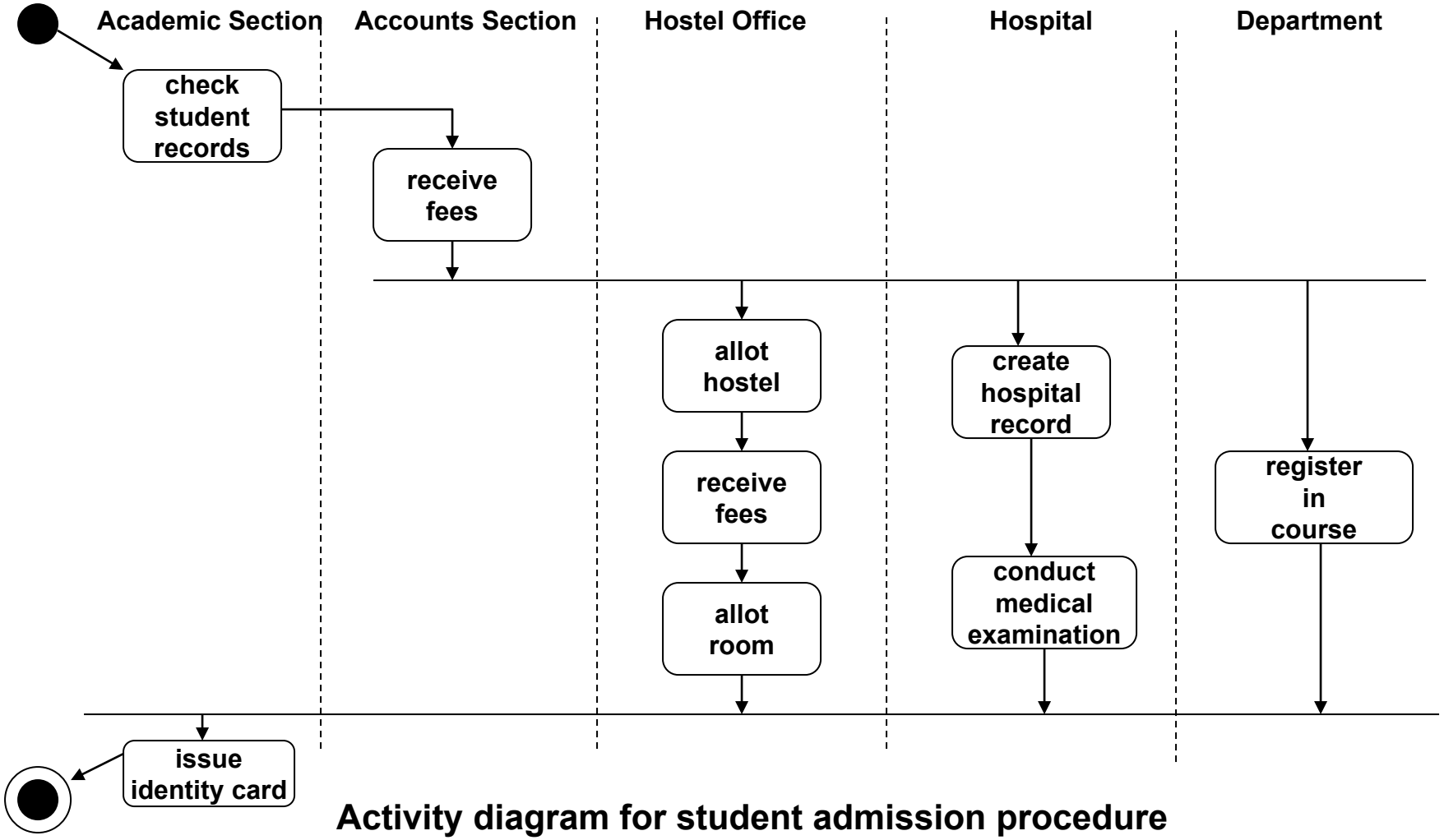
# Activity diagram

- Activity is a state with an internal action and one/many outgoing transition which automatically follow the termination of the internal activity.
- Can represent parallel activity and synchronization aspects involved in different activities unlike procedural flow chart.
- Parallel activities are represented by Swim lanes enable to group activities based on who is performing them
- Example: academic department vs. hostel

# Activity diagram

- The activities in a swim lanes can be assigned to some model elements, e.g. classes or some component.
- Normally employed in business process modelling
- Carried out during initial stage of requirement analysis and specification
- Understand complex processing activities involving the roles played by many components.
- Can be used to develop interaction diagrams

# Example of Activity diagram




**Activity diagram for student admission procedure**

# State Chart diagram

- Based on the work of David Harel [1990]
- Model how the state of an object changes in its lifetime
- Based on finite state machine (FSM) formalism

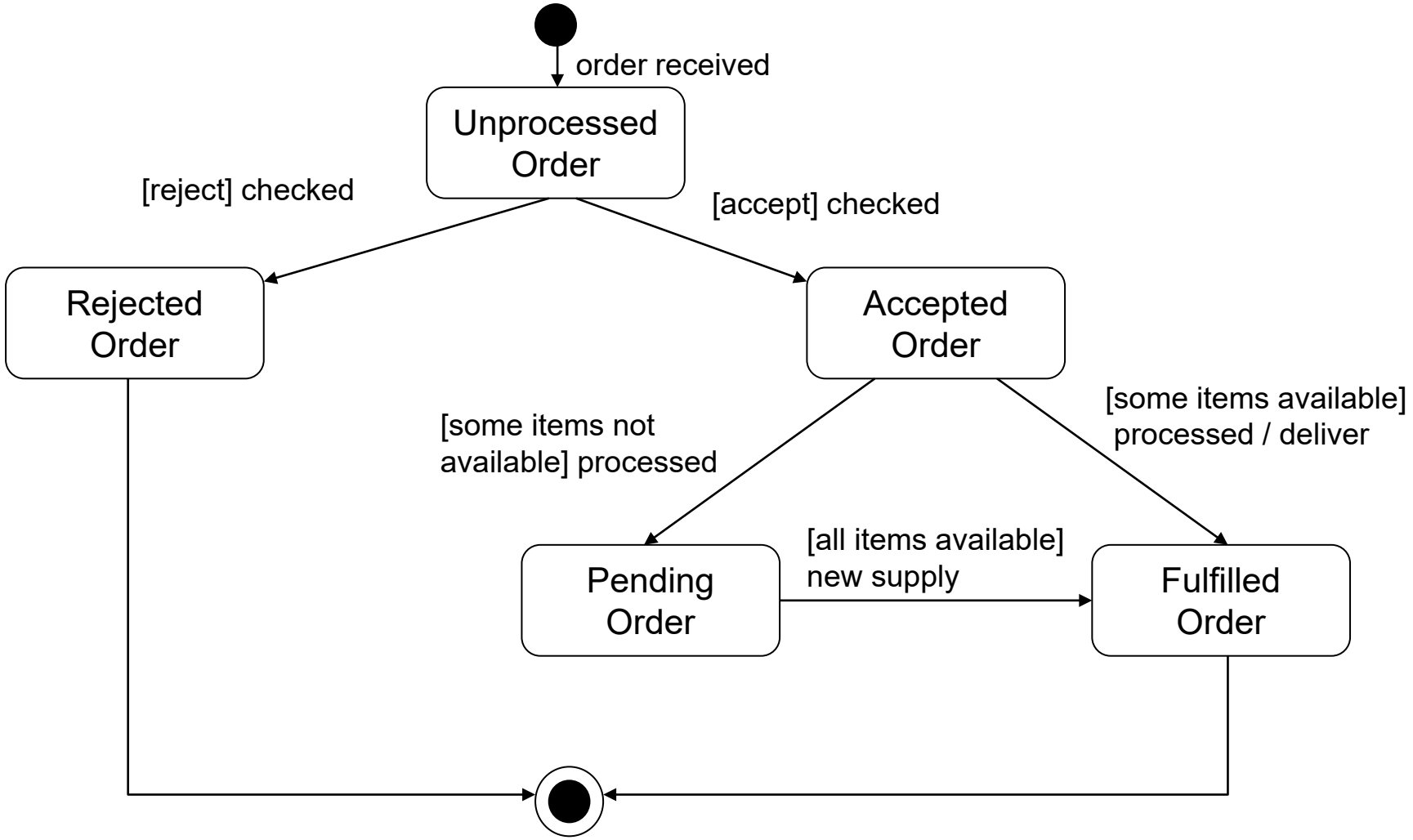
# State Chart diagram

- State chart avoids problem of state explosion as in FSM
- Hierarchical model of a system, represents composite state (nested)

# State Chart diagram

- Elements of state chart diagram
- Initial State: Filled circle
- Final State: Filled circle inside larger circle
- State: Rectangle with rounded corners
- Transitions: Arrow between states, also boolean logic condition (guard)

# Example of State Chart diagram




**Example: State chart diagram for an order object**

# Object-oriented Software Design & Patterns

- Objects are identified by examining nouns in problem description
- Many OOAD techniques are proposed, e.g. by Grady Booch [1991] etc.
- OOA refers to developing an initial model of a S/W product from an analysis of its requirements specification.
- From requirements specification, initial model is developed (OOA)
- Analysis model is refined into a design model
- Design model is implemented using OO concepts

# The Unified Process

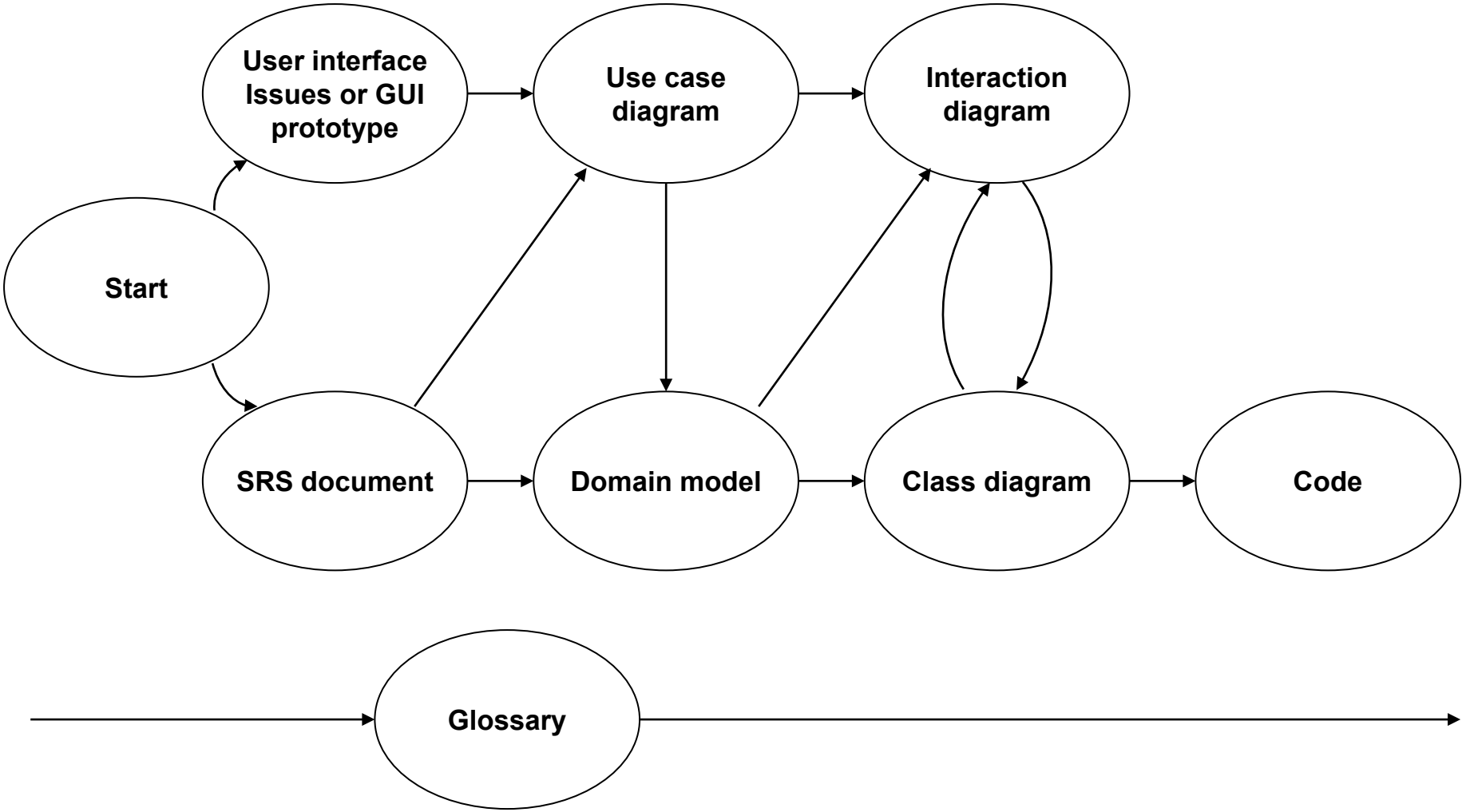
- The Unified Process is an extensible framework which needs to be customized for specific type of projects.
- There are characteristics of Unified Process
  - Use Case Driven
  - Architecture-centric
    - Since no single model is sufficient to cover all aspects of a system, the Unified Process supports multiple architectural models and views.
  - Iterative and Incremental
    - Each iteration results in an increment
    - The Inception phase may also be divided into iterations for a large project.

# Four Phases of Unified Process

- **Inception**
  - In this phase, Scope of project is defined & prototypes may be developed about the project.
    - Develop an approximate vision of the system,
    - make the business case,
    - Produce rough estimate for cost and schedule.
- **Elaboration**
  - In this phase, Functional & Non-Functional requirements are captured.

- **Construction**
  - During this phase, Analysis, Design & Implementation activities are carried out.
  - System features are implemented in iterations.
  - Each iteration results in executable release of the software.
- **Transition**
  - In this phase, product is installed in the user's environment and maintained.

# Object-oriented Software Design Process



# Use Case Model Development

- An overriding principle while identifying and packaging use cases
  - There should be a strong correlation between the GUI prototype, the contents of the users' manual and the use case model of the system.
- Each of the menu options in the top-level menu of the GUI would usually correspond to a package in the use case diagram.
- Use cases should not be too tightly tied to the GUI.

  Example: GUI element such as pushButton and radioButton may change frequently during software development

# Common mistakes committed in use case model development

- **Clutter:** large number of use cases are present in the top-level of the use case diagram
- **Too detailed:** confuse sub-steps of use cases with separate use cases.

  example: print receipt be a sub-step of the withdraw cash use case.
- **Omitting text description:** difficult to gain full understanding of the use case
- **Overlooking some alternate scenarios:** capture all alternate scenarios of each use case.

# Domain Modelling

- It is also known as Conceptual Modelling.
- Representation of concepts or objects appearing in the problem domain
- Also captures relationships among objects
- Only classes but no methods and data are represented.
- Three types of objects are identified
- Boundary objects: handle user interface
- Entity objects: corresponds to physical entities.
- Controller objects: objects that are entirely conceptual

# Boundary Objects

- Boundary objects with which the actors interact.
  - Includes screens, menus, forms, dialogs etc.
  - Do not perform processing but validates, formats etc.
  - They were earlier being called as the interface objects.
- Interface class term used for these in Java, COM/DCOM & UML
- The initial identification of the boundary classes can be made by defining one boundary class per actor/use case pair.
- Different categories of user have different privileges and different levels familiarity.

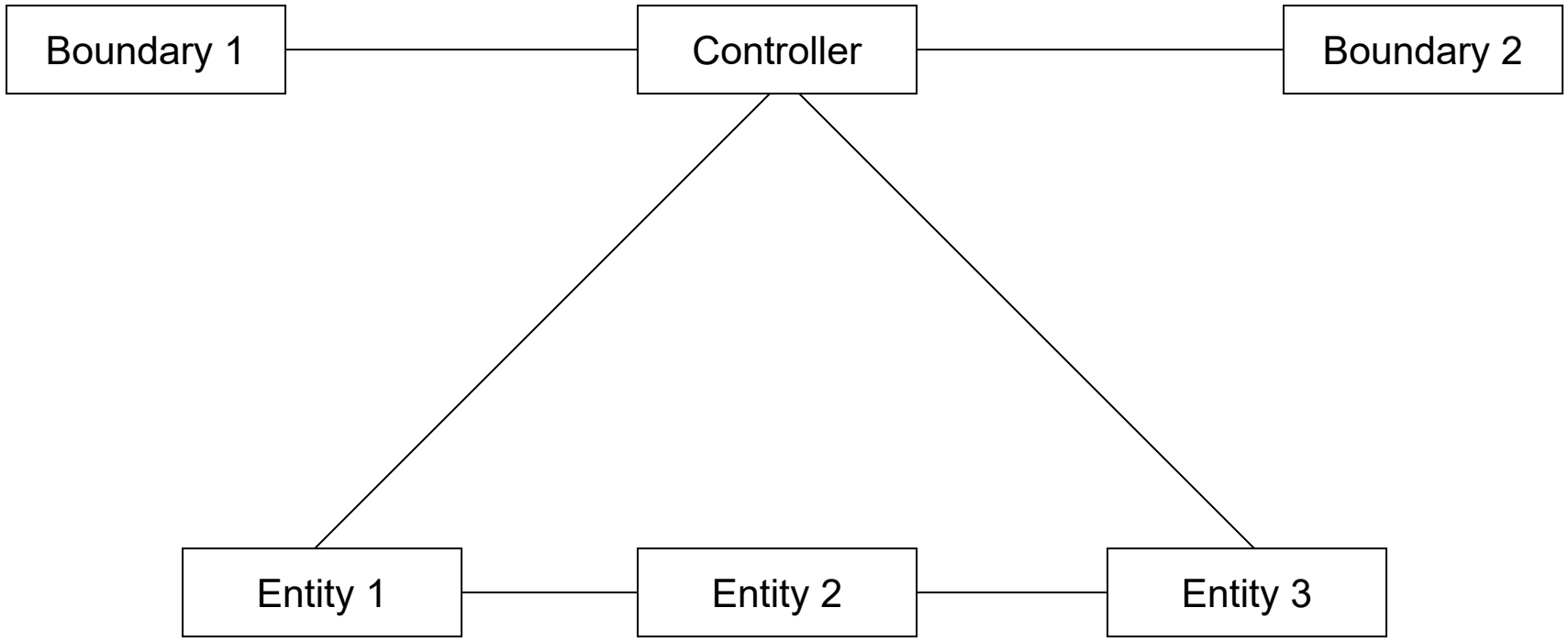
# Entity Objects

- Hold information such as data tables & files, e.g. Book, BookRegister
- Many of these are dumb servers
- Responsible for storing data, fetching data etc.

# Controller Objects

- Coordinate the activities of a set of entity objects
- Interface with the boundary objects
- Realizes use case
- Embody most of the logic involved with the use case realization
- There can be more than one controller

# Use Case Realization




**Realization of use case through the collaboration of Boundary, controller and entity objects**

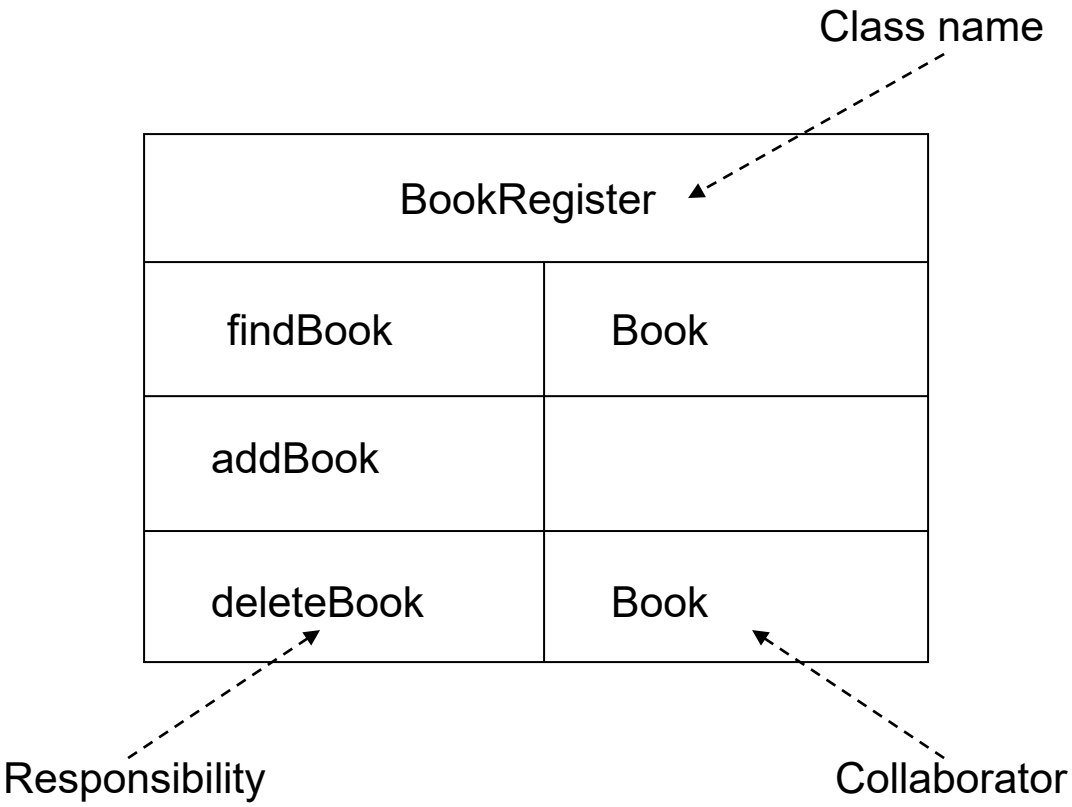
# Class-Responsibility-Collaborator(CRC) Cards

- Pioneered by Ward Cunningham and Kent Beck
- Developing the interaction diagram for use cases (that are more complex in nature) may require participation of a team of developers through the use of CRC cards.
- Index cards prepared one each per class
- Responsibility is written on these cards
- Responsibility of collaborating object is also written

# Class-Responsibility-Collaborator(CRC) Cards

- Interaction diagrams are developed by flipping through CRC cards
- Required for complex use cases
- Team members participate to determine responsibility of classes involved in the use case realization

# Example: CRC Cards




**CRC card for the BookRegister class**

# Design Patterns

- Originated in the field of architecture, where large buildings are designed in specific pattern solution.
- Working out the design solution, experienced designers consciously or unconsciously reuse solutions that they might have worked out in the past.
- Such reuse of the design solutions is systematized by the concept of patterns.
- Commonly accepted solutions to be reused by everybody familiar with the pattern.

# Design Patterns

- Design patterns are commonly accepted solutions to some problems that recur during designing different applications.
- They have been found to make design process efficient.
- Use of it reduces the number of design iterations, and at the same time improves the quality of the final design solution.

# Design Patterns

- Every non-trivial problem consisting of a large number of sub-problems.
- Solving any two problems, several sub-problems that are common between the two can be identified.
- Mastering the commonly accepted solutions to few important sub-problems that repeat across different problems.
- This captures central idea behind patterns.

# Design Patterns

- Patterns are well-documented building blocks for software design.
- Standard solutions to commonly recurring problems
- The basic idea behind patterns is that if you can master a few important patterns, you can easily spot them in application development problems and effortlessly use the pattern solutions.

# Design Patterns

- If Patterns are really based on common sense, then what is the point learning patterns?

- It has been observed that while grappling with the nitty gritty of complex design problems,
  - designer often forget common sense.

# Design Patterns

- Pattern has four important parts
  - The problem
  - The context (problem occur)
  - The solution
  - The context within the solution work and would not work

# Types of Patterns

- Very high-level designs termed as Architectural designs, pattern solutions have been defined for use in concrete designs.
- There are 3 types of patterns:
  - Architectural patterns
  - Design patterns
  - Idioms

# Architectural Patterns

- Identify & provide solutions to problems that are identifiable while carrying out architectural designs.
  - Architectural designs cannot be directly translated to code.
- Pattern suggest a set or predefined subsystems, specify responsibilities and includes rules and guidelines for organizing the relationship among them.
  - Are generally constructed for very large problems.

# Design Patterns

- Suggest a scheme for structuring the classes in a design solution and defines the interactions among those classes.
- Describes some commonly recurring structure of communicating classes that can be used to solve some general design problems
- Are typically described in terms of classes, their instances, their roles and collaborations.

# Example: Factory Method

```
// A design without factory pattern
#include <iostream>
using namespace std;

// Library classes
class Vehicle {
public:
        virtual void printVehicle() = 0;
};
```

# Example: Factory Method

```
class TwoWheeler : public Vehicle {
public:
        void printVehicle() {
                cout << "I am two wheeler" << endl;      }
};
class FourWheeler : public Vehicle {
        public:
        void printVehicle() {
                cout << "I am four wheeler" << endl;      }
};
```

# Example: Factory Method

```
// Client (or user) class
class Client {
  public:
        Client(int type) {
        // Client explicitly creates classes according to type
                if (type == 1)
                        pVehicle = new TwoWheeler();
                else if (type == 2)
                        pVehicle = new FourWheeler();
                else                    pVehicle = NULL;
        }
```

# Example: Factory Method

```
~Client() {
                if (pVehicle)
                {       delete[] pVehicle;
                        pVehicle = NULL;
                }
        }
Vehicle* getVehicle() { return pVehicle;    }
private:
        Vehicle *pVehicle;
};
```

# Example: Factory Method

```
// Driver program
int main() {
        Client *pClient = new Client(1);
        Vehicle * pVehicle = pClient->getVehicle();
        pVehicle->printVehicle();
        return 0;
}
```

# Solution to design problem

```
// C++ program to demonstrate factory method design pattern
#include <iostream>
using namespace std;
enum VehicleType {
    VT_TwoWheeler, VT_ThreeWheeler, VT_FourWheeler
};
```

# Solution to design problem

```
// Library classes
class Vehicle {
public:
        virtual void printVehicle() = 0;
        static Vehicle* Create(VehicleType type);
};
class TwoWheeler : public Vehicle {
public:
        void printVehicle() {
                cout << "I am two wheeler" << endl;
        } };
```

# Solution to design problem

```
class ThreeWheeler : public Vehicle {
public:
        void printVehicle() {
                cout << "I am three wheeler" << endl;
        }
};
class FourWheeler : public Vehicle {
        public:
        void printVehicle() {
                cout << "I am four wheeler" << endl;
        } };
```

# Solution to design problem

```
// Factory method to create objects of different types.
Vehicle* Vehicle::Create(VehicleType type) {
        if (type == VT_TwoWheeler)
                return new TwoWheeler();
        else if (type == VT_ThreeWheeler)
                return new ThreeWheeler();
        else if (type == VT_FourWheeler)
                return new FourWheeler();
        else return NULL;
}
```

# Solution to design problem

```
// Client class
class Client {
public:
        // Client doesn't explicitly create objects
        Client()
        {
                VehicleType type = VT_ThreeWheeler;
                pVehicle = Vehicle::Create(type);
        }
```

# Solution to design problem

```
        ~Client() {
                if (pVehicle) {  delete[] pVehicle;
                                  pVehicle = NULL;
                               }
        }
        Vehicle* getVehicle() { return pVehicle; }
private:
        Vehicle *pVehicle;
    };
```

# Solution to design problem

```
// Driver program
int main() {
        Client *pClient = new Client();
        Vehicle * pVehicle = pClient->getVehicle();
        pVehicle->printVehicle();
        return 0;
}
```

# Idioms

- Are low level patterns that are programming language specific.
- Describe how to implement a solution to a particular problem using the features of a given programming language.
- Helps in improving the quality of code.

# Idioms…

- Reduces the bug detection and corrections iterations.
- In contrast to algorithms, patterns are more concerned with aspects such as Maintainability and ease of development rather than space and time efficiency.

# Patterns Vs Algorithms

- Patterns & Algorithms are in some respect similar since both attempt to provide reusable solutions.

- Algorithms primarily focus on solving problems with reduced space and/or time requirements, whereas patterns focus on understandability and maintainability of design and easier development.

# Pros and Cons of Design Patterns

- Strength
  - Provide a common vocabulary that helps to improve communication among the developers.
  - Help to capture and disseminate expert knowledge
  - Help designers to produce designs that are flexible, efficient and easily maintainable.
  - Guide developers to arrive at correct design decisions and help to improve the quality of design

# pros

- Reduce the number of design iterations and help improve the designer productivity.

# Con's of Design Patterns

- Design patterns do not directly lead to code reuse.
- No methodology is available that can be used to select the right design patterns at the right point during a design exercise.

# Antipattern

- Antipattern represent lessons learned from a bad design.
- There are two popular antipatterns
  - Those that describe bad solutions to problems which leads to bad situations.
  - Those that describe how to avoid bad solutions to problems.

# Interesting Anti patterns

- **Input Kludge**
  - This concerns failing to specify and implement a mechanism for handling invalid inputs.
- **Magic pushbutton**
  - Concerned with coding implementation logic directly within the code of user interface, rather than performing them in separate class
- **Race hazard**
  - Concerned failing to see the consequences of all the different orders in which events take place in practice

# Example Pattern

- Expert
  - Problem: Which class should be responsible for doing certain things
  - Solution: Assign responsibility to the class that has the information necessary to fulfil the required responsibility

# Expert Pattern

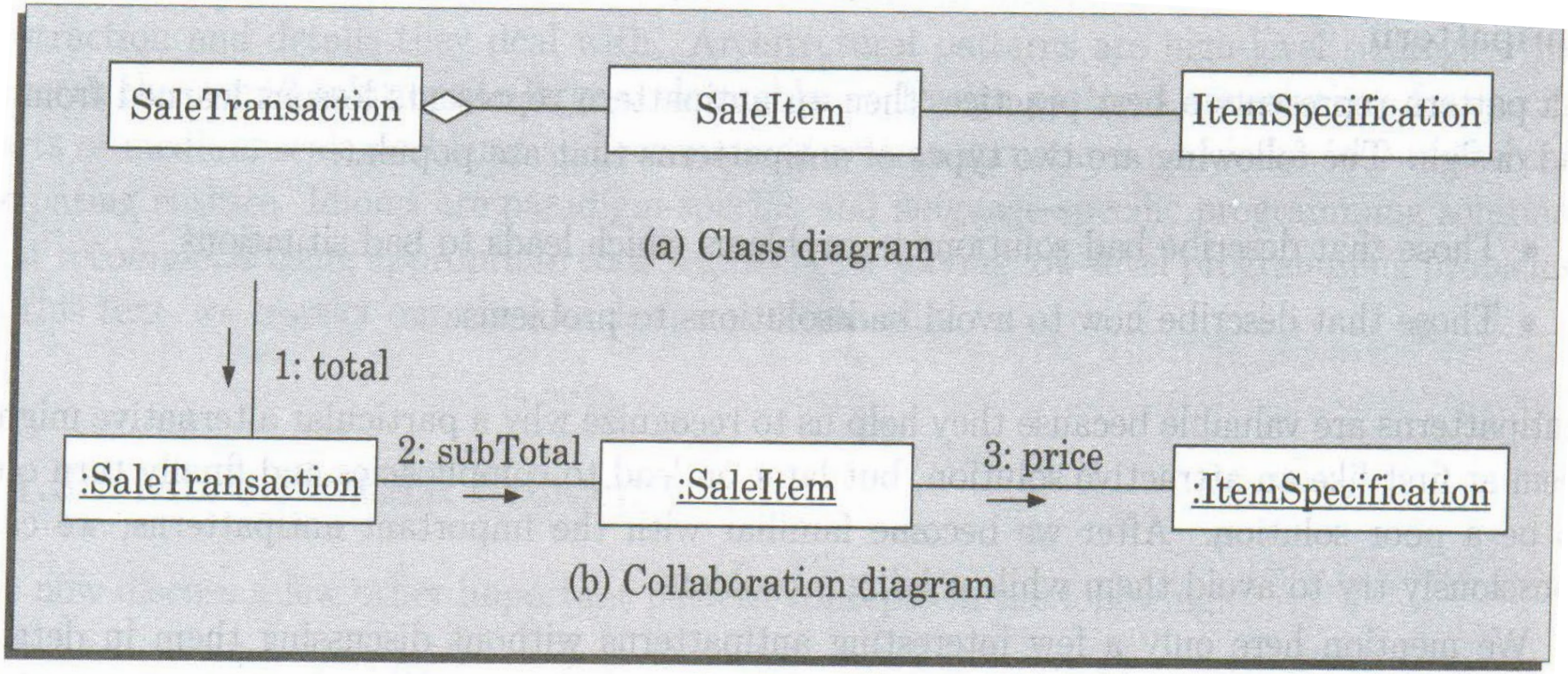



**Figure 8.1:** Expert pattern.

# Example Pattern

- Creator
  - Problem: Which class should be responsible for creating a new instance of some class?
  - Solution: Assign a class C1 the responsibility to create class C2 if
    - C1 is an aggregation of objects of type C2
    - C1 contains object of type C2

# Example Pattern

- Controller
  - Problem: Who should be responsible for handling the actor requests?
  - Solution: Separate controller object for each use case.

# Example Pattern

- Facade
  - Problem: How should the services be requested from a service package?
  - Context (problem): A package (cohesive set of classes), example: RDBMS interface package
  - Solution: A class (DBfacade) can be created which provides a common interface to the services of the package

# Facade pattern

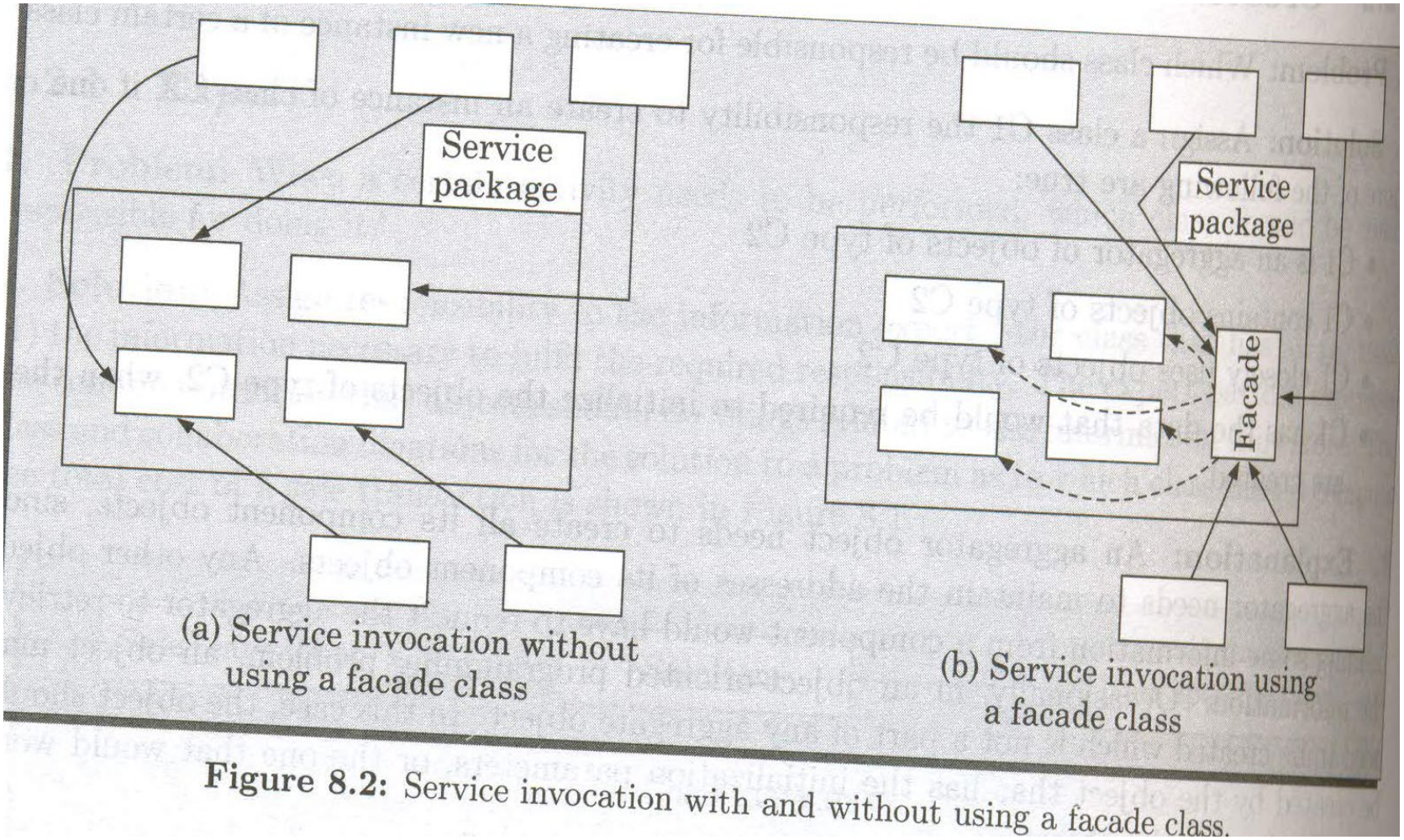



**Figure 8.2:** Service invocation with and without using a facade class.

# Model View Separation Patterns

- **Problem:** Should Non-GUI Classes communicate with the GUI Classes & Vice versa?
- **Context in which problem occurs**

  -This is a commonly occurring problem

  - Here **View** is a synonym for GUI objects & **Model** for Domain Layer (Non-GUI Objects)

  - Domain layer objects are responsible for providing the required service whereas the presentation layer objects are responsible for handling only the interactions with the user.

## 3. Solution

Depending on the context, the solution provided by either of observer pattern, model-view controller(MVC) pattern, or the publish-subscribe pattern can be used.

## 4. Explanation

The main idea is to achieve loose coupling between the model and view objects

- The complexity of the design is reduced and re-use of the model objects is enhanced.

# Observer Pattern

- **Problem :** How should the interactions b/w the Model & View objects be structured, when a model object is accessed by several view objects.
- **Solution :**
  - Observers should register themselves with model object.
  - Model object maintains list of all registered observers, so that when a change occurs to the model object it would notify all the registered observers.

# Cont…

- Each observer can then query the model object to get any specific information about the changes that might require.
- This pattern therefore uses both the Push & Pull models.

- The interaction diagram is shown in the next slide…

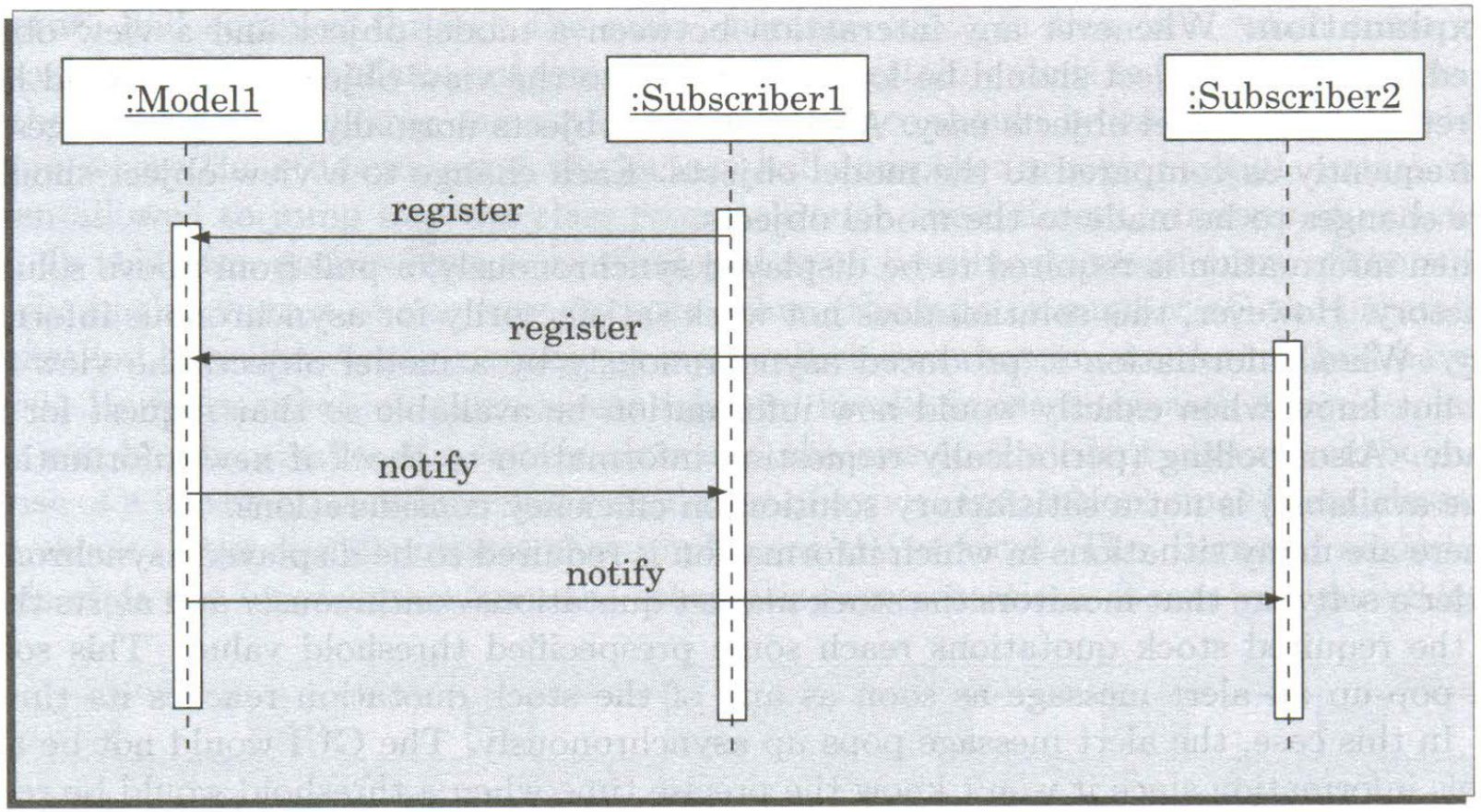



**Figure 8.3:** Interaction diagram for the observer pattern.

# Cont..

Explanation:

- The observer pattern has the following limitations
  - The model objects incur a substantial overhead to support registration of the observers and also to respond to the queries from the observers.
  - The two stage process: notification and query reduces coupling between the model and observer objects.

# Model View Controller (MVC) Pattern

- **Problem:** How should the GUI Objects interact with model objects?

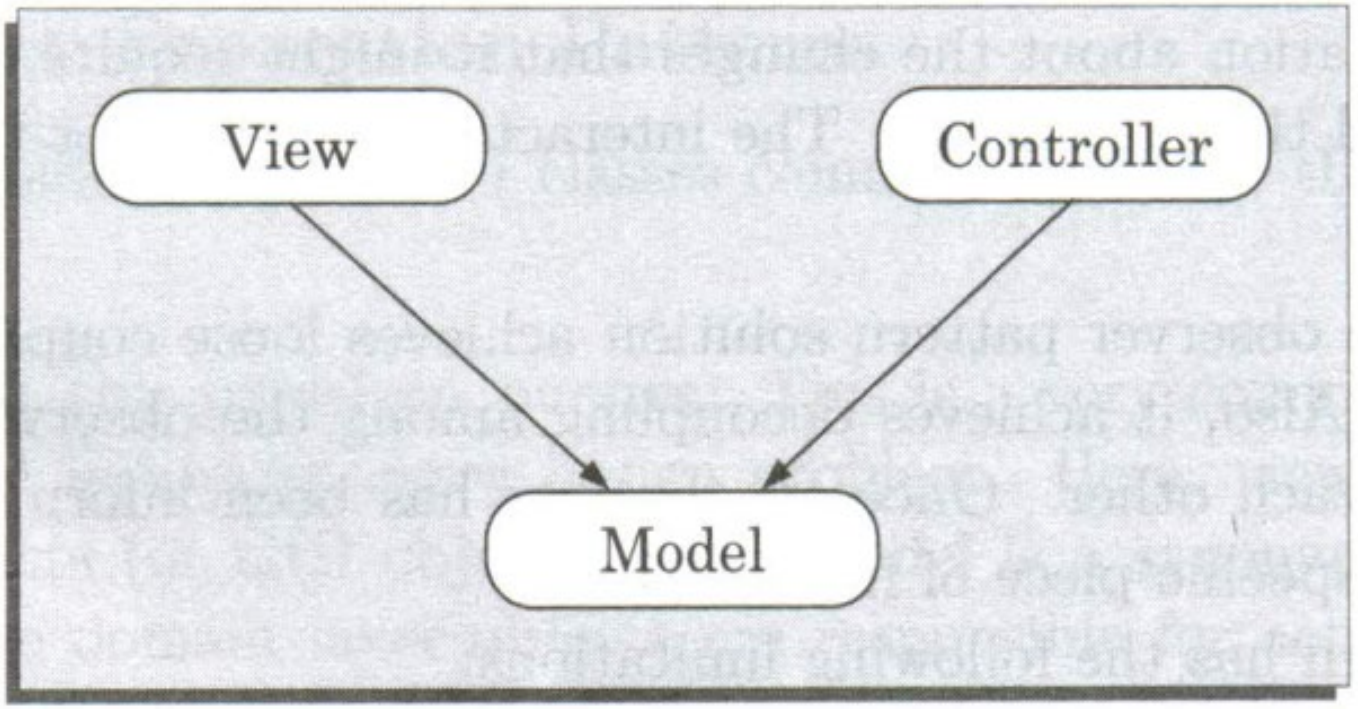



**Figure 8.4:** Class structure for the MVC pattern.

# Model View Controller (MVC) Pattern

**2. Solution:** The GUI objects need to be separated from view and controller types.

**3. Explanation:** This pattern is useful in applications in which the model object can change its state asynchronously and

- Multiple consistent view of the model object are to be handled effectively.

# Model View Controller (MVC) Pattern

- As a simple example, the MVC Pattern can be used when for the same input data separate line plot, bar chart and pie chart representations need to be shown.

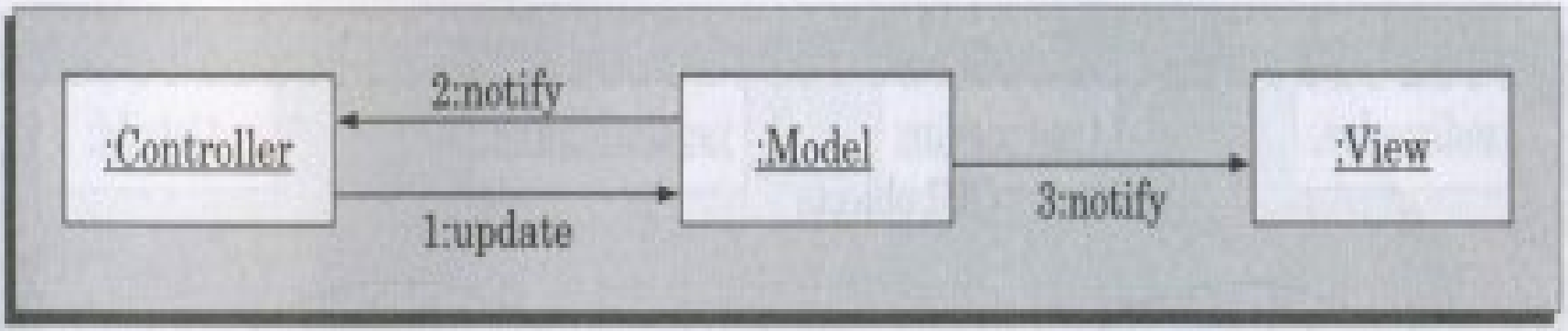



Figure 8.5: Interaction model for the MVC pattern.

# Publish-Subscribe Pattern

- This pattern is a more general form of the observer pattern and overcomes many of the shortcomings of the observer pattern.

- **Problem :** When a given model object is accessed by a large number of view objects & the model state changes asynchronously, how should the interactions be structured.

# Publish-Subscribe Pattern

- **Solution** : This pattern suggests than an event notification system should be implemented through which the publisher (model objects) can indirectly notify the subscribers as soon as the necessary information becomes available.
- The Publish –Subscribe pattern has been schematically shown in the next slide…

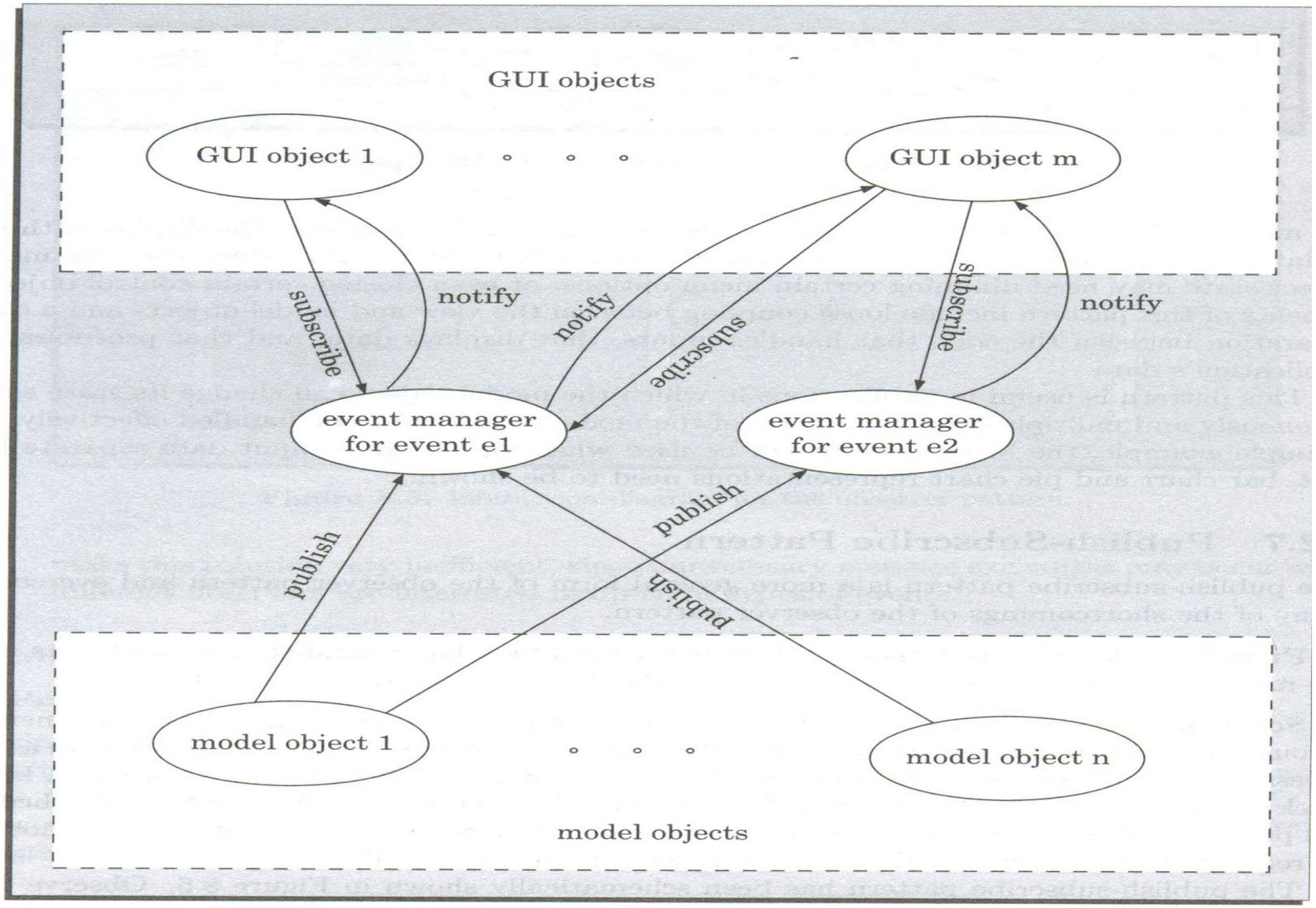



**Figure 8.6:** A schematic representation of the publish-subscribe pattern.

# Publish-Subscribe Pattern

- **Explanation :** Compared to the observer pattern, this pattern frees the model object from handling registration of observer objects and notification objects.
- Thus this pattern has an obvious advantage over the observer pattern, especially when the number of observers is large.

# Interaction Model for Publish- Subscribe Pattern

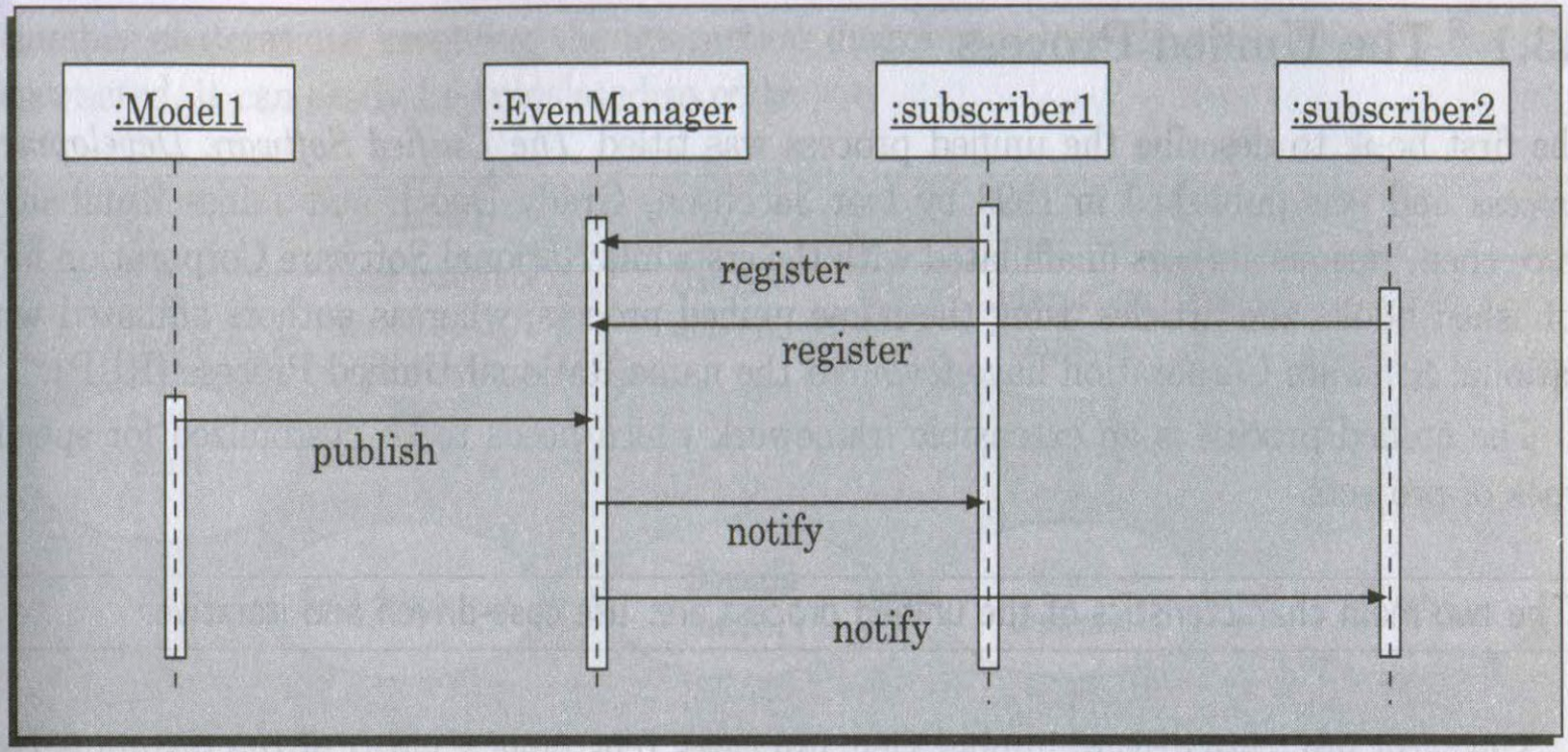



**Figure 8.7:** Interaction model of the publish-subscribe pattern.

# OOD Goodness Criteria

- Some of the accepted criteria for judging the goodness of a design are Coupling & Cohesion.
  - Coupling: is the degree to which each program module relies on each one of the other modules.
  - Cohesion: is a measure of how strongly-related the functionality expressed by the source code of a software module.

# 1. Coupling Guidelines

i. Number of messages between two objects or group of objects should be minimum.

ii. Increase in the number of message exchanges between two objects results in increased coupling.

iii. Excessive coupling should be reduced as it prevents reuse.

# 2. Cohesion Guidelines

Concerned with cohesion at Three levels.

i. Cohesiveness of the individual methods
ii. Cohesiveness of the data and methods within the class.
iii. Cohesiveness of an entire class hierarchy.

Note: A good software design should have LOW COUPLING & HIGH COHESION

# 3. Hierarchy & Factoring Guidelines

- A base class should not have too many subclasses. Approximately 7 (+2 or -2) derived classes from a base class at any level.

# 4. Keeping message protocols simple

- Complex message protocols are an indication of excessive coupling among objects.
- message requires more than 3 parameters then it is inferior design.

# 5. Number of methods

- If a class has more number of methods, then maintaining & debugging can be problematic.

# 6. Depth of the inheritance tree

- Deeper is a class in inheritance hierarchy, greater is the number of methods it is likely to inherit.
- Complexity increases when the level increases

# 7. Number of messages per use case

- A single use case should not result in excessive message generation & transmission in a system.

# 8. Response for a class

- It is defined as the maximum number of methods of other objects that an instance of this class invokes.
- A class which calls more than about seven different methods is susceptible to errors.

# Summary

- We discussed object-oriented concepts
  - Basic mechanisms: Such as objects, class, methods, inheritance etc.
  - Key concepts: Such as abstraction, encapsulation, polymorphism, composite objects etc.

# Summary

- We discussed an important OO language UML
  - Its origin, as a standard, as a model
  - Use case representation, its factorisation such as generalization, includes and extends
  - Different diagrams for UML representation
  - In class diagram we discussed some relationships association, aggregation, composition and inheritance

# Summary

- Some more diagrams such as interaction diagrams (sequence and collaboration), activity diagrams, state chart diagram
- We discussed OO software development process and patterns
  - In this we discussed some patterns example and domain modelling