



OOAD using UML



Design principles

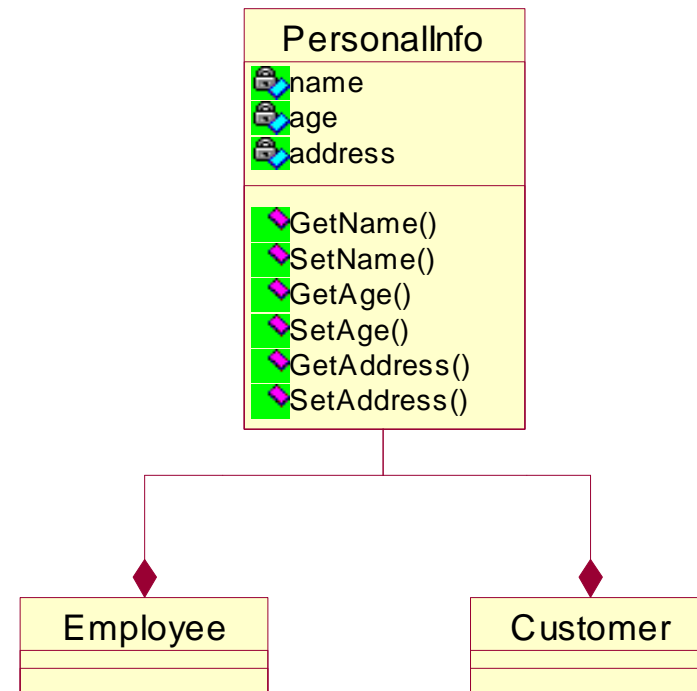
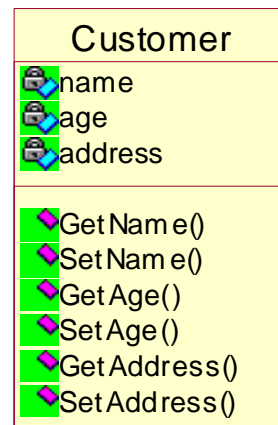
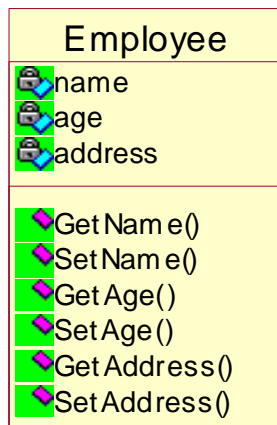
- Design principle is a basic technique that can be applied to a design to make it more maintainable, flexible and extensible.



Design principles

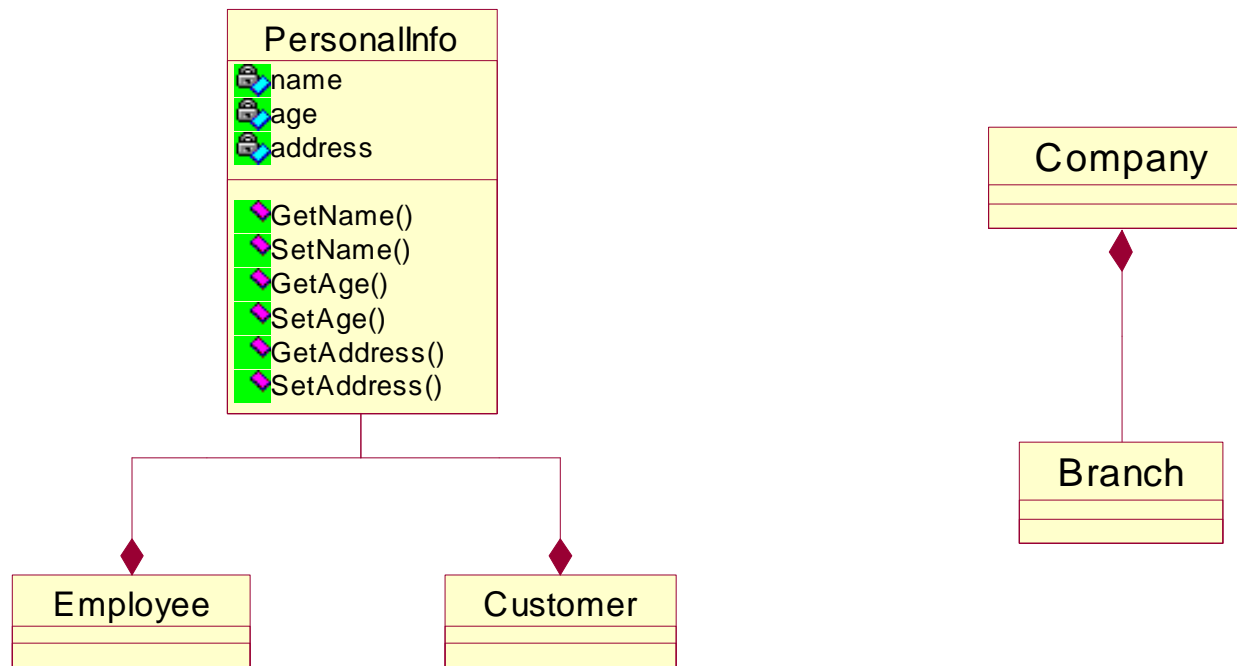
- Don't repeat yourself (DRY)
- Single responsibility principle (SRP)
- Liskov substitution principle (LSP)
- Encapsulate what varies
- Program to the interface not to an implementation

Don't repeat yourself (DRY)

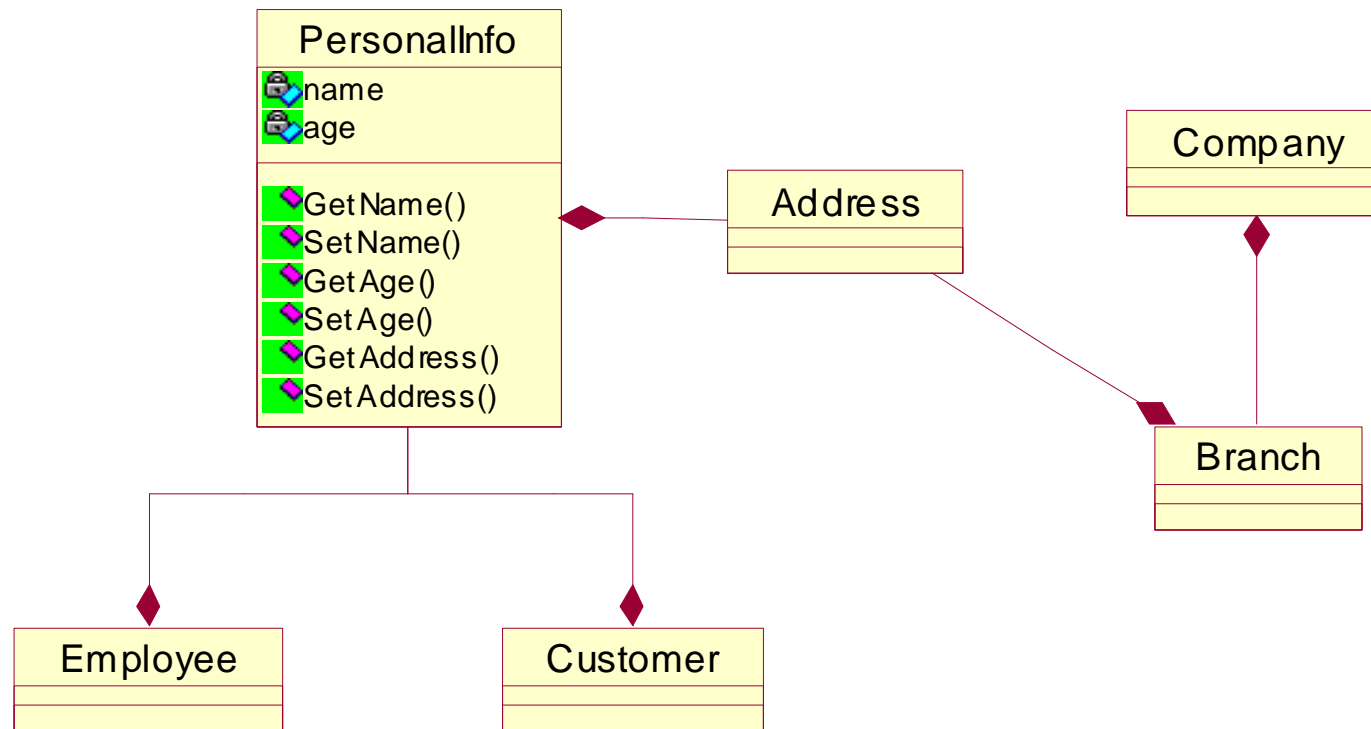


DRY

- Avoid duplicate code by abstracting things that are common and placing them in a single location



DRY





Single responsibility principle (SRP)

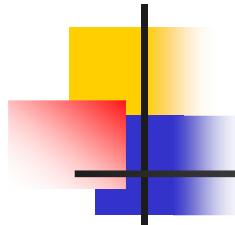
- A class should have only one reason to change
- Also known as high cohesion



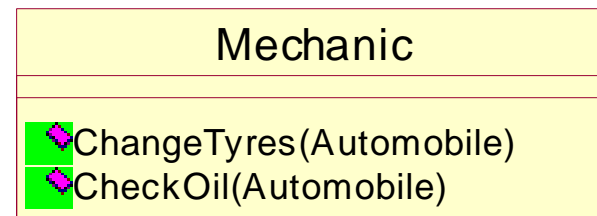
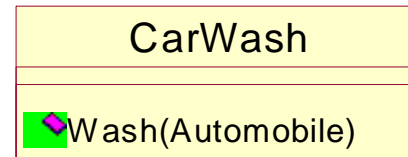
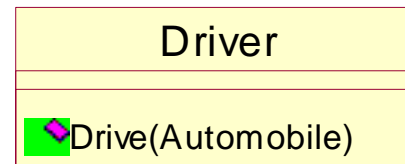
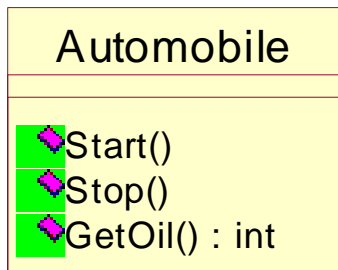
Spotting multiple responsibilities

Automobile

- Start()
- Stop()
- ChangeTyres()
- Drive()
- Wash()
- CheckOil()
- GetOil() : int



SRP

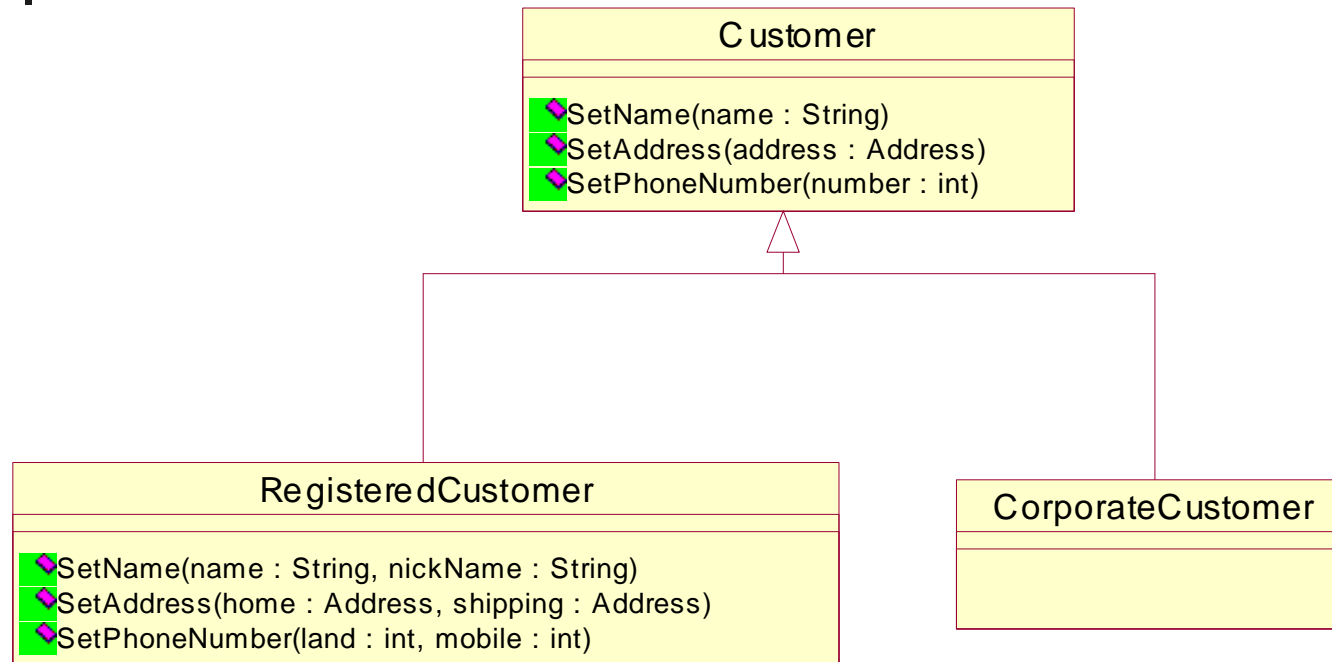




Liskov substitution principle (LSP)

- Sub types must be substitutable for their base types.
- Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing them.

Improper Inheritance

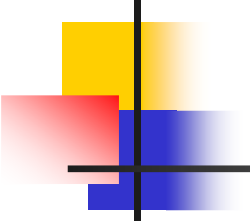


- Registered customer cannot be substituted to Customer. Method calls will result in calling base class implementation.

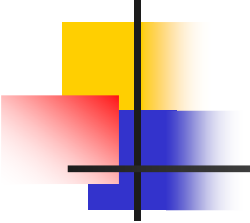


Encapsulate what varies

- What you hide you can change.
- Minimize the impact what varies -- what changes.
- By identifying what varies and hiding it (its implementation) behind an interface, then you can change its implementation without violating its *contract*

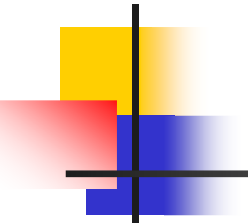


```
public class Class1 {  
    public int x;  
}  
  
public class Client {  
    private Class1 myClass1 = new Class1();  
  
    public int process(){  
        int intialValue = myClass1.x;  
        return intialValue * 4;  
    }  
}
```



```
public class Class1 {
    private int x;
    public int GetX() {
        return x;
    }
    public void SetX(int anInt) {
        x = anInt;
    }
}

public class Client {
    private Class1 myClass1 = new Class1();
    public int process(){
        int initialValue = myClass1.GetX();
        return initialValue * 4;
    }
}
```



```
public class Class1 {
    private String x = "0";    // x no longer an int
    public int GetX() {
        return Integer.parseInt(x);    // convert when needed
    }
    public void SetX(int anInt){
        x = new Integer(anInt).ToString();    // convert back
    }
}

public class Client {
    private Class1 myClass1 = new Class1();
    public int process() {
        int initialValue = myClass1.GetX();    // Client stays unchanged
        return initialValue * 4;
    }
}
```



Program to the interface

- Avoid referencing concrete classes, declare interfaces only
- Benefits of programming to an interface:
 - Clients remain unaware of the classes that implement the interface
 - Greatly reduces implementation dependencies
 - Don't declare variables to be instances of concrete classes if possible



Understanding Interface and Implementation

Relationships	Interface	Implementation
Generalization		
Realization		
Composition		

- Fill the table based on reuse Interface / implementation for each relationship



Understanding Interface and Implementation

- Generalization
 - Reuse of interface and implementation
 - Implementation(+ Interface) Inheritance
- Realization
 - Reuse of only interface
 - Interface Inheritance
- Composition
 - Reuse of only implementation
 - Implementation reuse and not the interface



Understanding Interface and Implementation

Relationships	Interface	Implementation
Generalization	Y	Y
Realization	Y	N
Composition	N	Y

- Therefore, it is possible to equate
Generalization = Realization + Composition



Capturing Requirements



The goals of the requirements workflow

- To establish and maintain agreement with the customers and other stakeholders on what the system should do—and why!
- To provide system developers with a better understanding of the system requirements
- To provide a basis for planning the technical contents of iterations
- To provide a basis for estimating cost and time to develop the system
- To define a user interface for the system, focusing on the needs and goals of the users



Requirements

- What is a requirement?
 - A condition or capability to which a system must conform
- Functional Requirements
 - What the system should do on behalf of the user
- Nonfunctional Requirements
 - The system should exhibit wide variety of attributes to deliver the desired quality to the end user



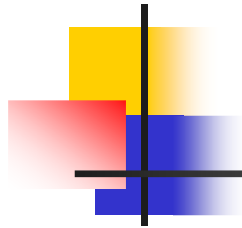
Requirements Workflow

- Analyze the Problem.
- Understand Stakeholder Needs.
- Define the System.
- Manage the Scope of the System.
- Refine the System Definition.
- Manage Changing Requirements.



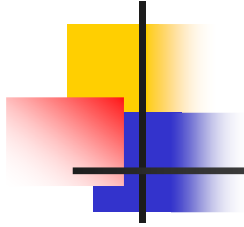
Workers in requirements

- System Analyst
 - Leads and coordinates requirements elicitation and use-case modeling by outlining the system's functionality
- Use-Case Specifier
 - Details all or part of the systems functionality by describing the requirements aspect of one or several use cases.
- User-Interface Designer
 - Selects a set of use cases to demonstrate the essential interactions of the users with the system.



Key Artifacts used

- A "wish list" of what different stakeholders of the project expect or desire the system to include
- Vision document
- Use-Case Model
- Software Requirements Specification (SRS)
- Glossary
- Use-Case Storyboard
- User-Interface Prototype



■ **Identifying Use Cases**



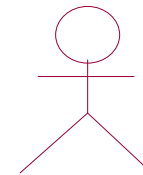
Use Case Model

- A model of the system's intended functions and its surroundings
- Serves as a contract between the customer and the developers
- Use-case model is the result of the Requirements discipline
 - Used as input to Analysis & Design and Test disciplines



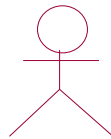
Actors

- An actor defines a coherent set of roles that users of the system can play when interacting with it
- An actor instance can be played by either an individual or an external system
- The following questions help identify actors:
 - Who does system administration?
 - Who is using the system?
 - Who is affected by the system?
 - Which external hardware or other systems use the system?

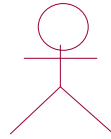




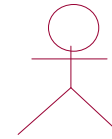
Actors



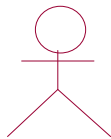
Project
Manager



Resource
Manager



System Admin

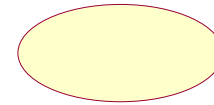


Backup system



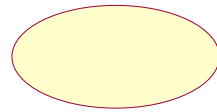
Use Cases

- A use case is a sequence of actions a system performs that yields an observable result of value to a particular actor.
- For each actor you have identified, what are the tasks in which the system would be involved?
- Does the actor need to be informed about certain occurrences in the system?
- Can all features be performed by the use cases you have identified?

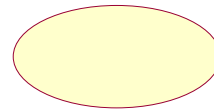




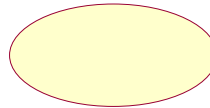
Use cases



Manage Project



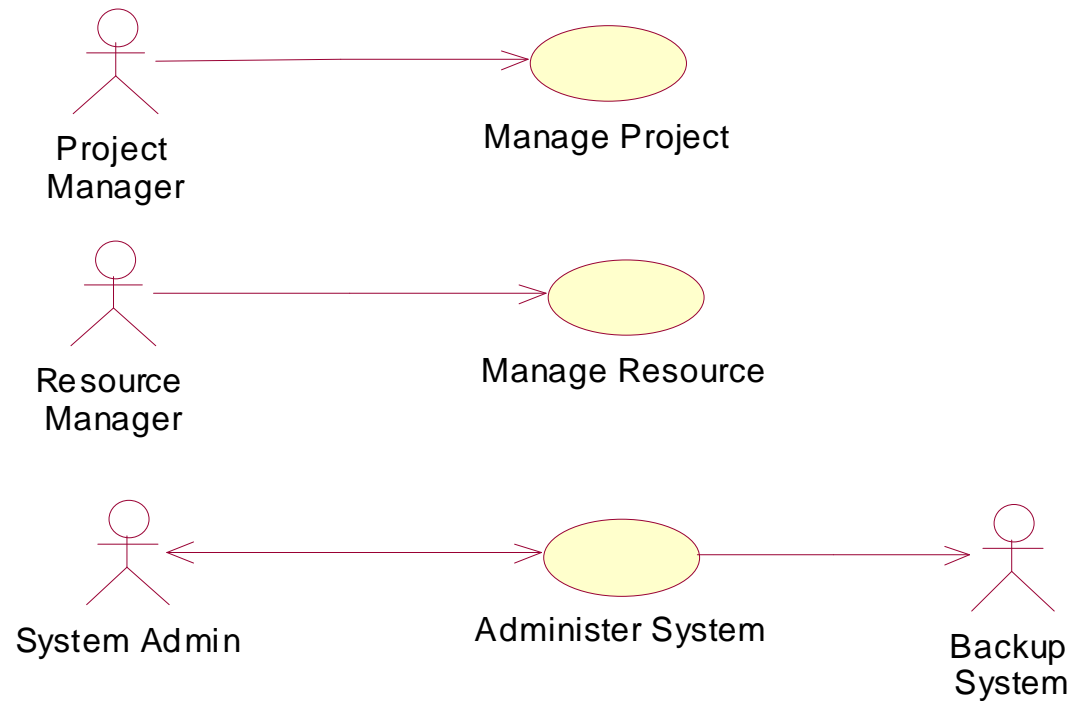
Manage Resource



Administer System

Communicate Associations

- Shows how actors and use cases are related and which actors participate in or initiate use cases.

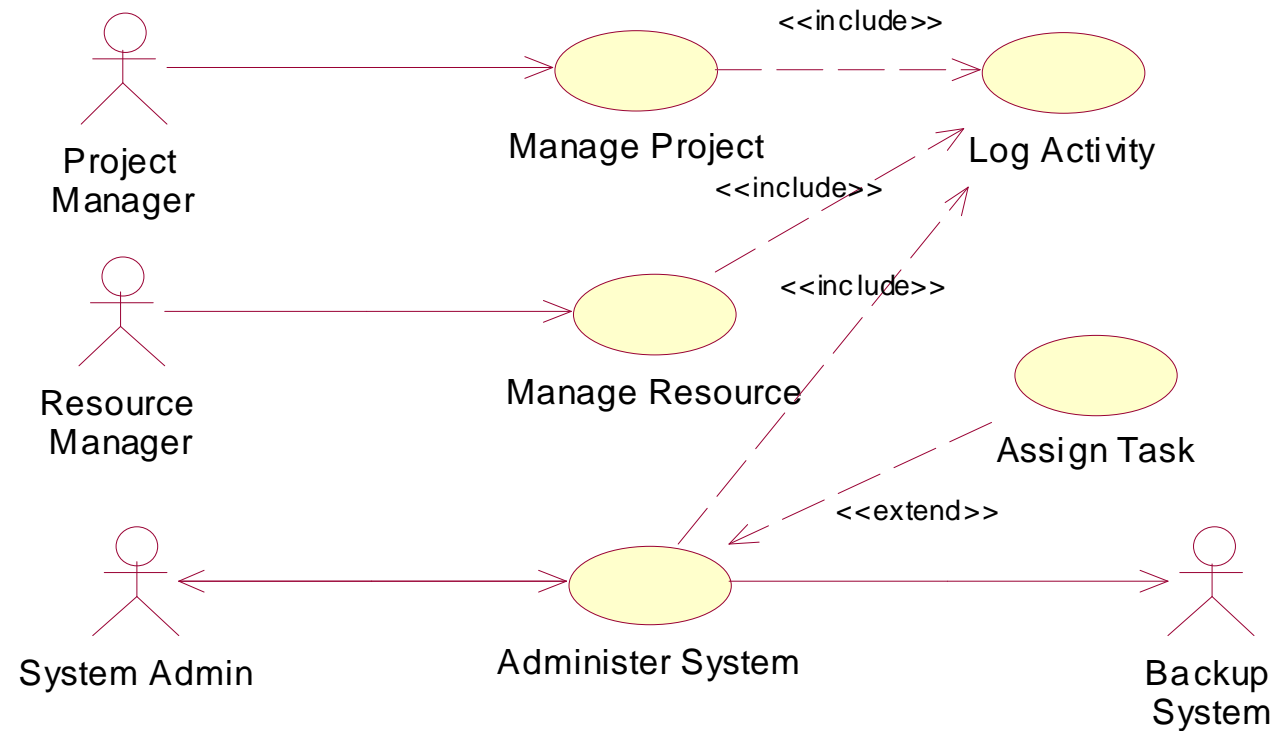


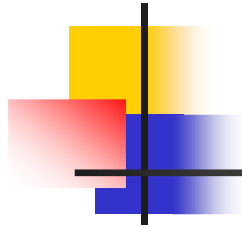


Use Case Collaborations

- The two major relationships between Use Cases are:
- Include Dependency :
 - A use case that is unconditionally incorporated into the execution of another use case.
 - One use case always calls another use case.
- Extends
 - A use case that conditionally interrupts the execution of another use case to augment its functionality.
 - One use case might call another use case.

Include and extend dependencies





Use Cases Documentation

- Name
- Brief Description
- Use-Case Diagrams
- Preconditions
- Post conditions
- Flow of Events
- Primary Flow
- Alternate Flow

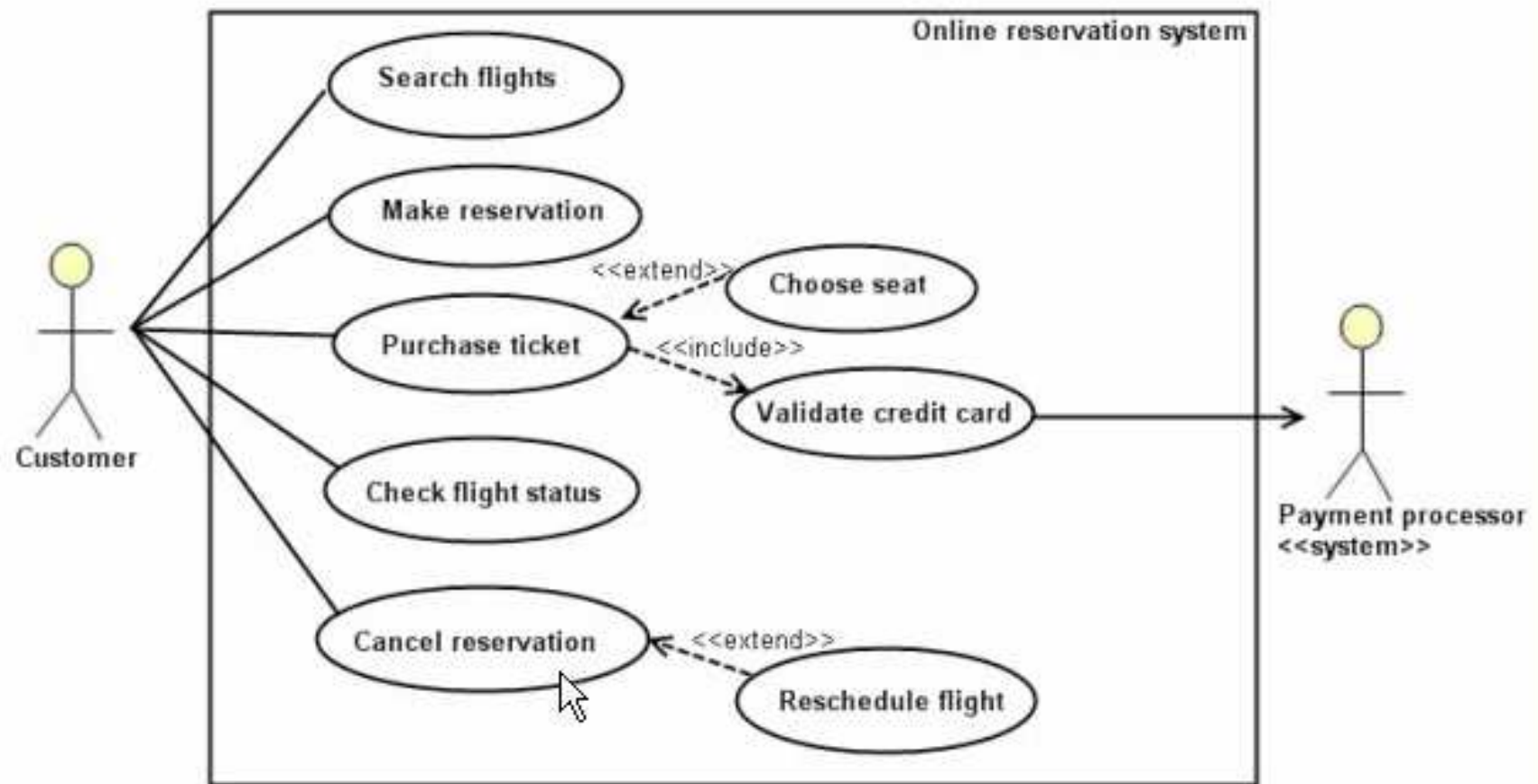


Exercise1: Online Flight ticket reservation system

In this exercise the participants are encouraged to do use case diagram based on the requirements given

1. The **customer** should be able to **search for flights** and **make a reservation**.
2. He should also be able to **check the flight status** (check any delays).
3. The customer should be able to **cancel the reservation**.
4. If the customer cancels the current reservation, he should be given an option to **reschedule the flight**.
5. The customer also should be able to **purchase a ticket online** (through Credit card) in which case the current system uses the **Payment processor system** for **credit card validation**.
6. While purchasing a ticket, the customer has an option to **choose a seat of his choice**.

Solution



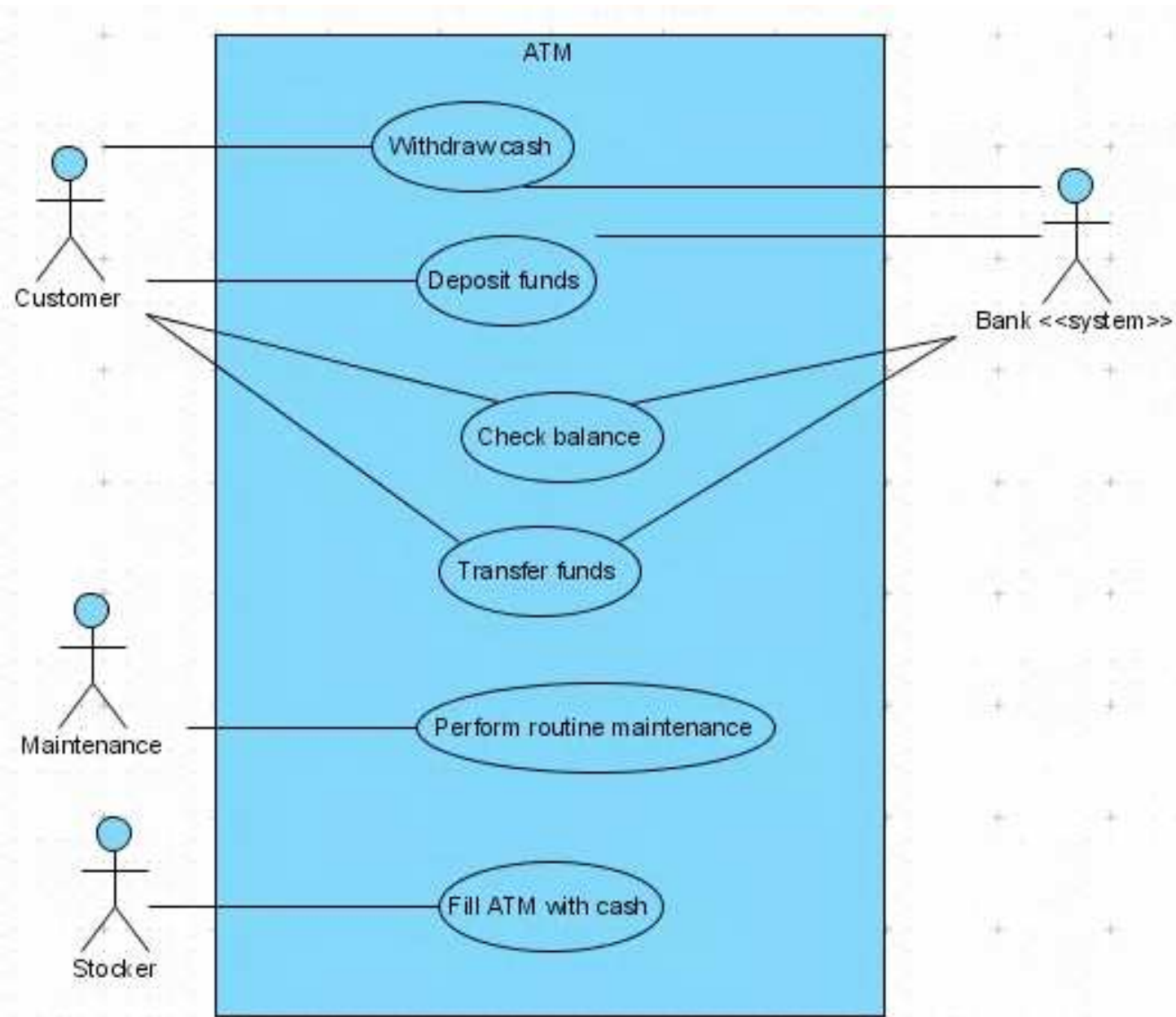
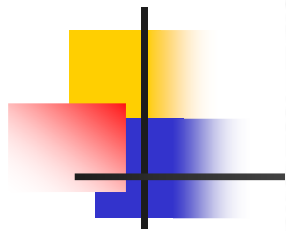


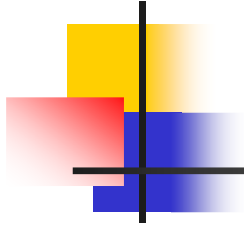
Exercise2: ATM System

In this exercise the participants are encouraged to do use case diagram based on the requirements given

1. The customer should be able to Withdraw money, deposit money and transfer money between accounts.
2. He should also be able to check the balance in his account.
3. The ATM also needs to interact with the bank system while the customer withdraws/deposits/transfers money or checks the balance in his account.
4. The ATM system needs a maintainance person to do the routine maintainance.
5. The Stocker does the job of filling the ATM with cash.

Solution





- Activity Diagrams



Activity Diagrams

- Activity Diagram is behavioral diagram describing the behavior of the system.
- Useful for checking the behavior of the system for each use case.



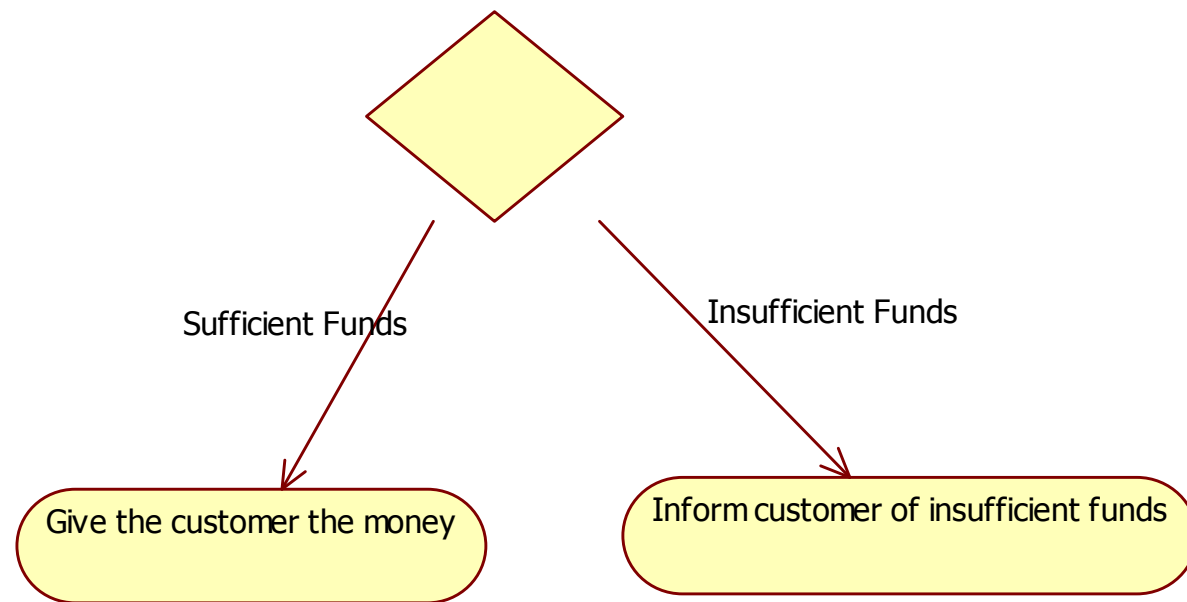
Activities and Transitions

- An activity is a step in a process where some work is getting done.
- A series of activities are linked by transitions, the arrows connecting each activity



Decisions

- The Activity Diagram diamond is a decision icon just as in a normal flowchart diagram.
- This notation can also involve a number of choices that the workflow can execute.





Initial and Final Action States

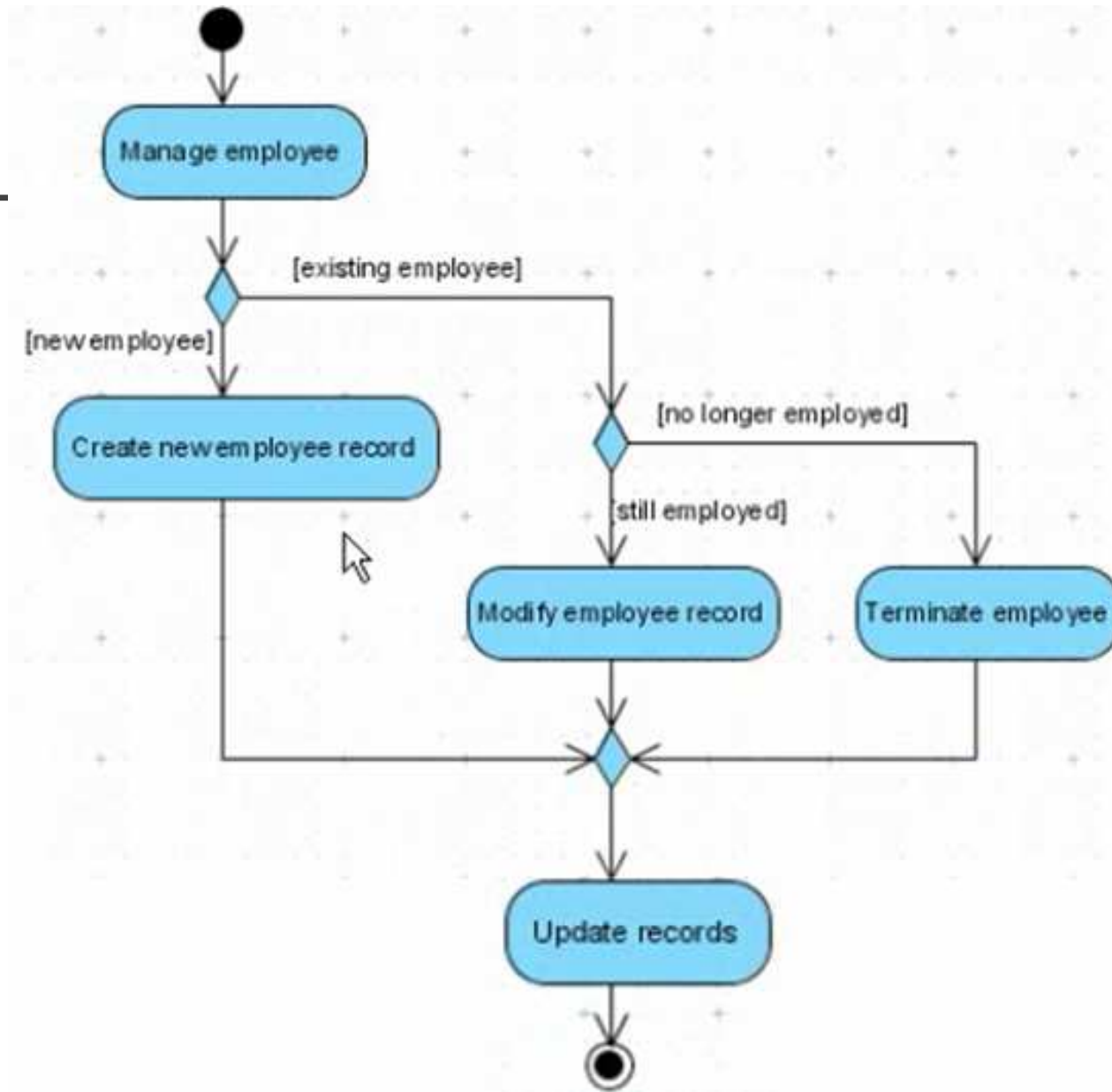
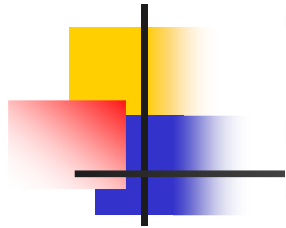
- The uml also provides notations to mark begin and end a Workflow



Start Point



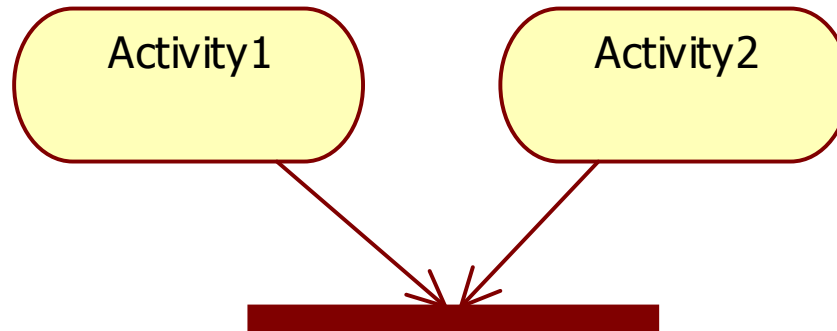
End Point

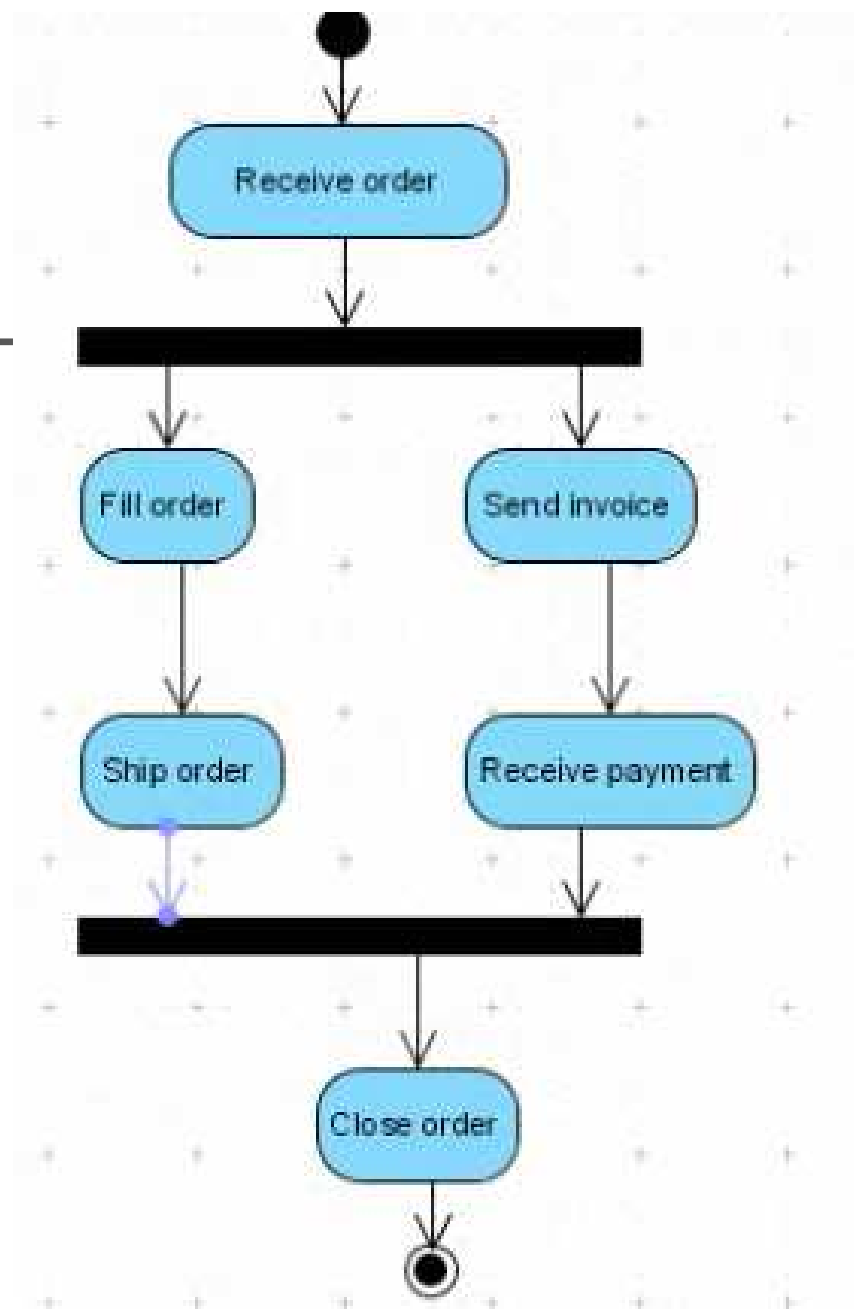


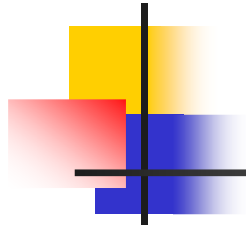


Concurrent Activities

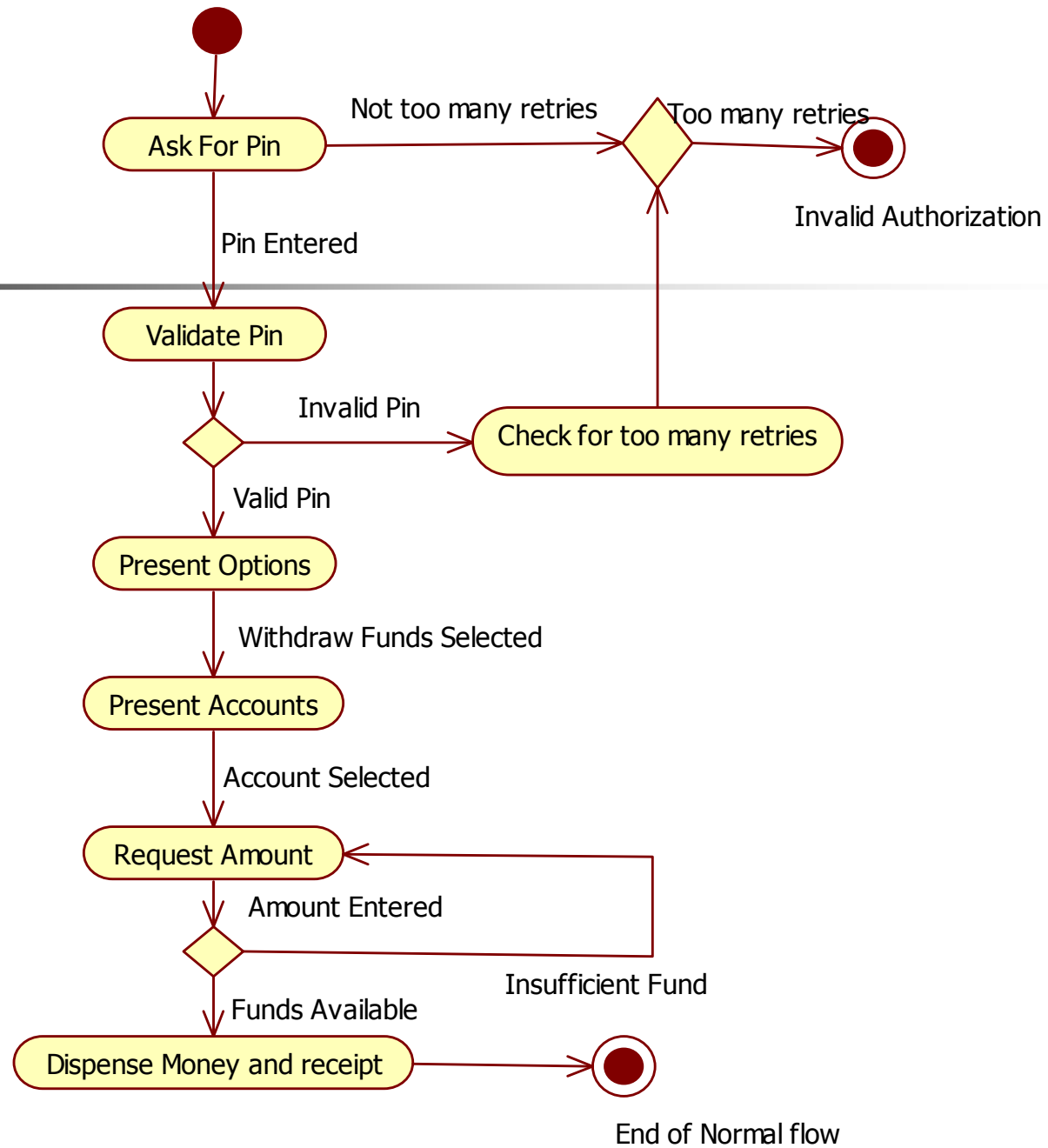
- Some activities happen in parallel. Such activities are called as concurrent activities.

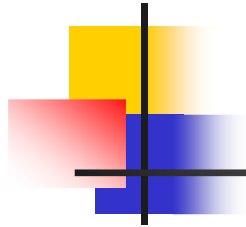




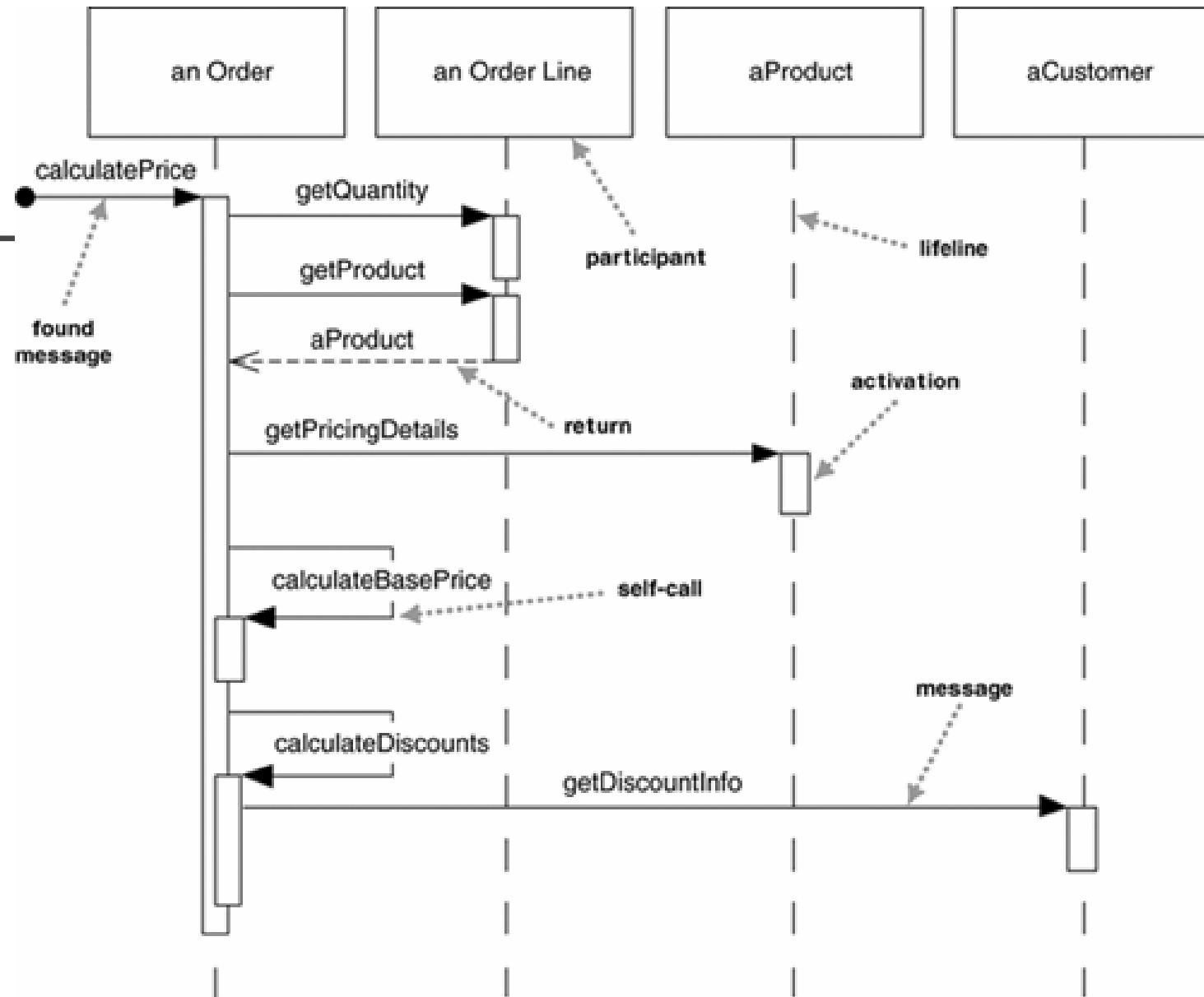
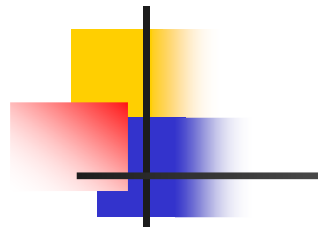


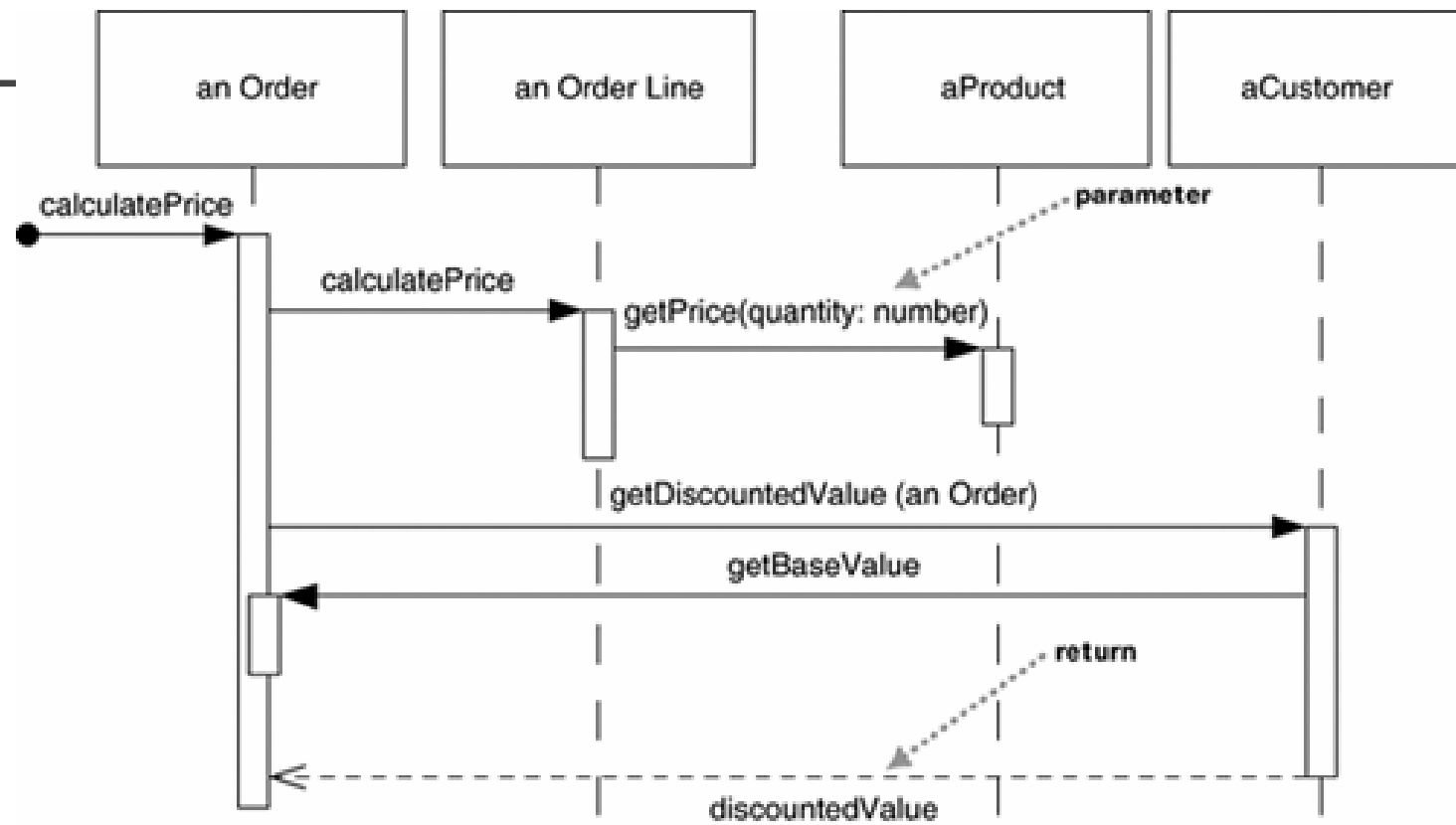
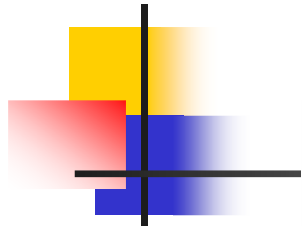
Activity Diagram for withdraw Money

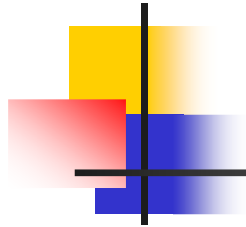




Sequence Diagrams

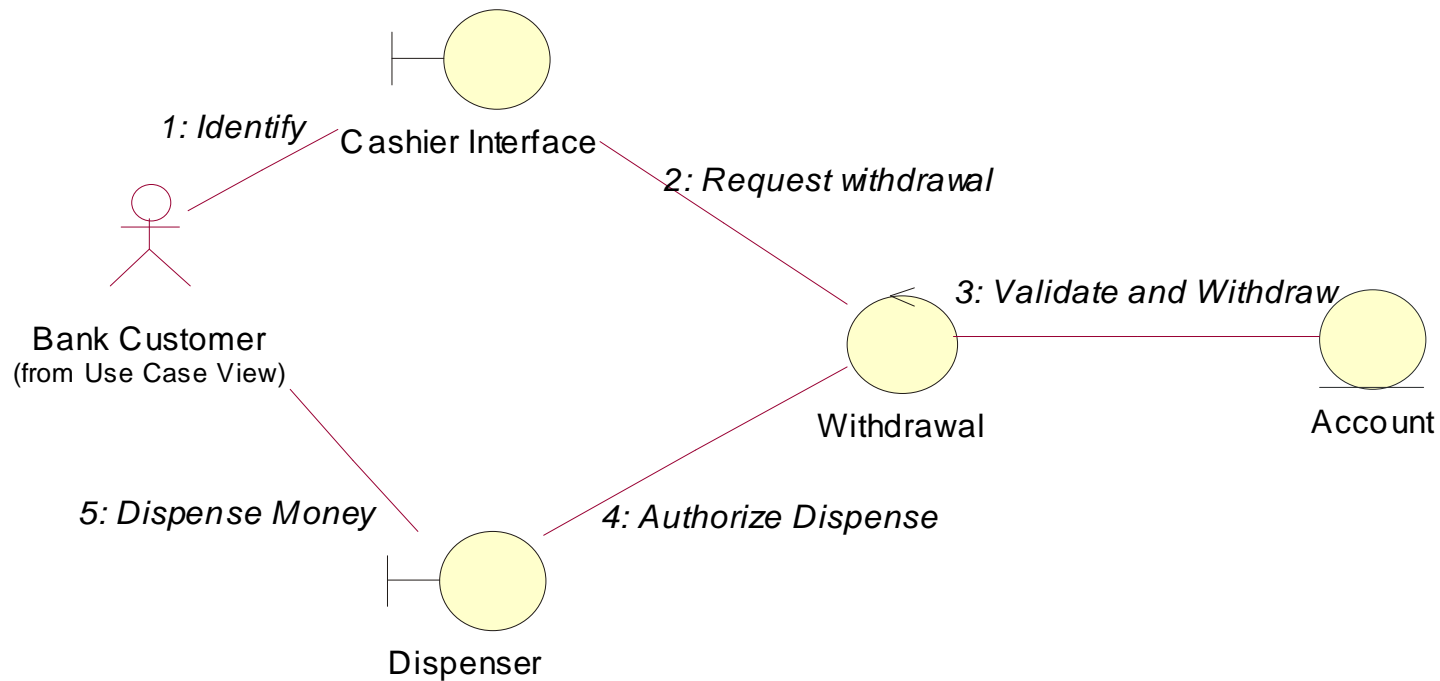




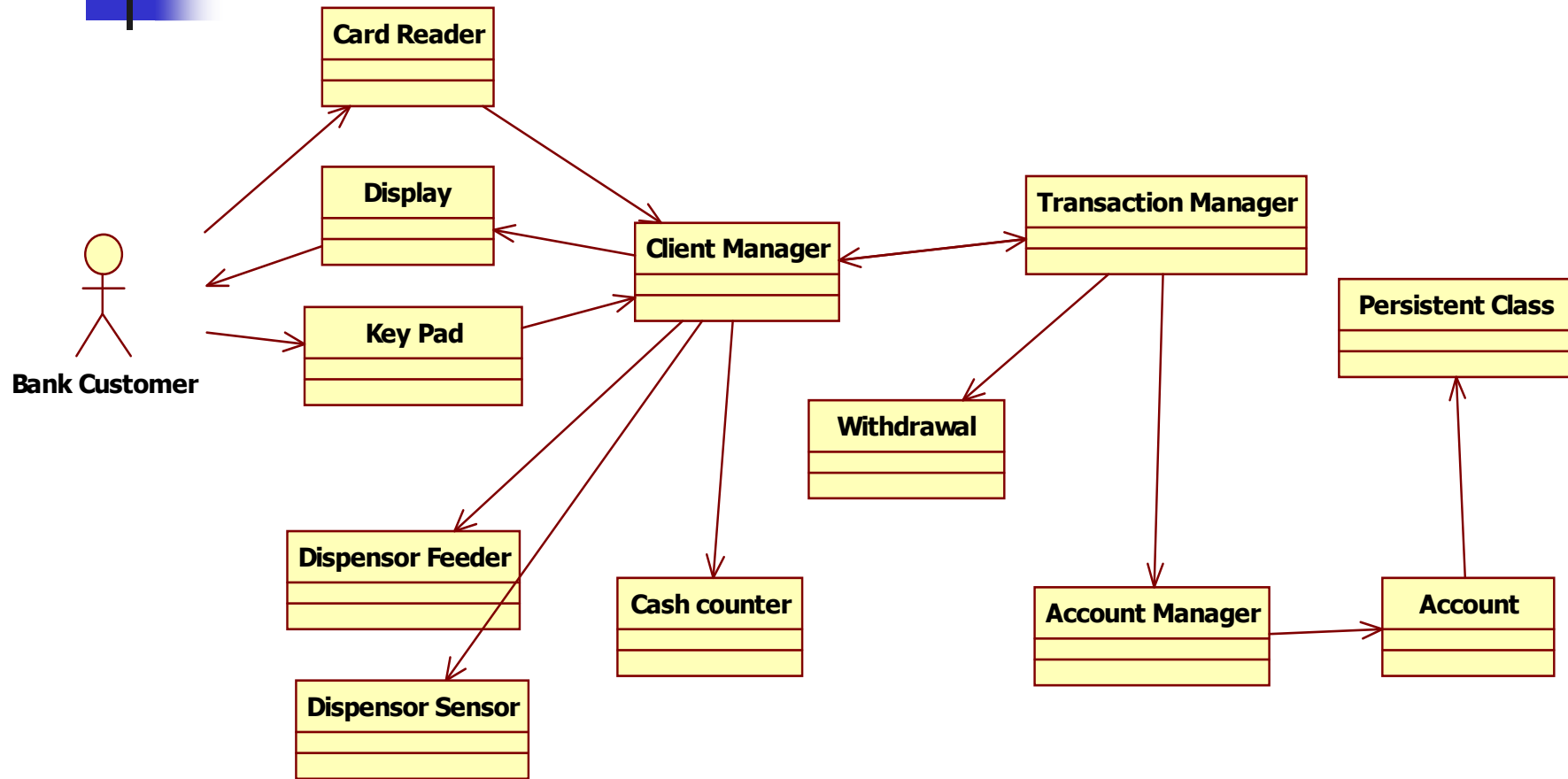


Steps for doing sequence diagram for
withdraw money use case

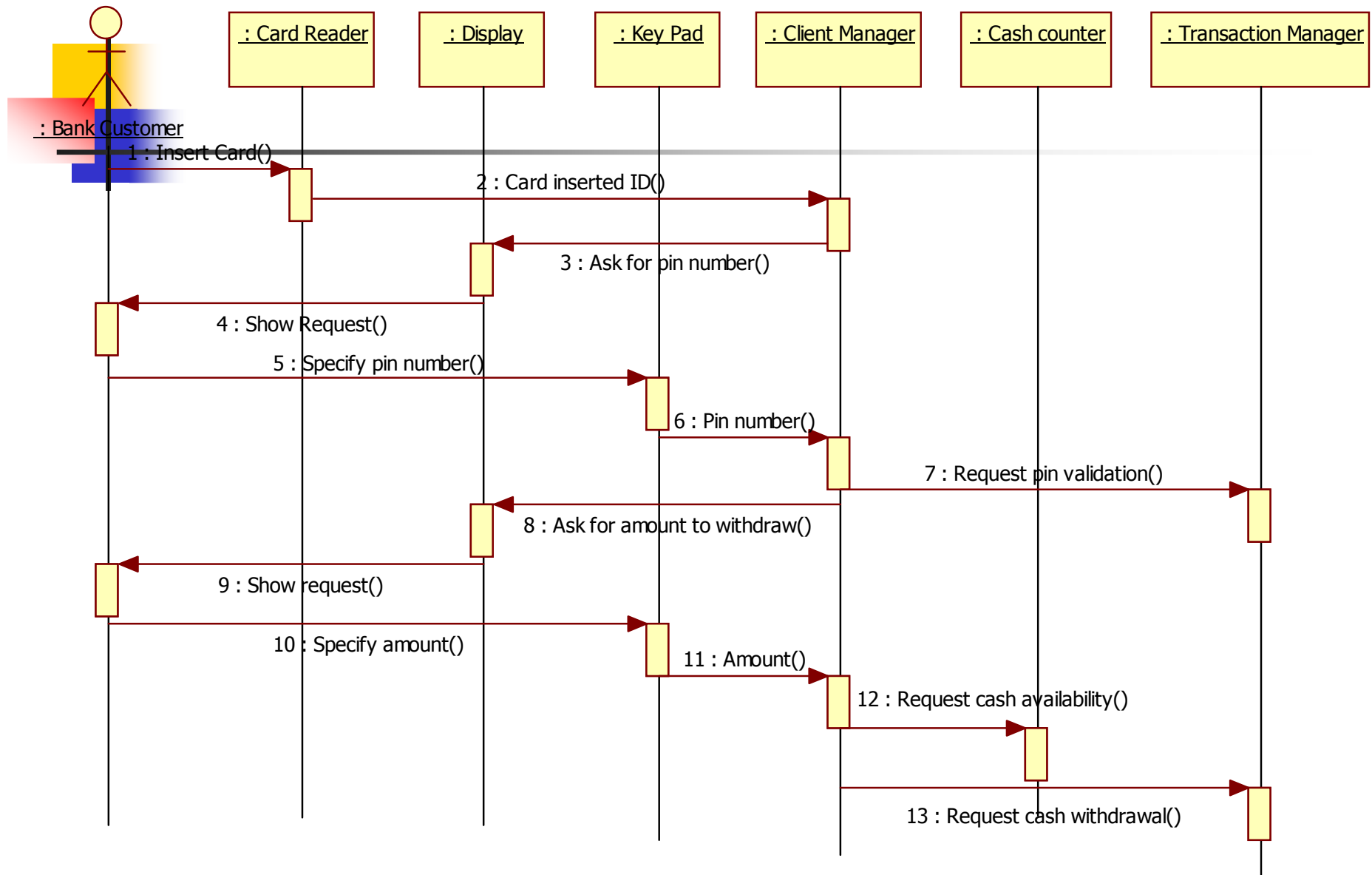
Robustness diagram



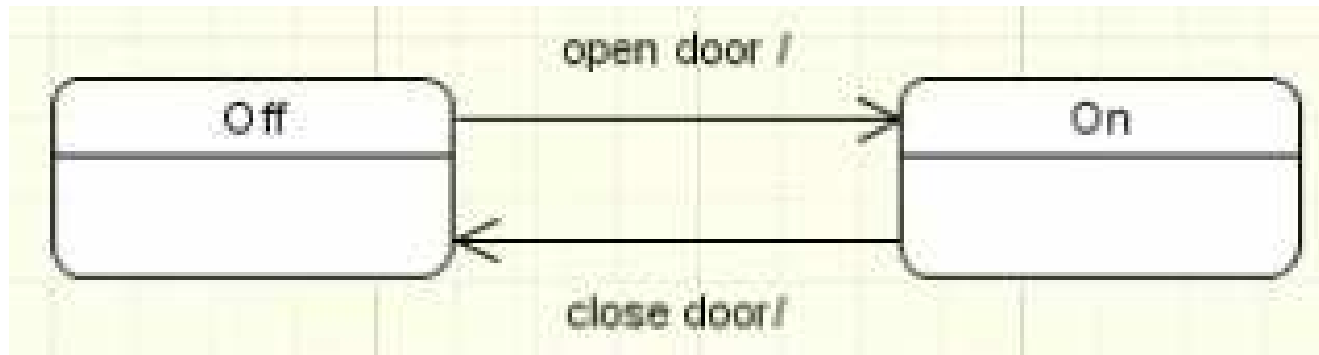
Class Diagram



Sequence Diagram



State Diagram





Activities for Fax machine

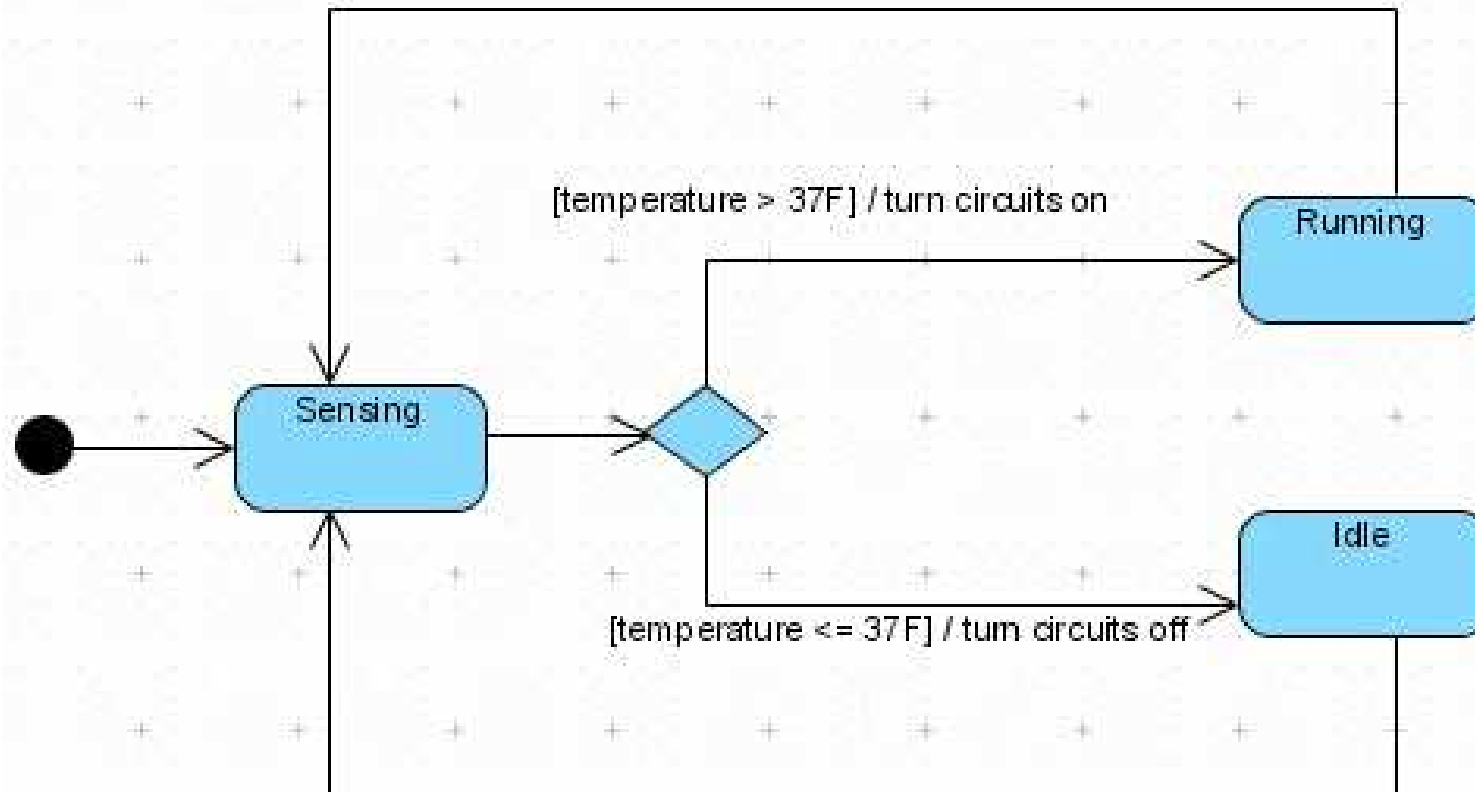
State

entry / activity
do / activity
exit / activity

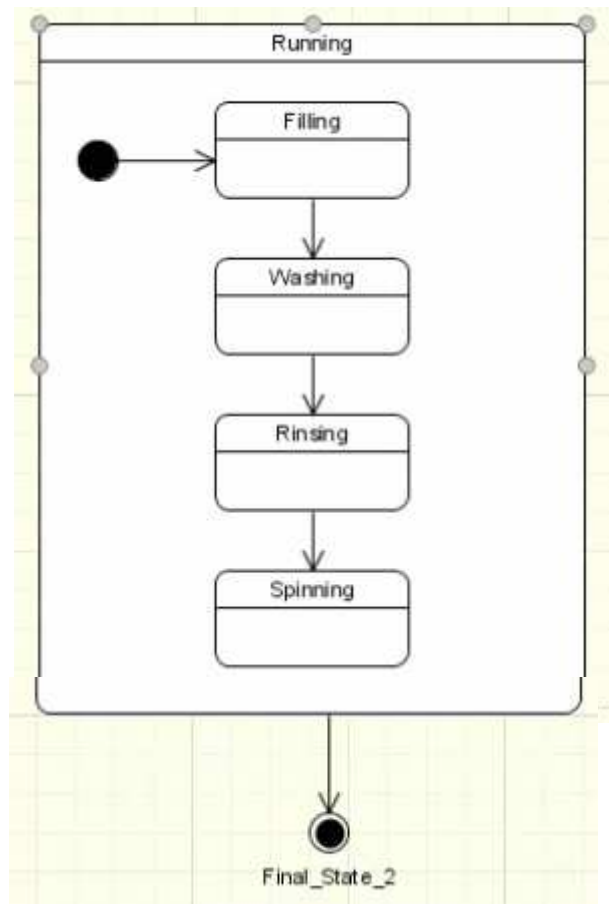
Receiving

entry / connect
do / print
exit / disconnect

States with Guard conditions



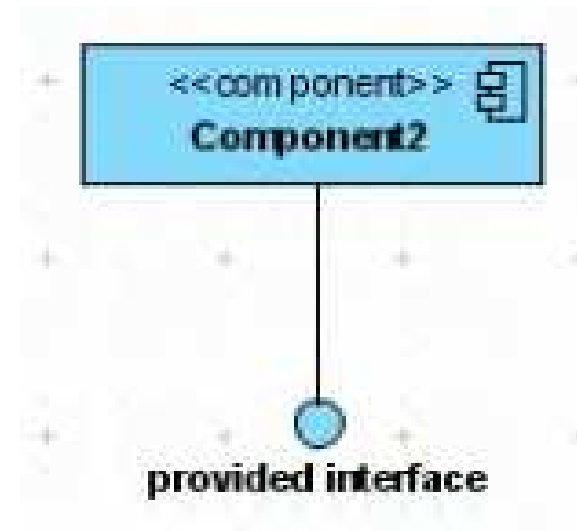
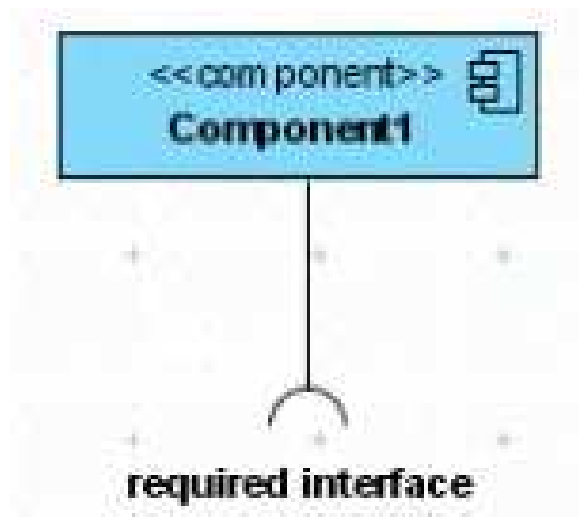
Composite States

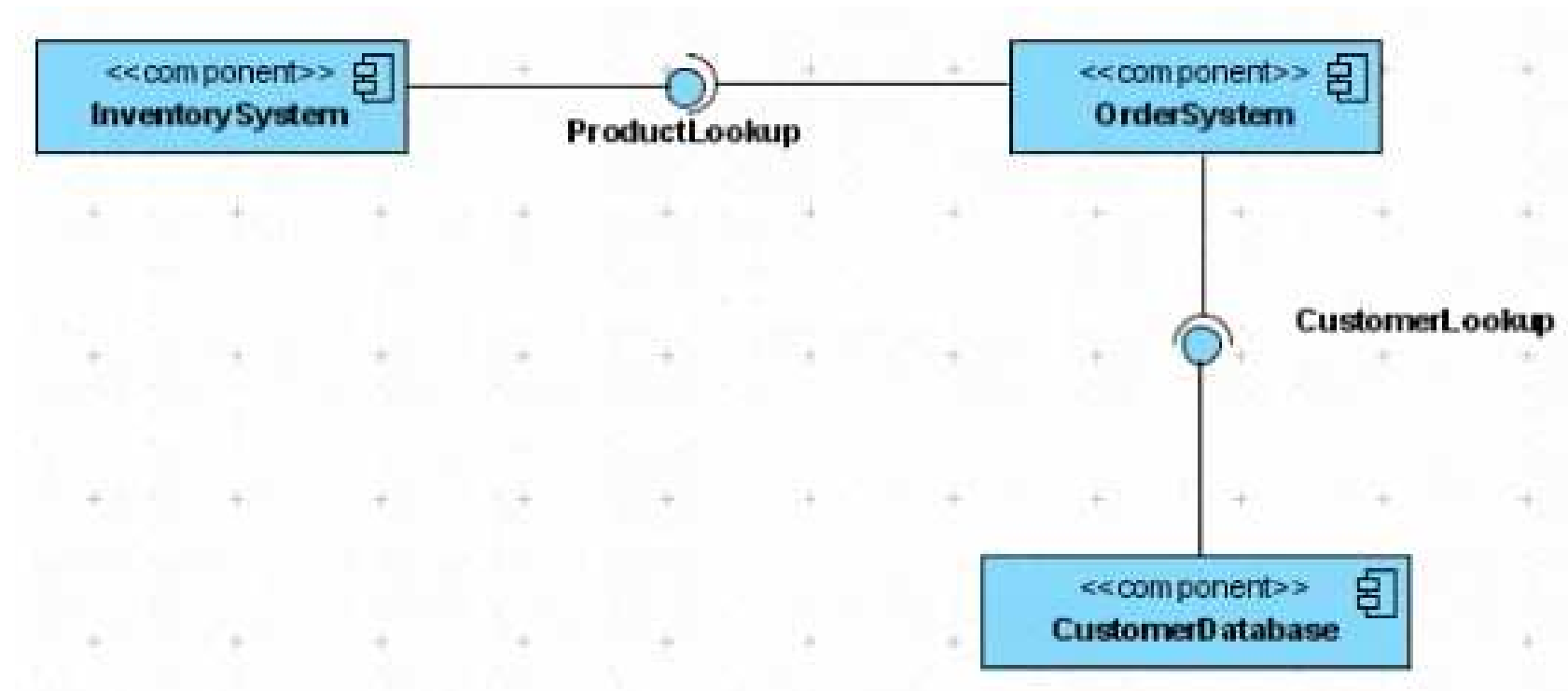
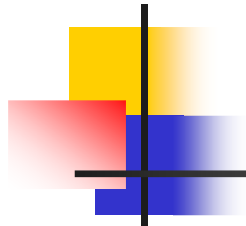


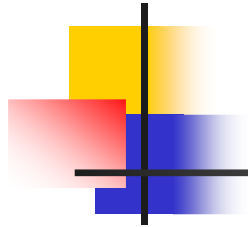


Package Diagrams: Demo

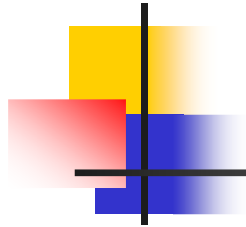
Component Diagram







Architecture



Deployment Diagram

