EDU

EAST DELTA
UNIVERSITY

CSE 411
Software Engineering and System Analysis and Design

Topic 3: UML Diagram

UML

UML → "Unified Modeling Language"

Language: express idea, not a methodology

Modeling:
Describing a software system at a high level of abstraction

Unified:
UML has become a world standard
Object Management Group (OMG): www.omg.org

- It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

- The UML uses mostly graphical notations to express the OO analysis and design of software projects.

- Simplifies the complex process of software design.

Types of UML Diagrams:

- Use Case Diagram
- Class Diagram
- Sequence Diagram
- State Diagram

This is only a subset of diagrams … but are most widely used

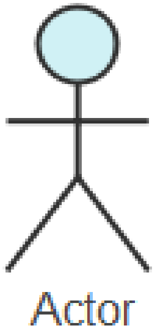
USE CASE DIAGRAM

- A use-case diagram is a set of use cases.

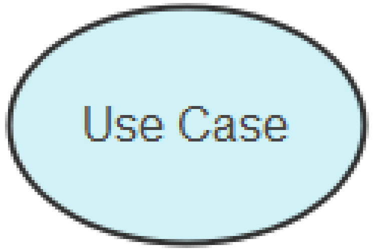
- A use case is a model of the interaction between
  - External users of a software product (actors) and
  - The software product itself
  - More precisely, an actor is a user playing a specific role

- Describing a set of user **scenarios**
- Capturing user requirements
- **Contract** between end user and software developers

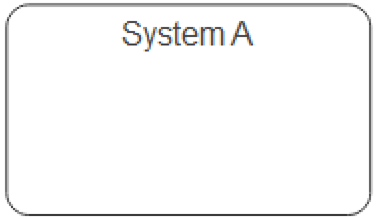
- **Actors:** A role that a user plays with respect to the system, including human users and other systems.

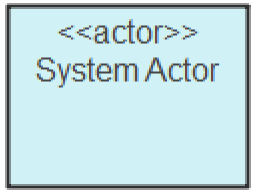


- **Use case:** A set of scenarios that describing an interaction between a user and a system, including alternatives.



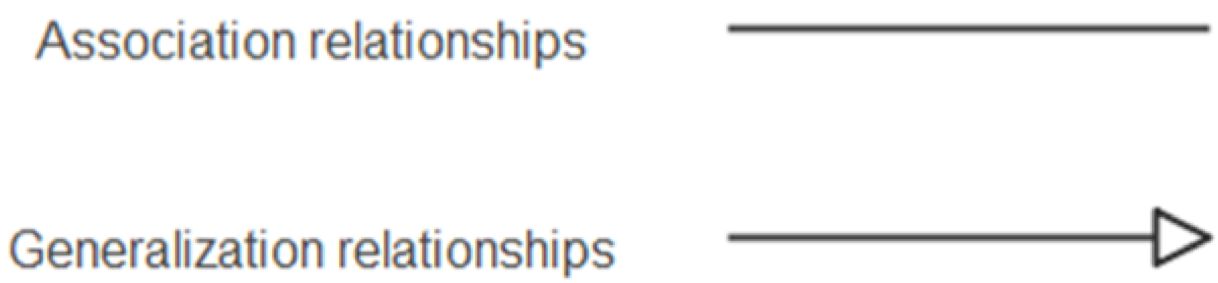
- **System boundary**: rectangle diagram representing the boundary between the actors and the system.





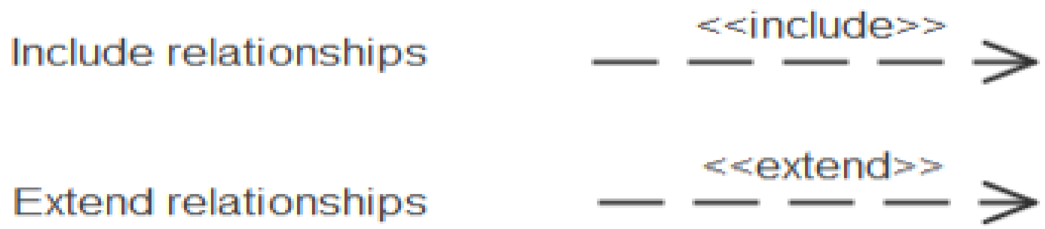
- **Association:** Communication between an actor and a use case; Represented by a solid line.

- **Generalization:** Relationship between one general use case and a special use case (used for defining special alternatives) Represented by a line with a triangular arrow head toward the parent use case.



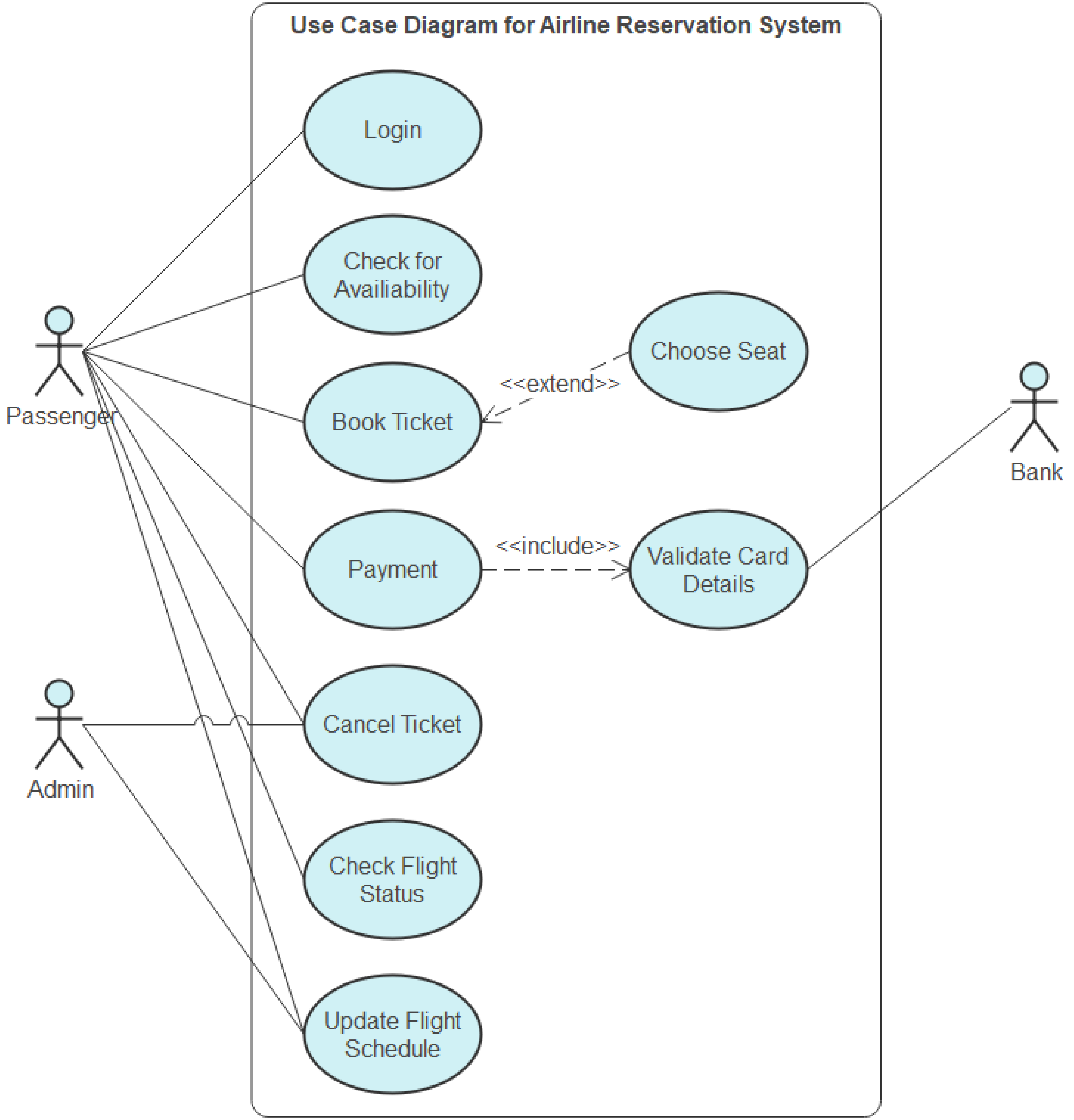
Include: A dotted line labeled **<<include>>** beginning at base use case and ending with an arrows pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use “include” in stead of copying the description of that behavior.

Extend: A dotted line labeled **<<extend>>** with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares “extension points”.



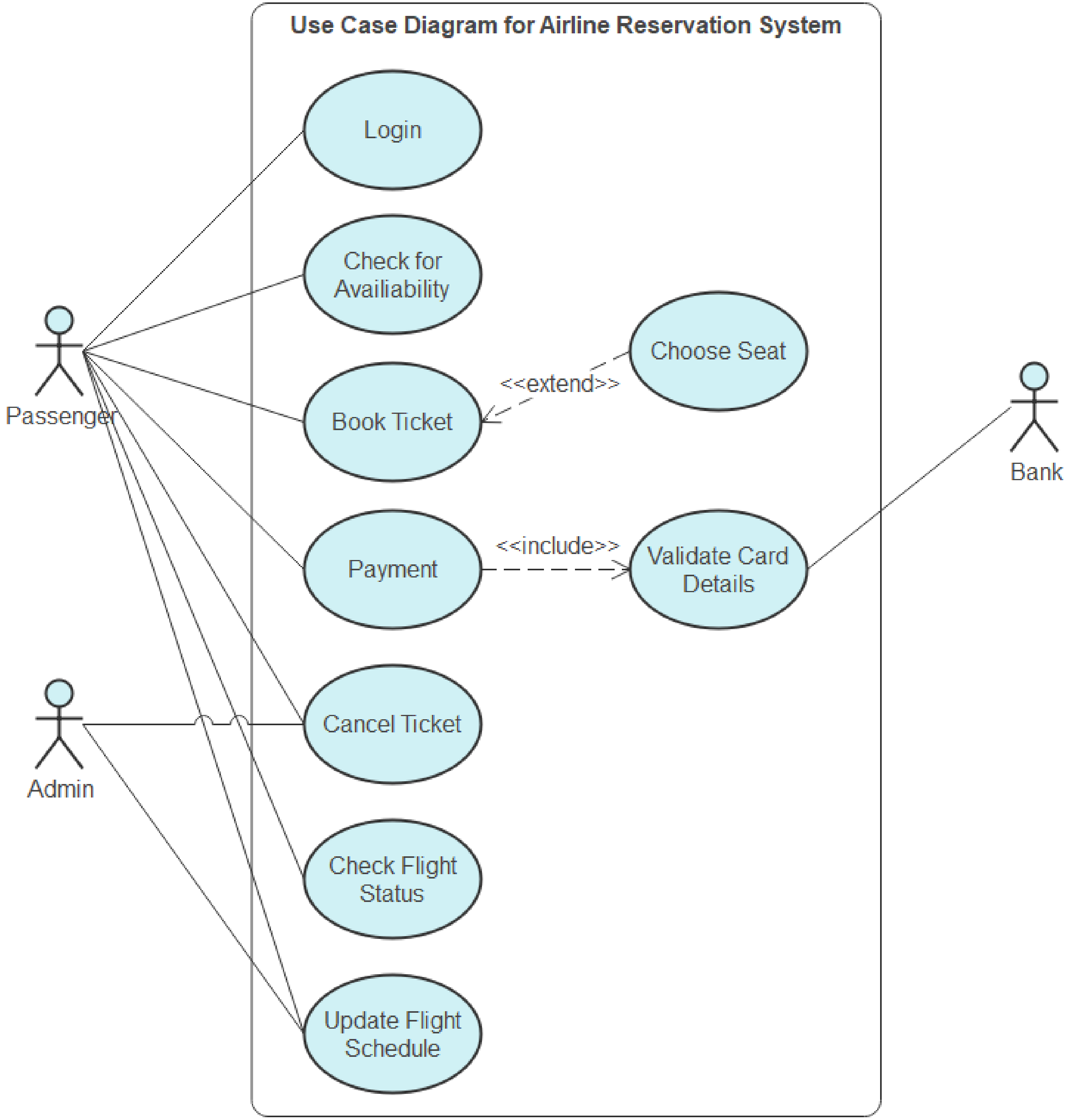
Case 1:

Passenger wants reserve a seat on airline. He needs to login the airline reservation system. First he needs to check whether the seat is available or not. Then he will book a ticket where optionally he can choose his seat. Next he need to pay by card. The bank will verify his card details. However, the admin of the system can cancel his ticket, check flight status and update flight schedule on the system



Case 1:

Passenger wants reserve a seat on airline. He needs to login the airline reservation system. First he needs to check whether the seat is available or not. Then he will book a ticket where optionally he can choose his seat. Next he need to pay by card. The bank will verify his card details. However, the admin of the system can cancel his ticket, check flight status and update flight schedule on the system.

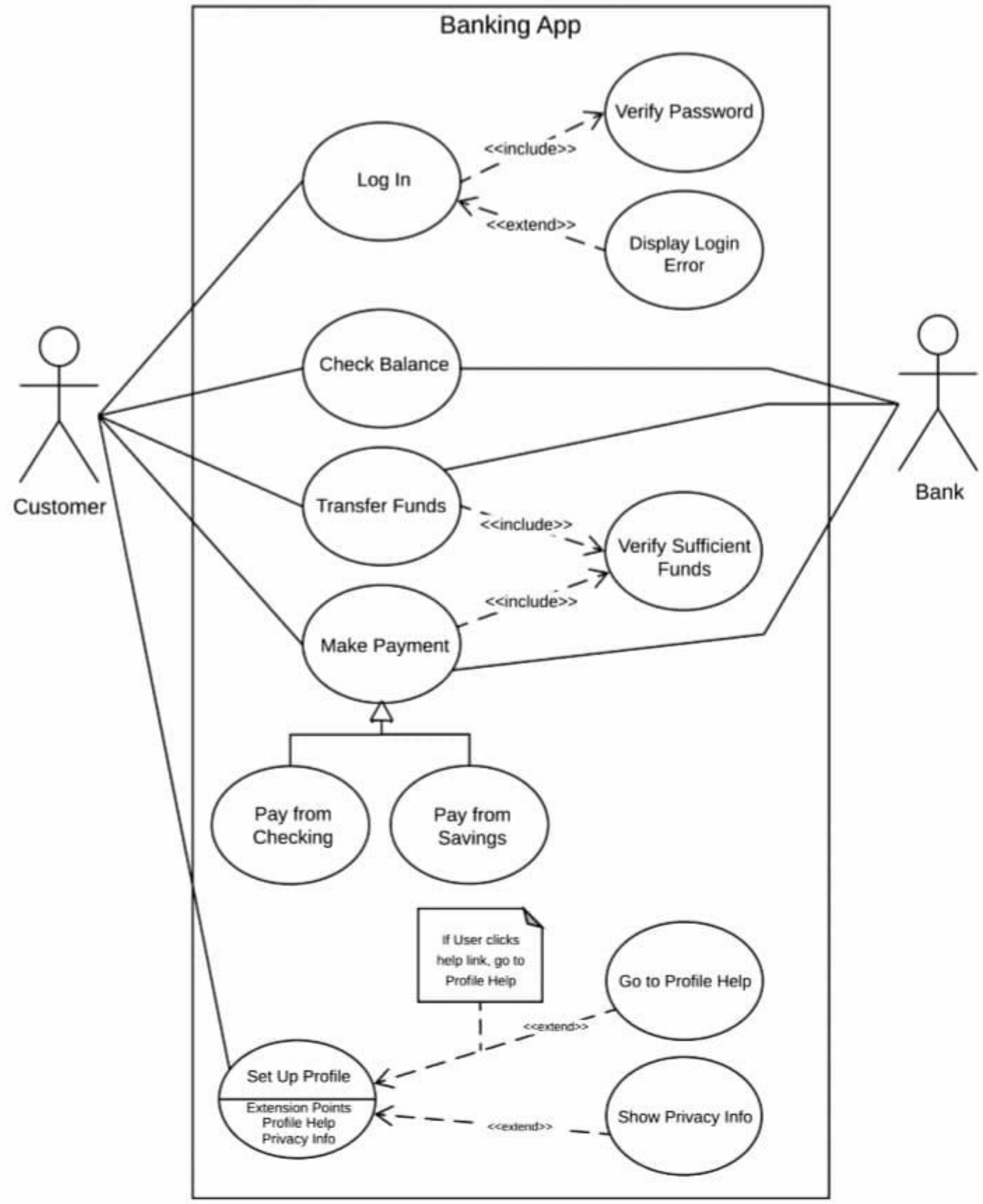


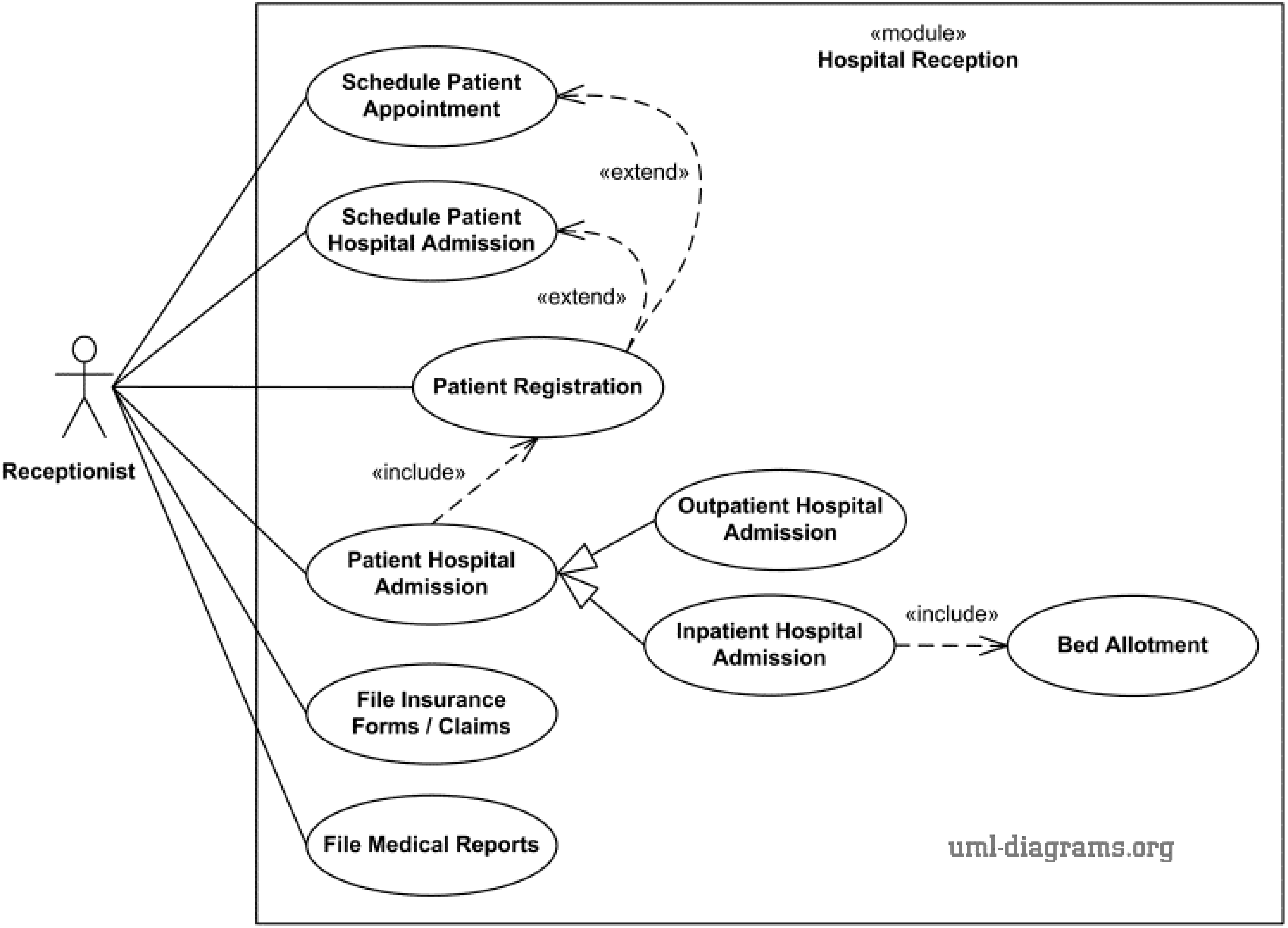
Case 2:

Customer wants to use a banking app. He needs to login the banking system. The login function must verify the customer id and password and may display error message if login fails. In the system customer and bank both party can check balance, transfer funds and obviously can make payment optionally from checking or savings. The process of transferring the fund and make payment essentially needs to verify if the sufficient fund is available or not. In the system customer can set his profile.

Case 2:

Customer wants to use a banking app. He needs to login the banking system. The login function must verify the customer id and password and may display error message if login fails. In the system customer and bank both party can check balance, transfer funds and obviously can make payment optionally from checking or savings. The process of transferring the fund and make payment essentially needs to verify if the sufficient fund is available or not. In the system customer can set his profile.




«module»
Hospital Reception
Receptionist
Schedule Patient Appointment
Schedule Patient Hospital Admission
«extend»
«extend»
Patient Registration
«include»
Patient Hospital Admission
Outpatient Hospital Admission
Inpatient Hospital Admission
«include»
Bed Allotment
File Insurance Forms / Claims
File Medical Reports
uml-diagrams.org

Case Study of Use case diagram

Suppose you need to make a software in which when the user confirms order and confirmation need the confirmation depends upon the product selection, calculation of price with tax and payment. Payment can be through PayPal or credit card.

Draw the use case diagram.

Solution:

First step: Identify the requirements and put them in an oval shape.

Second step: Identify the users

Third step: Identify the relationships include, extend and generalization(parent/child).

Fourth step: Draw the Boundary

Fifth step: Connect the actor/user with the use case.

Solution:

For example, in this case study following are the functional requirements;

The user can confirm the order

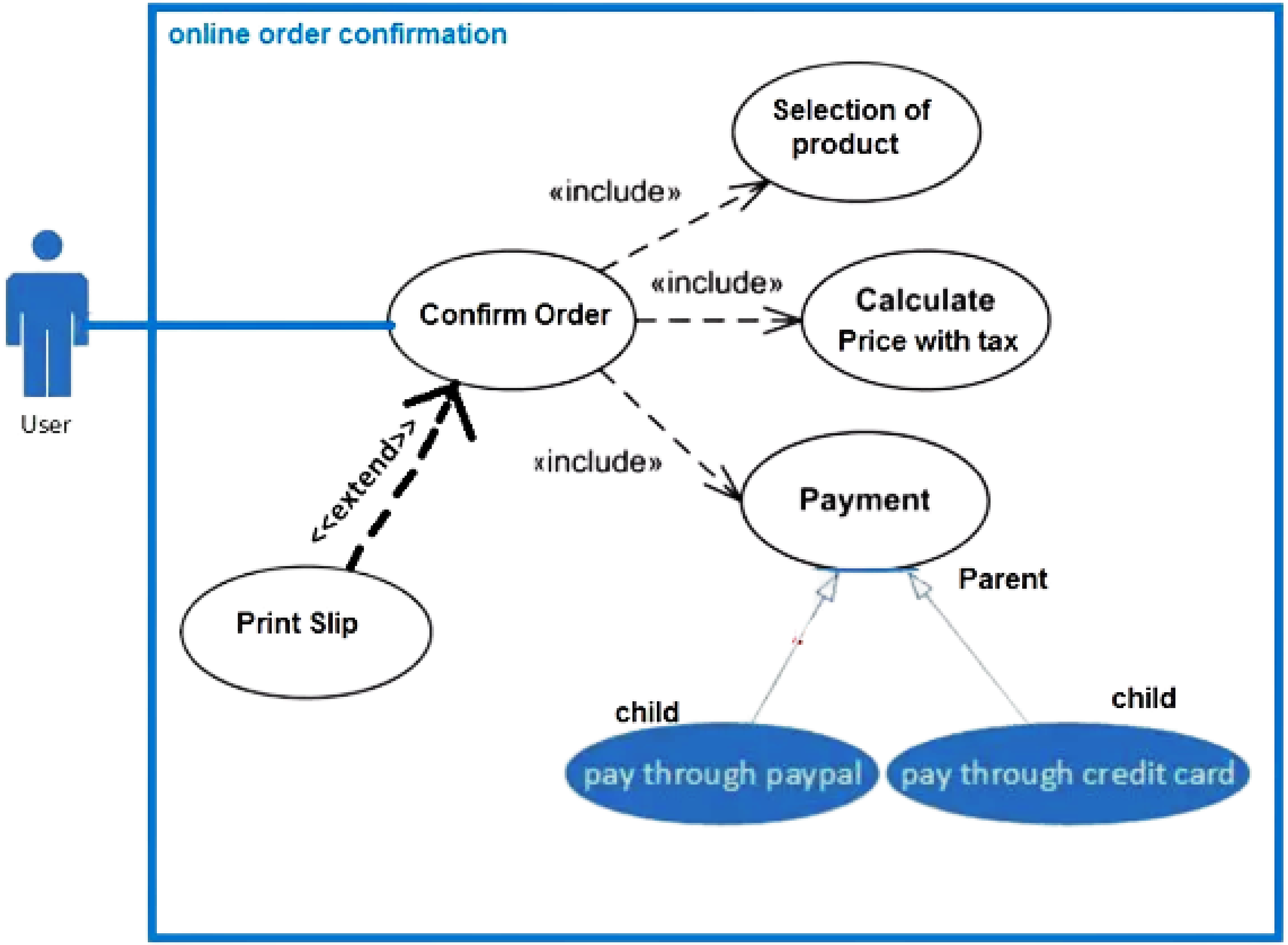
The order confirmation must be followed by the selection of the product.

The order confirmation must be followed by the Calculate price with tax.

The order confirmation must be followed by the Payment.

The payment can be done through PayPal or credit card.

Solution:


online order confirmation
User
Confirm Order
«include»
Selection of product
«include»
Calculate Price with tax
«include»
Payment
Parent
<<extend>>
Print Slip
child
child
pay through paypal
pay through credit card

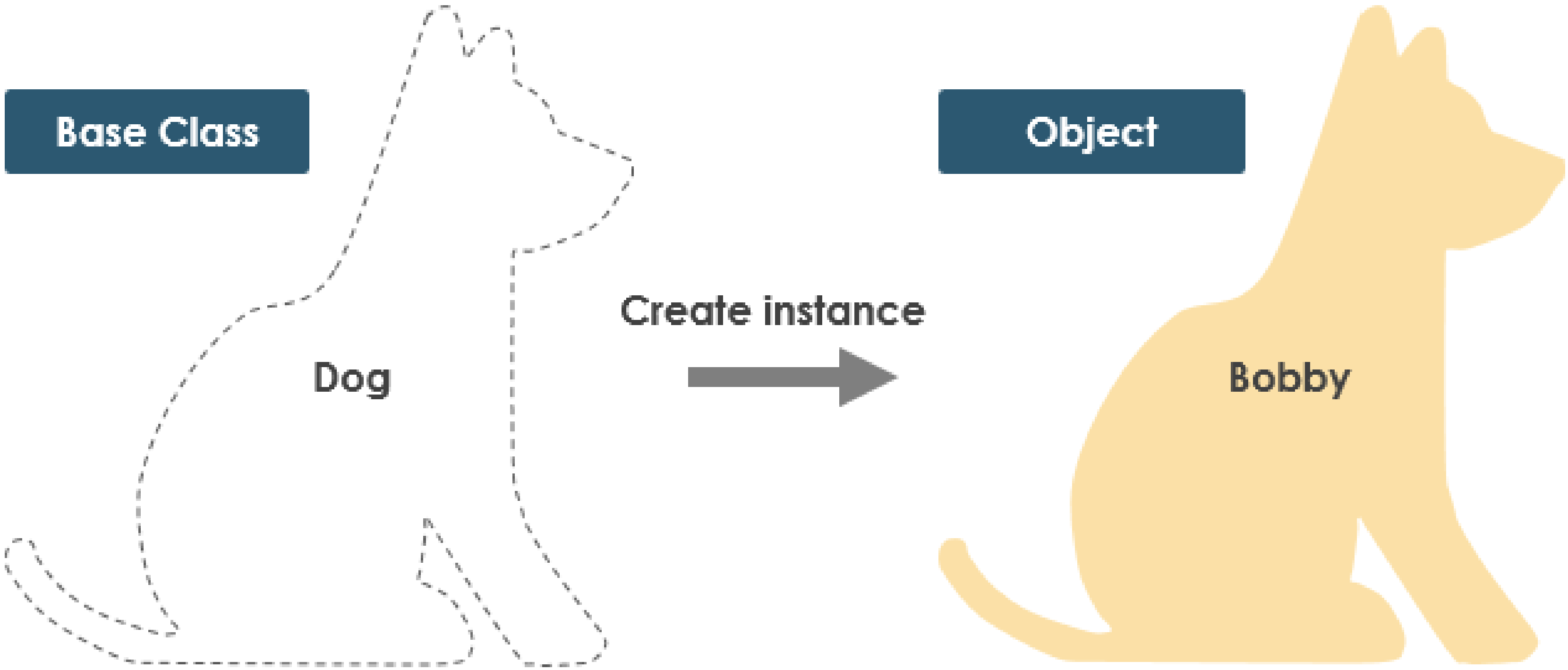
CLASS DIAGRAM

- **A class diagram depicts classes and their interrelationships**
- Used for describing structure and behavior in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

- **A Class is a blueprint for an object.** Objects and classes go hand in hand.

- Classes describe the type of objects, while objects are usable instances of classes.

- Each Object was built from the same set of blueprints and therefore contains the same components (properties and methods).



Base Class

Dog

Create instance

Object

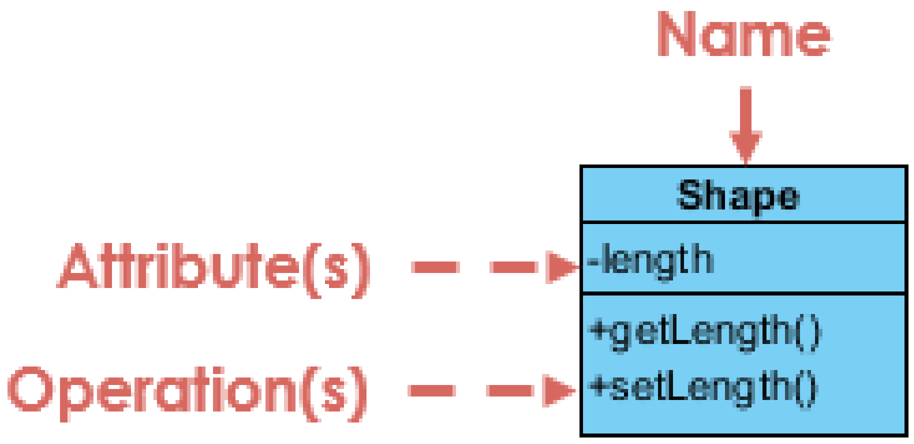
Bobby

Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

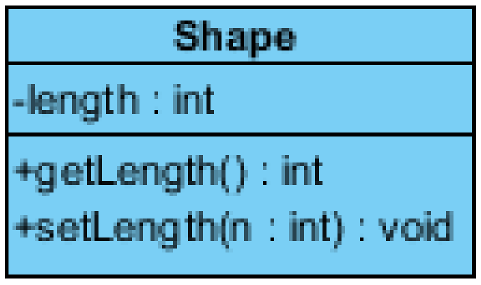
Property Values	Methods
Color: Yellow	Sit
Eye Color: Brown	Lay Down
Height: 17 in	Shake
Length: 35 in	Come
Weight: 24 pounds	

UML Class Notation

A class represent a concept which **encapsulates state (attributes) and behavior (operations).** Each attribute has a type. Each operation has a signature. The class name is the only mandatory information.



Class without signature



Class with signature

Class Name:

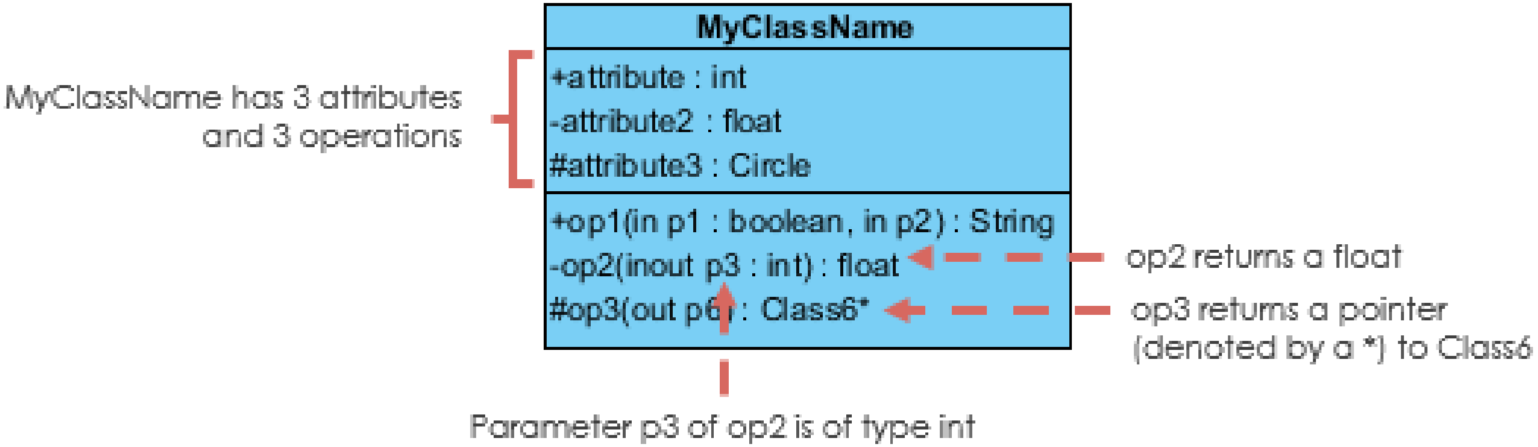
The name of the class appears in the first partition.

Class Attributes:

- ***Attributes are shown in the second partition.***
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

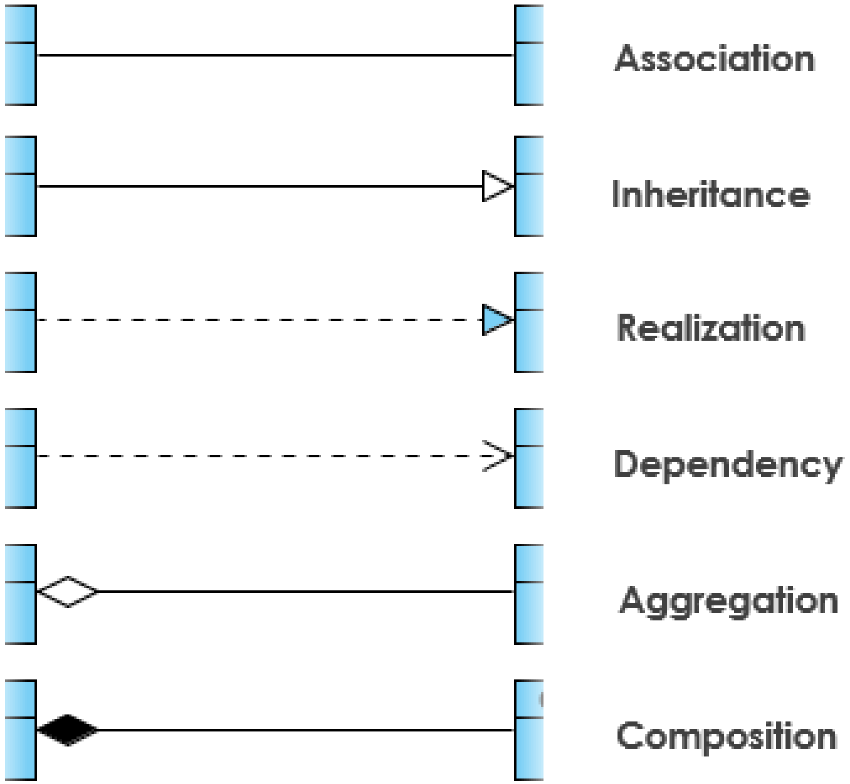
Class Operations (Methods):

- ***Operations are shown in the third partition.*** They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code

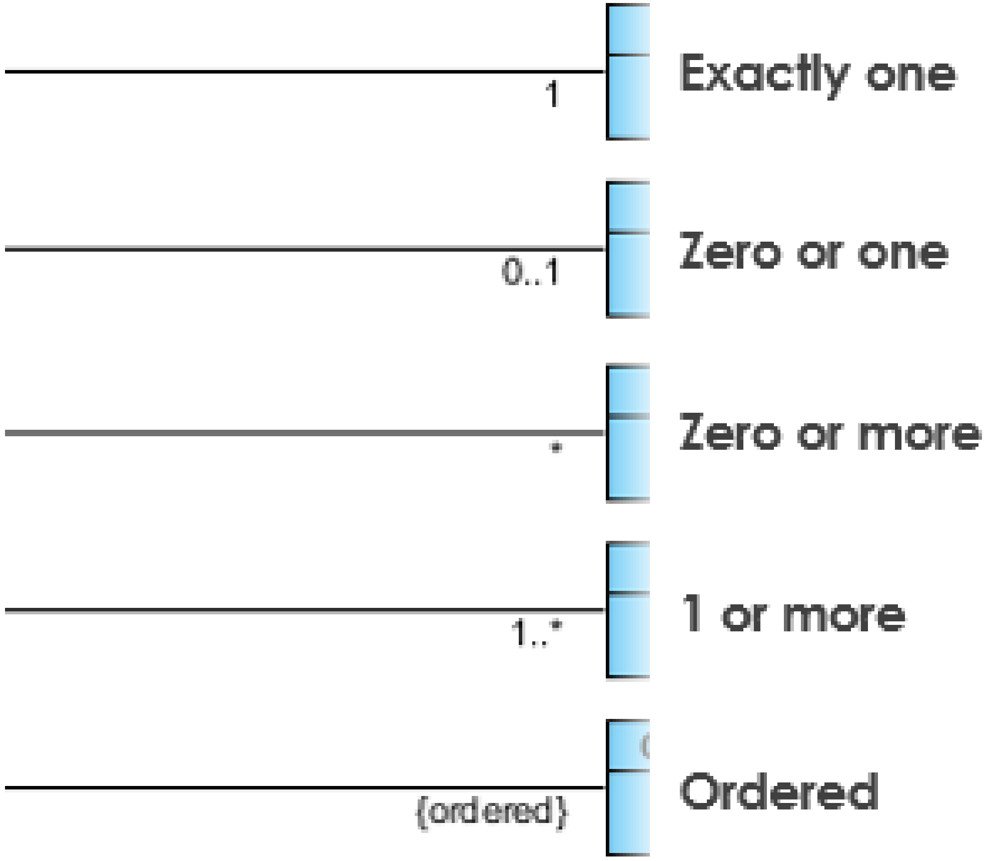


MyClassName
+attribute : int
-attribute2 : float
#attribute3 : Circle
+op1(in p1 : boolean, in p2) : String
-op2(inout p3 : int) : float
#op3(out p6) : Class6*
MyClassName has 3 attributes and 3 operations
op2 returns a float
op3 returns a pointer (denoted by a *) to Class6
Parameter p3 of op2 is of type int

Relationships

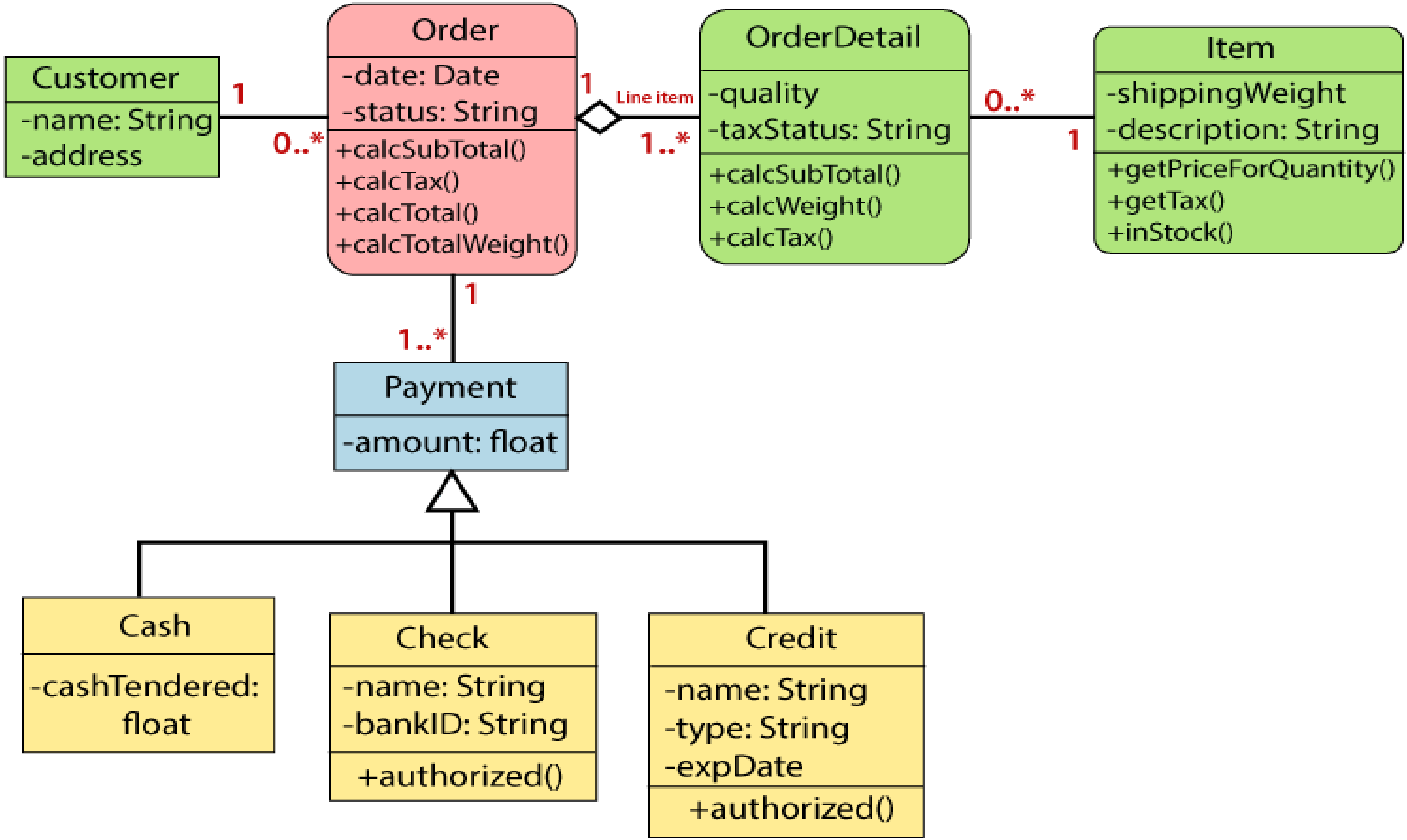


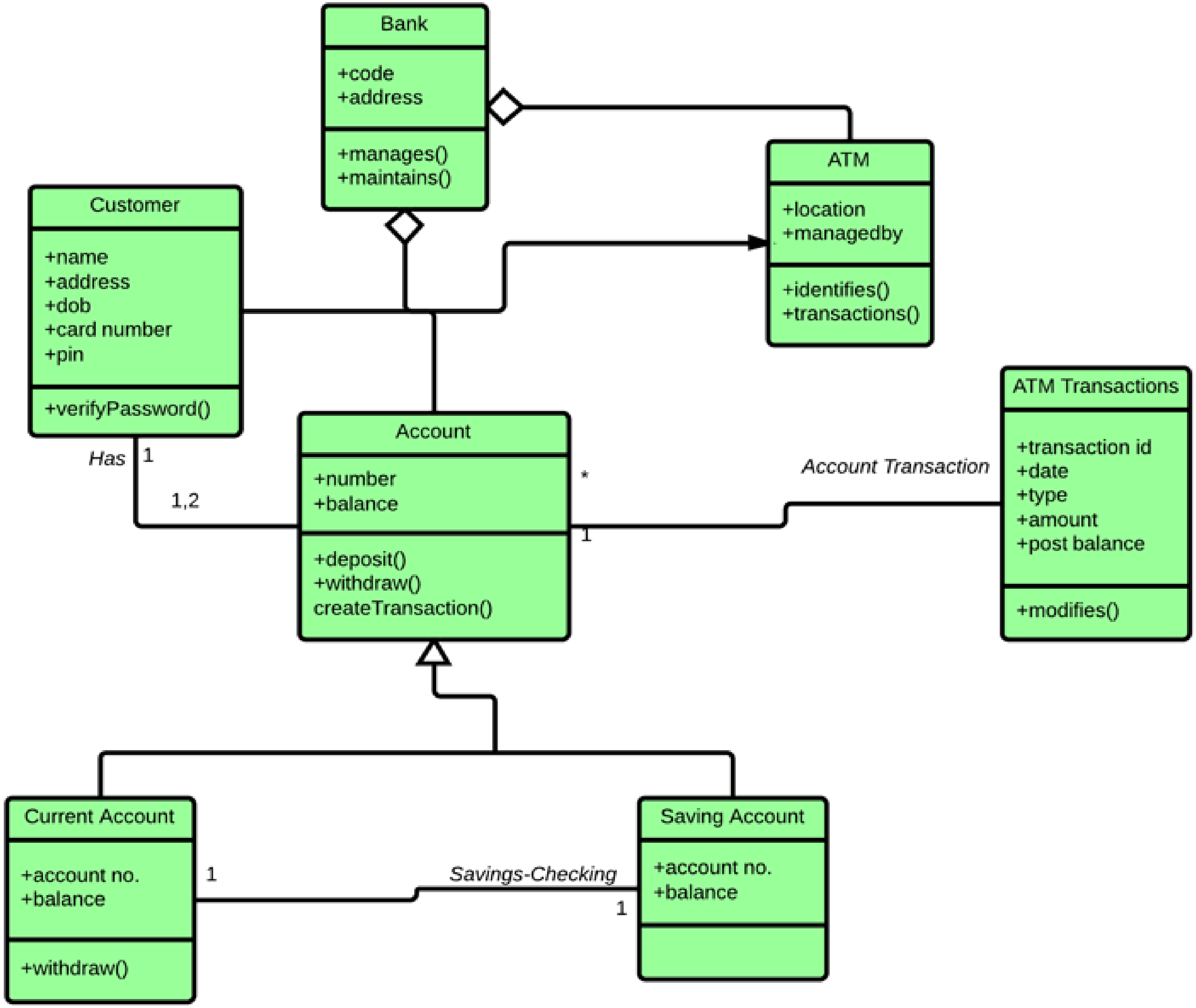
Cardinality

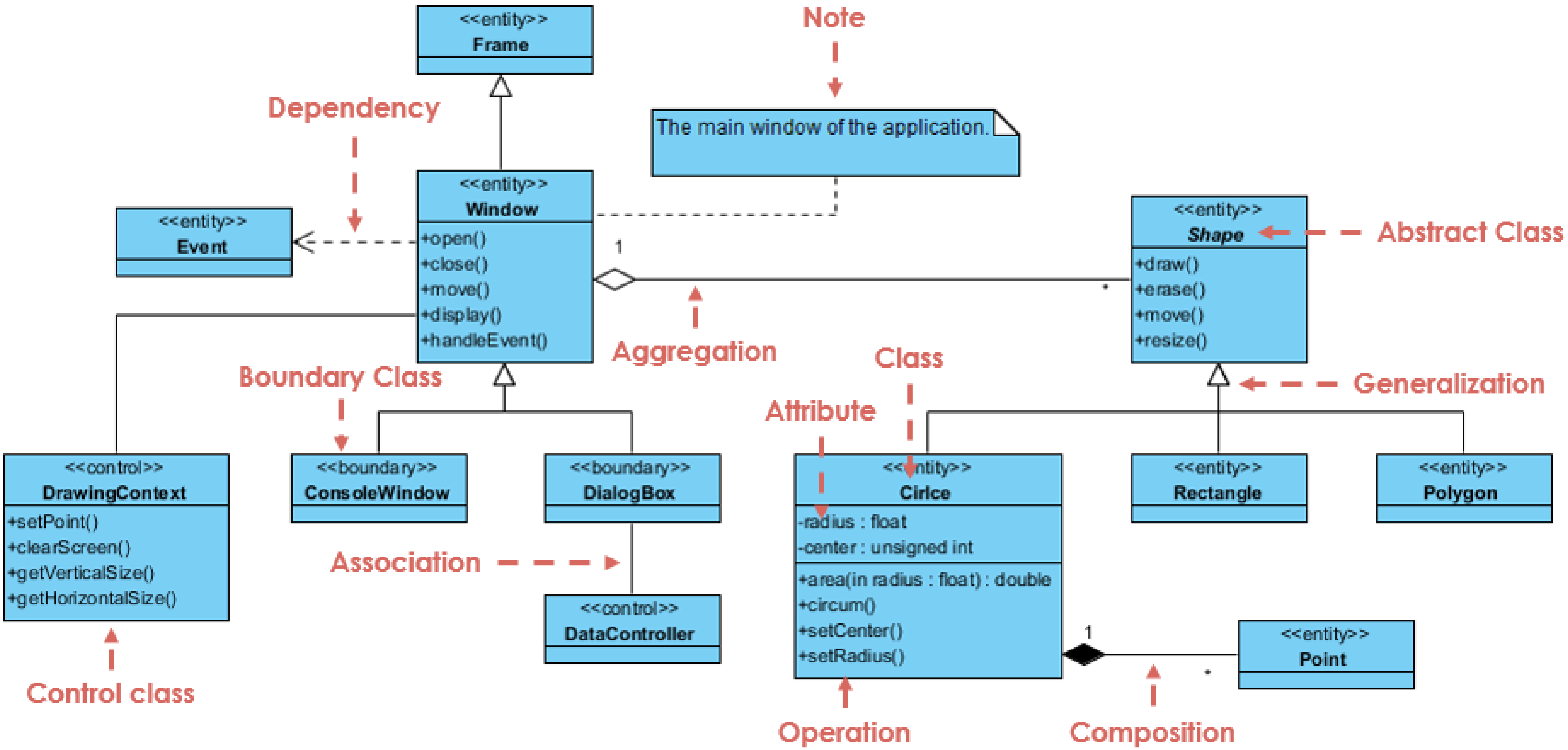


Aggregation vs. Composition

Aggregation	Composition
Aggregation indicates a relationship where the **child can exist separately from their parent class**. Example: Automobile (Parent) and Car (Child). So, If you delete the Automobile, the child Car still exist.	Composition display relationship where the **child will never exist independent of the parent**. Example: House (parent) and Room (child). Rooms will never separate into a House.


Customer
-name: String
-address
1
0..*
Order
-date: Date
-status: String
+calcSubTotal()
+calcTax()
+calcTotal()
+calcTotalWeight()
1
Line item
1..*
OrderDetail
-quality
-taxStatus: String
+calcSubTotal()
+calcWeight()
+calcTax()
0..*
1
Item
-shippingWeight
-description: String
+getPriceForQuantity()
+getTax()
+inStock()
1
1..*
Payment
-amount: float
Cash
-cashTendered: float
Check
-name: String
-bankID: String
+authorized()
Credit
-name: String
-type: String
-expDate
+authorized()

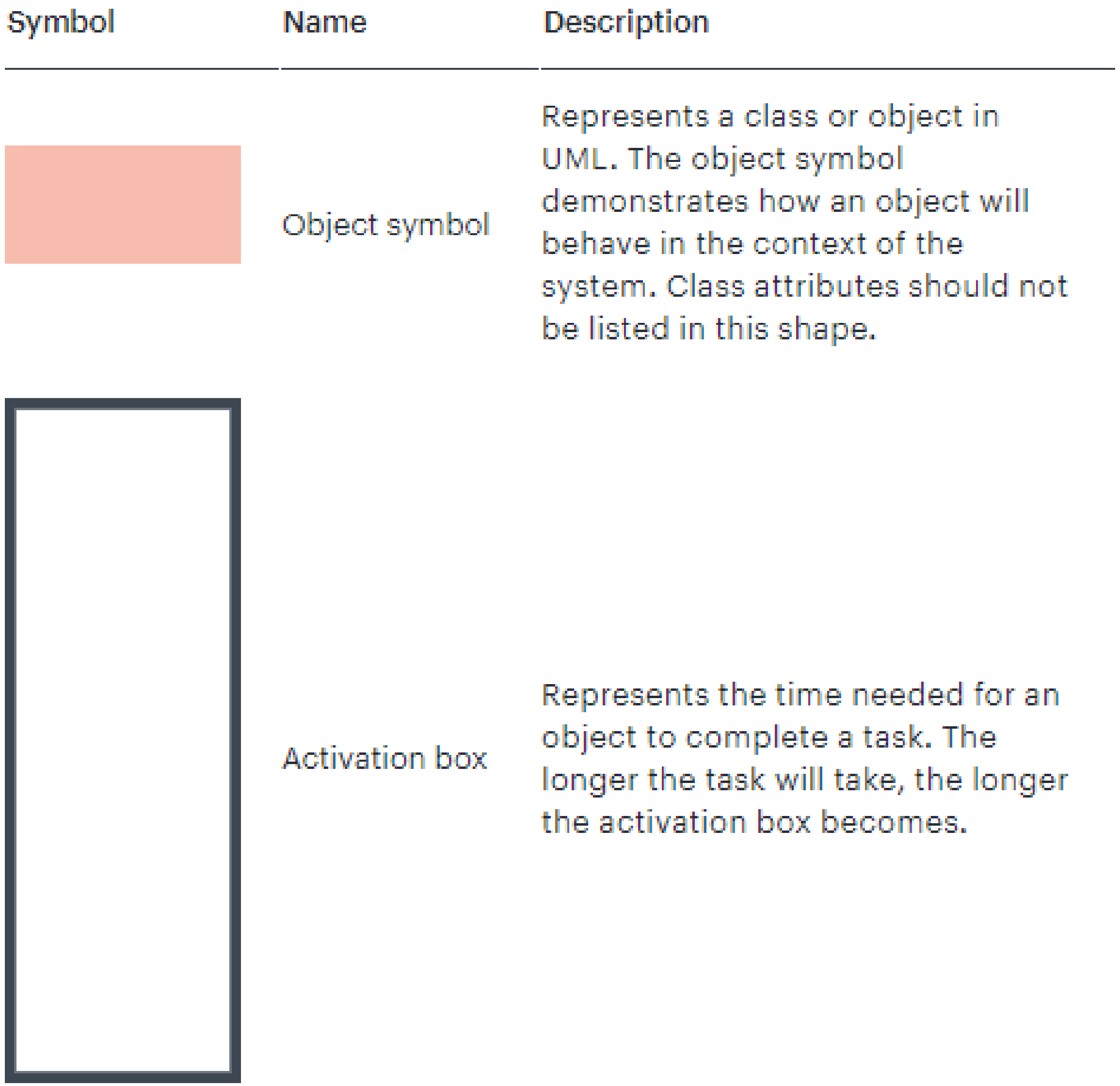

Bank
+code
+address
+manages()
+maintains()
ATM
+location
+managedby
+identifies()
+transactions()
Customer
+name
+address
+dob
+card number
+pin
+verifyPassword()
Has
1
1,2
Account
+number
+balance
+deposit()
+withdraw()
createTransaction()
*
1
Account Transaction
ATM Transactions
+transaction id
+date
+type
+amount
+post balance
+modifies()
Current Account
+account no.
+balance
+withdraw()
1
Savings-Checking
1
Saving Account
+account no.
+balance


Note
The main window of the application.
Dependency
<<entity>>
Frame
<<entity>>
Window
+open()
+close()
+move()
+display()
+handleEvent()
<<entity>>
Event
1
*
<<entity>>
Shape
+draw()
+erase()
+move()
+resize()
Abstract Class
Aggregation
Class
Generalization
Boundary Class
Attribute
<<control>>
DrawingContext
+setPoint()
+clearScreen()
+getVerticalSize()
+getHorizontalSize()
<<boundary>>
ConsoleWindow
<<boundary>>
DialogBox
<<entity>>
Cirlce
-radius : float
-center : unsigned int
+area(in radius : float) : double
+circum()
+setCenter()
+setRadius()
<<entity>>
Rectangle
<<entity>>
Polygon
Association
<<control>>
DataController
1
*
<<entity>>
Point
Control class
Operation
Composition

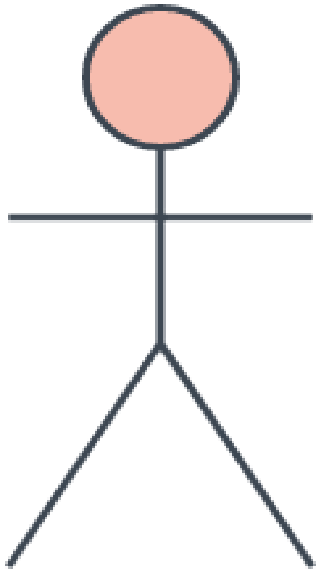
CASES

- Class Diagram for ATM
- Class Diagram for Hotel Management System
- Class Diagram for Library Management System
- Class Diagram for Online Shopping
- Class Diagram for Hospital Management System
- Class Diagram for Student Registration System
- Class Diagram for Taxi Reservation System

Basic symbols and components

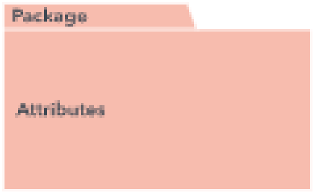


Symbol	Name	Description
🟥	Object symbol	Represents a class or object in UML. The object symbol demonstrates how an object will behave in the context of the system. Class attributes should not be listed in this shape.
▯	Activation box	Represents the time needed for an object to complete a task. The longer the task will take, the longer the activation box becomes.



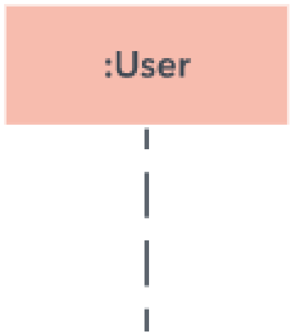
Actor symbol

Shows entities that interact with or are external to the system.



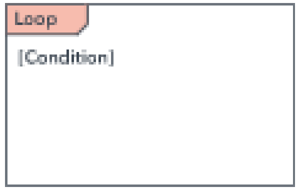
Package symbol

Used in UML 2.0 notation to contain interactive elements of the diagram. Also known as a frame, this rectangular shape has a small inner rectangle for labeling the diagram.



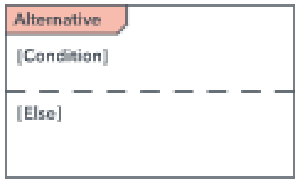
Lifeline symbol

Represents the passage of time as it extends downward. This dashed vertical line shows the sequential events that occur to an object during the charted process. Lifelines may begin with a labeled rectangle shape or an actor symbol.



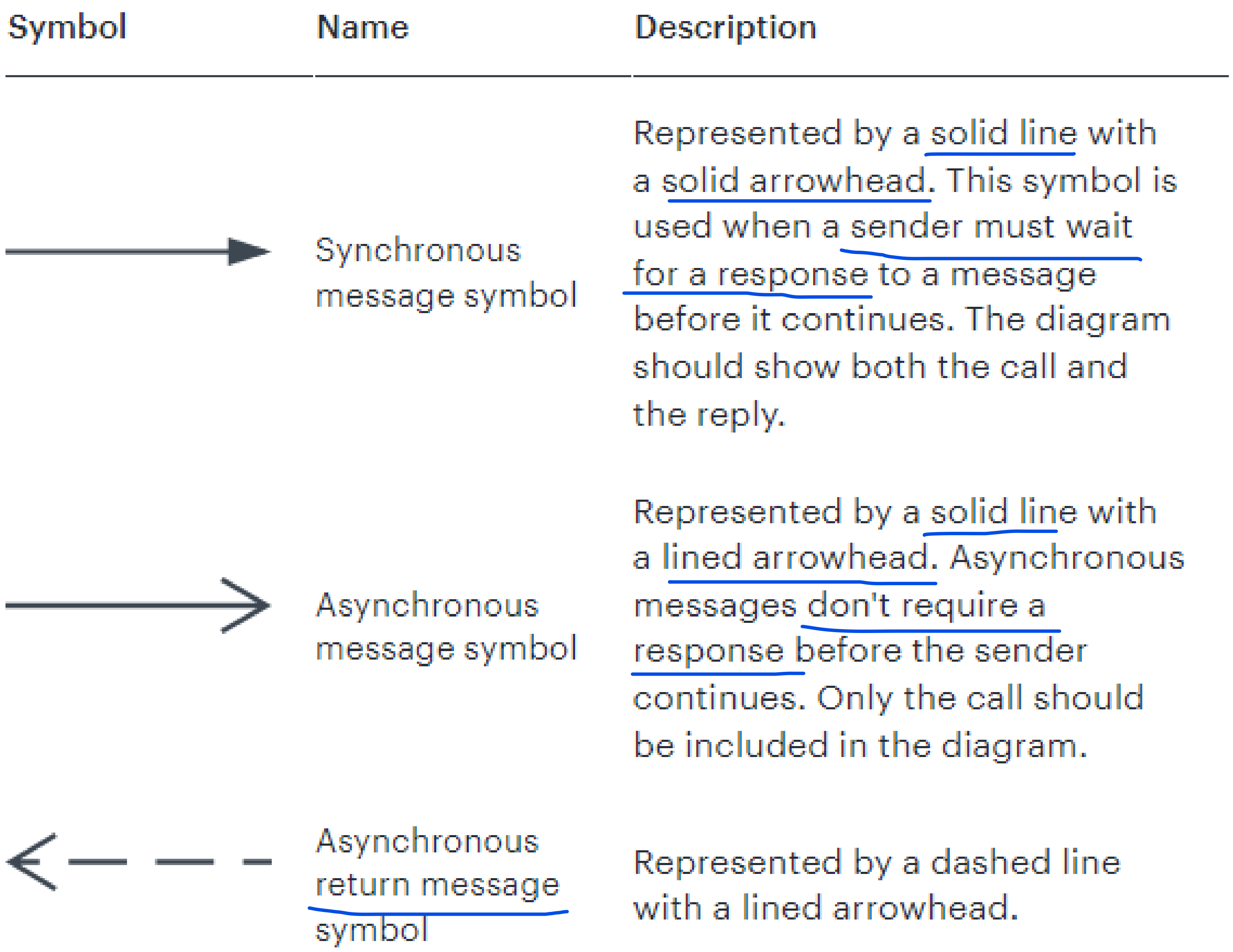
Option loop symbol

Used to model if/then scenarios, i.e., a circumstance that will only occur under certain conditions.

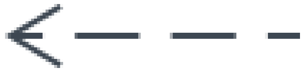
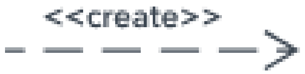
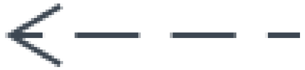
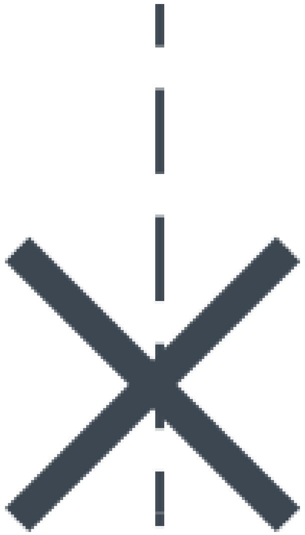


Alternative symbol

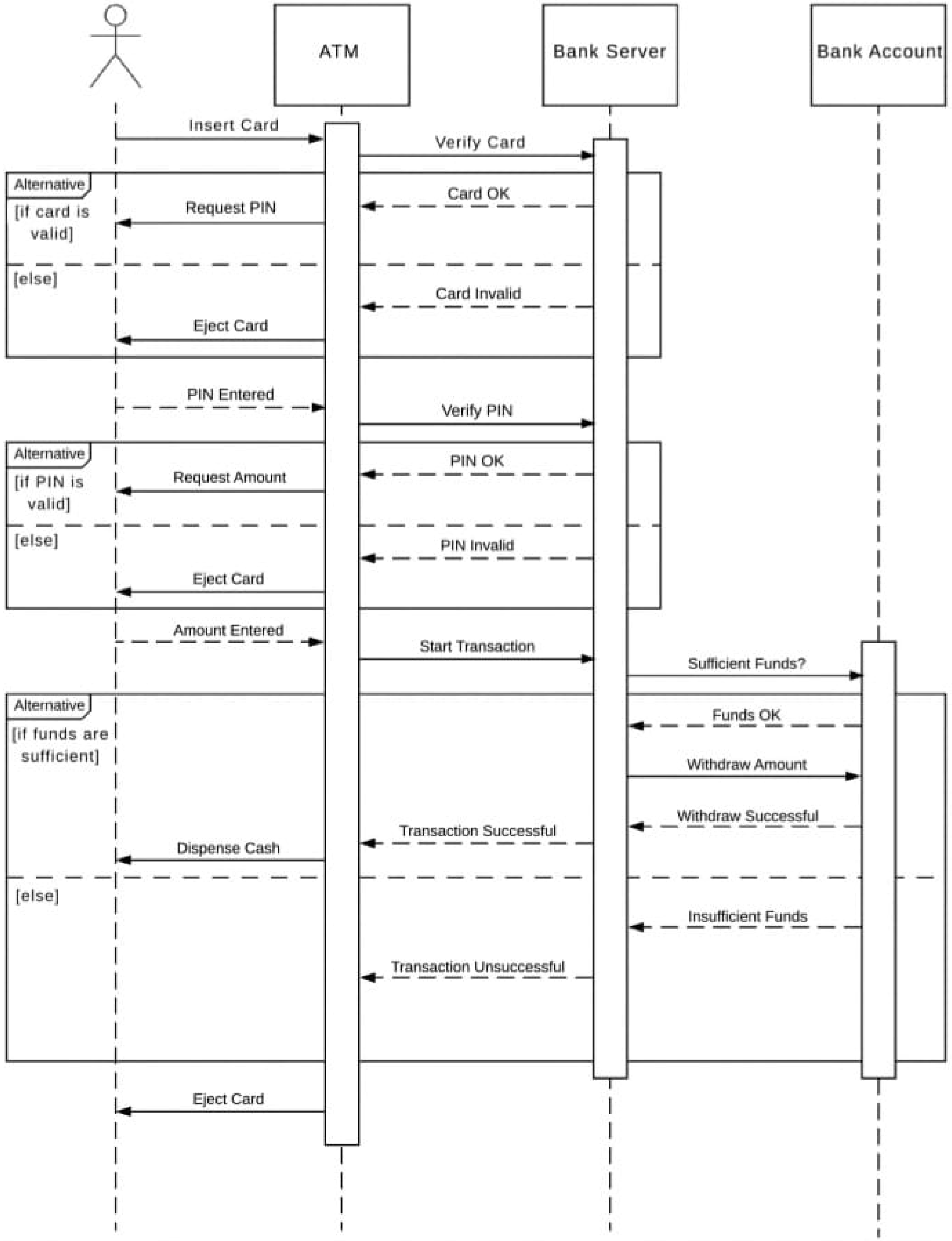
Symbolizes a choice (that is usually mutually exclusive) between two or more message sequences. To represent alternatives, use the labeled rectangle shape with a dashed line inside.



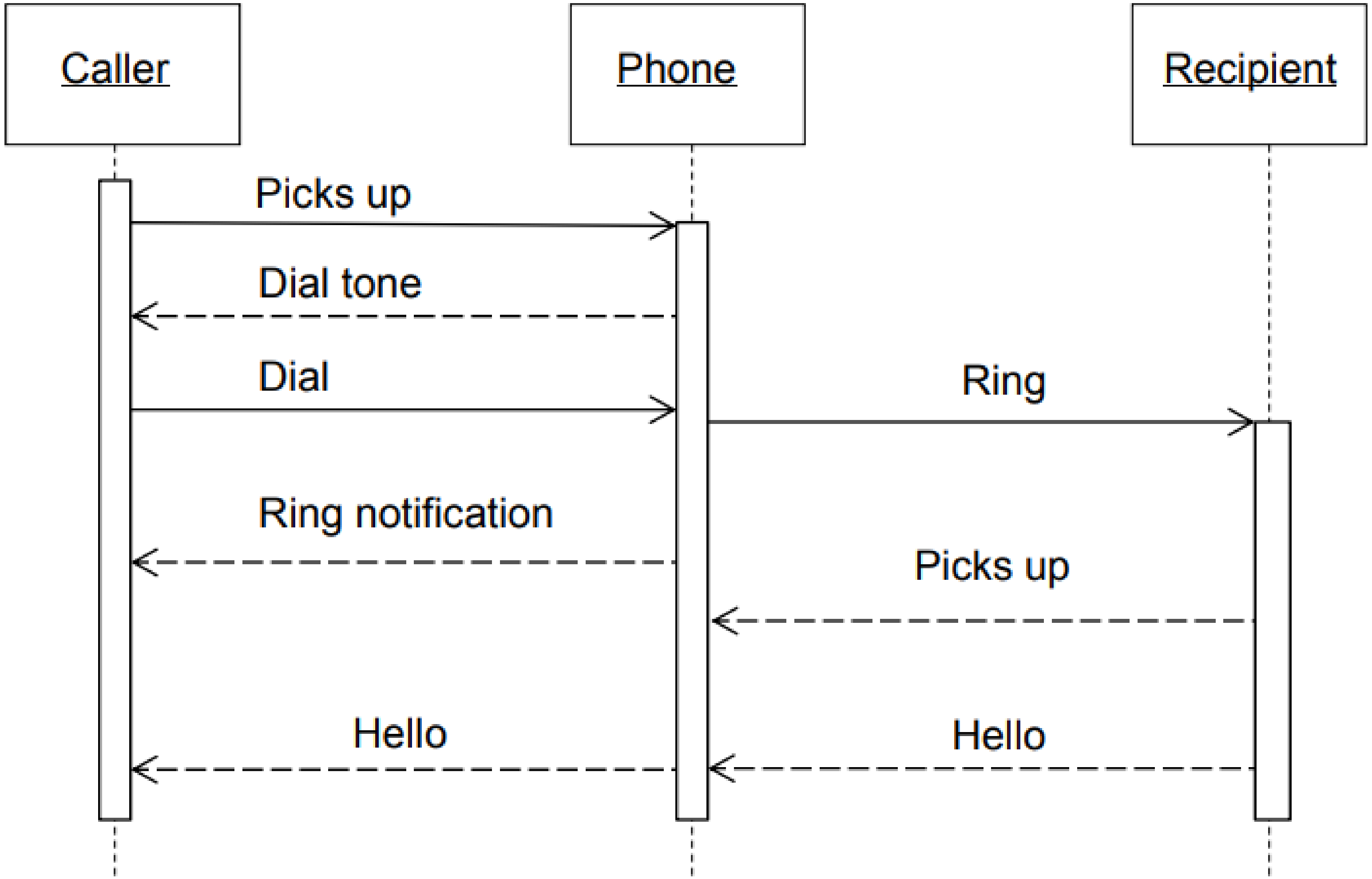
Symbol	Name	Description
───▶	Synchronous message symbol	Represented by a solid line with a solid arrowhead. This symbol is used when a sender must wait for a response to a message before it continues. The diagram should show both the call and the reply.
───>	Asynchronous message symbol	Represented by a solid line with a lined arrowhead. Asynchronous messages don't require a response before the sender continues. Only the call should be included in the diagram.
<– – –	Asynchronous return message symbol	Represented by a dashed line with a lined arrowhead.

	Asynchronous return message symbol	Represented by a dashed line with a lined arrowhead.
	Asynchronous create message symbol	Represented by a dashed line with a lined arrowhead. This message creates a new object.
	Reply message symbol	Represented by a dashed line with a lined arrowhead, these messages are replies to calls.
	Delete message symbol	Represented by a solid line with a solid arrowhead, followed by an X. This message destroys an object.

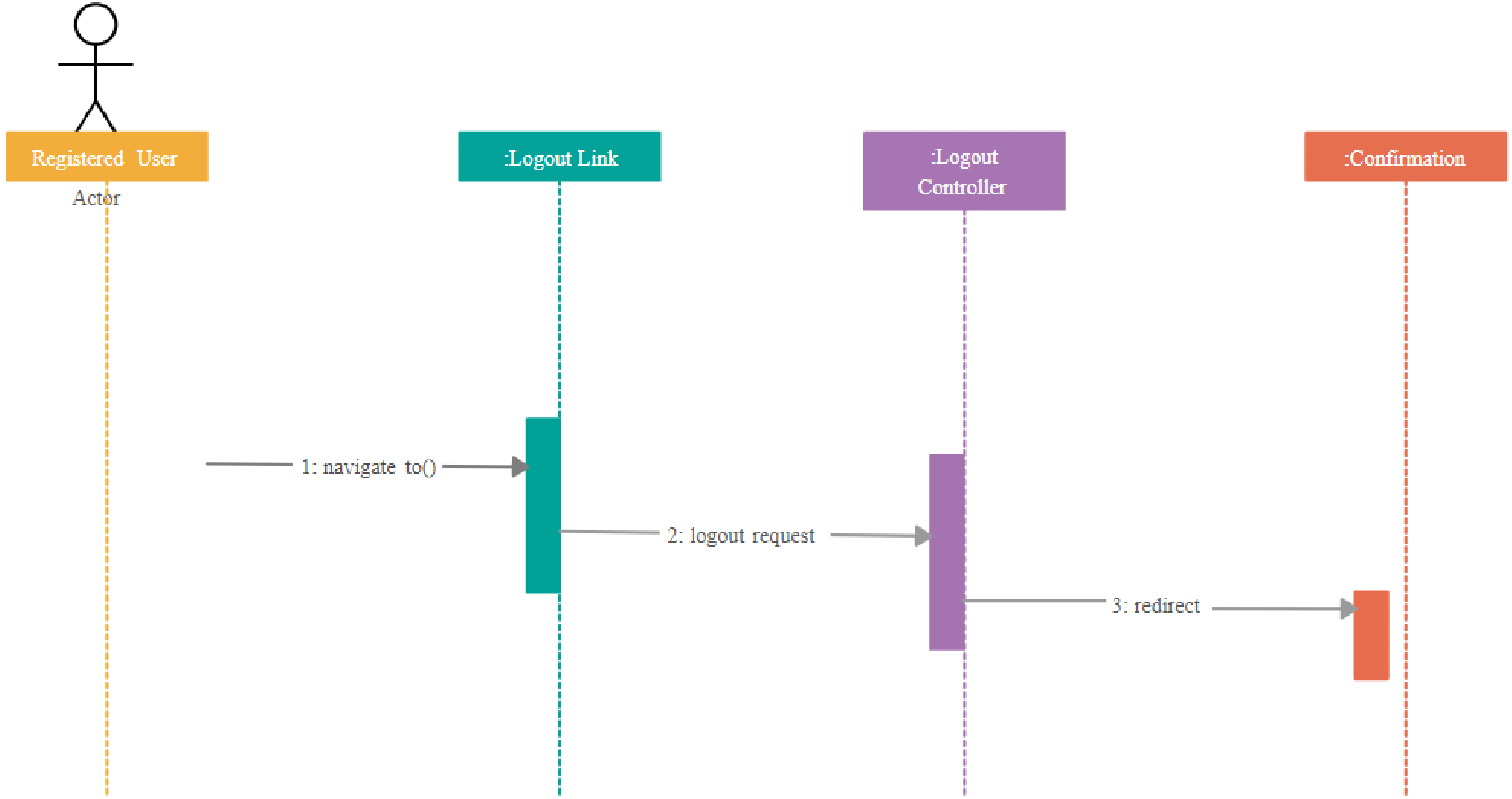
Sequence Diagram (ATM)



Sequence Diagram (make a phone call)



Logout Sequence Diagram



Study

https://github.com/wyaadarsh/Grokking-OOD/blob/master/object-oriented-design-case-studies/design-a-movie-ticket-booking-system.md

End of Topic 3