Object-Oriented Analysis and Design with UML

Agenda

- https://www.uml-diagrams.org/uml-25-diagrams.html
- UML Introduction
- UML Diagrams
- UML Implementation Example
- UML Conventions
- Review

What is UML?

- UML (Unified Modelling Language)
  - An emerging standard for modelling object-oriented software.
  - Resulted from the convergence of notations from three leading object-oriented methods:
    - OMT (James Rumbaugh)
    - OOSE (Ivar Jacobson)
    - Booch (Grady Booch)
- Reference: “design patterns by GOF”
- Supported by many CASE tools
  - Rational Rose, Visual Paradigm, etc

What is modelling?

- Modelling consists of building an abstraction of reality.
- Abstractions are simplifications because:
  - They ignore irrelevant details and
  - They only represent the relevant details.
- What is *relevant* or *irrelevant* depends on the purpose of the model.

Why model software?

Why model software?

- Software is getting increasingly more complex
  - Windows XP > 40 million lines of code
  - A single programmer cannot manage this amount of code in its entirety.
- Code is not easily understandable by developers who did not write it
- We need simpler representations for complex systems
  - Modelling is a means for dealing with complexity

Systems, Models and Views

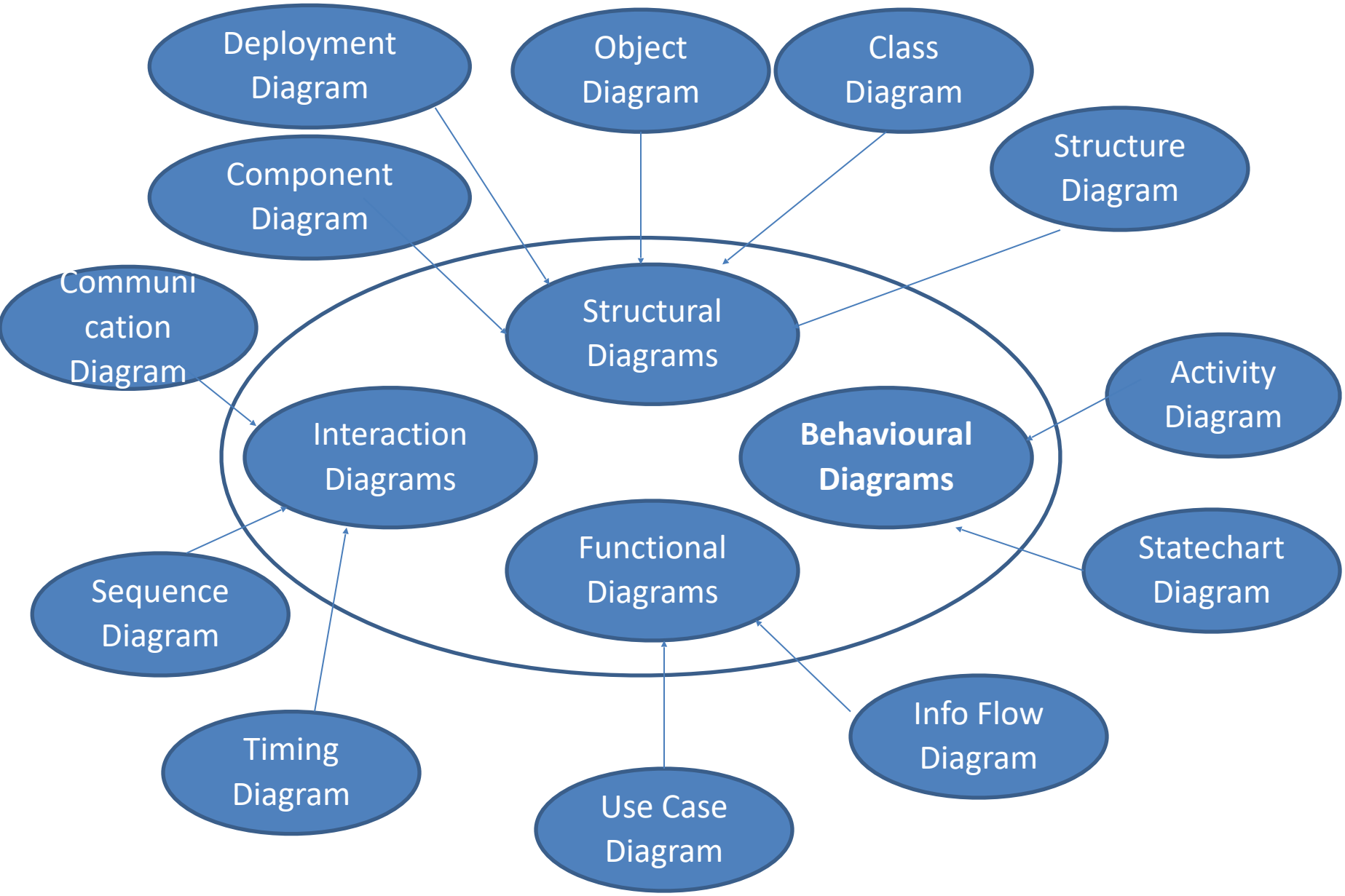
- A ***model*** is an abstraction describing a subset of a system
- A ***view*** depicts selected aspects of a model
- A ***notation*** is a set of graphical or textual rules for depicting views
- Views and models of a single system may overlap each other

Examples:

- System: Aircraft
- Models: Flight simulator, scale model
- Views: All blueprints, electrical wiring, fuel system

Building Blocks

- Things : Structural, Behavioural, Grouping, Annotations
- Relationships
- Diagrams


Deployment Diagram
Object Diagram
Class Diagram
Component Diagram
Structure Diagram
Communi cation Diagram
Structural Diagrams
Activity Diagram
Interaction Diagrams
Behavioural Diagrams
Functional Diagrams
Statechart Diagram
Sequence Diagram
Info Flow Diagram
Timing Diagram
Use Case Diagram

UML

UML

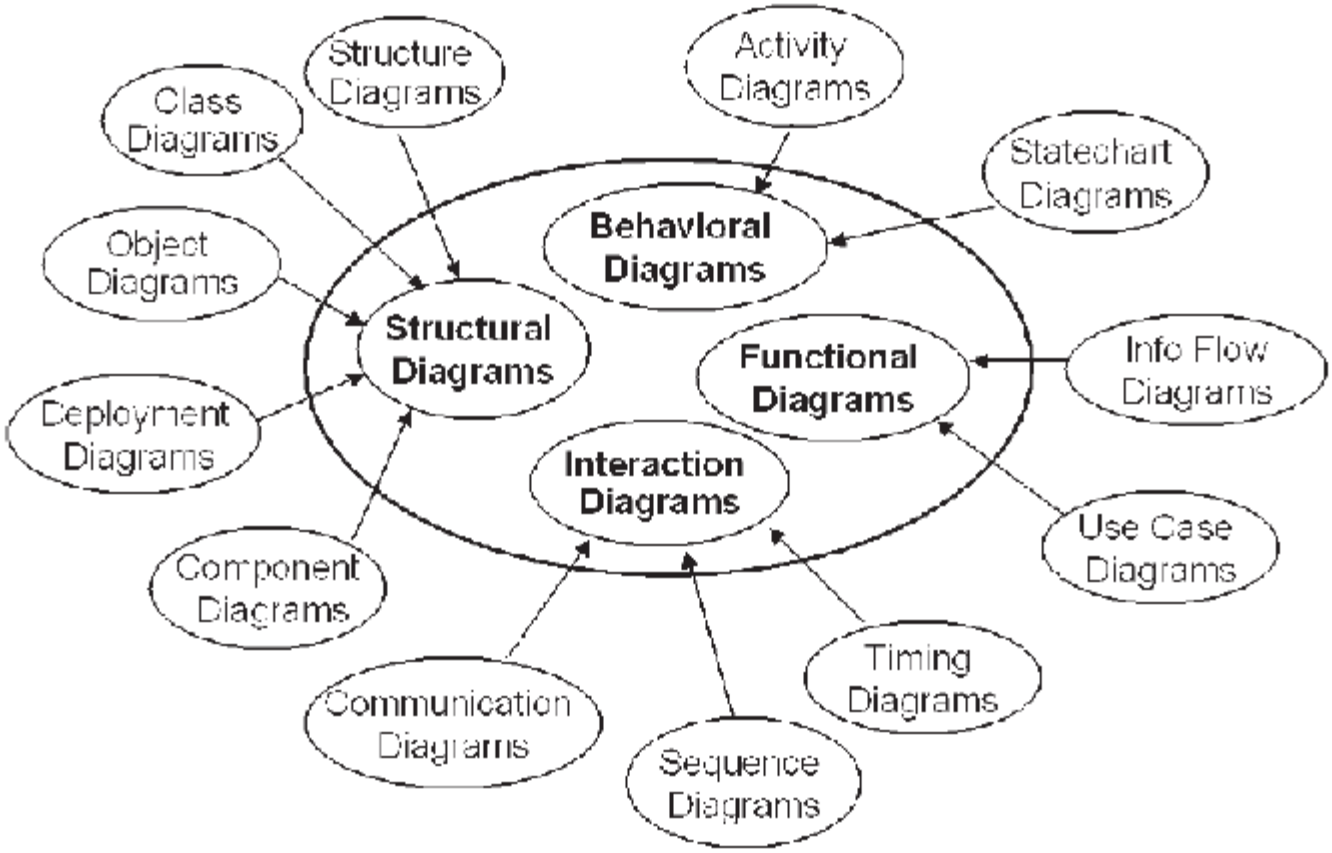
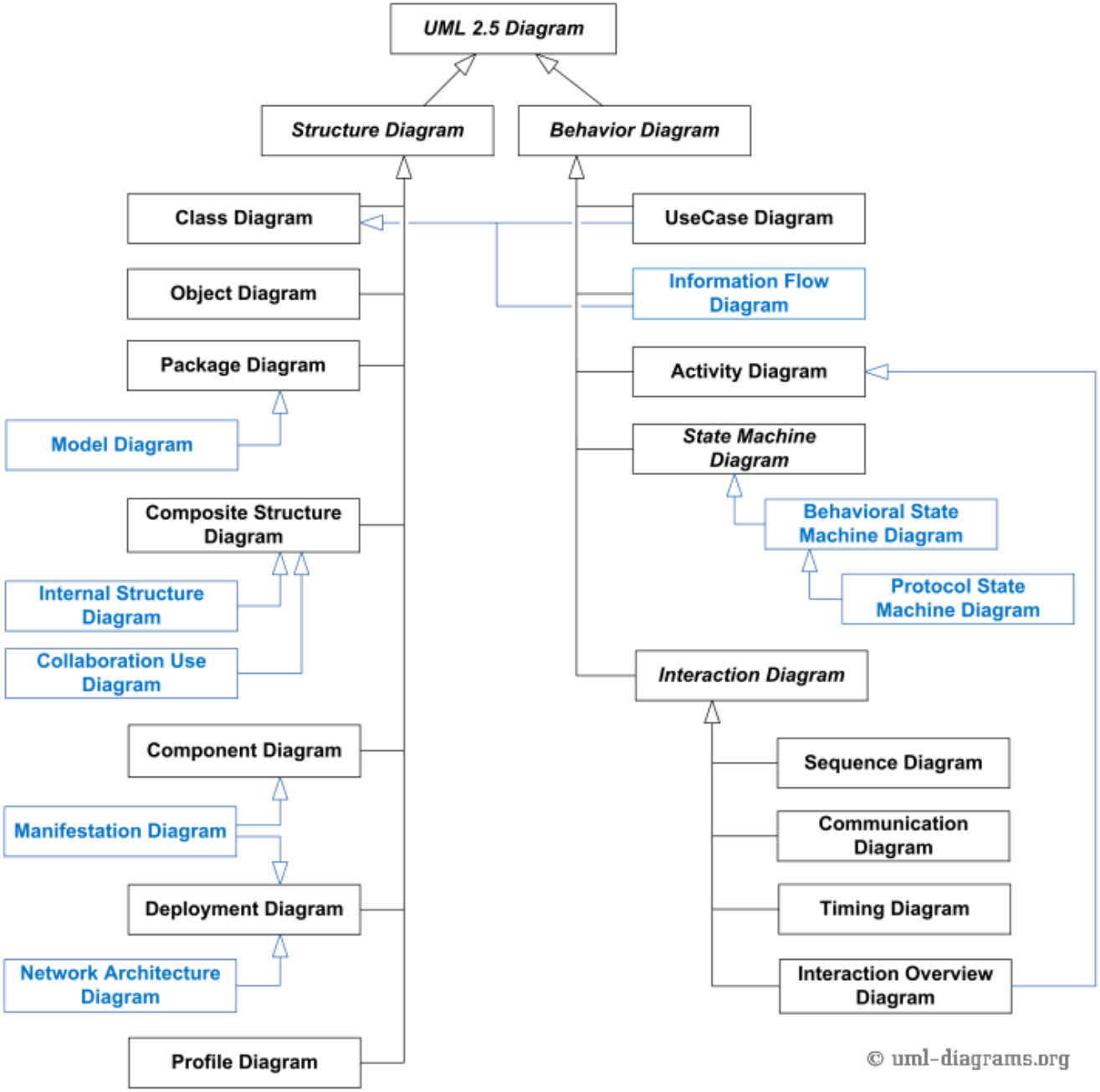


Figure 1.1 UML diagram types

UML

UML

UML

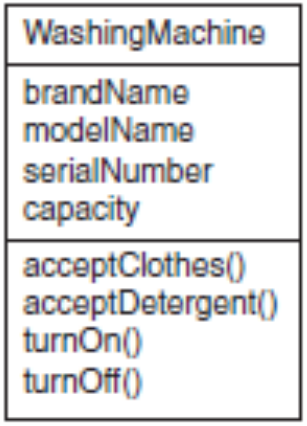

UML 2.5 Diagram
Structure Diagram
Behavior Diagram
Class Diagram
Object Diagram
Package Diagram
Model Diagram
Composite Structure Diagram
Internal Structure Diagram
Collaboration Use Diagram
Component Diagram
Manifestation Diagram
Deployment Diagram
Network Architecture Diagram
Profile Diagram
UseCase Diagram
Information Flow Diagram
Activity Diagram
State Machine Diagram
Behavioral State Machine Diagram
Protocol State Machine Diagram
Interaction Diagram
Sequence Diagram
Communication Diagram
Timing Diagram
Interaction Overview Diagram
© uml-diagrams.org

Diagrams in UML

- Class diagram
- Object diagram
- Component diagram
- Composite structure diagram
- Use case diagram
- Sequence diagram
- Communication diagram
- State diagram
- Activity diagram
- Deployment diagram
- Package diagram
- Timing diagram
- Interaction overview diagram

Class Diagram

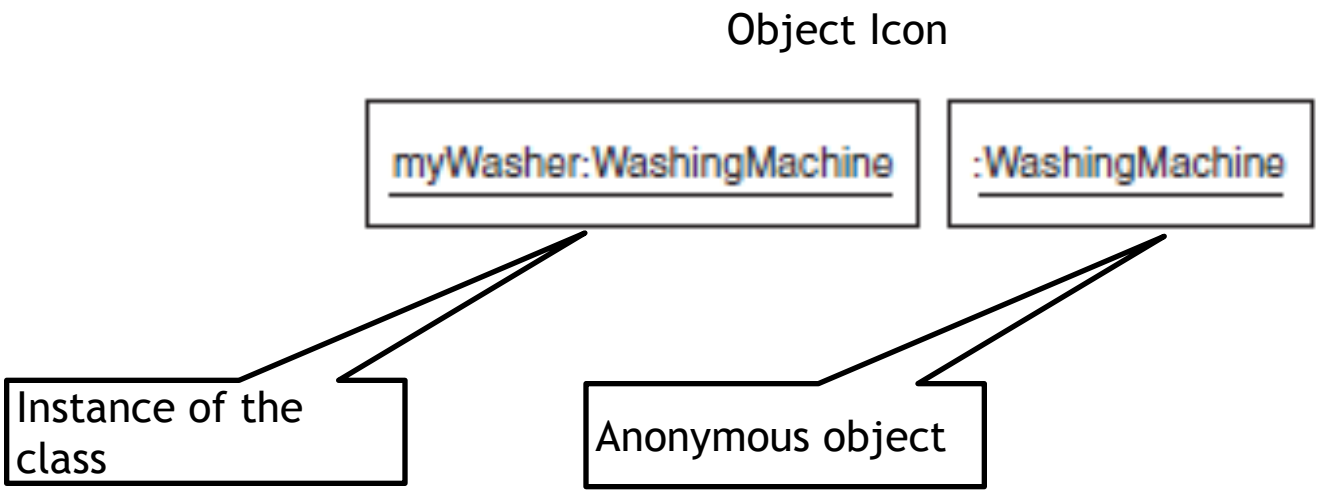
- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- Class diagrams address the static design view of a system.

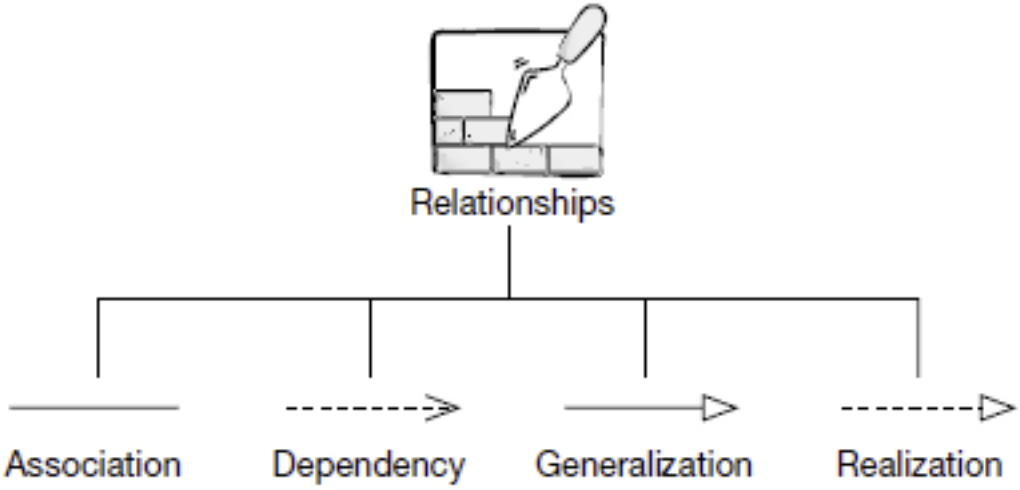


Class Icon

Object Diagram

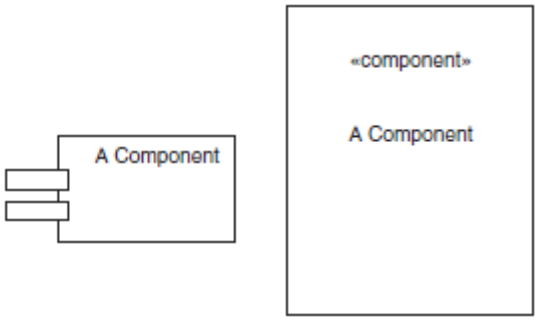
- An object diagram shows a set of objects and their relationships.
- Object diagrams represent static snapshots of instances of the things found in class diagrams.




Relationships
Association
Dependency
Generalization
Realization

Component Diagram

- A component diagram is shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors.
- Component diagrams address the static design implementation view of a system. They are important for building large systems from smaller parts. .



Component Icon

Class

Class: **Bank**

Abstract class: *Italics*

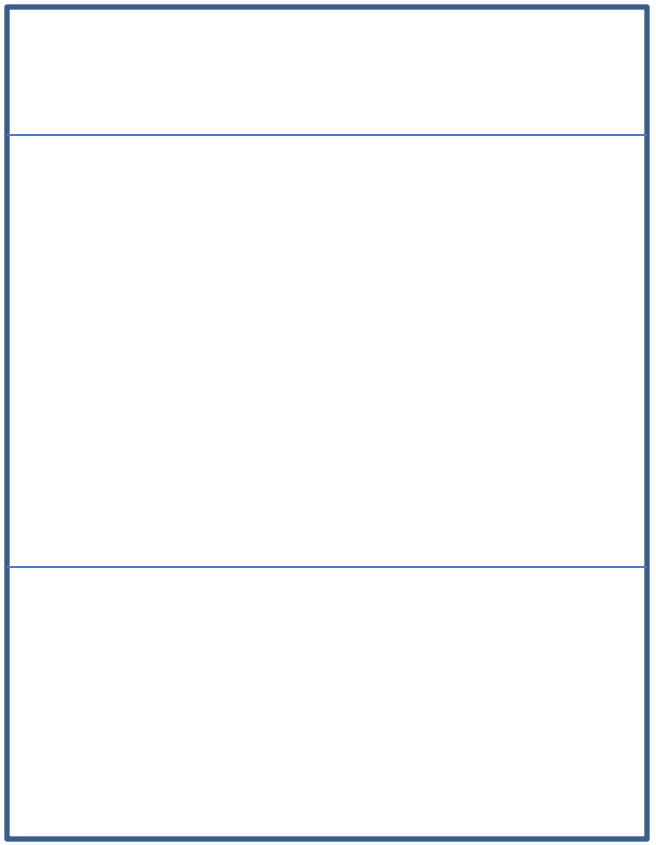
Interface: <<InterfaceName>>

+ public

- private

protected

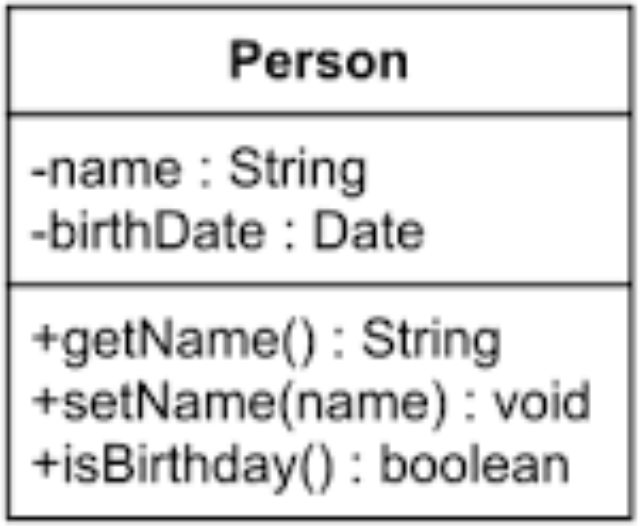
~ default

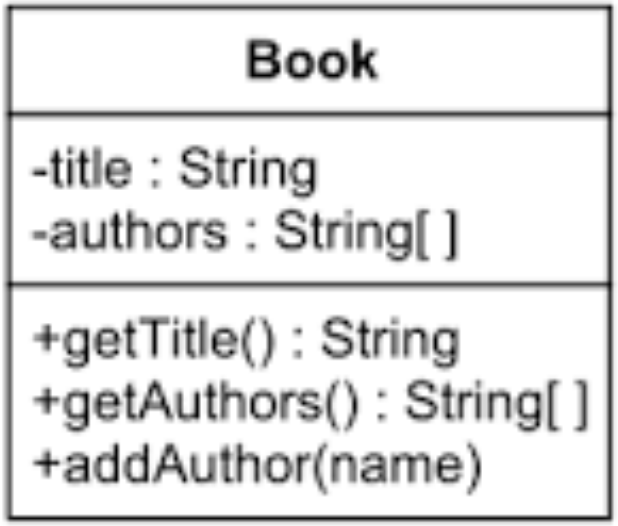


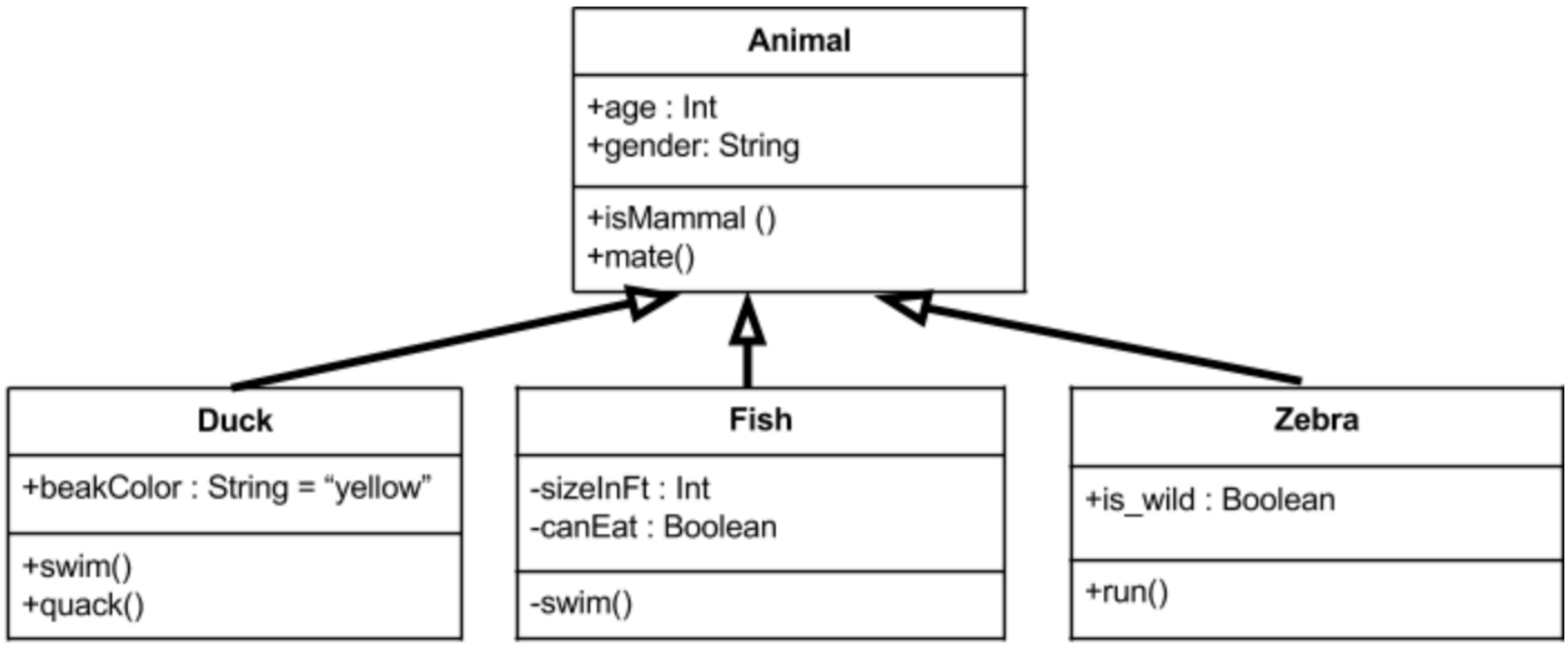
Class

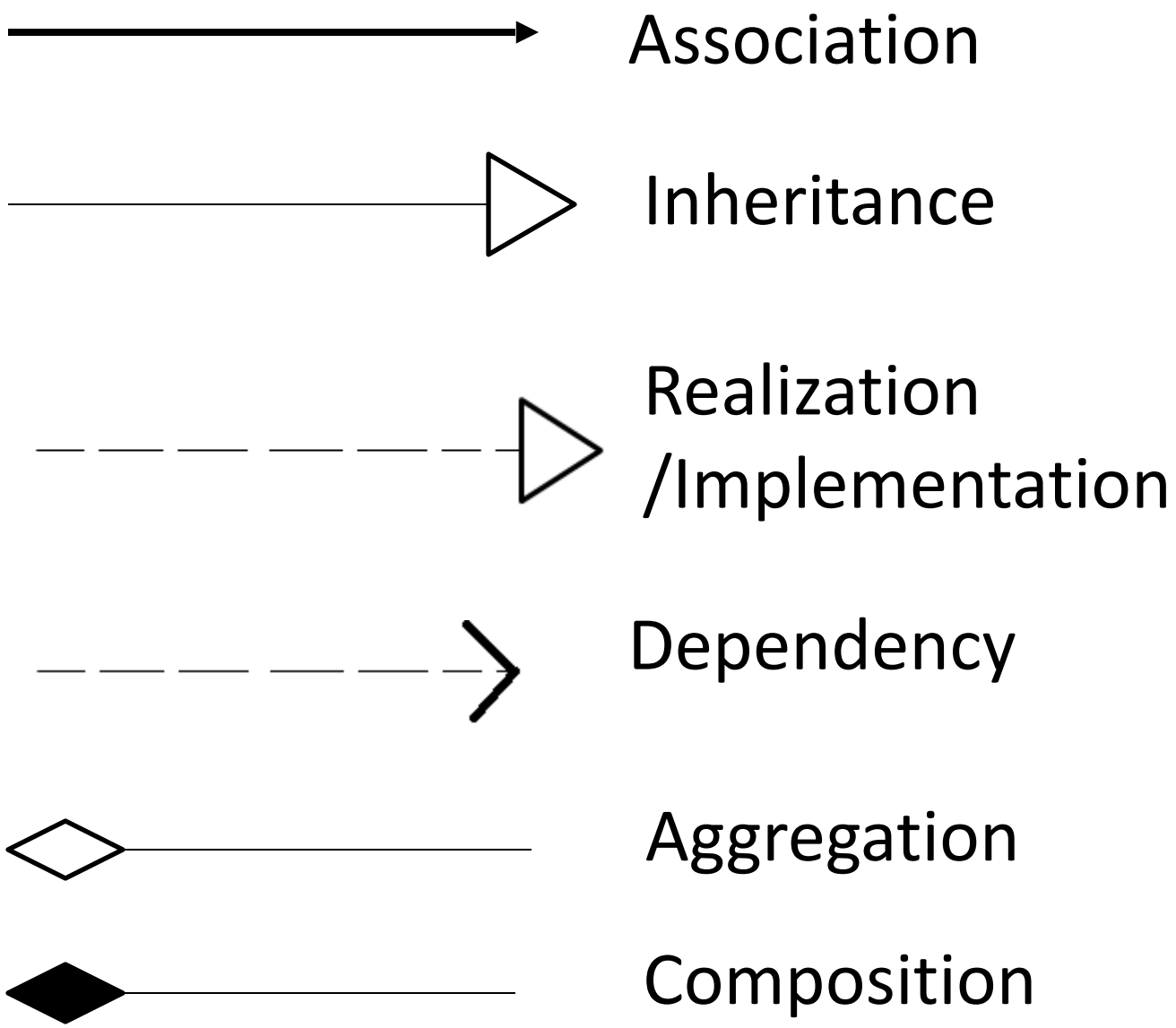
```
class Bank{

   private :

        int cash ;

  public  :

        float getCash() ;
       void  setCash( float tmp_cash) ;

 };
```

Data Encapsulation






Animal
+age : Int
+gender: String
+isMammal ()
+mate()
Duck
+beakColor : String = “yellow”
+swim()
+quack()
Fish
-sizeInFt : Int
-canEat : Boolean
-swim()
Zebra
+is_wild : Boolean
+run()


Association
Inheritance
Realization
/Implementation
Dependency
Aggregation
Composition

Encapsulation

Encapsulation is accomplished when each object maintains a private state, inside a class. Other objects can not access this state directly, instead, they can only invoke a list of public functions. The object manages its own state via these functions and no other class can alter it unless explicitly allowed. In order to communicate with the object, you will need to utilize the methods provided. One way I like to think of encapsulation is by using the example of people and their dogs. If we want to apply encapsulation, we do so by encapsulating all “dog” logic into a Dog class. The “state” of the dog is in the private variables playful, hungry and energy and each of these variables has their respective fields.

There is also a private method: bark(). The dog class can call this whenever it wants, and the other classes can not tell the dog when to bark. There are also public methods such as sleep(), play() and eat() that are available to other classes. Each of these functions modifies the internal state of the Dog class and may invoke bark(), when this happens the private state and public methods are bonded.

Abstraction

Abstraction is an extension of encapsulation. It is the process of selecting data from a larger pool to show only the relevant details to the object. Suppose you want to create a dating application and you are asked to collect all the information about your users. You might receive the following data from your user: Full name, address, phone number, favorite food, favorite movie, hobbies, tax information, social security number, credit score. This amount of data is great however not all of it is required to create a dating profile. You only need to select the information that is pertinent to your dating application from that pool. Data like Full name, favorite food, favorite movie, and hobbies make sense for a dating application. The process of fetching/removing/selecting the user information from a larger pool is referred to as Abstraction. One of the advantages of Abstraction is being able to apply the same information you used for the dating application to other applications with little or no modification.

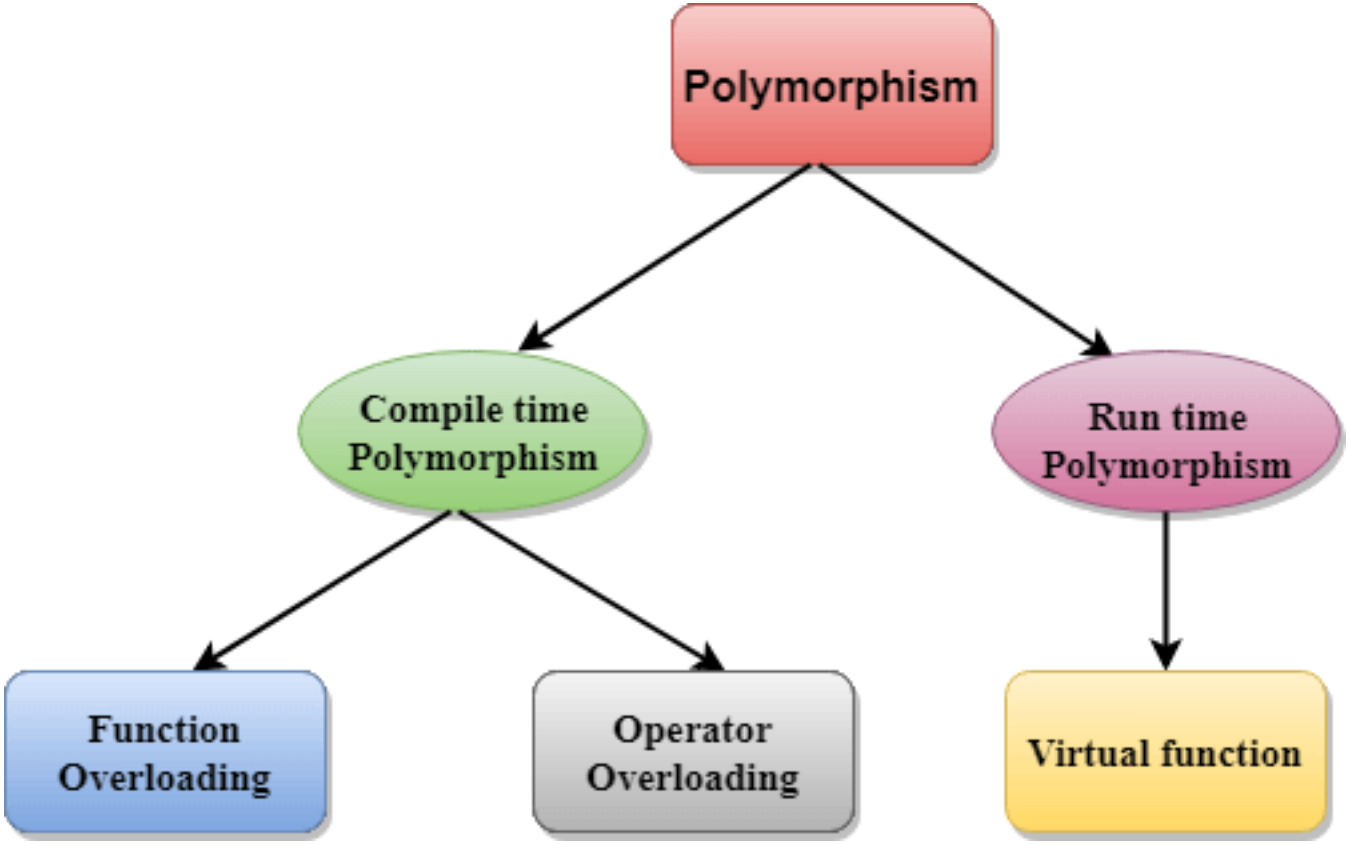
Inheritance

Inheritance is the ability of one object to acquire some/all properties of another object. For example, a child inherits the traits of his/her parents. With inheritance, reusability is a major advantage. You can reuse the fields and methods of the existing class. In Java, there are various types of inheritances: single, multiple, multilevel, hierarchical, and hybrid. For example, Apple is a fruit so assume that we have a class Fruit and a subclass of it named Apple. Our Apple acquires the properties of the Fruit class. Other classifications could be grape, pear, and mango, etc. Fruit defines a class of foods that are mature ovaries of a plant, fleshy, contains a large seed within or numerous tiny seeds. Apple the sub-class acquires these properties from Fruit and has some unique properties, which are different from other sub-classes of Fruit such as

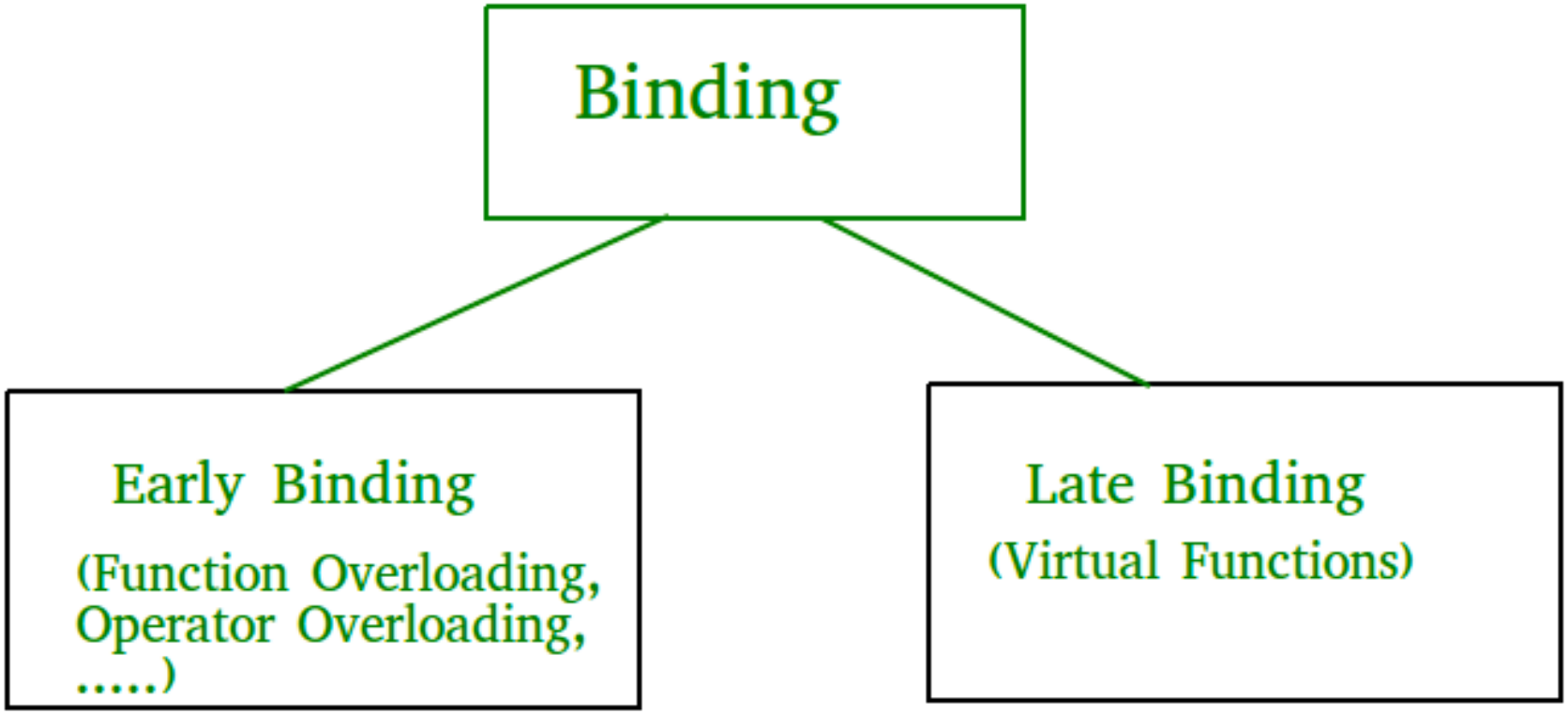
Polymorphism

Polymorphism gives us a way to use a class exactly like its parent so there is no confusion with mixing types. This being said, each child sub-class keeps its own functions/methods as they are. If we had a superclass called Mammal that has a method called mammalSound(). The sub-classes of Mammals could be Dogs, whales, elephants, and horses. Each of these would have their own iteration of a mammal sound (dog-barks, whale-clicks).

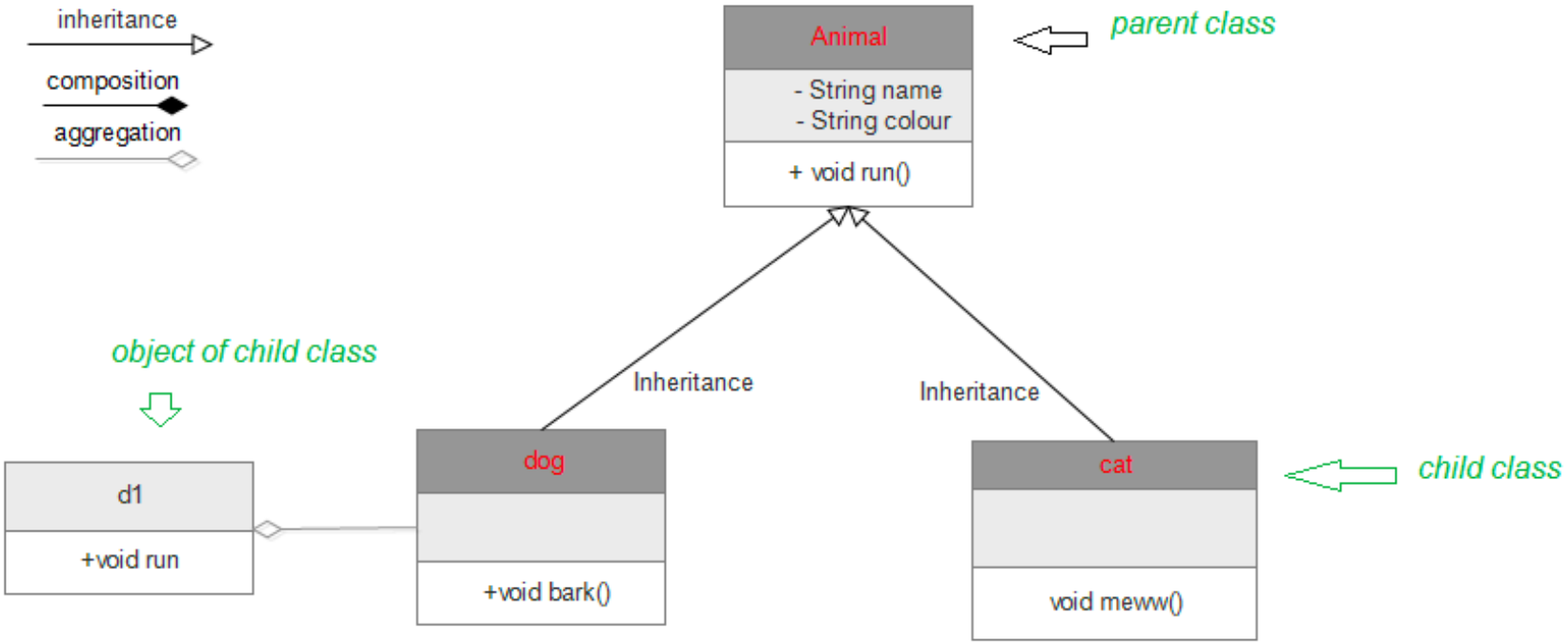
Polymorphism: Compile time ,Run time



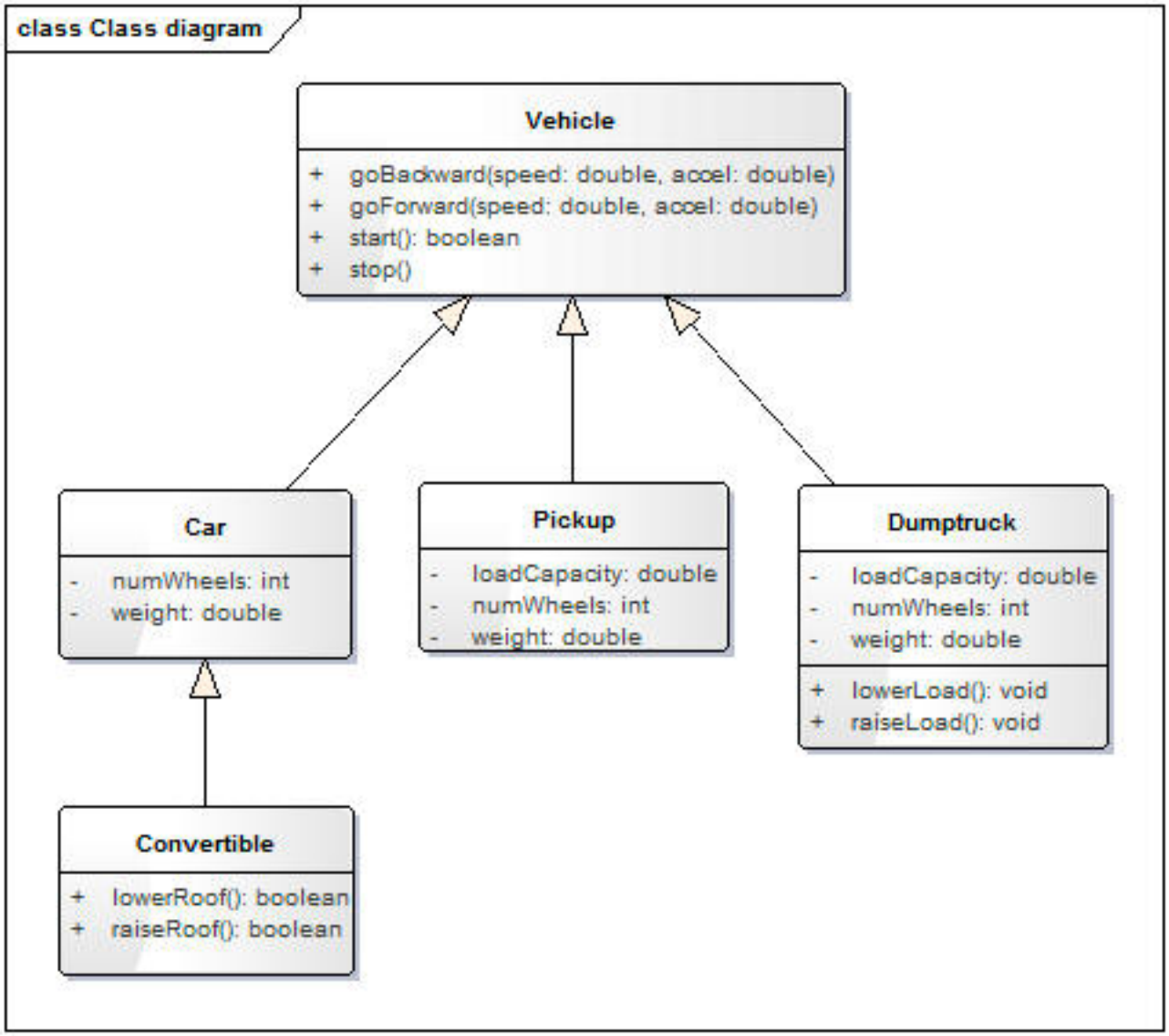
Polymorphism Early v/s late binding



Inheritance



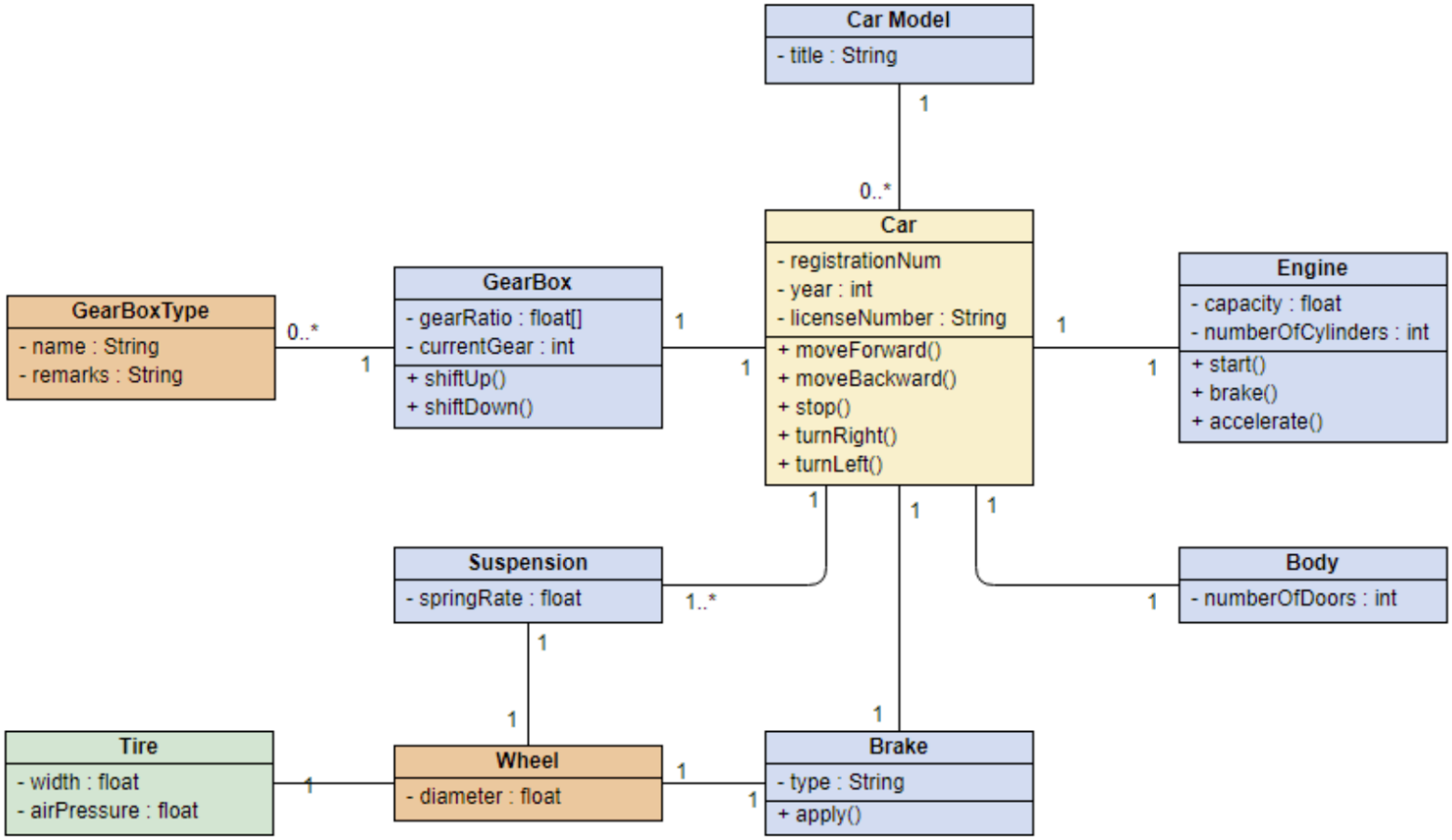
Inheritance



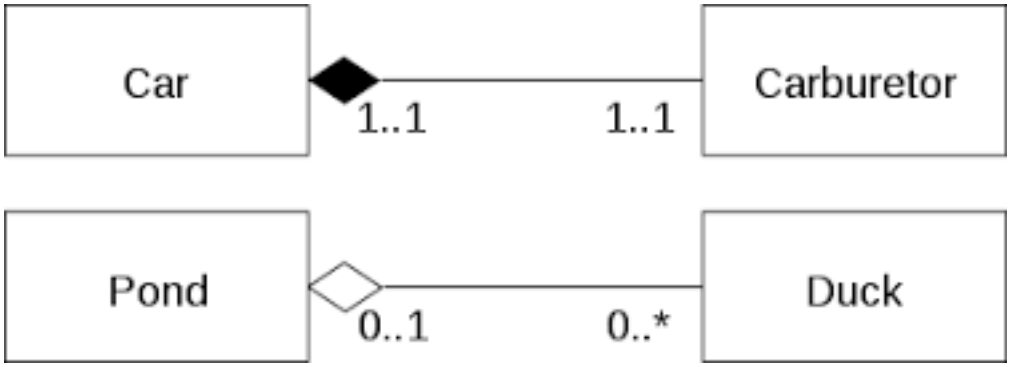
Class

```
class Bank{

   private :

        int cash ;

  public  :

        float getCash() ;
       void  setCash( float tmp_cash) ;

 };
```

Composed Car components

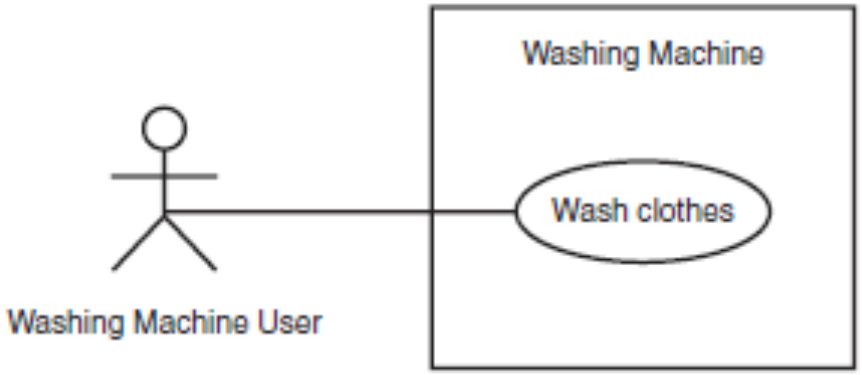


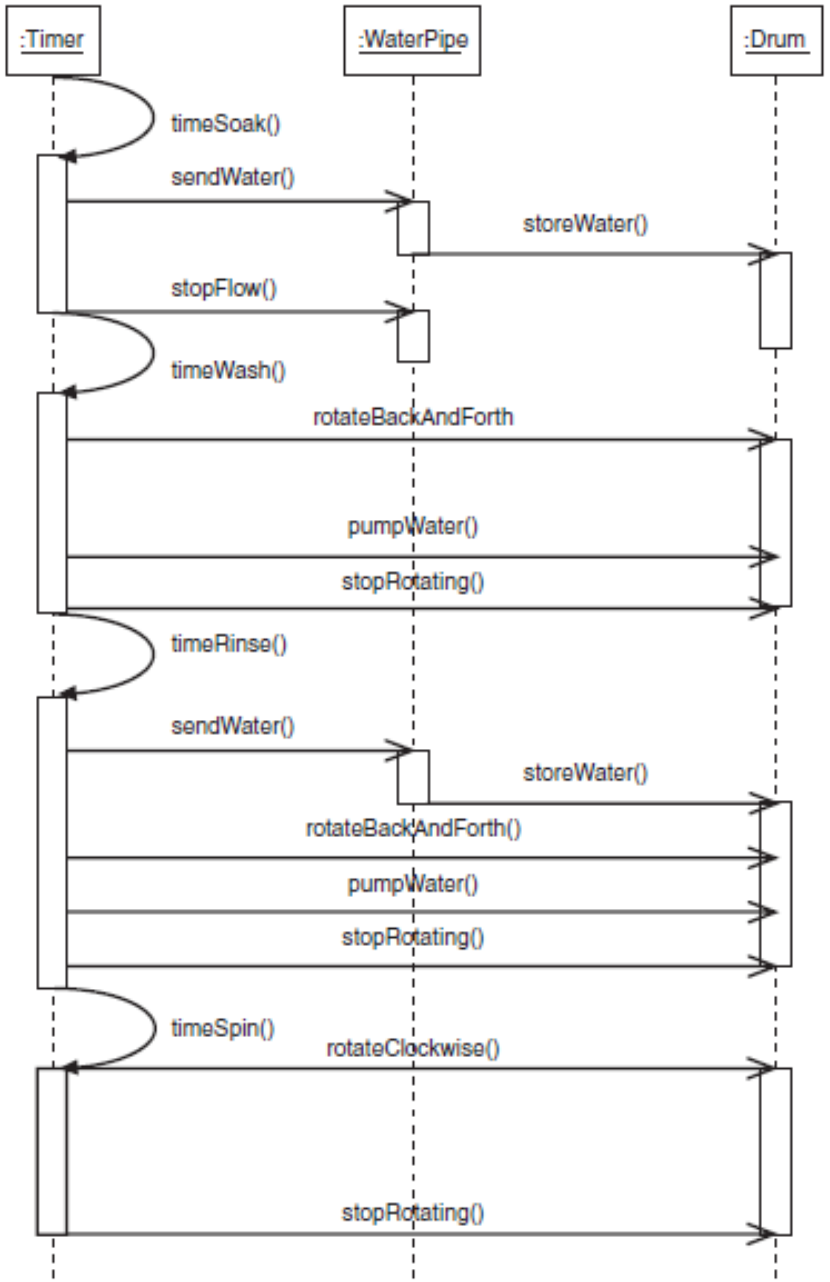
Composition vs Aggregation



Use-Case Diagram

- A use case diagram shows a set of use cases and actors (a special kind of class) and their relationships.
- Use case diagrams address the static use case view of a system. These diagrams are especially important in organising and modelling the behaviours of a system.



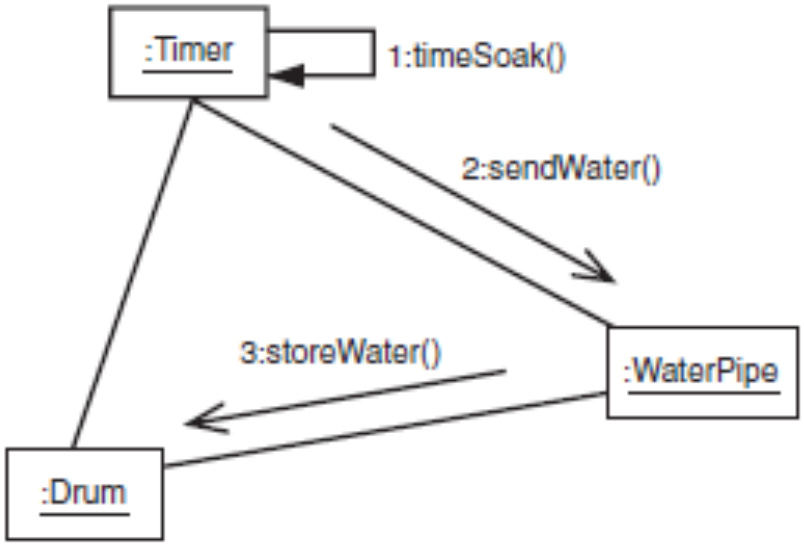


Sequence Diagram

A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages

Communication Diagram

- A communication diagram is an interaction diagram that emphasises the structural organisation of the objects or roles that send and receive messages

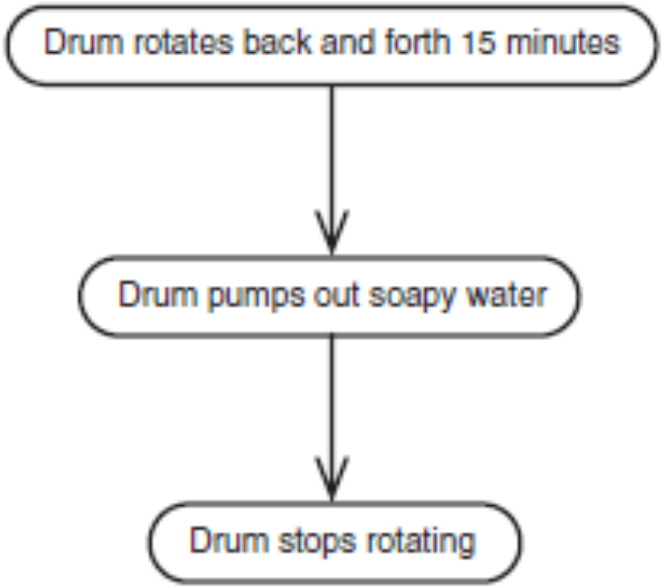


State Diagram

- A state diagram shows a state machine, consisting of states, transitions, events, and activities. A state diagram shows the dynamic view of an object.

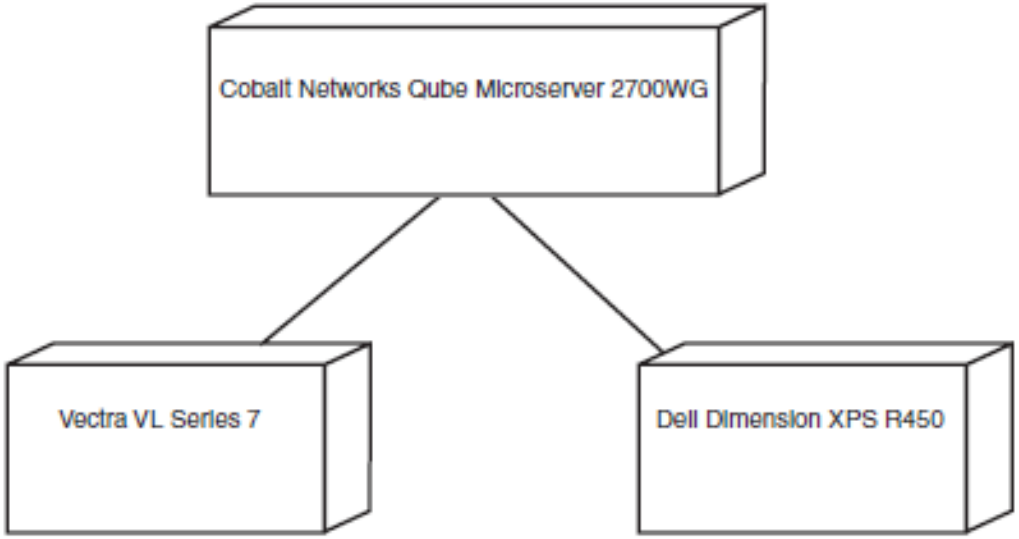
Activity Diagram

- An activity diagram shows the structure of a process or other computation as the flow of control and data from step to step within the computation. Activity diagrams address the dynamic view of a system.



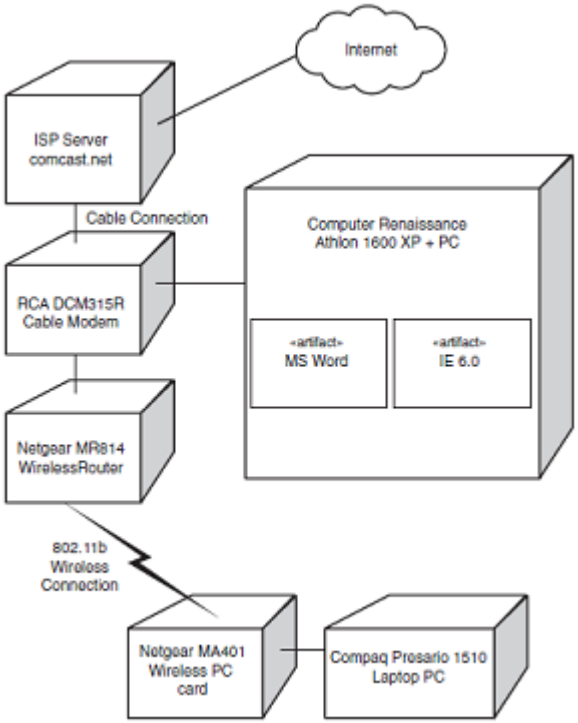
Deployment Diagram

- A deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of an architecture.



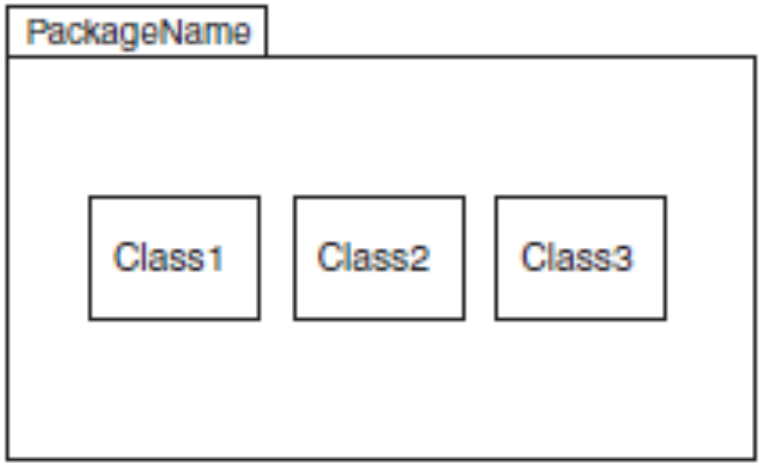
Artefacts Diagram

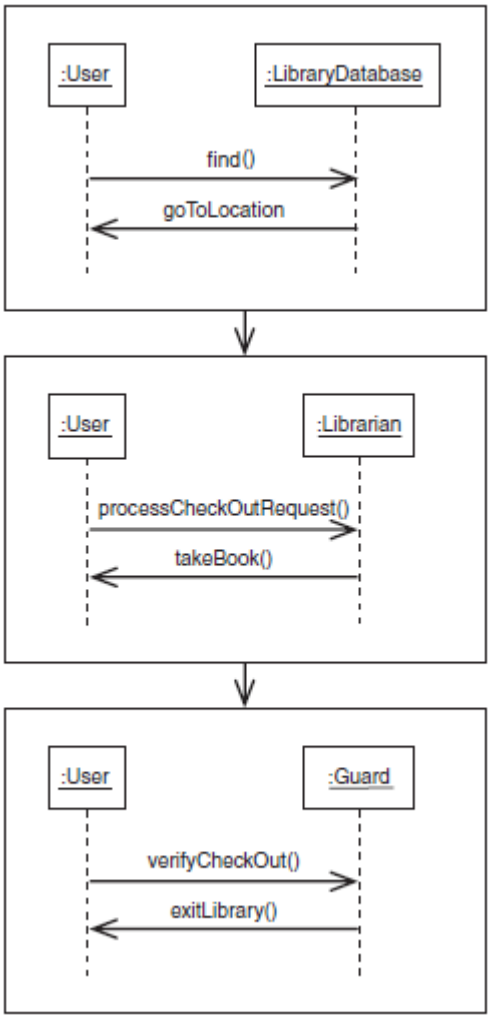
- An artefact diagram shows the physical constituents of a system on the computer. Artefacts include files, databases, and similar physical collections of bits. Artefacts are often used in conjunction with deployment diagrams.



Package Diagram

- A package diagram shows the decomposition of the model itself into organisation units and their dependencies.



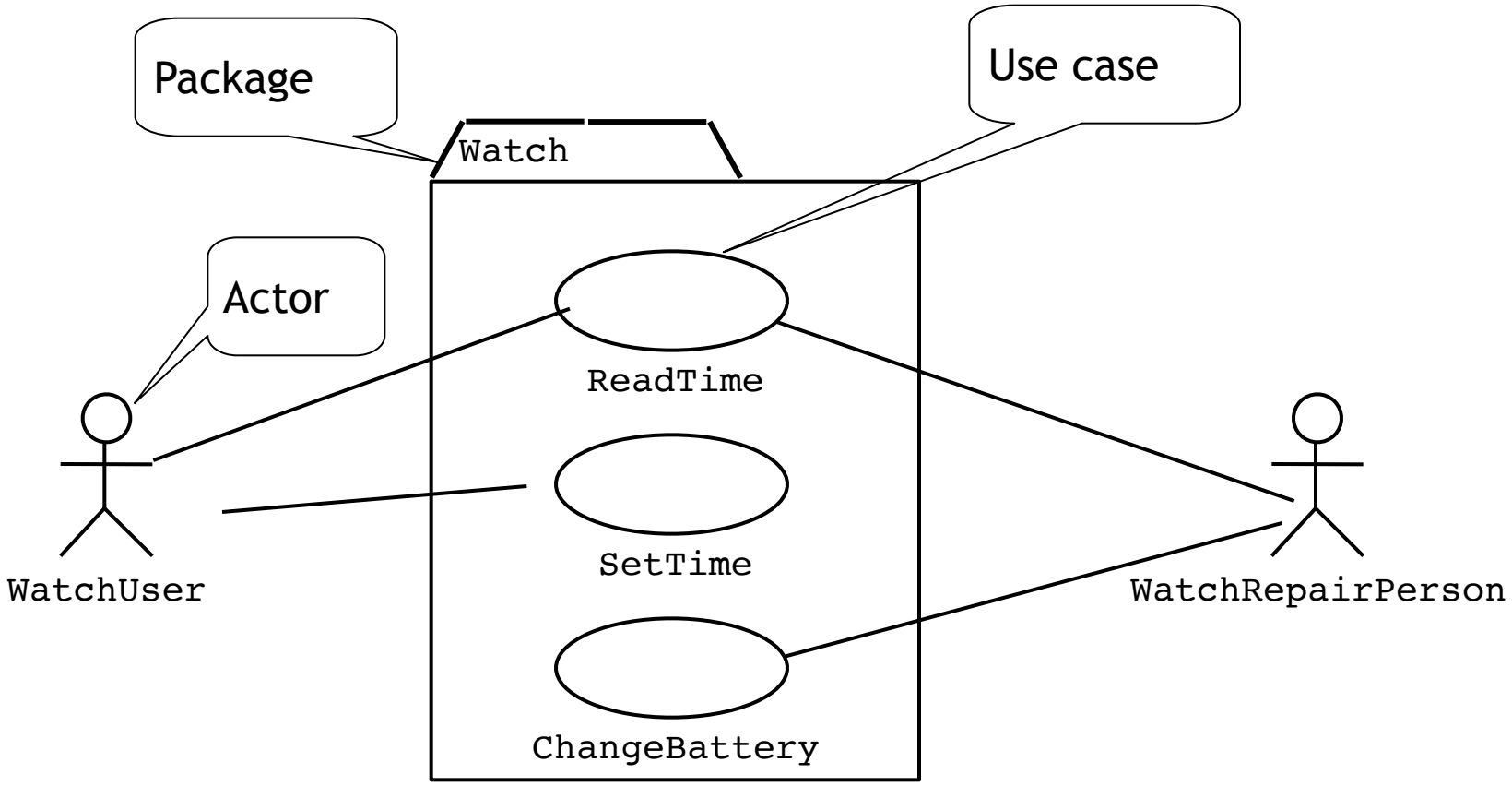


Interaction Overview Diagram

- An interaction overview diagram is a hybrid of an activity diagram and a sequence diagram

A simple example – Digital Watch

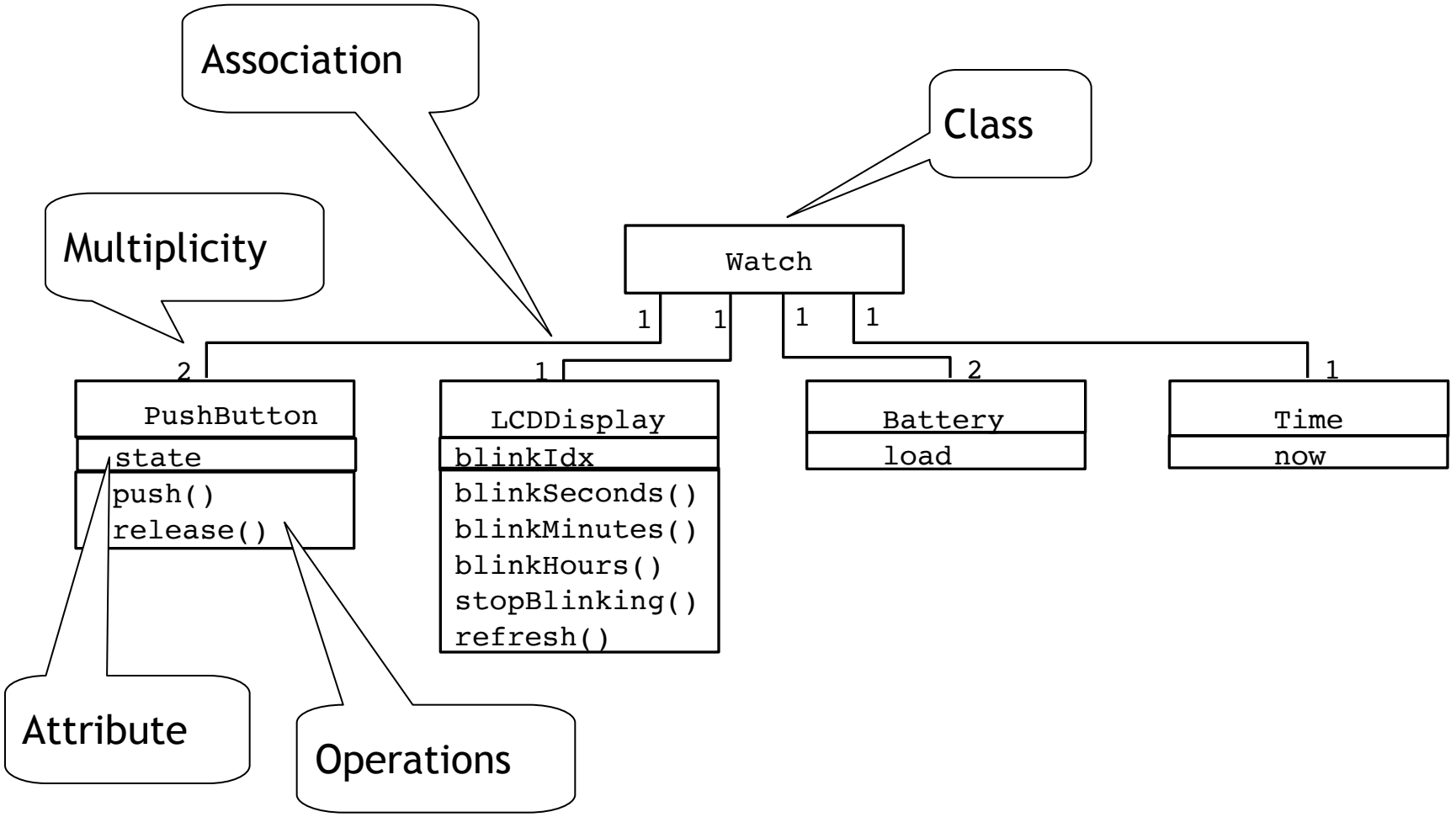
Use case diagrams



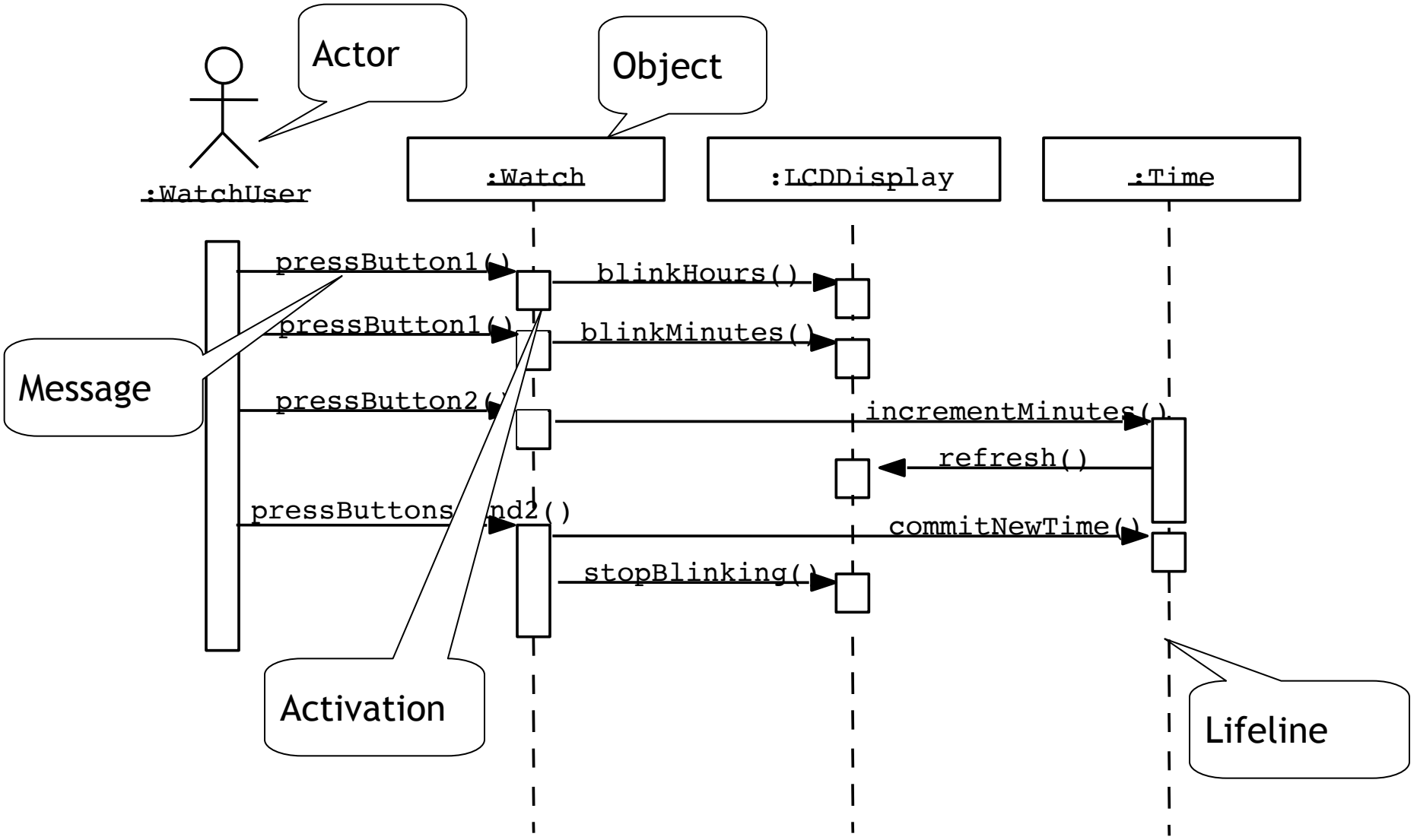
Use case diagrams represent the functionality of the system from user's point of view

Class diagrams

Class diagrams represent the structure of the system

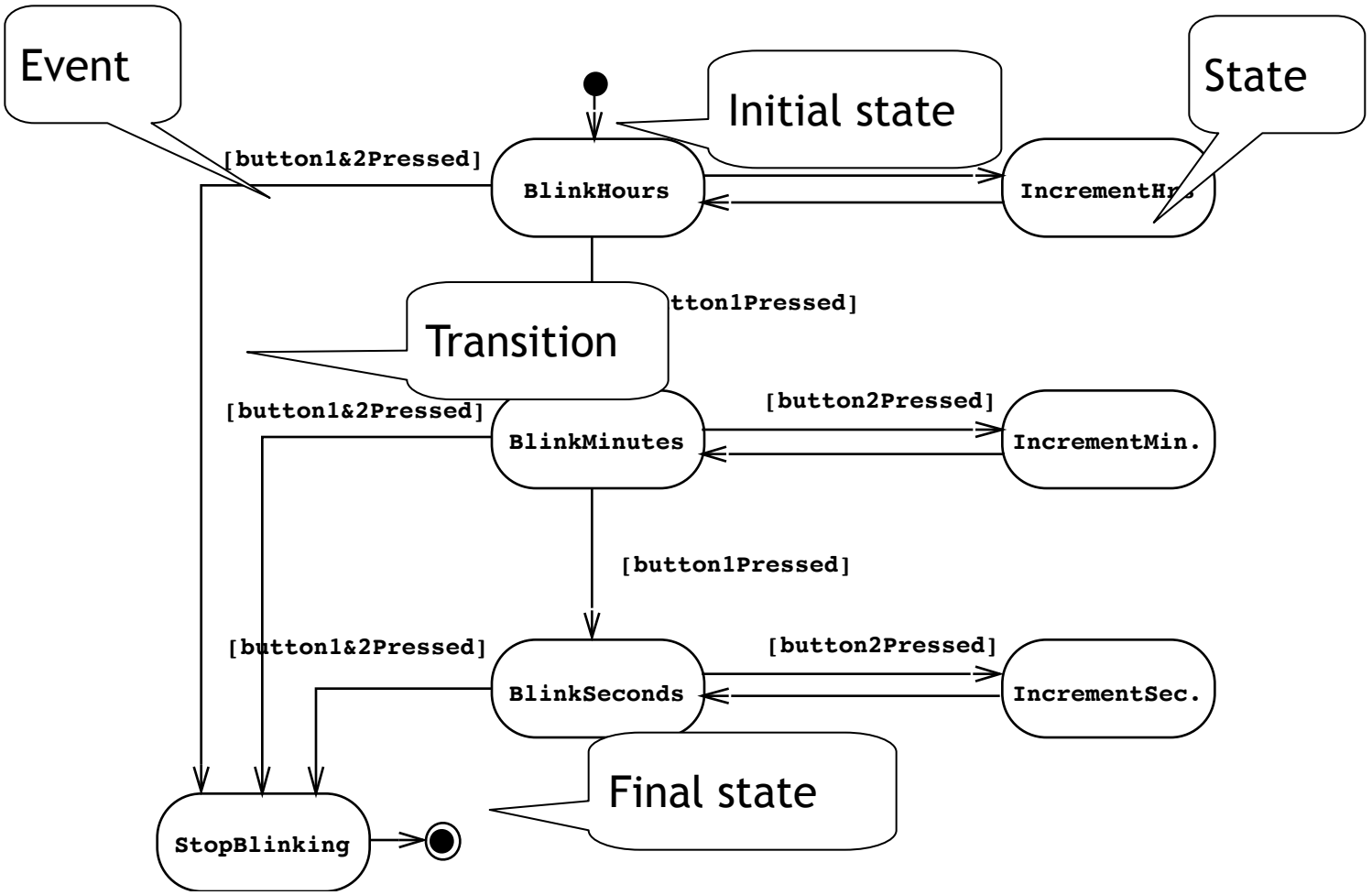


Sequence diagrams



Sequence diagrams represent the behavior as interactions

State diagrams

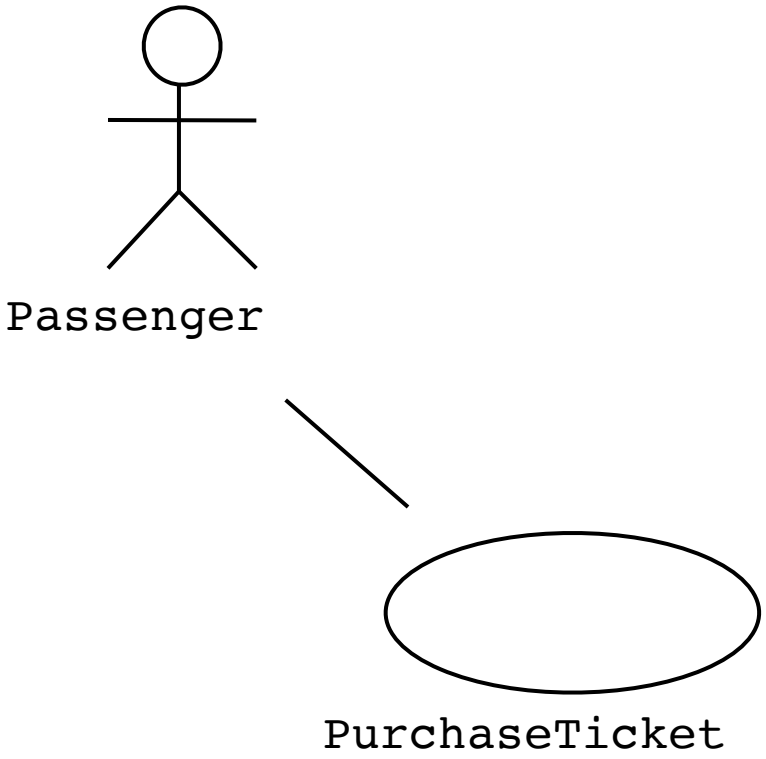


Represent behaviour as states and transitions
for objects with interesting dynamic behaviour

UML Core Conventions

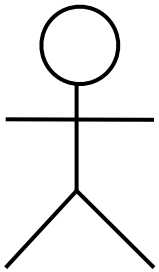
- Rectangles are classes or instances
- Ovals are functions or use cases
- Instances are denoted with an underlined names
  - `myWatch:SimpleWatch`
  - `Joe:Firefighter`
- Types are denoted with non-underlined names
  - `SimpleWatch`
  - `Firefighter`
- Diagrams are graphs
  - Nodes are entities
  - Arcs are relationships between entities

Use Case Diagrams



- Used during requirements elicitation to represent external behavior
- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

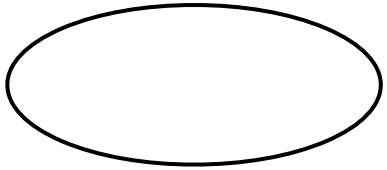
Actors



- An actor models an external entity which communicates with the system:
  - User
  - External system
  - Physical environment
- An actor has a unique name and an optional description.
- Examples:
  - Passenger: A person in the train
  - GPS satellite: Provides the system with GPS coordinates

Use Case

A use case represents a class of functionality provided by the system as an event flow.



PurchaseTicket

A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

Use Case: Example

Name: `Purchase ticket`

Participating actor: `Passenger`

Entry condition:

- `Passenger` standing in front of ticket distributor.
- `Passenger` has sufficient money to purchase ticket.

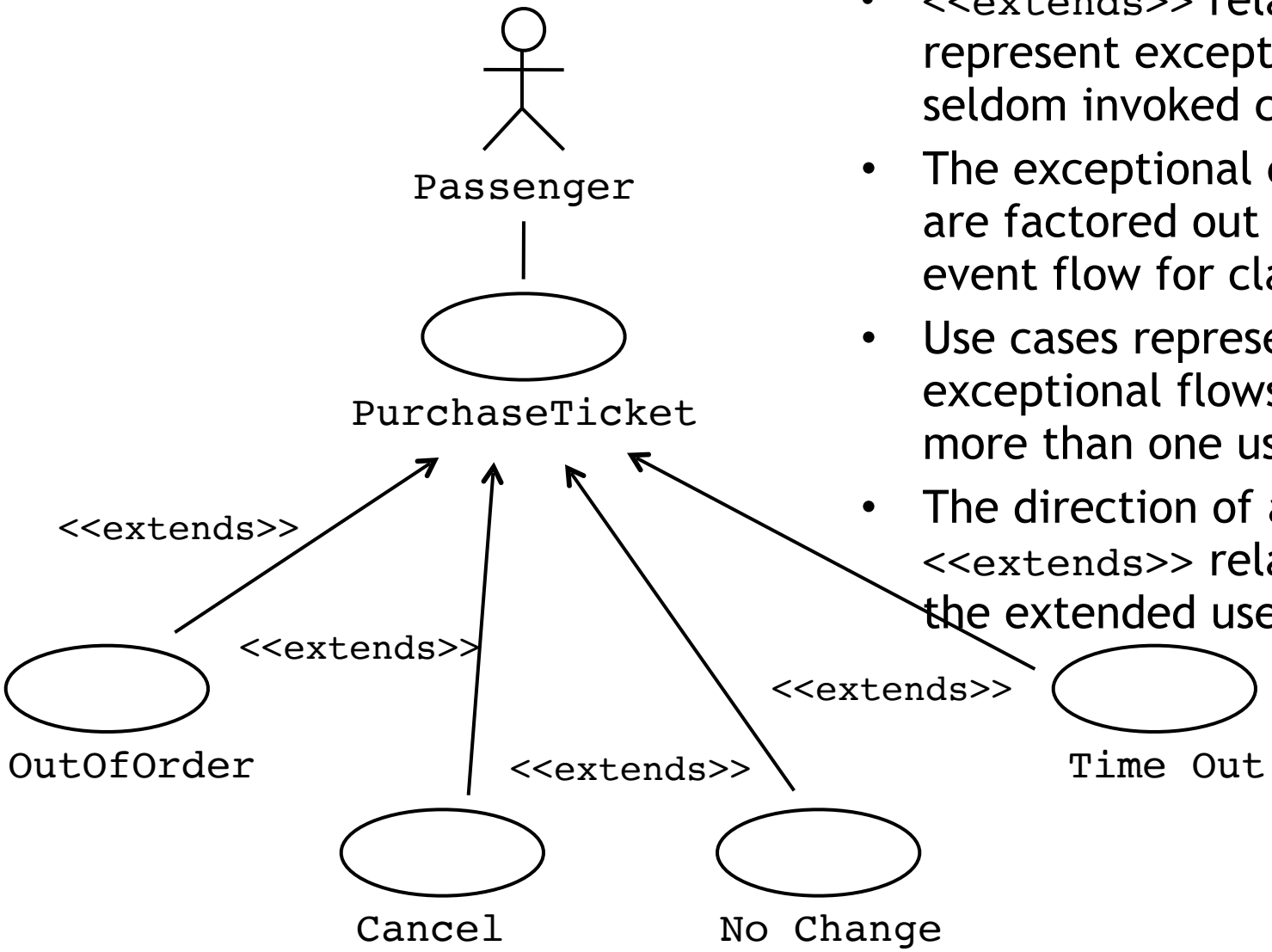
Exit condition:

- `Passenger` has ticket.

Event flow:

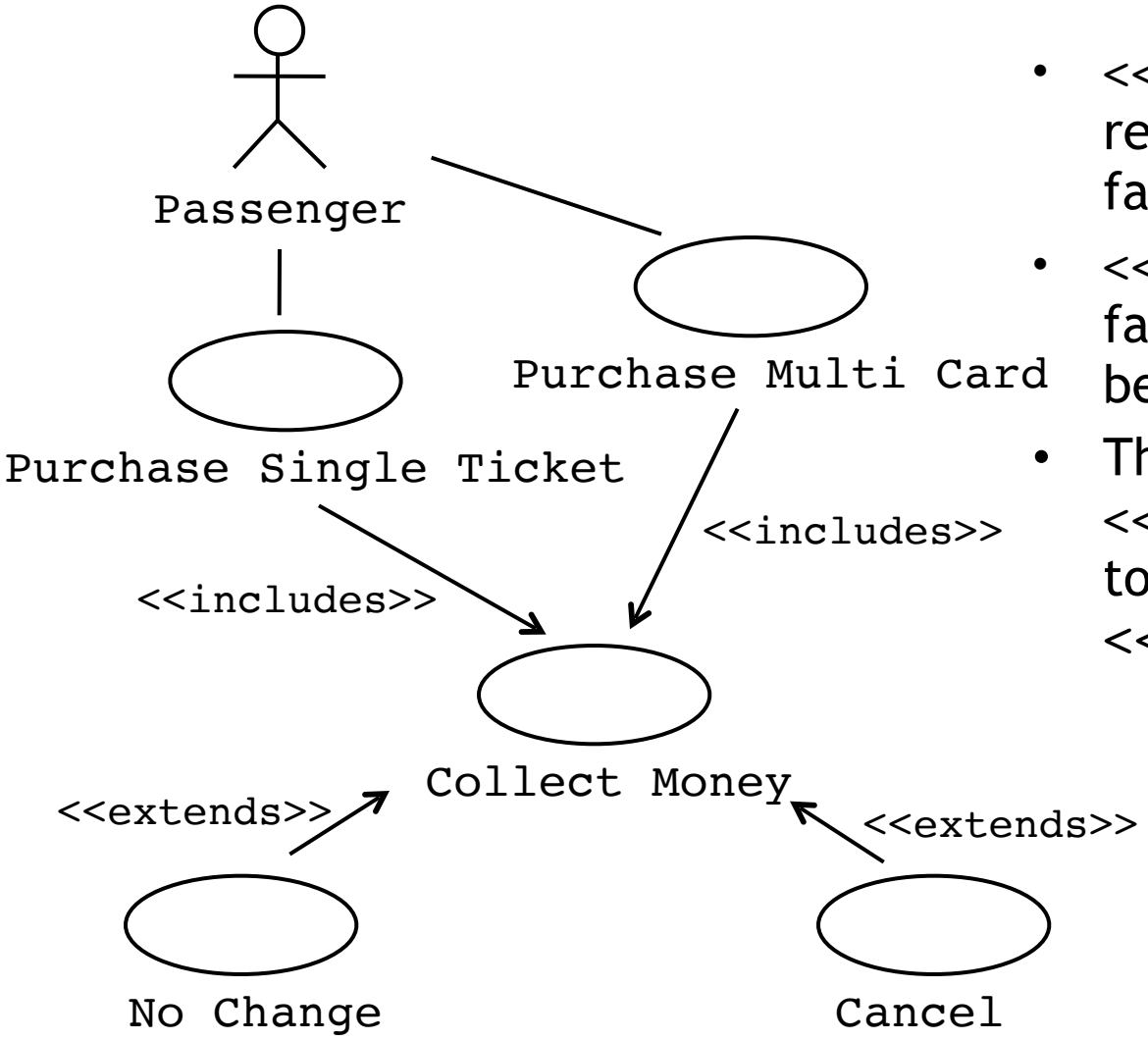
1. `Passenger` selects the number of zones to be traveled.
2. `Distributor` displays the amount due.
3. `Passenger` inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

The <<extends>> Relationship



- <<extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of an <<extends>> relationship is to the extended use case

The <<includes>> Relationship

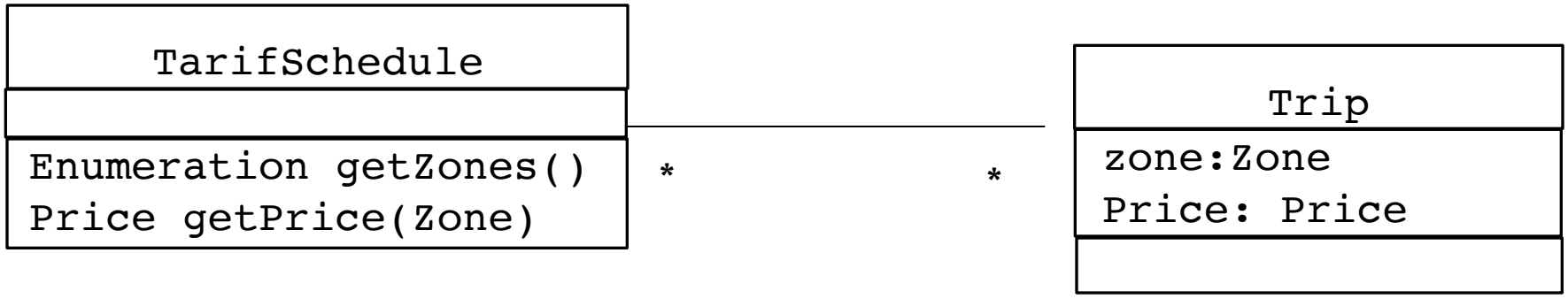


- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).

Use Case Diagrams: Summary

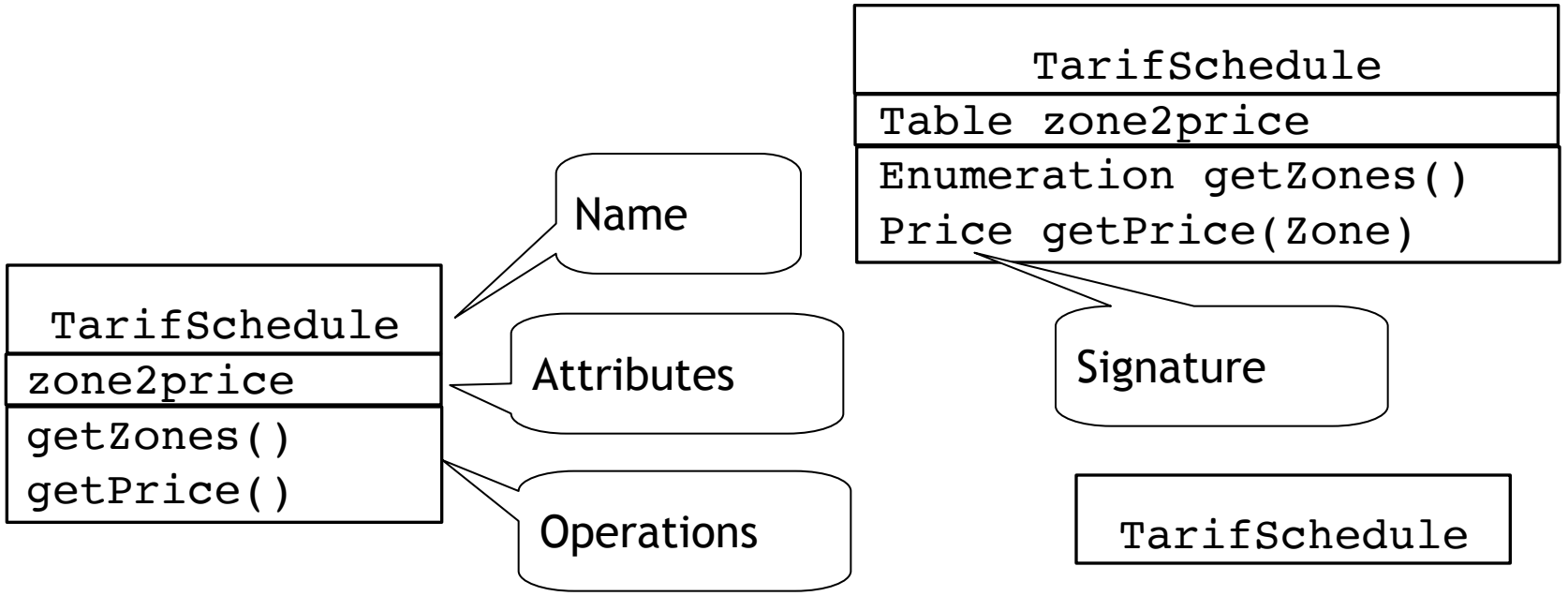
- Use case diagrams represent external behavior
- Use case diagrams are useful as an index into the use cases
- Use case descriptions provide meat of model, not the use case diagrams.
- All use cases need to be described for the model to be useful.

Class Diagrams



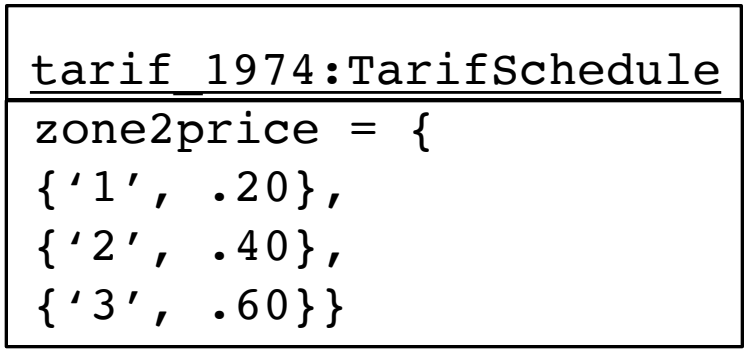
- Class diagrams represent the structure of the system.
- Used
  - during requirements analysis to model problem domain concepts
  - during system design to model subsystems and interfaces
  - during object design to model classes.

Classes



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

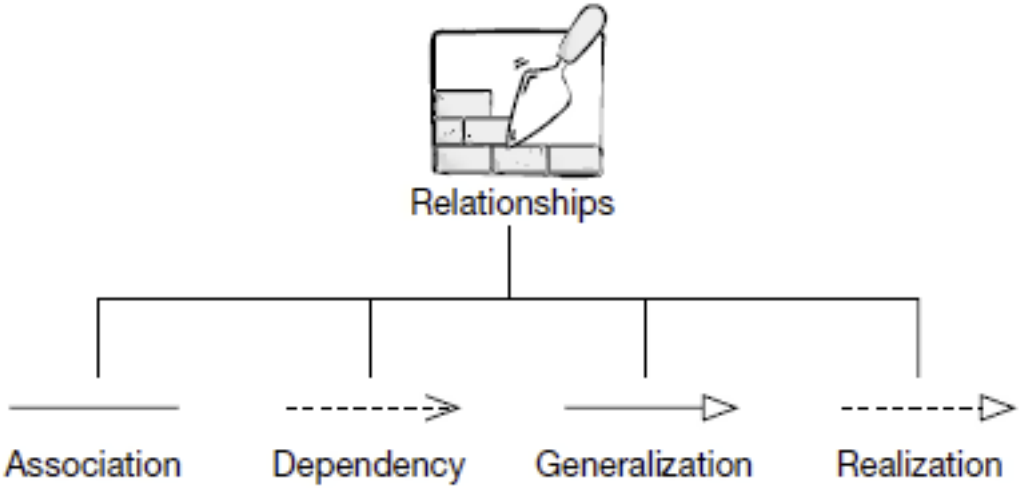
Instances



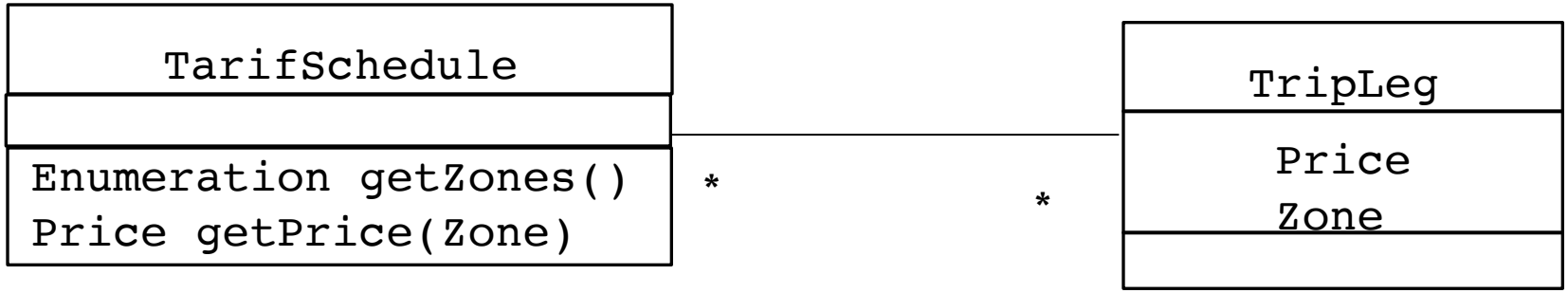
- An ***instance*** represents a phenomenon.
- The name of an instance is underlined and can contain the class of the instance.
- The attributes are represented with their ***values***.

Actor vs Instances

- What is the difference between an *actor*, a *class*, and an *instance*?
- Actor:
  - An entity outside the system to be modeled, interacting with the system (“Passenger”)
- Class:
  - An abstraction modeling an entity in the problem domain, must be modeled inside the system (“User”)
- Object:
  - A specific instance of a class (“Joe, the passenger who is purchasing a ticket from the ticket distributor”).

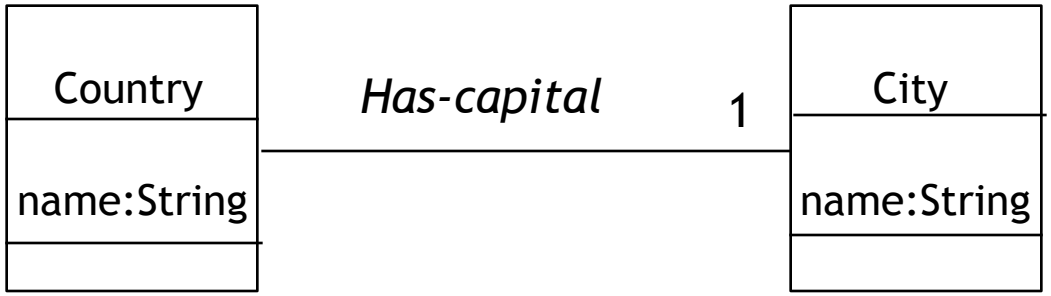

Relationships
Association
Dependency
Generalization
Realization

Associations



- Associations denote relationships between classes.
- The multiplicity of an association end denotes how many objects the source object can legitimately reference.

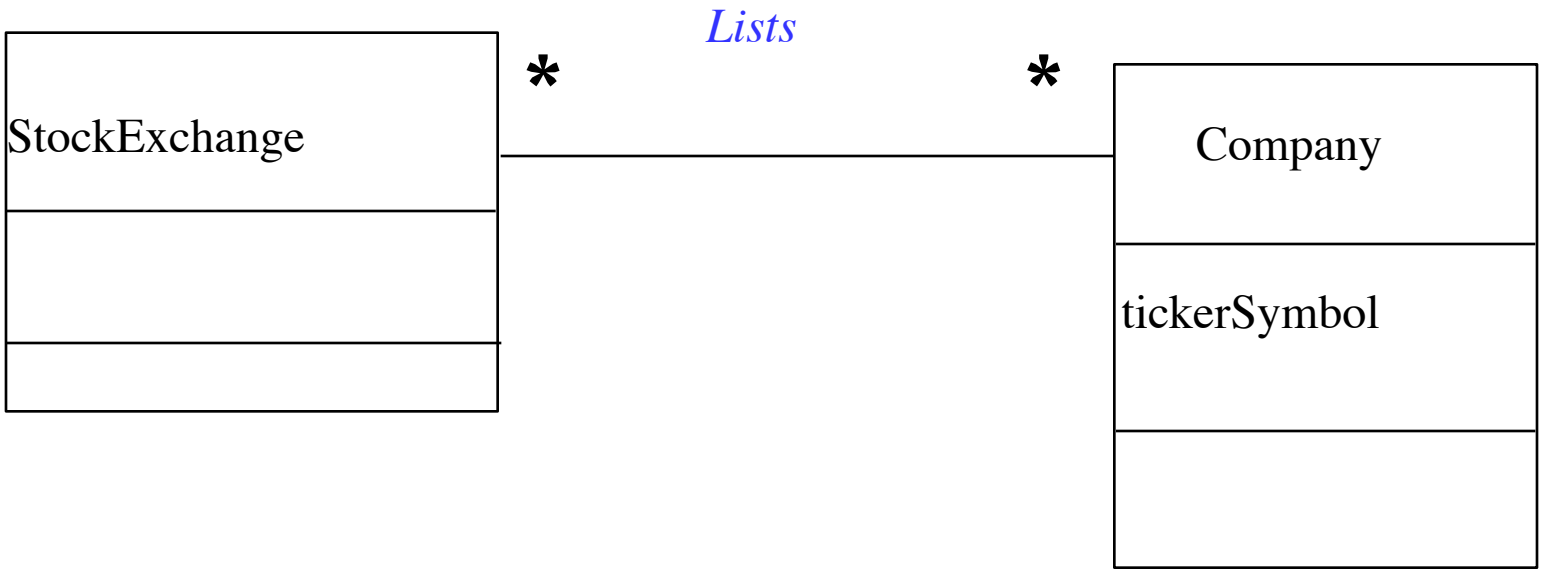
1-to-1 and 1-to-many Associations



One-to-one association

One-to-many association

Many-to-Many Associations



Dependency

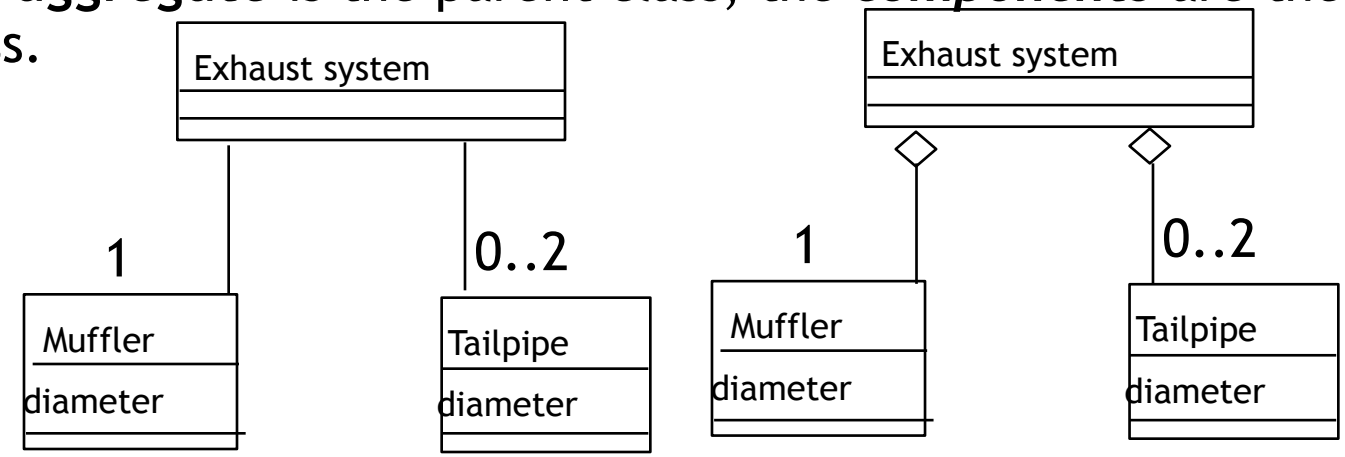
Using a dependency relationship in UML,

one can relate how various things inside a particular system are dependent on each other.

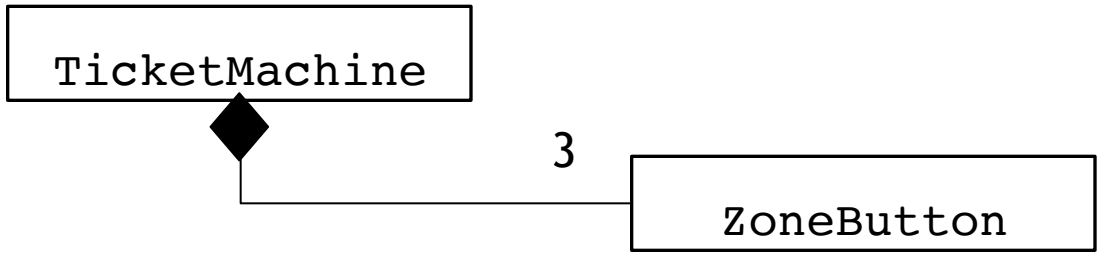
Dependency is used to describe the relationship between various elements in UML that are dependent upon each other.

Aggregation

- An ***aggregation*** is a special case of association denoting a “consists of” hierarchy.
- The ***aggregate*** is the parent class, the ***components*** are the children class.



- A solid diamond denotes ***composition***, a strong form of aggregation where components cannot exist without the aggregate. (Bill of Material)



Aggregation

is a relationship between two classes that is best described as a "has-a" and "whole/part" relationship. Car has-a Engine

- It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
- In **Aggregation**, **both the entries can survive individually** which means ending one entity will not effect the other entity

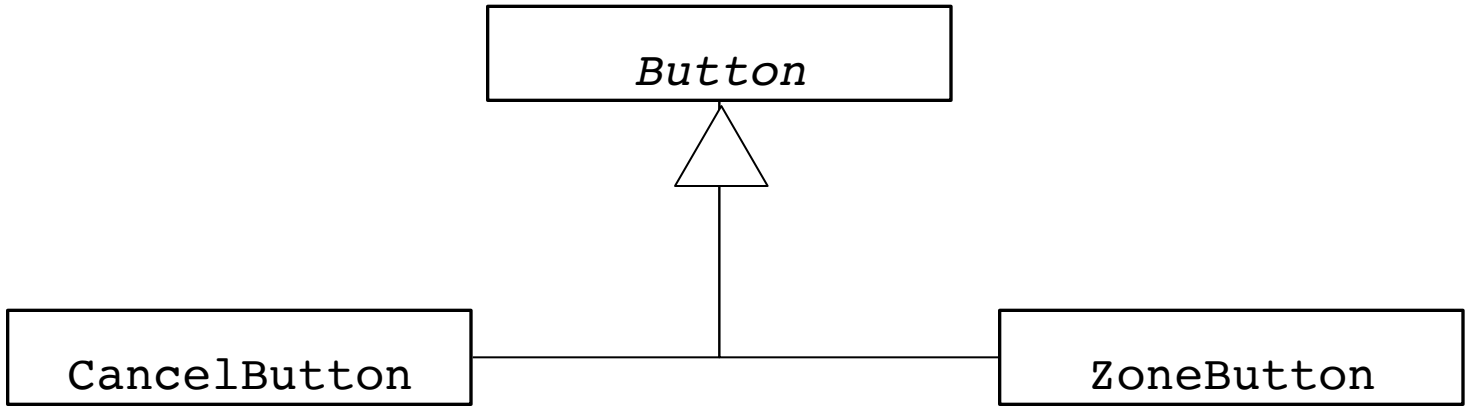
Composition

It represents **part-of** relationship Apple is a fruit

Composition is a restricted form of Aggregation
in which two entities are highly dependent on each other.

- both the entities are dependent on each other.
- When there is a composition between two entities, the composed object **cannot exist** without the other entity.

Inheritance



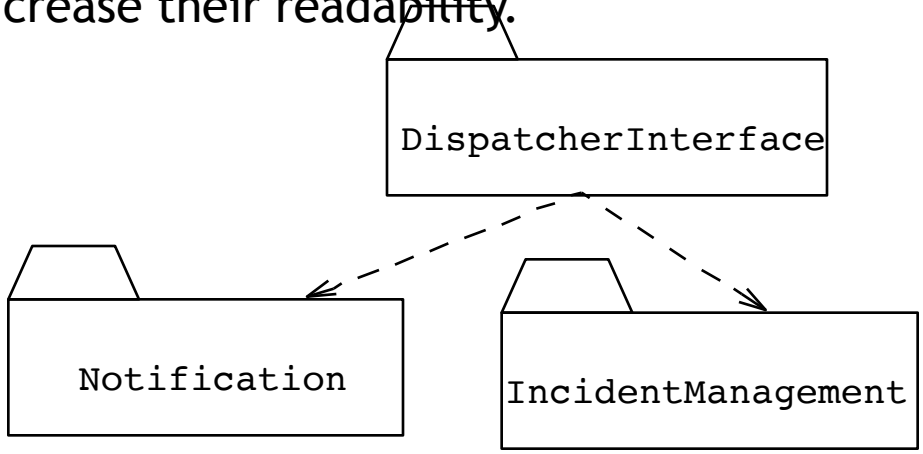
- The **children classes** inherit the attributes and operations of the **parent class**.
- Inheritance simplifies the model by eliminating redundancy.

```
Class Bird {
        void fly(){
            “I can fly” }
    };

Class Parrot extends Bird{
                fly(); }

Class Ostrich extends Bird{
                fly(); }
```

// Oops! We are going WRONG here……
// Breaking the rule of SOLID (Liskov) Principle
// Use Composition over Inheritance

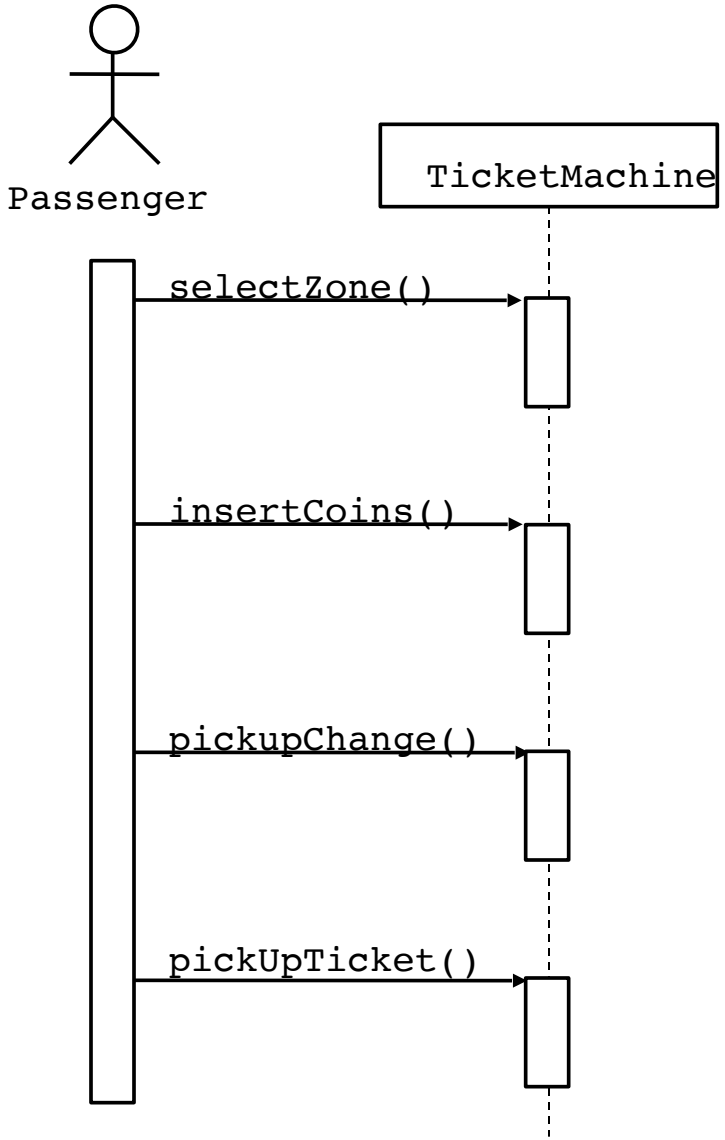
Packages

- A package is a UML mechanism for organizing elements into groups (usually not an application domain concept)
- Packages are the basic grouping construct with which you may organize UML models to increase their readability.



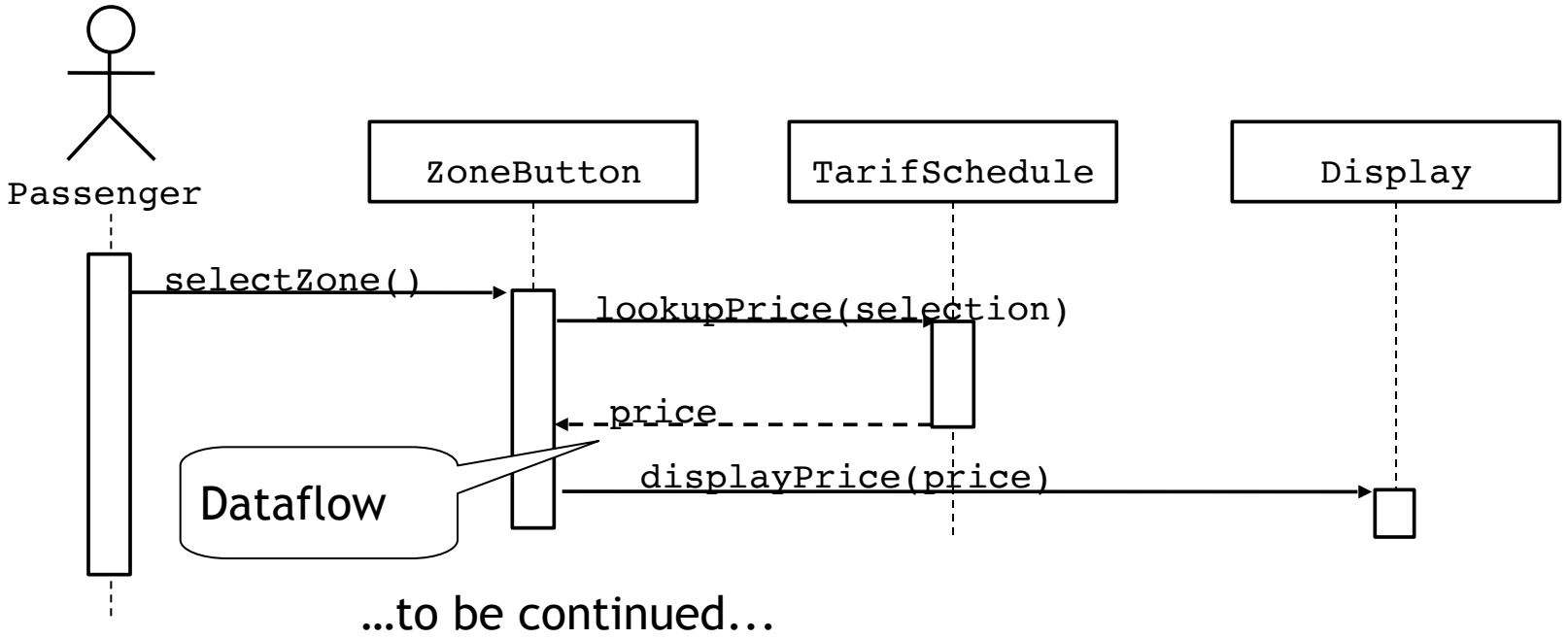
- A complex system can be decomposed into subsystems, where each subsystem is modeled as a package

UML sequence diagrams



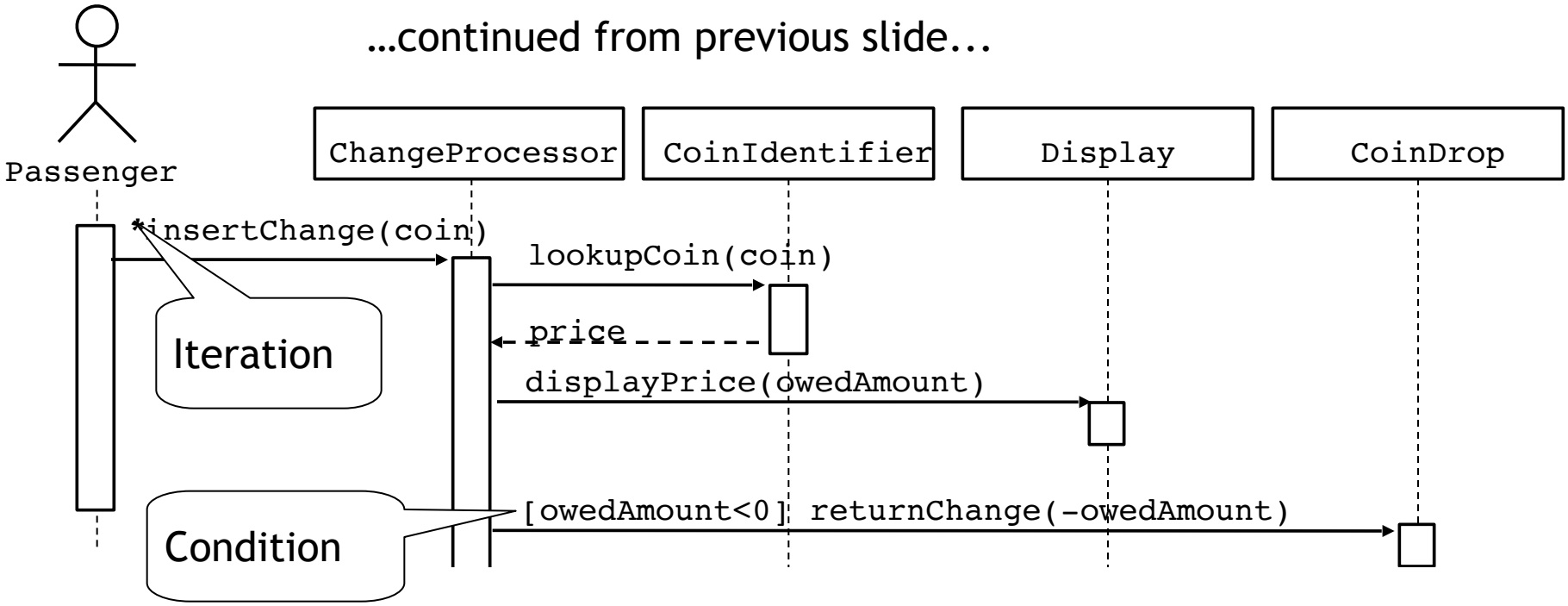
- Used during requirements analysis
  - To refine use case descriptions
  - to find additional objects ("participating objects")
- Used during system design
  - to refine subsystem interfaces
- ***Classes*** are represented by columns
- ***Messages*** are represented by arrows
- ***Activations*** are represented by narrow rectangles
- ***Lifelines*** are represented by dashed lines

Nested messages



- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations
- Horizontal dashed arrows indicate data flow
- Vertical dashed lines indicate lifelines

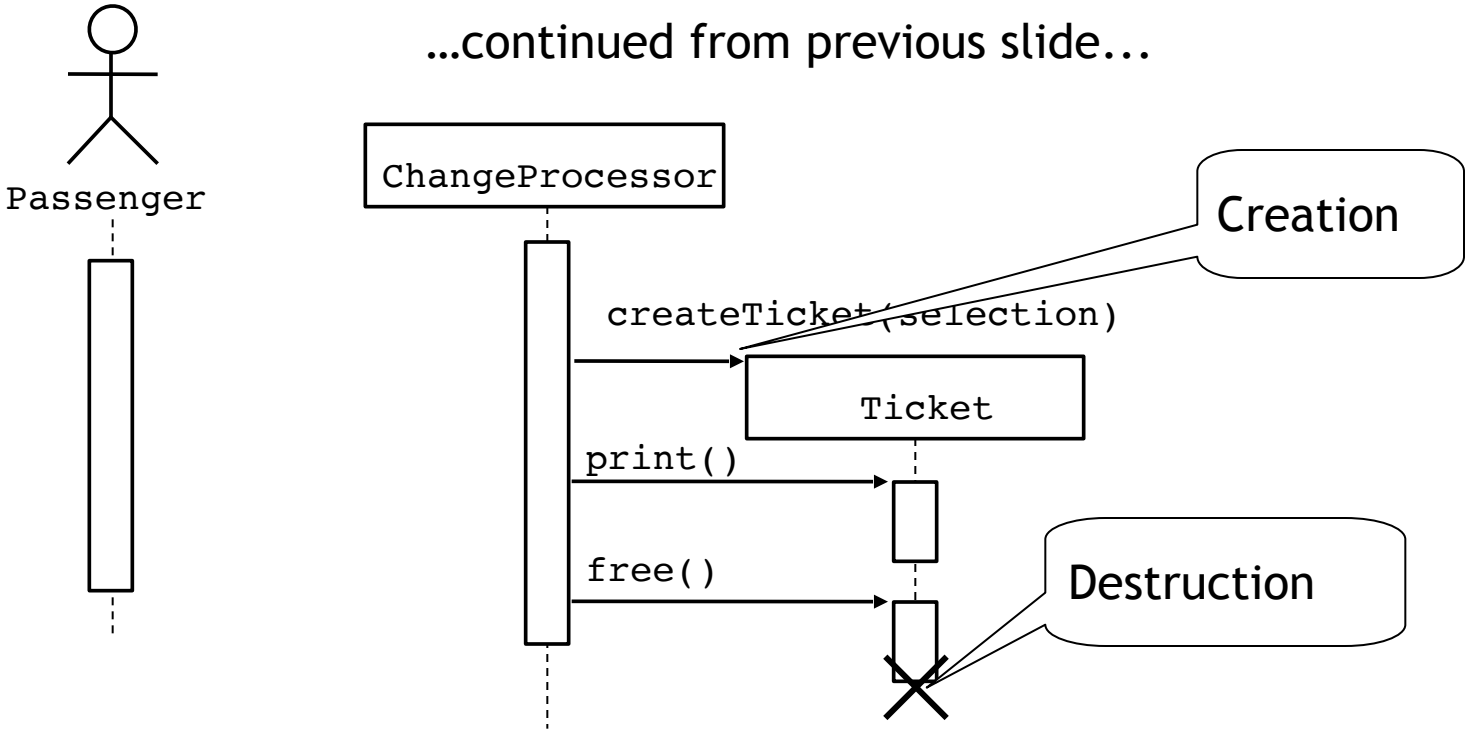
Iteration & condition



- Iteration is denoted by a * preceding the message name
- Condition is denoted by boolean expression in [ ] before the message name

Creation and destruction

…continued from previous slide...

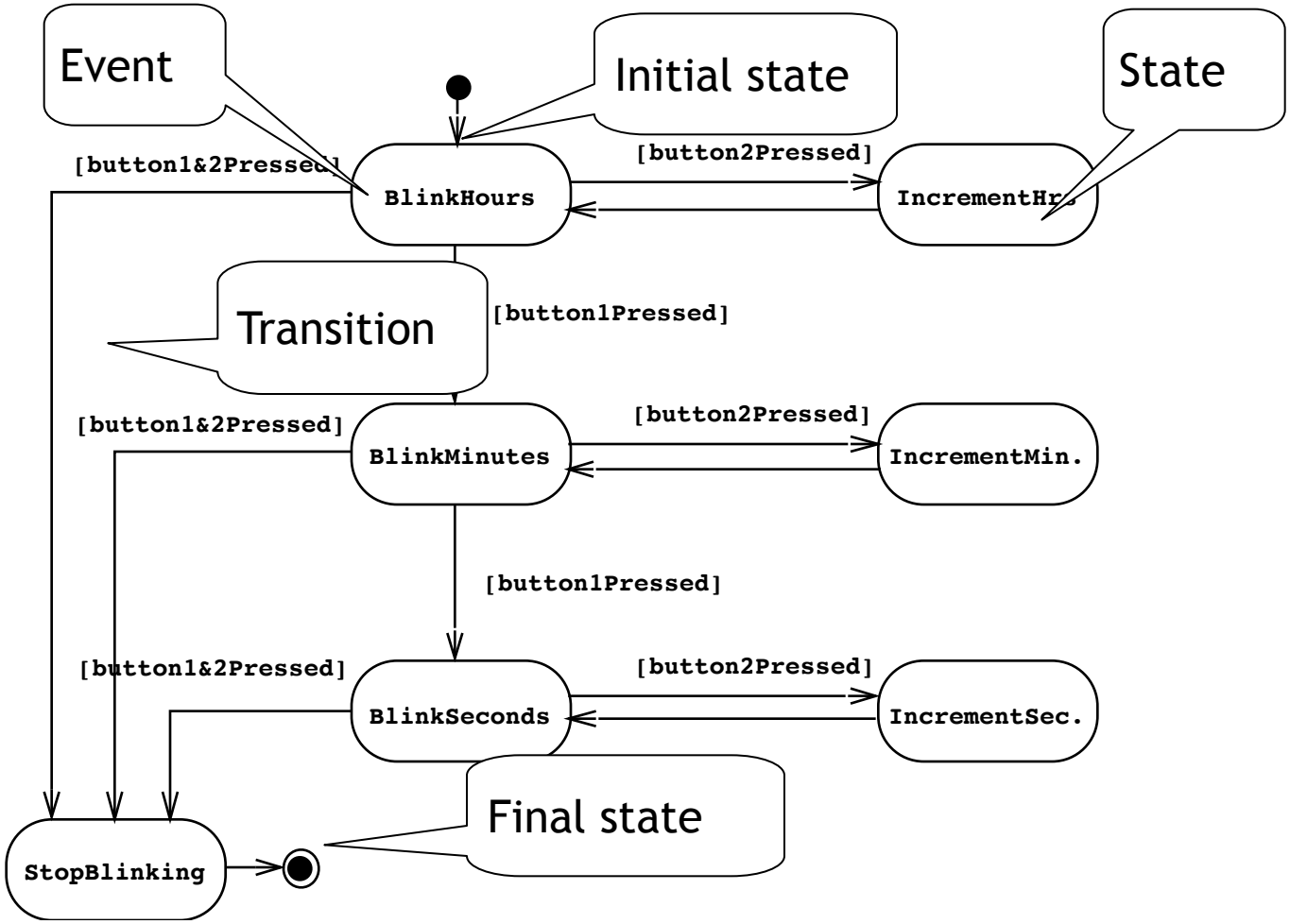


- Creation is denoted by a message arrow pointing to the object.
- Destruction is denoted by an X mark at the end of the destruction activation.
- In garbage collection environments, destruction can be used to denote the end of the useful life of an object.

Sequence Diagram Summary

- UML sequence diagram represent behavior in terms of interactions.
- Useful to find missing objects.
- Time consuming to build but worth the investment.
- Complement the class diagrams (which represent structure).

State Chart Diagrams

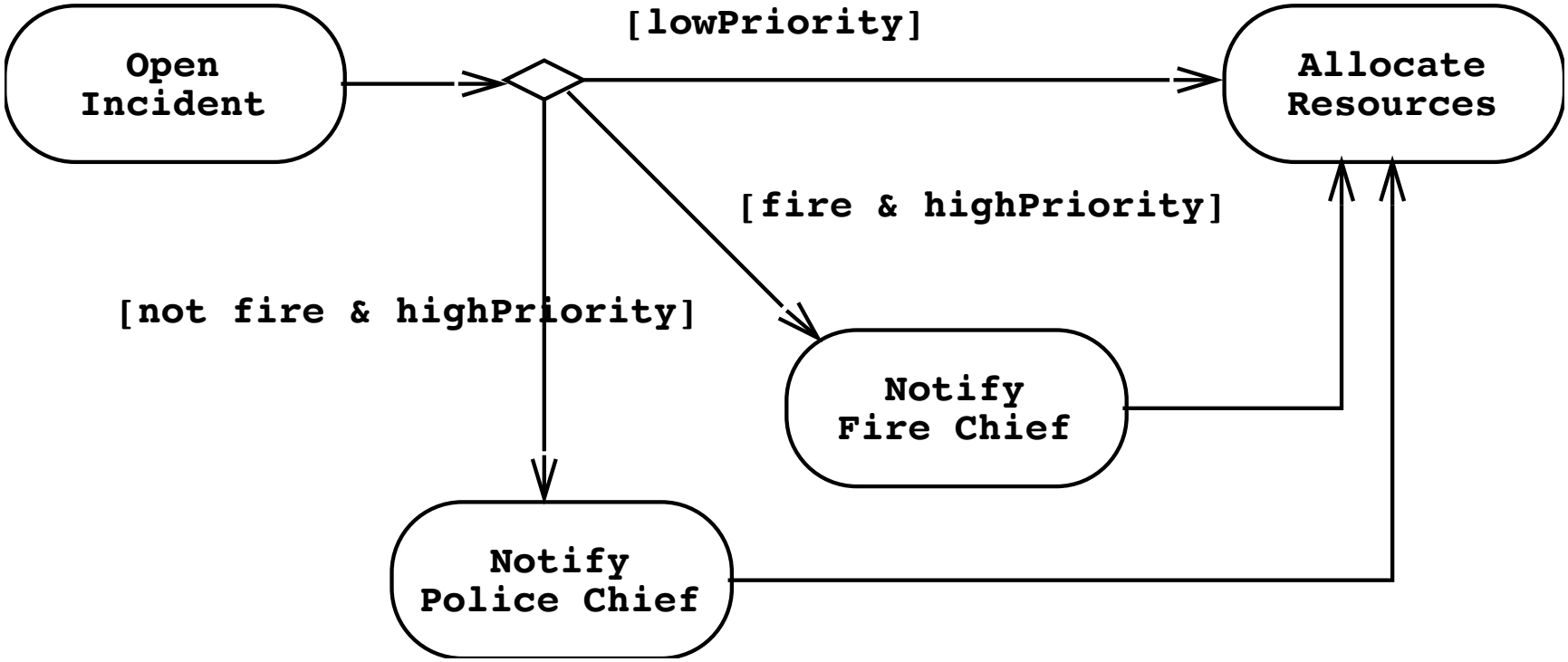


Represent behavior as states and transitions

Activity Diagrams

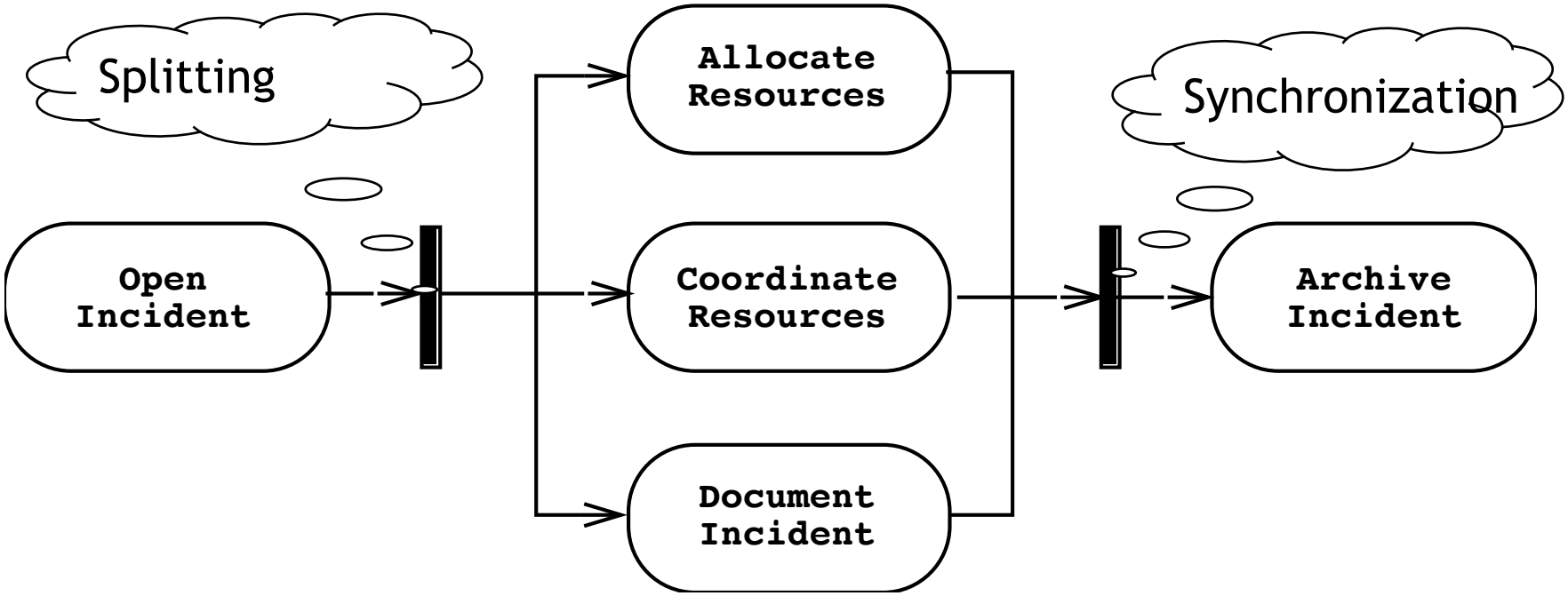
- An activity diagram shows flow control within a system

Activity Diagram: Modeling Decisions



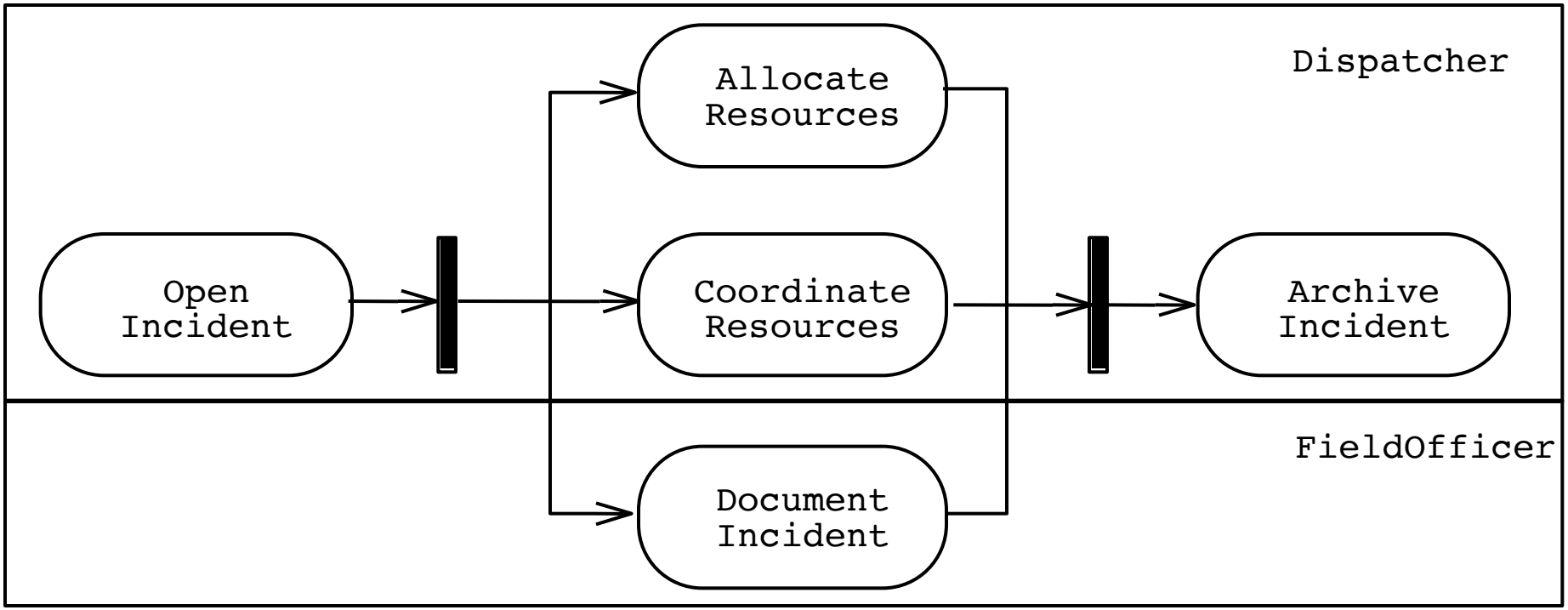
Activity Diagrams: Modeling Concurrency

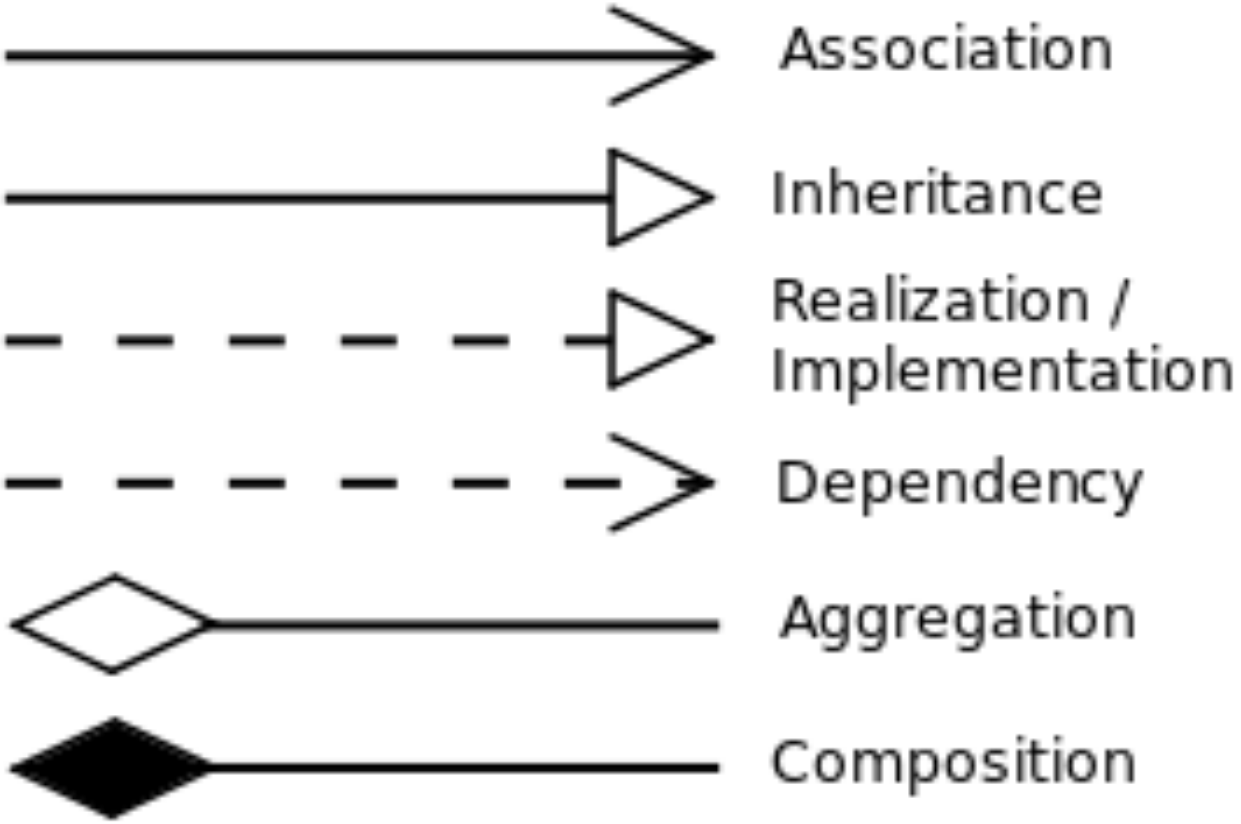
- Synchronization of multiple activities
- Splitting the flow of control into multiple threads



Activity Diagrams: Swimlanes

- Actions may be grouped into swimlanes to denote the object or subsystem that implements the actions.

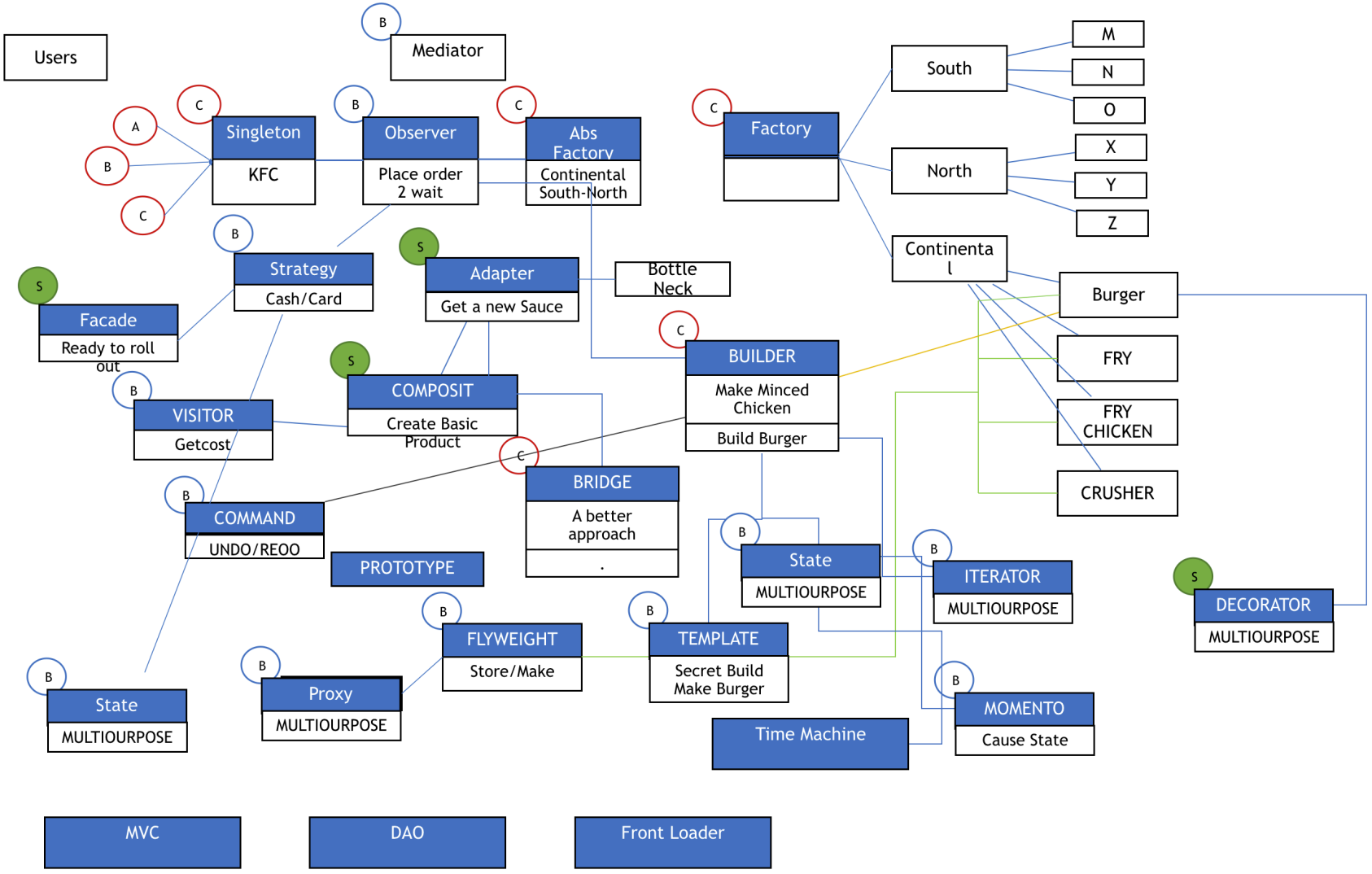



Association
Inheritance
Realization /
Implementation
Dependency
Aggregation
Composition

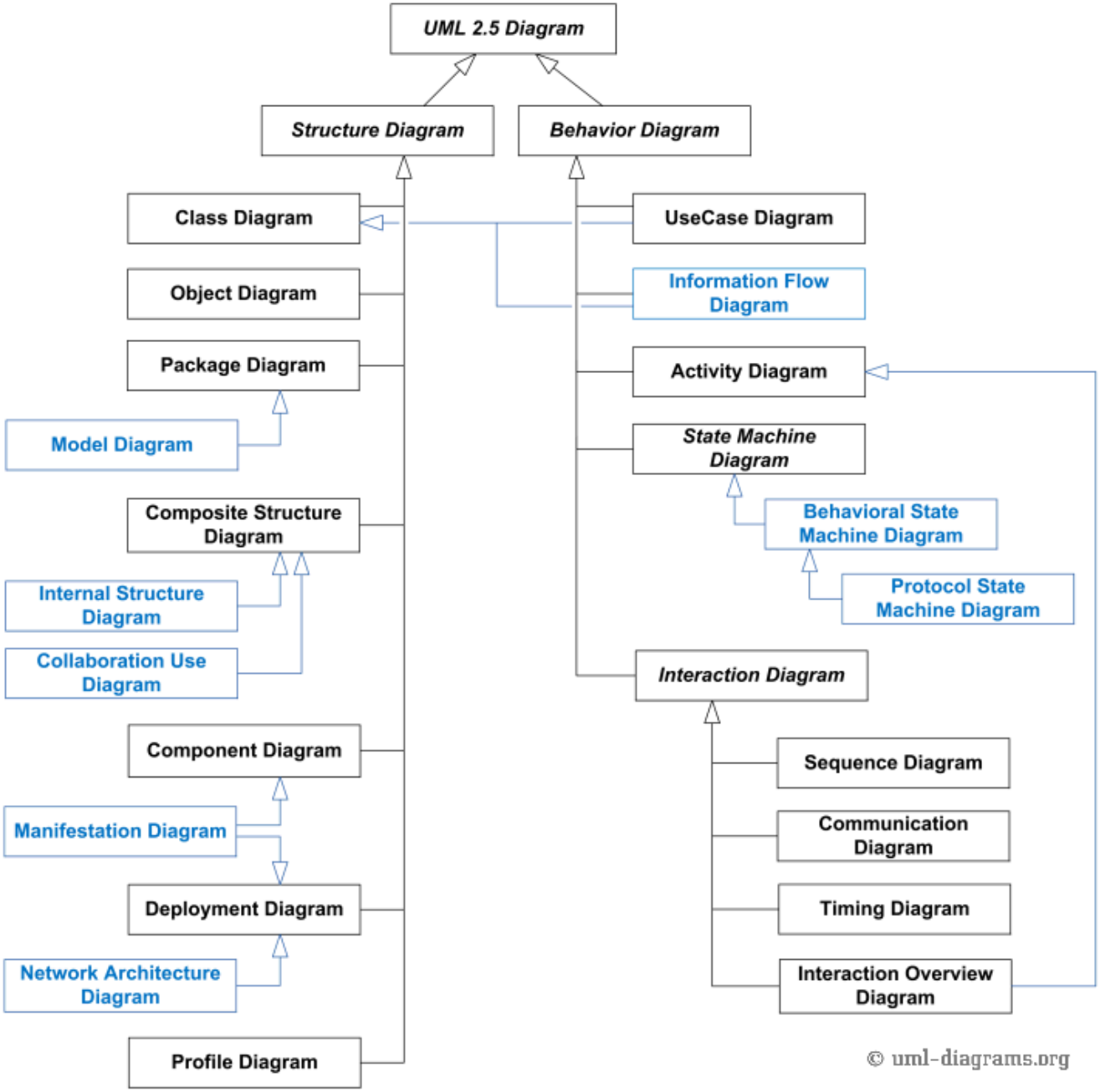
S.O.L.I.D Principals

1. S: Single responsibility principle
2. O: Open-closed principle
3. L: Liskov substitution principle
4. I: Interface segregation principle
5. D: Dependency inversion principle

Use : https://www.uml-diagrams.org/uml-25-diagrams.html



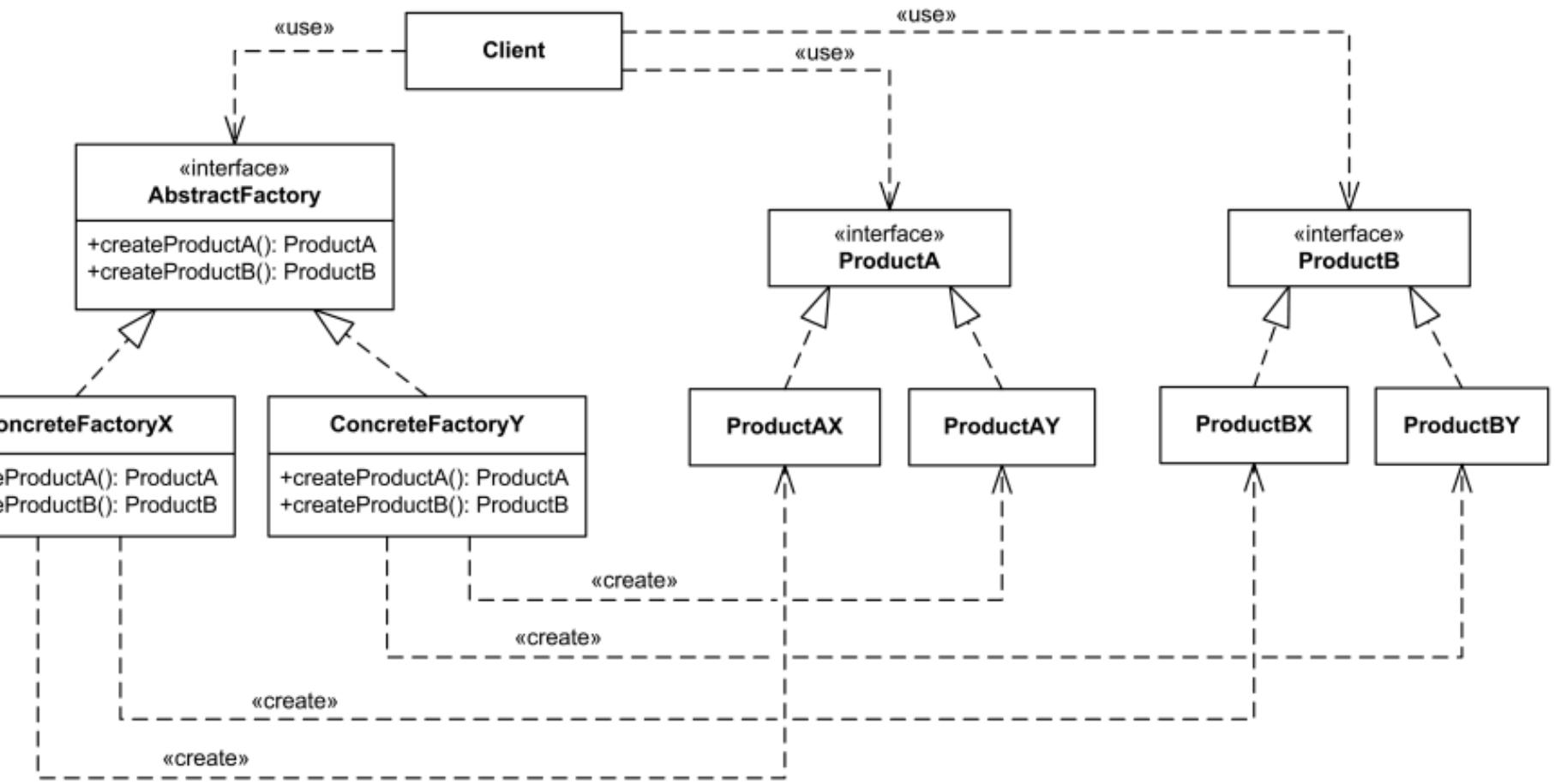
Activity diagram


UML 2.5 Diagram
Structure Diagram
Behavior Diagram
Class Diagram
Object Diagram
Package Diagram
Model Diagram
Composite Structure Diagram
Internal Structure Diagram
Collaboration Use Diagram
Component Diagram
Manifestation Diagram
Deployment Diagram
Network Architecture Diagram
Profile Diagram
UseCase Diagram
Information Flow Diagram
Activity Diagram
State Machine Diagram
Behavioral State Machine Diagram
Protocol State Machine Diagram
Interaction Diagram
Sequence Diagram
Communication Diagram
Timing Diagram
Interaction Overview Diagram
© uml-diagrams.org

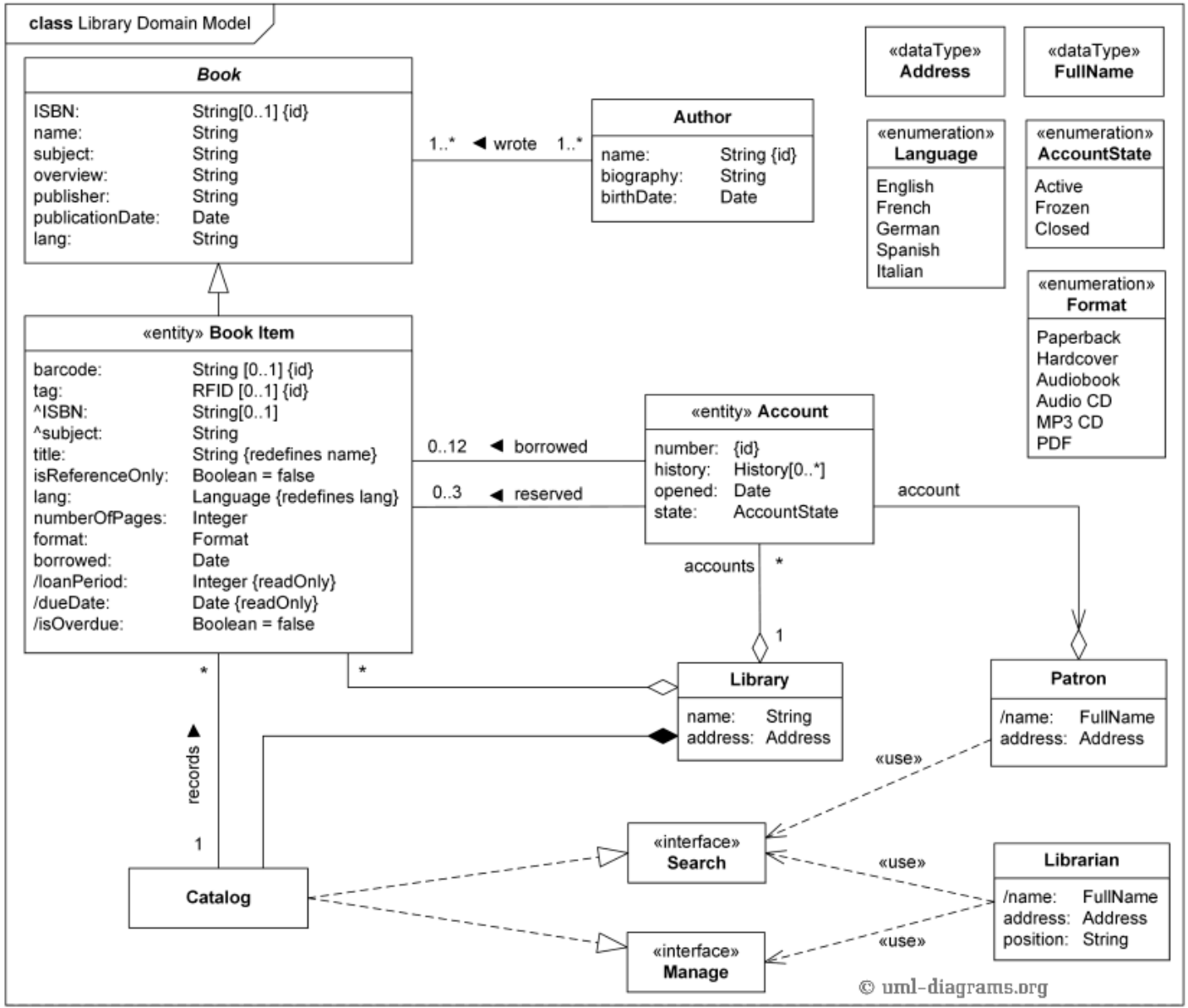
Diagrams in UML

- Class diagram
- Object diagram
- Component diagram
- Composite structure diagram
- Use case diagram
- Sequence diagram
- Communication diagram
- State diagram
- Activity diagram
- Deployment diagram
- Package diagram
- Timing diagram
- Interaction overview diagram

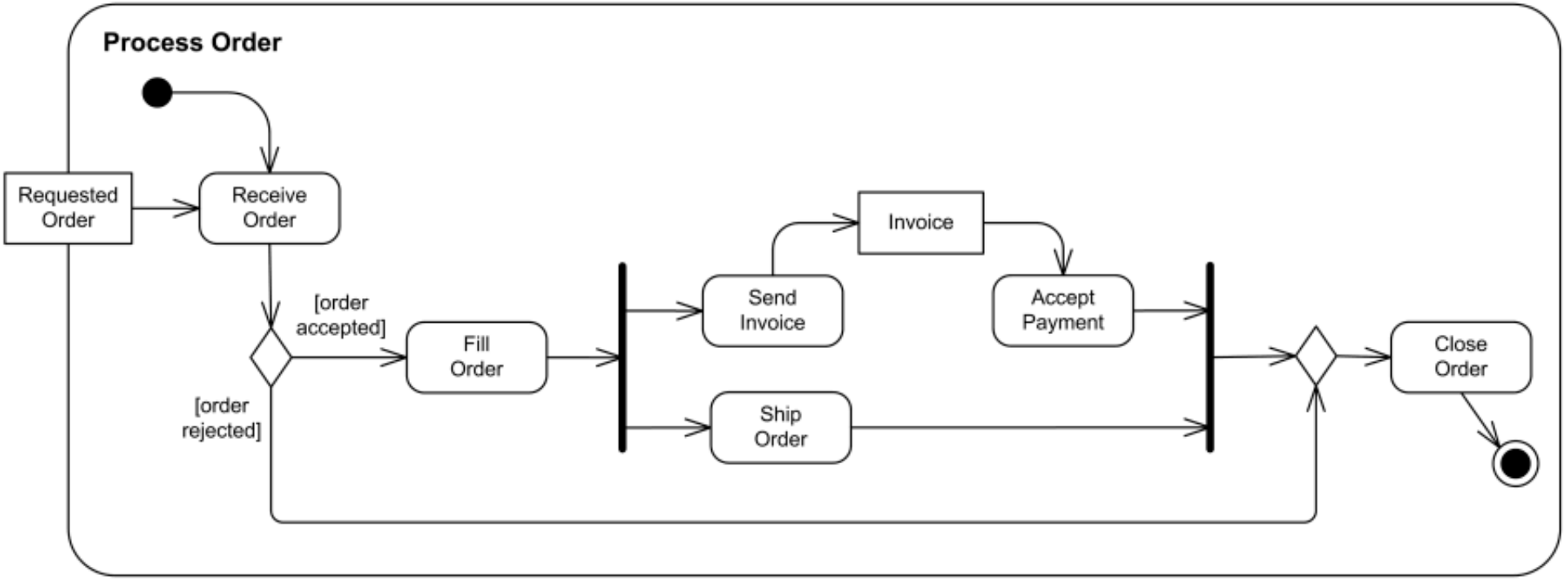
Abstract factory provider diagram



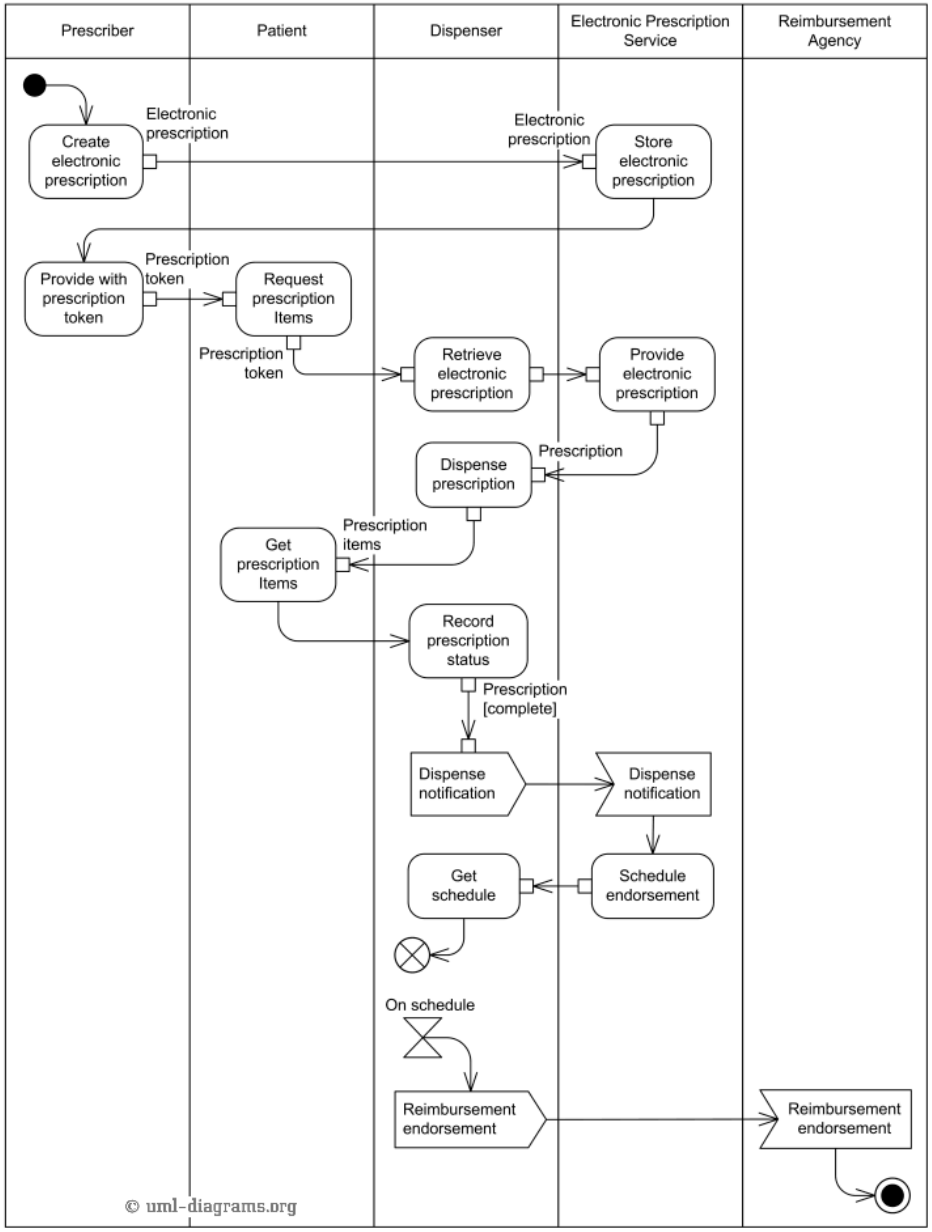
UML Class diagram



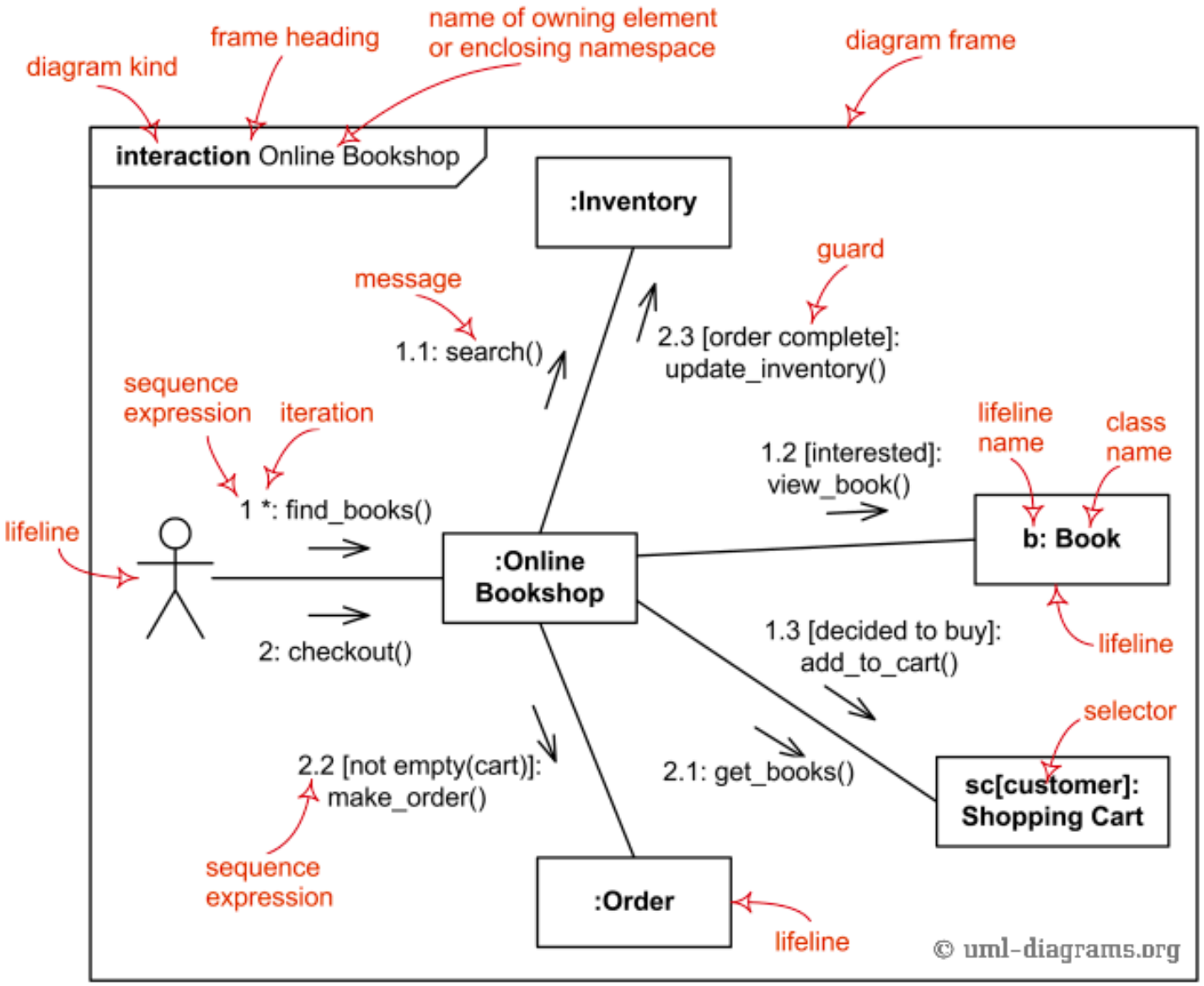
Activity diagram



Activity diagram



Communication diagram

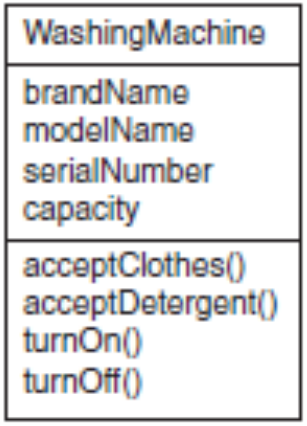


Timing diagram with states

Class diagram

Class Diagram

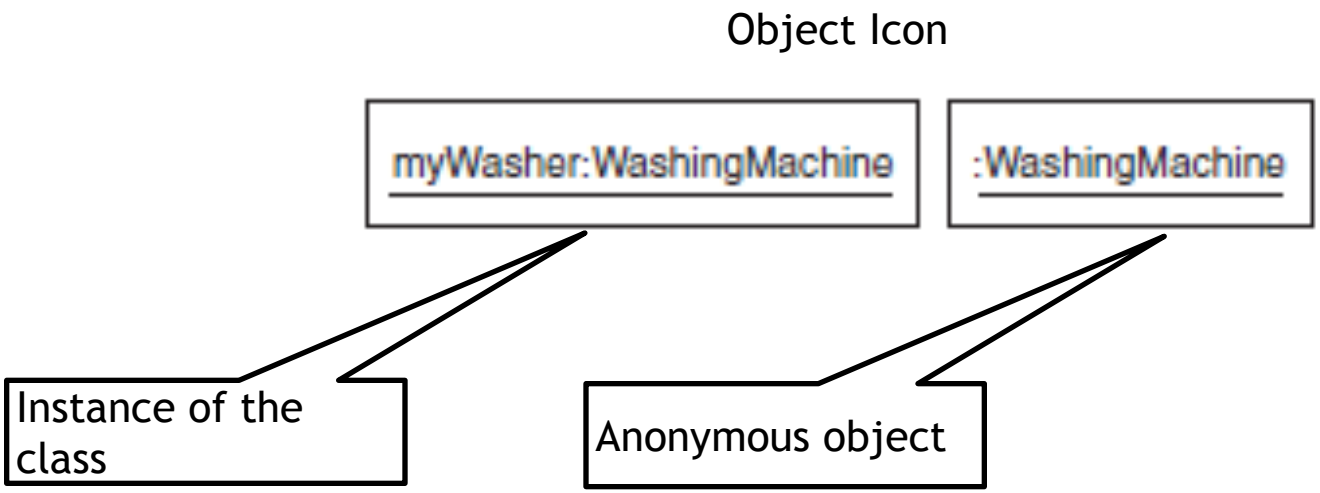
- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- Class diagrams address the static design view of a system.

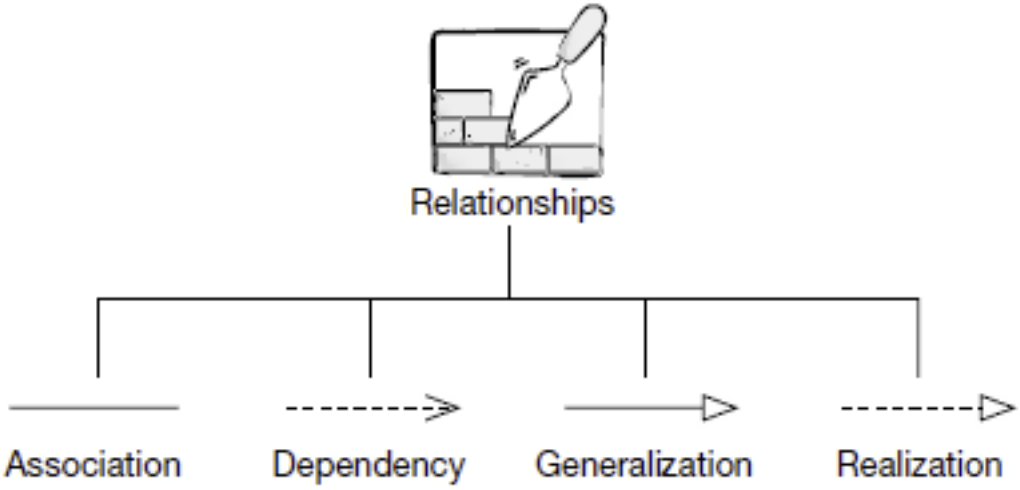


Class Icon

Object Diagram

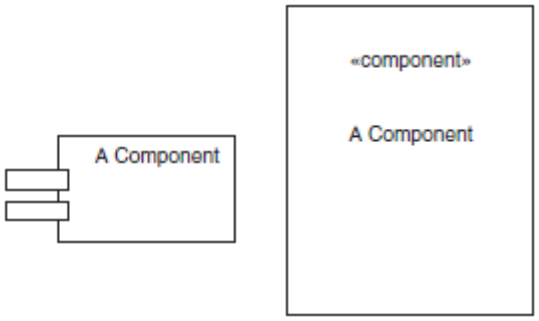
- An object diagram shows a set of objects and their relationships.
- Object diagrams represent static snapshots of instances of the things found in class diagrams.




Relationships
Association
Dependency
Generalization
Realization

Component Diagram

- A component diagram is shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors.
- Component diagrams address the static design implementation view of a system. They are important for building large systems from smaller parts. .



Component Icon

Class

Class: **Bank**

Abstract class: *Italics*

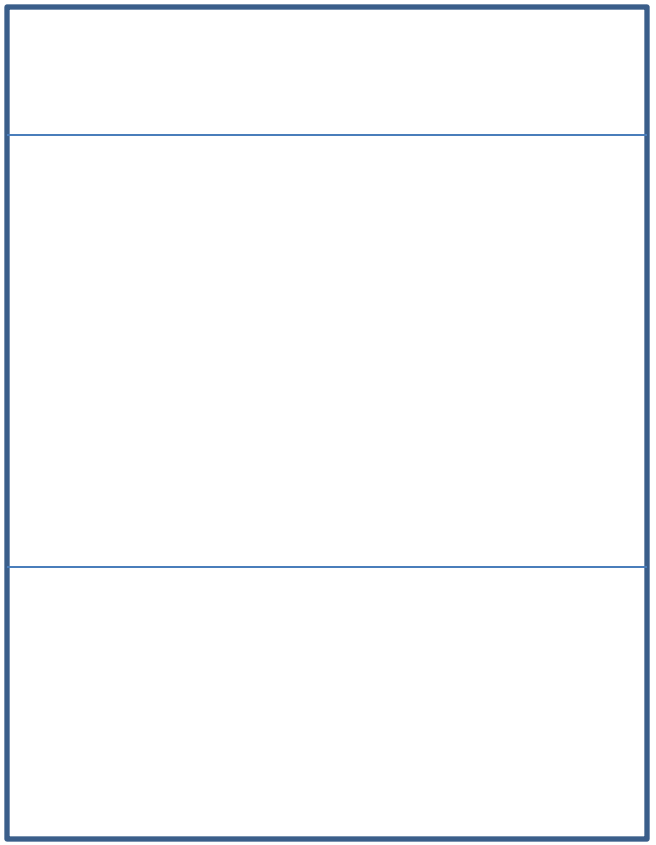
Interface: <<InterfaceName>>

+ public

- private

protected

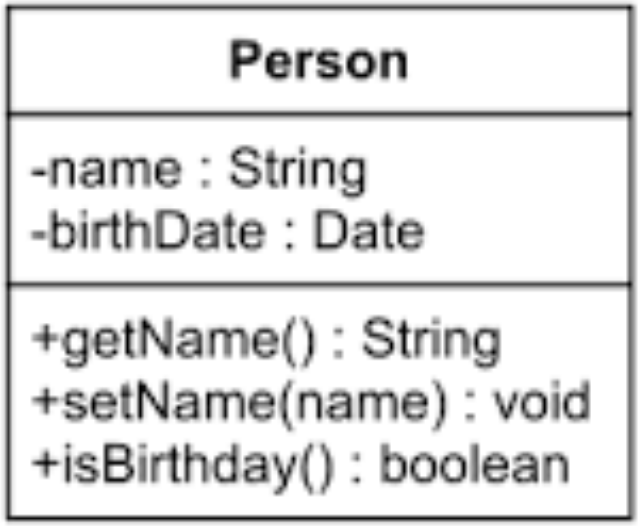
~ default

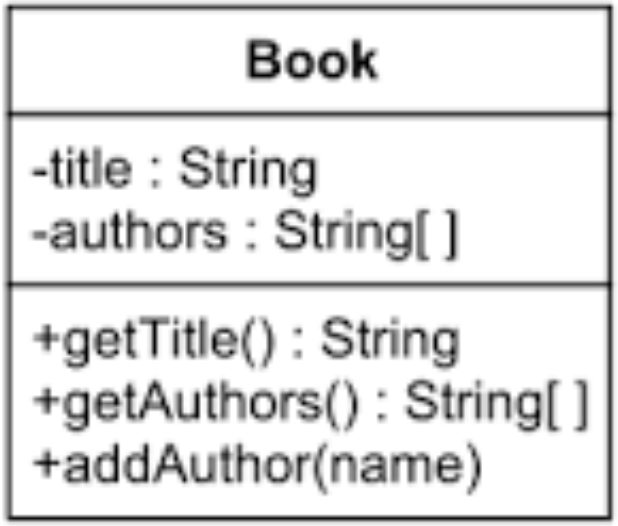


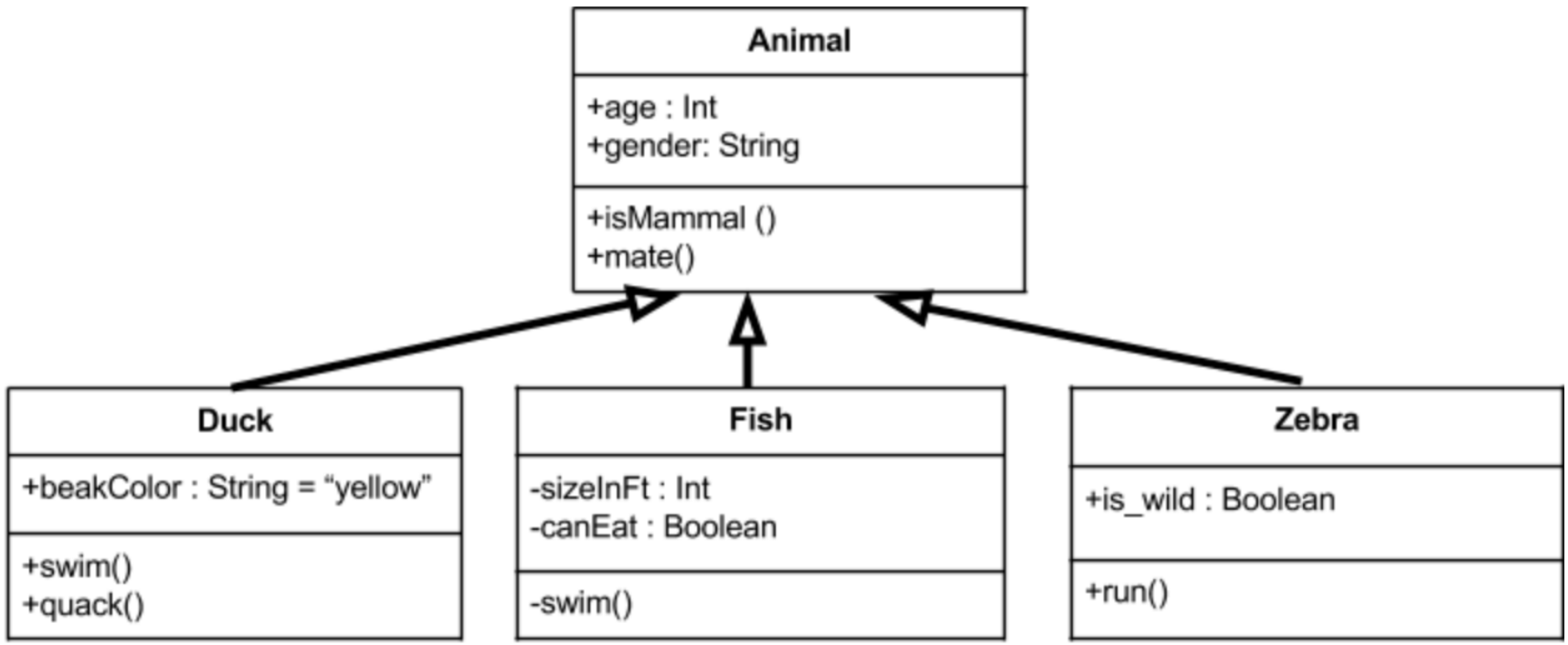
Class

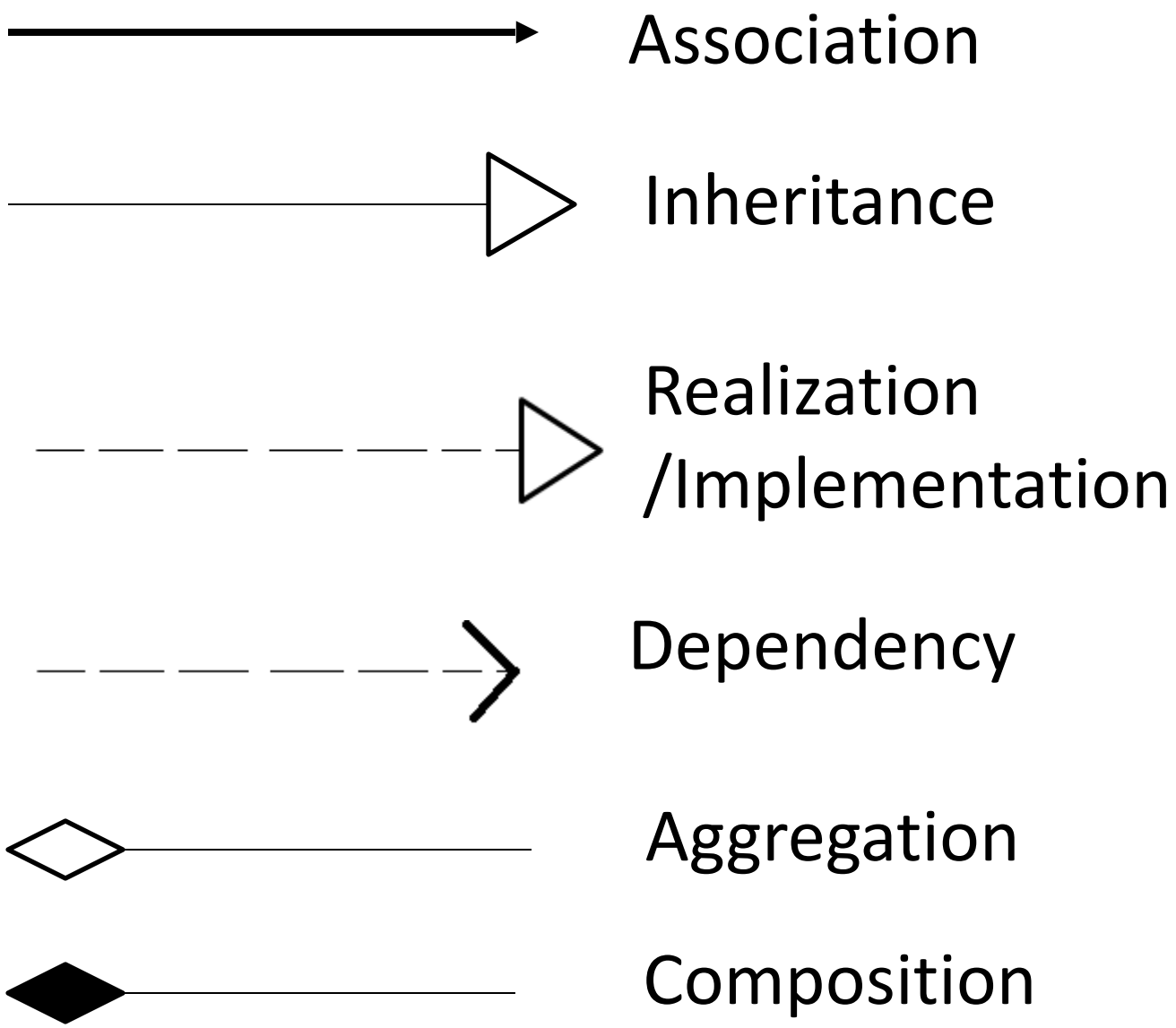
```
class Bank{

   private :

        int cash ;

  public  :

        float getCash() ;
       void  setCash( float tmp_cash) ;

 };
```

Data Encapsulation






Animal
+age : Int
+gender: String
+isMammal ()
+mate()
Duck
+beakColor : String = “yellow”
+swim()
+quack()
Fish
-sizeInFt : Int
-canEat : Boolean
-swim()
Zebra
+is_wild : Boolean
+run()


Association
Inheritance
Realization
/Implementation
Dependency
Aggregation
Composition

Encapsulation

Encapsulation is accomplished when each object maintains a private state, inside a class. Other objects can not access this state directly, instead, they can only invoke a list of public functions. The object manages its own state via these functions and no other class can alter it unless explicitly allowed. In order to communicate with the object, you will need to utilize the methods provided. One way I like to think of encapsulation is by using the example of people and their dogs. If we want to apply encapsulation, we do so by encapsulating all “dog” logic into a Dog class. The “state” of the dog is in the private variables playful, hungry and energy and each of these variables has their respective fields.

There is also a private method: bark(). The dog class can call this whenever it wants, and the other classes can not tell the dog when to bark. There are also public methods such as sleep(), play() and eat() that are available to other classes. Each of these functions modifies the internal state of the Dog class and may invoke bark(), when this happens the private state and public methods are bonded.

Abstraction

Abstraction is an extension of encapsulation. It is the process of selecting data from a larger pool to show only the relevant details to the object. Suppose you want to create a dating application and you are asked to collect all the information about your users. You might receive the following data from your user: Full name, address, phone number, favorite food, favorite movie, hobbies, tax information, social security number, credit score. This amount of data is great however not all of it is required to create a dating profile. You only need to select the information that is pertinent to your dating application from that pool. Data like Full name, favorite food, favorite movie, and hobbies make sense for a dating application. The process of fetching/removing/selecting the user information from a larger pool is referred to as Abstraction. One of the advantages of Abstraction is being able to apply the same information you used for the dating application to other applications with little or no modification.

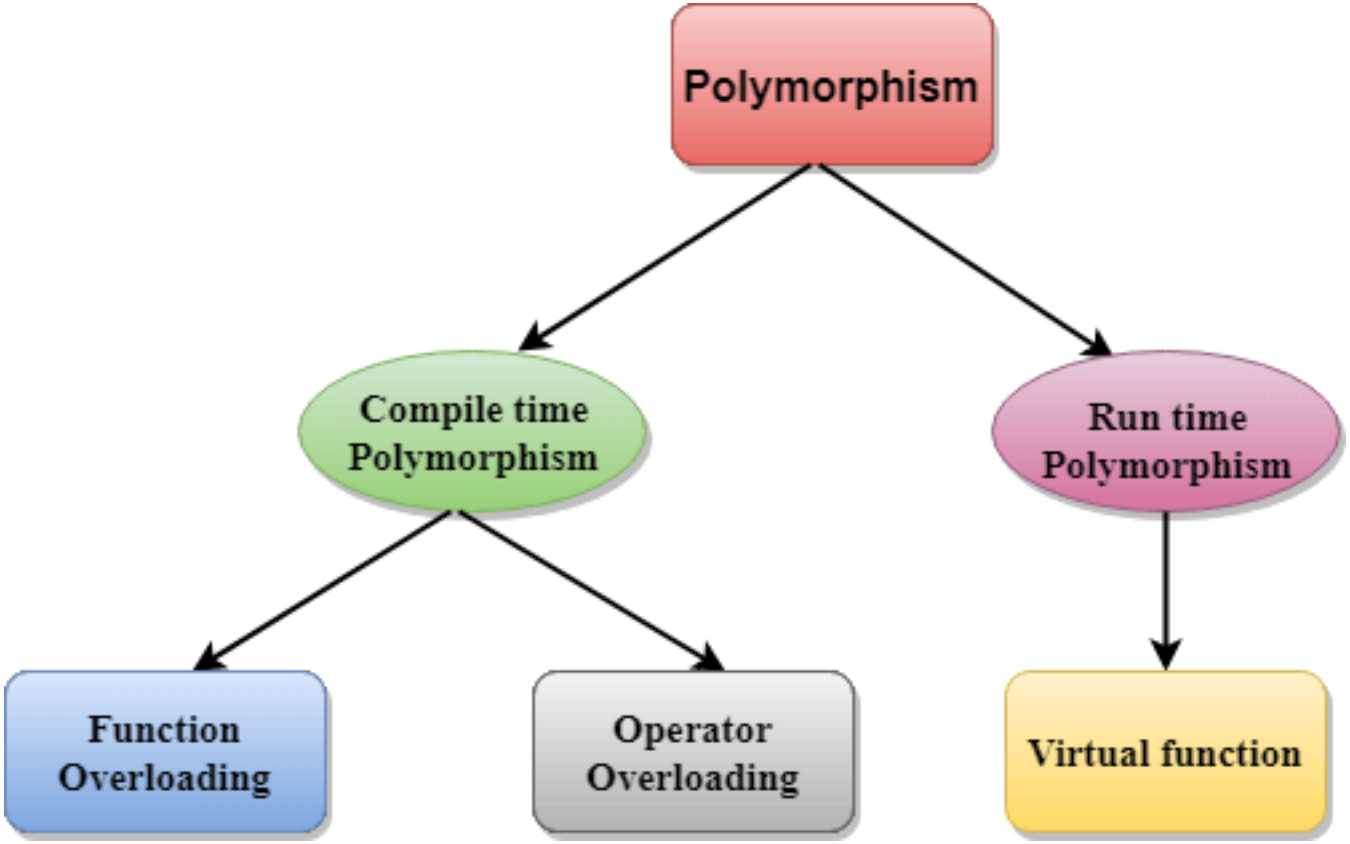
Inheritance

Inheritance is the ability of one object to acquire some/all properties of another object. For example, a child inherits the traits of his/her parents. With inheritance, reusability is a major advantage. You can reuse the fields and methods of the existing class. In Java, there are various types of inheritances: single, multiple, multilevel, hierarchical, and hybrid. For example, Apple is a fruit so assume that we have a class Fruit and a subclass of it named Apple. Our Apple acquires the properties of the Fruit class. Other classifications could be grape, pear, and mango, etc. Fruit defines a class of foods that are mature ovaries of a plant, fleshy, contains a large seed within or numerous tiny seeds. Apple the sub-class acquires these properties from Fruit and has some unique properties, which are different from other sub-classes of Fruit such as

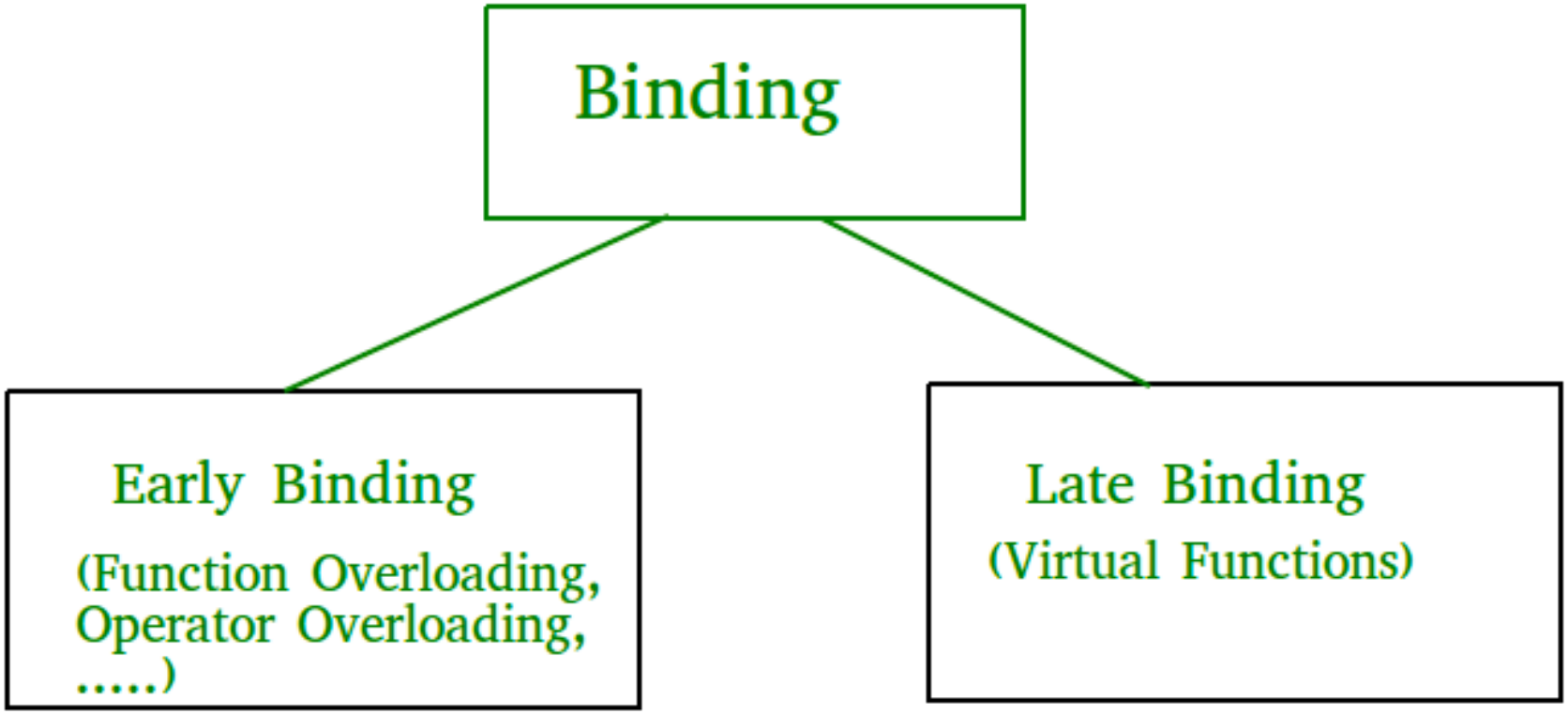
Polymorphism

Polymorphism gives us a way to use a class exactly like its parent so there is no confusion with mixing types. This being said, each child sub-class keeps its own functions/methods as they are. If we had a superclass called Mammal that has a method called mammalSound(). The sub-classes of Mammals could be Dogs, whales, elephants, and horses. Each of these would have their own iteration of a mammal sound (dog-barks, whale-clicks).

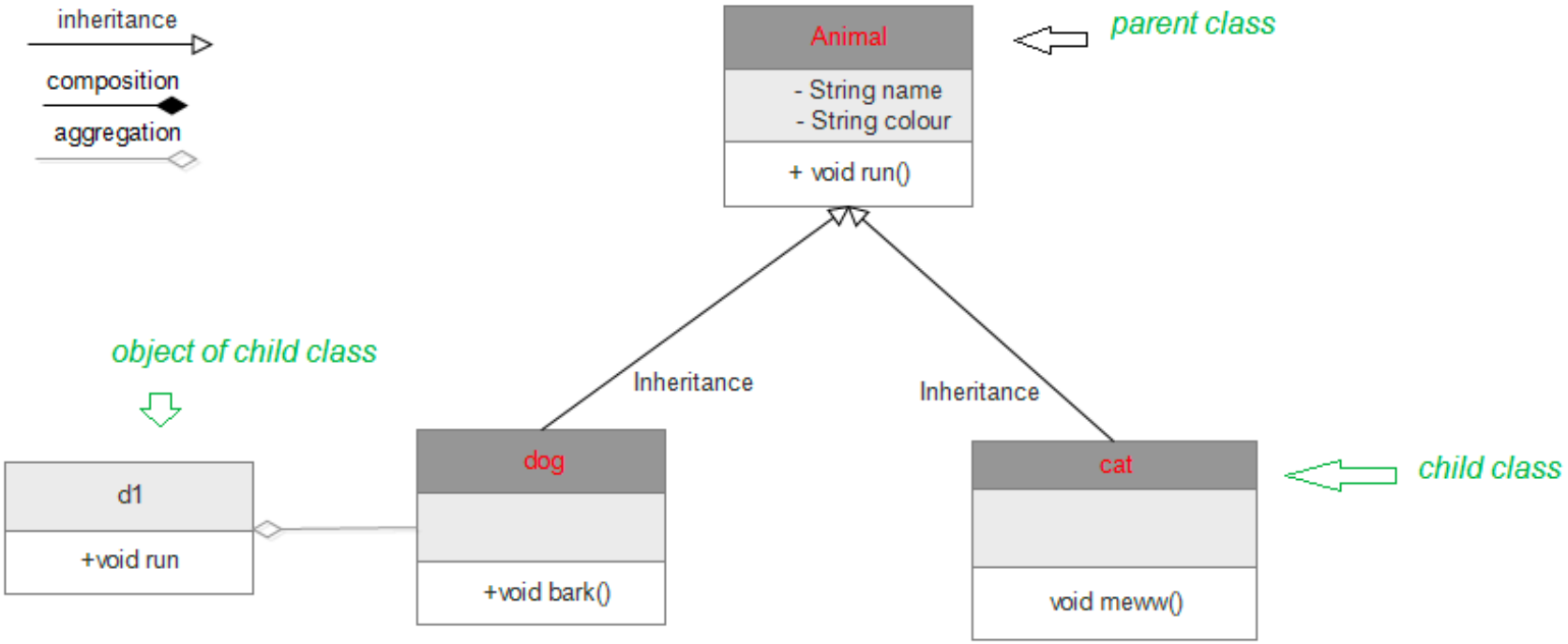
Polymorphism: Compile time ,Run time



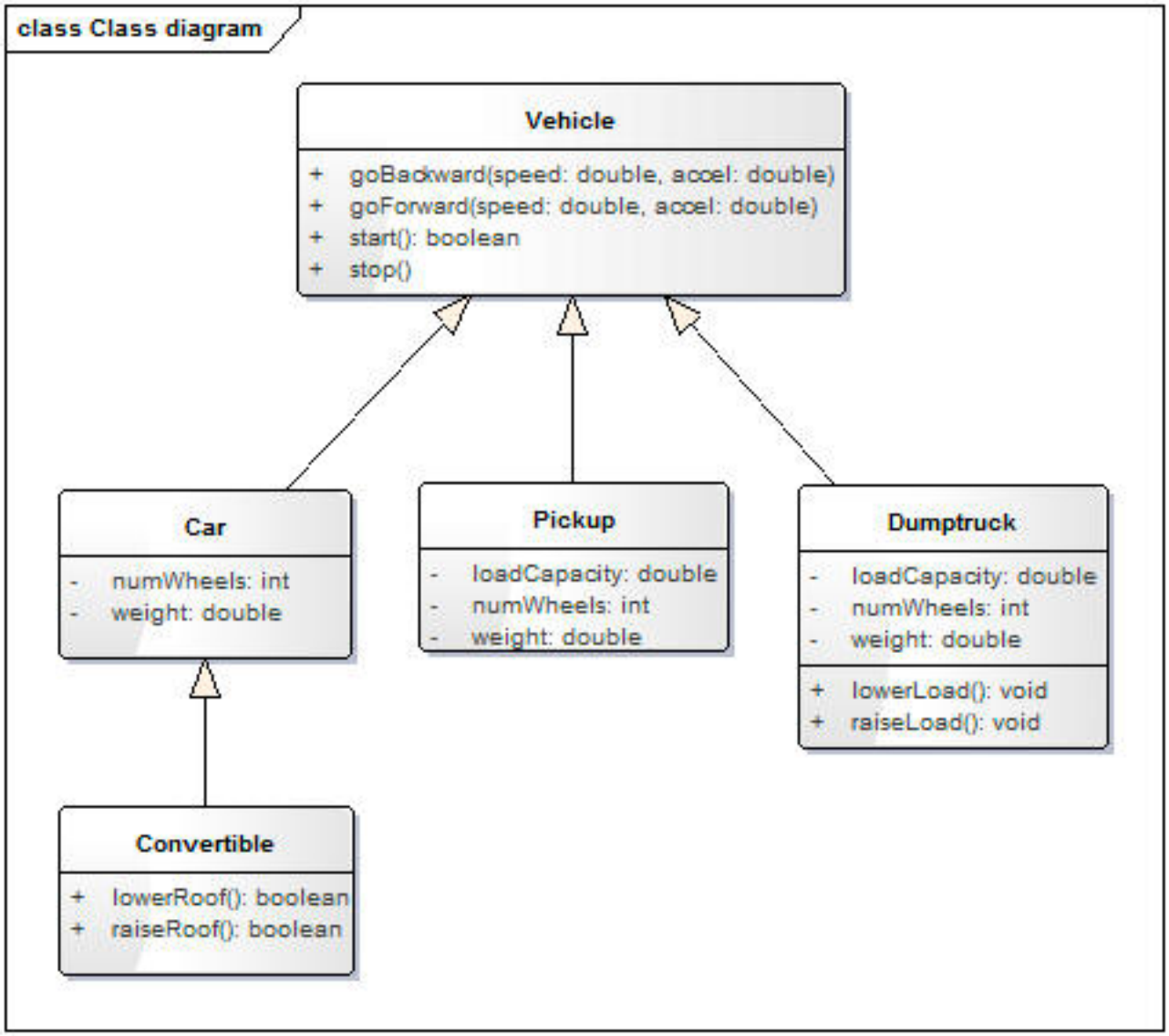
Polymorphism Early v/s late binding



Inheritance



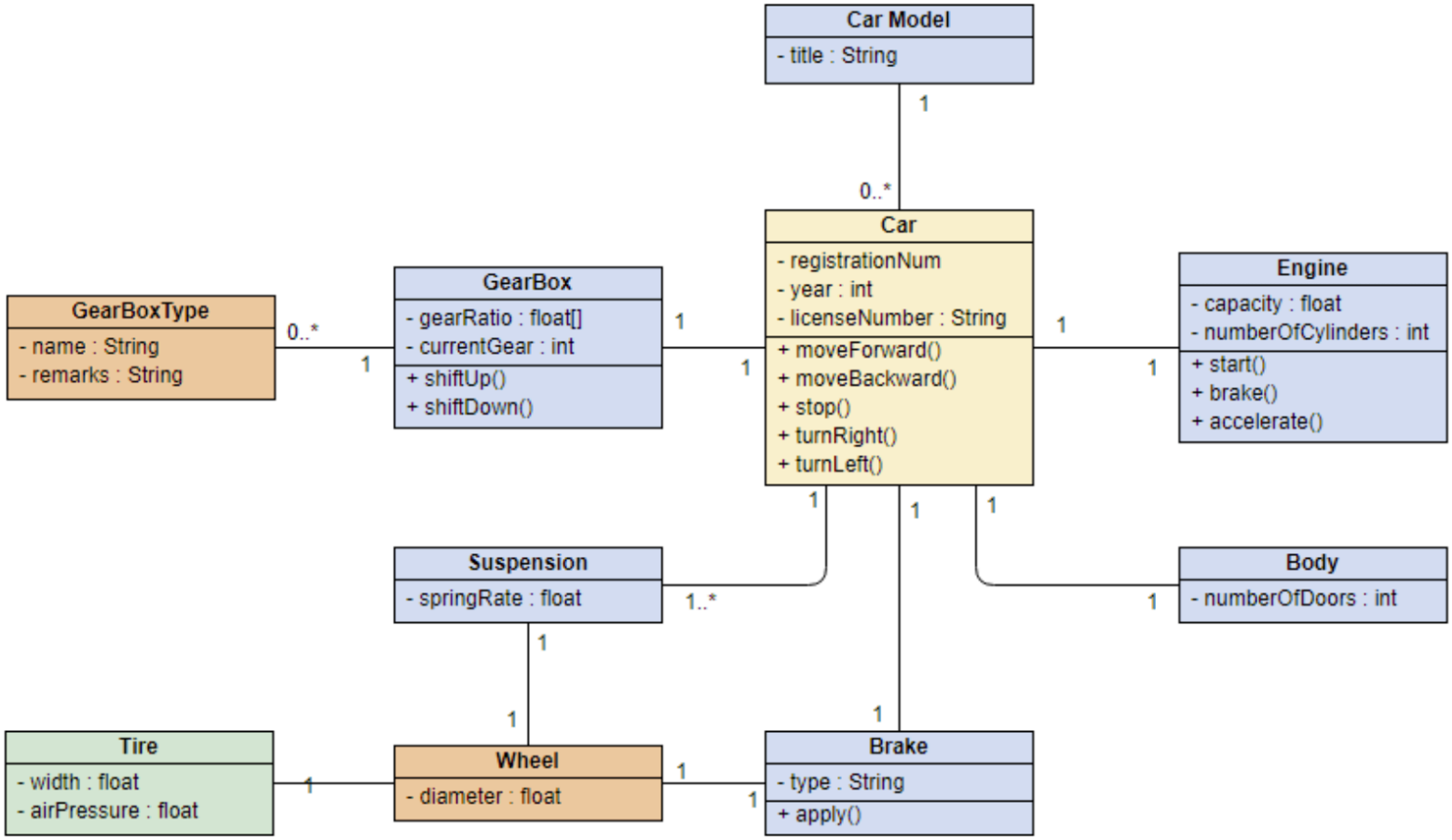
Inheritance



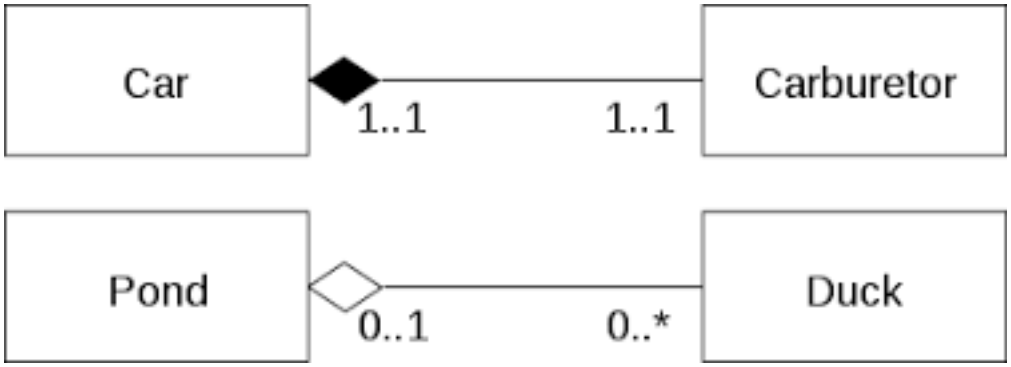
Class

```
class Bank{

   private :

        int cash ;

  public  :

        float getCash() ;
       void  setCash( float tmp_cash) ;

 };
```

Composed Car components



Composition vs Aggregation

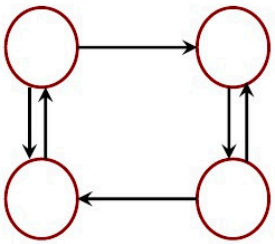


Class diagram

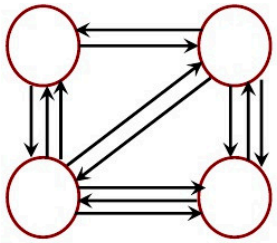
Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only *n* (where *n* > 1)
0..n	Zero to *n* (where *n* > 1)
1..n	One to *n* (where *n* > 1)

Class diagram

Coupling



Loosely coupled:
some dependencies



Highly coupled:
many dependencies

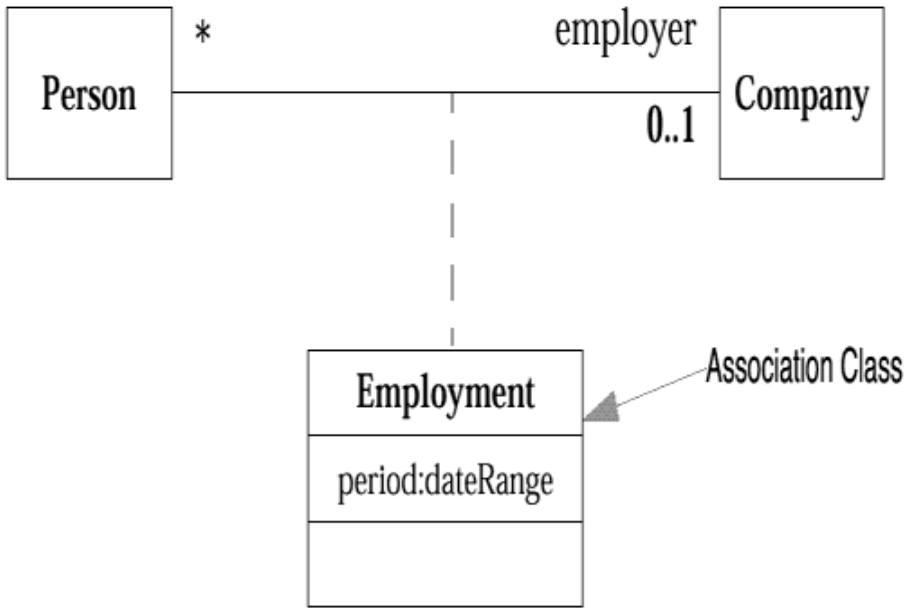
Inheritance

Aggregation

Composition

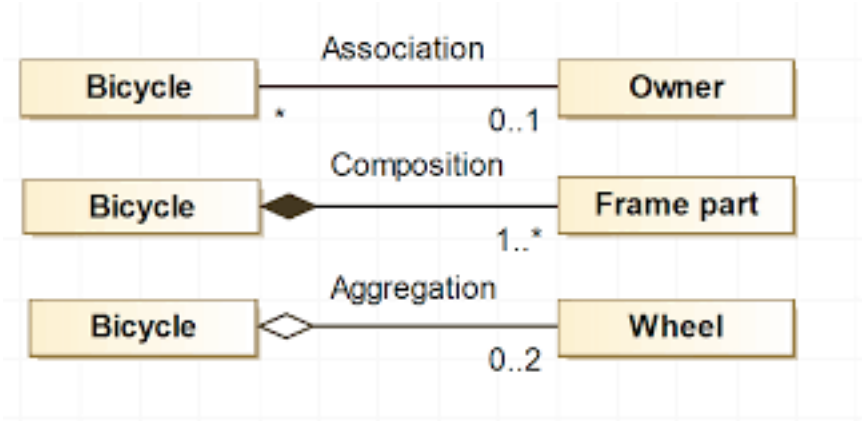
Class diagram

Association

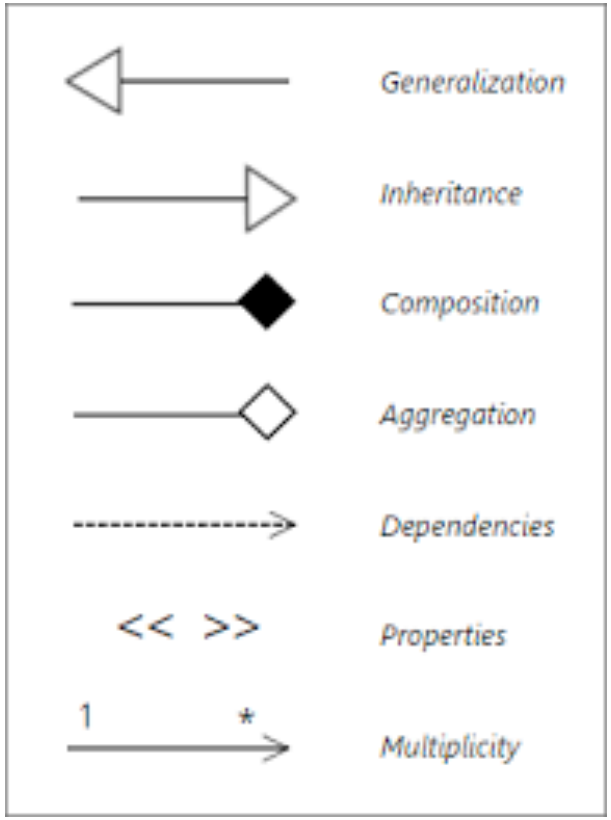


UML

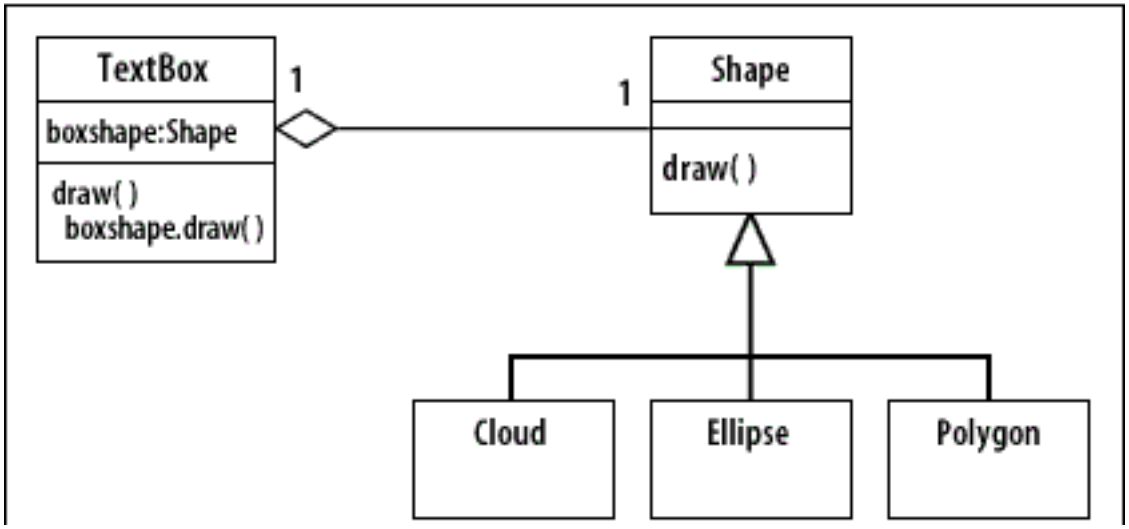
- Association Composition Aggregation



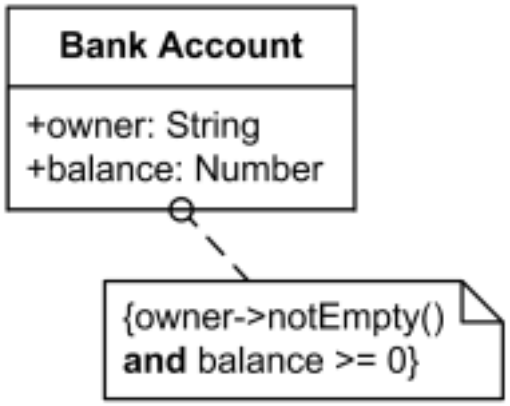
UML Various arrows



UML delegate

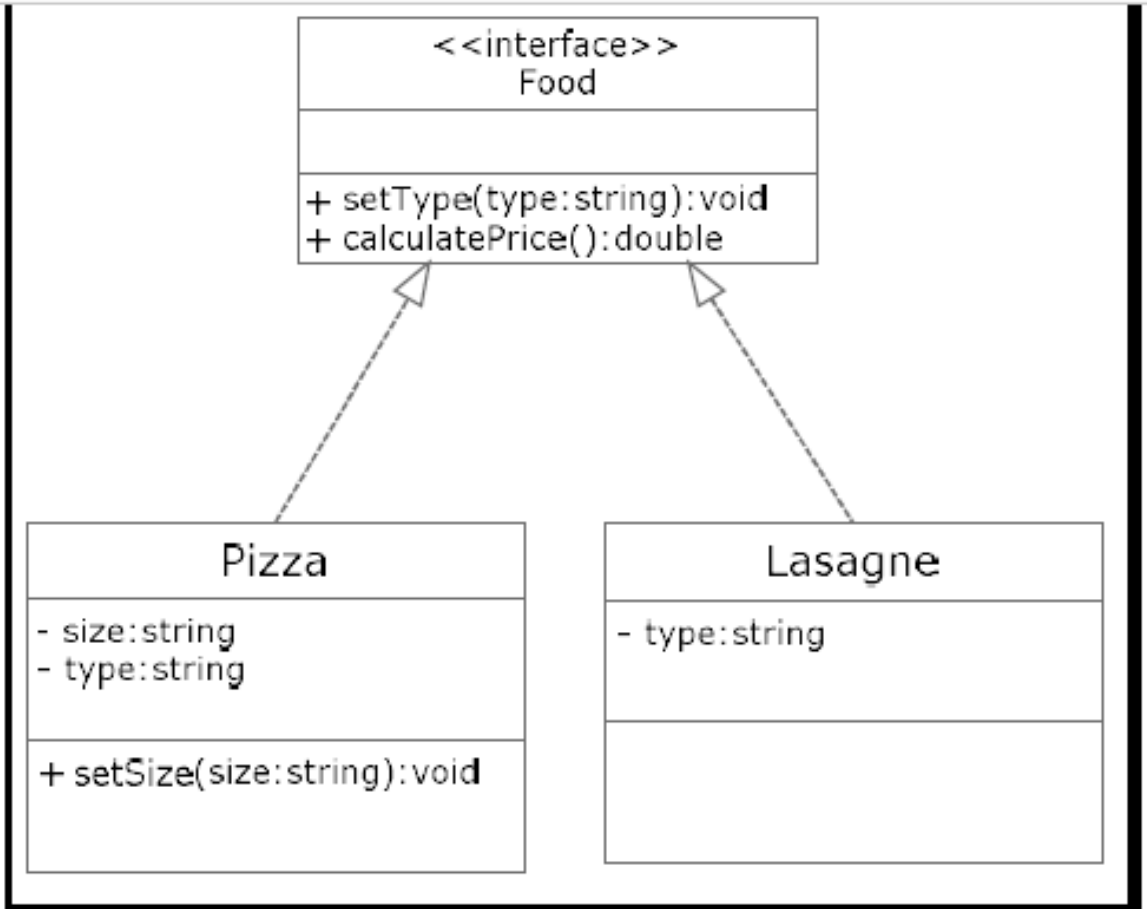


Pre constraints class

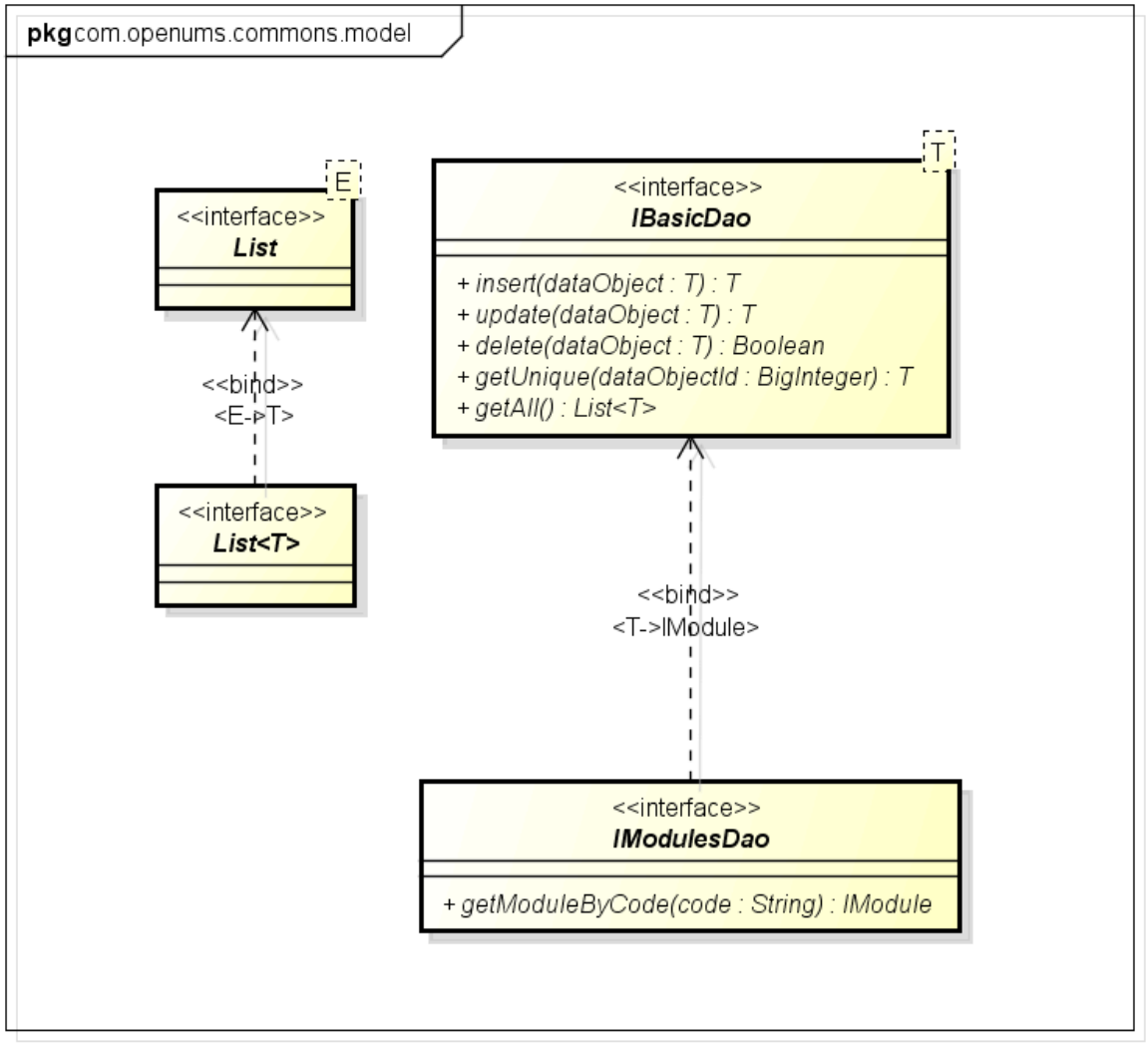


Should have enough funds

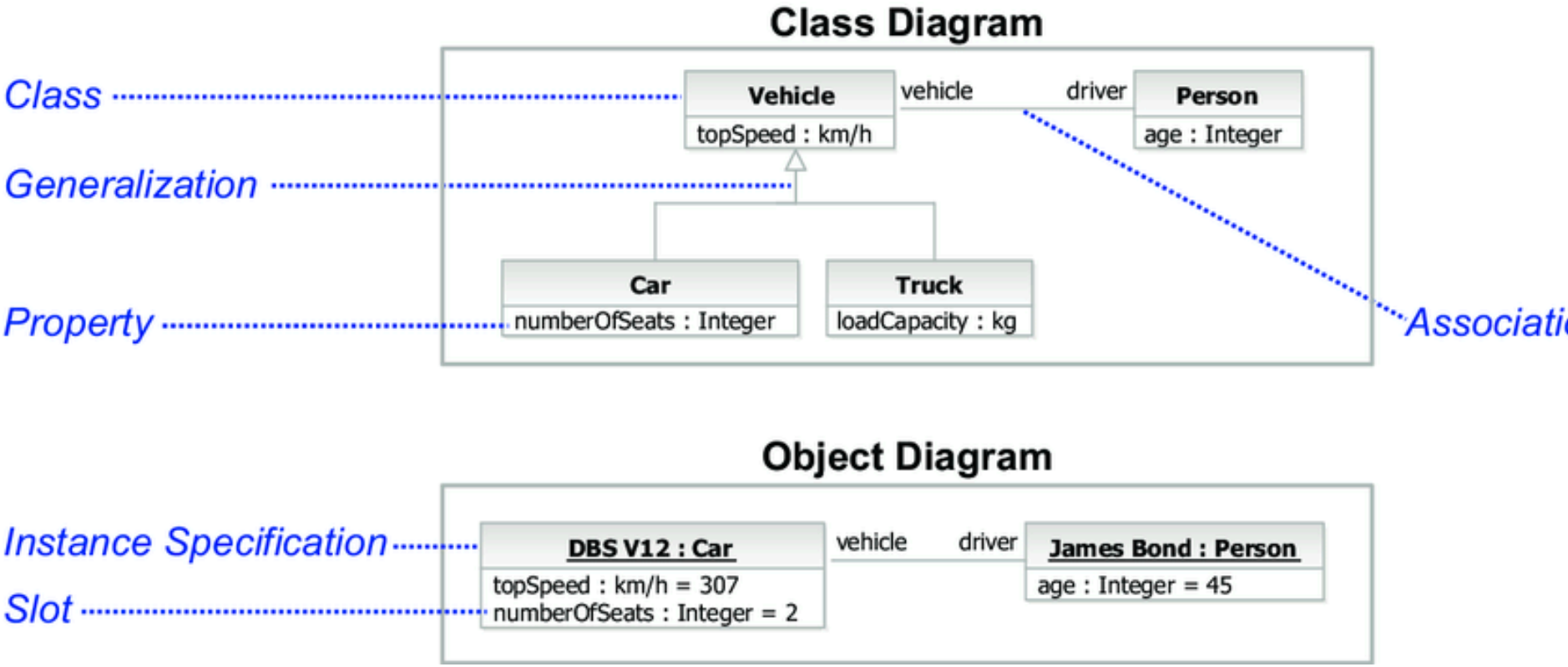
Abstract Class diagram



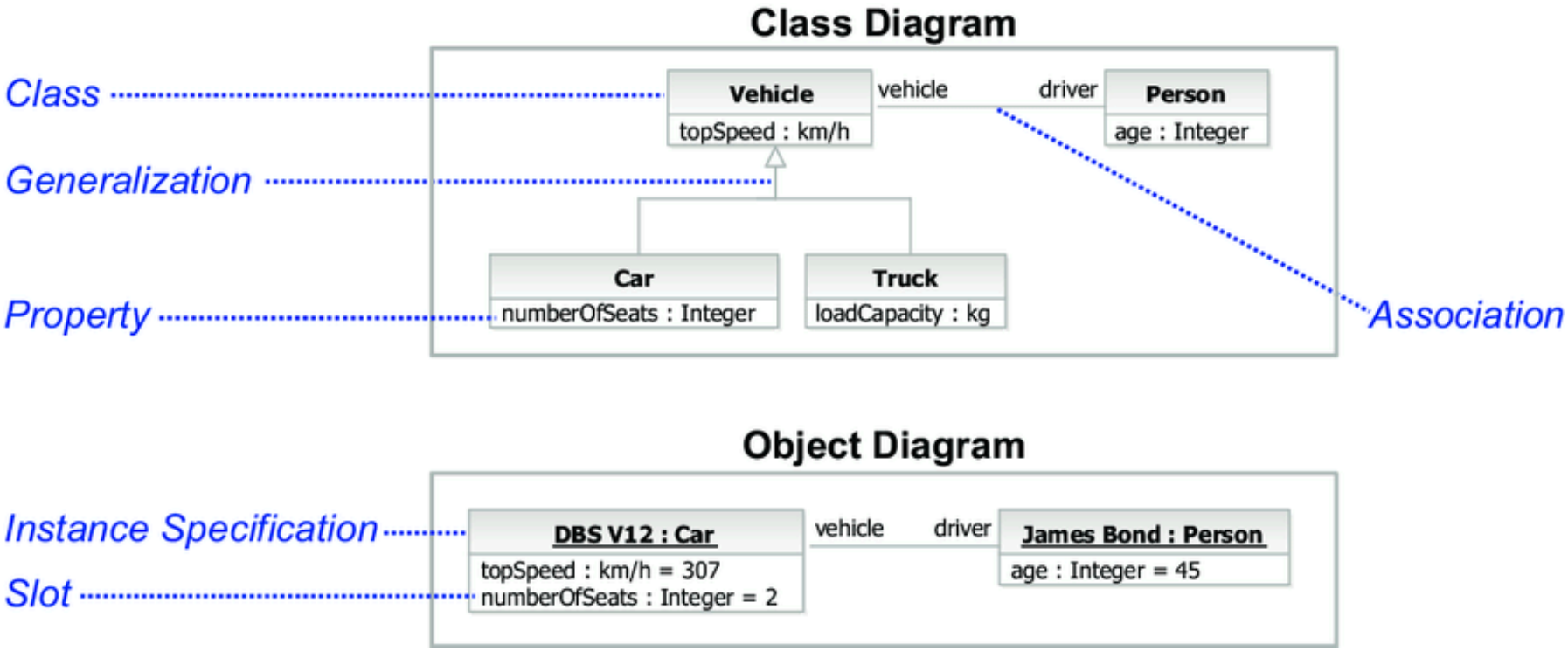
Interface Class diagram



object diagram



Class diagram



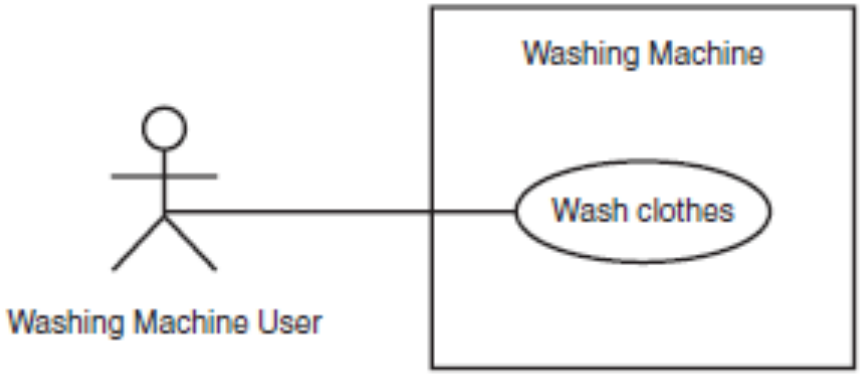
Class diagram

Class diagram

Class diagram

Use-Case Diagram

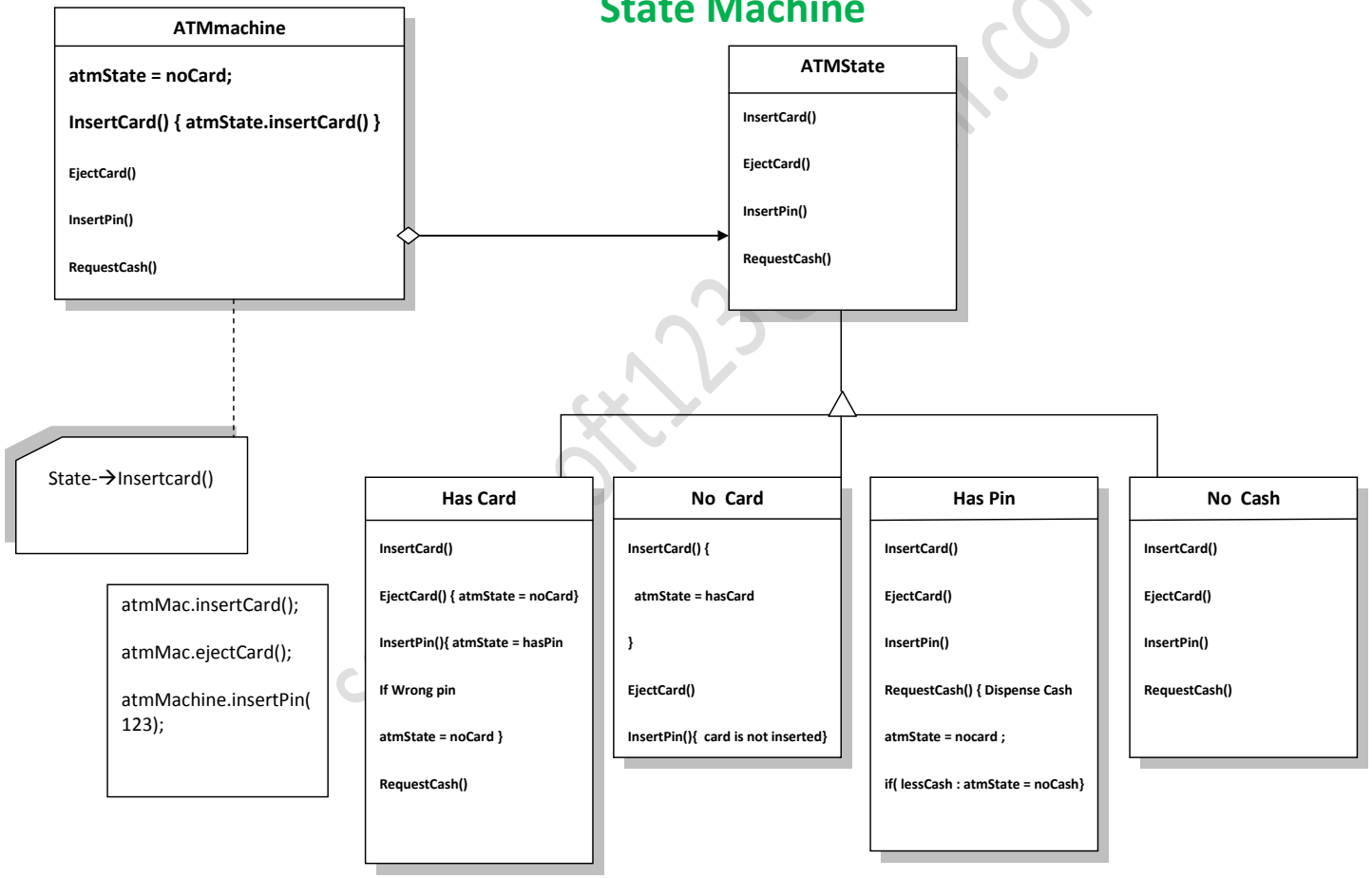
- A use case diagram shows a set of use cases and actors (a special kind of class) and their relationships.
- Use case diagrams address the static use case view of a system. These diagrams are especially important in organising and modelling the behaviours of a system.



Class diagram

State

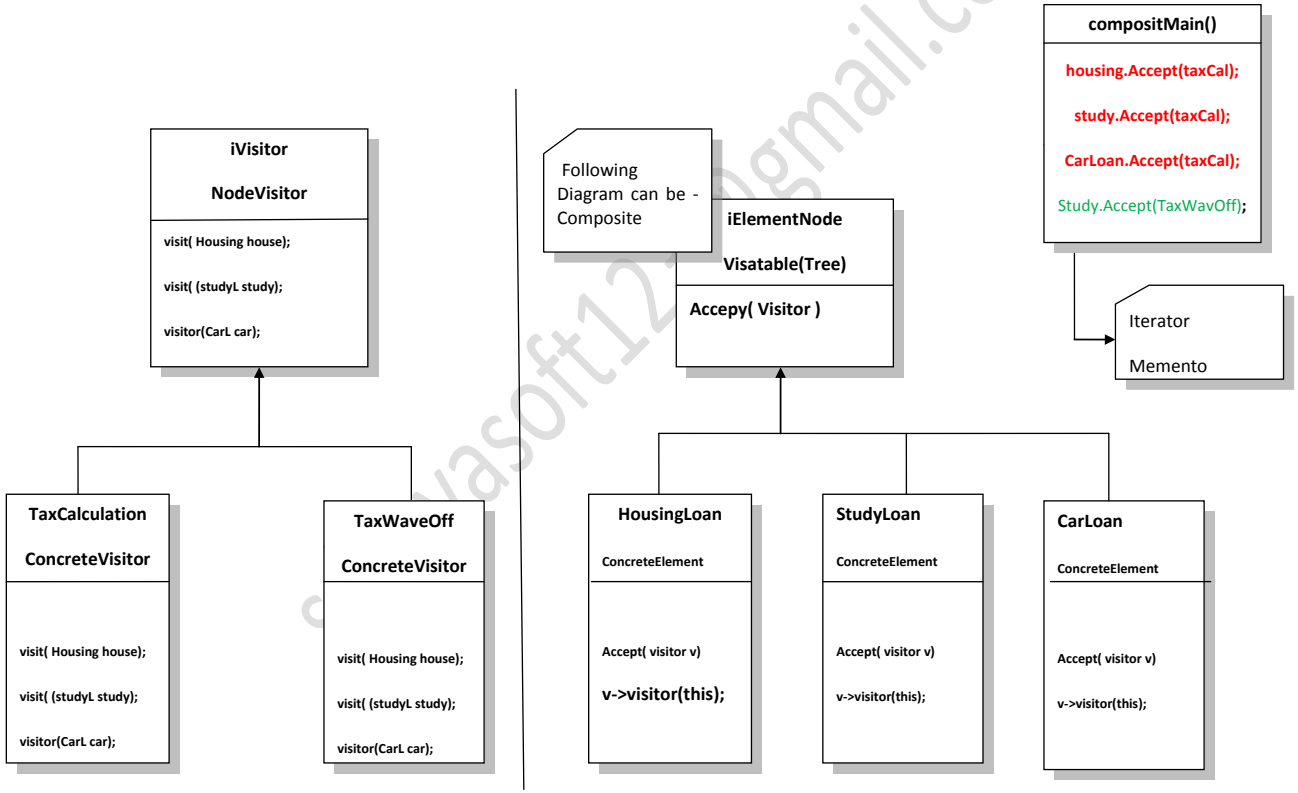
State Machine



Visitor

Design Patterns

Visitor



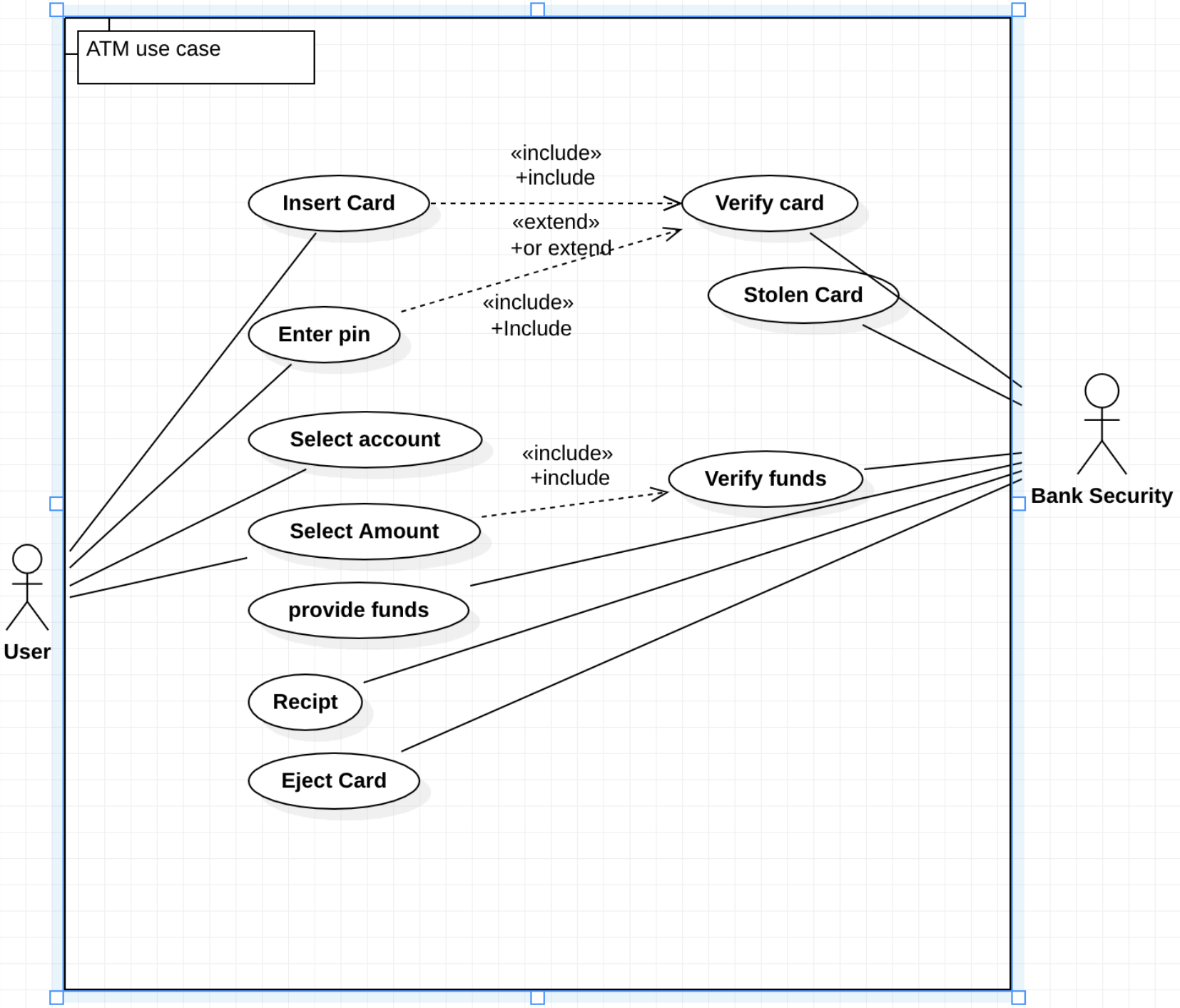
/

Use case Planning

Predictive planning

If you can list all requirements in advance

Adaptive planing - Agile development


ATM use case
«include»
+include
Insert Card
Verify card
«extend»
+or extend
Stolen Card
«include»
+Include
Enter pin
Select account
«include»
+include
Verify funds
Select Amount
provide funds
Recipt
Eject Card
User
Bank Security

Use case

Trigger

machine receives a card

User enters a PIN

Actors

Customer

Bank Security

Use case

Pre conditions

1. Secure connection to bank
2. bank has cash

Goals

1. Secure client accounts
2. Provide customer with funds

Class diagram

Use case

Failed conclusions

1. Invalid card
2. Invalid PIN
3. Customer insufficient funds
4. ATM insufficient funds
5. Over daily limit
6. Stolen card

Extensions

1. if PIN is invalid 3 times S
2. If card marked as stolen Swallow card

Use Case

Requirements

1. Customer inserts card

1A. Card is invalid

1B. Eject card

2.Card is valid

3. Customer enters the PIN

3A. Pin is Invalid

3B. PIN is invalid 3 times

3C. Card marked as stolen

4. PIN is valid

5. Account is selected

6. Amount is selected

Continued… .

Use case

6A. Over daily Maximum

6B. Over Account funds Available

6C Over funds in machine

6D Ask for new Amount

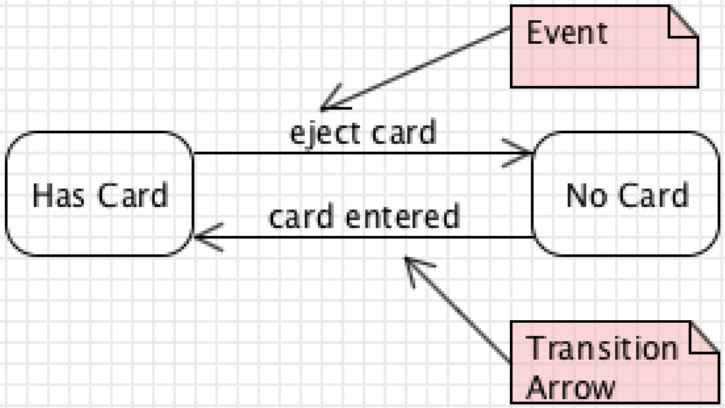
7 Provide money

8. Provide receipt

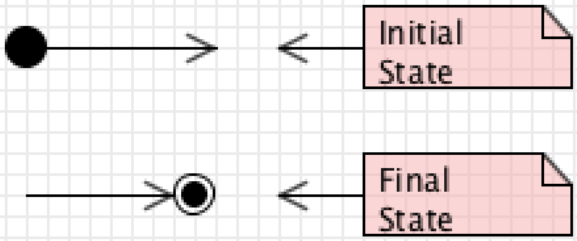
9. Eject Card

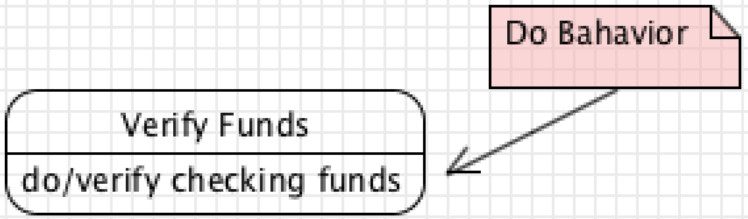
State machine diagram

STATE MACHINE DIAGRAMS

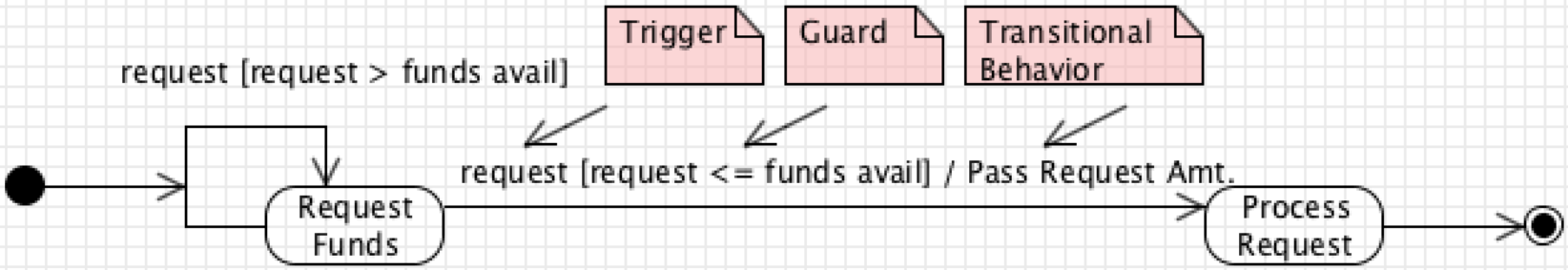


State / Trigger	Card Entered	Eject Card
No Card	Has Card	-
Has Card	-	No Card

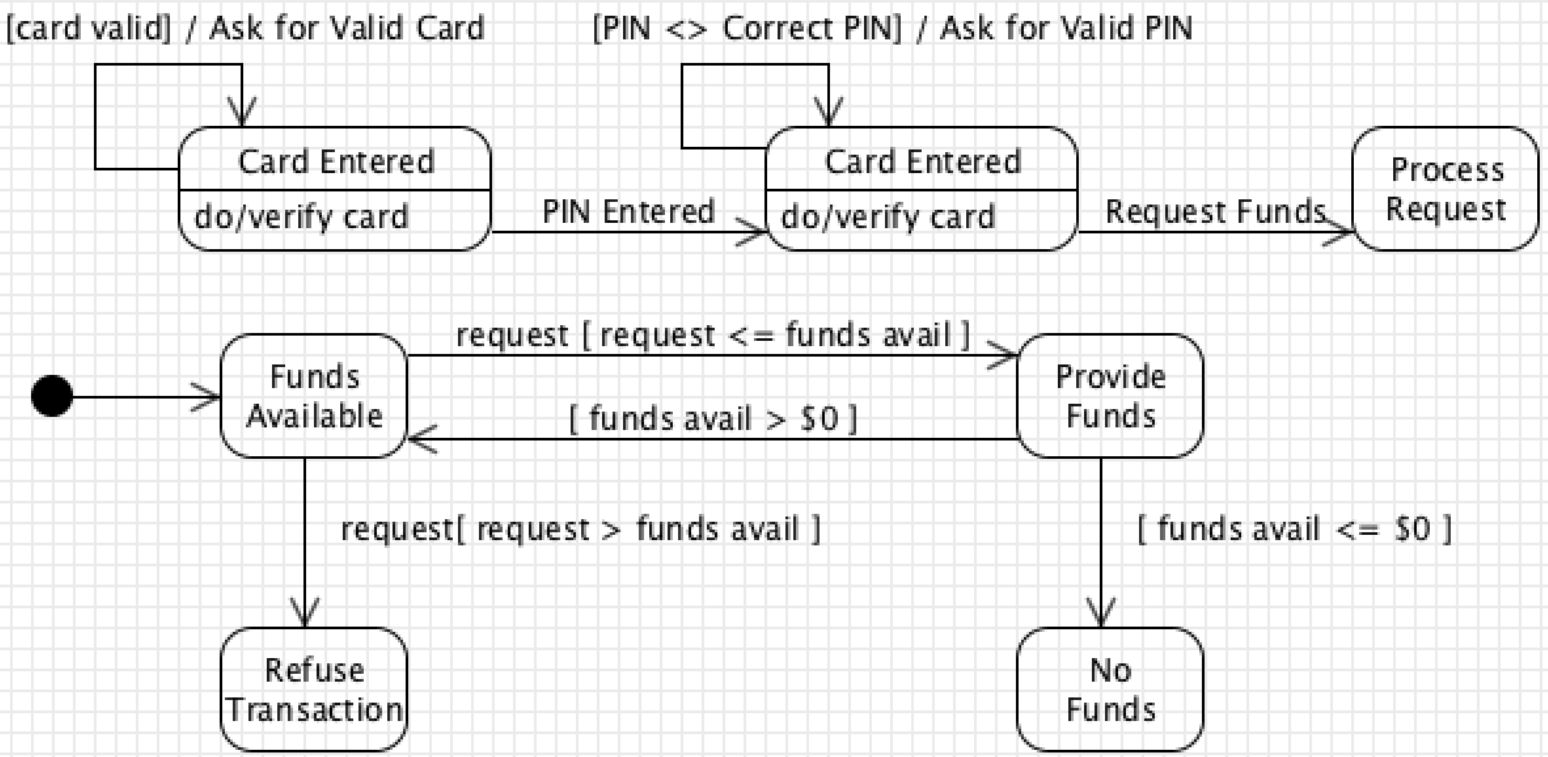




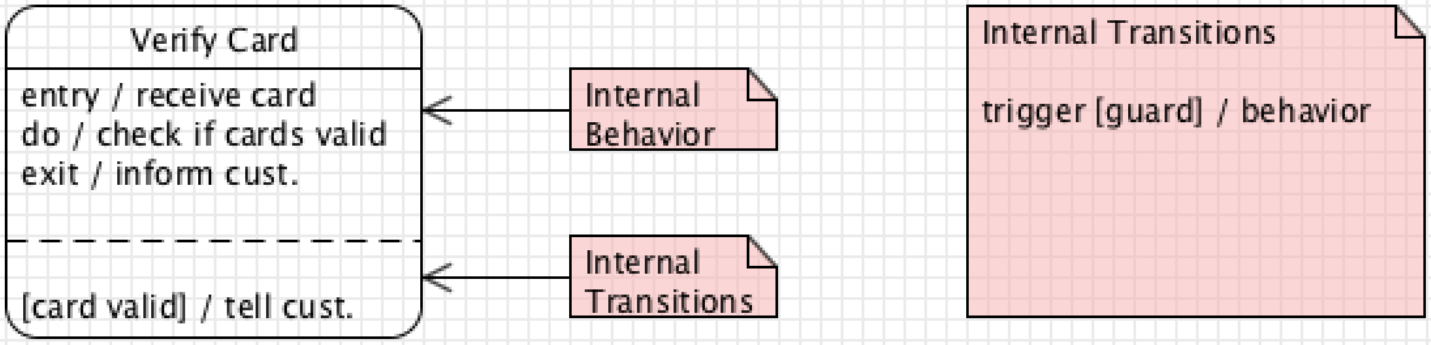
Transition Arrows



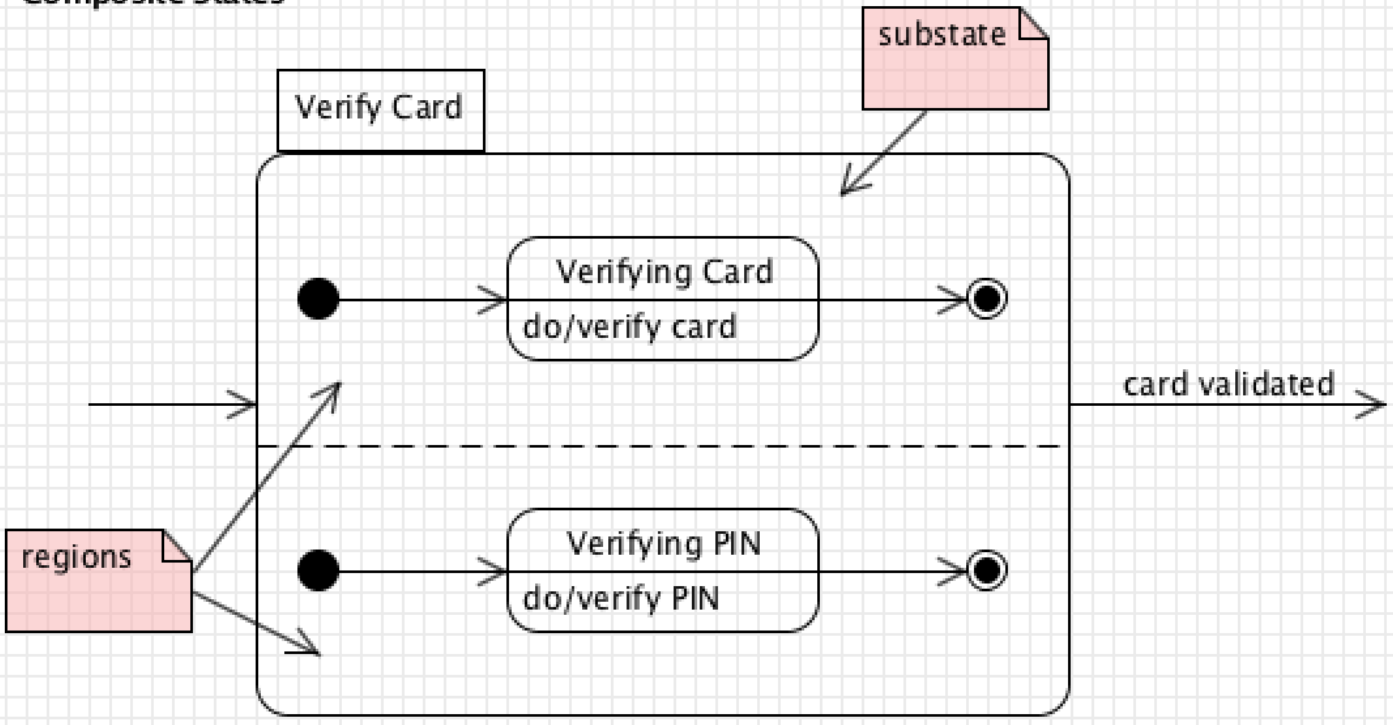
State Machine Diagrams



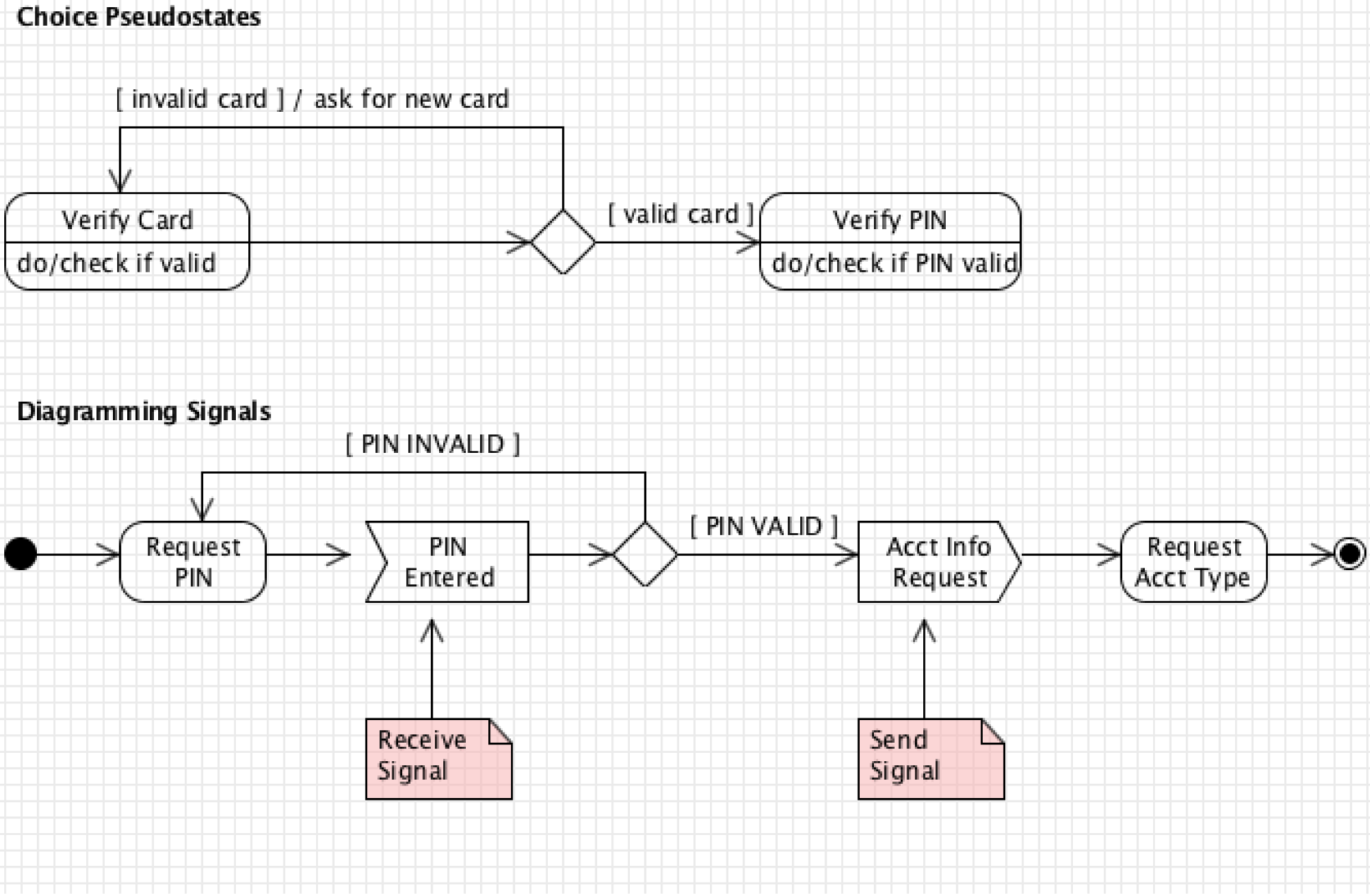
State Internal Behavior



Composite States

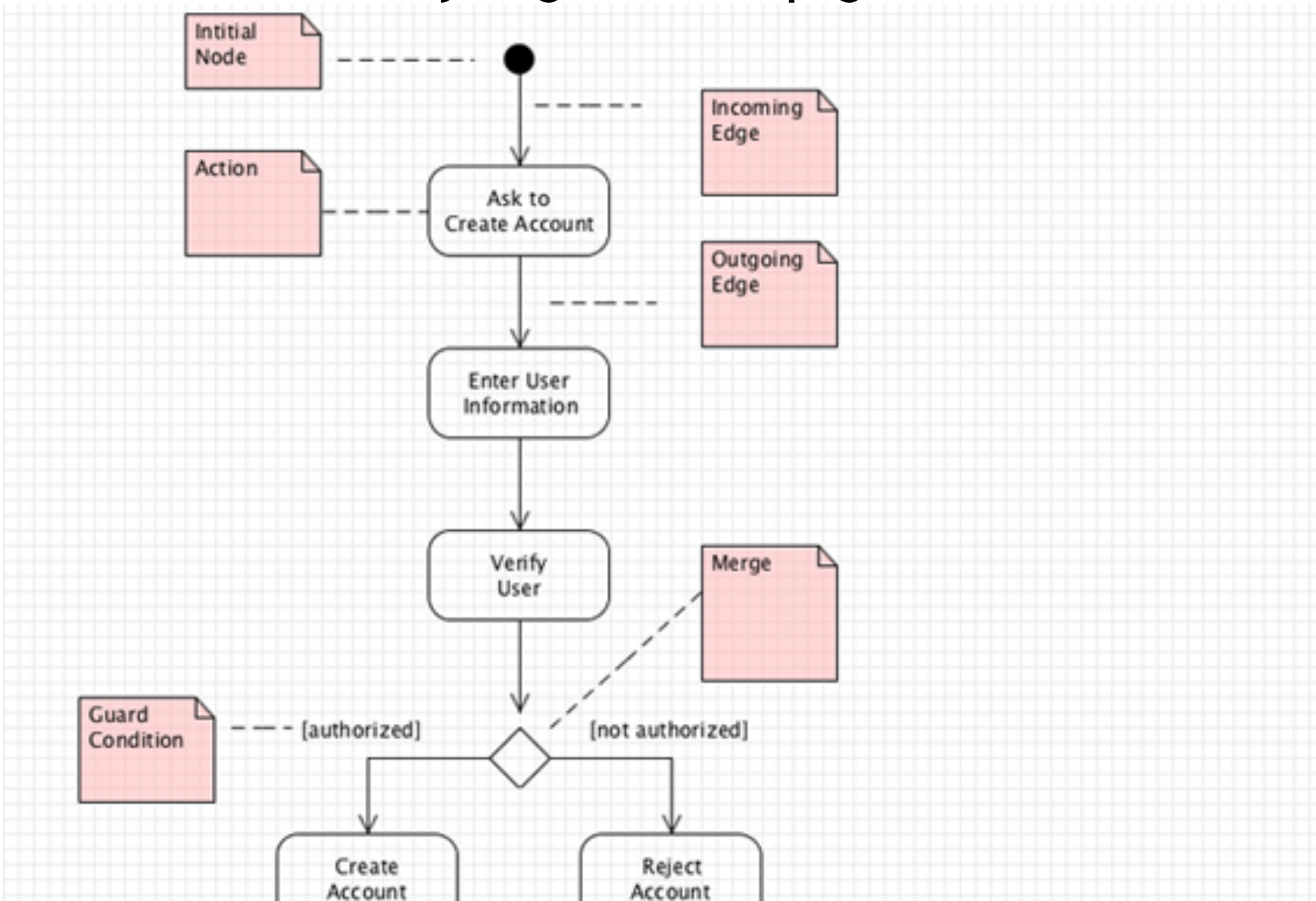


State machine diagram



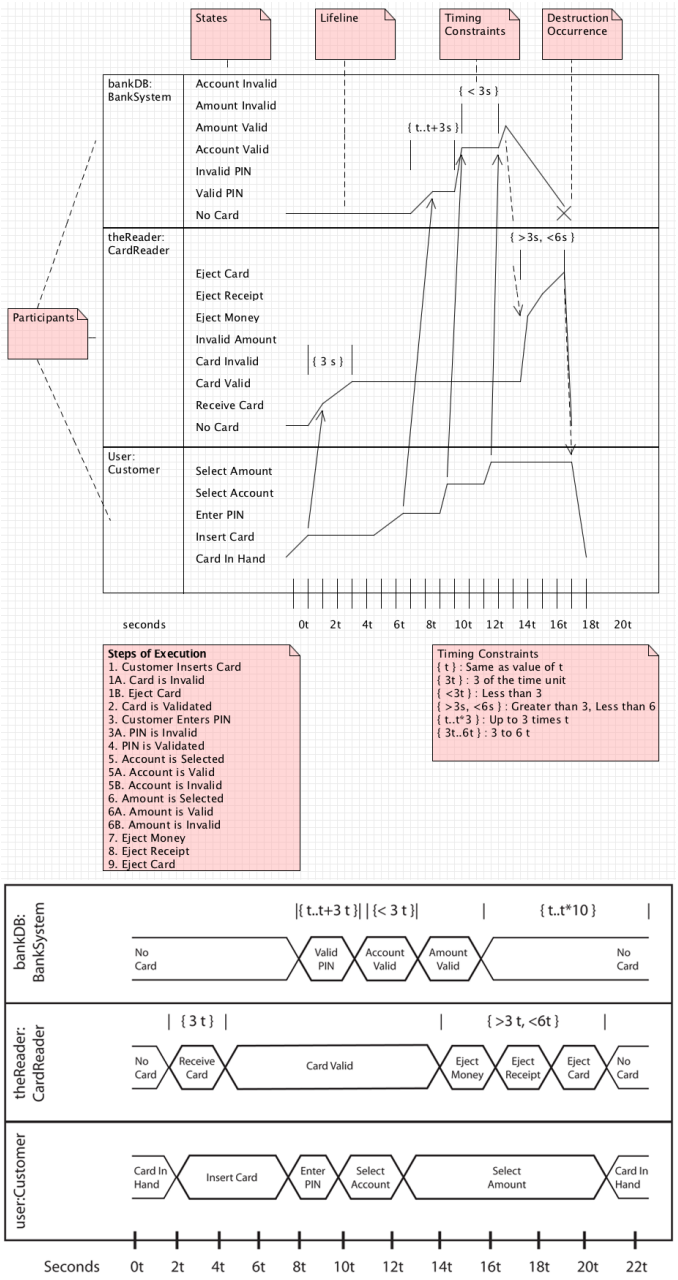
State machine diagram

Activity diagram - Use .png



Class diagram

Timing diagram



Timing diagram with states

1. Customer inserts card
2. Card is valid
3. Eject card
4. Card is valid
5. Customer enters PIN
6. PIN is invalid
7. PIN is validated
8. Account is selected
9. Amount is selected
10. Provide funds
11. Provide receipt
12. Eject Card

Timing diagram with states Three component

Customer

Card reader

bank system

Timing diagram with states

Customer

Card in hand

Insert card

Enter PIN

Select Account

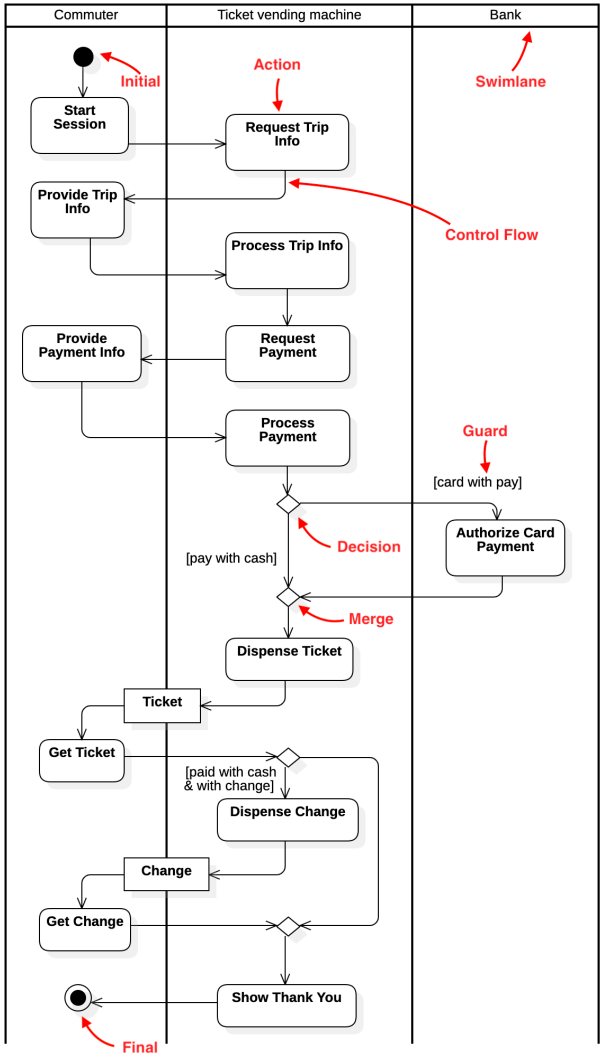
Select Amount

Timing diagram with states

Card reader

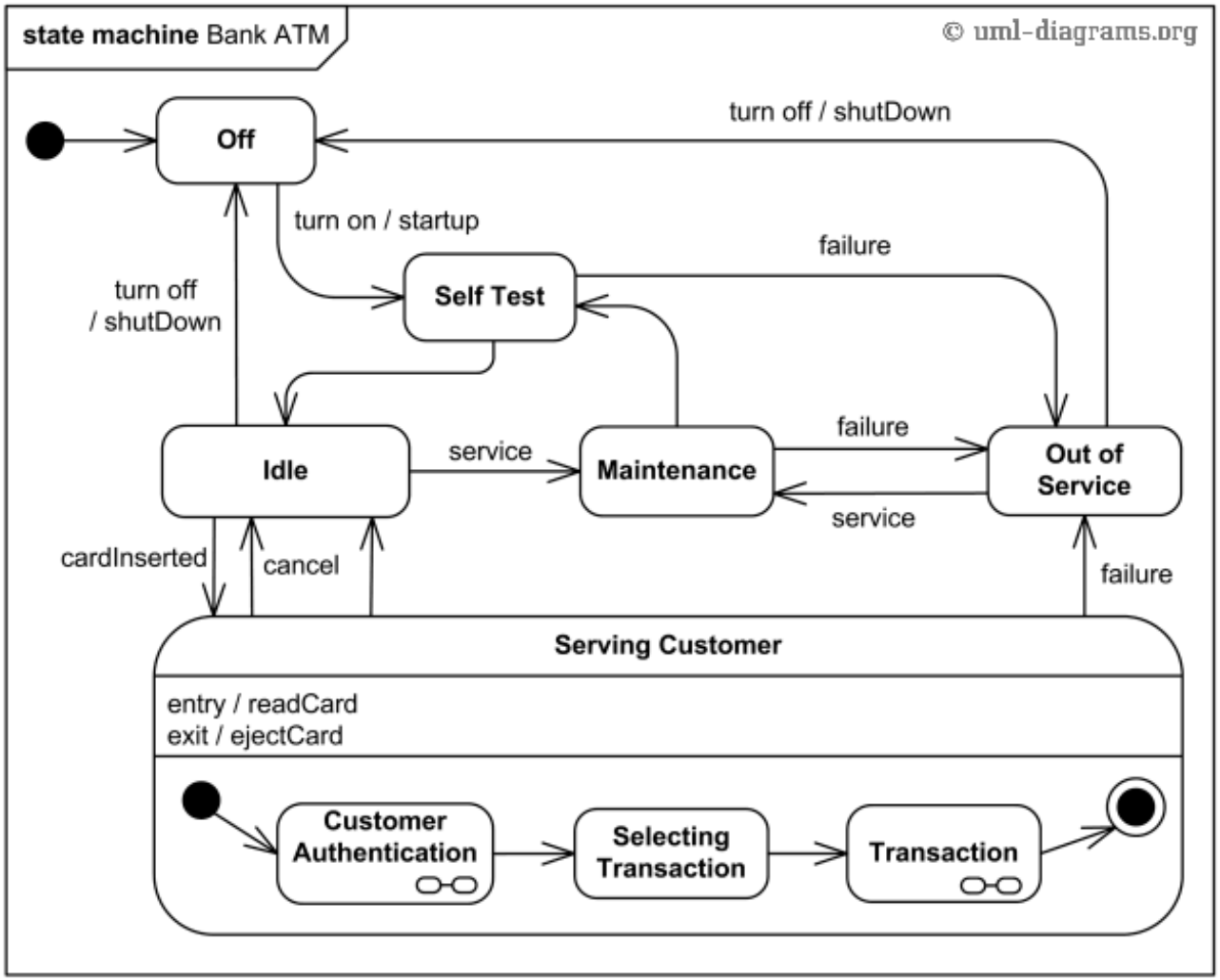
1. No Card
2. Received card
3. Card valid
4. Card invalid
5. Invalid Amount
6. Eject money
7. Eject receipt
8. Eject Card

Activity diagram

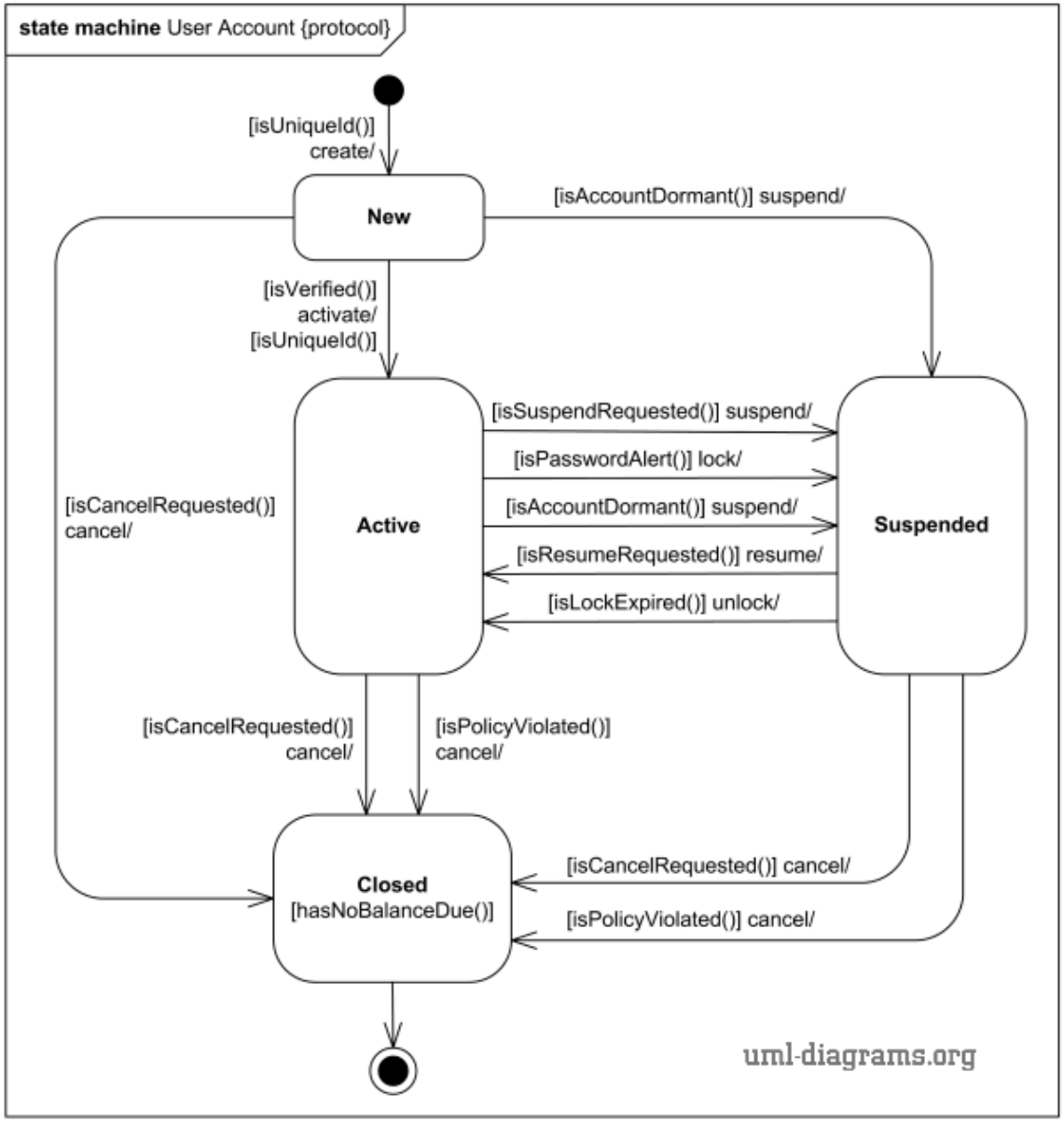


ATM

e diagram https://www.uml-diagrams.org/state-machine-diagrams.html

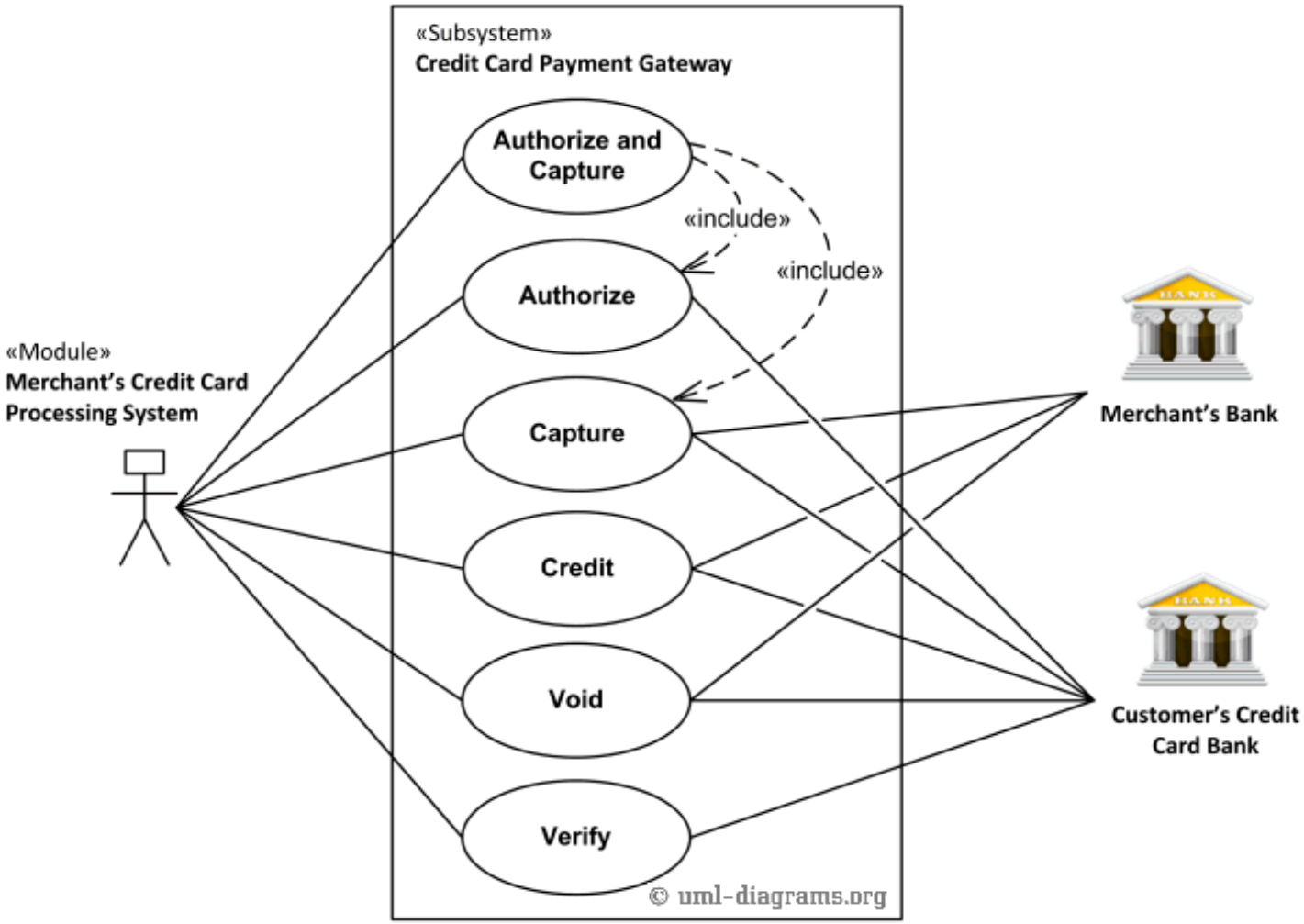


State diagram



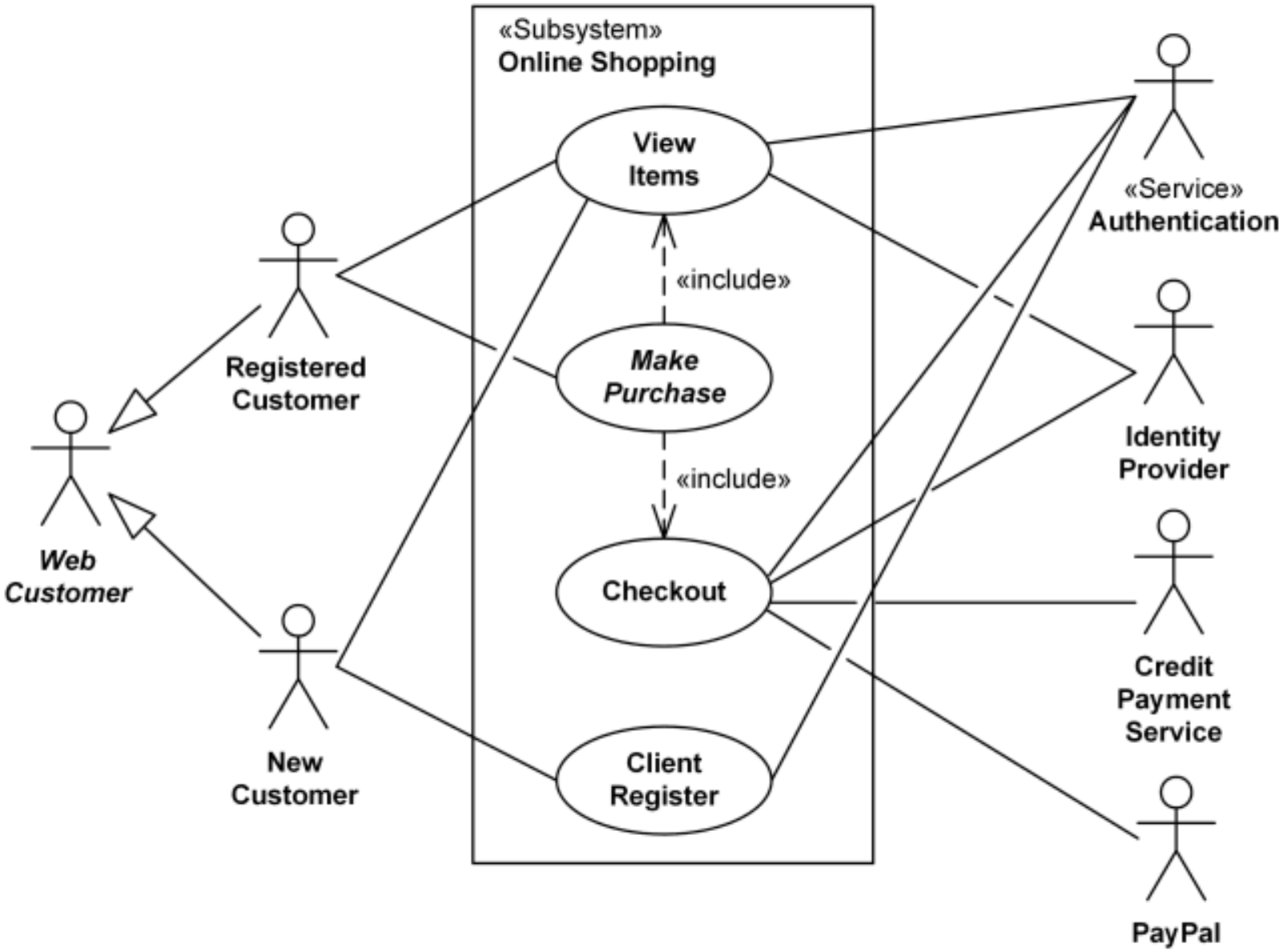
Use case diagram

Credit Cards Processing

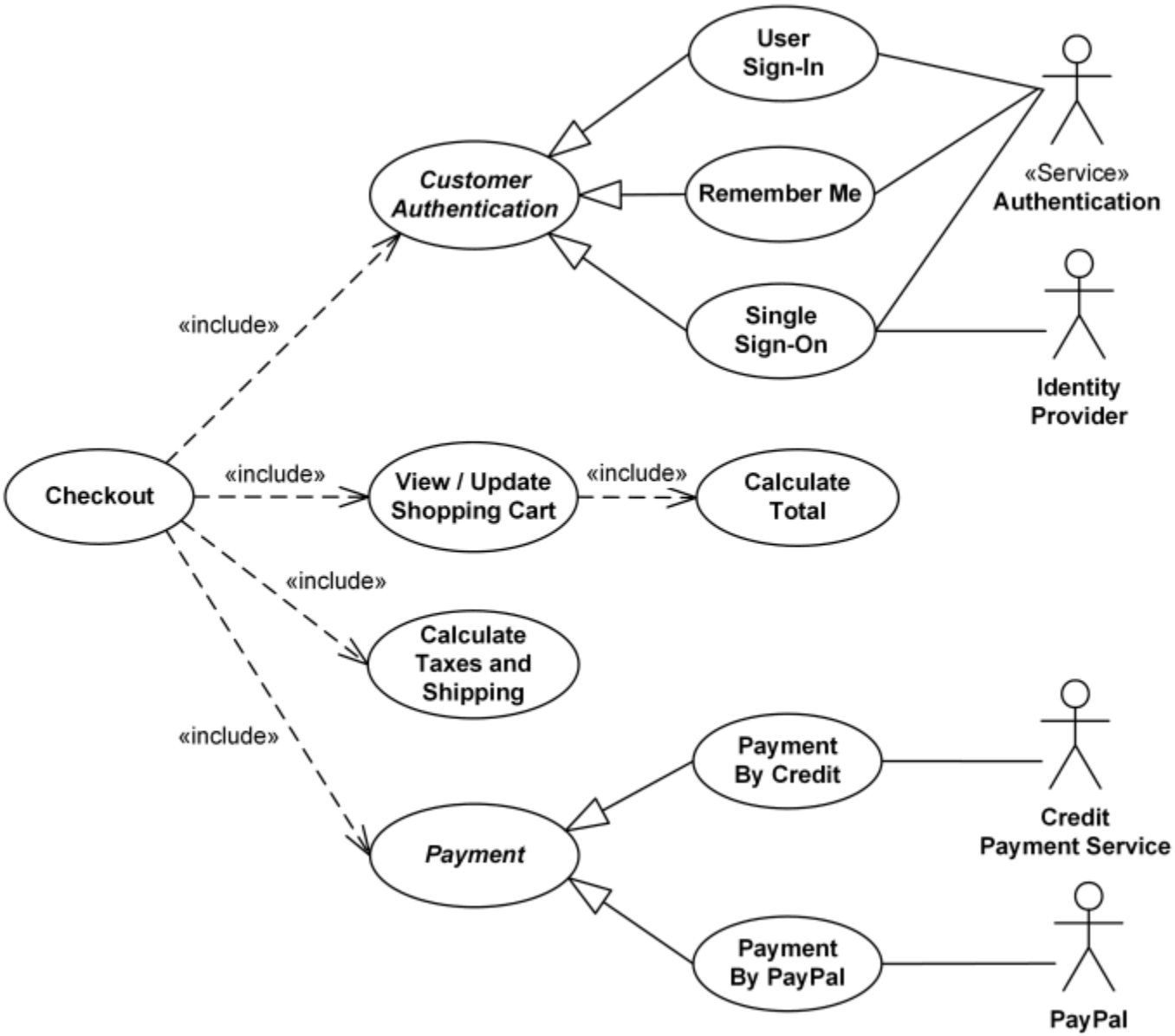


Use case diagram

On line Credit Cards Processing

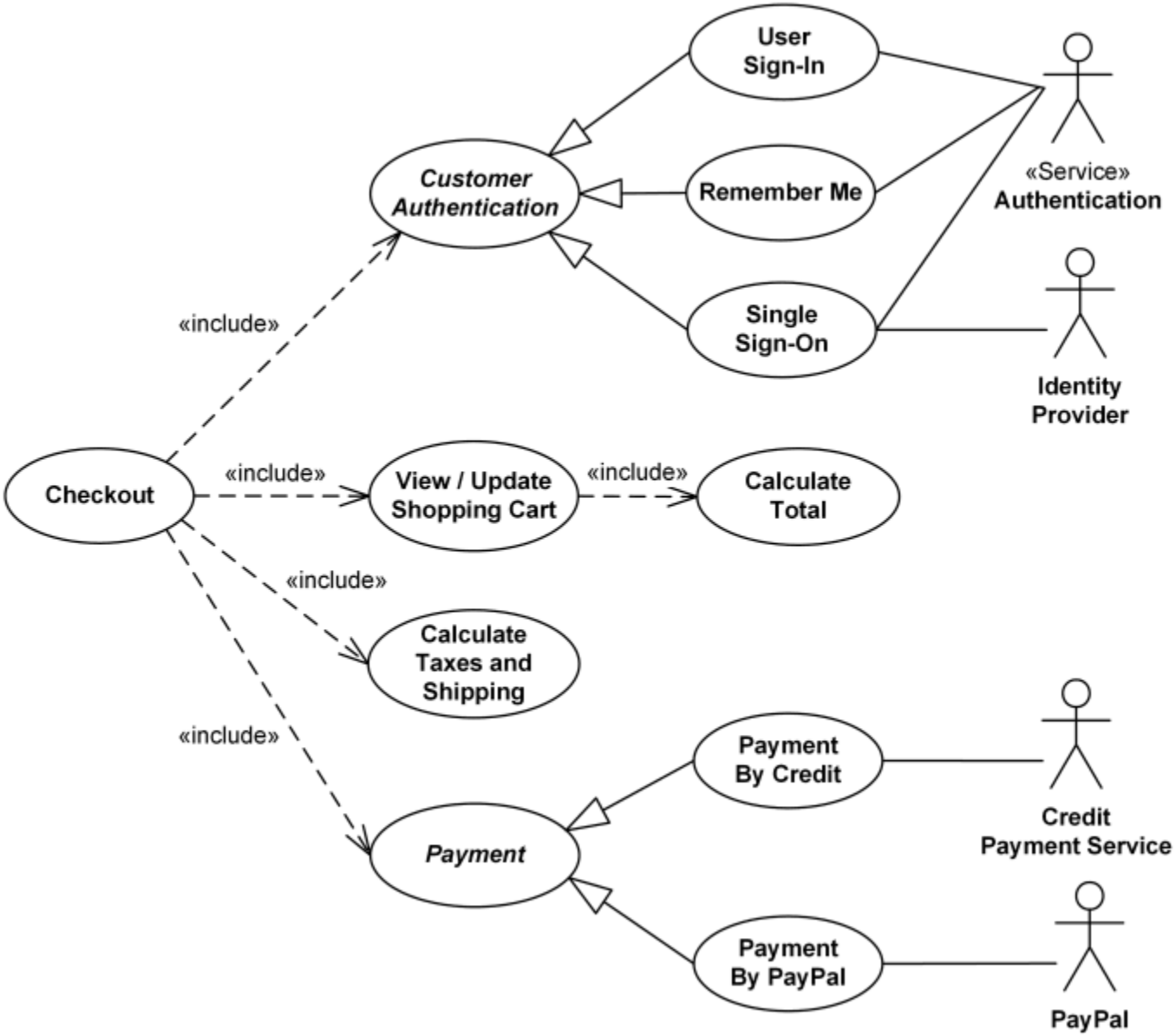


Class diagram

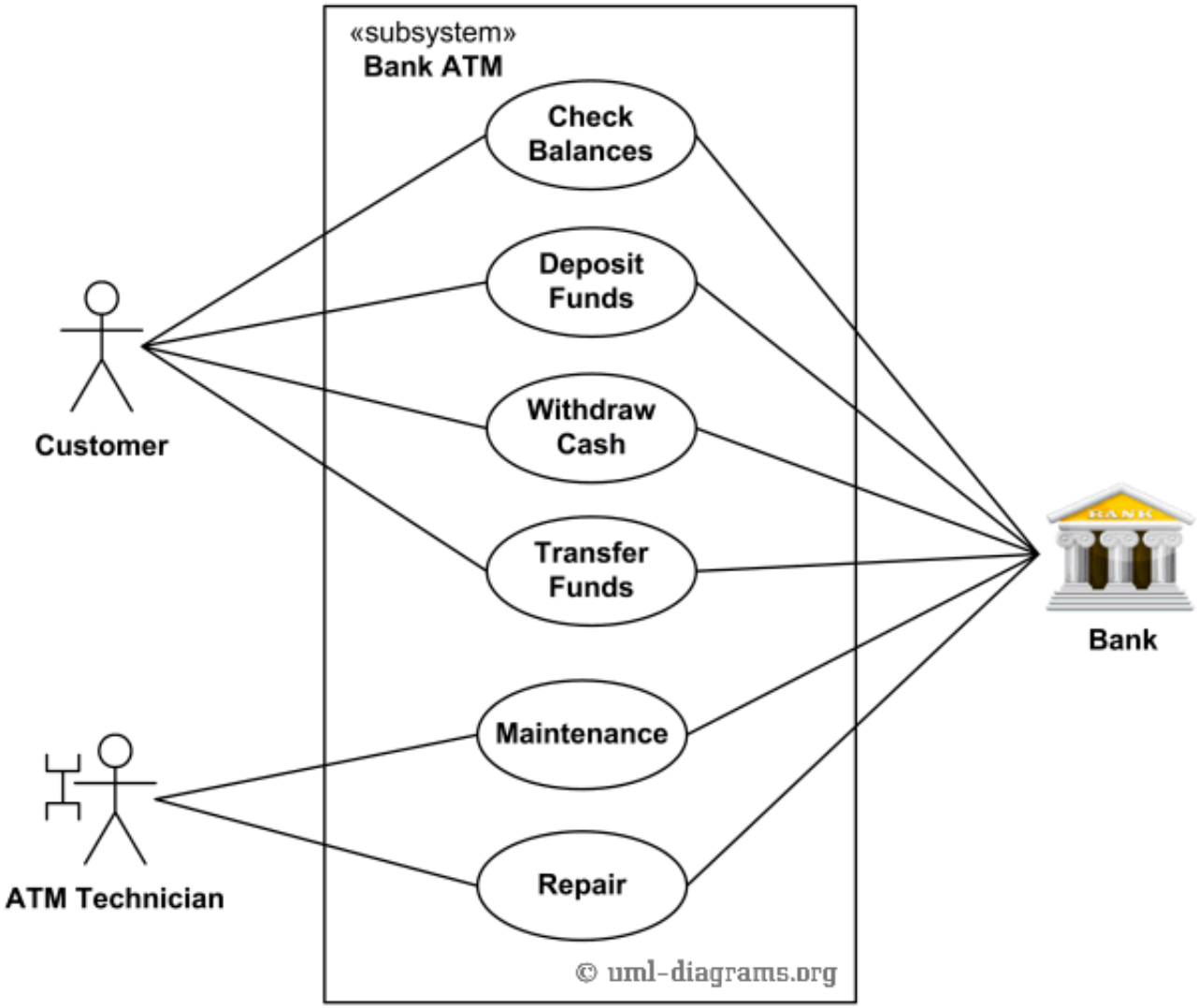


Use case diagram

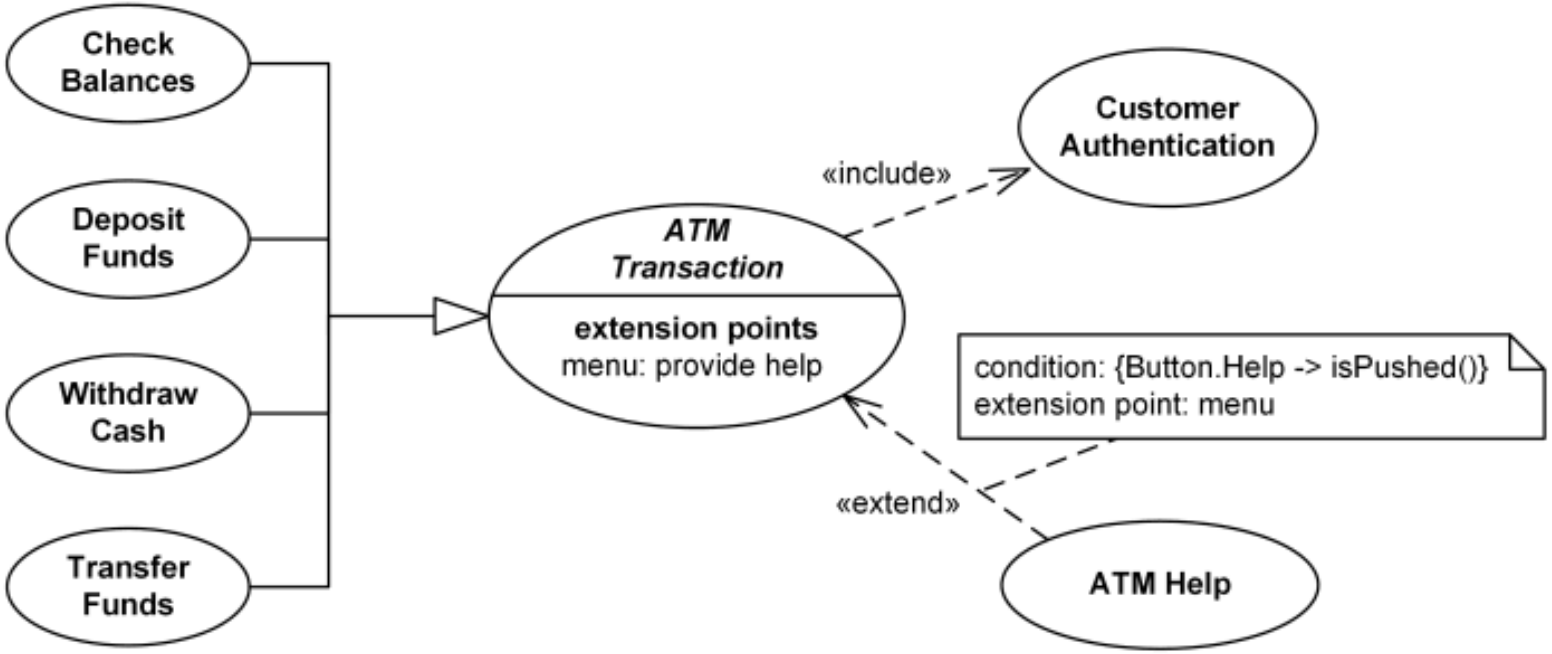
Credit Cards Processing



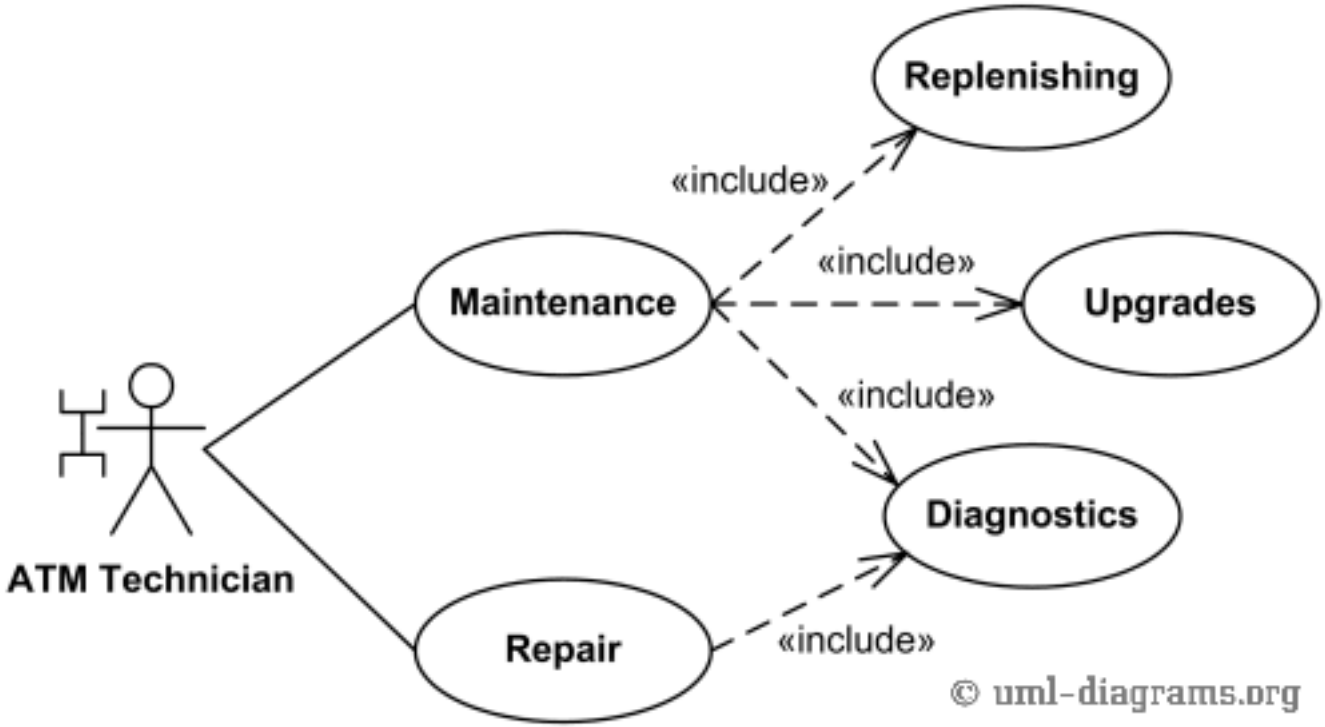
Use case diagram
Bank ATM



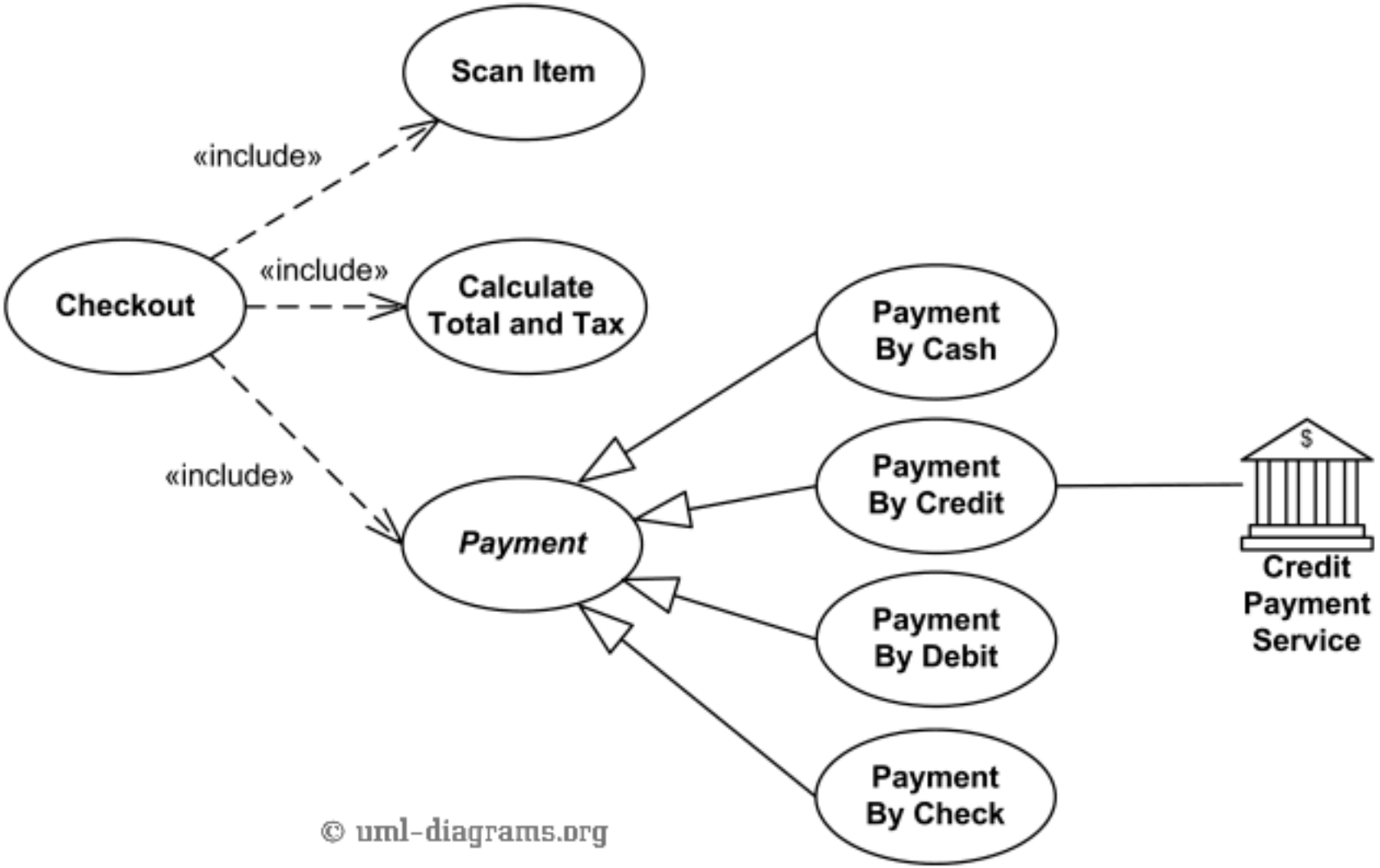
Use case diagram
Bank ATM



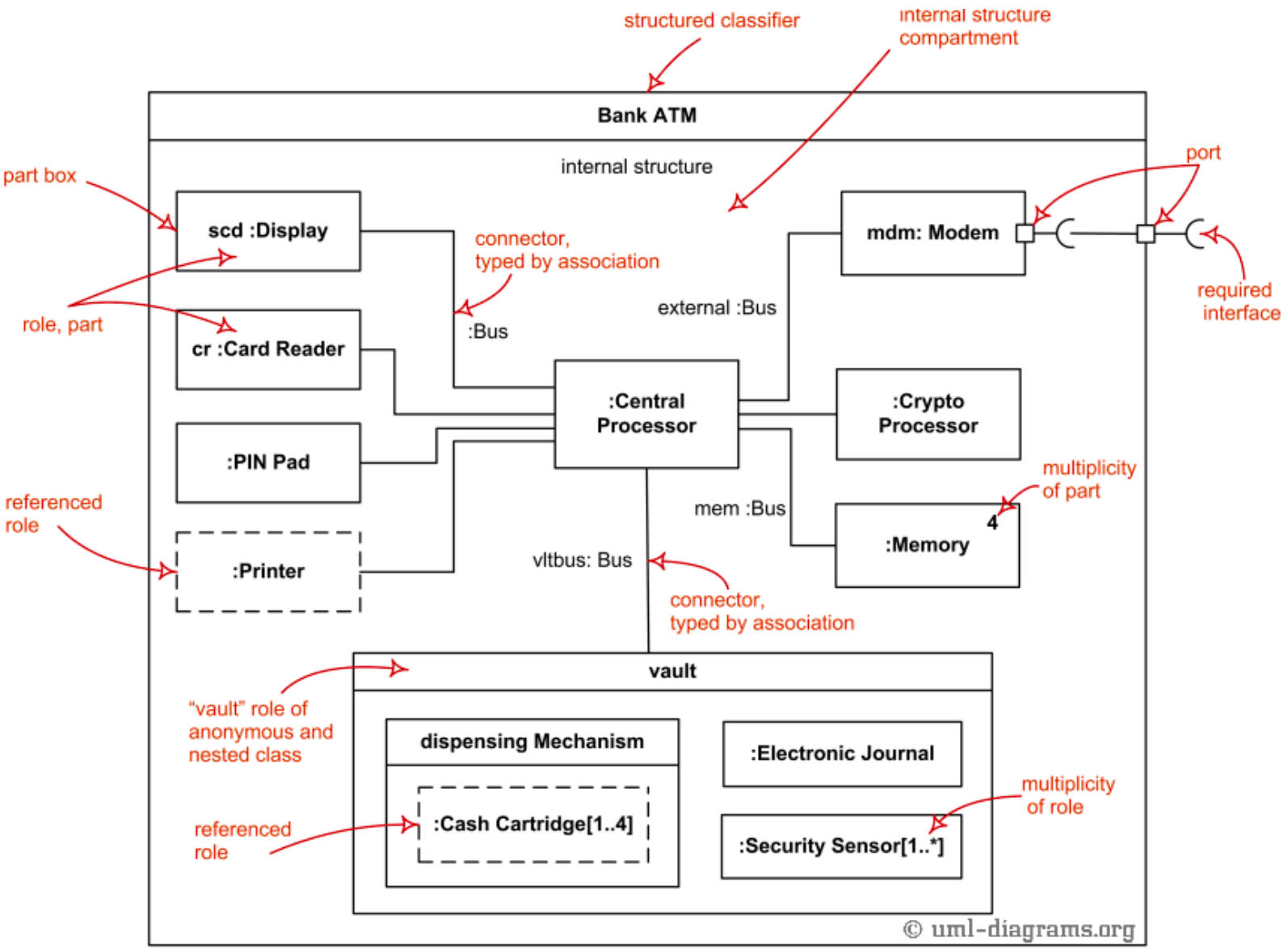
Use case diagram
Bank ATM



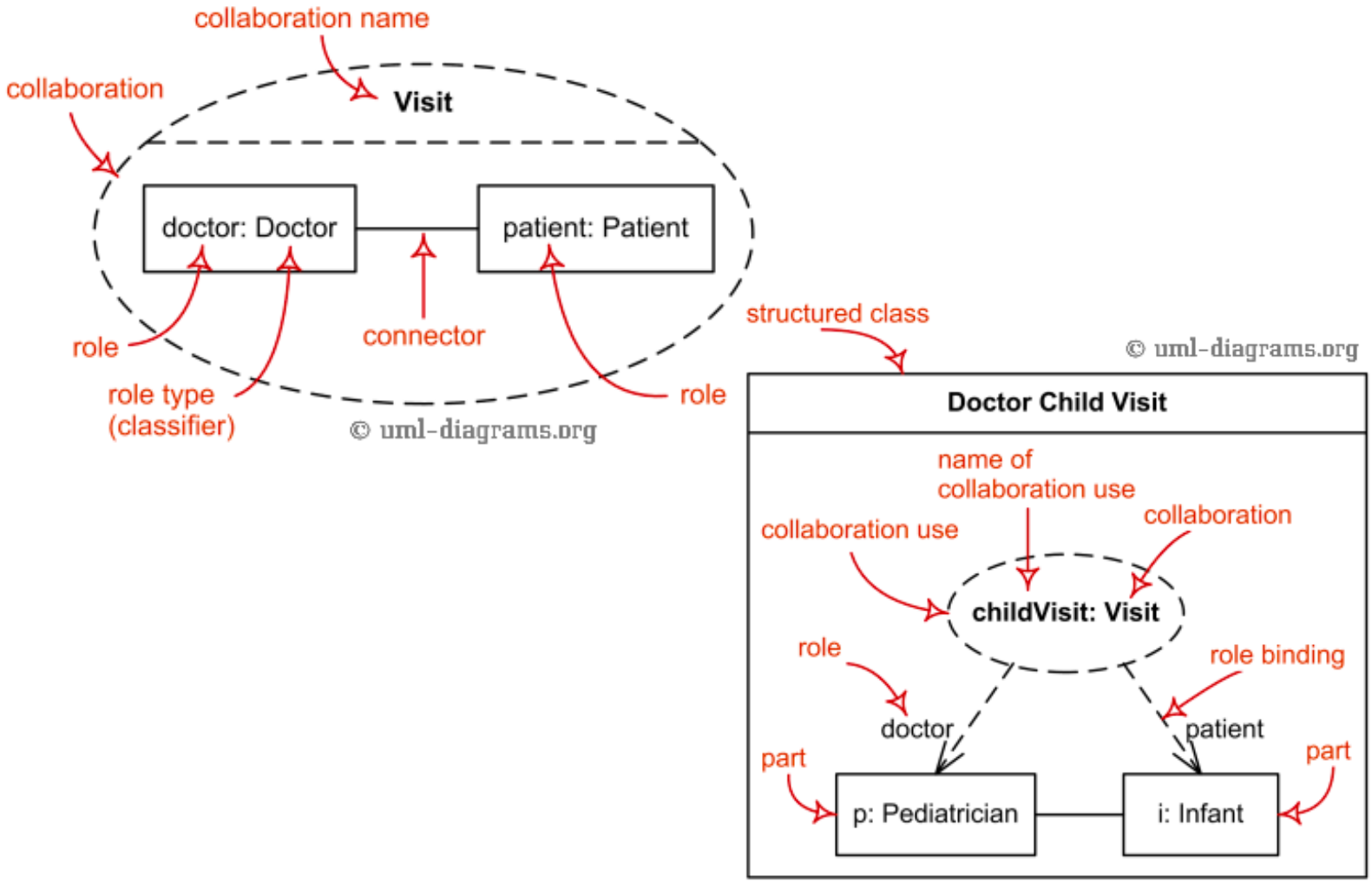
Use case diagram
Point of sales terminal



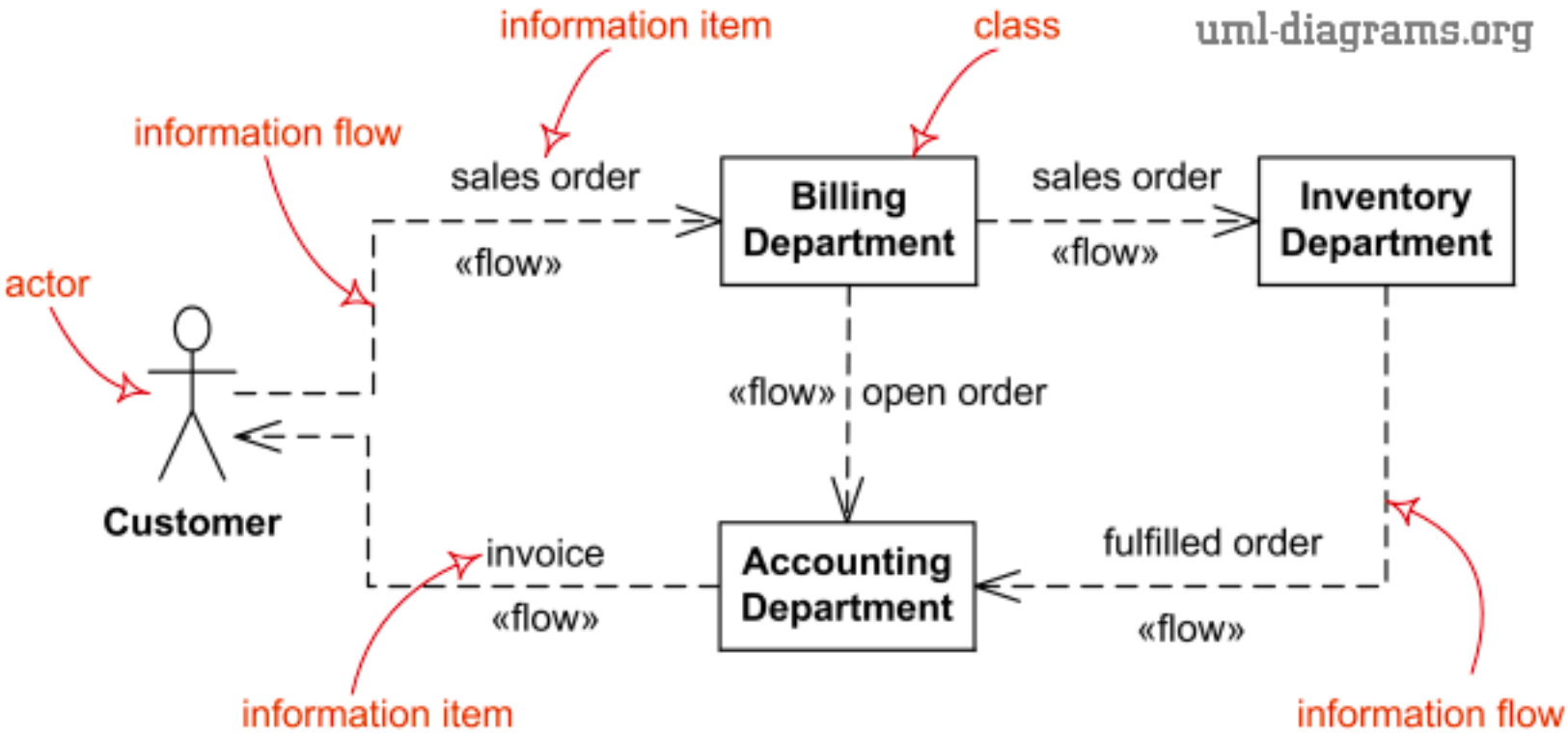
Composite structure diagram



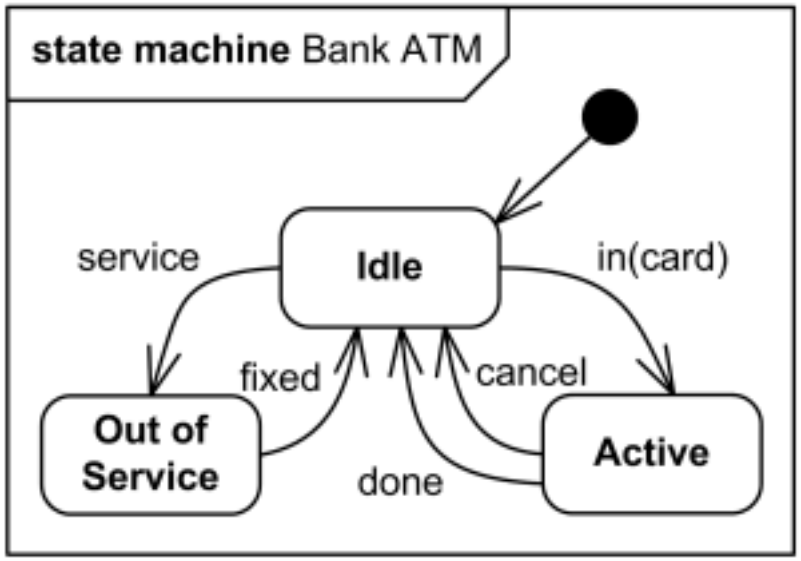
Composite structure diagram



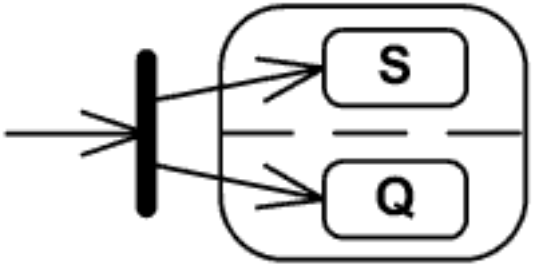
Information flow diagram High level communication



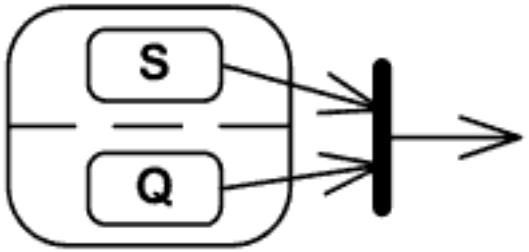
State machine



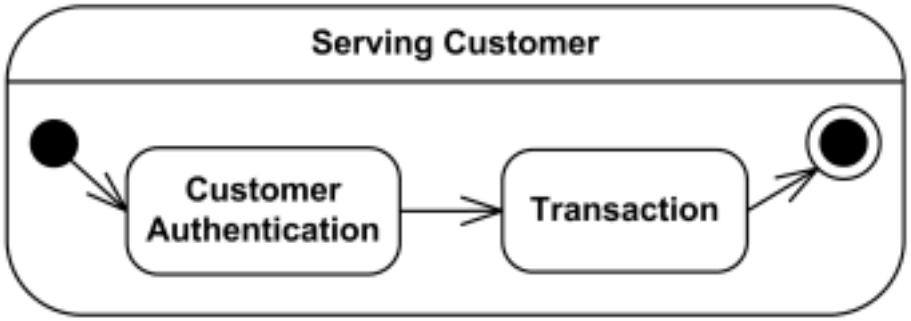
Fork



Join

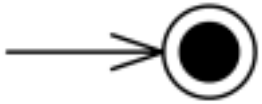


diagram



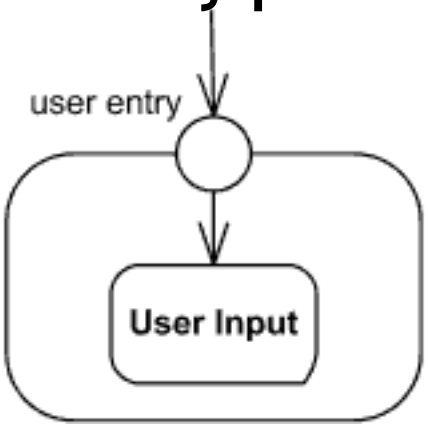
Initial state

Final state

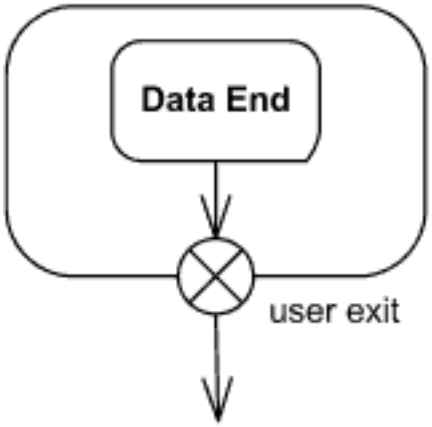


Class diagram

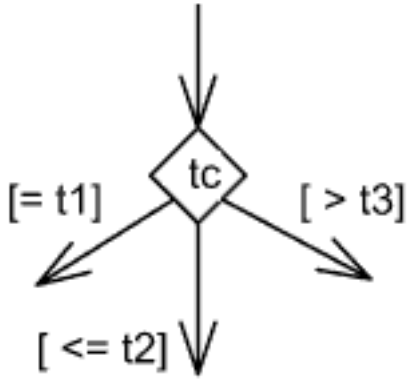
entry point



Exit point

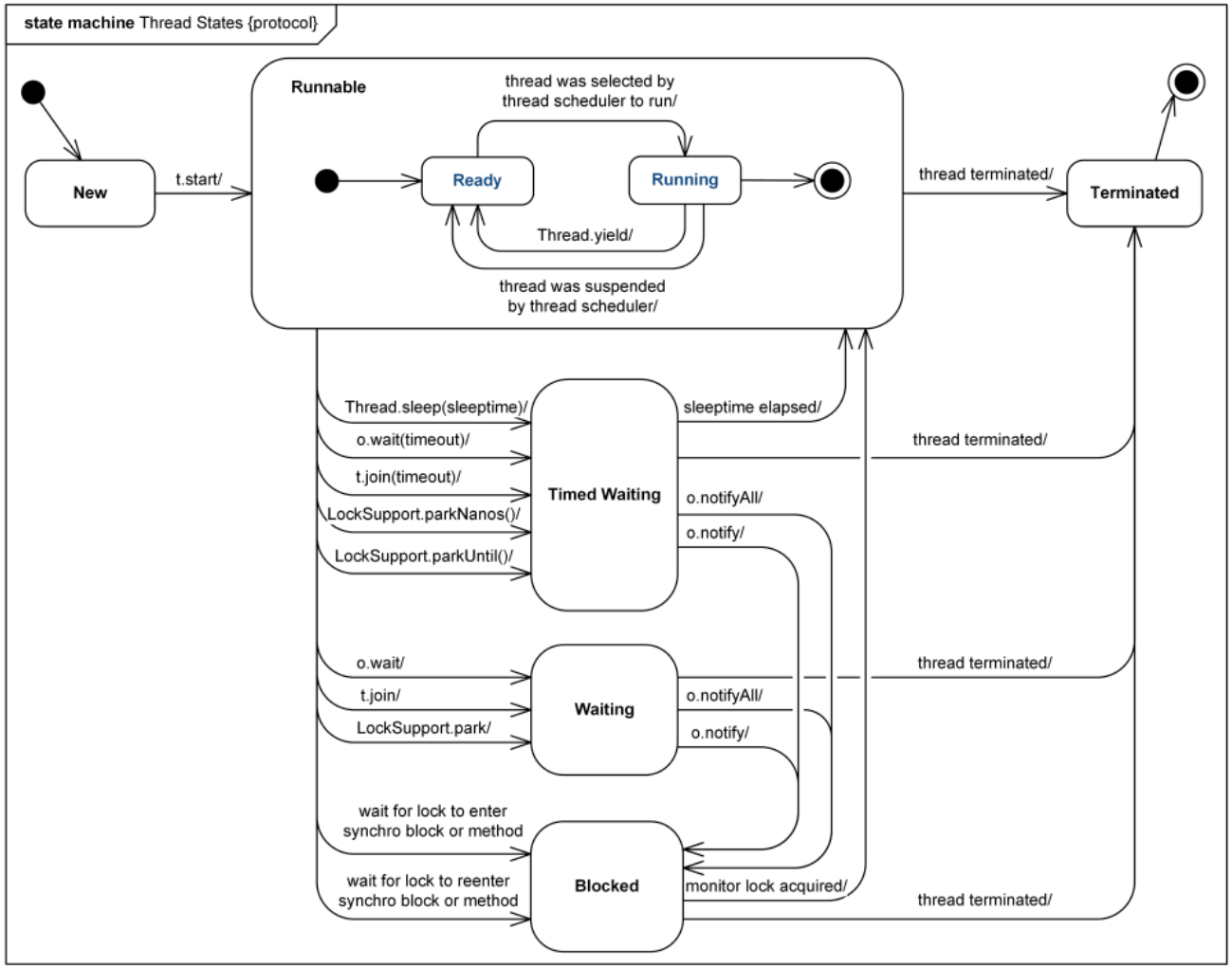


Choice



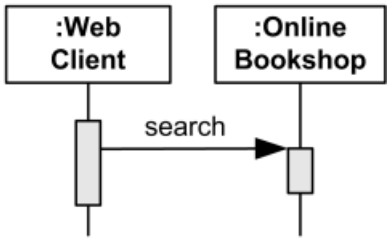
Protocol state machine diagram

w runnable, Timed waiting blocked, terminating a thread

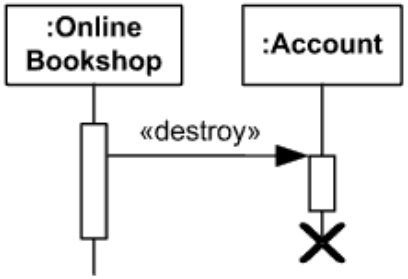


UML Message diagram

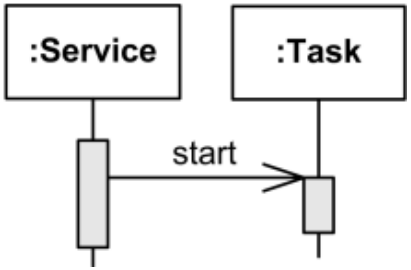
Synchronous Call



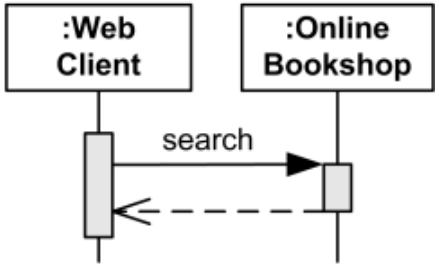
Delete Message



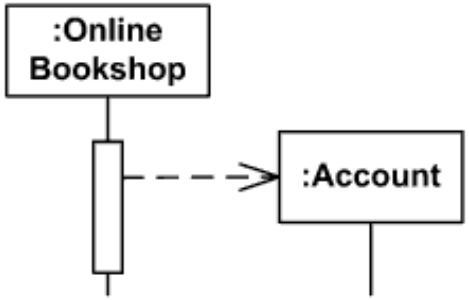
Asynchronous call



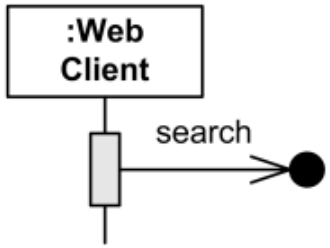
Reply message



Create message

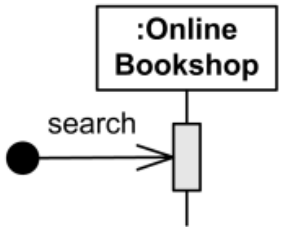


Lost Message

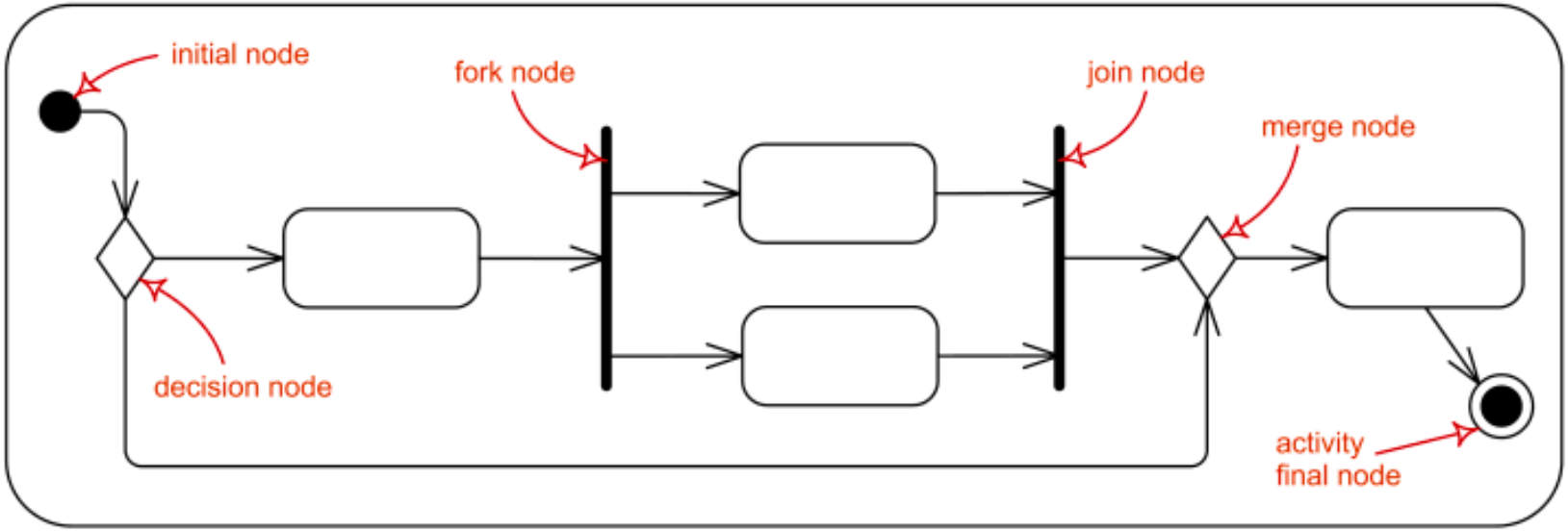


Class diagram

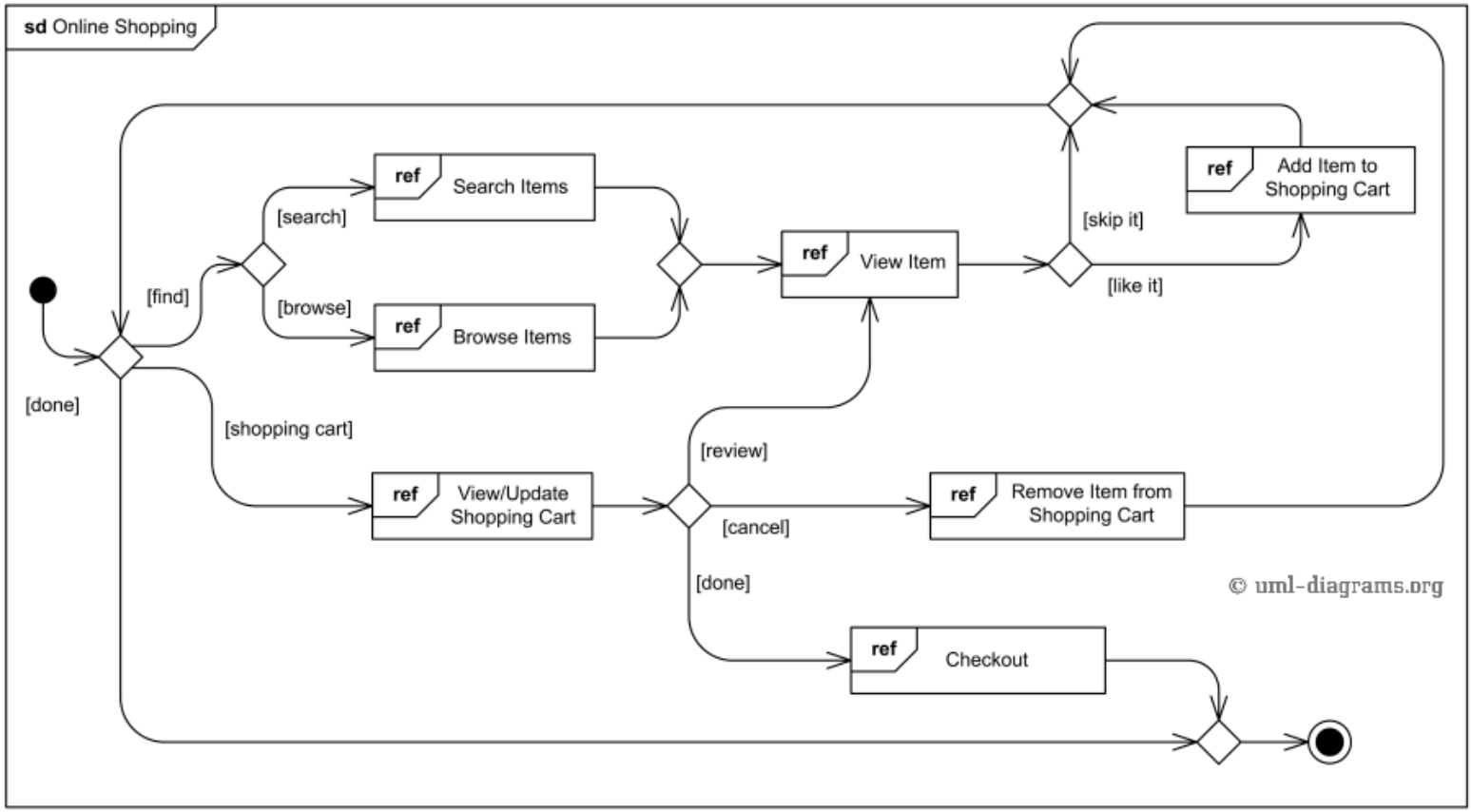
Found Message

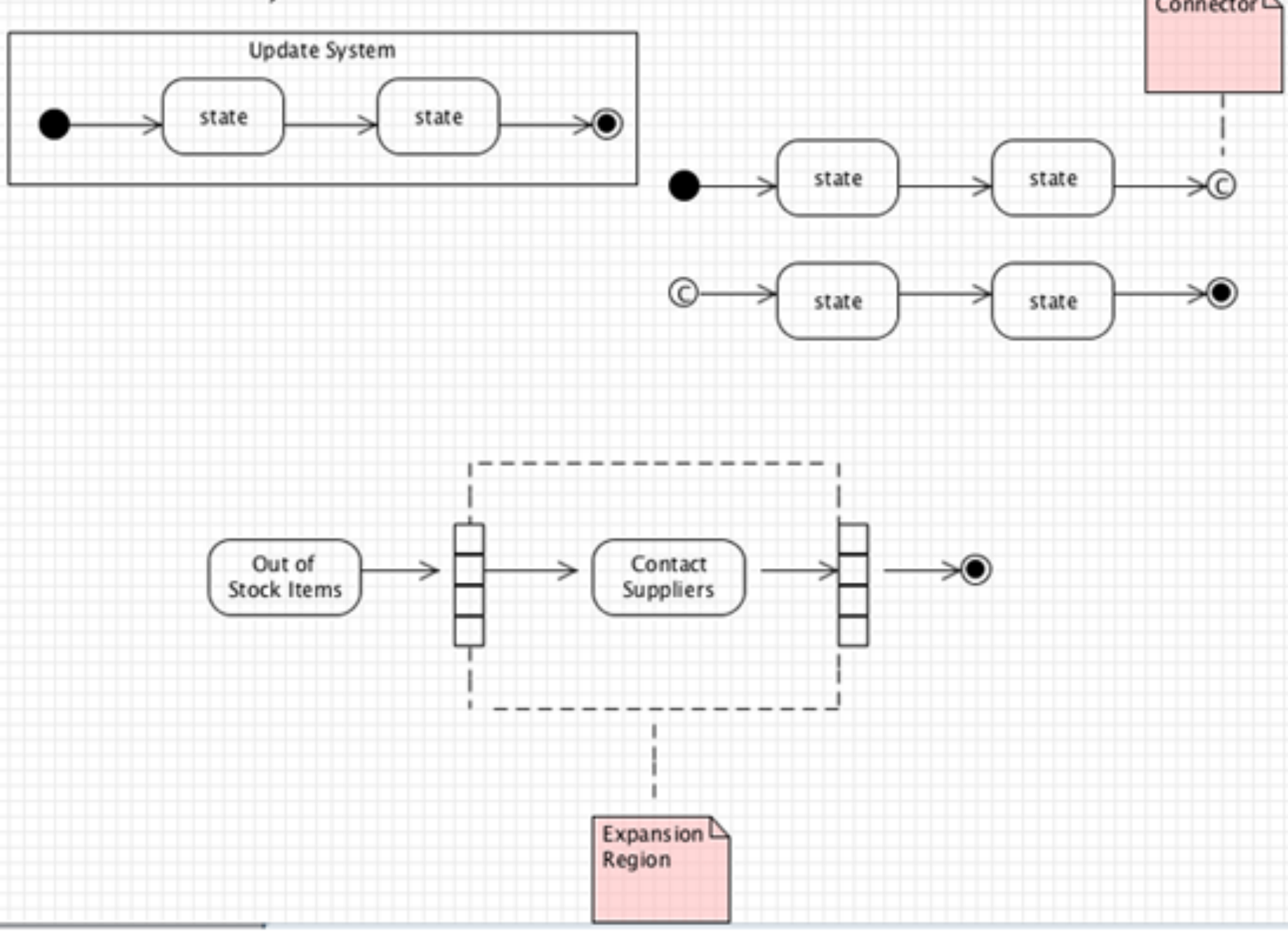


Activity Diagram Control



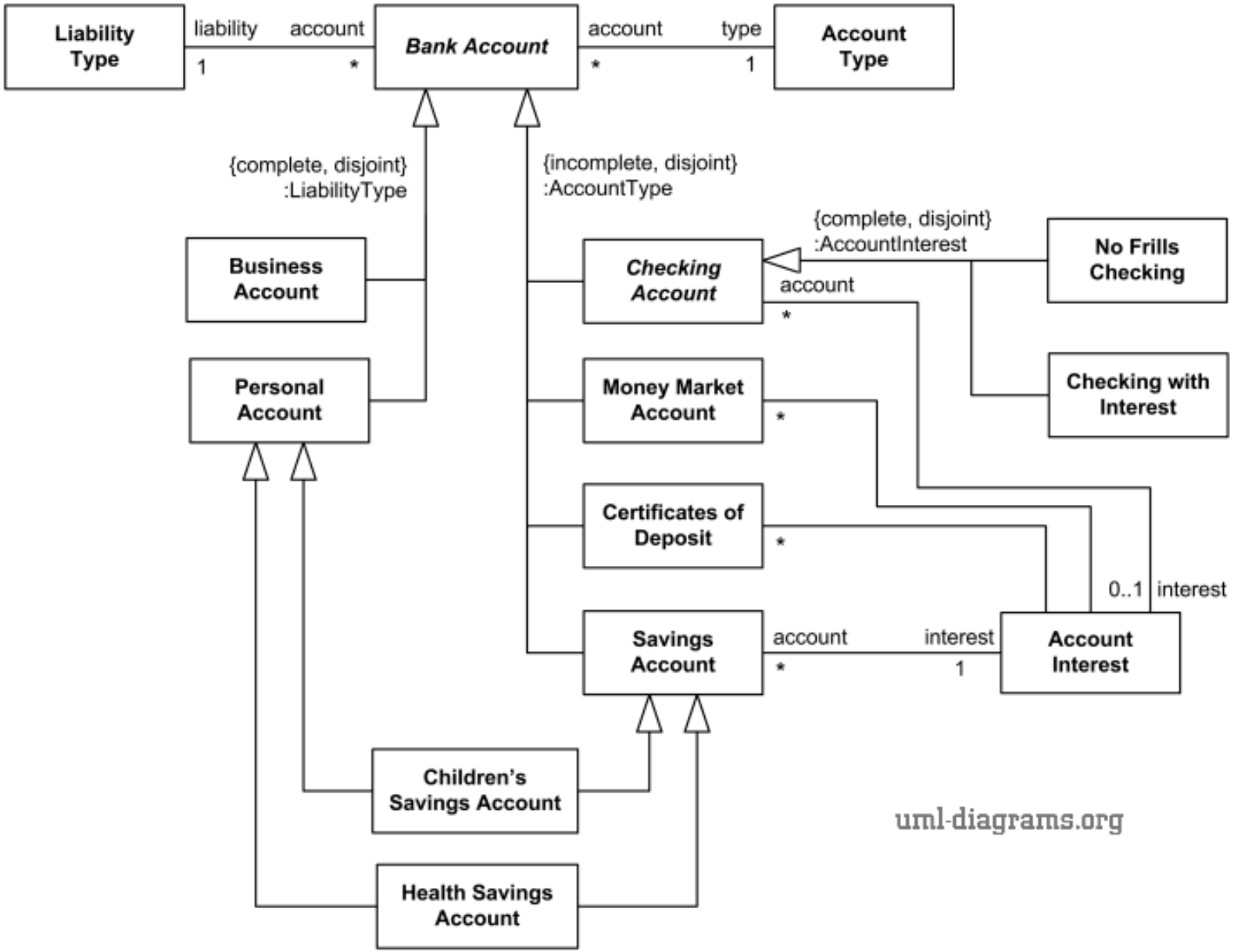
Interaction overview diagram



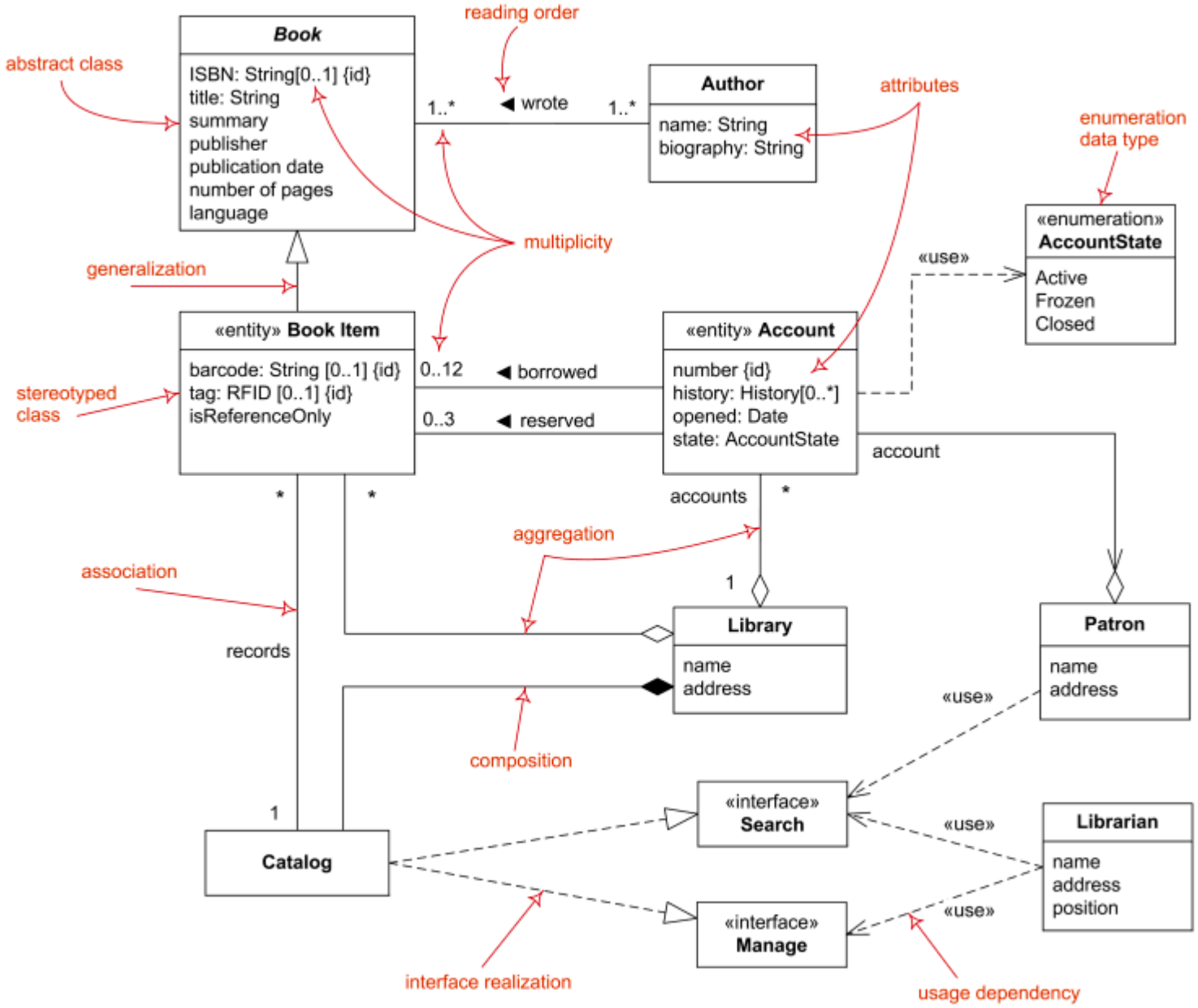

Update System
state
state
Connector
state
state
state
state
Out of
Stock Items
Contact
Suppliers
Expansion
Region

Class diagram

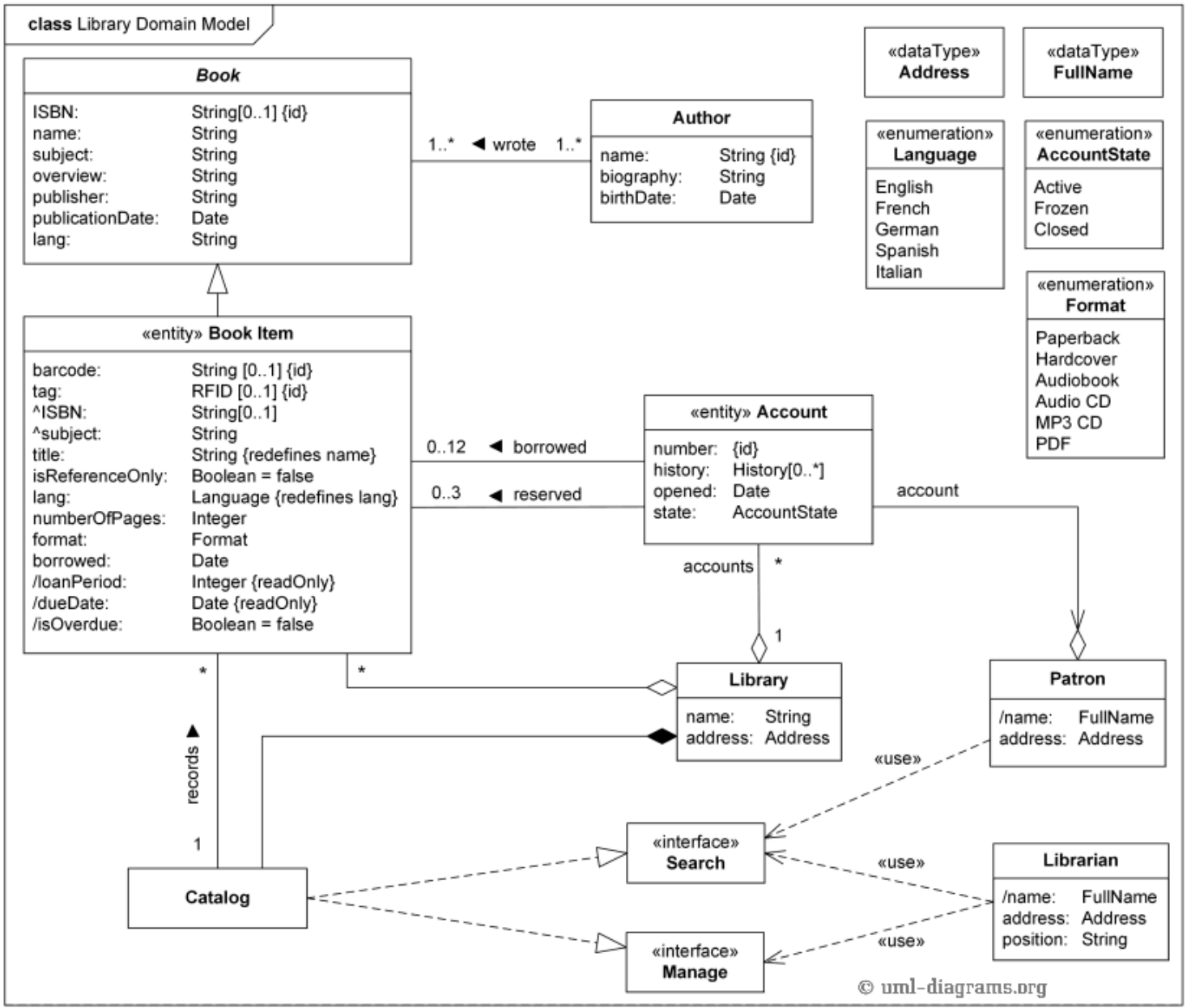
UML Bank Class diagram

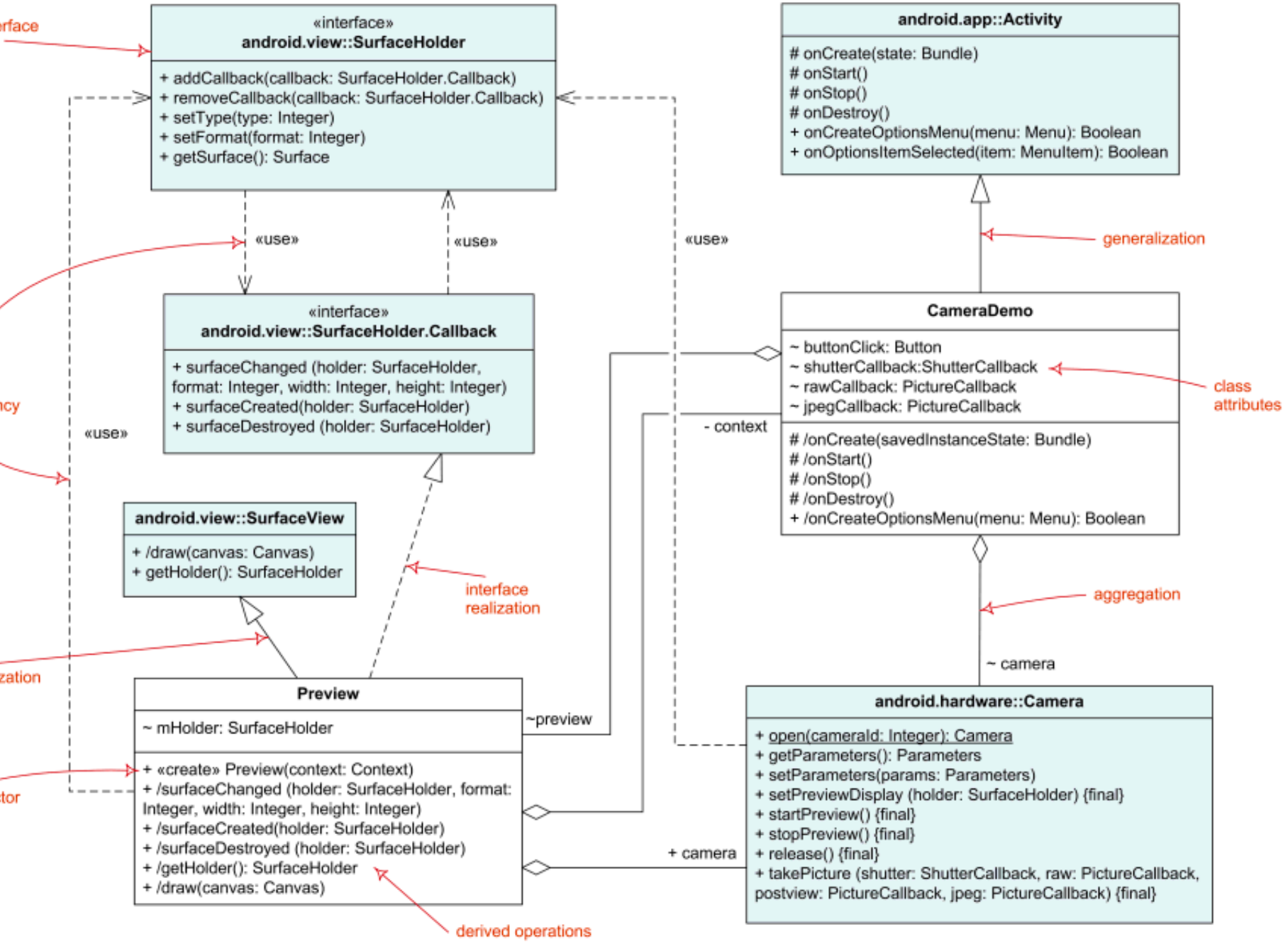


Domine model diagram From https://www.uml-diagrams.org/class-diagrams-overview.html

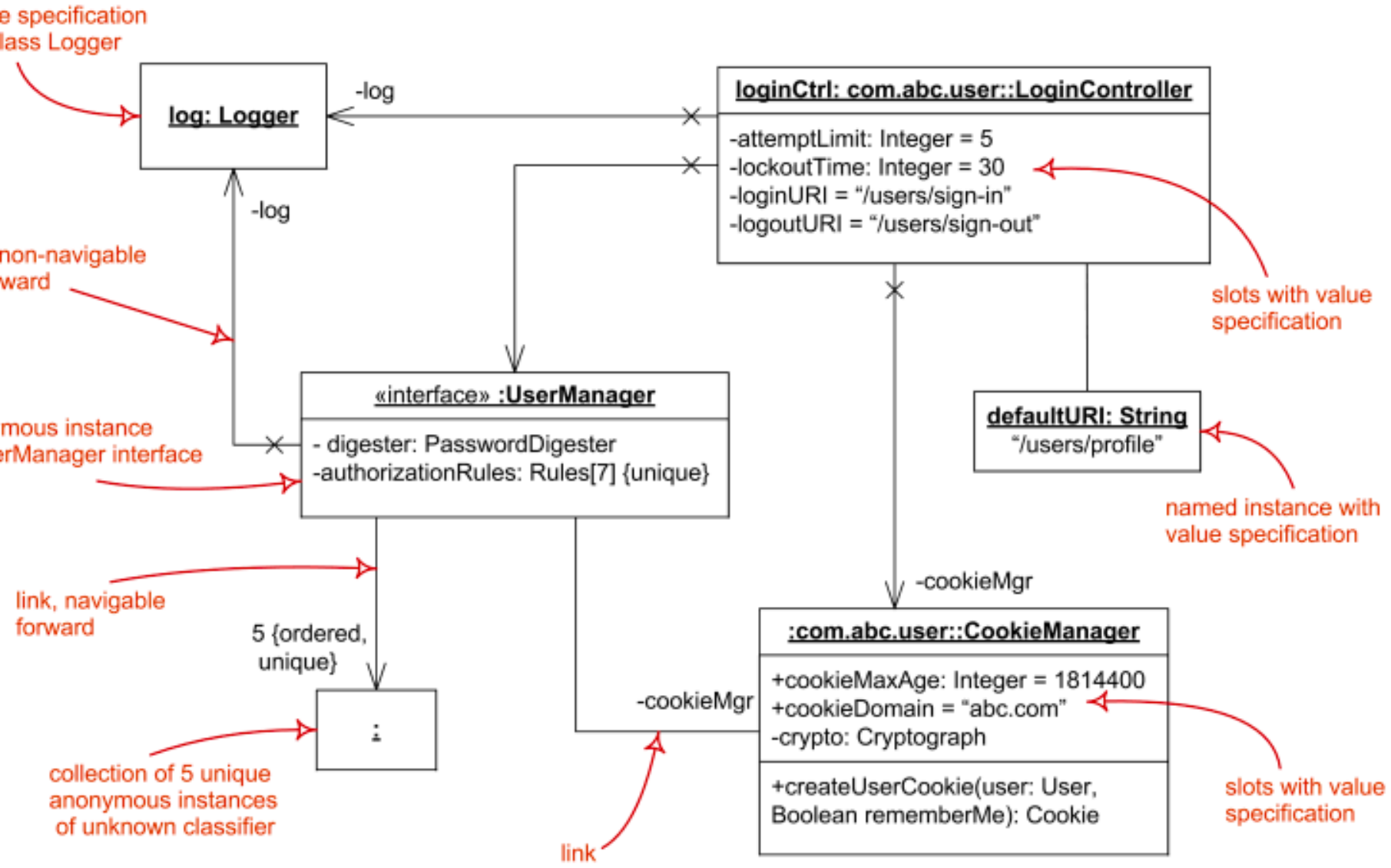


Domine model diagram

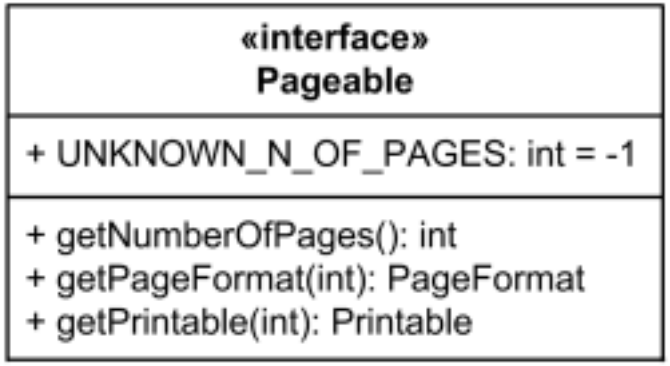


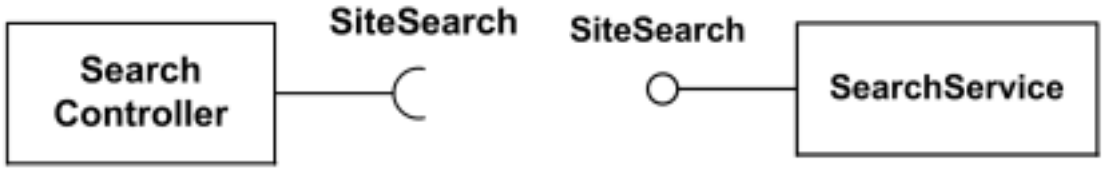

«interface»
android.view::SurfaceHolder
+ addCallback(callback: SurfaceHolder.Callback)
+ removeCallback(callback: SurfaceHolder.Callback)
+ setType(type: Integer)
+ setFormat(format: Integer)
+ getSurface(): Surface
android.app::Activity
onCreate(state: Bundle)
onStart()
onStop()
onDestroy()
+ onCreateOptionsMenu(menu: Menu): Boolean
+ onOptionsItemSelected(item: MenuItem): Boolean
«use»
«use»
«use»
generalization
«interface»
android.view::SurfaceHolder.Callback
+ surfaceChanged (holder: SurfaceHolder, format: Integer, width: Integer, height: Integer)
+ surfaceCreated(holder: SurfaceHolder)
+ surfaceDestroyed (holder: SurfaceHolder)
CameraDemo
~ buttonClick: Button
~ shutterCallback:ShutterCallback
~ rawCallback: PictureCallback
~ jpegCallback: PictureCallback
class attributes
- context
/onCreate(savedInstanceState: Bundle)
/onStart()
/onStop()
/onDestroy()
+ /onCreateOptionsMenu(menu: Menu): Boolean
«use»
android.view::SurfaceView
+ /draw(canvas: Canvas)
+ getHolder(): SurfaceHolder
interface realization
aggregation
~ camera
Preview
~ mHolder: SurfaceHolder
~preview
+ «create» Preview(context: Context)
+ /surfaceChanged (holder: SurfaceHolder, format: Integer, width: Integer, height: Integer)
+ /surfaceCreated(holder: SurfaceHolder)
+ /surfaceDestroyed (holder: SurfaceHolder)
+ /getHolder(): SurfaceHolder
+ /draw(canvas: Canvas)
android.hardware::Camera
+ open(cameraId: Integer): Camera
+ getParameters(): Parameters
+ setParameters(params: Parameters)
+ setPreviewDisplay (holder: SurfaceHolder) {final}
+ startPreview() {final}
+ stopPreview() {final}
+ release() {final}
+ takePicture (shutter: ShutterCallback, raw: PictureCallback, postview: PictureCallback, jpeg: PictureCallback) {final}
+ camera
derived operations

Object diagram



Domine model diagram





Multiplicity diagram

Multiplicity

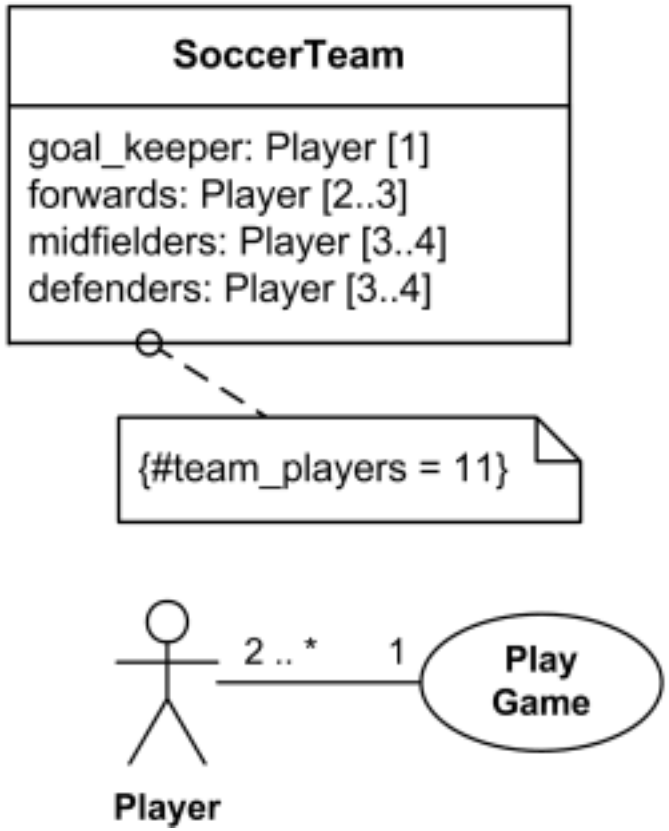
Option Cardinality

0..0
0
Collection must be empty
0..1

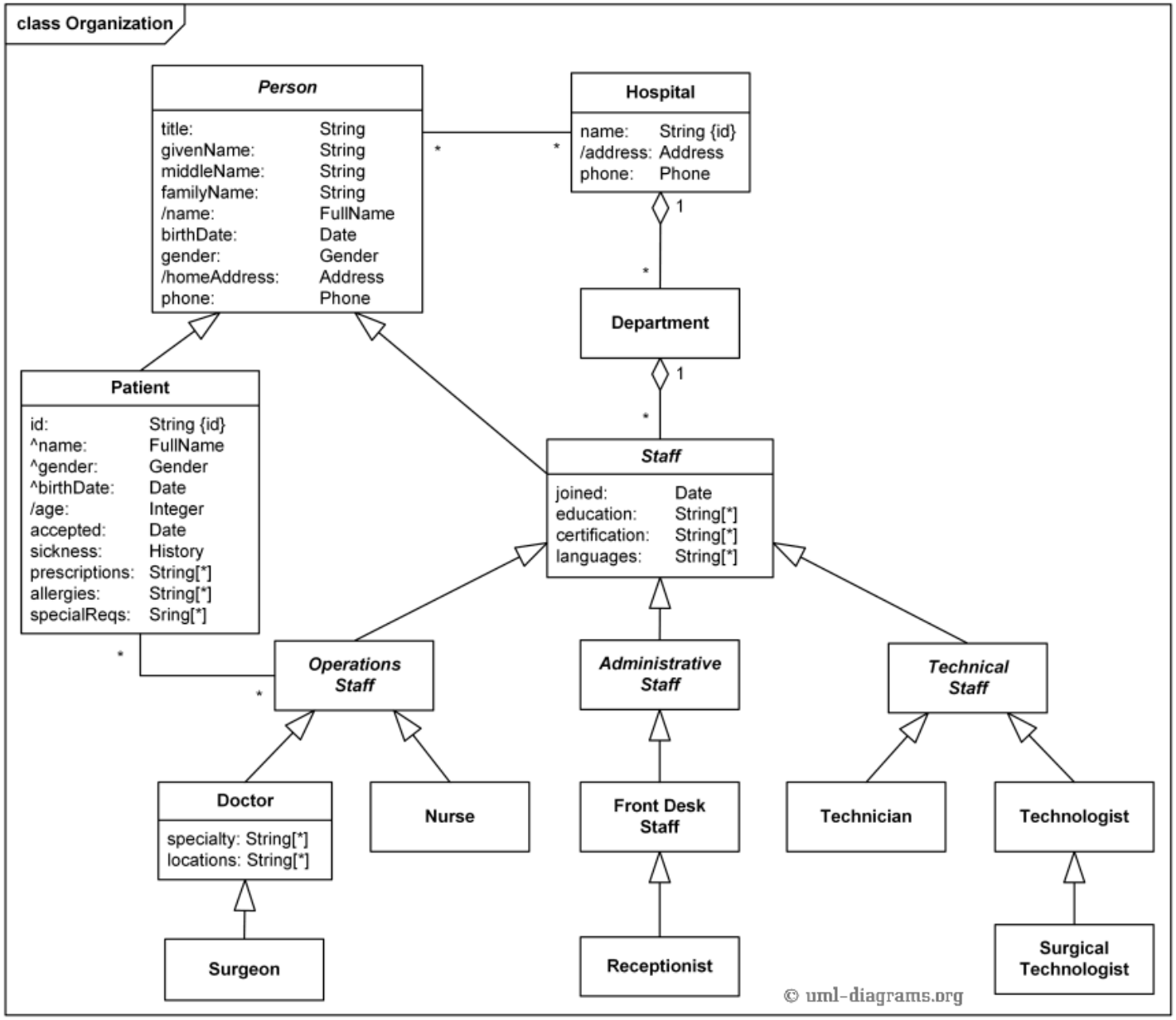
No instances or one instance
1..1
1
Exactly one instance
0..*
*
Zero or more instances
1..*

At least one instance
5..5
5
Exactly 5 instances
m..n

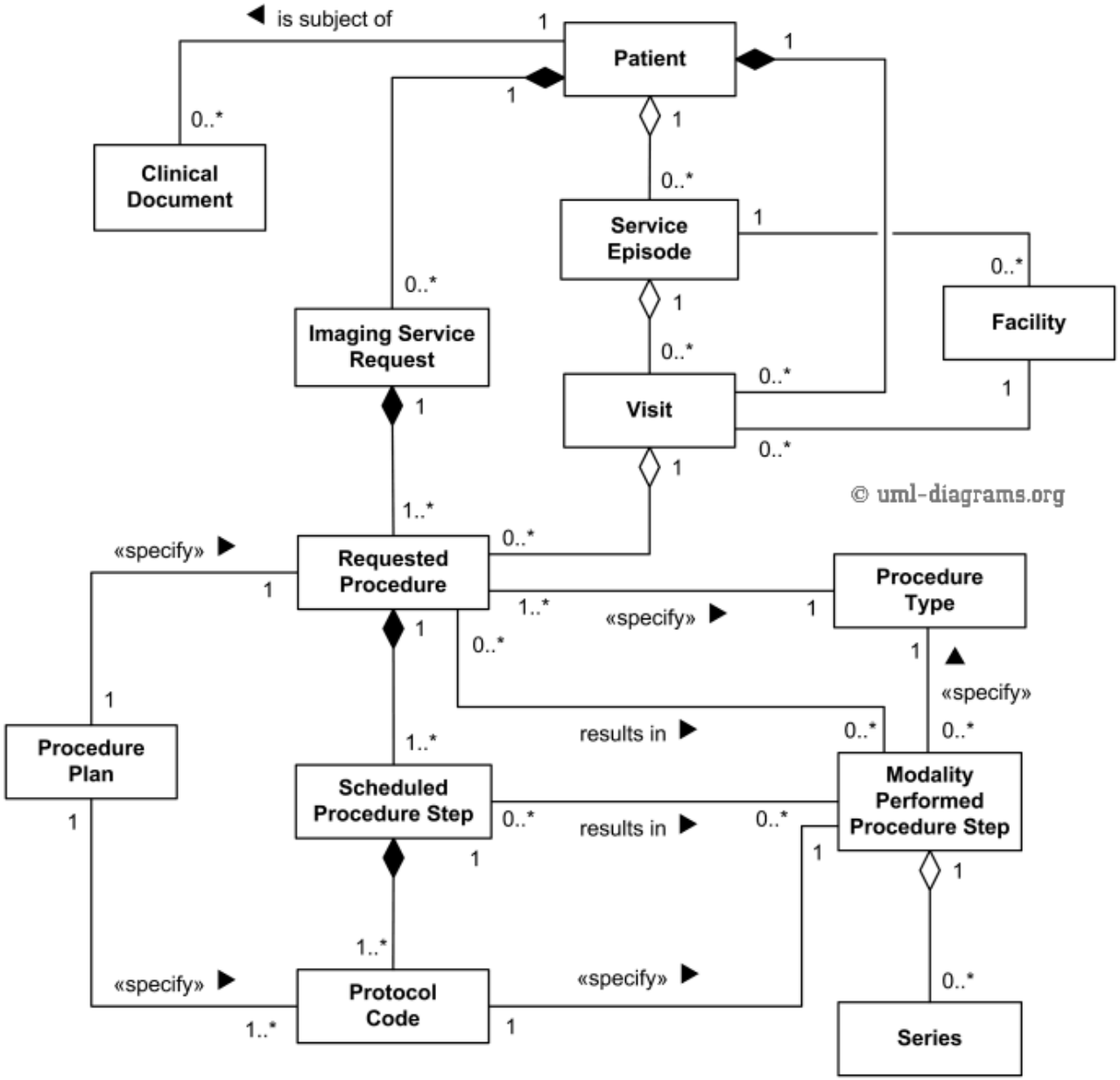
At least m but no more than n instances



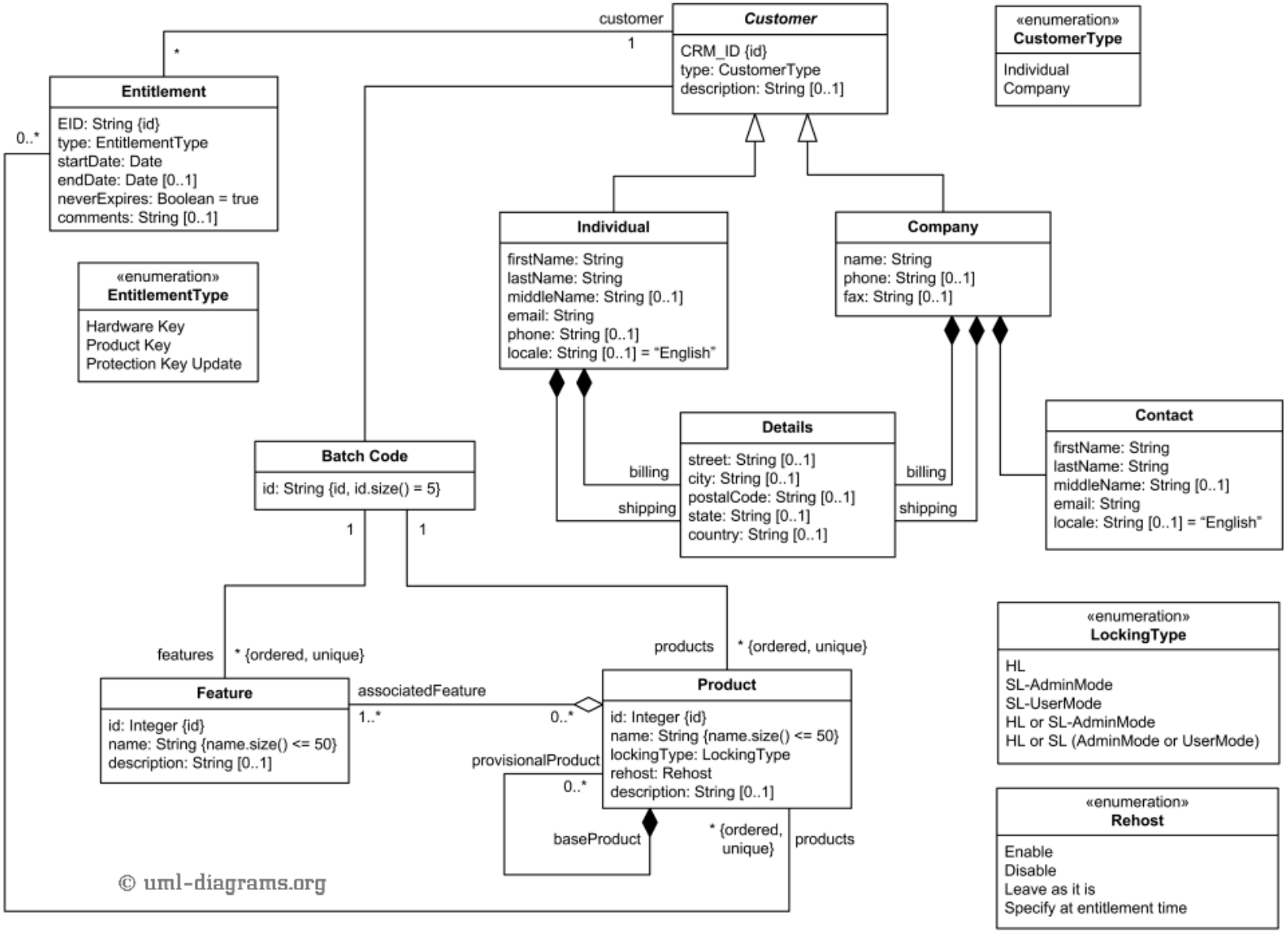
UML Class diagram

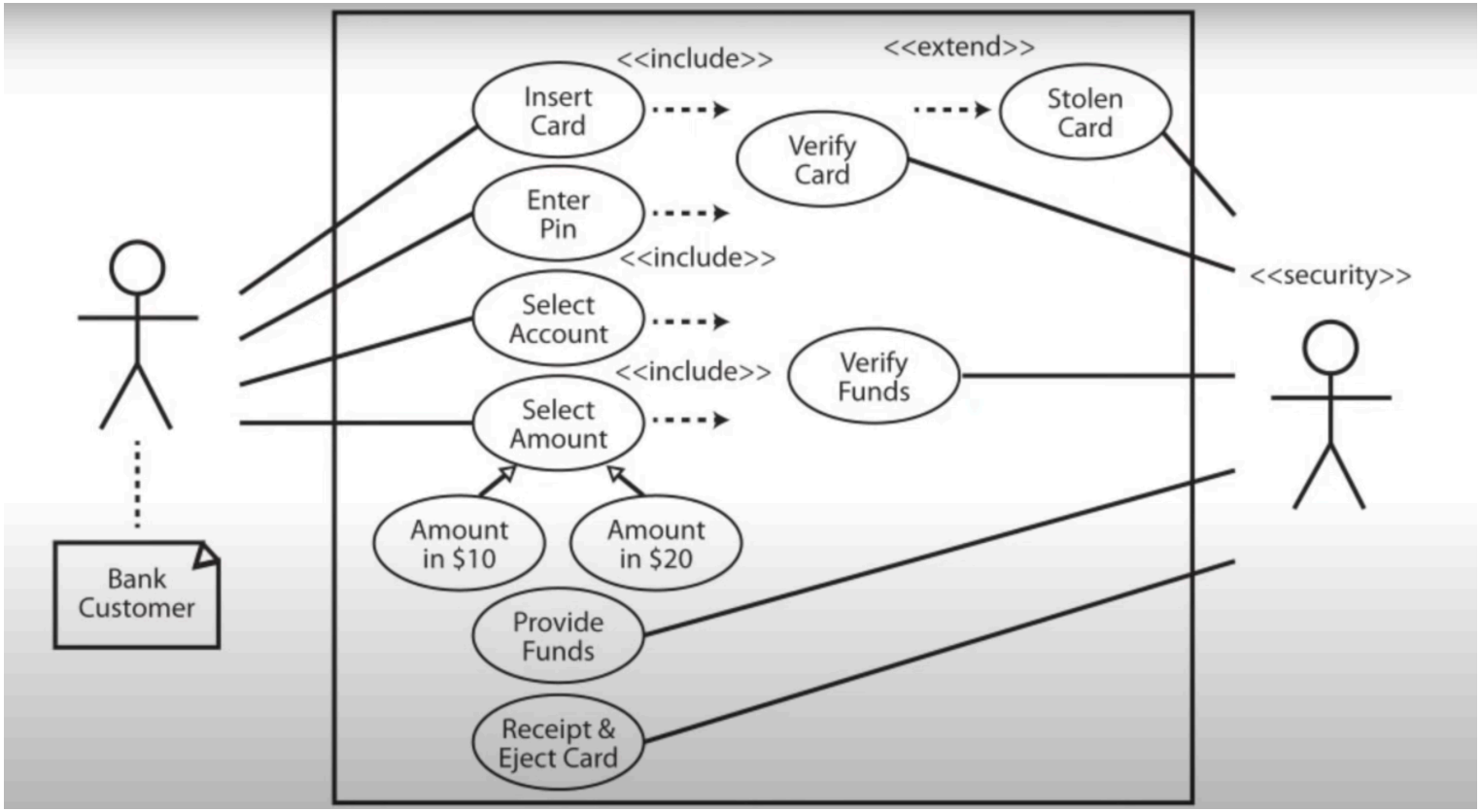


Class diagram



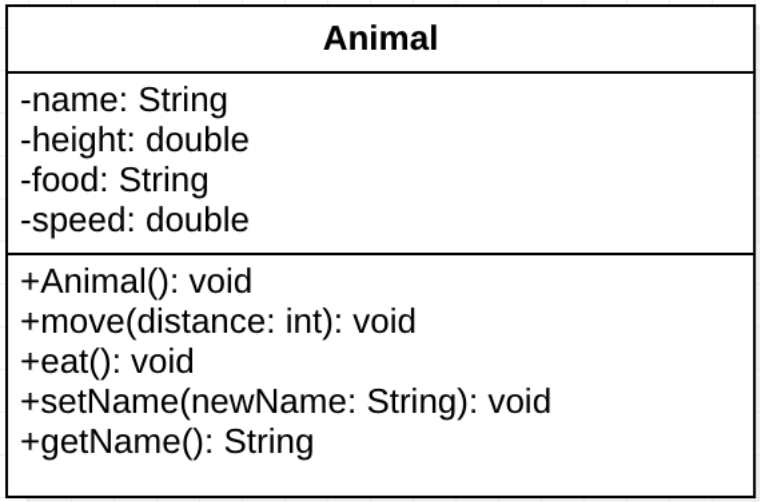
Sentinel diagram

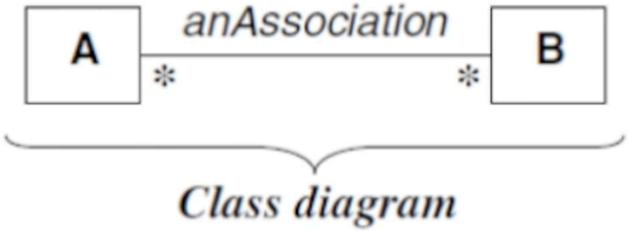



<<include>>
<<extend>>
Insert Card
Verify Card
Stolen Card
Enter Pin
<<include>>
<<security>>
Select Account
<<include>>
Verify Funds
Select Amount
Amount in $10
Amount in $20
Bank Customer
Provide Funds
Receipt & Eject Card

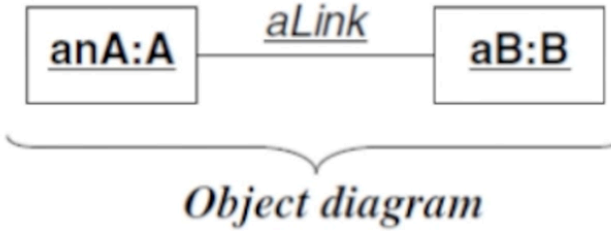
Thank you

- P. Sathya

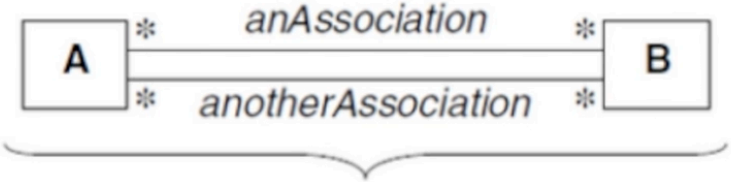

Animal
-name: String
-height: double
-food: String
-speed: double
+Animal(): void
+move(distance: int): void
+eat(): void
+setName(newName: String): void
+getName(): String



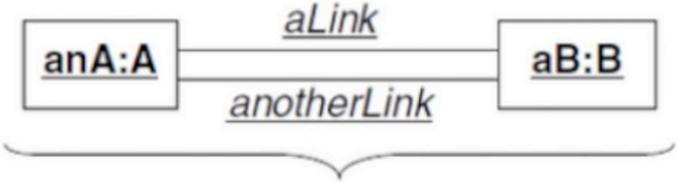
Class diagram



Object diagram

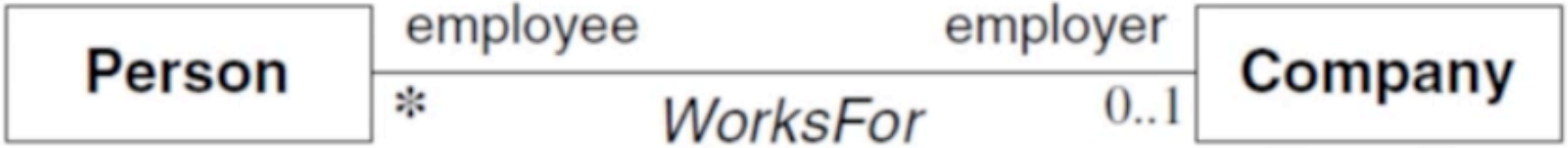


Class diagram

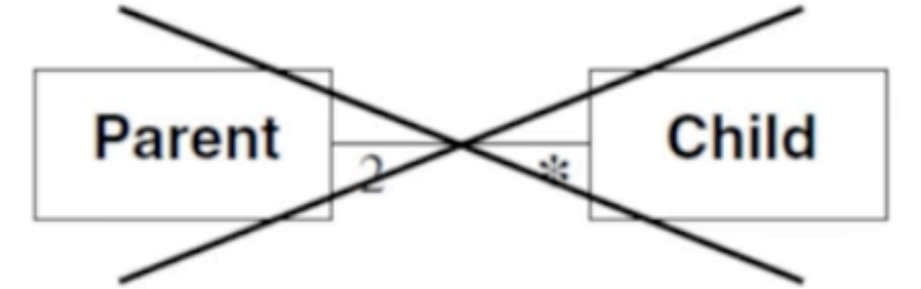


Object diagram

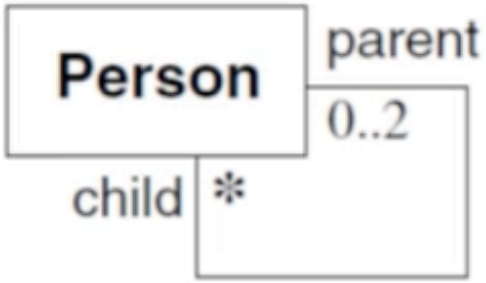
Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class



You should properly use association end names and not introduce a separate class for each reference



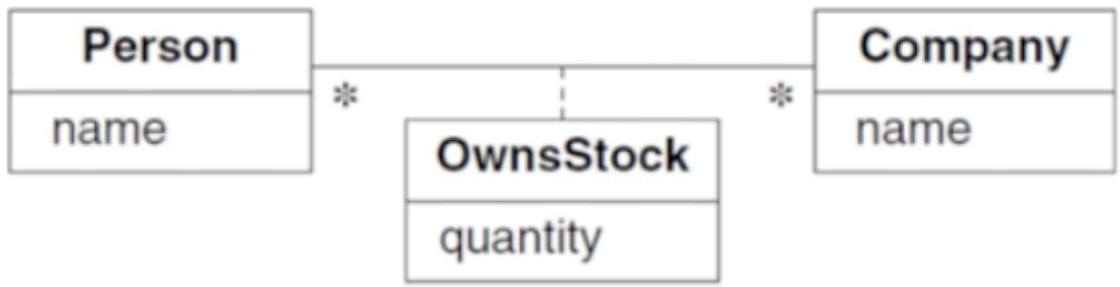
Wrong model



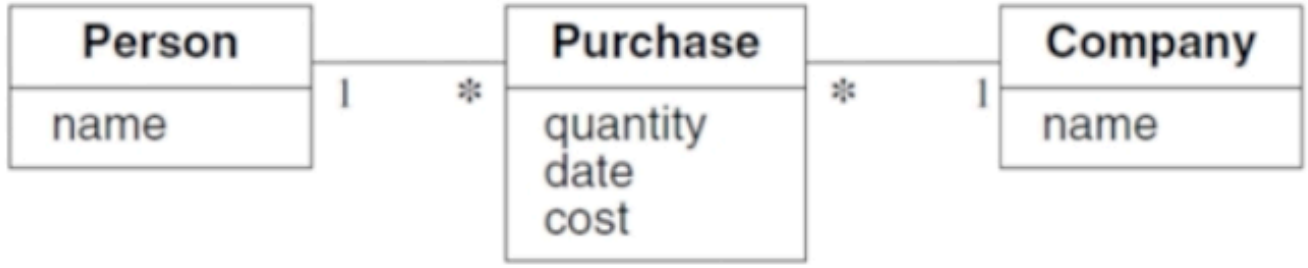
Correct model

An association class is an association that is also a class.

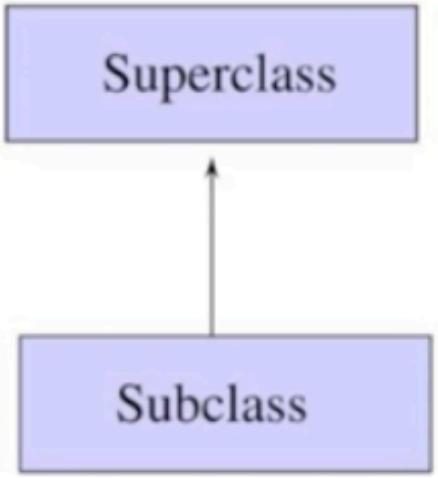
Association class



Ordinary class

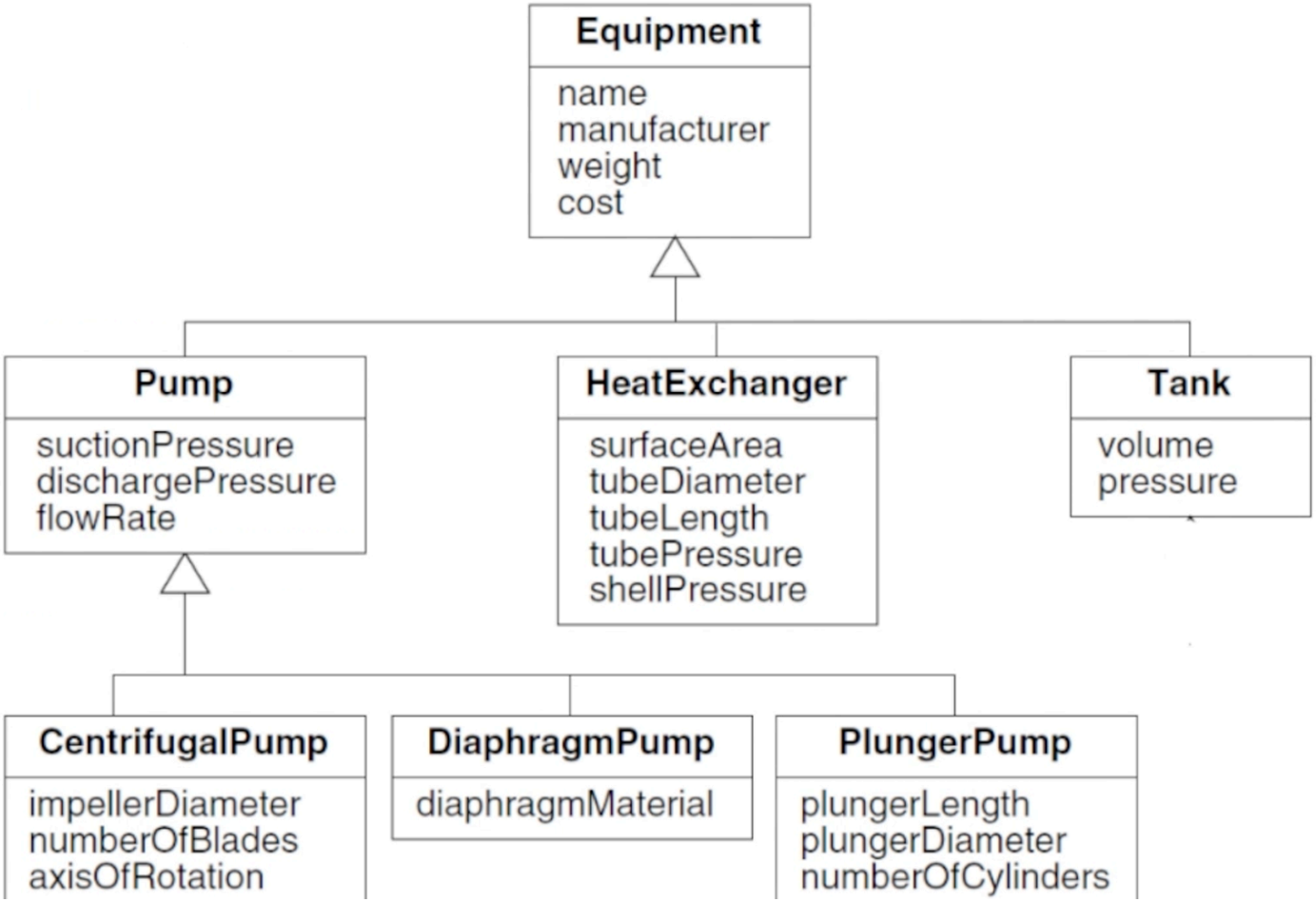


Generalization is the relationship between a class (the superclass) and one or more variations of the class (the subclasses).



Generalization organizes classes by their similarities and differences, structuring the description of objects.

Each subclass inherits the attributes, operations, and associations of its superclasses.


Equipment
name
manufacturer
weight
cost
Pump
suctionPressure
dischargePressure
flowRate
HeatExchanger
surfaceArea
tubeDiameter
tubeLength
tubePressure
shellPressure
Tank
volume
pressure
CentrifugalPump
impellerDiameter
numberOfBlades
axisOfRotation
DiaphragmPump
diaphragmMaterial
PlungerPump
plungerLength
plungerDiameter
numberOfCylinders

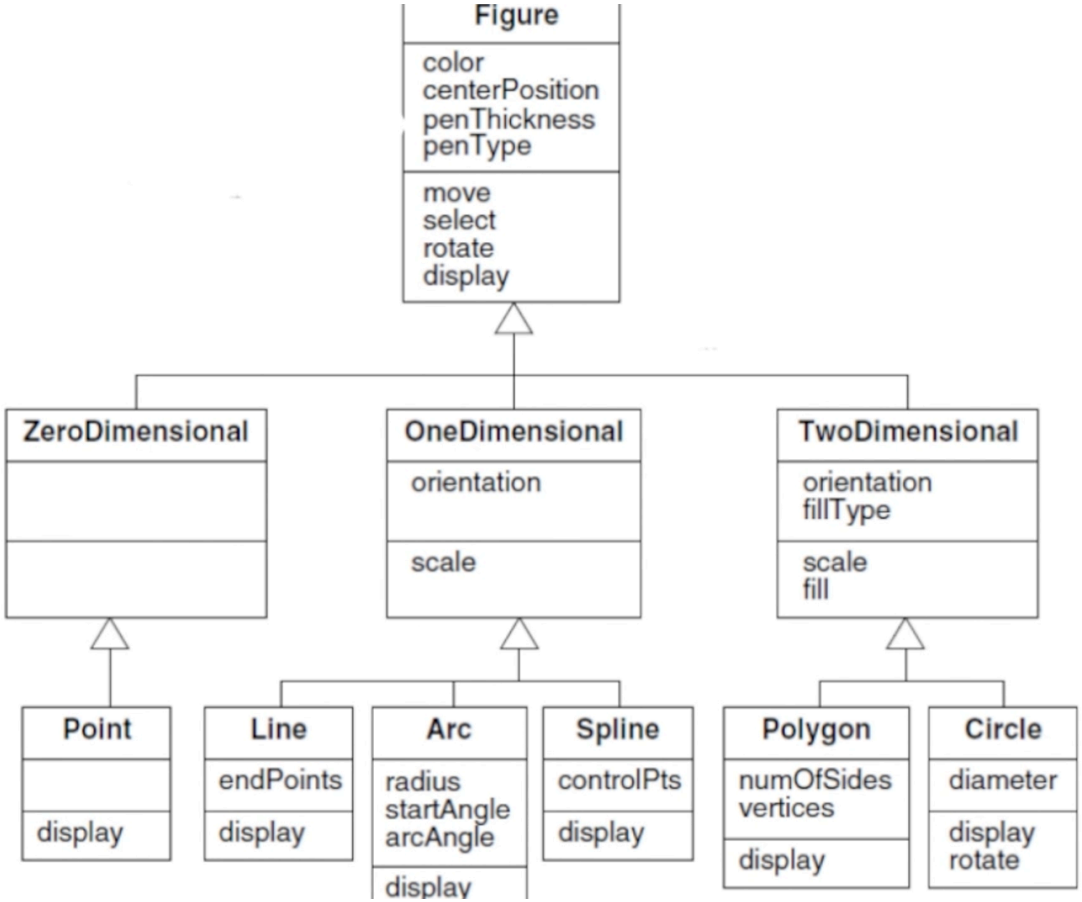
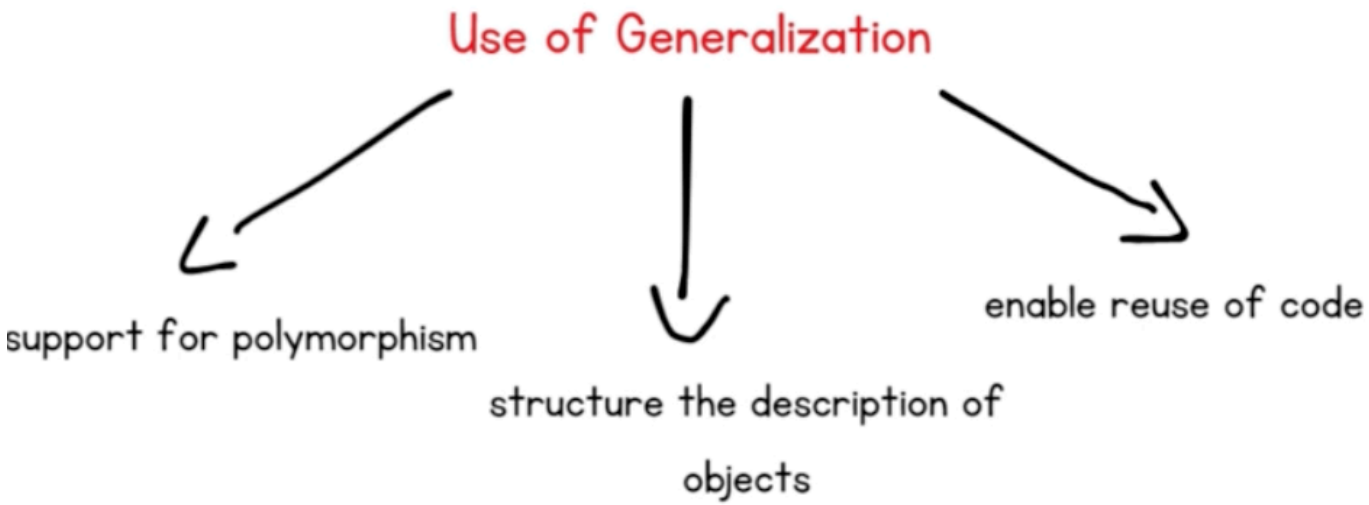


Figure
color
centerPosition
penThickness
penType
move
select
rotate
display
ZeroDimensional
OneDimensional
orientation
scale
TwoDimensional
orientation
fillType
scale
fill
Point
display
Line
endPoints
display
Arc
radius
startAngle
arcAngle
display
Spline
controlPts
display
Polygon
numOfSides
vertices
display
Circle
diameter
display
rotate

An inheritance hierarchy that is two or three levels deep is certainly acceptable;

ten levels deep is probably excessive;

five or six levels may or may not be proper.

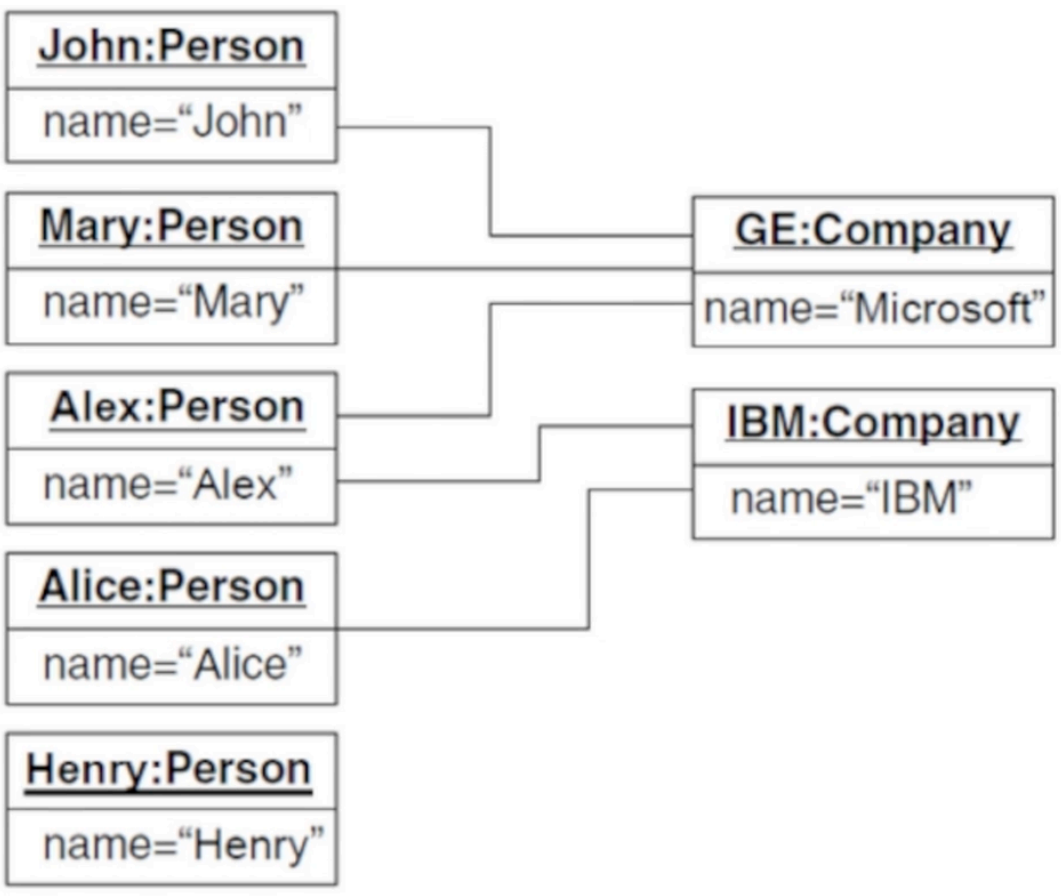

Use of Generalization
support for polymorphism
structure the description of objects
enable reuse of code

A link is a physical or conceptual connection among objects.

A link is an instance of an association.

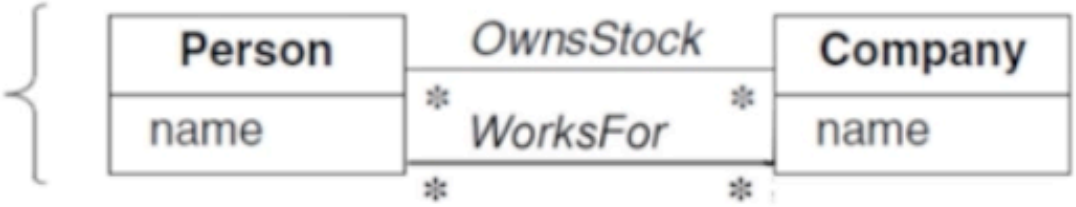
An association is a description of a group of links with common structure and common semantics.

Object diagram

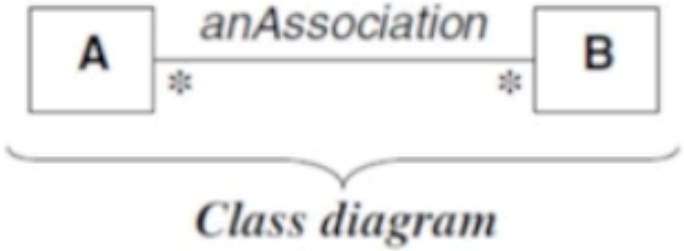


Class diagram

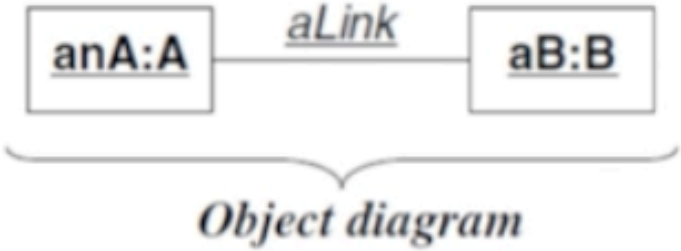
Class diagram



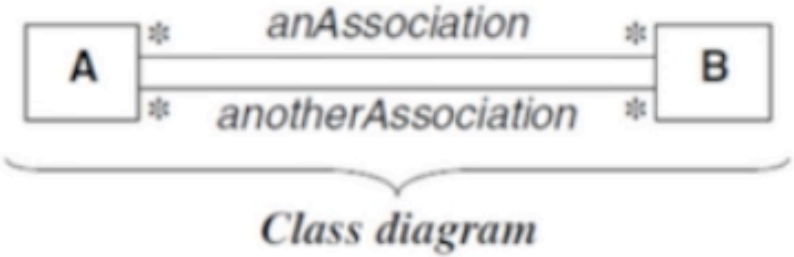
Class diagram



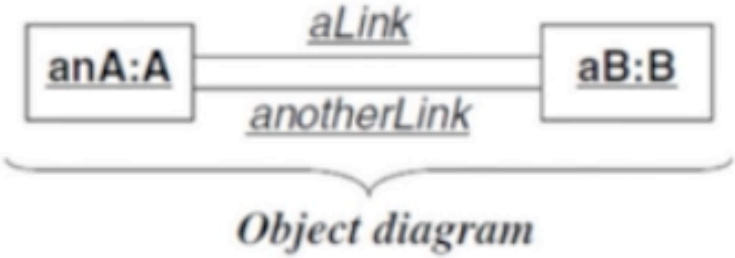
Class diagram



Object diagram



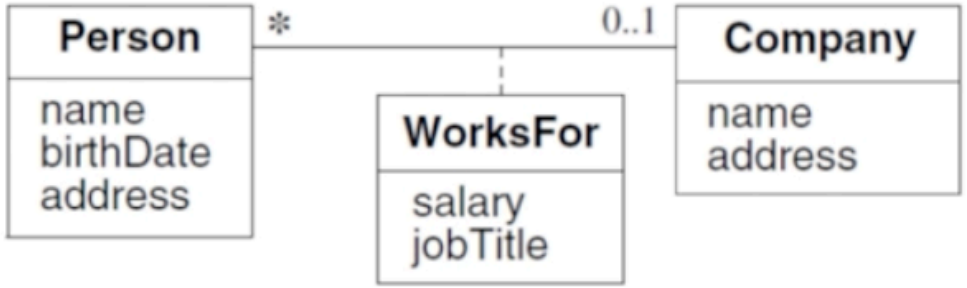
Class diagram



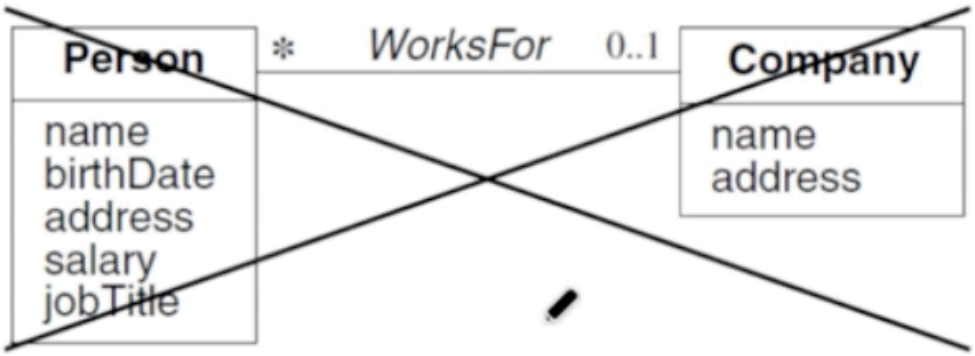
Object diagram

An association class is an association that is also a class.

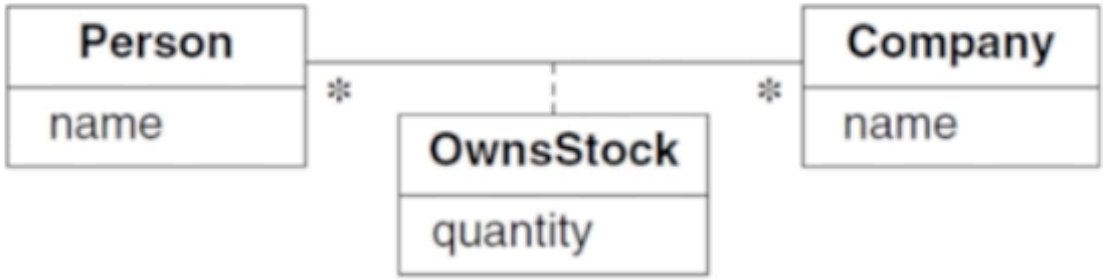
Preferred form



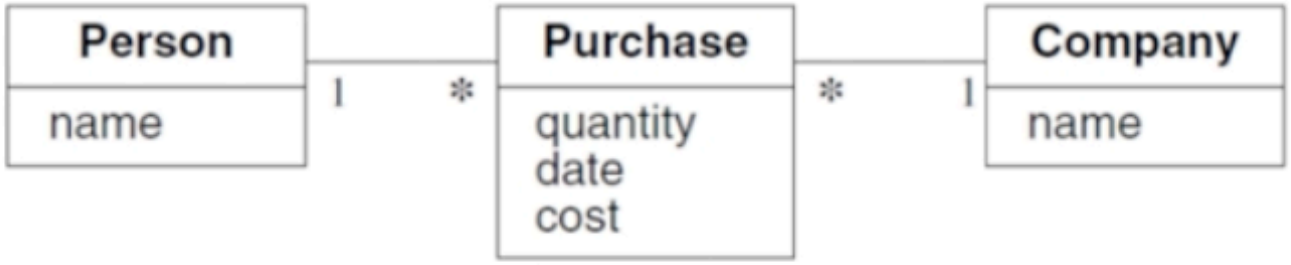
Discouraged form

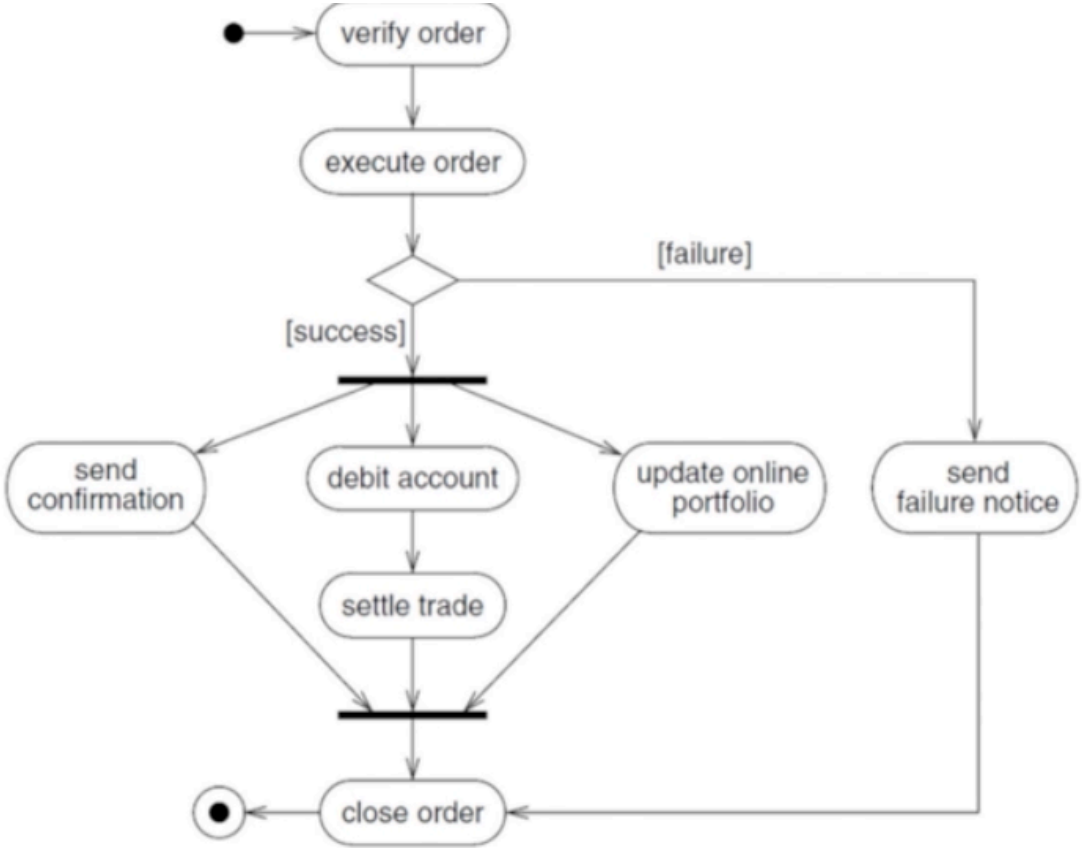


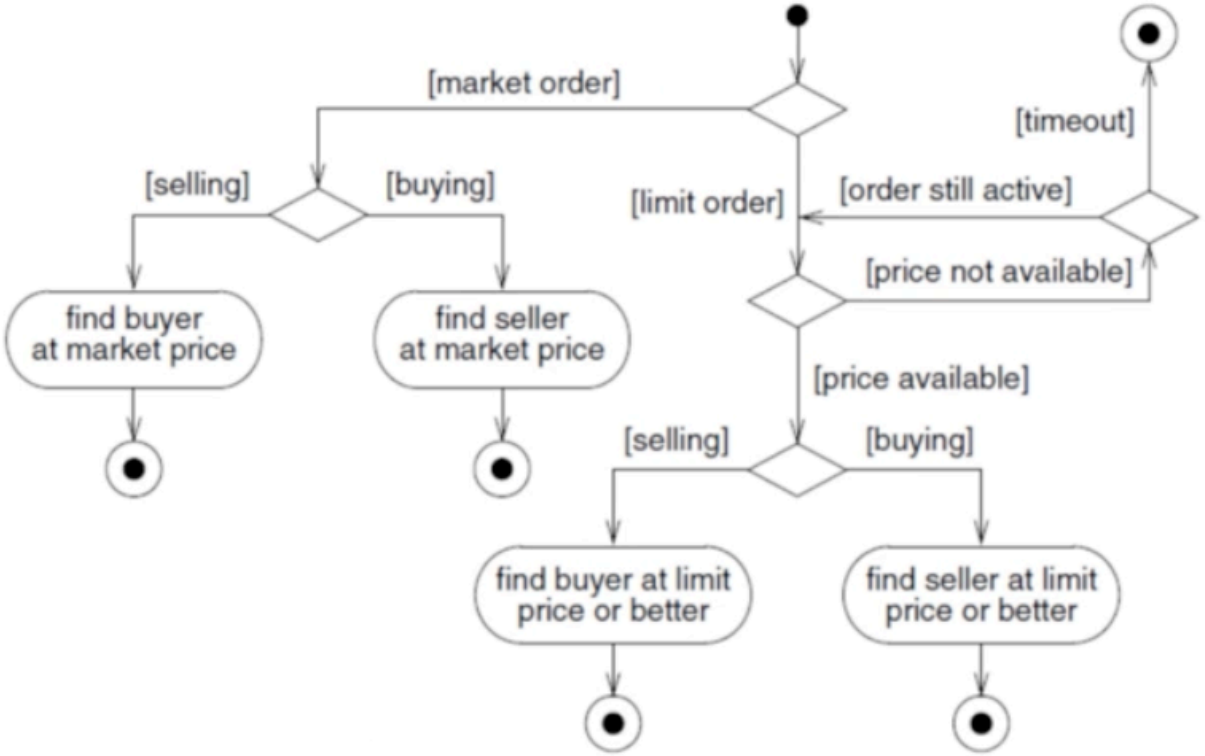
Association class

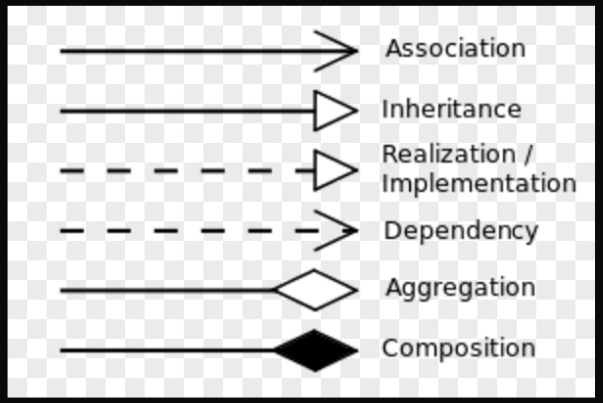


Ordinary class




verify order
execute order
[failure]
[success]
send
confirmation
debit account
update online
portfolio
send
failure notice
settle trade
close order

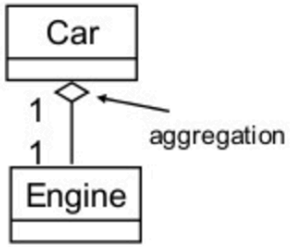

[market order]
[limit order]
[timeout]
[order still active]
[selling]
[buying]
[price not available]
find buyer
at market price
find seller
at market price
[price available]
[selling]
[buying]
find buyer at limit
price or better
find seller at limit
price or better


Association
Inheritance
Realization /
Implementation
Dependency
Aggregation
Composition

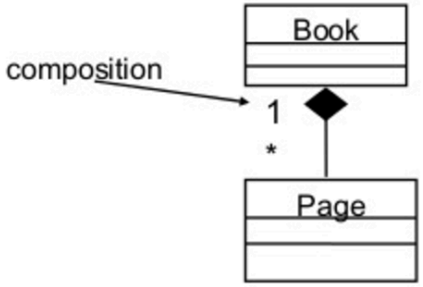
Association types

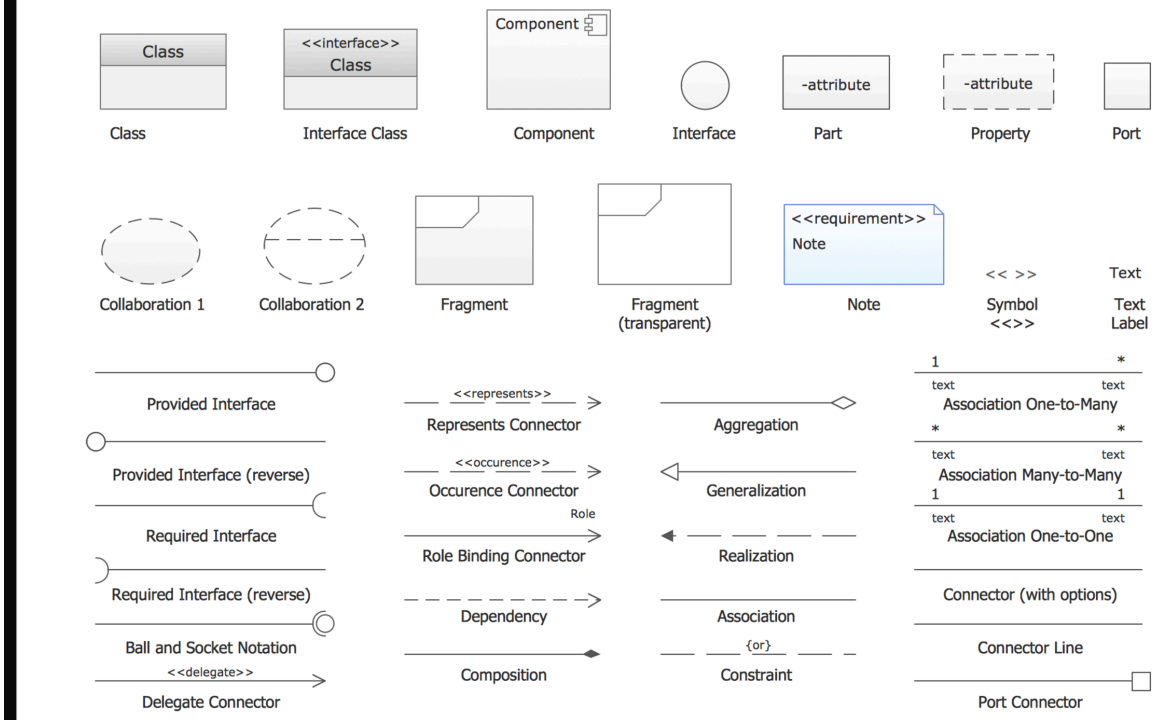
- **aggregation**: "is part of" / "has a"
  - symbolized by a clear white diamond

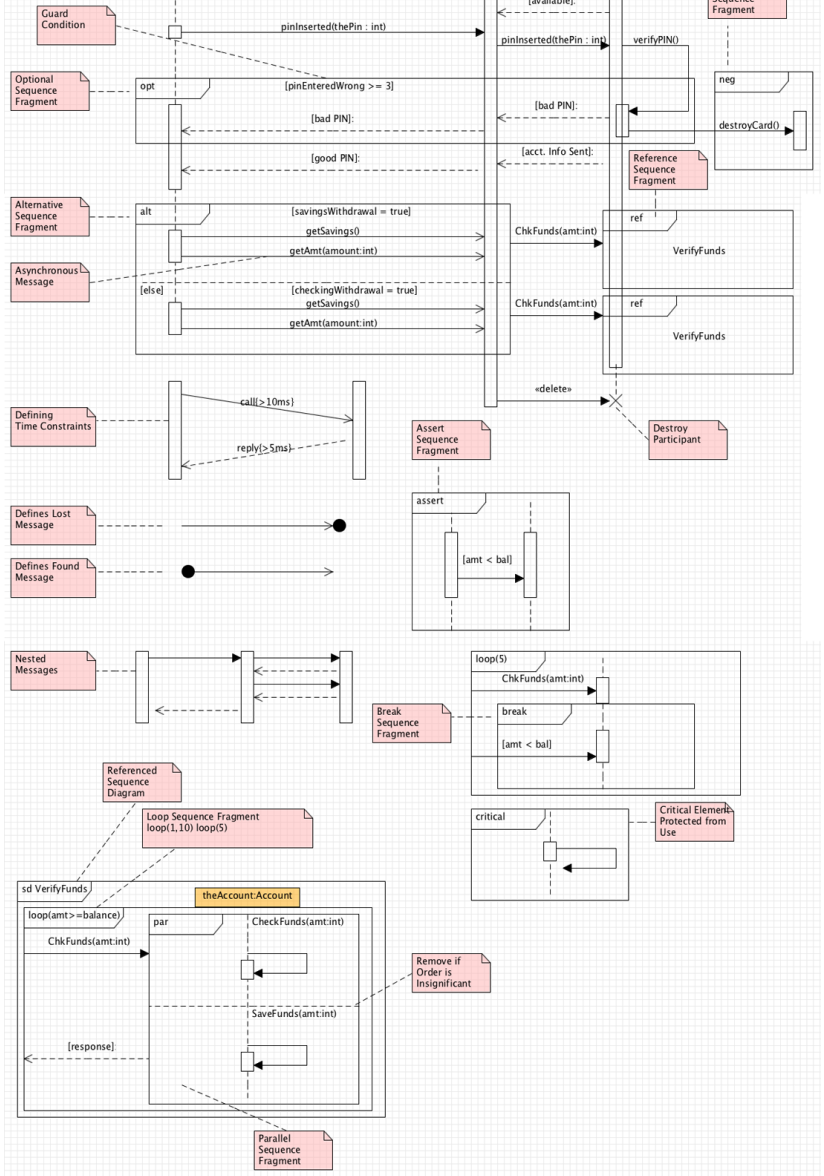
  * refers to the formation of a particular class as a result of one class being aggregated or built as a collection.

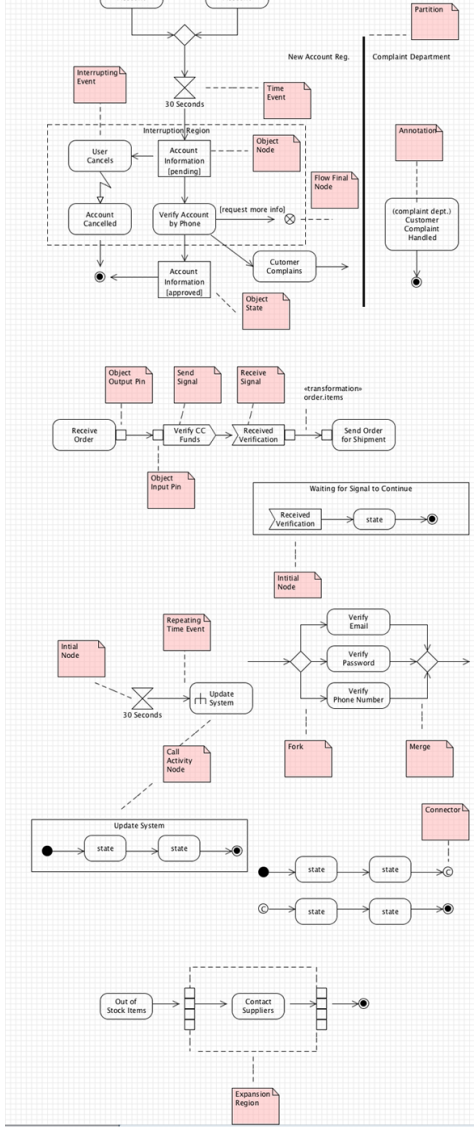


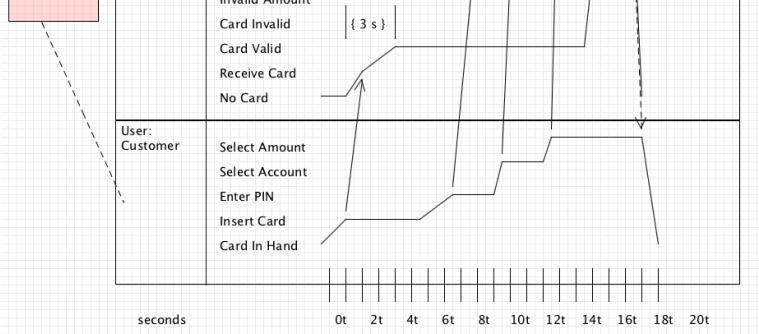
- **composition**: "is entirely made of"
  - stronger version of aggregation
  - the parts live and die with the whole
  - symbolized by a black diamond




Class
Class
<<interface>>
Class
Interface Class
Component
Component
Interface
-attribute
Part
-attribute
Property
Port
Collaboration 1
Collaboration 2
Fragment
Fragment
(transparent)
<<requirement>>
Note
Note
<< >>
Symbol
<<>>
Text
Text
Label
Provided Interface
Provided Interface (reverse)
Required Interface
Required Interface (reverse)
Ball and Socket Notation
<<delegate>>
Delegate Connector
<<represents>>
Represents Connector
<<occurence>>
Occurence Connector
Role
Role Binding Connector
Dependency
Composition
Aggregation
Generalization
Realization
Association
{or}
Constraint
1
*
text
text
Association One-to-Many
*
*
text
text
Association Many-to-Many
1
1
text
text
Association One-to-One
Connector (with options)
Connector Line
Port Connector

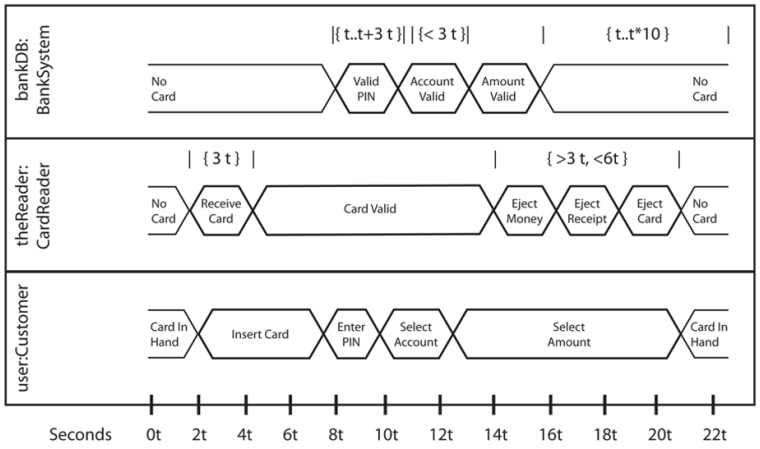

Guard Condition
pinInserted(thePin : int)
pinInserted(thePin : int)
verifyPIN()
Optional Sequence Fragment
opt
[pinEnteredWrong >= 3]
[bad PIN]:
[bad PIN]:
neg
destroyCard()
[good PIN]:
[acct. Info Sent]:
Reference Sequence Fragment
Alternative Sequence Fragment
alt
[savingsWithdrawal = true]
getSavings()
getAmt(amount:int)
ChkFunds(amt:int)
ref
VerifyFunds
Asynchronous Message
[else]
[checkingWithdrawal = true]
getSavings()
getAmt(amount:int)
ChkFunds(amt:int)
ref
VerifyFunds
«delete»
Defining Time Constraints
call[>10ms]
reply[>5ms]
Assert Sequence Fragment
Destroy Participant
assert
[amt < bal]
Defines Lost Message
Defines Found Message
Nested Messages
loop(5)
ChkFunds(amt:int)
Break Sequence Fragment
break
[amt < bal]
Referenced Sequence Diagram
Loop Sequence Fragment loop(1, 10) loop(5)
critical
Critical Element Protected from Use
sd VerifyFunds
theAccount:Account
loop(amt>=balance)
par
CheckFunds(amt:int)
ChkFunds(amt:int)
Remove if Order is Insignificant
SaveFunds(amt:int)
[response]:
Parallel Sequence Fragment


Partition
New Account Reg.
Complaint Department
Interrupting Event
30 Seconds
Time Event
Interruption Region
User Cancels
Account Information [pending]
Object Node
Flow Final Node
Annotation
Account Cancelled
Verify Account by Phone
[request more info]
(complaint dept.) Customer Complaint Handled
Account Information [approved]
Cutomer Complains
Object State
Object Output Pin
Send Signal
Receive Signal
«transformation» order.items
Receive Order
Verify CC Funds
Received Verification
Send Order for Shipment
Object Input Pin
Waiting for Signal to Continue
Received Verification
state
Initial Node
Repeating Time Event
Intial Node
30 Seconds
Update System
Verify Email
Verify Password
Verify Phone Number
Call Activity Node
Fork
Merge
Connector
Update System
state
state
state
state
state
state
Out of Stock Items
Contact Suppliers
Expansion Region

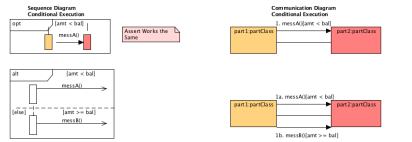


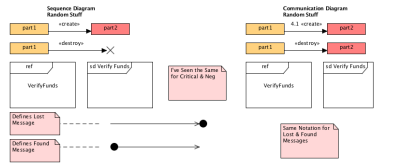
Steps of Execution
1. Customer Inserts Card
1A. Card is Invalid
1B. Eject Card
2. Card is Validated
3. Customer Enters PIN
3A. PIN is Invalid
4. PIN is Validated
5. Account is Selected
5A. Account is Valid
5B. Account is Invalid
6. Amount is Selected
6A. Amount is Valid
6B. Amount is Invalid
7. Eject Money
8. Eject Receipt
9. Eject Card

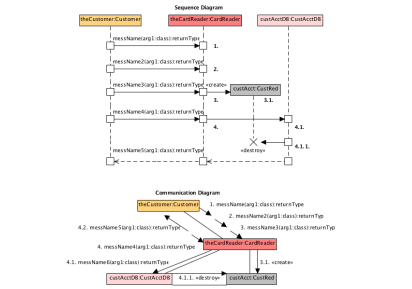
Timing Constraints
{ t } : Same as value of t
{ 3t } : 3 of the time unit
{ <3t } : Less than 3
{ >3s, <6s } : Greater than 3, Less than 6
{ t..t*3 } : Up to 3 times t
{ 3t..6t } : 3 to 6 t

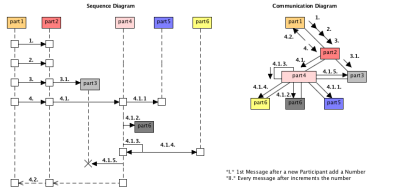


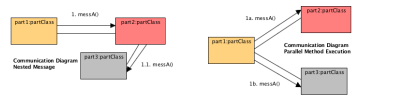

Sequence Diagram
Execute Multiple Times
loop(10)
messA()
Communication Diagram
Execute Multiple Times
part1:partClass
1: messA()"i=0..9"
part2:partClass

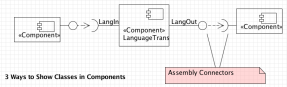

Sequence Diagram
Conditional Execution
opt
[amt < bal]
messA()
Assert Works the Same
alt
[amt < bal]
messA()
[else]
[amt >= bal]
messB()
Communication Diagram
Conditional Execution
1: messA()[amt < bal]
part1:partClass
part2:partClass
1a. messA()[amt < bal]
1b. messB()[amt >= bal]


Sequence Diagram
Random Stuff
part1
part2
«create»
«destroy»
ref
VerifyFunds
sd Verify Funds
I've Seen the Same for Critical & Neg
Defines Lost Message
Defines Found Message
Communication Diagram
Random Stuff
4.1 «create»
Same Notation for Lost & Found Messages


Sequence Diagram
theCustomer:Customer
theCardReader:CardReader
custAcctDB:CustAcctDB
messName(arg1: class): returnType
messName2(arg1: class): returnType
messName3(arg1: class): returnType
«create»
custAcct:CustRcd
messName4(arg1: class): returnType
messName5(arg1: class): returnType
«destroy»
Communication Diagram
1. messName(arg1: class): returnType
2. messName2(arg1: class): returnTyp
3. messName3(arg1: class): returnTyp
4.2. messName5(arg1: class): returnType
4. messName4(arg1: class): returnType
4.1. messName6(arg1: class): returnType
3.1. «create»
4.1.1. «destroy»

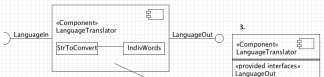

Sequence Diagram
part1
part2
part4
part3
part5
part6
Communication Diagram
1. 1st Message after a new Participant add a Number
#. Every message after increments the number


part1:partClass
1. messA()
part2:partClass
Communication Diagram
Nested Message
part3:partClass
1.1. messA()
1a. messA()
1b. messA()
Communication Diagram
Parallel Method Execution

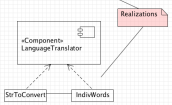


3 Ways to Show Classes in Components

1.

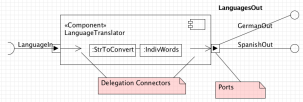


2.

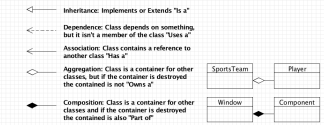


3.

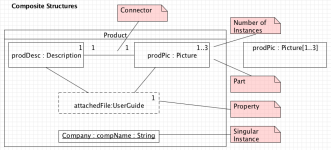
Ports & Delegation Connectors

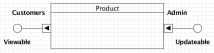


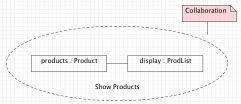
Review

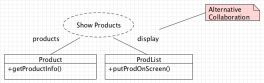


Composite Structures

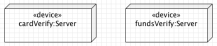




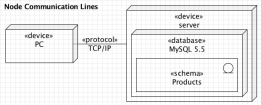




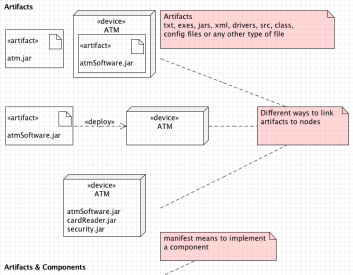
Named Nodes



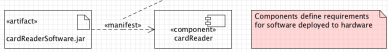
Node Communication Lines



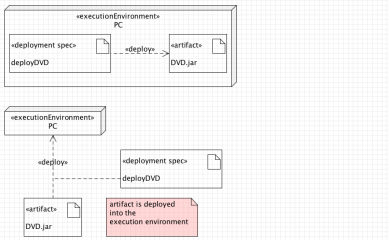
Artifacts



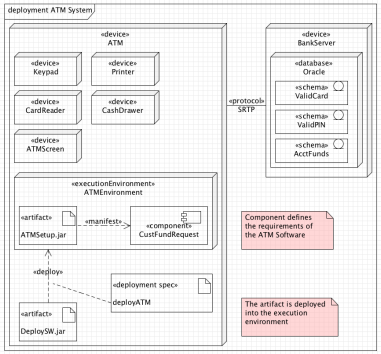
Artifacts & Components



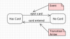
Deployment Specifications



Specification Deployment Diagram

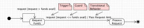


STATE MACHINE DIAGRAMS

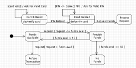


State / Trigger	Card Entered	Eject Card
No Card	Has Card	-
Has Card	-	No Card

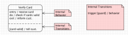
Transition Arrows



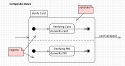
State Machine Diagrams



State Internal Behavior

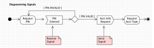


Composite States

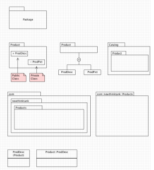


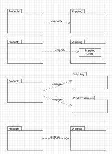
Choice Pseudostates

Diagramming Signals



Package Diagrams





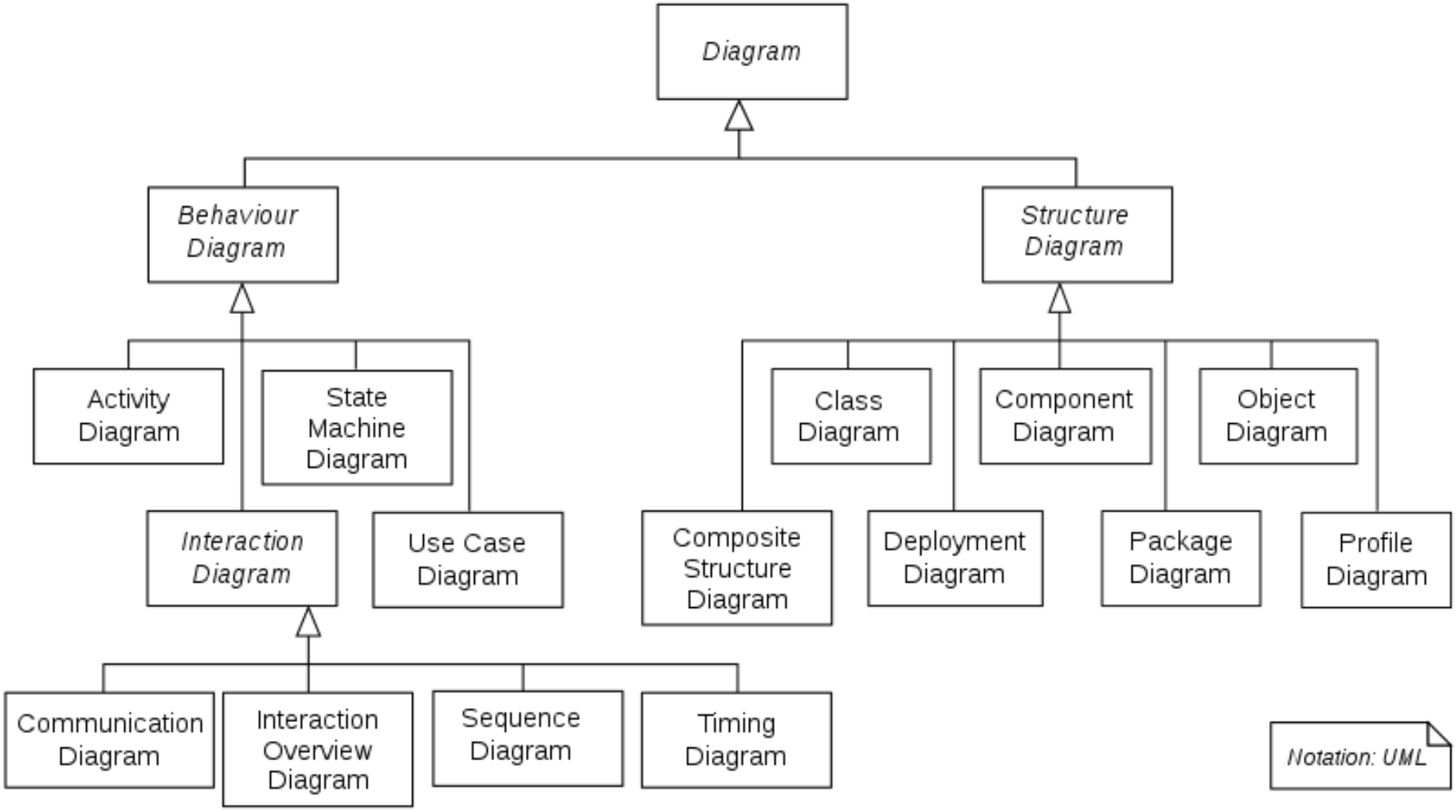
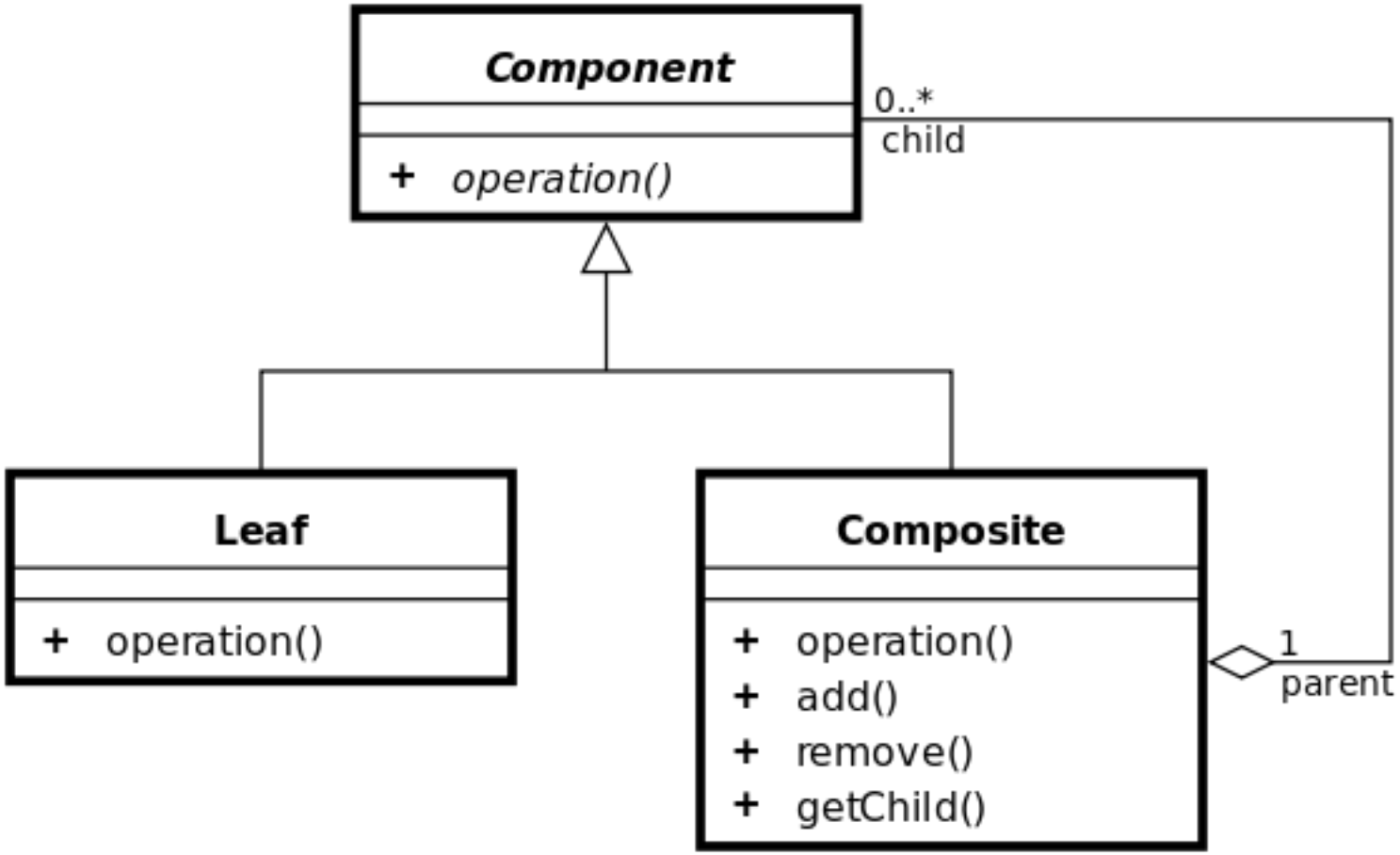
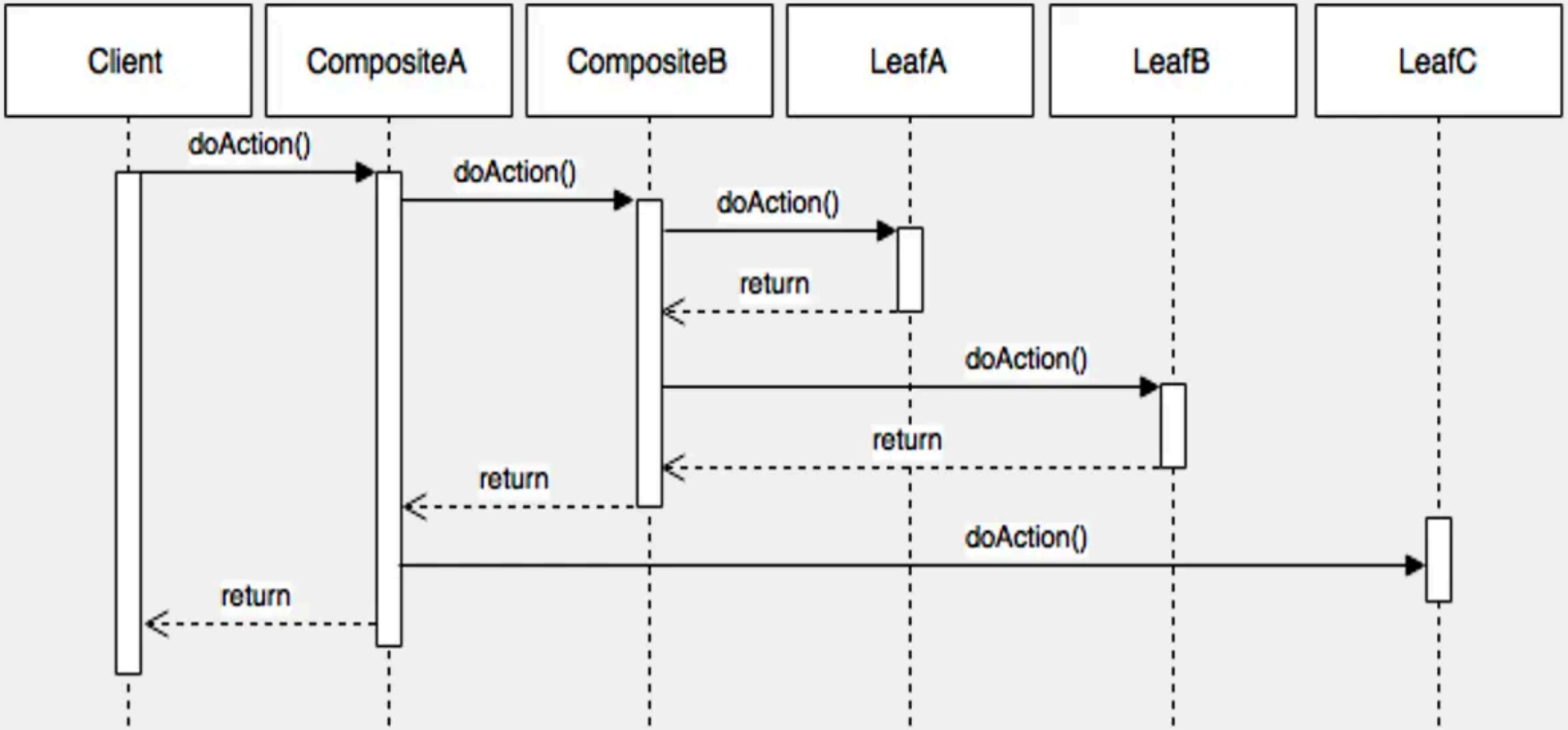
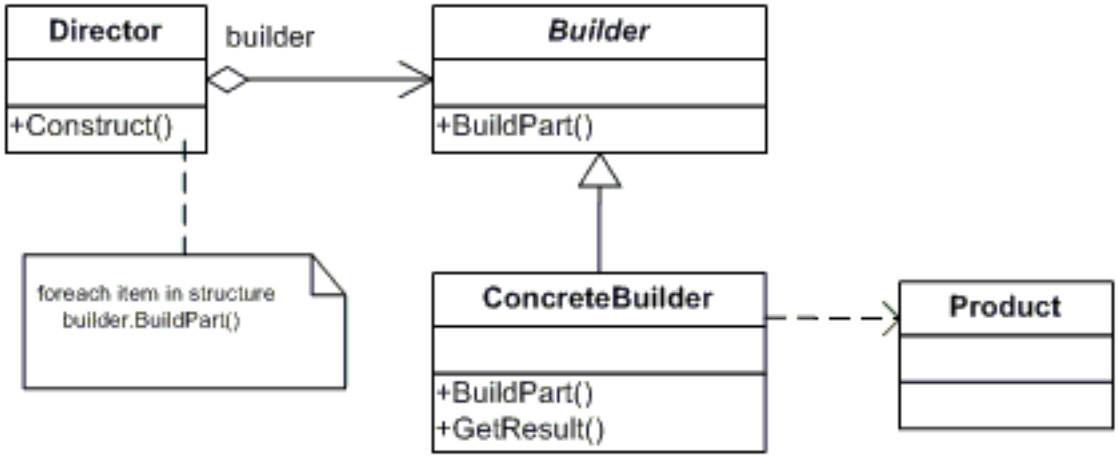


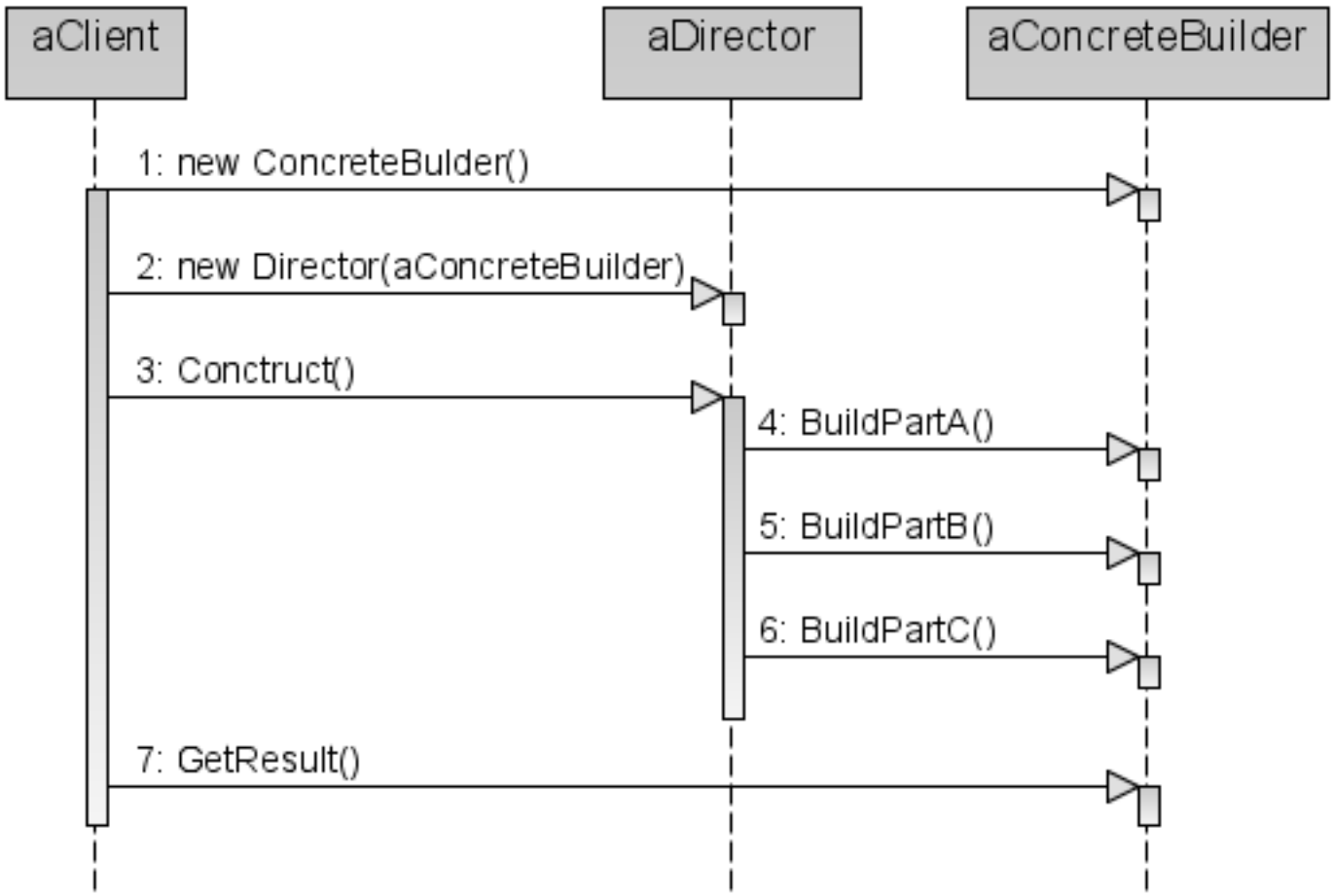
Diagram
Behaviour Diagram
Structure Diagram
Activity Diagram
State Machine Diagram
Interaction Diagram
Use Case Diagram
Class Diagram
Component Diagram
Object Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Profile Diagram
Communication Diagram
Interaction Overview Diagram
Sequence Diagram
Timing Diagram
Notation: UML

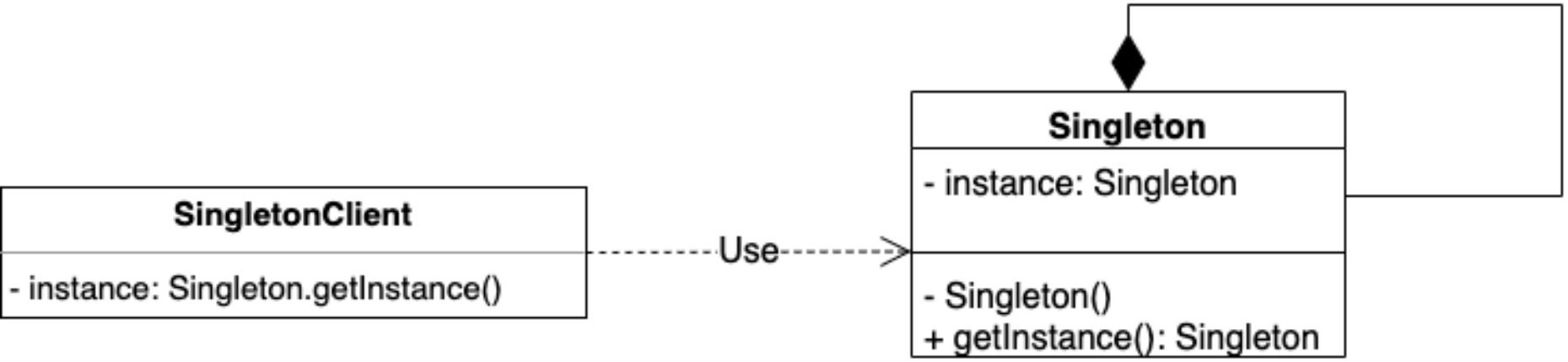

Component
+ operation()
0..*
child
Leaf
+ operation()
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent

Composite pattern – Diagram of sequence




Director
builder
Builder
+Construct()
+BuildPart()
foreach item in structure
builder.BuildPart()
ConcreteBuilder
Product
+BuildPart()
+GetResult()


aClient
aDirector
aConcreteBuilder
1: new ConcreteBulder()
2: new Director(aConcreteBuilder)
3: Conctruct()
4: BuildPartA()
5: BuildPartB()
6: BuildPartC()
7: GetResult()


Singleton
- instance: Singleton
- Singleton()
+ getInstance(): Singleton
SingletonClient
- instance: Singleton.getInstance()
Use

Types

- Creational
  - ➢Singleton
  - ➢Factory
  - ➢Abstract Factory
  - ➢Builder
  - ➢Prototype
- Structural
  - ➢Adapter
  - ➢Bridge
  - ➢Composite
  - ➢Decorator
  - ➢Façade
  - ➢Flyweight
  - ➢Proxy
- Behavioral
  - ➢ Chain of Responsibility
  - ➢ Command
  - ➢ Interpreter
  - ➢ Iterator
  - ➢ Mediator
  - ➢ Memento
  - ➢ Observer
  - ➢ State
  - ➢ Strategy
  - ➢ Template Method
  - ➢ Visitor