Low Level Design (LLD)

Part 4

Object Relationships

1. Composition
2. Aggregation
3. Association

Composition

To qualify as a composition, an object and a part must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can only belong to one object (class) at a time
- The part (member) has its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

Composition

A good real-life example of a composition is the relationship between a person’s body and a heart.

Aggregation

To qualify as an aggregation, a whole object and its parts must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can (if desired) belong to more than one object (class) at a time
- The part (member) does not have its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

Aggregation

Consider the relationship between a person and their home address. In this example, for simplicity, we’ll say every person has an address. However, that address can belong to more than one person at a time.

Association

To qualify as an association, an object and another object must have the following relationship:

- The associated object (member) is otherwise unrelated to the object (class)
- The associated object (member) can belong to more than one object (class) at a time
- The associated object (member) does not have its existence managed by the object (class)
- The associated object (member) may or may not know about the existence of the object (class)

Association

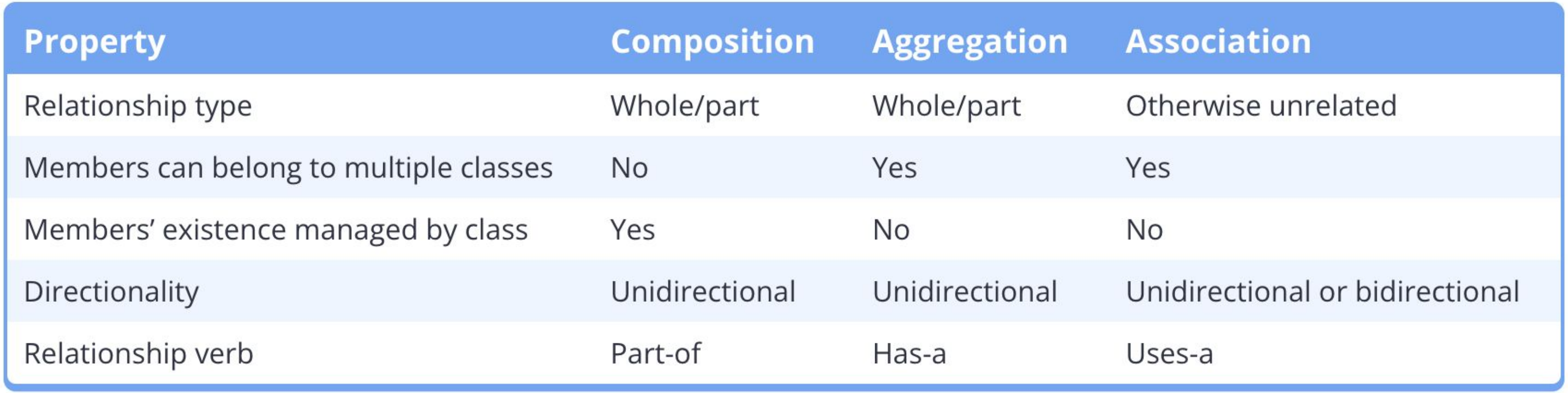
To qualify as an association, an object and another object must have the following relationship:

- The associated object (member) is otherwise unrelated to the object (class)
- The associated object (member) can belong to more than one object (class) at a time
- The associated object (member) does not have its existence managed by the object (class)
- The associated object (member) may or may not know about the existence of the object (class)

Association

The relationship between doctors and patients is a great example of an association. The doctor clearly has a relationship with his patients, but conceptually it’s not a part/whole (object composition) relationship. A doctor can see many patients in a day, and a patient can see many doctors (perhaps they want a second opinion, or they are visiting different types of doctors). Neither of the object’s lifespans are tied to the other.

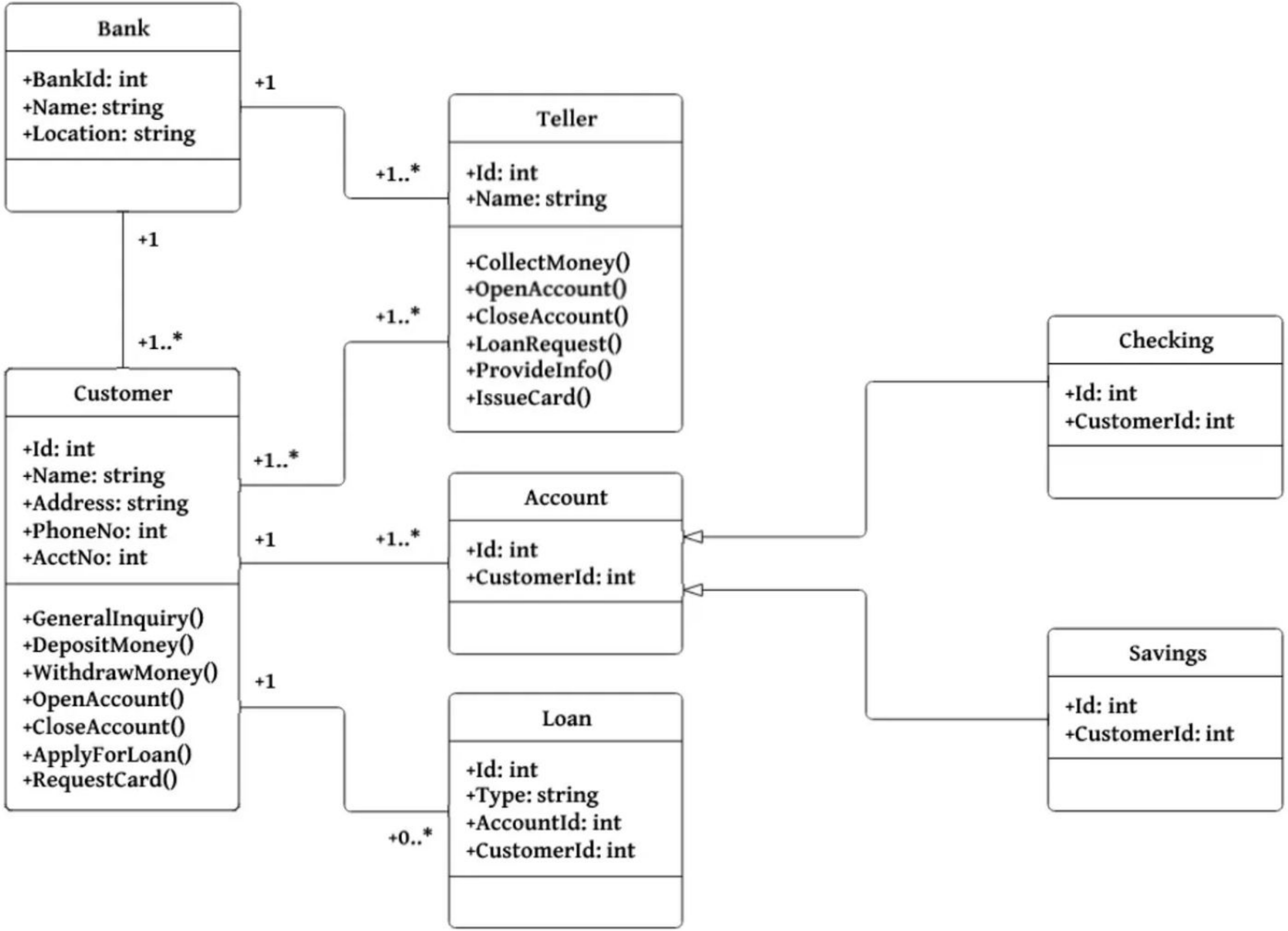
Object Relationships



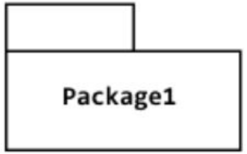
Property	Composition	Aggregation	Association
Relationship type	Whole/part	Whole/part	Otherwise unrelated
Members can belong to multiple classes	No	Yes	Yes
Members' existence managed by class	Yes	No	No
Directionality	Unidirectional	Unidirectional	Unidirectional or bidirectional
Relationship verb	Part-of	Has-a	Uses-a

UML Class Diagram

Class diagrams are a neat way of visualizing the classes in your system before you actually start coding them up. They’re a static representation of your system structure.

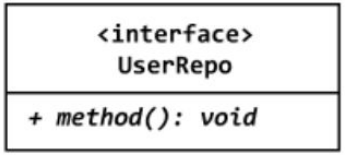

Bank
+BankId: int
+Name: string
+Location: string
+1
+1..*
Teller
+Id: int
+Name: string
+CollectMoney()
+OpenAccount()
+CloseAccount()
+LoanRequest()
+ProvideInfo()
+IssueCard()
+1
+1..*
+1..*
+1..*
Customer
+Id: int
+Name: string
+Address: string
+PhoneNo: int
+AcctNo: int
+GeneralInquiry()
+DepositMoney()
+WithdrawMoney()
+OpenAccount()
+CloseAccount()
+ApplyForLoan()
+RequestCard()
Checking
+Id: int
+CustomerId: int
+1
+1..*
Account
+Id: int
+CustomerId: int
Savings
+Id: int
+CustomerId: int
+1
Loan
+Id: int
+Type: string
+AccountId: int
+CustomerId: int
+0..*

UML Class Diagram



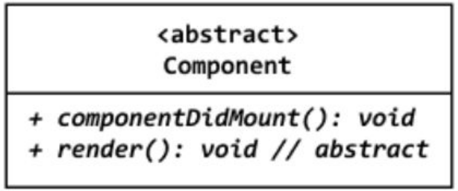
Package

A collection of classes and interfaces.



Interface

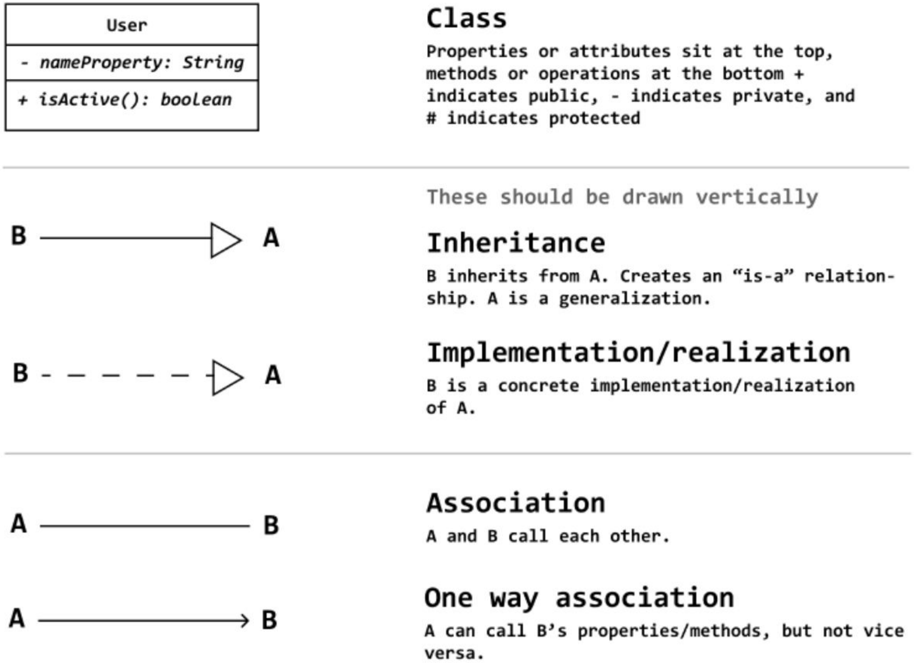
Interface name written underneath the <interface> annotation. Methods underneath.



Abstract class

Same as the interface shape. Abstract methods marked as abstract with comments or “abstract methodName(): returnType”.

UML Class Diagram



Class

Properties or attributes sit at the top, methods or operations at the bottom + indicates public, - indicates private, and # indicates protected

These should be drawn vertically

Inheritance

B inherits from A. Creates an "is-a" relationship. A is a generalization.

Implementation/realization

B is a concrete implementation/realization of A.

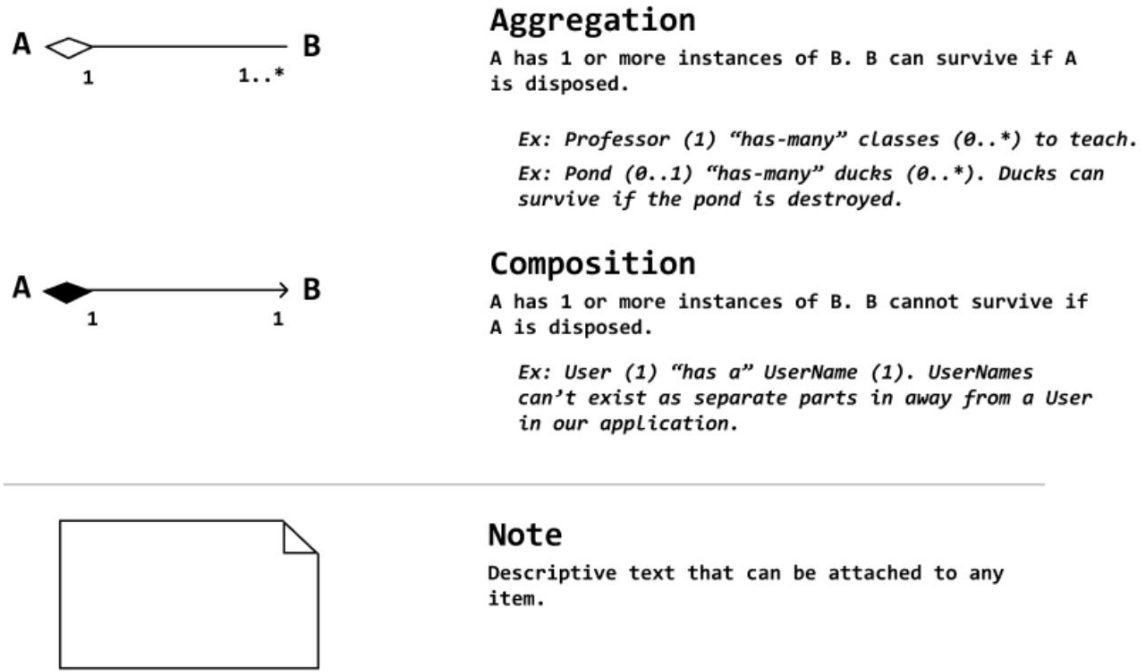
Association

A and B call each other.

One way association

A can call B's properties/methods, but not vice versa.

UML Class Diagram



Aggregation

A has 1 or more instances of B. B can survive if A is disposed.

Ex: Professor (1) “has-many” classes (0..) to teach.*

Ex: Pond (0..1) “has-many” ducks (0..). Ducks can survive if the pond is destroyed.*

Composition

A has 1 or more instances of B. B cannot survive if A is disposed.

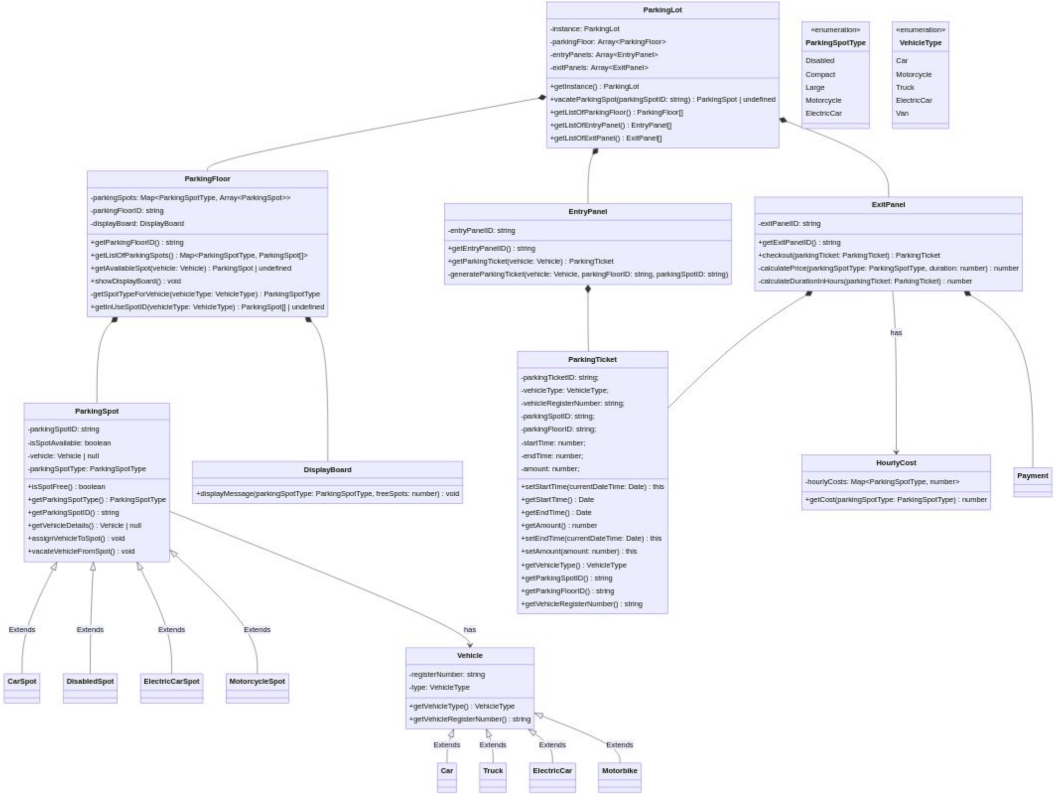
Ex: User (1) “has a” UserName (1). UserNames can’t exist as separate parts in away from a User in our application.

Note

Descriptive text that can be attached to any item.

Design Parking Lot

Design Parking Lot



```
ParkingLot
-instance: ParkingLot
-parkingFloor: Array<ParkingFloor>
-entryPanels: Array<EntryPanel>
-exitPanels: Array<ExitPanel>
+getInstance() : ParkingLot
+vacateParkingSpot(parkingSpotID: string) : ParkingSpot | undefined
+getListOfParkingFloor() : ParkingFloor[]
+getListOfEntryPanel() : EntryPanel[]
+getListOfExitPanel() : ExitPanel[]

«enumeration»
ParkingSpotType
Disabled
Compact
Large
Motorcycle
ElectricCar

«enumeration»
VehicleType
Car
Motorcycle
Truck
ElectricCar
Van

ParkingFloor
-parkingSpots: Map<ParkingSpotType, Array<ParkingSpot>>
-parkingFloorID: string
-displayBoard: DisplayBoard
+getParkingFloorID() : string
+getListOfParkingSpots() : Map<ParkingSpotType, ParkingSpot[]>
+getAvailableSpot(vehicle: Vehicle) : ParkingSpot | undefined
+showDisplayBoard() : void
-getSpotTypeForVehicle(vehicleType: VehicleType) : ParkingSpotType
+getInUseSpotID(vehicleType: VehicleType) : ParkingSpot[] | undefined

EntryPanel
-entryPanelID: string
+getEntryPanelID() : string
+getParkingTicket(vehicle: Vehicle) : ParkingTicket
-generateParkingTicket(vehicle: Vehicle, parkingFloorID: string, parkingSpotID: string)

ExitPanel
-exitPanelID: string
+getExitPanelID() : string
+checkout(parkingTicket: ParkingTicket) : ParkingTicket
-calculatePrice(parkingSpotType: ParkingSpotType, duration: number) : number
-calculateDurationInHours(parkingTicket: ParkingTicket) : number

ParkingSpot
-parkingSpotID: string
-isSpotAvailable: boolean
-vehicle: Vehicle | null
-parkingSpotType: ParkingSpotType
+isSpotFree() : boolean
+getParkingSpotType() : ParkingSpotType
+getParkingSpotID() : string
+getVehicleDetails() : Vehicle | null
+assignVehicleToSpot() : void
+vacateVehicleFromSpot() : void

DisplayBoard
+displayMessage(parkingSpotType: ParkingSpotType, freeSpots: number) : void

ParkingTicket
-parkingTicketID: string;
-vehicleType: VehicleType;
-vehicleRegisterNumber: string;
-parkingSpotID: string;
-parkingFloorID: string;
-startTime: number;
-endTime: number;
-amount: number;
+setStartTime(currentDateTime: Date) : this
+getStartTime() : Date
+getEndTime() : Date
+getAmount() : number
+setEndTime(currentDateTime: Date) : this
+setAmount(amount: number) : this
+getVehicleType() : VehicleType
+getParkingSpotID() : string
+getParkingFloorID() : string
+getVehicleRegisterNumber() : string

HourlyCost
-hourlyCosts: Map<ParkingSpotType, number>
+getCost(parkingSpotType: ParkingSpotType) : number

Payment

CarSpot

DisabledSpot

ElectricCarSpot

MotorcycleSpot

Vehicle
-registerNumber: string
-type: VehicleType
+getVehicleType() : VehicleType
+getVehicleRegisterNumber() : string

Car

Truck

ElectricCar

Motorbike
```

S.O.L.I.D. Principles

These five software development principles are guidelines to follow when building software so that it is easier to scale and maintain. They were made popular by a software engineer, Robert C. Martin.

S — Single Responsibility

A class should have a single responsibility

O — Open-Closed

Classes should be open for extension, but closed for modification

L — Liskov Substitution

If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.

I — Interface Segregation

Clients should not be forced to depend on methods that they do not use.

D — Dependency Inversion

High-level modules should not depend on low-level modules. Both should depend on the abstraction.

Abstractions should not depend on details. Details should depend on abstractions.

Thank You!