COMS W4156 Advanced Software Engineering (ASE)

November 18, 2021

shared google doc for discussion during class

Demos !

SAVS

AAPI

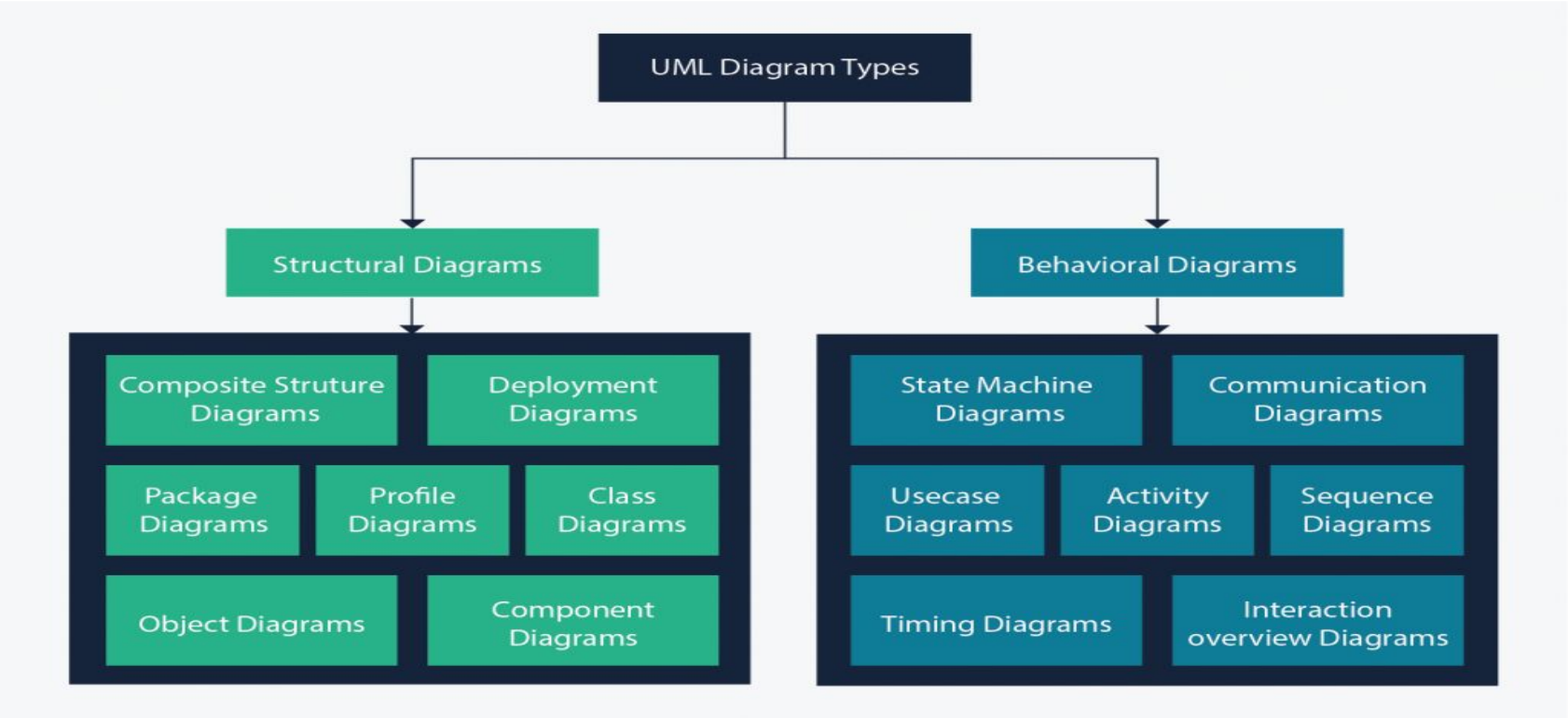
UML

UML = Unified Modeling Language

Called “unified” because before UML there were three different competing modeling languages. The three main proponents of those competitors got together to jointly produce UML in 1995 (they were all hired away from wherever they worked before to the same company, Rational Software, now owned by IBM)

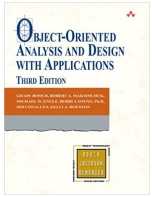
Adopted as a “standard” by ISO (International Organization for Standardization) in 2005, revised most recently in 2017 as UML 2.5.1

The UML “Language” Consists Entirely of Diagrams



What is the Language Modeling?

UML is a graphical notation that visually “models” software artifacts and processes - blueprints for software engineers



Used for “object-oriented analysis and design” (OOAD)

CRC (Component/Class Responsibilities Collaborators) is the simplified “agile” notion of OOAD, see October 5th lecture notes

Software engineers working on government, defense, or safety-critical software will probably use more sophisticated OOAD and most or all of the UML diagrams

Some tools generate code from UML and vice versa, “round-trip engineering”

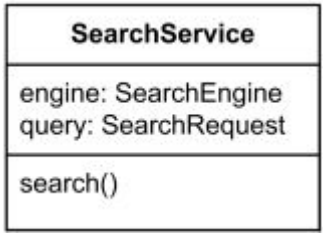
Class Diagrams

Class diagrams most common diagrams used in UML and most likely to be encountered in software designs and documentation concerned with conventional business and consumer applications

Show the classes of the system, their interrelationships, and the operations and attributes of the classes

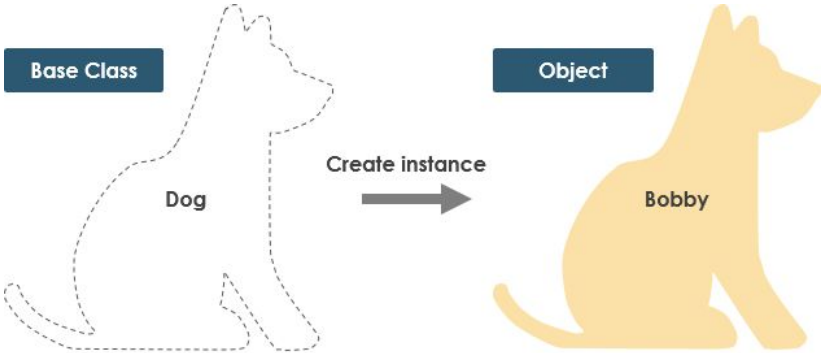
Convey static structure not dynamic behavior

Same kind of information as CRC cards but with more detail



Similar to E-R diagrams (from relational databases)

Class Represents Type or Singleton



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

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

A **class** in a class diagram can be a type of object (that can be instantiated many times) or a singleton object (instantiated at most once)

Applies to non-OO languages, however the language defines data types and singleton data entities

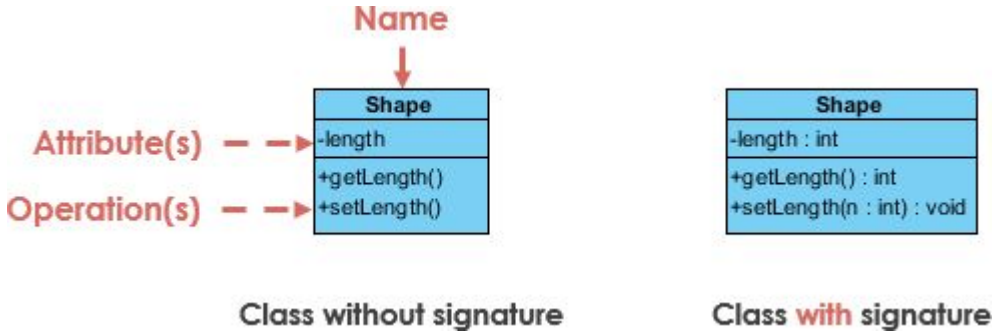
Attributes are properties or state of the object, may be primitive values, data structures, or references to collaborators

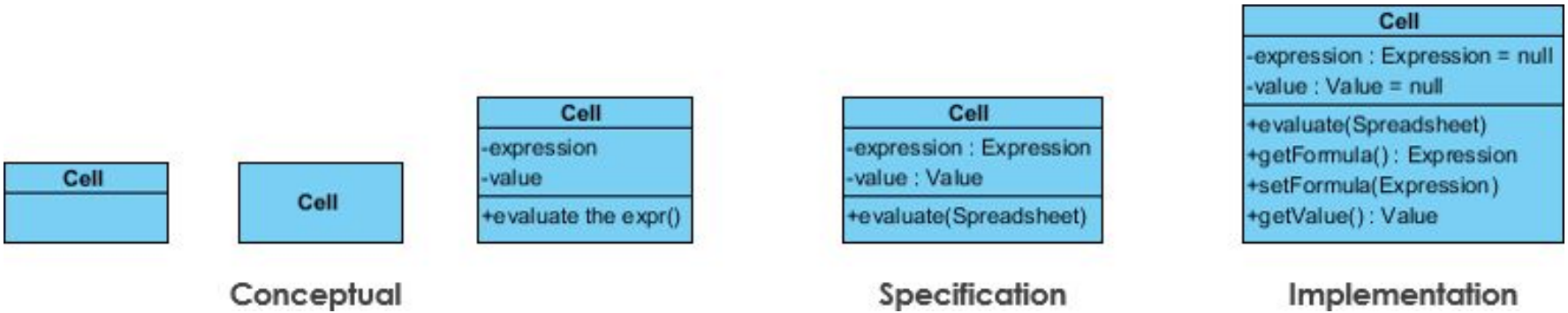
Operations are responsibilities or behaviors, often omit CRUD (create, read, update, delete)

Varying Detail

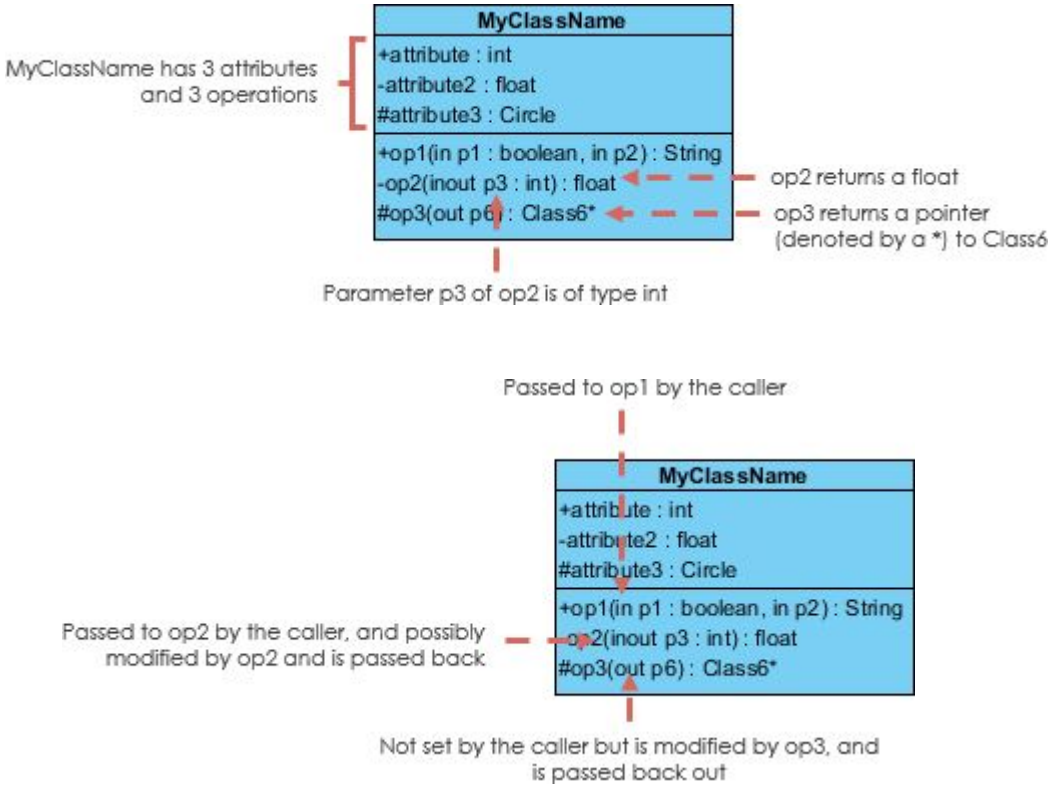
Class diagrams show varying amounts of detail, class name is only mandatory part

Sometimes include **signatures** (parameter types, return type)





More Detail



Sometimes indicates **visibility** (+ public, - private, # protected)

Sometimes indicates parameter **directionality** (in, out, inout)

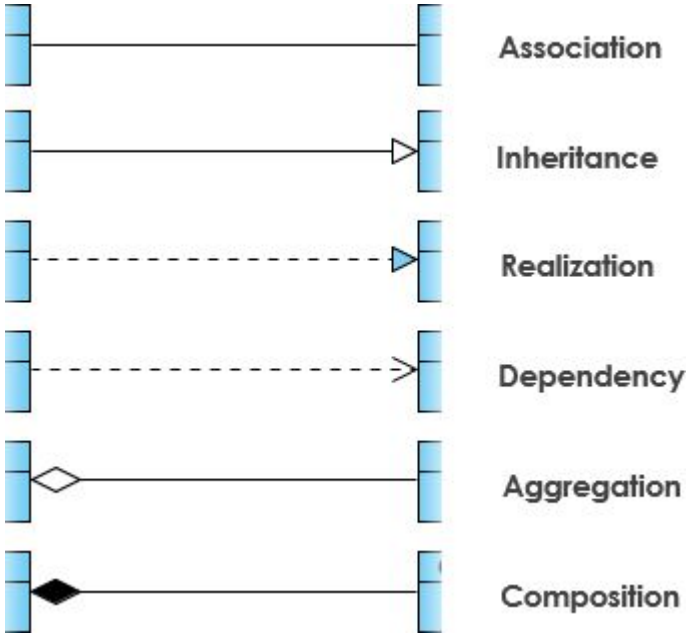
Names of **static** attributes and operations are underlined

SearchService
- config: Configuration - engine: SearchEngine
+ search(query: SearchRequest): SearchResult - createEngine(): SearchEngine

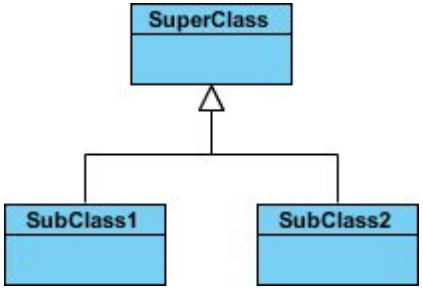
Relationships among Classes and Instances of Classes

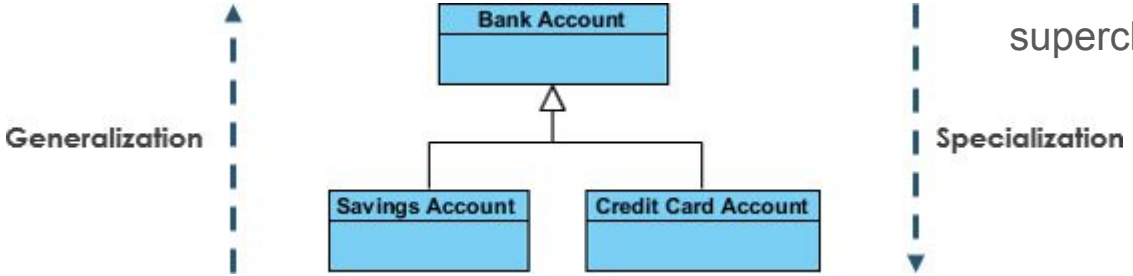
Some class diagrams show the structure of a single class, with its attributes and operations, while others focus on the *relationships* among classes

On CRC cards, *any* kind of relationship is fine for “collaborators”, but class diagrams refine the kind of relationship



Generalization





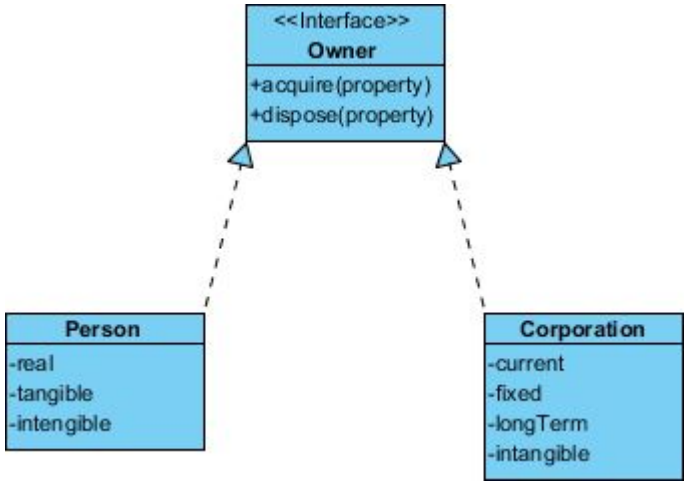
Generalization or Inheritance is a taxonomic relationship corresponding to IS-A

Between more general superclass (abstraction, base class) and more specific subclass (specialization, extension, derived class)

Applies even to non-OO languages - generalized types and specialized types

Solid line with open/hollow (unfilled) arrowhead points from subclass to superclass

Realization



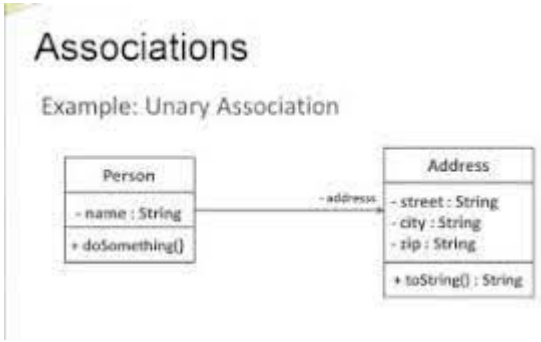
Realization or Implementation of an interface

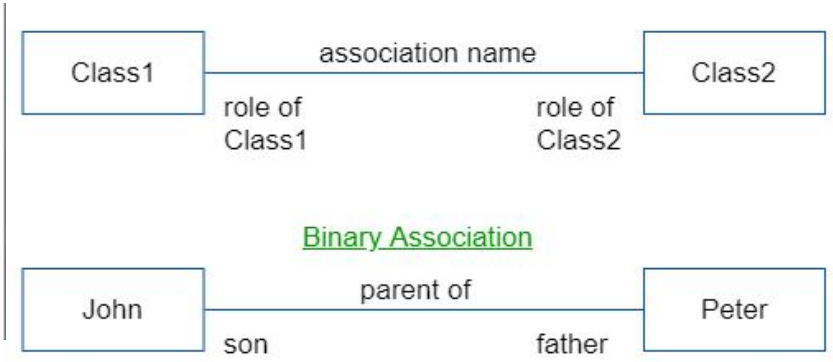
Applies even to languages without notion of “interface” - conceptual, not necessarily explicit in the code

Dashed line with open arrowhead (triangle) from implementing class to interface

Like class but with << interface name >>

Association

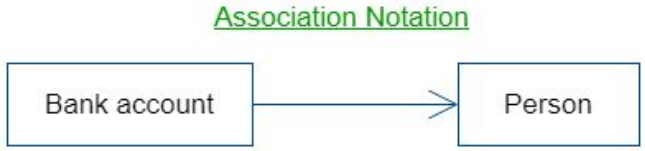


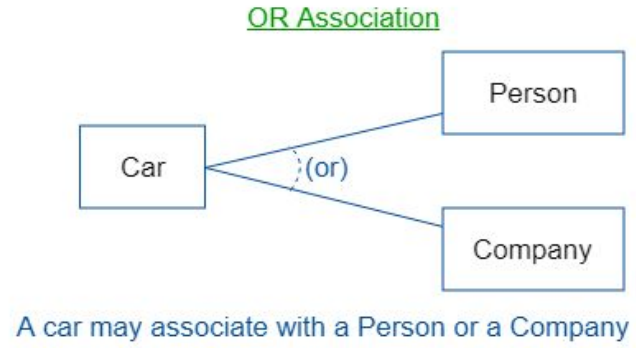


Association is most general, just means instances of one class know about or communicate with instances of another class (or the same class)

Unidirectional represented as line with arrow, bidirectional represented as just plain line

The name of the association (or **role**) is given above/below the association line



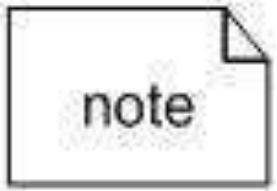


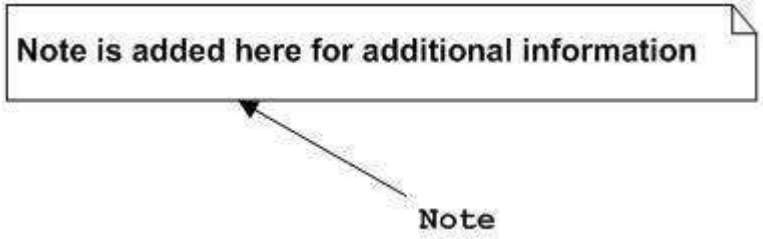
Cardinality

Multiplicity of instances on corresponding end of association line between the classes

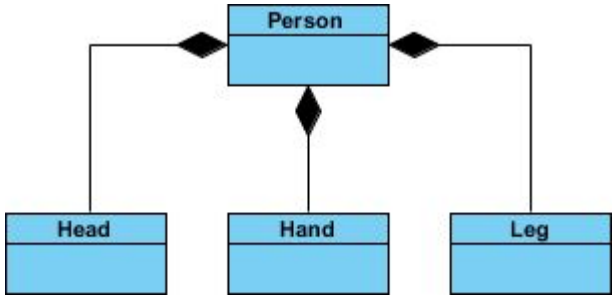
One to one, one to many, many to many

Other constraints may be attached to association lines

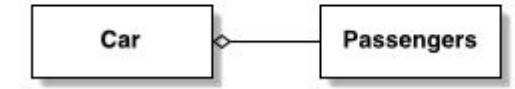




Composition



Composition: every car has an engine.



Aggregation: cars may have passengers, they come and go

Composition (sometimes called strong aggregation) is a special case of association, corresponds to CONSISTS-OF

Strong lifecycle relationship (live and die together) = deleting parent automatically deletes all the children, or all the children must have already been deleted to delete the parent

Children can belong to only one parent and cannot exist on their own, no cycles in the composition graph

Line with closed/filled diamond at container class

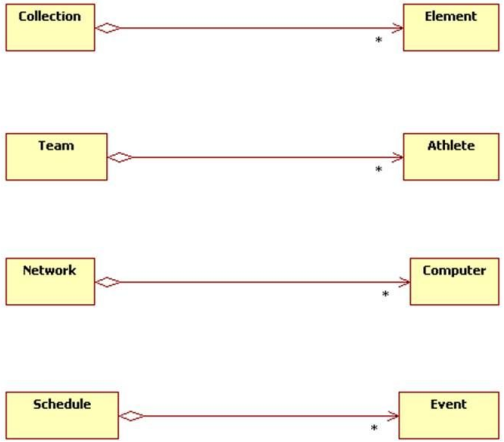
Aggregation

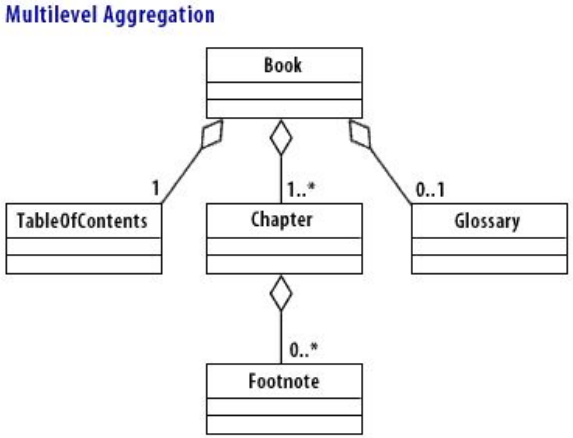
Aggregation (sometimes called open aggregation or weak aggregation) is another special case of association, corresponds to HAS-A or PART-OF (from other end)

No lifecycle relationship (separate lifetimes)

Children can have arbitrary number of parents (and can even be a parent of their own parent)

Line with open diamond at container class

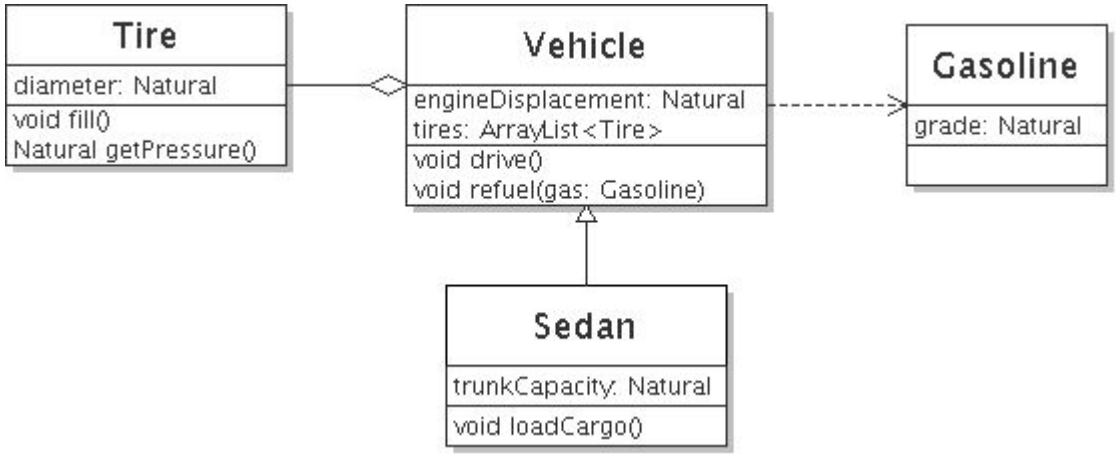




Dependency

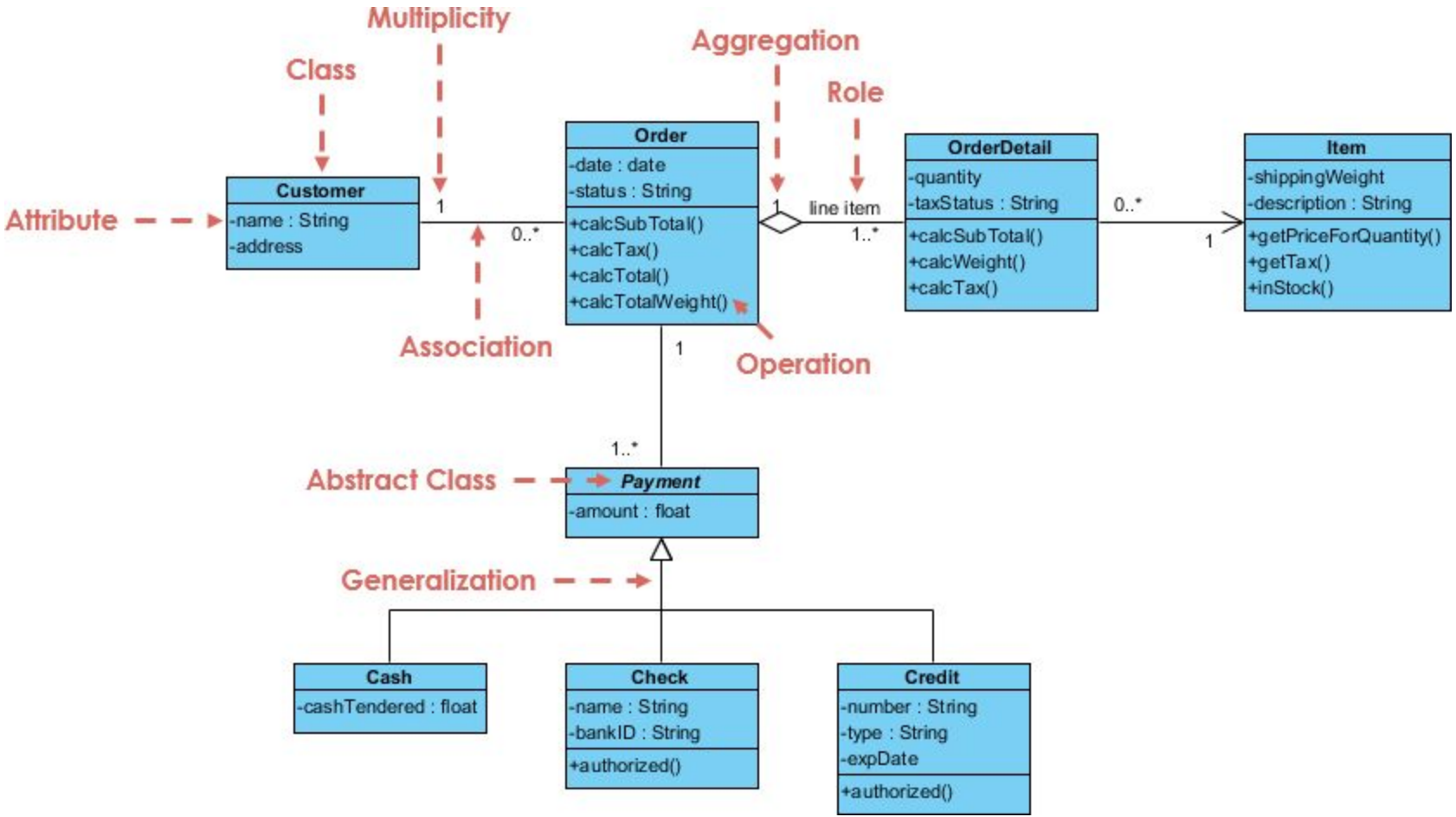
Object of one class uses an object of another class, e.g., as a parameter to an operation, but the object is used transiently and not usually stored as an attribute

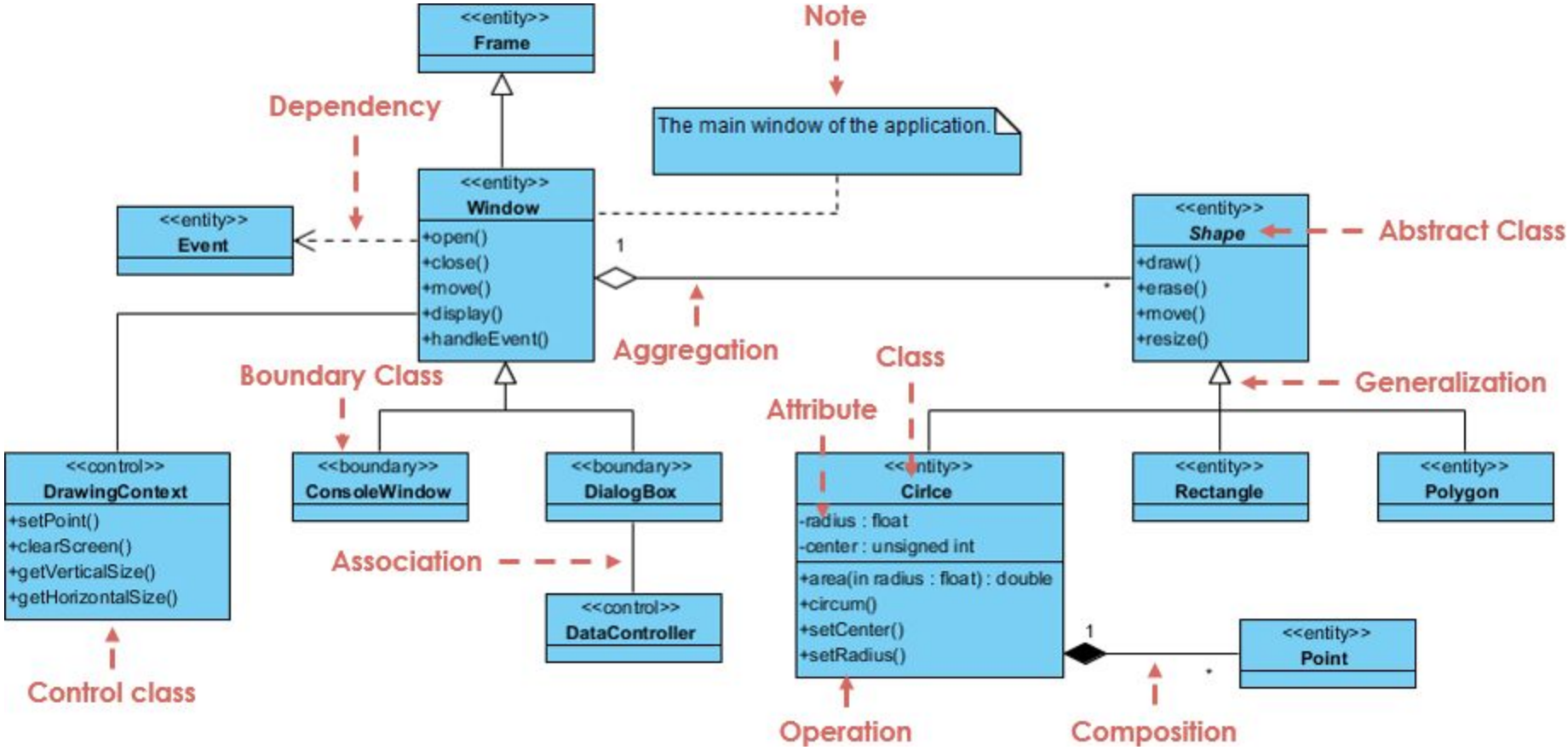
Dependency is a directed supplier-client relationship, where supplier provides something to the client, and thus the client is in some sense incomplete while semantically or structurally dependent on the supplier element(s)



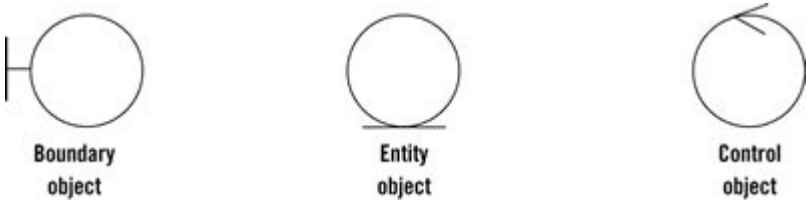
Changes to definition of one may cause or require changes to the other

Dashed arrow


Multiplicity
Class
Aggregation
Role
Attribute
Customer
-name : String
-address
1
0..*
Order
-date : date
-status : String
+calcSubTotal()
+calcTax()
+calcTotal()
+calcTotalWeight()
1
line item
1..*
OrderDetail
-quantity
-taxStatus : String
+calcSubTotal()
+calcWeight()
+calcTax()
0..*
1
Item
-shippingWeight
-description : String
+getPriceForQuantity()
+getTax()
+inStock()
Association
Operation
1
1..*
Abstract Class
Payment
-amount : float
Generalization
Cash
-cashTendered : float
Check
-name : String
-bankID : String
+authorized()
Credit
-number : String
-type : String
-expDate
+authorized()


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

Class “Stereotypes”

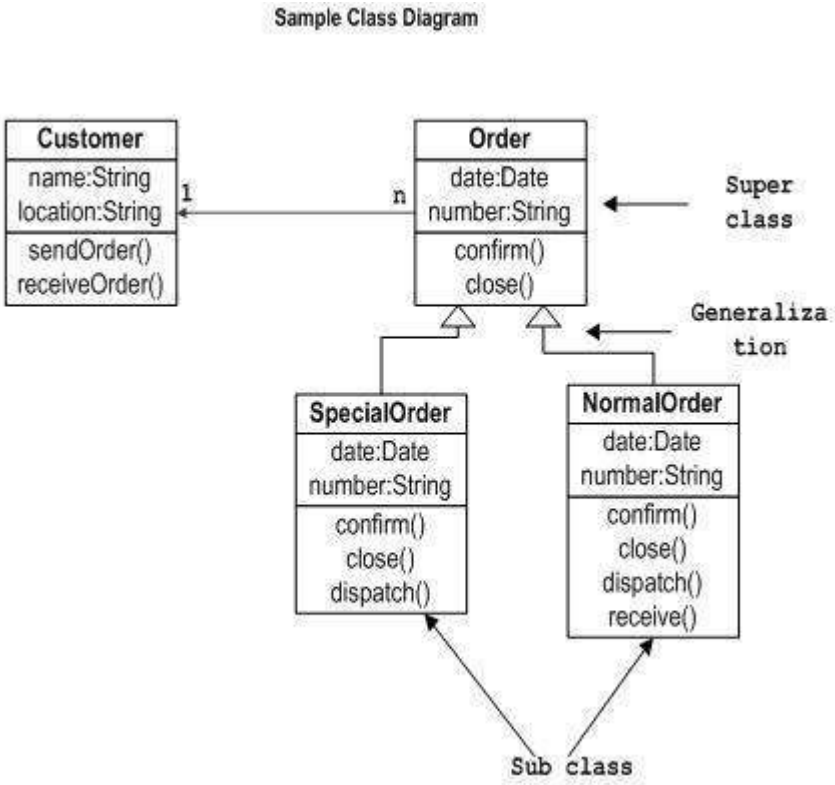


Boundary classes: Model interaction between the system and its actors (users and external systems). Represents some system boundary, e.g. a user interface screen, system interface or device interface object

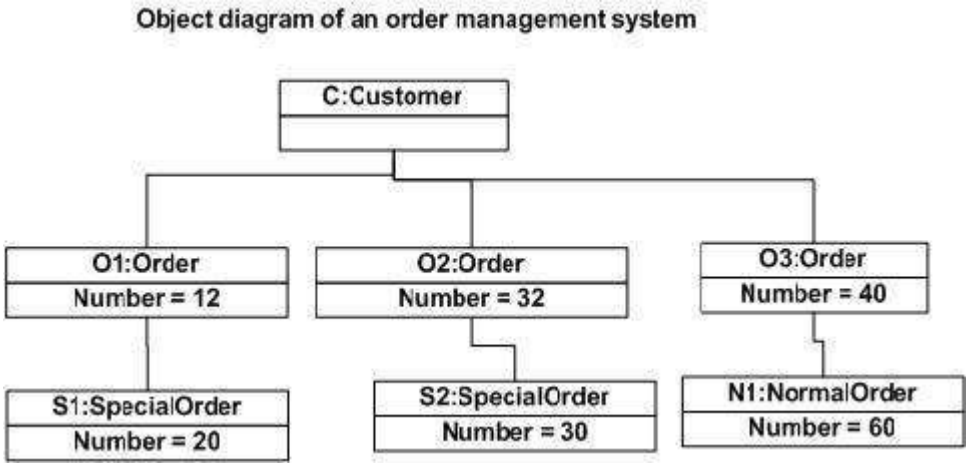
Entity classes: Model information and associated behavior of some phenomenon or concept that is generally long lived (persistent). May reflect a real-world entity or may be needed to perform tasks internal to the system

Control classes: Represent coordination, sequencing, transactions, and control of other objects. Often used to encapsulate control related to a specific use case. Also represent complex derivations and calculations, such as business logic, that cannot be related to any specific, long-lived information stored by the system. Think of a control class as "running" or "executing" the use case and responsible for the flow of events

Class vs. Object Diagrams

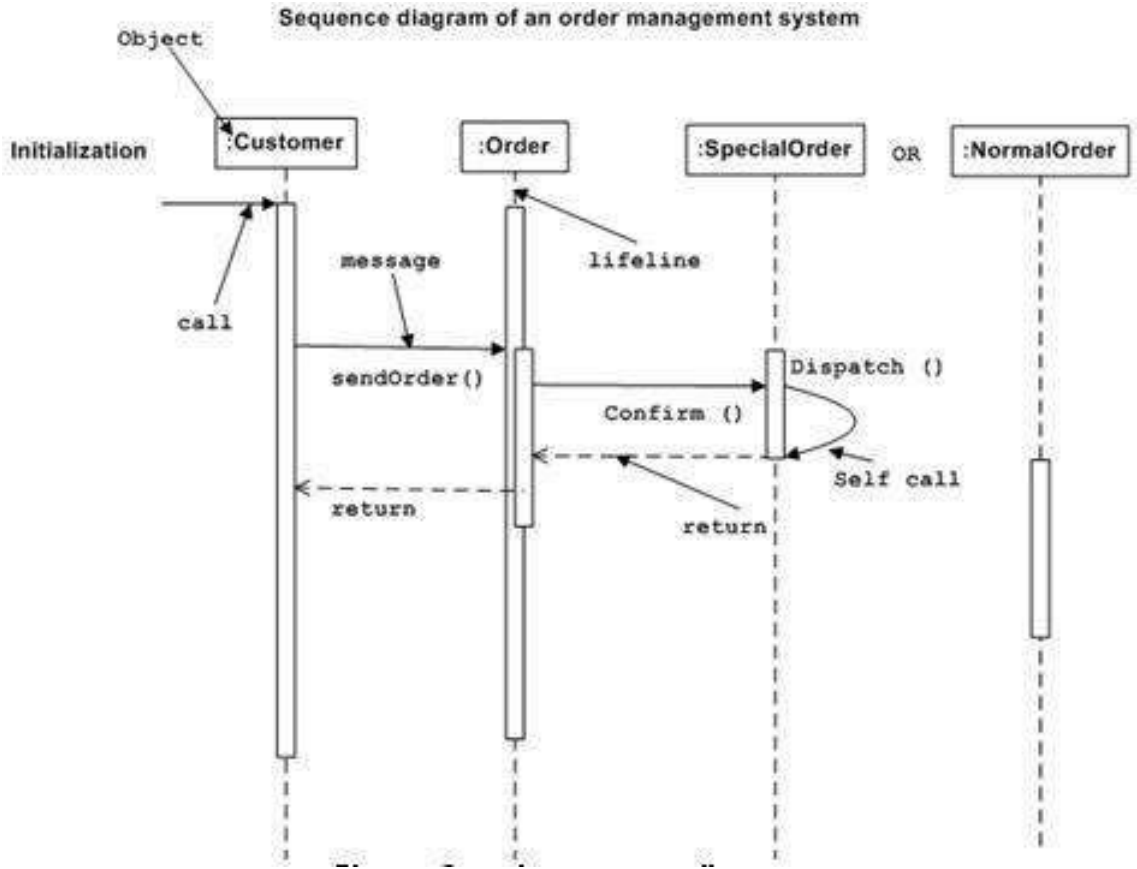


Similar to class diagrams but show snapshot of running system



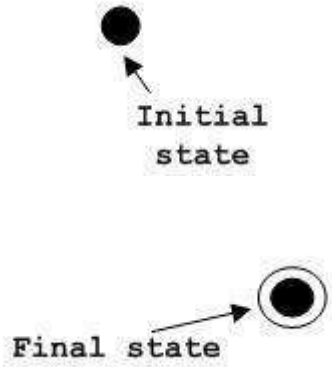
Sequence Diagrams (aka Interaction Diagrams)

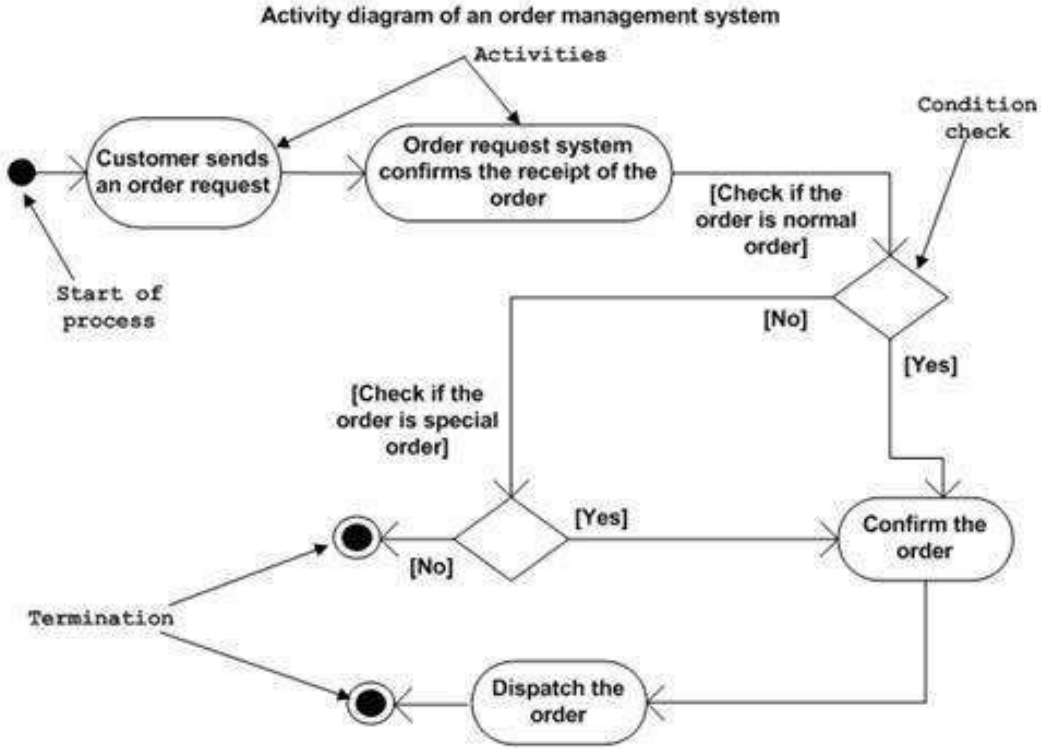
Sequence of messages or calls flowing from one object (an instance of a class) to another



Activity Diagrams

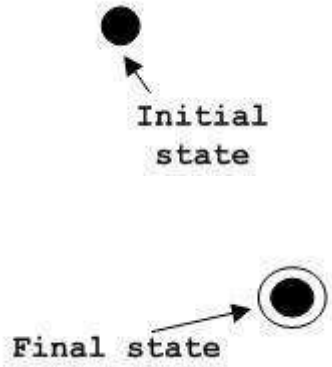
Describes flow of control through system, can be sequential, branched, or concurrent

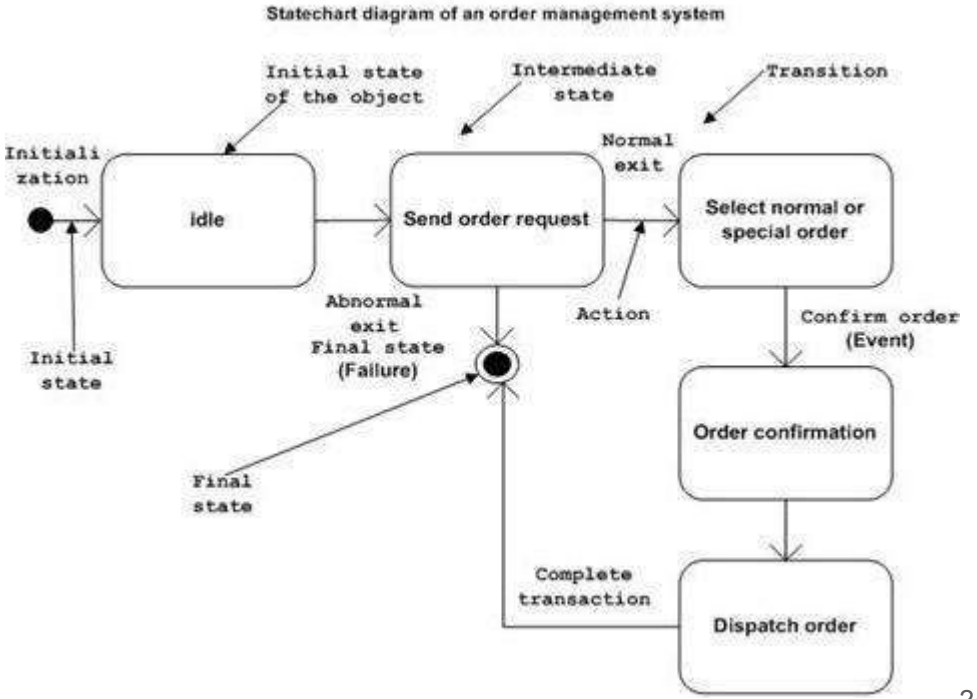




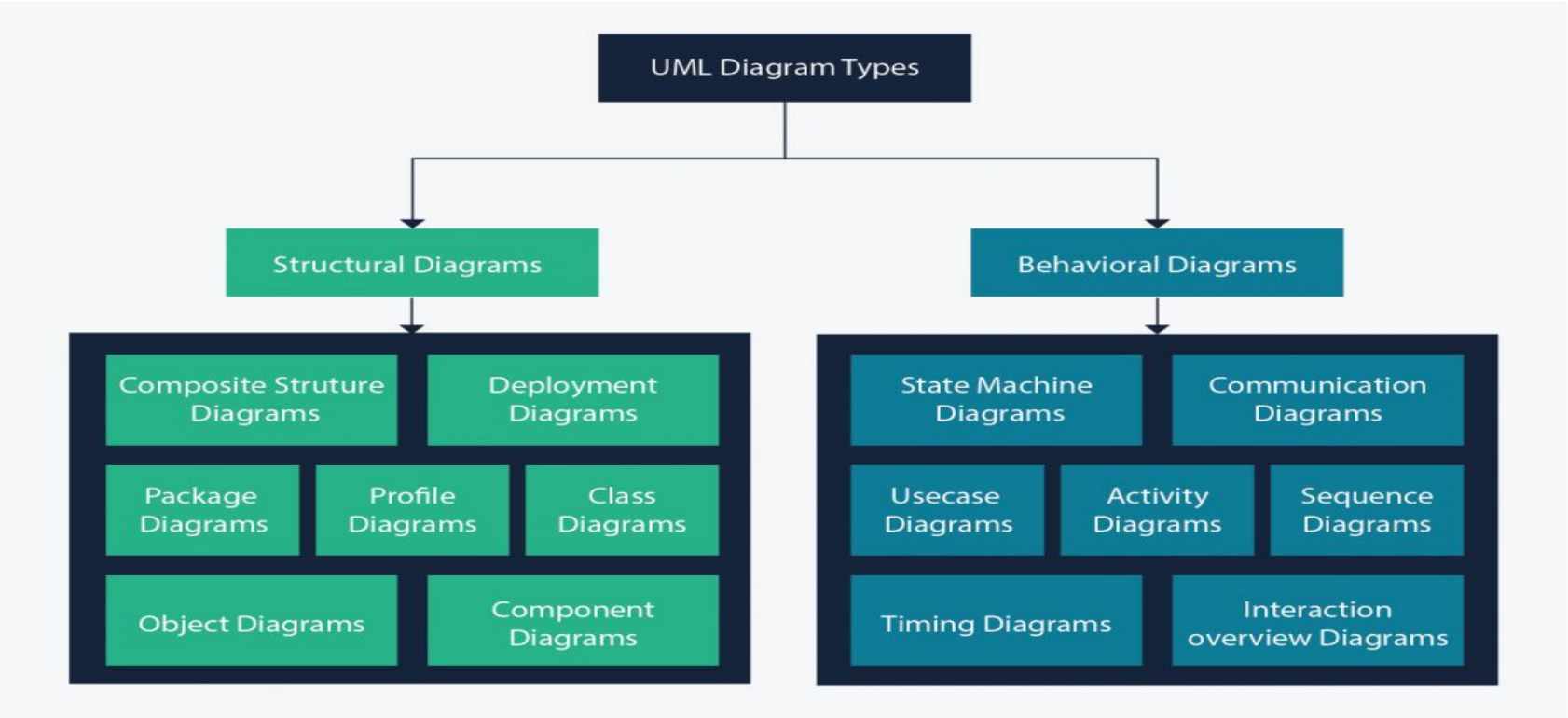
Statecharts

Similar to activity diagrams but focused on object lifecycle and on internal and external events





And More...


UML Diagram Types
Structural Diagrams
Behavioral Diagrams
Composite Struture Diagrams
Deployment Diagrams
Package Diagrams
Profile Diagrams
Class Diagrams
Object Diagrams
Component Diagrams
State Machine Diagrams
Communication Diagrams
Usecase Diagrams
Activity Diagrams
Sequence Diagrams
Timing Diagrams
Interaction overview Diagrams

Team Project Reminder: First Iteration due!

Assignment T3: First Iteration past due

Assignment T4: First Iteration Demo due tomorrow

Assignment T5: Second Iteration due December 11

Assignment T6: Second Iteration Demo due December 15

Assignment T7: Demo Day on December 20