UML DESIGN

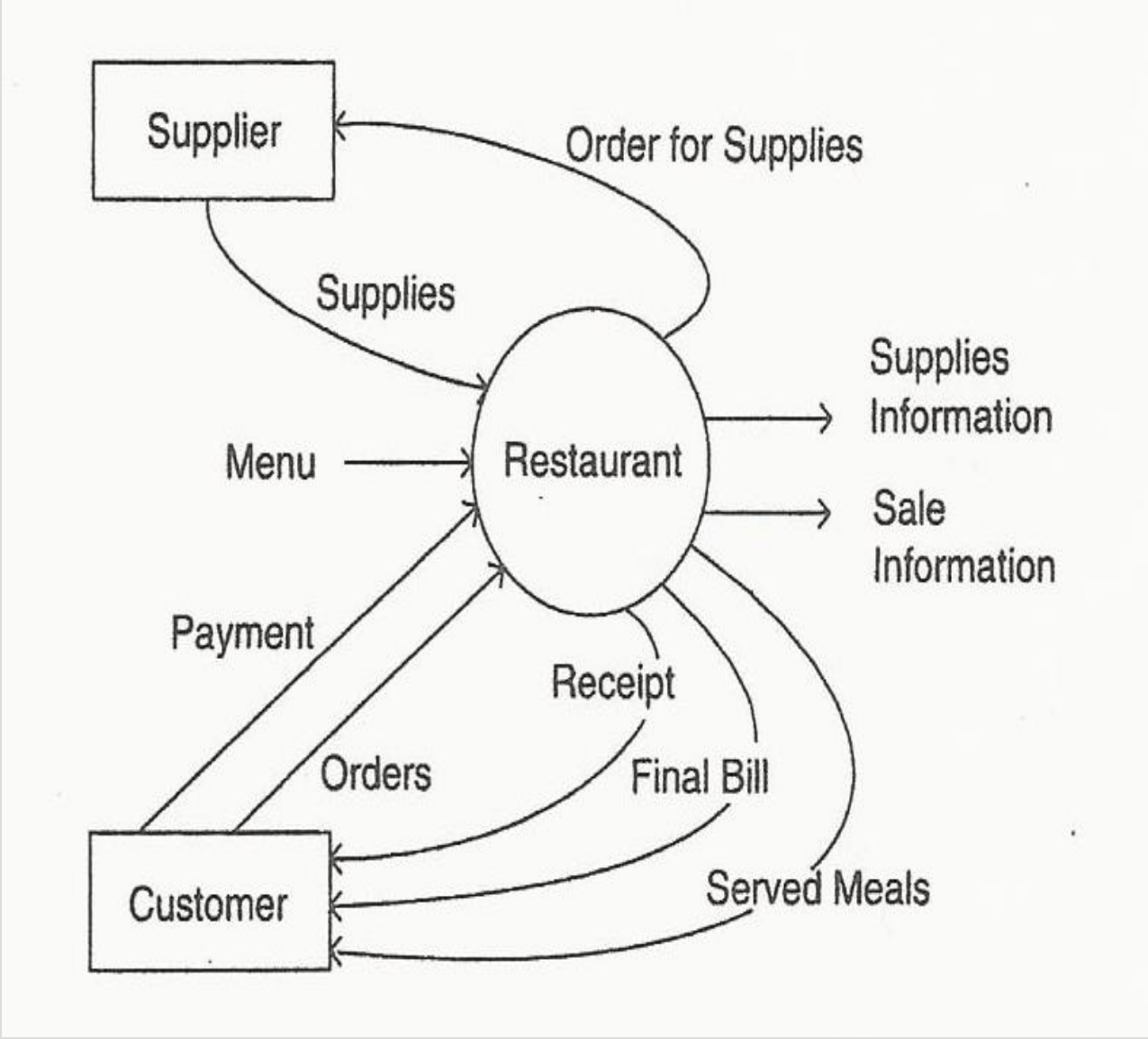
CS 320

Fall 2017

TODAY'S CLASS

- Explain Object-Oriented Design using UML
- Introduce Design Patterns
- Show Examples

ACTIVITY : EXAMPLE RESTAURANT



HLD AND LLD PROCESSES

- **HLD:**
  - Gives the overall System Design in terms of Functional Architecture- the large components and how they connect to each other.
- **Low Level Design Document:**
  - The HLD view(s) of the application is decomposed into modules.
- **LLD gives *design* of actual program code (it is not the actual code).**
- **System code can be developed directly from a good LLD with minimal effort, debugging, and testing**.

LLD DOCUMENT

- **Overview:**
  - the audience is mainly developers.
- **UML design:**
  - The UML class diagram will do this.
- **APIs:**
  - Specify interfaces and data structures.
- **Database design:**
  - ER diagram and data tables.
- **Test Plan (next lecture).**

LLD DETAILS

- **LL Design: specifying the structure of how a software system will be written and function, without actually writing the complete implementation.**
- **A transition from "what" the system must do, to "*how*" the system will do it. For e.g.**
  - What classes will we need to implement a system that meets our requirements?
  - What fields and methods will each class have?
  - How will the classes interact with each other?

DESIGN METHODOLOGY

- **Goal: Partition system into cohesive modules that are loosely coupled.**
- **Method:**
  - Functional abstraction (system is a hierarchy of functions).
  - Data abstraction (data + operations)
    - **Object Oriented** approach is a data abstraction.

UNIFIED MODELING LANGUAGE (UML)

- **Graphical notation useful for OO analysis and design.**
  - Allows representing various aspects of the system.
- **Various notations are used to build different models for the system.**
- **Object-oriented analysis and design (OOAD) methodologies use UML to represent the models they create.**

UML

Three leading object oriented programming researchers joined ranks to combine their languages and came up with an industry standard [mid 1990's].

- Grady Booch (BOOCH)
- Jim Rumbaugh (OML: object modeling technique)
- Ivar Jacobsen (OOSE: object oriented software eng.)

UML – DIAGRAMS

- Use case diagrams
- **Class diagrams**
- **Object diagrams**
- Sequence diagrams
- Collaboration diagrams
- Statechart diagrams
- Activity diagrams
- Component diagrams
- Deployment diagrams
  - …. Standard that has been embraced by the industry.

Uses for UML

- **As a sketch: to communicate aspects of system**
  - forward design: doing UML before coding
  - backward design: doing UML after coding as documentation
  - often done on whiteboard or paper
  - used to get rough selective ideas
- **As a blueprint: a complete design to be implemented**
  - Sometimes done with CASE (Computer-Aided Software Engineering) tools
- **As a programming language: with the right tools, code can be auto-generated and executed from UML**
  - only good if this is faster than coding in a "real" language

UML CLASS DIAGRAMS

What is a UML class diagram?

- A picture of the classes in an OO system, their fields and methods
- Connections between the classes that interact or inherit from each other

What are some things that are not represented in a UML class diagram?

- Details of how the classes interact with each other
- Algorithmic details
- How a particular behavior is implemented

How do we design classes?

- **Class identification from project spec / requirements**
  - nouns are potential classes, objects, fields
  - verbs are potential methods or responsibilities of a class

- **CRC card exercises**
  - write down classes' names on index cards
  - next to each class, list the following:
    - **responsibilities**: problems to be solved; short verb phrases
    - **collaborators**: other classes that are sent messages by this class)

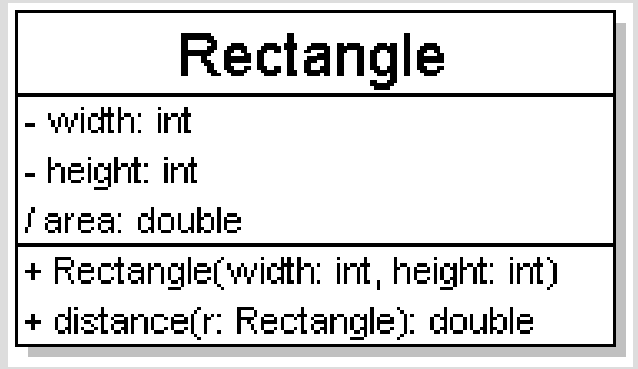
- **UML diagrams**
  - Class diagrams
  - Sequence diagrams (with classes that are identified)

CRC CARD

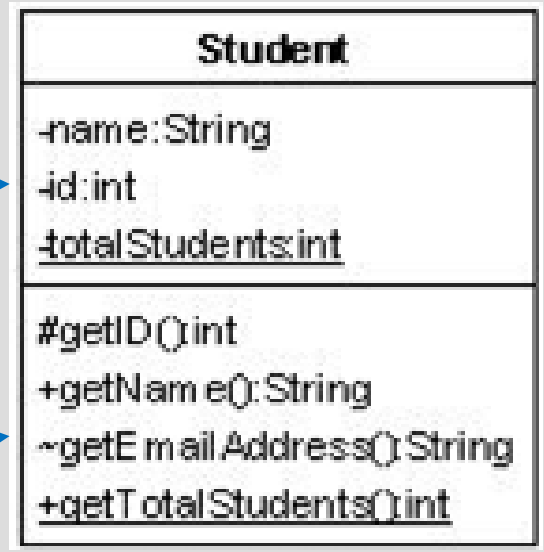
Class name:	
Class type: (e.g., device, property, role, event)	
Class characteristic: (e.g., tangible, atomic, concurrent)	
responsibilities:	collaborations:

EXAMPLE CLASS DIAGRAM

- **Class name in top of box**
  - write <<interface>> on top of interfaces' names
  - use *italics* for an *abstract class* name



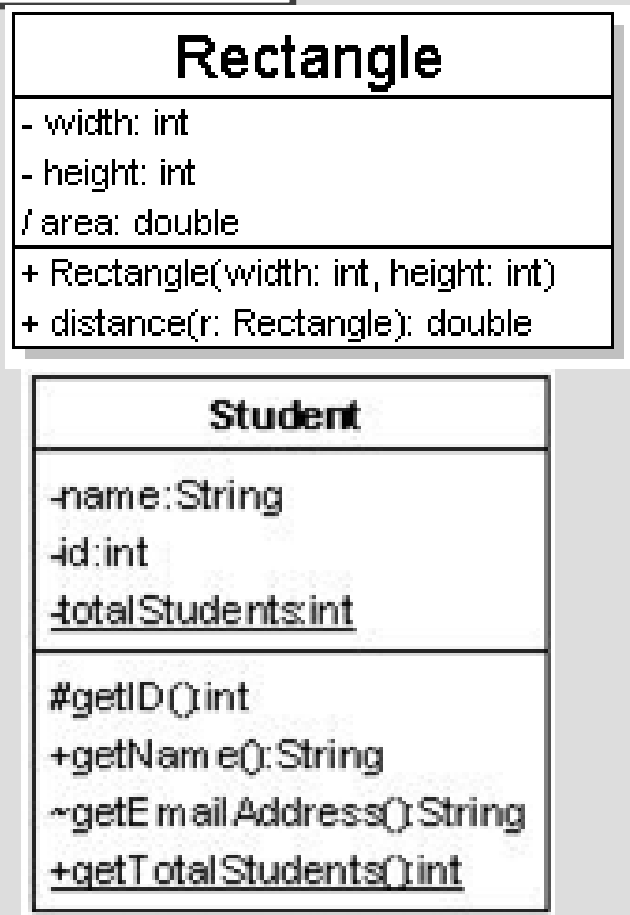
- **Attributes**
  - should include all fields of the object



- **Operations / methods)**
  - should not include inherited methods

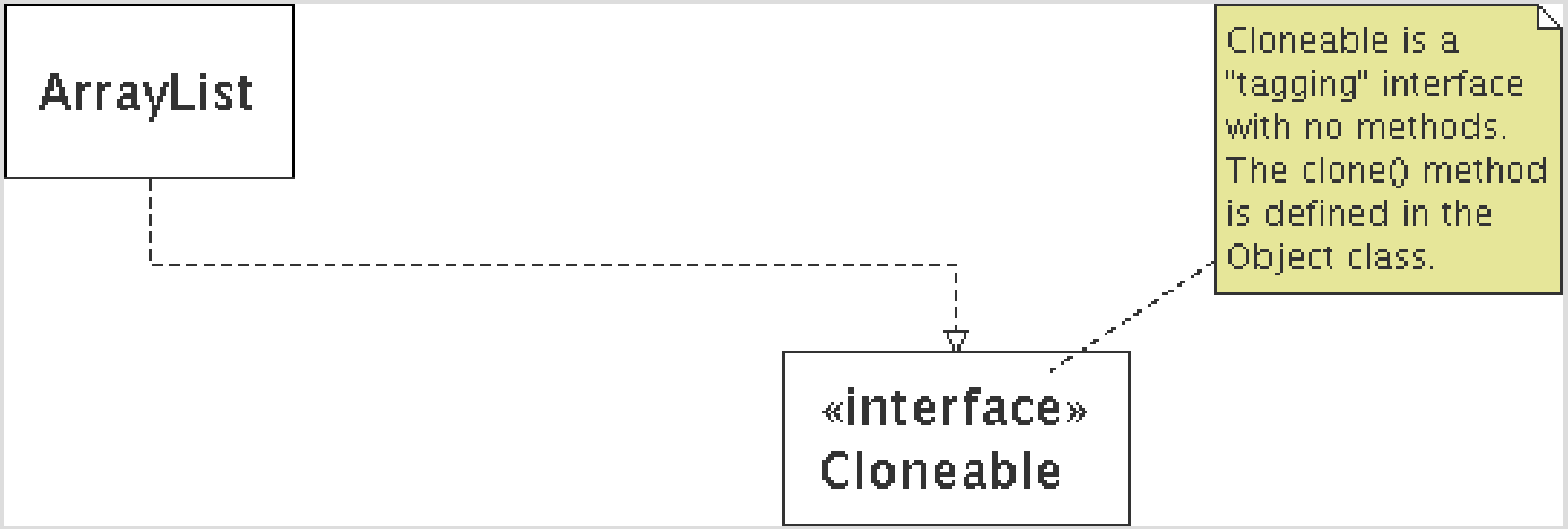
CLASS ATTRIBUTES

- **Attributes (fields, instance variables)**
  - *visibility name : type*
  - Visibility: + public
    # protected
    - private
    ~ package (default)
    / derived
  - Underline static attributes
  - **Derived attribute**: not stored, but can be computed from other attribute values



COMMENTS

- **Represented as a folded note, attached to the appropriate class/method/etc. by a dashed line**

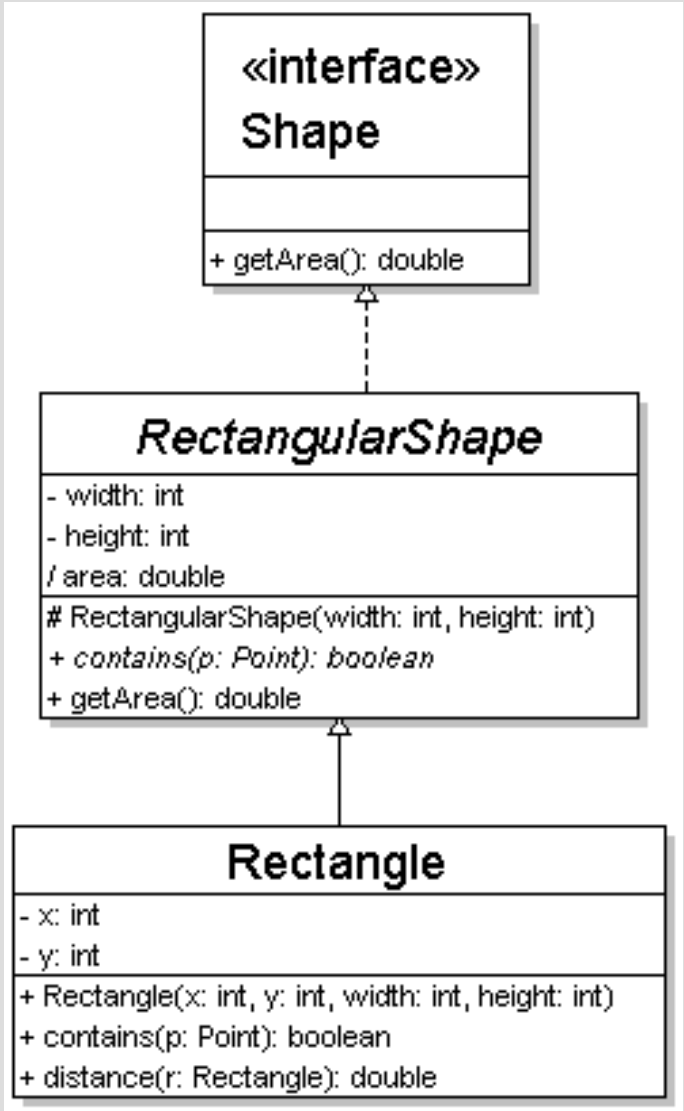


RELATIONSHIPS BETWEEN CLASSES

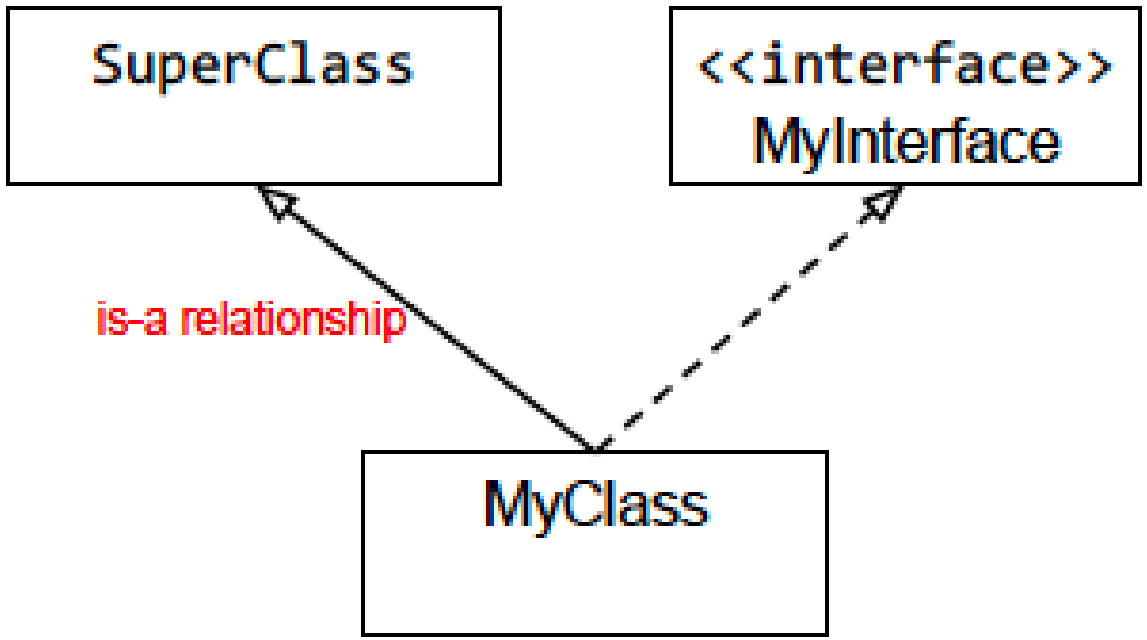
- **Generalization: an inheritance relationship**
  - Inheritance between classes
  - Interface implementation
- **Association: a usage relationship**
  - Dependency
  - Aggregation
  - Composition

GENERALIZATION

- **Inheritance relationships**
  - Hierarchies drawn top-down with arrows pointing upward to parent
  - Line/arrow styles differ, based on whether parent is a(n):
    - **class**:
      solid line, black arrow
    - **abstract class**:
      solid line, white arrow
    - **interface**:
      dashed line, white arrow



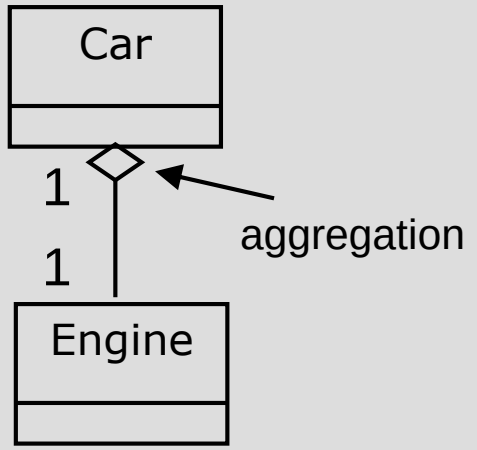
UML CLASS DIAGRAM; INHERITANCE

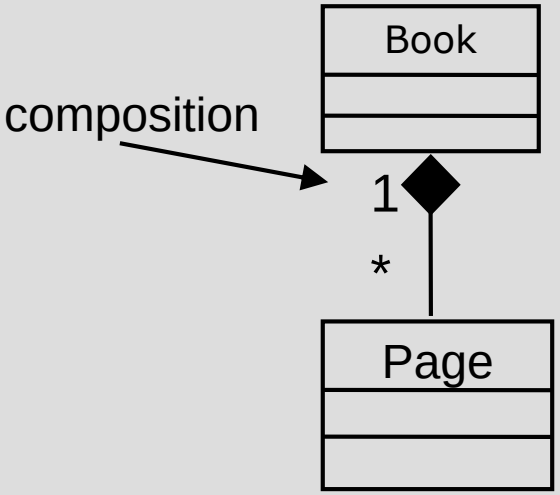


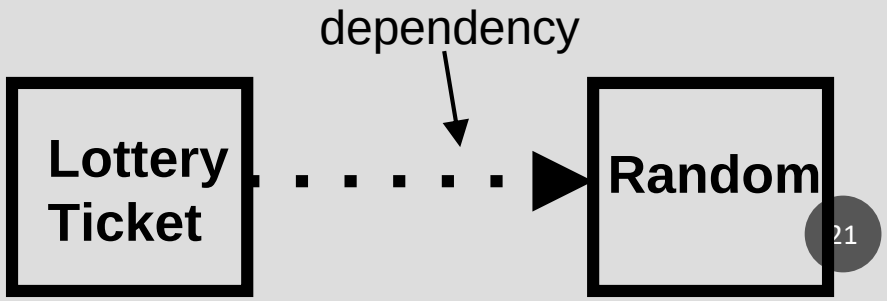

```
public class MyClass extends SuperClass implements MyInterface
```

ASSOCIATION TYPES

- **Aggregation: "is part of"**
  - symbolized by a clear white diamond
- **Composition: "is entirely made of"**
  - stronger version of aggregation
  - the parts live and die with the whole
  - symbolized by a black diamond
- **Dependency: "uses temporarily"**
  - symbolized by dotted line
  - often is an implementation detail, not an intrinsic part of that object's state

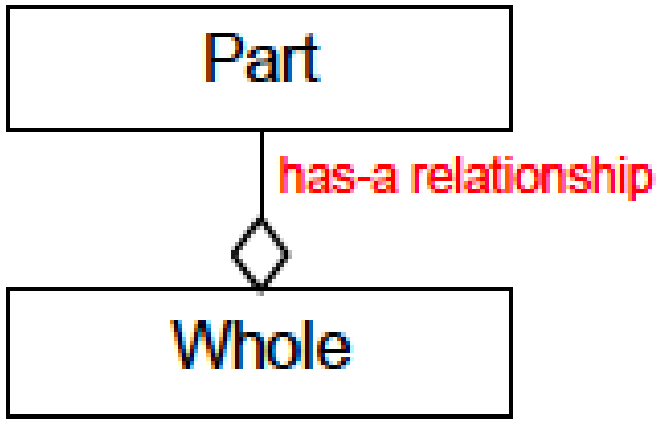






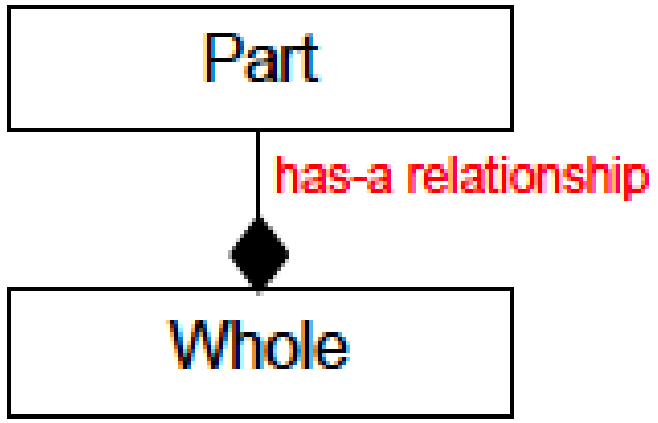
UML: AGGREGATION & COMPOSITION

Aggregation



- Existence of Part does not depend on the existence of Whole.
- Whole does not own Part.
- Part might be shared with other instances of Whole.

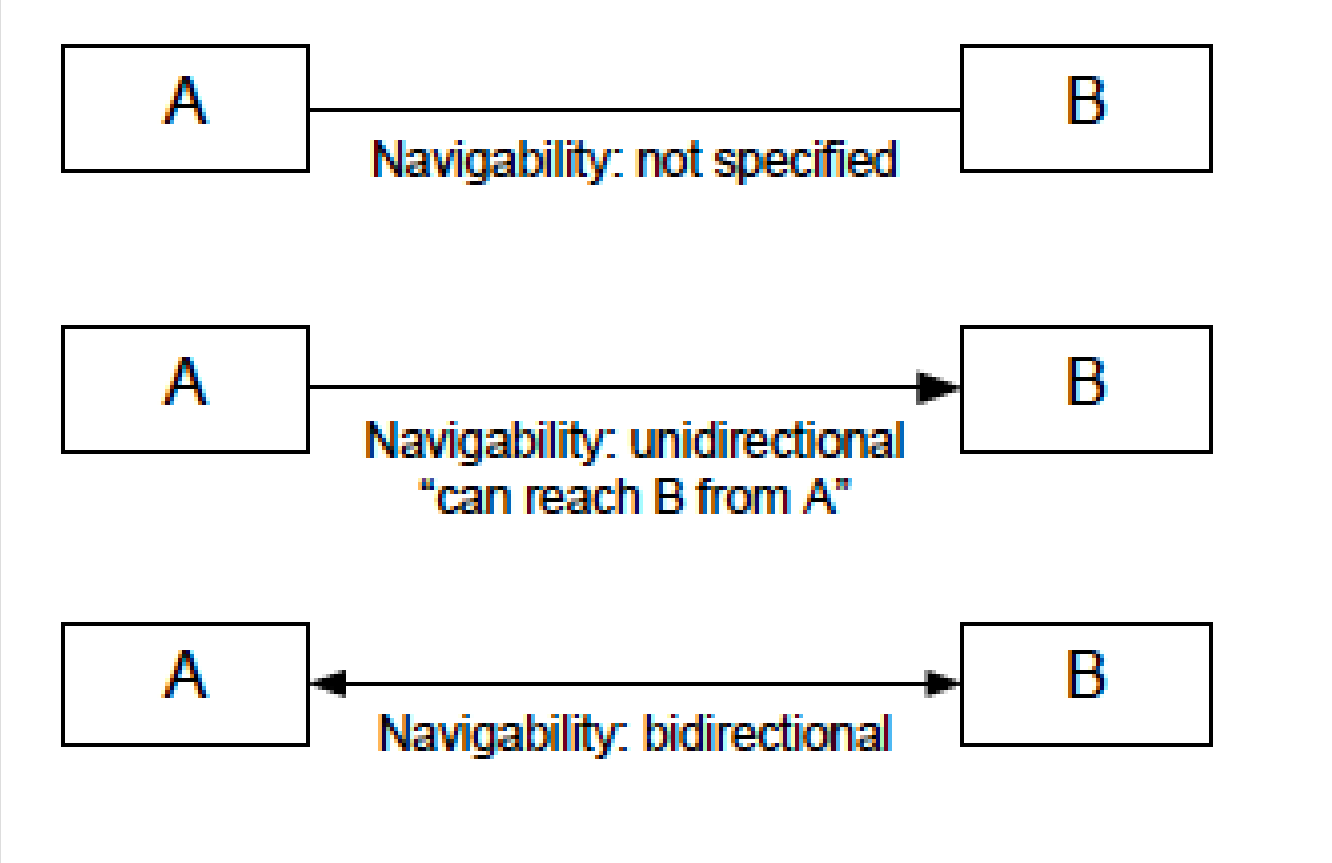
Composition



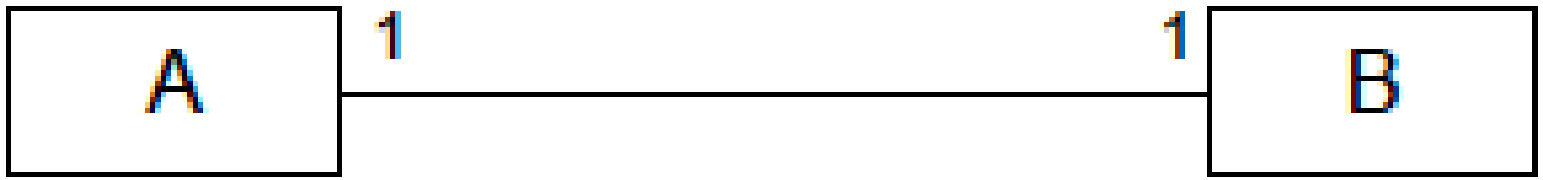
- Part cannot exist without Whole.
- The lifetime of Part is controlled by Whole.
- Whole is the single owner of Part.

Don't confuse an **is-a** relationship with a **has-a** relationship!

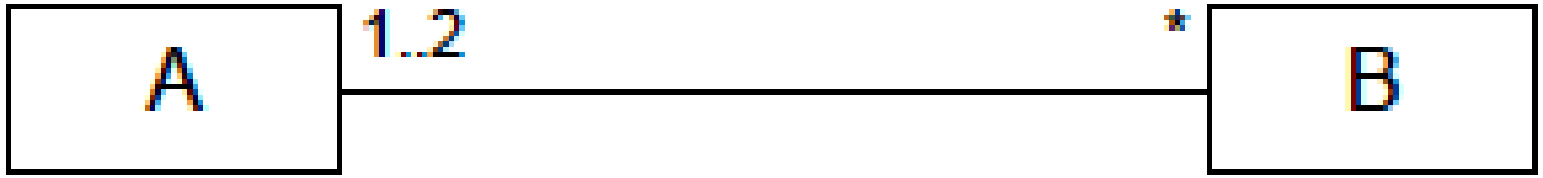
UML: NAVIGABILITY



UML: MULTIPLICITY



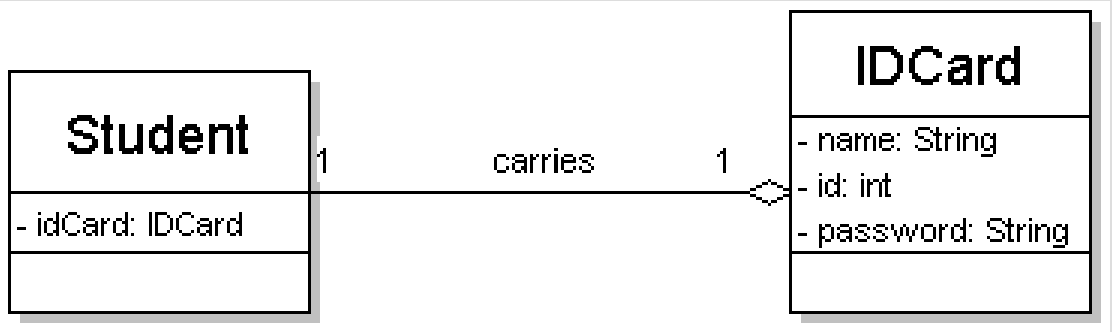
Each A is associated with exactly one B
Each B is associated with exactly one A



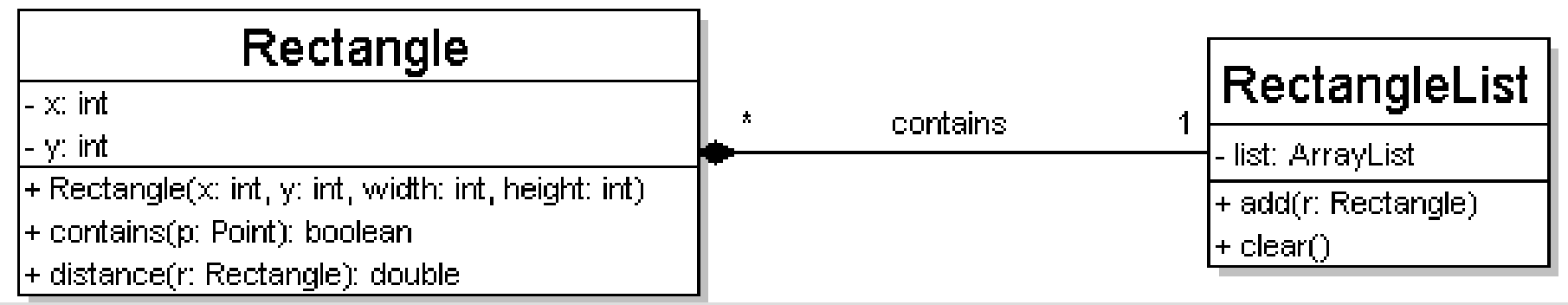
Each A is associated with any number of Bs
Each B is associated with exactly one or two As

MULTIPLICITY OF ASSOCIATIONS

- **One-to-one**
  - each student must carry exactly one ID card

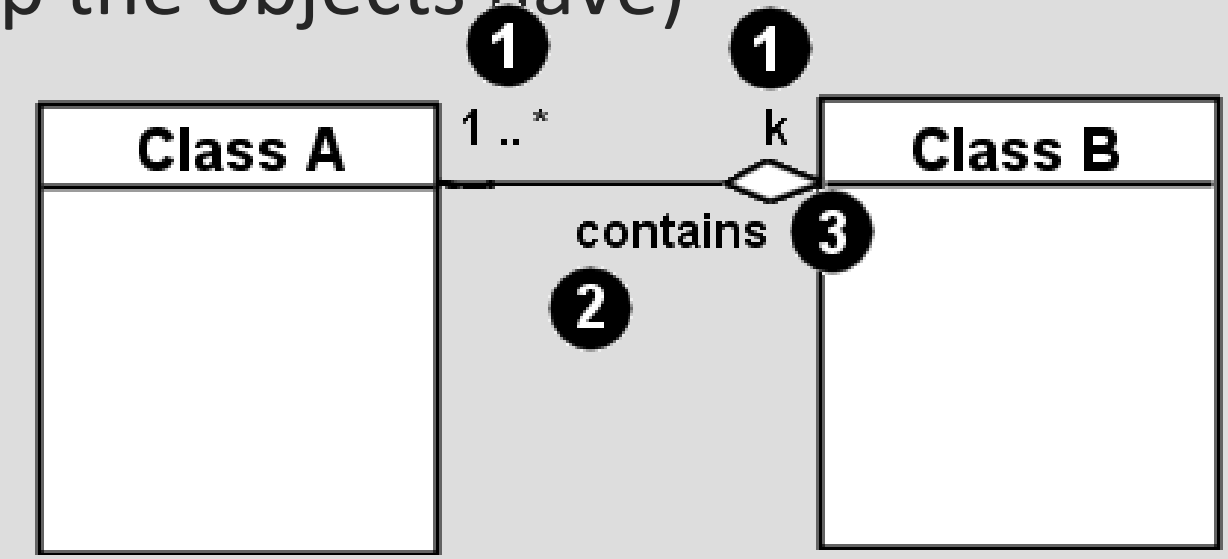


- **One-to-many**
  - one rectangle list can contain many rectangles

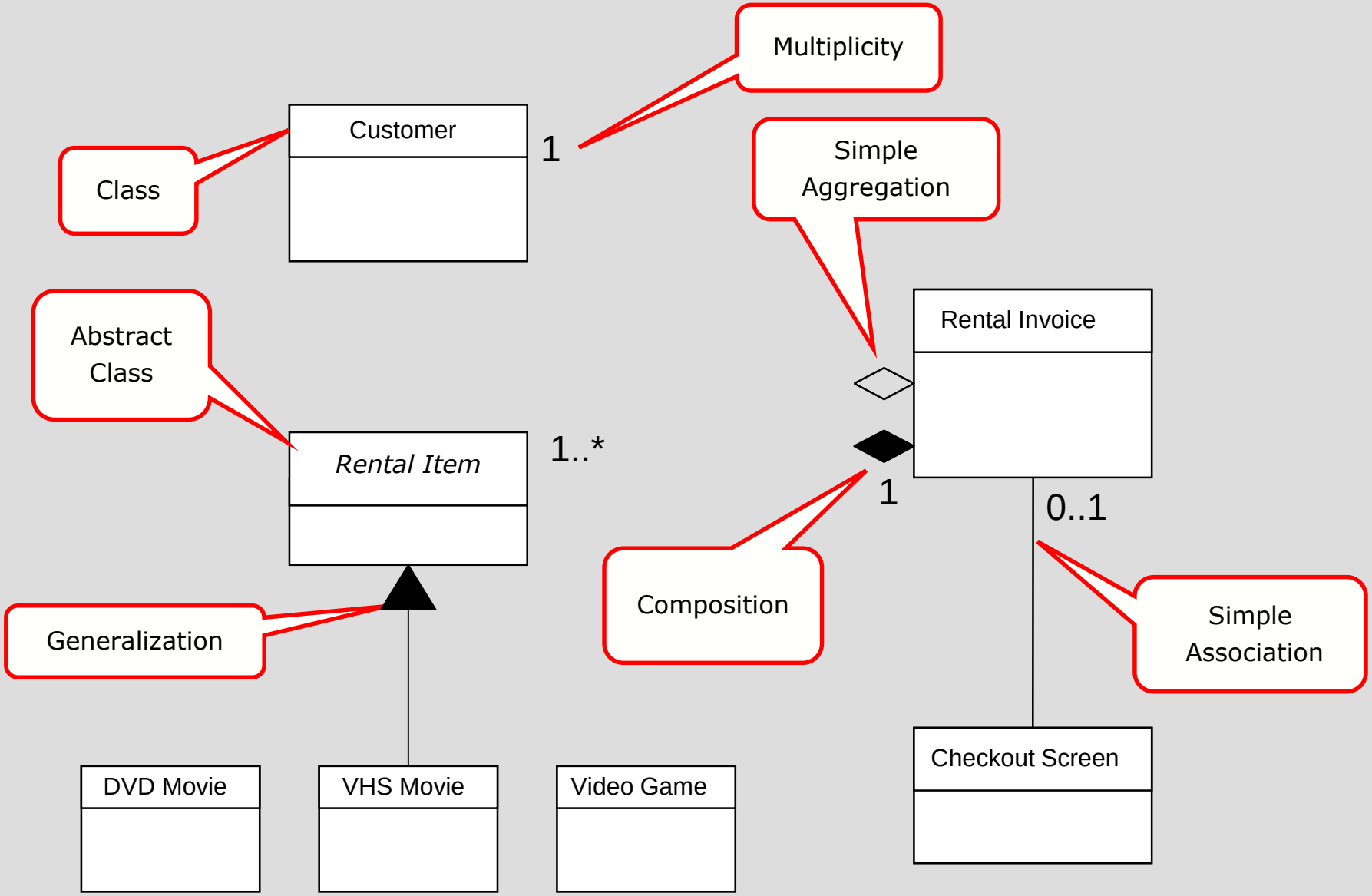


ASSOCIATION

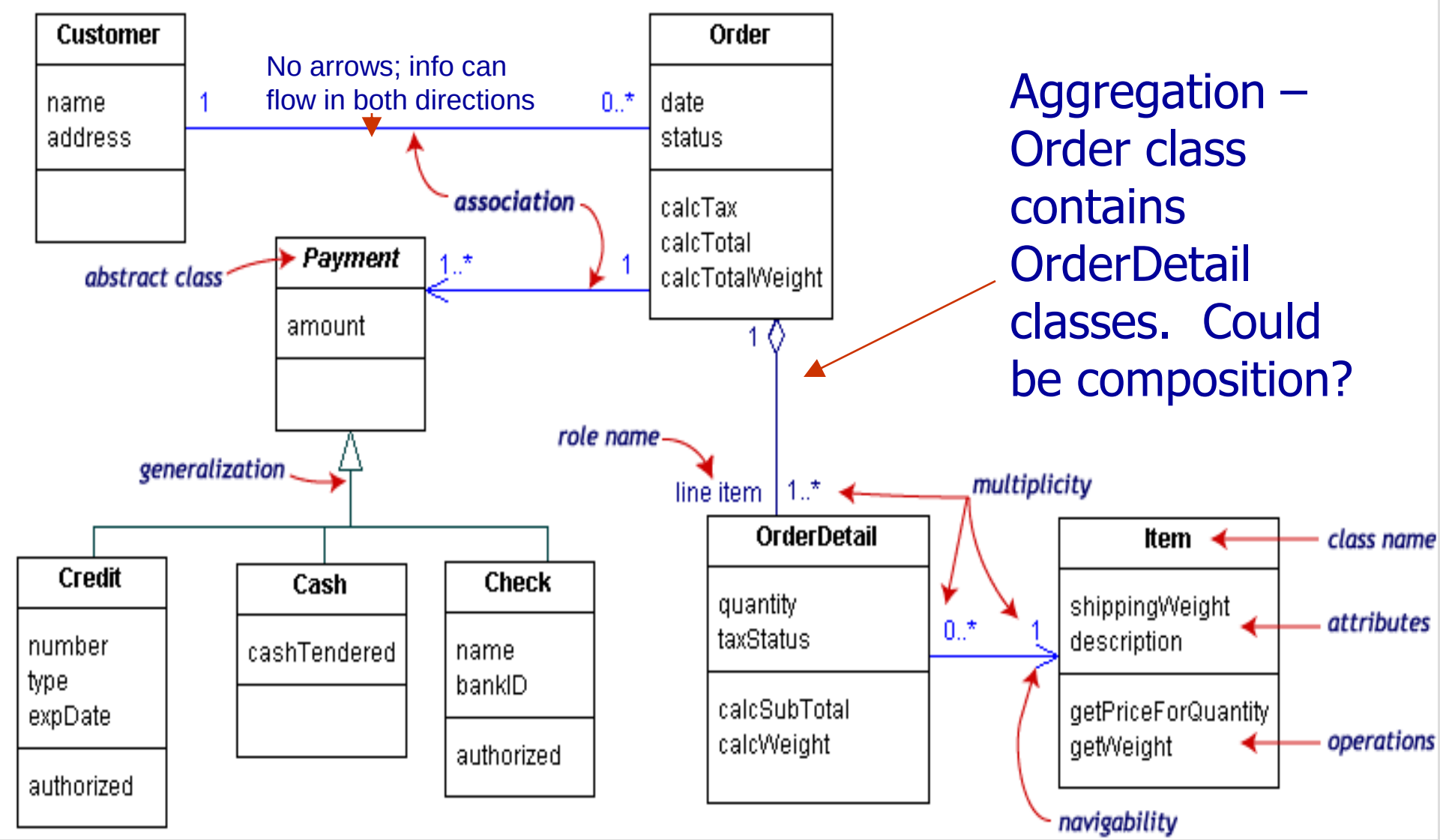
1. **Multiplicity** (how many are used)
   - * ⇒ 1 or more
   - 1 ⇒ 1 exactly
   - 2..4 ⇒ between 2 and 4, inclusive
   - 3..* ⇒ 3 or more
2. **Name** (what relationship the objects have)
3. **Navigability** (direction)



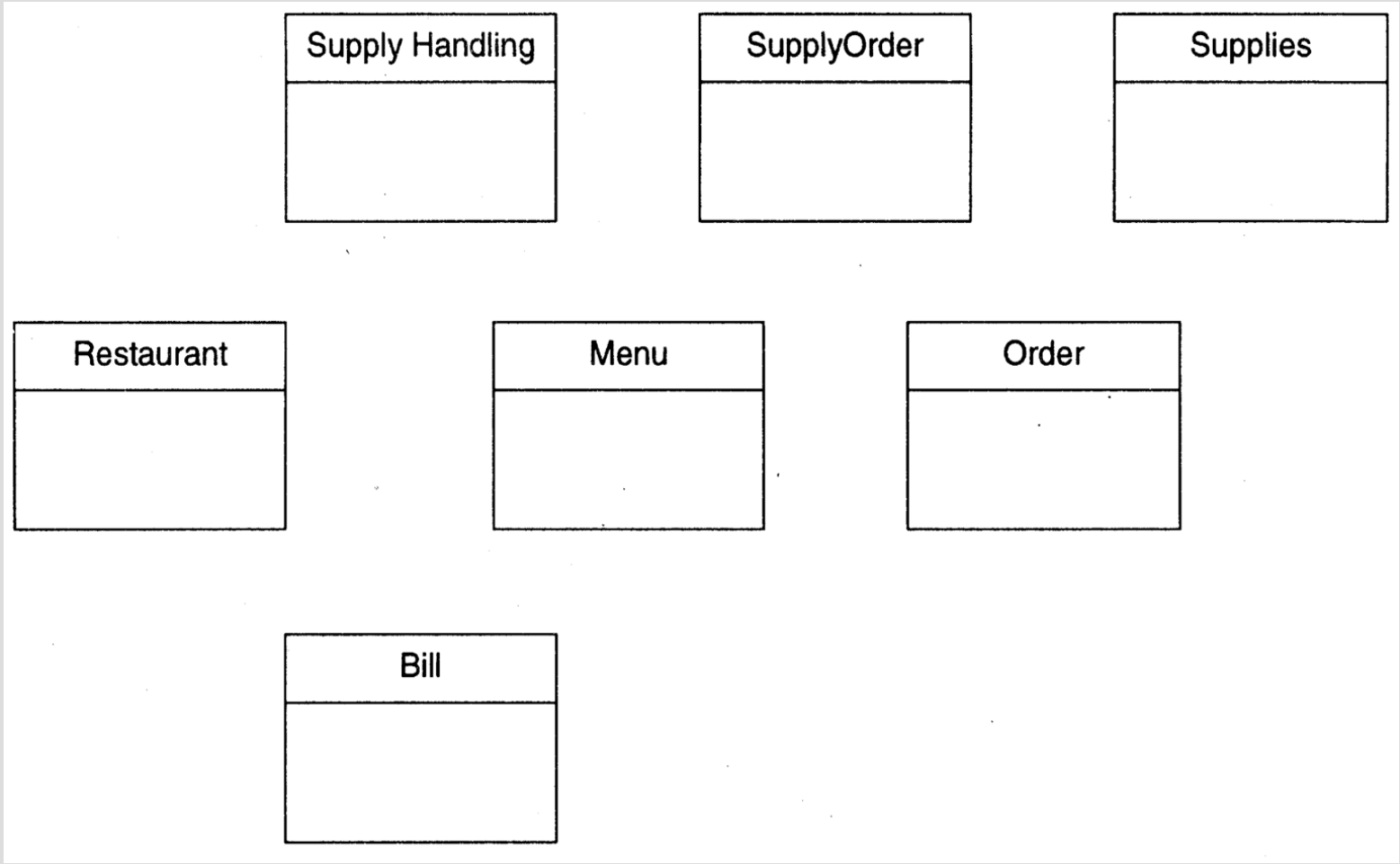
CLASS DIAGRAM EXAMPLE 1



CLASS DIAGRAM EXAMPLE 2



RESTAURANT EXAMPLE: INITIAL CLASSES



DESIGN PATTERNS

- Way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

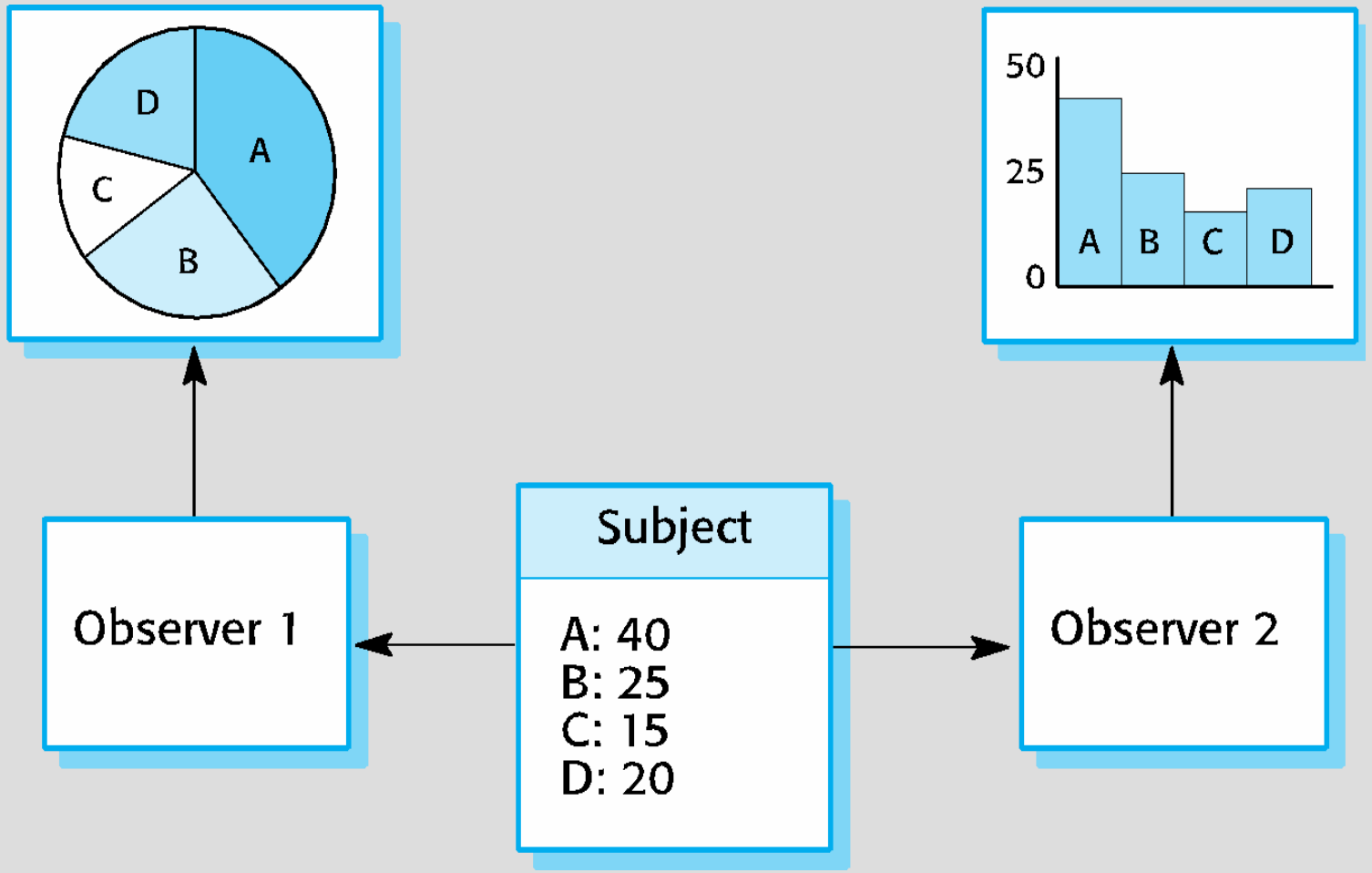
KINDS OF PATTERNS 1

- ***Creational patterns*** **focus on the "creation, composition, and representation of objects, e.g.,**
  - **Abstract factory pattern:** centralize decision of what factory to instantiate.
  - **Factory method pattern:** centralize creation of an object of a specific type choosing one of several implementations.
- ***Structural patterns*** **focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure, e.g.,**
  - **Adapter pattern:** 'adapts' one interface for a class into one that a client expects.
  - **Aggregate pattern:** a version of the Composite pattern with methods for aggregation of children.

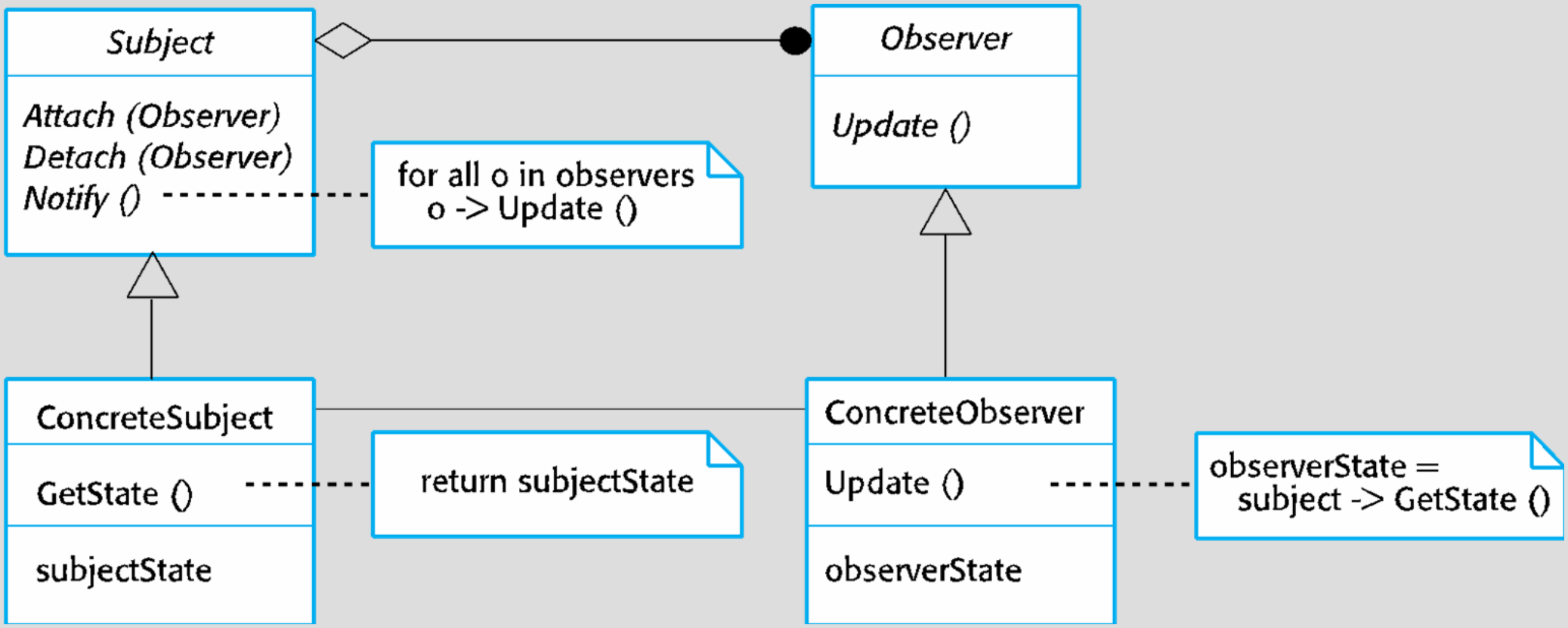
KINDS OF PATTERNS 2

- ***Behavioral patterns*** **address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects, e.g.,**
  - **Chain of responsibility pattern:** Command objects are handled or passed on to other objects by logic-containing processing objects.
  - **Command pattern:** Command objects encapsulate an action and its parameters.

EXAMPLE - MULTIPLE DISPLAYS USING THE OBSERVER PATTERN



UML MODEL OF THE OBSERVER PATTERN



COUPLING

- Coupling is an inter-module concept, captures the strength of interconnection between modules.
- More **tightly coupled** the modules, the more they depend on each other, more difficult to modify one.
- **Low coupling** is desirable for making systems understandable and modifiable.
- In OO, three types of coupling exists – interaction, component, and inheritance.

COHESION

- **Cohesion is an intra-module concept**
- **Focuses on why elements are together**
  - Only elements tightly related should exist together in a module.
  - This gives a module clear abstraction and makes it easier to understand.
- **Higher cohesion leads to lower coupling** – many interacting elements are in the module.
- Goal is to have higher cohesion in modules.
- Three types of cohesion in OO – method, class, and inheritance.

BASIC DESIGN PRINCIPLES

- **Open-Closed Principle (OCP):** *"A module [component] should be open for extension but closed for modification.*
- **Liskov Substitution Principle (LSP):** *"Subclasses should be substitutable for their base classes.*
- **Dependency Inversion Principle (DIP):** *"Depend on abstractions. Do not depend on concretions."*
- **Interface Segregation Principle (ISP):** *"Many client-specific interfaces are better than one general purpose interface.*
- **Release Reuse Equivalency Principle (REP):** *"The granule of reuse is the granule of release."*
- **Common Closure Principle (CCP):** *"Classes that change together belong together."*
- **Common Reuse Principle (CRP):** *"Classes that aren't reused together should not be grouped together."*

DESIGN GUIDELINES

- **Components**
  - Naming conventions should be established for components specified as part of architectural model and then refined and elaborated as part of component-level model
- **Interfaces**
  - Provide important information about communication and collaboration (as well as helping us to achieve the OPC)
- **Dependencies and Inheritance**
  - Have model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

TOOLS FOR CREATING UML DIAGRAMS