SOFTWARE DESIGN AND ARCHITECTURE
SE2002

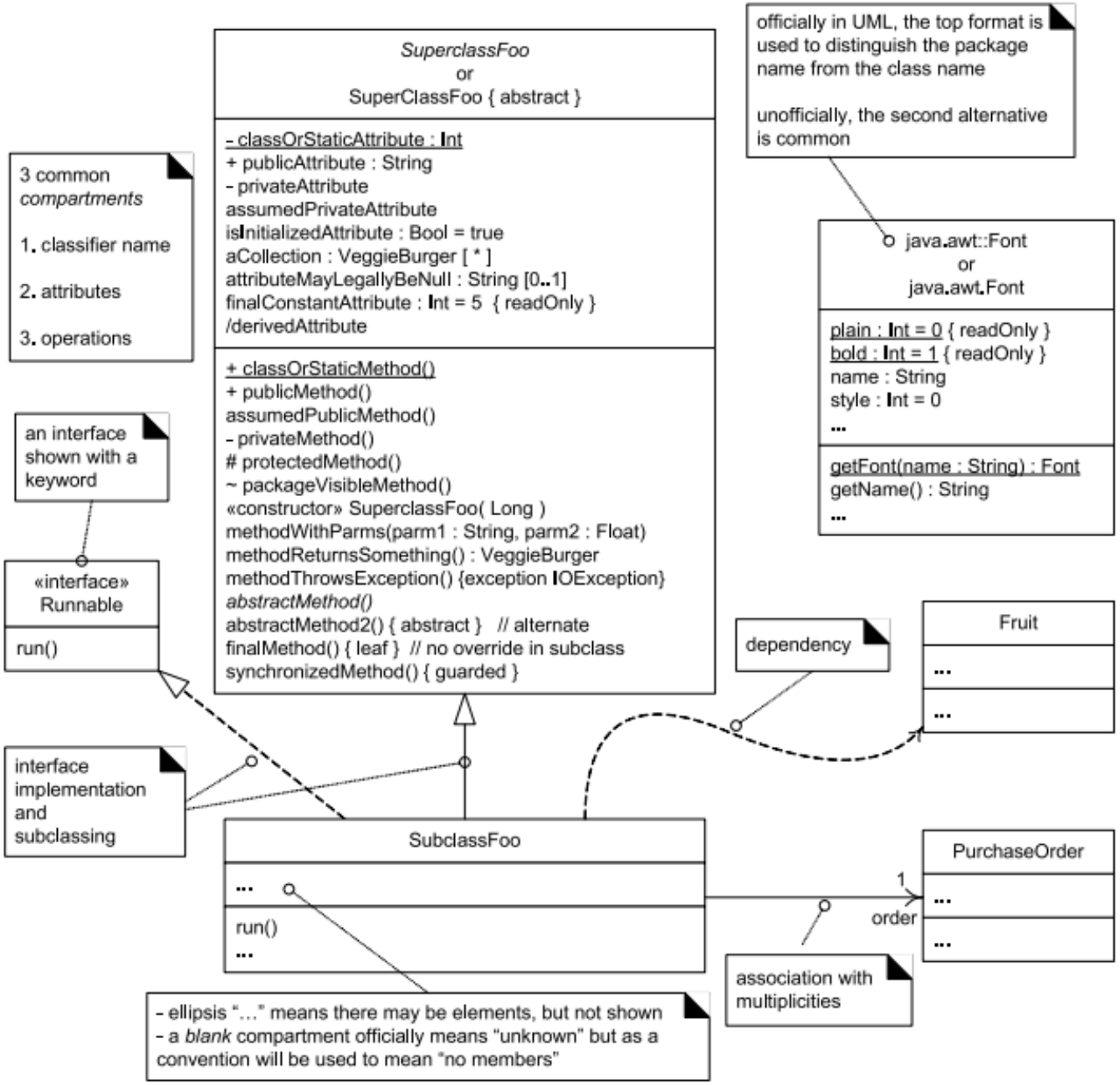
Structural Models
UML Class diagram

Spring 2023
Dr. Muhammad Bilal

Outline

- Class Diagram Fundamentals
  - Domain Model
  - Class Responsibility Collaborator (CRC)
- Class Diagram Notation

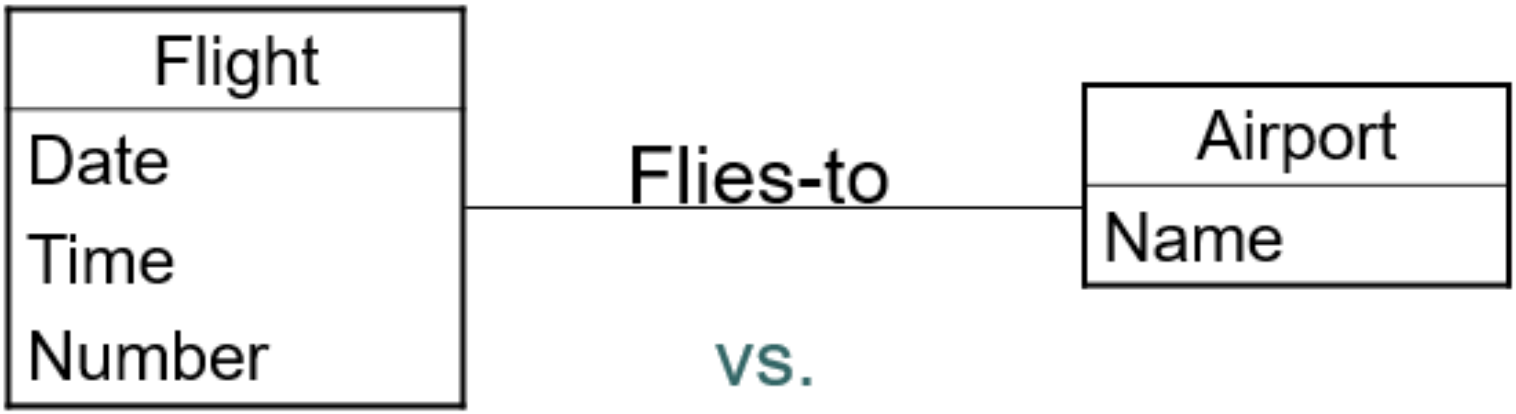
CD Summary

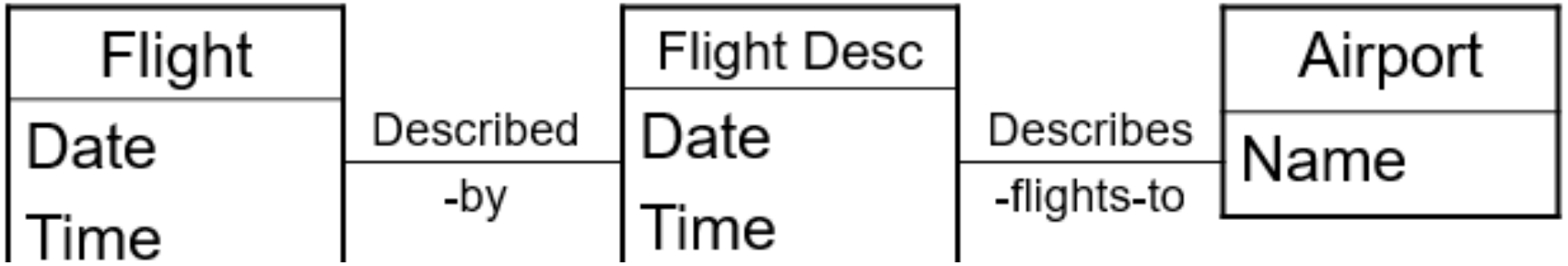

SuperclassFoo
or
SuperClassFoo { abstract }
- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }
3 common compartments
1. classifier name
2. attributes
3. operations
officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common
java.awt::Font
or
java.awt.Font
plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...
getFont(name : String) : Font
getName() : String
...
an interface shown with a keyword
«interface»
Runnable
run()
dependency
Fruit
...
...
interface implementation and subclassing
SubclassFoo
...
run()
...
PurchaseOrder
...
...
1
order
association with multiplicities
- ellipsis "..." means there may be elements, but not shown
- a blank compartment officially means "unknown" but as a convention will be used to mean "no members"

Class Diagrams Fundamentals – Domain Model

- It helps to get the *rough idea* of the system structure and *potential* conceptual classes.
- illustrates meaningful *conceptual classes* in a problem domain.
- is a representation of real-world concepts, not software components.
- is NOT a set of diagrams describing software classes, or software objects and their responsibilities.
- It may show:
  - concepts
  - associations between concepts
  - attributes of concepts

Class Diagrams Fundamentals – Domain Model





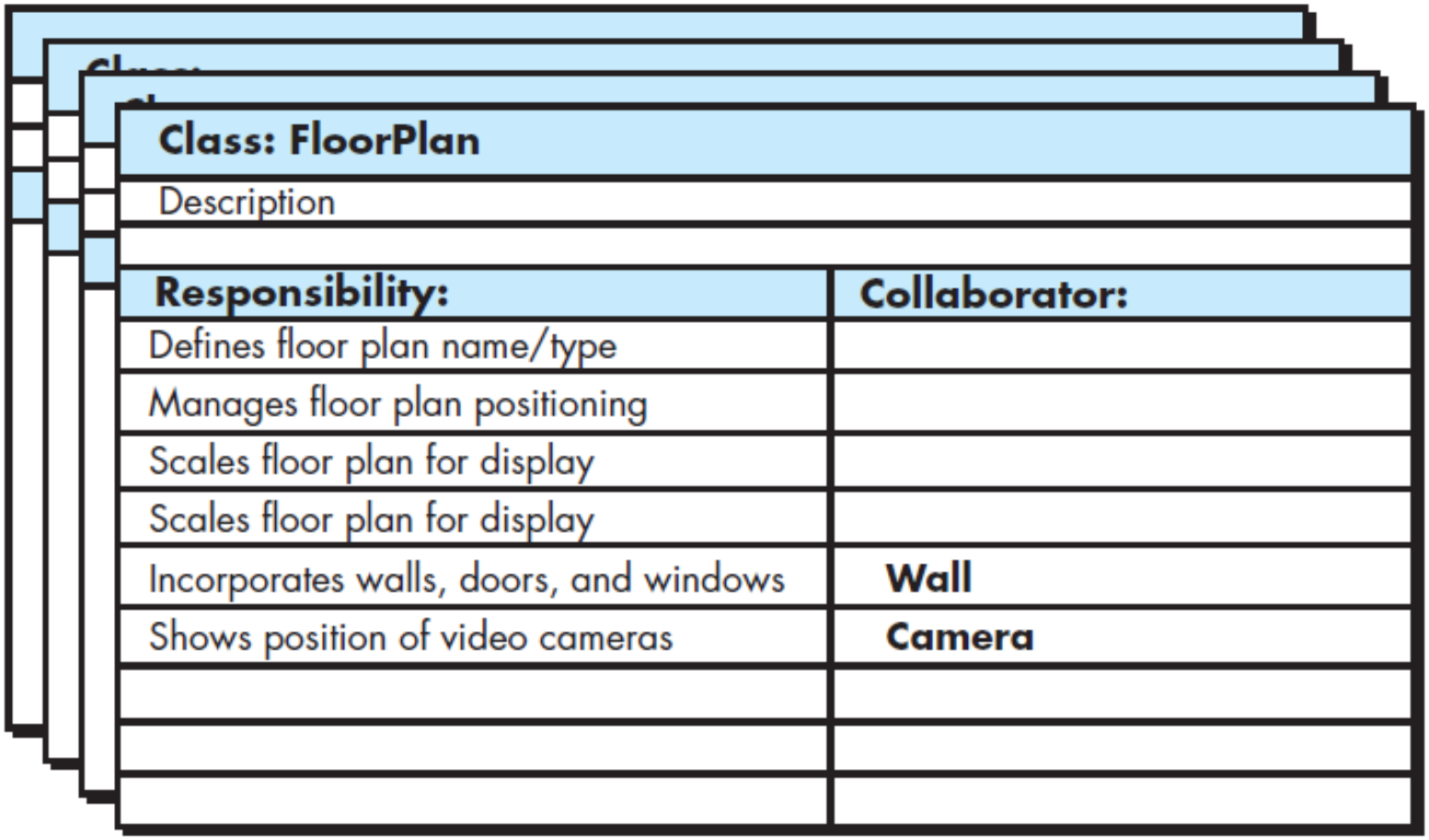
Class Diagrams Fundamentals – CRC

- *CRC modeling* provides a simple means for *identifying* and organizing the classes that are relevant to system or product requirements.

- *Responsibilities* are the *attributes and operations* that are relevant for the class

- *Collaborators* are those classes that *are required* to provide a class with the information needed to *complete a responsibility*.

Class Diagrams Fundamentals – CRC

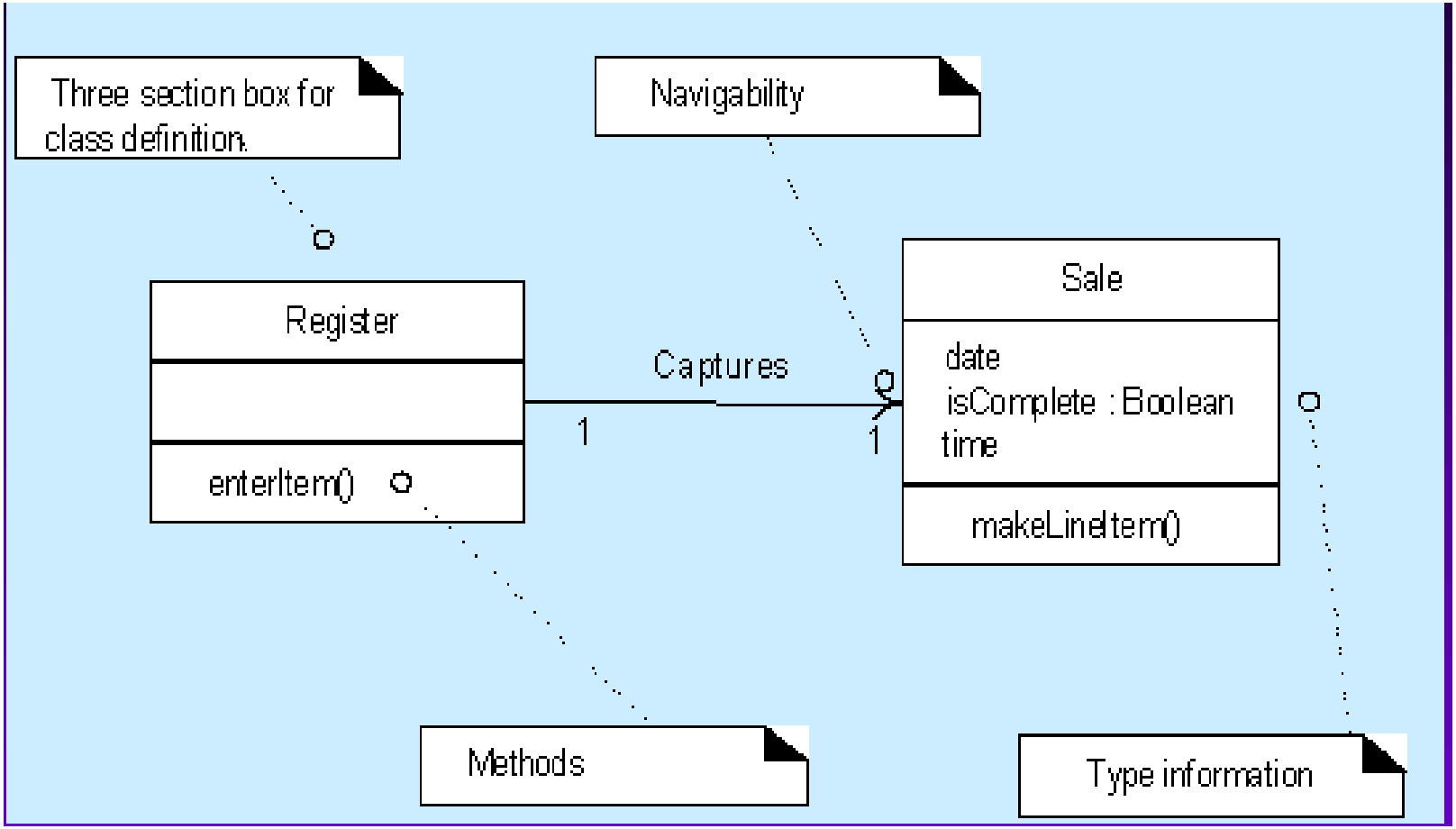


Class: FloorPlan	
Description	
Responsibility:	**Collaborator:**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**

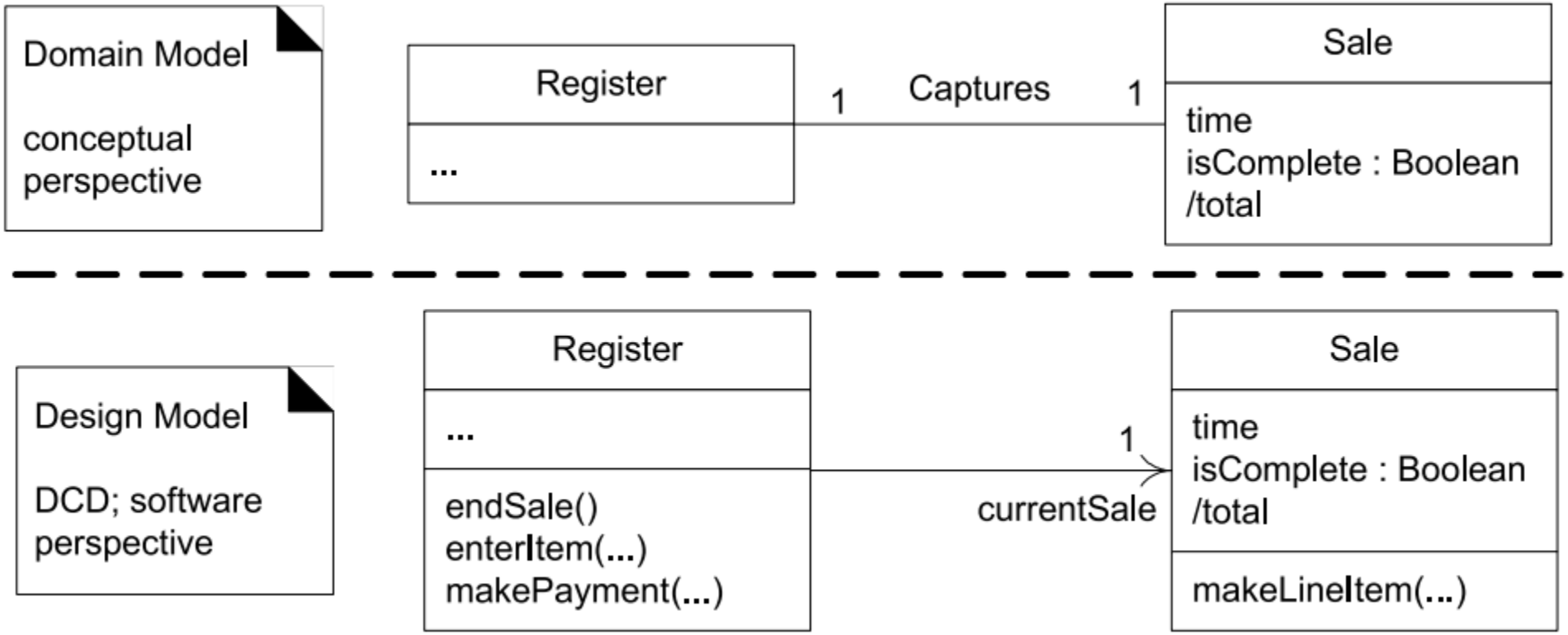
Information in a CD

- A CD illustrates the specifications for software classes and interfaces
- Typical information included:
  - Classes, associations, and attributes
  - Interfaces, with their operations and constants
  - Methods
  - Attribute type information
  - Navigability
  - Dependencies

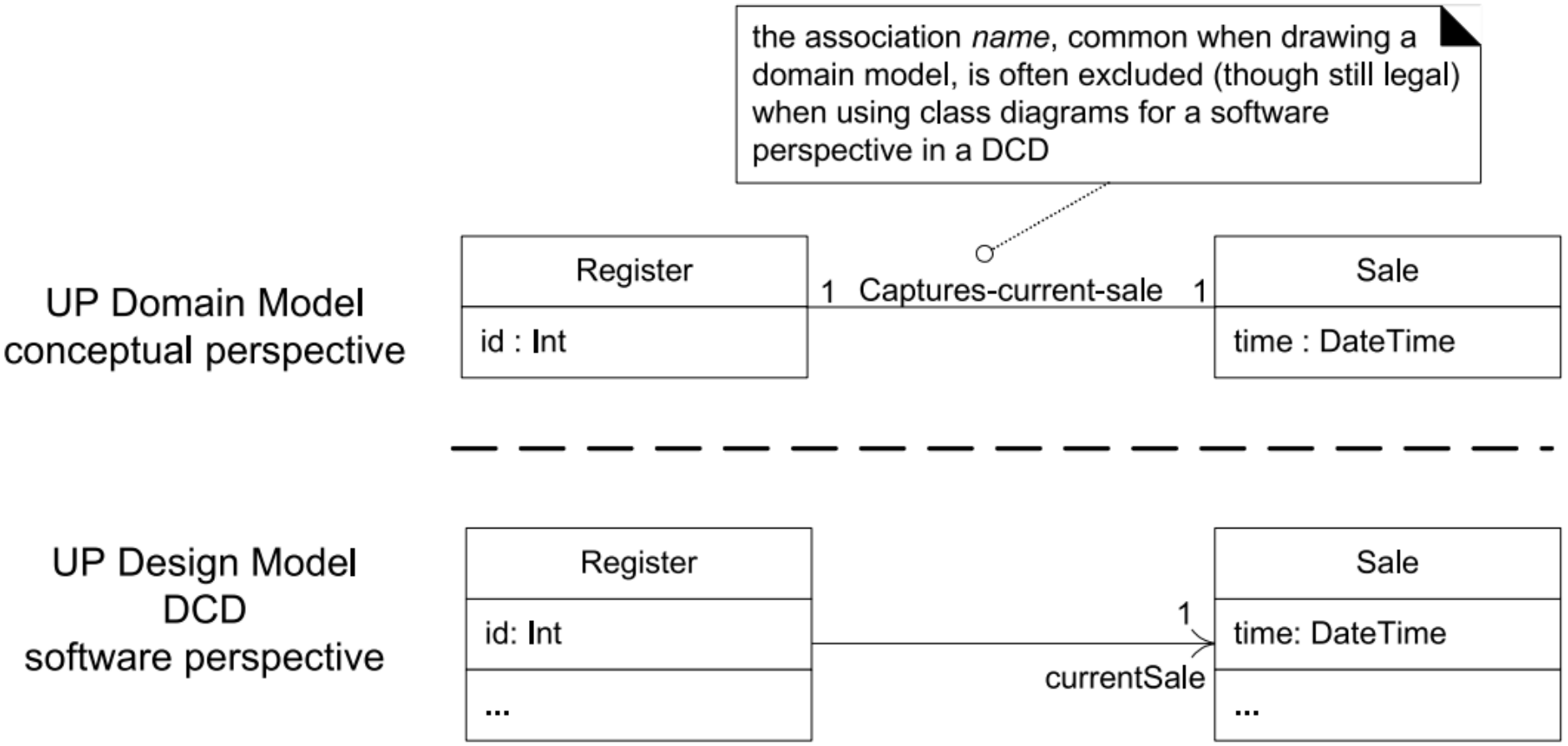
Example of a CD



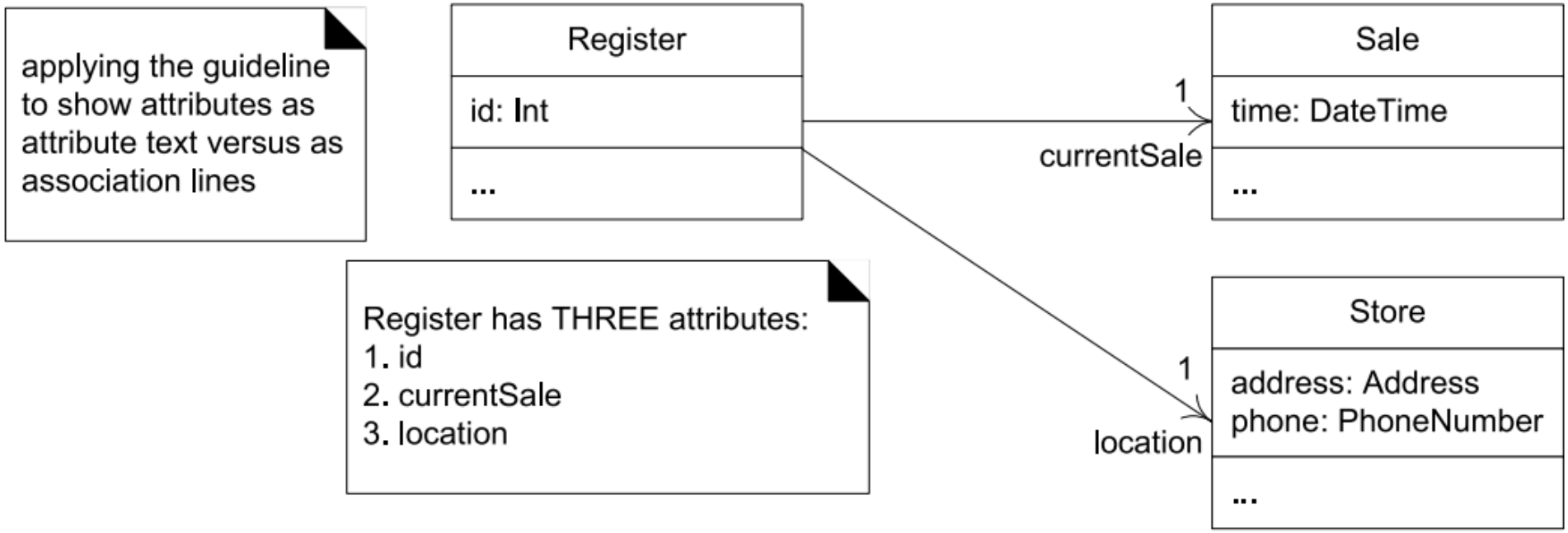
Domain model Vs. CD



Guideline for CD



Guideline for DCD



Guidelines for Attributes and Operations

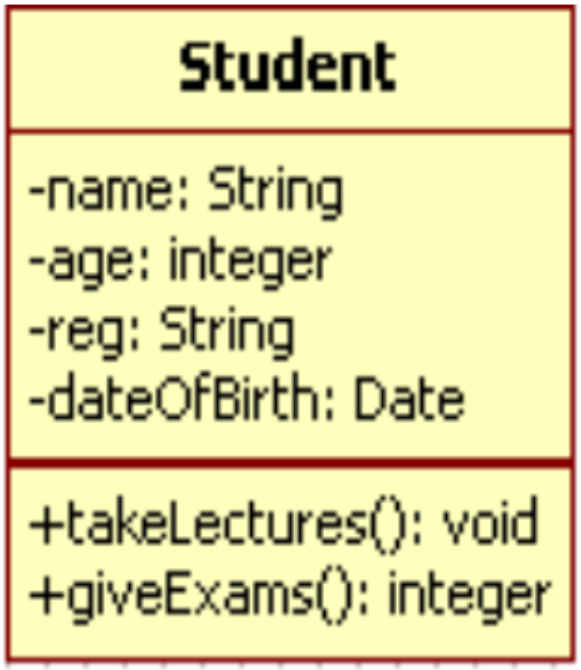
- Each attribute and operation are listed one per line in the appropriate compartment
- Each attribute and operation name should start with a lower case letter
- The only exception to this is a constructor operations of a class, which will have the same name as the class itself in exactly the same uppercase and lowercase format
- Attribute and Operation names should have no spaces between multiple words in the name but should start each word with a capital letter e.g

➢ **giveQuiz instead of givequiz or**

➢ **dateRegistered instead of dateregistered**

Attributes and Operations Types

- The data type for an attribute and return type for an operation can be represented in the class diagram as shown in the figure

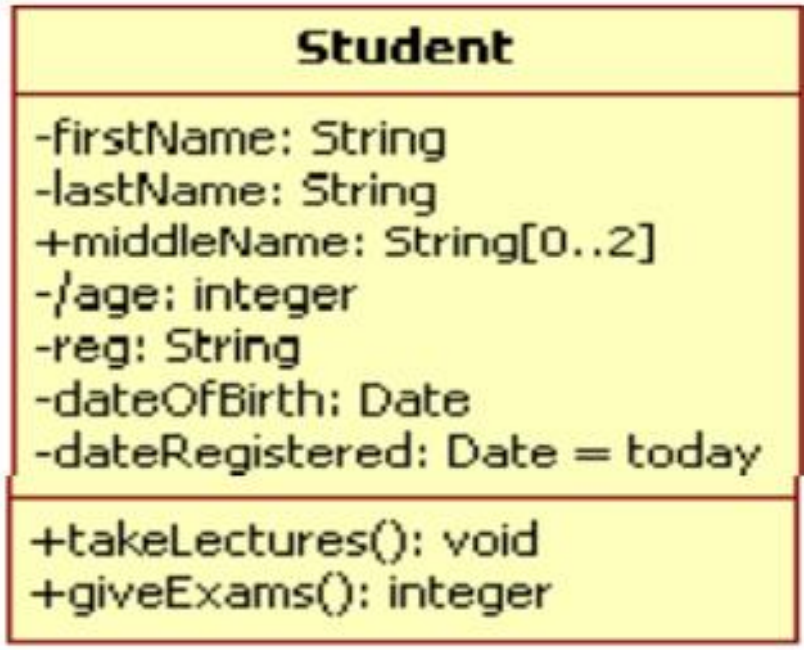


Attributes and Operations Visibility

- Attributes and operations can be assigned a level of visibility on the class diagram with a visibility indicator.

- The visibility of a feature can be defined by either a keyword or a symbol
  - There are three specific types of visibility
  - ➢ Private [represented with a symbol -]
  - ➢ Public [represented with a symbol +]
  - ➢ Protected [represented with a symbol #]
  - There is no default value for visibility

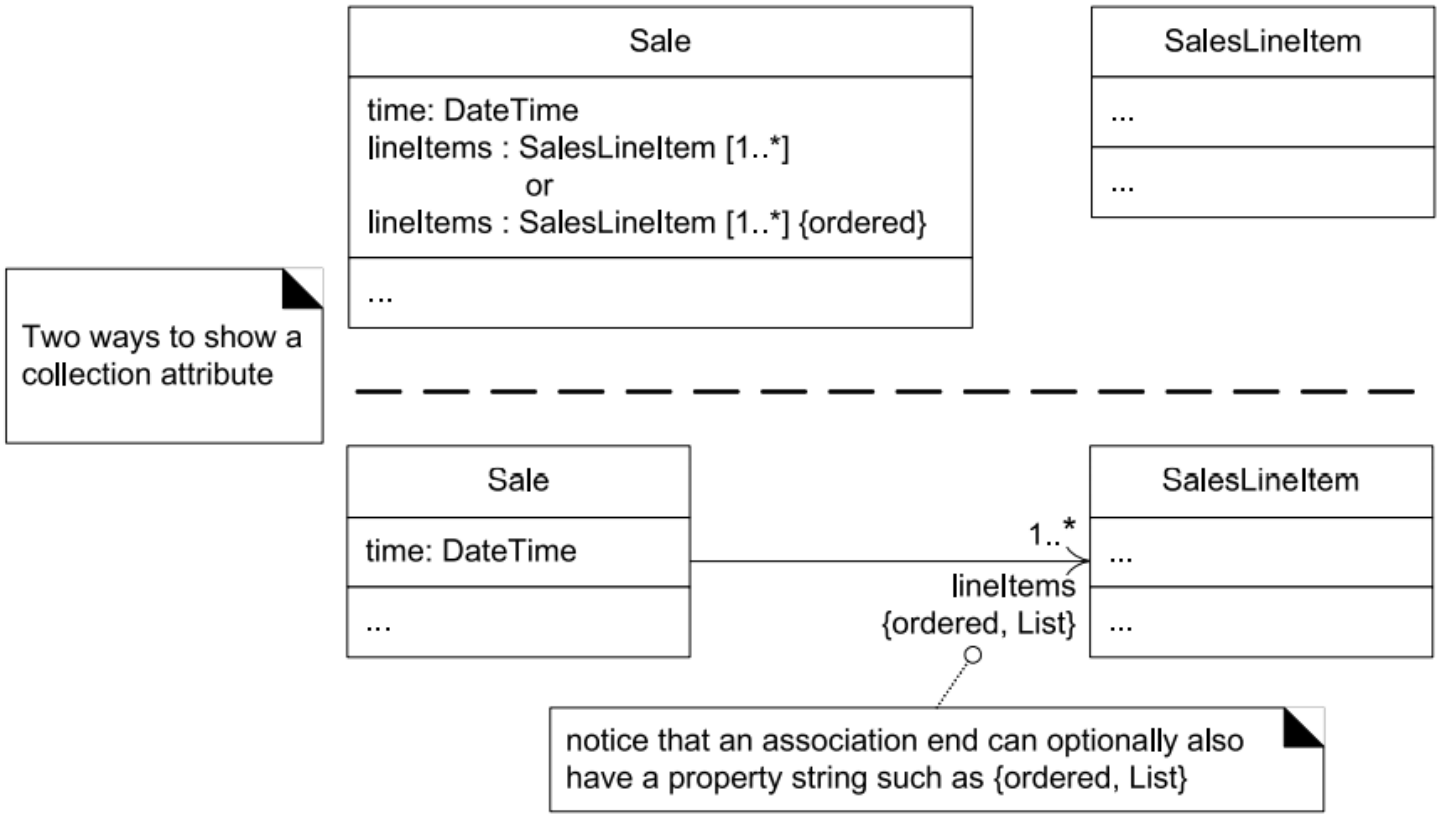
Attributes Properties



Collection Attribute

```
public class Sale
{
private List<SalesLineItem> lineItems =
                            new ArrayList<SalesLineItem>();
// …
}
```

Figure 7.6 shows two ways to illustrate a collection attribute in class diagrams.



Add method name

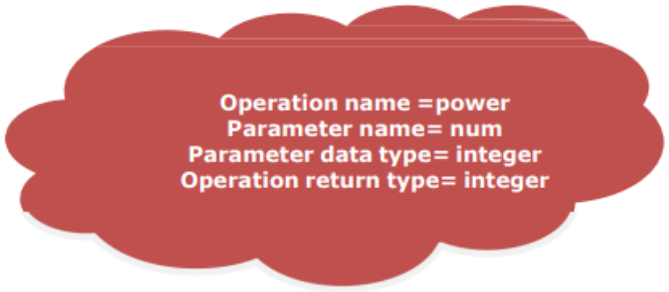
Method names come from interaction diagrams

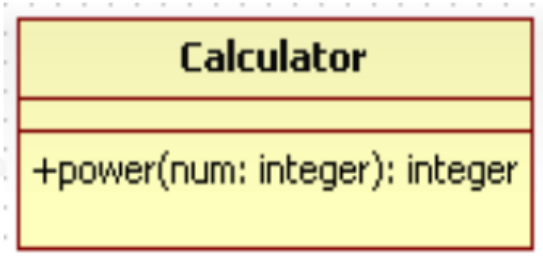
Add methods

- The name, parameter list and return type of an operation are collectively known as its *Signature*

- It is possible to have several operations with the *same name* and *return type* in one class provided that those operations each has a different *parameter list* to the other same name operations.

- Knowing the operation signatures provides a clear specification for the collaboration between operations.

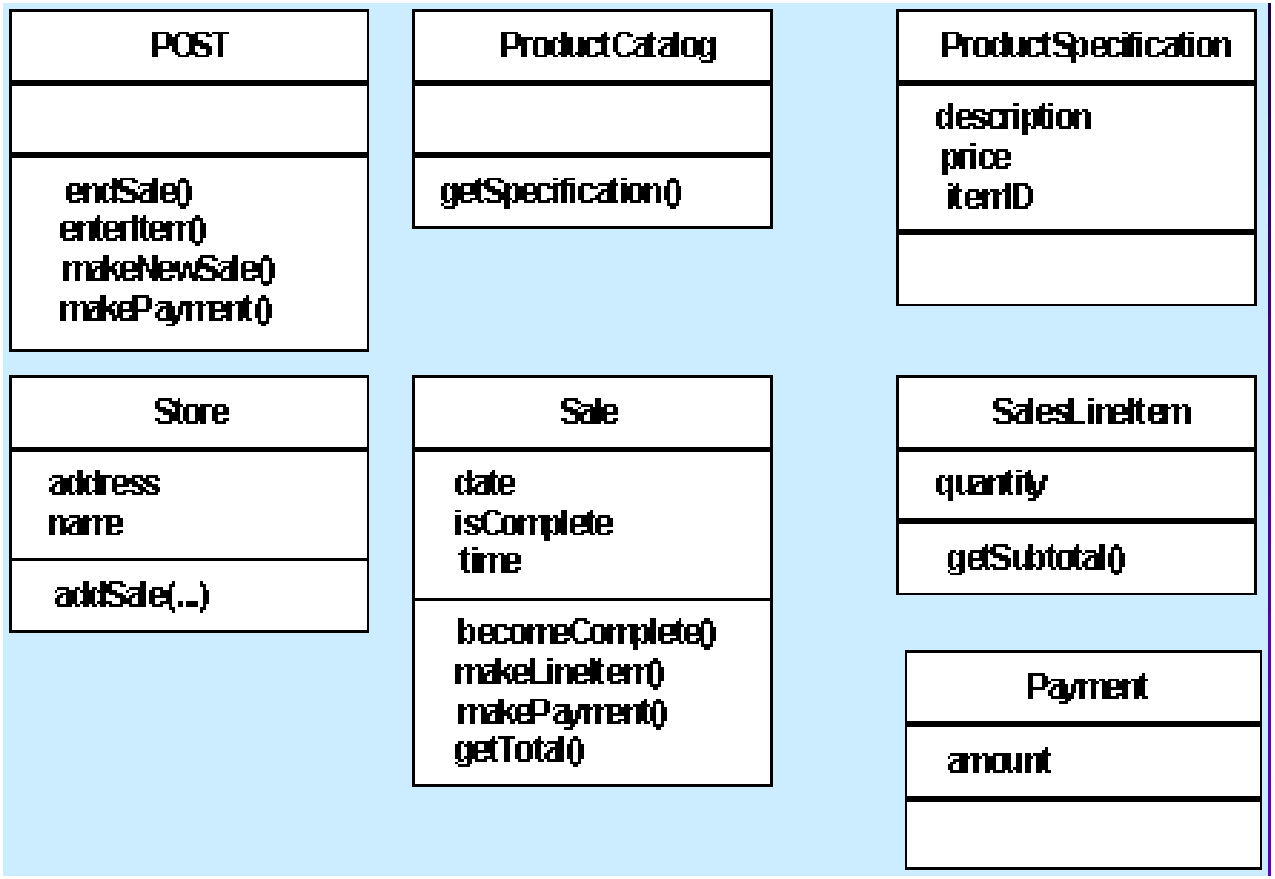




Keywords

Keyword	Meaning	Example Usage
«actor»	classifier is an actor	in class diagram, above classifier name
«interface»	classifier is an interface	in class diagram, above classifier name
{abstract}	abstract element; can't be instantiated	in class diagrams, after classifier name or operation name
{ordered}	a set of objects have some imposed order	in class diagrams, at an association end

After Methods Have Been Added



Issues Related to Method Names

- The following special issues must be considered with respect to method names:
  - Interpretation of the create() message.
  - Depiction of accessing methods.
  - Interpretation of messages to multi-objects.
  - Language-dependent syntax.

Method names: “create” issue

- The create message is the UML language independent form to indicate instantiation and initialization.
  - Equivalent to calling the constructor method of a class
- When translating the design to an OOPL it must be expressed in terms of its idioms for instantiation and initialization.

Method names: Accessing methods

- Accessing methods are those which *retrieve attribute values* (accessor method - get) or set attribute values (mutator method)

- It is common idiom to have an accessor and mutator for each attribute, and to declare all attributes private (to *enforce encapsulation*).

Method names: Language dependent syntax

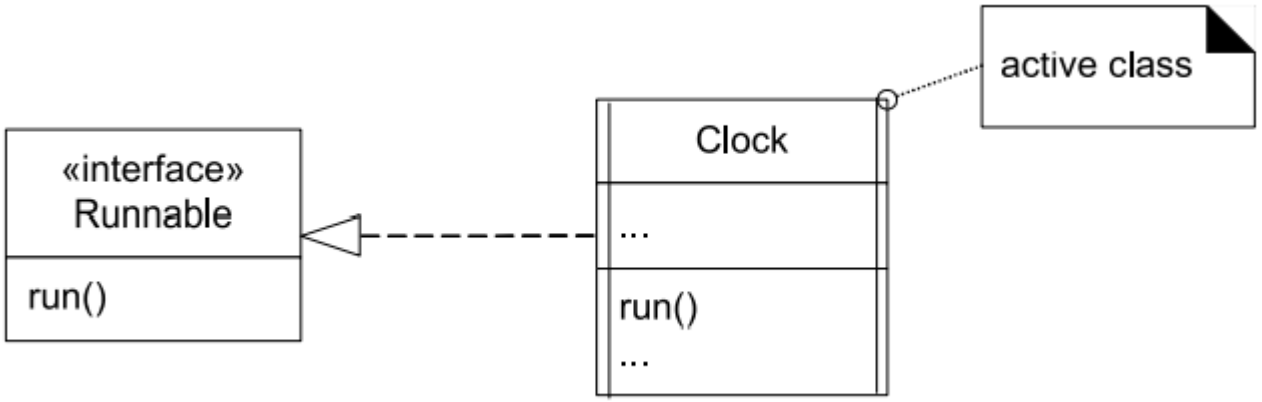
- The basic UML format for methods:

  methodName(parameterList)

- The class diagram should be created by considering the audience.
  - If it is being created in a *CASE tool* with automatic code generation, *full and exhaustive* details are necessary
  - If it is being created for software developers to read, exhaustive low-level detail may *add too much noise*

Active Class

- An **active object** runs on and controls its own thread of execution. Not surprisingly, the class of an active object is an **active class**.



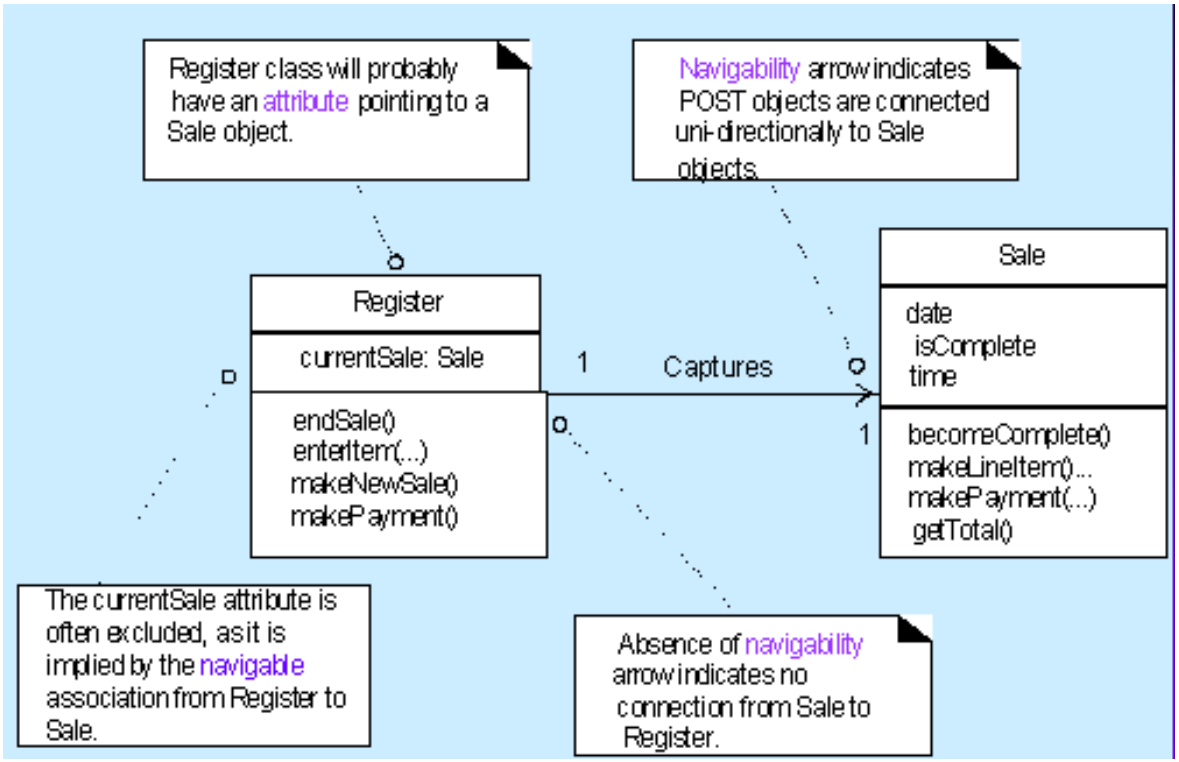
Adding Associations and Navigability

- An object oriented system is built from class types that collaborate with each other by passing messages and receiving responses
- When instance of one class pass messages to instances of another class, an *association is implied* between those two classes
- UML shows an association between the two classes as a *solid line*
- The association can be labeled with a name to indicate the nature of the association
- If an association is labeled then an *arrow head* should be used with the association name to indicate the direction in which the text of the association name should be interpreted.

Adding Navigability

- Navigability is a property of the role which indicates that it is possible to navigate
  - uni-directionally across the association from objects of the source to target class.
- Navigability implies visibility
- Most, if not all, associations in design-oriented class diagrams should be adorned with the necessary navigability arrows.

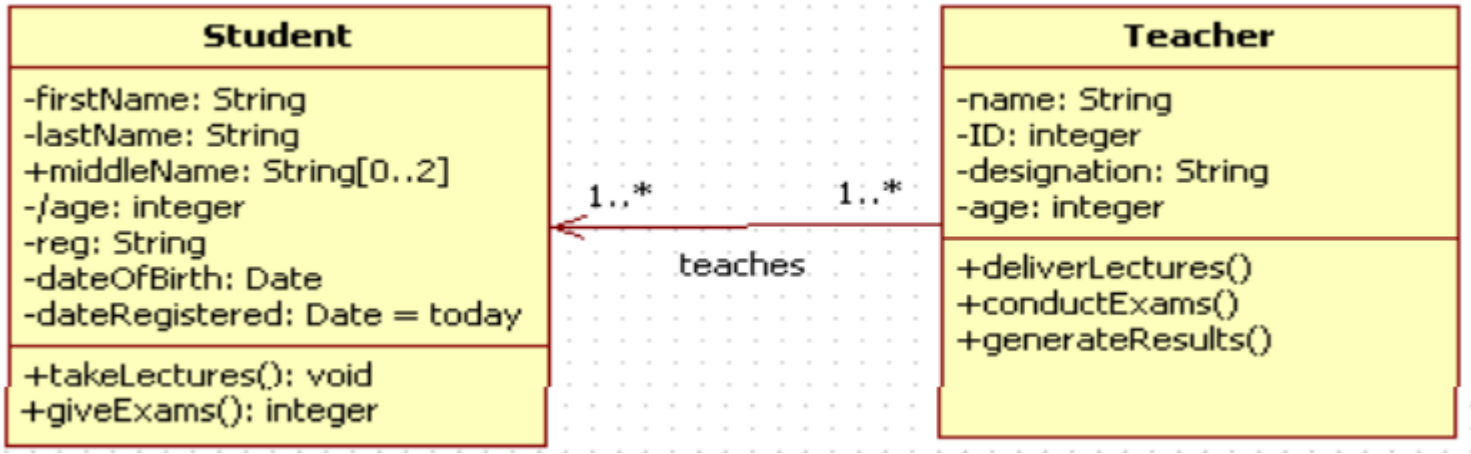
Showing Navigability or Attribute Visibility



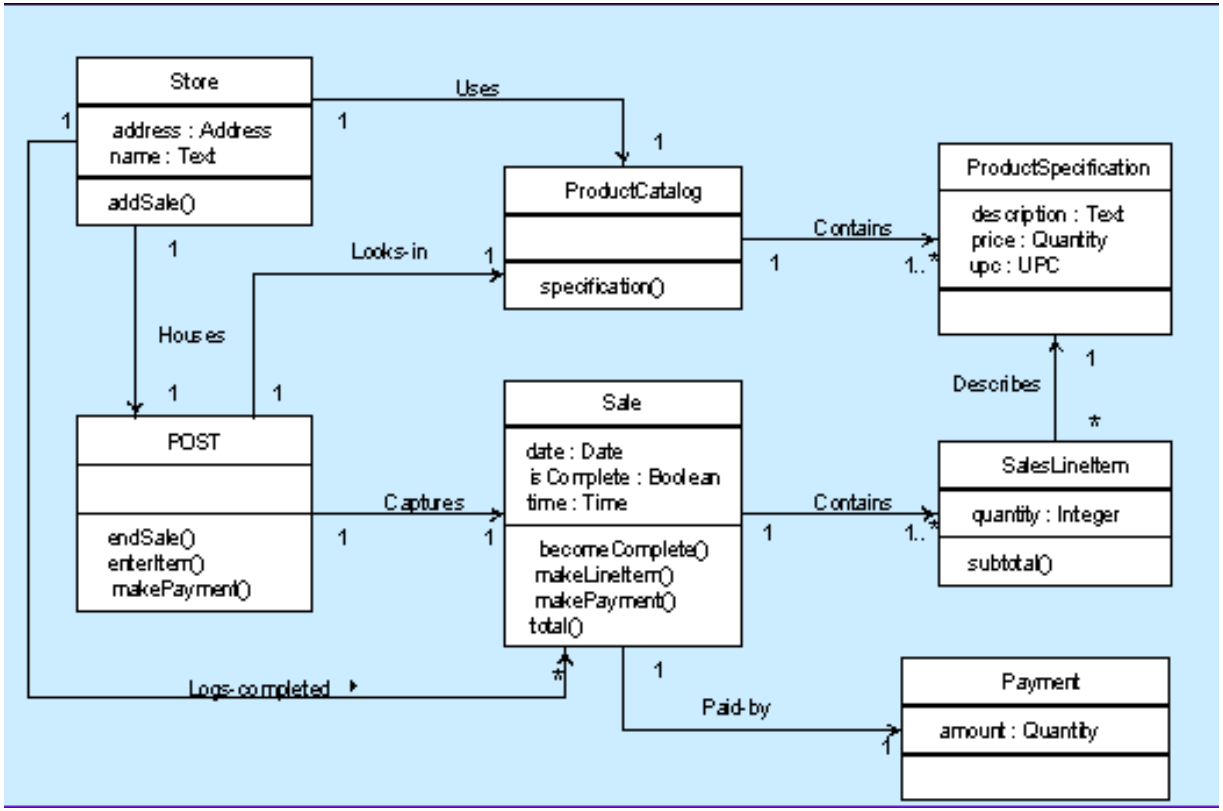
Adding Multiplicity

- In terms of an *association multiplicity* indicates the number of object instances of the class at the far end of an association for one instance of the class at the near end of an association

- Here it indicates that **minimum of 1 teacher and 1 student** will be a part of the association between these two classes and **maximum of [many] teacher and [many] stu**dents will be a part of the association between these two classes



Associations with Navigability

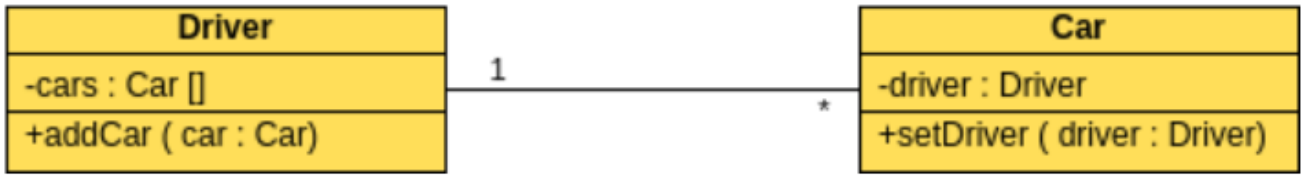


Associations Types

- General association
- Is-A association (generalization)
- Part-Of association
  - Aggregation
  - Composition

Associations Types

- General association
  - A class only uses behaviors/functionalities (methods) of another class but does not change them by overriding them.
  - A class does not inherit another class.
  - A class does not include (own) another class as a public member.
  - Both classes have independent lifetime where disposing of one does not automatically dispose of another.

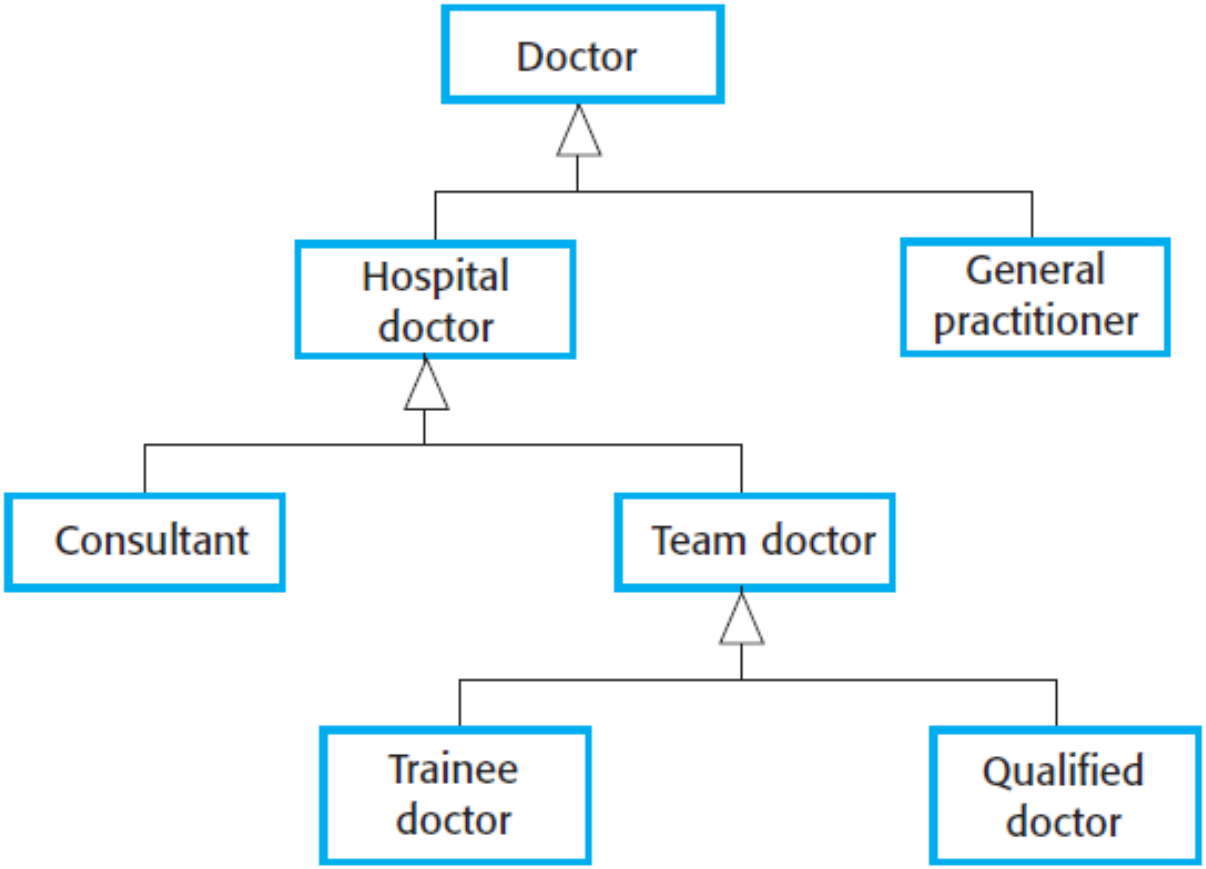


Associations Types

- Is-A Association (Generalization)
  - Sometimes, some entities have few *commonalities* and *differences* among them, and these entities can be *generalized* based on the commonalities.
  - In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.
  - In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.
  - The lower-level classes are subclasses *inherit* the attributes and operations from their superclasses. These lower-level classes then add more *specific* attributes and operations.

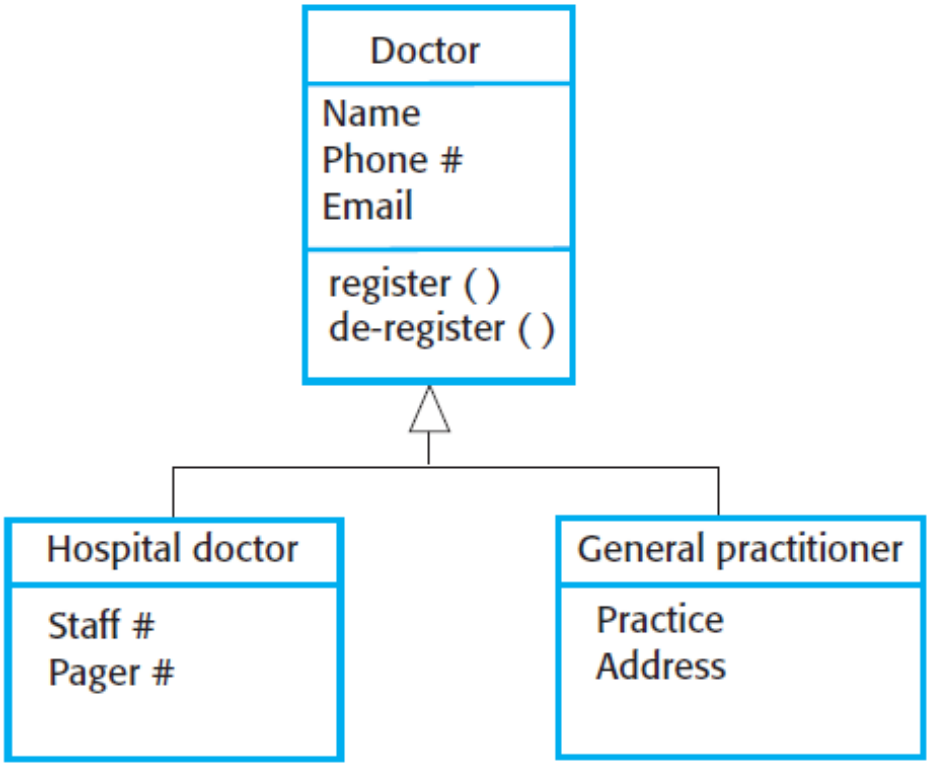
Associations Types

- Is-A Association (Generalization)



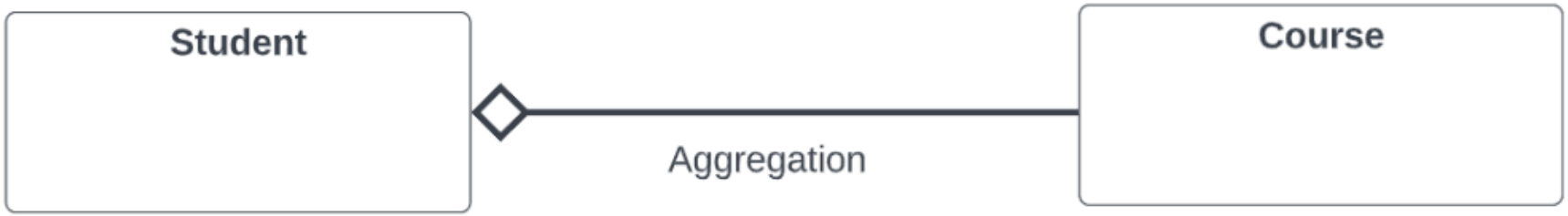
Associations Types

- Is-A Association (Generalization)



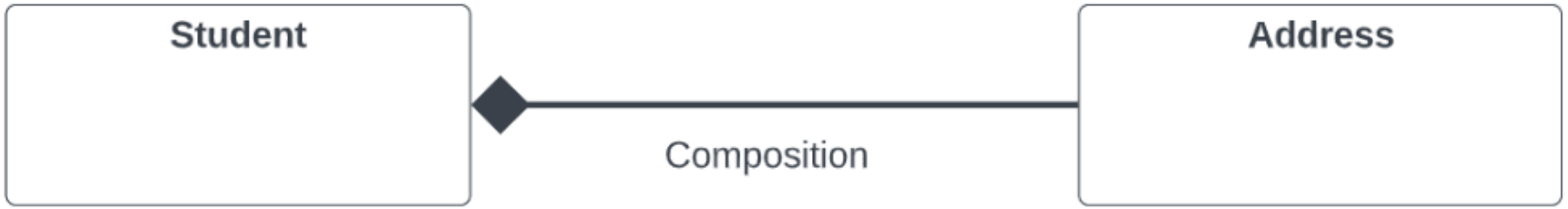
Associations Types -- Aggregation

- Part-A Association (Aggregation (has-A)
- Aggregation is another type of composition ("has a" relation).
- A class (parent) contains a *reference* to another class (child) where both classes can exist *independently*.



Associations Types -- Composition

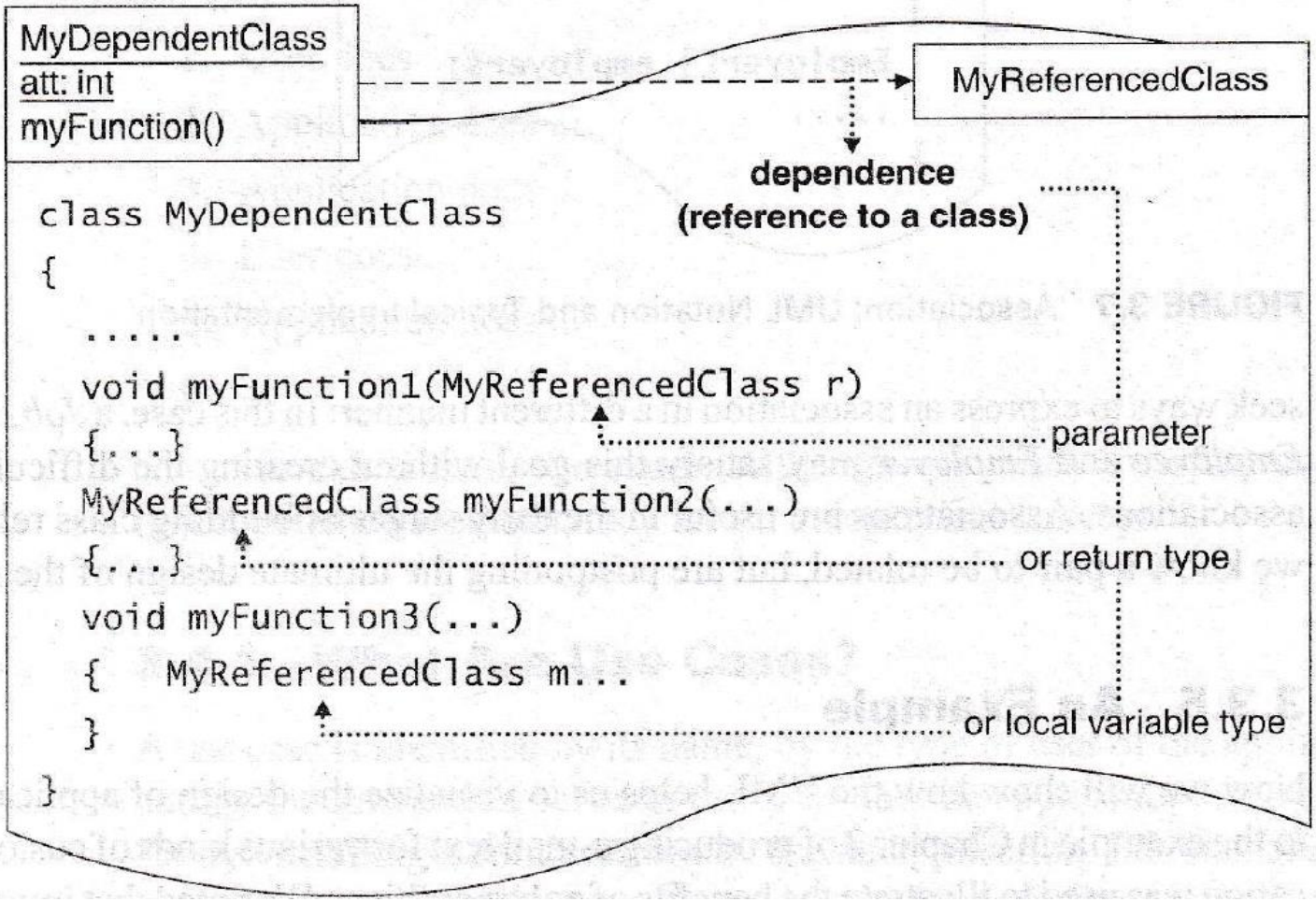
- Part-A Association (Composition (has-A)
- A class (parent) contains a reference to another class (child).
- The child class *doesn't exist* without the parent class.
- Deleting the parent class will also delete the child class.



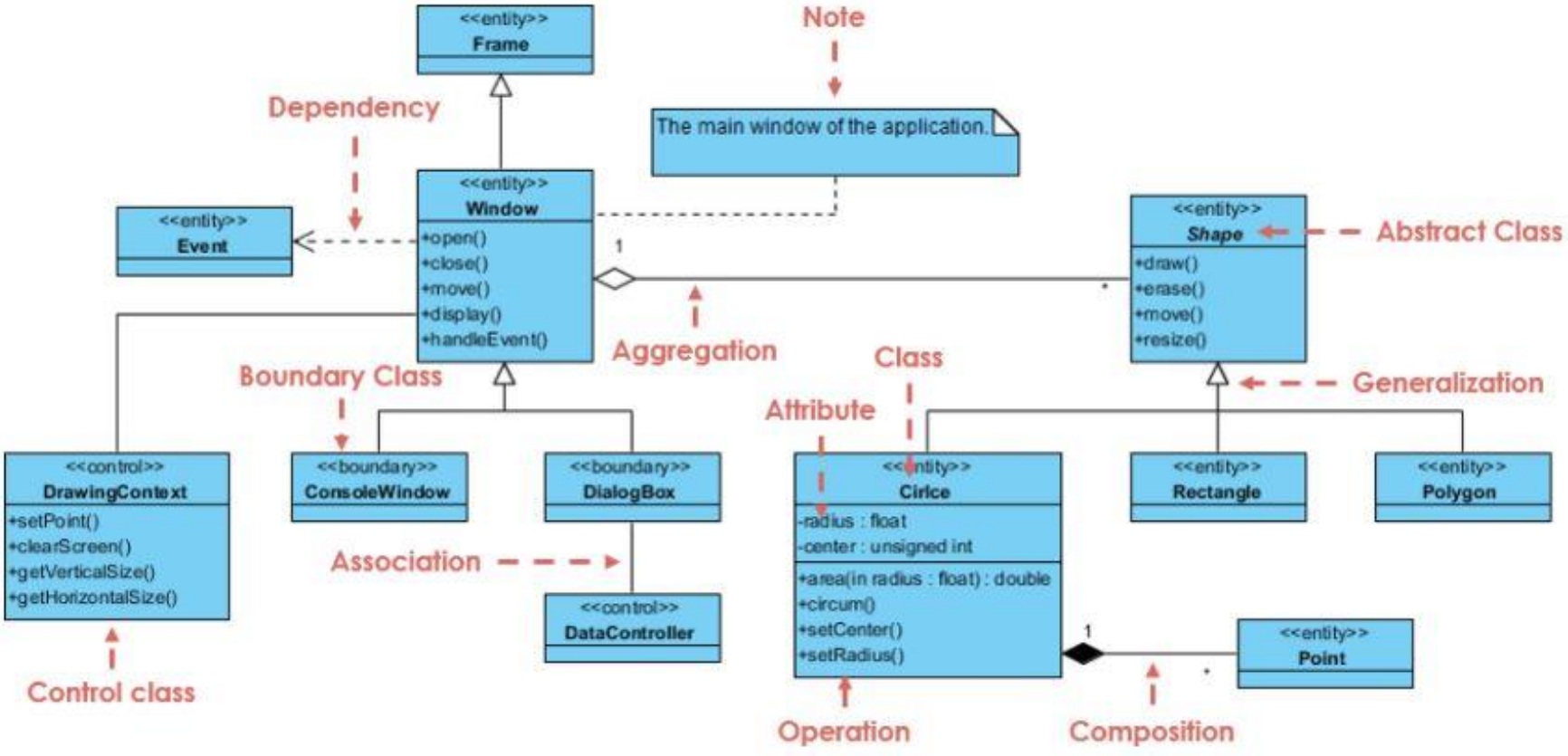
Adding Dependency Relationships

- Dependency relationship indicates that one element (of any kind, including classes, use cases, and so on) has knowledge of another element.
  - A dependency is a using relationship that states a change in specification of one thing affect another thing that uses it, but not necessarily the reverse.

Adding Dependency Relationships



Associations Types -- Composition



Interfaces

- Interfaces allow you to specify what methods a class should implement.
- Interfaces cannot have properties, while abstract classes can
- All interface methods must be public, while abstract class methods is public or protected.
- All methods in an interface are abstract, so they cannot be implemented in code and the abstract keyword is not necessary.
- Classes can implement an interface while inheriting from another class at the same time.
- It helps to solve the problem of multiple inheritance, *how?*

Interfaces

```
class Draw
{     ...
      int setColor(String) {...}
      Pen getStandardPen() {...}
      int getLogoStyle() {...}
      void setColor(int) {...}
      void drawLogo(int, int) {...}
      void speedUpPen(int) {...}
      ...
}
```

Interfaces

- Improves Maintainability, information hiding
- Reduce coupling
- Most designers use a dependency arrow and the «interface» stereotype.