There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated tha

UML Class Diagram

By Pakita Shamoi

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies

C.A.R. Hoare

Designing a system

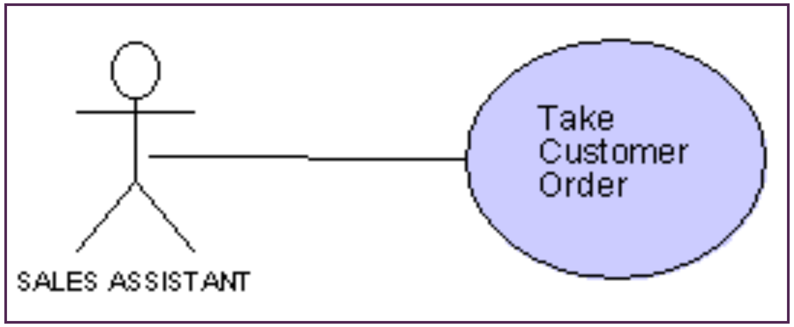
- Identify classes in the system
- Identify the responsibilities of each class
- Identifythe relationships among classes

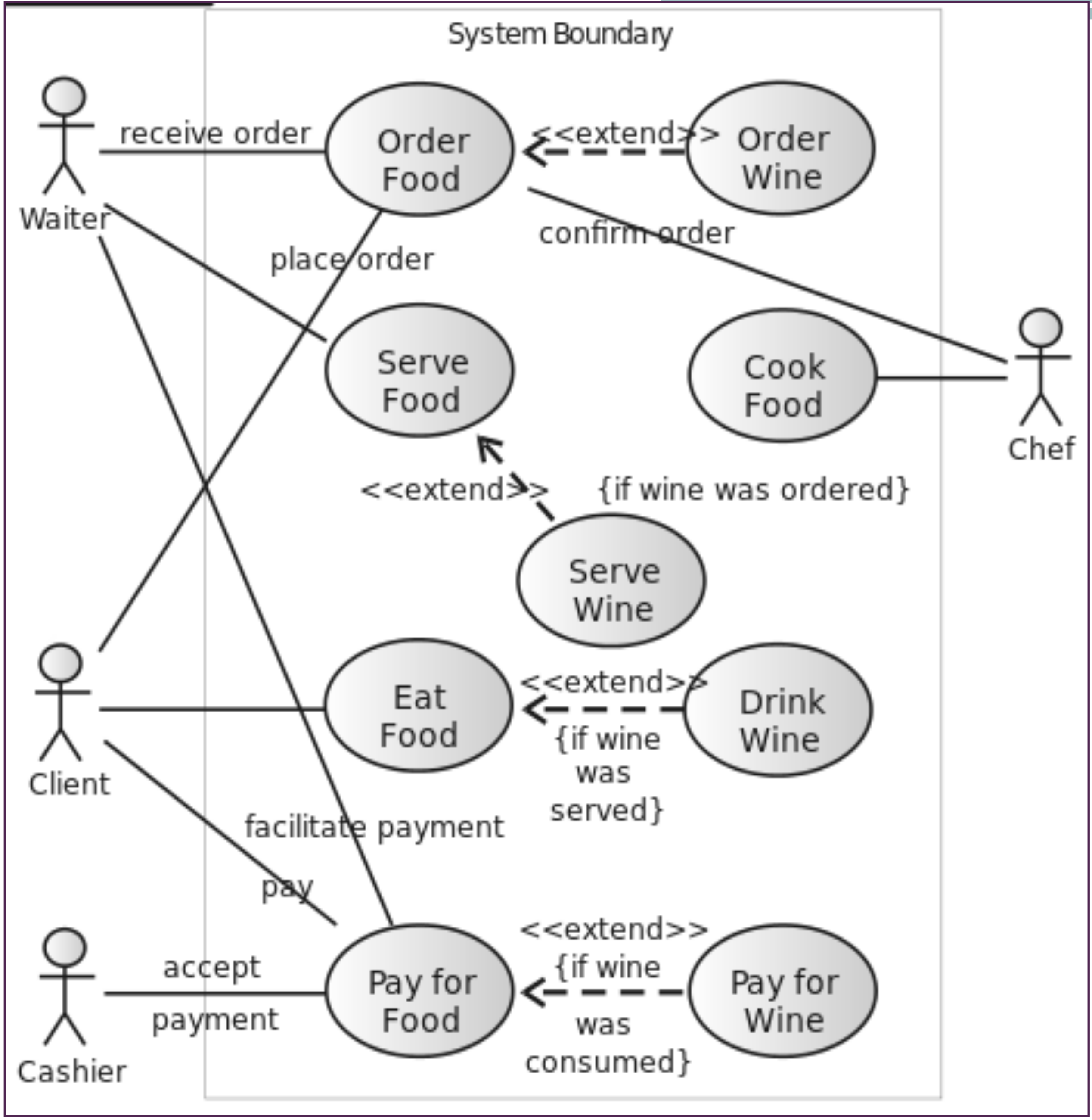
Types of models

- What is a model?
- ***Functional Model*** - presents the functionality of the system from the user's perspective. Example: *Use Case Diagrams*.
- ***Object Model*** - represents the system structure using classes, attributes, operations, and various types of associations among them. Example: *Class Diagrams*.
- ***Dynamic Model*** - describes the internal behavior of the system. Example: *Sequence Diagrams*.

Use Case diagram

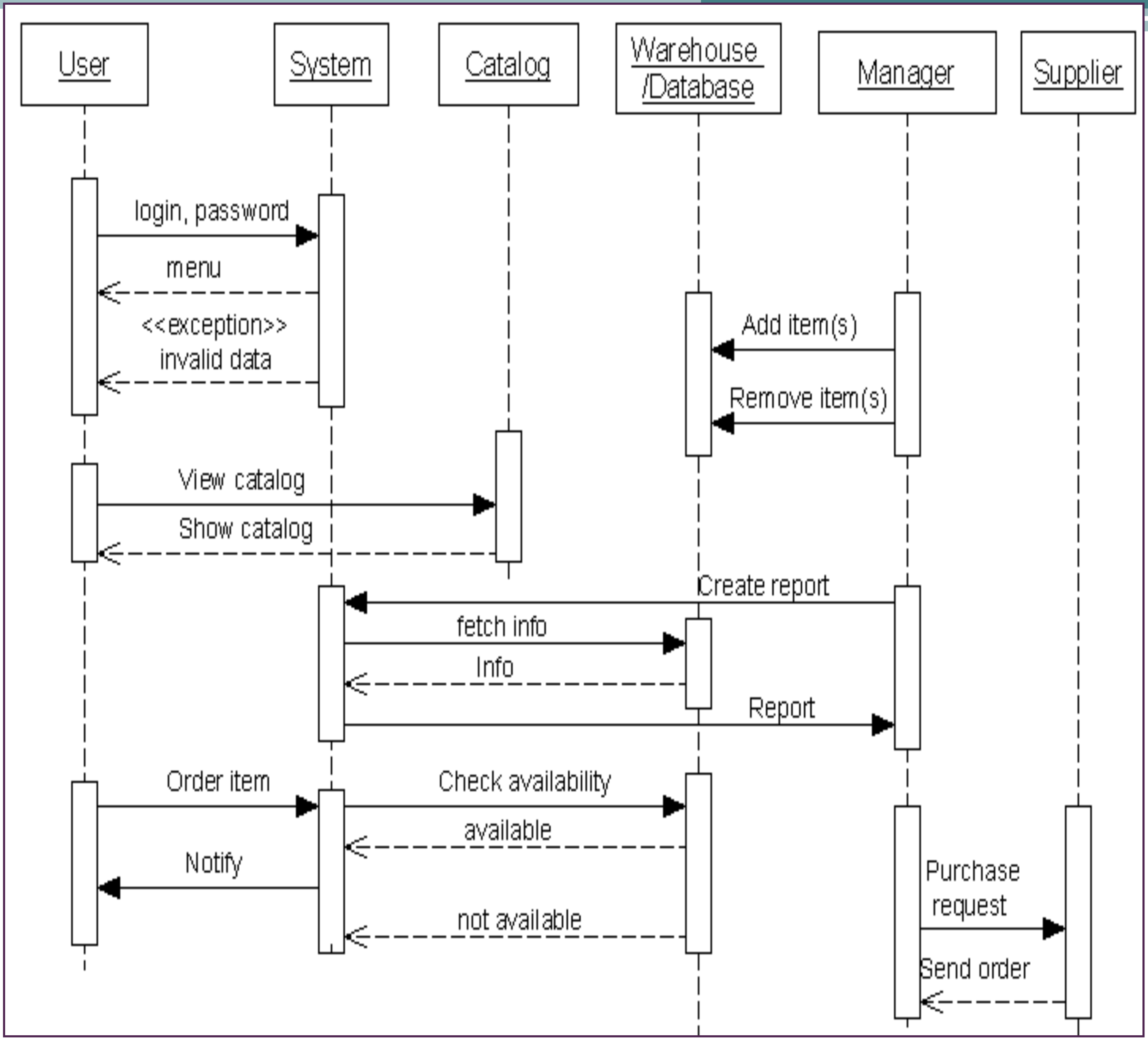
- A **Use Case Diagram** is a type of functional diagram that is used to present a graphical overview of the functionality system affords in terms of **actors** (usually users of the system), their actions - **use cases**, and any **dependencies** between those use cases.




System Boundary
Waiter
receive order
Order Food
<<extend>>
Order Wine
confirm order
place order
Serve Food
Cook Food
Chef
<<extend>>
{if wine was ordered}
Serve Wine
Client
Eat Food
<<extend>>
{if wine was served}
Drink Wine
facilitate payment
pay
Cashier
accept payment
Pay for Food
<<extend>>
{if wine was consumed}
Pay for Wine

Sequence diagrams

- The main convenience of a ***Sequence Diagram*** is that it allows to visually specify simple dynamic scenarios.
- *Sequence Diagrams* are composed of objects, activations and messages


User
System
Catalog
Warehouse /Database
Manager
Supplier
login, password
menu
<<exception>> invalid data
Add item(s)
Remove item(s)
View catalog
Show catalog
Create report
fetch info
Info
Report
Order item
Check availability
available
Notify
not available
Purchase request
Send order

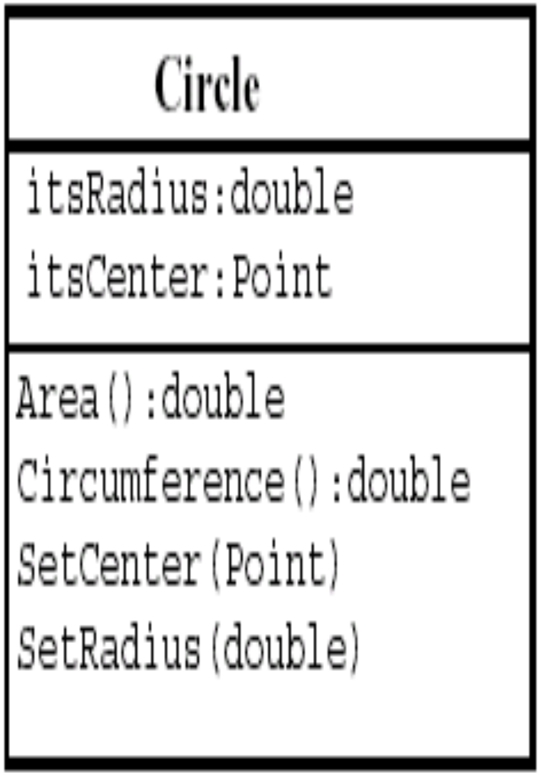
What is a Class Diagram?

- A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them.
  - A graphical representation of a static view on declarative static elements.
- A central modeling technique that runs through nearly all object-oriented methods.
- The richest notation in UML.

The class icon

- It's a rectangle divided into three compartments.

- The class icon has
  - Name
  - Attributes
  - Methods



Circle

itsRadius:double
itsCenter:Point

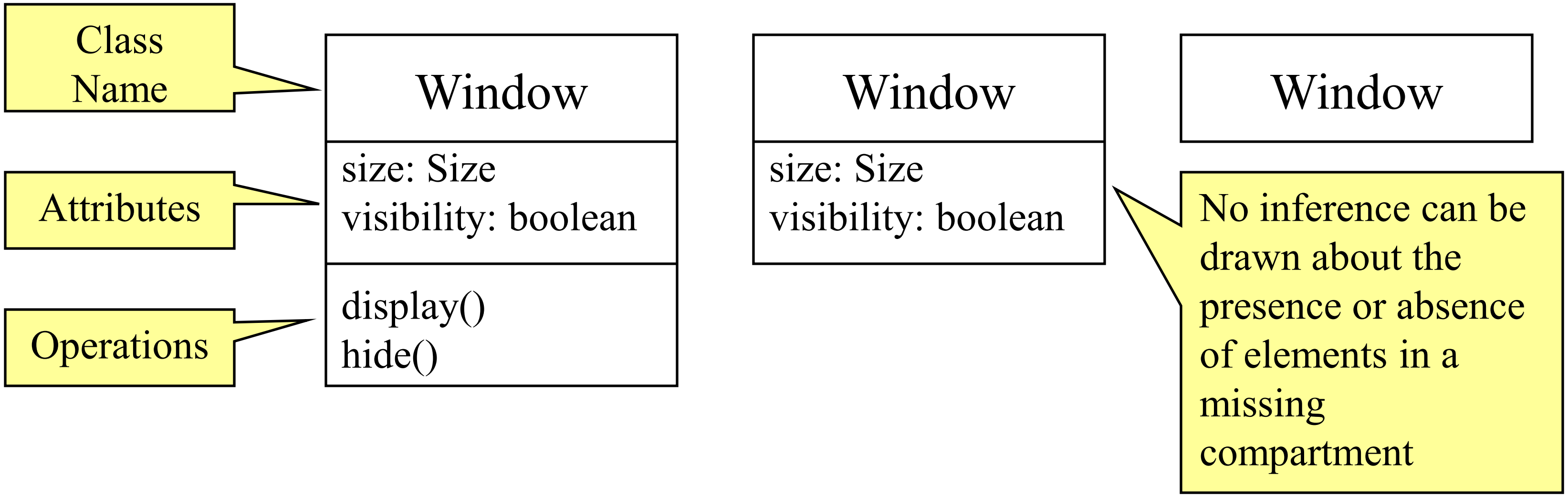
Area():double
Circumference():double
SetCenter(Point)
SetRadius(double)

Essential Elements of a UML Class Diagram

- Class
- Attributes
- Operations
- Relationships
    - Associations
    - Generalization
    - Dependency
    - Realization
- Comments and Notes

Classes

- A class is the description of a set of objects having similar attributes, operations, relationships and behavior.



Class Name

Attributes

Operations

Window

size: Size
visibility: boolean

display()
hide()

Window

size: Size
visibility: boolean

Window

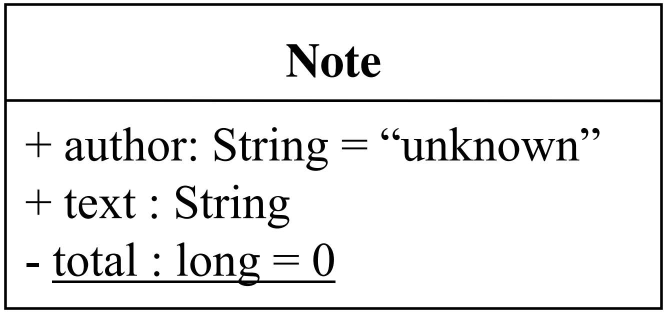
No inference can be drawn about the presence or absence of elements in a missing compartment

Attributes

- Classes have attributes that describe the characteristics of their objects.
- Attributes are atomic entities with no responsibilities.
- Attribute syntax (partial):
  - *[visibility] name [: type] [= defaultValue]*
- Class scope attributes are underlined

Attributes (cont.)

- Example (Java implementation):



Note

+ author: String = “unknown”
+ text : String
- total : long = 0

```
public class Note
{
  public String author = “unknown”;
  public String text;
  private static long total = 0;
  ...
}
```

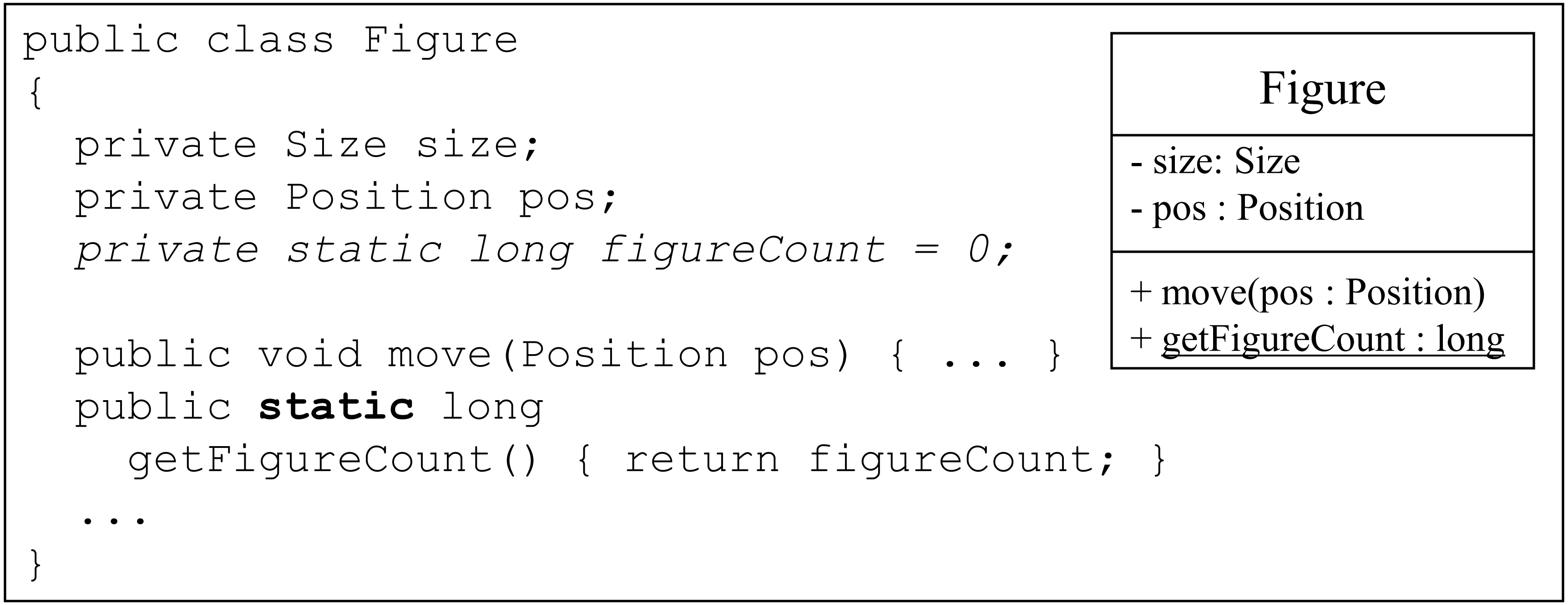
Operations

- Operations are the processes that objects of a class (or the class itself) carry out.
- Operation syntax (partial):
  - *[visibility] name ([parameter-list]) [: return-type]*
  - where *parameter-list* is a comma separated list of formal parameters, each specified using the syntax: *name : type [= defaultValue]*
- Example:

  max(a : int, b : int) : int

Operations (cont.)

- Class scope operations are underlined.
- Java implementation:



```
public class Figure
{
  private Size size;
  private Position pos;
  private static long figureCount = 0;

  public void move(Position pos) { ... }
  public static long
    getFigureCount() { return figureCount; }
  ...
}
```

Figure

- size: Size
- pos : Position

+ move(pos : Position)
+ getFigureCount : long

Visibility

- Visibility describes whether an attribute or operation is visible and can be referenced from classes other than the one in which they are defined.

- UML provides four visibility abbreviations:
    - + (public)
    - – (private)
    - # (protected)
    - ~ (package)

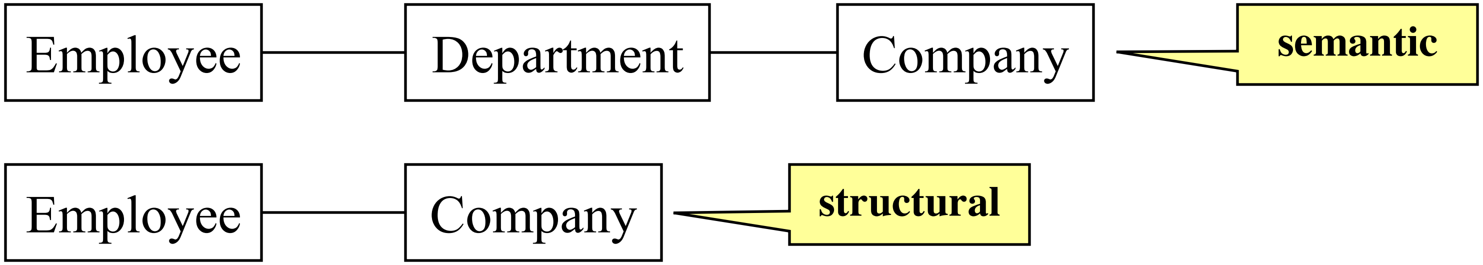
Multiplicity

- Multiplicity
  - Provides a lower and upper bound on the number of instances.

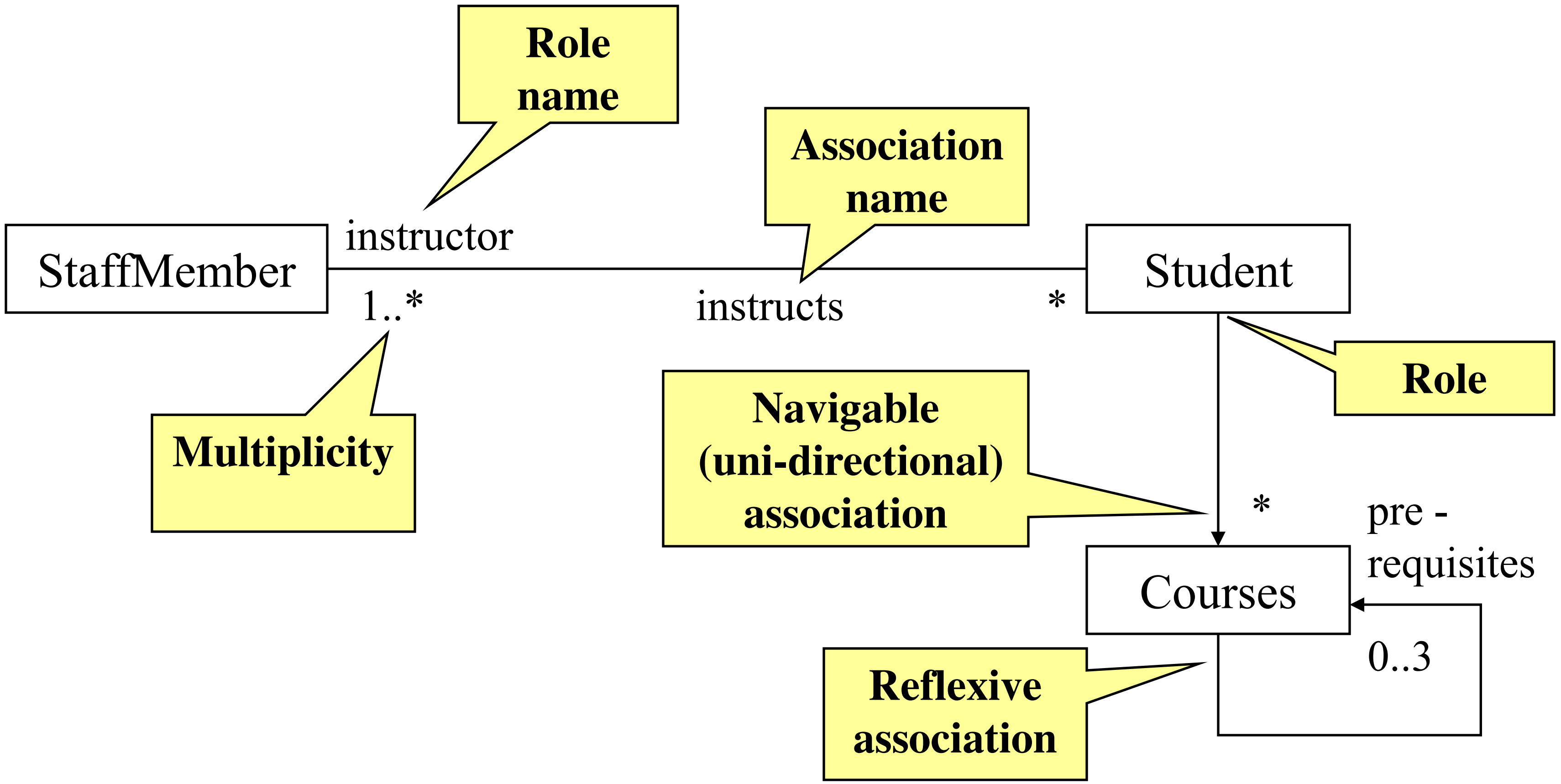
Exactly one	1
Zero or more (unlimited)	* (0..*)
One or more	1..*
Zero or one (optional association)	0..1
Specified range	2..4
Multiple, disjoint ranges	2, 4..6, 8

Associations

- A semantic relationship between two or more classes that specifies connections among their instances.
- A structural relationship, specifying that objects of one class are connected to objects of a second (possibly the same) class.
- Example: “An Employee works for a Company”
- An association between two classes indicates that objects at one end of an association “recognize” objects at the other end and may send messages to them.



Associations (cont.)



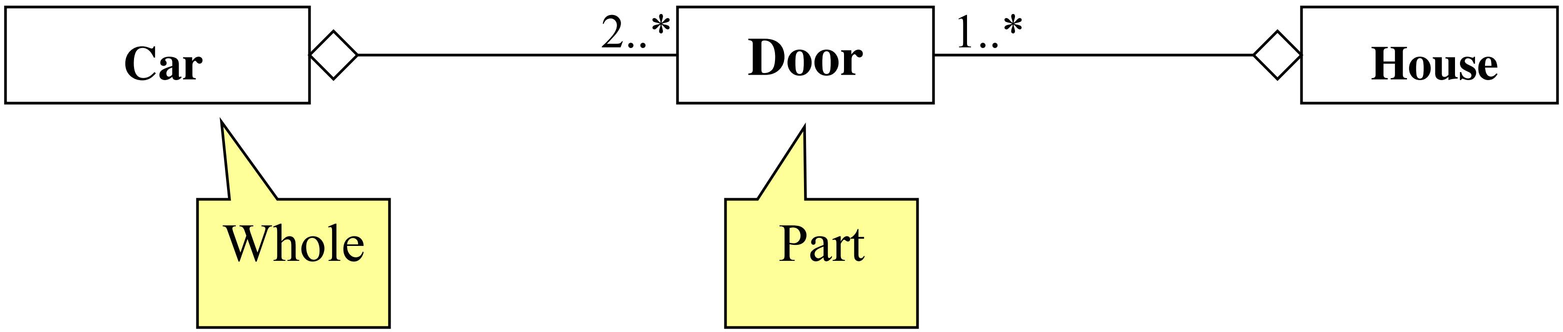
Associations (cont.)

- To clarify its meaning, an association may be named.
  - The name is represented as a label placed midway along the association line.
  - Usually a verb or a verb phrase.

- A **role** is an end of an association where it connects to a class.
  - May be named to indicate the role played by the class attached to the end of the association path.
    - Usually a noun or noun phrase
    - Mandatory for reflexive associations

Aggregation

- A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
  - Models a "is a part-part of" relationship.



Aggregation (cont.)

- Aggregation tests:
  - Is the phrase “part of” used to describe the relationship?
    - **A door is “part of” a car**
  - Are some operations on the whole automatically applied to its parts?
    - **Move the car, move the door.**
  - Are some attribute values propagated from the whole to all or some of its parts?
    - **The car is blue, therefore the door is blue.**
  - Is there an intrinsic asymmetry to the relationship where one class is subordinate to the other?
    - A door **is** part of a car. A car **is not** part of a door.

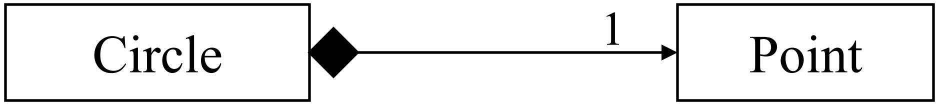
Aggregation (cont.)

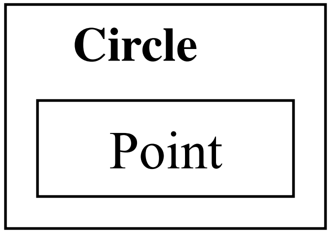
- Java implementation:

```
public class Car {
  private Vector doors = new Vector();
  public void addDoor(Door door) { ... }
  ...
}

public static void main(String[] args)
{
  Door door = new Door();
  House house = new House(door);
  Car car = new Car();
  car.addDoor(door);
  ...
}
```

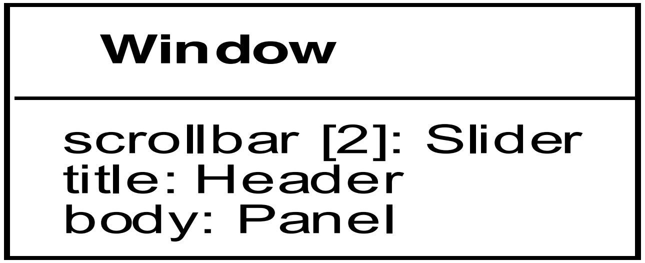
Composition

- A strong form of aggregation
  - **The whole is the sole owner of its part.**
    - The part object may belong to only one whole
  - Multiplicity on the whole side must be zero or one.
  - The life time of the part is dependent upon the whole.
    - The composite must manage the creation and destruction of its parts.



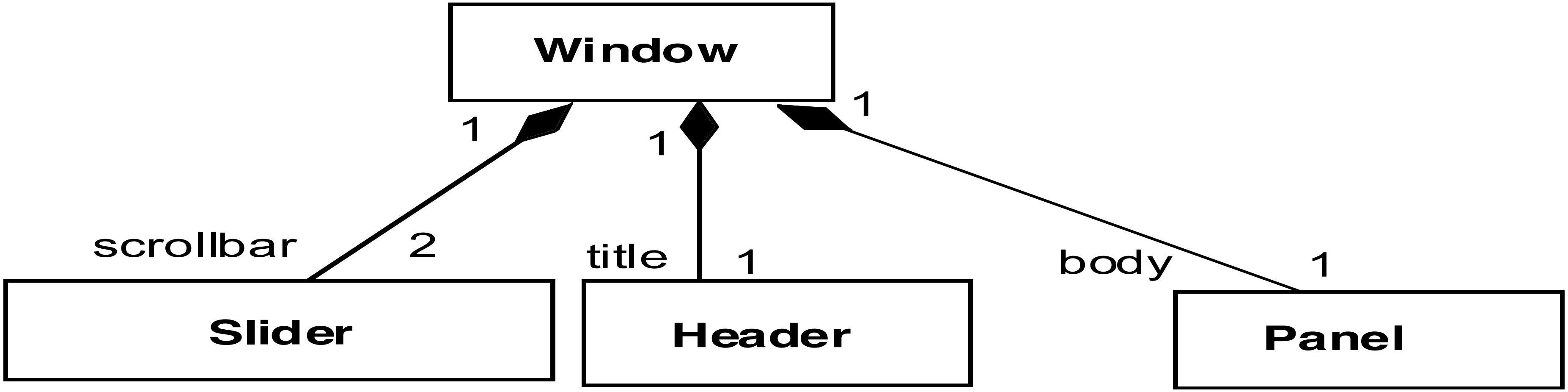


Composition



Window

scrollbar [2]: Slider
title: Header
body: Panel



Window

1

1

1

scrollbar

2

title

1

body

1

Slider

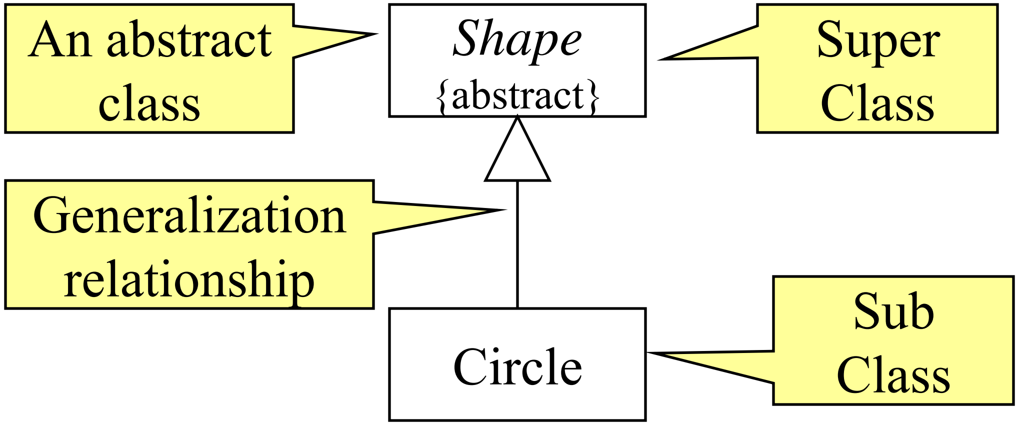
Header

Panel

Generalization

- Indicates inheritance
- Indicates objects of the specialized class (subclass) are objects of the generalized class (super-class).
    - "is kind of" relationship.

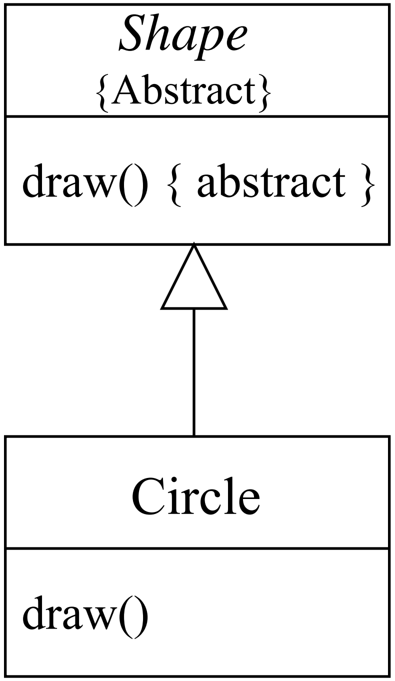
{abstract} is a tagged value that indicates that the class is abstract. The name of an abstract class should be italicized



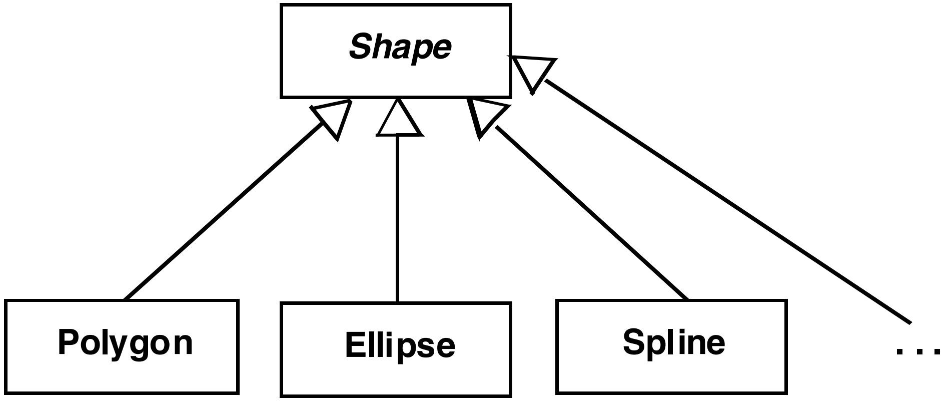
Generalization

- Java implementation:

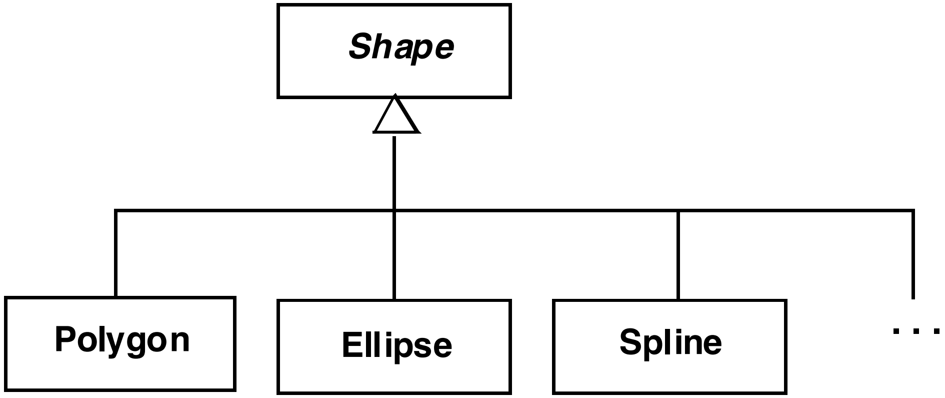
```
public abstract class Shape
{
  public abstract void draw();
  ...
}

public class Circle extends Shape
{
  public void draw() { ... }
  ...
}
```



Generalization



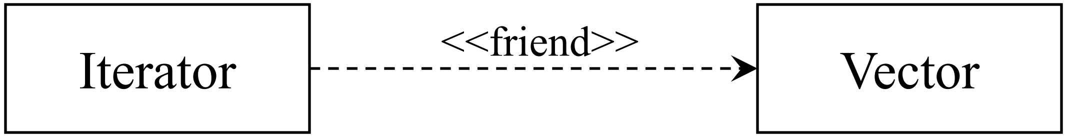
Separate Target Style



Shared Target Style

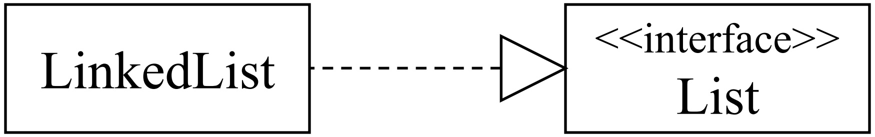
Dependency

- A dependency is a relation between two classes in which a change in one may force changes in the other although there is no explicit association between them.
- A stereotype may be used to denote the type of the dependency.
  - Example - a class calls a class scope operation of another class.



Realization

- A realization relationship indicates that one class implements a behavior specified by another class (an interface or protocol).
- An interface can be realized by many classes.
- A class may realize many interfaces.



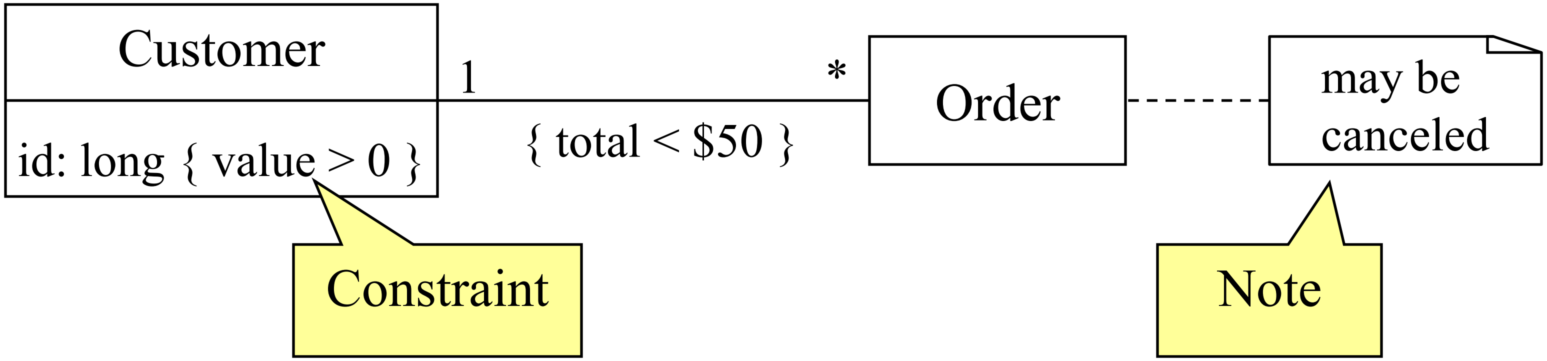
Realization (cont.)

- Java implementation:

```
public interface List
{
  boolean add(Object o);
  ...
}

public class LinkedList implements List
{
  public boolean add(Object o) { ... }
  ...
}
```

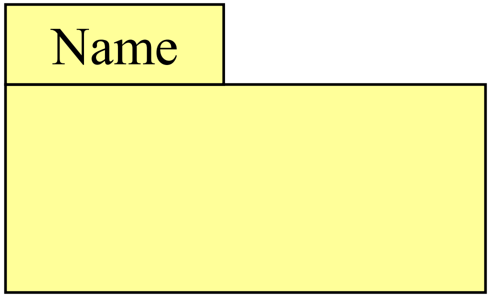
Constraint Rules and Notes

- **Constraints** and **notes** annotate among other things associations, attributes, operations and classes.
- Constraints are semantic restrictions noted as expressions.
  - UML offers many pre-defined constraints.



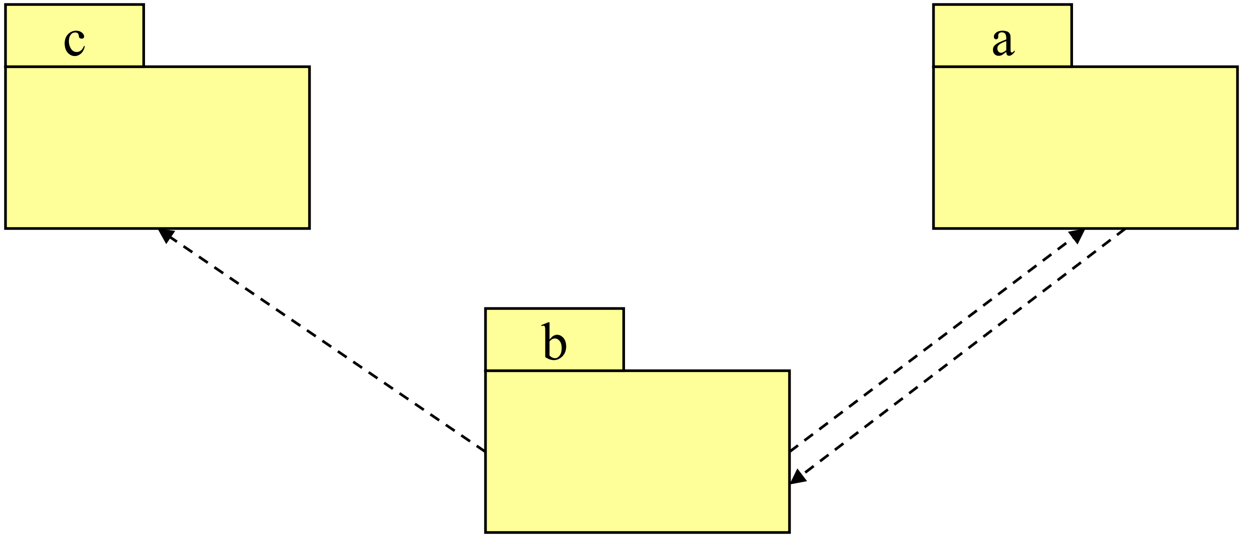
UML Packages

- A package is a general purpose grouping mechanism.
  - Can be used to group any UML element (e.g. use case, actors, classes, components and other packages.
- Commonly used for specifying the logical distribution of classes.



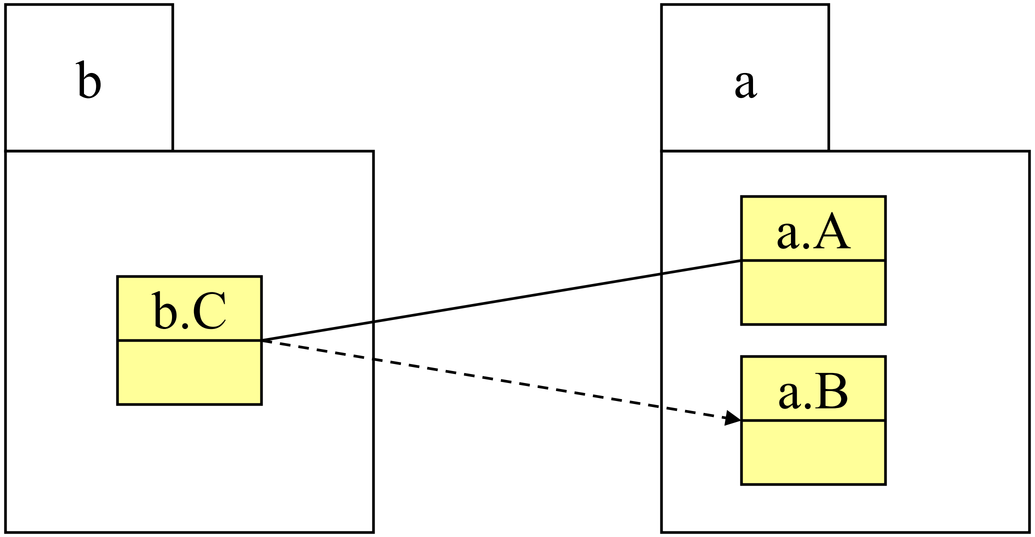
Packages and Class Diagrams

- Emphasize the logical structure of the system (High level view)



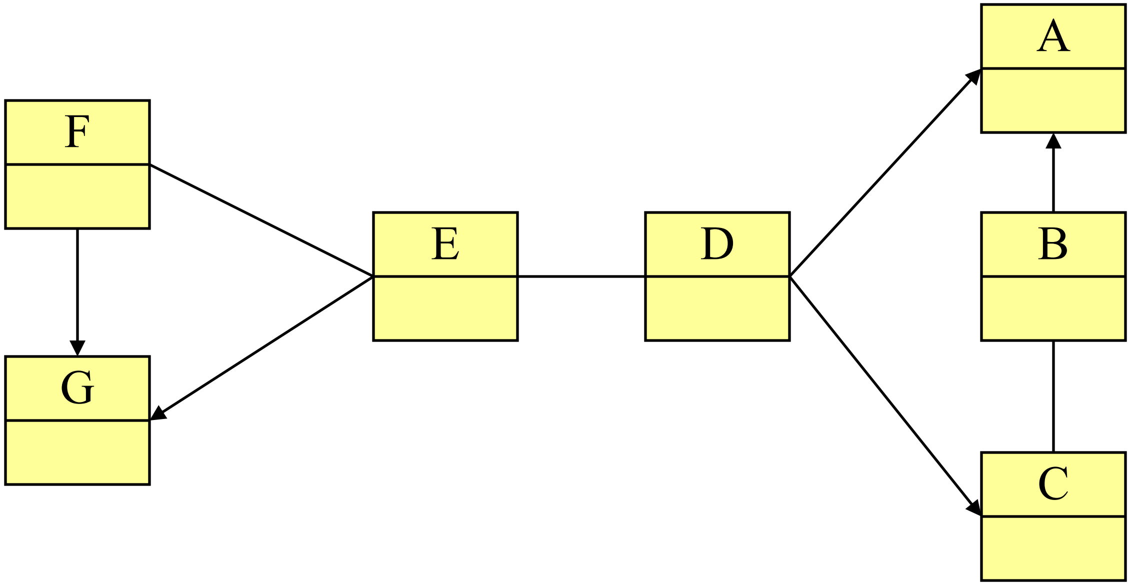
Packages and Class Diagrams (cont.)

- Emphasize the interface between packages by showing relations and dependencies between public classes

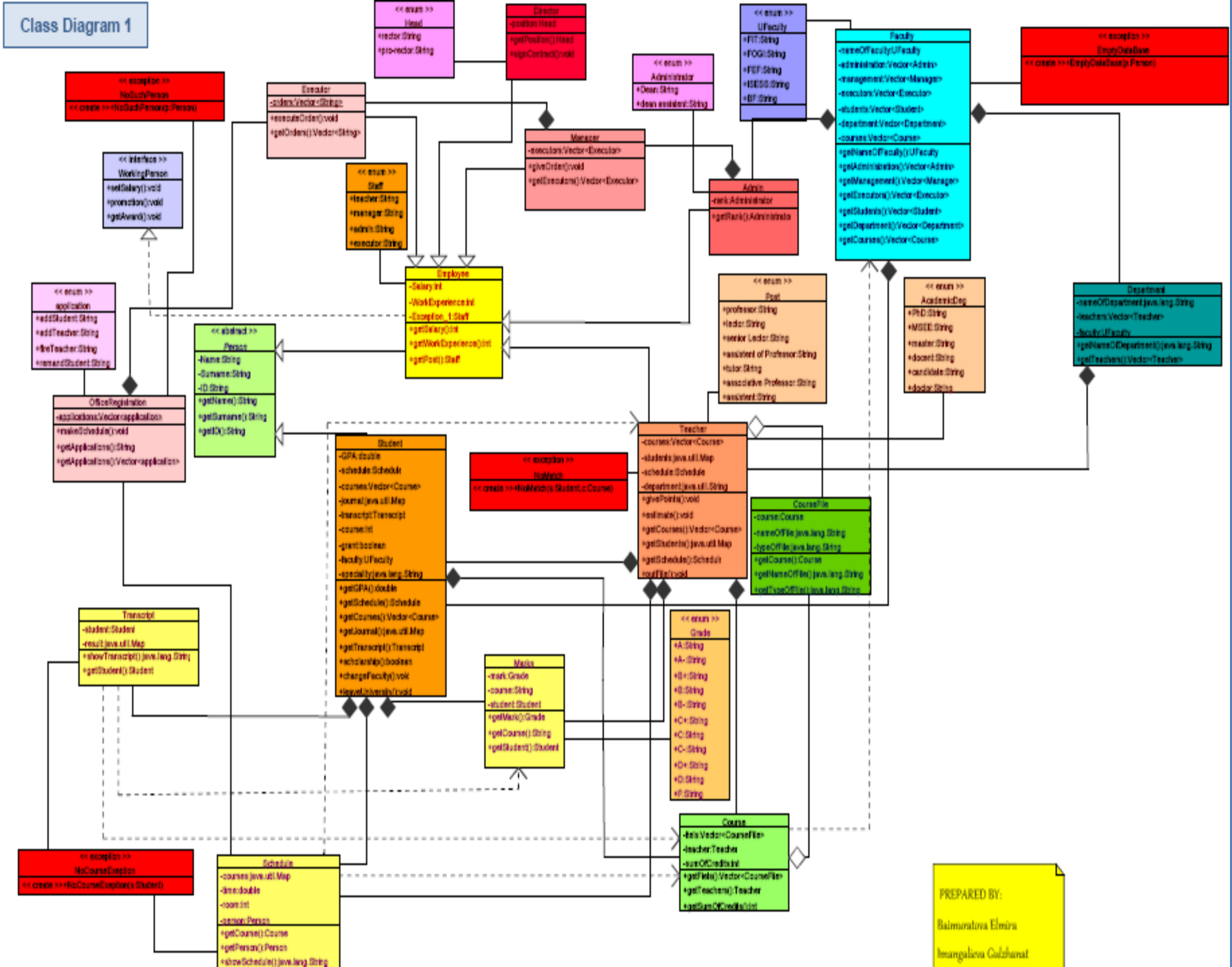


Packages and Class Diagrams (cont.)

- Add package information to class diagrams



Class Diagram 1



<< enum >>
Head
+rector:String
+pro-rector:String

Director
-position:Head
+getPosition():Head
+signContract():void

<< enum >>
Administrator
+Dean:String
+dean assistent:String

<< enum >>
UFaculty
+FIT:String
+FOGI:String
+FEF:String
+ISESS:String
+BF:String

Faculty
-nameOfFaculty:UFaculty
-administration:Vector<Admin>
-management:Vector<Manager>
-executors:Vector<Executor>
-students:Vector<Student>
-department:Vector<Department>
-courses:Vector<Course>
+getNameOfFaculty():UFaculty
+getAdministration():Vector<Admin>
+getManagement():Vector<Manager>
+getExecutors():Vector<Executor>
+getStudents():Vector<Student>
+getDepartment():Vector<Department>
+getCourses():Vector<Course>

<< exception >>
EmptyDataBase
<< create >>+EmptyDataBase(p:Person)

<< exception >>
NoSuchPerson
<< create >>+NoSuchPerson(p:Person)

Executor
-orders:Vector<String>
+executeOrder():void
+getOrders():Vector<String>

Manager
-executors:Vector<Executor>
+giveOrder():void
+getExecutors():Vector<Executor>

Admin
-rank:Administrator
+getRank():Administrato

<< interface >>
WorkingPerson
+setSalary():void
+promotion():void
+getAward():void

<< enum >>
Staff
+teacher:String
+manager:String
+admin:String
+executor:String

Employee
-Salary:int
-WorkExperience:int
-Exception_1:Staff
+getSalary():int
+getWorkExperience():int
+getPost():Staff

<< enum >>
application
+addStudent:String
+addTeacher:String
+fireTeacher:String
+remandStudent:String

<< abstract >>
Person
-Name:String
-Surname:String
-ID:String
+getName():String
+getSurname():String
+getID():String

<< enum >>
Post
+professor:String
+lector:String
+senior Lector:String
+assistent of Professor:String
+tutor:String
+associative Professor:String
+assistent:String

<< enum >>
AcademicDeg
+PhD:String
+MSEE:String
+master:String
+docent:String
+candidate:String
+doctor:String

Department
-nameOfDepartment:java.lang.String
-teachers:Vector<Teacher>
-faculty:UFaculty
+getNameOfDepartment():java.lang.String
+getTeachers():Vector<Teacher>

OfficeRegistration
-applications:Vector<application>
+makeSchedule():void
+getApplications():String
+getApplications():Vector<application>

Student
-GPA:double
-schedule:Schedule
-courses:Vector<Course>
-journal:java.util.Map
-transcript:Transcript
-course:int
-grant:boolean
-faculty:UFaculty
-speciality:java.lang.String
+getGPA():double
+getSchedule():Schedule
+getCourses():Vector<Course>
+getJournal():java.util.Map
+getTranscript():Transcript
+scholarship():boolean
+changeFaculty():void
+leaveUniversity():void

<< exception >>
NoMatch
<< create >>+NoMatch(s:Student,c:Course)

Teacher
-courses:Vector<Course>
-students:java.util.Map
-schedule:Schedule
-department:java.util.String
+givePoints():void
+estimate():void
+getCourses():Vector<Course>
+getStudents():java.util.Map
+getSchedule():Schedule
+putFile():void

CourseFile
-course:Course
-nameOfFile:java.lang.String
-typeOfFile:java.lang.String
+getCourse():Course
+getNameOfFile():java.lang.String
+getTypeOfFile():java.lang.String

Transcript
-student:Student
-result:java.util.Map
+showTranscript():java.lang.String
+getStudent():Student

Marks
-mark:Grade
-course:String
-student:Student
+getMark():Grade
+getCourse():String
+getStudent():Student

<< enum >>
Grade
+A:String
+A-:String
+B+:String
+B:String
+B-:String
+C+:String
+C:String
+C-:String
+D+:String
+D:String
+F:String

<< exception >>
NoCourseException
<< create >>+NoCourseException(s:Student)

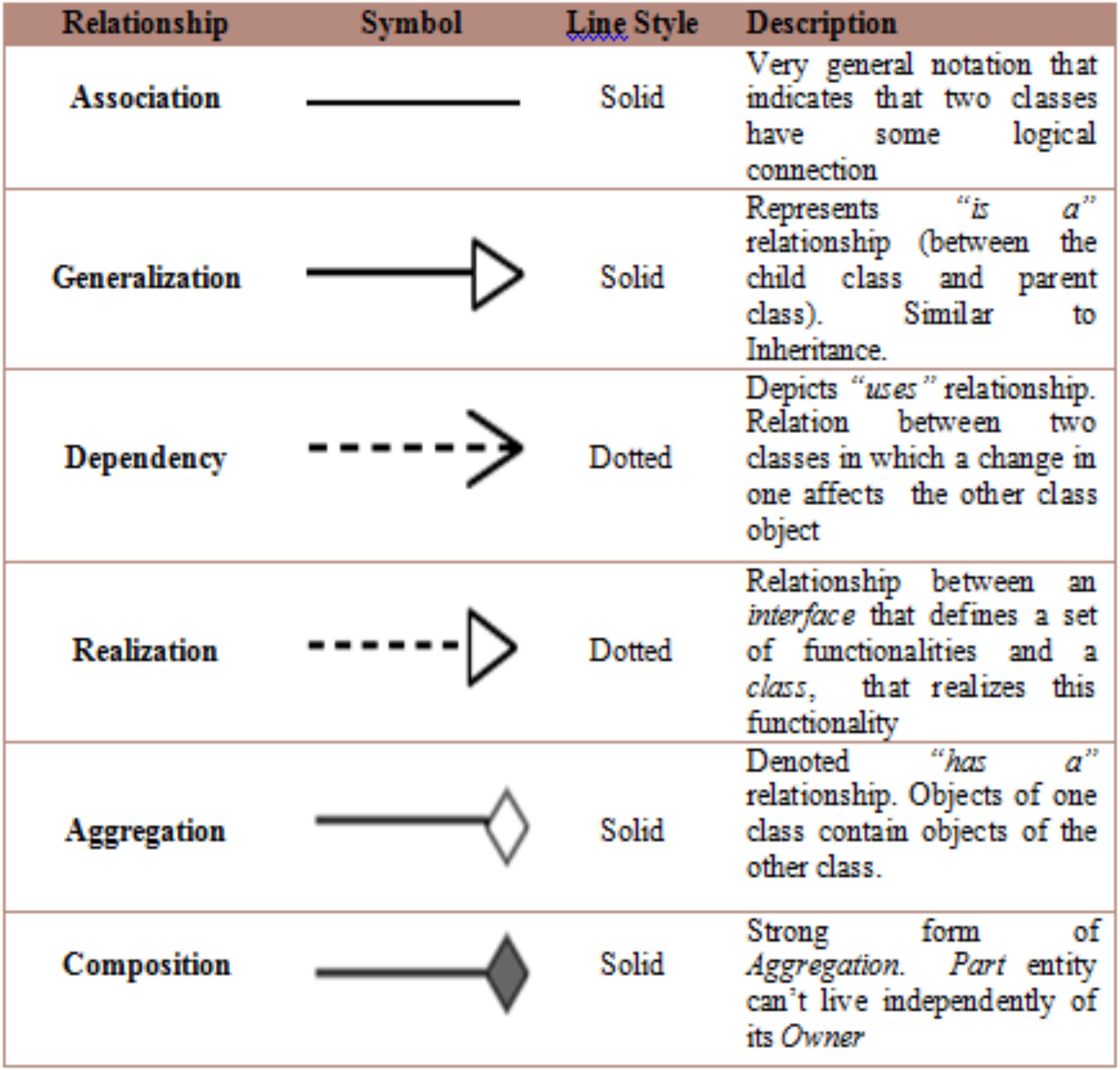
Schedule
-courses:java.util.Map
-time:double
-room:int
-person:Person
+getCourse():Course
+getPerson():Person
+showSchedule():java.lang.String

Course
-files:Vector<CourseFile>
-teacher:Teacher
-sumOfCredits:int
+getFiles():Vector<CourseFile>
+getTeachers():Teacher
+getSumOfCredits():int

PREPARED BY:

Baimuratova Elmira

Imangalieva Gulzhanat



Relationship	Symbol	Line Style	Description
Association	———	Solid	Very general notation that indicates that two classes have some logical connection
Generalization	——▷	Solid	Represents “*is a*” relationship (between the child class and parent class). Similar to Inheritance.
Dependency	- - - - ->	Dotted	Depicts “*uses*” relationship. Relation between two classes in which a change in one affects the other class object
Realization	- - - - -▷	Dotted	Relationship between an *interface* that defines a set of functionalities and a *class*, that realizes this functionality
Aggregation	——◇	Solid	Denoted “*has a*” relationship. Objects of one class contain objects of the other class.
Composition	——◆	Solid	Strong form of *Aggregation*. *Part* entity can’t live independently of its *Owner*

Tips

- Don’t try to use all the various notations.
- Don’t draw models for everything, concentrate on the key areas.
- You can also create models that describe some particular part of the system, not all system

Review: steps followed

- Draw class symbol in the editor and name it
- List the class attributes
- List the class operations/methods
- Make the links and associations
- Give notations