Hoang Huu Hanh, Hue University

hanh-at-hueuni.edu.vn

Modeling Class Architecture

with UML Class Diagrams

based on the presentation at http://www.technion.ac.il/~erant

Outline

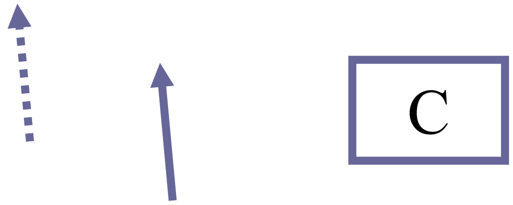
- Introduction
- Classes, attributes and operations
- Relations
- Generalization
- Guidelines for effective class modeling

System Development Process

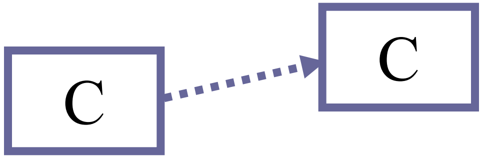
Phase	Actions	Outcome
Initiation	Raising a business need	Business documents
Requirements	Interviewing stakeholders, exploring the system environment	Organized documentation
Analysis & Specification	Analyze the engineering aspect of the system, building system concepts	Logical System Model
Design	Define architecture, components, data types, algorithms	Implementation Model
Implementation	Program, build, unit-testing, integrate, documentation	Testable system
Testing & Integration	Integrate all components, verification, validation, installation, guidance	Testing results, Working sys
Maintenance	Bug fixes, modifications, adaptation	System versions

Elements of Modelling Language

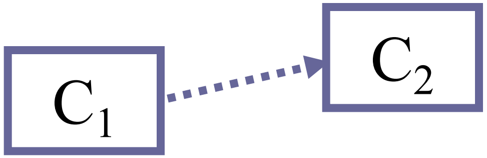
- Symbols: Standard set of symbols



- Syntax: Acceptable ways of combining symbols



- Semantics: Meaning given to language expressions



C_1 sends a message to C_2

Advanced Properties

- Expressiveness: What the language can say

OK: C_1 sends messages to C_2
Not OK: C_1 sends messages to C_2, after all messages of C_2 were recieved

- Methodology: Procedures to be followed

1. Model all classes
2. Model all relations
3. Model all inheritance

- Guidelines: Suggestions on how to build effective models

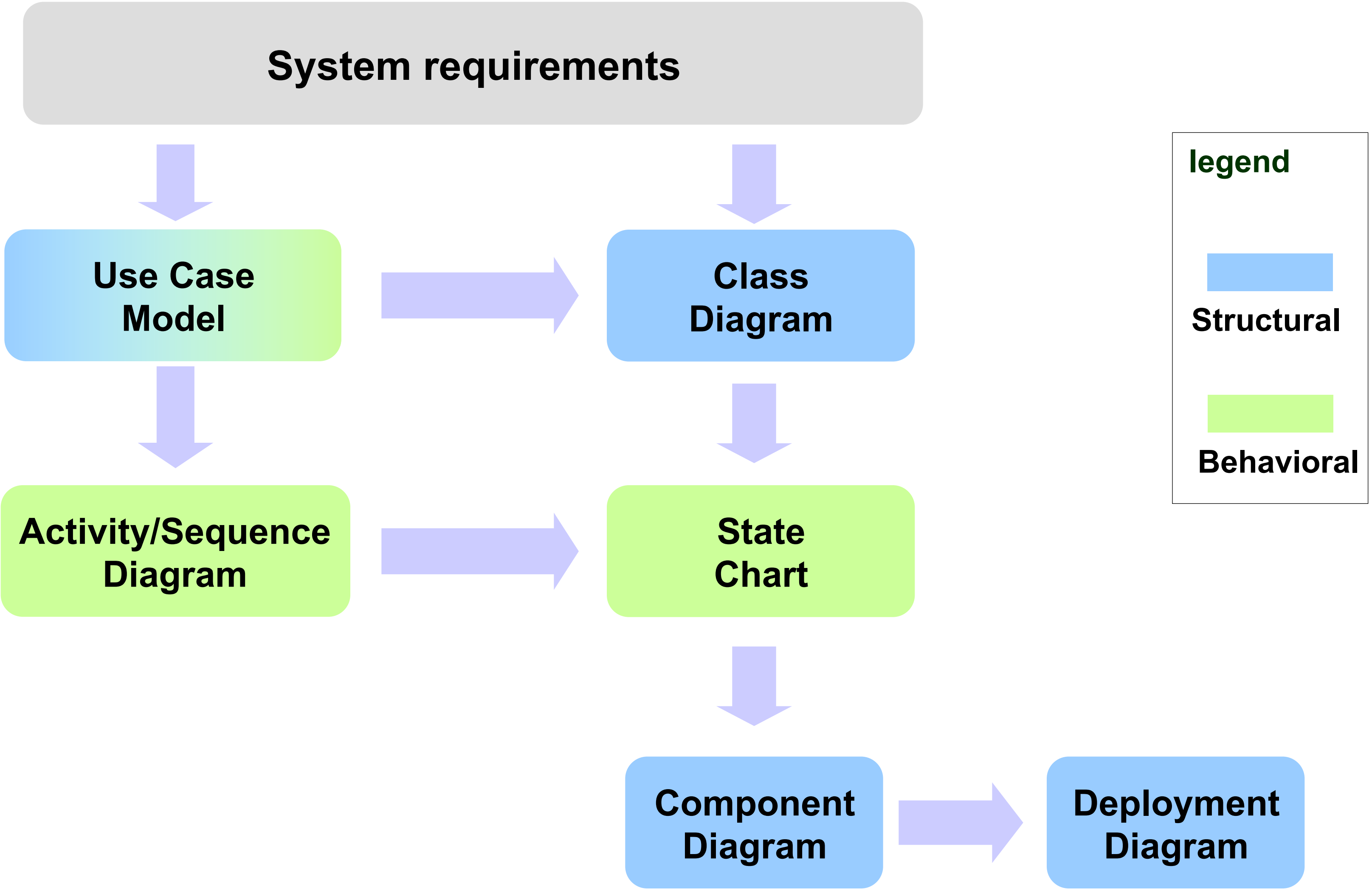
Try to model classes with a balanced number of associations

Modeling Approaches

Modeling approaches differ from each other according to their view of the world

Object-Oriented	**Process-Oriented**	**State-Oriented**
Focused on objects, which are concrete elements, combining information and actions	Focused on processes, which are patterns of transformation (of something). Processes can be concrete or abstract)	Focused on the different states – values and status of the system, and how and why these states change.

Design Process



System requirements

Use Case Model

Class Diagram

Activity/Sequence Diagram

State Chart

Component Diagram

Deployment Diagram

legend

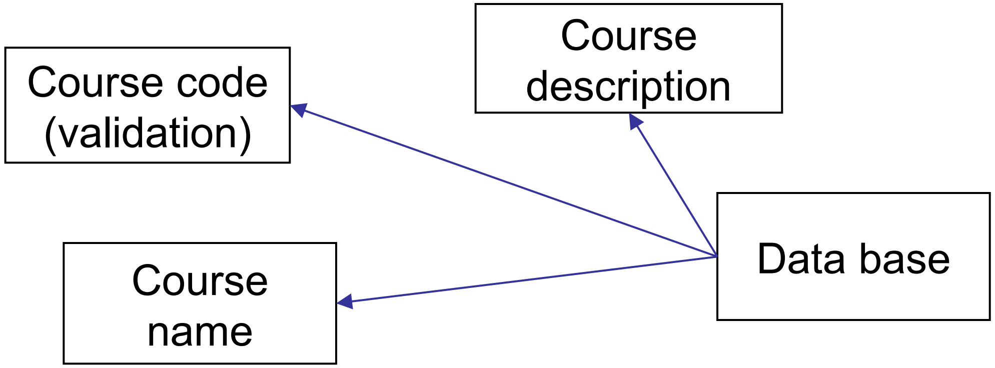
Structural

Behavioral

From Requirements to Structure

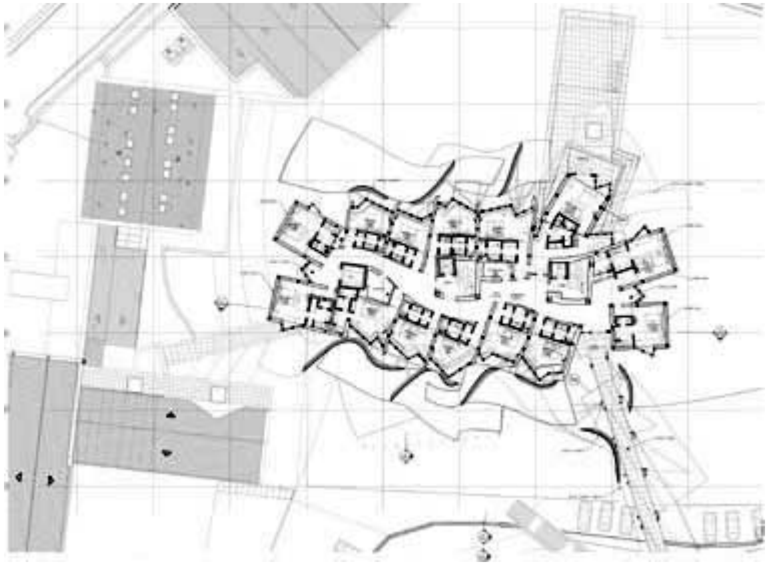
1. **Administrator enters course name, code and description**
2. **System validates course code**
3. **System adds the course to the data base and shows a confirmation message**

Requirements Document



Structure (what's the constant things of the system)

What is Structural Modeling?



A structural design defines the artifact unchanging characteristics, which do not change over time.

Structural Modeling in Information Systems

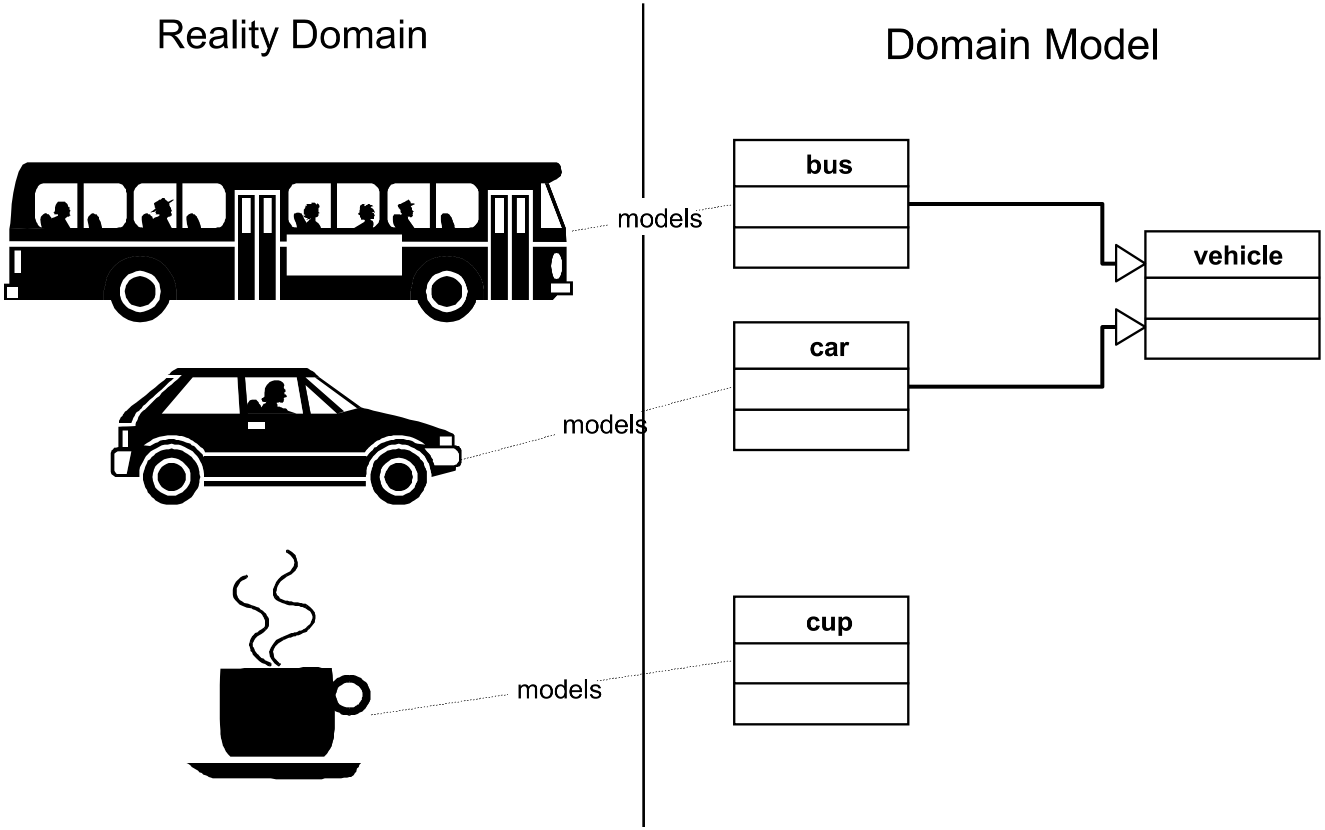
- Static structure of the model
  - the entities that exist (e.g., classes, interfaces, components, nodes)
  - relationship between entities
  - internal structure
- Do not show
  - temporal information
  - Behavior
  - Runtime constraints

Outline

- Introduction
- Classes, attributes and operations
- Relations
- Generalization
- Guidelines for effective class modeling

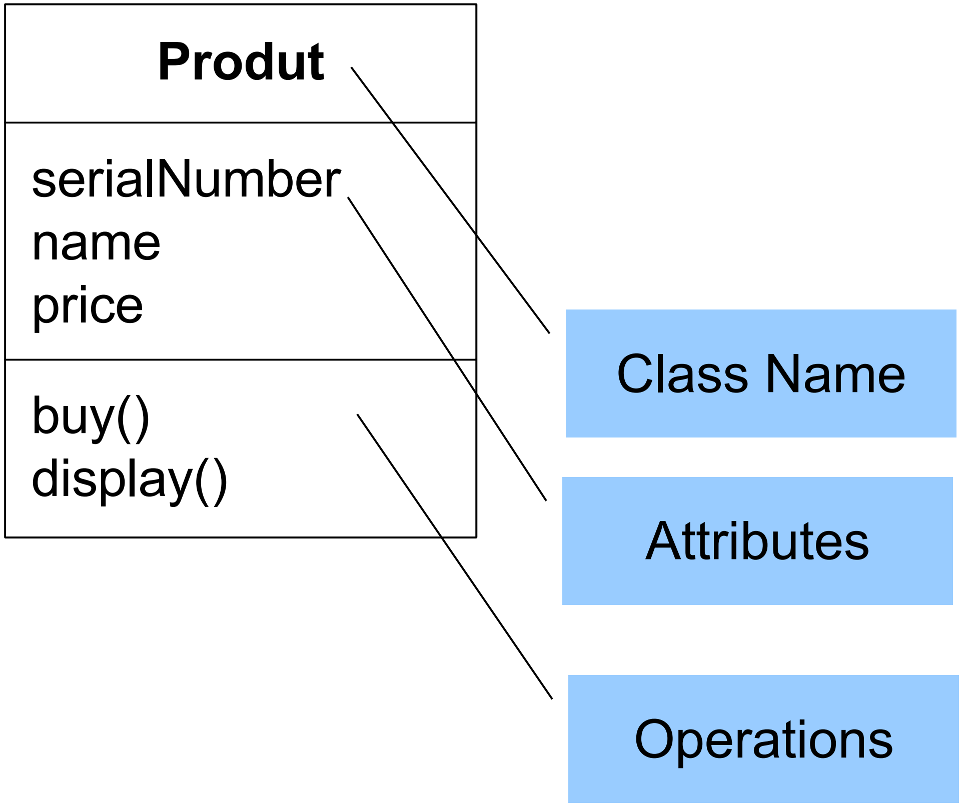
Object-Oriented Approach

- Objects are abstractions of real-world or system entities



Classes

- A class is a template for actual, in-memory, instances



Produt

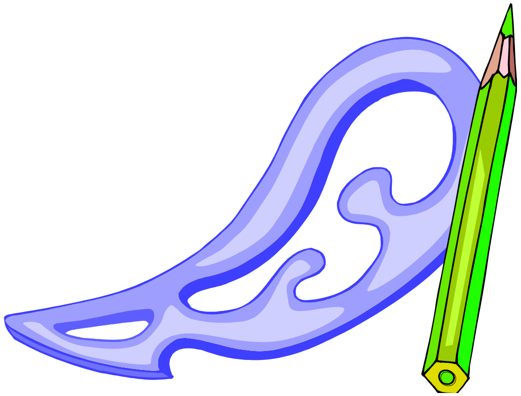
serialNumber
name
price

buy()
display()

Class Name

Attributes

Operations



Domain Model

Attributes - Signature

[visibility] **name** [[multiplicity]] [: type] [=initial value] [{property}]

- visibility: the access rights to the attribute ?
- multiplicity: how many instances of the attribute are they:
  - middleName [0..1] : String, phoneNumber [1..*]
- Type: the type of the attribute (integer, String, Person, Course)
- initial value: a default value of the attribute
  - salary : Real = 10000, position : Point = (0,0)
- property: predefined properties of the attribute
  - Changeable, readOnly, addOnly, frozen (C++: const, Java: final)

Attributes - Examples

```
+ isLightOn : boolean = false
- numOfPeople : int
mySport
+ passengers : Customer[0..10]
- id : long {readOnly}
```

Operations - Signature

[visibility] **name** [(parameter-list)] [: return-type] [{property}]

- An operation can have zero or more parameters, each has the syntax:
  - [direction] name : type [=default-value]
  - Direction can be: in (input paremter - can't be modified), out (output parameter - may be modified), inout (both, may be modified)
- Property:
  - {leaf} – concrete operation
  - {abstract} – cannot be called directly
  - {isQuery} – operation leaves the state of the operation unchanged
  - …

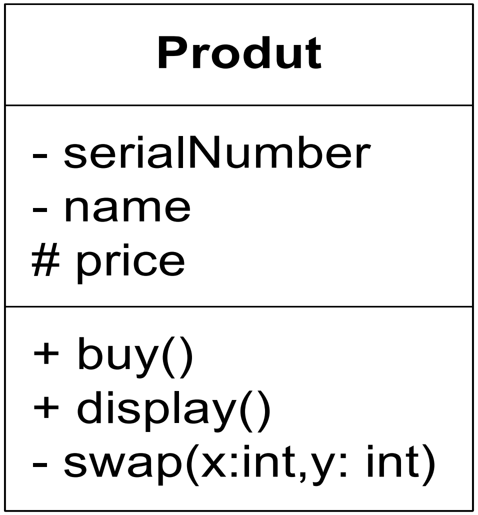
Operations - Examples

```
+ isLightOn() : boolean
+ addColor(newColor : Color)
+ addColor(newColor : Color) : void
# convertToPoint(x : int, y : int) : Point
- changeItem([in] key : string, [out] newItem :
      Item) : int
```

What’s the difference?

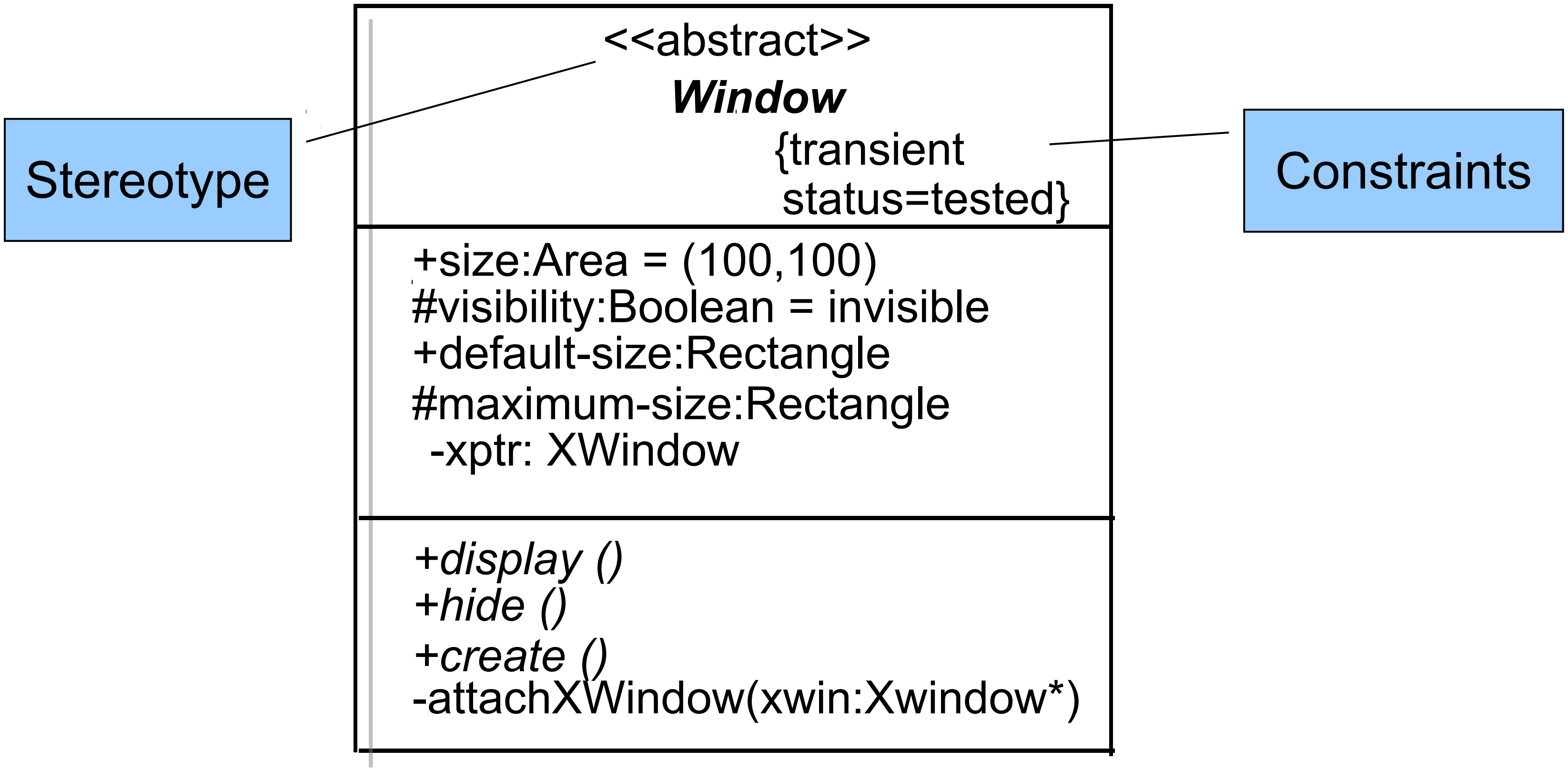
Visibility

- public (**+**) – external objects can access the member
- private (**-**) – only internal methods can access the member
- protected (**#**) – only internal methods, or methods of specialized objects can access the member



We will try to keep the visibility as minimal as possible

Full Blown Class



Stereotype

<<abstract>>

Window

{transient
status=tested}

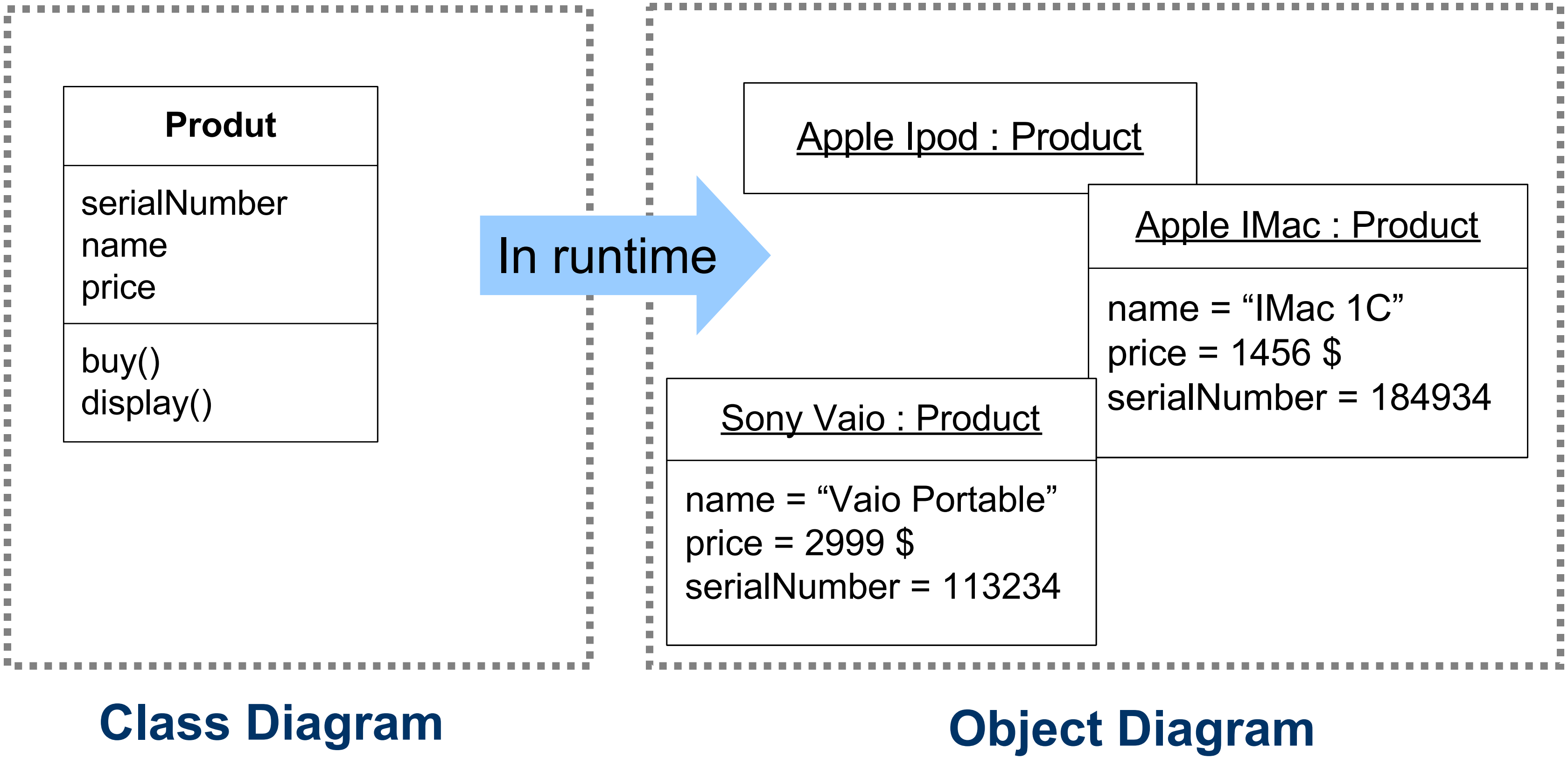
Constraints

+size:Area = (100,100)
#visibility:Boolean = invisible
+default-size:Rectangle
#maximum-size:Rectangle
-xptr: XWindow

+*display ()*
+*hide ()*
+*create ()*
-attachXWindow(xwin:Xwindow*)

Object Diagram

- In an Object Diagram, class **instances** can be modeled

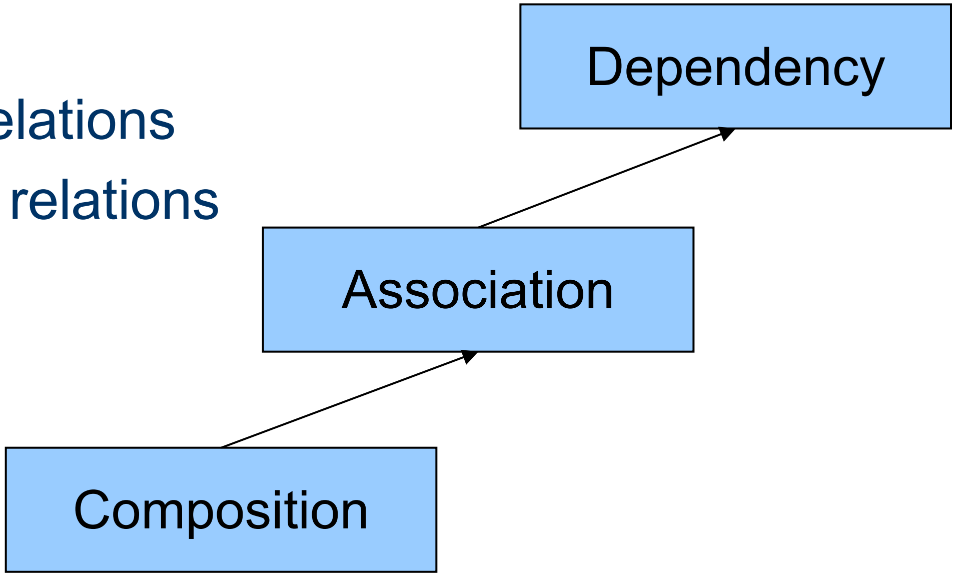


Outline

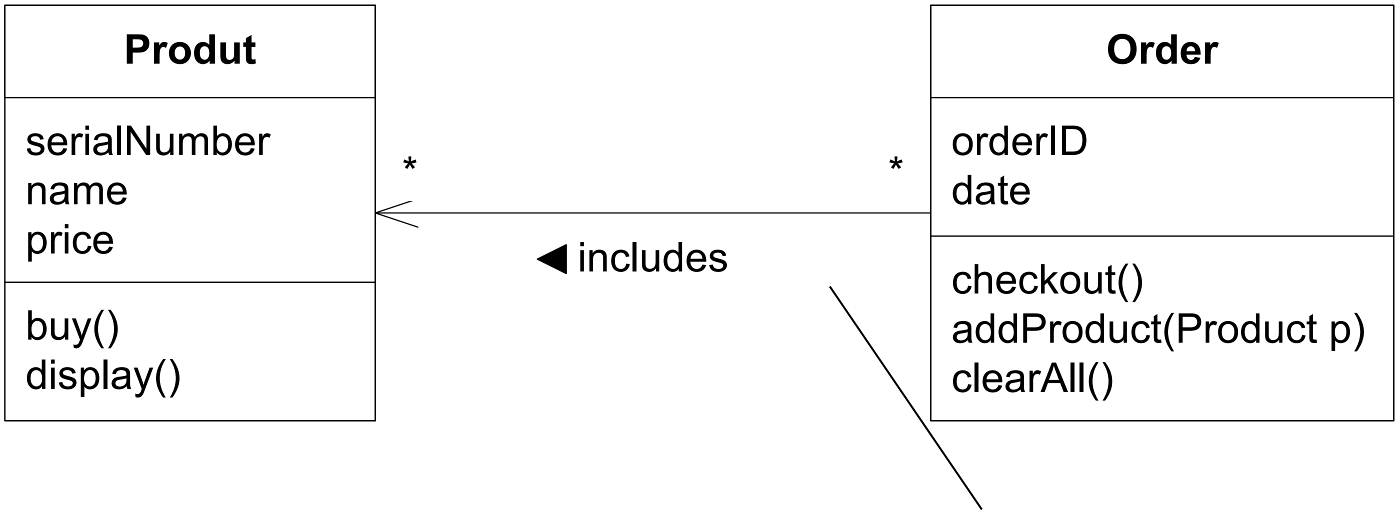
- Introduction
- Classes, attributes and operations
- Relations
- Generalization
- Guidelines for effective class modeling

Relations

- A relation is a template for a connection between two instances.
- Relations are organized in a Hierarchy:
  - Dependency: dynamic relations
  - Associations: consistent relations
  - Composition: whole-part relations



Associations



Produt

serialNumber
name
price

buy()
display()

*

*

◀ includes

Order

orderID
date

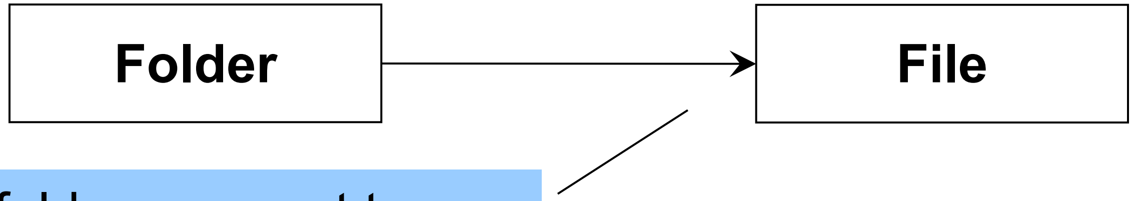
checkout()
addProduct(Product p)
clearAll()

Multiplicity

Indicates cardinality
•1:1 – default
•3 – exactly 3 object
•* (or n) - unbounded
•1..* - 1 to eternity
•3..9 – 3 to 9

- Objects on both sides of the association can find each other
- The relation is consistent in time (unless removed)

Navigation

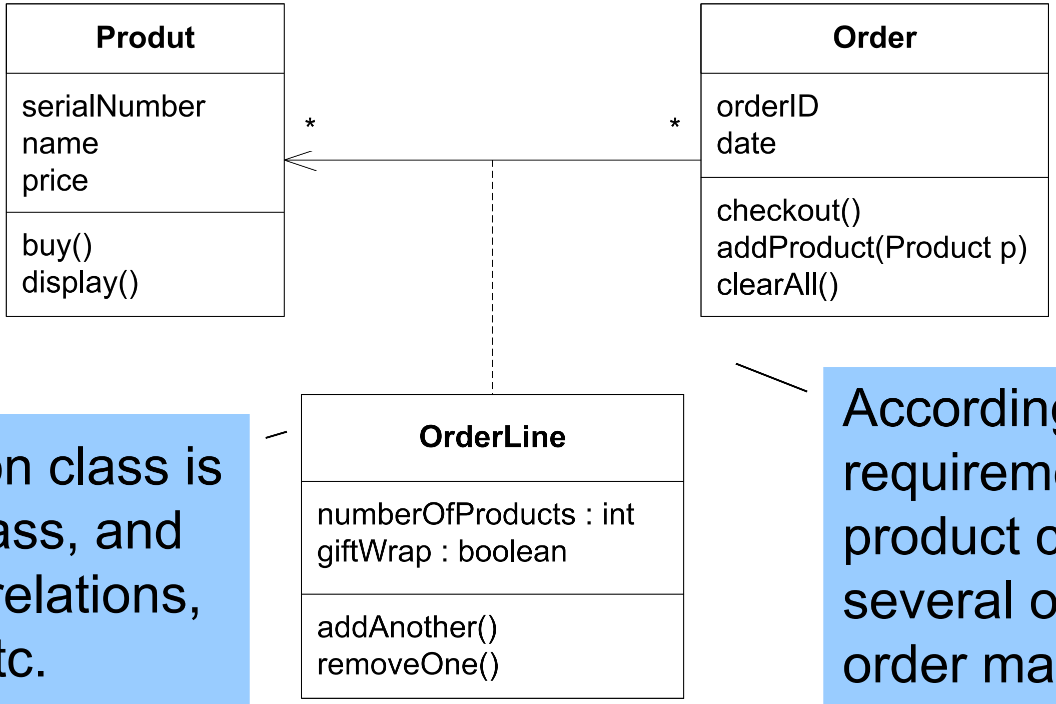


Given a folder, we want to know the files of each folder. However, we do not have a requirement for knowing the folder of each file.

- If an association is directed, messages can pass only on that direction
- If the association does not have directions, then it’s a bidirectional association
- By default, all relations should be **directed,** unless the requirements dictate a bidirectional relation

Association Classes

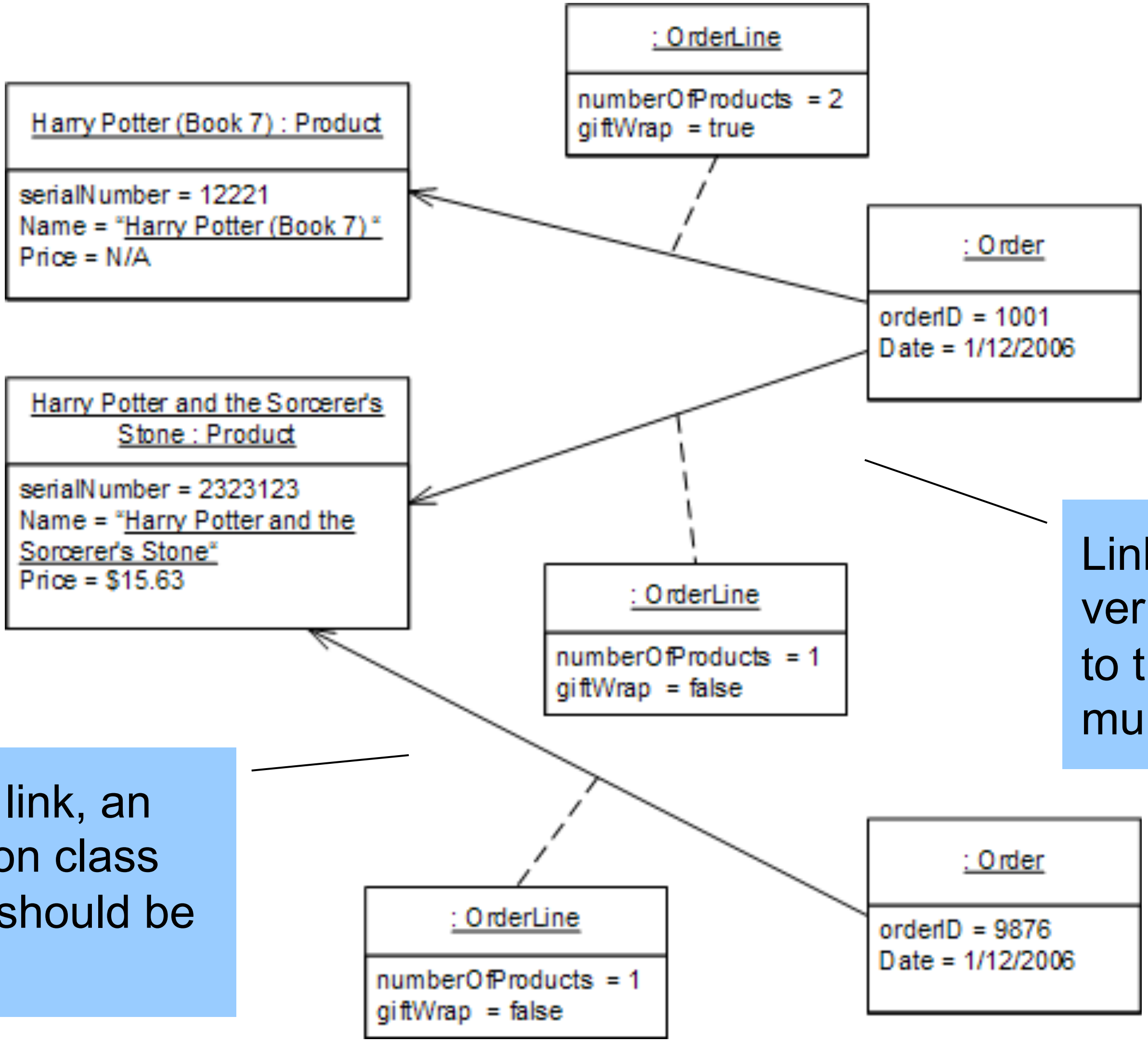
Denoted as a class attached to the association, and specify properties of the association



An association class is a “normal” class, and may include relations, inheritance etc.

According to the requirements, each product can appear is several orders, and each order may include several products

Association Class - Objects

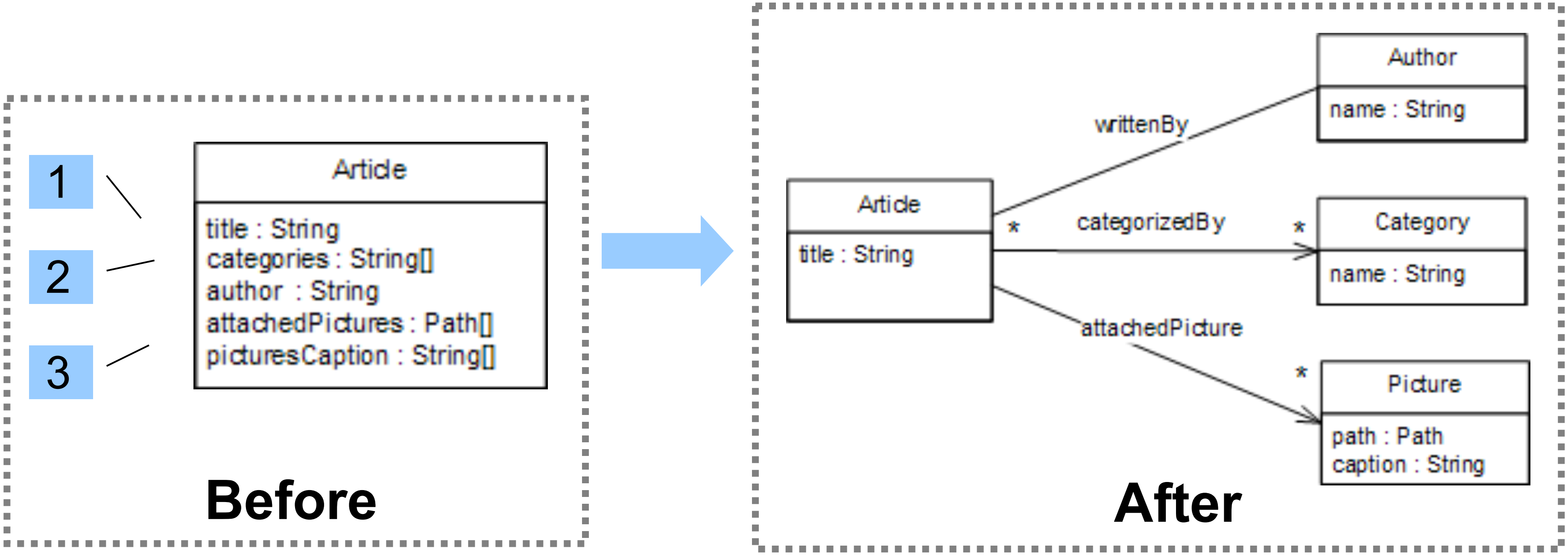


Links should be verified, according to the association multiplicity

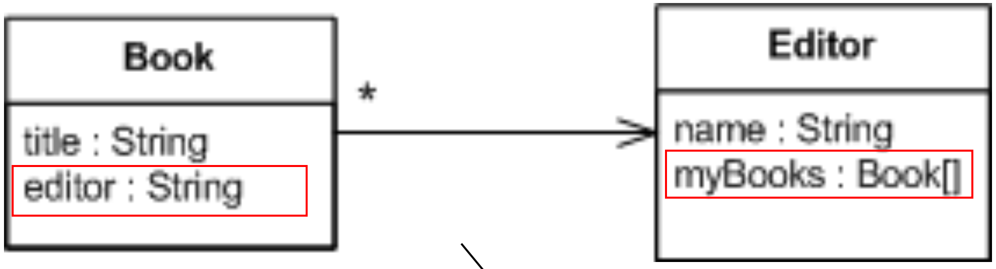
For each link, an association class instance should be declared

Class Normalization

- Classes should be normalized, if:
  1. Attributes are selected from large or infinite sets
  2. Relations with attributes are in n:n form
  3. Groups of attributes are related



Relations & Attributes

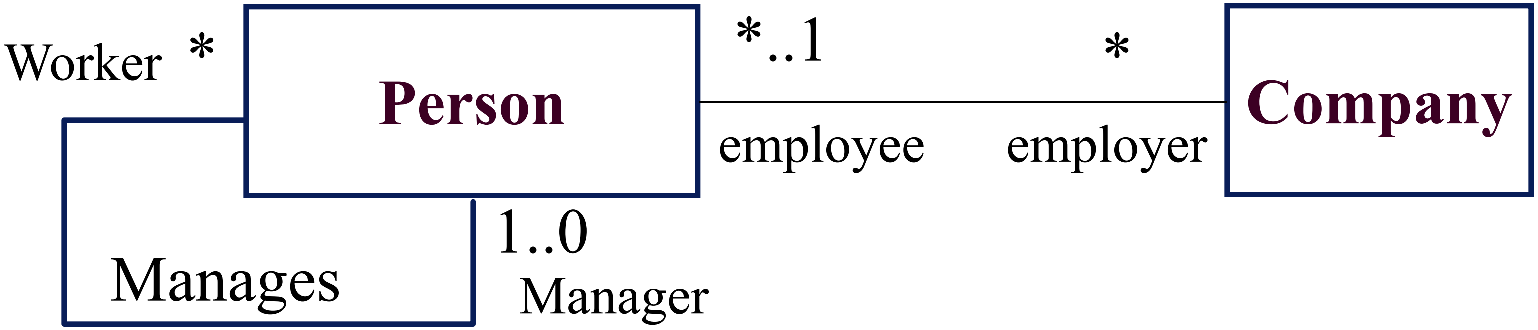


What is the problem?

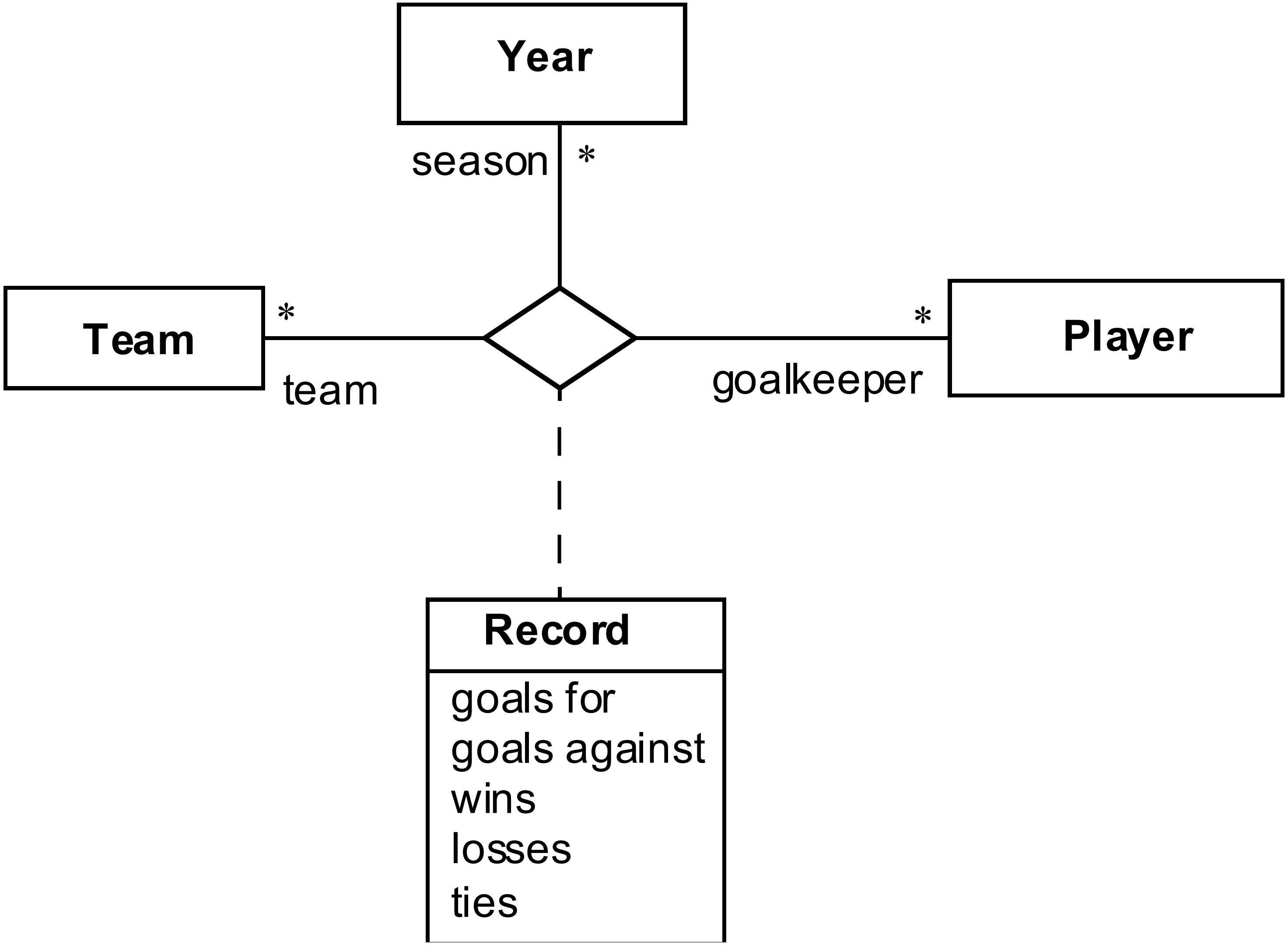
- Relations are denoted with associations, not attributes.
- Implementation (pointers, arrays, vectors, ids etc) is left to the detailed design phase.

Role Names

- Names may be added at each end of the association
- Provide better understanding of the association meaning
- Especially helpful in self-associated classes

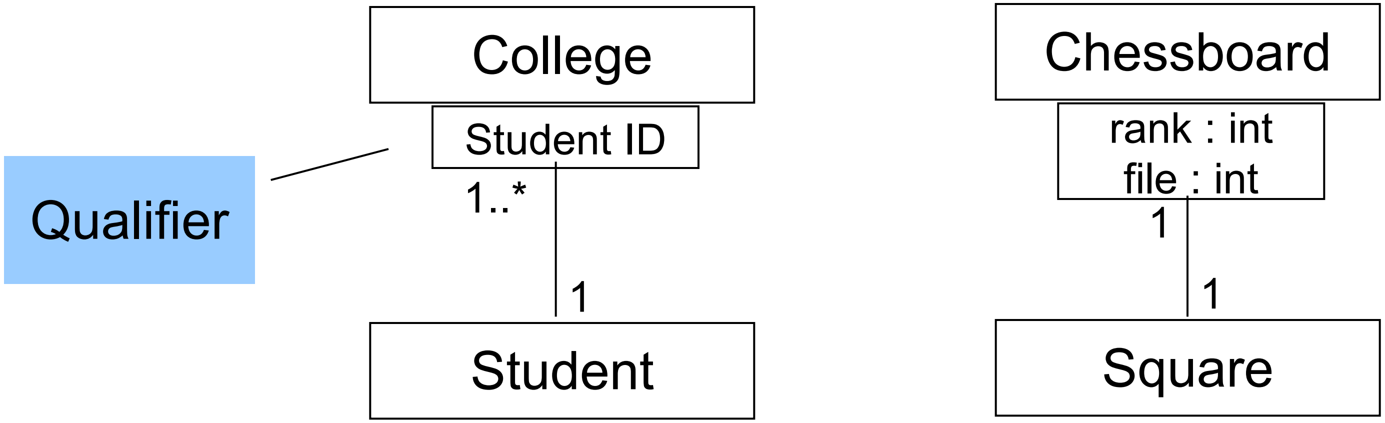


Ternary Associations



Qualifiers

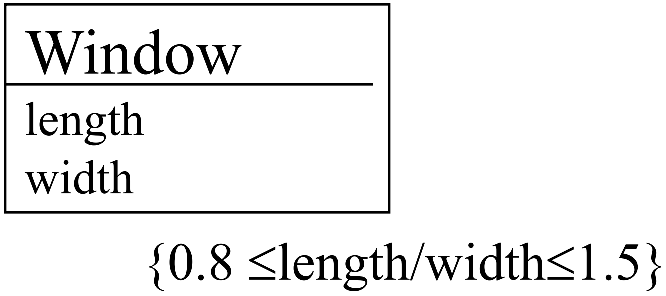
- A qualifier is an attribute or list of attributes whose values serve to partition the set of objects associated with an object across an association



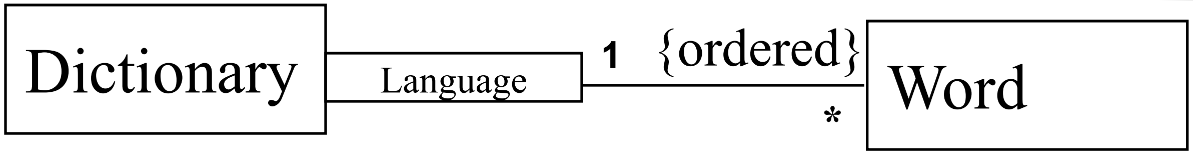
- The qualifier limits the multiplicity of the target object according to the qualifier attribute. Thus, even though a Bank has **many** persons, it has one or zero person with a particular account #

Constraints

- Constrains are simple properties of associations, classes and many other things in UML
- Specify limitations that implementers need to satisfy

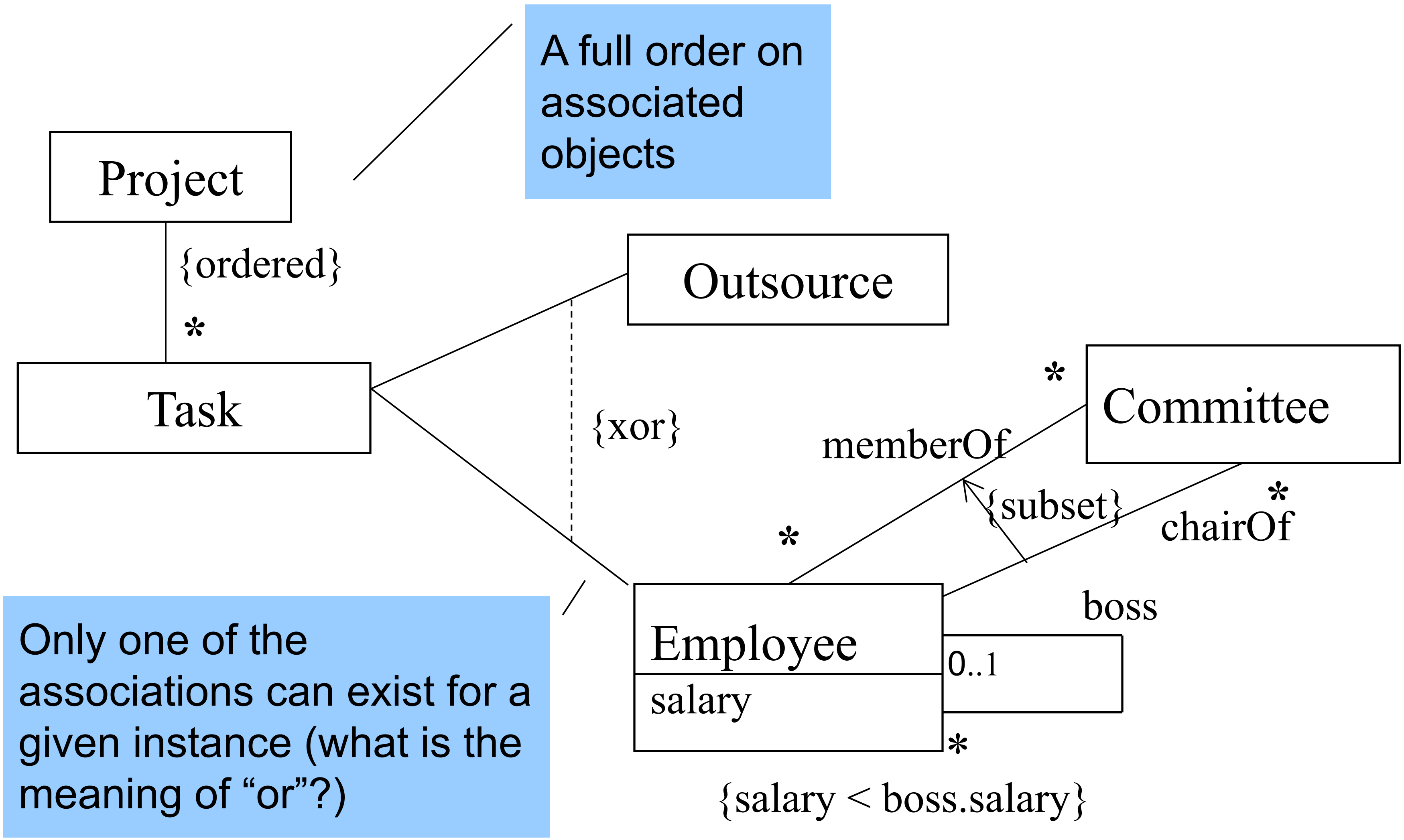


Property Constraints



Denotes explicit order of instance

Constraints - cont’d



A full order on associated objects

Only one of the associations can exist for a given instance (what is the meaning of “or”?)

Constraints

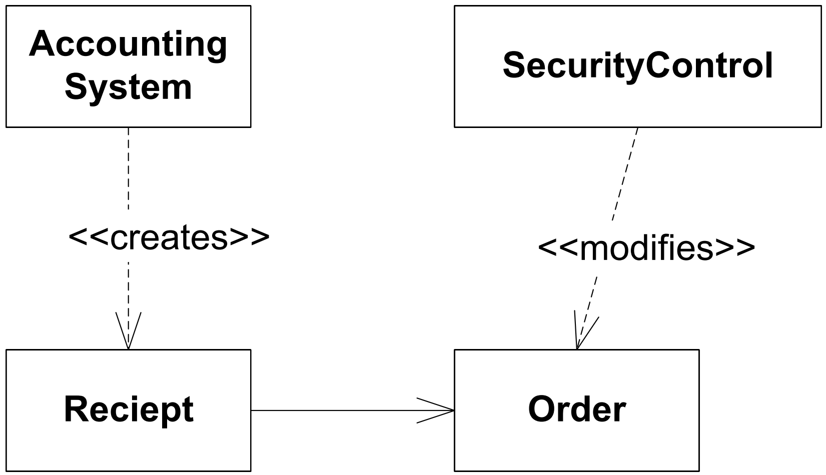
- Constraints can be applied to almost every element in UML diagrams, using:
  - natural language
  - mathematical notation
  - OCL (Object Constraint Language)
- Expressing:
  - Invariants: interest > 3%
  - Preconditions: before loan() takes place, salary > 5,000$
  - Postconditions: after loan() takes place, dayCollect = 1 or 10

See http://www.klasse.nl/ocl/index.html

Dependency

- Notated by a dotted line - - - - - - - - - - - - - - - →
- The most general relation between classes
- Indicates that an object affects another object

AccountingSystem creates a **Receipt** object

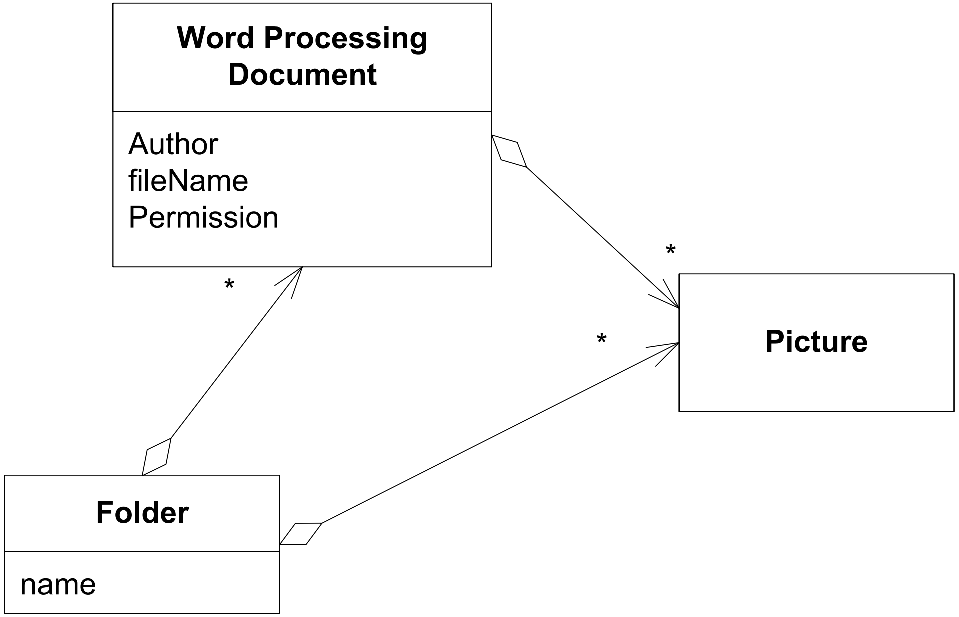


Dependency – cont'd

- Dependencies are the most abstract type of relations.
- Properties:
  - Dependencies are always directed (If a given class depends on another, it does not mean the other way around).
  - Dependencies do not have cardinality.
- If instances of two classes send messages to each other, but are not tied to each other, then dependency is appropriated.
- Types:
  - «call»
  - «create»

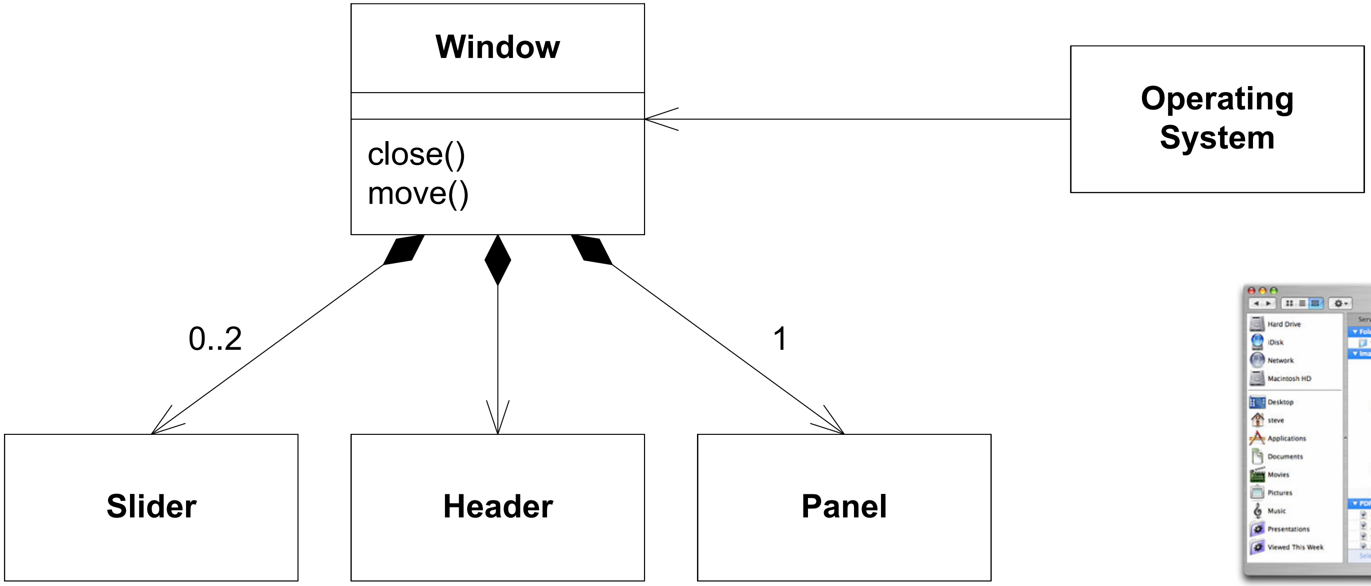
Aggregation

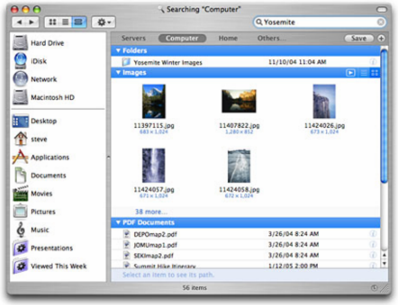
- “Whole-part” relationship between classes
- Assemble a class from other classes
  - Combined with “many” - assemble a class from a couple of instances of that class



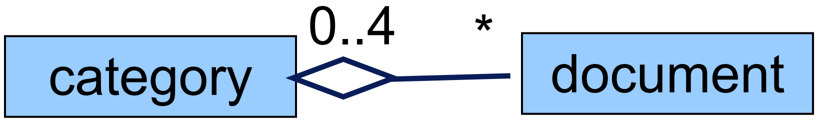
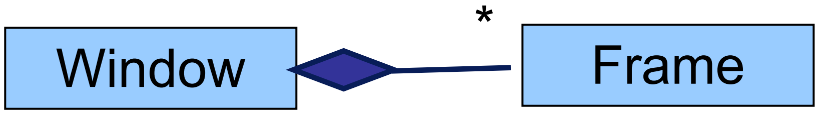
Composition

- Composition is a stronger form of aggregation
- Contained objects that live and die with the container
- Container creates and destroys the contained objects





Composition vs. Aggregation

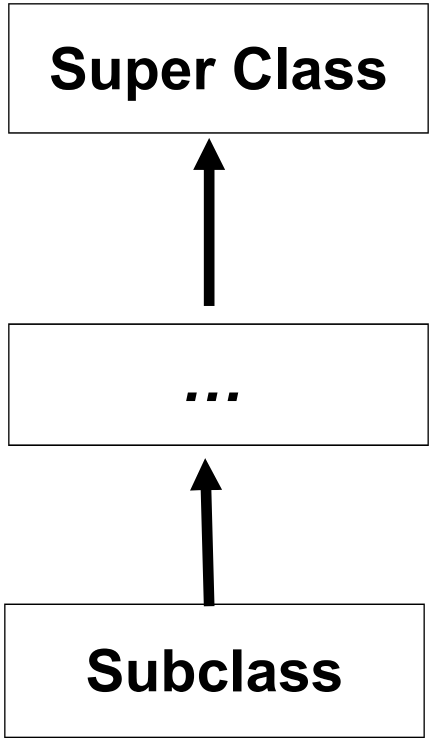
Aggregation	**Composition**
Part can be shared by several wholes  	Part is always a part of a single whole 
Parts can live independently (i.e., whole cardinality can be 0..*)	Parts exist only as part of the whole. When the wall is destroyed, they are destroyed
Whole is not solely responsible for the object	Whole is responsible and should create/destroy the objects

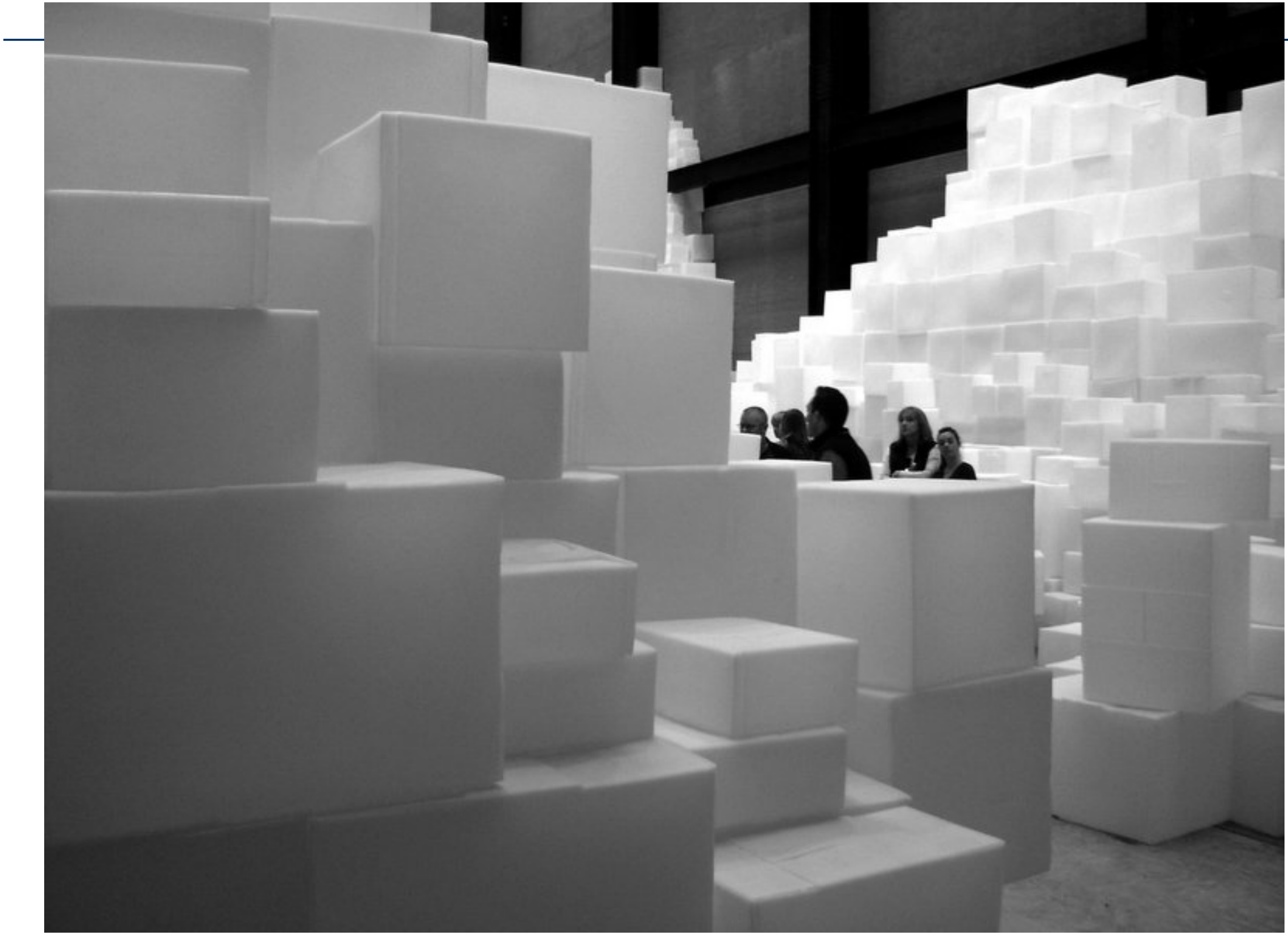
Outline

- Introduction
- Classes, attributes and operations
- Relations
- Generalization
- Guidelines for effective class modeling

Generalization – Definitions

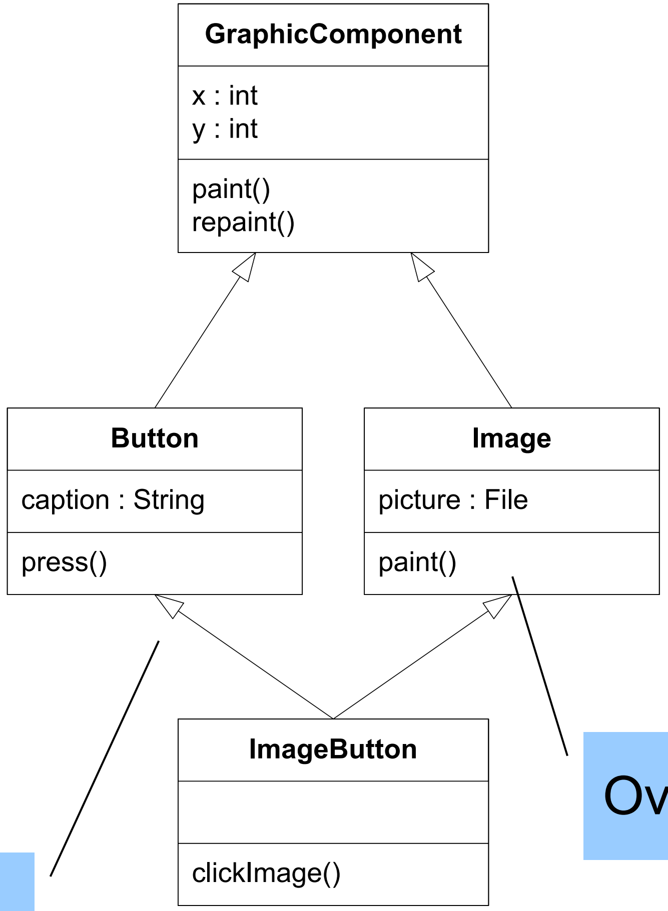
- Super Class (Base class)
  - Provides common functionality and data members
- Subclass (Derived class)
  - Inherits public and protected members from the *super class*
  - Can extend or change behavior of super class by *overriding* methods
- Overriding
  - Subclass may override the behavior of its super class





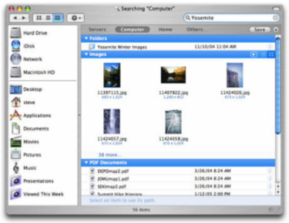
Generalization – advantages

- Modularity:
  - Eliminate the details
  - Find common characteristics among classes
  - Define hierarchies
- Reuse:
  - Allow state and behavior to be specialized



Multiple Inheritance

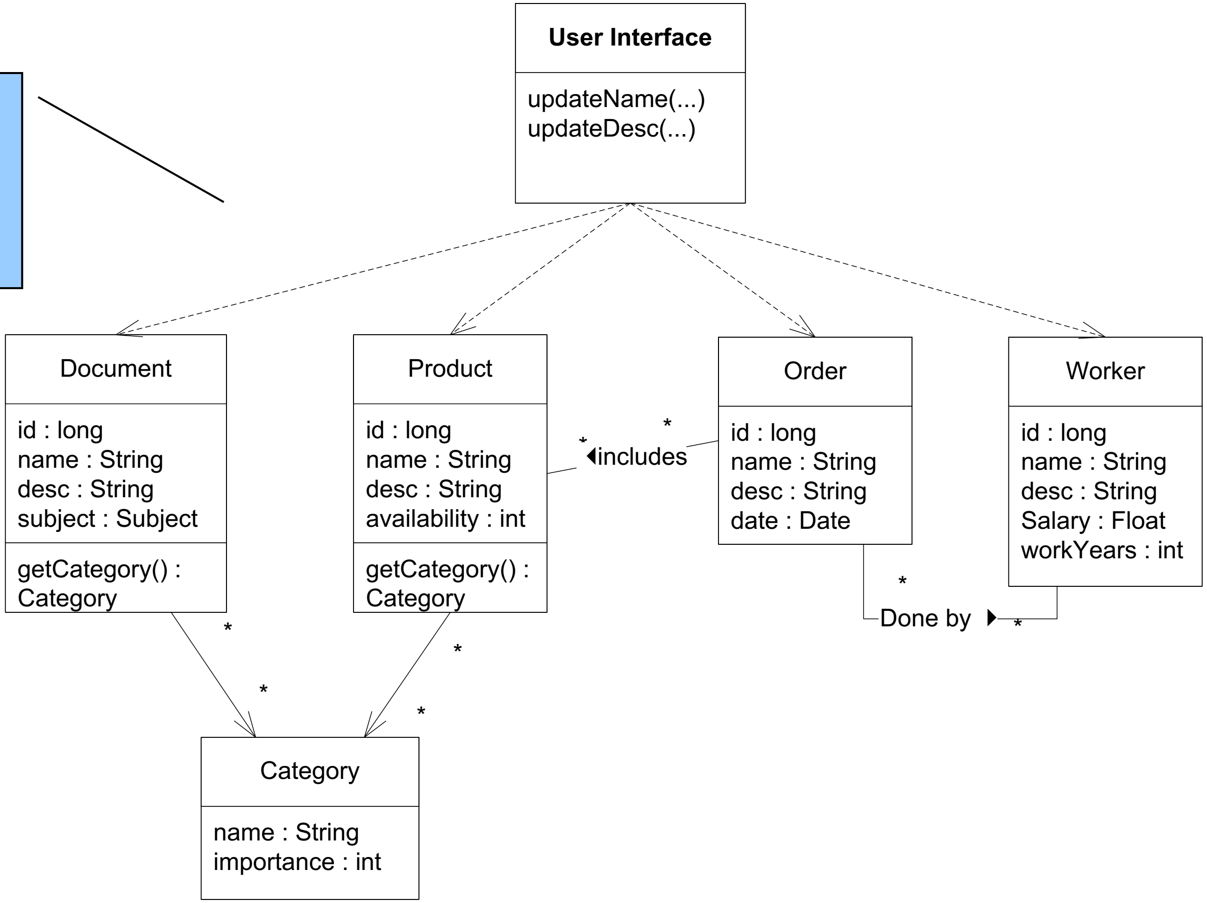
Overriding



Generalization Guidelines

- Look carefully for similar properties between objects, sometimes they are not so obvious

What's the problem here?



User Interface

updateName(...)
updateDesc(...)

Document

id : long
name : String
desc : String
subject : Subject

getCategory() : Category

Product

id : long
name : String
desc : String
availability : int

getCategory() : Category

Order

id : long
name : String
desc : String
date : Date

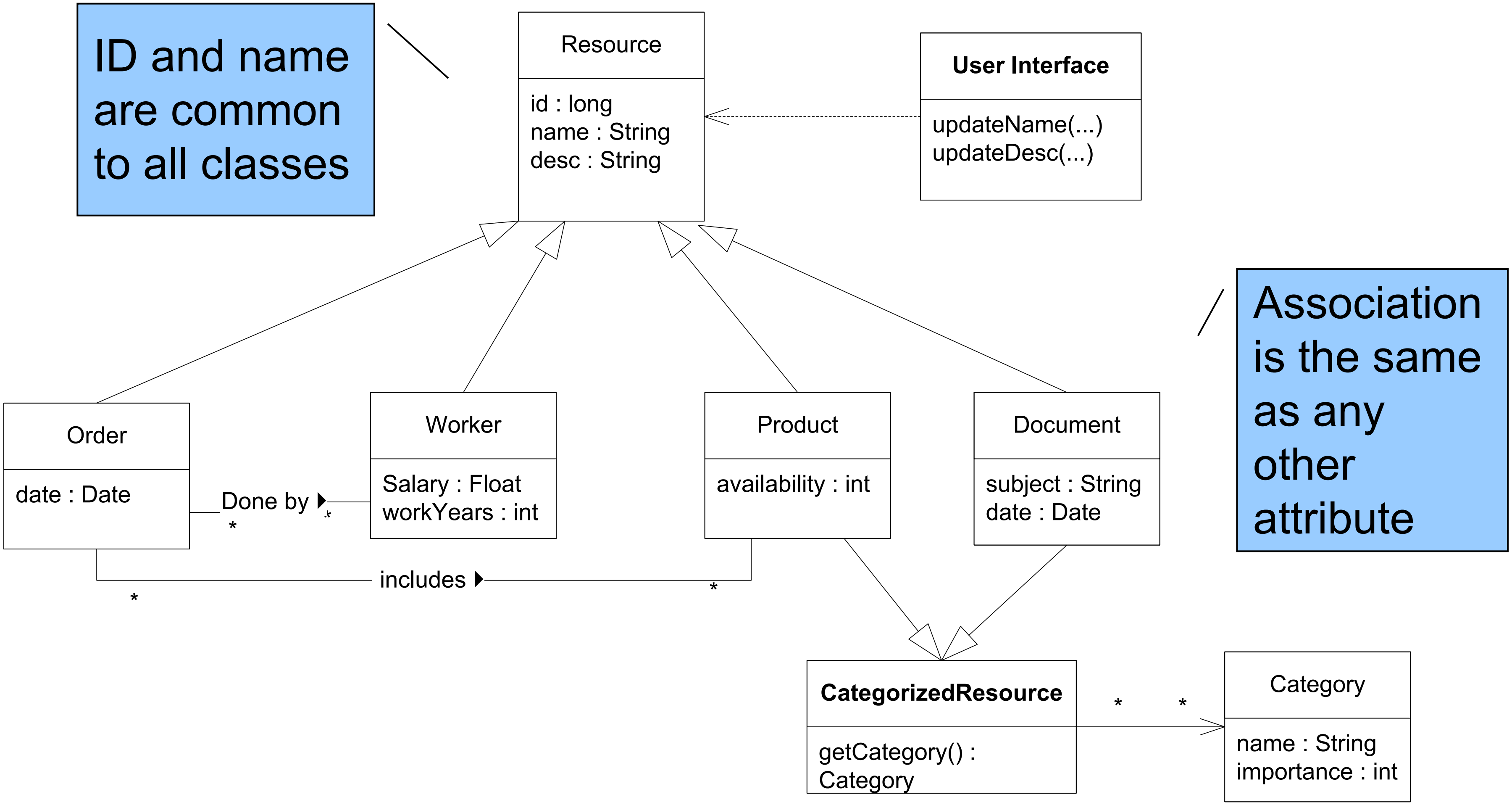
Worker

id : long
name : String
desc : String
Salary : Float
workYears : int

Category

name : String
importance : int

Generalization – cont’d



ID and name are common to all classes

Association is the same as any other attribute

Resource

id : long
name : String
desc : String

User Interface

updateName(...)
updateDesc(...)

Order

date : Date

Worker

Salary : Float
workYears : int

Product

availability : int

Document

subject : String
date : Date

CategorizedResource

getCategory() : Category

Category

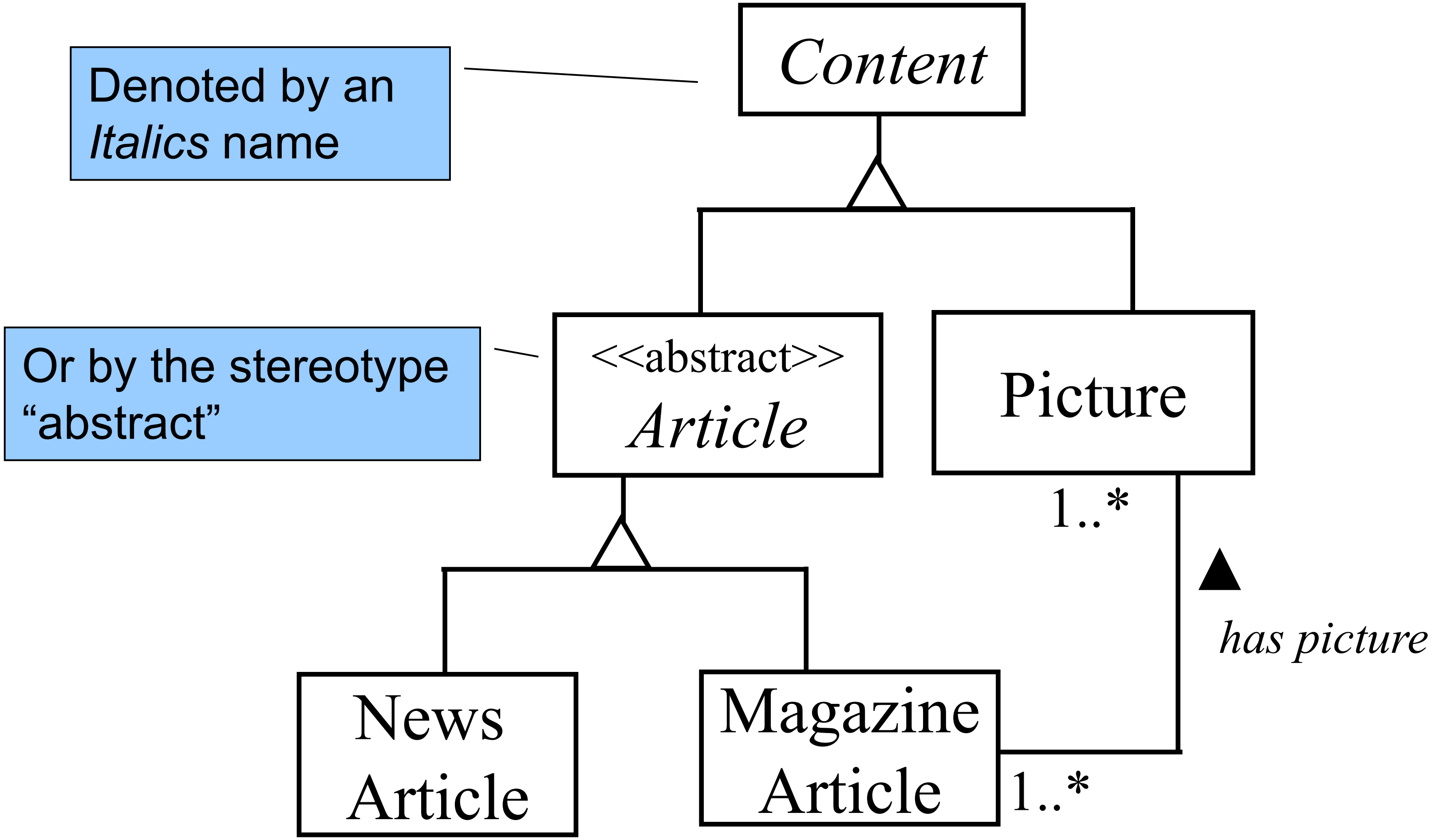
name : String
importance : int

Done by ▸

includes ▸

Abstract Class

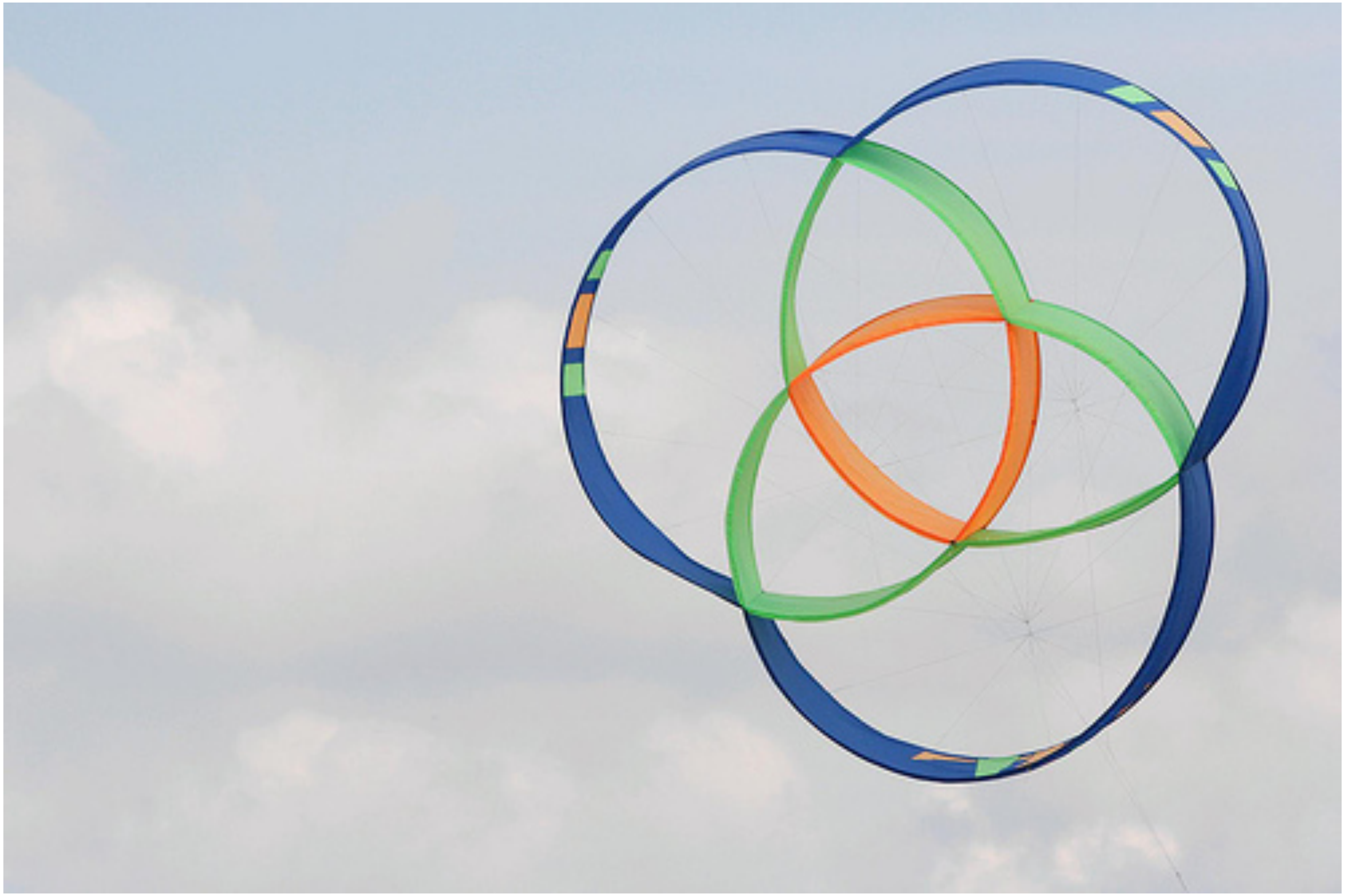
- A class that has no direct instances



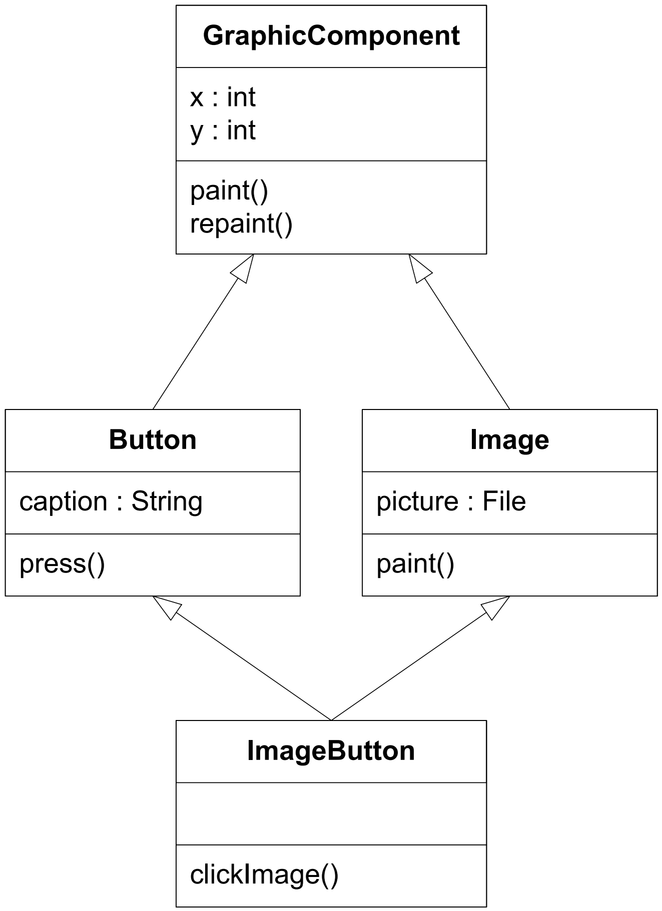
Denoted by an *Italics* name

Or by the stereotype "abstract"

Models and Sets



Generalization and Sets



GraphicComponent

x : int
y : int

paint()
repaint()

Button

caption : String

press()

Image

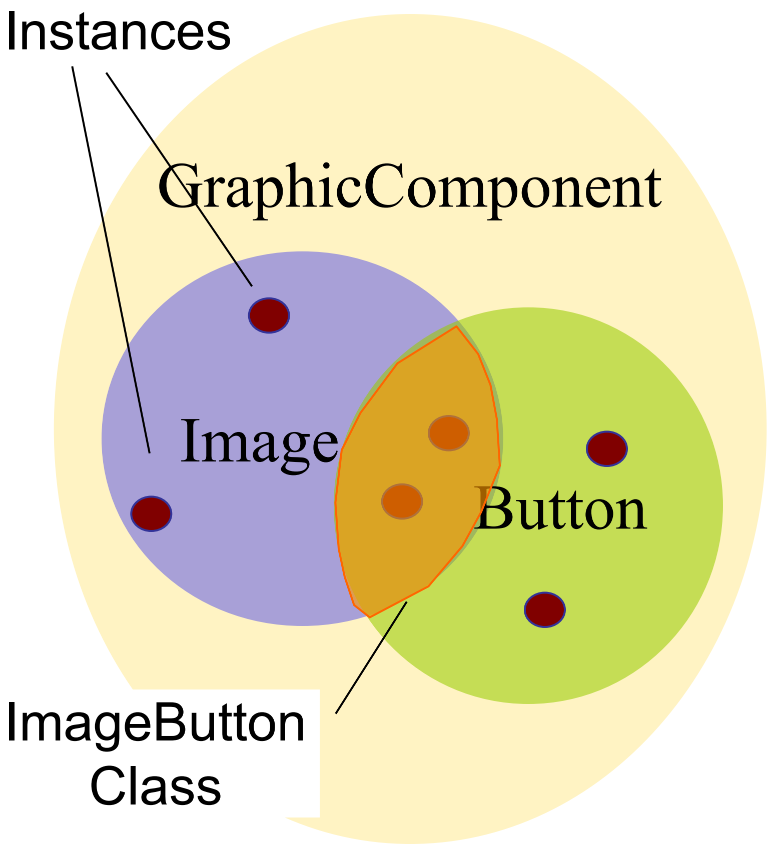
picture : File

paint()

ImageButton

clickImage()

Class Representation

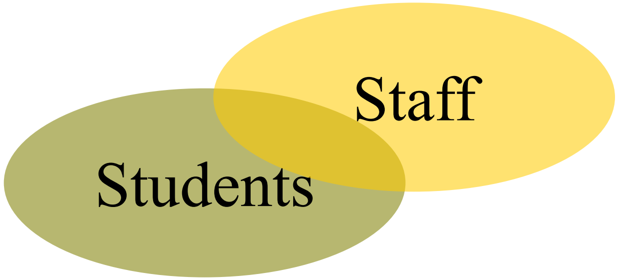


Set Representation

What Relations are Missing?

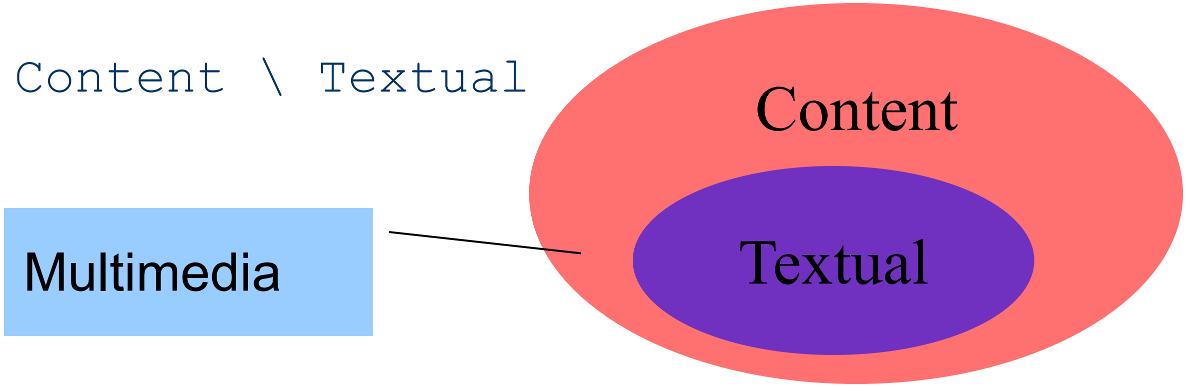
- Union
  - We cannot define a class such as:

```
allPeopleInTheTechnion =  students ∪ Staff
```



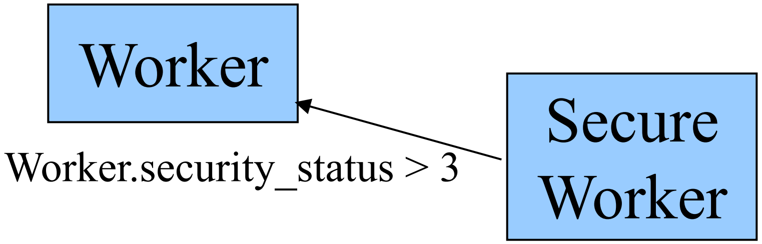
- Complementary
  - We cannot create classes which take some of the super-class properties but omit some of them:

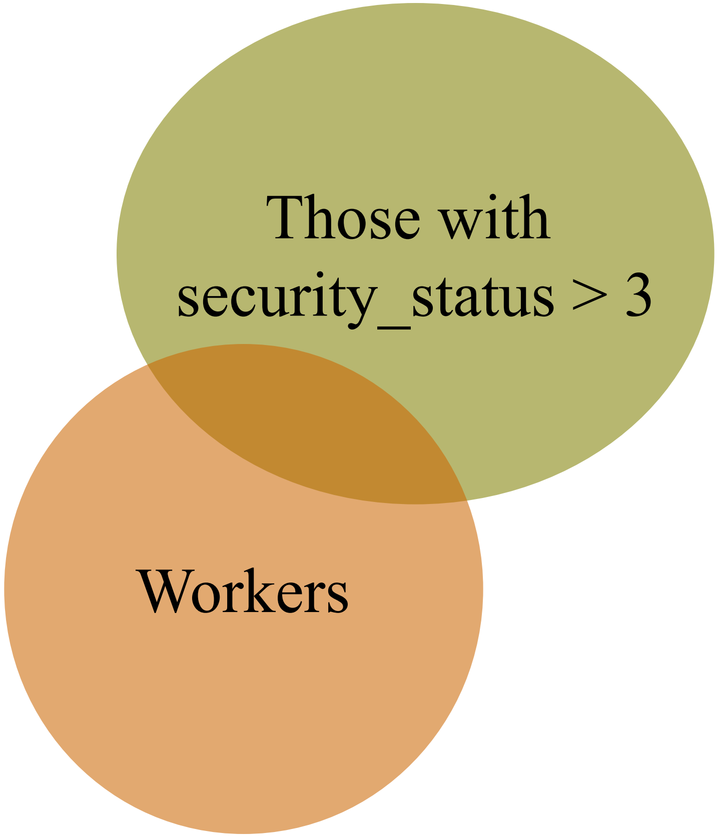
```
MultiMedia = Content \ Textual
```



Dynamic Relations

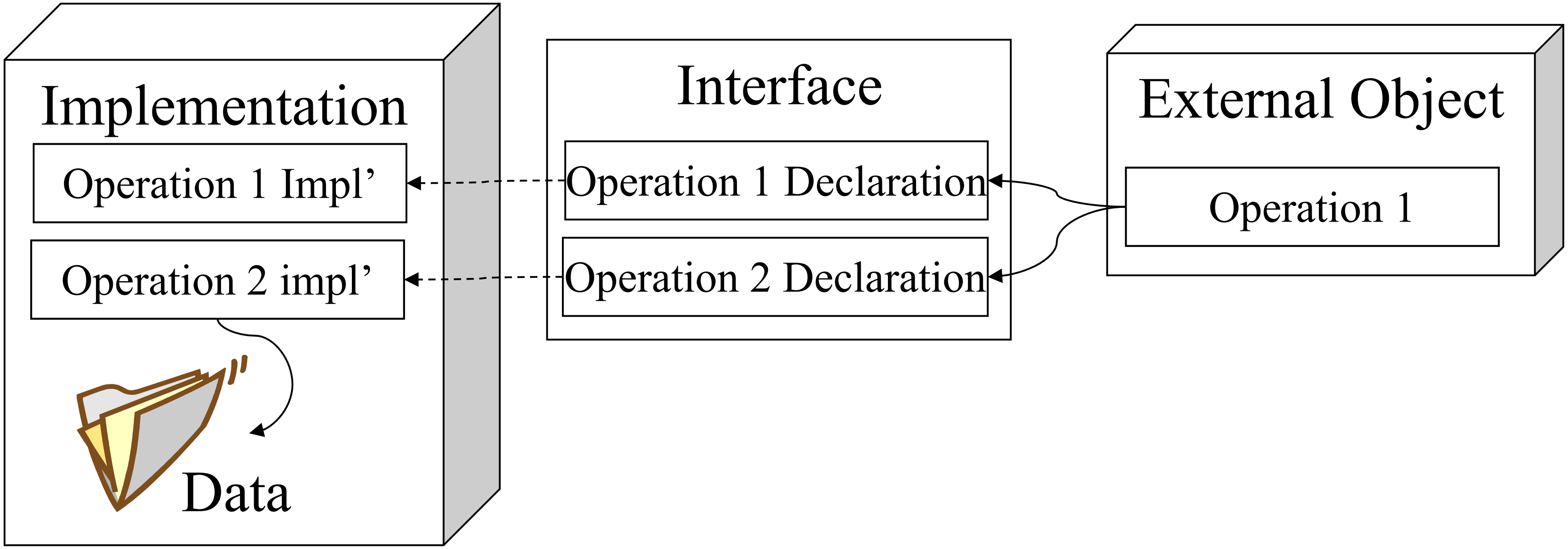
- Dynamic Intersection
  - We cannot create classes by dynamically intersecting between class properties



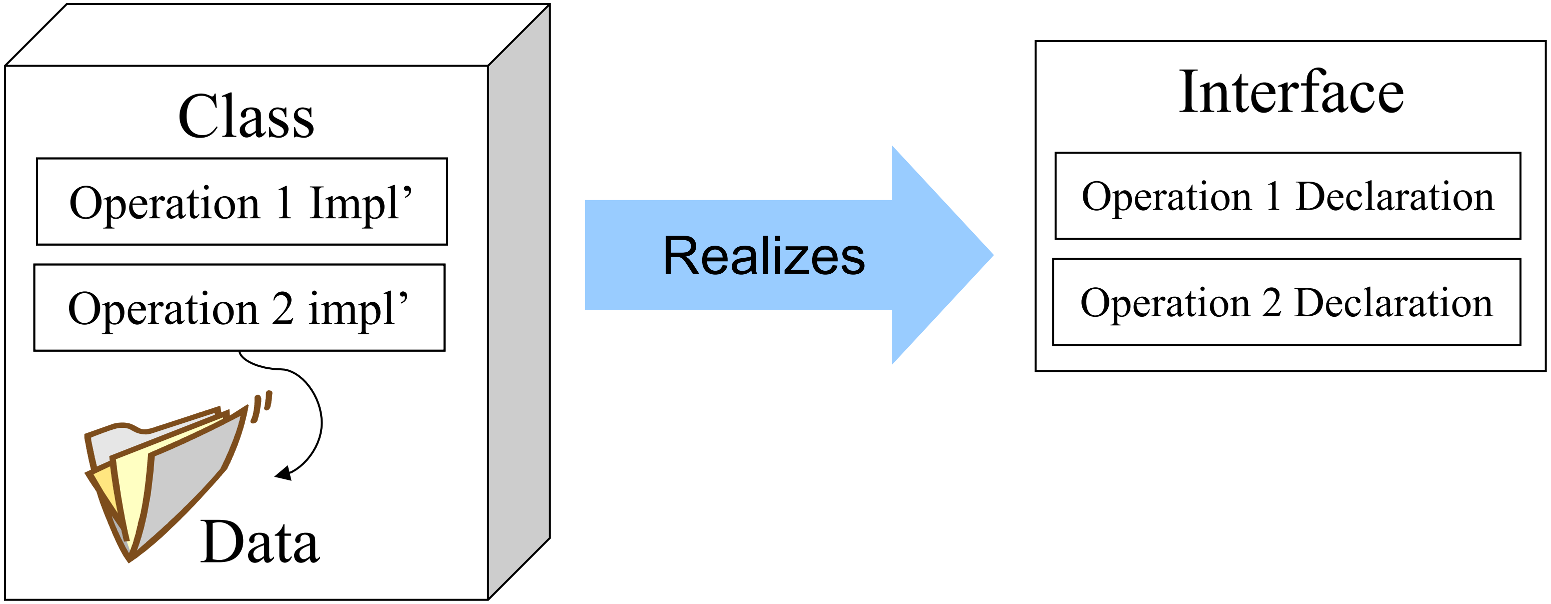


Encapsulation & Information Hiding

- Encapsulation is the separation between the external aspects of an object and its internals
- An *Interface* is:
  - A collection of method definitions for a set of behaviors – a "**contract**".
  - No implementation provided.

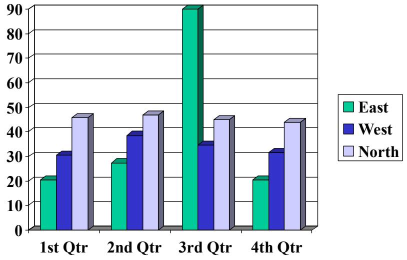


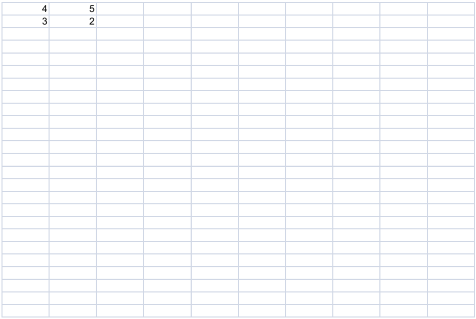
Interface Terminology



- Realization relation: - - - - - - - - - -▷

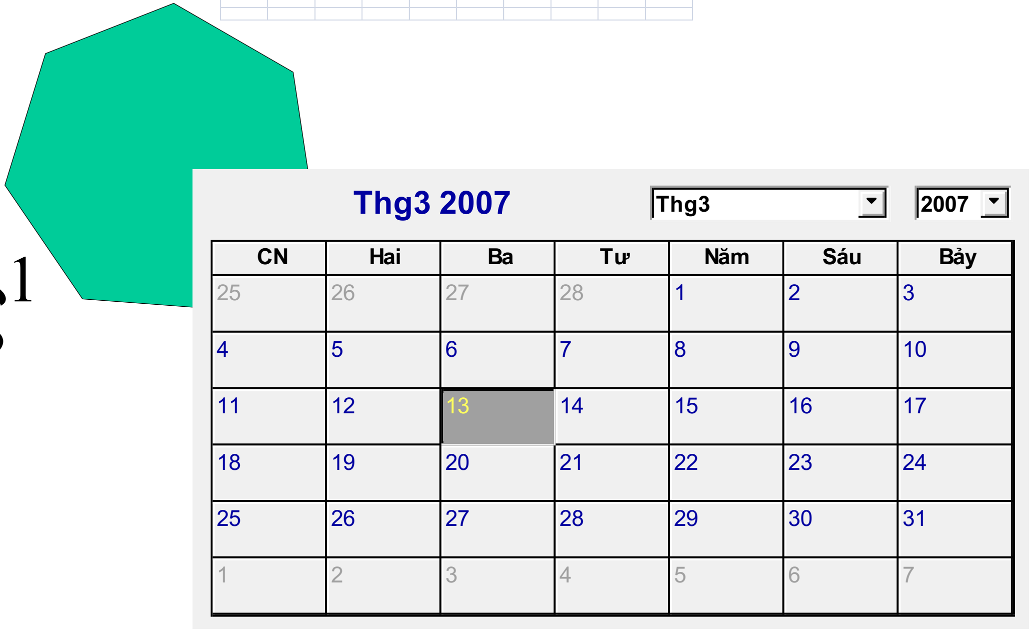
Example: Microsoft's Common Object Model





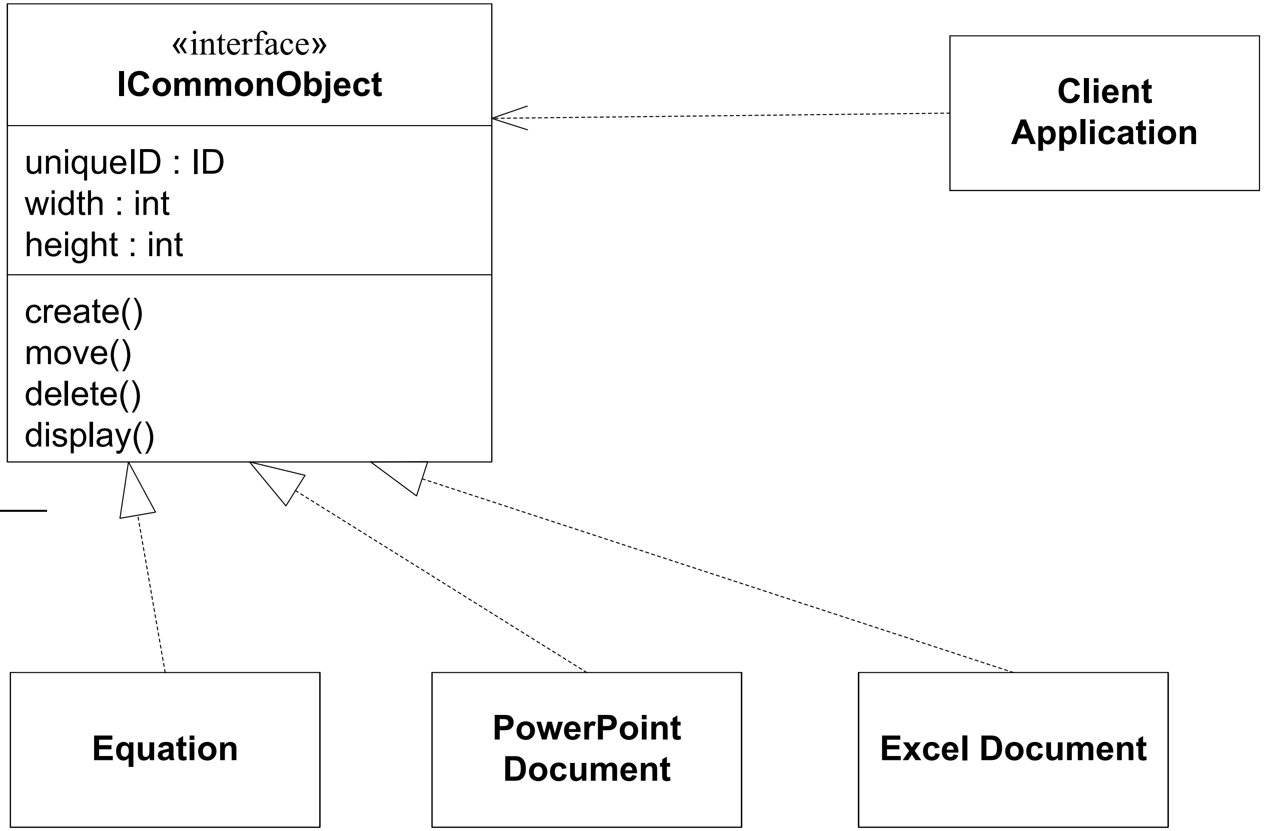
4	5
3	2

$$\sum_{0..\infty} \Lambda \nu \notin \aleph^1$$



CN	Hai	Ba	Tư	Năm	Sáu	Bảy
25	26	27	28	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7

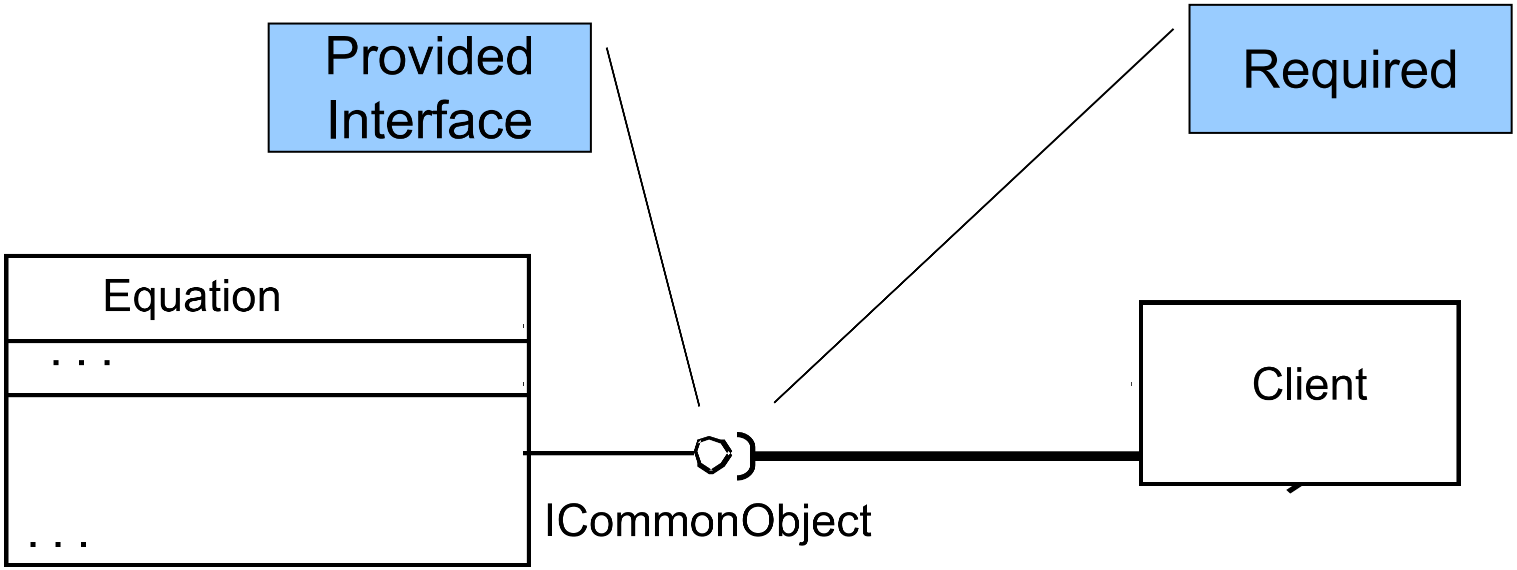
Interfaces Notation



Realization: The object guarantees to carry out the contract specified by the interface

Interfaces Notations - cont'd

- Another way to notate interfaces:



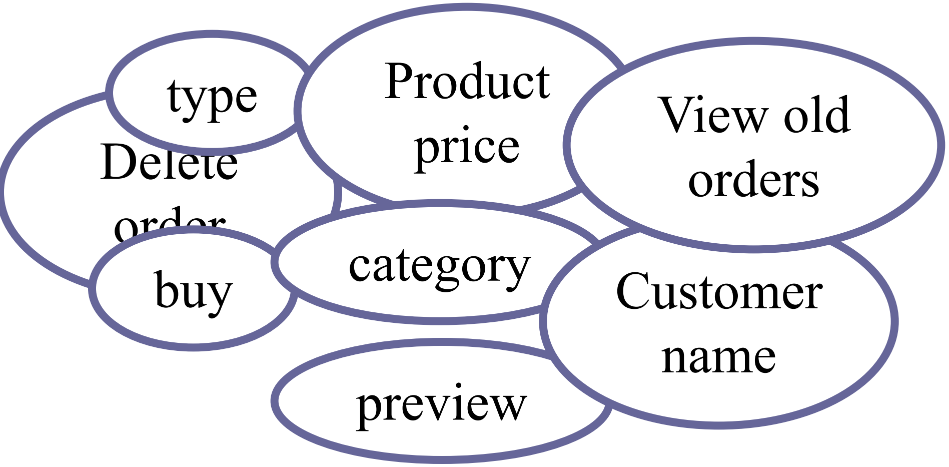
Outline

- Introduction
- Classes, attributes and operations
- Relations
- Generalization & Encapsulation
- Guidelines for effective class modeling

How to Model?

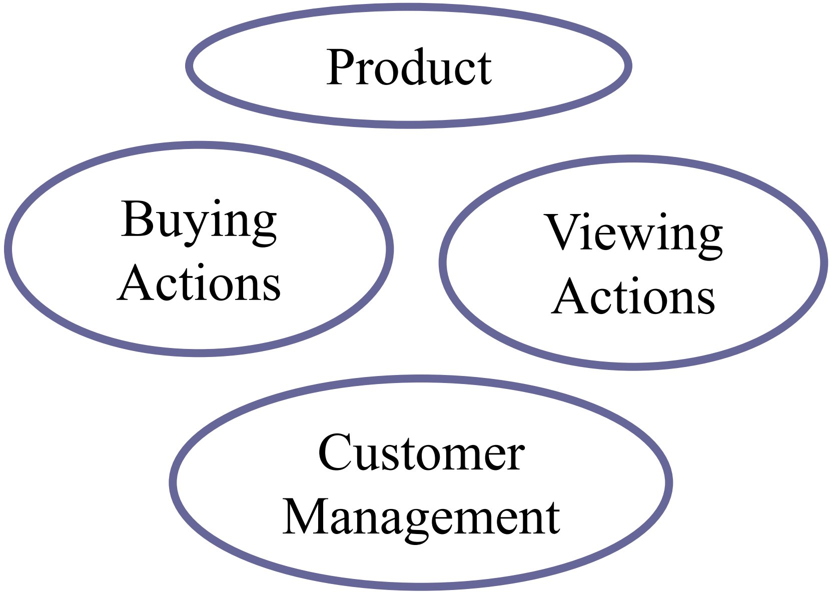
Bottom-up Process

Starting with throwing all classes on the page, and then combining them:



Top-down Process

Starting with an overview of the system, and then splitting classes

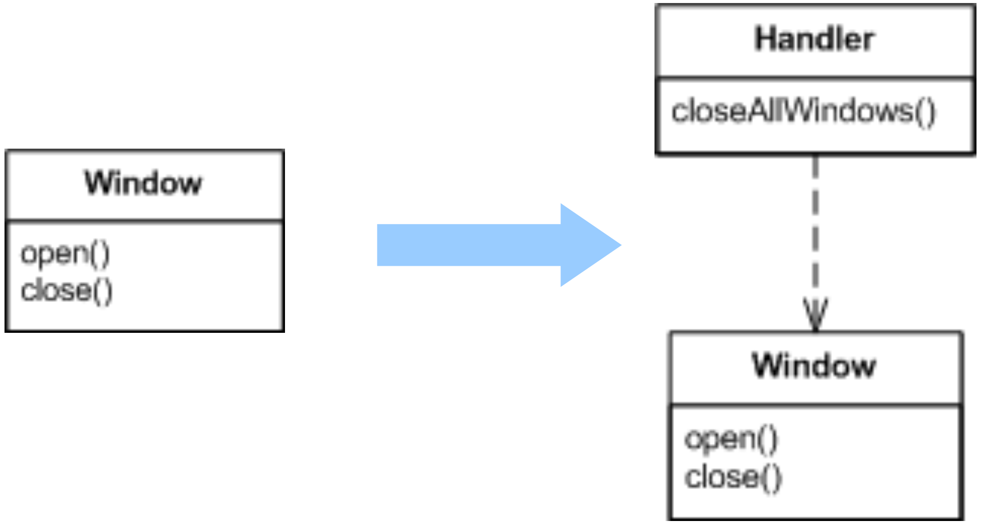


CRC Cards

- CRC Cards:
  - Class, Responsibility, Collaboration

Guidelines for Effective Class Diagram

- Identifying classes
  - Very similar to identifying data repositories in DFD. Identify data elements, and model them.
  - Plus, think of classes that handle processes. If operations are complicated enough, we might want to model them separately.
  - Plus, think of the actors. Are all their needs covered by existing operations?

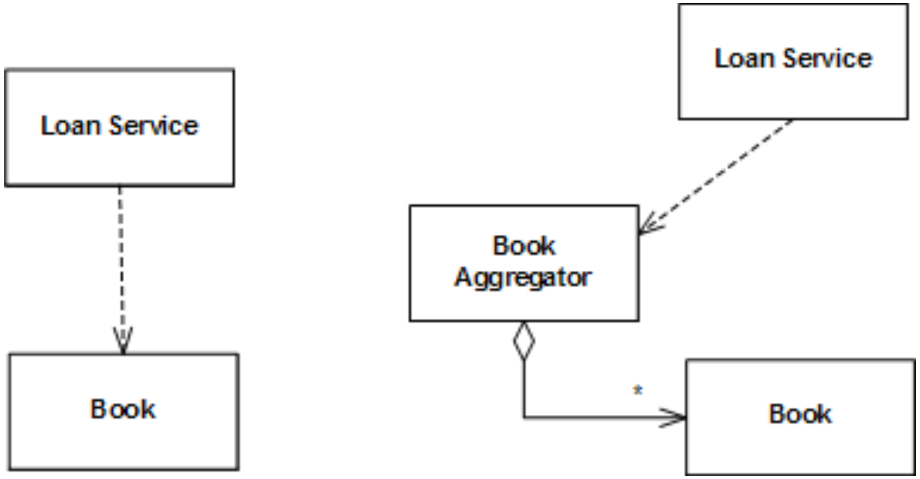


General Assumptions

- Access
  - Users can execute any public operation of the classes (except when using specialized stereotypes).
- Lifespan
  - Objects (except transient objects) have an endless life span.
  - We don’t need to bother with their serialization.
- Simplicity
  - No need for get/set.
  - No need for constructors / distracters .

Finding Objects

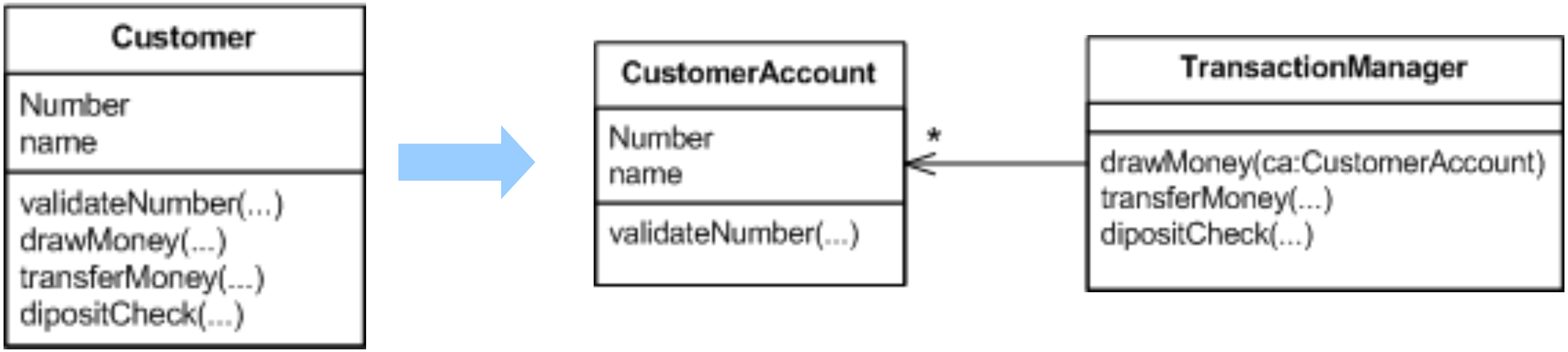
- Objects can be found, browsed through and located without any aggregating class.



That's enough for Loan Service to access all instances of Book

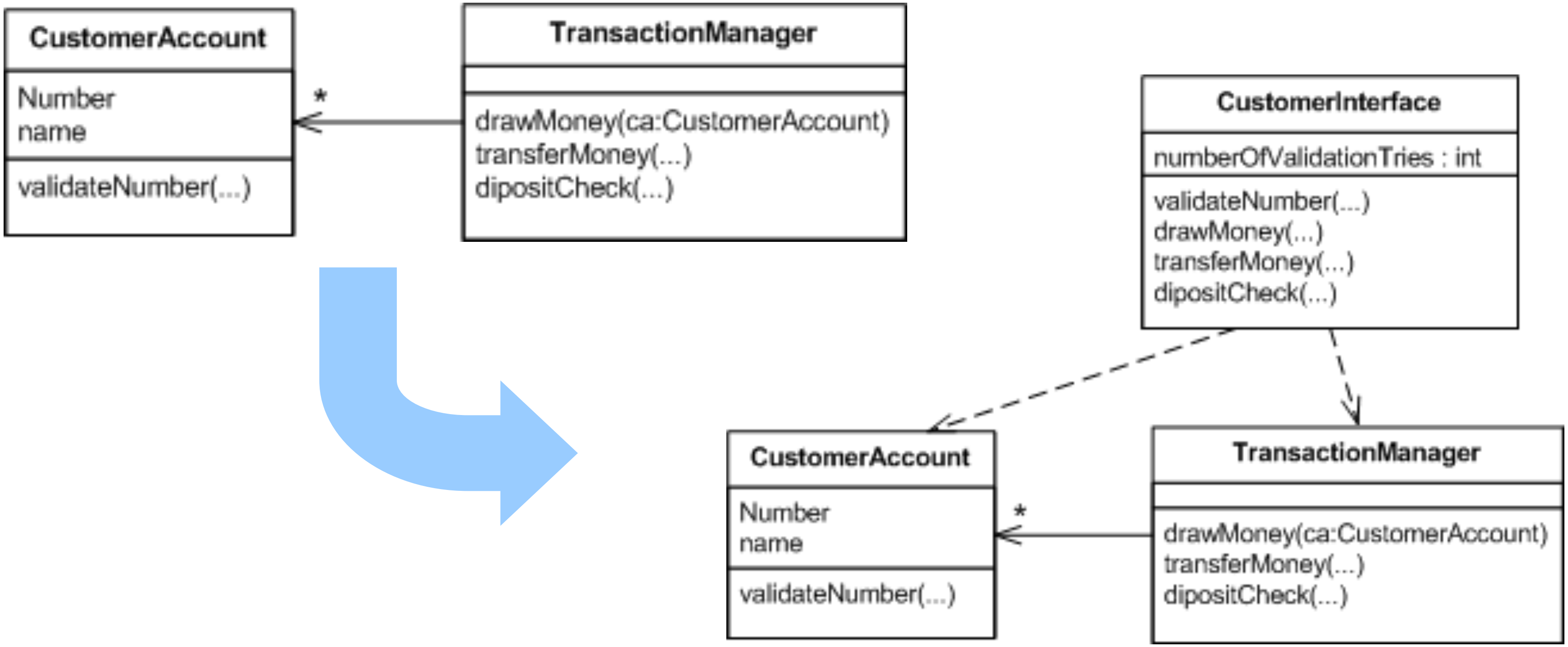
Guidelines – Modeling Actors

- A common mistake is to model **actors as classes**
- Remember -
  - Actors interact with the system directly, they don't need to be represented a priory
  - Sometimes, the system saves data about customers, but it does not mean that they do all their actions through this class



Guidelines – User Interfaces

- If the user has complicated interactions with the system, then we may want to dedicate a special class as a “user interface”
- Remember – it’s not the same class as the class that contains data about the actor

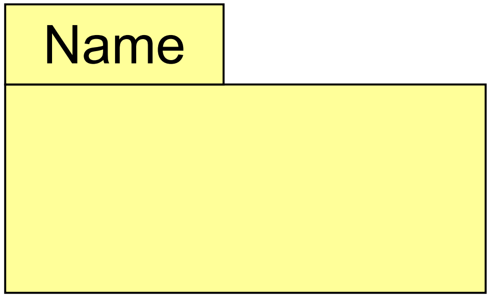


Summary

- ✓ Introduction
  - Structural modeling
- ✓ Classes
  - Attributes and operations
- ✓ Relations
  - Associations, constraints
  - Dependencies, compositions
- ✓ Generalization
  - Inheritance
  - Interfaces
- ✓ Object Diagrams
- ✓ Guidelines for effective class modeling

UML Packages

- A package is a general purpose grouping mechanism.
- Commonly used for specifying the logical architecture of the system.
- A package does not necessarily translate into a physical sub-system.

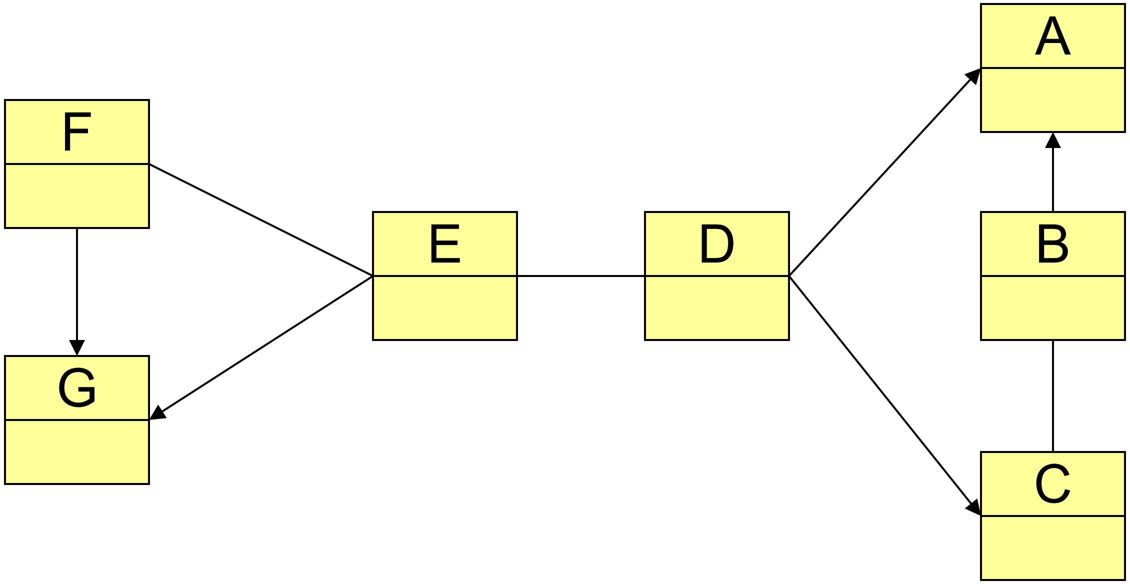


UML Packages (cont’d)

- Emphasize the logical structure of the system (High level view)
- Higher level of abstraction over classes.
- Aids in administration and coordination of the development process.

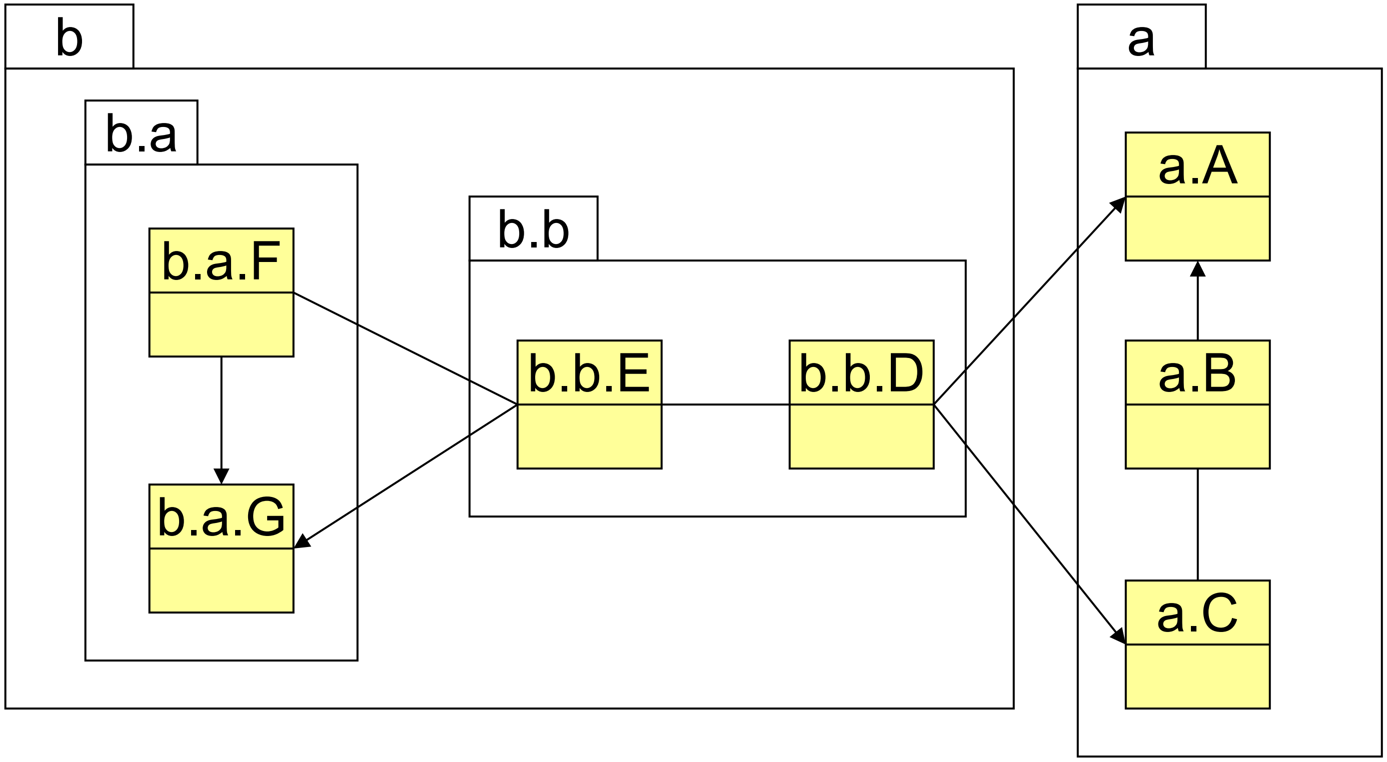
Packages and Class Diagrams

- Add package information to class diagrams



Packages and Class Diagrams

- Add package information to class diagrams



Analysis Classes

- A technique for finding analysis classes which uses three different perspectives of the system:
  - The boundary between the system and its actors
  - The information the system uses
  - The control logic of the system

Boundary Classes

- Models the interaction between the system's surroundings and its inner workings
- User interface classes
  - Concentrate on what information is presented
  - Don't concentrate on visual asspects
  - Example: ReportDetailsForm
- System / Device interface classes
  - Concentrate on what protocols must be defined.
  - Don't concentrate on how the protocols are implemented

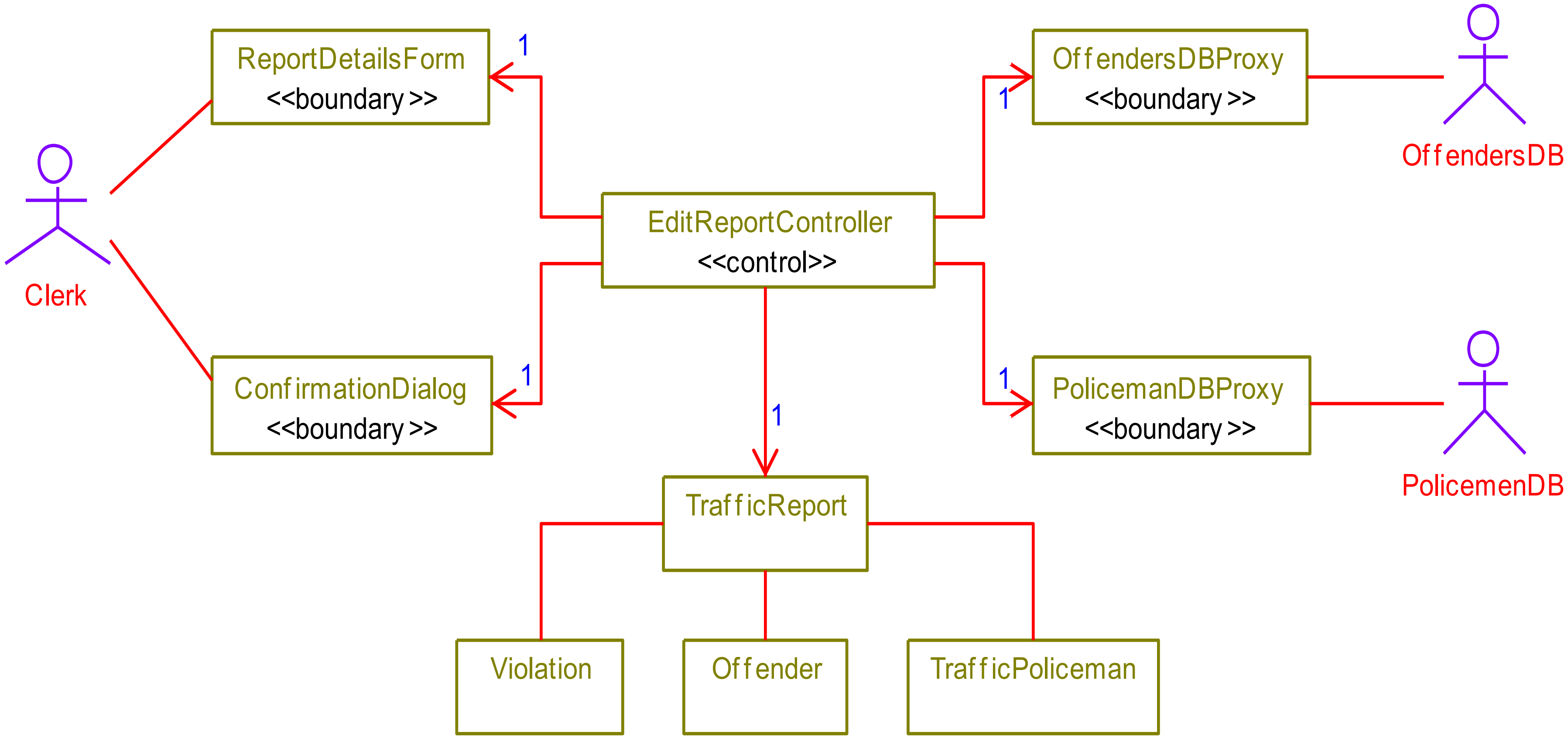
Entity Classes

- Models the key concepts of the system
- Usually models information that is persistent
- Can be used in multiple behaviors
- Example: Violation, Report, Offender.

Control Classes

- Controls and coordinates the behavior of the system
- Delegates the work to other classes
- Control classes decouple boundary and entity classes
- Example:
  - EditReportController
  - AddViolationController

TVRS Example



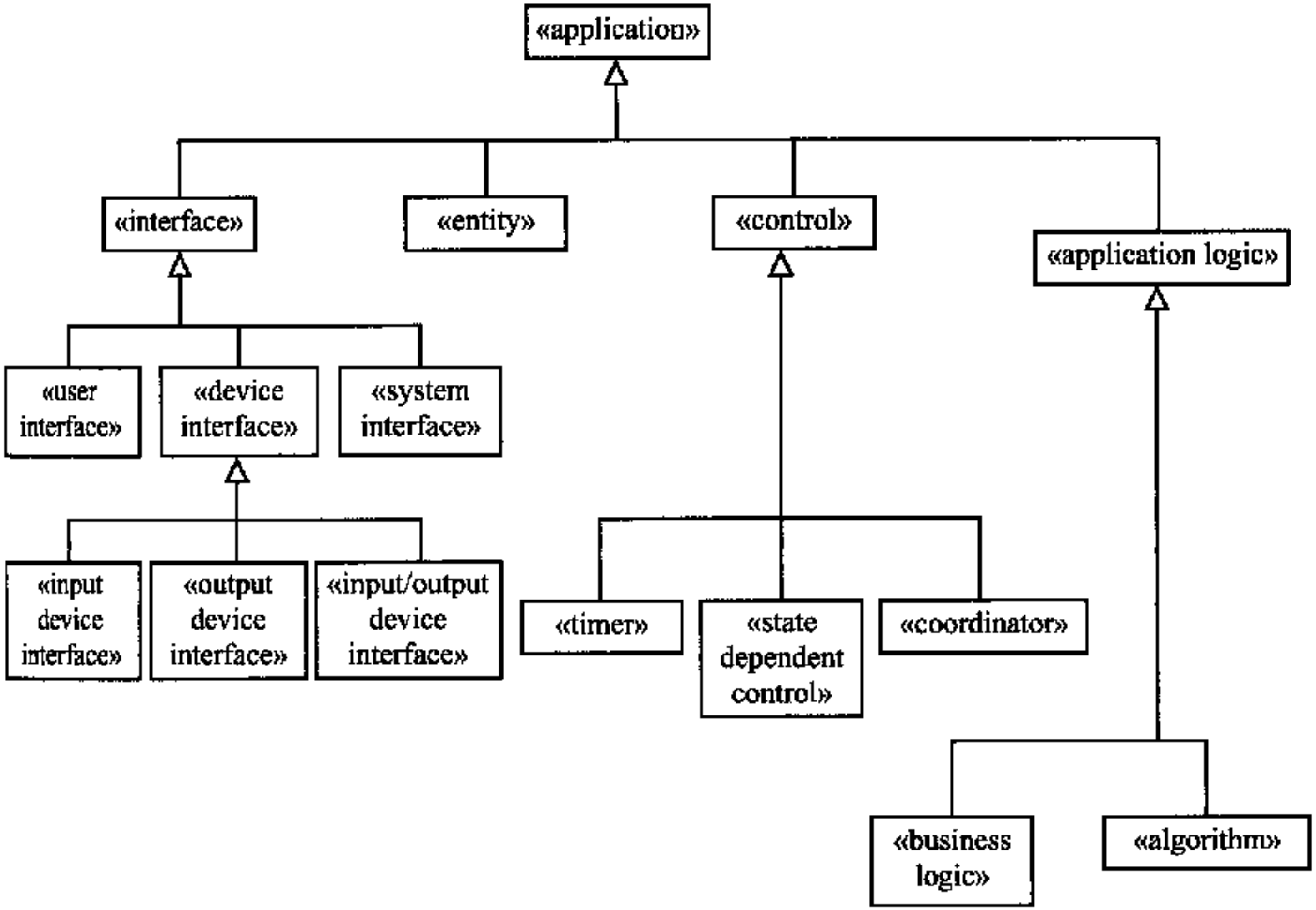
OBJECT AND CLASS CONSTRUCTING

Objectives

- Provide guidelines on how to determine the classes/objects in the system

- Define class/object structuring criteria

Categorization of Application Classes

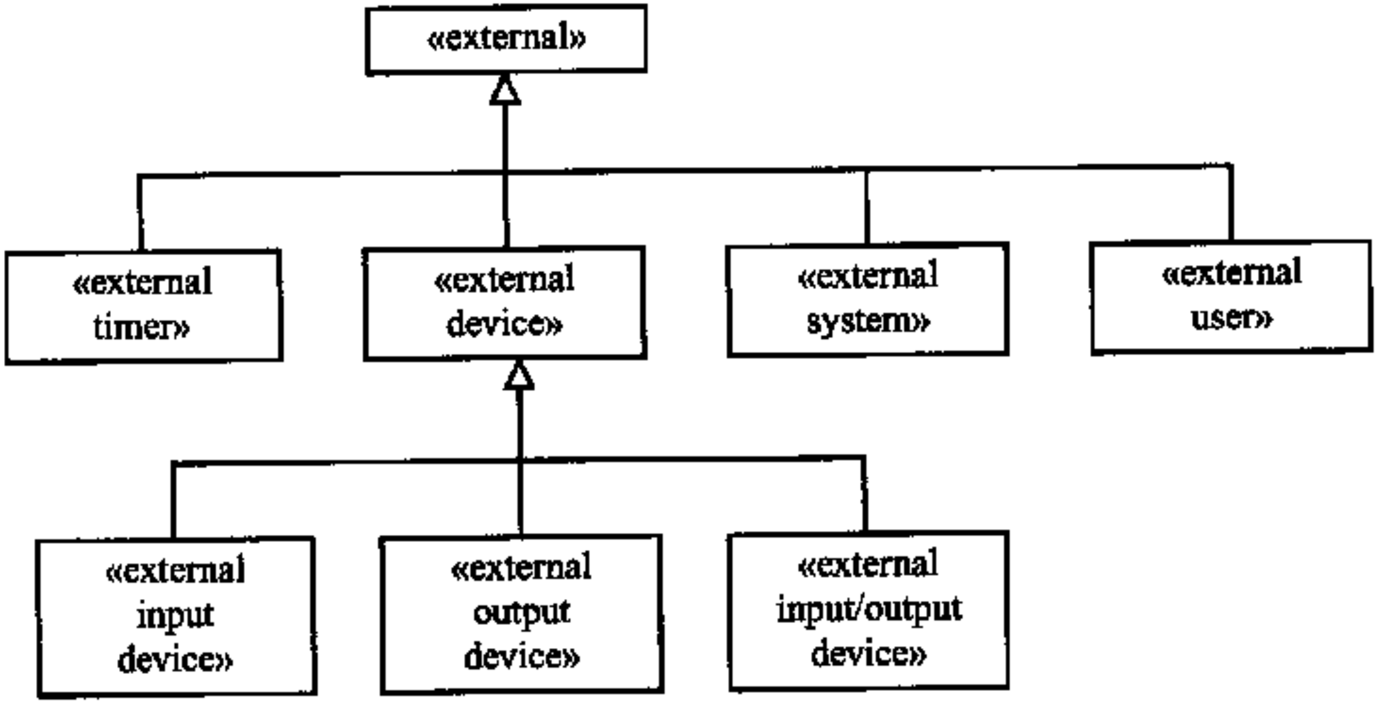


External Classes and Interface Classes

- External classes are classes that are external to the system and that interface to the system.

- Interface (boundary) classes are classes internal to the system that interface to the external classes.

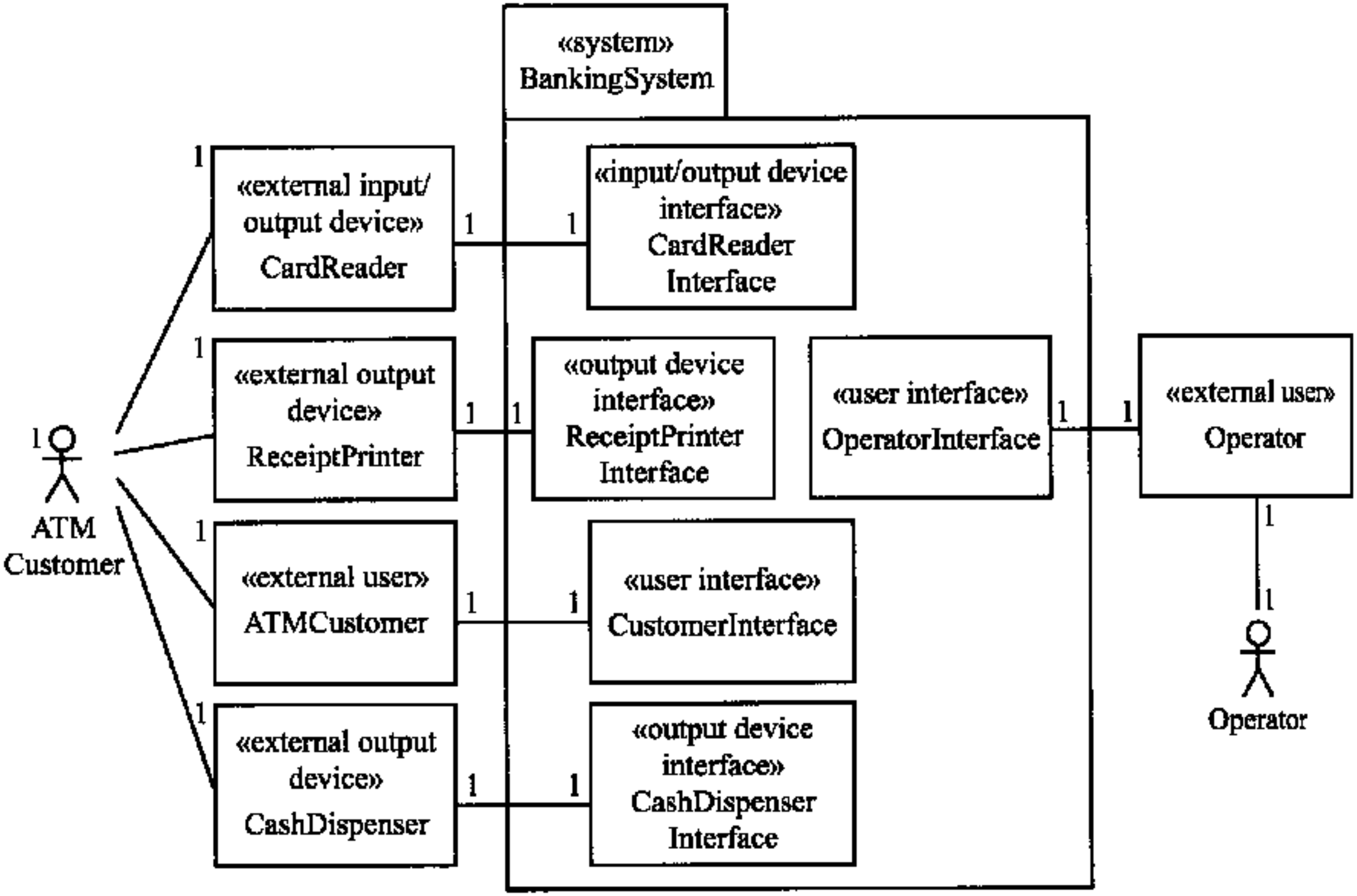
Categorization of External Classes



Identifying Interface Classes

- Each of the external classes interfaces to an interface class in the system.
  - An external user class interfaces to a user interface class
  - An external system class interfaces to a system interface class
  - An external input device class interfaces to an input device interface class
  - An external output device class interfaces to an output device interface class
  - An external I/O device class interfaces to an I/O device interface class
  - An external timer class interfaces to an internal timer class

Banking System: External Classes and Interface Classes


«system» BankingSystem
«external input/ output device» CardReader
«input/output device interface» CardReader Interface
«external output device» ReceiptPrinter
«output device interface» ReceiptPrinter Interface
«user interface» OperatorInterface
«external user» Operator
«external user» ATMCustomer
«user interface» CustomerInterface
«external output device» CashDispenser
«output device interface» CashDispenser Interface
ATM Customer
Operator
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1

Entity Classes

- Store information

- Often mapped to relational database during design

Control Classes

- A control class provides the overall coordination for execution of a use case.
- Makes overall decision
- Control objects decides when, and in what order, other objects participate in use case
  - Interface objects
  - Entity objects
- Simple use cases do not need control objects.

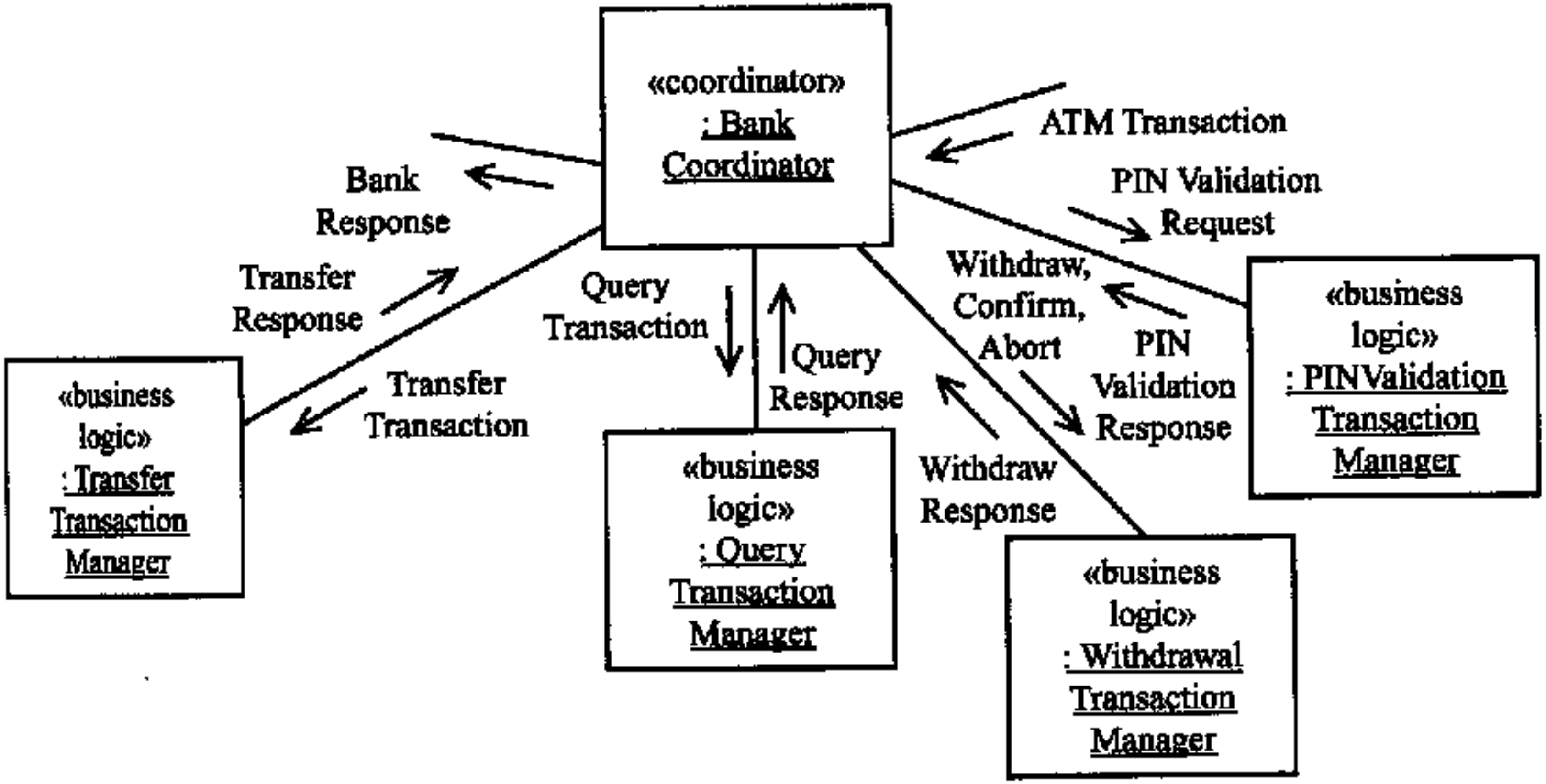
Kinds of Control Classes

- Coordinator class
  - Provides sequencing for use case
  - Is not state dependent

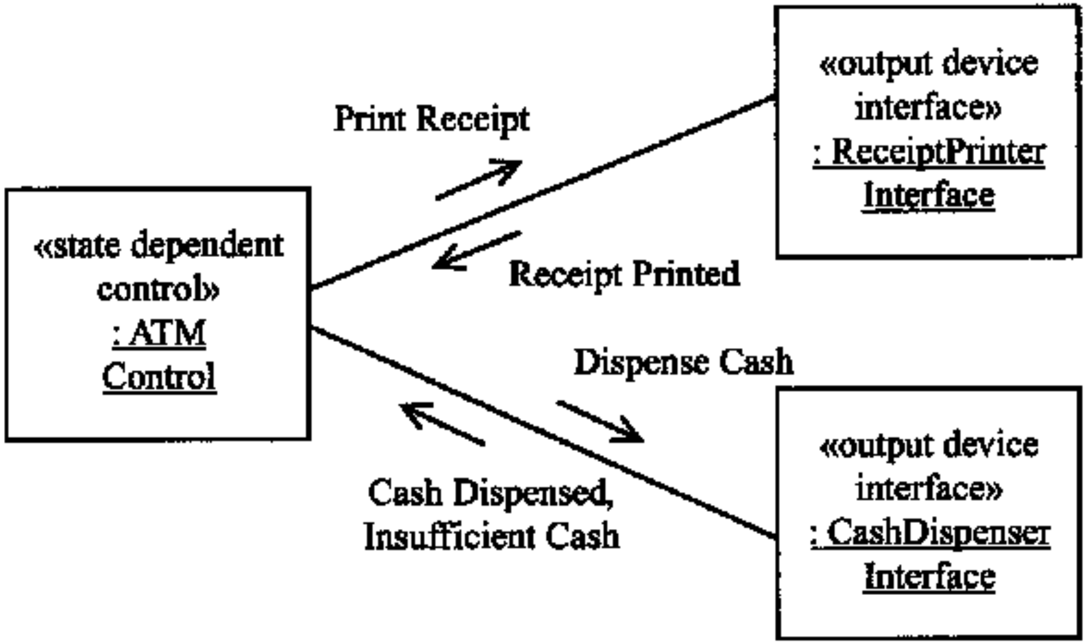
- State dependent control class
  - Defined by finite state machine

- Timer class
  - Activated periodically

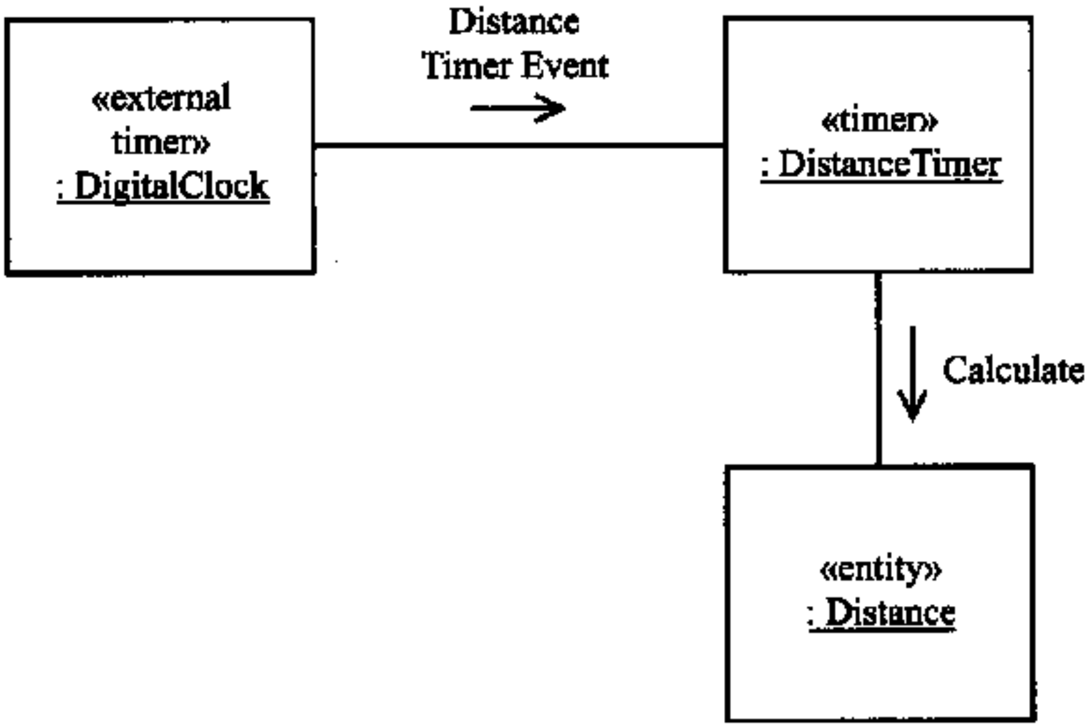
Example: Coordinator Object



Example: State Dependent Control Object


«state dependent control»
: ATM Control
Print Receipt
Receipt Printed
«output device interface»
: ReceiptPrinter Interface
Dispense Cash
Cash Dispensed, Insufficient Cash
«output device interface»
: CashDispenser Interface

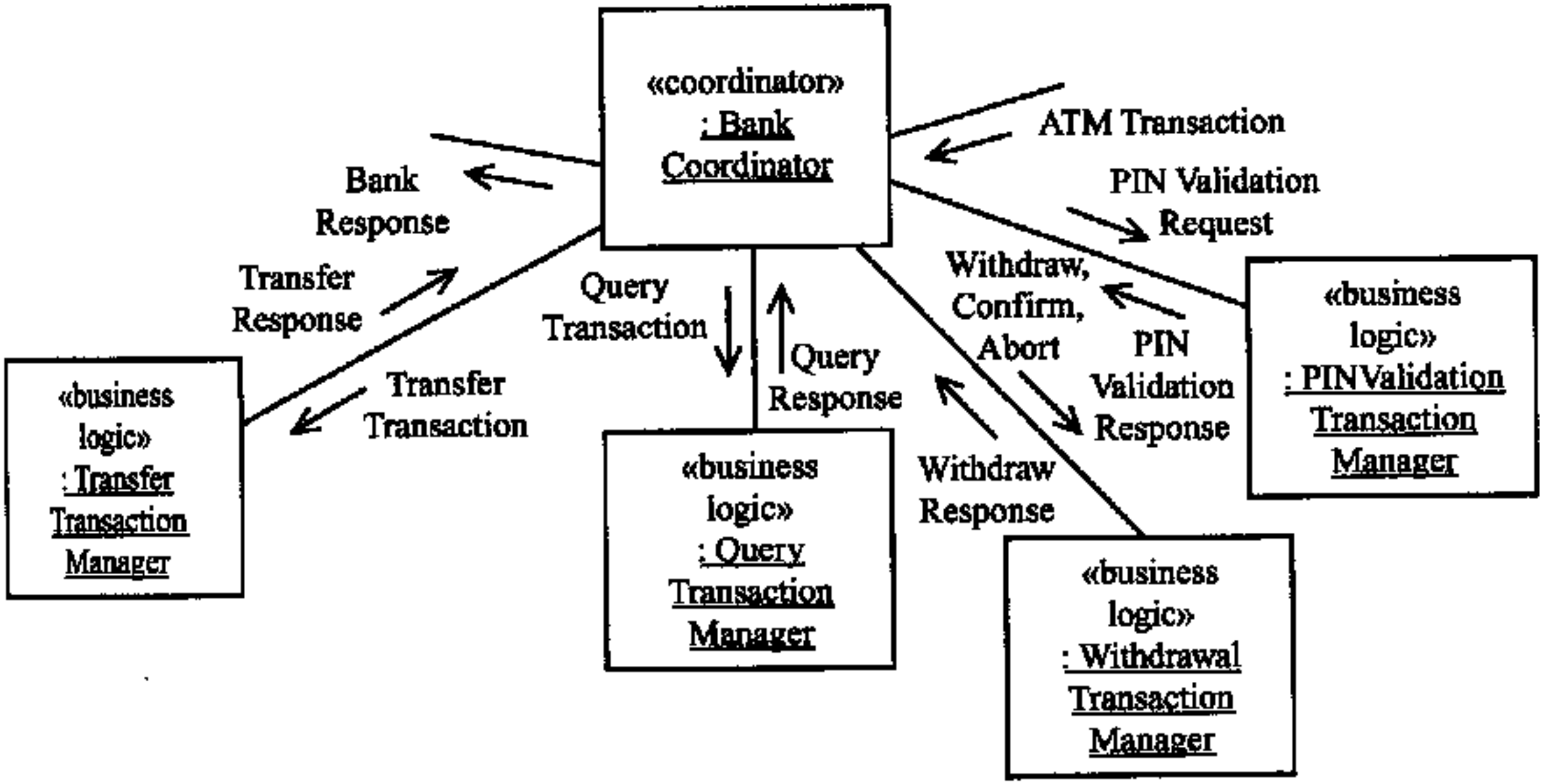
Example: Timer Object



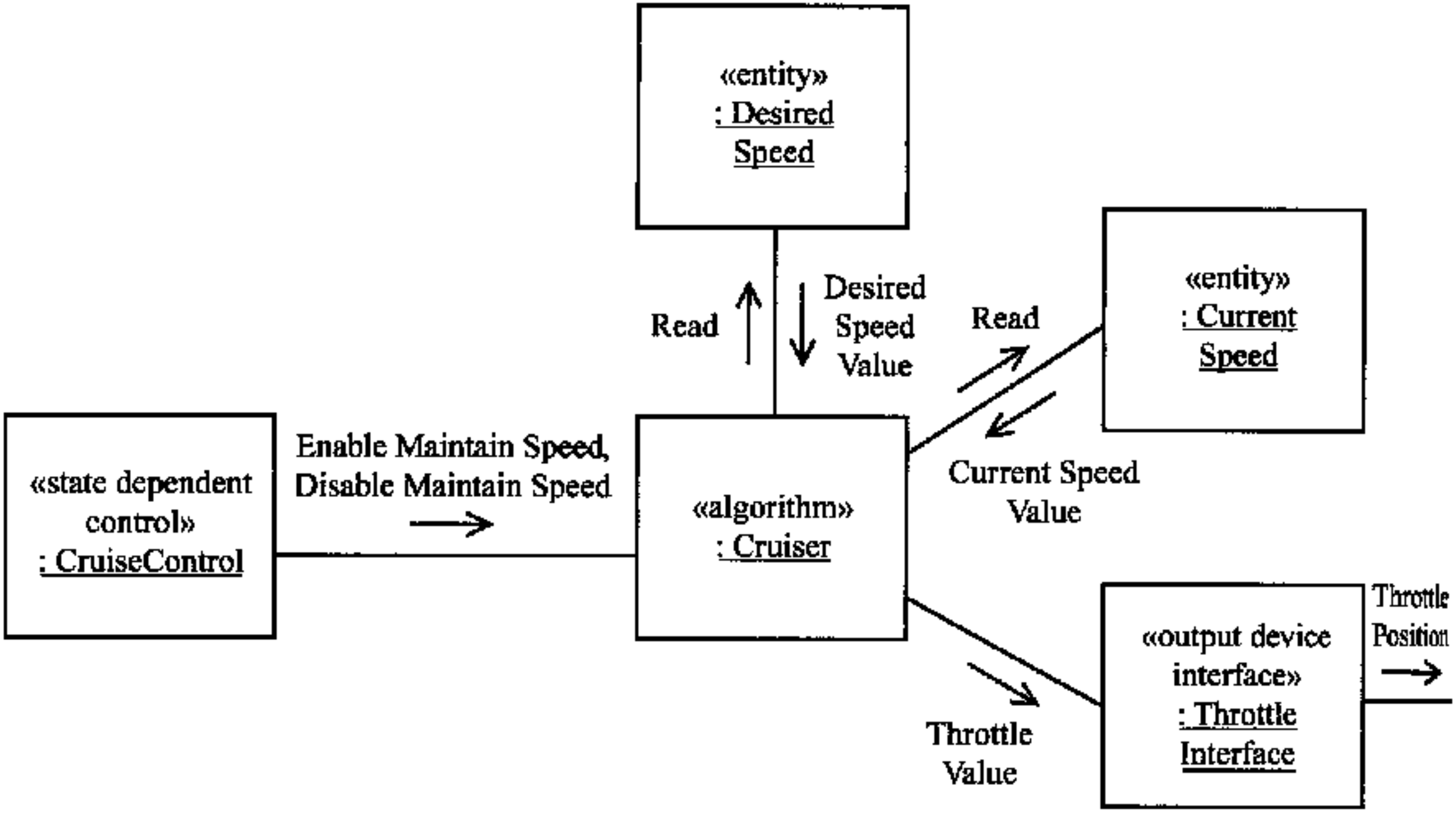
Application Logic Classes

- Business logic class
  - Defines business-specific application logic (rules) for processing a client request
  - Usually accesses more that one entity object
- Algorithm class
  - Encapsulates algorithm used in problem domain
  - More usual in scientific, engineering, real-time domains

Example: Business Logic Object



Example: Algorithm Object



Tips

- Don’t try to use all the various notations.
- Don’t draw models for everything, concentrate on the key areas.
- Draw implementation models only when illustrating a particular implementation technique.