O-O, Classes and Class Diagrams

What is Object-Oriented?

- Encapsulation 封装
- Information/Implementation hiding
- State retention 状态保持
- Object identity
- Messages
- Classes
- Inheritance 继承
- Polymorphism 多态
- Genericity 泛型

Example: An Auction

- Two classes: Auction and AuctionItem
- Objects of these types interact, by sending messages
- They are used in a larger program

AuctionItem

New:AuctionItem(details: text)

//creates and returns a new instance of
//AuctionItem, replete with description, starting bid,
//reserve, end date

CurrentBid: Currency

// returns the highest bid so far

BidMade(amount:Currency, **out** accepted:Boolean)

// adds a bid for the item, returns true if greater than
//previous highest bid

IsSold: Boolean

// returns true if highest bid greater than reserve &
//end date passed

Display:

// shows AuctionItem details on screen

Auction

New:Auction

//creates and returns a new instance of Auction

Insert(newItem:AuctionItem)

// adds a new item to its collection

Remove(soldItem: AuctionItem)

// removes an item from its collection

ListenForNewBid(itemID: int, amount: Currency **out** isNewBid Boolean)

// sees if a new bid has been submitted

ListenForNewItem(details: text **out** isNewItem Boolean)

// sees if a new auction item has been submitted

NextItem: AuctionItem

// retrieves next AuctionItem in collection

FirstItem: AuctionItem

LastItem: AuctionItem

DisplayItems

// shows all AuctionItems currently in its collection

```
Auction auction = new Auction;
text details;
int itemID;
int bidPrice;
AuctionItem ai;
Boolean bidAccepted;

while(true) {
  for (ai = auction.First() through ai = auction.Last() ) {
    if (ai.IsSold() )
      Remove(ai);
    auction.DisplayItems();
  }
    if ( ListenForNewBid(itemID, bidPrice) )
      AuctionItem(itemID).BidMade(bidPrice, bidAccepted);
    if ( ListenForNewItem(details) )
      auction.Insert(new AuctionItem(details) );
}
```

Encapsulation

- "…is the grouping of related ideas into one unit, which can thereafter be referred to by a single name."
- Subroutines are an older example.
- Objects encapsulate attributes and methods.
- Attributes are available *only* through methods.

Info/Implementation Hiding

- "…is the use of encapsulation to restrict from external visibility certain information or implementation details…"
- Inside vs. outside views of an object.
- How many items in Auction's collection?
- How is the collection of items managed?
- Well-designed objects are "black boxes".

State Retention

- When a function finishes, its just goes away; its state is lost.
- Objects can retain their state from method call to method call.
- (Of course objects themselves can go out of scope and thus be lost.)
- Encapsulation, hiding and state retention characterize an *abstract data type*.

Object Identity

- "…the property by which each object can be identified and treated as a distinct software entity."
- `Auction auction = new Auction;`
  - The new Auction object has a handle, which is stored in the variable auction.
  - The same handle remains with the object throughout its lifetime.
  - No two objects have the same handle.
  - The handle is often a memory address, but may be a location on the net, or…

Messages

- "…the vehicle by which a sender object **obj1** conveys to a target object **obj2** a demand for **obj2** to apply one of its methods."
- `auction.Insert(new AuctionItem(details) );`
- Need
  - Object handle
  - Name of method
  - Additional information (arguments)

Messages (cont.)

- "Pre-OO" style:
  - `call insert(auction, details);`
- In O-O, the "additional details" are themselves objects (with handles, and perhaps their own methods)
- So, an object can be
  - A sender
  - A target
  - Pointed to by a variable in another object
  - Pointed to by an argument passed forward or backward in a message

Messages Types

- Informative: provides info for an object to update itself
  - `auctionItem1.BidMade(amount, accepted)`
- Interrogative: request for object to provide info
  - `auctionItem1.IsSold()`
- Imperative: request that an object take some action
  - `auction.Insert(auctionItem3)`

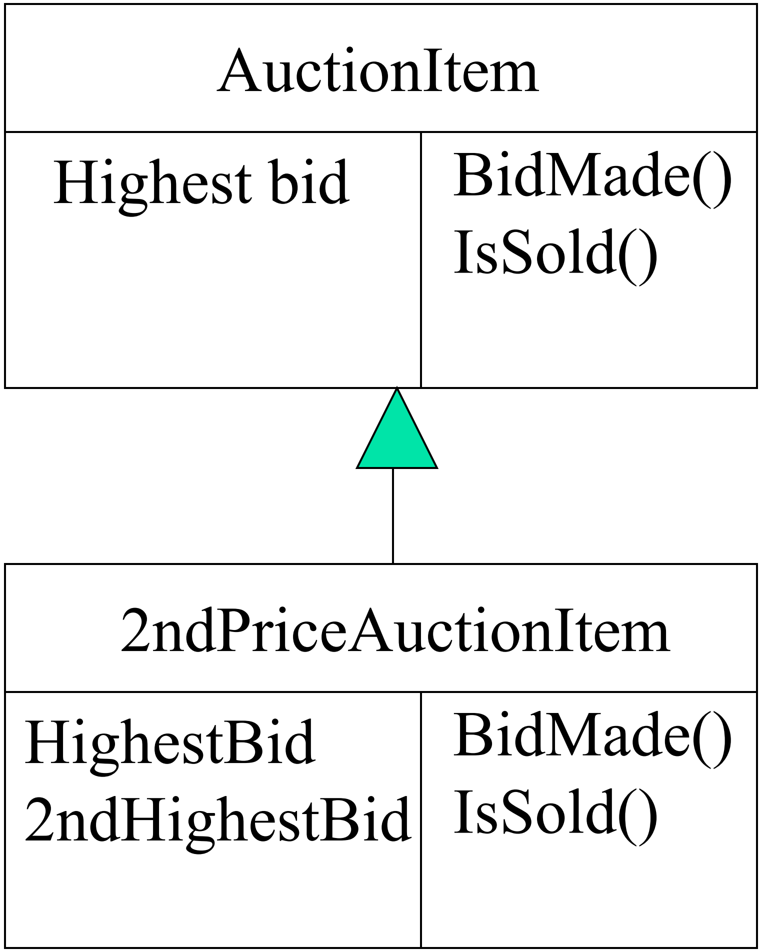
Classes

- ‟ a stencil from which objects (instances) are created”
- Each object from the same class has the same attributes and methods, but different handles and states.
- Each object of a given class has its own attributes, but methods are shared.
- The class itself may have attributes and methods (e.g., static variables, constructor).

Inheritance

- When class **child** inherits from class **parent**, **child** has defined for it each attribute and operation defined for **parent**.
- We build a class for the general case, then specialize.
- Start with Auction, then derive SecondPriceAuction, DutchAuction etc.

AuctionItems



AuctionItem

Highest bid

BidMade()
IsSold()

2ndPriceAuctionItem

HighestBid
2ndHighestBid

BidMade()
IsSold()

Polymorphism

- The same attribute or operation may be defined on more than one class, with possibly different implementations.
- Also, the ability of a handle to point to objects of different classes at different times.
  - Ex: polymorphic operation: BidMade(amount)
  - Ex:

```
AuctionItem item1;
item1 = new 2ndPriceAuction;
```

Minor Definitions

- Dynamic binding: determining the exact piece of code to execute at run-time rather than compile-time.
- Overriding: redefining a method of a superclass within a subclass
- Overloading: several operations, in the same class, with the same name.

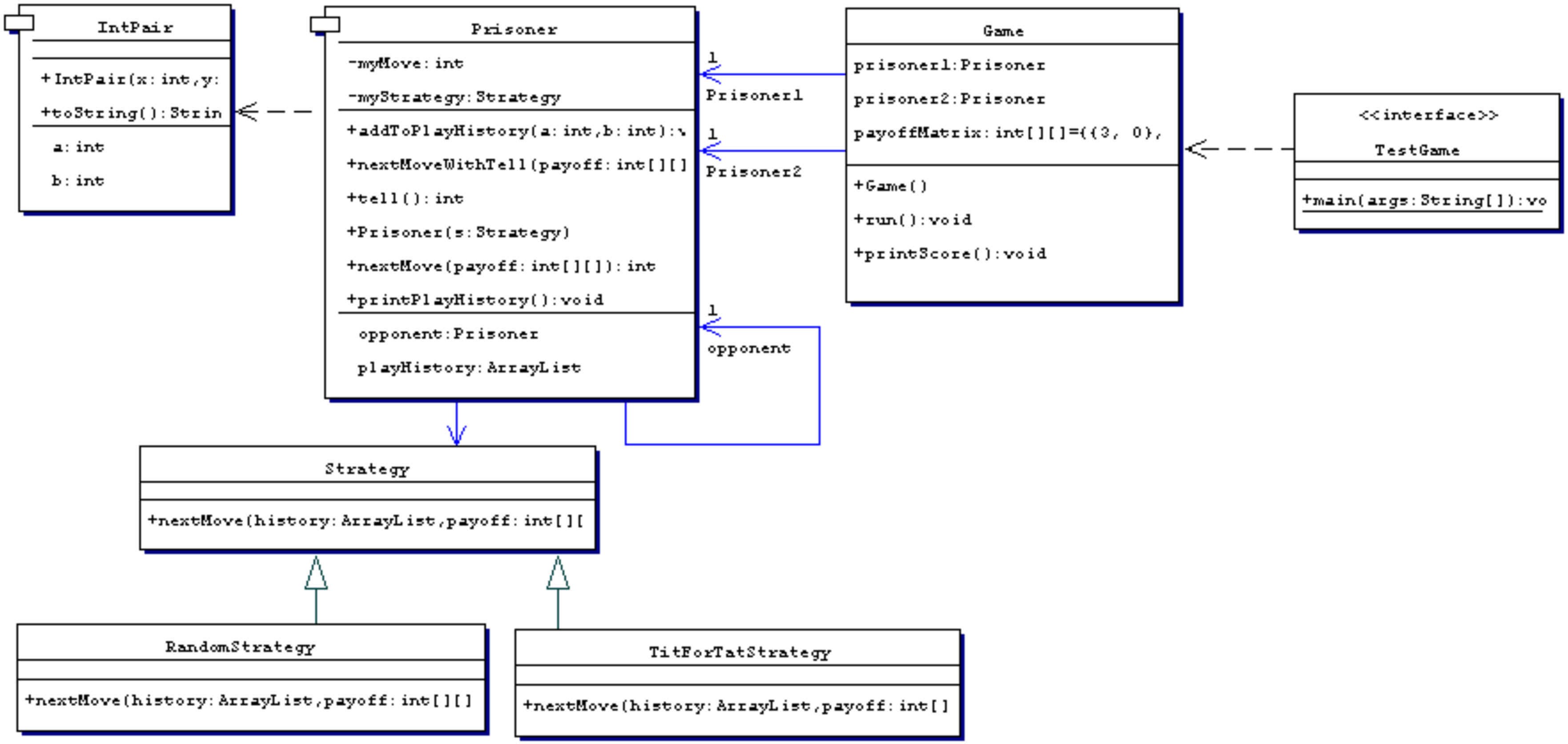
Genericity

- "…the construction of a class **C** so that one or more of the classes it uses internally is supplied only at run-time."
- Template classes are an example, sort of…The type must be specified at compile-time.
- Better example: the Strategy pattern

World Digest, 1949

- “Professor Neumann, of Princeton University, USA, has revealed that 40 per cent of the 800 scientists and technicians employed on the construction of ‘mechanical brains’ have gone mad. The majority, installed in American military mental homes, spend their time in prodigious arithmetical calculations.”

Classes and Class Diagrams



IntPair

+IntPair(x:int,y:

+toString():Strin

a:int

b:int

Prisoner

-myMove:int

-myStrategy:Strategy

+addToPlayHistory(a:int,b:int):\

+nextMoveWithTell(payoff:int[][]

+tell():int

+Prisoner(s:Strategy)

+nextMove(payoff:int[][]):int

+printPlayHistory():void

opponent:Prisoner

playHistory:ArrayList

1

Prisoner1

1

Prisoner2

1

opponent

Game

prisoner1:Prisoner

prisoner2:Prisoner

payoffMatrix:int[][]={{3, 0},

+Game()

+run():void

+printScore():void

<<interface>>

TestGame

+main(args:String[]):vo

Strategy

+nextMove(history:ArrayList,payoff:int[][

RandomStrategy

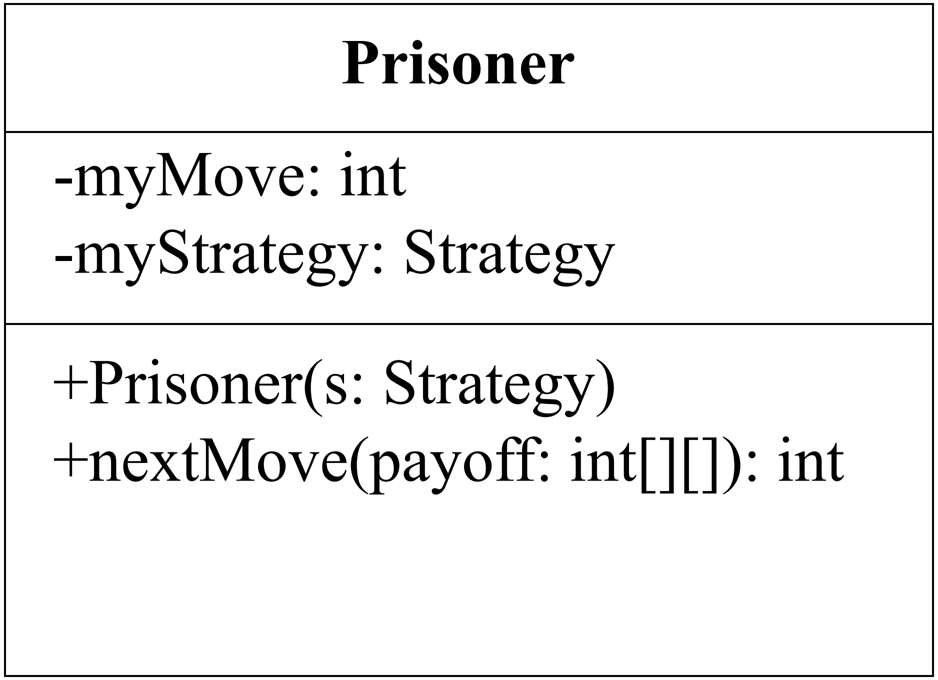
+nextMove(history:ArrayList,payoff:int[][

TitForTatStrategy

+nextMove(history:ArrayList,payoff:int[]

Notation for a Class

- Name, attributes, operations:



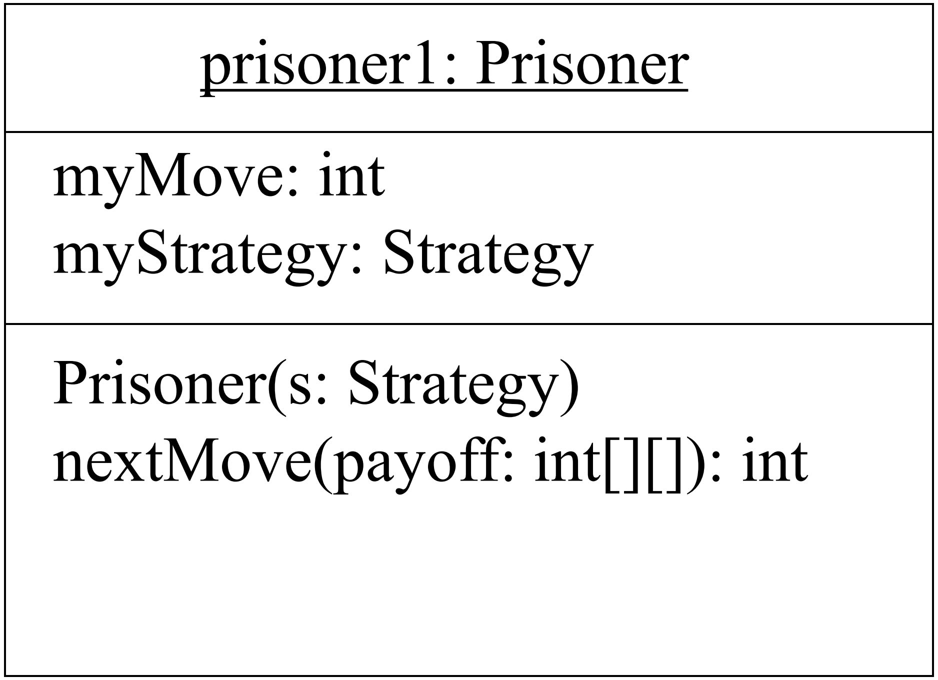
Prisoner

-myMove: int
-myStrategy: Strategy

+Prisoner(s: Strategy)
+nextMove(payoff: int[][]): int

Notation for an Object

- Object name, class name, attributes, operations:



prisoner1: Prisoner

myMove: int
myStrategy: Strategy

Prisoner(s: Strategy)
nextMove(payoff: int[][]): int

Attributes

- Info about an object
- Not quite the same as *variable*; attributes are abstractly defined properties, independent of internal representation
- *Public* (+) attributes are generally gettable and settable
- Some attributes may of course be private (-) or protected (#), or package local (friendly)

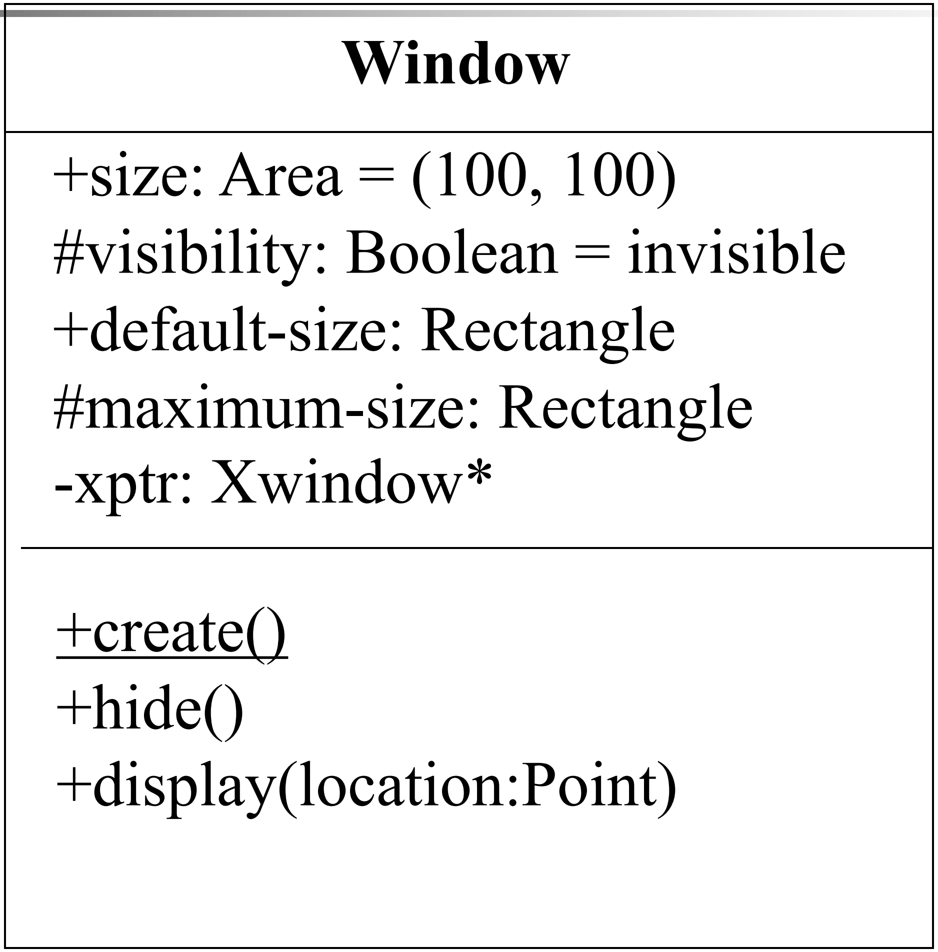
Operations

- Name, argument signature, and return type.
- Public, protected, and private.
- Together takes care of the notation, if you know how to write the code.
- Conversely, there are dialog boxes, millions of them!
- Together has a separate box for gettable/settable attributes, so get/set methods aren’t shown.

More on Notation

- Private attributes and operations aren’t part of the “interface,” so can (should?) be hidden.
- Any reasonable tool should give you the option.
- “Class-scope” operations are supposed to be underlined. Together doesn’t seem to recognize constructors this way.

Complete Example



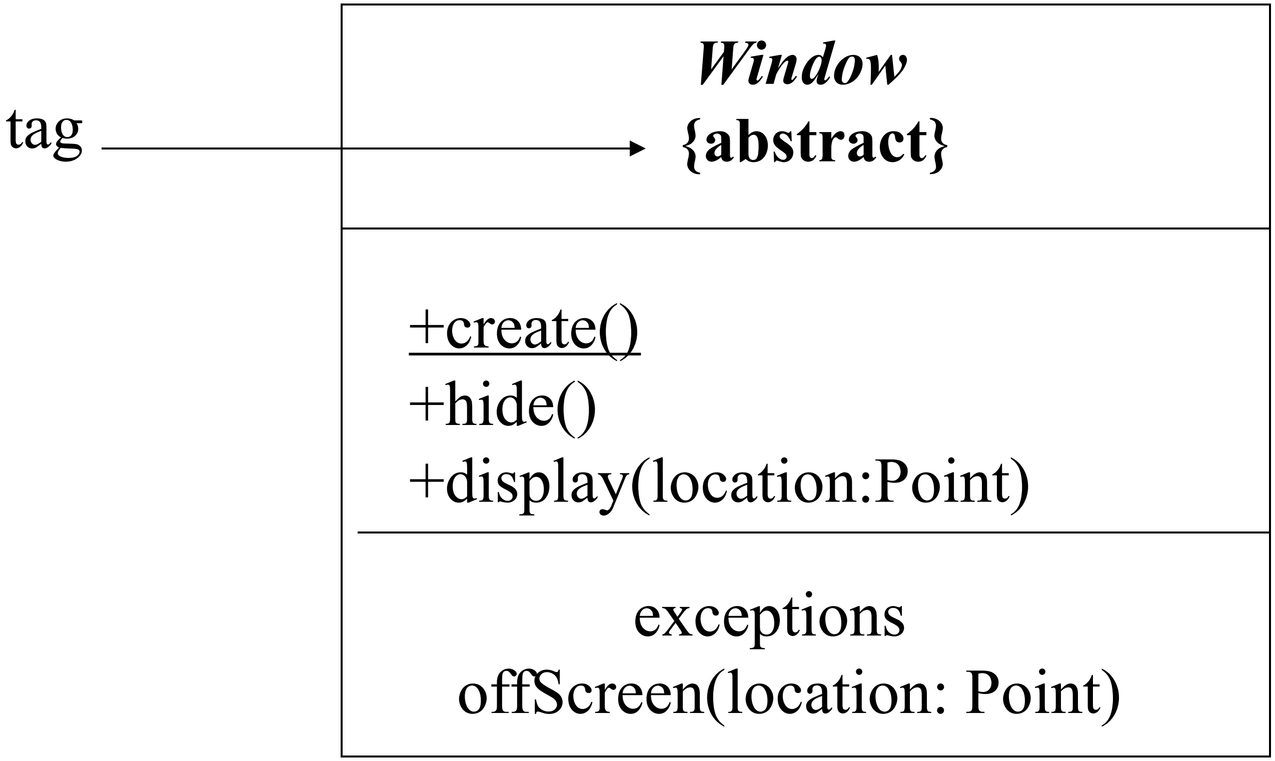
Window

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

+create()
+hide()
+display(location:Point)

Additional UML Stuff

It’s “legal” to add additional compartments.
Abstract classes get special treatment.



tag

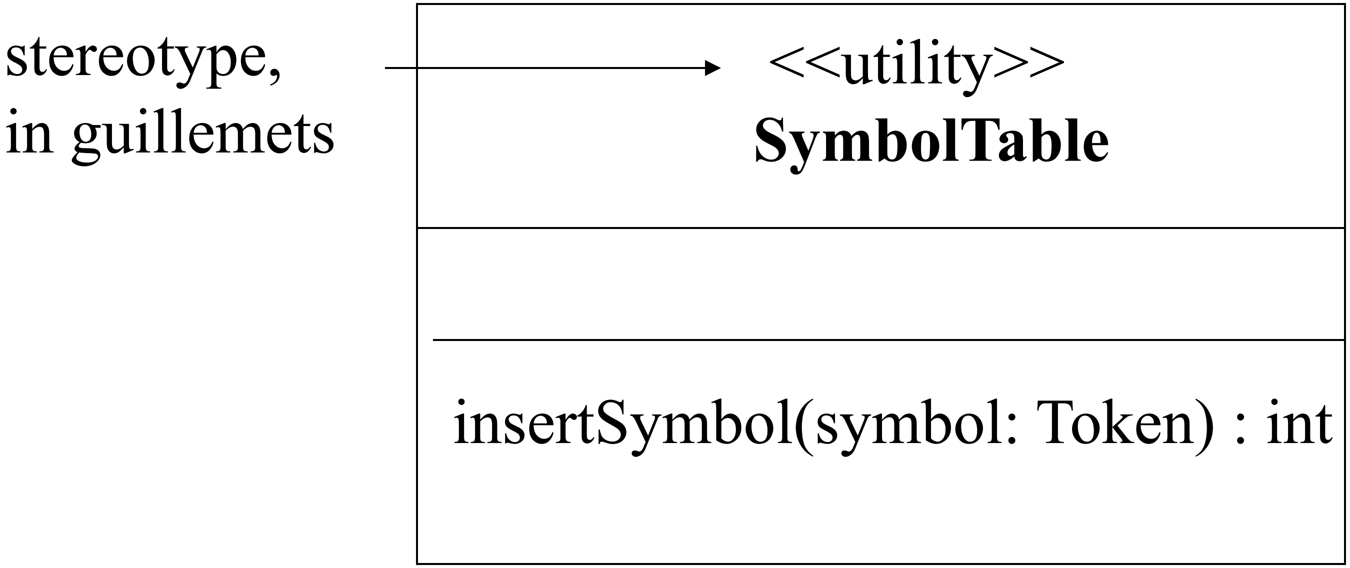
Window
{abstract}

+create()
+hide()
+display(location:Point)

exceptions
offScreen(location: Point)

Additional UML Stuff

Stereotypes represent usage distinctions, like container, abstract factory, utility, exception, etc. Together knows some common ones, you can define your own.



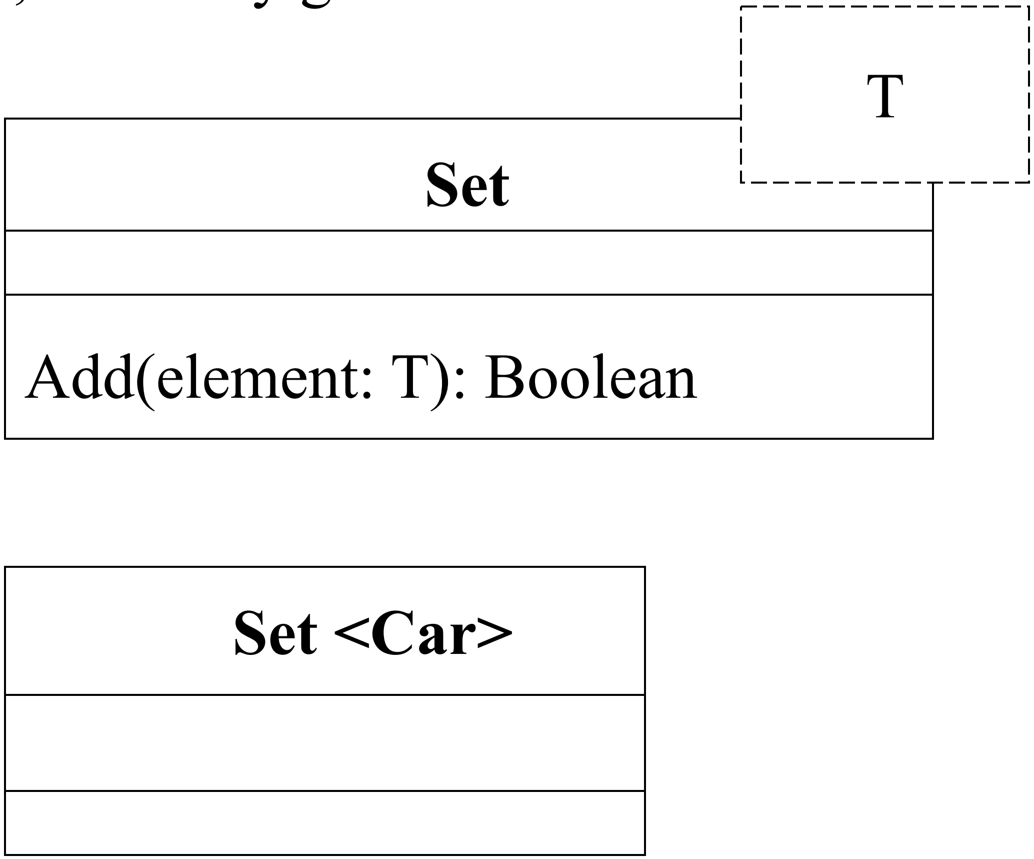
stereotype,
in guillemets

<<utility>>
SymbolTable

insertSymbol(symbol: Token) : int

Additional UML Stuff

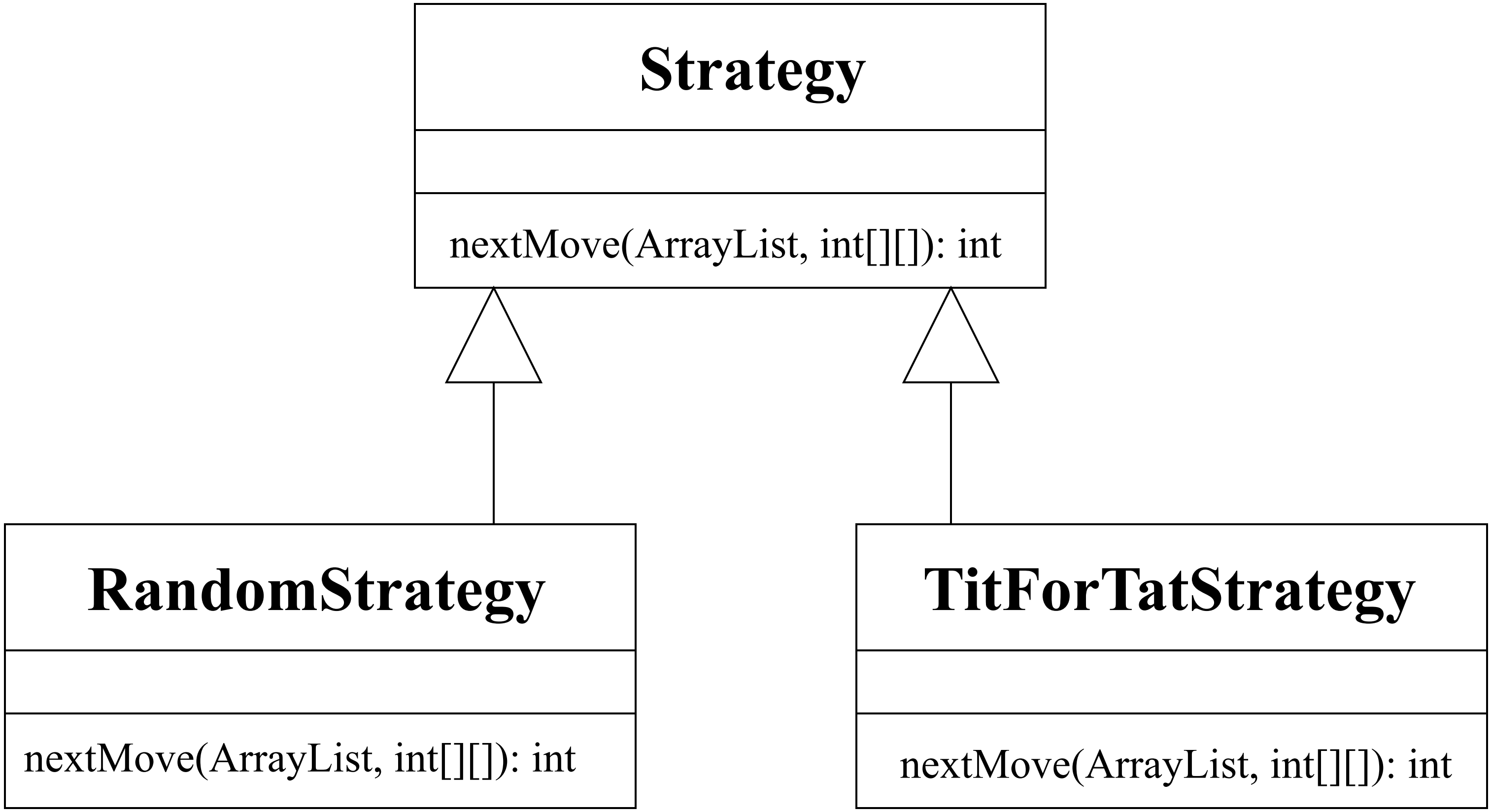
Parameterized classes get special treatment too. Important in C++, Java may get them sometime.



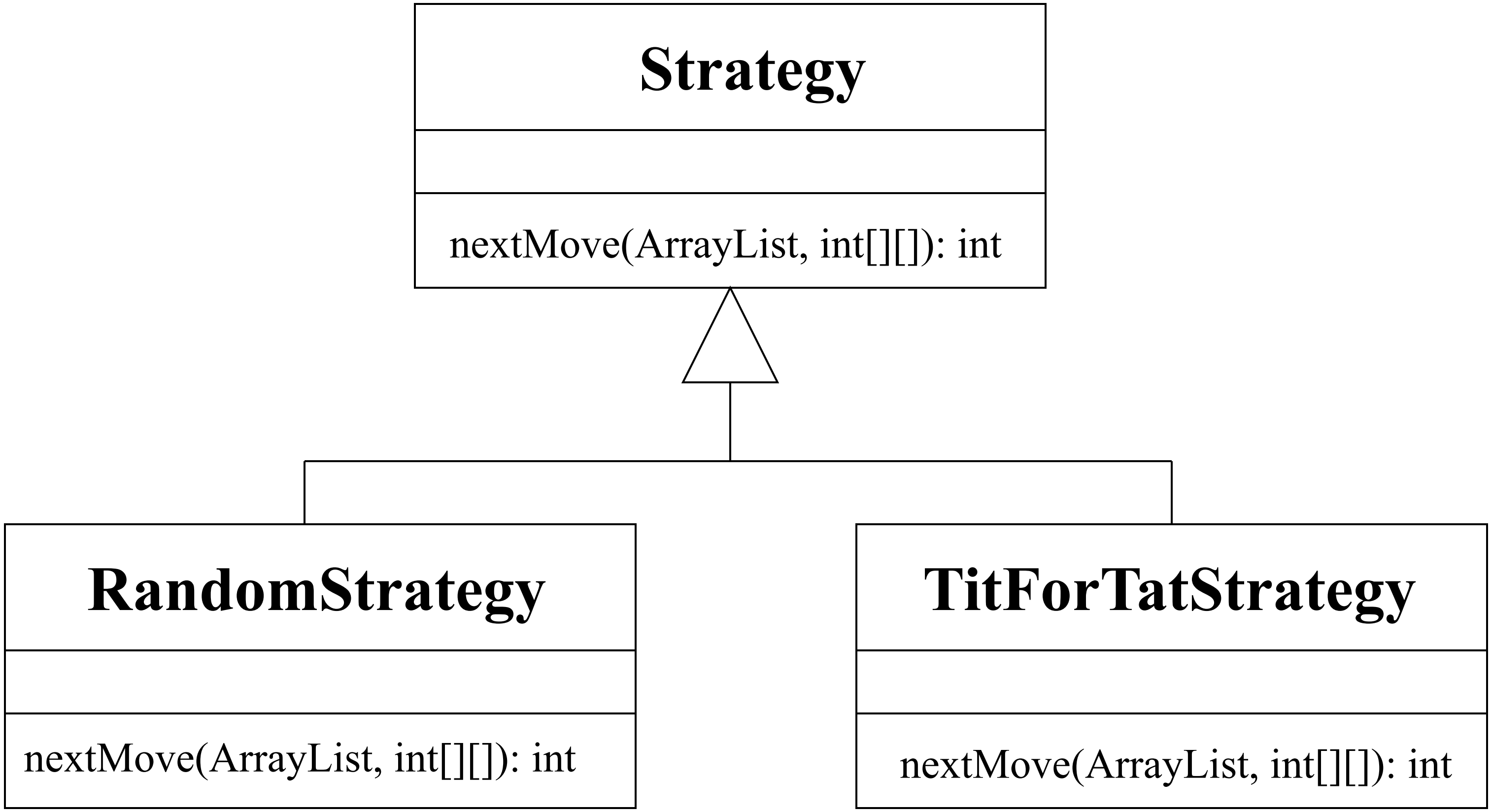
Class Diagrams

- Classes don’t operate in a vacuum; there are relationships:
  - Inheritance
  - Association
  - Whole/part
    - Composition
    - Aggregation

Single Inheritance


Strategy
nextMove(ArrayList, int[][]): int
RandomStrategy
nextMove(ArrayList, int[][]): int
TitForTatStrategy
nextMove(ArrayList, int[][]): int

Alternate Notation



Strategy

nextMove(ArrayList, int[][]): int

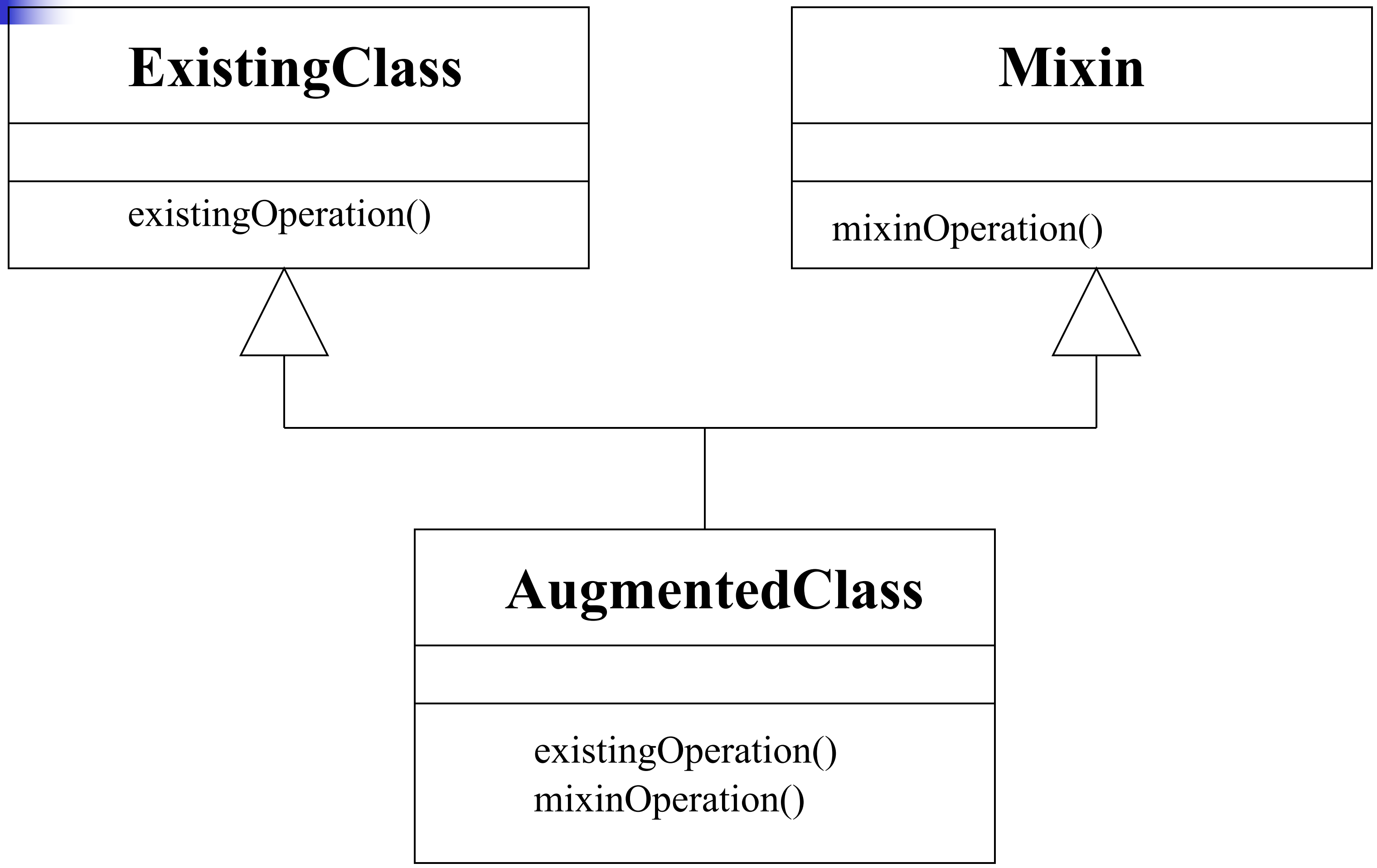
RandomStrategy

nextMove(ArrayList, int[][]): int

TitForTatStrategy

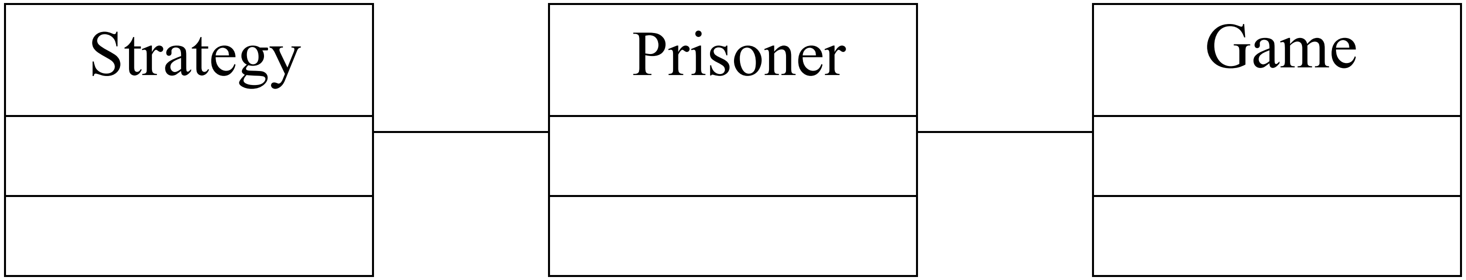
nextMove(ArrayList, int[][]): int

Multiple Inheritance



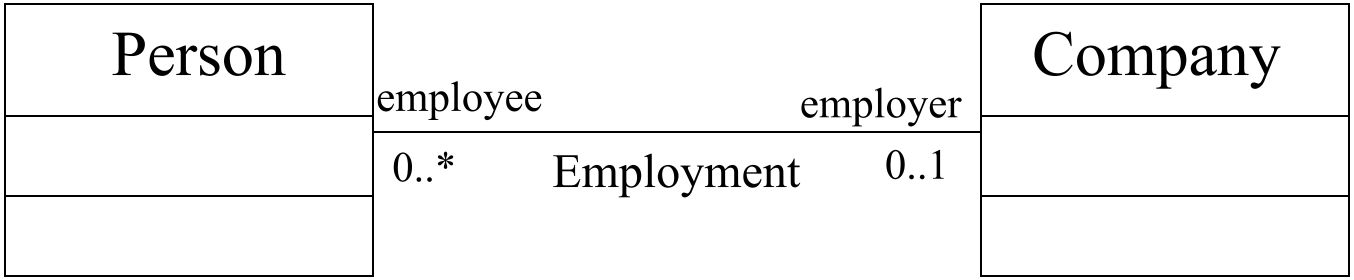
Associations

- Like the “R” in “ERD”
- A *link* couples an instance of one class with an instance of the other.
- An *association* is a collection of links.
- Diagram doesn’t show *how* the association is accomplished.



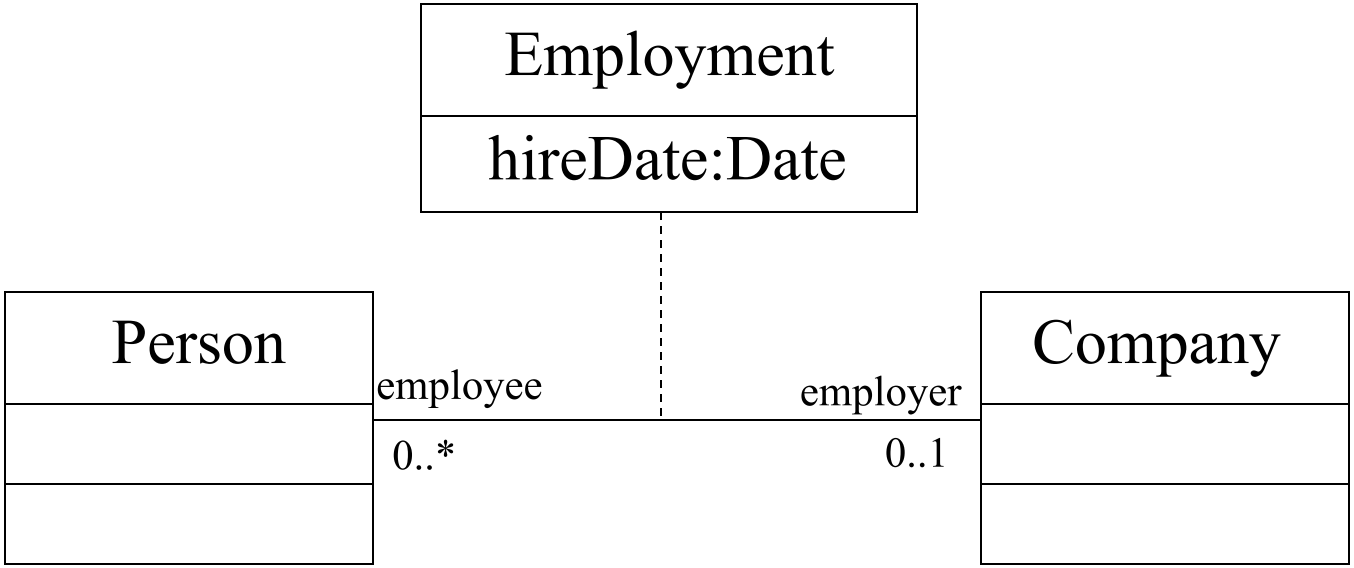
Association Names and Roles

- Just as for “R”s in “ERD”s, we can add names, roles and cardinalities.
- Sometimes this is hard: Can’t find the right name? Don’t know the numbers?
- Could an association be a class?



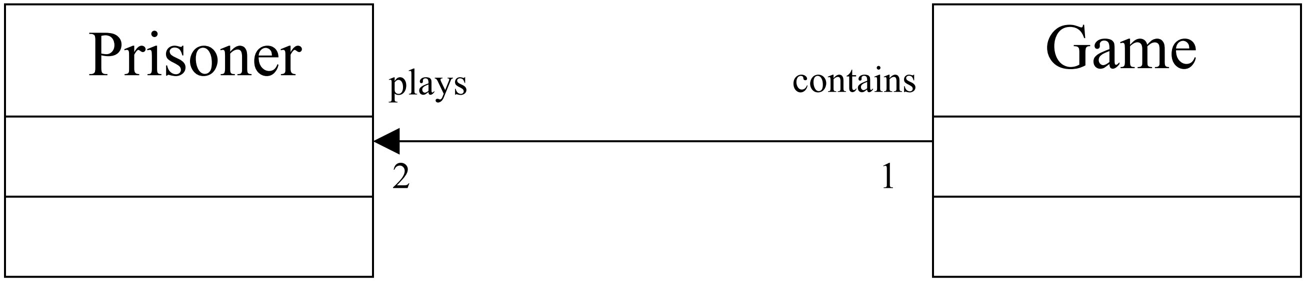
Association as Class

- An association might have properties and methods. So, make it a class (think bridge entity).



Navigability

- Plain old associations are just right at the analysis stage.
- But when designing, knowing “how to get there from here” is important.

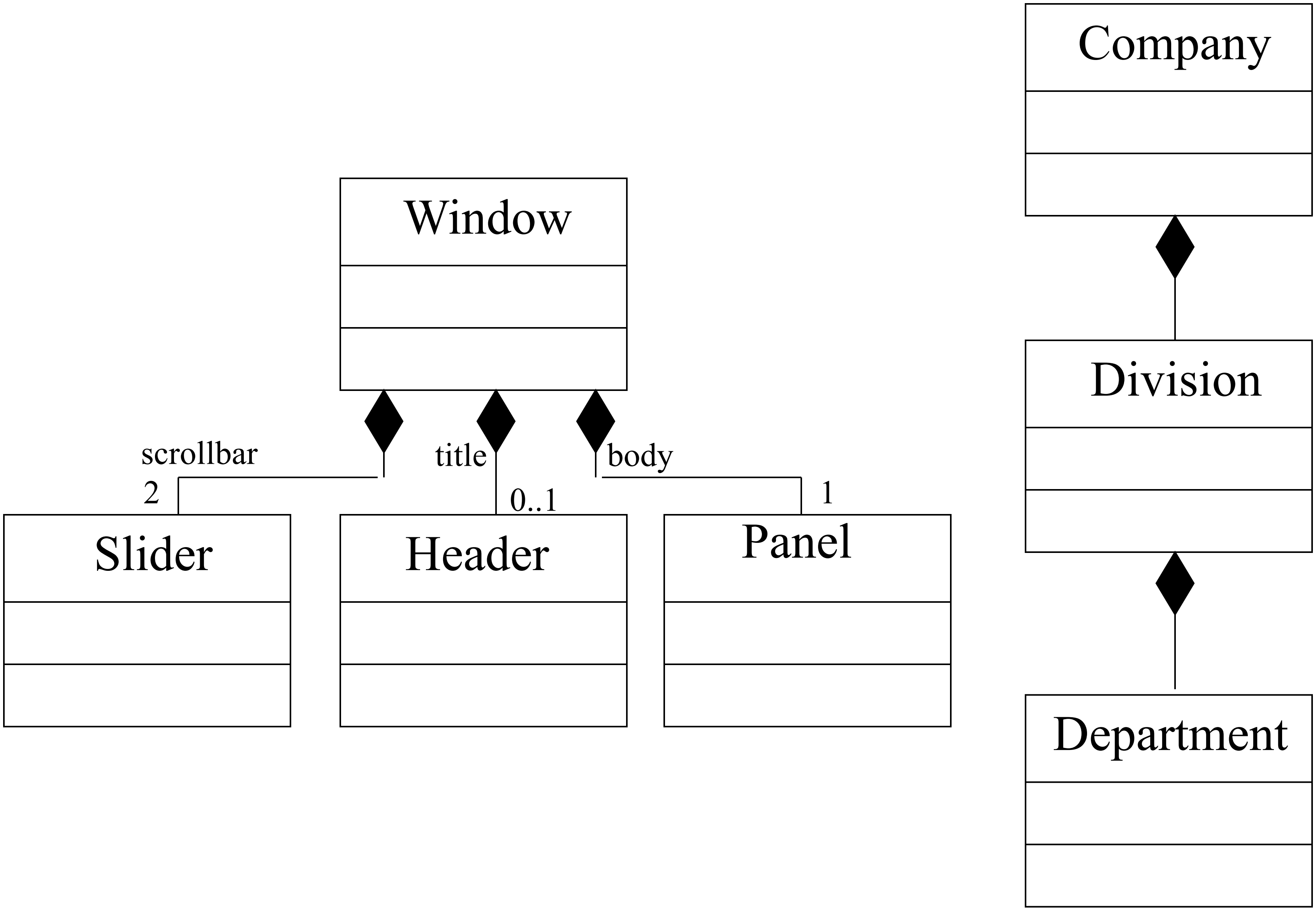


Specialized Associations

- Composition and aggregation
- Why are these special?
- Why just two?
- Can’t this be done with cardinalities?
- Isn’t this part of design?

Composition

- An email message is composed of header and paragraphs; paragraphs are composed of sentences; etc.
- The composite object doesn’t exist without its components.
- At any time, a specific component can only be part of one composite object.
- Components will likely be of mixed types.


Company
Window
Division
scrollbar
title
body
2
0..1
1
Slider
Header
Panel
Department

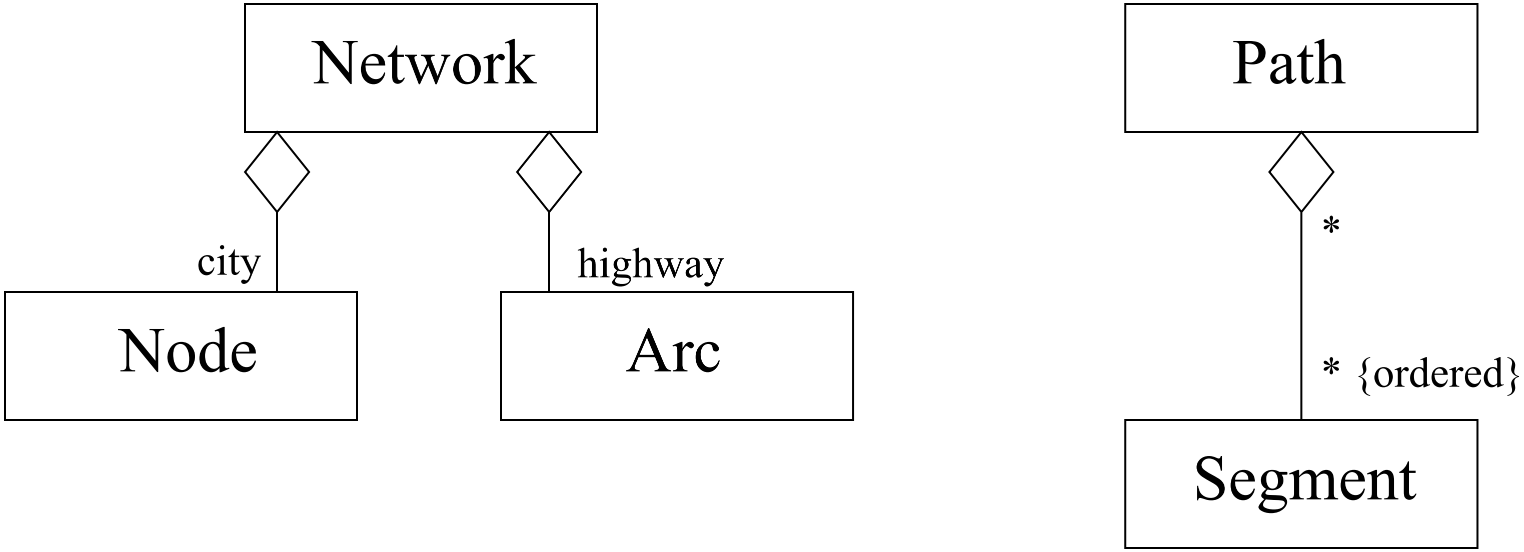
More on Composition

- The association line itself typically has no name (it wouldn’t add much).
- Role names are important, though.
- Should we have two composition lines for the two (horizontal and vertical) scrollbars?
- Use navigability arrowheads as before. Most often, from composite to components, but consider
  - How often and fast must navigation be done?
  - Does a component have a life independent of the composite?

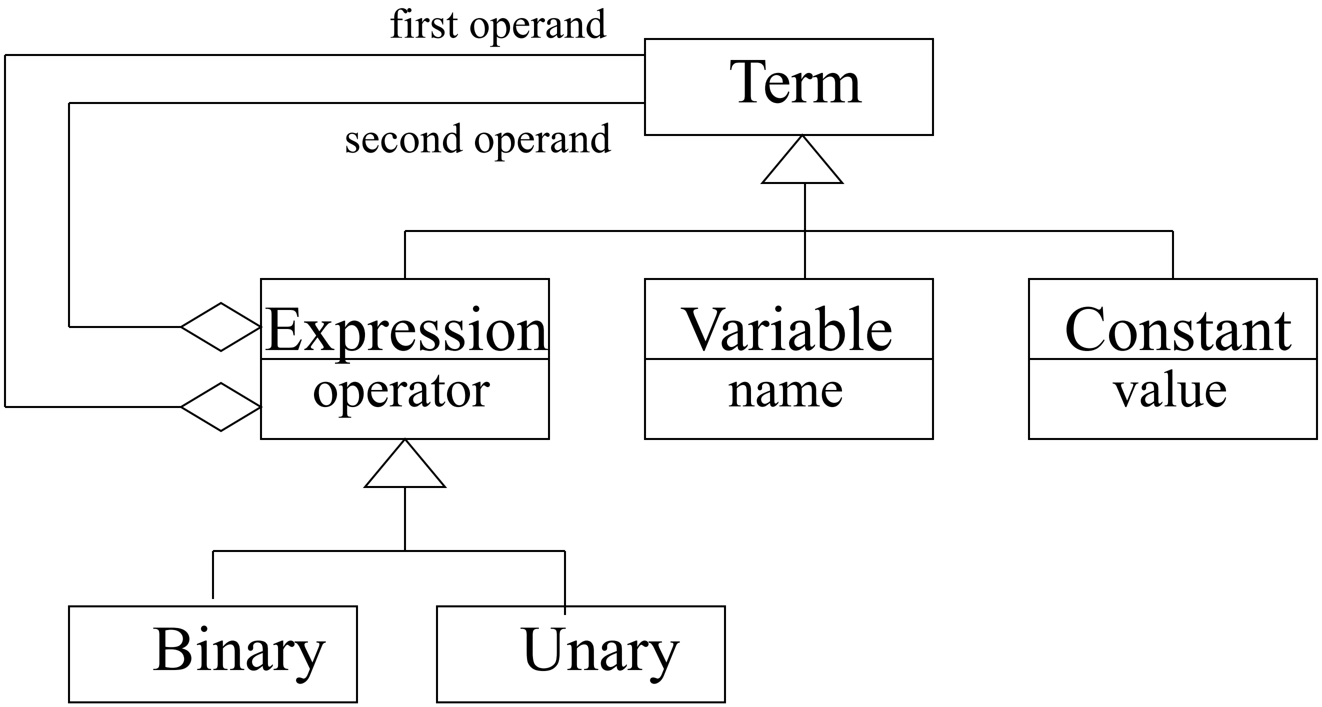
Aggregation

- A list contains items; a country contains states, a book has chapters, etc.
- The association is called *aggregation*
- The whole is called the *aggregate*
- The part is called the *constituent*
- Distinguishing features:
  - The aggregate may potentially exist without some or all of its constituents.
  - At any time, an object may be part of more than one aggregate.
  - Constituents tend to be of the same type (class).

Aggregation Notation



Example: Arithmetic Expressions



Jim Rumbaugh on Aggregation

- Would you use the phrase *part of*?
- Are some operations on the whole automatically applied to its parts?
- Are some attribute values propagated from the whole to some or all of its parts?
- Is there an intrinsic asymmetry, where one class is subordinate to the other?

Refining Association

- Composition:
  - Composite doesn’t exist without its components
  - A component can be part of only one composite
  - Components likely to be of mixed types.
- Aggregation:
  - Aggregate may exist without constituents
  - Constituent may be part of more than one aggregate
  - Constituents usually of the same class

The Three Amigos

- Aggregation conveys the thought that the aggregate is the sum of its parts. The only real semantics that it adds to association is the constraint that chains of aggregate links may nor form cycles…In spite of the few semantics attached to aggregation, everybody thinks it is necessary (for different reasons). Think of it as a modeling placebo.

The Three Amigos

- A composite is an aggregation association with the additional constraints that an object may be part of only one composite at a time, and that the composite object has sole responsibility for all of its parts.

Class Diagram Examples

- A family tree
- A computer program for playing cards.
- A directed graph.

Characterize the hierarchical relationship(s) implied by the following descriptions.

- The relationships may be "is-a" (inheritance) or "has-a" (composition).
- A prose explanation is sufficient (i.e., no class diagrams are required).

Questions

- A baseball team consists of a manager and 25 players. Each player normally plays a single defensive position (e.g., pitcher, catcher), but some players are utility players who can serve in a range of positions.

Questions

- The main window for a library book browser has a title bar and three subpanes, one for selecting book categories, another for choosing particular books within a category, and a third for displaying an abstract of a book. Each subpane has its own scroll bar.

Questions

- A heating control system has three separate thermostats. Two of the thermostats are basic thermometers that must be adjusted manually, but the third can be adjusted either manually or by setting a timer.

Questions

- The ATM provides two types of transaction, withdrawals and deposits. For every transaction, the user specifies a reference account (for deposit to or withdrawal from) and a dollar amount. Withdrawals can either be made in cash to the user or as a transfer to another account. For cash withdrawals, the user chooses the denominations of the payment; for transfers, the user specifies a target account. Recently, the ATM has begun offering a transfer+cash option, in which the user can carry out both sorts of withdrawals in a single transaction.

Questions

- Suppose you are designing a system to simplify the scheduling and scoring of judged athletic competitions such as gymnastics, diving and figure skating. There are several events and competitors. Each competitor may enter several events and each event has many competitors. Each event has five judges who subjectively rate the performance of competitors in that event. A judge rates each competitor for an event. In some cases, a judge may score more than one event. The focal points of the competition are trials. Each trial is an attempt by one competitor to turn in the best performance possible in one event. A trial is scored by the panel of judges for that event and a net score is determined.
- Determine classes and associations, and draw a class diagram in Together. Do not show navigability, but include cardinalities. For aggregate and composition associations, attach a note explaining your analysis decision.

Object-Interaction Diagrams

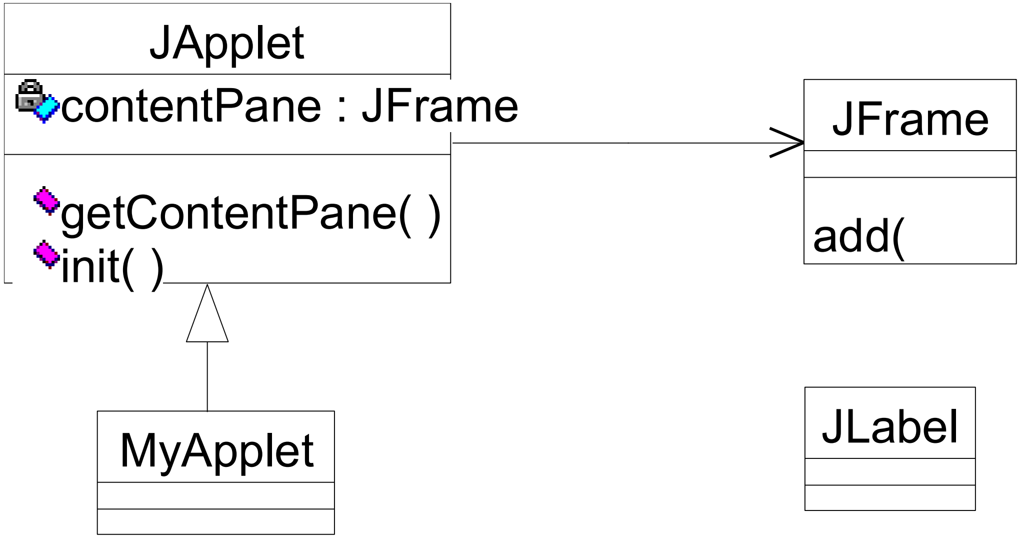
- Class diagrams represent the static structure of an O-O system.
- The dynamic (runtime) structure must be represented differently.
- We use object-interaction diagrams
  - Collaboration diagrams
  - Sequence diagrams

Object-Interaction Diagrams

- Typically, interaction diagrams are written for each use case.
- If the use case is complicated, or has many alternate courses, several diagrams can be drawn.
- The idea is to show message passing arrangements between objects.
- Requires that objects be determined!

A Class Diagram

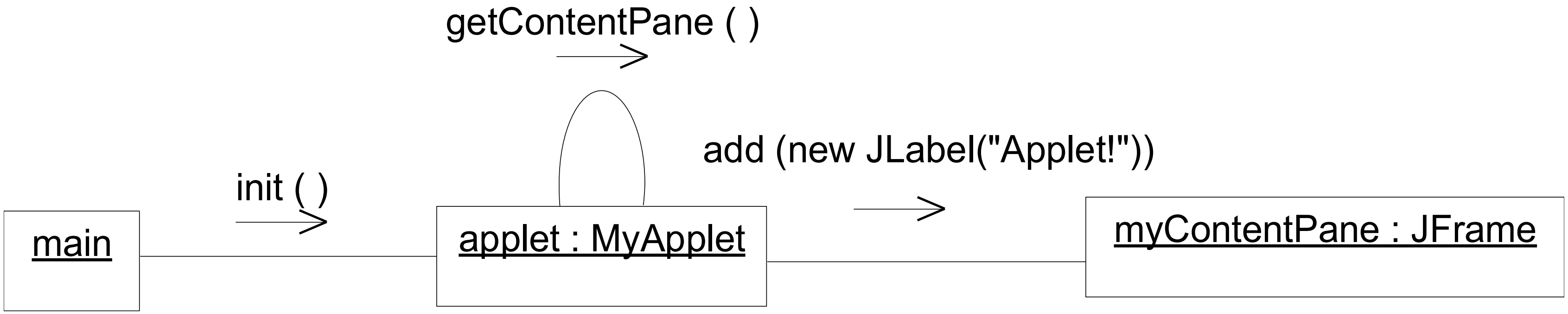
- Some classes:



```
public class MyApplet extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("Applet!);
  }
}
```

Collaboration Diagram

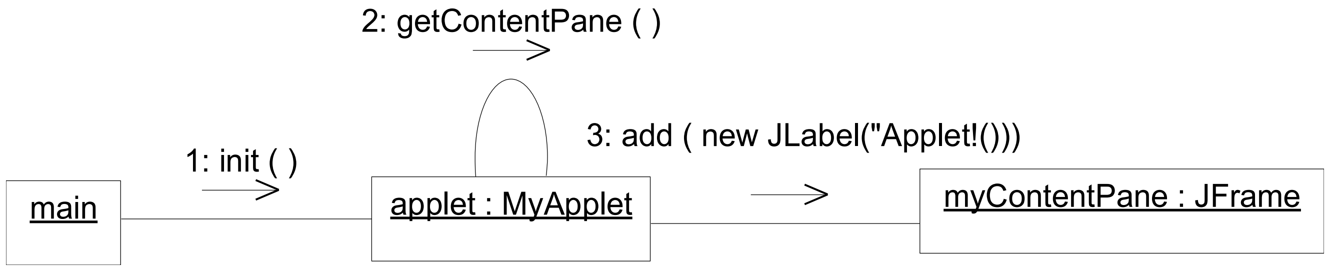
- Boxes are objects, arrows are messages.



- Lines joining objects are links; **they should exist** (as associations) in the class diagram.

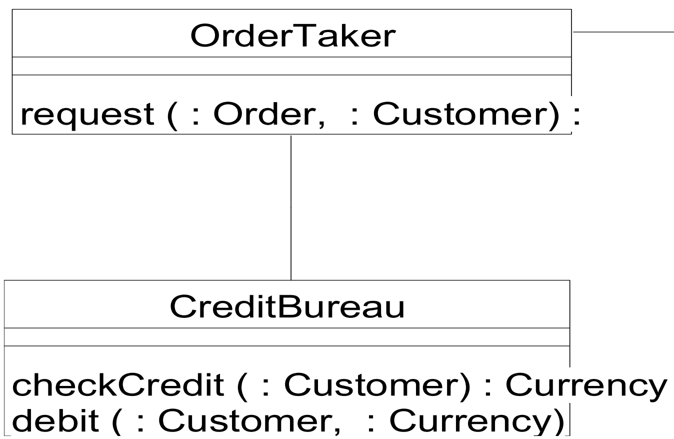
Those Little Numbers

- Indicate the sequence of messages

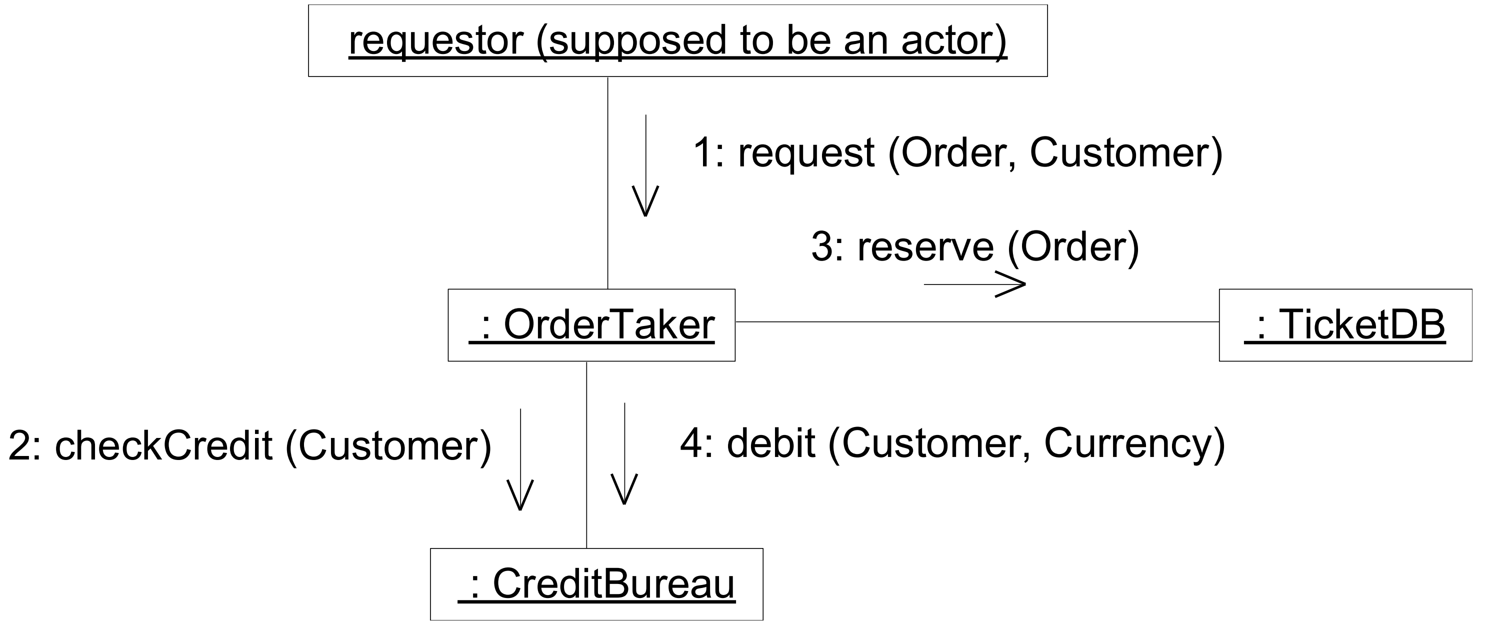


- These can be a pain; it may take several tries to get this right!
- But automatic tool conversion to sequence diagrams requires them…

Another Example

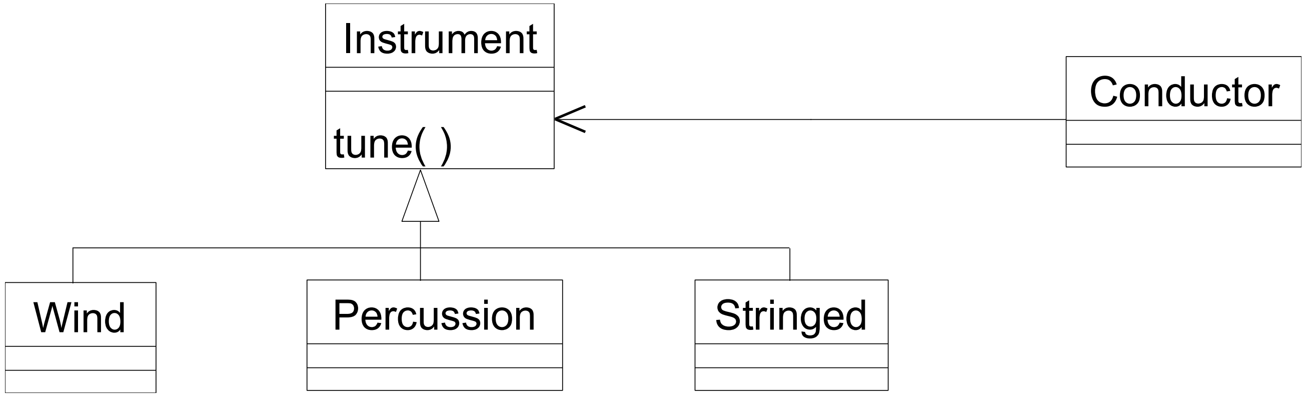

OrderTaker
request (: Order, : Customer) :
TicketDB
reserve (anOrder : Order) : Cu
CreditBureau
checkCredit (: Customer) : Currency
debit (: Customer, : Currency)

And the Collaboration Diagram


requestor (supposed to be an actor)
1: request (Order, Customer)
3: reserve (Order)
: OrderTaker
: TicketDB
2: checkCredit (Customer)
4: debit (Customer, Currency)
: CreditBureau

Polymorphism

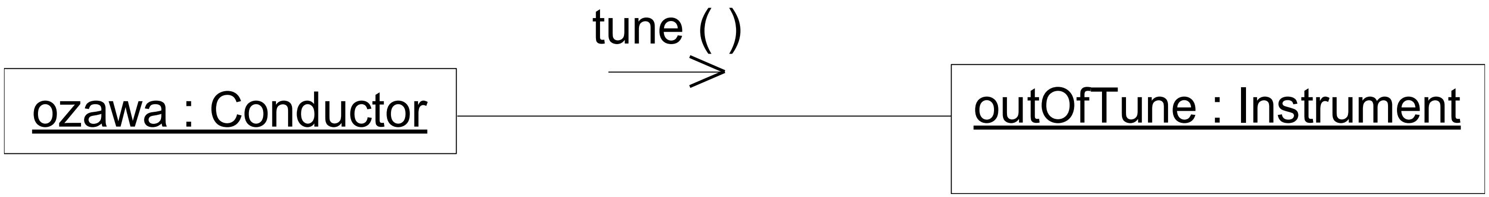
- What to do if we don't know the exact target of a message?



```
Instrument[] orchestra = new Instrument[5];
tuneAll(orchestra);
```

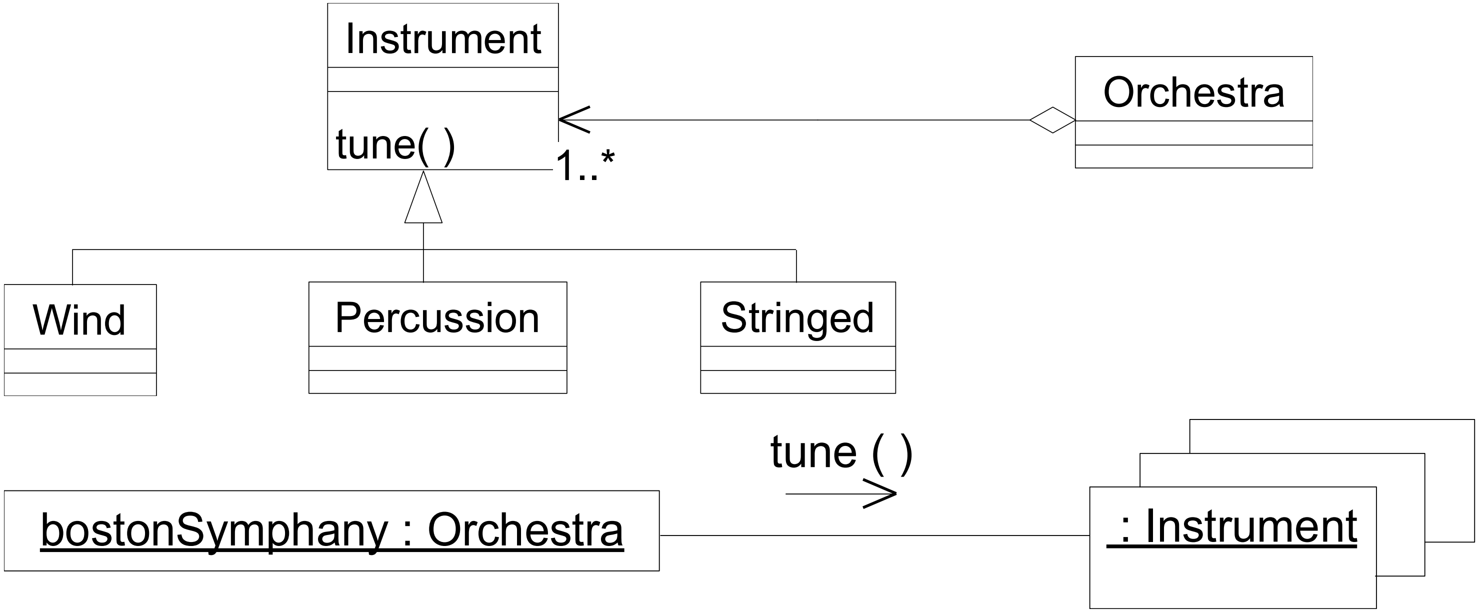
Polymorphism (cont.)

- Send the message to the highest class containing all possible objects.



- Page-Jones recommends putting the class name in parentheses.

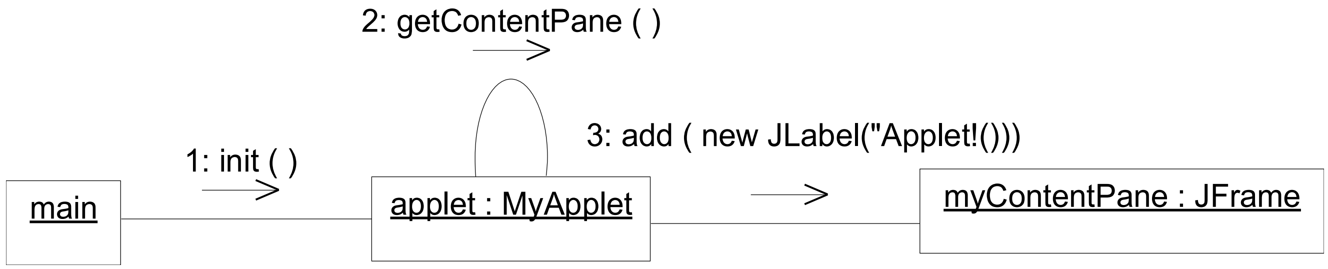
Iterated Messages



- Probably should be able to add aggregation diamond to the collaboration diagram.
- If you’re using an iterator, you may or may not show calls to first() and last().

Self-Messages

- No need to refer to "self" ("this") here:

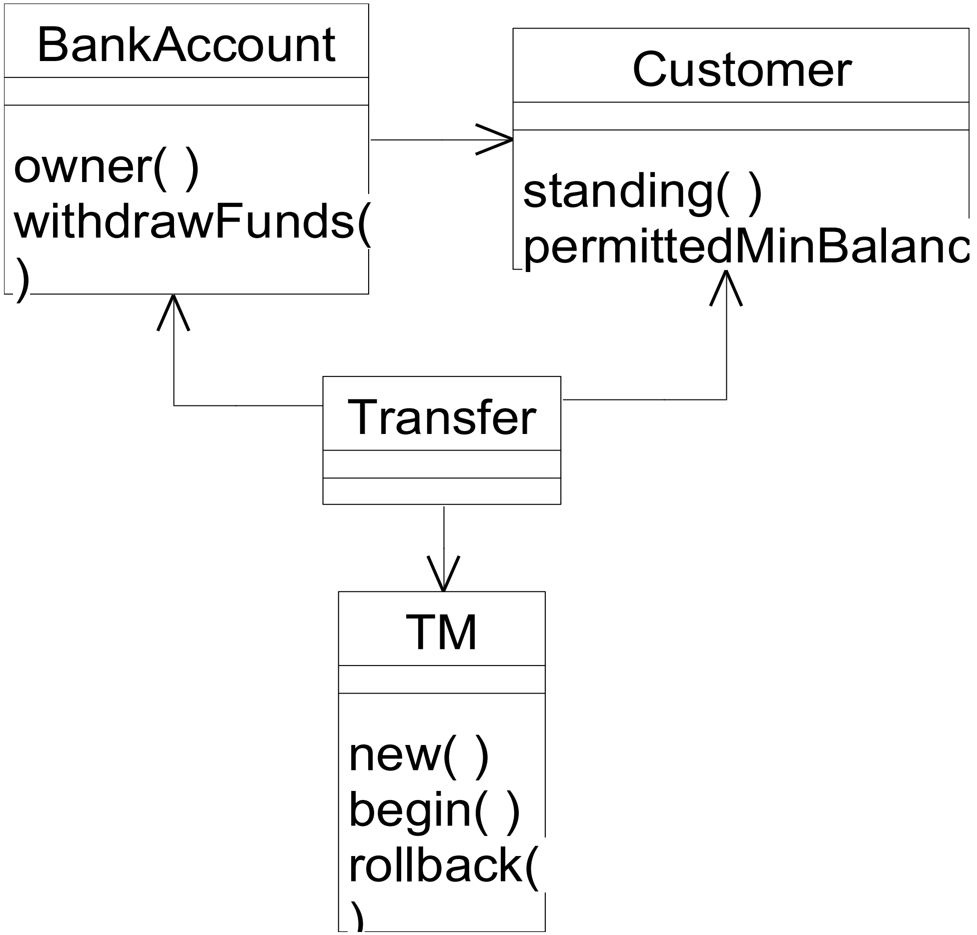


- However, if you do need to pass a handle in a message, show it explicitly.

Sequence Diagrams

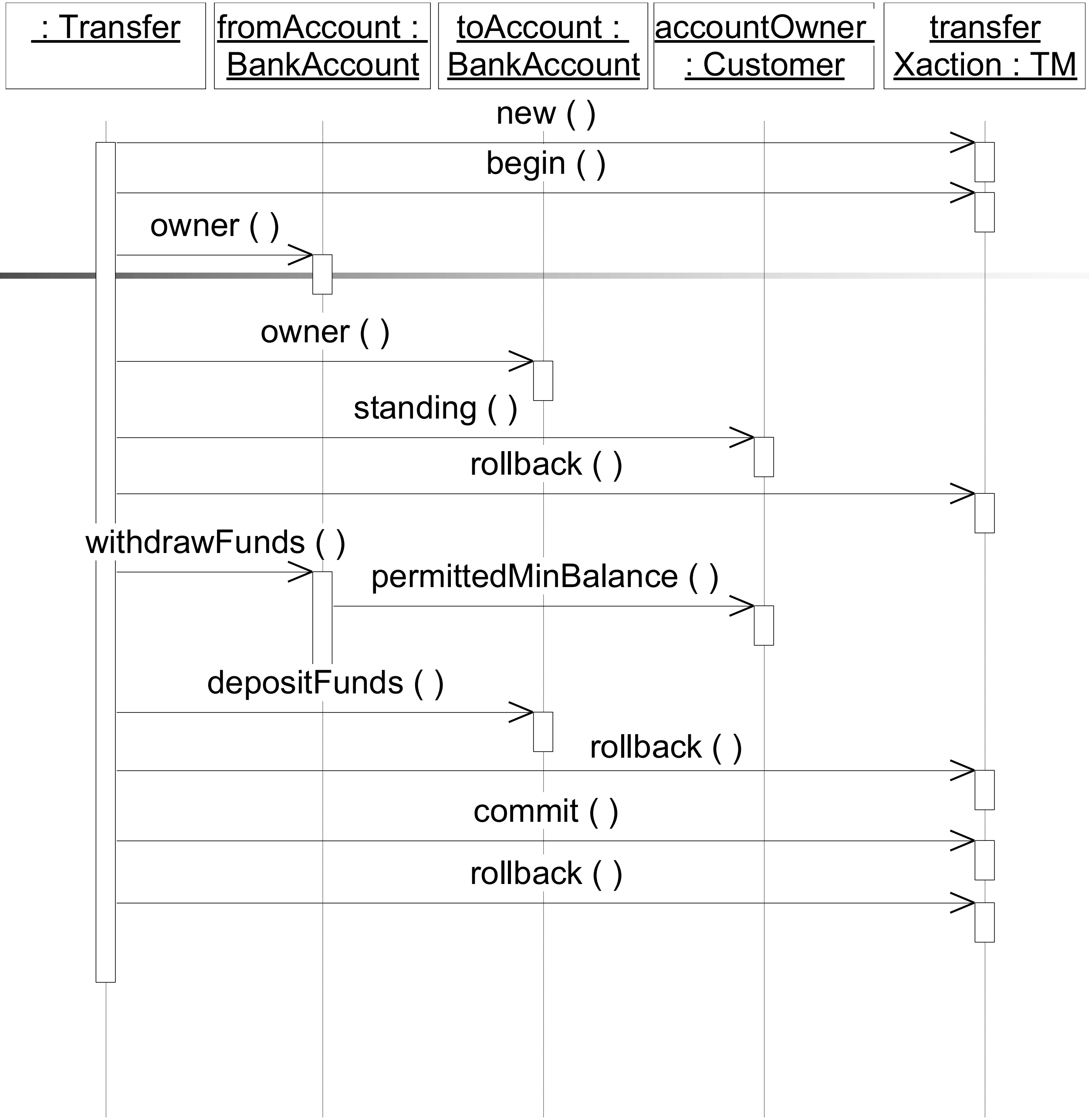
- Just a different format for “collaboration diagrams with numbers.”
- The temporal sequence is much clearer.
- Time “moves” downward.

Class Diagram



DESCRIPTION of :Transfer.makeTransfer : Boolean

```
create a new transfer transaction
begin transaction
establish fromAccount
establish toAccount
if the two owners are the same customer, who also
  has good customer standing,
then fromAccount.withdrawFunds(amt, out withdrawal OK)
else transferXaction.rollback()
endif
if withdrawalOK
then toAccount.depositFunds(amt, out depositOK)
else transferXaction.rollback()
endif
if depositOK
then transferXaction.commit()
           return true
else transferXaction.rollback()
           return false
endif
```



: Transfer

fromAccount : BankAccount

toAccount : BankAccount

accountOwner : Customer

transfer Xaction : TM

new ( )

begin ( )

owner ( )

owner ( )

standing ( )

rollback ( )

withdrawFunds ( )

permittedMinBalance ( )

depositFunds ( )

rollback ( )

commit ( )

rollback ( )