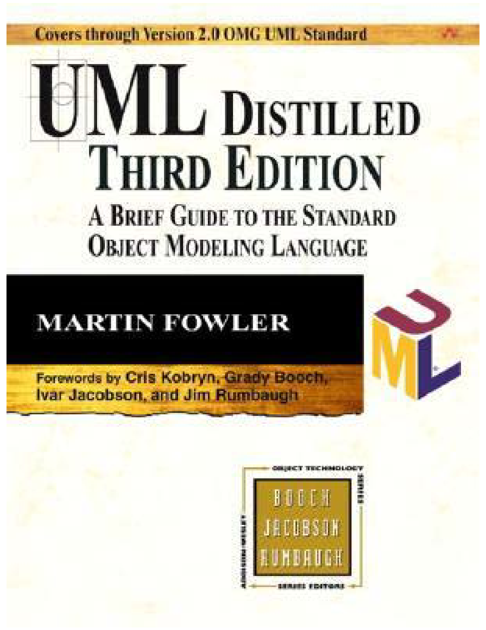


Covers through Version 2.0 OMG UML Standard

UML Distilled
Third Edition
A Brief Guide to the Standard Object Modeling Language

Martin Fowler

Forewords by Cris Kobryn, Grady Booch, Ivar Jacobson, and Jim Rumbaugh

Object Technology Series
Booch
Jacobson
Rumbaugh
Series Editors
Addison-Wesley

Design and UML Class Diagrams

Suggested reading:

Practical UML: A hands on introduction for developers
http://dn.codegear.com/article/31863

UML Distilled Ch. 3, by M. Fowler

Big questions

- What is UML?
  - Why should I bother? Do people really use UML?

- What is a UML class diagram?
  - What kind of information goes into it?
  - How do I create it?
  - When should I create it?

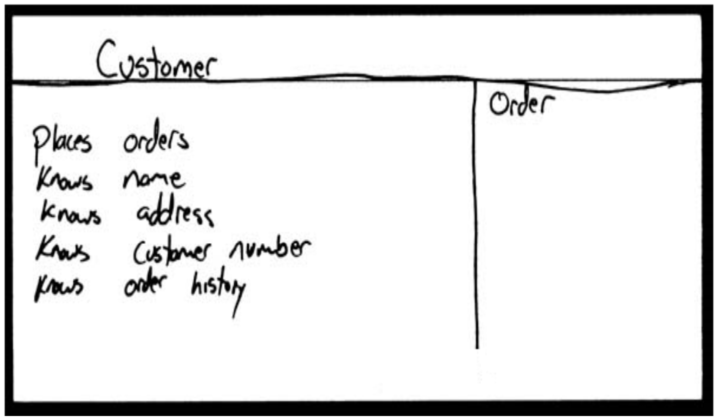
Design phase

- **design**: specifying the structure of how a software system will be written and function, without actually writing the complete implementation

- a transition from "what" the system must do, to "how" the system will do it
  - What classes will we need to implement a system that meets our requirements?
  - What fields and methods will each class have?
  - How will the classes interact with each other?

How do we design classes?

- class identification from project spec / requirements
  - nouns are potential classes, objects, fields
  - verbs are potential methods or responsibilities of a class
- CRC card exercises
  - write down classes' names on index cards
  - next to each class, list the following:
    - **responsibilities**: problems to be solved; short verb phrases
    - **collaborators**: other classes that are sent messages by this class (asymmetric)
- UML diagrams
  - class diagrams (today)
  - sequence diagrams
  - ...



UML

In an effort to promote Object Oriented designs, three leading object oriented programming researchers joined ranks to combine their languages:

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

and come up with an industry standard [mid 1990's].

UML – Unified Modeling Language

- The result is large (as one might expect)
  - Union of all Modeling Languages
    - Use case diagrams
    - Class diagrams
    - Object diagrams
    - Sequence diagrams
    - Collaboration diagrams
    - Statechart diagrams
    - Activity diagrams
    - Component diagrams
    - Deployment diagrams
    - ….
  - But it’s a nice standard that has been embraced by the industry.

Introduction to UML

- UML: pictures of an OO system
  - programming languages are not abstract enough for OO design
  - UML is an open standard; lots of companies use it

- What is legal UML?
  - a *descriptive* language: rigid formal syntax (like programming)
  - a *prescriptive* language: shaped by usage and convention
  - it's okay to omit things from UML diagrams if they aren't needed by team/supervisor/instructor

Uses for UML

- as a sketch: to communicate aspects of system
  - forward design: doing UML before coding
  - backward design: doing UML after coding as documentation
  - often done on whiteboard or paper
  - used to get rough selective ideas

- as a blueprint: a complete design to be implemented
  - sometimes done with CASE (Computer-Aided Software Engineering) tools

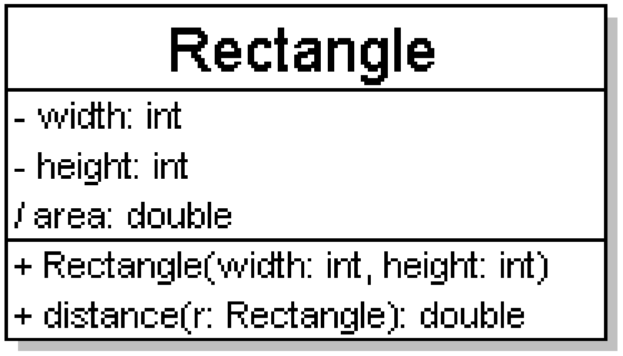
- as a programming language: with the right tools, code can be auto-generated and executed from UML
  - only good if this is faster than coding in a "real" language

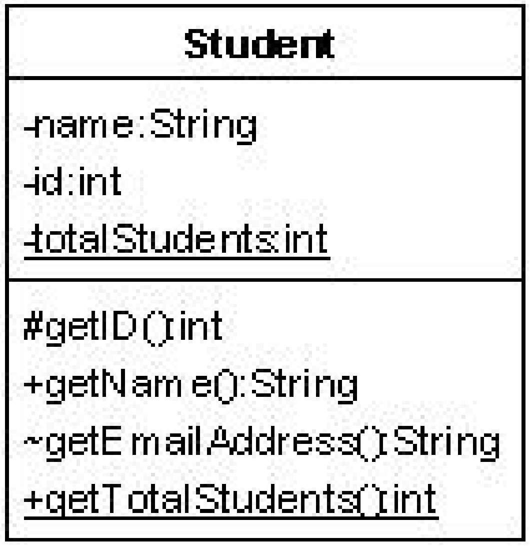
UML class diagrams

- What is a UML class diagram?
  - **UML class diagram**: a picture of
    - the classes in an OO system
    - their fields and methods
    - connections between the classes
      - that interact or inherit from each other
- What are some things that are <u>not</u> represented in a UML class diagram?
  - details of how the classes interact with each other
  - algorithmic details; how a particular behavior is implemented

Diagram of one class

- class name in top of box
  - write <<interface>> on top of interfaces' names
  - use *italics* for an *abstract class* name
- attributes (optional)
  - should include all fields of the object
- operations / methods (optional)
  - may omit trivial (get/set) methods
    - but don't omit any methods from an interface!
  - should not include inherited methods


Rectangle
- width: int
- height: int
/ area: double
+ Rectangle(width: int, height: int)
+ distance(r: Rectangle): double

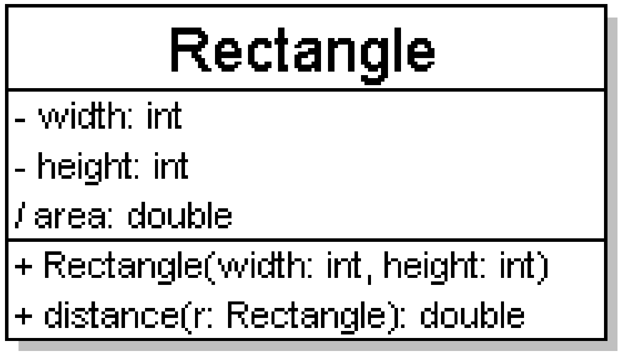


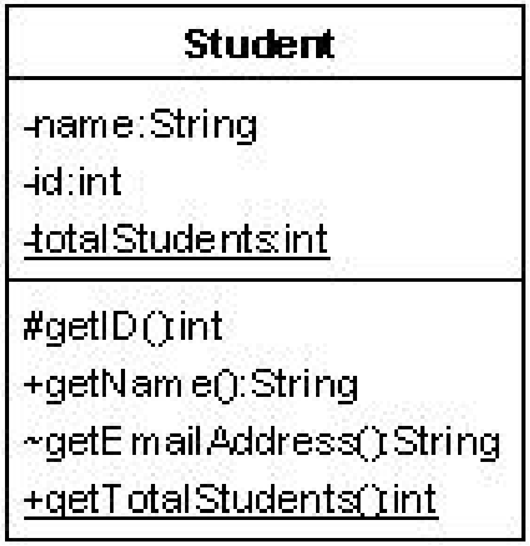
Class attributes

- attributes (fields, instance variables)
  - *visibility name* : *type* [*count*] = *default_value*
  - visibility:

+	public
#	protected
-	private
~	package (default)
/	derived

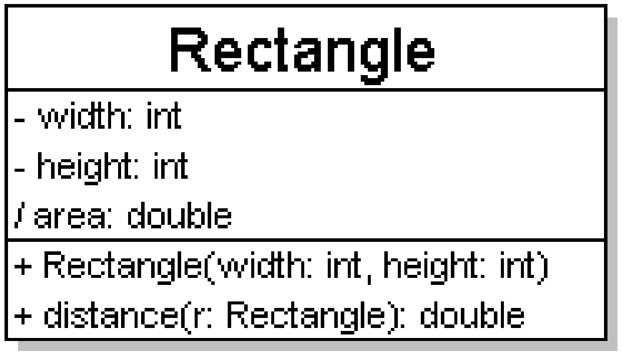
  - underline <u>static attributes</u>
  - **derived attribute**: not stored, but can be computed from other attribute values
  - attribute example:
    - balance : double = 0.00

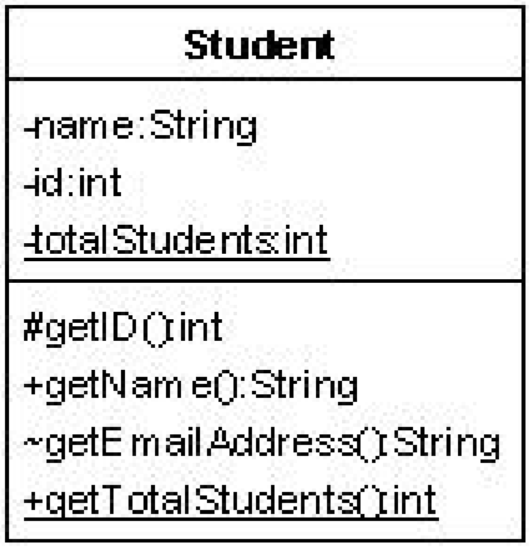

Rectangle
- width: int
- height: int
/ area: double
+ Rectangle(width: int, height: int)
+ distance(r: Rectangle): double



Class operations / methods

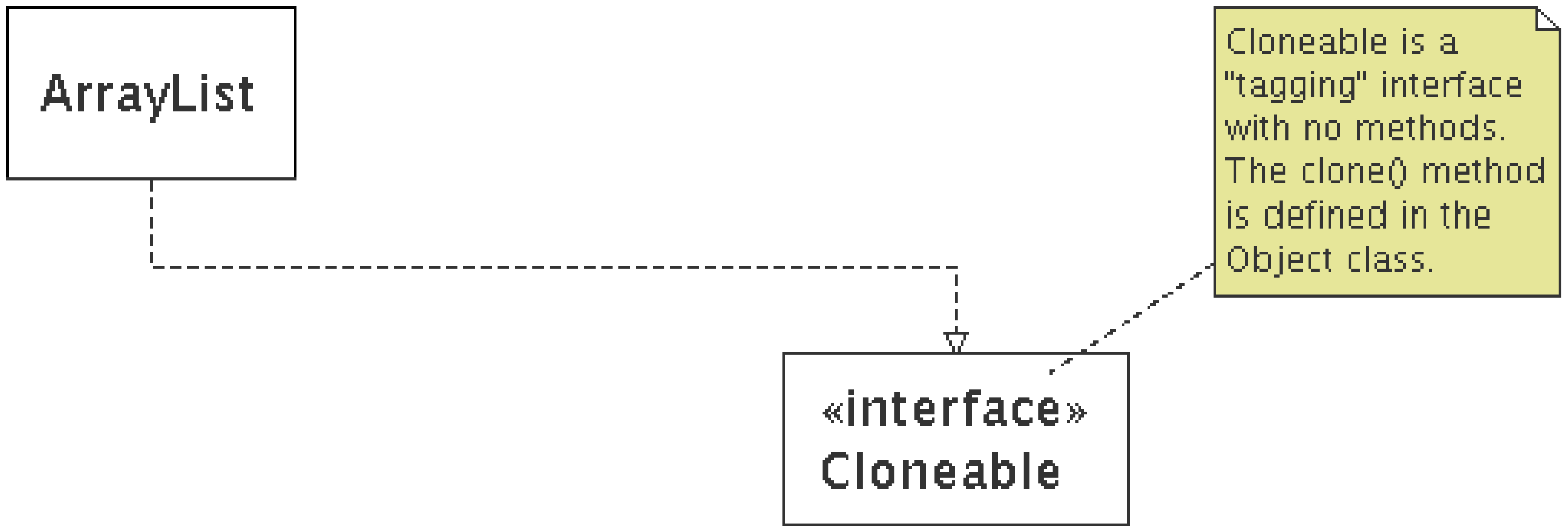
- operations / methods
  - *visibility name* (*parameters*) : *return_type*
  - visibility:
    - + public
    - # protected
    - - private
    - ~ package (default)
  - underline <u>static methods</u>
  - parameter types listed as (name: type)
  - omit *return_type* on constructors and when return type is void
  - method example:
    + distance(p1: Point, p2: Point): double


Rectangle
- width: int
- height: int
/ area: double
+ Rectangle(width: int, height: int)
+ distance(r: Rectangle): double



Comments

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


ArrayList
«interface»
Cloneable
Cloneable is a "tagging" interface with no methods. The clone() method is defined in the Object class.

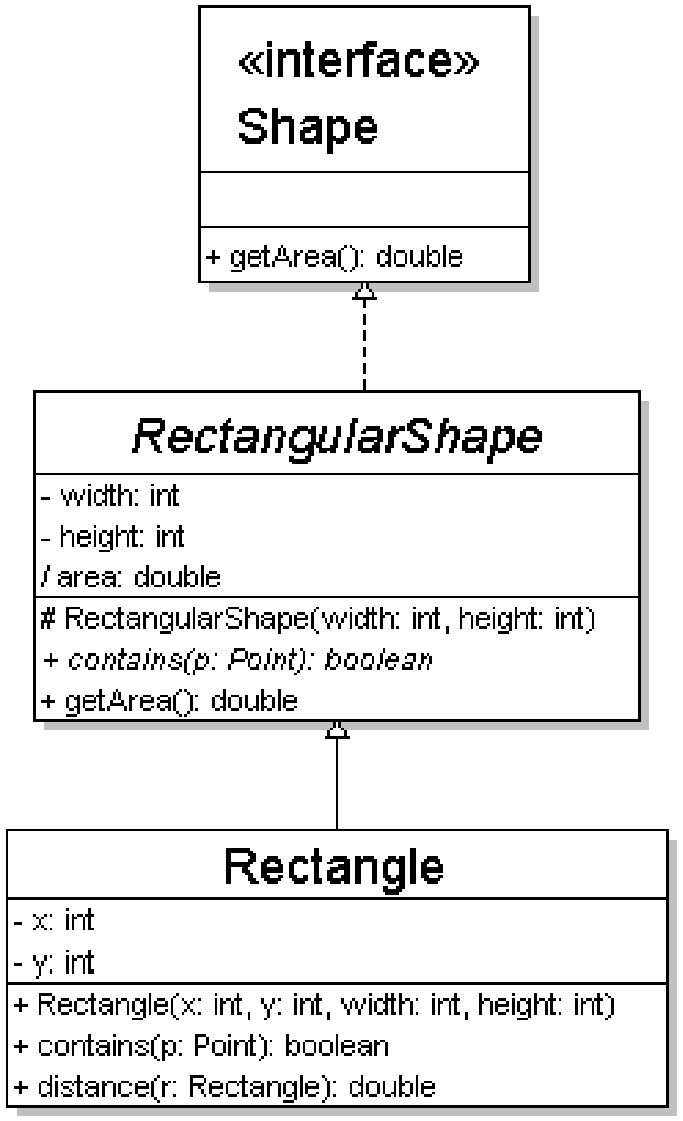
Relationships btwn. classes

- **generalization**: an inheritance relationship
  - inheritance between classes
  - interface implementation

- **association**: a usage relationship
  - dependency
  - aggregation
  - composition

Generalization relationships

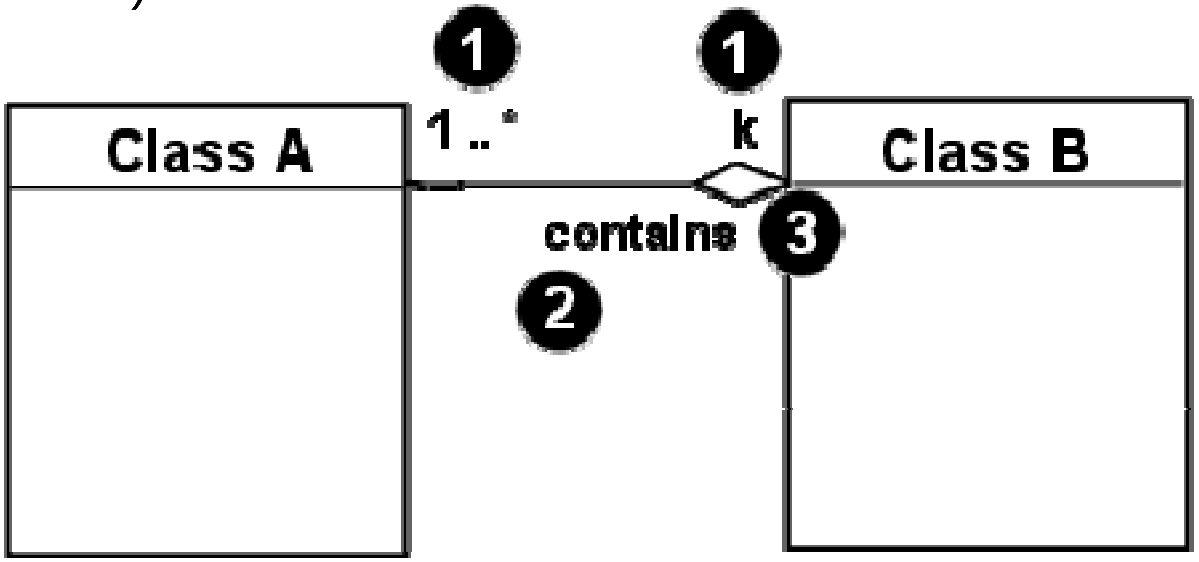
- generalization (inheritance) relationships
  - hierarchies drawn top-down with arrows pointing upward to parent
  - line/arrow styles differ, based on whether parent is a(n):
    - <u>class</u>:
      solid line, black arrow
    - <u>abstract class</u>:
      solid line, white arrow
    - <u>interface</u>:
      dashed line, white arrow
  - we often don't draw trivial / obvious generalization relationships, such as drawing the Object class as a parent



«interface»
Shape
+ getArea(): double
RectangularShape
- width: int
- height: int
/ area: double
# RectangularShape(width: int, height: int)
+ *contains(p: Point): boolean*
+ getArea(): double
Rectangle
- x: int
- y: int
+ Rectangle(x: int, y: int, width: int, height: int)
+ contains(p: Point): boolean
+ distance(r: Rectangle): double

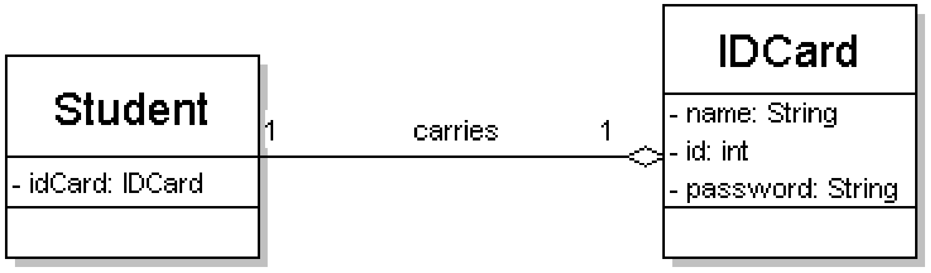
Associational relationships

- associational (usage) relationships
  1. multiplicity (how many are used)
     - * ⇒ 0, 1, or more
     - 1 ⇒ 1 exactly
     - 2..4 ⇒ between 2 and 4, inclusive
     - 3..* ⇒ 3 or more
  2. name (what relationship the objects have)
  3. navigability (direction)

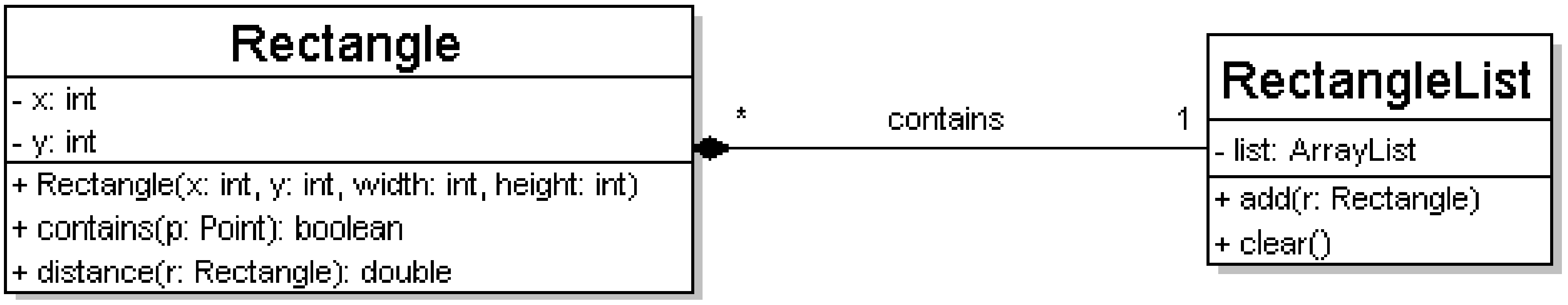


Multiplicity of associations

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

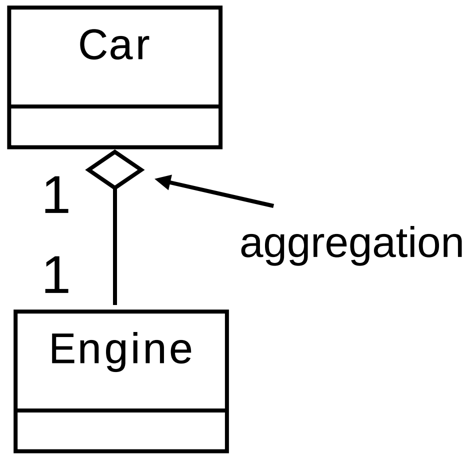


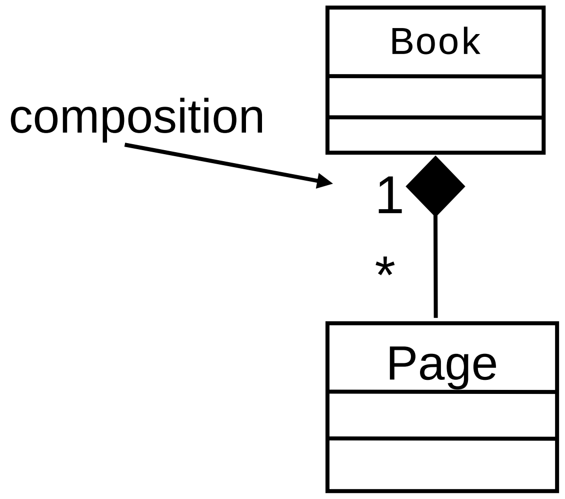
- one-to-many
  - one rectangle list can contain many rectangles

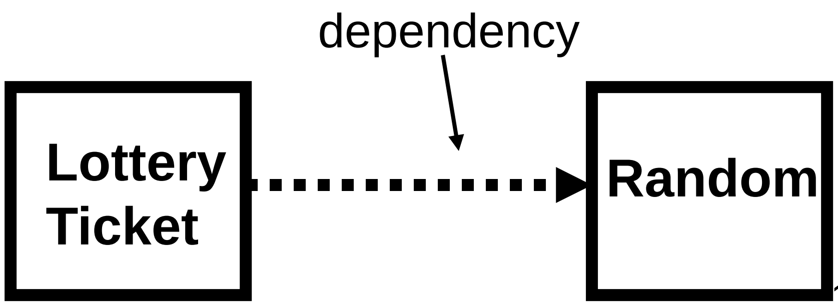

Rectangle
- x: int
- y: int
+ Rectangle(x: int, y: int, width: int, height: int)
+ contains(p: Point): boolean
+ distance(r: Rectangle): double
*
contains
1
RectangleList
- list: ArrayList
+ add(r: Rectangle)
+ clear()

Association types

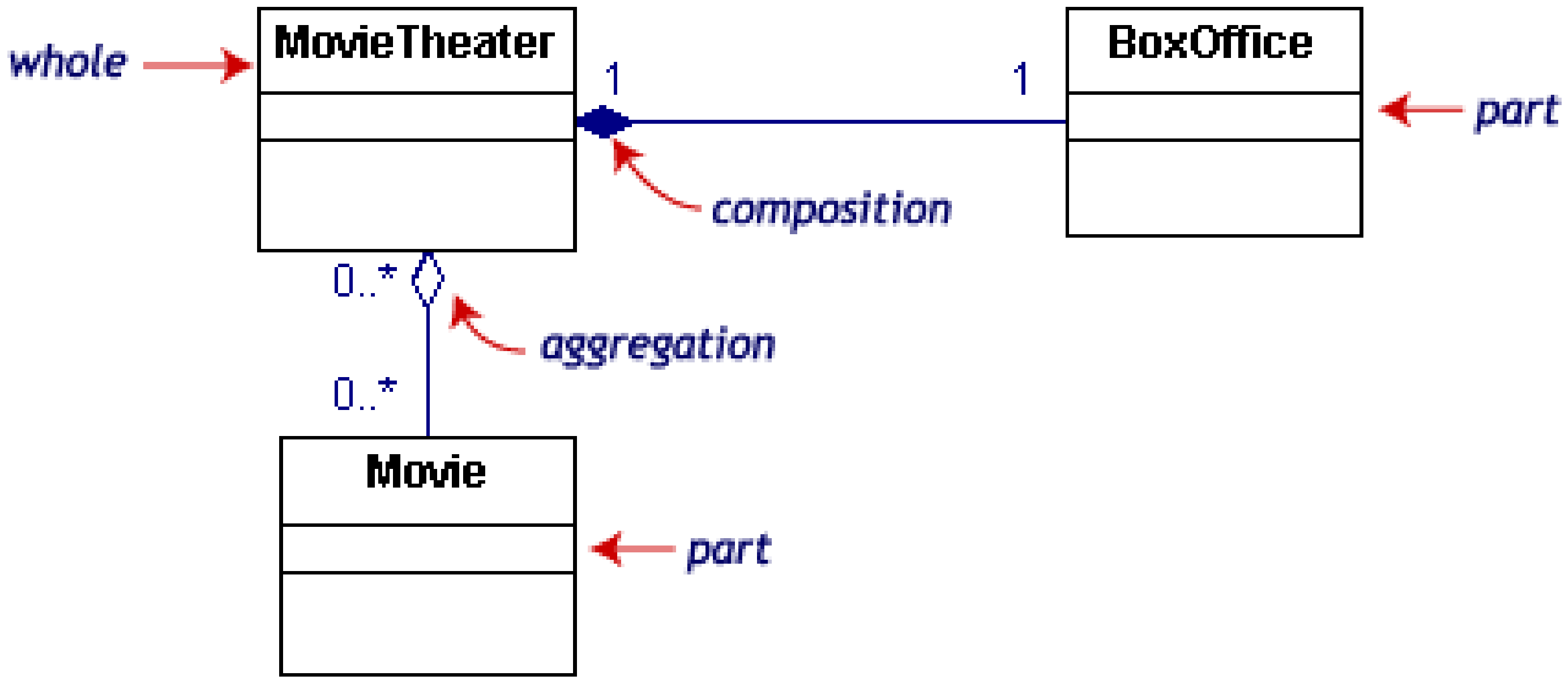
- **aggregation**: "is part of"
  - symbolized by a clear white diamond
- **composition**: "is entirely made of"
  - stronger version of aggregation
  - the parts live and die with the whole
  - symbolized by a black diamond
- **dependency**: "uses temporarily"
  - symbolized by dotted line
  - often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





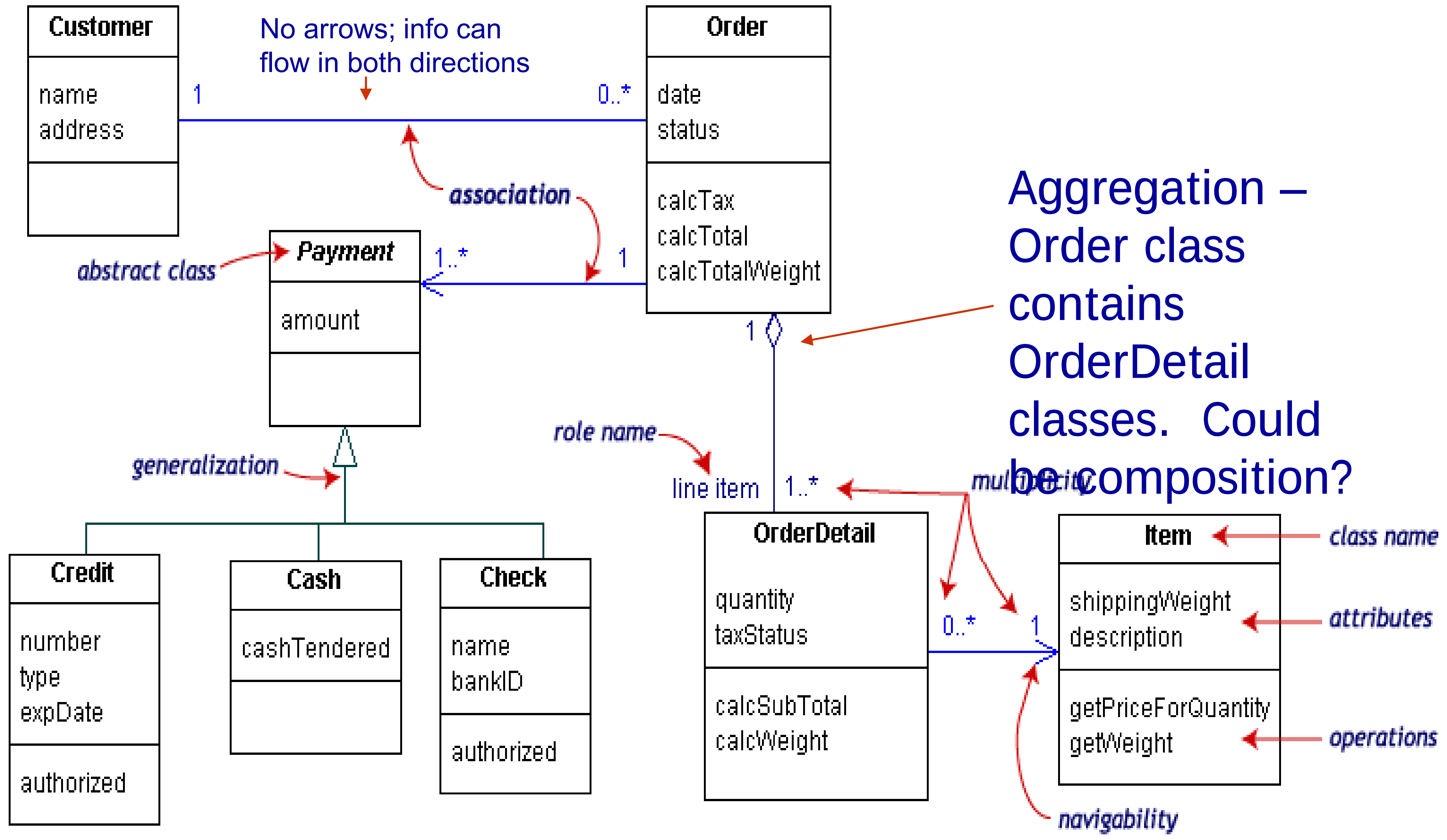


Composition/aggregation example



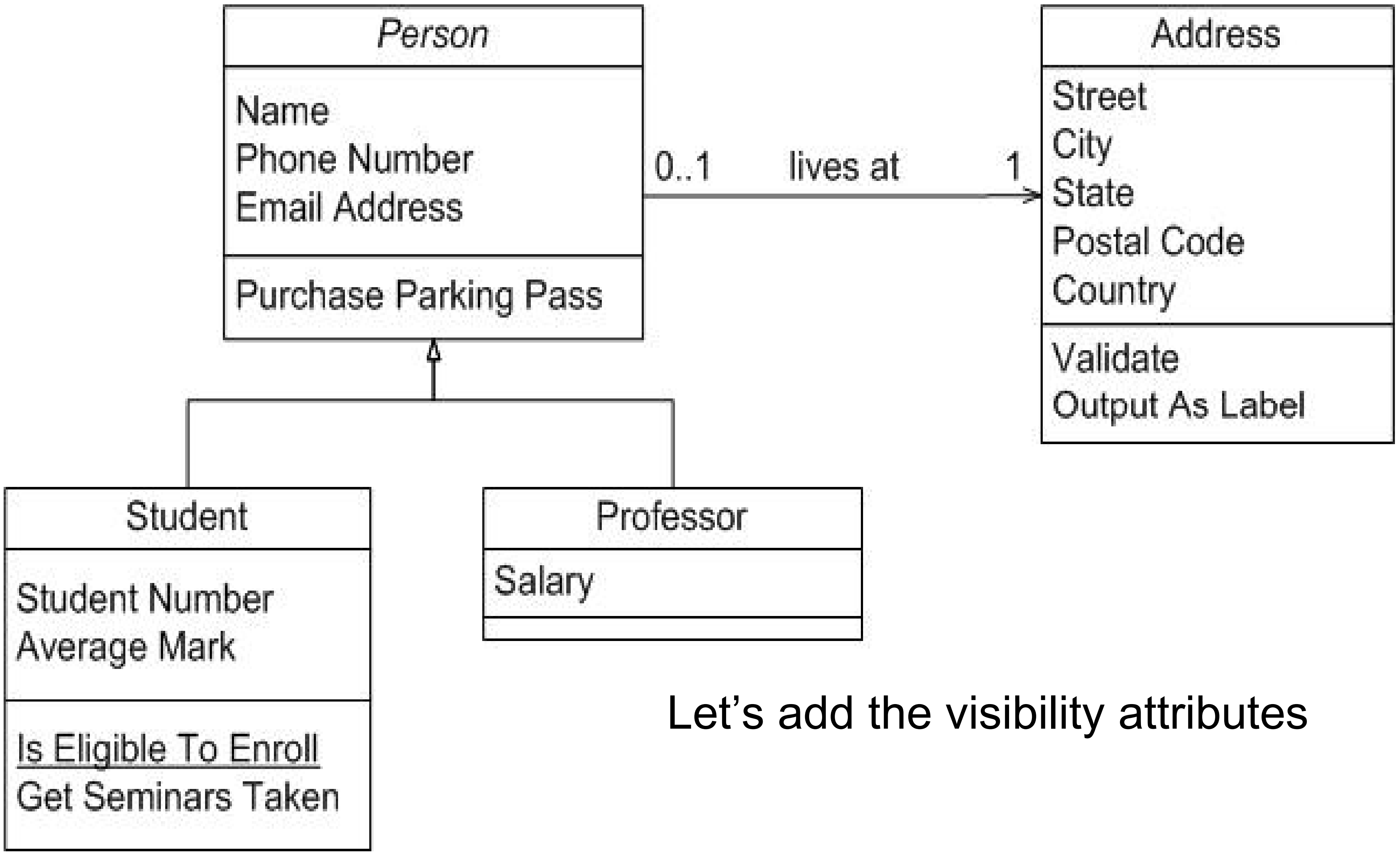
If the movie theatre goes away
so does the box office => composition
but movies may still exist => aggregation

Class diagram example



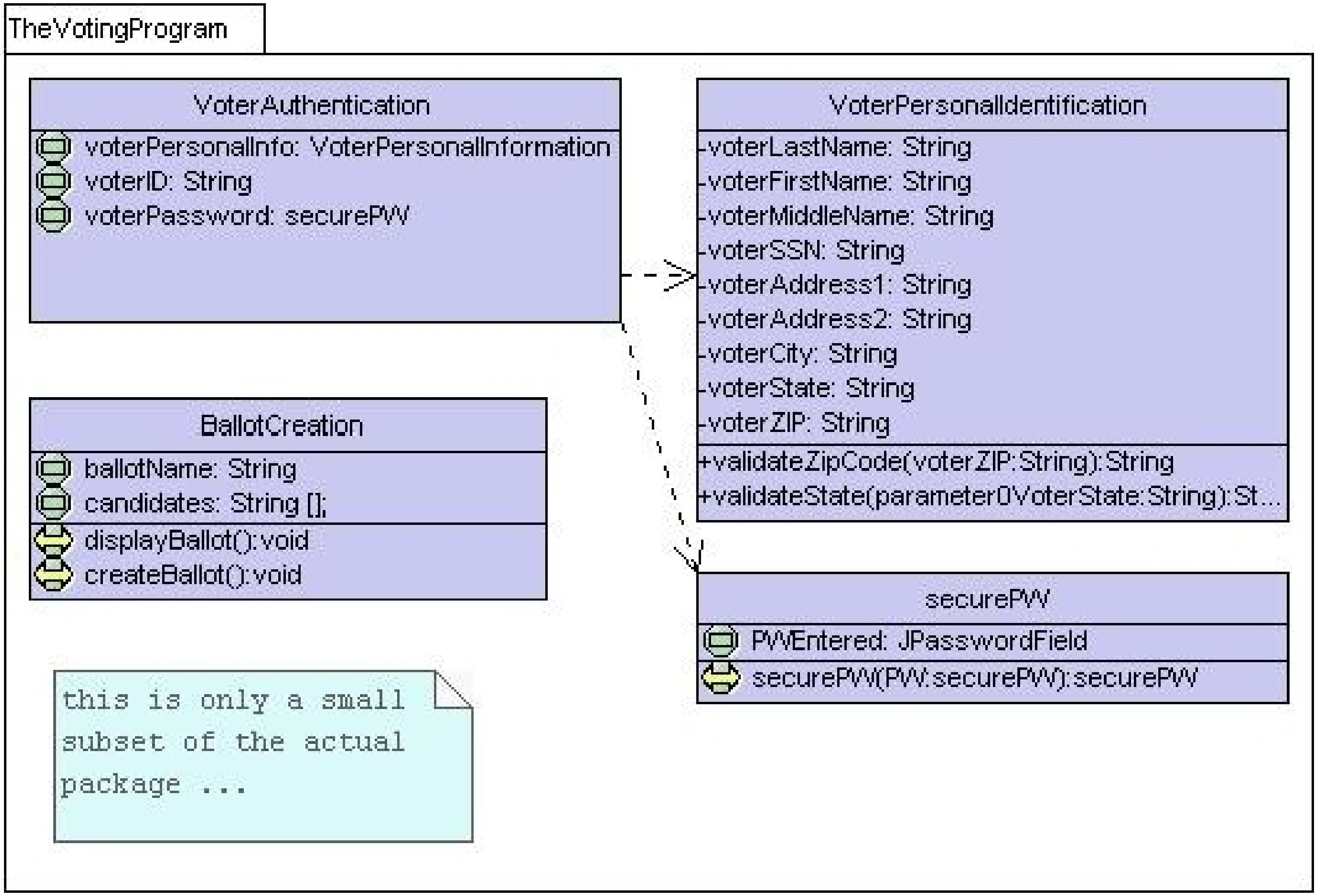
Aggregation – Order class contains OrderDetail classes. Could be composition?

UML example: people

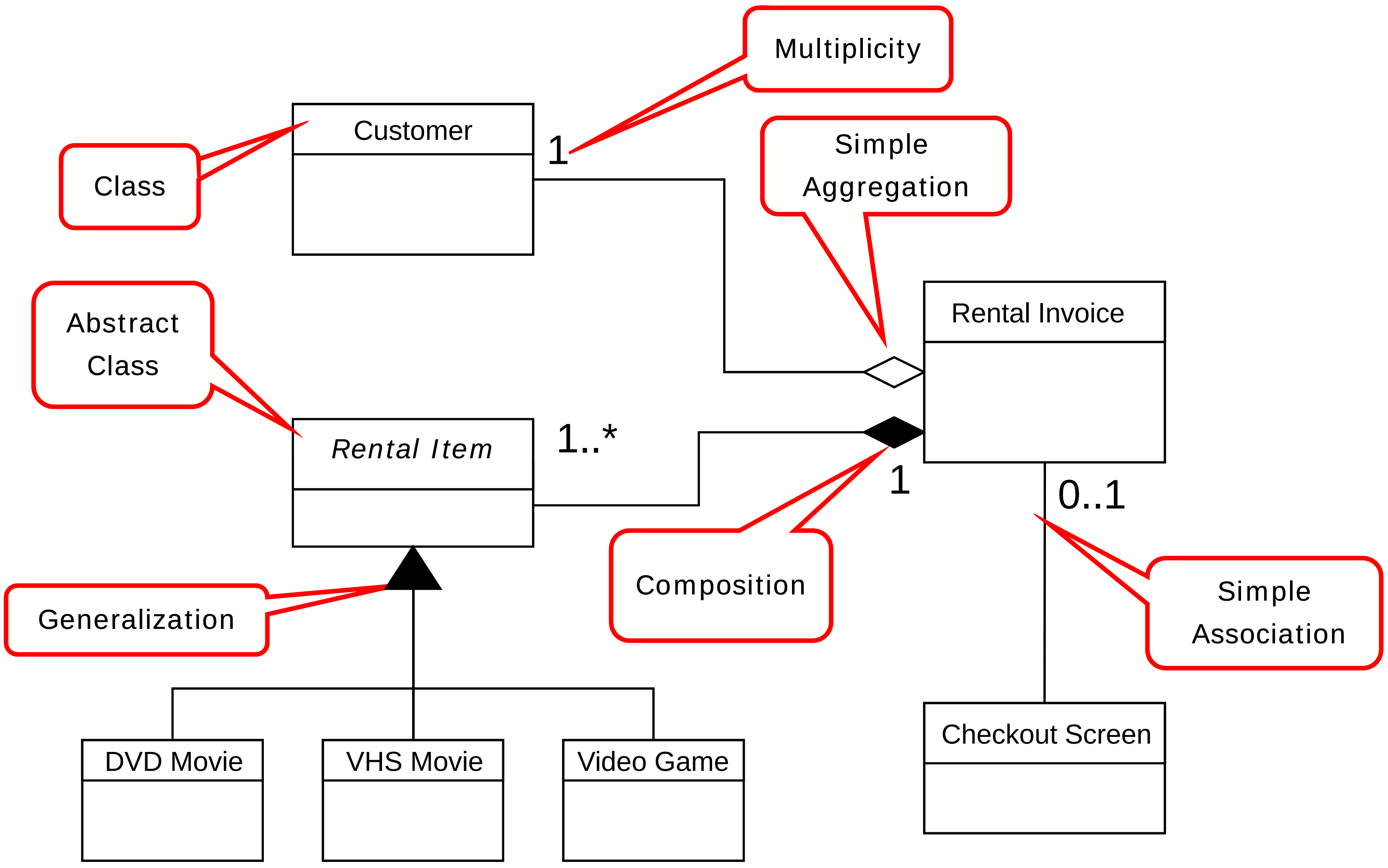


Let’s add the visibility attributes

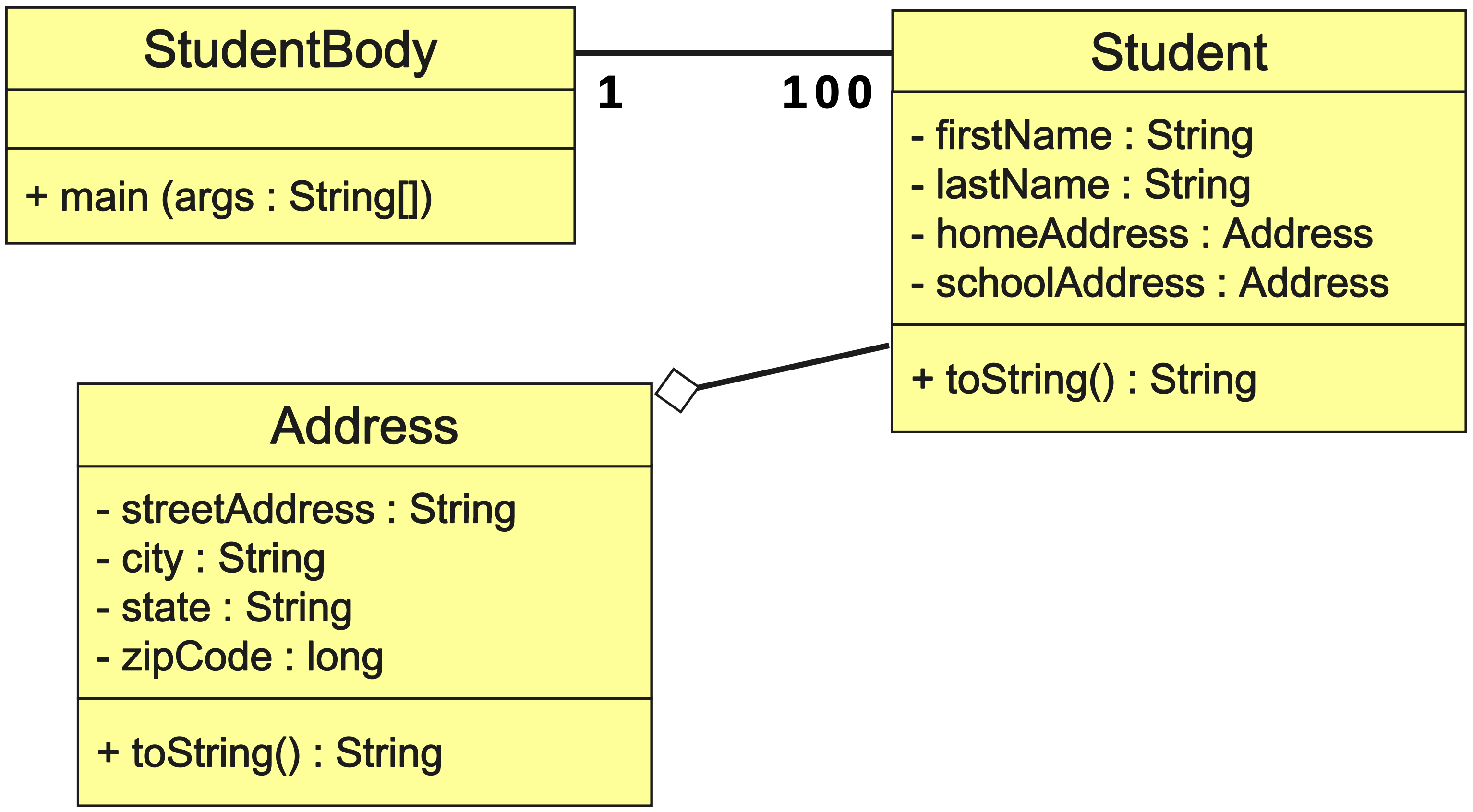
Class diagram: voters



Class diagram example: video store



Class diagram example: student



Tools for creating UML diags.

- Violet (free)
  - http://horstmann.com/violet/

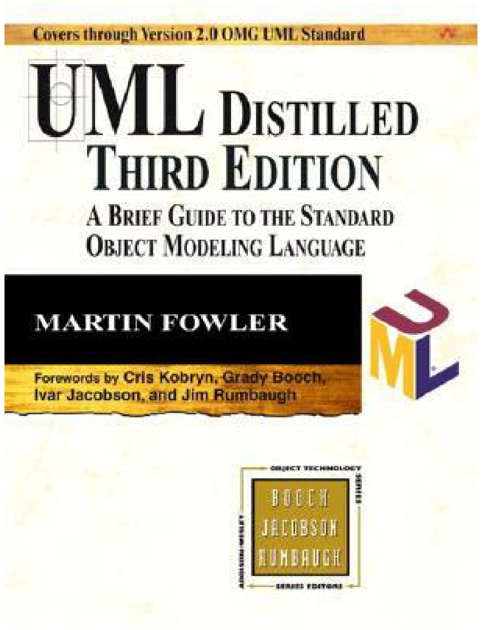
- Rational Rose
  - http://www.rational.com/

- Visual Paradigm UML Suite (trial)
  - http://www.visual-paradigm.com/
  - (nearly) direct download link:
    http://www.visual-paradigm.com/vp/download.jsp?product=vpuml&edition=ce

(there are many others, but most are commercial)

Class design exercise

- Consider this Texas Hold 'em poker game system:
  - 2 to 8 human or computer players
  - Each player has a name and stack of chips
  - Computer players have a difficulty setting: easy, medium, hard
  - Summary of each hand:
    - Dealer collects ante from appropriate players, shuffles the deck, and deals each player a hand of 2 cards from the deck.
    - A betting round occurs, followed by dealing 3 shared cards from the deck.
    - As shared cards are dealt, more betting rounds occur, where each player can fold, check, or raise.
    - At the end of a round, if more than one player is remaining, players' hands are compared, and the best hand wins the pot of all chips bet so far.
  - What classes are in this system? What are their responsibilities? Which classes collaborate?
  - Draw a class diagram for this system. Include relationships between classes (generalization and associational).



CSE 403

UML Sequence Diagrams

Suggested reading:

UML Distilled Ch. 4, by M. Fowler

UML sequence diagrams

- **sequence diagram**: an "interaction diagram" that models a single scenario executing in the system
  - perhaps 2nd most used UML diagram (behind class diagram)

- relation of UML diagrams to other exercises:
  - CRC cards -> class diagram
  - use cases -> sequence diagrams

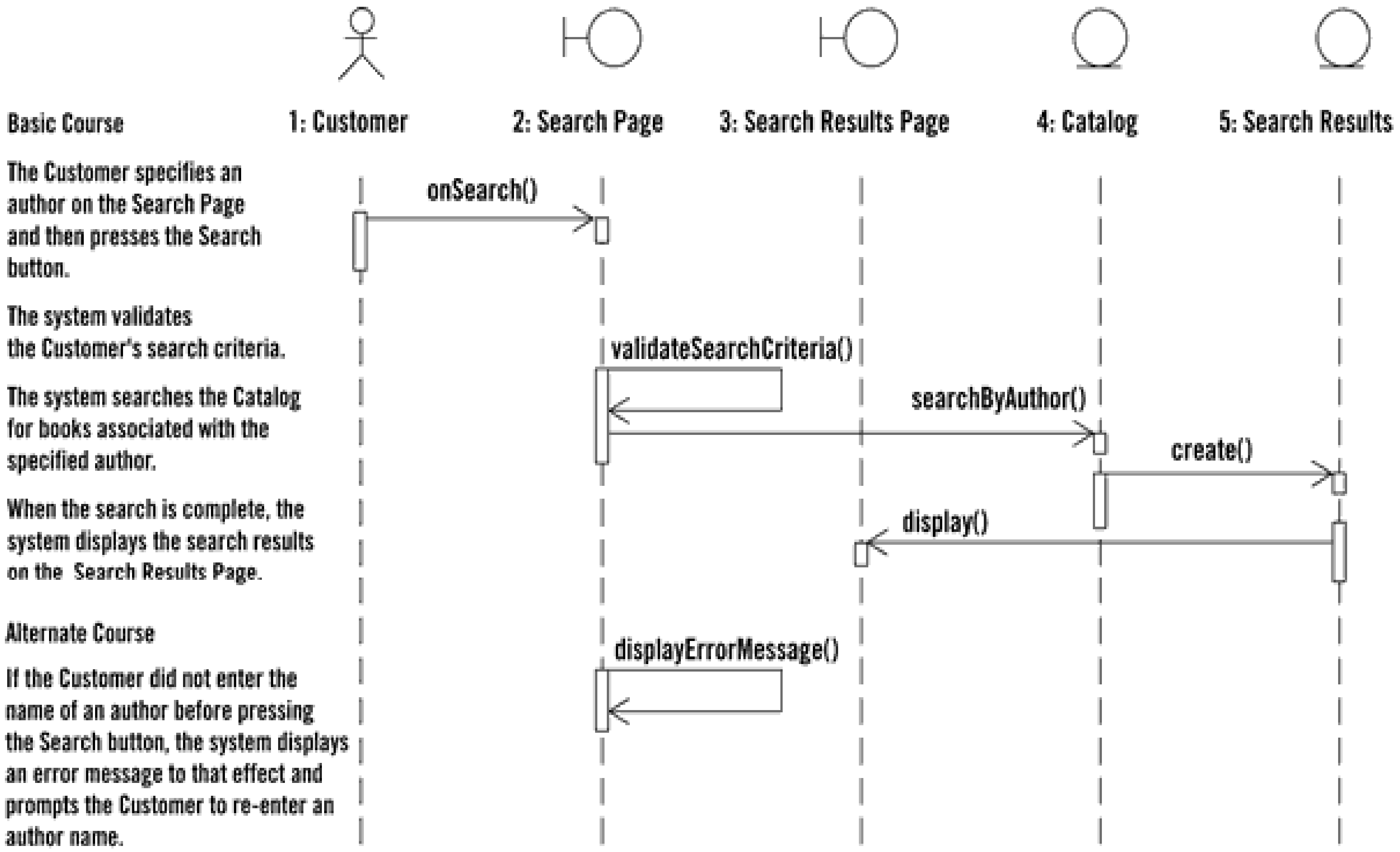
Key parts of a sequence diag.

- **participant**: an object or entity that acts in the sequence diagram
  - sequence diagram starts with an unattached "found message" arrow

- **message**: communication between participant objects

- the axes in a sequence diagram:
  - horizontal: which object/participant is acting
  - vertical: time (down -> forward in time)

Sequence dg. from use case



Basic Course

The Customer specifies an author on the Search Page and then presses the Search button.

The system validates the Customer's search criteria.

The system searches the Catalog for books associated with the specified author.

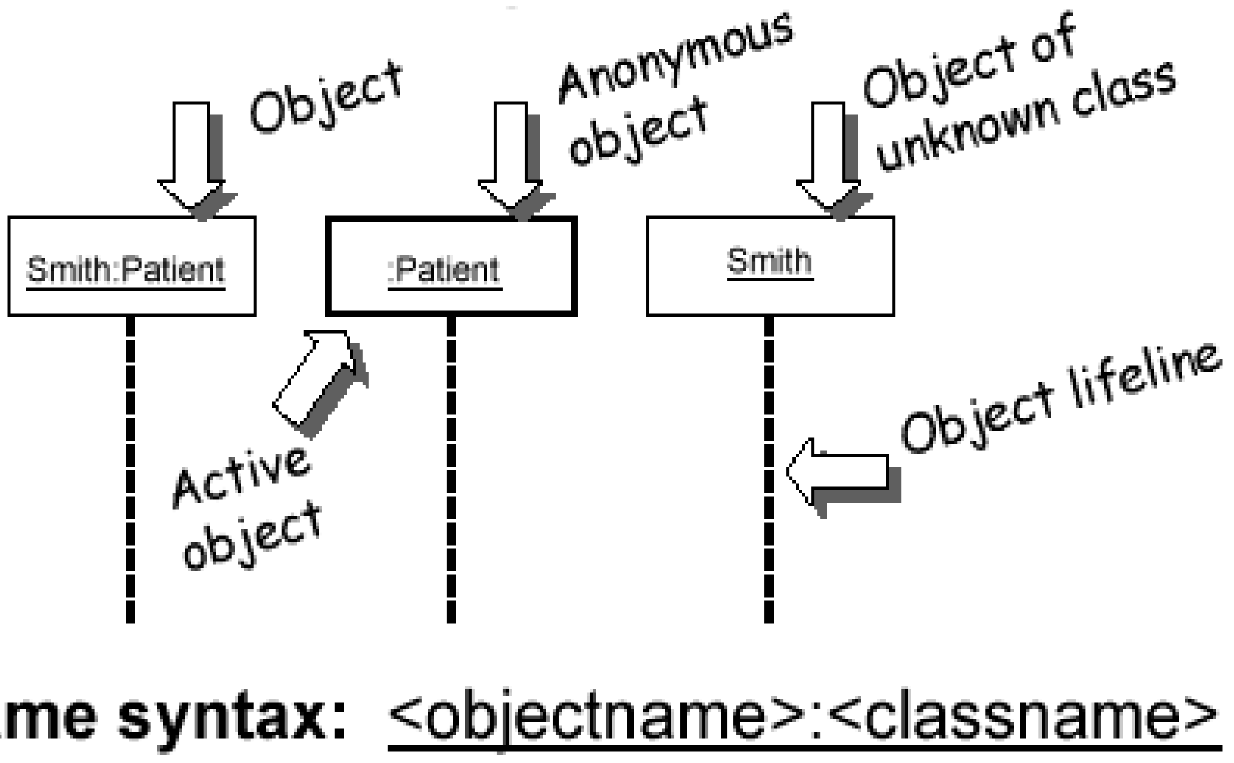
When the search is complete, the system displays the search results on the Search Results Page.

Alternate Course

If the Customer did not enter the name of an author before pressing the Search button, the system displays an error message to that effect and prompts the Customer to re-enter an author name.

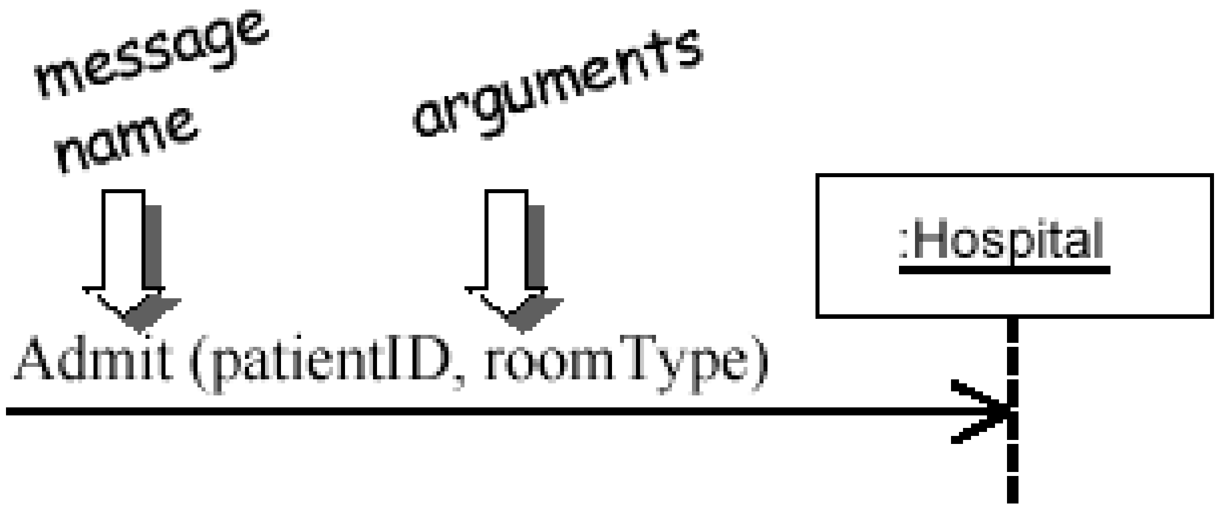
Representing objects

- Squares with object type, optionally preceded by object name and colon
  - write object's name if it clarifies the diagram
  - object's "life line" represented by dashed vert. line



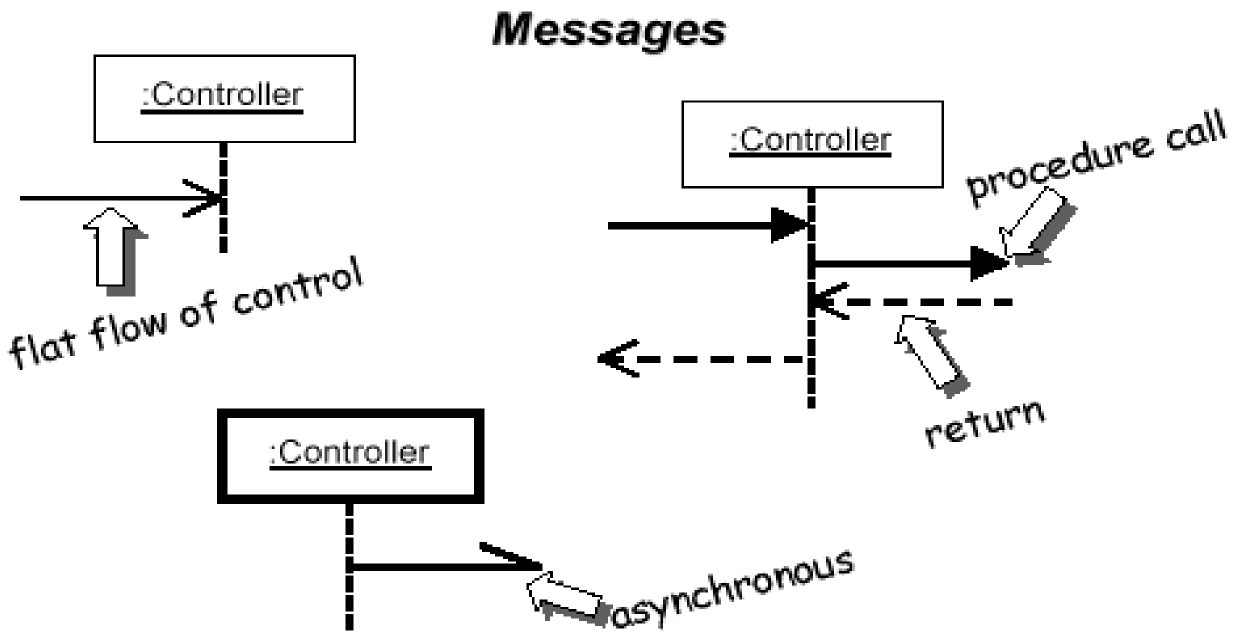
Messages between objects

- message (method call) indicated by horizontal arrow to other object
  - write message name and arguments above arrow



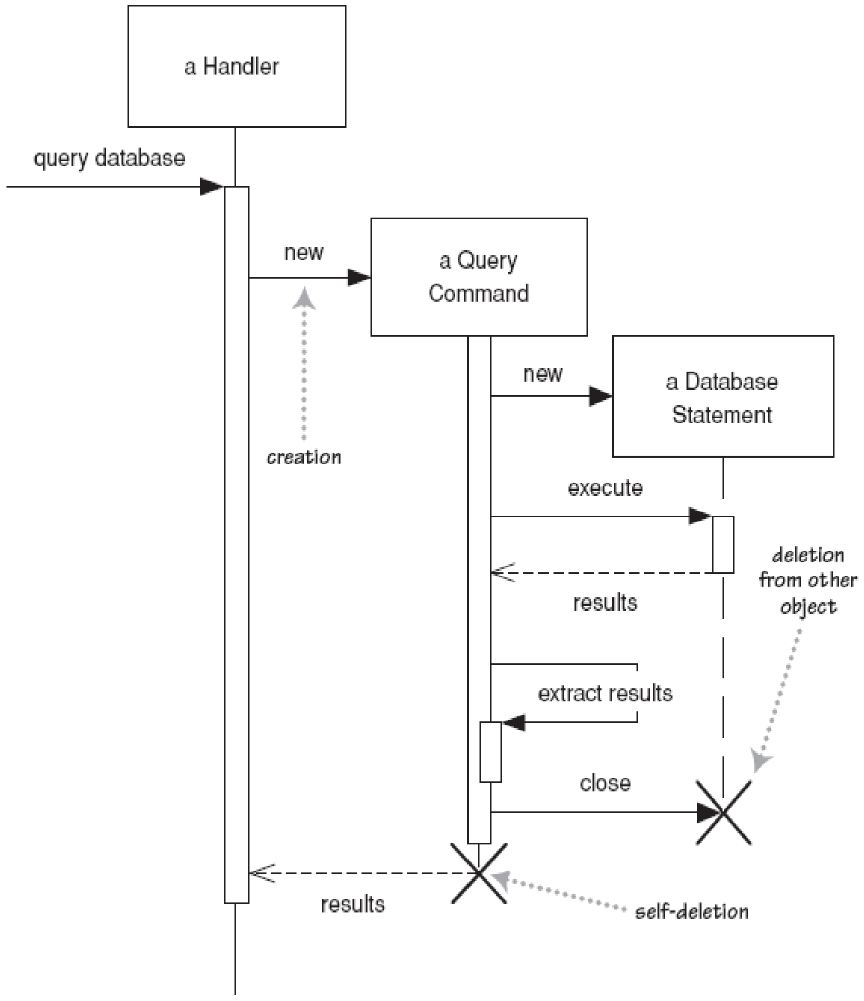
Messages, continued

- message (method call) indicated by horizontal arrow to other object
  - dashed arrow back indicates return
  - different arrowheads for normal / concurrent (asynchronous) methods



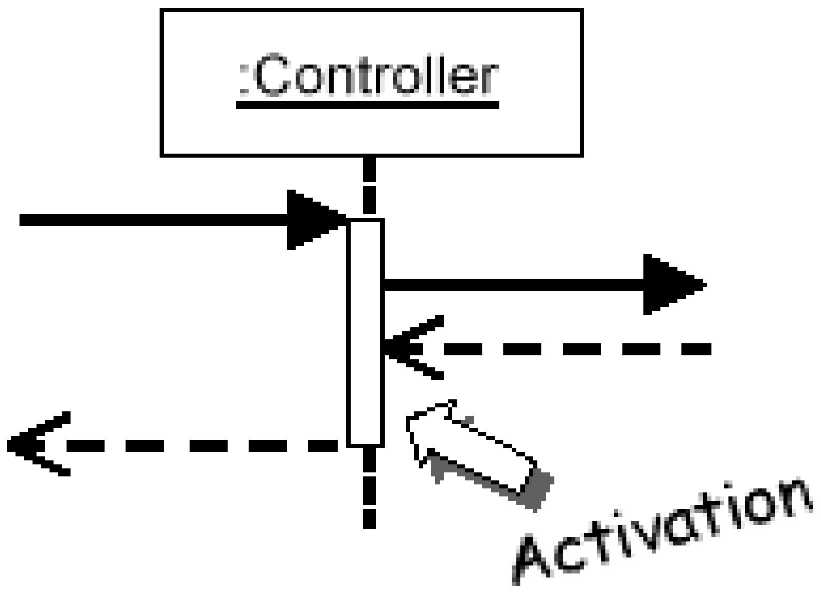
Lifetime of objects

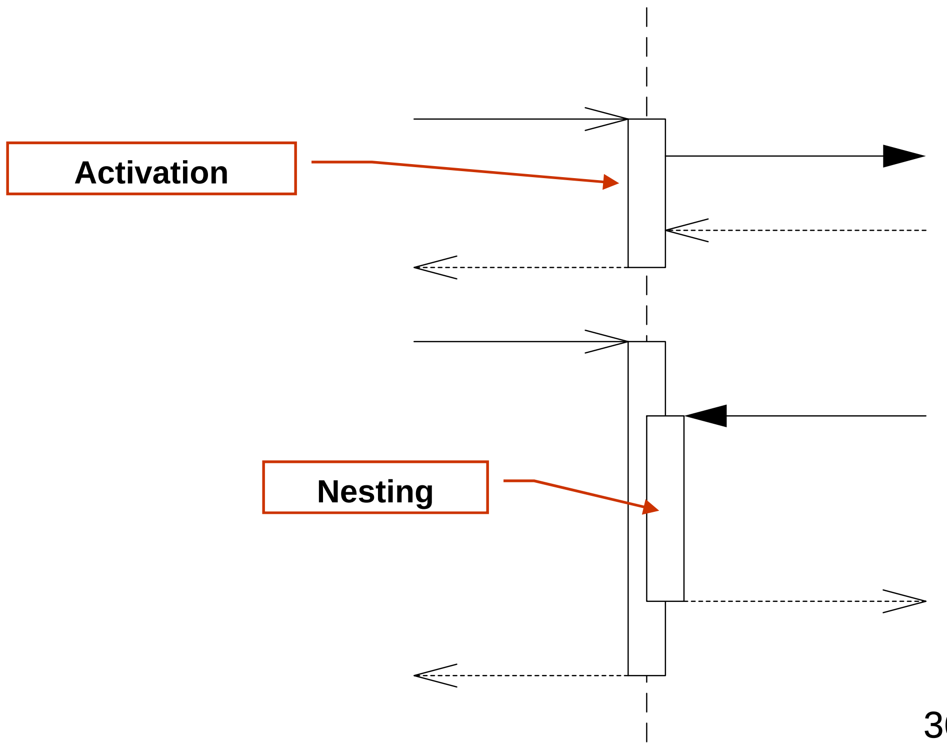
- *creation*: arrow with 'new' written above it
  - notice that an object created after the start of the scenario appears lower than the others
- *deletion*: an X at bottom of object's lifeline
  - Java doesn't explicitly delete objects; they fall out of scope and are garbage-collected



Indicating method calls

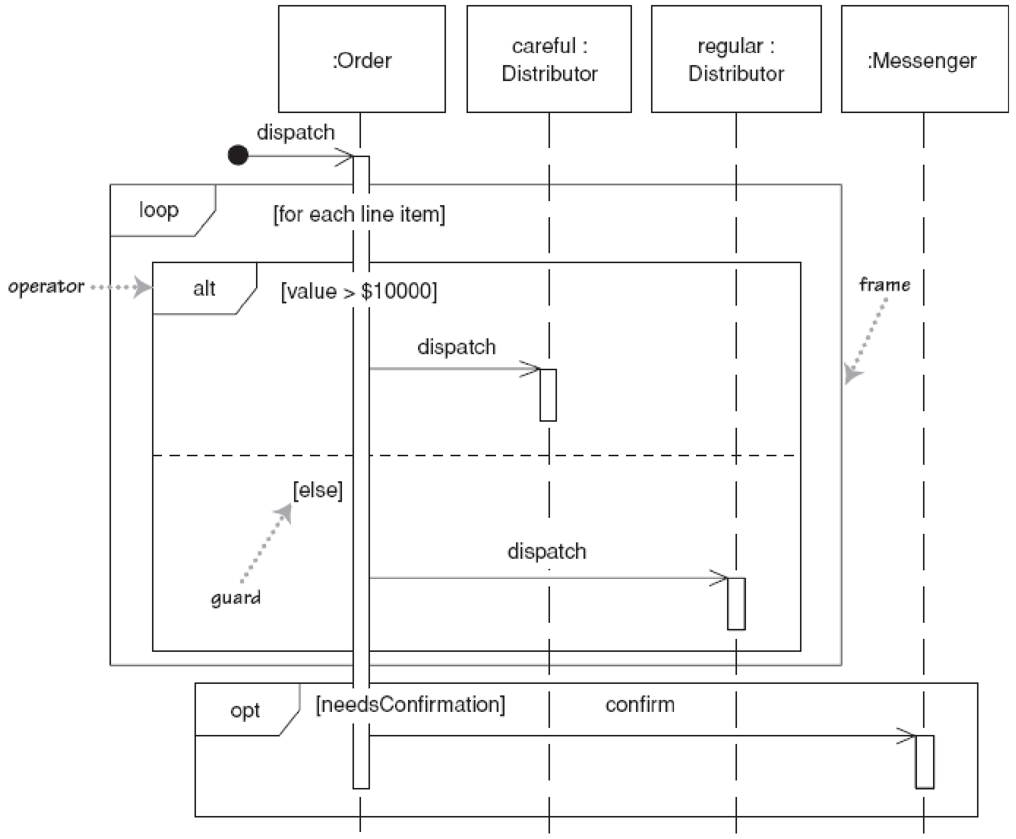
- **activation**: thick box over object's life line; drawn when object's method is on the stack
  - either that object is running its code, or it is on the stack waiting for another object's method to finish
  - nest to indicate recursion





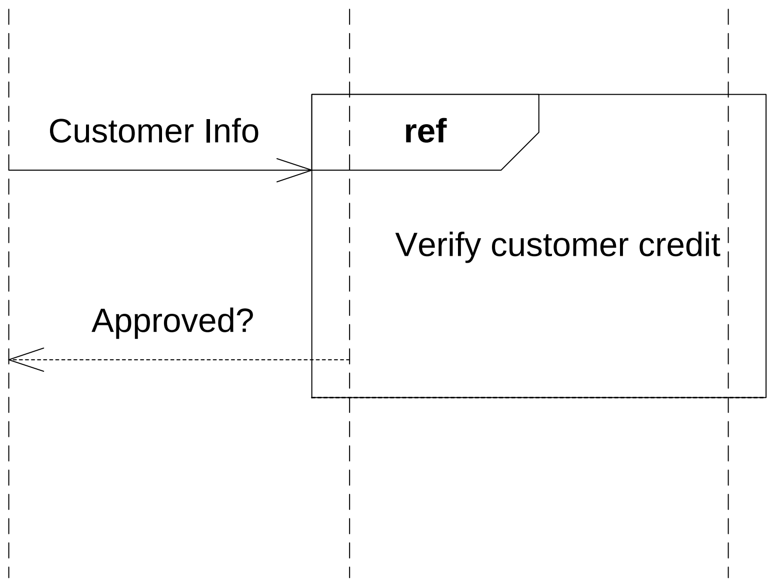
Indicating selection and loops

- frame: box around part of a sequence diagram to indicate selection or loop
  - `if       ->` (opt) [condition]
  - `if/else ->` (alt) [condition], separated by horizontal dashed line
  - `loop     ->` (loop) [condition or items to loop over]



linking sequence diagrams

- if one sequence diagram is too large or refers to another diagram, indicate it with either:
  - an unfinished arrow and comment
  - a "ref" frame that names the other diagram
  - when would this occur in our system?



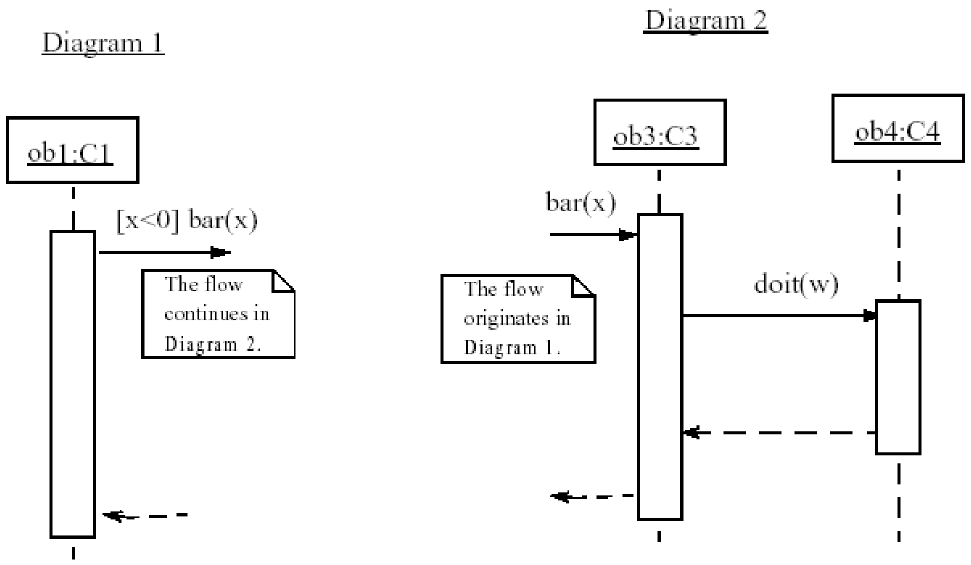
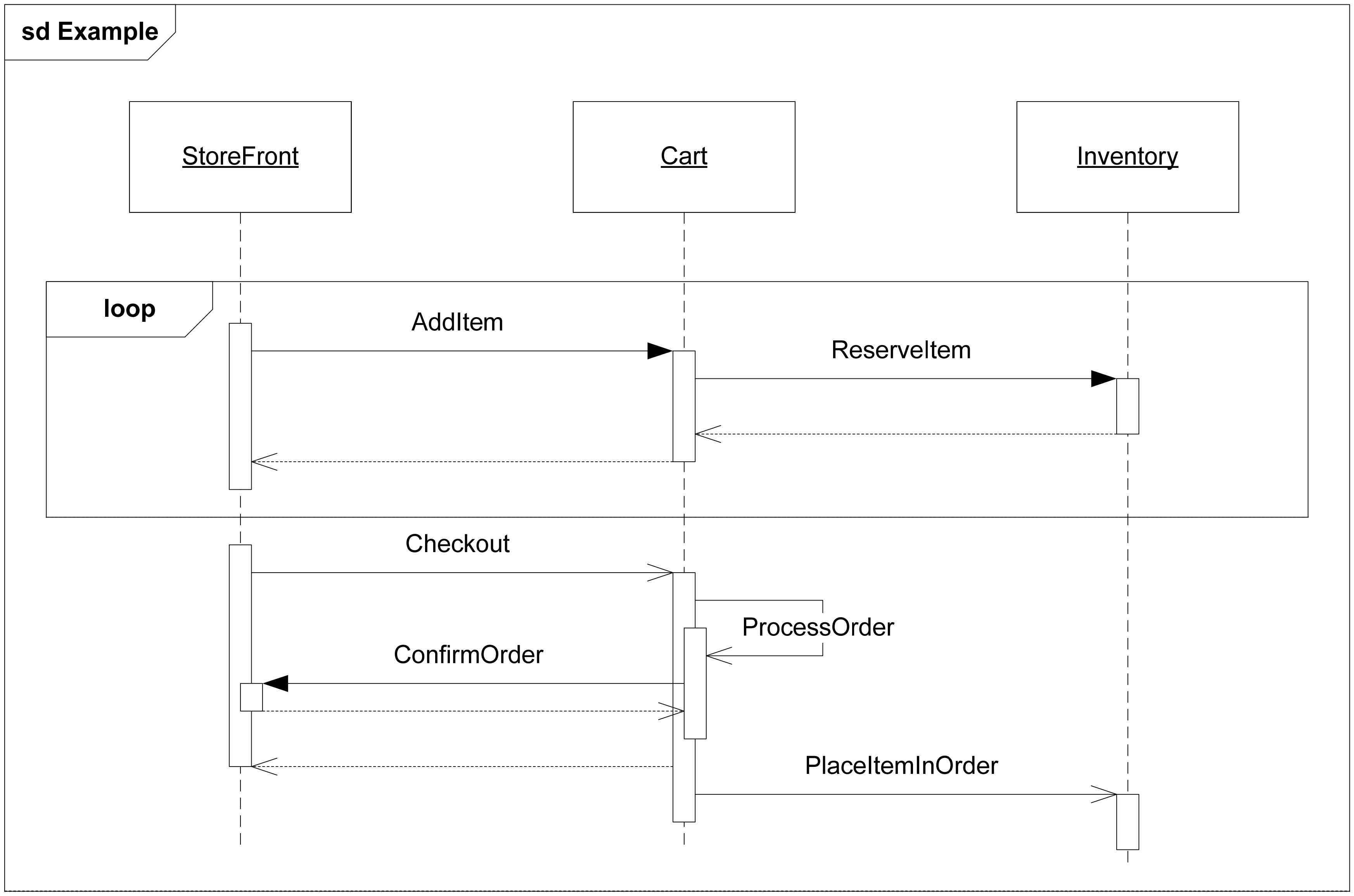


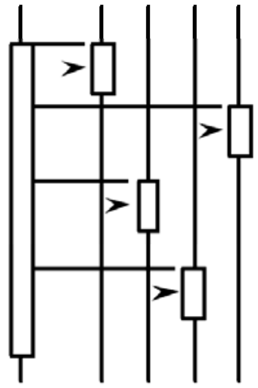
Diagram 1
ob1:C1
[x<0] bar(x)
The flow continues in Diagram 2.
Diagram 2
ob3:C3
ob4:C4
bar(x)
doit(w)
The flow originates in Diagram 1.

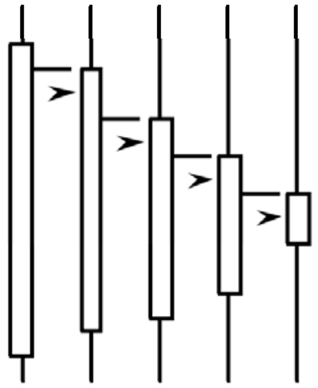
Example sequence diagram

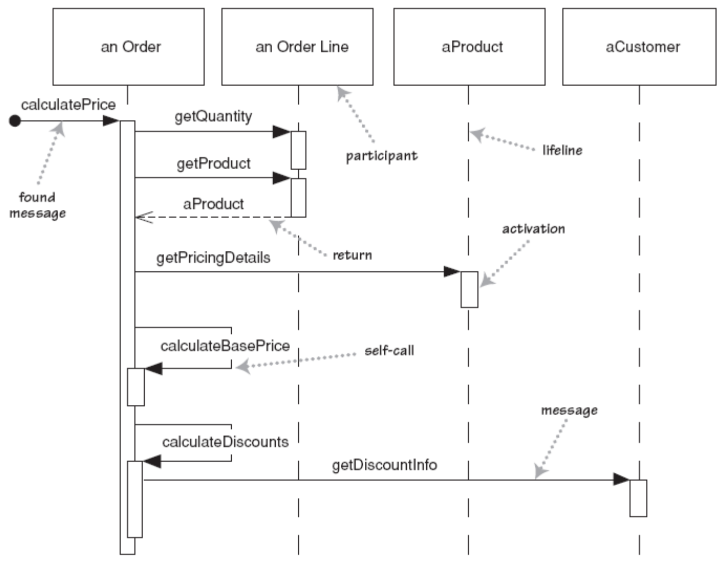


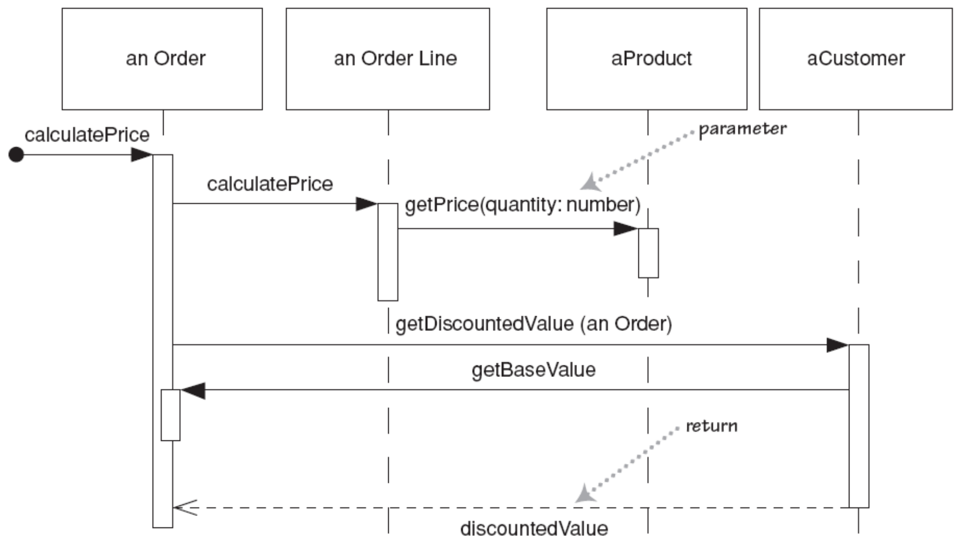
Forms of system control

- What can you say about the control flow of each of the following systems?
  - Is it centralized?
  - Is it distributed?









Why not just code it?

- Sequence diagrams can be somewhat close to the code level. So why not just code up that algorithm rather than drawing it as a sequence diagram?
  - a good sequence diagram is still a bit above the level of the real code (not all code is drawn on diagram)
  - sequence diagrams are language-agnostic (can be implemented in many different languages
  - non-coders can do sequence diagrams
  - easier to do sequence diagrams as a team
  - can see many objects/classes at a time on same page (visual bandwidth)

Sequence diagram exercise 1

- Let's do a sequence diagram for the following casual use case, *Start New Poker Round* :

  The scenario begins when the player chooses to start a new round in the UI. The UI asks whether any new players want to join the round; if so, the new players are added using the UI.

  All players' hands are emptied into the deck, which is then shuffled. The player left of the dealer supplies an ante bet of the proper amount. Next each player is dealt a hand of two cards from the deck in a round-robin fashion; one card to each player, then the second card.

  If the player left of the dealer doesn't have enough money to ante, he/she is removed from the game, and the next player supplies the ante. If that player also cannot afford the ante, this cycle continues until such a player is found or all players are removed.

Sequence diagram exercise 2

- Let's do a sequence diagram for the following casual use case, *Add Calendar Appointment* :

  The scenario begins when the user chooses to add a new appointment in the UI. The UI notices which part of the calendar is active and pops up an Add Appointment window for that date and time.

  The user enters the necessary information about the appointment's name, location, start and end times. The UI will prevent the user from entering an appointment that has invalid information, such as an empty name or negative duration. The calendar records the new appointment in the user's list of appointments. Any reminder selected by the user is added to the list of reminders.

  If the user already has an appointment at that time, the user is shown a warning message and asked to choose an available time or replace the previous appointment. If the user enters an appointment with the same name and duration as an existing group meeting, the calendar asks the user whether he/she intended to join that group meeting instead. If so, the user is added to that group meeting's list of participants.