

# **UML Class Diagrams**

# Outline

- Designing classes
- Overview of UML
- UML class diagrams
  - Syntax and semantics
  - Examples



**design phase: from requirements to code**

# Software design

- **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 to design classes?

Identify classes and interactions from project requirements:

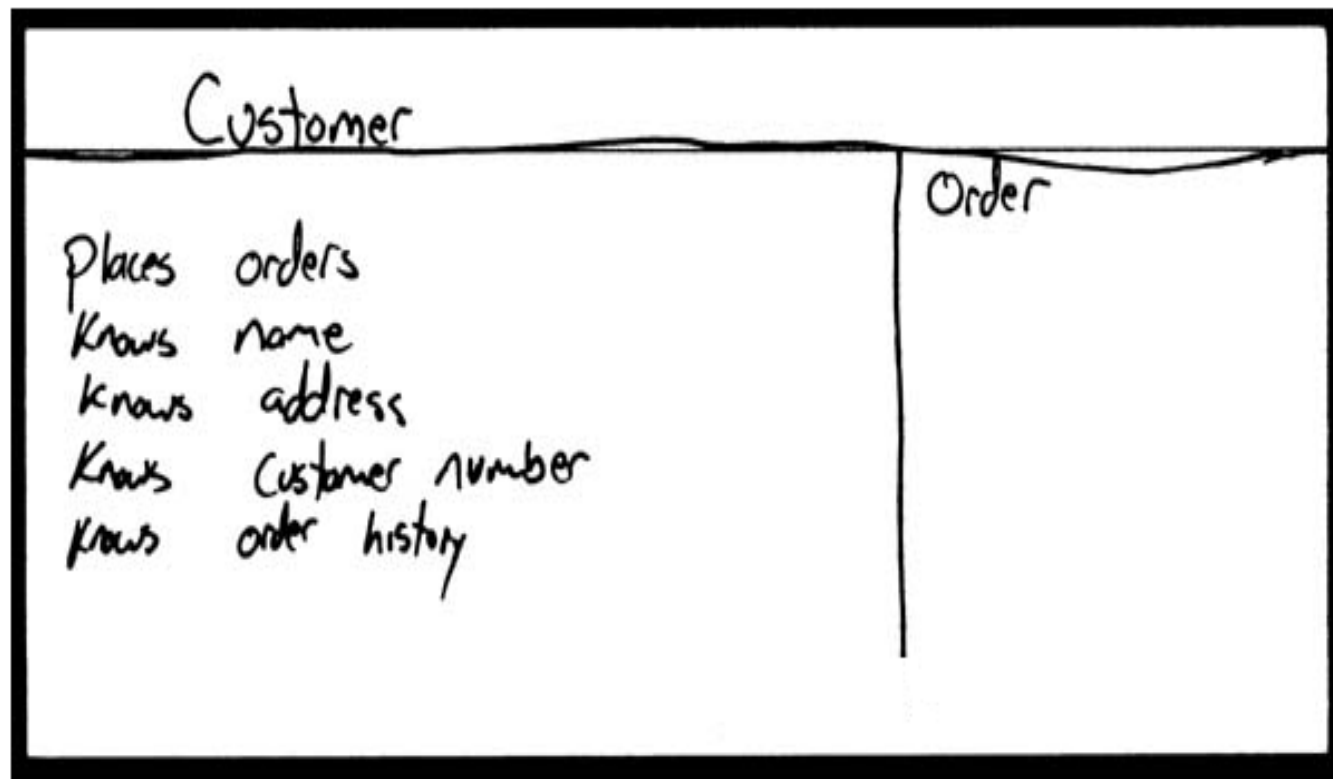
- **Nouns** are potential classes, objects, and fields
- **Verbs** are potential methods or responsibilities of a class
- **Relationships** between nouns are potential interactions (containment, generalization, dependence, etc.)

- Which nouns in your project should be classes?
- Which ones are fields?
- What verbs should be methods?
- What are potential interactions between your classes?

# Describing designs with CRC cards

## CRC (class-responsibility-collaborators) cards

- on top of the card, write down the name of the class
- below the name, list the following:
  - **responsibilities:** problems to be solved; short verb phrases
  - **collaborators:** other classes that are sent messages by this class



# basics

## **describing designs with UML: an overview**

# What is UML?

- Pictures or views 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



# UML: Unified Modeling Language

- **Union of Many Languages**

- Use case diagrams
- Class diagrams
- Object diagrams
- Sequence diagrams
- Collaboration diagrams
- Statechart diagrams
- Activity diagrams
- Component diagrams
- Deployment diagrams
- ....



A very big language!

# 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

# 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

learn

## **UML class diagrams**

# What is a UML class diagram?

- A UML class diagram is a picture of
  - the classes in an OO system
  - their fields and methods
  - connections between the classes that interact or inherit from each other
- Not 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 a single class

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String

# Diagram of a single class

- Class name
  - write «interface» on top of interfaces' names
  - use *italics* for an abstract class name

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String

# Diagram of a single class

- Class name
  - write «interface» on top of interfaces' names
  - use *italics* for an abstract class name
- Attributes (optional)
  - fields of the class

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String



# Diagram of a single class

- Class name
  - write «interface» on top of interfaces' names
  - use *italics* for an abstract class name
- Attributes (optional)
  - fields of the class
- 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(w: int, h: int) + distance(r: Rectangle): double

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String

# Class attributes (fields, instance variables)

visibility name : type [count] = default\_value

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String

# Class attributes (fields, instance variables)

visibility name : type [count] = default\_value

- visibility

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

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String

# Class attributes (fields, instance variables)

visibility name : type [count] = default\_value

- visibility

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

- underline static attributes

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String

# Class attributes (fields, instance variables)

visibility name : type [count] = default\_value

- **visibility**
  - + public
  - # protected
  - private
  - ~ package (default)
  - / derived
- underline static attributes
- **derived attribute**: not stored, but can be computed from other attribute values
  - “specification fields” from CSE 331

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String

# Class operations / methods

visibility name(parameters) : return\_type

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String

# Class operations / methods

visibility name(parameters) : return\_type

- visibility

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

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String

# Class operations / methods

visibility name(parameters) : return\_type

- **visibility**
  - + public
  - # protected
  - private
  - ~ package (default)
- underline static methods

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String



# Class operations / methods

visibility name(parameters) : return\_type

- **visibility**
  - + public
  - # protected
  - private
  - ~ package (default)
- underline static methods
- **parameters** listed as **name : type**

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String

# Class operations / methods

visibility name(parameters) : return\_type

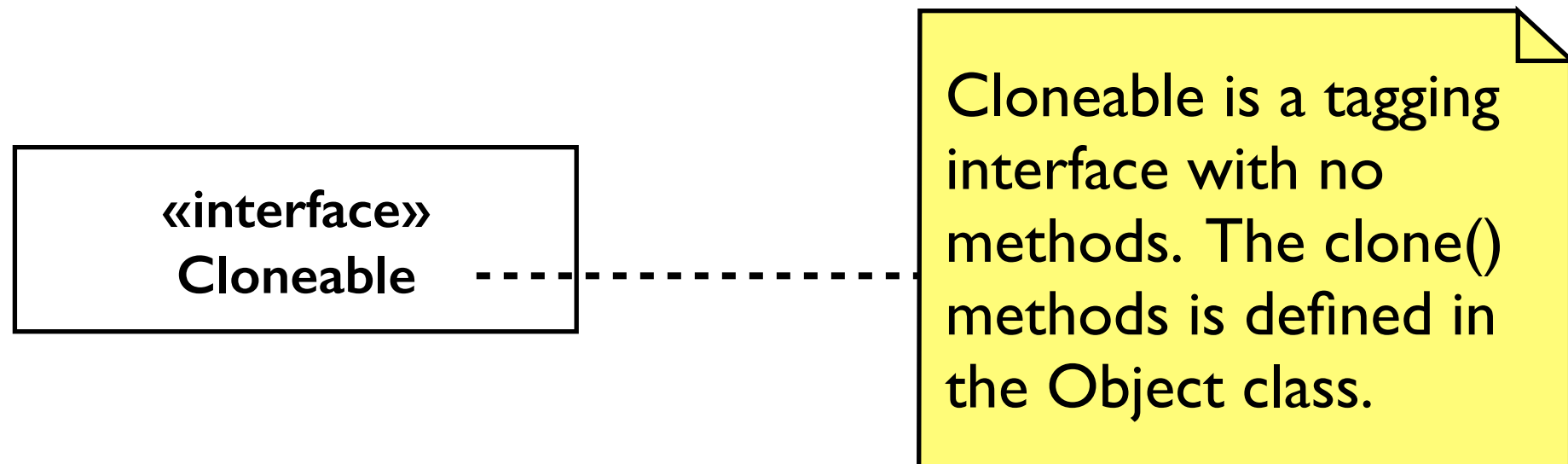
- **visibility**
  - + public
  - # protected
  - private
  - ~ package (default)
- underline static methods
- **parameters** listed as **name : type**
- omit **return\_type** on constructors and when return type is void

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

Student
- name: String - id: int - <u>totalStudents: int</u>
# getID(): int ~ getEmail(): String

# Comments

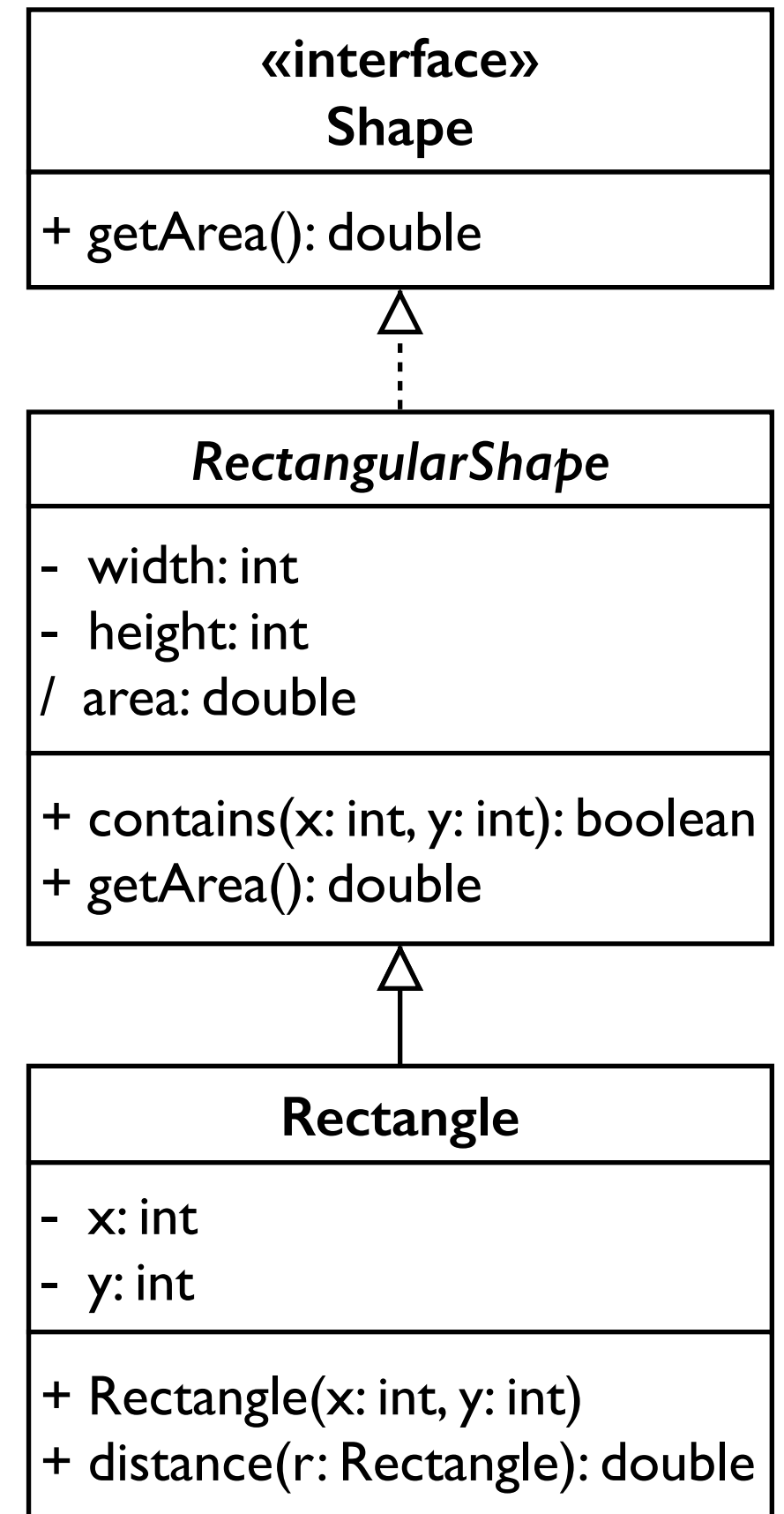
Represented as a folded note, attached to the appropriate class/method/etc by a dashed line



# Relationships between classes

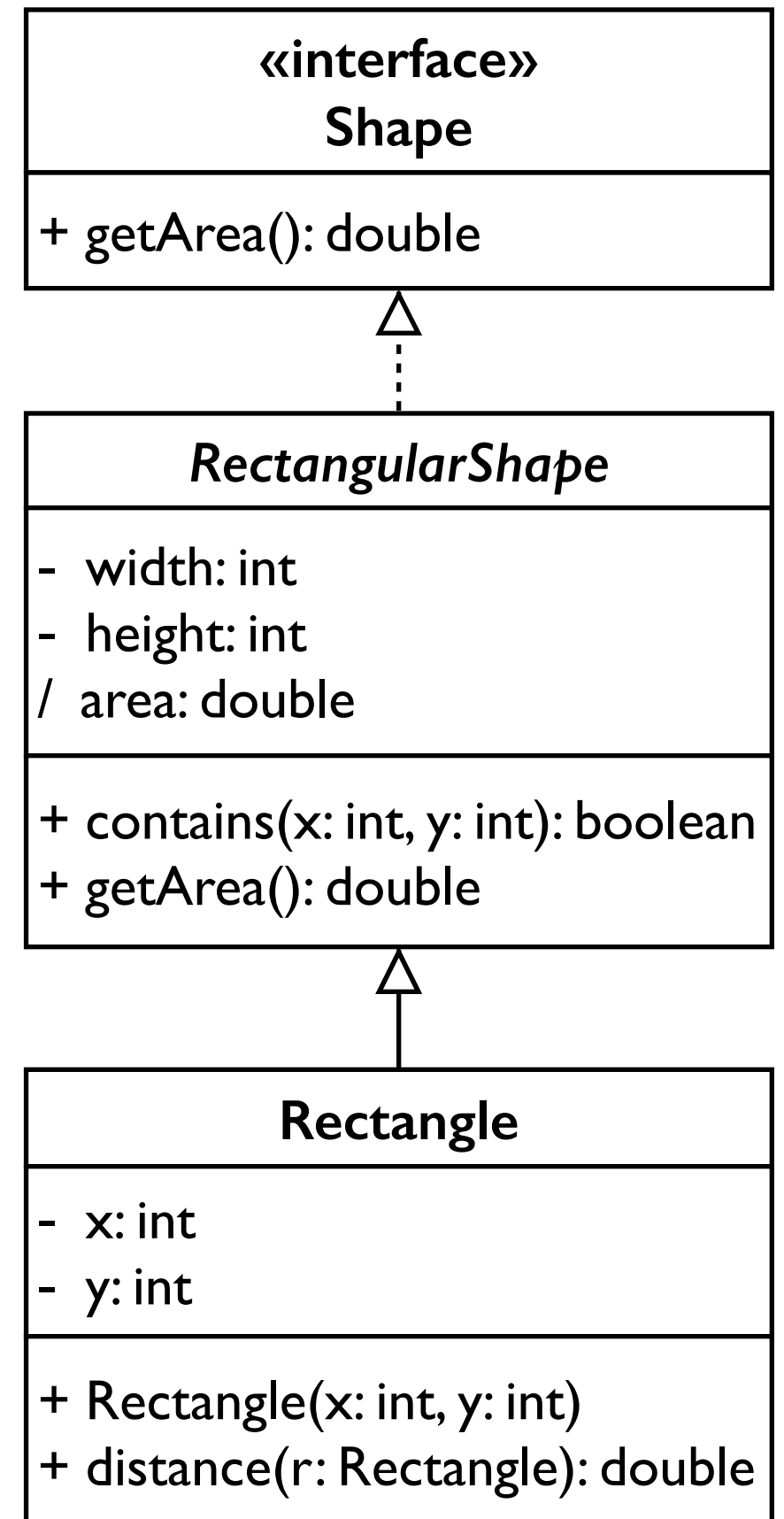
- **Generalization:** an inheritance relationship
  - inheritance between classes
  - interface implementation
- **Association:** a usage relationship
  - dependency
  - aggregation
  - composition

# Generalization relationships



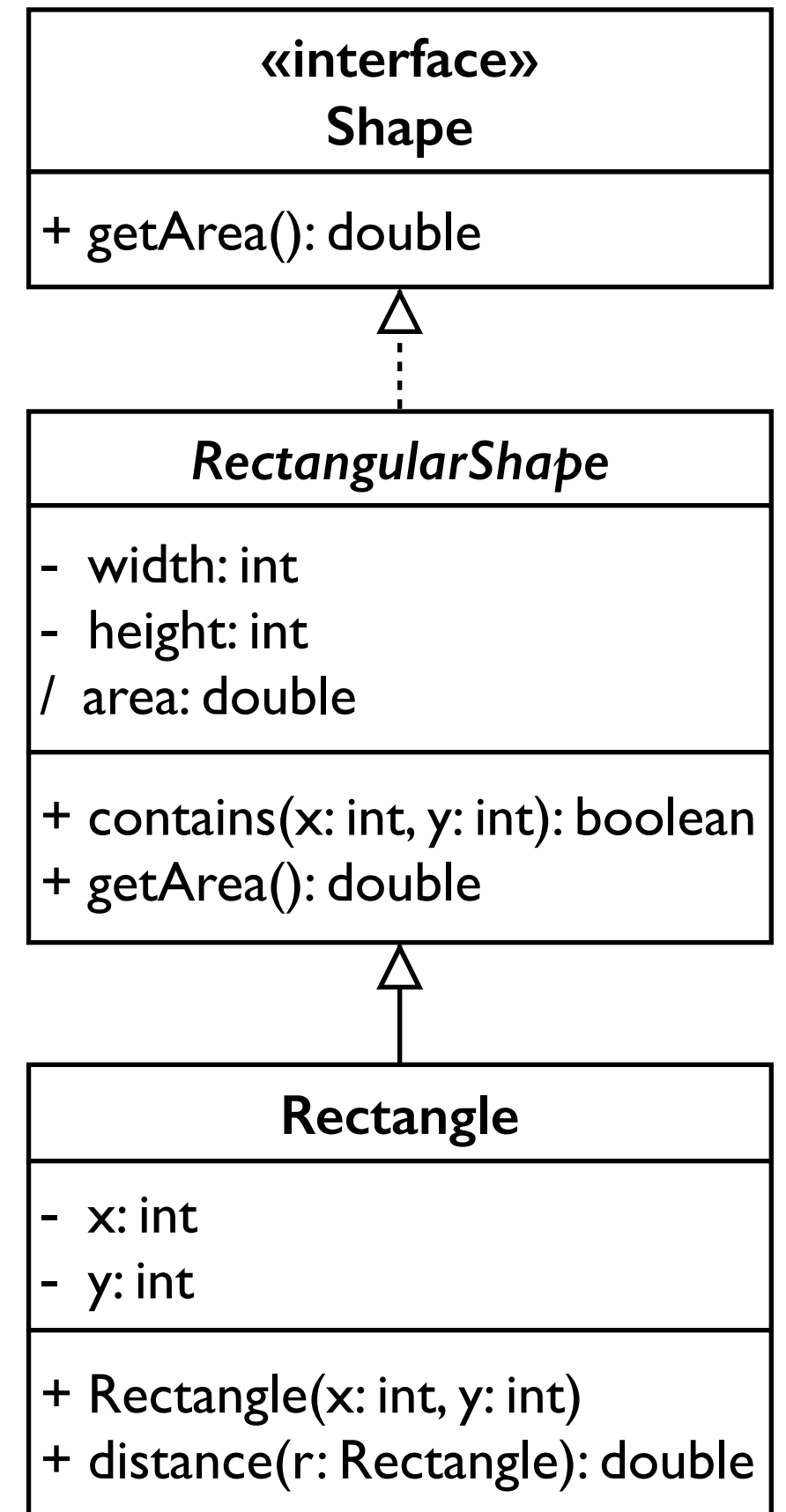
# Generalization relationships

- Hierarchies drawn top-down



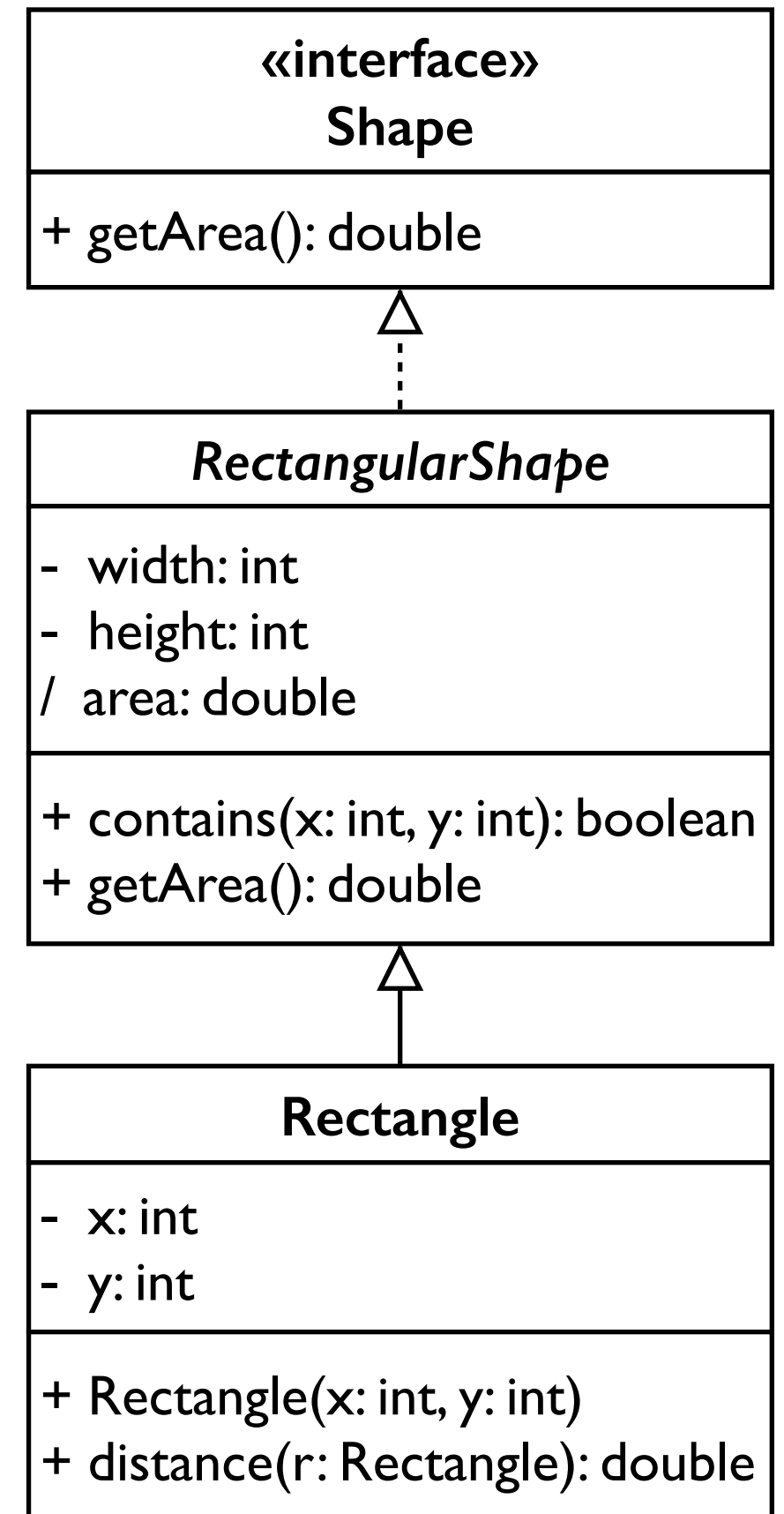
# Generalization relationships

- Hierarchies drawn top-down
- Arrows point upward to parent



# Generalization relationships

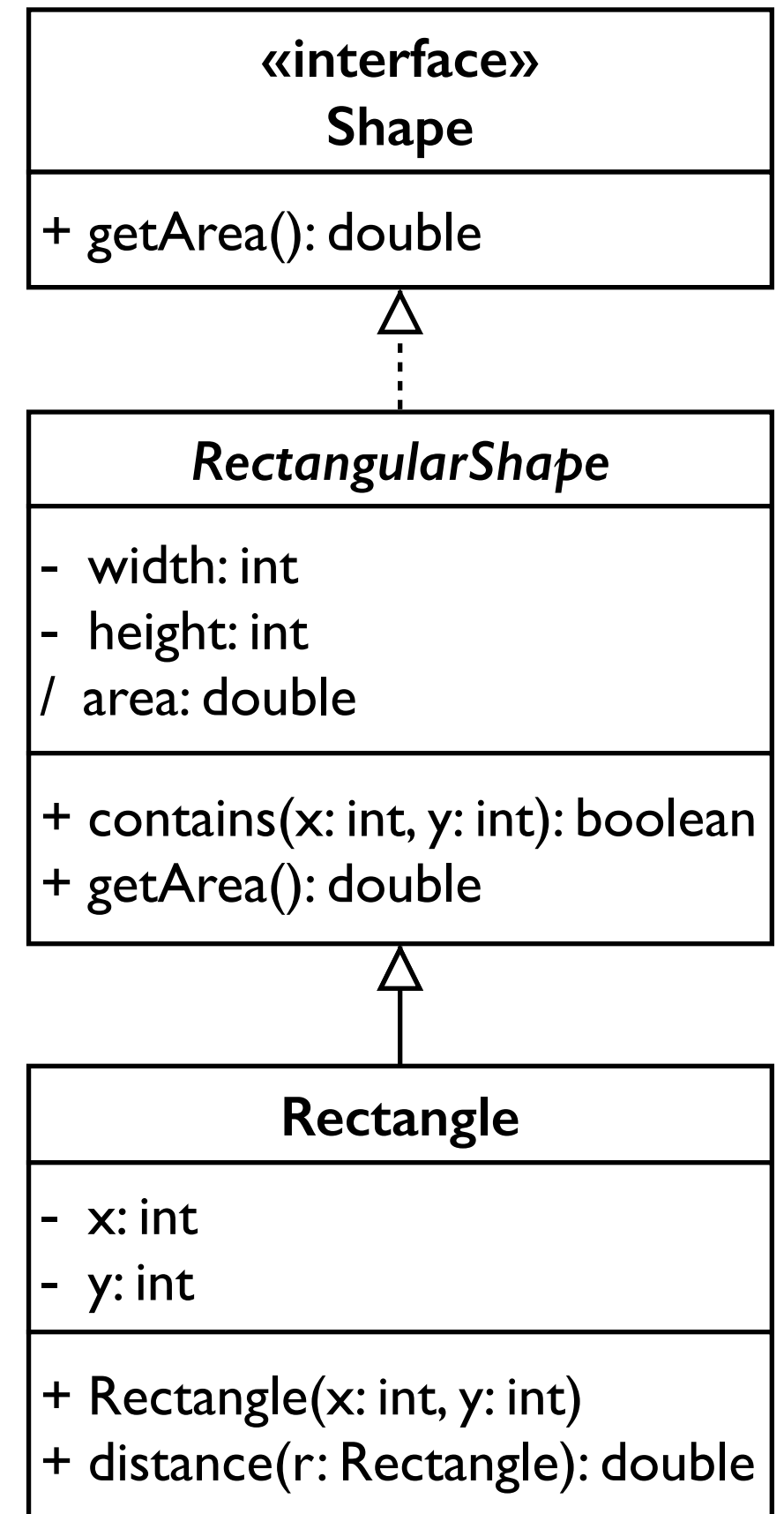
- Hierarchies drawn top-down
- Arrows point upward to parent
- Line/arrow styles indicate if parent is a(n):
  - **class**: solid line, black arrow
  - **abstract class**: solid line, white arrow
  - **interface**: dashed line, white arrow



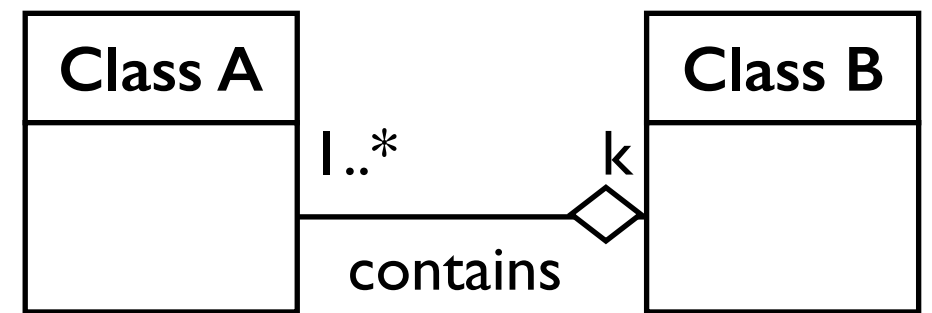


# Generalization relationships

- Hierarchies drawn top-down
- Arrows point upward to parent
- Line/arrow styles indicate if parent is a(n):
  - **class**: solid line, black arrow
  - **abstract class**: solid line, white arrow
  - **interface**: dashed line, white arrow
- Often omit trivial / obvious generalization relationships, such as drawing the Object class as a parent



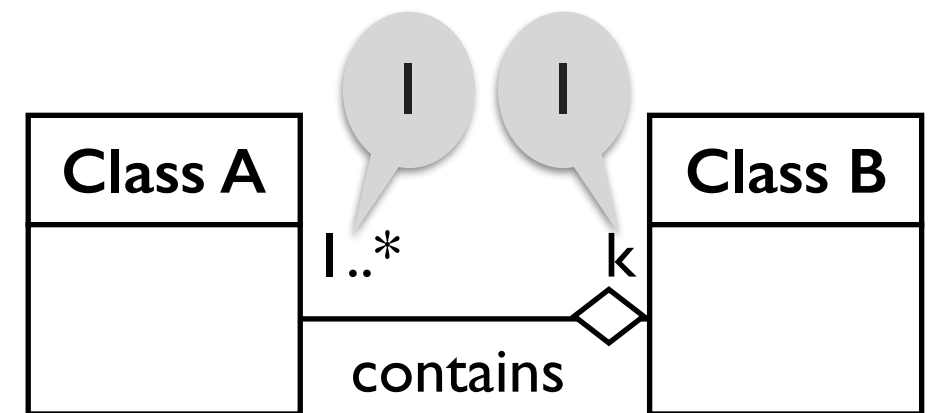
# Associational (usage) relationships



# Associational (usage) relationships

## I. Multiplicity (how many are used)

- \* (zero or more)
- 1 (exactly one)
- 2..4 (between 2 and 4, inclusive)
- 3..\* (3 or more, \* may be omitted)

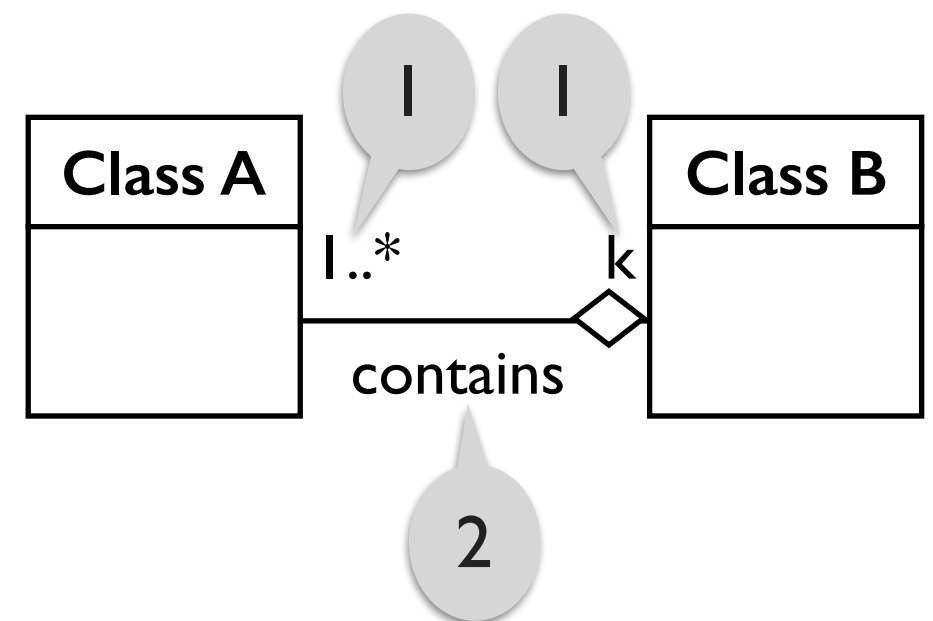


# Associational (usage) relationships

## 1. Multiplicity (how many are used)

- \* (zero or more)
- 1 (exactly one)
- 2..4 (between 2 and 4, inclusive)
- 3..\* (3 or more, \* may be omitted)

## 2. Name (what relationship the objects have)



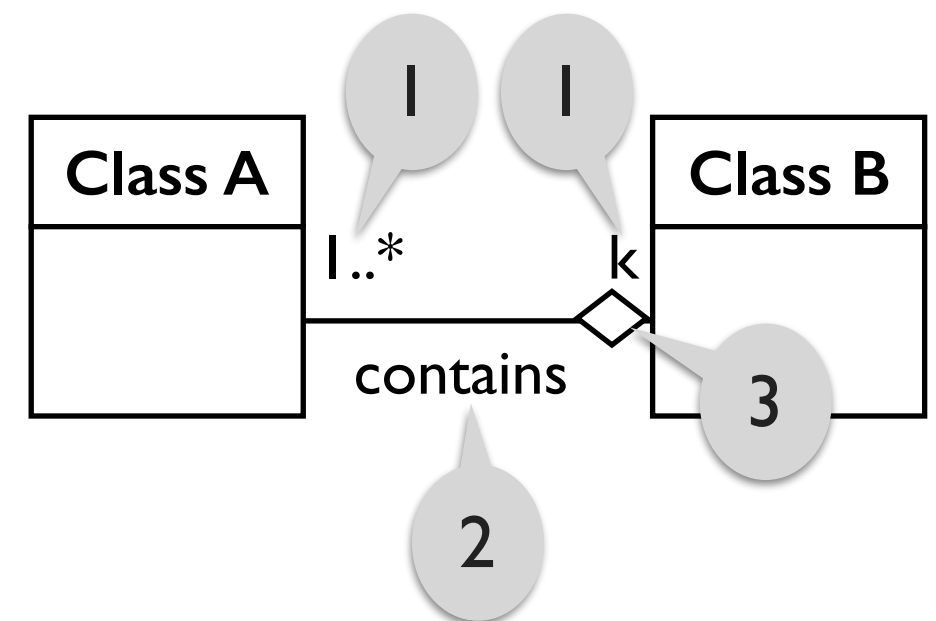
# Associational (usage) relationships

## 1. Multiplicity (how many are used)

- \* (zero or more)
- 1 (exactly one)
- 2..4 (between 2 and 4, inclusive)
- 3..\* (3 or more, \* may be omitted)

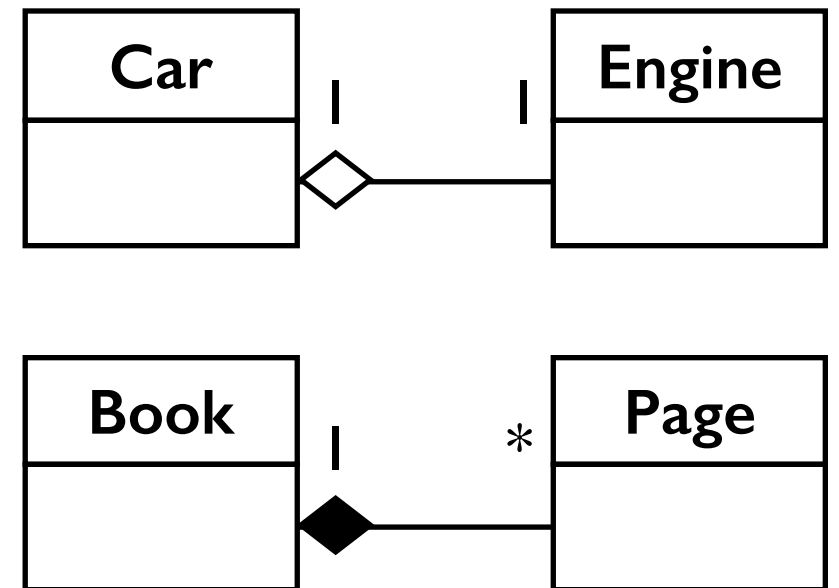
## 2. Name (what relationship the objects have)

## 3. Navigability (direction)

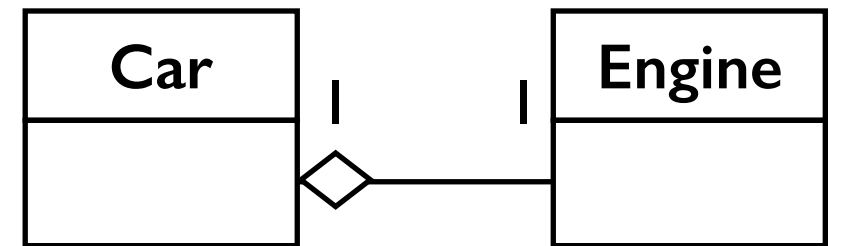


# Association multiplicities

- **One-to-one**
  - Each car has exactly one engine.
  - Each engine belongs to exactly one car.
- **One-to-many**
  - Each book has many pages.
  - Each page belongs to exactly one book.

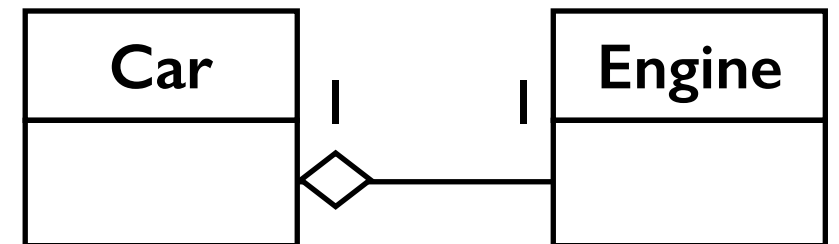


# Association types



# Association types

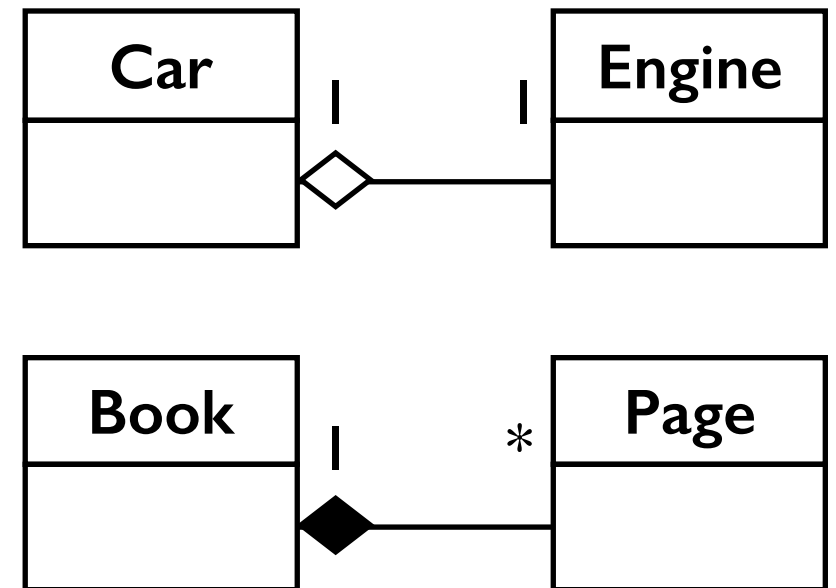
- **Aggregation:** “is part of”
  - symbolized by a clear white diamond





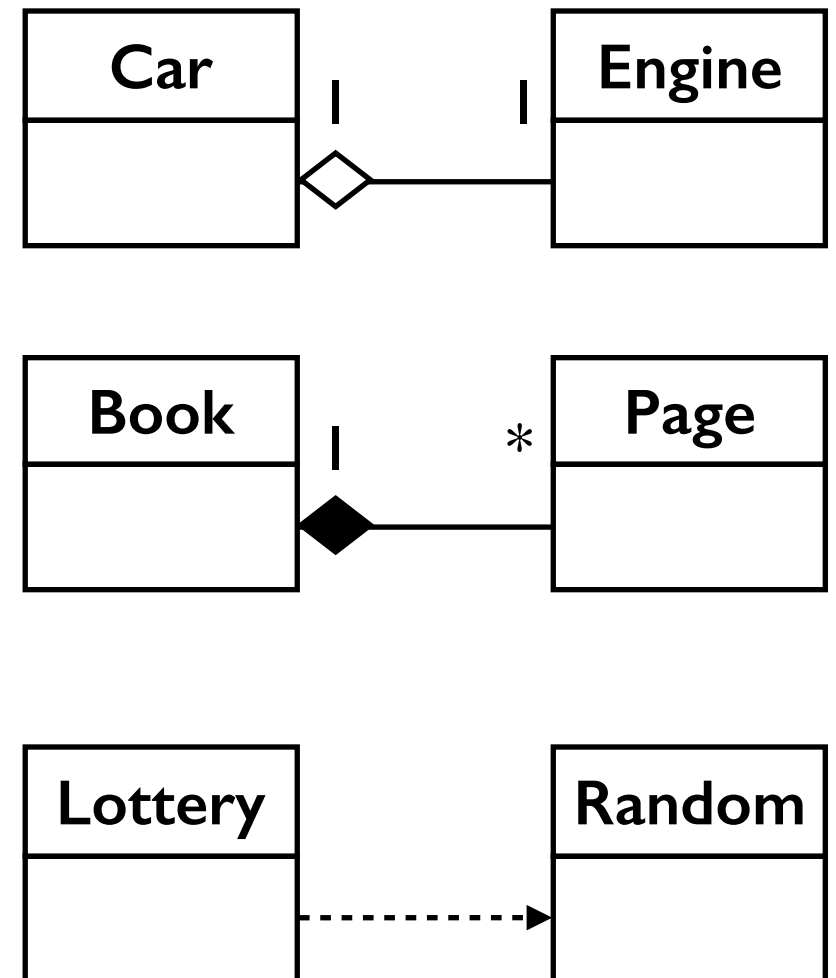
# Association types

- **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



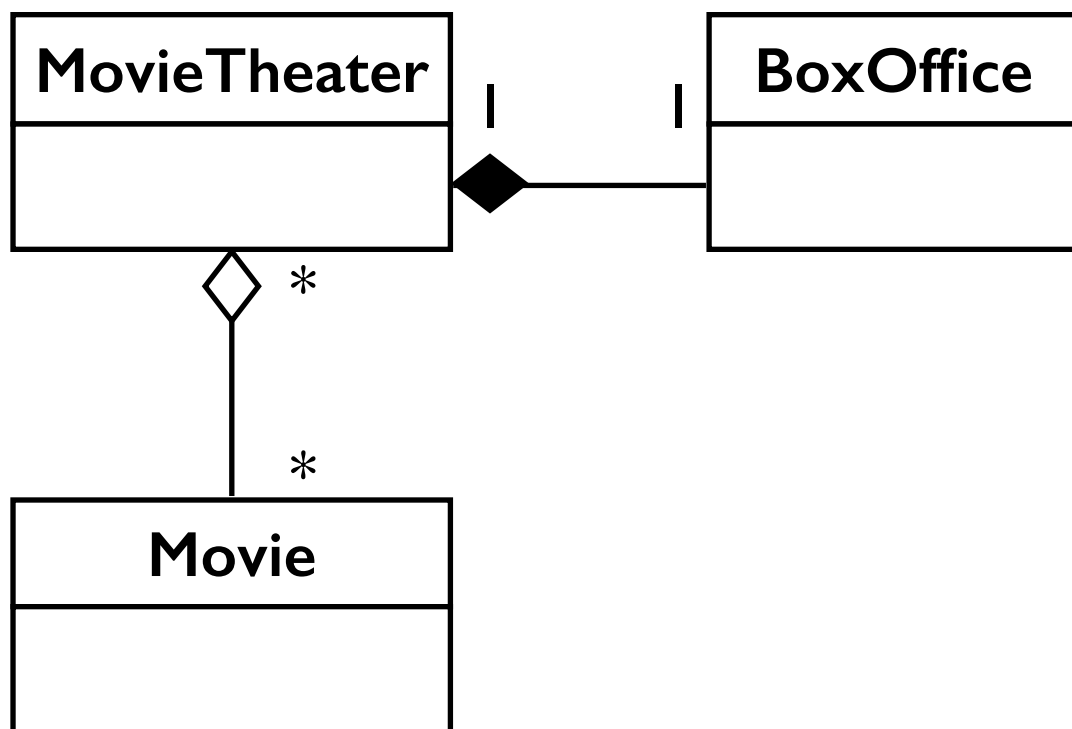
# Association types

- **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 the object's state

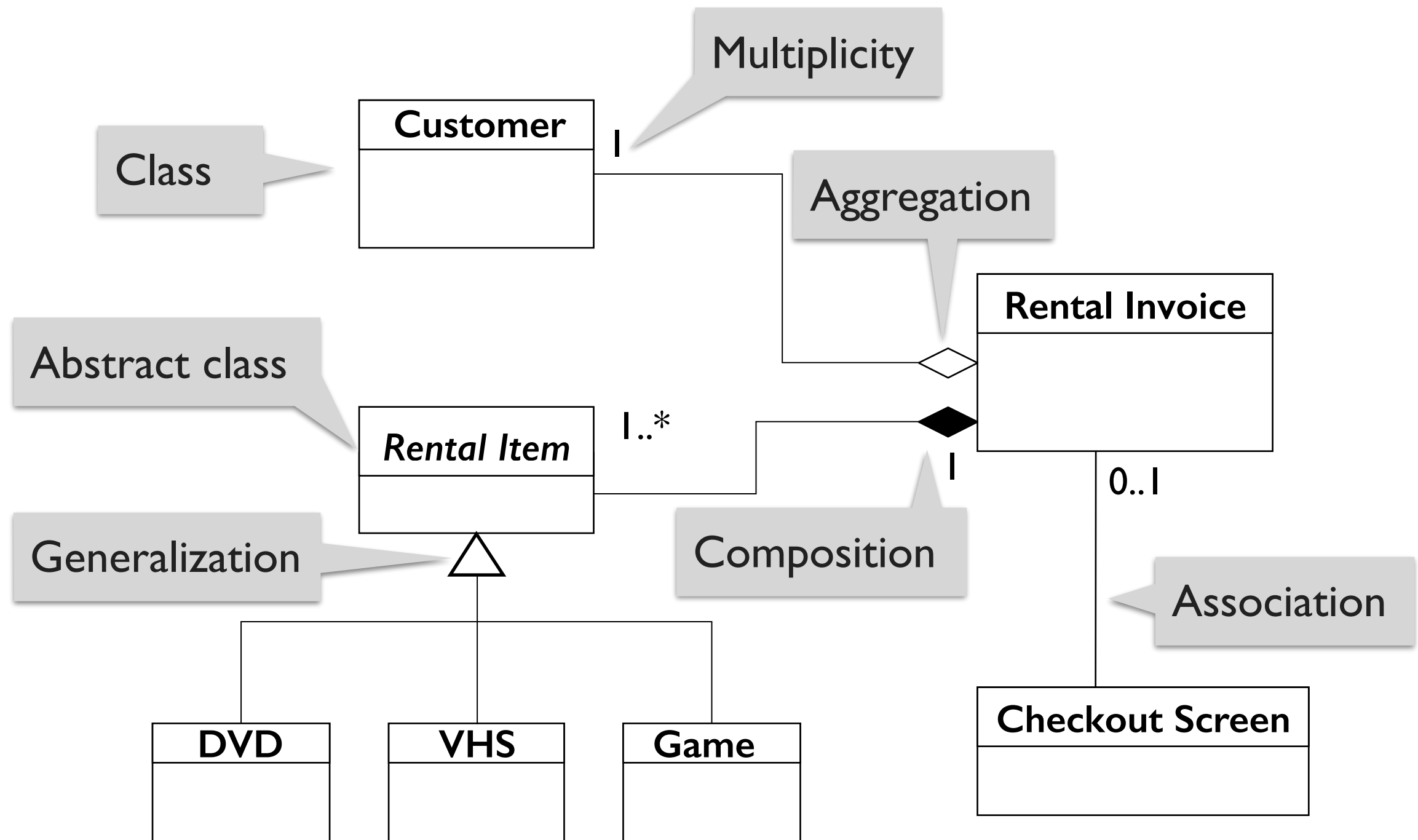


# Aggregation / composition example

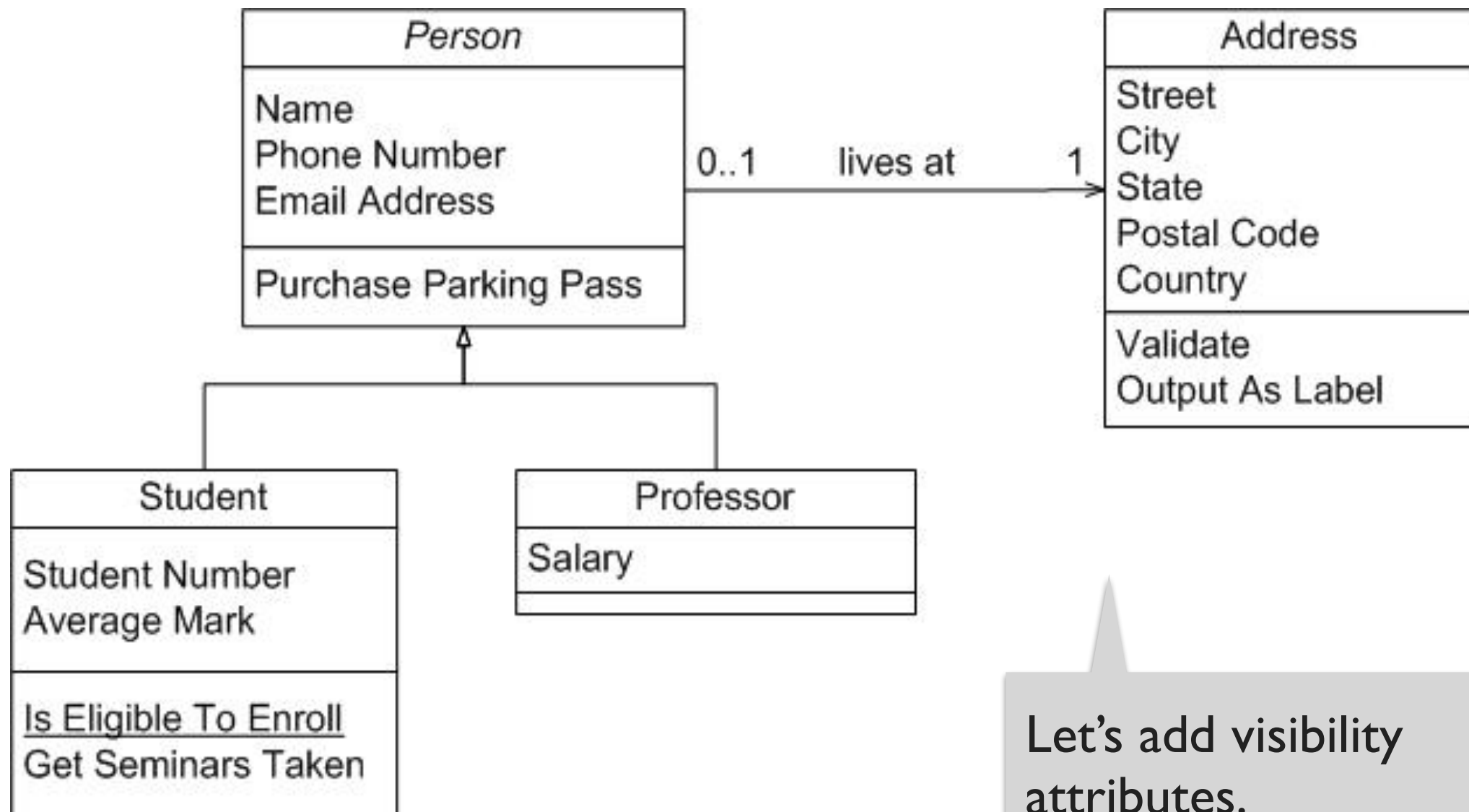
- If the movie theater goes away
  - so does the box office: composition
  - but movies may still exist: aggregation



# Class diagram example: video store

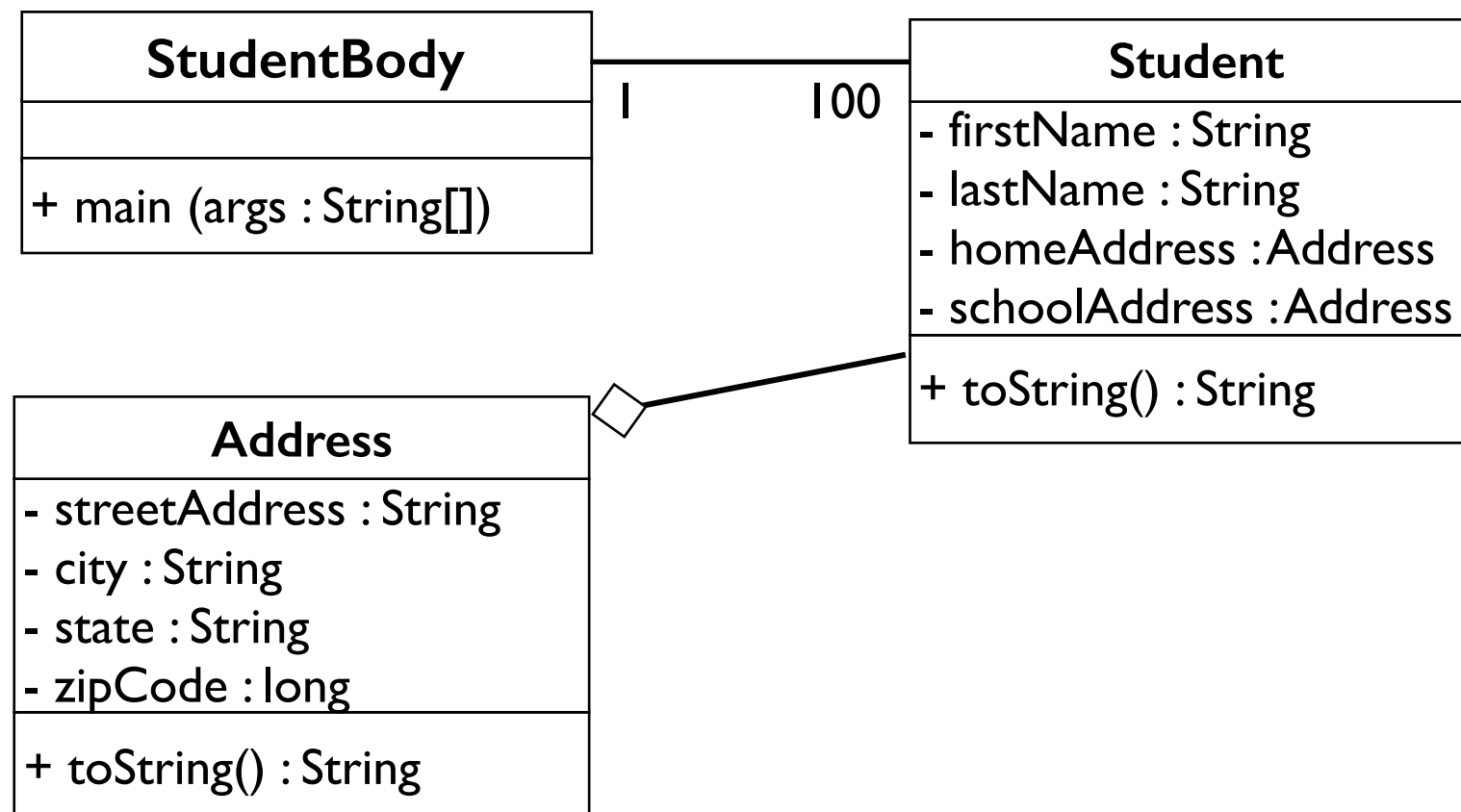


# Class diagram example: people



Let's add visibility attributes.

# Class diagram example: student



# Tools for creating UML diagrams

- Violet (free)
  - <http://horstmann.com/violet/>
- Rational Rose
  - <http://www.rational.com/>
- Visual Paradigm UML Suite (trial)
  - <http://www.visual-paradigm.com/>
- There are many others, but most are commercial

# **What (not) to use class diagrams for**



# What (not) to use class diagrams for

- Class diagrams are great for:
  - discovering related data and attributes
  - getting a quick picture of the important entities in a system
  - seeing whether you have too few/many classes
  - seeing whether the relationships between objects are too complex, too many in number, simple enough, etc.
  - spotting dependencies between one class/object and another

# What (not) to use class diagrams for

- Class diagrams are great for:
  - discovering related data and attributes
  - getting a quick picture of the important entities in a system
  - seeing whether you have too few/many classes
  - seeing whether the relationships between objects are too complex, too many in number, simple enough, etc.
  - spotting dependencies between one class/object and another
- Not so great for:
  - discovering algorithmic (not data-driven) behavior
  - finding the flow of steps for objects to solve a given problem
  - understanding the app's overall control flow (event-driven? web-based? sequential? etc.)

# Summary

- A design specifies the structure of how a software system will be written and function.
- UML is a language for describing various aspects of software designs.
- UML class diagrams present a static view of the system, displaying classes and relationships between them.

