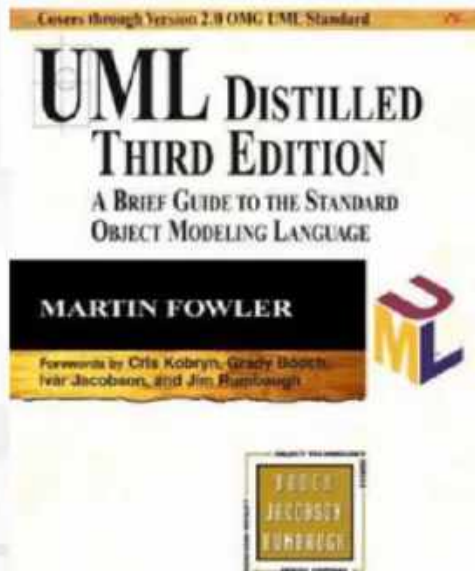




UML Class Diagram



Reading:
UML Distilled Ch. 3, by M. Fowler

These lecture slides are copyright (C) Marty Stepp, 2007. They may not be rehosted, sold, or modified without expressed permission from the author. All rights reserved.



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?





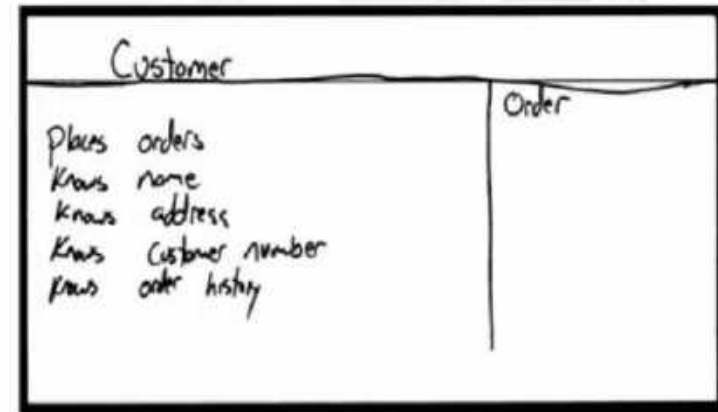
Design phase

- **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
 - ...





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



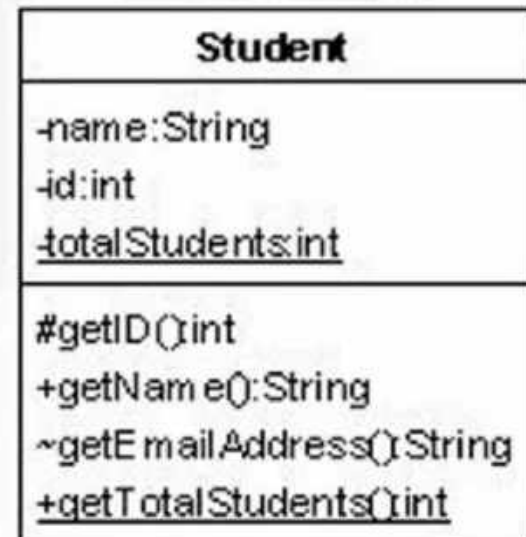
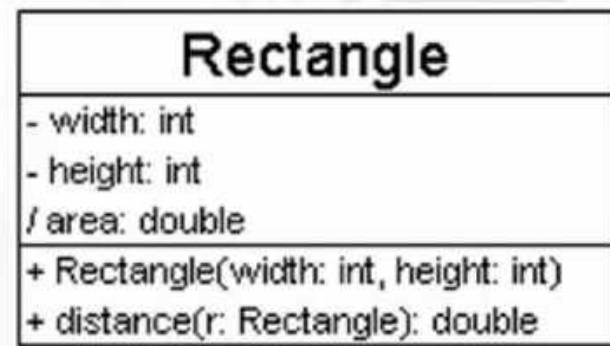
UML class diagrams

- What is a UML class diagram?
 - **UML class diagram**: a picture of the classes in an OO system, their fields and methods, and connections between the classes that interact or inherit from each other
- What are some things that are 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 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





Class attributes

- 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
 - attribute example:
 - balance : double = 0.00

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

Student
-name:String
-id:int
<u>-totalStudents:int</u>
#getID()int
+getName():String
~getEmailAdress()String
<u>+getTotalStudents()int</u>



Class operations / methods

- operations / methods
 - *visibility name (parameters) : return_type*
 - visibility: + public
 # protected
 - private
 ~ package (default)
 - underline static methods
 - 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

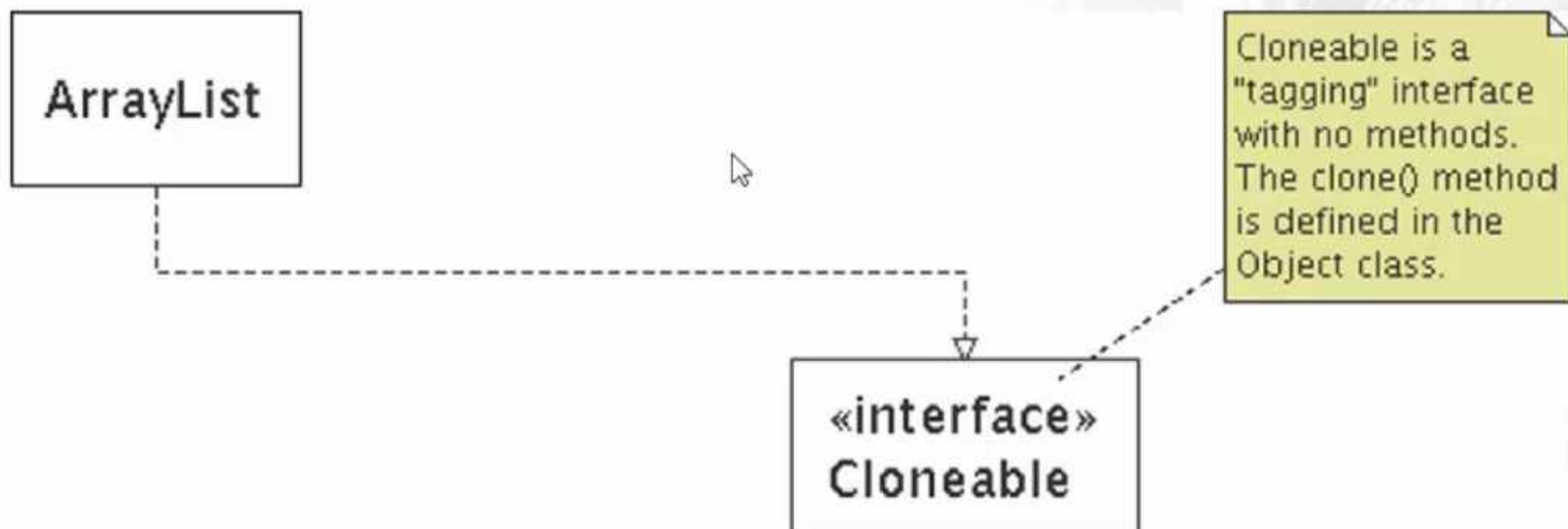
Rectangle
- width: int
- height: int
/ area: double
+ Rectangle(width: int, height: int)
+ distance(r: Rectangle): double

Student
-name:String
-id:int
<u>-totalStudents:int</u>
#getID()int
+getNam e():String
~getEmailAdress()String
<u>+getTotalStudents()int</u>



Comments

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





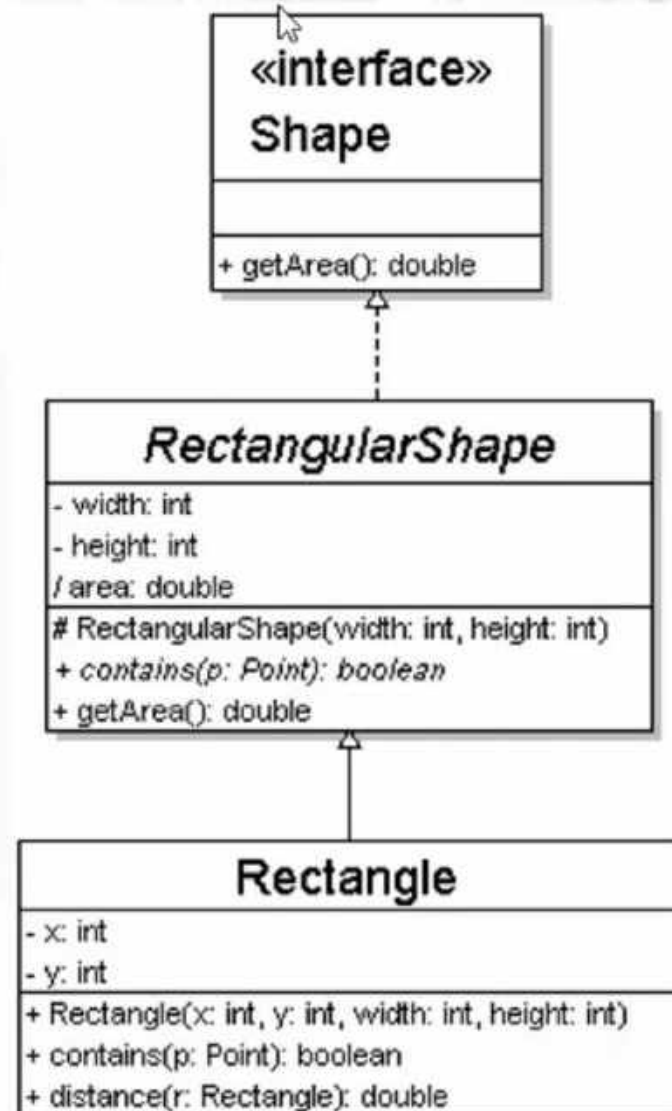
Relationships btwn. classes

- **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):
 - class:
solid line, black arrow
 - abstract class:
solid line, white arrow
 - interface:
dashed line, white arrow
 - we often don't draw trivial / obvious generalization relationships, such as drawing the Object class as a parent





Associational relationships

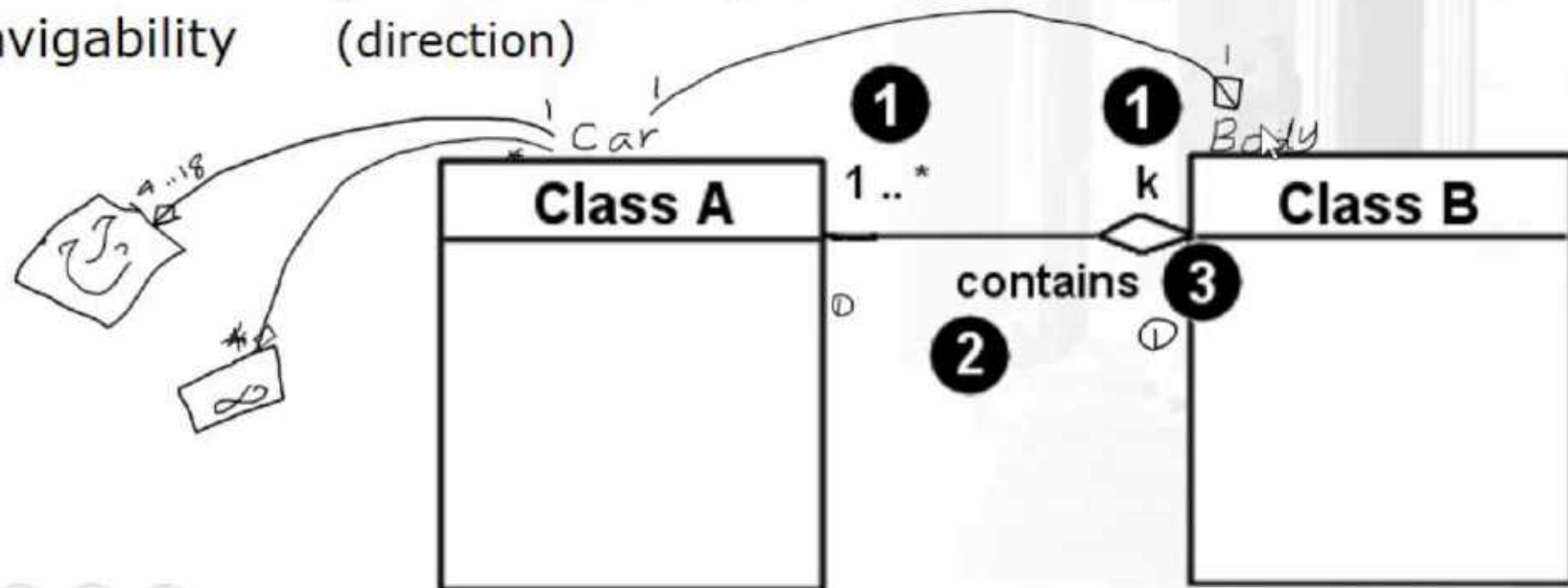
■ associational (usage) relationships

1. multiplicity (how many are used)

- * \Rightarrow 0, 1, or more
- ✓ • 1 \Rightarrow 1 exactly
- ✓ • 2..4 \Rightarrow between 2 and 4, inclusive
- 3..* \Rightarrow 3 or more

2. name (what relationship the objects have)

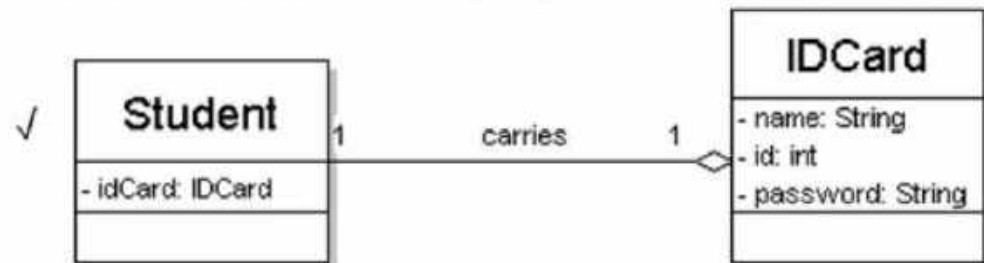
3. navigability (direction)



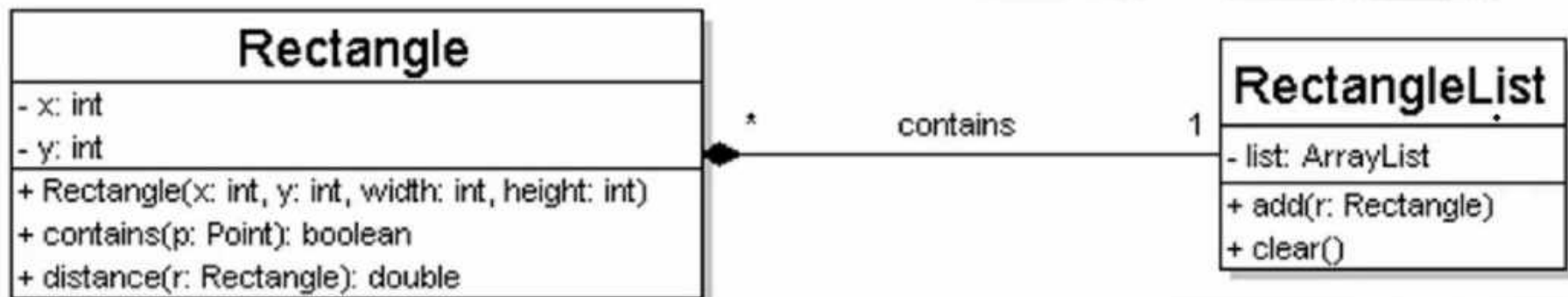


Multiplicity of associations

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

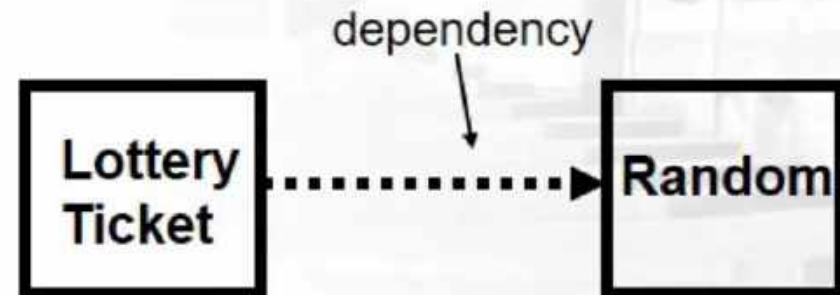
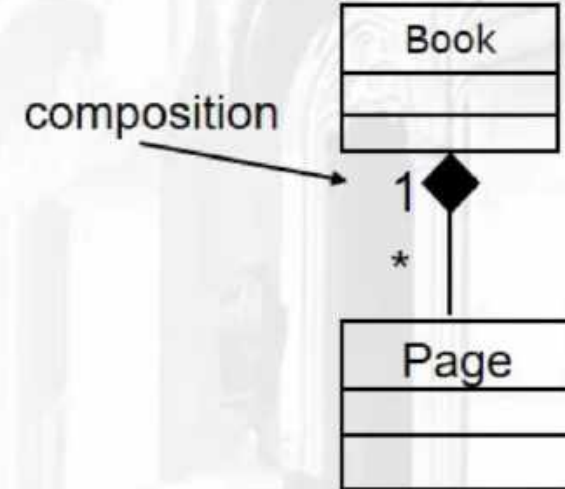
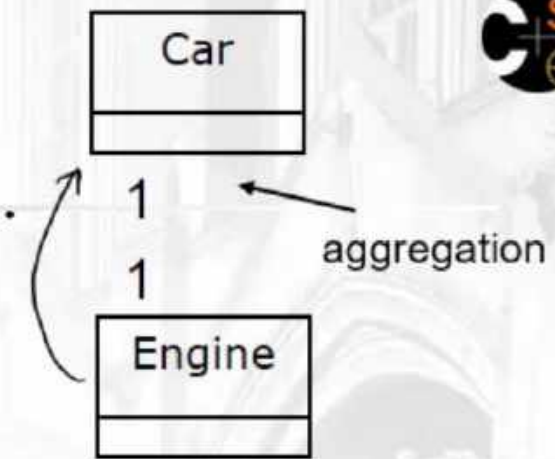


- one-to-many
 - one rectangle list can contain many rectangles

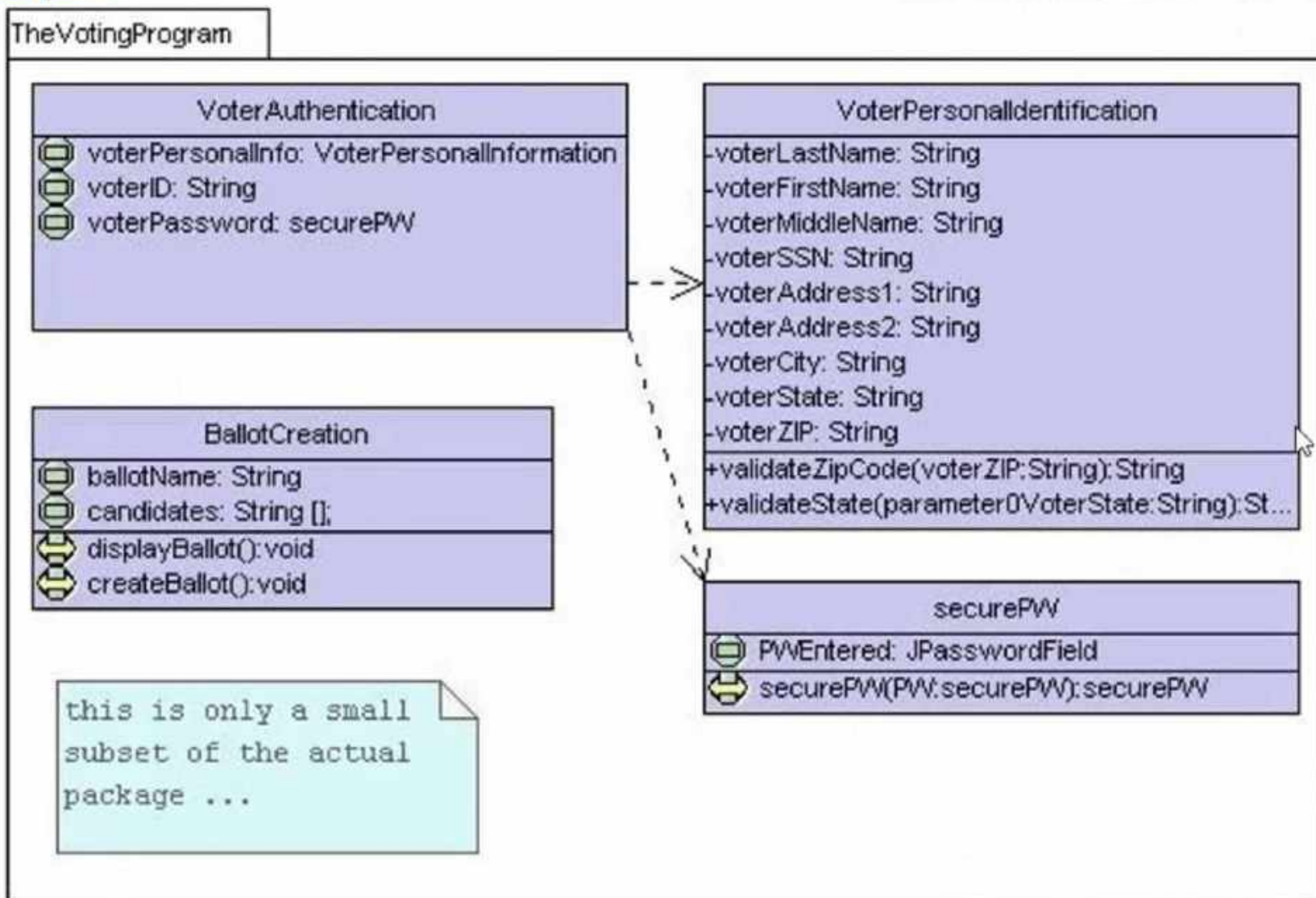


Association types

- **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

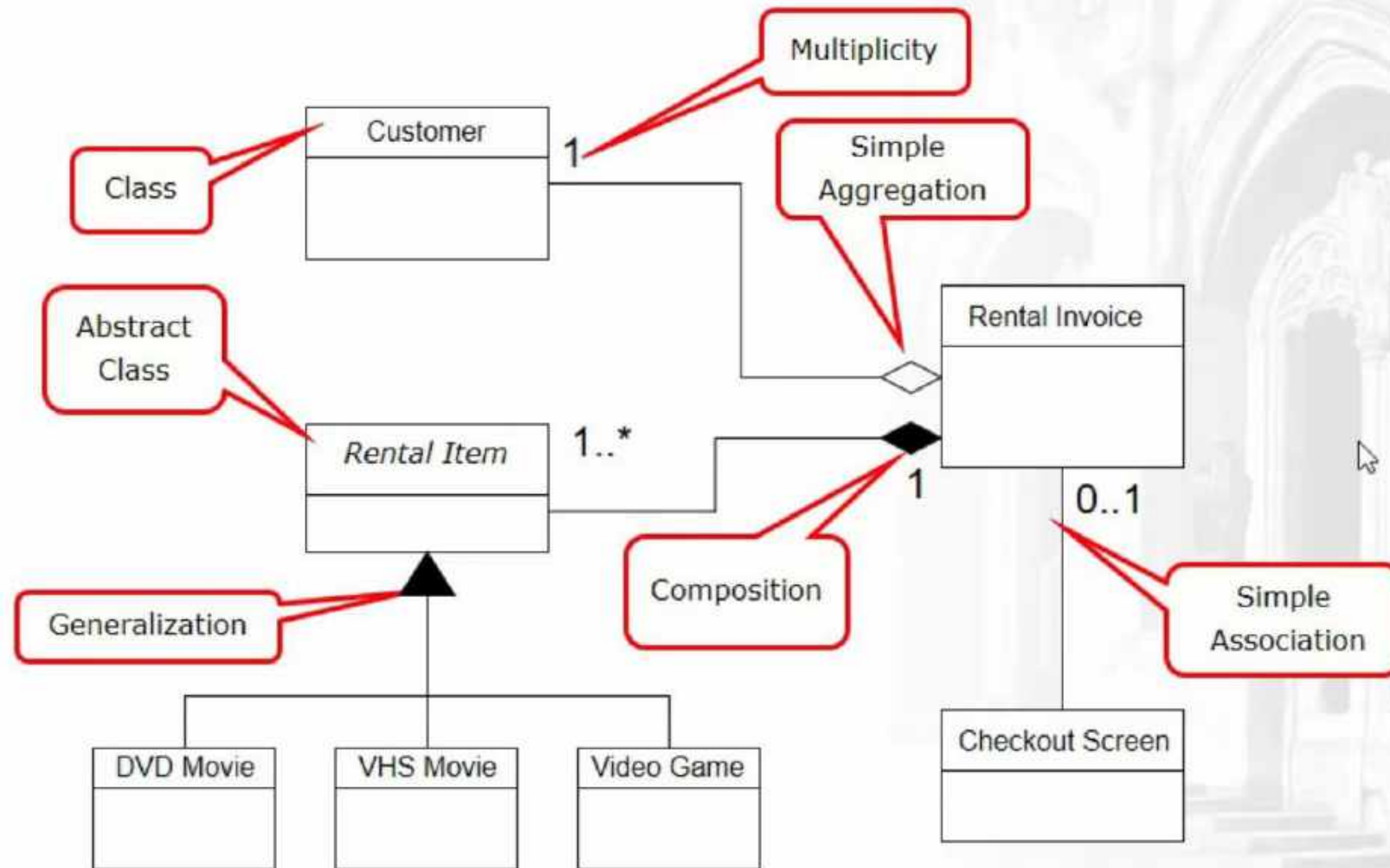


Class diagram example 1





Class diagram example 2





Class diagram example 3

