# Software Architecture

## Lecture 5

## System Modeling

Néstor Cataño

Innopolis University

Spring 2016

# Today's Topics and Sources

1. Introduction to System Modeling and UML
2. Class Diagrams
3. Class Diagrams & OCL
4. State Machine Diagrams
5. Activity Diagrams
6. Sequence Diagrams


## Lecture Sources

- UML Distilled: A Brief Guide to the Standard Object Modelling Language, 3rd edition, by Martin Fowler.
- UML 2.1. Specification -- http://www.omg.org/spec/UML/2.1.2/

# System modeling

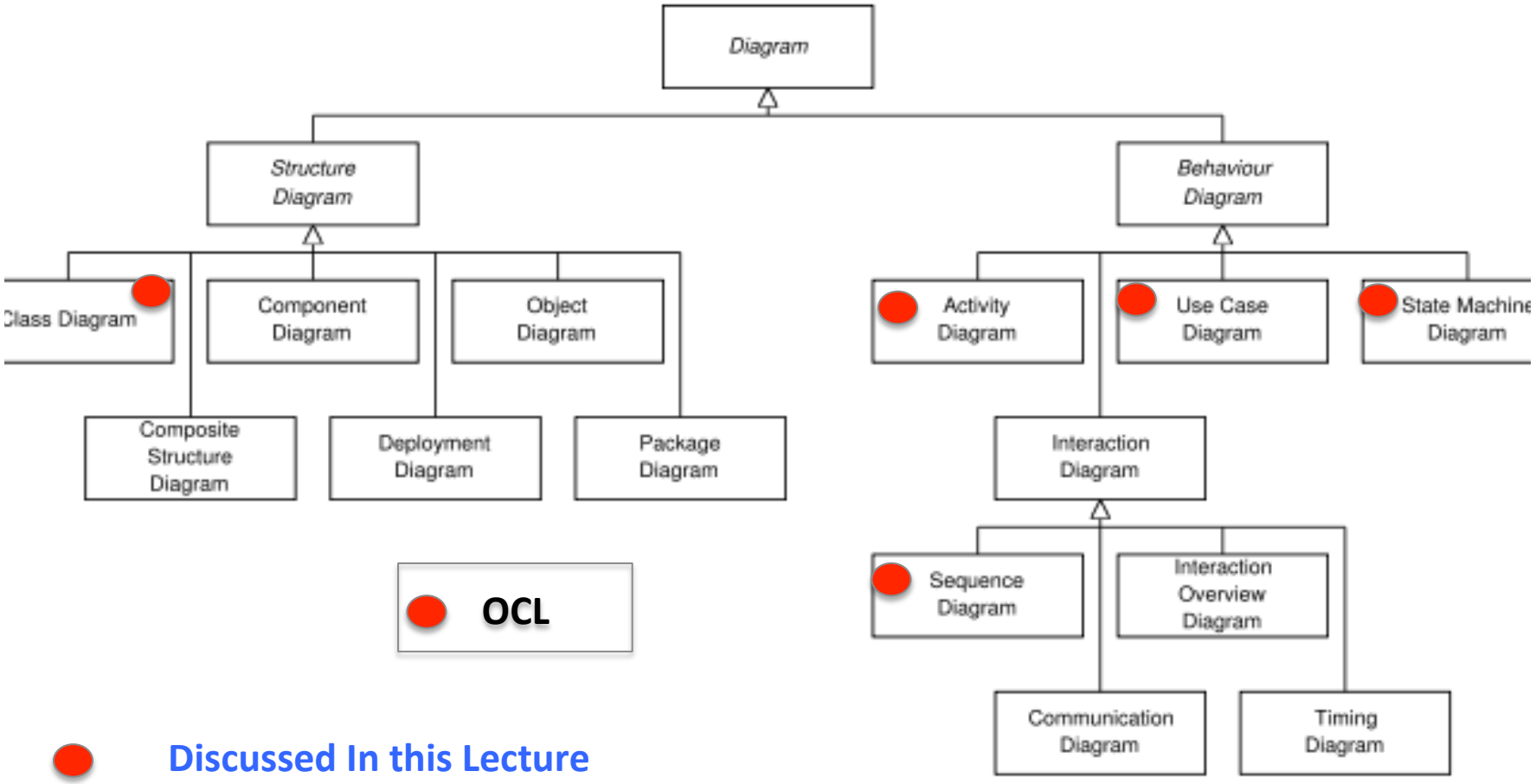
## Goal

- "to introduce some types of **system models** that may be developed as part of the **requirements engineering** and **system design** processes"

# What is UML (Unified Modeling Language) ?

- It defines a set of formal languages
  - **Syntax and semantics**
- Has a "formal" meta-model (in UML)
  - **UML Infrastructure: core meta-model**
  - **UML Superstructure: notation and semantics**
- Other specifications
  - **Object Constraint Language**
  - **Diagram Interchange Specification**
- Has some built-in extension mechanisms
  - **Stereotypes: create new model elements**
  - **Stereotype attributes or tagged values**
  - **Profiles: collections of extensions**

# What is UML?
# UML Diagram Types




**Discussed In this Lecture**

**Diagram derived from Wikipedia**

# What is UML?

## Structural Modeling

- **Class Diagrams** – Conceptual Model / Domain Model, Type Structure
- **Package diagrams** – model management
- **Deployment diagrams** – allocation to hardware
- **Object diagrams** – software units at runtime

# What is UML?

- **Behavioral modeling notations**
  - **State machine diagrams** (**Statecharts**) – states of objects
  - **OCL** – predicates
  - **Activity diagrams** – high level business dataflow
  - **Interaction diagrams**
    - **Sequence diagrams** – focus on time
    - **Communication diagrams** – focus on structure (known as **collaboration diagrams** in UML 1.x)

# Outline

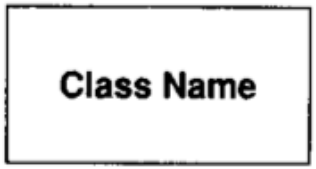
1. Introduction to System Modeling and UML
2. **Class Diagrams**
3. Class Diagrams & OCL
4. State Machine Diagrams
5. Activity Diagrams
6. Sequence Diagrams

# Class Diagrams

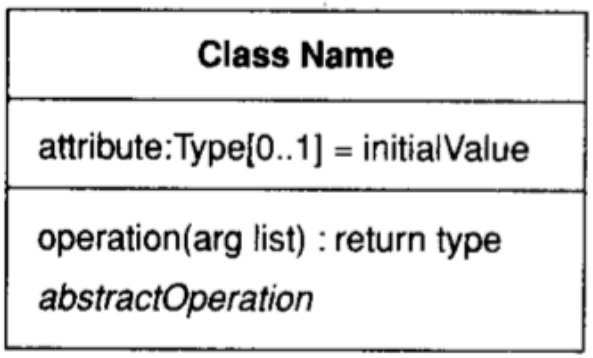
- If someone were to come up to you in a dark alley and say "**I want to see a UML diagram?**", that diagram would probably be a **class diagram**
- Class diagrams describe the types of **objects** in a system and the kinds of static **relationships** that exist among them.
- Class diagrams also show **properties** and operations of a class and the constraints that apply to the way objects are connected (**OCL**).

# Class Diagrams: Notation

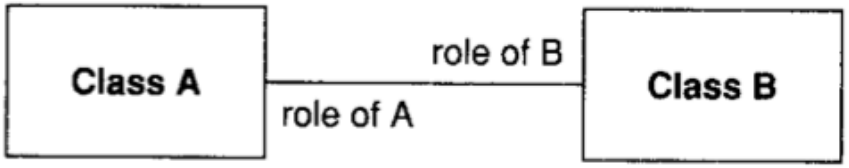
**Class**



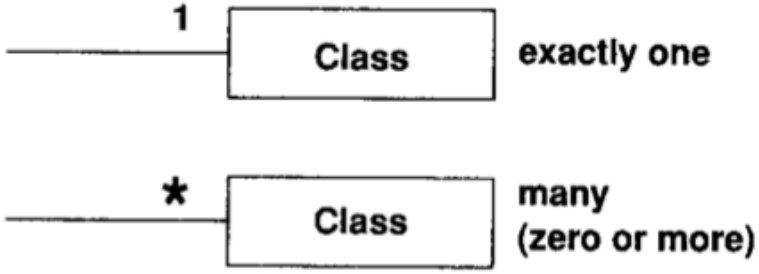

*p. 35*




**Association** *p. 37*

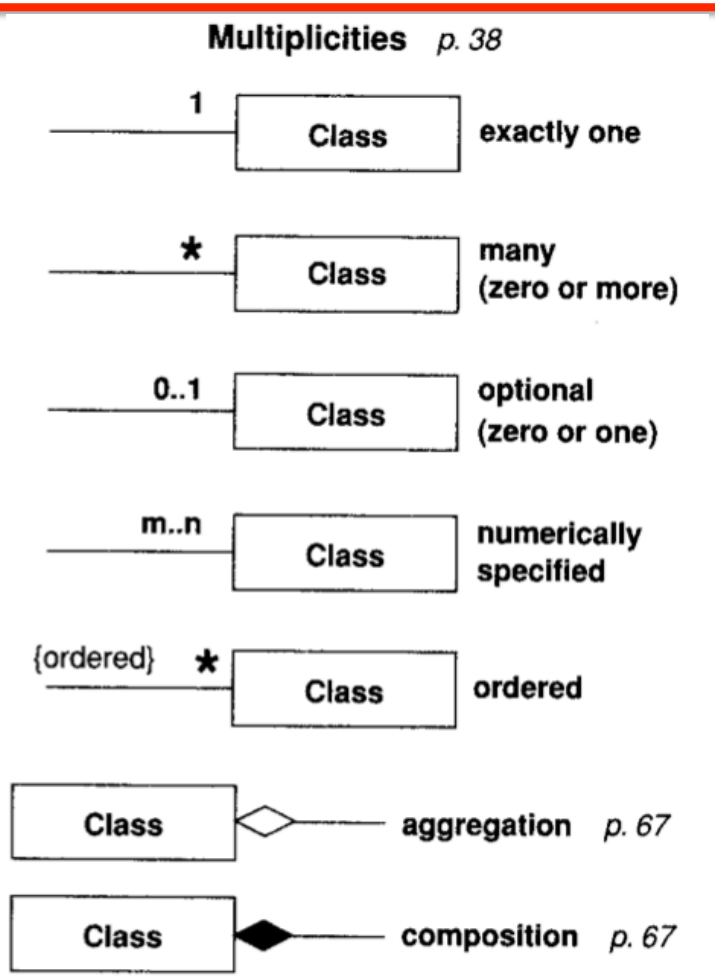



**Multiplicities** *p. 38*




**Take from UML Distilled**

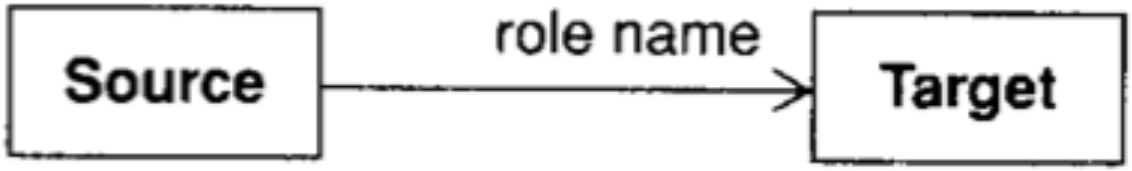
# Class Diagrams: Notation




Take from UML Distilled

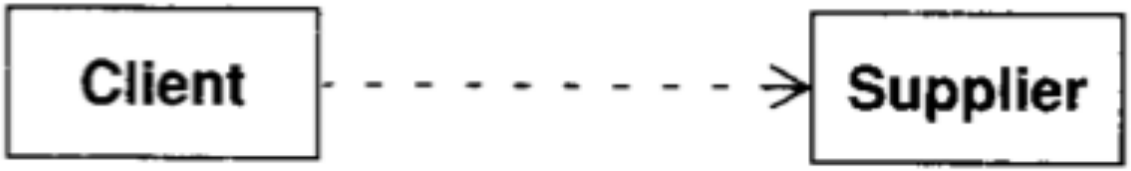
# Class Diagrams: Notation

**Navigability** *p. 42*

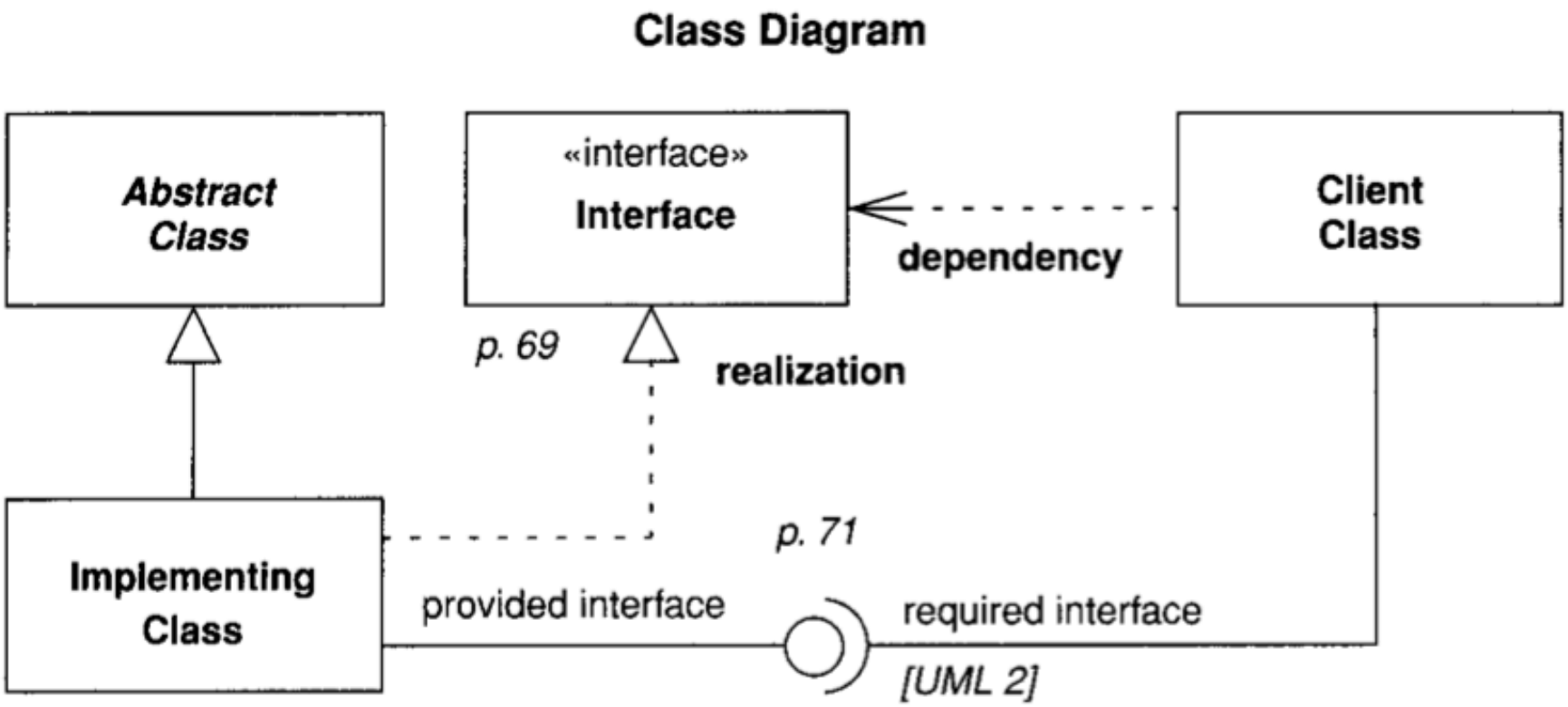



Take from UML Distilled

**Dependency** *p. 47*

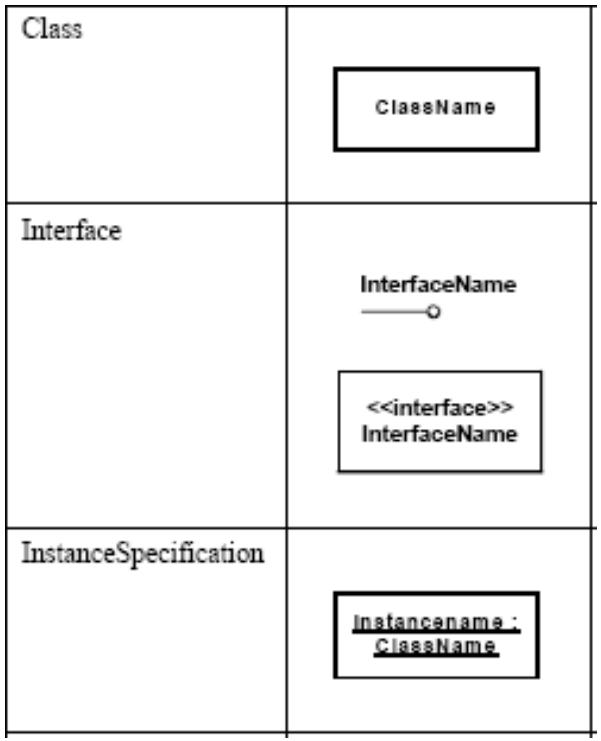


# Class Diagrams: Notation

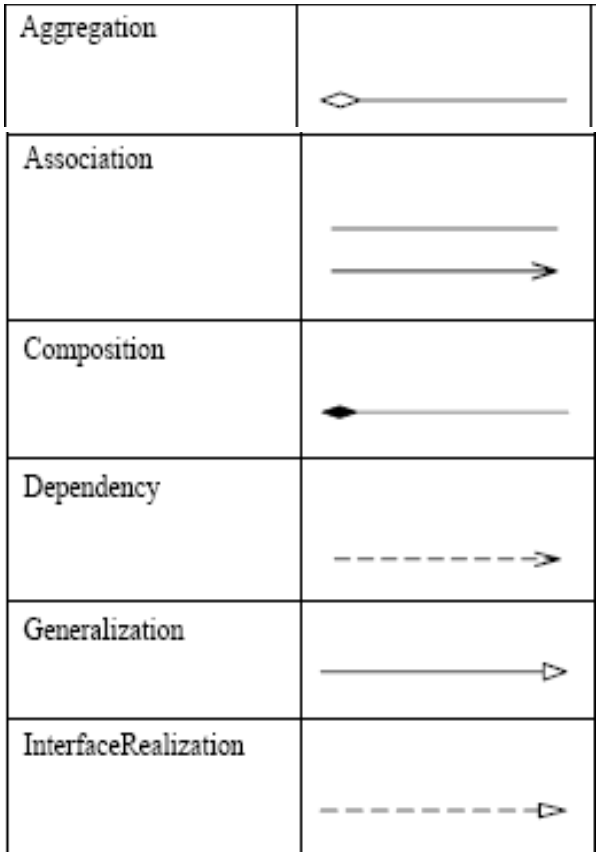


# Class Diagrams: Notation

- **Nodes**

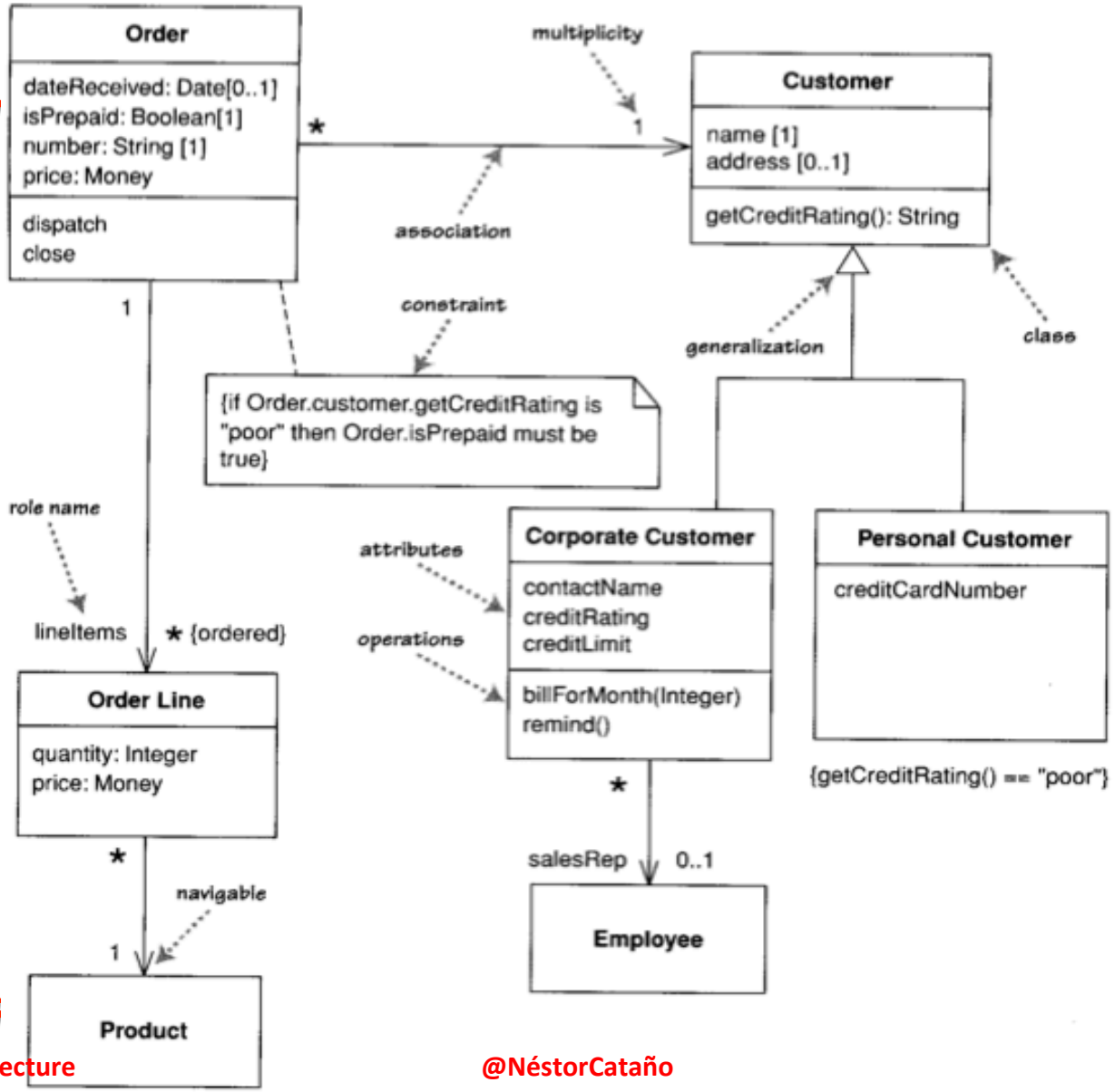



- **Edges**



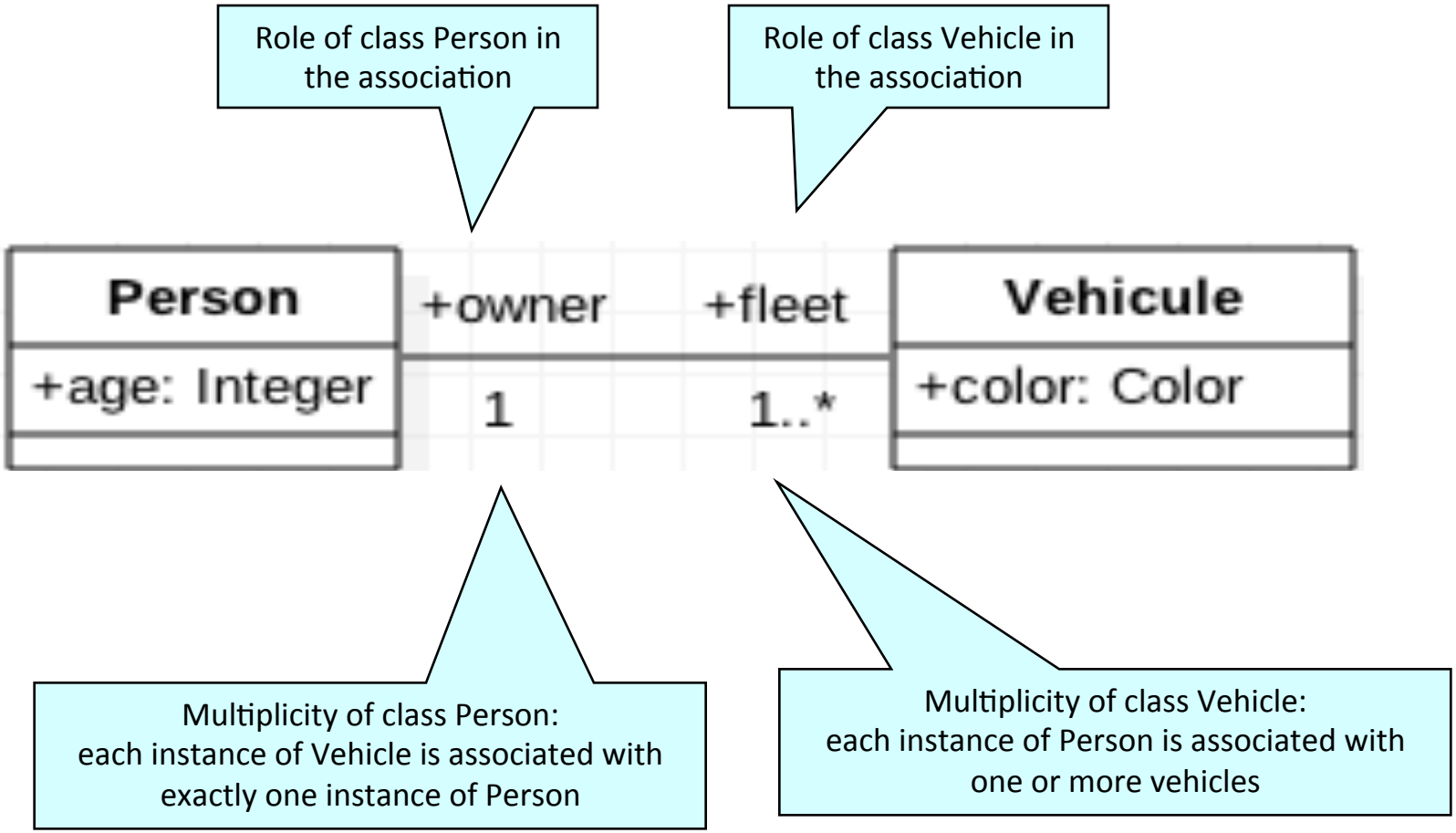

**From: Unified Modeling Language: Superstructure, version 2.0**

# Class Diagrams: Complete Example



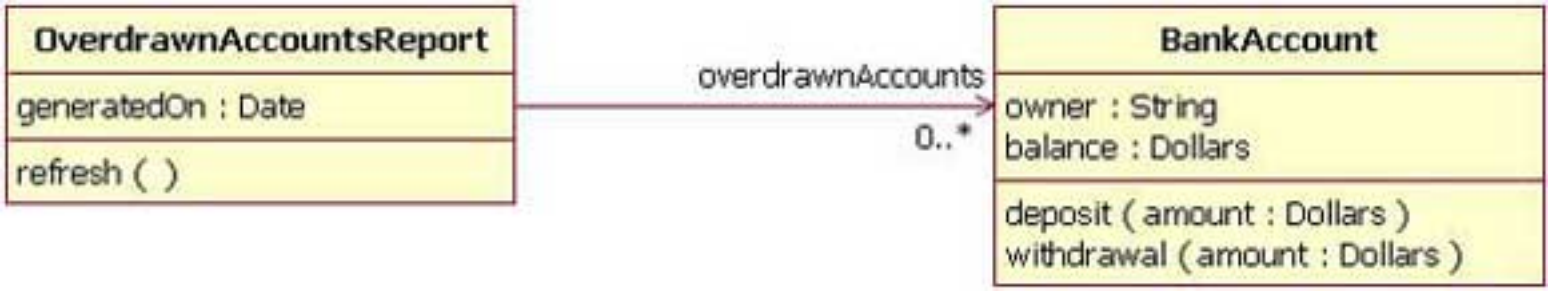
# Class Diagrams

- **Example**



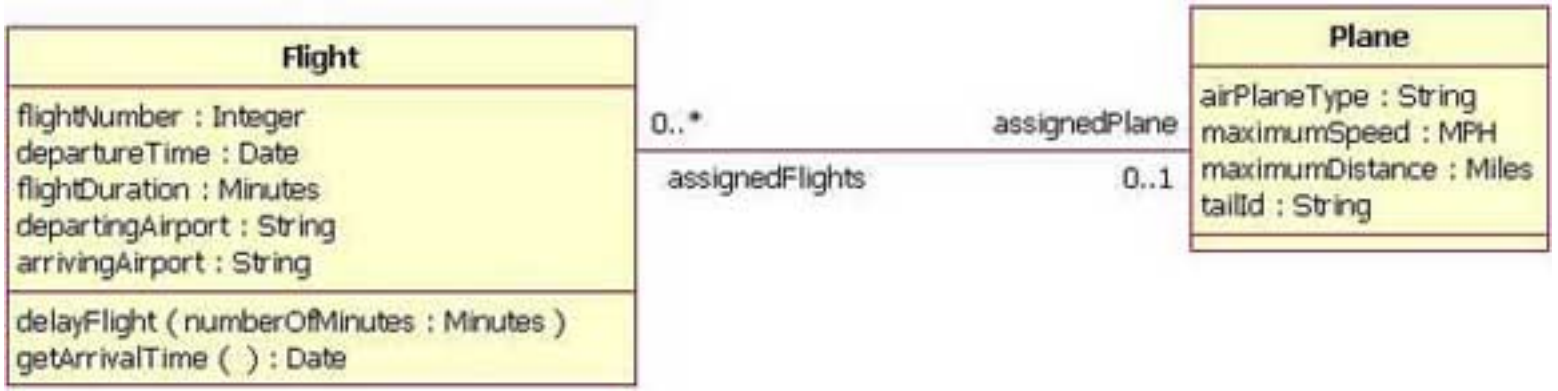
# Unidirectional associations

**The OverdrawnAccountsReport class knows about the BankAccount class, but the BankAccount class does not know about the association**




- If an instance of a **OverdrawnAccountsReport** exists, then it's associated to **0** or **more** instances of a **BankAccount**

# Bidirectional associations

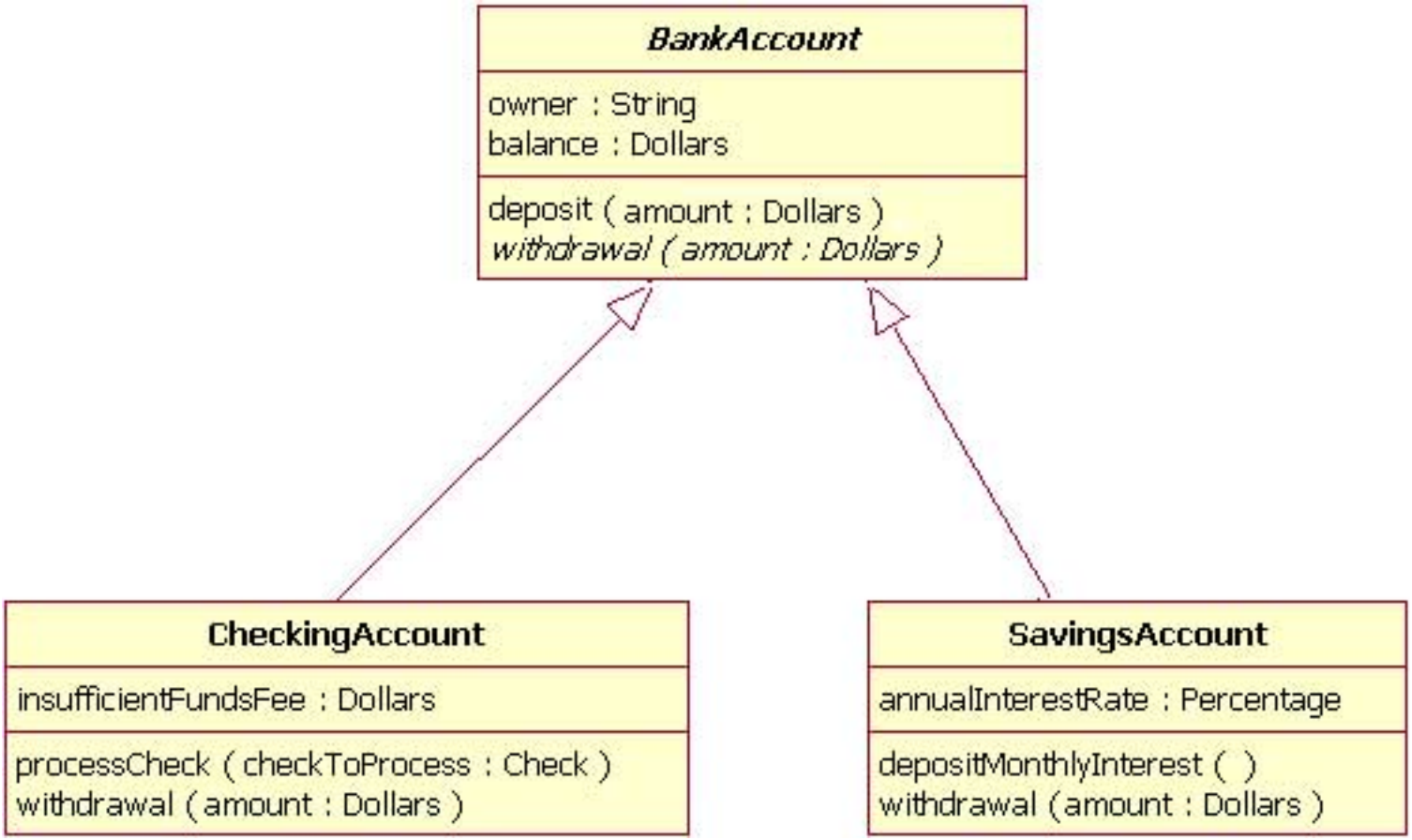



- If an instance of a **Flight** exists, then it's associated to 0 or 1 instances of a **Plane**
- If an instance of **Plane** exists, then it's associated to 0 or more instances of a **Flight**

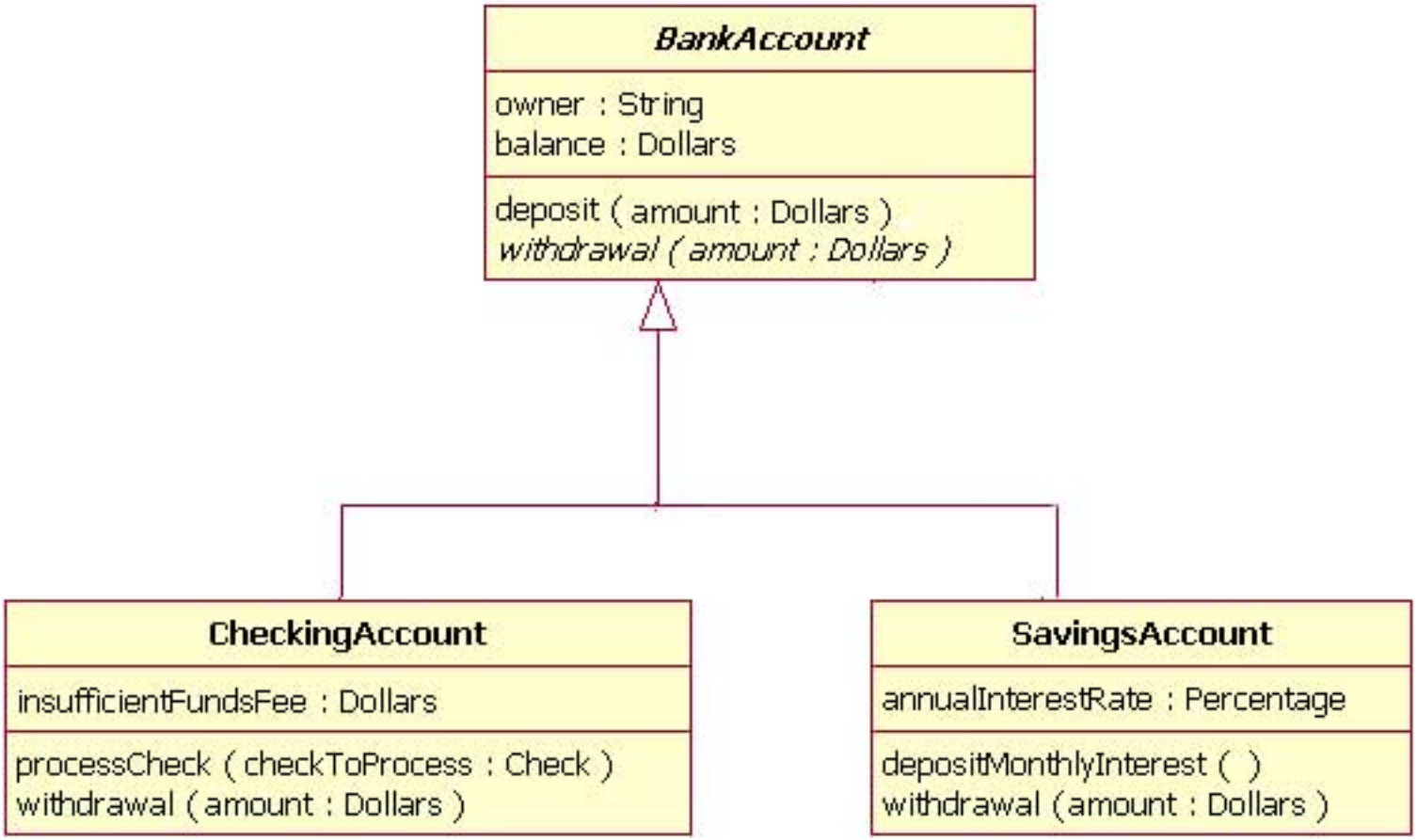
# Multiplicity values and indicators

| Indicator | Meaning |
|---|---|
| 0..1 | Zero or one |
| 1 | One only |
| 0..* | Zero or more |
| * | Zero or more |
| 1..* | One or more |
| 3 | Three only |
| 0..5 | Zero to five |
| 5..15 | Five to Fifteen |

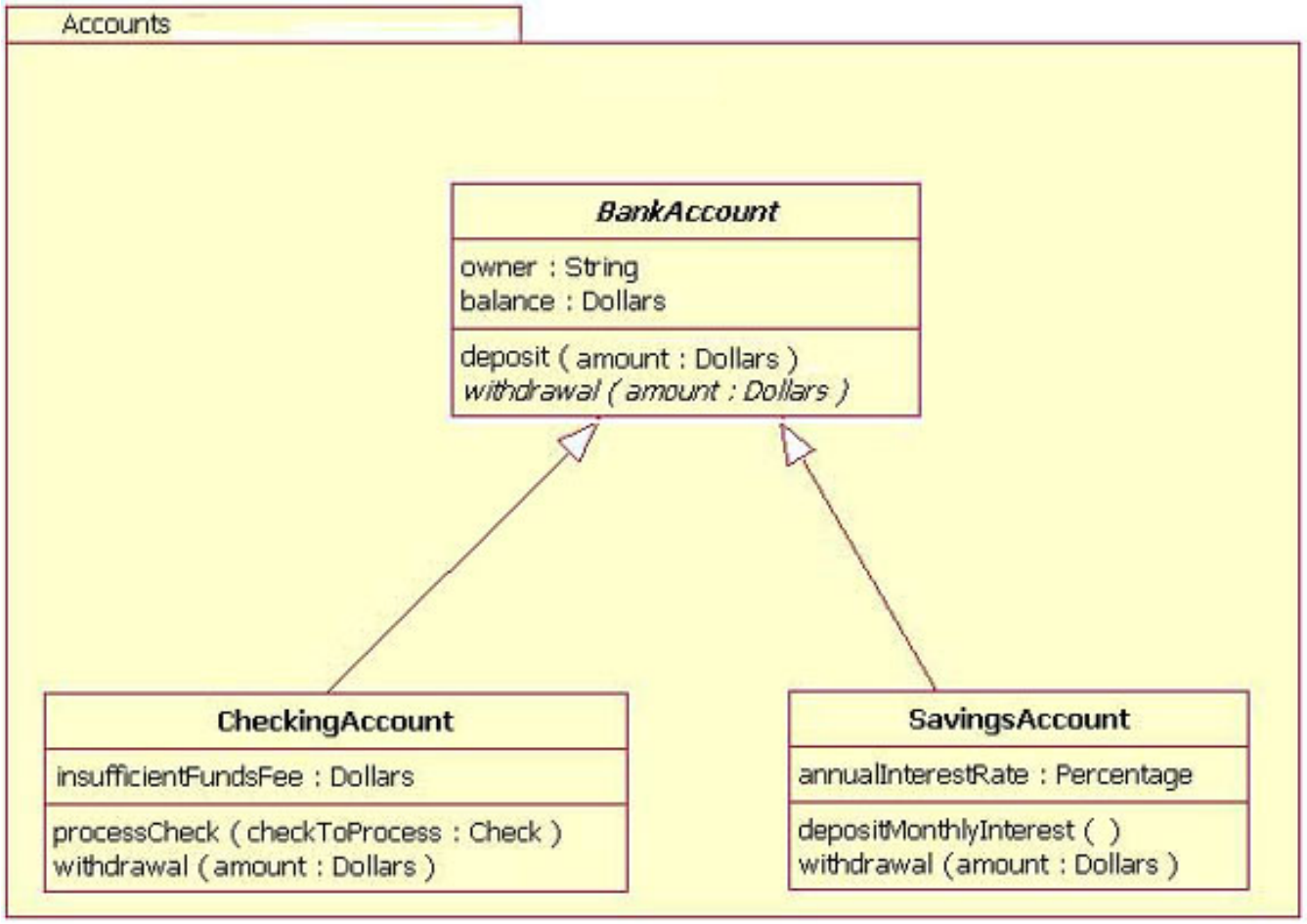
# Inheritance



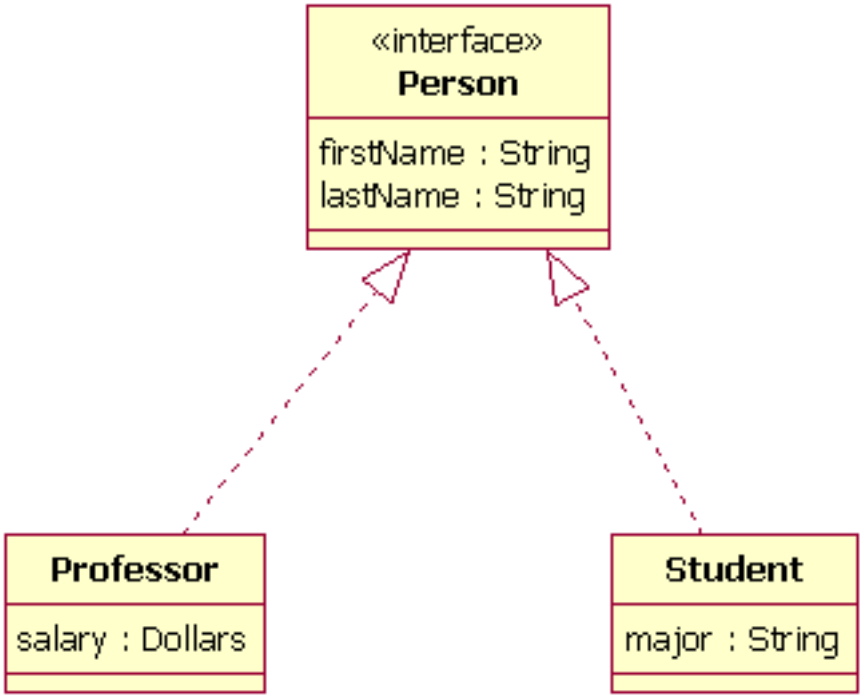
# Inheritance



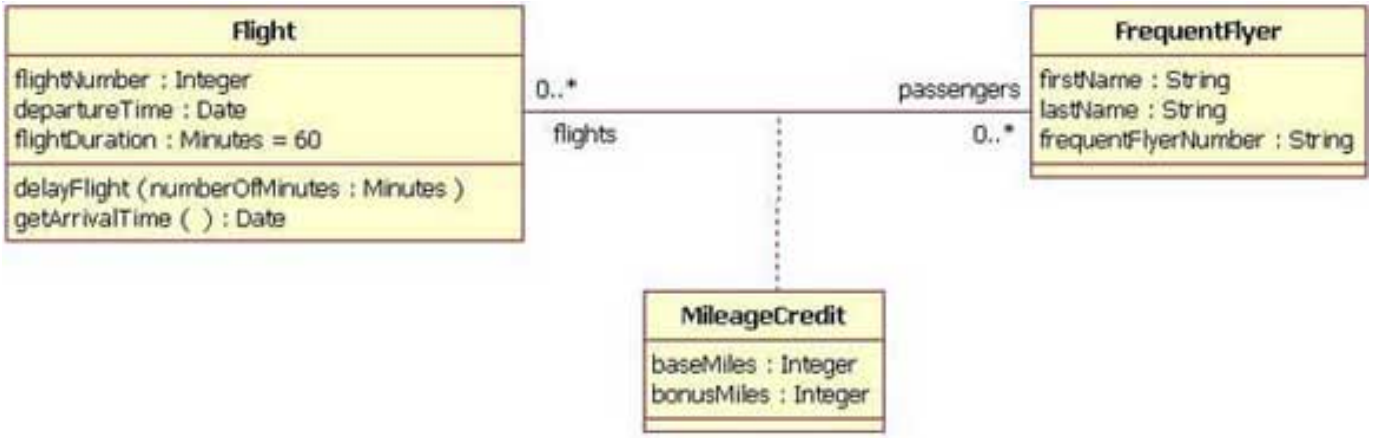
# Package


Accounts
BankAccount
owner : String
balance : Dollars
deposit ( amount : Dollars )
withdrawal ( amount : Dollars )
CheckingAccount
insufficientFundsFee : Dollars
processCheck ( checkToProcess : Check )
withdrawal ( amount : Dollars )
SavingsAccount
annualInterestRate : Percentage
depositMonthlyInterest ( )
withdrawal ( amount : Dollars )

# Interfaces


«interface»
Person
firstName : String
lastName : String
Professor
salary : Dollars
Student
major : String

# Association Class



# Aggregation

# Composition

# Composition Vs. Aggregation: Car / Engine Example

- **Composition**
  - Engine is fully encapsulated by the car
  - The off-side world does not have an outside reference of the engine
  - If a car object is garbage collected then the engine is garbage collected too
- **Aggregation**
  - The engine is not internal part of the car: the outside world can have a reference to that engine

# Composition Vs. Aggregation: Car / Engine Example

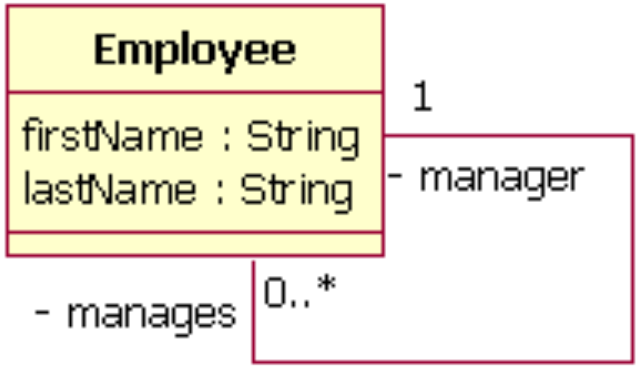
```
// Composition
class Car {
 private final Engine engine;

 Car(Data d) {
  engine = new Engine(d);
 }
 void move() {
    engine.work
 }
}
```

```
// Aggregation
class Car {
 private Engine engine;

 void setEngine(Engine e) {
  engine = e;
 }
 void move() {
   if(engine != null)
    engine.work
 }
}
```

# Reflexive associations



# Visibility

| Mark | Visibility |
| --- | --- |
| + | Public |
| # | Protected |
| - | Private |
| ~ | Package |

# Visibility


BankAccount
+ owner : String
+ balance : Dollars
+ deposit ( amount : Dollars )
+ withdrawal ( amount : Dollars )
# updateBalance (newBalance : Dollars )

# Object Constraint Language (OCL)

- **Defines first-order predicates**
  - > Text (no graphical symbols)
  - > Details in the OCL 2.0 Specification, chapter 7
- **Used to express various constraints:**
  - > Over attributes in a class
  - > Over all elements of a class
  - > Between associated objects
  - > Over collections of objects (sets, sequences, bags)
  - > Pre- and post-conditions of operations
- **Often stated in a separate document**
  - > Simple constraints can be included directly in class diagrams

# Basic OCL Syntax

- **Invariant constraints**

  **context** <class>
  **inv**: <OCL predicate>
  …
  **inv**: <OCL predicate>

- **Pre/Post constraints**

  **context** <class>::<operation>
  **pre**: <OCL predicate>
  …
  **pre**: <OCL predicate>
  **post**: <OCL predicate>
  …
  **post**: <OCL predicate>

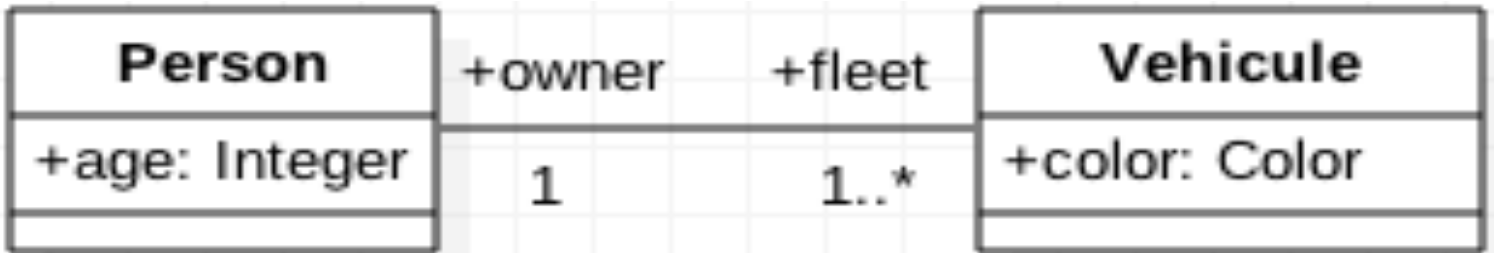
# Basic OCL Syntax (2)

- **Common operators**

  (In decreasing precedence order):

  - ‘pre’, ‘post’
    - “before state” “after state”
  - ‘.’ , ‘->’
    - <class>.<attribute or operation>
    - <class>.<opposite role in an association> *(navigation)*
    - <collection>-><built-in operation>
  - ‘not’
  - ‘if-then-else-endif’
  - ‘<’, ‘>’, ‘<=’, “>=
  - ‘=’, ‘<>’
  - ‘and’, ‘or’, ‘xor’
  - ‘implies’

# Revisiting the Example




**How can we express that:**

- **An owner of a vehicle must be at least 18 years old?**

```
context   Vehicle
inv: self.owner.age >= 18
```

Implicit universal quantification:

A x: Vehicle ∞ x.owner.age >= 18

# Basic OCL Syntax (3)

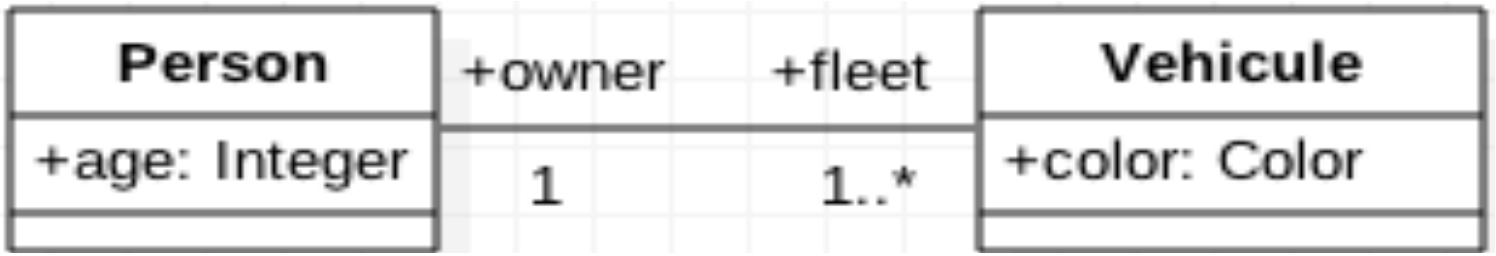
- **Commonly used operations on collections:**
  - Let c be any collection (e.g., person.fleet)

```
c->size()                          -- #c
c->includes(object)                -- object ε = dfdf c
c->excludes(object)                -- object ™ c
c->includesAll(collection)         -- collection ζ c
c->isEmpty()                       -- c = 0
c->notEmpty()                      -- c ⌊ 0
c->exists(x | predicate)           -- E x:c ∞ predicate
c->forAll(x | predicate)           -- A x:c ∞ predicate
c->including(object)               -- c Y { object }
c->excluding(object)               -- c \ { object }
c->union(collection)               -- c Y collection
c->intersection(collection)        -- c I collection
c->select(x | predicate)           -- { x:c | predicate }
```

# OCL Collection Types

- **Sets**
  - > Navigating an association from an object results in a Set
    - »The cardinality of the set is determined by the multiplicity at the opposite end of the association
- **Bags**
  - > Navigating an association from a set of objects results in a Bag
    - »A bag is a set that allows duplicates
  - > Navigating through more than one association with multiplicity greater than 1 results in a Bag

# Revisiting the Example (II)



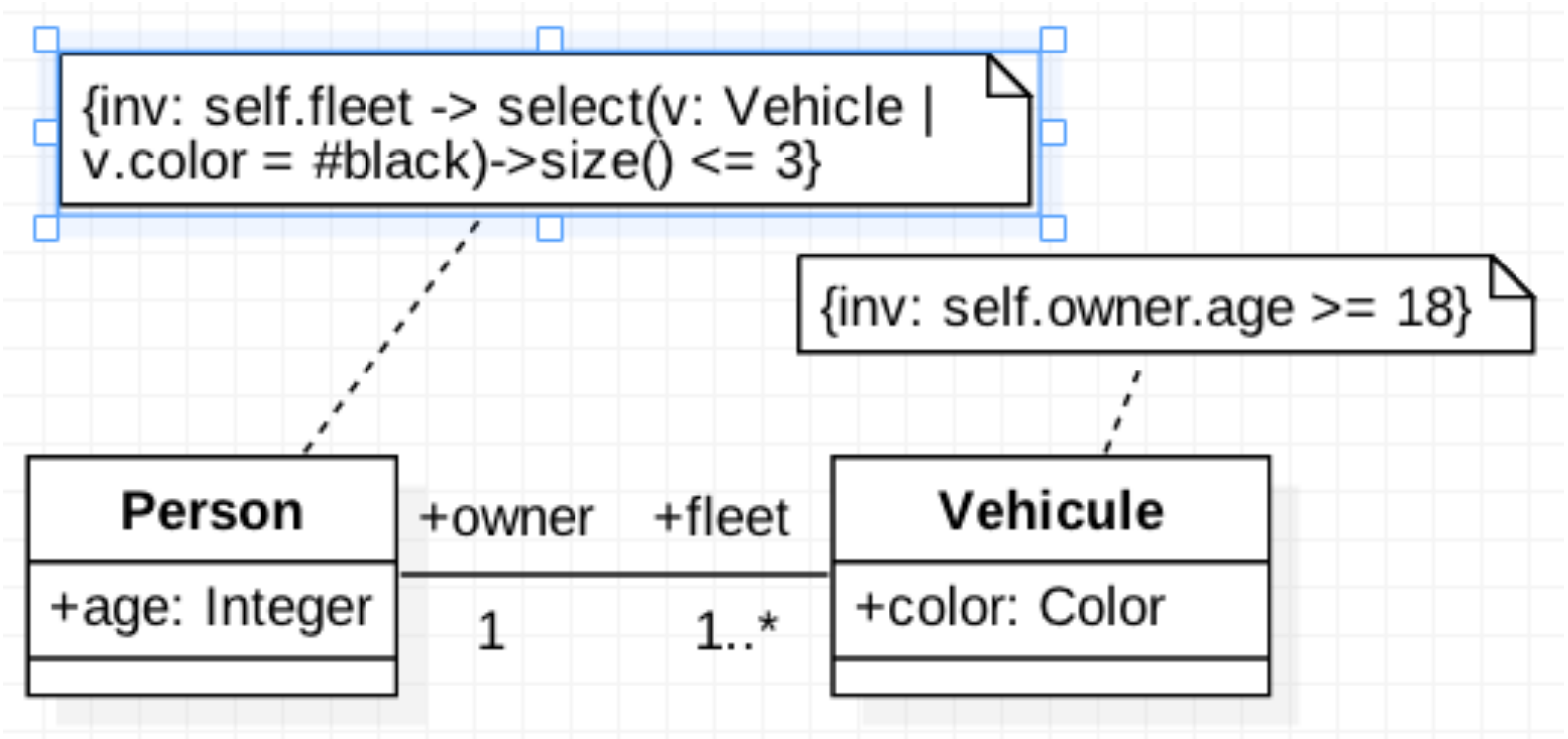

**How can we express that:**

- **A person cannot have more than 3 black cars?**

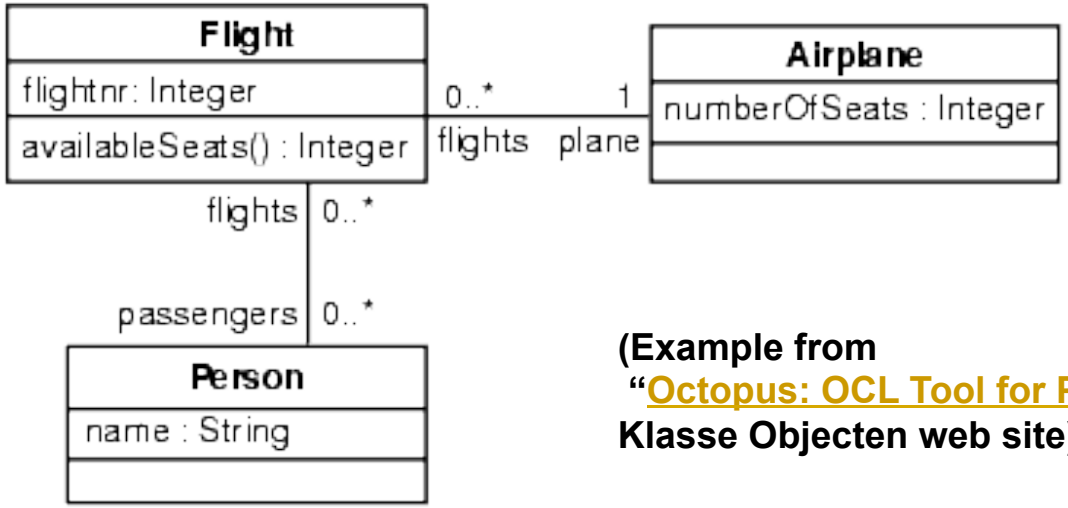
```
context  Person
inv: self.fleet->select(v:Vehicle | v.color = #black)->size() <= 3
```

# Revisiting the Example (III)

- **OCL constraints can be included in the class diagram as note boxes:**



# Example 2

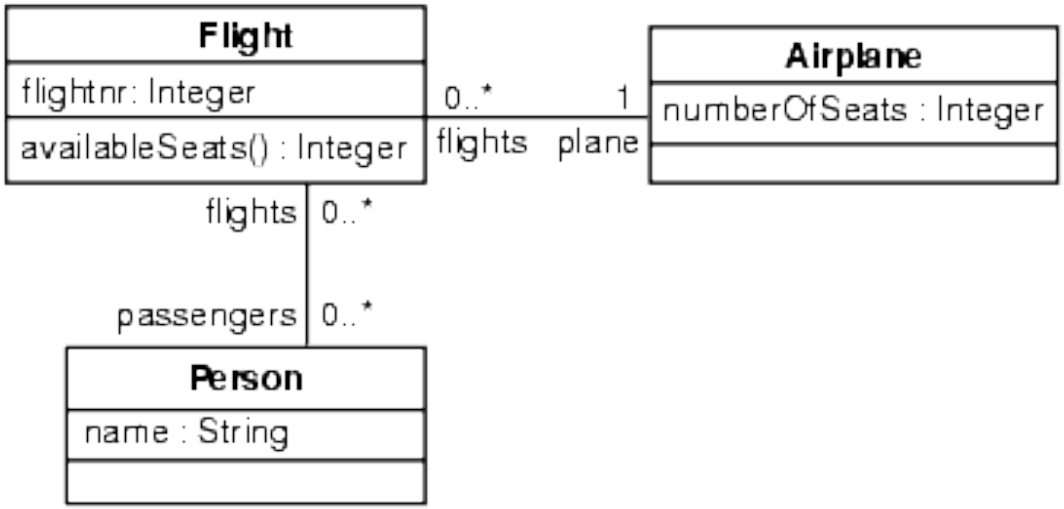



**(Example from "Octopus: OCL Tool for Precise Uml Specifications", Klasse Objecten web site)**

-- The number of passengers of a flight cannot exceed the number of seats of the corresponding plane.

context Flight
inv: self.passengers->size() <= self.plane.numberOfSeats

# Example 2 (2)



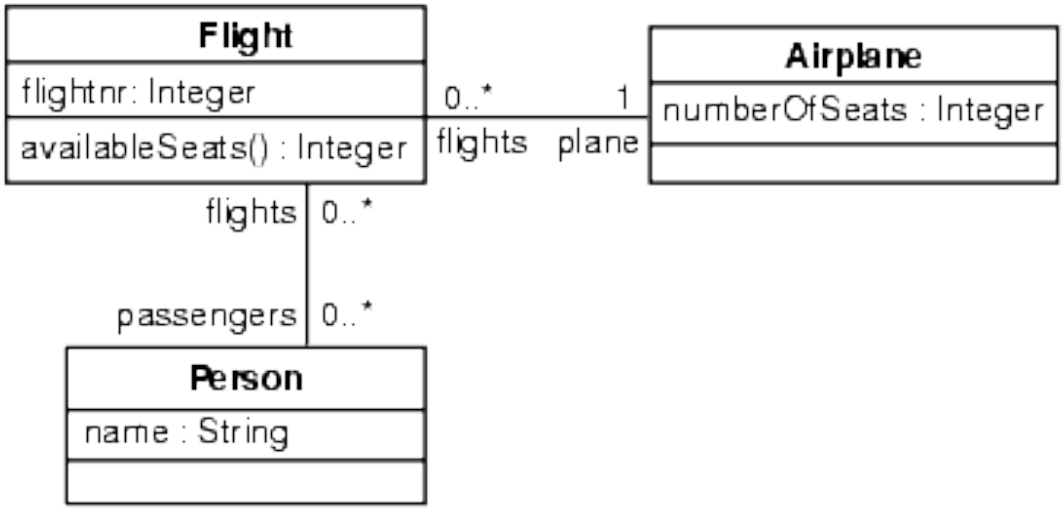

--There cannot be two flights with the same flight number

~~context Flight~~

inv: Flight.allInstances->forAll(f1, f2 | f1 <> f2 implies

f1.flightnr <> f2.flightnr)

# Example 2 (3)




-- Obtain the total number of passengers carried in this plane

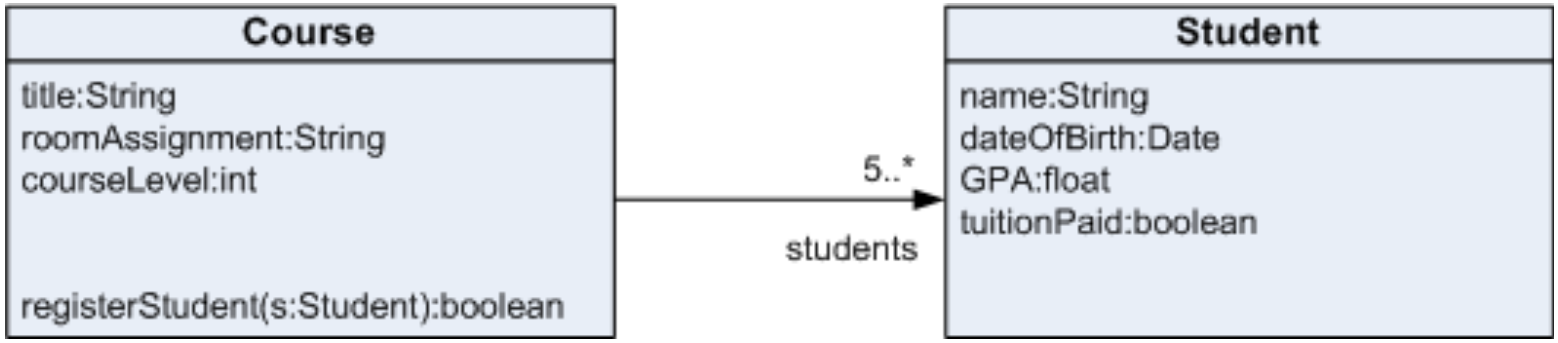
context Airplane::totalPassengersCarrried() : Integer

pre: self.flights->notEmpty()

post: result = self.flights.passengers->size()

Navigating across multiple associations produces a bag

# Example 3 – More Constraints on Operations




- Ensure that registered students have paid tuition
  context **Course::registerStudent(s:Student)** : boolean
  pre: **s.tuitionPaid = true**
  post: **result = true**

- Use @pre to refer to elements before operation
  context **Course::registerStudent(s:Student) : boolean**
  pre: **s.tuitionPaid = true**
  post: **result = true AND self.students->size() = self.students@pre->size()+1**

# Outline

1. Introduction to System Modeling and UML
2. Class Diagrams
3. Class Diagrams & OCL
4. **State Machine Diagrams**
5. Activity Diagrams
6. Sequence Diagrams

# State Machine Diagrams

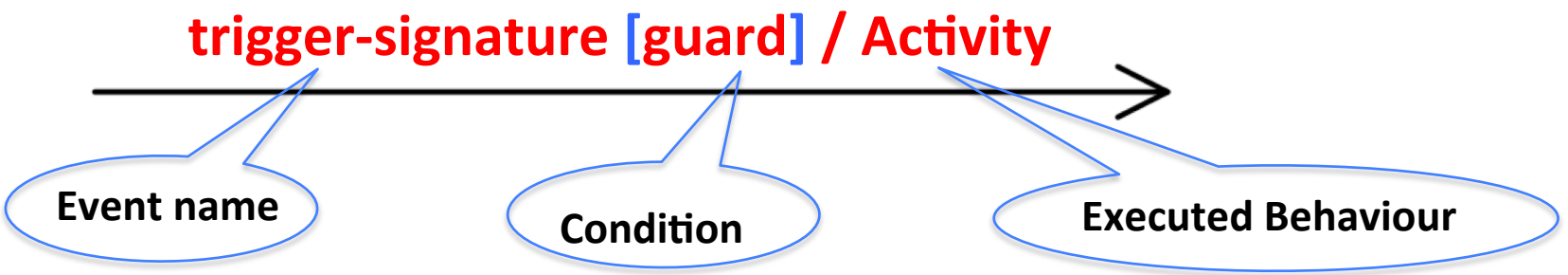
- **Used for modeling discrete behavior through finite state transition systems**

# State Machine Diagram Basic Notation

Initial Pseudo State

Final State

End of flow



# States

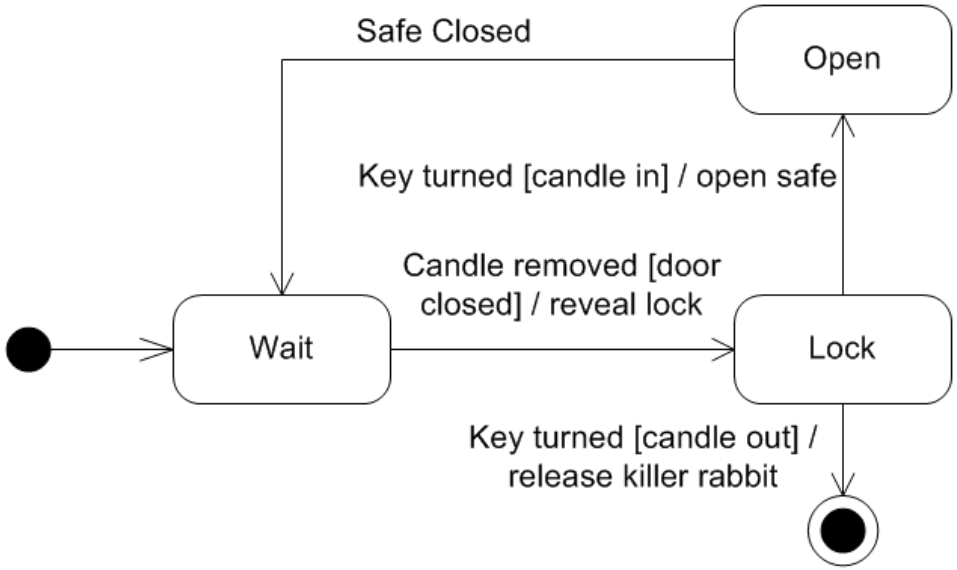
- A state models a situation during which some (usually implicit) condition holds.
- The condition may represent a static situation, such as an object waiting for some external event to occur.
- It can also represent dynamic conditions, such as the process of performing some activity

# When to Use State Diagrams?

- **State diagrams** are good at describing the behavior of **an object** across several **use cases**.
- State diagrams are not very good at describing behavior that involves a number of objects collaborating.

# State Machine Diagram Example

1. Mysterious Castle Secret Safe and Monster! *
2. Reveal lock when special candle removed & door closed
3. Can insert key into lock when lock revelead
4. For extra safety, must replace candle before turning key
5. If candle replaced, open safe
6. If candle not replaced, release killer rabbit!



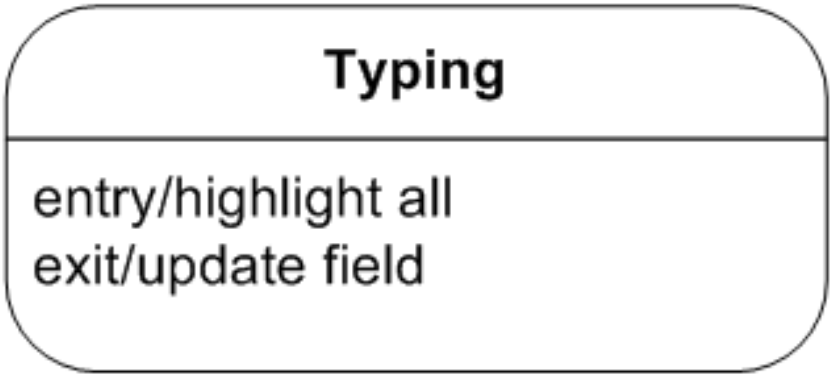

*From UML Distilled

# State Concepts

- A state can be active or inactive
  - > State becomes active when it is entered
  - > State becomes inactive when it is exited
- State is exited and entered in a self-transition

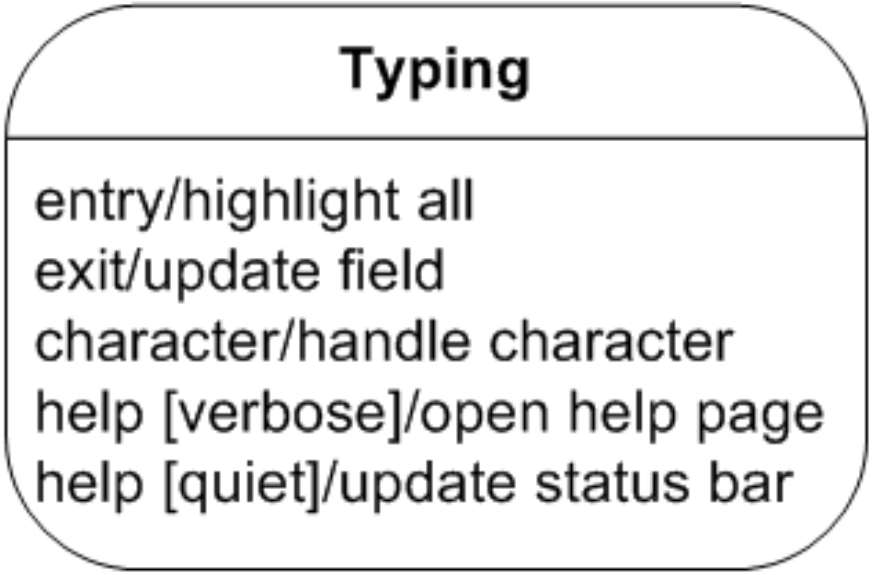
# State Concepts – Two Special Triggers

- **Entry**
  - > Fires automatically when entering the state
  - > Executes entry action *before* any other state action
- **Exit**
  - > Fires automatically when exiting the state
  - > Executes exit action *after* any other state action



# State Concepts – Internal Activities

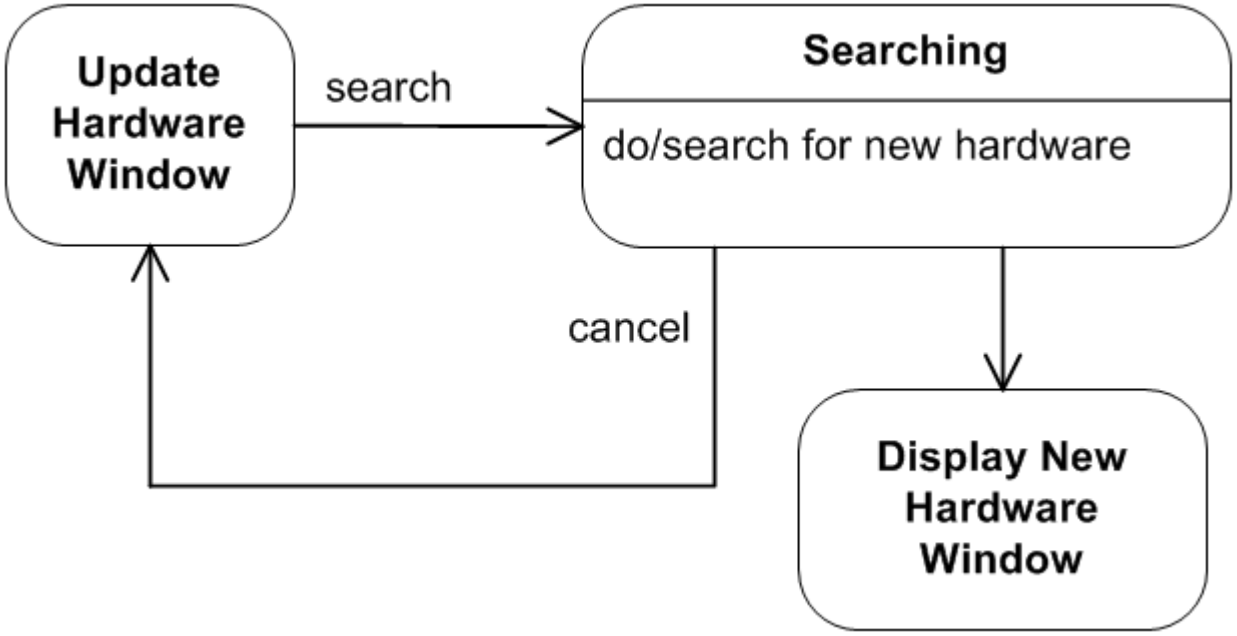
- States can react to events without transitions using internal activities
- Write event, guard and activity inside state
- Example:




From UML Distilled

# State Concepts – Activity States

- **Do action**
  - > **Do action fires repeatedly while state is active**
  - > **Starts following the entry action**
  - > **Raises a completion event if/when activity completes**
  - > **May be interrupted and terminated by cancel event**

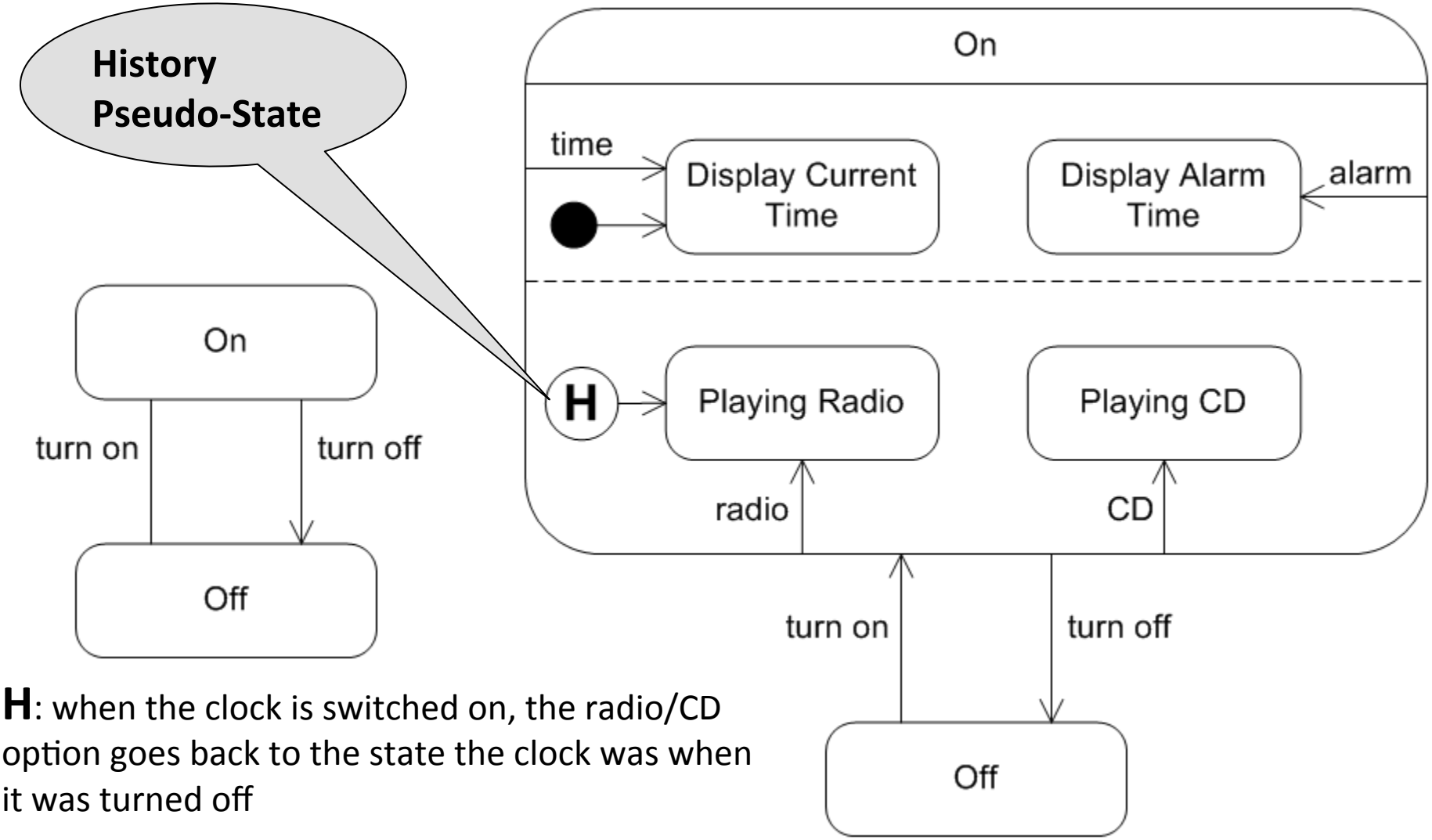



From UML Distilled

# Types of States

- **Simple state**
  - > A state that does not have sub-states
  - > All that we have seen so far
- **Composite state**
  - > A state that either contains one region or is decomposed into two or more orthogonal regions.
  - > Each region has a set of mutually exclusive disjoint sub-vertices and a set of transitions

# Composite State Machine Example – Concurrent States

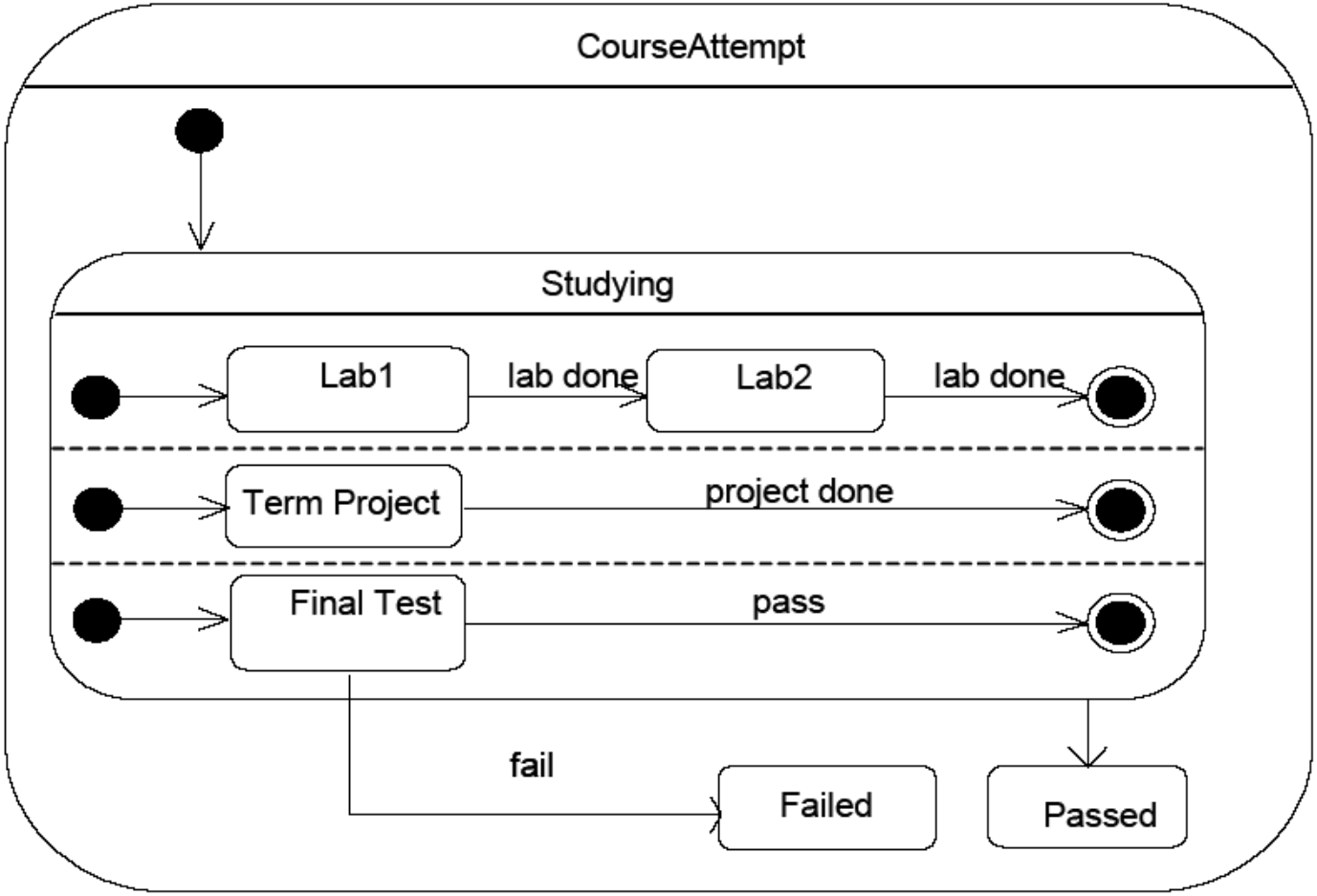



**H**: when the clock is switched on, the radio/CD option goes back to the state the clock was when it was turned off

# Types of Compound Execution

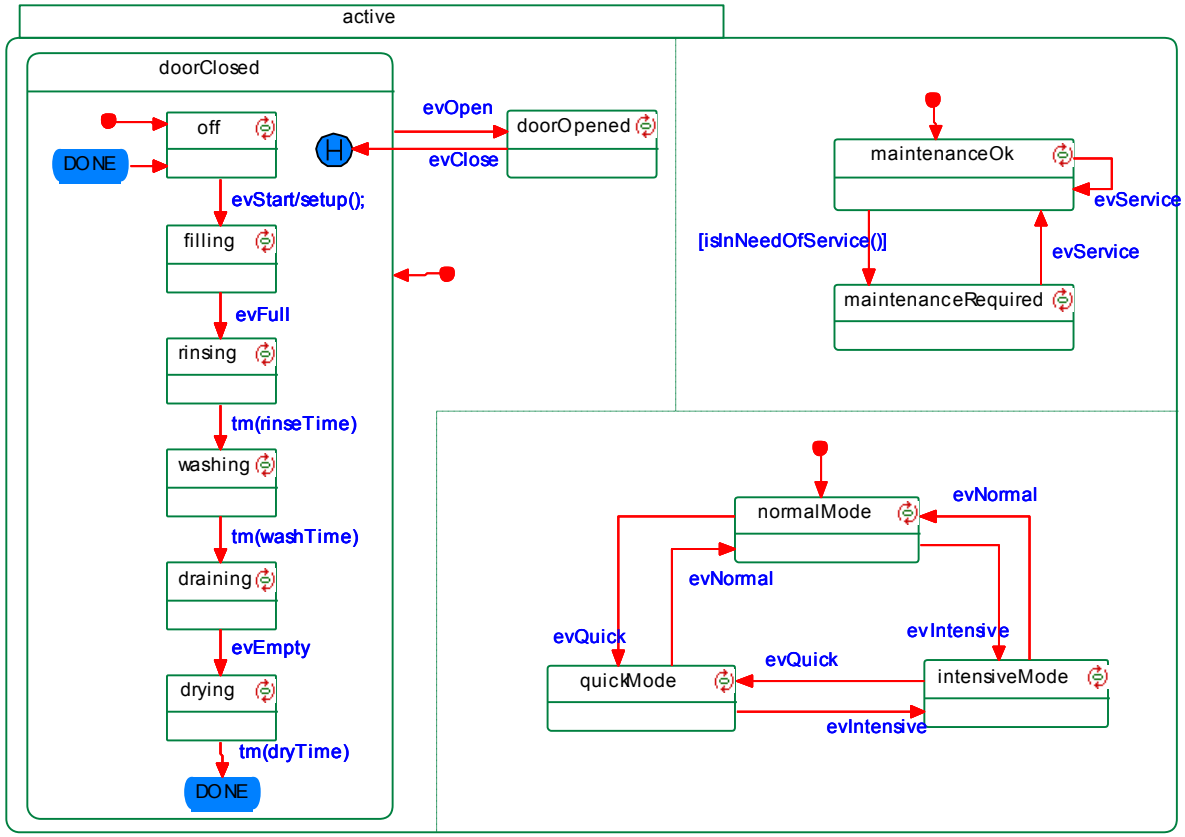
- Except during transition execution, the following invariants always apply to state configurations:
  - > If a composite state is active and not orthogonal, exactly one of its substates is active.
  - > If the composite state is active and orthogonal, all of its regions are active, one substate in each region.

# Orthogonal State with Regions



# Orthogonal State with Regions – 2

- **A Dishwasher**



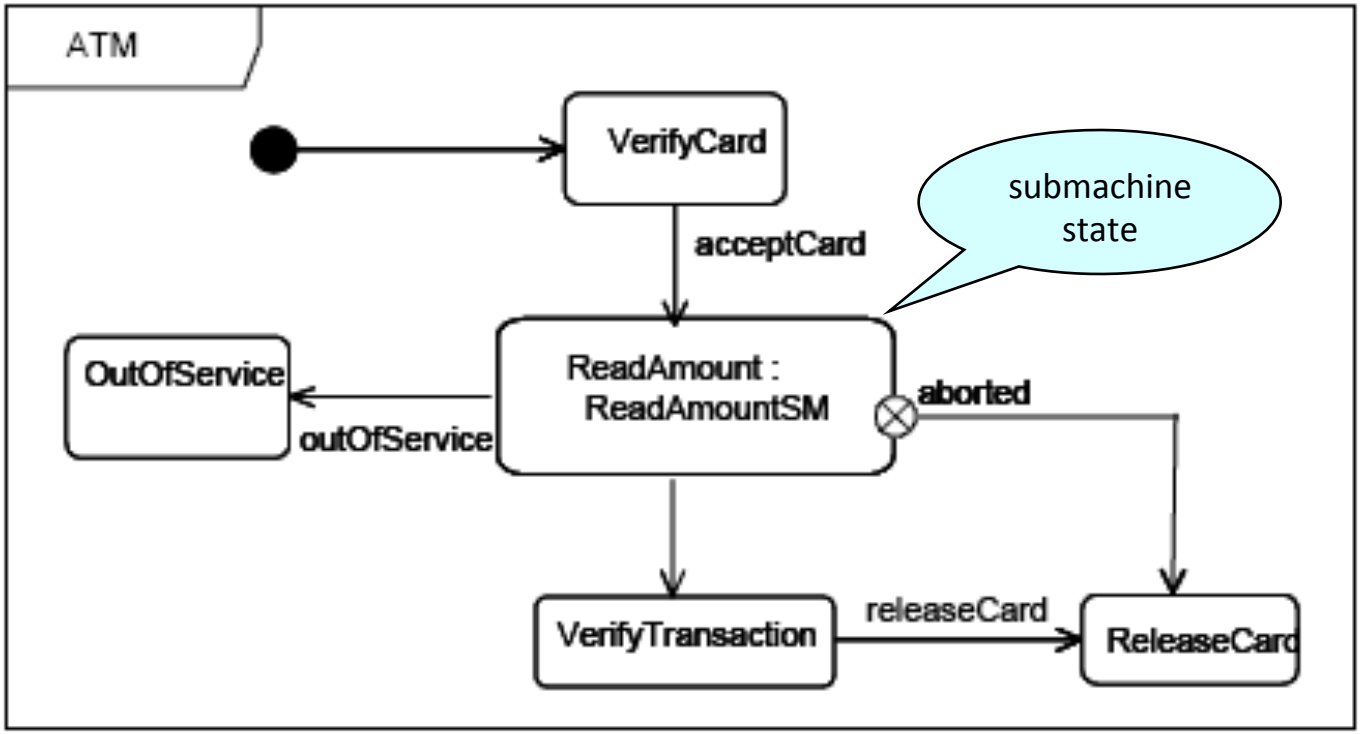

Example taken from I-Logix Rhapsody sample UML models

# Submachine States

- This is another mechanism to express nesting and also referencing other state machines
- The submachine state is depicted as a normal state where the string in the name compartment has the following syntax

  <state name> ‘:’ <name of referenced state machine>

# Submachine state example



# Submachine state example (2)

- **ReadAmountSM Submachine**