Object Oriented Analysis
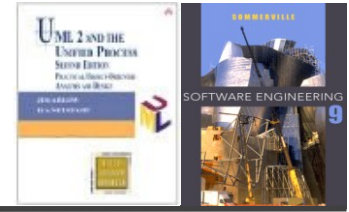
Lecture 5

Outline

✧ Objects and classes [Lecture 4]

✧ Finding analysis classes [Lecture 4]

✧ Relationships between objects and classes

- Links
- Associations
- Dependencies

✧ Inheritance and polymorphism

✧ State diagram

Relationships Between Objects and Classes

Lecture 5/Part 1

What is a link?

- ✧ Links are connections between objects
  - ▪ Think of a link as a **telephone line** connecting you and a friend. You can send messages back and forth using this link
- ✧ Links are the way that objects communicate
  - ▪ Objects **send messages to each other** via links
  - ▪ Messages **invoke operations**
- ✧ OO programming languages implement links as object references or pointers
  - ▪ When an object has a reference to another object, we say that there is a **link** between the objects
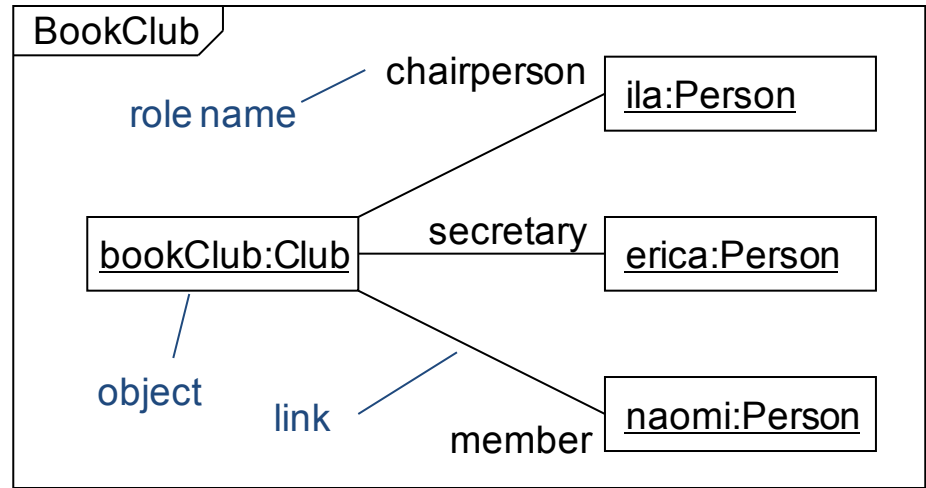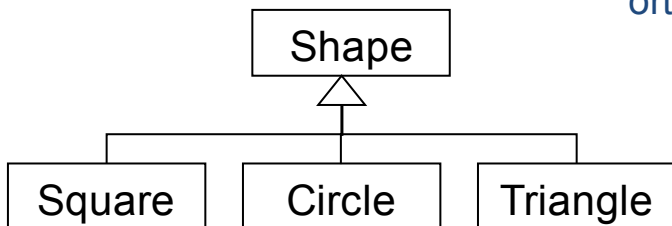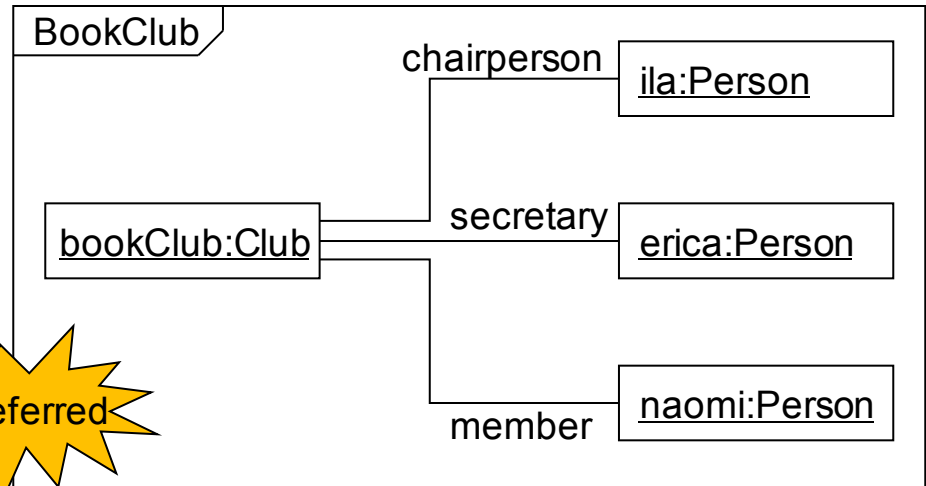
Object diagrams

- ✧ Paths in UML diagrams can be drawn as orthogonal, oblique or curved lines

- ✧ We can combine paths into a tree if each path has **the same properties**

BookClub

oblique path style

role name → chairperson — ila:Person

bookClub:Club — secretary — erica:Person

object — link — member — naomi:Person

Shape
├── Square
├── Circle
└── Triangle

BookClub

orthogonal path style

chairperson — ila:Person

bookClub:Club — secretary — erica:Person

member — naomi:Person

preferred

What is an association?

association

Club —— Person

«instantiate» «instantiate» «instantiate»

link

bookClub:Club —— chairman —— jim:Person

links instantiate associations
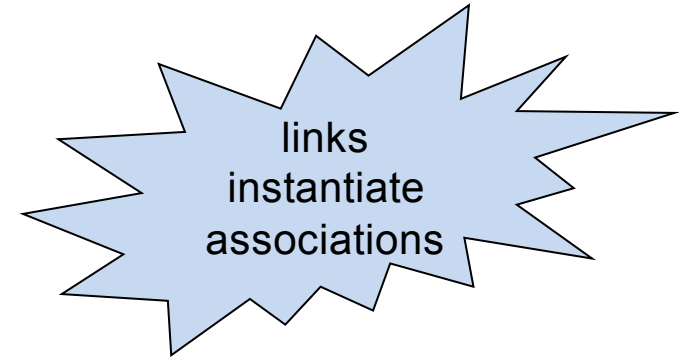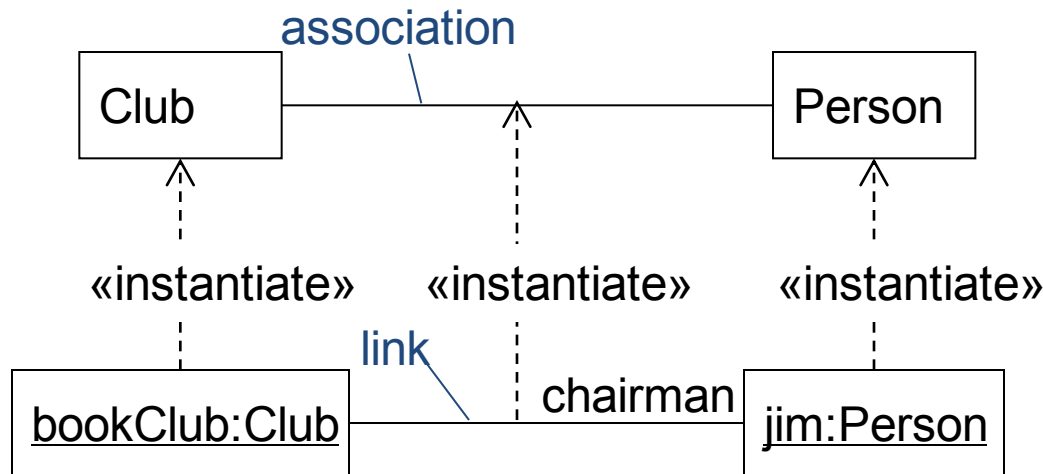
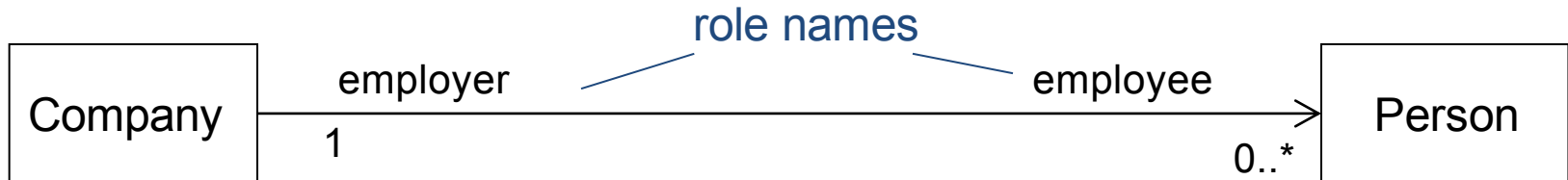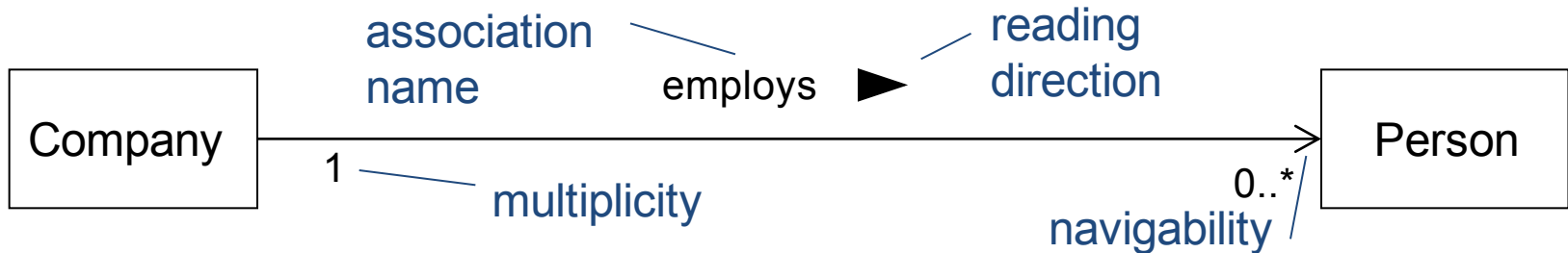✧ Associations are relationships between classes

✧ Associations between classes indicate that there **may be** links between objects of those classes, while links indicates that there **must be** associations

✧ Can there be a communication between objects of two classes that have no association between them?

Association syntax

association name — employs — reading direction

Company — Person

1 — multiplicity — 0..* — navigability

role names

Company — employer — employee — Person

1 — 0..*

✧ An association can have role names *or* an association name

✧ Multiplicity is a constraint that specifies the number of objects that can participate in a relationship at any point in time

▪ If multiplicity is not explicitly stated in the model then it is undecided – *there is no default multiplicity*
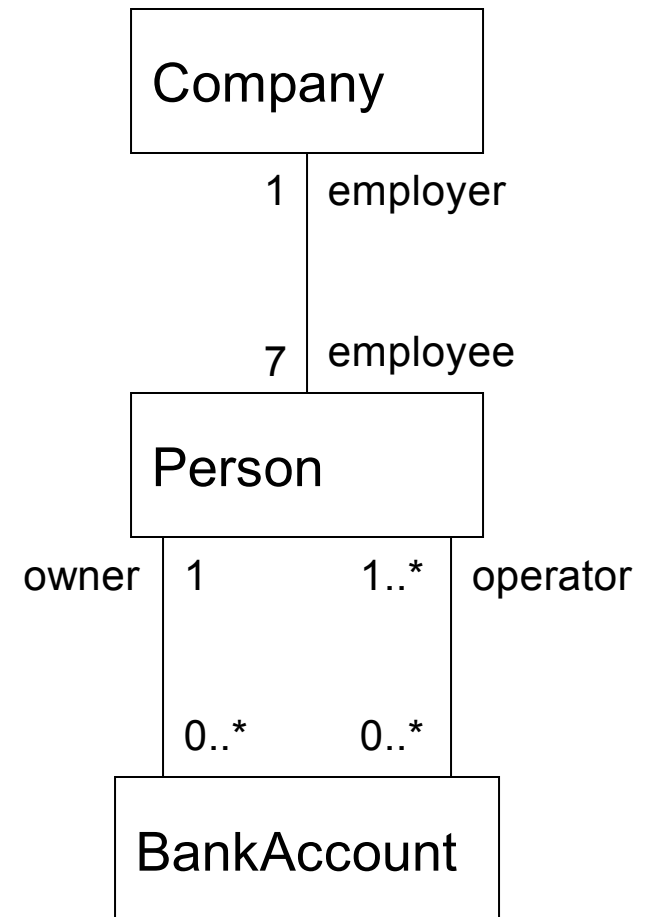
multiplicity: min..max	
0..1	zero or 1
1	exactly 1
0..*	zero or more
1..*	1 or more
1..6	1 to 6

Multiplicity exercise

✧ How many

- Employees can a Company have?
- Employers can a Person have?
- Owners can a BankAccount have?
- Operators can a BankAccount have?
- BankAccounts can a Person have?
- BankAccounts can a Person operate?

```
┌──────────────┐
│   Company    │
└──────────────┘
       │  1  employer
       │
       │  7  employee
┌──────────────┐
│    Person    │
└──────────────┘
  owner │ 1       1..* │ operator
        │              │
   0..* │         0..* │
┌──────────────┐
│  BankAccount │
└──────────────┘
```

Reflexive associations: file system example

subdirectory

0..* — **Directory** — 1 — 0..* — **File**

0..1
parent

reflexive association

C

autoexec

config

Windows   My Documents   Corel   To John

Command

directories

files
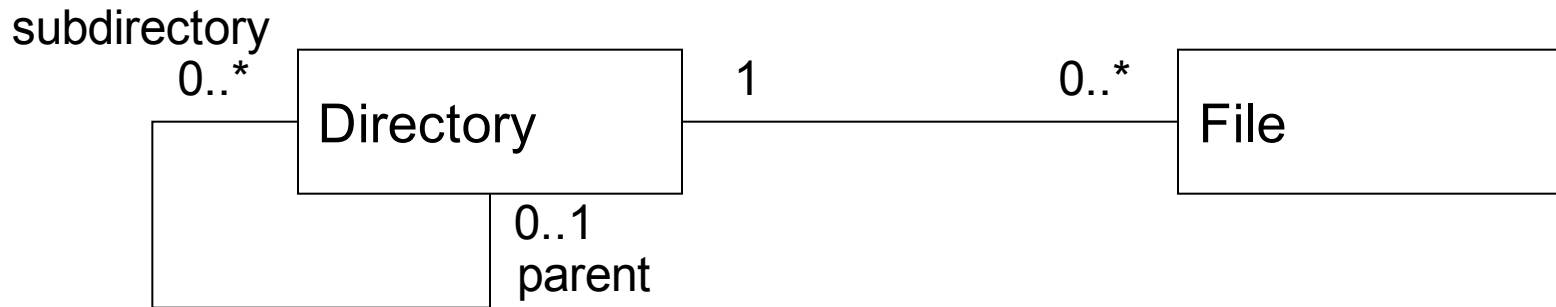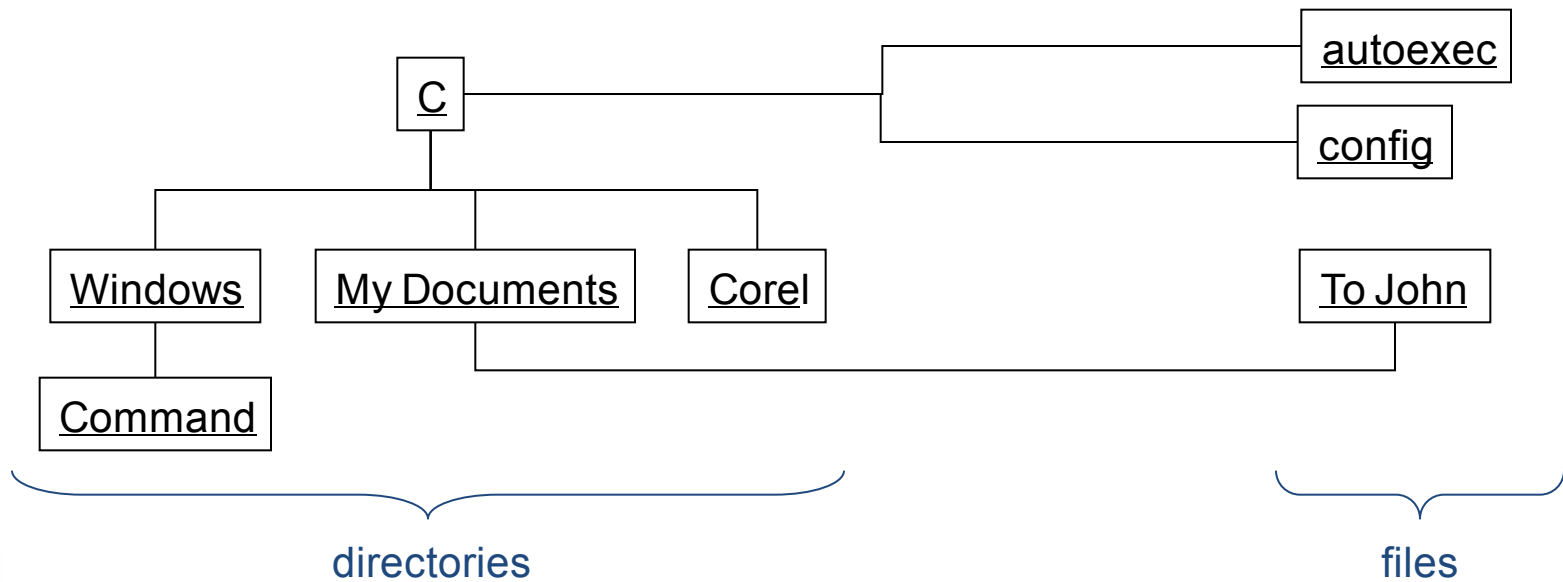
Hierarchies and networks

hierarchy	network
A 0..* 0..1	B 0..* 0..*
a1:A — b1:A, c1:A, d1:A — e1:A, f1:A, g1:A	c1:B, b1:B, f1:B, e1:B, d1:B, a1:B, g1:B
In an association hierarchy, each object has **zero or one** object directly above it.	In an association network, each object has **zero or many** objects directly above it.

Navigability

♢ Navigability indicates that it is possible to traverse from an object of the *source* class to objects of the *target* class

An Order object stores a list of Products
Navigable

navigability

Order   0..*   0..* → Product

source   target
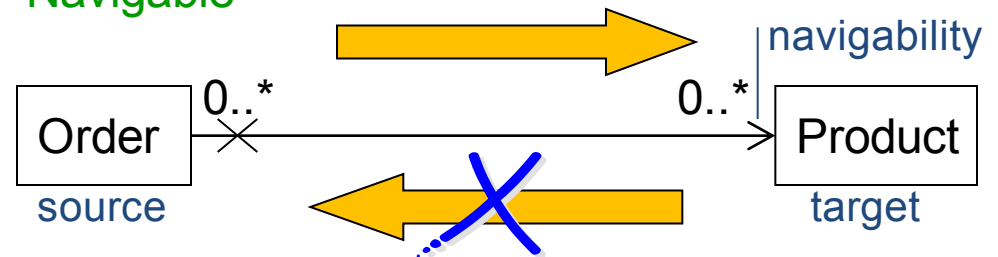
Not navigable
A Product object does not store a list of Orders

♢ Can there be a communication in a direction not supported by the navigability?

♢ Are some of the cases on the right equivalent?

| A ←————→ B | A to B is navigable
B to A is navigable |

| A ⊢—✕———→ B | A to B is navigable
B to A is not navigable |

| A ————→ B | A to B is navigable
B to A is undefined |

| A ———— B | A to B is undefined
B to A is undefined |
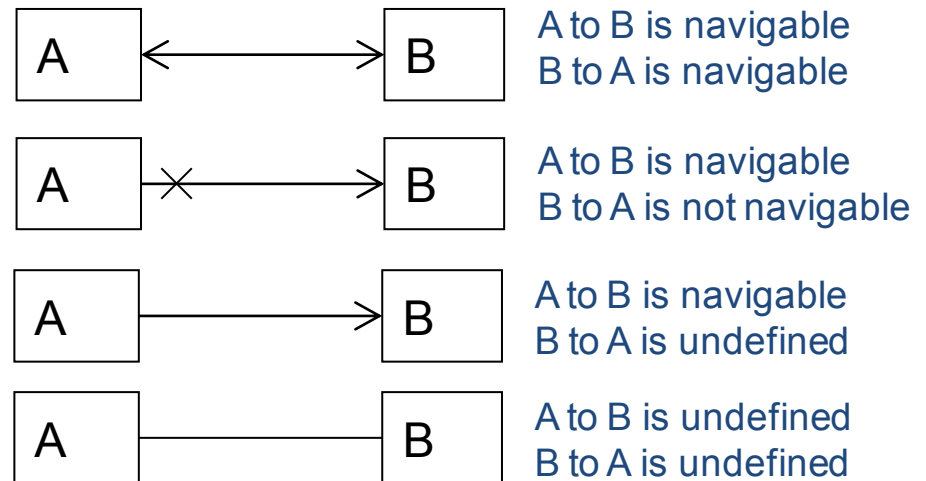
Associations and attributes

House — address → Address (1, 1) **=** House / address:Address

House / address:Address — pseudo-attribute

attribute

♢ An association is (through its role name) a **representation of an attribute**

♢ **Use associations when:**

- The target class is an important part of the model
- The target class is a class that you have designed yourself

♢ **Use attributes when:**

- The target class is not important, e.g. a primitive type such as number, string
- The target class is just an implementation detail such as a bought-in component or a library component e.g. Java.util.Vector (from the Java standard libraries)

Association classes
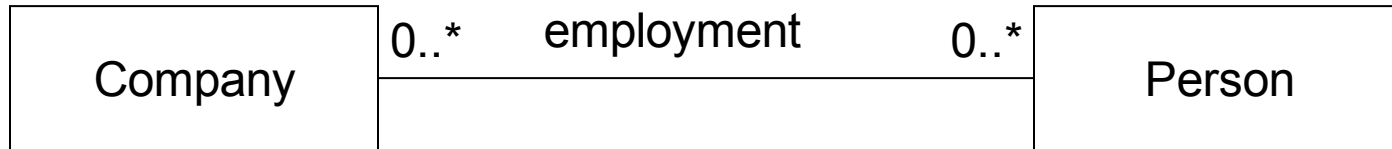
❖ Where do we record the Person's salary?

❖ We model the association itself as an association class. Exactly one instance of this class exists for each link between a Person and a Company.

❖ We can place the salary and any other attributes or operations which are really features of the association into this class

association class

the association class consists of the class, the association and the dashed line
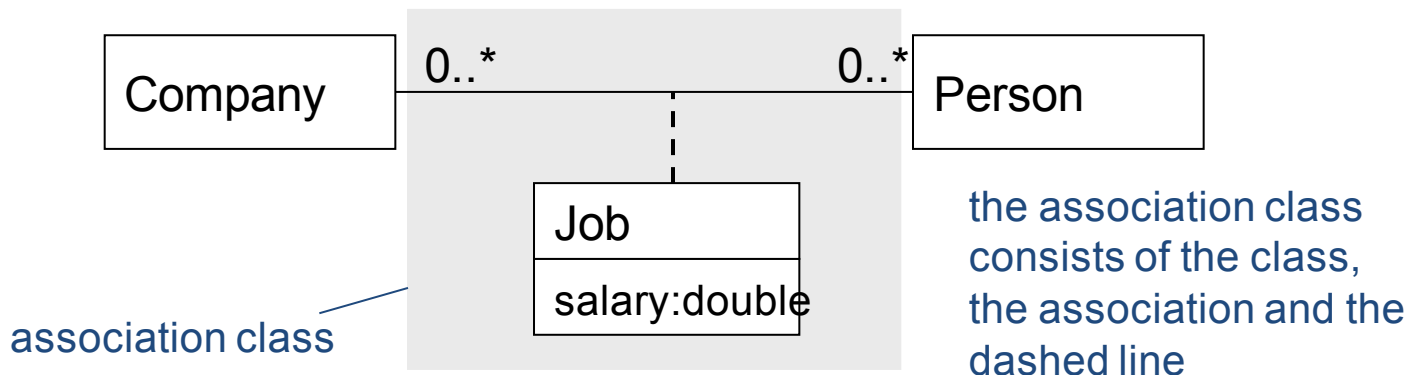
Using association classes

If we use an association class, then a particular Person can have only **one** Job with a particular Company

```
Company  0..* ------- 0..*  Person
              |
              | (dashed)
              |
            Job
        salary:double
```
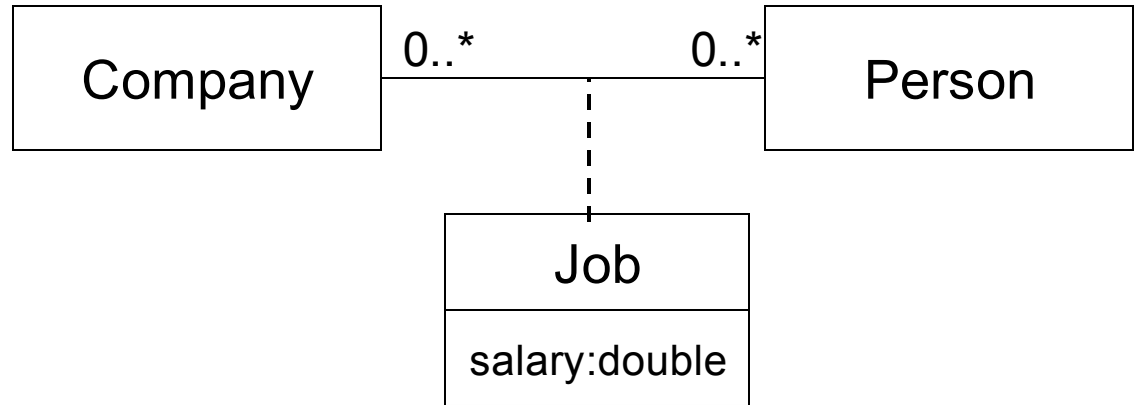
If, however a particular Person can have **multiple** jobs with the same Company, then we must use a reified association

```
Company  1  0..*  Job  0..*  1  Person
                salary:double
```
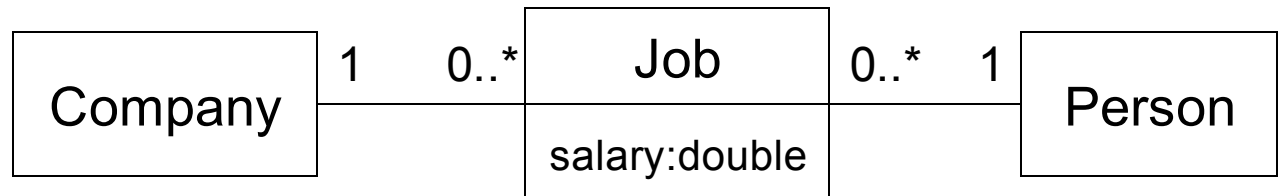
Dependencies

- ✧ "A dependency is a relationship between two elements where a change to one element (the supplier) may affect or supply information needed by the other element (the client)".
    - ▪ In other words, the client **depends** in some way on the supplier
    - ▪ Weaker type of relationship than **association**
    - ▪ Can there be both association and dependency between two classes?
- ✧ Three types of dependency:
    - ▪ **Usage** - the client uses some of the services made available by the supplier to implement its own behavior – this is the most commonly used type of dependency
    - ▪ **Abstraction** - a shift in the level of abstraction. The supplier is more abstract than the client
    - ▪ **Permission** - the supplier grants some sort of permission for the client to access its contents – this is a way for the supplier to control and limit access to its contents

Usage dependencies

✧ Stereotypes

- **«use»** - the client makes use of the supplier to implement its behaviour
- **«call»** - the client operation invokes the supplier operation
- **«parameter»** - the supplier is a parameter of the client operation
- **«send»** - the client (an operation) sends the supplier (a signal) to some unspecified target
- **«instantiate»** - the client is an instance of the supplier

the stereotype is often omitted

A
foo(b : B) bar() : B doSomething()

«use»

B

```
A :: doSomething() {
    B myB = new B();
}
```

A «use» dependency is generated between A and B when B is used in A as a parameter, return value or inside method body
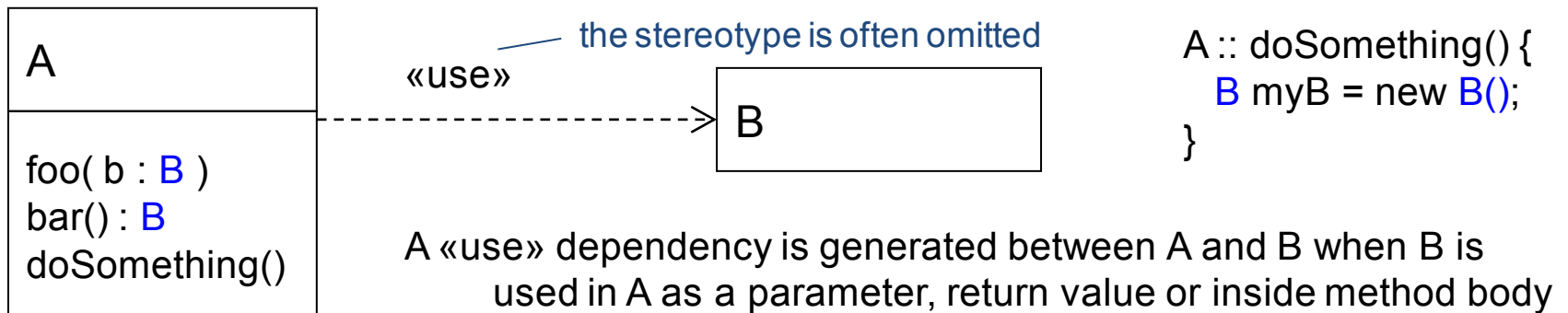
Abstraction and permission dependencies

✧ Abstraction dependencies

- **«trace»** - the client and the supplier represent the same concept but at different points in development

- **«substitute»** - the client may be substituted for the supplier at runtime. The client and supplier must realize a common contract. Use in environments that *don't* support specialization/generalization

- **«refine»** - the client represents a fuller specification of the supplier

- **«derive»** - the client may be derived from the supplier. The client is logically redundant, but may appear for implementation reasons

✧ Permission dependencies

- **«access»** the public contents of the supplier package are added as private elements to the namespace of the client package

- **«import»** the public contents of the supplier package are added as public elements to the namespace of the client package

- **«permit»** the client element has access to the supplier element despite the declared visibility of the supplier

Key points

◇ Links – relationships between objects

◇ Associations – relationships between classes

- role names
- multiplicity
- navigability
- association classes

◇ Dependencies – relationships between model elements

- usage
- abstraction
- permission

Inheritance and polymorphism

Lecture 5/Part 2

Generalisation

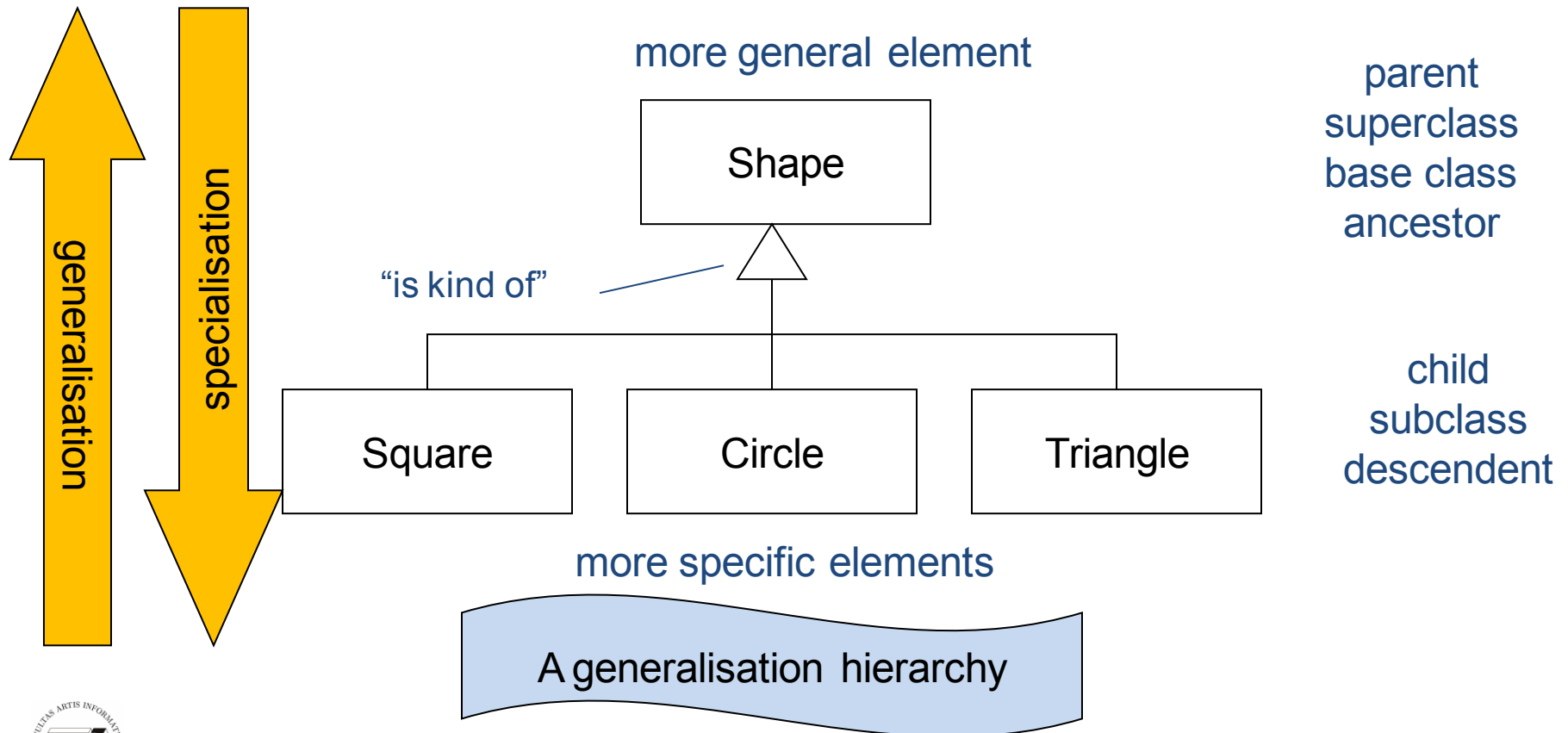A relationship between a more general element and a more specific element (with more information)

generalisation

specialisation

more general element

Shape

"is kind of"

Square   Circle   Triangle

more specific elements

parent
superclass
base class
ancestor

child
subclass
descendent

A generalisation hierarchy

Class inheritance

- ✧ Subclasses inherit **all** features of their superclasses:
  - ▪ attributes
  - ▪ operations
  - ▪ relationships
  - ▪ stereotypes, tags, constraints
- ✧ Subclasses can add new features
- ✧ Subclasses can override superclass operations
- ✧ We can use a subclass instance **anywhere** a superclass instance is expected

Substitutability Principle

Shape
origin : Point = (0,0) width : int {>0} height : int {>0}
draw(g : Graphics) getArea() : int getBoundingArea() : int

Square

Circle
radius : int = width/2
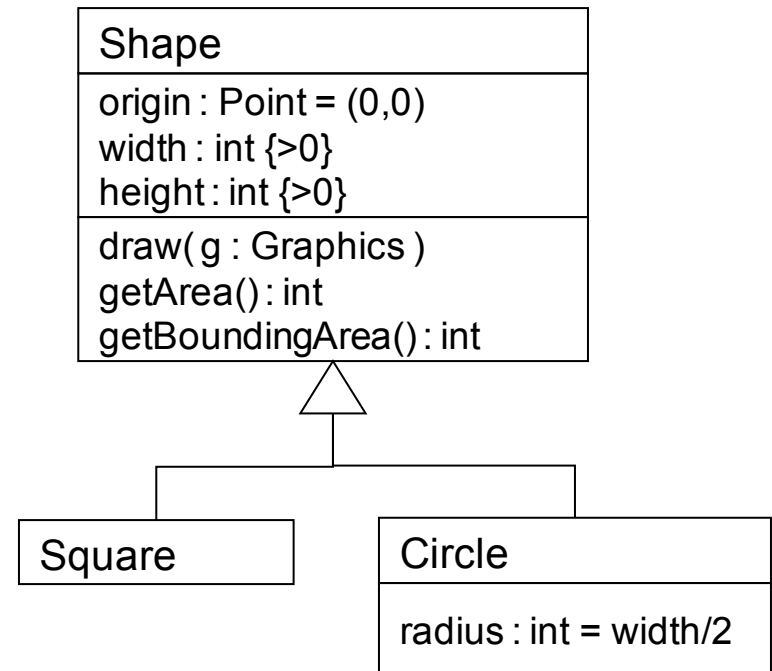
What's wrong with these subclasses?

Overriding

```
Shape
draw( g : Graphics )
getArea() : int
getBoundingArea() : int
```
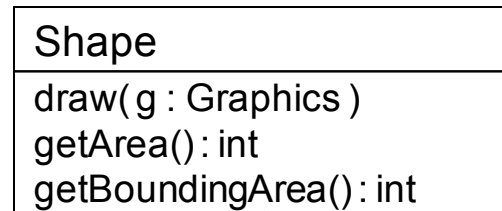
What's wrong with
the superclass?

```
Square
draw( g : Graphics )
getArea() : int
```

```
Circle
draw( g : Graphics )
getArea() : int
```

width x height

π x radius^2
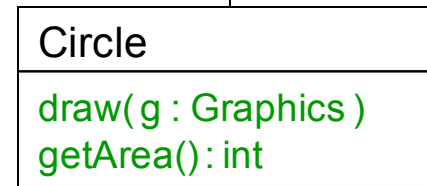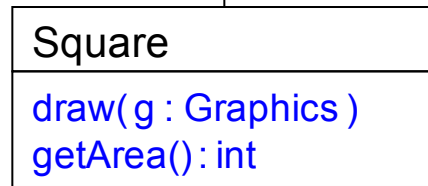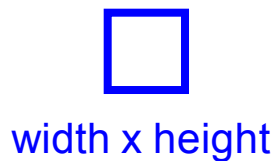
✧ Subclasses often need to **override** superclass behaviour

✧ To override a superclass operation, a subclass must provide an operation with the same signature

  ▪ The operation signature is the operation name, return type and types of all the parameters

Abstract operations & classes

abstract class ———

Shape

draw(g : Graphics)
getArea() : int
getBoundingArea() : int

abstract operations

abstract class and operation names *must* be in italics

concrete operations

concrete classes

Square
draw(g : Graphics)
getArea() : int

Circle
draw(g : Graphics)
getArea() : int
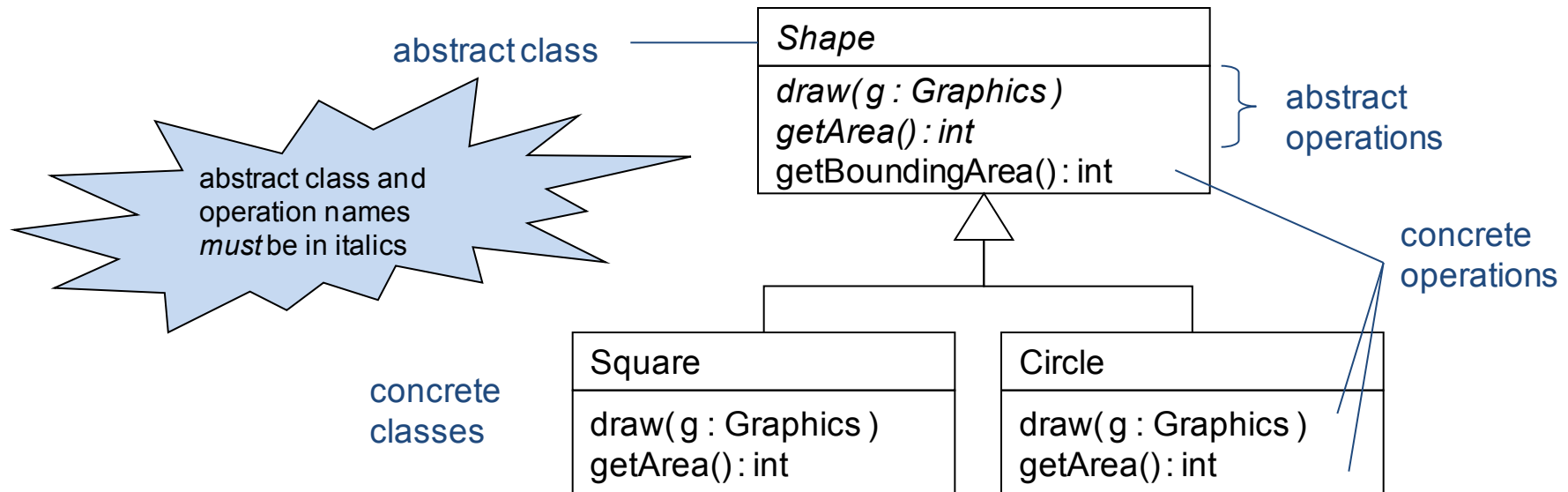
✧ We can't provide an implementation for
Shape :: draw(g : Graphics) or for
Shape :: getArea() : int
because we don't know how to draw or calculate the area for a "shape"!

✧ Operations that lack an implementation are *abstract* **operations**

✧ A class with any abstract operations **can't be instantiated** and is therefore an **abstract class**

Exercise

what's wrong with this model?

Vehicle
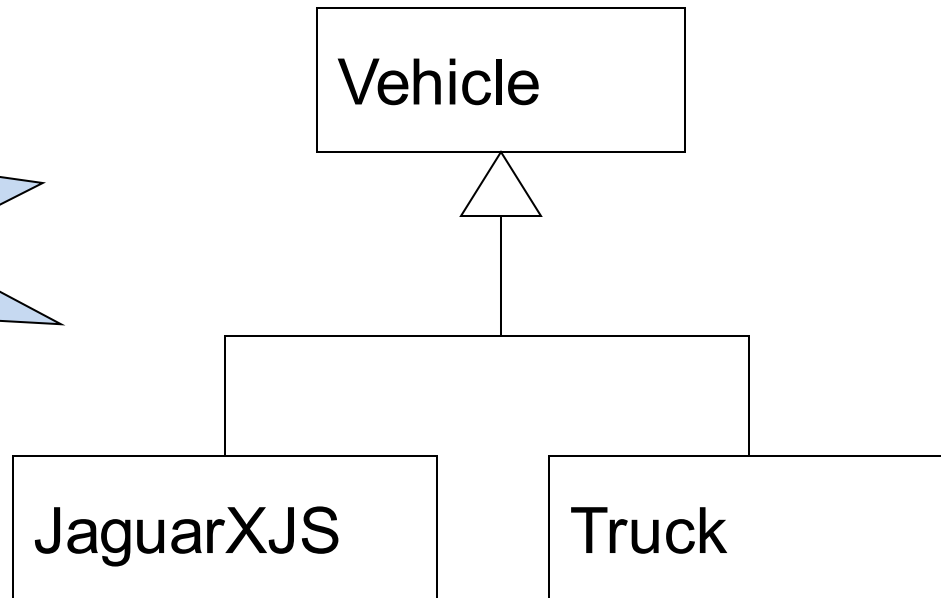
JaguarXJS
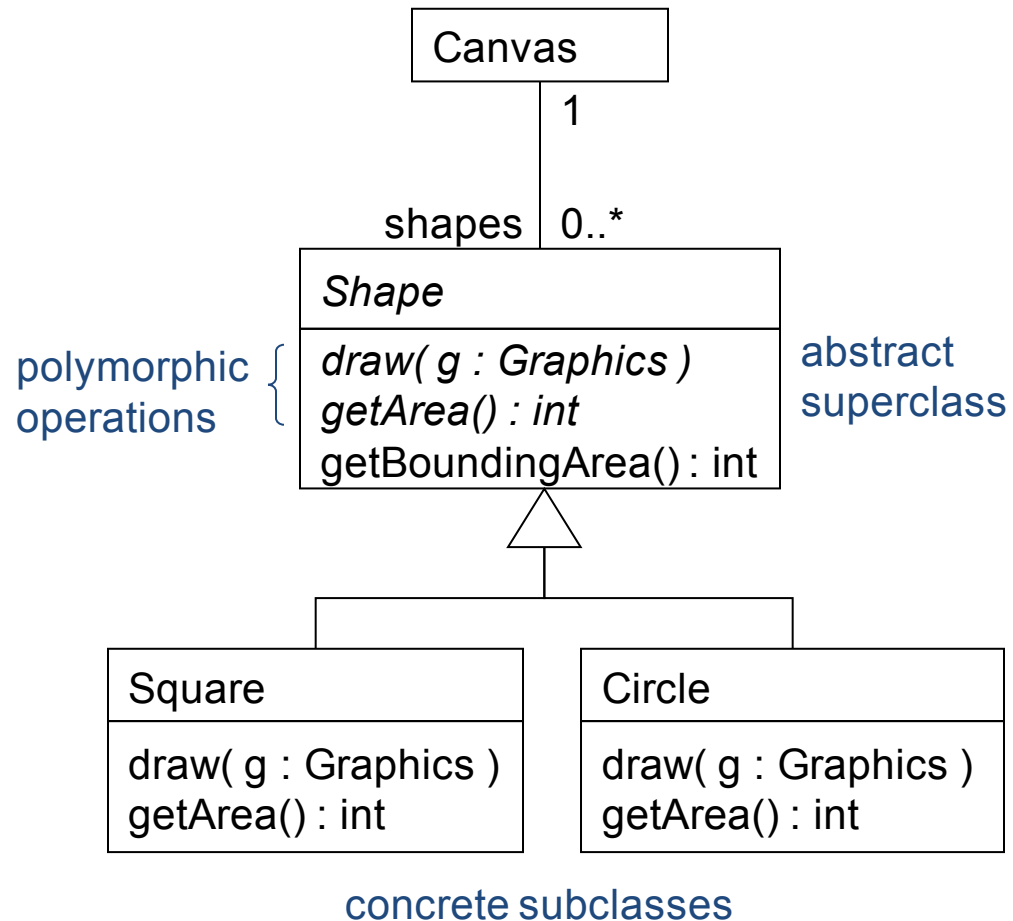
Truck

Polymorphism

- ✧ Polymorphism = "many forms"
  - ▪ A polymorphic operation has many implementations
  - ▪ Square and Circle provide implementations for the polymorphic operations *Shape::draw()* and *Shape::getArea()*
- ✧ The operation in Shape superclass defines a contract for the subclasses.

A Canvas object has a collection of *Shape* objects where each *Shape* may be a Square or a Circle

```
┌──────────────┐
│    Canvas    │
└──────────────┘
       │ 1
       │
shapes │ 0..*
┌────────────────────────────┐
│ Shape                      │        abstract
├────────────────────────────┤        superclass
│ draw( g : Graphics )       │ ← polymorphic
│ getArea() : int            │   operations
│ getBoundingArea() : int    │
└────────────────────────────┘
              △
     ┌────────┴────────┐
┌──────────────────┐  ┌──────────────────┐
│ Square           │  │ Circle           │
├──────────────────┤  ├──────────────────┤
│ draw( g : Graphics )│ │ draw( g : Graphics )│
│ getArea() : int  │  │ getArea() : int  │
└──────────────────┘  └──────────────────┘
```

polymorphic operations

abstract superclass

concrete subclasses

What happens?

- Each class of object has its own implementation of the draw() operation

- On receipt of the draw() message, each object invokes the draw() operation specified by its class

- We can say that each object "decides" how to interpret the draw() message based on its class
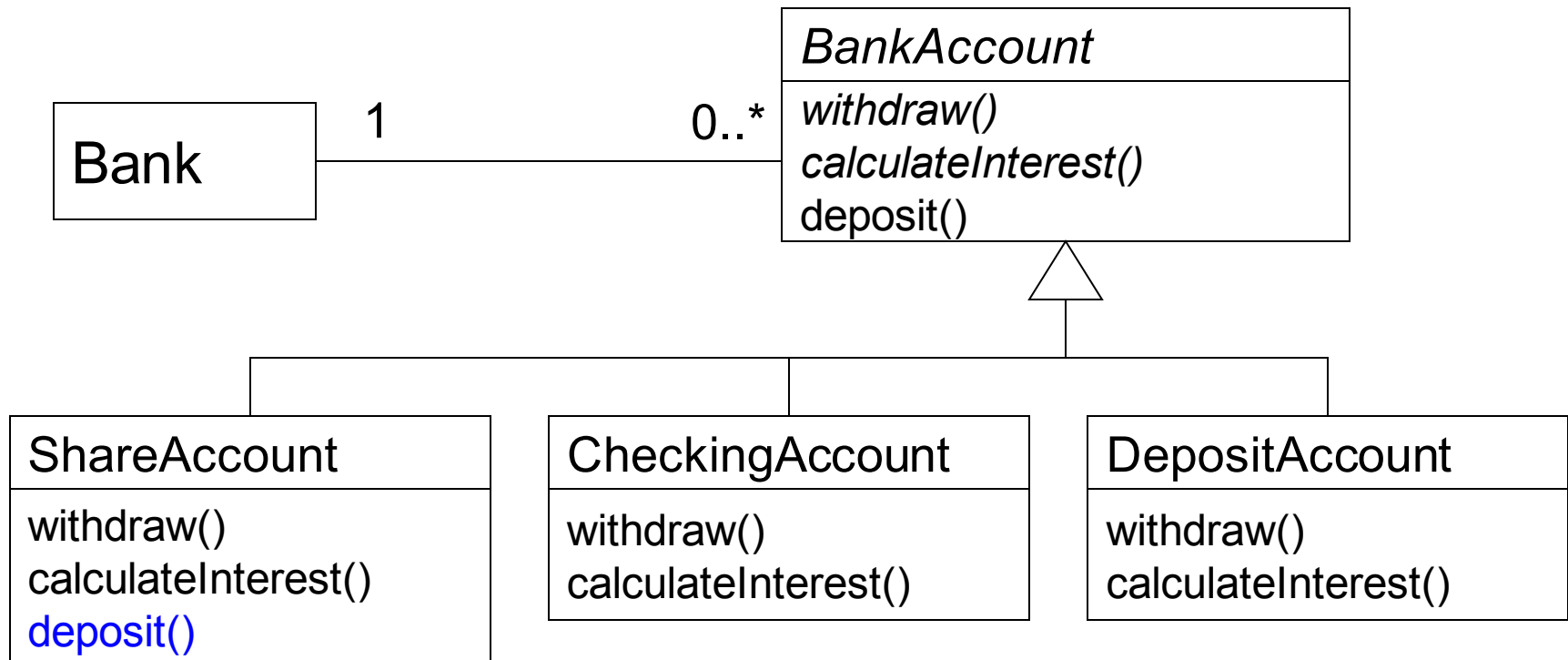
:Canvas

1.draw() → s1:Circle

2.draw() → s2:Square

3.draw() → s3:Circle

4.draw() → s4:Circle

BankAccount example

```
┌─────────────────────────────┐
│ BankAccount                 │
├─────────────────────────────┤
│ withdraw()                  │
│ calculateInterest()         │
│ deposit()                   │
└─────────────────────────────┘
```

Bank 1 ──────── 0..* BankAccount

┌──────────────────────┐ ┌──────────────────────┐ ┌──────────────────────┐
│ ShareAccount   │ │ CheckingAccount   │ │ DepositAccount   │
├──────────────────────┤ ├──────────────────────┤ ├──────────────────────┤
│ withdraw()   │ │ withdraw()   │ │ withdraw()   │
│ calculateInterest() │ │ calculateInterest() │ │ calculateInterest() │
│ deposit()   │ │   │ │   │
└──────────────────────┘ └──────────────────────┘ └──────────────────────┘
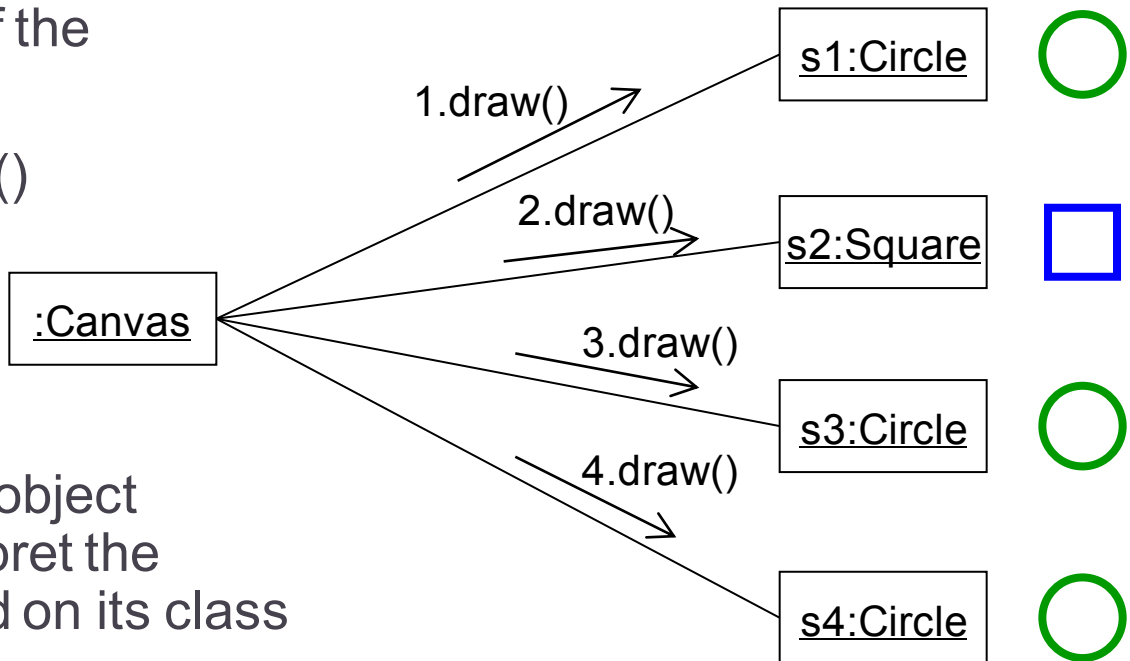
✧ We have overridden the deposit() operation even though it is **not** abstract.

Key points

◇ Generalisation, specialisation, inheritance

◇ Subclasses

- inherit all features from their parents including constraints and relationships
- may add **new features**, constraints and relationships
- may **override** superclass operations

◇ A class that can't be instantiated is an abstract class
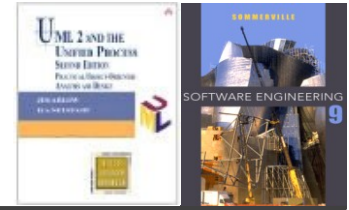
UML State Diagram

Lecture 7/Part 4

State machines

♦ Models life stages of a **single** model element – e.g. object, use case, module

♦ Every state machine exists in the context of a particular model element that:

  ▪ Has a clear life history modelled as a progression of *states, transitions* and *events*

  ▪ Responds to events dispatched from outside of the element

♦ There are two types of state machines:

  ▪ **Behavioural state machines** - define the behaviour of a model element

  ▪ **Protocol state machines** - model the protocol of a classifier

   • E.g. call conditions and call ordering of an interface that itself has no behaviour

Off   On

light bulb

Off ⟶ turnOn ⟶ On ⟶ burnOut

turnOff
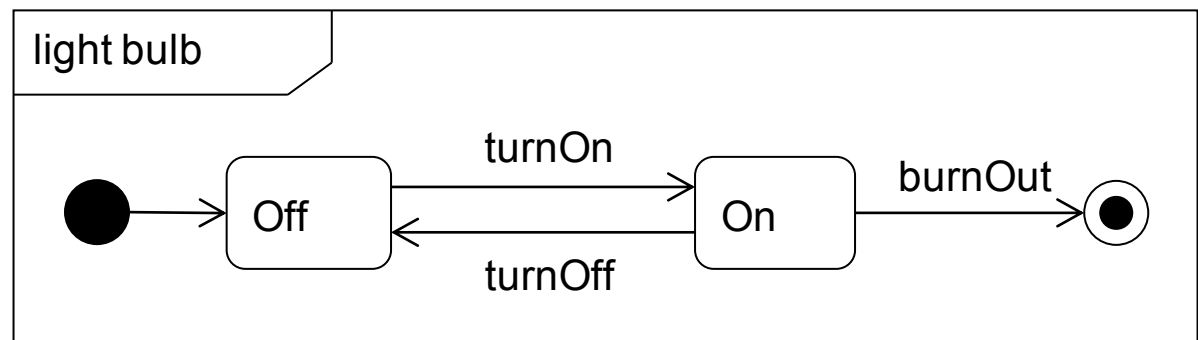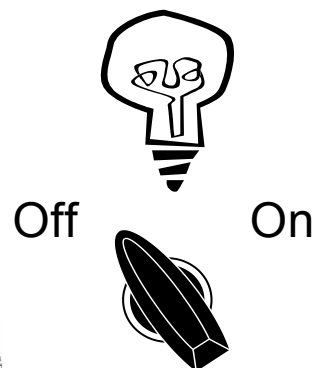
Basic state machine syntax
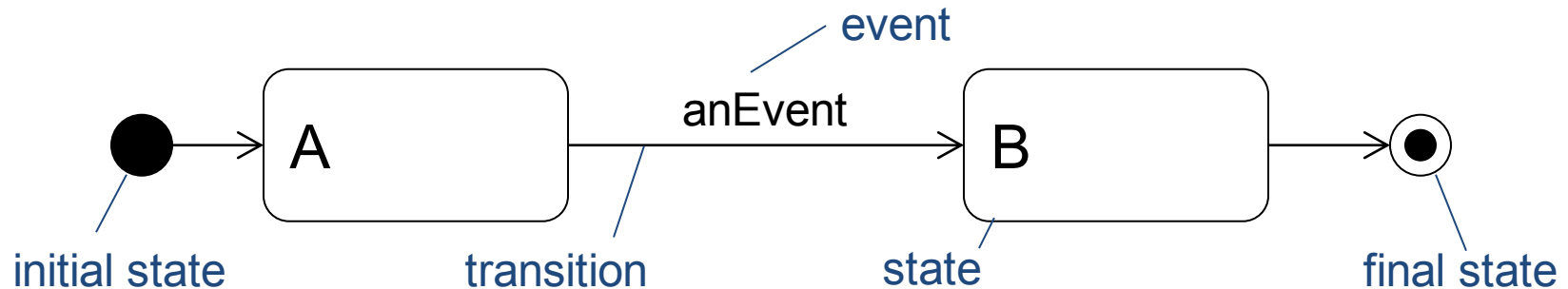
♢ State = a situation or condition during the life of an object

- Determined at any point in time by the **values of its attributes**, the relationships to other objects, or the **activities** it is performing.

♢ Every state machine should have one initial state which indicates the first state of the sequence

♢ Unless the states cycle endlessly, state machines should have a final state which terminates its lifecycle
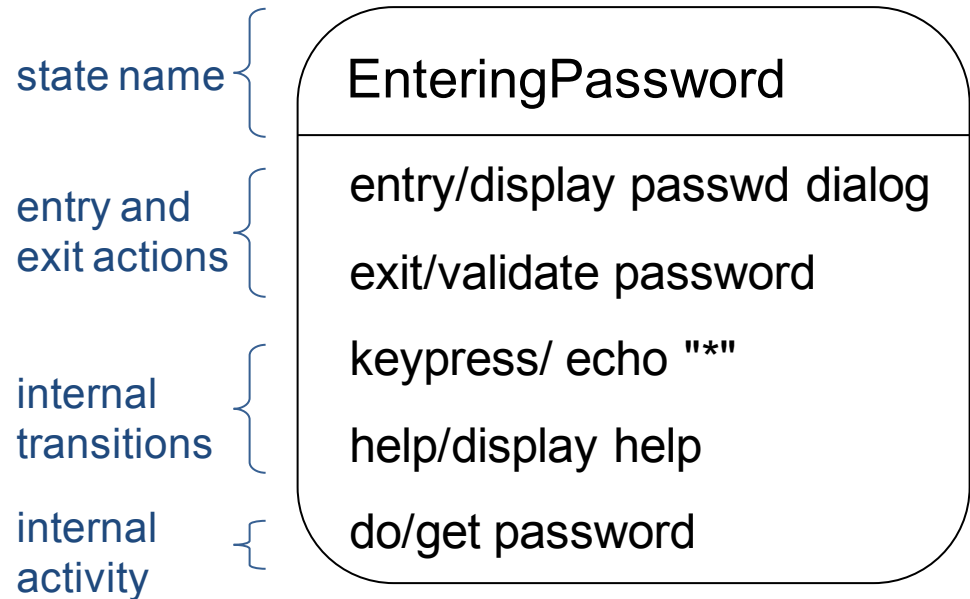
How many states?

Color
red : int green : int blue : int

State syntax

✧ Actions are **instantaneous** and **uninterruptible**

- Entry actions occur immediately on state entry
- Exit actions occur immediately on state leaving

✧ Internal transitions occur **within** the state. They do not fire transition to a new state

✧ Activities take a finite amount of time and are interruptible

state name ⟩
entry and exit actions ⟩
internal transitions ⟩
internal activity ⟩

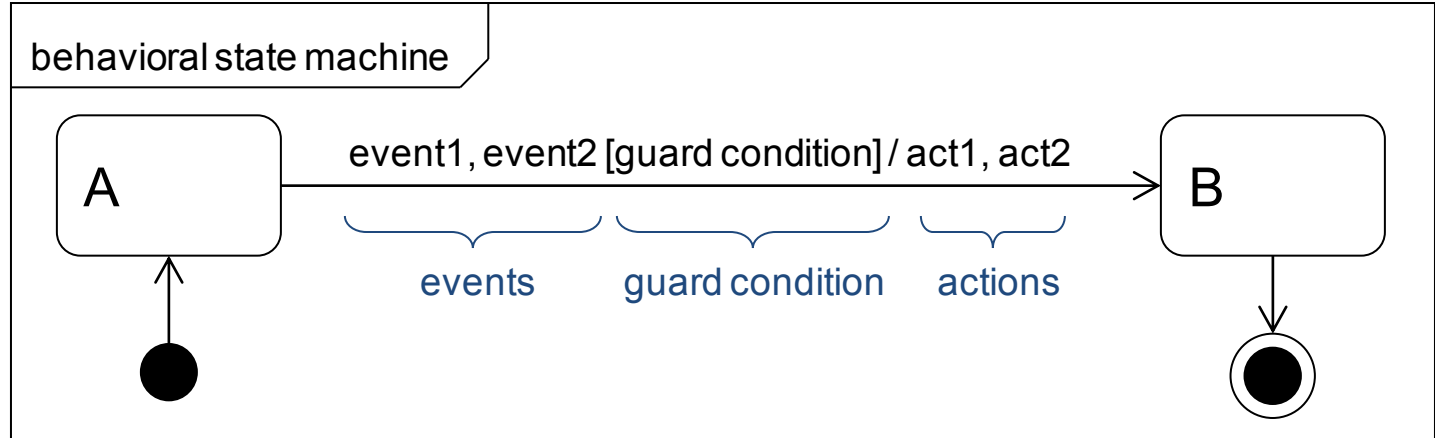| EnteringPassword |
| --- |
| entry/display passwd dialog |
| exit/validate password |
| keypress/ echo "*" |
| help/display help |
| do/get password |

Action syntax: eventTrigger / action
Activity syntax: do / activity

Transitions

Behavioral state machine

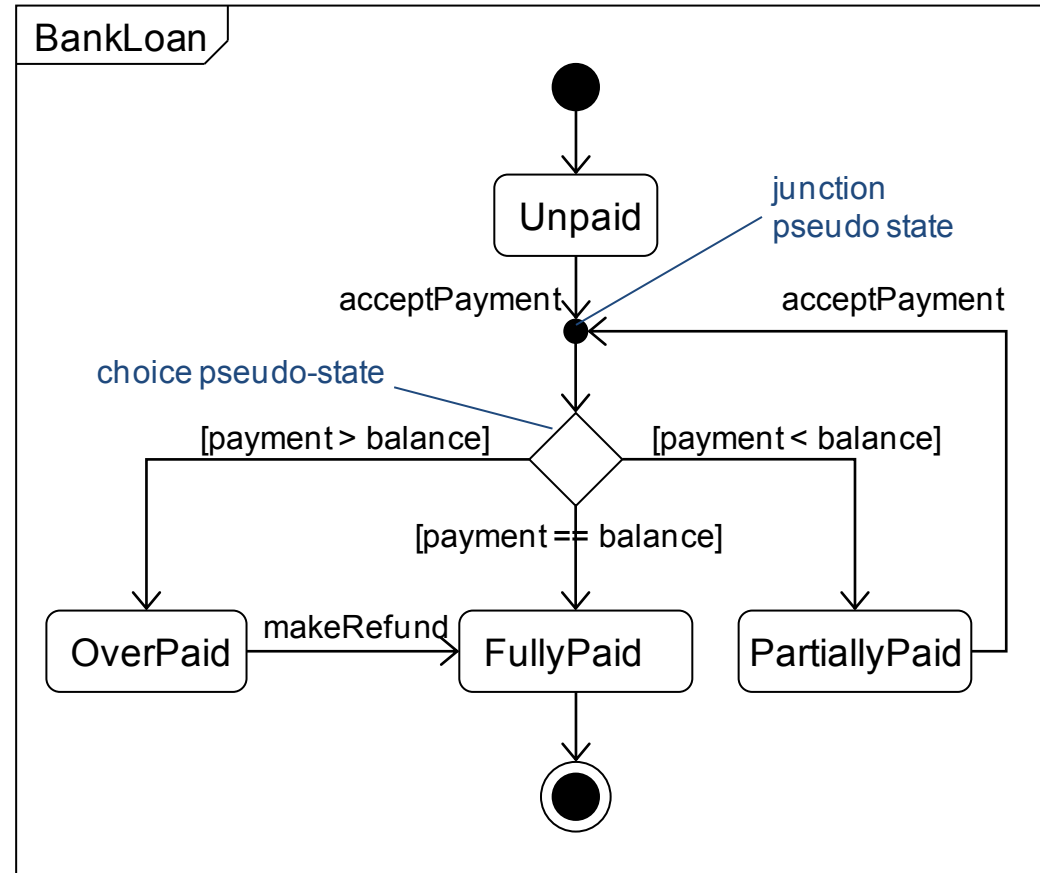Specifies object's reactions to events.

behavioral state machine

A → event1, event2 [guard condition] / act1, act2 → B

events   guard condition   actions

Protocol state machine

Specifies legal sequences of events.

protocol state machine {protocol}

C → [precondition] event1, event2 / [postcondition] → D

precondition   events   postcondition

Choice and junction pseudo states

✧ **Choice pseudo state** directs its single incoming transition to one of its outgoing transitions

- Each outgoing transition must have a mutually exclusive guard condition
- Equivalent to two outgoing transitions from one state

✧ **Junction pseudo state** connects multiple incoming transitions into one (or more) transitions.

- When there are more outgoing transitions, they must have guard conditions

BankLoan

Unpaid

junction pseudo state

acceptPayment

acceptPayment

choice pseudo-state

[payment > balance]   [payment < balance]

[payment == balance]
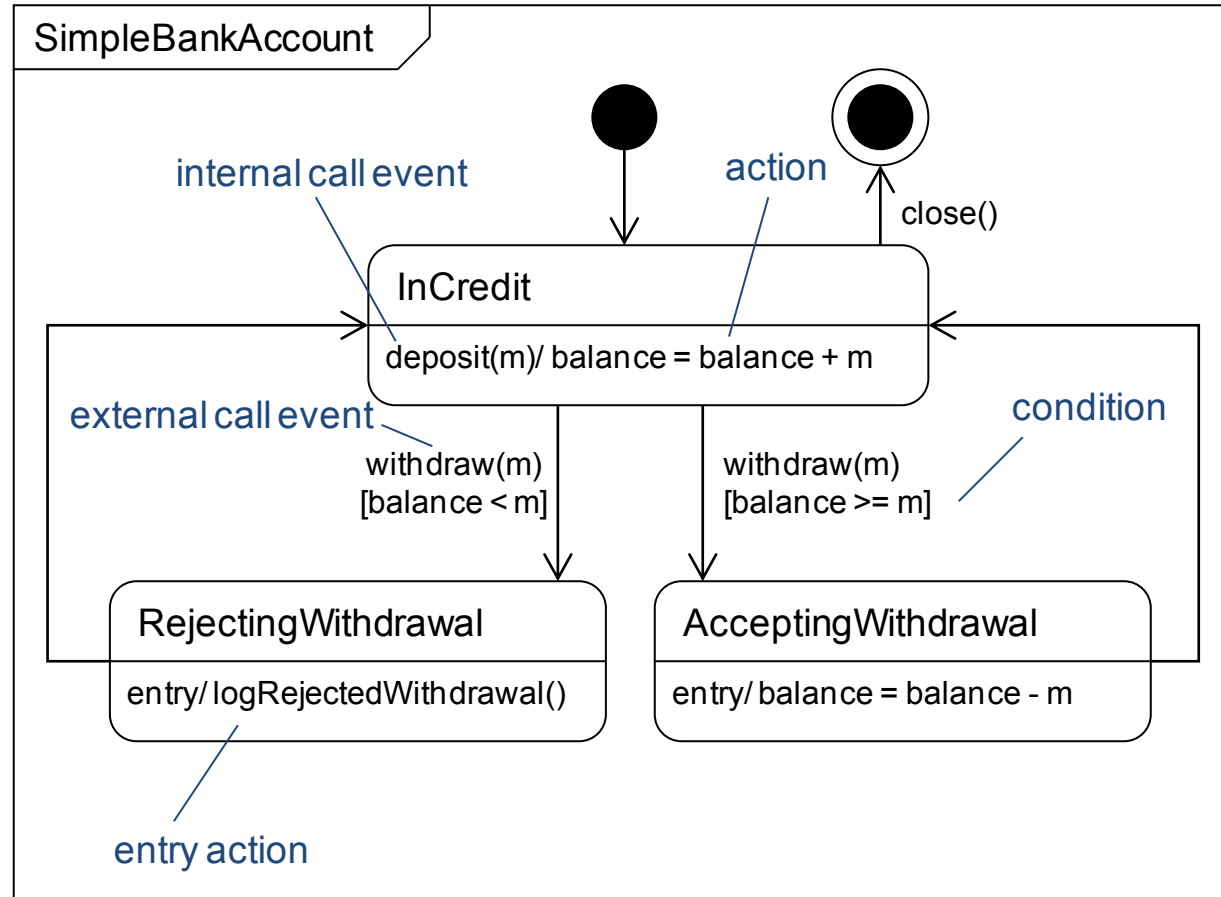
OverPaid —makeRefund→ FullyPaid   PartiallyPaid

Events

✧ "The specification of a noteworthy occurrence that has location in time and space"

✧ Events trigger transitions in state machines

✧ Events can be shown externally, on transitions, or internally within states (internal transitions)

✧ There are four types of event:

  ▪ Call event
  ▪ Signal event
  ▪ Change event
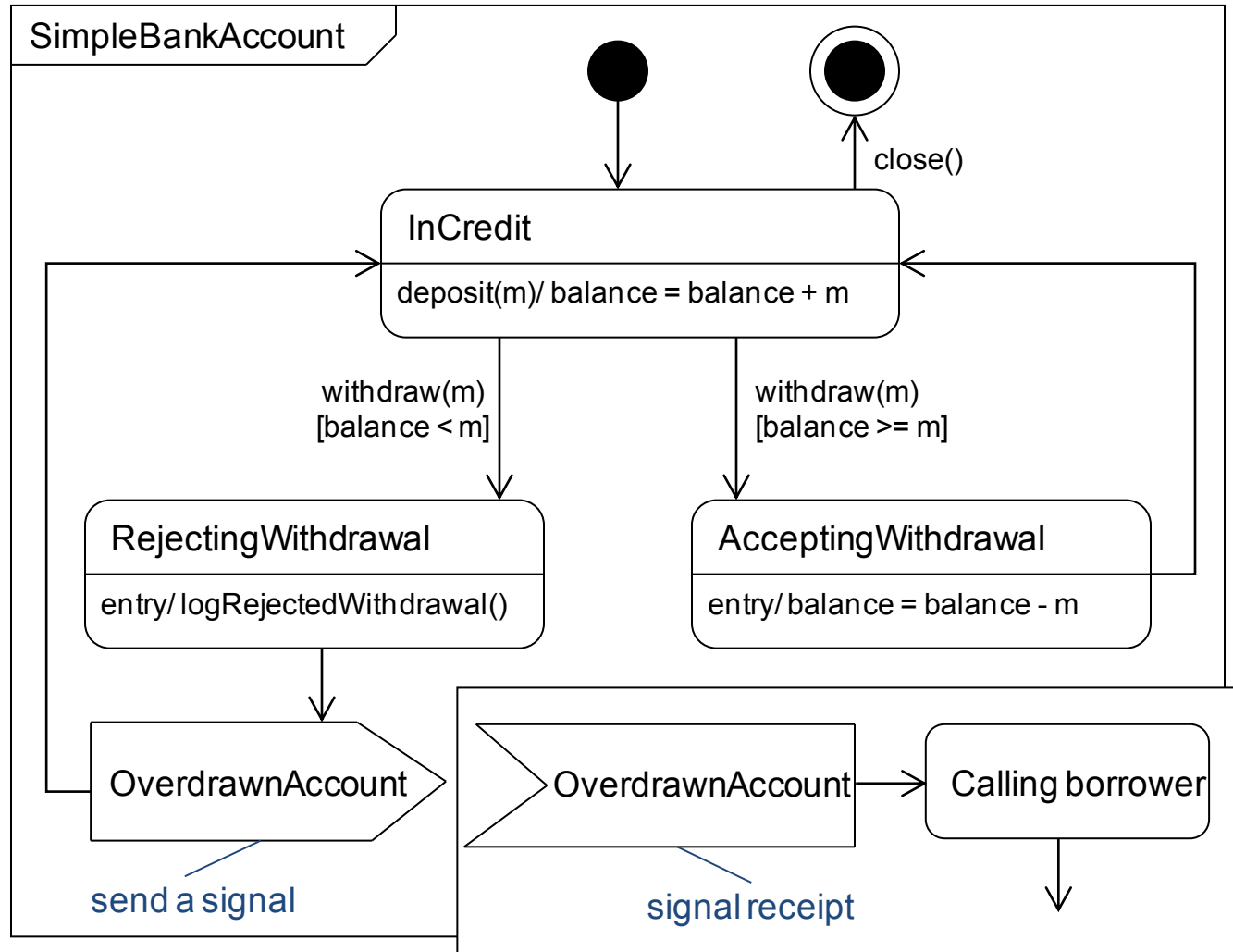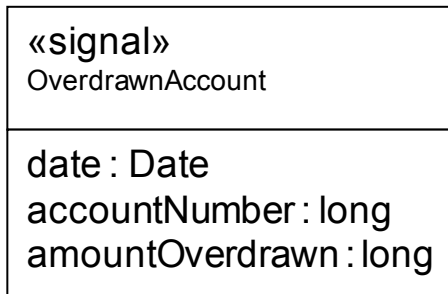 ▪ Time event

Off

turnOff   turnOn

event

On
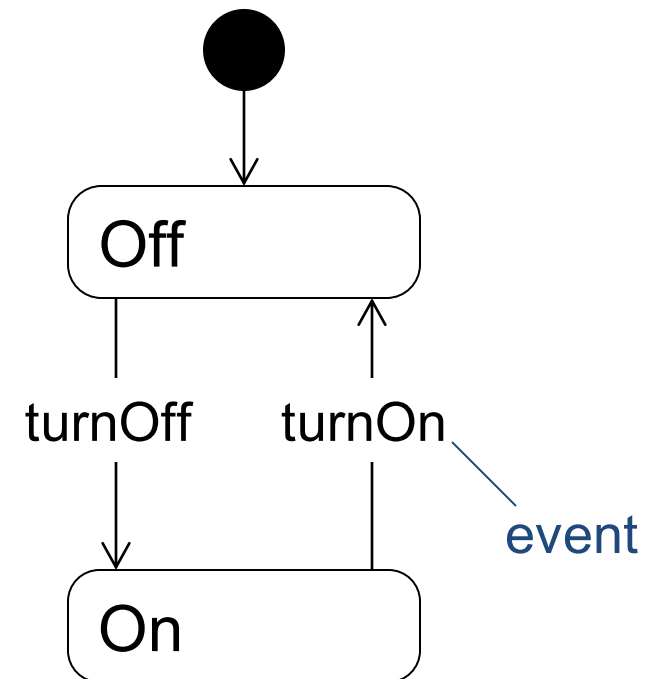
Call event

- A call for an operation execution

- The event should have the same signature as an operation of the context class

- A sequence of actions may be specified for a call event - they may use attributes and operations of the context class

- The return value must match the return type of the operation

SimpleBankAccount

internal call event

action

close()

InCredit

deposit(m)/ balance = balance + m

external call event

condition

withdraw(m)
[balance < m]

withdraw(m)
[balance >= m]

RejectingWithdrawal

entry/ logRejectedWithdrawal()

AcceptingWithdrawal

entry/ balance = balance - m

entry action

Signal events

✧ A signal is a package of information that is sent asynchronously between objects

«signal»
OverdrawnAccount

date : Date
accountNumber : long
amountOverdrawn : long

SimpleBankAccount

InCredit

deposit(m)/ balance = balance + m

close()

withdraw(m)
[balance < m]

withdraw(m)
[balance >= m]

RejectingWithdrawal

entry/ logRejectedWithdrawal()

AcceptingWithdrawal

entry/ balance = balance - m

OverdrawnAccount

send a signal

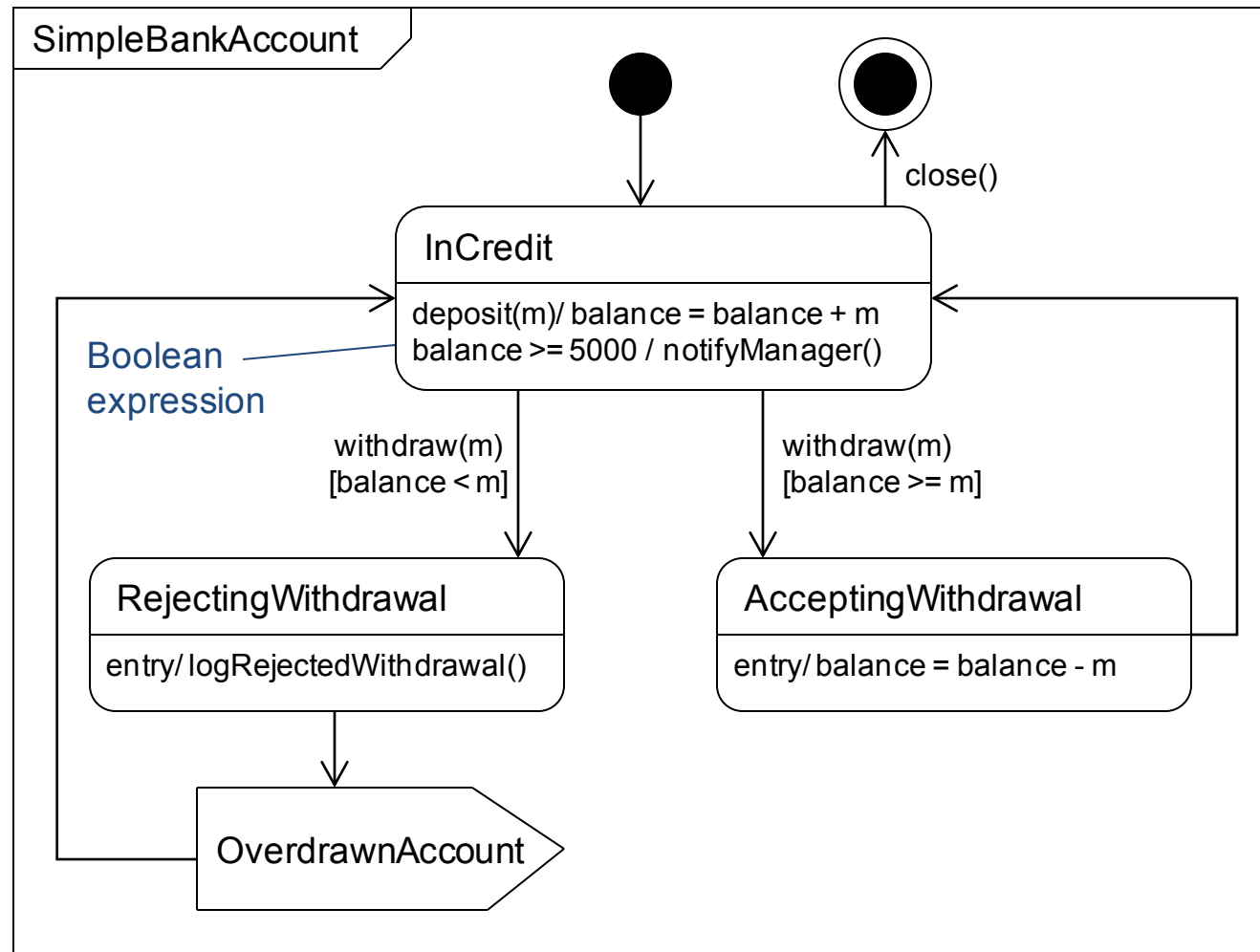OverdrawnAccount

signal receipt

Calling borrower

Change events

♦ The action is performed when the Boolean expression transitions from false to true

- The event is *edge triggered* on a false to true transition
- The values in the Boolean expression must be constants, globals or attributes of the context class
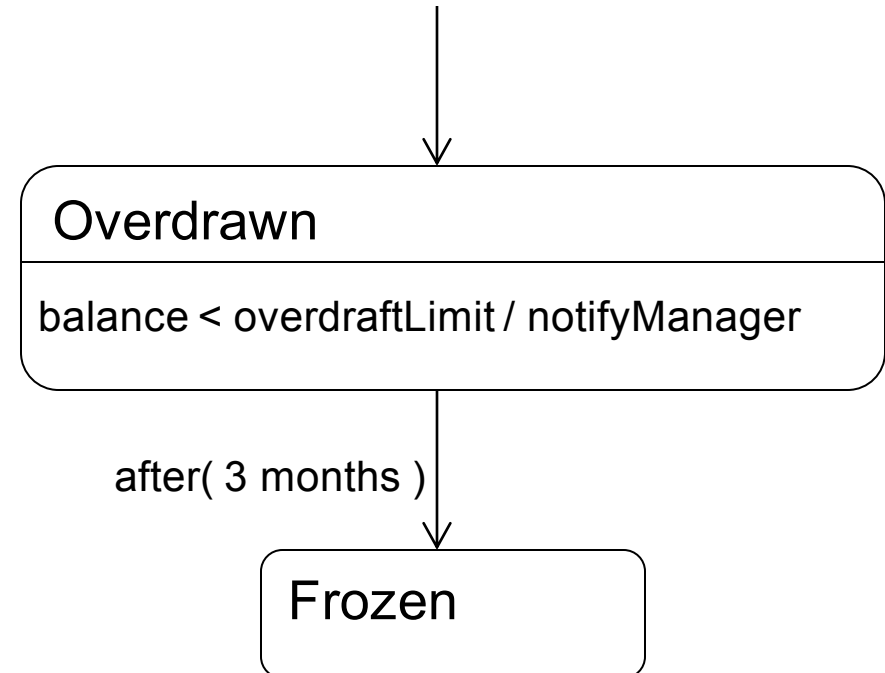
♦ A change event implies continually testing the condition whilst in the state

SimpleBankAccount

Boolean expression

close()

InCredit

deposit(m)/ balance = balance + m
balance >= 5000 / notifyManager()

withdraw(m)
[balance < m]

withdraw(m)
[balance >= m]

RejectingWithdrawal

entry/ logRejectedWithdrawal()

AcceptingWithdrawal

entry/ balance = balance - m

OverdrawnAccount

Time events

- ✧ Time events occur when a time expression becomes true

- ✧ There are two keywords, after and when

- ✧ Elapsed time:
  - ▪ after(3 months)

- ✧ Absolute time:
  - ▪ when(date =20/3/2000)

```
Overdrawn
─────────────────────────────
balance < overdraftLimit / notifyManager
```

after(3 months)

```
Frozen
```

Context: CreditAccount class

Composite states

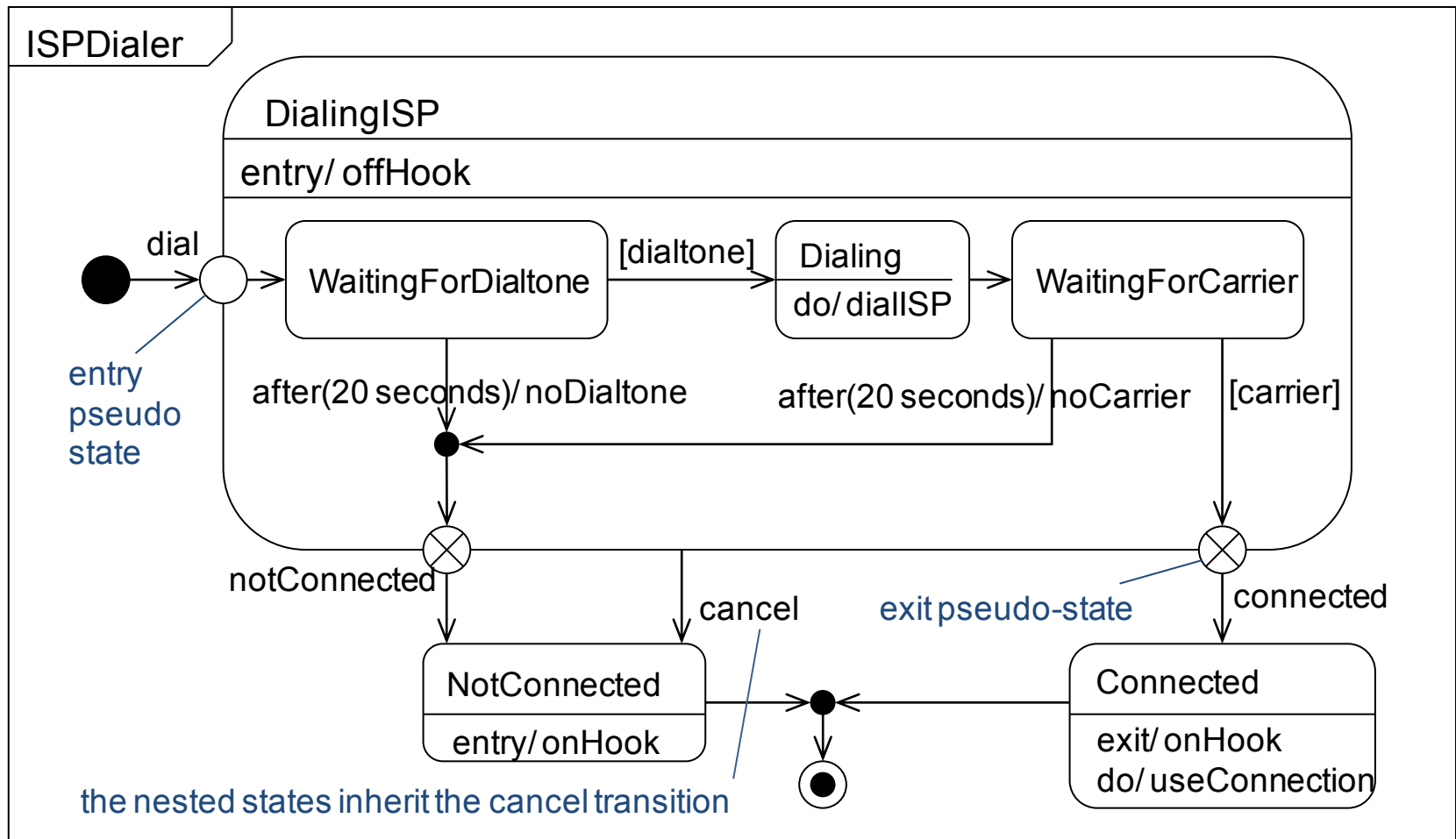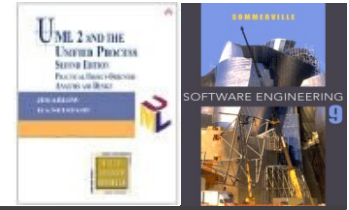✧ Have one or more regions that each contain a nested submachine

- Simple composite state
  - exactly one region
- Orthogonal composite state
  - two or more regions

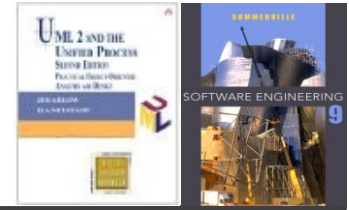✧ The final state terminates its enclosing region – all other regions continue to execute

✧ The terminate pseudo-state terminates the whole state machine

Simple composite states

ISPDialer

DialingISP

entry/ offHook

dial

WaitingForDialtone → [dialtone] → Dialing (do/ dialISP) → WaitingForCarrier

entry pseudo state

after(20 seconds)/ noDialtone   after(20 seconds)/ noCarrier   [carrier]

notConnected

cancel

exit pseudo-state   connected

NotConnected (entry/ onHook)

Connected (exit/ onHook, do/ useConnection)

*the nested states inherit the cancel transition*
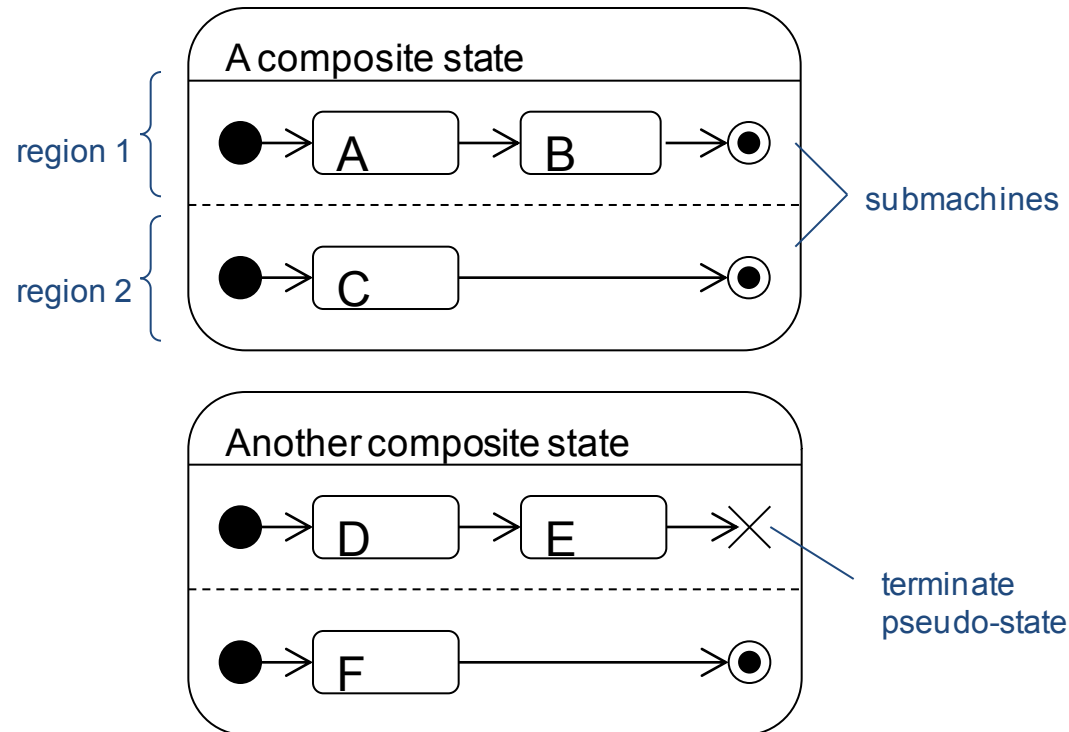
Orthogonal composite states

✧ Has two or more regions

✧ When we enter the superstate, both submachines start executing concurrently - this is an implicit fork

Synchronized exit - exit the superstate when *both* regions have terminated
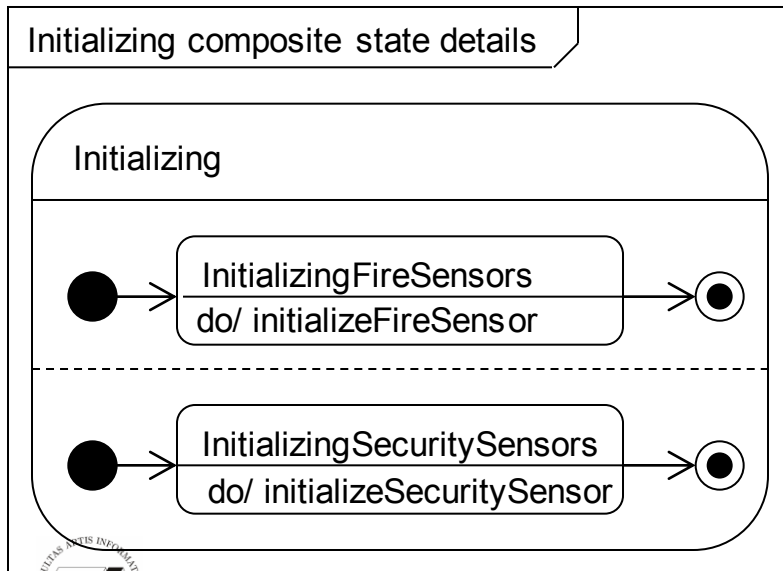
Unsynchronized exit - exit the superstate when *either* region terminates. The other region continues
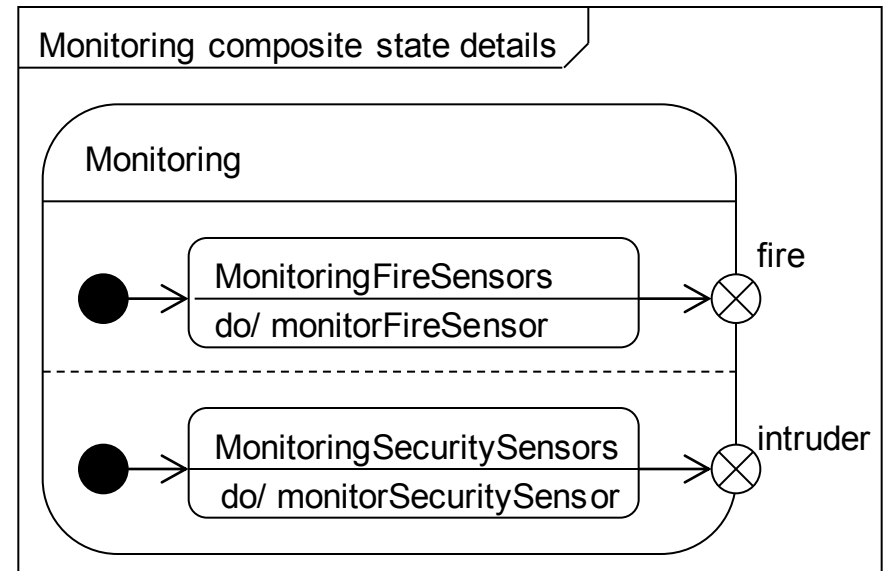
Key points

✧ Behavioral and protocol state machines

✧ States

- Initial and final
- Exit and entry actions, activities

✧ Transitions

- Guard conditions, actions

✧ Events

- Call, signal, change and time

✧ Composite states

- Simple and orthogonal composite states