IBM Software Group

Essentials of IBM® Rational® Rhapsody® for Software Engineers

UML v2.1 - Fundamentals

© IBM Corporation

Software Crisis

SW Resource Available

SW Resource Need

new technologies and
developments can‘t be
postponed, only missed

1990

2010

Development Process

Requirements Models
Requirements Analysis
Test Scenarios
Test Database
Test Scenarios
System Acceptance
Requirements Document
System/Performance Model
System Design & Analysis
System Test & Integration
HW/SW Requirements Specifications Document
Implementation Model
HW/SW Design
Module Test & Integration
Design Specification Document
HW/SW Unit Test & Implementation

Model-Driven/Model-Based Design

R

C

$v = Ri$

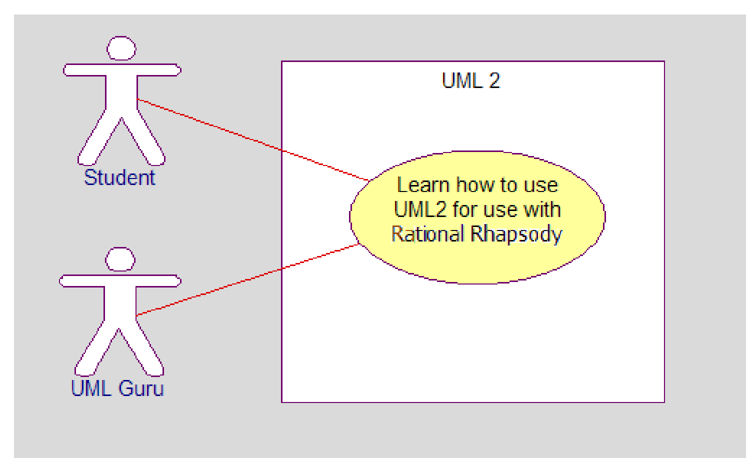
$i = C\frac{dv}{dt}$

Model

Modeling Language

UML 2 fundamentals

- This course is a one day introduction to the use of UML 2 for building embedded Real-Time software with Rational Rhapsody. The goal of this training is to understand what UML is and become familiar with the four most widely used UML diagrams which will always be needed in order to model most real-time embedded software:
  - Use Case Diagrams
  - Sequence Diagrams
  - Class / Object / Structure Diagrams
  - State Machine Diagrams



The course is generally given as either a single continuous day or interleaved with the Rational Rhapsody Tool Training. Completing the extended exercises requires an extra day.

UML 2 fundamentals part I

- What is UML?
- How do you describe structure using UML?

  Rational Rhapsody Tool Training – Hello World
- How do you describe behavior using UML part I?

  Rational Rhapsody Tool Training – Count down
- How do you describe behavior using UML part II?

  Rational Rhapsody Tool Training - Dishwasher
- How do you model communication using UML?

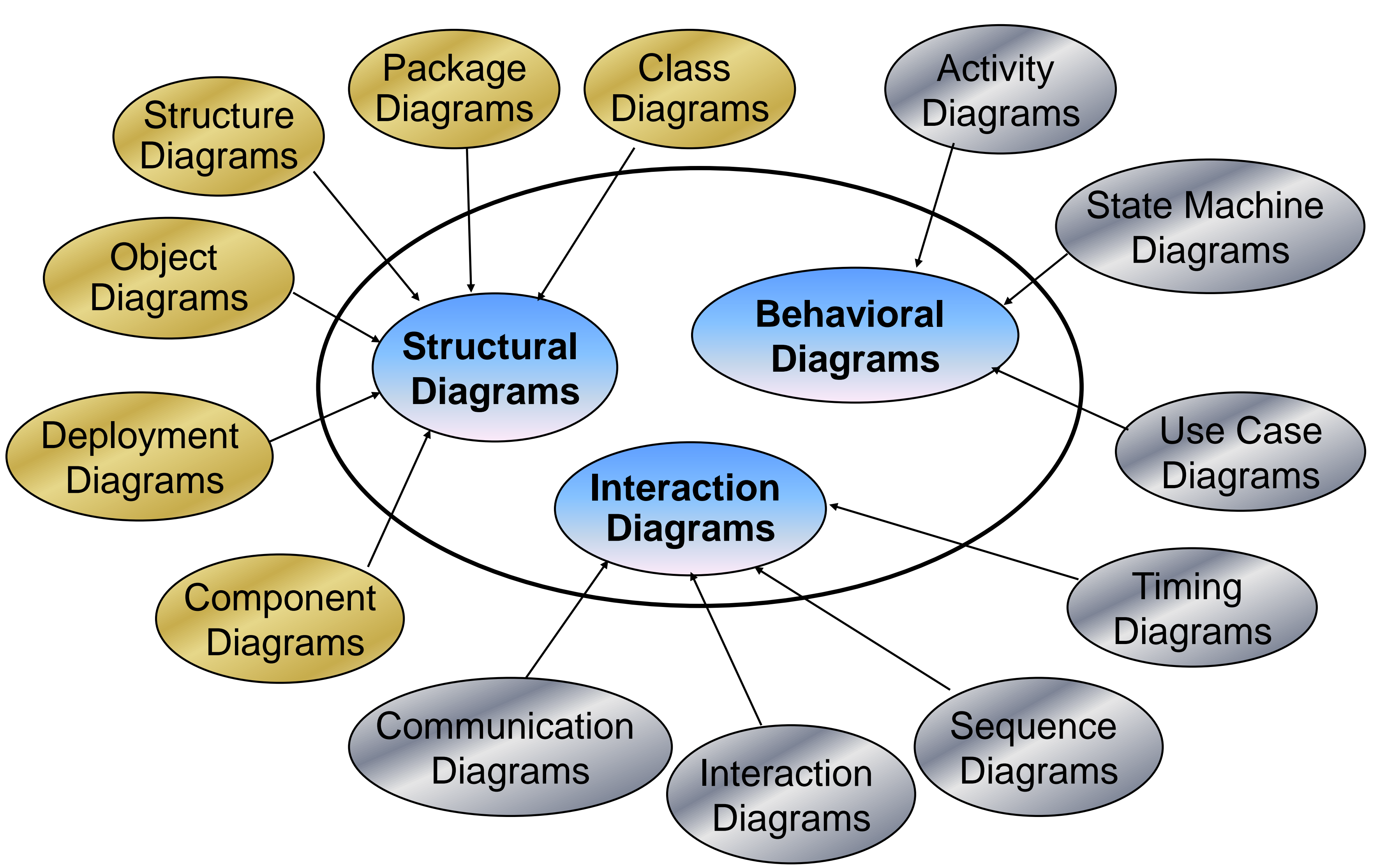
  Rational Rhapsody Tool Training – Dishwasher System

UML fundamentals

What is UML?

- **U**nified **M**odeling **L**anguage
- Comprehensive full life-cycle 3rd Generation modeling language
  - Standardized in 1997 by the OMG (Object Management Group)
  - Created by a consortium of 12 companies from various domains
  - IBM a key contributor to behavioral modeling
- Incorporates state of the art Software and Systems A&D concepts
- Matches the growing complexity of real-time systems
  - Large scale systems, Networking, Web enabling, Data management
- Extensible and configurable
- Unprecedented inter-disciplinary market penetration
- UML 2.1 is latest version

UML 2 diagrams



IBM

Use case diagram

- This diagram shows what the system does and who uses it.

Anesthesia System

Anesthesia System

Hospital Network

Physician

Display Patient Status

Alarm On Critical Event

Ventilate The Patient

Deliver Anesthesia

Deliver Anesthesia By Injection

Deliver Anesthesia By Inhalent

Remote Sugeon Display

Chart Recorder

ECG Monitor

Patient

Sequence diagram

- Sequence Diagrams show how instances communicate over time.

Observer Design Pattern

ENV
Observable
ObserverA
ObserverB
newData()
subscribe(this)
newData()
notify()
subscribe(this)
newData()
notify()
notify()

Class diagram

- Class diagrams show classes and relations between them.

Command Overview

OMOSMutex

itsMutex 1

Writer

fills 1

empties 1

Reader

CommandList

priority

*

«Interface»
Command

priority:int
identity:int

execute():v...

«Usage»

«Usage»

Object diagram

- Object Diagrams show instances of classes and show which ones are linked to others at run time.

CycleComputer Overview

1 Sensor

1 TripTimer
resetTripTimer():void
update():void
pulse():void
getAverageSpeed():long
getCurrentSpeed():long
getMaximumSpeed():long
getTotalDistance():long
getTripDistance():long
Init()

1 CycleComputer
units:int
circumference:int
evUpdate():void
evRight():void
evLeft():void
evLeftHeld():void
evRightHeld():void

1 Keyboard
evLeftPress():void
evLeftRelease():void
evRightPress():void
evRightRelease():...

1 RTClock
setTime(h:int,m:int,s:int):void
incrementMinutes(n:int):void
incrementHours(n:int):void
clockFlash(n:int):void
display():void
evSetTime():void

«Interface»
Display
clearDigits():void
clearSymbols(symbols:int):void
showDigits(thousands:int,hun...
showSymbols(symbols:int):void

<<Usage>>

Structure diagram

- This diagram shows the internal structure of classes.

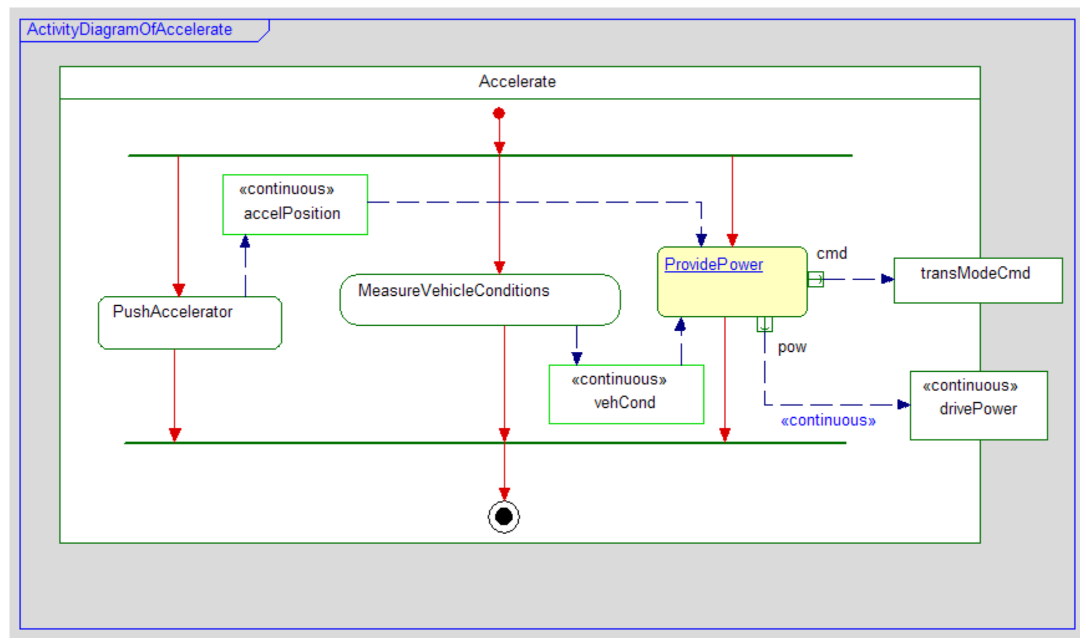
State machine diagram

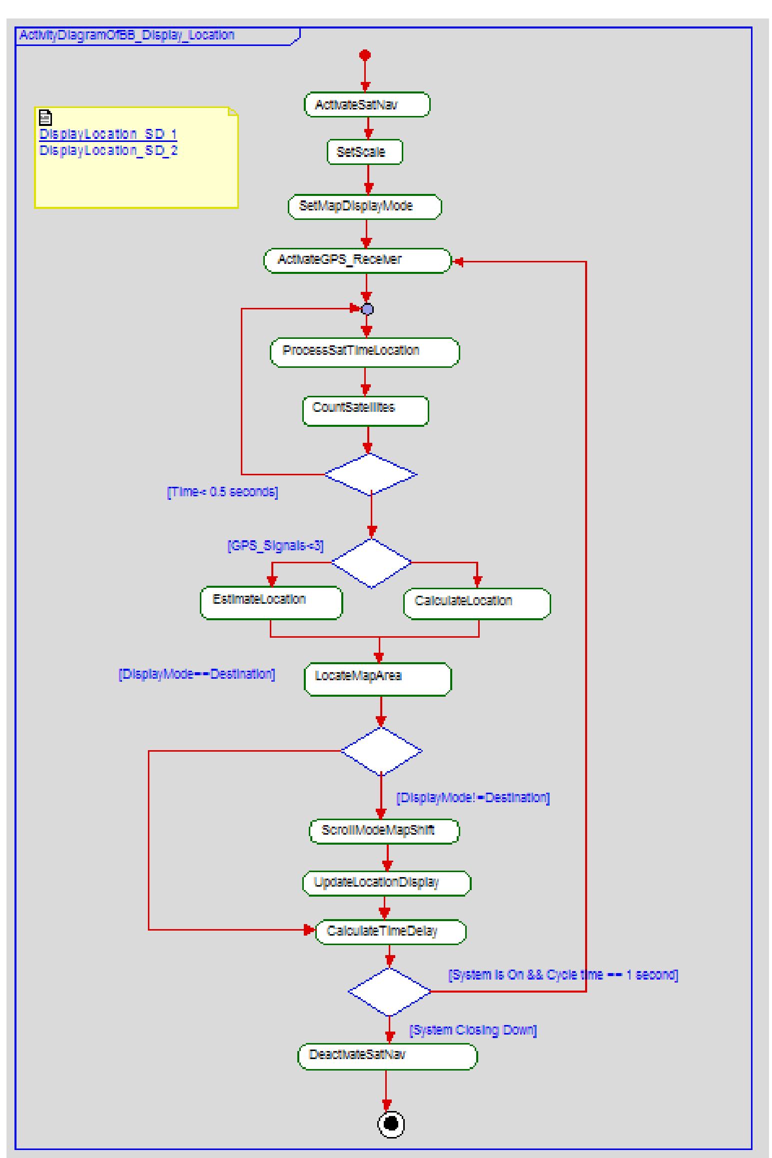
- State machines are used when you need to wait until something happens before going to a different state.

StatechartOfMobilePhone
off
evOnOff
evOnOff
on
Disconnected
evDisconnect
evConnect/
OPORT(blue)->GEN(evConnected);
Connected
tm(1000)/
OPORT(blue)->GEN(evAlive);
page
welcomePage
tm(2000)
volumePage
evIncVolume/
incVolume();
evDecVolume/
decVolume();

Activity diagram

- Activity diagrams are used to describe behavior for operations, classes, or use cases. As soon as one activity finishes, the next one starts.





Package diagram

- A package is similar to a folder and is used to organise the UML model elements.

Packages Overview

BuildPkg

TestSubsystemAPkg

SubsystemAPkg

TestSubsystemBPkg

SubsystemBPkg

TestSubsystemCPkg

SubsystemCPkg

TestSubsystemDPkg

SubsystemDPkg

TestSubsystemEPkg

SubsystemEPkg

TestSubsystemFPkg

TypesPkg

SubsystemFPkg

InterfacesPkg

Communication diagram

- This used to be known as a *Collaboration Diagram* and is similar to a Sequence Diagram, but generally less popular.

Reading a command

:Builder

1. evRead()

2. read()

2.1. aCommand=get()

2.1.3. removeItsCommand(aCommand)

2.1.4. print()

itsReader:Reader

itsCommandList:CommandL

2.1.2. get(aCommand)

2.1.1. lock()

2.1.4.1. getIdentity()

2.2. execute()

2.1.5. unlock()

2.3. destructor()

2.1.4.2. getPriority()

aCommand:Command

itsMutex:OMOSMutex

Command

Component diagram

- A component diagram shows how components such as .exe, .dll, .lib, are interconnected.

«Executable»
Gui

MFCgui

MFCgui.rc

MFCguiDlg

StdAfx

<<Usage>>

«Library»
Controller

Deployment diagram

- A Deployment Diagram shows how UML artifacts are deployed onto hardware nodes.

Arm9
300MHz
32MBytes ROM
8 MBytes RAM

HighPerformancePDA

«exe» Pda

«lib» Bluetooth

«lib» Wifi

«lib» Power

«lib» USB

MemoryCard

SD

usb / wifi / bluetooth

PC

usb

Arm7
50MHz
4 MBytes ROM
2 MBytes RAM

LowCostPDA

«exe» Pda

«lib» USB

«lib» Power

Timing diagram

- The Timing Diagram focuses on changing conditions within and among objects on a linear time axis.

send(value)
Transceiver
Idle
Receiving
Sending
Coil Driver
transmit(value)
evDone
Receiving::High
Receving::Low
{1 ms +/- 0.2ms}
Sending::High
Sending::Low
Tristate
tm(bitTime)
{3 ms +/- 0.2ms}
Monitor
send(value)
Initializing
Acquiring
Reporting
Idle
Time

Interaction overview diagram

- An Interaction Diagram is a mixture of an Activity Diagram and several Sequence Diagrams.

sd
Alarm
EventQueue
send(anEvent)
tm(200)
resend(anEvent)

sd
Clock
EventQueue
getEvent()
dispatch(evEvent)

ref
dispatch_event

How does UML apply to real-time?

- Real-Time UML is ***standard UML***
  - “Although there have been a number of calls to extend UML for the real-time domain … experience had proven this is not necessary.” Bran Selic, *Communications of the ACM*, Oct 1999
- Real-time and embedded applications
  - Special concerns about quality of service (QoS)
  - Special concerns about low-level programming
  - Special concerns about safety and reliability
- Real-time UML is about applying the UML to meet the specialized concerns of the real-time and embedded domains

How do you describe structure using UML?

What is an object ?

- An object is one of the common building blocks in a UML model. It can represent a system, a subsystem or a specific software class in a programming language.

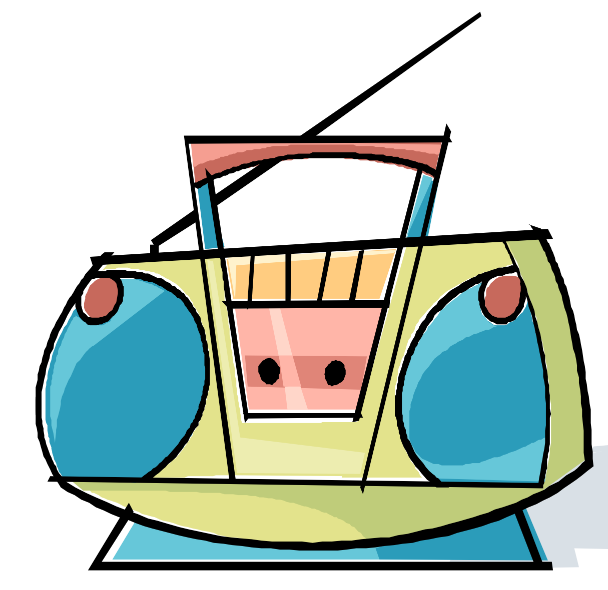
- Several definitions are available:
  - An object is a real-world or conceptual thing that has autonomy.
  - An object is a cohesive entity consisting of data and the operations that act on those data.
  - An object is a thing that has an interface that enforces protection of the encapsulation of its internal structure.

What is an object ?

- Every object has:
  - Responsibilities:
    - What does the object do? Why does it exist?
  - Attributes:
    - This is the internal data and can either be fixed in value or can vary in value.
  - Behaviors:
    - These are actions performed by the object to fulfil its responsibilities.

A radio

- Responsibilities:
  - Allow the user to listen to desired radio frequency
- Attributes:
  - Wavelength
  - Frequency
  - Volume
- Behaviors:
  - Tune to a frequency
  - Store / Recall a frequency
  - Change volume
  - Switch on / off

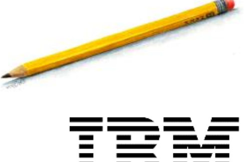


A digital camera

- Responsibilities:
  - Take digital photos
- Attributes:
  - Available Memory
  - Picture Resolution
  - Battery Level
- Behaviors:
  - Select Resolution
  - Focus
  - Take Photo
  - Upload Photos

A microwave oven

- Responsibilities?
- Attributes?
- Behaviors?



IBM

Objects can be …

- Software things
  - Occupy memory at some point in time
  - For example, CustomerRecord, ECGSample, Window, Font
- Electronic things
  - Occupy physical space at some point in time
  - For example, Thermometer, LCDDisplay, MotionSensor, DCMotor
- Mechanical things
  - Occupy physical space at some point in time
  - For example, WingSurface, Gear, Door, HydraulicPress
- Chemical things
  - Occupy physical space at some point in time
  - For example, Battery, GasMixture, Halothane
- System things
  - Occupy physical space at some point in time
  - For example, PowerSubsystem, RobotArm, Space Shuttle

Object identity

- All objects are unique even if their attributes are the same as another.
- For example, in this room, there are probably many mobile phones, of which several might well be identical. Even if they are identical, they are all unique.

Object views

- Objects have generally two different views:
  - Public view:
    - This is the view that can be seen from outside of the object.
  - Private view:
    - This is the internal only view, access is controlled and the details are hidden from the outside world.

Object interface

The Public view of an object is the interface that the object exposes to the outside world and which other objects can use to communicate with it.

Objects contain their own attributes and generally do not allow other objects to directly manipulate these attributes.

If other Objects need access to the attributes, then the object provides public operations (known as getters and setters) to manipulate these attributes.

This allows the freedom to change the type of the attribute without the clients knowing about it or indeed having to make any changes. All that is needed is to modify the getter and setter to manipulate the new attribute type.

Getters and Setters are sometimes called *Accessors* and *Mutators.*

Object attributes

- Attributes are *typed* values holding information known to the object, for example:
  - value: int
  - patientName: string

- All objects of the same type have the same *set of* attributes but different *copies* (hence they may have different *values*).

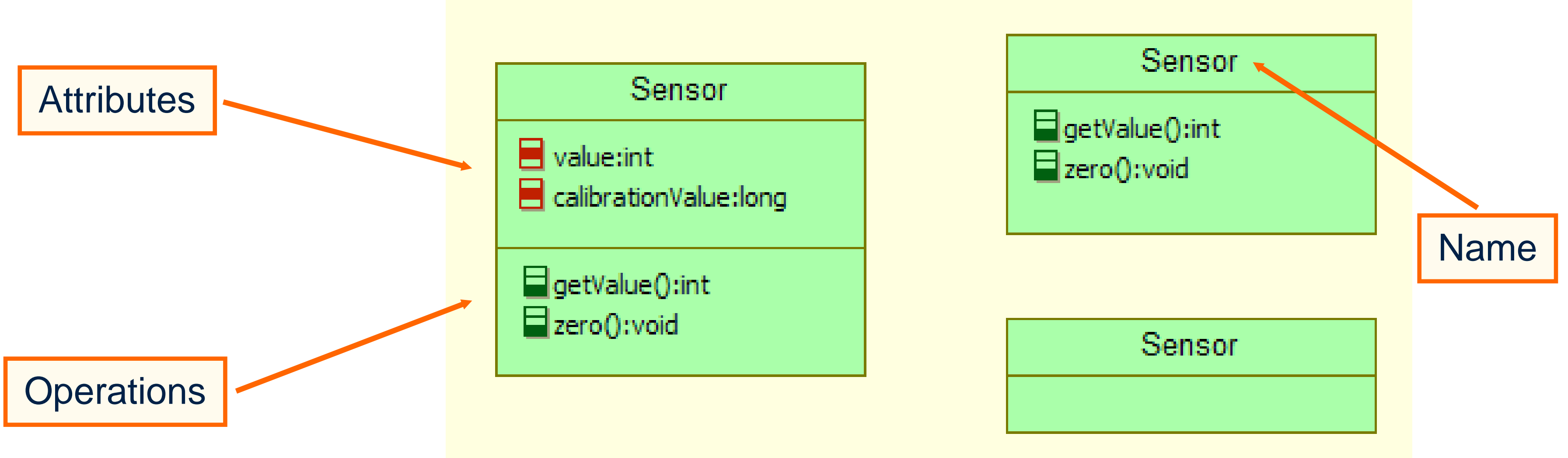
- Attributes are primitive in structure and behavior:
  - If they are rich, then they should be themselves objects.

Object operations

- Objects execute operations to implement behavior:
  - Operations are primitive behaviors such as:
    - ++x
    - y = sin(x)^2 + cos(x)^2
    - z = mySensor->getValue()
- Operations can manipulate the attributes, call other operations.
- Operations can be invoked from other objects as well as from a state machine or activity diagram attached to the object.

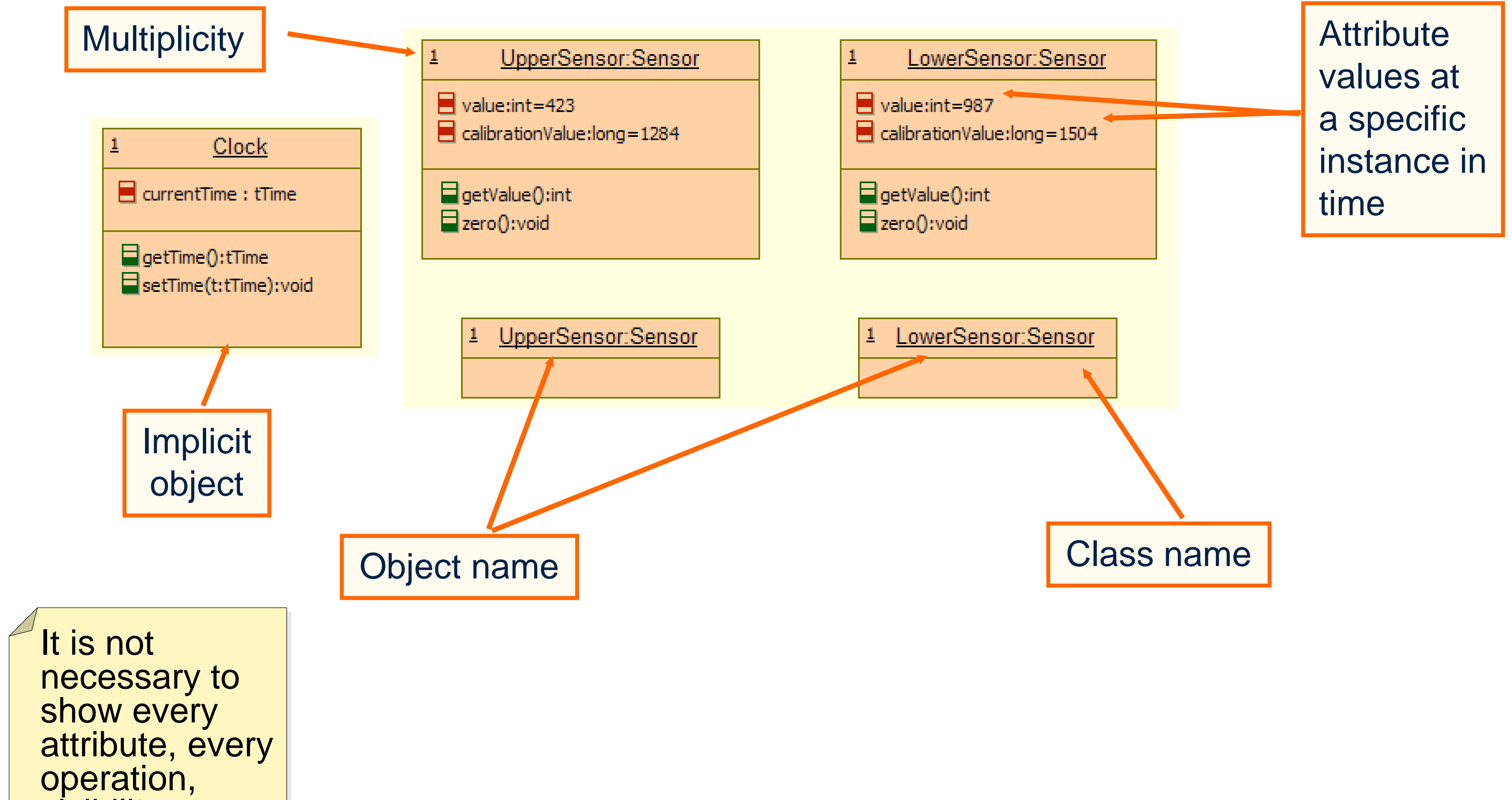
UML class

- A Class is the *definition* or *specification* of an object.
- An object is an *instance* of a class.
- An object has the attributes and behaviors defined by its class.
- A class can be shown on a Class Diagram in one of many ways.
- On some diagrams, you might want to show just the name of the class.
- Sometimes, you might want to show just the operations; other times, you might want to show all attributes and all operations.



UML object

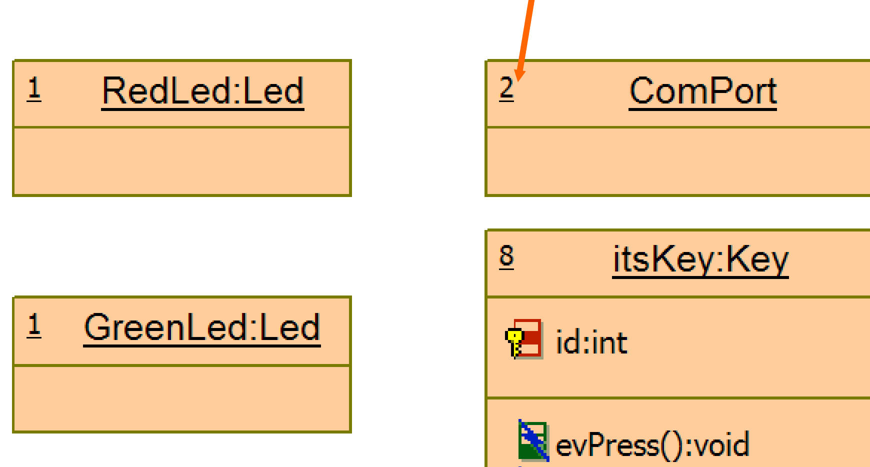
- As with a class, an Object (also known as an Instance) can also be shown on a Class/Object Diagram in many ways:



IBM

Multiplicity

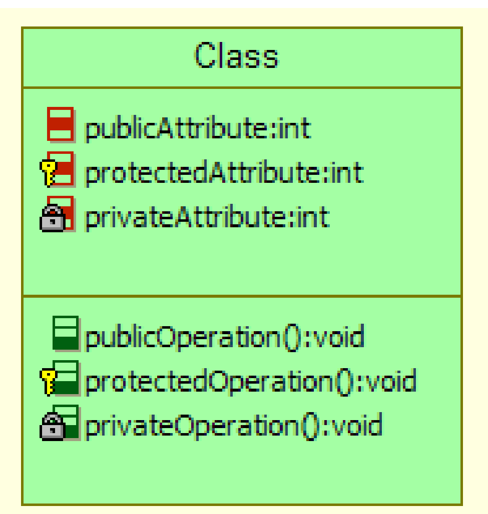
- When showing Objects, you need to indicate how many objects there are.
- This is done with the Multiplicity, which most of the time is 1.



Visibility defined

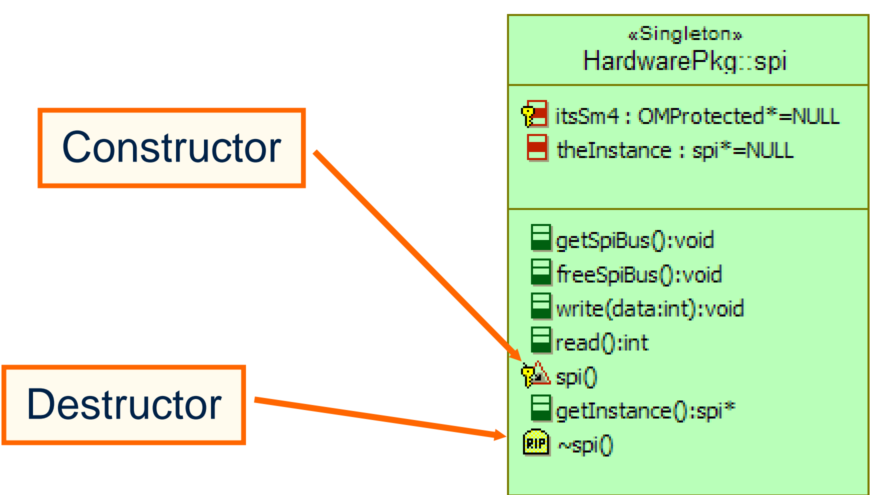
- Attributes and Methods or Operations are *features* of a class.
- Features have the visibility adornments:
  - + public
    - Accessible by any client of the class.
  - # protected
    - Accessible only from within the same class or subclasses.
  - - private
    - Accessible only from within the same class.

Rather than using these symbols, Rational Rhapsody uses more graphical ones which are easier to understand.



Constructors and destructors

- Every Class has a couple of special operations allowing Objects to be constructed / destroyed.
- A constructor is always called when an Object is created and generally initializes its attributes.
- A destructor is always called when the Object is destroyed and ensures that any allocated resources are properly returned.



Static and *abstract* operations

- A static operation is shown underlined.
- An abstract operation is shown in *italic.*

DataLogger

log(aMsg:char,aTime:tTime):void*
~DataLogger()
DataLogger()
clearLog():void

Abstract operation

«Usage»

«Singleton»
Printer

«Guarded» print(aMsg:char*):void
«Guarded» println(aMsg:char*):void
«Guarded» print(n:int):void
«Guarded» println(n:int):void
theInstance():Printer
Printer()

Static operation

Exercise 1

- What are the attributes, operations and responsibilities of the following?

Timer

Led

ComPort

How do you describe behavior – Part 1?

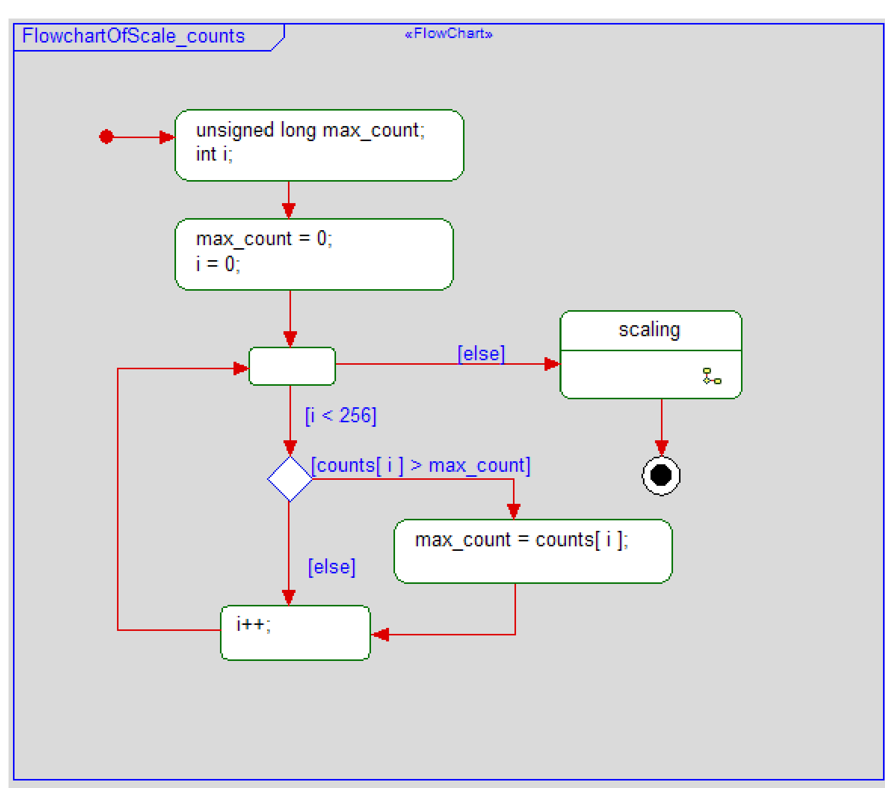
Hello World

Types of behavior

- Behavior can be *simple:*
  - Simple behavior does not depend on the object's history.
- Behavior can be *continuous:*
  - Continuous behavior depends on the object's history but in a smooth, continuous fashion.
- Behavior can be *state-driven:*
  - State-driven behavior means that the object's behavior can be divided into disjoint sets.

Simple behavior

- Simple behavior is not affected by the object's history:
  - cos(x)
  - getTemperature( )
  - setVoltage(v)
  - Max(a,b)
  - $\int_a^b e^{-x^2} dx$



- It may be represented by activity diagrams, if desired.

Continuous behavior

- Object's behavior depends on history in a continuous way

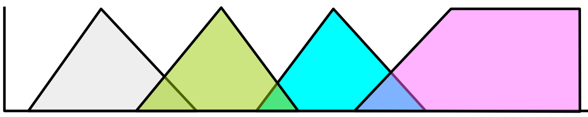
  - PID control loop:

    X_n K_a Y_n + - W_n Delay V_n + + Z_n K_b

  - Digital filter:

$$f_j = \frac{d_j + d_{j-1} + d_{j-2} + d_{j-3}}{4}$$

  - Fuzzy logic:
    - Uses partial set membership to compute a smooth, continuous output

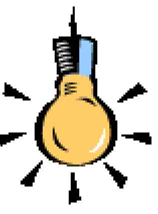


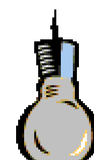
UML is not very good at describing continuous behavior.

State behavior

- Useful when an object
  - Exhibits discontinuous modes of behavior
  - Waits for and responds to incoming events

- For example a Light can be :
  - Off
  - On
  - Flashing



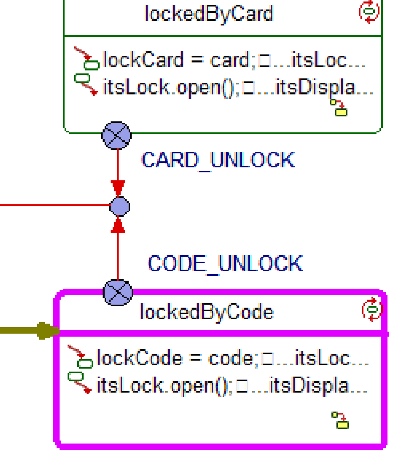


Why use state machines?

- A state machine is an abstraction of the desired behavior of a system with actions, activities, and constraints.
- An object's behavioral possibilities are defined in a primitive way by its defined operations.
- A state machine constrains the use of those operations into particular sequences.
- Thus constrained, the object's behavior is more understandable and more easily tested.
- State machines are a more abstract view of behavior, based on the object's perspective not the implementation.

State machines are executable

- Because state machines are formally defined, they form executable models.
- State machines can be executed and visualized at the design level of abstraction.
- State machines can be animated - their dynamic behavior shown graphically:
  - Standard debugging can be done, such as setting breakpoints, step through, step into, and so on.
- You can focus on the abstract behavior:
  - Was the door locked with a card or a code?
  - Rather than, is some variable 0 or 1?
- State Machines provide for easy testing.



State machine execution

active>

fillingUpper>

tm(5000)

evFull

upperFull>

upper

lower

EMPTY

empty

evStart

evEmpty

emptying>

tm(50)

EMPTY

fillingLower>

tm(1000)

lowerFull>

evTerminate

T

H

H

evPause

evResume

evAbort

pause

States / transitions / actions

- What is a state?

A **state** is a distinguishable, disjoint, orthogonal condition of existence of an object that persists for a significant period of time.

- What is a transition?

A **transition** is a response to an event of interest moving the object from a state to a state.

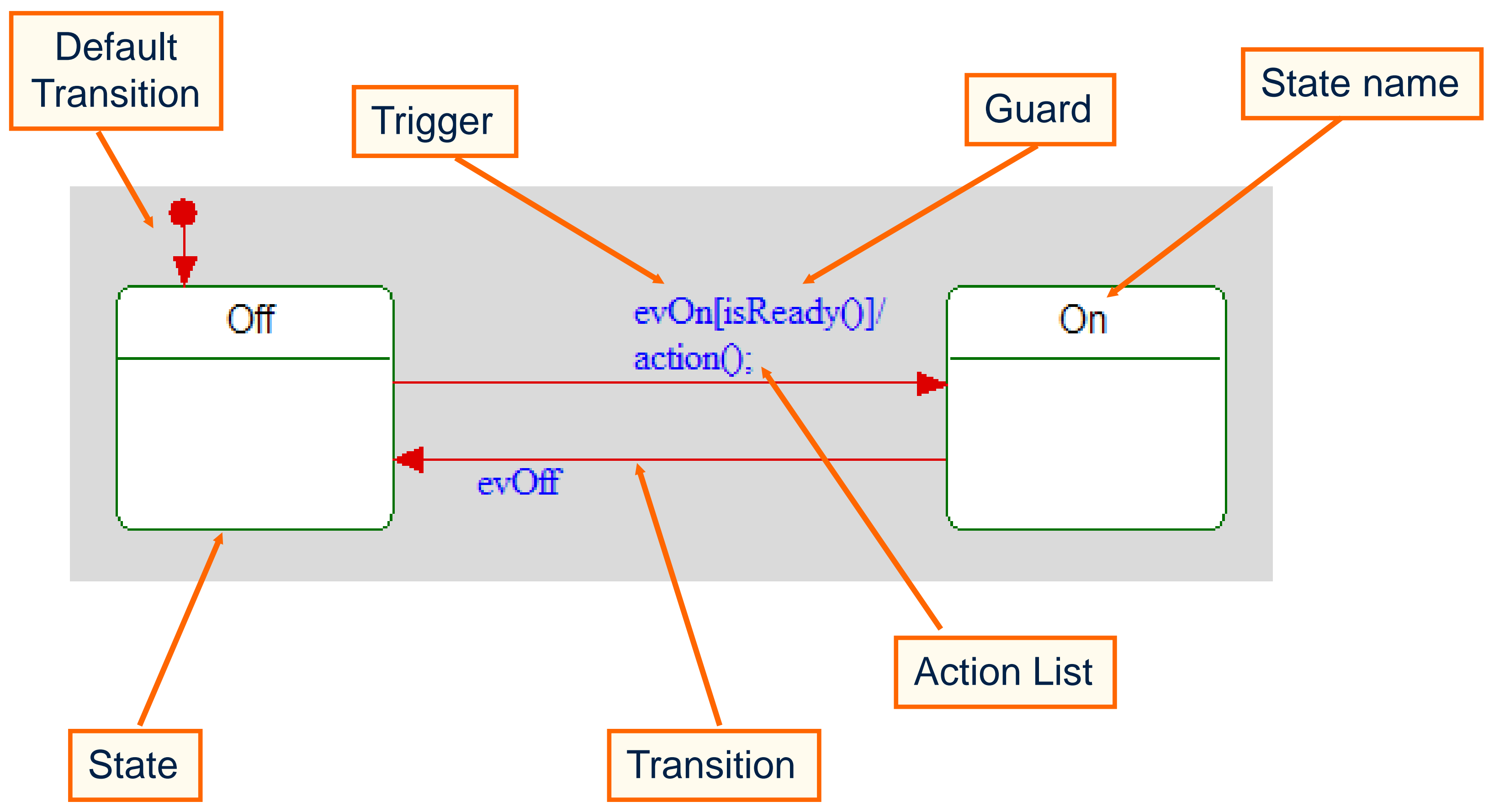
- What is an action?

An **action** is a run-to-completion behavior. The object does not accept or process any new events until the actions associated with the current event are complete.

IBM®

Basic state machine syntax

- Transition syntax
  - trigger [guard] / action list



IBM

Entry / exit actions

Entry actions

GREEN
greenOn();
greenOff();

evRed[isAllowed()]/
log("Green to Red");

RED
redOn();
redOff();

evGreen/
log("Red to Green");

Exit actions

ENV
:LED
Create()
greenOn()
When the LED is created, it starts in the GREEN state
GREEN
evRed()
isAllowed()
0 =isAllowed()
When evRed occurs and the guard is false, then the event is not taken
evRed()
isAllowed()
1 =isAllowed()
When evRed occurs and the guard is true, the event is taken, it executes the exit action, then the transition action and finally the entry action
greenOff()
log(aMsg = Green to Red)
In the RED state, evRed is discarded.
redOn()
RED
evRed()
evGreen()
redOff()
In the RED state, evGreen transitions back to the GREEN state
log(aMsg = Red to Green)
greenOn()
GREEN

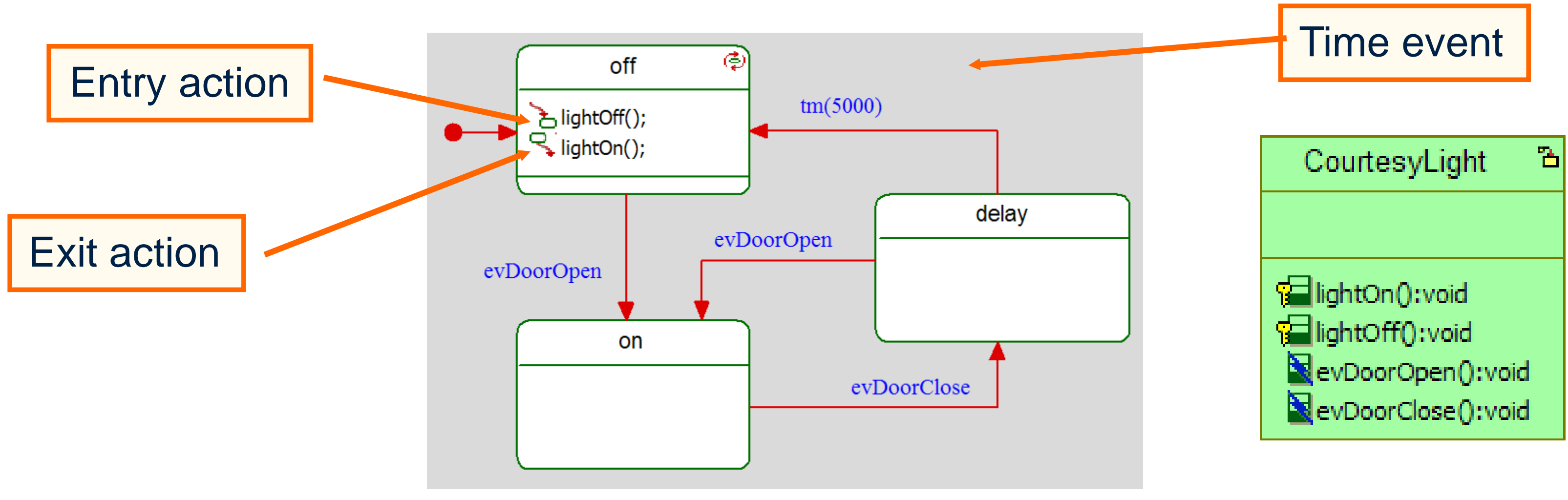
Note the order of execution of the actions and that the guard gets checked before any actions are taken.

Types of events

- UML defines 4 kinds of events:
  - Signal Event
    - Asynchronous signal received for example, evOn, evOff
  - Call Event
    - Operation call received, for example, opCall(a,b,c)
    - This is known as a *Triggered Operation* in Rational Rhapsody
  - Change Event
    - Change in value occurred
  - Time Event
    - Absolute time arrived
    - Relative time elapsed, for example, tm(PulseWidthTime)

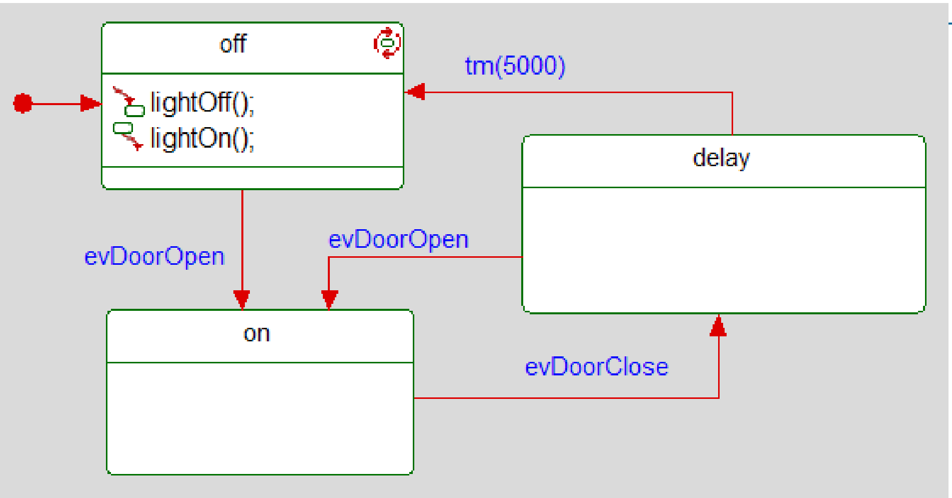
Time event

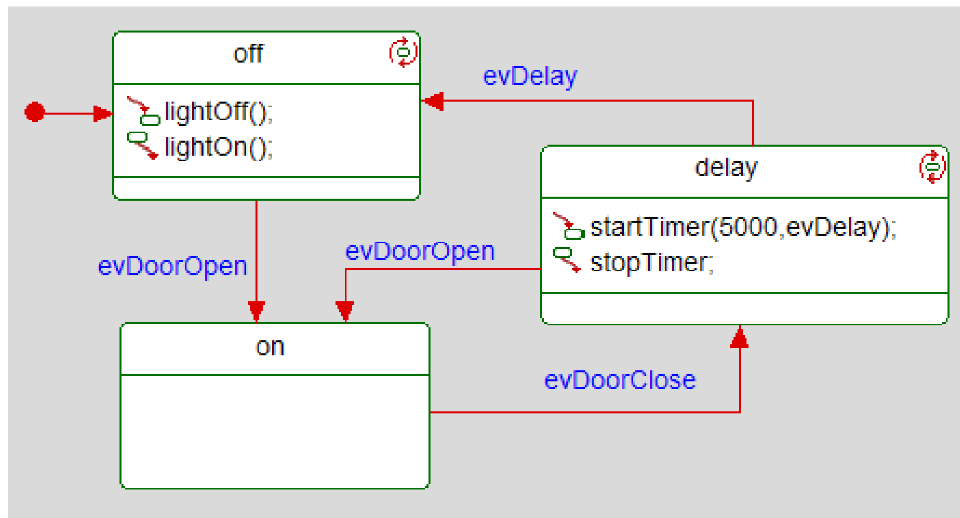
- When an object enters a state, any Timeout from that state is started. When the Timeout expires, the state machine receives the expiration as an event.
- When an object leaves a state, any timeout that was started on entry to that state is cancelled.
- Only one timeout can be used per state; nested states can be used if several timeouts are needed.



tm(delayTime)

- tm(delayTime) is specific to Rational Rhapsody and code is automatically generated to start and stop the timeout.
- This is equivalent to the second state-chart where a timer is started on entering the state and stopped on exiting the state. If the timer expires, then it would send the requested event, for example, evDelay.





UML actually defines the keyword after(Delay) instead of tm(Delay).

Handling transitions

- If an object is in a state S that responds to a named event evX, then it acts upon that event.
- It transitions to the specified state, if the event triggers a named transition and the guard (if any) on that transition evaluates to TRUE. It executes any actions associated with that transition.
- It handles the event without changing state if the event triggers a named reaction and execute all the list of actions associated with that reaction.

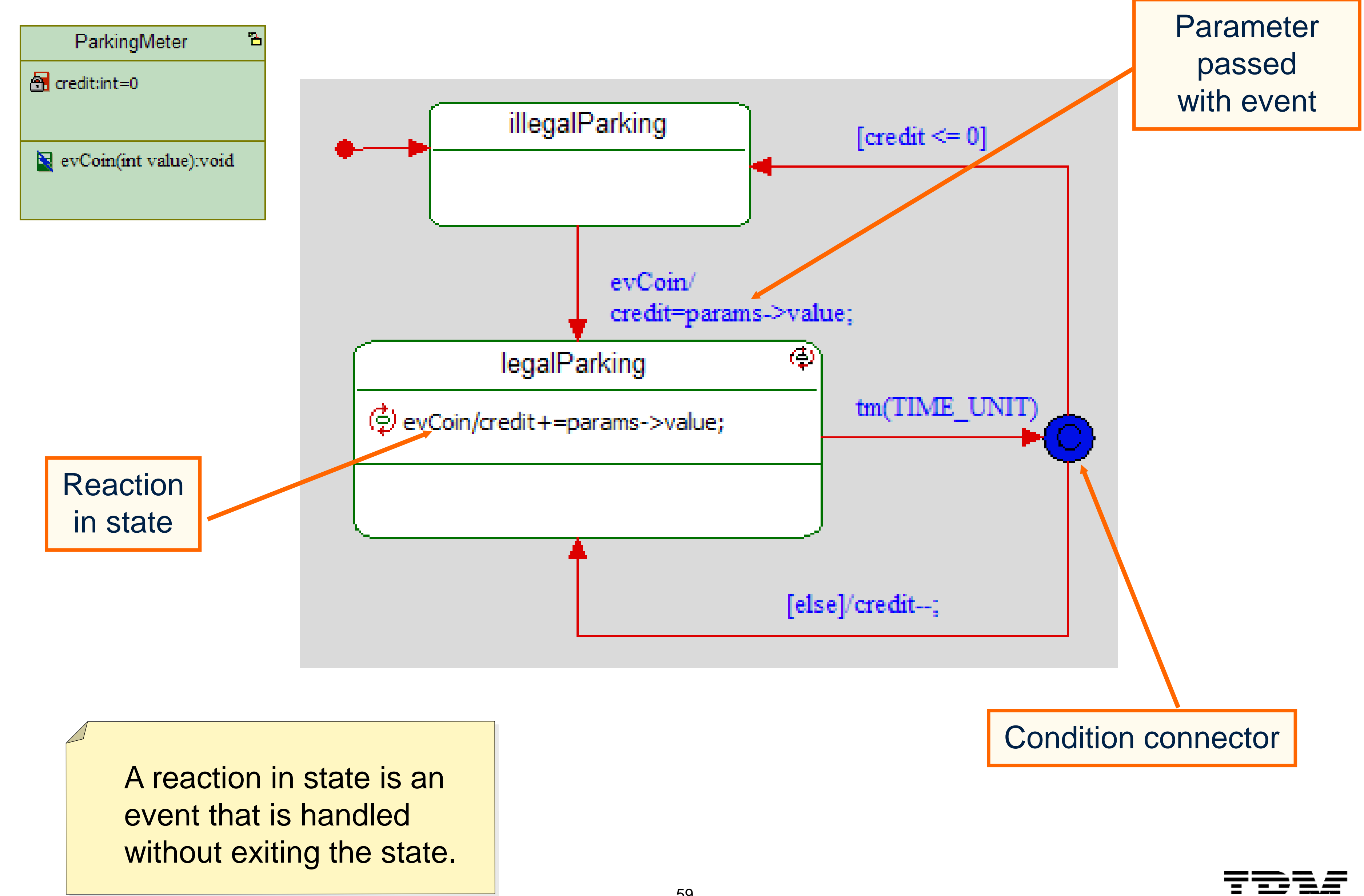
Handling transitions

- Events are quietly discarded if:
  - A transition is triggered, but the transition’s guard evaluates to FALSE.
  - A transition to a conditional pseudostate is triggered, but all exiting transition guards evaluate to FALSE.
  - The event does not explicitly trigger a transition or reaction.

- If an event cannot be handled, then UML allows an option where the event can be deferred until a suitable time when it can be handled.

> Rational Rhapsody does not yet handle deferred events.

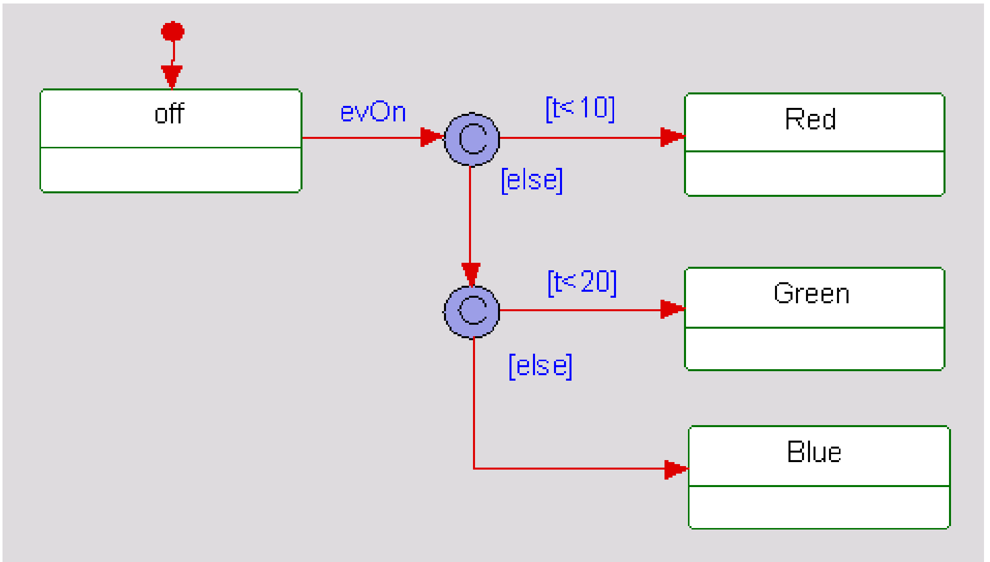
Reaction in state



Transitions: Guards

- A guard is some condition that must be met for the transition to be taken.
- Guards are evaluated *prior* to the execution of any action.
- Guards can be:
  - Variable range specification, for example: [cost<50]
  - Concurrent state machine is in some state [IS_IN(fault)]

If two guards are likely to be true at the same time, then it is a good idea to chain condition connectors, so that the order in which the guards are tested is known.



Actions

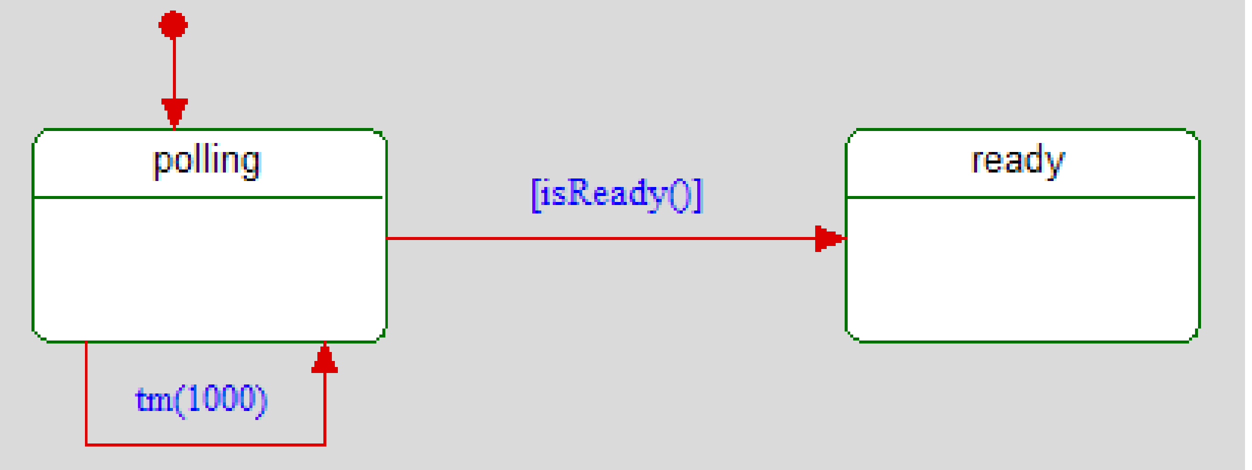
- Actions are run to completion:
  - Normally actions take an insignificant amount of time to perform
  - They may be interrupted by another thread execution, but that object will complete its action list before doing anything else
- Actions are implemented via:
  - An object's operations
  - Externally available functions
- They may occur when:
  - A transition is taken
  - A *reaction* occurs
  - A state is entered
  - A state is exited

Do not use actions that block.
For example reading a socket.

Null-triggered transitions

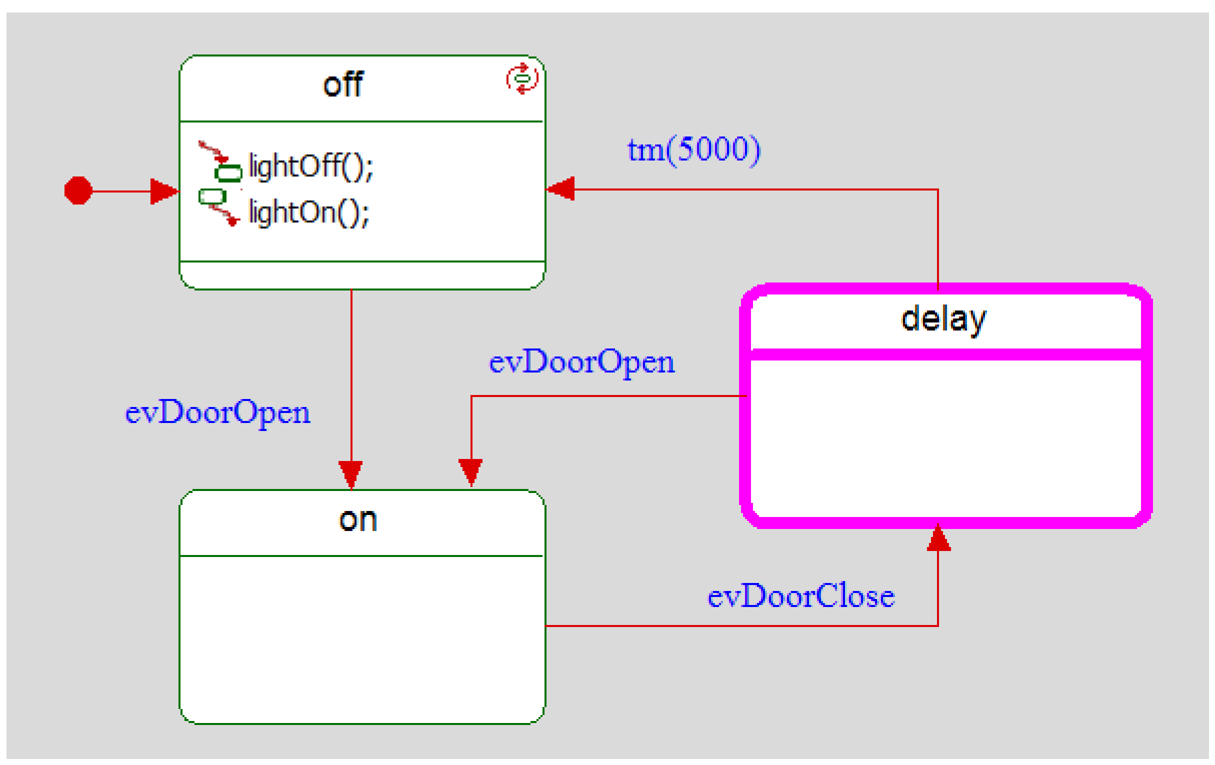
- Also known as *Completion Transitions.*
- Triggered upon completion of any entry actions.
- May contain a guard condition.
- Will only be evaluated once, even if guard condition later becomes true.

On entering the polling state, since there is no trigger, the guard is immediately tested. If it is true, the object moves to the ready state. The guard is retested every time that the timeout occurs.



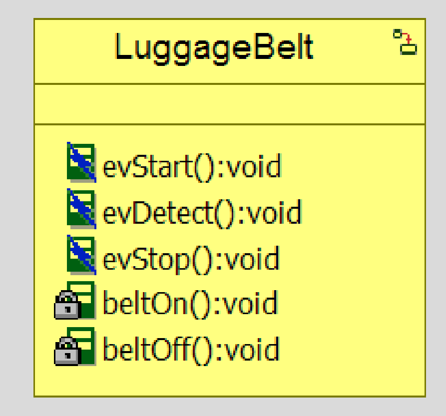
State machine syntax – OR states

- An object must always be in *exactly one* OR-state at a given level of abstraction.
  - The object must be in either *off* or *on* or *delay* – it cannot be in more than one or none.



Exercise 2: luggage belt system

- Draw the state machine for a luggage belt system. The belt is started when the start button is pressed and runs until either the stop button is pressed or until there is no luggage on the belt. This condition is when no luggage has been detected in the previous 60 seconds.



How do you describe behavior Part 2?

Count Down

State machine syntax – nested states

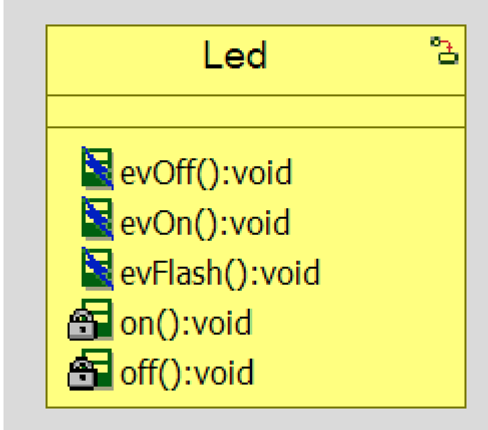
armed
itsLed[RED].off();
evTemporise
exiting
itsLed[RED].flash(100);
tm(EXIT_TIME)
active
itsLed[RED].on();
/init();
evDoor
off
entering
itsLed[RED].flash(100);
tm(SILENCE_TIME)
evMovement
evMovement
silence
evDisarm
intrusion
OUT_PORT(cp)->setSiren(ON);
OUT_PORT(cp)->setSiren(OFF);
tm(ENTRY_TIME)
tm(ALARM_TIME)

Nested state

If the event *evDisarm* is received when the object is in state armed, then irrespective of which nested state is active, the transition will be taken and the object will go into the off state.

Exercise 3: LED

- Draw the state machine for an LED class that can be in one of three modes: on, off and flashing at 1Hz.

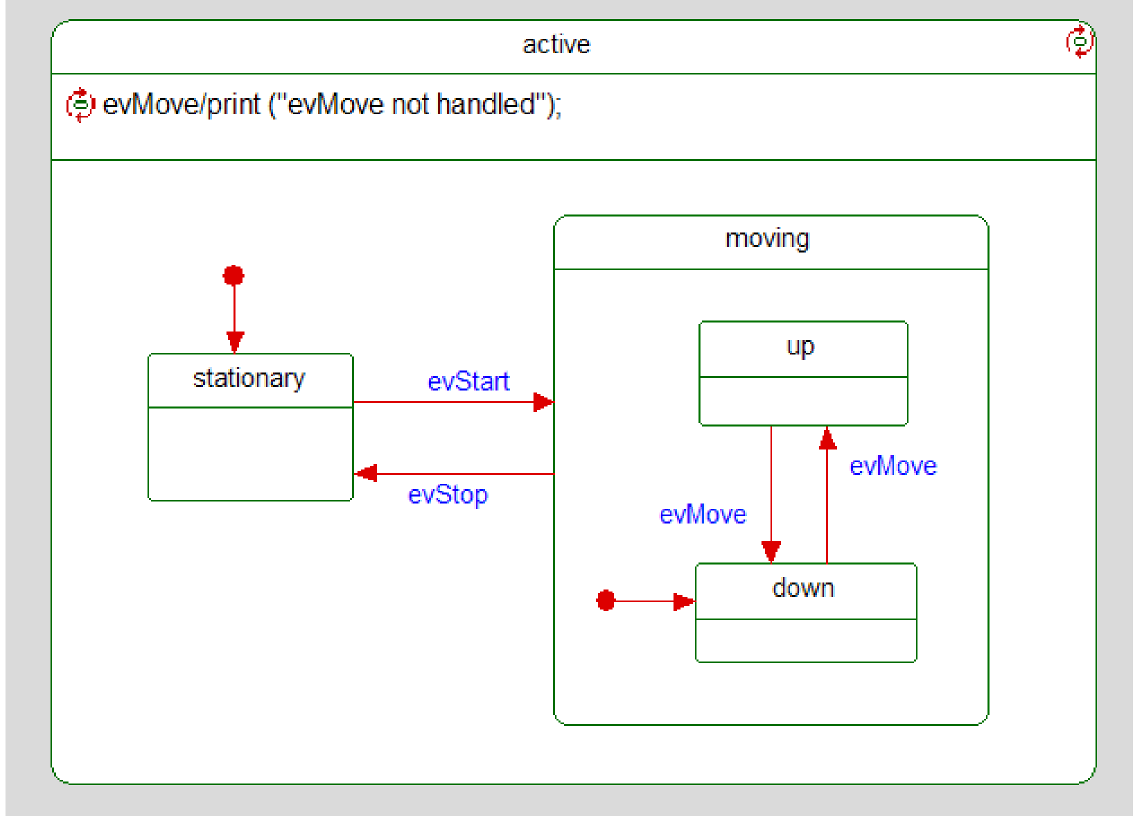


IBM

Nested states: hierarchical events

- When an event is received, it is processed in order, starting with the inner most state until the event is processed. For example, if the event *evMove* is received then:

If the active state is *up,* then the object goes to the *down* state.
If the active state is *down*, then the object goes to the *up* state.
If however, the active state is *stationary*, then the object prints out *evMove not handled*.



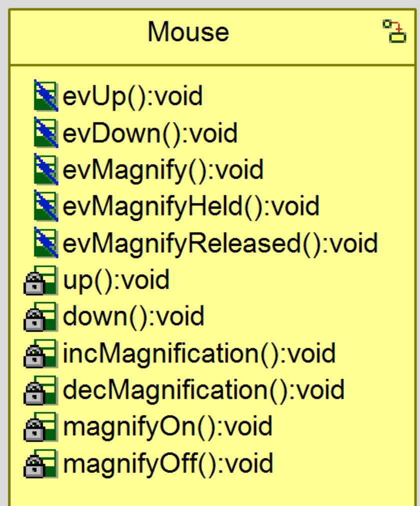
UML pseudostates

Symbol	Symbol Name	Symbol	Symbol Name
	Initial or Default Pseudostate	H	(Shallow) History Pseudostate
C	Branch Pseudostate (type of junction pseudostate)	H*	(Deep) History Pseudostate
	Junction Pseudostate	[g] [g]	Choice Pseudostate
	Fork Pseudostate		Join Pseudostate
	EntryPoint Pseudostate		ExitPoint Pseudostate
T or	Terminal or Final Pseudostate		

You cannot draw a *choice Pseudostate* in Rational Rhapsody, but the same effect can be achieved by replacing it with a state.

Exercise 4: mouse

- Draw the state machine for the following mouse that has three extra buttons:
  - One of these buttons allows the Mouse to magnify the area around the mouse. This *magnify* mode is invoked and exited by pressing the magnify button (*evMagnify*).
  - When in the *magnify* mode, if the magnify button is held (*evMagnifyHeld*), then the up (*evUp*) and down (*evDown*) buttons control the magnification, invoking operations *incMagnification()* and *decMagnification().* It remains in this mode until the magnify button is released (*evMagnifyReleased*).
  - When the magnify button is not held, the up (*evUp*) and down (*evDown*) buttons invoke operations *up()* and *down().*

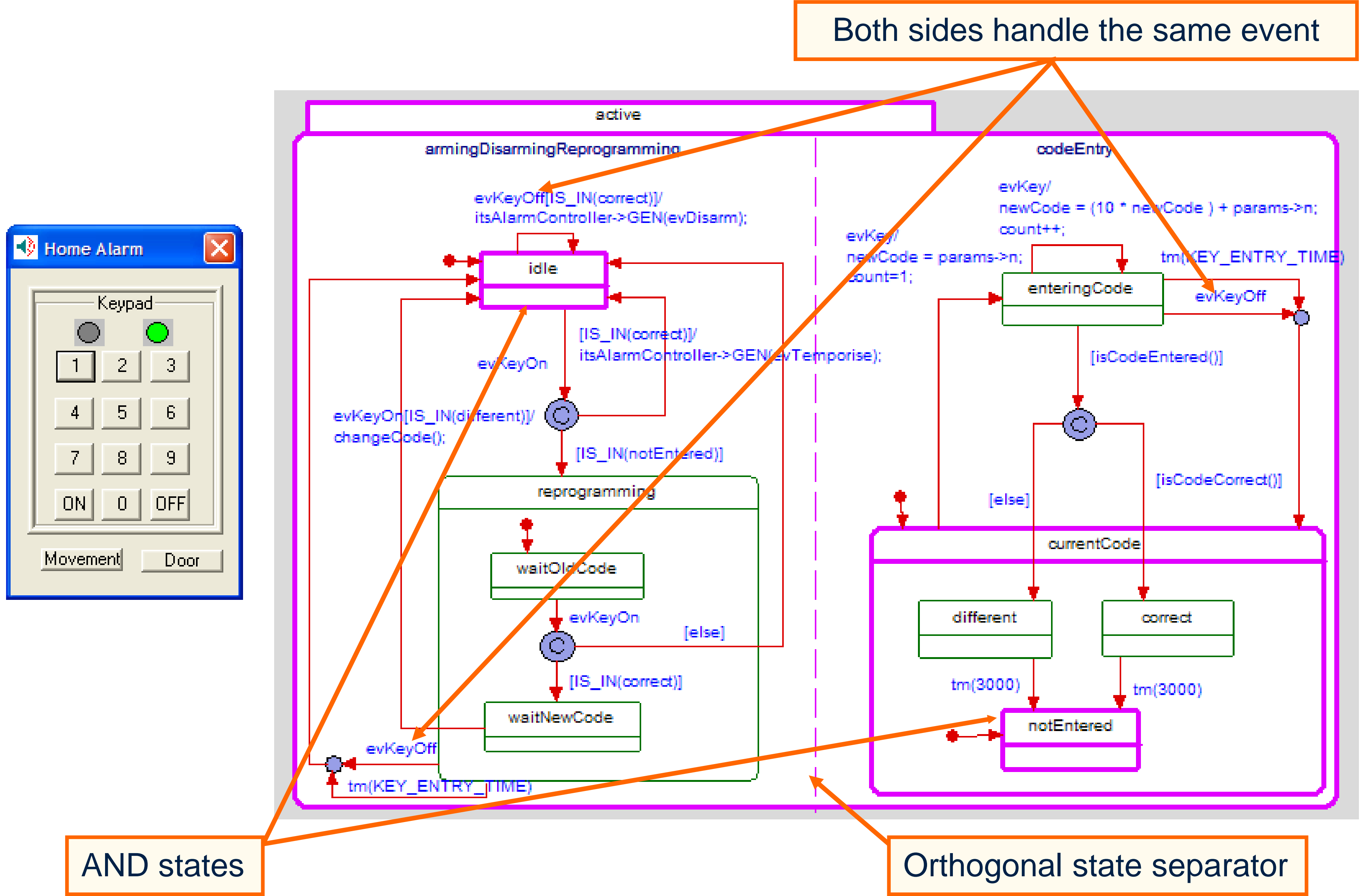


IBM

AND-states

- When a state has multiple AND-states, the object must be in exactly one substate of each active AND-State at the same time.
- AND-states are logically concurrent.
  - All active AND-states receive their own copy of any event the object receives and independently act on it or discard it.
  - Cannot tell *in principle* which and-state will execute the same event first.
  - Not necessarily concurrent in the thread or task sense.
  - UML uses active objects as the primary means of modeling concurrency.
  - AND-states may be implemented as concurrent threads, but that is not the only correct implementation strategy.

State machine syntax AND states



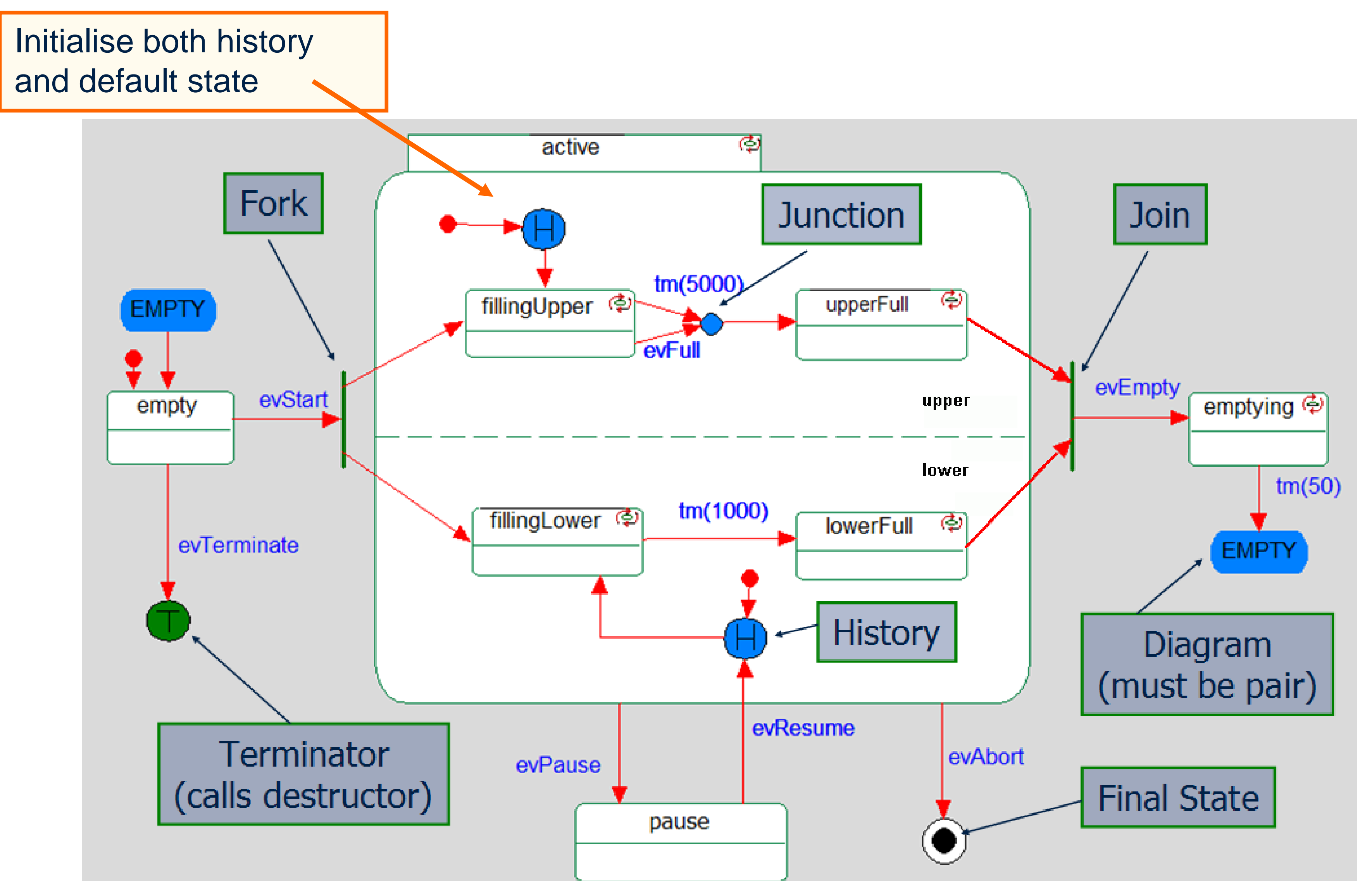
IBM

AND-state communication

- AND-states may communicate via:
  - Broadcast events
    - All active AND-states receive their own copy of each received event and are free to act on it or discard it.
  - Propagated events
    - A transition in one AND-state can send an event that affects another.
  - Guards
    - [IS_IN(state)] uses the substate of an AND-state in a guard.
  - Attributes
    - Since the AND-states are of the same object, they “see” all the attributes of the object.

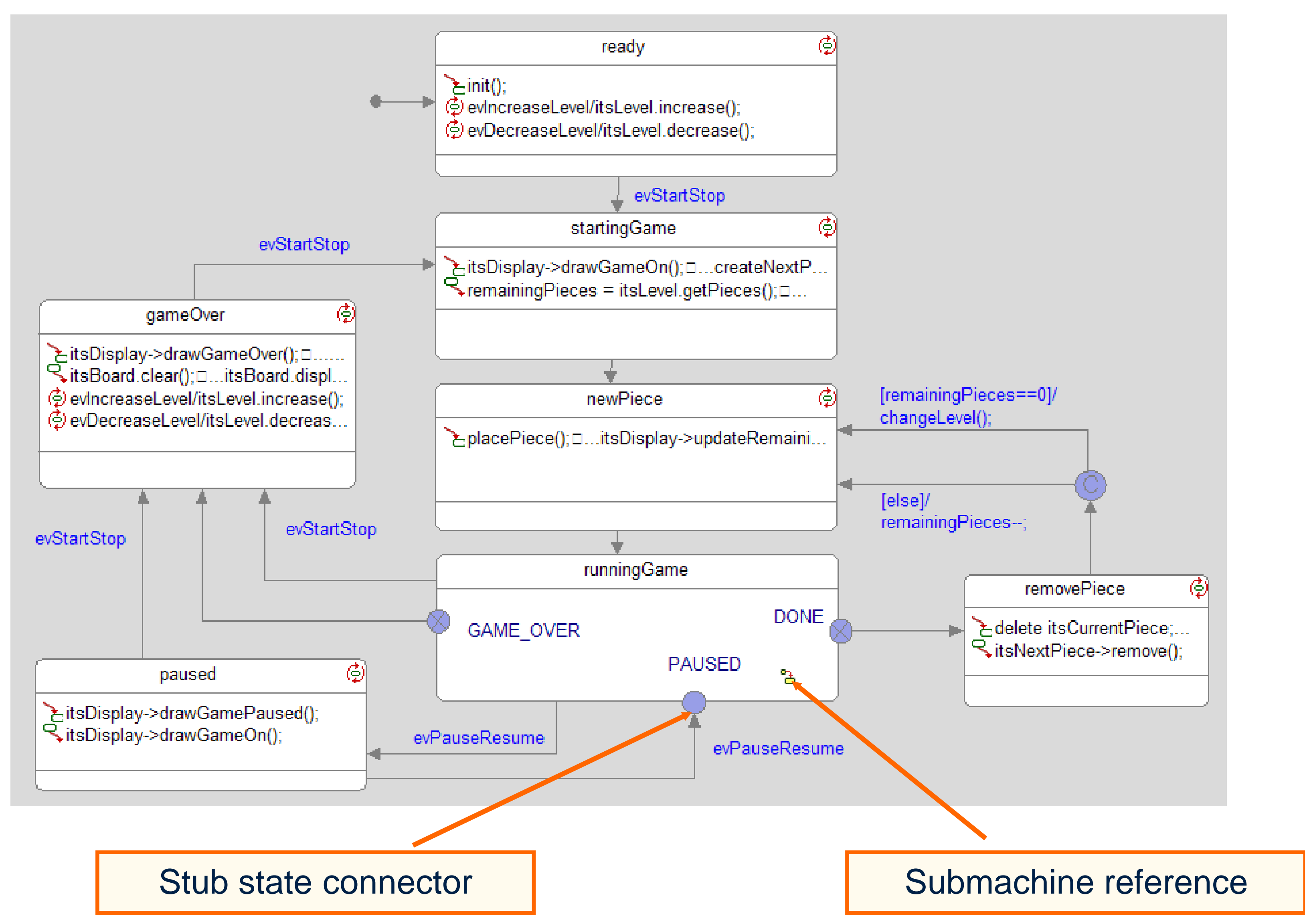
IS_IN is a Rational Rhapsody C++ macro that can be used to test to see if an object is in a particular state.

State machine syntax – connectors



IBM

Submachines: parent



IBM

Submachines: child

Submachine

runningGame

itsCurrentPiece->show();

startup

itsCurrentPiece->drop();

tm(itsLevel.getDropTime())

[else]

GAME_OVER

[isDropPossible()]

controlling

itsCurrentPiece->drop();
evLeft/itsCurrentPiece->moveLeft();
evRight/itsCurrentPiece->moveRight();
evRotate/itsCurrentPiece->rotate();

[isDropPossible()]

evDrop

dropping

itsCurrentPiece->drop();

[isDropPossible()]

tm(itsLevel.getDropTime())

[isDropPossible()]

tm(10)

[else]

[else]

dropped

itsCurrentPiece->drop();□...itsCurrentPiece->show();
SCORING_EVENT scoringEvent = itsBoard.check();□...if (sco...

[else]

DONE

PAUSED

Stub state connector

Timeouts revisited

- What is the output of the following state machine?

MeterReader

Electricity

idle

tm(1000)

reading

readElectricity();

tm(1000)

Gas

idle

tm(5000)

reading

readGas();

tm(1000)

tm(4000)/
logReadings();

Poorly formed state machine

Race condition

Must be same event

History not initialized

No default state

Conflicting transitions

Overlapping guards

Use before initialization

StatechartOfBadDesign

state_A

state_A1a

ev1/++x;

state_A2a

ev2/x=0;

state_A1b

ev1/x*=3;

state_A2b

ev4

ev5

state_E

state_E1

ev6

ev7

state_E2

H

[amount<=0]

ev2

state_B

evCoin/
amount=params->aCoin;

[x>5]

state_D

C

ev3

C

[x>0]

state_C

[amount>0]

Inherited state behavior 1

- Two approaches to inheritance for generalization of reactive classes:
  - Reuse (for example, inherit) state machines of parent
  - Use custom state machines for each subclass

- Reuse of state machines allows:
  - Specialization of existing behaviors
  - Addition of new states and transitions
  - Makes automatic code generation of reactive classes efficient in the presence of class generalization

Inherited state behavior 2

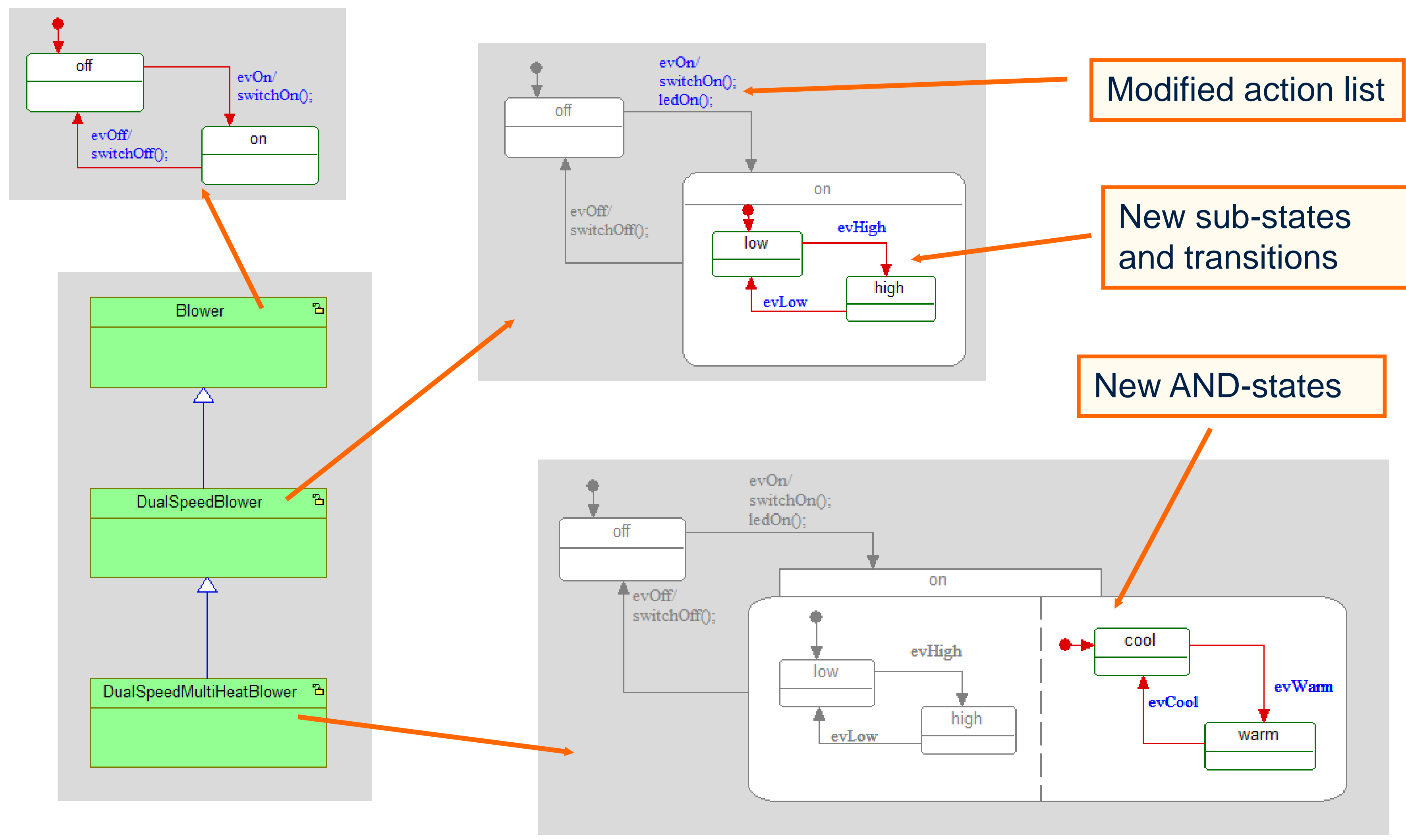
- Subclasses may be:
  - Specialized:
    - Sub-states may be added
    - Transitions may be rerouted
    - Action lists may be modified
  - Extended:
    - New states added
    - New transitions added
    - New action lists added

Inherited state behavior 3

A subclass must be freely substitutable for the super-class in any operation.

- Assumes Liskov substitution principle for generalization:
  - You can:
    - Add new states
    - Elaborate sub-states in inherited states
    - Add new transitions and actions
  - You cannot:
    - Delete inherited transitions or states

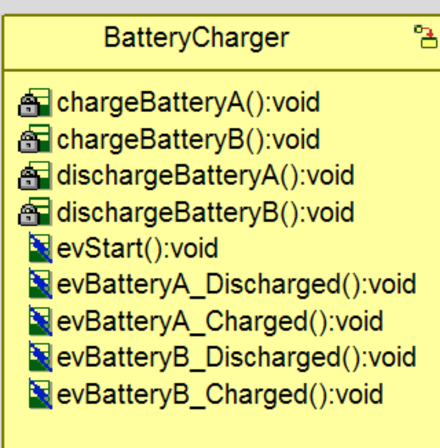
Example: Generalization



IBM

Exercise 5: Battery charger

- Draw the state machine for a simple Battery Charger that can charge two batteries in parallel. The charger has three modes: idle, discharging, and charging.
- A button can be pressed (evStart) to start charging the batteries. However, before each battery can be charged, it must be discharged.
- When each battery is discharged, it sends an event (evBatteryA_Discharged or evBatteryB_Discharged) to the Battery Charger.
- When each battery is charged, it sends an event (evBatteryA_Full or evBatteryB_Full) to the Battery Charger.
- When both batteries are charged, the Battery Charger returns to the idle mode.



How do you model communication using UML?

Dishwasher

Object collaboration

- Objects work together to form cohesive assemblies called *collaborations.*
- Collaborations work by the fulfillment of system-level behaviors called *use cases* via their behaviors and operations.

Messages

- Objects communicate between each other via messages
- Messages may be:
  - Synchronous for example, Function call
  - Asynchronous for example, Posting OS message

Relationships

- In order to send a message to another object, there must be some kind of relationship between the objects:
  - Objects may use the facilities of other objects with an *association.*
  - Objects may contain other objects with an **aggregation*.***
  - Objects may *strongly* aggregate others via *composition.*
  - Classes may derive attributes and behaviors from other classes with a *generalization.*
  - Classes may depend on others via a *dependency.*
  - Association, Aggregation, and Composition allow objects to communicate at run-time.

Generalization basically implies *Is a Kind of*.

Associations 1

- Allow instances of classes to communicate at run-time:
  - Instances of associations are called *links*.
  - Links may come and go during execution.
- Denote one object using the facilities of another.
- Lifecycles of the objects are independent.
- Allow objects to provide services to many others.

Associations 2

- Associations may have labels:
  - This is the *name* of the association.
- Associations may have role names:
  - Identifies the role (or responsibility) of the object in the association.
- Associations may indicate multiplicity:
  - Identifies the number of instances of the class that participate in the association.
- Associations may indicate *navigation* with an open arrowhead:
  - Unadorned associations are assumed to be symmetric
  - Most associations are unidirectional.

Multiplicity

- Denotes the number of instances of the related classes that participate in the association.
- Options:
  - N — fixed number, for example, 1, MAX_ELEVATORS
  - * — Zero or more
  - 1..* — One or more
  - X..Y — range, for example, "0..1" or "5..7"

Most associations have multiplicity 1.

Associations - 3

Relation Name

Multiplicity

Pack

always has

1..2

itsPack

itsCard

52

Card

CardGame

uses

has

Hand

*

itsCard

0..13

is type of

itsHand

has

Role name

Cribbage

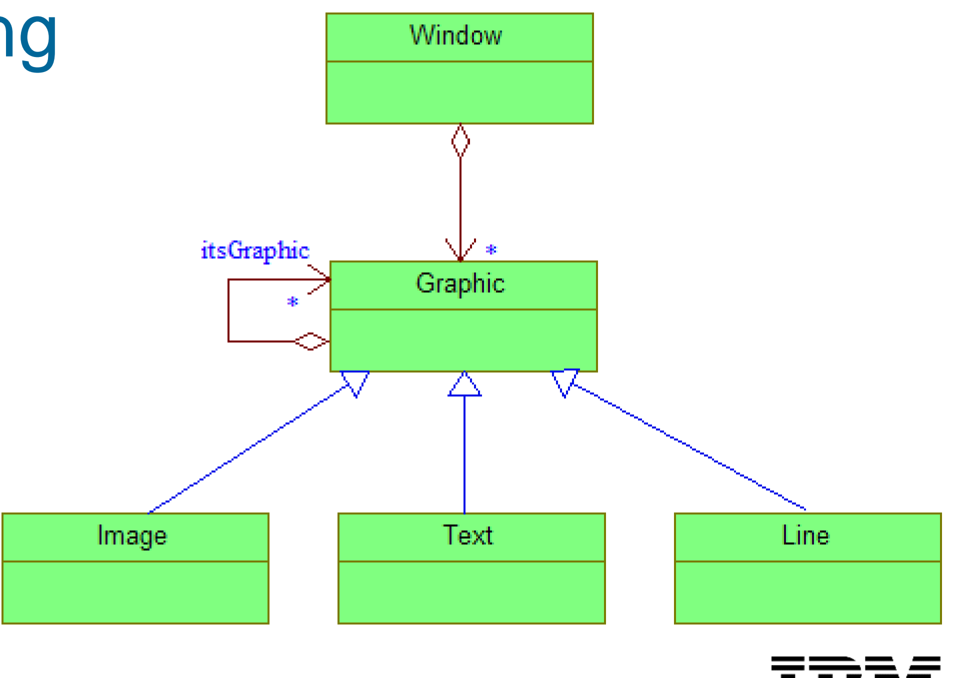
Whist

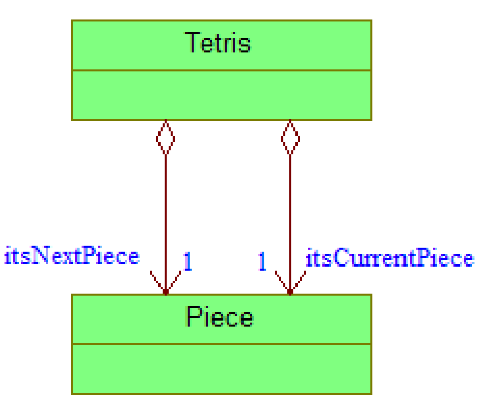
BlackJack

Navigation

Aggregation

- Indicated by a hollow diamond
- *Whole-part* relationship denotes one object logically or physically contains or *has* another
- *Weaker* form of aggregation. Nothing is implied about:
  - Navigation
  - Ownership
  - Lifetimes of participating objects

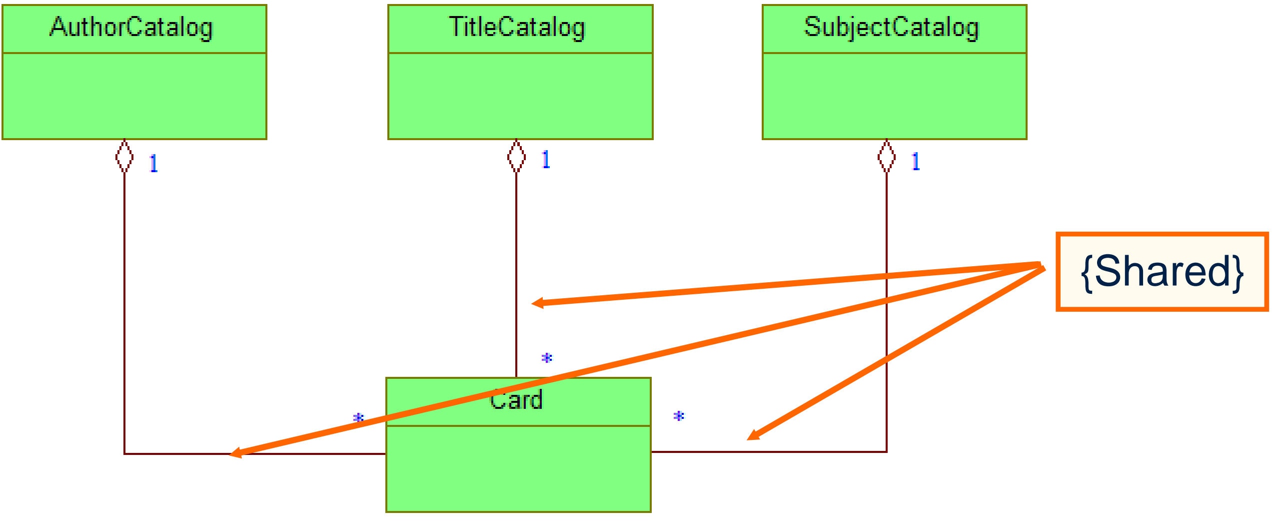




IBM

Aggregation

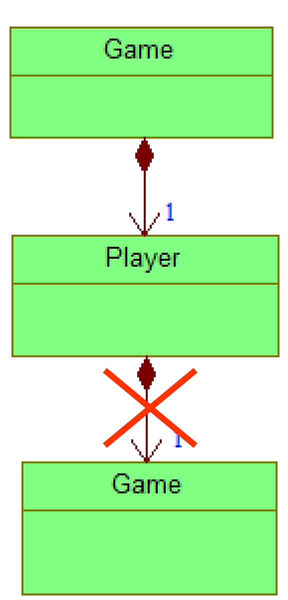
- Multiplicity of owners > 1 indicates a *shared part.*
- Forms a graph with its parts.

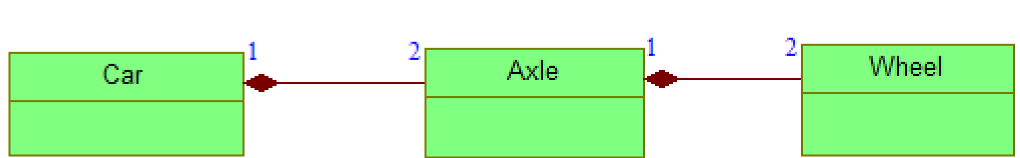


By forming a graph, it mean that a class can aggregate itself (although it must be a different instance) but in composition you can't (compositions can only form trees).

Composition

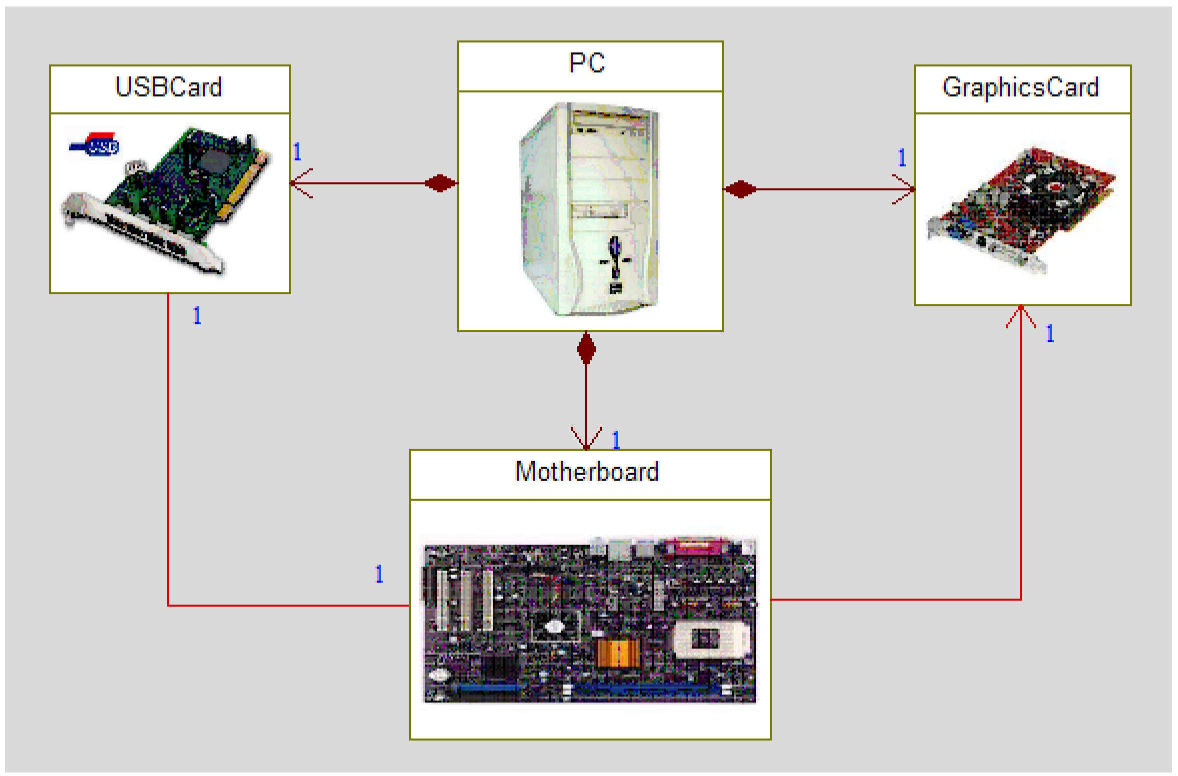
- Indicated by containment or a filled diamond.
- Whole both creates and destroys part objects.
- Composite is a higher level abstraction than the parts:
  - Allows the class model to be viewed and manipulated at many levels of abstraction
- Stronger form of aggregation:
  - Implies a multiplicity of no more than one with the whole
- Forms a tree with its parts.





Composition example

- This diagram shows that the PC creates instances of the other three classes, but it is not clear if the relations between the instance of the Motherboard to the GraphicsCard and USBCard get initialized.



Structured class example

- With a structured class, you can clearly see that the PC is composed of instances of USBCard, Motherboard, and GraphicsCard and that the relations between these instances are initialized.

Part

PC

1 UC:USBCard

1 MB:Motherboard

1 GC:GraphicsCard

Connector / Link

Qualified association

- Used with a relation of multiplicity *.
- Allows the relation to be sorted according to the value of an attribute.

Airline
numberOfMembers:int
getMilesFlown(anId:int):int
frequentFlyerNumber
*
Member
name:char*
address:char*
frequentFlyerNumber:int
milesFlown:int
getMiles():int

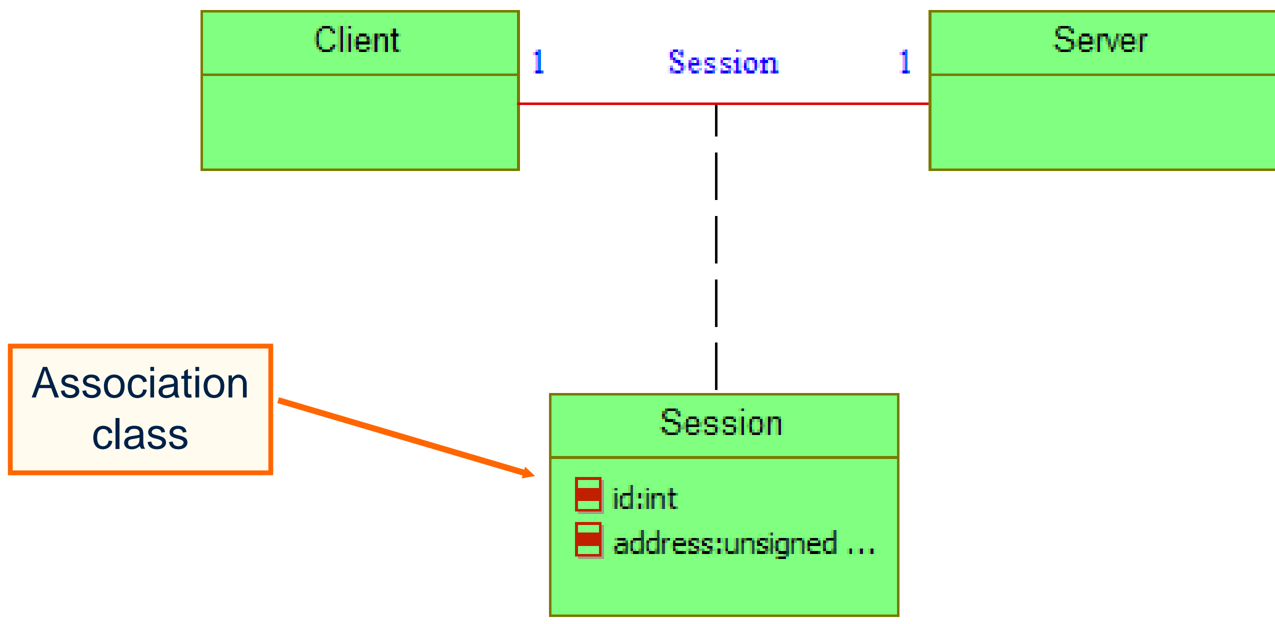
Qualifier

A Qualified association could be implemented as a balanced tree.

AuthorCatalog
author
NameCatalog
name
ISBNCatalog
ISBN
*
*
*
Book
ISBN:int
name:char*
author:char*

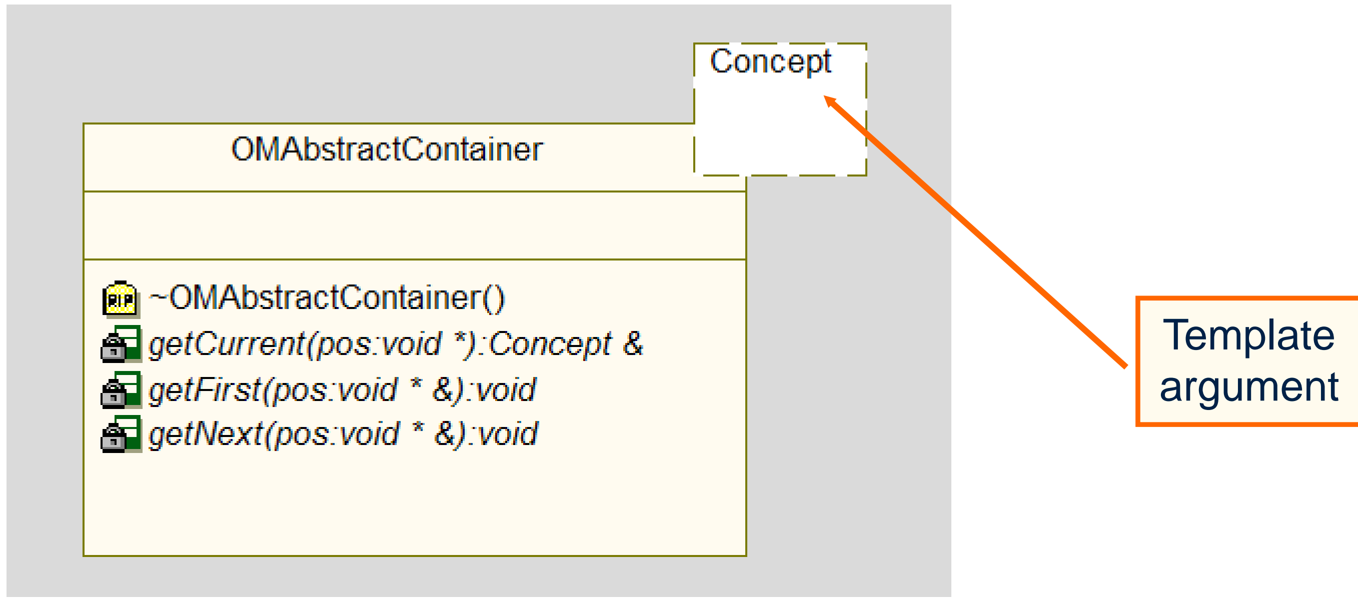
Association class

- An association class is used when information does not seem to belong to either object in the association or it belongs equally to both.



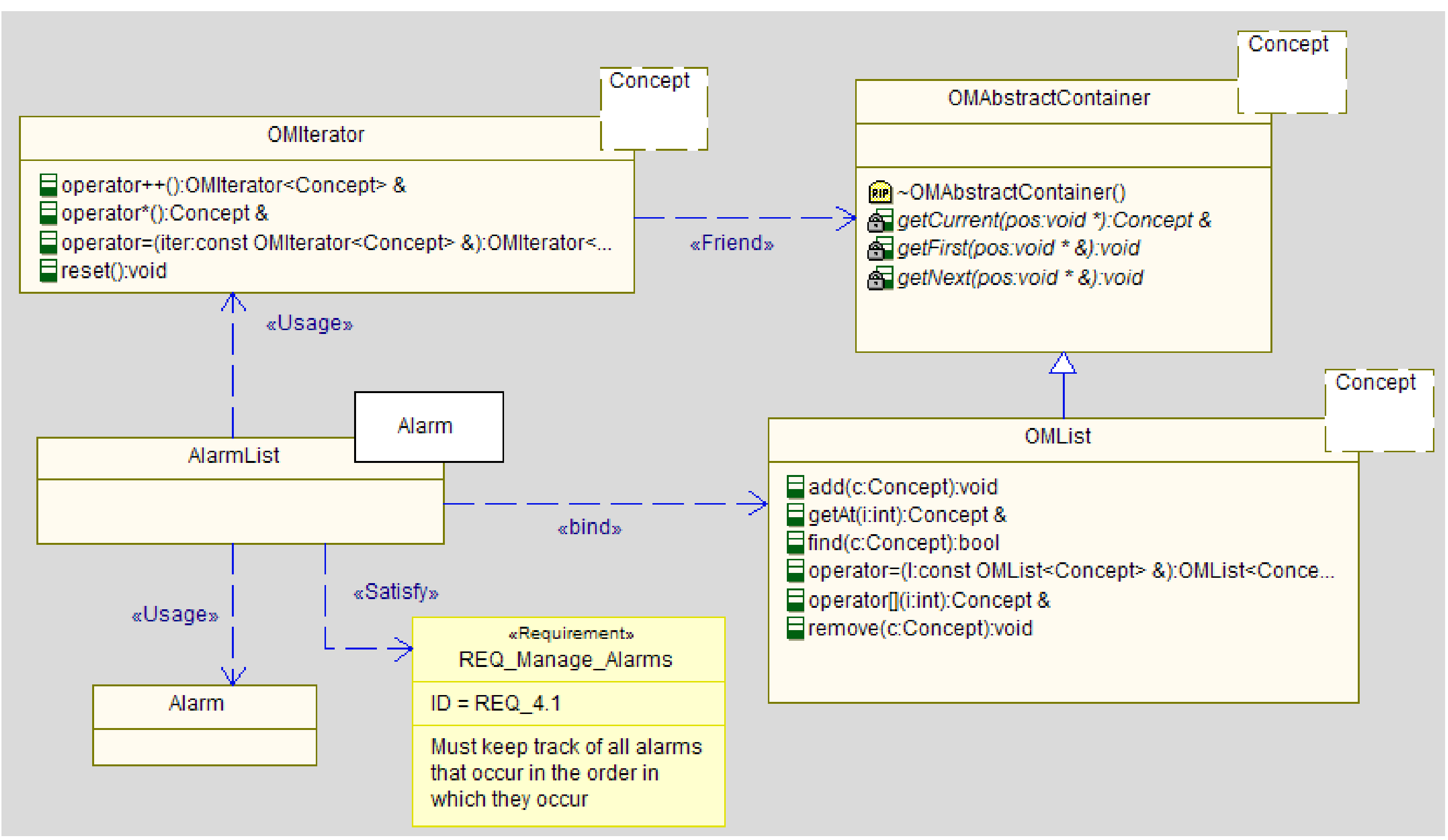
Template / generic classes

- Some languages such as C++, Ada, C#, and Java allow the use of template or generic classes. The UML allows template / generic classes to be drawn.



Dependencies

- A dependency can be used when a class has no direct relation to another class, but uses it in some way.
- The «stereotype» details how the item is dependent on the other.



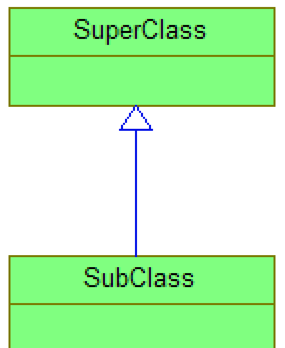
IBM

Generalization 1

- Generalization means two things in the UML:
  - **Inheritance**
    - The child acquires things from the parent(s). In UML, a child class acquires the attributes and operations (including state machines) from the parent class(es).
  - **Substitutability**
    - It is satisfactory to use the substitute instead of the intended item. In UML, a substituted object (instance) performs satisfactorily.

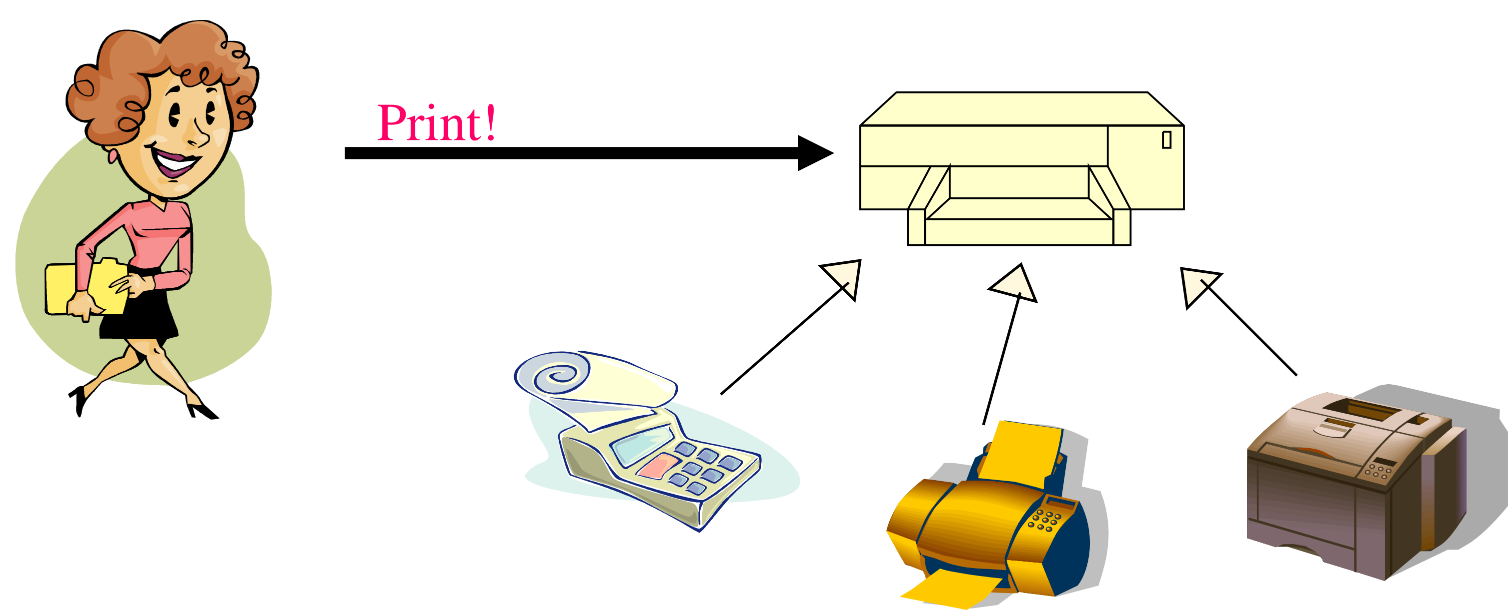
Generalization 2

- Subclass *is-a-type-of* the superclass.
- UML generalization means:
  - Subclass inherits structure and behavior from the superclass:
    - Attributes
    - Operations
    - Relations
    - State Machine or Activity Diagram
  - Instances of subclass are freely substitutable for instances of the superclass.



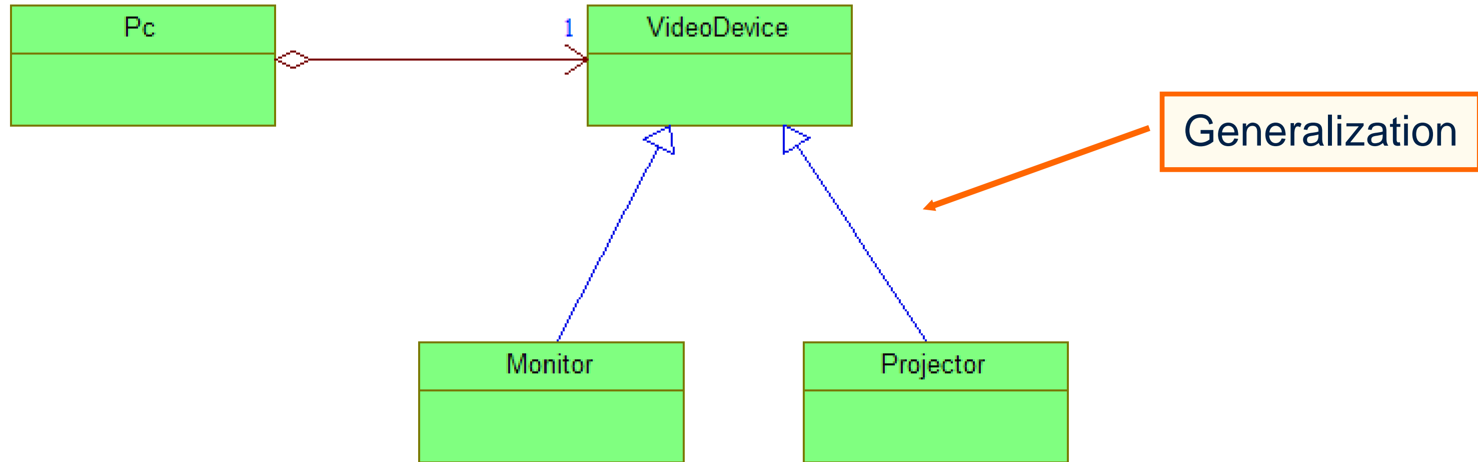
Generalization substitutability

- Liskov substitution principle:
  “*An instance of a subclass must be freely substitutable for an instance of its superclass*”.
- An object with a relation to a general object should be able to use the general object's derived objects without knowing it.



Generalization 3

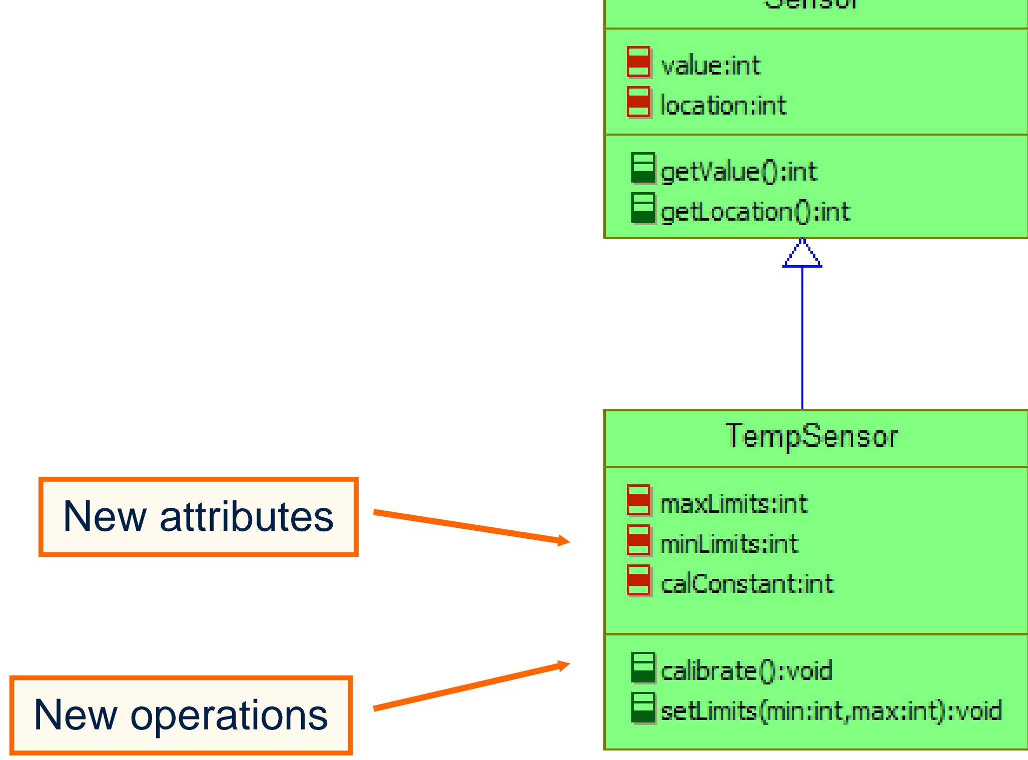
- Generalization is shown with a solid line and closed arrowhead pointing to the more general class.



Generally the *Super Class* is always shown above the *Sub Class*.

Generalization 4

- Subclasses may be ***extended*** by adding:
  - New attributes
  - New operations



Generalization 5

- Subclasses may be ***specialized*** by:
  - Redefining existing operations

Client

1
itsQueue

Queue

head:int
tail:int
size:int

insert(aMsg:char*):void
remove():void
getSize():int

CachedQueue

filename:char*

insert(aMsg:char*):void
remove():void
flush():void

Inherited operation

Virtual operations

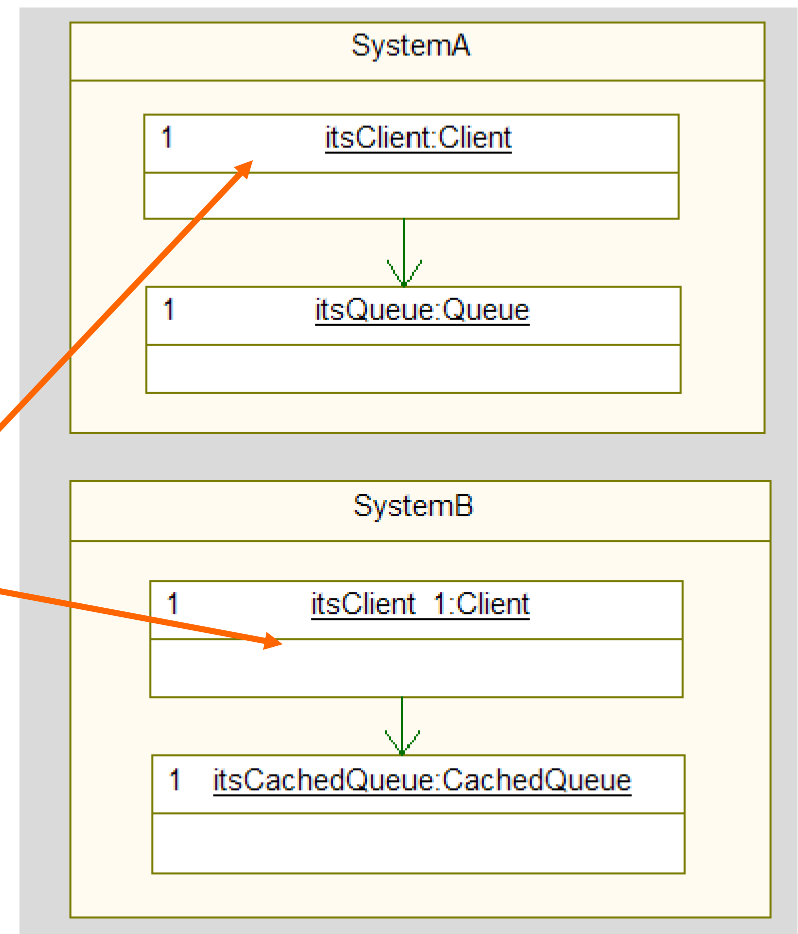
New attribute

Specialized operations

New operation

Queue and CachedQueue

- If the Client is using the *Queue*, then you could replace the *Queue* by the *CachedQueue* without needing to change the Client's code.



itsQueue->insert("abc");

The Client calls the *insert* operation without knowing if it is the Queue or the CachedQueue which is being used.

Good generalization

- These are all different types of buttons, all having different behavior.

Button

LockingButton

MomentaryButton

Turnknob

GroupButton

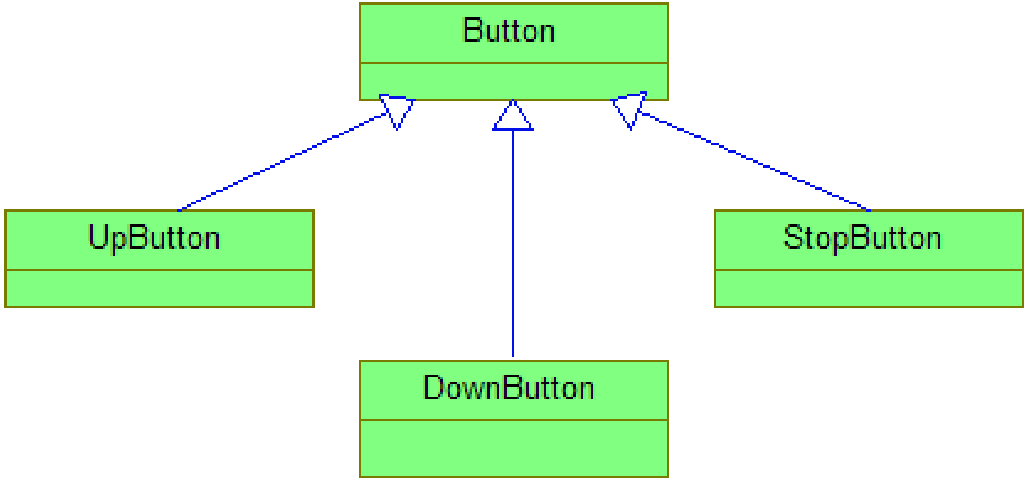
MultipositionButton

PushknobButton

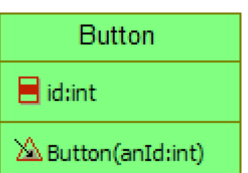
*

Bad generalization

- These are all basically the same class.



- They should all be of the class Button, with perhaps just an attribute indicating the type of button.



Ugly generalization

- Beware of multiple generalization:

Command
id:int
execute():void

Level1Command
level1:int
execute():void

Level2Command
level2:int
execute():void

MultiCommand
multiLevel:int

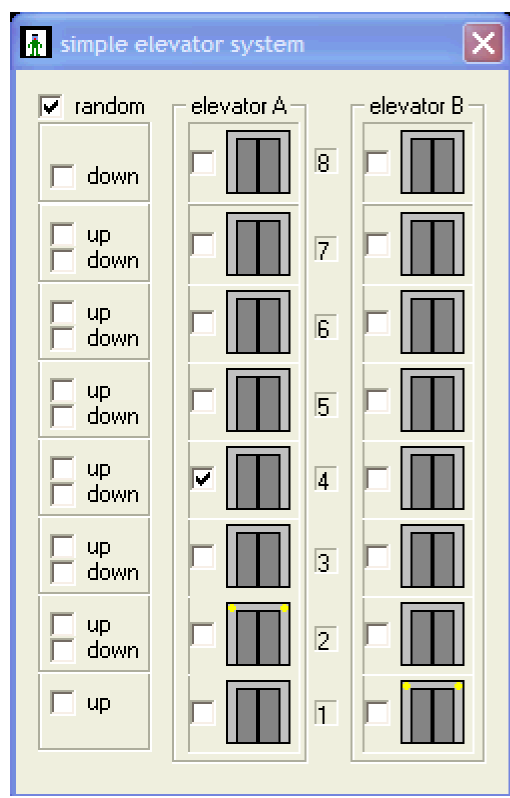
Which execute is run?

Gets two copies of Attribute id.

This is also known as the *Diamond of Death.*

Exercise 6

- An elevator system will have two elevators and eight floors.
- Each floor will have a couple of buttons to call the elevator.
- Each button has an indicator indicating that it has pressed.
- Each elevator will have a button and corresponding indicator for each floor.



1. Draw the class diagram.
2. Draw two different scenarios.

Drawing the scenarios help in verifying that the class diagram is correct.

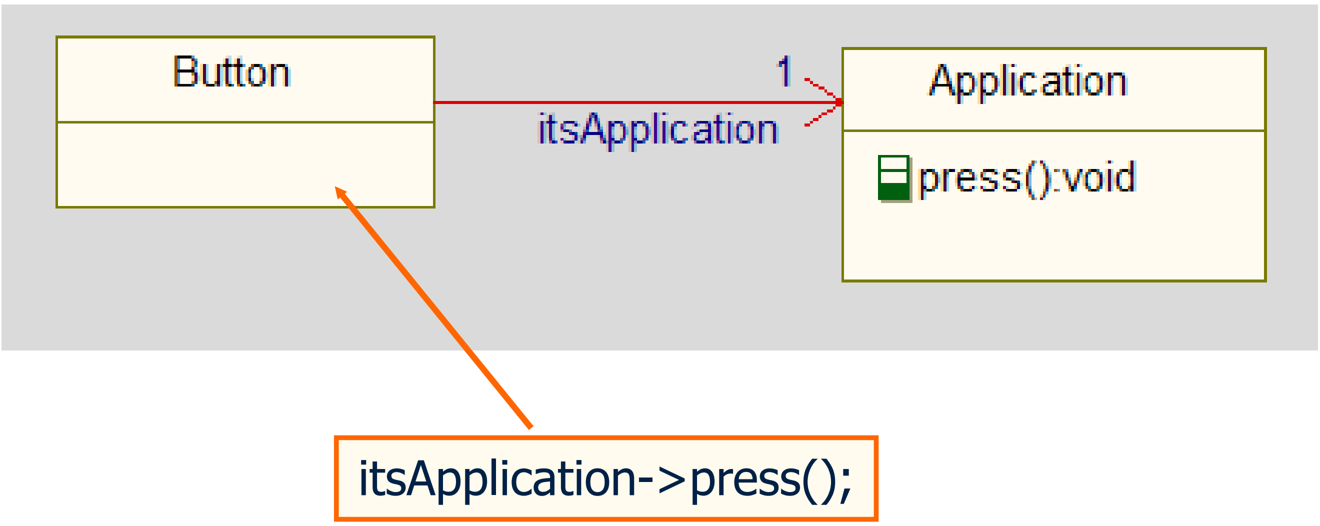
Elevator system: Class diagram

Elevator system: Scenario 1

Elevator system: Scenario 2

Interfaces: why do you need them?

- In the following model, when the button gets pressed, it invokes the press() operation on the application, for example:



- It is difficult to reuse the Button class since it is closely associated or coupled with the Application class.

Interface *IButtonListener*

- You could add a new class to isolate the Button from the Application. This class would be what is called an Interface class and would have just abstract operations having no implementation.
- The application classes would realize the *IButtonListener* class and implement the press() operation.

Button

1

itsIButtonListener

«Interface»

IButtonListener

press():void

Abstract operation

Realization

Elevator

press():void

Floor

press():void

itsIButtonListener->press();

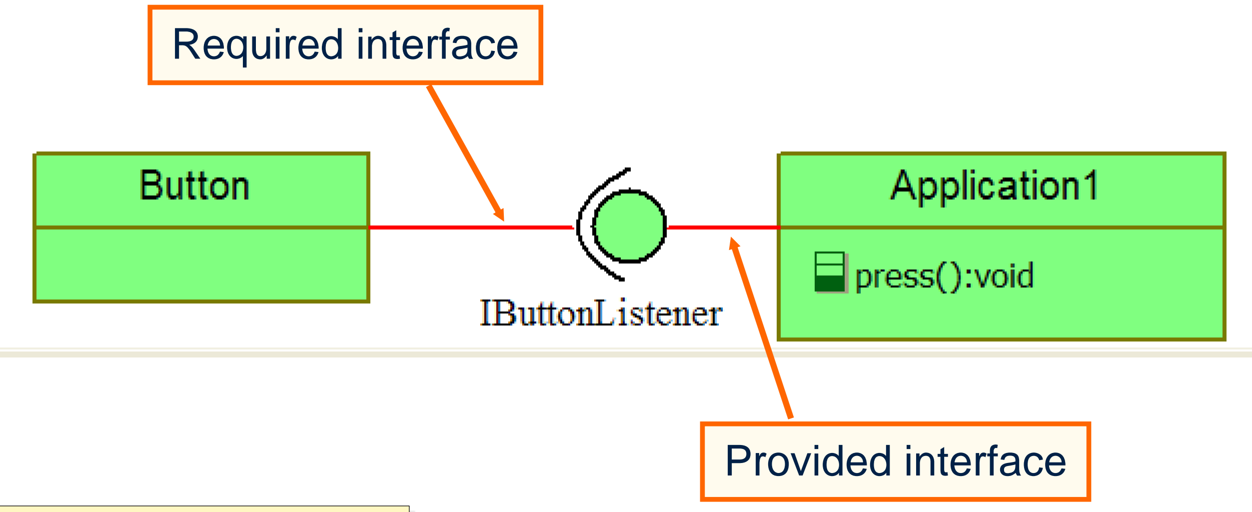
Interfaces

- An interface is *a named collection of operations.*
- UML interfaces specify only messages or operations.
- UML Interfaces have no implementation and generally have no attributes.
- Classes *realize* an interface by providing a method (implementation) of an operation:
  - A class that realizes an interface is said to be *compliant* with that interface.
  - Classes may realize any number of interfaces.
  - Interfaces may be realized by any number of classes.

> To make Interface classes more apparent, they are often named starting with an I. The name of the class and the abstract operations are generally italicized.

Ball and Socket notation

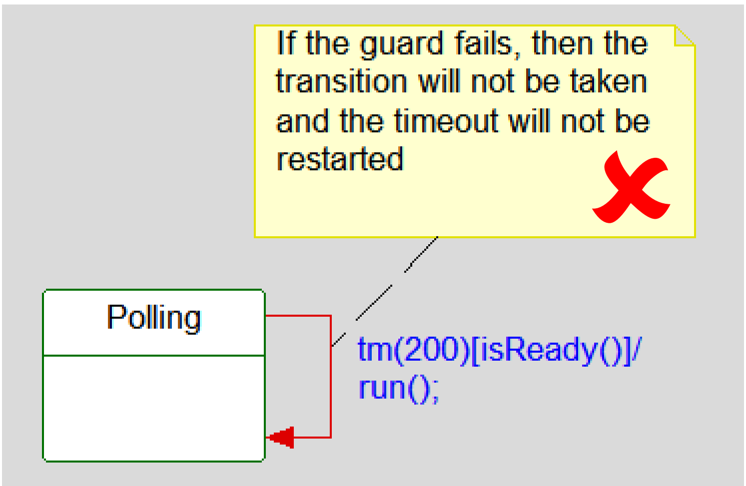
- Another way to draw an interface class and the realization of an interface is to use the Ball and Socket notation:

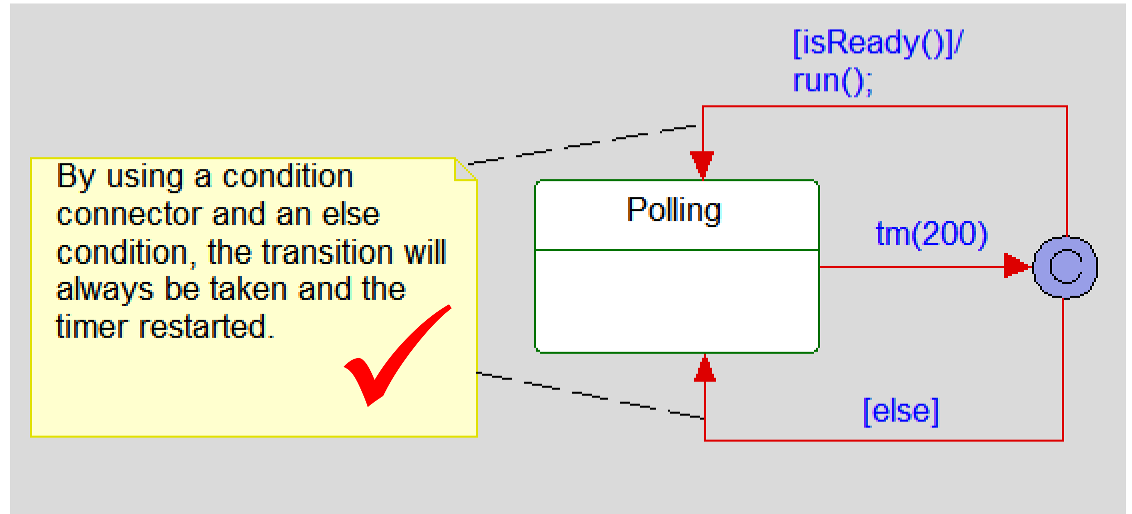


The *Ball* is sometimes referred to as a *lollipop*.
To recall if [ball symbol]
means required or provided, think that there is an O in prOvided.

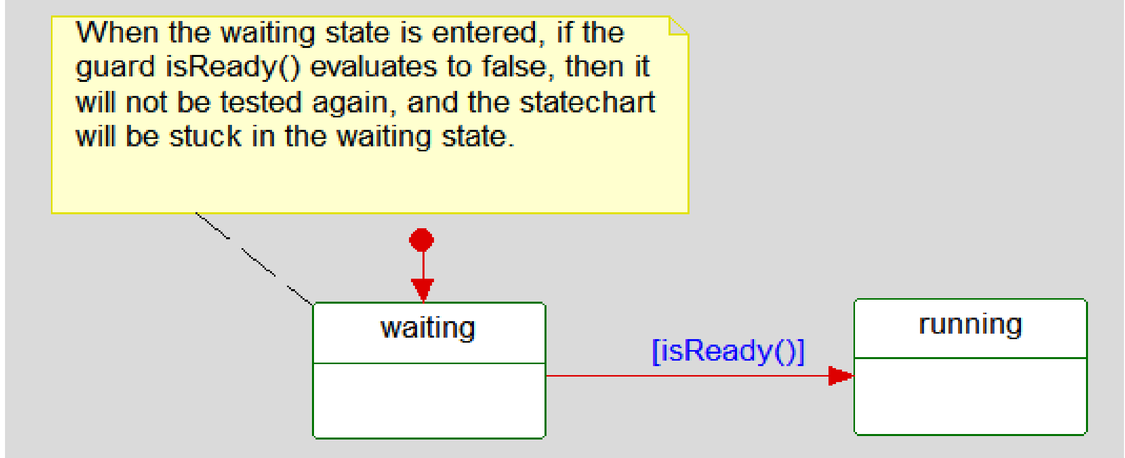
Avoiding common mistakes

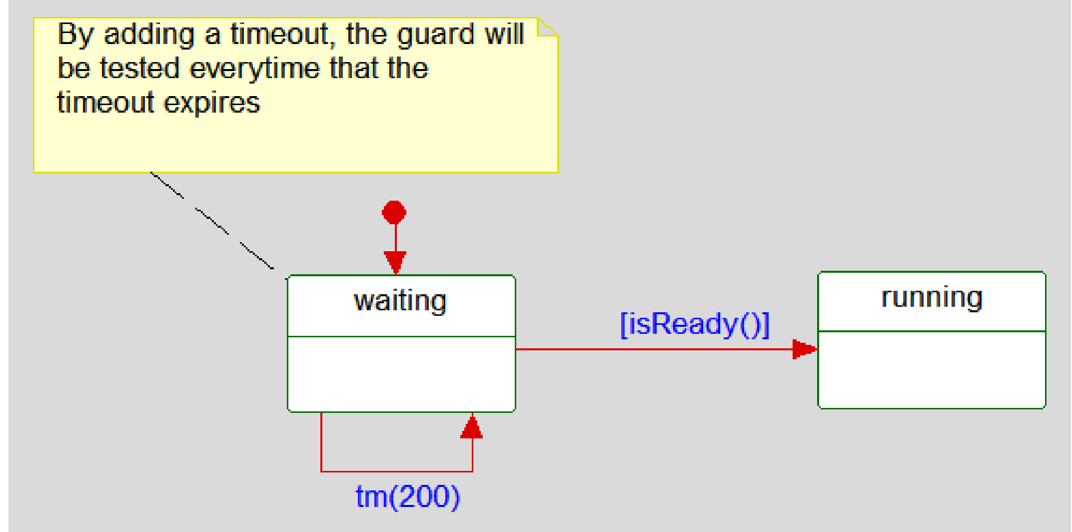
Dead lock situation 1



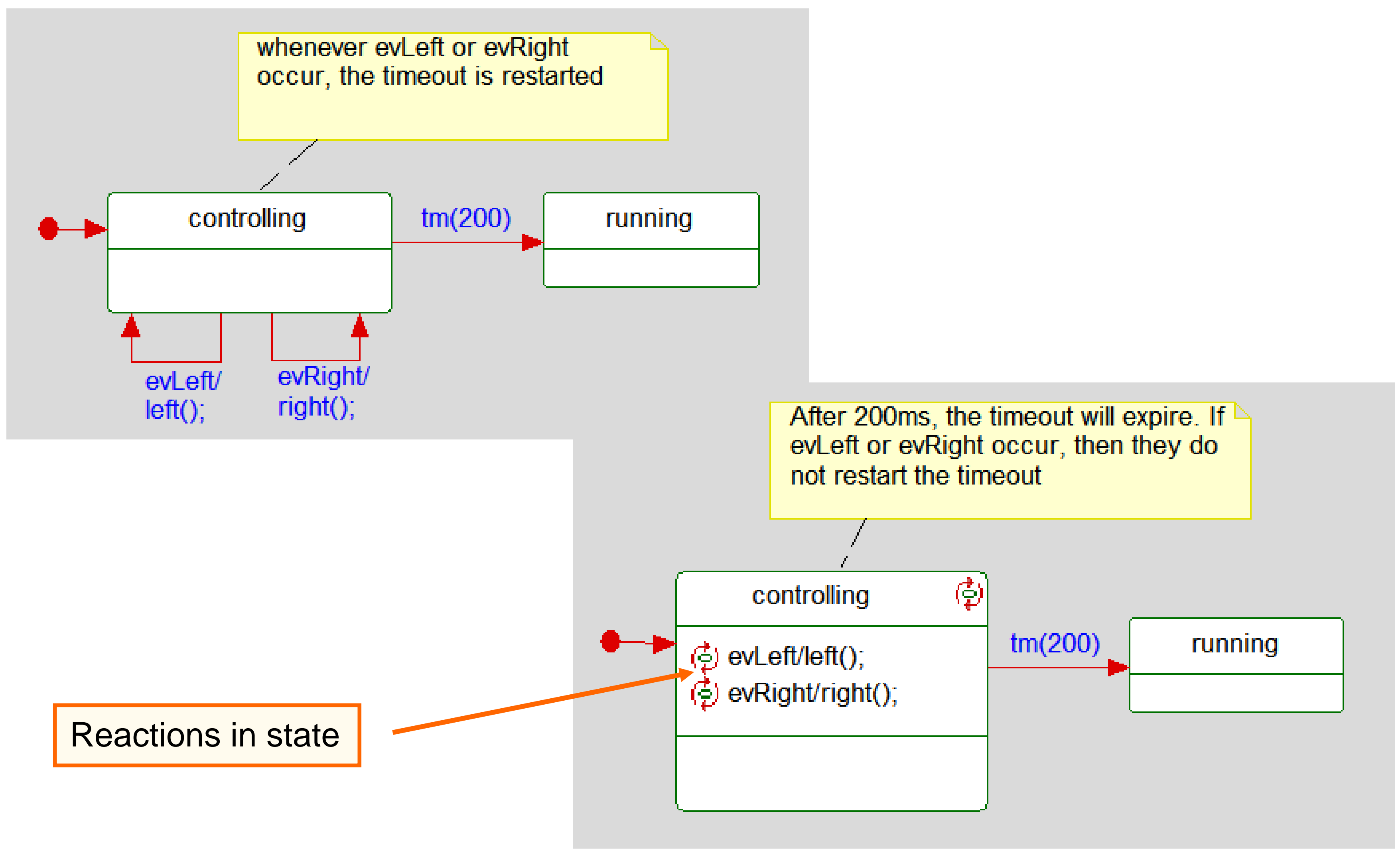


Dead lock situation 2





Transition or reaction in state?



IBM

Null transitions

- Each time a state is entered, Rational Rhapsody checks to see if there is a null transition that could be taken. If there is, then the transition is taken.
  - In the following example, there is no stable state since every state has a null transition with a guard that evaluates to True.
    - In this case, the following message is displayed: *Infinite loop of Null Transitions*.

A Null Transition is a transition that has no trigger.

/x=0;
y=0;

initialise

[x==0]

calibrating

[y==0]

moving

****

<Error Found Pitfall4[0] Infinite Loop of Null Transitions> Break point Active

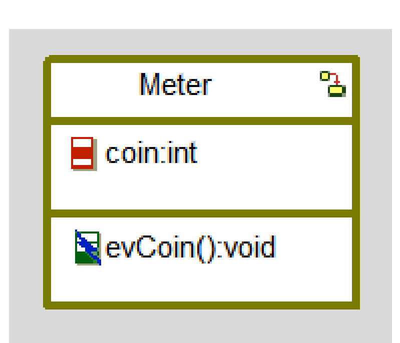
****

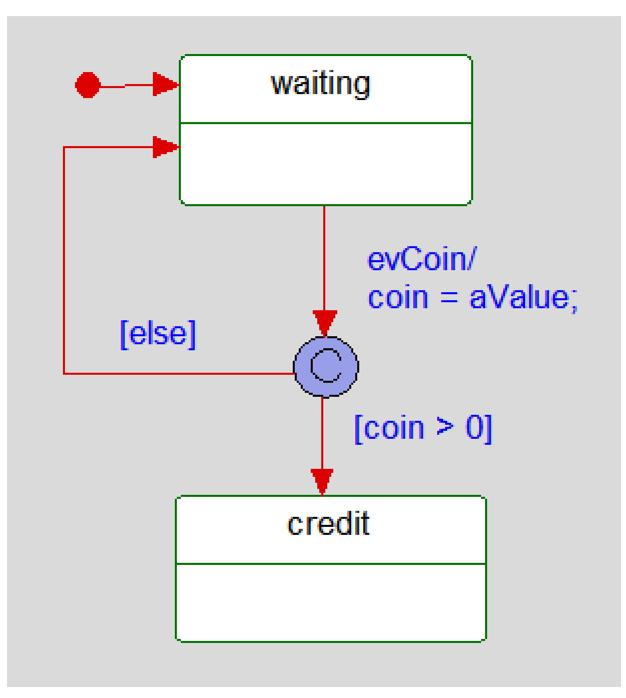
Executable is Idle

Build | Check Model | Configuration Management | **Animation**

What is done first?

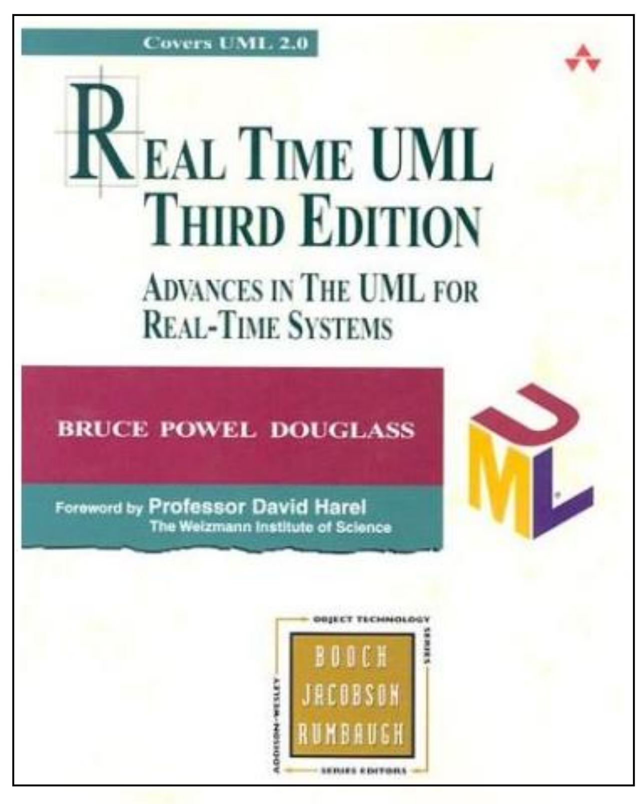
- When in the *waiting* state, what happens when the *evCoin* event is received?

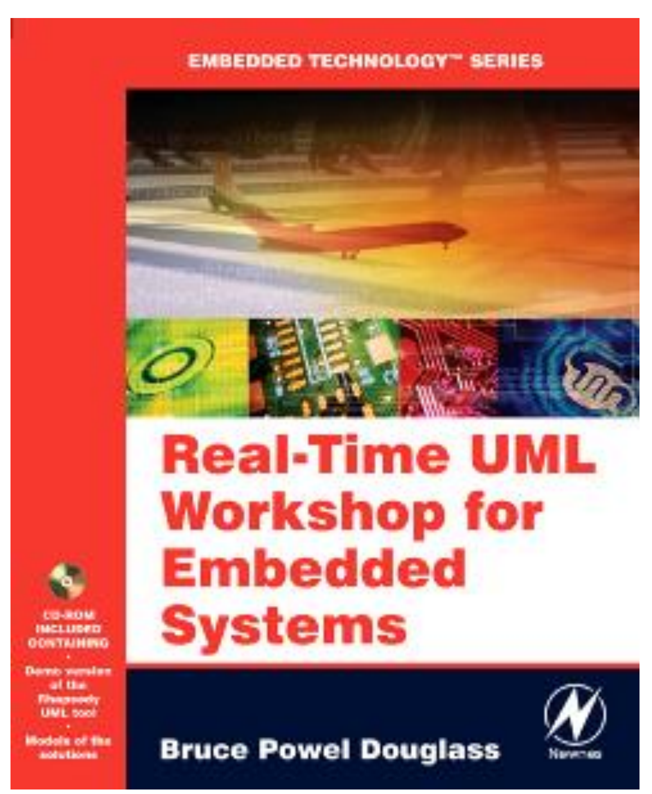




Answer : The value of coin is tested BEFORE it gets assigned!

Where to find out more





Cash Register

Page left intentionally blank