Object Modeling with OMG UML Tutorial Series

Behavioral Modeling

Gunnar Övergaard, Bran Selic and Conrad Bock

UML Revision Task Force
November 2000

Overview

- **Tutorial series**
- **UML Quick Tour**
- **Behavioral Modeling**
  - Part 1: Interactions and Collaborations
    - Gunnar Övergaard, Rational Software
  - Part 2: Statecharts
    - Bran Selic, Rational Software
  - Part 3: Activity Graphs
    - Conrad Bock, Kabira Technologies

Tutorial Series

- **Lecture 1: Introduction to UML: Structural Modeling and Use Cases**

- **Lecture 2: Behavioral Modeling with UML**

- **Lecture 3: Advanced Modeling with UML**

- **Lecture 4: Metadata Integration with UML, MOF and XMI**

Tutorial Goals

- ## What you will learn:
  - what the UML is and what is it not
  - UML's basic constructs, rules and diagram techniques
  - how the UML can model large, complex systems
  - how the UML can specify systems in an implementation-independent manner
  - how UML, XMI and MOF can facilitate metadata integration

- ## What you will not learn:
  - Object Modeling
  - Development Methods or Processes
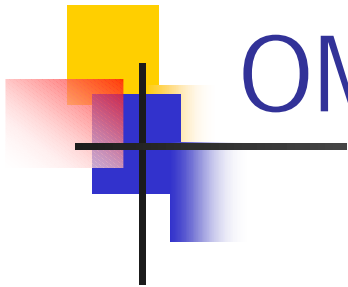 - Metamodeling

UML Quick Tour

- The UML is a graphical language for
  - specifying
  - visualizing
  - constructing
  - documenting

  the artifacts of software systems

- Added to the list of OMG adopted technologies in November 1997 as UML 1.1

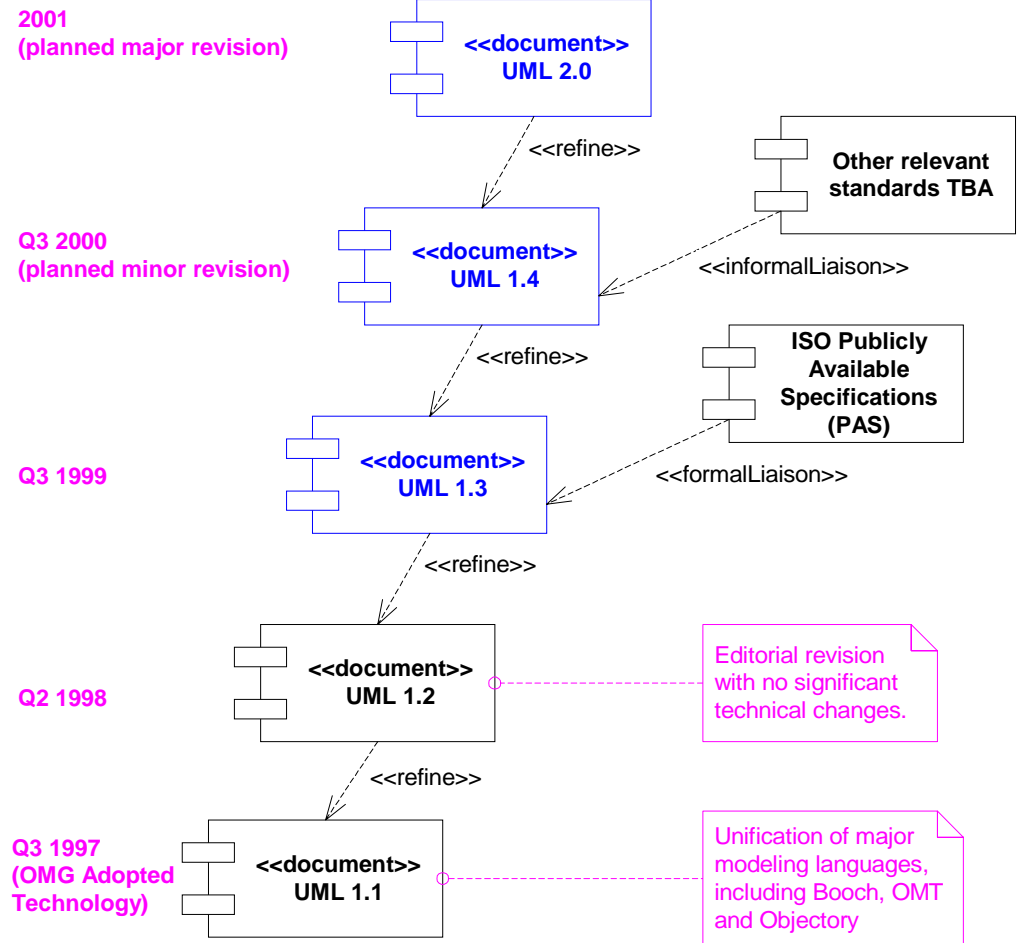- Most recent minor revision is UML 1.3 (November 1999)

UML Goals

- Define an easy-to-learn but semantically rich visual modeling language
- Unify the Booch, OMT, and Objectory modeling languages
- Include ideas from other modeling languages
- Incorporate industry best practices
- Address contemporary software development issues
  - scale, distribution, concurrency, executability, etc.
- Provide flexibility for applying different processes
- Enable model interchange and define repository interfaces

OMG UML Evolution

**2001
(planned major revision)**

<<document>>
UML 2.0

<<refine>>

**Other relevant
standards TBA**

**Q3 2000
(planned minor revision)**

<<document>>
UML 1.4

<<informalLiaison>>

<<refine>>

**ISO Publicly
Available
Specifications
(PAS)**

Q3 1999

<<document>>
UML 1.3

<<formalLiaison>>

<<refine>>

Q2 1998

<<document>>
UML 1.2

Editorial revision
with no significant
technical changes.

<<refine>>

**Q3 1997
(OMG Adopted
Technology)**

<<document>>
UML 1.1

Unification of major
modeling languages,
including Booch, OMT
and Objectory

OMG UML 1.3 Specification

- UML Summary
- UML Semantics
- UML Notation Guide
- UML Standard Profiles
  - Software Development Processes
  - Business Modeling
- UML CORBAfacility Interface Definition
- UML XML Metadata Interchange DTD
- Object Constraint Language

Tutorial Focus: the Language

- language = syntax + semantics
  - syntax = language elements (e.g. words) are assembled into expressions (e.g. phrases, clauses)
  - semantics = the meanings of the syntactic expressions
- *UML Notation Guide* – defines UML's graphic syntax
- *UML Semantics* – defines UML's semantics

Unifying Concepts

- **classifier-instance dichotomy**
  - e.g. an object is an instance of a class OR a class is the classifier of an object

- **specification-realization dichotomy**
  - e.g. an interface is a specification of a class OR a class is a realization of an interface

- **analysis-time vs. design-time vs. run-time**
  - modeling phases ("process creep")
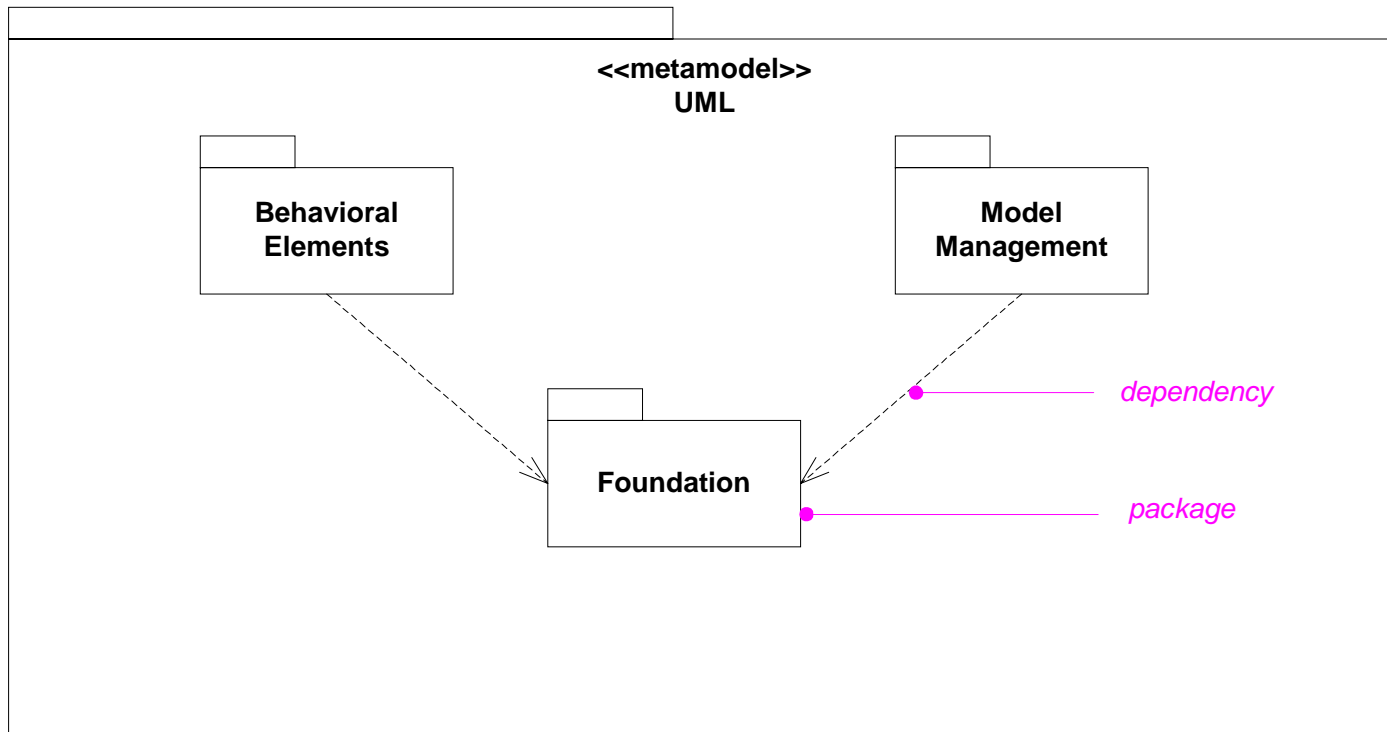 - usage guidelines suggested, not enforced

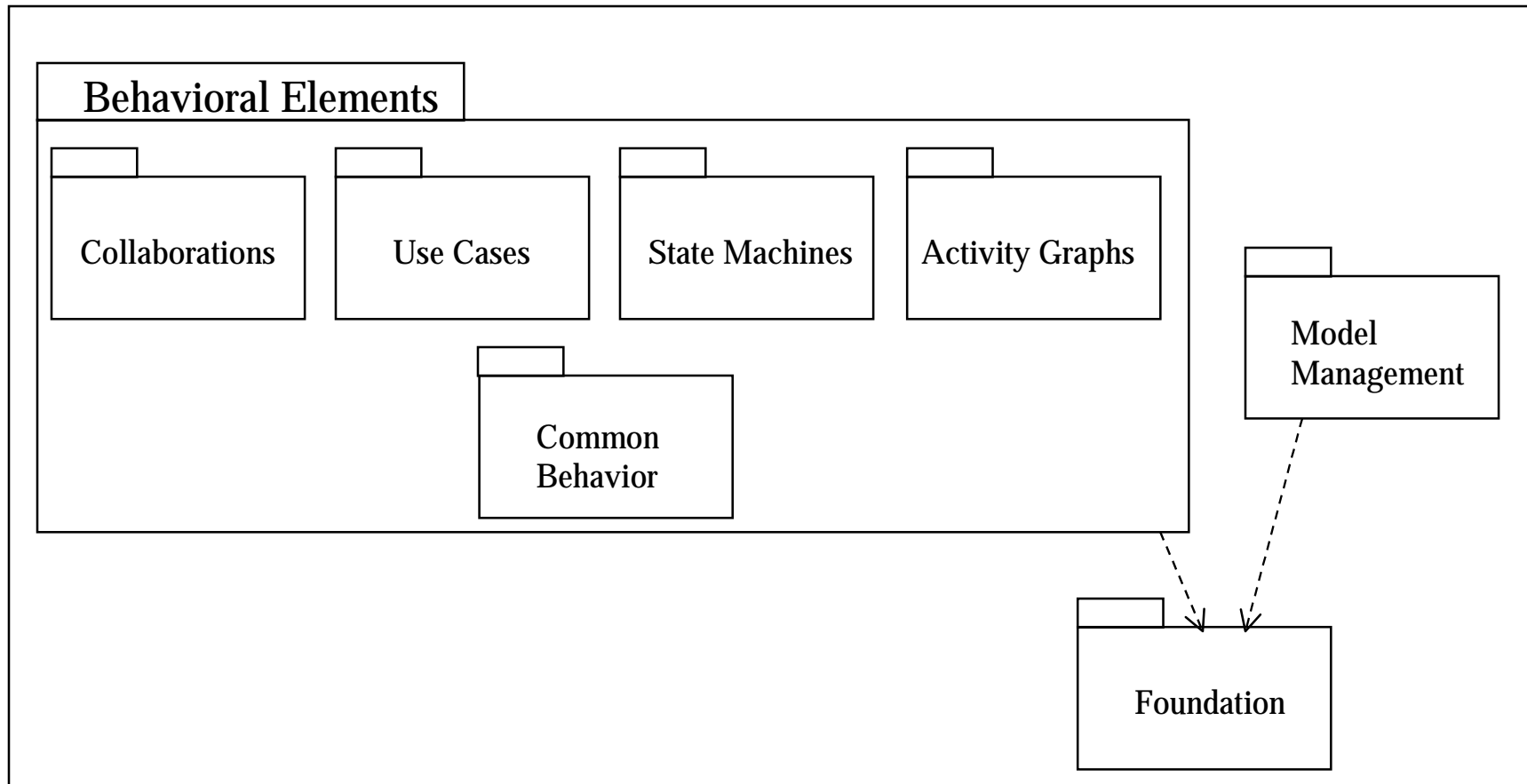# Language Architecture

- Metamodel architecture
- Package structure

Metamodel Architecture

```
┌──────────┐
│          │
┌──┴───────────┐
│ <<metamodel>>│  ○╌╌╌╌╌
│ MOF Meta-    │
│ metamodel    │
└──────────────┘
       △
       ┊
┌──────────┐
│          │
┌──┴───────────┐
│ <<metamodel>>│  ○╌╌╌╌╌
│ UML Metamodel│
└──────────────┘
       △
       ┊
┌──────────┐
│          │
┌──┴───────────┐
│              │
│  User Model  │  ○╌╌╌╌╌
│              │
└──────────────┘
```

Meta-Metamodel Layer (M3): Specifies meta-metaclasses for the UML metamodel

Metamodel Layer (M2): Specifies metaclasses for the UML metamodel, such as Class

Model Layer (M1): Specifies classes for the UML user models, such as Passenger, Ticket, TravelAgency

```
┌───────────┐   ┌───────────┐   ┌───────────┐
│   :Foo    │   │   :Bar    │   │   :Baz    │
└───────────┘   └───────────┘   └───────────┘
```

User Objects Layer (M0): User objects that are instances of UML user model classes, such as instances of Passenger, Ticket, TravelAgency

Package Structure

Package Structure

Behavioral Modeling

- Part 1: Interactions and Collaborations

- Part 2: Statecharts

- Part 3: Activity Diagrams

Interactions

- What are interactions?

- Core concepts

- Diagram tour

- When to model interactions

- Modeling tips

- Example: A Booking System

What are interactions?

- Interaction: a collection of communications between instances, including all ways to affect instances, like operation invocation, as well as creation and destruction of instances
- The communications are partially ordered (in time)

Interactions: Core Elements

Construct	Description	Syntax
Instance (object, data value, component instance etc.)	An entity with a unique identity and to which a set of operations can be applied (signals be sent) and which has a state that stores the effects of the operations (the signals).	 name ————— attr values
Action	A specification of an executable statement. A few different kinds of actions are predefined, e.g. CreateAction, CallAction, DestroyAction, and UninterpretedAction.	textual

Interaction: Core Elements (cont'd)

Construct	Description	Syntax
Stimulus	A communication between two instances.	⟶
Operation	A declaration of a service that can be requested from an instance to effect behavior.	textual
Signal	A specification of an asynchronous stimulus communicated between instances.	«Signal» Name / parameters

Interaction: Core Relationships

Construct	Description	Syntax
Link	A connection between instances.	————
Attribute Link	A named slot in an instance, which holds the value of an attribute.	textual

Example: Instance

underlined name

triangle : Polygon

triangle : Polygon

triangle : Polygon
center : Point = (2,2) vertices : Point* = ((0,0), (4, 0), (2,4)) borderColor : Color = black fillColor : Color = white

triangle

: Polygon

attribute links
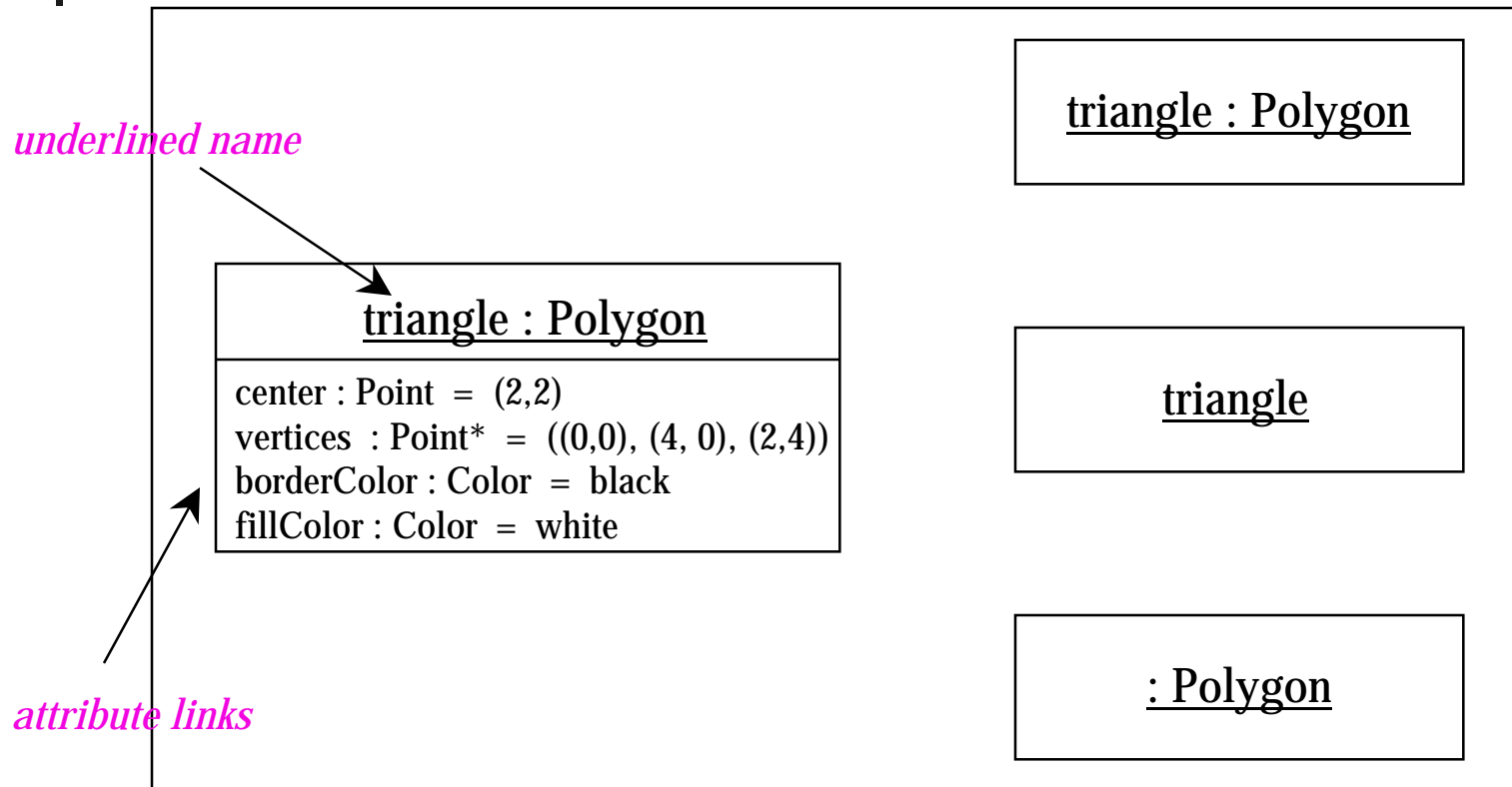
Example: Instances and Links

: Family

Joe : Person —— husband ———————— wife —— Jill : Person

Operation and Method

Triangle

+ move (**in** dist : Point)
+ scale (**in** factor : Real)

foreach v **in** vertices **do**
  v.x := v.x + dist.x;
  v.y := v.y + dist.y

foreach v **in** vertices **do**
  v.x := factor * v.x;
  v.y := factor * v.y
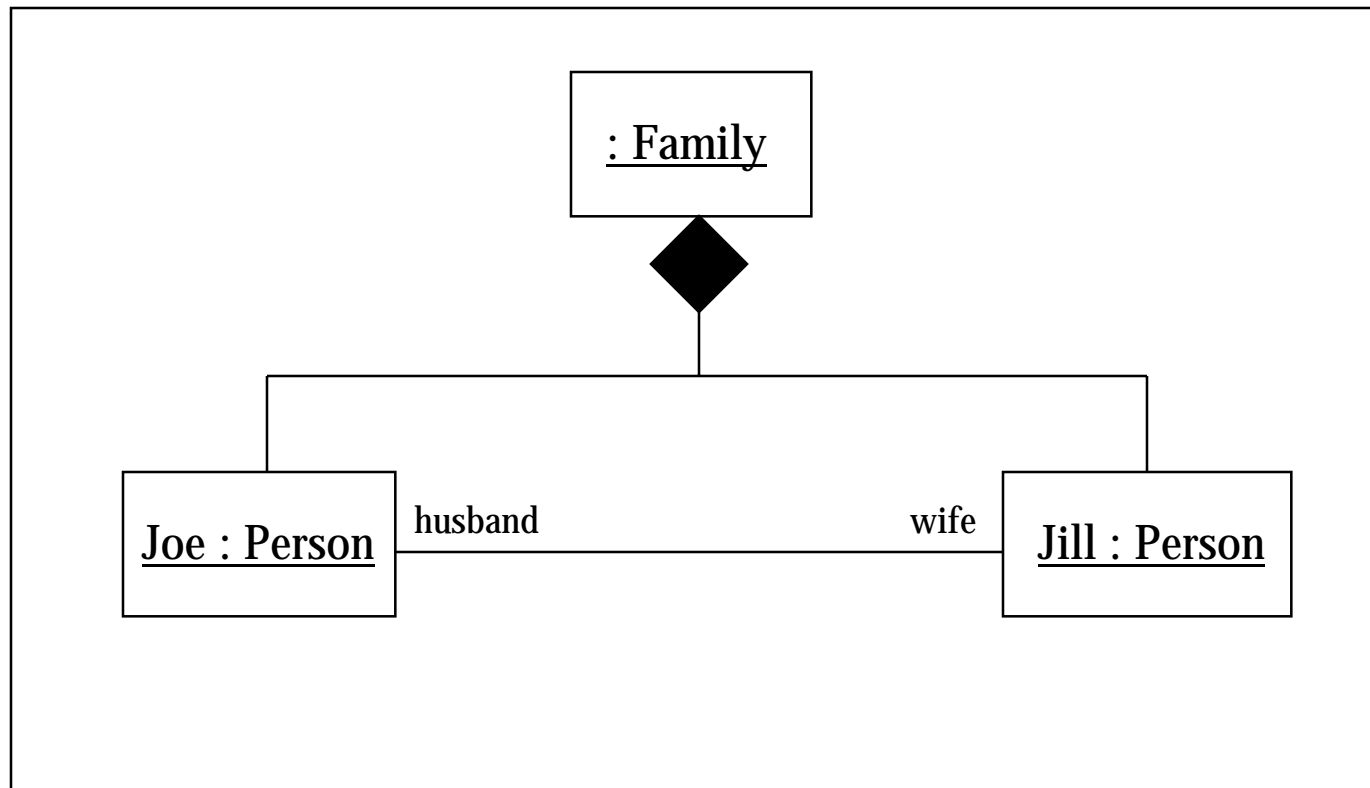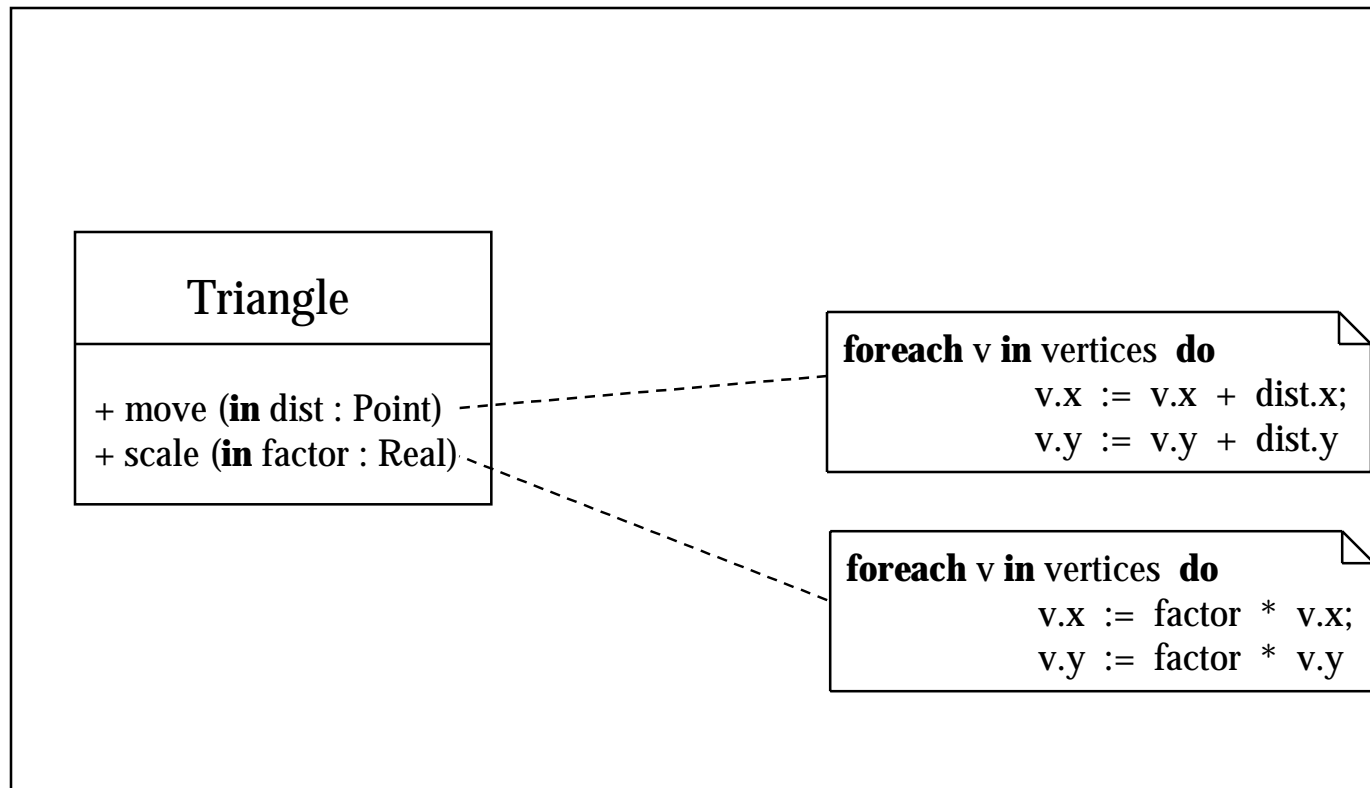
Interaction Diagram Tour

- Show interactions between instances in the model
  - graph of instances (possibly including links) and stimuli
  - existing instances
  - creation and deletion of instances
- Kinds
  - sequence diagram (temporal focus)
  - collaboration diagram (structural focus)

Interaction Diagrams

Sequence Diagram

x   y   z

a →

b →

c →

Collaboration Diagram

1.1: a
1.2: c

x ──→ y

1.1.1: b

z

Sequence Diagram

object symbol → | name : Class | | other |

lifeline →

stimulus

activation →

name (…)

new (…)

| : Class |

delete

return   *create*
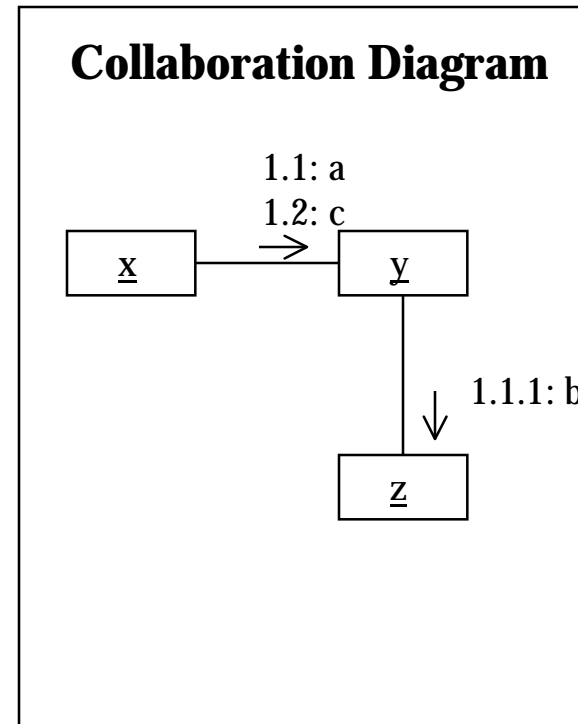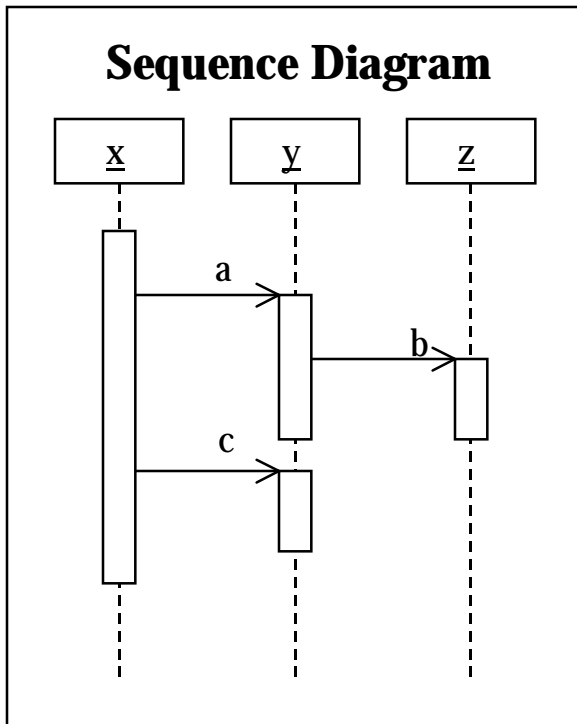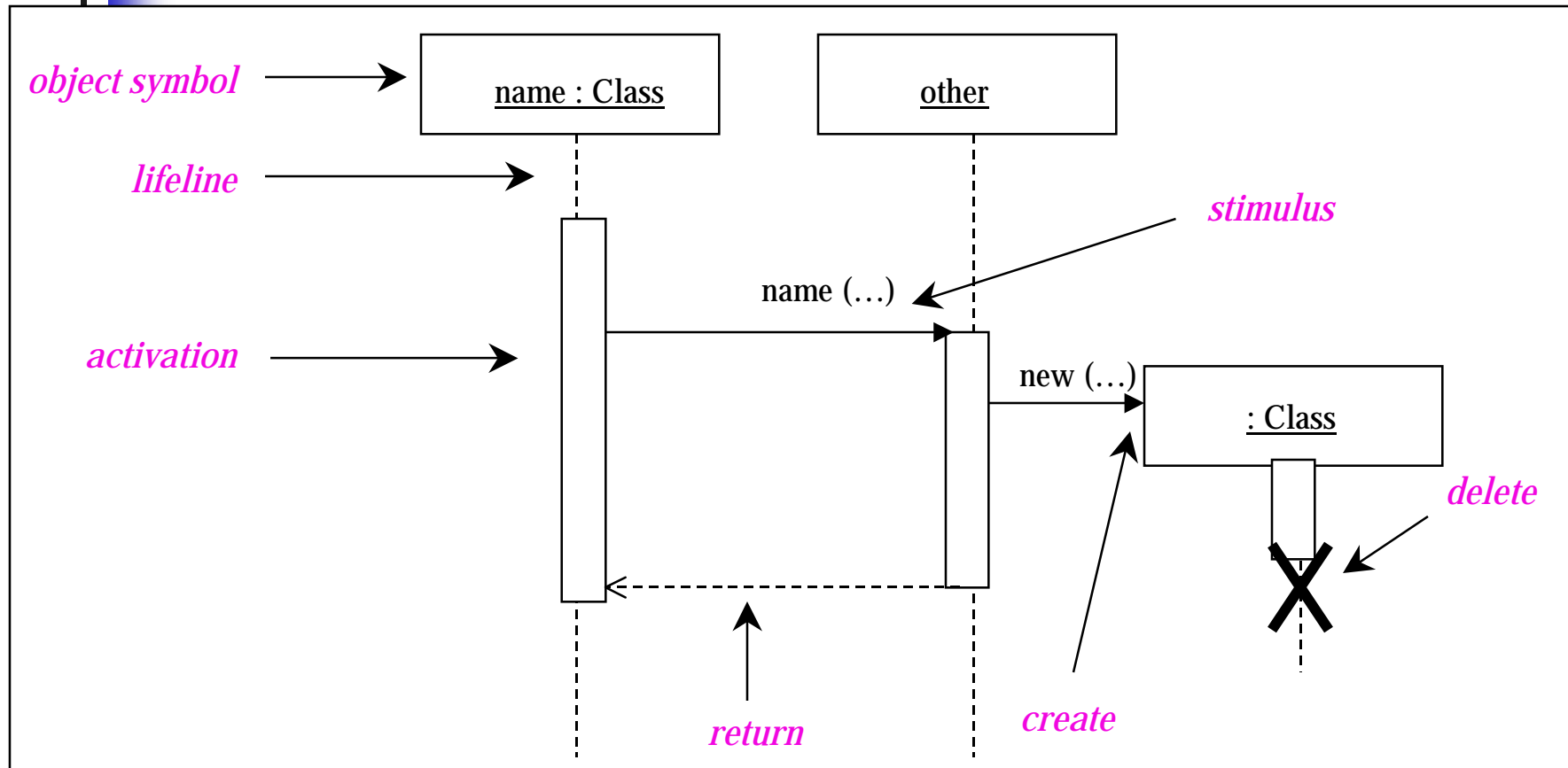
Arrow Label

predecessor guard-condition sequence-expression return-value := message-name argument-list

move (5, 7)

3.7.4: move (5, 7)

A3, B4 / [x < 0] C3.1: res := getLocation (fig)

3.7 *[1..5]: move (5, 7)

predecessor   guard   sequence number   iteration

return value   message name   argument list

3.7 [z > 0]: move (5, 7)   condition
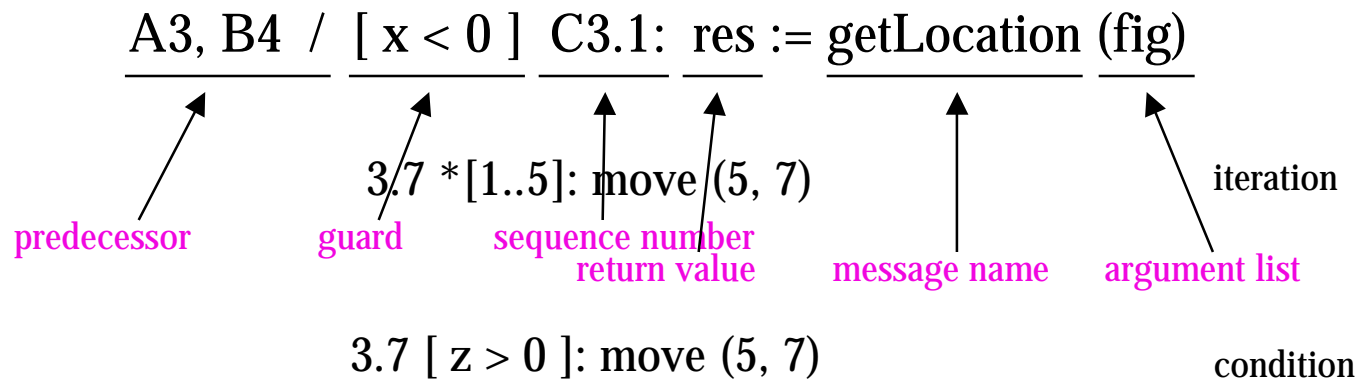
Different Kinds of Arrows
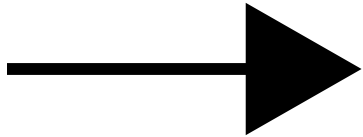
Procedure call or other kind of nested flow of control

Flat flow of control

Explicit asynchronous flow of control
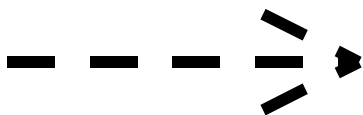
Return

Example: Different Arrows

Nested Flow	Flat Flow	Asynchronous Flow
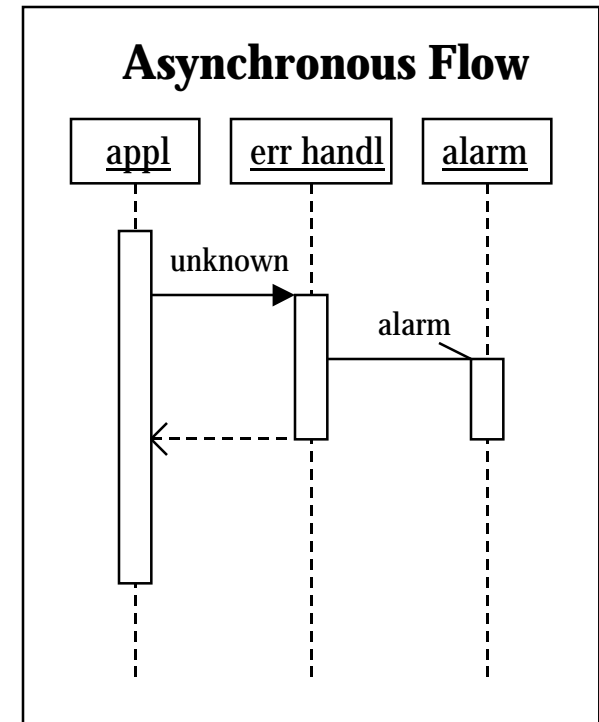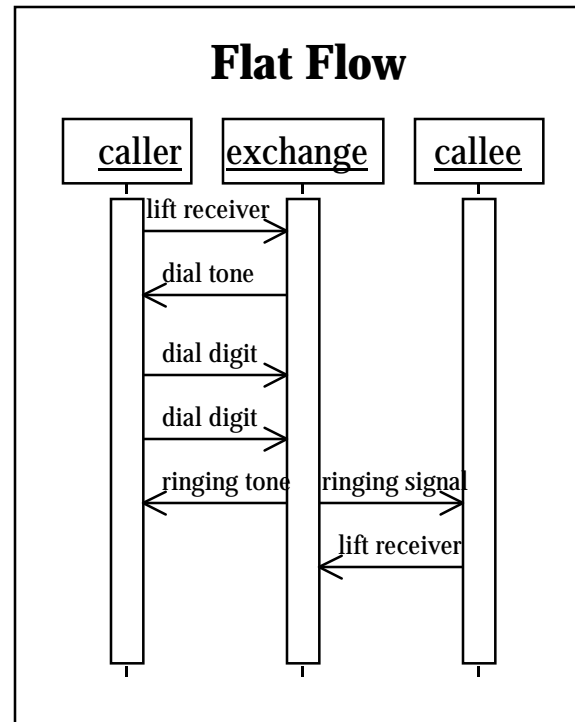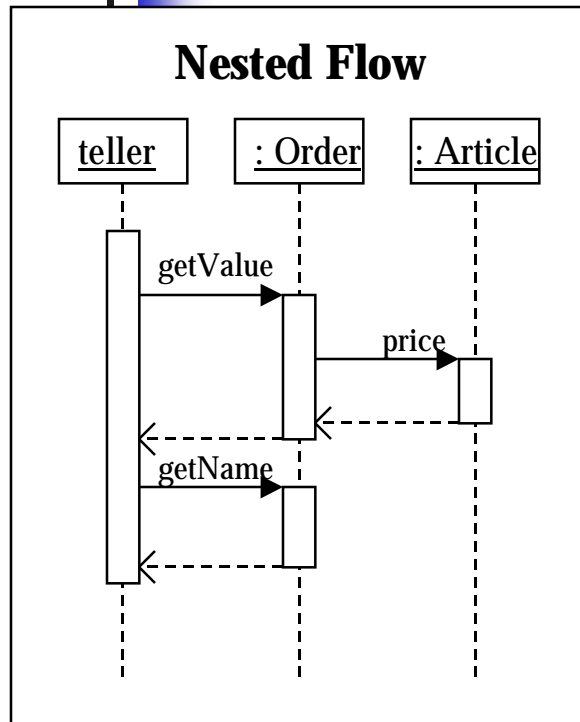
Recursion, Condition, etc.

calculator

filter

value

[x > 0] getValue ()

[x < 0] transform ()

getValue ()

iterate ()

Collaboration Diagram

object symbol

link symbol

standard
stereotype

stimulus

redisplay ()

: Controller

window

: Window

1: displayPositions (window)

standard
stereotype

standard
constraint

window «parameter»

1.1.3.1 add (self)

1.1 *[i := 1..n]: drawSegment (i)

wire

contents {new}

standard
stereotype

wire :Wire

«local» line

{new}
: Line

«self»

1.1.2: create (r0, r1)
1.1.3: display (window)

standard
constraint

1.1.1a: r0 := position ()

1.1.1b: r1 := position ()

left : Bead

right : Bead

Behavioral Modeling with UML

31

When to Model Interactions

- To specify how the instances are to interact with each other.

- To identify the interfaces of the classifiers.

- To distribute the requirements.

Interaction Modeling Tips

- Set the context for the interaction.
- Include only those features of the instances that are relevant.
- Express the flow from left to right and from top to bottom.
- Put active instances to the left/top and passive ones to the right/bottom.
- Use sequence diagrams
  - to show the explicit ordering between the stimuli
  - when modeling real-time
- Use collaboration diagrams
  - when structure is important
  - to concentrate on the effects on the instances

Example: A Booking System

Use Case Description: Change Flt Itinerary

- **Actors:** traveler, client account db, airline reservation system

- **Preconditions:** Traveler has logged in

- **Basic course:**
  - Traveler selects 'change flight itinerary' option
  - System retrieves traveler's account and flight itinerary from client account database
  - System asks traveler to select itinerary segment she wants to change; traveler selects itinerary segment.
  - System asks traveler for new departure and destination information; traveler provides information.
  - If flights are available then ...
  - ...
  - System displays transaction summary.

- **Alternative course:**
  - If no flights are available then...

Sequence Diagram: Change Flight Itinerary

Traveler	: Booking System	Client Account DBMS	Airline Reservation System

change flight itinerary

get customer account

get itinerary

present itinerary

select segment

present detailed info

update information

available flight
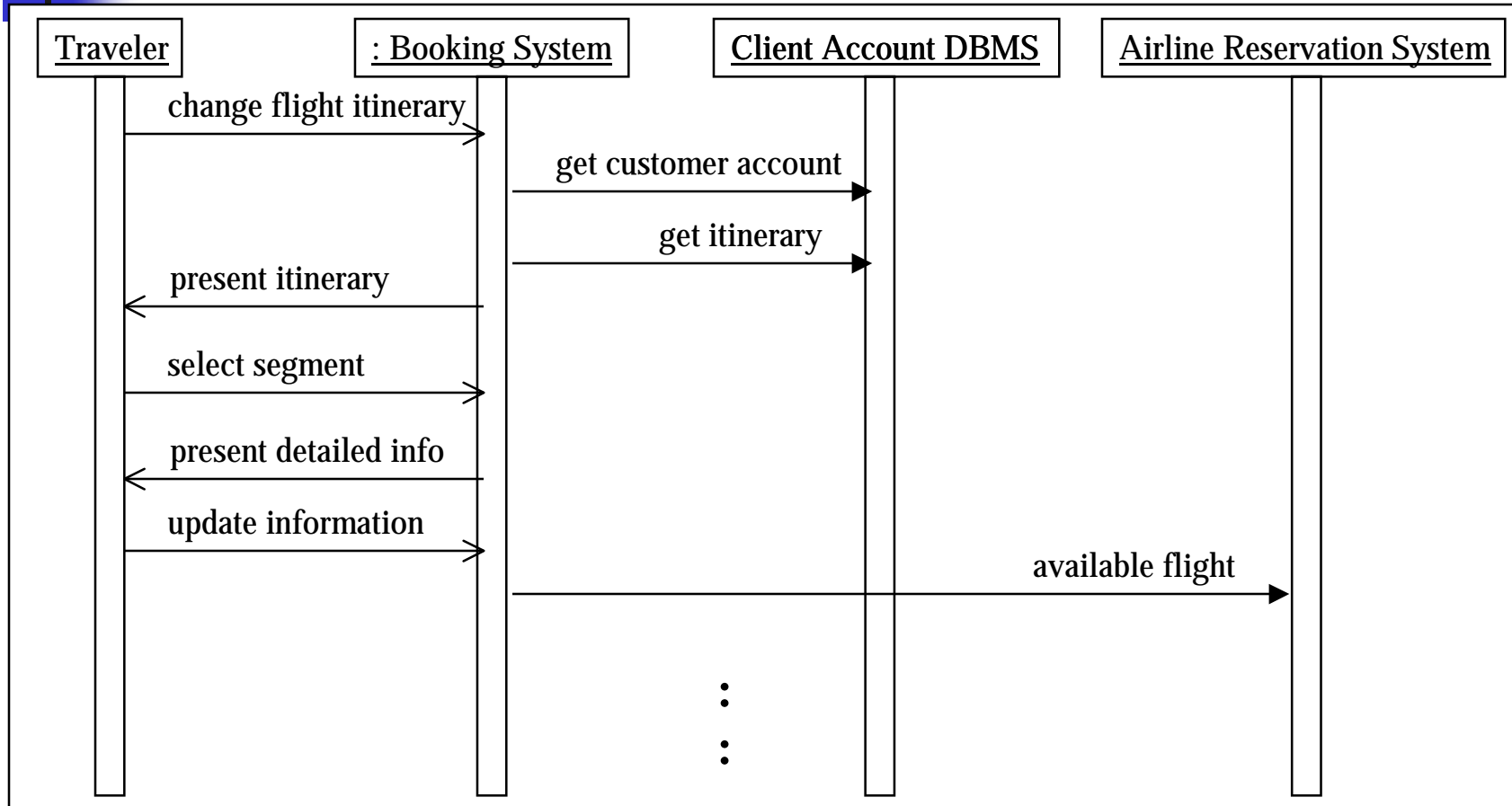
Collaboration Diagram: Change Flt Itinerary

1: change flight itinerary
5: select segment
7: update information

2: get customer account
3: get itinerary

: Booking System

4: present itinerary
6: present detailed info

Traveler

Client Account DBMS

8: available flight

Airline Reservation System
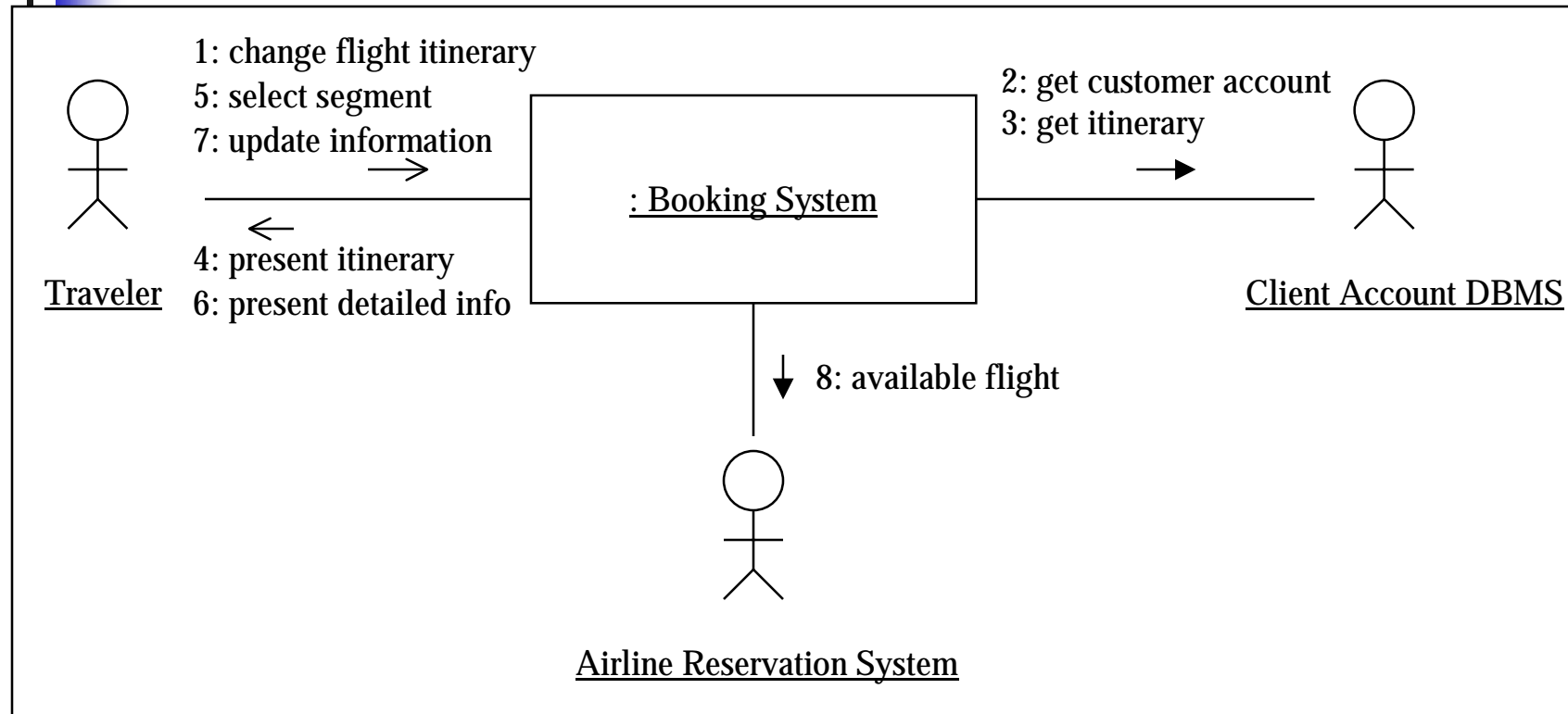
Collaboration

- What is a collaboration?

- Core concepts

- Diagram tour

- When to model collaborations

- Modeling tips

- Example: A Booking System
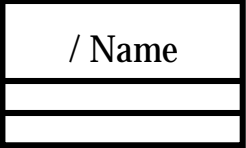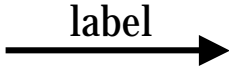
What is a collaboration?

- Collaboration: a collaboration defines the roles a set of instances play when performing a particular task, like an operation or a use case.

- Interaction:an interaction specifies a communication pattern to be performed by instances playing the roles of a collaboration.
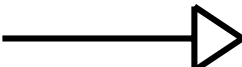
Collaborations: Core Elements

Construct	Description	Syntax
Collaboration	A collaboration describes how an operation or a classifier, like a use case, is realized by a set of classifiers and associations used in a specific way. The collaboration defines a set of roles to be played by instances and links, as well as a set of interactions that define the communication patterns between the instances when they play the roles.	Name

Collaborations: Core Elements (cont'd)

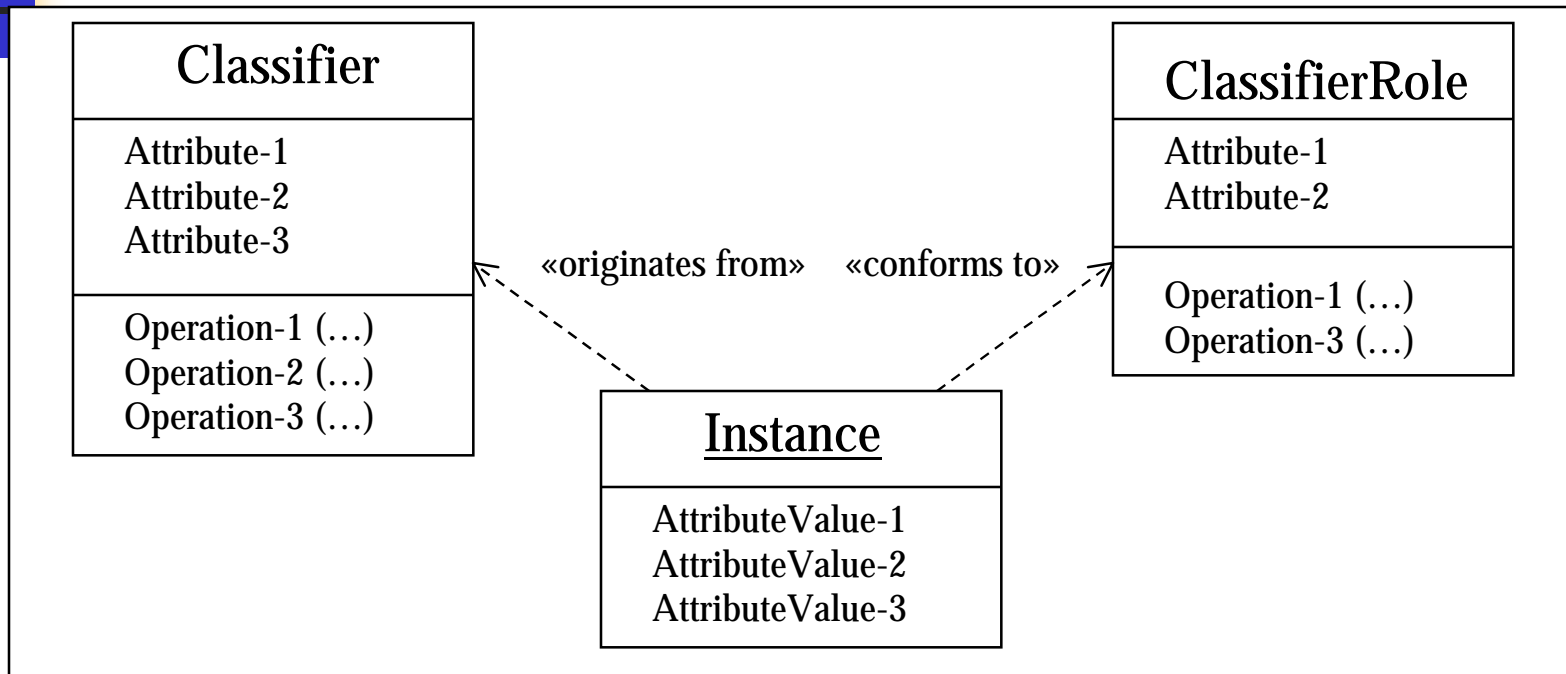| Construct	Description	Syntax
Classifier Role	A classifier role is a specific role played by a participant in a collaboration. It specifies a restricted view of a classifier, defined by what is required in the collaboration.	/ Name
Message	A message specifies one communication between instances. It is a part of the communication pattern given by an interaction.	label →

Collaborations: Core Relationships

Construct	Description	Syntax
Association Role	An association role is a specific usage of an association needed in a collaboration.	——
Generalization	A generalization is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element.	⟶▷

Classifier-Instance-Role Trichotomy

- An Instance is an entity with behavior and a state, and has a unqiue identity.

- A Classifier is a description of an Instance.

- A Classifier Role defines a usage (an abstraction) of an Instance.

id

«originates from»

ClassName

«conforms to»

/ RoleName

«view of»

Classifier-Instance-Role Trichotomy (cont'd)

Classifier

Attribute-1
Attribute-2
Attribute-3

Operation-1 (…)
Operation-2 (…)
Operation-3 (…)

ClassifierRole

Attribute-1
Attribute-2

Operation-1 (…)
Operation-3 (…)

«originates from»   «conforms to»
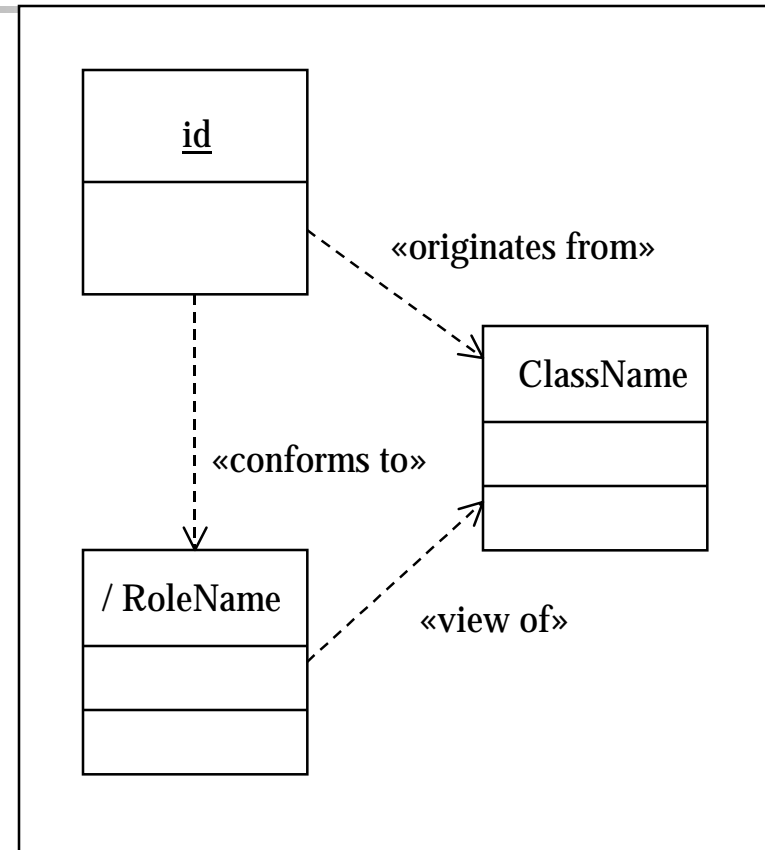
Instance

AttributeValue-1
AttributeValue-2
AttributeValue-3

•The attribute values of an Instance corresponds to the attributes of its Classifier.

•All attributes required by the ClassifierRole have corresponding attribute values in the Instance.

•All operations defined in the Instance's Classifier can be applied to the Instance.

•All operations required by the ClassifierRole are applicable to the Instance.

Different Ways to Name a Role

/ ClassifierRoleName : ClassifierName

A role name is preceeded by a '/'   *A classifier name is preceeded by a ':'*

Example: | / Parent : Person | | / Parent | | : Person |

instanceName / ClassifierRoleName : ClassifierName

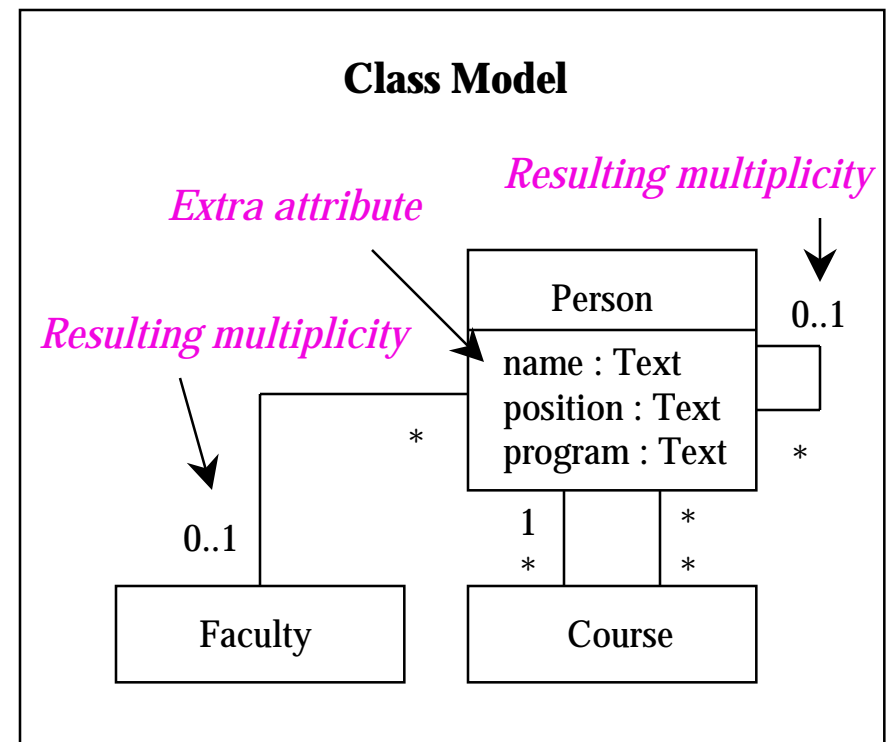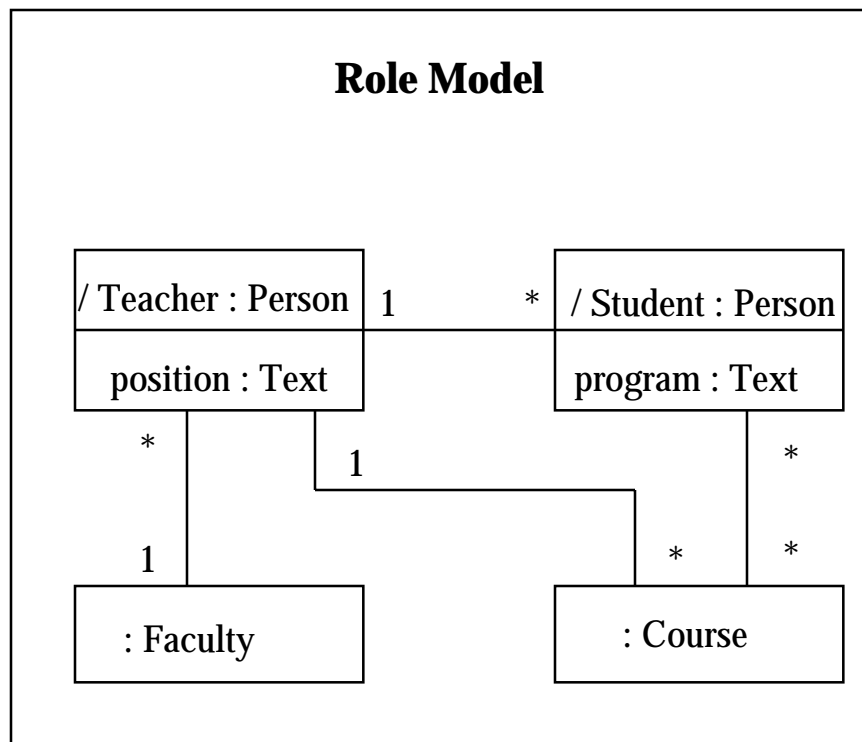Example: | : Person | | Charlie | | Charlie : Person |

| Charlie / Parent | | Charlie / Parent : Person |

Association and Association Role

•An Association Role specifies the required properties of a Link used in a Collaboration.
•The properties of an AssociationEnd may be restricted by a AssociationEndRole.

Example: A School

/ Teacher : Person
position : Text

1 tutor   student *

/ Student : Person
program : Text

faculty member *

1 lecturer

participant *

faculty 1

given course *

taken course *

: Faculty

: Course
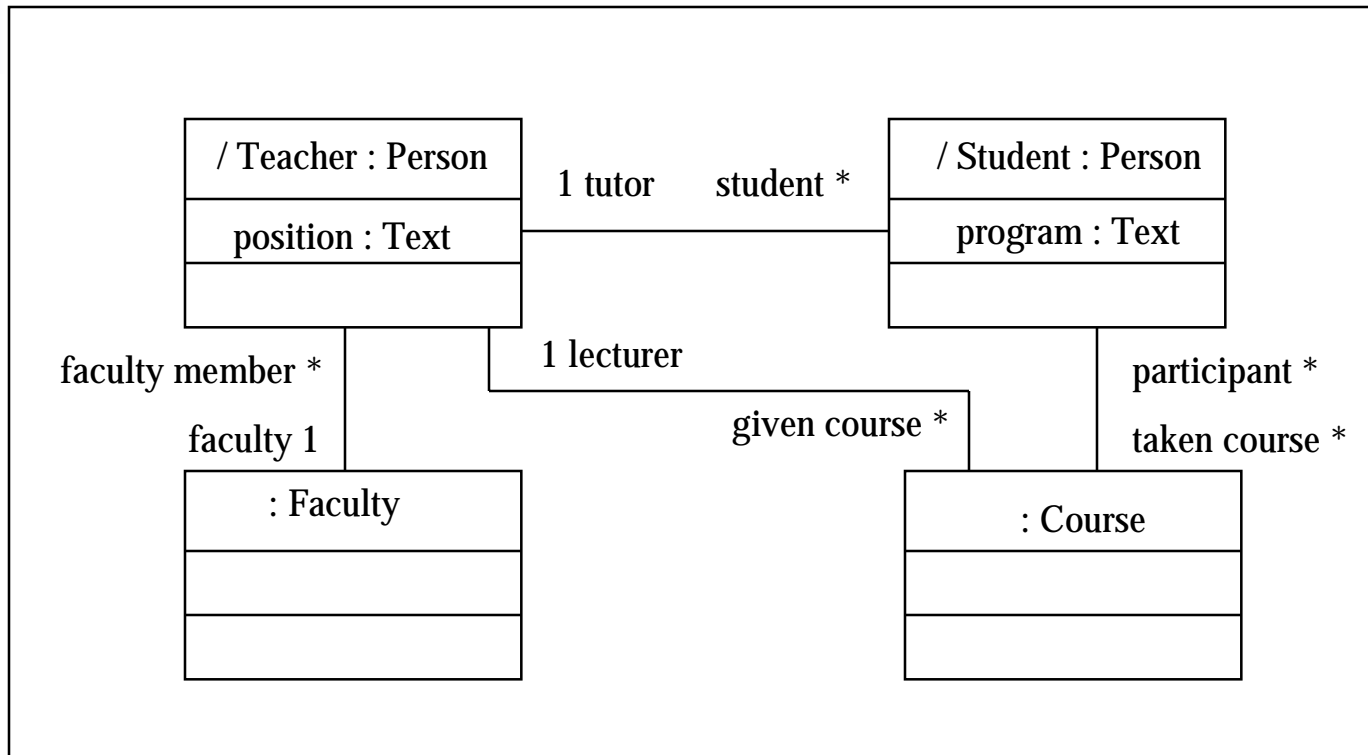
Role Model vs. Class Model

The Classes give the complete description while the Roles specify one usage.

Role Model

/ Teacher : Person	1
position : Text	

/ Student : Person
program : Text

*

1

*

*

*

1

: Faculty

: Course

Class Model

Extra attribute

Resulting multiplicity

Resulting multiplicity

Person
name : Text position : Text program : Text

0..1

*

*

0..1

1

*

*

*

*

Faculty

Course

A Collaboration and Its Roles

A Collaboration and how its roles are mapped onto a collection of Classifiers and Associations.

Patterns in UML

Constraint that must be fulfilled in each instance of this pattern.

Handler.reading = length (Subject.queue)
Handler.range = {0..Subject.capacity}

Observer

Subject

Handler

CallQueue

queue : List of Call
source : Object
waitAlarm : Alarm
capacity : Integer

SlidingBarIcon
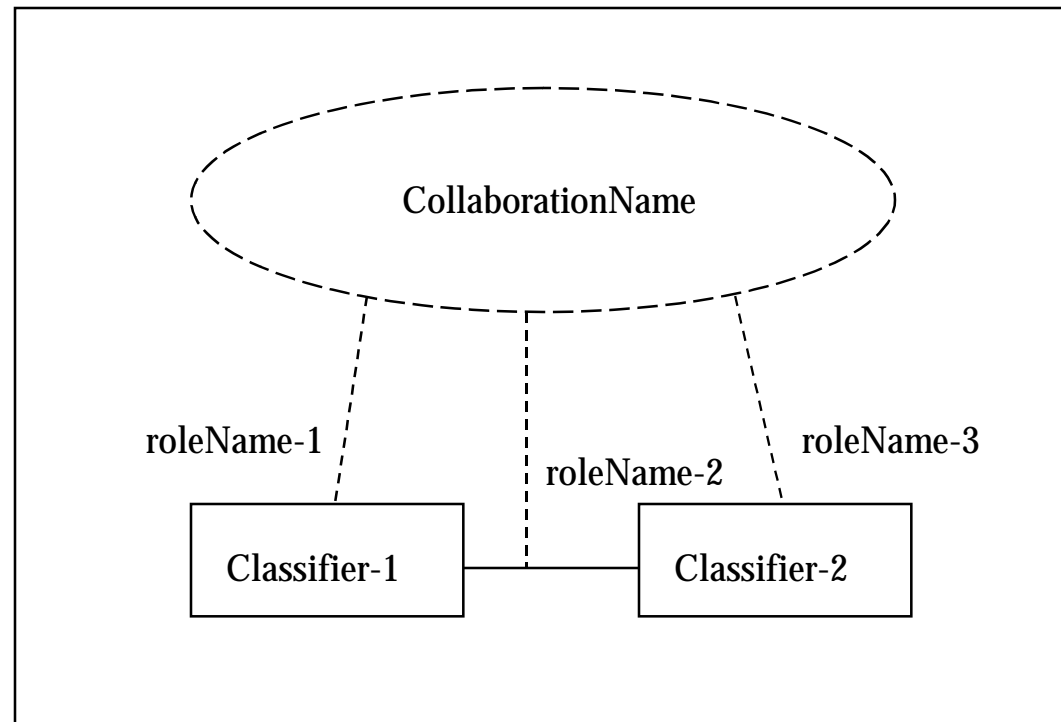
reading : Real
color : Color
range : Interval
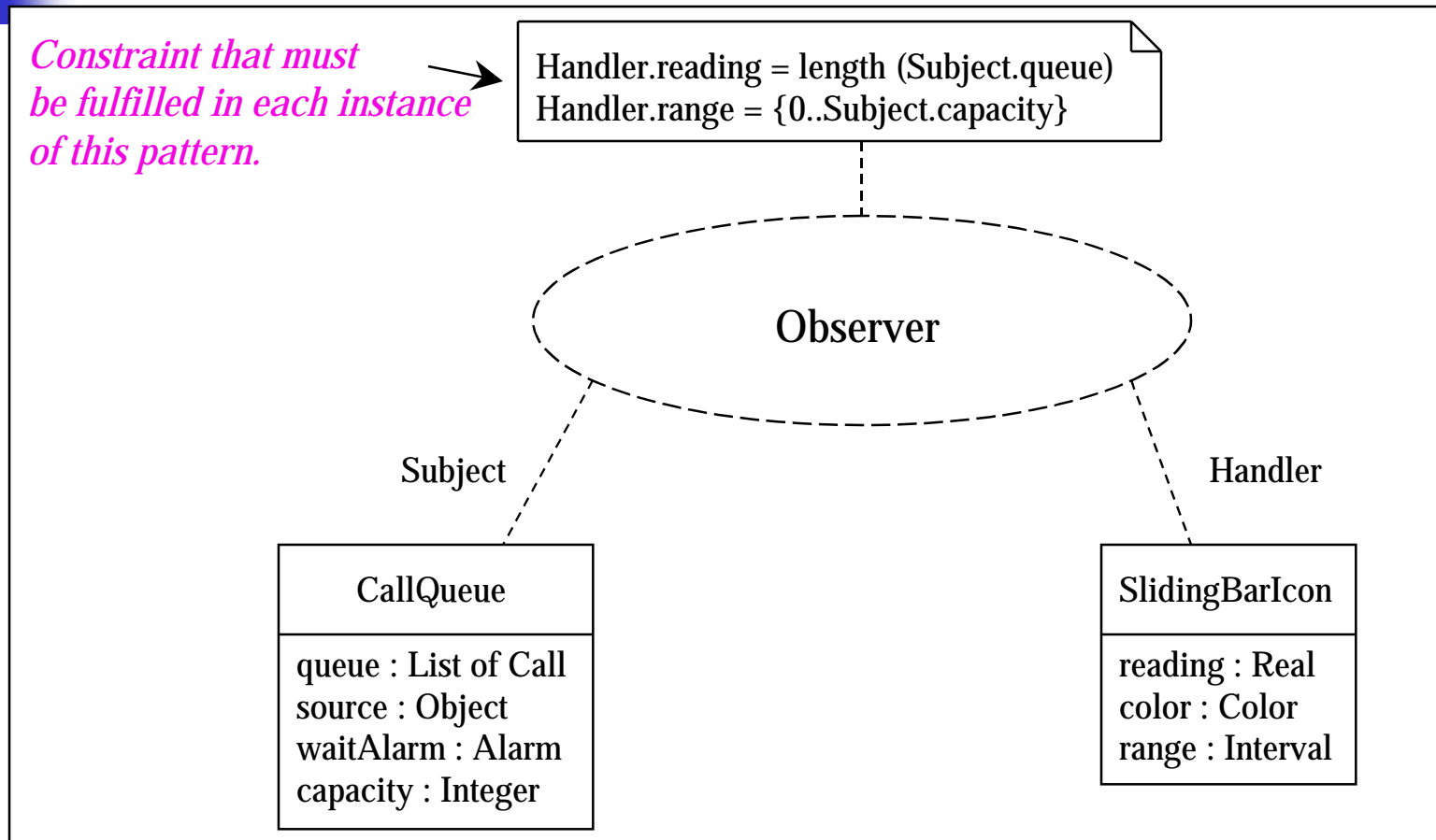
Generalization Between Collaborations
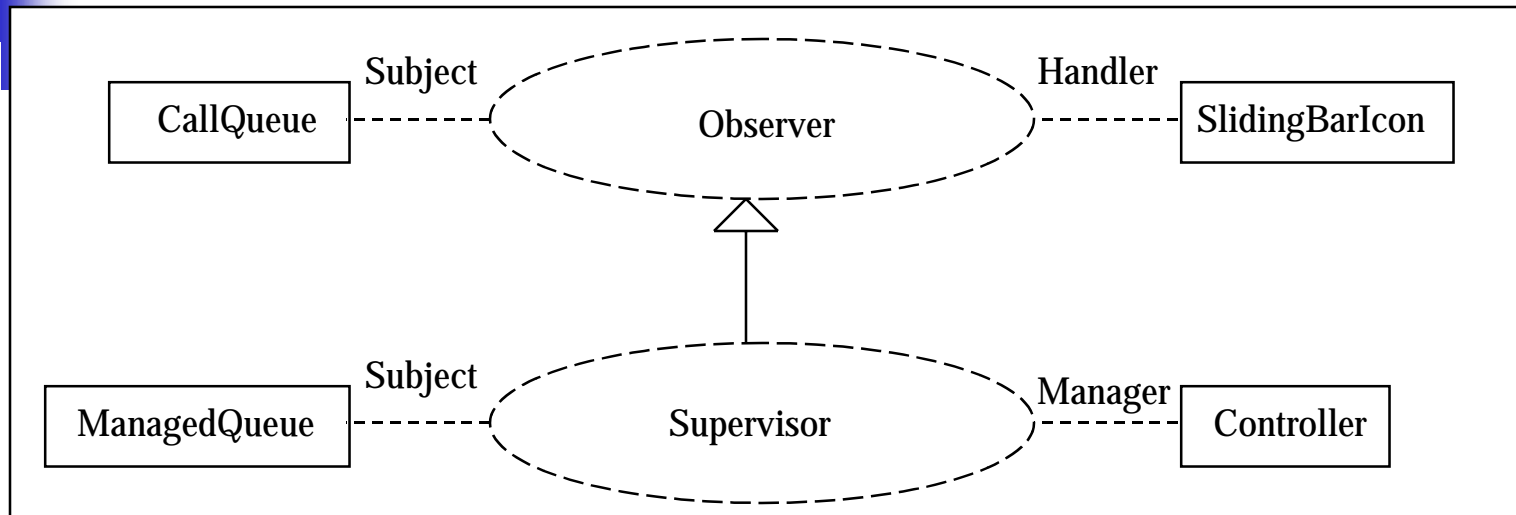
- All roles defined in the parent are present in the child.
- Some of the parent's roles may be overridden in the child.
- An overridden role is usually the parent of the new role.

Collaboration Diagram Tour

- Show Classifier Roles and Association Roles, possibly together with extra constraining elements
- Kinds
  - Instance level – Instances and Links
  - Specification level – Roles
- Static Diagrams are used for showing Collaborations explicitly

Collaboration Diagram at Specification Level

/ Teacher : Person	1 tutor	student *	/ Student : Person
position : Text			program : Text

faculty member *

1 lecturer

* participant

faculty 1

given course *

* taken course

: Faculty

: Course
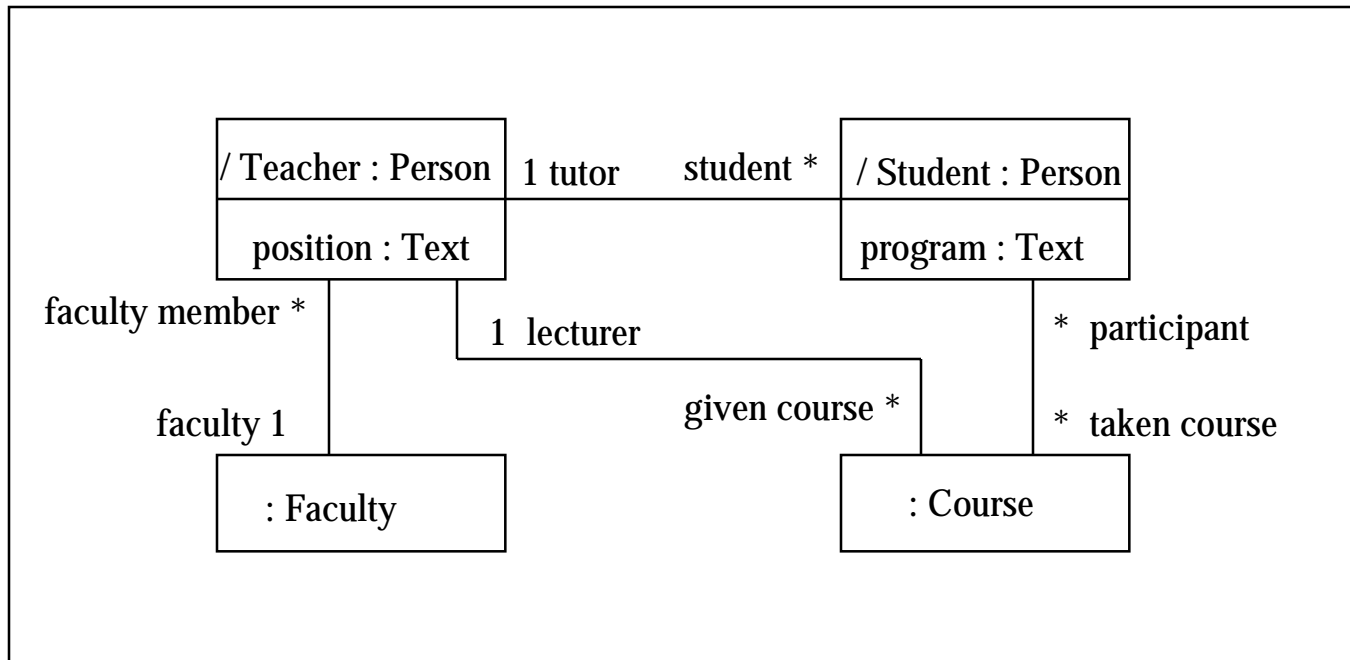
Collaboration Diagram at Instance Level
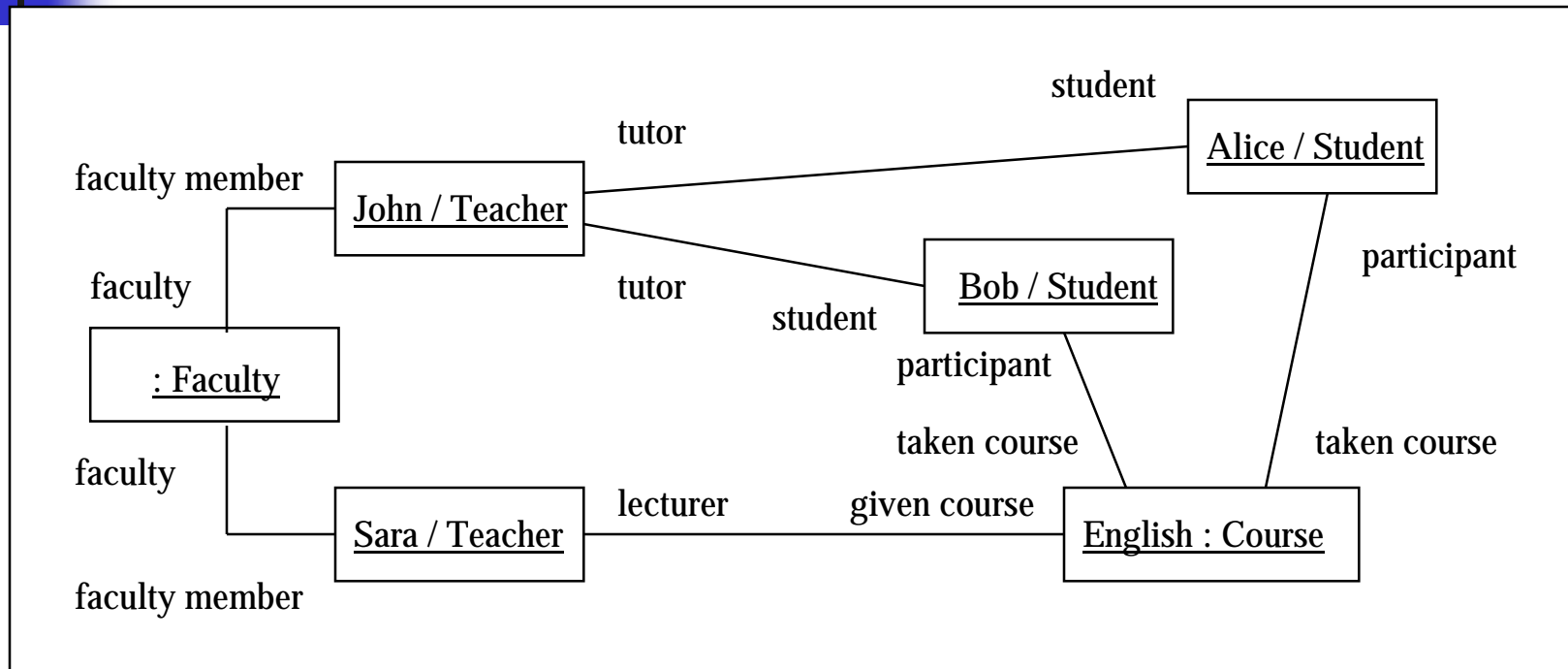
Collaborations including Interactions

Sequence Diagram

x	y	z

a

b

c

Collaboration Diagram

1.1: a
1.2: c

| x | | y |

1.1.1: b

| z |
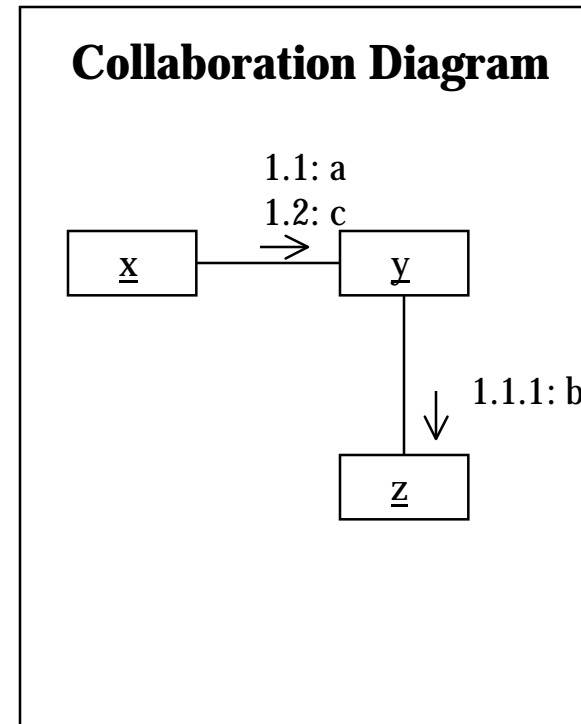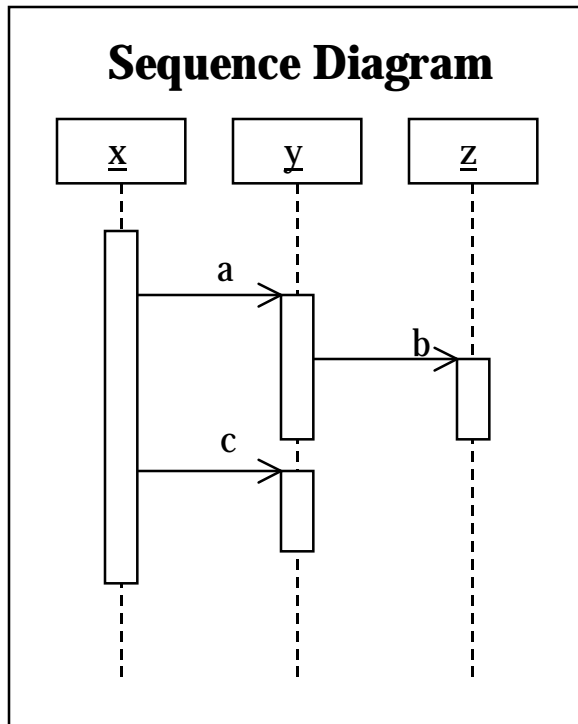
Collaboration Diagram with Constraining Elements

```
┌──────────────────────────────────────────────────────────────┐
│                                                                │
│   ┌─────────────────┐              ┌─────────────────┐        │
│   │   / Generator    │──────────────│  : Printer Device│        │
│   └─────────────────┘              └─────────────────┘        │
│                                             △                  │
│   Constraining Element                      │                  │
│   (A Generalization is not      ──────────▶ │                  │
│   an AssociationRole )              ┌───────┴───────┐          │
│                                     │               │          │
│                      ┌──────────────┐    ┌──────────────┐      │
│                      │ : Laser Printer│    │ : Line Printer │      │
│                      └──────────────┘    └──────────────┘      │
│                                                                │
└──────────────────────────────────────────────────────────────┘
```
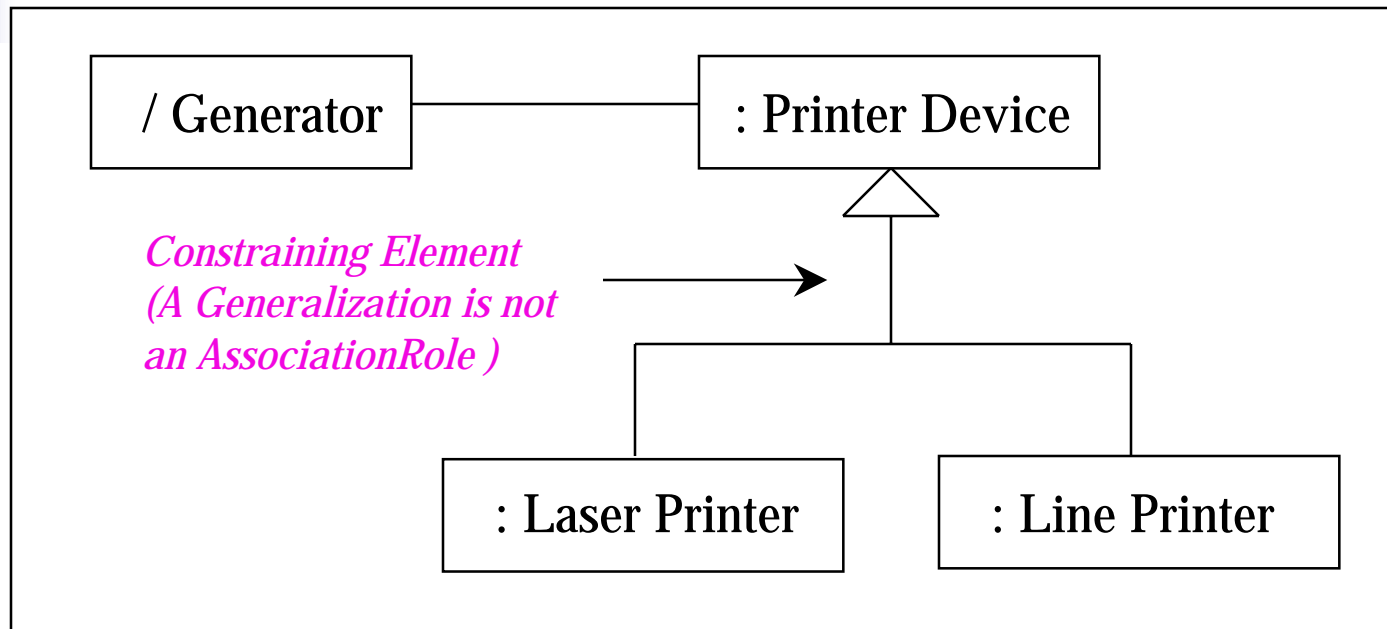
Static Diagram With Collaboration and Classifiers
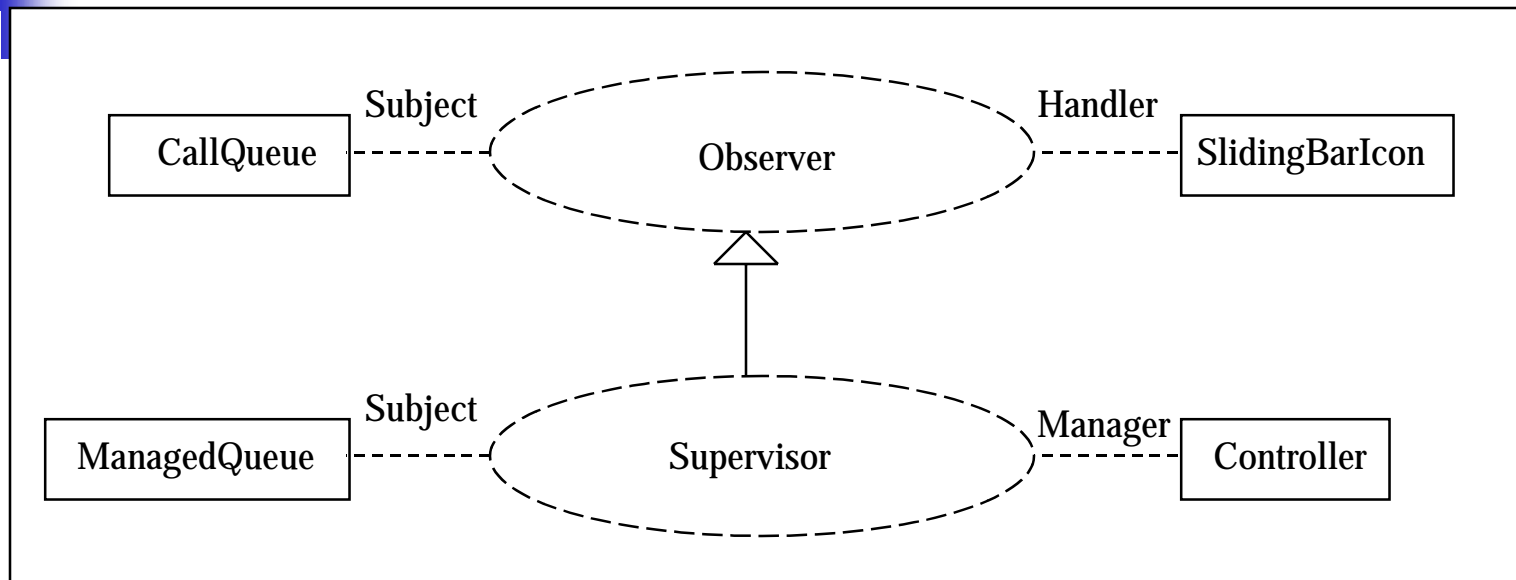
When to Model Collaborations

- Use Collaborations as a tool to find the Classifiers.

- Trace a Use Case / Operation onto Classifiers.

- Map the specification of a Subsystem onto its realization (Tutorial 3).

Collaboration Modeling Tips

- A collaboration should consist of both structure and behavior relevant for the task.

- A role is an abstraction of an instance, it is not a class.

- Look for
  - initiators (external)
  - handlers (active)
  - managed entities (passive)

Example: A Booking System

Use Case Description: Change Flt Itinerary

- **Actors:** traveler, client account db, airline reservation system
- **Preconditions:** Traveler has logged in
- **Basic course:**
  - Traveler selects 'change flight itinerary' option
  - System retrieves traveler's account and flight itinerary from client account database
  - System asks traveler to select itinerary segment she wants to change; traveler selects itinerary segment.
  - System asks traveler for new departure and destination information; traveler provides information.
  - If flights are available then ...
  - ...
  - System displays transaction summary.
- **Alternative course:**
  - If no flights are available then...

Booking System: Change Flt Itinerary Collaboration

4: get customer account

1: change flight itinerary

7: get itinerary

: Traveler — / Flight Itenerary Form

/ DBMS Protocol — : Client Account DBMS

10: present

2: create modifier

9: display

3: get customer account

6: get itinerary

/ Flight Itinerary Modifier

5: create

8: create

/ Account

/ Itinerary

/ ARS Protocol

: Airline Reservation System
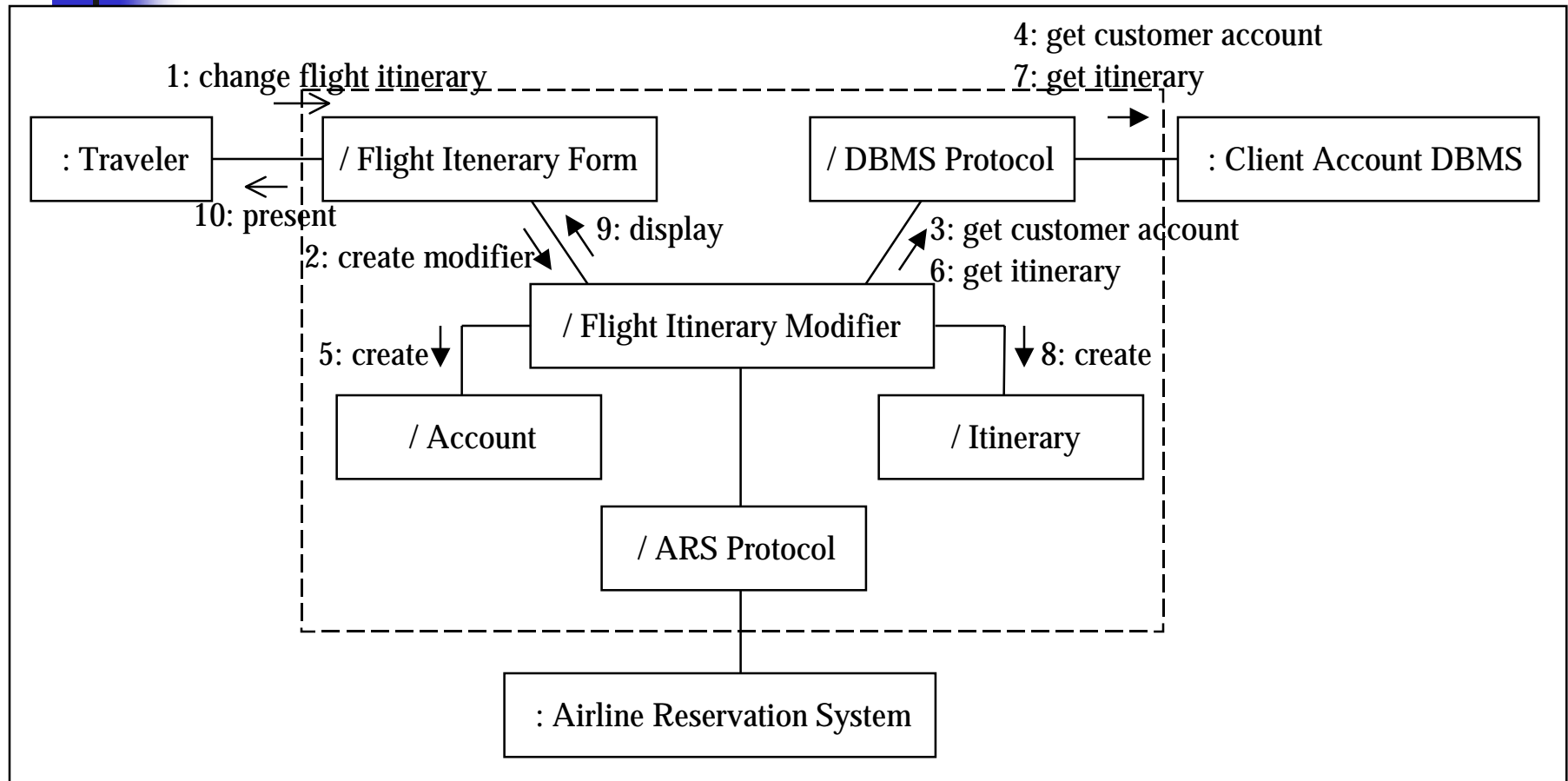
Wrap Up: Interactions & Collaborations

- Instances, Links and Stimuli are used for expressing the dynamics in a model.

- Collaboration is a tool for
  - identification of classifiers
  - specification of the usage of instances
  - expressing a mapping between different levels of abstraction

- Different kinds of diagrams focus on time or on structure

Behavioral Modeling

- Part 1: Interactions and Collaborations
- Part 2: Statecharts
- Part 3: Activity Diagrams

Overview

- Basic State Machine Concepts

- Statecharts and Objects

- Advanced Modeling Concepts

- Case Study

- Wrap Up

Automata

- A machine whose output behavior is not only a direct consequence of the current input, but of some past history of its inputs

- Characterized by an internal **state** which represents this past experience
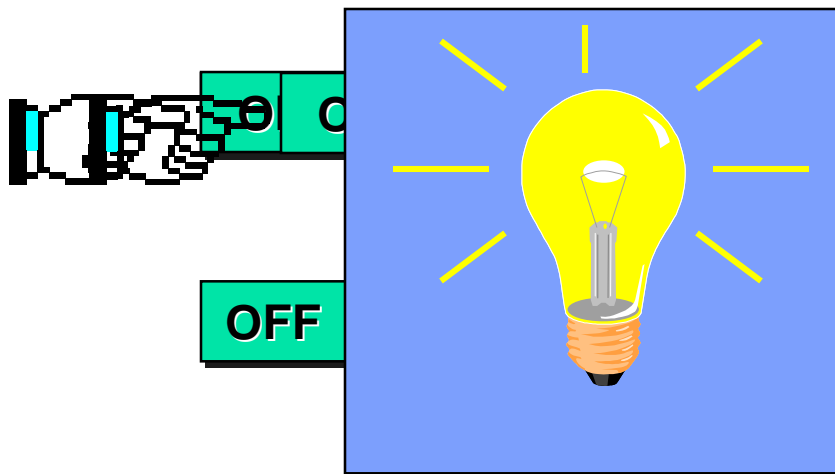
State Machine (Automaton) Diagram

- Graphical rendering of automata behavior
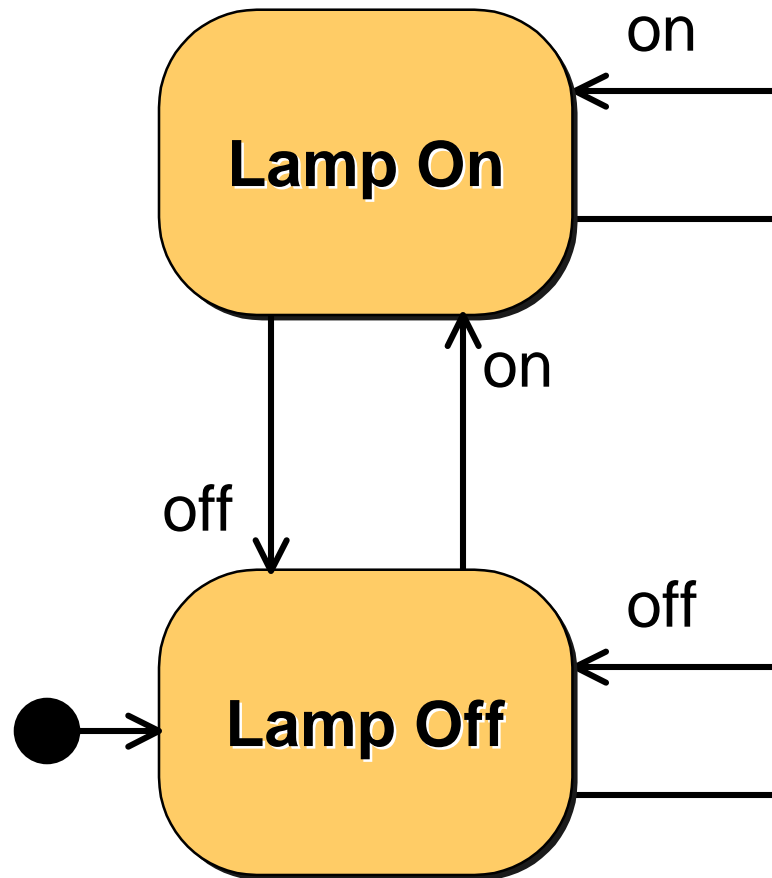
Outputs and Actions

- As the automaton changes state it can generate outputs:

Mealy automaton

```
          on
    ┌──────────┐
 ┌──────────┐  │
 │  Lamp    │◄─┘
 │   On     │
 └──────────┘
  ▲       │
  │ on/print("on")
  │       │ off
  │       ▼
 ┌──────────┐  off
 │  Lamp    │◄─┐
●─►│   Off    │ │
 └──────────┘──┘
```

Moore automaton

```
          on
    ┌──────────┐
 ┌──────────┐  │
 │ Lamp On  │◄─┘
 │print("on")│
 └──────────┘
  ▲       │
  │ on    │ off
  │       ▼
 ┌──────────┐  off
 │  Lamp    │◄─┐
●─►│   Off    │ │
 └──────────┘──┘
```
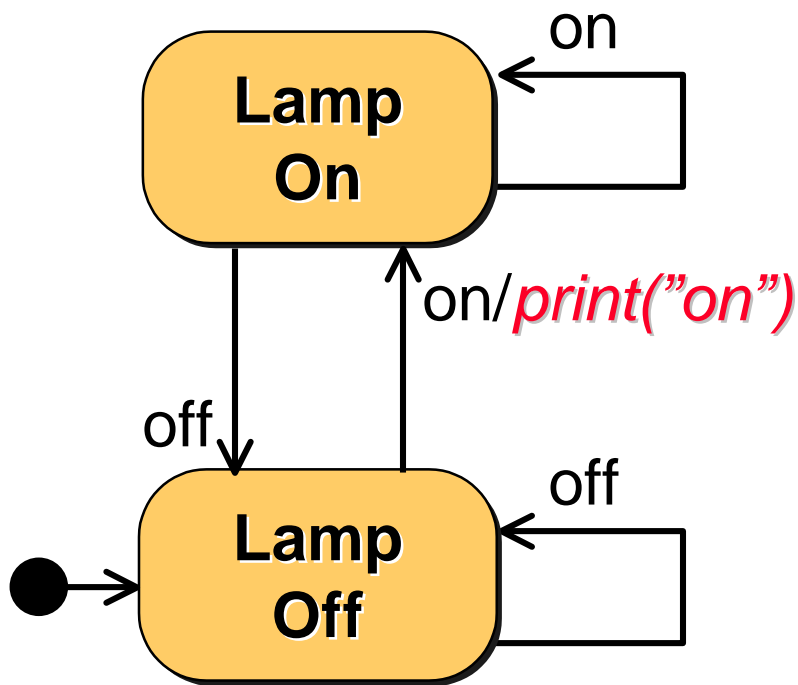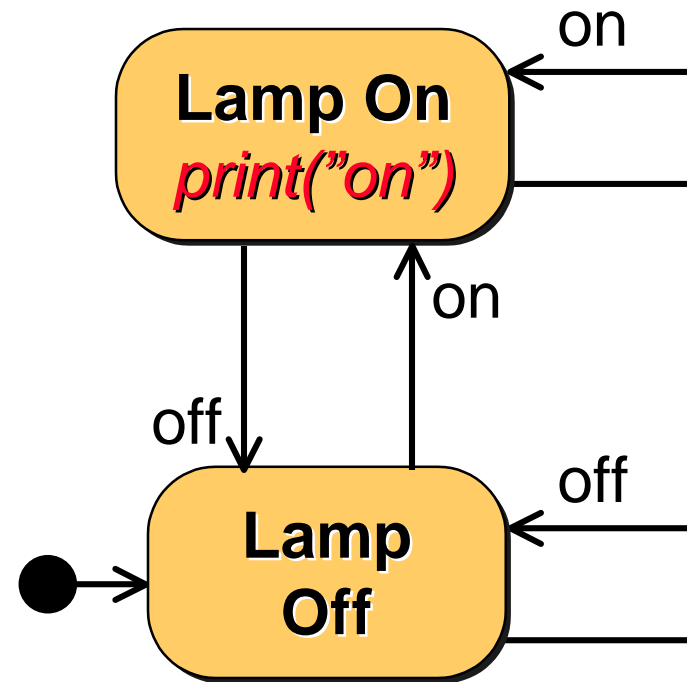
Extended State Machines

- Addition of variables ("**extended state**")

ctr : Integer

Lamp On

on

on/*ctr := ctr + 1*

off

Lamp Off
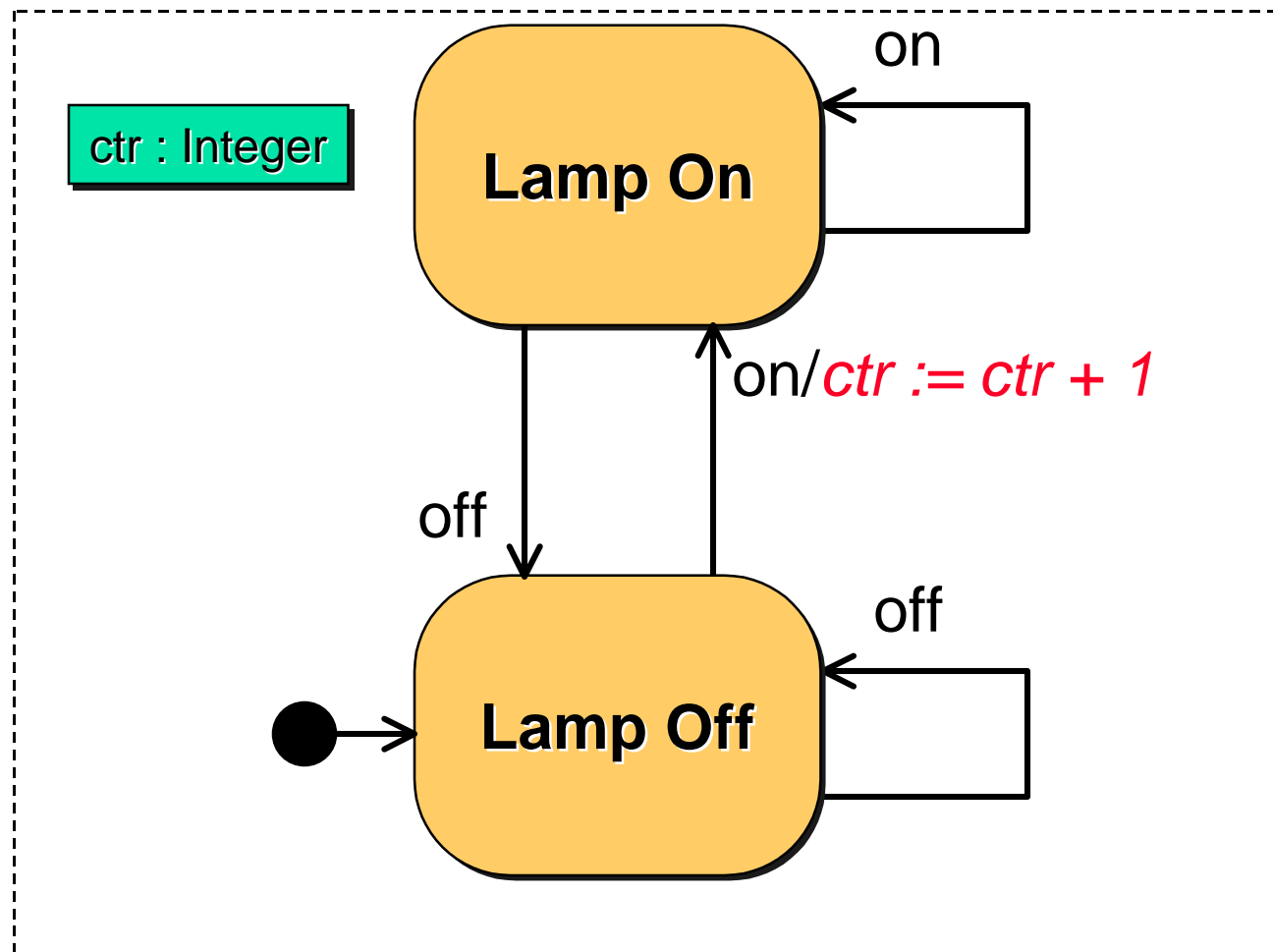
A Bit of Theory

- An extended (Mealy) state machine is defined by:
    - a set of input signals (input alphabet)
    - a set of output signals (output alphabet)
    - a set of states
    - a set of transitions
        - triggering signal
        - action
    - a set of extended state variables
    - an initial state designation
    - a set of final states (if terminating automaton)

Basic UML Statechart Diagram

"top" state

State

Initial pseudostate

top

Trigger

Ready

Transition

stop /ctr := 0

Done

Final state

Action

stop
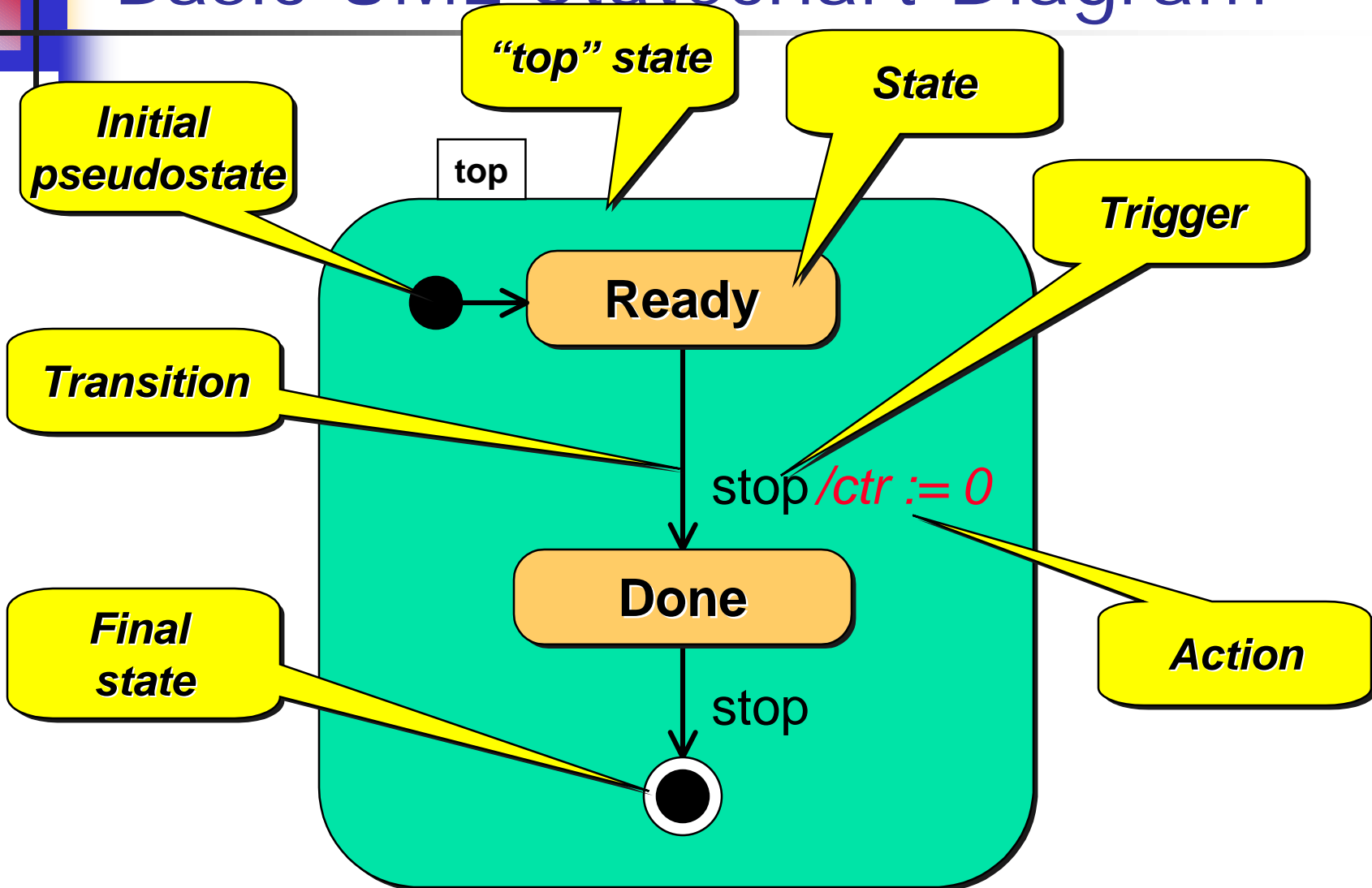
What Kind of Behavior?

- In general, state machines are suitable for describing event-driven, discrete behavior
  - inappropriate for modeling continuous behavior

threshold
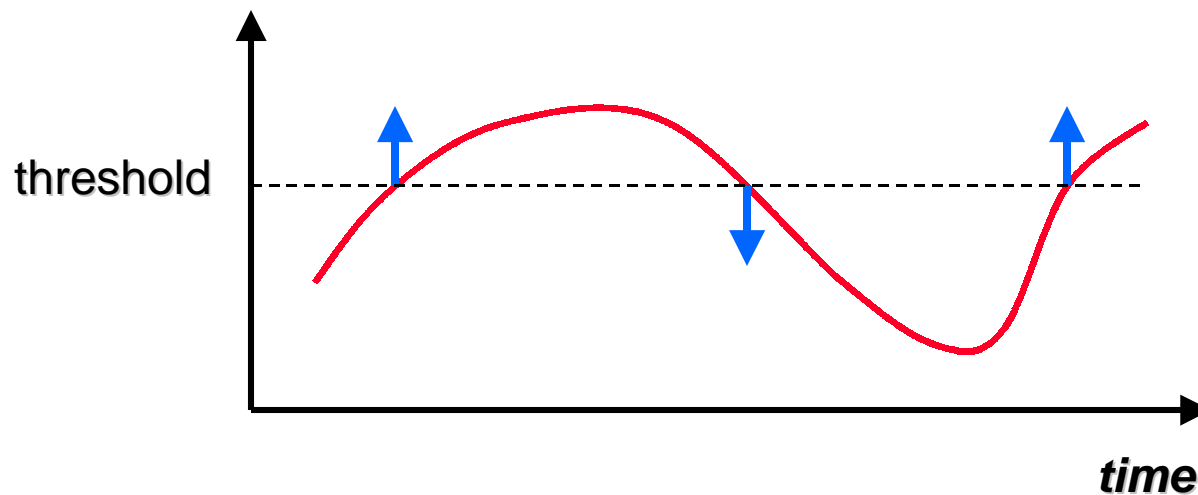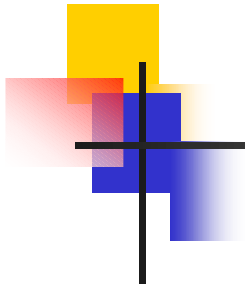
time

Event-Driven Behavior

- **Event** = a <u>type</u> of observable occurrence
  - interactions:
    - synchronous object operation invocation (**call event**)
    - asynchronous signal reception (**signal event**)
  - occurrence of time instants (**time event**)
    - interval expiry
    - calendar/clock time
  - change in value of some entity (**change event**)
- **Event Instance** = an instance of an event (type)
  - occurs at a particular time instant and has *no duration*

The Behavior of What?

- **In principle, anything that manifests event-driven behavior**
  - NB: there is no support currently in UML for modeling continuous behavior
- **In practice:**
  - the behavior of individual objects
  - object interactions
- **The dynamic semantics of UML state machines are currently mainly specified for the case of active objects**

Basic State Machine Concepts

Statecharts and Objects

Advanced Modeling Concepts

Case Study

Wrap Up

Object Behavior - General Model

- Simple **server** model:

Handling depends on specific request type

```
void:offHook ();
  {busy = true;
   obj.reqDialtone();
   ...
   };
```

Initialize Object

Wait for Request

Handle Request

Terminate Object
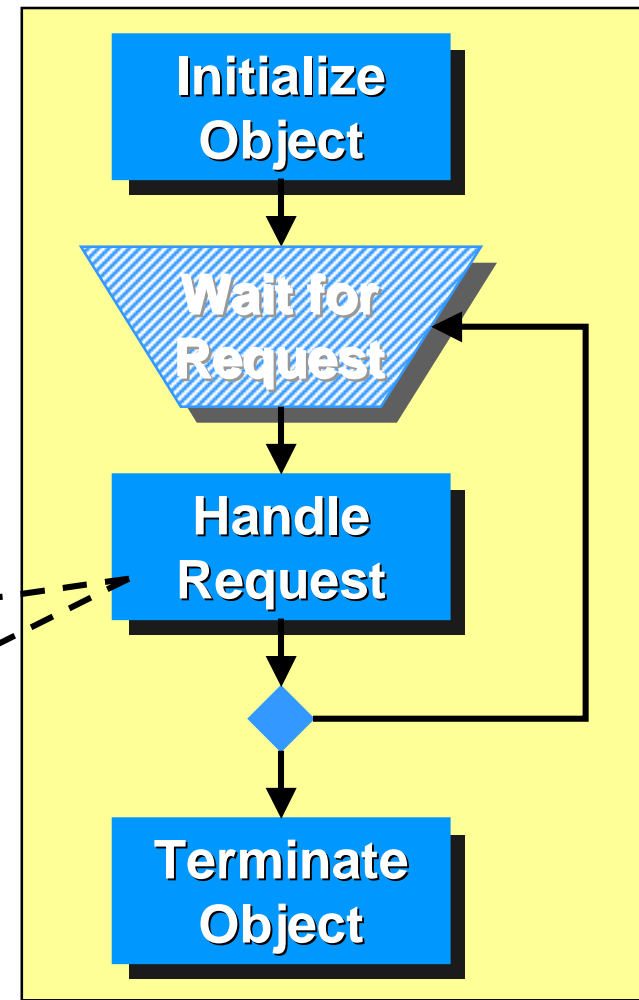
Object Behavior and State Machines

- Direct mapping:

Initialize Object

Wait for Event

Handle Event

Terminate Object

Lamp On

Lamp Off

on

on/print("on")

off

off

off

stop
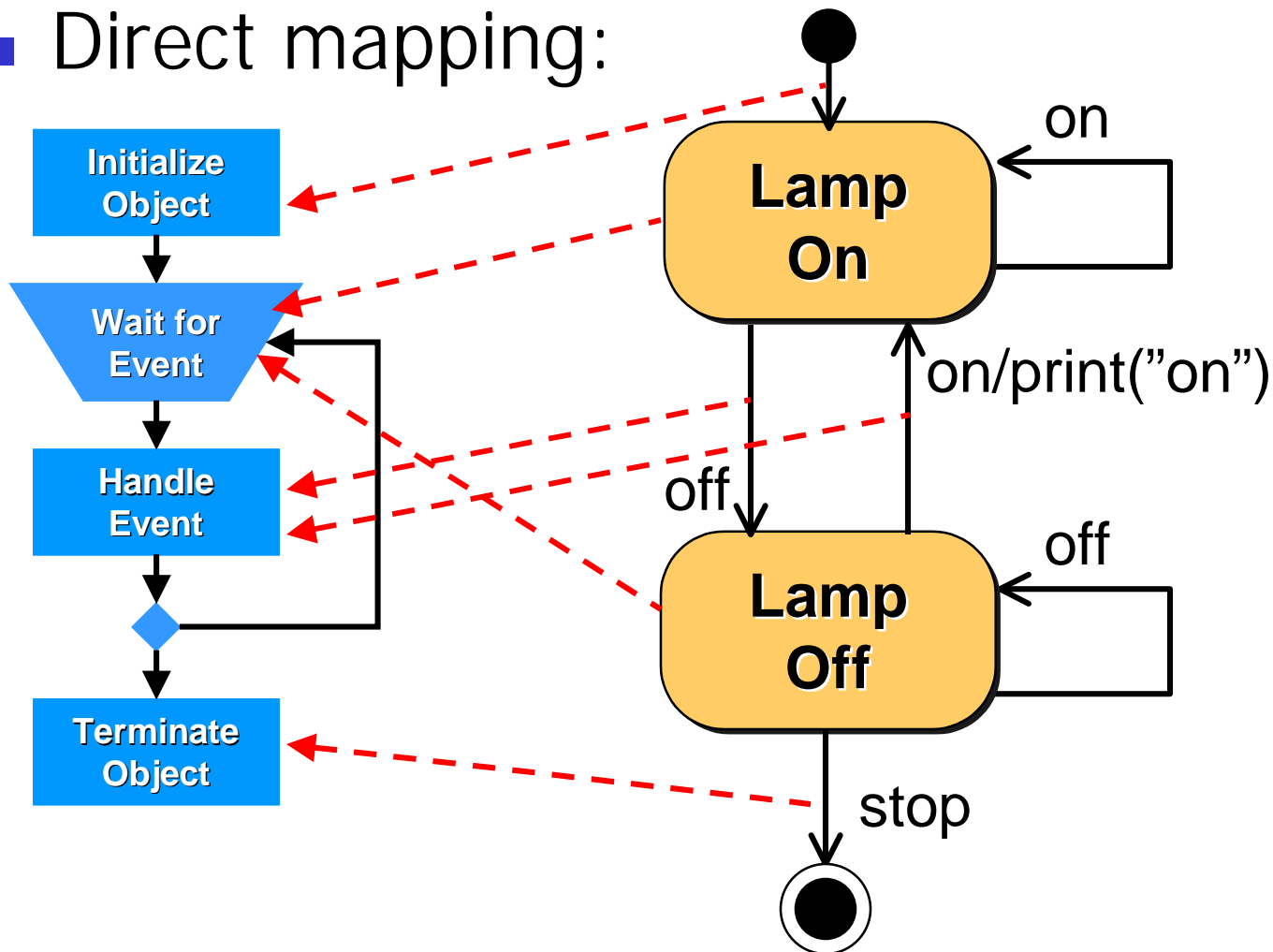
Object and Threads

- **Passive objects:** depend on external power (thread of execution)
- **Active objects:** self-powered (own thread of execution)
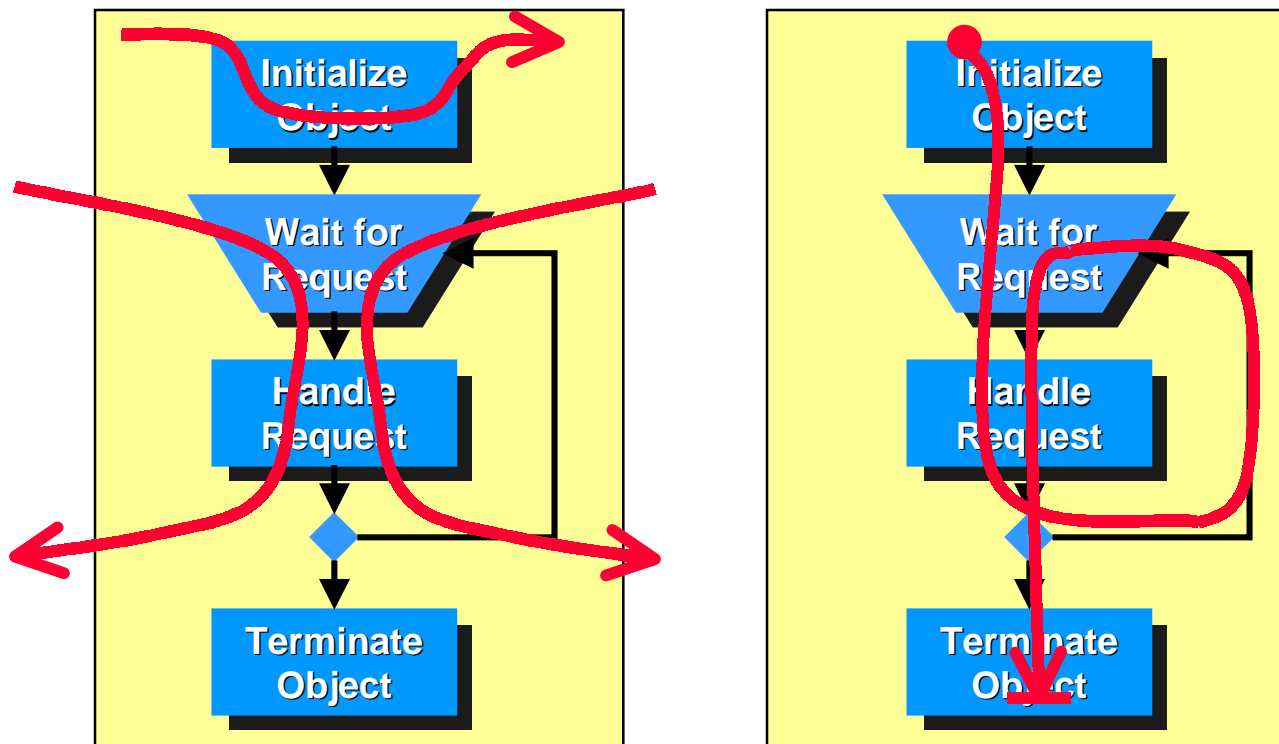
Passive Objects: Dynamic Semantics
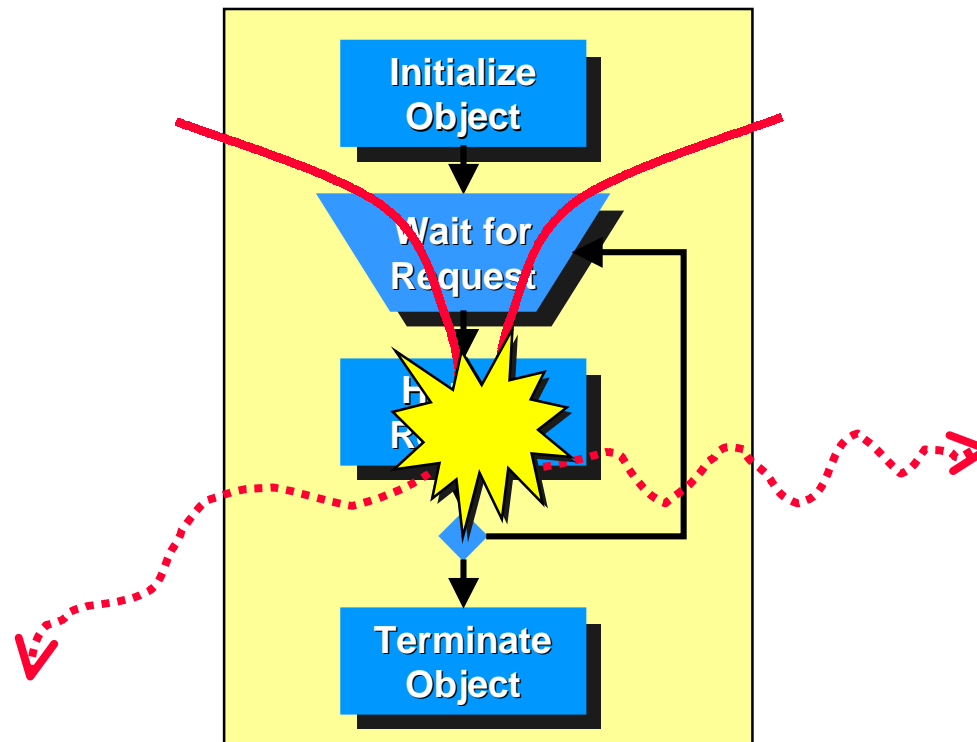
- Encapsulation does not protect the object from concurrency conflicts!
- Explicit synchronization is still required

Active Objects and State Machines

- Objects that encapsulate own thread of execution
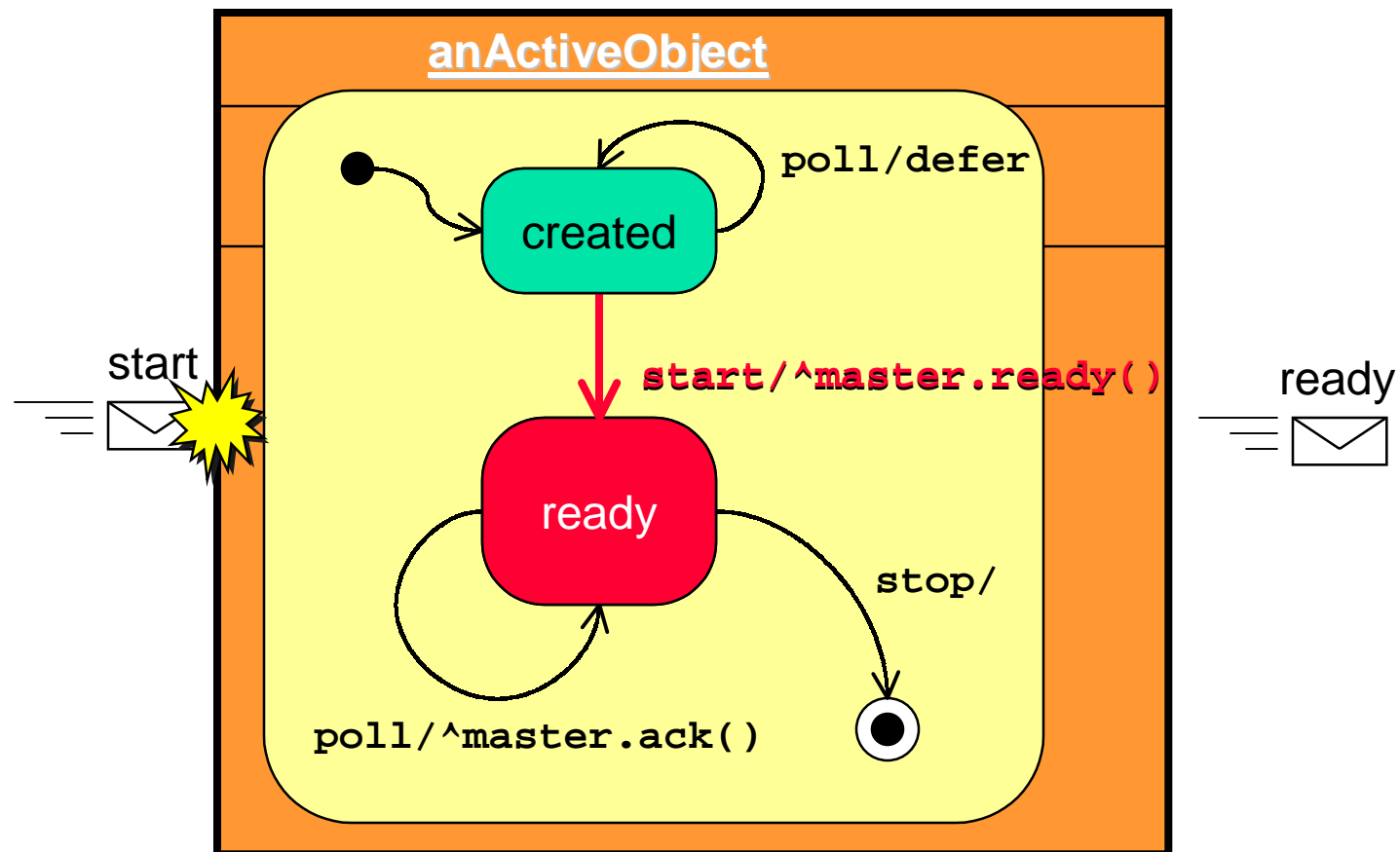
Active Objects: Dynamic Semantics
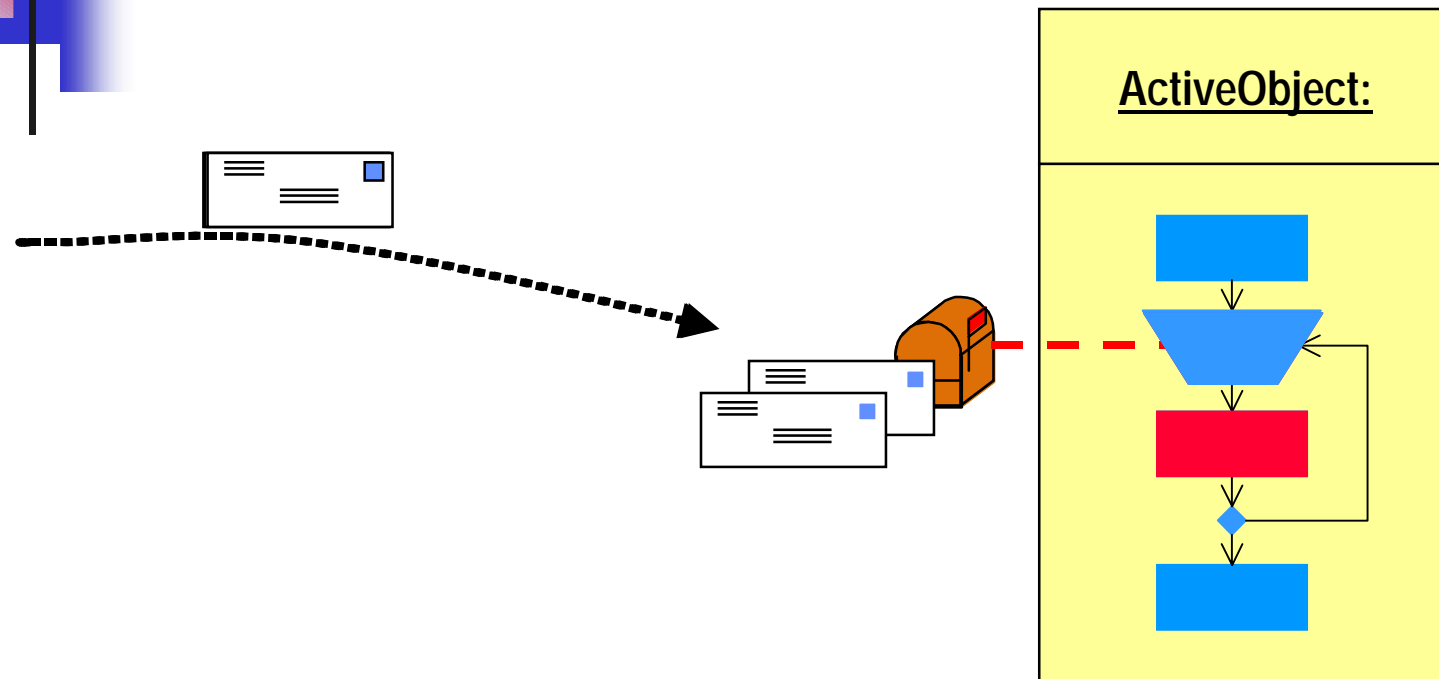
ActiveObject:

Run-to-completion model:
- serialized event handling
- eliminates internal concurrency
- minimal context switching overhead

The Run-to-Completion Model

- A high priority event for (another) active object *will* preempt an active object that is handling a low-priority event
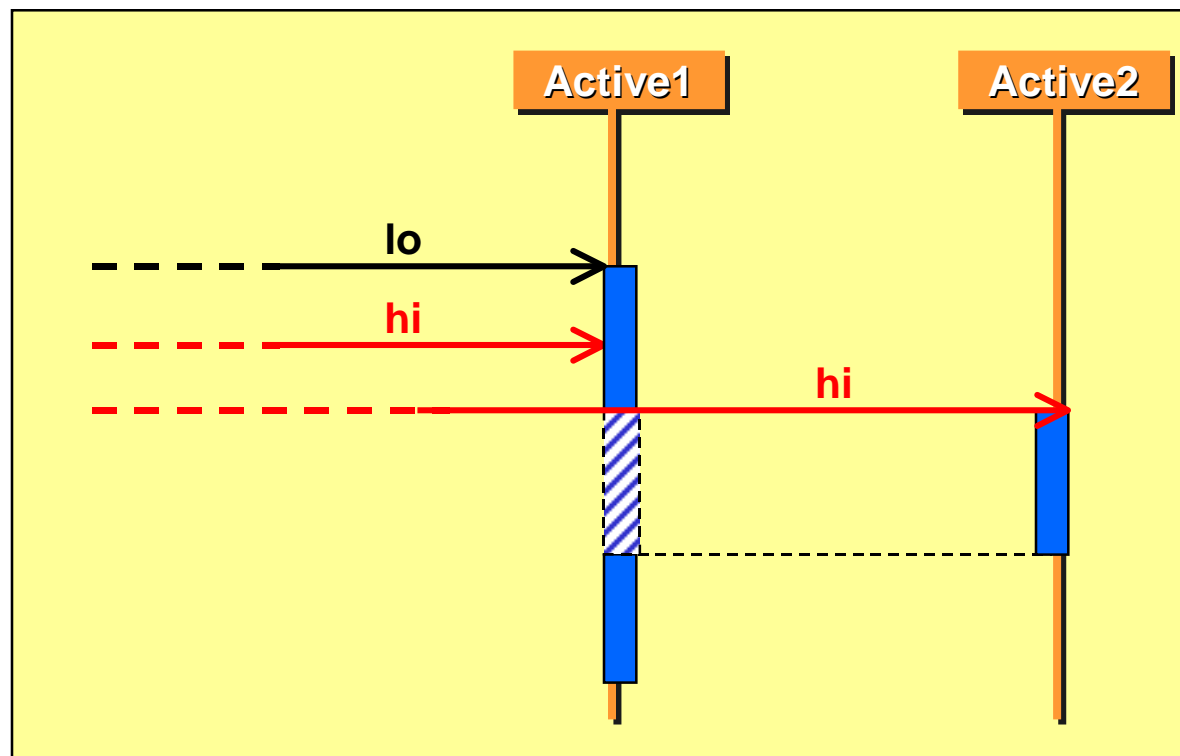
Basic State Machine Concepts

Statecharts and Objects

Advanced Modeling Concepts

Case Study

Wrap Up

State Entry and Exit Actions

- A dynamic assertion mechanism

```
        LampOn
  entry/lamp.on();         e2
  exit/lamp.off();
        e1
```
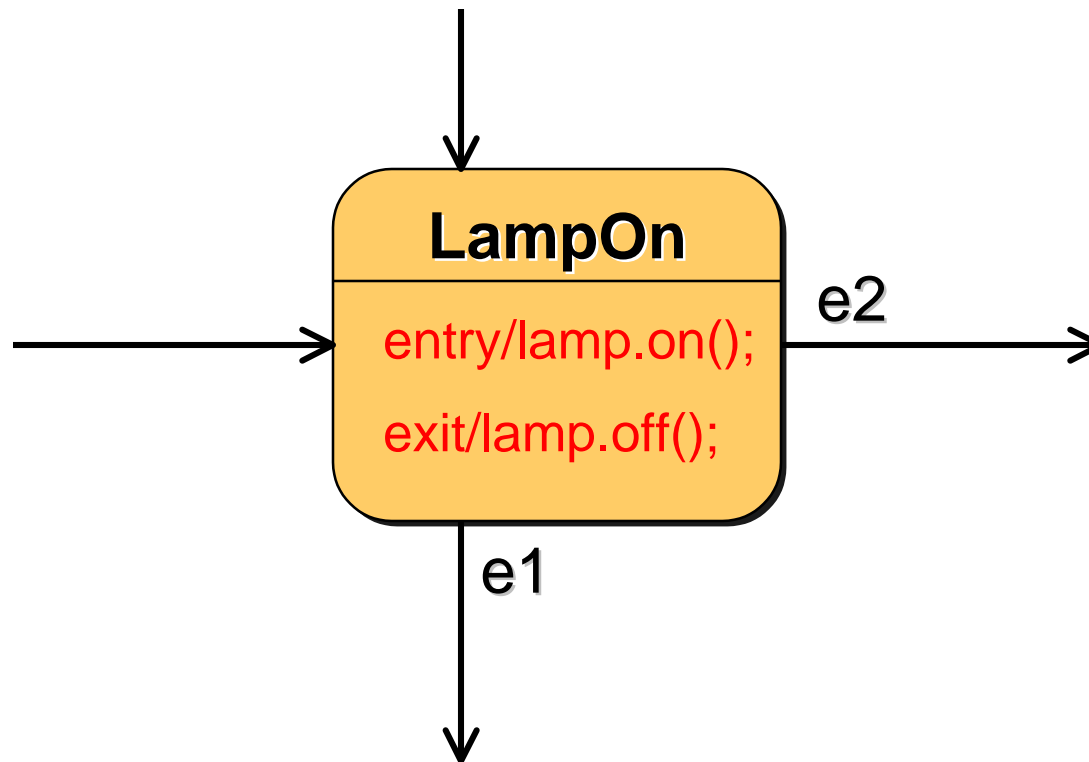
Order of Actions: Simple Case

- Exit actions prefix transition actions
- Entry action postfix transition actions

LampOn

entry/lamp.on();

exit/printf("exiting");

off/printf("to off");

LampOff

entry/lamp.off();

exit/printf("exiting");

off/printf("needless");
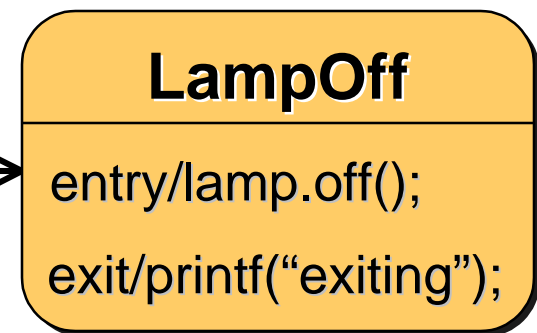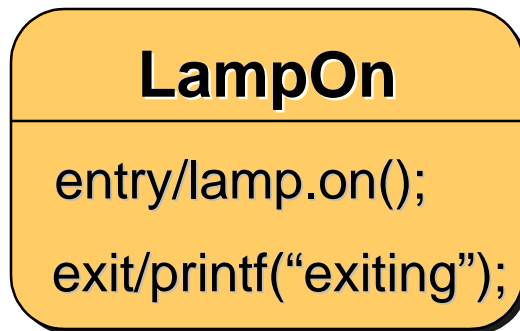
Resulting action sequence:
```
printf("exiting");
printf("to off");
lamp.off();
```

```
printf("exiting");
printf("needless");
lamp.off();
```

Internal Transitions

- Self-transitions that bypass entry and exit actions

Internal transition
triggered by
an "off" event

LampOff

entry/lamp.off();

exit/printf("exiting");

off/null;

State ("Do") Activities

- Forks a concurrent thread that executes until:
  - the action completes or
  - the state is exited through an outgoing transition

"do" activity

Error

entry/printf("error!")

do/while (true) alarm.ring();
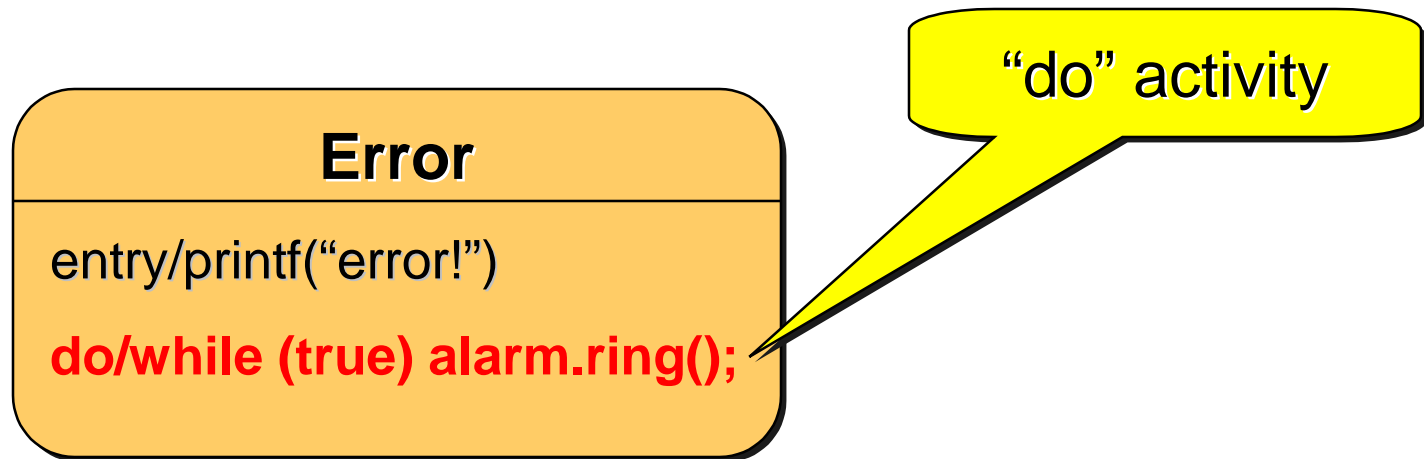
Guards

Conditional execution of transitions
- guards (Boolean predicates) must be side-effect free

bid [value < 100] /reject

Selling

bid [value >= 200] /sell → **Happy**

bid [(value >= 100) & (value < 200)] /sell → **Unhappy**
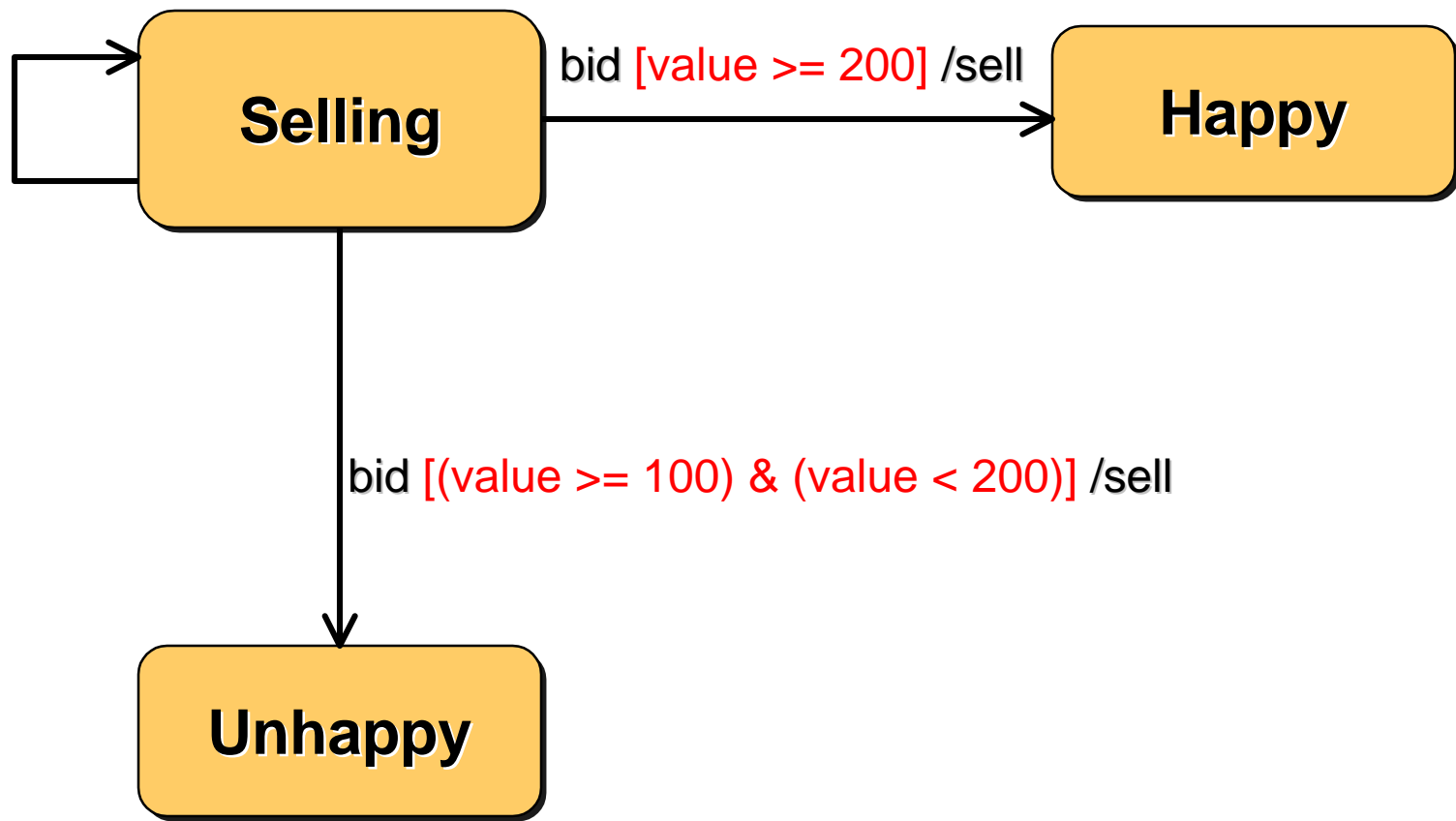
Static Conditional Branching

- Merely a graphical shortcut for convenient rendering of decision trees

Selling

Happy

bid

[value >= 200] /sell

[value < 100] /reject

[(value >= 100) & (value < 200)] /sell

Unhappy
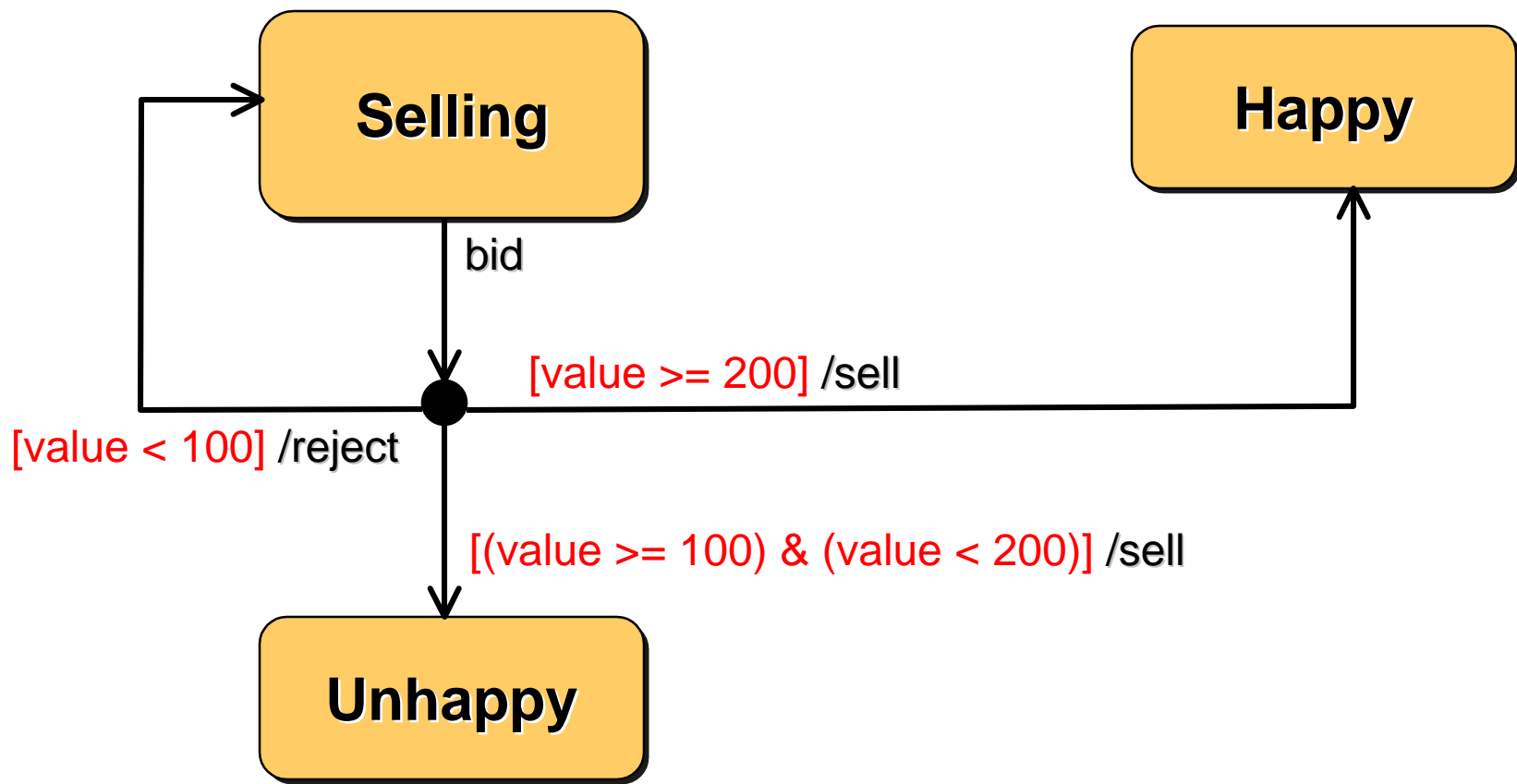
Dynamic Conditional Branching

- **Choice** pseudostate: guards are evaluated at the instant when the decision point is reached

Selling

Happy

bid /gain := calculatePotentialGain(value)

[gain >= 200] /sell

[gain < 100] /reject

Dynamic choicepoint

[(gain >= 100) & (gain < 200)] /sell

Unhappy
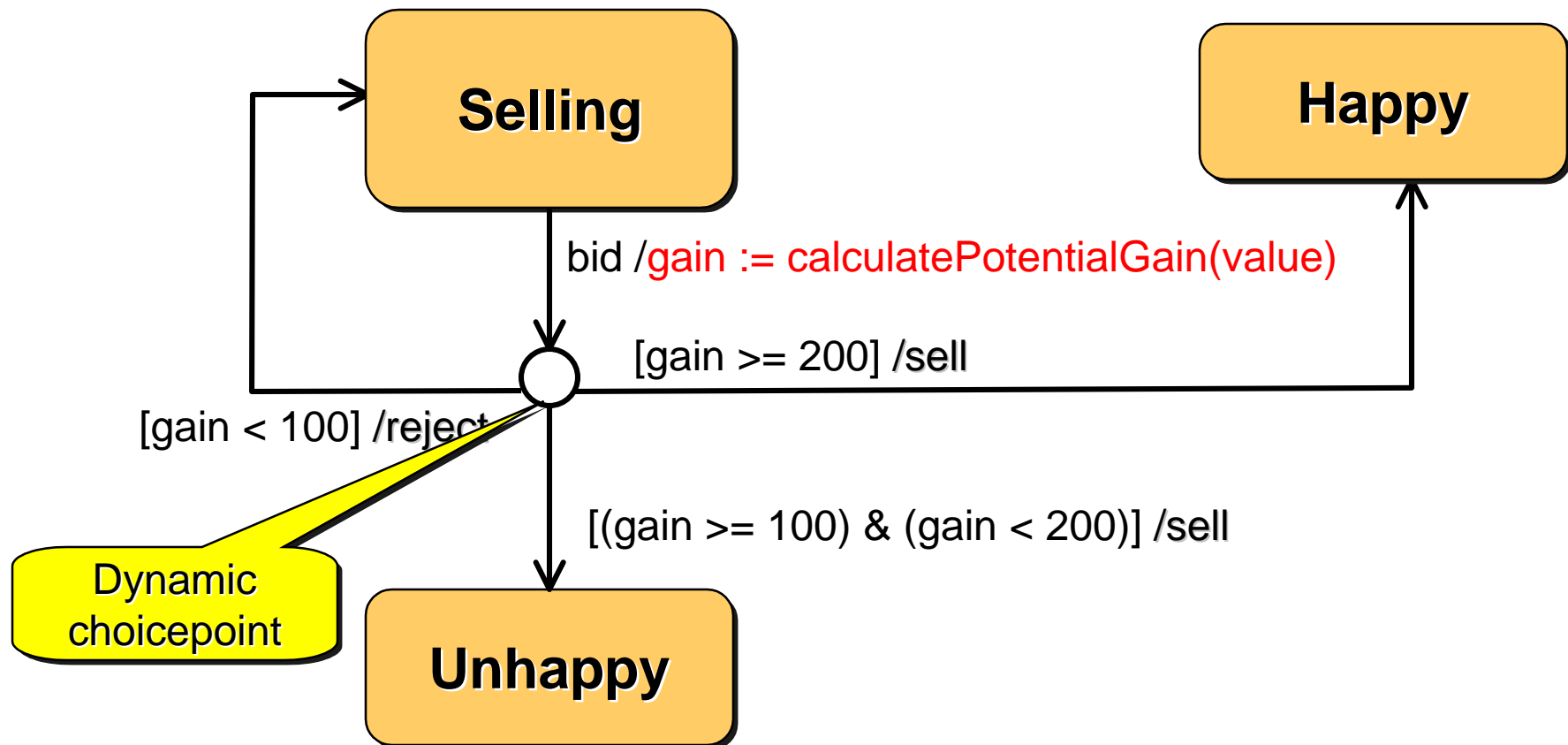
Hierarchical State Machines

- Graduated attack on complexity
  - states decomposed into state machines

LampOff

entry/lamp.off()

flash/

LampFlashing

FlashOn

entry/lamp.on()

off/

on/

on/

1sec/

1sec/

on/

FlashOff

entry/lamp.off()

LampOn

entry/lamp.on()
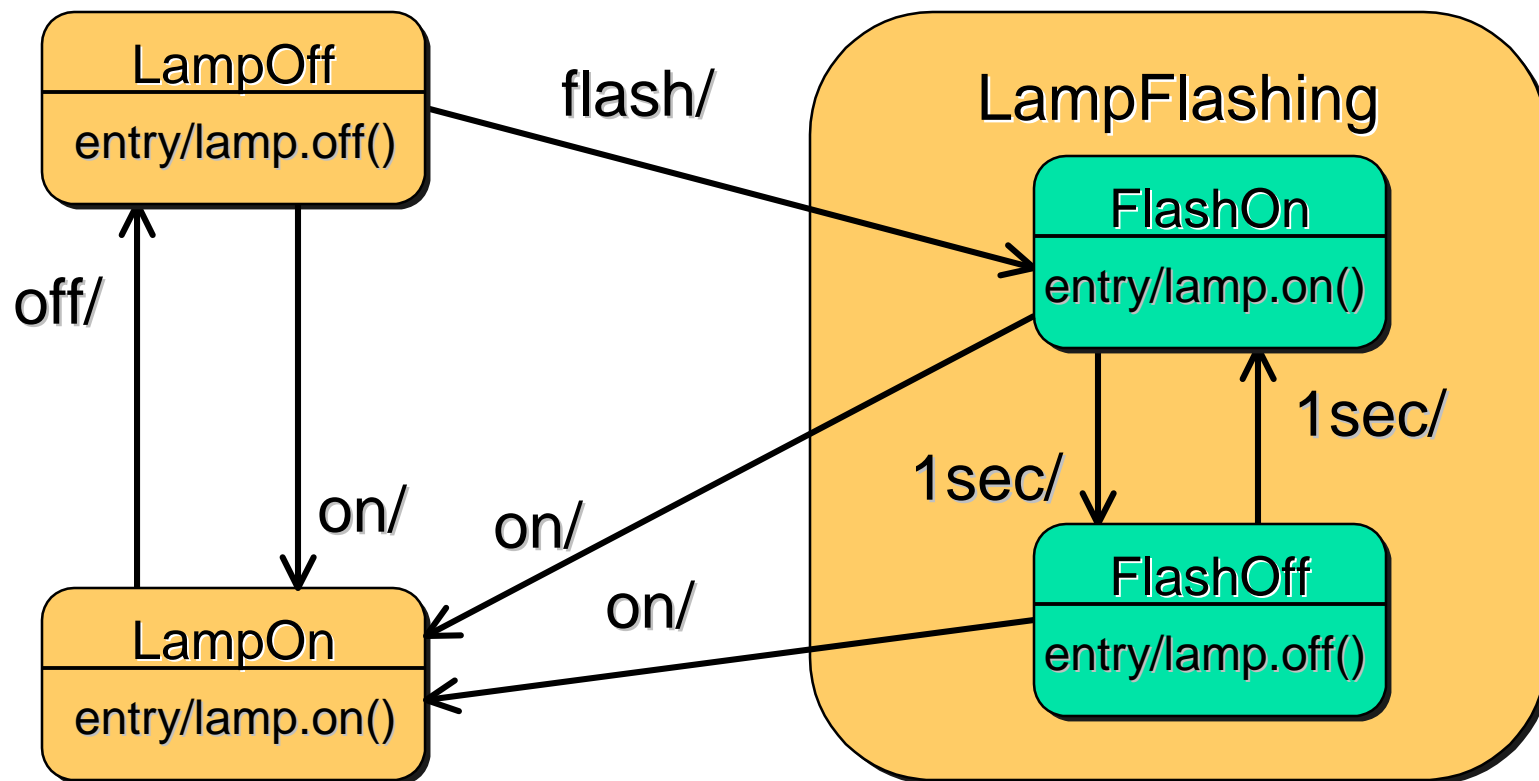
"Stub" Notation

- Notational shortcut: no semantic significance

LampOff
entry/lamp.off()

flash/

LampFlashing

FlashOn

off/

on/

on/

on/

FlashOff

LampOn
entry/lamp.on()
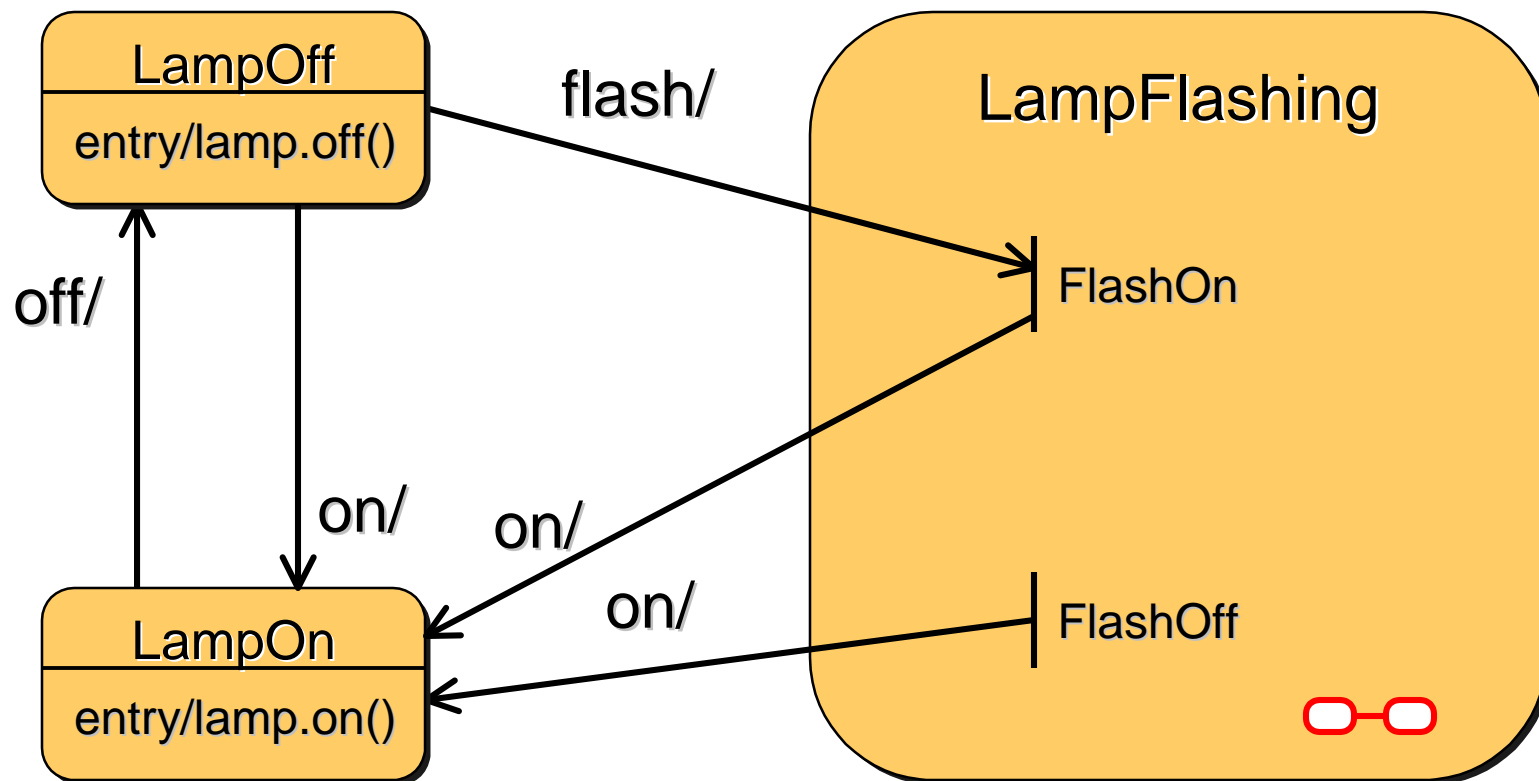
Group Transitions

- Higher-level transit... **Default transition to the initial pseudostate**

```
  LampOff
  ─────────────
  entry/lamp.off()          flash/

  off/        on/

  LampOn                         on/
  ─────────────
  entry/lamp.on()
```

LampFlashing

```
   FlashOn
   ─────────────
   entry/lamp.on()

        1sec/        1sec/

   FlashOff
   ─────────────
   entry/lamp.off()
```

Group transition
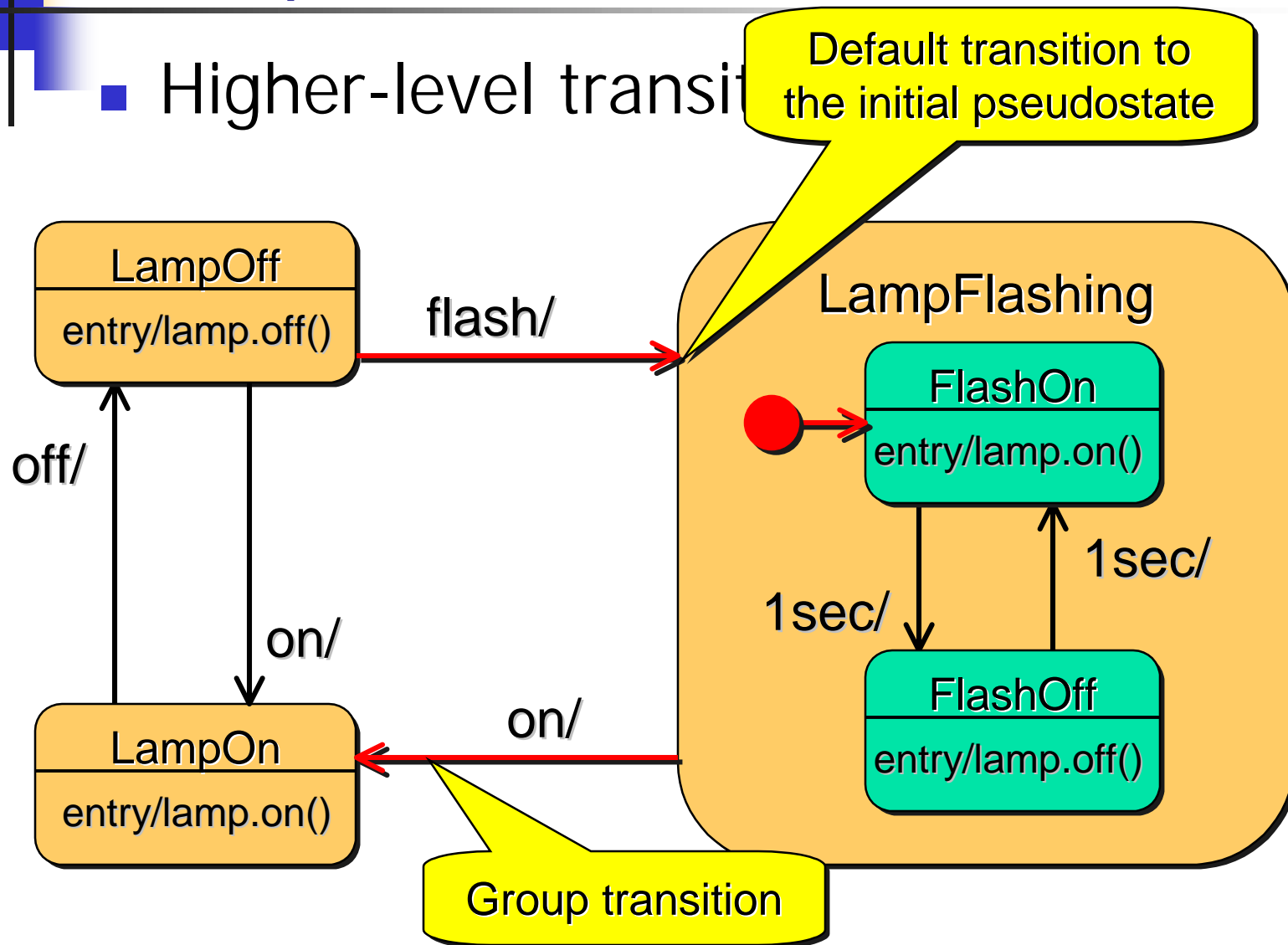
Completion Transitions

- Triggered by a **completion event**
  - generated automatically when an immediately nested state machine terminates

Committing

Phase1

Phase2

completion transition (no trigger)

CommitDone
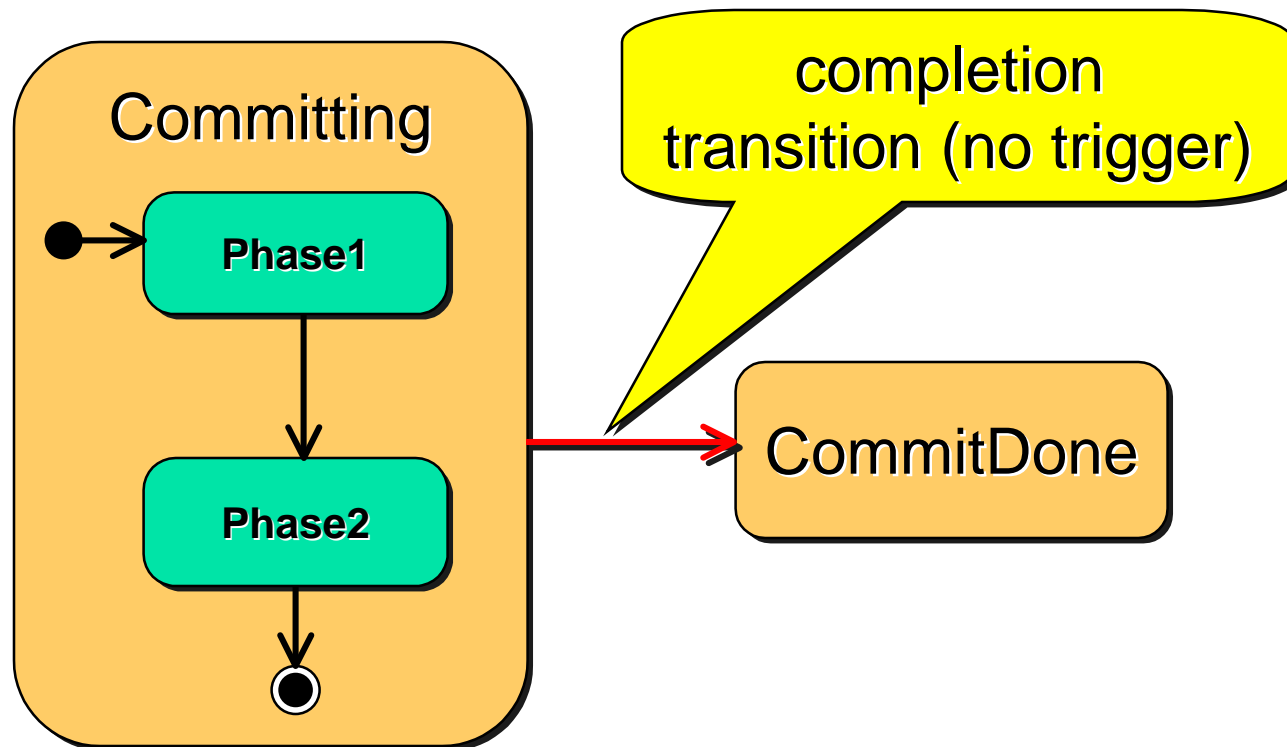
Triggering Rules

- Two or more transitions may have the same event trigger
  - innermost transition takes precedence
  - if no transition is triggered, event is discarded

LampFlashing

FlashOn

on/

off/

FlashOff

on/
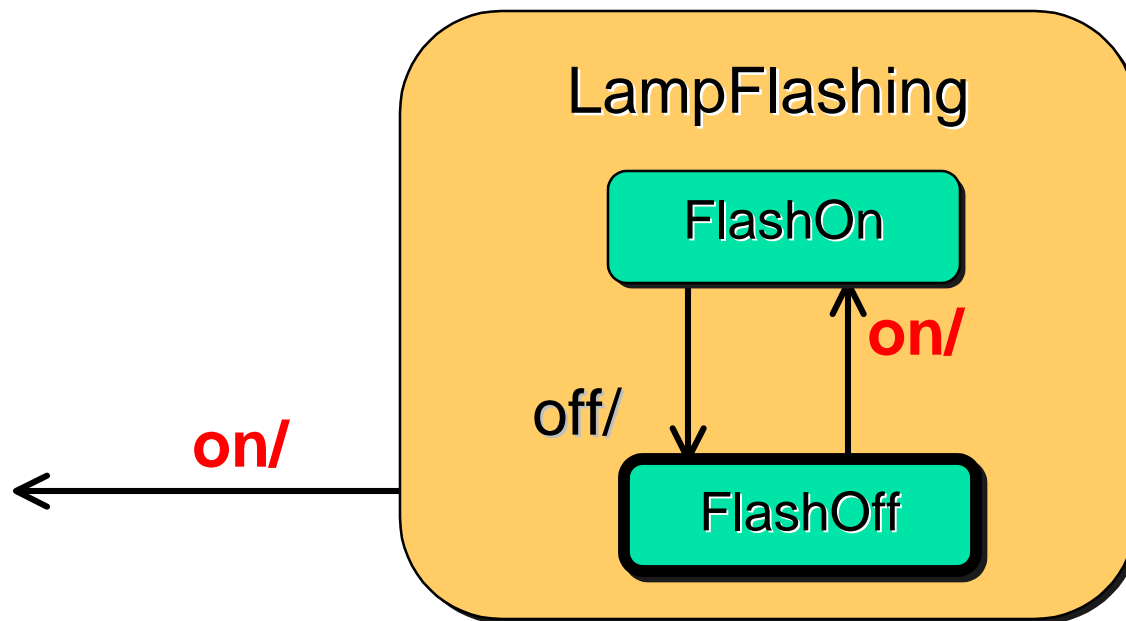
Order of Actions: Complex Case

- Same approach as for the simple case

S1
exit/exS1

S11
exit/exS11

E/actE →

S2
entry/enS2

initS2

S21
entry/enS21
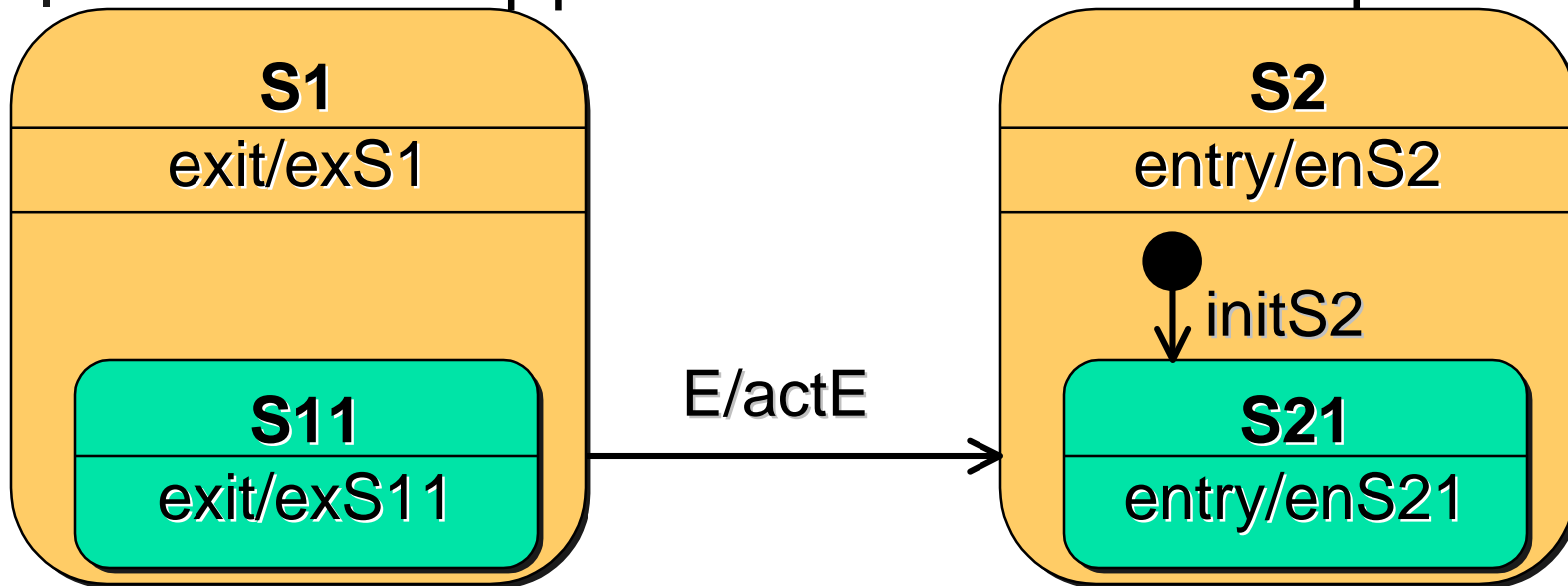
Actions execution sequence:

exS11 ⇨ exS1 ⇨ actE ⇨ enS2 ⇨ initS2 ⇨ enS21

History

- Return to a previously visited hierarchical state
  - deep and shallow history options
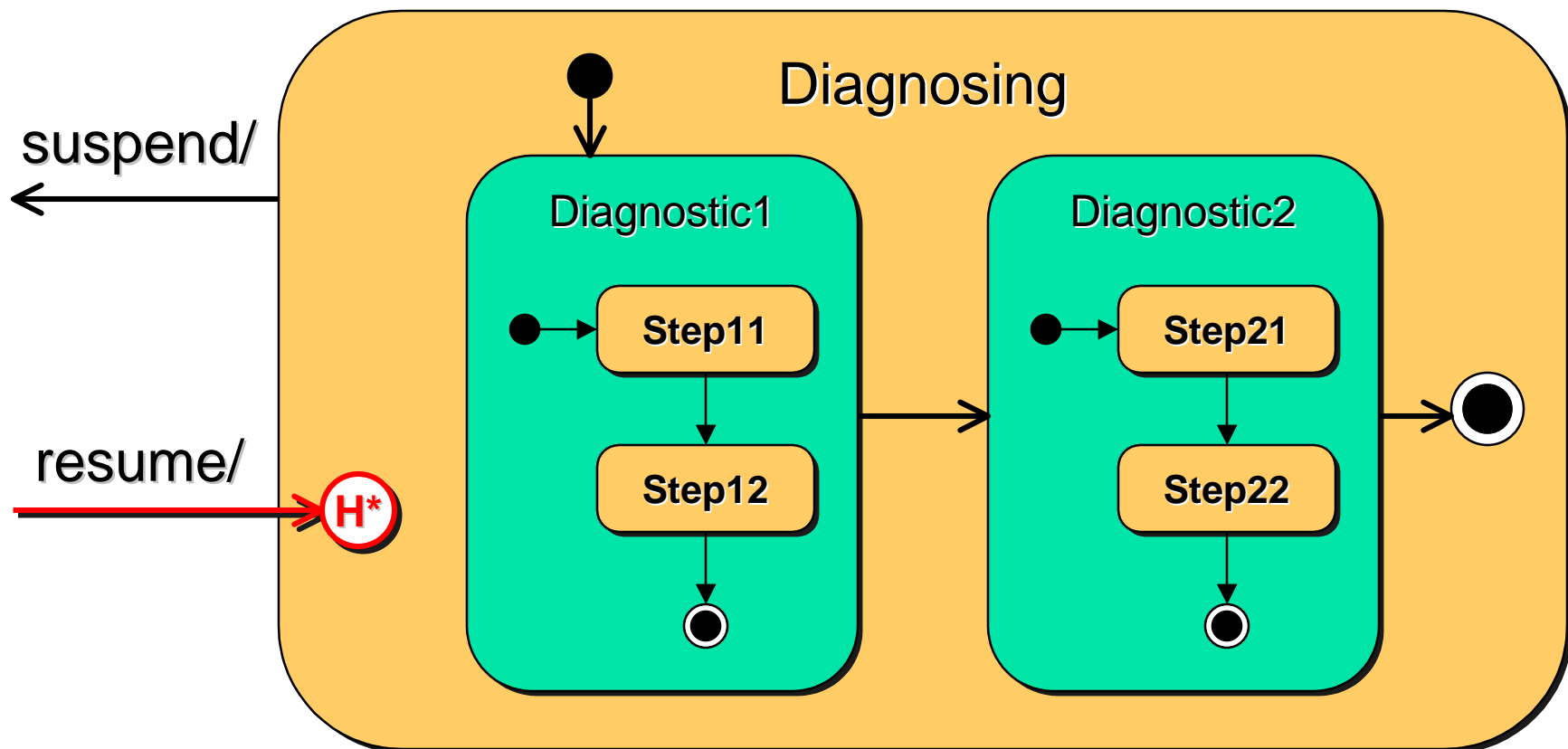
Orthogonality

- Multiple simultaneous perspectives on the same entity

age

Child
Adult
Retiree

financialStatus

Poor
Rich
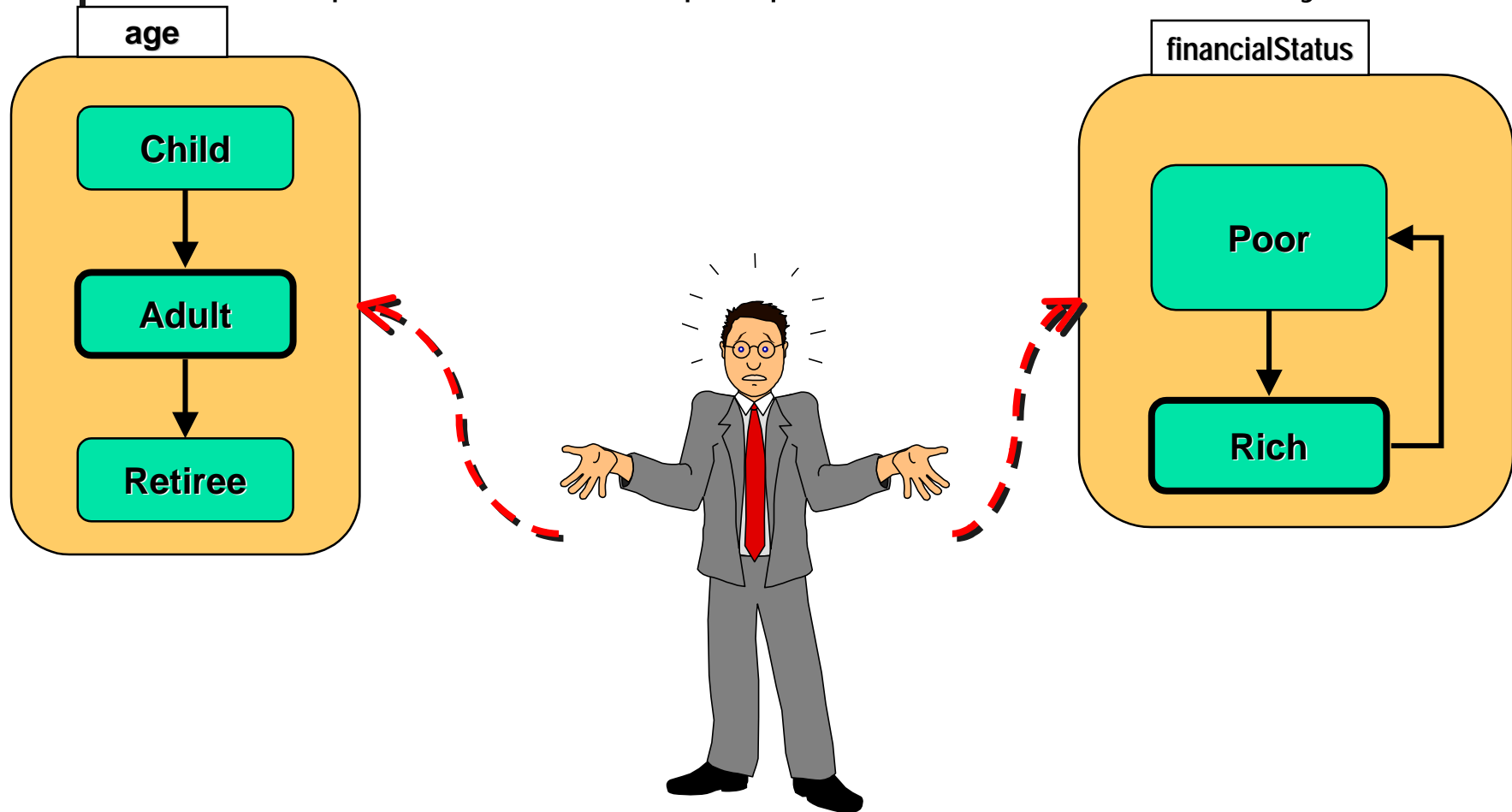
Orthogonal Regions

- Combine multiple simultaneous descriptions

age

Child
→
Adult
→
Retiree

financialStatus

Poor
→
Rich

age | **financialStatus**

Child
→
Adult
→
Retiree

Poor
→
Rich
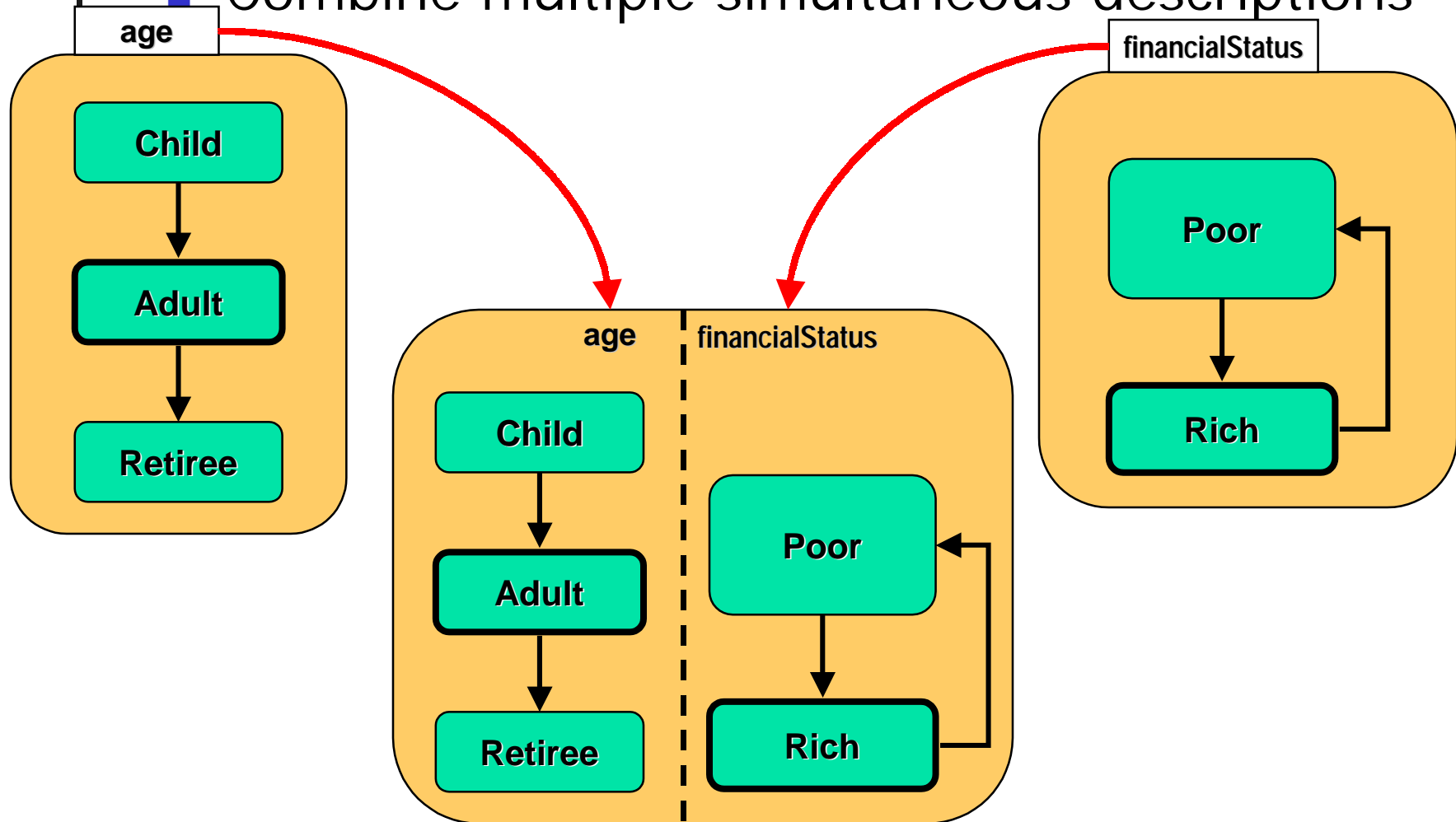
Orthogonal Regions - Semantics

- All mutually orthogonal regions detect the same events and respond to them "simultaneously"
  - usually reduces to interleaving of some kind
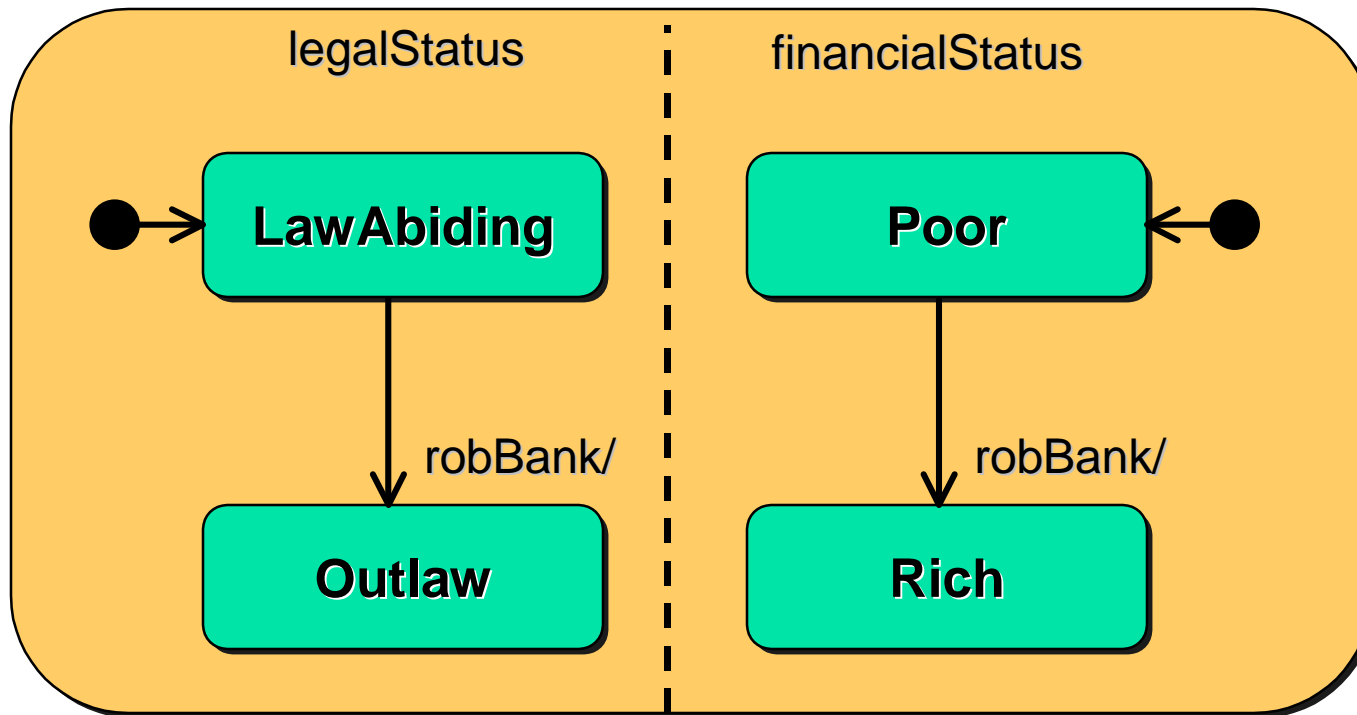
Interactions Between Regions

- Typically through shared variables or awareness of other regions' state changes

sane : Boolean

flying : Boolean

Catch22

sanityStatus

Crazy
entry/sane := false;

(flying)/

request
Grounding/

Sane
entry/sane := true;

flightStatus

Flying
entry/flying := true;

(sane)/

(~sane)/

Grounded
entry/flying := false;
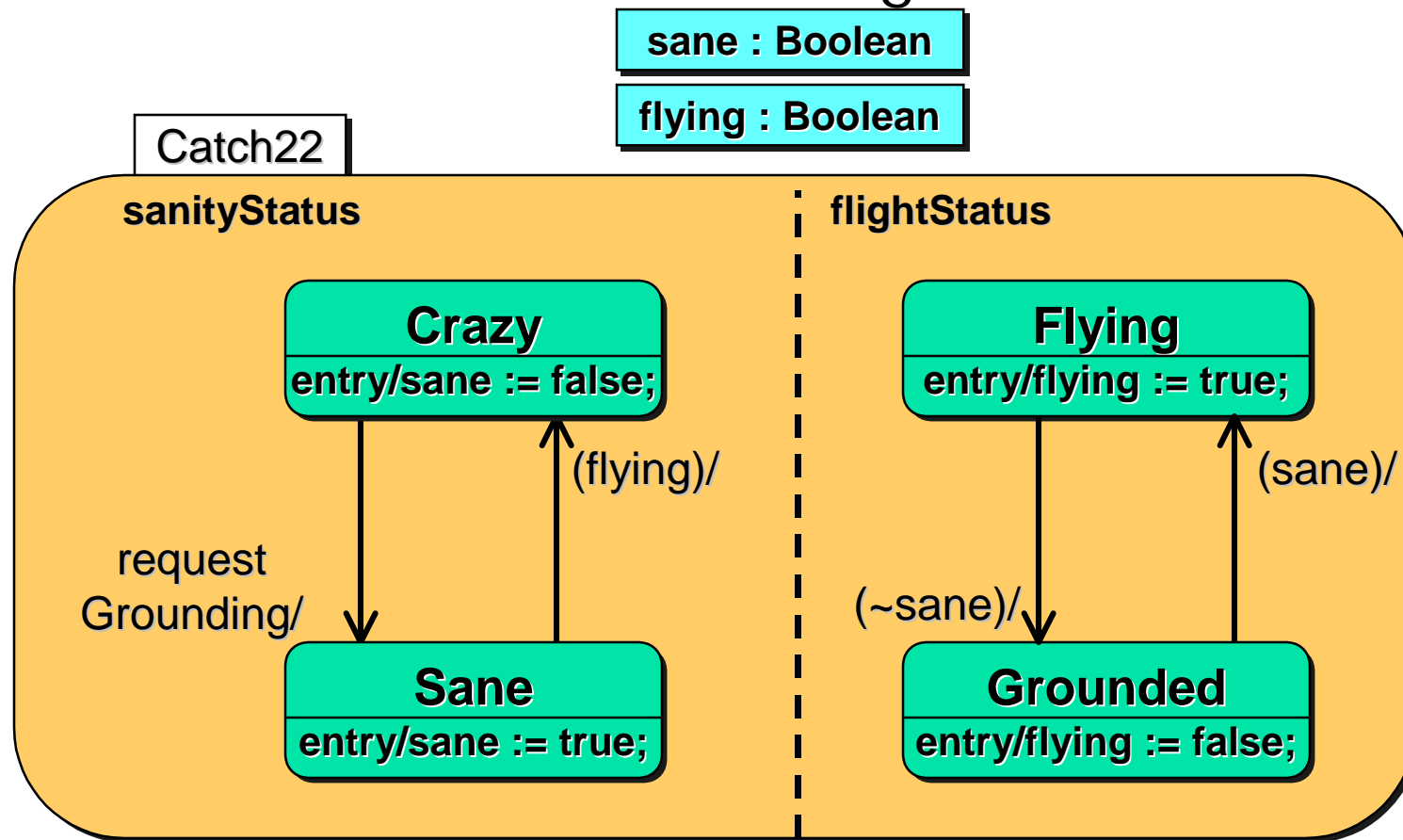
Transition Forks and Joins

- For transitions into/out of orthogonal regions:
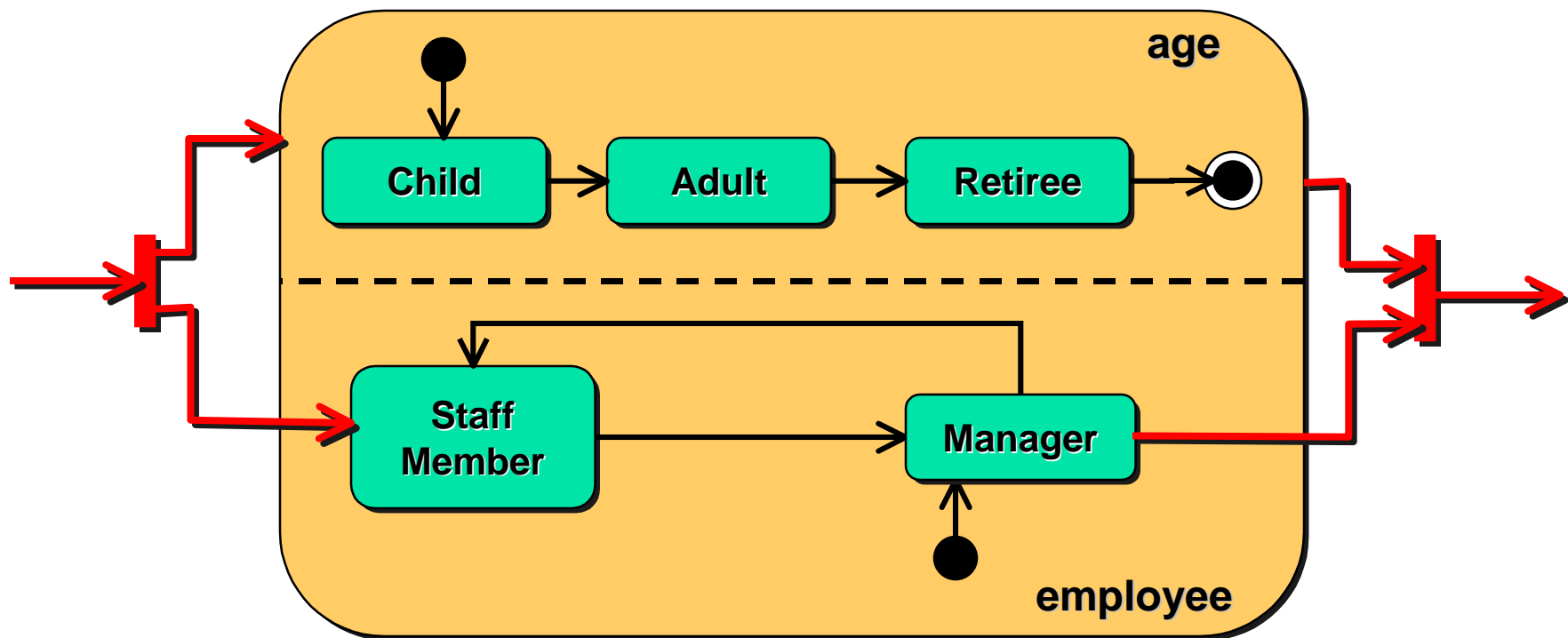
Common Misuse of Orthogonality

- Using regions to model independent objects
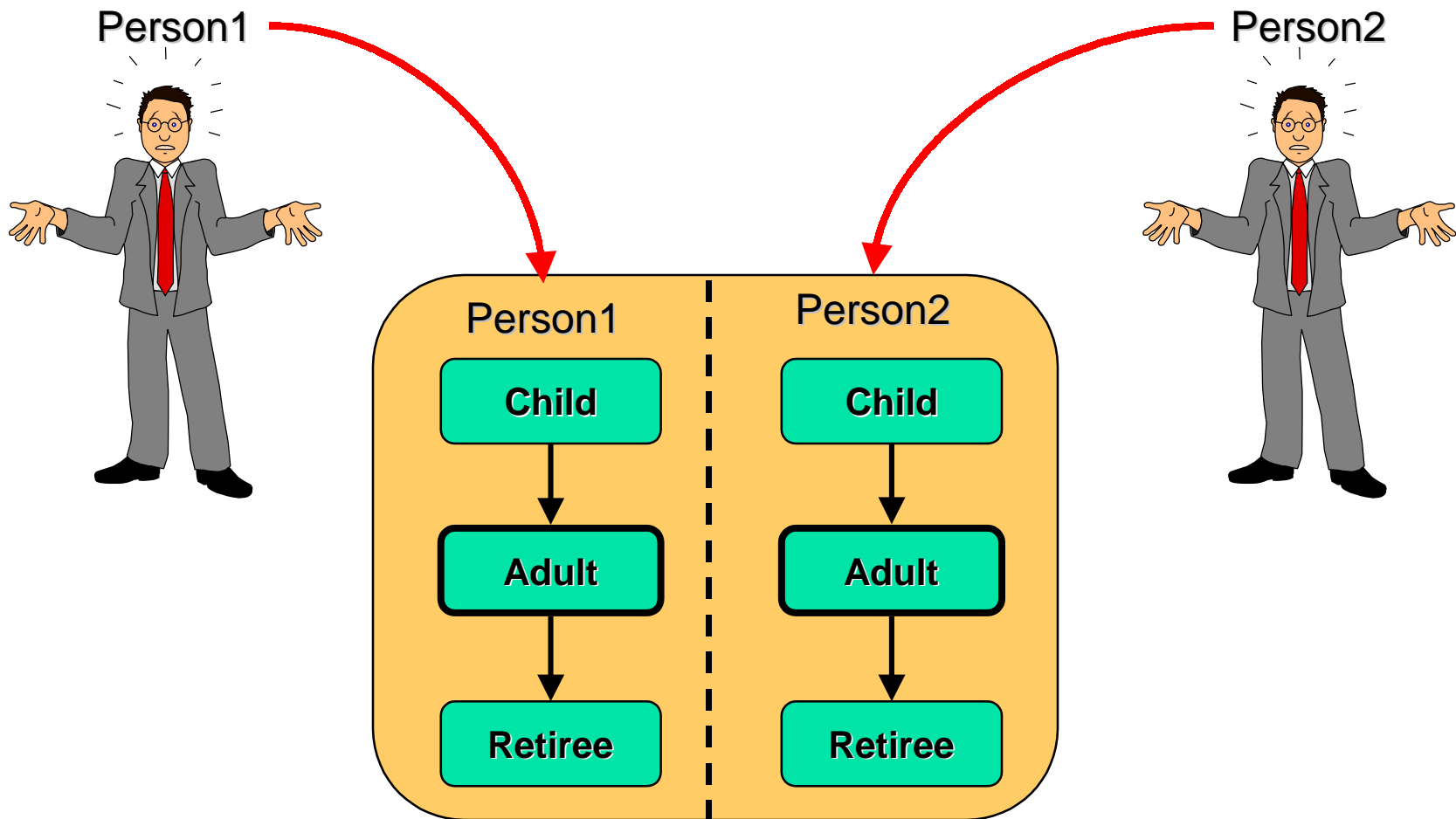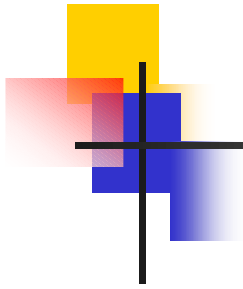
Basic State Machine Concepts

Statecharts and Objects

Advanced Modeling Concepts

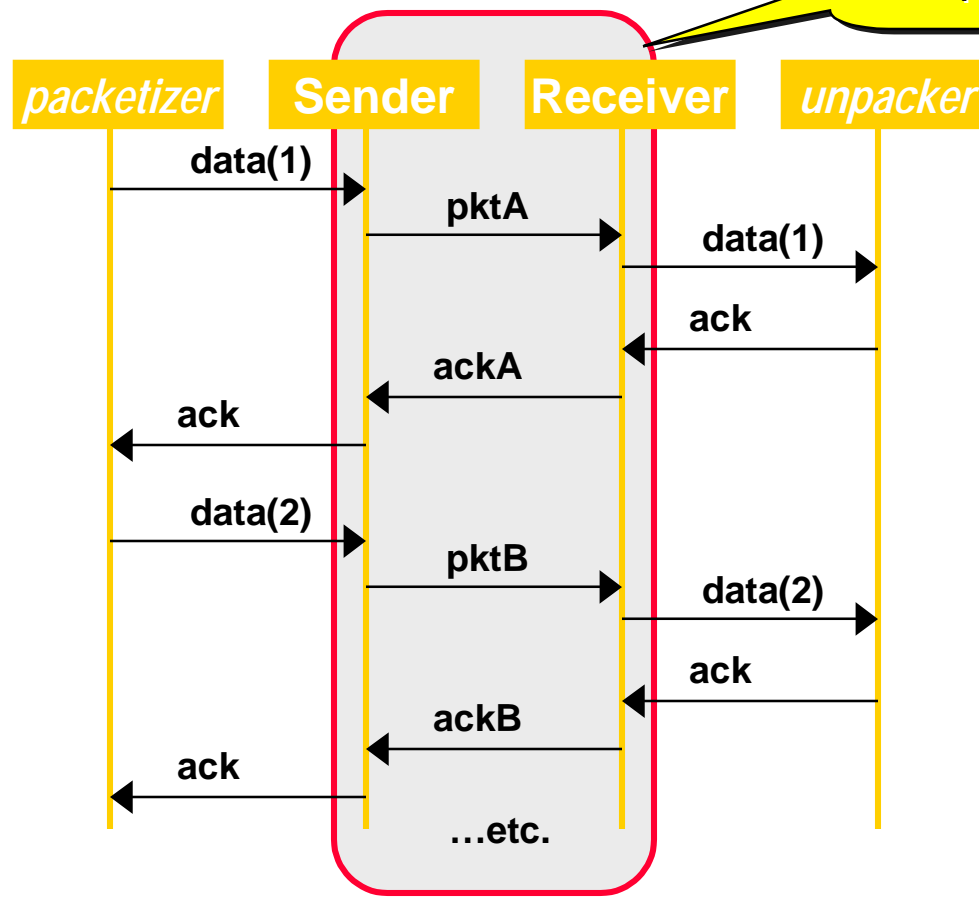Case Study

Wrap Up

Case Study: Protocol Handler

- A multi-line packet switch that uses the alternating-bit protocol as its link protocol
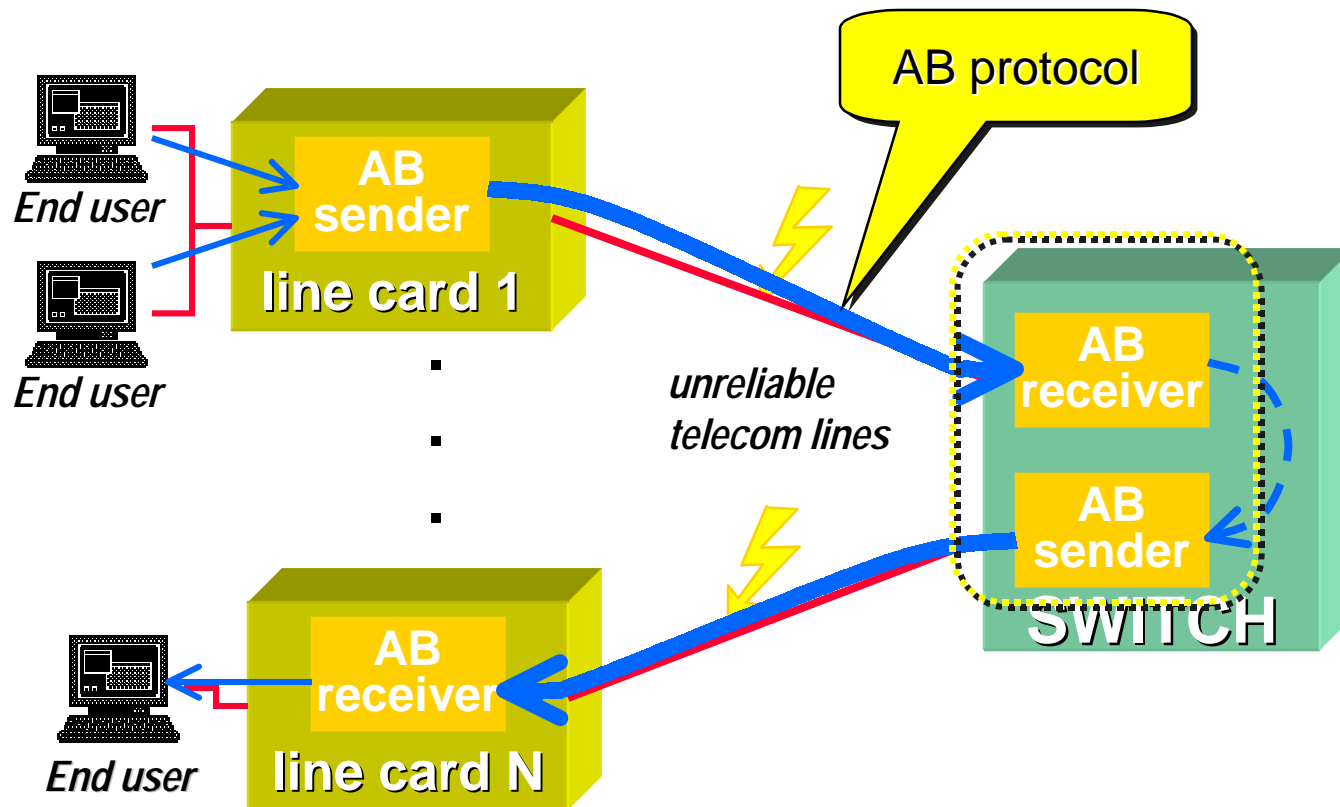
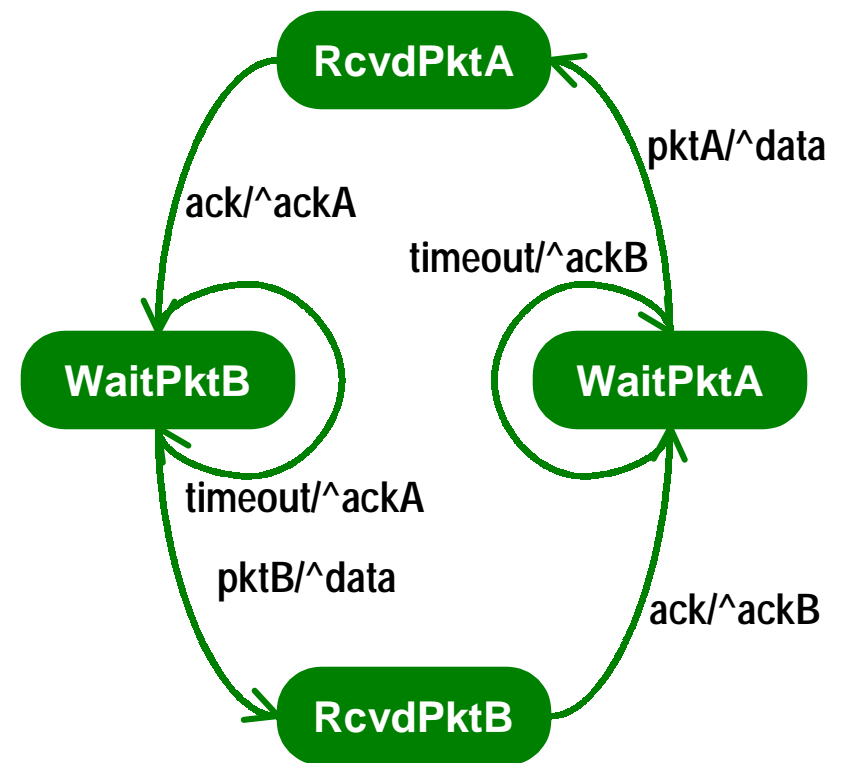# Alternating Bit Protocol (1)

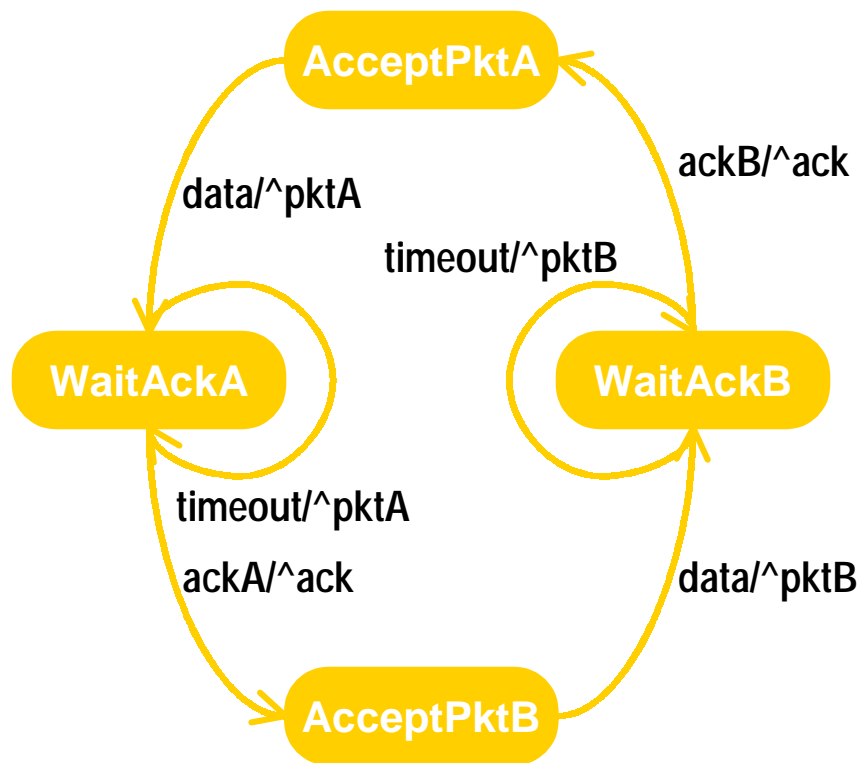- A simple one-way point-to-point packet protocol

AB protocol

packetizer	Sender	Receiver	unpacker

data(1) →

pktA →

data(1) →

← ack

← ackA

← ack

data(2) →

pktB →

data(2) →

← ack

← ackB

← ack

…etc.

Alternating Bit Protocol (2)

- State machine specification

Sender SM

Receiver SM

AcceptPktA

data/^pktA

ackB/^ack

timeout/^pktB

WaitAckA

WaitAckB

timeout/^pktA

ackA/^ack

data/^pktB

AcceptPktB

RcvdPktA

ack/^ackA

pktA/^data

timeout/^ackB

WaitPktB

WaitPktA

timeout/^ackA

pktB/^data

ack/^ackB

RcvdPktB

Additional Considerations

- Support (control) infrastructure
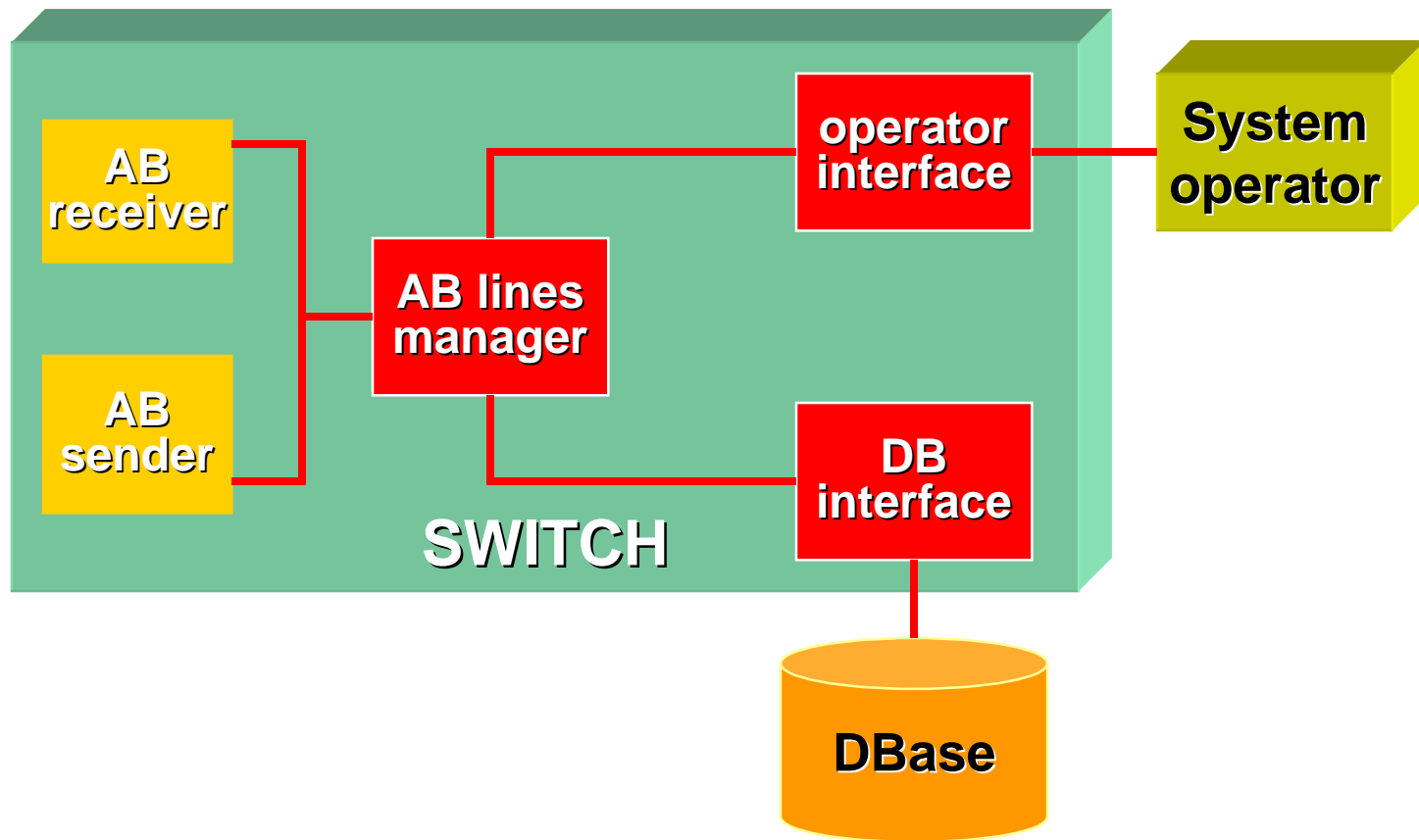
Control

The set of (additional) mechanisms and actions required to bring a system into the desired operational state and to maintain it in that state in the face of various planned and unplanned disruptions
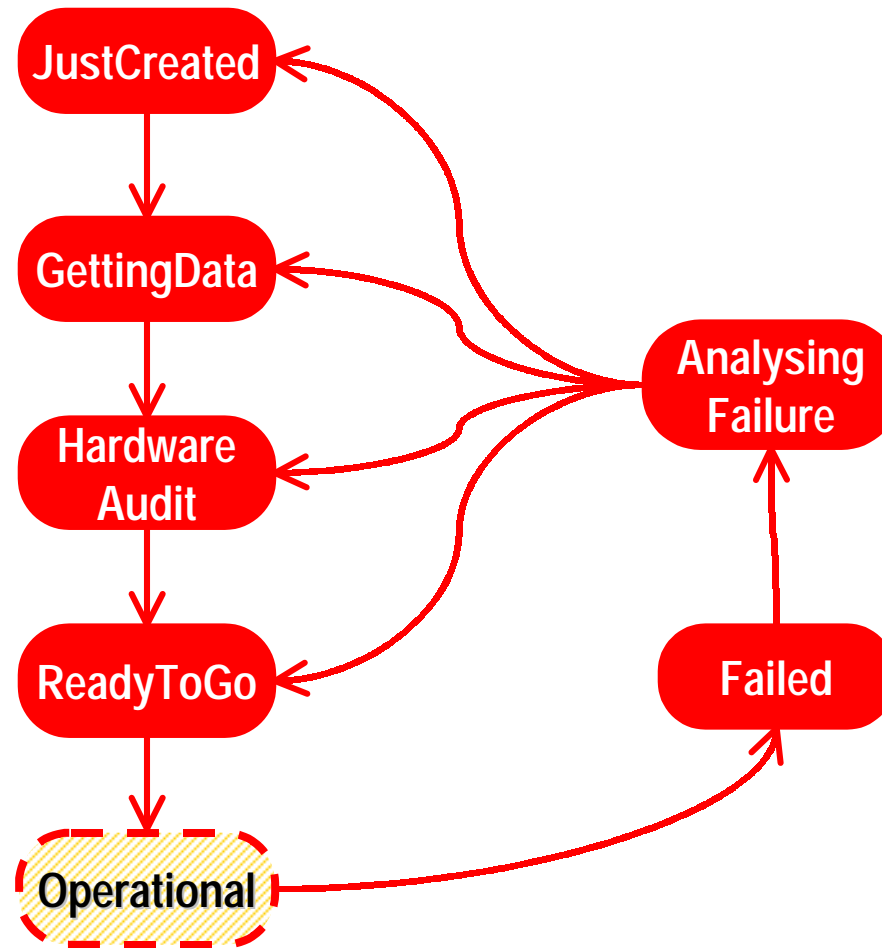
For software systems this includes:
- system/component start-up and shut-down
- failure detection/reporting/recovery
- system administration, maintenance, and provisioning
- (on-line) software upgrade
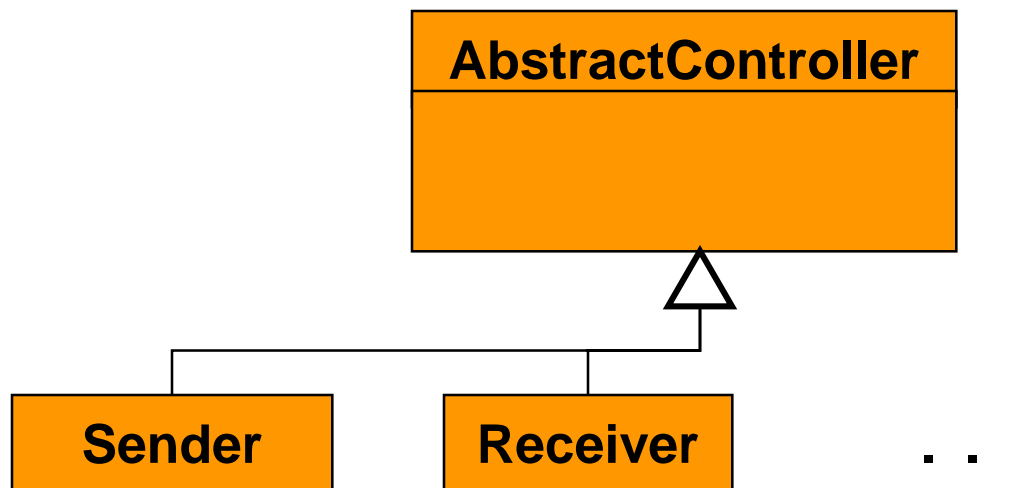
Retrofitting Control Behavior
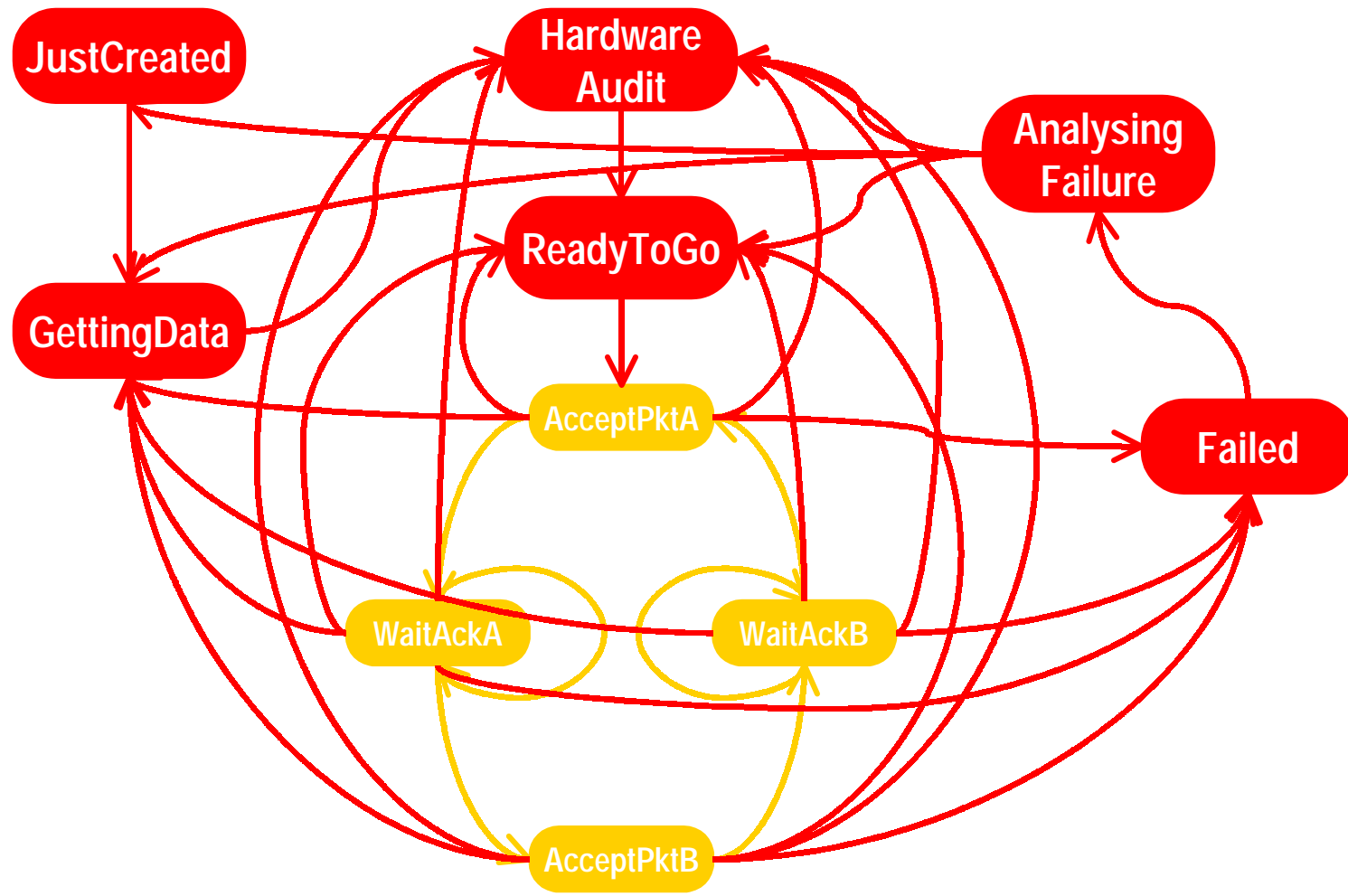
The Control Automaton

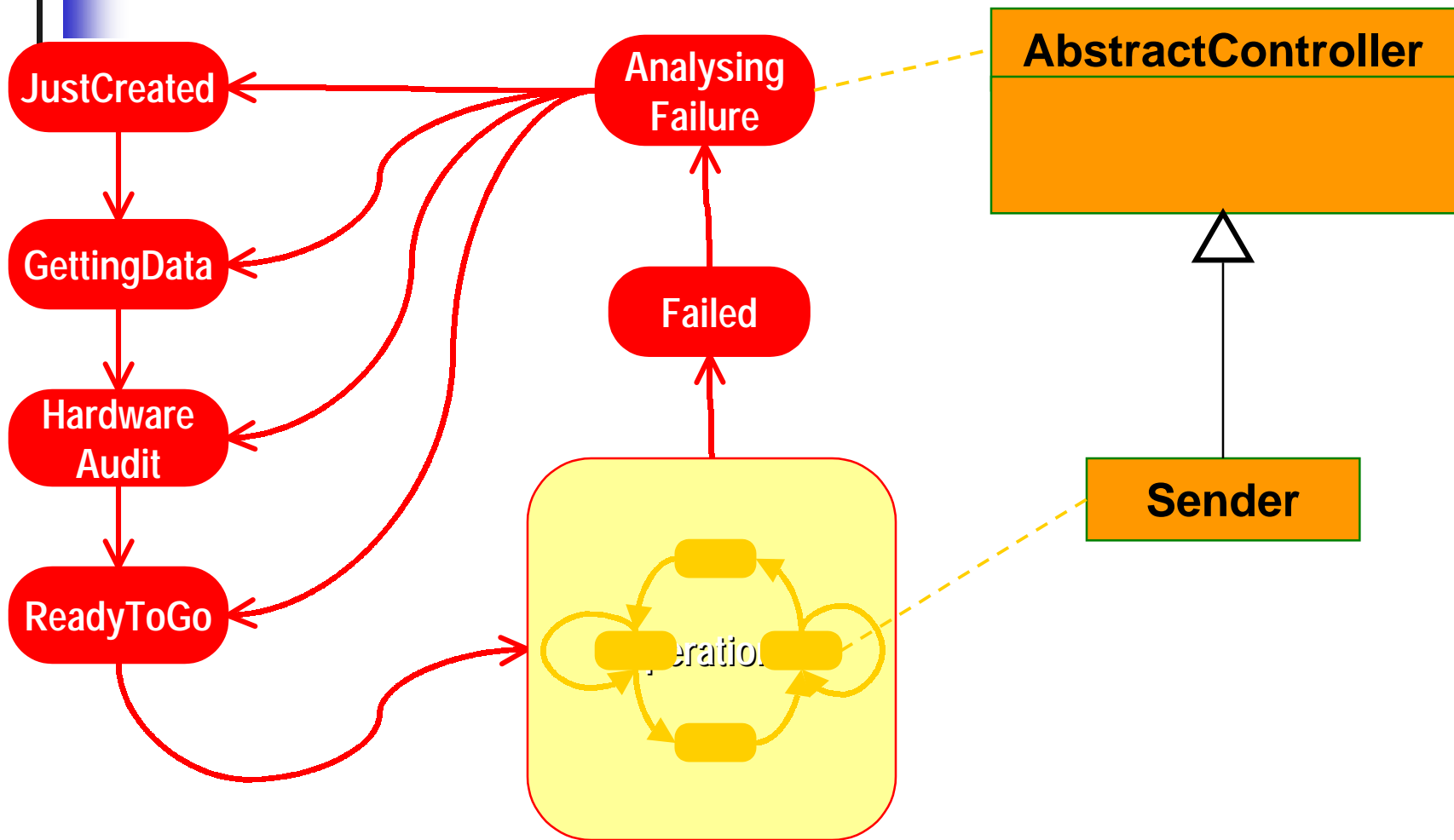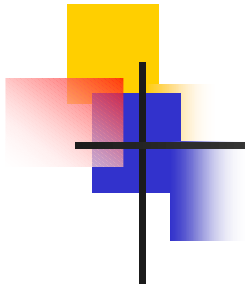- In isolation, the same control behavior appears much simpler

Exploiting Inheritance

- Abstract control classes can capture the common control behavior

```
         ┌─────────────────────────┐
         │   AbstractController     │
         ├─────────────────────────┤
         │                         │
         │                         │
         └─────────────────────────┘
                    △
          ┌─────────┴─────────┐
   ┌──────────┐      ┌──────────┐
   │  Sender  │      │ Receiver │     . . .
   └──────────┘      └──────────┘
```

Exploiting Hierarchical States

JustCreated

GettingData

Hardware Audit

ReadyToGo

Analysing Failure

Failed

Operation

AbstractController

Sender

Basic State Machine Concepts

Statecharts and Objects

Advanced Modeling Concepts

Case Study

Wrap Up

Wrap Up: Statecharts

- UML uses an object-oriented variant of Harel's statecharts
  - adjusted to software modeling needs
- Used to model event-driven (reactive) behavior
  - well-suited to the server model inherent in the object paradigm
- Primary use for modeling the behavior of active event-driven objects
  - systems modeled as networks of collaborating state machines
  - run-to-completion paradigm significantly simplifies concurrency management

Wrap Up: Statecharts (cont'd)

- Includes a number of sophisticated features that realize common state-machine usage patterns:
    - entry/exit actions
    - state activities
    - dynamic and static conditional branching
- Also, provides hierarchical modeling for dealing with very complex systems
    - hierarchical states
    - hierarchical transitions
    - orthogonality

Behavioral Modeling

- Part 1: Interactions and Collaborations
- Part 2: Statecharts
- Part 3: Activity Diagrams

Activity Diagram Applications

- Intended for applications that need control flow or object/data flow models ...

- ... rather than event-driven models like state machines.

- For example: business process modeling and workflow.

- The difference in the three models is how step in a process is initiated, especially with respect to how the step gets its inputs.

Control Flow

- Each step is taken when the previous one finishes ...
- ...regardless of whether inputs are available, accurate, or complete ("pull").
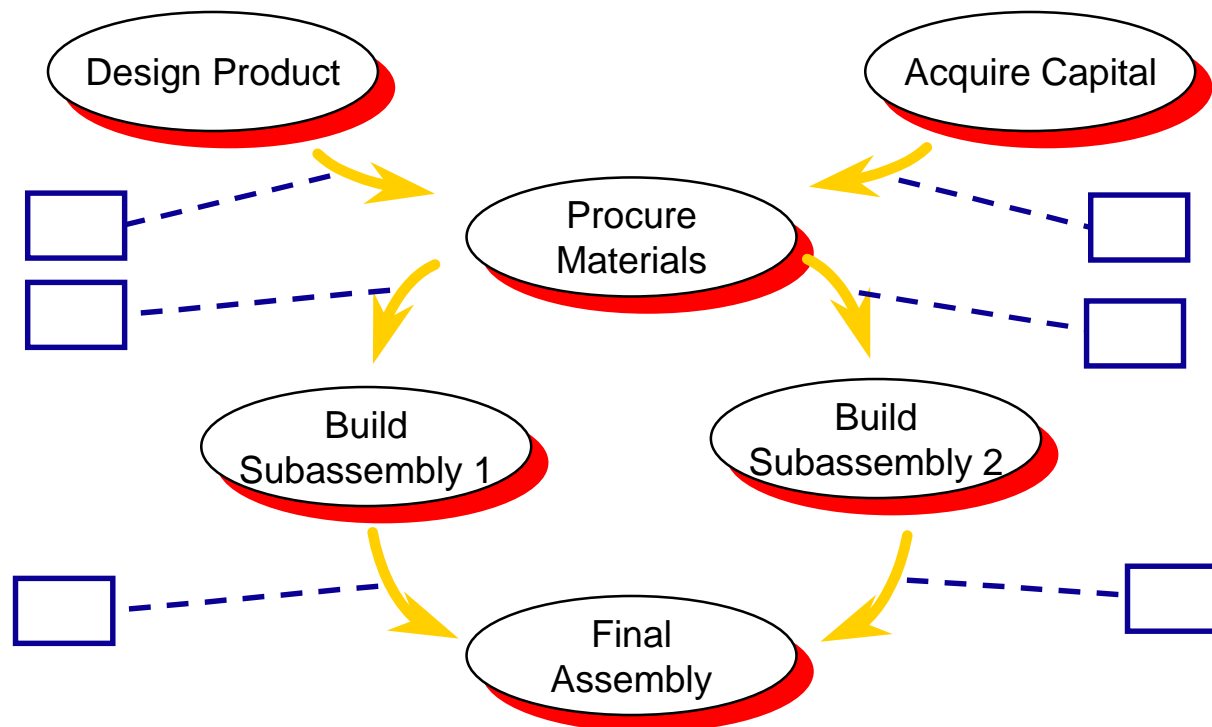- Emphasis is on order in which steps are taken.

Start

Weather Info

Analyze Weather Info

Not UML
Notation!

Chart Course

Cancel Trip
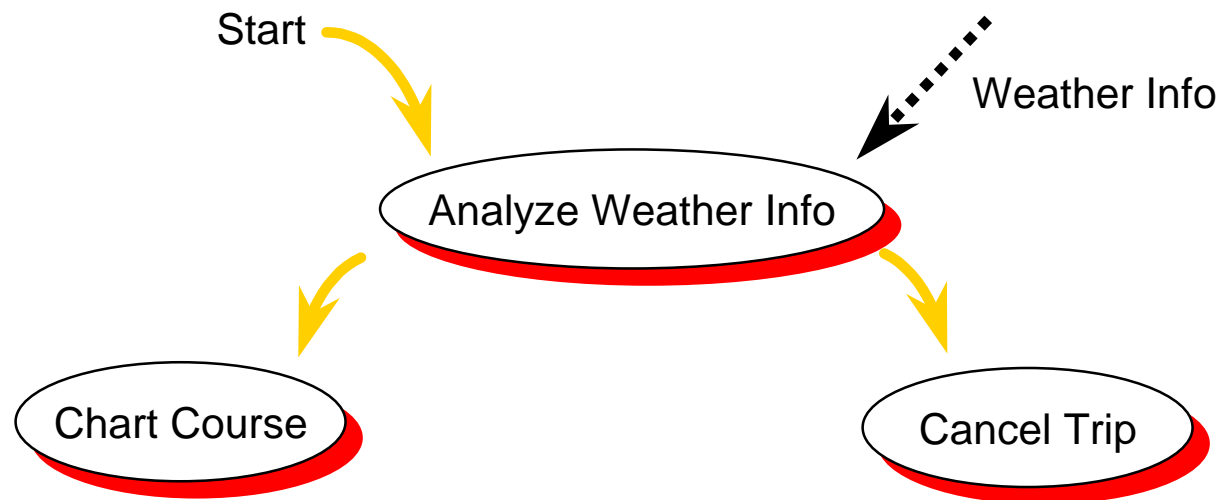
Object/Data Flow

- Each step is taken when all the required input objects/data are available ...

  - ... and only when all the inputs are available ("push").

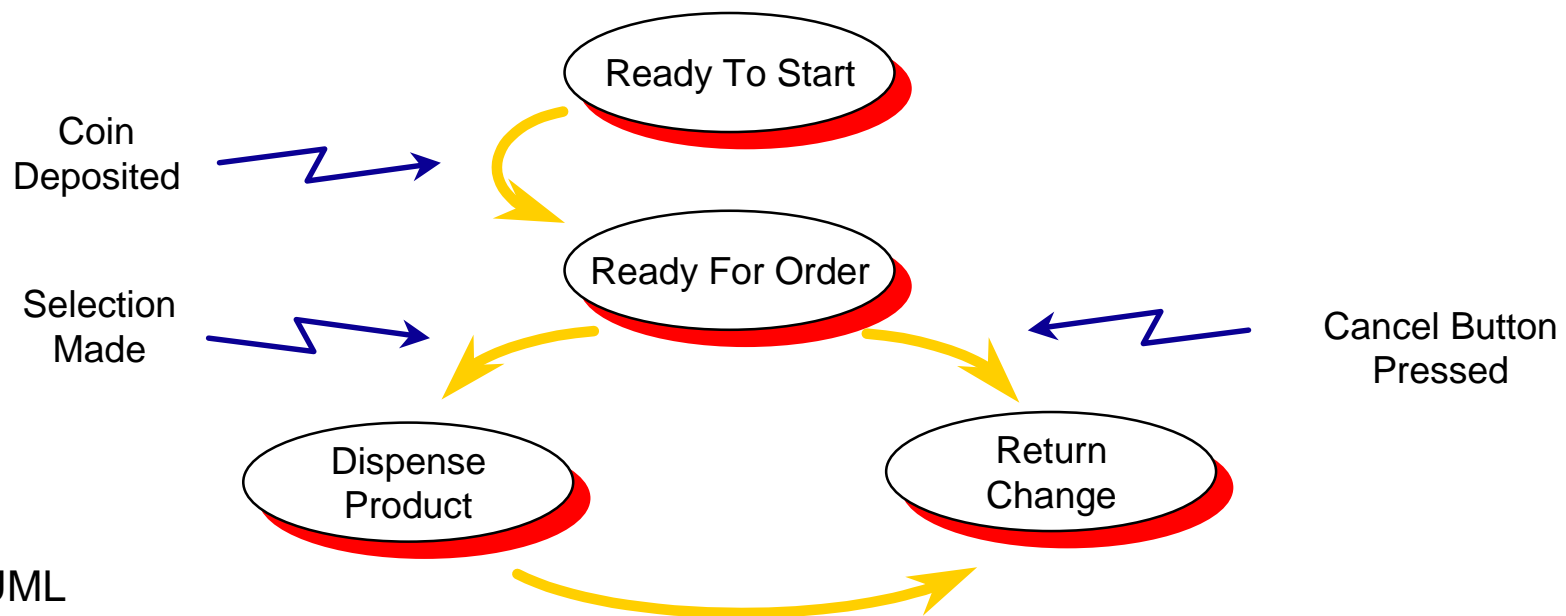 - Emphasis is on objects flowing between steps.

Design Product

Acquire Capital

Procure Materials

Build Subassembly 1

Build Subassembly 2

Final Assembly

Not UML Notation

State Machine

- Each step is taken when events are detected by the machine ...

- ... using inputs given by the event.

- Emphasis is on reacting to environment.

Coin
Deposited

Ready To Start

Ready For Order

Cancel Button
Pressed

Selection
Made

Dispense
Product

Return
Change

Not UML
Notation

Activity Diagrams Based on State Machines

- Currently activity graphs are modeled as a kind of state machine.

- Modeler doesn't normally need to be aware of this sleight-of-hand …

- … but will notice that "state" is used in the element names.

- Activity graphs will become independent of state machines in UML 2.0.

Kinds of Steps in Activity Diagrams

- ### Action (State)

  Action

- ### Subactivity (State)

  Subactivity

- Just like their state machine counterparts (simple state and submachine state) except that …

- … transitions coming out of them are taken when the step is finished, rather than being triggered by a external event, …
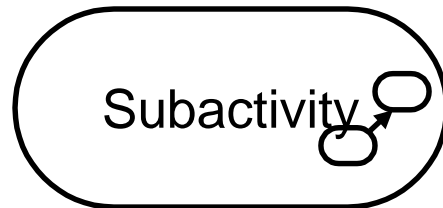
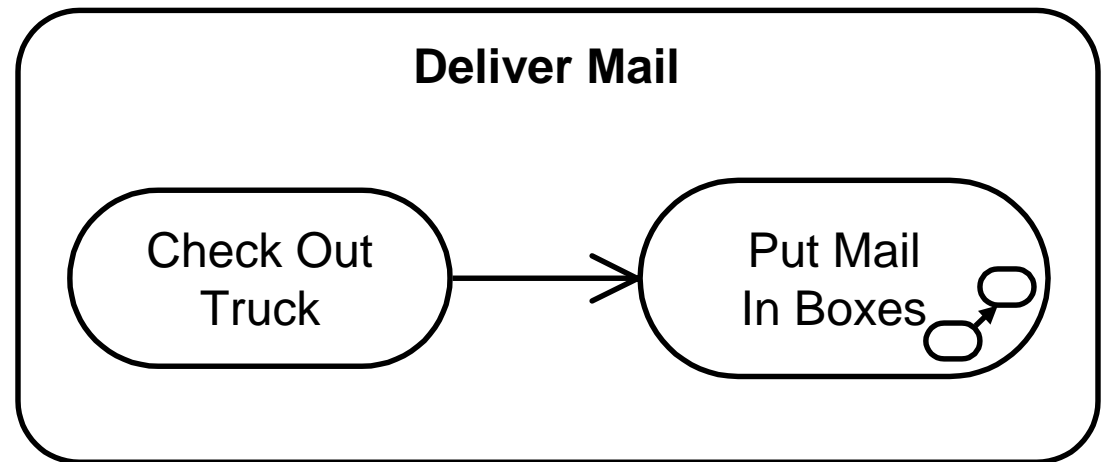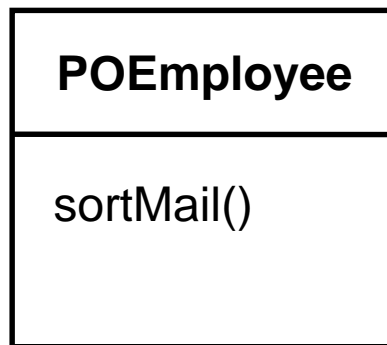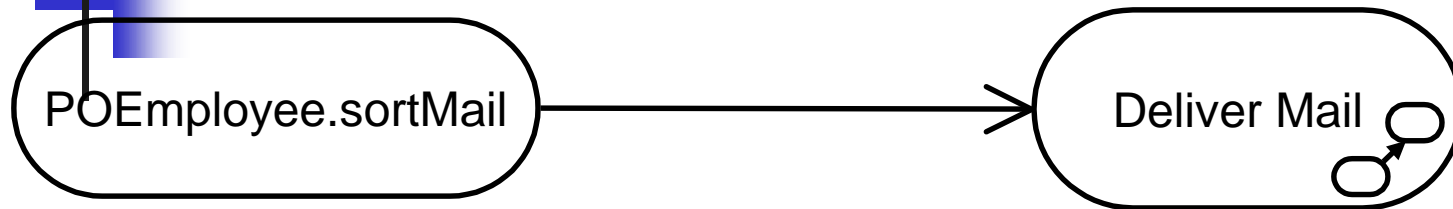- … and they support dynamic concurrency.

Action (State)

Action

- An action is used for anything that does not directly start another activity graph, like invoking an operation on an object, or running a user-specified action.

- However, an action can invoke an operation that has another activity graph as a method (possible polymorphism).
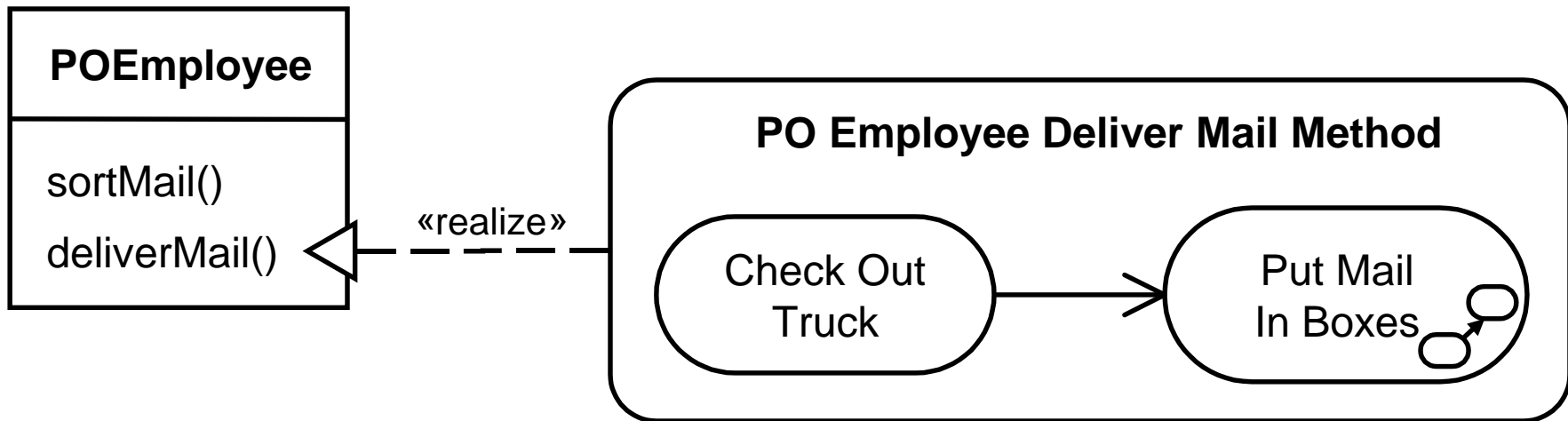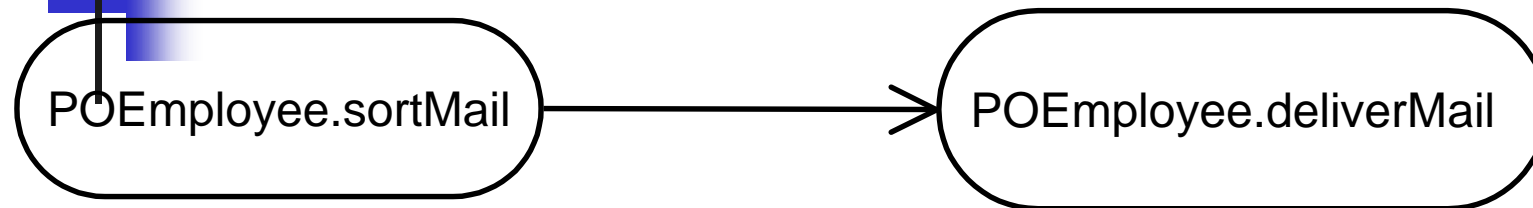
Subactivity (State)

- A subactivity (state) starts another activity graph without using an operation.

- Used for functional decomposition, non-polymorphic applications, like many workflow systems.

- The invoked activity graph can be used by many subactivity states.

Example

POEmployee.sortMail → Deliver Mail

POEmployee

sortMail()

Deliver Mail

Check Out Truck → Put Mail In Boxes

Activity Graph as Method

POEmployee.sortMail $\longrightarrow$ POEmployee.deliverMail

POEmployee

sortMail()

deliverMail()

«realize»

PO Employee Deliver Mail Method

Check Out Truck $\longrightarrow$ Put Mail In Boxes

- Application is completely OO when all action states invoke operations

- All activity graphs are methods for operations.

Dynamic concurrency

> Action/Subactivity *

- Applies to actions and subactivities.

- Not inherited from state machines.

- Invokes an action or subactivity any number of times in parallel, as determined by an expression evaluated at runtime. Expression also determines arguments.

- Upper right-hand corner shows a multiplicity restricting the number of parallel invocations.

- Outgoing transition triggered when all invocations are done.

- Currently no standard notation for concurrency expression or how arguments are accessed by actions. Attach a note as workaround for expression. Issue for UML 1.4.
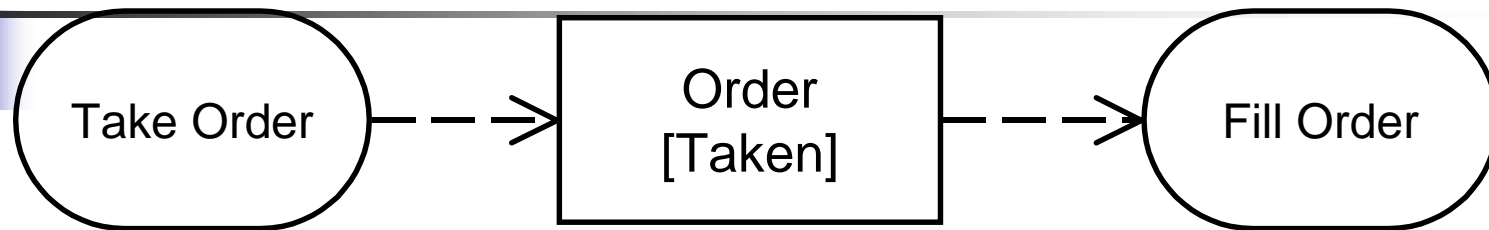
Object Flow (State)

```
┌─────────────────┐
│      Class      │
│     [State]     │
└─────────────────┘
```

- A special sort of step (state) that represents the availability of a particular kind of object, perhaps in a particular state.

- No action or subactivity is invoked and control passes immediately to the next step (state).

- Places constraints on input and output parameters of steps before and after it.

Object Flow (State)

```
  _____                 _____                  _____
 /        \               |   Order   |                /        \
|Take Order| - - - - ->   | [Taken]   |  - - - - ->   | Fill Order|
 _____/                |_____|                _____/
```
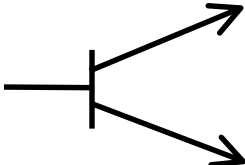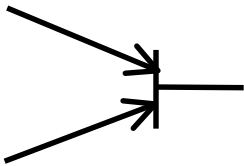
- Take Order must have an output parameter giving an order, or one of its subtypes.

- Fill Order must have an input parameter taking an order, or one of its supertypes.

- Dashed lines used with object flow have the same semantics as any other state transition.

Coordinating Steps
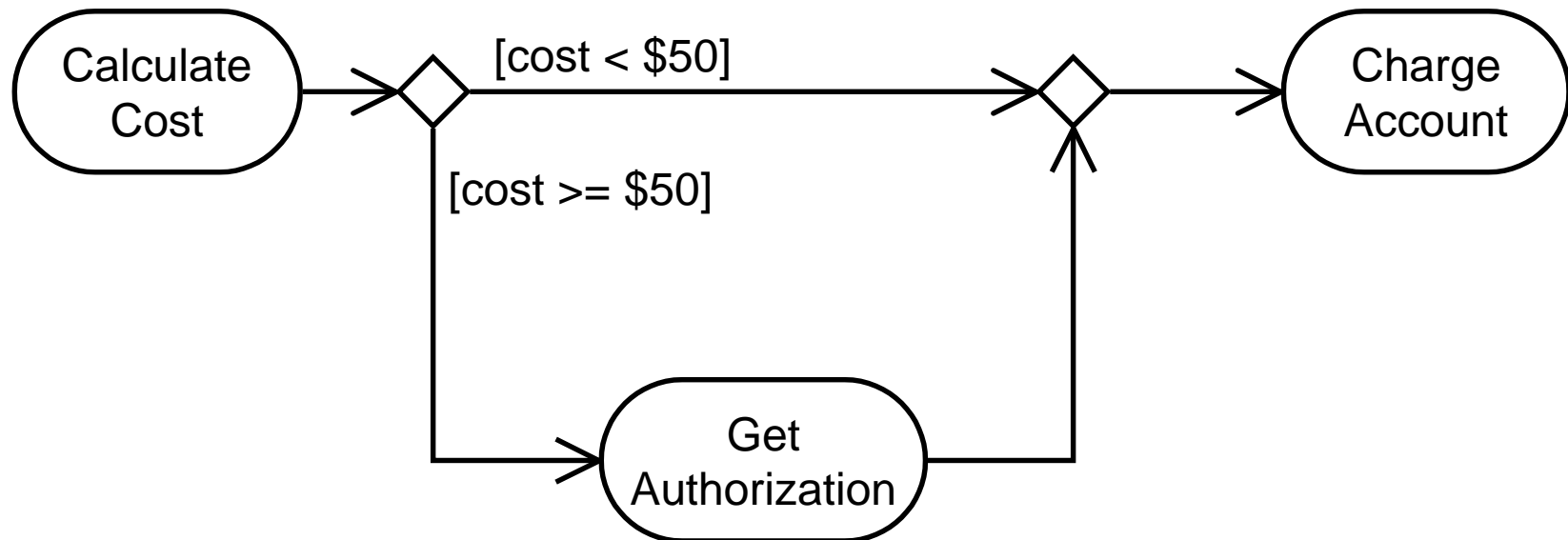
- **Inherited from state machines**

- **Initial state** ●

- **Final state** ◉

- **Fork and join**

Coordinating Steps

- *Decision point* and *merge* (◇)
  are inherited from state machines.

- For modeling conventional flow chart
  decisions.

```
┌──────────┐          [cost < $50]                    ┌──────────┐
│Calculate │─────►◇──────────────────────►◇──────────►│ Charge   │
│  Cost    │                                          │ Account  │
└──────────┘                                          └──────────┘
               [cost >= $50]
                              ┌──────────────┐
                              │     Get      │
                         ────►│Authorization │────►
                              └──────────────┘
```

Coordinating Steps

- *Synch state* (◯) is inherited from state machines but used mostly in activity graphs.
- Provides communication capability between parallel processes.

State machine notation
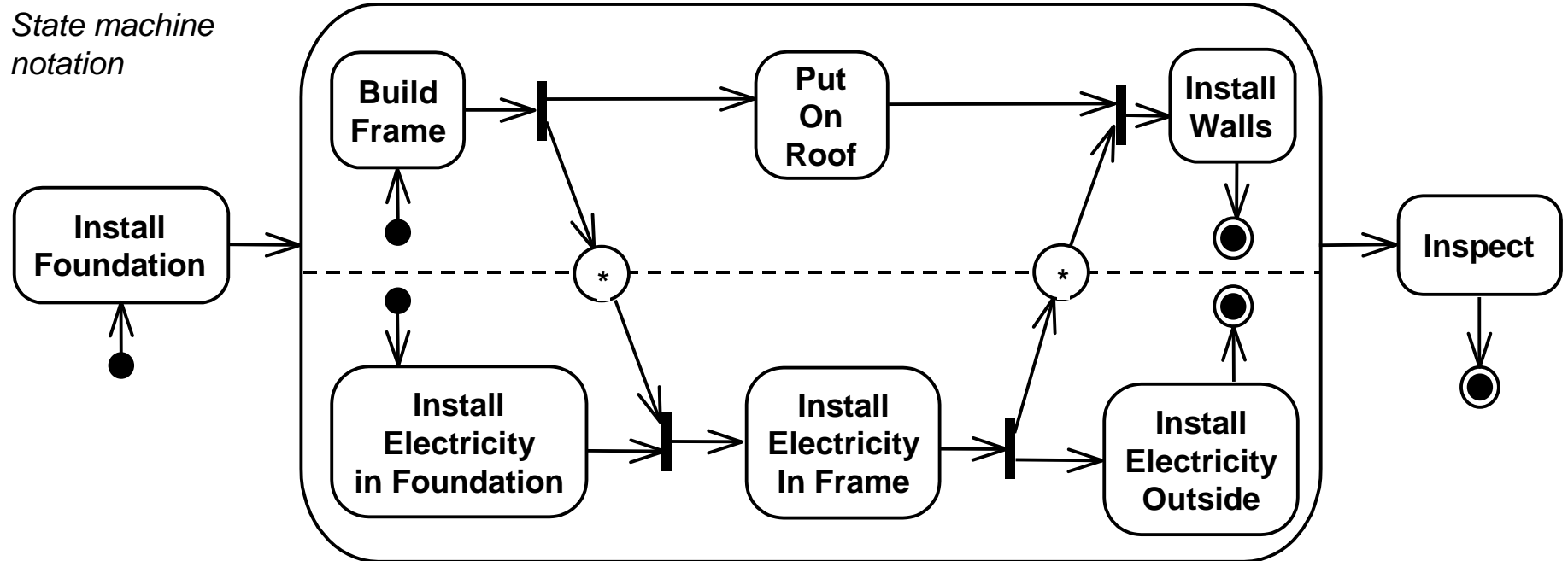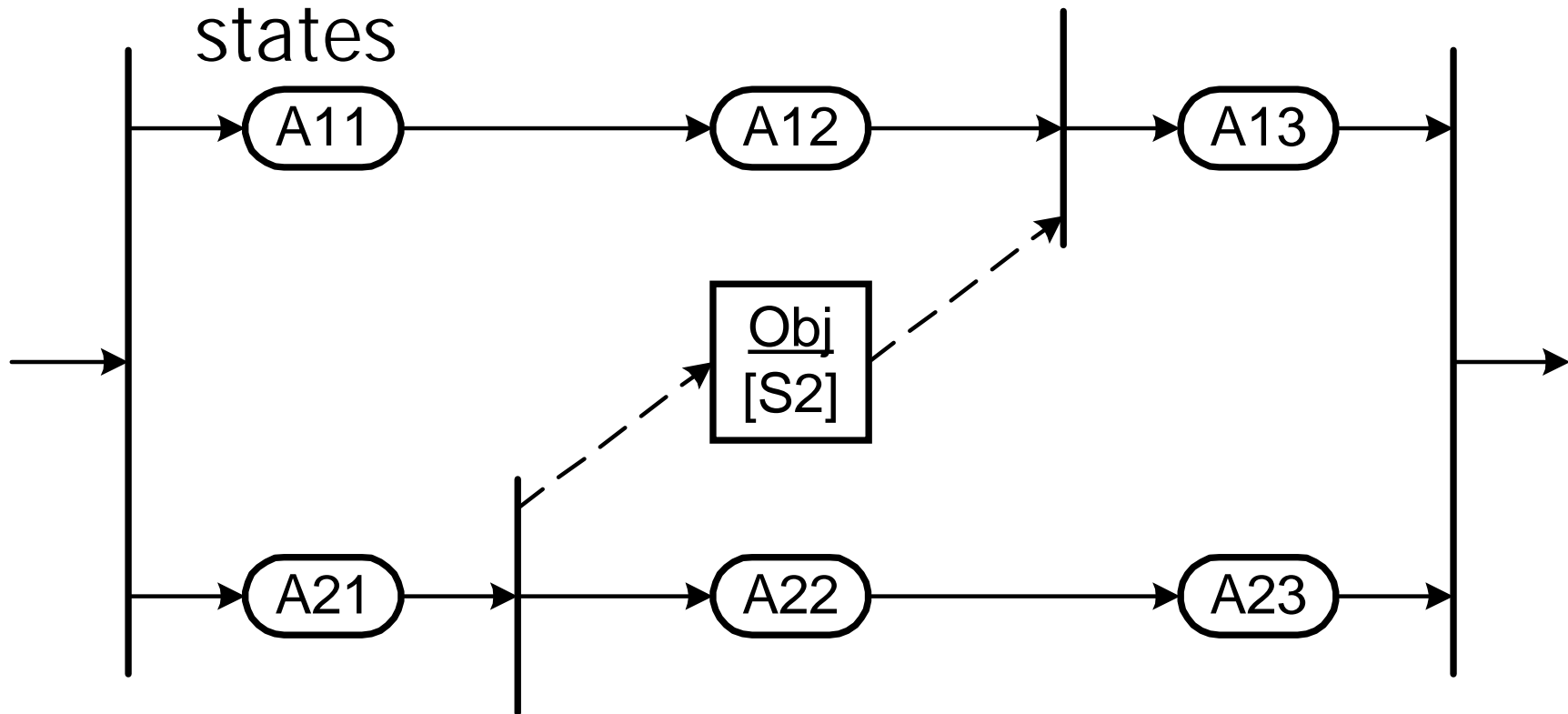
Convenience Features (Synch State)

- Forks and joins do not require composite states.

- Synch states may be omitted for the common case (unlimited bound and one incoming and outgoing transition).
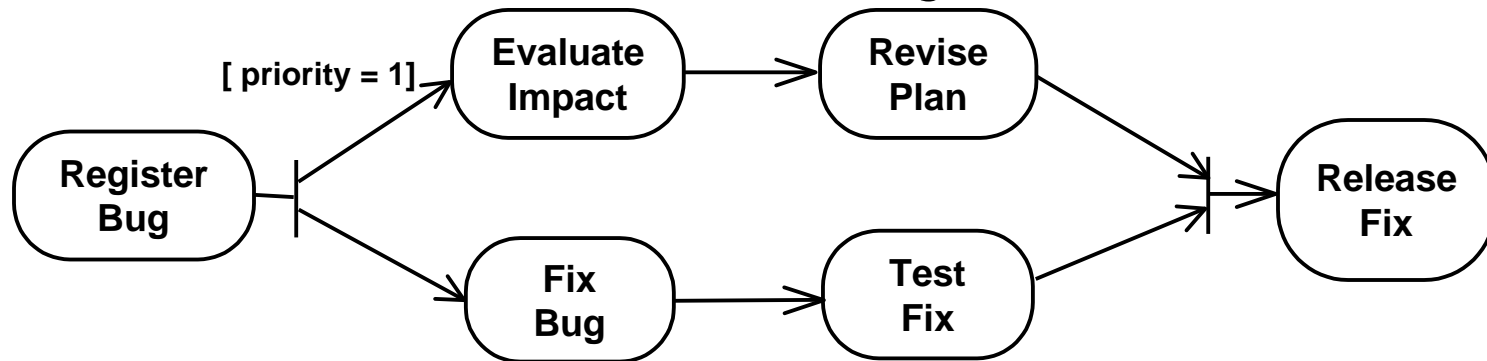
Activity diagram notation
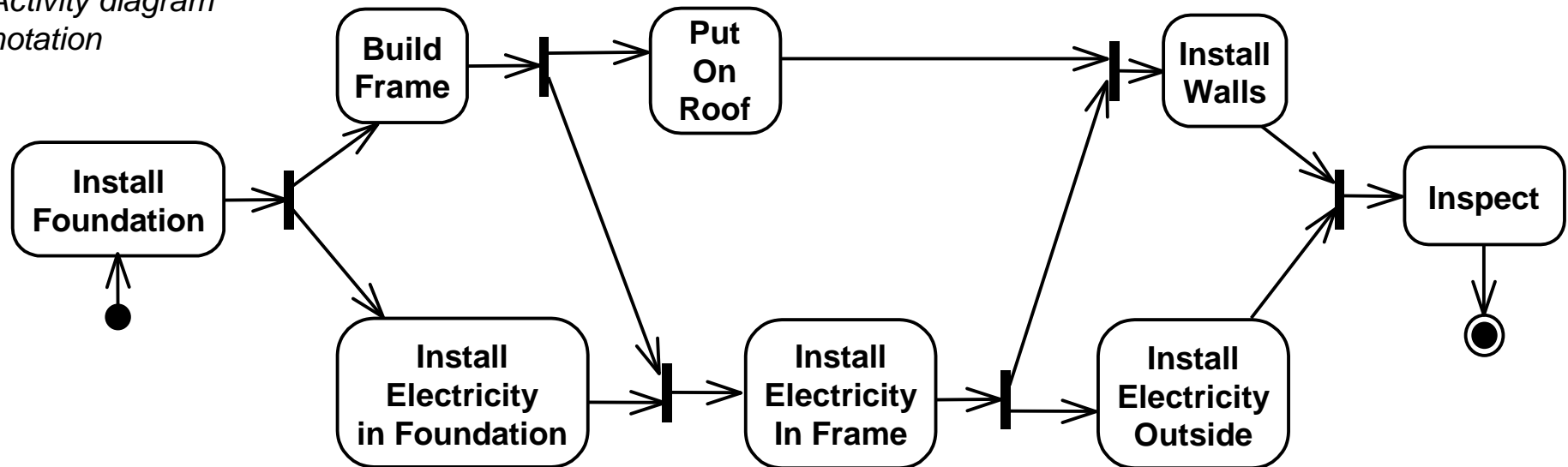
Convenience Features (Synch State)

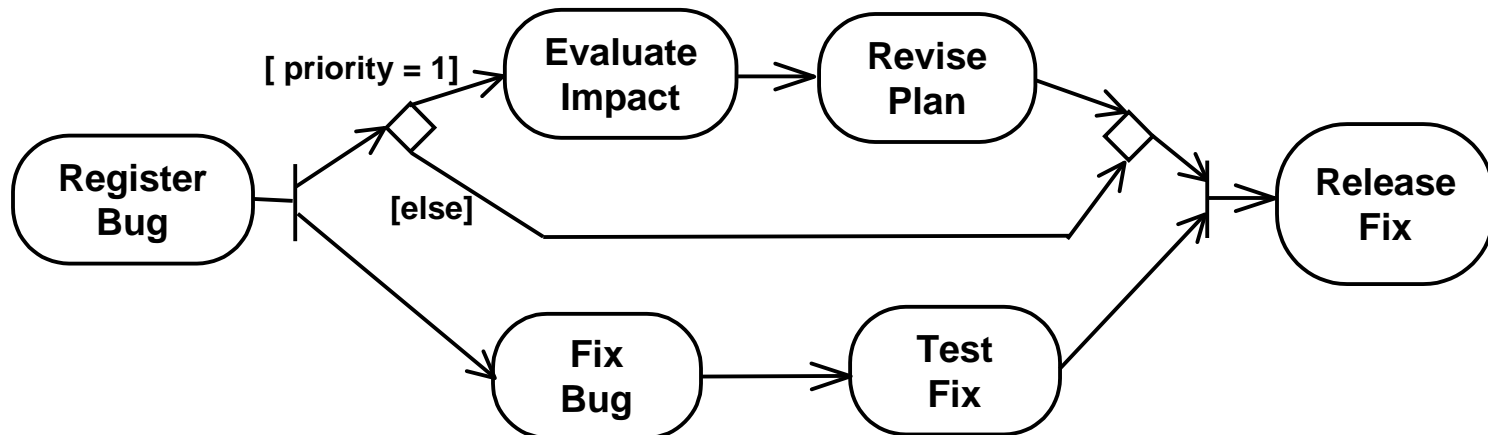- Object flow states can be synch states

Convenience Features

- Fork transitions can have guards.

[priority = 1]

| Register Bug | → | Evaluate Impact | → | Revise Plan | → | Release Fix |
| Fix Bug | → | Test Fix |

- Instead of doing this:

[priority = 1]

[else]

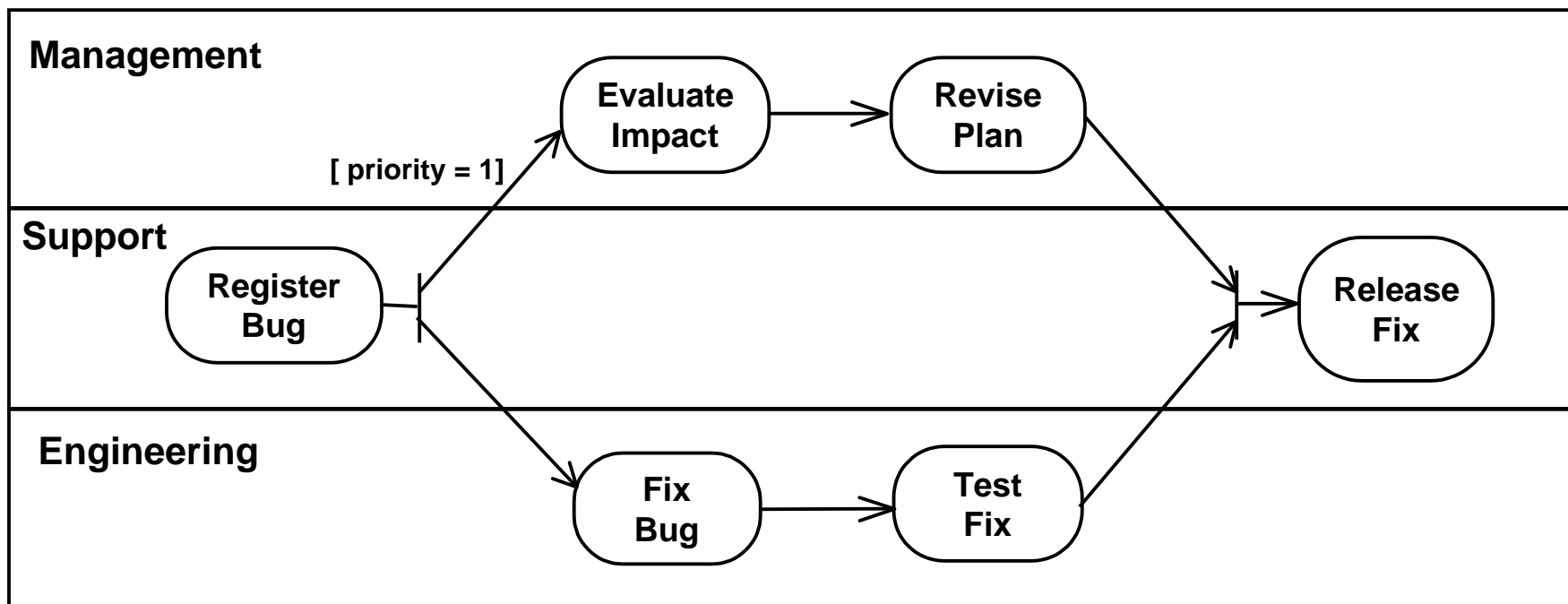| Register Bug | → | Evaluate Impact | → | Revise Plan | → | Release Fix |
| Fix Bug | → | Test Fix |

Convenience Features

- *Partitions* are a grouping mechanism.
- *Swimlanes* are the notation for partitions.
- They do not provide domain-specific semantics.
- Tools can generate swimlane presentation from domain-specific information without partitions.
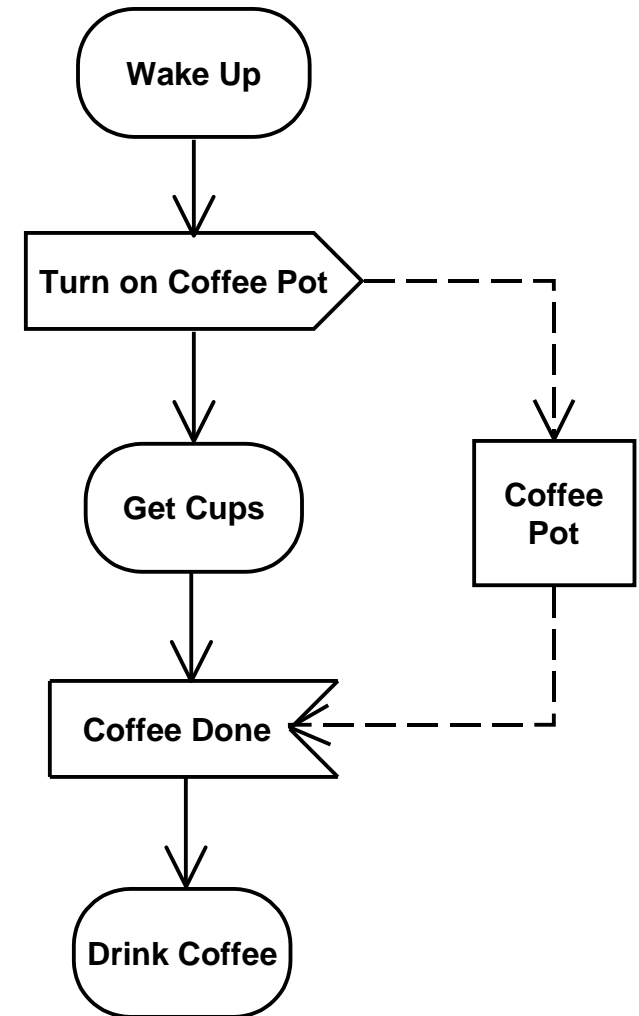
Convenience Features

- ## Signal send icon

  > Signal ⟩

- ## … translates to a transition with a send action.

- ## Signal receipt icon

  > Signal ⟨

- ## … translates to a wait state (an state with no action and a signal trigger event).
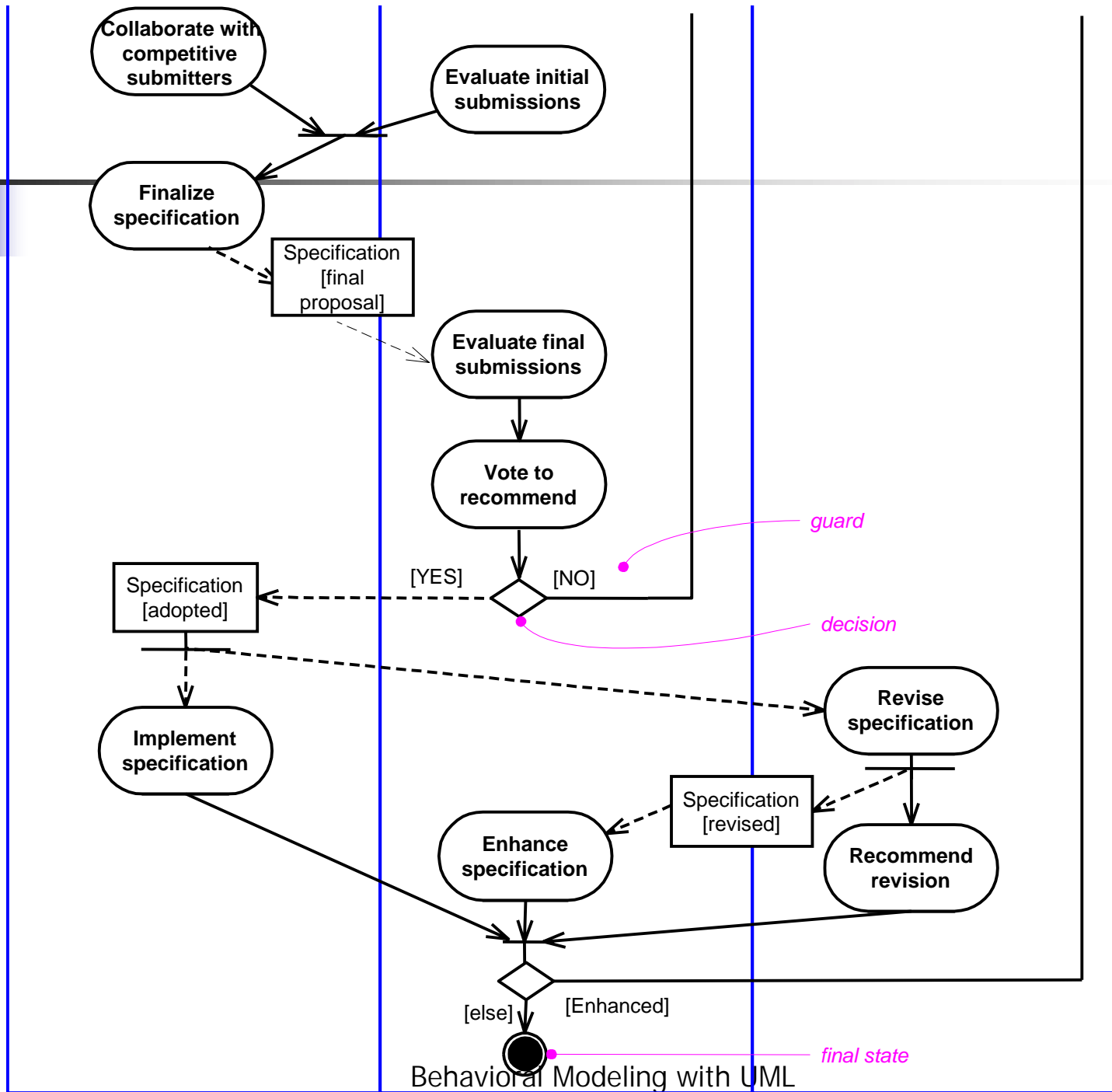
Wake Up

↓

Turn on Coffee Pot ⟩ ┈┈┈┐

↓   ↓

Get Cups   **Coffee Pot**

↓   ┊

Coffee Done ⟨┈┈┈┈┘

↓

Drink Coffee

Case Study

partition

Submission Team	Task Force	Revision Task Force

initial state

action state   *control flow*

Begin

fork of control

Develop technology specification   **Issue RFP**

RFP [issued]

conditional fork

join of control

Submit specification draft

[optional]

Specification [initial proposal]   *object flow*

input value

Collaborate with competitive submitters

Evaluate initial submissions

Finalize specification

Specification [final proposal]

Behavioral Modeling with UML   139

Case Study

Collaborate with competitive submitters

Evaluate initial submissions

Finalize specification

Specification [final proposal]

Evaluate final submissions

Vote to recommend

guard

[YES]   [NO]

Specification [adopted]

decision

Revise specification

Implement specification

Specification [revised]

Enhance specification

Recommend revision

[else]   [Enhanced]

final state

Behavioral Modeling with UML

140

When to Use Activity Diagrams

- Use activity diagrams when the behavior you are modeling …

  - does not depend much on external events.

  - mostly has steps that run to completion, rather than being interrupted by events.

  - requires object/data flow between steps.

  - is being constructed at a stage when you are more concerned with which activities happen, rather than which objects are responsible for them (except partitions possibly).
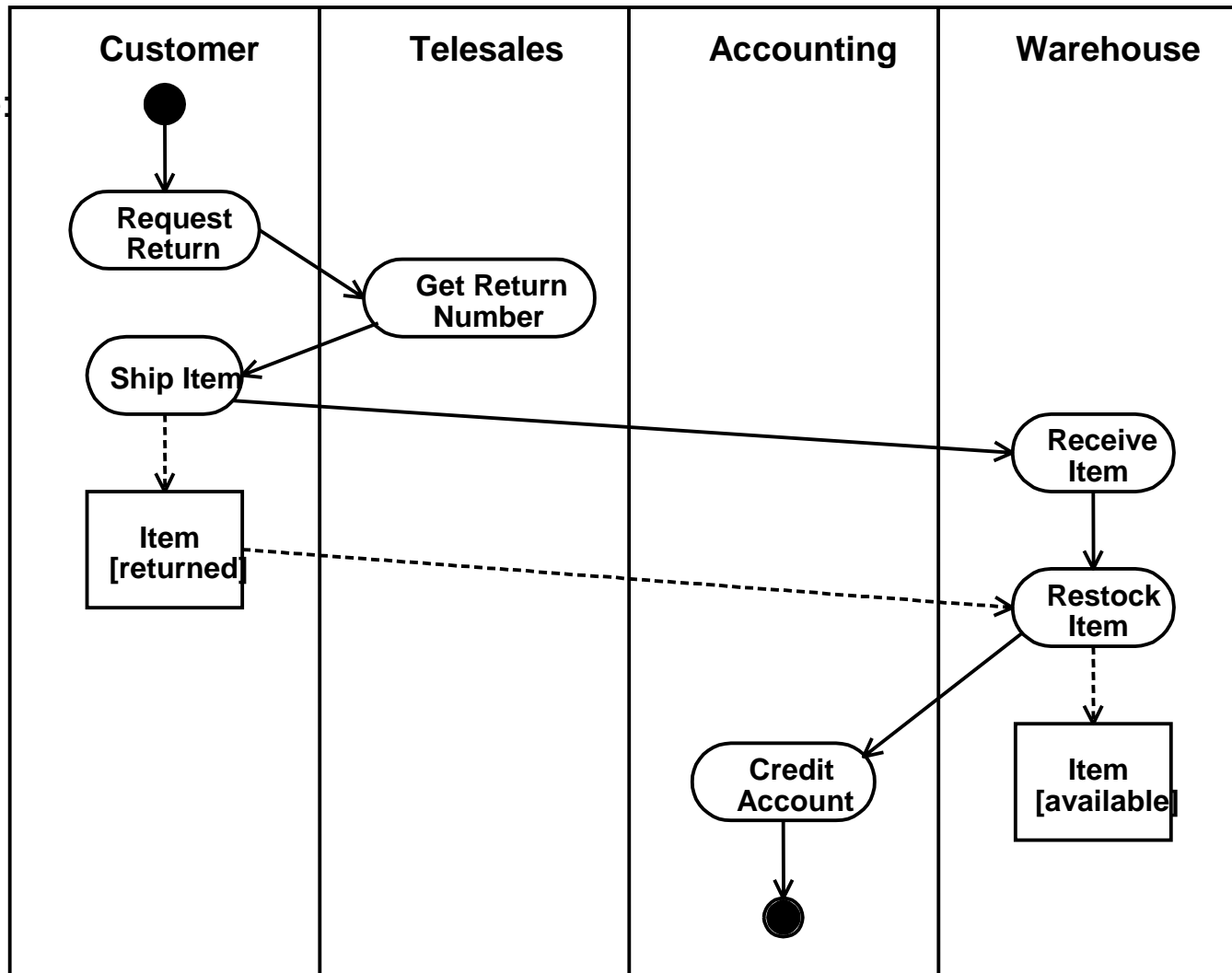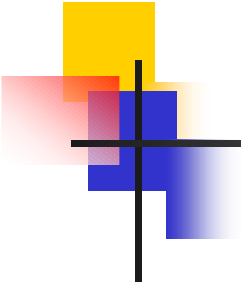
Activity Diagram Modeling Tips

- Control flow and object flow are not separate. Both are modeled with state transitions.

- Dashed object flow lines are also control flow.

- You can mix state machine and control/object flow constructs on the same diagram (though you probably do not want to).
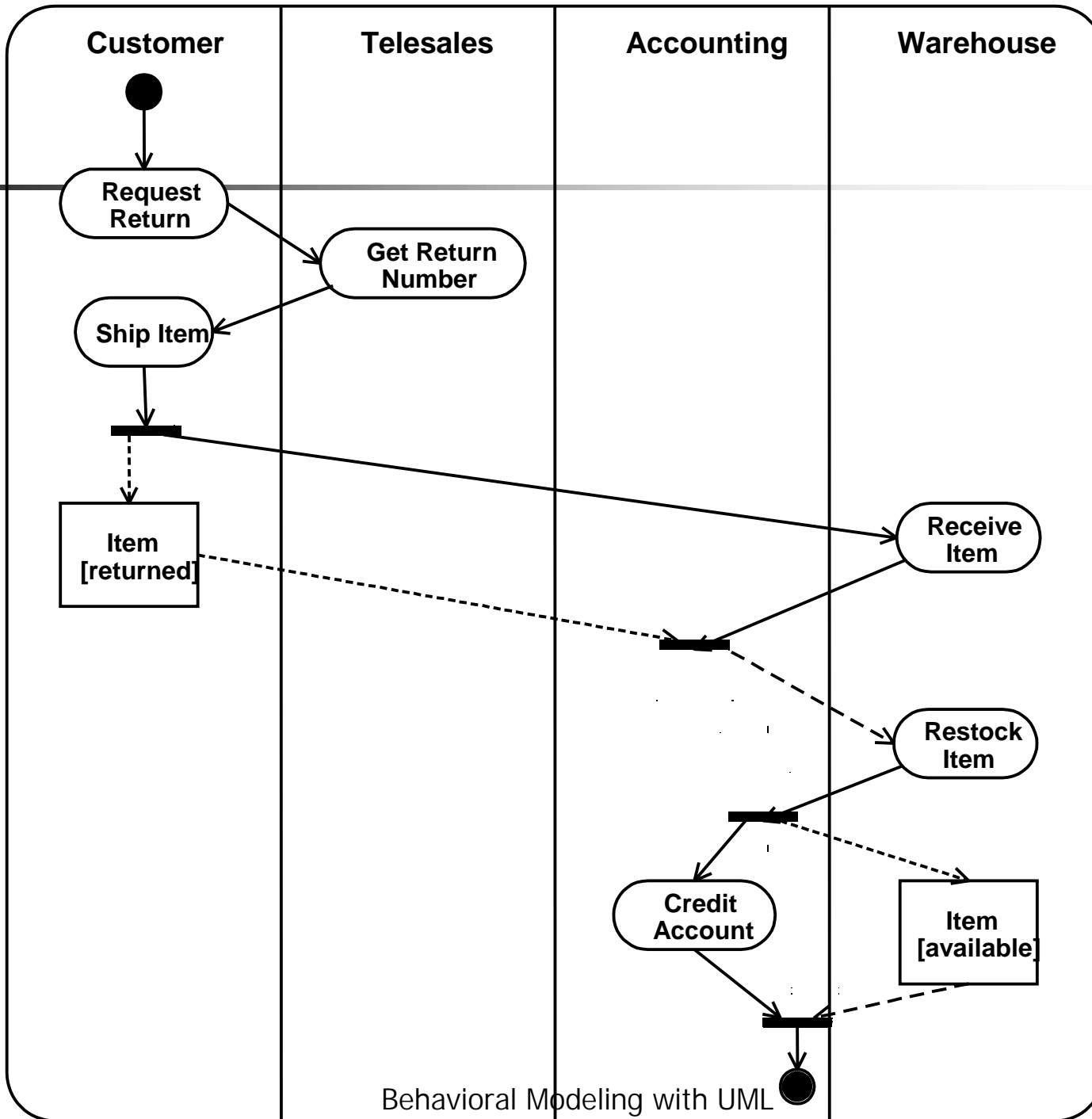
Activity Diagram Modeling Tips

From UML
User Guide:

Activity Modeling Tips

Customer	Telesales	Accounting	Warehouse

Request Return

Get Return Number

Ship Item

Item [returned]

Receive Item

Restock Item

Credit Account

Item [available]

Activity Diagram Modeling Tips

- Activity diagrams inherit from state machines the requirement for well-structured nesting of composite states.

- This means you should either model as if composite states were there by matching all forks/decisions with a correspond join/merges ...

- ... or check that the diagram can be translated to one that is well-nested.

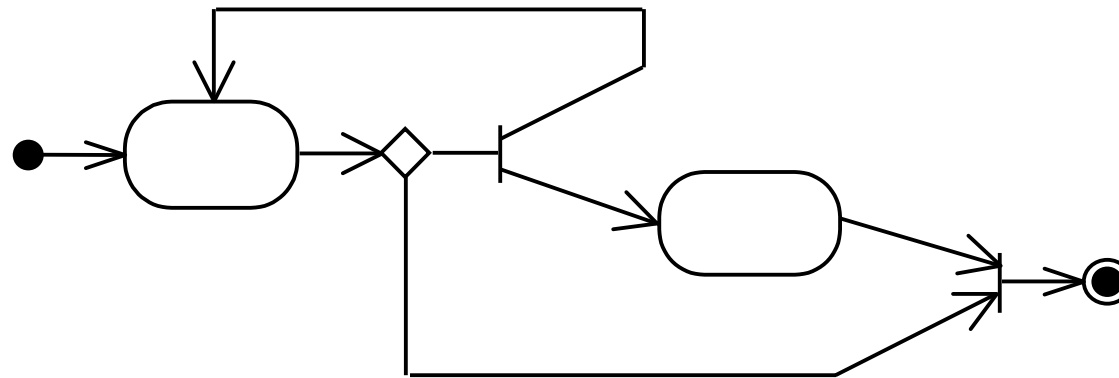- This insures that diagram is executable under state machine semantics.

Activity Diagram Modeling Tips

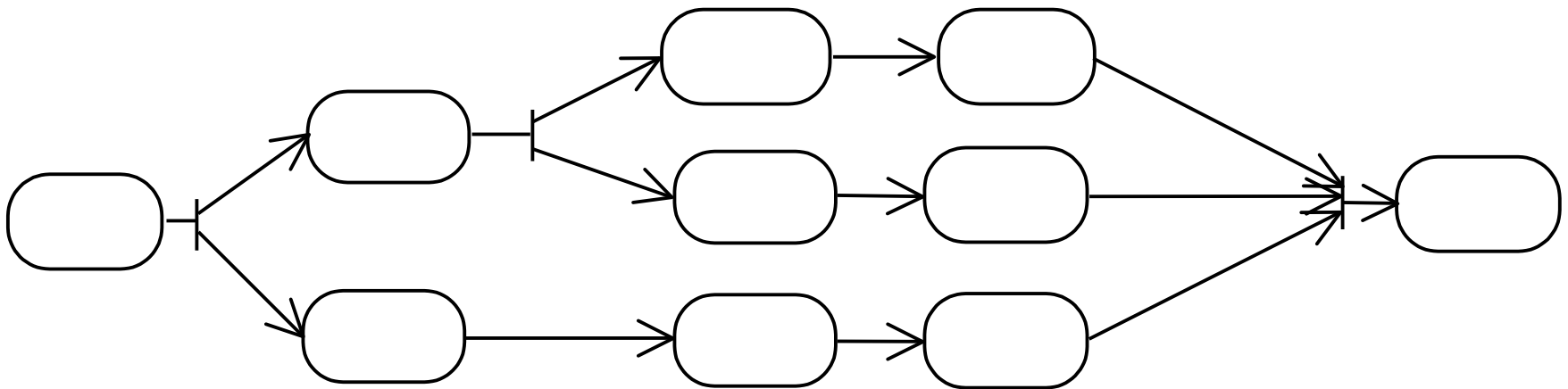Well-nested:
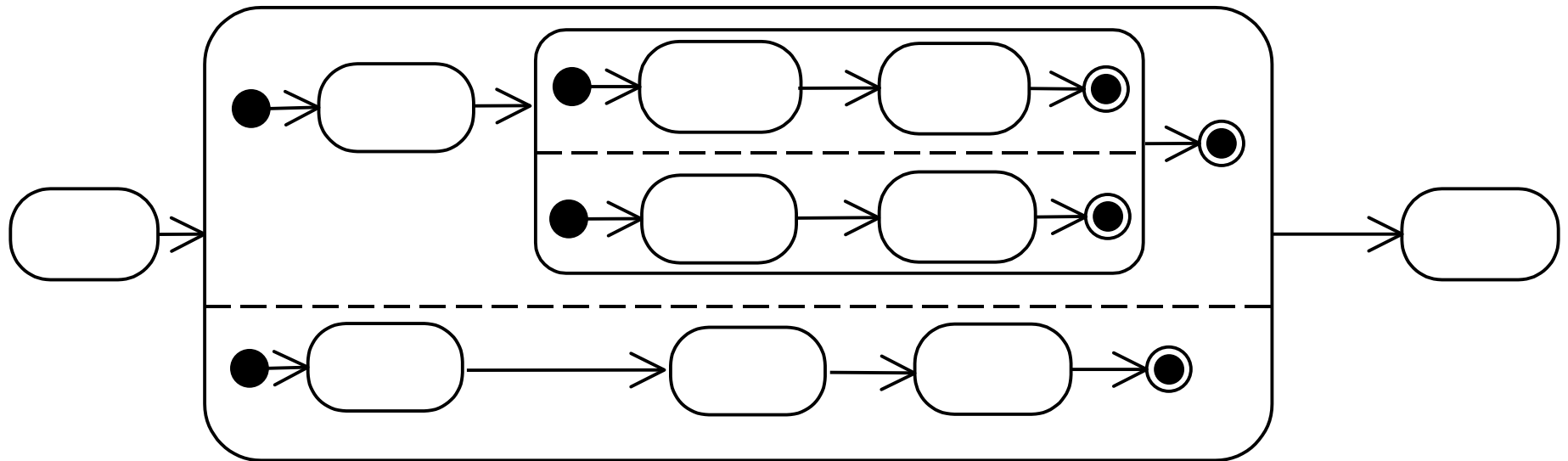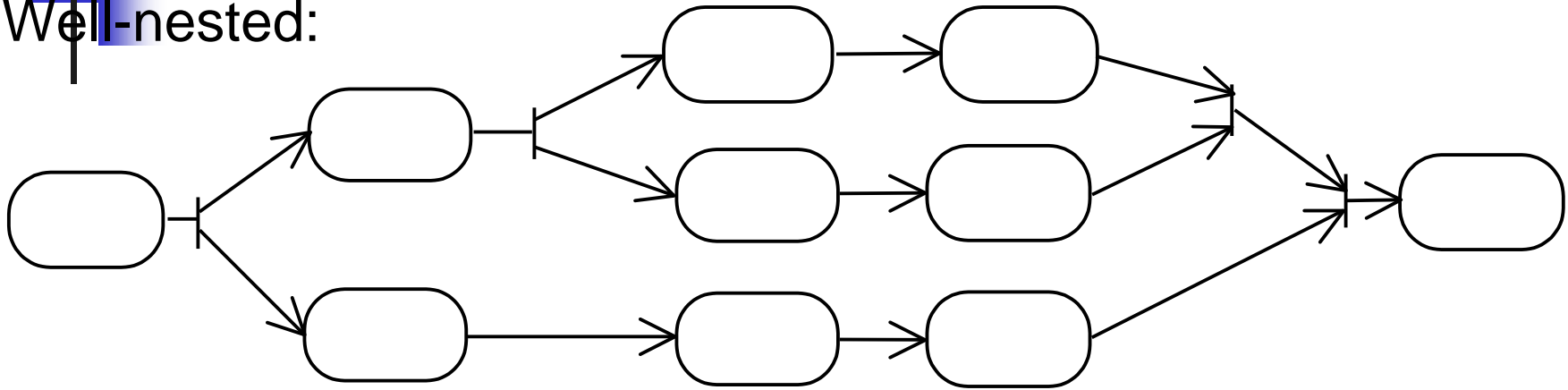
Activity Diagram Modeling Tips

Not well-nested:

Apply structured coding principles. (Be careful with goto's!)

Activity Diagram Modeling Tips

Can be translated to well-nested
diagram on earlier slide:

Wrap Up: Activity Diagrams

- Use Activity Diagrams for applications that are primarily control and data-driven, like business modeling ...

- ... rather than event-driven applications like embedded systems.

- Activity diagrams are a kind of state machine until UML 2.0 ...

- ... so control and object/data flow do not have separate semantics.

- UML 1.3 has new features for business modeling that increase power and convenience. Check it out and give feedback!

Preview - Next Tutorial

- **Advanced Modeling with UML**
  - Model management
  - Standard elements and profiles
  - Object Constraint Language (OCL)

Further Info

- Web:
  - UML 1.4 RTF: www.celigent.com/omg/umlrtf
  - OMG UML Tutorials: www.celigent.com/omg/umlrtf/tutorials.htm
  - UML 2.0 Working Group: www.celigent.com/omg/adptf/wgs/uml2wg.htm
  - OMG UML Resources: www.omg.org/uml/
- Email
  - Gunnar Övergaard: gunnaro@it.kth.se
  - Bran Selic: bran@objectime.com
  - Conrad Bock: conrad.bock@kabira.com
- Conferences & workshops
  - OMG UML Workshop: *UML in the .com Enterprise*, Palm Springs, California, Nov. 2000
  - UML World 2001, location and dates TBA
  - UML 2001, Toronto, Canada, Oct. 2001