

# Object Modeling with OMG UML Tutorial Series

## Behavioral Modeling

Gunnar Övergaard, Bran Selic, Conrad  
Bock and Morgan Björkander

UML Revision Task Force



© 1999-2001 OMG and Contributors: Crossmeta, EDS, IBM, Enea Data, Hewlett-Packard, IntelliCorp, Kabira Technologies, Klasse Objecten, Rational Software, Telelogic, Unisys



# Overview

---

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



# Tutorial Series

---

- Lecture 1: Introduction to UML: Structural and Use Case Modeling
- Lecture 2: Behavioral Modeling with UML
- Lecture 3: Advanced Modeling with UML

[Note: This version of the tutorial series is based on *OMG UML Specification v. 1.4*, UML Revision Task Force recommended final draft, OMG doc# ad/01-02-13.]



# 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



# Quick Tour

---

- The UML is a graphical language for
  - specifying
  - visualizing
  - constructing
  - documentingthe 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, adopted in November 1999
- Next minor revision will be UML 1.4, planned to be adopted in Q2 2001
- Next major revision will be UML 2.0, planned to be completed in 2002

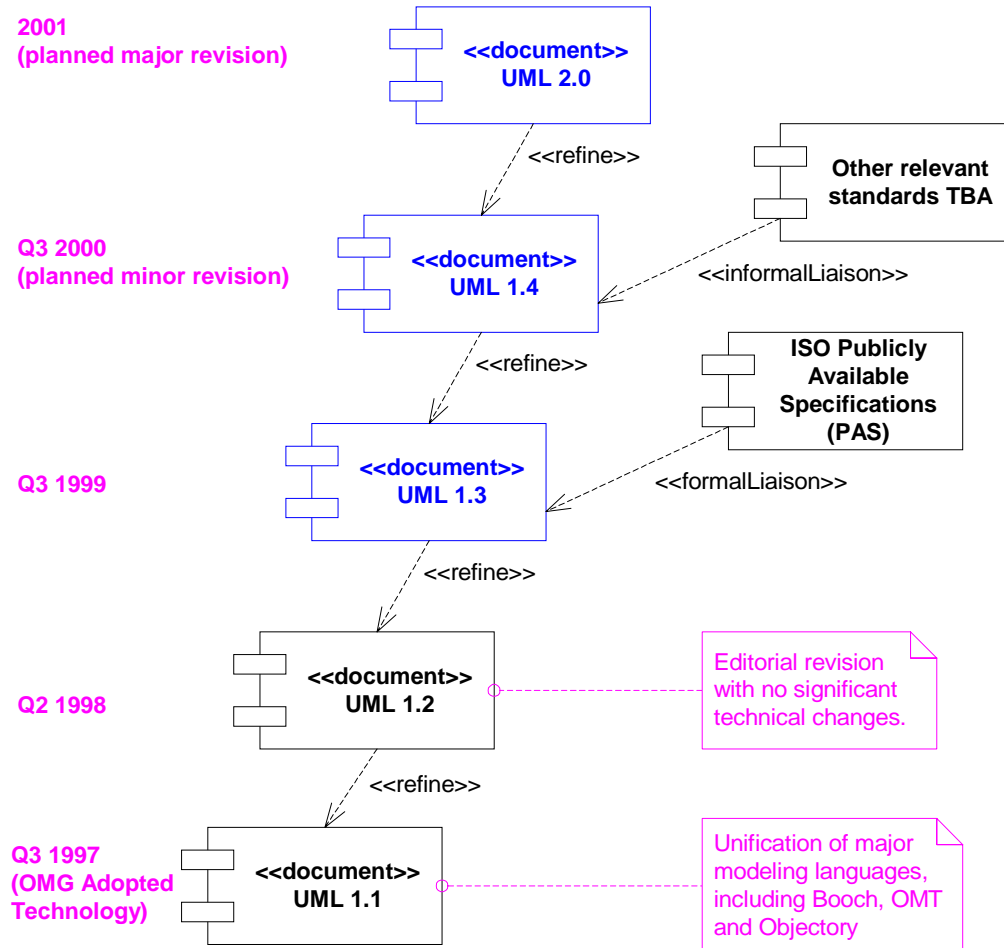


# UML Goals

---

- 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





# 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

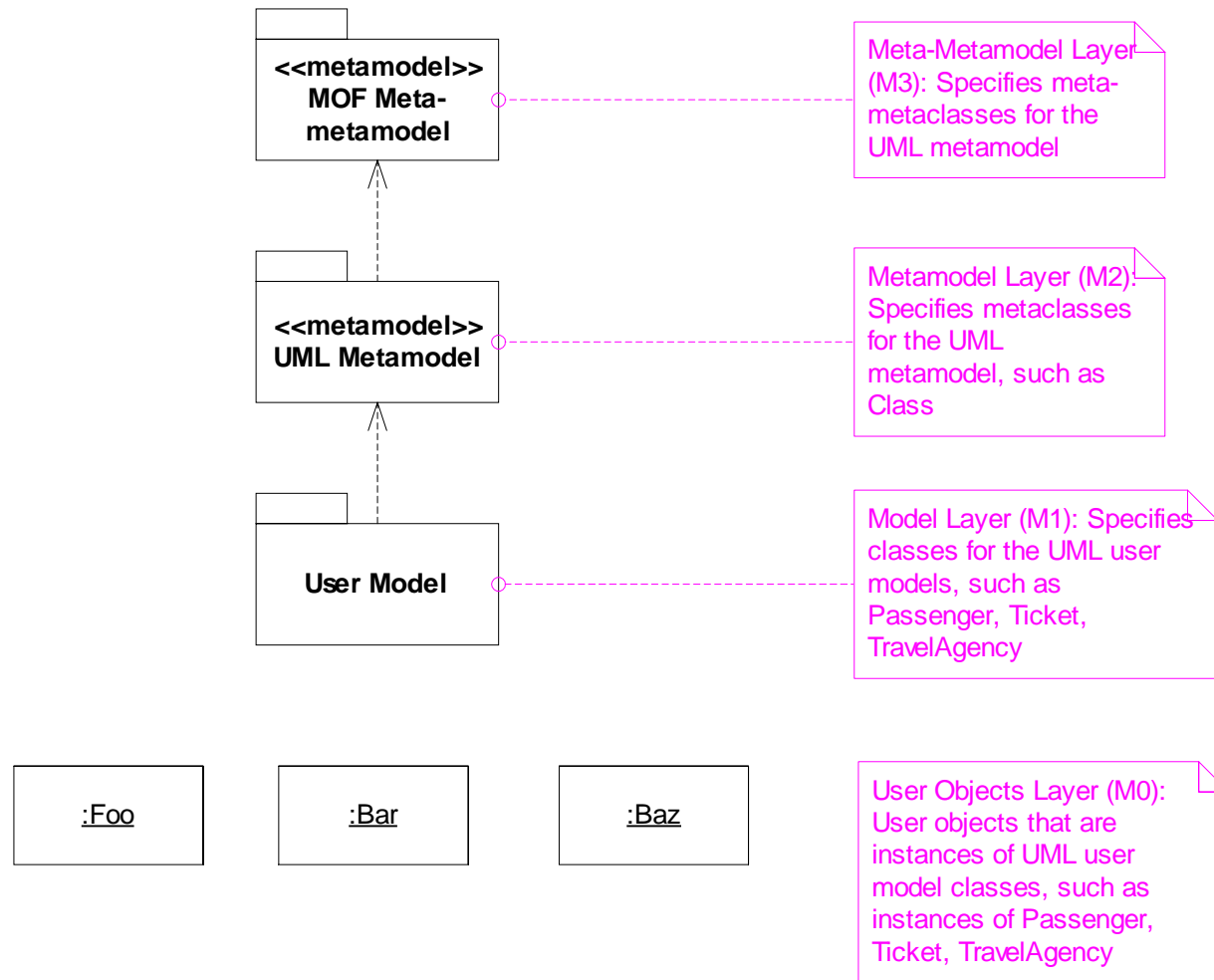


# Language Architecture

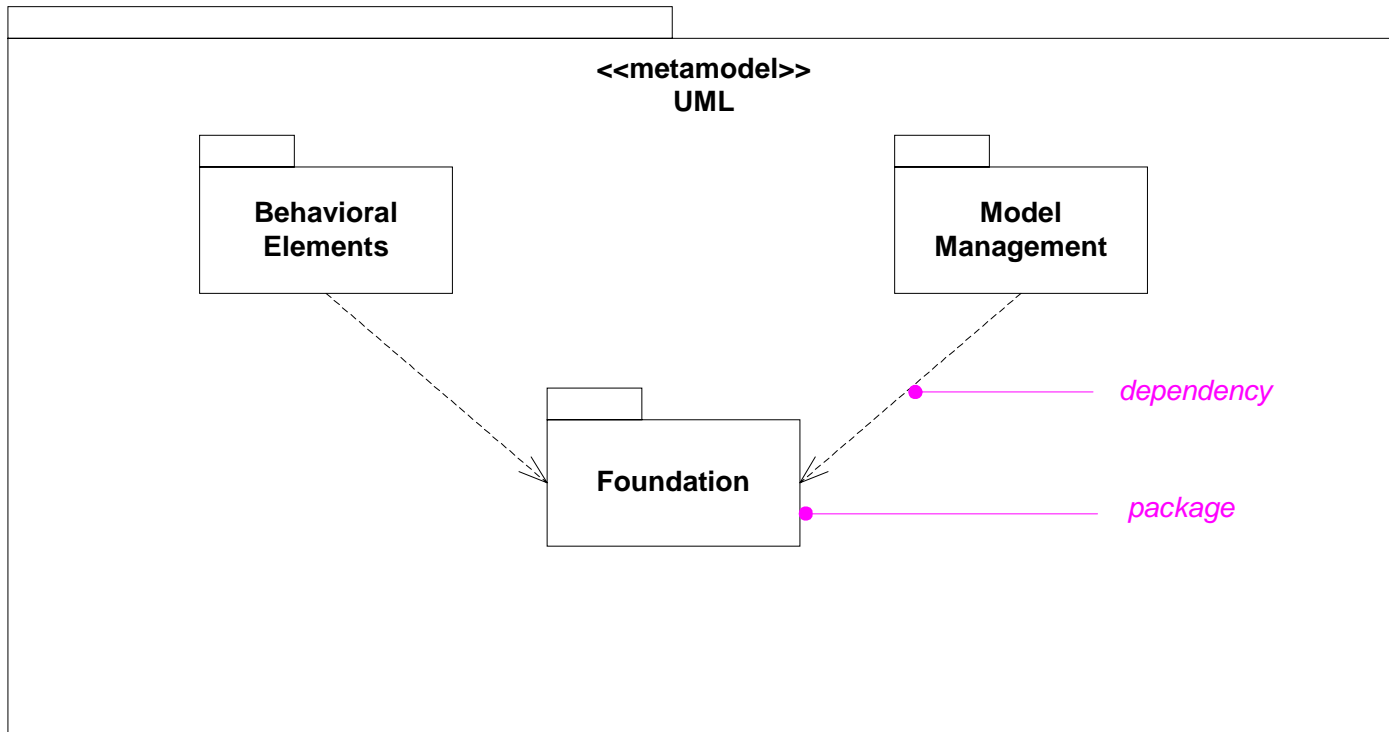
---

- Metamodel architecture
- Package structure

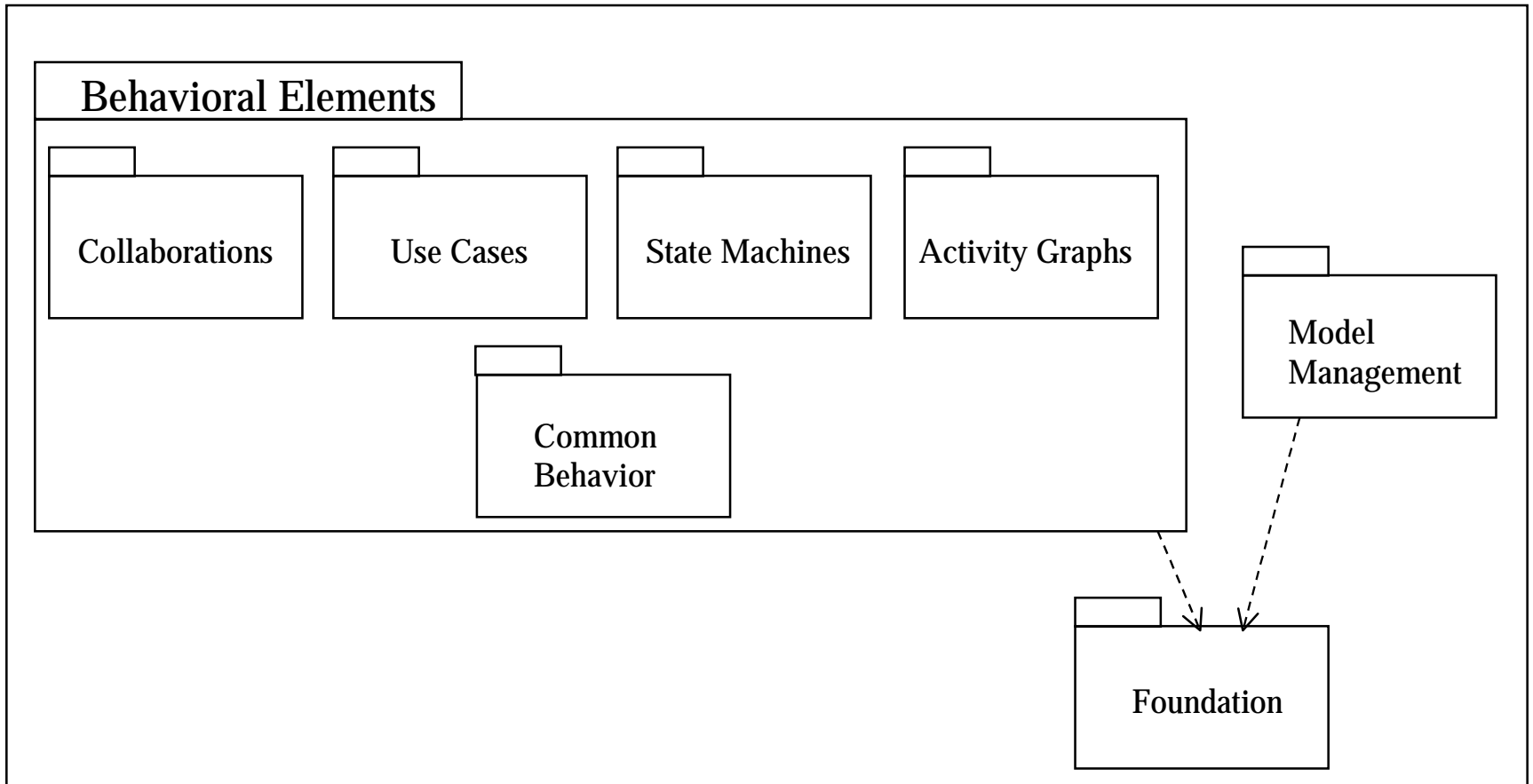
# Metamodel Architecture



# 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

<b>Construct</b>	<b>Description</b>	<b>Syntax</b>		
<b>Instance</b> (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).	<table border="1" data-bbox="1420 462 1648 625"><tr><td data-bbox="1420 462 1648 544"><u>name</u></td></tr><tr><td data-bbox="1420 544 1648 625">attr values</td></tr></table>	<u>name</u>	attr values
<u>name</u>				
attr values				
<b>Action</b>	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)

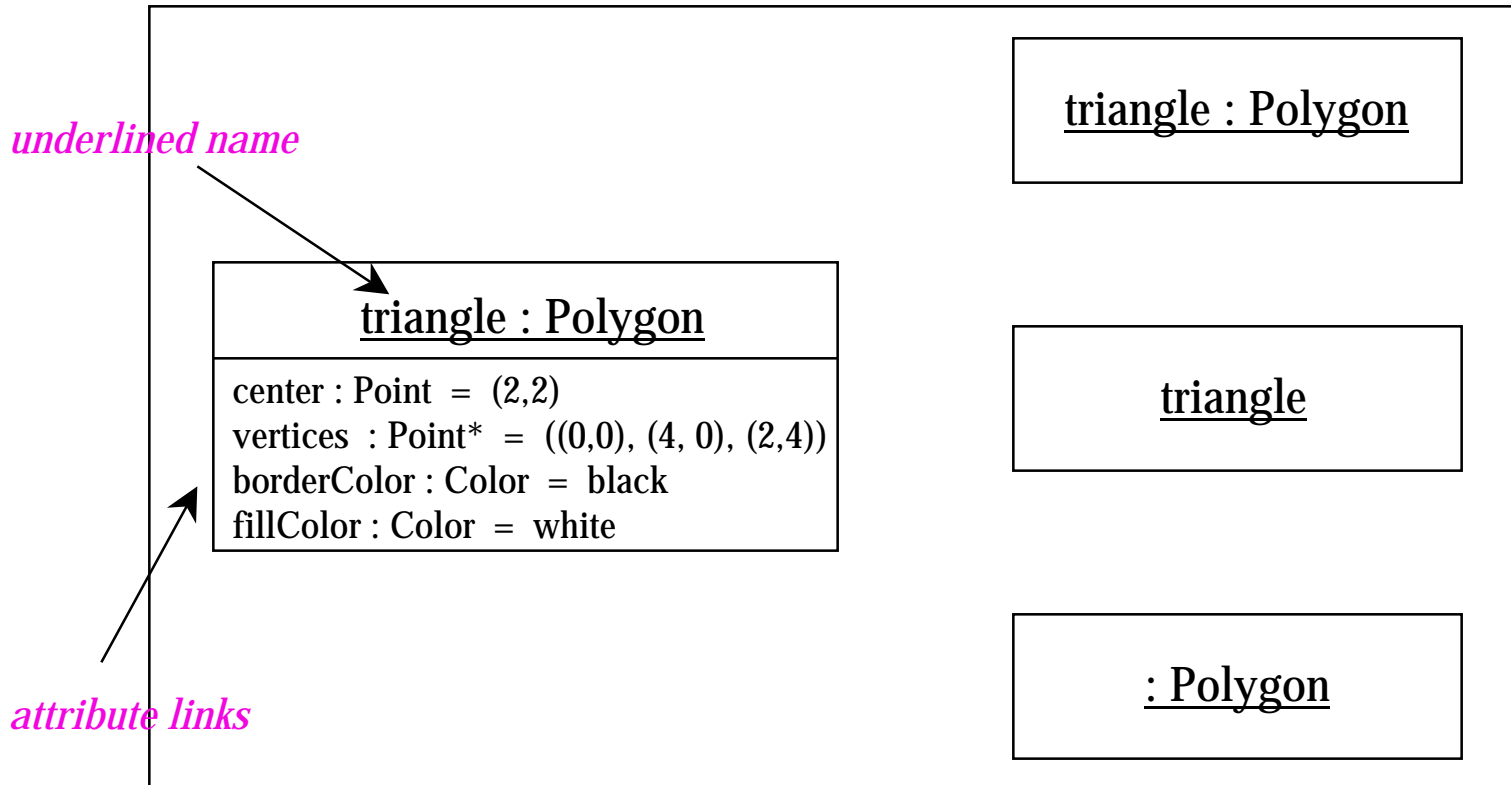
<b>Construct</b>	<b>Description</b>	<b>Syntax</b>		
<b>Stimulus</b>	A communication between two instances.			
<b>Operation</b>	A declaration of a service that can be requested from an instance to effect behavior.	textual		
<b>Signal</b>	A specification of an asynchronous stimulus communicated between instances.	<table border="1"><tr><td>«Signal» Name</td></tr><tr><td>parameters</td></tr></table>	«Signal» Name	parameters
«Signal» Name				
parameters				



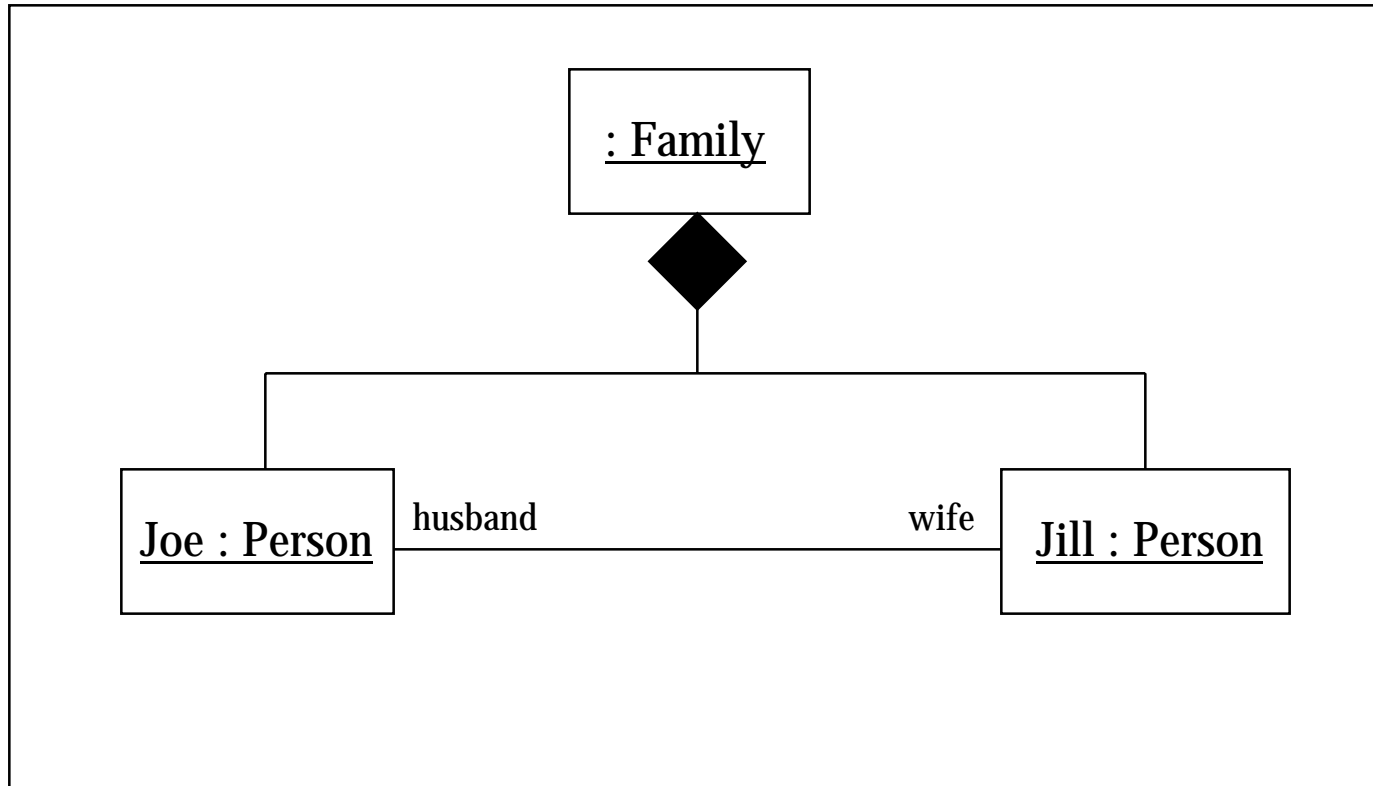
# *Interaction: Core Relationships*

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

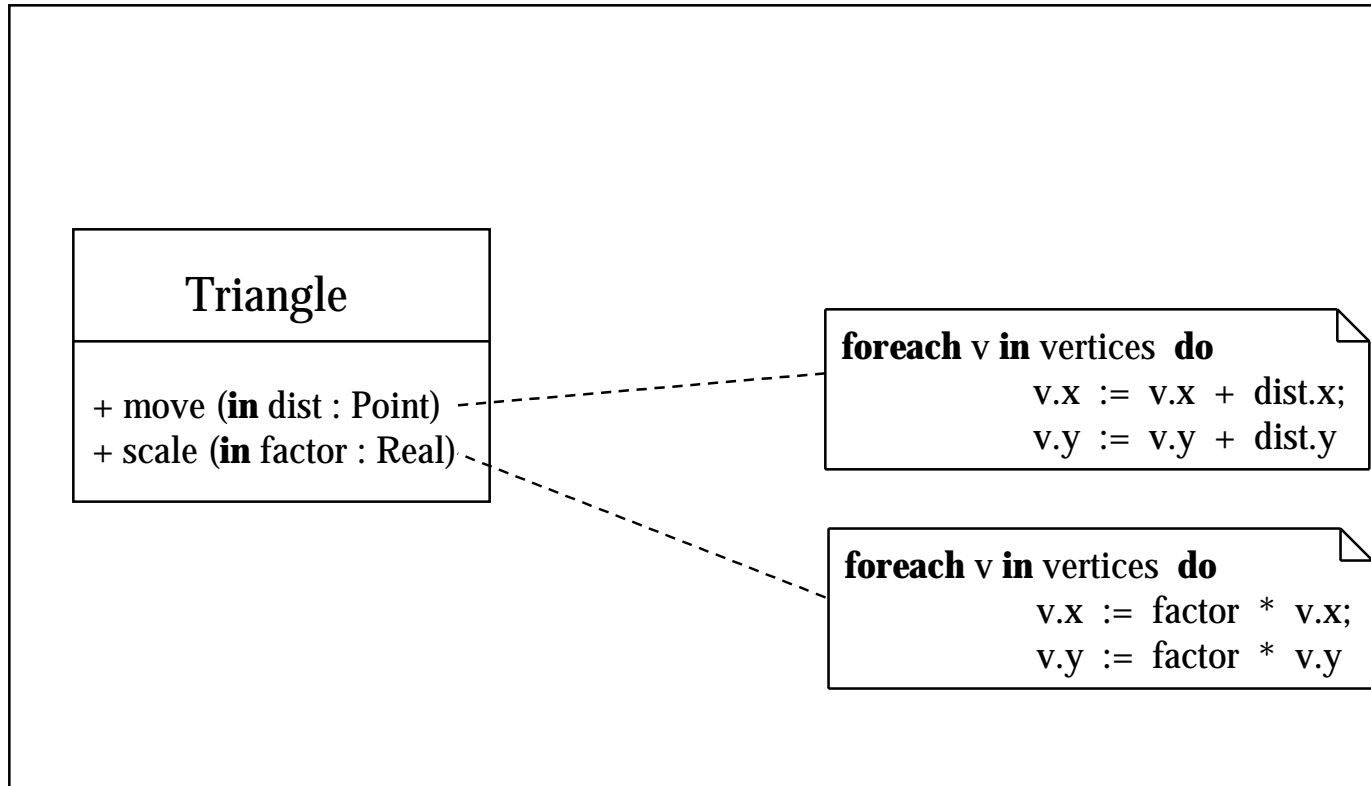
# Example: Instance



# Example: Instances and Links



# Operation and Method



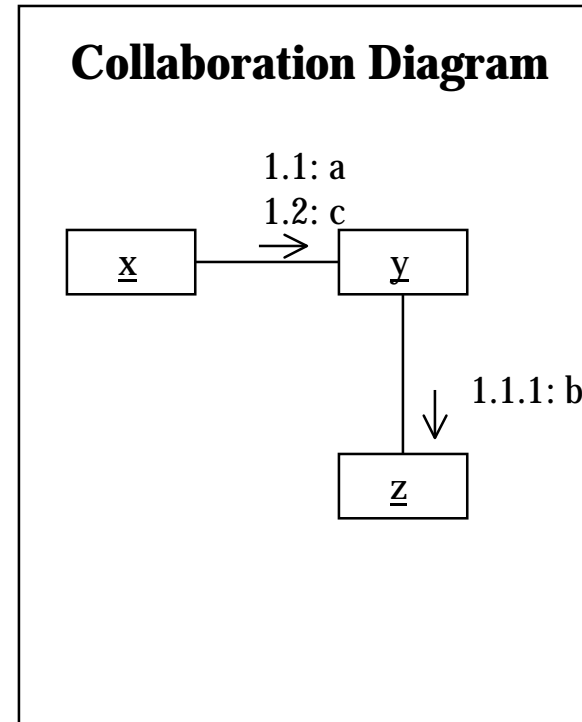
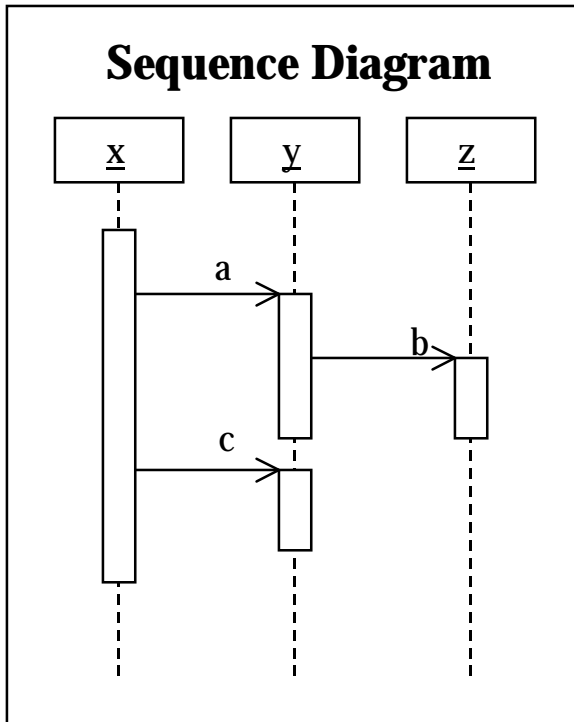


# Interaction Diagram Tour

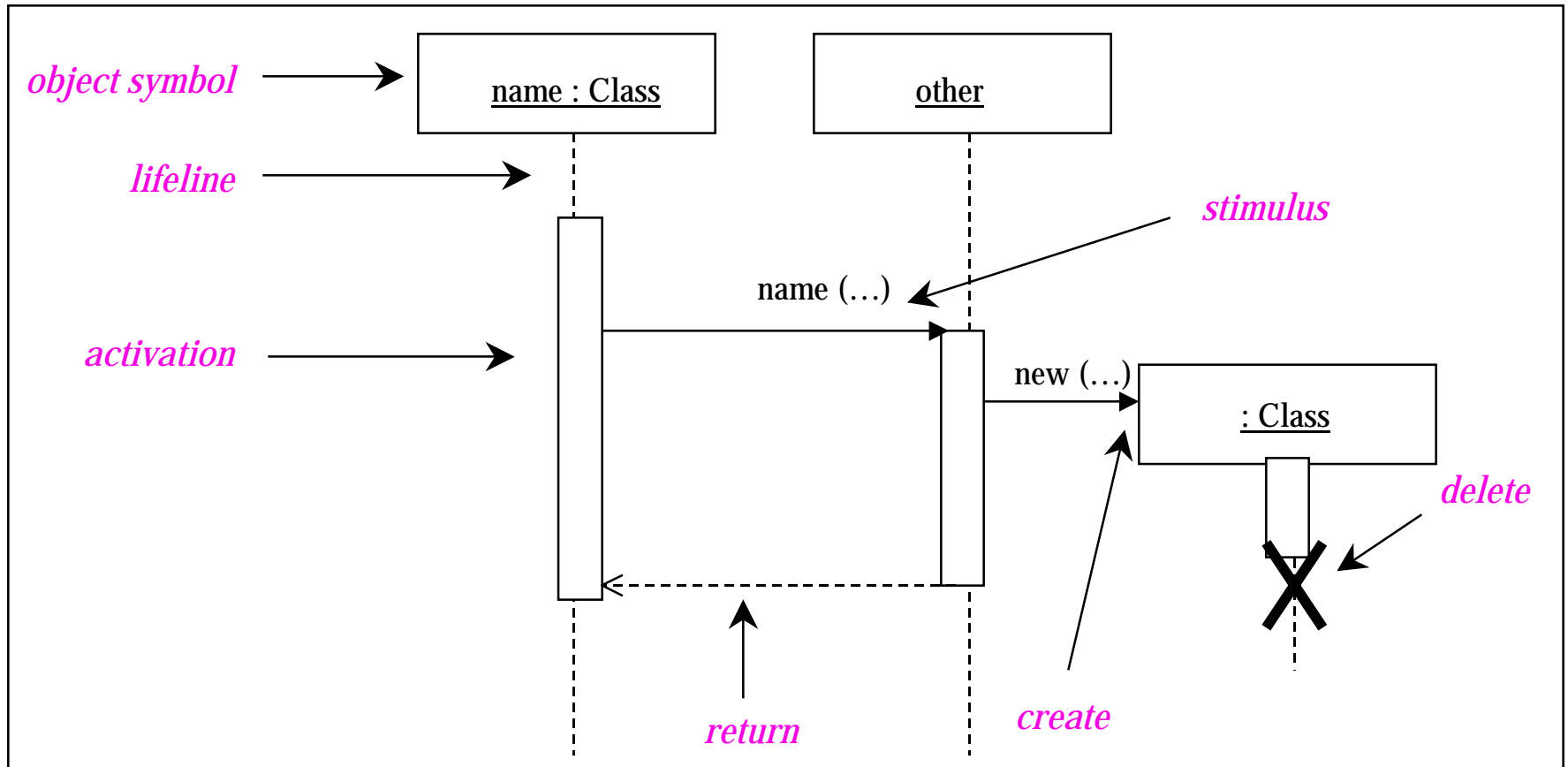
---

- 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

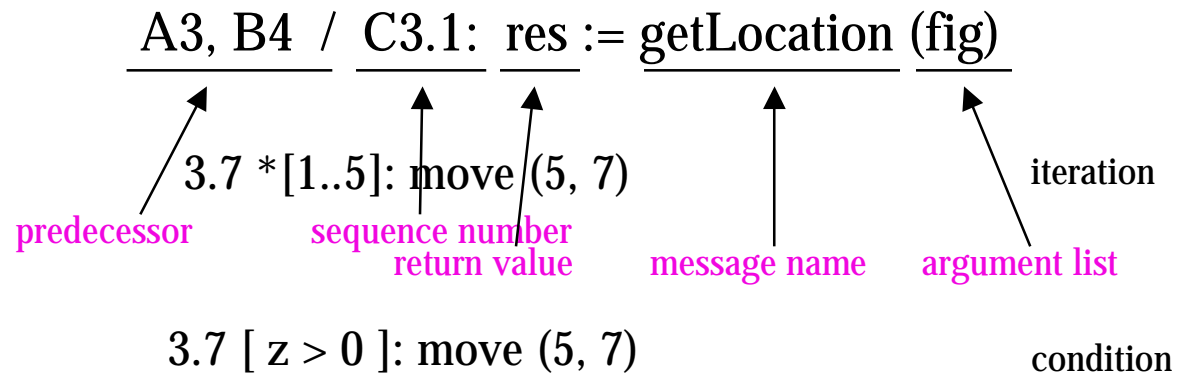


# Arrow Label

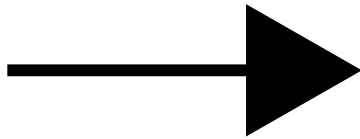
*predecessor sequence-expression return-value := message-name argument-list*

move (5, 7)

3.7.4: move (5, 7)



# Different Kinds of Arrows



Procedure call or other  
kind of nested flow of  
control

UML 1.4: Asynchronous



~~Flat flow of control~~



~~Explicit asynchronous  
flow of control~~



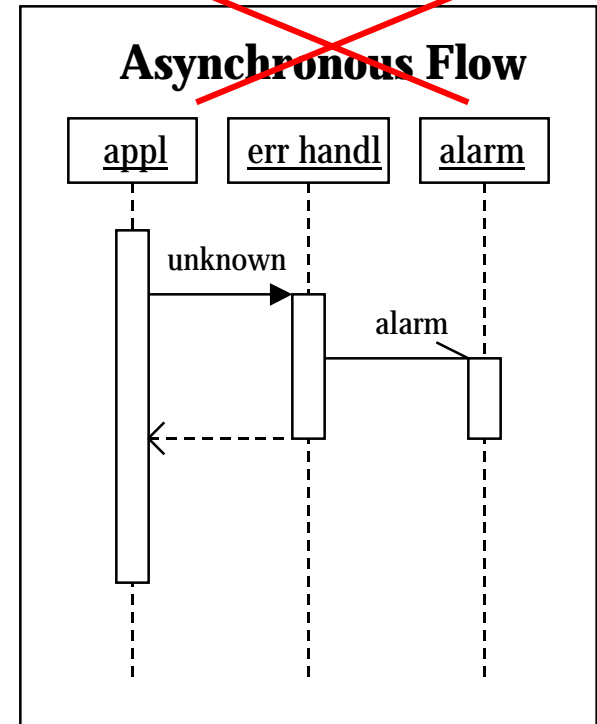
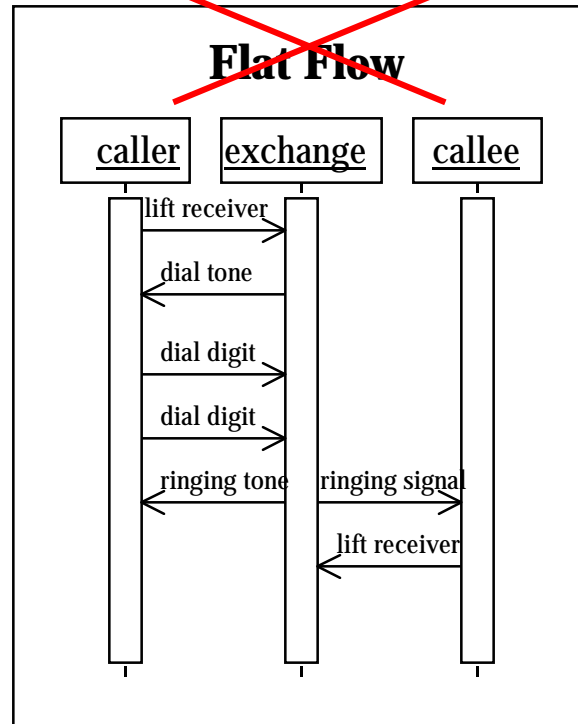
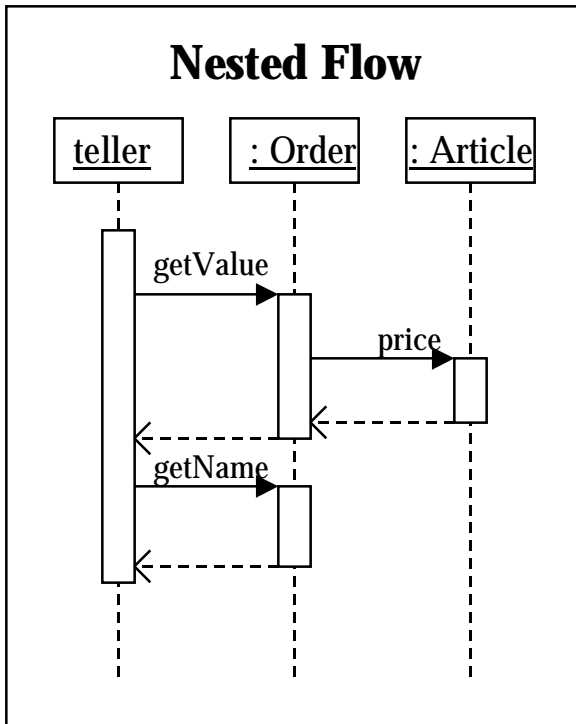
UML 1.4: Variant of async

Return

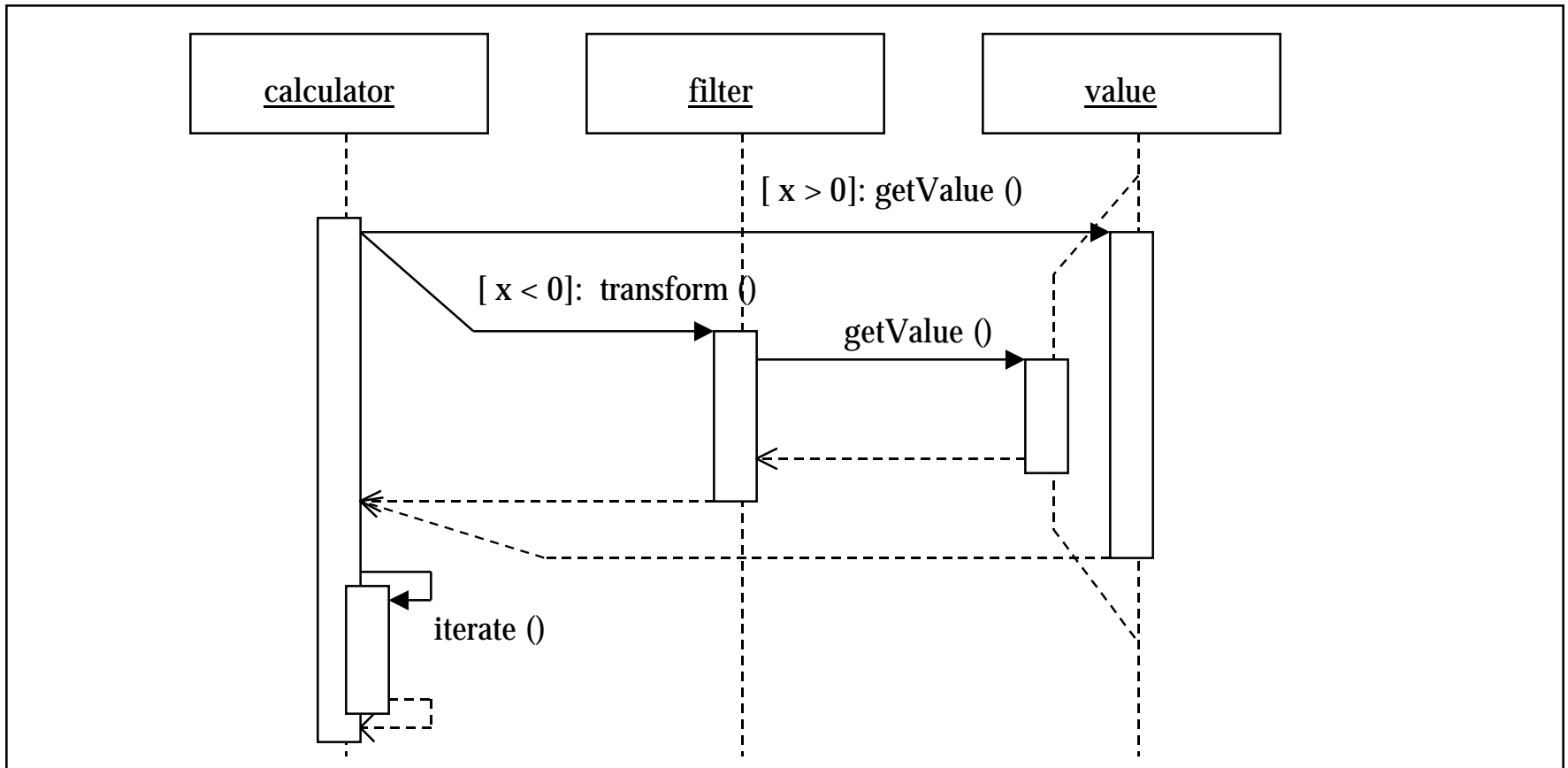
# Example: Different Arrows

UML 1.4: Alternative notation

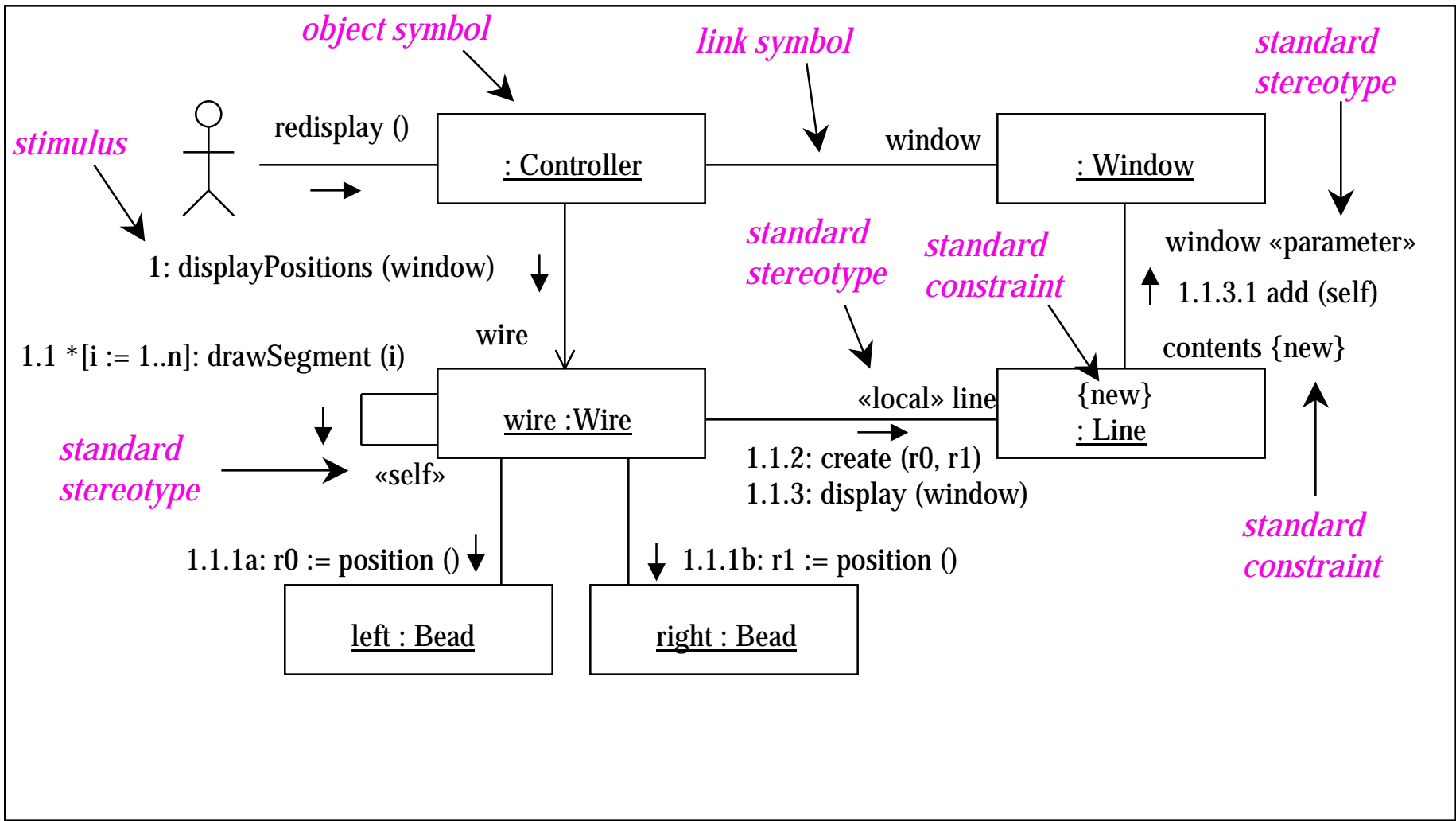
UML 1.4: Asynchronous flow for async flow



# Recursion, Condition, etc.



# Collaboration Diagram





# When to Model Interactions

---

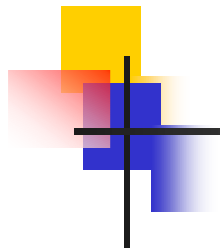
- To specify how the instances are to interact with each other.
- To identify the interfaces of the classifiers.
- To distribute the requirements.



# 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

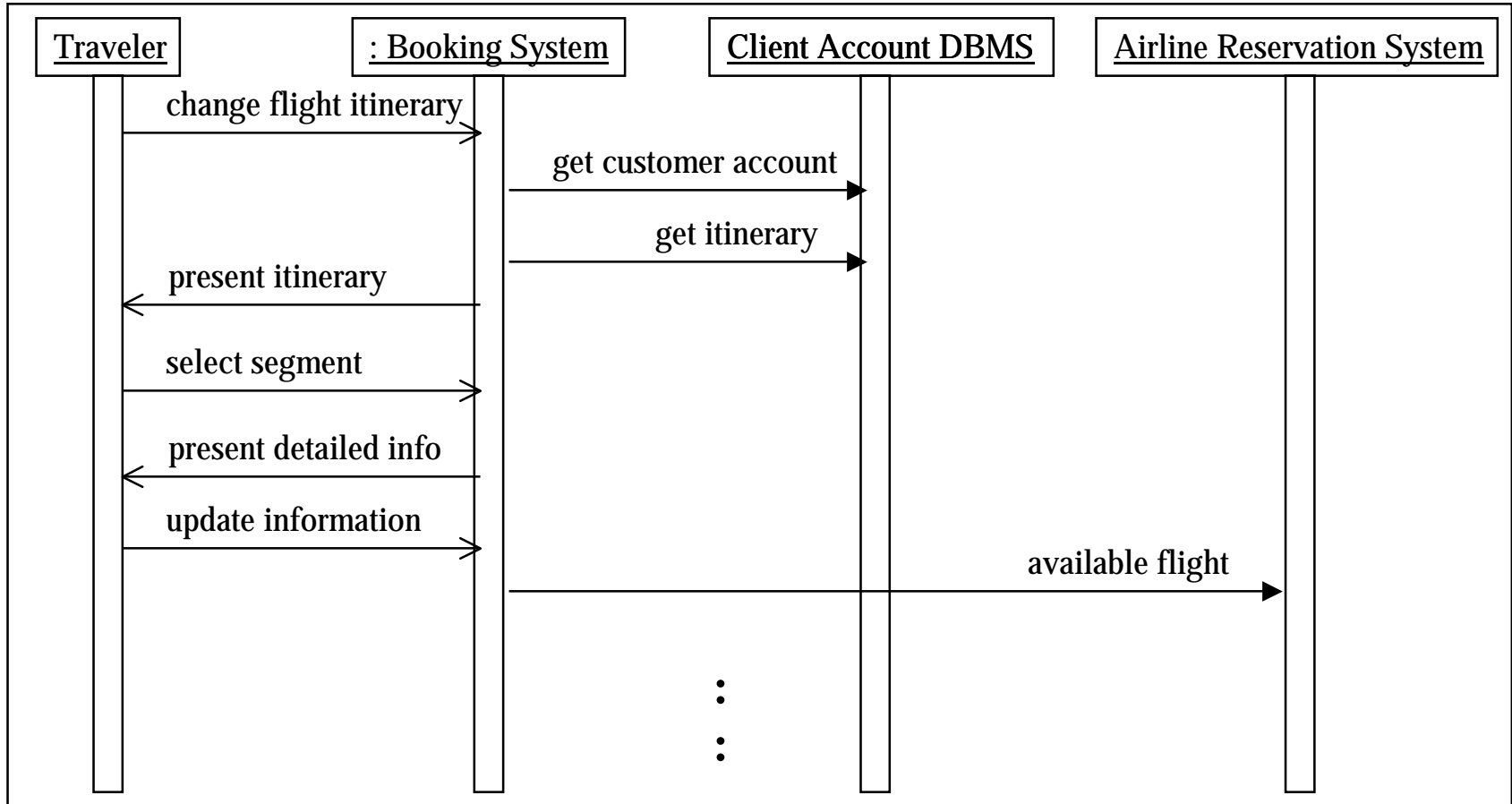


# Use Case Description: Change Flt Itinerary

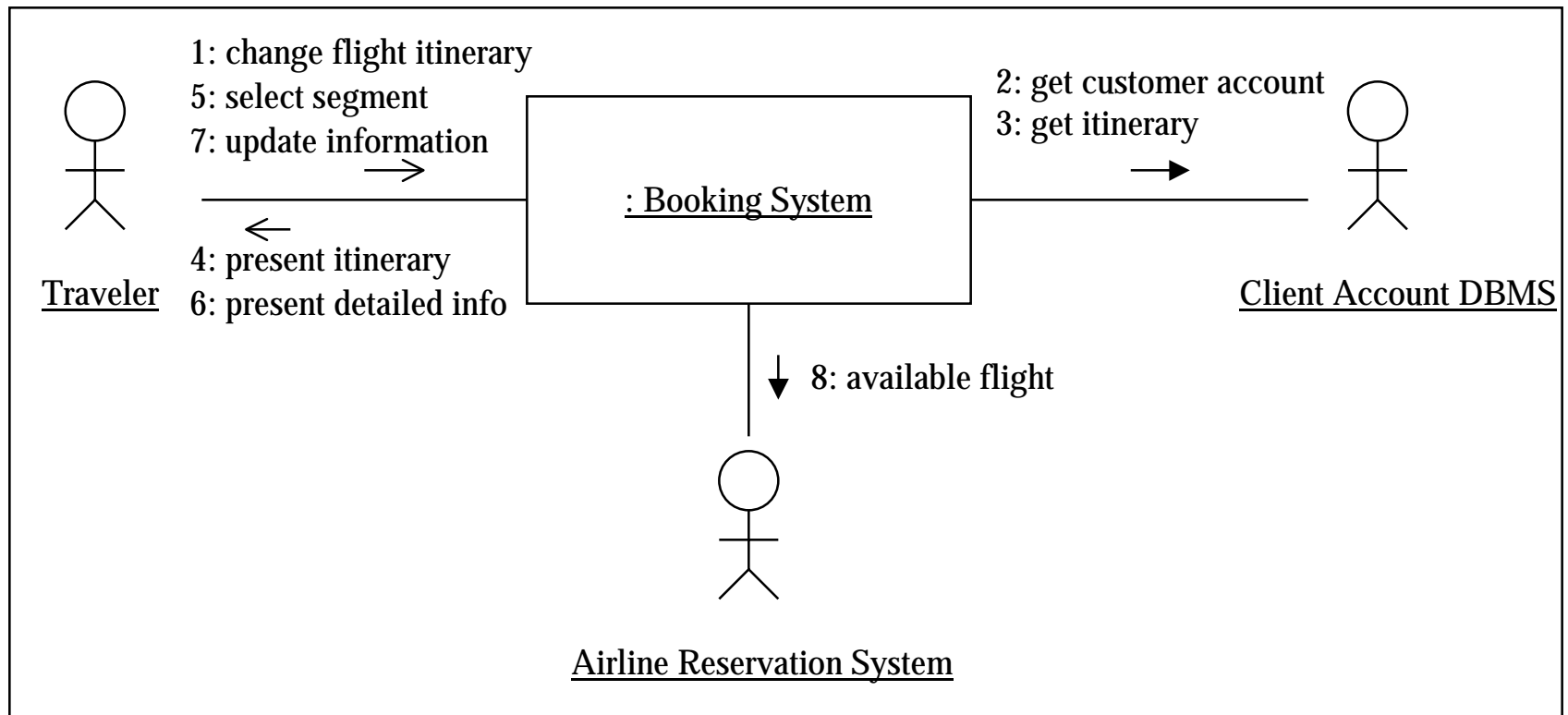
---

- **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



# Collaboration Diagram: Change Flt Itinerary





# 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

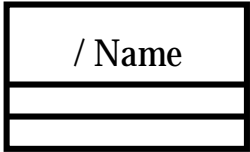
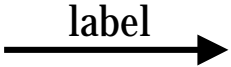
<b>Construct</b>	<b>Description</b>	<b>Syntax</b>
<b>Collaboration</b>	<p>A collaboration describes how an operation or a classifier, like a use case, is realized by a set of classifiers and associations used in a specific way.</p> <p>The collaboration defines a set of roles to be played by instances and links, possibly including a collection of interactions.</p>	
<b>Interaction</b>	<p>An interaction describes a communication pattern between instances when they play the roles of the collaboration.</p>	

# Collaborations: Core Elements (cont'd)



<b>Construct</b>	<b>Description</b>	<b>Syntax</b>
<b>Collaboration-Instance</b>	A collection of instances which together play the roles declared in a collaboration.	
<b>Interaction-Instance</b>	A collection of stimuli exchanged between instances playing specific roles according to the communication pattern of an interaction.	

All new in UML 1.4

# Collaborations: Core Elements (cont'd)

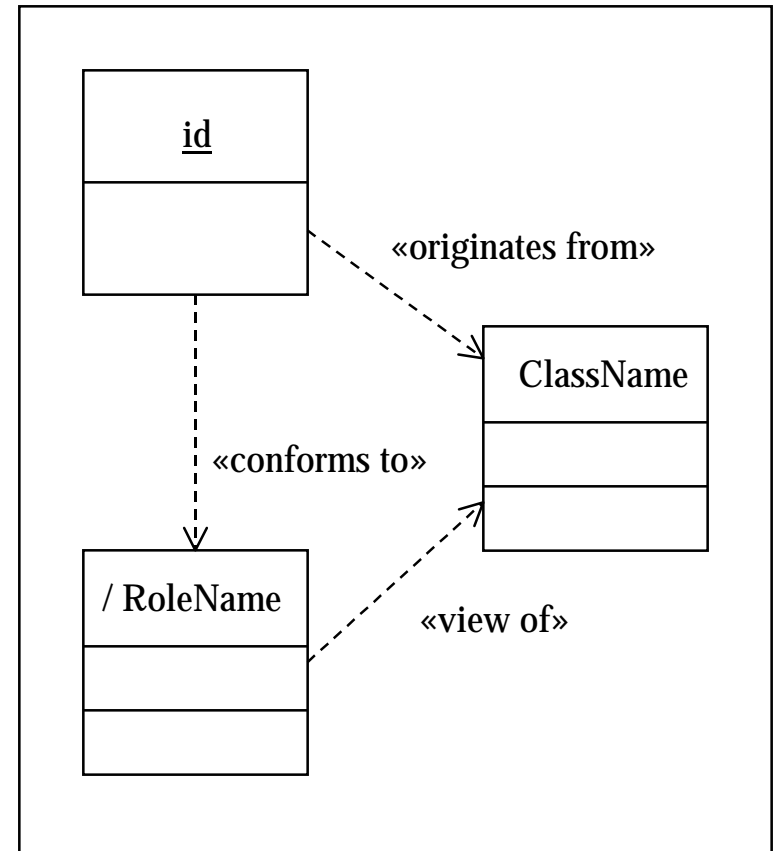
<b>Construct</b>	<b>Description</b>	<b>Syntax</b>
<b>Classifier Role</b>	A classifier role is a specific role played by a participant in a collaboration. It specifies a restricted view of a classifier, defined by what is required in the collaboration.	 A rectangular box representing a classifier role. The top portion of the box contains the text "/ Name". Below this text are two horizontal lines, representing the role's restricted view.
<b>Message</b>	A message specifies one communication between instances. It is a part of the communication pattern given by an interaction.	 A horizontal arrow pointing to the right. The word "label" is positioned above the arrow, representing the message's content.

# Collaborations: Core Relationships

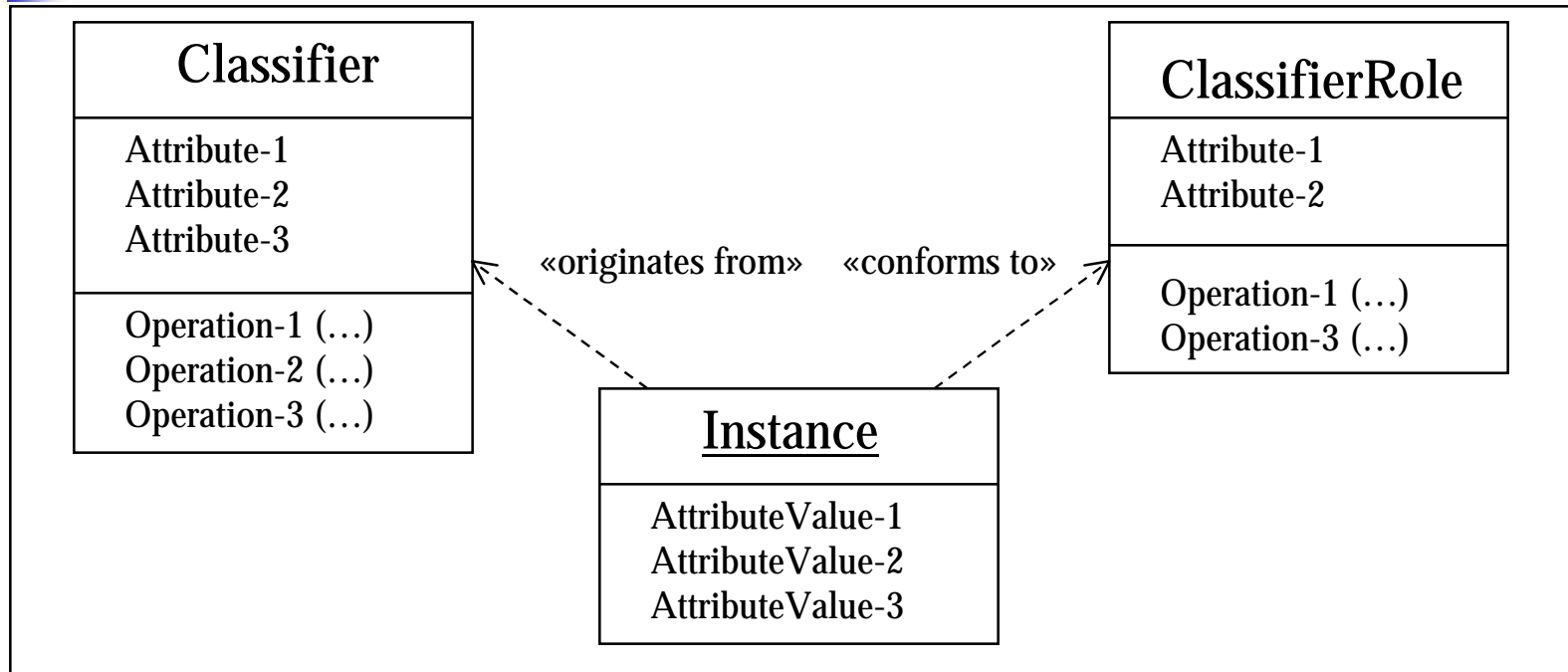
<b>Construct</b>	<b>Description</b>	<b>Syntax</b>
<b>Association Role</b>	An association role is a specific usage of an association needed in a collaboration.	
<b>Generalization</b>	A generalization is a taxonomic relationship between a more general element and a more specific element. The more specific 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 unique identity.
- A Classifier is a description of an Instance.
- A Classifier Role defines a usage (an abstraction) of an Instance.



# Classifier-Instance-Role Trichotomy (cont'd)



- 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 preceded by a '/'*

*A classifier name is preceded by a ':'*

Example:

/ Parent : Person

/ Parent

: Person

instanceName / ClassifierRoleName : ClassifierName

Example:

: Person

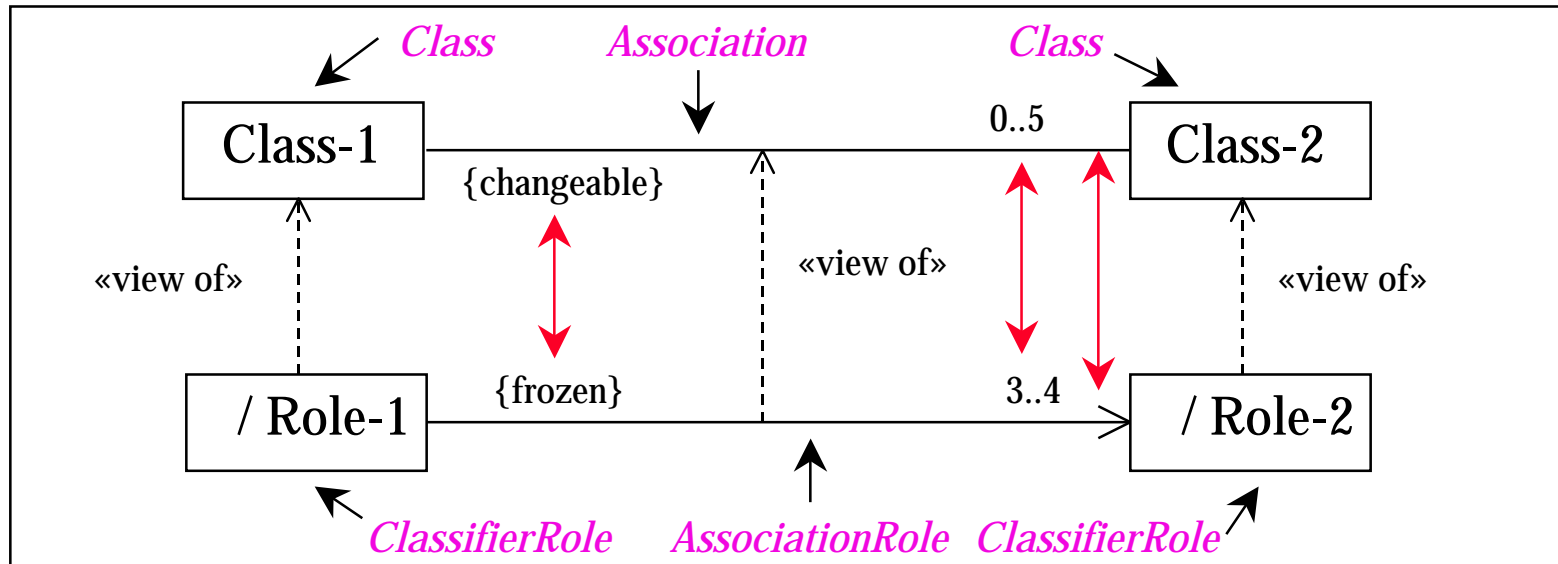
Charlie

Charlie : Person

Charlie / Parent

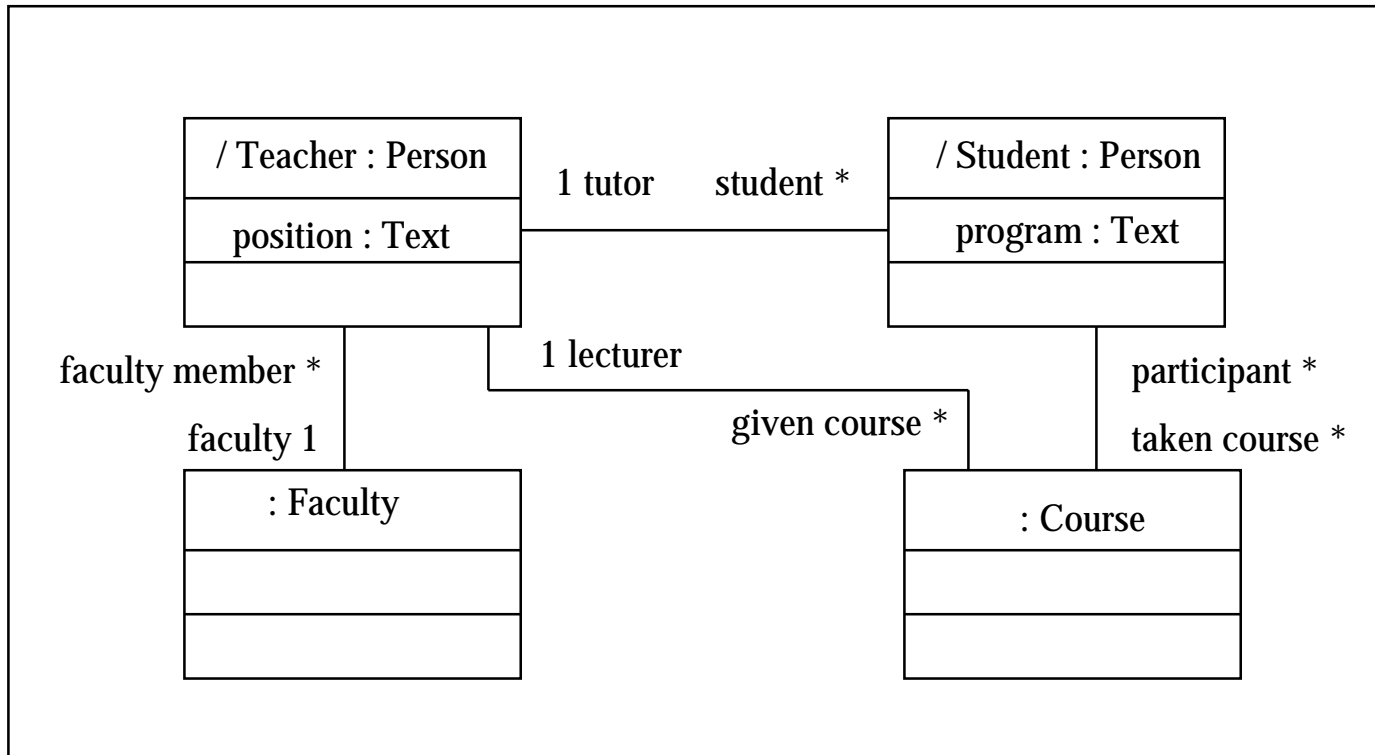
Charlie / Parent : Person

# Association and Association Role



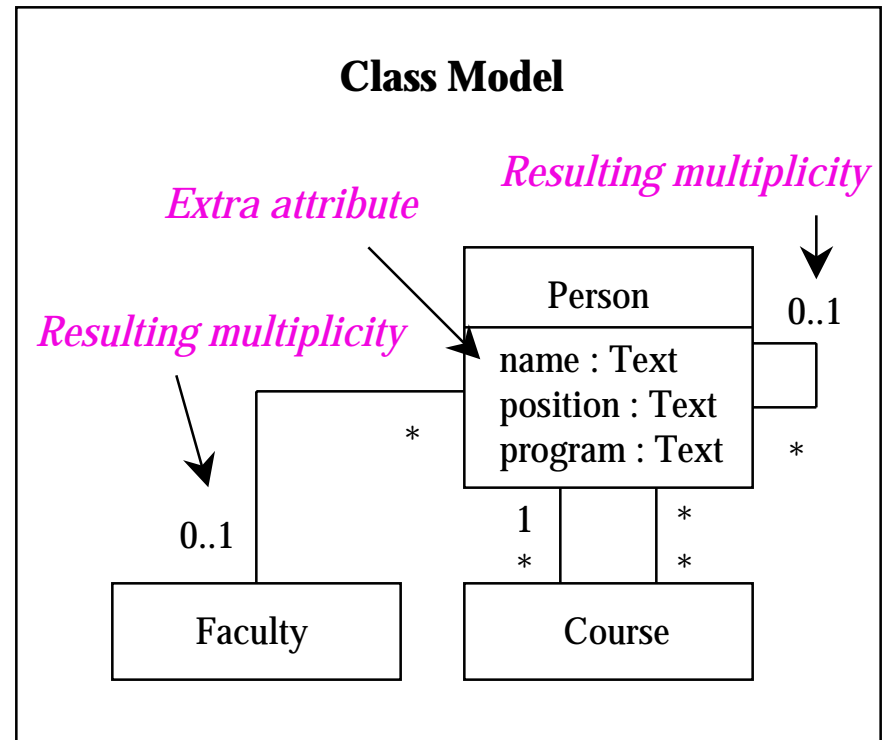
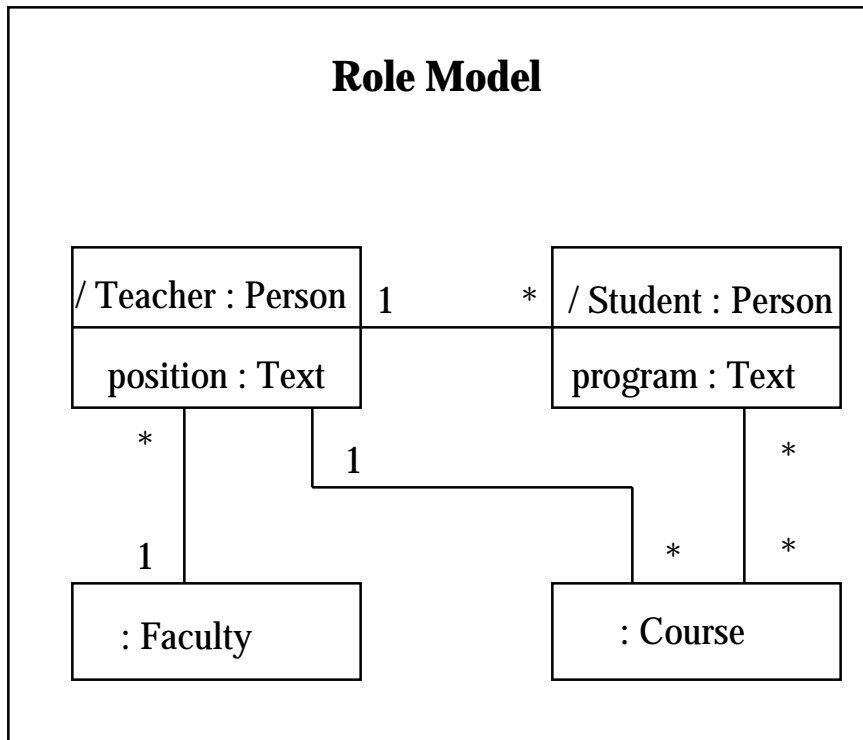
- An Association Role specifies the required properties of a Link used in a Collaboration.
- The properties of an AssociationEnd may be restricted by an AssociationEndRole.

# Example: A School



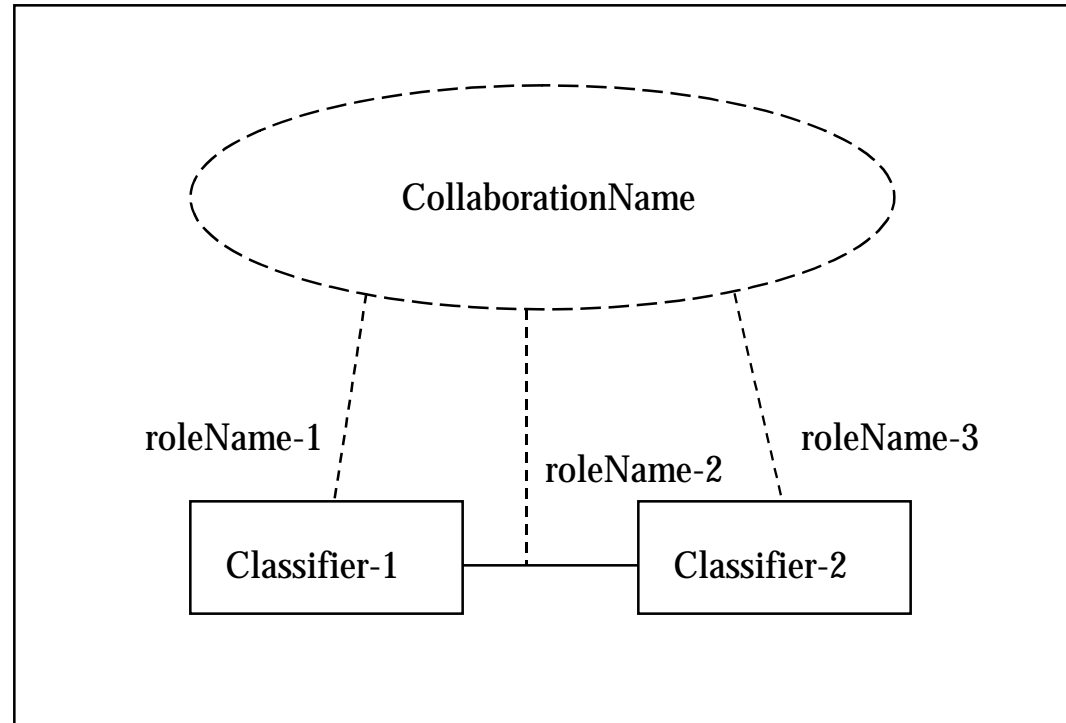
# Role Model vs. Class Model

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

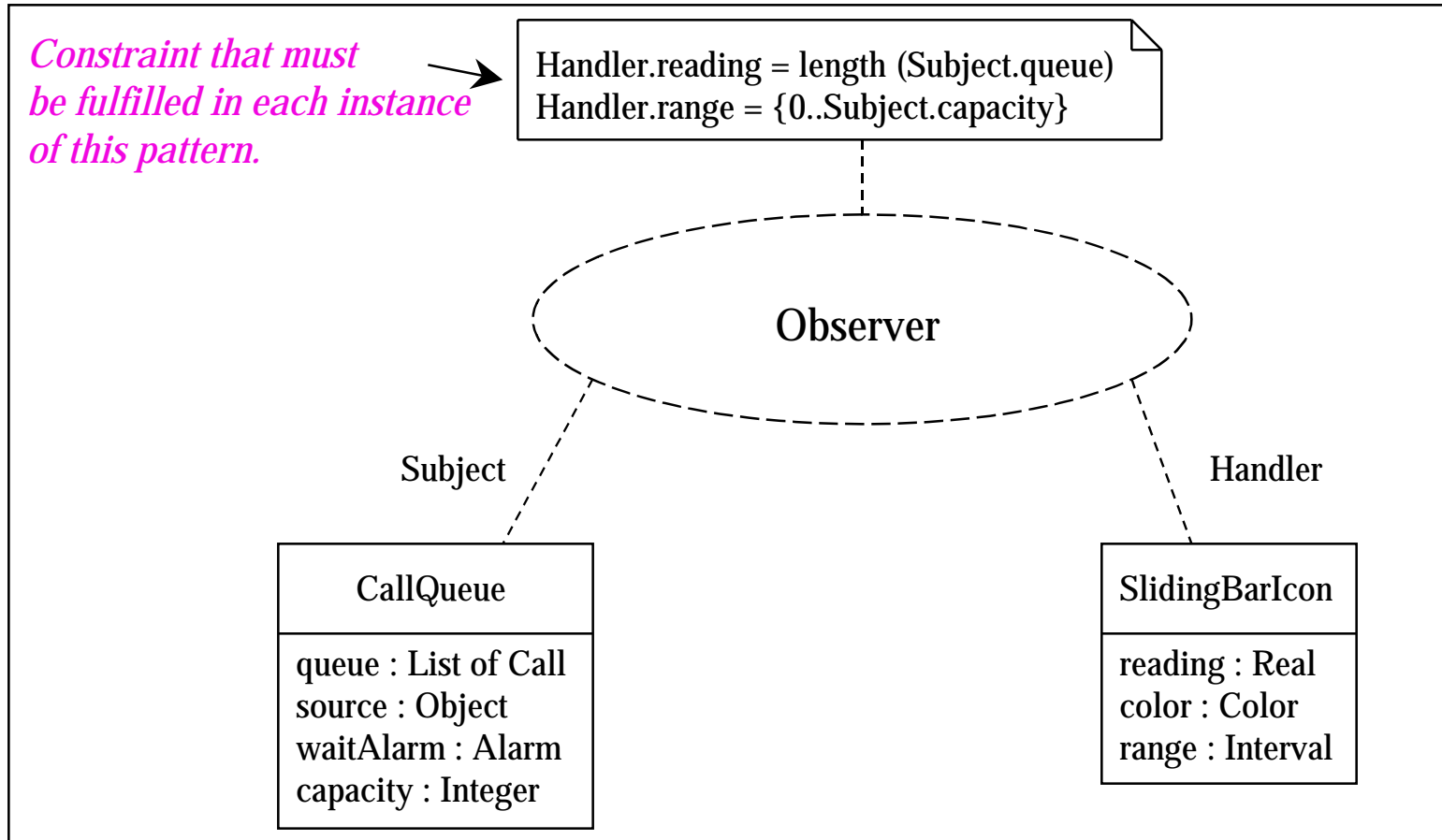


# A Collaboration and Its Roles

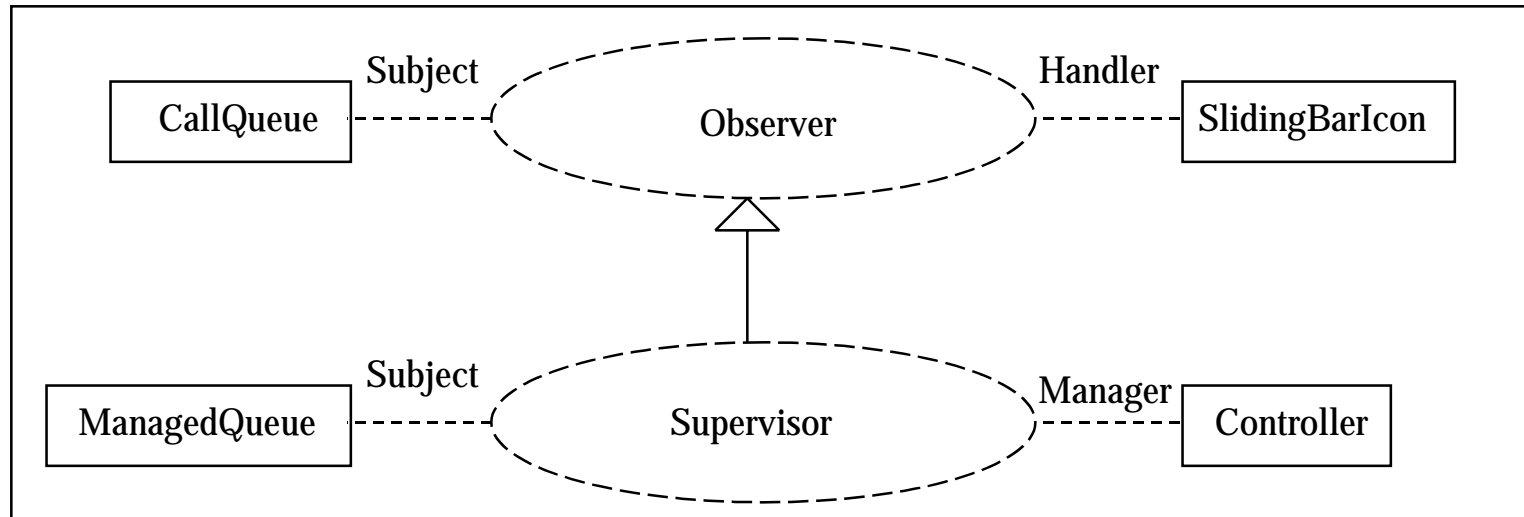
A Collaboration and how its roles are mapped onto a collection of Classifiers and Associations.



# Patterns in UML



# 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.

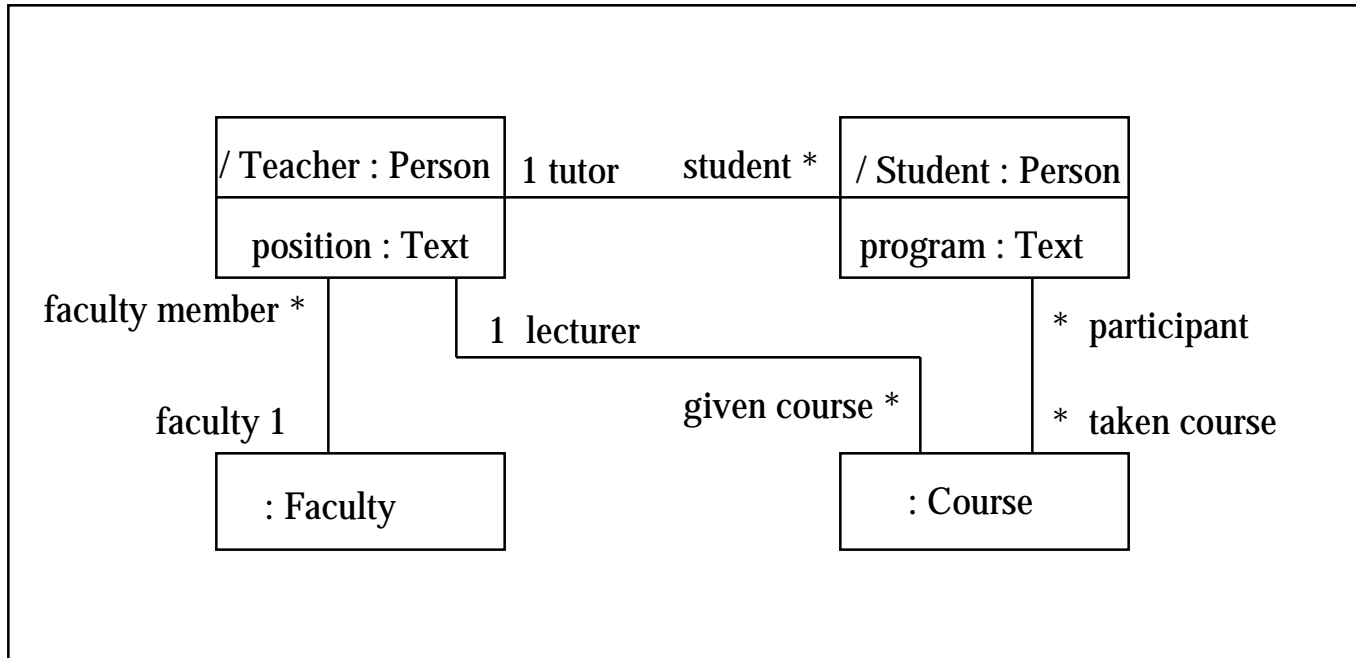


# Collaboration Diagram Tour

---

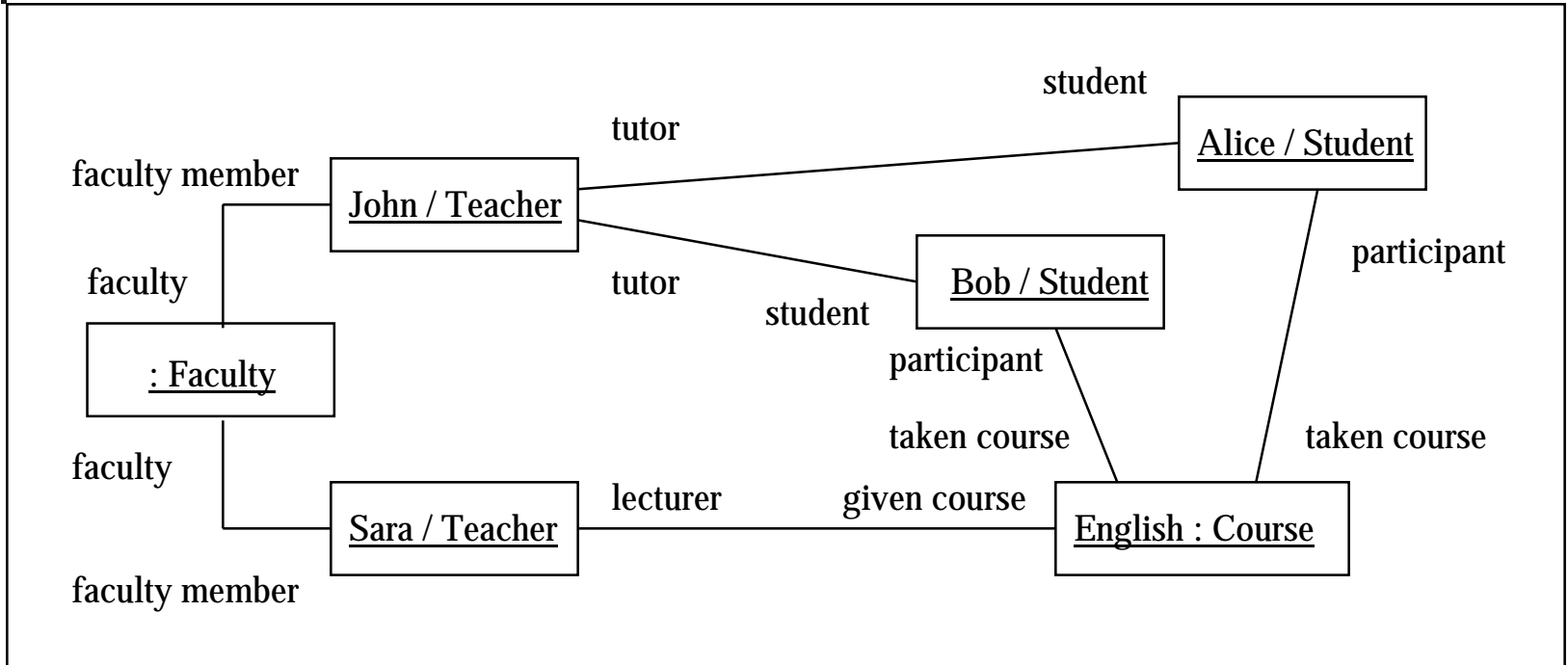
- Show Classifier Roles and Association Roles, possibly together with extra constraining elements
- Kinds
  - Instance level – Instances and Links
  - Specification level – Roles
- Static Diagrams are used for showing Collaborations explicitly

# Collaboration Diagram at Specification Level



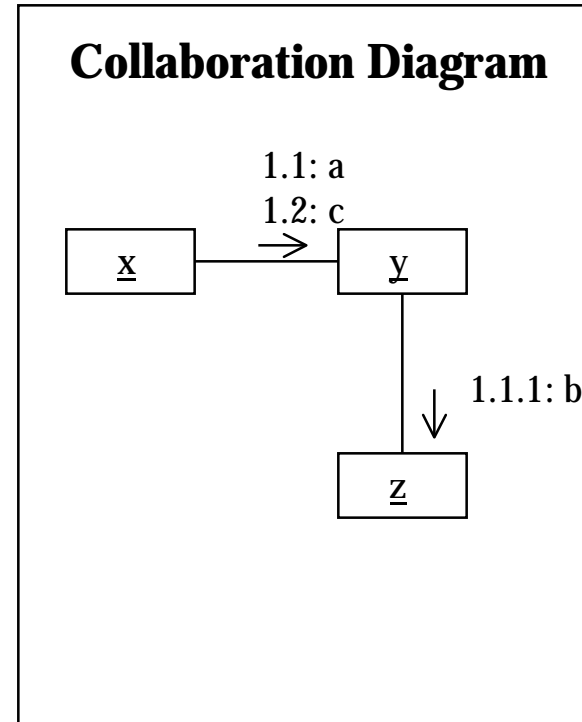
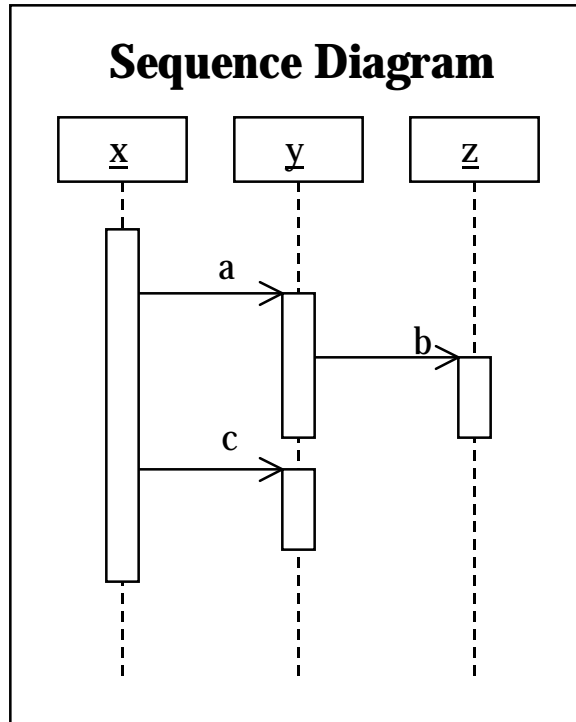
UML 1.4: The diagram shows the contents of a Collaboration

# Collaboration Diagram at Instance Level



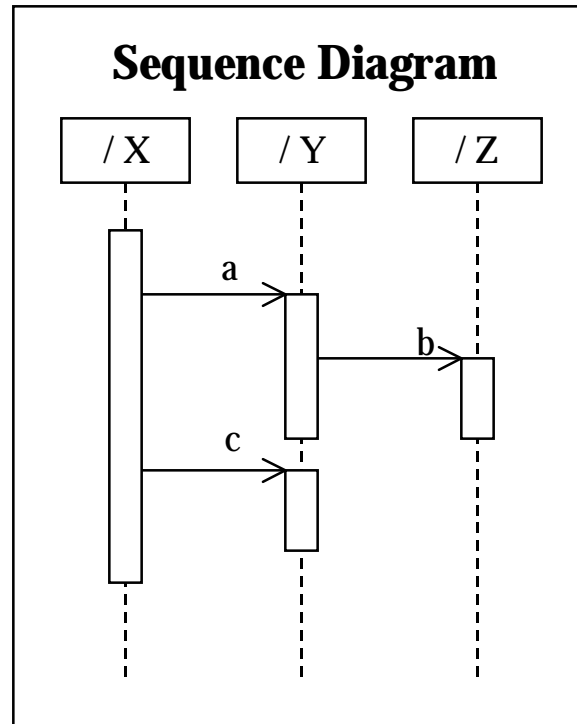
UML 1.4: The diagram shows the contents of a CollaborationInstance

# Collaborations including Interactions



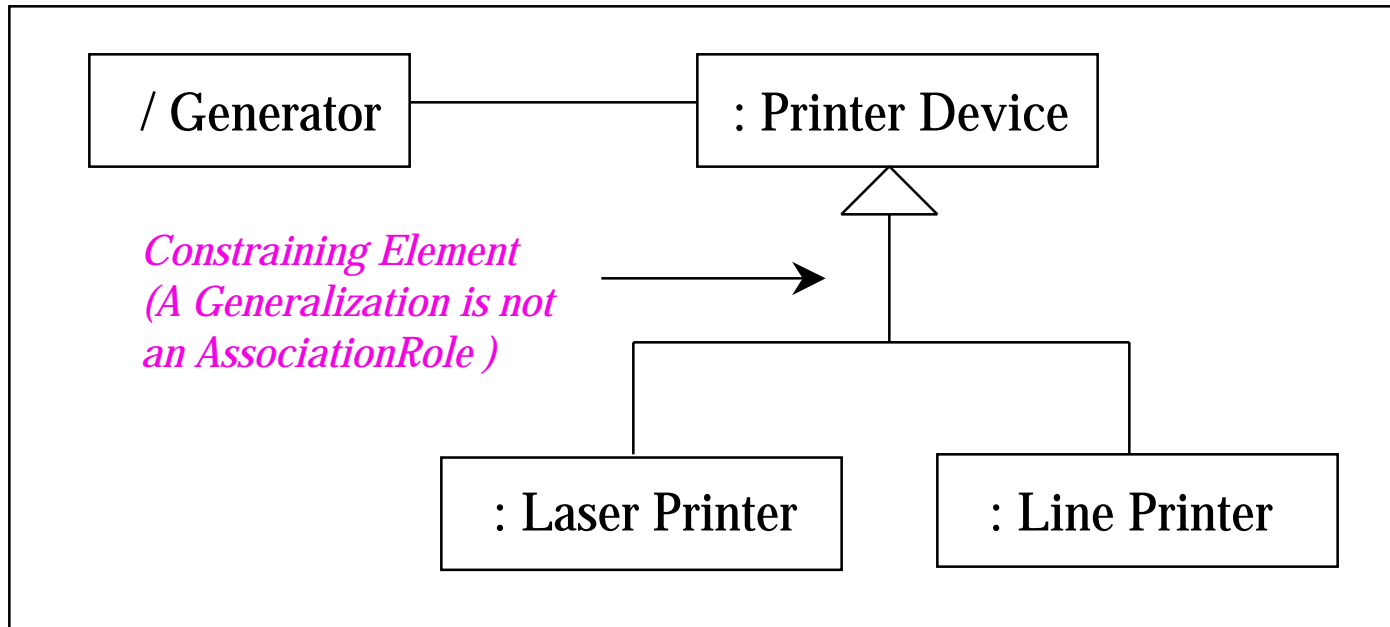
UML 1.4: The diagrams show the contents of a CollaborationInstance with an InteractionInstance superposed

# Roles on Sequence Diagrams

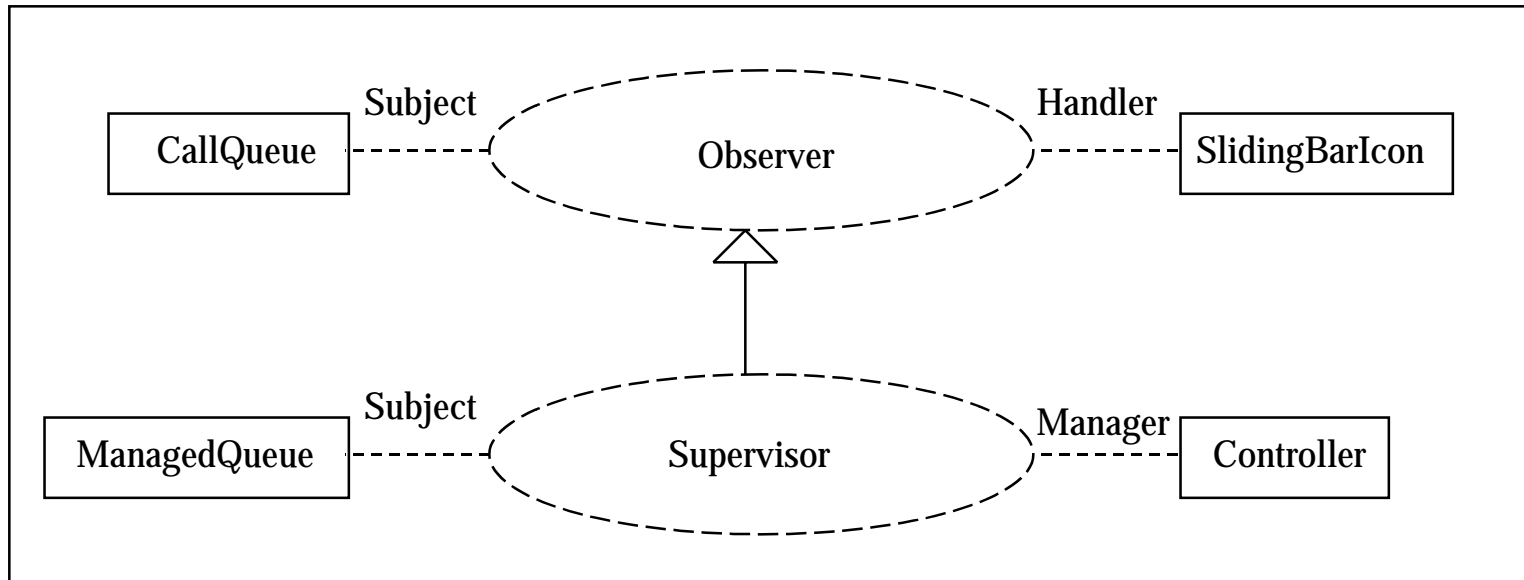


UML 1.4: The diagram shows the contents of a Collaboration with an Interaction superposed

# Collaboration Diagram with Constraining Elements



# Static Diagram With Collaboration and Classifiers





# When to Model Collaborations

---

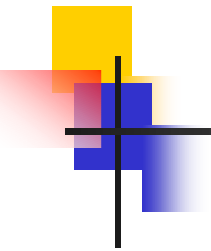
- Use Collaborations as a tool to find the Classifiers.
- Trace a Use Case / Operation onto Classifiers.
- Map the specification of a Subsystem onto its realization (Tutorial 3).



# 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

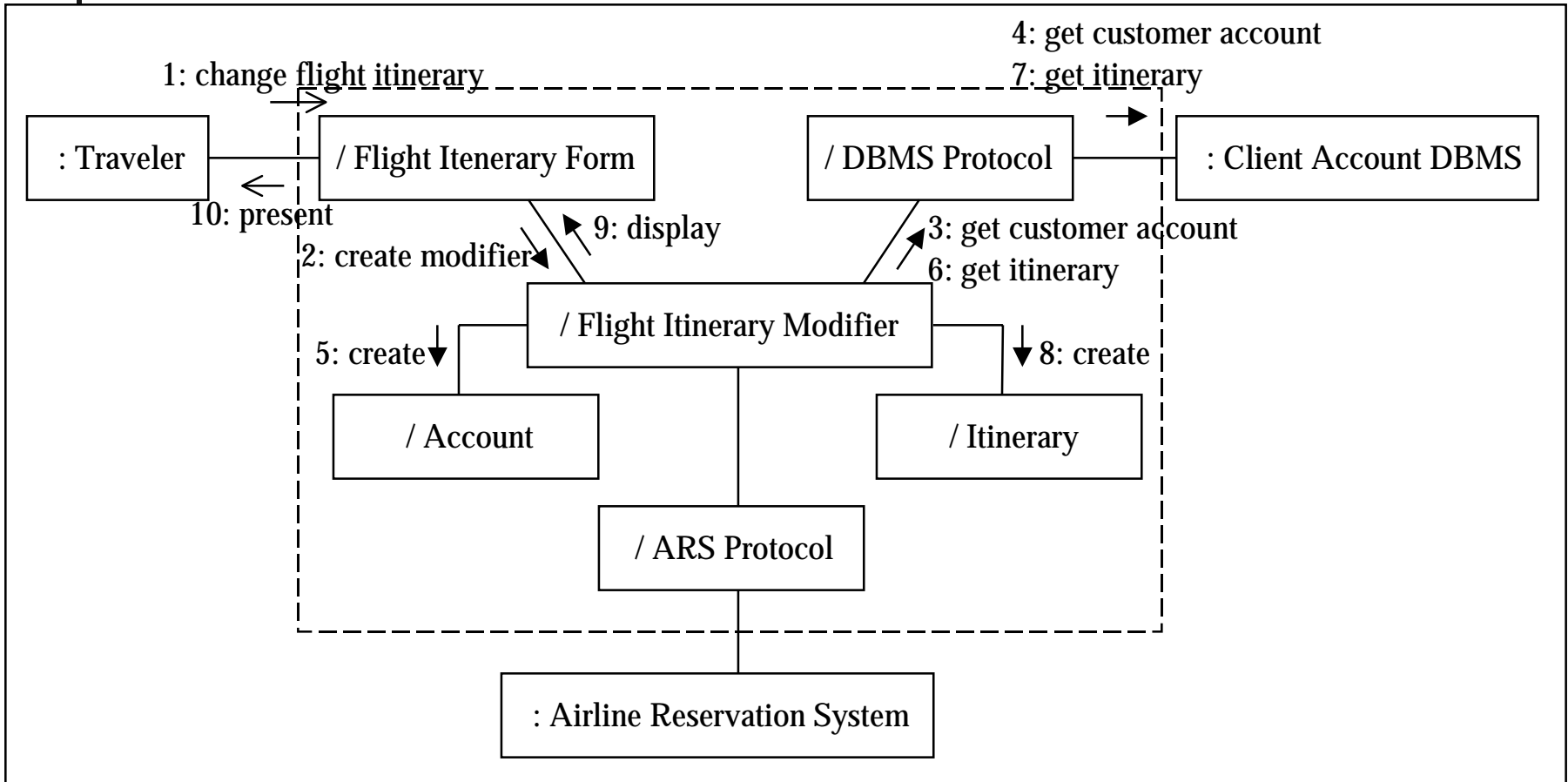


# Use Case Description: Change Flt Itinerary

---

- **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





# 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



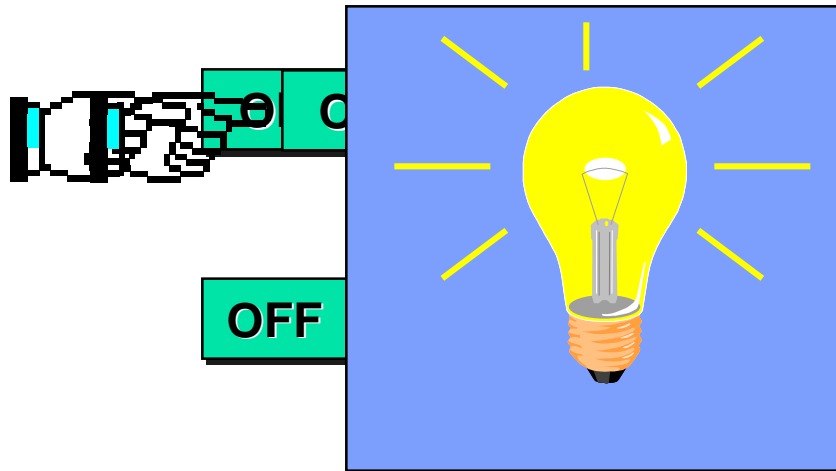
# Overview

---

- Basic State Machine Concepts
- Statecharts and Objects
- Advanced Modeling Concepts
- Case Study
- Wrap Up

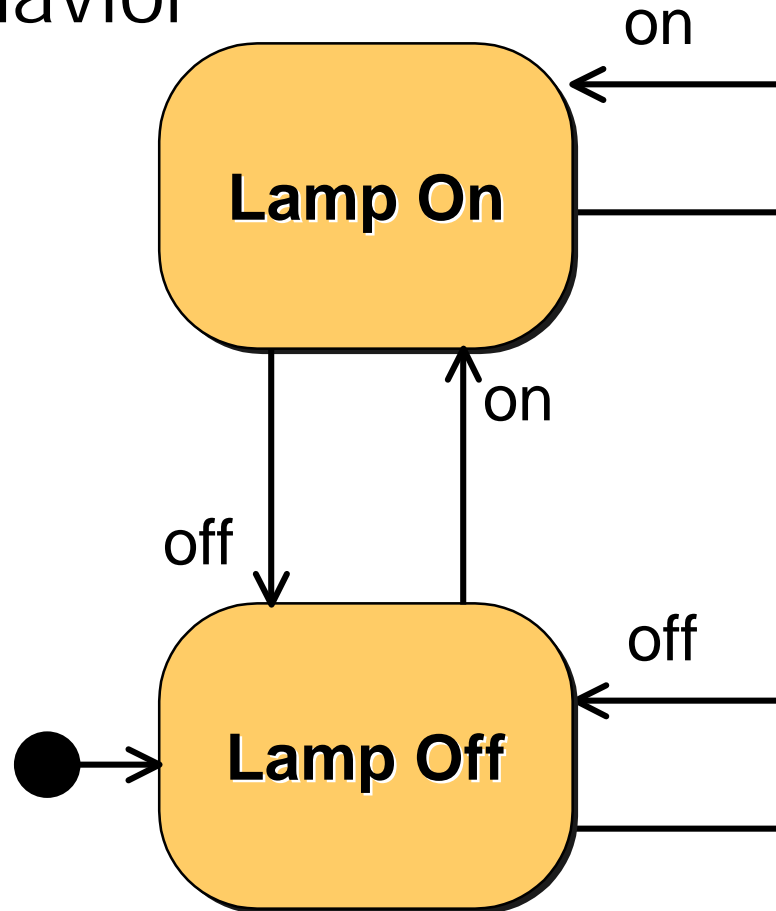
# Automata

- 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



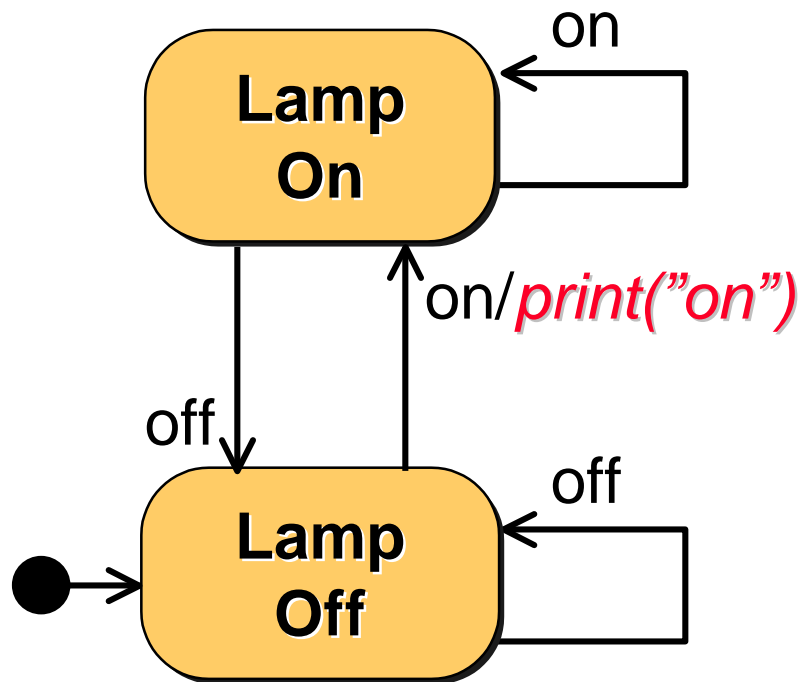
# State Machine (Automaton) Diagram

- Graphical rendering of automata behavior

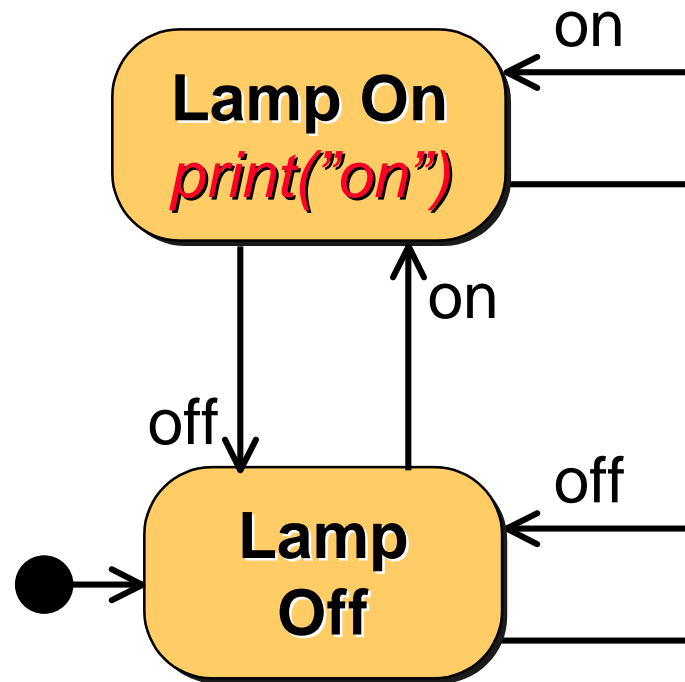


# Outputs and Actions

- As the automaton changes state it can generate outputs:



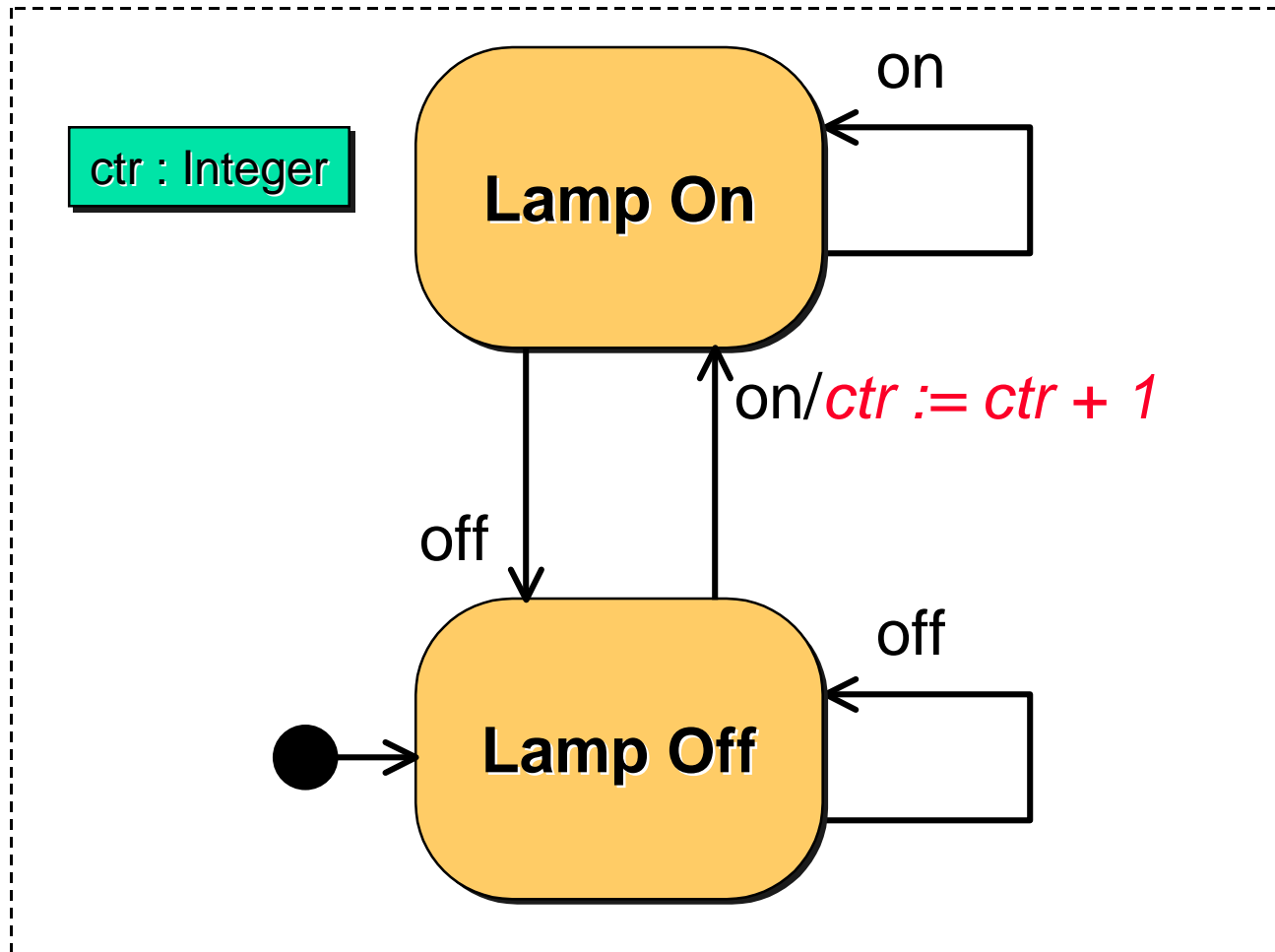
**Mealy** automaton



**Moore** automaton

# Extended State Machines

- Addition of variables ("extended state")



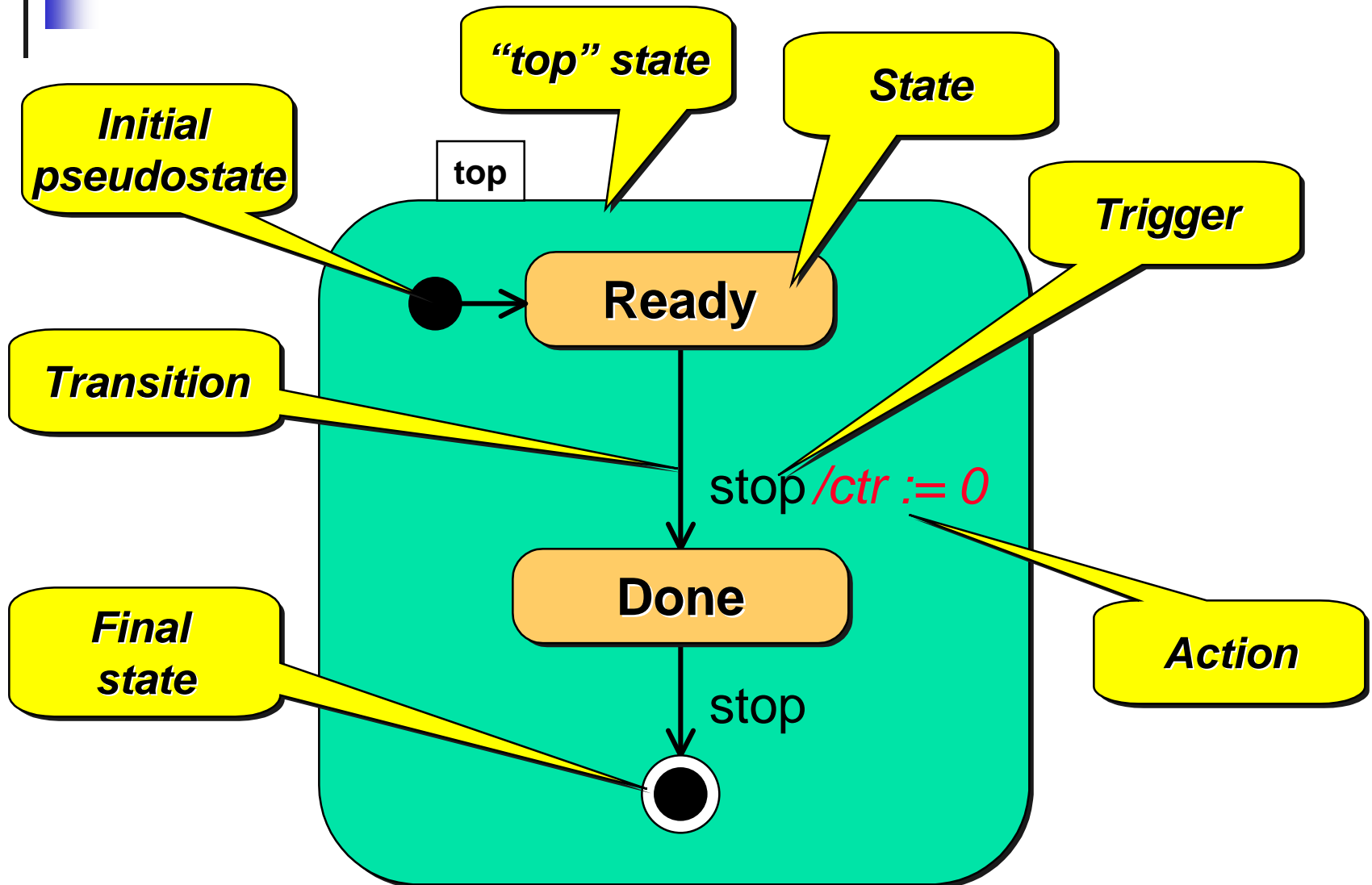


# A Bit of Theory

---

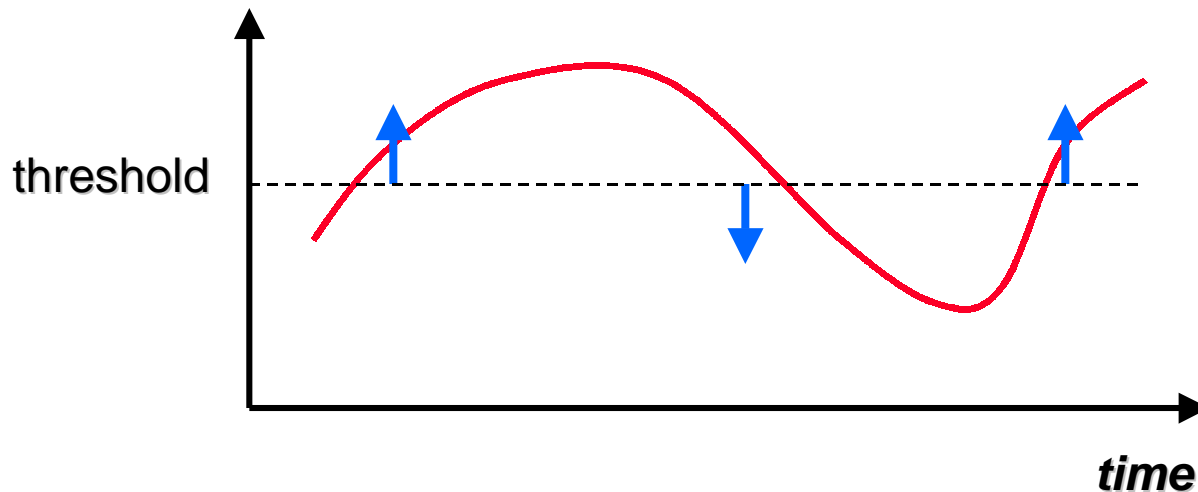
- An extended (Mealy) state machine is defined by:
  - a set of input signals (input alphabet)
  - a set of output signals (output alphabet)
  - a set of states
  - a set of transitions
    - triggering signal
    - action
  - a set of extended state variables
  - an initial state designation
  - a set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

# Basic UML Statechart Diagram



# What Kind of Behavior?

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





# Event-Driven Behavior

---

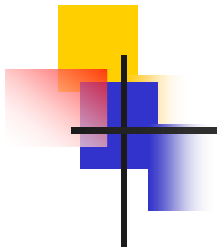
- Event = a type of observable occurrence
  - interactions:
    - synchronous object operation invocation (call event)
    - asynchronous signal reception (signal event)
  - occurrence of time instants (time event)
    - interval expiry
    - calendar/clock time
  - change in value of some entity (change 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

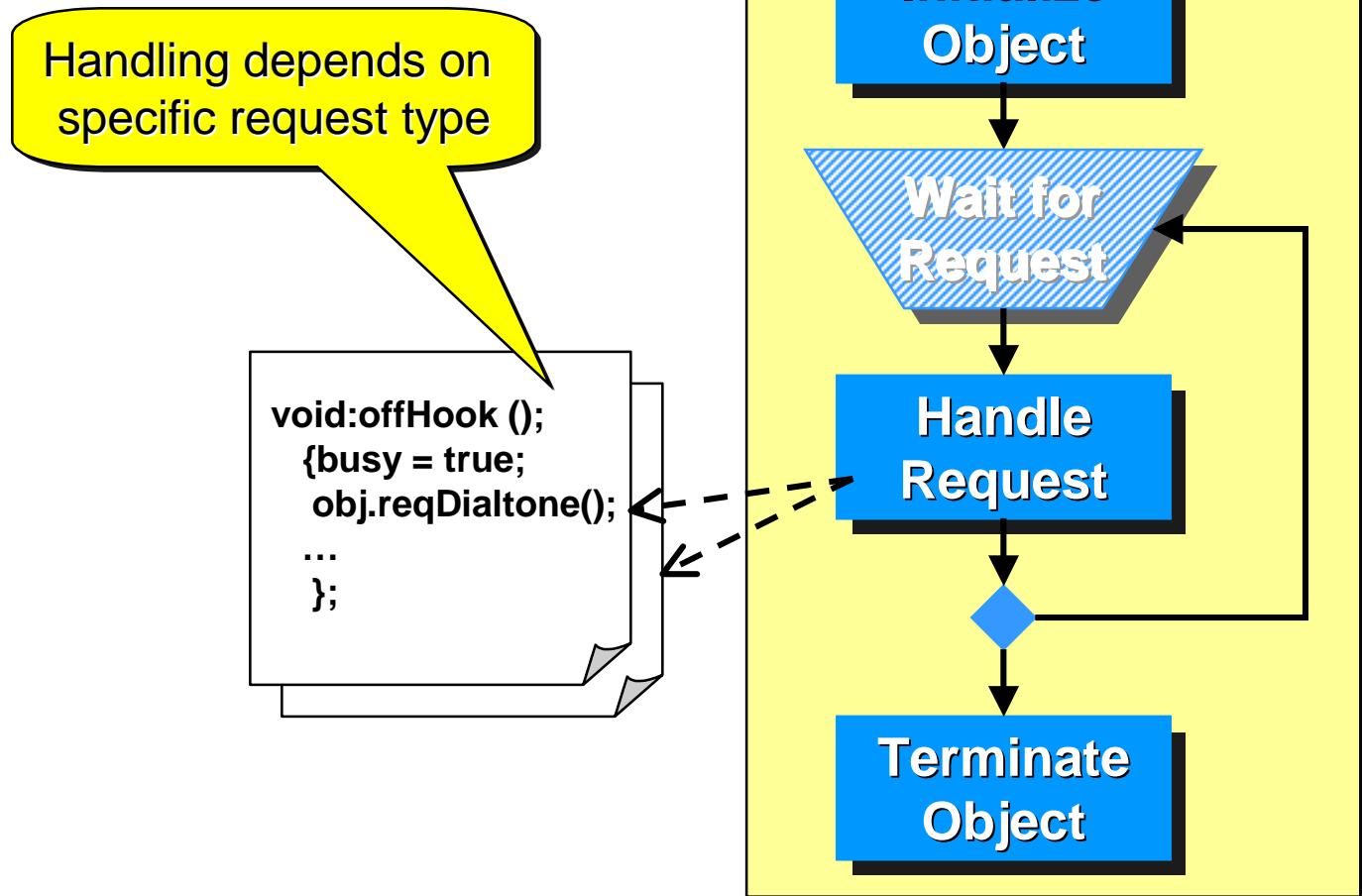
Advanced Modeling Concepts

Case Study

Wrap Up

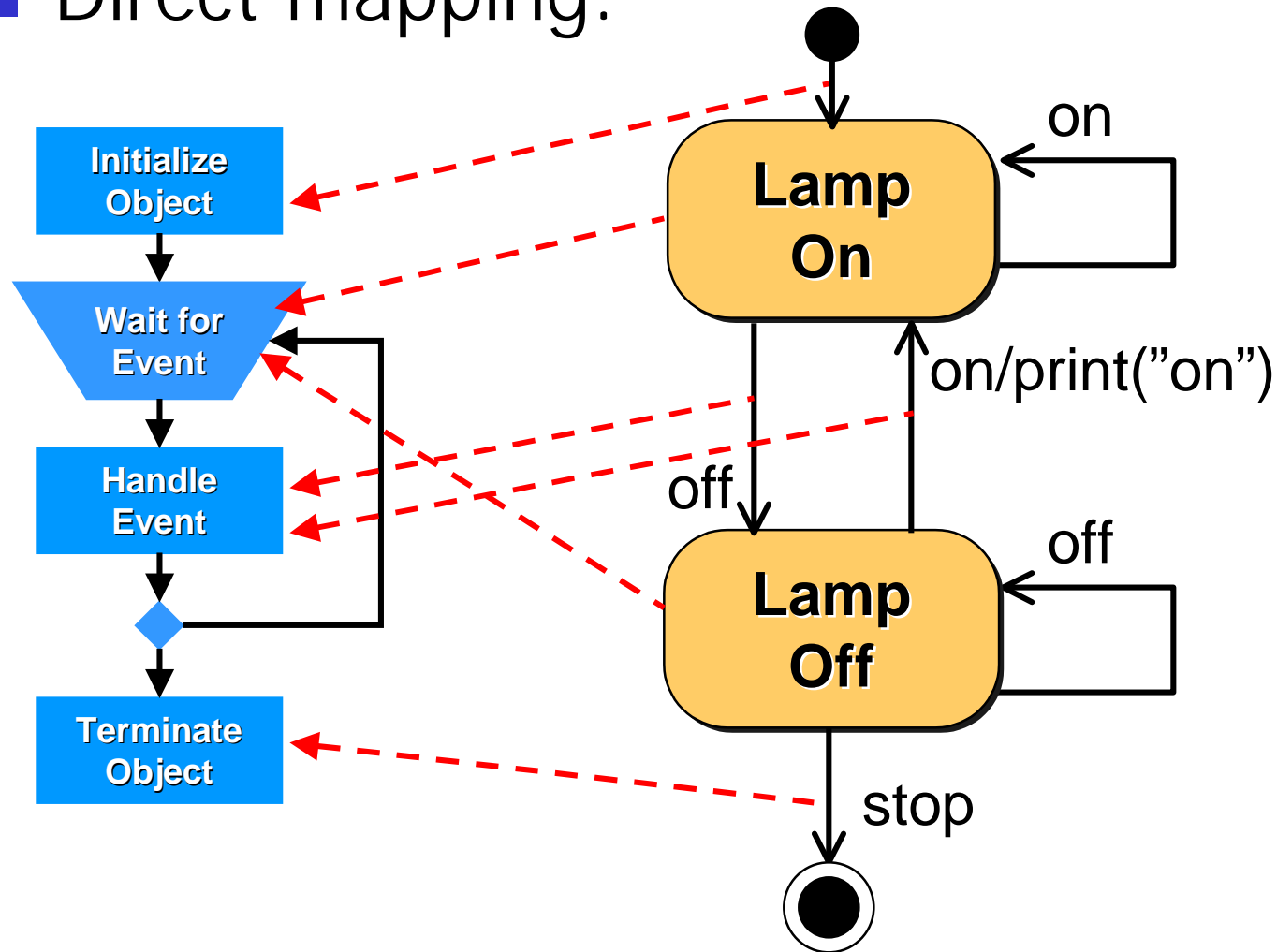
# Object Behavior - General Model

- Simple server model:



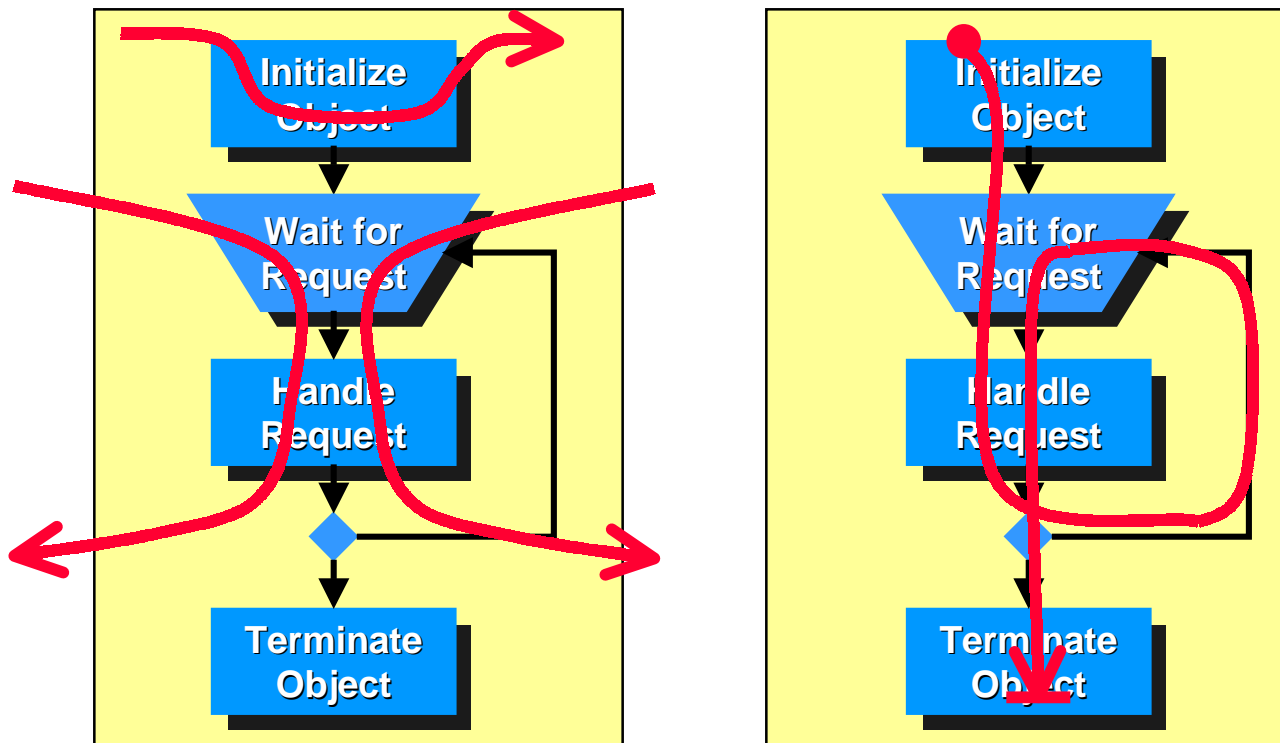
# Object Behavior and State Machines

- Direct mapping:

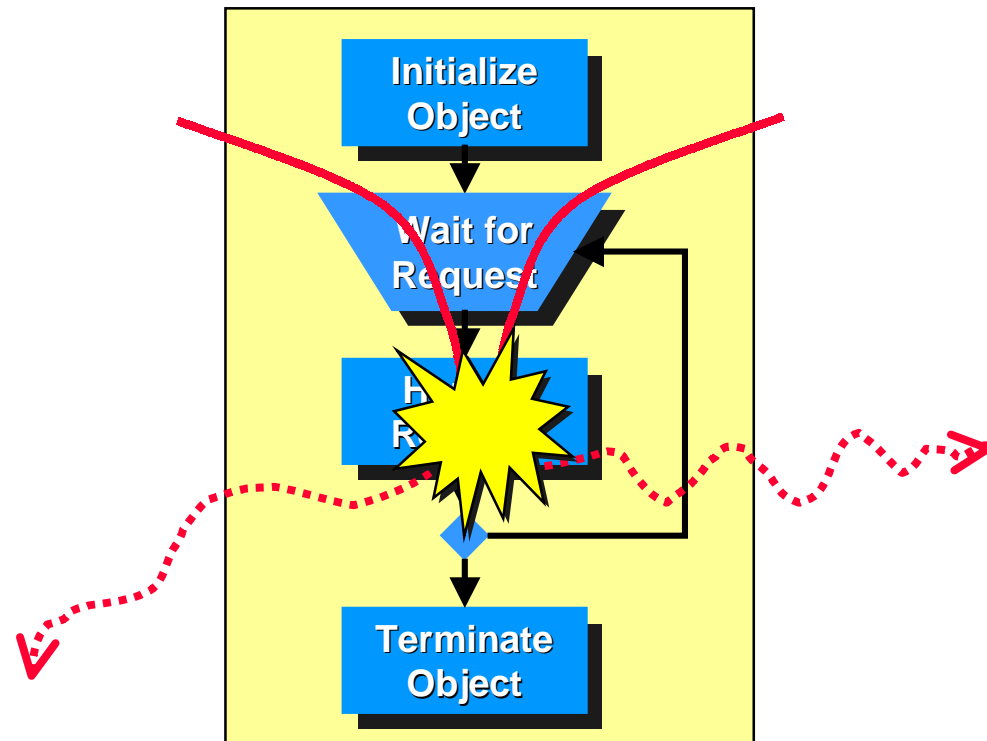


# Object and Threads

- **Passive objects:** depend on external power (thread of execution)
- **Active objects:** self-powered (own thread of execution)



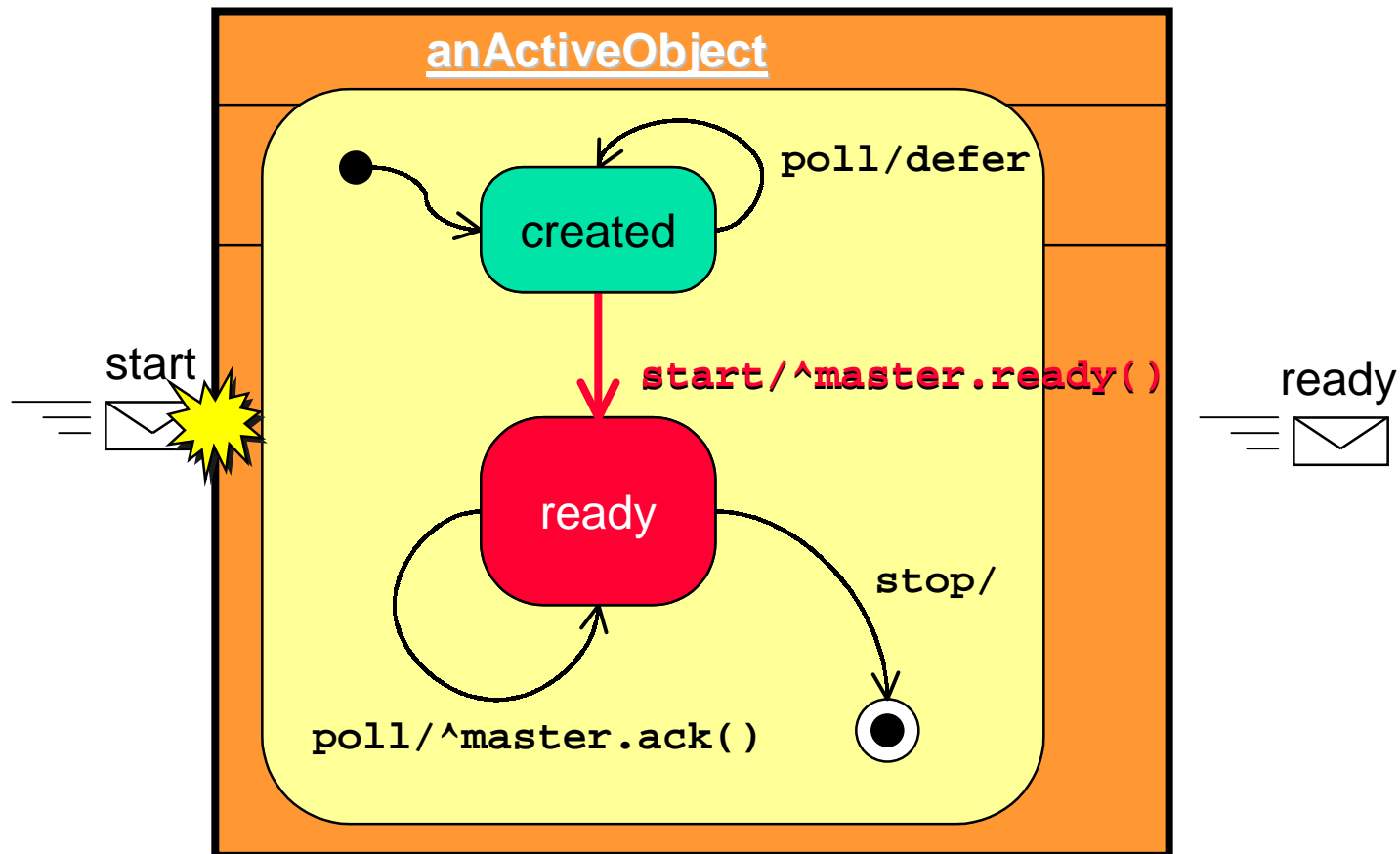
# Passive Objects: Dynamic Semantics



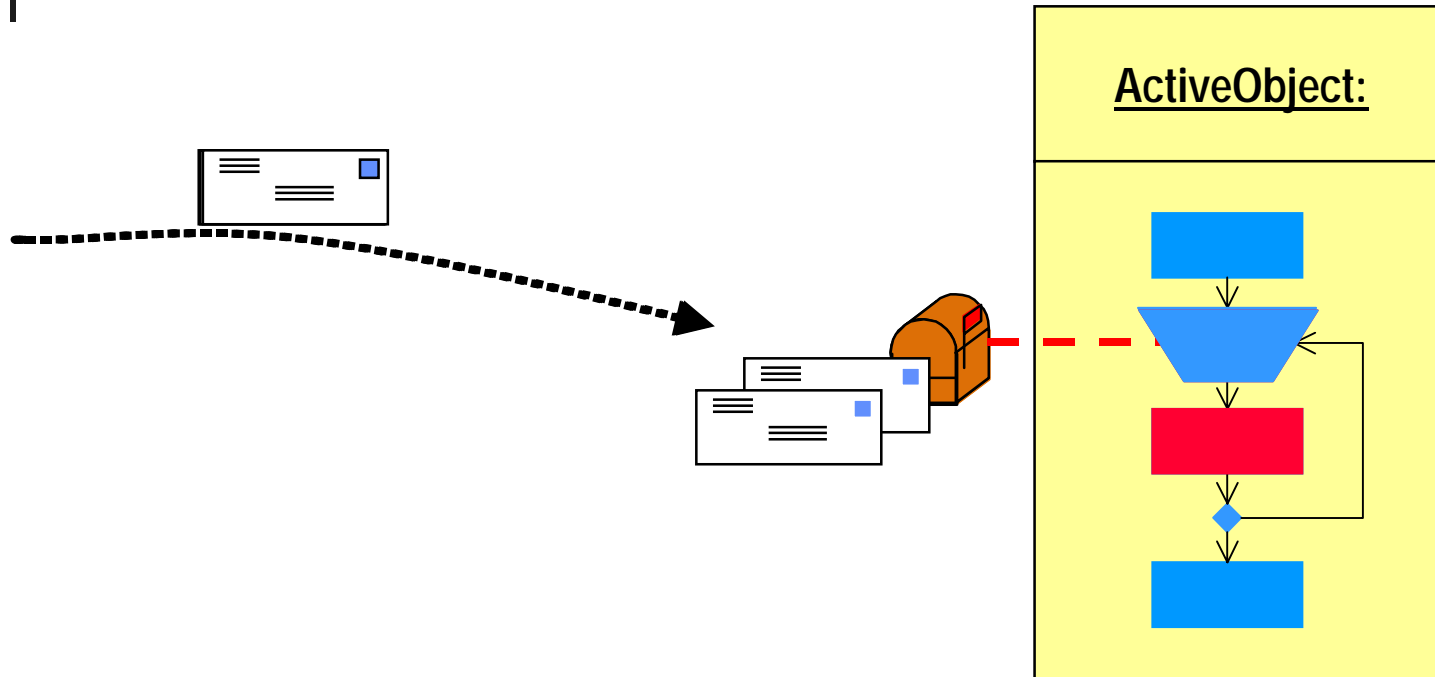
- Encapsulation does not protect the object from concurrency conflicts!
- Explicit synchronization is still required

# Active Objects and State Machines

- Objects that encapsulate own thread of execution



# Active Objects: Dynamic Semantics

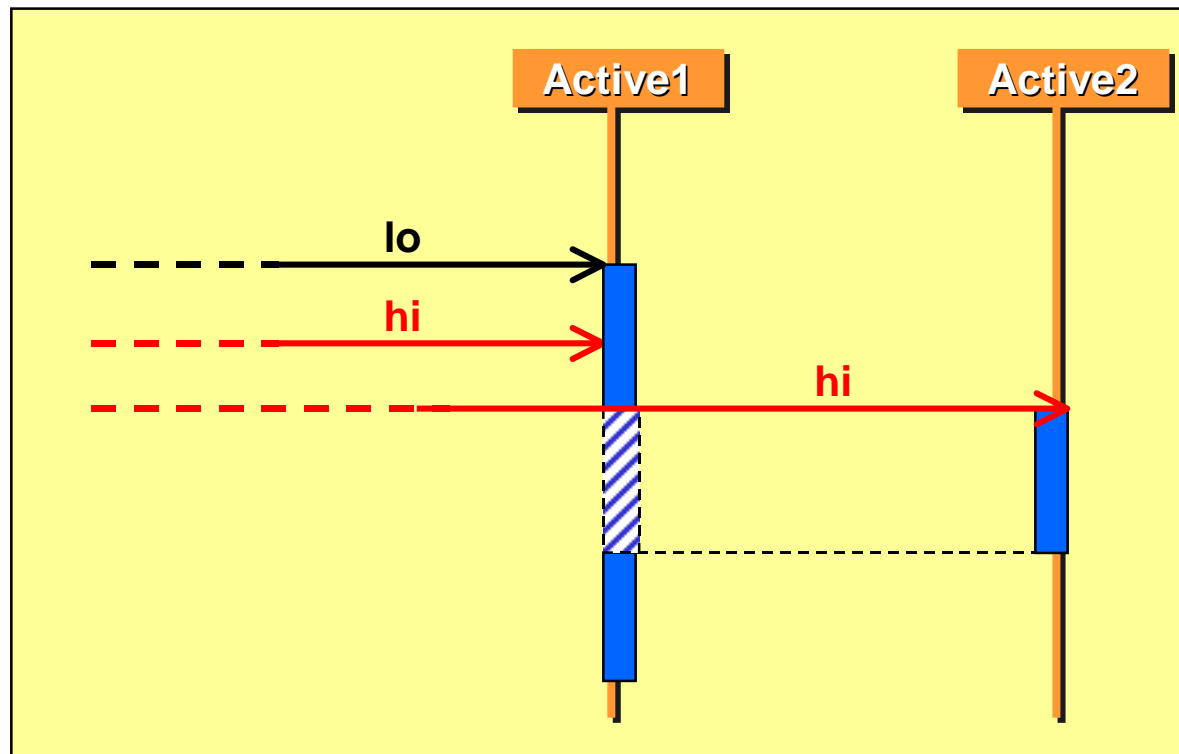


## Run-to-completion model:

- 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

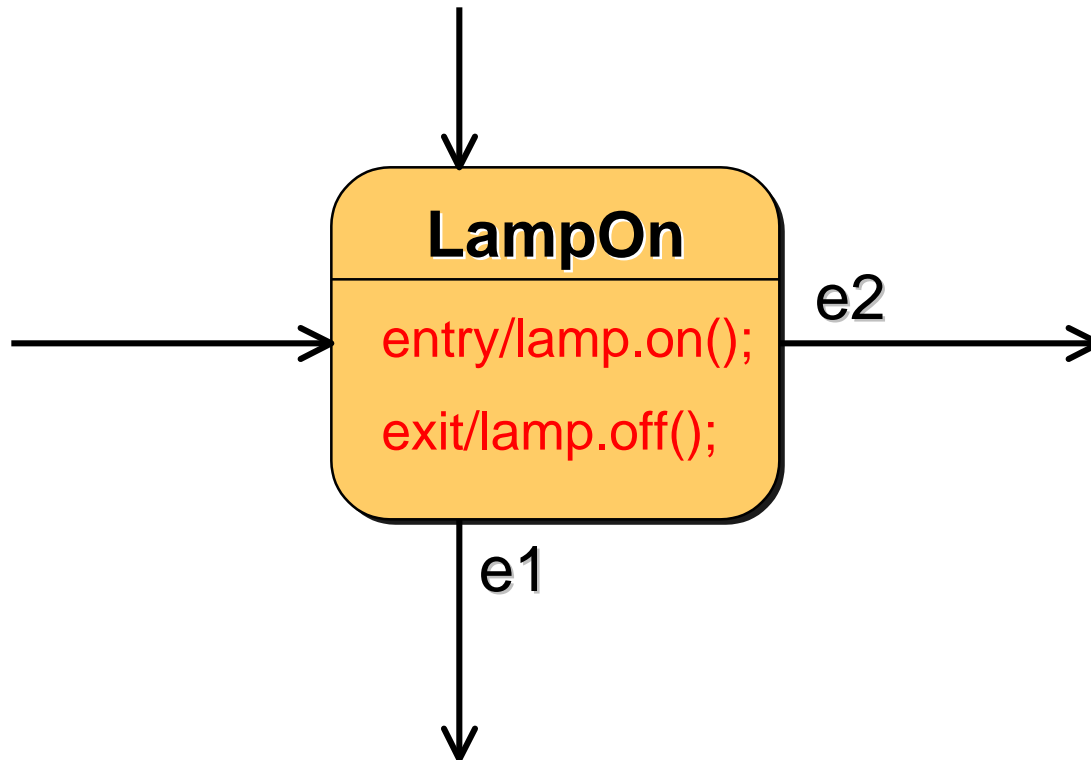
**Advanced Modeling Concepts**

Case Study

Wrap Up

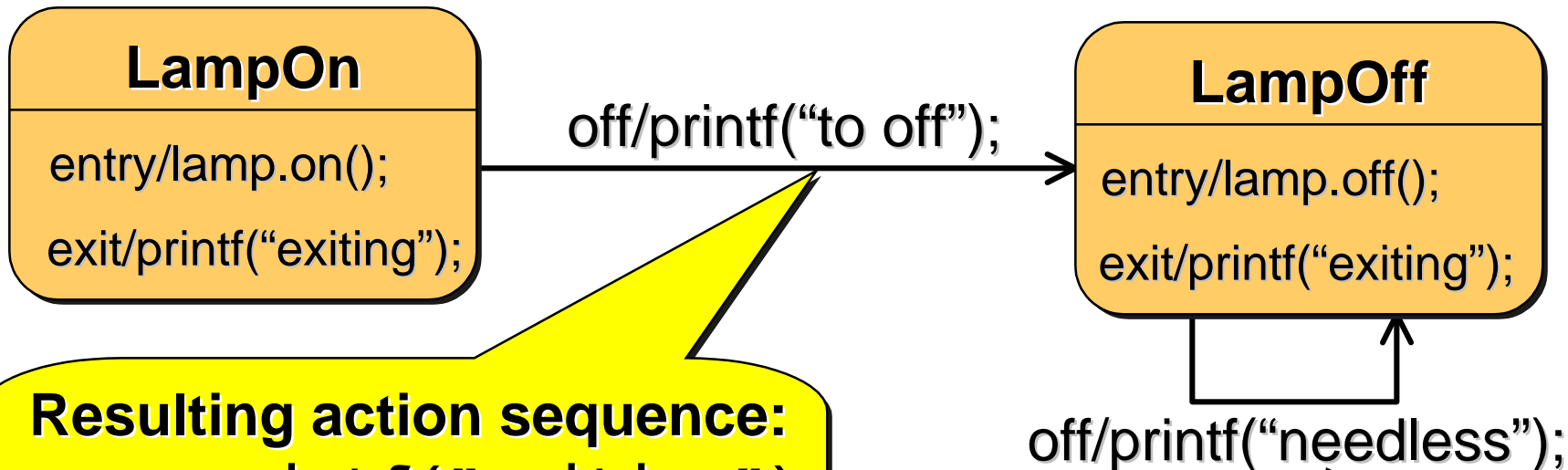
# State Entry and Exit Actions

- A dynamic assertion mechanism



# Order of Actions: Simple Case

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

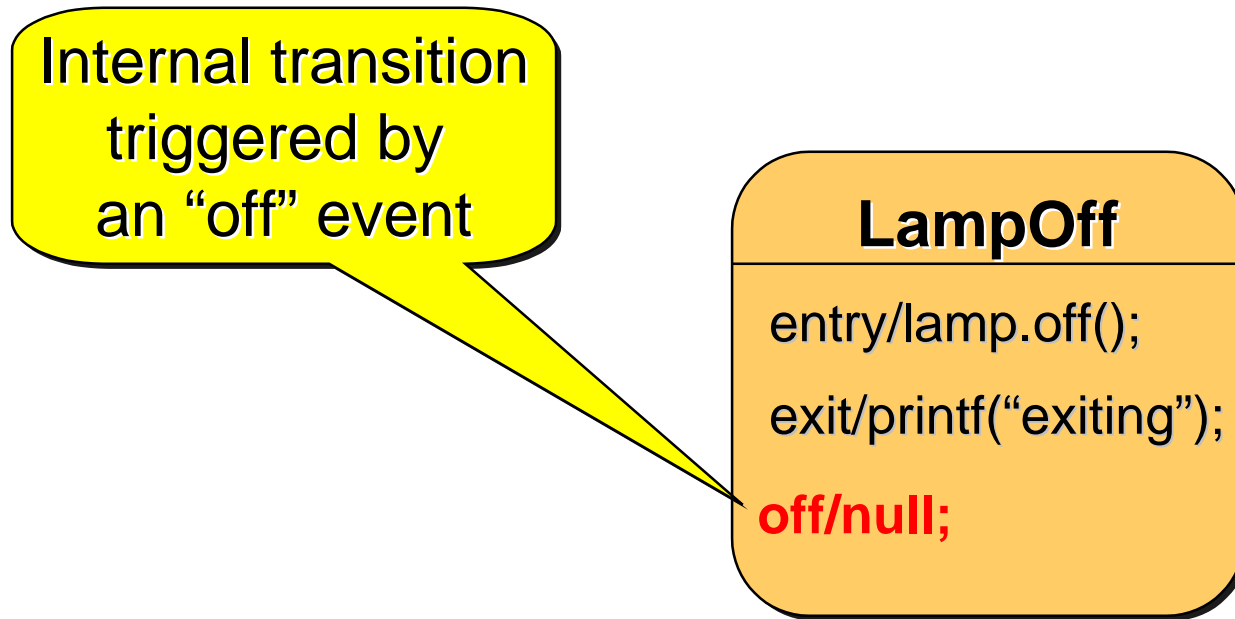


**Resulting action sequence:**  
`printf("exiting");`  
`printf("to off");`  
`lamp.off();`

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

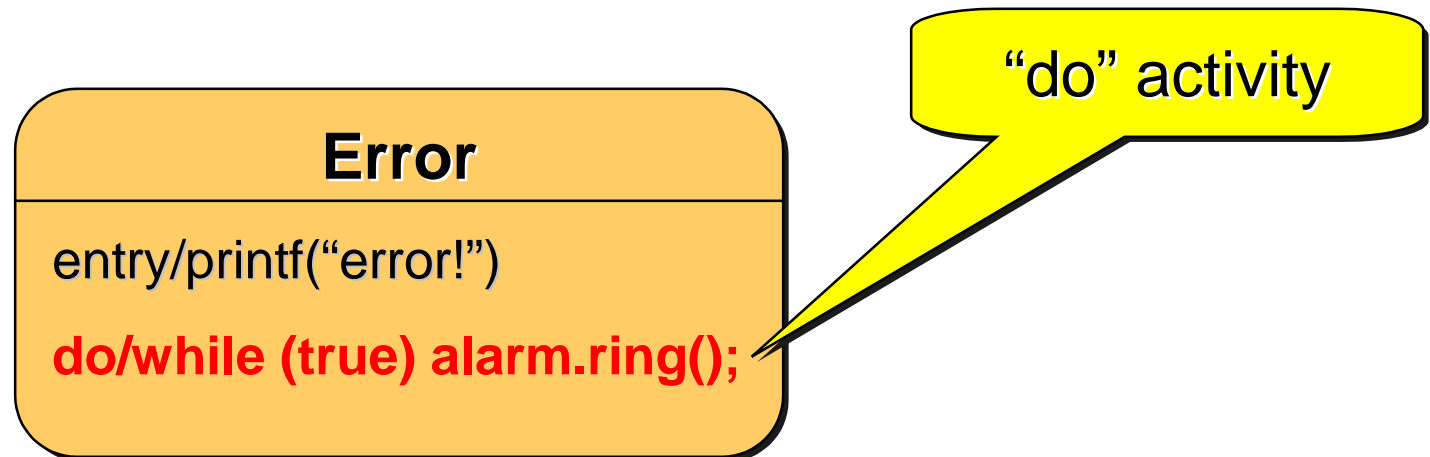
# Internal Transitions

- Self-transitions that bypass entry and exit actions



# State ("Do") Activities

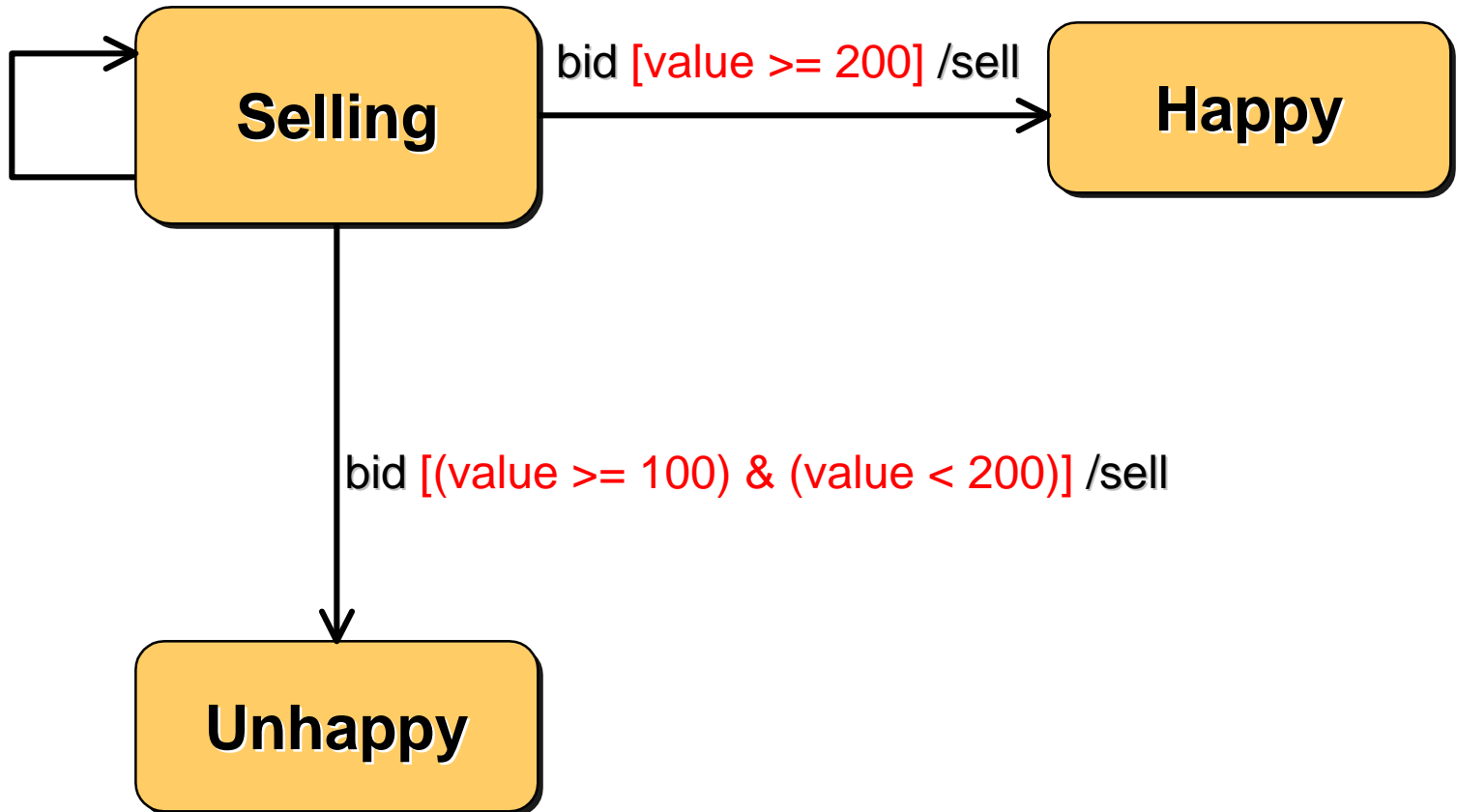
- Forks a concurrent thread that executes until:
  - the action completes or
  - the state is exited through an outgoing transition



# Guards

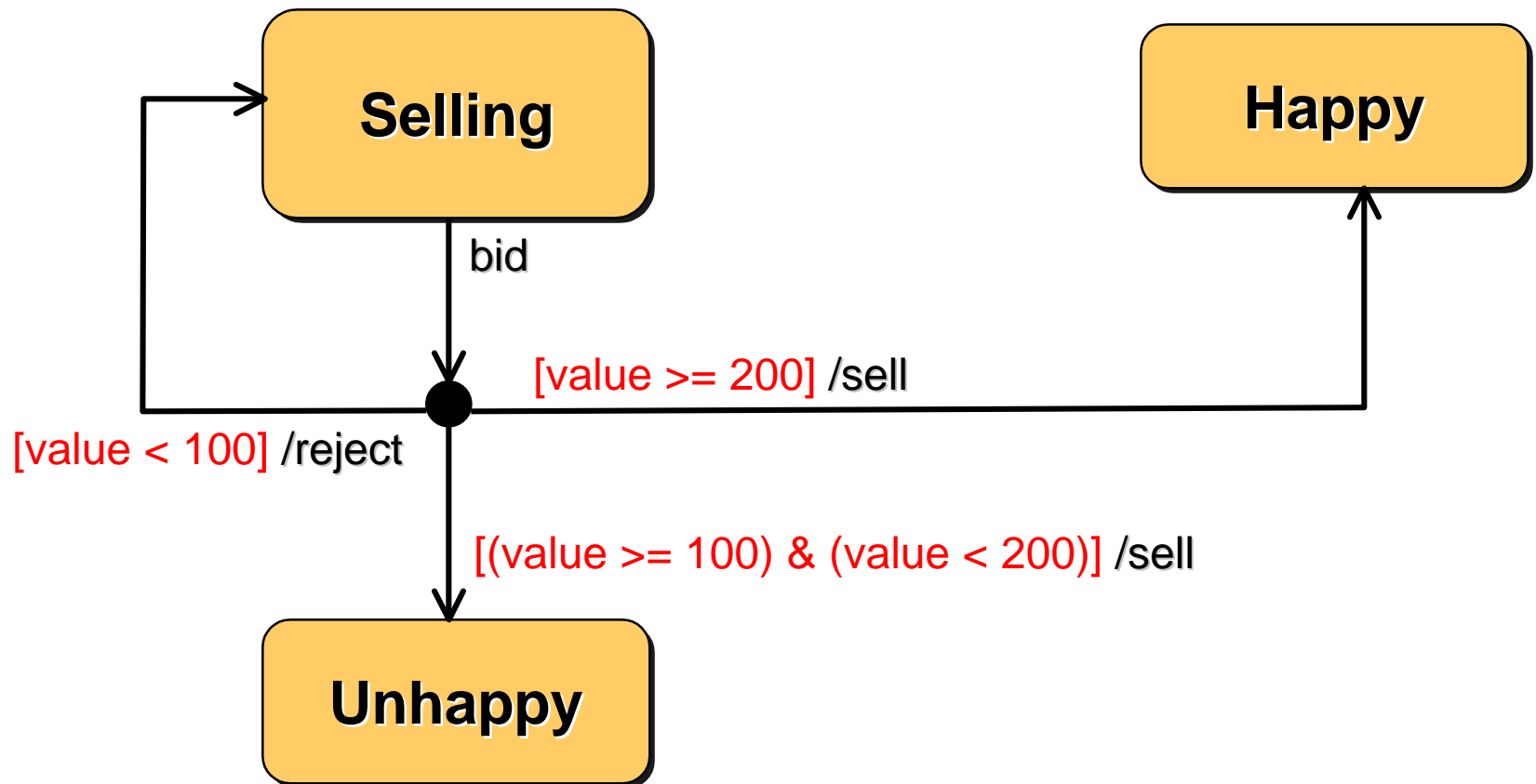
- Conditional execution of transitions
  - guards (Boolean predicates) must be side-effect free

bid [value < 100] /reject



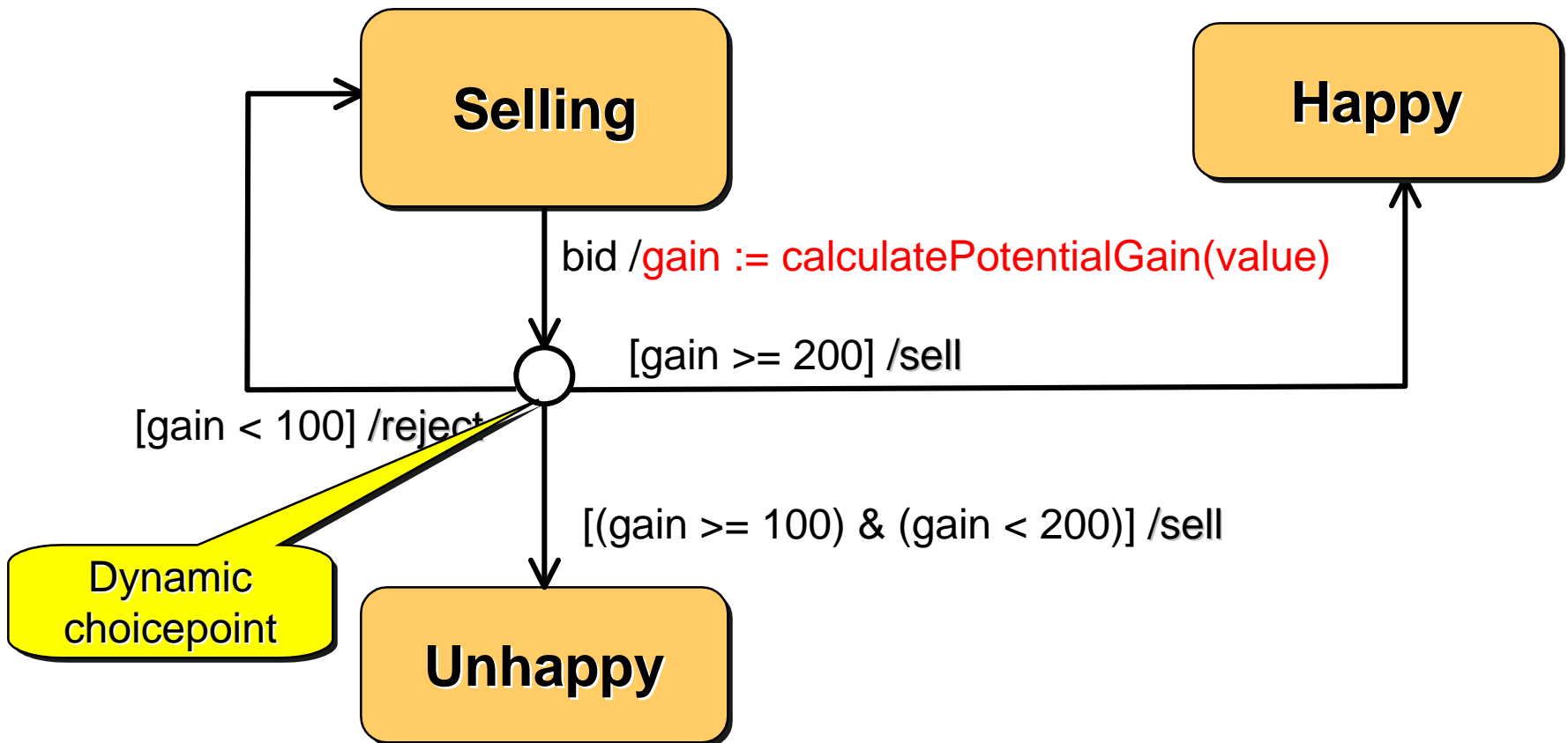
# Static Conditional Branching

- Merely a graphical shortcut for convenient rendering of decision trees



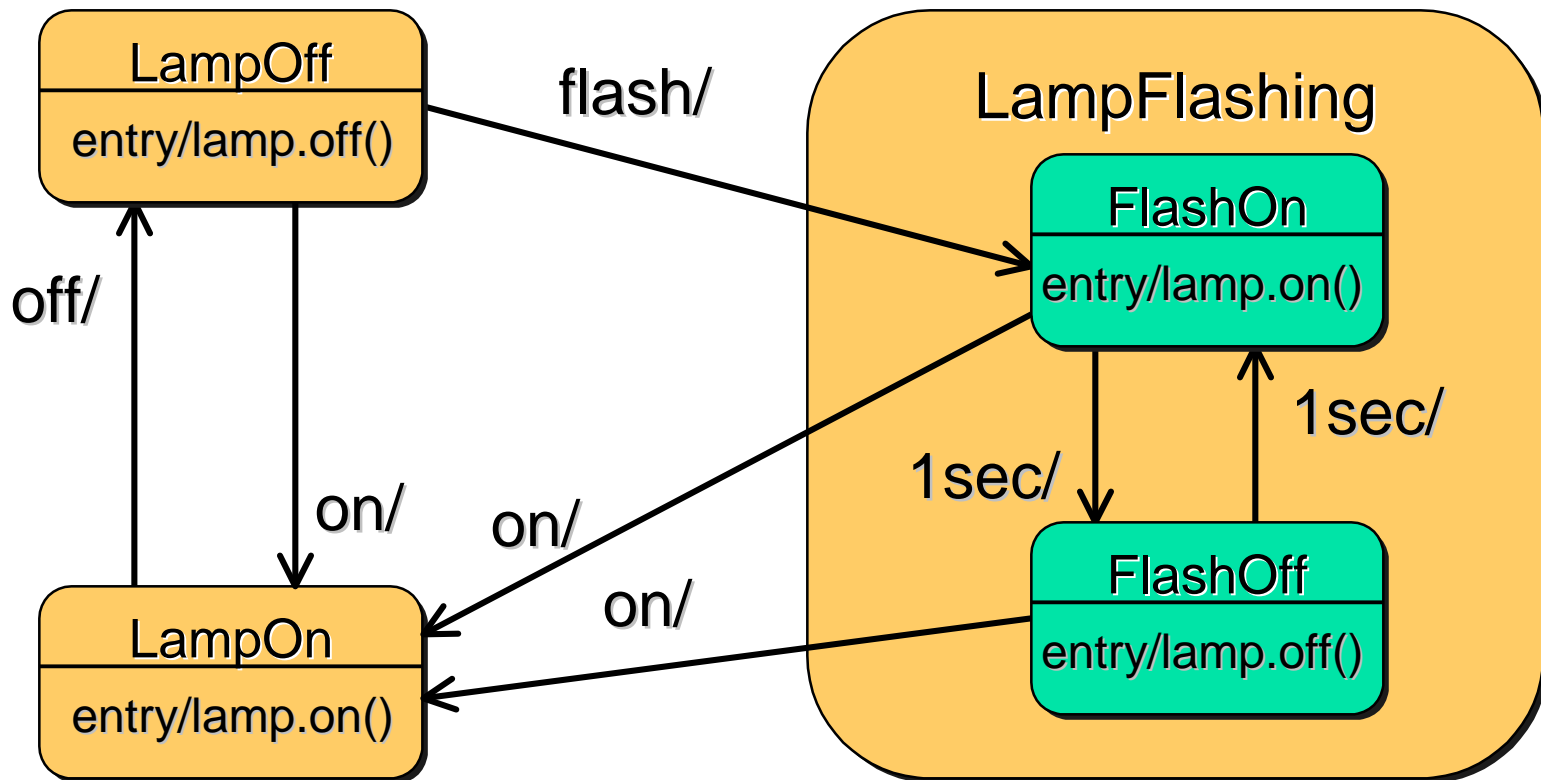
# Dynamic Conditional Branching

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



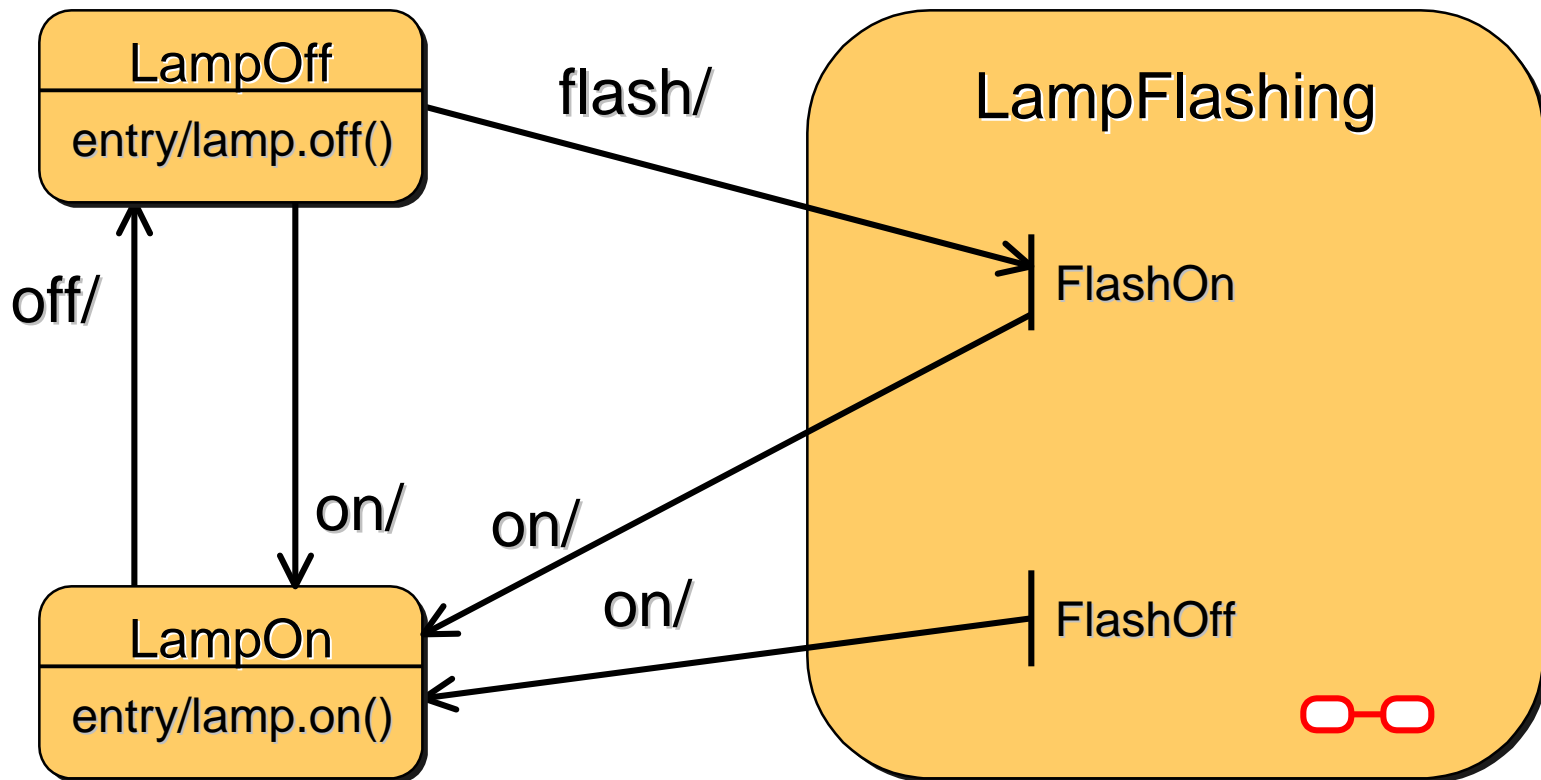
# Hierarchical State Machines

- Graduated attack on complexity
  - states decomposed into state machines



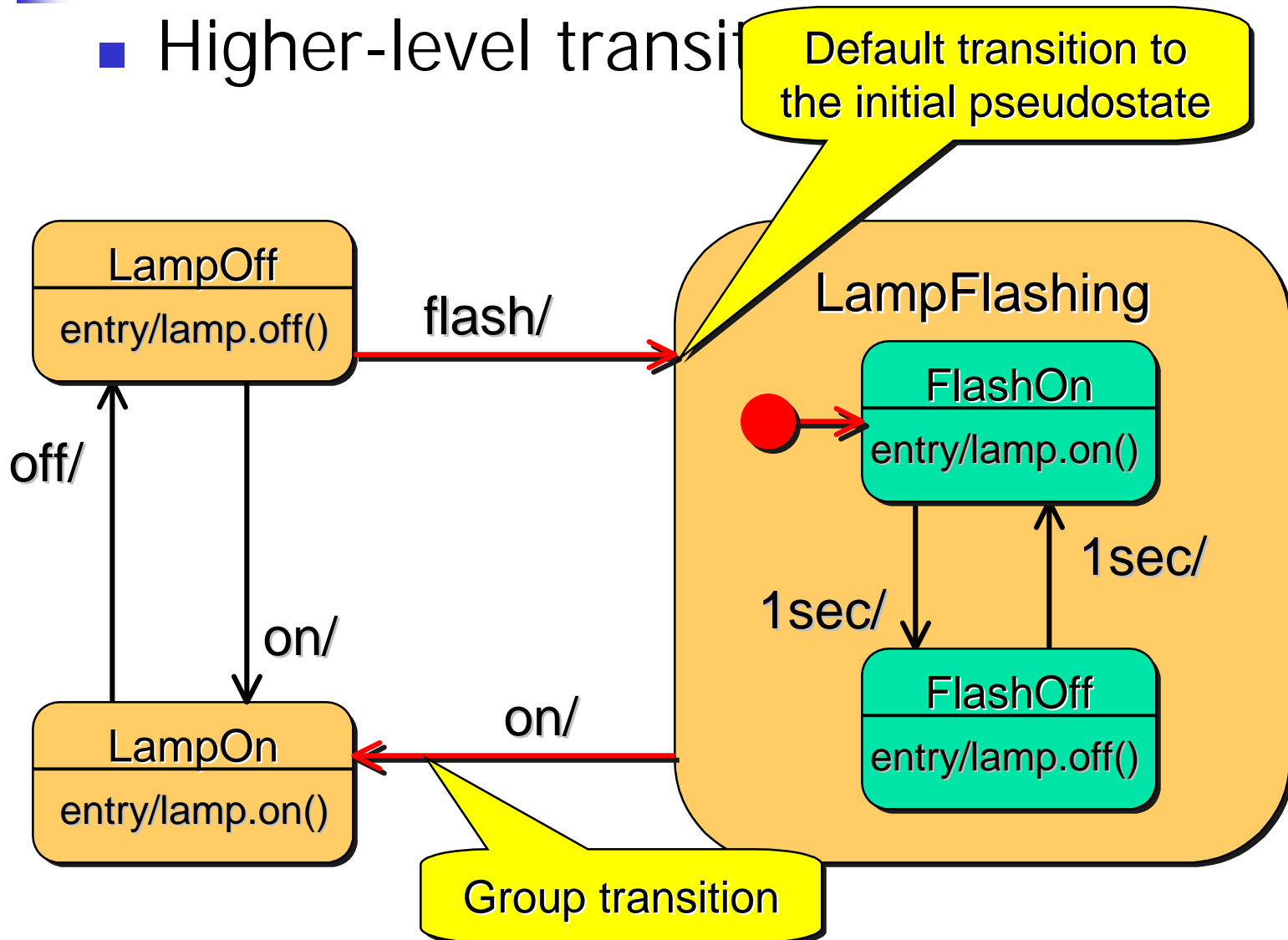
# "Stub" Notation

- Notational shortcut: no semantic significance



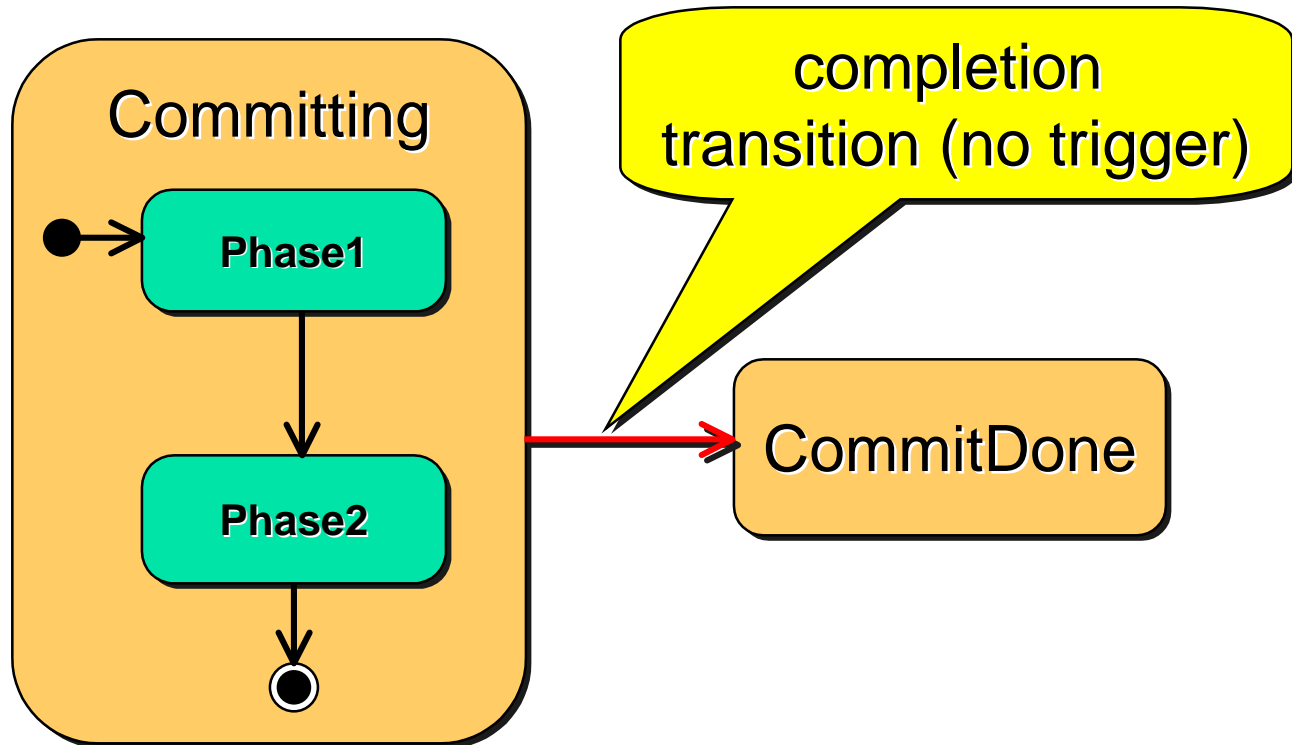
# Group Transitions

- Higher-level transition



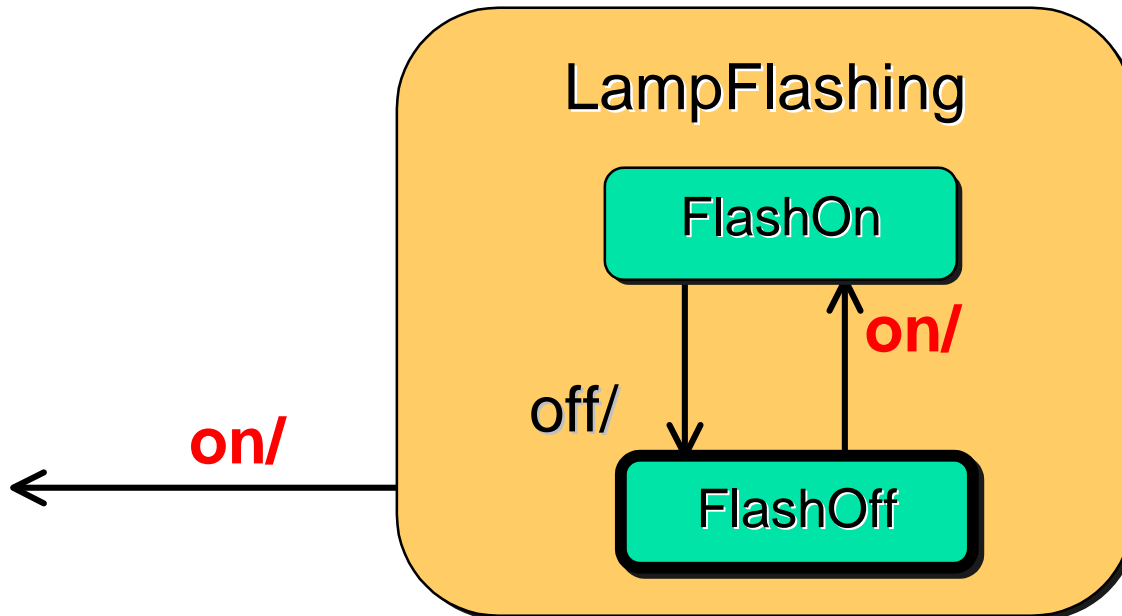
# Completion Transitions

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



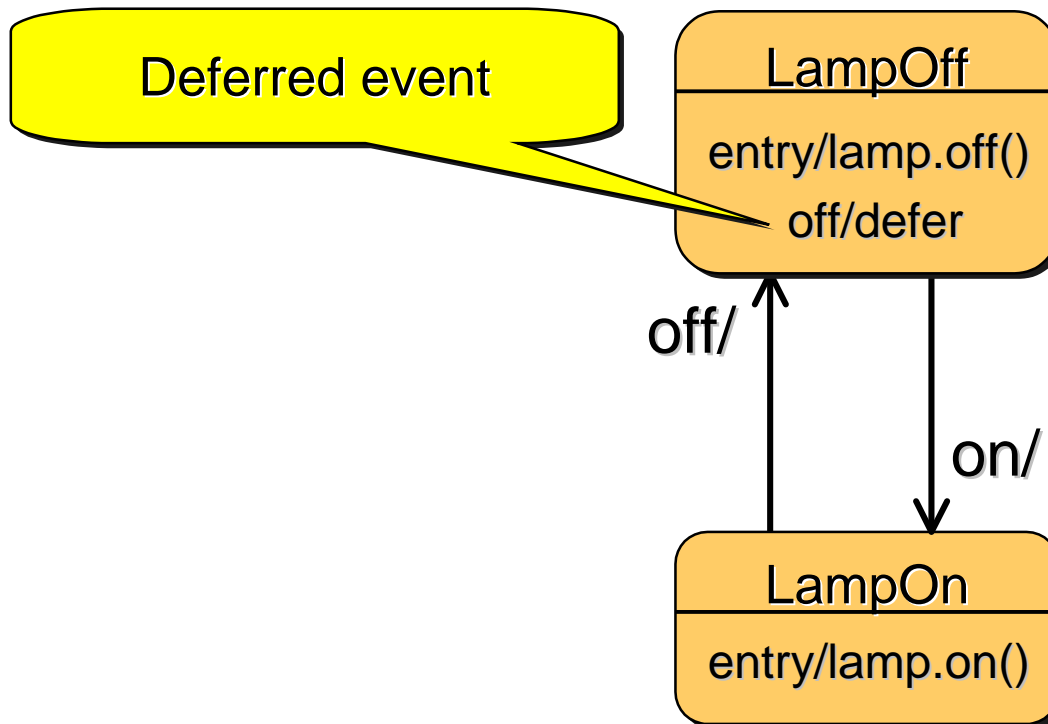
# Triggering Rules

- Two or more transitions may have the same event trigger
  - innermost transition takes precedence
  - event is discarded whether or not it triggers a transition



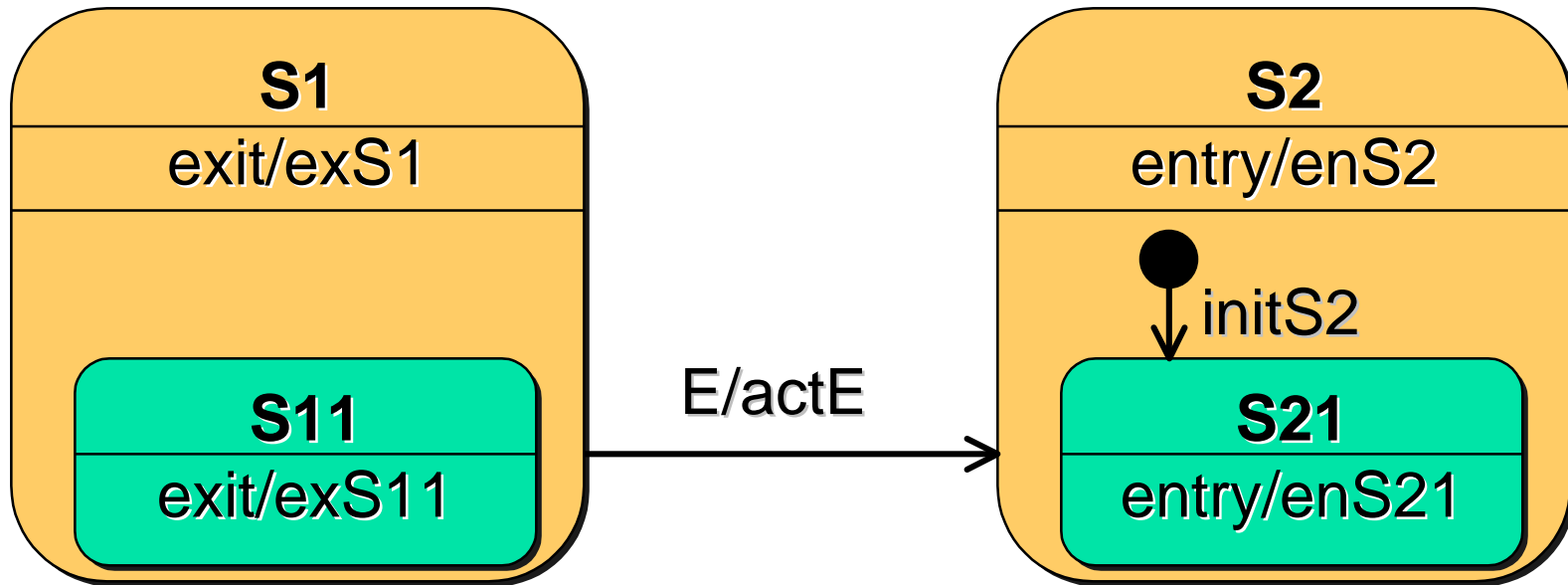
# Deferred Events

- Events can be retained if they do not trigger a transition



# Order of Actions: Complex Case

- Same approach as for the simple case

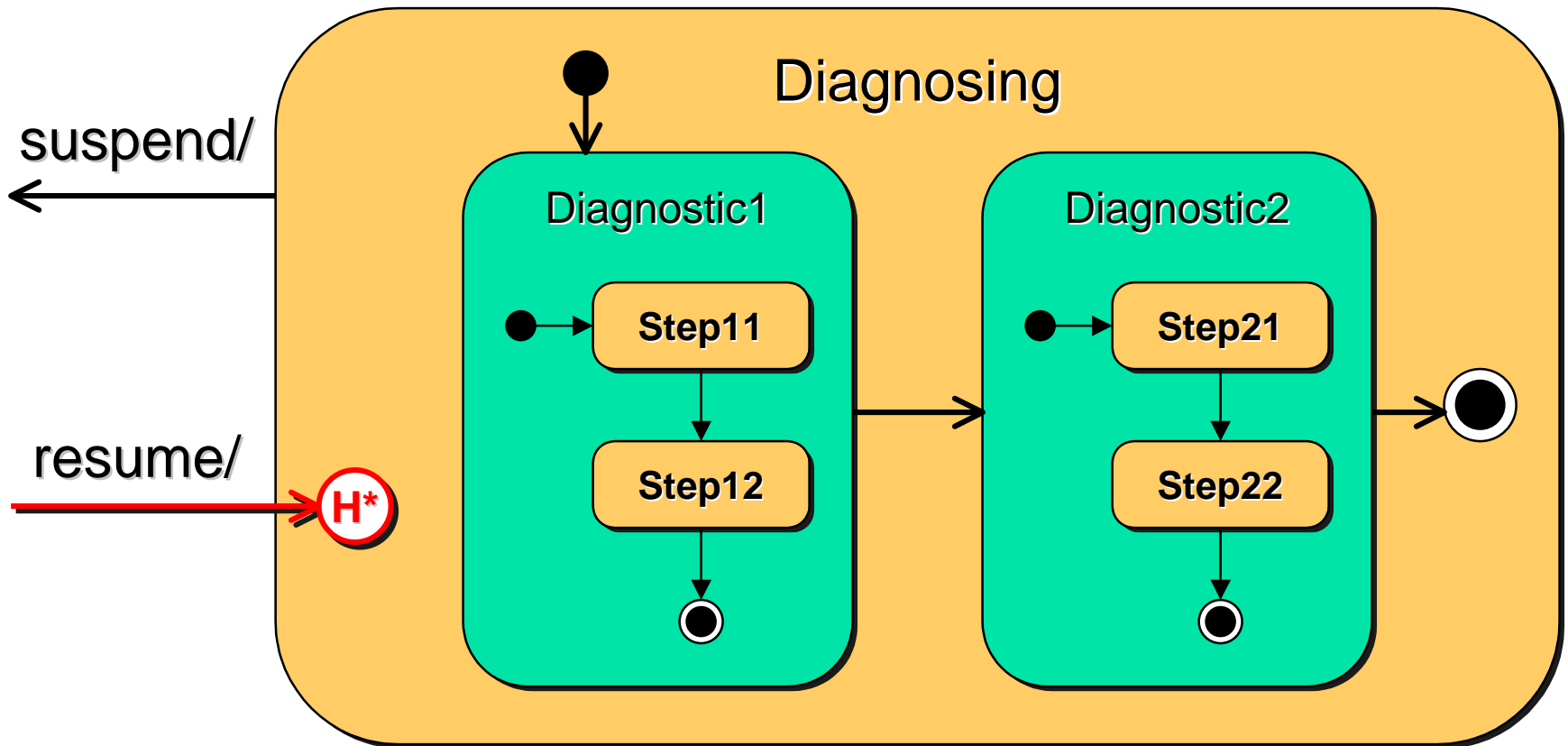


## Actions execution sequence:

`exS11`  $\Rightarrow$  `exS1`  $\Rightarrow$  `actE`  $\Rightarrow$  `enS2`  $\Rightarrow$  `initS2`  $\Rightarrow$  `enS21`

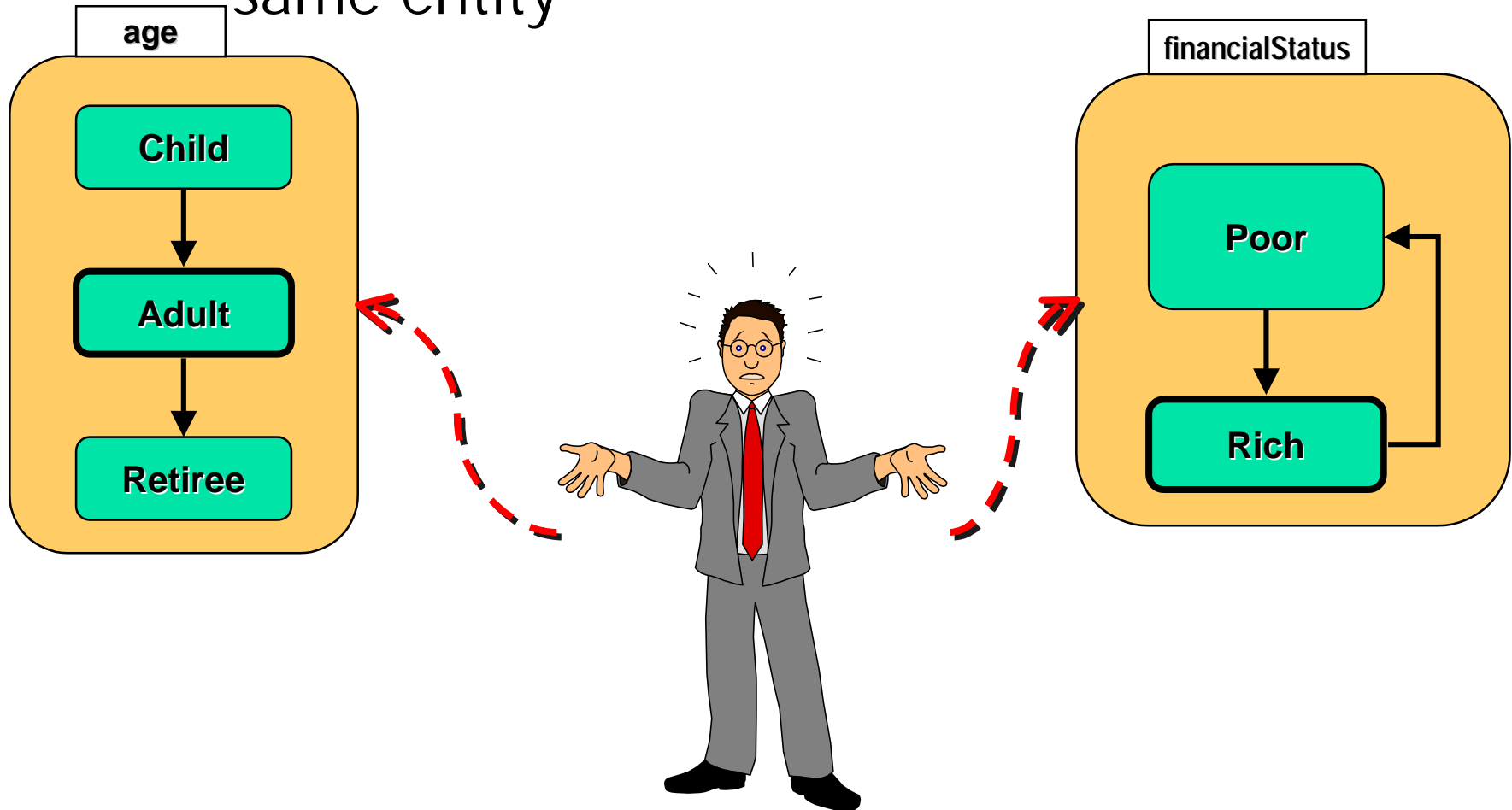
# History

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



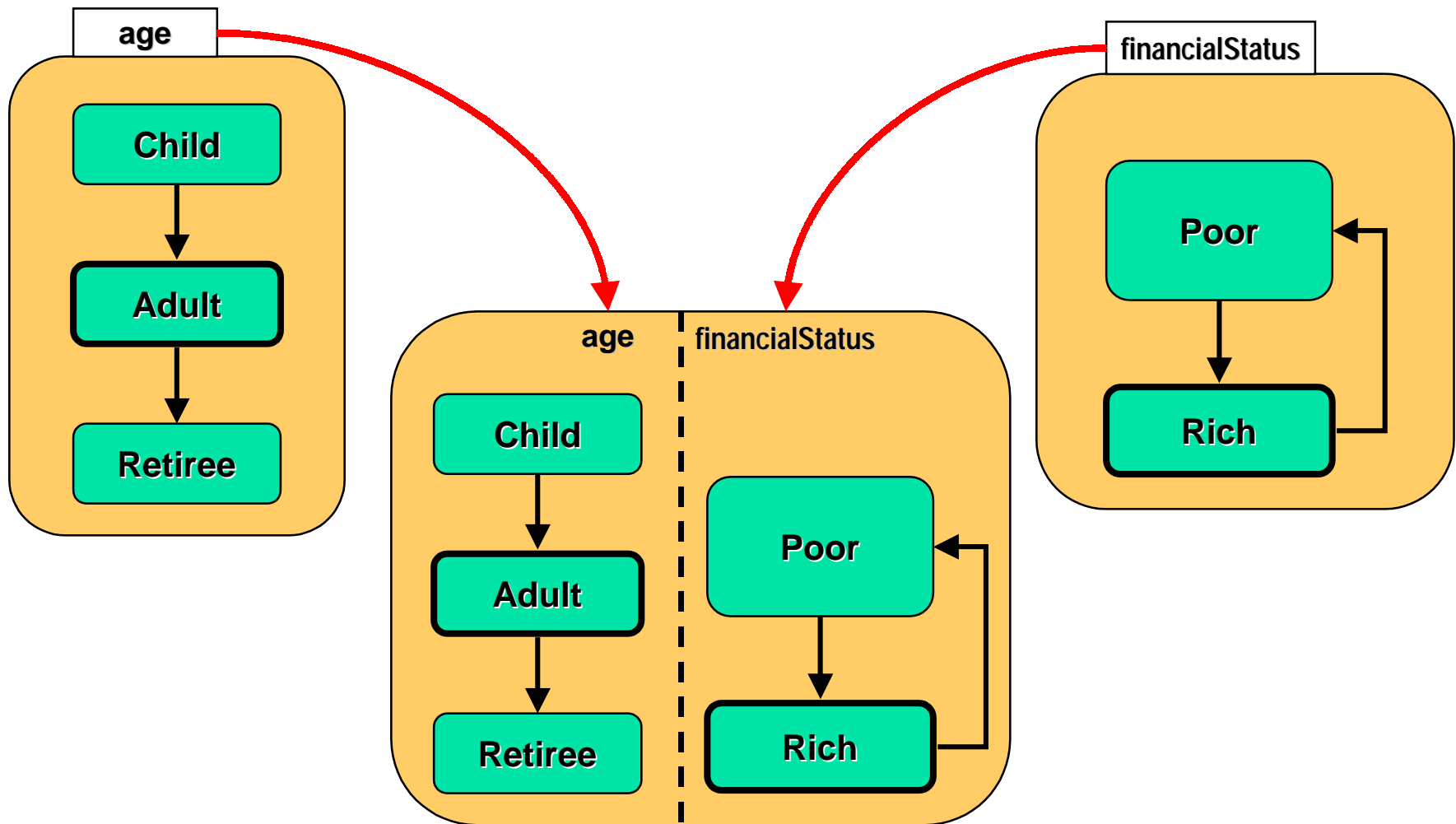
# Orthogonality

- Multiple simultaneous perspectives on the same entity



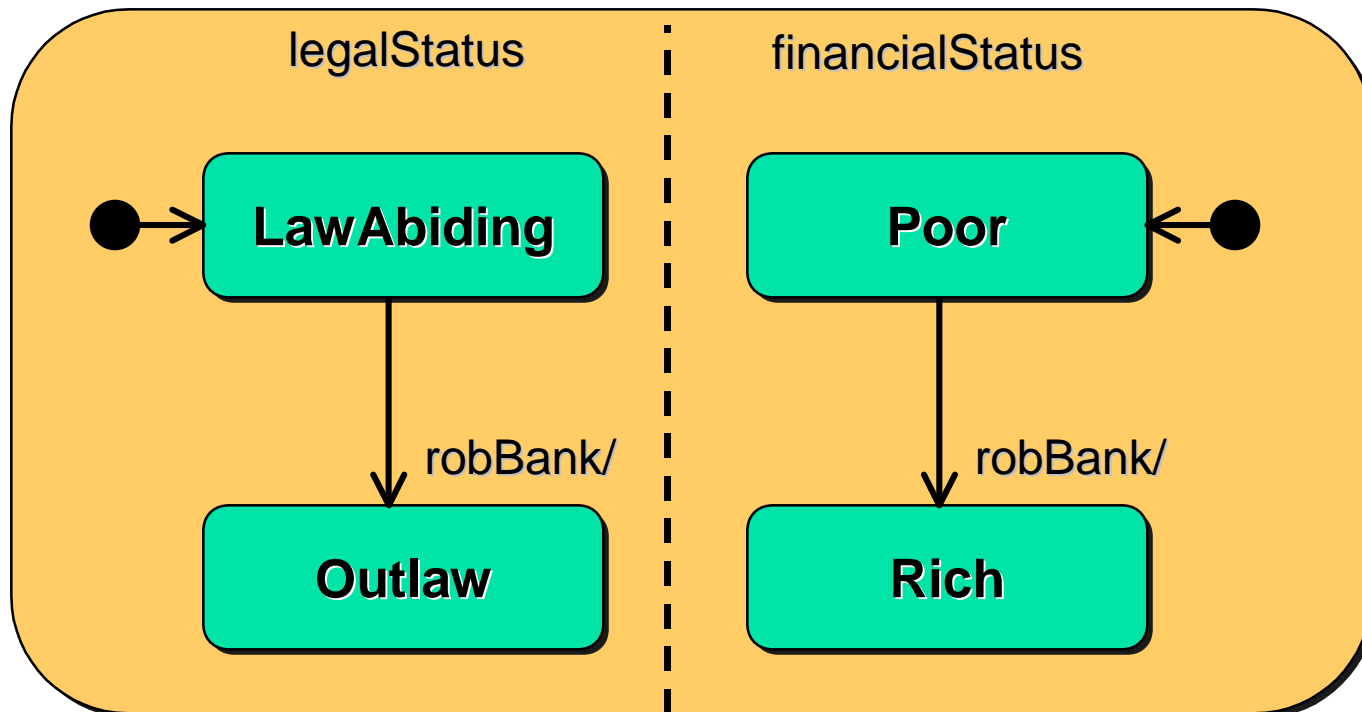
# Orthogonal Regions

- Combine multiple simultaneous descriptions



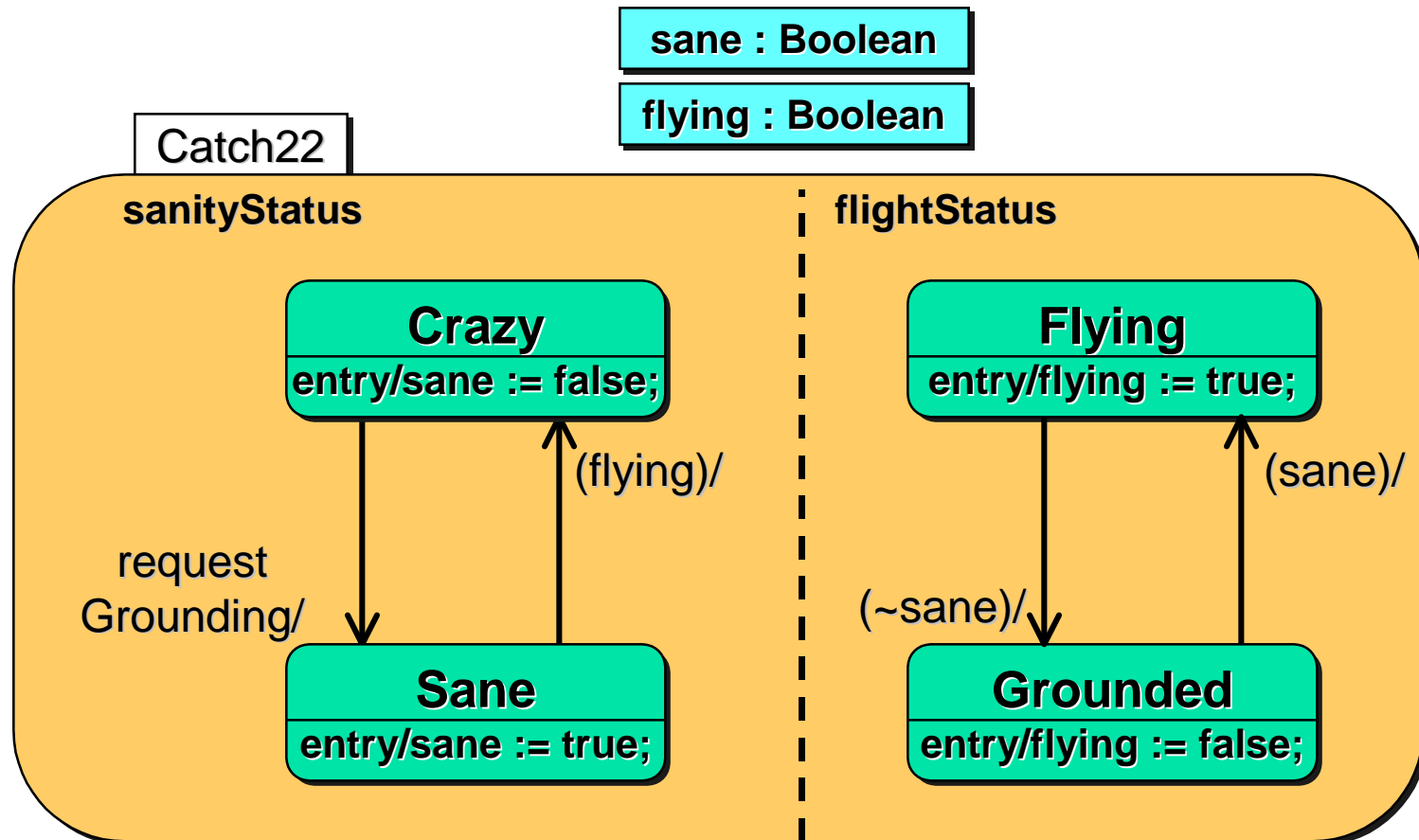
# Orthogonal Regions - Semantics

- All mutually orthogonal regions detect the same events and respond to them “simultaneously”
  - usually reduces to interleaving of some kind



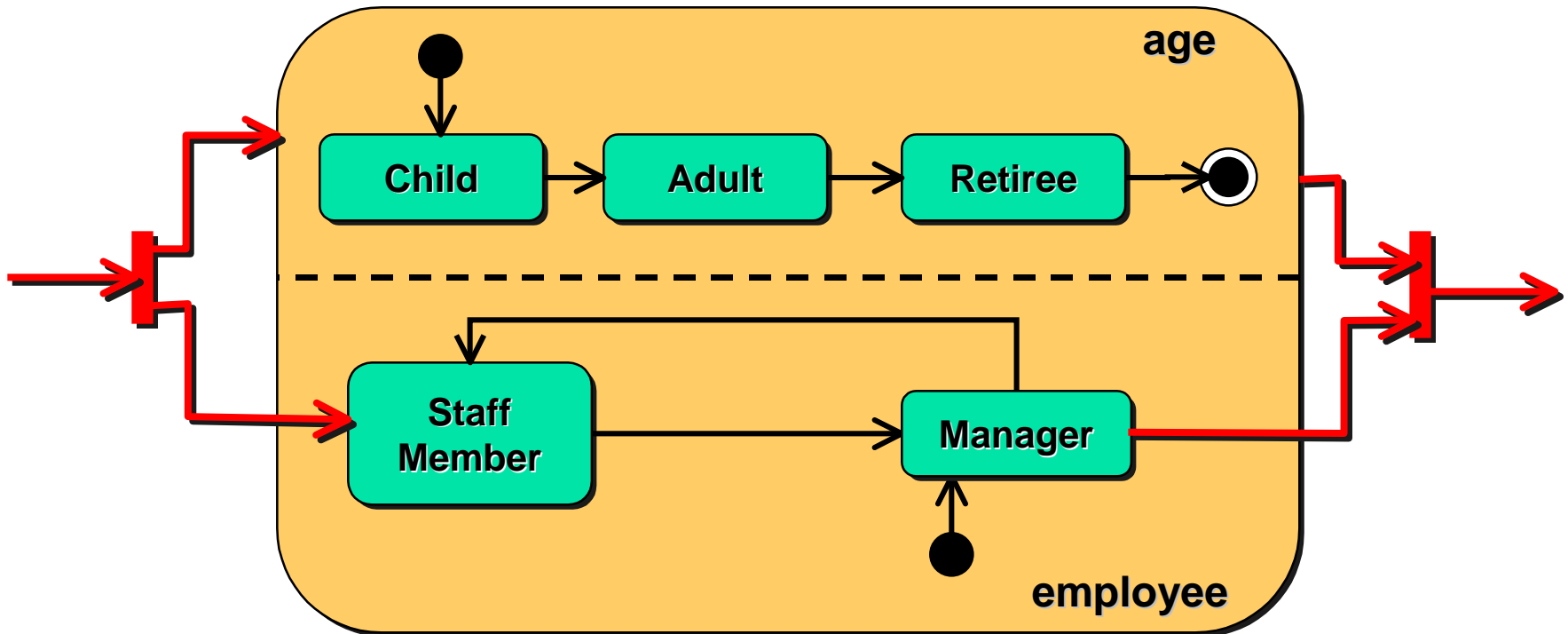
# Interactions Between Regions

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



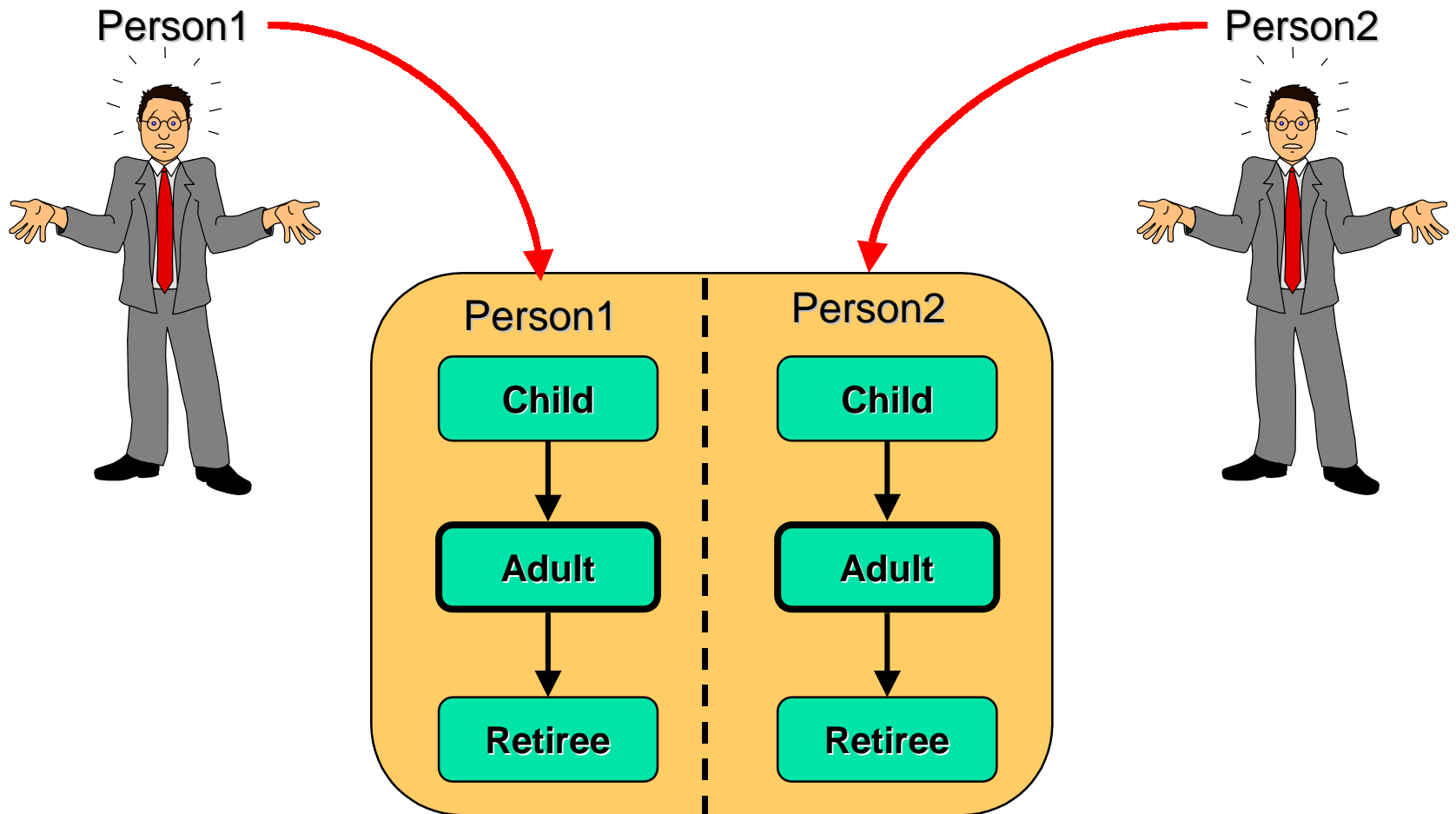
# Transition Forks and Joins

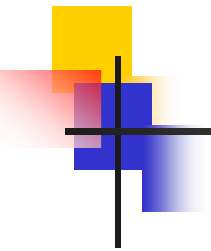
- For transitions into/out of orthogonal regions:



# Common Misuse of Orthogonality

- Using regions to model independent objects



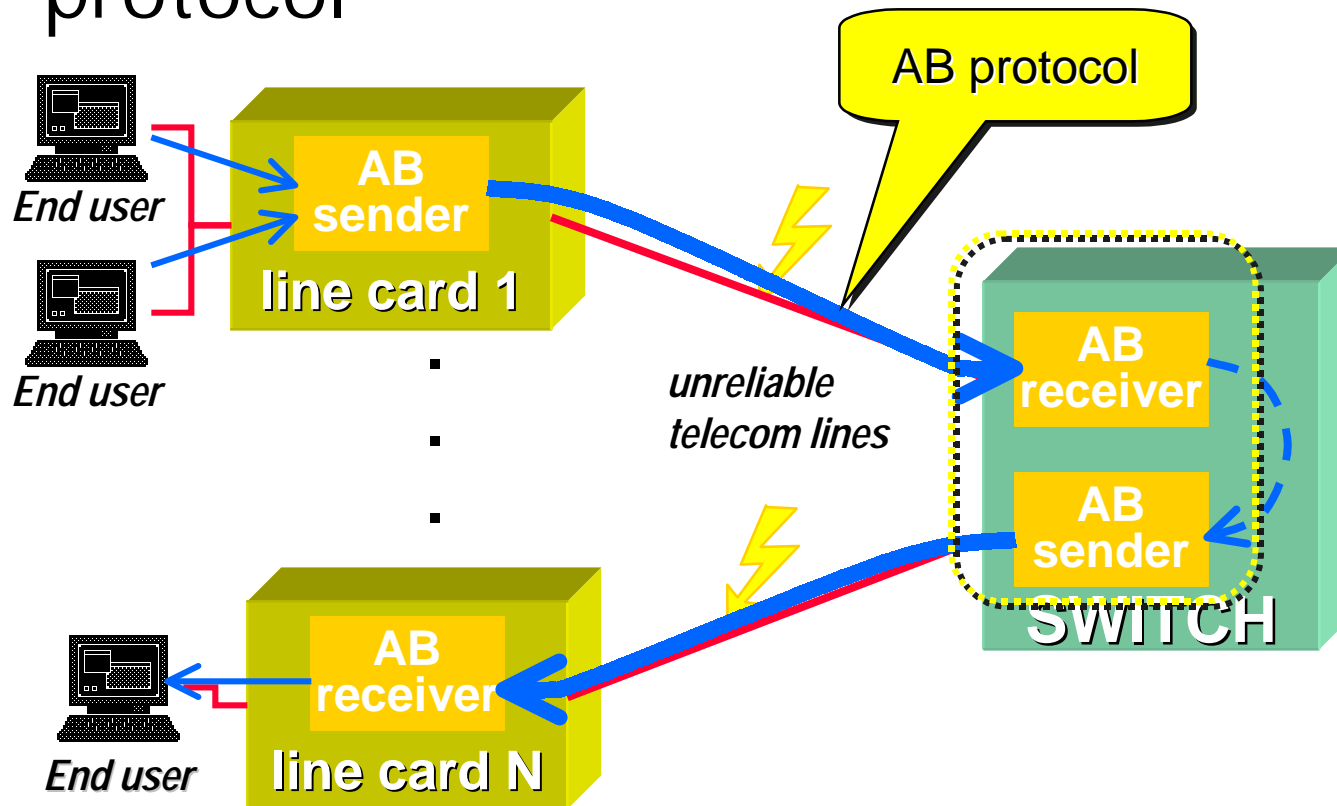


---

Basic State Machine Concepts  
Statecharts and Objects  
Advanced Modeling Concepts  
**Case Study**  
Wrap Up

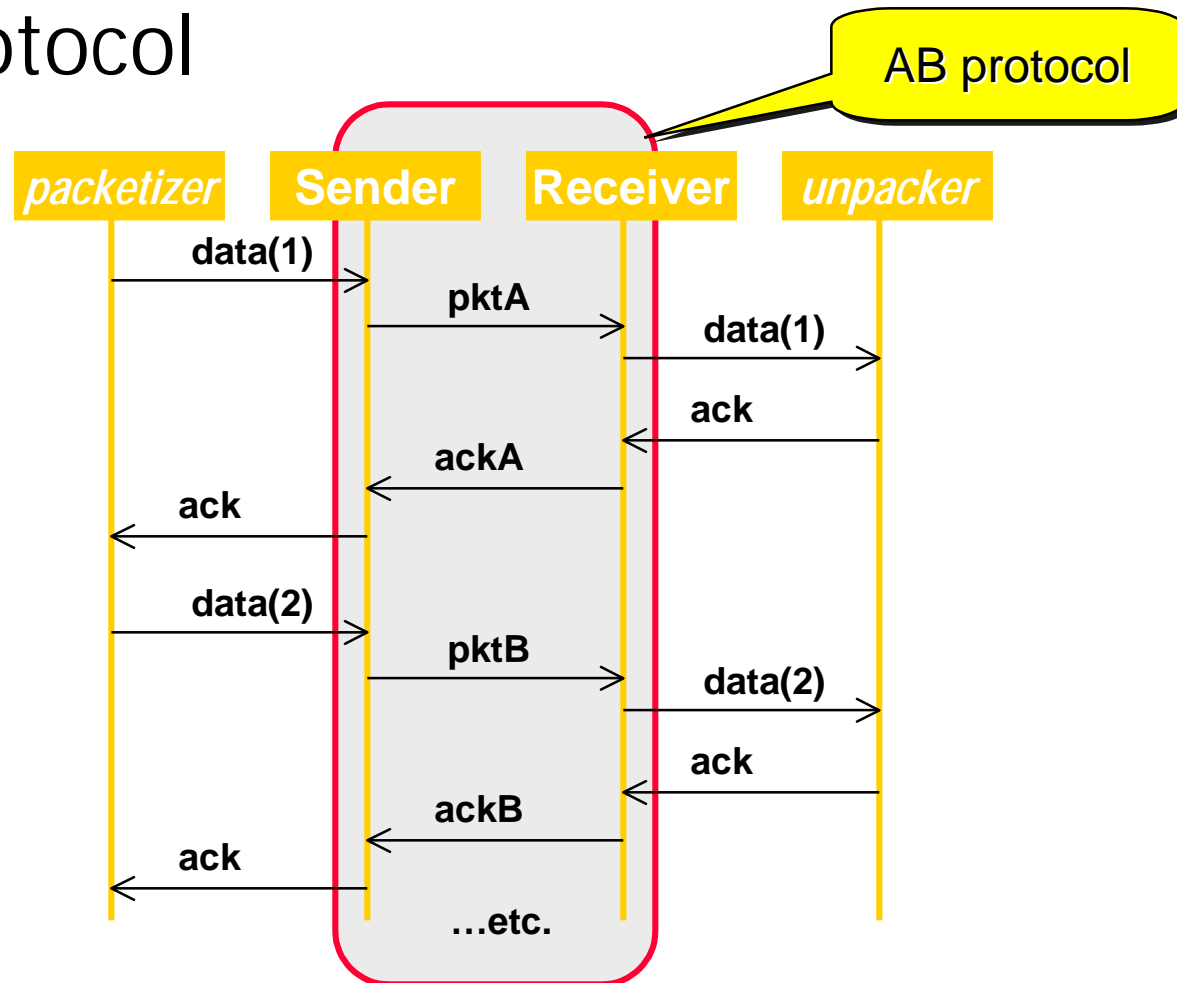
# Case Study: Protocol Handler

- A multi-line packet switch that uses the alternating-bit protocol as its link protocol



# Alternating Bit Protocol (1)

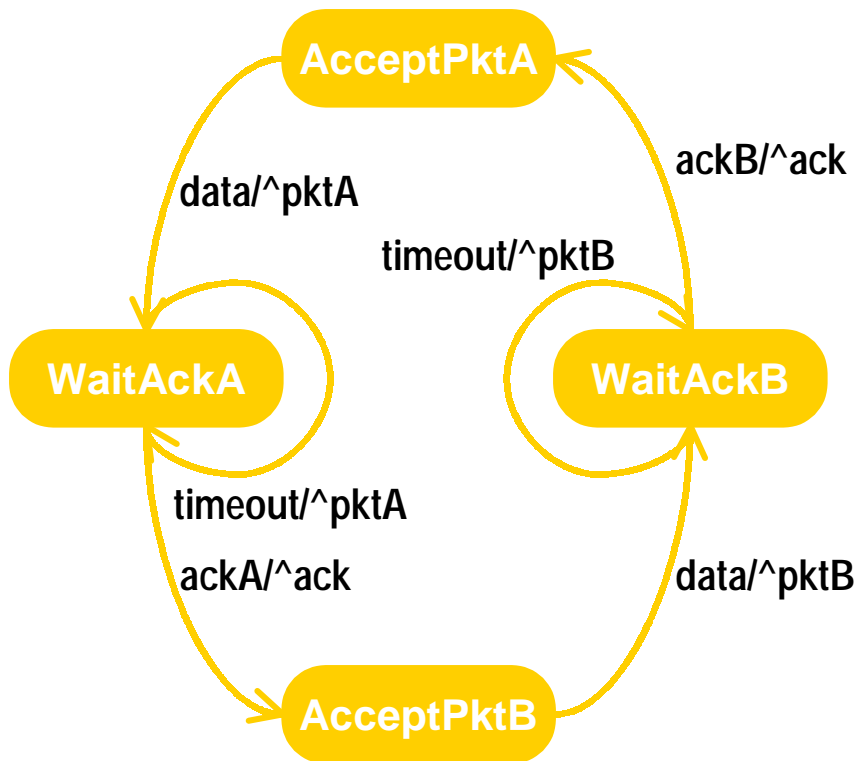
- A simple one-way point-to-point packet protocol



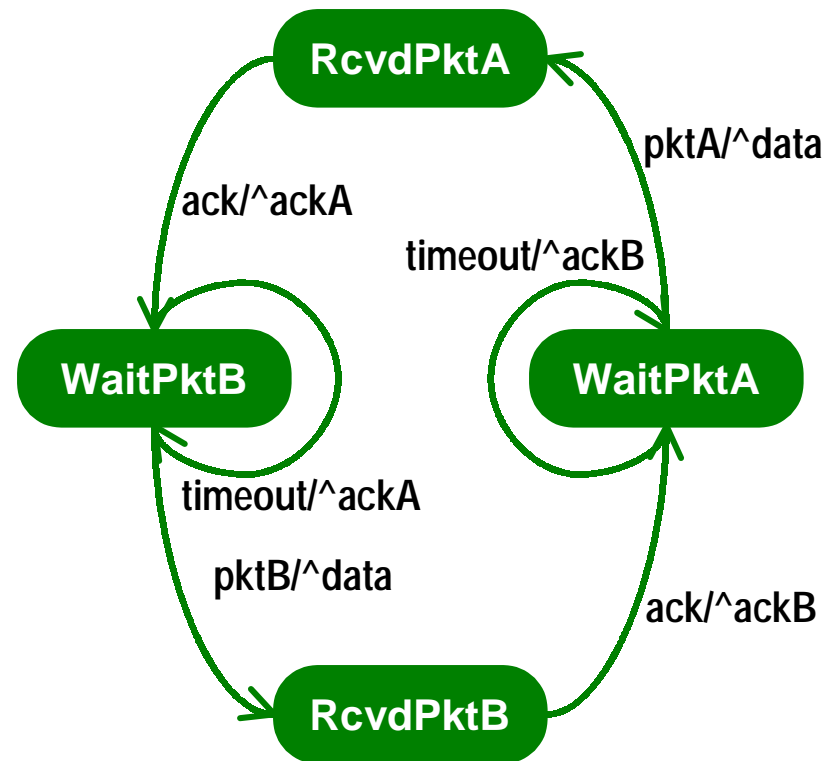
# Alternating Bit Protocol (2)

- State machine specification

Sender SM

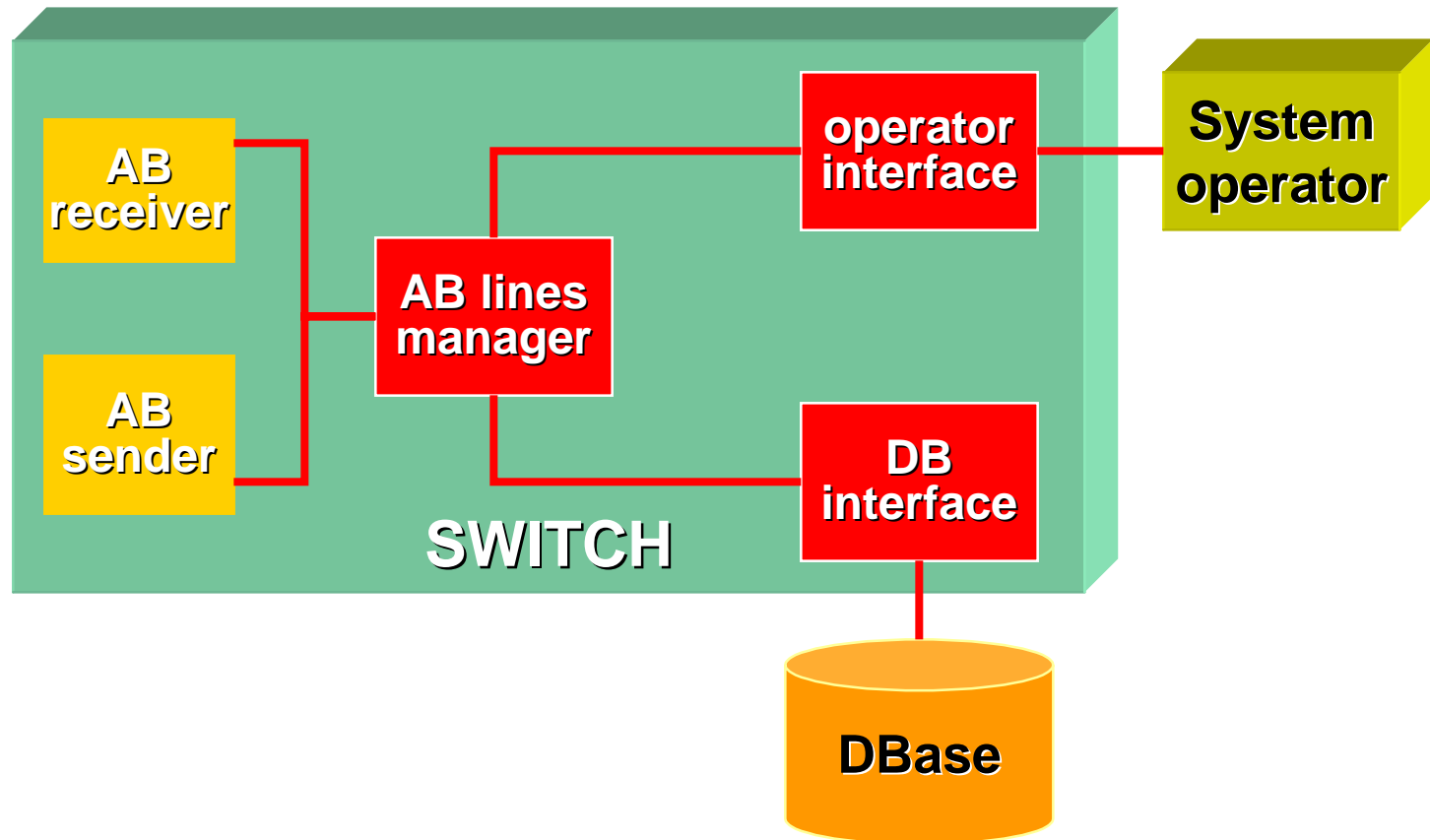


Receiver SM



# Additional Considerations

- Support (control) infrastructure





# Control

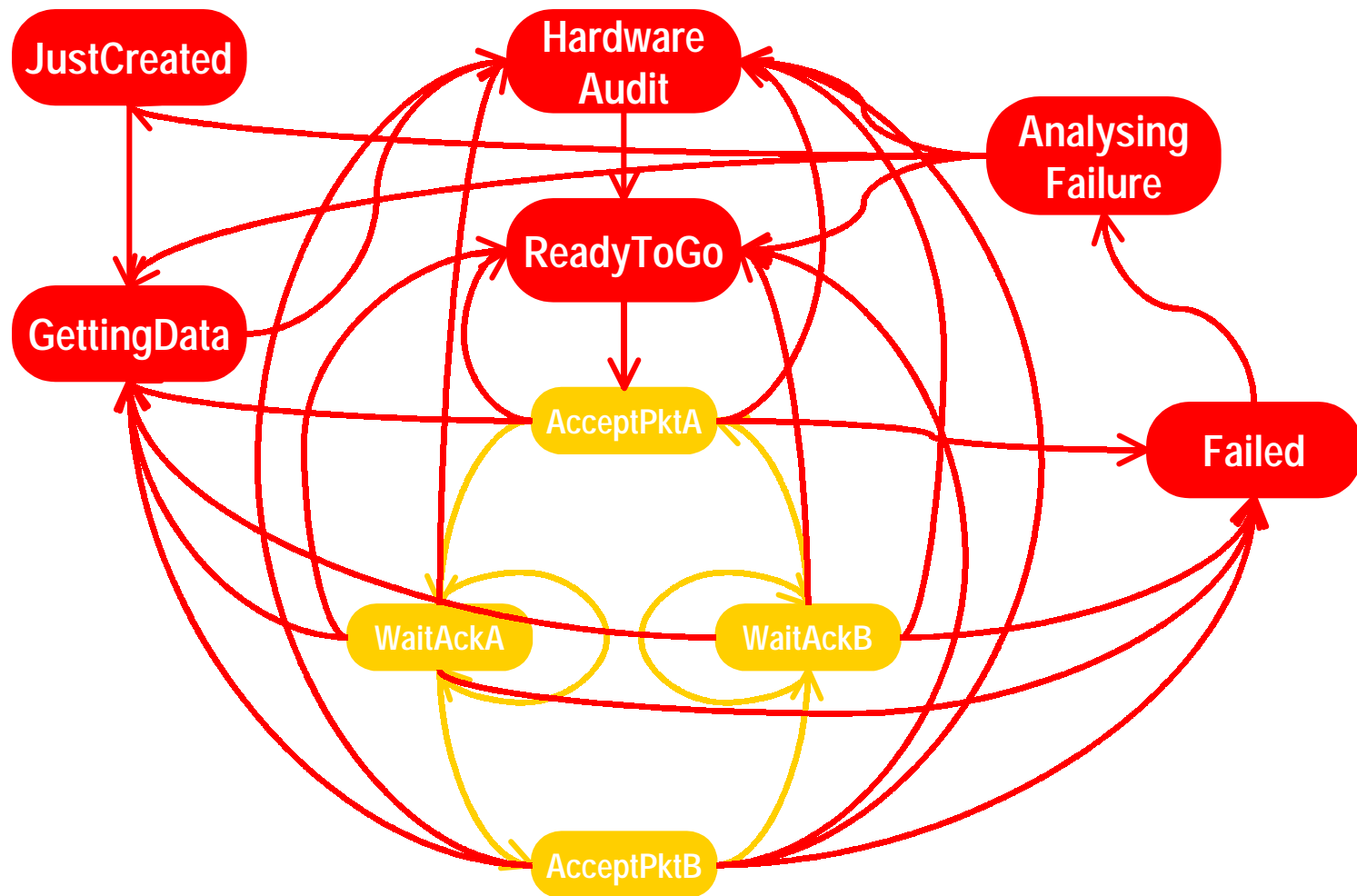
---

*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:

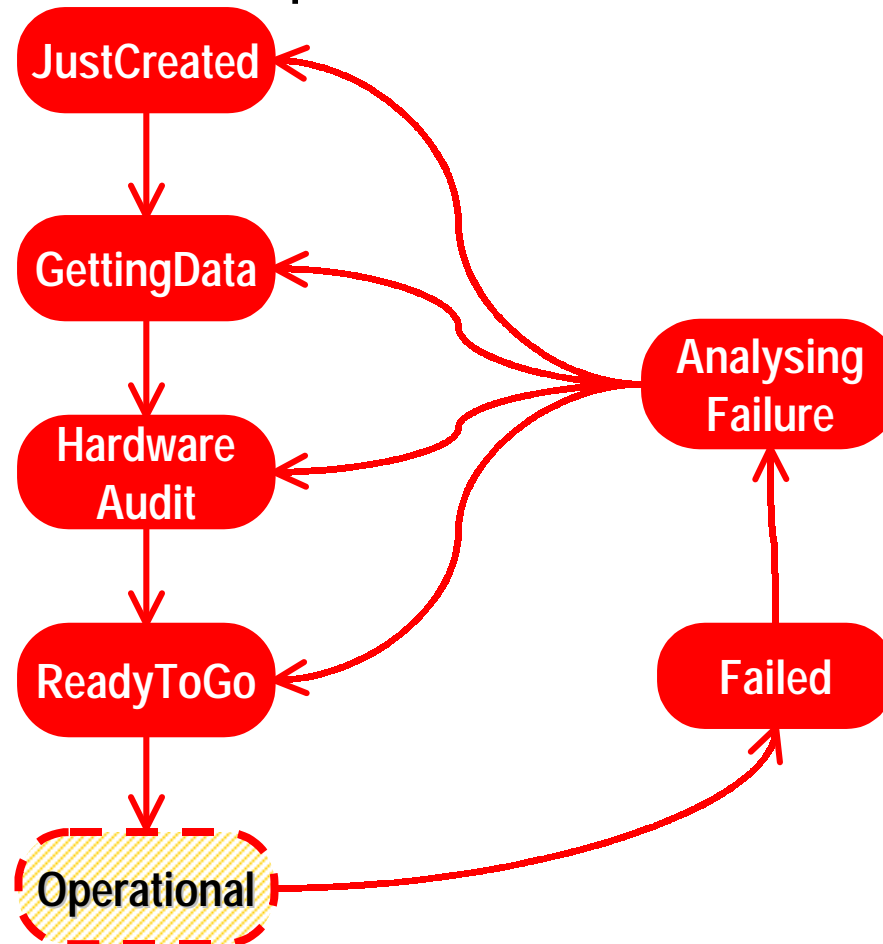
- system/component start-up and shut-down
- failure detection/reporting/recovery
- system administration, maintenance, and provisioning
- (on-line) software upgrade

# Retrofitting Control Behavior



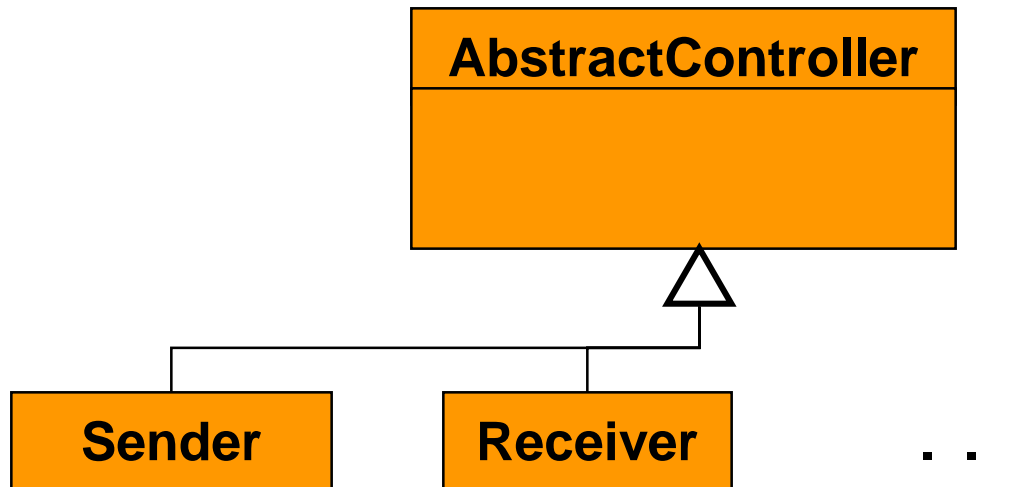
# The Control Automaton

- In isolation, the same control behavior appears much simpler

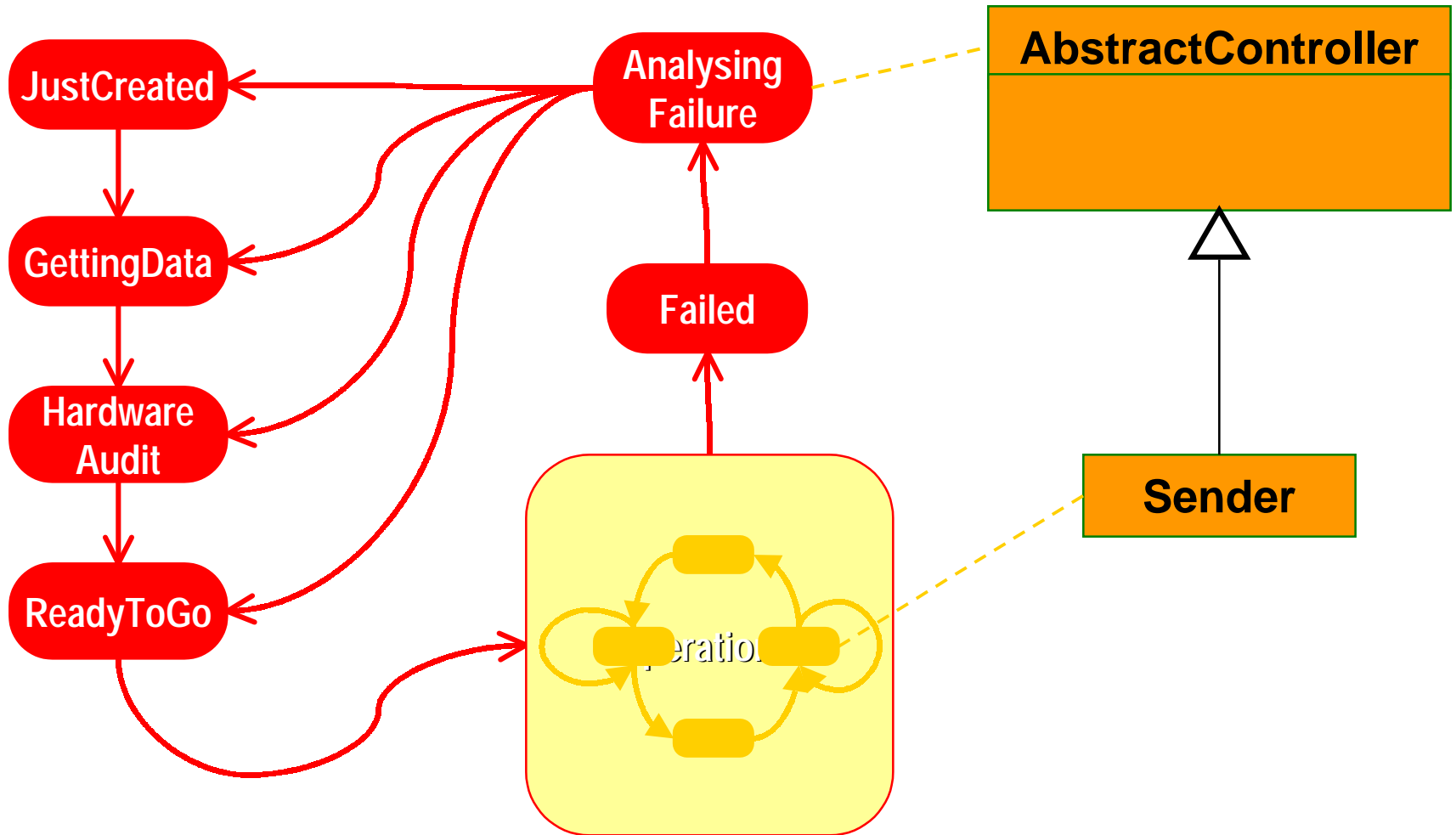


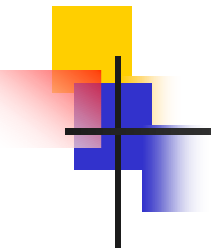
# Exploiting Inheritance

- Abstract control classes can capture the common control behavior



# Exploiting Hierarchical States





---

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



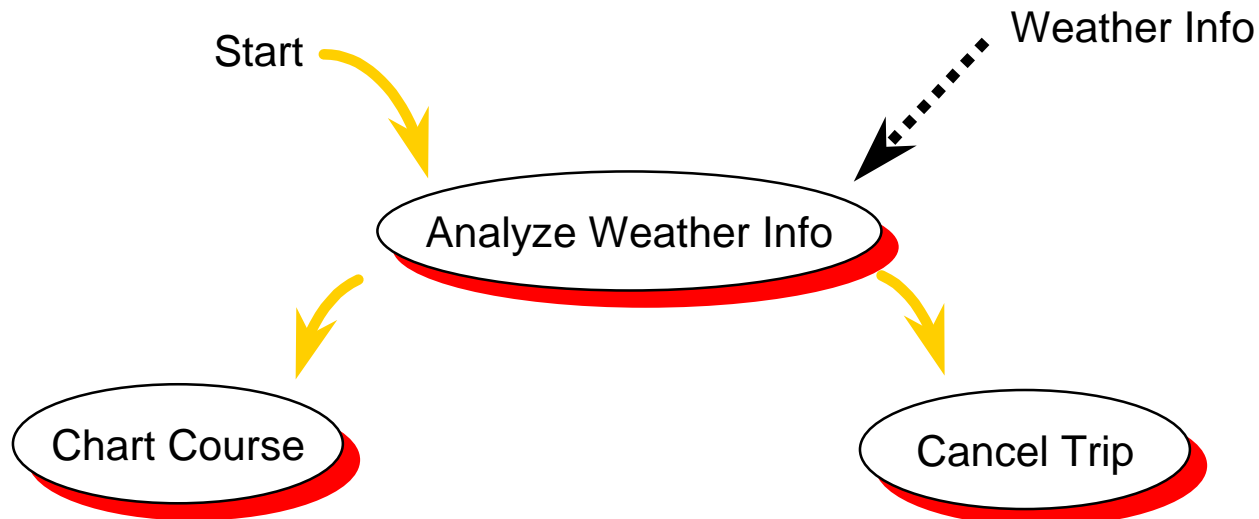
# Activity Diagram Applications

---

- 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

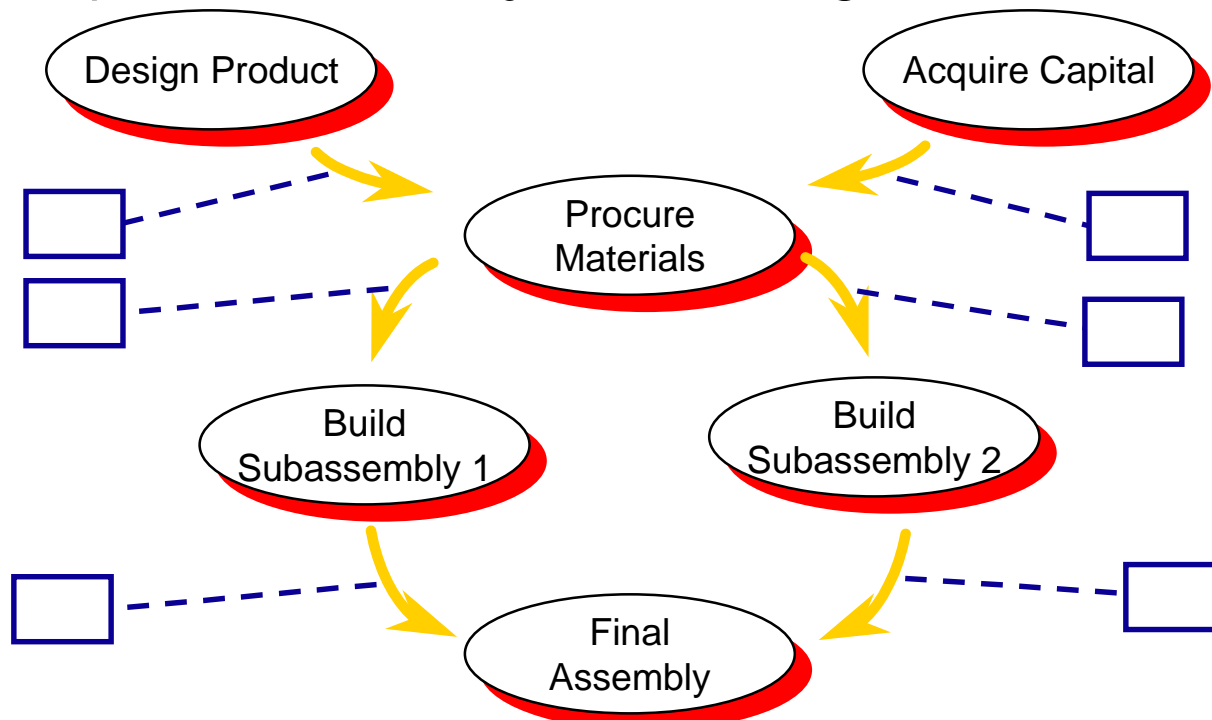
- Each step is taken when the previous one finishes ...
- ...regardless of whether inputs are available, accurate, or complete (“pull”).
- Emphasis is on order in which steps are taken.



Not UML  
Notation!

# Object/Data Flow

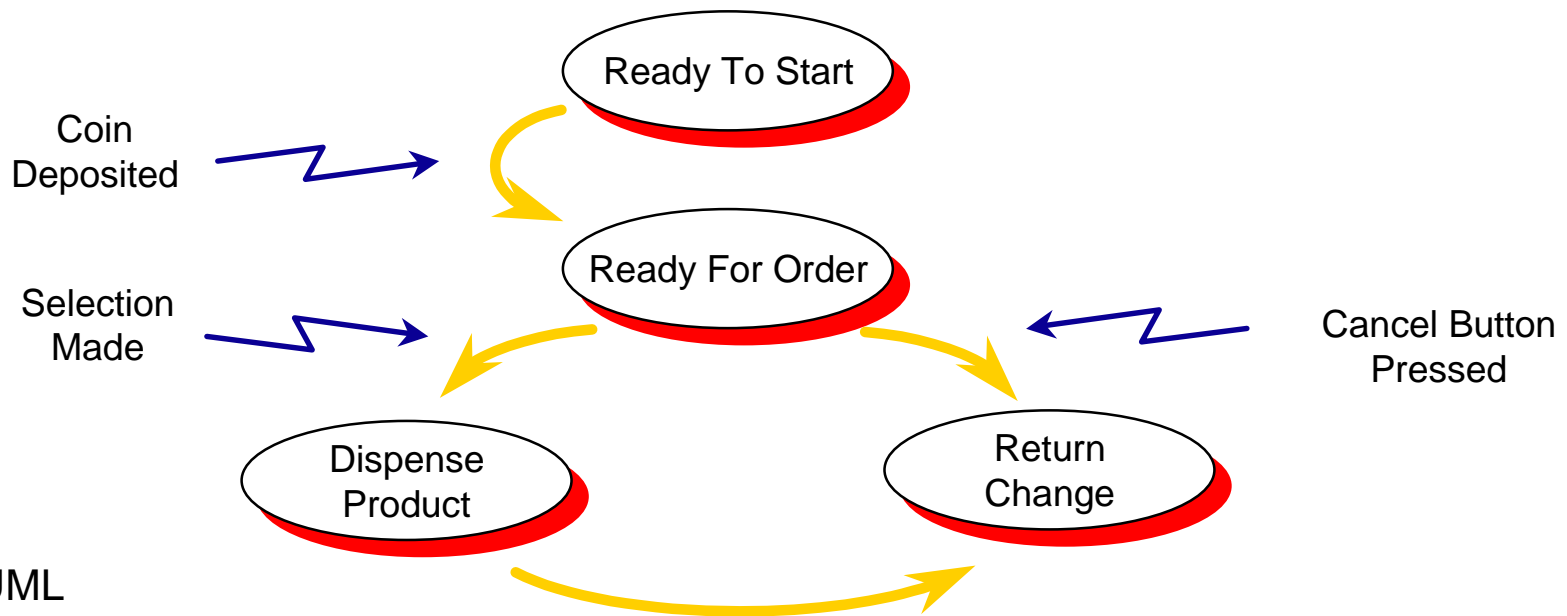
- Each step is taken when all the required input objects/data are available ...
- ... and only when all the inputs are available ("push").
- Emphasis is on objects flowing between steps.



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.



Not UML  
Notation



# Activity Diagrams Based on State Machines

---

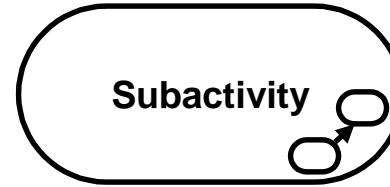
- Currently activity graphs are modeled as a kind of state machine.
- Modeler doesn't normally need to be aware of this sleight-of-hand ...
- ... but will notice that "state" is used in the element names.
- Activity graphs will become independent of state machines in UML 2.0.

# Kinds of Steps in Activity Diagrams

- Action (State)



- Subactivity (State)



- 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.

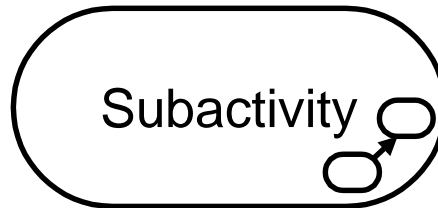
# Action (State)



Action

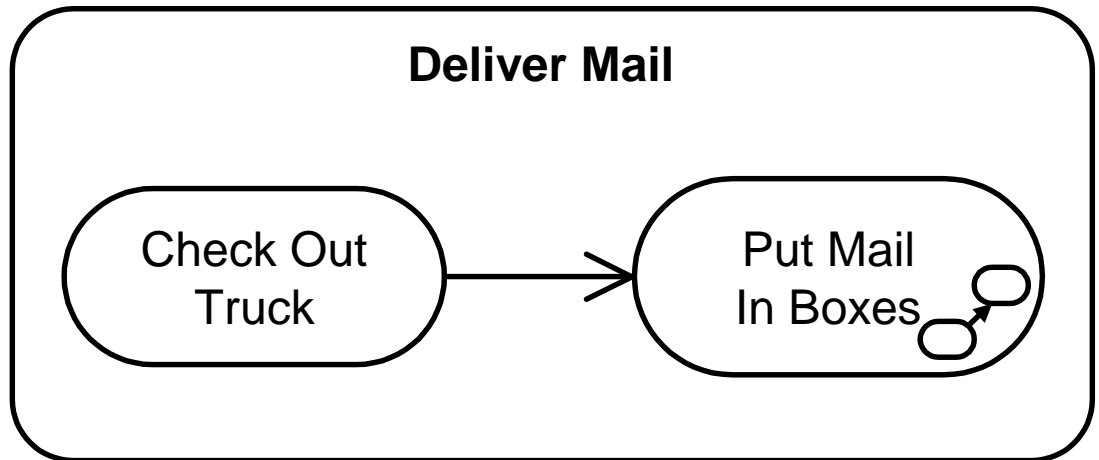
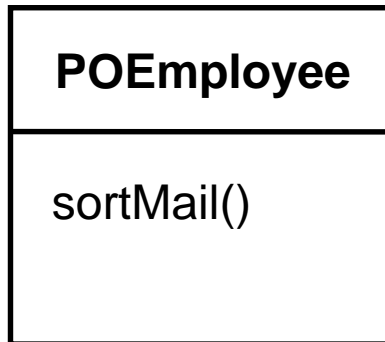
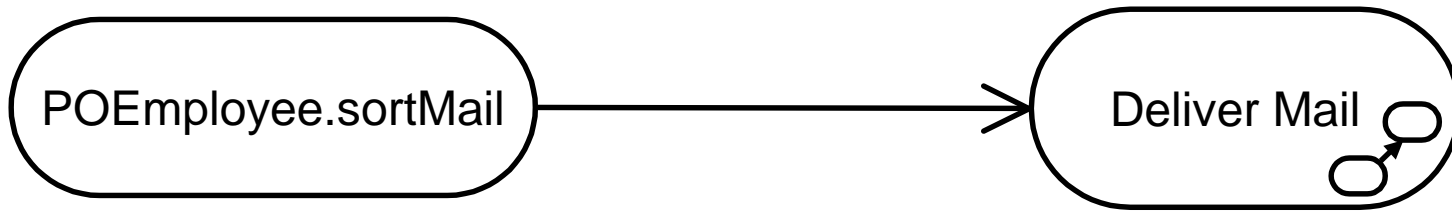
- An action is used for anything that does not directly start another activity graph, like invoking an operation on an object, or running a user-specified action.
- However, an action can invoke an operation that has another activity graph as a method (possible polymorphism).

# Subactivity (State)

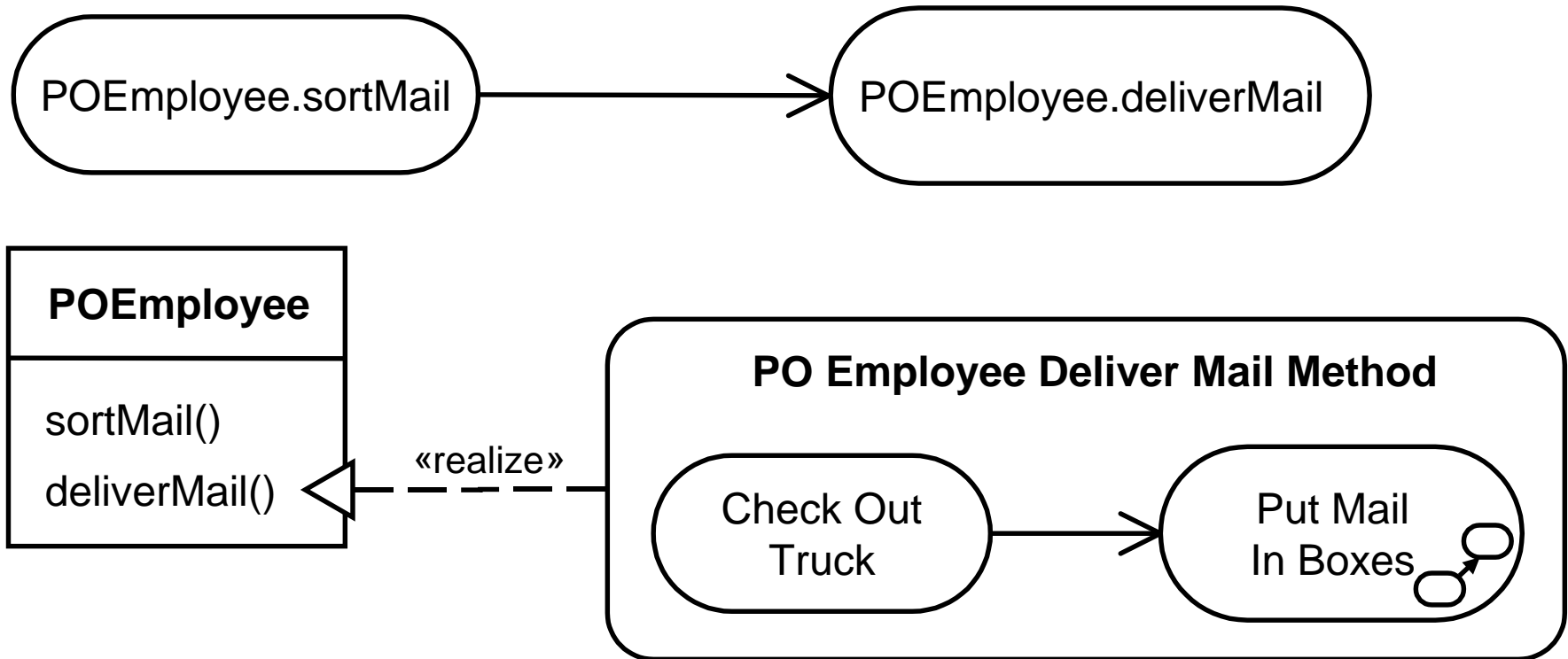


- A subactivity (state) starts another activity graph without using an operation.
- Used for functional decomposition, non-polymorphic applications, like many workflow systems.
- The invoked activity graph can be used by many subactivity states.

# Example



# Activity Graph as Method



- 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 2.0.

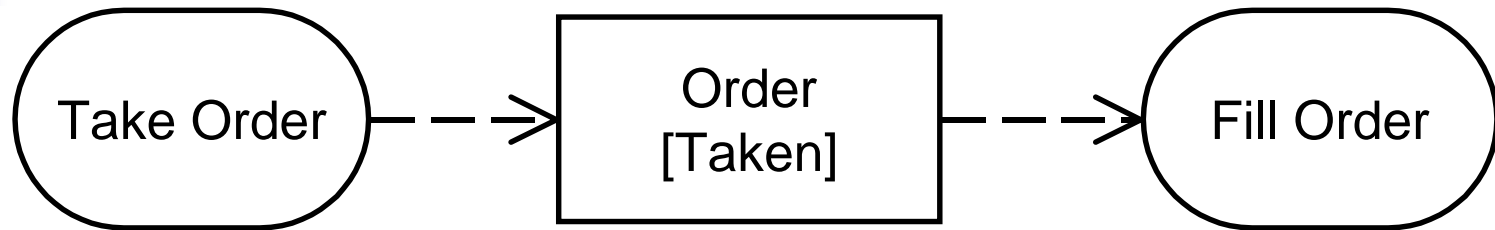
# Object Flow (State)



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)



- 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.

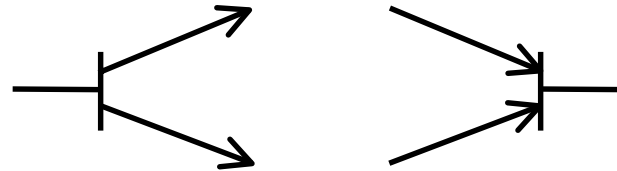
# Coordinating Steps

- Inherited from state machines

- Initial state 

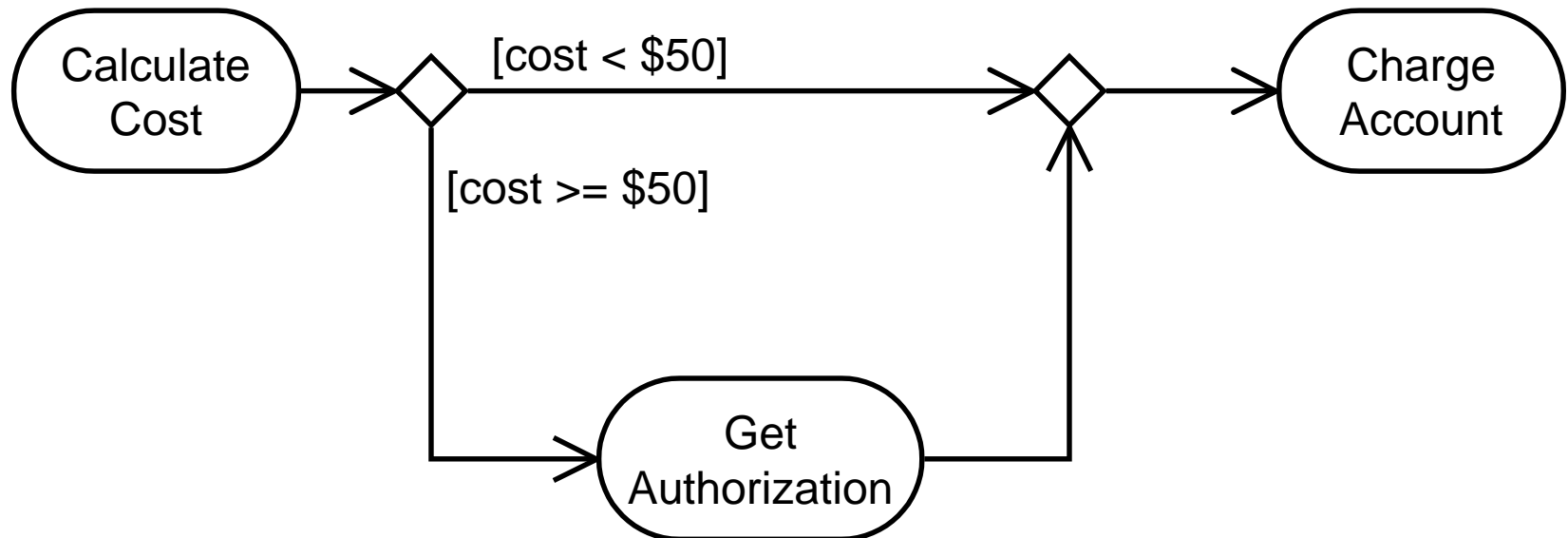
- Final state 

- Fork and join



# Coordinating Steps

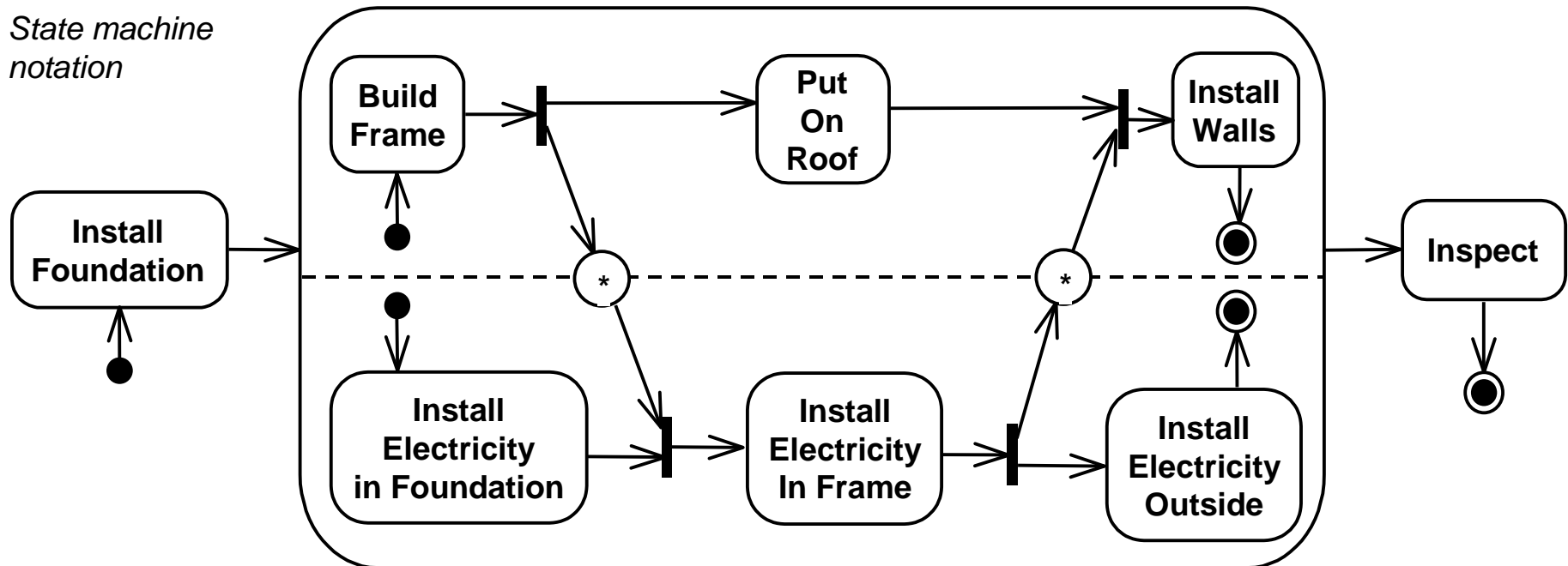
- Decision point and merge ( $\diamond$ ) are inherited from state machines.
- For modeling conventional flow chart decisions.



# Coordinating Steps

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

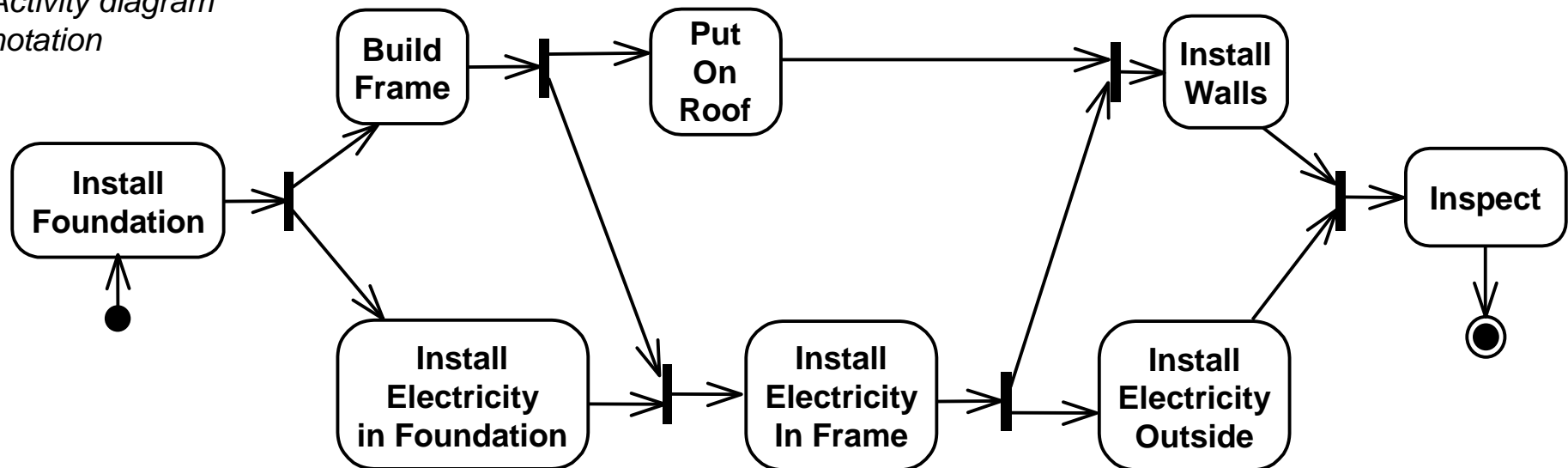
*State machine notation*



# Convenience Features (Synch State)

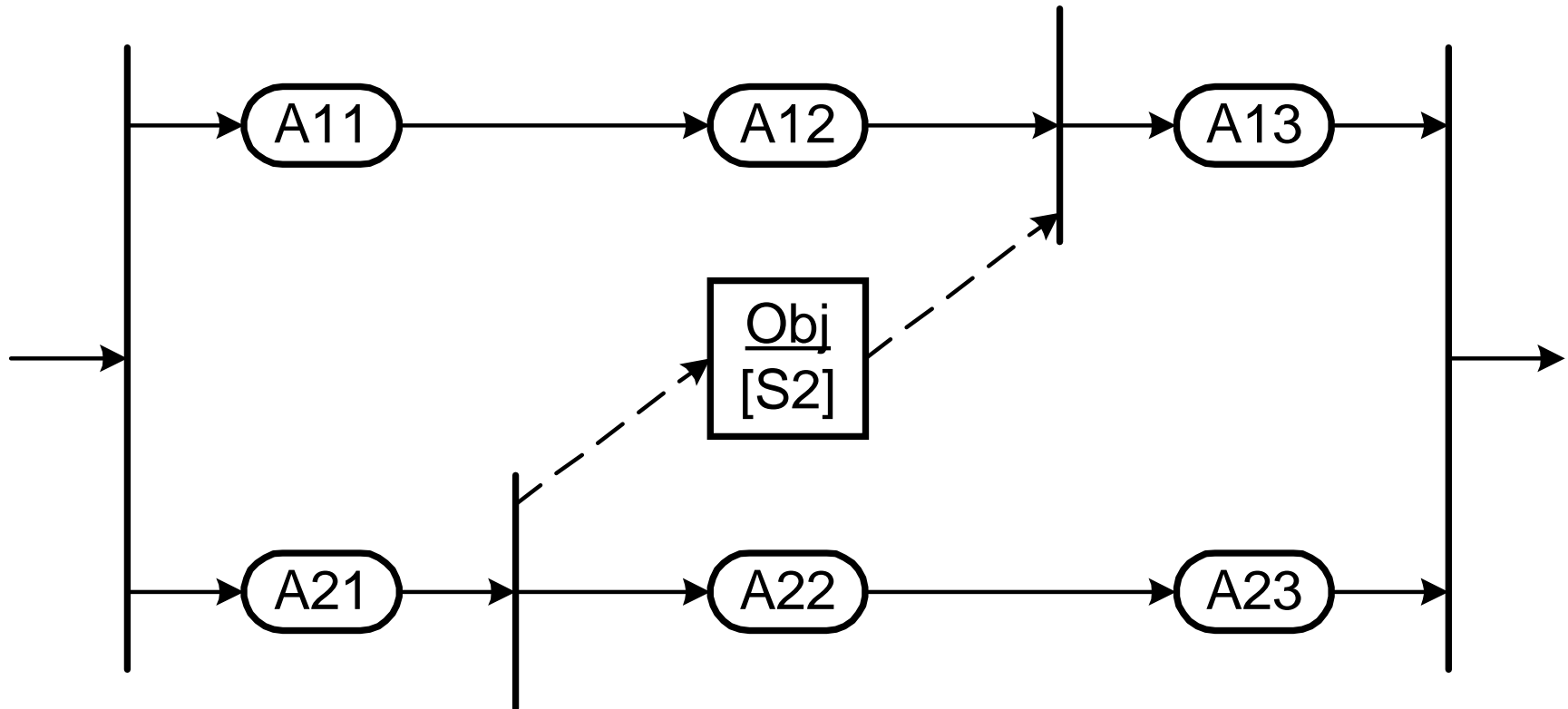
- Forks and joins do not require composite states.
- Synch states may be omitted for the common case (unlimited bound and one incoming and outgoing transition).

*Activity diagram notation*



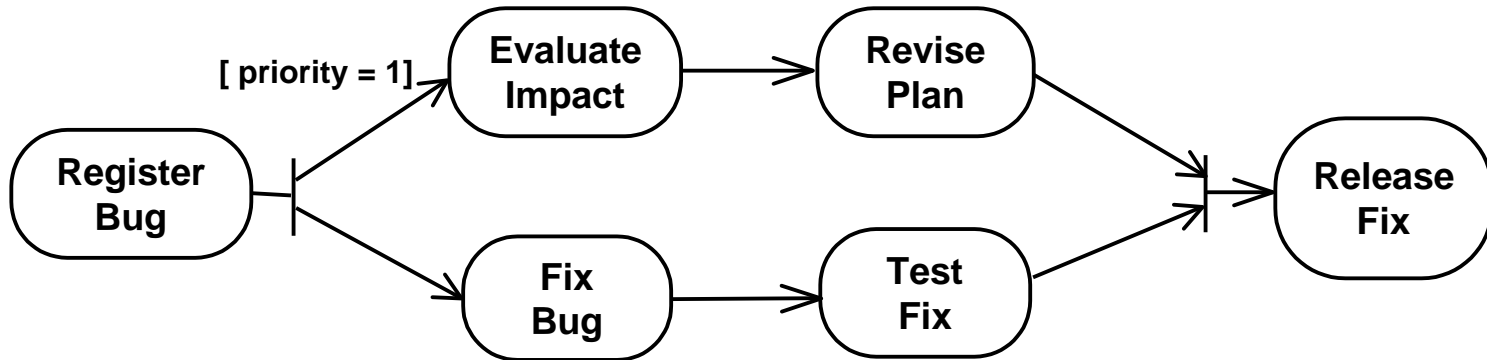
# Convenience Features (Synch State)

- Object flow states can be synch states

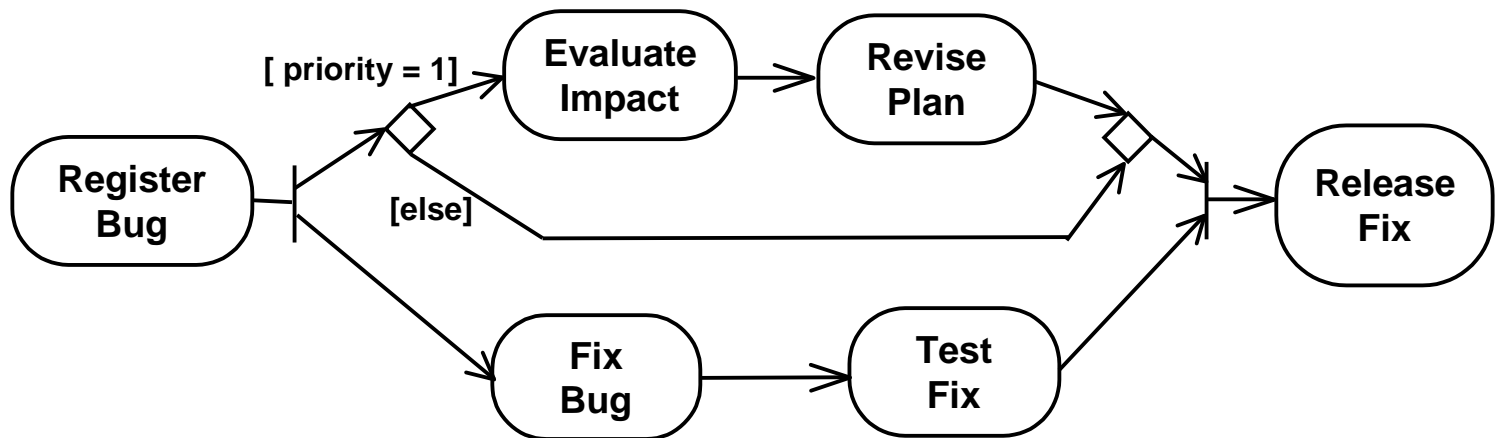


# Convenience Features

- Fork transitions can have guards.

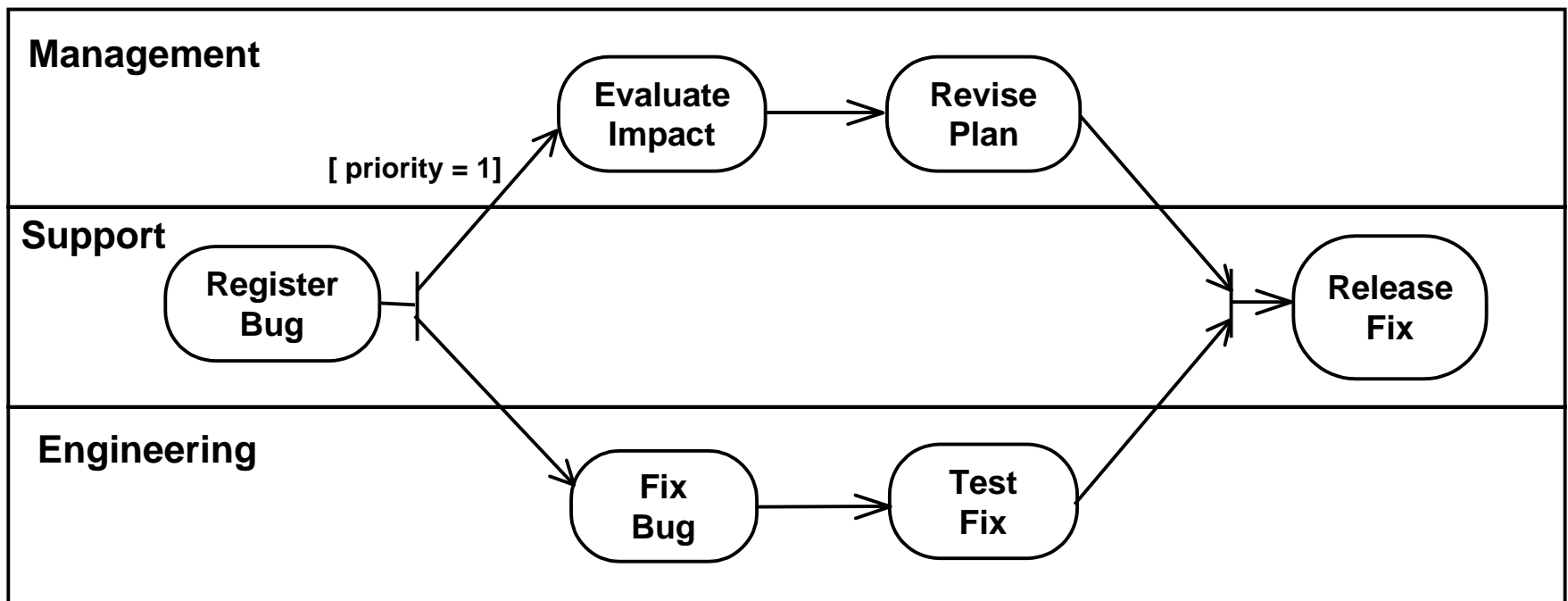


- Instead of doing this:



# 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

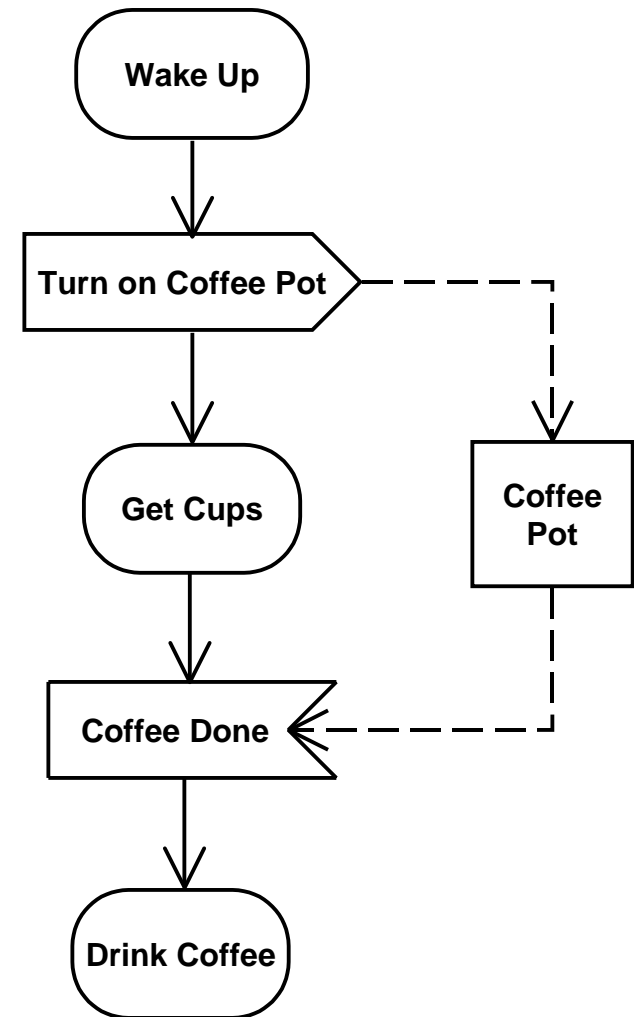


- ... translates to a transition with a send action.

- Signal receipt icon

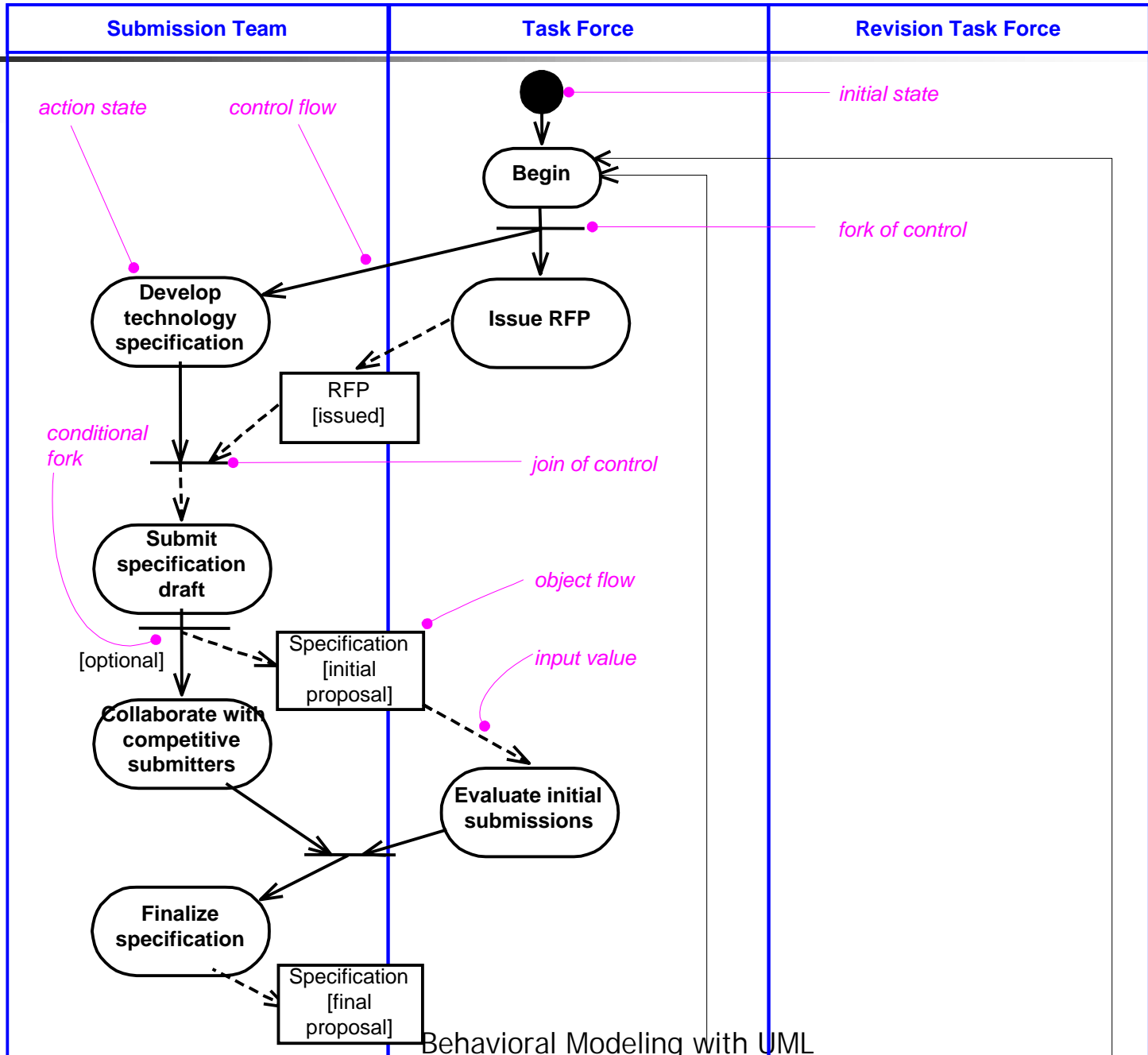


- ... translates to a wait state (a state with no action and a signal trigger event).



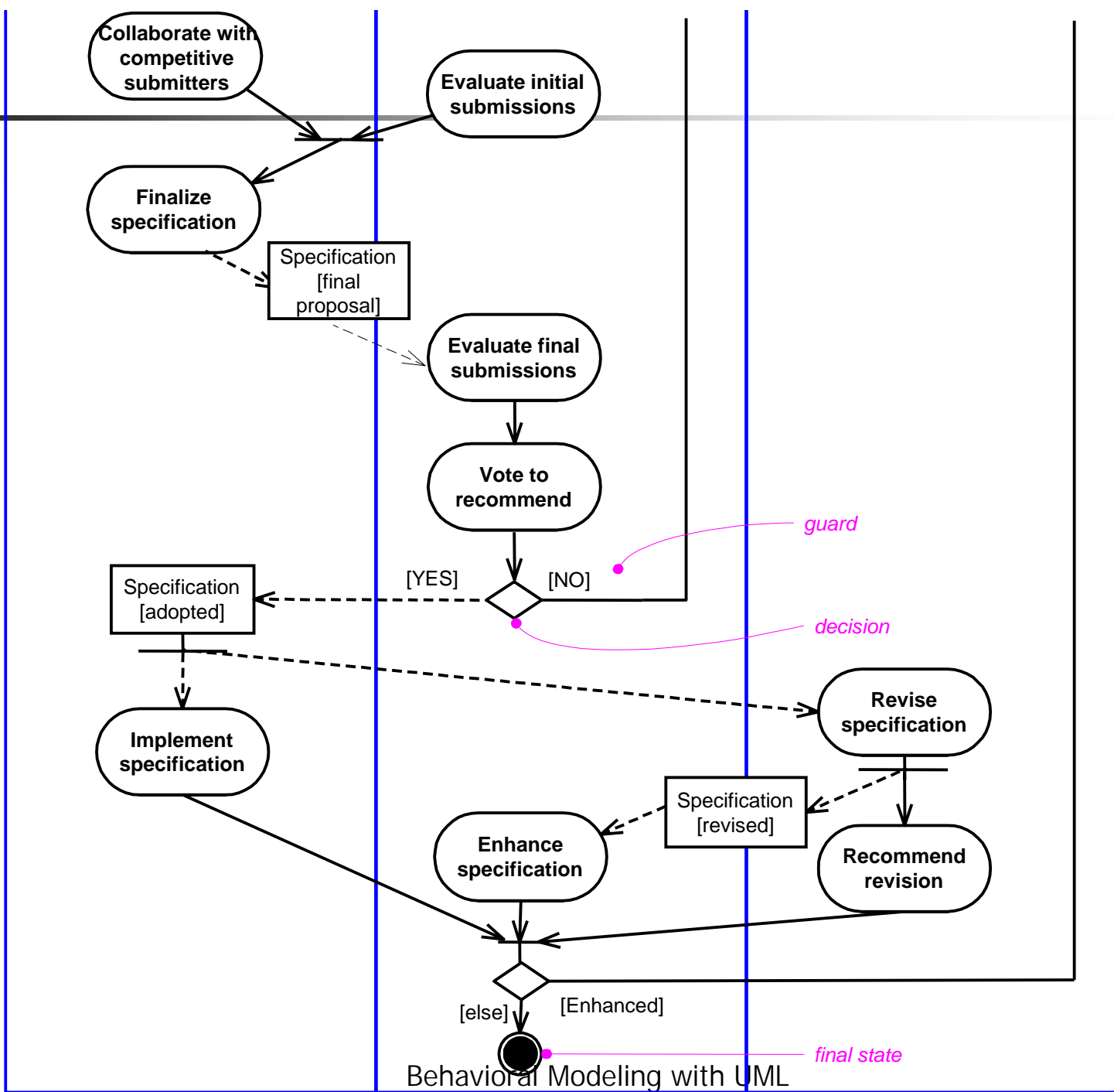
# Case Study

Adapted from  
Kobryn, "UML 2001"  
Communications of the ACM  
October 1999



# Case Study

Adapted from  
Kobryn, "UML 2001"  
Communications of the ACM  
October 1999





# 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).



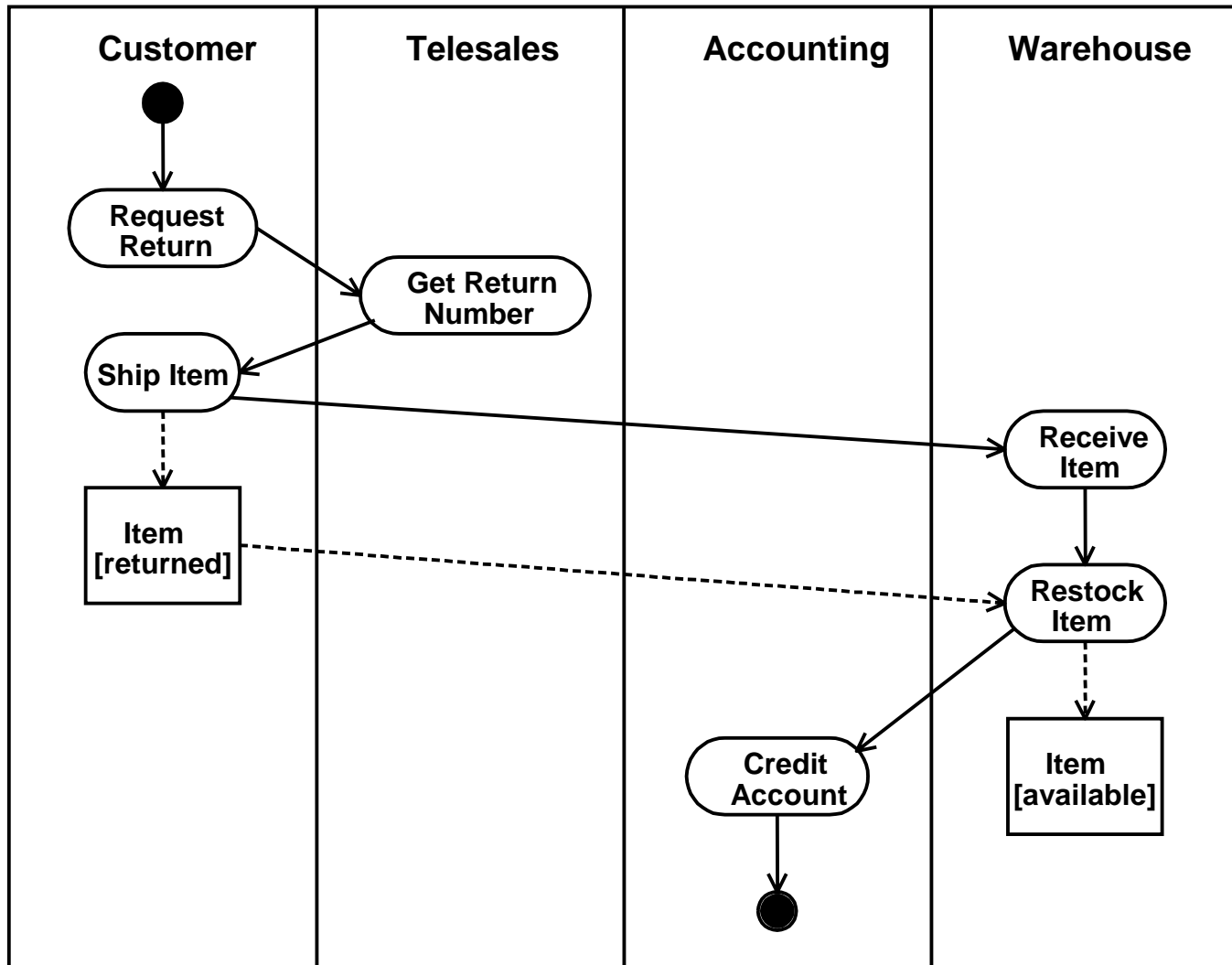
# Activity Diagram Modeling Tips

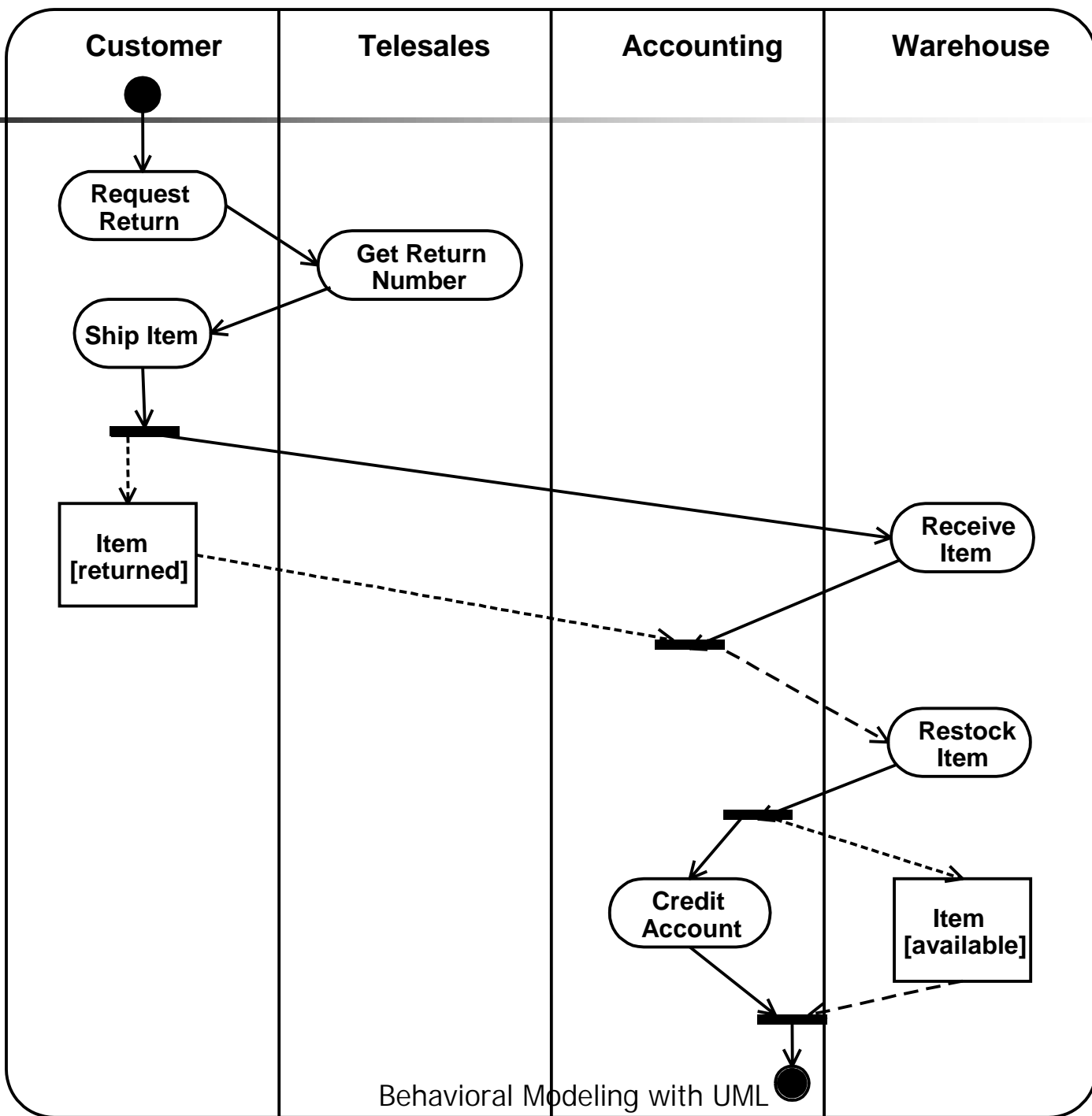
---

- 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:







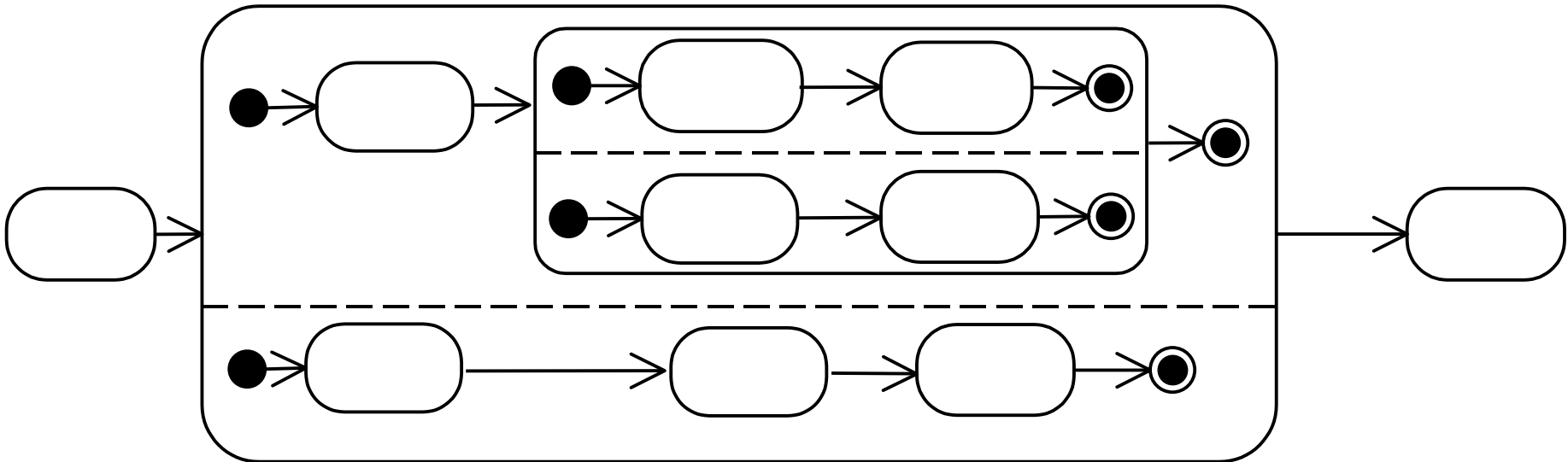
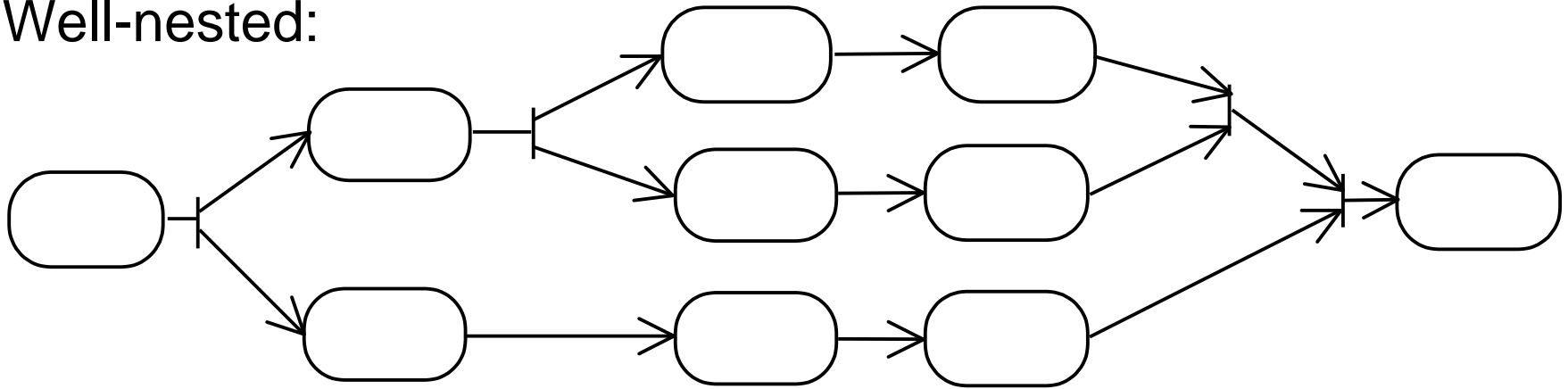
# Activity Diagram Modeling Tips

---

- 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.

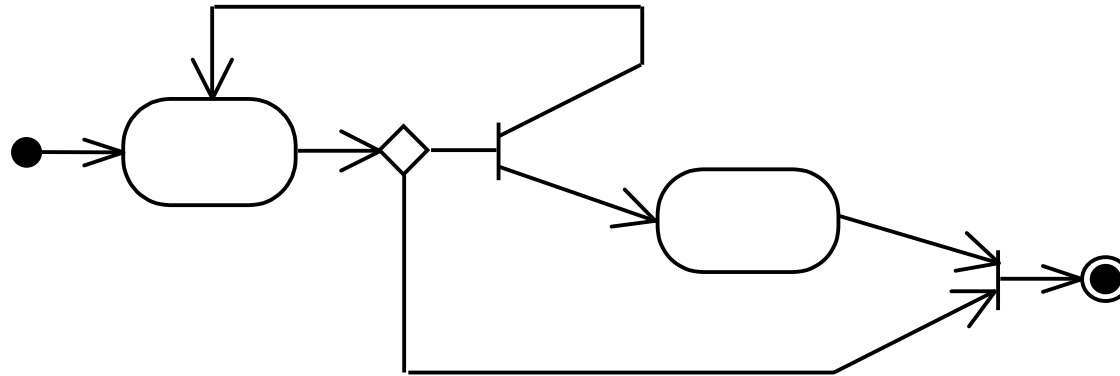
# Activity Diagram Modeling Tips

Well-nested:



# Activity Diagram Modeling Tips

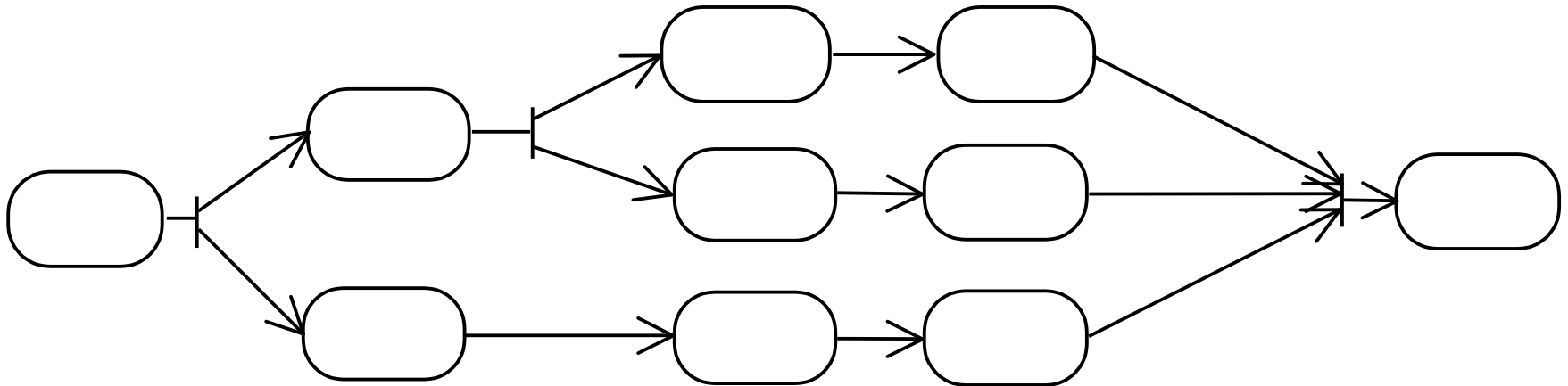
Not well-nested:



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)



# References

---

- [UML 1.3] *OMG UML Specification v. 1.3*,  
OMG doc# ad/06-08-99
- [UML 1.4] *OMG UML Specification v. 1.4*, UML  
Revision Task Force recommended final draft,  
OMG doc# ad/01-02-13.



# Further Info

---

## ■ Web:

- UML 1.4 RTF: [www.celigent.com/omg/umlrtf](http://www.celigent.com/omg/umlrtf)
- OMG UML Tutorials: [www.celigent.com/omg/umlrtf/tutorials.htm](http://www.celigent.com/omg/umlrtf/tutorials.htm)
- UML 2.0 Working Group:  
[www.celigent.com/omg/adptf/wgs/uml2wg.htm](http://www.celigent.com/omg/adptf/wgs/uml2wg.htm)
- OMG UML Resources: [www.omg.org/uml/](http://www.omg.org/uml/)

## ■ Email

- [uml-rtf@omg.org](mailto:uml-rtf@omg.org)
- Contributors:
  - Gunnar Övergaard: [gunnar.overgaard@jaczone.com](mailto:gunnar.overgaard@jaczone.com)
  - Bran Selic: [bselic@rational.com](mailto:bselic@rational.com)
  - Conrad Bock: [conrad.bock@kabira.com](mailto:conrad.bock@kabira.com)
  - Morgan Björkander: [morgan.bjorkander@telelogic.se](mailto:morgan.bjorkander@telelogic.se)

## ■ Conferences & workshops

- UML World 2001, New York, June 11-14, 2001
- UML 2001, Toronto, Canada, Oct. 1-5, 2001
- OMG UML Workshop 2001, San Francisco, Dec. 3-6, 2001