
6. Design Phase: Modeling of System Behavior

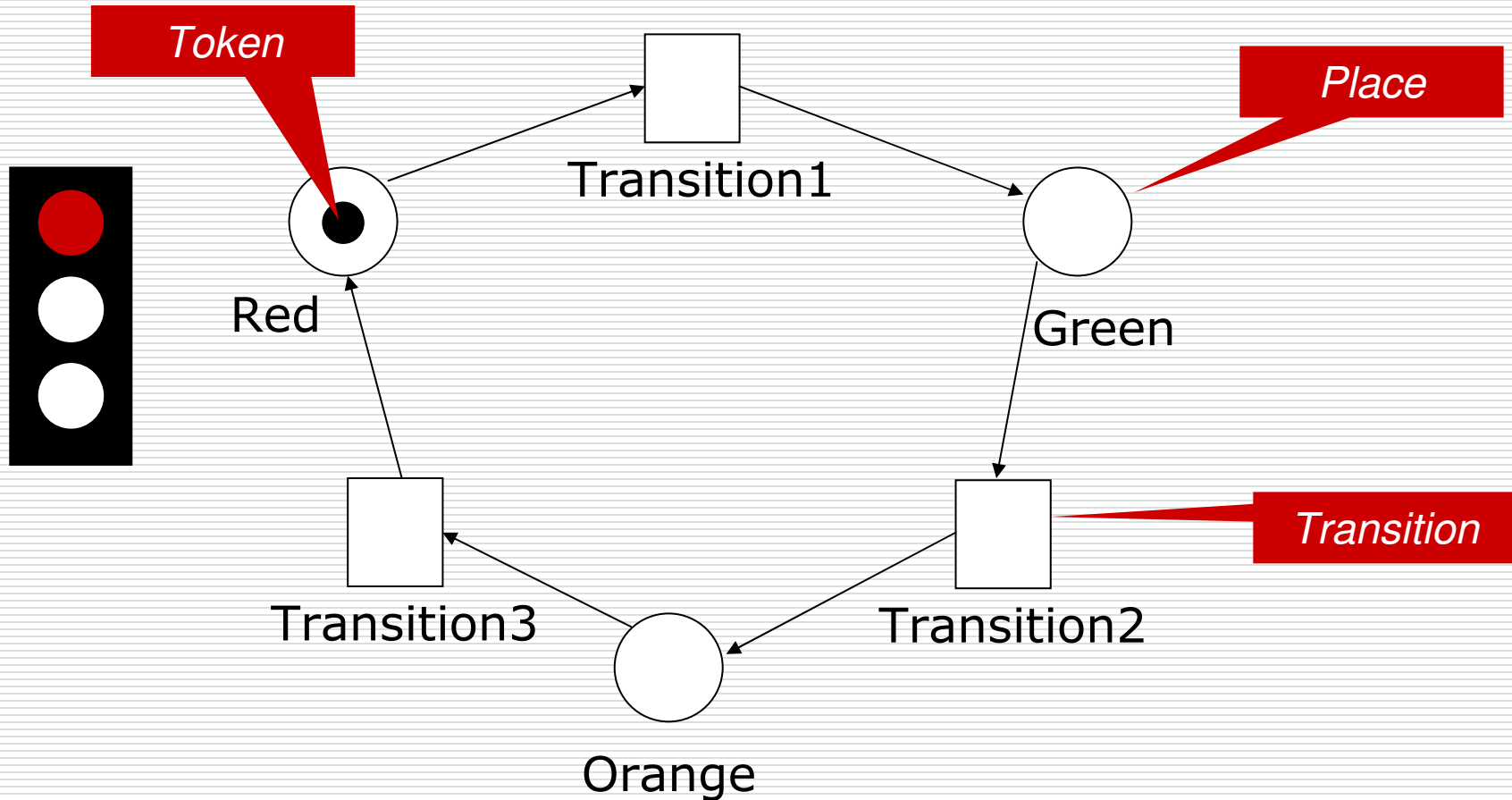
Modeling of System Behavior

- How do systems behave?
- Modeling system **behavior** is so **much harder** than modeling system structure
- Use cases ⇒
 - scenarios/processes → interaction diagrams
 - workflows → activity diagrams

Specification of Behavior

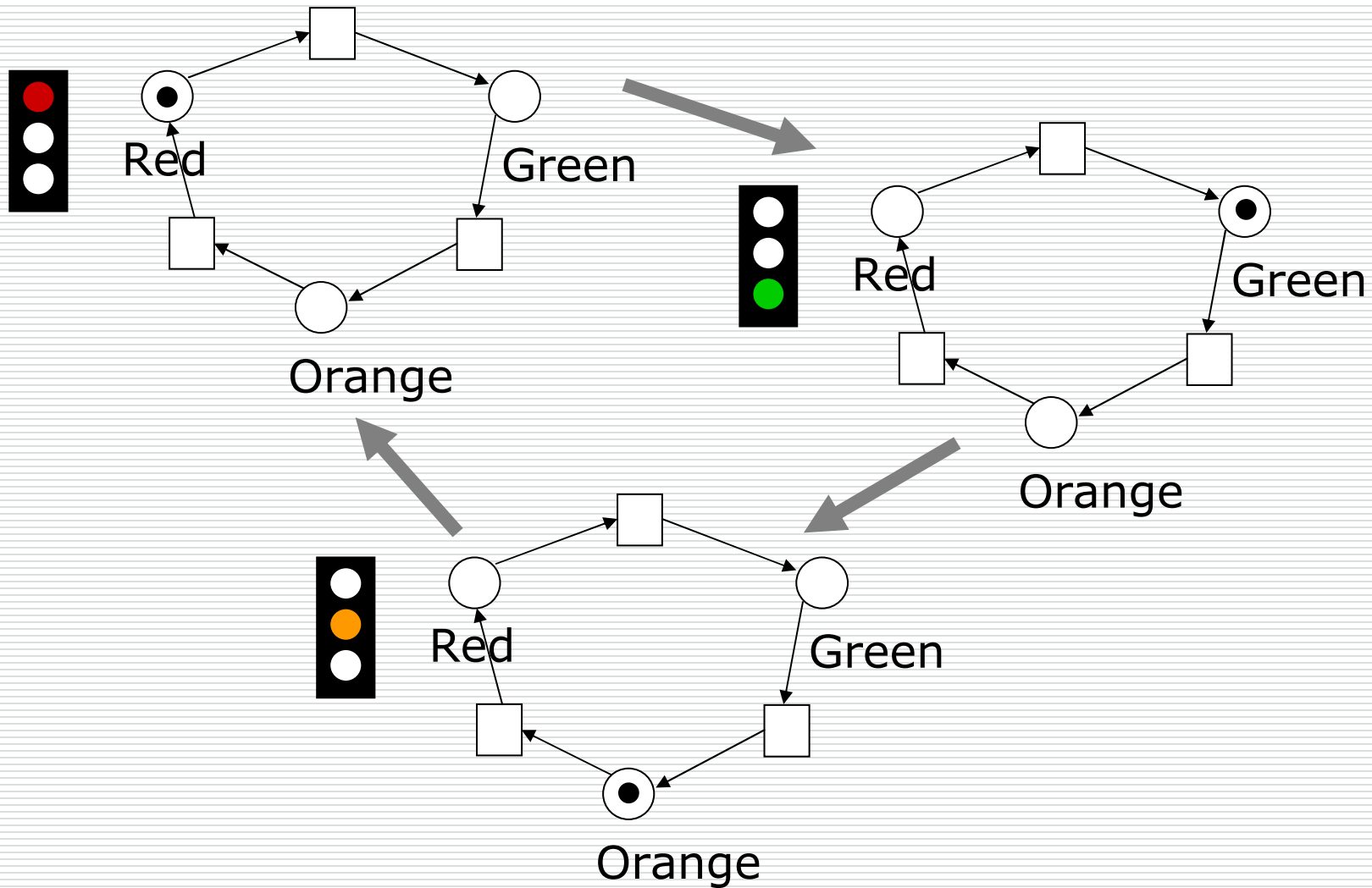
- Computer Science developed several models for behavior specification
 - logic-oriented models: predicate transformers on pre- and post-conditions
 - graph-oriented models:
 - Petri nets
 - state machines
 - activity diagrams
- Goals
 - Specification of state changes of an object (or an interaction) triggered by an external event or a received signal.
 - Definition of protocols, i. e., legal sequences of operations of a class or an interface.

Petri Net Example: Traffic Light



example from [1]

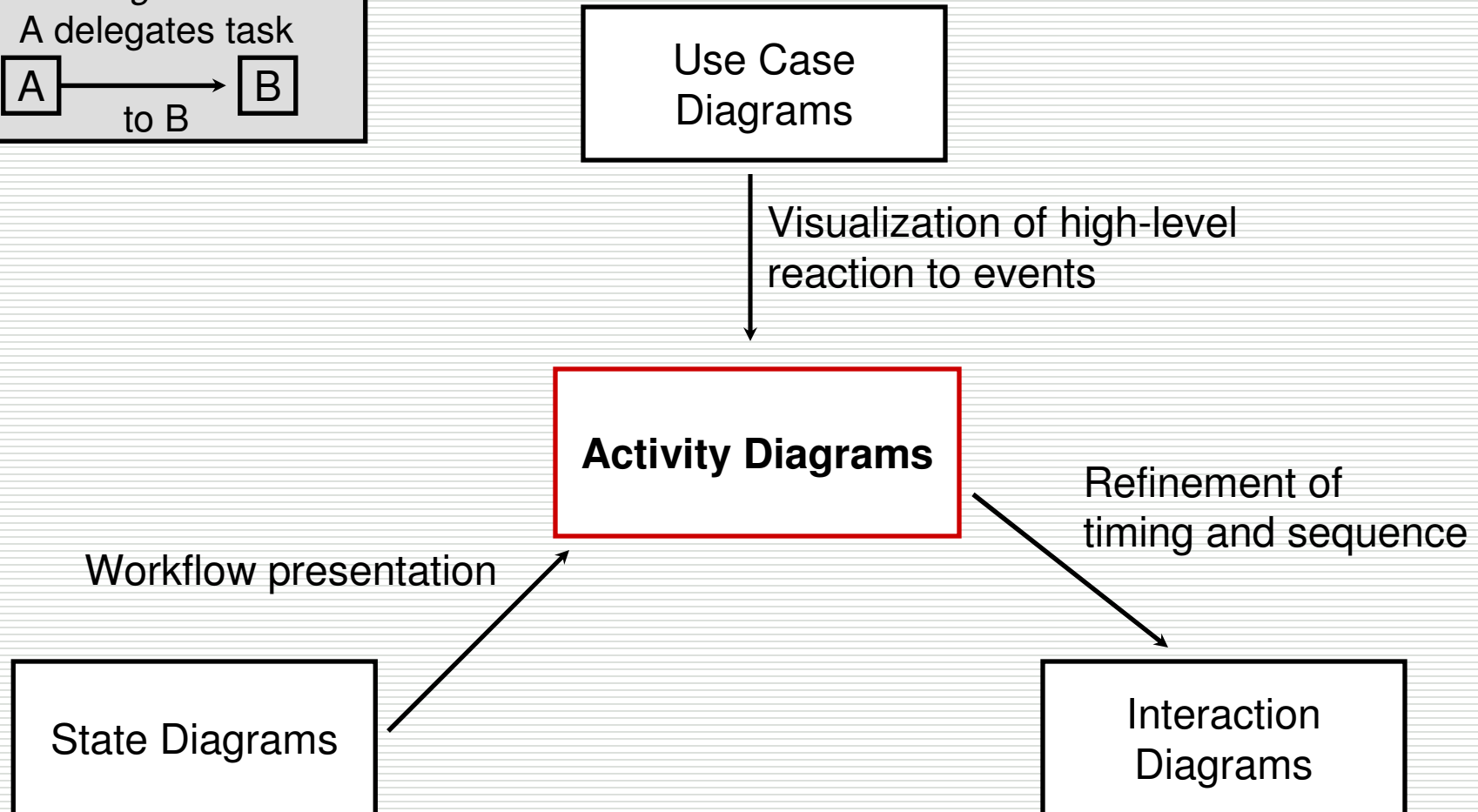
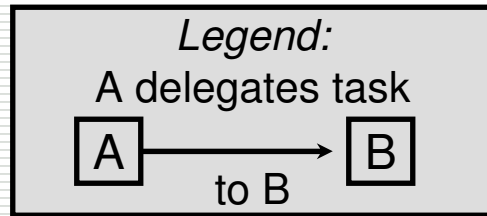
Petri Net Example in Action



Activity Diagrams

- Elements
 - States
 - Actions
 - Transitions
 - Branches, Merge
 - Concurrency, Synchronization
 - Swimlanes, Object Flow
 - Signals

Role of Activity Diagrams in UML



Activity Diagrams

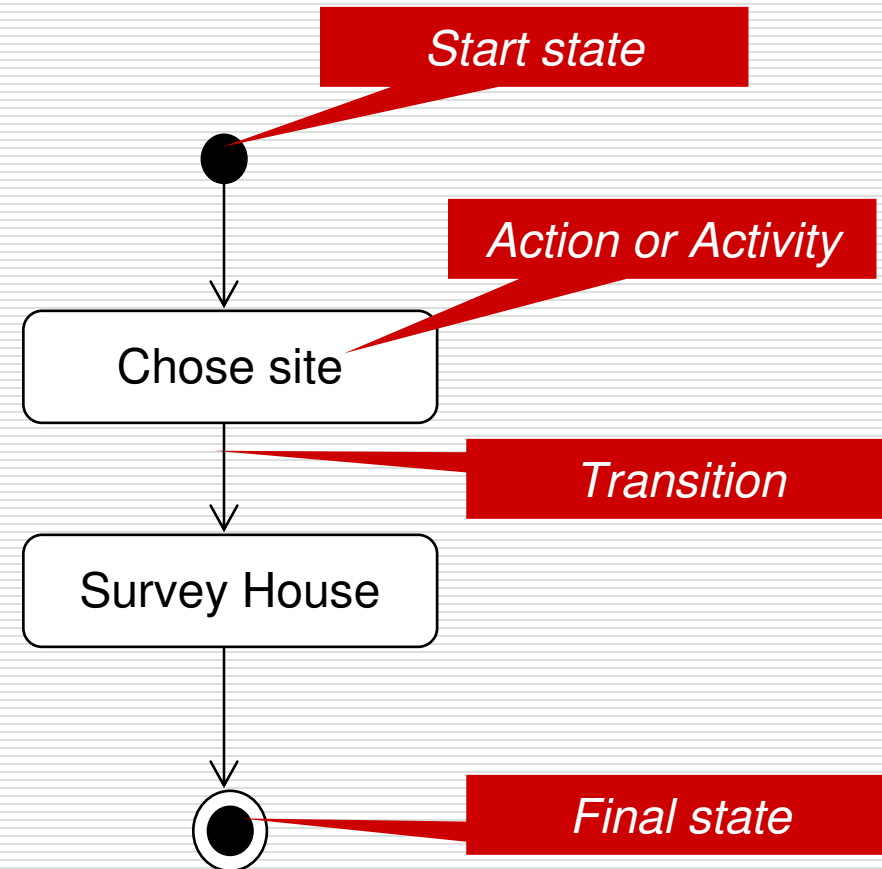
- The states are *action states* or *activity states* and the transitions are fired (triggered) by the termination of activities.
- In activity diagrams the concept of *state* does not refer to a static situation, but to named clusters of *acts*.

Example: Building a house



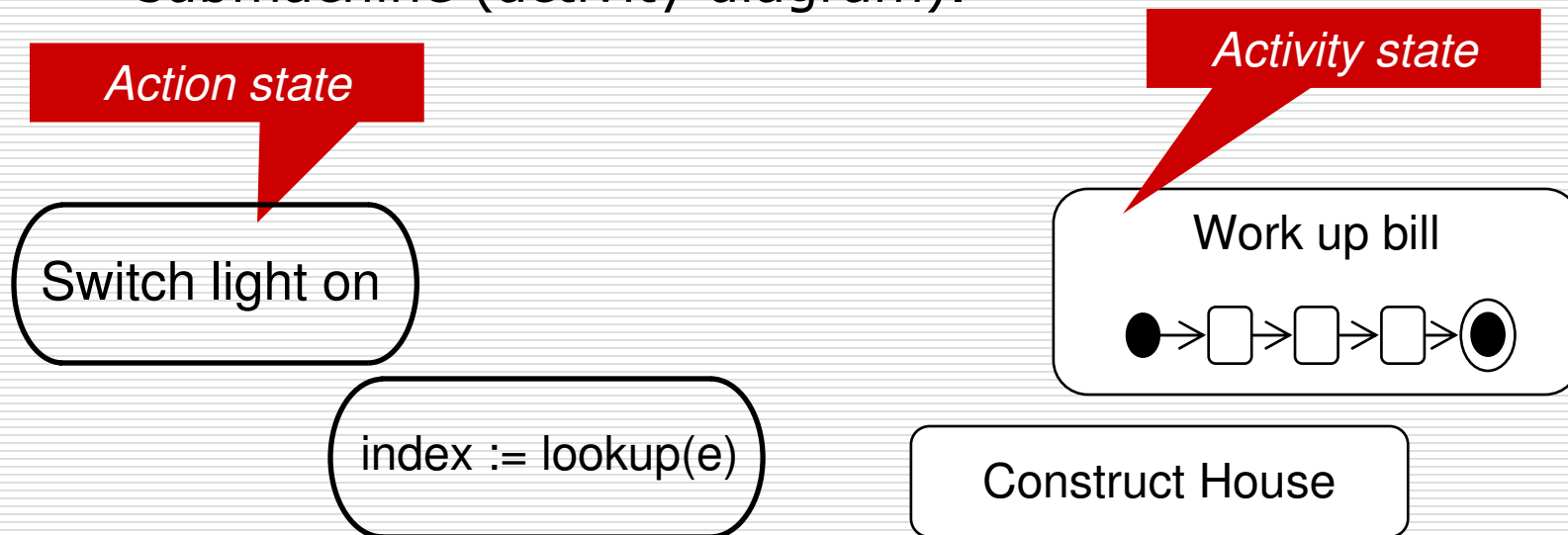
Transitions

- Transitions describe how to get from one state into another.
- A transition is executed when the previous action or activity is terminated.



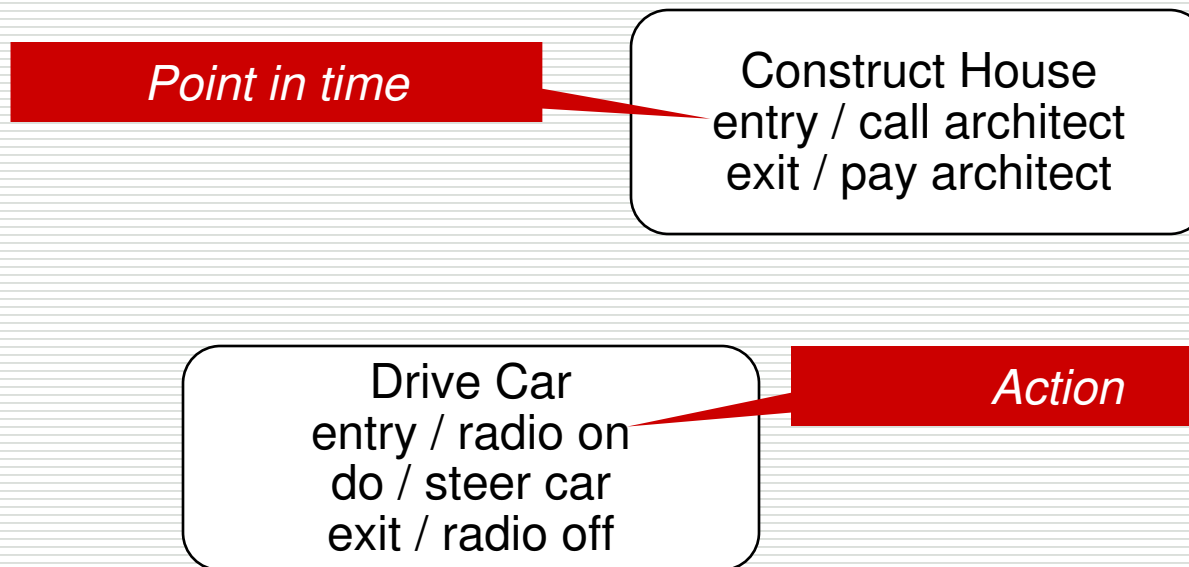
Action States and Activity States

- ❑ An **action state** in an activity diagram describes an atomic change of a system's state without temporary fine structure, e. g., an operation call or the calculation of a value. Action states cannot be decomposed.
- ❑ An **activity state** describes an enduring activity which can be interrupted and typically is described by a submachine (activity diagram).



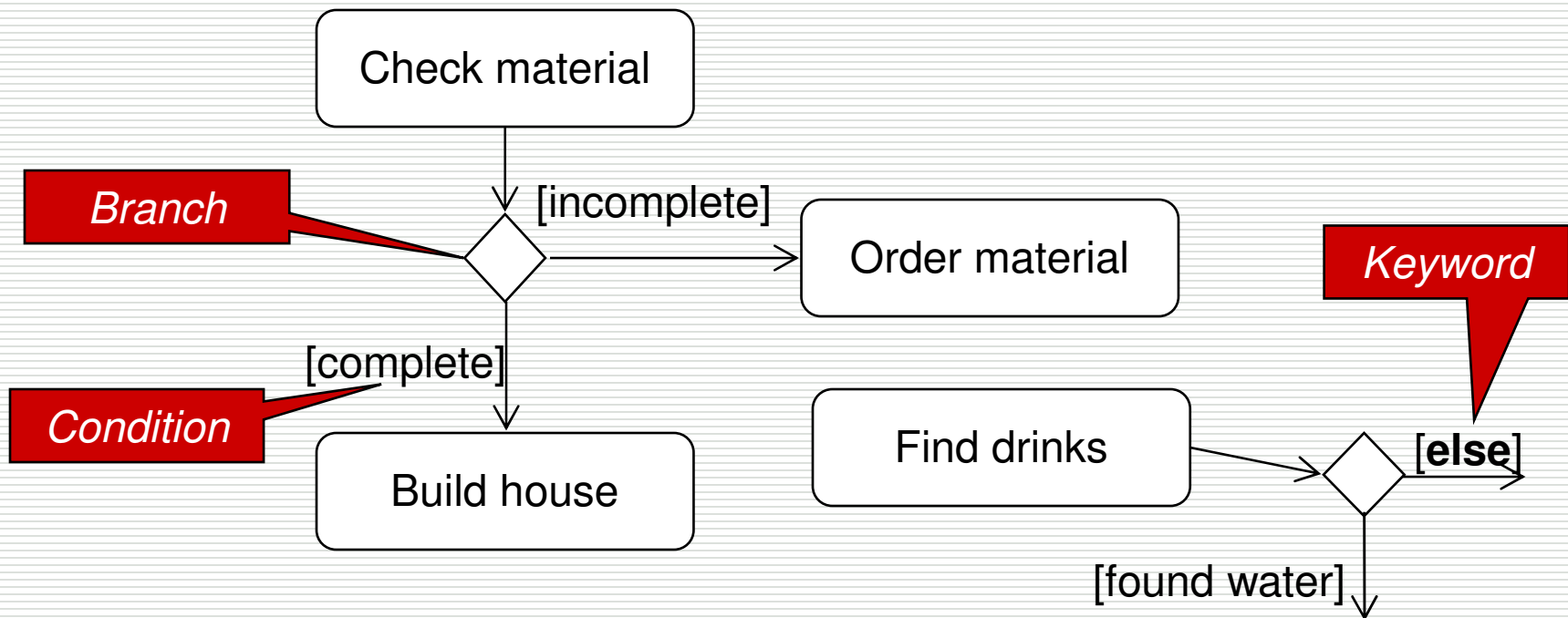
Activity States: Actions

- In activity states actions can be executed at the beginning (entry point) or end (exit point) of an activity.



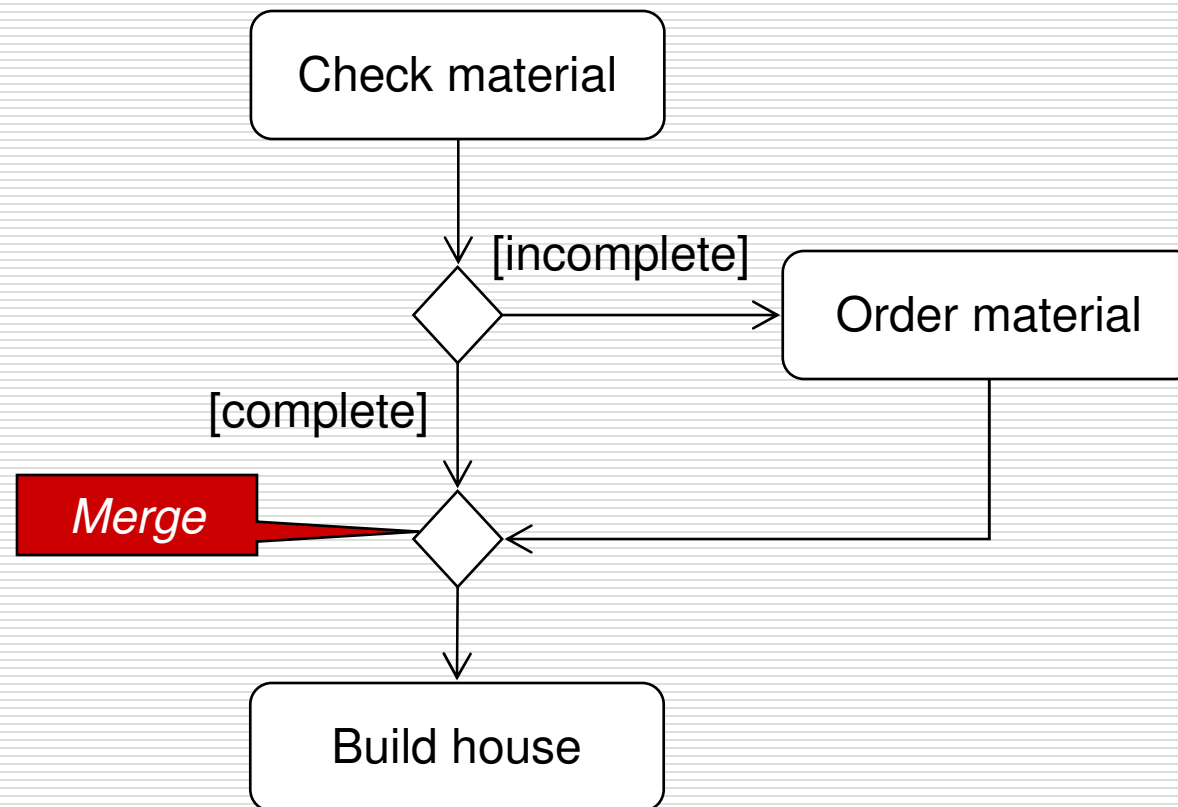
Branches

- A **branch** appears, if a decision is to be made and depending on a condition several following actions/activities are to be initiated.



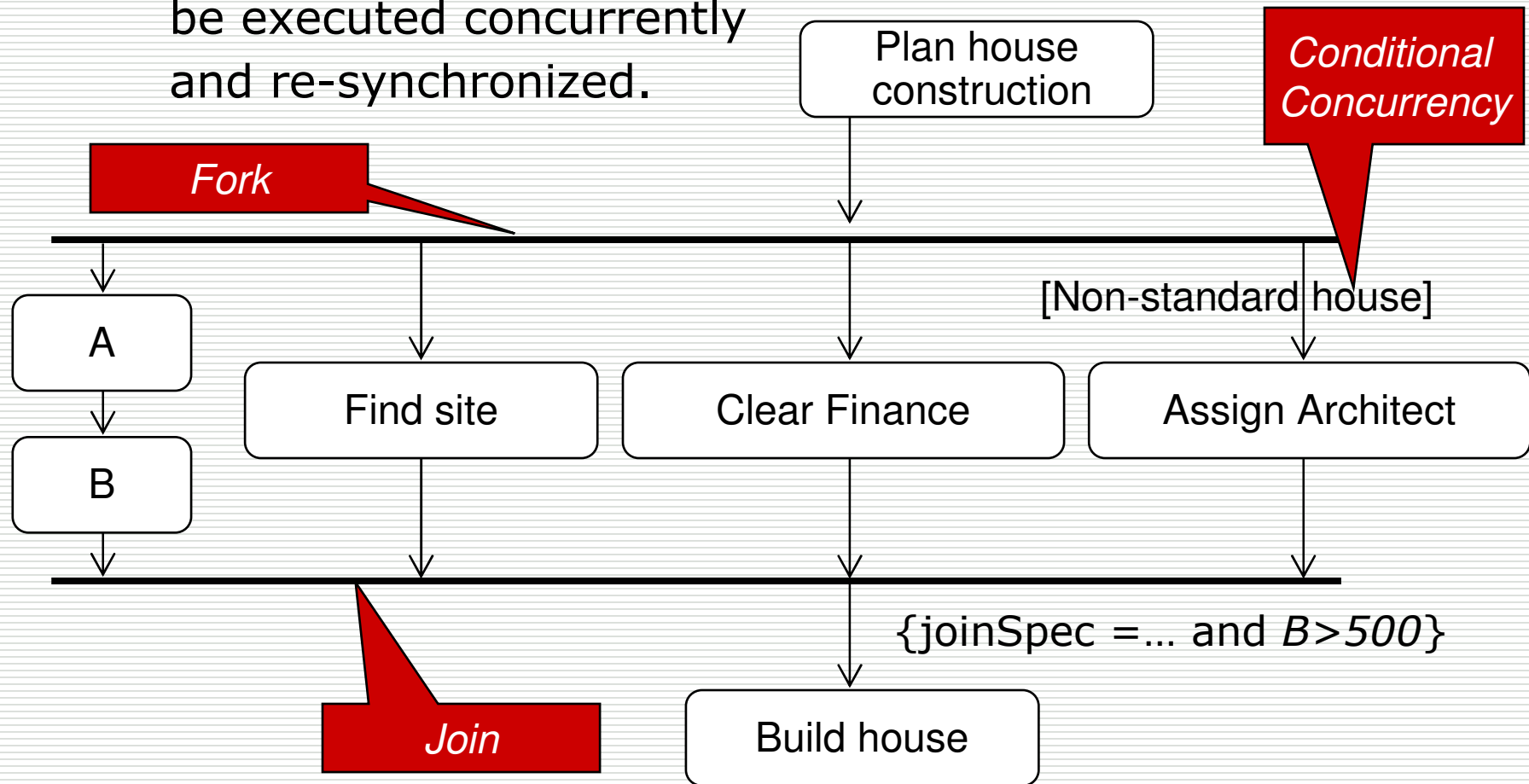
Merge

- A **merge** merges optional branches in activity diagrams.



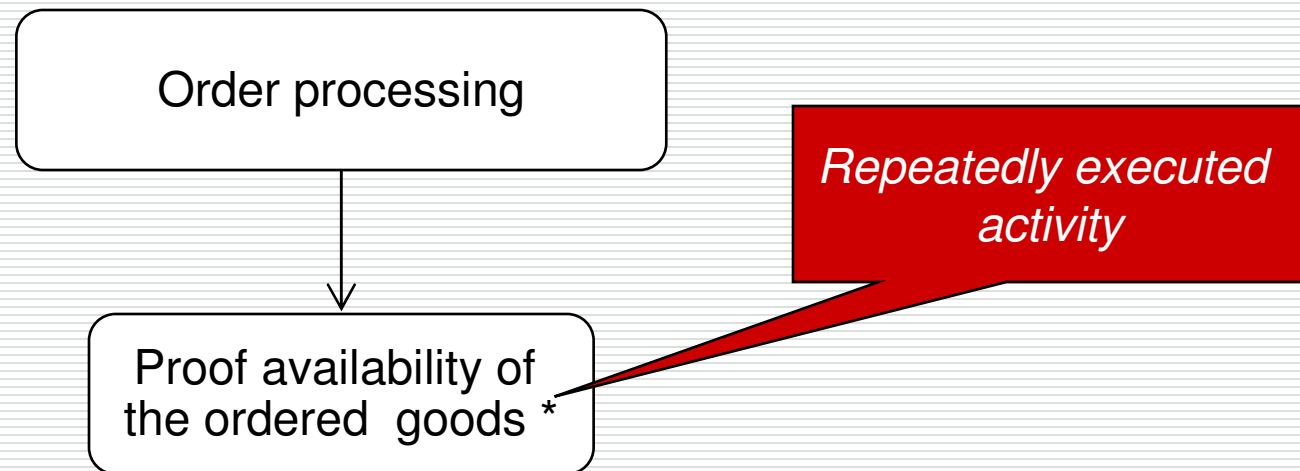
Concurrency

- Actions and Activities can be executed concurrently and re-synchronized.



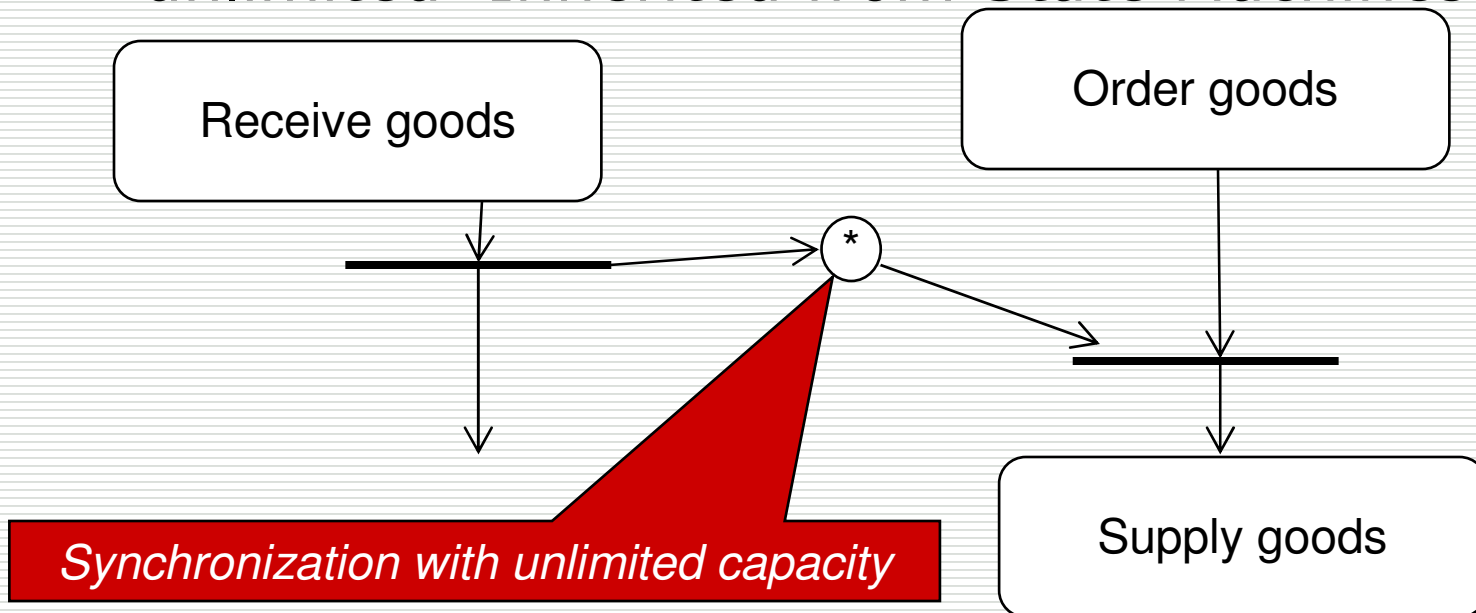
Dynamic Concurrency

- An activity is executed concurrently multiple times where the number of concurrent activities depends on a list of arguments.



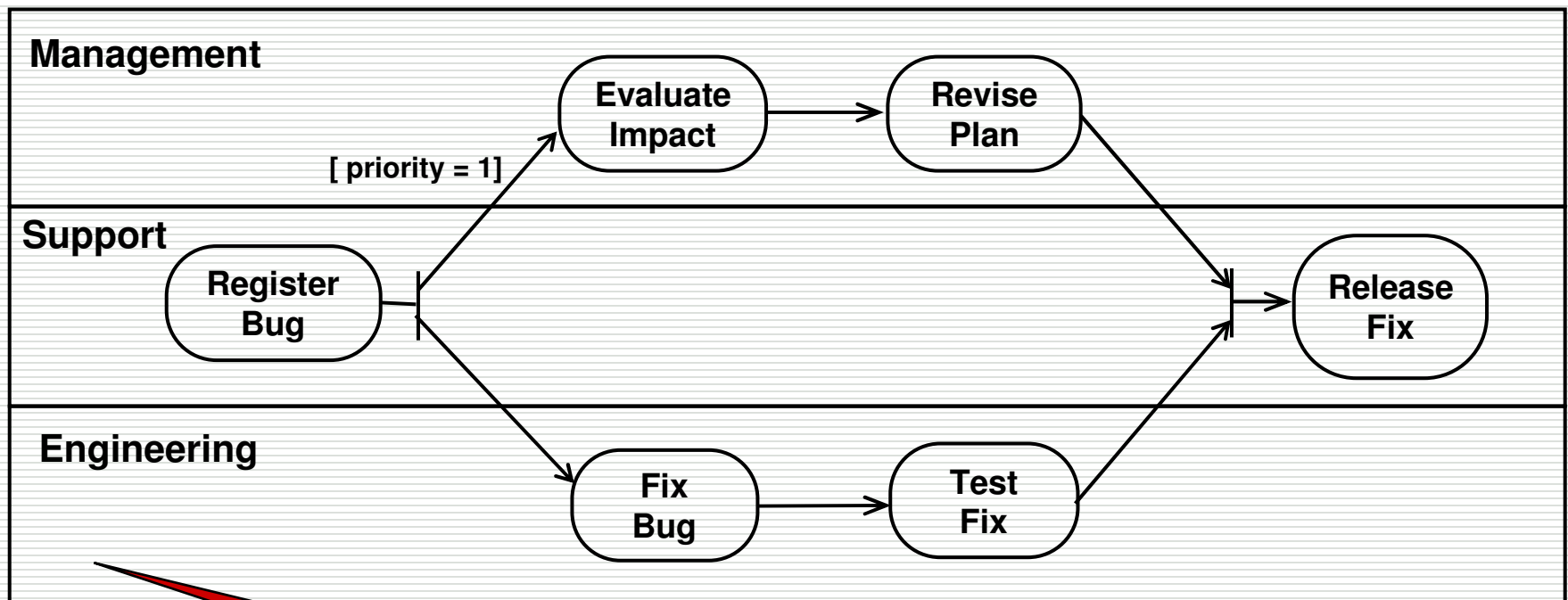
Synchronization of Concurrent Activities

- Concurrent activities can be synchronized by a synchronization state. The state contains implicit a counter representing the number of waiting activities. The counter can be limited or unlimited. Inherited from State Machines



Swimlanes

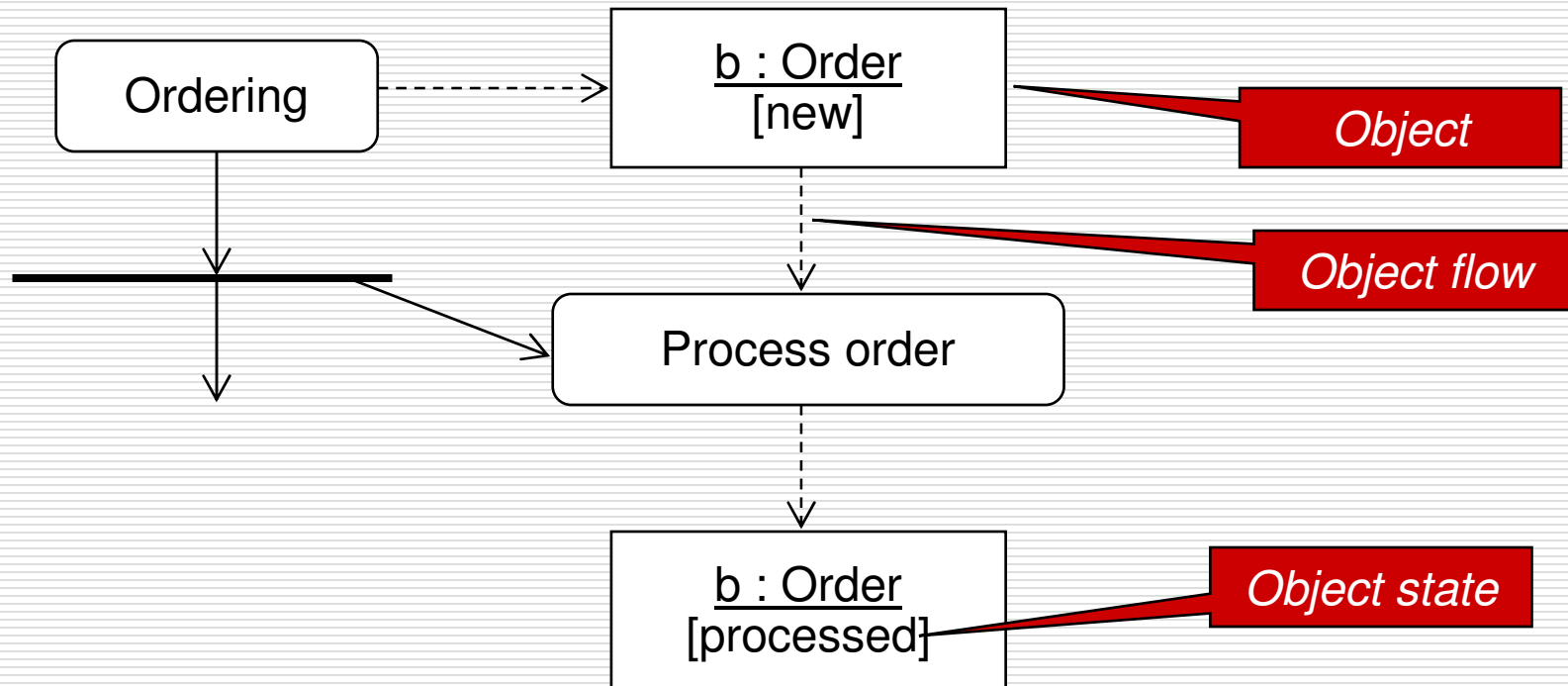
- ❑ **Swimlanes** structure actions and activities and show them clustered by execution unit (object or a class, mostly concurrent to other actions/activities).



Swimlane

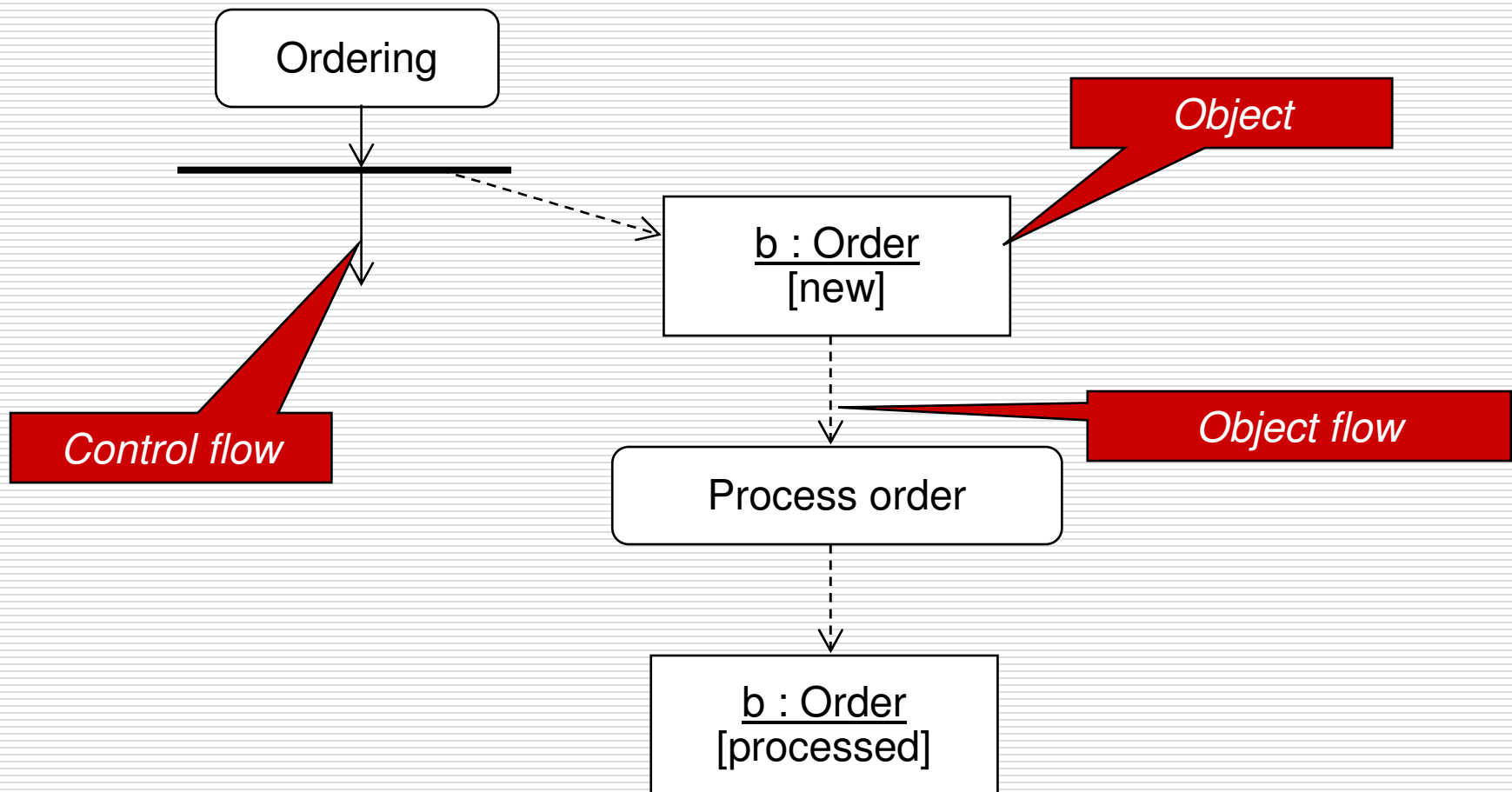
Object Flow

- An **object flow** shows objects with their states being generated or used by actions or activities in activity diagrams.



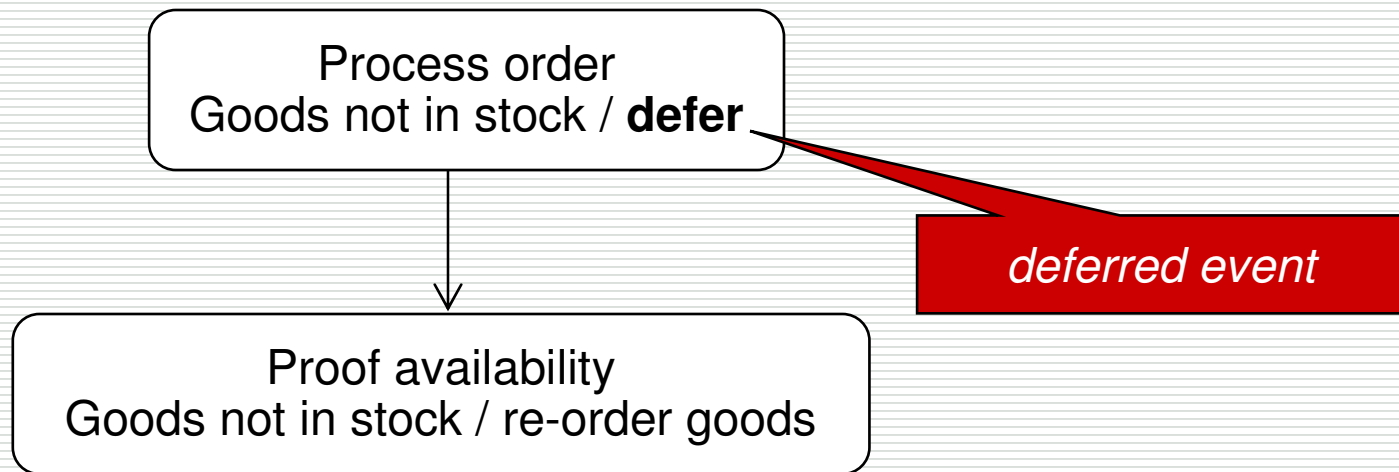
Object and Control Flow

- Object flow is combined with control flow.



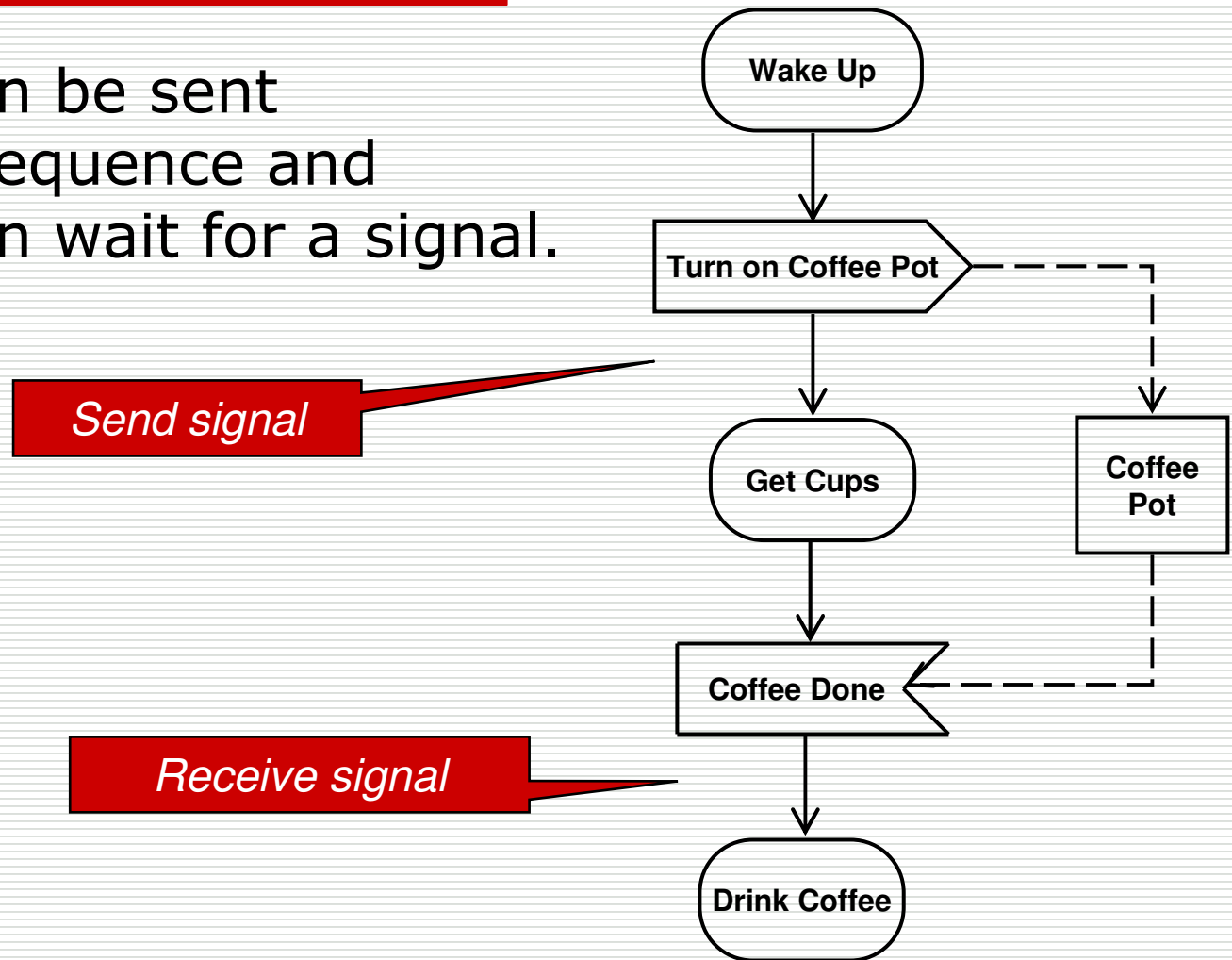
Deferred Events

- Events which cannot be reacted upon in the current action, can be **deferred** (queued).
- Deferred events can later lead to a reaction. Other events expire if they are not consumed immediately.



Sending and Receiving Signals

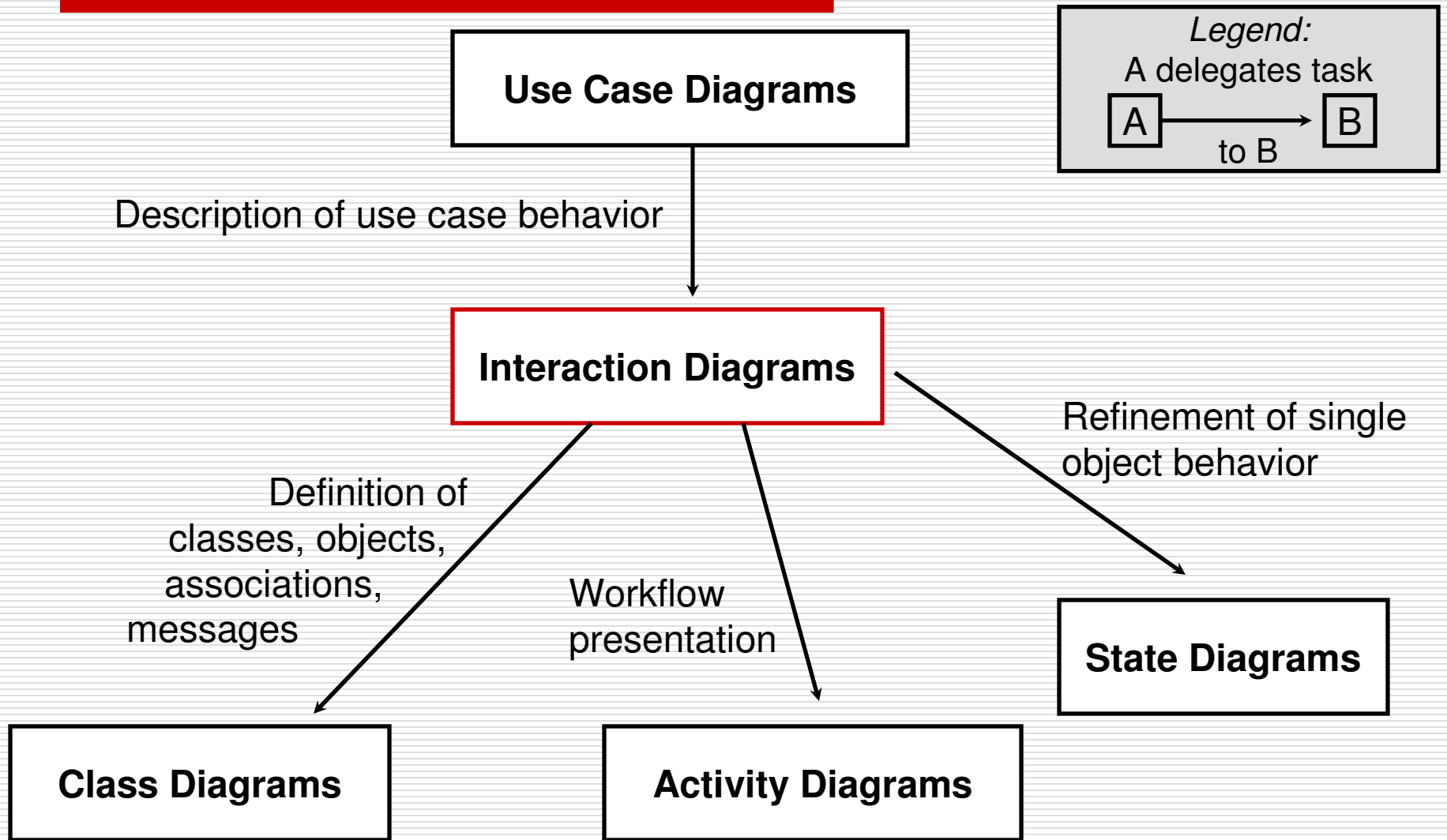
- Signals can be sent during a sequence and objects can wait for a signal.



On Interaction Modeling

- Need to model dynamic aspects of interaction between objects:
 - specifies details of system functionality;
 - functionality is realized as interaction (*message passing*) between objects;
 - uses the elements specified in structure diagrams (e.g. instances of classes).
- Individual interaction diagrams usually show a much smaller part of the system than static diagrams.

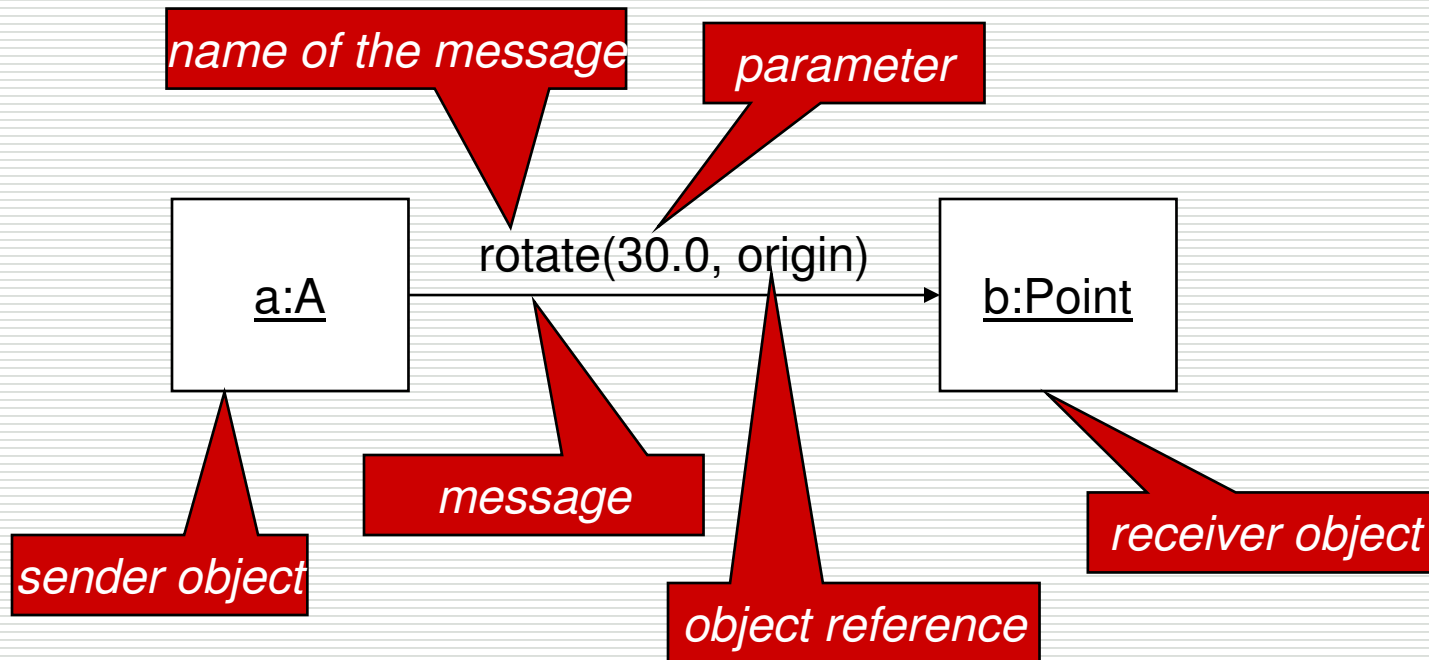
Role of Interaction Diagrams in UML



Interaction Diagrams in UML

- **Interaction diagrams** are a series of diagrams describing the behavior of an object-oriented system with object interaction.
 - A **sequence diagram** stresses the temporal sequence of object interactions.
 - A **collaboration diagram** stresses the objects taking part in the object interaction.
- For the description the following terms are used:
 - **message** (asynchronous **signal** and synchronous **method call**)
 - **activation** of an object
 - **active object**

Conceptual Modeling of Interactions

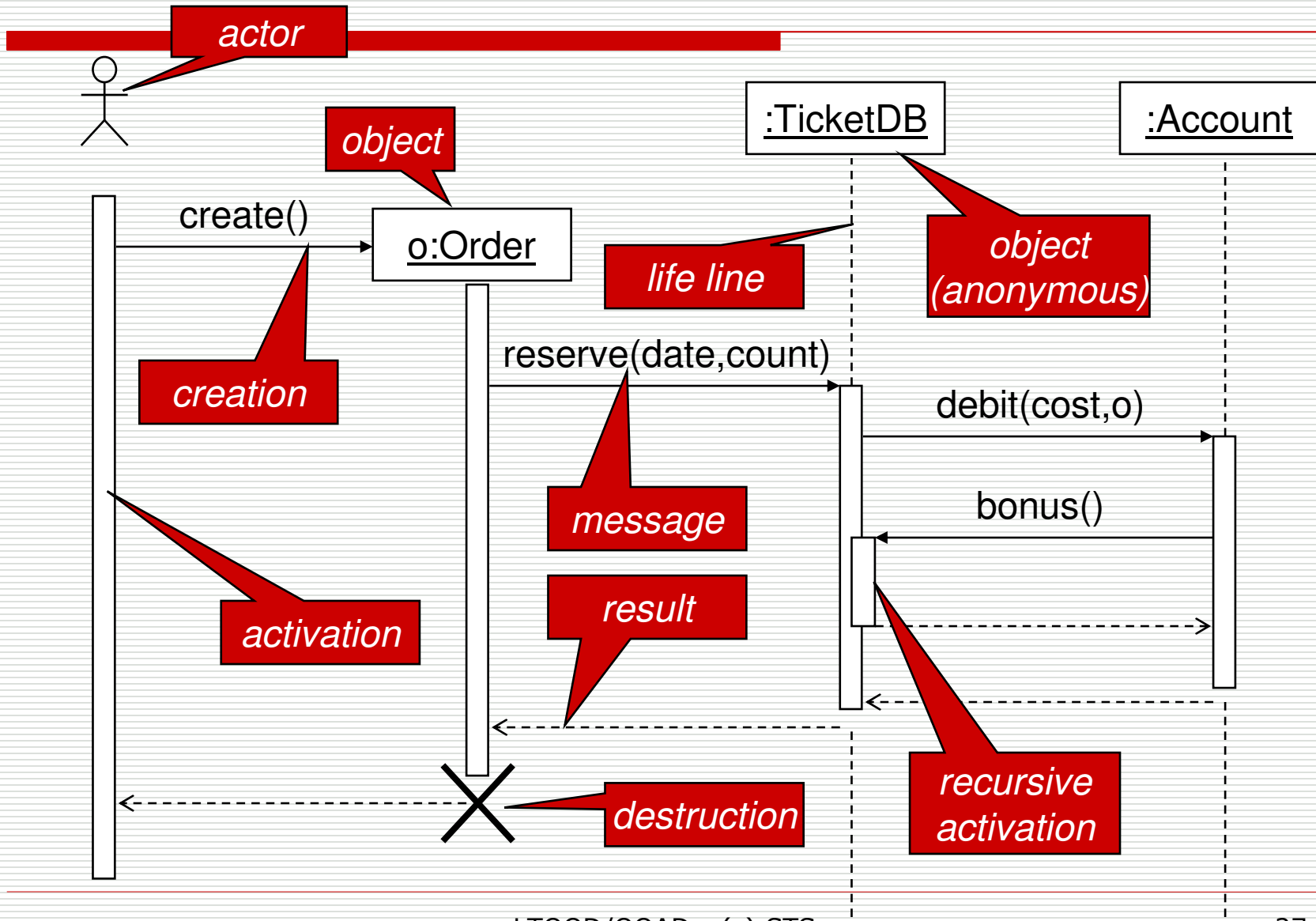


Synchronous (Procedural) Interaction

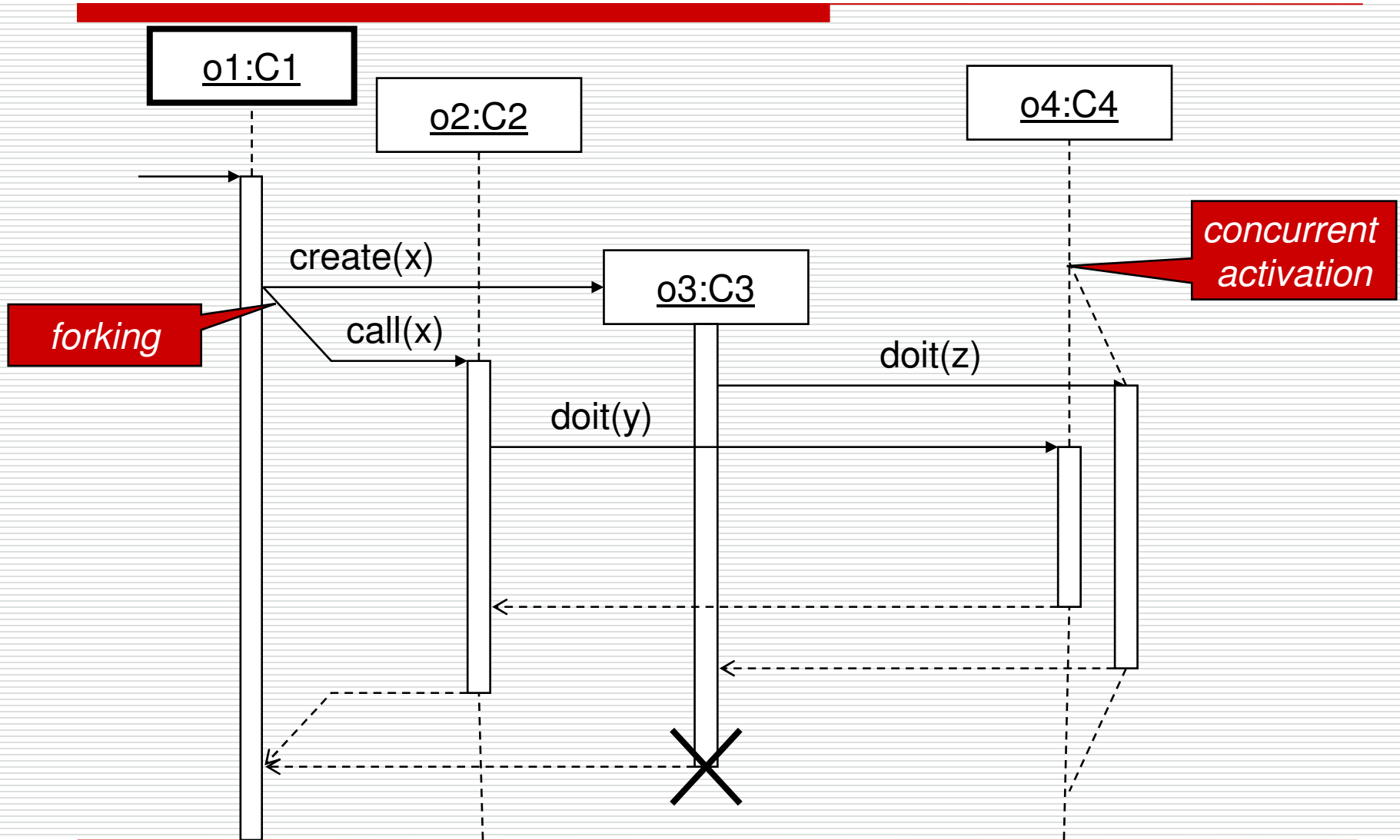
- Synchronous interaction
 - **message** sent between objects
 - sending object must be **activated** and activates the receiving object. The sender object blocks (i.e., waits) until control returns.

- Asynchronous interaction
 - objects send **signals**
 - **sender stays active** as well but does not block (i.e., continues)
 - **receiver** consumes signal and is activated **in parallel**

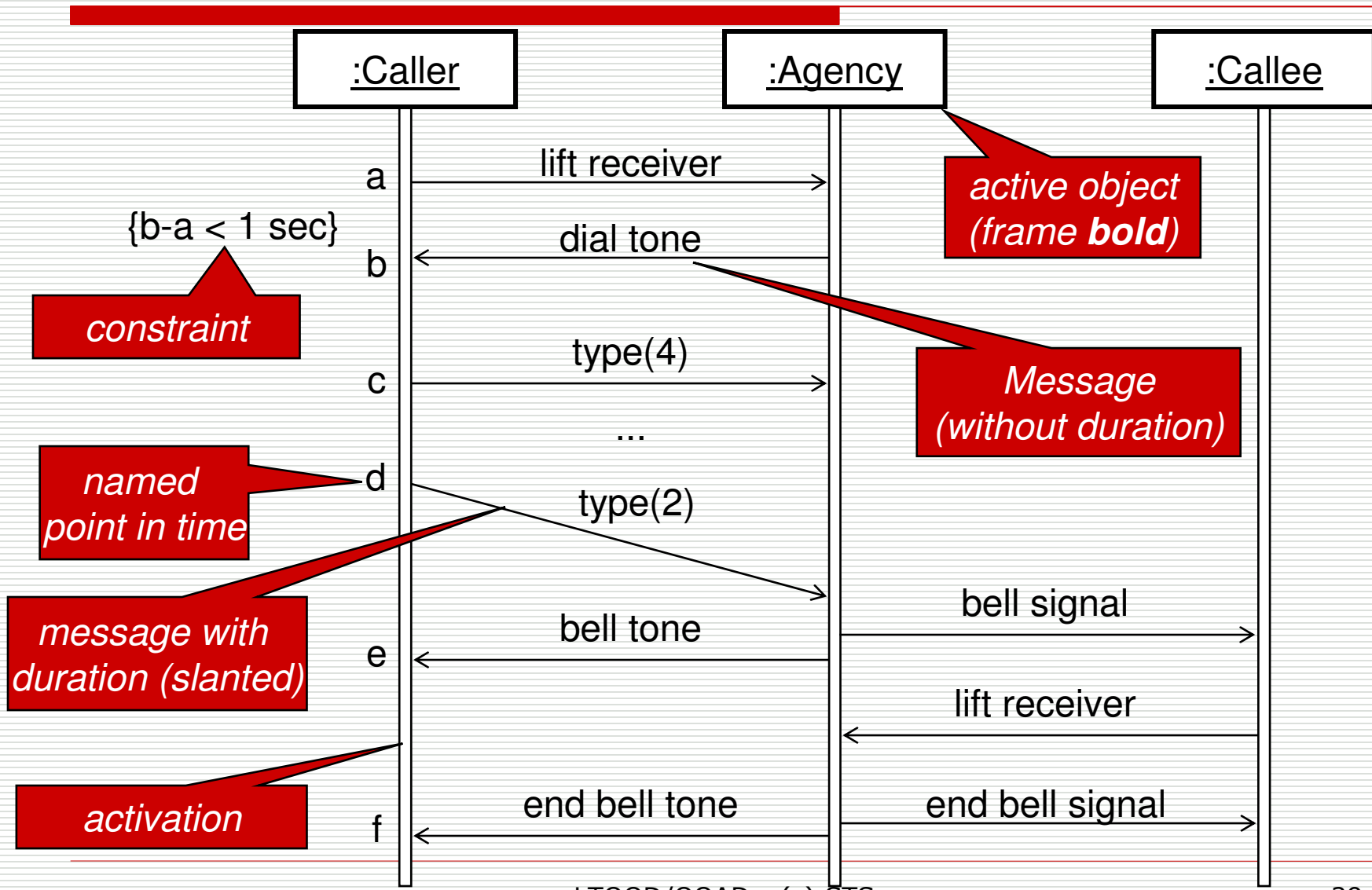
Sequence Diagram: Synchronous Interaction



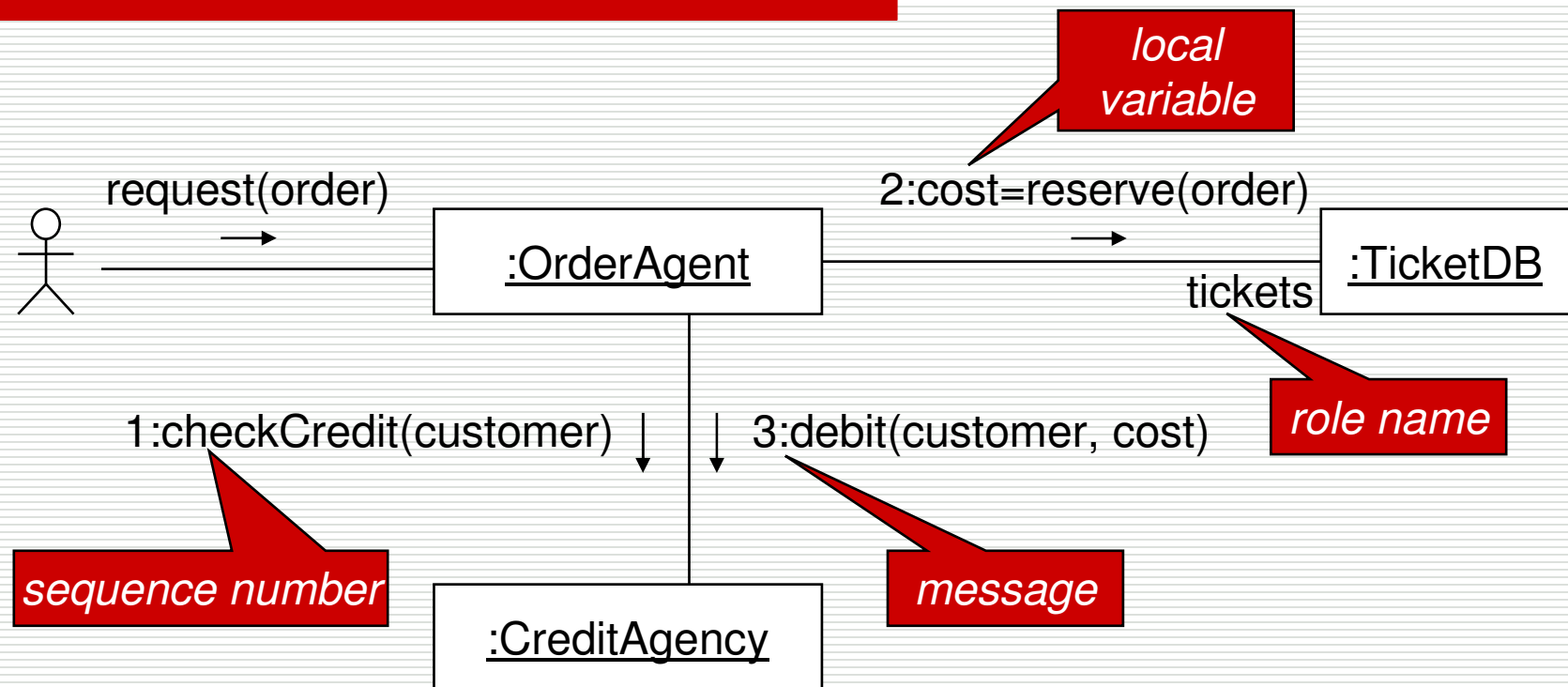
Sequence Diagram: Details



Sequence Diagram: Asynchronous Interaction



Collaboration Diagram: Example



Sequence numbers denote chronology of events.

- Simple scheme: just number messages
- Decimal scheme: nested numbering of messages where call-stack depth is visible

Message Labels

Message	Semantics
2: display(x, y)	Simple message with sequence number
1.2.1: p=find(name)	nested call with decimal sequence number
[x<4] 2: invert(rect)	conditional message (x less than 4) with simple sequence number
3.1*: update	iteration
3.2*[i=1..8]: draw(v[i])	iteration

State Diagrams

- Elements:
 - States
 - Transitions
 - Events
 - Composition
 - Concurrency
 - Branch
 - History

Intra-Class Behaviour: State Machines

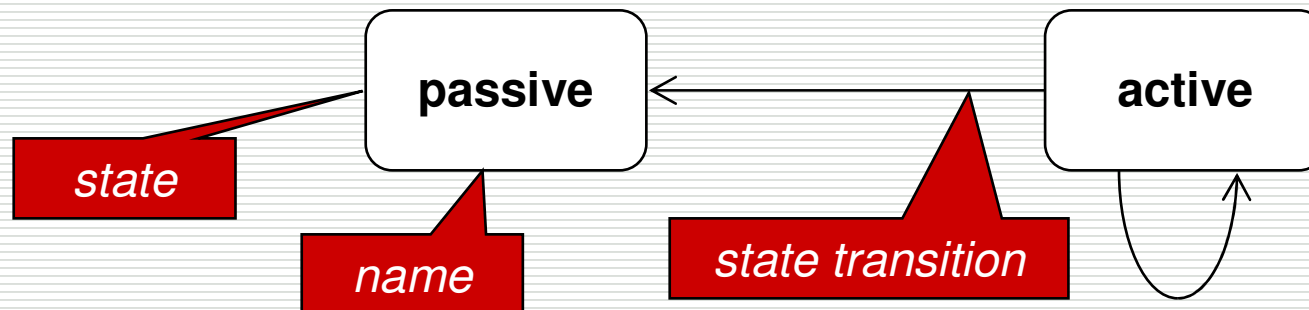
- Most objects keep an internal state.
 - It usually cannot change in arbitrary ways.
 - There needs to be a precise model that makes explicit which state changes are legal.
 - Well-known concept for this: state machines.
- A state machine ...
 - specifies **states** and **state transitions** of an object
 - specifies possible **actions** in each states
 - is attached to exactly one class
 - is inherited by subclasses (but a concept of the refinement is not defined precisely)
- A state machine thus defines a **protocol** (i.e., legal sequences of operations) for its class.

Elements of State Machines

- State:
 - in a specific *state* only certain actions are possible
 - a state is a phase in the life cycle of an object
- Transition:
 - concrete actions are associated with *state transitions*
- Event:
 - transitions and their actions are triggered by *events*;
- Guards:
 - *guards* may prevent transitions from taking place
 - guards are modeled by boolean conditions.

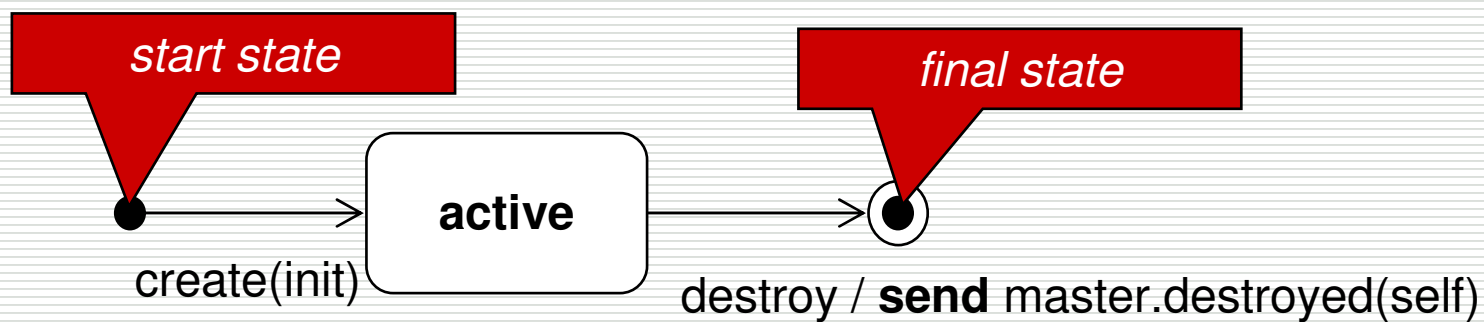
States and State Diagrams

- A **state diagram** is a graph consisting of
 - **states**
 - simple states
 - composite states (states refined by nested state diagrams)
 - **state transitions** connecting the states.



Start and Final States

- Need well defined beginning and end of life-cycle.
- Start state: State transition is executed immediately during the creation of the object.
- Only possible event: create(parameter)



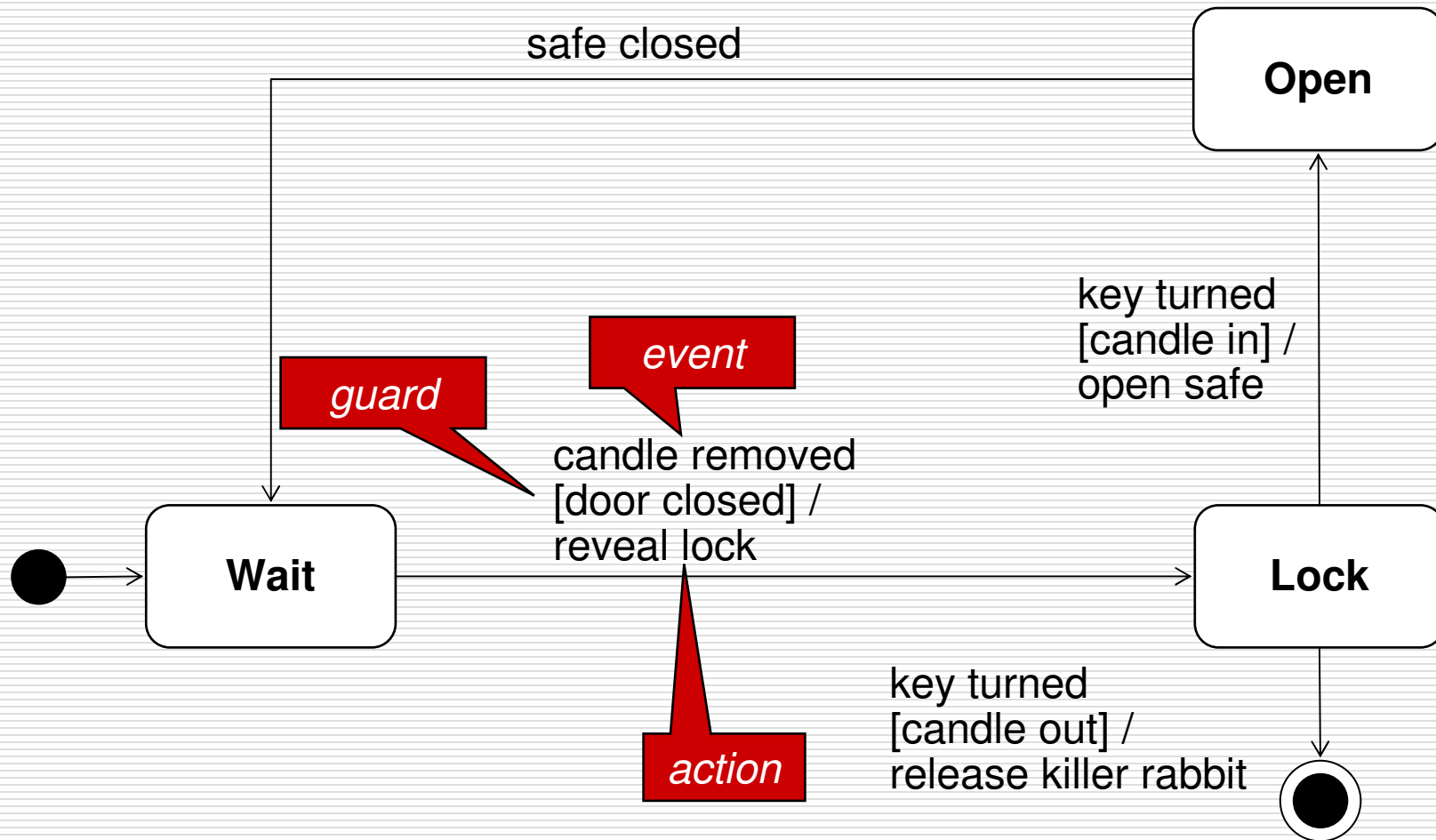
Transitions

- ❑ Transitions connect two states
- ❑ Transition can include a **triggering event**, a **guard** and **actions** to be executed.
- ❑ Transitions without event and guard are executed **immediately** (possibly after passing through all sub states).

... I decided to use a controller for a secret panel in a Gothic castle. In this castle, I want to keep my valuables in a safe that's hard to find. So to reveal the lock to the safe, I have to remove a strategic candle from its holder, but this will reveal the lock only while the door is closed.

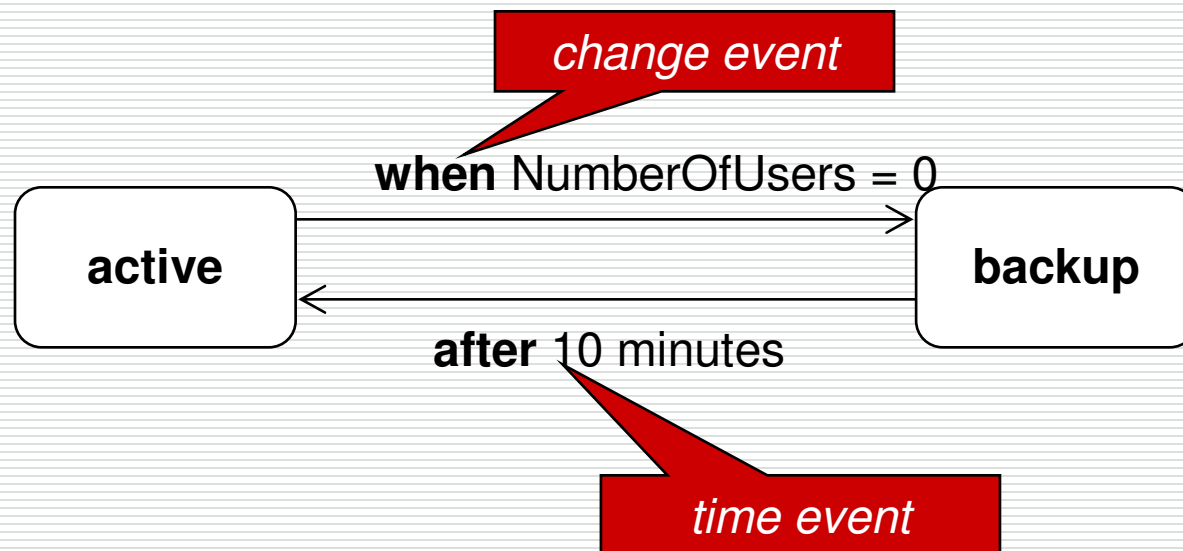
Once I can see the lock, I can insert my key to open the safe. For extra safety, I make sure that I can open the safe only if I replace the candle first. If a thief neglects this precaution, I'll unleash a nasty monster to devour him

Example



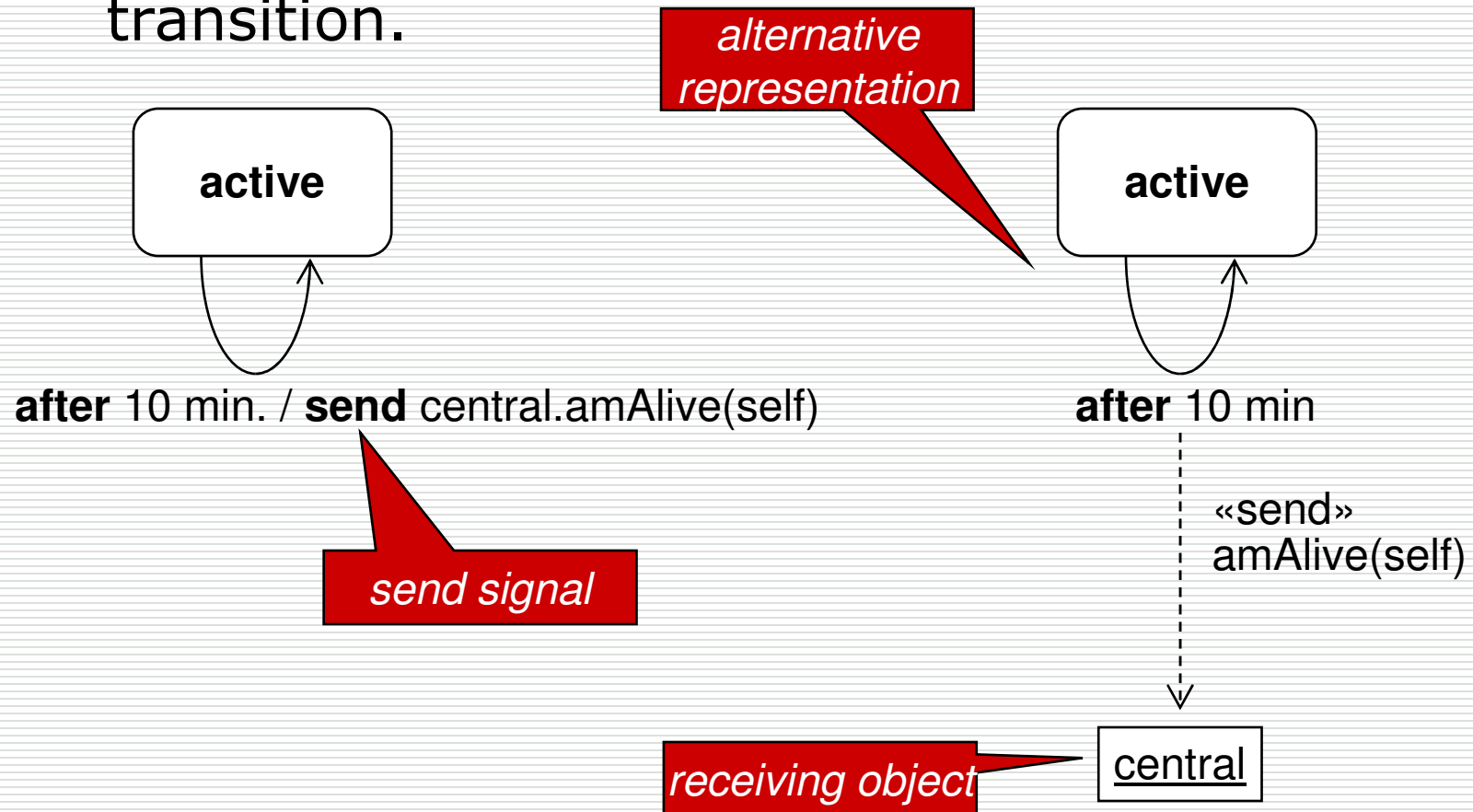
Time and Change Events

- A **change event** occurs if a specific constraint is fulfilled. The transition is made as soon as the constraint expression evaluates to true.
- A **time event** appears after the expiration of a time period.

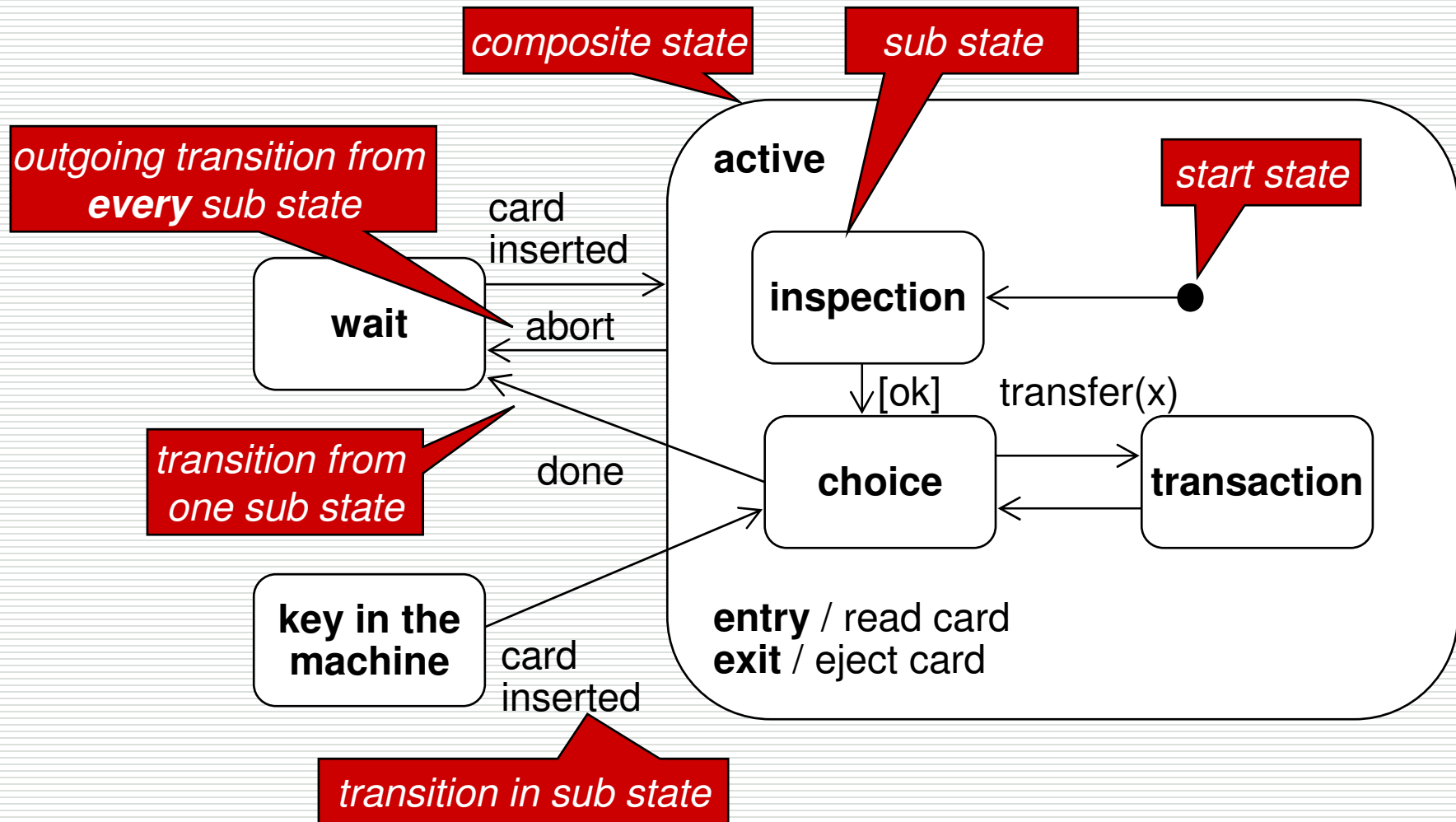


Sending Signals

- Signals can be sent to other objects during a transition.

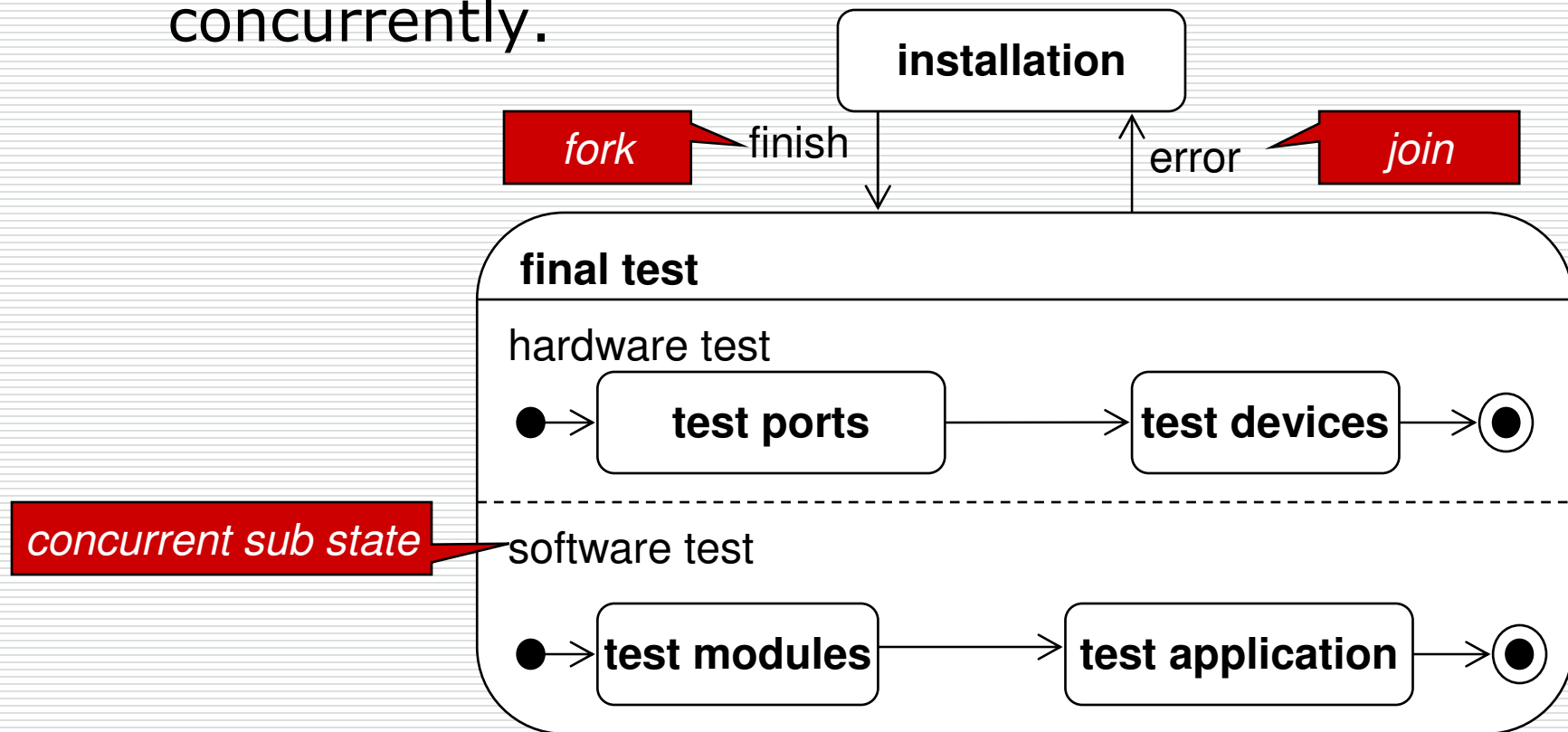


Composite States

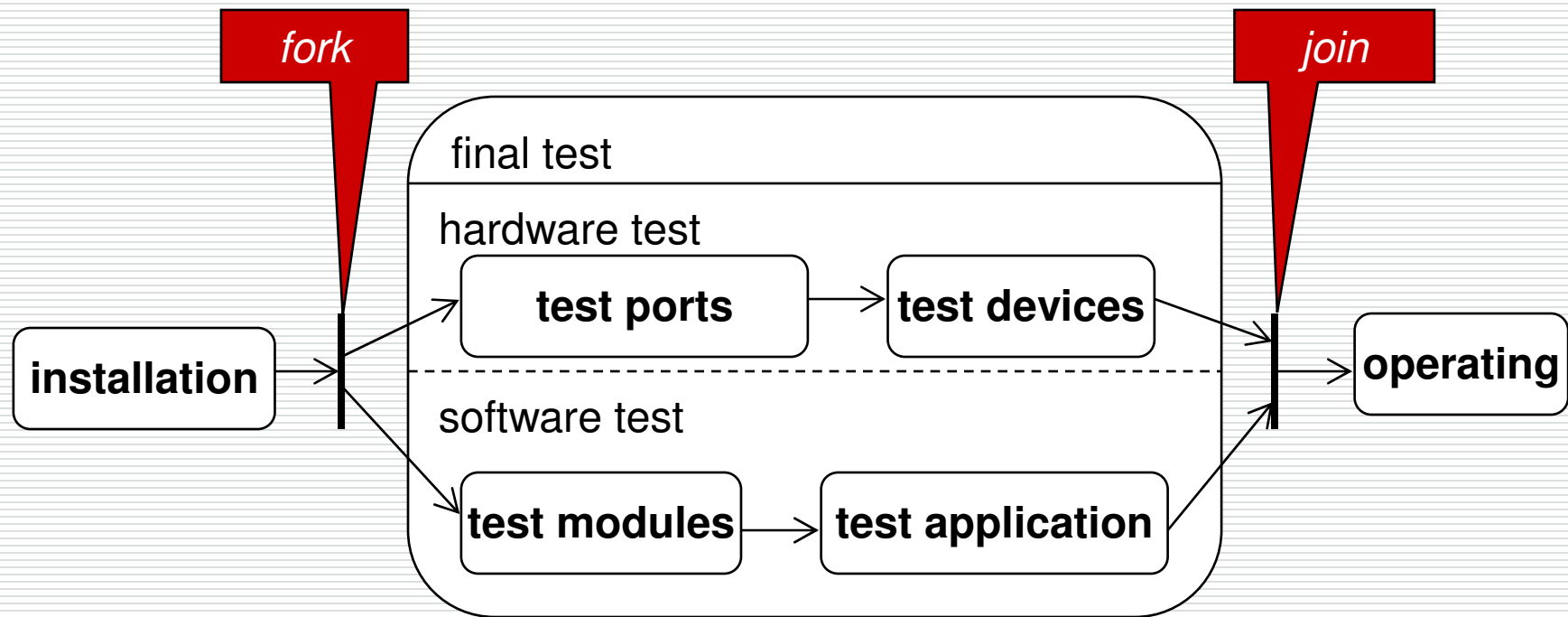


Concurrent Sub States

- In a state several sequences of sub states described by state machines can be performed concurrently.



Concurrent Sub States: Alternative

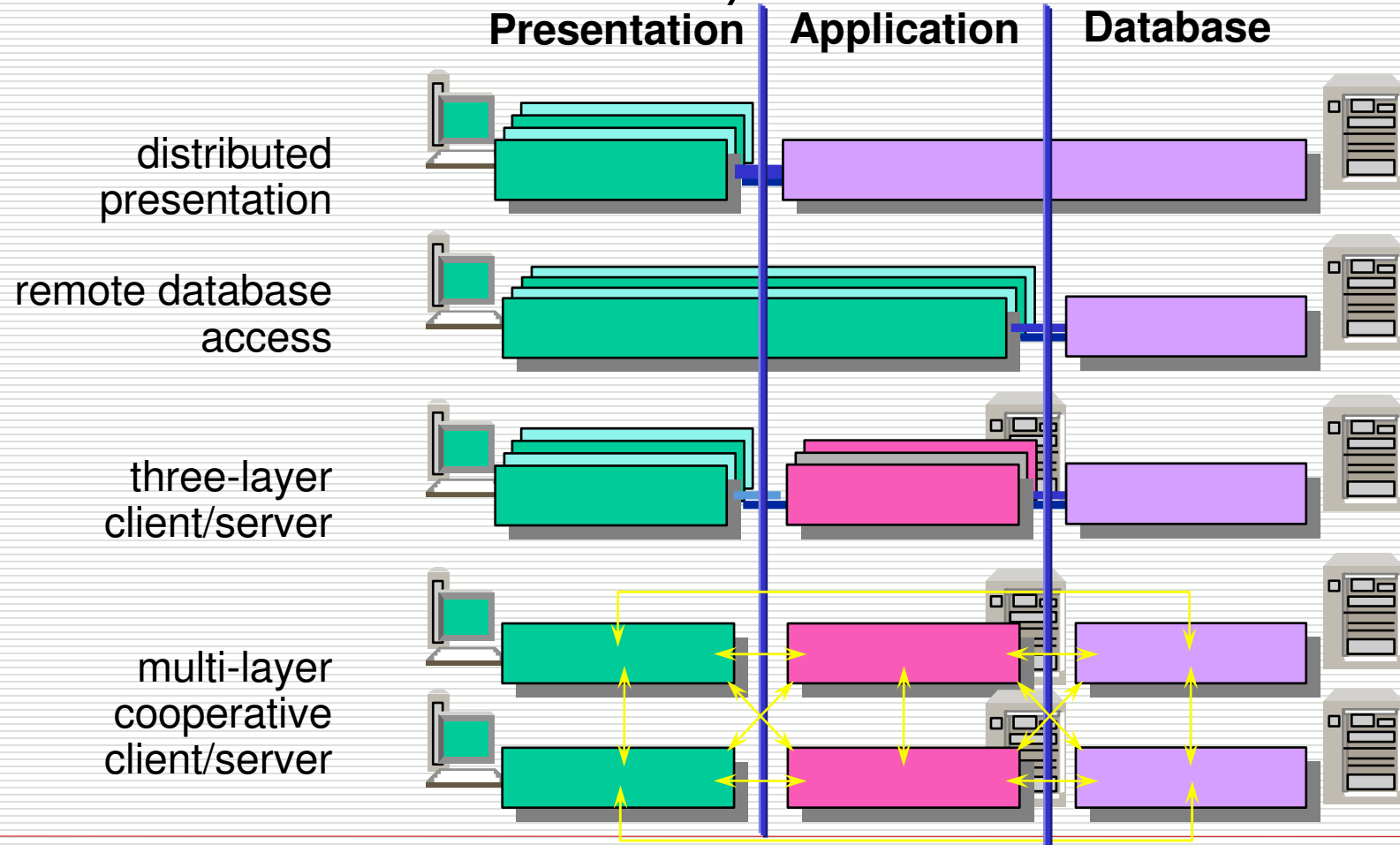


Architecture Design Models

- An **architecture model** (structure model) is a model of a data processing system describing the static structure of the components of a system.
- For large systems you should model this explicitly.
- Examples:
 - network topology (hardware)
 - function tree (software)
 - deployment diagram, package diagram (software)
 - module diagram (hard/software)
 - organization chart in a company model

Example: SAP R/3

- Flexible three-tier *client/server*-architecture

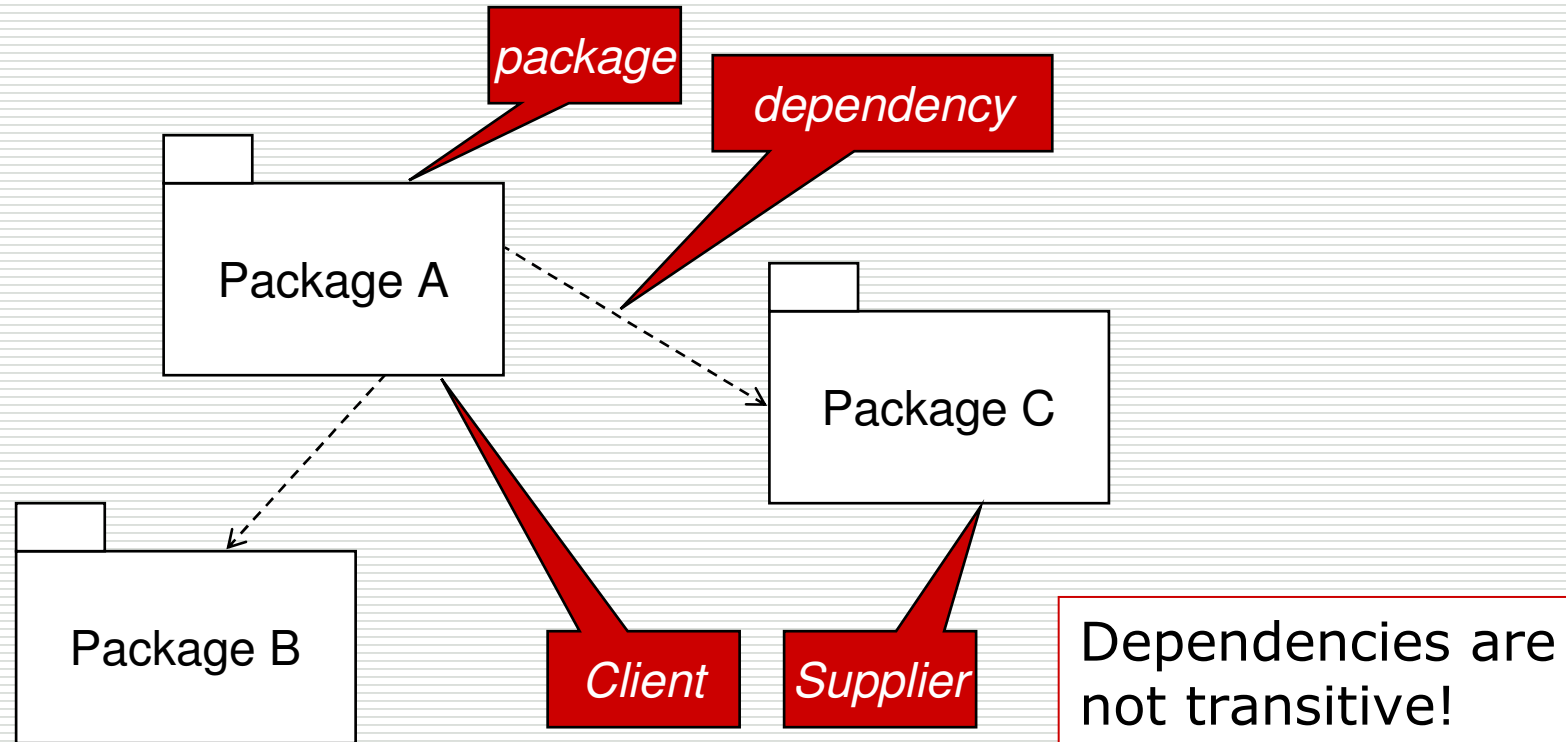


Specification Models in UML

- Architecture diagrams have these elements:
 - Package Diagrams:
 - Classes, Interfaces & Packages
 - Nested Packages
 - Dependencies
 - Deployment Diagrams:
 - Nodes & Components
- Specification models for architecture design :
 - **logical** structures: packages with package diagrams; subsystems; interfaces
 - **physical** structures: components in component diagrams; nodes; deployment diagrams

Package Diagram

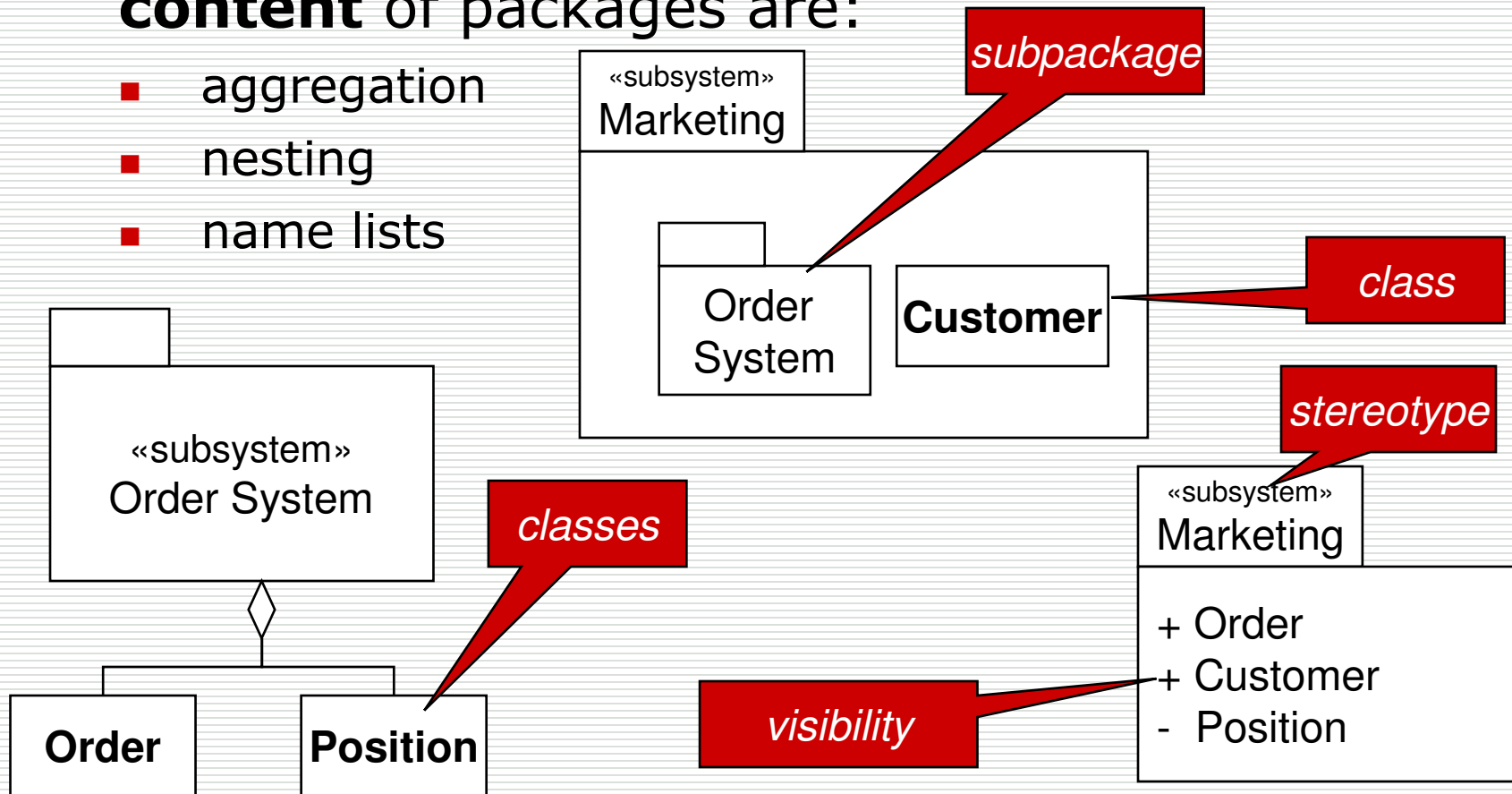
- A **package diagram** shows the coherence (*dependencies*) between different *packages* of the system.



Package Contents

- Alternatives for the representation of the **content** of packages are:

- aggregation
- nesting
- name lists



Subsystems

- Problem: How do I break down a large system into smaller systems?

Functional Decomposition

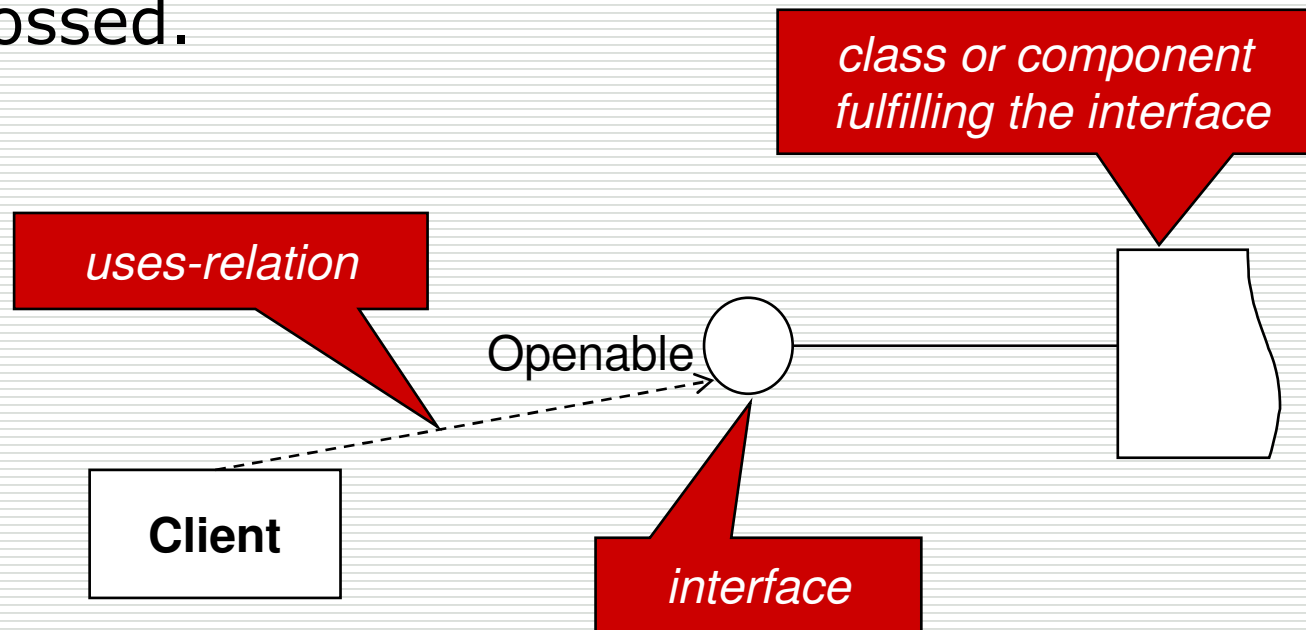
Map the total system on functions and subfunctions, starting with the use case.

OO Packages

- Collect classes into subsystems.
- Build layers of subsystems.
- **Abstraction:** Concentrate on essentials.
- **Locality:** Group together related components (data and algorithm).
- **Hiding:** Restrict the visibility of details, so that only those parts of a system that need to know the details have access to them.

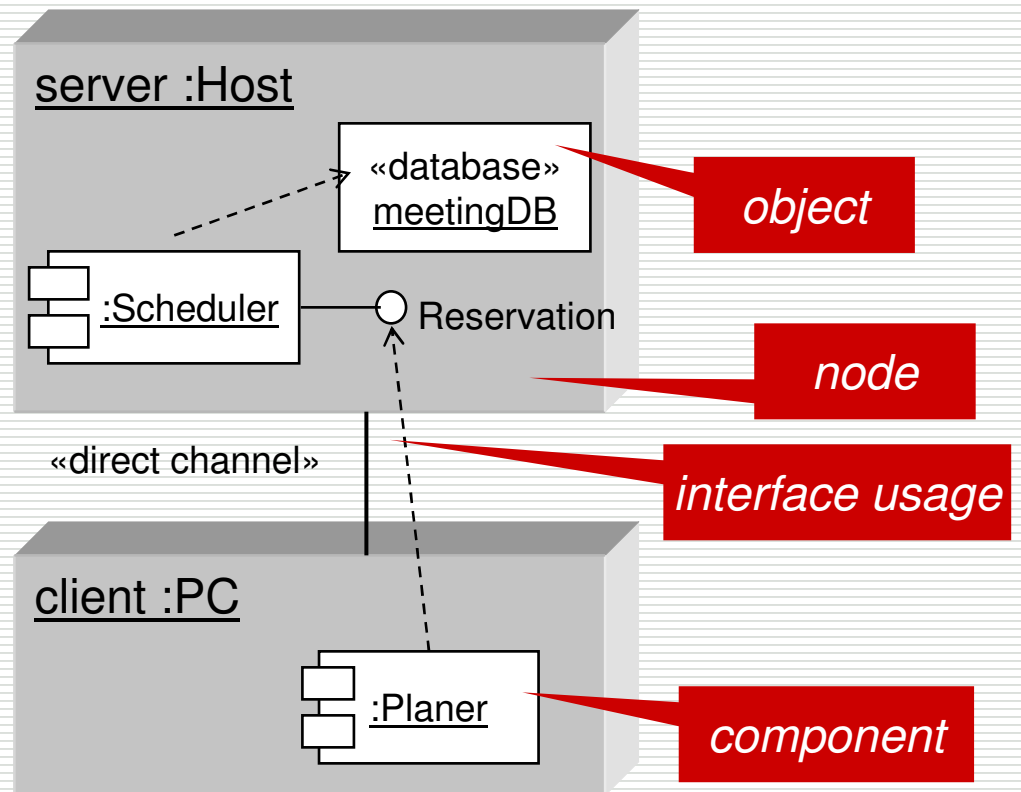
Interfaces in Architecture

- ❑ Interfaces let you specify the outward appearance.
- ❑ They are important in architecture diagrams, because they show how system boundaries are crossed.



Deployment Diagram

- A **deployment diagram** shows the configuration of a *node* at runtime as well as the *components* (instances) and *objects* residing on it.
- Components not existing as runtime object (instance) should appear in component diagrams only .



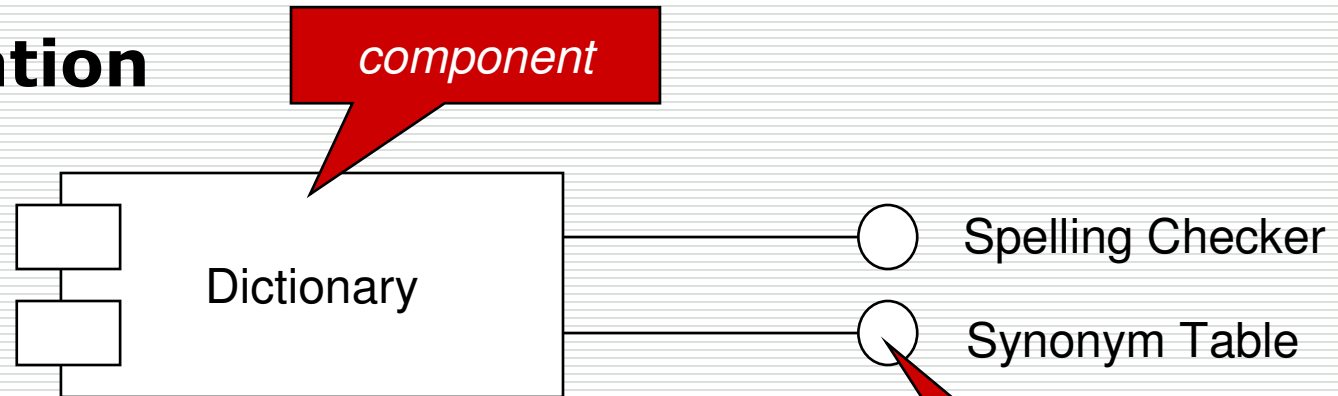
Components (1)

- A **component** is a physical, replaceable part of a system containing an implementation which realizes a set of interfaces
- Components have two aspects:
 - **Code:** A component consists of code, components can contain or use components.
 - **Identity:** A component can have identity and state represented by objects. An object which wants to use services of the component must specify the instance. (e.g., *Bean* reference, *DLL handle*, process, CORBA-IOR,...)

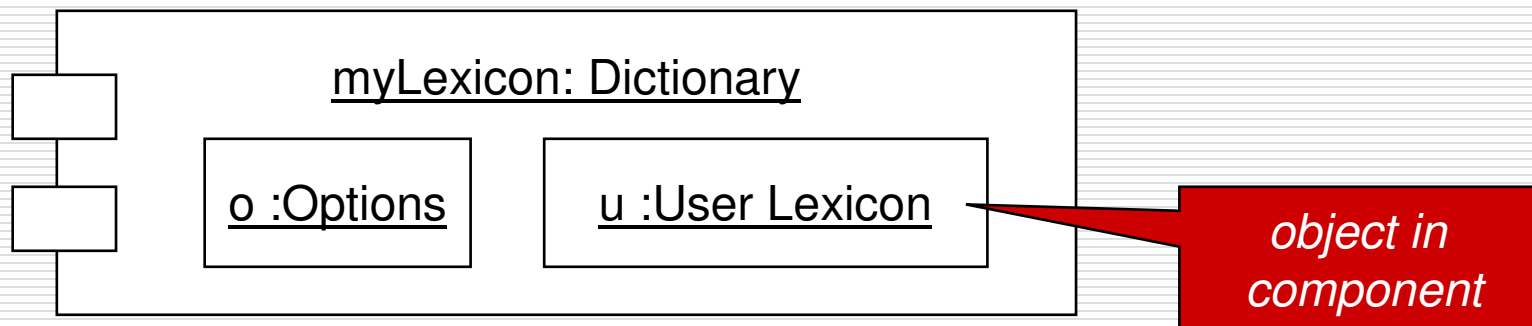
Example: Spelling checker as component:
Identity and state by user dictionary,
different versions and languages.

Components (2)

□ Notation

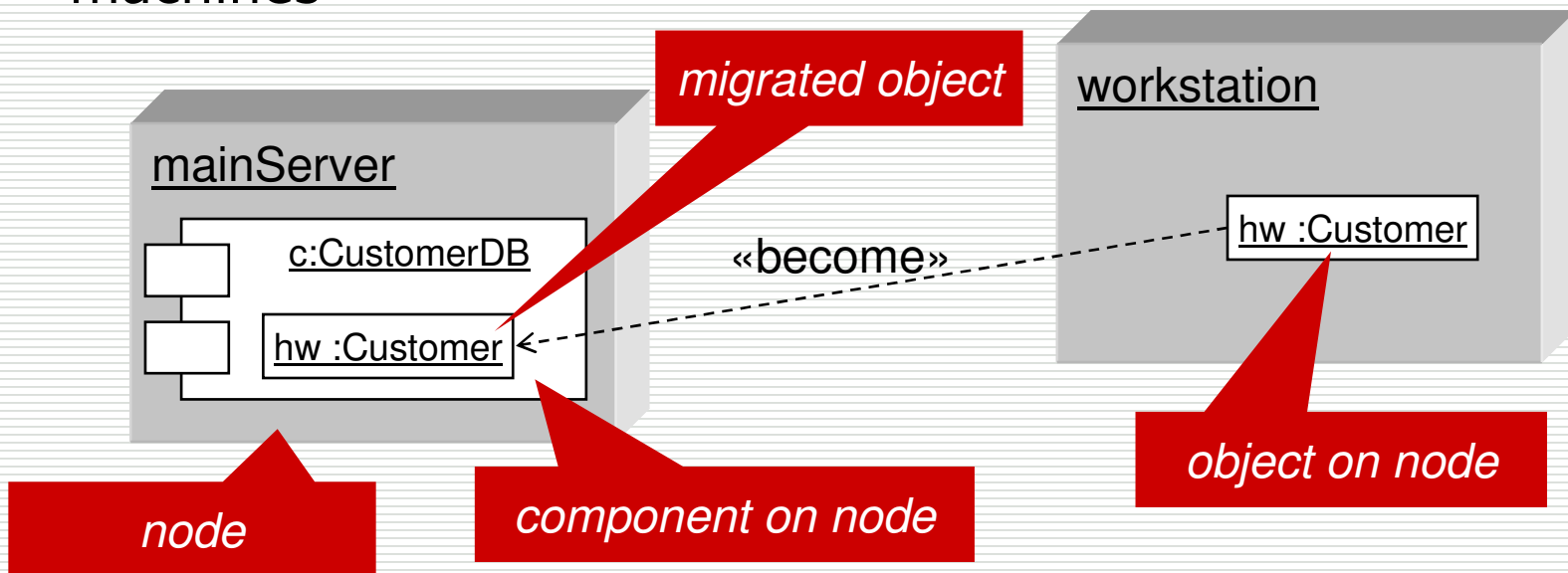


□ component with identity



Node

- A **node** is a physical entity existing at runtime
 - represents processing resources
 - has storage and computation capacity
 - On a node objects and components can be live
- Nodes can be computers, humans, or other devices or machines



Discussion

- Components ...
 - are conceptually and functionally larger than a class.
 - unite behavior and collaboration of a group of classes (see collaboration diagrams)
 - are independent of other components but usually collaborate with them
 - are replaceable with other realizations of the same interface
 - are fundamental building blocks of component-based architectures
 - can be built recursively. A system can be a component on the next higher inspection level.

References

□ Petri Nets

- Introduction:

- http://www.daimi.au.dk/PetriNets/introductions/pn2000_introtut.pdf

- Examples:

- <http://www.daimi.au.dk/PetriNets/introductions/aalst>