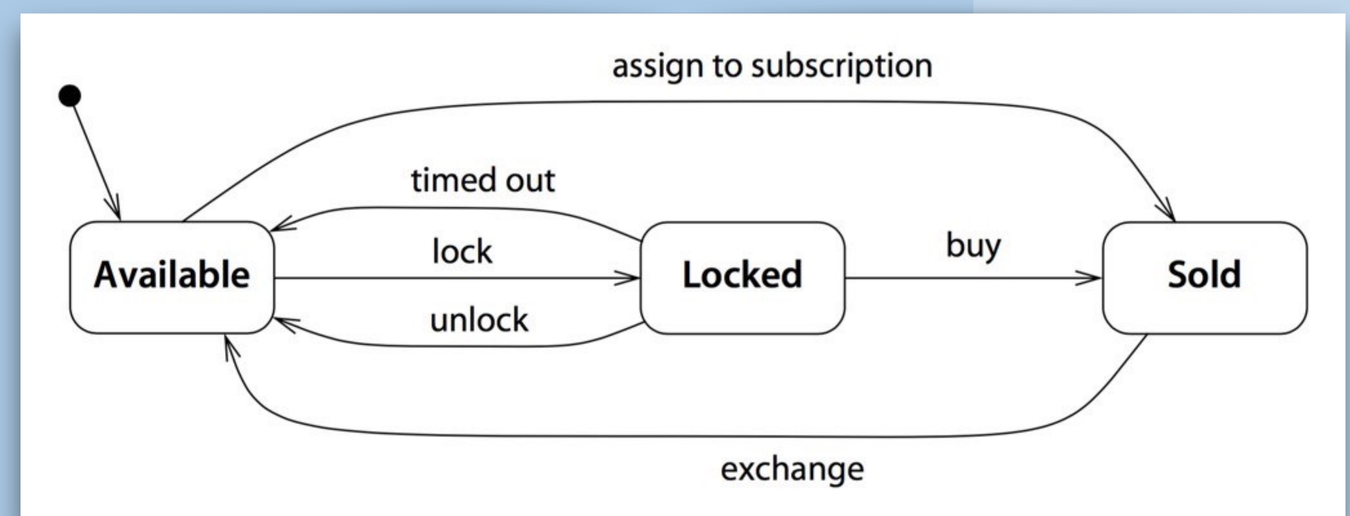
u^b

b UNIVERSITÄT BERN

Introduction to Software Engineering

Modeling Behaviour

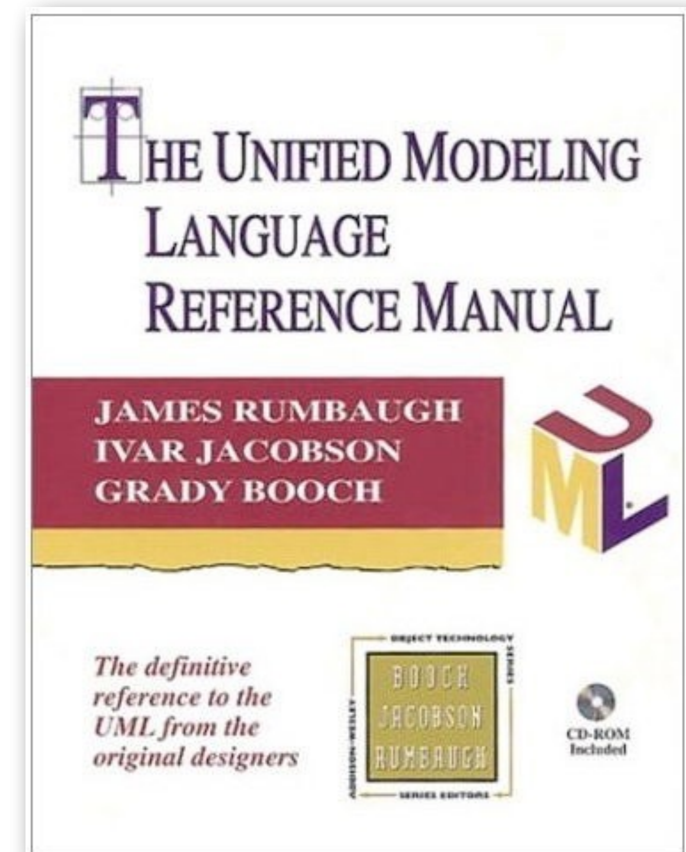


Roadmap

- Use Case Diagrams
- Sequence Diagrams
- Collaboration (Communication) Diagrams
- Activity Diagrams
- Statechart Diagrams
  - Nested statecharts
  - Concurrent substates
- Using UML

Source

- *The Unified Modeling Language Reference Manual*, James Rumbaugh, Ivar Jacobson and Grady Booch, Addison Wesley, 2005, 2nd edition.



Roadmap

- **Use Case Diagrams**
- Sequence Diagrams
- Collaboration (Communication) Diagrams
- Activity Diagrams
- Statechart Diagrams
  - Nested statecharts
  - Concurrent substates
- Using UML

Use Case Diagrams

A <u>use case</u> is a *generic description of an entire transaction* involving several actors.

A <u>use case diagram</u> presents a *set of use cases* (ellipses) and the external actors that interact with the system.

Dependencies and *associations* between use cases may be indicated.

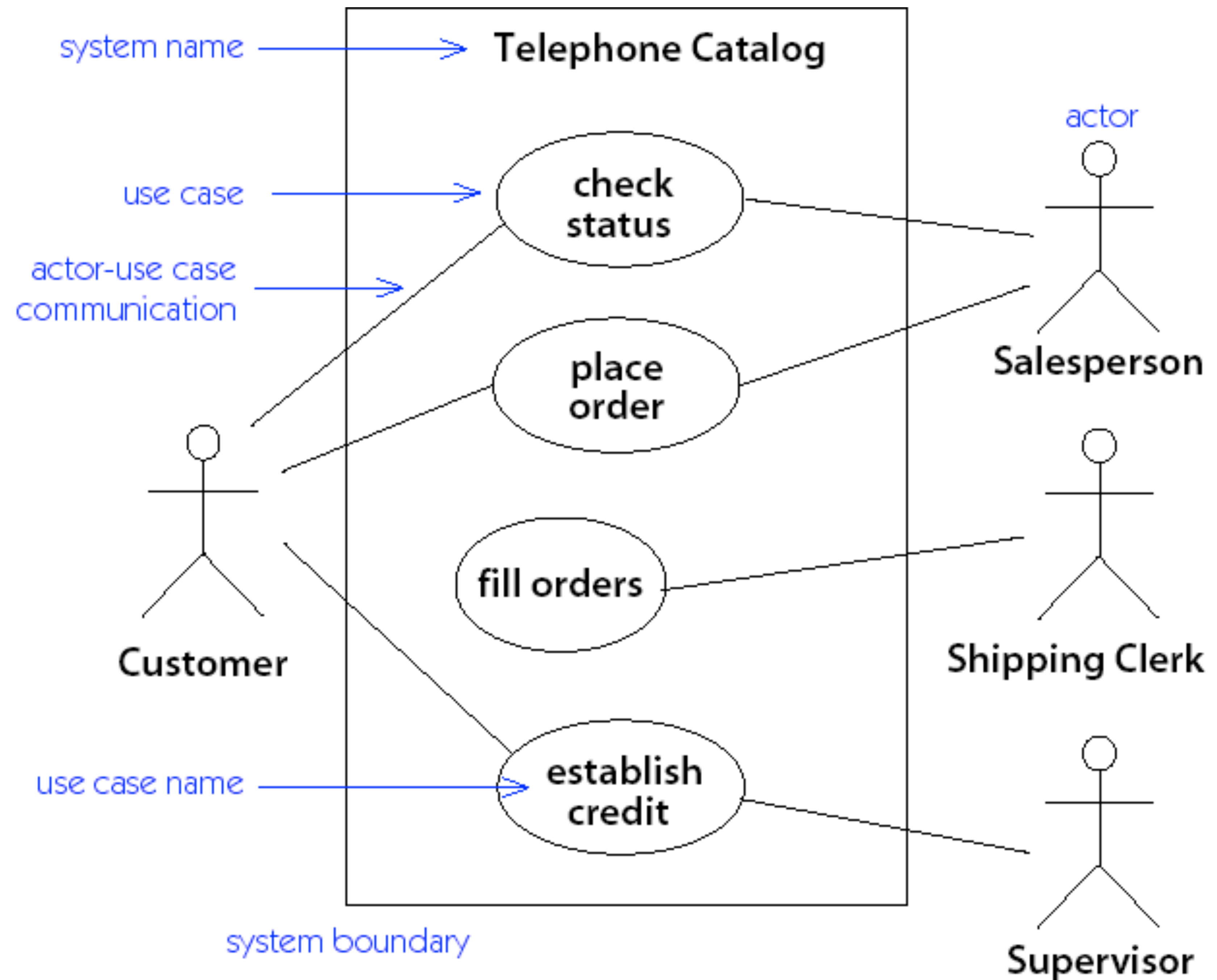


Figure 5-1. *Use case diagram*

Use case diagrams are very basic. They only show (i) what is the name of a use case; (ii) which actors participate in each use case; and (iii) any relationships between use cases (for example, generalization).

Using Use Case Diagrams

> "A use case is a *snapshot of one aspect* of your system. The sum of all use cases is *the external picture* of your system …"

— *UML Distilled*

> "As use cases appear, assess their impact on the domain model."

—Use cases can *drive domain modeling* by highlighting the important concepts.

Roadmap

- Use Case Diagrams
- **Sequence Diagrams**
- Collaboration (Communication) Diagrams
- Activity Diagrams
- Statechart Diagrams
  - Nested statecharts
  - Concurrent substates
- Using UML

Scenarios

A scenario is an *instance* of a use case showing a *typical example* of its execution.

Scenarios can be presented in UML using either *sequence diagrams* or *collaboration diagrams*.

Note that a scenario only describes an example of a use case, so conditionality cannot be expressed!

Sequence diagrams and collaboration diagrams are more or less equivalent in expressive power (you can translate one to the other), but they emphasize very different aspects, as we shall see.

Sequence Diagrams

A sequence diagram depicts a scenario by showing the interactions among a set of objects in *temporal order*.

Objects (not classes!) are shown as *vertical bars*. *Events* or message dispatches are shown as horizontal (or slanted) *arrows* from the sender to the receiver.

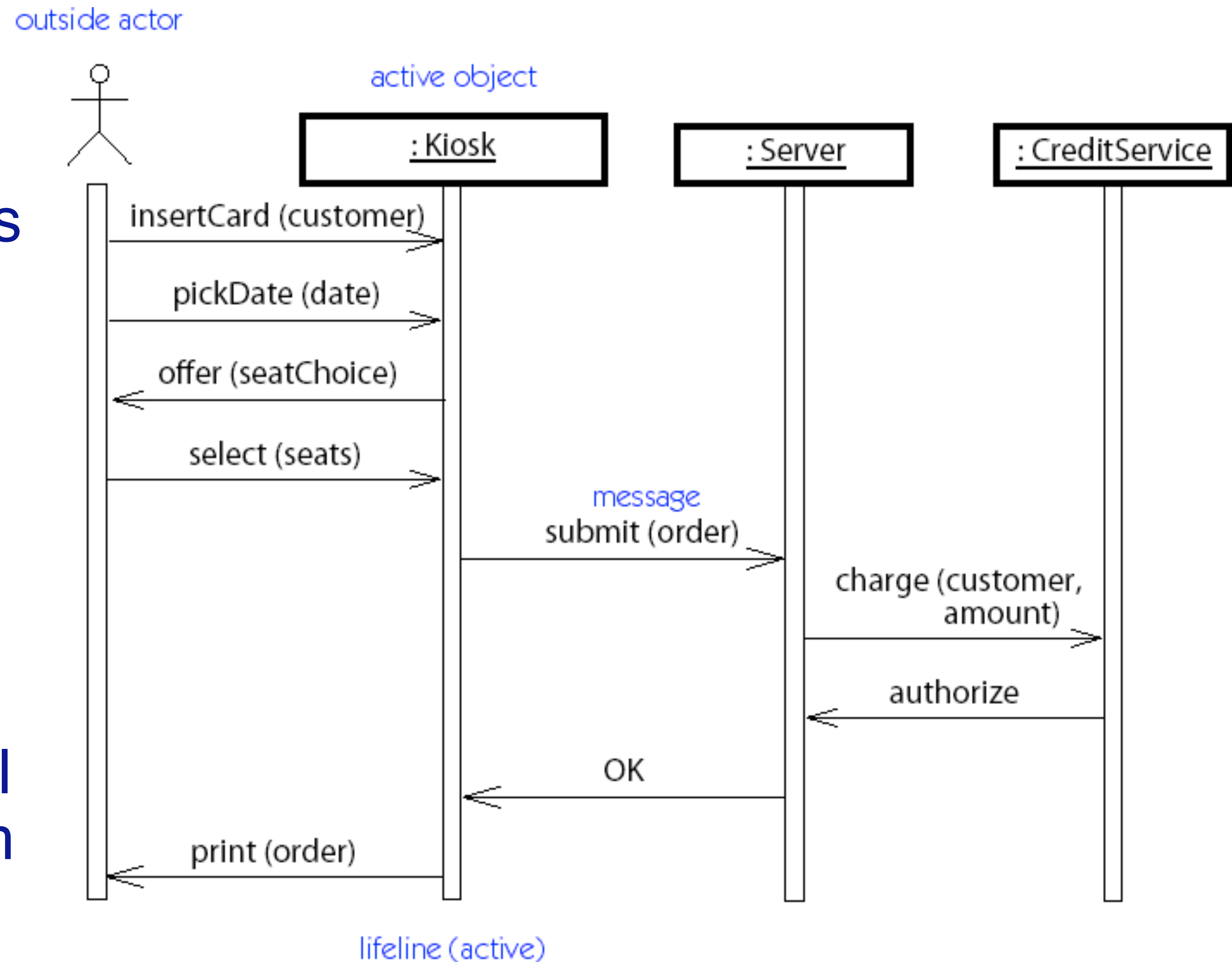


Figure 8-1. *Sequence diagram*

Sequence diagrams show a sequence of events in temporal order, from top to bottom. Each participating actor is given a timeline (shown as a fat line when it is active). The top of each timeline shows the object involved. Events are interactions between objects shown as arrows annotated with the kind of interaction (event or message).

Read the diagram from top to bottom: A customer inserts a card into a `Kiosk` machine, picks a date, obtains an offer for a choice of seats, and selects a seat. The `Kiosk` then submits an order to a `Server`, which charges a `CreditService`. The charge is authorized, the `Server` returns the ok to the `Kiosk`, which prints the ticket for the customer.

Note that a scenario always expresses a concrete sequence of events without any conditionality. To express, for example, what happens if the credit charge is not authorized we would need a separate scenario.

Activations

Avoid returns in sequence diagrams unless they add clarity.

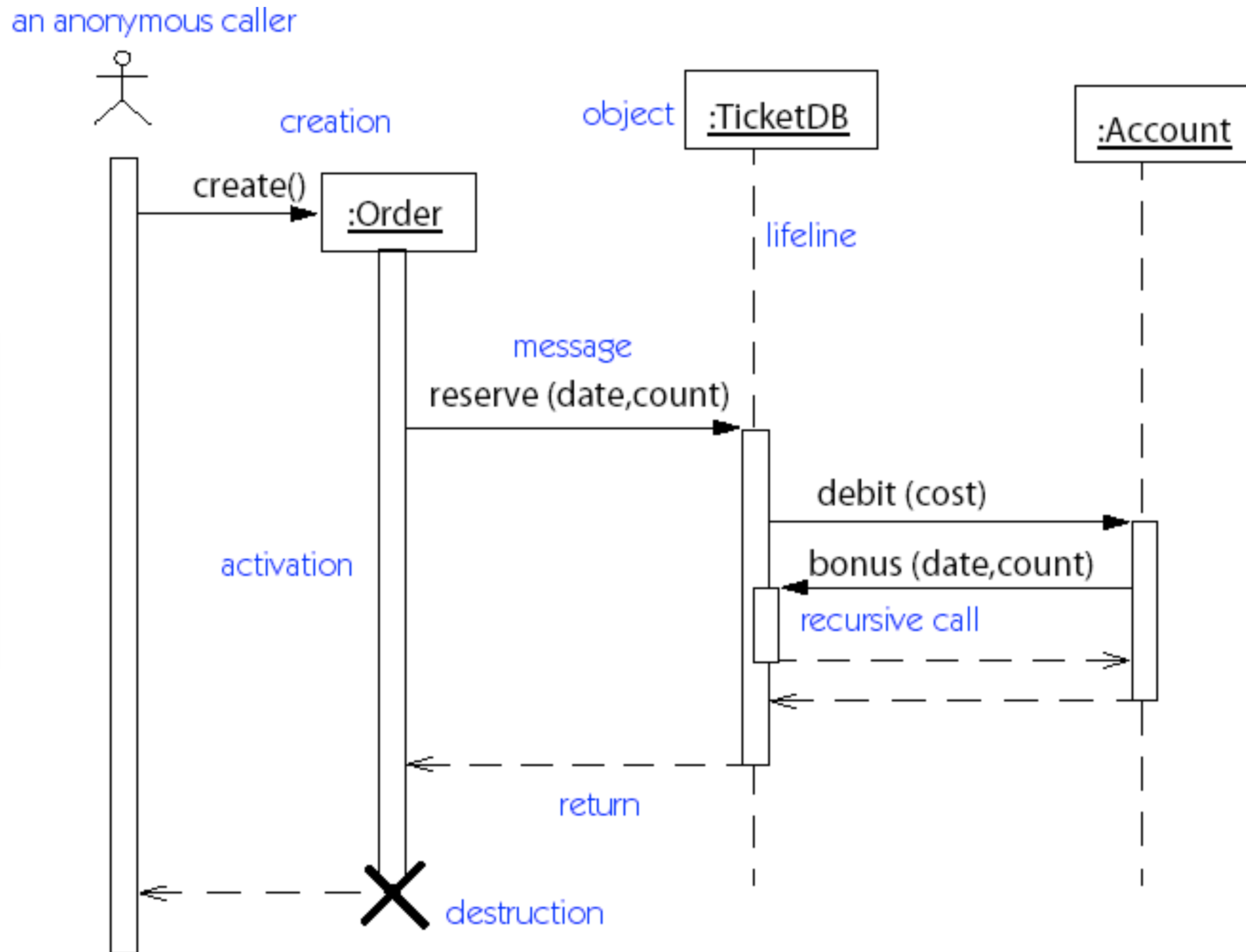


Figure 8-2. *Sequence diagram with activations*

In this sequence diagram we see several new features:

- The `Order` object does not exist at the start of the scenario but is created by the caller. It is also destroyed at the end.
- The `TicketDB` and the `Account` objects are only active when processing a message (timelines fat during processing).
- The interactions are not asynchronous messages but operation invocations (methods) which return results (dashed arrows). Note the different styles of arrows used.

Asynchrony and Constraints

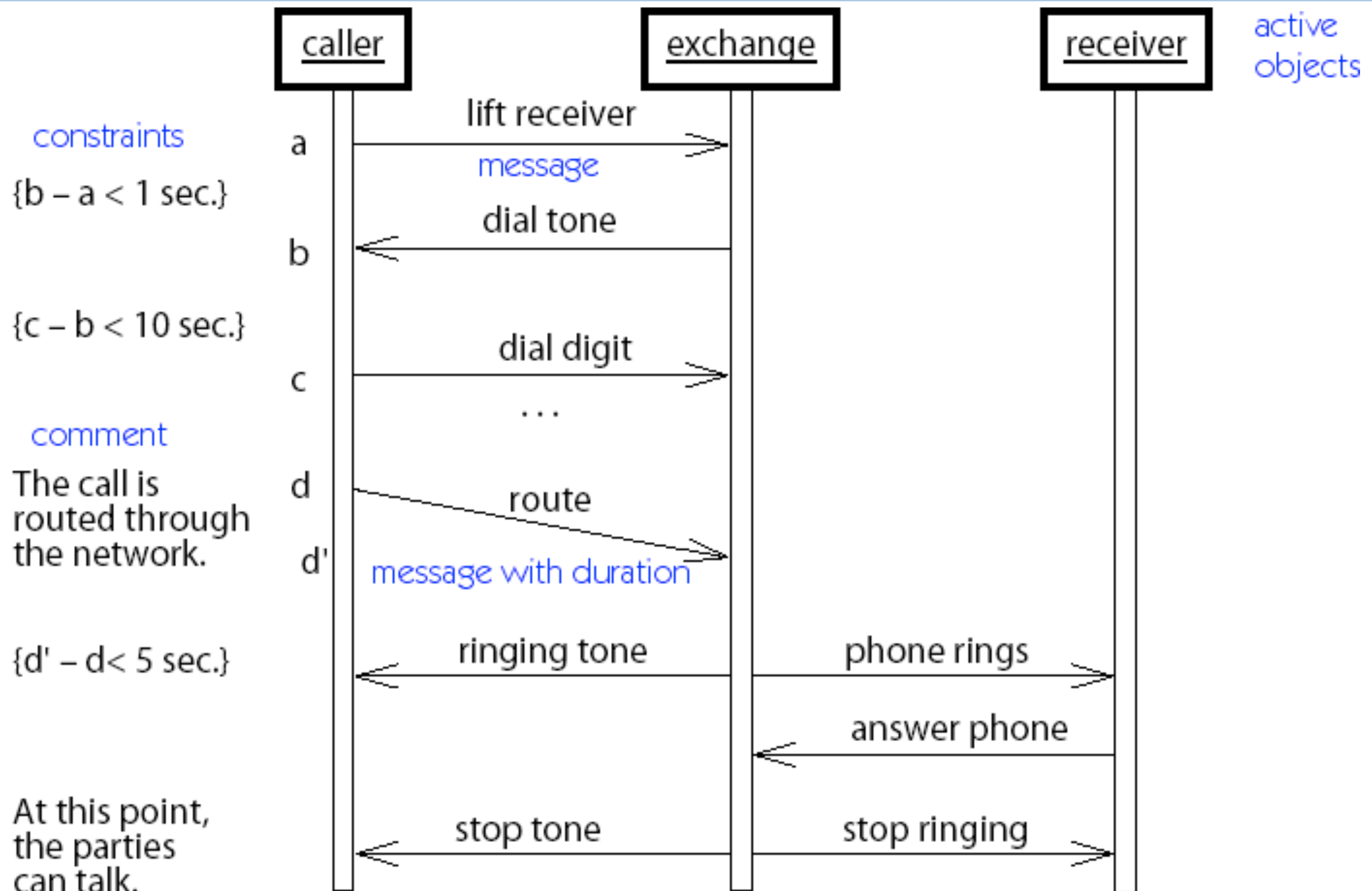


Figure 13-161. *Sequence diagram with asynchronous control*

In this scenario we also see timing constraints between asynchronous events. The dial tone must start within 1 second of the telephone receiver being lifted, and so on.

We also see an event with a duration (from d to d').

Exercise: turn this into a UML sequence diagram

Every committee is created at a Faculty meeting, where its members and the Chair are proposed and approved. The Chair makes a request to the Faculty secretary to schedule a meeting. All members are contacted and asked to fill out a "doodle" of possible dates. The secretary picks a date available to the Chair and a maximum of members. The secretary books a meeting room and informs all committee members of the selected time and location.

Exercise: turn this into a UML sequence diagram ...

Be sure to describe just a *single straight-line scenario*, without any conditional flow.

Ask yourself, “what are the participating *objects*?”, “what *events* occur?”, “what is the *flow* of events?”

Roadmap

> Use Case Diagrams
> Sequence Diagrams
> **Collaboration (Communication) Diagrams**
> Activity Diagrams
> Statechart Diagrams
—Nested statecharts
—Concurrent substates
> Using UML

Collaboration Diagrams

Collaboration diagrams (called *Communication* diagrams in UML 2.0) depict scenarios as *flows of messages* between objects:

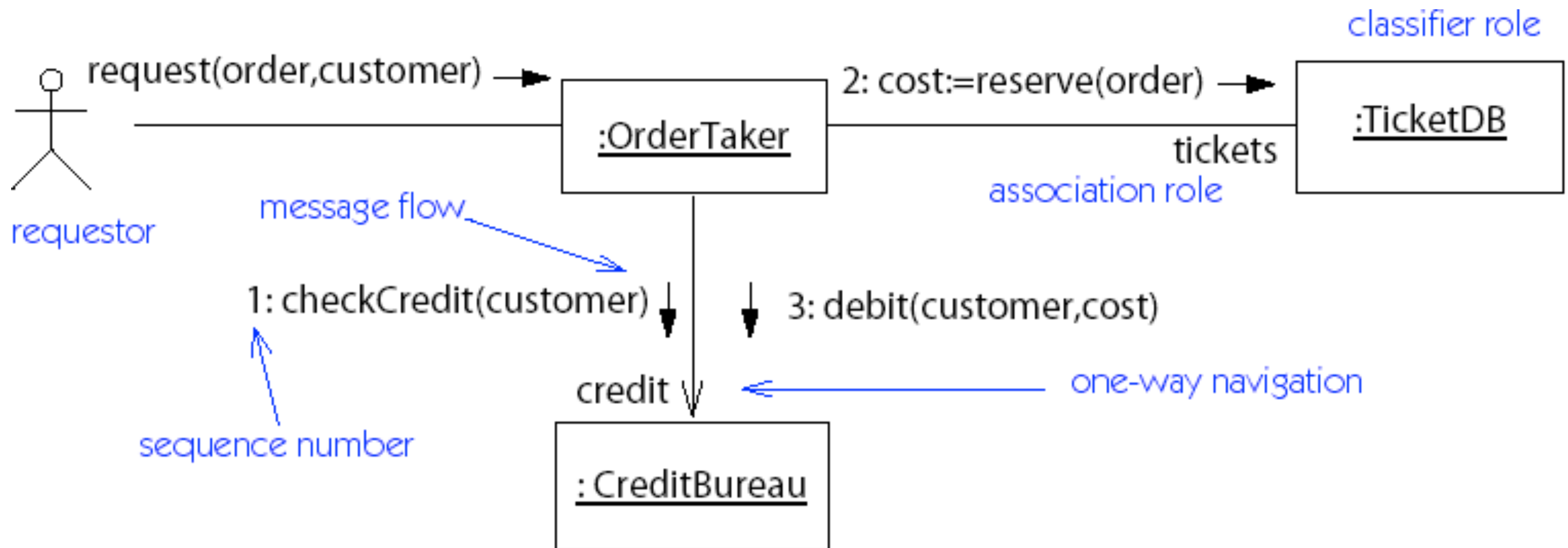


Figure 8-3. *Collaboration diagram*

Communication diagrams show essentially the same information as sequence diagrams (a scenario of events between interaction objects), but display that information in a fundamentally different way that emphasizes relationships between object rather than time. The sequence of events is indicated by numbering the messages. That way two dimensions (rather than just one) are available to lay out the objects, but time is harder to grasp.

Message Labels

Messages from one object to another are labelled with text strings showing the *direction* of message flow and information indicating the message *sequence*.

1. *Prior messages* from other threads (e.g. "[A1.3, B6.7.1]")
   - *only needed with* concurrent *flow of control*
2. Dot-separated list of sequencing elements
   - sequencing *integer (e.g., "3.1.2" is invoked by "3.1" and follows "3.1.1")*
   - *letter indicating* concurrent *threads (e.g., "1.2a" and "1.2b")*
   - iteration *indicator (e.g., "1.1*[i=1..n]")*
   - conditional *indicator (e.g., "2.3 [#items = 0]")*
3. *Return value* binding (e.g., "status :=")
4. Message name
   - *event or operation name*
5. Argument list

Nested Message Flows

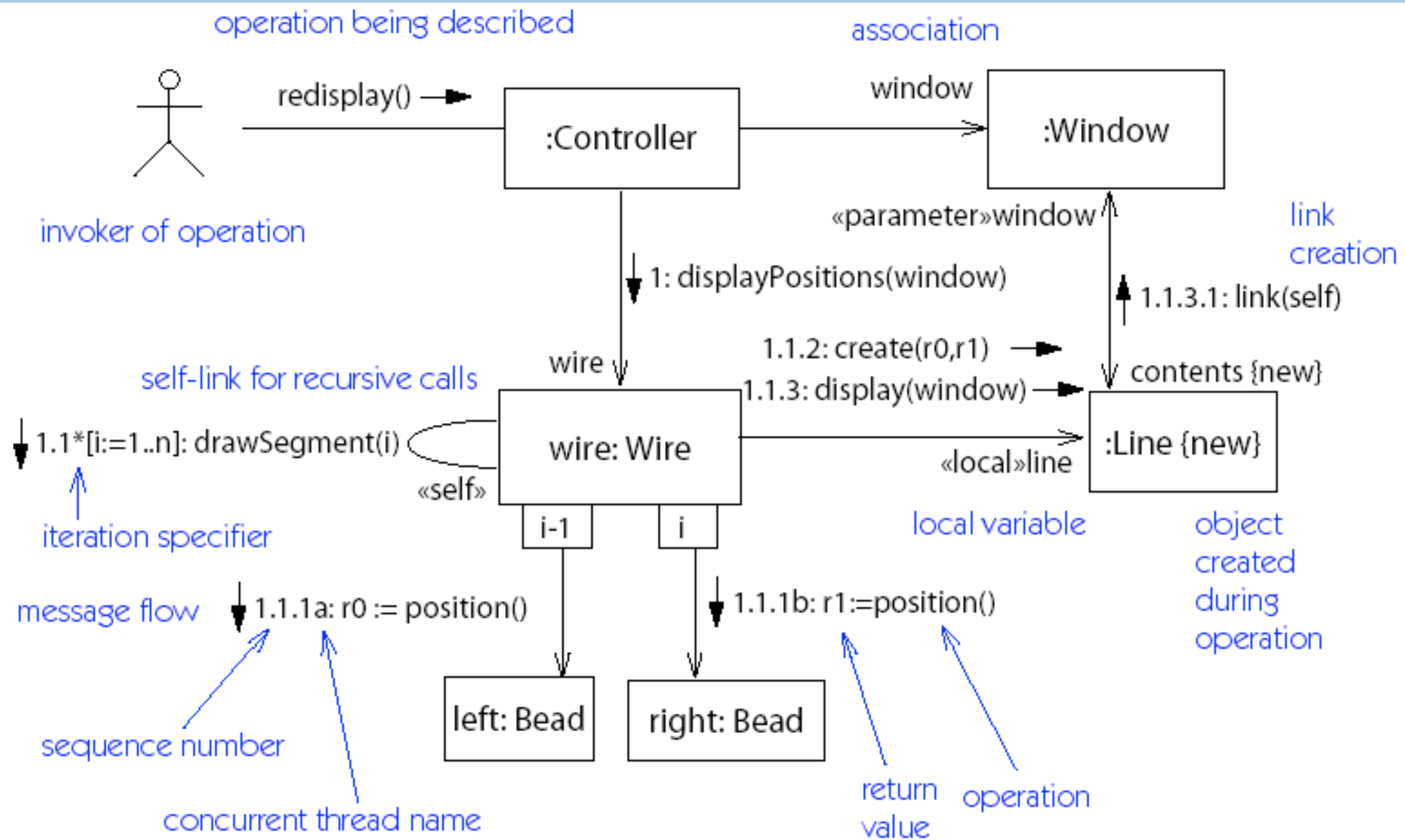


Figure 13-51. *Collaboration diagram with message flows*

In this scenario a user requests a redisplay of a graphical “`Wire`” made up of `Line` objects. The `Controller` asks the wire to to display itself (1) on the window, passed as a parameter. Each line segment is iteratively displayed (1.1*) by obtaining the left and right “bead” of each wire segment (1.1.1a and 1.1.1b). For each segment a Line is created (1.1.2) and displayed (1.1.3) by “linking” itself to the window (1.1.3.1).

Roadmap

- Use Case Diagrams
- Sequence Diagrams
- Collaboration (Communication) Diagrams
- **Activity Diagrams**
- Statechart Diagrams
  - Nested statecharts
  - Concurrent substates
- Using UML

Activity Diagrams

An activity diagram models the *control flow* (i.e., execution states) of a computation or workflow

In other words: an *object-oriented flowchart*

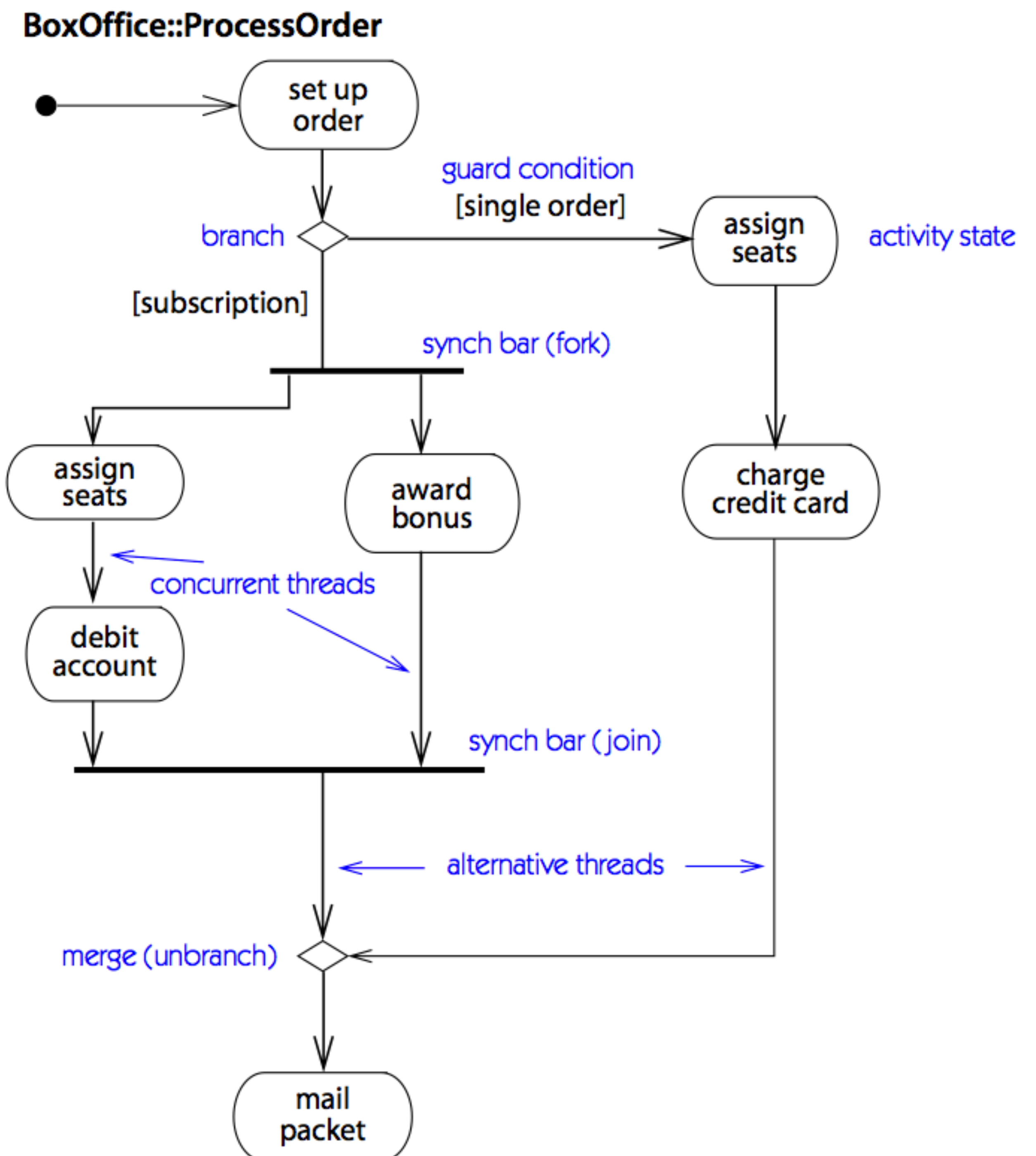


Figure 7-1. *Activity diagram*

An activity diagram is similar to an old-fashioned flowchart, except that it can model concurrency, which flowcharts do not.

https://en.wikipedia.org/wiki/Activity_diagram

Swimlanes and object flows

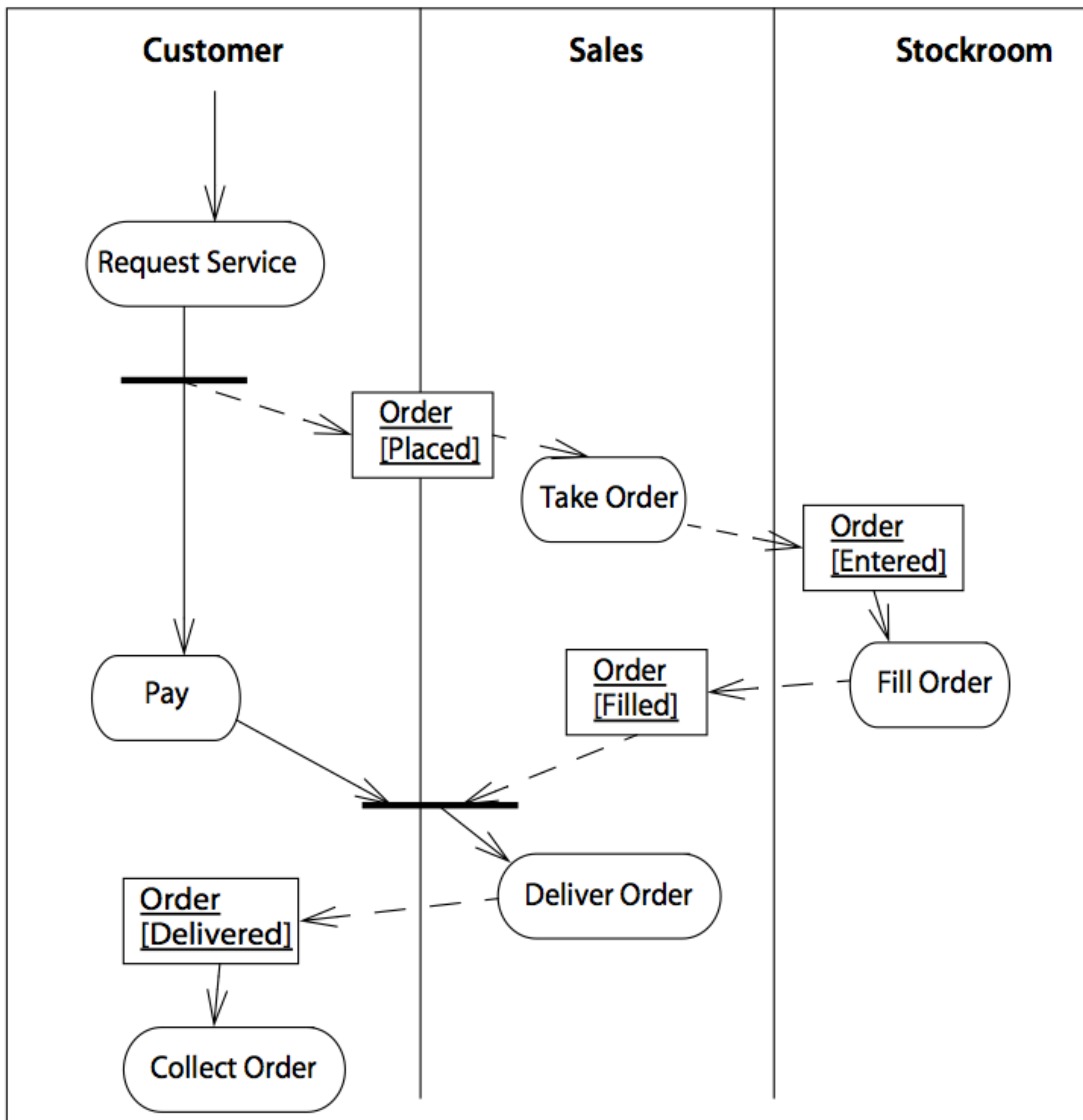


Figure 7-2. *Swimlanes and object flows*

Activity diagrams can express *collaboration*.

Swimlanes group activities by *responsibilities*.

Object flows depict objects that are the *outputs* or *inputs* of activities.

Activity diagrams depict collaboration between objects places into separate “swim lanes”. The resulting diagrams can express coordination similar to the way that Petri nets do.

The horizontal bars in the example resemble “transitions” in Petri net that wait for all inputs to be available before being triggered, and may produce multiple outputs. In the example, a service request triggers both payment and order placement activities. Both payment must be received and the order must be filled before the order can be delivered.

See also:

https://en.wikipedia.org/wiki/Petri_net

Roadmap

- Use Case Diagrams
- Sequence Diagrams
- Collaboration (Communication) Diagrams
- Activity Diagrams
- **Statechart Diagrams**
  - Nested statecharts
  - Concurrent substates
- Using UML

Statechart Diagrams

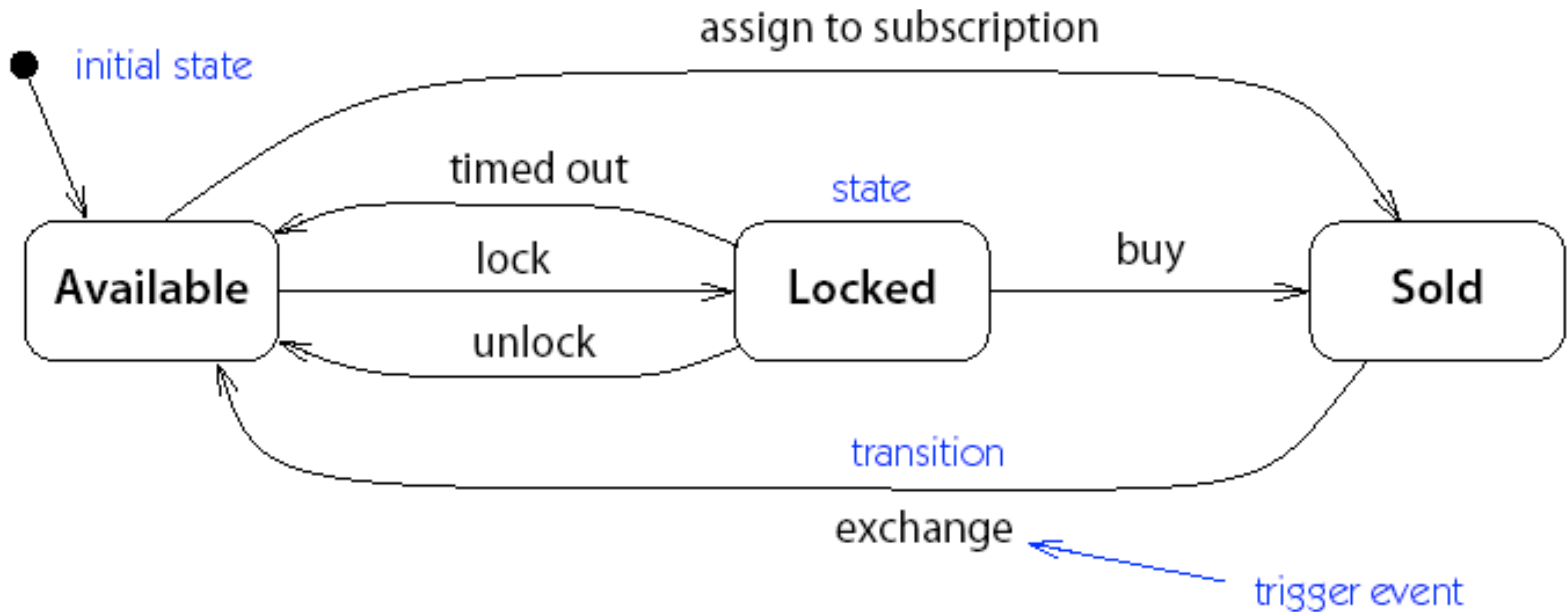


Figure 3-5. *Statechart diagram*

Statecharts were introduced by David Harel in 1987 as a more compact visual formalism for representing state diagrams (finite state machines). They have been subsequently incorporated into the UML.

This particular statechart is identical to its equivalent state diagram. There are three states — *Available*, *Locked* and *Sold* — with transitions between them.

https://en.wikipedia.org/wiki/State_diagram#Harel_statechart

Statechart Diagram Notation

A Statechart Diagram describes the *temporal evolution* of an object of a given class in response to *interactions* with other objects inside or outside the system.

An event is a one-way (asynchronous) communication from one object to another:

- —*atomic* (non-interruptible)
- —includes events from *hardware* and real-world objects e.g., message receipt, input event, elapsed time, ...
- —notation: ***eventName(parameter: type, ...)***
- —may cause object to make a *transition* between states

Statechart Diagram Notation ...

A <u>state</u> is a period of time during which an object is *waiting* for an event to occur:

- depicted as *rounded box* with (up to) three sections:
  - name — *optional*
  - state *variables — name: type = value (valid only for that state)*
  - triggered operations — *internal transitions and ongoing operations*
- may be *nested*

State Box with Regions

The *entry event* occurs whenever a transition is made into this state, and the *exit operation* is triggered when a transition is made out of this state.

The *help* and *character* events cause internal transitions with no change of state, so the entry and exit operations are not performed.

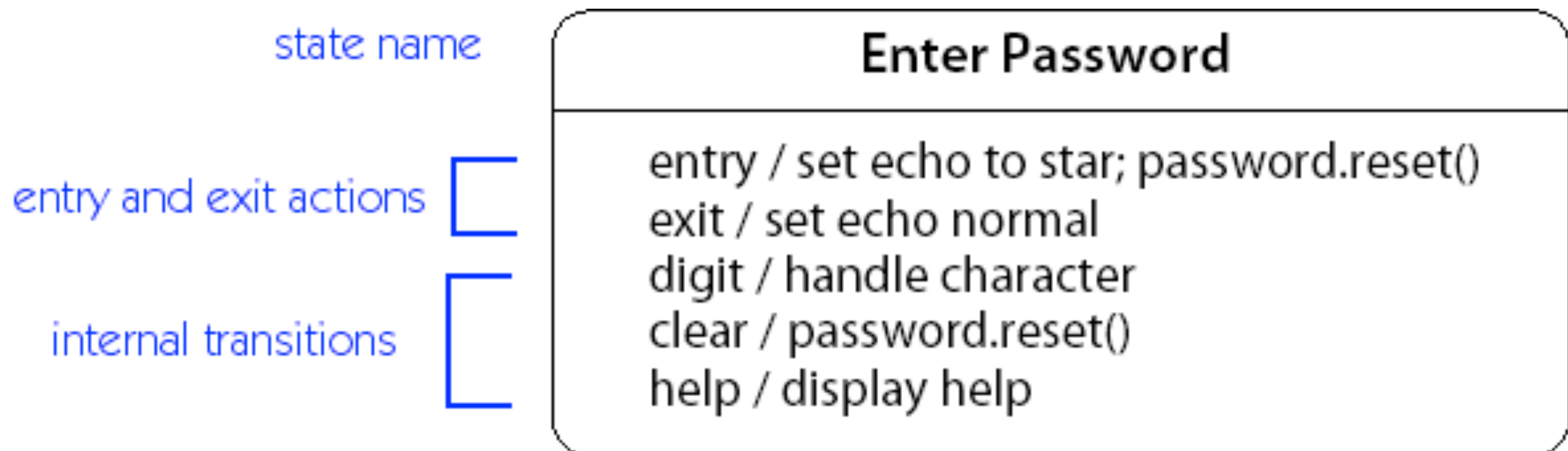


Figure 6-4. *Internal transitions, and entry and exit actions*

Transitions

A transition is an *response to an external event* received by an object in a *given state*

- —May *invoke* an operation, and cause the object to change state
- —May *send* an event to an external object
- —Transition syntax (each part is optional):

  event(arguments) [condition]
  / ^target.sendEvent operation(arguments)
- —*External transitions* label arcs between states
- —*Internal transitions* are part of the triggered operations of a state

Operations and Activities

An operation is an *atomic action* invoked by a transition

— *Entry and exit operations* can be associated with states

An activity is an *ongoing operation* that takes place while object is in a given state

— Modelled as "internal transitions" labelled with the pseudo-event ***do***

Roadmap

> Use Case Diagrams
> Sequence Diagrams
> Collaboration (Communication) Diagrams
> Activity Diagrams
> Statechart Diagrams
  — **Nested statecharts**
  — Concurrent substates
> Using UML

Nested Statecharts

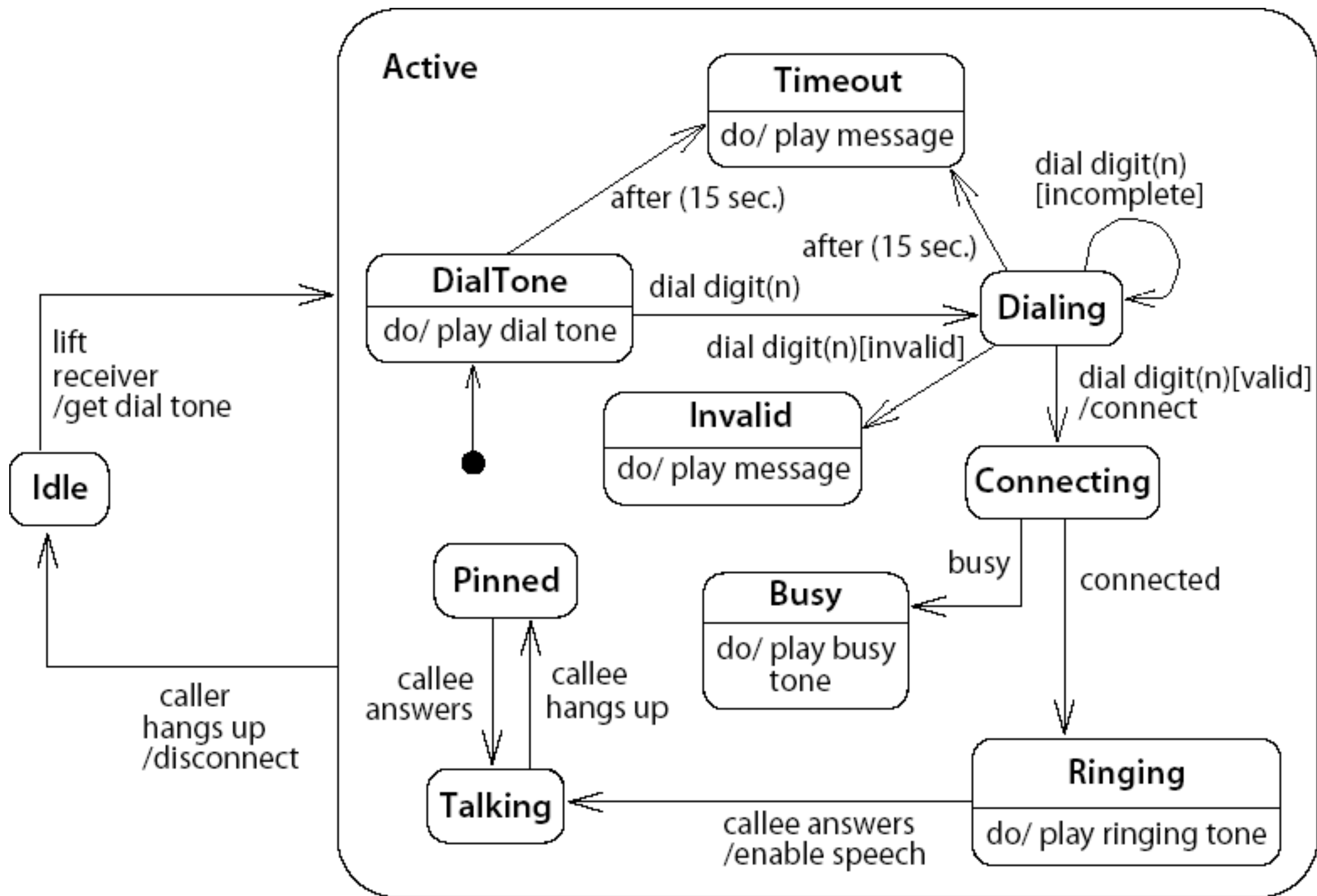


Figure 13-169. *State diagram*

Here we see the advantage of nested statecharts over traditional state diagrams. In a conventional state diagram we would require a transition from each substate in the *Active* region back to *Idle* in case the caller hangs up. Here we only need to indicate that transitions once from the top-level *Active* state to *Idle*.

Within the *Active* region we enter the initial substate *DialTone*.

Note the distinction between ongoing activities within a state (such a do/play dial tone) and atomic operations associated with transitions (such as dial digit(n)).

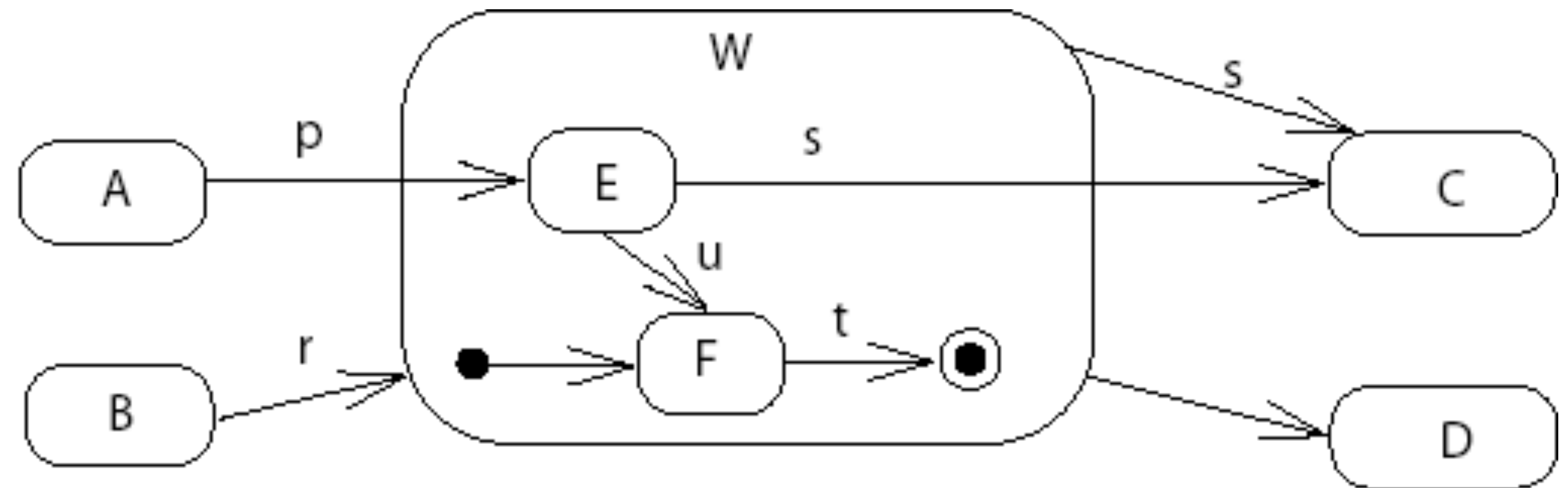
Also note that the statechart stays in the terminal substates *Invalid* or *Busy* until the caller hangs up.

Composite States

Composite states may depicted either as high-level or low-level views.

"Stubbed transitions" indicate the presence of internal states:

Initial and terminal substates are shown as black spots and "bulls-eyes"



may be abstracted as

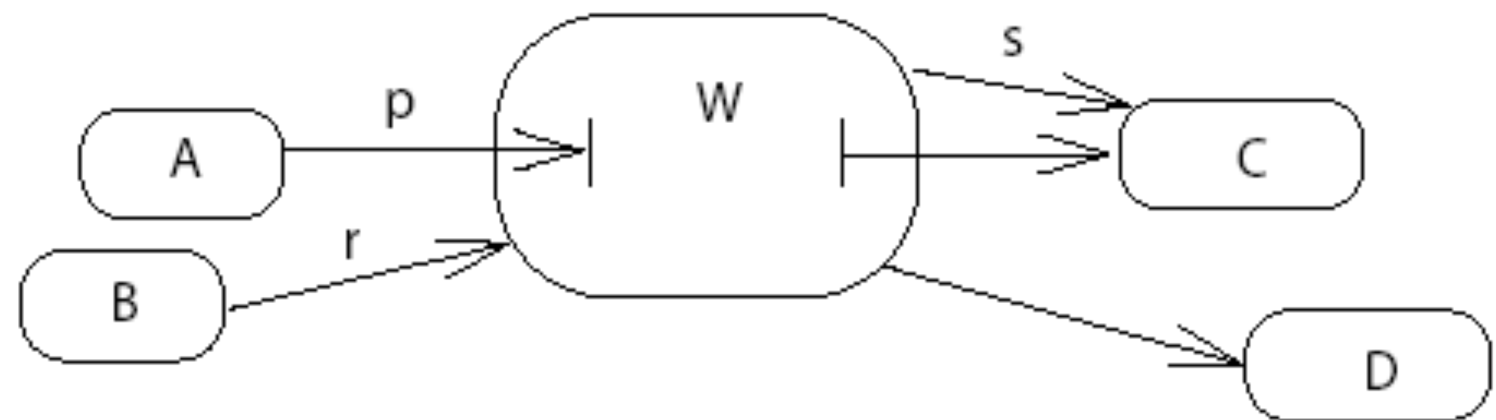


Figure 13-172. *Stubbed transition*

Sending Events between Objects

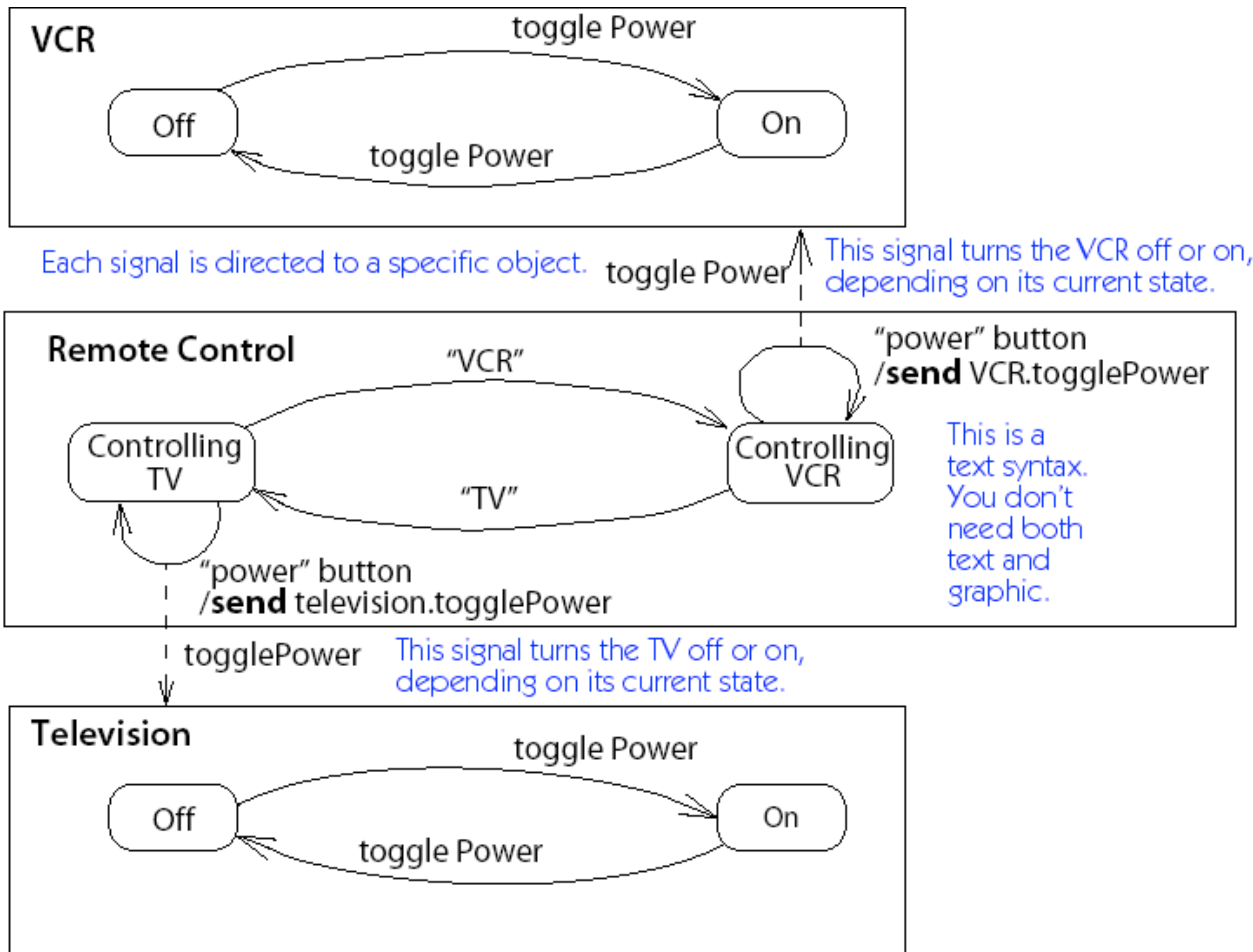


Figure 13-160. *Sending signals between objects*

In this example the *RemoteControl* statechart controls either the *VCR* or the *Television*.

There are two separate ways to specify the interaction (both are shown in the example, but only one is needed).

Either we specify an "*send event*" operation as party of a transition (e.g., *send television.togglePower*), or we connect the transition to the other statechart with a dashed arrow labeled by the event that is signaled (*togglePower*).

Roadmap

- > Use Case Diagrams
- > Sequence Diagrams
- > Collaboration (Communication) Diagrams
- > Activity Diagrams
- > Statechart Diagrams
  - —Nested statecharts
  - —**Concurrent substates**
- > Using UML

Concurrent Substates

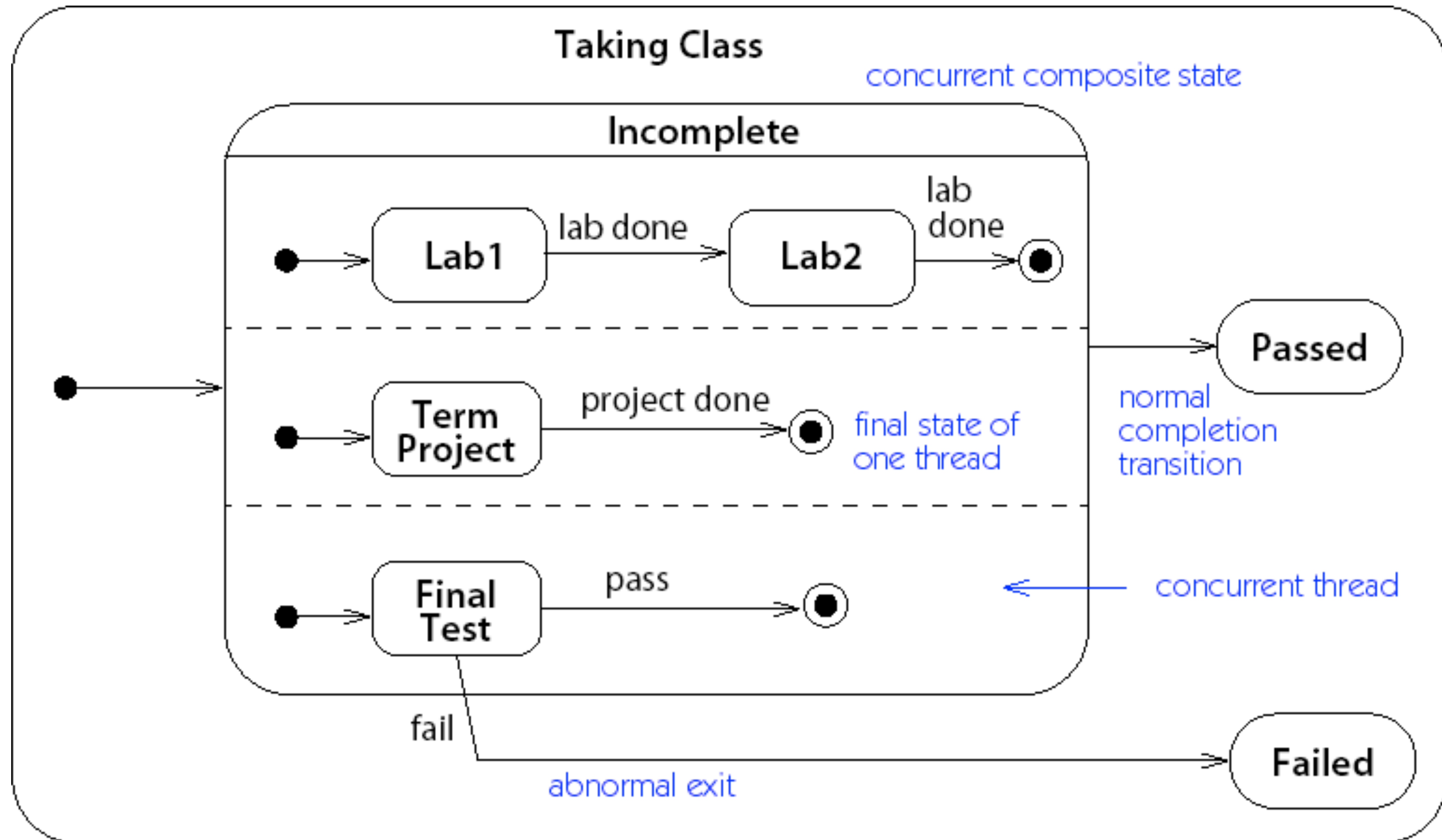


Figure 6-6. *State machine with concurrent composite state*

In addition to nested substates, concurrent substates can also greatly reduce the complexity of state diagrams. When entering a substate with multiple concurrent substates, we enter each subchart simultaneously.

When we start to take a class, we concurrently enter three parallel statecharts, one for the *Lab*, another for the *Term Project* and one for the *Final Test*. We leave these subcharts only when they all reach their final state (the bullseye), or an event occurs to abnormally leave all three (fail).

Expressing this behaviour with a traditional state diagram would require us to draw a state machine with one state for each possible combination of substates.

Branching and Merging

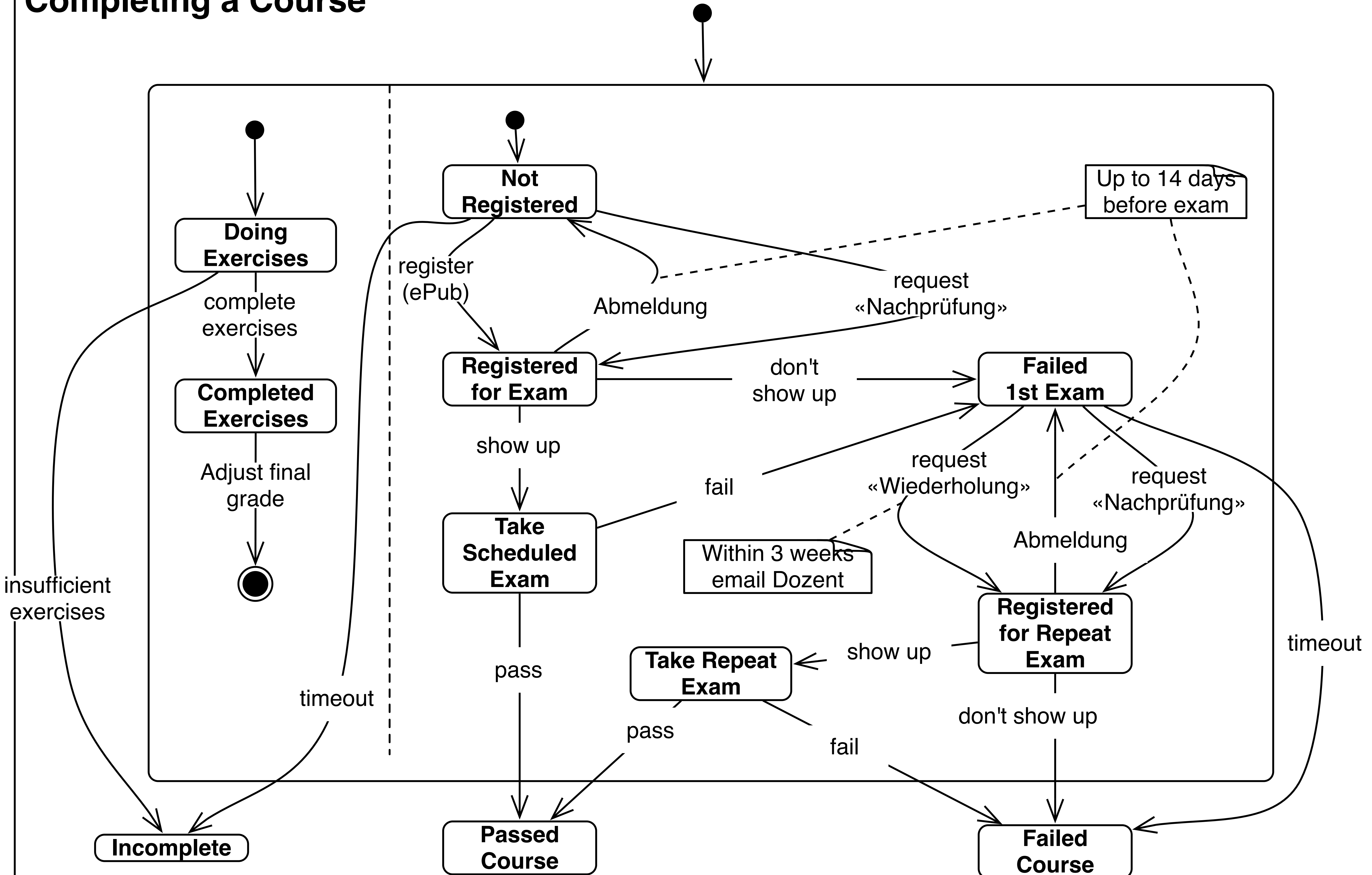
Entering concurrent states:

Entering a state with concurrent substates means that *each of the substates is entered concurrently* (one logical thread per substate).

Leaving concurrent states:

A *labelled transition* out of any of the substates *terminates all of the substates.*

An *unlabelled transition* out of the overall state *waits* for all substates to terminate.



Is it correct?

I prepared this real example some years ago in an effort to understand whether the regulations governing exams in the Computer Science Institute at the University of Bern were complete and consistent. The point is not so much whether the diagram correctly interprets the regulations, but that it serves as a basis for discussion and analysis.

Exercise: turn this into a UML statechart …

Committees may be formed for a fixed duration (e.g., hiring a professor) or for an indeterminate duration (e.g., finances). A committee is first proposed, and then approved by the Faculty who names its members. If a member or a chair retires from a committee, the faculty should name its replacements. Once a committee is formed, it may meet at regular or irregular intervals. Once a committee has fulfilled its task it is dissolved and any decisions or reports are archived.

Exercise: turn this into a UML statechart …

What are the *entities* for which we want to capture states? What are these *states*? What *events* trigger the *transitions* between states? Are there *nested states*?

Roadmap

- Use Case Diagrams
- Sequence Diagrams
- Collaboration (Communication) Diagrams
- Activity Diagrams
- Statechart Diagrams
  - Nested statecharts
  - Concurrent substates
- **Using UML**

Perspectives

Three perspectives in drawing UML diagrams:

1. ***Conceptual***
   —Represent domain concepts
   - *Ignore software issues*
2. ***Specification***
   —Focus on visible interfaces and behaviour
   - *Ignore internal implementation*
3. ***Implementation***
   —Document implementation choices
   - *Most common, but least useful perspective(!)*

— *UML Distilled*

Using the Notations

The diagrams introduced here complement class and object diagrams.

During Analysis:

— Use case, sequence and collaboration diagrams document *use cases and their scenarios* during requirements specification

During Design:

— Sequence and collaboration diagrams can be used to document *implementation scenarios* or refine use case scenarios

— State diagrams document *internal behaviour* of classes and must be *validated* against the specified use cases

What you should know!

- What is the purpose of a use case diagram?
- Why do scenarios depict objects but not classes?
- How can timing constraints be expressed in scenarios?
- How do you specify and interpret message labels in a scenario?
- How do you use nested state diagrams to model object behaviour?
- What is the difference between “external” and “internal” transitions?
- How can you model interaction between state diagrams for several classes?

Can you answer the following questions?

- Can a sequence diagram always be translated to an collaboration diagram?
- Or vice versa?
- Why are arrows depicted with the message labels rather than with links?
- When should you use concurrent substates?

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format
Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/