State Chart Diagrams

Introduction to State Diagrams

- Interaction diagrams are very good for communication
  - between clients and designers
  - between designers and other designers
  - between designers and implementors
- But they are not part of the **constructive core**
  - Constructive core means that part from which an actual implementation could be generated
- The constructive core of the UML is
  - class diagrams (for the static part)
  - state diagrams (for the dynamic part)
- State diagrams can describe very complex dynamic behavior

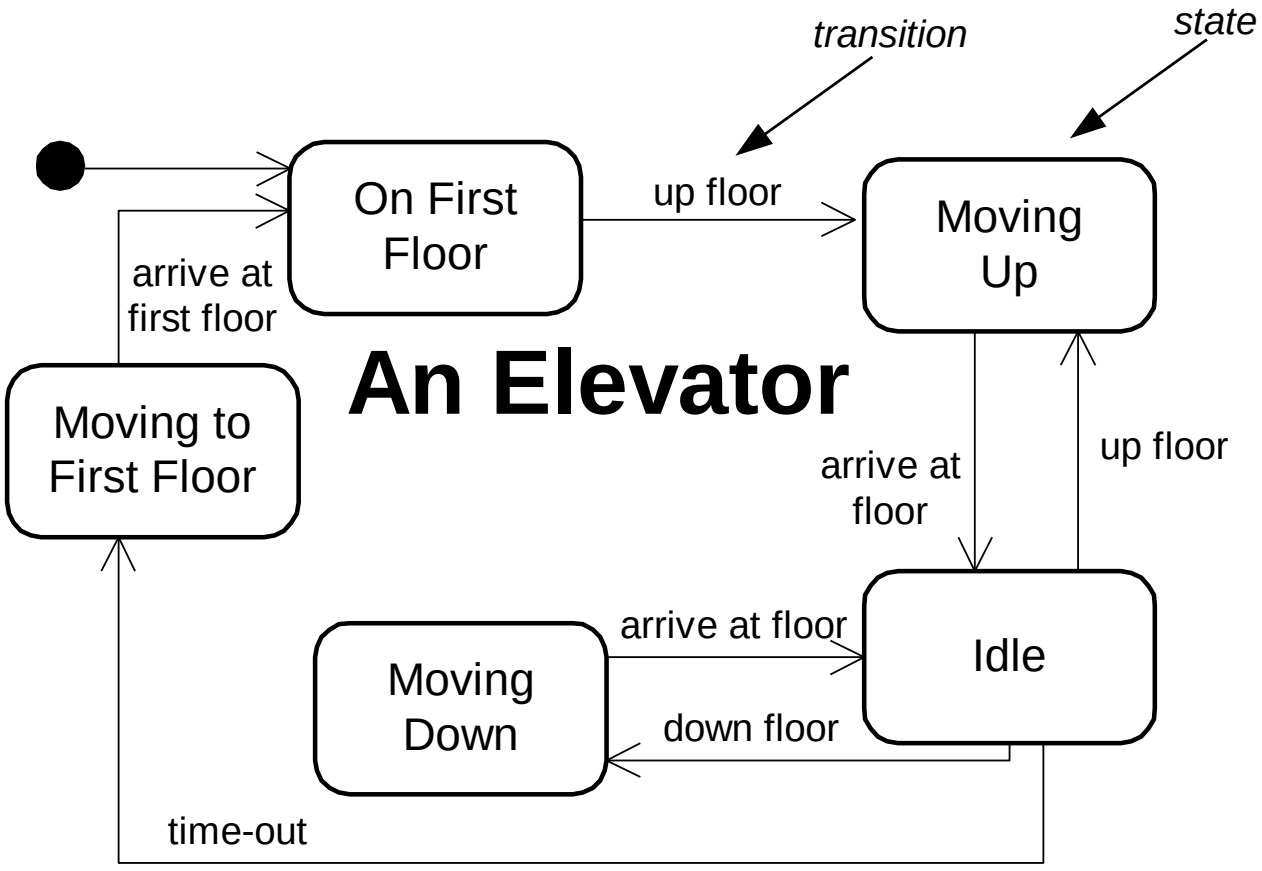
Elements of State Diagrams

- The basic elements of state diagrams are
  - *states* – the state in which the object finds itself at any moment
  - *transitions* – take the object from one state to another
  - *events* – cause the transition from one state to another
  - *actions* – take place as a result of a transition
- In the UML, many other extensions have been added, such as:
  - generalization/Inheritance among events (i.e. a mouse click and keyboard input as subclasses of user interface events)
  - hierarchical diagrams for managing complexity (from StateCharts)
  - guards – conditions on transitions
- State Diagrams should only be used for objects with significant dynamic behavior

Types of Events

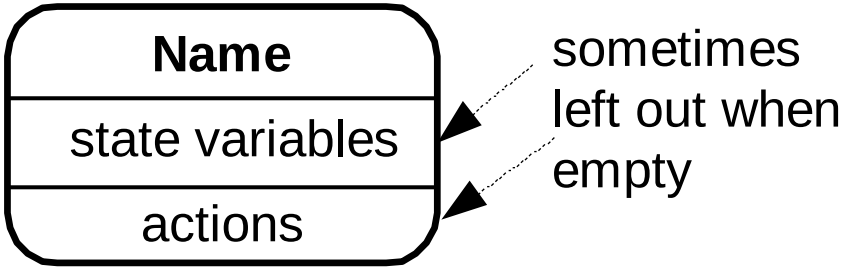
- **ChangeEvent** (i.e.: when(x>y))
  - A change event models an event that occurs when an explicit boolean expression becomes true as a result of a change in value of one or more attributes or associations.
  - A change event is raised implicitly and is not the result of some explicit change event action.
- **CallEvent** (i.e.: conflictsWith(Appointment))
  - A call event represents the reception of a request to synchronously invoke a specific operation.
  - Note that a call event instance is distinct from the call action that caused it.
- **SignalEvent** (i.e.: leftButtonDown)
  - A signal event represents the reception of a particular (asynchronous) signal.
- **TimeEvent** (i.e.: after(1 second))
  - A TimeEvent models the expiration of a specific deadline. Note that the time of occurrence of a time event instance (i.e., the expiration of the deadline) is the same as the time of its reception.

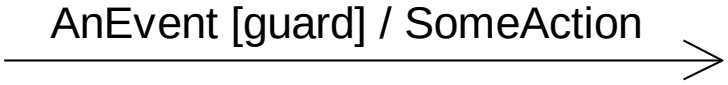
States and Transitions



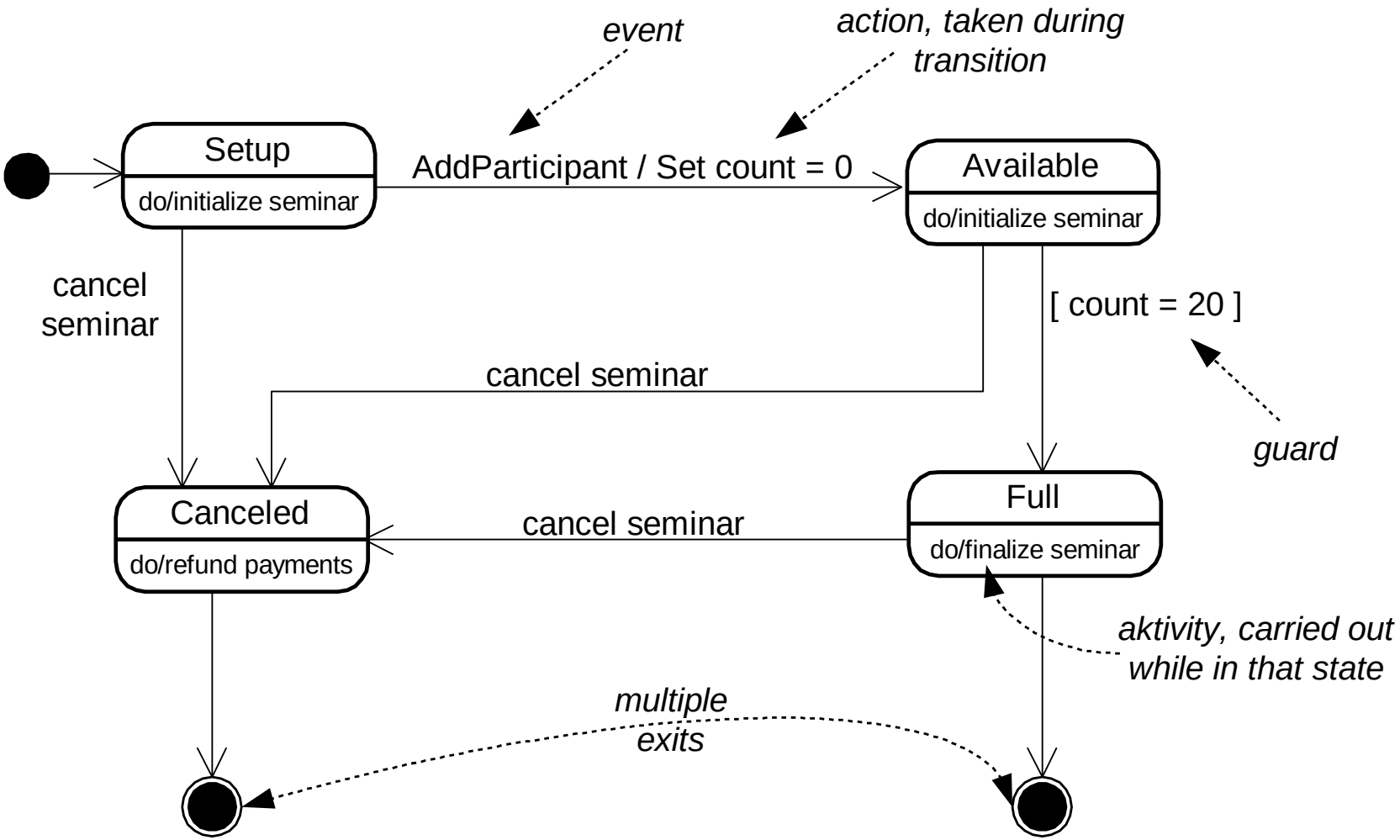
Basic UML State Diagram Syntax

- A state is drawn with a round box, with three compartments for
  - name
  - state variables (if any)
  - actions to be performed
- A transition is drawn with an arrow, possibly labeled with
  - event causing the transaction
  - guard condition
  - Action to perform





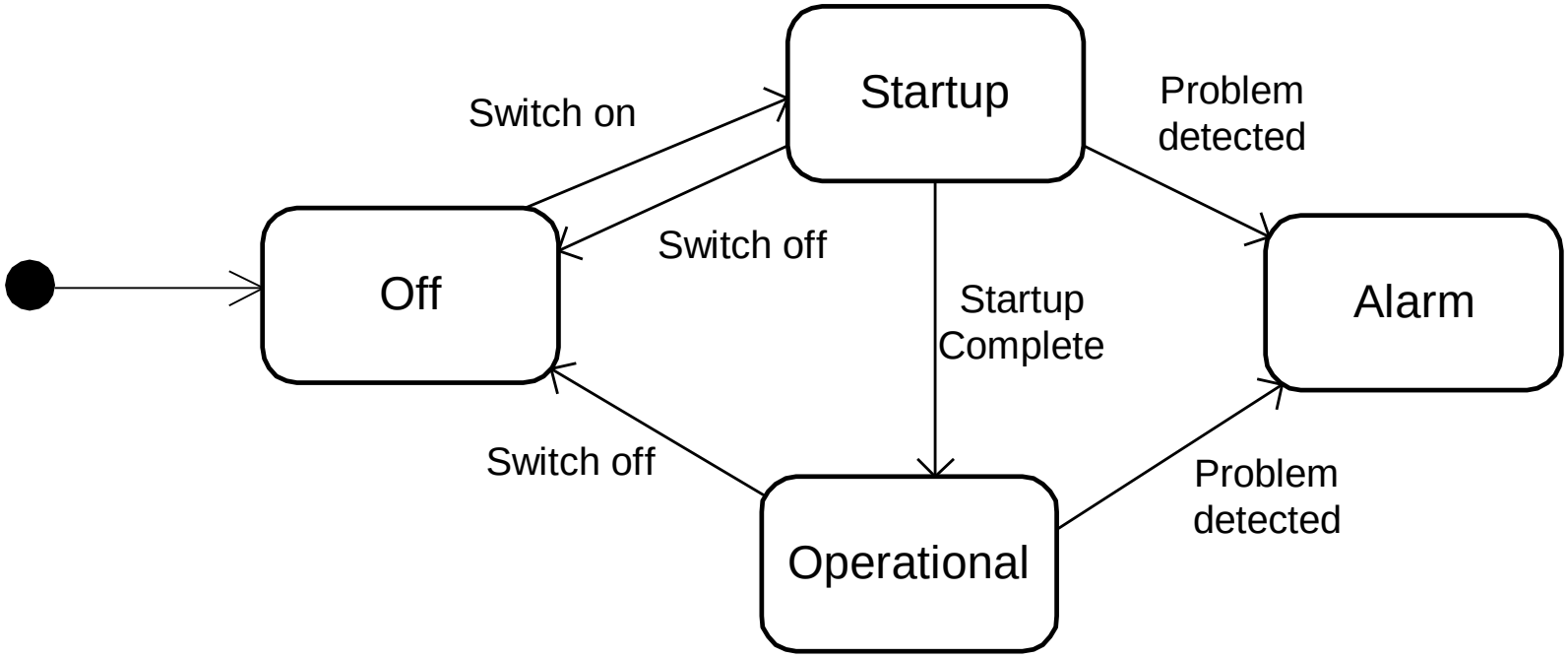
State Diagram for Seminar Registration



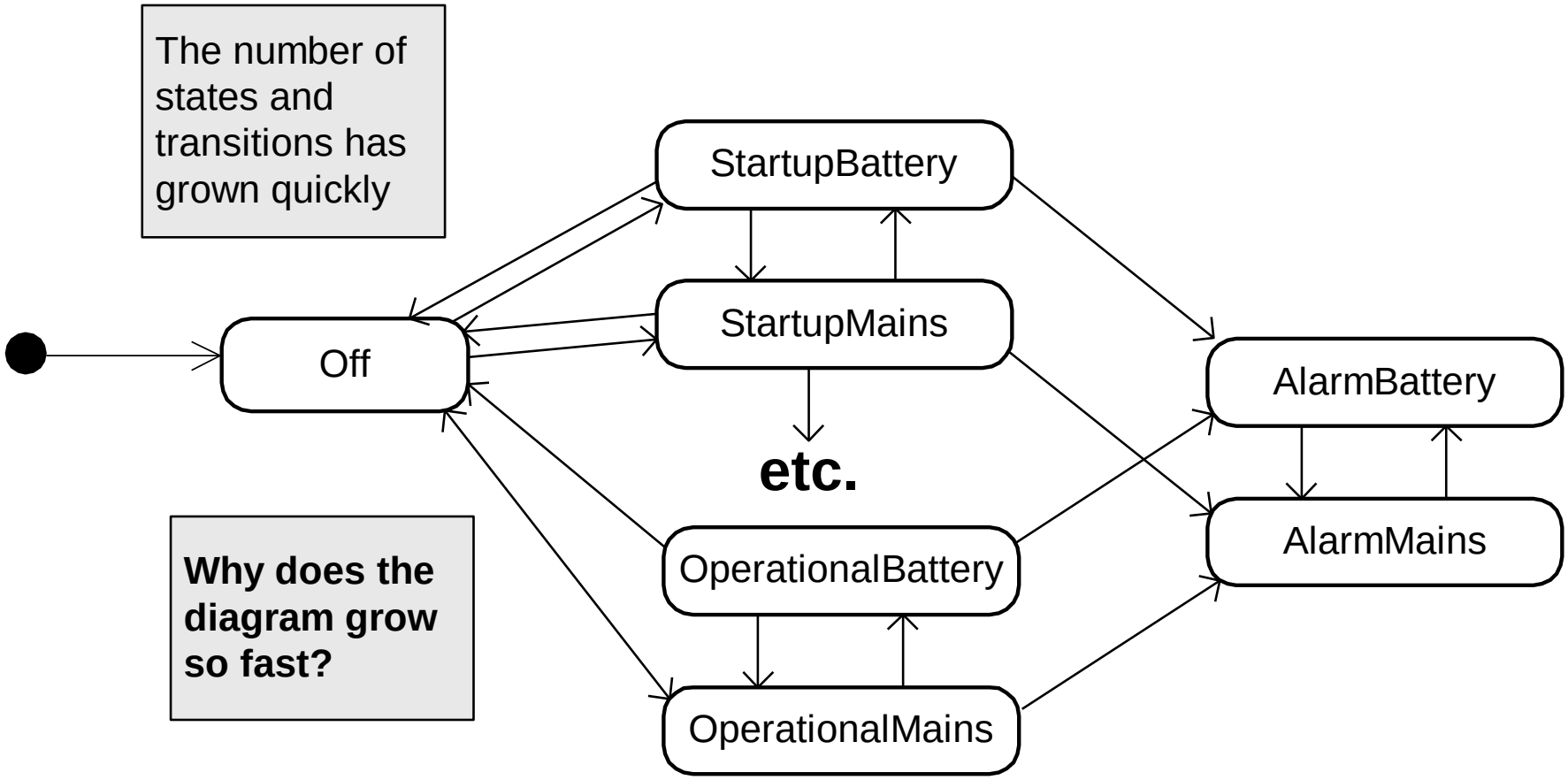
Predefined Action Labels

- **“entry/”**
  - identifies an action, specified by the corresponding action expression, which is performed upon entry to the state (entry action)
- **“exit/”**
  - identifies an action, specified by the corresponding action expression, that is performed upon exit from the state (exit action)
- **“do/”**
  - identifies an ongoing activity (“do activity”) that is performed as long as the modeled element is in the state or until the computation specified by the action expression is completed (the latter may result in a completion event being generated).
- **“include/”**
  - is used to identify a submachine invocation. The action expression contains the name of the submachine that is to be invoked.

Simple Diagram for Heart Monitor Applications

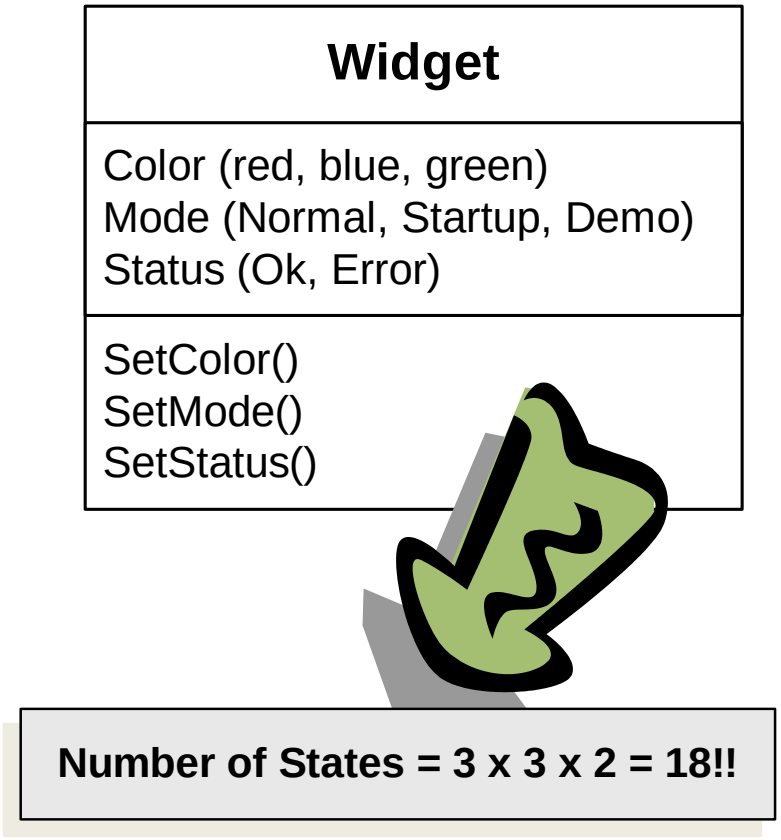


Full Diagram for Heart Monitor Applications



Independent State Components

- We can explore the phenomenon of rapid state diagram growth by examining a typical class with three attributes
- Each combination of values of the attributes of this class constitute a different state for the class
- Note that the attributes are effectively **independent** of each other
  - any combination is valid

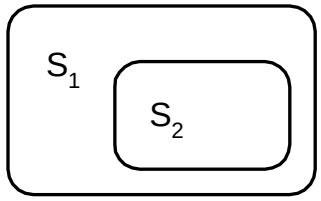


Combinatorial Explosion

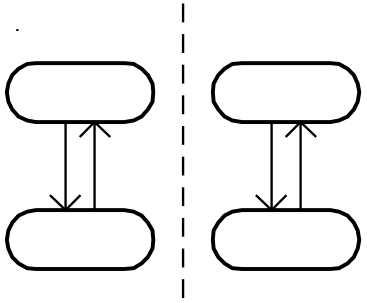
- The number of states and the number of transitions grows very quickly as you model complex state machines
  (→ **combinatorial explosion**)
- It has prevented the widespread use of state machines in the past

Features of Statecharts

- Harel statecharts directly address the problems of traditional state diagrams
- Two major features are introduced for controlling complexity and combinatorial explosion
  - **nested** state diagrams
  - **concurrent** state diagrams
- Many other features are also added
  - propagated transitions
  - broadcast messages
  - actions on state entry, exit
  - …



Nested State Diagrams



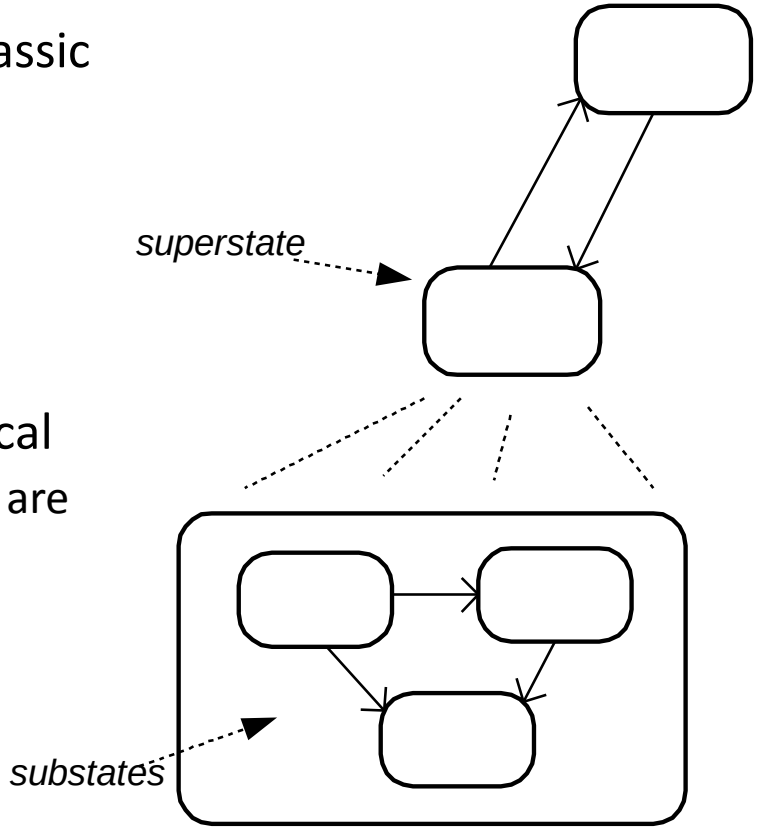
Concurrent State Diagrams

UML Statechart Diagrams

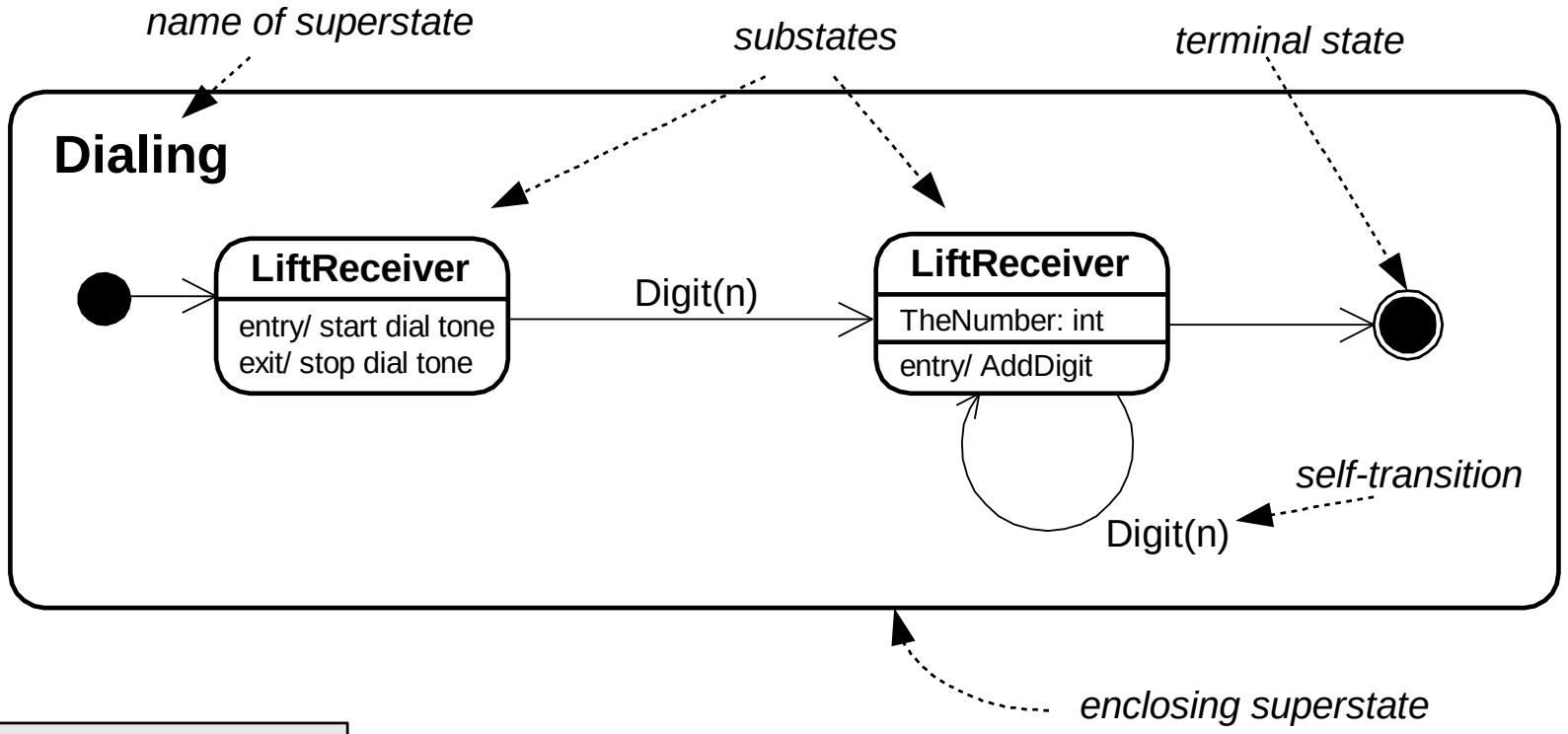
- Harel statecharts are the basis for the UML state diagram notation
  - formally “UML statechart diagrams” in the UML documentation
  - informally usually just referred to as “state diagrams”
- Original statechart notation was not object-oriented
  - based on traditional function-oriented, structured analysis paradigm
- UML added features to support object orientation
  - Support of inheritance (of events, transitions, etc.)
  - Delegation of messages within composite objects
  - Etc.

Nested State Diagrams

- Hierarchical organization is a classic way to control complexity
  - of programs
  - of documentation
  - of objects
  - …
- Even diagrams can be hierarchical
  - Traditional data flow diagrams are hierarchical
- Why not state diagrams?
  - superstates
  - substates



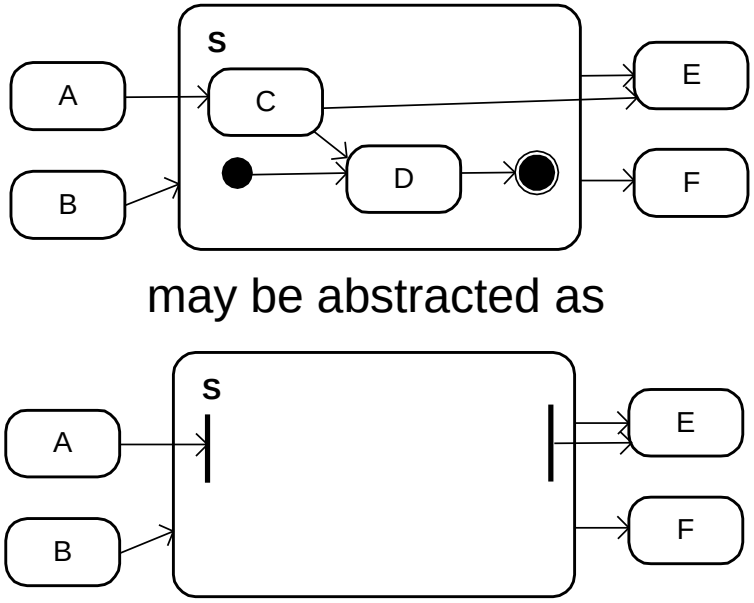
Superstates and Substates



State model for telephone dialing

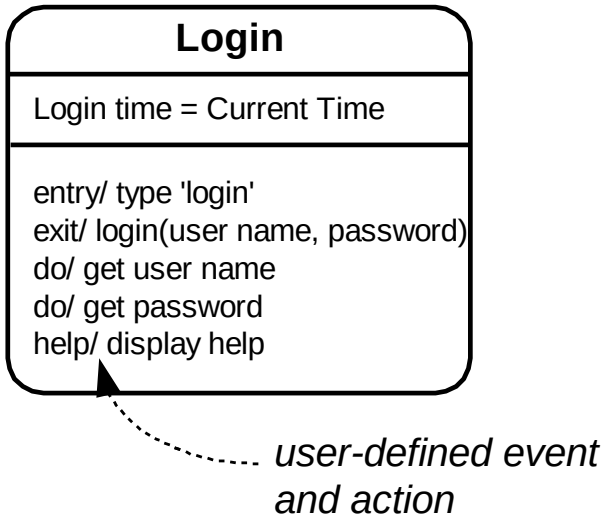
Stubbed Transitions

- Nested states may be suppressed
- Subsumed transitions that do not come from an unlabeled final state or go to an unlabeled initial pseudostate may (but need not) be shown as coming from or going to *stubs*
- A *stub* is shown as a small vertical line (bar) drawn inside the boundary of the enclosing state



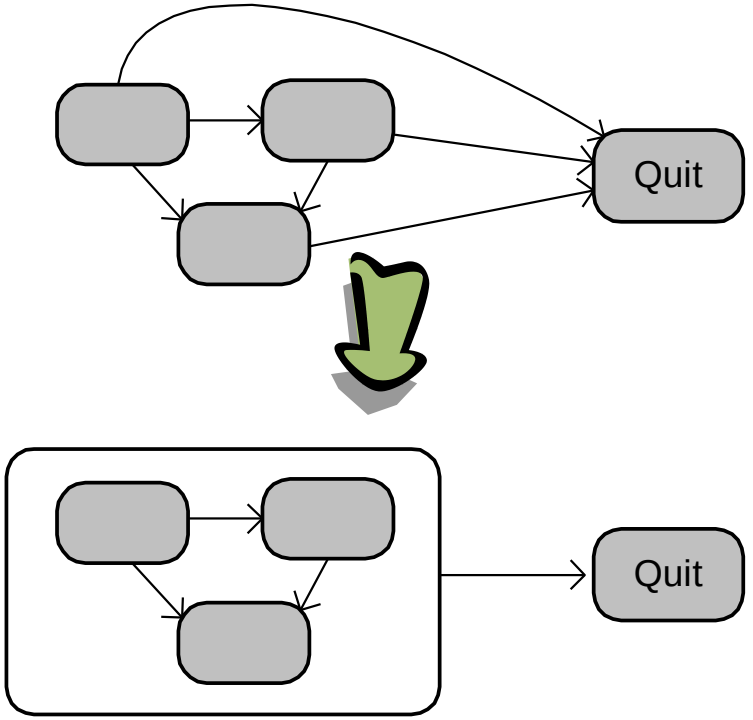
Some More UML State Diagram Notation

- Nested states are show by an enclosing rectangle
  - The name of the superstate is indicated in the upper left-hand corner
- A middle compartment can contain **state variables** (if any)
- **Entry, exit, do** actions may be defined
  - 'entry', 'exit', 'do' are reserved words
- **Self-transitions** may occur
  - from the state back to itself
  - the entry and exit conditions are performed in that case

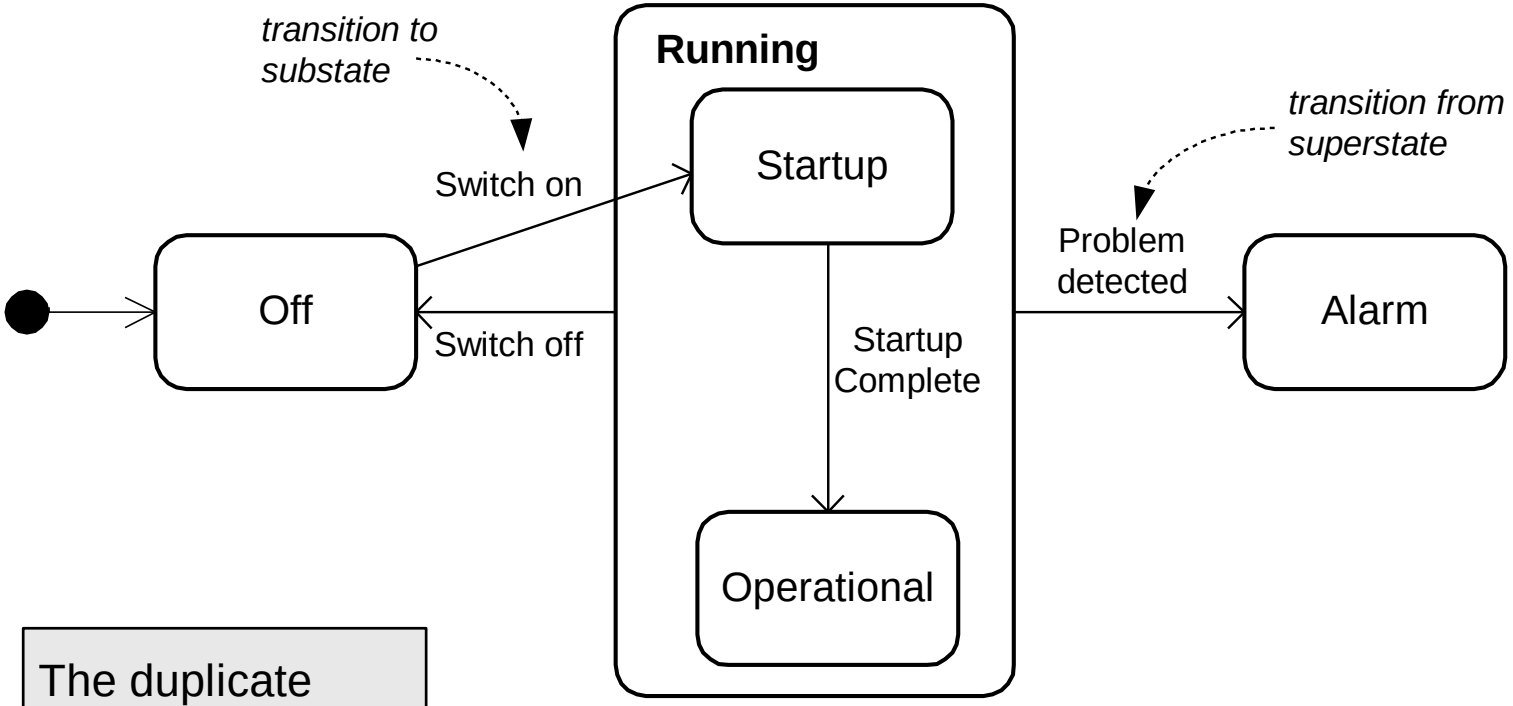


Eliminating Duplicated Transitions

- Transitions are often duplicated when there is some transition that can happen from every state
  - "error"
  - "quit"
  - "abort"
- These duplicates can be combined into a single transition
  - a transition from a superstate is valid for all of its substates!



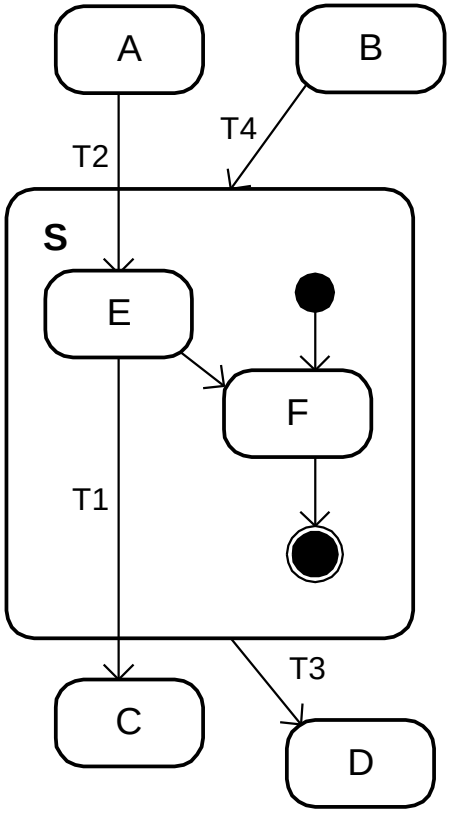
The Heart Monitor Revisited



The duplicate transitios have been elimiated!

Transitions to and from Nested States

- Transitions can be **specific**
  - A transition can be from a specific substate (**T1**)
  - A transition can be to a specific substate inside the nested state (**T2**)
- Transitions can be **general**
  - We saw that a transition from the superstate is valid for all substates (**T3**)
  - A transition into the superstate (**T4**) normally goes to the default initial state (start state leading to F)

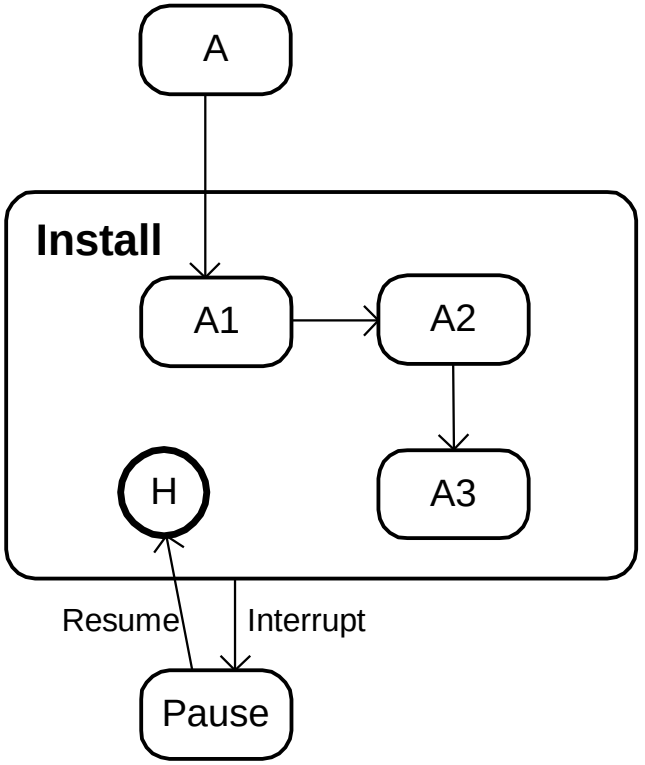


Handling Interrupted States

- Sometimes it is necessary to interrupt a complicated procedure and then resume where you stopped
- Consider an “install” program on your computer
  - “Out of memory: please delete some files and hit continue”
- Any situation in which a normal course of events is **interrupted** and then **resumed**
  - How can this be handled in a nested state diagram?

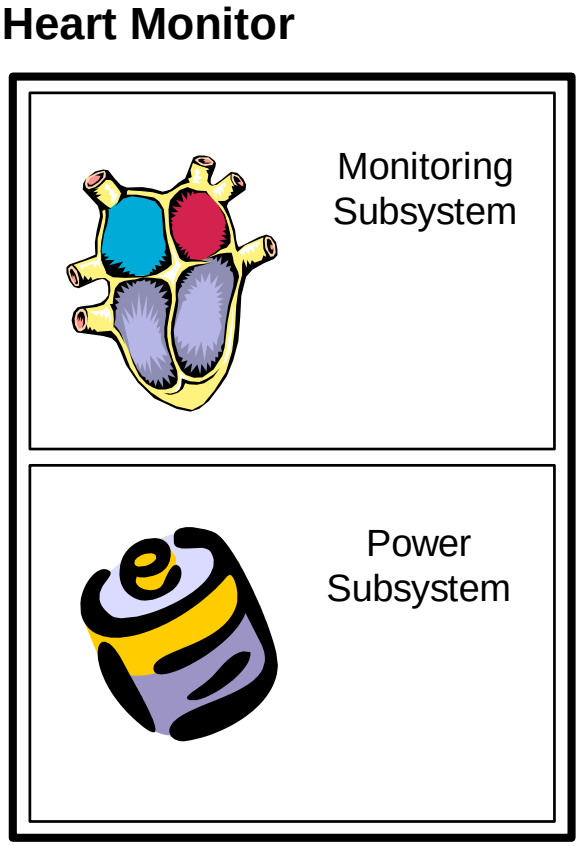
The State History Indicator

- Suppose an *interrupt* transition occurs from any one of the substates (A1, A2, A3) out of the *Install* superstate
- The rescue transition takes it back into the *Install* superstate
- The **history indicator** is shown with an "H" inside a circle
  - it means "return to the last state you were in before leaving"

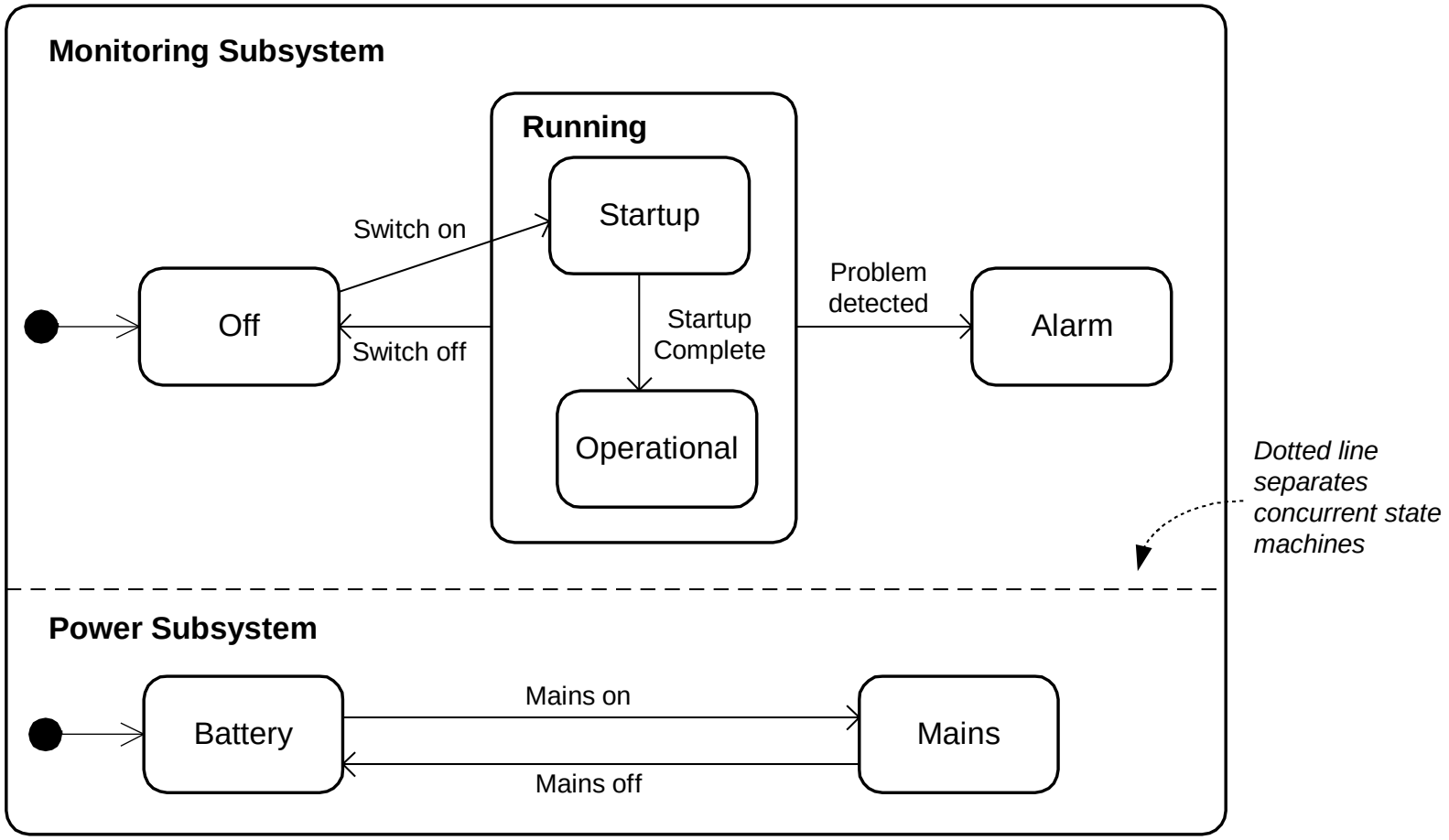


Concurrent State Machines

- Complex systems usually have concurrency
  - "subsystems" that operate (mostly) independently
- Back to the heart monitor device
  - The power supply and the heart monitoring application are really concurrent subsystems
  - *They should be modeled that way!!*
  - They are mostly independent: the monitoring application doesn't care where it gets its power

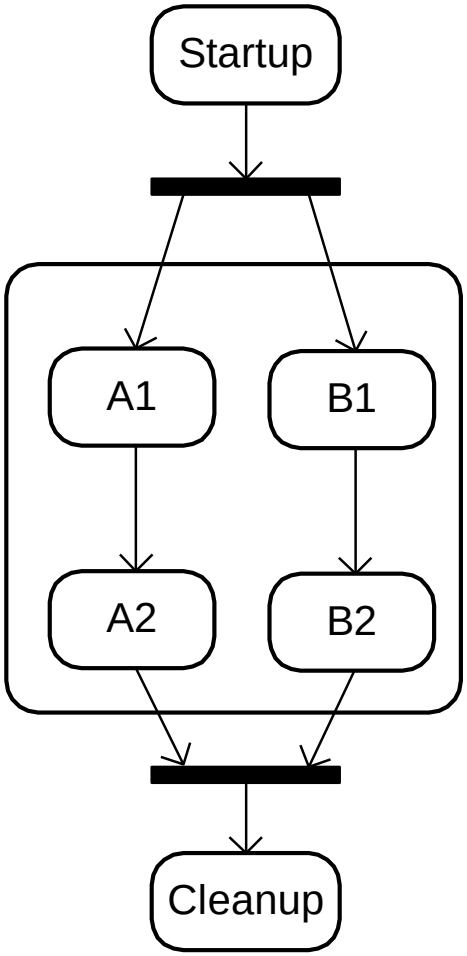


Heart Monitor as Concurrent State Machine



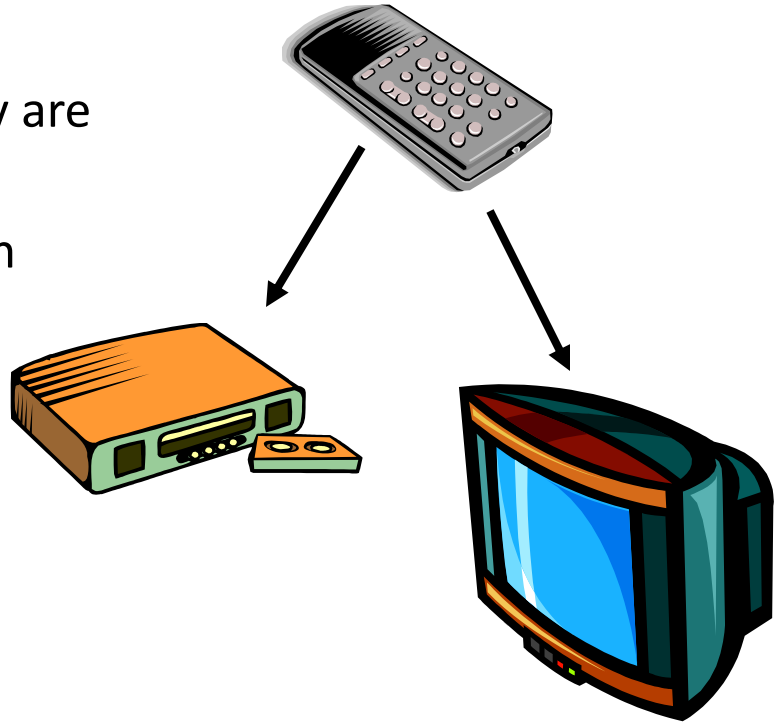
Complex Transitions

- Concurrent state machines sometimes need to be synchronized with each other
  - start together
  - run through independently until their terminal state
  - re-sychronize at the end
- **Complex transitions** allow this kind of synchronization
  - Petri net notation

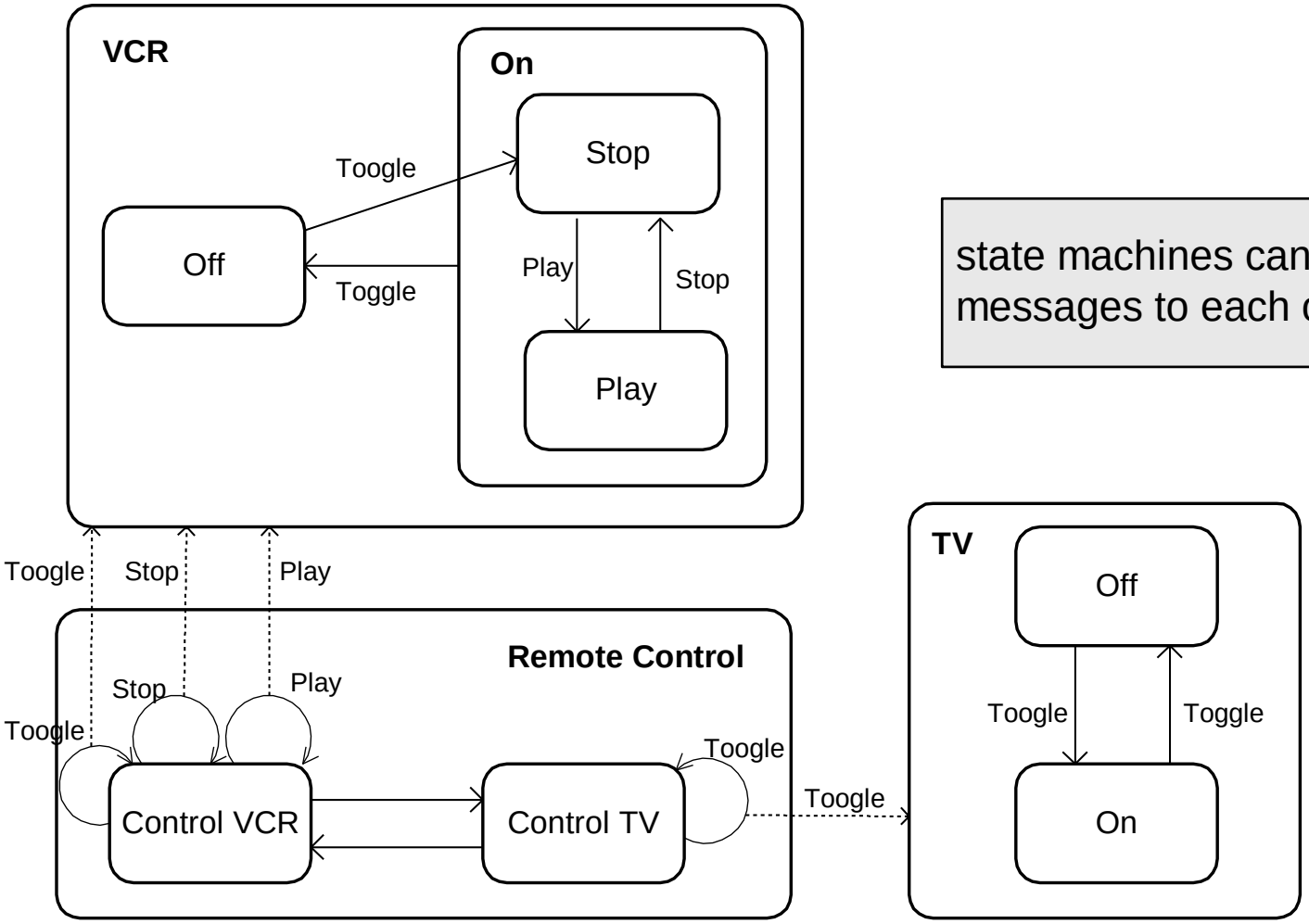


Cooperating Independent State Machines

- Concurrent and independent state machines will normally have to **communicate** with each other if they are cooperating
- Example: remote control of television and VCR
  - three independent systems
  - three independent state machines
  - but clearly they cooperate with each other
- How to model this in UML?



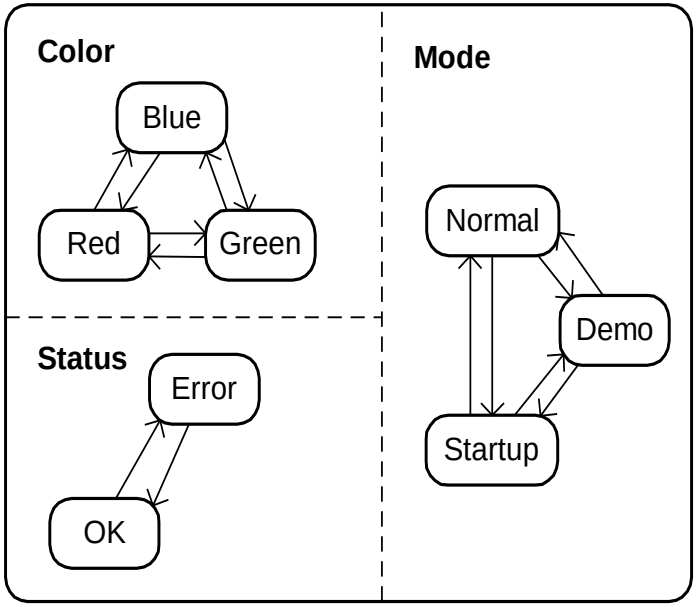
Sending Messages Between State Diagrams



state machines can send messages to each othe

Orthogonal Components

- The concurrent state machine notation can control **combinatorial of states**
- The three components of our widget class (color, mode, status) are orthogonal
  - they change states independently
- Their orthogonality can be modeled explicitly with statechart notation



Concurrency and Orthogonality

- The heart monitor is an example of two concurrent subobjects
  - the internal battery and the heart monitoring application are two (almost) independent FSMs
  - they exhibit true concurrency
- The widget is an example of an object with orthogonal components
  - it is a single object with a complex specification logic
  - there are not concurrent threads of execution
- The are NOT the same thing
  - but the UML notation is useful for handling the complexity introduced by each of them

When to Use State Diagrams

- Interaction diagrams (collaboration & sequence) are most appropriate for describing **inter-object behavior**
- State diagrams are particularly intended for describing **intra-object behavior**
  - precise description of behavior inside a single class with complex behavior
- But there are sometimes exceptions
  - the remote control example showed that state diagrams are sometimes useful to model interaction among objects
  - they are sometimes useful to describe complicated behavior inside a use case

Domain Model Refinement

Larman, chapter 31

When to make generalization hierarchies? Why is the following a good example?

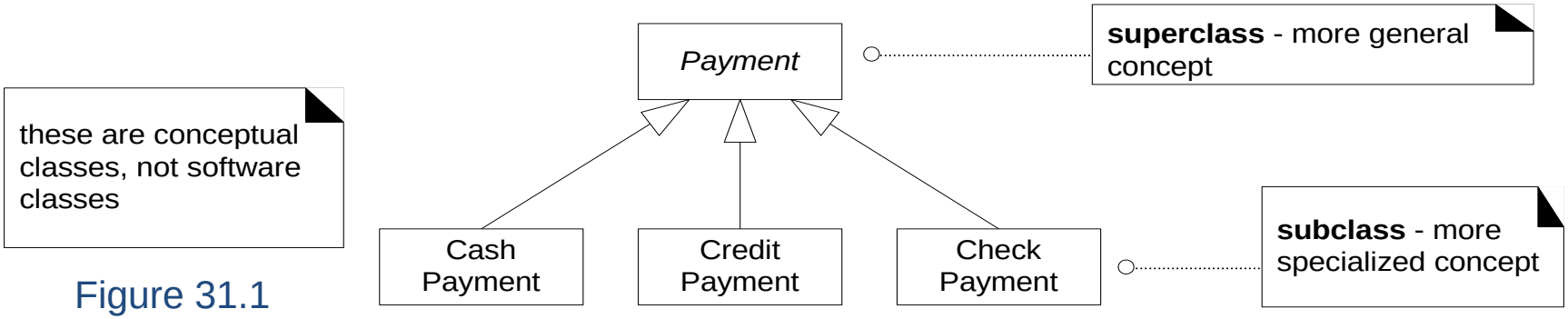
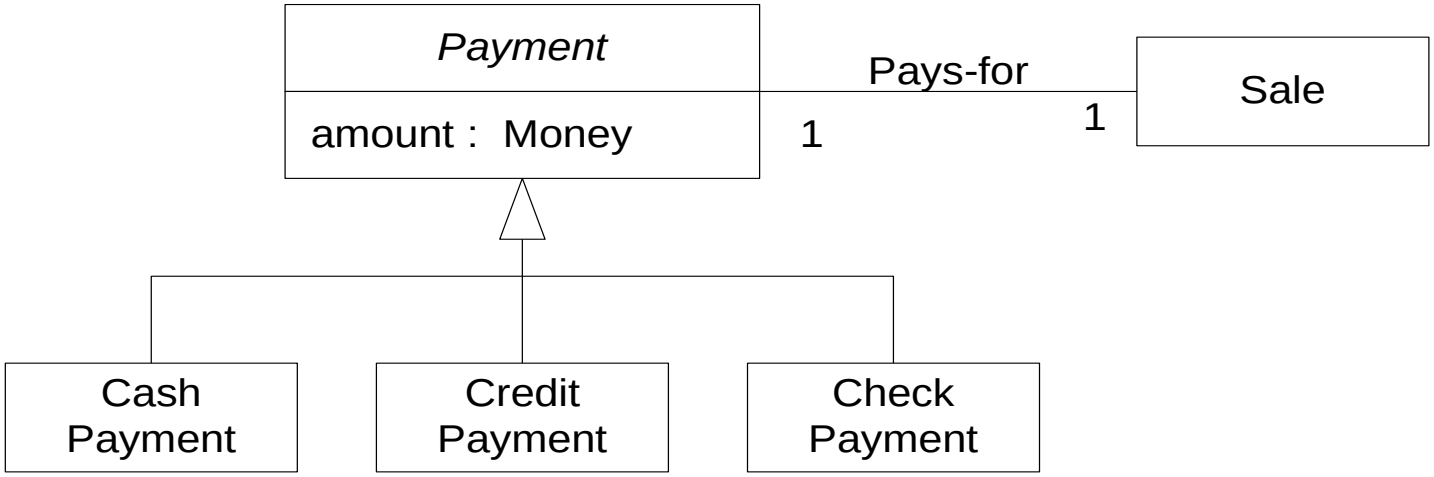


Figure 31.1

Guidelines:

- Identify superclasses and subclasses when they help us understand concepts in more general, abstract terms and reduce repeated information.
- Expand class hierarchy relevant to the current iteration and show them in the Domain Model.

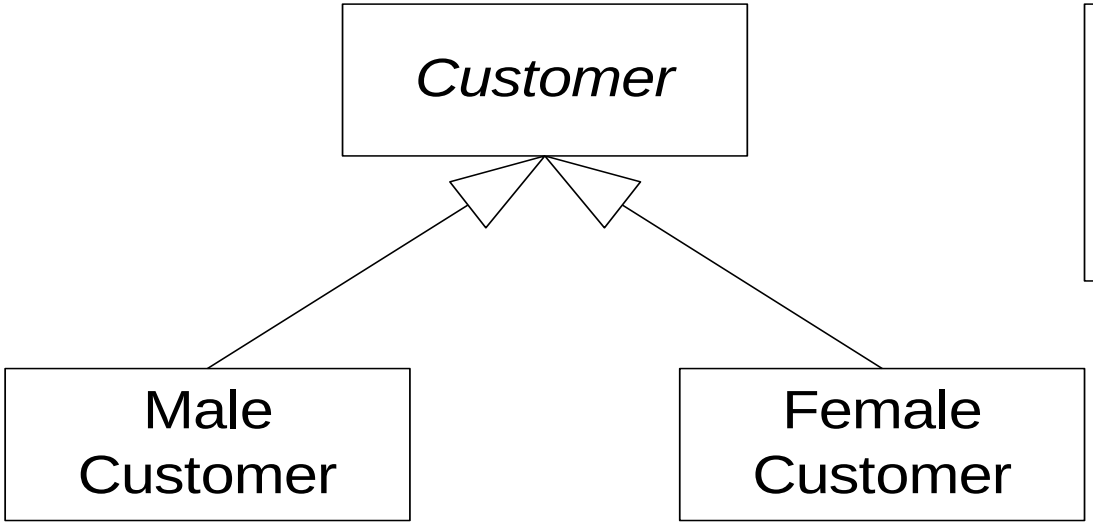
100% and Is-a rules



100% rule: Subclasses must conform to 100% of superclass's attributes and associations. Do they above?

Is-a rule (informal test): "Subclass is a Superclass"

What about this hierarchy?



Correct subclasses.

But useful?

Fig. 31.6

Guidelines for creating conceptual subclasses:

- Subclass has additional attributes or associations of interest
- Subclass behaves or is operated on, or handled or manipulated differently than superclass or other subclasses

Is this hierarchy OK? What does it add to our understanding of the domain?

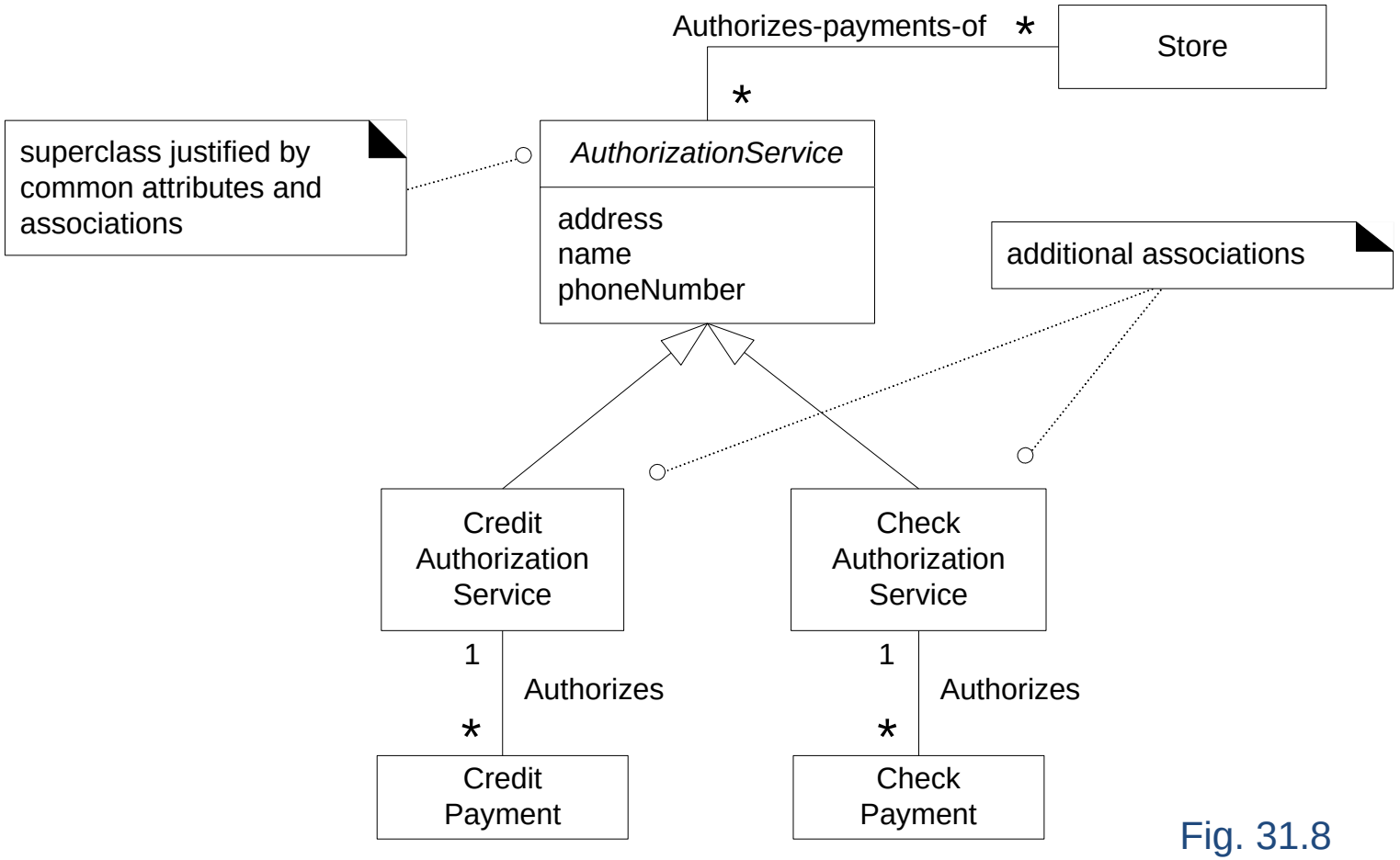


Fig. 31.8

Models transactions with external services which are important in POS domain

When to design Association classes?

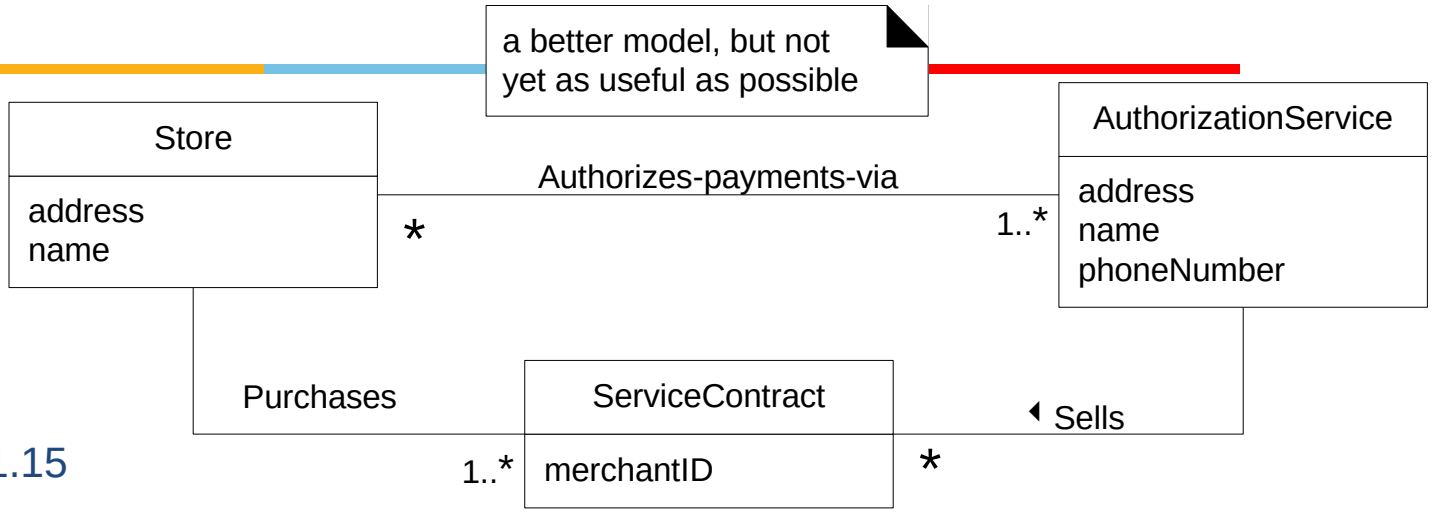


Fig. 31.15

ServiceContract records merchantID—good.
But ServiceContract is dependent on relationship between two classes

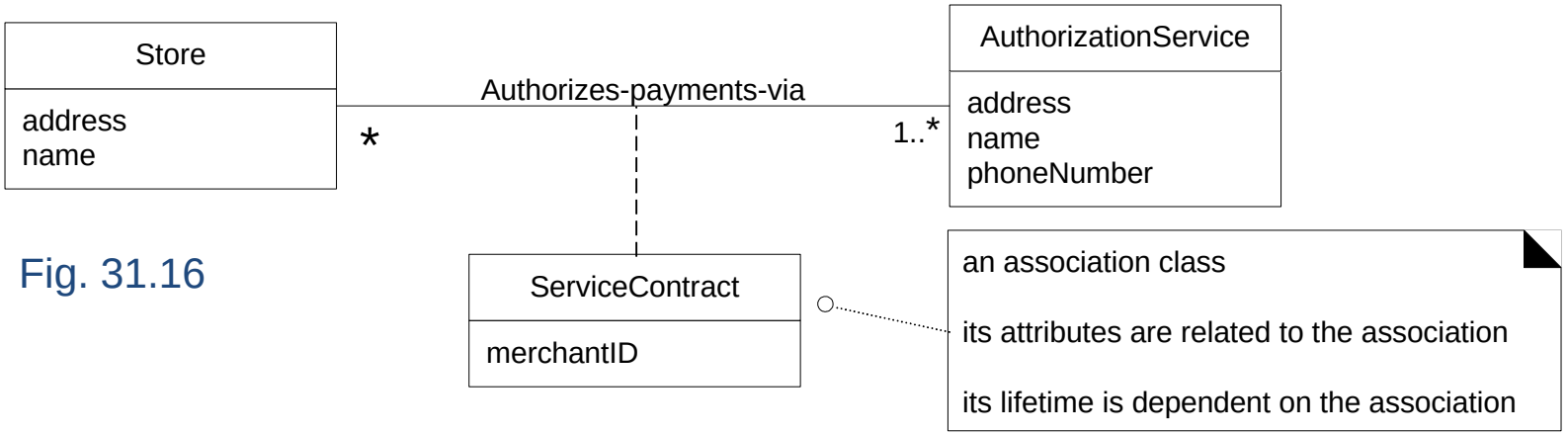


Fig. 31.16

What about this hierarchy? (Why not?)

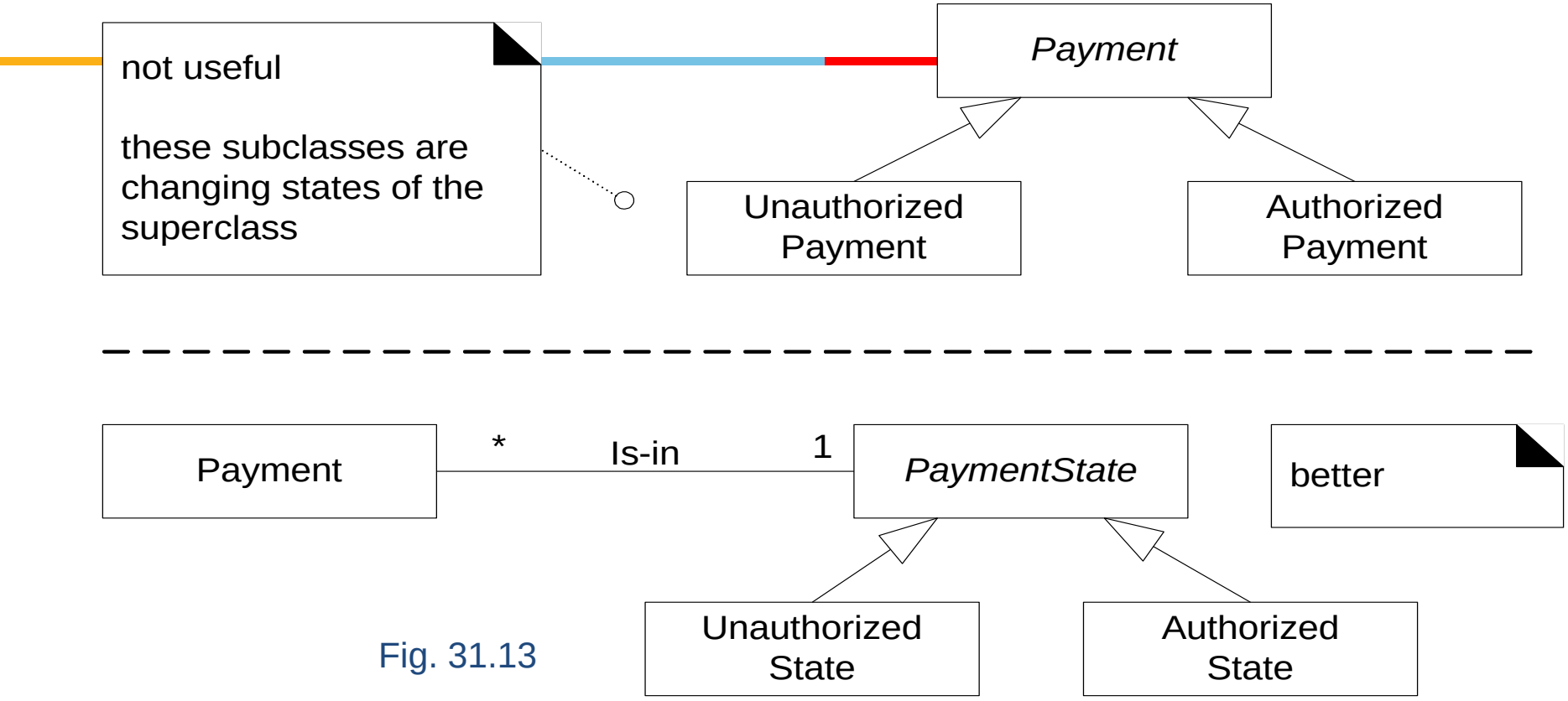


Fig. 31.13

- Classes should be invariant
- Classes can maintain state information as attributes
- Rather than subclasses, model changing states with state charts

More Association classes

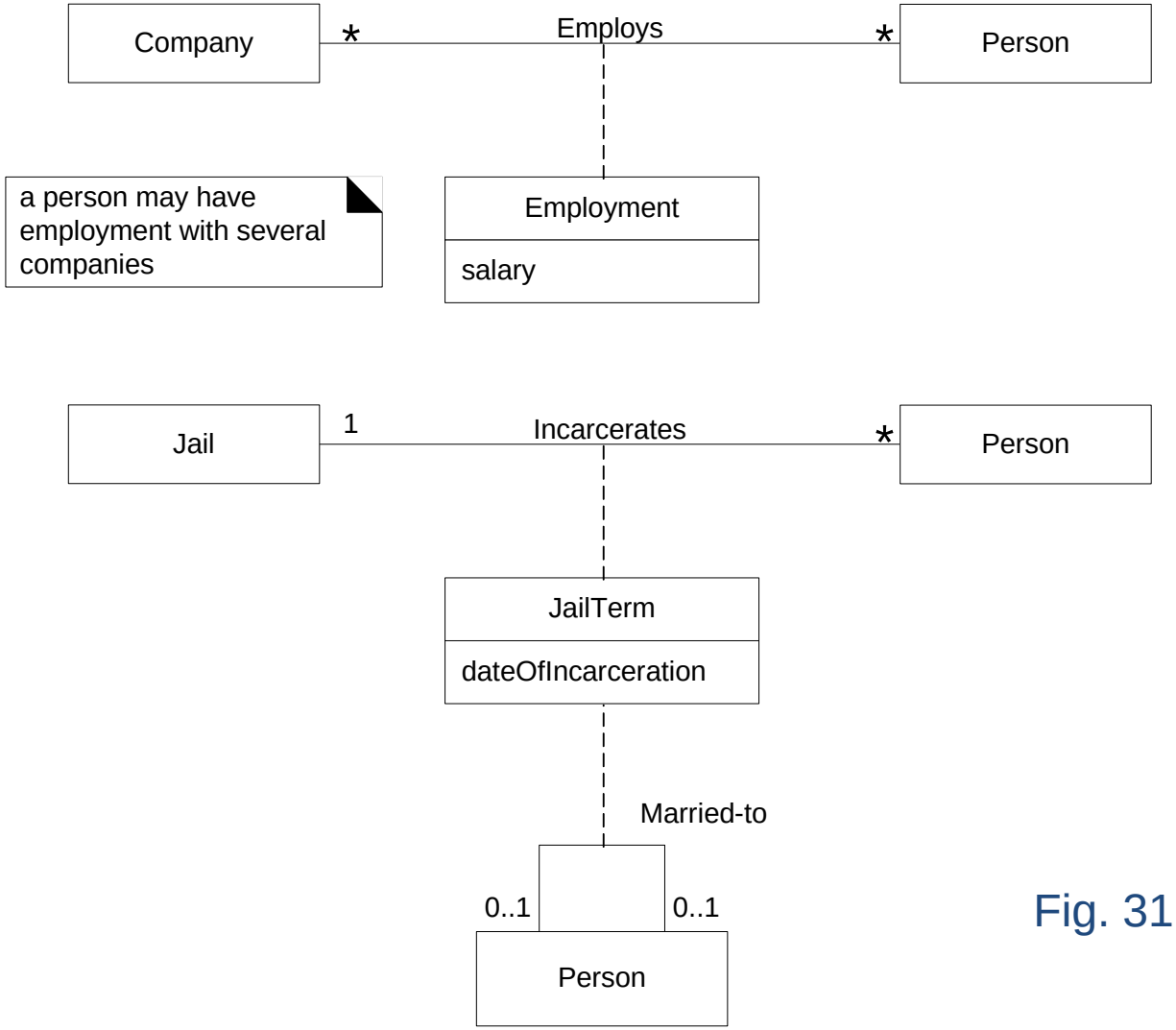
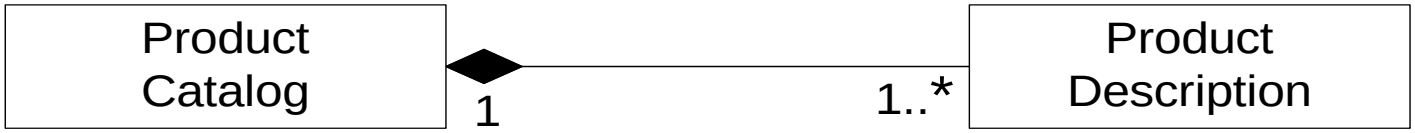


Fig. 31.17

When to add Composition notation?

Fig. 31.18



Guidelines for drawing Composition (whole-part) relationships:

- Obvious whole-part physical or logical assembly
- Lifetime of part is bound within lifetime of whole (create-delete dependency)
- Some properties of whole propagate to parts, e.g., location
- Operations of whole propagate to parts, e.g., movement
- But: If in doubt, leave it out.

UML package diagrams

- UML packages for higher level structure than classes
- Denoted by box with smaller box on top
  - Note dependency arrows
  - A dependency indicates that changes to one element may cause changes to the other
- Guidelines for partitioning the domain model into packages:
  - Place elements together if they are in same class hierarchy, participate in the same use cases, or closely related by concept or purpose, or strongly associated
- Packages can be grouped in higher-order packages
  - Packages may include packages
  - Common package as <<global>> means all packages in system have dependency to this one
  - General package marked {abstract} means this package is an interface, with subtypes
- Guidelines: divide classes into packages; analyze dependencies; refactor to reduce dependencies

Higher order package for POS domain

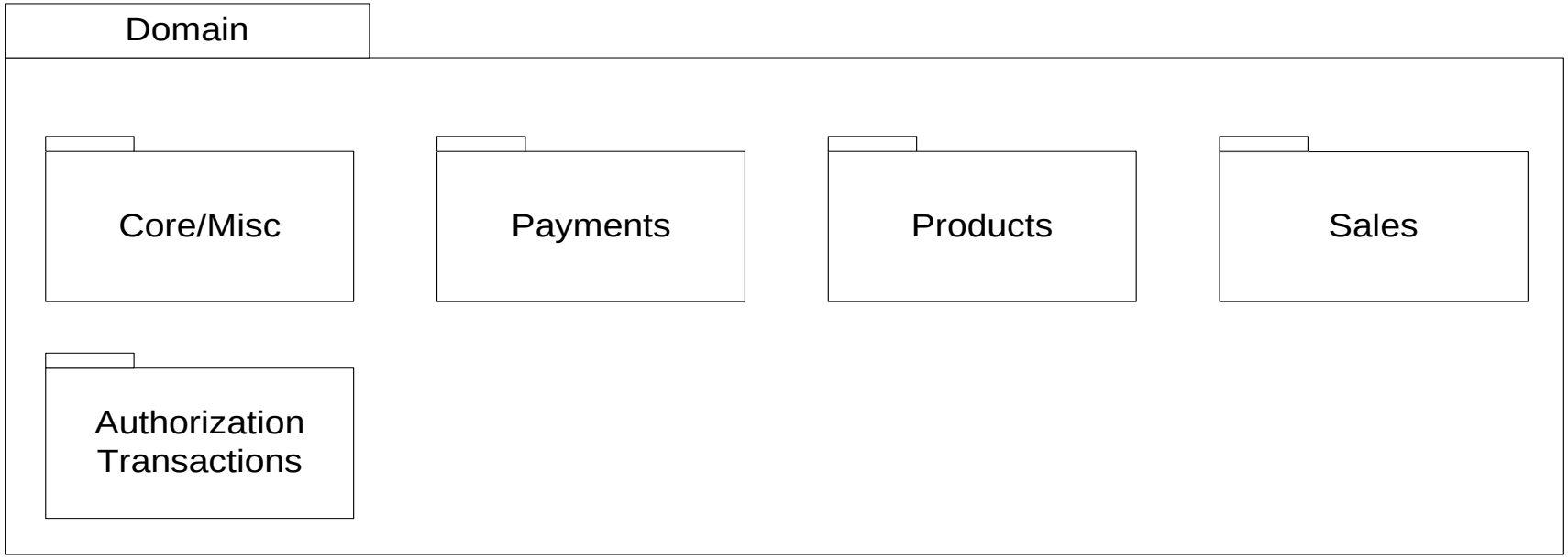


Fig. 31.29

Core/Misc package

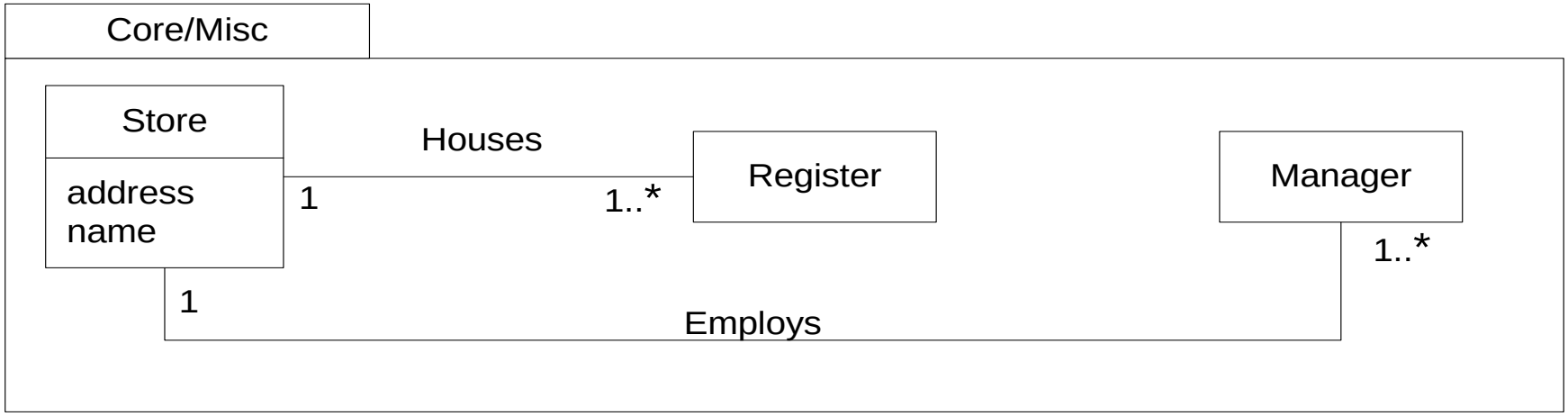


Fig. 31.30

Why call this package Core for the POS domain?

A rich package

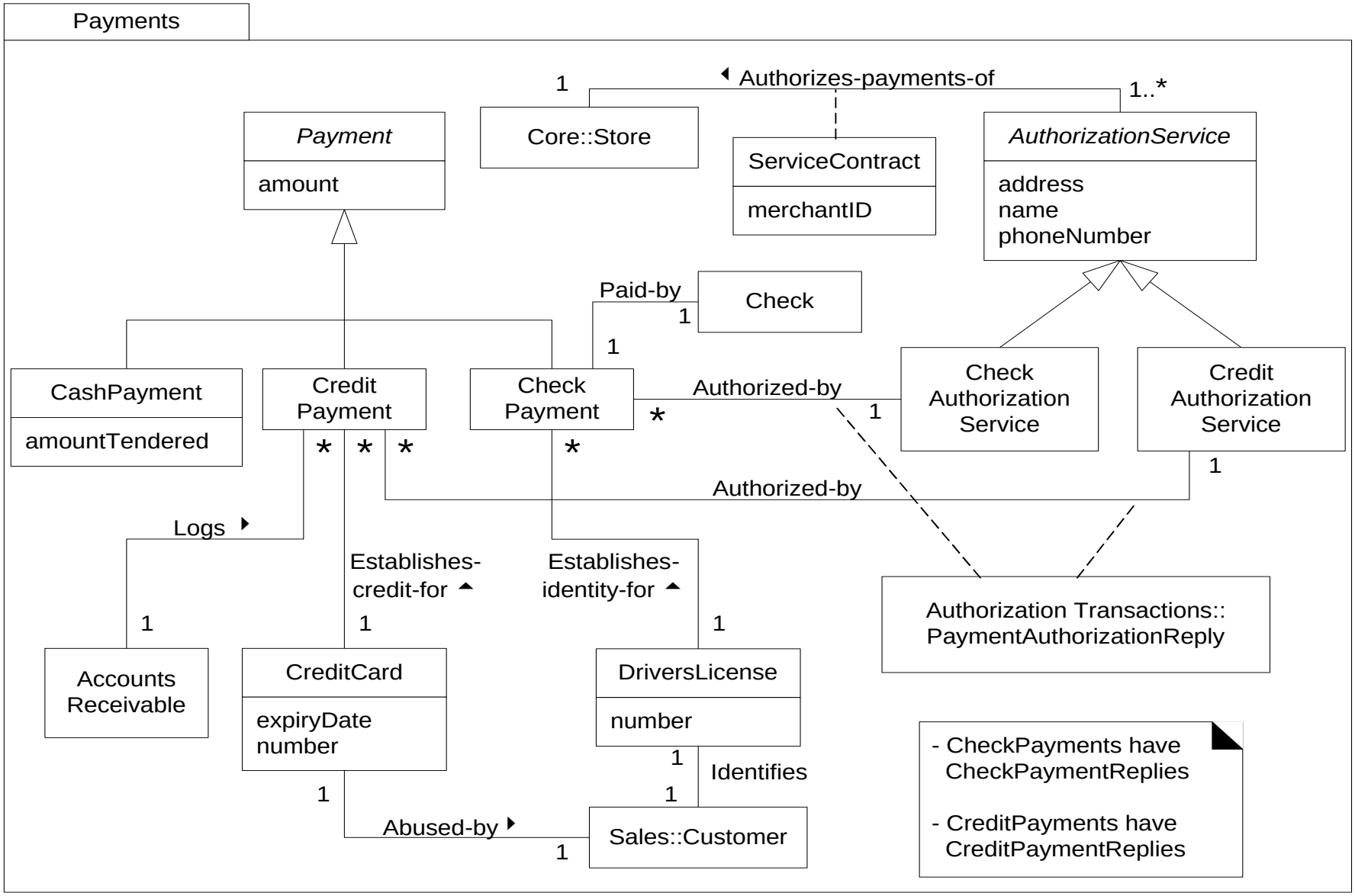


Fig. 31.31

Composition and tie to Core package

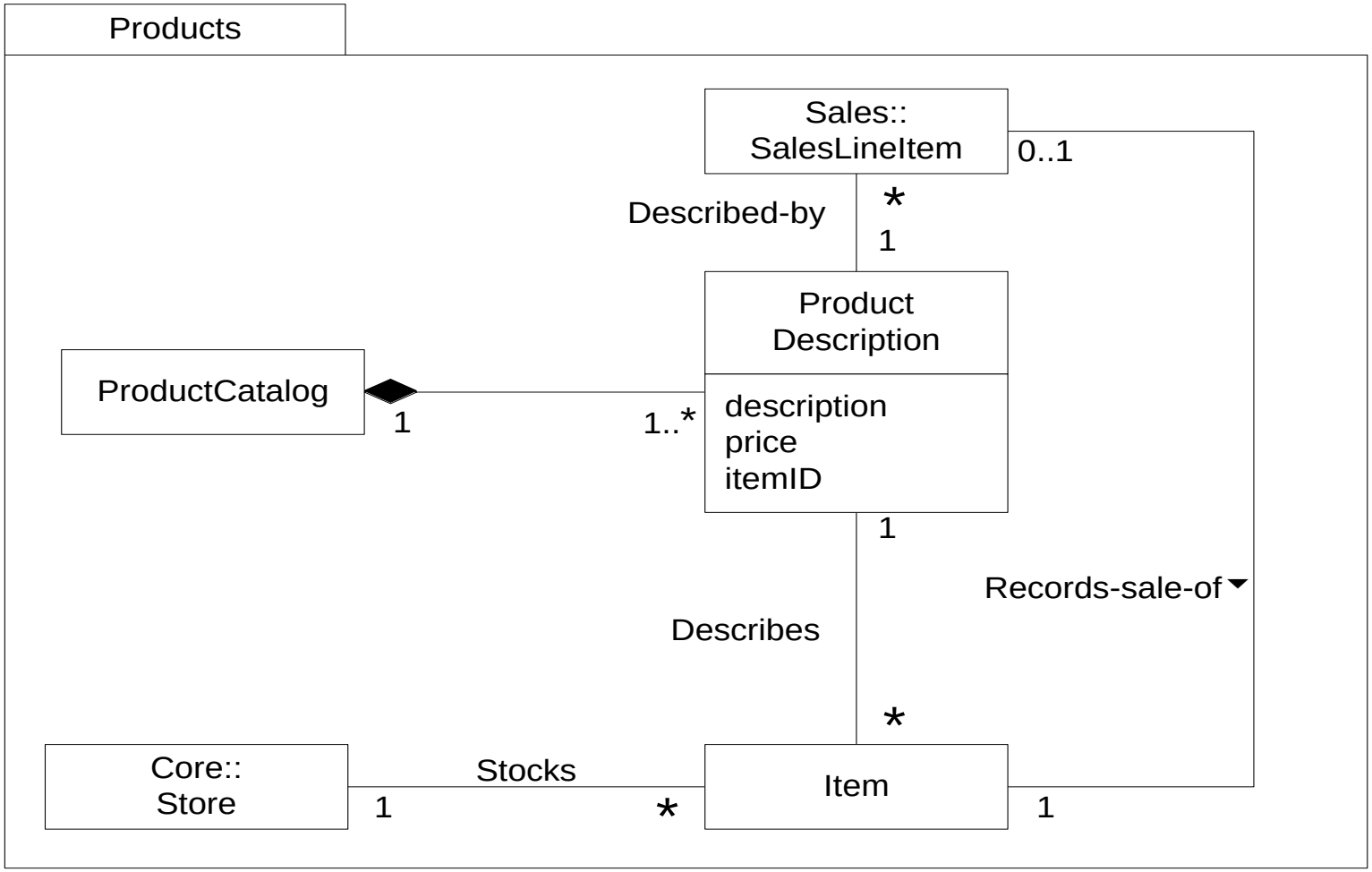


Fig. 31.32

Thank You