Chapter 29

UML State Machine Diagrams and Modeling

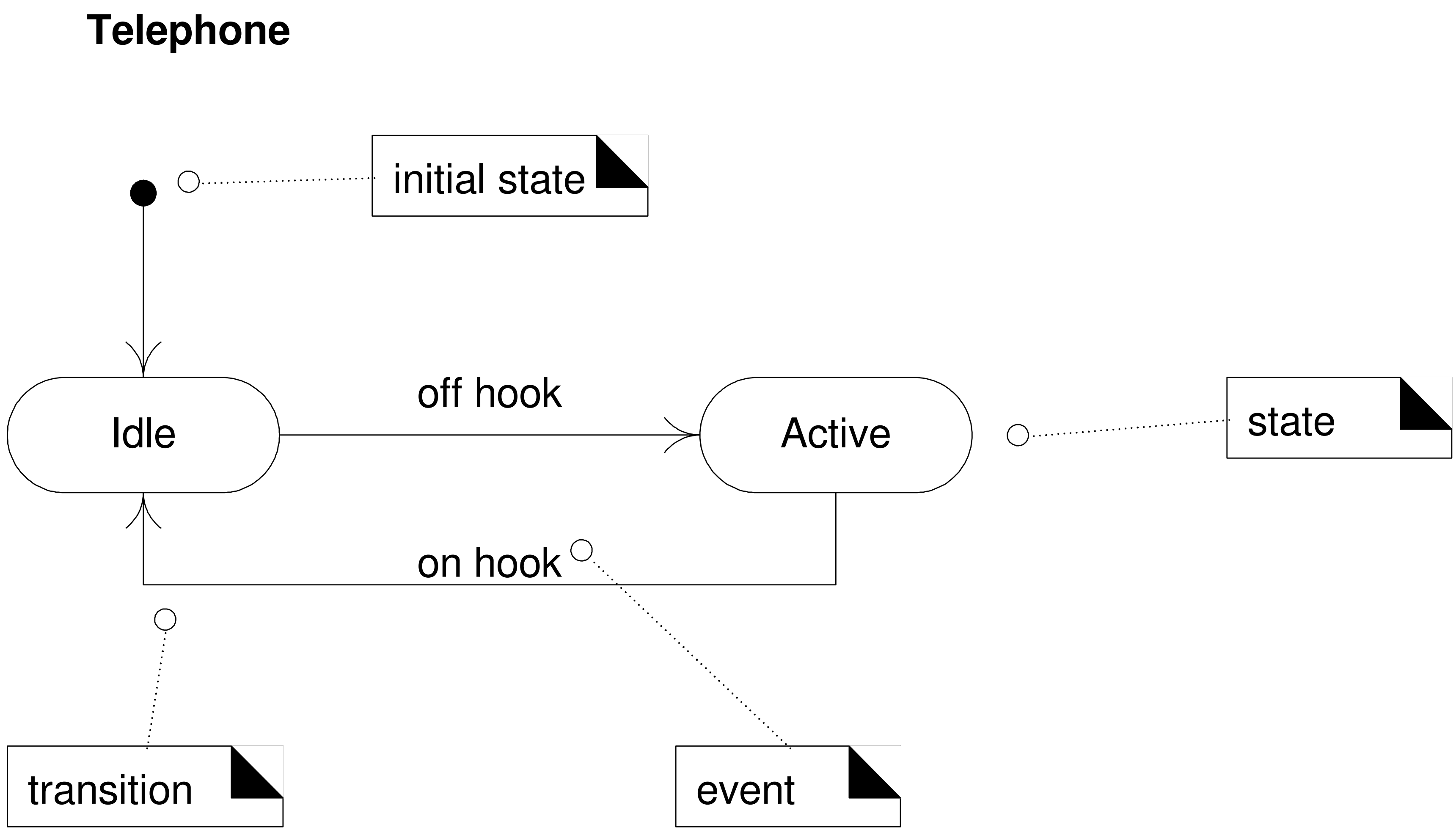
State Machine Diagram

- Illustrates the interesting events and states of an object and the behavior of an object in reaction to an event.
  - Event: significant or noteworthy occurrence.
    - E.g., telephone receiver taken off hook.
  - State: the condition of an object at a moment in time (between events).
  - Transition: a relationship between two states; when an event occurs, the object moves from the current state to a related state.

UML State Machine Diagram

- States shown as rounded rectangles.
- Transitions shown as arrows.
- Events shown as labels on transition arrows.
- Initial pseudo-state automatically transitions to a particular state on object instantiation.
- Events with no corresponding transitions are ignored.

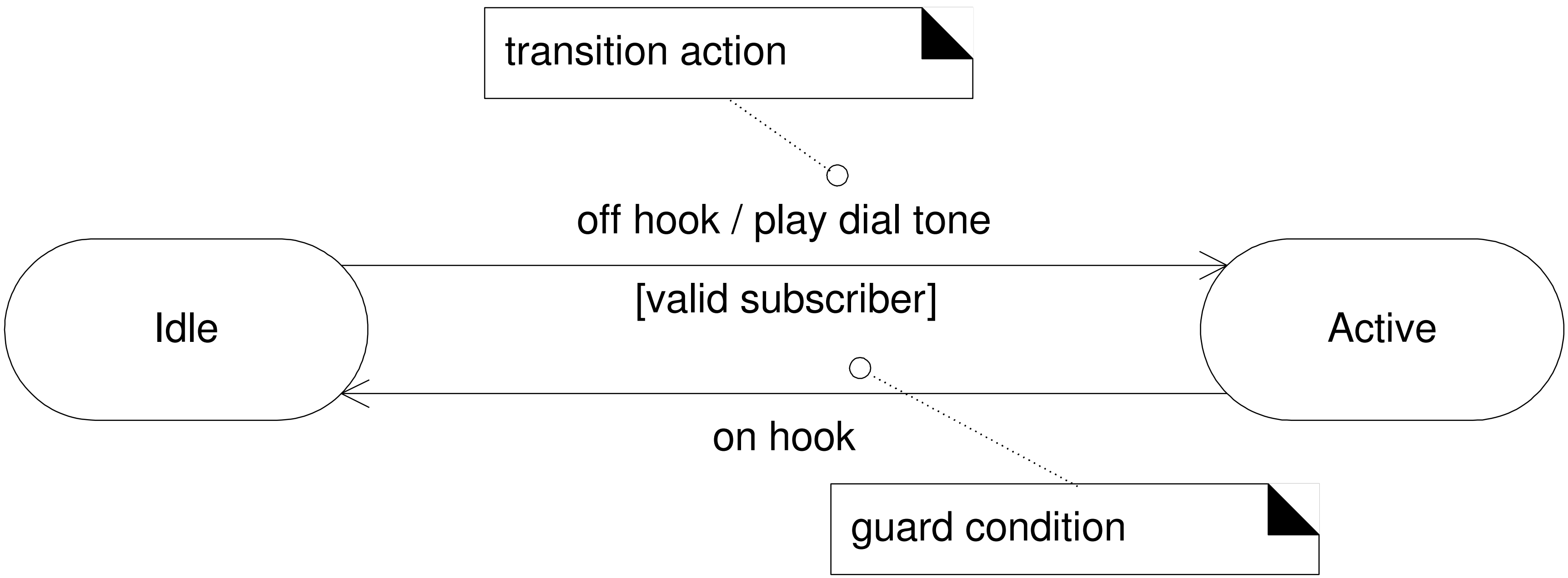
Fig. 29.1 State machine diagram for a telephone



Transition Actions and Guards

- A transition can cause an action to fire.
  - In software implementation, a method of the class of the state machine is invoked.
- A transition may have a conditional guard.
  - The transition occurs only if the test passes.

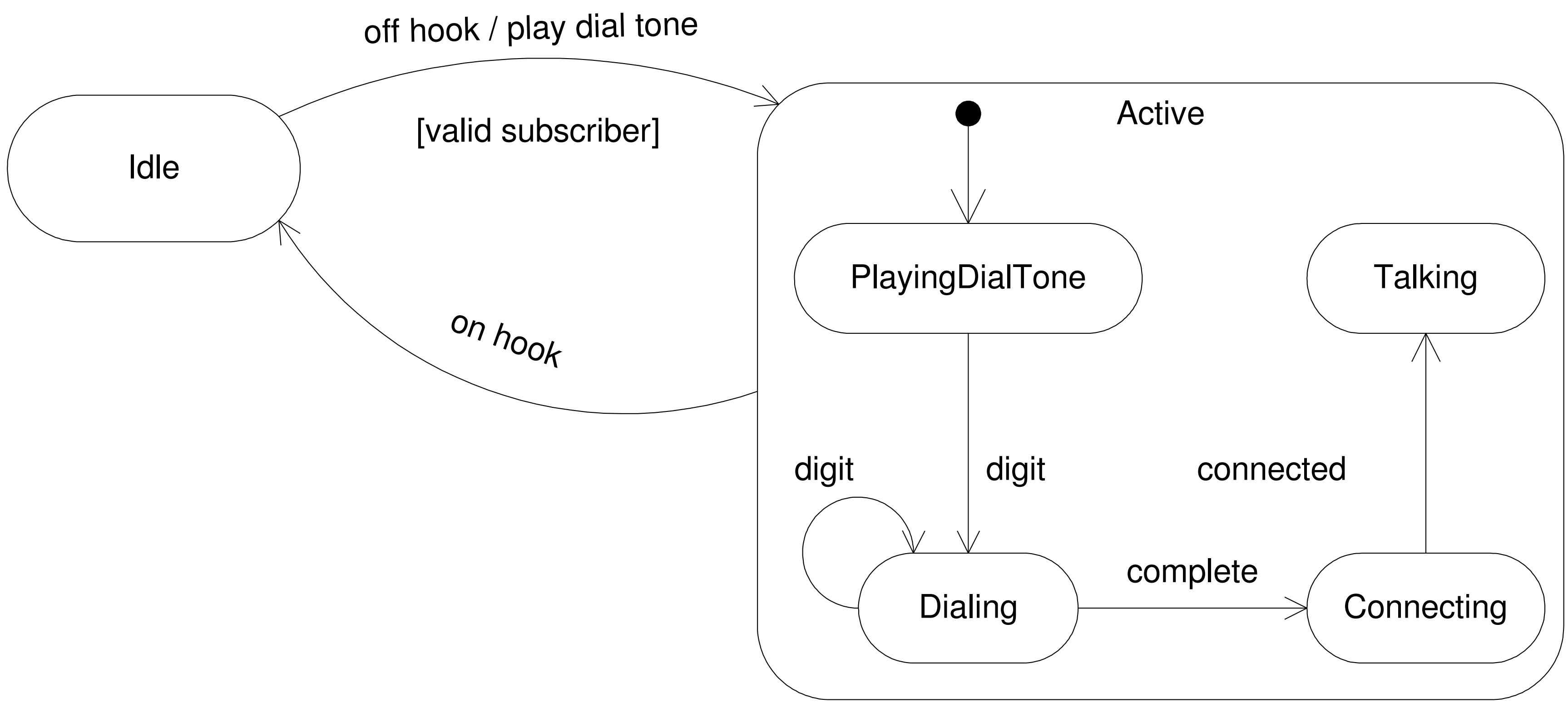
Fig. 29.2 Transition action and guard notation



Nested States

- A state may be represented as nested substates.
  - In UML, substates are shown by nesting them in a superstate box.
- A substate inherits the transitions of its superstate.
  - Allows succinct state machine diagrams.

Fig. 29.3 Nested states



State-Independent vs. State-Dependent

- State-independent (modeless) — type of object that always responds the same way to an event.
- State-dependent (modal) — type of object that reacts differently to events depending on its state or mode.

  Use state machine diagrams for modeling state-dependent objects with complex behavior, or to model legal sequences of operations.

Modeling State-dependent Objects

- Complex reactive objects
  - Physical devices controlled by software
    - E.g., phone, microwave oven, thermostat
  - Transactions and related business objects
- Protocols and legal sequences
  - Communication protocols (e.g., TCP)
  - UI page/window flow or navigation
  - UI flow controllers or sessions
  - Use case system operations

Fig. 29.4 Web page navigation modeling

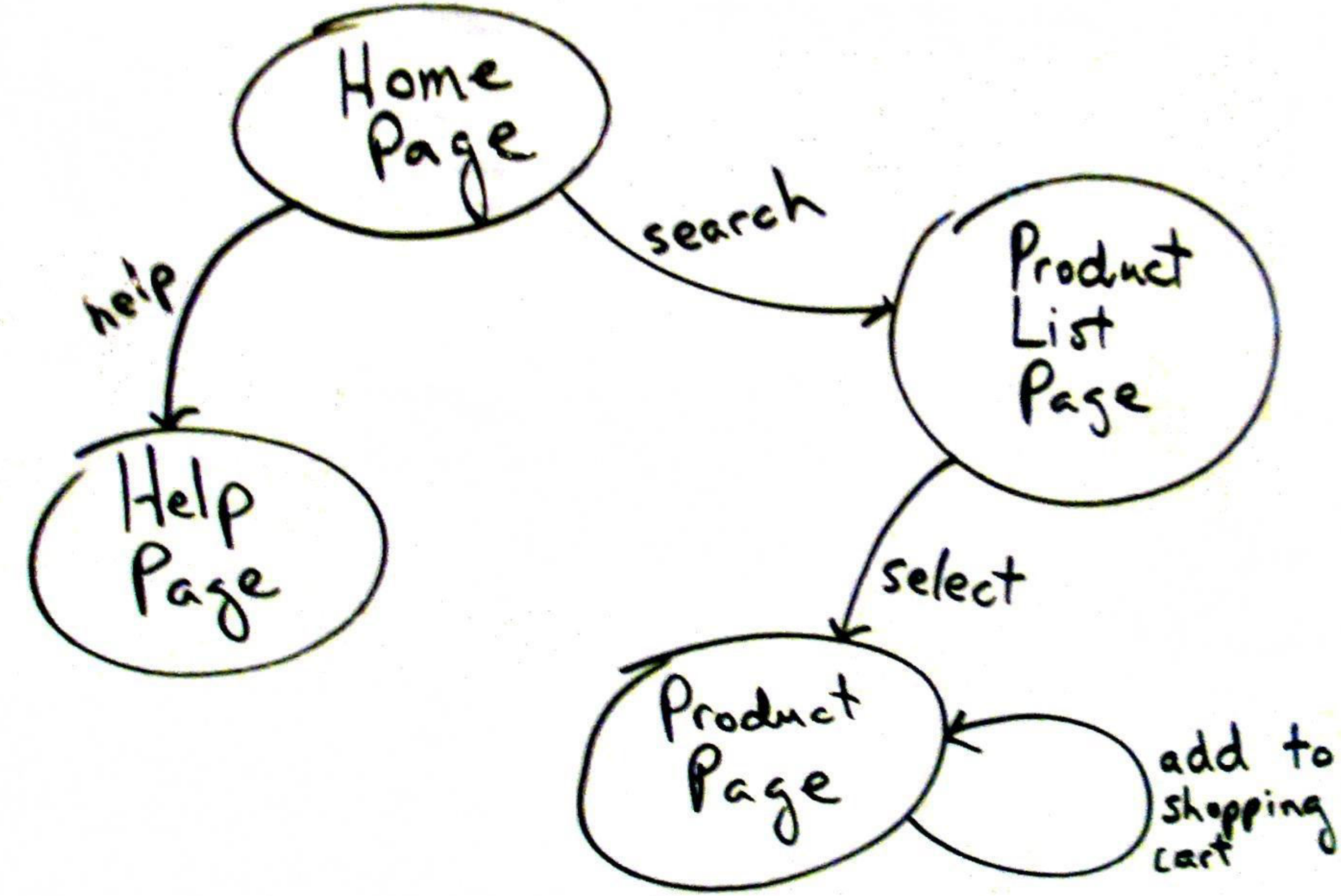
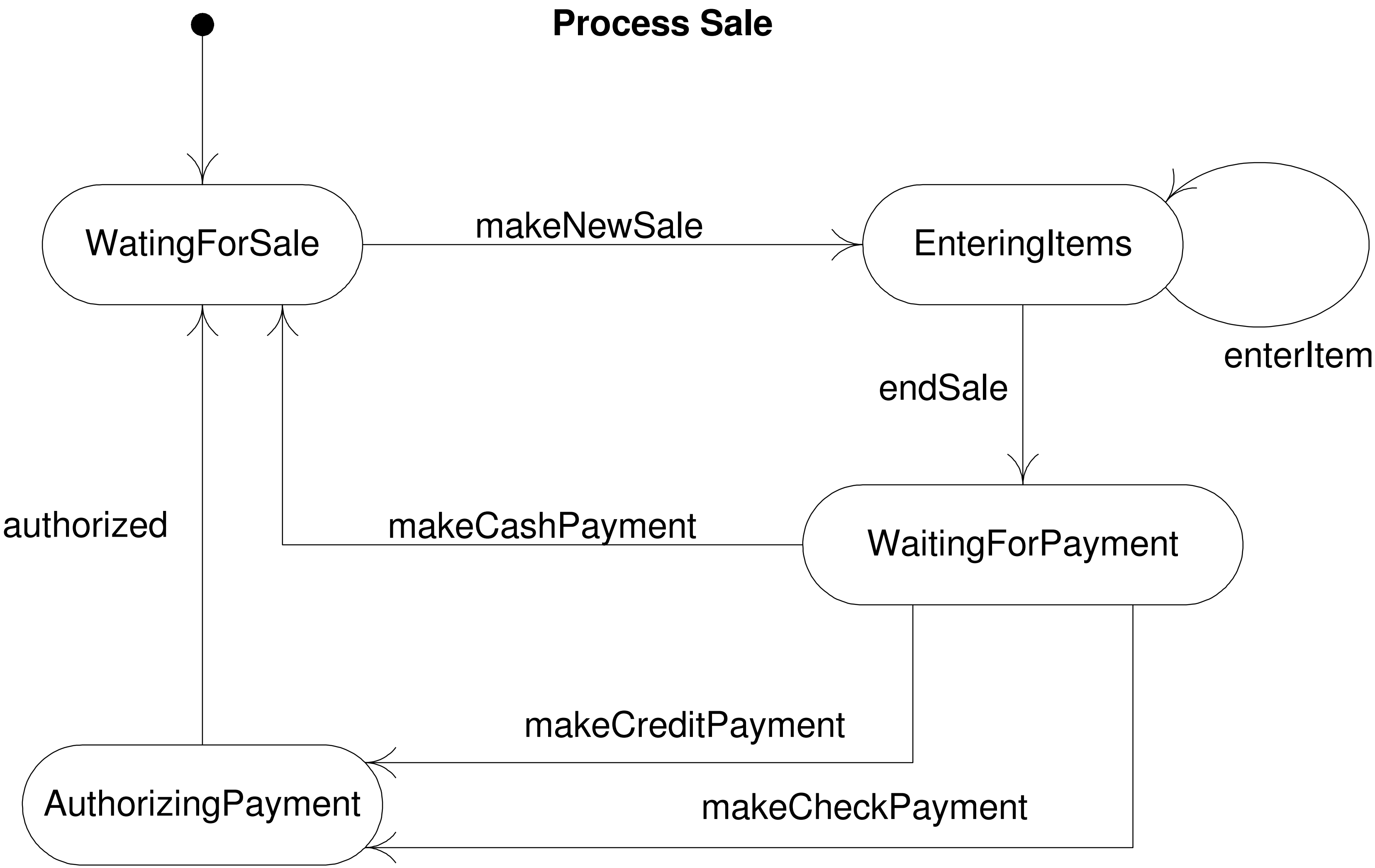


Fig. 29.5 Legal sequence of use case operations



GoF State Pattern

- Problem:
  - An object's behavior is dependent on its state, and its methods contain case logic reflecting conditional state-dependent actions.
- Solution:
  - Create a state class for each state, implementing a common interface.
  - Delegate state-dependent operations from the context object to its current state object.
  - Ensure context object always points to a state object reflecting its current state.

Example: Transactional States

- A transactional support system typically keeps track of the state of each persistent object.
  - Modifying a persistent object does not cause an immediate database update — an explicit *commit* operation must be performed.
  - A *delete* or *save* causes change of state, not an immediate database delete or save.
  - A *commit* operation updates the database if an object was modified (“dirty”), but does nothing if the object is “clean”.

Fig. 38.12 Statechart for *PersistentObject*

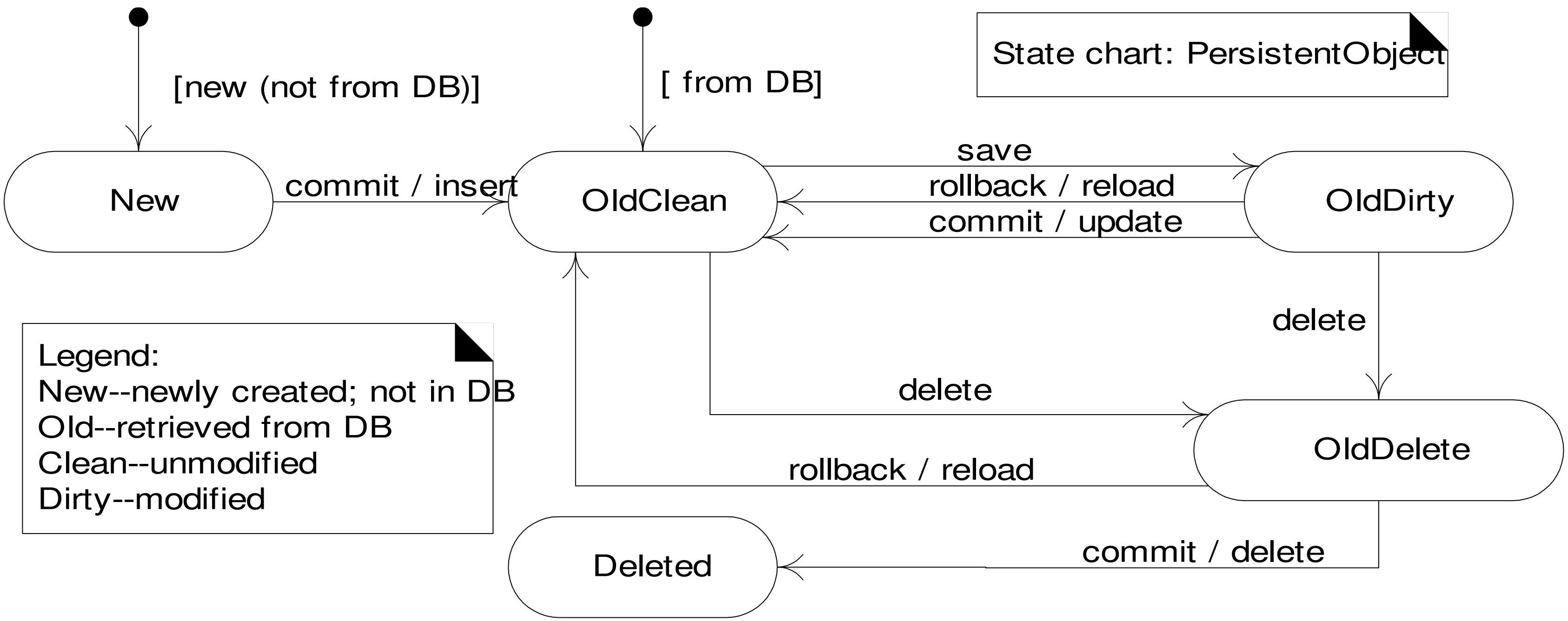
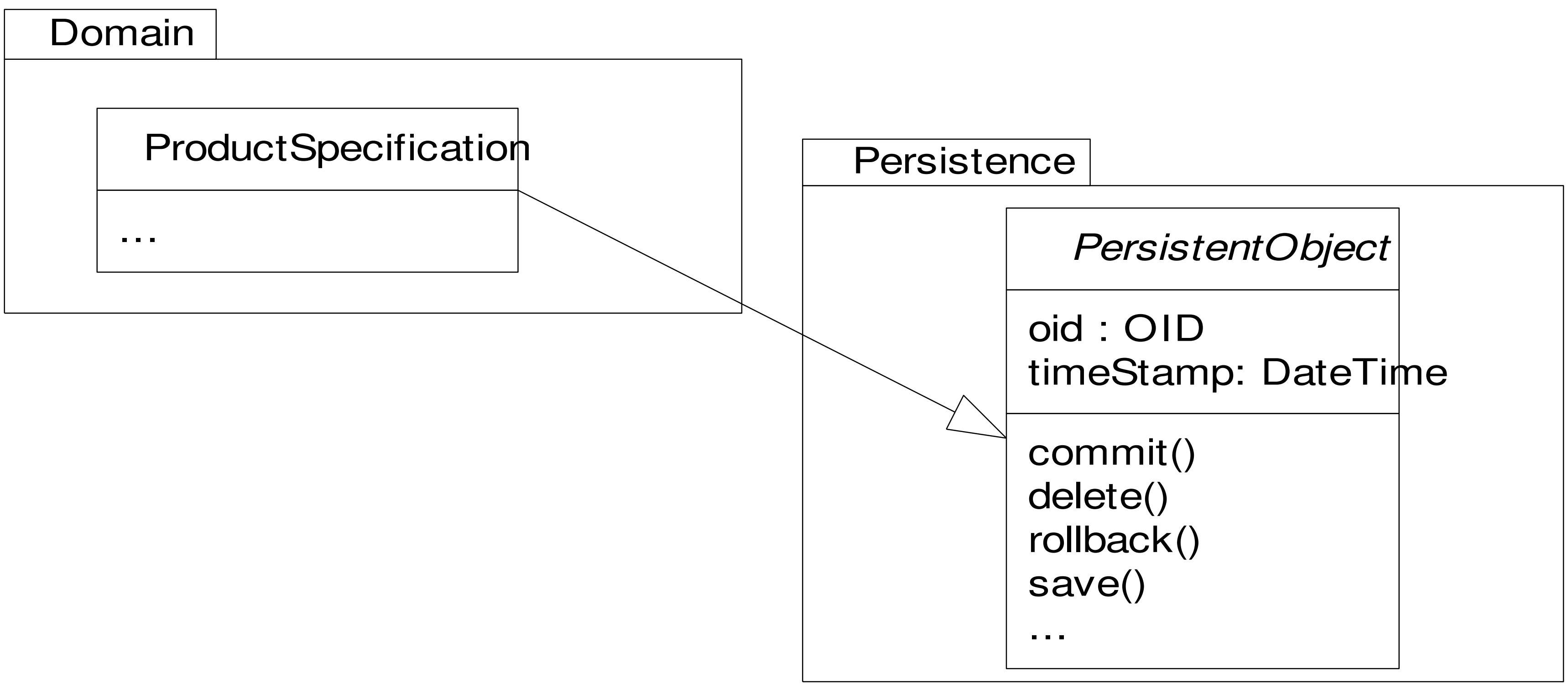


Fig. 38.13 Persistent Objects

- Assume all persistent object classes extend a *PersistentObject* class that provides common technical services for persistence.



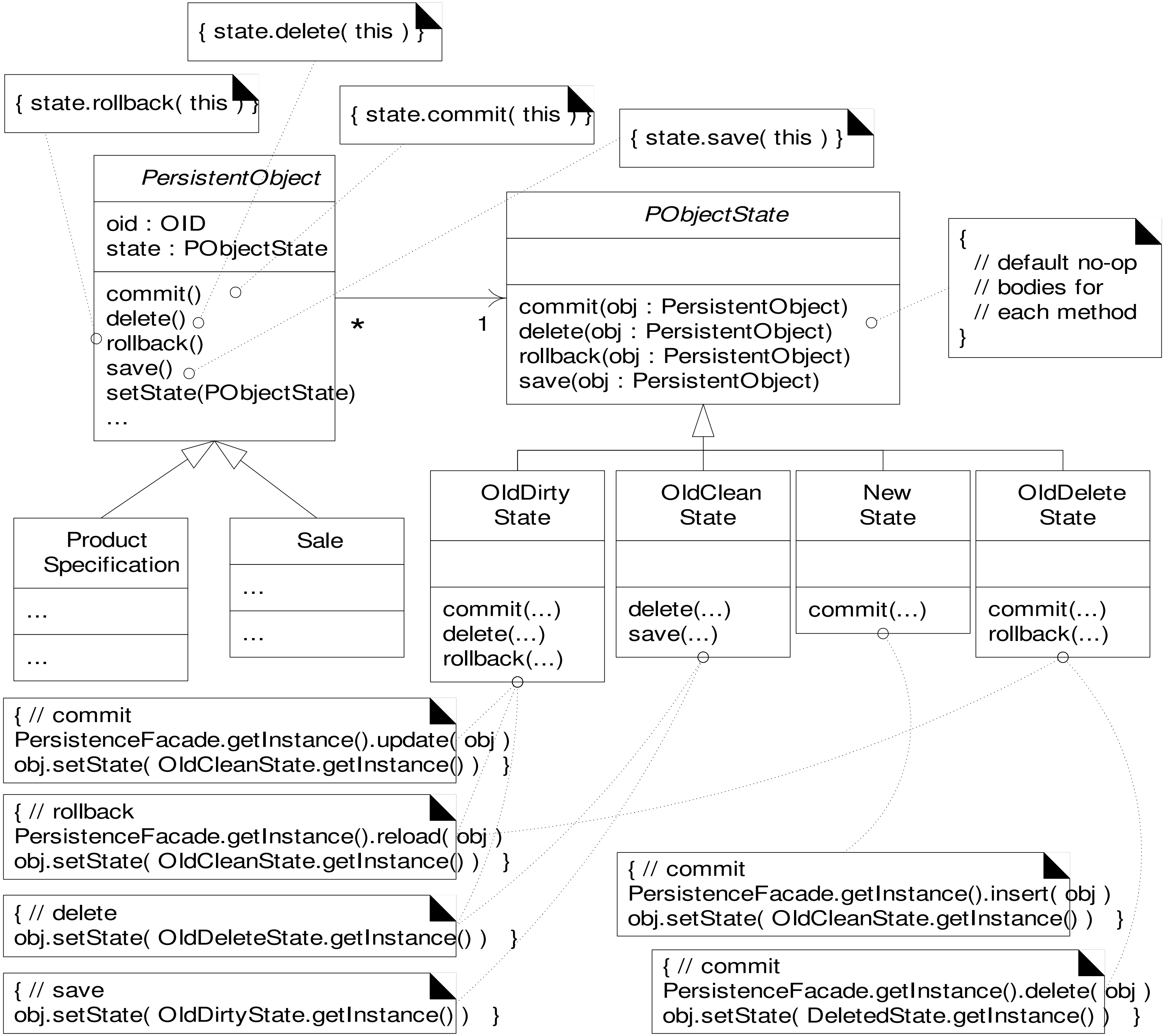
Case-logic Structure

- Using case logic, *commit* and *rollback* methods perform different actions, but have a similar logic structure.

```
public void commit()
{
   switch ( state )
   {
      case OLD_DIRTY:
         // . . .
         break;
      case OLD_CLEAN:
         // . . .
         break;
   . . .
```

State Transition Model using State Pattern

- Implementing transactional states:
  - Create static singleton objects for each state that are specializations of *PObjectState.*
    - The *commit* method is implemented differently in each state object.
  - *PersistentObject* is the context object.
    - Keeps a reference to a state object representing the current state.
    - Methods in the state objects call *setState*() to cause a transistion to the next state.
- No case logic is needed.



{ state.delete(this) }
{ state.rollback(this) }
{ state.commit(this) }
{ state.save(this) }
PersistentObject
oid : OID
state : PObjectState
commit()
delete()
rollback()
save()
setState(PObjectState)
...
*
1
PObjectState
commit(obj : PersistentObject)
delete(obj : PersistentObject)
rollback(obj : PersistentObject)
save(obj : PersistentObject)
{
// default no-op
// bodies for
// each method
}
Product Specification
...
...
Sale
...
...
OldDirty State
commit(...)
delete(...)
rollback(...)
OldClean State
delete(...)
save(...)
New State
commit(...)
OldDelete State
commit(...)
rollback(...)
{ // commit
PersistenceFacade.getInstance().update(obj)
obj.setState(OldCleanState.getInstance()) }
{ // rollback
PersistenceFacade.getInstance().reload(obj)
obj.setState(OldCleanState.getInstance()) }
{ // delete
obj.setState(OldDeleteState.getInstance()) }
{ // save
obj.setState(OldDirtyState.getInstance()) }
{ // commit
PersistenceFacade.getInstance().insert(obj)
obj.setState(OldCleanState.getInstance()) }
{ // commit
PersistenceFacade.getInstance().delete(obj)
obj.setState(DeletedState.getInstance()) }