Object- Oriented Design with UML and Java

Part X: UML State Diagrams

Version 2023

State

The composite of an object's data, including its ***identity*** and/or ***memory address***.

When designing a UML state diagram:

- The pertinent ***state*** to consider is the set of attributes and links that affect behavior.
- By definition in this context, no two states have identical responses to all ***events***.

Stateful Objects

It may be useful to model state-dependent behavior with UML

- Easier to understand and debug than code
- Can make good documentation

Common ***stateful*** objects:

- User-Sessions
  - A *Shopping Cart* remembers your purchases.
  - Can be designed to be ***stateless*** in Java, backed by a relational database , *or* ***stateful*** using an EJB3 “stateful session bean.”
- Controllers:
  - A *Clerk* in the video store may be *busy* or *free.*
- Devices:
  - A *Modem* object could be *dialing*, *sending*, *receiving*, etc.

State Diagrams

- A type of finite state machine.
- Model how an object moves from state to state for its entire ***lifetime***.
- A class has its own state diagram *if* it has interesting dynamic behavior.

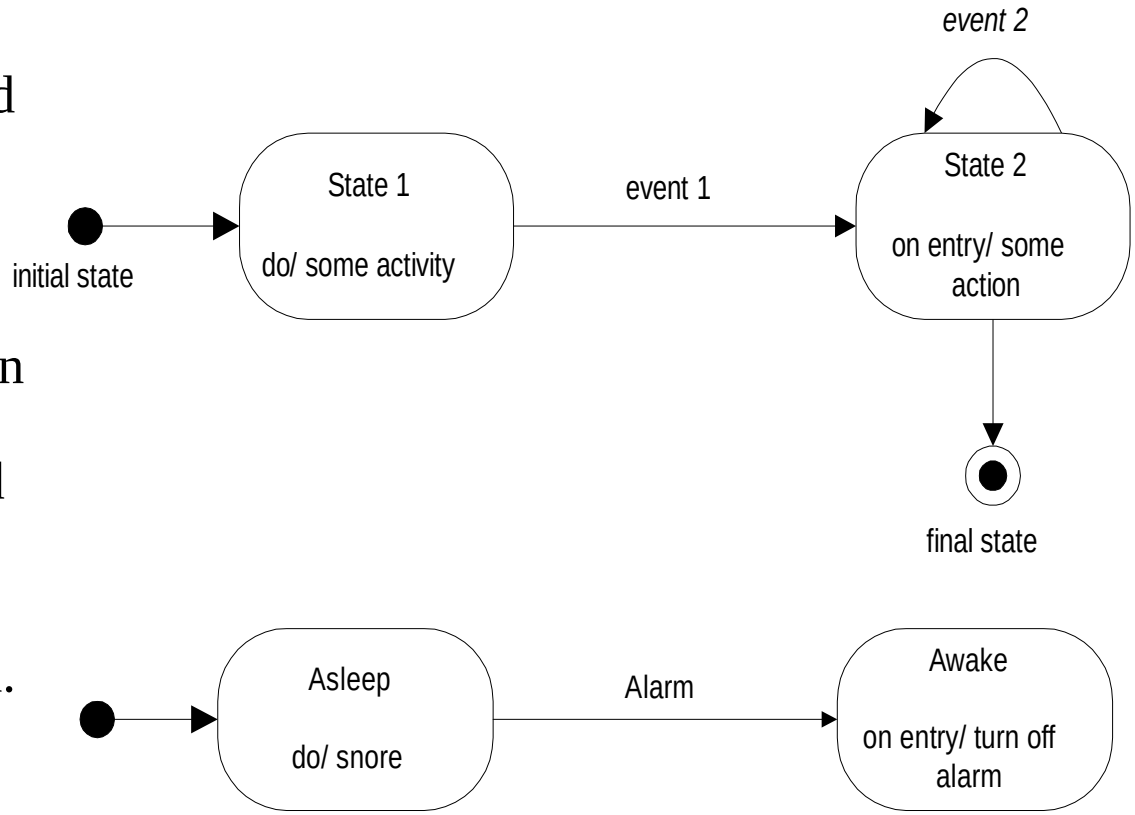
Example: The state of a String class is the ASCII value of the String; this probably doesn't need to be modeled.

Example: The states of a Telephone Connection class (dial tone, dialing, ringing, connected, hung up) is probably interesting enough to model.

- One class' state diagram may refer to the state of another class.
- The complete state diagram of a system is a collection of sub-diagrams that interact by sending events to each other.
- Not ideal for situations involving many collaborating objects.

State Diagram Symbols

- States are represented by ovals.
- A state may or may not have a name.
- Directed arcs between states represent transitions associated with events.
- The *source* of the event is not specified.

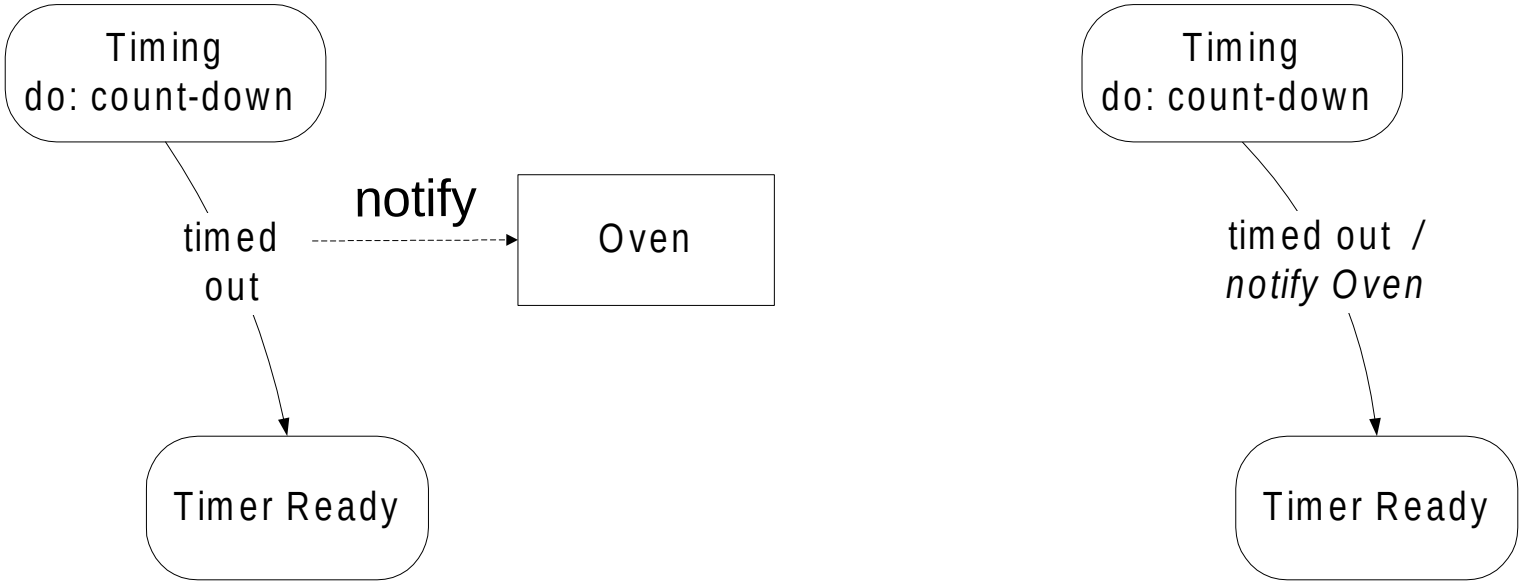


Activities, Actions, Events

- **Activities** are operations that take time:
  - Generating microwaves.
  - Writing to a disk.
  - Can often be modeled as nested state diagrams.
- **Actions** are of very short duration:
  - Beep.
  - Display a menu.
  - Set a flag.
- **Events** cause changes in state:
  - State transitions can trigger actions.
  - Transitions are essentially instantaneous.

Action on a Transition

Example from Microwave Timer:



Model Syntax

The general syntax for *arc adornments*:

- **event1 (attributes) [conditions] / actions**
- IF event1 occurs AND IF the conditions are true THEN make the state transition specified by the arc AND spawn the specified action(s).

Keywords used within a state:

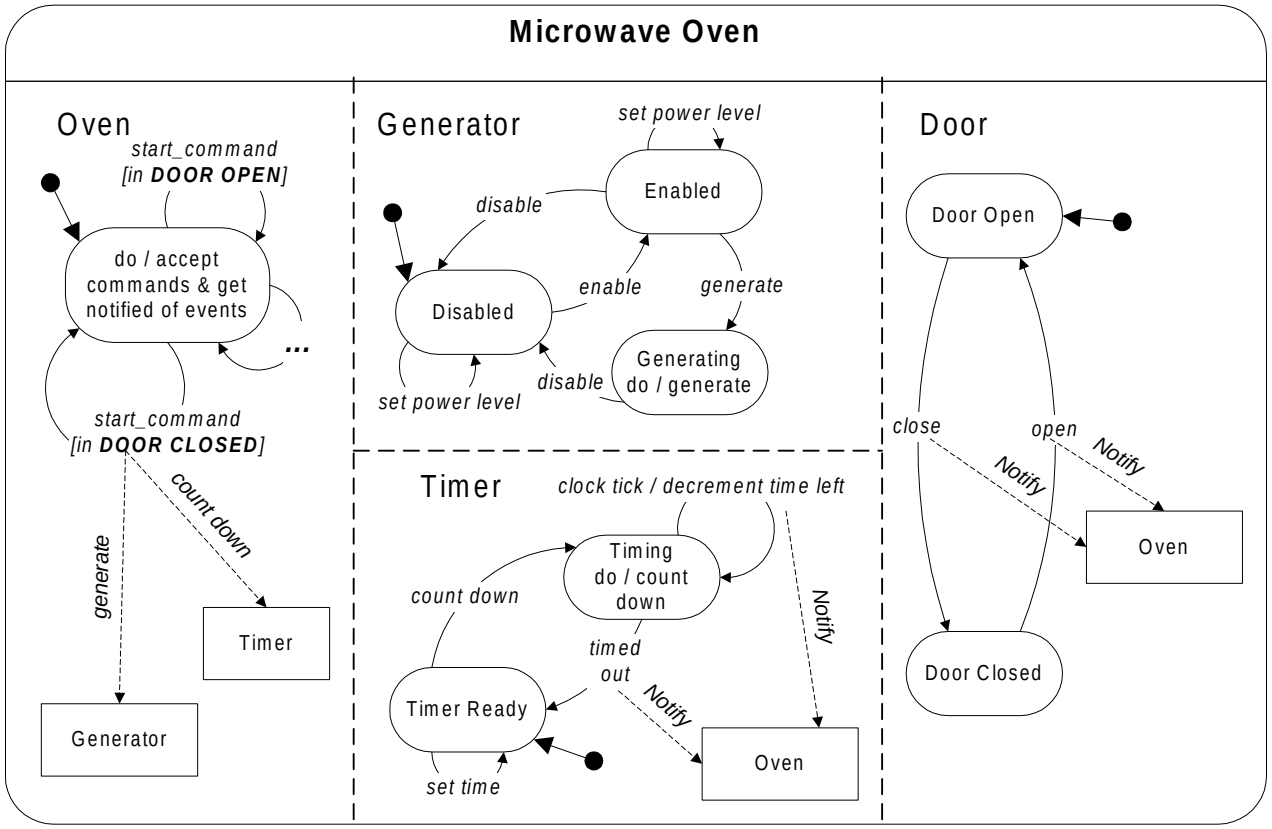
- **Actions** can be spawned on *entry* and *exit* to states:
  - **entry** / entry-action(s)
  - **exit** / exit-action(s)
- **Activities** may be indicated:
  - **do** / activity(ies)
- **Internal events** may be indicated (instead of self-directed arcs):
  - internal-event1 / action(s)

Concurrent State Machines

If an object is an aggregation of other objects, it is possible to have concurrent state machines, one for each of the aggregate objects. Arc conditionals can refer to the states of the other aggregate objects.

- For example, a microwave oven can have concurrent state models for the timer, the generator, and the door. The “push start button” event will have no effect if the state of the door is *open*; if the door is *closed*, however, then a message is sent to both the timer and the generator.

Example: Microwave Aggregation

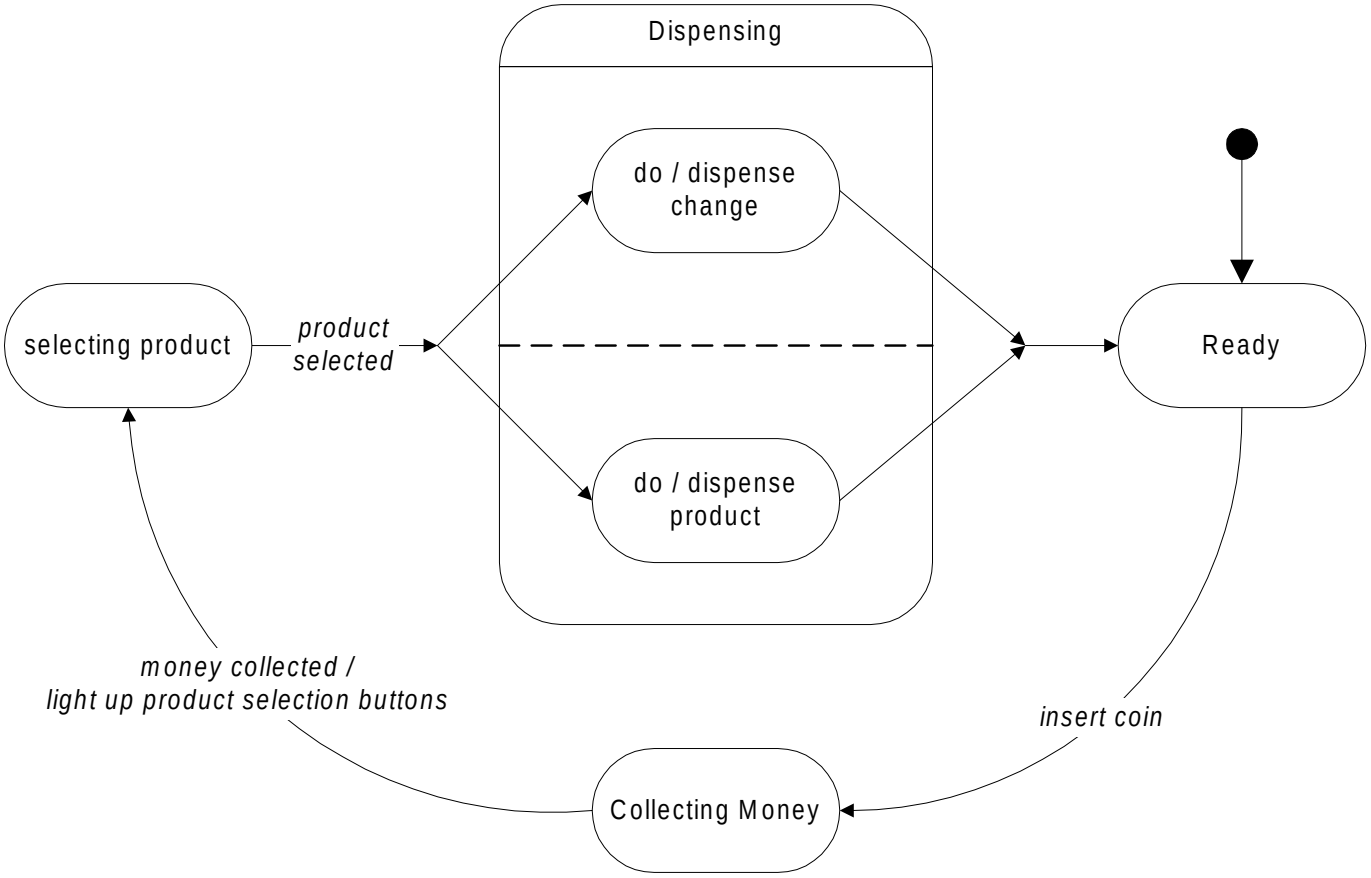


Splitting and Synchronization

Sometimes an object must perform two or more activities concurrently, both of which must complete before the next state can be reached; this is called “splitting and synchronization”.

- For example, a vending machine might have to dispense both change and product before being ready for its next transaction.
- In the model on the next slide, note how the two concurrent “dispense” states can be combined into a “superstate” (also known as a generalization relationship).

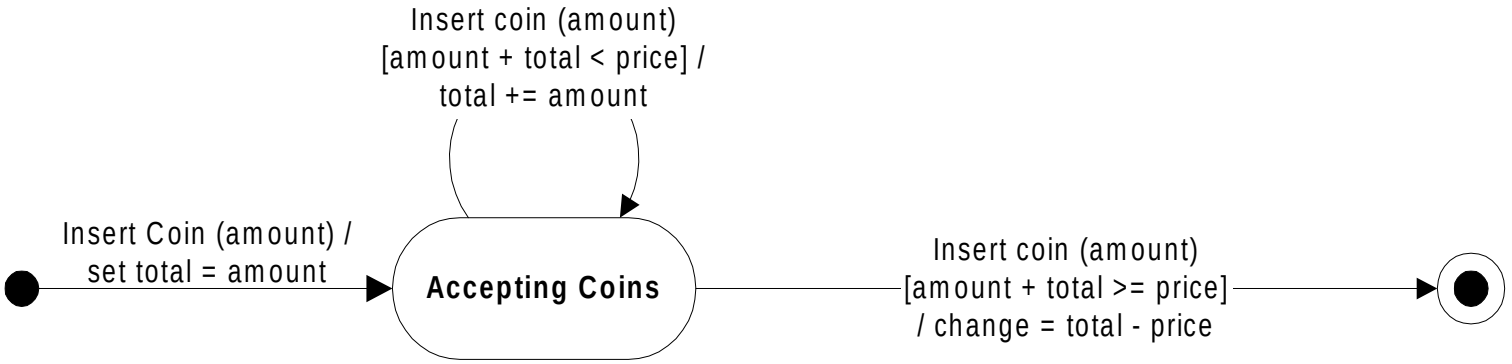
Splitting and Synchronization



Level of detail

- It is possible to decrease the visual complexity of a model by generalizing activities, actions, events and states as higher-level, more abstract elements, and then showing them broken down in separate diagrams.
- For example, the vending machine state “collecting money” is really more complex than it appears from the previous slide...
- This is also useful when the same event(s) cause the same action(s) for multiple states... it becomes possible to generalize.

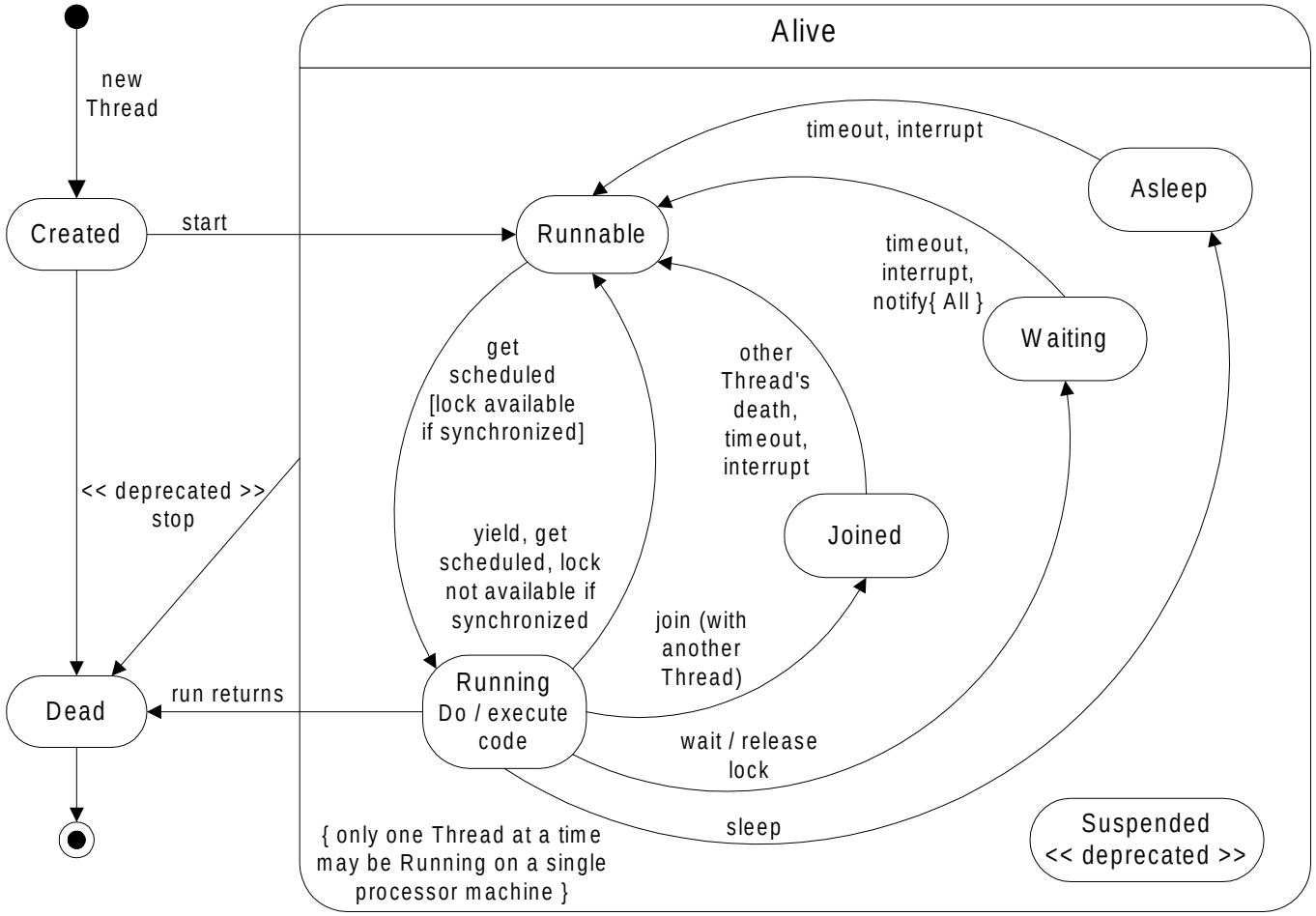
Example: Vending Machine



Consider also the Sticks Game Referee...

There could be a high level “Running Game” state...

Example: Java Thread Life Cycle



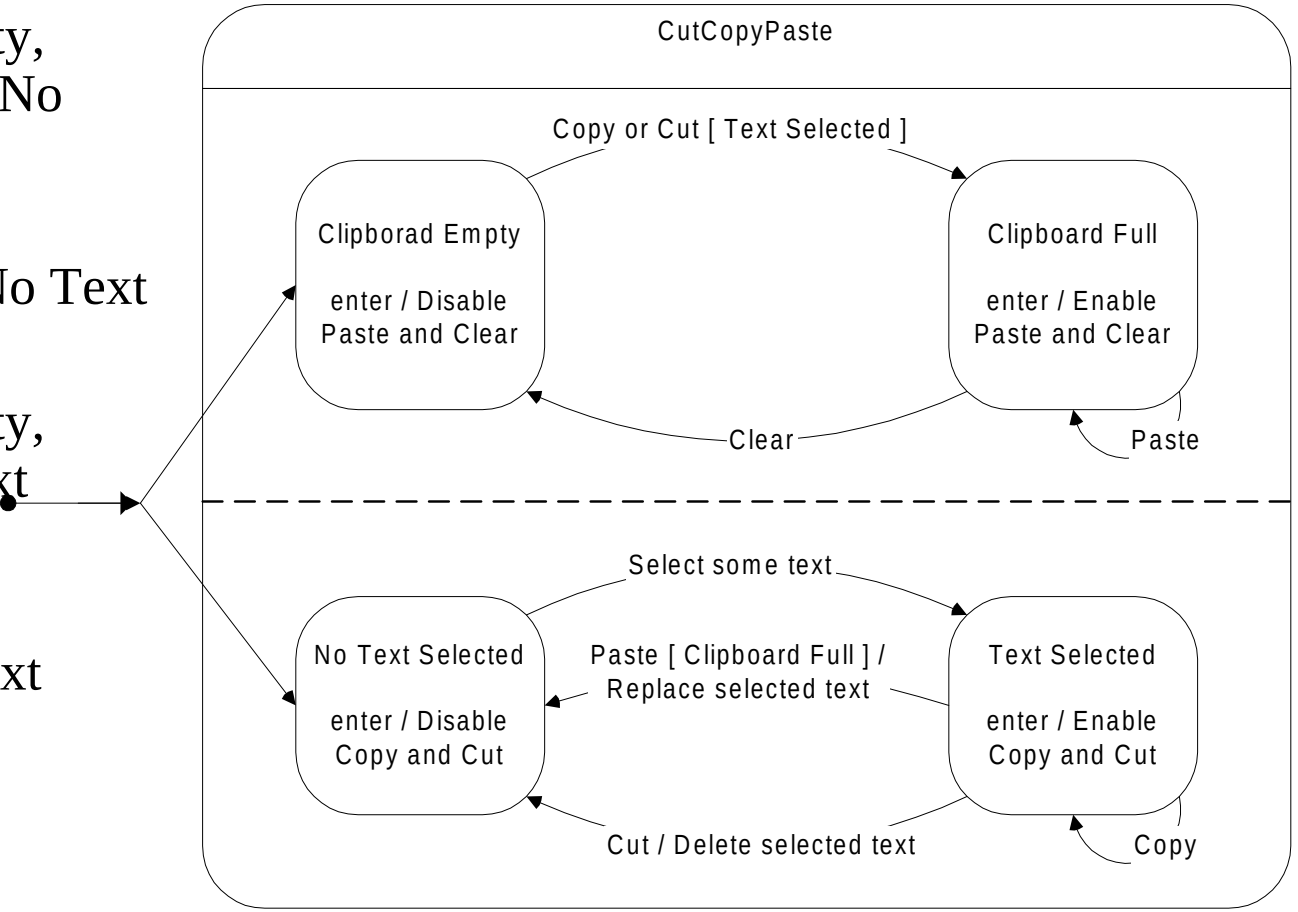
Example: Cut, Copy and Paste

- A text editor UI has cut, copy & paste functionality. Paste only works if there is text in the clipboard. As the result of a paste command, any selected text is replaced with the text from the clipboard; the resulting, pasted text, is not selected. The clipborad may only be filled with a copy or cut command. Cut and copy only work if there is text selected. The clipboard may be emptied with an additional clear command. Cut deletes the previously selected text. The UI provides cut, copy, paste, and clear push buttons as shortcuts to this functionality (text is selected using the mouse). These push buttons should be grayed-out when they are not appropriate; for example, the user should not be able to invoke paste when there is no text in the clipboard.

Cut, Copy, and Paste (cont.)

4 states:

- Clipboard Empty, No Text Selected
- Clipboard Full, No Text Selected
- Clipboard Empty, Text Selected
- Clipboard Full, Text Selected

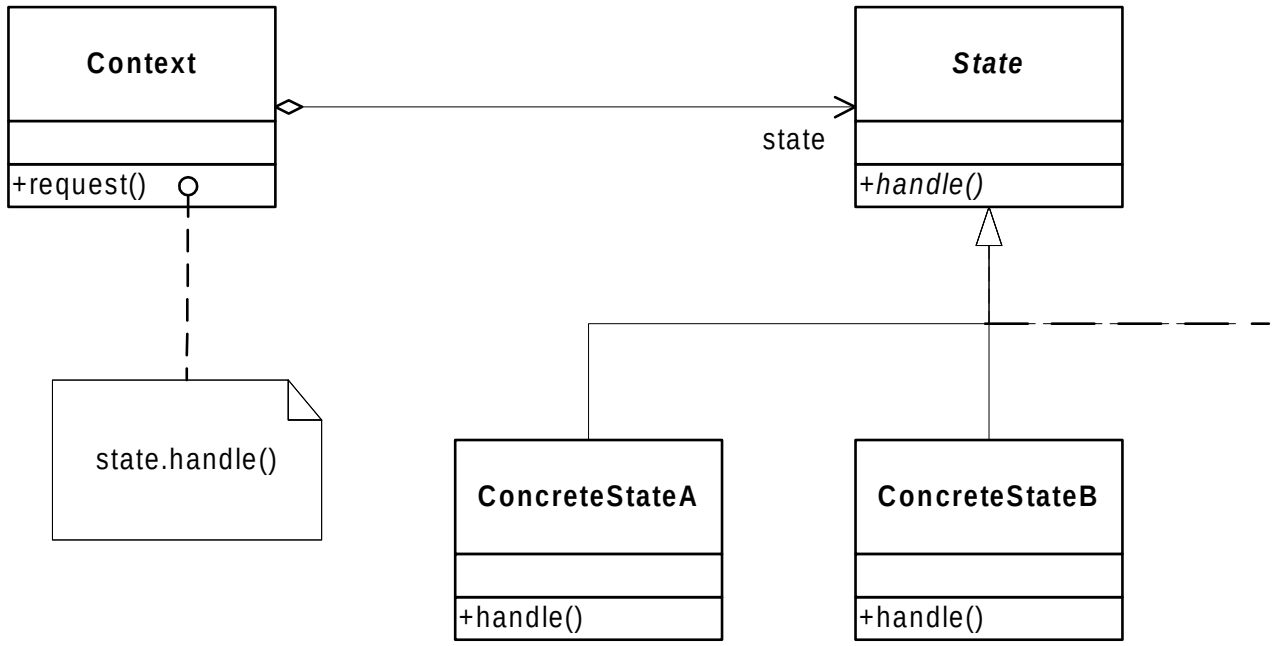


Implementing States

- Tables
  - Common procedural approach.
  - Compact and efficient.
  - Good for large state machines.
  - Can be difficult to maintain.
  - Require good documentation.
- Switch statements
  - Easier to understand than tables.
  - Tend to be self-documenting.
  - Can be difficult to maintain.
- *State* design pattern
  - Easy to extend and modify.

Design Pattern: *State*

Intent: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class from the point of view of the calling client (meaning: its behavior will change).



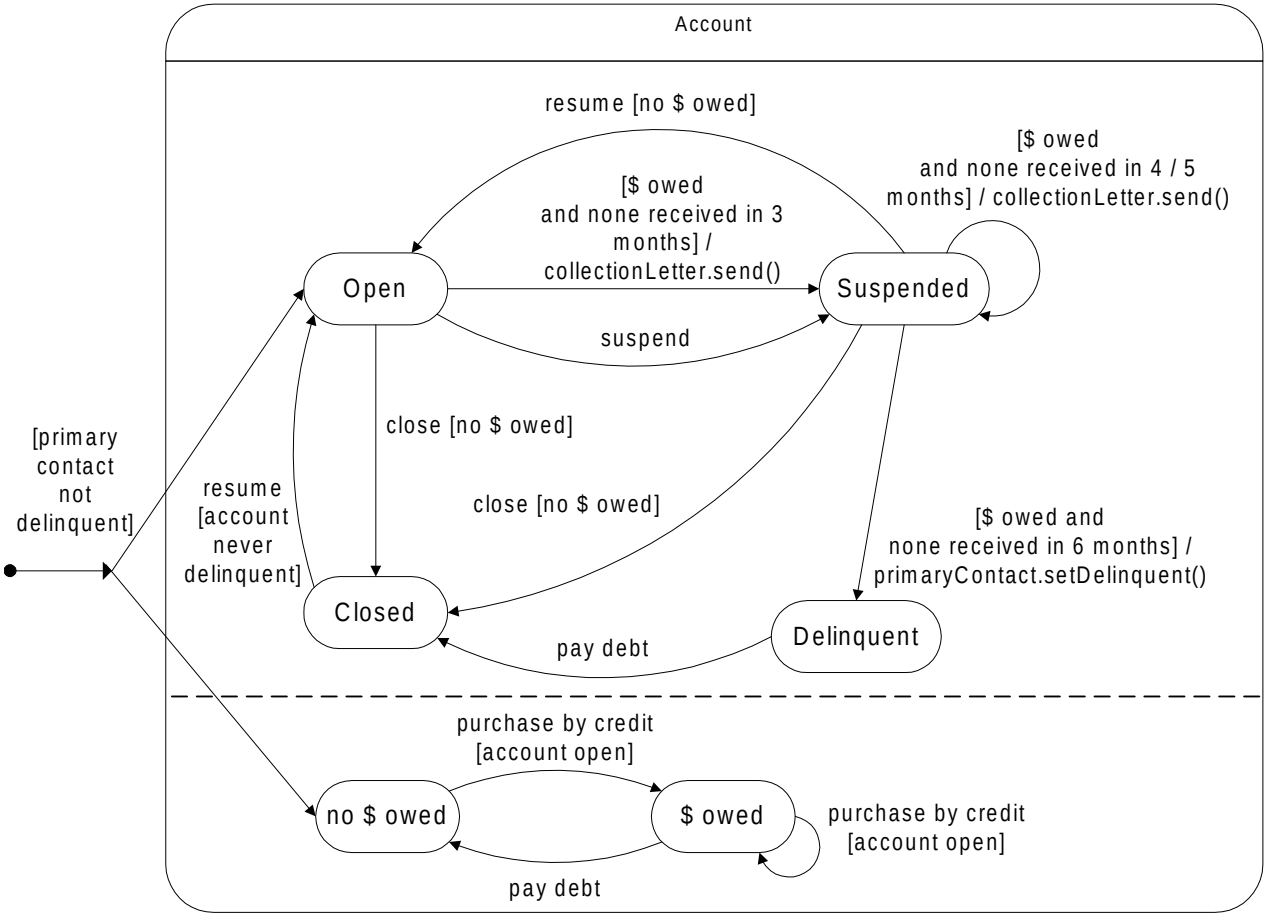
State Design Pattern Detail

- How does a State know what state to go to next on a transition?
  - Each class can have its own table or switch statement, or a hash table of transitions keyed by their trigger Events (and guard conditions).
  - Consider using *State, Action, Event* and *Transition* classes.
  - Note: The *Action* class might be implemented using the *Command* design pattern.

Example: Customer Account

- A “customer care” application is used by Customer Service Representatives (CSRs) who talk to customers over the telephone. A given call consists of a conversation between a CSR and a contact person representing a customer account, creating “service requests”, such as: purchase goods on credit, receive payment, and “change account status” requests to open, close, suspend, or resume an account. Purchases may only be made on open accounts. An account cannot be closed if there is money owed. A suspended account may have service resumed only if there is no money owed. The only valid service request for a closed account is resume. An account will automatically get suspended if money is owed and none has been received for 3 months, in which case a collection letter will also be sent. More collection letters will be sent if no money has been received after 4 & 5 months; after 6 months, the account becomes delinquent, and may never be reopened, but of course the customer can pay his or her debt to close the account; furthermore, the primary contact for a delinquent account may never again open an account.

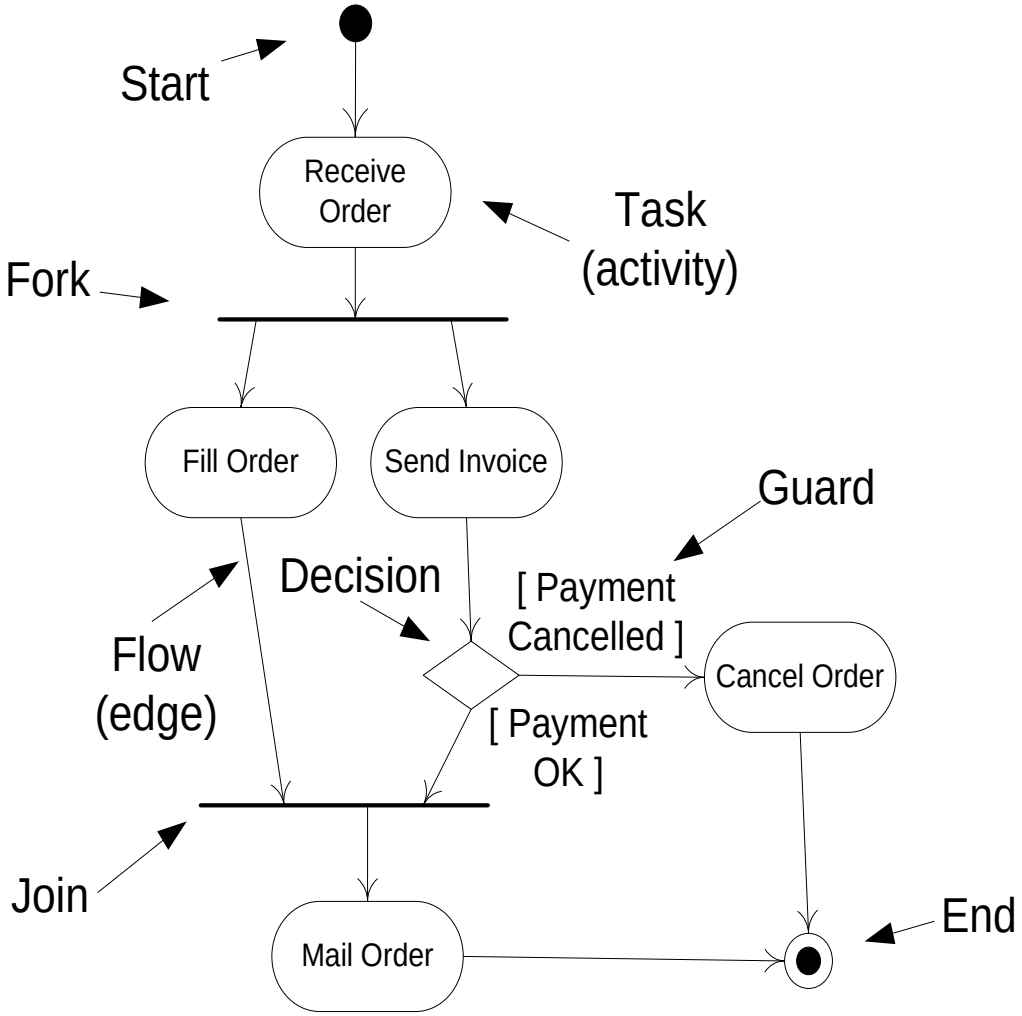
Example: Customer Account



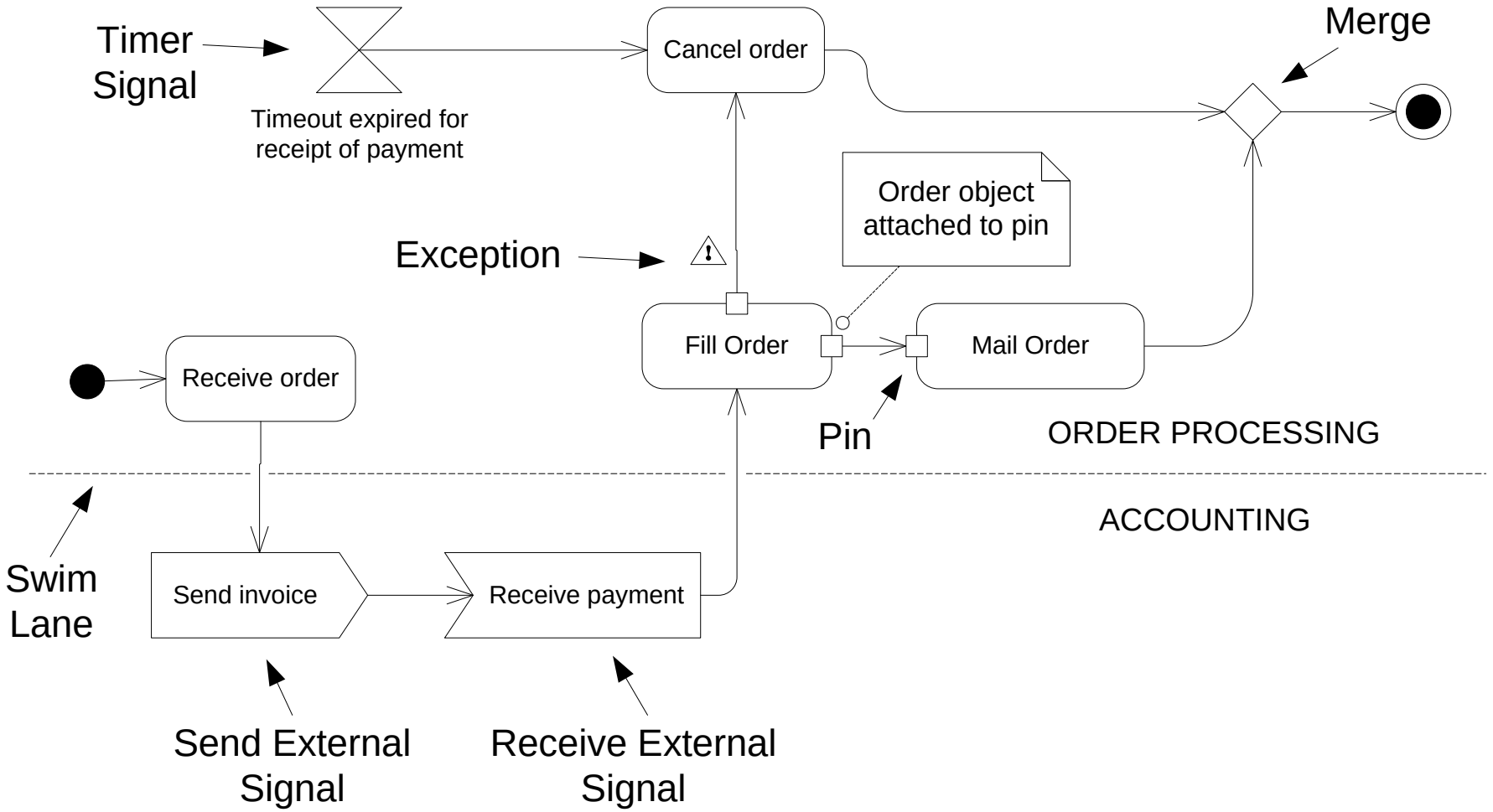
Activity Diagram

- Similar to State Diagram except that a state represents the performance of an operation (a step in a business process); a transition from one state to another is triggered by the completion of the operation.
- Use Activity Diagrams to model process knowledge.
- The emphasis is on I/O dependencies, sequencing and conditions (responsibilities are secondary).

Activity Diagram Vocabulary



More Activity Diagram Vocabulary



Summary

State Diagrams **-** Ideal for one class when it has interesting dynamic behavior depending on its state.

Activity Diagrams **-** Like a flowchart except better. Activity Diagrams are excellent for modeling business workflows. These diagrams enable visualization of parallel activities and their sequencing and synchronization.

Good Code **-** With good comments also makes excellent documentation. But how easy is it for non-programmers to visualize on a white board?