Software Design Refinement Using Design Patterns Part II The FSM and the StateChart Patterns

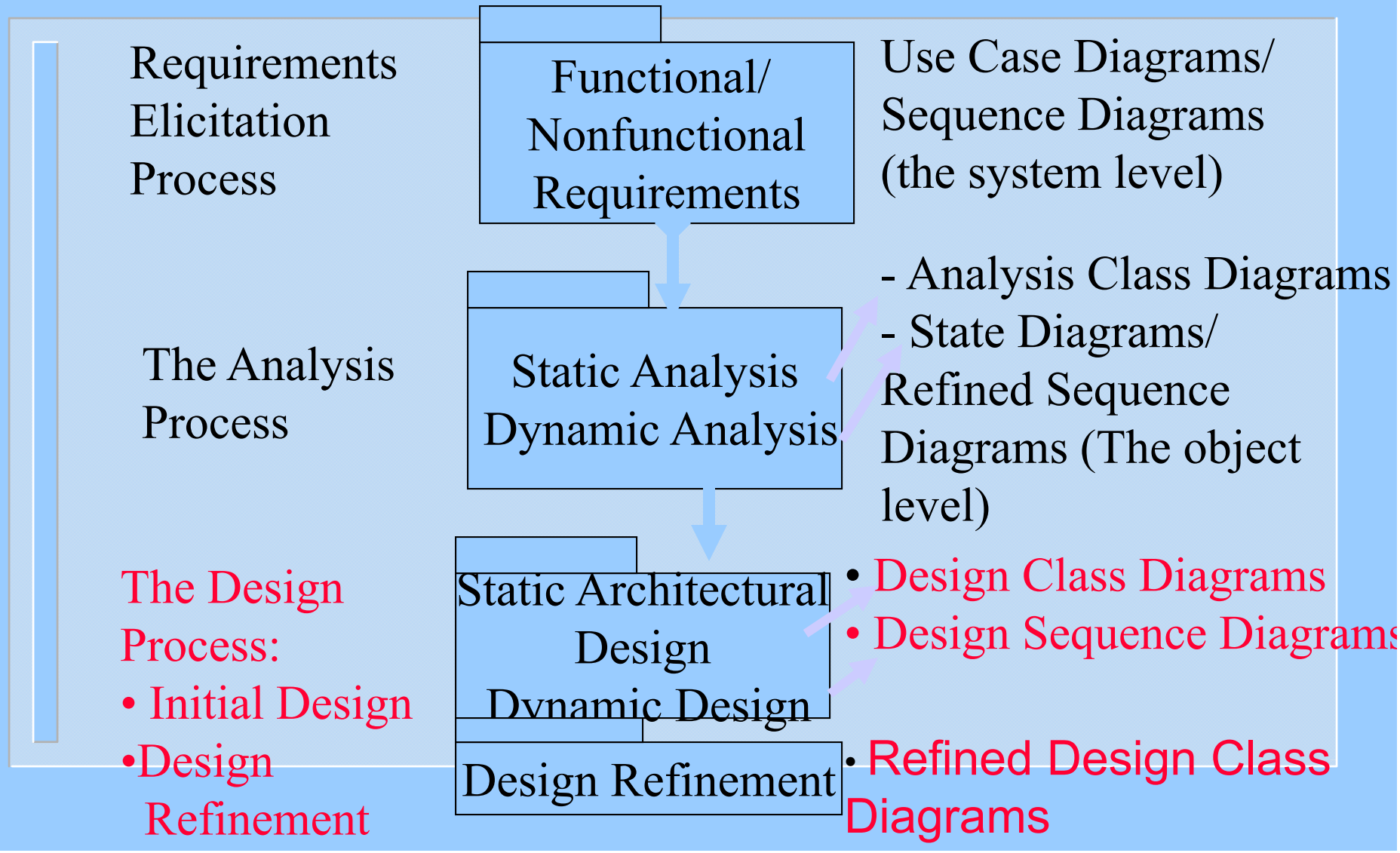
Instructor: Dr. Hany H. Ammar

Dept. of Computer Science and Electrical Engineering, WVU

Outline

- **Review**
  - The Requirements, Analysis, Design, and Design Refinement Models
  - Design refinement and Design Patterns
  - Examples of Design Patterns: The State Pattern
- **Finite State Machine Pattern Language**
  - Basic FSM, State-Driven Transitions
  - Interface Organization, Layered Organization
- **A Pattern Language for StateCharts**
  - Basic StateCharts, Hierarchical Statechart
  - Orthogonal Behavior

The Requirements, Analysis, Design, and Design Refinement Models



Design Refinement

- It is difficult to obtain a quality design from the initial design
- The initial design is refined to enhance design quality using the software design criteria of modularity, information hiding, complexity, testability, and reusability.
- New components (or new classes) are defined and existing components (or classes) structures are refined to enhance design quality
- The design refinement step is an essential step before implementation and testing.

Class Diagram Refinement Using Design Patterns

- Design Class Diagrams are further refined to enhance design quality (i.e., reduce coupling, increase cohesion, and reduce component complexity) **using design patterns**
- A design pattern is a documented good design solution of a design problem
- Repositories of design patterns were developed for many application domains (communication software, agent-based systems, web applications)
- Many generic design patterns were defined and can be used to enhance the design of systems in different application domains

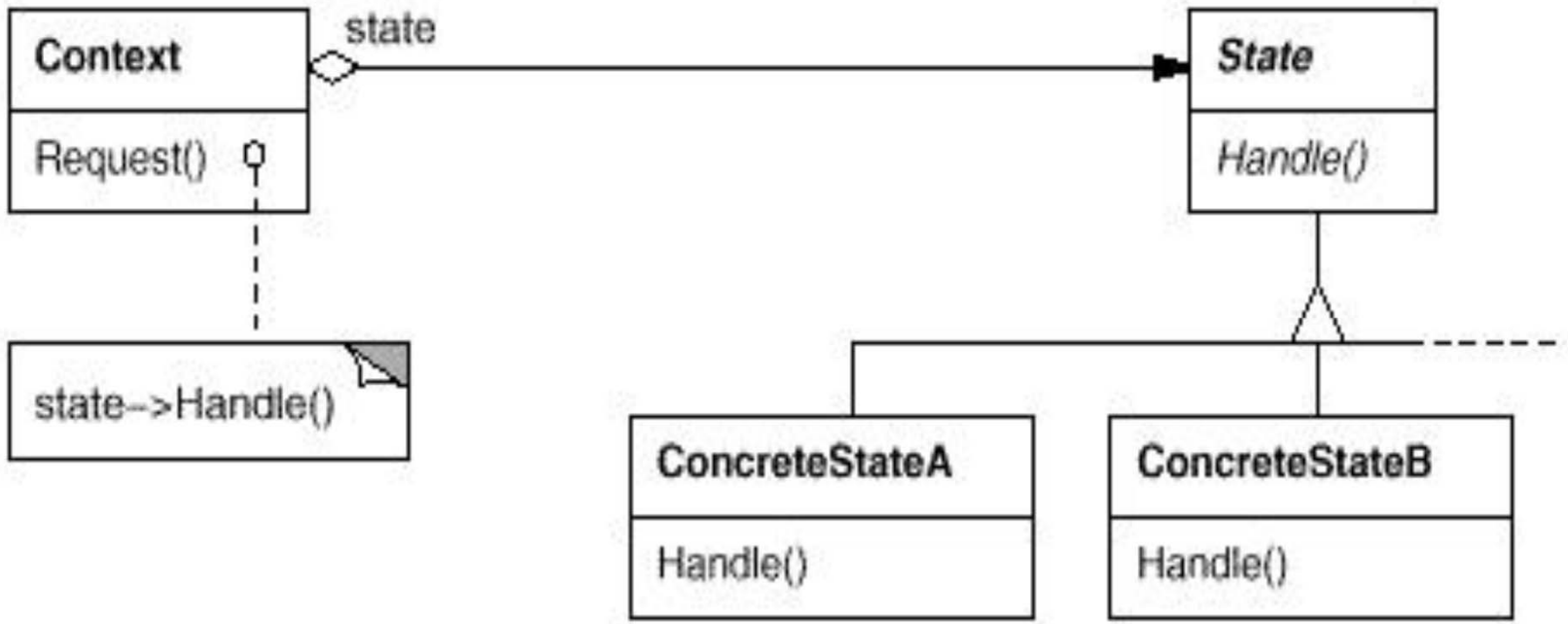
What is a Design Pattern

- What is a Design Pattern?

  A design pattern describes a design problem which repeatedly occurred in previous designs, and then describes the core of the solution to that problem
- Solutions are expressed in terms of classes of objects and interfaces (object-oriented design patterns)
- A design pattern names, abstracts, and identifies the key aspects of a high quality design structure that make it useful for creating reusable object-oriented designs

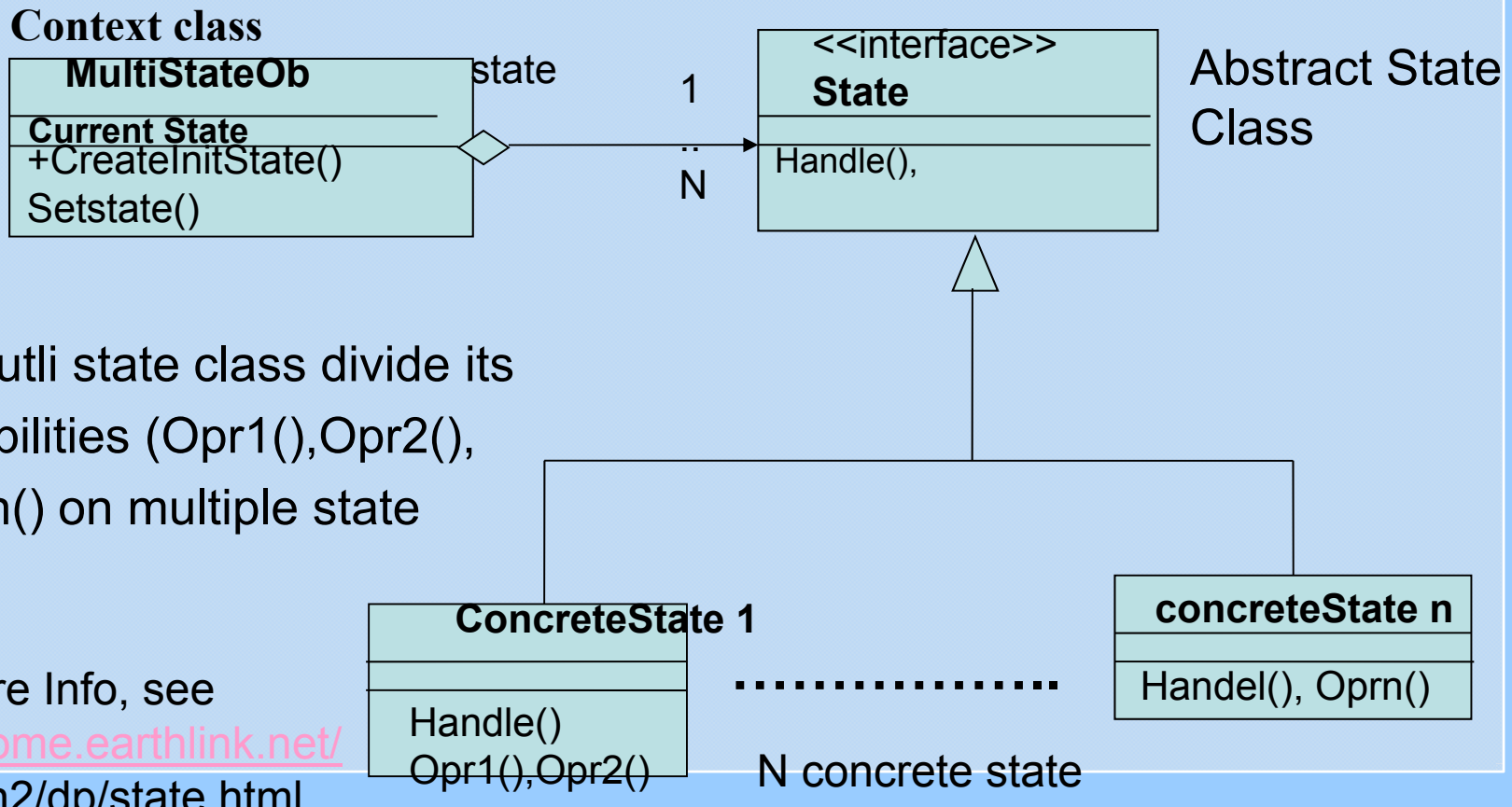
Recall Examples of Design Patterns
The State Pattern
(Examples of State and Strategy Patterns)

The State Pattern: is a solution to the problem of how to make the behavior of an object depend on its state.



Examples of Design Patterns
The State Pattern

- The State Pattern can be used for example to encapsulate the states of a controller as objects



The Context Class

Abstract State Class

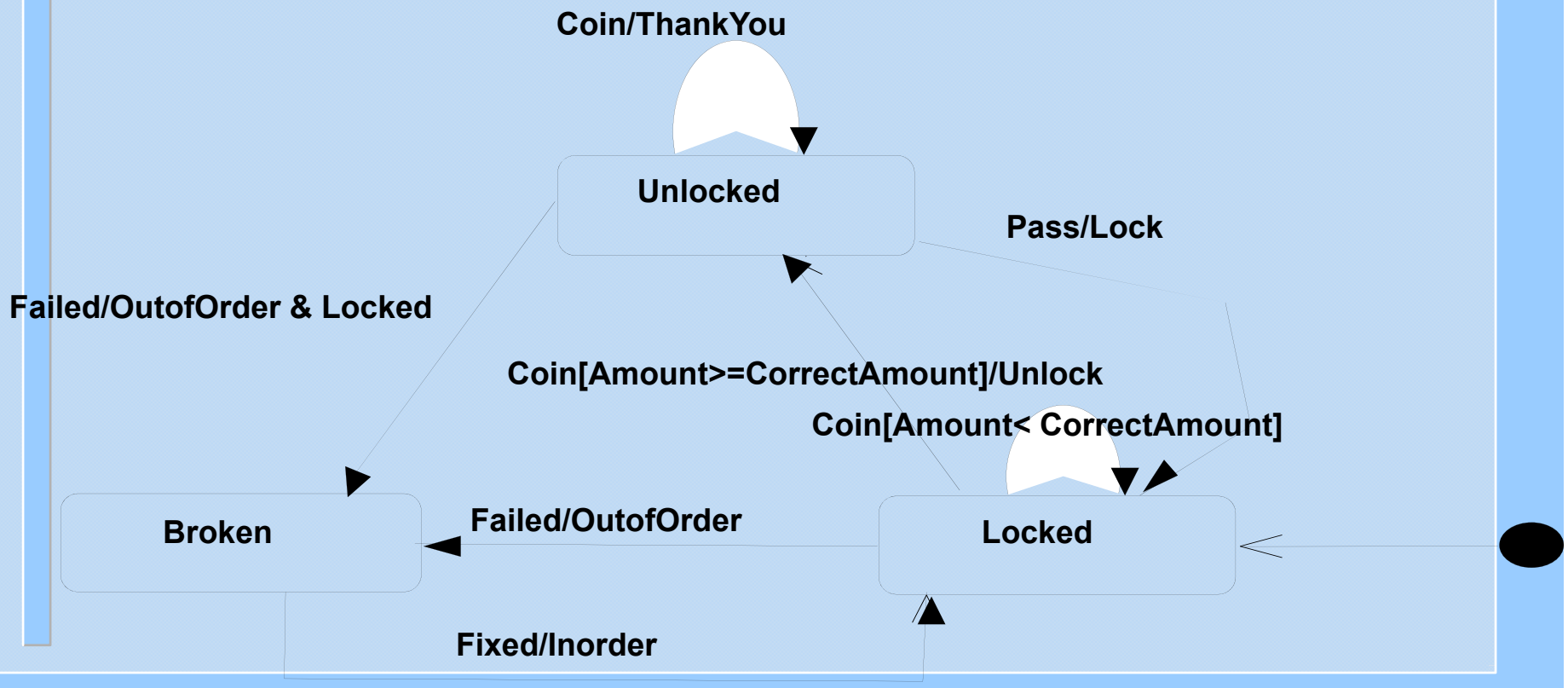
Lets a mutli state class divide its responsibilities (Opr1(),Opr2(), and Oprn() on multiple state classes.

N concrete state classes

For more Info, see http://home.earthlink.net/~huston2/dp/state.html

Example: Turn style coin machine

The machine starts in a locked state (Locked). When a coin is detected (Coin), the machine changes to the unlocked state (UnLocked) and open the turnstyle gate for the person to pass. When the machine detects that a person has passed (Pass) it turns back to the locked state.



Illustrating Example: Turn style coin machine

- If a person attempts to pass while the machine is locked, an alarm is generated.
- If a coin is inserted while the machine is unlocked, a Thankyou message is displayed.
- When the machine fails to open or close the gate, a failure event (Failed) is generated and the machine enters the broken state (Broken).
- When the repair person fixes the machine, the fixed event (Fixed) is generated and the machine returns to the locked state.

Outline

- **Review**
  - The Requirements, Analysis, Design, and Design Refinement Models
  - Design refinement and Design Patterns
  - Examples of Design Patterns: The State Pattern
- **Finite State Machine Pattern Language**
  - Basic FSM, State-Driven Transitions
  - Interface Organization, Layered Organization
- **A Pattern Language for StateCharts**
  - Basic StateCharts, Hierarchical Statechart
  - Orthogonal Behavior

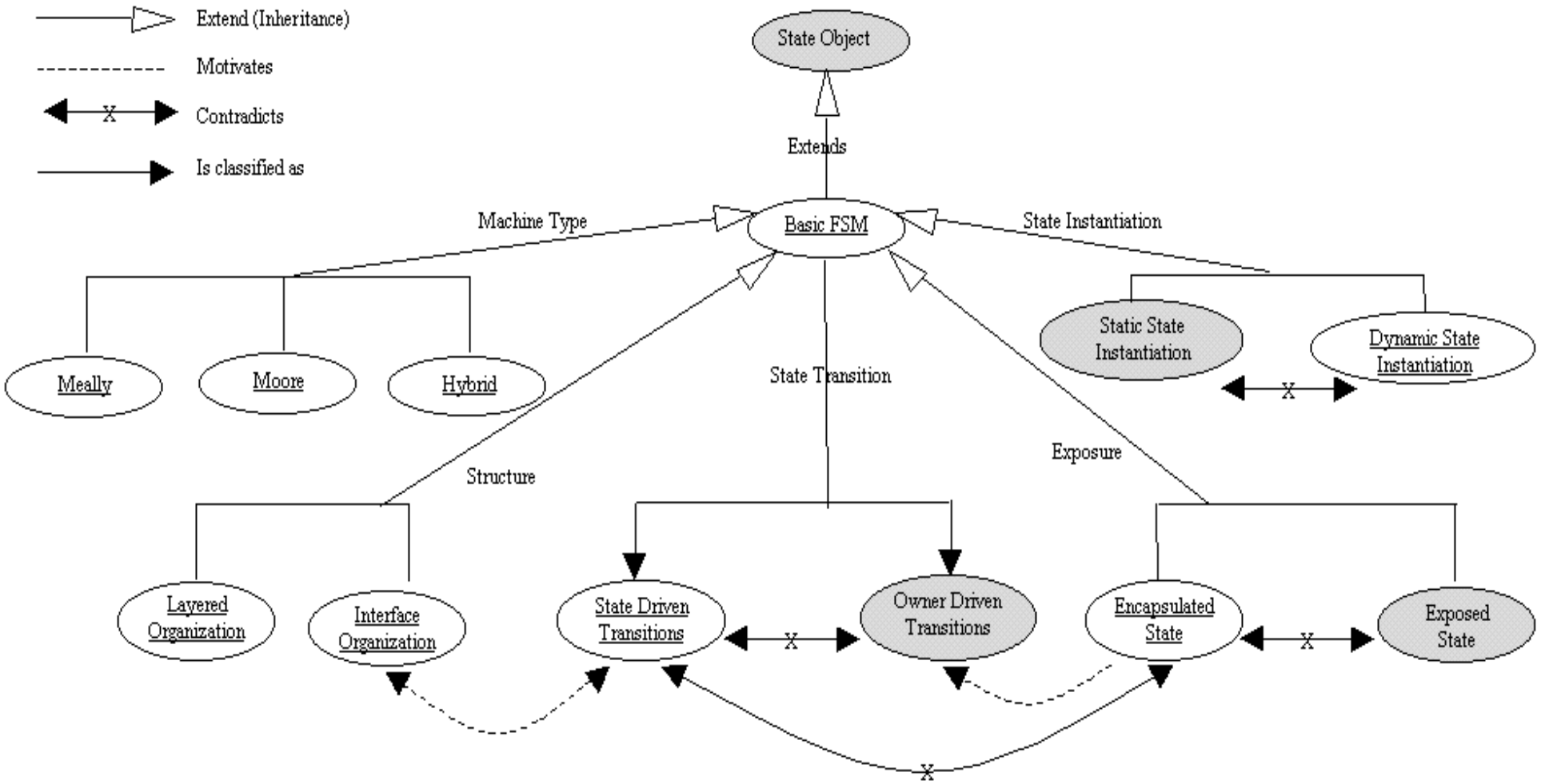
FSM Pattern Language (FSM Pattern:

Yacoub PhD Dissertation, Ch. 10, WVU, 1999)

Finite State Machine Patterns; European Pattern Languages of Programming conference, EuroPLoP (1998)

- FSM pattern language addresses several recurring design problems in implementing a finite state machine specification in an object-oriented design.
- The pattern language includes a basic design pattern for FSMs whose design evolves from the GOF State pattern.
- The basic pattern is extended to support solutions for other design problems that commonly challenge system designers.
- These design decisions include state-transition mechanisms, design structure

Pattern Language of Finite State Machines (FSM Pattern:



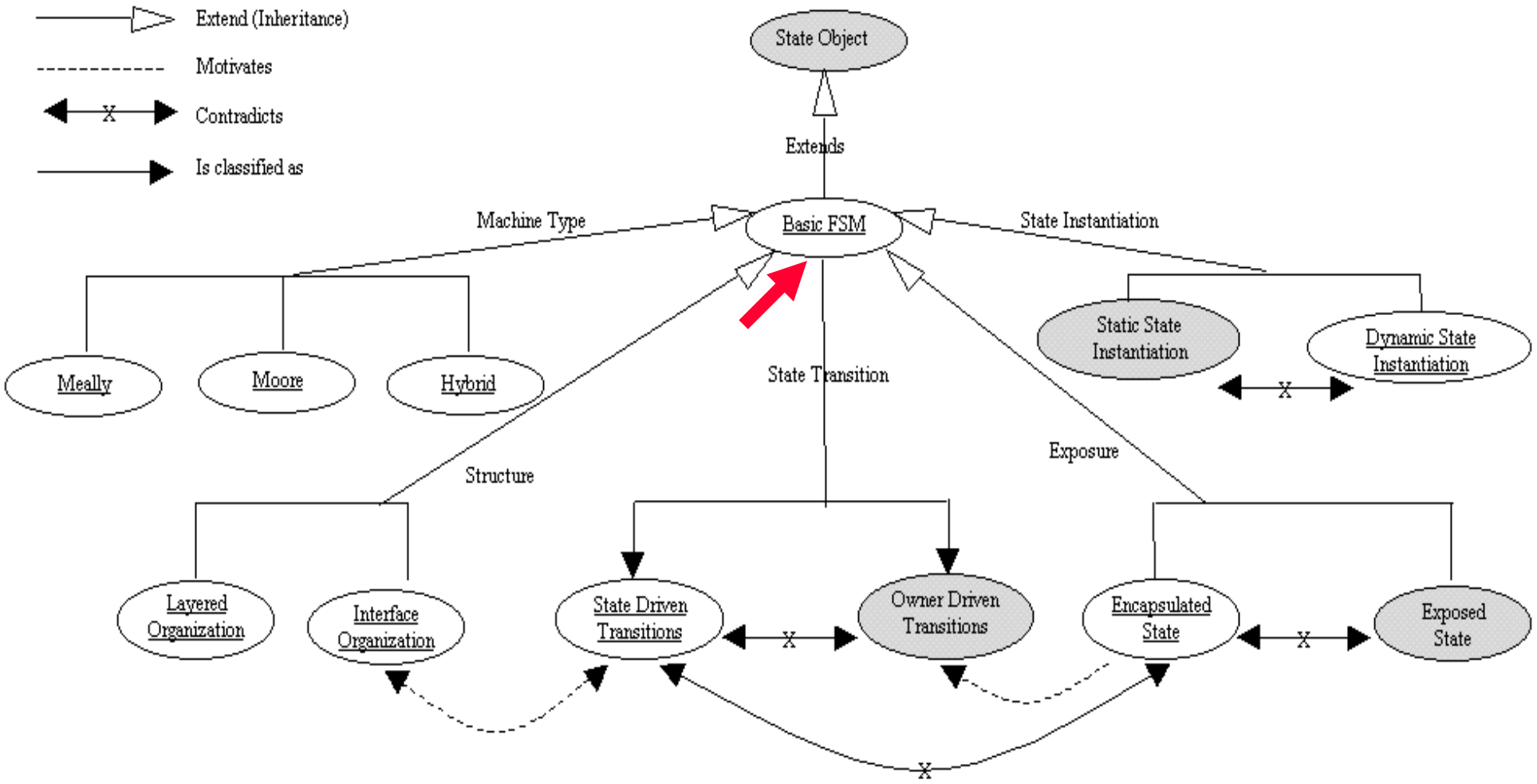
FSM Pattern Language

	Pattern Name	Problem	Solution
	State Object (GoF State Pattern)	How can you get different behavior from an entity if it differs according to the entity's state?	Create states classes for the entity, describe its behavior in each state, attach a state to the entity, and delegate the action from the entity to its current state.
Events	Basic FSM	Your entity's state changes according to events in the system. The state transitions are determined from the entity specification. How can you implement the entity behavior in your design?	Use the *State Object* pattern and add state transition mechanisms in response to state transition events. FSM pattern = State Object pattern + State Transition Mechanism
State-Transition	State-Driven Transitions	How would you implement the state transition logic but yet keep the entity class simple?	Have the states of the entity initiate the transition from self to the new state in response to the state-transition event.
	Owner-Driven Transitions	You want your states to be simple and shareable with other entities, and you want the entity to have control on its current state. How can you achieve this?	Make the entity respond to the events causing the state transitions and encapsulate the transition logic in the entity
Structure	Layered Organization	You are using an FSM pattern, how can you make your design maintainable, easily readable, and eligible for reuse?	Organize your design in a layered structure that decouples the logic of state transition from the entity's behavior, which is defined by actions and events
	Interface Organization	How can other application entities communicate and interface to an entity whose behavior is described by an FSM?	Encapsulate the states classes and state transition logic inside the machine and provide a simple interface to other application entities that receive events and dispatch them to the current state.

FSM Pattern Language (cont.)

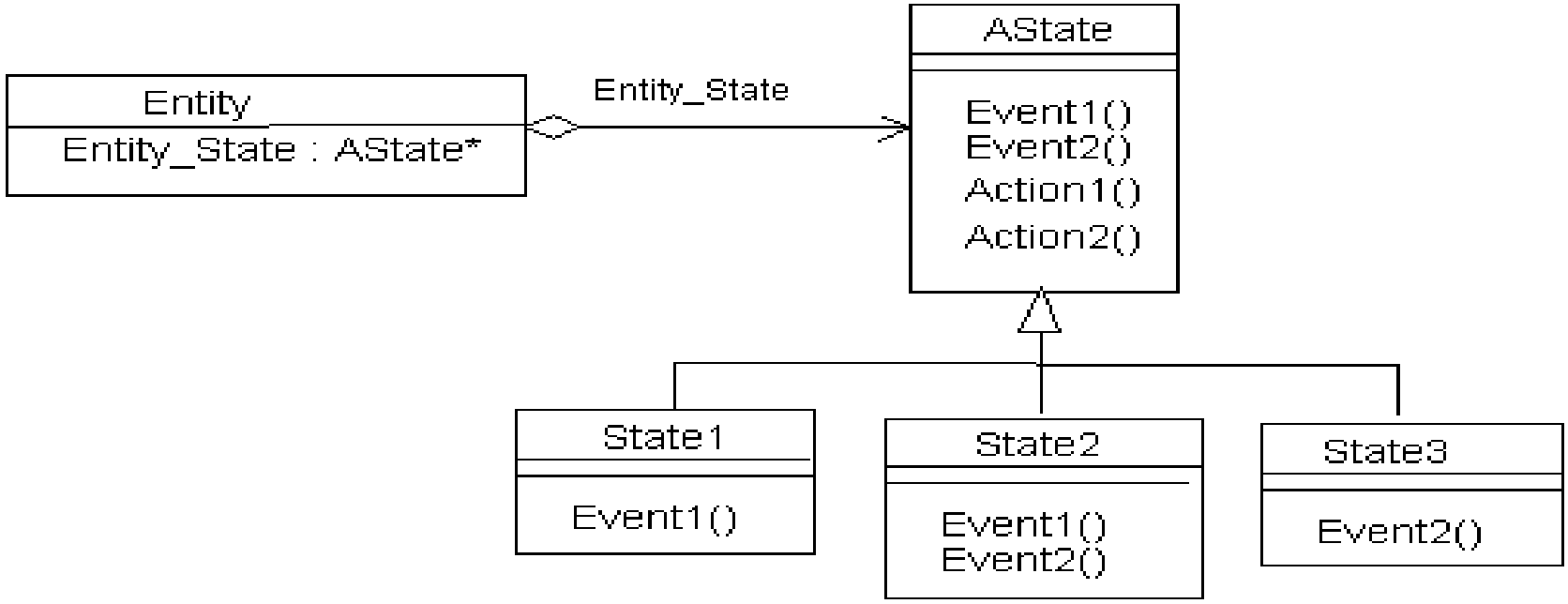
Machine Type Actions or outputs	Meally	How do you activate the FSM outputs if they should be produced at specific events while the entity is in a particular state?	Make the concrete event method of each state call the required (output) action method in response to the event.
	Moore	How do you activate the FSM outputs if they are produced only at the state and each state has a specific set of outputs?	Implement an output method in each state that calls the required actions. Make the state transition mechanism call the output method of the next upcoming state.
	Hybrid	What do you do if some FSM outputs are activated on events and some other outputs are activated as the result of being in a particular state only?	Make the event method of each state produce the event-dependent outputs, and make the state transition mechanism call an output method of the upcoming state to produce the state-dependent output.
Exposure	Exposed State	You want to allow other external entities in your application to know of your entity's state and have access to call some of the state's methods.	Provide a method that exposes the state of the entity and allows access to the current state.
	Encapsulated State	Your FSM should follow a sequence of state changes that should not be changed by other application entities. How can you ensure that no state changes are enforced to your entity?	Encapsulate the current state inside the entity itself and keep the state reference as a private attribute. Only the entity itself can change its state by handling the events causing the state change but still delegate the behavior implementation to the current state.
State Instantiation	Static State Instantiation	Your application is small and it has few states. Speed is a critical issue in state transitions. How do you instantiate your entity's states?	Create instances of all possible states on the entity instantiation. Switch from current to next state by altering the reference to the next state
	Dynamic State Instantiation	Your application is large and you have too many states. How do you instantiate the states in your application?	Don’t initially create all states; make each state knowledgeable of the next upcoming states. Create instances of upcoming states on state entry and delete them on state exit.

Pattern Language of Finite State Machines (FSM Pattern:

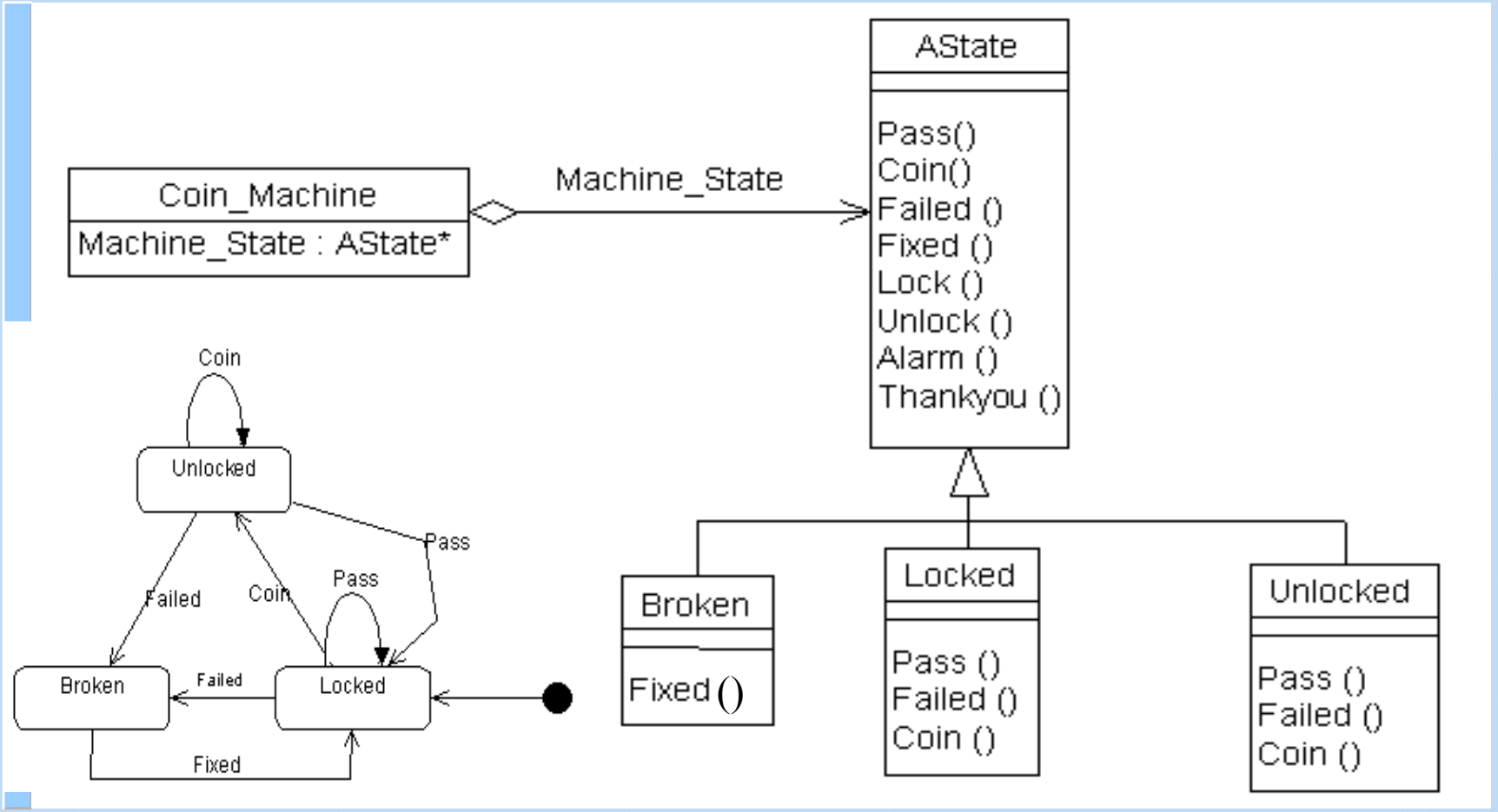


The Basic FSM Pattern Structure

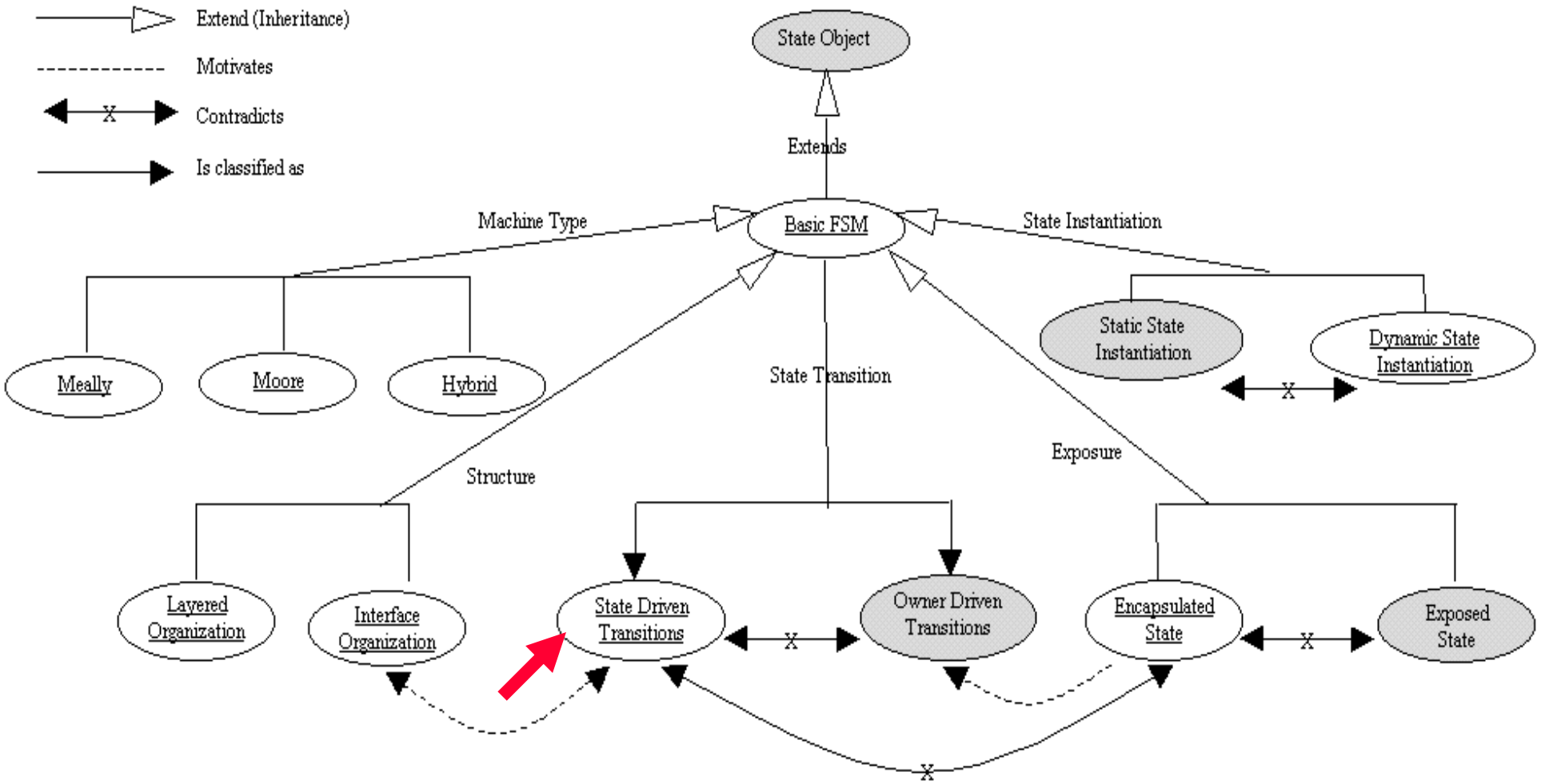
- **Context**: Your application contains an entity whose behavior depends on its state. The entity's state changes according to events in the system, and the state transitions are determined from the entity specification.
- **Problem**; How can you implement the behavior of the entity in your design?
- **Solution**: Implement Event methods in each state class



The coin machine design using the Basic FSM pattern



Pattern Language of Finite State Machines (FSM Pattern:



The State-Driven Transitions Pattern (extends Basic FSM)

- Problem: How would you implement the state transition logic but yet keep the entity class simple?
- Solution:
  - Delegates the state transition logic to the state classes, make each state knowledgeable of the next upcoming state, and have the concrete states of the entity initiate the transition from self to the new state.
  - Use the pointer to self NextStates in the abstract class AState to provide generic pointers to upcoming states.

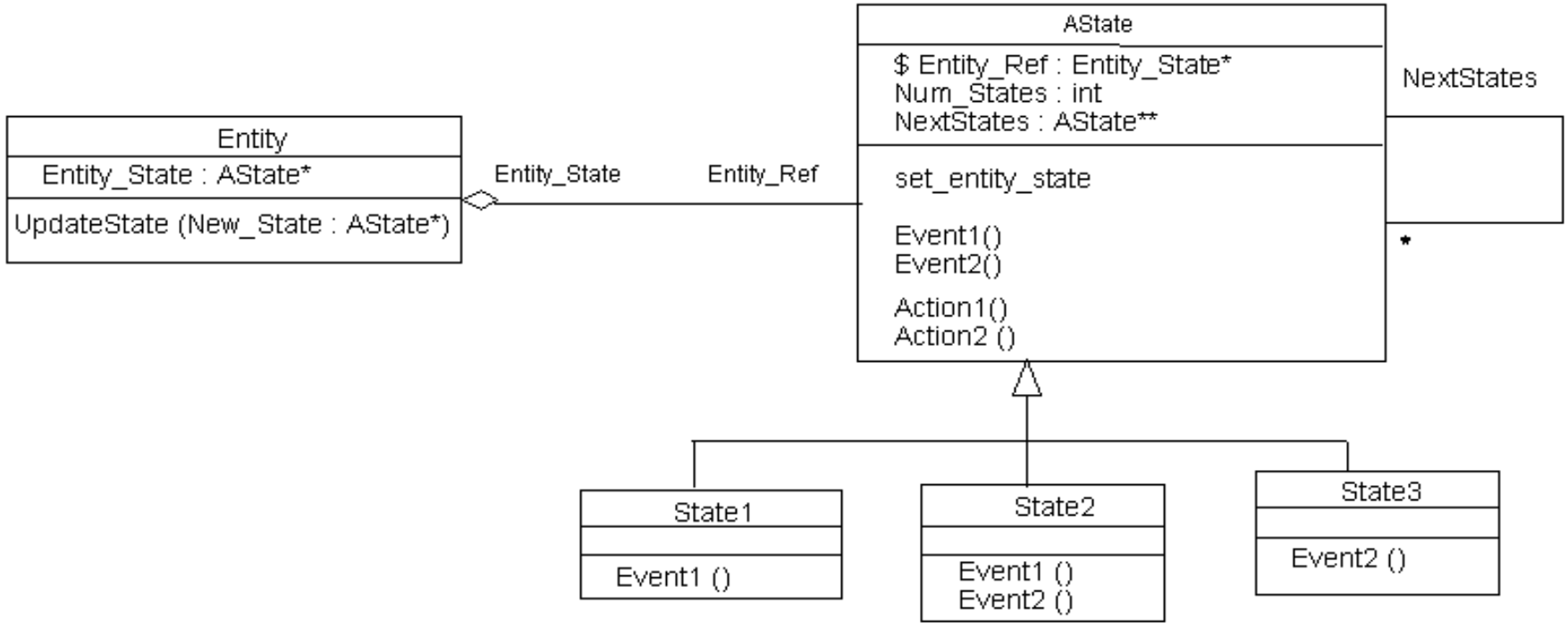
The structure of the State-Driven Transitions pattern (extends Basic FSM)

- **Context**

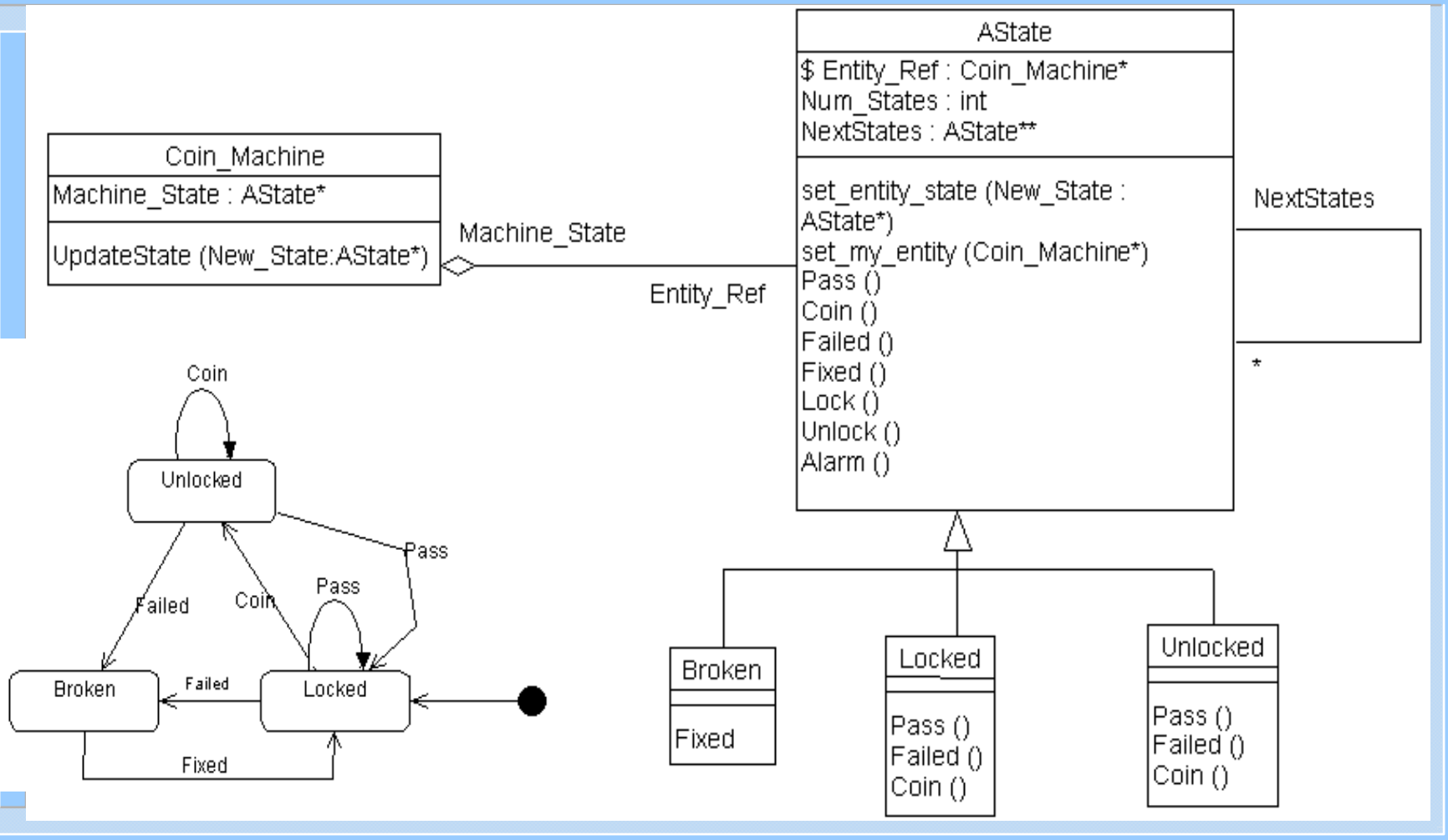
  You are using the *Basic FSM*. You need to specify a state transition mechanism to complete the entity's behavior implementation of the *Basic FSM*.

- **Problem**

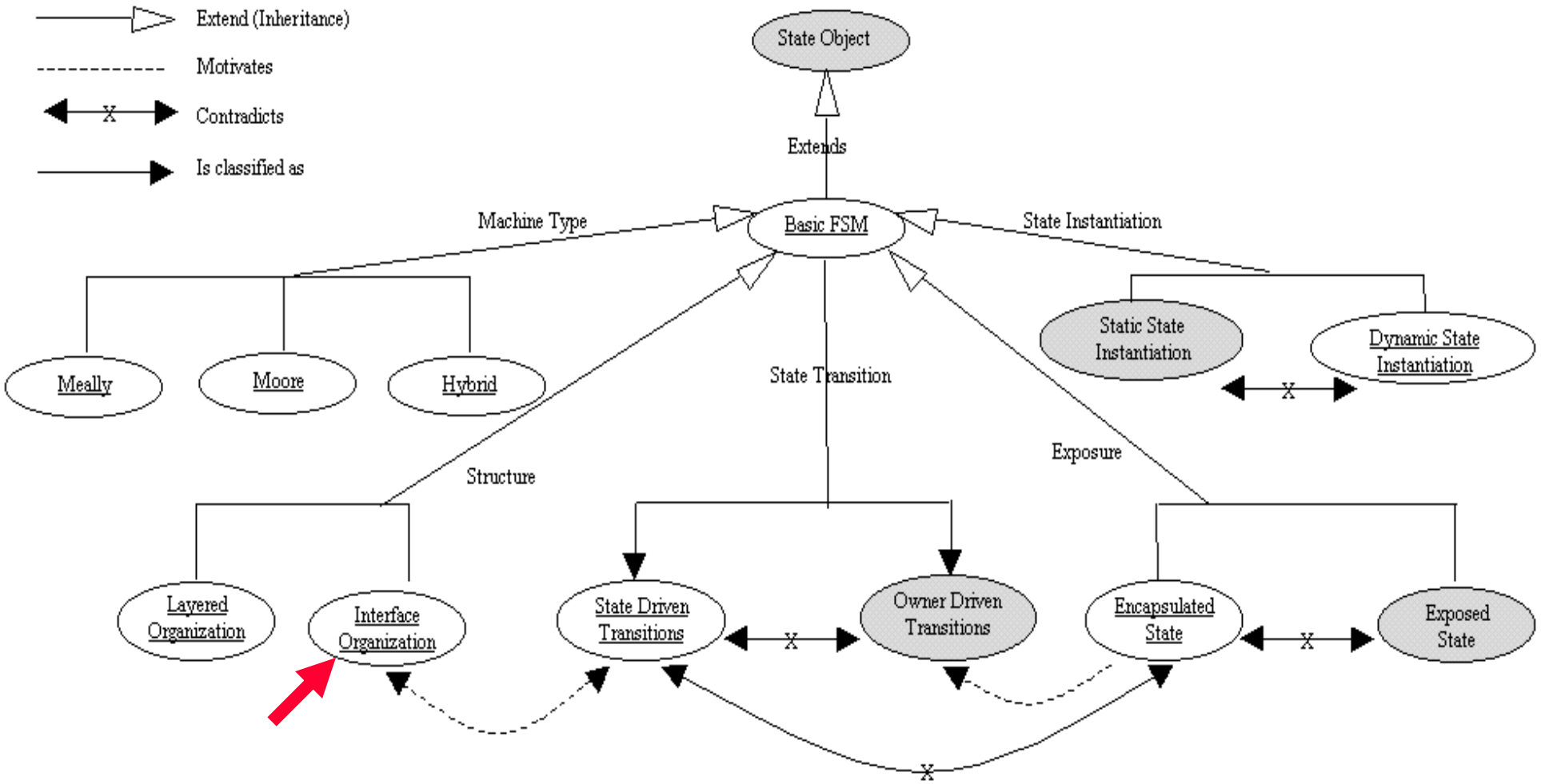
  How would you implement the state transition logic but yet keep the entity class simple?



The coin machine design using State-Driven Transitions pattern



Pattern Language of Finite State Machines (FSM Pattern:



The Interface Organization pattern

- **Context:** You are using the *Basic FSM* to implement the behavior of an entity
- **Problem**: How can other application entities communicate and interface to your entity?
- **Solution:**
  - Encapsulate the transition logic in the states and hide it from the entity interface i.e., use a state-driven transition mechanism.
  - Design the FSM to distinguish the interface that receives events and the states that handle events, invoke actions, and maintain the correct current state of the entity.

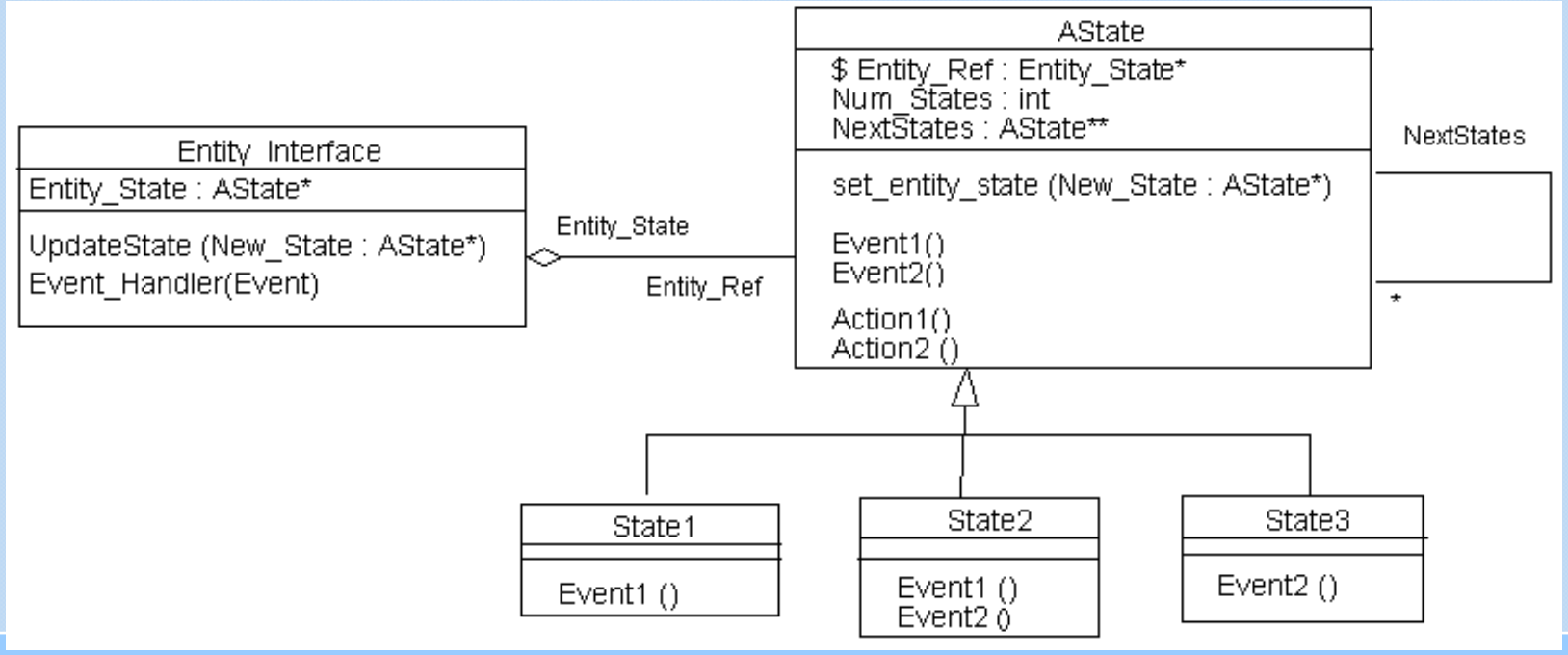
The structure of the Interface Organization pattern

- **Context**

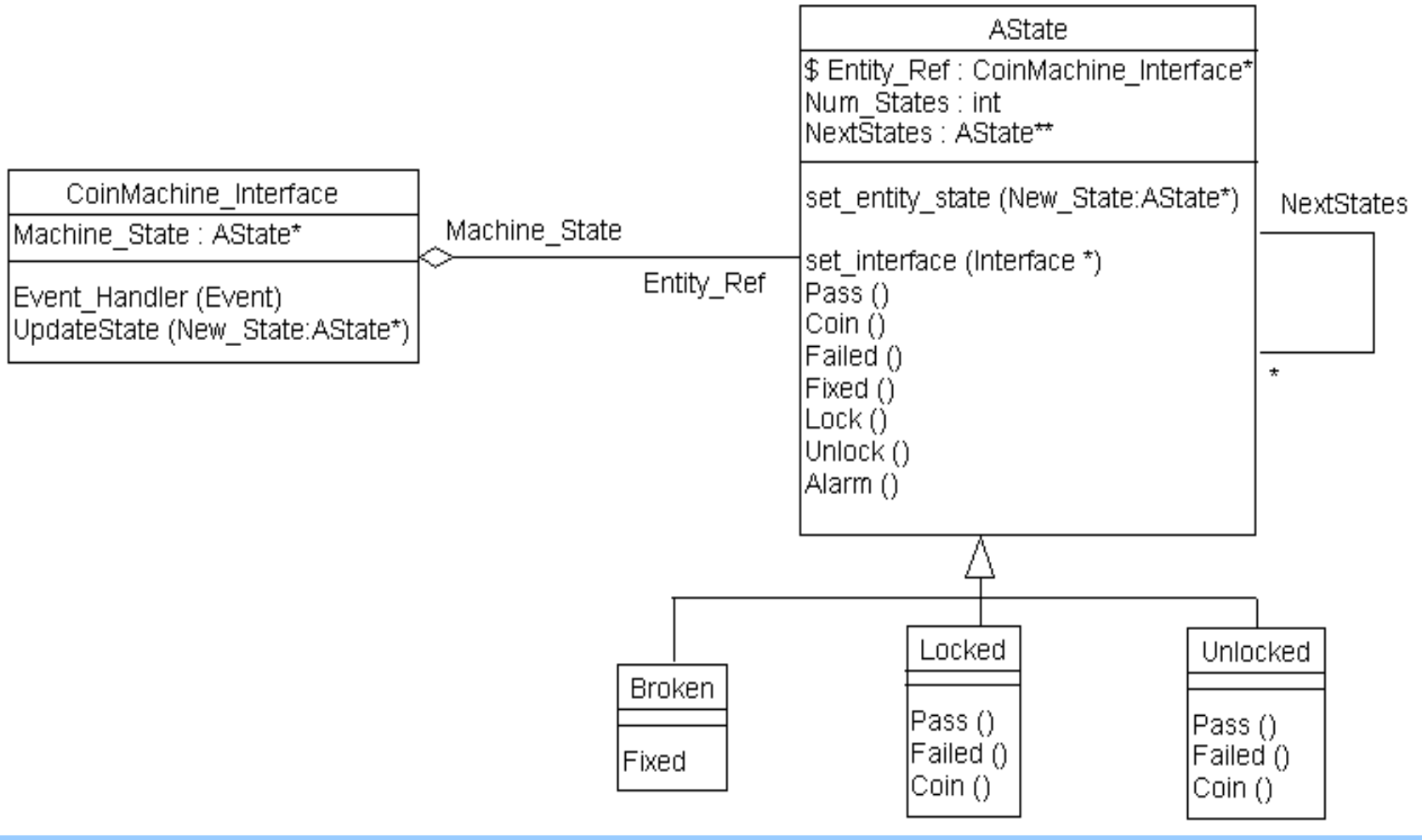
  You are using the *Basic FSM* to implement the behavior of an entity

- **Problem**

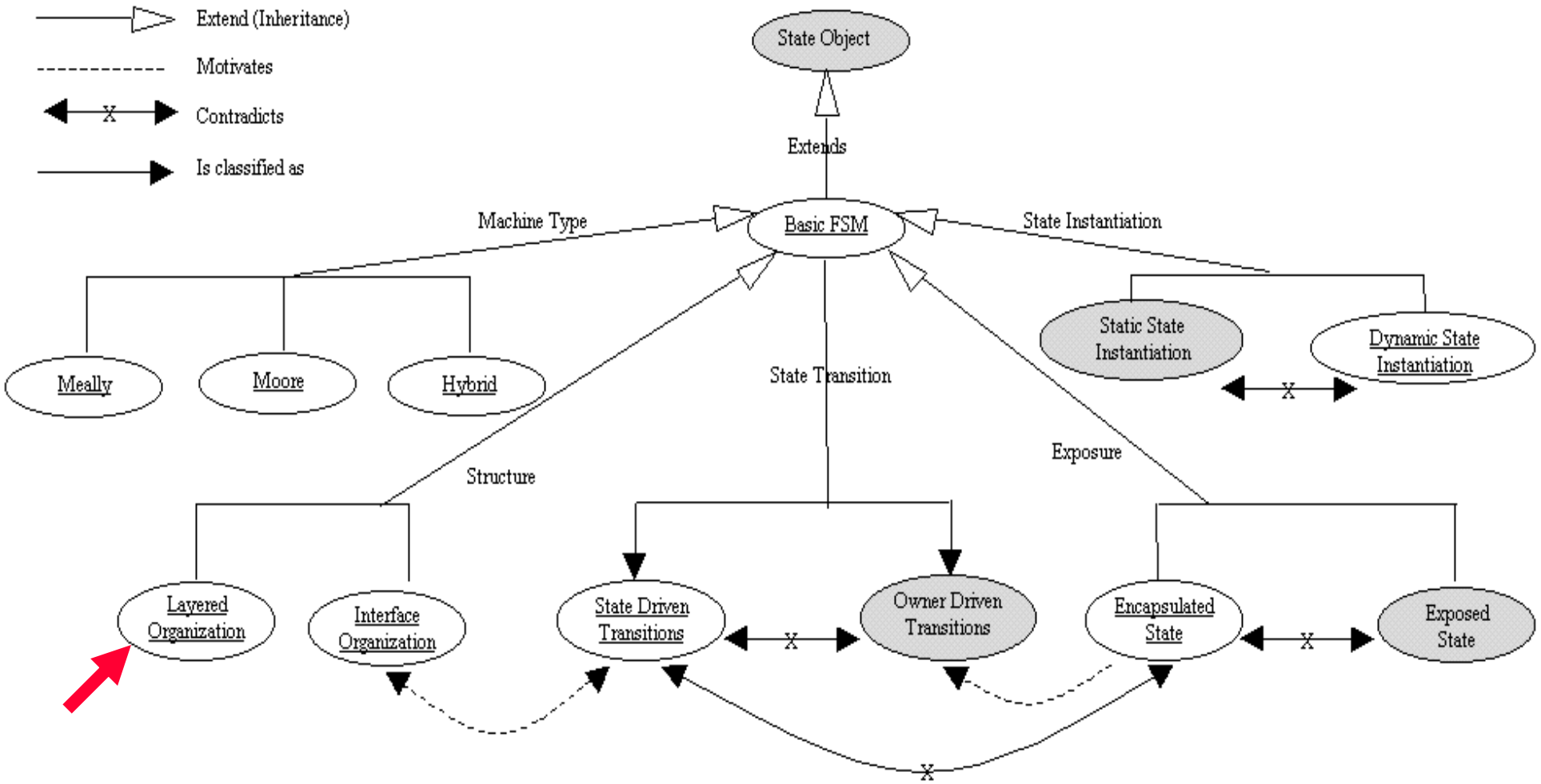
  How can other application entities communicate and interface to your entity?



The coin machine design using the Interface Organization pattern



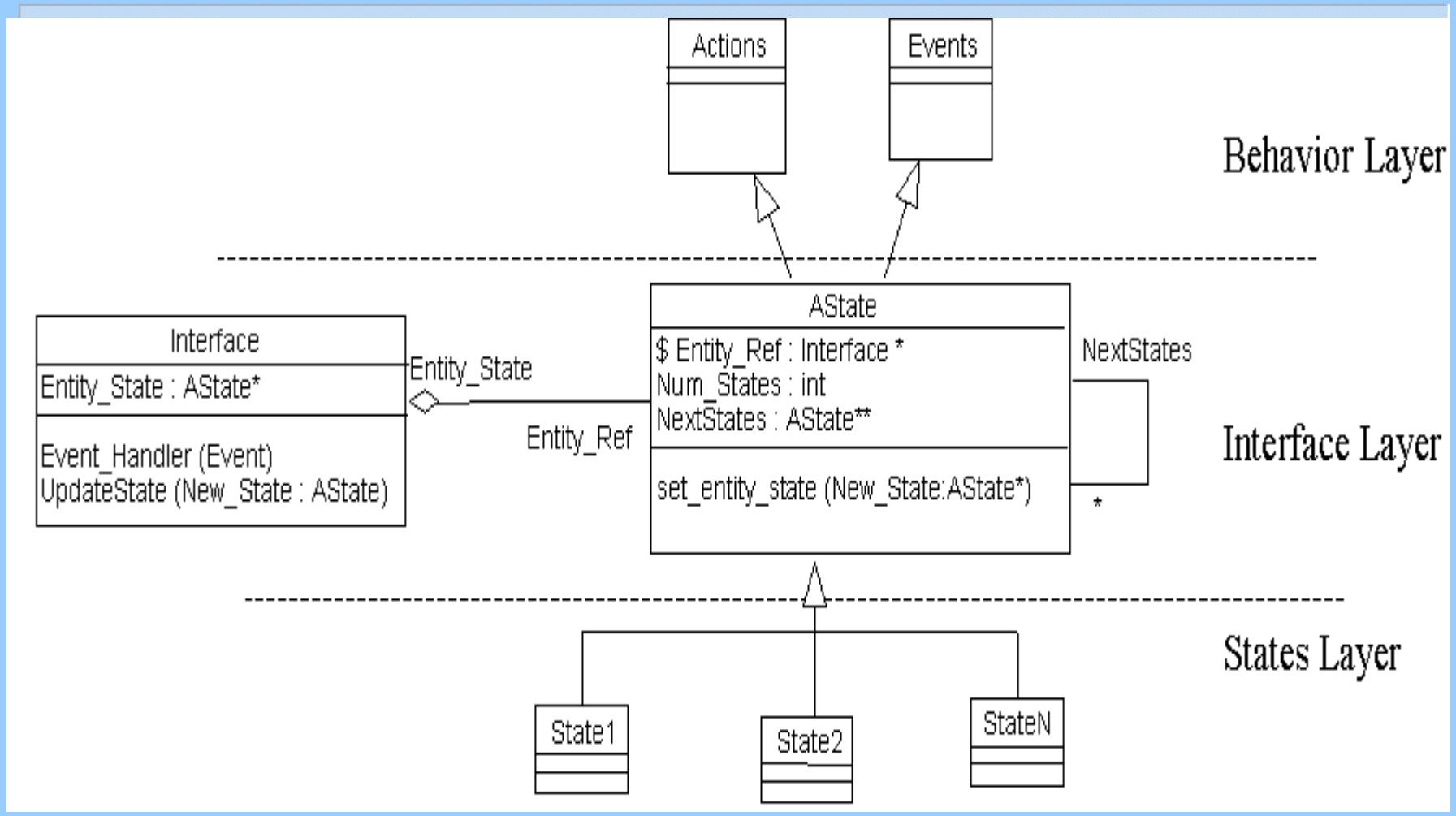
Pattern Language of Finite State Machines (FSM Pattern:



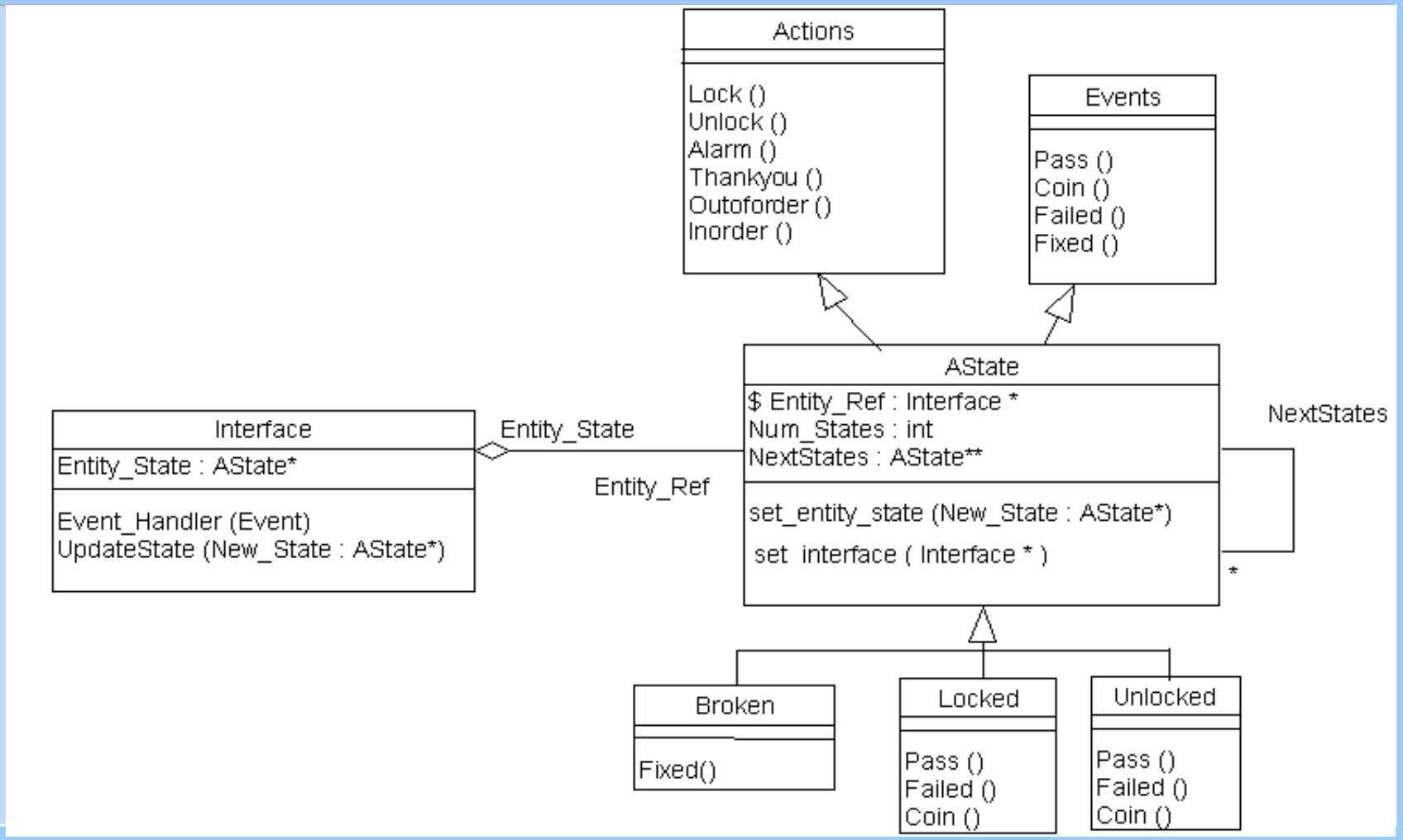
The Layered Organization Pattern

- **Context:** You are using the *Basic FSM* to implement the behavior of an entity
- **Problem**: How can you make your design maintainable, easily readable, and eligible for reuse?
- **Solution** Organize your design in a layered structure that decouples the logic of state transitions from the entity's behavior as it is defined by actions and events.

The structure of the Layered Organization Pattern



The coin machine design using the Layered Organization Pattern



Outline

- **Review**
  - The Requirements, Analysis, Design, and Design Refinement Models
  - Design refinement and Design Patterns
  - Examples of Design Patterns: The State Pattern
- **Finite State Machine Pattern Language**

  Basic FSM, State-Driven Transitions

  Interface Organization, Layered Organization
- **A Pattern Language for StateCharts**

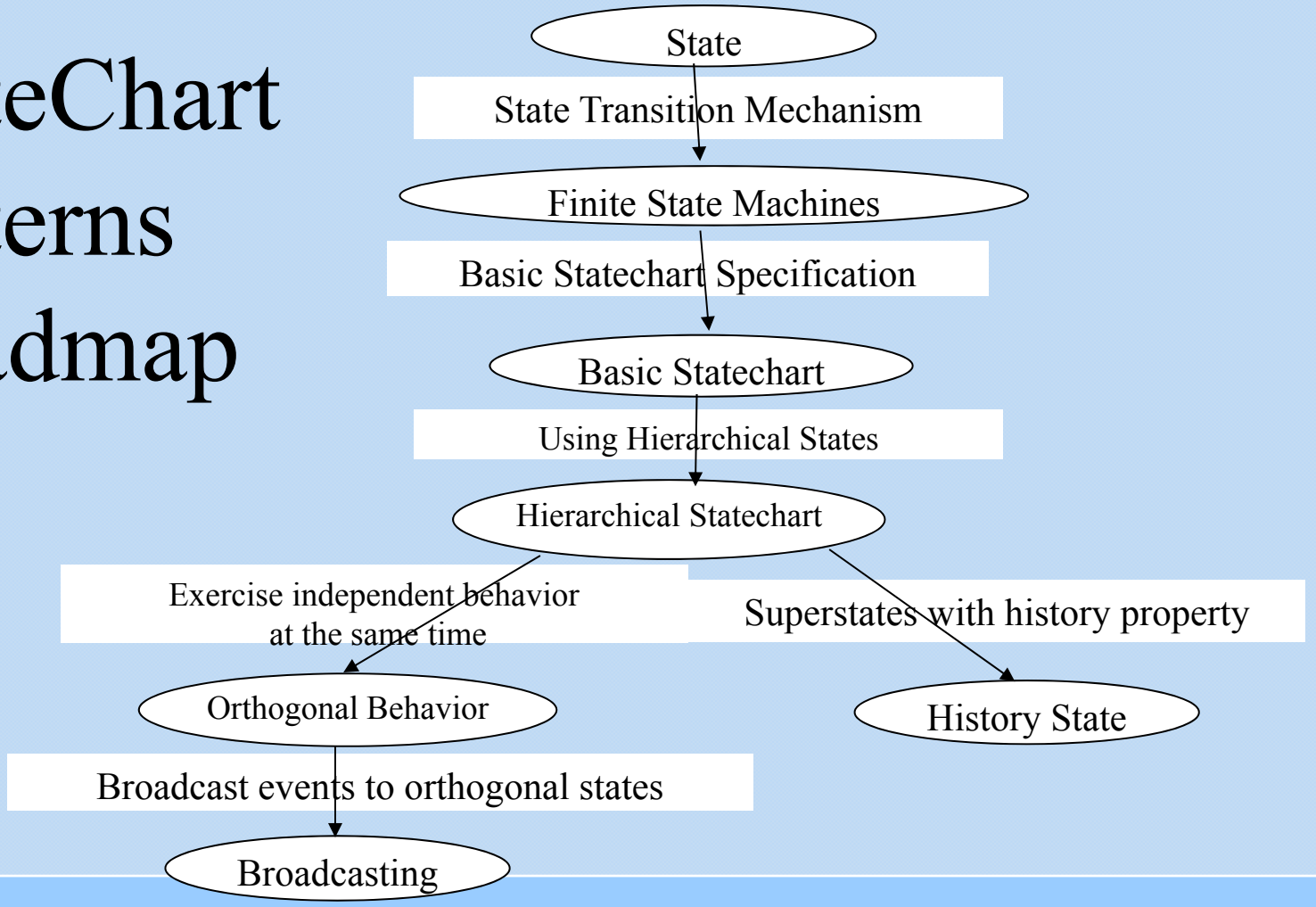
  Basic StateCharts, Hierarchical Statechart

  Orthogonal Behavior

A Pattern Language for StateCharts

Yacoub-Ammar paper, In Pattern Languages of Programs (PLOP 1998)
Yacoub PhD Dissertation, Ch. 11, WVU, 1999)

StateChart Patterns Roadmap



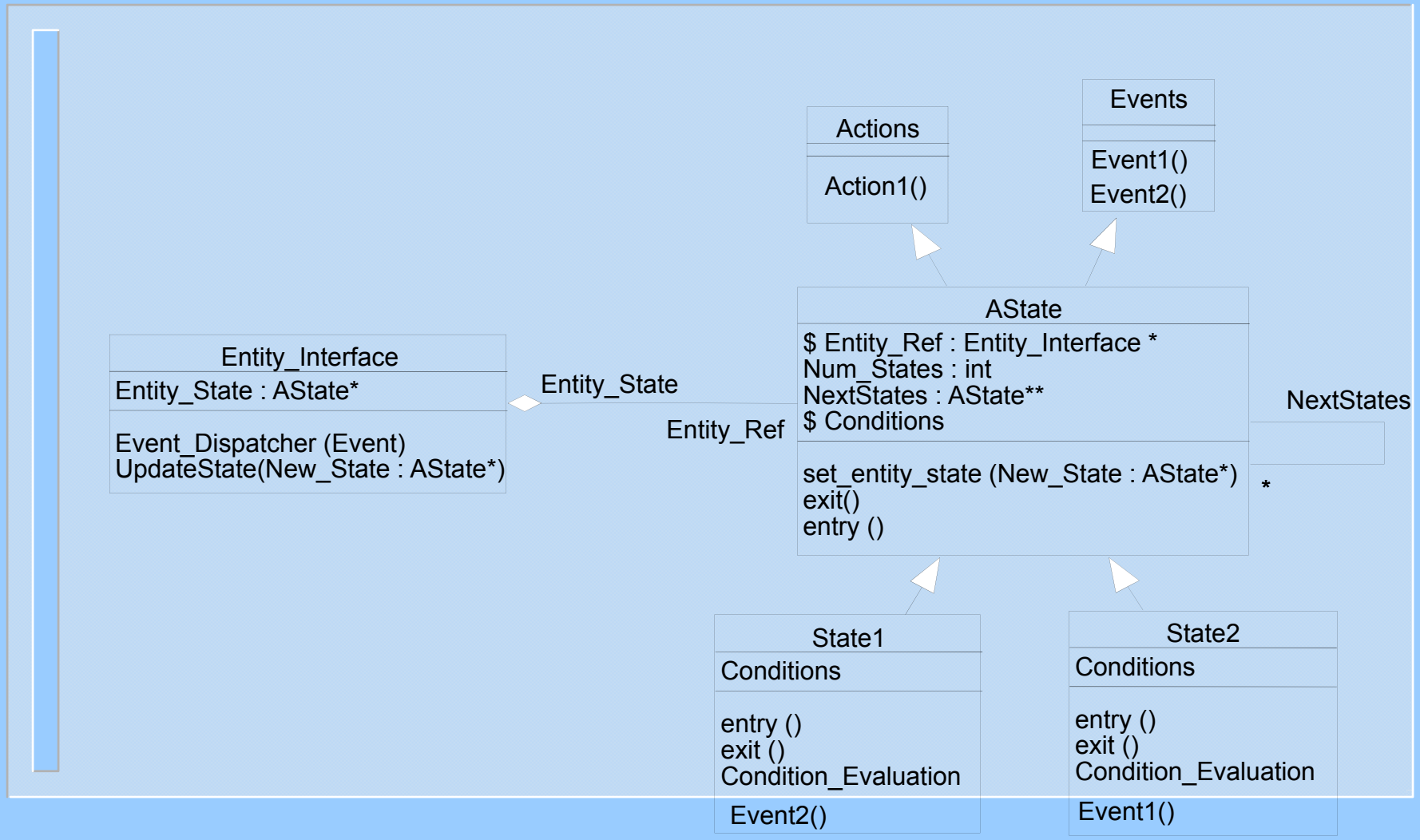
A Pattern Language for StateCharts

Pattern Name	Problem	Solution
Basic Statechart	Your application contains an entity whose behavior depends on its state. You have decided to use statechart's specifications to specify the entity's behavior. How do you implement the statechart specification into design?	Use an object oriented design that encapsulates the state of the entity into separate classes that correspond to the states defined in the specification. Distinguish the events, conditions, actions, entry and exit activities in each state class as methods and attributes of the state classes.
Hierarchical Statechart	You are using the *Basic Statechart*. The application is large and your states seem to have a hierarchical nature. How do you implement the states hierarchy in your design?	Use superstates classes that are inherited from the abstract state class. Use the *Composite* pattern [Gamma+95] to allow the superstate to contain other states. Keep the superstate knowledgeable of the current active state and dispatch events to it.
Orthogonal Behavior	You are using the *Hierarchical Statechart*. Your entity has several independent behaviors that it exercises at the same time. How do you deploy the entity's orthogonal behaviors in your design?	Identify the superstates that run independently in your specification, then define a "*Virtual superstate*" as a collection of superstates that process the same events, dispatch the events to each state.
Broadcasting	You are using the *Orthogonal Behavior*. How can you broadcast a stimulated event produced from another event occurring in an orthogonal state?	When a new event is stimulated, make the broadcasting state inject the event directly to the entity interface which dispatches it to the virtual superstate. Eventually, the virtual supertate dispatches the event to all of its orthogonal states.
History State	If one of the superstates has a history property, how do you keep its history in your design?	Initialize the current active state class pointer of the superstate object once on creation, use it throughout the entity's lifetime, and do not reinitialize it on the superstate entry method.

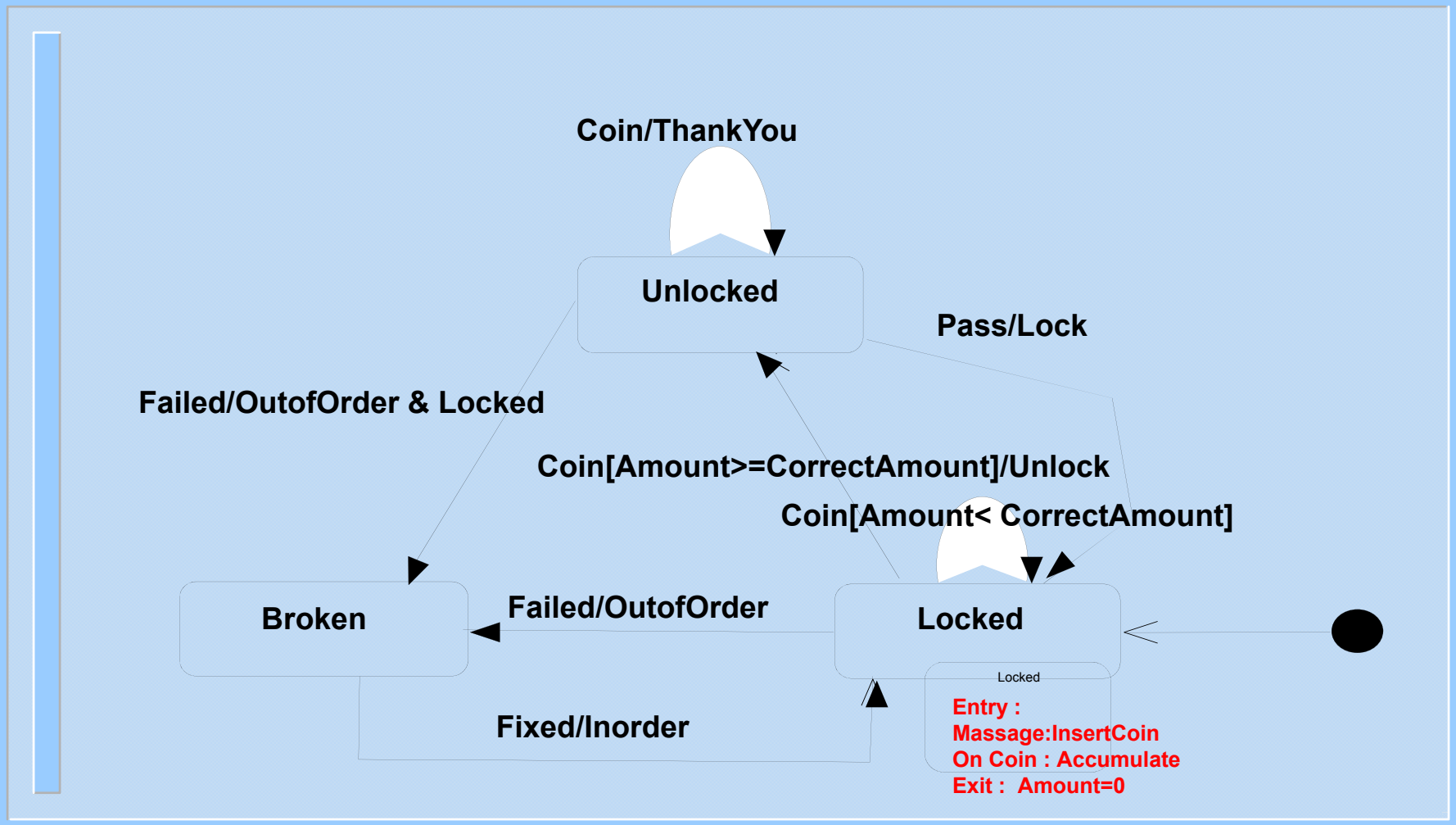
A Pattern Language for StateCharts: Basic StateCharts Pattern

- **Context** Your application contains an entity whose behavior depends on its state. You are using a statechart to specify the entity's behavior
- **Problem** How do you implement the statechart specification into design?
- **Solution:** define a state class for each entity's state defined in the specification. Distinguish the events, conditions, actions, entry and exit procedures in each state class using FSM pattern language.

The structure of the *Basic Statechart* pattern



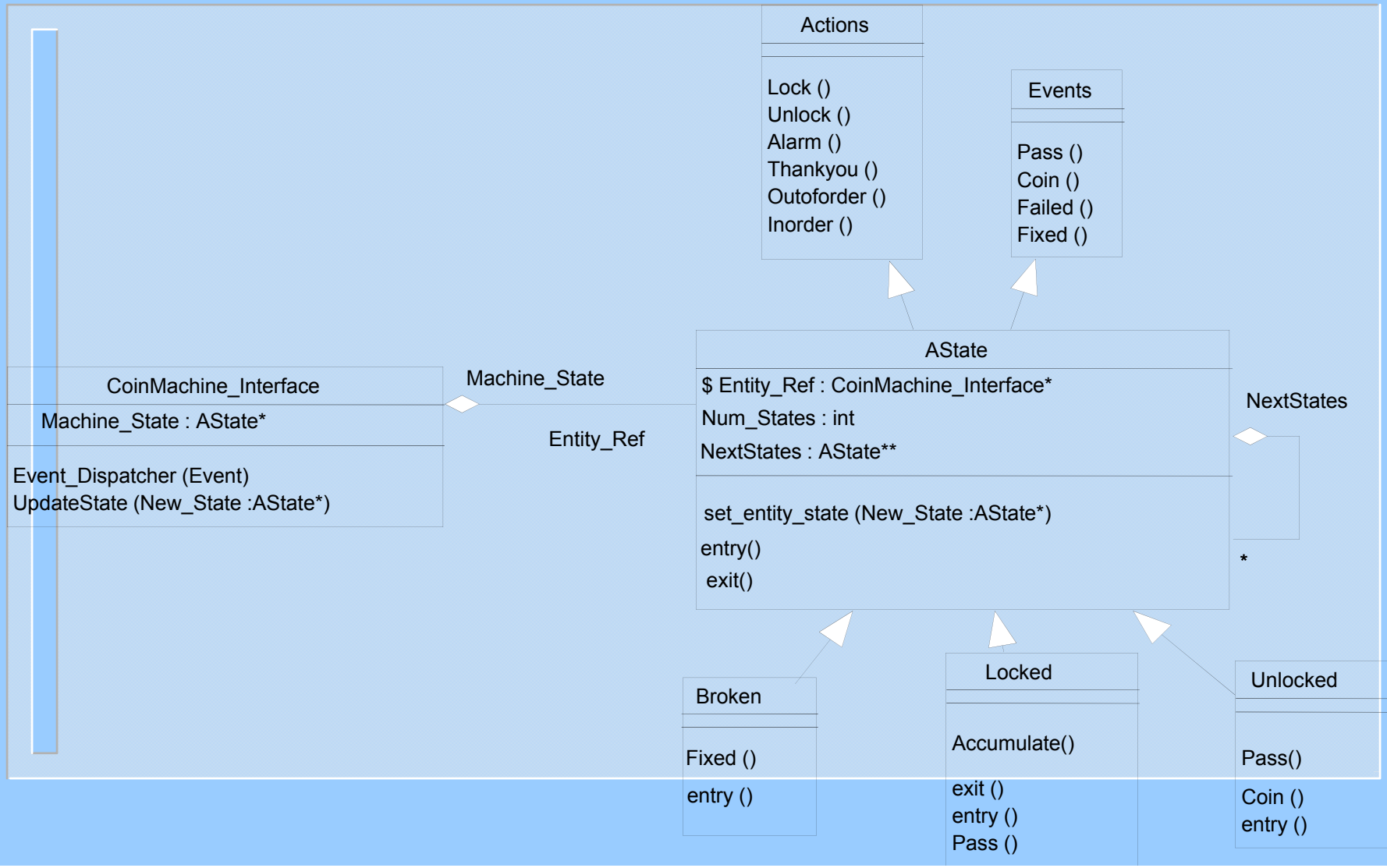
The turn style coin machine specification extended



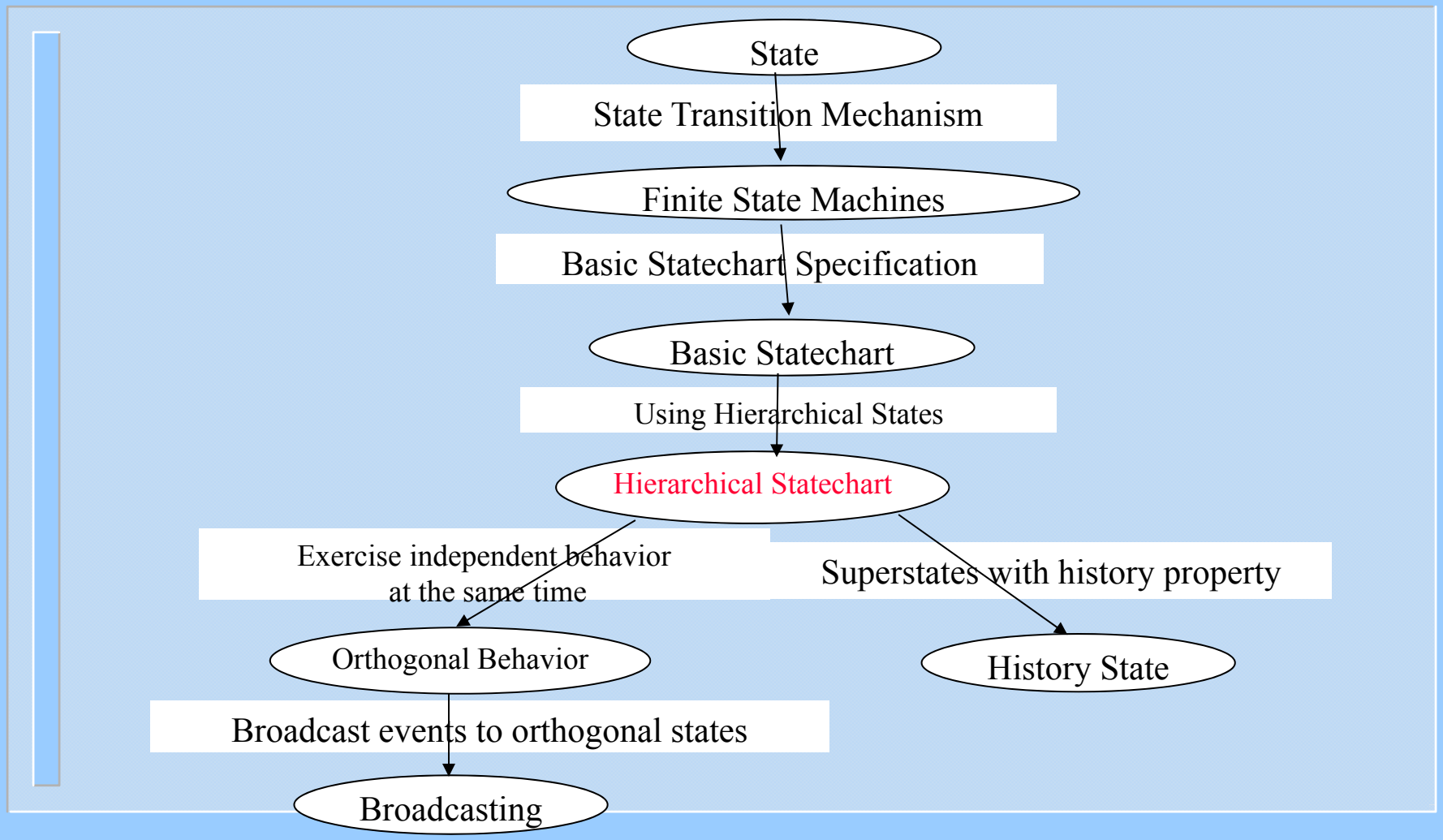
The turn style coin machine specification extended

- Implement the entry and exit specification as methods in each state class.
- For example, the coin machine should keep track of the amount of coins inserted. So, in the Locked state the machine keeps counting the amount inserted using Accumulate() method.
- On entering the Locked state the machine displays a message telling the user to insert coins to pass, thus on the entry() method the message is displayed.
- Each time the machine leaves the lock state it should clear the amount of accumulated amount to zero, thus the exit() method clears the amount.

The coin machine design using the *Basic Statechart* Pattern



A Pattern Language for StateCharts
StateChart Patterns Roadmap



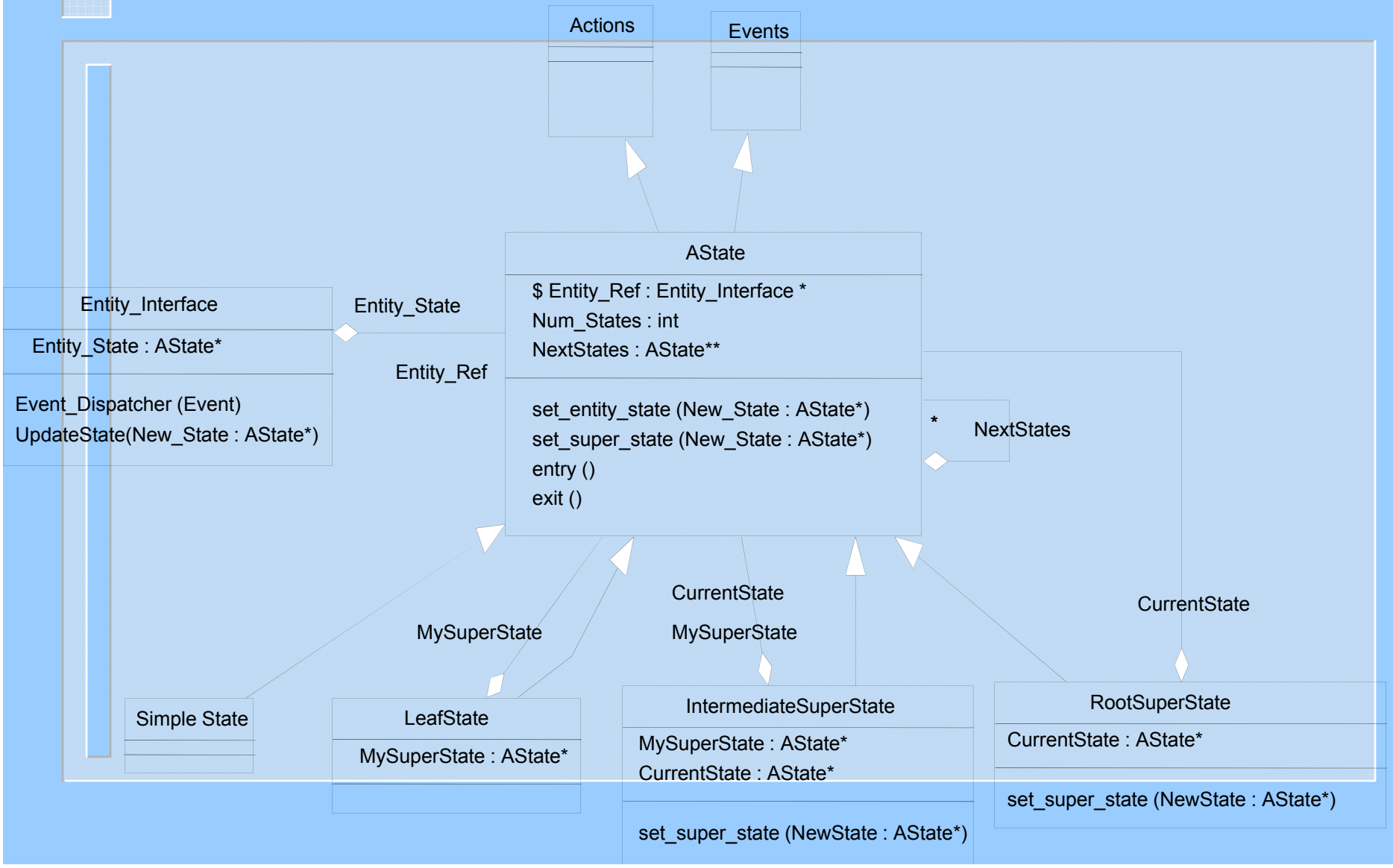
The *Hierarchical Statechart* Pattern

- **Context**: You are using the *Basic Statechart*. The application is large and the states seem to have a hierarchical nature.
- **Problem**: How do you implement the states hierarchy (Macro states) in your design?

The *Hierarchical Statechart* pattern

- **Solution:** To implement hierarchy in your design, you have to distinguish different types of states:
  - A SimpleState : a state that is not part of any superstate and doesn't contain any child state. (no parent and no children)
  - A Leaf State: a state that is child of a superstate but doesn't have any children (has a parent but has no children).
  - A Root SuperState: a state that encapsulates a group of other states (children) but has no parent.
  - An Intermediate SuperState: a state that encapsulates a group of other states (children) and has a parent state.

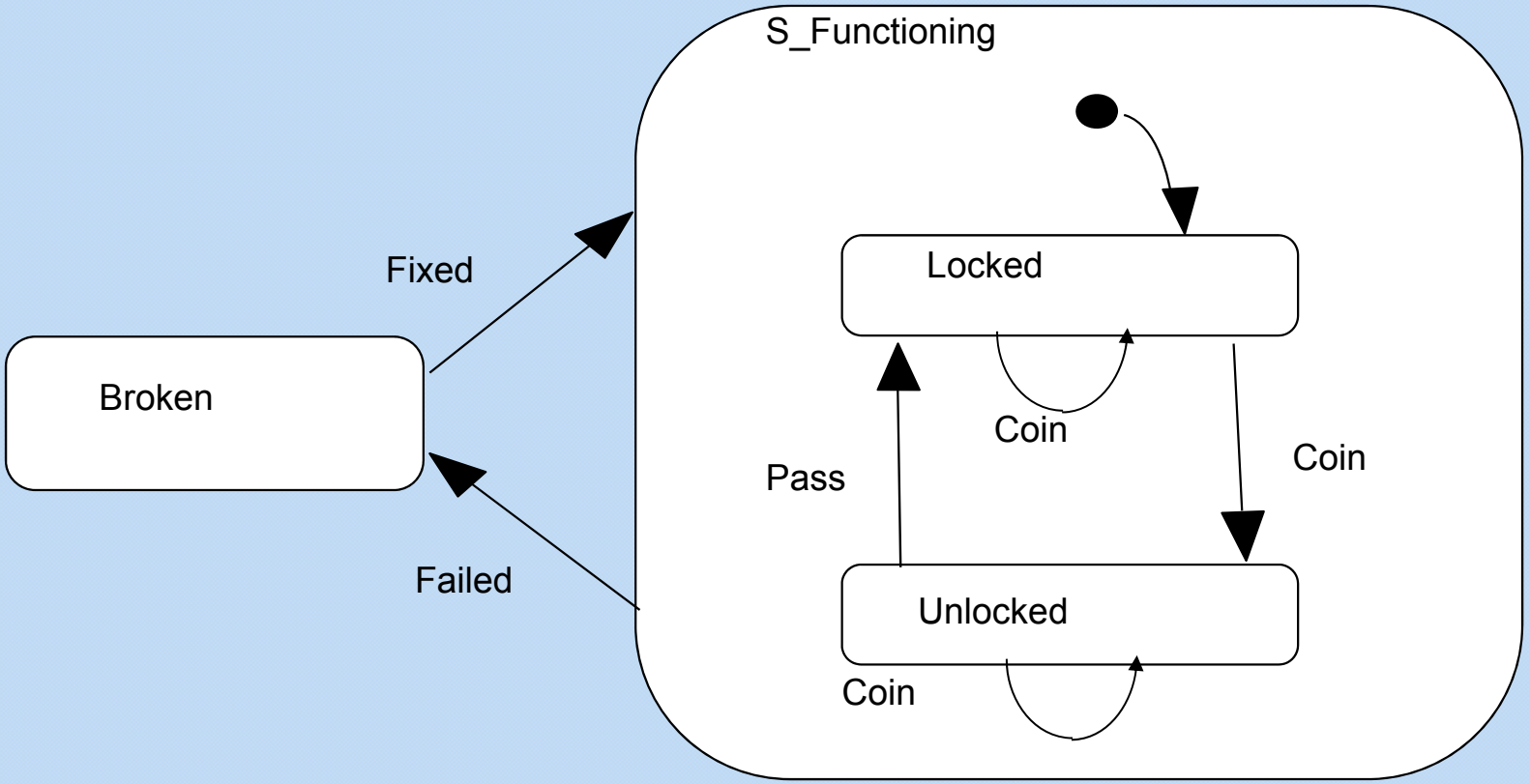
The *Hierarchical Statechart* pattern structure



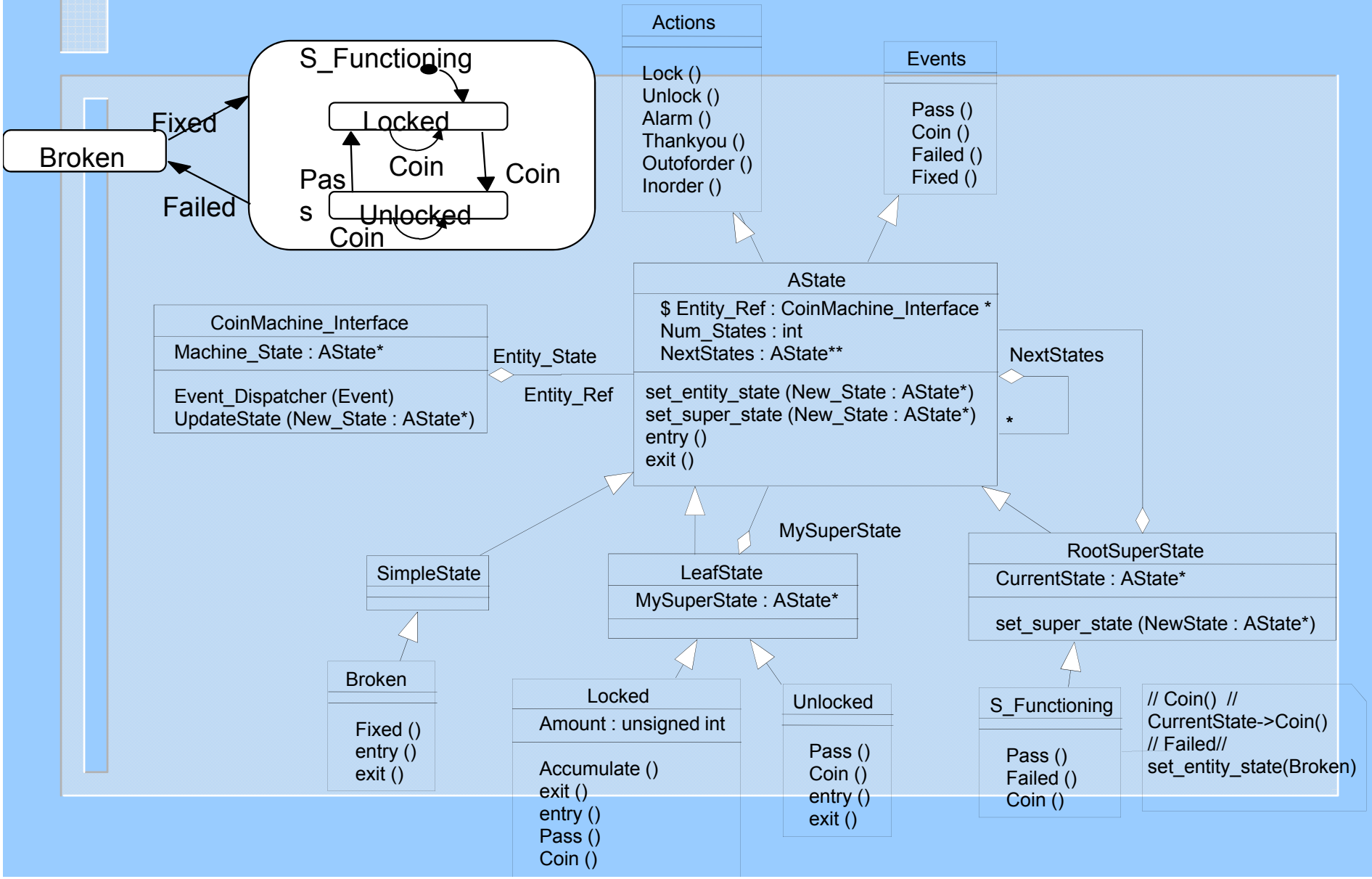
The *Hierarchical Statechart* pattern structure

- **RootSuperState**
  - Keeps track of which of its children is the current state using CurrentState
  - Handles event addressed to the group and dispatches them to the current state to respond accordingly.
  - Produces common outputs for children states, and it can also implement the common event handling methods on their behalf.
  - Performs state-driven transitions from self to the next upcoming states.
  - Implements the entry and exit methods for the whole superstate.
- **IntermediateSuperState**
  - Does the functionality of both the RootSuperState and the LeafState.
- **LeafState**
  - Does the same functionality as a SimpleState and additionally uses a MySuperState pointer to change the current active state of its parent class.

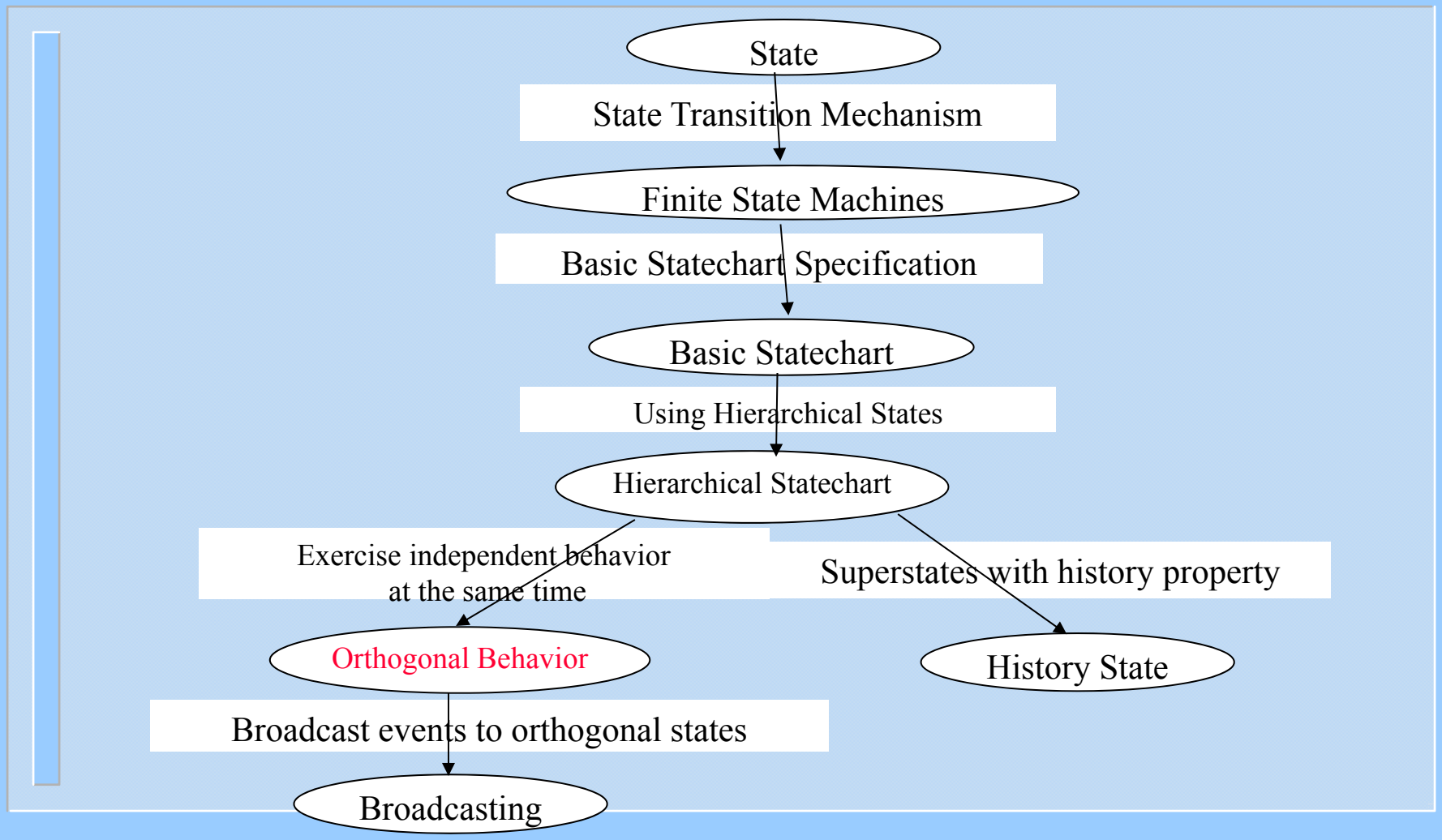
A hierarchical statechart for the coin machine example



The coin machine design using the *Hierarchical Statechart* pattern



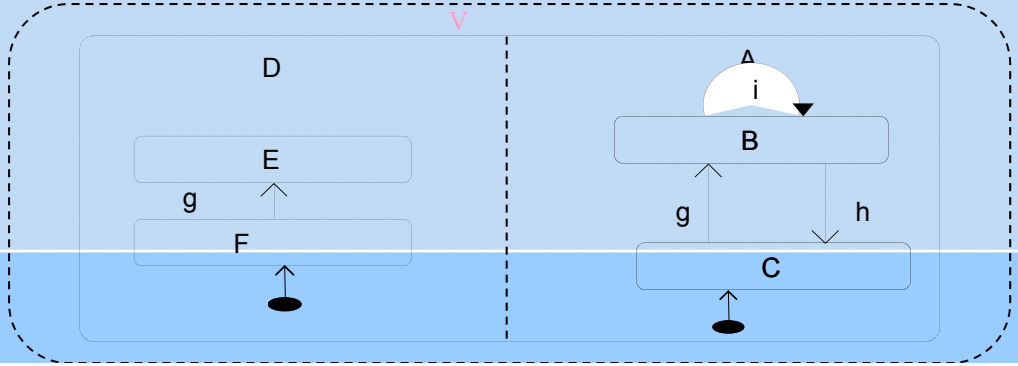
A pattern Language for StateCharts
StateChart Patterns Roadmap



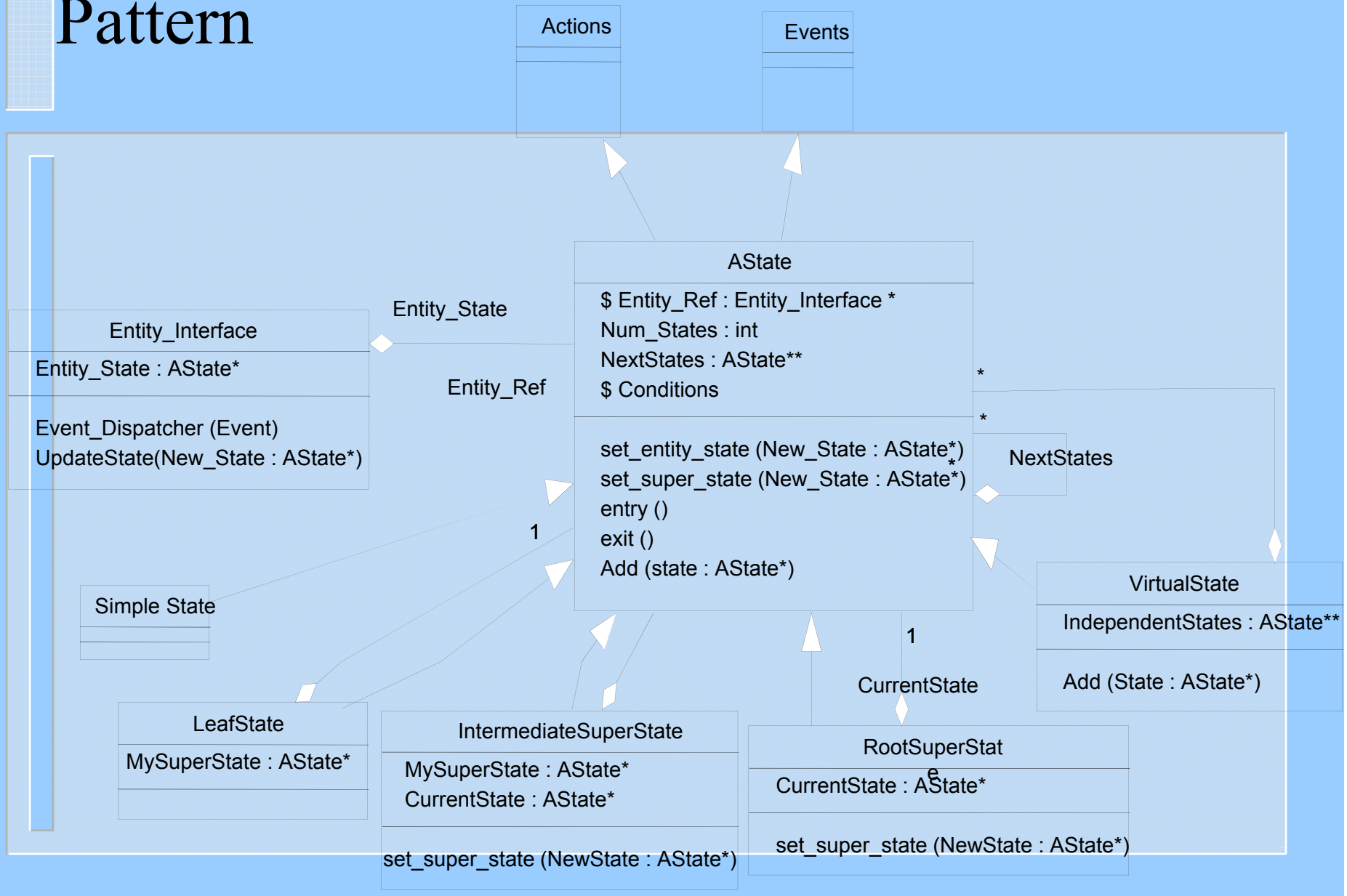
The *Orthogonal Behavior* pattern

- **Context**: You are using *Hierarchical Statechart*. Your entity has several independent behaviors that it exercises at the same time.
- **Problem**: How can you deploy the entity's orthogonal behaviors in your design?
- **Solution:**
  - identify those super states that run orthogonal (concurrently) and dispatch the events to each of those states.
  - Define a *Virtual superstate* as a collection of superstates that process same events. Then group these states in a virtual superstate whose event method will call all the event method of the attached superstates.

Example of a virtual superstate

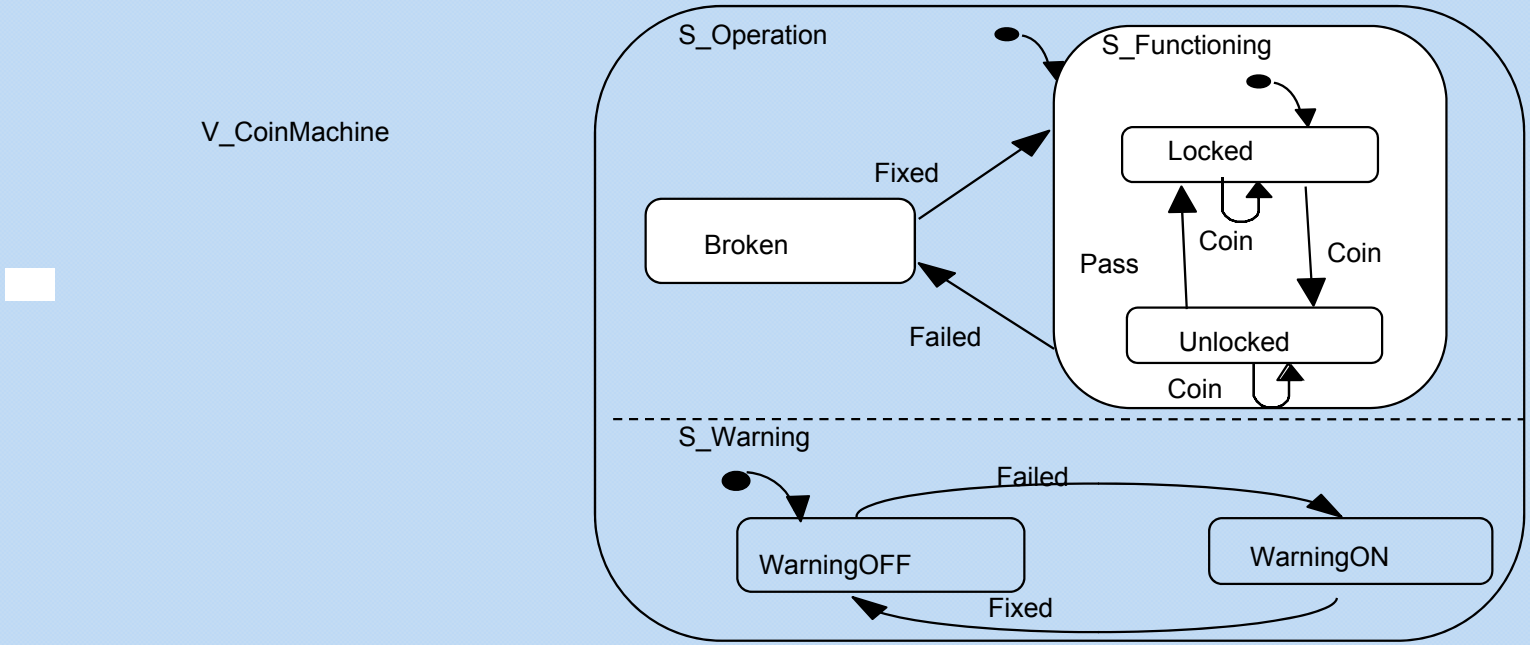


The structure of the *Orthogonal Behavior* Pattern

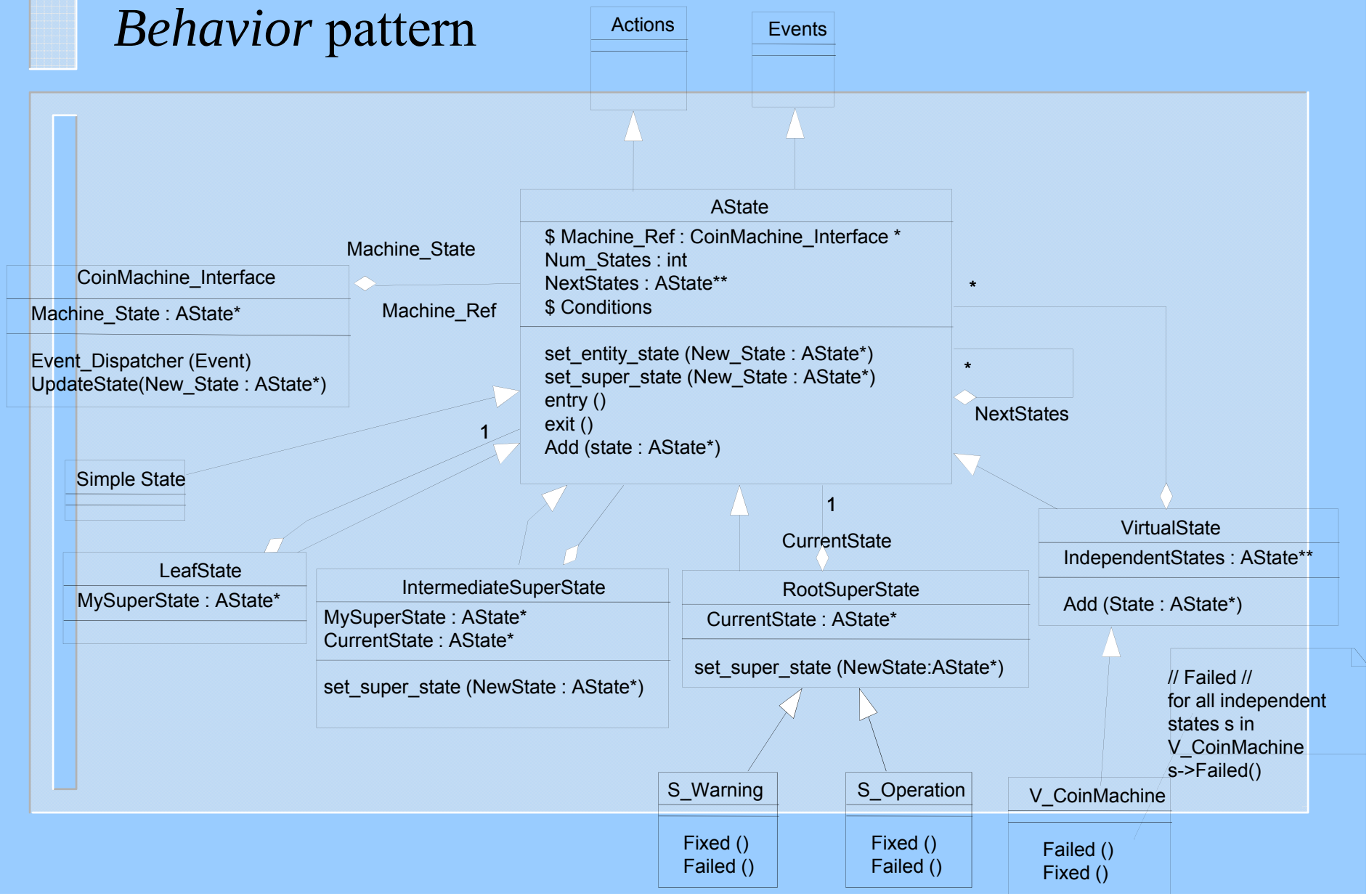


An orthogonal statechart of the coin machine

Independent behavior describing the warning and operation concurrent behavior



The coin machine design using the *Orthogonal Behavior* pattern



Outline

- **Review**
  - The Requirements, Analysis, Design, and Design Refinement Models
  - Design refinement and Design Patterns
  - Examples of Design Patterns: The State Pattern
- **Finite State Machine Pattern Language**

  Basic FSM, State-Driven Transitions

  Interface Organization, Layered Organization
- **A Pattern Language for StateCharts**

  Basic StateCharts, Hierarchical Statechart

  Orthogonal Behavior

Conclusions

- **Finite State Machine Pattern Language**

  - Presented an FSM pattern language that addresses several recurring design problems in implementing a state machine in an object-oriented design.

- **A Pattern Language for StateCharts**

  - Extended the FSM language presented above to support Statechart behavioral models.

Conclusions

These pattern languages can used to develop the Design of controller subsystems From statechart analysis models

The ATM Controller Statechart

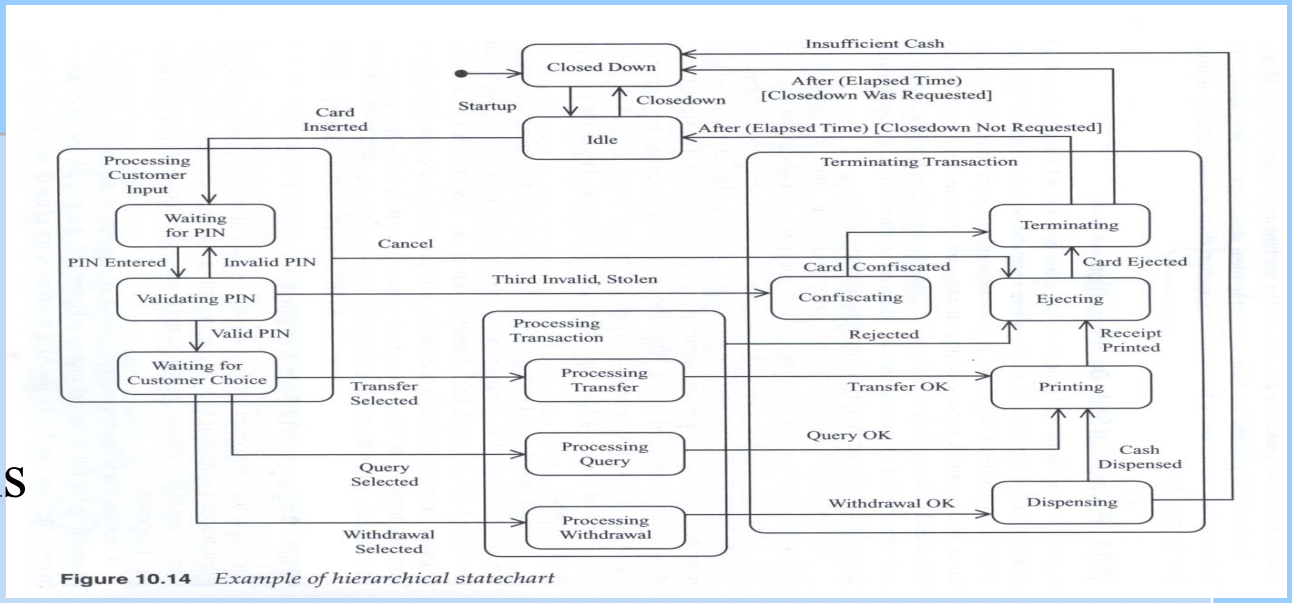


Figure 10.14 *Example of hierarchical statechart*

The Cruise Controller Statechart

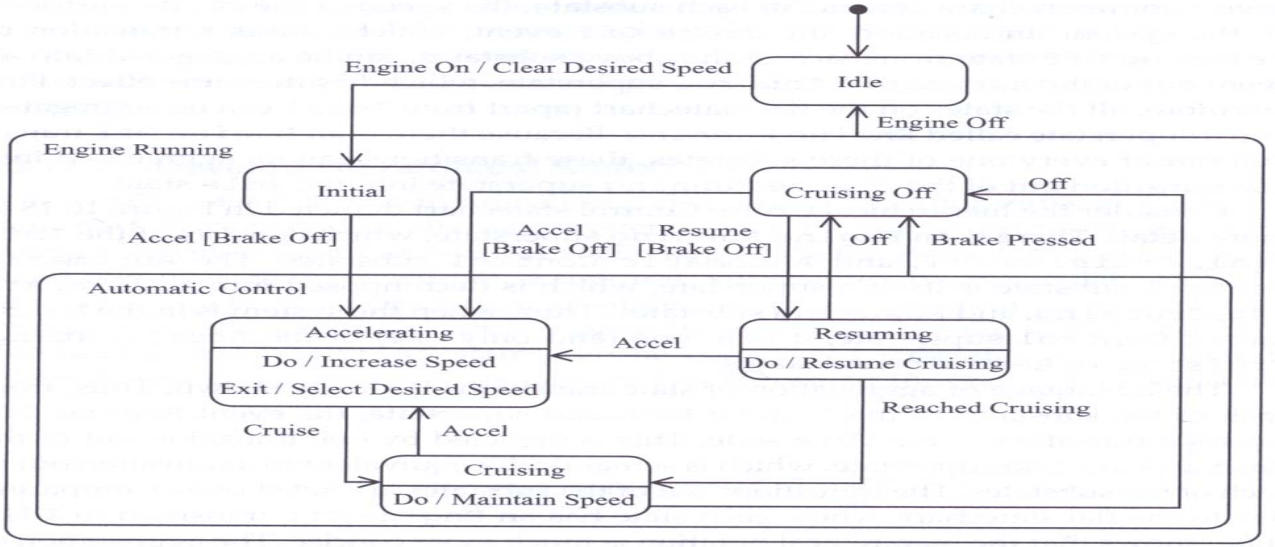


Figure 10.19 *Hierarchical Cruise Control statechart with activities and exit action*

Conclusions

This provides a way to go from Analysis directly to Design Refinement