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

Requirements Elicitation Process

Functional/Nonfunctional Requirements

Use Case Diagrams/Sequence Diagrams (the system level)
- Analysis Class Diagrams
- State Diagrams/Refined Sequence Diagrams (the object level)

The Analysis Process

Static Analysis

Static Architectural Design

- Design Class Diagrams
- Design Sequence Diagrams

Dynamic Analysis

- Refined Design Class Diagrams

The Design Process:

• Initial Design
• Design Refinement

Design Refinement
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?

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

Let's a multi state class divide its responsibilities (Opr1(), Opr2(), and Oprn()) on multiple 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

- 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:}
<table>
<thead>
<tr>
<th>FSM Pattern Language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pattern Name</strong></td>
</tr>
<tr>
<td>State Object</td>
</tr>
<tr>
<td>(GoF State Pattern)</td>
</tr>
<tr>
<td>Events</td>
</tr>
<tr>
<td>Basic FSM</td>
</tr>
<tr>
<td>State-Transition</td>
</tr>
<tr>
<td>State-Driven Transitions</td>
</tr>
<tr>
<td>Owner-Driven Transitions</td>
</tr>
<tr>
<td>Structure</td>
</tr>
<tr>
<td>Layered Organization</td>
</tr>
<tr>
<td>Interface Organization</td>
</tr>
<tr>
<td>Machine Type Actions or Outputs</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>Moore</td>
</tr>
<tr>
<td>Hybrid</td>
</tr>
<tr>
<td>Exposure</td>
</tr>
<tr>
<td>Encapsulated State</td>
</tr>
<tr>
<td>State Instantiation</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Pattern Language of Finite State Machines (FSM Pattern: [Diagram of FSM patterns and relationships])
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

[Diagram showing state transitions and actions such as Pass(), Coin(), Failed(), Fixed(), Lock(), Unlock(), Alarm(), and Thankyou()]
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
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

- State
- State Transition Mechanism
- Finite State Machines
- Basic Statechart Specification
- Basic Statechart
- Using Hierarchical States
- Hierarchical Statechart
- Exercise independent behavior at the same time
- Superstates with history property
- Orthogonal Behavior
- Broadcast events to orthogonal states
- Broadcasting
- History State
## A Pattern Language for StateCharts

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Statechart</strong></td>
<td>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?</td>
<td>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.</td>
</tr>
<tr>
<td><strong>Hierarchical Statechart</strong></td>
<td>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?</td>
<td>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.</td>
</tr>
<tr>
<td><strong>Orthogonal Behavior</strong></td>
<td>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?</td>
<td>Identify the superstates that run independently in your specification, then define a &quot;Virtual superstate&quot; as a collection of superstates that process the same events, dispatch the events to each state.</td>
</tr>
<tr>
<td><strong>Broadcasting</strong></td>
<td>You are using the Orthogonal Behavior. How can you broadcast a stimulated event produced from another event occurring in an orthogonal state?</td>
<td>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.</td>
</tr>
<tr>
<td><strong>History State</strong></td>
<td>If one of the superstates has a history property, how do you keep its history in your design?</td>
<td>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.</td>
</tr>
</tbody>
</table>
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

```
<table>
<thead>
<tr>
<th>Entity_Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity_State : AState*</td>
</tr>
<tr>
<td>Event_Dispatcher (Event)</td>
</tr>
<tr>
<td>UpdateState(New_State : AState*)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entity_State</th>
</tr>
</thead>
<tbody>
<tr>
<td>AState</td>
</tr>
<tr>
<td>$ Entity_Ref : Entity_Interface *</td>
</tr>
<tr>
<td>Num_States : int</td>
</tr>
<tr>
<td>NextStates : AState**</td>
</tr>
<tr>
<td>$ Conditions</td>
</tr>
<tr>
<td>set_entity_state (New_State : AState*)</td>
</tr>
<tr>
<td>entry()</td>
</tr>
<tr>
<td>exit()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditions</td>
</tr>
<tr>
<td>entry ()</td>
</tr>
<tr>
<td>exit ()</td>
</tr>
<tr>
<td>Condition_Evaluation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditions</td>
</tr>
<tr>
<td>entry ()</td>
</tr>
<tr>
<td>exit ()</td>
</tr>
<tr>
<td>Condition_Evaluation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event1()</td>
</tr>
<tr>
<td>Event2()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action1()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NextStates</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
</tr>
</tbody>
</table>
```
The turn style coin machine specification extended

- **Entry**: Massage:InsertCoin
- **On Coin**: Accumulate
- **Exit**: Amount=0

**States**:
- **Unlocked**
  - Coin/ThankYou
  - Pass/Lock
  - Coin[Amount>=CorrectAmount]/Unlock
  - Coin[Amount< CorrectAmount]

- **Fixed/Inorder**
  - Failed/OutofOrder & Locked
  - Broken

- **Failed/OutofOrder**
  - Locked

- **Locked**
  - Passed

- **Locked**
  - Unlocked
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

<table>
<thead>
<tr>
<th>Actions</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock ()</td>
<td>Pass ()</td>
</tr>
<tr>
<td>Unlock ()</td>
<td>Coin ()</td>
</tr>
<tr>
<td>Alarm ()</td>
<td>Failed ()</td>
</tr>
<tr>
<td>Thankyou ()</td>
<td>Fixed ()</td>
</tr>
<tr>
<td>Outoforder ()</td>
<td></td>
</tr>
<tr>
<td>Inorder ()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Breakdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoinMachine_Interface</td>
</tr>
<tr>
<td>Machine_State</td>
</tr>
<tr>
<td>Entity_Ref</td>
</tr>
<tr>
<td>Event_Dispatcher (Event)</td>
</tr>
<tr>
<td>UpdateState (New_State : AState*)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AState</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ Entity_Ref : CoinMachine_Interface*</td>
</tr>
<tr>
<td>Num_States : int</td>
</tr>
<tr>
<td>NextStates : AState**</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NextStates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broken</td>
</tr>
<tr>
<td>Fixed ()</td>
</tr>
<tr>
<td>entry ()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Locked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulate()</td>
</tr>
<tr>
<td>exit ()</td>
</tr>
<tr>
<td>entry ()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unlocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass()</td>
</tr>
<tr>
<td>Coin () entry ()</td>
</tr>
</tbody>
</table>

Machine_State : AState*
Event_Dispatcher (Event) UpdateState (New_State : AState*)
A Pattern Language for StateCharts

StateChart Patterns Roadmap

- State Transition Mechanism
- Finite State Machines
- Basic Statechart Specification
  - Basic Statechart
  - Using Hierarchical States
    - Hierarchical Statechart
      - Exercise independent behavior at the same time
      - Superstates with history property
      - Orthogonal Behavior
      - Broadcast events to orthogonal states
      - Broadcasting
      - History State
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

### Actions
- Lock()
- Unlock()
- Alarm()
- Thankyou()
- Outoforder()
- Inorder()

### Events
- Pass()
- Coin()
- Failed()
- Fixed()

### Statechart Diagram:
- **Machine State**: AState*
- **Event Dispatcher**: Event_Dispatcher (Event)
- **Update State**: UpdateState (New_State : AState*)

#### Leaf State:
- **S_Functioning**:
  - Locked
  - Unlocked

#### MySuperState:
- **CoinMachine_Interface**: CoinMachine_Interface*
- **Machine State**: AState*
- **Event Dispatcher**: Event_Dispatcher (Event)
- **Update State**: UpdateState (New_State : AState*)

#### AState:
- $Entity_Ref : CoinMachine_Interface*
- Num_States : int
- NextStates : AState**

#### Entity State:
- **Entity_Ref**: Entity_Ref*

#### Simple State:
- **Broken**: Fixed()
- **Failed**: entry()
- **Unlocked**: Coin()
- **Locked**: Amount : unsigned int
  - Accumulate()
  - exit()
  - entry()
  - Pass()
  - Coin()

#### Next States:
- **Broken**
- **Fixed**
- **Failed**
- **Locked**
- **Unlocked**

#### Root Super State:
- **Current State**: AState*
- **set super state**: set_super_state (NewState : AState*)

### Code Snippet:
```c
// Coin() //
CurrentState->Coin()
// Failed://
set_entity_state(Broken)
```
A pattern Language for StateCharts
StateChart Patterns Roadmap

- State
  - State Transition Mechanism
  - Finite State Machines
    - Basic Statechart Specification
      - Basic Statechart
        - Using Hierarchical States
          - Hierarchical Statechart
            - Exercise independent behavior at the same time
            - Superstates with history property
              - Orthogonal Behavior
                - Broadcast events to orthogonal states
              - History State
              - Broadcasting
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

- **Actions**
  - Entity_State
    - Entity_State: AState*
  - Event_Dispatcher (Event)
    - UpdateState(New_State: AState*)

- **Events**
  - Entity_Ref
    - CurrentState
      - set_super_state(NewState: AState*)
  - IntermediateSuperState
    - MySuperState: AState*
      - set_super_state(NewState: AState*)
  - LeafState
    - MySuperState: AState*
      - set_super_state(NewState: AState*)
      - Add(state: AState*)
  - RootSuperStat
    - CurrentState: AState*
      - set_super_state(NewState: AState*)
  - VirtualState
    - IndependentStates: AState**
      - Add(State: AState*)

- **Conditions**
  - set_entity_state(New_State: AState*)
  - set_super_state(New_State: AState*)
  - entry()
  - exit()

- **Variables**
  - Num_States: int
  - NextStates: AState**
  - Entity_Ref: Entity_Interface *
  - AState:
    - Entity_Ref: Entity_Interface *
    - Num_States: int
    - NextStates: AState**
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 be used to develop the Design of controller subsystems. From statechart analysis models:

The ATM Controller Statechart

The Cruise Controller Statechart
Conclusions
This provides a way to go from Analysis directly to Design Refinement

Requirements Elicitation Process

The Analysis Process
- Static Analysis
- Dynamic Analysis

The Design Process:
- Initial Design
- Dynamic Design
- Design Refinement

Functional/Nonfunctional Requirements

Use Case Diagrams/Sequence Diagrams (the system level)
- Analysis Class Diagrams
- State Diagrams/Refined Sequence Diagrams (The object level)

Static Architectural Design

Design Refinement
- Design Class Diagrams
- Design Sequence Diagrams
- Refined Design Class Diagrams