State Pattern

From Gamma et al.

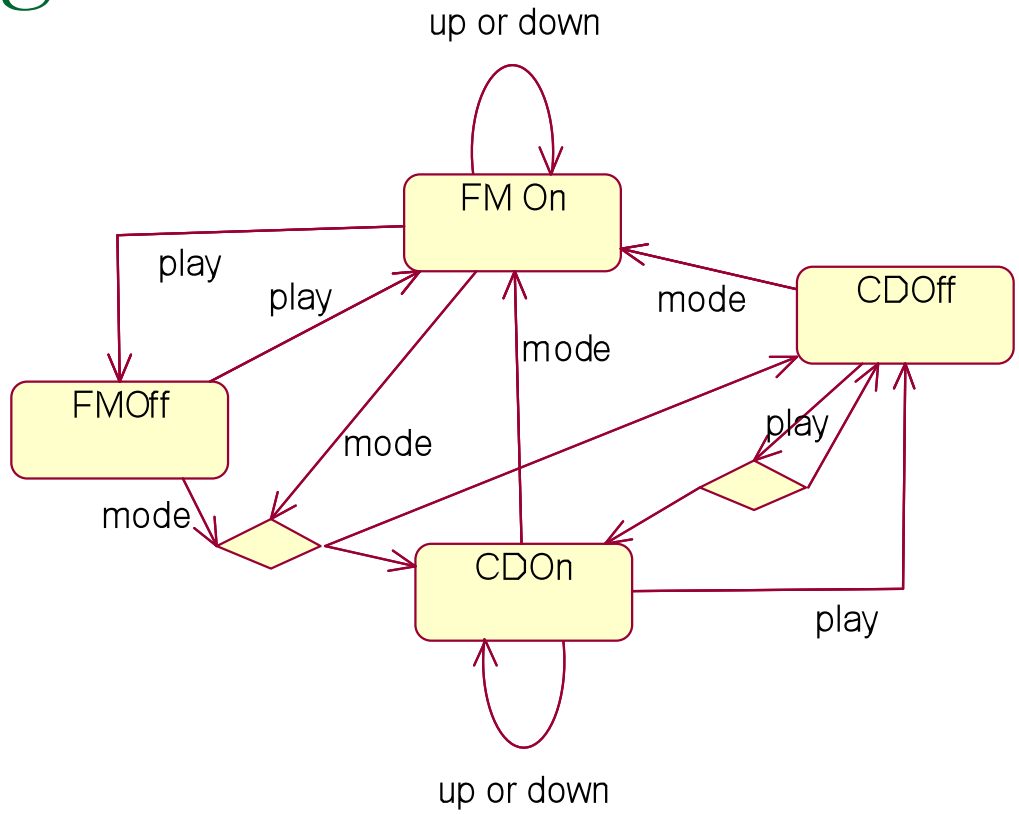
Finite State Machines

- A finite state machine (or finite state automaton) is an object that behaves in a finite set of distinct ways based on its past input and on its own choices.
- We can often model the behaviour of things using finite state machines.

A Car Stereo

- You need to design the software for a car stereo.
- The stereo has 4 push buttons.
  - Mode. Changes mode between “FM radio mode” and CD player mode.
  - Play. Turns the radio or CD player on or off.
  - Up. Moves to next track or scans to higher radio station.
  - Down. Moves to previous track or scans to lower radio station.

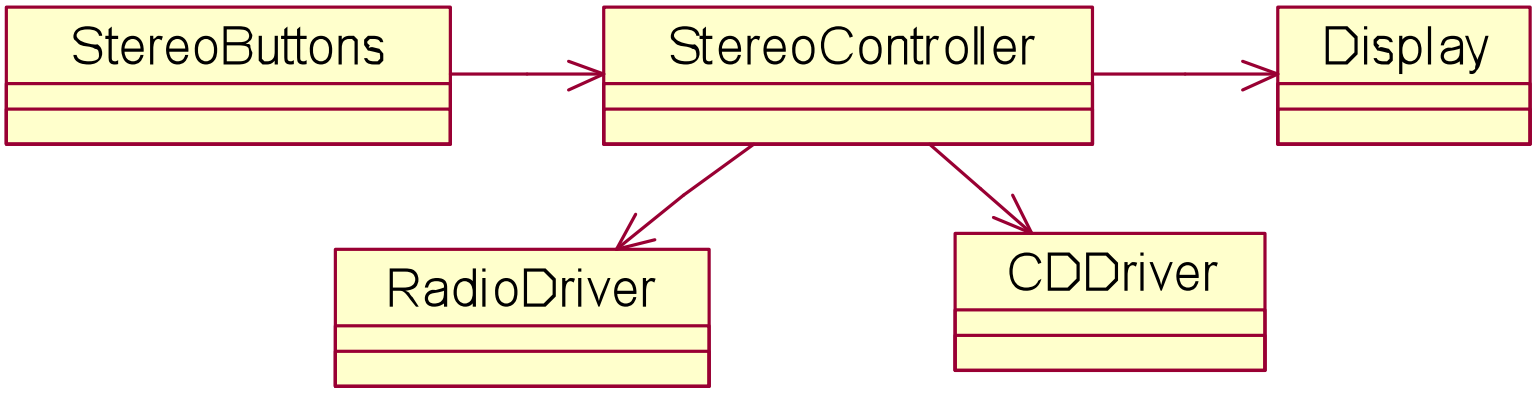
In a diagram we have



- The “CD On” state is only entered if there is a CD in the player

The Controller Class

- We need to implement a controller class that will interpret button clicks coming from the console and turn them into commands to the StereoDriver



First Cut

- Our First approach is actually a big improvement over any ad hoc approach.
- We represent each state with a unique integer. (Could also use an Enum type.)

First Cut

```
public class StereoController {
    private static final int FMOFF = 0, FMON = 1, CDON
        = 2, CDOFF = 3 ;
    private int currentState = FMOFF ;
    …
```

First Cut

```
public void play() {
    switch( currentState ) {
    case FMOFF: {
        radio.turnOn() ;
        currentState = FMON ;
        break ; }
    case FMON : {
        radio.turnOff() ;
        currentState = FMOFF ;
        break ; }
    case CDOFF: {
        if( cd.isCDInserted() ) {
            cd.turnOn() ;
            current = CDON ; }
        break ; }
    case CDON : {
        cd.turnOff() ;
        currentState = CDOFF ;
        break ; } }
    default: assert false ;
}
```

And so on for all other input events.

Reflection

- At this point we have an approach that is at least organized as opposed to ad hoc
- Unlike our state diagram, it is organized by events.
  - Information about events is concentrated.
  - Information about states is dispersed among many methods (but still encapsulated in the class)

Resilience to Change

- A state machine has several axes of change
  - New states may be added or removed
  - New events may be added or removed
  - New transitions may be added or removed
- Time for next year's model.
  - There is a new mode. "External input". So that people can plug in their MP3 player.
  - The current design can be adapted, but we must make changes in many places.

Is there another organization?

- To better deal with changes to the set of states, we will reorganize the class to concentrate information on each state in one place

The reorganized class

- Delegates events to a currentState object.

```
public class StereoController {
    static final State fmOffState = new FMOffState() ;
    static final State fmOnState = new FMOnState() ;
    static final State cdOffState = new CDOffState() ;
    static final State cdOnState = new CDOnState() ;
    private State currentState = fmOffState ;

    public void play() { currentState.play( this ) ; }
    public void mode() { currentState.mode( this ) ; }
    public void up() { currentState.up( this ) ; }
    public void down() { currentState.down( this ) ; }
    …
}
```

The reorganized class

```
abstract class State {
    void play( StereoController c ) {}
    void mode ( StereoController c ) {}
    void up ( StereoController c ) {}
    void down ( StereoController c ) {}
}
```

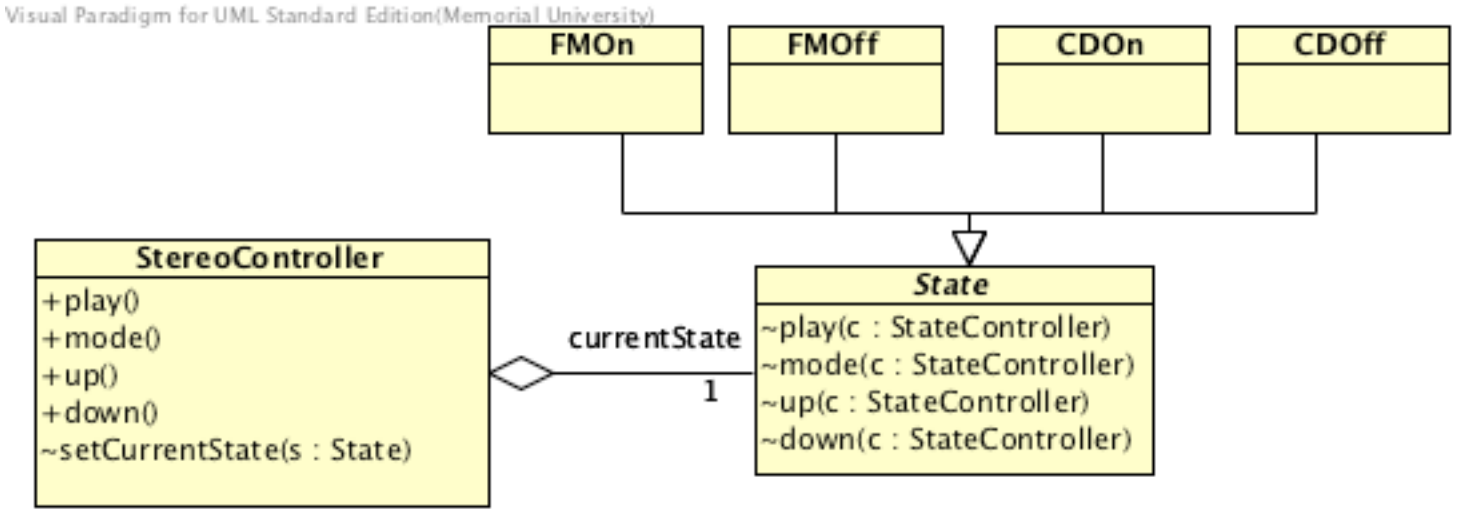
- This class provides a default behaviour for each event, which is to ignore the event.
- As an alternative, we might choose not to provide any body for these methods. This forces the programmer to provide an implementation in any concrete subclass.

The reorganized class

```
class FMOffState extends State {
    void play( StereoController c ) {
        c.turnRadioOn() ;
        c.setCurrentState( c.fmOnState ) ; }

    void mode( StereoController c ) {
        c.setDisplayModelToCD() ;
        if( c.isCDInserted() ) {
            c.turnCDOn() ;
            c.setCurrentState( c.cdOnState ) ; }
        else {
            c.setCurrentState( c.cdOffState ) ; } }
}
```

In a diagram


Visual Paradigm for UML Standard Edition(Memorial University)
FMOn
FMOff
CDOn
CDOff
StereoController
+play()
+mode()
+up()
+down()
~setCurrentState(s : State)
currentState
1
State
~play(c : StateController)
~mode(c : StateController)
~up(c : StateController)
~down(c : StateController)

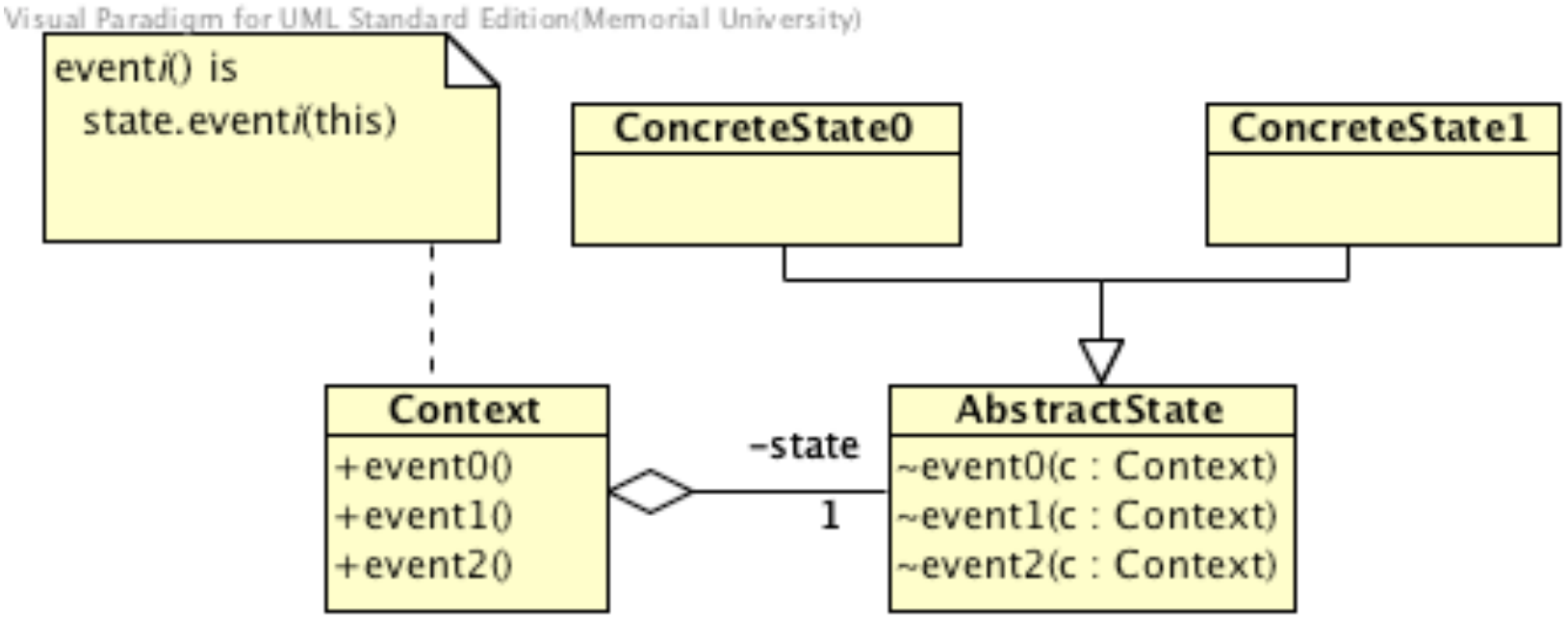
Now add the new state

- At this point we can add a new state. This minimally impacts the other states.

The State Pattern

- Intent: “Allow on object to alter its behaviour when its internal state changes. The object will appear to change its class.” [Gamma 94]
- Applicability: Use when
  - “An objects behaviour depends on its state and it must change its behaviour depending on state”

Structure


Visual Paradigm for UML Standard Edition(Memorial University)
eventi() is
state.eventi(this)
ConcreteState0
ConcreteState1
Context
+event0()
+event1()
+event2()
-state
1
AbstractState
~event0(c : Context)
~event1(c : Context)
~event2(c : Context)

Consequences

- + State specific behaviour is localized
  - Makes it easy to add and remove states
  - Allows states to be arranged in an inheritance hierarchy to share common behaviour
- + Avoids conditional branching
  - Thus simplifying the logic
- + Makes state model explicit
  - If state information is spread over multiple variables, the state model is obscured. Consider

    ```
    deviceEnum currentDevice ; // FM or CD
    boolean on ; // Is the current device “on”
    ```

  - The meaning of “on” depends on the value of “currentDevice ”
- - Responsibility is spread over more classes.
  - The context will typically have to expose its internal design to the state classes.
  - For simple problems, the State pattern may be over-design.