Using Java Enums to Implement State Machines

Jauhar Ali

https://scialert.net/fulltext/?doi=jse.2010.215.230

Air-Condition (AC) Controller

- The controller will be in one of three states: Off (default), FanOnly and AC.
- If Off is the current state and the PowerBut event occurs, the state will change to FanOnly.
- Similarly, if FanOnly or AC is the current state and the PowerBut event occurs, the stopFan action will execute and the state will change to Off.
- The ACBut event will change the state from FanOnly to AC and vice versa.
- Whenever the AC state is entered, the startCondenser action will execute.
- Similarly, whenever the AC state is exited, the stopCondenser action will execute.

Air-Condition (AC) Controller

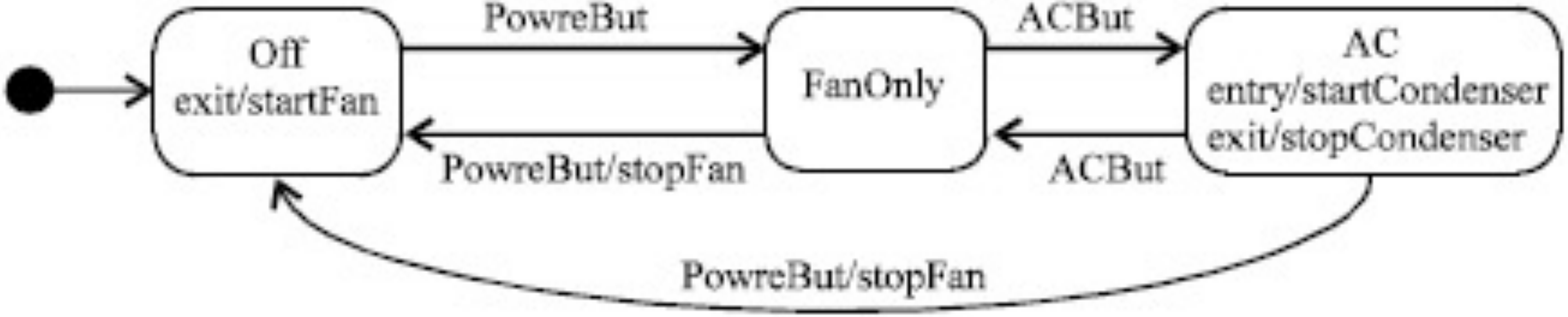
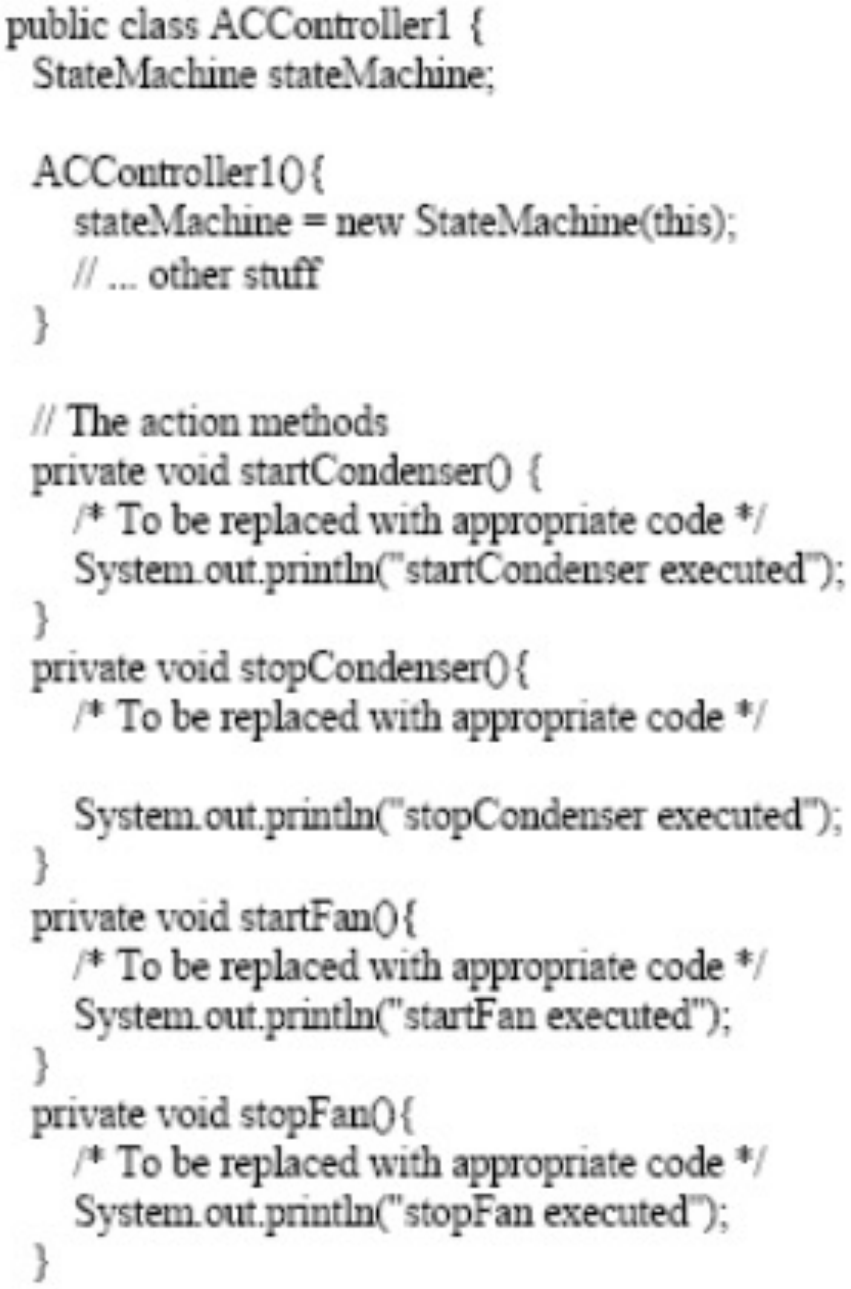


Fig. 1: Simple state machine diagram for air-condition controller

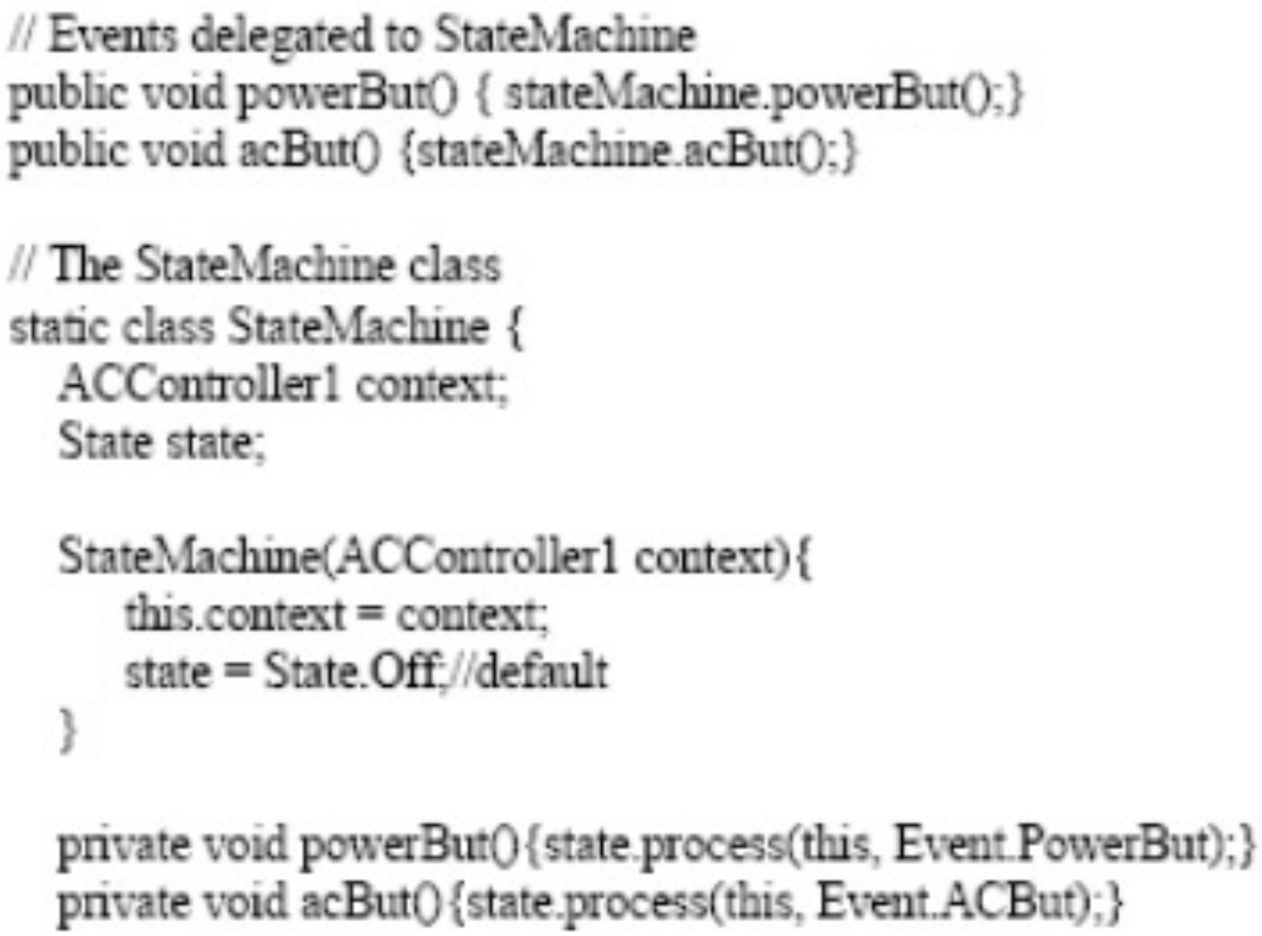
To implement the AC Controller's state machine

use a nested class, called StateMachine, inside the ACController1 class.

- The StateMachine class encapsulates almost all aspects of the state machine.
- All actions in the state machine become methods in the ACController1 class (lines 10- 25).
- The ACController1 and the StateMachine classes have references to each other (lines 2 and 33), through which they can call each other's methods.
- All events received by the ACController1 are delegated to the StateMachine (lines 28- 29).



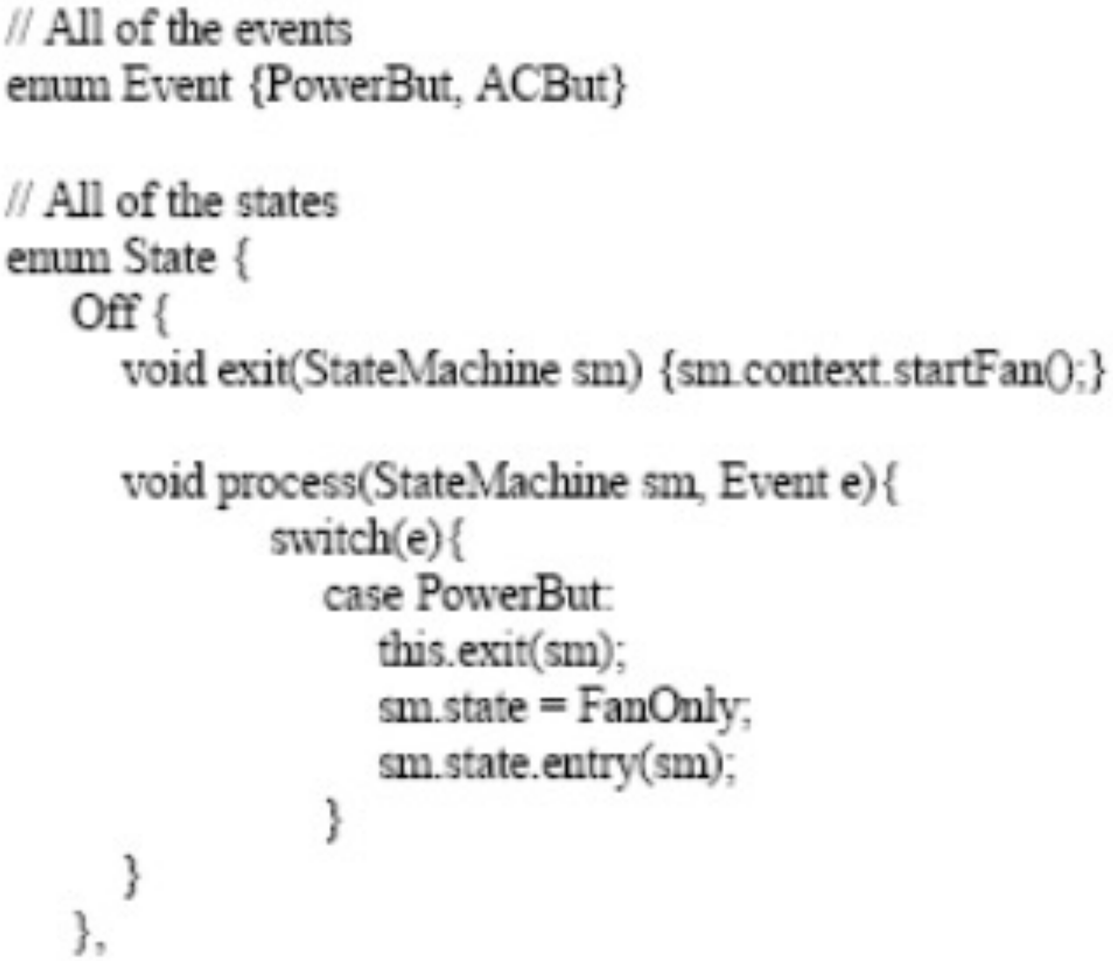
```
public class ACController1 {
  StateMachine stateMachine;

  ACController1(){
     stateMachine = new StateMachine(this);
     // ... other stuff
  }

  // The action methods
  private void startCondenser() {
     /* To be replaced with appropriate code */
     System.out.println("startCondenser executed");
  }
  private void stopCondenser(){
     /* To be replaced with appropriate code */

     System.out.println("stopCondenser executed");
  }
  private void startFan(){
     /* To be replaced with appropriate code */
     System.out.println("startFan executed");
  }
  private void stopFan(){
     /* To be replaced with appropriate code */
     System.out.println("stopFan executed");
  }
```



```
// Events delegated to StateMachine
public void powerBut() { stateMachine.powerBut();}
public void acBut() {stateMachine.acBut();}

// The StateMachine class
static class StateMachine {
   ACController1 context;
   State state;

   StateMachine(ACController1 context){
       this.context = context;
       state = State.Off;//default
   }

   private void powerBut(){state.process(this, Event.PowerBut);}
   private void acBut(){state.process(this, Event.ACBut);}
```

To implement the AC Controller's state machine

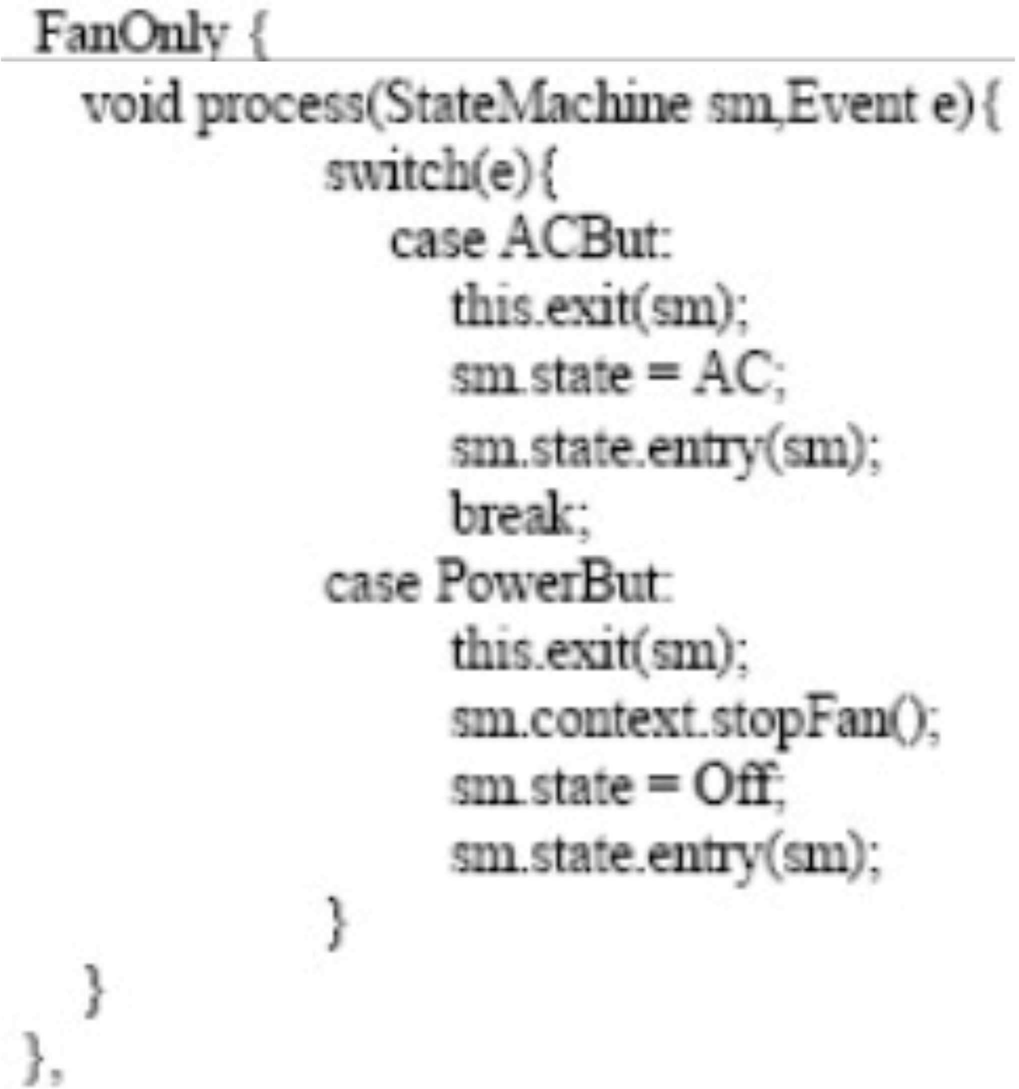
- Inside the StateMachine class, we use two Java enums:
- Event (line 45) and State (line 48).
- The Event enum represents all events and the State enum represents all states in the state machine.
- Each event and state becomes an enum value. For example, off (line 49) and FanOnly (line 62) become enum values inside State.
- The state (line 34) reference inside StateMachine represents the current state of the state machine.



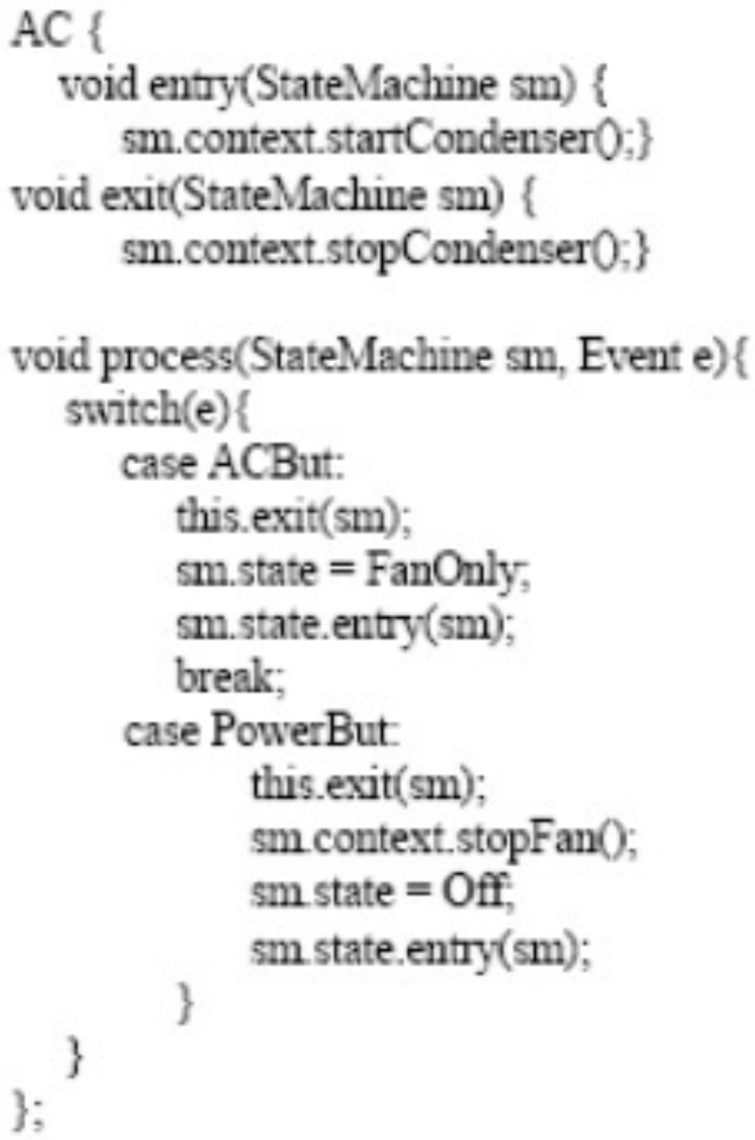
```
// All of the events
enum Event {PowerBut, ACBut}

// All of the states
enum State {
    Off {
       void exit(StateMachine sm) {sm.context.startFan();}

       void process(StateMachine sm, Event e){
                switch(e){
                   case PowerBut:
                      this.exit(sm);
                      sm.state = FanOnly;
                      sm.state.entry(sm);
                   }
       }
    },
```

To implement the AC Controller's state machine

- Java allows having methods and data members inside enums (Sun Microsystems, 2010). Each enum value can override the methods.
- The State enum has empty entry (line 102) and exit methods (line 103) .
- The AC state (line 79) overrides these methods because it has entry and exit actions in the state machine.
- The State enum has also an abstract method, named process (line 101), which is overridden by all states.
- It is called by the StateMachine on the current state whenever an event is delegated to the StateMachine (lines 41 and 42).



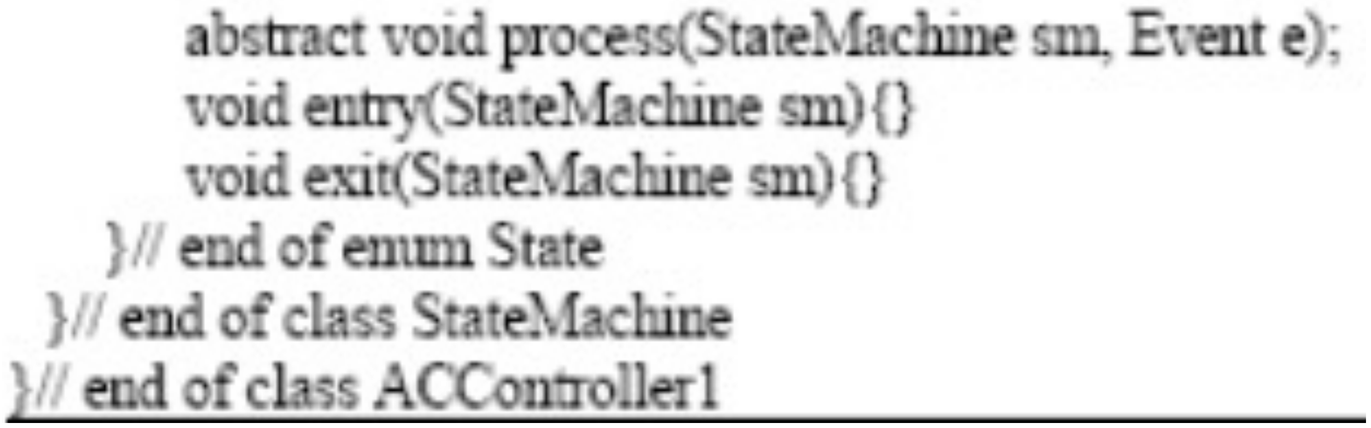
```
FanOnly {
   void process(StateMachine sm,Event e){
              switch(e){
                  case ACBut:
                      this.exit(sm);
                      sm.state = AC;
                      sm.state.entry(sm);
                      break;
              case PowerBut:
                      this.exit(sm);
                      sm.context.stopFan();
                      sm.state = Off;
                      sm.state.entry(sm);
              }
   }
},
```



```
AC {
   void entry(StateMachine sm) {
       sm.context.startCondenser();}
void exit(StateMachine sm) {
       sm.context.stopCondenser();}

void process(StateMachine sm, Event e){
    switch(e){
       case ACBut:
           this.exit(sm);
           sm.state = FanOnly;
           sm.state.entry(sm);
           break;
       case PowerBut:
                this.exit(sm);
                sm.context.stopFan();
                sm.state = Off;
                sm.state.entry(sm);
           }
    }
};
```

To implement the AC Controller's state machine

- All transitions from a state are implemented in the process method for that state.
- The process method takes an event as parameter and chooses one case from the switch statement depending on the event.
- Each case corresponds to one transition. For example, the first case in the process method of the FanOnly state implements the transition on the ACBut event (line 65).
- Inside each case (which corresponds to a transition), three methods are called in the given order: (1) the exit method of the current state, (2) the action method (if any) for the transaction and (3) the entry method of the new state.



```
            abstract void process(StateMachine sm, Event e);
            void entry(StateMachine sm){}
            void exit(StateMachine sm){}
        }// end of emum State
    }// end of class StateMachine
}// end of class ACController1
```