Example Elevator design

Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Logistiek

October 9, 2017

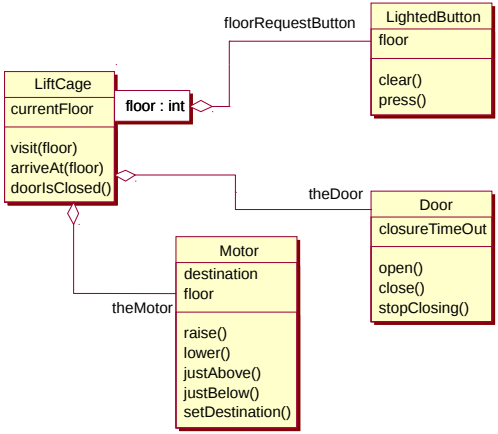
Content

State

- **Classification:** Behavioral
- **Intent:** Allow an object to alter its behavior when its internal state changes. This can be used to make an object appear as if it changed its class.
- **Motivation:**
  - In reactive systems operations (events) have a different result in different states of an object
  - In OO design often the behavior of a class is conveniently expressed in terms of state charts. One would like to implement the behavior straightforwardly from these state charts. State charts are rather difficult to make. One does not want to throw them away when implementing classes.
  - If one has no concurrency primitives offered in the programming language or OS and it is necessary to make a sequential program one can "fake" concurrency using the state pattern

Elevator example

- Class LiftCage:
  - Events:
    - visit
    - arriveAt
    - doorIsclosed



- States:
  - Cruising: the lift cage is moving towards some destination floor
  - DoorOpen: the lift cage is at a floor with the door open
  - Waiting: the lift cage has no pending requests and waits at a floor with the doors closed

State

- **Participants:**
  - *Context:*
    - defines the interface of interest to clients.
    - maintains an instance of a ConcreteState subclass that defines the current state.
  - *State:*
    - defines an interface for encapsulating the behavior associated with a particular state of the context
  - *ConcreteState:*
    - each subclass implements behavior associated with a state of the context.

State

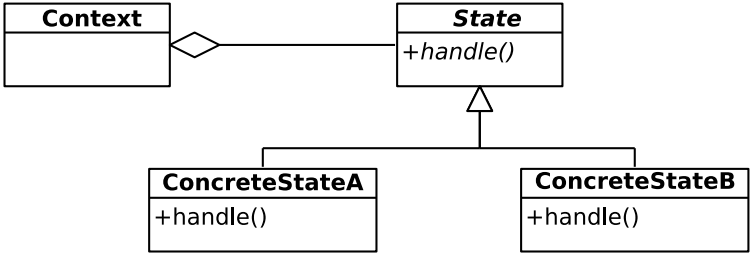
- **Applicability:**
  - An object behavior depends on its state. This is often the case in reactive systems and is expressed in state charts of objects.
  - Operations may have large multi-part conditional statements that involve the objects state. The state pattern puts each branch of the conditional in a separate subclass of the state abstraction.
  - It is expected that the behavior of a class, in particular the number of states, may change later or is likely to change by sub-classing the context.
- **Prevents coupling:**
  - No hard-coded operations in context
  - Algorithms can be changed by changing state implementation
  - Easier to modify class according to updated design (state chart).

State

- **Consequences:**
  - It localizes state-specific behavior.
  - It makes transitions explicit.
  - State objects can be shared between different contexts. (If the state itself is state-less).
  - It may help to implement behavior expressed in complex state charts.
  - It requires either that the context's state variables are part of the concrete state objects, or that the concrete states have access to the state variables of the context. To avoid breaking encapsulation such state variables must be shared using C++ "friend"-like mechanisms. In Java this can be achieved in package private classes.

The GoF State Pattern

The state pattern uses an interface that defines the "events" that are to be accepted and uses a different class object to define the reaction in any of the

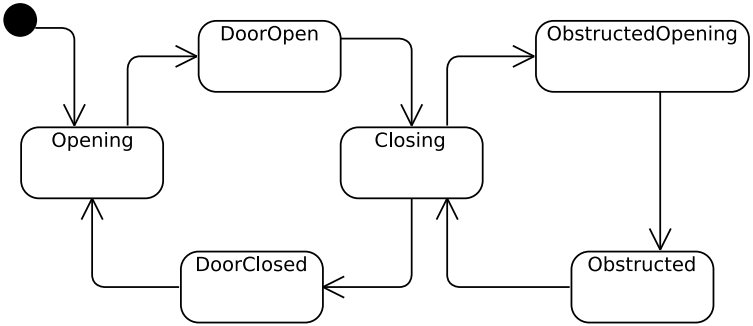


defined states.

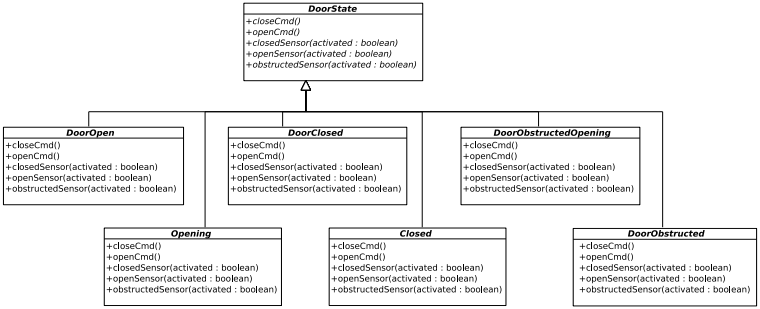
Advantages:

- The states can be defined in separate files and separately tested.
- On level of checking is dropped.
- Default reactions (e.q. logging of events) can be reused.

State pattern example, state diagram



State pattern example, class diagram for state pattern



Implementation remarks

Note that the state class and its subclasses do/should not have any attributes. This is typical for the state pattern. A state IS a state and does NOT have state (i.e. attributes).

The operations the state can execute are defined in the context (the door in the example). A typical Java implementation (c|w)ould use inner classes for the states (with the potential problem of one big state machine file).

The C++ solution would be to pass the context as a parameter to the event methods, thereby preventing to have to pass the context as a attribute for the constructor. This can also be used in Java of course, although it would force you to increase the visibility of the implementation, thereby making the implementation more brittle.

State behavior

We assume that the desired state behavior is documented in the UML state diagram notation.

- Dependent on the tool that you use, it is possible to derive the necessary transition information from such a diagram.
- This information can be used to create an implementation.
- There are numerous ways to derive the implementation from the notation.

State behavior notation aspect

All the information that is possible and potentially available in the the UML state diagram can be notated in textual or even tabular form.

The modern UML tools support a XML based format called XMI (for XML Meta-data Interchange) that may contain state model information.

In all cases, a correct drawing of your state model can be used to derive a notation that can be used for automatic code generation.

A textual notation

Notation: (by the way: this is not XML)

≪state≫;≪event≫[≪guard≫]/{≪action list≫};≪newstate≫

where the guard is optional.

For the state we have the notation:

@≪name≫;ENTRY ≪entry actions≫;EXIT ≪exit actions≫;≪initial state≫;≪history state≫

For nested state we use the notation ≪parent≫.≪child≫

Examples:

```
S1;e1[g1]/a1();S3.S31

@VISITFLOOR;ENTRY -; EXIT -; VISITFLOOR.OPENING
```

Nested switch statement

Traditionally the state of can be implemented using a doubly nested switch statement.

Advantages

- Simple to understand.
- Useable in both OO and non OO languages

Disadvantages

- Only maintainable in one ever growing file.

Variations:

- Top level is state, nested are the events.
- Top level are events, nested are the states.

State − > event example in C++

```
void StateMachine::handleButton(Button b){
  switch (state) {
  case S1:
    switch (b.type){
      //...
    case UP:
      //...
      break
    }
  case S2:
    switch (b.type){
      // ...
    case DOWN:
      //...
      break
    }
    break;
  }
}
```

Event− > state example in C++

```
void StateMachine::handleButton(Button b){
  switch (b.type){
  case UP:
    switch (state) {
      //...
    case S1:
      //...
      break
    }
  case DOWN:
    switch (state){
      // ...
    case S2:
      //...
      break
    }
    break;
  }
}
```

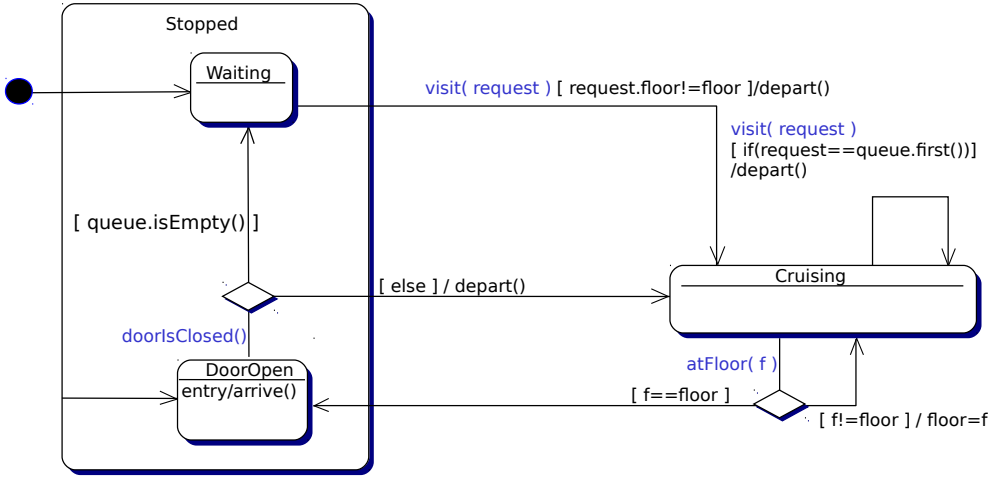
Discussion of alternatives

The event $->$ state is advantageous in the case that different event are encoded as different method calls. Then all reaction to this event is specified in one method body.

Example: State model of the Elevator Cage

Note: this state model is simplified.

- State chart
  - In each of the states the response to events is different



Implementing Complex State Behavior

- Attempt 0: Keeping implicit state variables. Implicite means here: derive from inputs and state of outputs, like *is motor moving*
  - Disadvantage: relation to state diagrams obscure, hard to maintain.
- Attempt 1: Introducing explicit state variables
  - Disadvantages: complex conditionals, still not easy to maintain

Explicit state variable code

Attempt 1 implementation:

```
class LiftCage {
  protected int curentFloor = 0;
  protected final int DoorOpen = 0;
  protected final int Waiting  = 1;
  protected final int Cruising = 2;
  // current state
  protected int state = Waiting;
  public LiftCage() {
    // ...
  }
  public void visit(int floor)  {
    if (state==Cruising) {...}
    else if (state==Waiting) {...}
    else if (state==DoorOpen) {...}
  }
  public void doorIsClosed() {
    if ( state==DoorOpen )  {
  // ...
    }
  }
  public void atFloor(int floor){
    if( state==Cruising
        && floor==currentFloor){
  //...
    }
  }
}
```

- Attempt 1: Introducing explicit state variables
  - Disadvantages: complex conditionals, still not easy to maintain

Implementing Complex State Behavior

- Attempt 2: Introducing explicit state objects (cont'd)

```
class LiftCage {
    // to be accessed by states:
    public final State doorOpen
  = new DoorOpen(this);
    public final State waiting
  = new Waiting(this);
    public final State cruising
  = new Cruising(this);
    protected State state=waiting;
    protected int currentFloor=0;
    public int getFloor() {
  return currentFloor();
    }
    public void setFloor(int floor) {
  currentFloor = floor;
    }
    public void setState(State s) {
  state=s;
    }

    // to be called by clients:
    public void visit(int floor) {
  state.visit(floor);
    }
    public void doorIsClosed() {
  state.doorIsClosed();
    }
    public void atFloor(int floor){
  state.atFloor(floor);
    }
}
```

Implementing Complex State Behavior

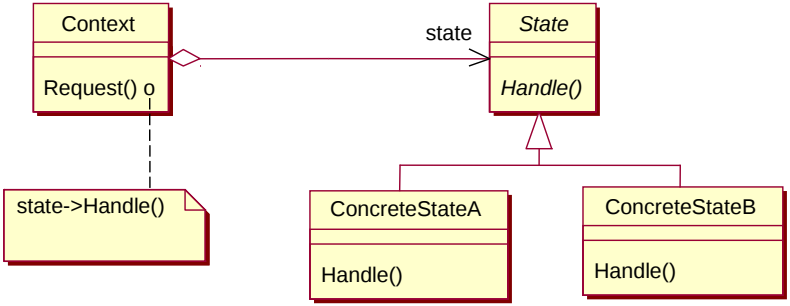
- Attempt 2: Introducing explicit state objects

```
interface State {
  void visit(int floor);
  void doorIsClosed();
  void atFloor(int floor);
}

class Cruising implements State {
  protected LiftCage context;
  public Cruising(LiftCage c) { context = c; }
  // ....
  public void atFloor(int floor) {
     if ( floor== context.getFloor()) {
        context.setState(context.doorOpen);
     }
  }
}
```

State

- State Pattern Structure



Note: Often, the state needs some resources of the context, saying that most implementations will pass the context as in `handle(Context ctx)` method.

State-**less** State

- In attempt 2, each state constructor is bound (coupling, dependency) to one context.
- A better approach would be to pass the context reference with each method call. This allows state-**less** states, making it possible to share state objects between machines. These state objects could then be created once and reused, making the object resource usage minimal.
- **Consequences:**
  - Pass the context object to each state method as in `void atFloor(LiftCage cage, int floor )`.
  - Implement method `setNewState(State ns)` in `Context`
  - To top it all off, in `setNewState(State ns)` call `state.exitState()` and `state.enterState()`, to get exit and entry behaviour almost for free.

State change in context

```
public State changeState(State newState){
    State previous = currentState;
    previous.exit();
    currentState= newState;
    currentState.enter();
    return previous;
}
```