elements of software construction

implementing state machines

Daniel Jackson
September 24, 2009
what are design patterns?
design patterns

design?
\· code design, not behavioral design
\· some language-dependent

where they came from
\· pattern idea due to Christopher Alexander
\· popularized in software engineering by “Gang of Four” book

controversies
\· sometimes complicated, work around language defects

our patterns
\· we’ll use patterns throughout the course
\· some Gang of Four, some standard but unnamed patterns
pattern elements

how we’ll explain patterns in this course

- **name**: essential in design discourse
- **motivation & tradeoffs**: why the pattern was invented, +/-’s
- **prototype**: what the structure looks like on a simplified example
- **example**: applied to a non-trivial example
levels of understanding

three levels of understanding patterns

mechanics
• how the pattern works at runtime
• basic Java knowledge: you should grasp this very quickly

motivation
• why the pattern was invented, when to use it
• you should understand this with a little practice

subtleties
• implications of using the pattern
• this level of understanding will come with time
state machine patterns

machine as class
  • state machine is implemented as a class
  • related to Gang of Four’s Singleton pattern

machine as object
  • class represents set of state machines
  • the standard use of objects

state as object
  • called State by Gang of Four

state as enumeration
  • factoring out control state
  • can be used in machine as class, machine as object
starting point: state machine
state machine to implement

```plaintext
state
List<Event> lastRecording, recording = <>;
boolean REC = false;

op pr_R
  do if (REC) lastRecording = recording else recording = <>; REC = ! REC

op pr (k)
  do if (REC) recording = recording ^ <pr_k>

op rel (k)
  do if (REC) recording = recording ^ <rel_k>

op pr_P
  do // enq events in last
```
machine as class pattern
motivation & tradeoffs

**simple imperative idiom**

- state stored in static fields
- operations implemented as static methods
- each operation handles the events in some event class

**advantages**

- no allocation, so good for real-time applications
- operations can be invoked using globals from anywhere

**disadvantages**

- can’t create multiple instances of a machine
- modularity in Java is based on objects, not classes
- can’t pass around and access anonymously or through interface
prototype

machine

class Machine {
    static StateComponent1 comp1 = ... ;
    static StateComponent2 comp2 = ... ;

    static void op1 () {
        comp1 = f11(comp1, comp2);
        comp2 = f21(comp1, comp2);
    }

    static void op2 () {
        comp1 = f12(comp1, comp2);
        comp2 = f22(comp1, comp2);
    }

    ...

}

dispatcher

    Machine.op1 ();
public class PianoMachine {
    private static boolean isRecording = false;
    private static List<NoteEvent> recording;
    private static List<NoteEvent> lastRecording = new ArrayList<NoteEvent>();
    private static Set<Pitch> pitchesPlaying = new HashSet<Pitch>();

    public static void toggleRecording() {
        if (isRecording)
            lastRecording = recording;
        else {
            recording = new ArrayList<NoteEvent>();
        }
        isRecording = !isRecording;
    }

    public static void beginNote(NoteEvent event) {
        Pitch pitch = event.getPitch();
        if (pitchesPlaying.contains(pitch))
            return;
        pitchesPlaying.add(pitch);
        midi.beginNote(event.getPitch().toMidiFrequency());
        if (isRecording) recording.add(event);
    }
    ...
}

why is the operation called toggleRecording and not pressR?

how is the state of the key submachine represented?
public class PianoApplet extends Applet {

    public void init() {
        addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                char key = (char) e.getKeyCode();
                switch (key) {
                    case 'P': PianoMachine.startPlayback(); return;
                    case 'R': PianoMachine.toggleRecording(); return;
                }
                NoteEvent ne = new BeginNote(keyToPitch(key));
                PianoMachine.beginNote(ne);
            }
        });
    }

    ...}

machine as object pattern
motivation & tradeoffs

standard object-oriented idiom

• state stored in fields or instance variables of object
• operations implemented as methods

advantages

• allow multiple machines of same type
• can pass machine object around and invoke operation on it anywhere
• can exploit modularity and decoupling

disadvantages

• indirection and aliasing can make code harder to understand
prototype

machine

class Machine {
    StateComponent1 comp1 = ... ;
    StateComponent2 comp2 = ... ;

    void op1 () {
        comp1 = f11(comp1, comp2);
        comp2 = f21(comp1, comp2);
    }
    void op2 () {
        comp1 = f12(comp1, comp2);
        comp2 = f22(comp1, comp2);
    }
    ...
}

dispatcher

Machine m = new Machine ();  // make machine
m.op1 ();                   // use machine
public class PianoMachine {
    private boolean isRecording = false;
    private List<NoteEvent> recording, lastRecording;
    private Set<Pitch> pitchesPlaying;

    public PianoMachine() {
        lastRecording = new ArrayList<NoteEvent>();
        pitchesPlaying = new HashSet<Pitch>();
    }

    public void toggleRecording() {
        if (isRecording)
            lastRecording = recording;
        else {
            recording = new ArrayList<NoteEvent>();
        }
        isRecording = !isRecording;
    }

    public void beginNote(NoteEvent event) {
        Pitch pitch = event.getPitch();
        if (pitchesPlaying.contains(pitch)) return;
        pitchesPlaying.add(pitch);
        midi.beginNote(pitch.toMidiFrequency());
        if (isRecording) recording.add(event);
    }
...

what's the difference between the meaning of the expression recording in this pattern and the last one?

think of recording as being its own state machine, nested in the larger one. what pattern is used?
example: dispatcher

```java
public class PianoApplet extends Applet {

    public void init() {
        final PianoMachine machine = new PianoMachine();
        addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                char key = (char) e.getKeyCode();
                switch (key) {
                case 'P': machine.startPlayback(); return;
                case 'R': machine.toggleRecording(); return;
                }
                NoteEvent ne = new BeginNote (keyToPitch (key));
                machine.execute (ne);
            }
        });
    }

    ...
```
state as enumeration pattern
motivation & tradeoffs

syntactic clarity

• reflect state transitions clearly in syntax
• uniform decisions without nested ifs

advantages

• direct correspondence with diagram
• easy to read, write, generate automatically

disadvantages

• Java inherited C-style “fall through”; if exploited, code gets hard to read
• assumes one ‘mode’ or ‘superstate’ variable
prototype

in context of machine as class

class Machine {
    enum State { S1, S2, S3 }
    static State state;

    static void op1 () {
        switch (state) {
            case S1: if ... state = State.S2 else state = State.S3 ...; break;
            case S2: ... state = State.S3; break;
            case S3: ... state = State.S1; break;
        }
    }
}
public class PianoMachine {
    private List<NoteEvent> recording, lastRecording;

    private enum State {
        RECORDING, PLAYING
    }
    private State state;

    public void toggleRecording() {
        switch (state) {
            case PLAYING:
                state = State.RECORDING;
                recording = new ArrayList<NoteEvent>();
                return;
            case RECORDING:
                lastRecording = recording;
                state = State.PLAYING;
        }
    }

    ...
}
state as object pattern
motivation

major and minor modes

- want to separate state machine behavior
- transitions between major modes (often simple and discrete)
- transitions within modes (often over complex data)

idea

- one class for each state (major mode)
- class contains additional state components for this mode
- state transition returns new state object, maybe from another class
interface declaring states

```java
interface State { State op1 (); State op2 (); ... }
```

class implementing a single (super) state

```java
class S1 implements State {
    int c = 0;

    State op1 () { c++; return this; }
    State op2 () {
        if (c > 10) return new S2 ();
        else return this;
    }
    ...
}
```

dispatcher

```java
State state = new S1 ();
...
state = state.op1 ();
```
example: dispatcher

```java
public class PianoMachine {

    private PianoState state;

    public PianoMachine() {
        state = new PlayingState(new ArrayList<NoteEvent>());
    }

    public void toggleRecording() {
        state = state.toggleRecording();
    }

    public void startPlayback() {
        state = state.startPlayback();
    }

    public void beginNote(NoteEvent event) {
        state = state.beginNote(event);
    }

    public void endNote(NoteEvent event) {
        state = state.endNote(event);
    }
}
```
example: machine

```java
public interface PianoState {
    public PianoState toggleRecording();
    ...
}

public class RecordingState implements PianoState {
    private final List<NoteEvent> recording, lastRecording;

    public RecordingState (List<NoteEvent> recording) {
        this.lastRecording = recording;
        this.recording = new ArrayList<NoteEvent>();
    }

    public PianoState toggleRecording() {
        return new PlayingState (recording);
    }

    public PianoState beginNote(NoteEvent event) {
        midi.beginNote(event.getPitch().toMidiFrequency());
        recording.add(event);
        return this;
    }
    ...
}
```

why is `recording` variable (and not `lastRecording`) passed to constructor of new state?

handling of key repeats is missing. how would it be added?
tradeoffs

advantages

‣ clean scoping of nested components
‣ in each mode, only the relevant subcomponents appear

disadvantages

‣ can’t handle orthogonal components
‣ need wrapper class to avoid exposing state-replacing mechanism
‣ like Switch on Enum, need to characterize states with one mode ‘variable’
‣ state components that persist across modes must be passed around
‣ allocate on every transition, or need to create singleton states
summary
which pattern to use?

- **machine as class** doesn’t give the modularity we need
- **state as object** is clumsy here: too many components to pass around
- **state as enumeration** doesn’t help: not enough discrete states
- so choose **machine as object**

how to handle playback

- implement queue idea explained last time

modularity issues

- look at dependence diagram
- introduce interfaces for decoupling
summary
what did we do?

four patterns, each with +'s and -'s

- most useful: machine as object
- good to know: state as object, state as enumeration
- usually avoid: machine as class
understanding midi piano code

• [easy] The variable `lastRecording` is initialized in `PianoMachine`’s constructor. Is this necessary? What alternatives are there?

• [easy] What pattern is used to change color in the applet? Why isn't this done in the machine class?

• [hard] `PianoMachine` does not modify `lastRecording`. Why is this significant?

• [obscure] In state pattern of piano, why not put `state = state.op()` assignments in the applet class, instead of adding a separate machine class?