implementing state machines

Daniel Jackson
February 11, 2008
what are design patterns?
design patterns

design?

• code design, not behavioural design
• some language-dependent

where they came from

• pattern idea due to Christopher Alexander
• popularized in software engineering by “gang of four” book

controversies

• some are complicated and ugly
• some are workarounds for missing language features
• but the general idea of patterns is surely good
pattern elements

how we’ll explain patterns in this course

- name: essential in design discourse
- motivation: why the pattern was invented
- prototype: what the structure looks like
- example: applied to a non-trivial example
- consequences: both good and bad
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
implementing a midi piano
state
List<Event> last, current = <>;
boolean REC = false;

op R-pr
  do if (REC) last = current else current = <>; REC = !REC

op pr (k)  // note how I made k a parameter to generalize over all keys
  do if (REC) current = current ^ <k>
state machine patterns

patterns

• machine as class (“Singleton”)
• machine as object (“Object as machine” -- my name)
• state as object (“State”)
• state as enum constant (“Switch on enumeration” -- my name)
singleton pattern
motivation

simple imperative idiom

- state stored in static fields
- operations implemented as static methods

advantages

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

“operation”: code that handles all events in some event class
machine: handle event with operation

```java
class Machine {
    static boolean component = false;

    static void op () {
        component = true;
    }
    ...
}
```

client: when event occurs

```java
Machine.op ();
```
public class PianoMachine {
    private static boolean isRecording = false;
    private static List<NoteEvent> recording;
    private static List<NoteEvent> lastRecording = new ArrayList<NoteEvent>();

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

public class PianoApplet extends Applet {

    public void init() {
        addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                char key = (char) e.getKeyCode();
                switch (key) {
                case 'R': PianoMachine.toggleRecording(); return;
                ...
                }
            }
        });
    }
}
consequences

only one instance

\cdot good for keeping track of allocation
\cdot bad if >1 state machine instance needed

namespace access

\cdot static elements are globally named (modulo access rules)
\cdot so can call operation from anywhere in code; no need to pass machine

modularity

\cdot modularity in Java is based on objects, not classes
\cdot can’t use interfaces for decoupling: singleton never anonymous to client
object as machine pattern
motivation

standard object-oriented idiom

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

advantages

• multiple machines
• decoupling with interfaces
prototype

machine

class Machine {
    boolean component = false;

    void op () {
        component = true;
    }
    ...
}

client

\• make machine

    Machine m = new Machine ();

\• use machine

    m.op ();
example (full code in handout)

```java
public class PianoMachine implements MusicMachine {
    private boolean isRecording = false;
    private List<NoteEvent> recording, lastRecording;
    public PianoMachine() {}
    public void toggleRecording() {
        if (isRecording)
            lastRecording = recording;
        else {
            recording = new ArrayList<NoteEvent>();
        }
        isRecording = !isRecording;
    }
    ...
}

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 'R': machine.toggleRecording(); return;
                    ...
                }
            }
        ...
    }
}
```

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?

why does `PianoMachine` implement `MusicMachine`?
dynamic access and modularity

- can pass machine object around and invoke operation on it anywhere
- can exploit modularity and decoupling

element: handling note events

```java
addKeyListener(new KeyAdapter() {
    public void keyPressed(KeyEvent e) {
        char key = (char) e.getKeyCode();
        NoteEvent ne = new BeginNote(keyToPitch(key));
        ne.execute(machine);
    }
});

public abstract class NoteEvent {
    abstract public void execute(MusicMachine m);
    ...
}
public class BeginNote extends NoteEvent {
    public void execute(MusicMachine m) {
        m.beginNote(this);
    }
    ...
}```
switch on enum pattern
motivation

syntactic clarity

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

advantages

• easy to read, write, generate automatically
class Machine {
    enum State { S1, S2, S3 }
    static State state;

    static void op () {
        switch (state) {
            case S1: ... state = State.S2; break;
            case S2: ... state = State.S3; break;
            case S3: ... state = State.S1; break;
        }
        ...
    }
}

often used when events read from stream

Event e = read();
switch (state) {
    case S1: if (Event1(e)) state = State.S2; else ... break;
    ...
}
public class PianoMachine implements MusicMachine {
    private List<NoteEvent> recording, lastRecording;

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

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

    ...
consequences

correspondence with model
\* very direct and in principle easy to check

switch statement liabilities
\* Java inherited C-style “fall through”
\* if you exploit this, code gets hard to read
\* best to execute only one non-empty case
\* must remember break or return statement
state pattern
motivation

major and minor modes

• want to separate state machine behaviour
• 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

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

state

    class S1 implements State {
        int c = 0;

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

client

    State state = new S1 ();
    ...
    state = state.op1 ();
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 class PianoMachine implements MusicMachine {
    private PianoState state;
    public void toggleRecording() {
        state = state.toggleRecording();
    }
    ...
}
consequences

need wrapper class
• best not to expose state-replacing mechanism to client

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

but can’t handle orthogonal components
• like Switch on Enum, need to characterize states with one variable

passing state
• state components that persist across modes must be passed around

allocation
• allocate on every transition?
• but can often create singletons for states
piano design issues
what else?

dependency design
• for clarity and maintainability
• not just package structure
• also interfaces (see next lecture)

interleaving manual play and playback
• object-passing games
• queue-based concurrency (see later in course)
dependency design

dependency diagram
• arrow means “depends on”
• or “uses”

dependences avoided
• music on midi and vv.
• anything on PianoApplet
• music on piano

why avoid each of these dependences?
public class PianoPlayer {
    private final BlockingQueue<NoteEvent> queue, delayQueue;
    private final PianoMachine machine;

    public PianoPlayer () {
        queue = new LinkedBlockingQueue<NoteEvent> ();
        delayQueue = new LinkedBlockingQueue<NoteEvent> ();
        machine = new PianoMachine(this);
        spawn processQueue (); // pseudocode
        spawn processDelayQueue (); // pseudocode
    }

    public void playbackRecording (List<NoteEvent> recording) {
        for (NoteEvent e: recording)
            delayQueue.put(e);
    }

    public void processQueue () {
        while (true) {
            NoteEvent e = queue.take();
            e.execute (machine);
        }
    }

    public void processDelayQueue () {
        while (true) {
            NoteEvent e = delayQueue.take();
            midi.Midi.wait (e.getDelay());
            queue.put(e);
        }
    }
}

public class PianoMachine implements MusicMachine {
    private final PianoPlayer player;
    public PianoMachine(PianoPlayer player) {
        this.player = player;
    }

    public void startPlayBack () {
        player.playbackRecording(lastRecording);
    }
}

how many threads are running here?
summary
what did we do?

four patterns, each with +’s and -’s

• most useful: Object as Machine
• good to know: State, Switch on Enum
• usually avoid: Singleton

issues arising

• mechanics: object creation; field update and initialization
• namespace: when one element can access another
• syntactic clarity: getting program structure to match problem structure
• interfaces: more next lecture
lecture exercise

by next lecture (Wednesday) in your LNBs

understanding midi piano code

• [easy] The variable lastRecording is initialized in PianoMachine’s constructor. This is not necessary. Remove this initialization, and modify the code elsewhere to make it work. Is this better or worse?

• [easy] What pattern is used to change colour 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?

for each pattern we covered today

• sketch how to modify the code to ignore R-pr events when any key is down
feedback

please help us improve the course

• write a few sentences in your LNB after each lecture

questions

• what was the most important idea in today's lecture?
• what did you find confusing?
• what examples or exercises helped or didn't help you?