6.005
elements of
software
construction

implementing state machines

Daniel Jackson

what are design patterns?

design patterns

Images of book covers removed due to copyright restrictions.
A Pattern Language (Christopher Alexander et al.),
and Design Patterns: Elements of Reusable
Object-Oriented Software (Erich Gamma et al.).

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

Design patterns are just a way of codifying knowledge, but they're a very effective way of getting novices up to speed with ideas that they would otherwise have to learn through long experience. The DP book is one of the most successful books in software engineering history, and is well worth owning and reading. It's in C++ but most of the lessons apply to Java. Whether they apply to more modern languages (such as Scala or Haskell) is another matter, since many of the patterns are workarounds for missing language features. Most SE courses teach design patterns as a big catalog. Instead, we're going to learn the patterns that are relevant to moving from behavioral design to code for each of the paradigms. So today is patterns for implementing state machines.

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

The name is more important than you might think. When you're discussing a design, it's useful to say "let's use the Composite pattern here" and have your team mate immediately understand you. Every pattern has not only positives but also negatives; as always, engineering is a tradeoff. One thing you should be aware of is that most patterns make the code more _complicated_, usually to achieve some kind of decoupling. This creates a risk that you'll go pattern crazy and produce code with so many patterns in it it's close to incomprehensible.

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

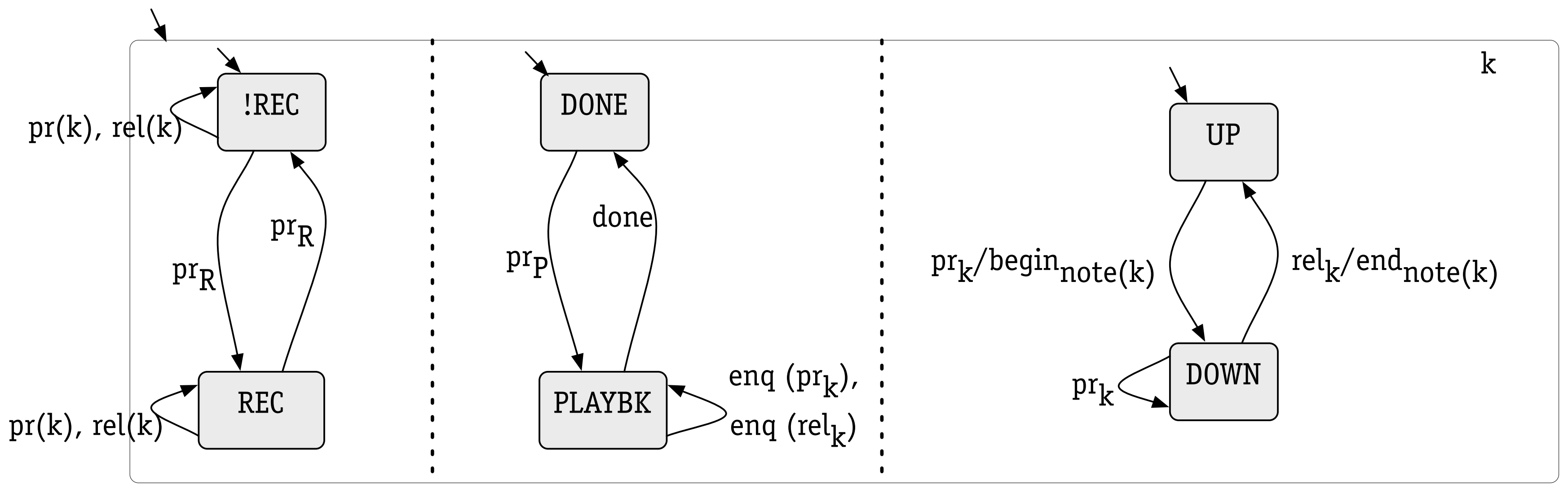
state as enumeration

- factoring out control state
- can be used in machine as class, machine as object

Singleton is a name used when a class has only one instance; we're using this idea in the context of state machines, which is narrower than the pattern's intent. Machine as object doesn't get a name in the GOF book because it's really the standard OO idiom. State as Object, or State as the GOF call it, is very nice in some specialized circumstances but not very widely applicable, and quite clumsy if it doesn't fit well. State as Enumeration is an old-fashioned pattern that's been used for years in state machine code, especially autogenerated code.

starting point: state machine

state machine to implement



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

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

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

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

op prP
  do // enq events in last
```

This is a summary of where we got to in our state machine behavior design, and it will be the starting point for our design of the mechanism. You can think of the textual part as describing another component state machine that is in parallel with all the others. The first two diagrammatic component machines just summarize this machine from the perspectives of the recording and playback keys. We'll illustrate the patterns on the recording part -- the submachine on the left and the textual submachine -- but we'll also look at playback later in the lecture.

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

By "event class", I mean class in the state machine semantics sense. You don't have to have a Java class for the events (although that's often convenient, and we will actually do that).

Note the namespace advantage. With objects, you need to be holding an object to call one of its methods. But to call a static method m of a class C, you just write C.m(), and so long as you have access to C and m (that is, they're not hidden by access modifiers), you can call the method.

The modularity issue is rather subtle and won't make sense until you've seen later at the end of the lecture what we can do with objects (and can't do with classes). This is just how Java works; in other programming languages, you can pass modules around the way Java lets you pass around objects.

prototype

machine

```
class Machine {
    static StateComponent comp1 = ... ;
    static StateComponent 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 ();
```

There are some number of state components, comp1, comp2, etc. In each operation, some of these are updated by computing new values based on old values of the components. Note that the dispatcher just needs to access the name of the class -- there are no objects here.

example: machine

```
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);
   }
...}
```

Recall that in our state machine formalism, an event might not be able to happen in some state. But in all the implementation patterns that we'll be looking at, input events can always happen -- although in some states they might have no effect.

Note how the set of key state machines, each with a boolean state component, is implemented here as a single state machine with a state component that is a set. This is a standard transformation.

Rather than calling the operation pressR, I've called it toggleRecording. I did this to decouple the meaning of the events from the particular bindings we chose for them. If we changed to using a different key, we wouldn't want to change all the names in the program.

example: machine

```
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****?*

Recall that in our state machine formalism, an event might not be able to happen in some state. But in all the implementation patterns that we'll be looking at, input events can always happen -- although in some states they might have no effect.

Note how the set of key state machines, each with a boolean state component, is implemented here as a single state machine with a state component that is a set. This is a standard transformation.

Rather than calling the operation pressR, I've called it toggleRecording. I did this to decouple the meaning of the events from the particular bindings we chose for them. If we changed to using a different key, we wouldn't want to change all the names in the program.

example: machine

```
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?*

Recall that in our state machine formalism, an event might not be able to happen in some state. But in all the implementation patterns that we'll be looking at, input events can always happen -- although in some states they might have no effect.

Note how the set of key state machines, each with a boolean state component, is implemented here as a single state machine with a state component that is a set. This is a standard transformation.

Rather than calling the operation pressR, I've called it toggleRecording. I did this to decouple the meaning of the events from the particular bindings we chose for them. If we changed to using a different key, we wouldn't want to change all the names in the program.

example: dispatcher

```
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);
             }
...}
```

This is a standard pattern called Listener. We create an object called a listener that is attached (by the addKeyListener) call to the applet; the applet then calls the method keyPressed on this listener when a key is pressed. The syntax of the expression that is passed to addKeyListener is a bit complicated but all it's doing is creating an object of an anonymous subclass of KeyAdapter. You can think of it as saying: make an object of type KeyAdapter but with this method for KeyPressed.

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

Indirection means that you can have a sequence of calls that are passed through various intermediate objects. This can be used to good effect, but it can make the code hard to follow as you have to trace through lots of calls. Aliasing means that two different names can refer to the same object, so that calls to one are equivalent to calls to the other: x.m() and y.m() might actually have the same effect. With singletons, life is much simpler: each class name describes a disjoint part of the state, and a call to C.m() can never be the same as a call to D.m().

prototype

machine

```
class Machine {
    StateComponent comp1 = ... ;
    StateComponent 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
```

This is just like Machine as Class, except that we have methods instead of procedures (static methods), and the state components are local to the object representing the machine. Note that the machine has to be created before we can use it.

example: 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);
   }
...}
```

The variable recording can be viewed as an example of the Machine as Object pattern -- in fact, just about every mutable object fits this pattern (which is probably why it doesn't have an official name!).

example: 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?*

The variable recording can be viewed as an example of the Machine as Object pattern -- in fact, just about every mutable object fits this pattern (which is probably why it doesn't have an official name!).

example: 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?*

The variable recording can be viewed as an example of the Machine as Object pattern -- in fact, just about every mutable object fits this pattern (which is probably why it doesn't have an official name!).

example: dispatcher

```
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);
         }
     });

...}
```

Listener pattern again, but something subtle here. The keyPressed method inside the newly created listener object makes method calls on the object named by the variable machine, which is declared outside. This is called a 'closure'. Functional languages (such as LISP, Scheme and ML) offer more powerful closure mechanisms. In Java, there a bit restricted. Note that any variable referred to inside the closure that came from outside must be declared as final. This can be quite a nuisance.

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

Fall through in a switch statement in C was responsible for a bug that brought down the AT&T phone system in January 1990. See:

http://catless.ncl.ac.uk/Risks/9.61.html#subj2
http://catless.ncl.ac.uk/Risks/9.69.html#subj5

What I mean by one 'mode' variable is this. Often the state components involve various data structures, counters, etc but only one 'control state' component that has typically between 2 and 10 values. These values are called 'modes'.

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;
        }
    ...
}
```

example: machine

```
  public class PianoMachine {
     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;
         }
     }
 ...
 }
```

PLAYING_PRIOR_TO_FIRST_RECORDING, RECORDING, PLAYING are the 'modes'.
Note the return statement in the switch branch, and how the first case falls through into the second.
The additional state is used in the playback operation: if it's prior to the first recording, no playback occurs.

The main state machine pattern here is Machine as Object.

example: machine

```
 public class PianoMachine {
   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;
       }
   }
...
}
```

what 's the main state machine pattern being used here?

PLAYING_PRIOR_TO_FIRST_RECORDING, RECORDING, PLAYING are the 'modes'.
Note the return statement in the switch branch, and how the first case falls through into the second.
The additional state is used in the playback operation: if it's prior to the first recording, no playback occurs.

The main state machine pattern here is Machine as Object.

example: machine

```
public class PianoMachine {
  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;
      }
  }
...
}
```

what 's the main state machine pattern being used here?

*what code is executed when state is **PLAYING**? when state is **PLAYING_PRIOR**...?*

PLAYING_PRIOR_TO_FIRST_RECORDING, RECORDING, PLAYING are the 'modes'.
Note the return statement in the switch branch, and how the first case falls through into the second.
The additional state is used in the playback operation: if it's prior to the first recording, no playback occurs.

The main state machine pattern here is Machine as Object.

example: machine

```
  public class PianoMachine {
     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;
         }
     }
 ...
 }
```

what 's the main state machine pattern being used here?

*what code is executed when state is **PLAYING**? when state is **PLAYING_PRIOR**...?*

*how is additional state **PLAYING_PRIOR...** used?*

PLAYING_PRIOR_TO_FIRST_RECORDING, RECORDING, PLAYING are the 'modes'.
Note the return statement in the switch branch, and how the first case falls through into the second.
The additional state is used in the playback operation: if it's prior to the first recording, no playback occurs.

The main state machine pattern here is Machine as Object.

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

I like to think of the major transitions between modes as ‘low frequency’ behavior, and the transitions within modes as ‘high frequency’ behavior. Think of a protocol for getting email from a server: you login, download the mail messages, display them, etc. Each of these phases might correspond to a mode. Within each mode, there are lots of smaller state changes, such as going from one message to the next.

prototype

interface declaring states

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

class implementing a single (super) state

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

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

dispatcher

```
State state = new S1 ();
...
state = state.op1 ();
```

The basic idea here is that each transition creates a new state object, which gets bound to the variable called state in the dispatcher.

The dispatcher really looks different in this pattern. It's actually inconvenient enough to have to call the operations like this that it's often good to wrap the calls and introduce another class that's used in the Machine as Object style. This is what I actually did for the midipiano -- see sample code on next slide.

example: dispatcher

```
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);
    }
}
```

This is an example of wrapping the simple dispatch code. It's easy to understand but tedious to write. Note how the empty recording list has to be passed to the first playing state. We'll see more of this passing of state components on the next slide; it's a liability of this pattern.

example: machine

```
 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;
    }
...
}
```

You just need to grok this code to figure out what's going on. This pattern is a bit kinky but it can be effective in some cases. Note how the constructor takes a state component as an argument. In the (rare) ideal case, the state consists of just the main modes (such as Recording, Playing, etc), and additional subcomponents that are fully nested within these modes and never cross from mode to mode. But that doesn't work here, because the recording sequence, for example, is created in the Recording mode and then has to be remembered in other modes so that it can be played back later. There is one nice example of nested state here, though: only one recording sequence is needed in the other states, since it's only in the recording mode that you need to distinguish the current and last recording sequences.

I omitted the code for absorbing the key repeats here just to make this simpler. If I included it, I'd have to add another state component and pass it around. Aagh!

example: machine

```
 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?*

You just need to grok this code to figure out what's going on. This pattern is a bit kinky but it can be effective in some cases. Note how the constructor takes a state component as an argument. In the (rare) ideal case, the state consists of just the main modes (such as Recording, Playing, etc), and additional subcomponents that are fully nested within these modes and never cross from mode to mode. But that doesn't work here, because the recording sequence, for example, is created in the Recording mode and then has to be remembered in other modes so that it can be played back later. There is one nice example of nested state here, though: only one recording sequence is needed in the other states, since it's only in the recording mode that you need to distinguish the current and last recording sequences.

I omitted the code for absorbing the key repeats here just to make this simpler. If I included it, I'd have to add another state component and pass it around. Aagh!

example: machine

```
 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?*

state machine initialization is missing from this fragment. where does it go?

You just need to grok this code to figure out what's going on. This pattern is a bit kinky but it can be effective in some cases. Note how the constructor takes a state component as an argument. In the (rare) ideal case, the state consists of just the main modes (such as Recording, Playing, etc), and additional subcomponents that are fully nested within these modes and never cross from mode to mode. But that doesn't work here, because the recording sequence, for example, is created in the Recording mode and then has to be remembered in other modes so that it can be played back later. There is one nice example of nested state here, though: only one recording sequence is needed in the other states, since it's only in the recording mode that you need to distinguish the current and last recording sequences.

I omitted the code for absorbing the key repeats here just to make this simpler. If I included it, I'd have to add another state component and pass it around. Aagh!

example: machine

```
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?*

state machine initialization is missing from this fragment. where does it go?

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

You just need to grok this code to figure out what's going on. This pattern is a bit kinky but it can be effective in some cases. Note how the constructor takes a state component as an argument. In the (rare) ideal case, the state consists of just the main modes (such as Recording, Playing, etc), and additional subcomponents that are fully nested within these modes and never cross from mode to mode. But that doesn't work here, because the recording sequence, for example, is created in the Recording mode and then has to be remembered in other modes so that it can be played back later. There is one nice example of nested state here, though: only one recording sequence is needed in the other states, since it's only in the recording mode that you need to distinguish the current and last recording sequences.

I omitted the code for absorbing the key repeats here just to make this simpler. If I included it, I'd have to add another state component and pass it around. Aagh!

tradeoffs

advantages

- clean scoping of <u>nested</u> components
- in each mode, only the relevant subcomponents appear

disadvantages

- can't handle <u>orthogonal</u> 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

By orthogonal components, I mean that we have two components X and Y say, and the states are formed as combinations (X,Y). By nested components, I mean that we have a component X with values X1, X2, etc, and for each of these values we have some different substates: say values of a component Y in X1, of a component Z in X2, etc. In a statechart, orthogonal components appear as parallel submachines, and nested components appear as multiple states within a larger superstate. The State as Object pattern handles nested components well, but doesn't do well for orthogonal components since they need to be passed around.

putting things together

elements

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

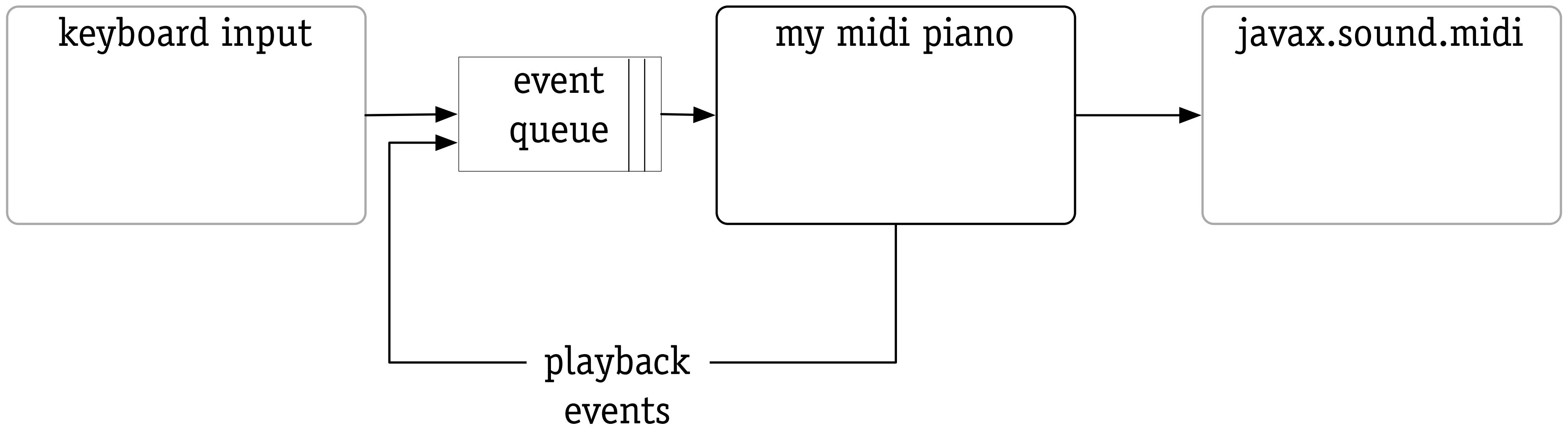
how to handle playback

- implement queue idea explained last time

modularity issues

- look at dependence diagram
- introduce interfaces for decoupling

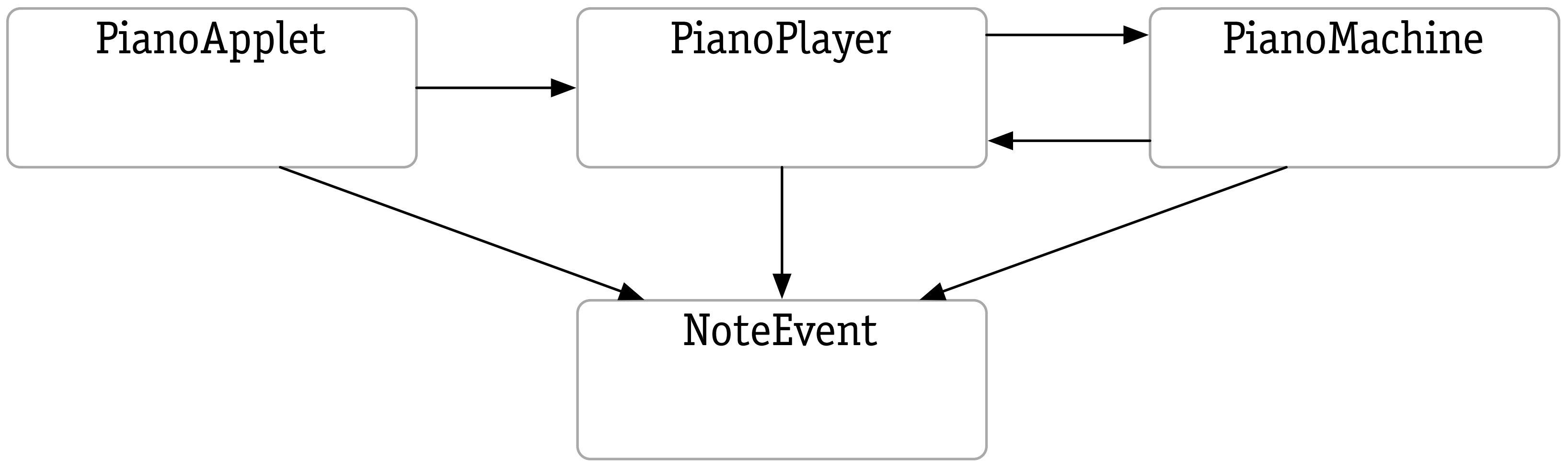
queue idea, from last time



idea

- playback generates press and release events
- merge these events on a queue with incoming keyboard events
- midi piano sees just one stream of events

dependency diagram



components

- PianoApplet: creates NoteEvents from key presses and passes to PianoPlayer
- PianoPlayer: maintains queue of events to be played
- PianoMachine: plays events, calls PianoPlayer to enqueue playback events

In a dependence diagram, an edge from A to B means that A knows about B and uses it: the name B appears somewhere in the code of A, and calls to B are made by A.

Note the backedge from PianoMachine to PianoPlayer: that's for playback and is due to the PianoMachine inserting events into the queue of the PianoPlayer. All the main modules use NoteEvent and its subclasses (not shown) because they all handle events.

event objects

```
public abstract class NoteEvent {
    protected final Pitch pitch;
    protected final int delay;

    public NoteEvent (Pitch pitch) {
        this (pitch, 0);
    }

    public NoteEvent (Pitch pitch, int delay) {
        this.delay = delay; this.pitch = pitch;
    }

    abstract public NoteEvent delayed (int delay);
    abstract public void execute (PianoMachine m);
    ...
}

public class BeginNote extends NoteEvent {
    ....
    public void execute (PianoMachine m) {
        m.beginNote (this);
    }

    public BeginNote delayed (int delay) {
        return new BeginNote (pitch, delay);
    }
```

The delay field is used for the playback events. The idea is that we measure the elapsed time between each pair of consecutive events, and attach that time to the second event. Then, when the events are taken off the queue and played back, we wait that amount of time before playing the event.

We'll see in a minute how the execute method is used.

event objects

```
public abstract class NoteEvent {
    protected final Pitch pitch;
    protected final int delay;

    public NoteEvent (Pitch pitch) {
        this (pitch, 0);
    }

    public NoteEvent (Pitch pitch, int delay) {
        this.delay = delay; this.pitch = pitch;
    }

    abstract public NoteEvent delayed (int delay);
    abstract public void execute (PianoMachine m);
    ...
}

public class BeginNote extends NoteEvent {
    ....
    public void execute (PianoMachine m) {
        m.beginNote (this);
    }

    public BeginNote delayed (int delay) {
        return new BeginNote (pitch, delay);
    }
```

*what's the **delay** field for?*

The delay field is used for the playback events. The idea is that we measure the elapsed time between each pair of consecutive events, and attach that time to the second event. Then, when the events are taken off the queue and played back, we wait that amount of time before playing the event.

We'll see in a minute how the execute method is used.

event objects

```
public abstract class NoteEvent {
    protected final Pitch pitch;
    protected final int delay;

    public NoteEvent (Pitch pitch) {
        this (pitch, 0);
    }

    public NoteEvent (Pitch pitch, int delay) {
        this.delay = delay; this.pitch = pitch;
    }

    abstract public NoteEvent delayed (int delay);
    abstract public void execute (PianoMachine m);
    ...
}

public class BeginNote extends NoteEvent {
    ....
    public void execute (PianoMachine m) {
        m.beginNote (this);
    }

    public BeginNote delayed (int delay) {
        return new BeginNote (pitch, delay);
    }
```

*what's the **delay** field for?*

*note the **execute** method*

The delay field is used for the playback events. The idea is that we measure the elapsed time between each pair of consecutive events, and attach that time to the second event. Then, when the events are taken off the queue and played back, we wait that amount of time before playing the event.

We'll see in a minute how the execute method is used.

dispatcher

```
    public class PianoApplet extends Applet {
    public void init() {
        final PianoPlayer player = new PianoPlayer();

        addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                char key = (char) e.getKeyCode();
                switch (key) {
                case 'I': player.nextInstrument(); return;
                case 'P': player.requestPlayback(); return;
                case 'R': player.toggleRecording(); return;
                }
                NoteEvent ne = new BeginNote (keyToPitch (key));
                player.request (ne);
            }
        });
...}
```

Unlike in the earlier simplified example, PianoApplet calls PianoPlayer, not PianoMachine, since this object holds the queues and handles the feedback.

player

```
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 request (NoteEvent e) {
         queue.put(e);
    }

     public void requestPlayback (){
         machine.requestPlayBack();
     }

    public void toggleRecording (){
        machine.toggleRecording();
    }

     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);
    }
}
...}
```

There are two queues, one for events to be executed immediately (called just queue), and one for events that are delayed (called delayQueue). There are two threads running here, one taking events off each queue. The thread taking events off the delayed queue waits the appropriate amount of time for each event, then puts it on the immediate queue. This way the events being played back are timed appropriately but the manual keypresses are not kept waiting, since they go straight on the immediate queue.

Now, seeing this code, you should be able to understand why the execute method of NoteEvent was defined. Without the ‘receiver flipping’ (my term) in which a method call is made on an event, and the event then calls the method on the state machine, we would have to write code here to handle the cases for the different events (begin and end). This way, the cases are nicely separated into the subclasses of NoteEvent.

player

how many threads are running here?

```
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 request (NoteEvent e) {
        queue.put(e);
    }

    public void requestPlayback (){
        machine.requestPlayBack();
    }

    public void toggleRecording (){
        machine.toggleRecording();
    }

    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);
    }
}
                                    ...}
```

There are two queues, one for events to be executed immediately (called just queue), and one for events that are delayed (called delayQueue). There are two threads running here, one taking events off each queue. The thread taking events off the delayed queue waits the appropriate amount of time for each event, then puts it on the immediate queue. This way the events being played back are timed appropriately but the manual keypresses are not kept waiting, since they go straight on the immediate queue.

Now, seeing this code, you should be able to understand why the execute method of NoteEvent was defined. Without the 'receiver flipping' (my term) in which a method call is made on an event, and the event then calls the method on the state machine, we would have to write code here to handle the cases for the different events (begin and end). This way, the cases are nicely separated into the subclasses of NoteEvent.

player

how many threads are running here?

why two queues?

```
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 request (NoteEvent e) {
        queue.put(e);
    }

    public void requestPlayback (){
        machine.requestPlayBack();
    }

    public void toggleRecording (){
        machine.toggleRecording();
    }

    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);
    }
}
...}
```

There are two queues, one for events to be executed immediately (called just queue), and one for events that are delayed (called delayQueue). There are two threads running here, one taking events off each queue. The thread taking events off the delayed queue waits the appropriate amount of time for each event, then puts it on the immediate queue. This way the events being played back are timed appropriately but the manual keypresses are not kept waiting, since they go straight on the immediate queue.

Now, seeing this code, you should be able to understand why the execute method of NoteEvent was defined. Without the ‘receiver flipping’ (my term) in which a method call is made on an event, and the event then calls the method on the state machine, we would have to write code here to handle the cases for the different events (begin and end). This way, the cases are nicely separated into the subclasses of NoteEvent.

player

how many threads are running here?

why two queues?

```
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 request (NoteEvent e) {
        queue.put(e);
    }

    public void requestPlayback (){
        machine.requestPlayBack();
    }

    public void toggleRecording (){
        machine.toggleRecording();
    }

    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);
        }
    }
...}
```

*why requeue the event in **processDelayQueue**? why not just execute it directly?*

There are two queues, one for events to be executed immediately (called just queue), and one for events that are delayed (called delayQueue). There are two threads running here, one taking events off each queue. The thread taking events off the delayed queue waits the appropriate amount of time for each event, then puts it on the immediate queue. This way the events being played back are timed appropriately but the manual keypresses are not kept waiting, since they go straight on the immediate queue.

Now, seeing this code, you should be able to understand why the execute method of NoteEvent was defined. Without the ‘receiver flipping’ (my term) in which a method call is made on an event, and the event then calls the method on the state machine, we would have to write code here to handle the cases for the different events (begin and end). This way, the cases are nicely separated into the subclasses of NoteEvent.

player

how many threads are running here?

why two queues?

*why the receiver flipping in **e.execute(m)**?*

```
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 request (NoteEvent e) {
        queue.put(e);
    }

    public void requestPlayback (){
        machine.requestPlayBack();
    }

    public void toggleRecording (){
        machine.toggleRecording();
    }

    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);
    }
}
...}
```

*why requeue the event in **processDelayQueue**? why not just execute it directly?*

There are two queues, one for events to be executed immediately (called just queue), and one for events that are delayed (called delayQueue). There are two threads running here, one taking events off each queue. The thread taking events off the delayed queue waits the appropriate amount of time for each event, then puts it on the immediate queue. This way the events being played back are timed appropriately but the manual keypresses are not kept waiting, since they go straight on the immediate queue.

Now, seeing this code, you should be able to understand why the execute method of NoteEvent was defined. Without the ‘receiver flipping’ (my term) in which a method call is made on an event, and the event then calls the method on the state machine, we would have to write code here to handle the cases for the different events (begin and end). This way, the cases are nicely separated into the subclasses of NoteEvent.

machine

```
public class PianoMachine {
    private final PianoPlayer player;
    private final Midi midi;

    public PianoMachine(PianoPlayer player) {
        ...
        this.player = player;                  // this allows the dependence back edge
    }

    public void beginNote (NoteEvent event) {
        Pitch pitch = event.getPitch();
        if (pitchesPlaying.contains(pitch))
            return;
        pitchesPlaying.add(pitch);
        midi.beginNote(pitch.toMidiFrequency());
        addToRecording(event);                     // encapsulates adding delay to event
    }

    public void requestPlayBack () {
        player.playbackRecording(lastRecording);
    }
    ...
}
```

The Midi state machine is implemented using Machine as Object; you can tell because the state of PianoMachine includes an instance of Midi. It's initialized in the constructor (although I omitted that here because the code's a bit lengthy as an exception needs to be handled).

The private method addToRecording (which is used here but whose declaration is not shown) encapsulates the timing between events and setting the delays on the created events.

machine

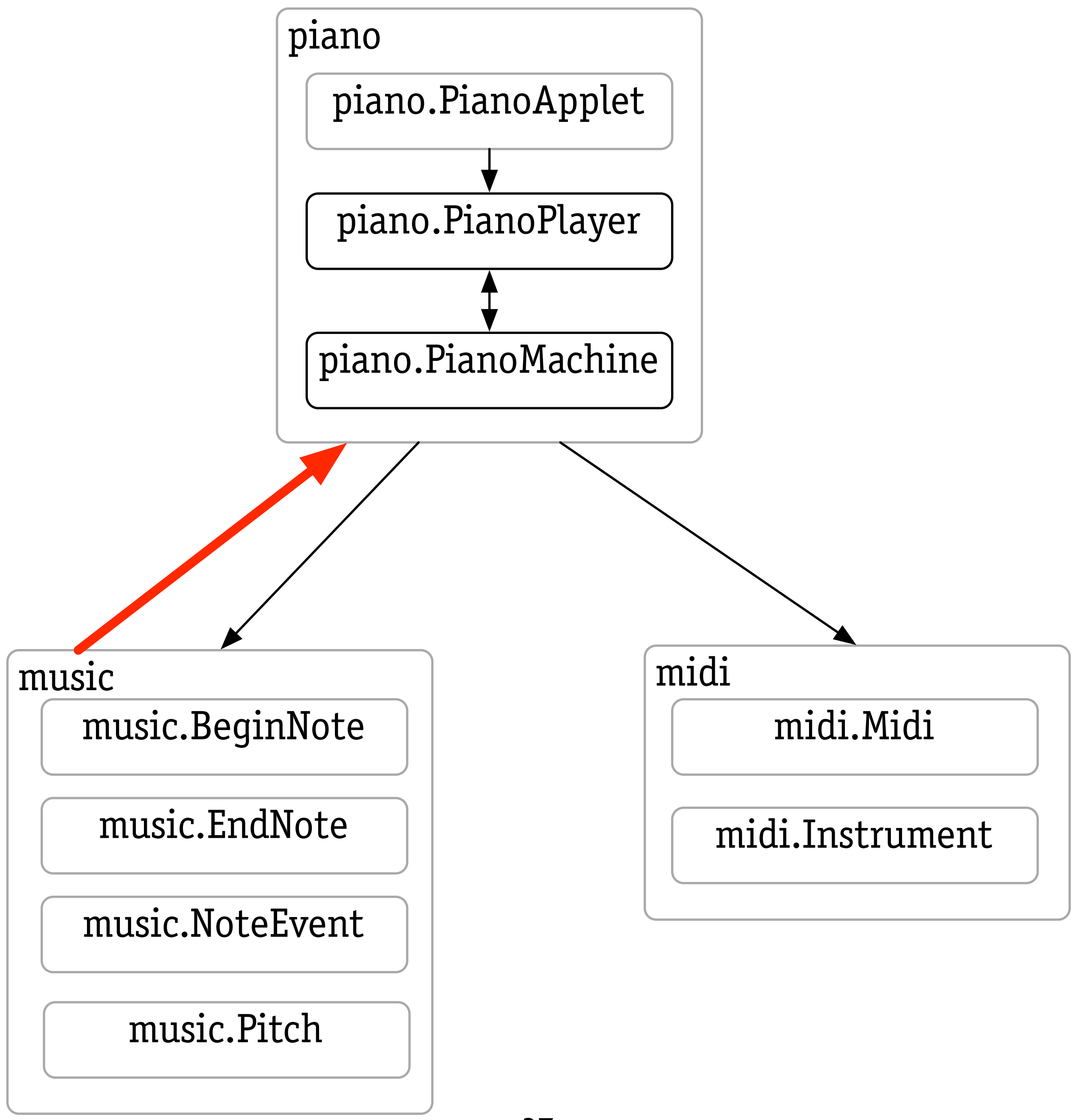
looking just at ***PianoMachine****, can you guess what state machine pattern* ***Midi*** *uses?*

```
public class PianoMachine {
    private final PianoPlayer player;
    private final Midi midi;

    public PianoMachine(PianoPlayer player) {
        ...
        this.player = player;                   // this allows the dependence back edge
    }

    public void beginNote (NoteEvent event) {
        Pitch pitch = event.getPitch();
        if (pitchesPlaying.contains(pitch))
            return;
        pitchesPlaying.add(pitch);
        midi.beginNote(pitch.toMidiFrequency());
        addToRecording(event);                      // encapsulates adding delay to event
    }

    public void requestPlayBack () {
        player.playbackRecording(lastRecording);
    }
    ...
}
```

The Midi state machine is implemented using Machine as Object; you can tell because the state of PianoMachine includes an instance of Midi. It’s initialized in the constructor (although I omitted that here because the code’s a bit lengthy as an exception needs to be handled).

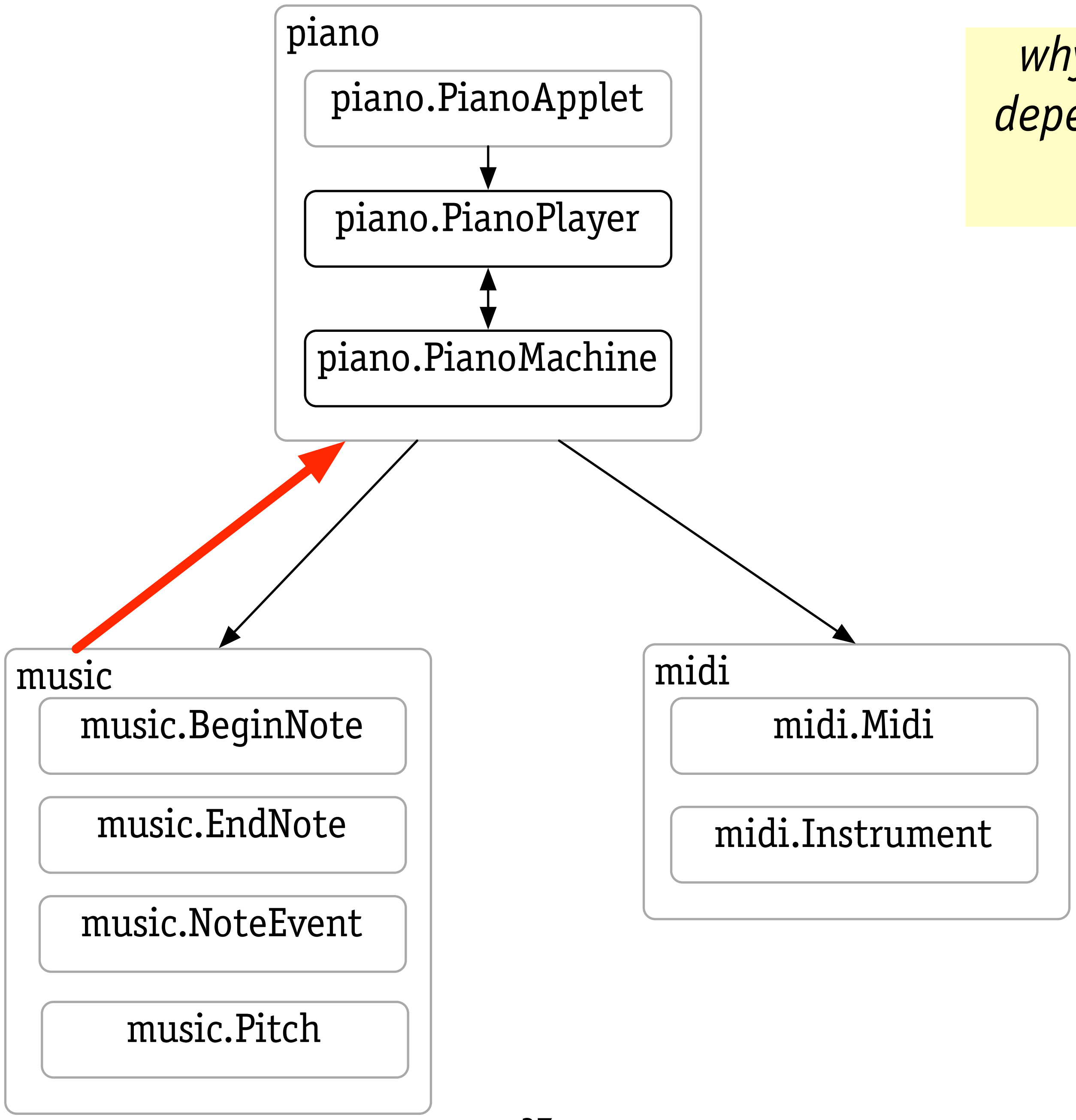
The private method addToRecording (which is used here but whose declaration is not shown) encapsulates the timing between events and setting the delays on the created events.

dependency design



Note the insidious backedge in the dependence diagram. This occurs because of that call in the execute method of the subclasses of NoteEvent to a PianoMachine object. With this backedge present, we don't have a layered architecture: the lower layer depends on the upper layer, and we couldn't even compile the music package without having piano.PianoMachine around.

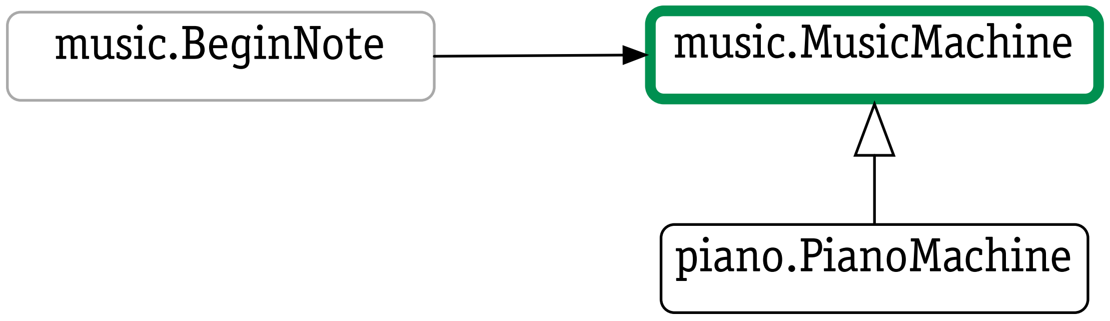
dependency design



*why does **music** depend on **piano**?*

Note the insidious backedge in the dependence diagram. This occurs because of that call in the execute method of the subclasses of NoteEvent to a PianoMachine object. With this backedge present, we don't have a layered architecture: the lower layer depends on the upper layer, and we couldn't even compile the music package without having piano.PianoMachine around.

interfaces to the rescue!



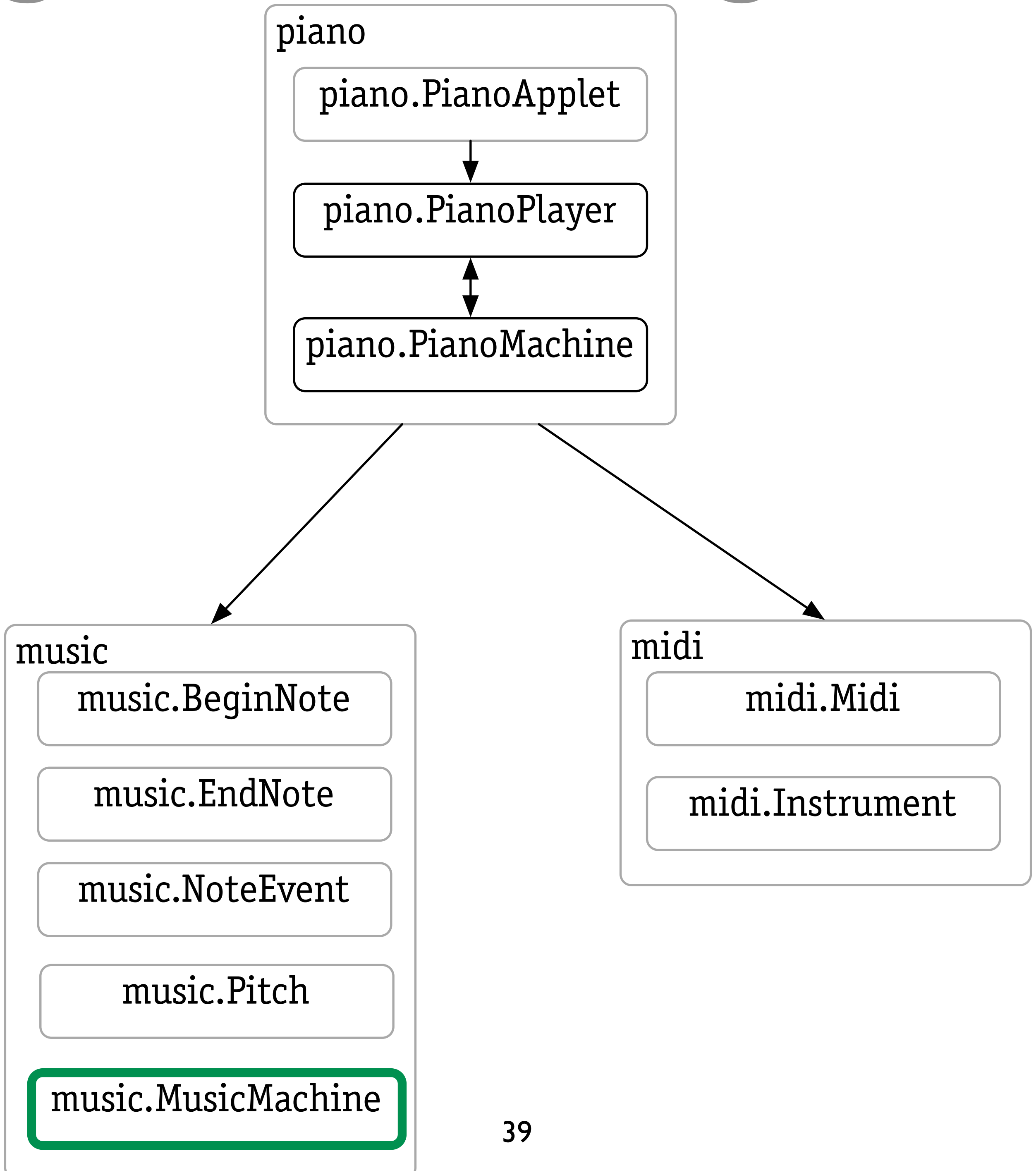
how it works

- NoteEvent classes access PianoMachine through interface MusicMachine
- no longer dependent on PianoMachine

```
public interface MusicMachine {
    public void beginNote (NoteEvent event);
    public void endNote (NoteEvent event);
}

public class BeginNote extends NoteEvent {
    public void execute (MusicMachine m) {
        m.beginNote (this);
    }
    ...
}
```

We can eliminate the dependence by factoring out the properties of PianoMachine that NoteEvent and its subclasses need: just the existence of two methods. Now a class like BeginNote no longer has a dependence on piano.MusicMachine (even though it makes calls to it at runtime).

cutting the back edge



Here's the new dependency diagram. By adding the new interface in the music package, we've made the music package no longer dependent on the piano package, and the system is now layered, so that music could be reused independently of the other packages.

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

principle: layering is good

- cycles make programs hard to understand

principle: use interfaces for decoupling

- here, we broke a cycle by introducing an interface
- more on this in decoupling lecture later

puzzles for the enthusiastic

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?

Here are some puzzles for you to think about when you're studying the code of the midi piano (which is available in the repository). This is optional but you'll learn a lot from trying to do it.