elements of software construction

implementing state

Fall 2010
Recall: a midi piano

**functionality required**
- play notes on computer keyboard
- sustain by holding key down
- cycling through instruments
- record and playback
- layered recordings

**context**
- piano depends on keyboard, midi API
- this is a dependence diagram -- more later

**need to start by understanding**
- keyboard input: press and release
- MIDI interface: commands
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
This set of machines monitor key presses and enqueue begin and end note events. They ignore the P and R keys.
Recording machine receives keyboard R input and dequeued events from queue. If recording mode is ON, it updates the **current** and **last** recording lists. It sends dequeued events to the MIDI API to be played.

```
prR / current = <>
```

deq bn(k) / current += bn(k), play bn(k)
deq en(k) / current += en(k), play en(k)

```
prR / last = current
```

deq functionality could have been specified separately and is only included here for convenience. Essentially all dequeued events are played by sending to the MIDI API. If recording mode is ON, then the dequeued events are added to the current recording list.
Playback machine receives keyboard P input. It enqueues begin note and end note events from the last recording list onto queue if playback mode is enabled.
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.
PianoApplet calls PianoPlayer since this object holds the queues and handles the feedback. addKeyListener and KeyAdapter are from packages in java.awt.* and “listen” for keys. We will talk about the listener design pattern later in the course.
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. event.execute(machine) will result in machine.beginNote(event) or machine.endNote(event) depending on whether the event is a BeginNote or EndNote.
beginNote sends the event to MIDI and also updates the recording list.

The private method addToRecording encapsulates the timing between events and setting the delays on the created 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.

NoteEvent executes the event on PianoMachine and hence depends on PianoMachine.
Note the insidious red edge 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 edge 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

```java
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).
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.