Until now, we’ve been doing a crash course in Java. This is the first lecture of the course proper, starting with state machines -- our first software paradigm.
a lesson from failure
A tragic story reported in the Washington Post.
what can we learn from

accidents are complex
‣ rarely one cause, so be wary of simple explanations
‣ often human factors + technology

lessons from this case
‣ design for robustness against all likely failure modes
‣ defaults are dangerous: user should be warned
‣ describe and analyze all usage scenarios

There are lots of problems here that could be analyzed. For example, there's clearly a user interface problem that the device didn't clearly indicate when it was showing calculated coordinates versus the coordinates of the user. Like many accidents, this one involved human factors issues in which the designers didn't think through all the ways that user might interact with the machine. But whatever the problem was, it's clear that we need some way to talk about the possible scenarios in simple but precise way.
State machines give us this simple and precise way of talking about scenarios. Don't confuse abstract with vague! They're abstract in the sense that they allow you to express only the important bits, but that isn't the same as being vague. A state machine doesn't tell you everything there is to know about a system, but what it does tell you -- namely what orders events can happen in -- it tells you very precisely.
designing a midi piano
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

We’ll see more of these kinds of diagrams later. They’re called _dependency diagrams_. Note that the edges aren’t labelled: that’s a clue that they’re not state machines! Sometimes people try and make these various notations look different from one another (eg, by making the boxes slightly different shapes), but our experience is that it’s more trouble than it’s worth: easier just to use plain old boxes and arrows and label your diagrams so you know what they are.

Note, btw, that there’s a big decision that’s already been made here: what the context actually is. We could have chosen to use a different API for making the musical sounds, and we could have implemented our own keyboard driver. Just defining where your system sits with respect to the outside world and your assumptions about it is a big step.
modeling the context
a single key

the state machine
' two states, UP and DOWN
' UP is the initial state
' two input event classes, with these designations:
  pr: the keyboard driver reports a key press
  rel: the keyboard driver reports a release

meaning: a set of traces
' a trace is a sequence of events
' traces of this machine are
  <>
  <pr>
  <pr, rel>
  <pr, pr, rel>
  ...

The designations are more important than you might think. The biggest mistake novices make with state machine modeling is that the event classes aren't right. Formulating a designation -- trying to say exactly what comprises the occurrence of an event -- is a big help in checking whether your notion of the event class is reasonable, and in conveying it to others. Here the crucial idea is that pr, for example, doesn't mean that the user actually pressed the key, which is an important distinction.

Interestingly, not all keyboard drivers generate key repeats in this form. In Gnome, for example, the driver actually repeats release-press pairs -- in the notation we'll see next week, the repeat is (rel pr)* rather than pr*. That's bad news, because it's not at all clear how the generated repeats can be distinguished from the real key presses.

Note that the trace set is infinite if the machine has a loop in it, although each trace is itself finite -- the history of events up to some point in time. This notion of traces was invented by Tony Hoare in a machine formalism called Communicating Sequential Processes. You may be a bit confused if you've seen the notion of languages and state machines in a computer science theory setting. There, the language usually consists of the set of all complete sequences, which are defined because there are some special states that are marked as final. Here we only have initial states.
the whole keyboard

represent as parallel combination of state machines

' each key's machine can take steps independently
(behavior: shared events must be synchronized, but no sharing here)

' traces include <->, <pr1>, <pr1, rel1>, <pr1, rel1, pr2>, <pr1, pr2, rel1>

The parallelism here is just conceptual. To implement this kind of parallelism, you could actually have threads running concurrently, but usually you just have separate state components, one for each of the submachines, and the whole machine runs sequentially with one of the submachines taking a step at a time. When the ‘alphabets’ of events of the component submachines overlap, the shared events have to happen as one. In this case, there aren’t any shared events, but we’ll see examples of this below.
This is an overapproximation because it includes traces that can’t happen in practice. In a real keyboard driver, the keys are not independent. In many, for example, the key repeat on one key is terminated when another key is pressed, so the trace <pr1, pr2, pr1> can’t occur, since the pressing of key 2 prevents repeats of key 1.

Overapproximation is good especially when it allows us to use a simpler model. It doesn’t matter if we handle more input scenarios than actually happen.
state machine semantics
Here’s the mathematical definition of a state machine that we’re using. The diagram can be translated easily into this structure: each box becomes a member of the set of states, each label name becomes an event class, each edge becomes a tuple in the transition relation. In terms of what the machine actually means though we only care about the traces: the states are just an artifact to help us define the order in which the events can occur.

In general, there can be more than one initial state. This allows you to describe overapproximations.
There are different ways that a state machine formalism can handle inputs and outputs. This is perhaps the simplest way. We just say that after the input occurs, the output can happen -- and no other inputs until it’s happened. Note that we don’t actually distinguish inputs from outputs in any formal way, although of course when you designate the events you’ll say which are which. And we don’t have any notion of outputs following inputs immediately: if this was a component submachine in a parallel combination, another machine could take a step between the i and the o, so that the o doesn’t immediately follow the i. In most situations this is exactly what we want anyway.
Exercise

Draw state machines for these simple examples

- web download with start, cancel, complete
- MP3 player with select, play, pause, end
first design draft
This shows the result of starting with the basic keyboard and adding output events. I’ve written this as a generic machine, with the understanding that there is one of these for each key. I’ve assumed some function, note, that takes a key and returns a musical note associated with it. So begin note(k) means the begin event for the note associated with key k. Note that there’s no output generated for the repeated press events.
recording
adding recording

**design strategy**
- add a **parallel** machine that maintains additional state

**new events and states**
- events
  - pr\(_R\): keyboard driver reports press of R key
  - pr\(_k\): keyboard driver reports press of any key except for R
- states
  - !REC: recording off
  - REC: recording on

**design question**
- What about key repeats?

This machine parses a sequence of keyboard events, deciding whether we’re in the recording mode or not after a given sequence. If the recording mode is on, it adds keyboard events to a recording list.

There’s no need to worry about key repeats in this case, as we don’t expect the user to hold the R key down. If we were worried about them, we could filter them out the way we filtered them out for other keys.
playback
playback

let's add a playback mode

• use P key for playing back
  \( \text{pr}_P: \) keyboard driver reports P key pressed

• add an event to signal end of playback
  done: output event to signal end of playback

Assumptions

• No key presses during playback
• playback only enabled when recording is off

This is the most simple notion of playback - it occurs in a mode in which no recording or manual playing is allowed. Playing the recording means sending the begin note and end note events to the midi API.
More Functionality

We would like to:

‣ Merge and play key presses during playback
‣ Create layered recordings

For example:

‣ recording = A B C
‣ Start recording
‣ Start playback   A    B    C
‣ Press keys       D   E
‣ To get               A D B E C    in recording

How many recording lists will we need?
When the playback key is pressed, all the events in the recording are added to an input queue, along with whatever events are being generated by the keyboard driver. They can be interleaved in an arbitrary way so long as none of the events are delayed too long. There is, btw, a tricky issue to do with how the events are timed, which I’ve ignored. A simple solution is to timestamp each press or release with the delay since the last press or release, and have a process that delays putting the next playback event into the queue by that amount.
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.

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.
Issues Ignored in Modeling

‣ Timestamping of events

‣ How are playback events merged with key presses?

‣ How are the parallel state machines implemented?

Will need to worry about these in Java implementation (next time!)