elements of software construction

designing state machines

Daniel Jackson
September 22, 2008
formulas for software?
suppose you’re asked to write a program
  ‣ that calculates compound interest
  ‣ given principle P and rate R, what's value V after Y years?

what would you do first?
  ‣ you’d formalize it, eg: \( V = P (1+R)^Y \)

formulating helps pose questions
  ‣ how is the rate presented? as a fraction or percentage?
  ‣ what does “after” Y years mean?
  ‣ continuous or periodic compounding?
  ‣ accuracy required? range of values required?

lesson: need to **model** the problem precisely
a lesson from failure
friendly fire in afghanistan

Afghanistan, December 2001

• US soldier uses plugger* to mark Taliban position for air-strike
• notices battery-low warning, so replaces battery and calls in coordinates
• resulting strike kills user and two comrades and wounds 20 others

what happened?

• replacing battery reset to current position

*PLGR: precision lightweight global-positioning satellite receiver

what can we learn from this?

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
state machines

what they are

- a simple notation for describing behaviors
- succinct and abstract -- but not vague!
- basis for analysis and for implementation
our path

general strategy
- design behavior --> design mechanism
- three paradigms covering different aspects of software design

first paradigm: state machines
- today -- state machines for designing behavior
- thursday -- design patterns for implementing state machines
- tuesday -- analyzing state machines with invariants
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

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
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 whole keyboard

represent as parallel combination of state machines

- each key’s machine can take steps independently
  (general rule: shared events must be synchronized, but no sharing here)
- traces include $\langle\rangle, \langle\text{pr1}\rangle, \langle\text{pr1, rel1}\rangle, \langle\text{pr1, rel1, pr2}\rangle, \langle\text{pr1, pr2, pr1}\rangle$
approximate models

over-approximation

• this model is actually an over-approximation
• it allows more traces than can happen (eg \(<pr_1, pr_2, pr_1>\)) -- try it and see
• but this is OK: problem is handling fewer inputs than can occur
modelling the midi API

simple interface

' just turn each note on and off; need explicit initialization
first design draft
design strategy

- base structure on input and embed output events
- absorb superfluous presses
state machine semantics
the i/o shorthand

input/output shorthand
• i/o label is just a shorthand

subtle consequence
• o need not follow i immediately
• another event can intervene
a state machine is ...

• a set of states
  \[ \text{State} = \{ \text{UP, DOWN, PRESSED, RELEASED} \} \]

• a set of initial states
  \[ \text{Init} = \{ \text{UP} \} \]

• a set of events
  \[ \text{Event} = \{ \text{pr, rel, begin, end} \} \]

• a transition relation
  \[ \text{trans} \subseteq \text{State} \times \text{Event} \times \text{State} = \]
  \[ \{ (\text{UP, pr, PRESSED}), (\text{PRESSED, begin, DOWN}), (\text{DOWN, pr, DOWN}), (\text{DOWN, rel, RELEASED}), (\text{RELEASED, end, UP}) \} \]

• trace set (derived from \text{trans} and \text{Init})
  \[ \text{traces} = \{ <>, <\text{pr}>, <\text{pr, begin}>, <\text{pr, begin, pr}>, <\text{pr, begin, rel}>, ... \} \]
recording
draw a state machine where

- pressing R key toggles between recording on and off
- and switches indicator light to red (on) and green (off)
- ignore actual recording for now
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

• do we need to worry about key repeats?
state = (control-state, data-state)

- diagram only shows control state
- to show data state, use textual notation

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

op pr-R
  when true
  do
    if (! REC) recording = <>;
    REC = !REC

op pr (k)
  when true
  do
    if (REC)
      recording = recording ^ <pr_k>
```
check your understanding

what difference would this make?

• instead of test in postcondition

\[
\text{op \ pr (k)}
\]
\[
\text{when true}
\]
\[
\text{do}
\]
\[
\text{if (REC)}
\]
\[
\text{recording} = \text{recording} \land <\text{pr}_k>
\]

• use precondition

\[
\text{op \ pr (k)}
\]
\[
\text{when REC}
\]
\[
\text{do}
\]
\[
\text{recording} = \text{recording} \land <\text{pr}_k>
\]
analyzing recording

consider traces

• how about this trace?
  \(<pr(1), pr_R, rel(1), pr_R>\)

• very strange behavior: recording starts with key release!

two simple options

• filter out initial key releases
• or just allow it (which is what we’ll do)

or perhaps

• regard as evidence of larger issue
• maybe recording should be of musical structure and not of user events?
playback
playback

let’s add a playback mode

- use P key for playing back
  \( pr_P \): keyboard driver reports P key pressed
- add an event to signal end of playback
  \( done \): output event to signal end of playback

first design option

- playback only enabled when recording is off
- no key presses during playback
record during playback

can you record during playback?
  · very useful, but much trickier

what does playback involve?
  · generated playback events merged with user keyboard events?
an asynchronous solution

---

**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
the playback machine

another over-approximation

- which enq events are generated will depend on what was recorded
- (that is, reads recording state component)
more on recording state

what would happen if

• playback is pressed during record
• generated events get played
• and added to recording
• so they get played back again...

oops!

• keep two recordings
  - **current**: holds events being recorded
  - **last**: holds events being played back
• show how these are updated in textual syntax
state
List<Event> last, current = <>;
boolean REC = false;

op pr_R
  do
    if (REC) last = current else current = <>
    REC = ! REC

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

op pr_P
  do // enq events in last

// .. and similar op for rel(k)
final state machine

```
state
    List<Event> last, current = <>;
    boolean REC = false;

op pr_R
    do if (REC) last = current else current = <>; REC = ! REC

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

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

op pr_P
    do // enq events in last
```
summary
principles

description before invention

• before you invent something, understand what's there
• especially important for your design’s environment
• eg, keyboard driver and midi API

behavior before mechanism, or ‘what before how’

• design the behavior of a system first
• easier to design the mechanism when you know what the behavior is
• and easier to analyze the behavior when you can ignore mechanism
• eg, state machines before Java classes

get the details right

• details matter for quality, and can hide major structural flaws
• eg, details of playback during record determine gross structure
addendum
modeling advice

grounding in reality

• select events first and thoughtfully
• give explicit designations saying what they mean
• must be atomic and recognizable

bad smells

• events not designated, event classes not really disjoint
• no initial state marked
• states or transitions without labels

a state machine is not a control flow graph

• no behavior inside states
• no ‘decision edges’