Event-Driven Systems

Learning Goals

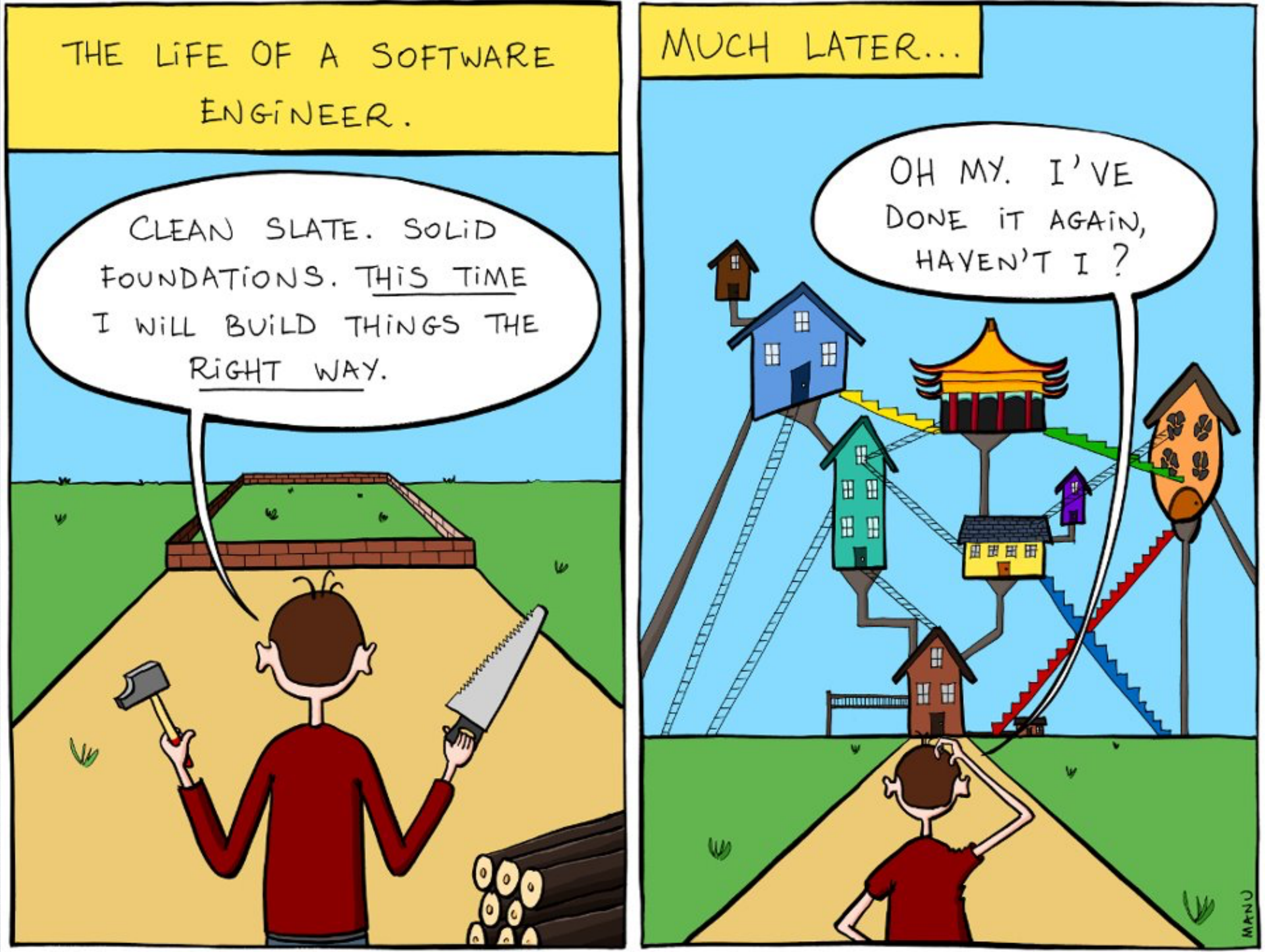
- Be able to explain why architectural patterns are helpful in structuring software.
- Understand the principles of event-driven systems.

Context

- Software Architecture is about dividing systems into pieces (components)
- Enables reasoning about components individually
- Consider the alternative
- "Hey, if I change this, will it break YOUR code?"

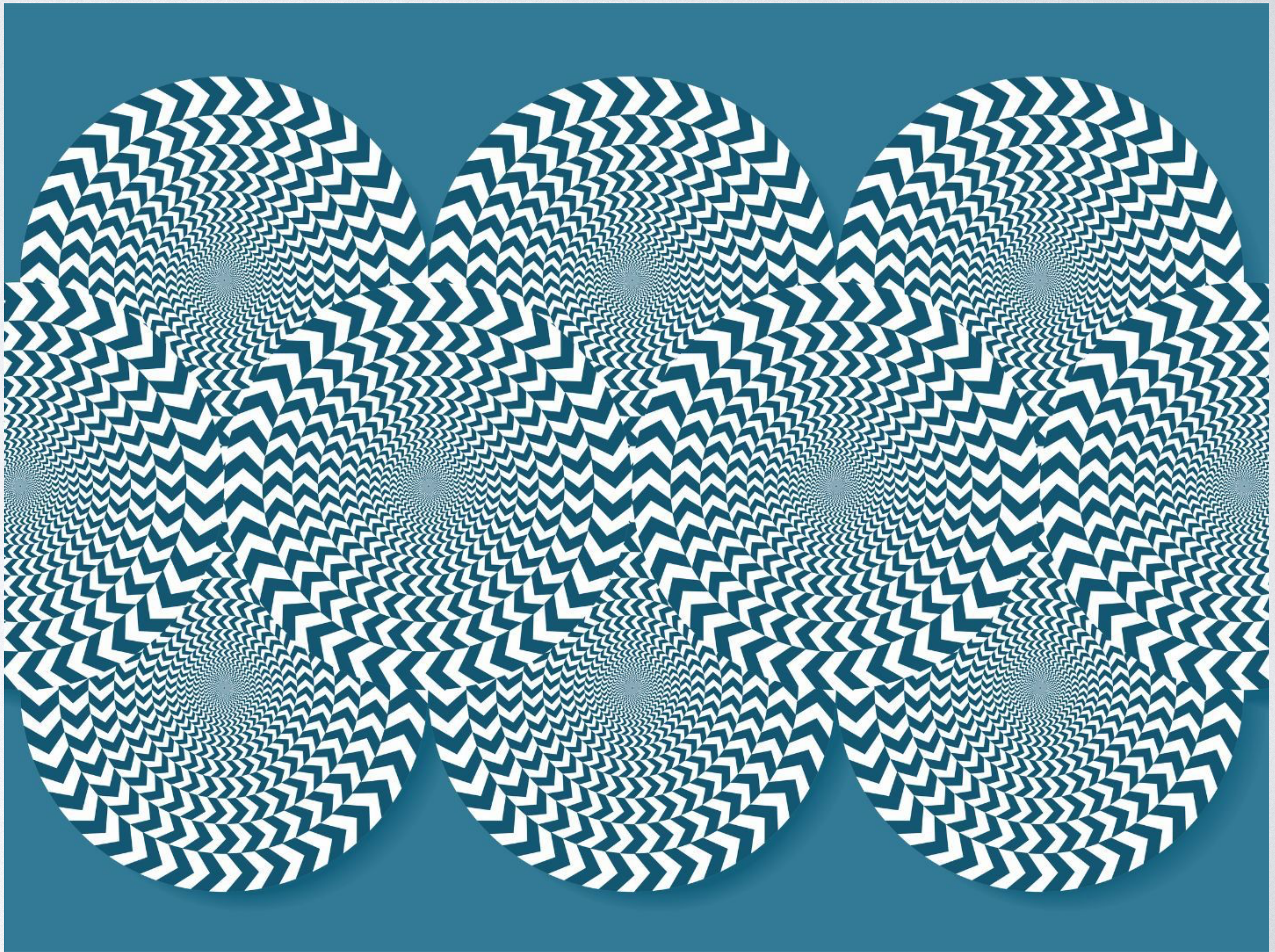
Architecture Is Big

- Maybe you haven't seen systems big enough to really need it
- But every system has an architecture
  - If no one designed it, it's probably not a good one


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU

I Will Train You To See Patterns

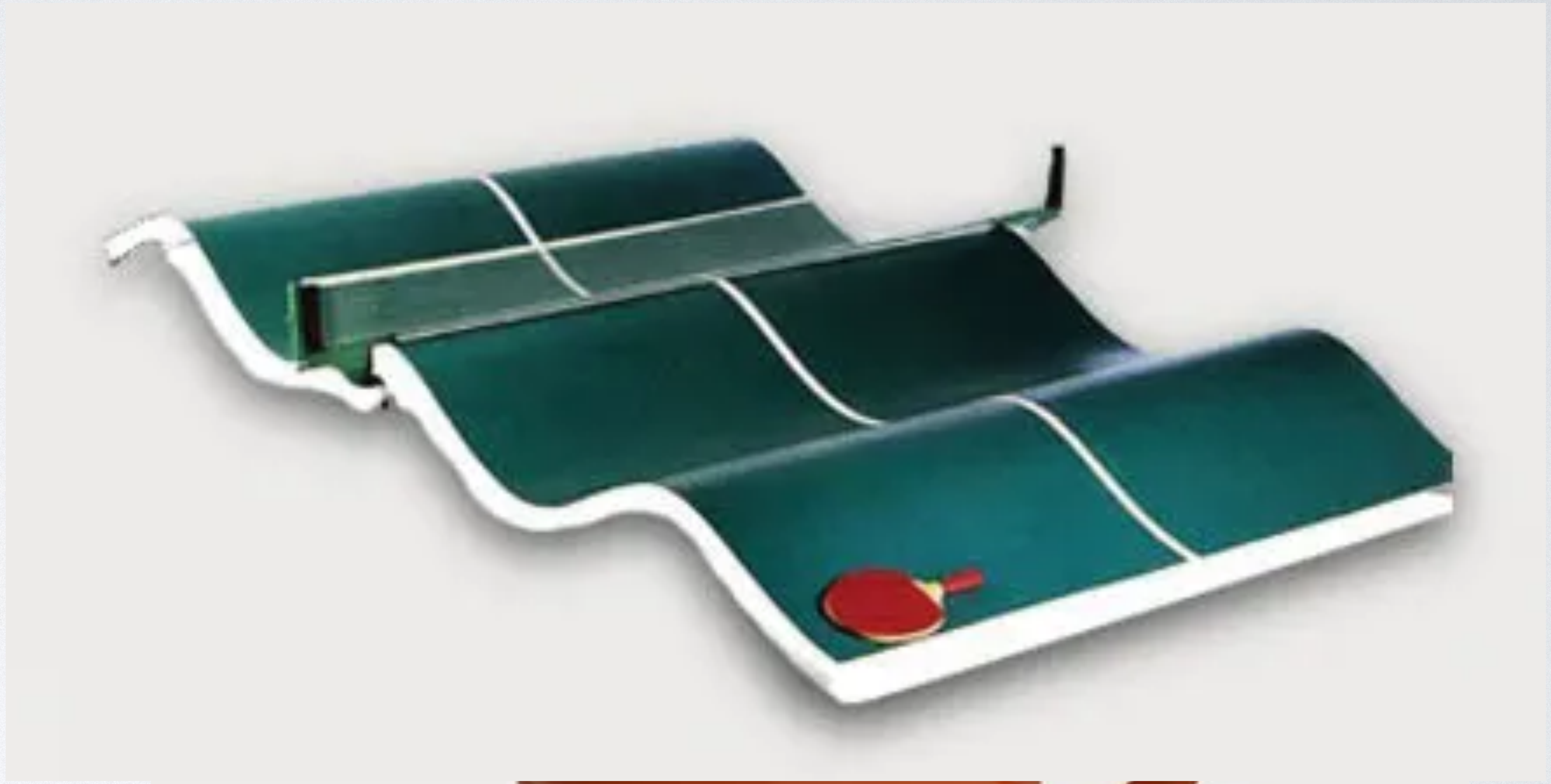
- You see circles.
- But are there really circles here?



Patterns Have Purpose

- This camouflage pattern is intended to help its wearer blend into the background

Form Follows Function



- Patterns serve purposes

- We study patterns so you can achieve goals

https://www.itsnicethat.com/articles/impossible-objects

Innovation

- Before innovating, first know the existing patterns
- Afterward, you can break the rules
- Standard solutions solve common problems
- Innovative solutions solve novel problems

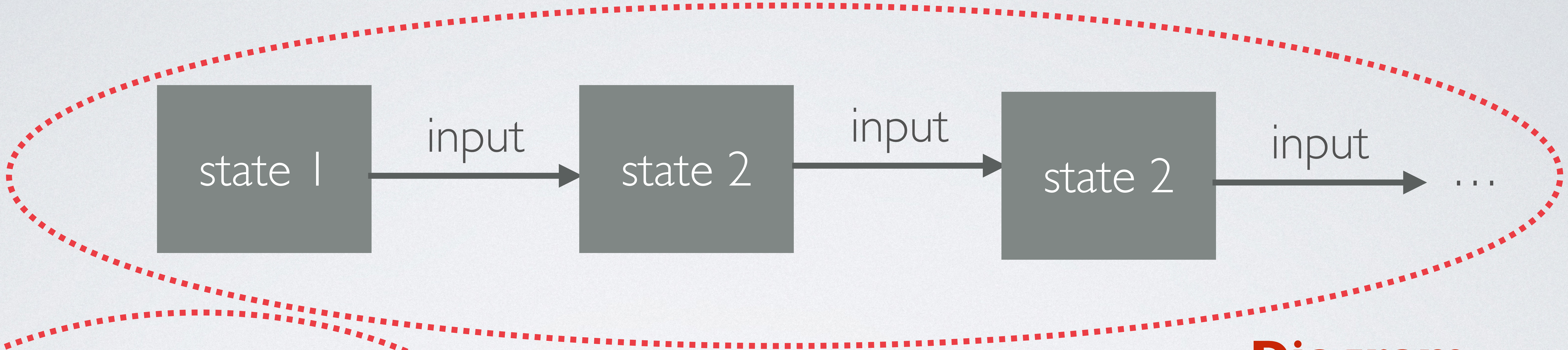
Let's Get Concrete

- In the past, I tried to start by teaching particular patterns
- But the patterns are *abstract*
- This year, I'm trying something different.
- Let's start with a concrete design problem and try to discover good patterns.

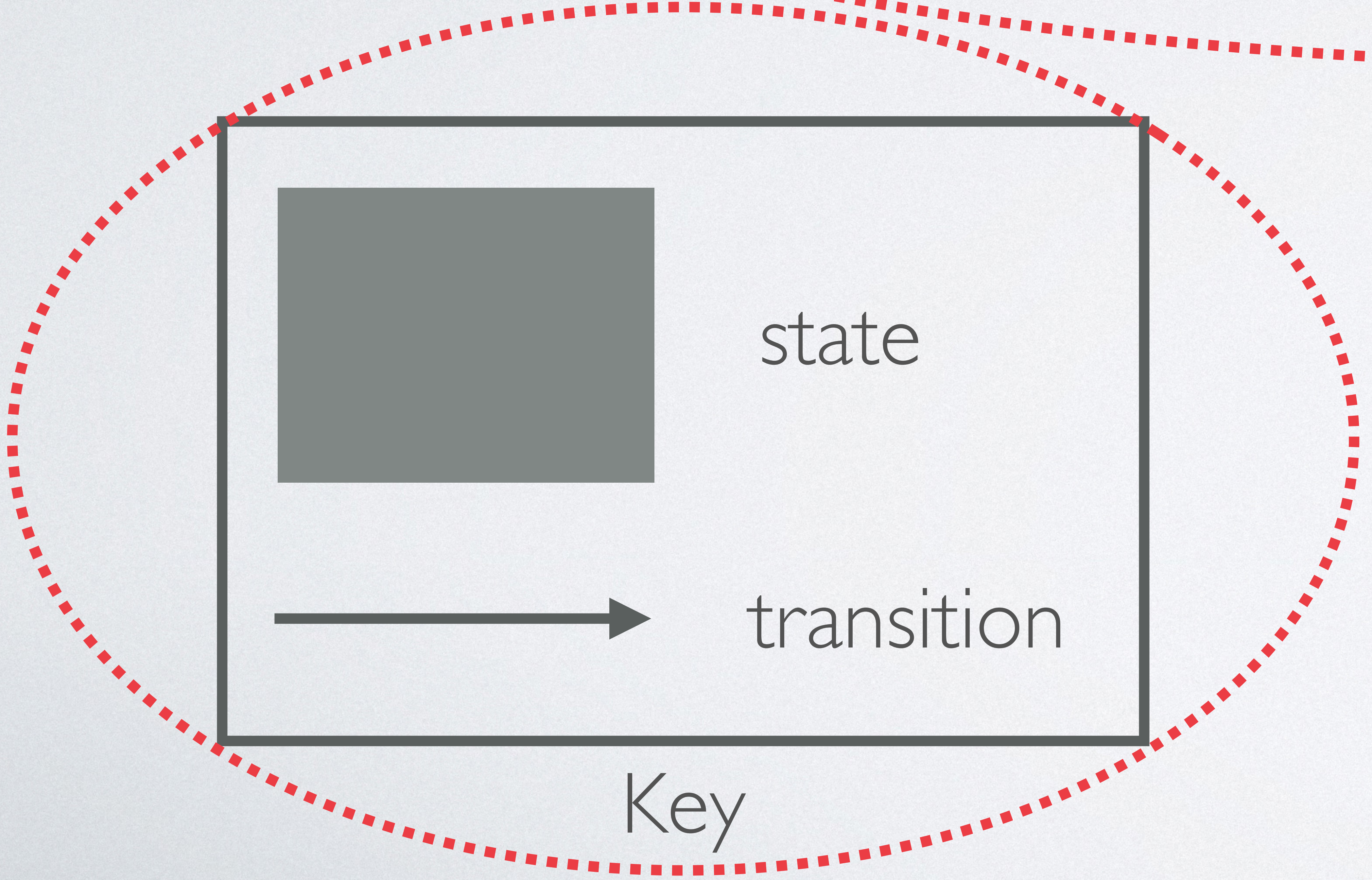
Interactive Systems

- Perhaps you are accustomed to *batch* systems:
  - Program takes input, runs, generates output, terminates.
- But most software you use is *interactive*
  - You get upset when your web browser exits!

Interactive Systems Respond to Input



Diagram

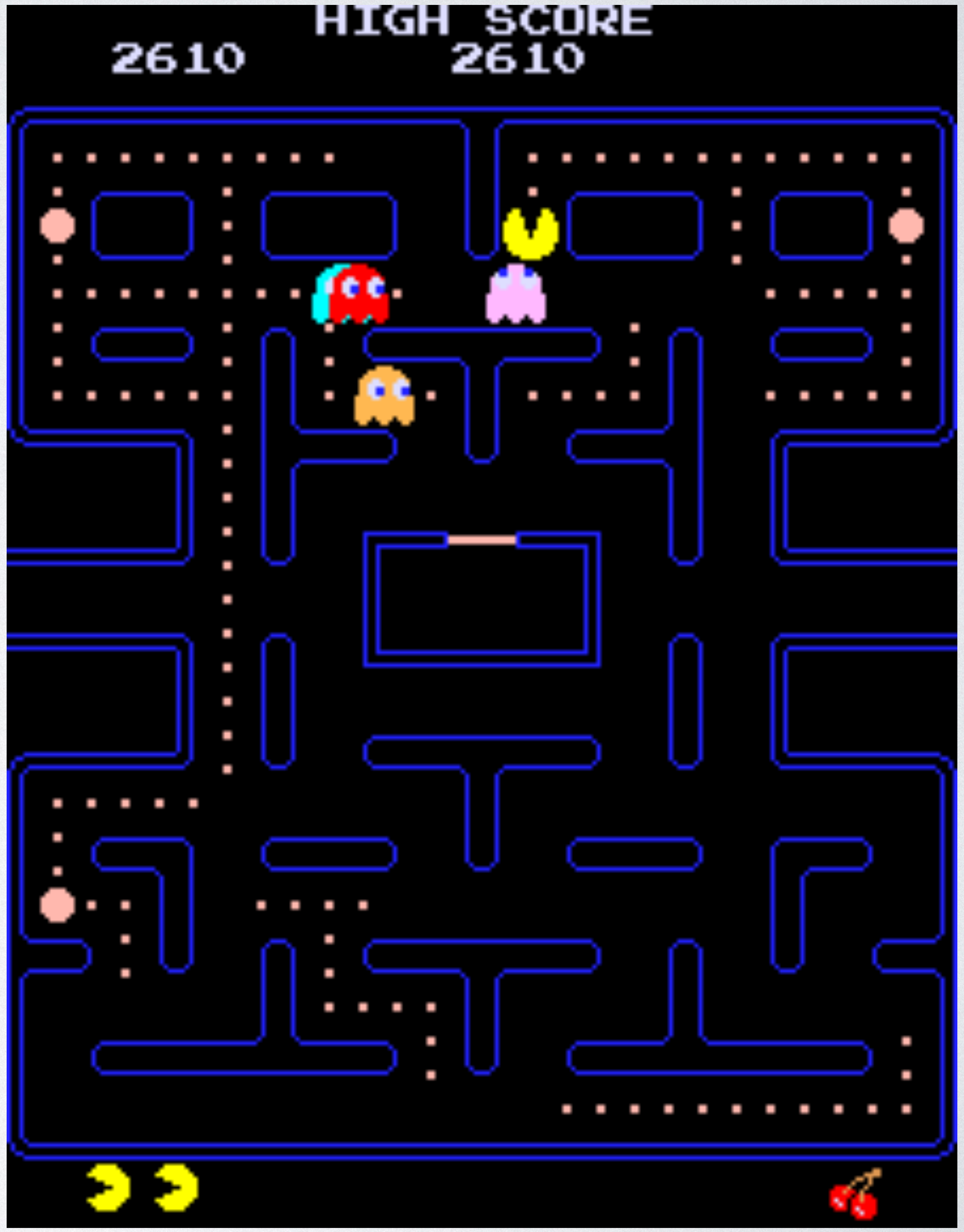


Key

- Diagrams communicate designs
- This is a *state transition diagram*
- Every diagram needs a key

A First Try

- Goal: implement Pac-Man
- Pac-Man is stopped or is moving
  ⬆️⬅️➡️⬇️
- Interaction: pressing arrow keys changes direction



A First Try

```
dir = STOP;
state = initialState();
while (!gameOver) {
  state.update(dir); // move one unit in the current direction
  if keyboard.leftArrowDown() {
    dir = LEFT;
  }
  … // more cases and game logic here
}
```

Analysis

- Simple design (good!)
- As CPU speed increases, game becomes unplayable
- Old PCs had "turbo" buttons to set clock speed!

https://commons.wikimedia.org/wiki/File:486er_33MHz_DX2_Double_Speed_Iomega_Tape.jpg

Timing

```
dir = STOP;
state = initialState();
while(!gameOver) {
  state.update(dir); // move one unit in the current direction
  if keyboard.leftArrowDown() {
    dir = LEFT;
  }
  … // more cases and game logic here
}
```

What happens if the user presses the key, but releases it before this check?

The Alternative: Events

- Wikipedia: "An event is a detectable occurrence or change in state that the system is designed to monitor, such as user input, hardware interrupt, system notification, or change in data or conditions."
- Me: An event represents something that happened.
- Examples: mouse down; mouse drag; key press; tap on touchscreen

The Event Loop

```
void main() {
  state = initialState();
  startStateUpdateThread(state);
  while (event = getNextEvent()) {
    handleEvent(event);
  }
}
void handleEvent(Event e) {
  switch (e.kind) {
    case KEYBOARD.LEFT:
      setDirection(LEFT);
    …
}
```

- All interaction happens in the event loop
- Time-based updates happen in another thread
- Code that implements interaction is isolated in `handleEvent` (doesn't pollute `main`)

A Real Example: MacPaint (1984)

```
BEGIN  { main program }
  InitGraf(@thePort);
  …
  OpenFirstDoc;

  REPEAT
    …
    IF GetNextEvent(everyEvent,theEvent) THEN ProcessTheEvent;

    …

    IF quitFlag THEN QuitProgram;

  UNTIL quitFlag;
  ExitToShell;
END.
```

https://computerhistory.org/blog/macpaint-and-quickdraw-source-code/

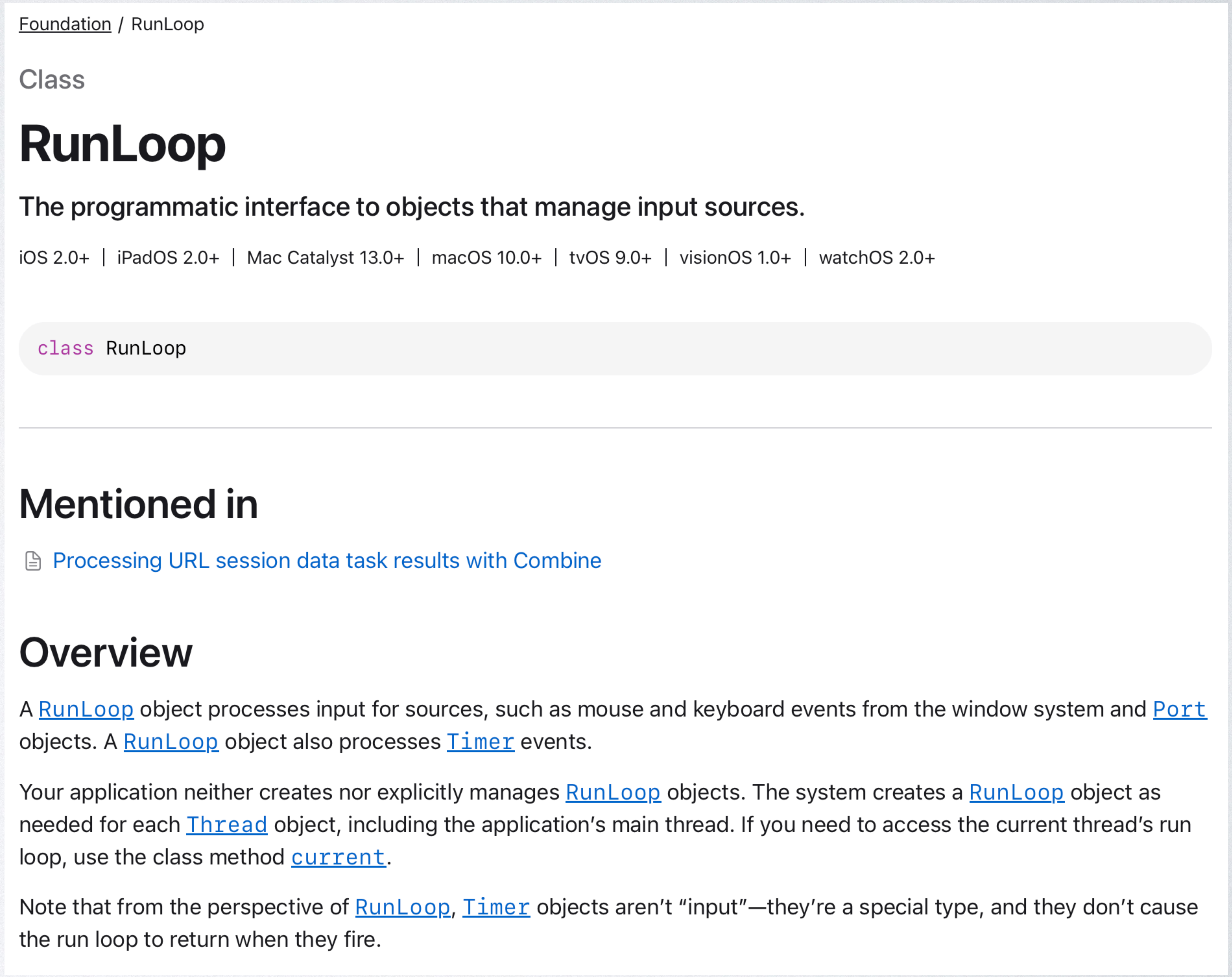
MacPaint (Simplified)

```
PROCEDURE ProcessTheEvent;
BEGIN
  shiftFlag   := (BitAnd(theEvent.modifiers,shiftKey) <> 0);
  CASE theEvent.what OF …
    mouseDown:
      BEGIN
        code := FindWindow(theEvent.where,whichWindow);

        IF (theEvent.when < clickTime + GetDoubleTime)
        AND NearPt(theEvent.where,clickLoc,4)
        THEN clickCount := clickCount + 1
        ELSE clickCount := 1;
        CASE code OF
          inSysWindow: SystemClick(theEvent,whichWindow);

          inMenuBar:
             BEGIN
              CheckMenus;   { enable or disable items }
              CursorNormal;
              menuResult := MenuSelect(theEvent.where);
          inContent,inGrow:
```

- If it's a mouseDown, was it a double click?
- If it was in the menu bar, make the menus go
- If it was in the window, handle that…

Modern Event Loops

- You won't be managing so much manually (applications don't have to re-implement double clicks or menu bars anymore)
- Swift (at right) uses event loops
- As do every other modern interactive framework



Foundation / RunLoop

Class

RunLoop

The programmatic interface to objects that manage input sources.

iOS 2.0+ | iPadOS 2.0+ | Mac Catalyst 13.0+ | macOS 10.0+ | tvOS 9.0+ | visionOS 1.0+ | watchOS 2.0+

```
class RunLoop
```

Mentioned in

Processing URL session data task results with Combine

Overview

A `RunLoop` object processes input for sources, such as mouse and keyboard events from the window system and `Port` objects. A `RunLoop` object also processes `Timer` events.

Your application neither creates nor explicitly manages `RunLoop` objects. The system creates a `RunLoop` object as needed for each `Thread` object, including the application's main thread. If you need to access the current thread's run loop, use the class method `current`.

Note that from the perspective of `RunLoop`, `Timer` objects aren't "input"—they're a special type, and they don't cause the run loop to return when they fire.

Programming With Events

- You will be building an interactive application, so you will need to handle events!
- Usually you won't see the event loop directly
  - Instead, you implement event handlers
  - To receive events, register those handlers with a dispatcher.

Next: a Stepper

- Goal: every time the user clicks a button, show progress.
- Start at 0%. Ten clicks represents 100%.
- As we make progress, *be critical* of the design.
    - We're going to make a bit of a mess.
    - We'll clean this up next time.

Our Mess

- We have two things all mixed up:
  - View logic (how to draw the bar)
  - Behavior (interaction: what happens when)
- At least we have the model separated.
- Next time: Model-View-Controller