15 & 16: High-level and detailed design (FSMs)
Left side of V model

- **Product requirements**: What the product does from the customer POV
- **Software requirements**: What the product does from the SW POV (high-level, not the “how”)
- **High level/architecture design**: What modules there are in the system, which module performs which function, how modules communicate
- **Low level/module design**: Flowcharts, statecharts/finite state machines, algorithms...
Product requirements

Our electric height-adjustable table allows you to easily and effortlessly change from sitting to standing positions throughout your day. Raising and lowering the table is simple, using its ultra quiet, feature-rich electric mechanism. It's an essential tool to any modern workspace.

Details:

- Changing your posture often keeps you more engaged and more comfortable
- Meetings are significantly shorter when standing vs. sitting
- Height-adjustable tables are essential to modern workspaces and prized by office workers everywhere
- Push-button activation with height display readout
- 3 memory positions
Software requirements

Written with specific wording and format

“Shall” - the software **must** do this to meet requirements

“Should” - the software has this goal

Labeled or numbered (RS-1, RS-2, RS-2.a…)

Precise and measurable

  Quantitative over qualitative

  Can be tested

*What* the software does, not *how*
Adjustable height desk inputs

Current height*

Buttons: 1, 2, 3, up, down, M
Adjustable height desk outputs

Motor command (stopped, up, down)

Display
Adjustable height desk requirements

R1: If the desk is not at its maximum height, and the up button is held, the motor shall be commanded UP

R2: If the M button is pressed and released, and one of the numbered buttons [1, 2, 3] is pressed and released within 10 seconds, then the current height shall be stored as a preset for the corresponding numbered button

R3: If one of the numbered buttons [1, 2, 3] is held, the motor should be commanded such that the desk height moves to the corresponding preset height
Come up with additional requirement(s) that refine the preset behavior.

R3: If one of the numbered buttons [1, 2, 3] is held, the motor should be commanded such that the desk height moves to the corresponding preset height.
Refined requirements

R3: If one of the numbered buttons [1, 2, 3] is held, the motor should be commanded such that the desk height moves to the corresponding preset height

R3-A: If the corresponding preset is not stored, the motor shall be commanded STOPPED

R3-B: If the desk height is at the preset height, the motor shall be commanded STOPPED

R3-C: If the desk height is higher/lower than the preset height, the motor shall be correspondingly commanded DOWN/UP as long as the numbered button is held and the desk has not yet reached the preset height.
High-level/architecture design

How components fit together and what the interfaces are:

- Boxes-and-arrows diagram: **boxes** are components, **arrows** are interfaces.

General rule: should fit on one page

- Details of components are left to detailed design.
Motor controller*—Motor command (up, down, or stopped)

Microcontroller—Current height of desk

LED Display—Height to display

Button array—Button [1, 2, 3, M, Up, Down] pressed or held down

Boxes-and-arrows for adjustable height desk

*Here we make the assumption that the motor controller is able to output the current height of the desk (for example, based on initial calibration and on how long the motor is commanded on). This may not actually be how the controller receives the height on the product, but it's an assumption of how there might be bidirectional communication between two components in a product architecture.
Sequence diagrams

Shows interaction between components
Columns: components
Arrows between columns: data sent across interfaces
Temporally arranged (lower is later)
Usually one for each customer scenario
Scenario is variant of a use case
Scenario: user wants to raise desk, presses up button and desk rises

Button array
- button UP pressed
  - get current height
  - receive current height
  - display current height
  - command UP
  - command STOPPED

Microcontroller
- button UP released
  - get current height
  - receive current height
  - display current height
  - command UP
  - command STOPPED

LED Display

Motor Controller
Scenario: store current height as preset 2

- Button array
  - M pressed, M released
  - 2 pressed
  - Start 10s timer
  - Store current height in preset 2

- Microcontroller
  - Get current height
  - Current height
  - Display current height

- LED Display
  - Display nothing, display current height, display nothing

- Motor Controller
  - Display nothing, display current height, display nothing

Software requirements

High level/architecture design

Low level/module design

Product requirements
Alternative scenario for same use case

Button array

Microcontroller

LED Display

Motor Controller

M pressed, M released

Get current height

Current height

Display current height

Start 10s timer

10s timer finishes

Display nothing

Product requirements

Software requirements

High level/architecture design

Low level/module design
Finite state machines

Low-level design for a module

Shows the change in state of a module

Contrast with flowchart, which just shows flow of computation

At basic level, composed of:

States (one state is initial state)

Guards (predicates on inputs)

Actions (setting outputs)
Variants

Multiple ways to define statecharts/FSMs

Mealy vs. Moore, deterministic vs non-deterministic, etc

Extended FSMs: state variables (variables that are not inputs or outputs) can appear in guards and actions

We will use **deterministic, extended** FSMs as defined by Lee/Seshia

Will be useful when we talk about modeling

Translate well to coding
 FSM example: HW problem

Consider a variant of the thermostat of example 3.5. In this variant, there is only one temperature threshold, and to avoid chattering the thermostat simply leaves the heat on or off for at least a fixed amount of time. In the initial state, if the temperature is less than or equal to 20 degrees Celsius, it turns the heater on, and leaves it on for at least 30 seconds. After that, if the temperature is greater than 20 degrees, it turns the heater off and leaves it off for at least 2 minutes. It turns it on again only if the temperature is less than or equal to 20 degrees.

Design an FSM that behaves as described, assuming it reacts exactly once every 30 seconds.
Inputs:
- temp

Outputs:
- heat (ON/OFF); initially OFF

Extended FSM (alternative design)
Inputs:
- Current temp

Outputs:
- Heater (ON/OFF); initially OFF

Variables: counter; initial value doesn't matter
FSM conventions/rules

- Define inputs, outputs, and variables
  - Define initial values for outputs/variables
- Label each state with a number and a short, descriptive name
  - Label the start state
- Guards for transitions out of a state:
  - should be mutually exclusive
  - should only be predicated on inputs and variables
- Outputs on transitions should only set outputs and variables
Adjustable-height desk FSM

Whiteboard
Inputs:
Button (1, 2, 3, UP, DOWN, M) pressed: boolean
Current desk height: fixed-point number

Outputs:
Motor command: (UP, DOWN, STOPPED): enum
LED display: fixed-point number or NONE

Constants:
maxHeight, minHeight (fixed point numbers)

Variables (for extended FSM):
Presets 1, 2, 3 (fixed-point numbers) or NONE

Initial values:
Motor command: STOPPED
LED display: NONE
Presets 1, 2, 3: NONE

(note: this is just a subset of the entire FSM that we would draw. We also need to enforce that no other buttons are being pressed in our guards, as we talked about in class.)
typedef enum { STOPPED = 1, MOVE_UP = 2 } state;

state updateFSM(state currentState, bool buttonUPpressed, ...) {
    state nextState = currentState; // stay in same state by default
    switch(currentState) {
    case STOPPED:
        if (buttonUPpressed && (deskHeight != maxHeight)) {
            nextState = MOVE_UP;
            setMotorControl(UP);
        }
        break;
    case MOVE_UP:
        if (!buttonUPpressed || (deskHeight == maxHeight)) {
            nextState = STOPPED;
            setMotorControl(STOPPED);
        }
        break;
    default:
        error("invalid state!");  
    }
    return nextState;
}
When should `updateFSM` be called?

```cpp
state updateFSM(state currentState, bool buttonUPpressed, ...)
```
Time-triggered vs. event-triggered design

Time-triggered:
computation to (potentially) change state happens every x ms, regardless if inputs have changed

Event-triggered:
computation to (potentially) change state happens when an input changes

Timer interrupt OR schedule in loop
e.g.

```c
void loop() {
    static state S = STOPPED;
    state = updateFSM(...);
    delay(100);
}
```

Call `updateFSM` in every interrupt/task that polls input
Pros/cons of time- vs. event-triggered design?
How do we know that our design has met our requirements?
Traceability

Ensures that all requirements have been implemented and tested

Often done using a **traceability matrix**

Example: each column is a requirement; each row is a transition

“x” in a cell if the transition helps meet the requirement

If a column has no x’s, means requirement isn’t being met

If a row has no x’s, means transition is unnecessary (or requirement is missing/wasn’t stated!)
Example: requirements to FSM traceability

R1: If the desk is not at its maximum height, and the up button is held, the motor shall be commanded UP

R2: If the M button is pressed and released, and one of the numbered buttons [1, 2, 3] is pressed and released within 10 seconds, then the current height shall be stored as a preset for the corresponding numbered button

\[
\neg \text{buttonUPpressed} \lor (\text{deskHeight} = \text{maxHeight}) / \text{motorControl} := \text{STOPPED}
\]

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T 1-2</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T 2-1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{buttonUPpressed} \land \neg (\text{deskHeight} = \text{maxHeight}) / \text{motorControl} := \text{UP}
\]