Modeling UI with Statecharts

Kelli Rockwell

We as developers and designers are really good at making the impossible, possible.

Sometimes, we're so good at it, we do it unintentionally!

mysutro.com

SUTRO

Home | How it Works | Why Sutro | About | FAQ | Partners | Try Now

How It Works

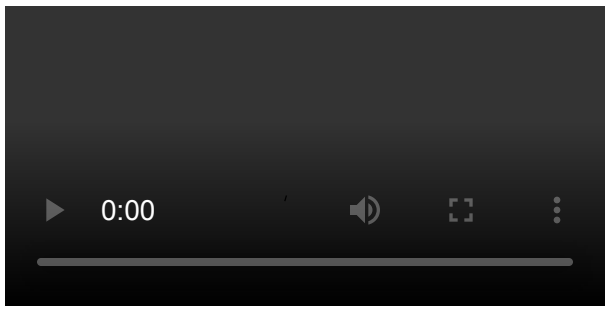
The Sutro Smart System is the Simple, Safe, and Seamless way to ensure that your pool (or spa) is ready when you are. Our patented smart water monitoring technology takes the hassle out of pool or spa water maintenance.

With Sutro, you can measure, monitor, and get recommendations on treatment options without having to leave the comfort of your home. With Sutro, you get what you need, when you need it, from partners that you trust.

Sutro Smart Monitor

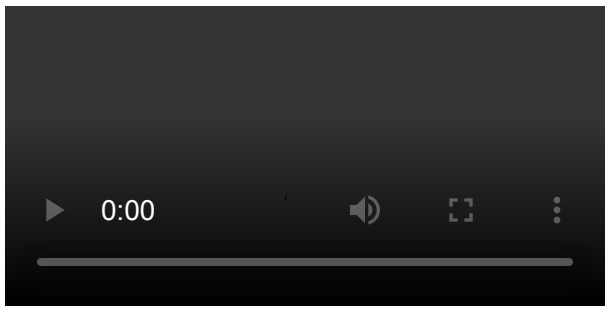
Your Waters Condition

Trusted Recommendations



*Dramatization, not actual user recording

...



*Dramatization, not actual user recording

These are both "impossible states."
So how did the user end up here?

How did the user end up here?

- We changed the application logic and the UI got out of sync

How did the user end up here?

- We changed the application logic and the UI got out of sync
- We tried to take a shortcut

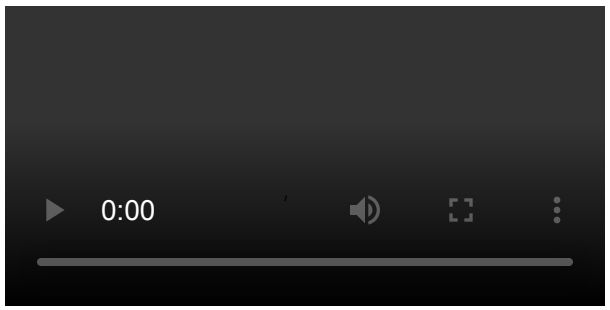
Default case is supposedly impossible to reach:

```
switch(calibration.step) {
  case "PRECHECK1":
    return <PreCheck1 />;
  case "PRECHECK2":
    return <PreCheck2Confirmation />;
  // This screen shouldn't be visible on other steps
  default:
    return <Text>{"Calibration in progress"}</Text>;
}

```

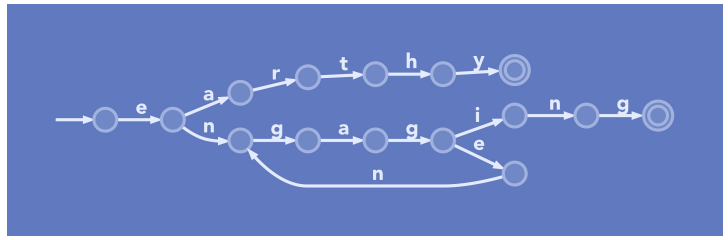
How did the user end up here?

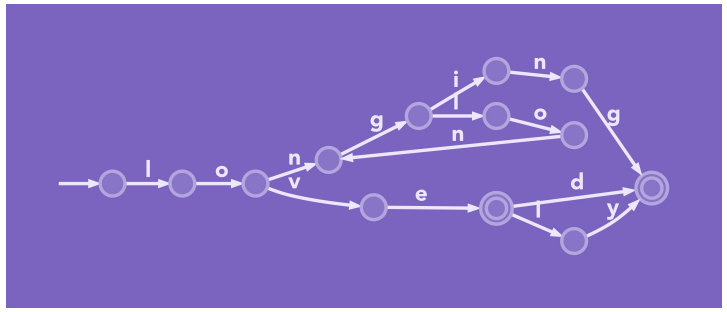
- We changed the application logic and the UI got out of sync
- We tried to take a shortcut
- We forgot to reset a condition
- We missed an edge case
- We didn't handle an API response properly
- ...

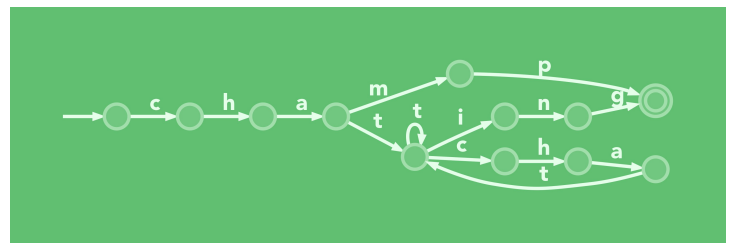
We need a better model for our state.


0:00

Finite State Machines!







We're already used to thinking in state machines.

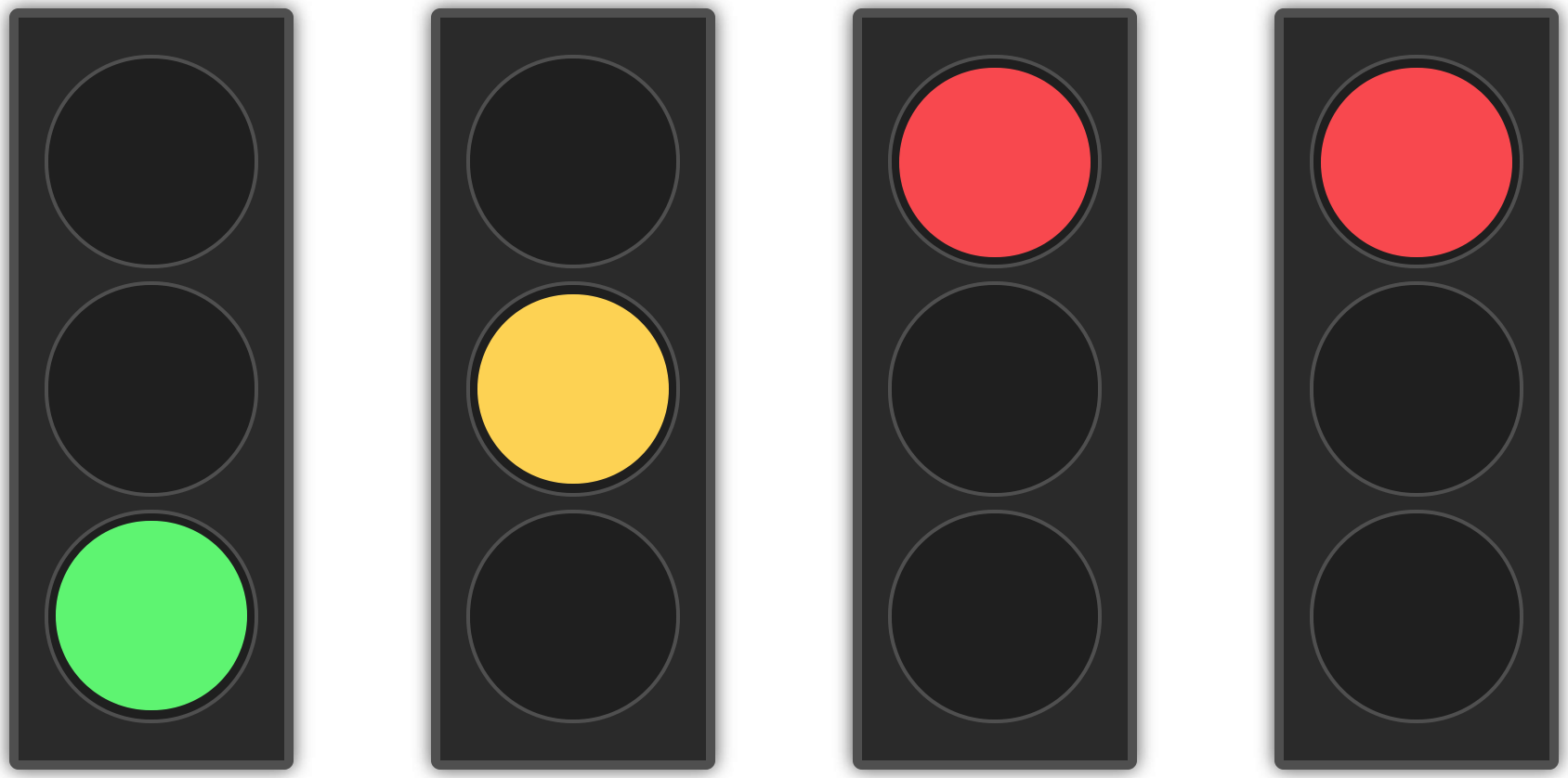
Next

First rule of Finite State Machines:

You can only be in one state at any given time.

Second rule of Finite State Machines:

You have a finite number of discrete states.



Third rule of Finite State Machines:

You transition from one state to another based on an event.

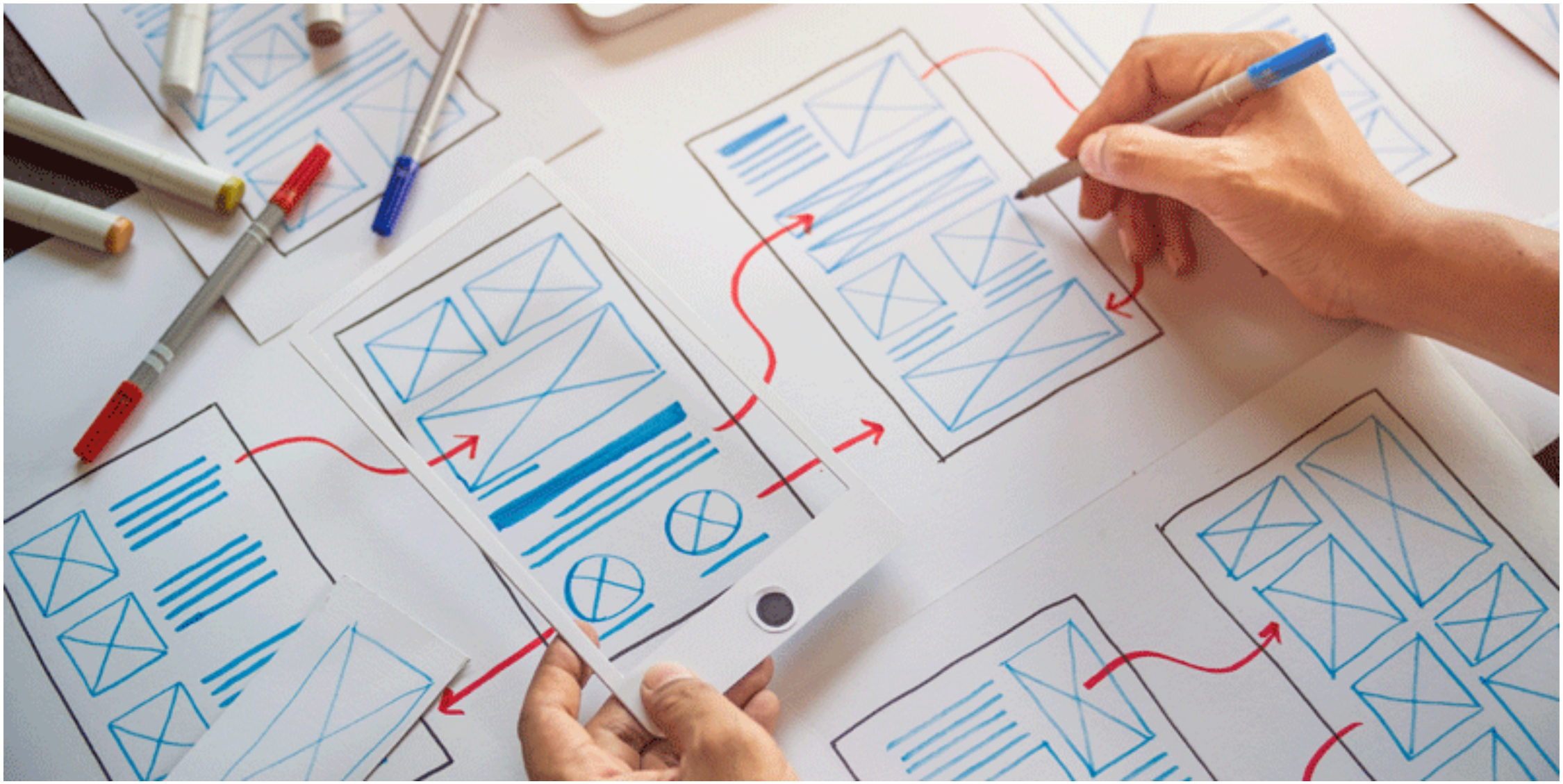
Next

As a driver,

Given, the light is red,

When, 30 seconds have elapsed,

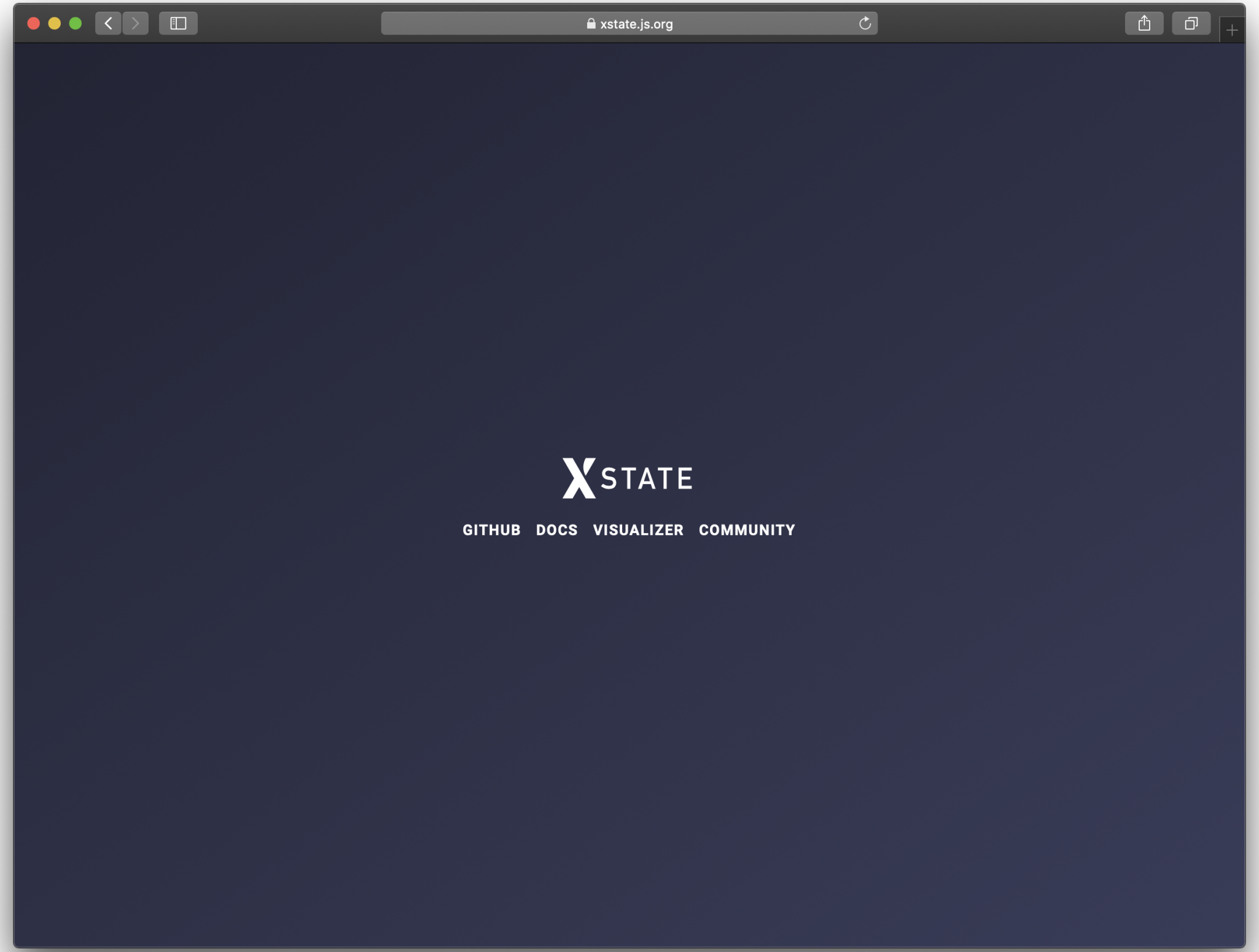
Then, the light turns green.



```
const TRANSITION_SCHEMA = {
  green: { timer: 'yellow' },
  yellow: { timer: 'red' },
  red: { timer: 'green' },
};

class StateMachine {
  constructor(initialState, transitions) {
    this.state = initialState;
    this.transitions = transitions;
  }

  transition(event) {
    this.state = this.transitions[this.state][event];
  }
}

const machine = new StateMachine('green', TRANSITION_SCHEMA);

machine.transition('timer');
console.log(machine.state); // => 'yellow'

machine.transition('timer');
console.log(machine.state); // => 'red'

machine.transition('timer');
console.log(machine.state); // => 'green'

```

The problem here isn't the switch statement, it's that **the state machine is implicit**

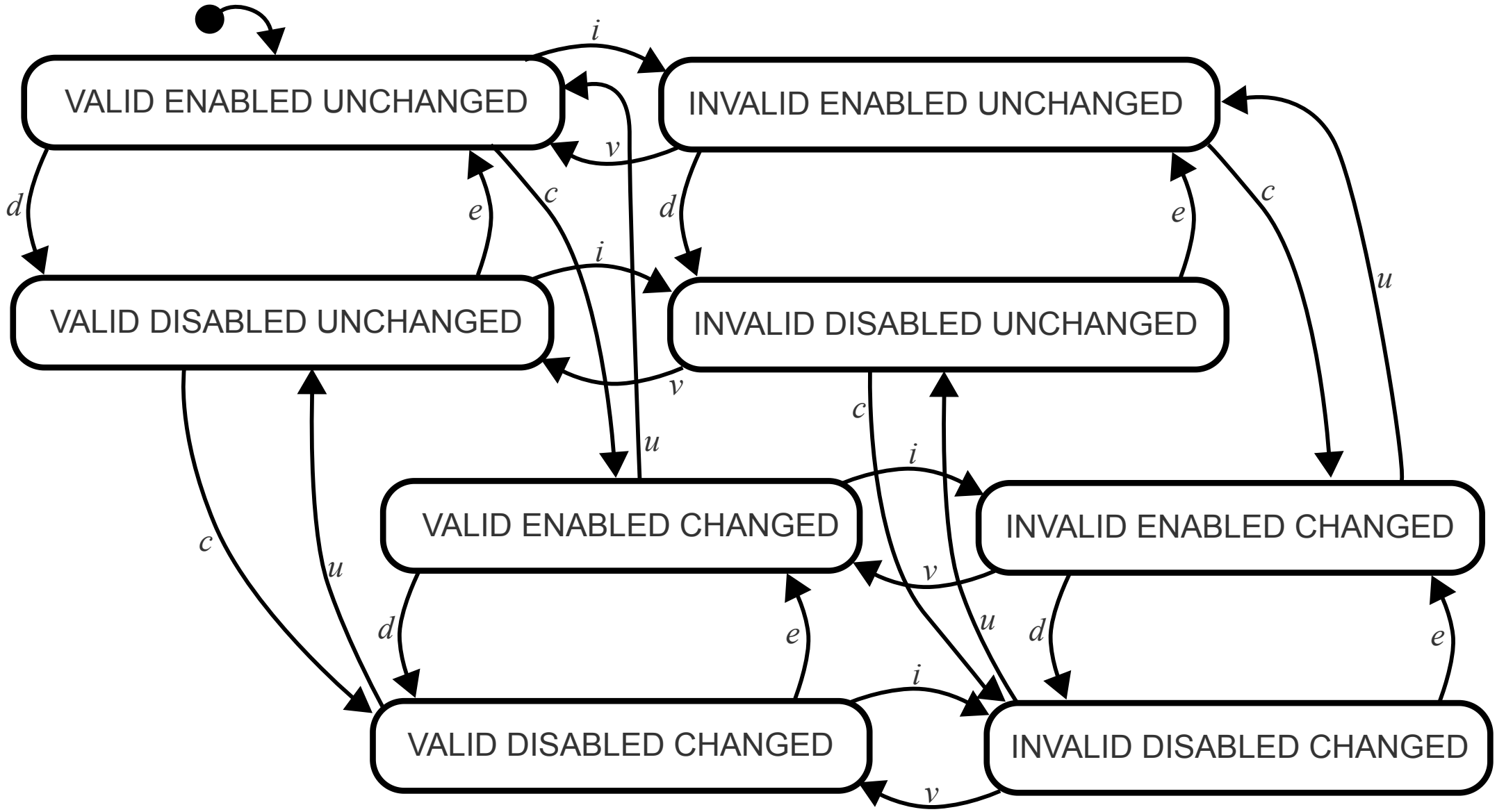
```
switch(calibration.step) {
  case "PRECHECK1":
    return <PreCheck1 />;
  case "PRECHECK2":
    return <PreCheck2Confirmation />;
  // This screen shouldn't be visible on other steps
  default:
    return <Text>{"Calibration in progress"}</Text>;
}

```

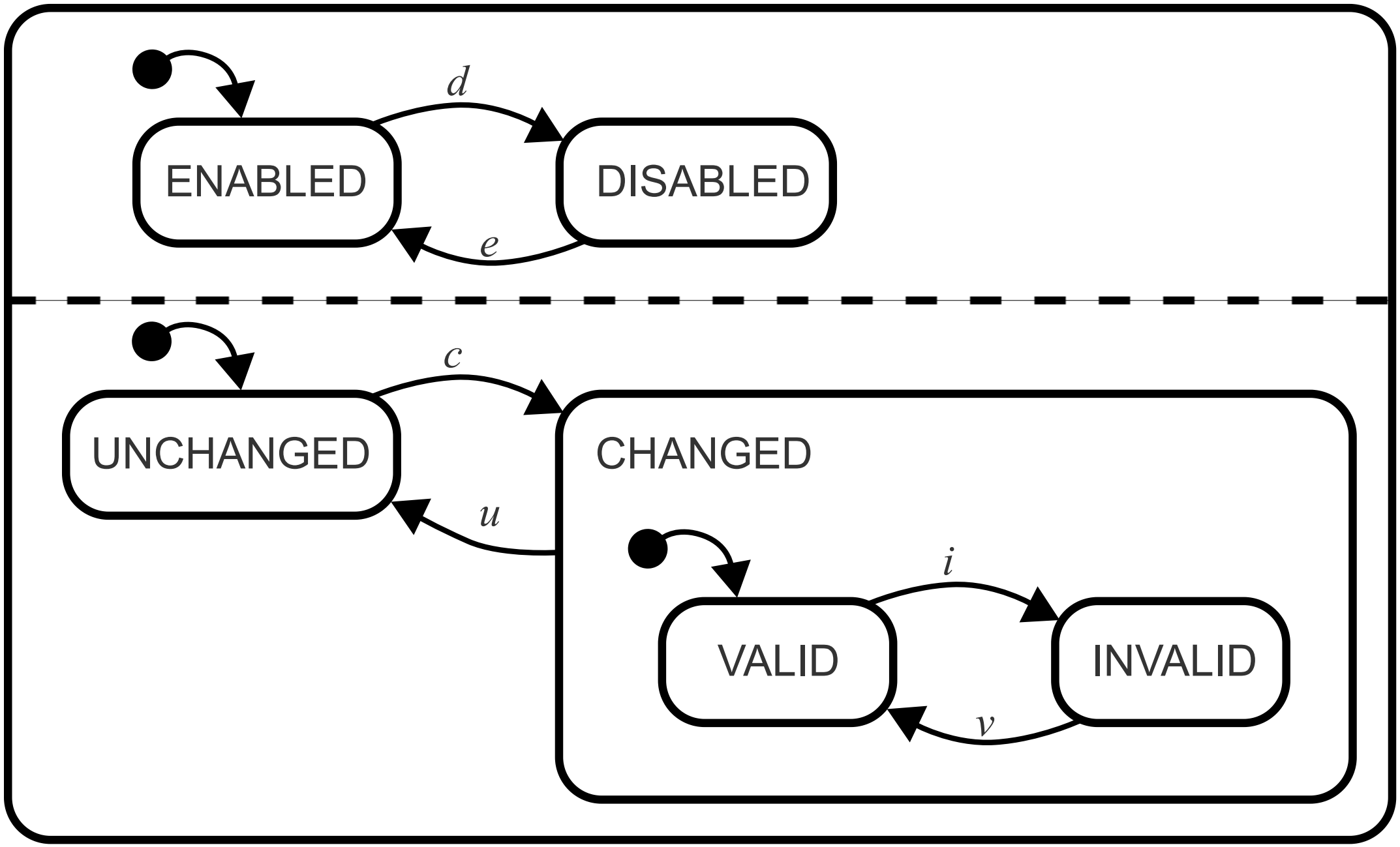

xstate.js.org
XSTATE
GITHUB DOCS VISUALIZER COMMUNITY

```
const trafficLightMachine = Machine({
  id: 'trafficLight',
  initial: 'green',
  states: {
    green: {
      on: { TIME_UP: 'yellow' },
    },
    yellow: {
      on: { TIME_UP: 'red' },
    },
    red: {
      on: { TIME_UP: 'green' },
    },
  },
});

```

```
const expandedTrafficLightMachine = Machine({
  id: 'expandedTrafficLight',
  initial: 'connected',
  states: {
    disconnected: {
      on: { SIGNAL_FOUND: 'connected' },
    },
    connected: {
      initial: 'green',
      on: { SIGNAL_LOST: 'disconnected' },
      states: {
        green: {
          on: { TIME_UP: 'yellow' },
        },
        yellow: {
          on: { TIME_UP: 'red' },
        },
        red: {
          on: { TIME_UP: 'green' },
        },
      },
    },
  },
});

```

Statechart = finite state machine with nested and/or parallel states


VALID ENABLED UNCHANGED
INVALID ENABLED UNCHANGED
VALID DISABLED UNCHANGED
INVALID DISABLED UNCHANGED
VALID ENABLED CHANGED
INVALID ENABLED CHANGED
VALID DISABLED CHANGED
INVALID DISABLED CHANGED
i
v
d
e
c
u


ENABLED
DISABLED
d
e
UNCHANGED
CHANGED
c
u
VALID
INVALID
i
v

```
const expandedTrafficLightMachine = Machine({
  id: 'expandedTrafficLight',
  initial: 'connected',
  states: {
    disconnected: {
      on: { SIGNAL_FOUND: 'connected' },
    },
    connected: {
      initial: 'green',
      on: { SIGNAL_LOST: 'disconnected' },
      states: {
        green: {
          on: { TIME_UP: 'yellow' },
        },
        yellow: {
          on: { TIME_UP: 'red' },
        },
        red: {
          on: { TIME_UP: 'green' },
        },
      },
    },
  },
});

```

```tsx
const TrafficLightWithControl: FC<{}> = () => {
  const [state, send] = useMachine(expandedTrafficLightMachine);

  const [currentColor, setCurrentColor] = useState<StateColor>('green');

  const getColor = (state: ExpandedState): StateColor => {
    switch (true) {
      case state.matches({ connected: 'green' }):
        return 'green';
      case state.matches({ connected: 'yellow' }):
        return 'yellow';
      case state.matches({ connected: 'red' }):
        return 'red';
      case state.matches('disconnected'):
        return 'hazard';
    }
  };

  // When the state changes, set the color of the traffic light to match
  useEffect(() => {
    setCurrentColor(getColor(state));
  }, [state]);

  // Sends a "SIGNAL_FOUND" or "SIGNAL_LOST" event depending on the current state
  const toggleSignal = () =>
    state.matches('disconnected') ? send('SIGNAL_FOUND') : send('SIGNAL_LOST');

  const buttons = (
    <div style={styles.flex}>
      <button onClick={() => send('TIME_UP')}>Next</button>
      <button onClick={toggleSignal} style={styles.invisible} />
    </div>
  );

  return (
    <>
      <TrafficLight currentColor={currentColor} />
      {buttons}
    </>
  );
};

```

Enough with the traffic lights already!

MACHINE-0.TS

```ts
export const calibrationMachine = Machine({
  id: 'machine',
  initial: 'dashboard',
  states: {
    dashboard: {
      on: { START_CALIBRATION: 'precheck1' },
    },
    precheck1: {
      on: { PASS_PRECHECK1: 'precheck2' },
    },
    precheck2: {
      on: { PASS_PRECHECK2: 'prime' },
    },
    prime: {
      on: { PASS_PRIME: 'dashboard' },
    },
  },
});

```

APP-0.TSX

```tsx
const getView = (
  state: StateValue,
  send: (event: CalibrationEvent) => void,
): JSX.Element[] => {
  switch (state) {
    case 'dashboard': {
      return [
        <h2>Dashboard</h2>,
        <button onClick={() => send({ type: 'START_CAL
          Start Calibration
        </button>,
      ];
    }
    case 'precheck1': {
      return [
        <h2>Precheck1</h2>,
        <button onClick={() => send({ type: 'PASS_PREC
          Pass Precheck1
        </button>,
      ];
    }
    case 'precheck2': {
```

Dashboard

Start Calibration

0

MACHINE-1.TS

```ts
/** Builds a polling activity for a given polling func
const generatePollingActivity = (pollingFn: () => void)
  // Start the polling activity
  const interval = setInterval(pollingFn, 1000);

  // Return a function that stops the polling activity
  return () => {
    console.log('stopping poll');
    clearInterval(interval);
  };
};

export const calibrationMachine = Machine(
  {
    id: 'machine',
    initial: 'dashboard',
    states: {
      dashboard: {
        on: {
          START_CALIBRATION: 'precheck1',
        },
      },
```

APP-1.TSX

```tsx
const getView = (
  state: CalibrationState,
  send: (event: CalibrationEvent) => void,
): JSX.Element[] => {
  switch (true) {
    case state.matches('dashboard'): {
      return [
        <h2>Dashboard</h2>,
        <button onClick={() => send({ type: 'START_CAL
          Start Calibration
        </button>,
      ];
    }
    case state.matches('precheck1'): {
      return [
        <h2>Precheck1</h2>,
        <button onClick={() => send({ type: 'PASS_PREC
          Pass Precheck1
        </button>,
        getRetryButton(state, send),
      ];
    }
```

Dashboard

Start Calibration Error

MACHINE-2.TS

```ts
export type CalibrationContextStatus = 'COMPLETE' | '
export type CalibrationContextStep = 'PRECHECK1' | 'P


/** The context (extended state) of the machine */
export type CalibrationContext = {
  needsCalibration: boolean;
  status: CalibrationContextStatus;
  step: CalibrationContextStep;
};


/** Builds a polling activity for a given polling fun
const generatePollingActivity = (pollingFn: () => voi
  // Start the polling activity
  const interval = setInterval(pollingFn, 1000);

  // Return a function that stops the polling activit
  return () => {
    console.log('stopping poll');
    clearInterval(interval);
  };
```

APP-2.TSX

```tsx
const getView = (
  state: CalibrationState,
  send: (event: CalibrationEvent) => void,
): JSX.Element[] => {
  switch (true) {
    case state.matches('dashboard'): {
      const disabled = !state.context.needsCalibratio
      const button =
        state.context.status === 'IN_PROGRESS' ? (
          <button
            disabled={disabled}
            key="resume"
            onClick={() => send({ type: 'RESUME' })}
          >
            Resume
          </button>
        ) : (
          <button
            disabled={disabled}
            key="button"
            onClick={() => send({ type: 'START_CALIBR
          >
```

Dashboard

Start Calibration Error

Initial Context from the "Server"

Status complete

Step done

Needs Calibration? ☐


```
{
  "needsCalibration": false,
  "status": "COMPLETE",
  "step": "DONE"
}
```

MACHINE-3.TS

```ts
export type CalibrationContextStatus = 'COMPLETE' | '
export type CalibrationContextStep = 'PRECHECK1' | 'P


/** The context (extended state) of Precheck1 */
export type Precheck1Context = {
  firmwareUpToDate: 'waiting' | boolean;
  lidClosed: 'waiting' | boolean;
  batteryCharged: 'waiting' | boolean;
  cartridgeInstalled: 'waiting' | boolean;
};

export const INITIAL_PRECHECK1_CONTEXT: Precheck1Cont
  firmwareUpToDate: 'waiting',
  lidClosed: 'waiting',
  batteryCharged: 'waiting',
  cartridgeInstalled: 'waiting',
};


/** The context (extended state) of the machine */
export type CalibrationContext = {
```

APP-3.TSX

```tsx
const getDisplayText = (contextValue: 'waiting' | boo
  switch (contextValue) {
    case 'waiting':
      return '?';
    case true:
      return 'PASS';
    case false:
      return 'FAIL';
  }
};

const Precheck1: FC<Precheck1Context & { send: (event
  firmwareUpToDate,
  lidClosed,
  batteryCharged,
  cartridgeInstalled,
  send,
}) => {
  // Run effect anytime a status updates to check if
  useEffect(() => {
    if (
      isBoolean(firmwareUpToDate) &&
```

Dashboard

Start Calibration

Fail Chance

50

Takeaways

Takeaways

- Keeping UI in sync with application logic is universally hard

Takeaways

- Keeping UI in sync with application logic is universally hard
- Implicit state machines make it harder

Takeaways

- Keeping UI in sync with application logic is universally hard
- Implicit state machines make it harder
- You're probably already thinking in state machines

Takeaways

- Keeping UI in sync with application logic is universally hard
- Implicit state machines make it harder
- You're probably already thinking in state machines
- Statecharts = beefed up state machines

Takeaways

- Keeping UI in sync with application logic is universally hard
- Implicit state machines make it harder
- You're probably already thinking in state machines
- Statecharts = beefed up state machines
- **Modeling UI with statecharts is something you should consider doing :)**

Thank you!