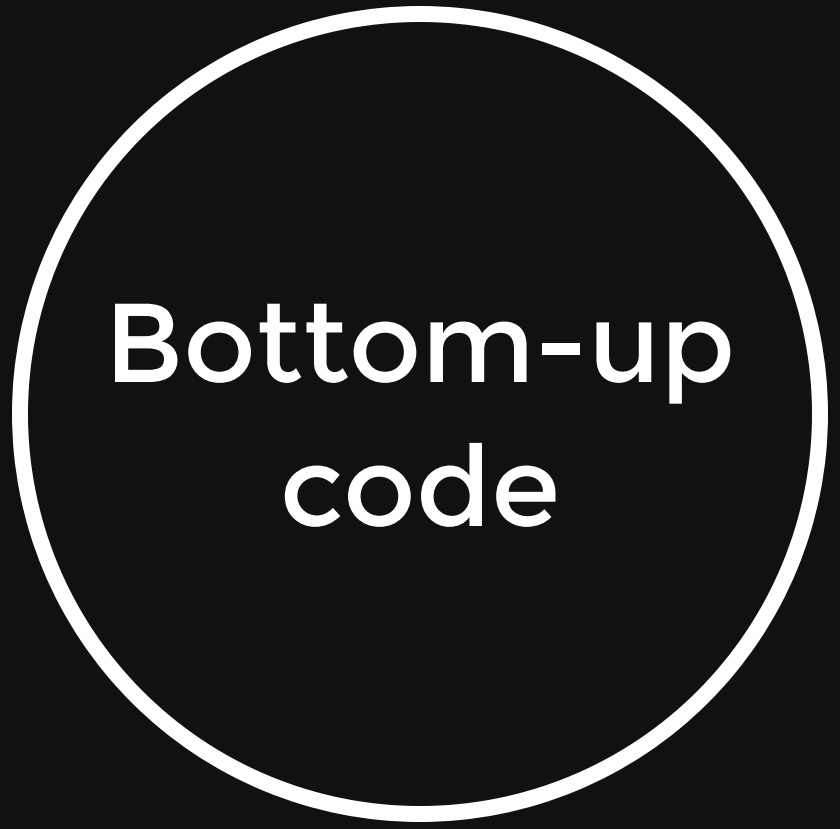
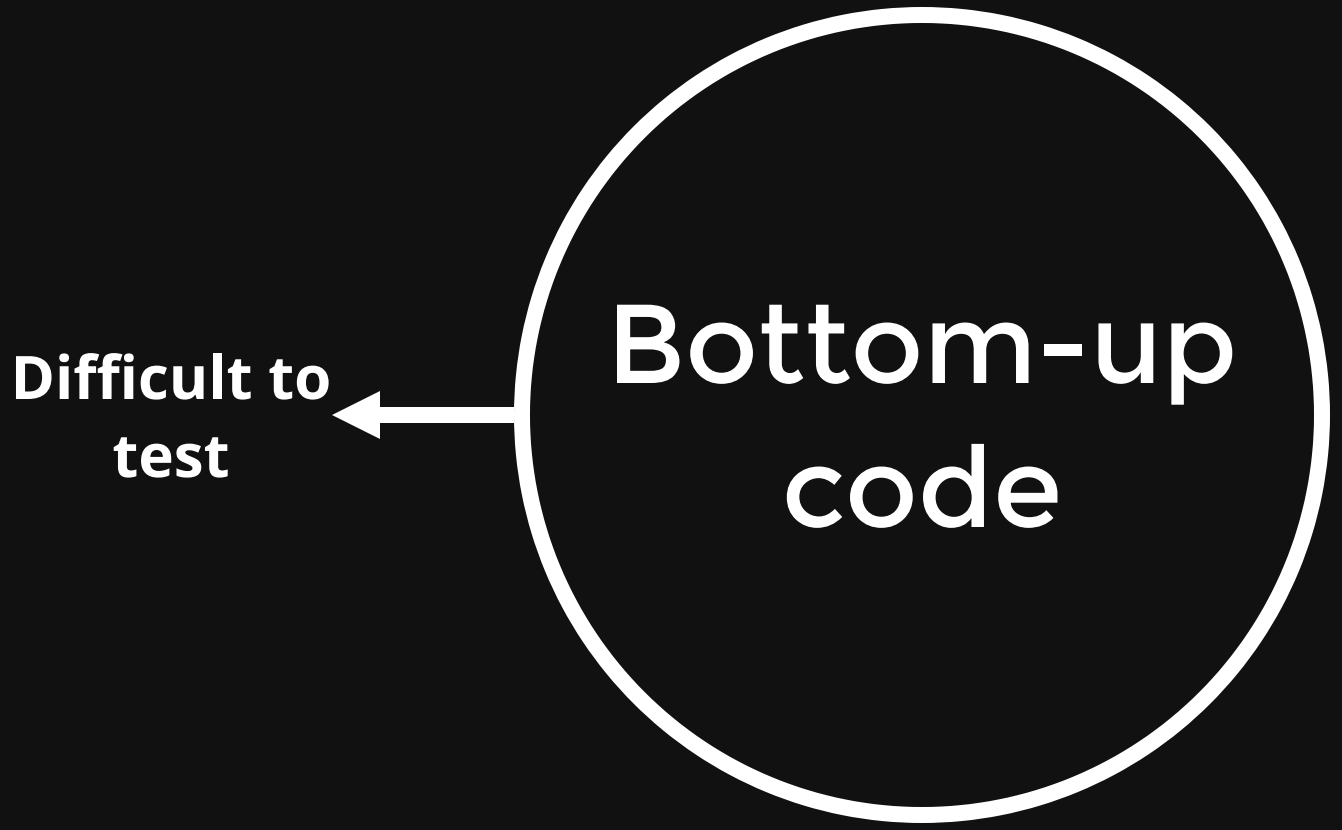
Frontend Masters

JavaScript State Machines & XSTATE

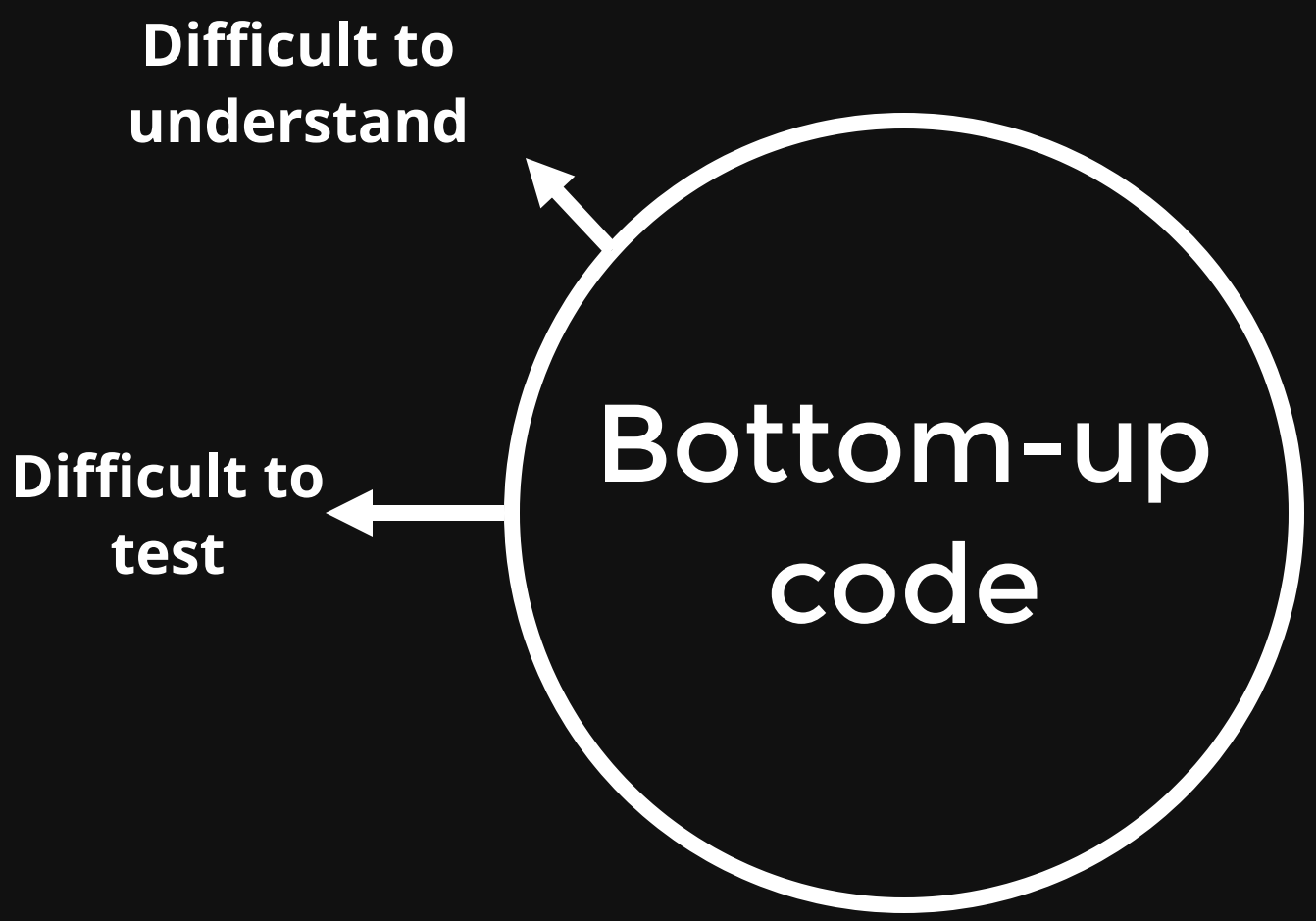
David Khourshid · **@davidkpiano**



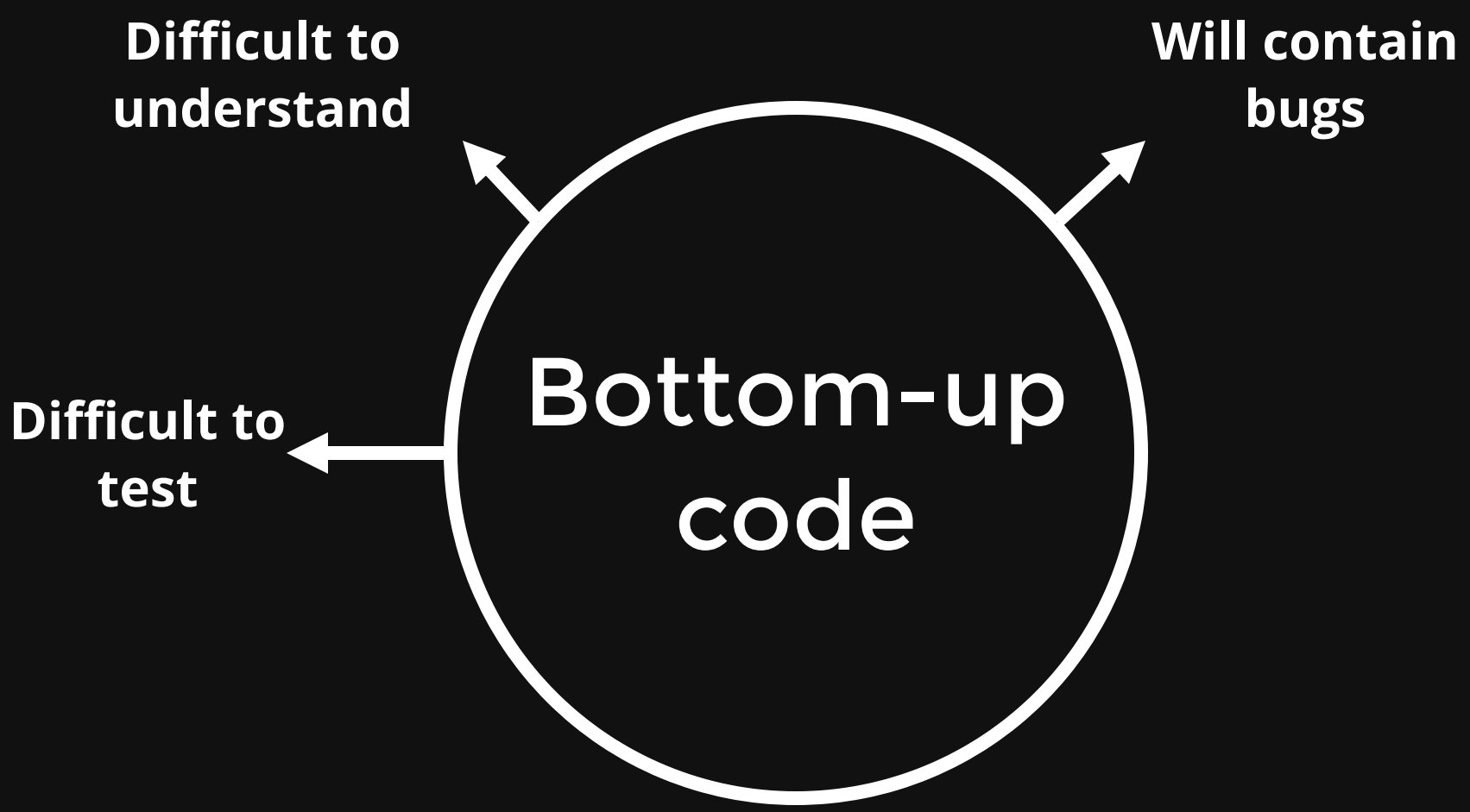
Source: Ian Horrocks, "Constructing the User Interface with Statecharts", ch. 3 pg. 17



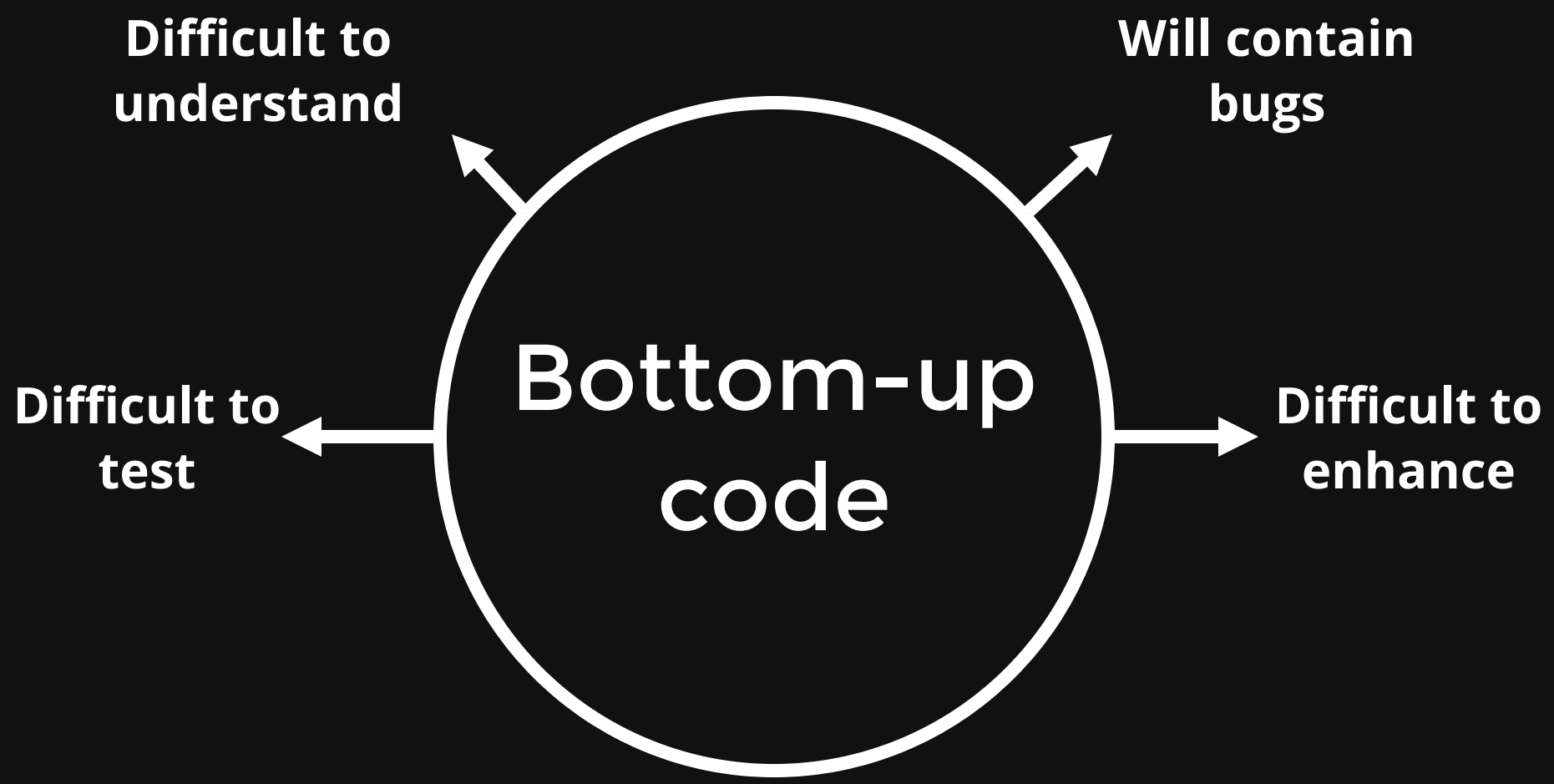
Source: Ian Horrocks, "Constructing the User Interface with Statecharts", ch. 3 pg. 17



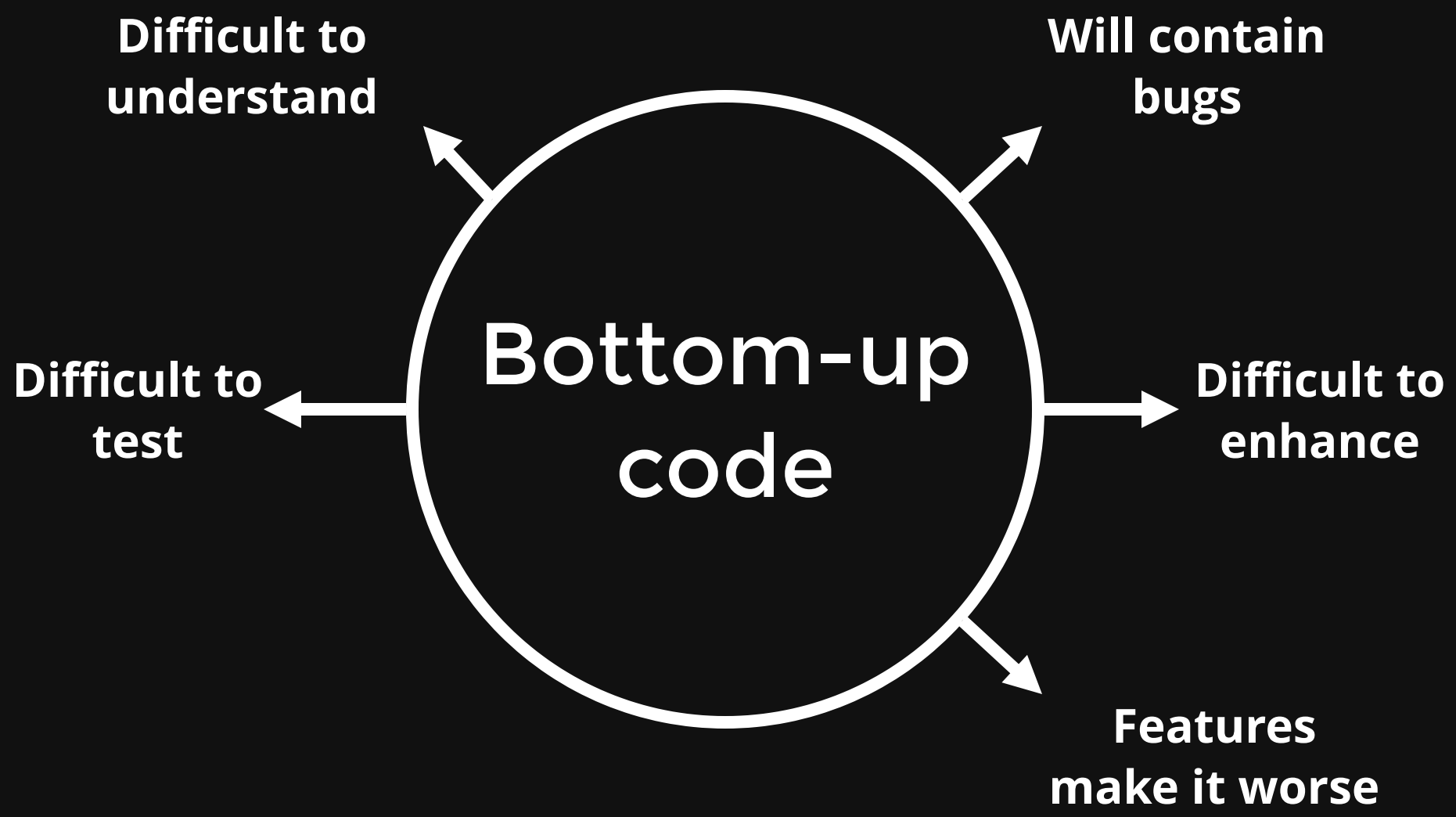
Source: Ian Horrocks, "Constructing the User Interface with Statecharts", ch. 3 pg. 17



Source: Ian Horrocks, "Constructing the User Interface with Statecharts", ch. 3 pg. 17



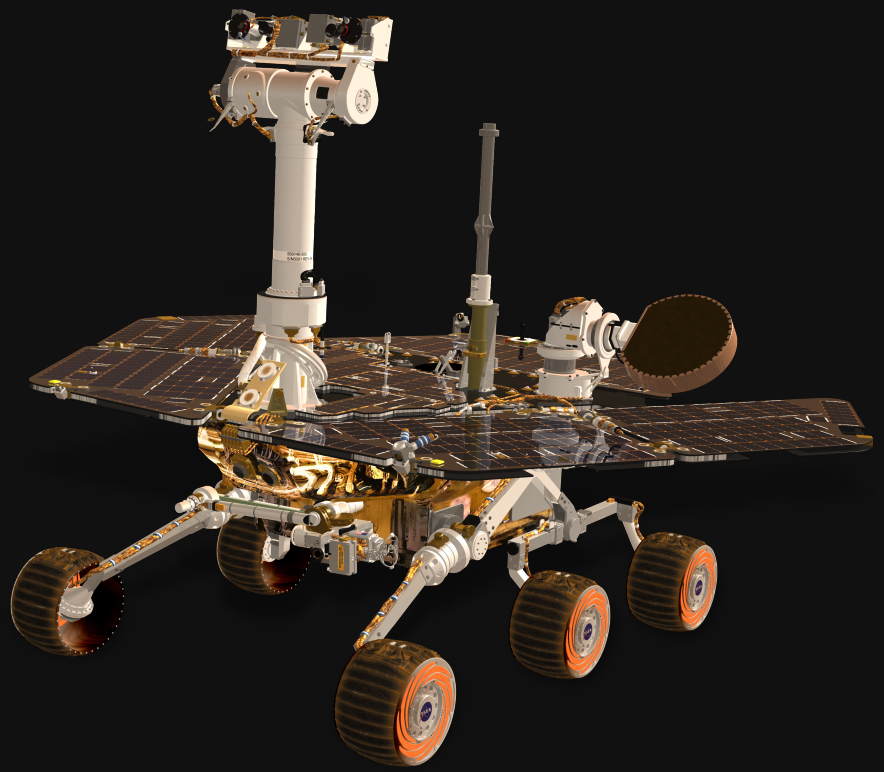
Source: Ian Horrocks, "Constructing the User Interface with Statecharts", ch. 3 pg. 17



Source: Ian Horrocks, "Constructing the User Interface with Statecharts", ch. 3 pg. 17

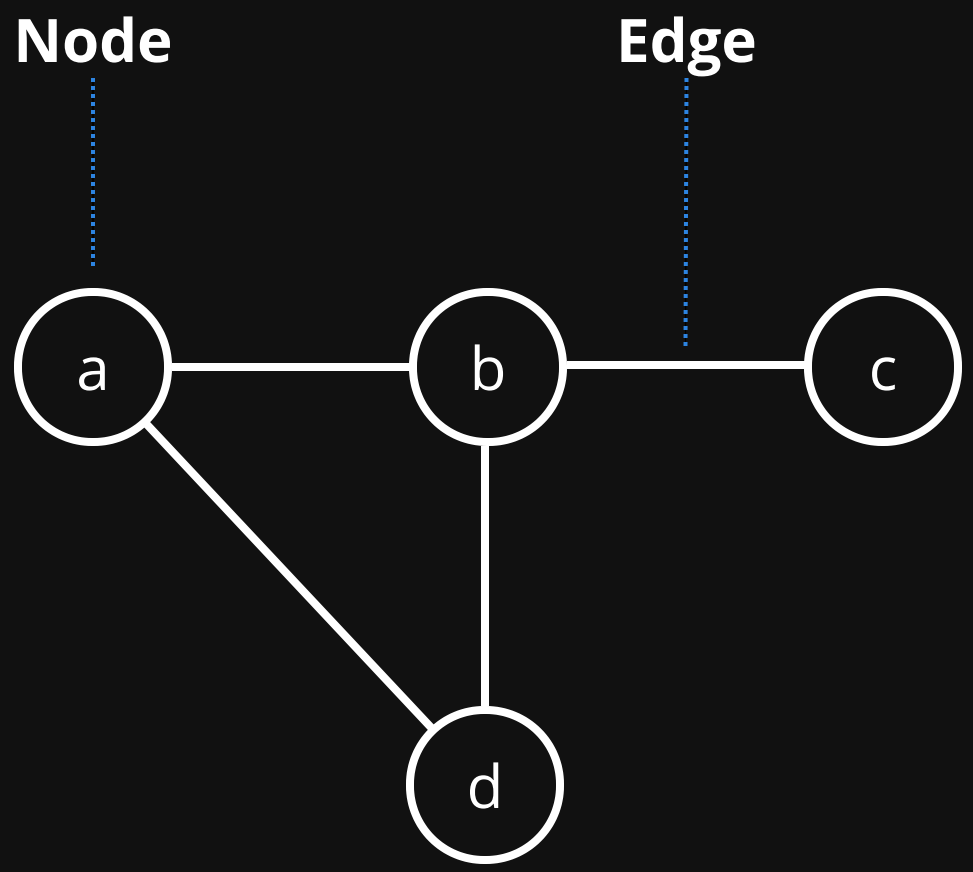
Why use state machines & statecharts?

- Visualized **modeling**
- Precise **diagrams**
- Automatic **code generation**
- Comprehensive **test coverage**
- Accommodation of late-breaking requirements **changes**

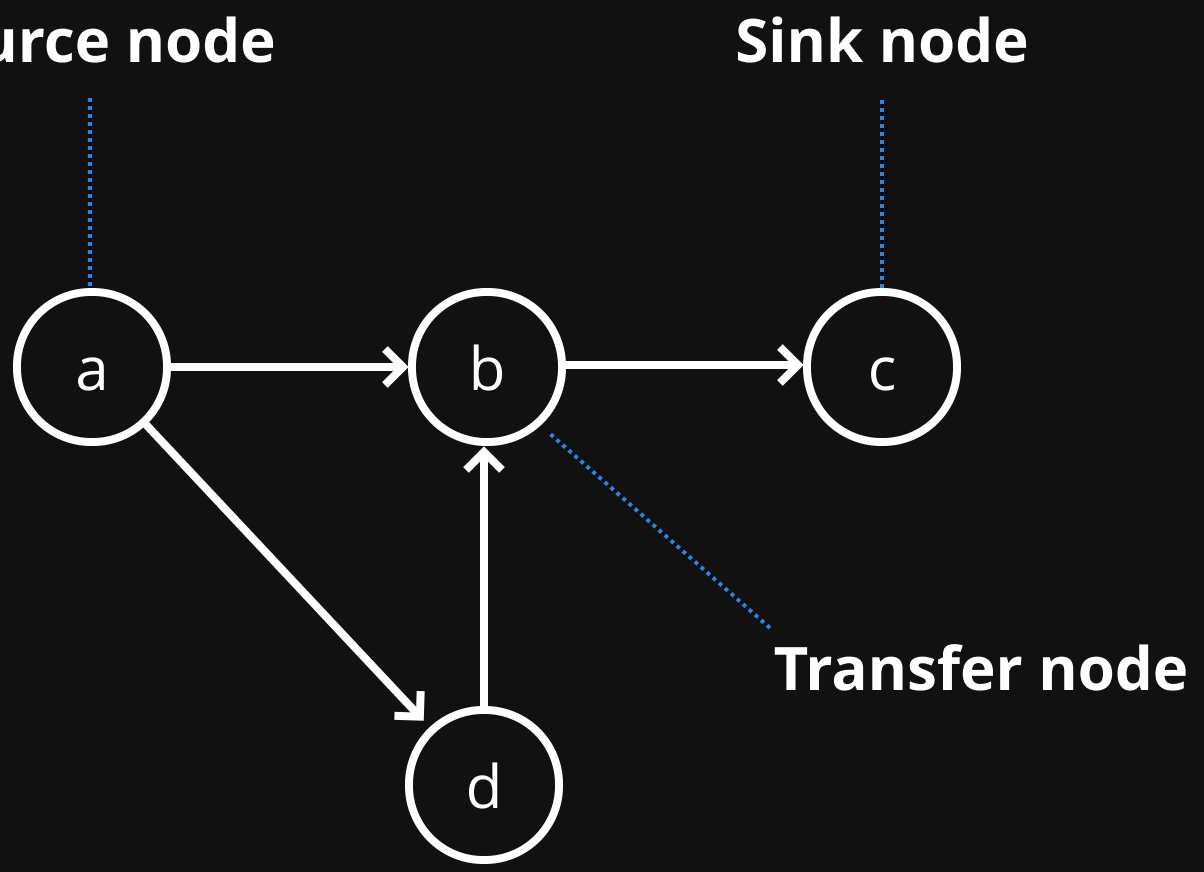


Statechart Autocoding for Curiosity Rover

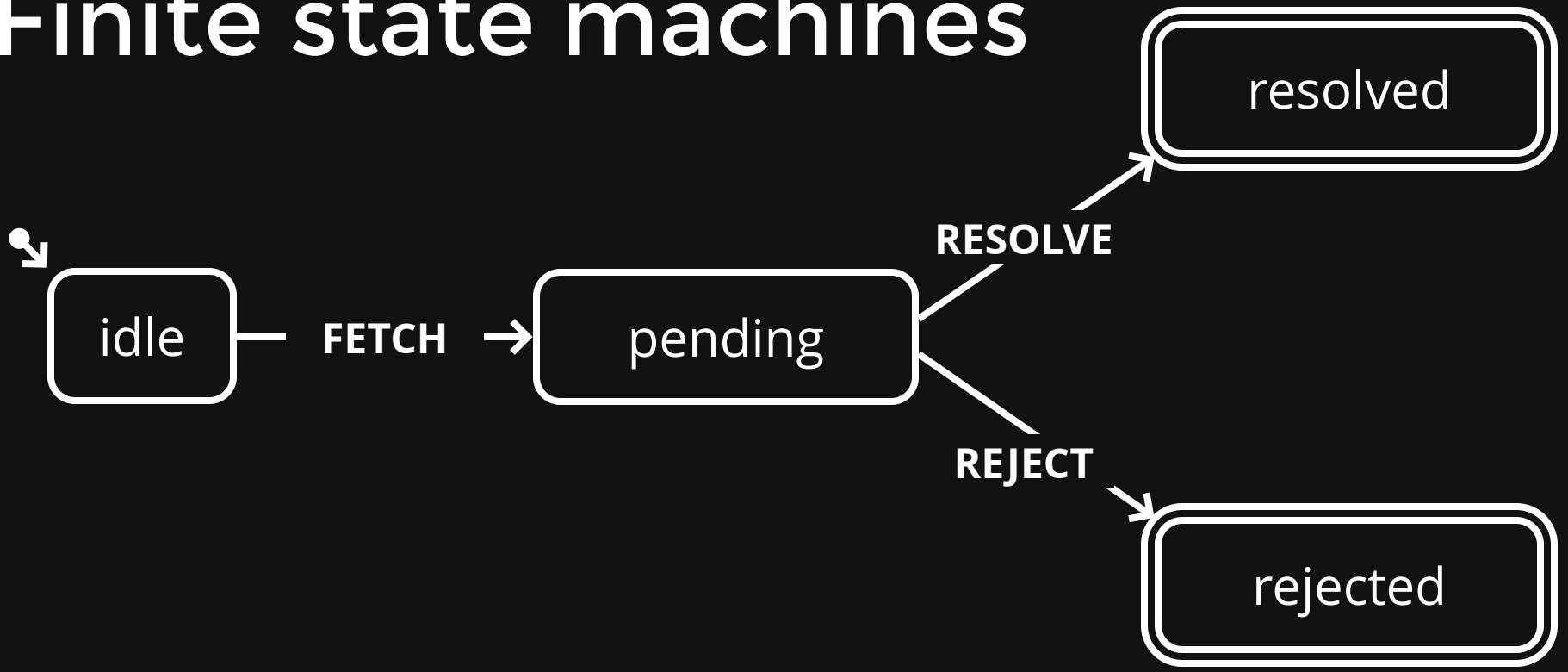
Graphs



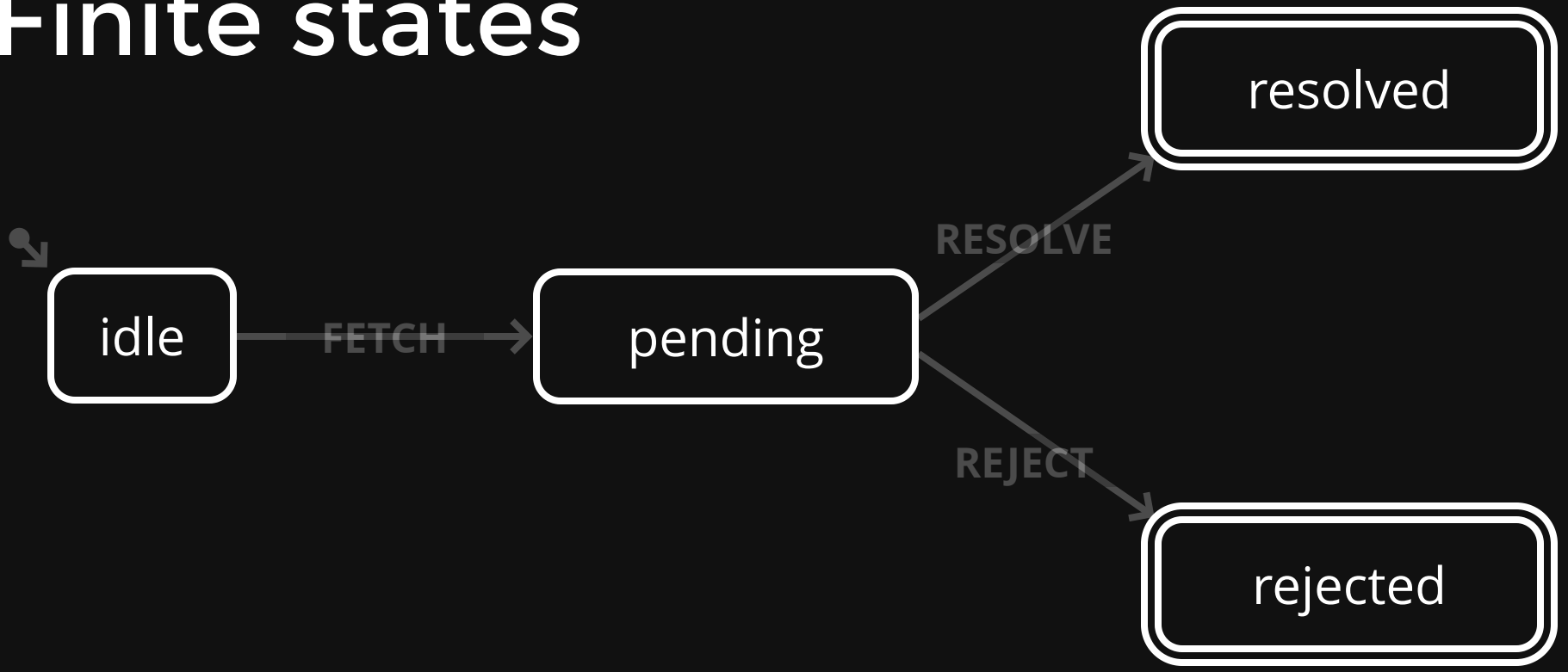
Directed graphs



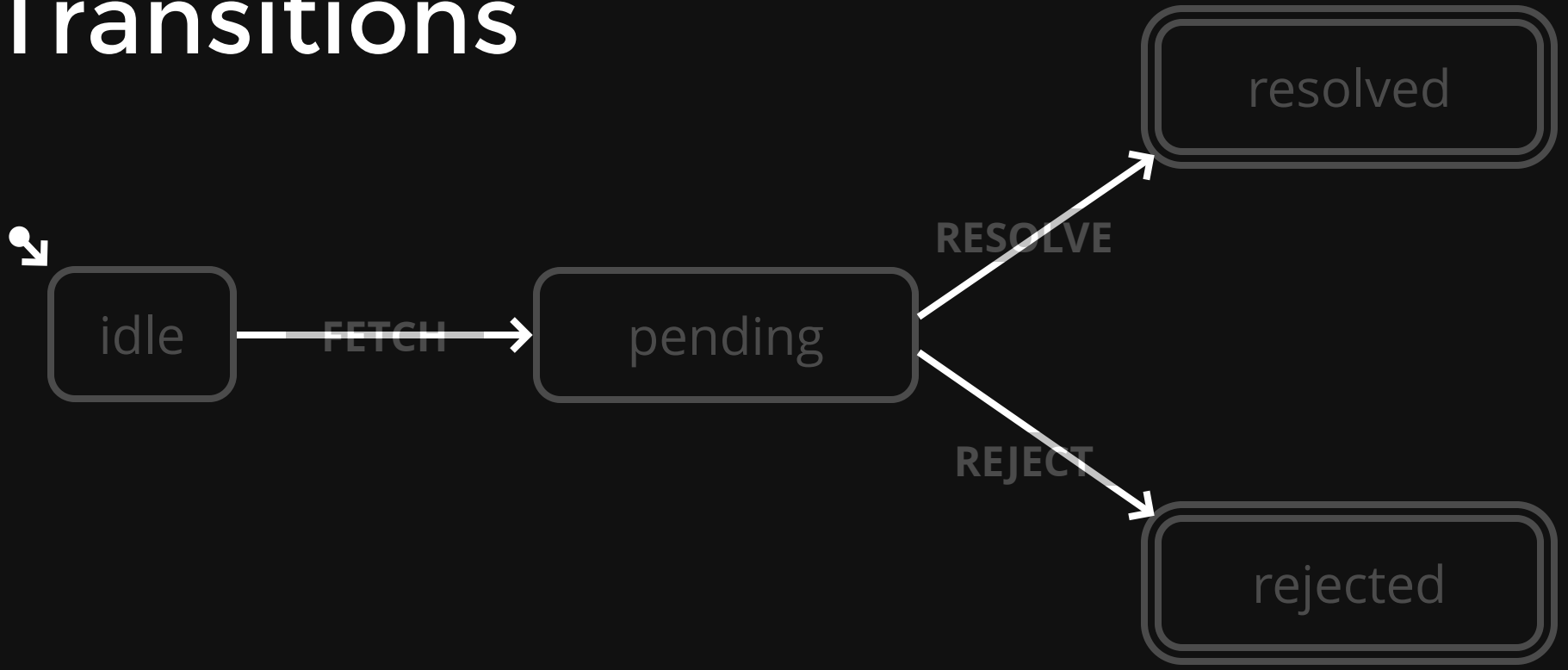
Finite state machines



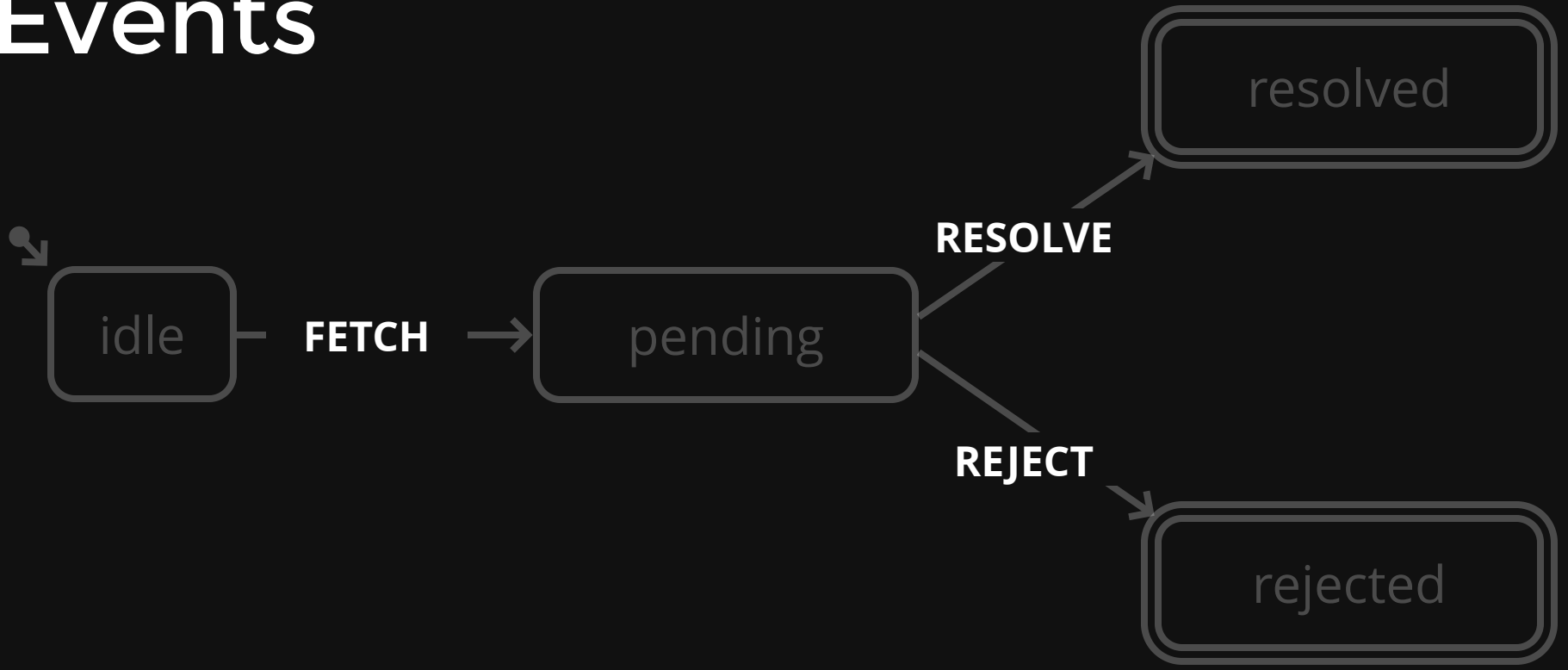
Finite states



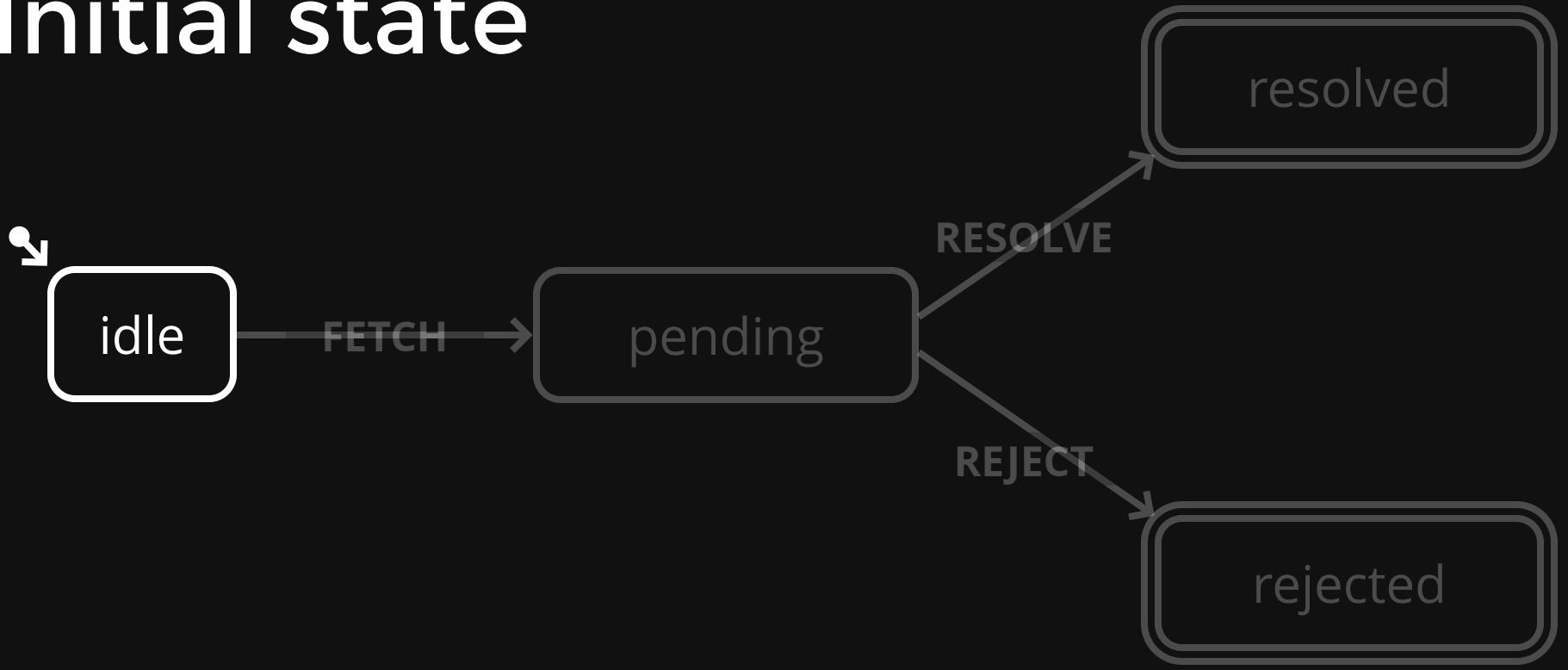
Transitions



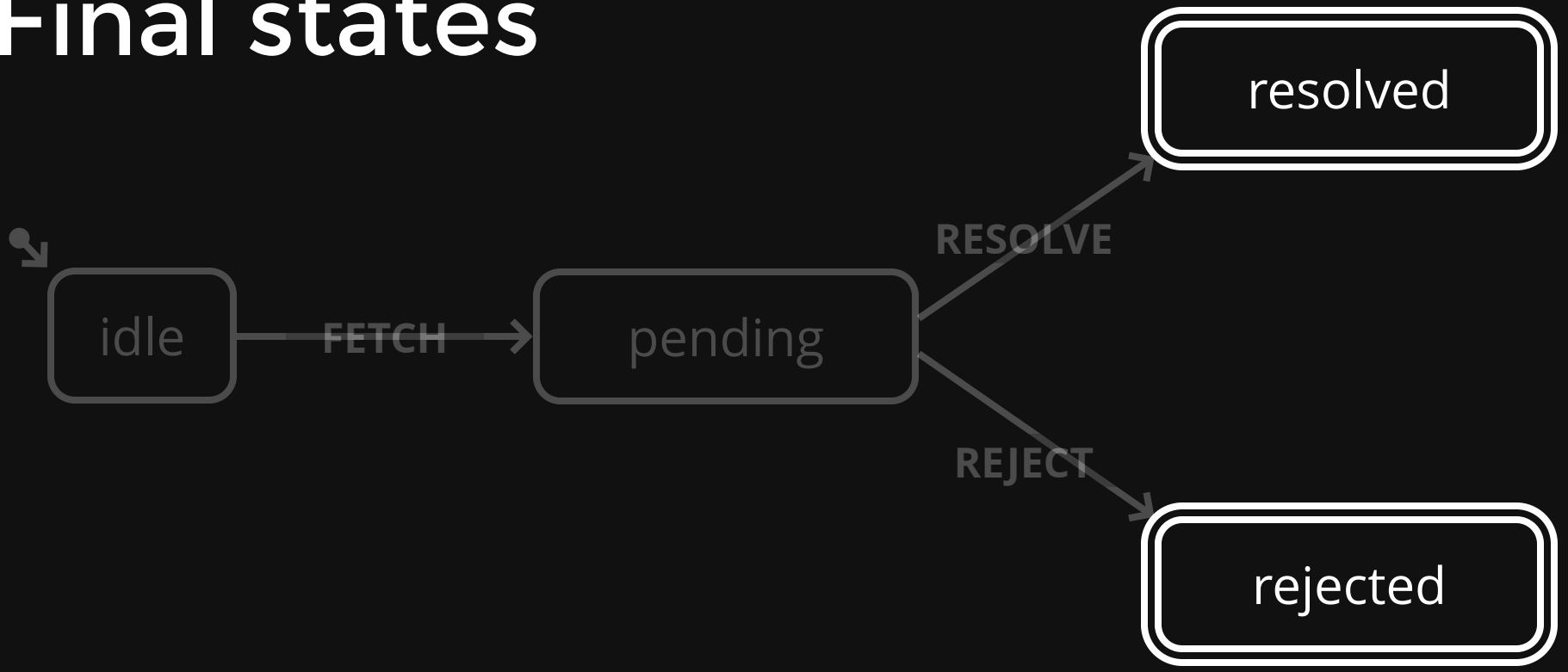
Events



Initial state



Final states



Using switch statements

```
function transition(state, event) {
  switch (state) {
    case 'idle':
      switch (event) {
        case 'FETCH':
          return 'pending';
        default:
          return state;
      }
    case 'pending':
      switch (event) {
        case 'RESOLVE':
          return 'resolved';
        case 'REJECT':
          return 'rejected';
        default:
          return state;
      }
    case 'resolved':
    case 'rejected':
    default:
      return state;
  }
}
```

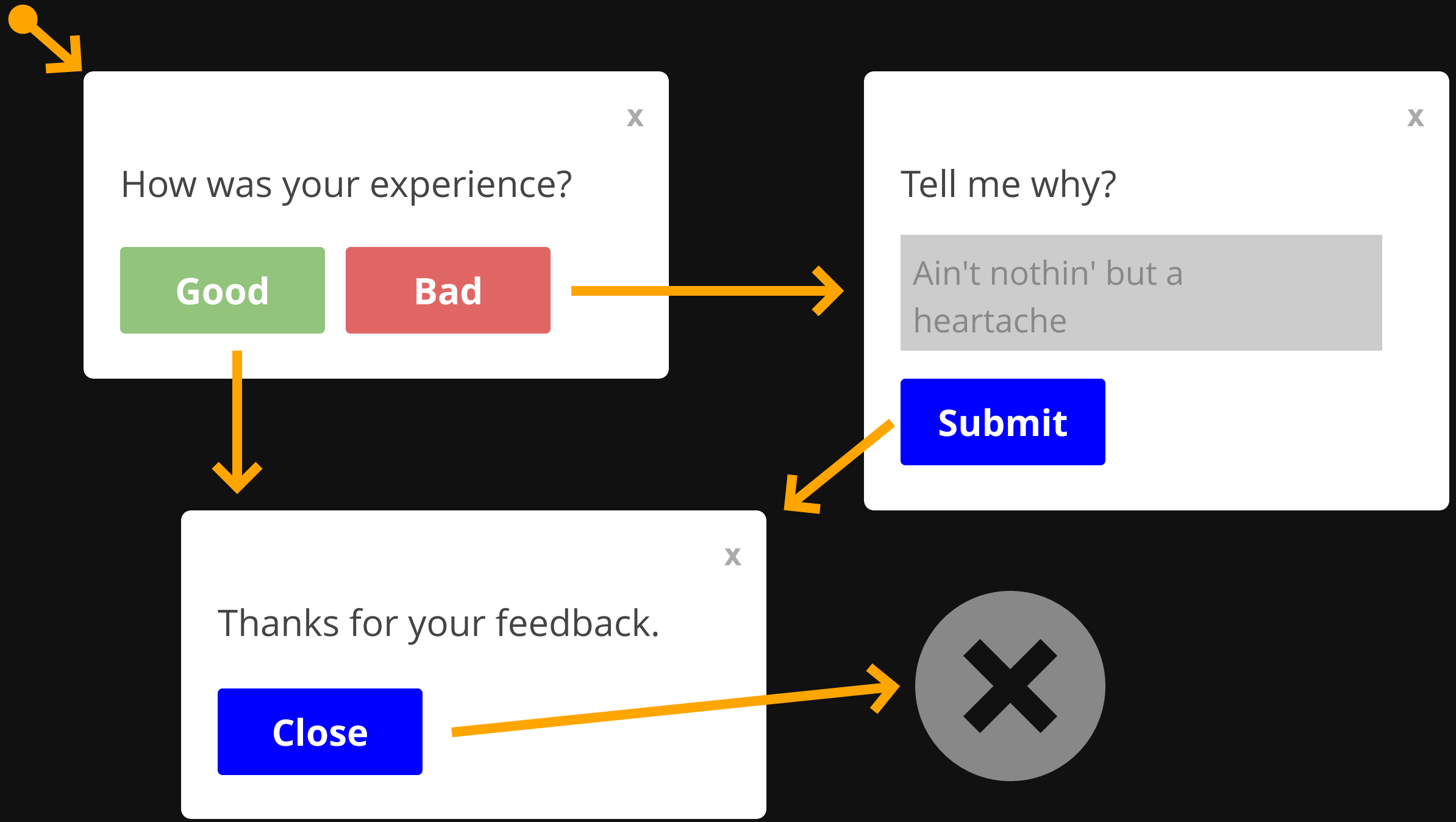
Using objects

```
const machine = {
  initial: 'idle',
  states: {
    idle: {
      on: {
        FETCH: 'pending'
      }
    },
    pending: {
      on: {
        RESOLVE: 'resolved',
        REJECT: 'rejected'
      }
    },
    resolved: {},
    rejected: {}
  }
}

function transition(state, event) {
  return machine
    .states[state]?
    .on?.[event]
    || state;
}
```

Interpreting state machines

```
// Keep track of the current state
let currentState = machine.initial;

// Receive events
function send(event) {

  // Determine the next state
  const nextState =
    transition(currentState, event);

  // Update the current state
  currentState = nextState;
}

// Send some event
send("CLICK");
```

Exercise 01


x
How was your experience?
Good
Bad
x
Tell me why?
Ain't nothin' but a heartache
Submit
x
Thanks for your feedback.
Close

Add states

```
// Setup the machine
const feedbackMachine = {
  initial: 'question',
  states: {
    question: {
      // ...
    },
    form: {
      // ...
    },
    thanks: {
      // ...
    },
    closed: {
      // ...
    }
  }
}
```

Add events

```
// Setup the machine
const feedbackMachine = {
  initial: 'question',
  states: {
    question: {
      on: {
        CLICK_GOOD: 'thanks',
        CLICK_BAD: 'form'
      }
    },
    form: {
      on: {
        SUBMIT: 'thanks'
      }
    },
    thanks: {
      on: {
        CLOSE: 'closed'
      }
    },
    closed: {}
  }
};
```

Add transition function

```
// Setup the machine
const feedbackMachine = {
  // ...
};

// Setup the state transition function
function transition(state, event) {
  return feedbackMachine
    .states[state]?
    .on?.[event]
    || state;
}
```

Track current state

```
// Setup the machine
const feedbackMachine = {
  // ...
};

let currentState = feedbackMachine.initial;

// Setup the state transition function
function transition(state, event) {
  return feedbackMachine
    .states[state]?
    .on?.[event]
    || state;
}
```

Receive events

```
// Setup the machine
const feedbackMachine = {
  // ...
};

let currentState = feedbackMachine.initial;

// Setup the state transition function
function transition(state, event) {
  return feedbackMachine
    .states[state]?
    .on?.[event]
    || state;
}

function send(event) {
  const nextState = transition(currentState, eve

  currentState = nextState;
}
```

Interpreting state machines

```
1
2  function interpret(machine) {
3    // Keep track of the current state
4    // in a closure
5    let currentState = machine.initial;
6
7    // Receive events
8    const send = (event) => {
9      const nextState =
10       transition(currentState, event);
11
12     // Update the current state
13     currentState = nextState;
14   }
15
16   // Expose the send function
17   return {
18     send
19   };
20 }
```

Interpreting state machines

```
function interpret(machine) {
  let currentState = machine.initial;

  // Keep track of listeners
  const listeners = new Set();

  const send = (event) => {
    const nextState =
      transition(currentState, event);

    currentState = nextState;
  }

  // Register listeners
  const onTransition = (listener) => {
    listeners.add(listener);
  }

  return {
    // Expose way to register listeners
    onTransition,
    send
  };
}
```

Interpreting state machines

```
1
2  function interpret(machine) {
3    let currentState = machine.initial;
4    let status = 1; // 1 means started, 2 means st
5
6    const listeners = new Set();
7
8    const send = (event) => {
9      // Don't accept events if stopped
10     if (status === 2) { return; }
11     // ...
12   }
13
14   const onTransition = (listener) => {
15     listeners.add(listener);
16   }
17
18   // Cleanup
19   const stop = () => {
20     status = 2;
21     listeners.clear();
22   }
23
24   return {
25     onTransition,
26     // Expose cleanup
27     stop,
28     send
29   };
```


```
npm install xstate
```

```
yarn add xstate
```

📚 xstate.js.org/docs

Creating a machine

```
import { createMachine } from 'xstate';

const feedbackMachine = createMachine({
  initial: 'question',
  states: {
    question: {
      // ...
    },
    form: {
      // ...
    },
    thanks: {
      // ...
    },
    closed: {
      // ...
    }
  }
});
```

Adding transitions

Shorthand syntax

```
import { createMachine } from 'xstate';

const feedbackMachine = createMachine({
  initial: 'question',
  states: {
    question: {
      on: {
        CLICK_GOOD: 'thanks',
        CLICK_BAD: 'form'
      }
    },
    form: {
      on: {
        SUBMIT: 'thanks'
      }
    },
    thanks: {
      on: {
        CLOSE: 'closed'
      }
    },
    closed: {
      type: 'final'
    }
  }
});
```

Adding transitions

Object syntax

```
import { createMachine } from 'xstate';

const feedbackMachine = createMachine({
  initial: 'question',
  states: {
    question: {
      on: {
        CLICK_GOOD: 'thanks',
        CLICK_BAD: 'form'
      }
    },
    form: {
      on: {
        SUBMIT: {
          target: 'thanks'
        }
      }
    },
    thanks: {
      on: {
        CLOSE: 'closed'
      }
    },
    closed: {
      type: 'final'
    }
  }
});
```

State objects

```
1 import { createMachine } from 'xstate';
2
3 const feedbackMachine = createMachine({
4   // ...
5 });
6
7 const initialState = feedbackMachine.initialState;
8
9 console.log(initialState);
```

```
// State
{
  value: 'question',
  event: { type: 'xstate.init' },
  context: undefined,
  actions: [],
  // ...
}
```

Event objects

```
import { createMachine } from 'xstate';

const feedbackMachine = createMachine({
  // ...
});

const initialState = feedbackMachine.initialState;

// An event object
const clickGoodEvent = {
  type: 'CLICK_GOOD'
};

// An event object with payload
const submitEvent = {
  type: 'SUBMIT',
  feedback: 'Very good, thank you'
};
```

Transitioning

```
import { createMachine } from 'xstate';

const feedbackMachine = createMachine({
  // ...
});

const initialState = feedbackMachine.initialState;

const clickGoodEvent = {
  type: 'CLICK_GOOD'
};

// Determine next state based on
// current state and event
const nextState = feedbackMachine
  .transition(initialState, clickGoodEvent);

// The event type can be used
// if there is no "payload"
const otherState = feedbackMachine
  .transition(nextState, "CLOSE");

// This is equivalent to passing
// in an event object
const otherState = feedbackMachine
  .transition(nextState, { type: "CLOSE" });
```

Interpreting machines

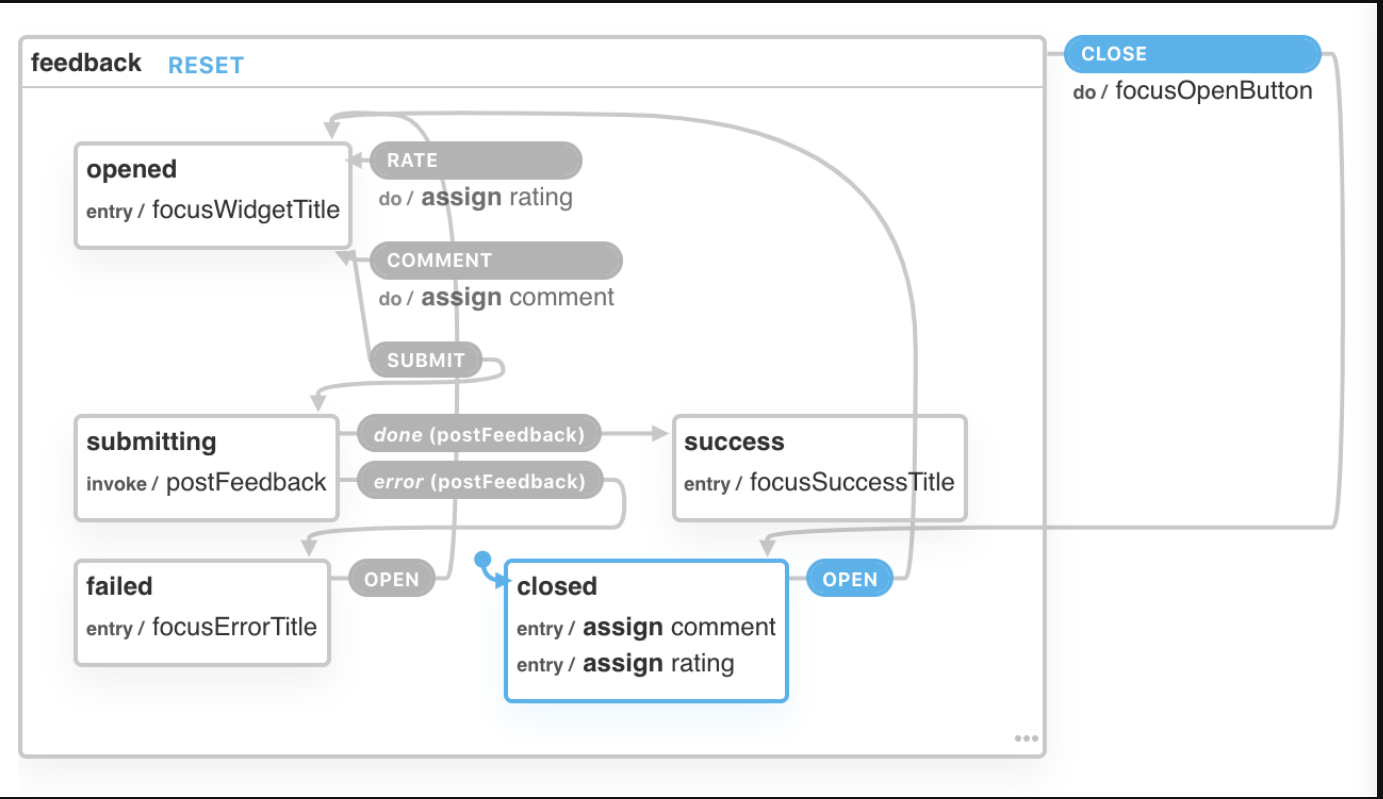
```
import { createMachine, interpret } from 'xstate';

const feedbackMachine = createMachine({
  // ...
});

// Create a "service" instance
const feedbackService = interpret(feedbackMachine);

// You can add a listener that receives
// the current state on every transition
feedbackService.onTransition(state => {
  console.log(state);
});

// At first, the service didn't start yet...
// So start it!
feedbackService.start();
```

Interpreting machines

```
import { createMachine, interpret } from 'xstate';

const feedbackMachine = createMachine({
  // ...
});

// Create a "service" instance and start it
const feedbackService = interpret(feedbackMachine);
  .onTransition(state => {
    console.log(state);
  })
  .start();

// Events can be sent to the service
service.send({ type: 'CLICK_GOOD' });

// If there is no payload, only the
// event type is necessary
service.send("CLOSE");

// When the service no longer needs to run,
// stop it for disposal/cleanup
service.stop();
```

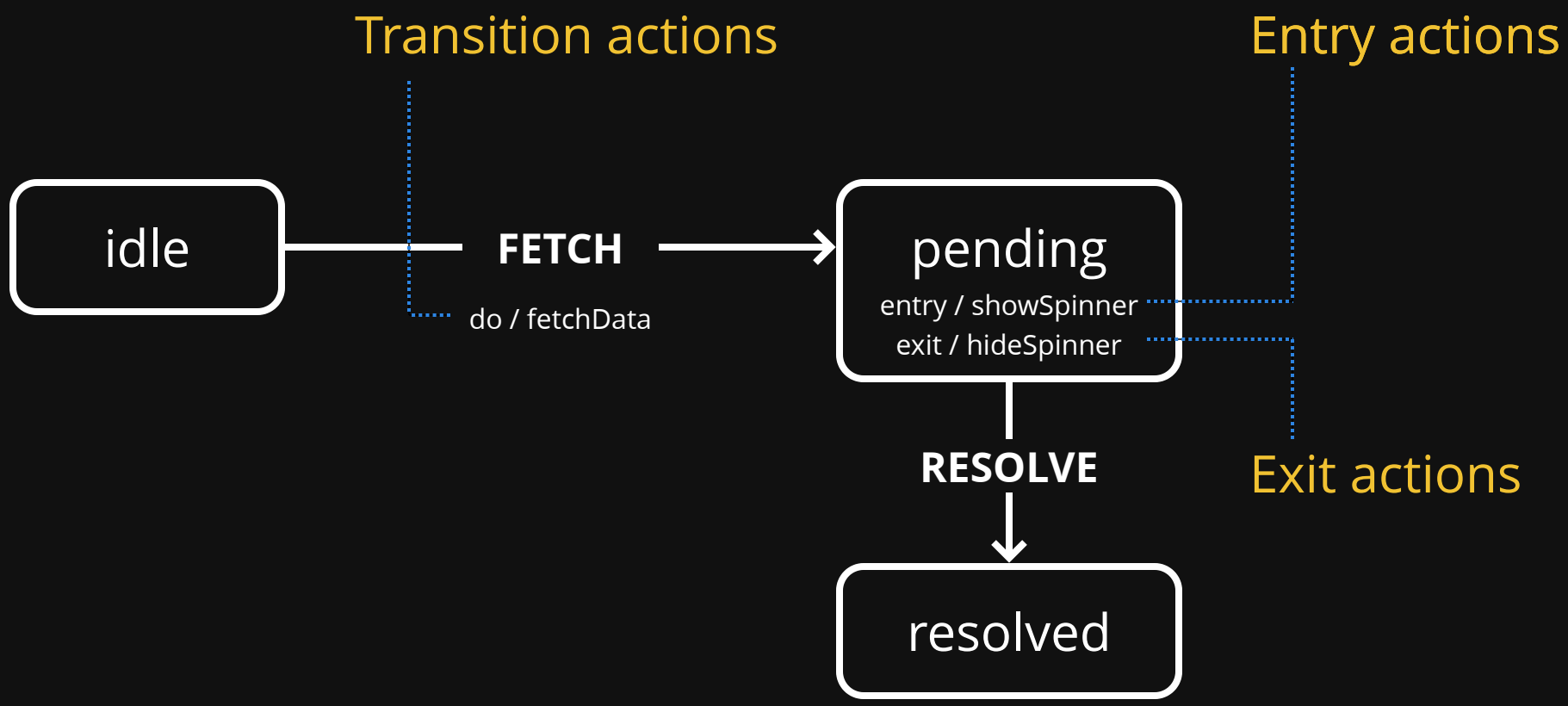
Visualizing machines

xstate.js.org/viz

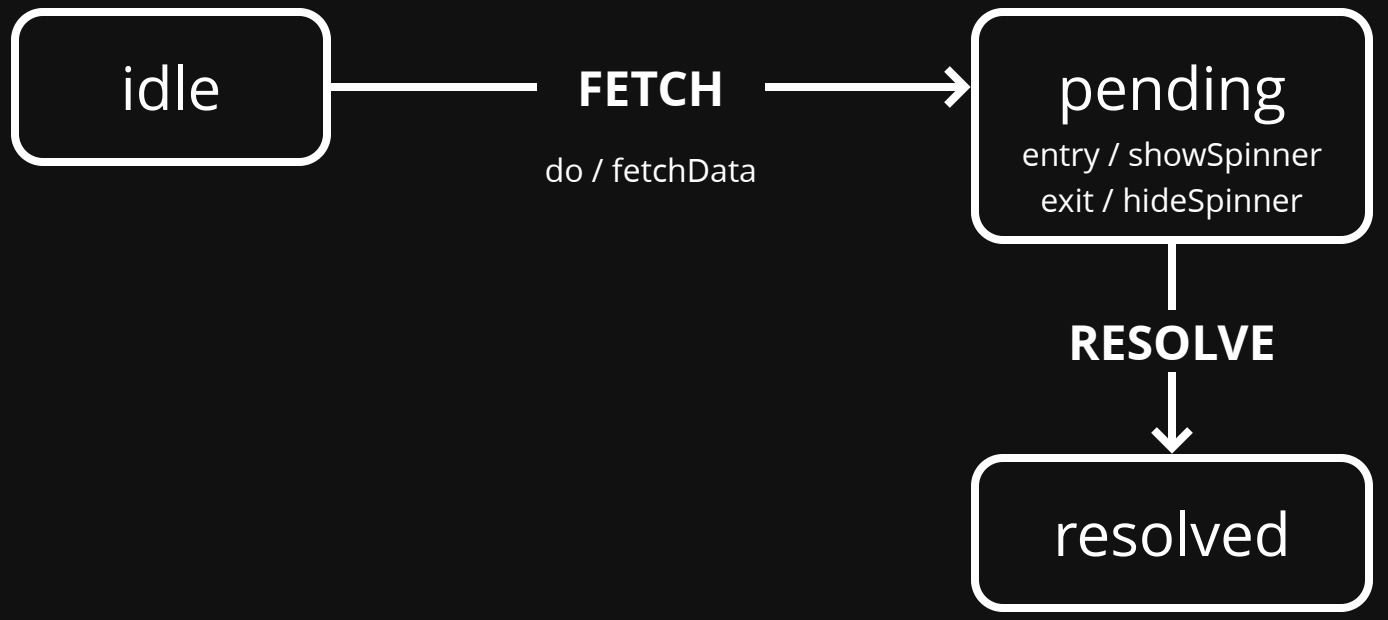


Exercise 03

Actions


Transition actions
Entry actions
idle
FETCH
do / fetchData
pending
entry / showSpinner
exit / hideSpinner
RESOLVE
Exit actions
resolved

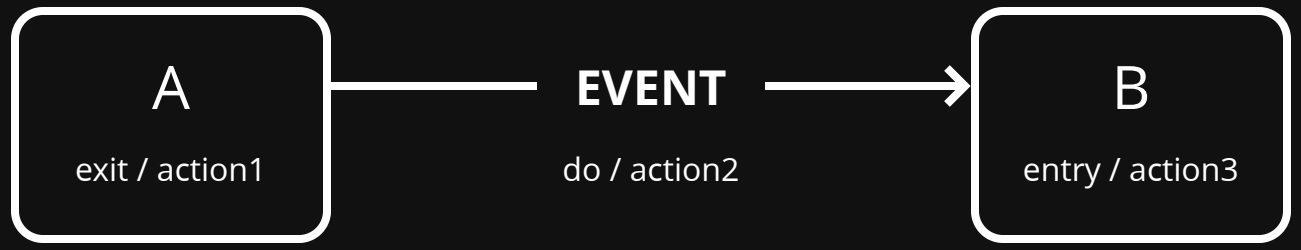
Actions



- state: **idle**
- event: **FETCH**
- state: **pending**
  - action: **fetchData**
  - action: **showSpinner**
- event: **RESOLVE**
- state: **resolved**
  - action: **hideSpinner**

Action order

1. Exit action(s)
2. Transition action(s)
3. Entry action(s)



Actions in XState

```
import { createMachine } from 'xstate';

const someMachine = createMachine({
  initial: 'active',
  states: {
    active: {
      entry: (context, event) => {
        console.log('Entered active');
      },
      on: {
        CLICK: {
          target: 'inactive',
          actions: (context, event) => {
            console.log('Clicked on active');
            console.log(event);
          }
        }
      },
      exit: (context, event) => {
        console.log('Exited active')
      }
    },
    inactive: {
      entry: (context, event) => {
        console.log('Entered inactive');
      }
    }
  }
});
```

Actions in XState

```
import { createMachine } from 'xstate';

const enterActive = () => {
  console.log('Entered active');
};

const clickActive = () => {
  console.log('Clicked on active');
};

const exitActive = () => {
  console.log('Exited active');
};

const enterInactive = () => {
  console.log('Entered inactive');
};

const someMachine = createMachine({
  initial: 'active',
  states: {
    active: {
      entry: enterActive,
      on: {
        CLICK: {
          target: 'inactive',
          actions: clickActive
        }
      },
```

Actions in XState

Multiple actions

```
1  import { createMachine } from 'xstate';
2
3  const sendTelemetry = () => {/* ... */}
4
5  const enterActive = () => {
6    console.log('Entered active');
7  };
8
9  const clickActive = () => {
10   console.log('Clicked on active');
11 };
12
13 const exitActive = () => {
14   console.log('Exited active');
15 };
16
17 const enterInactive = () => {
18   console.log('Entered inactive');
19 };
20
21 const someMachine = createMachine({
22   initial: 'active',
23   states: {
24     active: {
25       entry: [enterActive, sendTelemetry],
26       on: {
27         CLICK: {
28           target: 'inactive',
29           actions: clickActive
```

Actions in XState

Multiple actions

```
import { createMachine } from 'xstate';

const someMachine = createMachine({
  initial: 'active',
  states: {
    active: {
      entry: ['enterActive', 'sendTelemetry'],
      on: {
        CLICK: {
          target: 'inactive',
          actions: 'clickActive'
        }
      },
      exit: 'exitActive'
    },
    inactive: {
      entry: 'enterInactive'
    }
  }
}, {
  actions: {
    sendTelemetry: () => {/* ... */},
    enterActive: () => {
      console.log('Entered active');
    },
    clickActive: () => {
      console.log('Clicked on active');
    },
    exitActive: () => {
```

Exercise 04

Statecharts

Statecharts

Invented by David Harel

Statecharts: A Visual Formalism for Complex Systems (1987)

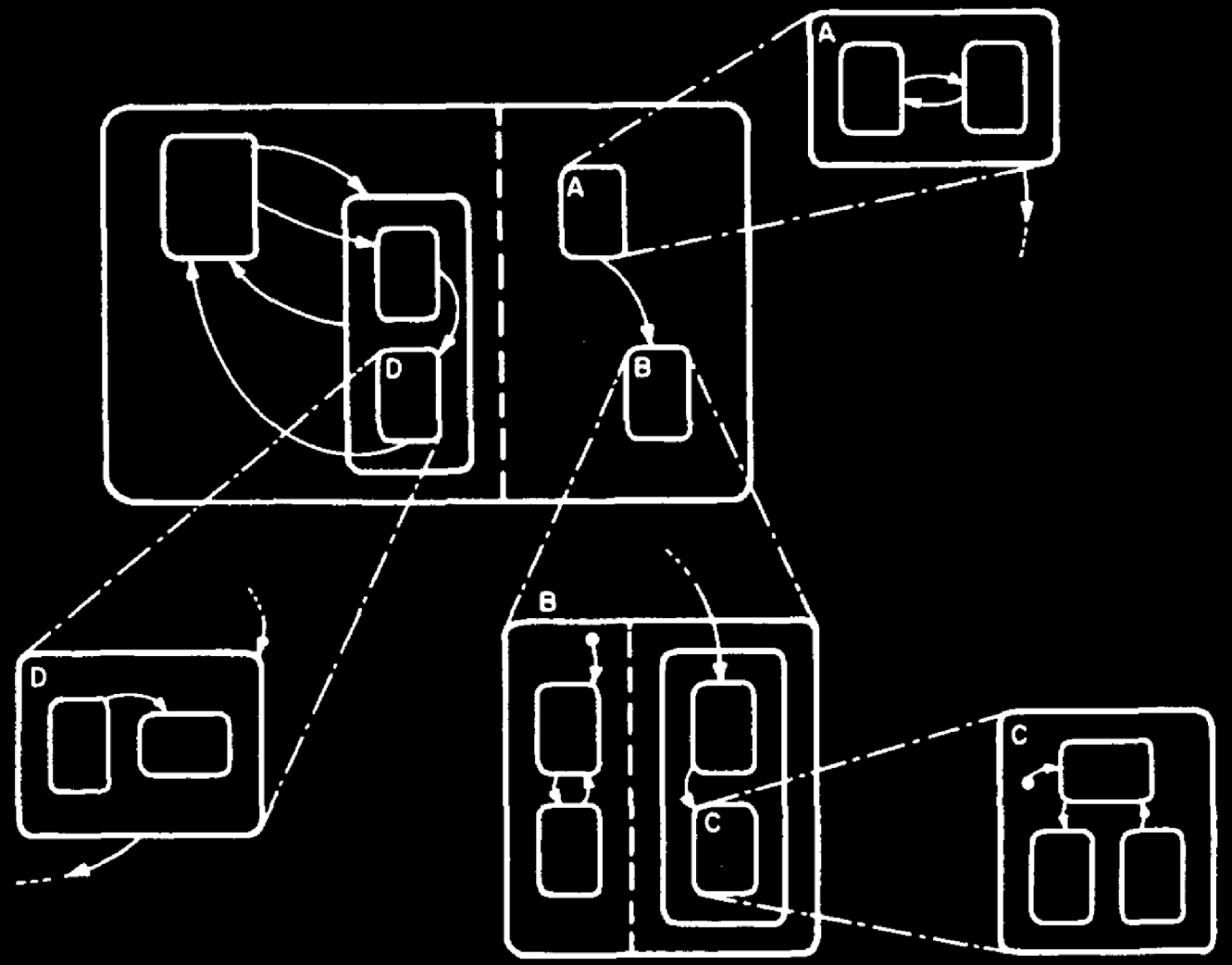


Fig. 36.

Extended state

```
import { createMachine } from 'xstate';

const lightbulbMachine = createMachine({
  initial: 'turnedOff',
  states: {
    turnedOff: {
      on: {
        TOGGLE: {
          target: 'turnedOn',
          actions: 'turnOn'
        }
      }
    },
    turnedOn: {
      on: {
        TOGGLE: {
          target: 'turnedOff',
          actions: 'turnOff'
        }
      }
    }
  }
});
```

Extended state

```
import { createMachine } from 'xstate';

const lightbulbMachine = createMachine({
  initial: 'turnedOff',
  context: {
    count: 0 // # of times it was turned on
  },
  states: {
    turnedOff: {
      on: {
        TOGGLE: {
          target: 'turnedOn',
          actions: 'turnOn'
        }
      }
    },
    turnedOn: {
      on: {
        TOGGLE: {
          target: 'turnedOff',
          actions: 'turnOff'
        }
      }
    }
  }
});
```

Assignment

```
import { createMachine, assign } from 'xstate';

const lightbulbMachine = createMachine({
  initial: 'turnedOff',
  context: {
    count: 0 // # of times it was turned on
  },
  states: {
    turnedOff: {
      on: {
        TOGGLE: {
          target: 'turnedOn',
          actions: 'turnOn'
        }
      }
    },
    turnedOn: {
      entry: assign({
        count: (context, event) => {
          return context.count + 1;
        }
      }),
      on: {
        TOGGLE: {
          target: 'turnedOff',
          actions: 'turnOff'
        }
      }
    }
```

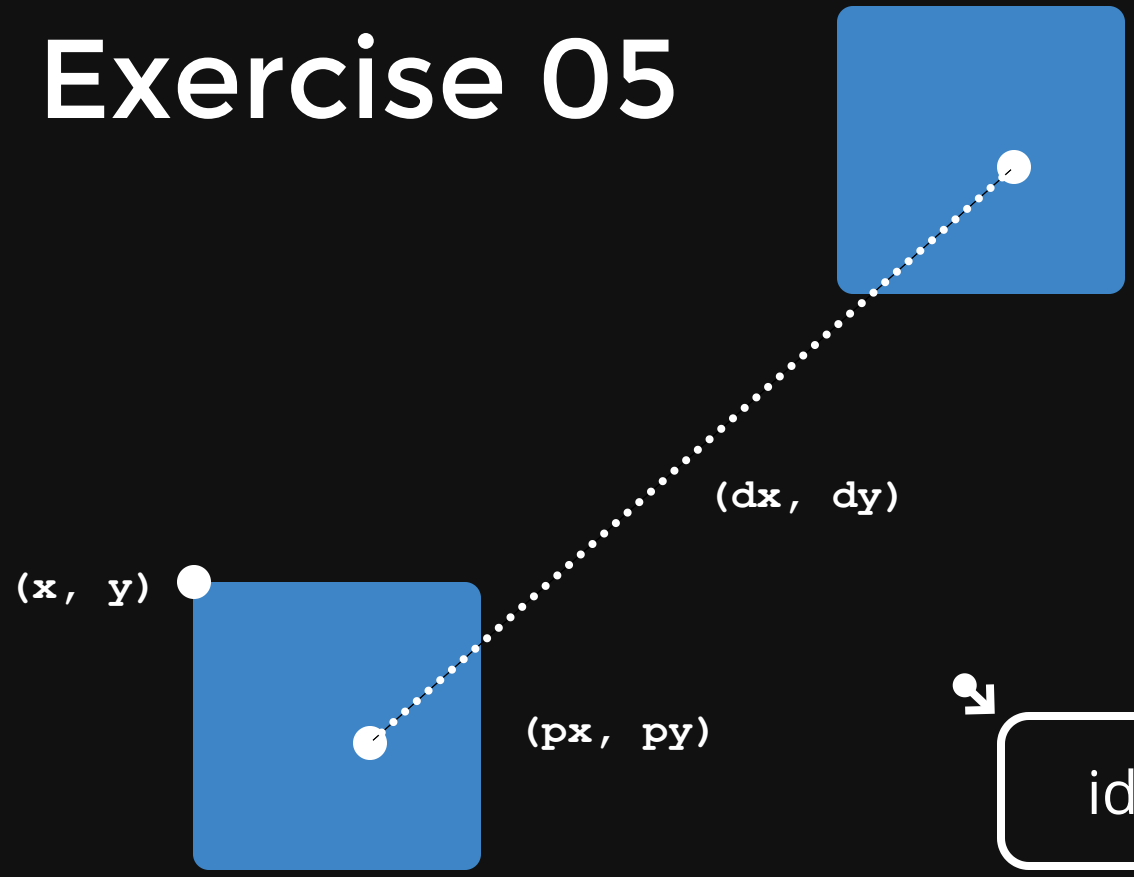
Assignment

```
import { createMachine, assign } from 'xstate';

const assignAction = assign({
  // static assignment
  message: 'Hello',

  // assignment based on context + event
  count: (context, event) => {
    return context.count + event.value;
  }
});

console.log(assignAction);
```

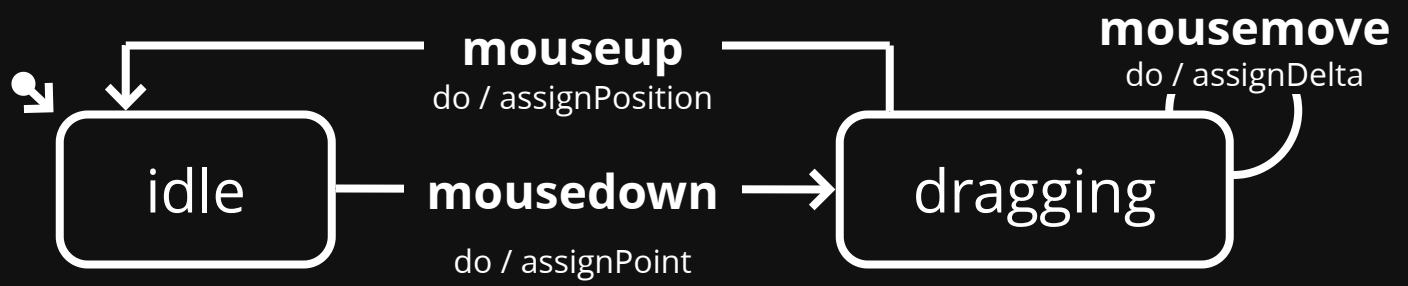
Assignment

```
1  import { createMachine, assign } from 'xstate';
2
3  const assignAction = assign((context, event) => {
4    return {
5      message: 'hello',
6      count: context.count + event.value
7    };
8  });
9
10 console.log(assignAction);
```

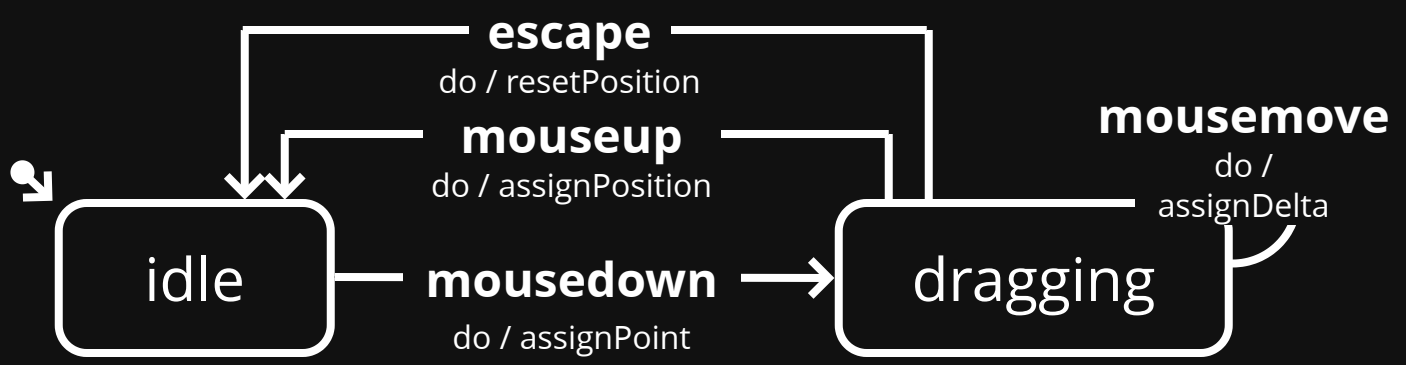
Exercise 05



Exercise 05


mouseup
do / assignPosition
mousemove
do / assignDelta
idle
mousedown
do / assignPoint
dragging

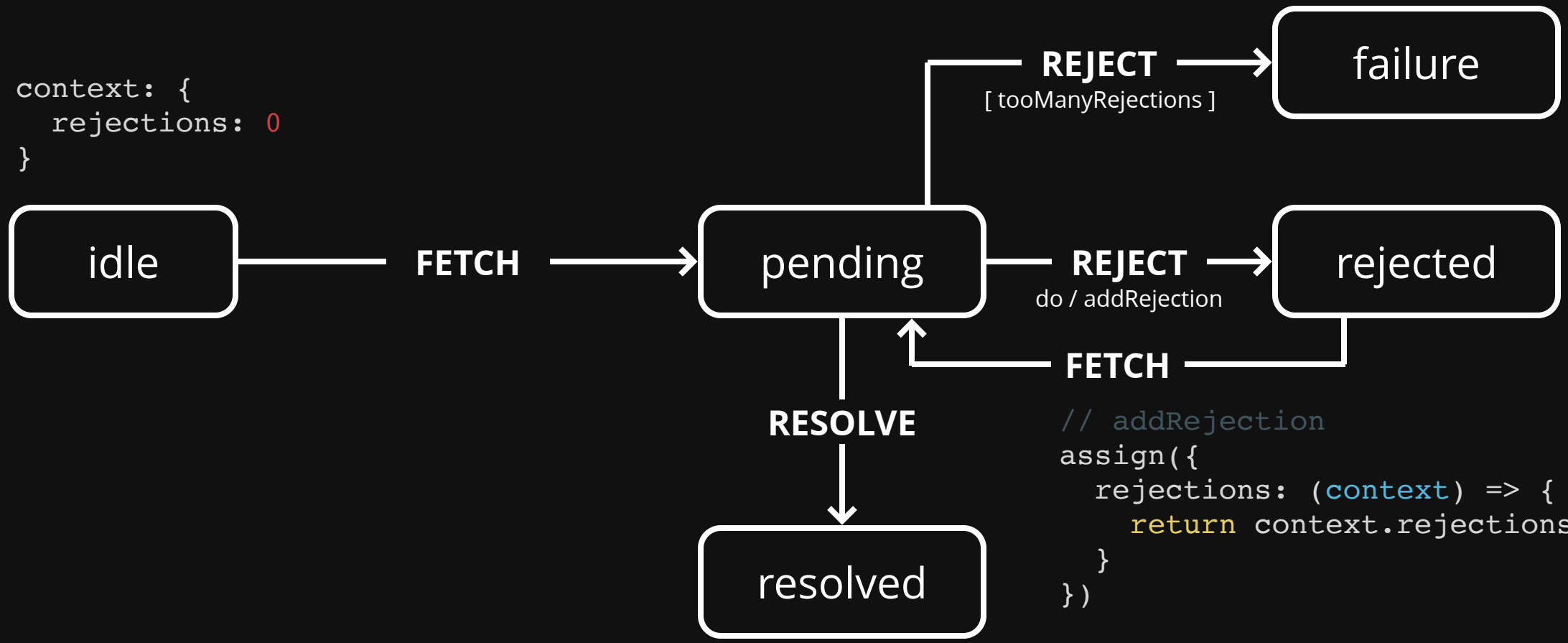
Exercise 05


escape
do / resetPosition
mouseup
do / assignPosition
mousemove
do /
assignDelta
idle
mousedown
do / assignPoint
dragging

Guarded transitions

```
const tooManyRejections = (context, event) =>
  return context.rejections >= 5;
}
```

```
context: {
  rejections: 0
}
```




```
// addRejection
assign({
  rejections: (context) => {
    return context.rejections + 1;
  }
})
```

Guarded transitions

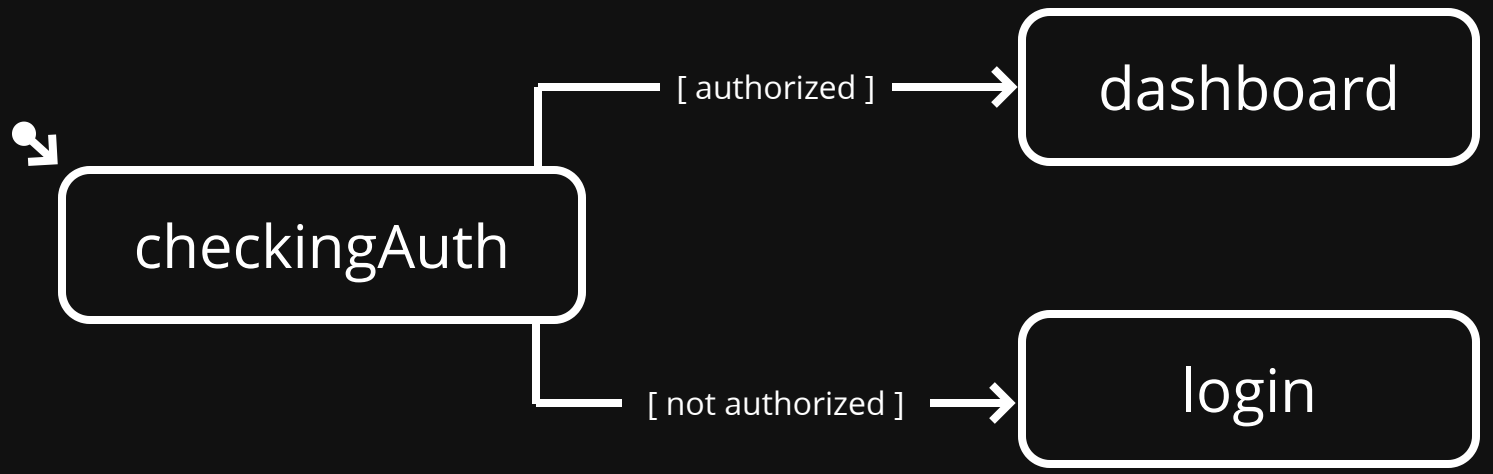
```
import { createMachine } from 'xstate';

const authMachine = createMachine({
  initial: 'idle',
  context: {
    rejections: 0
  },
  states: {
    idle: {
      on: {
        FETCH: 'pending'
      }
    },
    pending: {
      on: {
        REJECT: [
          {
            target: 'failure',
            cond: (context) => {
              return context.rejections >= 5
            }
          },
          {
            target: 'rejected',
            actions: assign({
              retries: (context) => context.retries + 1
            })
          }
        ],
```

Guarded transitions

```
import { createMachine } from 'xstate';

const tooManyRejections = (context) => {
  return context.rejections >= 5
};

const addRejection = assign({
  retries: (context) => context.retries + 1
});

const authMachine = createMachine({
  initial: 'idle',
  context: {
    rejections: 0
  },
  states: {
    idle: {
      on: {
        FETCH: 'pending'
      }
    },
    pending: {
      on: {
        REJECT: [
          {
            target: 'failure',
            cond: tooManyRejections
          },
          {
```

Exercise 06

Transient transitions



Transient transitions

```
import { createMachine } from 'xstate';

const isAuthorized = (context, event) => {
  return !!context.user;
}

const appMachine = createMachine({
  initial: 'checkingAuth',
  context: {
    user: null
  },
  states: {
    checkingAuth: {
      on: {
        '': [
          {
            target: 'dashboard',
            cond: isAuthorized
          },
          { target: 'login' }
        ]
      }
    },
    login: {},
    dashboard: {}
  }
});
```

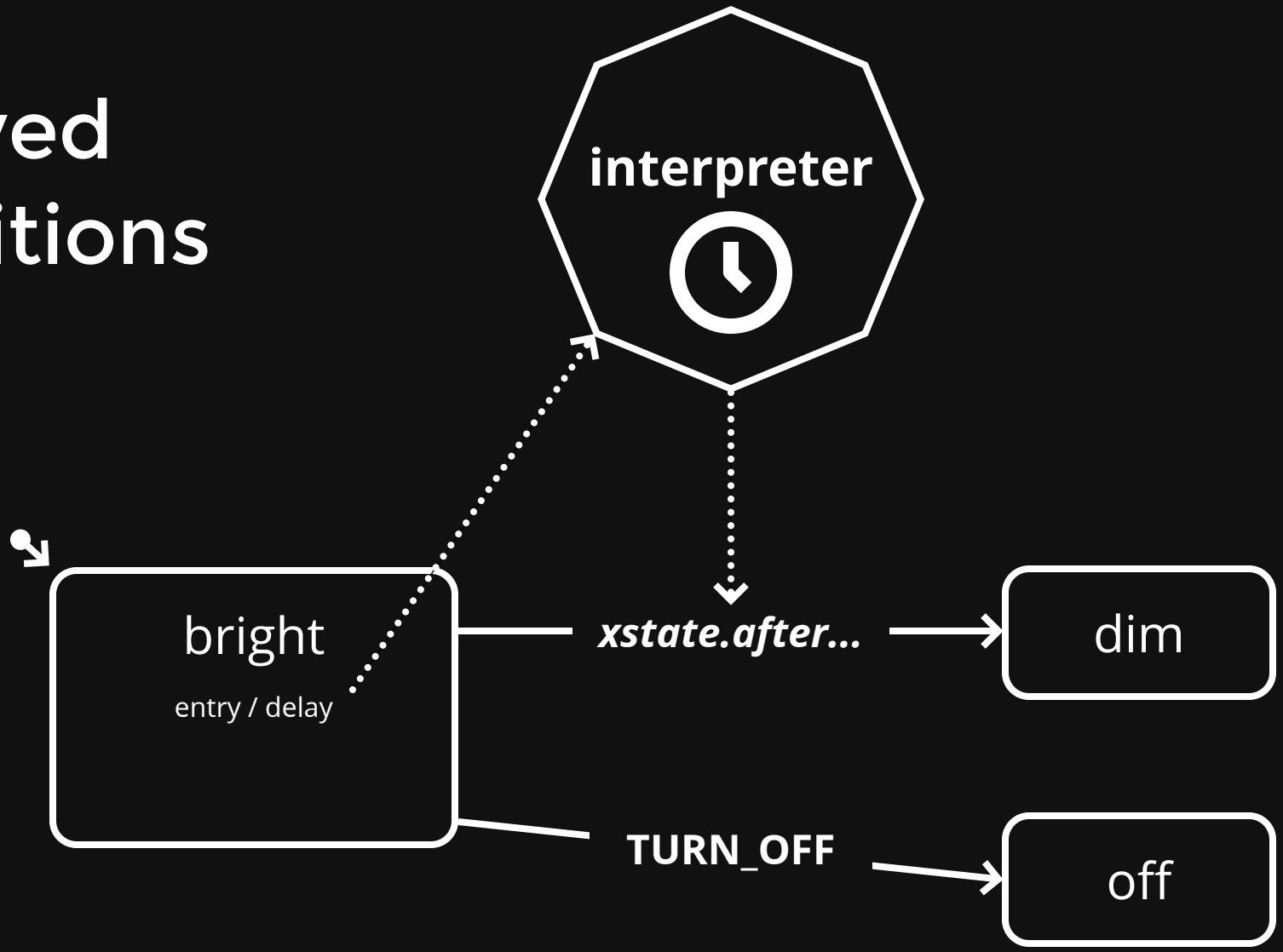
Transient transitions

```
import { createMachine } from 'xstate';

const isAuthorized = (context, event) => {
  return !!context.user;
}

const appMachine = createMachine({
  initial: 'checkingAuth',
  context: {
    user: null
  },
  states: {
    checkingAuth: {
      on: {
        '': [
          {
            target: 'dashboard',
            cond: 'isAuthorized'
          },
          { target: 'login' }
        ]
      }
    },
    login: {},
    dashboard: {}
  }
}, {
  guards: {
    isAuthorized
```

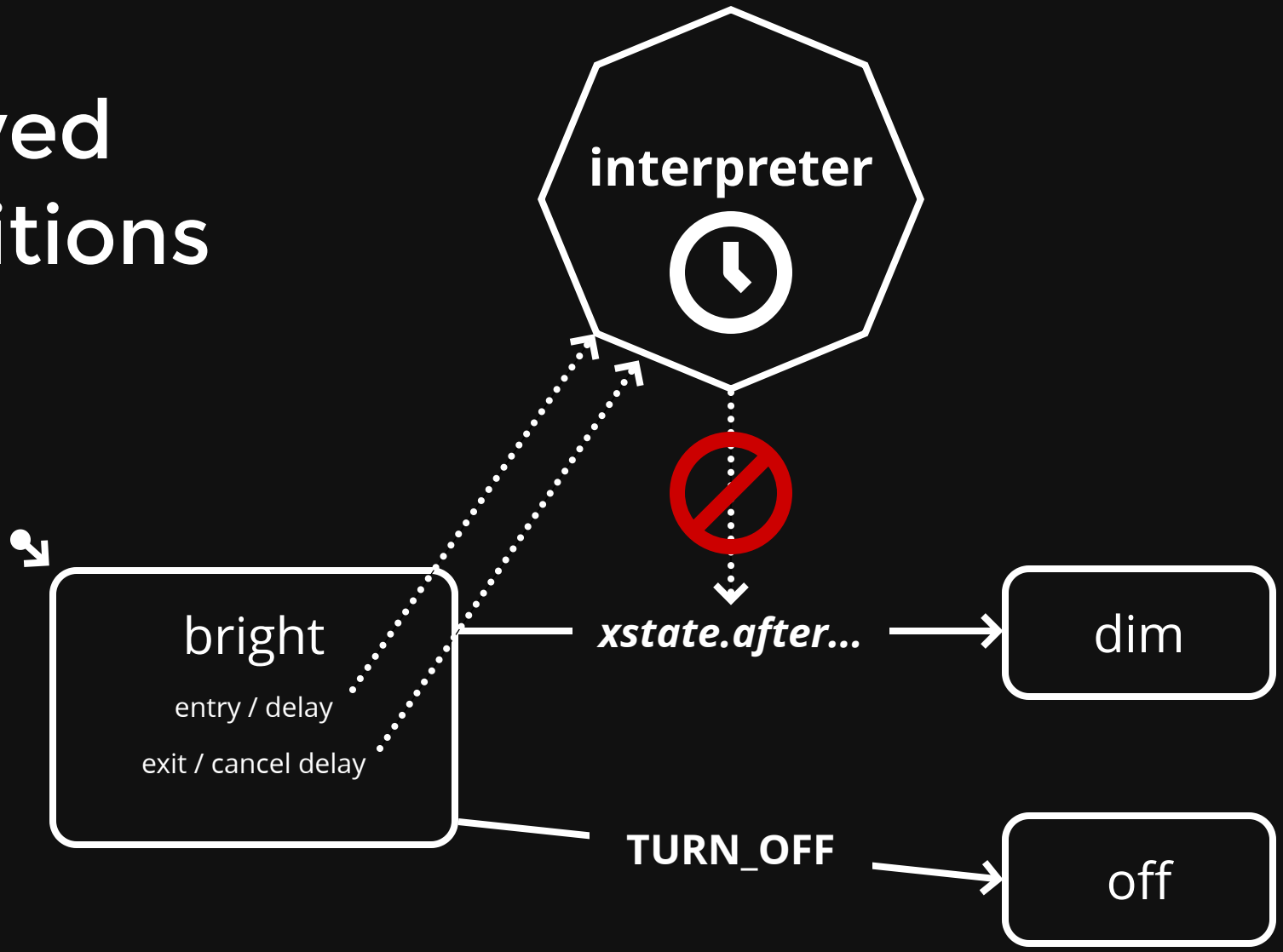
Exercise 07

Delayed transitions

Delayed transitions

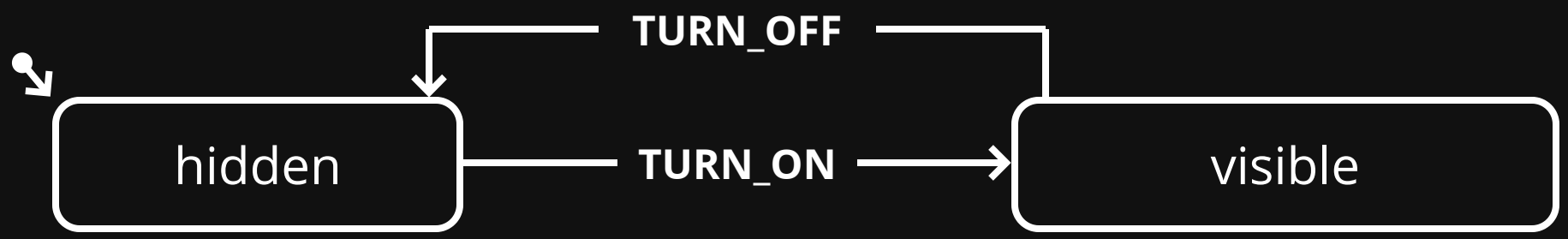


Delayed transitions

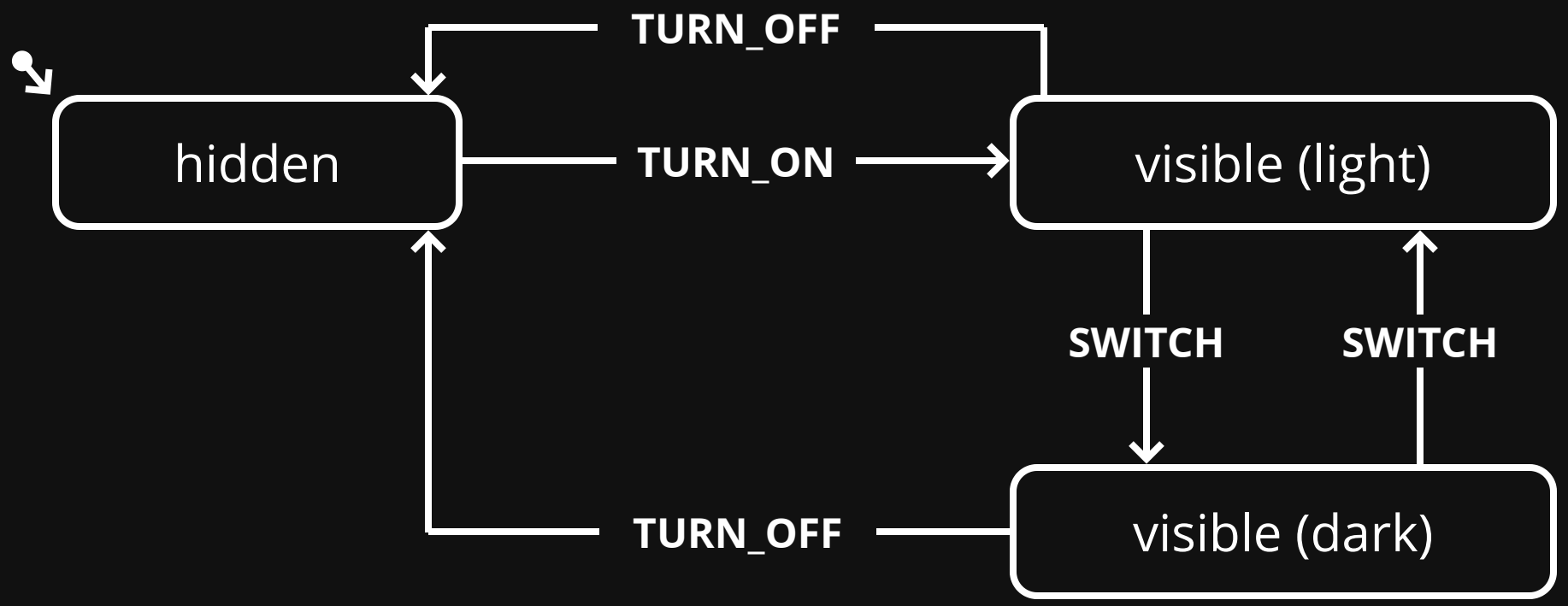


Exercise 08

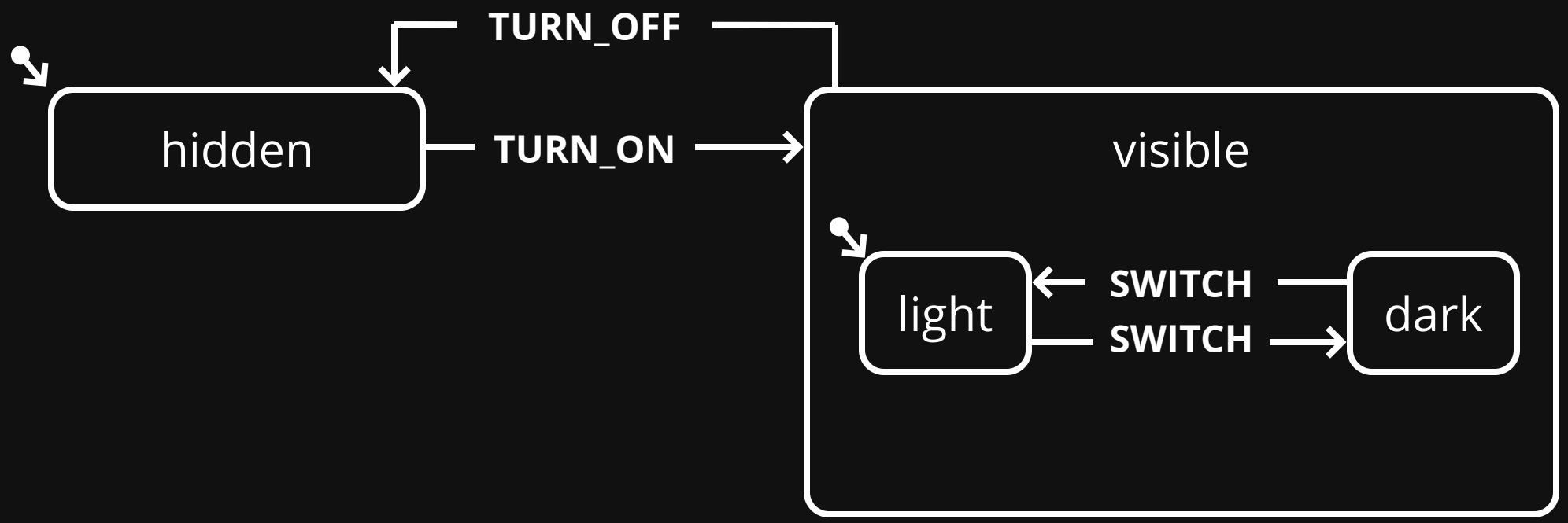
Hierarchical states



Hierarchical states

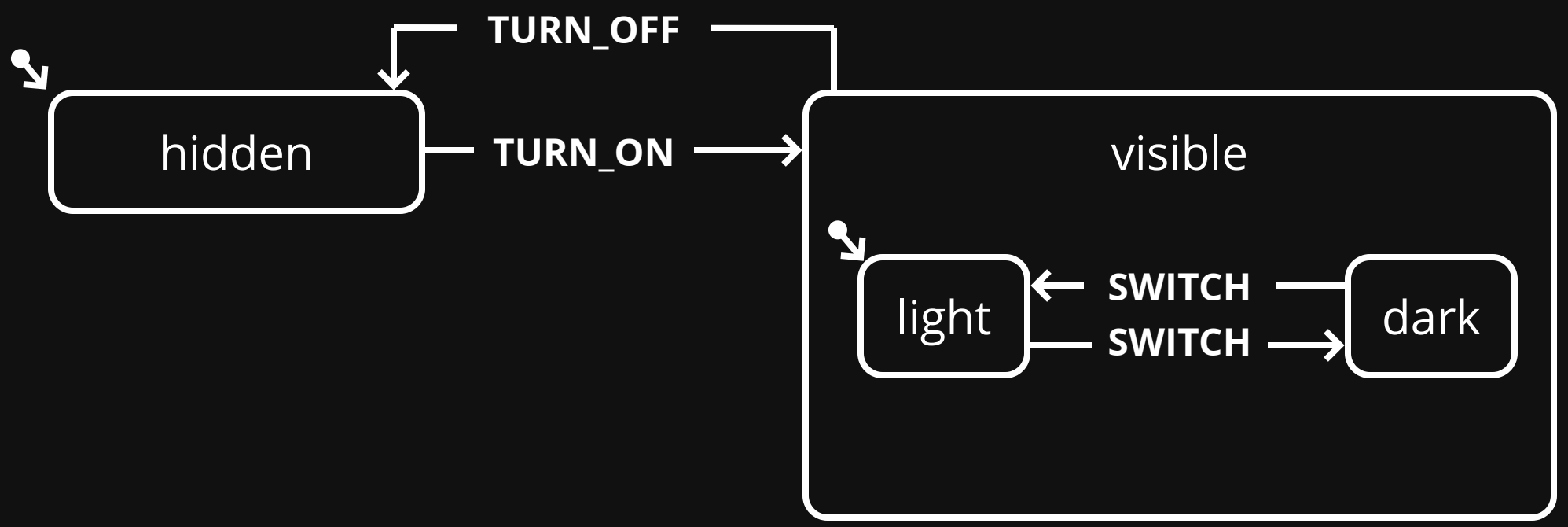


Hierarchical states

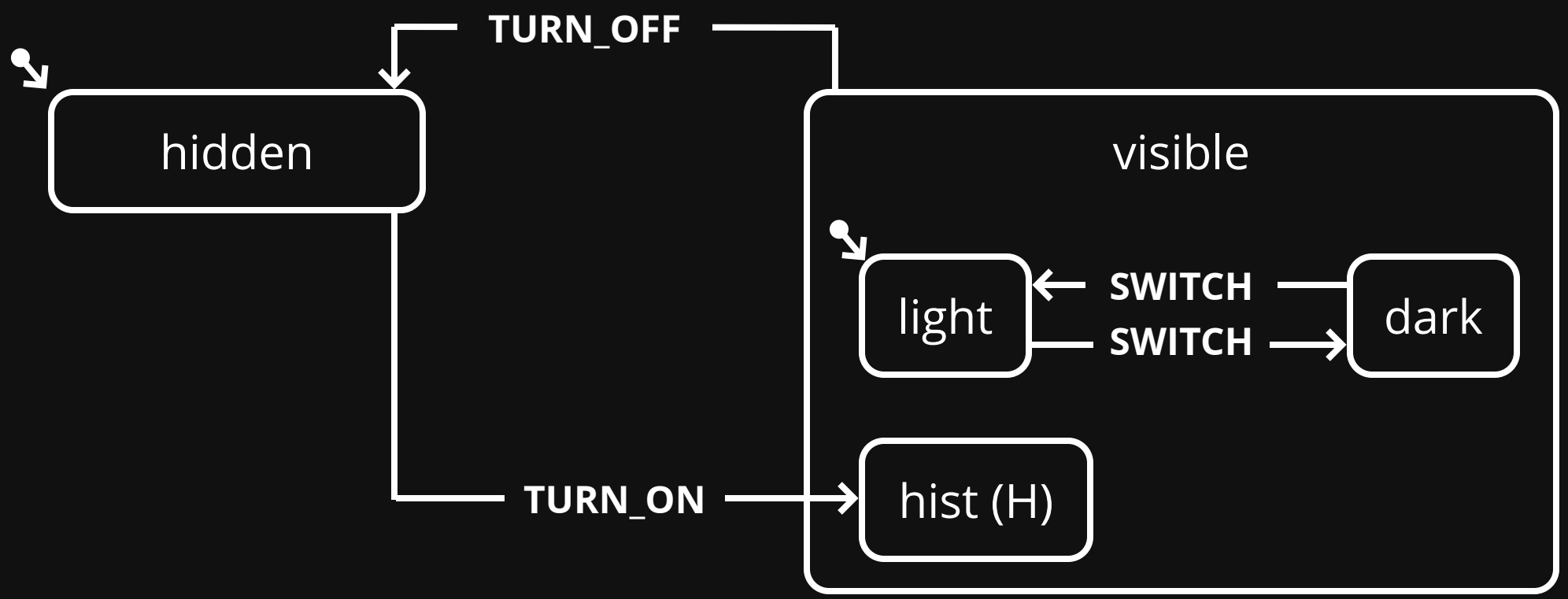


Exercise 09

History states

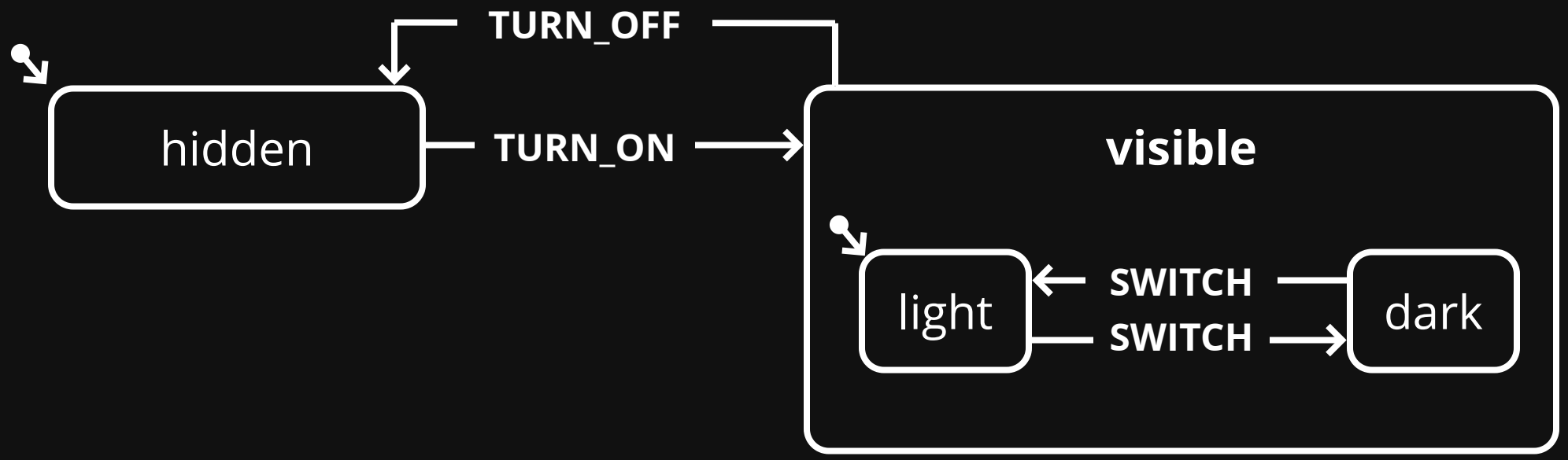


History states

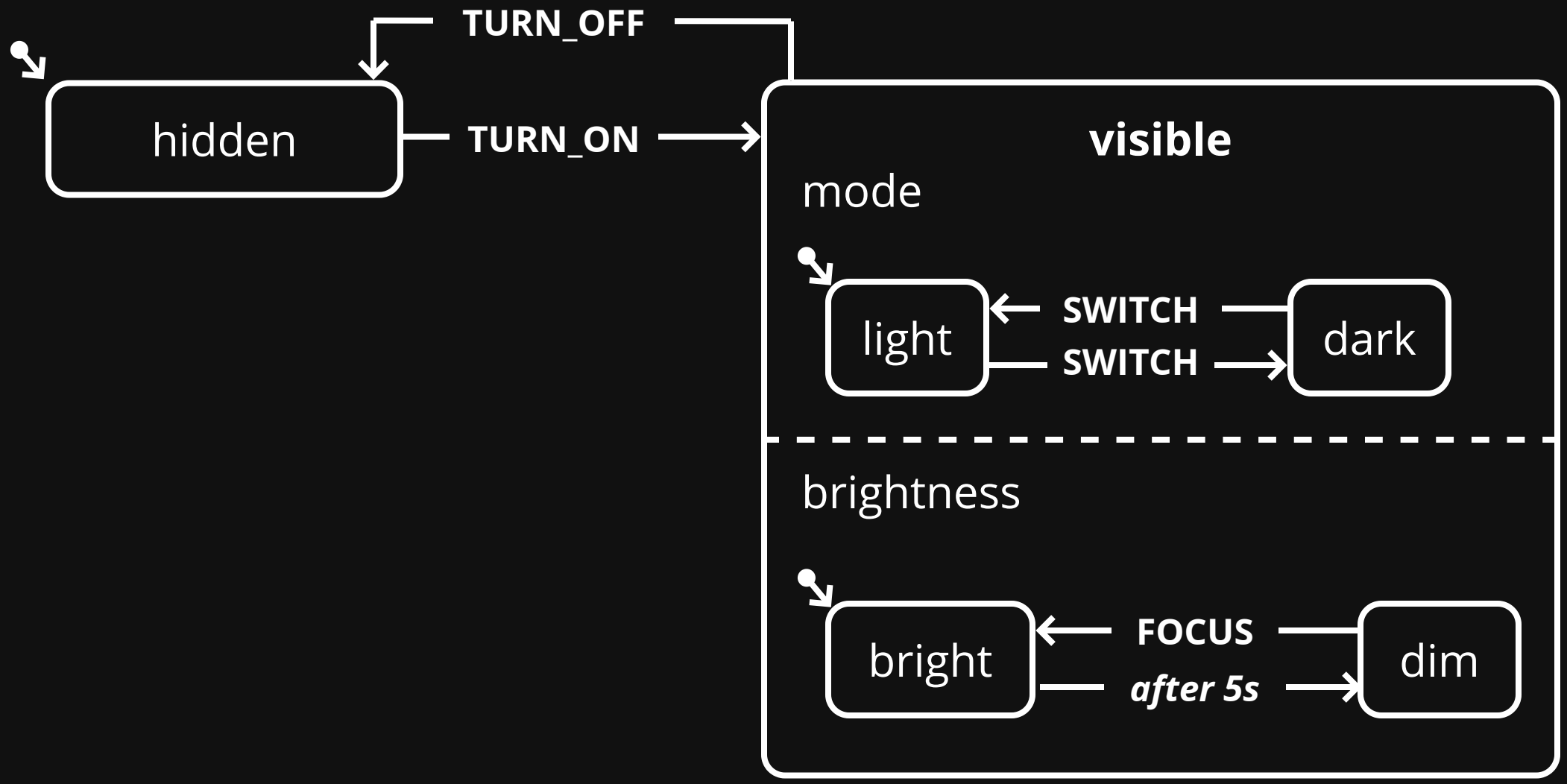


Exercise 10

Parallel states



Parallel states

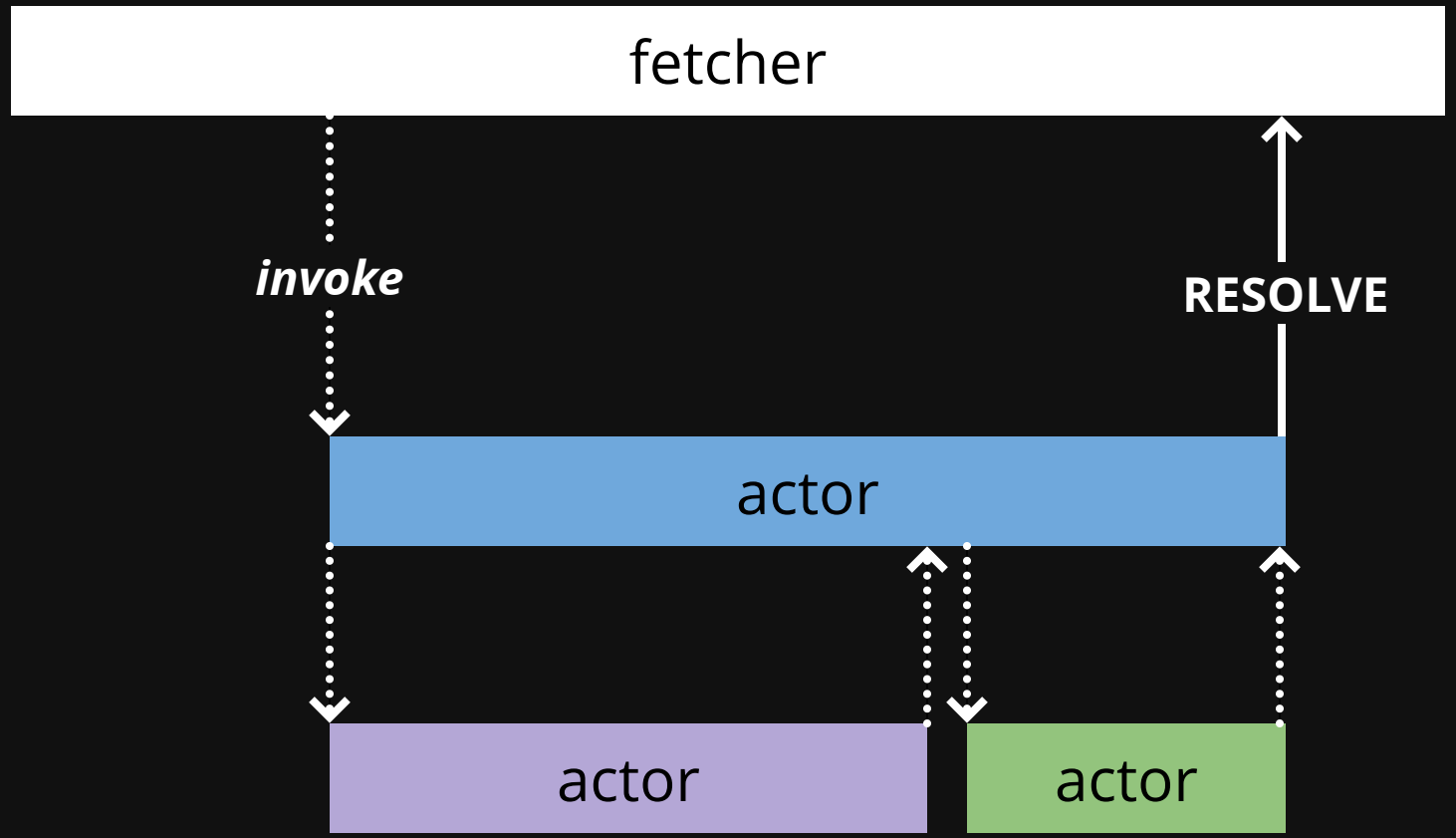


Exercise 11

The actor model

An actor can:

- send a message
- spawn new actor(s)
- change its behavior



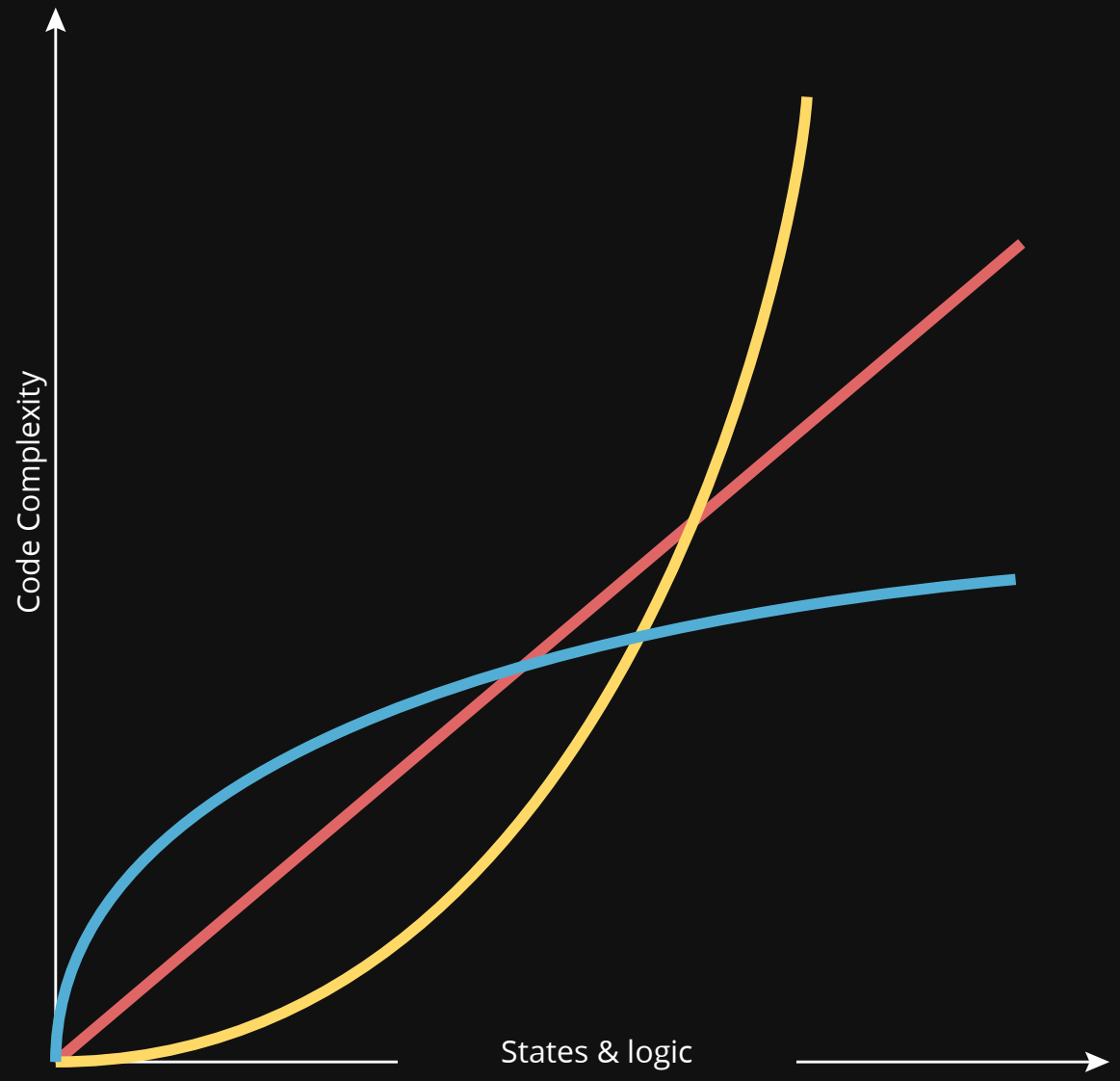
Exercise 12

Further topics

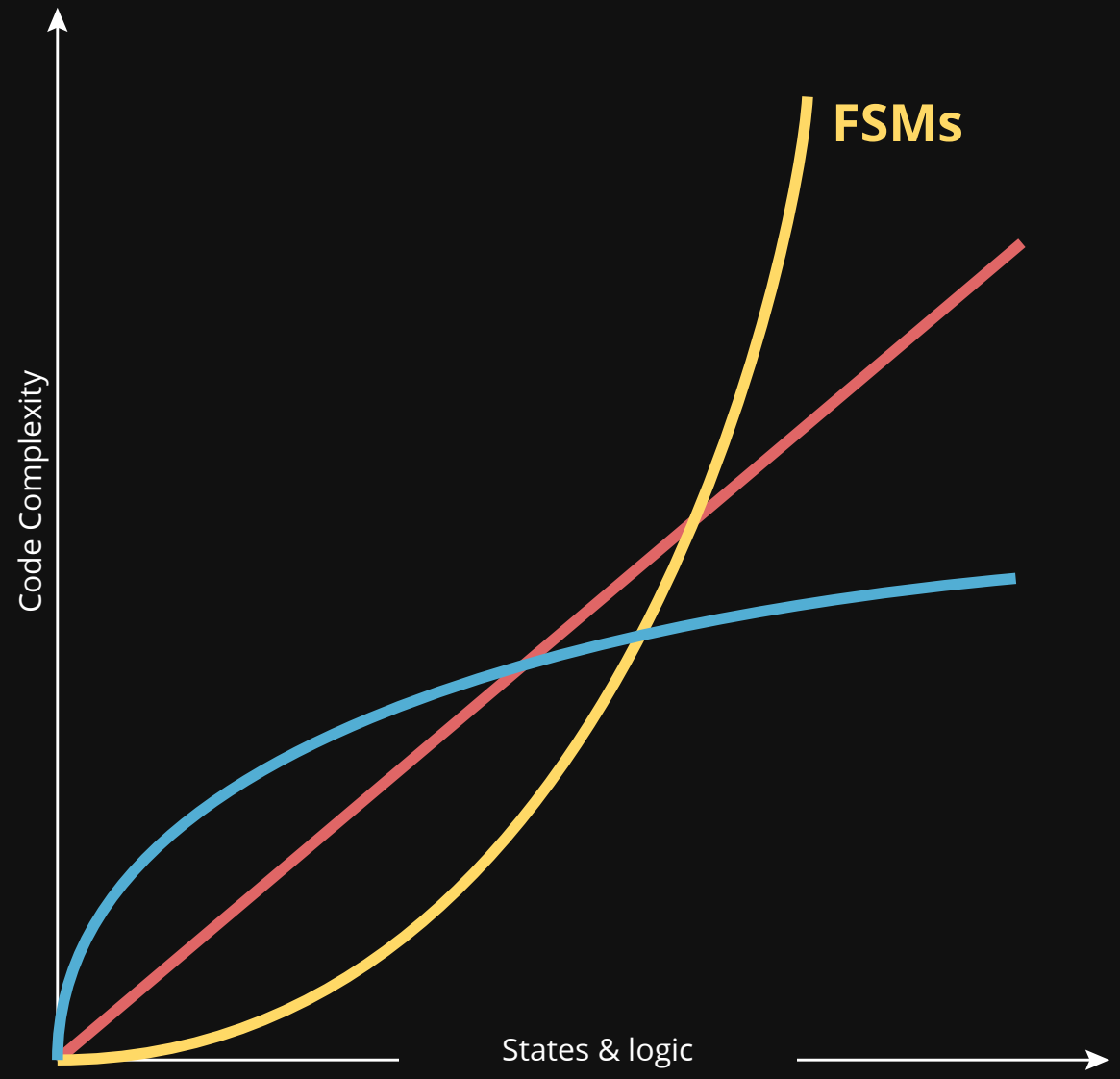
- XState + React with **@xstate/react**
- XState + Vue with **@xstate/vue**
- *XState + Angular (coming soon)*
- *XState + Svelte (coming soon)*
- Model-based-testing with **@xstate/test**
- Graph generation with **@xstate/graph**

📚 xstate.js.org/docs

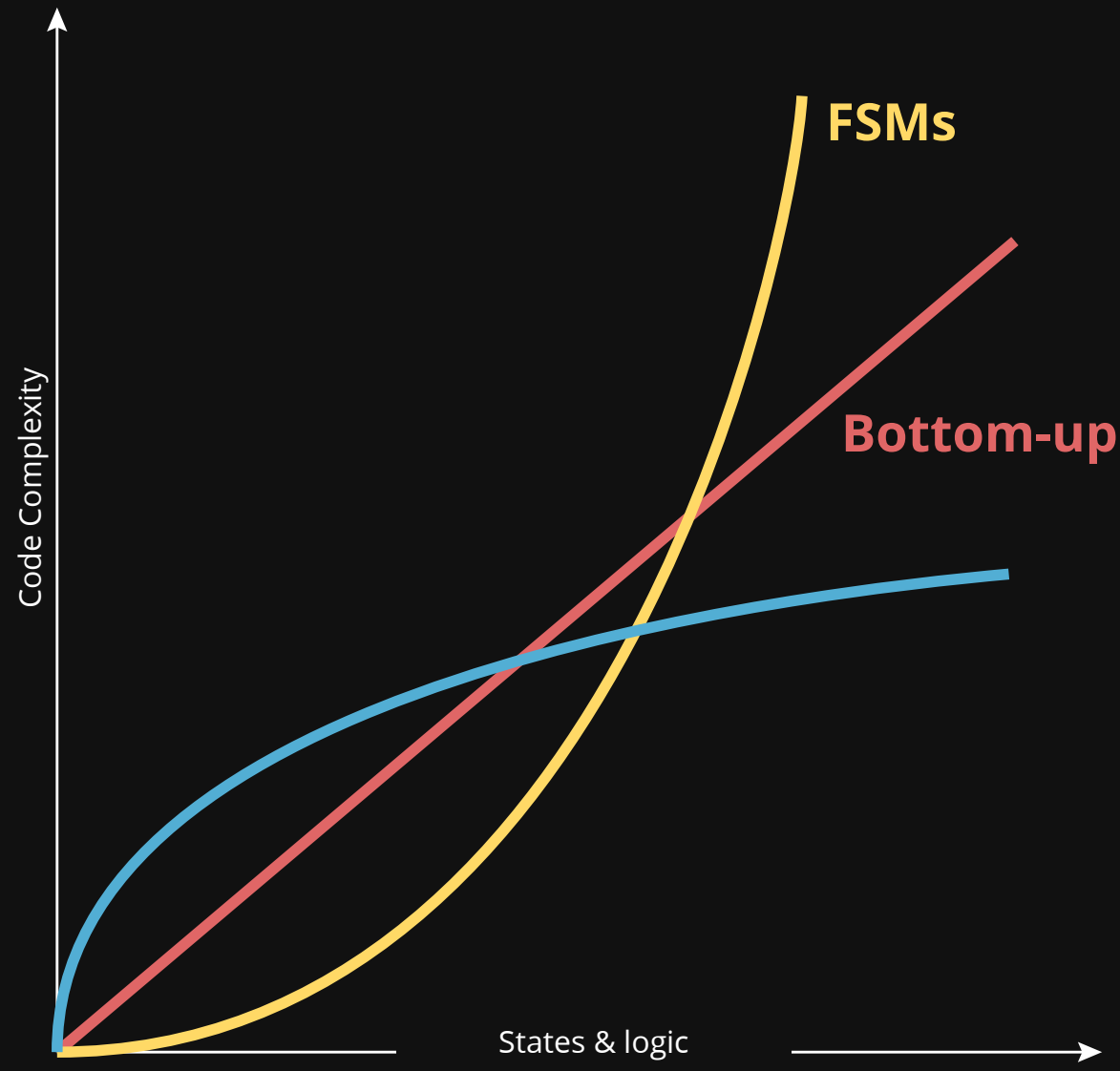
Complexity trade-offs



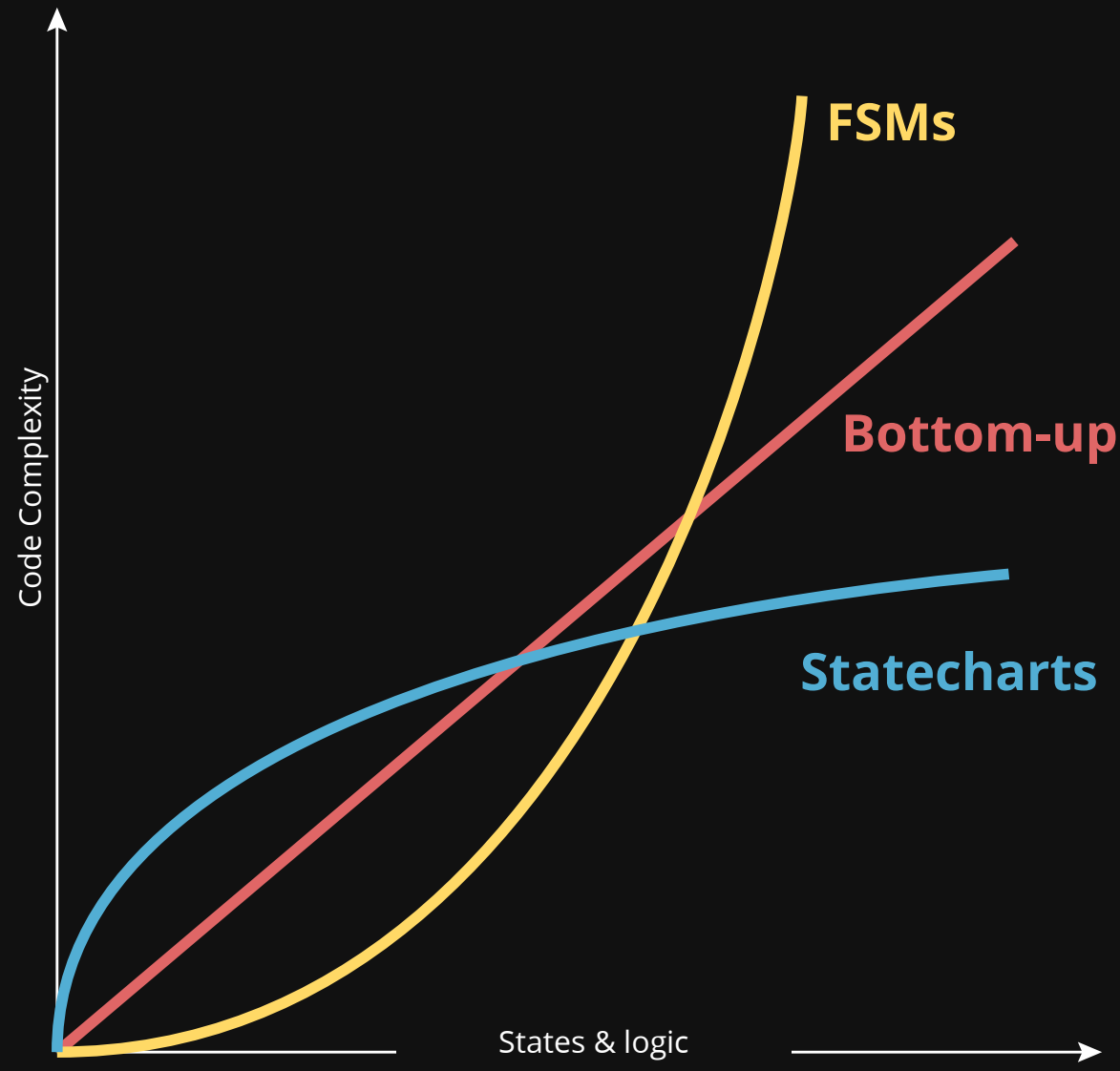
Complexity trade-offs



Complexity trade-offs



Complexity trade-offs



The world of statecharts

statecharts.github.io

Spectrum community

spectrum.chat/statecharts

XState Documentation

xstate.js.org/docs