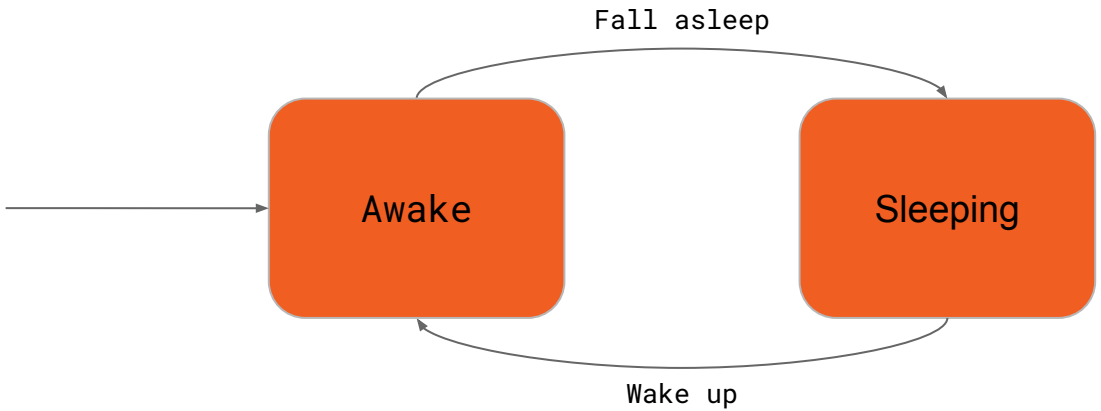
# The
# Typestate Pattern

“Easy to use and hard to misuse”

# Typestate? What?

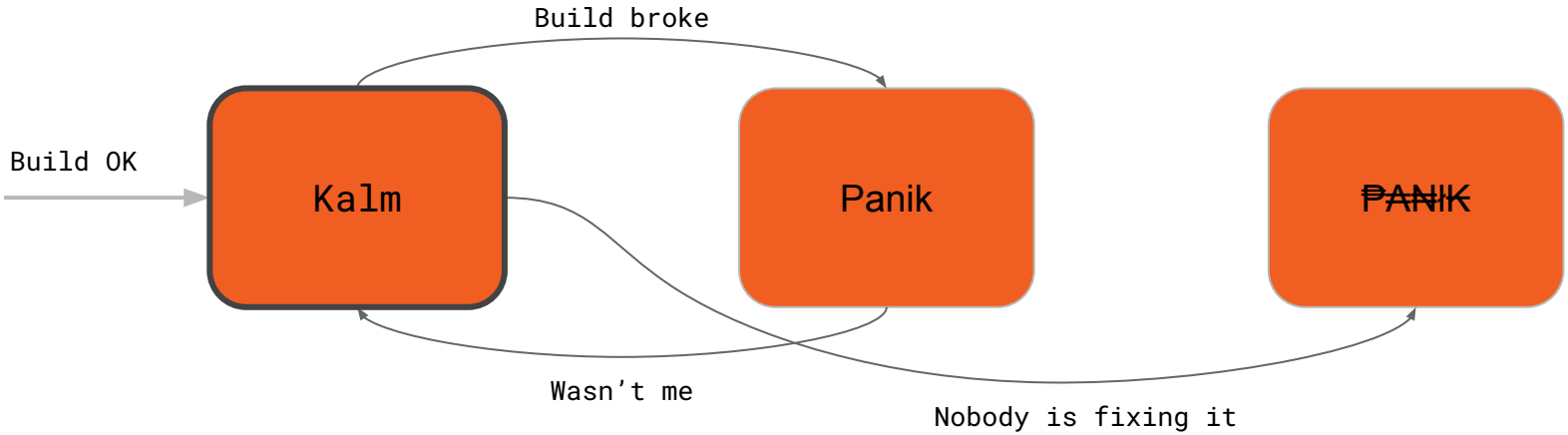
- Express State Machines in code
  - Graph-like structures, think “Coffee Machine”
  - Structures must be known statically
  - State transitions can be I/O, communication, …

# State Machine? What?

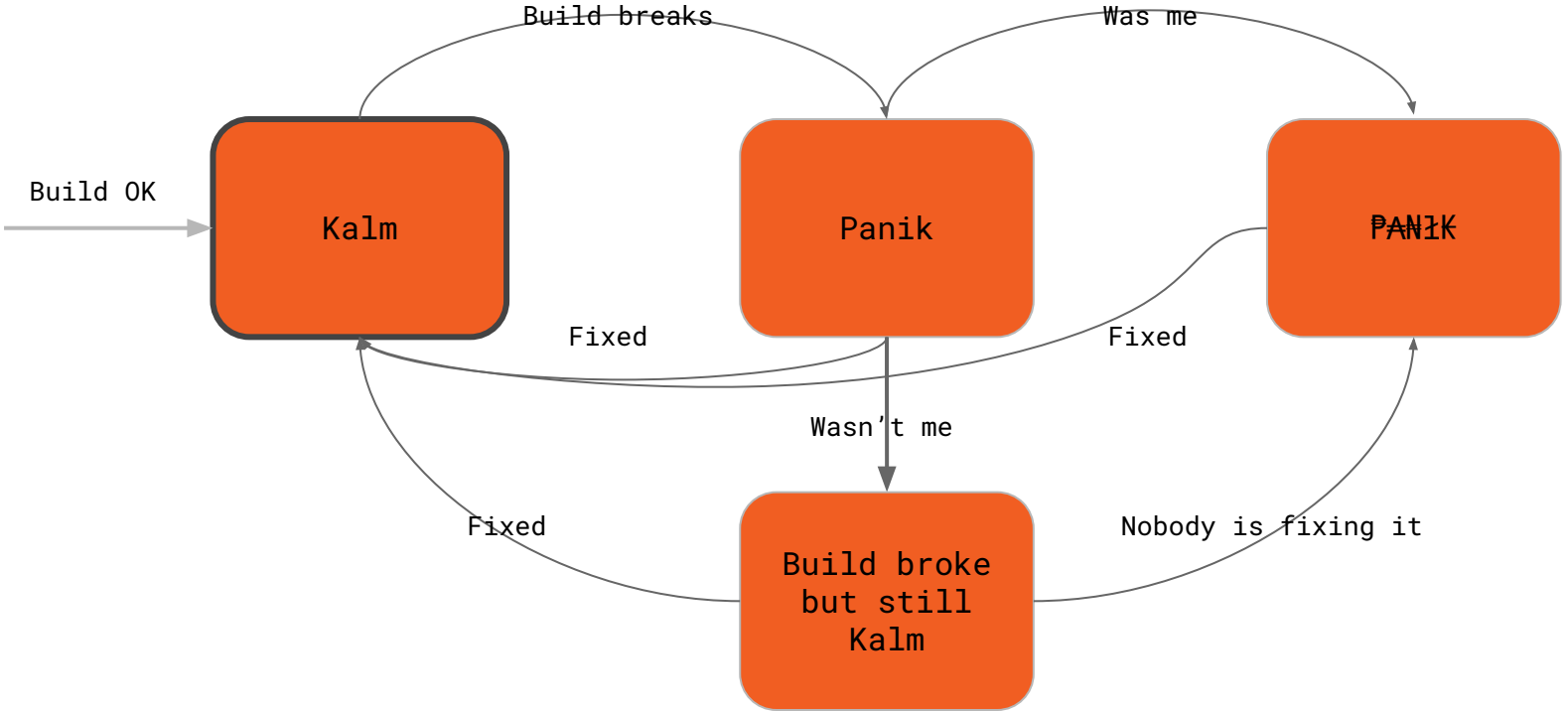


# State Machine? What?

# State Machine? What? Bad Example

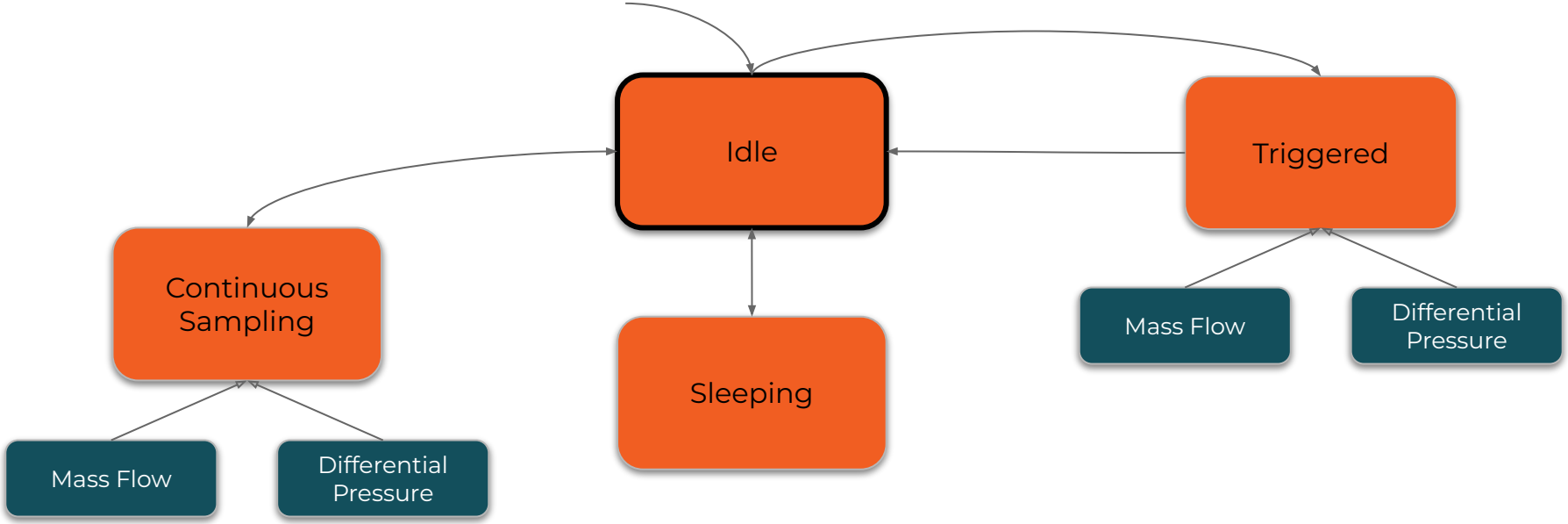


# State Machine? What? Better Example



# State Machine? What? Another Example

SDP8xx Sensor Datasheet lists operation modes

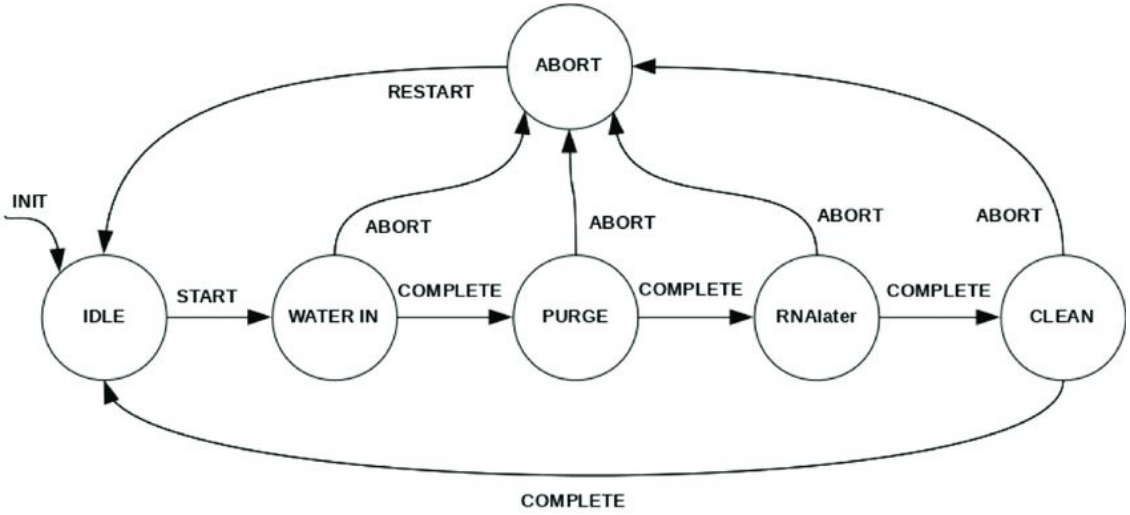



https://github.com/barafael/sdp8xx-rs

datasheet:
https://www.sensirion.com/de/durchflusssensoren/differenzdrucksensoren/sdp800-proven-and-improved/

# State Machines are EVERYWHERE

- Hardware drivers
- Protocol decoding
- Aspects of async/await
- Physical processes
- Model Checking




Ribeiro, H., Martins, A., Goncalves, M., Guedes, M., Tomasino, M., Dias, N., Dias, A., Mucha, A., Carvalho, M., Almeida, C., Ramos, S., Almeida, J., Silva, E., & Magalhães, C. (2019). Development of an autonomous biosampler to capture in situ aquatic microbiomes. *PLOS ONE, 14, e0216882.*

# How to implement? YOLO Approach

```
readBuffer == "AT+C001";
digitalWrite(setPin, LOW);        // Set HC-12 into AT Command mode
delay(100);                       // Wait for the HC-12 to enter AT Command mode
HC12.print(readBuffer);           // Send AT Command to HC-12 ("AT+C001")
delay(200);
while (HC12.available()) {        // If HC-12 has data (the AT Command response)
  Serial.write(HC12.read());      // Send the data to Serial monitor
}
Serial.println("Channel successfully changed");
digitalWrite(setPin, HIGH);       // Exit AT Command mode
readBuffer = "";
```

# How to implement? YOLO Approach

```
readBuffer == "AT+C001";
digitalWrite(setPin, LOW);        // Set HC-12 into AT Command mode
delay(100);                       // Wait for the HC-12 to enter AT Command mode
HC12.print(readBuffer);           // Send AT Command to HC-12 ("AT+C001")
delay(200);
while (HC12.available()) {        // If HC-12 has data (the AT Command response)
  Serial.write(HC12.read());      // Send the data to Serial monitor
}
Serial.println("Channel successfully changed");
digitalWrite(setPin, HIGH);       // Exit AT Command mode
readBuffer = "";
```

What are the (many) drawbacks here?

# How to implement? Enum Approach

Javadoc path: com.google.gdata.data.projecthosting **Enum State.Value**

An issue can be:

| Enum Constant Summary |
| --- |
| **CLOSED**<br>Closed state. |
| **OPEN**<br>Open state. |

Issue state needs to be checked for each operation.

An issue can be closed more than once.

State-local data “validity” depends on values of other variables.

com.google.gdata.data.projecthosting

## Enum State.Value

```
java.lang.Object
  └ java.lang.Enum<State.Value>
      └ com.google.gdata.data.projecthosting.State.Value
```

**All Implemented Interfaces:**
java.io.Serializable, java.lang.Comparable<State.Value>

**Enclosing class:**
State

---

```
public static enum State.Value
extends java.lang.Enum<State.Value>
```

Value.

```
function process(state, event) {
    switch (state) {
        case 'start':
            if (event === 'SUBMIT') {
                return 'loading';
            }
            break;
        case 'loading':
            if (event === 'RESOLVE') {
                return 'success';
            } else if (event === 'REJECT') {
                return 'error';
            }
            break;
        case 'success':
            // Accept no further events
            break;
        case 'error':
            if (event === 'SUBMIT') {
                return 'loading';
            }
            break;
        default:
            // This should never occur
            return undefined;
    }
}
```

- Easy
- Runtime state management
- Illegal states/events/transitions must be handled
- May require Union/Option/... for state-local data

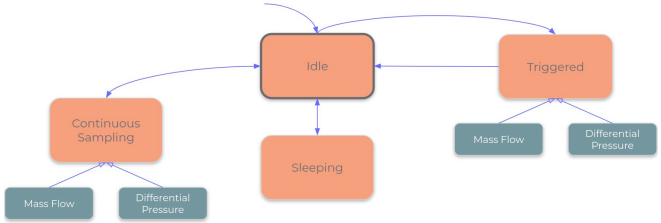
https://24ways.org/2018/state-machines-in-user-interfaces/

# Type System Approach

- States are types (with optional data)
- Transitions are functions
  - Signatures state transitions from type to type
  - Pass-by-move
  - Perform necessary I/O, communication, etc.

# Type System Approach

State the operation modes as types




```rust
pub struct IdleState;
pub struct TriggeredState;
pub struct SleepState;

pub struct ContinuousSamplingState<MeasurementType> {
    meas_type: PhantomData<MeasurementType>,
}

pub struct DifferentialPressure;
pub struct MassFlow;
```

https://github.com/barafael/sdp8xx-rs/blob/main/sdp8xx/src/states.rs

# Type System Approach

Define the possible state transitions as types

```rust
/// Transition from Idle to Sleep state
pub type IdleToSleep<I2C, D> = Sdp8xx<I2C, D, SleepState>;

/// Transition from Sleep to Idle state
pub type SleepToIdle<I2C, D> = Result<Sdp8xx<I2C, D, IdleState>, Error<I2C>>;
```

Why use `Result<T, E>`?: Some transitions are fallible.

https://github.com/barafael/sdp8xx-rs/blob/main/sdp8xx/src/states.rs

# Type System Approach

Implement the possible state transitions as **owning** methods

```rust
/// Transition from Idle to Sleep state
pub fn go_to_sleep(mut self) -> IdleToSleep<I2C, D>;

/// Transition from Sleep to Idle state
pub fn wake_up(mut self) -> SleepToIdle<I2C, D>;
```

https://github.com/barafael/sdp8xx-rs/blob/main/sdp8xx/src/lib.rs

# Type System Approach

Construction of initial state requires **ownership** of resources!

```
impl<I2C, D, E> Sdp8xx<I2C, D, IdleState>
    where /* snip */ {
    pub fn new(i2c: I2C, address: u8, delay: D) -> Self {
        Sdp8xx {
            i2c,
            address,
            delay,
            state: PhantomData::<IdleState>,
        }
    }
}
```

https://github.com/barafael/sdp8xx-rs/blob/0590f5801ce2d915399e6a2f4e505bf83d0f626a/sdp8xx/src/lib.rs#L155

# Type System Approach

Transition from Sleep state to Idle state:

```
impl<I2C, D, E> Sdp8xx<I2C, D, SleepState>
    where /* snip */ {
    pub fn wake_up(mut self) -> SleepToIdle<I2C, D> {
        self.send_command(Command::WakeUp)?;
        // some more work here ...
        Ok(Sdp8xx {
            i2c: self.i2c,
            address: self.address,
            delay: self.delay,
            state: PhantomData::<IdleState>,
        })
    }
}
```

https://github.com/barafael/sdp8xx-rs/blob/0590f5801ce2d915399e6a2f4e505bf83d0f626a/sdp8xx/src/lib.rs#L301

# Ingredients of Rust Typestate

| | |
|---|---|
| ▪ Move Semantics: | *State transitions* |
| ▪ Modularity: | *Hide boilerplate* |
| ▪ Visibility Rules: | *Prevent invalid use* |
| ▪ Zero-Sized Types: | *Make it cheap* |

# Advantages

The state transitions are **by-move**.

The new state **consumes** the previous one.

Memory cost:

```
core::mem::size_of::<ContinuousSamplingState<MassFlow>>() == 0
```

(ignoring runtime state, of course)

# Advantages

- Operations and data are available only in “their” state
- “Function call ordering” is enforced
- Impossible to misuse
- Autocomplete goodness
- Nice compiler errors

```
mismatched types
expected struct `SwitchMonitor<switch_monitor::Active>`
   found struct `SwitchMonitor<switch_monitor::Passive>`
```

# Example: Sensirion Pressure Sensor

```rust
#[test]
fn go_to_sleep() {
    let bytes: [u8; 2] = Command::EnterSleepMode.into();
    let expectations = [
        Transaction::write(0x25, bytes.into()),
        /* dummy data */ Transaction::write(0x25, vec![]),
        Transaction::write(0x25, vec![]).with_error(MockError::Io(ErrorKind::Other)),
        Transaction::write(0x25, vec![]),
    ];
    let mock = I2cMock::new(&expectations);
    let sdp = Sdp8xx::new(mock, 0x25, DelayMock);
    let sleeping = sdp.go_to_sleep().unwrap();
    let sdp = sleeping.wake_up().unwrap();
    sdp.release().done();
}
```

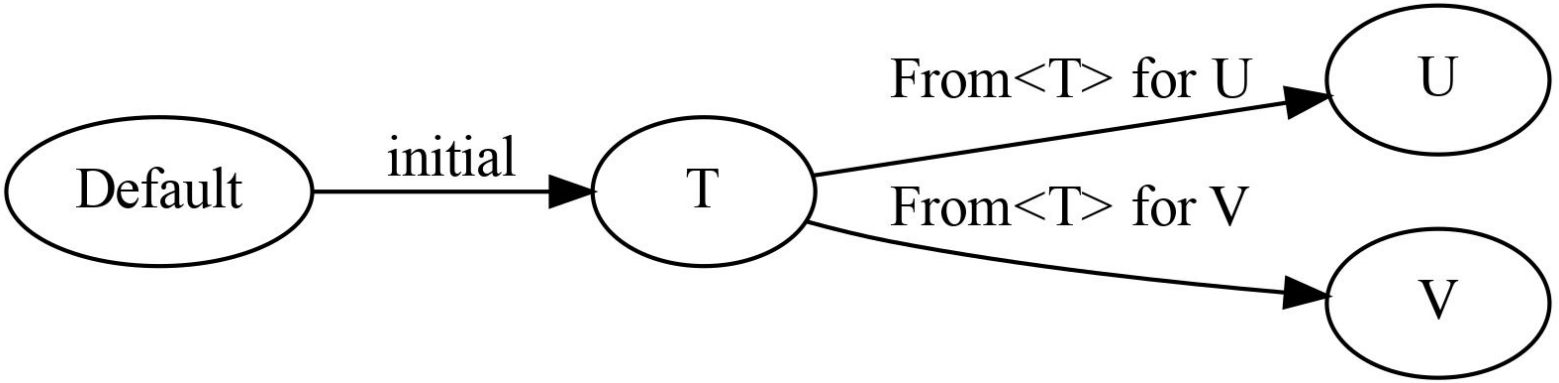
https://github.com/barafael/sdp8xx-rs/blob/0590f5801ce2d915399e6a2f4e505bf83d0f626a/sdp8xx/src/test.rs#L179

# Example: Hc-12 Wireless Module

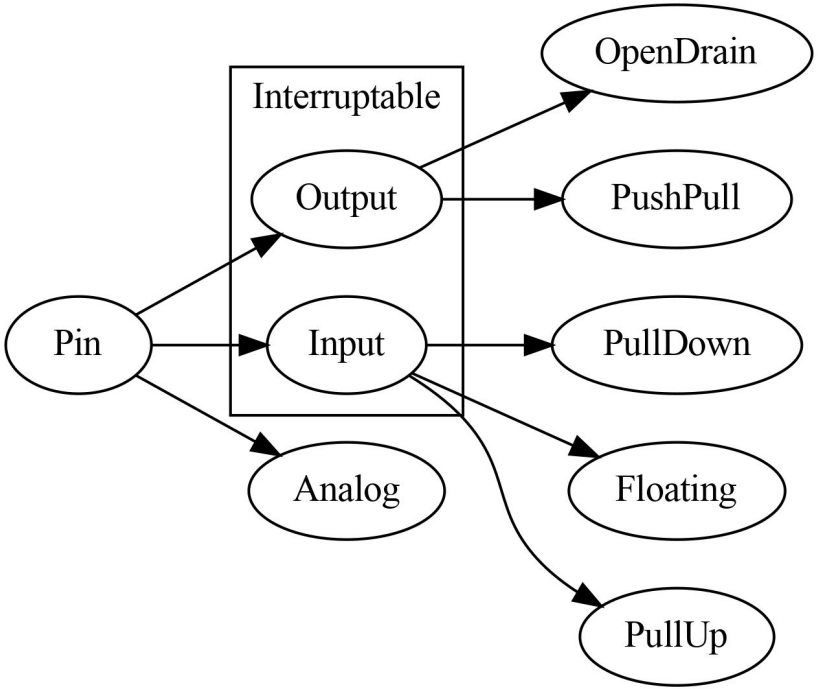
```
let serial = serial::Mock::new(&transactions);
let hc12 = Hc12::new(serial, set_pin, delay);
let mut hc12 = hc12.into_configuration_mode().unwrap();

assert!(hc12.is_ok());

assert_eq!(hc12.get_version(), b"VERSION-42\r\n");

let params = hc12.get_parameters().unwrap();
assert_eq!(...);

let mut hc12 = hc12.into_normal_mode().unwrap();

hc12.write_buffer(b"some data bla bla\r\n").unwrap();
```

https://github.com/barafael/hc12-at-rs/blob/acb27697977134660d549a12372ab3233f69abfa/hc12-at/src/hc12/test.rs#L62

# Example: Default and From Traits



# Example: GPIO Pins in stm32f4 HAL




https://github.com/stm32-rs/stm32f4xx-hal/blob/master/src/gpio/convert.rs

# However: Deep Type State

```
let tx: TxMode<
    embedded_nrf24l01::NRF24L01<
        Infallible,
        gpio::gpioa::PA8<Output<PushPull>>,
        stm32f0xx_hal::gpio::gpioa::PA10<Output<PushPull>>,
        Spi<
            stm32::SPI1,
            gpio::gpioa::PA5<Alternate<gpio::AF0>>,
            gpio::gpioa::PA6<Alternate<gpio::AF0>>,
            gpio::gpioa::PA7<Alternate<gpio::AF0>>,
        >,
    >,
> = nrf24.tx().unwrap();
```

# However: Deep Type State

```
let tx: TxMode<
    embedded_nrf24l01::NRF24L01<
        Infallible,
        gpio::gpioa::PA8<Output<PushPull>>,
        stm32f0xx_hal::gpio::gpioa::PA10<Output<PushPull>>,
        Spi<
            stm32::SPI1,
            gpio::gpioa::PA5<Alternate<gpio::AF0>>,
            gpio::gpioa::PA6<Alternate<gpio::AF0>>,
            gpio::gpioa::PA7<Alternate<gpio::AF0>>,
        >,
    >,
> = nrf24.tx().unwrap();
```

Type Inference helps a little

# However: Deep Type State

```
let tx: NrfModuleTx<
        Irq,
        CsPin,
        Spi6,
    > = nrf24.tx().unwrap();
```

Type aliases may help some more

https://github.com/rustype/typestate-rs

# Crate shoutout: typestate-rs

typestate-rs does all that behind the curtains:

```rust
#[typestate]
mod traffic_light {
    #[automaton]
    pub struct TrafficLight {
        pub cycles: u64,
    }

    #[state] pub struct Green;
    #[state] pub struct Yellow;
    #[state] pub struct Red;
}
```

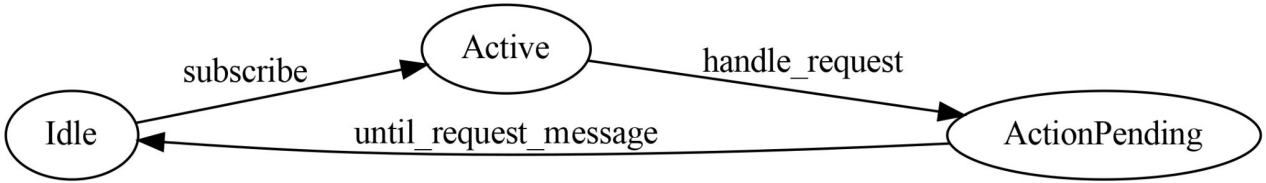
Now also generates dot files!

# Typestate + Dot = ♡

generate/manually create `.dot` file describing your state machine:

```
digraph G {
    rankdir="LR"

    Idle -> Active [label = "subscribe"]
    Active -> ActionPending [label = "handle_request"]
    ActionPending -> Idle [label = "until_request_message"]
}
```

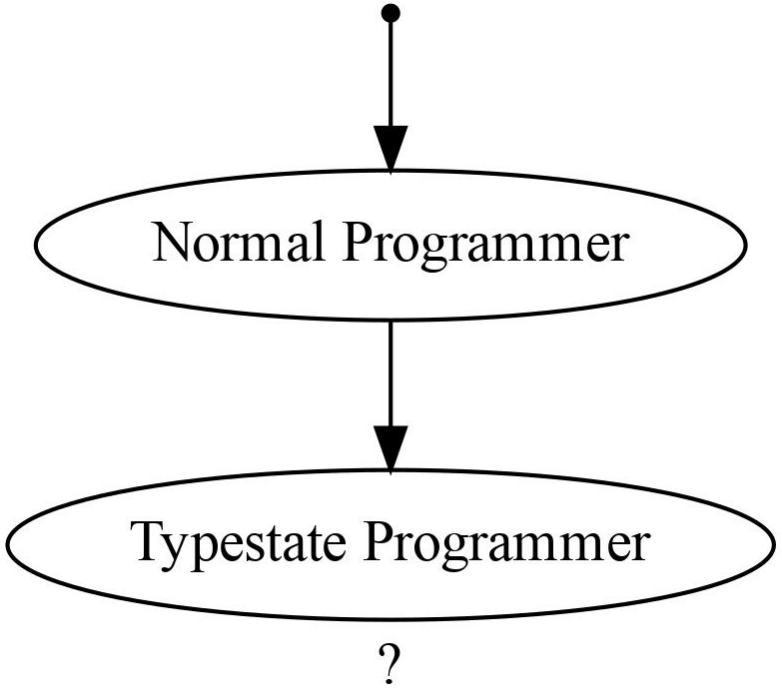
# Typestate + Dot = ♡



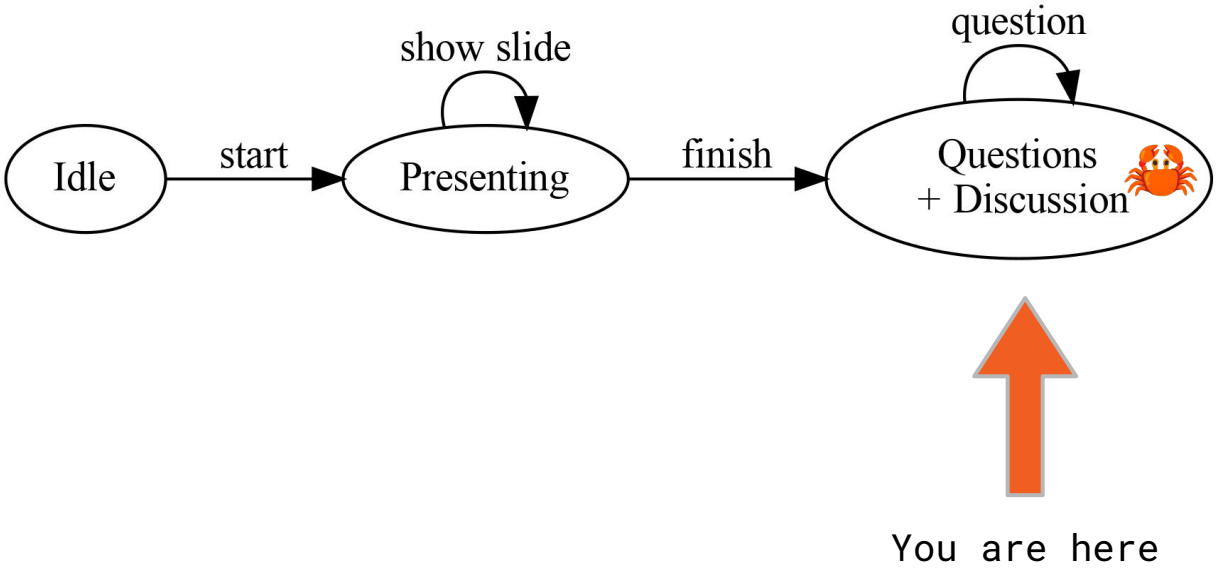
# Takeaway

When faced with finite state machine, consider typing it out :)

“I know typestate programming so to me everything is a state machine”



# Thanks :)


show slide
question
Idle
start
Presenting
finish
Questions
+ Discussion
You are here