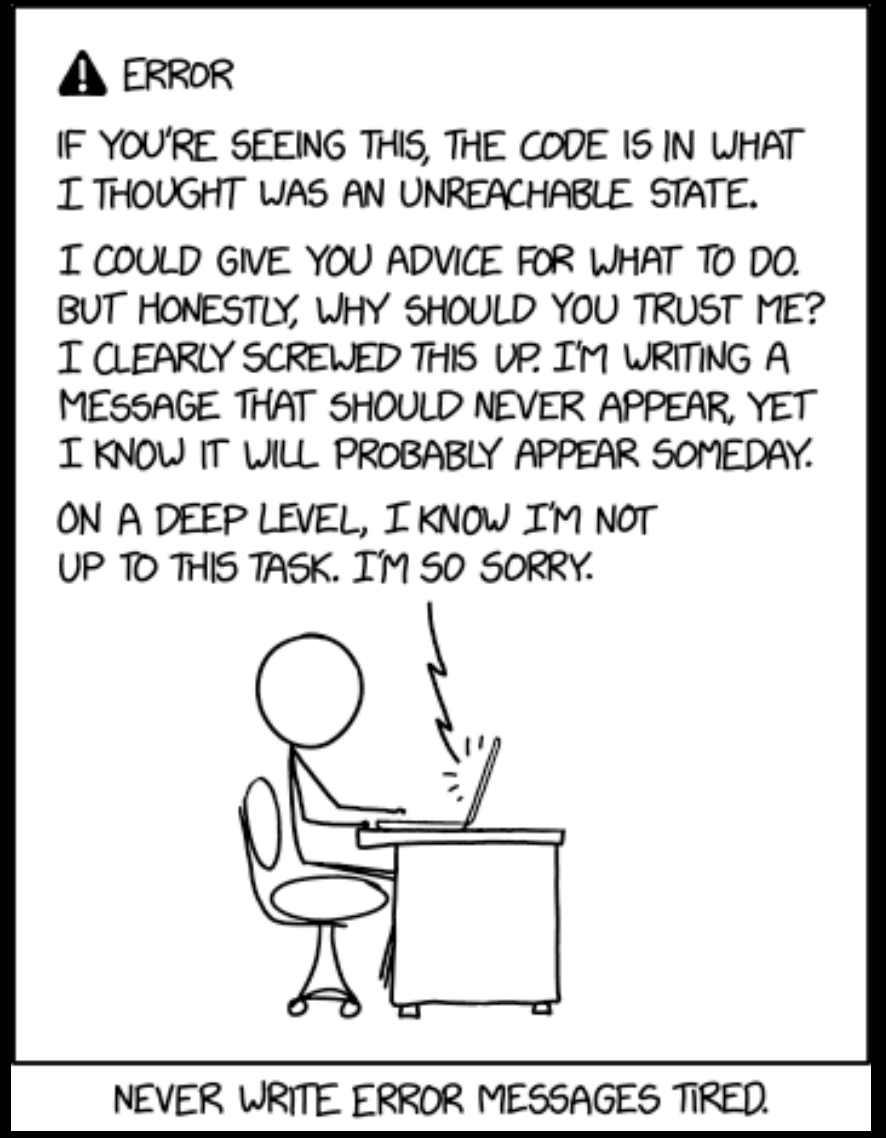


https://xkcd.com/2200/

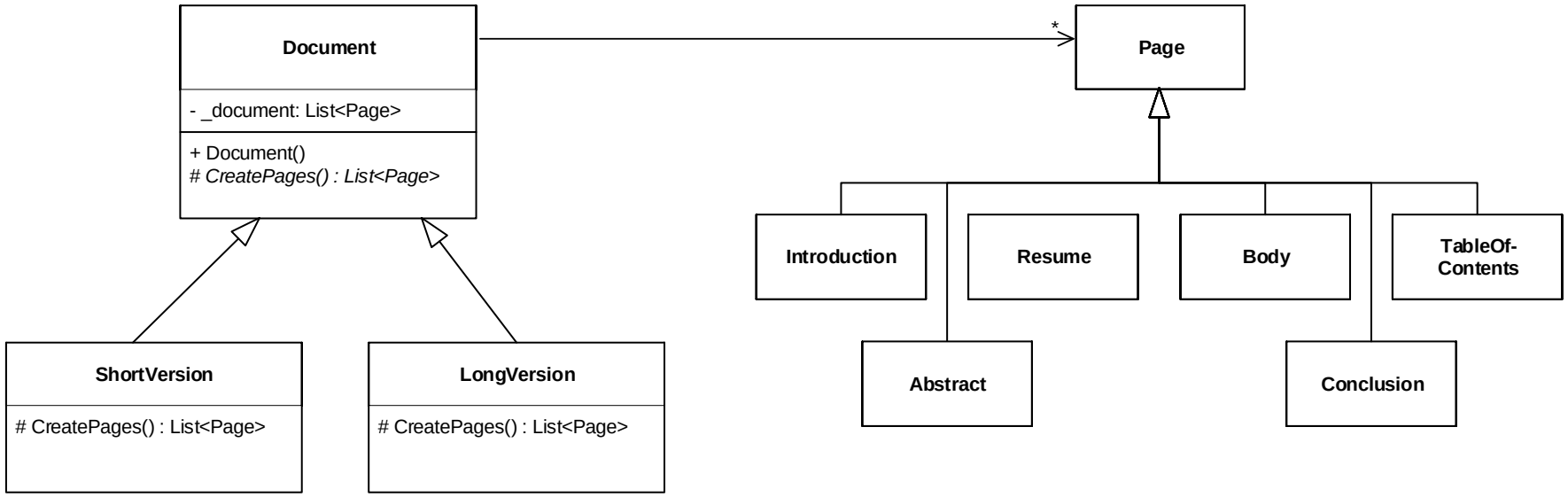
Software design patterns

State Machines

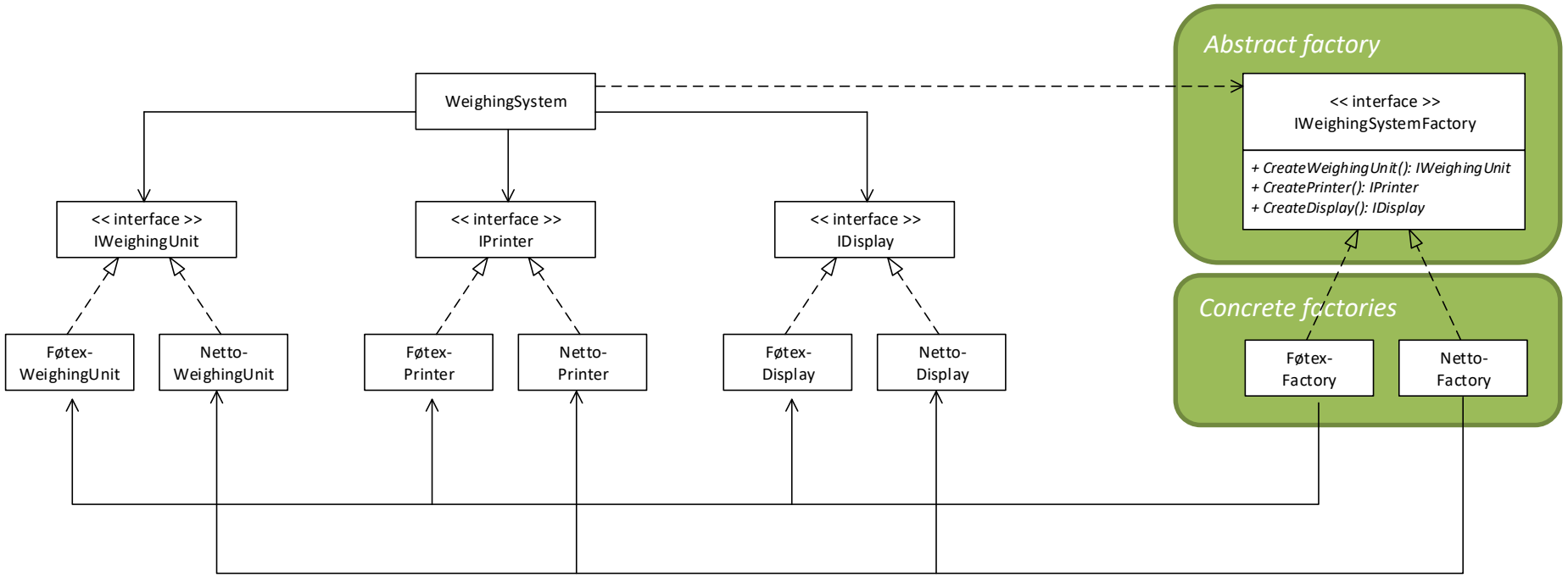
version: 1.0.2

AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

Previously… Factory Method



Previously… Abstract Factory



Agenda

- State machines
- State machine implementations
  - Switch-case
  - Table based
  - **GoF state pattern**

State Machines

State machines

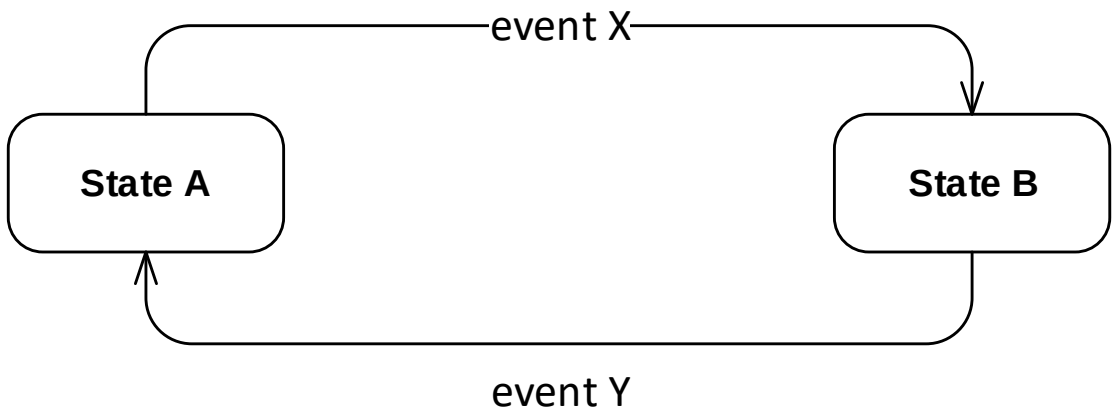
- A state machine (STM) is a model of a special kind of system.

- At any time, the STM is in one of a finite set of *states*
  - A light switch: { ON | OFF }
  - A process: {READY | RUNNING | BLOCKED }

- The STM can *transition* between states when *events* occur

State machines in UML

Simple UML STM
States, transitions, events

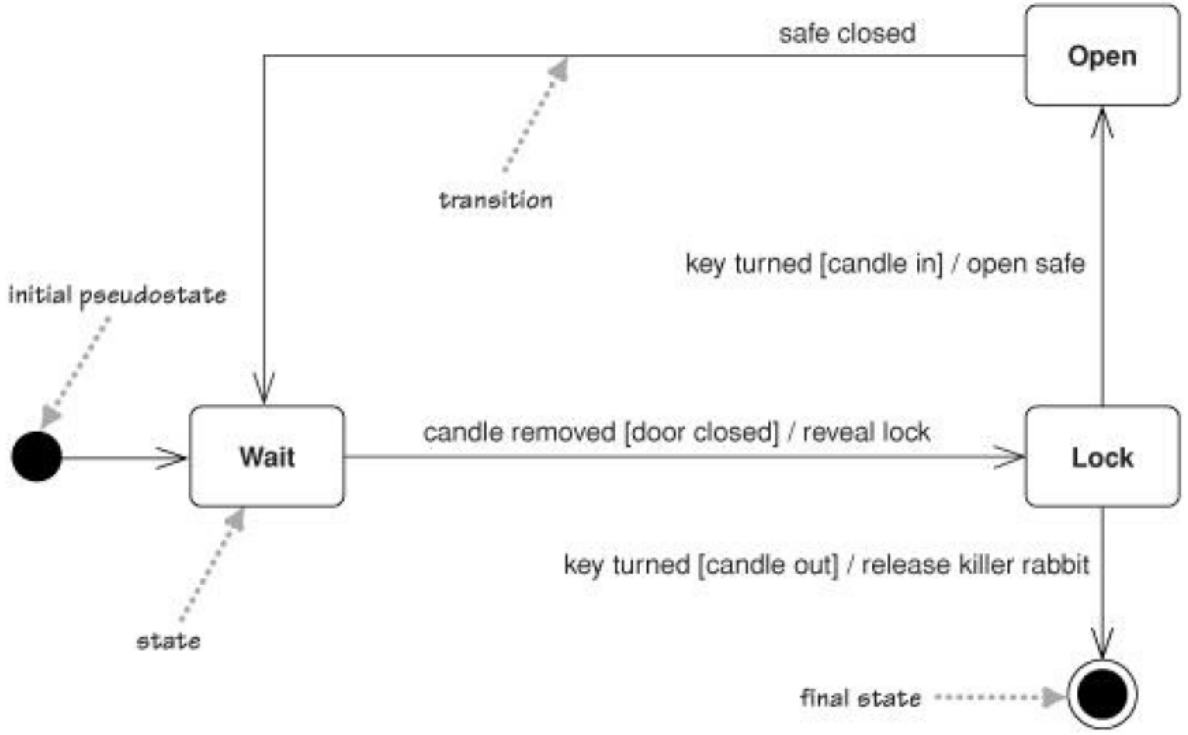


transition:
trigger-signature [guard] / activity

Things not covered yet:

- Nested states
- Orthogonal states

Example state diagram



Whenever people write about state machines, the examples are inevitably cruise controls or vending machines.

As I'm a little bored with them, I decided to use a controller for a secret panel in a Gothic castle. In this castle, I want to keep my valuables in a safe that's hard to find.

So to reveal the lock to the safe, I have to remove a strategic candle from its holder, but this will reveal the lock only while the door is closed.

Once I can see the lock, I can insert my key to open the safe. For extra safety, I make sure that I can open the safe only if I replace the candle first. If a thief neglects this precaution, I'll unleash a nasty monster to devour him.

Source: Martin Fowler – UML Distilled 3rd edition

State Machine Implementations

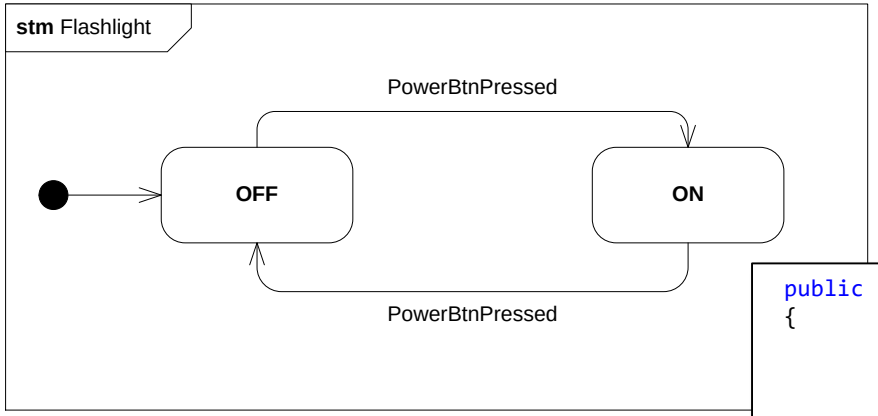
State machines

State machine implementations

- There are three common implementations of a state machine:
  1. Switch-case
  2. State/event tables
  3. GoF State Pattern

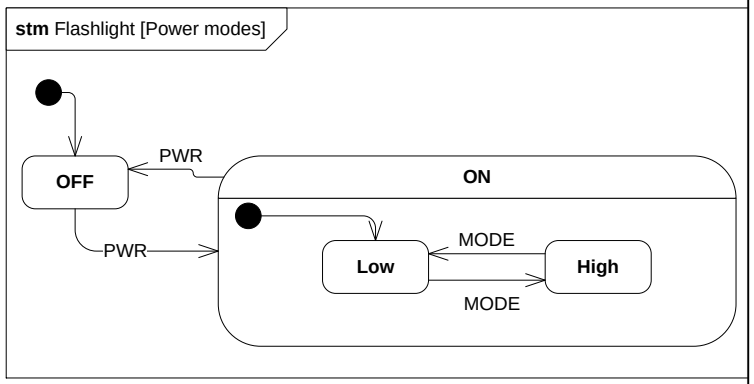
- Each implementation maps the STM to code
- Each has advantages and drawbacks

Switch/case implementation



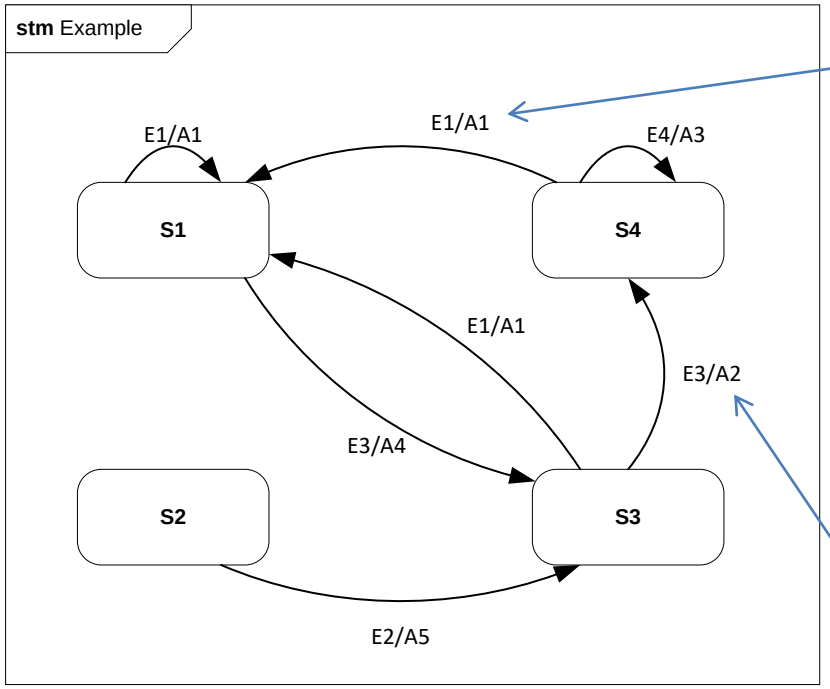

```
public class FlashLight
{
        public enum FlashLightEvent { PowerBtnPressed }
        enum FlashLightState  { On, Off }

        private FlashLightState _currentState;

        public FlashLight()
        {
            _currentState = FlashLightState.Off;
        }

        public void HandleEvent(FlashLightEvent evt)
        {
            switch (_currentState)
            {
                case FlashLightState.On:
                    _currentState = FlashLightState.Off;
                    break;

                case FlashLightState.Off:
                    _currentState = FlashLightState.On;
                    break;
            }
        }
    }
```

Switch/case implementation




```
class FlashLightWithModes
    {
        public enum FlashLightEvent { PWR, MODE }
        enum FlashLightState  { On, Off }
        enum PwrOnSubStates { Low, High }

        private FlashLightState _currentState;
        private PwrOnSubStates _currentPwrOnSubState;

        public FlashLightWithModes()
        {
            _currentState = FlashLightState.Off;
            _currentPwrOnSubState = PwrOnSubStates.Low;
        }

        public void HandleEvent(FlashLightEvent evt)
        {
            switch (_currentState)
            {
                case FlashLightState.Off:
                    switch (evt)
                    {
                        case FlashLightEvent.PWR:
                            _currentState = FlashLightState.On;
                            _currentPwrOnSubState = PwrOnSubStates.Low;
                            break;
                    }
                    break;

                case FlashLightState.On:
                    switch (evt)
                    {
                        case FlashLightEvent.PWR:
                            _currentState = FlashLightState.Off;
                            break;

                        case FlashLightEvent.MODE:
                            switch(_currentPwrOnSubState)
                            {
                                case PwrOnSubStates.Low:
                                    _currentPwrOnSubState = PwrOnSubStates.High;
                                    break;

                                case PwrOnSubStates.High:
                                    _currentPwrOnSubState = PwrOnSubStates.Low;
                                    break;
                            }
                            break;
                    }
                    break;
            }
        }
    }
```

Table-based implementation



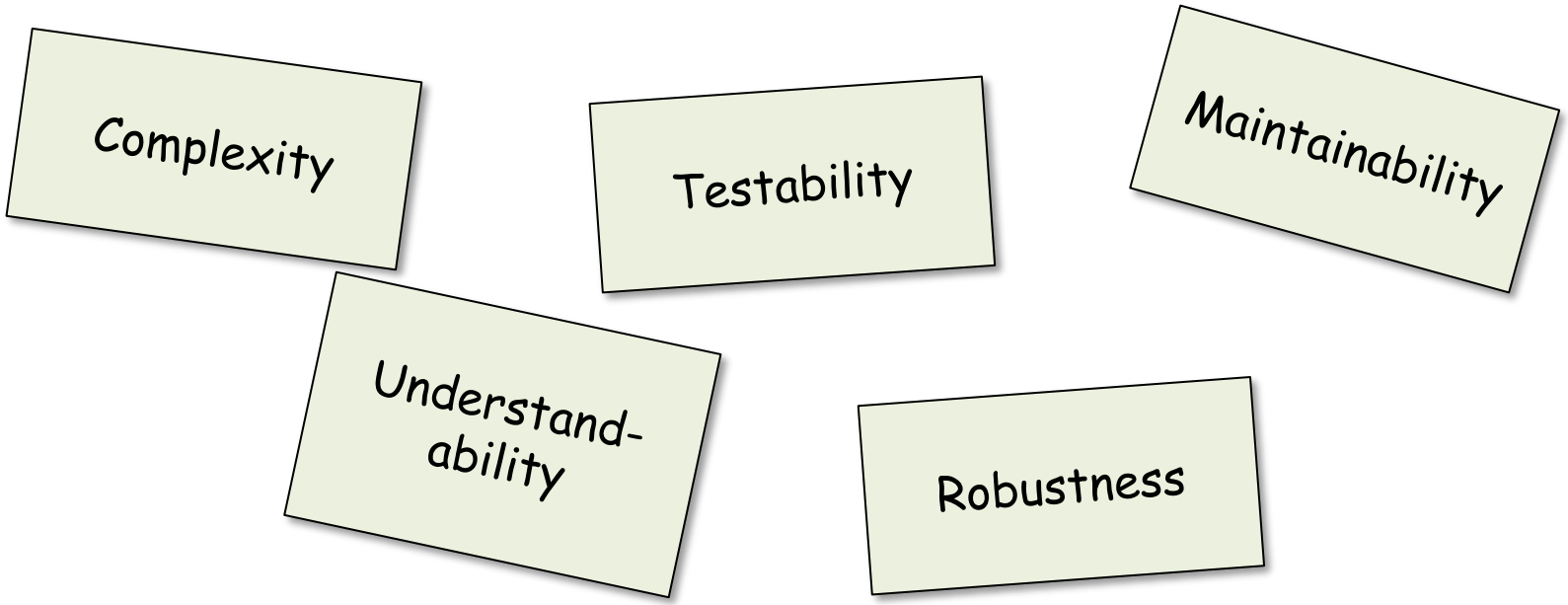
In state *S4*, event *E1* will cause action *A1* and a transition to state *S1*

State/Event	E1	E2	E3	E4
S1	A1/S1	-	A4/S3	-
S2	-	A5/S3	-	-
S3	A1/S1	-	A2/S4	-
S4	A1/S1	-	-	A3/S4

In state *S3*, event *E3* will cause action *A2* and a transition to state *S4*

Pros and cons

- What are the pros and cons of switch/case and table-based implementations in terms of…



- It’s time for change!

GoF State Pattern details

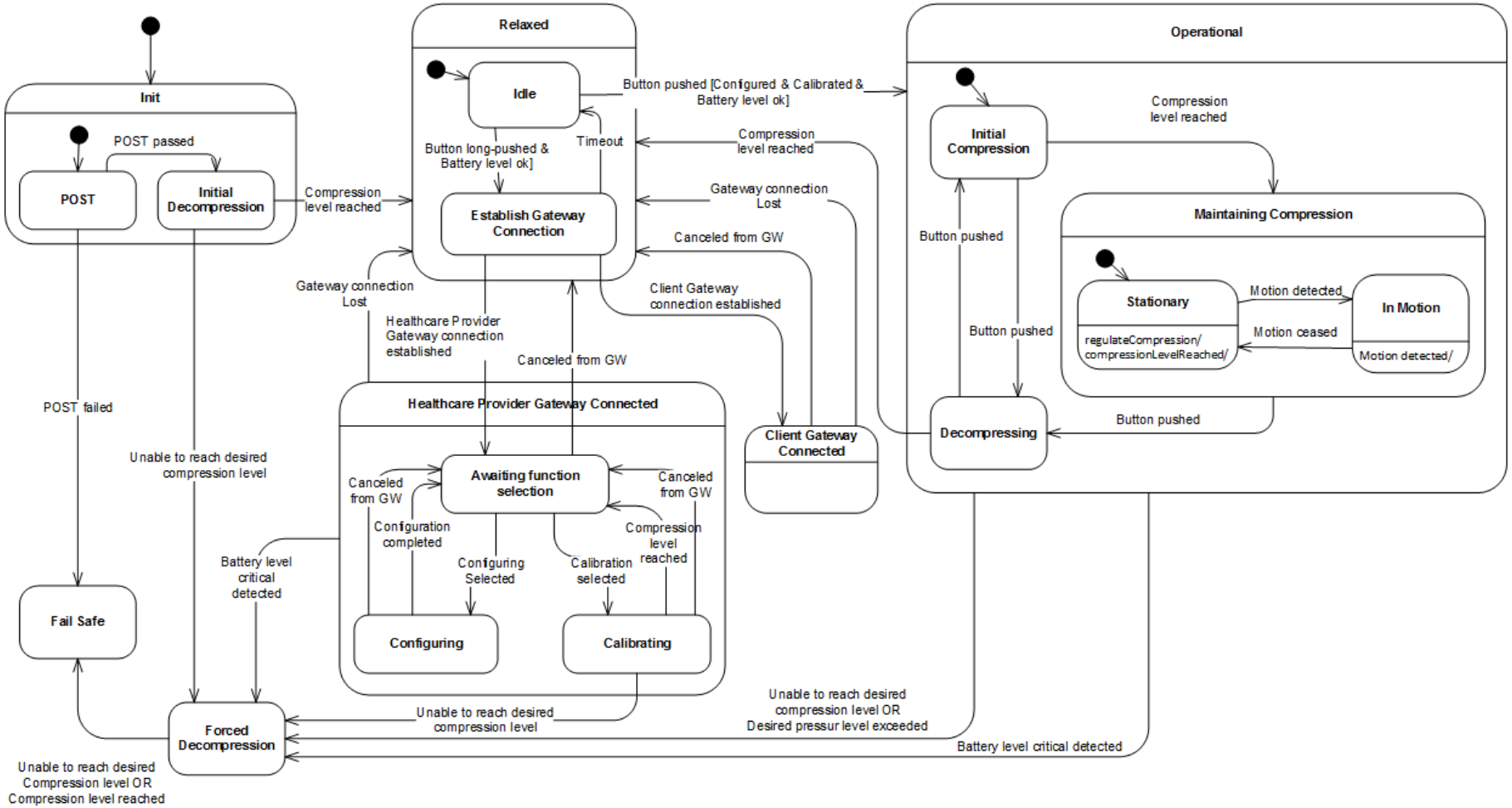
State machines

GoF State pattern

- When STMs get complex, switch-case and table implementations become very messy and *extremely* difficult to test

- Time to introduce the GoF State Pattern!

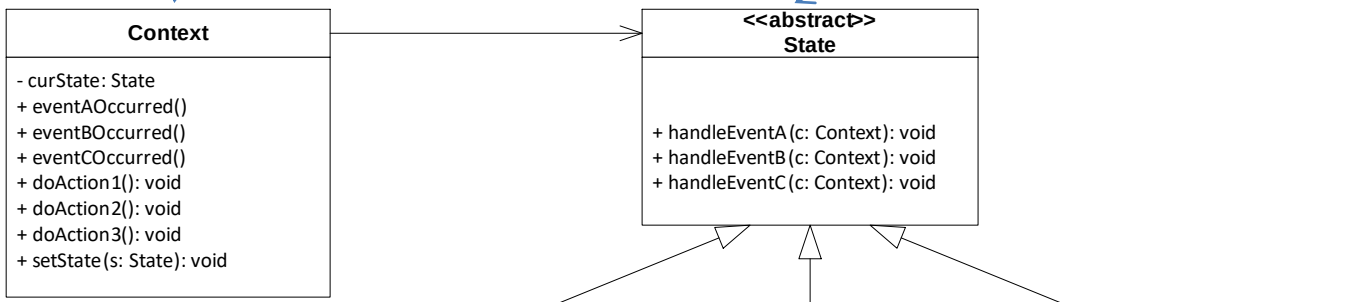
...but do STMs really get that complex?



STM implementations: GoF State

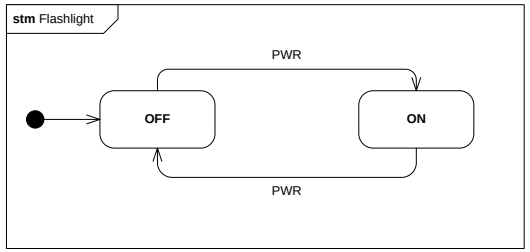
`Context` is the "owner" of the state machine. Contains *event handlers* and *actions*

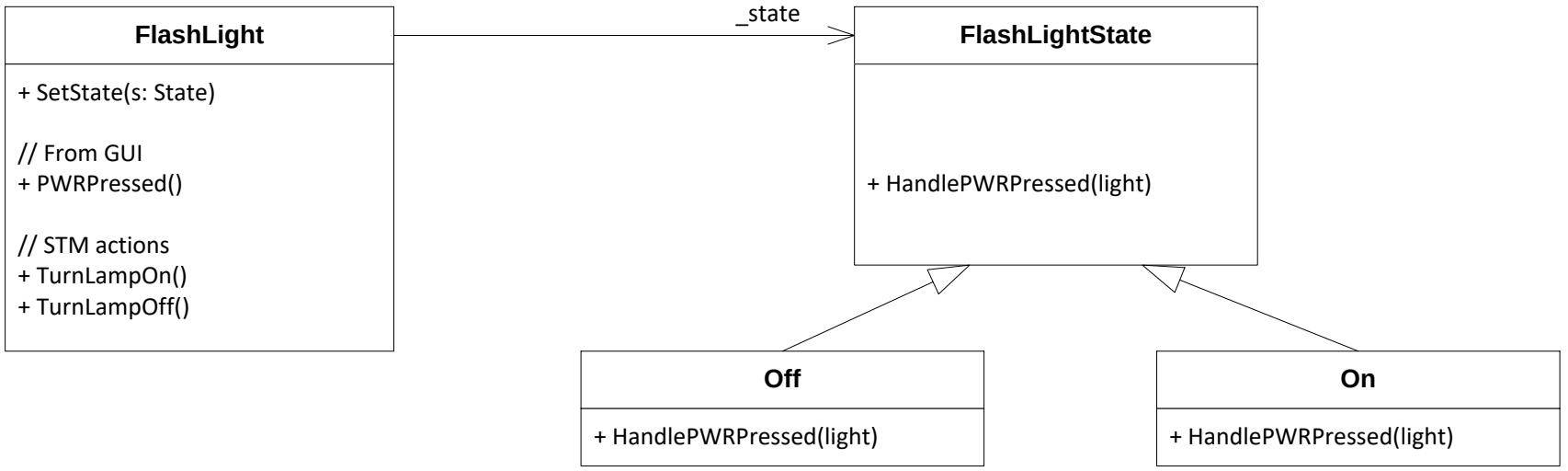
Top-level state, usually an abstract class with empty default impl. of all event handlers (why?)



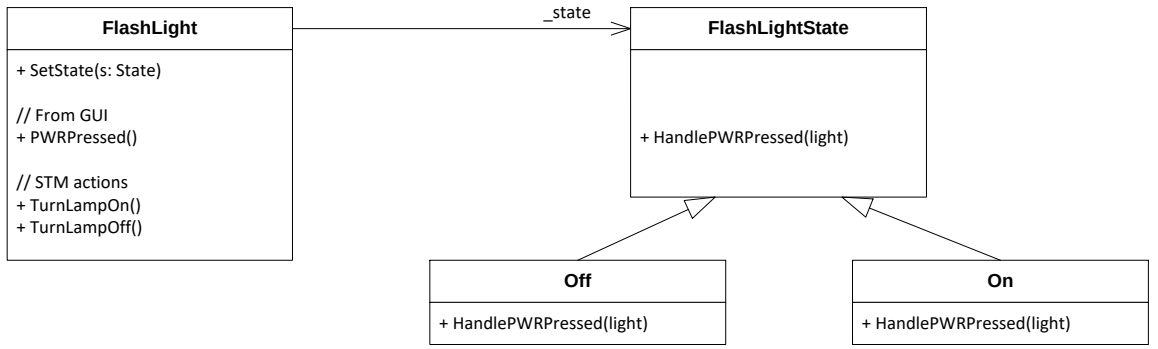
Implementation of concrete states and the event handlers for each state.

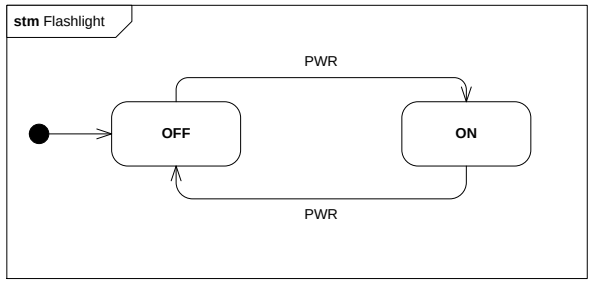
Flashlight example: Structure

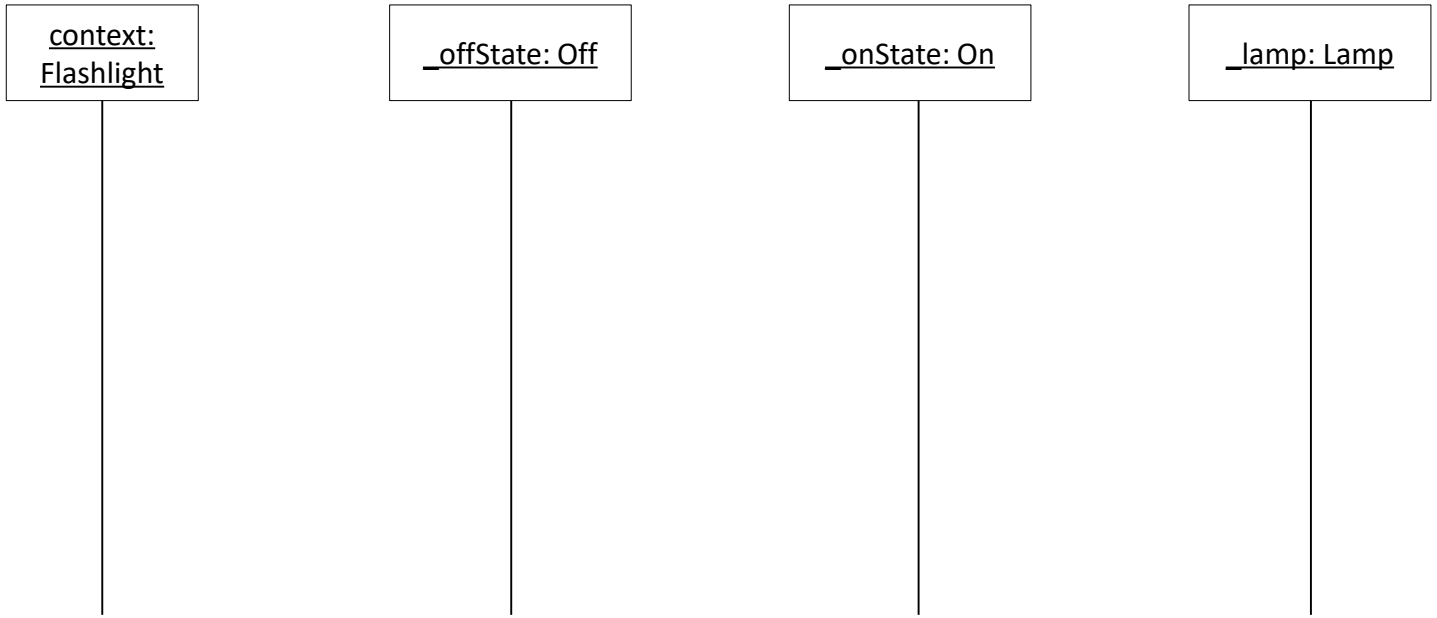




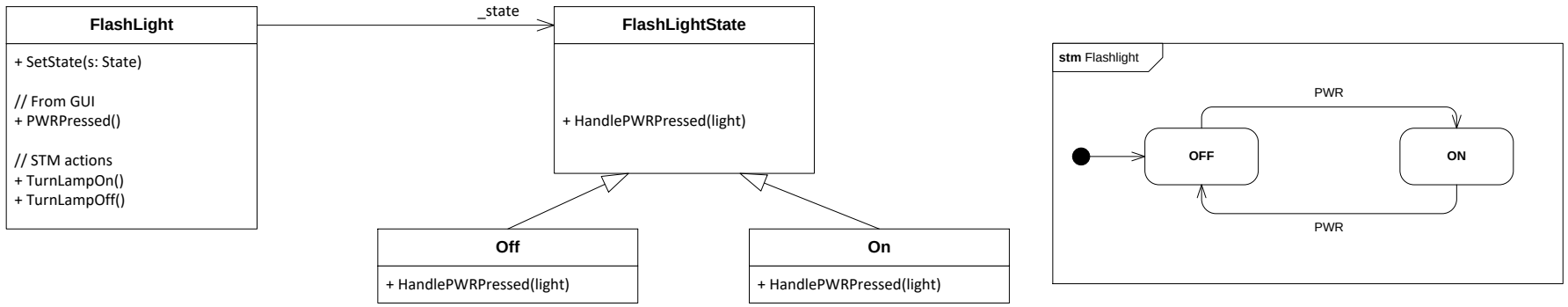
Flashlight example - transitions

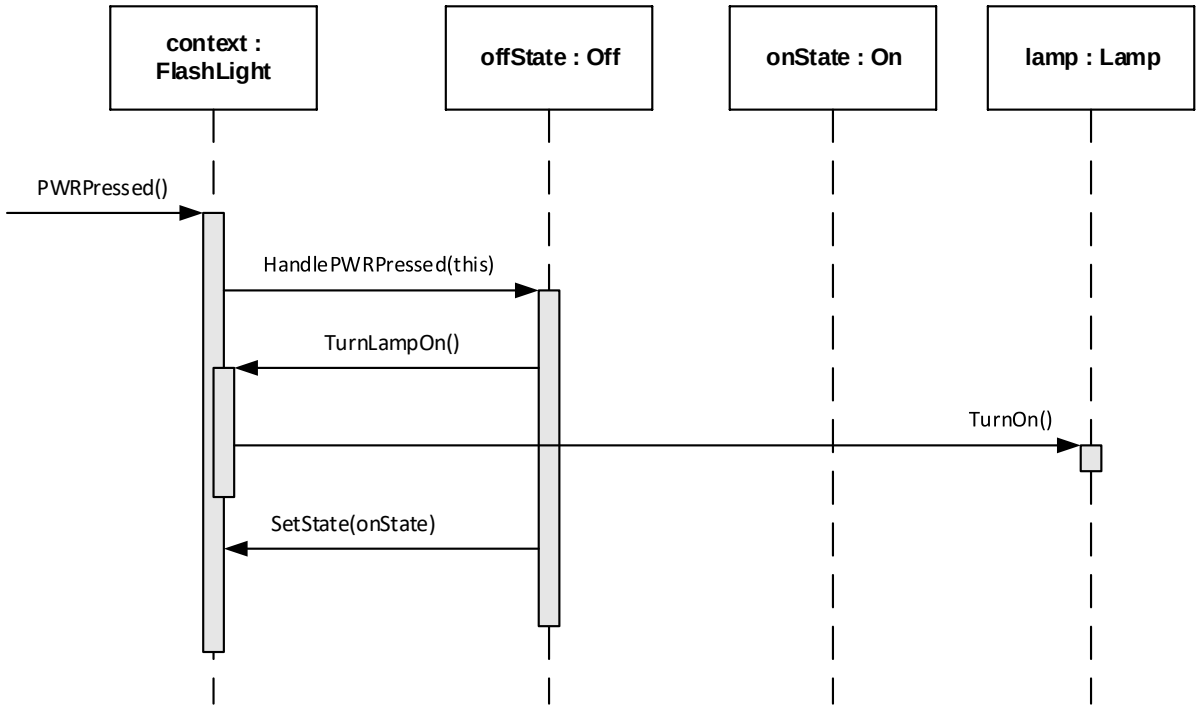






Flashlight example - transitions





GoF State – the upshots

- In the GoF State pattern, each state is implemented as a subclass of a common abstract state class.
- At any time, the *context* references exactly one state object – the context's *current state*
- The context delegates all of its state-dependent behavior to this state object
  - We say that the context's behavior is *externalized* in the states
- The state classes should be kept stateless so they can be singleton implementations

GoF State
– event reception and handling

- When the context receives an event it immediately forwards it to its state object.
- If an action shall be performed, the state object calls back into the context to perform the action.
- If a state change shall be performed, the state object will call back to the context to set the context's *new* state object
  - Important! The context is **not** responsible for setting the new state – the current state is!

Your turn!

- State patterns exercise 1 (intro), question 1-4