Lecture 21: State Pattern
IN628: Programming 4
Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

State Pattern: GoF

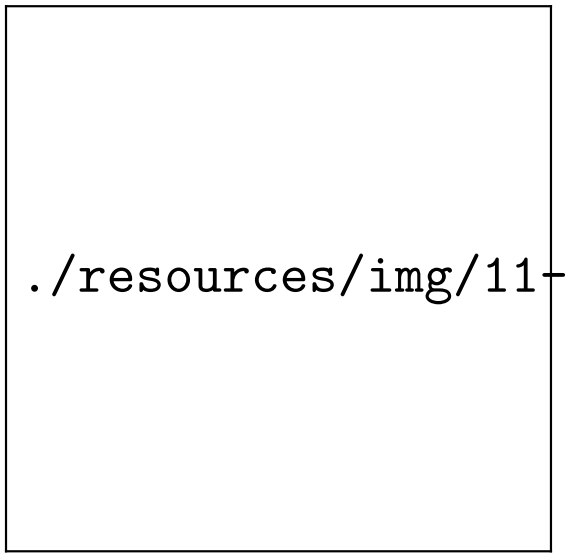
- GoF definition & UML

State Pattern: Definition

- Behavourial pattern
- Allows an object to alter its behaviour when its internal state changes
- Close to the concept of finite-state machines
- Can be interpreted as a strategy pattern
- Used to encapsulate varying behaviour for the same object
- Cleaner way for an object to change its behavior at runtime

State Pattern: Problem

- Writing for publication
- Sent in for review by the author or co-authors
- Reviewed & approved by the moderator
- Review hasn't passed and returned back to the author/co-authors
- Publication has expired
- Published by the moderator



State Pattern: Solution

- Publication class
- State interface class
- Draft (state) class

State Pattern: Real World Analogy

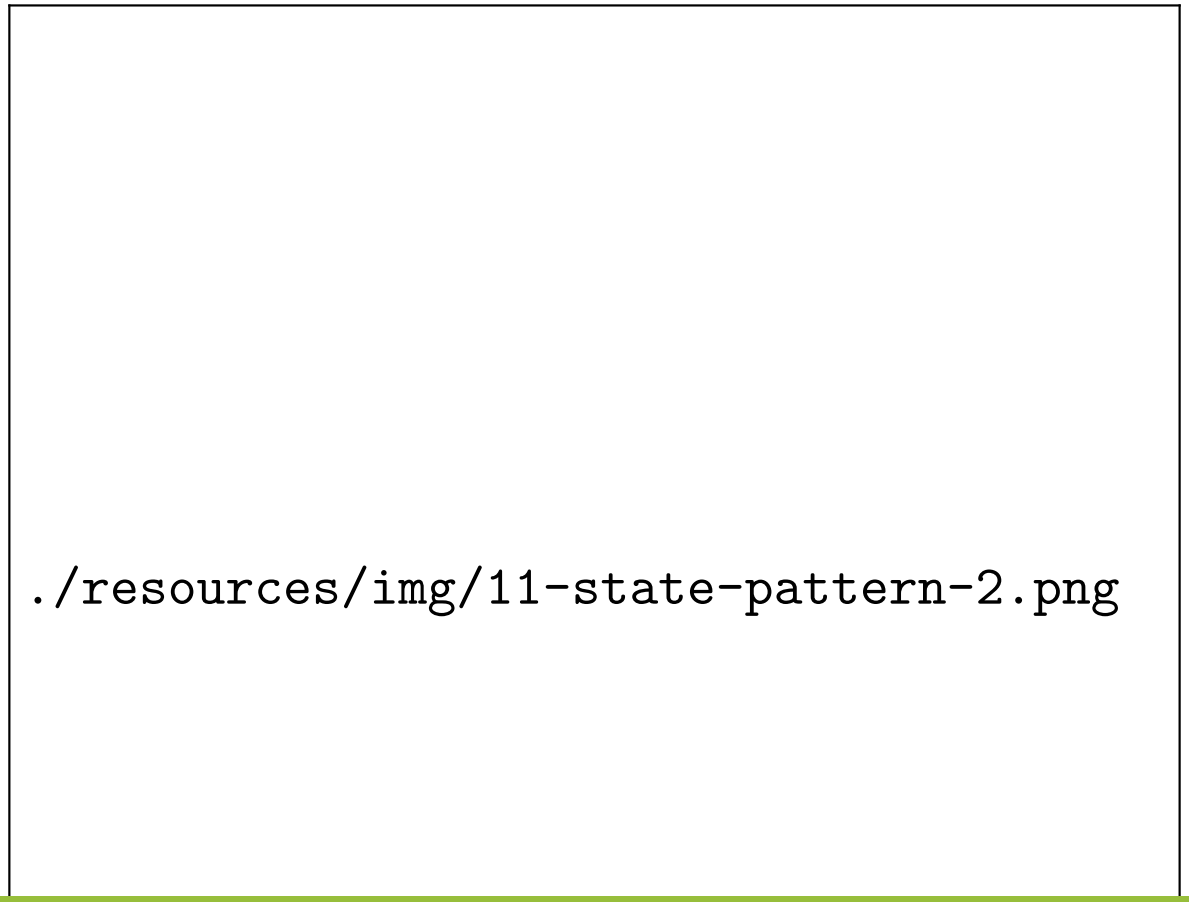
- Purchasing an item from a vending machine
- When money is deposited & an item is selected, the vending machine will either:
  - Return the item & no change
  - Return the item & change
  - Return no item due to an insufficient amount deposited
  - Return no item due to an inventory depletion

State Pattern: UML

- Consider the following UML diagram:

State Pattern: UML

- Context class
- Doesn't implement state-specific behaviour directly
- Refers to the state interface class for performing state-specific behaviour
- Delegates state-specific behavior to different state objects

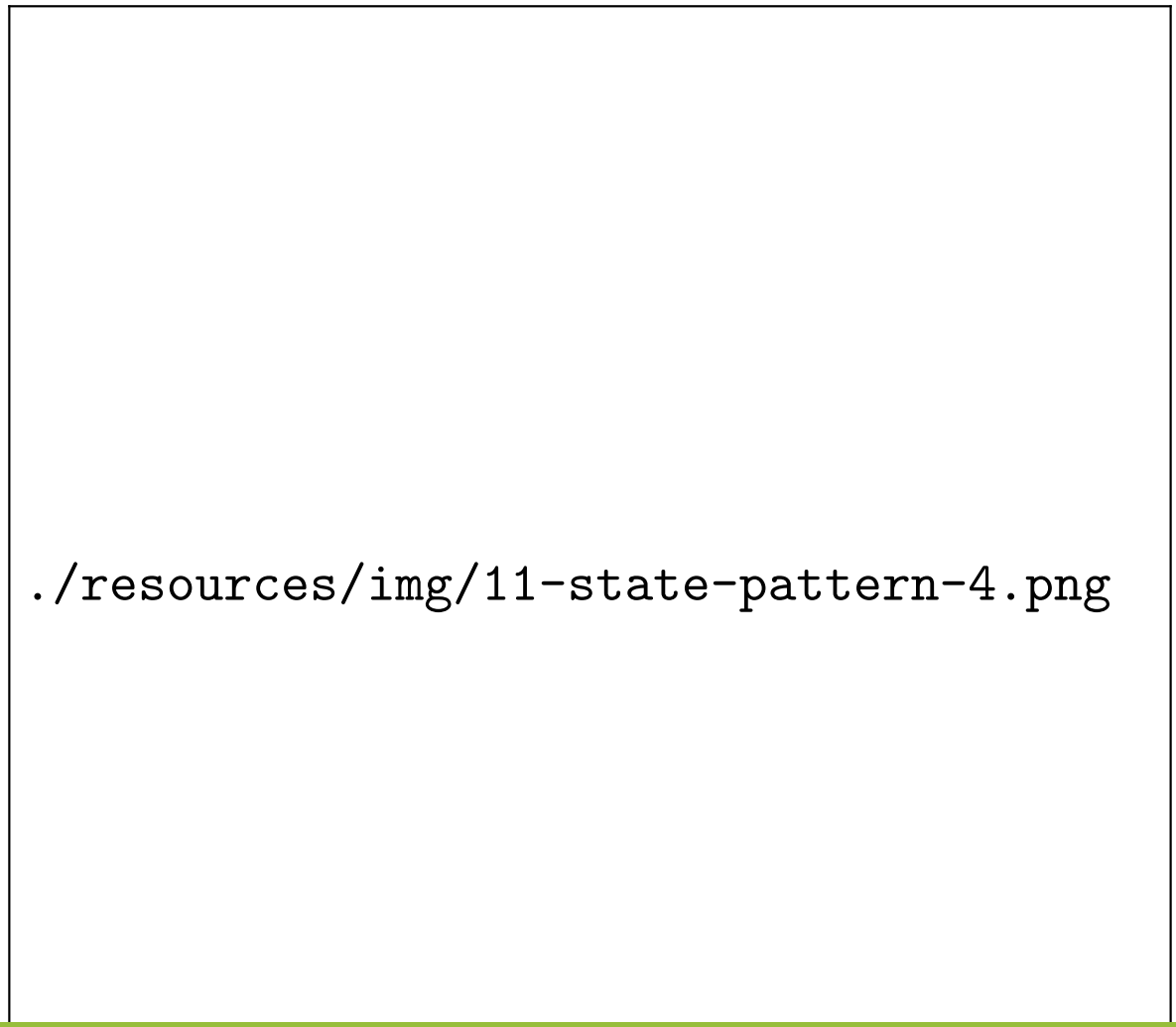


State Pattern: UML

- State interface class

State Pattern: UML

- Concrete state classes
- Implement the state interface class
- Encapsulate the state-specific behaviour for each state



State Pattern: Implementation

```
from abc import ABC, abstractmethod

class State(ABC):
    @abstractmethod
    def write_name(self, state_context, name):
        pass

class LowercaseState(State):
    def write_name(self, state_context, name):
        print(name.lower())
        state_context.state = UppercaseState()

class UppercaseState(State):
    def write_name(self, state_context, name):
        print(name.upper())
        state_context.state = LowercaseState()
```

State Pattern: Implementation

```
class StateContext:
    def __init__(self):
        self.__state = UppercaseState()

    @property
    def state(self):
        return self.__state

    @state.setter
    def state(self, state):
        self.__state = state

    def request(self, name):
        self.__state.write_name(self, name)

def main():
    state_context = StateContext()
    state_context.request('Monday')
    state_context.request('Tuesday')
    state_context.request('Wednesday')
    state_context.request('Thursday')
    state_context.request('Friday')
    state_context.request('Saturday')
    state_context.request('Sunday')

if __name__ == '__main__':
    main()
```

State Pattern: Pros

- Particular states are organised into separate class
- New states can be introduced without having to change the existing state classes or the context
- By eliminating large state machine conditionals, the context code is simplified

State Pattern: Cons

- If a state machine has only a few states or rarely changes, the state pattern can be an overkill

Practical

- Series of tasks covering today's lecture
- Worth 1% of your final mark for the Programming 4 course
- Deadline: Friday, 12 June at 5pm