OTAGO
POLYTECHNIC
Te Kura Matatini ki Otago

Lecture 11: State Pattern
IN710: Object-Oriented Systems
Development
Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

Tuesday, 24 March

LECTURE 10: BUILDER PATTERN RECAP

- ▶ Design pattern 06: builder pattern
  - ▶ Definition
  - ▶ Problem/solution
  - ▶ UML & implementation
  - ▶ Applicability
  - ▶ Pros & cons

LECTURE 11: STATE PATTERN TOPICS

- ► Design pattern 07: state pattern
    - ► Definition
    - ► Problem/solution
    - ► Real world analogy
    - ► UML & implementation
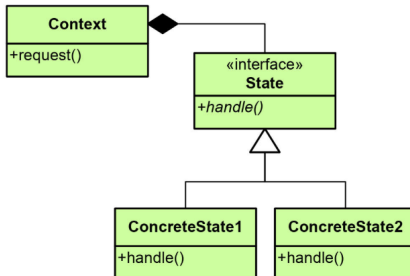   - ► Pros & cons

STATE PATTERN: GOF

▶ GoF definition & UML

State

Type: Behavioral

What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

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
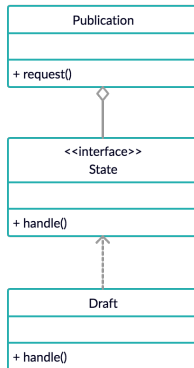- ▶ Cleaner way for an object to change its behavior at runtime

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
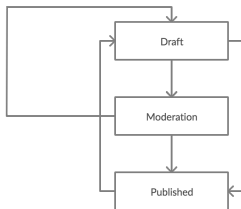
STATE PATTERN: SOLUTION

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

```
┌─────────────────────┐
│     Publication     │
├─────────────────────┤
│                     │
├─────────────────────┤
│ + request()         │
└─────────────────────┘
          ◇
          │
┌─────────────────────┐
│    <<interface>>    │
│       State         │
├─────────────────────┤
│                     │
├─────────────────────┤
│ + handle()          │
└─────────────────────┘
          △
          ┊
┌─────────────────────┐
│       Draft         │
├─────────────────────┤
│                     │
├─────────────────────┤
│ + handle()          │
└─────────────────────┘
```
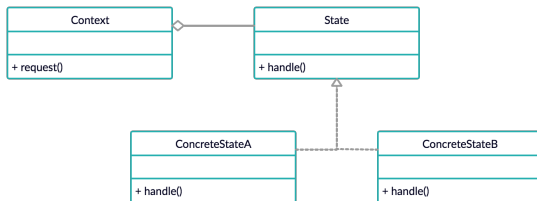
State Pattern: Real World Analogy

- ▶ 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
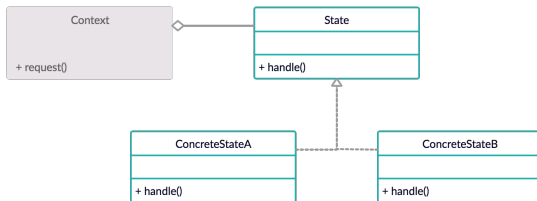 - ▶ Return no item due to an inventory depletion

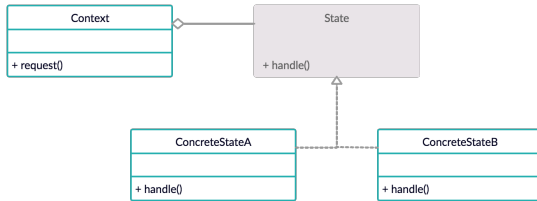# STATE PATTERN: UML

▶ 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
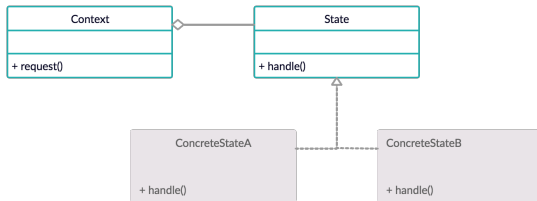- ► Delegates state-specific behavior to different state objects

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

```python
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

```python
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 Object-Oriented Systems Development course
- ► Deadline: Friday, 12 June at 5pm

Exam 02 Results

- Average 41 out of 50 or 4.93% out of 6%

LECTURE 12: FLYWEIGHT PATTERN TOPICS

- Design pattern 08: flyweight pattern
  - Definition
  - Problem/solution
  - UML
  - Immutability
  - Implementation
  - Pros & cons