Chapter 12 - The State Pattern

September 18, 2021

0.1 Overview

- `OOP` focuses on maintaining the states of objects that interact with each other
- a very handy tool to model `state transitions` when solving many problems is known as a `finite-state machine` or `state machine`

- a state machine is an abstract machine that has two key components, that is `state` and `transitions`
- a state is the current (active) status of a system

- for example if we have a radio receiver, two possible states for it are to be tuned to `FM` or `AM`
- another state is for it to be switching from one `FM/AM` radio station to another
- a transition is a switch from one state to another
- a transition is initiated by triggering event or conditions
- usually, an action or set of actions is executed before or after a transition occurs

- a nice feature of state machines is that they can be represented as `graphs`, where each state is a node and each transition is an edge between two nodes
- state machines can be used to solve many kinds of problems, but non-computational and computational
- non-computational examples include vending-machines, elevators, traffic lights, combination locks, parking meters, and automated gas pumps
- computational examples include game programming and other categories of computer programming, hardware design, protocol desgin, and programming languages parsing

- the question remains, how are `state machines` related to `state design pattern`?
- it turns out that state pattern is nothing more than a state machine applied to a particular software engineering problem

0.2 Real-World Examples

- vending machines have different states and react differently depending on the amount of money that we insert
- depending on our selection and the money we insert, the machine can do the following
  - reject our selection because the product we requested is out of stock
  - deliver the product and give change

0.3 Use Cases

- all problems that can be solved using state machines are a good use case for the `State` pattern

- programming language compiler implementation or process model are good use cases for the state pattern
- lexical and syntactic analysis can use states to build abstract syntax trees

- event-driven systems are yet another example
- in event-driven systems, the transition from one state to aonther triggers an event/message
- many computer games use this technique
- for example, if a monster might move from the guard state to the attach statw when the main hero approaches it

0.4 Implementation

- our state machine should cover the different states of a process and the transitions between them
- the `State` design pattern is ususally implemented using a parent `State` class that contains the common functionality of all the states, and a number of concrete classes derived from `State`, where each derived class contains only the state-specific required functionality
- the `State` pattern focuses on implementing a state machine
- the core parts of a state machine are the states and transitions between the states
- it doesn't matter how those parts are implemented

- to avoid reinventing the wheel, we can make use of existing Python modules that not only help us create state machines, but also do it in a Pythonic way
- a module that we will use is `state_machine`

- lets start with the `Process` class
- each created process has its own state machine
- the first step to create a state machine using the `state_machine` module is to use the `@acts_as_state_machine` decorator
- we then define the states of our state machine
- this is a one-to-one mapping of what we see in the state diagram
- the only difference is that we should give a hint about the initial state of the state machine
- we do this by setting the `initial` attribute value to `True`
- next we are going to define the transitions
- in the `state_machine` module, a transition is an instance of the `Event` class
- we define the possible transitions using the arguments `from_states` and `to_state`

- each process has a name
- ofically, a process name needs to have much more information to be useful (for example, ID, priority, status, and so forth) but let's keep it simple to focus on the pattern

- transitions are not very useful if nothing happens when they occur
- the `state_machine` provides us with the `@before`/`@after` decorators that can be used to execute actions before or after a transition occurs, respectively
- you can imagine updating some objects withing the system or sending an email or notification to someone
- for the purpose of this example, the actions are limited to printing information about the state change of the process

[21]:

```python
'''
from state_machine import (State, Event, acts_as_state_machine,
 after, before, InvalidStateTransition)
'''

def acts_as_state_machine():
    pass

class State:
    def __init__(self, initial=True):
        self.initalized = initial



class Event:
    def __init__(self, from_states=True, to_state=False):
        self.From_states = from_states
        self.to_state = to_state

@acts_as_state_machine
class Process:
    created = State(initial=True)
    waiting = State()
    running = State()
    terminated = State()
    blocked = State()
    swapped_out_waiting = State()
    swapped_out_blocked = State()

    wait = Event(from_states=(created,
                              running,
                              blocked,
                              swapped_out_waiting),
                 to_state=waiting)

    run = Event(from_states=waiting, to_state=running)

    terminate = Event(from_states=running, to_state=terminated)

    block = Event(from_states=(running,
                               swapped_out_blocked),
                  to_state=blocked)

    swap_wait = Event(from_states=waiting,
                      to_state=swapped_out_waiting)

    swap_block = Event(from_states=blocked,
```

```python
                          to_state=swapped_out_blocked)

    def __init__(self, name):
        self.name = name

    @after('wait')
    def wait_info(self):
        print(f'{self.name} entered waiting mode')

    @after('run')
    def run_info(self):
        print(f'{self.name} is running')

    @before('terminate')
    def terminate_info(self):
        print(f'{self.name} terminated')

    @after('block')
    def block_info(self):
        print(f'{self.name} is blocked')

    @after('swap_wait')
    def swap_wait_info(self):
        print(f'{self.name} is swapped out and waiting')

    @after('swap_block')
    def swap_block_info(self):
        print(f'{self.name} is swapped out and blocked')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-21-1459bb4c4d5b> in <module>
     19
     20 @acts_as_state_machine
---> 21 class Process:
     22     created = State(initial=True)
     23     waiting = State()

<ipython-input-21-1459bb4c4d5b> in Process()
     52
     53     @after('wait')
---> 54     def wait_info(self):
     55         print(f'{self.name} entered waiting mode')
     56
```

```
TypeError: 'NoneType' object is not callable
```

- next we need the `transition()` function, which accepts three arguments
  - `process`, which is an instance of `Process`
  - `event`, which is an instance of `Event` (`wait`, `run`, `terminate` and so forth)
  - `event_name`, which is the name of the event
- and the name of the event is printed if something goes wrong when trying to execute an `event`

[22]:
```python
def transition(process, event, event_name):
    try:
        event()
    except InvalidStateTransition as err:
        print(f'Error: transition of {process.name} from {process.
↪current_state} to {event_name} failed'
            )
```

- the `state_info()` function shows some basic infomation about the current (active) state of the process

[23]:
```python
def state_info(process):
    print(f'state of {process.name}: {process.current_state}')
```

- at the beginning of the `main()` function, we define some string constants, which are passed as `event_name`

[24]:
```python
def main():
    RUNNING = 'running'
    WAITING = 'waiting'
    BLOCKED = 'blocked'
    TERMINATED = 'terminated'

    p1, p2 = Process('process1'), Process('process2')
    [state_info(p) for p in (p1, p2)]

    print()
    transition(p1, p1.wait, WAITING)
    transition(p2, p2.terminate, TERMINATED)
    [state_info(p) for p in (p1, p2)]
    print()
    transition(p1, p1.run, RUNNING)
    transition(p2, p2.wait, WAITING)
    [state_info(p) for p in (p1, p2)]
    print()
    transition(p2, p2.run, RUNNING)
    [state_info(p) for p in (p1, p2)]
    print()
    [transition(p, p.block, BLOCKED) for p in (p1, p2)]
    [state_info(p) for p in (p1, p2)]
```

```
print()
[transition(p, p.terminate, TERMINATED) for p in (p1, p2)]
[state_info(p) for p in (p1, p2)]
```

- notice how using a good `state_machine` eliminates conditional logic
- there's no need to use long and error-prone `if...else` statements that check for each and every state transition and react to them