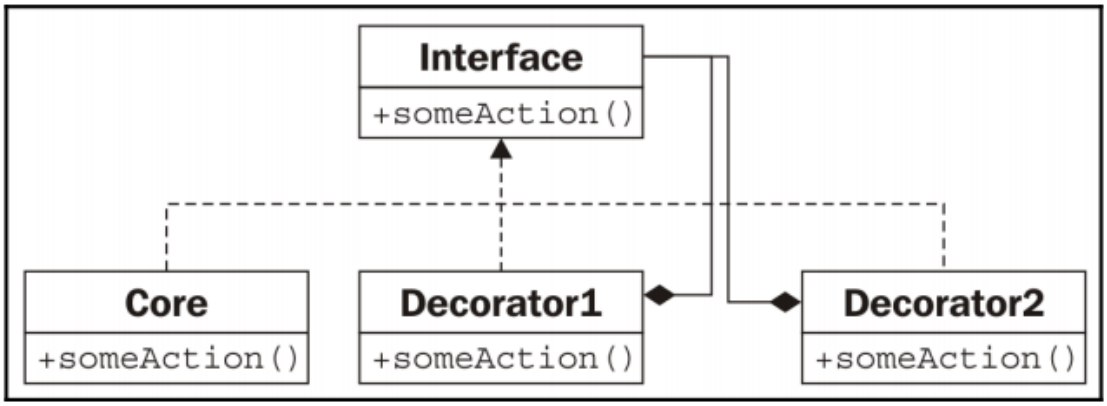
Chapter 10 - Python Design Patterns I

April 4, 2021

0.1 Decorator Pattern

- allows us to wrap any object that provides core functionality with other objects that alter this functionality
- the two primary uses of the decorator pattern are:
  - enchancing the response of a component as it sends data to a second component
  - supporting multiple optional behaviors
- the second option is often a suitable alternative to multiple inhertance
- we construct a core object, then create a decorator wrapping that core
- since the decorator object has the same interface as the core object, we can wrap the new object in other decorators

- `Core` and all the decorators implement a specific `interface`
- the decorators maintain a refrence to another instance of that `interface` via composition
- when called, the decorator does some added processing before or after calling its wrapped interface
- the wrapped object may be another decorator, or the core functionality
- while multiple decorators may wrap each other, the object in the `center` of all those decorators provides the core functionality



0.2 Decorator Example

- `response` function accepts a `socket` parameter and prompts for data to be sent as a reply, then sends it
- to use it, we construct a server socket and tell it to listen to port `2401`
- when a client conencts, it calls the response function, which requests data interactively and responds appropriately

- notice that the `respond` function only cares about two methods of the socket interface `send` and `close`

[ ]:
```
import socket

def respond(client):
    response = input("Enter a value: ")
    client.send(bytes(response, "uft8"))
    client.close()

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("localhost", 2401))
server.listen(1)

try:
    while True:
        client, addr = server.accept()
        respond(client)
finally:
    server.close()
```

[ ]:
```
import socket

client = socket.socket(socker.AF_INET, socket.SOCK_STREAM)
client.connect(("localhost", 2401))
print("Received: {0}".format(client.recv(1024))
client.close()
```

- we are going to create a pair of decorators that customize the socket behavior without having to extend or modify the socket itself

[ ]:
```
class LogSocket:
    def __init__(self, socket):
        self.socket = socket

    def send(send, data):
        print(
            f"Sending {data} to {self.socket.getpeername()[0]}"
            )
        self.socket.send(data)

    def close(self):
        self.socket.close()
```

- the `logSocket` class decorates a socket object and presents the `send` and `close` interface to client sockets
- to use the decorator we only have to change one line
- instead of calling `respond` with the socket, we call it with a decorated socket `respond(LogSocket(client))`

- we could have just extended the `socket` class and overried its `send` method and called `super().send`
- but we should always chose a decorator over inheritance when we need to modify the object dynamically, according to some condition

0.2.1 Decorators in Python

- in python, functions are objects too
- function decoration is so common in python that there is a special syntax to make it easy to apply decorators to functions

- the decorator function is simmilar to the example we explored earlier
- in those cases the decorator took a socket-like object and created a socket-like object
- this time, our decorator takes a function object and returns a new function object

- A function, `log_calls` accepts another function
- this function defines a new function named `wrapper` that does some extra work before calling the orginal function
- the inner function is returned from the outer function

- the syntax allows us to build up decorated function objects dynamically, just as we did with the socket example
- typically, these `decorators` are generally modifications that are applied permanently to different functions

[ ]:
```python
import time

def log_calls(func):
    def wrapper(*args, **kwargs):
        now = time.time()
        print(
            f"Calling {func.__name__} with {args} and {kwargs}"
        )
        return_value = func(*args, **kwargs)
        print(
            f"Executed {func.__name__} in {time.time() - now}ms"
        )
        return return_value
    return wrapper

def test1(a, b, c):
    print("\ttest1 called")

def test2(a, b):
    print("\ttest2 called")

def test3(a, b):
    print("\ttest3 called")
    time.sleep(1)
```

```
test1 = log_calls(test1)
test2 = log_calls(test2)
test3 = log_calls(test3)
test1(1, 2, 3)
test2(4, b=5)
test3(6, 7)
```

- instead of applying the decorator function after the method definition, we can use the `decorator` syntax to do it all at once
- we apply it at the beginning so people know that the function is being modified

[ ]:
```
@log_calls
def test1(a, b, c):
    print("\ttest1 called")
```

0.2.2 Decorators Brakdown

- https://www.youtube.com/watch?v=leNjxtzTGAc

- decorator is a function that takes in another function
- decorators are possible because functions are objects in pytyhon
- fucntions are first-order values
- decorators are a part of functional programming

- you can nest decorators
- order matters and the closer you are to the `def` the earlier you are called

[ ]:
```
def tracer(func):
    @functools.wrap(func)
    def wraper(*args, **kwargs):
        print("Enterning...")
        answer = func(*args, **kwargs)
        print('Exiting...')
        return answer
    return wrapper


@tracer
def hellow_word():
    print("hello world")

hello_world()
```

0.2.3 Common Decorators

Decorator	*Applies To*	*Purpose*
`@classmethod`	Methods	Makes method callable from class with class parameter
`@staticmethod`	Methods	Makes method callable from class without class parameter
`@property`	Methods	Adds getters and setters for attributes
`@app.route`	Functions	Flask: binds a function to a URL
`@pytest.mark.parametrize`	Functions	pytest: runs tests with different input combos

@classmethod decorator - the class method will turn any method into a class method instead of an instance method - that means the `hello` method below can be called directly from the class, rather then the object of the class - you can call the method from just the blueprint - notice that the class method uses `cls` by convention over using `self`

[ ]:
```
class Greeter

    @classmethod
    def hello(cls):
        name = cls.__name__
        print(f'hello from {name}')

Greeter.hello()
```

@staticmethod decorator - does not pass refrence to the method - notice that there is no `cls` or `self` - `staticmethod` is a simpler method then the `classmethod`

[ ]:
```
class Greeter:

    @staticmethod
    def hello():
        print('goodbye')
```

0.3 Observer Pattern

- the observer pattern is useful for state monitoring and event handling situations
- this pattern allows a given object to be monitored by an unknown and dynamic group of observer objects
- whenever a value on the core object changes,it lets all the observer objects know that a change has occured, by calling an `update()` method

0.3.1 Observer Example

- might be useful in a redundant backup system

- we can write a core object that maintains certain values and then have one or more observers create serialized copies of that object
- these copies might be stored in a database, on a remote host, or in a local file

- the object has two properties that, when set, call the `_update_observers` method on itself
- all this method does is loop over any registered observers and let each know that something has changed
- in this case, we call this observer object directly; the object will have to implement `__call__` to process the update

[ ]:
```
class Inventory:
    def __init__(self):
        self.observers = []
        self._product = None
        self._quantity = 0

    def attach(self, observer):
        self.observers.append(observers)

    @property
    def product(self):
        return self._product

    @product.setter
    def product(self, value):
        self._product = value
        self._update_observers()

    @property
    def quantity(self):
        return self._quantity

    @quantity.setter
    def quantity(self, value):
        self._quantity = value
        self._update_observers()

    def _update_observers(self):
        for observer in self.observers:
            observers()
```

- below is a simple observe
- the observed object is set up in the initializer and when the observer is called, we do *something*

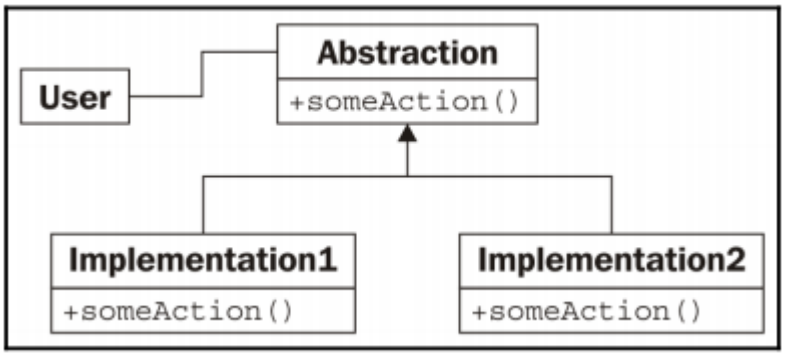
[ ]:
```
class ConsoleObserver:
    def __init__(self, inventory):
        self.inventroy = inventory
```

```
    def __call__(self):
        print(self.inventory.product)
        print(self.inventory.quantity)

i = Inventory()
c = ConsoleObserver(i)
i.attach(c)
i.product = 'Widget'
i.quantity = 5
```

- after attaching the observer to the `Inventory` object, whenever we change one of the two observed properties, the observer is called and its actions is invoked
- the key idea here is that we can easily and add totaly different types of observations that back up the data in a file, database or internet application at the same time

0.4 Strategy Pattern

- the pattern implements different solutions to a single problem, each in a different object
- the client code can then choose the most appropriate implementation dynamically at runtime
- typically, a different algorithms have different trade-offs
  - one might be faster than another but uses a lot more memory and anohter may be useful when multiple CPUs are present or a distributed system is provided
- the `User` code connecting to the stragety pattern simply needs to know that it is dealing with the `ABstraction` interface



0.4.1 Strategy Example

- the classic example of the strategy pattern is sort routines
- if we have a collection of objects we pass to the `sort` method, the object may be a `QuickSorter` or `MergeSorter` object
- you can imagine a desktop image resizer
- the advantage is we do not have to use akward `if/else` statements

[1]:
```
class TiledStrategy:
    def make_background(self, img_file, desktop_size):
        pass
```

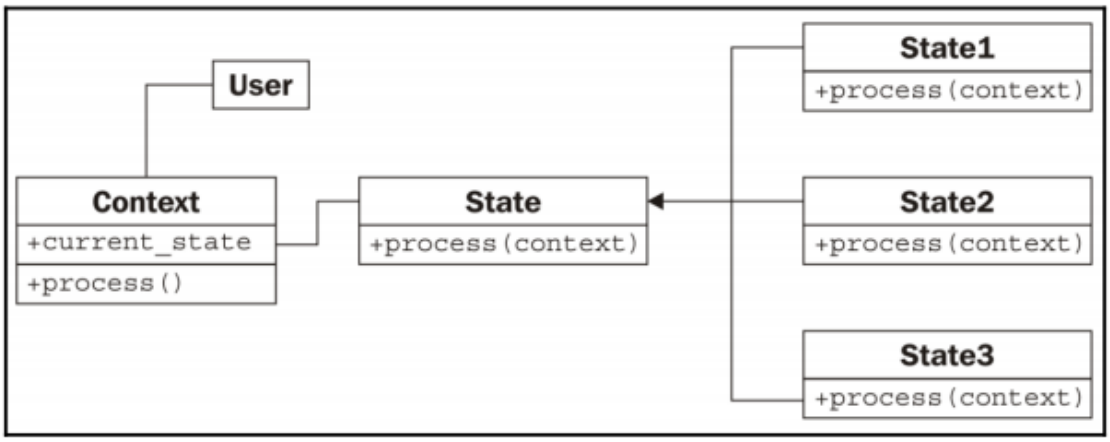
```
class CenteredStrategy:
    def make_background(self, img_file, desktop_size):
        pass

class ScaledStrategy:
    def make_background(self, img_file, desktop_size):
        pass
```

0.4.2 Strategy In Python

- seen in most other OOP languages but not seen in python
- since python has `first-class` functions, the stragety pattern is unnecessary

0.5 State Pattern

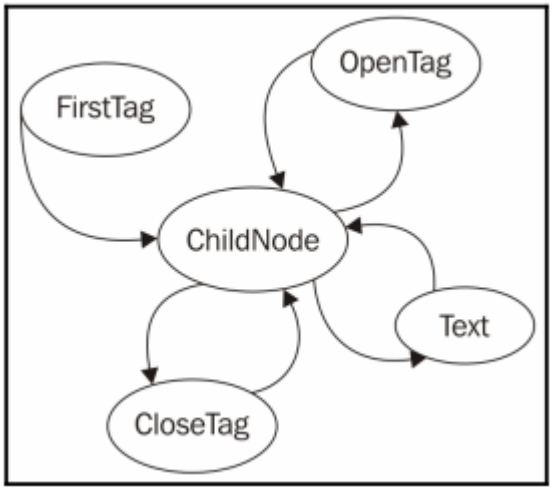
- simmilar to the stragety pattern but the intern and the purpose are different
- the goal of the state pattern is to represent `state-transition` systems
  - systems where is is obvious that an object can be in a specicfic state and that certain activities may drive it to a different state
- to make this work, we need a manager or context class that provides an interface for switching states
- internally, this class contains a pointer to the current state
- each state knows what the other state it is allowed to be in and will transition to those states depending on actions invoked upon it
- we have two types of classes: the context class and multiple state classes
- the context class maintains the current state and forwards actions to the state class
- the state classes are typically hidden from other objects that are calling the context
- it acts like a black box that happens to perform state management internally



0.5.1 State Example

- imagine a `XML` parsing tool

- the context class will be the parser itself. It will take a string as input and place the tool in an initial parsing state
- the verious parsing states will eat characters, looking for a specific value and when that value is found, change to a different state
- the goal is to create a tree of node objects for each tag and its contents
- we can imagine having the following states: `FirstTag`, `childNode`, `openTag`, `closeTag`, `Text`
- the `FirstTag` state will switch to `ChildNode` which is resposible for deciding which of the other three states to switch to
- when those states are done, they will switch back to `childNode`



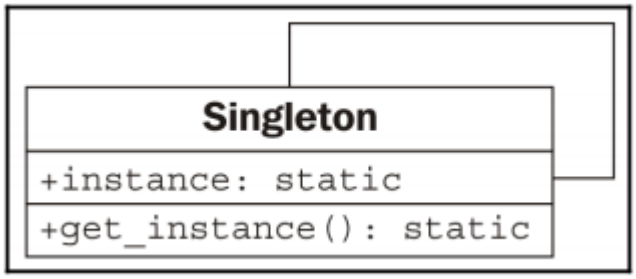
0.6 State vs Stragety

- the stragety pattern is used to choose an algorithm at runtime; generally, only one of those algorithms is going to be chosen for a particular use case
- the state pattern on the other hand, is designed to allow switching between different states dynamically, as some process evolves
- the primary difference is that the strategy pattern is not typically aware of other strategy patterns but the state pattern needs to know which other states that it can switch to

0.7 Singleton Pattern

- many have accused of it as being an anti-pattern
- in python if someone is using this, they are doing something wrong
- basic idea behind the singleton pattern is to allow exactly one instance of a certain object to exist
- typically this object is a sort of manager class
- the problem with that is such objects often need to be refrenced by a wide varity of other objects, and passing reference to the manager object around to the methods and constructors that need them can be hard to read
- instead, when a singleton is used, the separate objects request the single instance of the manager object from the class, so a refrence to it need not be passed around

- in most programming enviorments, singletons are enforced by making the constructor private; this prevents others from making additional instances of it
- you then provide a static method to retrieve the single instance
- this creates a new instance the first time it is called and then returns that same instance for all subsequent calls



0.7.1 Singleton Implementation

- when `__new__` is called, it normally constructs a new instance of that class
- when we overrride it, we first check whether our singleton instance had been created; if not, we create it using a `super` call
- thus we will always get the exact same instance
- generally dont use a `singleton` implementation

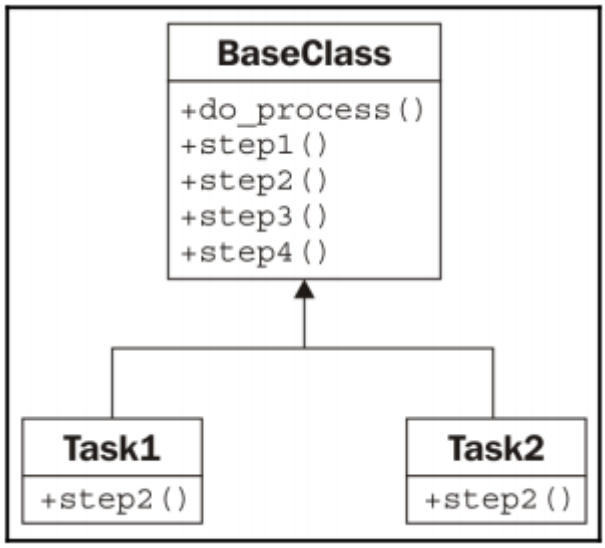
[4]:
```
class OneOnly:
    _singleTon = None

    def __new__(cls, *args, **kwargs):
        if not cls._singleton:
            cls._singleton = super(OneOnly, cls
                ).__new__(cls, *args, **kwargs)
        return cls._singleton
```

0.7.2 Module Variables can Mimic Singletons

- not as safe as a singleton in that people could reassign those varaible at any time, but that is acceptable
- ideally we should give them a mechanism to get access to the `default singleton` value while also allowing them to create other instances
- to use `module-level` variables instead of a singleton, we instantiate an instance of the class after we've defined it

0.8 Template Patern

- template pattern is useful for the `Don't Repeat Yourself` principle
- it is designed for situations where we have several different tasks to accomplish, that have some, but not all, steps in common
- the commons teps are implemented in the base class and the distinct steps are overriden in subclasses to provide custom behavior


BaseClass
+do_process()
+step1()
+step2()
+step3()
+step4()
Task1
+step2()
Task2
+step2()