The Decorator Pattern and Python Decorators

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

Introduction

In this session we will see the classic *Decorator* pattern from GoF. We also also look that the Python language feature with the same name and compare and contrast them.

Decorator Intent

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality"
(*GoF*)

The typical way to extend or modify a class's behaviour is to subclass it. But this modification is locked in at compile time. Sometimes we want to wait until runtime to apply the modification, perhaps to apply it selectively.

An Example

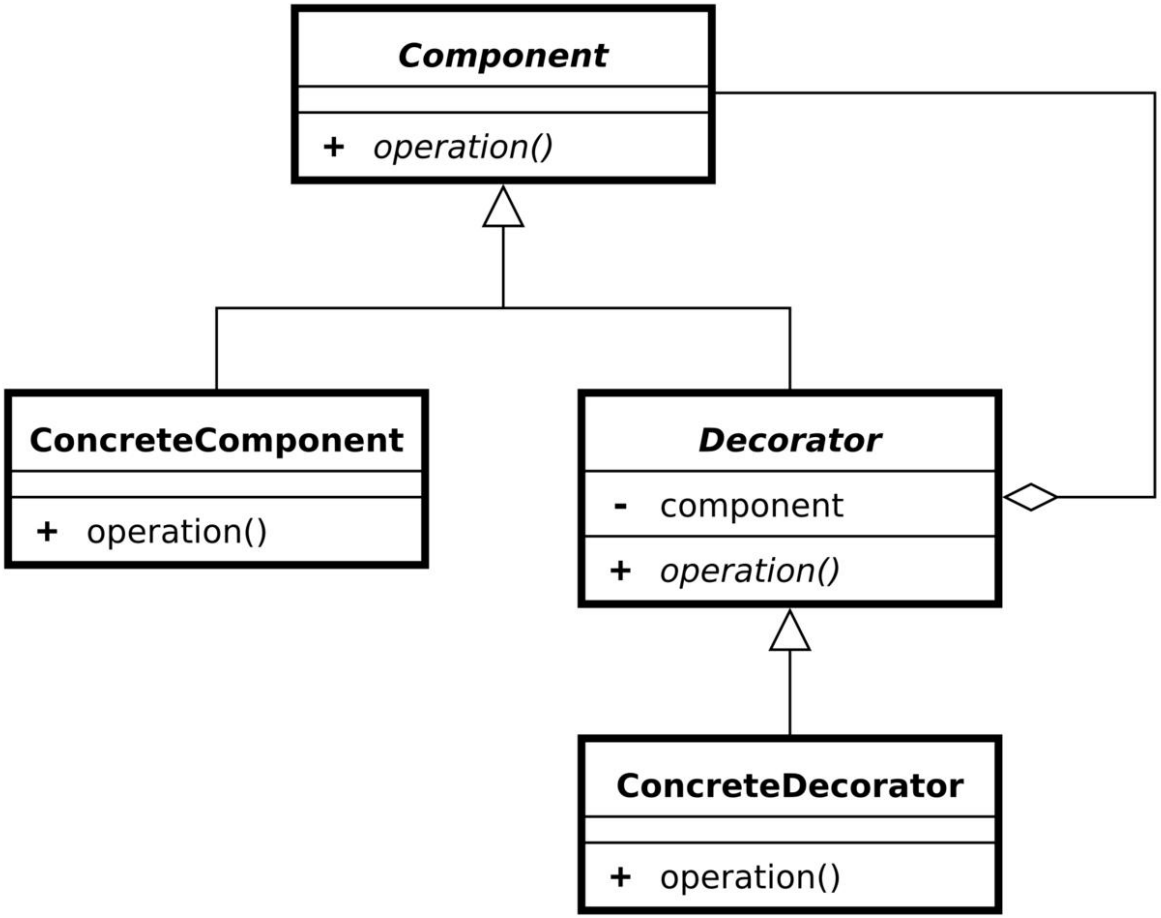
When a player of one of Runaway's games is having trouble and raises a support issue, we enable extra logging of that player's API calls. Then, at runtime we modify the behaviour of API calls *Just for that player*, to log the details of the calls.

The Plan

Start with a standard instance of a class, `basic`. From here we

- Construct a decorator instance that "wraps" basic.
- This decorator instance presents the same interface as `basic`, making it a replacement.
- However, the decorator can take over some of `basic`'s methods, while also leaving some unchanged.
- We can also recover the undecorated `basic` instance. It is unchanged by the process.
- Also, we can selectively choose whether to apply the decorator, or choose from a range of decorators to apply.

Structural Diagram



Decorator Implementation: Base

```
from abc import ABC, abstractmethod

class AbstractMessenger(ABC):

  @abstractmethod
  def say_hello(self):
      pass

  @abstractmethod
  def say_goodbye(self):
      pass

  @abstractmethod
  def tell_the_time(self):
      pass
```

Decorator Implementation: Concrete Subclass

```
class ConcreteMessenger(AbstractMessenger):

  def say_hello(self):
      return 'Hello'

  def say_goodbye(self):
      return 'Goodbye'

  def tell_the_time(self):
      return datetime.now()
```

Decorator Implementation: Decorators

```
class AbstractDecorator(AbstractMessenger):

  def __init__(self, messenger):
      self._messenger = messenger

  def tell_the_time(self):
      return self._messenger.tell_the_time()

class UppercaseDecorator(AbstractDecorator):

  def say_hello(self):
      msg = self._messenger.say_hello()
      return msg.upper()

  def say_goodbye(self):
      msg = self._messenger.say_goodbye()
      return msg.upper()
```

Decorator Implementation: In Use

```
basic_messenger = ConcreteMessenger()
print(basic_messenger.say_hello()) # prints 'Hello'

upper_messenger = UppercaseDecorator(basic_messenger)
print(upper_messenger.say_hello()) # prints 'HELLO'
```

Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `11-practical` in your practicals repo.
3. Add a subdirectory, `11-practical` and copy `11-practical.ipynb` from the class materials into it.
4. Open a shell, cd to this directory, and run `jupyter notebook` to open the notebook. Complete the first two questions.
5. We will discuss results in 20ish minutes.

Python Decorators

Python has a core language feature called a decorator. You've seen them already.

```
@abstractmethod
def say_hello(self):
    pass
```

A lot of literature on the topic will go out of its way to point out that Python decorators are not the same thing as the Decorator pattern. Except they kind of are. I mean, they're not... but sometimes they are.

Decorating Functions

In practice, Python decorators are usually applied to functions rather than clases, although we definitely can decorate classes.

```
@abstractmethod
def say_hello(self):
    pass
```

There are many decorators built into the language or provided in the standard library. We've already seen some like `@property` and `@classmethod`.

Implementing a Decorator

A function decorator generally "wraps" a function and returns a modified version of that function.

```
import functools

def shouty(wrapped_fn):
    @functools.wraps(wrapped_fn)
    def wrapper(*args, **kwargs):
        msg = wrapped_fn(*args, **kwargs)
        return msg.upper()
    return wrapper

@shouty
def hello():
    return 'hello'

hello() #prints 'HELLO'
```

Decorators With Arguments

A function decorator generally "wraps" a function and returns a modified version of that function.

```
def shouty_or_not(choice):
    def maybe(wrapped_fn):
        @functools.wraps(wrapped_fn)
        def wrapper(*args, **kwargs):
            msg = wrapped_fn(*args, **kwargs)
            if choice:
              return msg.upper()
            else:
              return msg
        return wrapper
    return maybe
```

A Class Decorator

Real Python has a nice example of a class decorator that make a class into a Singleton.

`https://realpython.com/primer-on-python-decorators/#creating-singletons`

Conclusions

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality"
(*GoF*)

- Python decorators have several uses, some of which don't satisfy this intent.
- They also have uses that **look** very different in their implementation than don't look like the textbook pattern, but nonetheless **do** satify the intent.
- Regardless of this, Python decorators are handy and widely used.