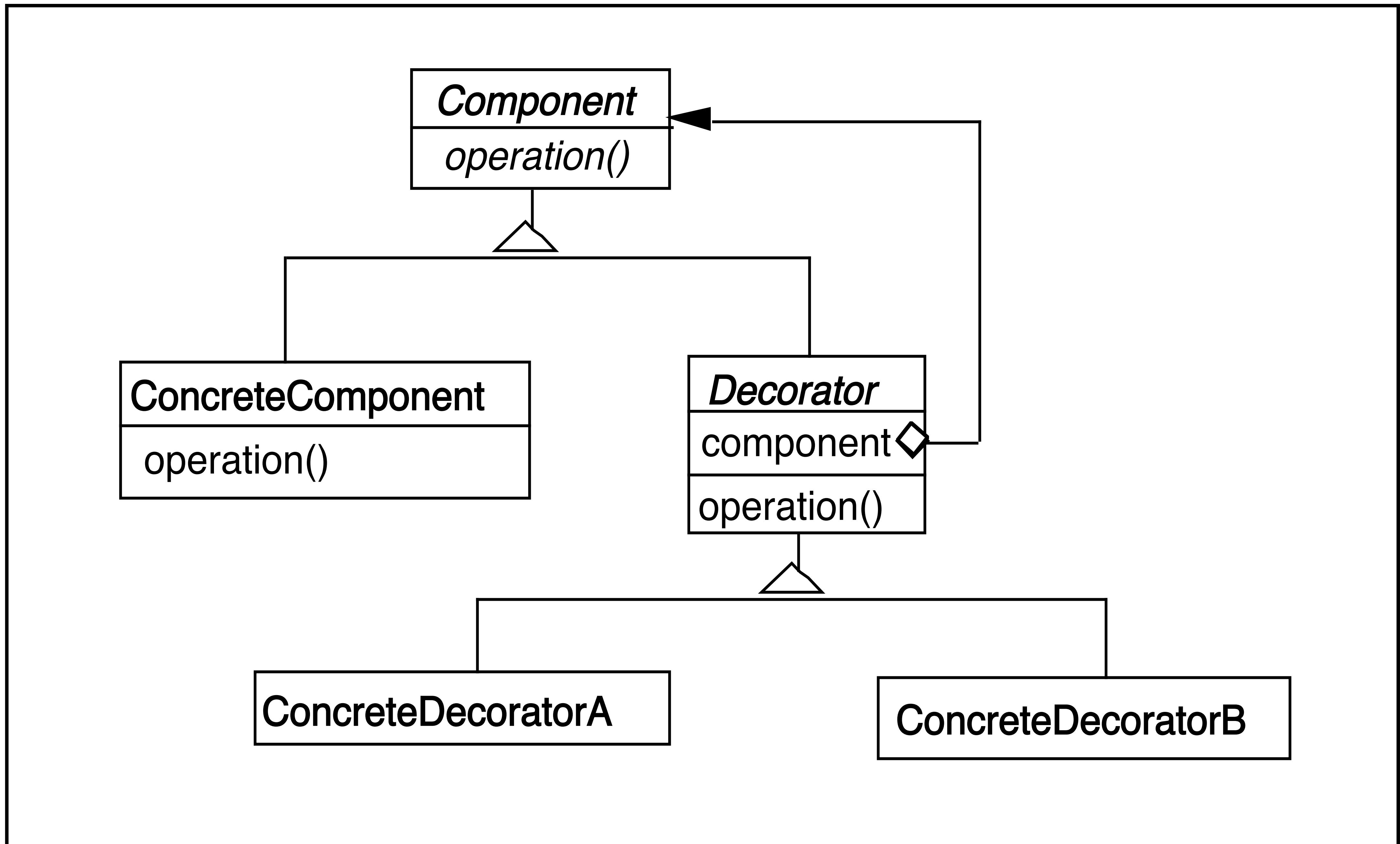
CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2021
Doc 11 Python Decorator, Singleton
Oct 14, 2021

Copyright ©, All rights reserved. 2021 SDSU & Roger Whitney,
5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (http://www.opencontent.org/opl.shtml) license
defines the copyright on this document.

The Harder Law

Once you eliminate your number one problem, you promote number two

-- G. Weinberg


Component
operation()
ConcreteComponent
operation()
Decorator
component
operation()
ConcreteDecoratorA
ConcreteDecoratorB

```python
class Counter:

    def __init__(self):
        self.count = 0

    def increase(self):
        self.count = self.count + 1

    def decrease(self):
        self.count = self.count - 1

    def get_count(self):
        return self.count

    def reset(self):
        self.count = 0
```

```python
def test_Counter(self):
    count = Decorator.Counter()
    count.increase()
    count2 = Decorator.Counter()
    count2.decrease()
    self.assertEqual(count.get_count(), 1)
```

A Decorator

```
class Counter:

    def __init__(self):
        self.count = 0

    def increase(self):
        self.count = self.count + 1

    def decrease(self):
        self.count = self.count - 1

    def get_count(self):
        return self.count

    def reset(self):
        self.count = 0
```

```
class MethodCounter:

    def __init__(self, counter):
        self.counter = counter
        self.method_count = 0

    def increase(self):
        self.counter.increase()
        self.method_count = self.method_count + 1

    def decrease(self):
        self.counter.decrease()
        self.method_count = self.method_count + 1

    def get_count(self):
        return self.counter.get_count()
```

```
def test_Decorator(self):
    counter = Decorator.Counter()
    counter.increase()

    counter = Decorator.MethodCounter(Decorator.Counter())
    counter.increase()
    self.assertEqual(counter.get_count(), 1)
```

Issue 1

Insuring decorator has same interface concrete component

```
class Counter:

    def __init__(self):
        self.count = 0

    def increase(self):
        self.count = self.count + 1

    def decrease(self):
        self.count = self.count - 1

    def get_count(self):
        return self.count

    def reset(self):
        self.count = 0
```

```
class MethodCounter:

    def __init__(self, counter):
        self.counter = counter
        self.method_count = 0

    def increase(self):
        self.counter.increase()
        self.method_count = self.method_count + 1

    def decrease(self):
        self.counter.decrease()
        self.method_count = self.method_count + 1

    def get_count(self):
        return self.counter.get_count()
```

Solutions to Issue 1

Team discipline

ABC

Abstract Class

```
from abc import ABC, abstractmethod
class Counter(ABC):

    @abstractmethod
    def increase(self):
        pass

    @abstractmethod
    def decrease(self):
        pass

    @abstractmethod
    def get_count(self):
        pass

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

```
class Concrete_Counter(Counter):

    def __init__(self):
        self.count = 0

    def increase(self):
        self.count = self.count + 1

    def decrease(self):
        self.count = self.count - 1

    def get_count(self):
        return self.count

    def reset(self):
        self.count = 0
```

```
class MethodCounter(Counter):

    def __init__(self, counter):
        self.counter = counter
        self.method_count = 0

    def increase(self):
        self.counter.increase()
        self.method_count = self.method_count + 1

    def decrease(self):
        self.counter.increase()
        self.method_count = self.method_count + 1

    def get_count(self):
        return self.counter.get_count()
```

Runtime error as MethodCounter is abstract

Issue 2 Decorators with added Functionality

```
class MethodCounter():

    def __init__(self, counter):
        self.counter = counter
        self.method_count = 0

    def increase(self):
        self.counter.increase()
        self.method_count = self.method_count + 1

    def decrease(self):
        self.counter.decrease()
        self.method_count = self.method_count + 1

    def get_count(self):
        return self.counter.get_count()

    def reset(self):
        self.counter.reset()
```

```
class LogCounter():

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

    def increase(self):
        self.counter.increase()
        print("Increase")

    def decrease(self):
        self.counter.decrease()
        print("Decrease")

    def get_count(self):
        print("get_count")
        return self.counter.get_count()

    def reset(self):
        self.counter.reset()
        print("reset")
```

Issue 2 Decorators with added Functionality

```
def test_two_Decorators(self):
    counter = Decorator.MethodCounter(Decorator.LogCounter(Decorator.Counter()))
    counter.increase()
    counter.decrease()
    self.assertEqual(counter.method_count, 2)


    counter = Decorator.LogCounter(Decorator.MethodCounter(Decorator.Counter()))
    counter.increase()
    counter.decrease()
    self.assertEqual(counter.method_count, 2)  #Error
```

Issue 3 Attributes are Public

```
def test_Decorator_attribute(self):
    counter = Decorator.Counter()
    counter.increase()
    self.assertEqual(counter.count, 1)

    counter = Decorator.MethodCounter(Decorator.Counter())
    counter.increase()
    self.assertEqual(counter.count, 1)
```

Decorator needs to provide access to concrete component's attributes

- Manually
- __getattr__

All attributes of Object

```
counter = Decorator.Counter()
print(dir(counter))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'bar', 'count', 'decrease', 'get_count', 'increase', 'reset']
```

If object is implemented in C there are a lot more

Adding Attribute access to Decorator

```
class MethodCounter():

    def __init__(self, counter):
        self.counter = counter
        self.method_count = 0

    def increase(self):
        self.counter.increase()
        self.method_count = self.method_count + 1

    def decrease(self):
        self.counter.decrease()
        self.method_count = self.method_count + 1

    def get_count(self):
        return self.counter.get_count()

    def reset(self):
        self.counter.reset()

    def __getattr__(self, name):
        return getattr(self.__dict__['counter'], name)

    def __setattr__(self, name, value):
        if name in ('method_count', 'counter'):
            self.__dict__[name] = value
        else:
            setattr(self.__dict__['counter'], name, value)

    def __delattr__(self, name):
        if name in ('method_count', 'counter'):
            self.__dict__.pop(name)
        else:
            delattr(self.__dict__['counter'], name)
```

```
def test_set_attribute(self):
    counter = Decorator.Counter()
    self.assertEqual(counter.count, 0)
    decorated_counter = Decorator.MethodCounter(counter)
    decorated_counter.count = 21
    self.assertEqual(counter.count, 21)


def test_remove_attribute(self):
    counter = Decorator.Counter()
    self.assertTrue( 'count' in dir(counter))
    decorated_counter = Decorator.MethodCounter(counter)
    delattr(decorated_counter, 'count')
    self.assertFalse('count' in dir(counter))
```

Explain This

```
class FancyDecorator:

    def __init__(self, counter):
        self.counter = counter
        self.method_count = 0

    def __getattr__(self, name):
        self.method_count = self.method_count + 1
        __get_attr__ = getattr(self.__dict__['counter'], name)
        def intercepted(*args):
            return __get_attr__(*args)
        if str(type(__get_attr__)) == "<class 'method'>":
            return intercepted
        else:
            return __get_attr__
```

```
def test_fancy_decorator(self):
    counter = Decorator.Counter()
    self.assertEqual(counter.count, 0)
    decorated_counter = Decorator.FancyDecorator(counter)
    decorated_counter.increase()
    self.assertEqual(counter.count, 1)
    decorated_counter.count = 25
    self.assertEqual(counter.count, 25)
    self.assertEqual(decorated_counter.count, 25)
```

Python & GOF Patterns

https://python-patterns.guide

Singleton

Warning

Simplest pattern
- But has subtle issues particularly in Java

Most controversial pattern

Intent

Ensure a class only has one instance

Provide global point of access to single instance

Singleton

```
public class Counter {
    private int count = 0;
    private static Counter instance;
    private Counter() { }

    public static Counter instance() {
        if (instance == null)
            instance = new Counter();
        return instance;
    }

    public int increase()  {return ++count;}
}
```

One instance

Global access

Some Uses

Java Security Manager

Logging a Server

Null Object

Globals are Evil

Why Singletons Are Controversial(Evil)

Singletons provide global access point for some service

Hidden dependencies

Is there a different design that does not need singletons

- Pass a reference

Why Singletons Are Controversial(Evil)

Singletons allow you to limit creation of objects of a class

- Should that be the responsibility of the class?
- Class should do one thing
- Use factory or builder to limit the creation

Why Singletons Are Controversial(Evil)

Singletons tightly couple you to the exact type of the singleton object

- No polymorphism
- Hard to subclass

Why Singletons Are Controversial(Evil)

Singletons carry state with them that last as long as the program lasts

Persistent state makes testing hard and error prone

Why Singletons Are Controversial(Evil)

A Singleton today is a multiple tomorrow

Singleton pattern makes it hard to change to allow multiple objects

Why Singletons Are Controversial(Evil)

In Java Singletons are a lie

More on this later

Singleton Implementation - Python

Strait forward Translation

```
class Foo:
    _instance = None

    def __init__(self):
        raise RuntimeError('Call instance() instead')

    @classmethod
    def instance(cls):
        if cls._instance is None:
            print('Creating new instance')
            cls._instance = cls.__new__(cls)

        return cls._instance
```

```
x = Foo.instance()
```

https://python-patterns.guide/gang-of-four/singleton/

Simpler Implementation

```
class Foo(object):
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            print('Creating the object')
            cls._instance = super(Logger, cls).__new__(cls)
            # Put any initialization here.
        return cls._instance



     x = Foo()
```

Singleton Implementation - Java

Why Not Use This?

```
public class Counter {
    private static int count = 0;

    public static int increase()  {return ++count;}
}
```

Why Not Use This?

```
public class Counter {
    private int count = 0;
    private Counter() { }

    public static Counter instance = new Counter();

    public int increase()  {return ++count;}
}
```

Two Useful Features

Lazy

- Only created when needed

Thread safe

Recommended Implementation

```
public class Counter {
   private int count = 0;
   private Counter() { }

   private static class SingletonHolder {
      private final static Counter INSTANCE = new Counter();
   }

   public static Counter instance() {
         return SingletonHolder.INSTANCE;
   }

   public int increase()  {return ++count;}
}
```

Correct but not Lazy

```
public class Counter {
    private int count = 0;
    protected Counter() { }

    private final static Counter INSTANCE = new Counter();

    public static Counter instance() {
            return INSTANCE;
    }

    public int increase()  {return ++count;}
}
```

Lazy, Thread safe with Overhead

```
public class Counter {
    private int count = 0;
    private static Counter instance;
    private Counter() { }

    public static synchronized Counter instance() {
        if (instance == null)
            instance = new Counter();
        return instance();
    }

    public int increase()  {return ++count;}
}
```

Java Templates & Singleton

Does not compile

```
public class TemplateSingleton<Type> {
    Type foo;

    public static TemplateSingleton<Type> instance =
      new TemplateSingleton<Type>();
}
```

When is a Singleton not a Singleton?

When Java Garbage Collects Classes

Singleton class can be garbage collected
Singleton loses any value it had

Solution

Turn off garbage collection of classes (-Xnoclassgc)

Make sure there is always a reference to the class/instance

When Multiple Java Class Loaders are Used

When loaded by two different class loaders there will be two versions of the class

Some servlet engines use different class loader for each servlet

Using custom class loaders can cause this

Purposely Reloading a Java Class

Servlet engines can force a class to be reloaded

Serialize and Deserialize Singleton Object

Serialize the singleton
Deserialize the singleton
You now have two copies

One way to serialize a Java object is using ObjectOutputStream

Ruby Marshal.dump() will not marshal a singleton