# Decorator Pattern

CS342 Fall 16

## Design Problem

You are writing software to receive simple plaintext data from a URI, ‘mytexttransfer.edu’ over the network and write it out to a file. You will always need to write the text out to a file, but sometimes you will need additional operations depending on the file requirement. You may need to:

- number each line as you write it out
- add a time stamp to each line
- add a checksum from the text so that you can ensure that it was written and stored properly.

You might need to do all of these, or just some, or none. Design the software to accomplish this.

## Methods for Every Function

- Solution 1:
  - Write a method for each of these, then let the client worry about calling the appropriate methods
    - client has to know and call methods for every line. Every line is a chance for a bug.
    - All the code is organized into a single class, violating the “do 1 thing” principle
- Solution 2:
  - Use inheritance to make a subclass for each operation
    - You are limited to one operation, line-numbering or checksum, not both, unless you create a subclass for each combination as well (class bloat)

## Base Class

- What about a runtime (compositional) solution?
    - Dynamically create your object from the spare classes lying around
        - as long as they all have the same interface, the clients don’t have to know which class they are using
- Let’s create a base class to identify a shared interface each operation can have
    - Create a base class, WriterDecorator to define the interface for various writer operation classes
- This interface will be a wrapper for all the Writer classes

## Build Your Own Operation

- Create a SimpleWriter class that just receives the file and writes it out as plain text
    - This is the base operation, so this is the primary writer class
- Create a decorator class for each required operation that you can insert between the client and the operation
    - NumberingWriter, TimeStampWriter, CheckSumWriter
        - These classes decorate SimpleWriter

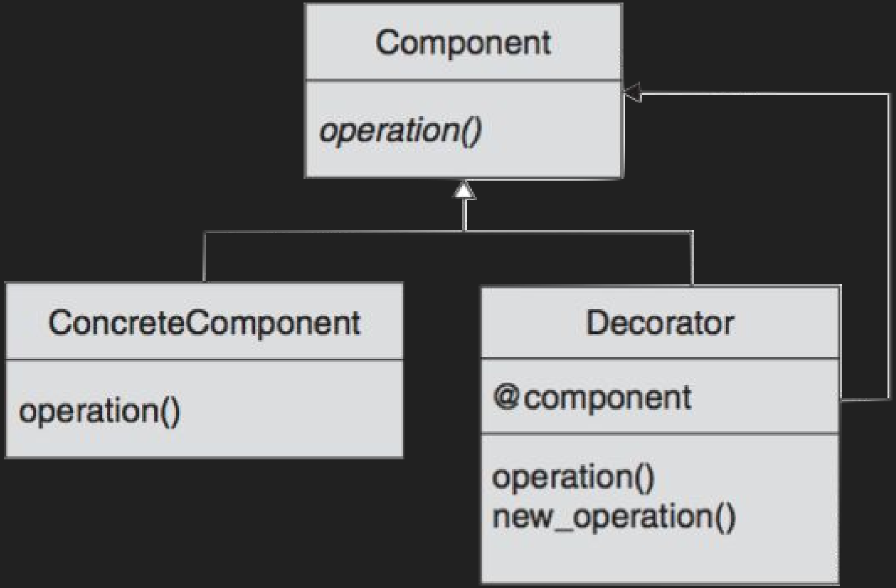
## Objects Assemble!

- We can build our command in 1 line

  -
    ```
    writer = NumberingWriter.new(SimpleWriter.new('final.txt'))
    writer.write_line('Hello out there')
    ```
- We can also alter or add to our command at runtime

  -
    ```
    writer =NumberingWriter.new(SimpleWriter.new('final.txt'))
    writer.write_line('Hello out there')
    writer = CheckSummingWriter.new(TimeStampingWriter.new((writer))
    writer.write_line('Hello again')
    ```

## Decorator Class Structure



- The ConcreteComponent is the “real” object, the object that implements the basic component functionality.
- The Decorator class has a reference to a Component—the next Component in the decorator chain—and it implements all of the methods of the Component type.

## Decorator Pattern Classes

```
class Component
    def intialize(component)
        @component
    end

    def operation()
        raise ‘abstract method’
    end
end

class ConcreteComponent
    def operation()
end
```

```
class Decorator
    @component

    def operation()
        …
        @component.operation()
    end
end
```

## The Decorator Pattern

- The Decorator pattern allows you to:
  - add features to your program without turning the whole thing into a huge, unmanageable mess
  - vary the responsibilities of an object during runtime
- Decorator pattern describes decorators that are essentially shells:
  - Each takes in a method call,
  - adds its own special twist,
  - and passes the call on to the next component in line until getting to the final, real object, which actually completes the basic request.

## alias

- ‘alias’ allows us to give another name to a method
  - alias <<new_name>> <<old_name>>
    - alias old_write_line write_line
- using object runtime modification and aliasing we can add simple decorator methods, but there’s a problem...

  -
    ```
    class << w
        alias old_write_line write_line
        def write_line(line)
            old_write_line("#{Time.new}: #{line}")
        end
    ```

## When to use decorators

- For simple decorators, ROM and aliasing are useful
    - one or two level decorator methods
        - However, name collisions can become a problem
- For more complex operations, Decorator classes manage the complexity
    - many optional decorator operations that require additional operations

## Decorator Cons

- Often easier on the developer than the client
    - separation of concerns
    - modularity and encapsulation
- Client still has to build the object with an ugly line like:

    -
      ```
      CheckSummingWriter.new(TimeStampingWriter.new(
                          NumberingWriter.new(SimpleWriter.new('final.txt'))))
      ```
        - there are patterns to help with this (Builder)
- Decorators incur a performance hit
    - especially as they grow longer and longer

## Classwork:
## Home Theater System

## Classwork:
## Display Student