Decorator Pattern

Objectives

- Discuss typical overuse of inheritance
- How to decorate your classes at runtime using a form of object composition
- Able to add new responsibilities to your objects without making any code changes

Case Study - Car Dealership

- The Car Dealer XYZ needs to update the ordering system
- The existing system has these classes
  - Automobile - abstract class, subclassed by all vehicles offered in the car dealer. It has a price() method that is abstract.
  - Subclasses - Sedan, Truck, SUV, Van.
  - Each vehicle can have several options like sunroof, security, entertainment, and advanced safety.
  - With all possible combinations that causes Class explosion when using inheritance.
  - Each price() method computes the price of the vehicle along with other options.
  - That pricing is a big maintenance problem.

The First Attempt Of Improving Existing Classes

- Let's add Security, Safety, and Sunroof in the base class Automobile.
- Implement the cost method in the base class to calculate the cost of options.
- The subclasses still overrides the cost(), but they will invoke the superclass's cost method
- The superclass cost() will calculate all options, while the overridden cost() in the subclasses will extend that functionality to include the cost of the specific automobile type.

Other Factors Affect The Design

- The first attempt looked good initially, but after we considered other factors as listed below, we would need to consider other design options.
  - Price changes
  - New options
  - New type of vehicle
  - What if double options?

The Open-Closed Principle

- Classes should be open for extension, but closed for modification.

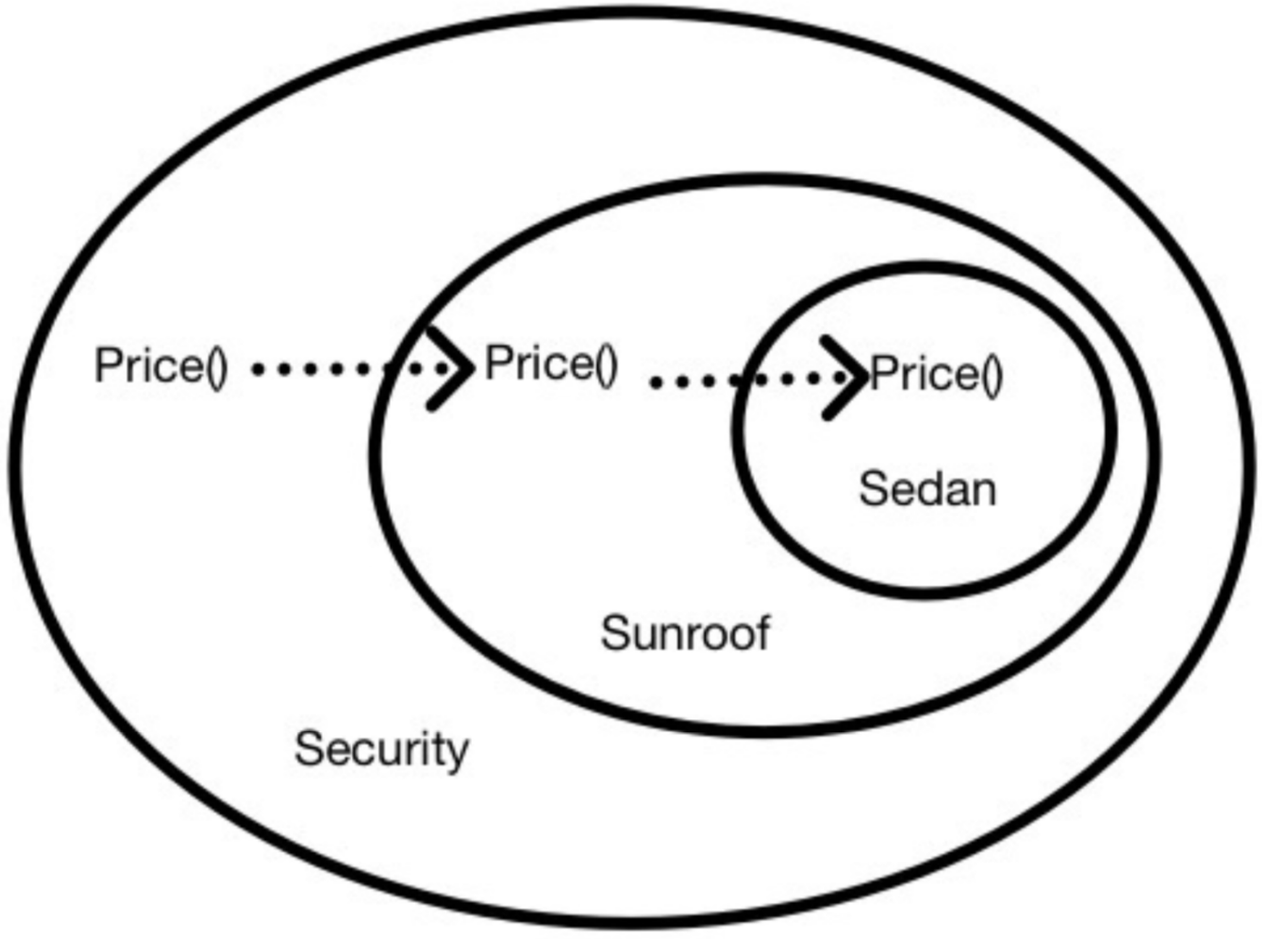
- Allow classes to be easily extended to incorporate new behavior without modifying existing code.

- Example - Observer pattern allows you to add new observers without adding code to the subject.

- Be careful when choosing the areas for applying the Open-Closed principle. Because it will increase complexity and make your code hard to understand.

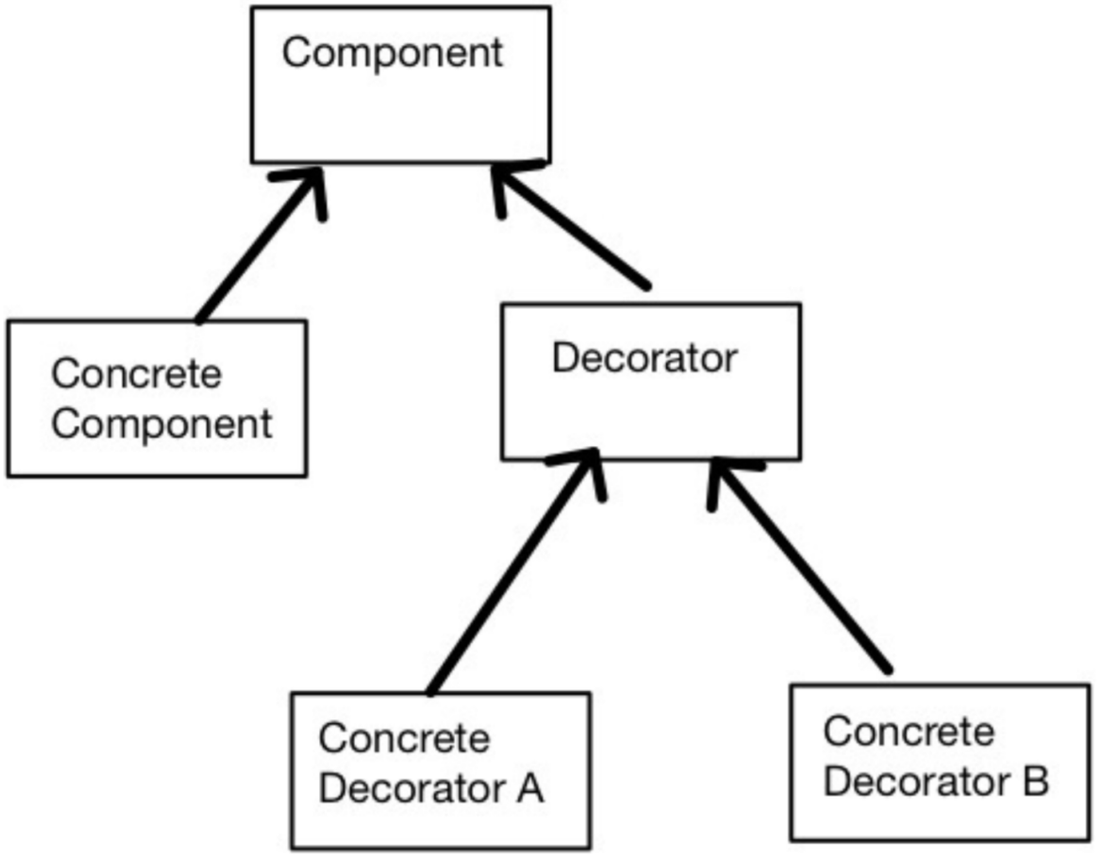
The Idea of Decorator

1. Take a Sedan object.
2. Decorate it with a Sunroof object that wraps the Sedan
3. Decorate it with Security object that wraps the Sunroof object.
4. Call price() method and rely on delegation to add on the option price.We do this by calling the outermost decorator Security and it delegates the price to the Sunroof object and so on. The chain action acts like making recursive calls.



The Decorator Pattern

- Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionalities
- Each component can be used on its own, or wrapped by a decorator.
- Each decorator HAS-A a component.
- Decorator implement the same interface as the component.
- Decorator can be extended. So it can add new methods. But normally adding logic before or after an existing method from the component.
- The concrete component is the one we are going to add new behavior dynamically.



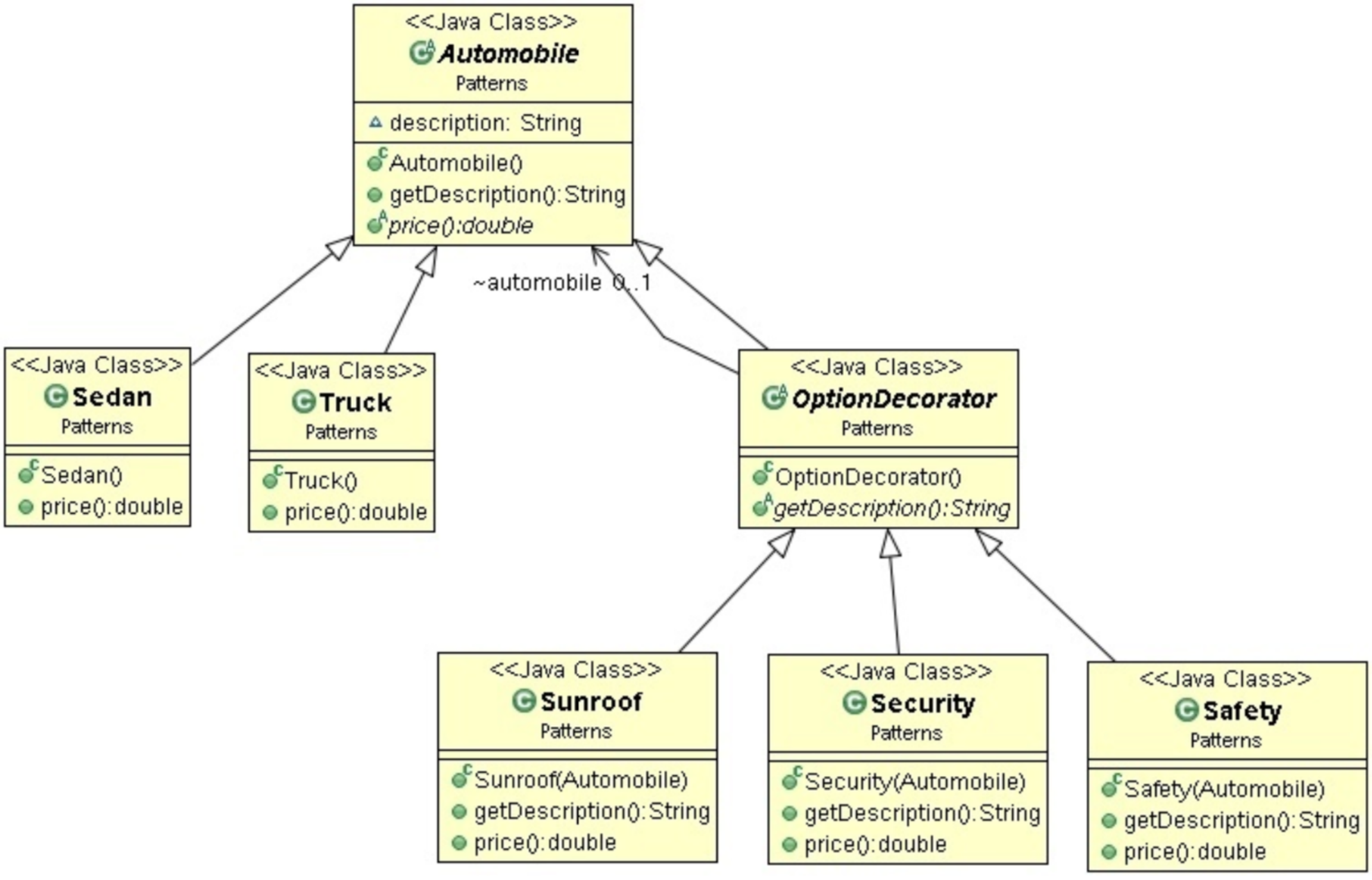
Notes

- Decorators have the same supertype as the objects they decorate.
- Use one or more decorators.
- Allow us to the decorated object as the original object because they have the same type.
- The decorator can add its own behavior before or after delegating to the object it decorates.
- Able to decorate objects at runtime.

Back to Car Dealer Application

- Automobile acts as our abstract component class.
- Four concrete components, one for each vehicle type
- Four options decorators.
- The decorator subclassing the abstract class Automobile in order to have the same type. But we are not trying to inherit its behavior. Instead we gets the behavior through the composition with the basic component.
- Inheritance can only give you the behaviors that determined statically at compile time. Composition allows you mix and match any way at runtime.

The Class Diagram of Car Dealer Application



automobile_decorators.py

```
from abc import ABC, abstractmethod, abstractproperty
import enum

class Automobile(ABC):
    @abstractproperty
    def description(self):
        pass

    @abstractproperty
    def price(self):
        pass

    def display(self):
        print(self.description + ", $" + str(self.price))

class Sedan(Automobile):
    @property
    def price(self):
        return 20000

    @property
    def description(self):
        return "Passenger Car"

class Truck(Automobile):
    @property
    def price(self):
        return 25000

    @property
    def description(self):
        return "Cargo Vehicle"

    def price(self):
        return 1000 + self.automobile.price
```

automobile_decorators.py

```
class OptionDecorator(Automobile):
    def __init__(self):
        self.__automobile = None

    @property
    def automobile(self):
        return self.__automobile

    @automobile.setter
    def automobile(self, automobile):
        self.__automobile = automobile

class Sunroof(OptionDecorator):
    def __init__(self, automobile):
        self.automobile = automobile

    @property
    def description(self):
        return self.automobile.description + ", Sun Roof"

    @property
    def price(self):
        return 1000 + self.automobile.price

class Security(OptionDecorator):
    def __init__(self, automobile):
        self.automobile = automobile

    @property
    def description(self):
        return self.automobile.description + ", Advanced Security
System"

    @property
    def price(self):
        return 800 + self.automobile.price
```

automobile_decorators.py

```
class Safety(OptionDecorator):
    def __init__(self, automobile):
        self.automobile = automobile

    @property
    def description(self):
        return self.automobile.description + ", Advanced Safety
Features"

    @property
    def price(self):
        return 500 + self.automobile.price

def main():
    auto1 = Sedan()
    auto1.display()

    auto2 = Sedan()
    auto2 = Sunroof(auto2)
    auto2 = Sunroof(auto2)
    auto2 = Security(auto2)
    auto2.display()

    auto3 = Security(Safety(Sunroof(Truck())))
    auto3.display()

main()
```

- Example - Truck with three options: Sunroof, Safety, and Security
    1. Call the outermost decorator Security
    2. Security calls price() on Safety.
    3. Safety calls price() on Sunroof.
    4. Sunroof calls price() on Truck.
    5. Truck returns it's price.
    6. Sunroof adds its price to the result from Truck.
    7. Safety adds its price to the result from Sunroof.
    8. Security adds its price to the result from Safety

Summary

- Inheritance is one form of extensions, but not necessarily the best way to achieve flexibility.
- We should allow behavior to be extended without modifying existing code.
- Composition and delegation can add behaviors at runtime.
- The decorator pattern provides an alternative way for extending behavior.
- The decorator pattern consists of a set of decorator classes that are used to wrap concrete components.
- The decorator classes have the same type as the component they decorate.
- Decorators changes the behavior of their objects by adding new functionality by adding new functionality before or after calls to the component.
- Reference
  - http://en.wikipedia.org/wiki/Decorator_pattern

Exercises

- Write a Python program that demonstrates the advantages of the Decorator pattern.
  - Create a class with two methods that produce output when they are called.
  - Create a decorator class for an object of that class that adds two additional methods that produce output when they are called.
  - Instantiate an undecorated object and have your program call both methods. Then instantiate a decorator, use it to "decorate" the original object, then call all four methods provided by the decorator. The output of the original two operations should be decorated in some way. That is, it should be obvious that the original output of the undecorated object is being manipluated in some way by the decorator.

Exercises

- Write a Python program that uses the decorator pattern to decorate an Engineer object with a variety of professional skills.
- For example, an engineer can have the following skills:
  - Software
  - Hardware
  - Construction
  - Management