# Adding Responsibilities with the Decorator Pattern

**Gerald Britton**
IT SOLUTIONS DESIGNER

@GeraldBritton www.linkedin.com/in/geraldbritton

# Motivation

**Car dealership**

**Three models:**

- Economy
- Luxury
- Sport

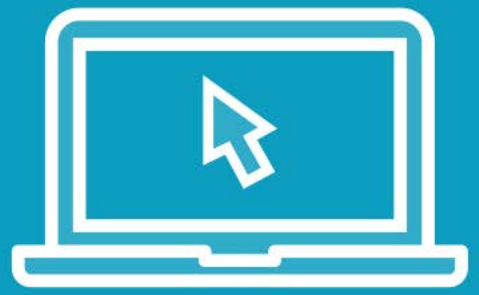
**Options:**

- Engine size: 4 or 6 cylinders
- Paint color: white, red or black
- Upholstery: leather or vinyl

**Cost depends on the model and options**

**Try a class-based approach**

# Demo



Start with an abstract car class

Define concrete class for each model

Subclass each model for option combinations

# Using Subclasses

One subclass per model/options combo

Only two combinations

3 models, 2 engines, 3 colors, 2 upholstery types

36 subclasses!

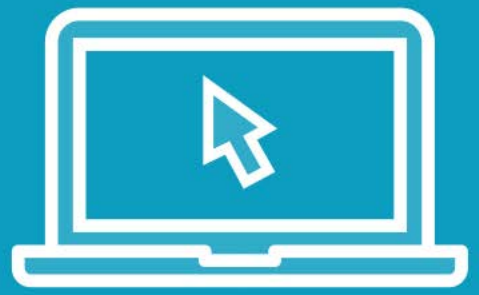
What about the real world?

Thousands of combinations

Subclass explosion

Maintenance nightmare

# Demo



Start with an abstract car class

Define a concrete class for each model

Use properties for the options

# Using Properties

One concrete class per model

More properties to implement

More complicated constructor

Maintenance?

What if the options' prices change?

- Open up ABC

Add interior color property?

- Open the ABC and concrete classes

# Principles Violated

Single Responsibility

Open/Closed

Interface Segregation

Dependency Inversion

Don't Repeat Yourself

# Decorator

Classification: Structural

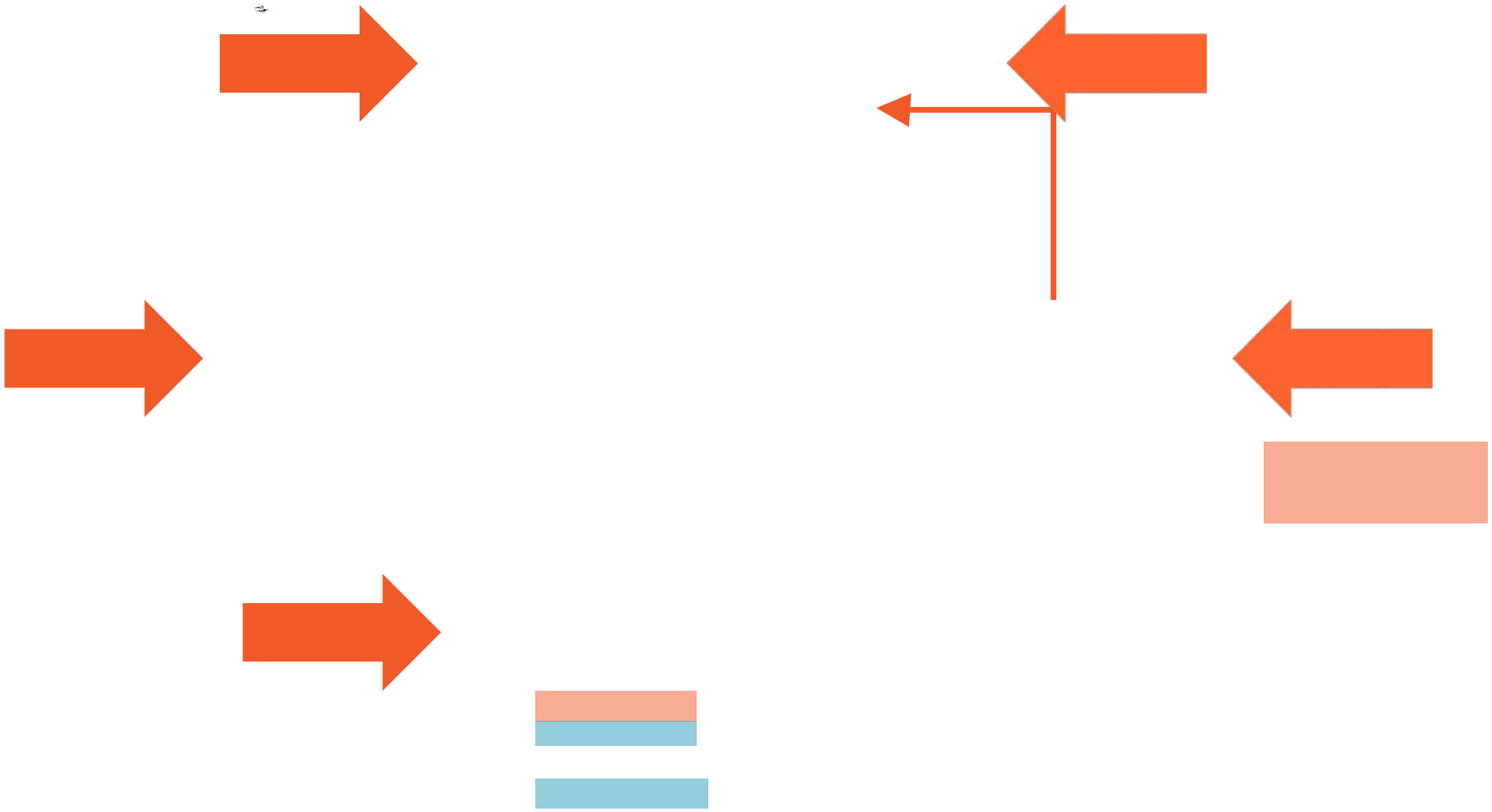
Adds new abilities to an object

Dynamically, at run time

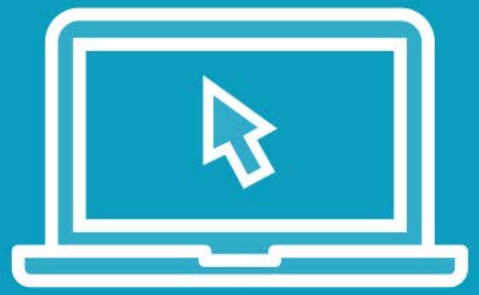
Flexible alternative to subclassing

Also know as the Wrapper Pattern

# Decorator Structure



# Demo



Follow the Decorator Pattern structure

Use decoration instead of subclassing

# Consequences

- More flexible than static inheritance
- Keeps things simple
- No practical limit to decorations
- Transparent to clients
- A decorator has a different type
- Many little objects
- Factory and Builder patterns can help

# Decorator Pattern vs Python Decorators

| Decorator Pattern | Python Decorators (@decorator) |
| --- | --- |
| Class definitions | Function definitions and the @ syntax |
| Wrap class instances | Wrap function or class definitions |
| Run time decoration | Compile time decoration |
| Add functionality to instances | Add functionality to functions and classes |
| Specific purpose | General purpose |
| Gang of Four | PEP 318 |
| | https://www.python.org/dev/peps |

# Summary

When to use Decorator?

Add new functionality to existing objects

Better than many subclasses

Better than many properties

Consider using Factory or Builder