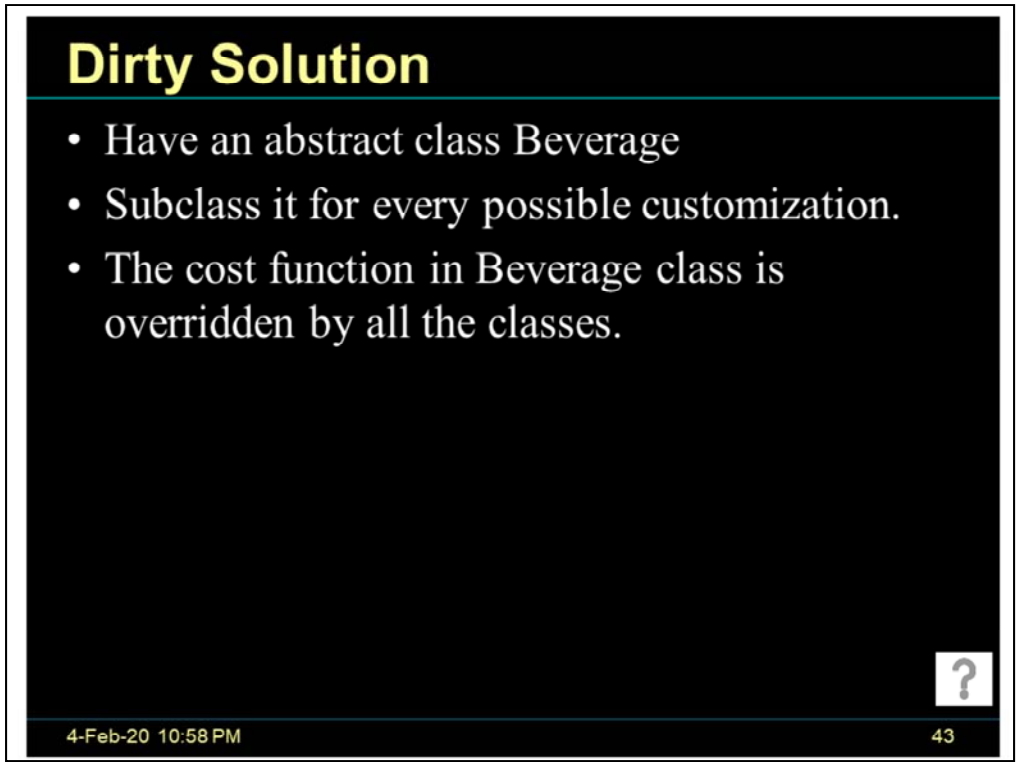

Decorator DP
4-Feb-20 10:58 PM
41

Problem

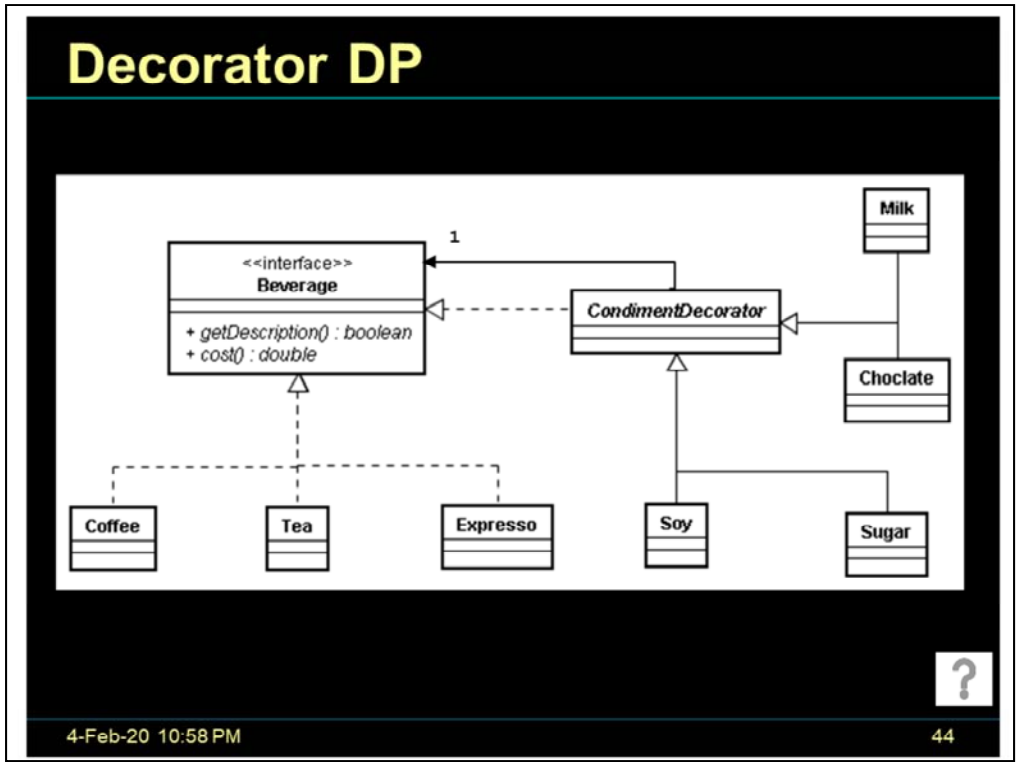
- A coffee house sells coffee/tea in a customized format.
  - some customers want black tea
  - some customers want sugarless tea.
  - some want chocolate in their coffee
  - some want it strong, some want it light
- The price should reflect the choice of the customer.



Problems in this solution:

Class explosion

Difficult to maintain



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

The change in classes should preferably be possible by Composition.

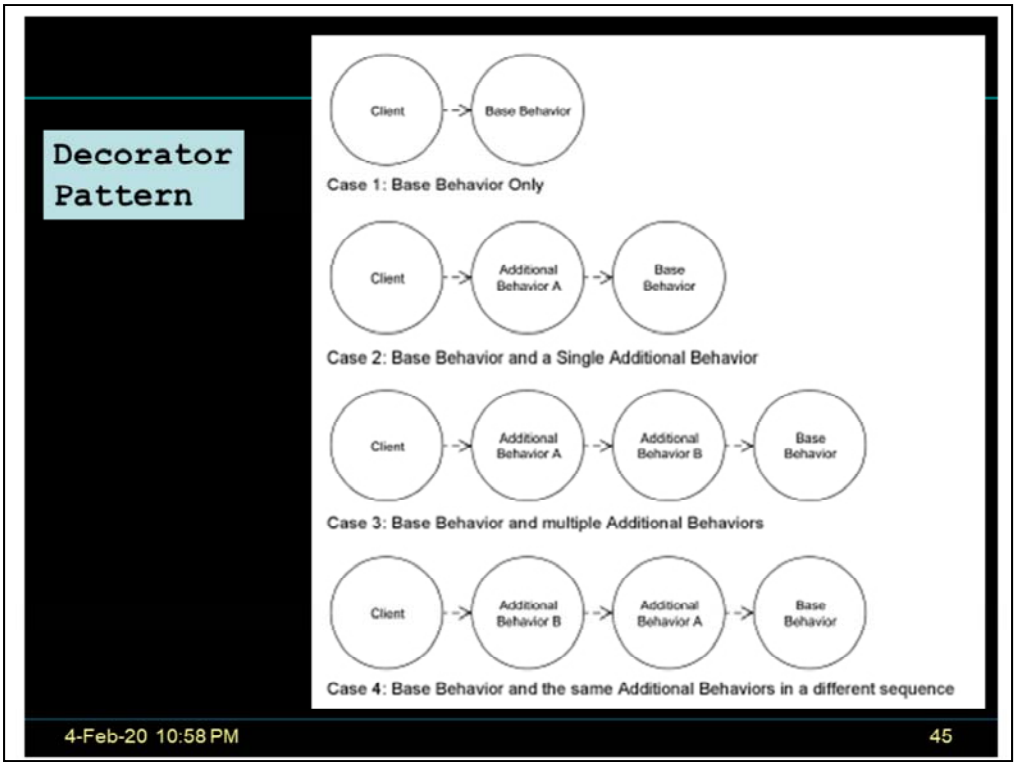
This implies that all member variables of a class should be private.

Limitations of this principle

The above principle should not be applied everywhere.

The above principle should be applied only in the areas that we think will definitely change in future.

Code example



A Decorator is a wrapper for an existing class.

Decorators have the same super type as the objects they decorate.

We can have one or more decorators for an object.

Decorators can be chained.

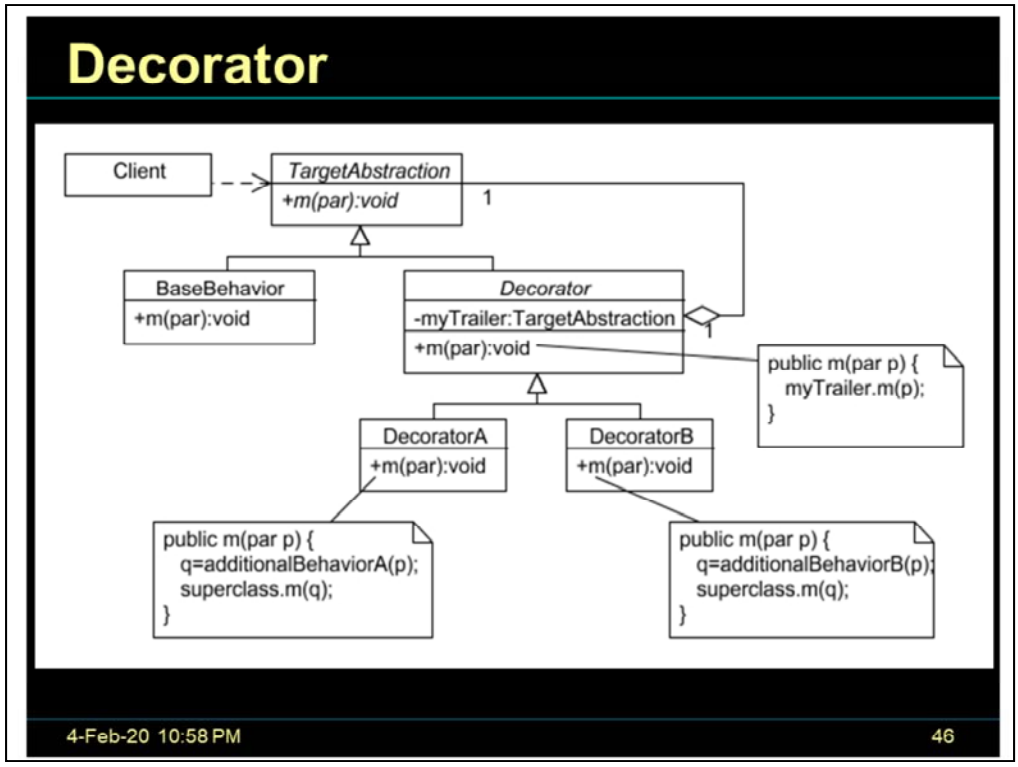
A wrapper can wrap another wrapper and so on.

The decorator adds its own behavior either before and/or after delegating to the object that it decorates to do the rest of the job.

With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them.

Decorators provide a flexible alternative to sub-classing for extending functionality.

We must not rely on object's identity, while using decorators.



If we rely on inheritance, the behavior is determined statically.

With composition, we mix and match the required decorators, at runtime.

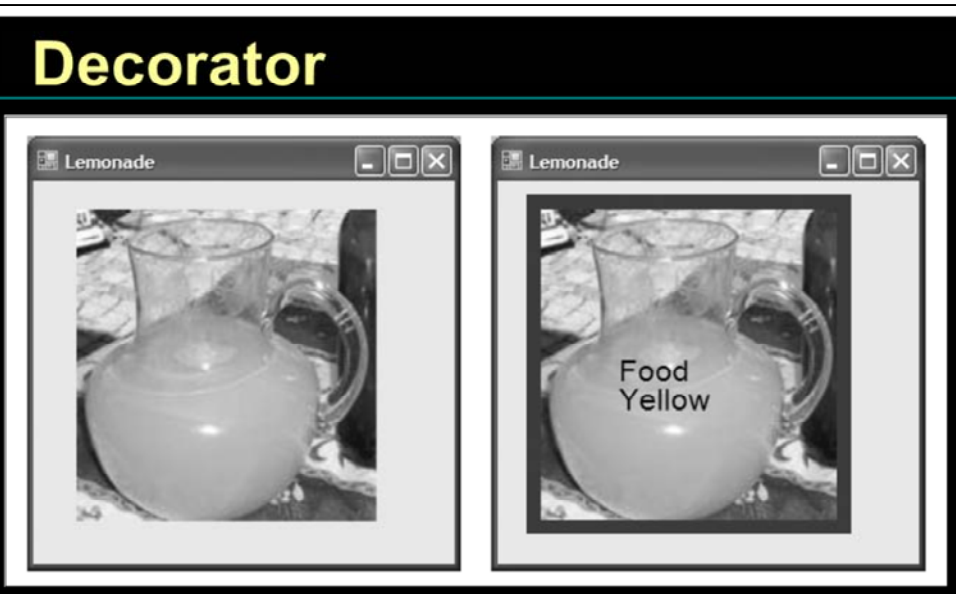
New decorators can be added easily later on.

Non Software Analog

4-Feb-20 10:58 PM

47

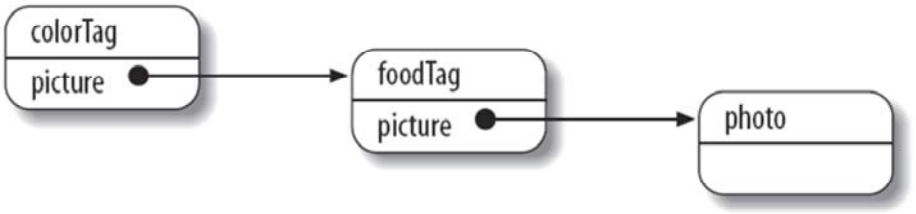
Decorator

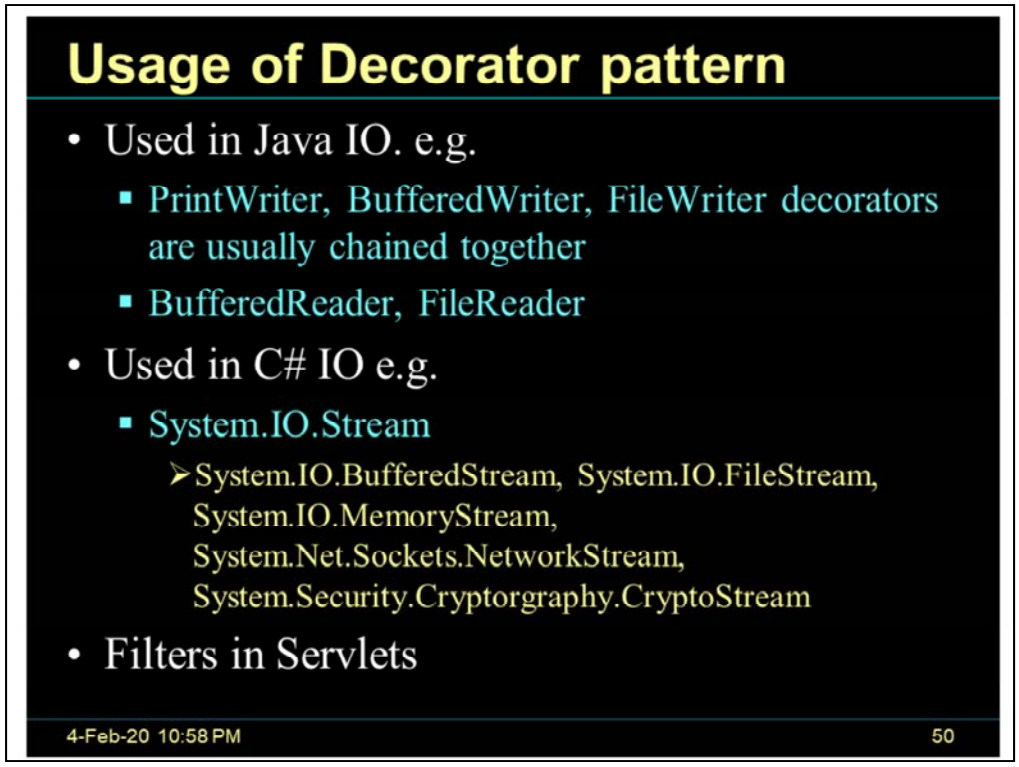


Plain photograph

Photograph with tags for color, contents and frame

Decorator





We use a Decorator pattern ONLY when the flexibility it gives is really needed.

This pattern causes lot of small classes that look alike.

If we need to use “instanceof” operator on a decorated object for the base object, it will fail.

E.g. In C#

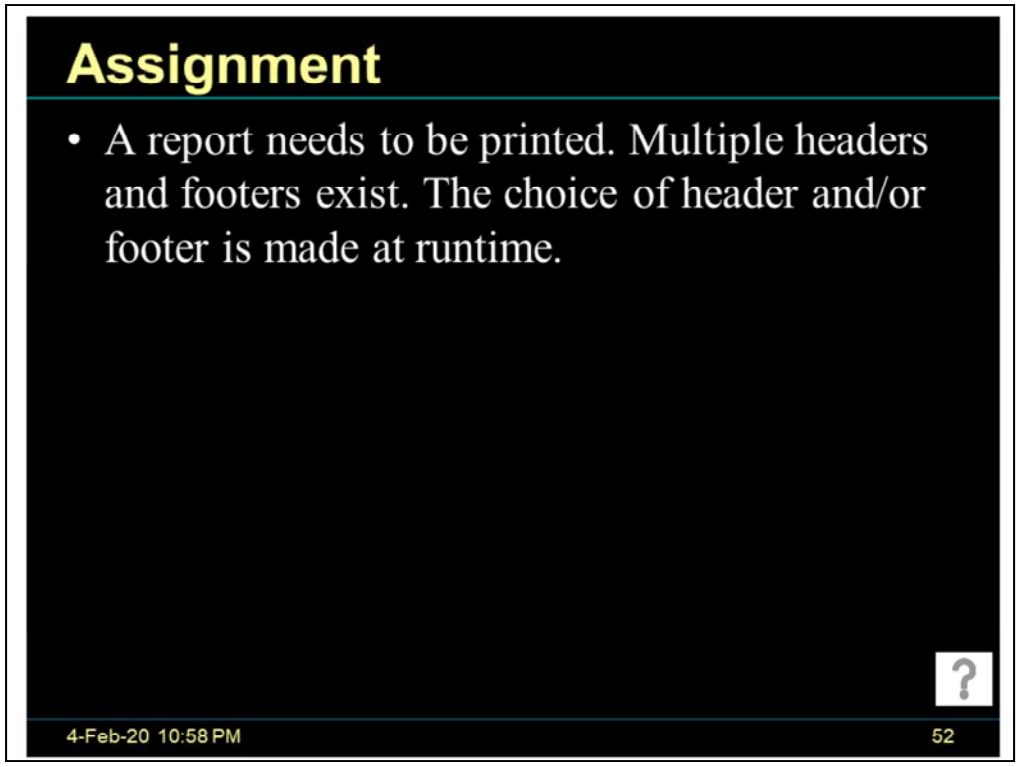
new BufferedStream(new NetworkStream(socket);

new CryptoStream(new memoryStream(bytes), …);

new CryptoStream(new FileSTream(filename, mode), …); //FileStream has buffer built in.

Assignment

- We are writing a rich GUI application.
  - For a text window object, we can have a scrollbar and a border.
  - Different elements in the editor i.e. text, table, paragraphs, etc can have borders or scrollbars.



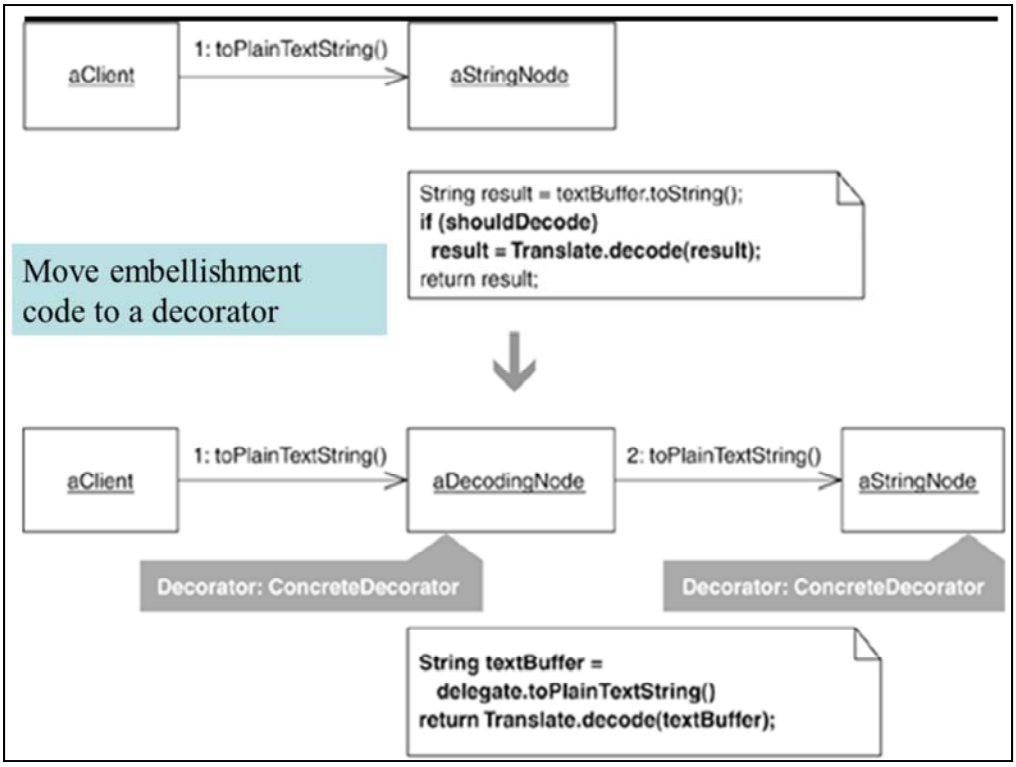
Solution in DecoratorAssignment class in Patterns project

Create a simple decorator system that models the fact that some birds fly and some don’t, some swim and some don’t, and some do both.

Code example

Assignment

- Implement the decorator pattern to create a Pizza restaurant, which has a set menu of choices as well as the option to design our own pizza.
  - Follow the compromise approach to create a menu consisting of a Margherita, Hawaiian, Regina, and Vegetarian pizzas, with toppings (decorators) of Garlic, Olives, Spinach, Avocado, Feta and Pepperdews.
  - Create a Hawaiian pizza decorated with Spinach, Feta, Pepperdews and Olives.



Benefits and Liabilities

+Simplifies a class by removing embellishments from it.

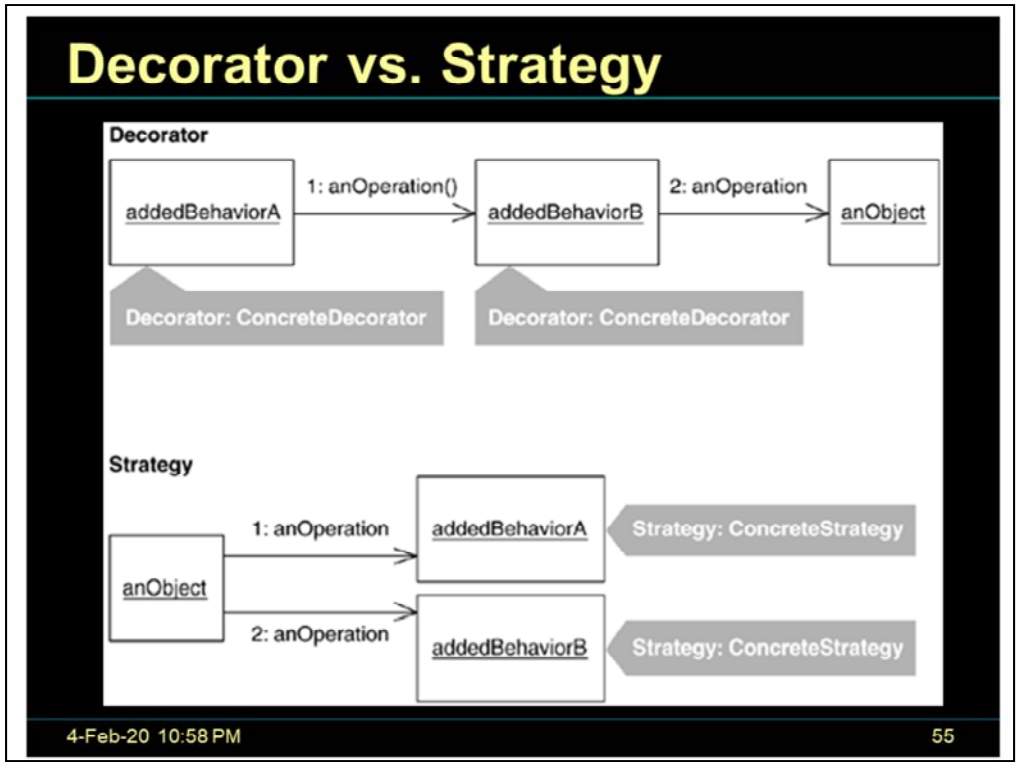
+Effectively distinguishes a class's core responsibility from its embellishments.

+Helps remove duplicated embellishment logic in several related classes.

–Changes the object identity of a decorated object.

–Can make code harder to understand and debug.

–Complicates a design when combinations of Decorators can adversely affect one another.



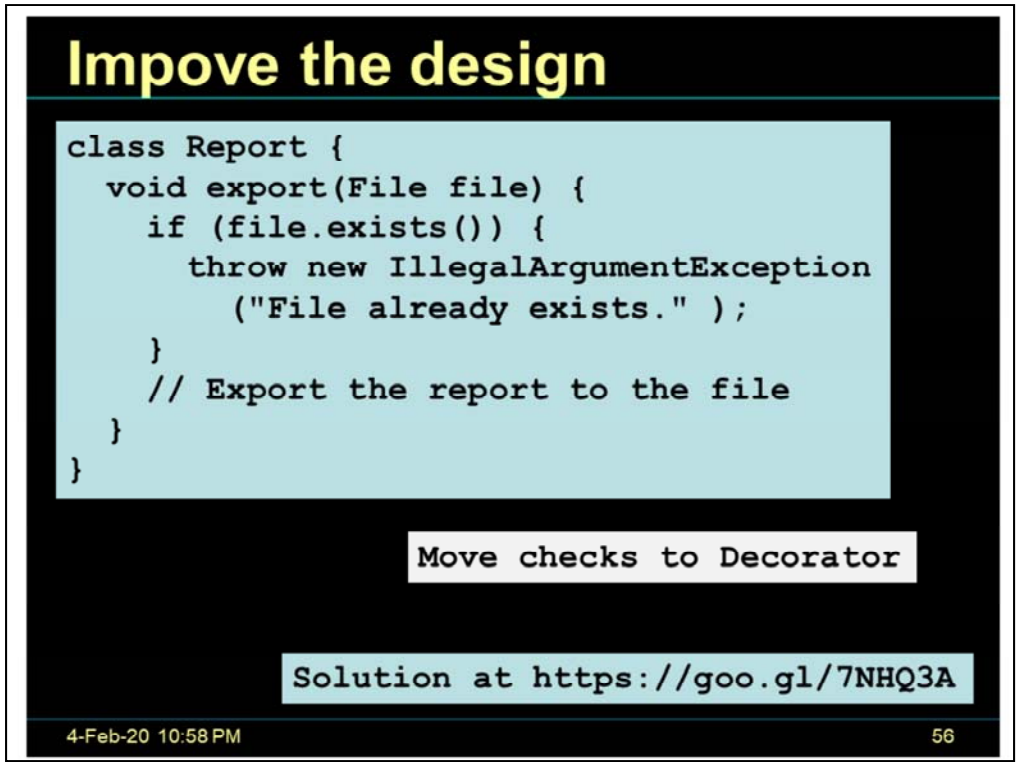
When does it make sense to refactor to Decorator versus to Strategy?

•You can't share Decorator instances—each instance wraps one object. On the other hand, you can easily share Strategy instances by means of the Singleton or Flyweight patterns [DP].

•A Strategy can have whatever interface it wants, while a Decorator must conform to the interface of the classes it decorates.

•Decorators can transparently add behavior to many different classes, as long as the classes share the same interface as the Decorators. On the other hand, classes that want to use Strategy objects must know about their existence and how to use them.

•Using one or more Strategies within a class that holds a lot of data or implements many public methods is common practice. On the other hand, Decorator classes become too heavy and require too much memory when they're used to decorate classes with lots of data and many public methods.




```
interface Report {
  void export(File file);
}
class DefaultReport implements Report {
  @Override void export(File file) {
    // Export the report to the file
  }
}
class NoWriteOverReport implements Report {
  private final Report origin;
  NoWriteOverReport(Report rep) {
    this.origin = rep;
  }
  @Override void export(File file) {
    if (file.exists()) {
      throw new IllegalArgumentException
        ("File already exists." );
```

```
    }
    this.origin.export(file);
  }
}
Report report = new NoWriteOverReport(new DefaultReport());
report.export(file);
```

Details of this example at https://dzone.com/articles/defensive-programming-via-validating-decorators or https://goo.gl/7NHQ3A