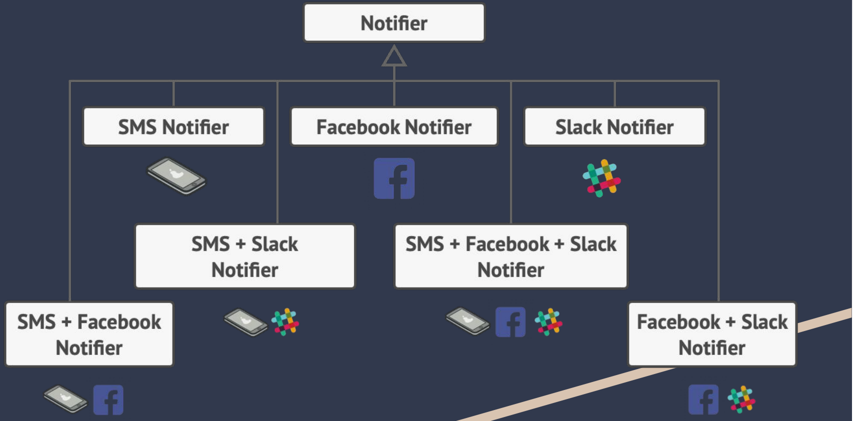
Decorator Design Pattern

By Sean Ouellette and Seth Holtzman

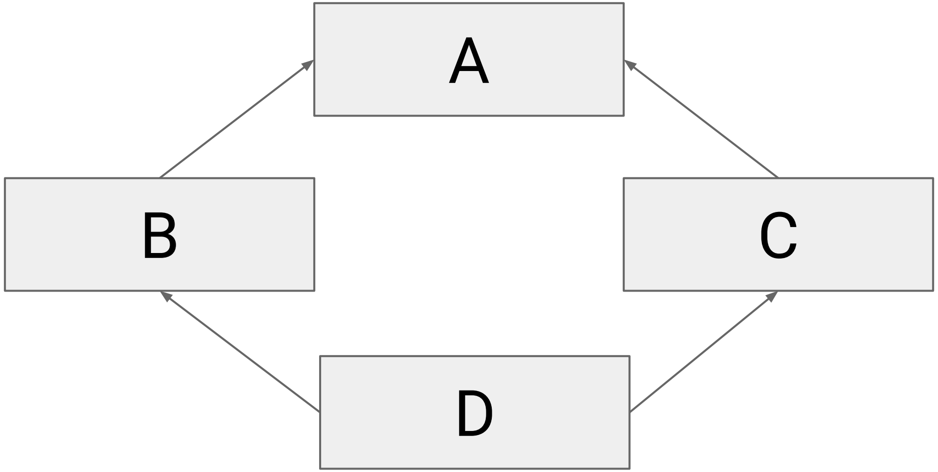
What is it?

- Structural pattern that to add new behavior to an object dynamically at run time.
- Example
    - Get dressed at house(compile time)
    - Walk to class (run-time)
    - Starts raining put on a raincoat(decorator)
    - How to support windy, cold, snowing and raining?

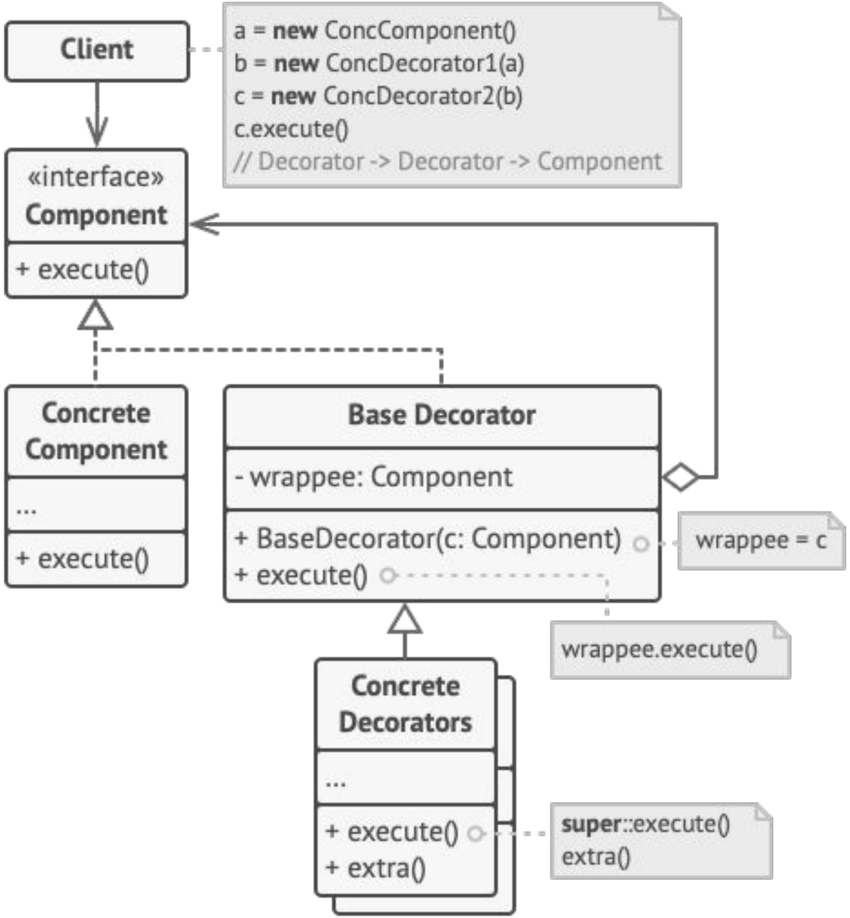
What Issue Does it Solve?



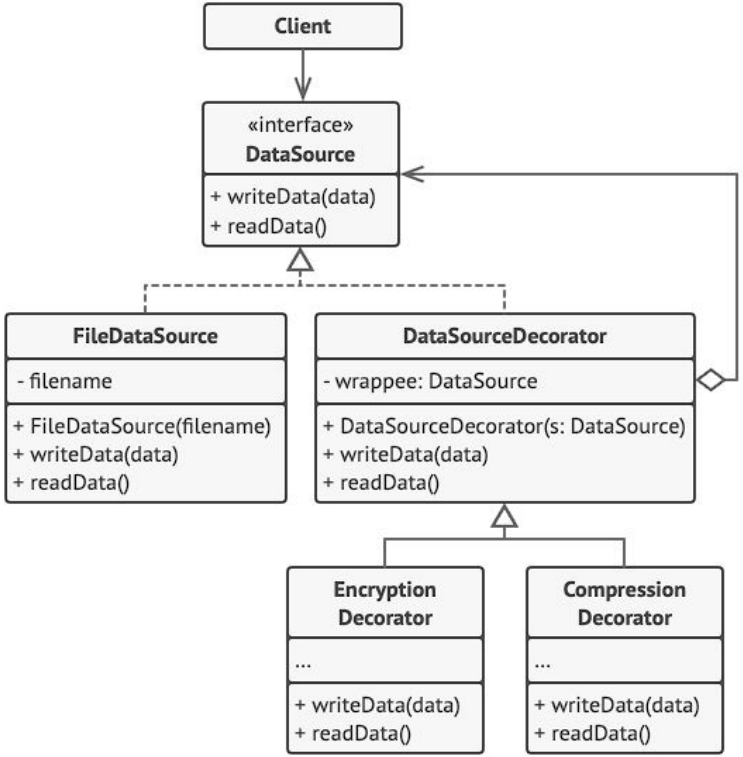
Diamond Problem
(Multiple Inheritance Problem)



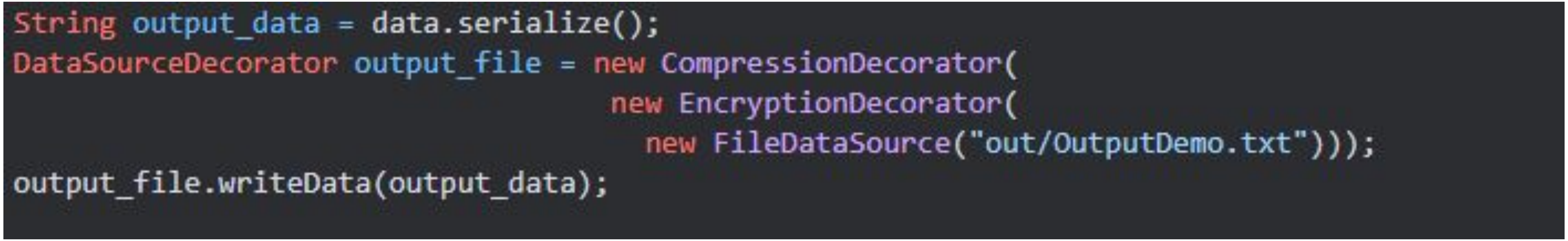
- Solves Redundancy Issues with Objects
- Solves the Diamond Problem
- Solves complex user interactions will the objects codebase


Client
a = **new** ConcComponent()
b = **new** ConcDecorator1(a)
c = **new** ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
«interface»
Component
+ execute()
Concrete Component
...
+ execute()
Base Decorator
- wrappee: Component
+ BaseDecorator(c: Component)
+ execute()
wrappee = c
wrappee.execute()
Concrete Decorators
...
+ execute()
+ extra()
super::execute()
extra()

File output example

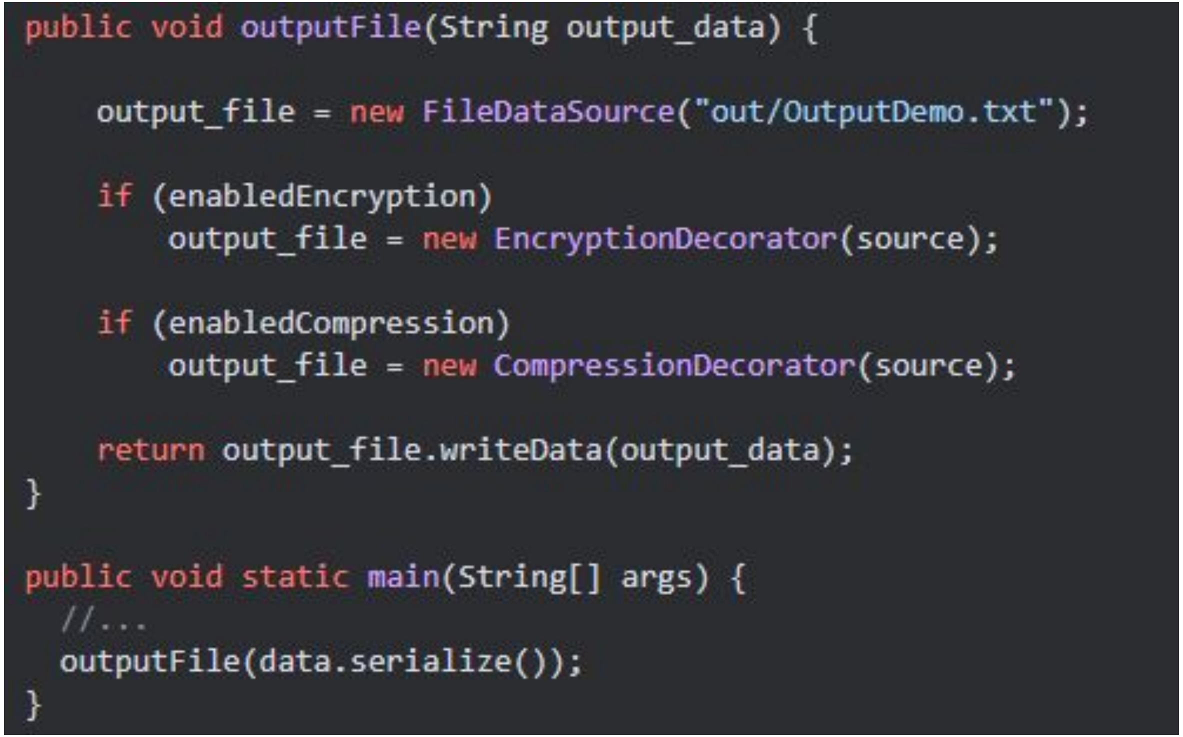


Using Decorators



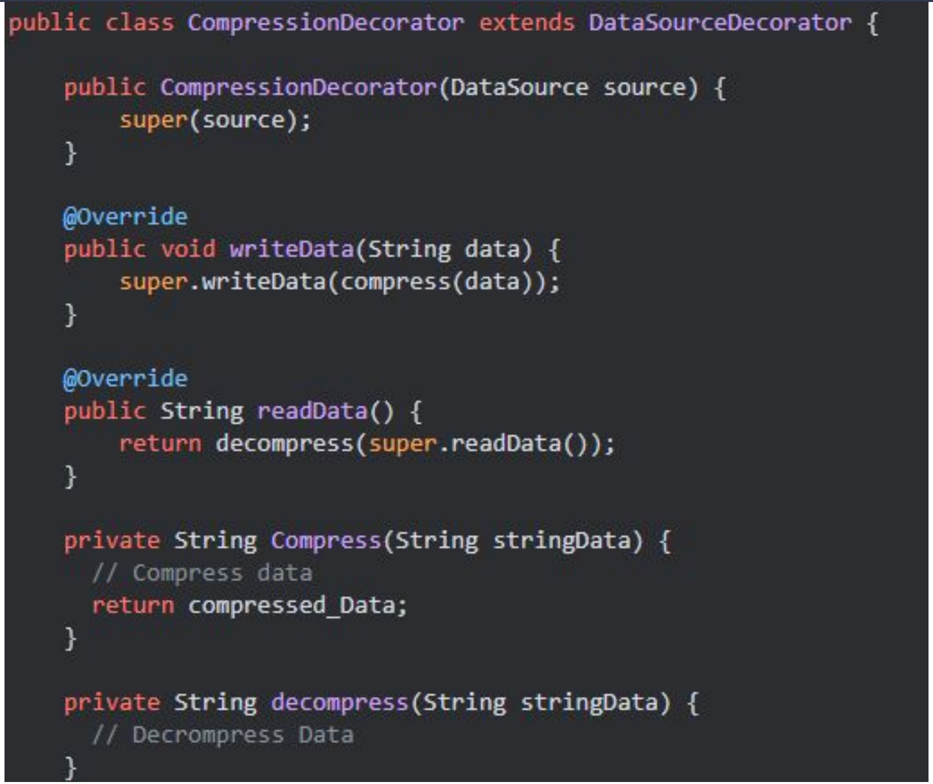
```
String output_data = data.serialize();
DataSourceDecorator output_file = new CompressionDecorator(
                                   new EncryptionDecorator(
                                     new FileDataSource("out/OutputDemo.txt")));
output_file.writeData(output_data);
```

Using Decorators dynamically

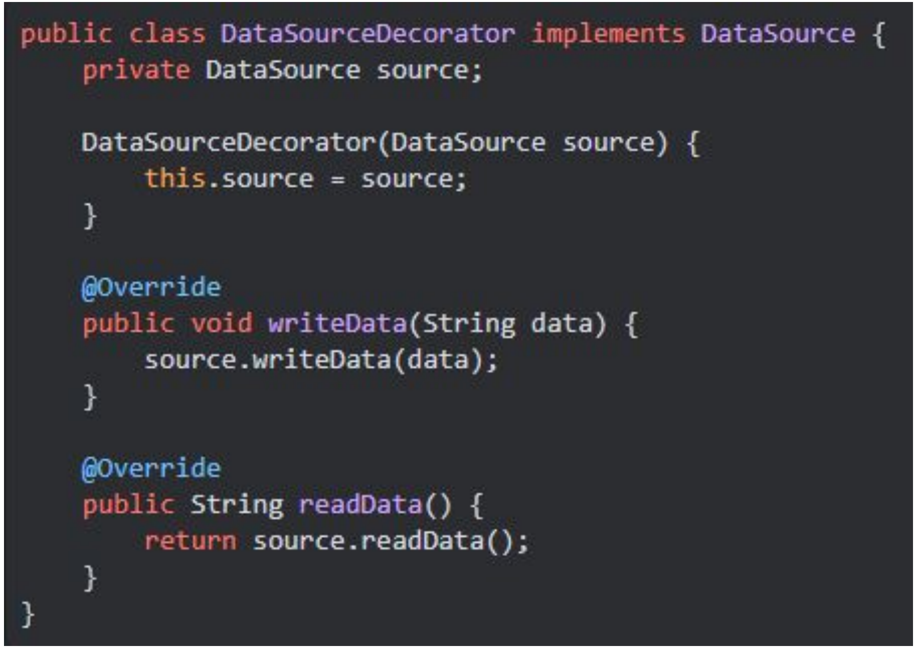


```
public void outputFile(String output_data) {

    output_file = new FileDataSource("out/OutputDemo.txt");

    if (enabledEncryption)
        output_file = new EncryptionDecorator(source);

    if (enabledCompression)
        output_file = new CompressionDecorator(source);

    return output_file.writeData(output_data);
}

public void static main(String[] args) {
  //...
  outputFile(data.serialize());
}
```

How to implement decorators

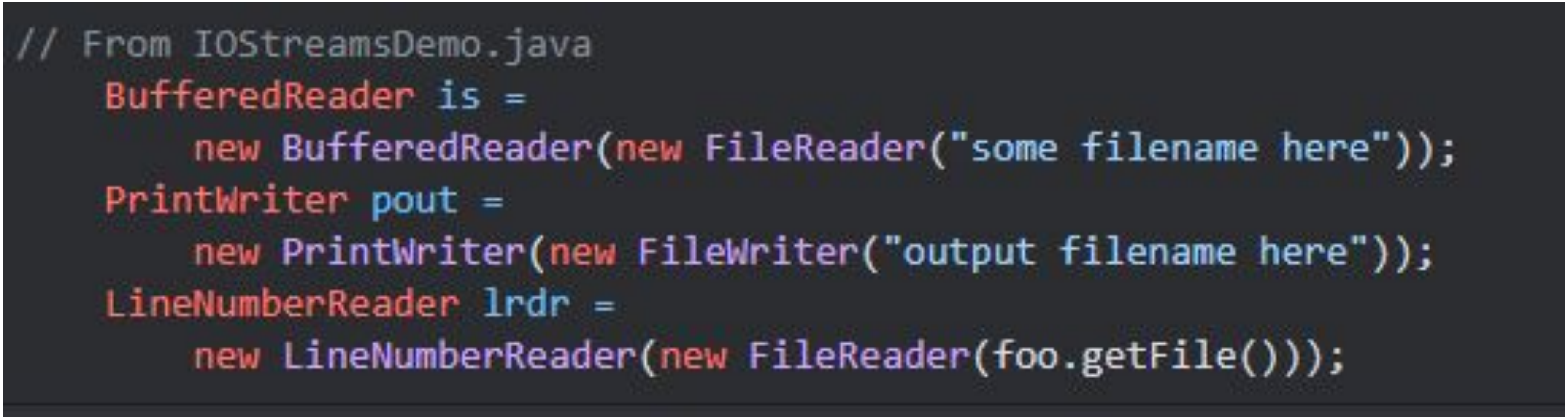


```
public class CompressionDecorator extends DataSourceDecorator {

    public CompressionDecorator(DataSource source) {
        super(source);
    }

    @Override
    public void writeData(String data) {
        super.writeData(compress(data));
    }

    @Override
    public String readData() {
        return decompress(super.readData());
    }

    private String Compress(String stringData) {
      // Compress data
      return compressed_Data;
    }

    private String decompress(String stringData) {
      // Decrompress Data
    }
```



```
public class DataSourceDecorator implements DataSource {
    private DataSource source;

    DataSourceDecorator(DataSource source) {
        this.source = source;
    }

    @Override
    public void writeData(String data) {
        source.writeData(data);
    }

    @Override
    public String readData() {
        return source.readData();
    }
}
```

Decorator in Java IO streams



```java
// From IOStreamsDemo.java
    BufferedReader is =
        new BufferedReader(new FileReader("some filename here"));
    PrintWriter pout =
        new PrintWriter(new FileWriter("output filename here"));
    LineNumberReader lrdr =
        new LineNumberReader(new FileReader(foo.getFile()));
```

Pros:

- Promotes code stability and reduces risk of introducing new bugs
- Granular control reduces the risk of over complicating the object
- Improved testing since they are small isolated components

Cons:

- Over uses of Decorators can lead to a complex hierarchy
- This can result in many different classes making the codebase hard to manage
- Can’t add new data members to the original object
- Not designed carefully can lead to unintended behavior
- Removing decorators is not that straightforward

Sources:

- https://blogs.oracle.com/javamagazine/post/the-decorator-pattern-in-depth
- Pentalog Blog on Decorator Design Pattern
- Diamond Problem Wiki page
- https://refactoring.guru/design-patterns/decorator