Design Patterns (II)

Chung-Kil Hur

(Credit: Byung-Gon Chun & Many Slides from UCB CS169 taught by Armando Fox, David Patterson, George Necula)

SWPP, CSE, SNU

Reference

- https://sourcemaking.com/design_patterns

SOLID OOP principles

(Robert C. Martin, co-author of Agile Manifesto)

- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Injection of dependencies
  - traditionally, Interface Segregation principle
- Demeter principle

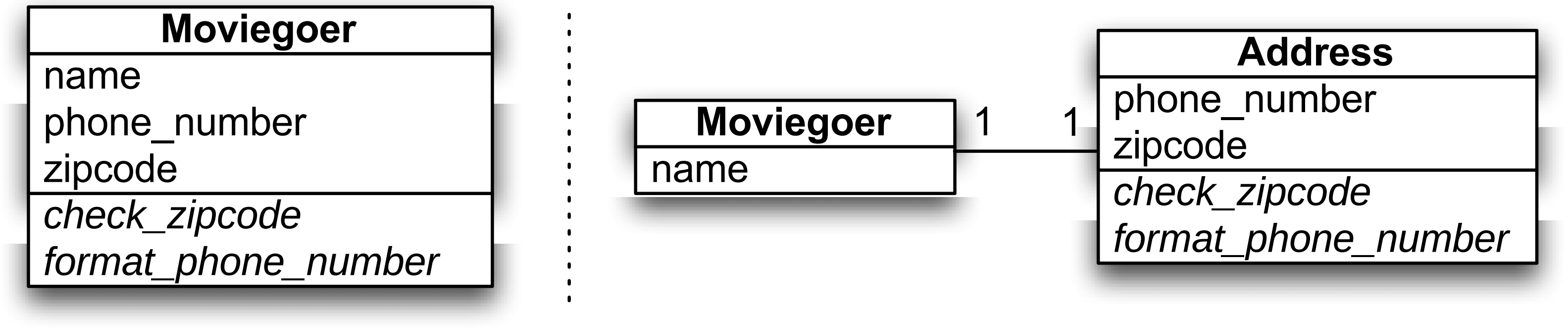
Single Responsibility Principle (SRP)

- A class should have *one and only one* reason to change
  - Each *responsibility* is a possible *axis of change*
  - Changes to one axis shouldn't affect others
- What is class's responsibility, in ≤25 words?
  - Part of the craft of OO design is *defining* responsibilities and then sticking to them
- Models with many sets of behaviors
  - eg a user is a moviegoer, and an authentication principal, and a social network member, ...etc.
  - really big class files are a tipoff

Lack of Cohesion of Methods

- Revised Henderson-Sellers
  LCOM=1–(sum(MV_i) / M*V) (between 0 and 1)
  - M = # instance methods
  - V = # instance variables
  - MV_i = # instance methods that access the *i*' th instance variable (excluding "trivial" getters/setters)
- LCOM-4 counts # of connected components in graph where related methods are connected by an edge
- High LCOM suggests possible SRP violation

Extract Class Refactoring



Which is true about a class's observance of the Single Responsibility Principle?

- ☐ In general, we would expect to see a correlation between poor cohesion score and poor SOFA metrics
- ☐ Low cohesion is a possible indicator of an opportunity to extract a class
- ☐ If a class respects SRP, its methods probably respect SOFA
- ☐ If a class's methods respect SOFA, the class probably respects SRP

Open/Closed Principle

```
public class Report {
  public void output(ReportData data) {
    switch (format) {
    case HTML:
      new HtmlFormatter().output(data);
    case PDF:
      new PdfFormatter().output(data);
    default: // no op
    }
  }
}
```

Open/Closed Principle

- Classes should be *open for extension*, but *closed for **source** modification*

```
public class Report {
  public void output(ReportData data) {
    switch (format) {
    case HTML:
      new HtmlFormatter().output(data)
    case PDF:
      new PdfFormatter().output(data)
    default: // no op
    }
  }
}
```

- Can’t extend (add new report types) without changing Report base class

Abstract Factory Pattern (#1): DRYing out construction

- How to avoid OCP violation in Report constructor, if output type isn't known until runtime?

- Statically typed language: *factory* pattern (#1)

```
public class FormatterFactory {
  public static Formatter newInstance(
      FormatterType formatterType) {
    switch (formatterType) {
    case HTML:
      return new HtmlFormatter();
    case PDF:
      return new PdfFormatter();
    default:
      // return code here…
    }
  }
}
```

```
public class Report {
  public void output(ReportData data) {
    FormatterFactory.newInstance(
      formatterType).output(data);
  }
}
```

Case 1

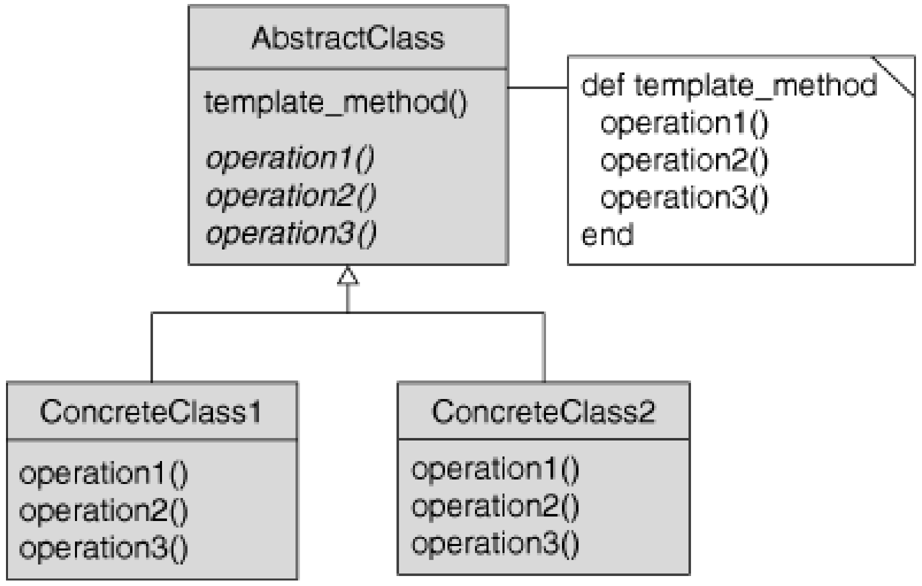
```
class Report {
 Formatter formatter;
 void output() {
   //call formatter method(s)
 }
}
```

Define Formatter, HtmlFormatter, and PdfFormatter

The implementation of each step in output may differ, but the set of steps is the same for all variants of the formatter. In this case, the steps are header(), body(), and footer().

Template Method Pattern (#2)

- *Template method*: set of steps is the same, but implementation of steps different
  - Inheritance: subclasses override abstract "step" methods

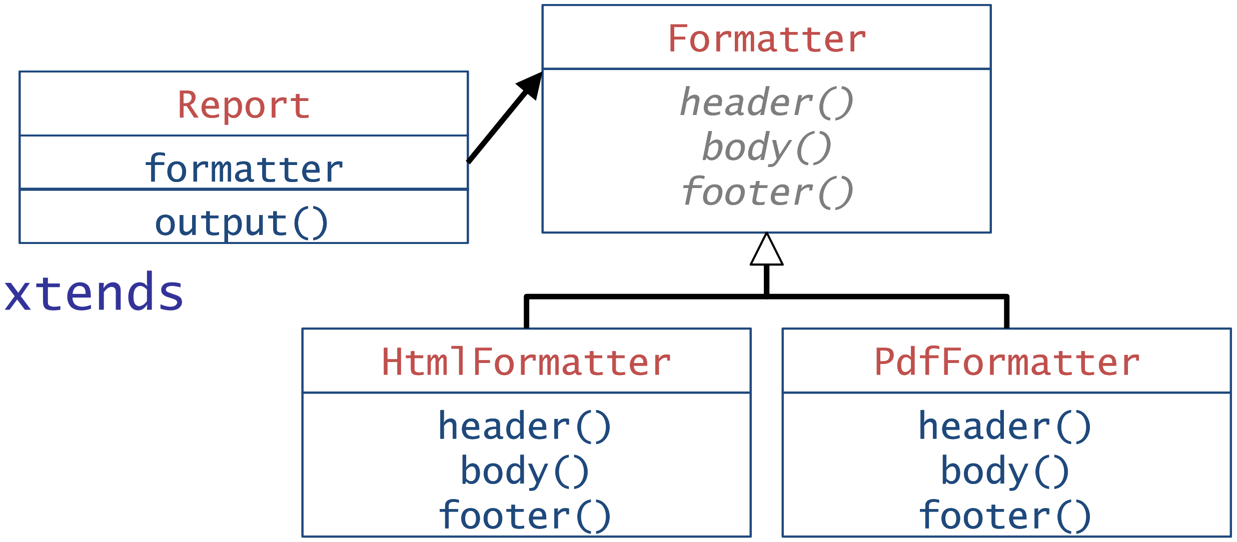


Report Generation Using Template

```
class Report {
 Formatter formatter;
 void output() {
   formatter.header();
   formatter.body();
   formatter.footer();
 }
}
```

Template method stays the same;
helpers overridden in subclass

```
class HtmlFormatter extends
Formatter {
 void header() {...}
 void body() {...}
 void footer() {...}
}
class PdfFormatter extends
Formatter {
 void header() {...}
 void body() {...}
 void footer() {...}
}
```



Case 2

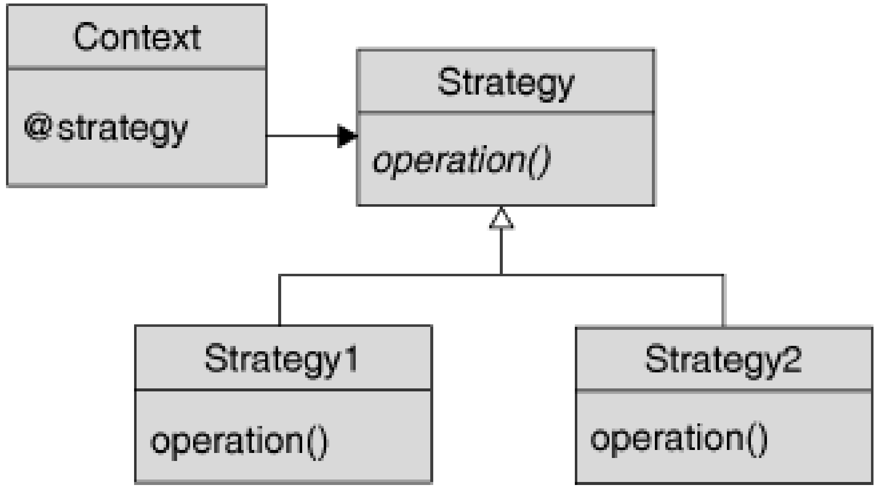
```
class Report {
 Formatter formatter;
 void output() {
   //call a formatter method
 }
}
```

Define Formatter, HtmlFormatter, and PdfFormatter

The overall task is the same, but the set of steps may be different in each variant of the formatter.
In HtmlFormatter, the steps are header(), cssStyles(), and body().
In PdfFormatter, the steps are header(), body(), and pdfTrailer().

Strategy Pattern (#3)

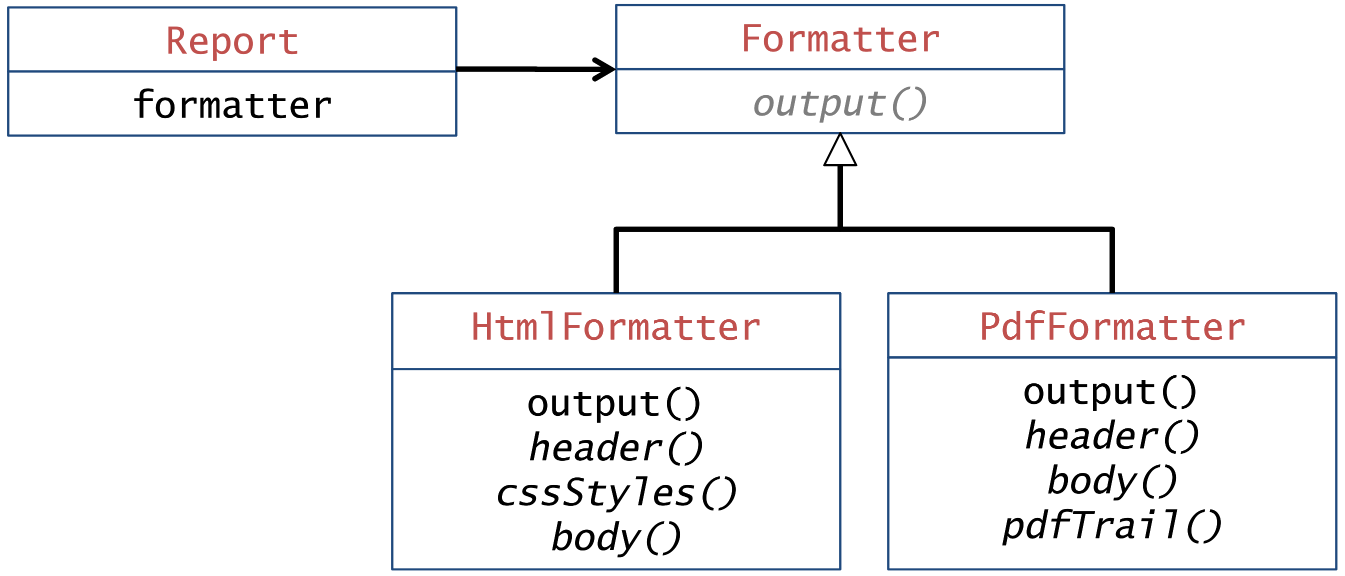
- *Strategy:* task is the same, but many ways to do it
  - composition: component classes implement whole task



Report Generation Using Strategy

```
class Report {
 Formatter formatter;
 void output() {
  formatter.output();
 }
}
```

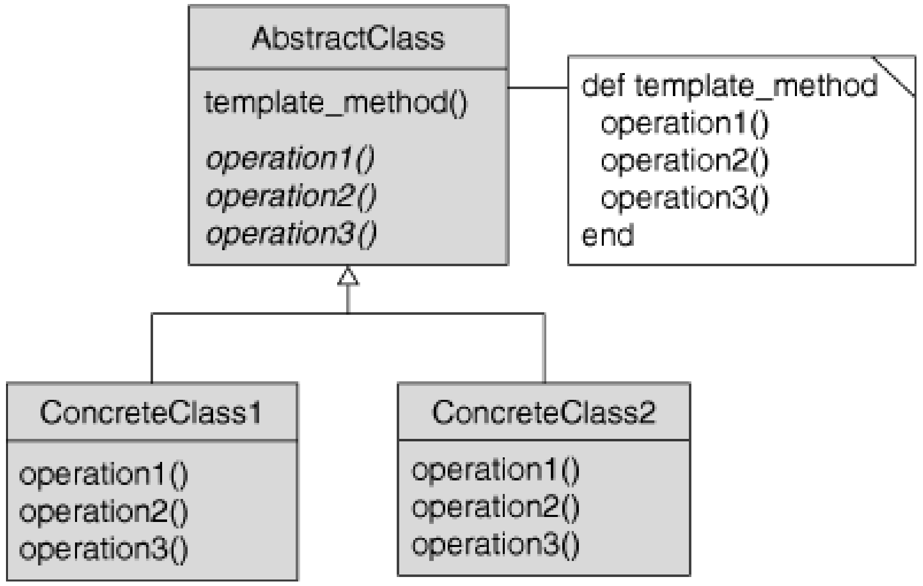
Delegation
(vs. inheritance)

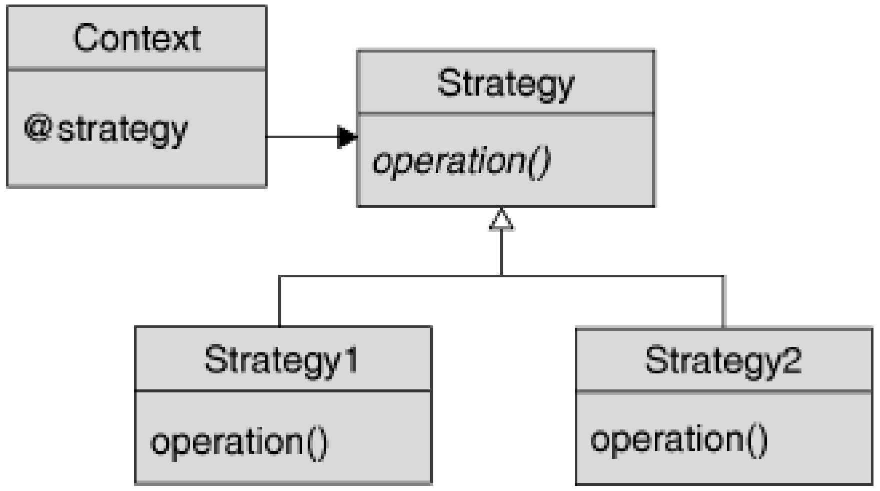


"Prefer composition over inheritance"

Template Method Pattern (#2) & Strategy Pattern (#3)

- *Template method*: set of steps is the same, but implementation of steps different
  - Inheritance: subclasses override abstract "step" methods
- *Strategy:* task is the same, but many ways to do it
  - composition: component classes implement whole task





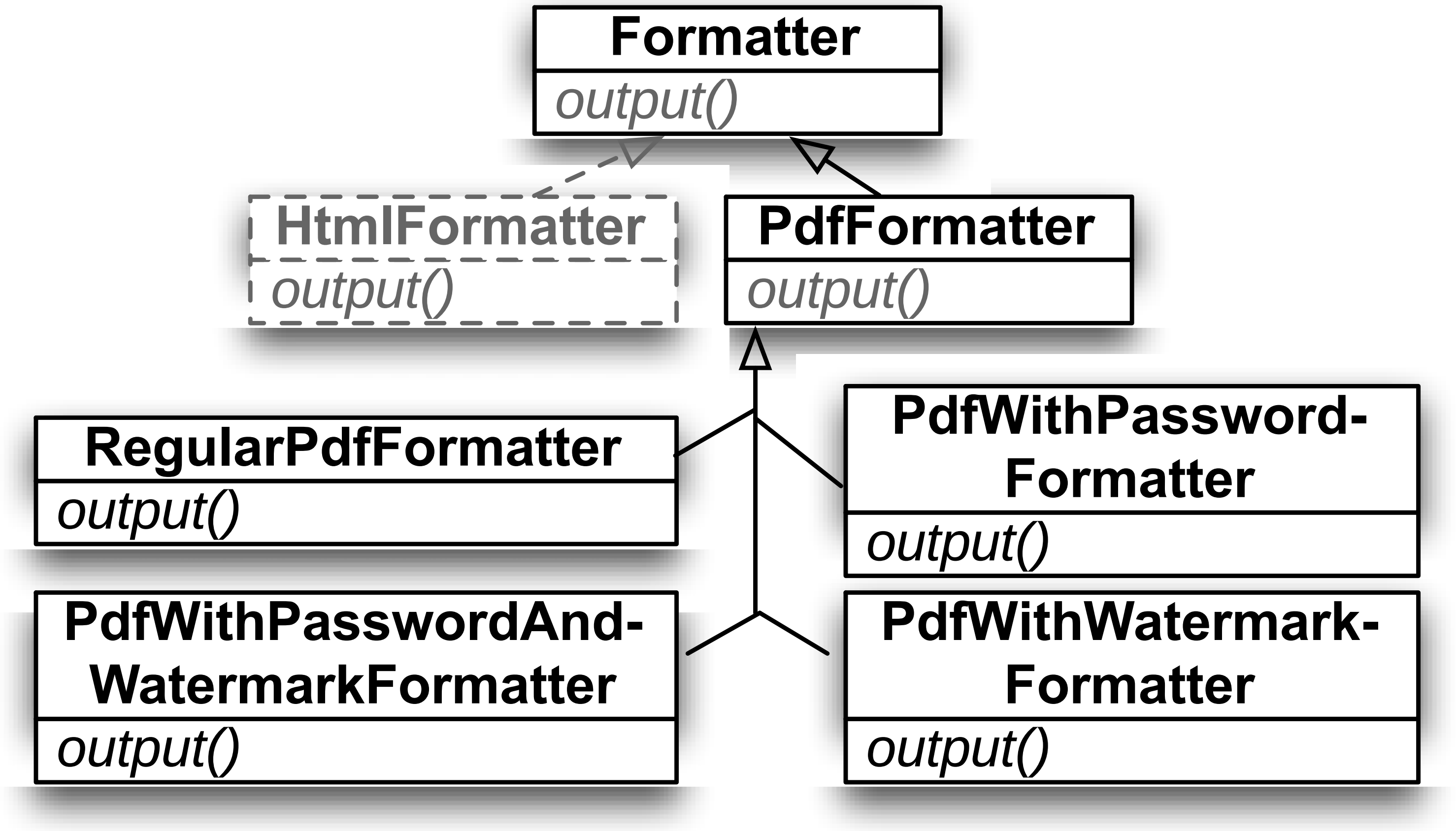
A Different Kinds of OCP Violation

- A different kinds of OCP violation when we want to *add* behaviors to an existing class and discover that we cannot do so without modifying it

Q. Adding Features to PdfFormatter

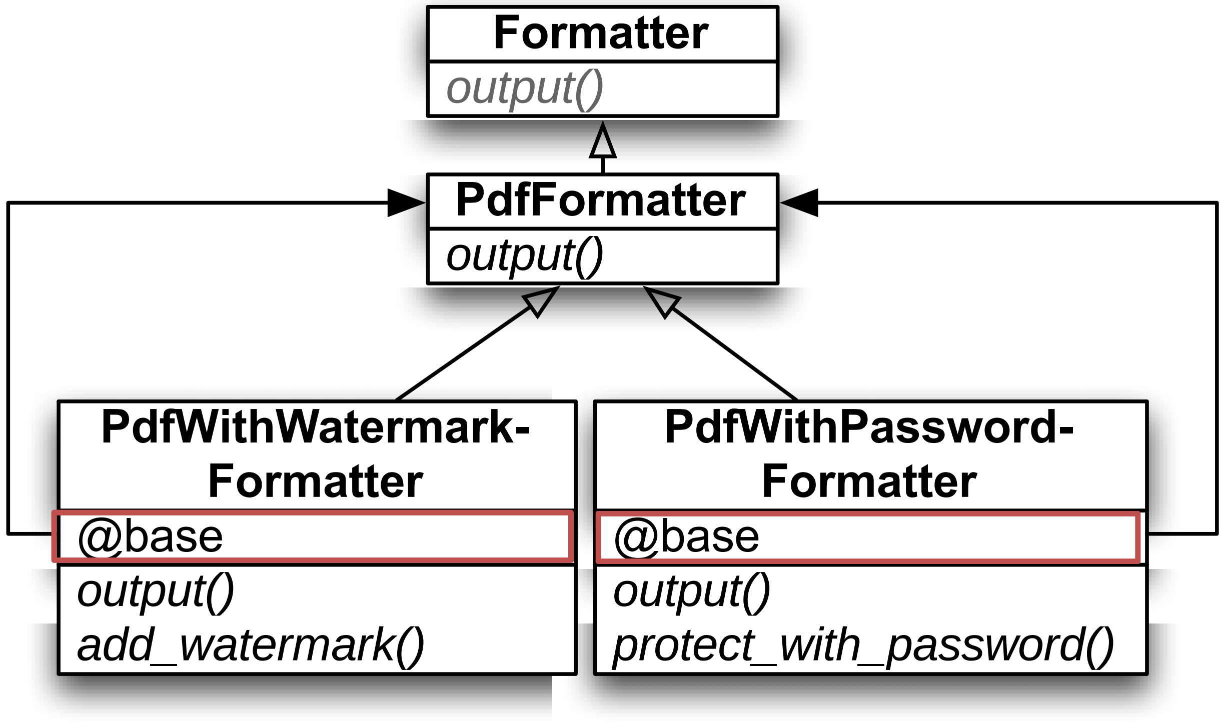
- PDF files can be generated with or without password protection (Password) and with or without a “Draft” watermark across the background (Watermark)

Take 1. Inheritance?



Decorator Pattern (#4): DRYing Out Extension Points

Decorate a class or method by wrapping it in an enhanced version that has the same API, allowing us to compose multiple decorations as needed



```
// Decorator
public class PdfWithWatermarkFormatter extends
PdfFormatter {

  private final PdfFormatter base;

  public PdfWithWatermarkFormatter(PdfFormatter
base) {
    this.base = base;
  }
  // wrap it in an enhanced version
  public void output() {
    base.output();
    add_watermark();
  }

  private void add_watermark() { ...
  }
}
```

OCP In Practice

- Can't close against *all* types of changes, so have to choose, and you might be wrong
- Agile methodology can help *expose important types of changes early*
  - Scenario-driven design with prioritized features
  - Short iterations
  - Test-first development
- Then you can try to close against *those types* of changes

OmniAuth defines a handful of RESTful endpoints your app must provide to handle authentication with a variety of third parties. To add a new auth provider, you create a gem that works with that provider. Which statement is FALSE about OmniAuth?

- ☐ OmniAuth is itself compliant with OCP
- ☐ Using OmniAuth helps your app follow OCP (with respect to 3rd-party authentication)
- ☐ OmniAuth is an example of the Template pattern
- ☐ OmniAuth is an example of the Strategy pattern