CSC 440/540: Software Engineering

GoF Design Patterns
Topics

1. GoF Design Patterns
2. Template Method
3. Strategy
4. Composite
5. Adapter
Design Patterns Book

- Classic text that started Design Patterns movement written by the Gang of Four (GoF).
- We’ll introduce several widely used patterns from the book.
- Useful solutions for certain problems, but if you don’t have the problem, don’t use the pattern.
# GoF Design Patterns Catalog

<table>
<thead>
<tr>
<th>Behavioral</th>
<th>Creational</th>
<th>Structural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter</td>
<td>Factory</td>
<td>Adapter</td>
</tr>
<tr>
<td>Template method</td>
<td>Abstract Factory</td>
<td>Bridge</td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td>Builder</td>
<td>Composite</td>
</tr>
<tr>
<td>Command</td>
<td>Prototype</td>
<td>Decorator</td>
</tr>
<tr>
<td>Iterator</td>
<td>Singleton</td>
<td>Façade</td>
</tr>
<tr>
<td>Mediator</td>
<td></td>
<td>Flyweight</td>
</tr>
<tr>
<td>Memento</td>
<td></td>
<td>Proxy</td>
</tr>
<tr>
<td>Observer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strategy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visitor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CSC 440: Software Engineering

Slide #4
GoF patterns are specializations and/or combinations of the more fundamental GRASP design patterns.

- Example: Adapter Pattern provides Protected Variations through use of Indirection and Polymorphism
Template Method

- **Pattern Name:** Template Method
- **Problem:** How to implement an algorithm that varies in certain circumstances?
- **Solution:** Define the skeleton of the algorithm in the superclass, deferring implementation of the individual steps to the subclasses.
class Report
  def output_report(format)
    if format == :plain
      puts(“*** #{@title} ***”)
    elsif format == :html
      puts(“<html><head>”)
      puts(“<title>#{@title}</title>”)
      puts(“</head>”)
    else
      raise “Unknown format”
    end
    # similar code for body, ending, etc.
  end
end
c
lass Report
def output_report(format)
    output_start()
    output_head()
    output_body()
    output_end()
end
end
class HTMLReport < Report
  def output_start
    puts("html")
  end
  
  def output_head()
    puts("<head><title>#{@title}</title></head>")
  end

  def output_body()
    puts("<body>...</body>")
  end

  def output_end()
    puts("</html>")
  end
end
class PlainTextReport < Report
  def output_start
    end
  def output_head()
    puts("*** #{@title} ***")
    end
  def output_body()
    puts("...")
    puts("...")
    end
  def output_end
    end
end
**Pattern Name:** Strategy

**Problem:** How to design for varying, but related, algorithms or policies without using inheritance?

**Solution:** Write each algorithm in its own class, with a common interface and use algorithm classes via object composition.
class Formatter
  def output_report(title, text)
    raise "Abstract method called."
  end
end
class HTMLFormatter < Formatter
  def output_report(title, text)
    puts("<html>")
    puts("<head><title>#{title}</title></head>")
    puts("<body>#{text}</body>")
    puts("</html>")
  end
end
PlainText Formatter Class

class PlainTextFormatter < Formatter
  def output_report(title, text)
    puts(“*** #{title} ***”)
    puts(text)
  end
end
POS Strategy Example

- How do we handle different sale pricing policies in the POS system?
  - 10% off
  - $10 off if over $200
- Solution: Create multiple SalePricingStrategy classes, each with a polymorphic `getTotal()` method that implements the particular policy described by that class.
Strategy Example

```java
public class PercentDiscountPricingStrategy {
    private float percentage;

    public PercentDiscountPricingStrategy(float percentage) {
        this.percentage = percentage;
    }

    @Override
    public Money getTotal(Sale s) {
        return s.getPreDiscountTotal() * percentage;
    }
}

public class AbsoluteDiscountOverThresholdPricingStrategy {
    private Money discount;
    private Money threshold;

    public AbsoluteDiscountOverThresholdPricingStrategy(Money discount, Money threshold) {
        this.discount = discount;
        this.threshold = threshold;
    }

    @Override
    public Money getTotal(Sale s) {
        Money pdt = s.getPreDiscountTotal();
        if (pdt < threshold) {
            return pdt;
        } else {
            return pdt - discount;
        }
    }
}
```
Context Objects

- A strategy object is attached to a context object—the object to which it applies its algorithm.
  - In this example, the context object is a Sale.
  - The context object is often passed as an argument to the strategy object as an argument, e.g. self or this.
- When a `getTotal()` method is sent to a Sale, it delegates some of the work to its strategy object.
Collaboration of Sale and Strategy Objects

\[ t = \text{getTotal}(s) \]

\[ \text{pdt} = \text{getPreDiscountTotal} \]

\[ \text{st} = \text{getSubtotal} \]

\[ t = \text{pdt} \times \text{percentage} \]

Note that the Sale \( s \) is passed to the Strategy so that it has parameter visibility to it for further collaboration.
Collaboration of Sale and Strategy Objects

Sale

date
...

getTotal()
...

getTotal()
{
...
return pricingStrategy.getTotal(this)
}

PercentDiscount
PricingStrategy

percentage : float

getTotal(Sale) : Money

AbsoluteDiscount
OverThreshold
PricingStrategy

discount : Money
threshold : Money

getTotal(Sale) : Money
Discussion

- Strategy provides Protected Variations with respect to changing algorithms.
- Strategy is a specialization of Polymorphism.
- Passing the context object as an argument enables easy collaboration and visibility, but it also increases coupling.
Composite

- **Pattern Name:** Composite
- **Problem:** How to treat a group of objects the same way (polymorphically) as a non-composite (leaf or atomic) object? How to create a tree of objects?
- **Solution:** Define classes for composite and leaf objects that implement identical interfaces.
Composite Example

![Composite Example Diagram]

- Manufacture Cake
  - Make Cake
    - Make Batter
    - Fill Pan
    - Bake
    - Frost
  - Package Cake
    - Box
    - Label
  - AddDryIngredients
  - AddLiquids
  - Mix
class Task
  def initialize(name)
    @name = name
  end
  def name
    return @name
  end
  def get_time_required
    return 0.0
  end
end
Example Leaf Class

class MixTask < Task
  def initialize(name)
    super("Mix the batter")
  end
  def get_time_required
    return 3.0
  end
end
Example Composite Class

class MakeBatterTask < Task
  def initialize(name)
    super(“Make batter”)
    @sub_tasks = Array.new
    add_sub_task(AddLiquidsTask.new)
    add_sub_task(MixTask.new)
  end

  def get_time_required
    time = 0.0
    @sub_tasks.each { |t| time += t.get_time_required } 
    return time
  end
end
Example Composite Class

class MakeBatterTask < Task
  def add_sub_task(task)
    @sub_tasks.push(task)
  end

  def remove_sub_task(task)
    @sub_tasks.delete(task)
  end
end
POS Composite Example

Problem: How do we handle the case of multiple, conflicting pricing policies?

- 20% senior discount
- preferred customer discount of 15% of sales over $400
- today there is a $50 off purchases over $500
- buy 1 case of Darjeeling, get 15% discount off everything
Composite Example

- Pricing strategies attach to sale by virtue of:
  - Time period.
  - Customer type.
  - A particular line item.
- Need to define a conflict resolution strategy.
- How can we modify design to Sale object doesn’t know if there are one or many pricing strategies and also provide for conflict resolution?
POS Composite Example

Add new `CompositePricingStrategy` class

- Abstract class.
- Has two concrete subclasses that provide for different conflict resolution strategies:
  - `CompositeBestForCustomerPricingStrategy`
  - `CompositeBestForStorePricingStrategy`
- Polymorphically provides `getTotal()` method, so it can be called like any other `PricingStrategy`.
- `add(PricingStrategy)` method allows set of `PricingStrategy` objects to be composed together.
### Example

#### Percentage Discount Pricing Strategy
- percentage : float
- `getTotal( Sale ) : Money`

#### Absolute Discount Over Threshold Pricing Strategy
- discount : Money
- threshold : Money
- `getTotal( Sale ) : Money`

#### Composite Pricing Strategy
- `add( ISalePricingStrategy )`
- `getTotal( Sale ) : Money`

#### Composite Best For Customer Pricing Strategy
- `getTotal( Sale ) : Money`

#### Composite Best For Store Pricing Strategy
- `getTotal( Sale ) : Money`

---

```java
{...
  return pricingStrategy.getTotal( this )
}
```

```java
{ return sale.getPreDiscountTotal() * percentage }
```

```java
{ lowestTotal = INTEGER.MAX
  for each ISalePricingStrategy strat in pricingStrategies
  { total := strat.getTotal( sale )
   lowestTotal = min( total, lowestTotal )
  }
  return lowestTotal
}
```

---

All composites maintain a list of contained strategies. Therefore, define a common superclass `CompositePricingStrategy` that defines this list (named `strategies`).
POS Composite Sequence

**Composite Best For Customer Pricing Strategy**

\[
\text{Sales Line Item} = \text{getTotal} (s) \]

\[
\text{Sale} = \text{getSubtotal} \]

\[
\text{loop} \]

\[ t = \text{getTotal} (s) \]

\[
\text{loop} \]

\[ x = \text{getTotal} (s) \]

The Sale object treats a Composite Strategy that contains other strategies just like any other ISalePricingStrategy.

**UML**: ISalePricingStrategy is an interface, not a class; this is the way in UML 2 to indicate an object of an unknown class, but that implements this interface.
Discussion

- Core of Composite pattern is:
  - Outer composite object contains a list of inner objects.
  - Both inner and outer objects implement the same interface.
- Sale object cannot distinguish whether it’s using an atomic or composite PricingStrategy object.
Adapter

- **Pattern Name:** Adapter
- **Problem:** How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
- **Solution:** Convert the original interface of a component into another interface, through an intermediate adapter object.
Adapters use interfaces and polymorphism to add a level of indirection to varying APIs in other components.
Adapter Example Sequence Diagram

Register : SAP Accounting Adapter

postSale(sale)

SOAP over HTTP

«actor» : SAP System

the Adapter adapts to interfaces in other components

makePayment
Updating the Domain Model

- Adapter design introduced a new domain concept: the `TaxLineItem`
- Discovery of new domain concepts is natural in an iterative process.
- Update Domain Model only if you will be using it in future iterations.
References