Introduction to Software Technology
5. Design Patterns

Klaus Ostermann
Topics of this Lecture

- Design Patterns
  - Template
  - Strategy
  - Bridge
  - Decorator
- These design patterns are less general than the GRASP patterns
  - They focus on specific design problems
- These are some of the most common and most important classical design patterns in OO design
The goal is to separate... policies from detailed mechanisms. invariant and variant parts.

Abstract classes...
define interfaces.
implement high-level policies.

Control sub-class extensions.
Avoid code duplication.
The Template Method Pattern is at the core of the design of object-oriented frameworks.
Using the Template Method Pattern for Bubble-Sort

```java
public abstract class BubbleSorter {
    protected int length = 0;
    protected void sort() {
        if (length <= 1) return;
        for (int nextToLast = length - 2; nextToLast >= 0; nextToLast --)
            for (int index = 0; index <= nextToLast; index++)
                if (outOfOrder(index)) swap(index);
    }
    protected abstract void swap(int index);
    protected abstract boolean outOfOrder(int index);
}
```
public class IntBubbleSorter extends BubbleSorter{
    private int[] array = null;
    public void sort(int[] theArray) {
        array = theArray;
        length = array.length;
        super.sort();
    }
    protected void swap(int index) {
        int temp = array[index];
        array[index] = array[index+ 1];
        array[index+1] = temp;
    }
    protected boolean outOfOrder(int index) {
        return(array[index] > array[index+ 1]);
    }
}
Discussion

![Diagram showing hierarchy of BubbleSorter, IntBubbleSorter, and DoubleBubbleSorter with a question mark.]
Discussion

- Template method forces detailed implementations to extend the template class.
- Detailed implementation depend on the template.
- Cannot re-use detailed implementations' functionality. (E.g., swap and out-of-order are generally useful.)
- If we want to re-use the handling of integer arrays with other sorting strategies we must remove the dependency
  - this leads us to the Strategy Pattern.
Strategy Pattern

- **Intent**
  - Define a *family of algorithms*, encapsulate each one, and make them *interchangeable*. Strategy lets the algorithm vary independently from clients that use it.

- **Comparison With Template**
  - Using the strategy pattern, both - the template and the detailed implementations - depend on abstractions.

- **Basic Structure**
Strategy Pattern: Example

```
  "Clients"
    BubbleSorter
    QuickSorter

  "interface"
  SortHandle

  IntSortHandle
  DoubleSortHandle
```

Einführung in die Softwaretechnik
Define a family of algorithms, encapsulate each one, and make them interchangeable
Strategy Pattern: Discussion

- Use if many related classes differ only in their behavior rather than implementing different related abstractions.
  - Strategies allow to configure a class with one of many behaviors.
- Use if you need different variants of an algorithm.
  - Strategies can be used when variants of algorithms are implemented as a class hierarchy.
- Use if a class defines many behaviors that appear as multiple conditional statements in its operations.
  - Move related conditional branches into a strategy
Strategy vs Subclassing

- Sub-classing Context mixes algorithm’s implementation with that of Context. Context harder to understand, maintain, extend.
- When using sub-classing we can't vary the algorithm dynamically.
- Sub-classing results in many related classes. Only differ in the algorithm or behavior they employ.
- Encapsulating the algorithm in Strategy...
  - lets you vary the algorithm independently of its context.
  - makes it easier to switch, understand, and extend the algorithm.
Passing Context Information to Strategy

- The Strategy interface is shared by all concrete Strategy classes whether the algorithms they implement are trivial or complex.
- Some concrete strategies won't use all the information passed to them
  - Simple concrete strategies may use none of it.
  - Context creates/initializes parameters that never get used.
- If this is an issue use a tighter coupling between Strategy and Context; let Strategy know about Context
Passing Context Information to Strategy

- Two possible strategies:
  - Pass the needed information as a parameter.
    - Context and Strategy decoupled
    - Communication overhead
    - Algorithm can’t be adapted to specific needs of context
  - Context passes itself as a parameter or Strategy has a reference to its Context.
    - Reduced communication overhead
    - Context must define a more elaborate interface to its data
    - Closer coupling of Strategy and Context.
    - Avoid closer coupling by defining an explicit interface for retrieving context, which is implemented by the context
Bridge Pattern: Motivation

*We want to support multiple operating systems...*

```
<table>
<thead>
<tr>
<th>Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>GnomeWindow</td>
</tr>
</tbody>
</table>
```

*We want to provide different types of windows...*

```
<table>
<thead>
<tr>
<th>Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
</tr>
</tbody>
</table>
```
Which alternative would be better represented using inheritance?
What technique can we use to provide both types of classifications?
Bridge Pattern

- **Intent**
  - Decouple an abstraction from its implementation so that the two can vary independently.

- **Structure**
By encapsulating the concept that varies we can avoid problems with inheritance conflicts.

This is very similar to the technique used in the Strategy pattern.
Bridge Pattern: Discussion

- **Decoupling interface and implementation:**
  - Implementation can be configured at run-time.
  - Implementation being used is hidden inside abstraction.

- **Improved extensibility**
  Abstraction and Implementer hierarchies can be extended independently.

- **Issues**
  - Most languages do not support parallel hierarchies very well
    - Type safety problems
Decorator Pattern

- **Intent**
  - We need to add responsibilities to existing individual objects
  - ... dynamically and transparently, without affecting other objects.
  - ... responsibilities can be withdrawn dynamically.

- **Problem:** Extension by subclassing is not practical:
  - Large number of independent extensions are possible.
  - Would produce an explosion of subclasses to support every combination.
  - No support for dynamic adaptation.
  - A class definition may be hidden or otherwise unavailable for subclassing
  - Cannot change all constructor calls to the class whose object are to be extended
Limitations of Inheritance: Example

Evolution:
Adding functionality to a ByteArrayInputStream to read whole sentences and not just single bytes.
Limitations of Inheritance: Example

Evolution:
We also want to have the possibility to read whole sentences using FileInputs...
After the n-th iteration...

- Problems:
  - ... a new class for each responsibility.
  - responsibility mix fixed statically. (How about PipedDataBufferedInputStream?)
  - non-reusable extensions; code duplication;
  - maintenance nightmare: exponential growth of number of classes
Multiple Inheritance is no Solution Either

- static responsibility mix
- naming conflicts
- hard to dispatch super calls correctly

“Multiple inheritance is good, but there is no good way to do it.”
A. SYNDER
Structure of the Decorator Pattern

Intent: We need to add responsibilities to existing individual objects dynamically and transparently, without affecting other
java.io abstracts various data sources and destinations, as well as processing algorithms:

- Programs **operate on stream objects** ...
- **independently of** ultimate data source / destination / shape of data.
Decorator Pattern: Discussion

- Decorator enables more flexibility than inheritance:
- Responsibilities can be added/removed at run-time.
- Different Decorator classes for a specific Component class enable to mix and match responsibilities.
- Easy to add a responsibility twice; e.g., for a double border, attach two BorderDecorators
- Decorator avoids incoherent classes:
  - feature-laden classes high up in the hierarchy pay-as-you-go approach: don't bloat, but extend using fine-grained Decorator classes
    - functionality can be composed from simple pieces.
    - an application does not need to pay for features it doesn't use.
Decorator: Problems

- Lots of little objects
  - A design that uses Decorator often results in systems composed of lots of little objects that all look alike.
  - Objects differ only in the way they are interconnected, not in their class or in the value of their variables. Imagine a class to draw a border around a component..
  - Such systems are easy to customize by those who understand them, but can be hard to learn and debug.

- Object identity
  - A decorator and its component aren't identical. From an object identity point of view, a decorated component is not identical to the component itself.
  - You shouldn't rely on object identity when you use decorators
Example: Streams in java.io

- A stream is normally addressed via the outermost Decorator.
- Sometimes, a reference to one of the internal objects is maintained and operated on
  - operation shouldn’t include actual reads or writes
  - good style: all read() operations are performed only to the head decorator in the composite stream object
- Reading from an internal object breaks the illusion of a single object accessed via a single reference, and makes the code more difficult to understand.

```java
FileInputStream fin = new FileInputStream("a.txt");
BufferedInputStream din = new
BufferedInputStream(fin);
fin.read();
```
Decorator: Problems

- No late binding
  - DELEGATION VERSUS FORWARD SEMANTICS

Forwarding with binding of this to method holder; "ask" an **object to do something on its own**.

Binding of this to message receiver: “ask” an object to **do something on behalf of the message receiver**.
No Late Binding: Example

Account
{abstract}

type : String

getType() : String

printHistory()

CheckingAccount

printHistory()

... getType(); ...

SavingsAccount

printHistory()

... getType(); ...

OnlineAccount

getType() : String

printHistory()

... getType(); ...

account

account.printHistory();
Decorator: Problems

- Need to implement forwarding methods for those methods not relevant to the decorator
  - A lot of repetitive programming work
  - A maintenance problem: What if the decorated class changes
    - Adding new methods or removing methods that are irrelevant to the decorators
    - Decorator classes need to change as well
    - This is a variant of the so-called “fragile base class problem”
Decorator: Issues

- Keep the common class (Component) lightweight:
  - it should focus on defining an interface (e.g. implemented as interface).
  - defer defining data representation to subclasses.
  - otherwise the complexity of Component might make the decorators too heavyweight to use in quantity.
- Putting a lot of functionality into Component makes it likely that subclasses will pay for features they don't need.

- These issues require pre-planning.
  - Difficult to apply decorator pattern to 3rd-party component class.

Chap. 9, 14, 15, 18