Principles of Software Construction: Objects, Design and Concurrency

Design Patterns and Java I/O

Jonathan Aldrich       Charlie Garrod
Design Patterns

• "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”
  – Christopher Alexander

• Every Composite has its own domain-specific interface
  ▪ But they share a common problem and solution
Example: Composite Windows

- **Problem**
  - Express a part-whole hierarchy of shapes
  - Allow treating a group of shapes just like shapes

- **Consequences**
  - Makes clients simple; they can ignore the difference
  - Easy to add new kinds of shapes
Elements of a Pattern

• Name
  ▪ Important because it becomes part of a design vocabulary
  ▪ Raises level of communication

• Problem
  ▪ When the pattern is applicable

• Solution
  ▪ Design elements and their relationships
  ▪ Abstract: must be specialized

• Consequences
  ▪ Tradeoffs of applying the pattern
    • Each pattern has costs as well as benefits
    • Issues include flexibility, extensibility, etc.
    • There may be variations in the pattern with different consequences
Example: Composite Windows

- **Problem**
  - Express a part-whole hierarchy of shapes
  - Allow treating a group of shapes just like shapes

- **Consequences**
  - Makes clients simple; they can ignore the difference
  - Easy to add new kinds of shapes

```
for s in components:
    s.draw()
```
Composite Pattern

- **Problem (generic)**
  - Express a part-whole hierarchy of components
  - Allow treating a composite just like a component

- **Consequences (generic)**
  - Makes clients simple; they can ignore the difference
  - Easy to add new kinds of components
  - Can be overly general – uniformity is not always good

```python
for c in components:
    c.operation()
```
Example: Shape Change Notification

- **Problem**
  - Drawing changes from shape to shape, but updating doesn’t - want to reuse updating code
  - Future shape implementations should not forget to update

- **Consequences**
  - Code reuse
  - Authors of subclasses will not unintentionally forget to do the update
Template Method Pattern

**Problem (generic)**
- Express an algorithm with varying and invariant parts
- When common behavior should be factored and localized
- When subclass extensions should be limited

**Consequences (generic)**
- Code reuse
- Inverted “Hollywood” control: don’t call us, we’ll call you
- Invariant algorithm parts are not changed by subclasses
The Template Method Pattern in the Virtual World

• Did you use a template method in the Virtual World assignment? How and why?

• Let’s look at the code...

• For more details, see the Piazza post “How to Reuse Code in hw2”
Problem: Line Breaking Implementations

• Context: document editor

• Many ways to break a paragraph into lines
  ▪ Blind: just cut off at 80 columns
  ▪ Greedy: fit as many words in this line, then wrap
  ▪ Global (e.g. TeX): minimize badness in entire paragraph
    • Might move a small word to next line if it reduces extra spaces there

• Option 1: We could put this in class Paragraph
  ▪ But this is not Paragraph’s main function
  ▪ Putting many algorithms into Paragraph makes it too big
  ▪ Other classes might need line breaking, too
  ▪ Adding new line breaking algorithms is difficult

• Option 2?
Option 2: Encapsulate the Line Breaking Strategy

- **Problem**
  - Paragraphs needs to break lines in different ways
  - Want to easily change or extend line breaking algorithm
  - Want to reuse algorithm in new places

- **Consequences**
  - Easy to add new line breaking strategies
  - Separates strategy → vary strategy, paragraph independently
  - Adds objects and dynamism → code harder to understand
**Strategy Pattern**

- **Problem (generic)**
  - Behavior varies among instances of an abstraction
  - An abstraction needs different variants of an algorithm

- **Consequences (generic)**
  - Easy to add new strategies (e.g. compared to conditionals)
  - Separates algorithm $\rightarrow$ vary algorithm, context independently
  - Adds objects and dynamism $\rightarrow$ code harder to understand
  - Fixed strategy interface $\rightarrow$ high overhead for some impls.
The Strategy Pattern in the Virtual World

• Did you see the strategy pattern in the Virtual World assignment? How and why?

• Let’s look at the code...
Tradeoffs

void sort(int[] list, String order) {
    ...
    boolean mustswap;
    if (order.equals("up")) {
        mustswap = list[i] < list[j];
    } else if (order.equals("down")) {
        mustswap = list[i] > list[j];
    }
    ...
}

void sort(int[] list, Comparator cmp) {
    ...
    boolean mustswap;
    mustswap = cmp.compare(list[i], list[j]);
    ...
}

interface Comparator {
    boolean compare(int i, int j);
}

class UpComparator implements Comparator {
    boolean compare(int I, int j) { return i<j; }
}

class DownComparator implements Comparator {
    boolean compare(int I, int j) { return i>j; }
}
Fundamental OO Design Principles

• Patterns emerge from fundamental principles applied to recurring problems
  ▪ Design to interfaces
  ▪ Favor composition over inheritance
  ▪ Find what varies and encapsulate it

• Patterns are discovered, not invented
  ▪ Best practice by experienced developers
**Fundamental Principles underlying the Strategy Pattern**

- **Design to interfaces**
  - Strategy: the algorithm interface

- **Favor composition over inheritance**
  - Strategy could be implemented with inheritance
    - Multiple subclasses of Context, each with an algorithm
    - Drawback: couples Context to algorithm, both become harder to change
    - Drawback: can’t change algorithm dynamically

- **Find what varies and encapsulate it**
  - Strategy: the algorithm used

- **Side note: how do you implement the Strategy pattern in functional languages?**
Kinds of Patterns

• Categories
  ▪ Structural – vary object structure
  ▪ Behavioral – vary the behavior you want
  ▪ Creational – vary object creation

• Derived from scenarios

• UML diagram credit: Pekka Nikander
  ▪ http://www.tml.tkk.fi/~pnr/GoF-models/html/
Patterns to Know

• Façade, Adapter, Composite, Strategy, Bridge, Abstract Factory, Factory Method, Decorator, Observer, Template Method, Singleton, Command, State, Proxy, and Model-View-Controller

• Know pattern name, problem, solution, and consequences
Java Streams – and their Patterns

• What is System.out? Let's look at the Javadoc
System.out is a java.io.PrintStream

• **java.io.PrintStream**: Allows you to conveniently print common types of data

  ```java
  void close();
  void flush();
  void print(String s);
  void print(int i);
  void print(boolean b);
  void print(Object o);
  ...
  void println(String s);
  void println(int i);
  void println(boolean b);
  void println(Object o);
  ...
  ```
Let’s look at the stream design
The fundamental I/O abstraction: a stream of data

- `java.io.InputStream`
  ```java
  void close();
  abstract int read();
  int read(byte[] b);
  ```

- `java.io.OutputStream`
  ```java
  void close();
  void flush();
  abstract void write(int b);
  void write(byte[] b);
  ```

- Aside: If you have an `OutputStream` you can construct a `PrintStream`:
  ```java
  PrintStream(OutputStream out);
  PrintStream(File file);
  PrintStream(String filename);
  ```
  ...
Design Problem: how to add functionality to streams?

- We could do lots of things to a stream of data
  - Compress it
  - Encrypt it
  - Compute (or check) a checksum or digest
  - Translate it
  - (your idea here)

- It’s unreasonable to add all this functionality explicitly to OutputStream

- What can we do instead?
The Decorator Pattern

```
Component
+ operation()

ConcreteComponent
+ operation()

Decorator
+ operation()

ConcreteDecoratorA
- addedState
+ operation()

ConcreteDecoratorB
# newAfterBehavior()
# newBeforeBehavior()
+ operation()
```

```
component.operation()
```

```
newBeforeBehavior() super.operation() newAfterBehavior()
```

```
1
```
Structural: Decorator

- **Applicability**
  - To add responsibilities to individual objects dynamically and transparently
  - For responsibilities that can be withdrawn
  - When extension by subclassing is impractical

- **Consequences**
  - More flexible than static inheritance
  - Avoids monolithic classes
  - Breaks object identity
  - Lots of little objects
FilterOutputStream as a Decorator
Why “Decorator?”

- Origins in GUIs
- Imagine you have a window that can display a lot of text on any size screen, but doesn’t scroll
- Scrolling can be added via a decorator that:
  - Overrides draw
  - Draws a scrollbar
  - Scales and moves the viewport according to the scrolling position
  - Calls draw() on the underlying window