Principles of OO Design, or Everything I Know About Programming, I Learned from Dilbert

Alan Knight
THE KEY TO CAREER SUCCESS IS FINDING YOUR SPECIAL GIFT.

MY SPECIAL GIFT IS GETTING PAID FOR DOING NOTHING BUT BABBLING JARGON.

MAYBE I SHOULD LEAD BY EXAMPLE.

MAYBE YOU ALREADY DID.
1. Never do any work that you can get someone else to do for you

Example 1 Total of bills that have been paid this quarter for a factory

```java
total = 0;
Vector billings = aFactory.billings();
for (Bill billing : billings)
    if ((billing.status() == "paid") && (billing.date() > startDate))
        total = total + billing.amount();
```

versus

```java
total = aFactory.totalBillingsPaidSince(startDate).
```
1. Never do any work that you can get someone else to do for you

Excuse me Smithers. I need to know the total bills that have been paid so far this quarter. No, don’t trouble yourself. If you’ll just lend me the key to your filing cabinet I’ll go through the records myself. I’m not that familiar with your filing system, but how complicated can it be? I’ll try not to make too much of a mess.

Verses

SMITHERS! I need the total bills that have been paid since the beginning of the quarter. No, I’m not interested in the petty details of your filing system. I want that total, and I’ll expect it on my desk within the next half millisecond.
1. Never do any work that you can get someone else to do for you

```
somebody.clients().add( new Client());

verses

somebody.addClient( new Client());
```

Less work

somebody just returns collection

Needs

```
addClient:
removeClient:
more?
```
1. Never do any work that you can get someone else to do for you

```java
somebody.clients().add( new Client());
```

verses

```java
somebody.addClient( new Client());
```

Information Leakage

All code using class needs to know you have a collection of clients

Information hiding
Encapsulation & Responsibility

Encapsulation is about responsibility

Who does the work

Who should do the work
2. Avoid Responsibility

If you must accept a responsibility, keep it as vague as possible.

For any responsibility you accept, try to pass the real work off to somebody else.
Strategy Pattern
Favor Composition over Inheritance
Sortable List

Sorted
Reverse Sorted
Random
One size does not fit all
Issue 1 - Orthogonal Features

Order
Sorted
Reverse Sorted
Random

Threads
Synchronized
Unsynchronized

Mutability
Mutable
Non-mutable
Issue 2 - Flexibility
Change behavior at runtime

```java
OrderableList x = new OrderableList();
x.makeSorted();
x.add(foo);
x.add(bar);
x.makeRandom();
```
Configure objects behavior at runtime
Strategy Pattern

class OrderableList {
    private Object[] elements;
    private Algorithm orderer;

    public OrderableList(Algorithm x) {
        orderer = x;
    }

    public void add(Object element) {
        elements = ordered.add(elements, element);
    }
}
Structure

Context
  contextInterface()

Strategy
  algorithmInterface()

ConcreteStrategyA
  algorithmInterface()

ConcreteStrategyB
  algorithmInterface()
The algorithm is the operation

Context contains the data

How does this work?
Prime Directive
Data + Operations
How does Strategy Get the Data?

Pass needed data as parameters in strategy method

Give strategy object reference to context
Strategy extracts needed data from context
import java.awt.*;
class FlowExample extends Frame {

    public FlowExample( int width, int height ) {
        setTitle( "Flow Example" );
        setSize( width, height );
        setLayout( new FlowLayout( FlowLayout.LEFT) );

        for ( int label = 1; label < 10; label++ )
            add( new Button( String.valueOf( label ) ) );
        show();
    }

    public static void main( String args[] ) {
        new FlowExample( 175, 100 );
        new FlowExample( 175, 100 );
    }
}
Example - Smalltalk Sort blocks

| list |
list := #( 1 6 2 3 9 5 ) asSortedCollection.
Transcript
    print: list;
    cr.
list sortBlock: [:x :y | x > y].
Transcript
    print: list;
    cr;
    flush.
Costs

Clients must be aware of different Strategies

Communication overhead between Strategy and Context

Increase number of objects
Benefits

Alternative to subclassing of Context

Eliminates conditional statements

Replace in Context code like:

```java
switch ( flag ) {
    case A: doA(); break;
    case B: doB(); break;
    case C: doC(); break;
}
```

With code like:

```java
strategy.do();
```

Gives a choice of implementations
Refactoring: Replace Conditional Logic with Strategy

Conditional logic in a method controls which of several variants of a calculation are executed

so

Create a Strategy for each variant and make the method delegate the calculation to a Strategy instance
Replace Conditional Logic with Strategy

class Foo {
    public void bar() {
        switch ( flag ) {
            case A: doA(); break;
            case B: doB(); break;
            case C: doC(); break;
        }
    }
}

class Foo {
    private strategy;
    public void bar() {
        strategy.do(data);
    }
}
Decorator
Prime Directive
Data + Operations
Decorator Pattern
Adds responsibilities to individual objects

Dynamically
Transparently
import java.io.*;
import sdsu.io.*;
class ReadingFileExample
{
    public static void main( String args[] ) throws Exception
    {
        FileInputStream inputFile;
        BufferedInputStream bufferedFile;
        ASCIIInputStream cin;

        inputFile = new FileInputStream( "ReadingFileExample.java" );
        bufferedFile = new BufferedInputStream( inputFile );
        cin = new ASCIIInputStream( bufferedFile );

        inputFile = new FileInputStream( "ReadingFileExample.java" );
        bufferedFile = new BufferedInputStream( inputFile );
        cin = new ASCIIInputStream( bufferedFile );
    }
}
Decorator forwards all component operations
Favor Composition over Inheritance
Refactoring: Move Embellishment to Decorator

Client → aBinaryTree
toArray

Client → anOddValueDecorator → aBinaryTree
toArray
Benefits & Liabilities

Benefits

Simplifies a class
Distinguishes a classes core responsibilities from embellishments

Liabilities

Changes the object identity of a decorated object
Code harder to understand and debug
Combinations of decorators may not work correctly together
Pipes and Filters
Pipes & Filters

ls | grep -i b | wc -l

Context
Processing data streams

Problem
Building a system that processes or transforms a stream of data

Forces
Small processing steps are easier to reuse than large components

Non-adjacent processing steps do not share information

System changes should be possible by exchanging or recombining processing steps, even by users

Final results should be presented or stored in different ways
Solution

Divide task into multiple sequential processing steps or filter components

Output of one filter is the input of the next filter

Filters process data incrementally

Filter does not wait to get all the data before processing
Solution Continued

Data source – input to the system

Data sink – output of the system

Pipes - connect the data source, filters and data sink

Pipe implements the data flow between adjacent processes steps

Processing pipeline – sequence of filters and pipes

Pipeline can process batches of data
http://wiki.cs.uiuc.edu/cs427/Python+-+Batch+Sequential
Intercepting Filter - Problem

Preprocessing and post-processing of a client Web request and response

A Web request often must pass several tests prior to the main processing
  Has the client been authenticated?
  Does the client have a valid session?
  Is the client's IP address from a trusted network?
  Does the request path violate any constraints?
  What encoding does the client use to send the data?
  Do we support the browser type of the client?

Nested if statements lead to fragile code
Intercepting Filter - Forces

Common processing, such as checking the data-encoding scheme or logging information about each request, completes per request.

Centralization of common logic is desired.

Services should be easy to add or remove unobtrusively without affecting existing components, so that they can be used in a variety of combinations, such as

Logging and authentication

Debugging and transformation of output for a specific client

Uncompressing and converting encoding scheme of input