Object Oriented Programming

Designing reusable code
Writing efficient code

Verónica Gaspes

School of Information Science, Computer and Electrical Engineering

CERES

April 22th, 2008

Design by abstraction

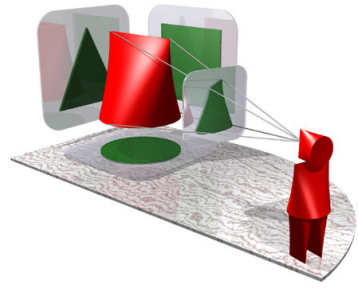
Designing generic components

- Reusable
- Extensible

Without having to modify the code!

Using:

- Abstract classes
- Interfaces
- Design patterns (and idioms)



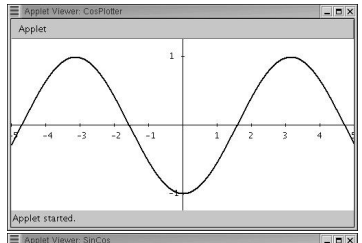
Running example

An applet for plotting functions

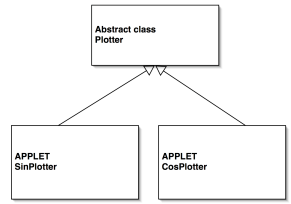
- Should be easy to adapt for different functions
- A generic applet that captures the common code

Will help us illustrate 2 techniques to factorize code:

1. the template pattern (using inheritance)
2. the strategy pattern (using delegation)



Our first generic plotter




```
public class Plotter extends JApplet {
  public void init(){
    read html parameters, size and scale
  }
  public void paint(Graphics g){
    draw the coordinate axis and the function graph
    in the interval given by the parameters!
  }
```

What function?

Typically we think of functions as being implemented by methods!

In Plotter

```
private double func(double x){
        ... ???
}
```

By making it abstract we can let other classes implement it!

Plotter becomes abstract

```
public abstract class Plotter extends JApplet {
    protected abstract double func(double x);
}
```

Plotter cannot be instantiated!

Providing functions

To plot sinus

```
public class SinPlotter extends Plotter{
    public double func(double x){
       return Math.sin(x);
    }
}
```

To plot cosinus

```
public class CosPlotter extends Plotter{
    public double func(double x){
       return Math.cos(x);
    }
}
```

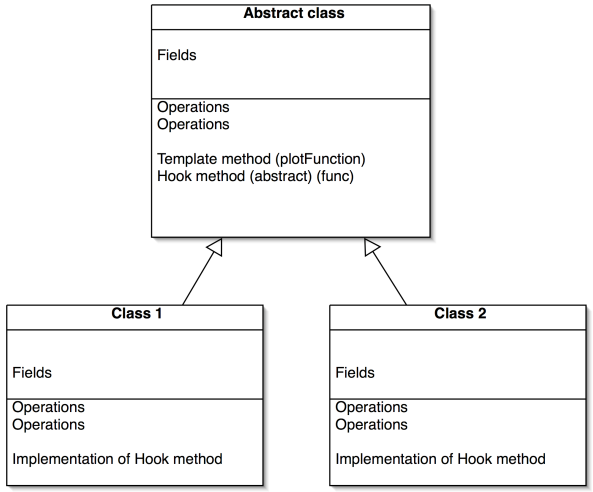

Looking inside Plotter

```
public abstract class Plotter extends JApplet {
  private  int   xorigin, yorigin, xratio, yratio, w, h;
  private  Color color = Color.black;

  protected abstract double func(double x);

  public void init(){
    // read parameters:
    w    = new Integer(getParameter("width"));
    h    = new Integer(getParameter("height"));
    xorigin = new Integer(getParameter("xorigin"));
    yorigin = new Integer(getParameter("yorigin"));
    xratio  = new Integer(getParameter("xratio"));
    yratio  = new Integer(getParameter("yratio"));
  }

...
```


Looking inside Plotter

```
public void paint(java.awt.Graphics g){
  drawCoordinates(g);
  plotFunction(g);
 }
private void plotFunction(java.awt.Graphics g){
  for(int px = 0; px < w; px ++){
    try{
      double x  = (double)(px - xorigin)/(double)xratio;
      double y  = func(x);
      int    py = yorigin - (int)(y * yratio);
      g.fillOval(px-1,py-1,3,3);
    }catch(Exception e)
  }
}
```

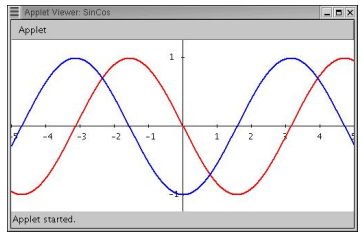
The Template pattern



How do we plot many functions?

Whats good!

With our version of plotter we just have to define new classes in order to plott new functions, we do not have to modify or recompile the plotter class!



The bad news

This doesn't apply if we want to plot several functions in the same graph!

Instead of a hook method we would like to have an array of hook methods!

Function objects??

We still want something like this:

```
public class Plotter extends JApplet {
  public void init(){
    read html parameters, size and scale
  }
  public void paint(Graphics g){
    draw the coordinate axis and the function graph
    in the interval given by the parameters!
  }
```

But we would like to have the function as

`Function f`

Or even better, possibly several functions as

`Function [] fns`

Function Objects!

```
public interface Function{
    public double apply(double x);
}
```

```
class Sin implements Function{
    public double apply(double x){
        return Math.sin(x);
    }
}
```

```
class Cos implements Function{
    public double apply(double x){
        return Math.cos(x);
    }
}
```

Plotter revisited

```
public abstract class MultiPlotter extends JApplet{
    private  Color[]       colors;
    private  Function[]    functions;
    private  int           numOfFunctions;

    private void plotFunctions(Graphics g){
      for(int i = 0; i < numOfFunctions; i++){
        g.setColor(colors[i]);
        for(int px = 0; px < dim.width; px ++){
           try{
             double x  = (double)(px - xorigin)/
                         (double)xratio;
             double y  = functions[i].apply(x);
             int    py = yorigin - (int)(y * yratio);
             g.fillOval(px-1,py-1,3,3);
           }catch(Exception e){}
```


Where do we get the functions from?

We still have not said how we provide functions to the multiplotter!

First attempt

Classes for plotting functions extend `MultiPlotter` by defining `init()` where

- Parameters are read from the html file
- The arrays of functions and colors are populated

Risk!

The programmer might forget to read the parameters that are used in the drawing methods!

Where do we get the functions from?

Second attempt

Let `init()` be a template method where the parameters are read and a hook method is called to populate the arrays!

```
public final void init(){
    // read parameters and then:
    colors         = new Color[maxFunctions];
    functions      = new Function[maxFunctions];
    numOfFunctions = 0;
    initMultiPlotter();
}

protected abstract void initMultiPlotter();
```


Populating the arrays

`MultiPlotter` also provides a way for populating the arrays that can be used in the classes that extend it

```
protected void addFunction(Function f, Color c){
    functions[numOfFunctions] = f;
    colors[numOfFunctions]    = c;
    numOfFunctions++;
}
```

Putting `MultiPlotter` to work

Sinus and Cosinus

```
public class SinCos extends MultiPlotter{
    protected void initMultiPlotter(){
        addFunction(new Sin(),java.awt.Color.red);
        addFunction(new Cos(),java.awt.Color.blue);
    }
}
```


Design Guidelines

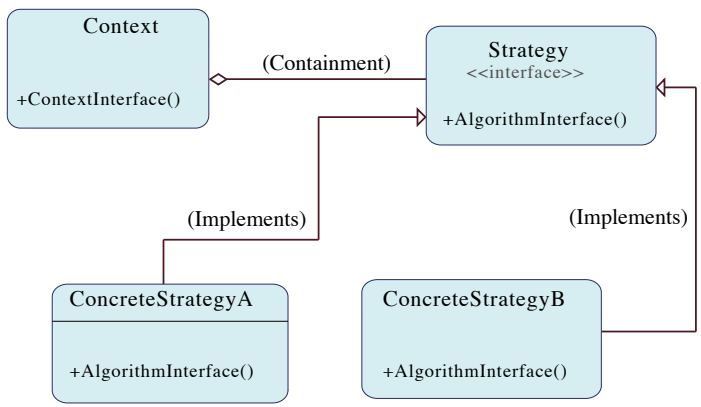
Maximize adaptability

The more adaptable (extensible) a component is, the better chances it will be reused!

Minimize risk for missuse!

Make some methods final and force the definition of a hook method instead of allowing for a redefinition! (as we did with `init` and `initMultiPlotter`!)

The strategy pattern: Delegate!



Some hints

This is just a guide to read
http://www.itu.dk/~sestoft/papers/performance.pdf