

TDA367/DIT211 – Project Course IT

Pattern workshop

Pelle Evensen

April 1, 2011

Abstract

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved¹.

In this workshop session we will see how to improve and adapt an existing (poor) design to increase maintainability and performance.

1 The stage

We just got a new job at Pelle's Venerable Software Shop. One of our clients have some terribly old hardware sitting around. One of their previous consultants have been kind enough to provide us with a Java class that let us control a venerable pen plotter² as shown in Fig. 1. The consultants were also nice enough to provide us with a class for on-screen simulation of the plotter. Both the controller class (which we do not have access to, not even as a class- or jar-file) and the simulation class (`orig.plotter.ColorPenPlotter`) implement the interface `orig.plotter.Plotter`.

To get an idea on how the output typically could look, run `orig.demo.Drawing-Program`.

2 Our problem(s).

The classes we are given in `orig.plotter.*` and `orig.screen.*` *must not be changed*. Let's say the source code was lost long ago and there may be other classes that rely on them. Let us also assume they have been thoroughly tested.

²If you've never seen one in action, take a look at http://www.hpmuseum.net/upload_htmlFile/Web7440Plot.mpg.

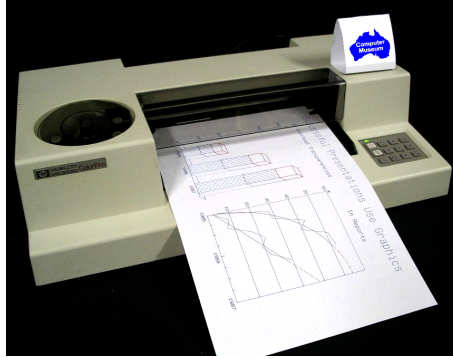


Figure 1: A HP74440A ColorPro pen-plotter.

2.1 Different simulation needs.

The client who still uses the plotter wants to be able to simulate the plotter's actions, by seeing on screen what the plotter do, given some commands. The class `orig.plotter.ColorPenPlotter` can be used for this.

At the same time, the client is sometimes more keen to see what the *result looks like* than *how* the plotter goes about doing its thing. The client has provided us with a class `orig.screen.ScreenDrawer` that can display a window where one can draw some simple graphic primitives. Alas, the public interface for `orig.plotter.ColorPenPlotter` is quite low level. Even worse, the two simulation classes have different interfaces.

2.1.1 Your adapter task

By using the *adapter pattern* we can consolidate the interfaces.

First create a new, empty, interface; `better.plotter.GenericPlotter`. Now create two *adapter classes*, both implementing `better.plotter.GenericPlotter`. These two classes should *adapt* the `orig.plotter.Plotter` interface and the `orig.screen.ScreenDrawer` class.

You will have to decide what methods should be in `GenericPlotter`.

Hint 1: Don't worry too much about the colours. The easy way to consolidate the colour handling is to ignore all colours but the ones defined in `orig.plotter.PenColor`.

Hint 2: When deciding on what methods the interface `GenericPlotter` should contain, question if we are interested in anything but drawing lines. Think about the proper level of abstraction.

2.1.2 Your wrapper/decorator task

One major problem with pen plotters is that if they do run out of ink while drawing a line segment, there is no easy way to resume drawing that particular segment.

The client demands a better interface for pen plotters that can give a good estimate on how much ink has been used as well as how long the pen-head has traveled. This lets the programmer raise an alarm before a pen has ran out of ink.

Create a new interface, `better.plotter.InstrumentedPlotter`, that extends `better.plotter.GenericPlotter`. An example of how the instrumentation could be implemented is in `demo.plotter.InstrumentedPlotterAdapter`. Note that the demo version most likely is incompatible with your `GenericPlotter`-interface. Copy and modify the instrumentation class and put it in the `better.plotter`-package. We provide the demo-class in the interest of time.

2.1.3 Proper level of abstraction

Now that you have extended the plotter to some extent, implement this method:

```
void circle(int xCenter, int yCenter, int radius, PenColor color).
```

In what class or interface should this method reside and why?

Reflect...

Why do we not let `better.plotter.InstrumentedPlotterAdapter` inherit the concrete plotter we would like to instrument? Do we lose any reliability, flexibility or both?

3 Can you tell the difference between speed and height?

The constructor for `orig.plotter.ColorPenPlotter` has five parameters. One problem is that we have no way (at compile-time) of detecting if we get the order wrong since every parameter is of type `int`. Might there even be sensible default values for some of these parameters so that we don't need to pass all of them?

3.1 First attempt at dealing with defaults.

We could provide five different constructors. Java does, after all, have constructor overloading...

This "solution" is problematic for several reasons;

1. It does nothing to prevent us from permuting the arguments when calling the constructor.
2. If we have an constructor with very many parameters,³ we may have to provide as many constructors as there are arguments (or more).

³As a somewhat pathological example, `java.awt.GridBagConstraints` has a constructor that takes 11 arguments...

3. We may want the first n values to take on defaults and just change parameter $n + 1$. We will still need to pass the first n values, even if we safely can assume that the class will have sensible defaults for them.

3.2 Second attempt at dealing with defaults.

Provide a constructor that takes no parameters and puts the object in some “sensible” state. Provide setters to access the values you would like to change.

This solution also comes with some problems;

1. There may be dependencies between parameters. Let’s say that the object needs a point that lies within a circle but takes two integers to describe it. Depending on the order of the calls to the setters, this may fail.
2. The object may be in an inconsistent state if we can not check several parameters at the same time.
3. We can not make the class immutable even if there are no other uses for the setters.

3.3 *Builder* to the rescue!

One use for the *builder pattern* is as a remedy for the *telescoping constructor anti-pattern*. The “solution” in section 3.1 above illustrates that pattern well; every constructor takes one more parameter than the previous one.

Study the builder example in Fig. 2. Use the same builder technique used in `orig.demo.GridBagConstraintsAlternative` on your adapter for the `orig.plotter.ColorPenPlotter`.

Reflect...

Suggested reading: Items 2 and 15 in [Blo08]. In this example, the builder pattern simulates *named optional parameters* as found in e.g. Ada and Python.

4 Conclusion

You have now seen a few usages of some common patterns, trying to remedy some regularly occurring potential problems.

In particular, the *decorator pattern* is something that always should be considered before one uses implementation inheritance. Aside from mostly preventing the *fragile base class problem*⁴ from occurring, the pattern will also let us *recombine* properties and functionality in a type-safe manner at run-time.

⁴Why extends is evil: <http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-toolbox.html>

```

public class GridBagConstraintsAlternative {
    private int gridx;
    private int gridy;
    private int gridwidth;
    private int gridheight;

    // Ignoring the other 7 parameters for the sake of brevity...

    private GridBagConstraintsAlternative(Builder builder) {
        gridx = builder.gridx;
        gridy = builder.gridy;
        gridwidth = builder.gridwidth;
        gridheight = builder.gridheight;
    }

    public static class Builder {
        private final int gridx; // Required parameter
        private final int gridy; // Required parameter
        // Set defaults here.
        private int gridwidth = 0;
        private int gridheight = 0;

        // And so on and so forth.

        public Builder(int gridx, int gridy) {
            this.gridx = gridx;
            this.gridy = gridy;
        }

        public Builder gridWidth(int gridWidth) {
            this.gridwidth = gridWidth;
            return this;
        }

        public Builder gridHeight(int gridHeight) {
            this.gridheight = gridHeight;
            return this;
        }

        public GridBagConstraintsAlternative build() {
            return new GridBagConstraintsAlternative(this);
        }
    }

    public static void main(String[] args) {
        // We want to build an object that sets the height but takes the default
        // value for the width.
        GridBagConstraintsAlternative gbc = new Builder(1, 2).gridHeight(2)
            .build();

        // Yay! We did not need to create bogus default values and we ran (almost) no
        // risk of getting the order of parameters wrong.
    }
}

```

Figure 2: The **GridBagConstraintsAlternative** class.

References

[Blo08] Joshua Bloch. *Effective Java*. Addison-Wesley, 2nd edition, 2008.