Retrospective of the Extensibility Problem

Alexandre Bergel
Alexandre.Bergel@cs.tcd.ie
www.cs.tcd.ie/Alexandre.Bergel

January 15, 2007

1 Introduction

Making a software evolve may be an easy task if you have few number of clients and/or very few people are involved in developing a software. However it could be a quite difficult and costly task if your developing a large software and have multiple clients. One typical problem that may result from this is maintainability.

The goal of this exercise is to give you a feeling of how such problem may arise. The extensibility problem [3,4] is a classical example of software evolution. In Section 2 you will build a small set of classes representing a graphical hierarchy for geometrical objects. In Section 3 you will extend it using different several different approaches.

2 Set of widgets as a base application

As a first task, you will build a simple class hierarchy of widgets with a factory. Figure 1 shows a class Component and 3 subclasses. Instances of widgets are produced by means of a Factory.

Your job: Implement the class hierarchy and the factory depicted by Figure 1. Define a Java package widgets that will contains the 4 classes. Just use an output on the standard output stream to implement draw().

3 Adding a colour concern

Congratulation! Your widgets hierarchy is a great success, and people are using it a lot. You're making big money and you're providing support for several major companies. Everything went fine, until a client asked for some extra features... Such as coloured widgets.

The main issue here is to not modify the original class hierarchy. This is important since many clients are using it, and they simply do not care about this colour concern. The challenge is to implement this concern without modifying the base system.

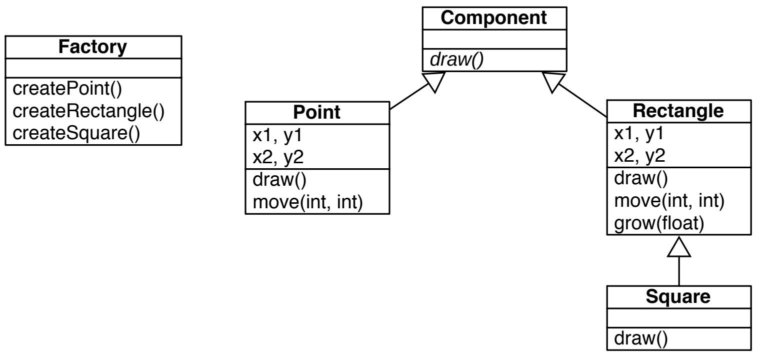

Figure 1: A simple class hierarchy.

Adding a colour feature essentially refers to adding a variable color, a method setColor(Color) and getColor(), and redefining the method draw() for each widget.

The colourless hierarchy is located in widgets package. The coloured version will be located into colouredwidgets package. To implement the coloured version, the widgets must not be modified (or else your clients will shout!).

3.1 Object-oriented approach

There are mainly two different ways to implement the colour concern: either by implementing the colour concern in a subclass of Component, or by implementing it by subclassing each widget.

Subclassing Component. The colour concern is encapsulated into a new class ColoredComponent. Each widget has to be implemented by a new class inheriting from ColoredComponent.

Your job: Implement the new hierarchy and the new factory into a coloredwidgets package.

Your job: What is the amount of code duplication?

Your job: From a typing point of view, what can you say?

Subclassing Point, Rectangle and Square. Instead of subclassing Component, subclass each of the concrete class (*e.g.,* ColoredPoint inheriting from Point, ...).

Your job: Implement the new hierarchy and the new factory into a coloredwidgets package.

Your job: What is the amount of code duplication?

Your job: From a typing point of view, what can you say?

So, which one is the best?. None of them. In fairly large application, a mix of these two techniques is used. For example, in the Swing Java library, you have JFrame which is a subclass of Frame, JWindow a subclass of Window, but JButton is a subclass of JComponent, itself being a subclass of Component.

3.2 Classboxes approach

In this subsection you will tackle the extensibility problem with classboxes. First of all, you need to install the classbox compiler on your machine. It is freely available from the classbox website. The classbox/J compiler is intended to be used through an xterm (and not as an Eclipse plugin).

Important notes: The Classbox/J compiler is an experimental product, you therefore have to be indulgent with the quality of output you might get with it, especially if what you give to it contains error. If what you provide to the compiler is not exact and error free, you might get a freeze of the compiler for instance.

Your job: Go to http://www.iam.unibe.ch/~scg/Research/Classboxes/index.html and download the classbox compiler for Linux.

Your job: Unzip the file (it will create a folder CBJ).

Your job: Some global variables need to be initialised. For this, go into this CBJ folder, and type (assuming you're using bash):

```
export PATH=$PATH:$PWD/bin
export CLASSBOXJ_PATH=$PWD
```

In order to make sure that everything is properly installed, some tests are available.

Your job: Enter the `tests` folder, and type `make`. The tests should pass.

In your working place, it is recommended to have two folders, `src` and `bin`. The first one will contain the classbox/J code, and the latter the binary (result of a compilation). The `cbj` unix command invokes the compiler. It has the same interface than the `javac` command.

A first example. As a first example, you will first redefine a class Flag with the German colour. Then you will refine this class to have the Austrian flag.

Your job: In a folder `src/flag` define the file `Flag.java` as the following:

```
package flag;

public class Flag {
  public void draw() {
  System.out.println(this.getColor());
  }
  public String getColor() {
    return "(black, red, yellow)";
  }
}
```

Your job: In a folder `src/austrianflag` define the file `Flag.java` as the following:

```
package austrianflag;

import flag.Flag;
refine Flag {
  public String getColor() {
    return "(red, white, red)";
  }
}
```

We have now two versions of the class Flag. We can create clients for each of them.

Your job: In a folder `src/app1` define the file `App.java` as the following:

```
package app1;
import flag.Flag;
public class App {
  public static void main(String[] args) {
    (new Flag()).draw();
  }
}
```

Your job: In a folder `src/app2` define the file `App.java` as the following:

```
package app2;
import austrianflag.Flag;
public class App {
  public static void main(String[] args) {
    (new Flag()).draw();
  }
}
```

Your job: Compile the files as the following: `cbj -d bin src/flag/*.java src/austrianflag/*.java src/app1/*.java src/app2/*.java`

Your job: Execute `java -cp bin app1.App` and `java -cp bin app2.App`

Here is an excerpt:

```
alexandre-bergels-computer@bergel:/tmp/F$ cbj -d bin src/flag/*.java src/austria
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

```
alexandre-bergels-computer@bergel:/tmp/F$ java -cp bin app1.App
(black, red, yellow)
alexandre-bergels-computer@bergel:/tmp/F$ java -cp bin app2.App
(red, white, red)
alexandre-bergels-computer@bergel:/tmp/F$ ls
bin     src
alexandre-bergels-computer@bergel:/tmp/F$ ls bin
app1            app2            classbox        flag
alexandre-bergels-computer@bergel:/tmp/F$
```

Extending of the hierarchy. Your `src` folder should contain a folder widgets that contains the classes Component, Point, Rectangle and Square as showed in Figure 1.

Your job: Create a package colouredwidgets that refine the original hierarchy with a colour concern.

References

[1] Aspect oriented software development. http://www.aosd.net.

[2] AspectJ home page. http://eclipse.org/aspectj/.

[3] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 94–104. ACM Press, 1998.

[4] M. Torgersen. The expression problem revisited — four new solutions using generics. In M. Odersky, editor, *Proceedings ECOOP 2004*, LNCS, Oslo, Norway, June 2004. Springer-Verlag.