Argument Accumulator

David Harvey

Intent

To simplify the writing and improve the communication of calls to methods which would otherwise require large numbers of arguments and to avoid the confusing use of null to denote optional arguments.

Also Known As

Named Arguments.

Applicability

Use this pattern when

- You are programming in a language which does not directly support default and keyword arguments
- You are writing a method which takes a large number of arguments, or
- You are creating a class whose instance constructor needs to be passed a large number of arguments, or
- You are encapsulating legacy procedural or object code which features a function, method or constructor with a large number of arguments.

The pattern is particularly helpful when the arguments are logically grouped, and when null values must otherwise be passed to indicate default or not-present arguments.

Motivation

Consider a Java GUI library, in which a window object can be created with a number of properties: position (x and y coordinates), size (width and height), background colour, perhaps a default pen, title, border (present or absent) and more. Many such systems allow these parameters to be passed when the object is constructed:

```
public class Window
{
    // . . .
    public Window(int x, int y,
                  int width, int height,
                  Color background,
                  Pen defaultPen,
                  String title,
                  boolean hasBorder)
    {
        // initialise window state
    }
}
```

There are several problems here. Firstly, using such a call usually involves recourse to documentation, to remember the order and meaning of the arguments. Secondly, and for the same reason, a reader of the code has to do the same: faced with the following the meaning of the call is not immediately apparent:

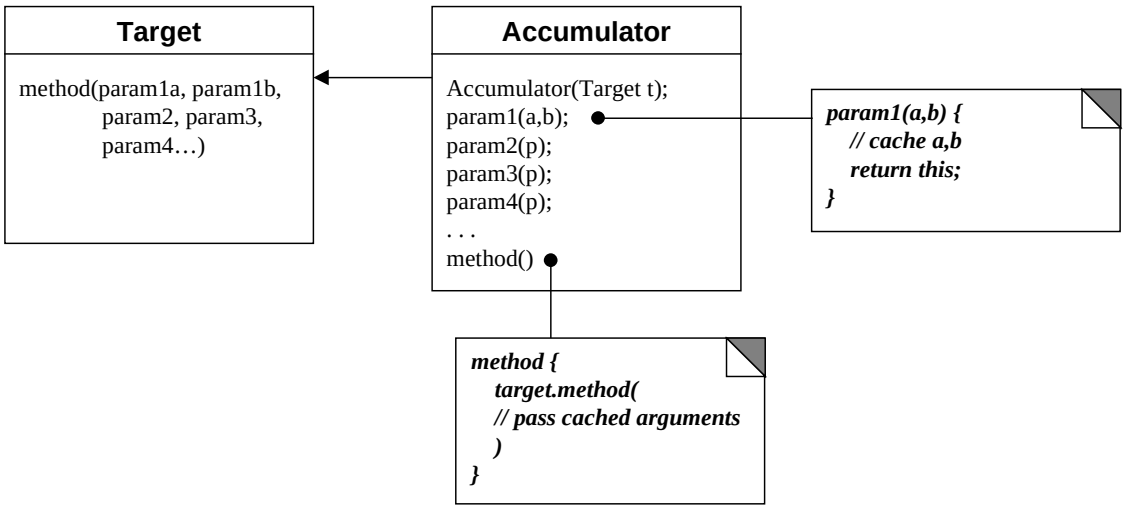
```
Window w = new Window(10,10,200,150,
                      Color.black,
                      MyPen,
                      null,
                      false);
```


Note here to the use of **null** as an argument. This is a common way of expressing a default selection, but it renders code even more obscure, as for example:

```
Window w = new Window(10,10,200,150,
                      null,null,null,false);
```

An aim of any programming style should be to increase the *readability* of the code – this is taken to an extreme in Extreme Programming, where the code is all there is to read. Without having recourse to heavy commenting, what should we do?

Structure



Participants

Target
The object whose method is ultimately called..

Accumulator
An object accumulating arguments to the target method, by means of a series of named methods. These methods logically group arguments, and may be called in any order. Because each of these methods returns **this** or the equivalent, calls may be chained. The accumulator is responsible for calling the method on the target.

Collaborations

- Client creates an instance of the accumulator
- Client identifies a target to the accumulator
- Client invokes accumulator methods to set up call
- Client invokes method on accumulator to request underlying call
- Accumulator calls underlying method on target, passing the return value of this call back to the client

Consequences

- Arguments are associated with descriptive names

These are easier to remember when coding, and the intent of the resulting code is clearer. Writers and readers do not have to remember the sequence of arguments and their roles.

- Arguments are logically grouped

This likewise improves ease of writing and reading.

- Arguments can be applied in any order

Sequence of arguments becomes unimportant

- A new class is necessary
- A method for each group of arguments must be provided on this class

Implementation

The simplest implementation involves simply coding the single new class required. Key features are the variables used to cache values to be used for the call, the accumulator methods themselves, the identification of a target, and the invocation of the underlying call.

Cached values

A member variable is declared for each argument passed to the underlying call. Appropriate default values must be set for each of these: alternatively, the method encapsulating the call can check the arguments and throw an exception if necessary initialisation has not been done through the accumulator calls, or exceptions thrown by the underling call can be propagated.

Accumulator methods

Each accumulator method is named to reflect the purpose of the arguments. Arguments are logically grouped by accumulator method: for example, instead of providing separate x() and y() methods, provide a method **position** with x and y parameters. The method caches the passed values and returns **this**.

Nominating a target

The target object can be passed as an argument to the constructor of the accumulator, in which case it is cached in the local state of the accumulator until the actual call is made. Alternatively, the object can be passed as an argument to the method responsible for effecting the call on the target. If the argument accumulator is used to construct an object, the method effecting the call itself encapsulates the object creation: there is no separate target object.

Method call

This is applied to the target object as described above. It is typically named after the method or function being wrapped, or **create** if the purpose is to encapsulate a constructor.

Sample Code

Here is a partial implementation of the Window class described above, with an argument accumulator used to collect values used for construction.

```
public class WindowCreator
{
    private int myX = 0;
    private int myY = 0;
    private int myWidth = 200;
    private int myHeight = 100;
    private Color myBg = Color.white;
    private Pen myPen = null;
    private String title = "";
    private boolean hasBorder = false();

    WindowCreator position(int x, int y)
    {
        myX = x;
        myY = y;
```



```
        return this;
    }

    WindowCreator size(int width, int height)
    {
        myWidth = width;
        myHeight = height;
        return this;
    }

    WindowCreator background(Color bg)
    {
        myBg = bg;
        return this;
    }

    WindowCreator pen(Pen p)
    {
        myPen = p;
        return this;
    }

    WindowCreator title(String t)
    {
        myTitle = t;
    }

    WindowCreator border()
    {
        hasBorder = true;
    }

    WindowCreator noBorder()
    {
        hasBorder = false;
    }

    Window create()
    {
        return new Window(myX, myY,
                          myWidth, myHeight,
                          myBg, myPen,
                          myTitle, hasBorder);
    }
}
```

And an example of its use:

```
Window w = new WindowCreator()
               .border()
               .title("Popup")
               .size(50,150)
               .create();
```

The pattern can be used to good effect to facilitate programming with the Java AWT **GridBagLayout** and **GridBagConstraints** classes. The Java library already abstracts parameterisation of a grid bag into a

separate class. This class, however, is simply a holder of data: individual fields must be set one-by-one. If the constraint object is reused, the developer (and reader) has to remember which fields are set to correct values for subsequent uses: if a new object is created for each setting then much repeated work setting up fields which are the same.

The interactions between the accumulator and target are more involved than the previous example: the mechanism is used to establish the connections between a **Container** object, a **Component** being placed in the frame, and the **GridBagConstraints** instance used to specify the position and characteristics of the component.

The implementation that follows is based on a **Factory Class** which performs three key roles. It encapsulates creation of the accumulator object itself, it maintains the association between the accumulator and the target across repeated additions of components, and holds a nominated instance of the accumulator as a default object. The factory class is named **GridBagHelper**, the argument accumulator methods are implemented on an inner class **GridBagConstraintsWrapper**, which caches arguments in a private instance of **GridBagConstraints**. Here is an example of this implementation in use:

```
// . . .

Frame f = new Frame("Test GBW");
GridBagHelper gbh = new GridBagHelper(f);

// Set default constraints
gbh.constrain().size(1,1).fixed().insets(3,3,3,3).def();

TextArea ta = new TextArea();
gbh.constrain()
    .extent(0,0,2,3)
    .free()
    .add(ta);

Button button = new Button("Foo");
gbh.constrain()
    .extent(2,0,1,3)
    .anchor("NE")
    .add(button);

Label label = new Label("text:");
gbh.constrain()
    .extent(0,3,1,1)
    .add(label);

TextField tf = new TextField();
gbh.constrain()
    .extent(1,3,2,1)
    .freew()
    .add(tf);

f.pack();
f.setVisible(true);

// . . .
```


Known Uses

- The motivation is developed from an example in Stroustrup94, which was produced to justify not including keyword arguments into C++.
- The pattern has been used successfully by the author to encapsulate programming with the Java AWT GridBag and GridBagLayout.
- The VERSANT Object Database uses this pattern in its Java API [Versant99]. The following example builds and executes a query:

```
Predicate pred = session.newAttrInt("age").gt(18);
ClassHandle ch = session.locateClass("Person");
HandleEnumeration e= ch
    .withDatabase("thatDB")
    .withSelectOptions(SELECT_DEEP)
    .withThisClassLockmode(IRLOCK)
    .withInstanceLockmode(NOLOCK)
    .select(pred);
```

Related Patterns

Default Object
Factory Method
Factory Object
No Null Arguments

References

Stroustrup94 Stroustrup, Bjarne, *The Design and Evolution of C++* (Addison-Wesley: Reading, Mass 1994)

Versant99 Versant Java API documentation. Contact details at http://www.versant.com

Contact Information

David Harvey
dihharvey@cix.co.uk
http://www.ftech.net/~honeyg/

