

# Codeblock

Exploring the core of the Command Pattern

Markus Völter, [markus.voelter@mathema.de](mailto:markus.voelter@mathema.de)

## Introduction

In Robert James' workshop "Patterns and Principles" at OT 2001 we were trying to identify the core behind several different GoF patterns. I was part of the *Command* group. The following describes my view about what the core behind the pattern really is.

## The Core

This section describes an artifact known as codeblock, which is what I consider the core of the Command pattern.

## Context

You have a program made up of several classes (objects at runtime). One object needs to pass "a piece of code" to another object. This is usually a kind of "configuration with a piece of code".

## Problem

In most programming languages, code and data are two completely different things. You can store data in variables, but you cannot store code in a variable. However, it is a frequent requirement to "pass code around" in order to use a piece of code prepared by one part of the program in another part.

Usually, you have to carry part of the environment in which the code block has been created over to the place where it is executed.

## Solution

Create a structure called a code block which can contain any kind of code, but which is always executed in the same way, independent of the code it contains. The code block has access to the environment in which it was created, and can also access the environment in which it is executed.

Depending on the programming language you use, this is supported by the language or not – and the implementations are also completely different.

## Implementations

As mentioned in the solution, the implementation of this concept varies significantly, depending on the used environment. I will give quite a lot of examples here in order to show how broadly this principle has been applied.

### Smalltalk

Smalltalk supports code blocks directly. Because Smalltalk uses an implicit notation quite often, you need to “pass code in” which should be executed in the other object. For example, Smalltalk uses internal iterators (as described below), as well as implicit conditional statements. Instead of using a normal *if* statement, you send the *ifTrue* message to a boolean expression. The block is used to define what it should do if it actually *is true*:

```
count isOdd ifTrue: [Transcript show: 'ODD']
             ifFalse: [Transcript show: 'EVEN'].
```

Note how the block accesses the environment in which it was created (*Transcript*).

For the internal iterators, the block requires one parameter to be passed in at the time of execution (in the example, *list* is a collection of character objects).

```
| count |
count := 0
list do: [ :elem | elem isDigit ifTrue: [ count:= count+1 ] ]
```

This piece of code iterates through all the elements in the list and for each one, it checks whether it is a digit. If so, it adds one to the *count*. Note how the code block accesses the environment where it was created (*count*), although the block is executed later by the list itself, for each of its elements. Note also, that the *do:* iterator requires that you have one parameter for each block you pass in.

The Clipper database programming language supports a similar feature.

### Lisp

For Lisp, code and data is the same. Thus, you can use code just as you can use any other kind of data. For example, you pass a function around, store it in variables and execute it later. The following piece of Lisp code defines a function *less*, which compares two integers:

```
(defun less(x y) (< x y) )
```

The following function takes two arguments: Two integers and a function should compare the two. You don't actually care how it compares them, it just needs to return a boolean result:

```
(defun compare(x y operator)
  (funcall operator( x y ) ) )
```

*funcall* calls the function (code block) which has been passed as a parameter passing in the two arguments.

## C

In C, the concept of a function pointer can be used to achieve the same effect. First, you define the signature to which the passed function must conform:

```
bool compare( int x, int y, bool (*operator)(int x, int y) ) {
    return operator(x,y);
}
```

This is again a function, which takes two *ints* and a function, which takes two *ints* as a parameter and returns a *bool*. Also here, you can access the context in which the function has been defined:

```
int lastComparedX;
int lastComparedY;

// ...

bool less( int x, int y ) {
    lastComparedX = x;
    lastComparedY = y;
    return x < y;
}
```

You can now pass the function as an argument to the other one, for example by setting a global “comparison strategy”:

```
boolean b = compare( 12, 24, less );
```

## Command Pattern

For object-oriented languages, where no language support for code blocks exists, the command pattern can be used to achieve the same goal. I use Java as an example language here, although C++ or Eiffel would also work.

To define “the signature” of the code block (i.e. the arguments needed when it will be executed) you define an abstract class with one operation *execute()*, which takes the necessary parameters.

```
abstract class SingularOperation {
    public abstract int execute( int x );
}
```

The next step would be to define a concrete class, which extends the abstract class:

```
class Add extends SingularOperation {
    private int addInt = 0;
    public Add( int i ) {
        addInt = i;
    }
    public int execute( int x ) {
        return x+addInt;
    }
}
```

Once again, you can pass information into the “code block” when and where it is created (the *i* argument in the constructor) and later on, an argument from the execution context is passed in (*x* in execute).

Thus, you can create another class which has a *SingularOperation* in an operation's signature and uses a concrete sub-operation:

```
class SomeClass {
    ...
    public void doForAllElements( SingularOperation op ) {
        Iterator it = ...
        while (it.hasNext() ) {
            int i = //next Element;
            int j = op.execute( i );
            // do something with j
        }
    }
}

SomeClass sc = new SomeClass();
...
sc.doForAllElements( new Add( 3 ) );
...
```

In C++ overriding the *operator()* function achieves the same, restricting the signature of the *execute()* operation (now played by the *operator()*) to no arguments.

## Known uses

This section shows briefly, where code blocks are generally used:

- ? All over Smalltalk for *ifTrue*, internal iterators, etc.
- ? In Lisp to define functionals and higher order functions
- ? In C, to create a poor man's *Strategy*
- ? In GUI applications (classical Command) for attaching “actions” to “events”, such as a pressed GUI button