Convenience Methods

Robert Hirschfeld

hirschfeld@windwardsolutions.com

28. July 1998

Abstract

By placing defaults and constants into a class interface, certain method signatures become more general but also longer with the use of more parameters. With *Convenience Methods* however, you will get a more flexible class interface as well as a more readable client code.

Name

Convenience Methods

Aliases

Default Arguments

Context

In order to design a more flexible class you try to avoid hard-coded constants or defaults in your code. So, you replace these constants or defaults with additional method parameters.

Problem

Although the new method interfaces are more general than before, they are also longer than before.

Forces

Client requests have to provide quite a few parameters. Since many of these parameters are defaults or constants, they are often the same. This leads to more complicated client code. Client code becomes worse/less readable and inflexible, too.

Many programming languages like Smalltalk and Java don't have the concept of default parameters ([GoRo89, Lewi95, ArGo96]). Although some languages like C++ provide them, there remains the restriction to be tied on the order of these parameters ([Stro91]).

In languages that support default parameters in method signatures (like C++), the change of defaults has to be done in the signature itself and is therefore also a change of the classes interface.

Solution

Provide both a general and an easy-to-use interface. Besides the more general methods with the complex signature and lots of parameters add some methods with fewer parameters. These methods with a simpler signature, *Convenience Methods*, call the more complex methods, e.g. by calling a sequence of them or by providing the missing parameters as filled-in values for constants or defaults (figure 1).

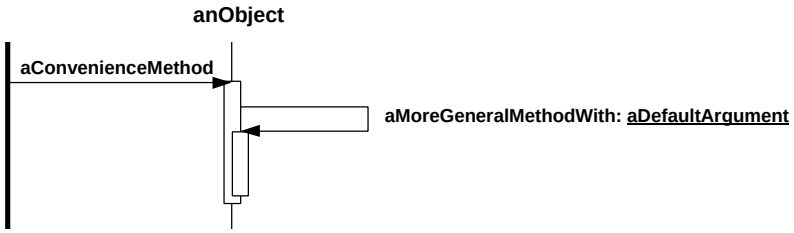

Figure 1

The client can choose between all these methods and can use that one which fits best with the set of available parameters.

Discussion

Pros:

Since the client has the ability to choose between several methods with the same semantics depending on the available set of parameters, the client code becomes more readable and therefore better to maintain.

Clients don't have to know about default parameters.

Convenience Methods make the code more flexible, too. It is easier to change the defaults because they are not distributed over the client code. Instead there is only one location for each default.

There are methods providing a more general interface (i.e. the longer methods with lots of parameters) and a more convenient one (i.e. the shorter methods with the fewer parameters).

Convenience Methods represent also a benefit for languages with default parameters (like C++), because if you have to change the defaults, you change them within the *Convenient Methods' implementations leaving the interface untouched.*

Cons:

By introducing *Convenience Methods* you introduce more levels of indirection as well. Sometimes this could result in slowing down your system.

The number of methods provided by a class supporting *Convenience Methods* increases and lets the class's interface appear more complex than one without these additional methods.

Developers of classes supporting *Convenience Methods* have to maintain consistency among the set of those methods building on each other.

If there is more than one default per method, they may be distributed across several *Convenience Methods*.

Sometimes, to use the *Convenience Methods*, you have to understand the more complex interface and the relevant defaults anyway. As such, *Convenience Methods* are only shorthand for the experienced or knowledgeable programmer.

Related Patterns

Parameters Object ([Beck95]). Parameters Objects are used to decouple instability in the parameter list from protocol instability. A change of the parameter list shouldn't always be the reason for a change or adaptation of the protocol. Providing defaults with Parameters Objects may be achieved by the initialization of them with these default values. With Parameters Objects you can't enforce a certain sequence of replacements of defaults.

Example

The following example was taken from the Smalltalk-80-based VisualWorks from ParcPlace. It is part of the well-known dependency framework provided by the Smalltalk-80 class library. Following steps performed by a DependentsCollection are left out because they are not relevant in the context these *Convenience Methods*.

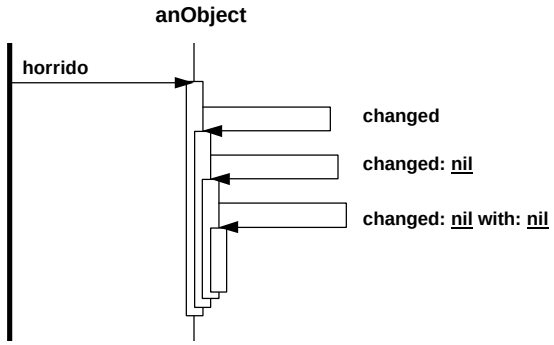

Figure 2

An object decides to propagate a general change of its state by sending a `#changed` message to itself. This *Convenience Method* starts a chain of method mappings from the specific to the more general ones (figure 2).

Because the object didn't provide an aspect denoting/describing the change, this method invokes the more general one by providing a default value for the expected aspect, the constant `nil`.

```
Object>> changed
	self changed: nil
```

The next method, `#changed:`, may be invoked by a client with a special aspect argument or by the `#changed` *Convenience Method* of the current object. Again, this method maps itself to a more general one by providing a default value for the expected additional parameter, the constant `nil`.

```
Object>>changed: anAspectSymbol
	self changed: anAspectSymbol with: nil
```

`#changed:with:` is the most general method in this chain and the last one, too. It can be invoked by the client itself or by the `#changed:` *Convenience Method* of the current object. This method informs all registered dependent objects about the change by providing all information needed by some of the dependents.

```
Object>> changed: anAspectSymbol with: aParameter
	self myDependents update: anAspectSymbol with: aParameter from: self
```

Acknowledgments

Thanks are due to Steven Abell, Antonio Rito da Silva, Jeff Eastman, Ralph Johnson and the EuroPLoP'96 writer's workshop participants for reviewing this paper and for giving helpful hints for improvement.

References

[Beck95] Beck, Kent:
Super + 1.
In: The Smalltalk Report, Vol. 4, No. 7, May 1995, pp.18-21

[GoRo89] Goldberg, Adele; Robson; David:
Smalltalk-80: The Language.
Addison-Wesley, 1989

[Lewi95] Lewis, Simon:
The Art and Science of Smalltalk.
Prentice-Hall, 1995

[Stro91] Stroustrup, Bjarne:
The C++ programming language, second edition.
Addison-Wesley, 1991

[ArGo96] Arnold, K.; Gosling, J.:
The Java Programming Language.
Addison-Wesley, 1996