Patterns in Java
One or Many?

Kevlin Henney
kevlin@curbralan.com

One or many? If you are thinking about applying the SINGLETON pattern, this is a question you need to ask yourself properly. It is also the question that many developers and development gurus should ask themselves when they think about how many pattern sources they base their design pattern knowledge on. Software developers, particularly those working with objects, increasingly claim some familiarity with patterns. But when you look more closely you discover that even in many of the expert cases it is no more than familiarity: it is rarely intimacy. Pattern expertise is often claimed based solely on knowledge acquired from the book entitled *Design Patterns* [Gamma+1995]. This is like claiming knowledge of C++ based solely on one's knowledge of Kernighan and Ritchie's classic (New Testament) *C Programming Language* [Kernighan+1988].

In *Test-Driven Development* [Beck2003], Kent Beck is quite frank about the successes of both design patterns and *Design Patterns*:

> The enormous success of design patterns is a testimonial to the commonality seen by object programmers. The success of the book *Design Patterns*, however, has stifled any diversity in expressing these patterns.

Design Patterns is without doubt seminal and, even before it came out, was for me a turning point in how I thought about design. However, roughly a decade separates these two books and the Gang of Four's success has cast a long shadow over the field.

Beyond the Gang of Four

In 1997 I ran a conference tutorial entitled "Beyond the Gang of Four", reacting to what I perceived at the time to be an increasingly narrow view of patterns by developers who were learning about patterns. Rather than treating *Design Patterns* as a beginning, many developers were viewing the book as an end in itself. Instead of treating the book as one of many in an exciting and growing area of software architecture, they were treating it like a manual for a system. And, therefore, why would you ever need more than one manual?

Sadly, six years on, the contents of my 1997 talk would still seem like news (or heresy, depending on your point of view) to many developers who take it is an article of faith that "design patterns" means *Design Patterns* and that other publications about patterns are either irrelevant, because they talk about other patterns, or fulfil a single simple purpose: explaining *Design Patterns* at a more elementary level or describing how to apply the 23 patterns in other programming languages. These books are generally not explaining and introducing design patterns: they are explaining *Design Patterns*. This secondary industry in explanatory books has in some ways made the problem worse: it has reinforced the perception that *Design Patterns* is all that matters; any other patterns are less important.

Given what we now know about how to write patterns and how to design object-oriented architectures, a critical look at *Design Patterns* suggests that it is dated. Many of the more common patterns that are used by OO developers, such as NULL OBJECT [Henney2002, Woolf1998], were not identified at the original time of writing. Instead, what we now know to be less commonly used patterns, such as INTERPRETER, are documented. Certainly, there is no doubting that INTERPRETER is a pattern, but it is not as general or as common as, for instance, PROXY.

The focus of *Design Patterns* was on general-purpose patterns rather than domain-specific ones. Our perception as to what qualifies as a general-purpose pattern has improved with time, so only with hindsight can we make this judgement. This is why the patterns community is focused on continuous discovery and documentation of patterns. Not only does our understanding of what practices constitute good design deepen, but what constitutes good design also shifts in response to changes in technology and programmer knowledge. Design does not stand still and we cannot predict the future, hence the body of pattern literature must be considered dynamic rather than static. There are few areas of knowledge in software development that are still so focused on a single book that is ten years old. Each domain has its classic references, but these domains also tend to acknowledge progress and diversity outside and beyond the classics.

We can also state confidently that patterns are not just about general-purpose design ideas. There are many patterns that are related to implementation domains, such as middleware construction [Schmidt+2000]. To actually build a specific kind of system we need design knowledge that is equally specific, hence the growth of interest in patterns that answer these less general and sometimes challenging design questions.

For example, the architecture of multi-tier Internet-based systems will certainly use patterns found in *Design Patterns* — PROXY, for instance, recurs at almost every level of the architecture — but such patterns touch on only a fraction of the common design knowledge needed by developers to build such systems. In this field and others the lack of applicability has prompted some developers to state that "Oh, design patterns are OK, but they don't apply to what I do".

This is unlikely.

Design patterns are about the application of known and proven design knowledge, whether the knowledge is from personal experience or has been shared. What those developers should be saying is "Oh, *Design Patterns* is OK, but most of the patterns that I use are not documented there". The architecture and techniques used in the majority of modern multi-tier Internet-based systems is, to be fair, unoriginal and repetitive enough that we can claim with some confidence that a solid pattern vocabulary is being used, even if developers don't know the identity of those patterns! It is the recognition of this shared and common architectural knowledge that has spurred the writing and success of books such as *Core J2EE Patterns* [Alur+2001] and *Patterns of Enterprise Application Architecture* [Fowler2003].

Global Domination

Another consequence of hindsight and critical deconstruction is recognising that some of the patterns documented in *Design Patterns* are incomplete in their descriptions, or have descriptions that no longer seem motivating. For example, we know that in most cases that COMMAND is applied, a COMMAND PROCESSOR [Buschmann+1996] is also a useful and explicit design element to apply. And, returning to where this article started, we can see that, for many reasons, SINGLETON is perhaps the weakest pattern in the catalogue. Even a cursory non-technical glance reveals that there is no motivating example and, according to

its list of consequences, it apparently has no liabilities, only benefits! From this perspective, SINGLETON sounds more like a marketing brochure than a pattern.

Kent Beck documents a number of relevant design patterns at the end of *Test-Driven Development* [Beck2003]. Each pattern takes up about a page, with one exception: SINGLETON. This is the full text for it:

> How do you provide global variables in languages without global variables? Don't. Your programs will thank you for taking the time to think about design instead.

In some senses this is a mischaracterisation of the problem that SINGLETON attempts to solve, but it is not very far from the truth. The most common perception of SINGLETON is that it is a way to provide global access to a single object. In other words, how to use global variables but without the associated social stigma. The following is the intent for the pattern listed in *Design Patterns*:

> Ensure a class only has one instance, and provide a global point of access to it.

The race to embrace the apparent convenience offered by second half of the sentence — "... global point of access..." — often eclipses the necessity of the first half and the classification of the pattern as a *creational* rather than a *structural* pattern.

The notion of providing a global point of access is seen by many as the main motivation for the pattern, whereas first and foremost SINGLETON is a factory pattern: it concerns the encapsulated creation of objects. What is it encapsulating? Instance control. In this case, to be precise, the existence of no more than a single instance — an exceedingly rare constraint in practice.

A quick examination of the majority of so-called SINGLETONs in code reveals that they are global variables and not SINGLETONs: they just happen to share some of the same solution structure, but not the same motivating problem, forces and consequences, all of which are required to correctly characterise a pattern. Most of these misapplications either do not enforce instance control or the uniqueness of the instance that they control happens to be a coincidence rather than a genuine constraint. Hence the question posed by this article: one or many? In the overwhelming majority of cases, there is no real constraint to have a lone instance: uniqueness is a property of how an application is run, not of a specific type.

Demotivating Example

Unfortunately, the examples listed in favour of SINGLETON in the original pattern are examples of application coincidence rather than genuine type constraint:

> Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

It is not hard to see that not one of these reflects a true type constraint! Every single one can be considered a counterexample, inviting the careful reader not to use SINGLETON. My system has more than one print spooler, as well as more than one printer. Depending on precisely what is meant by "a system", "a single file system" and "a single window manager", the uniqueness is either a property of the system's relationship with the file system or window manager — a cardinality of one rather than many — or the assertion is simply false. A relationship cardinality of one does not automatically become promoted (or demoted) to a SINGLETON if you happen only ever to have one instance of the object using the relationship. Hardware constraints not only have a habit of changing over time, but are precisely the thing that you want to isolate your code from in order to test it and evolve it. A test harness should be able to substitute a mock A/D converter without having to

perform stateful and magical incantations on the SINGLETON. My accountant deals with many small companies, but does not use a different accounting system for each one. These are all profoundly non-motivating examples that, when studied properly, reveal great potential for multiplicity rather than guaranteed and absolute uniqueness.

As is often the case with any hardwiring of numbers, hardwiring the property of uniqueness becomes a problem rather than a solution, and one that invites workarounds. A remarkable amount of print has been wasted on dealing with how to install separate instances of a SINGLETON in a class hierarchy, how to vary SINGLETON lifetimes, how to use different Singletons depending on whether a program is being deployed or tested, and so on. It is as if a tertiary industry has sprung up in creating, describing and programming variations, adaptations and solutions for SINGLETON, which is fine for those in that line of business but it does rather miss the point of patterns. If a pattern is inapplicable, don't apply it.

Solving the Solution

These are all examples of solving problems that arise from a solution rather than a genuine core problem. To help us out, *Design Patterns* presents the following principle of object-oriented design:

> Program to an interface, not an implementation.

We can extend this with a further principle:

> Program to an interface, not an instance.

This second principle can be considered a deeper reading and consequence of the first. Clearly, SINGLETON is in violation of this. Its application tends to restrict the evolvability, testability and comprehensibility of code. The apparent simplicity of the pattern — its class diagram involves only a single class — is deceptive. The class diagram also has two obvious omissions: there is no separate interface and a SINGLETON never has any clients. All the other creational patterns in the catalogue include these features. Perhaps the absence of users is a reflection of the way things should be, but in revising SINGLETON we can see that in addition to including proper motivating examples, listing liabilities and emphasising its creational and constraint-enforcing nature, we would revise the suggested structure to always include an EXPLICIT INTERFACE [Buschmann+2003]. The importance of global access is reduced in favour of being able to choose an appropriate implementation, leading to a more accurate revised intent:

> Ensure a class has one only instance.

Smaller, but more perfectly formed.

Conclusions

All this is not to say that you should never use SINGLETON, just that the advice on employing SINGLETON could be considered similar to Michael Jackson's oft quoted advice on optimisation [Bentley1988]:

> The First Rule of Optimization: Don't do it.
>
> The Second Rule of Optimization (For experts only): Don't do it yet.

To apply the pattern successfully requires that the designer understands both what the pattern actually says and what it needs to say. Both considerations taken together would depose SINGLETON's position as a commonplace feature in OO programs.

Such considerations also make sense for other patterns in *Design Patterns*, although with somewhat less severe consequences. Nearly ten years later, what constitutes good OO design has moved on significantly. Some of it is fashion, but much of it is more enduring architectural knowledge. The common but small subset of design knowledge represented by *Design Patterns* must now be seen against a larger and richer backdrop, where once upon a time it was the backdrop.

References

[Alur+2001] Deepak Alur, John Crupi and Dan Malks, *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall, 2001.

[Beck2003] Kent Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2003.

[Bentley1988] Jon Bentley, *More Programming Pearls*, Addison-Wesley, 1988.

[Buschmann+1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.

[Buschmann+2003] Frank Buschmann and Kevlin Henney, "Explicit Interface and Object Manager", *The Eighth European Conference on Pattern Languages of Programs*, June 2003.

[Fowler2003] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Henney2002] Kevlin Henney, "Null Object", *The Seventh European Conference on Pattern Languages of Programs*, 2002, also available from `http://www.curbralan.com`.

[Kernighan+1988] Brian W Kernighan and Dennis M Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[Schmidt+2000] Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000.

[Woolf1998] Bobby Woolf, "Null Object", *Pattern Languages of Program Design 3*, edited by Robert Martin, Dirk Riehle and Frank Buschmann, Addison-Wesley, 1998.