SINGLETON – THE ANTI-PATTERN!

A pattern captures and documents good practice that has been learned by experience. Patterns are a relative newcomer to software development, yet have actually existed in spirit within that community for as long as software has been developed. The point is this: skilled software developers have always known that when solving problems, some solutions seemed to work – with the benefit of prior experience, some solutions just *felt* right. A Pattern captures a problem and a solution that works, but that's not all, for it is very rare to find a solution that works in all circumstances. When experienced software developers apply a solution, they do so as a result of their experience, taking into account the context in which the problem occurs as well as the *tradeoffs* accepted in adopting that solution. Therefore, a Pattern captures a problem in context, together with a solution and its tradeoffs.

Patterns came to the attention of software developers in the 1990s and have accumulated a healthy body of literature. The book *Design Patterns* is the best known and the one responsible for getting the mainstream of the community interested. It is rather ironic and sad, that this very book is also responsible for one of the worst red herrings ever to mislead the software developers: SINGLETON!

According to *Design Patterns* the intent of SINGLETON is to:

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

Unfortunately this is rather vague, and this in itself causes some difficulties in discussing SINGLETON, because it fails to take into account that a SINGLETON is only meaningful if it has state. In the absence of state, ensuring there is only ever one object of a particular class is meaningless, because each instance is the same as every other one.

I have implemented SINGLETON many times over the past several years, and now, I can't think of one where the SINGLETON solution was actually a good solution to the problem it attempted to solve. It seems to me that there are some serious problems with the whole approach, because correspondence between problem and solution domain models, encapsulation, and the ability to perform initialisation, are all compromised. Further, it is now my belief that the *Design Patterns* examples of where deployment of SINGLETON is claimed to be a good approach, fail to stand up under scrutiny (but more about that shortly).

I have remarked that patterns capture good ways of solving problems. However, for every good way there are many bad ones and some of these can be found deployed several times in practice. One reason for the repeated deployment of a bad solution is that it appears to solve the problem, and usually this is for one (or, for that matter, both) of two reasons:

- The problem that needs solving has not been correctly identified, and this results in the deployed solution being a solution to the wrong problem
- The solution has been deployed because it really does solve the problem, but subject to a set of tradeoffs ranging in quality from less than optimal to downright unsuitable

Such a recurring solution – i.e. one that leads to a worse rather than better design context – is known as an *anti-pattern*. It is my belief that SINGLETON is not a pattern but an anti-pattern, and that its use causes *design damage*! In this article I will attempt to state my case. I will start by listing several reasons why I think SINGLETON is a bad idea, and finish by making some recommendations for alternative approaches.

Problems

According to *Design Patterns*, Singleton is a *design* pattern – this means it is either language independent, or at least applicable to several languages. I will detail what I think the critical problems with SINGLETON are – i.e. the reasons why I choose to use the strong term *design damage* – with this factor in mind.

Design Models

I think the best way to proceed here, is to start by going back to the basics of interface design, and in particular the questions of:

- How much knowledge should an object be able to assume of the outside world in which it will be used?
- How much responsibility should be captured within a single interface?

The answer to the first of these is simply this: an object's knowledge of the outside world should extend to what it is told via its interface. The purpose of an object is to provide certain functionality to its clients, and to this end an object should provide the minimum useful interface that makes this functionality accessible. This underpins modularity in a design. How an object is used in the outside world *beyond* its interface is something that it should – as a matter of design principle – not assume any knowledge of. Therefore it follows that an interface can't make any assumptions about how many objects that support that interface are needed, because that issue is resolvable only in the outside world.

The question of how much responsibility an interface should be charged with is somewhat less concrete. An interface should be cohesive to the point that it embodies one role in a design, but where the boundaries of a role definition lie is by no means hard and fast. Consider an interface supporting a simple FACTORY METHOD: we can see that beyond whatever functionality the interface provides, its role is extended to also serve up other objects with related roles. However, in the case of SINGLETON, the interface must not only serve up the single object, but must also – in addition to whatever design role it plays – promise to manage that object. Its role therefore extends to three responsibilities.

Object design affords the capability to preserve correspondence between the problem domain model, through the stages of modelling the solution domain, and down to the implementation code. In his *Design* presentation Kevlin Henney uses the term *modelarity* as meaning "a measure of the correspondence between the components of the problem being modelled and those in its solution". This *modelarity* factor alone plays a large part in accounting for the effectiveness of objects in software design. A key feature of a well designed system is the harmony between modelarity and modularity.

The whole premise on which SINGLETON is based is that there can only ever be one object of a certain class. When a design is viewed from the perspective of preserving modelarity, it suddenly becomes apparent that this premise is far from sound! In arguing this particular case, I will go back to *Design Patterns*, and examine what *it* claims are good uses of SINGLETON. In the section titled "Motivation" *Design Patterns* makes the following statements:

- Although there can be many printers in a system, there should only be one print spooler
- A digital filter will only have one A/D converter
- An accounting system will be dedicated to serving one company

The above three assertions all have something in common: they describe cases of a client (the system that uses the print spooler, the digital filter and the company) needing and using the services of only one instance of a supplier (the spooler, A/D converter and accounting system, respectively). Now *Design Patterns* asserts that the spooler, A/D converter and accounting system are therefore candidates for being SINGLETONs. This however is not the case and would compromise modelarity, because there is no inherent reason why these three types of service supplier can only have one instance – the *real* case is that only one instance is *needed* by the client, and the real problem is that of how the client should manage the one instance it needs. Forcing the supplier to only ever have one instance deprives the model of its opportunity to express the cardinality. The model suffers because the *wrong problem* has been solved!

There is another way in which the use of SINGLETON compromises the harmony between modelarity and modularity. Here the problem is more subtle because SINGLETON appears to underpin modularity by putting an interface and the management of its instance in one place, but this is actually an illusion. The problem domain is the source of and motivation for the model, but the management of instances is a facet of the design of a software system – it does not happen in the problem domain. Therefore, instance management is a concern in its own right, that should be separated from others. It follows that, while it could be argued putting an interface and the management of its instance in one place constitutes modularity, the argument that this is the *wrong* modularity is far more compelling!

Encapsulation

Encapsulation is fundamental to object oriented design. It is the principle by which concerns are compartmentalised, and boundaries are drawn around them. Specifically, encapsulation manifests itself in software design in the form of implementation detail being kept cordoned off and used only via a public interface. It is this principle – the encapsulated implementation being accessed via a well defined public interface – that underpins many of the benefits that good design brings with it: clear communication of intended usage, ease of testability etc.

Global variables have been known to be the enemy of encapsulation for some time. Singletons have but one instance, and it penetrates the scopes in which it is used via a route other than the public interface, making it the operational equivalent of a global variable. Therefore, SINGLETONs have many of the same drawbacks as global variables, and it is unfortunate that their appearance in *Design Patterns* has lead to so many software developers failing to notice this. In both cases – i.e. Singletons and global variables – it becomes difficult to specify the pre and post conditions for the client object's interface, because the workings of its implementation can be meddled with from outside, but without going through the interface. A consequence of the difficulties in specifying pre and post conditions is that unit tests become harder to specify.

Initialisation

Usually, at the start of a program you won't have the information needed for a SINGLETON's initialisation. Initialisation on first use is no good, because you won't know which path will be taken through the program until it is actually run. If there is only one instance of a SINGLETON, then it must be initialised only once, on or just before the (unique) object is first used. However, the path through the code will not be known until run time, and so there is no way to know the point at which the SINGLETON instance must be initialised.

One attempt to get around this I have seen (I'm deliberately not using the word "solution") is to attempt to initialise on every possible control flow on which a reference to the SINGLETON

is obtained, with an exception being raised if it is used prior to initialisation. Besides being plain ugly, this approach introduces a maintenance headache; specifically this means:

- If a new control flow is introduced into the program and a reference to the SINGLETON object is acquired on it, then it is also necessary to ensure the initialisation is executed on the new control flow.
- When the test suite is updated to take account of the new control flow, an additional test – i.e. a test that fails if acquiring a reference to the Singleton object raises an exception – will be needed to ensure initialisation has taken place

All this is not to say that initialising the SINGLETON's instance is impossible, but the number of necessary workarounds and overheads can easily be seen mounting up.

Recommendations

Having given reasons why the use of SINGLETON causes damage to software design, what recommendations can be made for alternative approaches? It seems logical to look at the drawbacks described above, and suggest approaches that do not suffer from the same drawbacks. I'll start by addressing the latter two drawbacks – i.e. initialisation difficulties and breach of encapsulation – and then assess the situation. Consider the following two approaches, when a `Client` object uses the services of a `Supplier` object – `Supplier` being the object that would have been a SINGLETON, had such a design approach been used.

Approach	When it makes sense
Pass the `Supplier` object directly into the `Client`'s methods that make use of it, by passing the `Supplier` object directly through the interfaces of those methods.	When the `Client` object is not the sole user of the `Supplier` object.
Pass the information (i.e. the arguments) needed to create the `Supplier` object to `Client` through its interface, so that `Client` can create the `Supplier` object within its implementation.	When the `Client` object is the sole user of the `Supplier` object.

Both these approaches are examples of the pattern PARAMETERISE FROM ABOVE (see *PfA*). Actually, there's something familiar about these two approaches, and so there should be, because they're just describing normal design practices!

It is obvious that using either of the above approaches, there will be no problems initialising the `Supplier` object. In both cases `Supplier` can be initialised when it is created, be it in `Client` itself (latter approach) or in `Client`'s client.

Encapsulation is also significantly strengthened, and a good way to demonstrate this is to consider what happens when `Client`'s interface is unit tested. When the design approach used makes `Supplier` a SINGLETON the behaviour of `Supplier` is unpredictable because it is outside the control of both `Client` and *its* client. In this scenario the behaviour of the component under test – and hence the outcome of the test – is affected by something invisible and uncontrollable. Replace this scenario with one where either of the above two approaches is used and this changes as follows:

- In the former case `Supplier` can be replaced with a test implementation exhibiting behaviour designed to test `Client`

– In the latter case `Supplier` is an implementation detail of `Client`, so the lifetime of `Supplier` is encapsulated completely within `Client` and therefore there is no element of randomness about it

Now that initialisation and encapsulation are taken care of, what of the issues related to models and class design? Well, interface design seems to be in good shape: what could be more natural than passing an object – either a supplier object, or the information needed to create one – through an object's interface? Unfortunately, the approaches recommended here will not automatically avoid compromising any models. What *is* for certain is that no models are automatically compromised either, which would be the case were an approach involving SINGLETON to be used (for reasons described earlier in this article). It was SINGLETON's mixing of concerns in its interface – its role in the model and the object management concern – that was problematic, but provided participating interfaces are designed with attention being paid to cohesion, this does not happen when a PARAMETERISE FROM ABOVE approach is adopted.

Summary

For a problem and its solution to be a pattern, the solution must be a good one *in practice*. SINGLETON is based on the premise that a class must only ever have one instance, and must itself enforce this singularity – but the premise is false because the client of the class, not the class itself, is in a position to know how many instances are needed. Further, breaching encapsulation and causing initialisation difficulties cannot be good for any set of design tradeoffs. Given the design damage that SINGLETON inflicts, it must be considered an *anti-pattern*.

Mark Radford

(`mark@twonine.co.uk`)

References

Design: Kevlin Henney, *Design: Concepts and Practices*, Keynote presentation at JaCC, Oxford, 18th September 1999 (available from www.curbralan.com).

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

PfA: PARAMETERISE FROM ABOVE is a term in use, but there is currently no formal write-up.

Acknowledgements

Thanks to Kevlin Henney and Alan Griffiths for their helpful comments.