What are drawbacks or disadvantages of singleton pattern? [closed]

Asked 16 years, 3 months ago Modified 1 year, 7 months ago

Viewed 662k times

2181

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 9 years ago.

The community reviewed whether to reopen this question last year and left it closed:

> Original close reason(s) were not resolved

Improve this question

The singleton pattern is a fully paid up member of the GoF's patterns book, but it lately seems rather orphaned by the developer world. I still use quite a lot of singletons, especially for factory classes, and while you have to be a bit careful about multithreading issues (like any class actually), I fail to see why they are so awful.

Stack Overflow especially seems to assume that everyone agrees that Singletons are evil. Why?

Please support your answers with "*facts, references, or specific expertise*"

design-patterns singleton

Share

Improve this question

Follow

edited Jun 20, 2021 at 7:56

community wiki
10 revs, 8 users 29%
m87

86 There's a lot of 'cons' in the answers, but I'd also like to see some good examples of when the pattern is good, to contrast with the bad... – DGM Oct 15, 2008 at 6:03

1 See also, on programmers: **The Singleton Pattern** – hippietrail Mar 19, 2012 at 19:59

2 @jalf I disagree. Just because it is a singleton does not mean you have to access it through `MySingleton.sharedInstance()` everywhere it is used. Pass it as an argument e.g. See my post: assoc.tumblr.com/post/51302471844/the-misunderstood-singleton – Erik Engheim May 25, 2013 at 14:48

59 The worst part of this whole topic is that the people who hate singletons rarely give concrete suggestions for what to use instead. The links to journal articles and self-published blogs all through this SO article, for example, go on and on about

why *not* to use singletons (and they're all excellent reasons), but they're extremely slim on replacements. Lots of handwaving, though. Those of us trying to teach new programmers why not to use singletons don't have many good third-party counterexamples to point to, only contrived examples. It's wearying. – Ti Strga Sep 8, 2015 at 18:32

3 I think it's a bad idea to have an overarching opinion on a design pattern. ALL design patterns have advantages and disadvantages that often depend on the use case. – Andy Dec 11, 2015 at 17:08

36 Answers

Sorted by: Highest score (default)

1 2 Next

1438

Paraphrased from Brian Button:

1. They are generally used as a global instance, why is that so bad? Because you hide the dependencies of your application in your code, instead of exposing them through the interfaces. Making something global to avoid passing it around is a code smell.
2. They violate the single responsibility principle: by virtue of the fact that they control their own creation and lifecycle.
3. They inherently cause code to be tightly coupled. This makes faking them out under test rather difficult in many cases.
4. They carry state around for the lifetime of the application. Another hit to testing since you can end

up with a situation where tests need to be ordered which is a big no no for unit tests. Why? Because each unit test should be independent from the other.

Share Improve this answer Follow

edited Aug 9, 2016 at 7:30

community wiki
4 revs, 4 users 65%
Jim Burger

363 I disagree with you. Since comments are allowed only 600 chars, I have written a Blog Post to comment on this, please see the link below. jorudolph.wordpress.com/2009/11/22/singleton-considerations – Johannes Rudolph Nov 22, 2009 at 14:35

62 On point 1 and 4, I think singletons are useful, and in fact almost perfect for caching data (especially from a DB). The increase in performance far outway the complexities involved in modeling unit tests. – Dai Bok Dec 2, 2009 at 15:46

60 @Dai Bok: "caching data (especially from a DB)" use the proxy pattern to do this... – oz10 Jan 13, 2010 at 23:08

67 Wow, great responses. I probably used overly agressive wording here, so please keep in mind I was answering a negatively geared question. My answer was meant to be a short list of the 'anti patterns' that occur from bad Singleton usage. Full disclosure; I too use Singletons from time to time. There are more neutrally posed questions on SO that would make excellent forums for when Singletons can be considered a good idea. For instance, stackoverflow.com/questions/228164/… – Jim Burger Apr 16, 2010 at 6:23

22 They aren't so great for caching in an multithreaded environment. You can easily defeat a cache with multiple threads fighting over a limited resource. [maintained by a singleton] – monksy Mar 28, 2011 at 14:56

496

Singletons solve one (and only one) problem.

Resource Contention.

If you have some resource that

(**1**) can only have a single instance, and

(**2**) you need to manage that single instance,

you need a **singleton**.

There aren't many examples. A log file is the big one. You don't want to just abandon a single log file. You want to flush, sync and close it properly. This is an example of a single shared resource that has to be managed.

It's rare that you need a singleton. The reason they're bad is that they feel like a global and they're a fully paid up member of the GoF *Design Patterns* book.

When you think you need a global, you're probably making a terrible design mistake.

Share Improve this answer Follow edited Jan 15, 2015 at 17:55

community wiki
3 revs, 3 users 57%
S.Lott

54 Hardware is also an example right? Embedded systems have lots of hardware that might use singletons - or maybe one big one? <grin> – Jeff Apr 28, 2009 at 0:29

192 Totally agree. There's a lot of determined "this practice is bad" talk floating around without any recognition that the practice may have its place. Often, the practice is only "bad" because it's frequently misused. There's nothing inherently wrong with the Singleton pattern as long as it's applied appropriately. – Damovisa Apr 28, 2009 at 0:34

55 Real, by-principle singletons are very rare (and a printer queue certainly isn't one, and neither is a log file - see log4j). More often than not, hardware is a singleton by coincidence rather than by principle. All hardware connected to a PC is at best a coincidental singleton (think multiple monitors, mice, printers, sound cards). Even a 500 mio $ particle detector is a singleton by coincidence and budget constraints - which don't apply in software, thus: no singleton. In telecommunications, neither phone number nor the physical phone are singletons (think ISDN, call center). – digitalarbeiter Apr 7, 2010 at 14:04

26 Hardware may have only one instance of a variety of resources, but hardware isn't software. There's no reason that a single serial port on a development board should be modeled as a Singleton. Indeed, modeling it so will only make it harder to port to the new board that has two ports! – dash-tom-bang Apr 7, 2010 at 19:02

21 So you'd write two identical classes, duplicating a lot of code, just so you can have two singletons representing two ports? How about just using two global SerialPort objects? How about not modelling the hardware as classes at all? And why would you use singletons, which also imply global

access? Do you want *every* function in your code to be able to access the serial port? – Stack Overflow is garbage Jun 25, 2010 at 2:31

388

Some coding snobs look down on them as just a glorified global. In the same way that many people hate the *goto* statement there are others that hate the idea of ever using a *global*. I have seen several developers go to extraordinary lengths to avoid a *global* because they considered using one as an admission of failure. Strange but true.

In practice the *Singleton* pattern is just a programming technique that is a useful part of your toolkit of concepts. From time to time you might find it is the ideal solution and so use it. But using it just so you can boast about using a *design pattern* is just as stupid as refusing to ever use it because it is just a *global*.

Share Improve this answer Follow

edited Sep 15, 2009 at 0:22

community wiki
3 revs, 2 users 83%
Phil Wright

17 The failure I see in Singleton is that folks use them instead of globals because they're somehow "better." The problem is (as I see it), that the things that Singleton brings to the table in these cases are irrelevant. (Construct-on-first-use is trivial to implement in a non-singleton, for example, even if it's not

actually using the constructor to do it.) – dash-tom-bang Apr 7, 2010 at 19:10

24 The reason "we" look down upon them is that "we" very frequently see them used wrong, and way too often. "We" do know that they have their place of cause. – Bjarke Freund-Hansen Feb 10, 2012 at 10:59

5 @Phil, You stated "from time to time you might find it is the ideal solution and so use it". Ok, so exactly in *which* case would we find a singleton useful? – Pacerier Jun 25, 2014 at 1:37

27 @Pacerier, whenever all of following conditions hold: (1) you only need one of something, (2) you need to pass that one thing as an argument in a large number of method calls, (3) you are willing to accept a chance of having to refactor someday in exchange for an immediate reduction in the size and complexity of your code by not passing the darn thing around everywhere. – antinome Sep 17, 2014 at 18:37

24 I don't feel that this answers anything, it just says that 'sometimes it might fit, other times it may not'. OK, but why, and when? What makes this answer more than an argument to moderation? – Guildenstern Oct 24, 2014 at 18:59

240

Misko Hevery, from Google, has some interesting articles on exactly this topic...

Singletons are Pathological Liars has a unit testing example that illustrates how singletons can make it difficult to figure out dependency chains and start or test an application. It is a fairly extreme example of abuse, but the point that he makes is still valid:

> Singletons are nothing more than global state. Global state makes it so your objects can secretly get hold of things which are not declared in their APIs, and, as a result, Singletons make your APIs into pathological liars.

Where have all the Singletons Gone makes the point that dependency injection has made it easy to get instances to constructors that require them, which alleviates the underlying need behind the bad, global Singletons decried in the first article.

Share Improve this answer Follow

edited Jan 19, 2021 at 19:03

community wiki
5 revs, 3 users 74%
jason

---

8 Misko's articles on the subject are the best thing going around on it at the moment. – Chris Mayer Sep 26, 2008 at 6:34

27 The first link doesn't actually address a problem with singletons, but rather assuming a static dependency inside the class. It's possible to fix the given example by passing in parameters, yet still use singletons. – DGM Oct 15, 2008 at 13:57

19 @DGM: Exactly - in fact, there is a huge logical disconnect between the 'rationale' part of the article, and the 'Singletons are the cause' part. – Harper Shelby Dec 29, 2008 at 20:56

1 Misko's CreditCard article is an *extreme* example of misuse of this pattern. – Alex Apr 25, 2012 at 13:17

2 Reading the second article Singletons are actually part of the object model described. However, instead of being accessible globally these are private to the Factory objects. – Marc P Oct 30, 2012 at 16:41

147

I think the confusion is caused by the fact that people don't know the real application of the Singleton pattern. I can't stress this enough. Singleton is **not** a pattern to wrap globals. Singleton pattern should only be used to guarantee that **one and only one instance of a given class** exists during run time.

People think Singleton is evil because they are using it for globals. It is because of this confusion that Singleton is looked down upon. Please, don't confuse Singletons and globals. If used for the purpose it was intended for, you will gain extreme benefits from the Singleton pattern.

Share Improve this answer Follow

edited Sep 15, 2009 at 0:28

community wiki
2 revs, 2 users 67%
Cem Catikkas

25 What is that one instance, available throughout the app? Oh. *Global*. "Singleton is **not** a pattern to wrap globals" is either naive or misleading, as *by definition*, the pattern wraps a

class around a global instance. – cHao Feb 19, 2013 at 7:10

29 Of course you're free to create one instance of the class when your application starts, and inject that instance, through an interface, into anything that uses it. The implementation shouldn't care that there can be only one. – Scott Whitlock Mar 31, 2014 at 19:33

5 @Dainius: There's really not, in the end. Sure, you don't get to arbitrarily replace the instance with another. (Except for those facepalm-inducing moments when *you do*. I've actually seen `setInstance` methods before.) That hardly matters, though -- that weenie that "needed" a singleton also didn't know a freaking thing about encapsulation or what's wrong with mutable global state, so he helpfully(?) provided setters for every. single. field. (And yes, this happens. A *lot*. Nearly every singleton i've ever seen in the wild was mutable by design, and often embarassingly so.) – cHao Jan 8, 2015 at 5:42

5 @Dainius: To a large degree, we already have. "Prefer composition over inheritance" has been a thing for quite a while now. When inheritance *is* demonstrably the best solution, though, you are of course free to use it. Same thing with singletons, globals, threading, `goto`, etc. They might *work* in many cases, but frankly, "works" isn't enough -- if you want to go against conventional wisdom, you had better be able to demonstrate how your approach is *better* than the conventional solutions. And i've yet to see such a case for the Singleton pattern. – cHao Jan 10, 2015 at 23:20

4 Lest we talk past each other, i'm not just talking about a globally available instance. There are plenty of cases for that. What i'm talking about (particularly when i say "capital-S Singleton") is the GoF's Singleton pattern, which embeds that single global instance in the class itself, exposes it via a `getInstance` or similarly named method, and prevents the existence of a second instance. Frankly, at that point, you

might as well not even have the *one* instance. – cHao Jan 12, 2015 at 12:30

76

One rather bad thing about singletons is that you can't extend them very easily. You basically have to build in some kind of decorator pattern or some such thing if you want to change their behavior. Also, if one day you want to have multiple ways of doing that one thing, it can be rather painful to change, depending on how you lay out your code.

One thing to note, if you DO use singletons, try to pass them in to whoever needs them rather than have them access it directly... Otherwise if you ever choose to have multiple ways of doing the thing that singleton does, it will be rather difficult to change as each class embeds a dependency if it accesses the singleton directly.

So basically:

```
public MyConstructor(Singleton singleton) {
    this.singleton = singleton;
}
```

rather than:

```
public MyConstructor() {
    this.singleton = Singleton.getInstance();
}
```

I believe this sort of pattern is called dependency injection and is generally considered a good thing.

Like any pattern though... Think about it and consider if its use in the given situation is inappropriate or not... Rules are made to be broken usually, and patterns should not be applied willy nilly without thought.

Share Improve this answer Follow

edited Mar 10, 2014 at 20:22

community wiki
3 revs, 3 users 73%
Mike Stone

19 Heh, if you do this everywhere, then you would have to pass around a reference to the singelton everywhere also, and thus you don't have a singleton any more. :) (Which is usually a good thing IMO.) – Bjarke Freund-Hansen Feb 10, 2012 at 11:02

4 @BjarkeFreund-Hansen - Nonsense, what are you talking about? A singleton is simply an instance of a class that is instatiated once. Referencing such an Singleton doesn't copy the actual object, it just references it - you still got the same object (read: Singleton). – M. Mimpen Dec 11, 2013 at 10:35

2 @M.Mimpen: No, a capital-S Singleton (which is what is being discussed here) is an instance of a class that (a) *guarantees* that only one instance will ever exist, and (b) is accessed via the class's own built-in global point of access. If you have declared that nothing should be calling `getInstance()`, then (b) is not quite true anymore. – cHao Feb 17, 2014 at 14:50

3 @cHao I don't follow you or you don't understand to whom I comment - which is to Bjarke Freund-Hansen. Bjarke states that having several references of a singleton results in have several singletons. That is certainly not true since there are no deep-copies. – M. Mimpen Feb 17, 2014 at 18:15

7 @M.Mimpen: I take his comment to refer more to the semantic effect. Once you outlaw calls to `getInstance()`, you've effectively tossed out the one useful difference between the Singleton pattern and an ordinary reference. As far as the rest of the code is concerned, singleness is no longer a property. Only the caller of `getInstance()` ever needs to know or even care how many instances there are. With only one caller, it costs more in effort and flexibility for the class to reliably enforce singleness than it does to have the caller simply store a reference and reuse it. – cHao Feb 17, 2014 at 18:52

71

The singleton pattern is not a problem in itself. The problem is that the pattern is often used by people developing software with object-oriented tools without having a solid grasp of OO concepts. When singletons are introduced in this context they tend to grow into unmanageable classes that contain helper methods for every little use.

Singletons are also a problem from a testing perspective. They tend to make isolated unit-tests difficult to write. Inversion of control (IoC) and dependency injection are patterns meant to overcome this problem in an object-oriented manner that lends itself to unit testing.

In a garbage collected environment singletons can quickly become an issue with regard to memory management.

There is also the multi-threaded scenario where singletons can become a bottleneck as well as a synchronization issue.

Share Improve this answer Follow

edited Mar 10, 2014 at 20:31

community wiki
5 revs, 3 users 52%
Kimoz

7 i know it is many year old thread. hi @Kimoz u said:- singletons can quickly become an issue with regard to memory management. would like to explain in more detail what kind of problem may arrive regarding singleton & garbage collection. – Thomas Apr 4, 2014 at 14:09

@Kimoz, The question is asking *"Why is the singleton pattern not a problem in itself?"* and you have merely repeated the point but provided not even a single valid use case of the singleton pattern. – Pacerier Jun 25, 2014 at 1:24

@Thomas, because a singleton by definition exists in only one instance. So, it's often complex to assign the only reference to null. It can be done, but it means that you fully control the point after which the singleton is not used in your app. And this is rare, and usually quite the opposite of what singleton enthusiast are looking for : a simple way to make a single instance *always* accessible. On some DI framworks like Guice or Dagger, it is not possible to get rid of a singleton and it stays forever in memory. (Though container provided singletons are far better than home made ones). – Snicolas May 28, 2015 at 4:49

56

A singleton gets implemented using a static method. Static methods are avoided by people who do unit testing because they cannot be mocked or stubbed. Most people on this site are big proponents of unit testing. The generally most accepted convention to avoid them is using the inversion of control pattern.

Share Improve this answer Follow

edited Mar 10, 2014 at 20:31

community wiki
2 revs, 2 users 67%
Peter Mortensen

11 That does sound more like a problem with the unit testing that can test objects (units), functions (units), whole libraries (units), but fail with anything static in class (also units). – v010dya Feb 3, 2014 at 4:35

3 aren't you suppose to moc all external references anyway? If you are, then what's a problem to moc singleton, if not, are you really doing unit testing? – Dainius Oct 24, 2014 at 11:28

@Dainius: Mocking *instances* is a lot less trouble than mocking classes. You could conceivably extract the class under test from the rest of the application and test with a fake `Singleton` class. However, that immensely complicates the testing process. For one, now you need to be able to unload classes at will (not really an option in most languages), or start a new VM for each test (read: tests may take thousands of times as long). For two, though, the dependency on `Singleton` is an implementation detail that is now leaking all over your tests. – cHao Jan 11, 2015 at 3:07

Powermock can mock static stuff. – Snicolas May 28, 2015 at 4:50

does mocking an object mean creating real object? If mocking does not create a real object then why does it matter whether the class is singleton or method is static? – Arun Raaj Aug 13, 2018 at 11:12

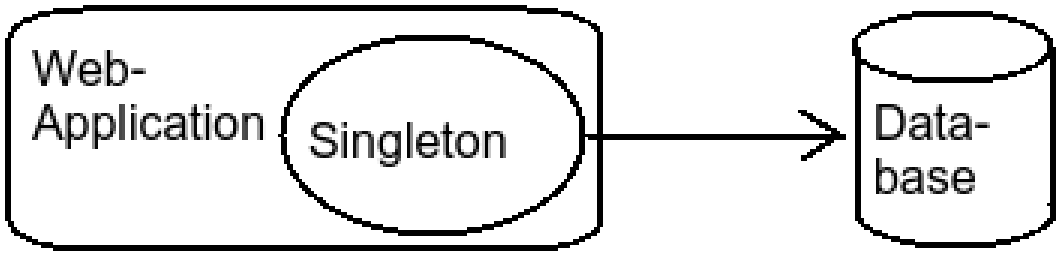
47

Singletons are also bad when it comes to **clustering**. Because then, you do not have "exactly one singleton" in your application anymore.

Consider the following situation: As a developer, you have to create a web application which accesses a database. To ensure that concurrent database calls do not conflict each other, you create a thread-save `SingletonDao`:

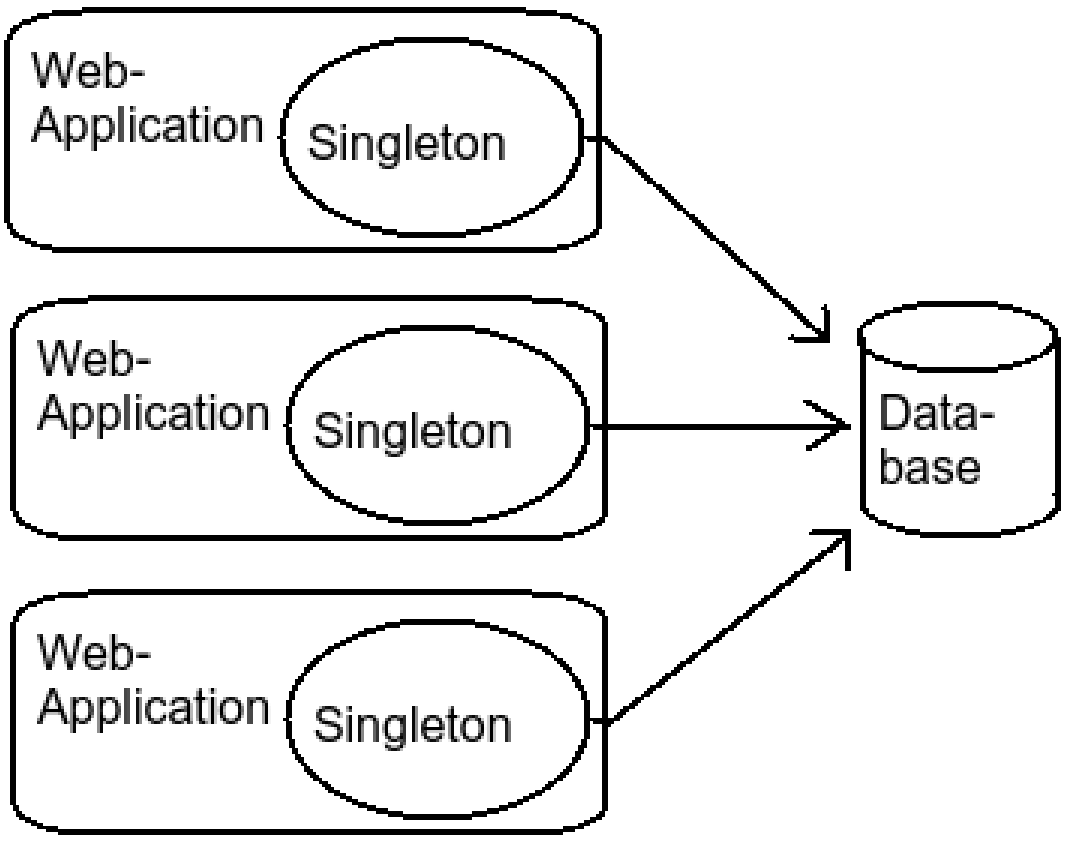
```
public class SingletonDao {
    // songleton's static variable and
getInstance() method etc. omitted
    public void writeXYZ(...){
        synchronized(...){
            // some database writing operations...
        }
    }
}
```

So you are sure that only one singleton in your application exists and all database go through this one and only `SingletonDao`. Your production environment now looks like this:



Everything is fine so far.

Now, consider you want to set up multiple instances of your web application in a cluster. Now, you suddenly have something like this:



That sounds weird, but **now you have many singletons in your application**. And that is exactly what a singleton is not supposed to be: Having many objects of it. This is especially bad if you, as shown in this example, want to make synchronized calls to a database.

Of course this is an example of a bad usage of a singleton. But the message of this example is: You can not rely that there is exactly one instance of a singleton in your application - especially when it comes to clustering.

Share Improve this answer Follow

answered Feb 22, 2013 at 14:34

community wiki
Uooo

4 if you don't know how to implement singleton, you shouldn't do that. If you don't know what you are doing, you should find that first, and only after that do what you need. – Dainius Oct 24, 2014 at 11:30

7 This is interesting, I have a question. So what exactly is the problem if singletons - each on a different machine/JVM - are connecting to a single database? Singletons scope is only for that particular JVM, and that is still true even in a cluster. Rather than just philosophically saying that this particular situation is bad because our intent was a single object across application I would be happy to see any technical problems that can arise because of this arrangement. – Saurabh Patil Oct 20, 2016 at 17:52

2 You describe solution for a problem that can be solved with db mechanism called Transaction. Frameworks like Spring used Singletons as default representation for beans and everyone is fine for years. You just picked wrong solution for your problem. – Mr Jedi Jul 9, 2020 at 16:37

1 "ensure that concurrent database calls do not conflict each other" - isn't it the task of a database to manage concurrent access? ACID? I=Isolation? – Thomas Weller May 26, 2021 at 16:38

45

1. It is easily (ab)used as a global variable.
2. Classes that depend on singletons are relatively harder to unit test in isolation.

Share Improve this answer Follow

edited May 14, 2017 at 21:26

community wiki
2 revs, 2 users 67%
jop

38

Singleton is not about single instance!

Unlike other answers I don't want to talk about what is wrong with Singletons but to show you how powerful and awesome they are when used correctly!

- **Problem**: Singletons can be a challenge in multi-threaded environments **Solution**: Use a single threaded bootstrap process to initialize all the dependencies of your singleton.
- **Problem**: It is hard to mock singletons.
  Solution: Use the Factory pattern for mocking

```
```

You can map `MyModel` to a `TestMyModel` class that inherits from it, everywhere when `MyModel` will be injected you will get `TestMyModel` instead.

- **Problem**: Singletons can cause memory leaks as they are never disposed of.
  Solution: Well, dispose of them! Implement a callback in your app to properly dispose of singletons. You should remove any data linked to them and finally: remove them from the Factory.

As I stated in the title, singletons are not about having a single instance.

- **Singletons improves readability**: You can look at your class and see what singleton it injected to figure out what it's dependencies are.
- **Singletons improves maintenance**: Once you remove a dependency from a class you just deleted some singleton injection, you don't need to go and edit a big link of other classes that just moved your dependency around (This is smelly code for me @Jim Burger)
- **Singletons improves memory and performance**: When something happens in your application, and it takes a long chain of callbacks to deliver, you are wasting memory and performance. By using a Singleton you are cutting out the middle man, and improving your performance and memory usage (by avoiding unnecessary local variables allocations).

Share Improve this answer Follow

edited May 4, 2023 at 23:51

community wiki
4 revs, 4 users 79%
Ilya Gazman

3 This doesn't address my main issue with Singletons which is they allow access to global state from any of the thousands of classes in your project. – LegendLength Jan 6, 2016 at 2:39

15 That would be the purpose... – Ilya Gazman Jan 6, 2016 at 5:19

3 LegendLength Why is that wrong? For example, in my application I have a singleton `Settings` object which is accessible from any widget so that each widget knows about how to format displayed numbers. If it was not a globally accessible object I would have to inject it in constructor to each and every widget in the constructor and keep the reference to it as a member variable. This is a terrible waste of memory and time. – HiFile.app - best file manager Aug 21, 2019 at 11:00

2 Perhaps if it's immutable it's okay, but to simply have global state available to modify from anywhere in the application means that you are going to have situations where it is difficult to track down where a problem is occurring. You won't have an idea which pieces of code are actually making changes nearly as easily. – Brian Reading May 26, 2022 at 3:03

36

Monopoly is the devil and singletons with non-readonly/mutable state are the 'real' problem...

After reading Singletons are Pathological Liars as suggested in jason's answer I came across this little tidbit that provides the best presented example of **how** singletons are often misused.

> Global is bad because:
>
> - a. It causes namespace conflict

> - b. It exposes the state in a unwarranted fashion
>
> When it comes to Singletons
>
> - a. The explicit OO way of calling them, prevents the conflicts, so point a. is not an issue
> - b. Singletons without state are (like factories) are not a problem. Singletons with state can again fall in two categories, those which are immutable or write once and read many (config/property files). These are not bad. Mutable Singletons, which are kind of reference holders are the ones which you are speaking of.

In the last statement he's referring to the blog's concept of 'singletons are liars'.

How does this apply to Monopoly?

To start a game of monopoly, first:

- we establish the rules first so everybody is on the same page
- everybody is given an equal start at the beginning of the game
- only one set of rules is presented to avoid confusion

- the rules aren't allowed to change throughout the game

Now, for anybody who hasn't **really** played monopoly, these standards are ideal at best. A defeat in monopoly is hard to swallow because, monopoly is about money, if you lose you have to painstakingly watch the rest of the players finish the game, and losses are usually swift and crushing. So, the rules usually get twisted at some point to serve the self-interest of some of the players at the expense of the others.

So you're playing monopoly with friends Bob, Joe, and Ed. You're swiftly building your empire and consuming market share at an exponential rate. Your opponents are weakening and you start to smell blood (figuratively). Your buddy Bob put all of his money into gridlocking as many low-value properties as possible but his isn't receiving a high return on investment the way he expected. Bob, as a stroke of bad luck, lands on your Boardwalk and is excised from the game.

Now the game goes from friendly dice-rolling to serious business. Bob has been made the example of failure and Joe and Ed don't want to end up like 'that guy'. So, being the leading player you, all of a sudden, become the enemy. Joe and Ed start practicing under-the-table trades, behind-the-back money injections, undervalued house-swapping and generally anything to weaken you as a player until one of them rises to the top.

Then, instead of one of them winning, the process starts all over. All of a sudden, a finite set of rules becomes a moving target and the game degenerates into the type of social interactions that would make up the foundation of every high-rated reality TV show since Survivor. Why, because the rules are changing and there's no consensus on how/why/what they're supposed to represent, and more importantly, there's no one person making the decisions. Every player in the game, at that point, is making his/her own rules and chaos ensues until two of the players are too tired to keep up the charade and slowly give up.

So, if a rulebook for a game accurately represented a singleton, the monopoly rulebook would be an example of abuse.

How does this apply to programming?

Aside from all of the obvious thread-safety and synchronization issues that mutable singletons present... If you have one set of data, that is capable of being read/manipulated by multiple different sources concurrently and exists during the lifetime of the application execution, it's probably a good time to step back and ask "am I using the right type of data structure here".

Personally, I have seen a programmer abuse a singleton by using it as some sort of twisted cross-thread database store within an application. Having worked on the code directly, I can attest that it was a slow (because of all the thread locks needed to make it thread-safe) and a nightmare to work on (because of the unpredictable/intermittent nature of synchronization bugs), and nearly impossible to test under 'production' conditions. Sure, a system could have been developed using polling/signaling to overcome some of the performance issues but that wouldn't solve the issues with testing and, why bother when a 'real' database can already accomplish the same functionality in a much more robust/scalable manner.

A Singleton is *only* an option if you need what a singleton provides. A write-one read-only instance of an object. That same rule should cascade to the object's properties/members as well.

Share Improve this answer Follow

edited May 23, 2017 at 12:26

community wiki
5 revs, 2 users 97%
Evan Plaice

---

1 What if singleton cashes some data? – Yola Dec 11, 2014 at 6:54

@Yola It would cut down on the number of writes if the singleton was scheduled to update on a predetermined and/or fixed schedule, thereby reducing thread contention. Still, you wouldn't be able to accurately test any of the code interacting with the singleton unless you mock a non-singleton instance that simulates the same usage. The TDD

folks would probably have a fit but it would work.
– Evan Plaice Dec 15, 2014 at 5:35

@EvanPlaice: Even with a mock, you'd have some issues with code that says `Singleton.getInstance()`. A language that supports reflection might be able to work around that by setting the field where the One True Instance is stored. IMO, though, tests become a bit less trustworthy once you've taken to monkeying around with another class's private state. – cHao Jan 9, 2015 at 17:39

26

See Wikipedia Singleton_pattern

> It is also considered an anti-pattern by some people, who feel that it is overly used, introducing unnecessary limitations in situations where a sole instance of a class is not actually required.[1][2][3][4]

References (only relevant references from the article)

1. ^ Alex Miller. Patterns I hate #1: Singleton, July 2007
2. ^ Scott Densmore. Why singletons are evil, May 2004
3. ^ Steve Yegge. Singletons considered stupid, September 2004
4. ^ J.B. Rainsberger, IBM. Use your singletons wisely, July 2001

Share Improve this answer edited Apr 8, 2016 at 9:19

Follow

community wiki
3 revs, 2 users 86%
GUI Junkie

11 A description about the pattern doesn't explain why it is considered as evil... – Jrgns Sep 26, 2008 at 9:31

2 Hardly fair: "It is also considered an anti-pattern by some people, who feel that it is overly used, introducing unnecessary limitations in situations where a sole instance of a class is not actually required." Look at the references... Anyway there's RTFM to me. – GUI Junkie Sep 26, 2008 at 10:58

25

My answer on how Singletons are bad is always, "they are hard to do right". Many of the foundational components of languages are singletons (classes, functions, namespaces and even operators), as are components in other aspects of computing (localhost, default route, virtual filesystem, etc.), and it is not by accident. While they cause trouble and frustration from time to time, they also can make a lot of things work a LOT better.

The two biggest screw ups I see are: treating it like a global & failing to define the Singleton closure.

Everyone talks about Singleton's as globals, because they basically are. However, much (sadly, not all) of the badness in a global comes not intrinsically from being global, but how you use it. Same goes for Singletons. Actually more so as "single instance" really doesn't need to mean "globally accessible". It is more a natural byproduct, and given all the bad that we know comes from it, we shouldn't be in such a hurry to exploit global accessibility. Once programmers see a Singleton they seem to always access it directly through its instance method. Instead, you should navigate to it just like you would any other object. Most code shouldn't even be aware it is dealing with a Singleton (loose coupling, right?). If only a small bit of code accesses the object like it is a global, a lot of harm is undone. I recommend enforcing it by restricting access to the instance function.

The Singleton context is also really important. The defining characteristic of a Singleton is that there is "only one", but the truth is it is "only one" within some kind of context/namespace. They are usually one of: one per thread, process, IP address or cluster, but can also be one per processor, machine, language namespace/class loader/whatever, subnet, Internet, etc.

The other, less common, mistake is to ignore the Singleton lifestyle. Just because there is only one doesn't mean a Singleton is some omnipotent "always was and always will be", nor is it generally desirable (objects without a begin and end violate all kinds of useful assumptions in code, and should be employed only in the most desperate of circumstances.

If you avoid those mistakes, Singletons can still be a PITA, bit it is ready to see a lot of the worst problems are significantly mitigated. Imagine a Java Singleton, that is explicitly defined as once per classloader (which means it needs a thread safety policy), with defined creation and destruction methods and a life cycle that dictates when and how they get invoked, and whose "instance" method has package protection so it is generally accessed through other, non-global objects. Still a potential source of trouble, but certainly much less trouble.

Sadly, rather than teaching good examples of how to do Singletons. We teach bad examples, let programmers run off using them for a while, and then tell them they are a bad design pattern.

Share Improve this answer Follow

edited Mar 10, 2014 at 20:42

community wiki
2 revs, 2 users 96%
Christopher Smith

20

It's not that singletons themselves are bad but the GoF design pattern is. The only really argument that is valid is that the GoF design pattern doesn't lend itself in regards to testing, especially if tests are run in parallel.

Using a single instance of an class is a valid construct as long as you apply the following means in code:

1. Make sure the class that will be used as a singleton implements an interface. This allows stubs or mocks to be implemented using the same interface
2. Make sure that the Singleton is thread-safe. That's a given.
3. The singleton should be simple in nature and not overly complicated.
4. During the runtime of you application, where singletons need to be passed to a given object, use a class factory that builds that object and have the class factory pass the singleton instance to the class that needs it.
5. During testing and to ensure deterministic behavior, create the singleton class as separate instance as either the actual class itself or a stub/mock that implements its behavior and pass it as is to the class that requires it. Don't use the class factor that creates that object under test that needs the singleton during test as it will pass the single global instance of it, which defeats the purpose.

We've used Singletons in our solutions with a great deal of success that are testable ensuring deterministic behavior in parallel test run streams.

Share Improve this answer Follow

edited Aug 12, 2012 at 16:48

community wiki

2 revs, 2 users 92%
user1578119

+1, *Finally* an answer that addresses *when* a singleton may be valid. – Pacerier Jun 25, 2014 at 1:30

19

I'd like to address the 4 points in the accepted answer, hopefully someone can explain why I'm wrong.

1. Why is hiding dependencies in your code bad? There are already dozens of hidden dependencies (C runtime calls, OS API calls, global function calls), and singleton dependencies are easy to find (search for instance()).

   "Making something global to avoid passing it around is a code smell." Why isn't passing something around to avoid making it a singleton a code smell?

   If you're passing an object through 10 functions in a call stack just to avoid a singleton, is that so great?

2. Single Responsibility Principle: I think this is a bit vague and depends on your definition of responsibility. A relevant question would be, why does adding this *specific* "responsibility" to a class matter?

3. Why does passing an object to a class make it more tightly coupled than using that object as a singleton from within the class?

4. Why does it change how long the state lasts? Singletons can be created or destroyed manually, so the control is still there, and you can make the lifetime the same as a non-singleton object's lifetime would be.

Regarding unit tests:

- not all classes need to be unit tested
- not all classes that need to be unit tested need to change the implementation of the singleton
- if they *do* need be unit tested and do need to change the implementation, it's easy to change a class from using a singleton to having the singleton passed to it via dependency injection.

Share Improve this answer Follow

answered Apr 7, 2010 at 13:31

community wiki
Jon

---

1 1. All those hidden dependencies? *Those are bad too.* Hidden dependencies are always evil. But in the case of the CRT and OS, they're already there, and they're the only way to get something done. (Try writing a C program without using the runtime or OS.) That ability to do stuff vastly outweighs their negatives. Singletons in our code don't get that luxury; since they're firmly within our sphere of control and responsibility, every usage (read: every additional hidden dependency) should be justifiable as the only reasonable way

to get stuff done. Very, *very* few of them actually are. – cHao Feb 12, 2013 at 14:17

2. "Responsibility" is typically defined as "reason to change". A singleton ends up both managing its lifetime and doing its real job. If you change how the job is done *or* how the object graph is built, you have to change the class. But if you have other code build the object graph, and your class just does its real job, that initialization code can set up the object graph however it likes. Everything's much more modular and testable, because you can insert test doubles and tear everything down at will, and you're no longer relying on hidden moving parts. – cHao Feb 12, 2013 at 14:39

1 @cHao: 1. It's good that you recognize that the "ability to do stuff vastly outweighs their negatives." An example would be a logging framework. It's evil to mandate passing a global logging object by dependency injection through layers of function calls if it means that developers are discouraged from logging. Hence, singleton. 3. Your tight coupling point is only relevant if you want clients of the class to control behavior via polymorphism. That's often not the case. – Jon Feb 13, 2013 at 22:48

2 4. I'm not sure that "cannot be destroyed manually" is a logical implication of "can be only one instance". If that's how you define singleton, then we're not talking about the same thing. All classes *should not* be unit tested. That's a business decision, and sometimes time-to-market or development cost will take precedence. – Jon Feb 13, 2013 at 22:48

1 3. If you don't want polymorphism, then you don't even need an instance. You might as well make it a static class, cause you're tying yourself to a single object and a single implementation anyway -- and at least then the API isn't lying to you as much. – cHao Feb 13, 2013 at 23:50

18

Vince Huston has these criteria, which seem reasonable to me:

> Singleton should be considered only if all three of the following criteria are satisfied:
>
> - Ownership of the single instance cannot be reasonably assigned
> - Lazy initialization is desirable
> - Global access is not otherwise provided for
>
> If ownership of the single instance, when and how initialization occurs, and global access are not issues, Singleton is not sufficiently interesting.

Share Improve this answer Follow

answered Nov 23, 2009 at 9:58

community wiki
js.

14

I'm not going to comment on the good/evil argument, but I haven't used them since Spring came along. Using dependency injection has pretty much removed my requirements for singleton, servicelocators and factories. I find this a much more productive and clean environment, at least for the type of work I do (Java-based web applications).

Share Improve this answer Follow

edited Mar 10, 2014 at 20:37

community wiki
2 revs, 2 users 67%
Peter Mortensen

4 aren't Spring beans, by default, Singletons? – slim Feb 8, 2010 at 15:50

3 Yeah, but I mean 'coding' singletons. I haven't 'written a singleton' (i.e. with a private constructor yada yada yada as per design pattern) - but yes, with spring I am using a single instance. – prule Feb 16, 2010 at 19:59

5 so when others do it, it's Ok (if I will use that after), but if others do it and I will not use it, it's evil? – Dainius Oct 24, 2014 at 11:32

14

Singletons are bad from a purist point of view.

From a pratical point of view, **a singleton is a trade-off developing time vs complexity**.

If you know your application won't change that much they are pretty OK to go with. Just know that you may need to refactor things up if your requirements change in an unexpected way (which is pretty OK in most cases).

Singletons sometimes also complicate unit testing.

Share Improve this answer Follow

edited Mar 10, 2014 at 20:40

community wiki
2 revs, 2 users 67%
tacone

3 My experience is that *starting* with singletons will really hurt you even in the short run, not counting the long run. This effect might me less so if the application is already existing for some time, and probably already infested with other singletons. Starting from scratch? Avoid them at all costs! Want a single instance of an object? Let a factory manage the instantiation of this object. – Sven May 17, 2013 at 19:36

1 AFAIK Singleton *is* a factory method. Thus I don't get your recommendation. – tacone May 17, 2013 at 22:37

3 "A factory" != "A factory_method that cannot be separated from the other useful stuff in the same class" – Sven May 18, 2013 at 0:12

13

There is nothing inherently wrong with the pattern, assuming it is being used for some aspect of your model which is truly single.

I believe the backlash is due to its overuse which, in turn, is due to the fact that it's the easiest pattern to understand and implement.

Share Improve this answer Follow

answered Sep 26, 2008 at 6:07

community wiki
Ben Hoffstein

6 I think the backlash has more to do with people practicing formalized unit testing realising that they are a nightmare to deal with. – Jim Burger Sep 26, 2008 at 6:15

2 Not sure why this is at -1. It's not the most descriptive answer but he's not wrong either. – Tim Frey Sep 26, 2008 at 17:28

12

Singleton is a pattern and can be used or abused just like any other tool.

The bad part of a singleton is generally the user (or should I say the inappropriate use of a singleton for things it is not designed to do). The biggest offender is using a singleton as a fake global variable.

Share Improve this answer Follow

edited Mar 10, 2014 at 20:40

community wiki
3 revs, 2 users 79%
Loki Astari

1 Thanks Loki :) Exactly my point. The whole witch hunt reminds me of the goto debate. Tools are for people who know how to use them; using a tool you don't like may be dangerous for you so avoid it but DON'T tell others not to use / not to learn to use it properly. I'm not exactly "pro-singleton" but I like the fact that I have tool like that and I can feel where

it is appropriate to use it. Period. – dkellner Dec 29, 2015 at 16:58

10

When you write code using singletons, say, a logger or a database connection, and afterwards you discover you need more than one log or more than one database, you’re in trouble.

Singletons make it very hard to move from them to regular objects.

Also, it’s too easy to write a non-thread-safe singleton.

Rather than using singletons, you should pass all the needed utility objects from function to function. That can be simplified if you wrap all them into a helper object, like this:

```
void some_class::some_function(parameters,
service_provider& srv)
{
    srv.get<error_logger>().log("Hi there!");
    this->another_function(some_other_parameters,
srv);
}
```

Share Improve this answer Follow

answered Sep 26, 2008 at 6:20

community wiki
Roman Odaisky

5 passing argument for every method is brilliant, it should go to at least top 10 ways how to pollute your api. – Dainius Oct 24, 2014 at 11:34

That’s an obvious drawback, and one that should have been handled by the language, by means of arguments that somehow propagate into nested calls, but absent such support, it has to be done manually. Consider that even natural singletons, like an object representing the system clock, have potential for trouble. For example, how are you going to cover clock-dependent code (think multiple tariff electricity meter) with tests? – Roman Odaisky Oct 26, 2014 at 13:23

depends what you are testing, if you aren't testing singleton, why you care how it behaves, if your 2 remote object depends on each other behavior, it's not singleton problem.. – Dainius Oct 27, 2014 at 16:32

Could you provide which language would propagation into nested calls? – Dainius Oct 27, 2014 at 16:33

1 If you have one module that depends on another and you don't know /can't moc it, you will have hard irregardless of is it singleton or not. People are using inheritance way to often, where they should use composition, but you don't say that inheritance are bad, and OOP should be avoid at all costs, because so many people are doing design mistakes there, or are you? – Dainius Oct 29, 2014 at 13:50

9

Recent article on this subject by Chris Reath at Coding Without Comments.

Note: Coding Without Comments is no longer valid. However, The article being linked to has been cloned by another user.

Link

Share Improve this answer Follow

edited Aug 6, 2022 at 14:04

community wiki
3 revs, 3 users 57%
stephenbayer

8

Too many people put objects which are not thread safe in a singleton pattern. I've seen examples of a DataContext (LINQ to SQL) done in a singleton pattern, despite the fact that the DataContext is not thread safe and is purely a unit-of-work object.

Share Improve this answer Follow

edited Mar 10, 2014 at 20:44

community wiki
2 revs, 2 users 67%
Peter Mortensen

many people write unsafe multi-threaded code, does that mean we should eliminate threads? – Dainius Oct 24, 2014 at 11:35

@Dainius: In fact, yes. IMO the default concurrency model should be multi-process, with a way for two processes to (1) pass messages to each other easily, and/or (2) share memory on an as-needed basis. Threading is useful if you want to share *everything*, but you never really want that. It could be emulated by sharing the whole address space, but that would also be considered an anti-pattern. – cHao Jan 9, 2015 at 18:20

@Dainius: And of course, that's assuming that honest-to-goodness concurrency is needed at all. Often all you want is to be able to do one thing while you wait for the OS to do something else. (Updating the UI while reading from a socket, for example.) A decent async API could make full-on threading unnecessary in a huge majority of cases. – cHao Jan 9, 2015 at 18:50

so if you have huge matrix of data, that you need process, it's better to do that in single thread, because to many people might do that wrong.. – Dainius Jan 10, 2015 at 19:19

@Dainius: In many cases, yes. (Threading could actually *slow you down*, if you add synchronization to the mix. This is a prime example of why multithreading shouldn't be most people's first thought.) In other cases, it'd be better to have two processes that share just the needed stuff. Of course, you'd have to arrange for the processes to share memory. But frankly, i see that as a *good* thing -- far better, at least, than defaulting to share-everything mode. It requires you to explicitly say (and so, ideally, to know) which parts are shared, and thus which parts need to be thread-safe. – cHao Jan 10, 2015 at 23:49

8

The problems with singletons is the issue of increased scope and therefore coupling. There is no denying that there are some of situations where you do need access to a single instance, and it can be accomplished other ways.

I now prefer to design around an inversion of control (IoC) container and allow the the lifetimes to be controlled by the container. This gives you the benefit of the classes that depend on the instance to be unaware of the fact that there is a single instance. The lifetime of the singleton can be changed in the future. Once such example I encountered recently was an easy adjustment from single threaded to multi-threaded.

FWIW, if it a PIA when you try to unit test it then it's going to PIA when you try to debug, bug fix or enhance it.

Share Improve this answer Follow

edited Mar 10, 2014 at 20:53

community wiki
2 revs, 2 users 67%
Mark Lindell

7

Singletons are NOT bad. It's only bad when you make something globally unique that isn't globally unique.

However, there are "application scope services" (think about a messaging system that makes components interact) - this CALLS for a singleton, a "MessageQueue" - class that has a method "SendMessage(...)".

You can then do the following from all over the place:

MessageQueue.Current.SendMessage(new MailArrivedMessage(...));

And, of course, do:

MessageQueue.Current.RegisterReceiver(this);

in classes that implement IMessageReceiver.

6 And if I want to create a second message queue, with a smaller scope, why shouldn't I be allowed to reuse your code to create it? A singleton prevents that. But if you'd just created a regular message queue class, and then created one global instance to act as the "application scope" one, I could create a second instance for other use. But if you make the class a singleton, I'd have to write a *second* message queue class. – Stack Overflow is garbage Nov 10, 2010 at 14:39

1 Also why shouldn't we just have a global CustomerOrderList so we can call it nicely from anywhere like MessageQueue? I believe the answer is the same for both: It's effectively making a global variable. – LegendLength Jan 6, 2016 at 2:32

6

Here is one more thing about singletons which nobody said yet.

In most cases "singletonity" is a detail of implementation for some class rather than characteristic of its interface. Inversion of Control Container may hide this characteristic from class users; you just need to mark your class as a singleton (with `@Singleton` annotation in Java for example) and that's it; IoCC will do the rest. You don't need to provide global access to your singleton instance because the access is already managed by IoCC. Thus there is nothing wrong with IoC Singletons.

GoF Singletons in opposite to IoC Singletons are supposed to expose "singletonity" in the interface through getInstance() method, and so that they suffer from everything said above.

Share Improve this answer Follow

answered Jun 1, 2010 at 21:29

community wiki
Volodymyr Frolov

3 In my opinion, "singletonity" is a run-time environment detail, and should not be considered by the programmer that wrote the class code. Rather, it is a consideration to be made by the class USER. only the USER knows how many instances are actually required. – Earth Engine Apr 12, 2013 at 4:50

6

Singletons *aren't* evil, if you use it *properly* & *minimally*. There are lot of other good design patterns which replaces the needs of singleton at some point (& also gives best results). But some programmers are unaware of those good patterns & uses the singleton for all the cases which makes the singleton evil for them.

Share Improve this answer Follow

answered Nov 10, 2013 at 12:11

community wiki
Sri

1 Awesome! Completely agree! But you could, and maybe should, elaborate much more. Such as expanding which design patterns are most commonly ignored and how to

"properly and minimally" use singletons. Easier said than done! :P – cregox Nov 16, 2013 at 10:11

Basically the alternative is to pass objects around via method parameters, rather than accessing them via global state. – LegendLength Jan 6, 2016 at 2:37

5

Because they are basically object oriented global variables, you can usually design your classes in such a way so that you don't need them.

Share Improve this answer Follow

answered Sep 26, 2008 at 6:05

community wiki
Ycros

If you class isn't limited to once instance, you'll need static members in your class managed by semaphores which comes out to pretty much the same thing! What is your proposed alternative? – Jeach Feb 18, 2010 at 15:21

I have a huge application with a "MainScreen" this screen opens up many smaller modal/non modal windows/ UI forms. IMHO I feel that the MainScreen should be a singleton so that for example, a widget somewhere in a far corner of the application wants to show its status in the MainScreen's status bar, all it has to do is MainScreen.getInstance().setStatus("Some Text"); What do you propose as an alternative ? Pass MainScreen all over the application ?? :D – Salvin Francis Apr 6, 2010 at 8:59

2 @SalvinFrancis: What i'd recommend is, you stop having objects that care about stuff they shouldn't care about and

sneak across the app to mess around with each other. :) Your example would much better done with events. When you do events correctly, a widget doesn't even have to care whether there's a MainScreen at all; it just broadcasts "hey, stuff happened", and *whatever* has subscribed to the "stuff happened" event (be it a MainScreen, a WidgetTest, or something else entirely!) decides how it wants to respond. That's how OOP is *supposed* to be done anyway. :) – cHao Feb 26, 2013 at 21:48

1 @Salvin Consider how hard it is to reason about MainScreen when debugging if it is being 'silently' updated by many components. Your example is a perfect reason why singletons are bad. – LegendLength Jan 6, 2016 at 2:35

5

Firstly a class and its collaborators should firstly perform their intended purpose rather than focusing on dependents. Lifecycle management (when instances are created and when they go out of scope) should not be part of the classes responsibility. The accepted best practice for this is to craft or configure a new component to manage dependencies using dependency injection.

Often software gets more complicated it makes sense to have multiple independent instances of the Singleton class with different state. Committing code to simply grab the singleton is wrong in such cases. Using `Singleton.getInstance()` might be ok for small simple systems but it doesn't work/scale when one might need a different instance of the same class.

No class should be thought of as a singleton but rather that should be an application of it's usage or how it is used to configure dependents. For a quick and nasty this does not matter - just luke hard coding say file paths does not matter but for bigger applications such dependencies need to be factored out and managed in more appropriate way using DI.

The problems that singleton cause in testing is a symptom of their hard coded single usage case/environment. The test suite and the many tests are each individual and separate something that is not compatible with hard coding a singleton.