Why should you prevent a class from being subclassed?

Asked 16 years, 4 months ago Modified 13 years, 10 months ago

Viewed 2k times

9

What can be reasons to prevent a class from being inherited? (e.g. using sealed on a c# class) Right now I can't think of any.

oop

Share

Improve this question

Follow

edited Aug 23, 2008 at 21:50

Chris

6,842 ● 6 ● 53 ● 67

asked Aug 23, 2008 at 21:35

FantaMango77

2,437 ● 4 ● 24 ● 27

14 Answers

Sorted by: Highest score (default)

22

Because writing classes to be substitutably extended is *damn hard* and requires you to make accurate predictions of how future users will want to extend what you've written.

Sealing your class forces them to use composition, which is much more robust.

Share Improve this answer

Follow

answered Aug 23, 2008 at 21:40

DrPizza

18.3k ● 7 ● 42 ● 53

9

How about if you are not sure about the interface yet and don't want any other code depending on the present interface? [That's off the top of my head, but I'd be interested in other reasons as well!]

Edit: A bit of googling gave the following: http://codebetter.com/blogs/patricksmacchia/archive/2008/01/05/rambling-on-the-sealed-keyword.aspx

Quoting:

There are three reasons why a sealed class is better than an unsealed class:

- **Versioning**: When a class is originally sealed, it can change to unsealed in the future without breaking compatibility. (…)
- **Performance**: (…) if the JIT compiler sees a call to a virtual method using a sealed types, the JIT compiler can produce more efficient code by calling the method non-virtually.(…)
- **Security and Predictability**: A class must protect its own state and not allow itself to ever become

corrupted. When a class is unsealed, a derived class can access and manipulate the base class's state if any data fields or methods that internally manipulate fields are accessible and not private.(…)

Share Improve this answer Follow

edited Aug 23, 2008 at 21:46

answered Aug 23, 2008 at 21:37

OysterD

6,750 ● 5 ● 35 ● 33

Meh. I strongly disagree. Why should classes care about their descendants? How does sealing a class help to manage object integrity? By forcing a programmer to reinvent the class again? Misunderstanding of the basic OOP principles is quite obvious from expressions such as "manipulate the base class's state". One doesn't manipulate class, instead, he manipulates a instance of the class. Sealing a class is no guarantee of safety. That's a perfect nonsense. – PJK Feb 13, 2011 at 21:42

8

I want to give you this message from "Code Complete":

> Inheritance - subclasses - tends to work against the primary technical imperative you have as a programmer, which is to manage complexity.For the sake of controlling complexity, you should maintain a heavy bias against inheritance.

Share Improve this answer

Follow

answered Aug 23, 2008 at 21:56

popopome

12.5k ● 15 ● 45 ● 36

2

The only legitimate use of inheritance is to define a particular case of a base class like, for example, when inherit from Shape to derive Circle. To check this look at the relation in opposite direction: is a Shape a generalization of Circle? If the answer is yes then it is ok to use inheritance.

So if you have a class for which there can not be any particular cases that specialize its behavior it should be sealed.

Also due to LSP (Liskov Substitution Principle) one can use derived class where base class is expected and this is actually imposes the greatest impact from use of inheritance: code using base class *may be given an inherited class and it still has to work as expected*. In order to protect external code when there is no obvious need for subclasses you seal the class and its clients can rely that its behavior will not be changed. Otherwise external code needs to be explicitly designed to expect possible changes in behavior in subclasses.

A more concrete example would be Singleton pattern. You need to seal singleton to ensure one can not break the "singletonness".

Share Improve this answer

answered Aug 24, 2008 at 12:34

Follow

Dima Malenko
2,835 ● 2 ● 27 ● 24

1

This may not apply to your code, but a lot of classes within the .NET framework are sealed purposely so that no one tries to create a sub-class.

There are certain situations where the internals are complex and require certain things to be controlled very specifically so the designer decided no one should inherit the class so that no one accidentally breaks functionality by using something in the wrong way.

Share Improve this answer

Follow

answered Aug 23, 2008 at 21:42

Dan Herbert
103k ● 51 ● 192 ● 221

1

@jjnguy

> Another user may want to re-use your code by sub-classing your class. I don't see a reason to stop this.

If they want to use the functionality of my class they can achieve that with containment, and they will have much less brittle code as a result.

Composition seems to be often overlooked; all too often people want to jump on the inheritance bandwagon. They

should not! Substitutability is difficult. Default to composition; you'll thank me in the long run.

Share Improve this answer

Follow

answered Aug 23, 2008 at 21:45

DrPizza

18.3k ● 7 ● 42 ● 53

It depends on if there was an interface that the superclass implemented. If there is not an interface close to the functionality you wish to extend, then composition will not work. – Jonathan Weatherhead Sep 14, 2011 at 5:01

1

I am in agreement with jjnguy... I think the reasons to seal a class are few and far between. Quite the contrary, I have been in the situation more than once where I want to extend a class, but couldn't because it was sealed.

As a perfect example, I was recently creating a small package (Java, not C#, but same principles) to wrap functionality around the memcached tool. I wanted an interface so in tests I could mock away the memcached client API I was using, and also so we could switch clients if the need arose (there are 2 clients listed on the memcached homepage). Additionally, I wanted to have the opportunity to replace the functionality altogether if the need or desire arose (such as if the memcached servers are down for some reason, we could potentially hot swap with a local cache implementation instead).

I exposed a minimal interface to interact with the client API, and it would have been awesome to extend the

client API class and then just add an implements clause with my new interface. The methods that I had in the interface that matched the actual interface would then need no further details and so I wouldn't have to explicitly implement them. However, the class was sealed, so I had to instead proxy calls to an internal reference to this class. The result: more work and a lot more code for no real good reason.

That said, I think there are potential times when you might want to make a class sealed... and the best thing I can think of is an API that you will invoke directly, but allow clients to implement. For example, a game where you can program against the game... if your classes were not sealed, then the players who are adding features could potentially exploit the API to their advantage. This is a very narrow case though, and I think any time you have full control over the codebase, there really is little if any reason to make a class sealed.

This is one reason I really like the Ruby programming language... even the core classes are open, not just to extend but to ADD AND CHANGE functionality dynamically, TO THE CLASS ITSELF! It's called monkeypatching and can be a nightmare if abused, but it's damn fun to play with!

edited Aug 23, 2008 at 22:02

answered Aug 23, 2008 at 21:57

Mike Stone
44.6k • 30 • 114 • 140

1

From an object-oriented perspective, sealing a class clearly documents the author's intent without the need for comments. When I seal a class I am trying to say that this class was designed to encapsulate some specific piece of knowledge or some specific service. It was not meant to be enhanced or subclassed further.

This goes well with the Template Method design pattern. I have an interface that says "I perform this service." I then have a class that implements that interface. But, what if performing that service relies on context that the base class doesn't know about (and shouldn't know about)? What happens is that the base class provides virtual methods, which are either protected or private, and these virtual methods are the hooks for subclasses to provide the piece of information or action that the base class does not know and cannot know. Meanwhile, the base class can contain code that is common for all the child classes. These subclasses would be sealed because they are meant to accomplish that one and only one concrete implementation of the service.

Can you make the argument that these subclasses should be further subclassed to enhance them? I would say no because if that subclass couldn't get the job done in the first place then it should never have derived from the base class. If you don't like it then you have the

original interface, go write your own implementation class.

Sealing these subclasses also discourages deep levels of inheritence, which works well for GUI frameworks but works poorly for business logic layers.

Share Improve this answer

Follow

answered Sep 5, 2008 at 1:26

Nicholas Salerno

141 • 3

0

Because you always want to be handed a reference to the class and not to a derived one for various reasons:
i. invariants that you have in some other part of your code
ii. security
etc

Also, because it's a safe bet with regards to backward compatibility - you'll never be able to close that class for inheritance if it's release unsealed.

Or maybe you didn't have enough time to test the interface that the class exposes to be sure that you can allow others to inherit from it.

Or maybe there's no point (that you see now) in having a subclass.

Or you don't want bug reports when people try to subclass and don't manage to get all the nitty-gritty details - cut support costs.

Share Improve this answer

Follow

answered Aug 23, 2008 at 21:43

kokos

43.6k ● 5 ● 37 ● 32

0

Sometimes your class interface just isn't meant to be inheirited. The public interface just isn't virtual and while someone could override the functionality that's in place it would just be wrong. Yes in general they shouldn't override the public interface, but you can insure that they don't by making the class non-inheritable.

The example I can think of right now are customized contained classes with deep clones in .Net. If you inherit from them you lose the deep clone ability.[I'm kind of fuzzy on this example, it's been a while since I worked with IClonable] If you have a true singelton class, you probably don't want inherited forms of it around, and a data persistence layer is not normally place you want a lot of inheritance.

Share Improve this answer

Follow

answered Aug 23, 2008 at 21:45

Dan Blair

2,381 ● 3 ● 21 ● 32

0

Not everything that's important in a class is asserted easily in code. There can be semantics and relationships present that are easily broken by inheriting and overriding methods. Overriding one method at a time is an easy way to do this. You design a class/object as a single meaningful entity and then someone comes along and

thinks if a method or two were 'better' it would do no harm. That may or may not be true. Maybe you can correctly separate all methods between private and not private or virtual and not virtual but that still may not be enough. Demanding inheritance of all classes also puts a huge additional burden on the original developer to foresee all the ways an inheriting class could screw things up.
I don't know of a perfect solution. I'm sympathetic to preventing inheritance but that's also a problem because it hinders unit testing.

Share Improve this answer

Follow

answered Aug 23, 2008 at 22:01

Harpreet

503 ● 5 ● 6

Inheritance should not concern the author of the original class; he's not the one that is going to get in trouble with an inherited class screwing things up. If the developer can foresee such gaping potential concerns, then the original class was not well designed well in the first place/

– Jonathan Weatherhead Sep 14, 2011 at 5:04

0

I exposed a minimal interface to interact with the client API, and it would have been awesome to extend the client API class and then just add an implements clause with my new interface. The methods that I had in the interface that matched the actual interface would then need no further details and so I wouldn't have to explicitly

> implement them. However, the class was sealed, so I had to instead proxy calls to an internal reference to this class. The result: more work and a lot more code for no real good reason.

Well, there is a reason: your code is now somewhat insulated from changes to the memcached interface.

Share Improve this answer

Follow

answered Aug 23, 2008 at 22:13

DrPizza

18.3k ● 7 ● 42 ● 53

0

> **Performance:** (…) if the JIT compiler sees a call to a virtual method using a sealed types, the JIT compiler can produce more efficient code by calling the method non-virtually.(…)

That's a great reason indeed. Thus, for performance-critical classes, `sealed` and friends make sense.

All the other reasons I've seen mentioned so far boil down to "nobody touches my class!". If you're worried someone might misunderstand its internals, you did a poor job documenting it. You can't possibly know that there's nothing useful to add to your class, or that you already know every imaginable use case for it. Even if you're right and the other developer shouldn't have used your class to solve their problem, using a keyword isn't a

great way of preventing such a mistake. Documentation is. If they ignore the documentation, their loss.

Share Improve this answer

Follow

answered Aug 23, 2008 at 22:24

Sören Kuklau

19.9k ● 8 ● 55 ● 90

0

Most of answers (when abstracted) state that sealed/finalized classes are tool to protect other programmers against potential mistakes. There is a blurry line between meaningful protection and pointless restriction. But as long as programmer is the one who is expected to understand the program, I see no hardly any reasons to restrict him from reusing parts of a class. Most of you talk about classes. But it's all about objects!

In his first post, DrPizza claims that designing inheritable class means anticipating possible extensions. Do I get it right that you think that class should be inheritable only if it's likely to be extended well? Looks as if you were used to design software from the most abstract classes. Allow me a brief explanation of how do I think when designing:

Starting from the very concrete objects, I find characteristics and [thus] functionality that they have in common and I abstract it to superclass of those particular objects. This is a way to reduce code duplicity.

Unless developing some specific product such as a framework, I should care about **my** code, not others (virtual) code. The fact that others might find it useful to

reuse my code is a nice bonus, not my primary goal. If they decide to do so, it's their responsibility to ensure validity of extensions. This applies team-wide. Up-front design is crucial to productivity.

Getting back to my idea: Your objects should primarily serve your purposes, not some possible shoulda/woulda/coulda functionality of their subtypes. Your goal is to solve given problem. Object oriented languages uses fact that many problems (or more likely their subproblems) are similar and therefore existing code can be used to accelerate further development.

Sealing a class forces people who could possibly take advantage of existing code WITHOUT ACTUALLY MODIFYING YOUR PRODUCT to reinvent the wheel. (This is a crucial idea of my thesis: Inheriting a class doesn't modify it! Which seems quite pedestrian and obvious, but it's being commonly ignored).

People are often scared that their "open" classes will be twisted to something that can not substitute its ascendants. So what? Why should you care? No tool can prevent bad programmer from creating bad software!

I'm not trying to denote inheritable classes as the ultimately correct way of designing, consider this more like an explanation of my inclination to inheritable classes. That's the beauty of programming - virtually infinite set of correct solutions, each with its own cons and pros. Your comments and arguments are welcome.

And finally, my answer to the original question: I'd finalize a class to let others know that I consider the class a leaf of the hierarchical class tree and I see absolutely no possibility that it could become a parent node. (And if anyone thinks that it actually could, then either I was wrong or they don't get me).

answered Feb 13, 2011 at 23:07

PJK

2,112 ● 3 ● 17 ● 29