Liskov substitution principle - no overriding/virtual methods?

Ask Question

53

20

My understanding of the Liskov substitution principle is that some property of the base class that is true or some implemented behaviour of the base class, should be true for the derived class as well.

I guess this would mean when a method is defined in a base class, it should never be overrided in the derived class - since then substituting the base class instead of the derived class would give different results. I guess this would also mean, having (non-pure) virtual methods is a bad thing?

I think I might have a wrong understanding of the principle. If I don't, I do not understand why is this principle good practice. Can someone explain this to me? Thanks

design-principles solid-principles liskov-substitution-principle

edited Oct 17 '10 at 13:59

Péter Török

101k 24 236 317

asked Nov 14 '09 at 18:30

Aishwar

4,589 7 45 72

Home

PUBLIC

Stack Overflow

Tags

Users

1 Thanks to everyone who answered. I think all of you contributed a lot to my understanding of how this works. I have given everyone an up-vote, I am not sure how I can determine the correct answer (everyone's answer helped me! :D) – Aishwar Nov 18 '09 at 13:48

5 Answers

51

Subclasses overriding methods in the base class are totally allowed by the Liskov Substituion Principle.

This might be simplifying it too much, but I remember it as ***"a subclass should require nothing more and promise nothing less"***

If a client is using a superclass `ABC` with a method `something(int i)`, then the client should be able to substitute any subclass of `ABC` without problems. Instead of thinking about this in terms of variable types, perhaps think about it in terms of preconditions and postconditions.

If our `something()` method in the `ABC` base class above has a relaxed precondition that permits *any* integer, then all subclasses of `ABC` **must** also permit *any* integer. A subclass `GreenABC` is not allowed to add an additional precondition to the `something()` method that requires the parameter to be a *positive* integer. This would violate the Liskov Substitution Principle (i.e., requiring more). Thus if a client is using subclass `BlueABC` and passing negative integers to `something()` the client won't break if we need to switch to `GreenABC`.

In reverse, if the base `ABC` class `something()` method has a postcondition - such as guaranteeing it will never return a value of zero - then all subclasses must also obey that same postcondition or they violate the Liskov Substitution Principle (i.e., promising less).

I hope this helps.

answered Nov 14 '09 at 19:35

dustmachine
7,799 3 21 26

2 If the value GreenABC returns is valid for the specification provided by ABC, then no, it is not in violation of the principle. – Grundlefleck Nov 14 '09 at 23:00

1 @Grundlefleck: sorry, if I don't seem to get it. So, a derived class cannot change the behaviour of the base class, only add on to it? i.e. I cannot override a method in a derived class to have a different implementation from the base class. – Aishwar Nov 15 '09 at 0:19

6 a `PizzaSharingService` interface or abstract class (base) has a `share(pizza,numSlices)` method that will return a list of slices. A precondition says that `numSlices` parameter can be any number 0-12 (inclusive, and if the number is zero then an empty list will be returned.) So the first team creates a `RoundPizzaSharingService` which cuts the pizza into triangular slices, obeying the rule regarding the number of slices being zero. The second team creates a `SquarePizzaSharingService`, but decides that if number of slices is zero it throws an exception or returns a null. This violates LSP. – dustmachine Nov 15 '09 at 1:59

5 @dustmachine, good example. IMO more examples == good... Say you have a `Rectangle` class that has `setWidth()` and `setHeight()`. A `Square` class subclasses `Rectangle` and overrides behaviour so that when a dimension is changed, the class takes care of keeping it a square shape. If someone has a reference to `Rectangle` and calls `setWidth(10); setHeight(5);` the contract for a rectangle states that the dimensions should now be 10x5, but if the reference is actually to a `Square` the dimensions will be modified to make it a square shape. The `Square` class is an example of violating LSP. – Grundlefleck Nov 15 '09 at 11:31

1 +1 - excellent description of design by contract. – james lewis Nov 17 '12 at 16:20

10

There is one popular example which says if it swims like a duck, quack likes a duck but requires batteries, then it breaks Liskov Substitution Principle.

Put it simply, you have a base Duck class which is being used by someone. Then you add hierarchy by introduction PlasticDuck with same overridden behaviors (like swimming, quacking etc.) as of a Duck but requires batteries to simulate those behaviors. This essentially means that you are introducing an extra pre-condition to the behavior of Sub Class to require batteries to do the same behavior that was earlier done by the Base Duck class without batteries. This might catch the consumer of your Duck class by surprise and might break the functionality built around the expected behavior of Base Duck class.

Here is a good link - http://lassala.net/2010/11/04/a-good-example-of-liskov-substitution-principle/

answered Oct 16 '13 at 4:42

vibhu
1,294 12 19

7

No, it tells that you should be able to use derived class in the same way as its base. There're many ways you can override a method without breaking this. A simple example, GetHashCode() in C# is in base for ALL classes, and still ALL of them can be used as "object" to calculate the hash code. A classic example of breaking the rule, as far as I remember, is derivin Square from Rectangle, since Square can't have both Width and Height - because setting one would change another and thus it's no more conforms to Rectangle

rules. You can, however, still have base Shape with .GetSize() since ALL shapes can do this - and thus any derived shape can be substituted and used as Shape.

answered Nov 14 '09 at 18:38

queen3
10.9k 7 48 104

hey, so you are saying that for any object in C# GetHashCode() can be called and you can be cast to object and still the GetHashCode() would return the same value. Or am I misunderstanding this? I am guessing the GetHashCode is implemented in the Object, and the derived objects don't override this. I don't see how this corresponds to the square/rectangle example. Can you please explain? – Aishwar Nov 14 '09 at 22:59

2 I think that queen3 means that many classes will need to override `GetHashCode()` , but clients that call it (like hash tables) won't know that they're not calling the method defined for Object. Since the contract is still met, then no violation has taken place. However, with the Square/Rectangle example, clients might be able to tell what the class is of the instance they have been passed, or might fail in some way, so a violation *has* taken place. – quamrana Nov 14 '09 at 23:16

4 No, not same value, but some value that satisfies base class contract about this value. As for shapes and hashes, see, if for derived SomeObject you can only call GetHashCode() *after* ToString() (for some reason), you break contract - now callers have to know if it's object or your SomeObject. Now you can't pass SomeObject to callers that expect object. Same with shapes - you can't pass Square to callers that expect Rectangle because they'll try to set Width and won't expect it changes Height. You can's substitute one class for another. Thus, LSP is broken. – queen3 Nov 15 '09 at 9:39

3

Overriding breaks Liskov Substitution Principle if you change any behavior defined by a base method. Which means that:

1. The weakest precondition for a child method should be not stronger than for the base method.

2. A postcondition for the child method implies a postcondition for the parent method. Where a postcondition is formed by: *a)* all side effects caused by a method execution and *b)* type and value of a returned expression.

From these two requirements you can imply that any new functionality in a child method that does not affect what is expected from a super method does not violate the principle. These conditions allow you to use a subclass instance where a superclass instance is required.

If these rules are not obeyed a class violates LSP. A classical example is the following hierarchy: class `Point(x,y)` , class `ColoredPoint(x,y,color)` that extends `Point(x,y)` and overridden method `equals(obj)` in `ColoredPoint` that reflects equality by color. Now if one have an instance of `Set<Point>` he can assume that two points with the same coordinates are equal in this set. Which is not the case with the overridden method `equals` and, in general, there is just no way to extend an instantiable class and add an aspect used in `equals` method without breaking LSP.

Thus every time you break this principle you implicitly introduce a potential bug that reveals when invariant for a parent class that is expected by the code is not satisfied. However, in real world often there is no obvious design solution that does not violate LSP, so one can use, for example, `@ViolatesLSP` class annotation to warn a client that it is not safe to use class instances in a polymorphic set or in any other kind of cases that rely on the Liskov substitution principle.

answered Apr 27 '11 at 21:24

Vitalii Fedorenko
79.4k 21 135 109

What would be wrong with specifying that X.Equals(Y) must return true if X and Y are the same instance, must always return the same value for any particular pair of fully-constructed instances X and Y, must return false if X and Y will ever behave as different object instances, and may return true if any combination of references to X could be replaced Y or vice

versa without affecting program behavior. To be sure, a lot of .net implementations of Equals don't work that way, but such a definition of Equals would be useful and would fully conform to the LSP. – supercat Sep 12 '11 at 22:11

1 The problem is that LSP does not imply that `equals must return false if X and Y will ever behave as different object instances` , but only that a descendant must behave like its parent. For example, if you put a member of the super class in a hash table as the key and then look it up using a subclass instance, you will not find it, because in your case they will be not equal. – Vitalii Fedorenko Sep 13 '11 at 1:27

It is often useful to have classes define an equivalence relation which is less specific than what should be implied by Object.Equals. Such equivalence relations may only be usable with objects derived from a certain type. If an equivalence relation is defined that specifies that two animals with the same Name should be considered equal, it would be a violation of the LSP for a SiameseCat named "Fred" not to compare equal to a Cat named "Fred or, for that matter, an Aardvark named "Fred". Unless one explicitly specifies an equivalence relation, though... – supercat Sep 13 '11 at 3:25

...the default equivalence relation should generally not regard any object as equivalent to any object of another class, regardless of class relationship. An equivalence relation which regards objects as equal only if one may be safely substituted for the other can be usefully applied (e.g. for caching or interning) even if one knows nothing about the objects in question. Other types of equivalence relations (e.g. matching on a "Name" field) should only be used in contexts where they have particular semantic meaning. – supercat Sep 13 '11 at 3:33

1

I think that you're literally correct in the way you describe the principle and only overriding pure virtual, or abstract methods will ensure that you don't violate it.

However, if you look at the principle from a client's point of view, that is, a method that takes a reference to the base class. If this method cannot tell (and certainly does not attempt to and does not need to find out) the class of any instance that is passed in, then you are also not violating the principle. So it may not matter that you override

a base class method (some sorts of decorators might do this, calling the base class method in the process).

If a client seems to need to find out the class of an instance passed in, then you're in for a maintenance nightmare, as you should really just be adding new classes as part of your maintenance effort, not modifying an existing routine. (see also OCP)

answered Nov 14 '09 at 22:55

quamrana
14.3k 8 40 54

So, in the commonly used square and rectangle example, it is not a violation for Square to inherit from Rectangle. It is a violation if the setWidth property of the rectangle, tried to identify it's real type (in case it was a Square passed to a function taking a Rectangle in) and tried to implement special logic for that case. Is this correct? – Aishwar Nov 14 '09 at 23:10

1 @aip.cd.aish: Its not a violation if clients can't tell, and it is a violation if a client has to employ special logic to fix things. Its much better if methods of classes either don't care what their type is, or can safely assume that their type is the one on which the method was originally defined. – quamrana Nov 14 '09 at 23:32

@quamrana: Based on your reply to my comment to queen3's answer, a derived class cannot change the behaviour of the base class, only add on to it? i.e. I cannot override a method in a derived class to have a different implementation from the base class. Is this what the principle states (in addition to the part about a method should not try to identify what class it actually is)? – Aishwar Nov 15 '09 at 0:24

1 @aip.cd.aish: You are not going to easily solve this if you think about this in terms of implementations. Better to think in terms of what the client expects. There are lots of things that you *can* do in an implementation that don't violate the expectations of a client, especially if a client has low or no expectations. Easy examples are the `GetHasCode()` or `Shape::GetArea()` in which its difficult for the client to detect faults. A Harder example is with Square and Rect where to conform you might have only Rectangle having `setHeight` and `setWidth` and only Square having `setSize` . – quamrana Nov 15 '09 at 22:14