


[Home](#)[PUBLIC](#)[Stack Overflow](#)[Tags](#)[Users](#)[Jobs](#)**Teams**
Q&A for work[Learn More](#)

What is an example of the Liskov Substitution Principle?

[Ask Question](#)
780

I have heard that the Liskov Substitution Principle (LSP) is a fundamental principle of object oriented design. What is it and what are some examples of its use?

[oop](#)[definition](#)[solid-principles](#)[design-principles](#)[liskov-substitution-principle](#)
357[edited Sep 1 '17 at 15:57](#)[Steve Chambers](#)**22.5k** 11 102 146[asked Sep 11 '08 at 15:17](#)[NotMyself](#)**15.2k** 15 49 71

More examples of LSP adherence and violation [here](#) – [StuartLC](#) May 15 '15 at 13:20

This question has infinitely many good answers and so is *too broad*. – [Raedwald](#) Dec 16 '18 at 15:30

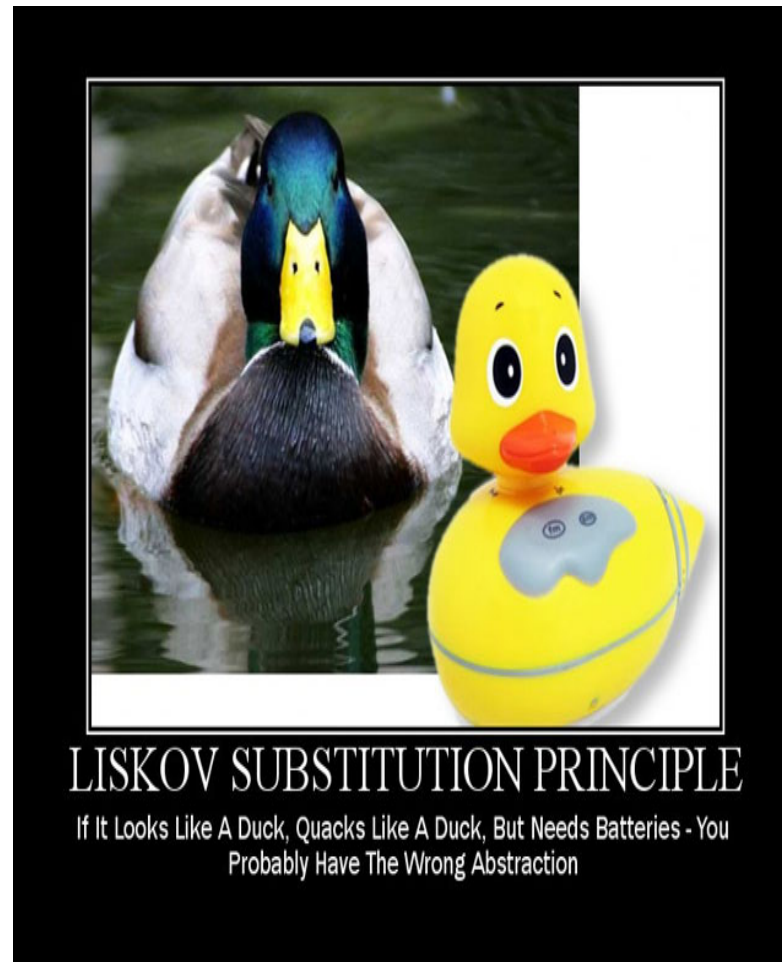
28 Answers

750

A great example illustrating LSP (given by Uncle Bob in a podcast I heard recently) was how sometimes something that sounds right in natural language doesn't quite work in code.

In mathematics, a `Square` is a `Rectangle`. Indeed it is a specialization of a rectangle. The "is a" makes you want to model this with inheritance. However if in code you made `Square` derive from `Rectangle`, then a `Square` should be usable anywhere you expect a `Rectangle`. This makes for some strange behavior.

Imagine you had `SetWidth` and `SetHeight` methods on your `Rectangle` base class; this seems perfectly logical. However if your `Rectangle` reference pointed to a `Square`, then `SetWidth` and `SetHeight` doesn't make sense because setting one would change the other to match it. In this case `Square` fails the Liskov Substitution Test with `Rectangle` and the abstraction of having `Square` inherit from `Rectangle` is a bad one.



Y'all should check out the other priceless [SOLID Principles Motivational Posters](#).

edited Mar 25 at 18:56



Josh Downes

39 7

answered Feb 25 '09 at 4:44




m-sharp

11.4k 1 20 25

12 @m-sharp What if it's an immutable Rectangle such that

instead of `SetWidth` and `SetHeight`, we have the methods `GetWidth` and `GetHeight` instead? – [Pacerier](#) Apr 26 '12 at 19:28

-
- 110 Moral of the story: model your classes based on behaviours not on properties; model your data based on properties and not on behaviours. If it behaves like a duck, it's certainly a bird. – [Skivvz](#) May 19 '12 at 21:43
-
- 146 Well, a square clearly IS a type of rectangle in the real world. Whether we can model this in our code depends on the spec. What the LSP indicates is that subtype behavior should match base type behavior as defined in the base type specification. If the rectangle base type spec says that height and width can be set independently, then LSP says that square cannot be a subtype of rectangle. If the rectangle spec says that a rectangle is immutable, then a square can be a subtype of rectangle. It's all about subtypes maintaining the behavior specified for the base type. – [SteveT](#) Sep 24 '12 at 15:46 
-
- 50 @Pacerier there is no issue if it's immutable. The real issue here is that we are not modeling rectangles, but rather "reshapable rectangles," i.e., rectangles whose width or height can be modified after creation (and we still consider it to be the same object). If we look at the rectangle class in this way, it is clear that a square is not a "reshapable rectangle", because a square cannot be reshaped and still be a square (in general). Mathematically, we don't see the problem because mutability doesn't even make sense in a mathematical context. – [asmeurer](#) Jan 20 '13 at 6:13
-
- 9 I have one question about the principle. Why would be the problem if `Square.setWidth(int width)` was implemented like this: `this.width = width; this.height = width;` ? In this case it is guaranteed that the width equals the height. – [MC Emperor](#) Oct 28 '15 at 0:45
-



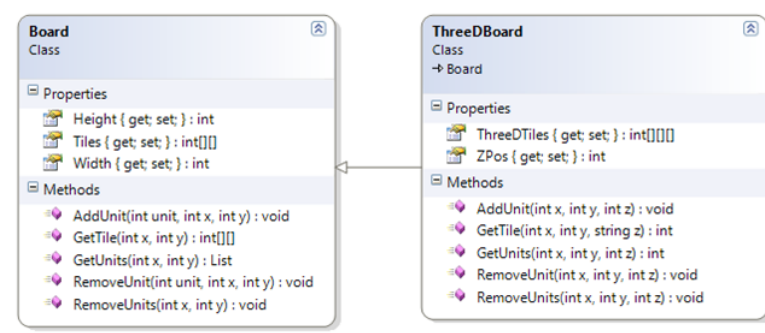
The Liskov Substitution Principle (LSP, [lsp](#)) is a concept in Object Oriented Programming that states:

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

At its heart LSP is about interfaces and contracts as well as how to decide when to extend a class vs. use another strategy such as composition to achieve your goal.

The most effective way I have seen to illustrate this point was in [Head First OOA&D](#). They present a scenario where you are a developer on a project to build a framework for strategy games.

They present a class that represents a board that looks like this:



All of the methods take X and Y coordinates as parameters to locate the tile position in the two-dimensional array of `Tiles`. This will allow a game developer to manage units in the board during the course of the game.

The book goes on to change the requirements to say that the game frame work must also support 3D game boards

to accommodate games that have flight. So a `ThreeDBoard` class is introduced that extends `Board`.

At first glance this seems like a good decision. `Board` provides both the `Height` and `Width` properties and `ThreeDBoard` provides the `Z` axis.

Where it breaks down is when you look at all the other members inherited from `Board`. The methods for `AddUnit`, `GetTile`, `GetUnits` and so on, all take both `X` and `Y` parameters in the `Board` class but the `ThreeDBoard` needs a `Z` parameter as well.

So you must implement those methods again with a `Z` parameter. The `Z` parameter has no context to the `Board` class and the inherited methods from the `Board` class lose their meaning. A unit of code attempting to use the `ThreeDBoard` class as its base class `Board` would be very out of luck.

Maybe we should find another approach. Instead of extending `Board`, `ThreeDBoard` should be composed of `Board` objects. One `Board` object per unit of the `Z` axis.

This allows us to use good object oriented principles like encapsulation and reuse and doesn't violate LSP.

edited Feb 8 '17 at 14:07



Community ♦

1 1

answered Sep 11 '08 at 15:34



NotMyself

15.2k 15 49 71

-
- 8 See also [Circle-Ellipse Problem](#) on Wikipedia for a similar but simpler example. – Brian Oct 21 '11 at 17:55

Requote from @NotMySelf: "I think the example is simply to demonstrate that inheriting from board does not make sense"

with in the context of ThreeDBoard and all of the method signatures are meaningless with a Z axis.". – [Contango](#) Jun 5 '13 at 16:40

- 1 So if we add another method to a Child class but all the functionality of Parent still makes sense in the Child class would it be breaking LSP? Since on one hand we modified the interface for using the Child a bit on the other hand if we up cast the Child to be a Parent the code that expects a Parent would work fine. – [Nickolay Kondratyev](#) Jun 18 '13 at 16:45
- 3 This is an anti-Liskov example. Liskov makes us to derive Rectangle from the Square. More-parameters-class from less-parameters-class. And you have nicely shown that it is bad. It is really a good joke to have marked as an answer and to have been upvoted 200 times an anti-liskov answer for liskov question. Is Liskov principle a fallacy really? – [Gangnus](#) Oct 18 '15 at 8:40
- 3 I've seen inheritance work the wrong way. Here is an example. The base class should be 3DBoard and the derived class Board. The Board still has a Z axis of $\text{Max}(Z) = \text{Min}(Z) = 1$ – [Paulustrious](#) Aug 5 '17 at 14:52



LSP concerns invariants.

The classic example is given by the following pseudo-code declaration (implementations omitted):

```
class Rectangle {
    int getHeight()
    void setHeight(int value)
    int getWidth()
    void setWidth(int value)
}

class Square : Rectangle { }
```

Now we have a problem although the interface matches. The reason is that we have violated invariants stemming from the mathematical definition of squares and

rectangles. The way getters and setters work, a `Rectangle` should satisfy the following invariant:

```
void invariant(Rectangle r) {
    r.setHeight(200)
    r.setWidth(100)
    assert(r.getHeight() == 200 and r.getWidth() == 100)
}
```

However, this invariant *must* be violated by a correct implementation of `Square`, therefore it is not a valid substitute of `Rectangle`.

edited May 6 '18 at 11:30

answered Sep 12 '08 at 13:45



[Konrad Rudolph](#)

407k 101 799 1043


29 And hence the difficulty of using "OO" to model anything we might want to actually model. – [DrPizza](#) Nov 30 '09 at 6:54

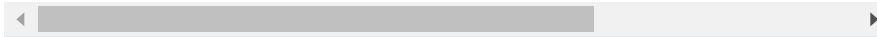
9 @DrPizza: Absolutely. However, two things. Firstly, such relationships can *still* be modelled in OOP, albeit incompletely or using more complex detours (pick whichever suits your problem). Secondly, there's no better alternative. Other mappings/modellings have the same or similar problems. ;-) – [Konrad Rudolph](#) Nov 30 '09 at 9:11

5 @NickW In some cases (but not in the above) you can simply invert the inheritance chain – logically speaking, a 2D point is a 3D point, where the third dimension is disregarded (or 0 – all points lie on the same plane in 3D space). But this is of course not really practical. In general, this is one of the cases where inheritance doesn't really help, and no natural relationship exists between the entities. Model them separately (at least I don't know of a better way). – [Konrad Rudolph](#) Jan 24 '12 at 9:49

7 OOP is meant to model behaviours and not data. Your classes violate encapsulation even before violating LSP. – [Sklivz](#) May

19 '12 at 21:47

- 1 @Skivvz I agree. It was hard to come up with a concise example though, and this one is the stock LSP example (you'll find it in several books). But yes, while common, this is far from good OOP. – [Konrad Rudolph](#) May 20 '12 at 8:37 



77

Substitutability is a principle in object-oriented programming stating that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S

let's do a simple example in Java:

Bad example

```
public class Bird{
    public void fly(){}
}
public class Duck extends Bird{}
```

The duck can fly because of it is a bird, But what about this:

```
public class Ostrich extends Bird{}
```

Ostrich is a bird, But it can't fly, Ostrich class is a subtype of class Bird, But it can't use the fly method, that means that we are breaking LSP principle.

Good example

```
public class Bird{
}
public class FlyingBirds extends Bird{
    public void fly(){}
}
```

```

}
public class Duck extends FlyingBirds{}
public class Ostrich extends Bird{}

```

edited Nov 21 '18 at 15:28

answered Jul 4 '17 at 19:58



Maysara Alhindi

1,245 14 26

-
- 2 Nice example, but what would you do if the client has `Bird bird` . You have to cast the object to `FlyingBirds` to make use of `fly`, which isn't nice right? – [Moody](#) Nov 20 '17 at 1:05
-
- 8 No. If the client has `Bird bird` , that means it cannot use `fly()` . That's it. Passing a `Duck` does not change this fact. If the client has `FlyingBirds bird` , then even if it gets passed a `Duck` it should always work the same way. – [Steve Chamillard](#) Feb 18 '18 at 18:56
-
- 3 Wouldn't this also serve as a good example for Interface Segregation? – [Saharsh](#) Mar 5 at 18:50
-



71

Robert Martin has an excellent [paper on the Liskov Substitution Principle](#). It discusses subtle and not-so-subtle ways in which the principle may be violated.



Some relevant parts of the paper (note that the second example is heavily condensed):

A Simple Example of a Violation of LSP

One of the most glaring violations of this principle is the use of C++ Run-Time Type Information (RTTI) to select

a function based upon the type of an object. i.e.:

```
void DrawShape(const Shape& s)
{
    if (typeid(s) == typeid(Square))
        DrawSquare(static_cast<Square&>(s));
    else if (typeid(s) == typeid(Circle))
        DrawCircle(static_cast<Circle&>(s));
}
```

Clearly the `DrawShape` function is badly formed. It must know about every possible derivative of the `Shape` class, and it must be changed whenever new derivatives of `Shape` are created. Indeed, many view the structure of this function as anathema to Object Oriented Design.

Square and Rectangle, a More Subtle Violation.

However, there are other, far more subtle, ways of violating the LSP. Consider an application which uses the `Rectangle` class as described below:

```
class Rectangle
{
public:
    void SetWidth(double w) {itsWidth=w;}
    void SetHeight(double h) {itsHeight=w;}
    double GetHeight() const {return itsHeight;}
    double GetWidth() const {return itsWidth;}
private:
    double itsWidth;
    double itsHeight;
};
```

[...] Imagine that one day the users demand the ability to manipulate squares in addition to rectangles. [...]

Clearly, a square is a rectangle for all normal intents and purposes. Since the ISA relationship holds, it is logical to model the `Square` class as being derived from `Rectangle`. [...]

`Square` will inherit the `SetWidth` and `SetHeight` functions. These functions are utterly inappropriate for a `Square`, since the width and height of a square are identical. This should be a significant clue that there is a problem with the design. However, there is a way to sidestep the problem. We could override `SetWidth` and `SetHeight`. [...]

But consider the following function:

```
void f(Rectangle& r)
{
    r.SetWidth(32); // calls Rectangle::SetWidth
}
```

If we pass a reference to a `Square` object into this function, the `Square` object will be corrupted because the height won't be changed. This is a clear violation of LSP. The function does not work for derivatives of its arguments.

[...]

edited Aug 16 '17 at 9:35



shA.t

13.3k 4 38 74

answered Sep 12 '08 at 13:34



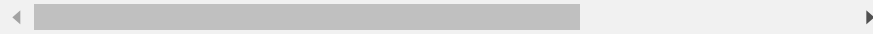
Phillip Wells

4,369 6 34 39

-
- 14 Way late, but I thought this was an interesting quote in that paper: Now the rule for the preconditions and postconditions for derivatives, as stated by Meyer is:

...when redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one. If a child-class precondition is stronger than a parent class pre-condition, you couldn't substitute a child for a parent without violating the pre-condition. Hence LSP. – [user2023861](#) Feb 11 '15 at 15:31

@user2023861 You are perfectly right. I'll write an answer based on this. – [inf3rno](#) Oct 9 '17 at 4:00



40



LSP is necessary where some code thinks it is calling the methods of a type T , and may unknowingly call the methods of a type S , where S extends T (i.e. S inherits, derives from, or is a subtype of, the supertype T).

For example, this occurs where a function with an input parameter of type T , is called (i.e. invoked) with an argument value of type S . Or, where an identifier of type T , is assigned a value of type S .

```
val id : T = new S() // id thinks it's a T, but is a S
```

LSP requires the expectations (i.e. invariants) for methods of type T (e.g. `Rectangle`), not be violated when the methods of type S (e.g. `Square`) are called instead.

```
val rect : Rectangle = new Square(5) // thinks it's a Rect
val rect2 : Rectangle = rect.setWidth(10) // height is 10,
```

Even a type with *immutable fields* still has invariants, e.g. the *immutable* `Rectangle` setters expect dimensions to be independently modified, but the *immutable* `Square` setters violate this expectation.

```
class Rectangle( val width : Int, val height : Int )
{
    def setWidth( w : Int ) = new Rectangle(w, height)
```

```

def setHeight( h : Int ) = new Rectangle(width, h)
}

class Square( val side : Int ) extends Rectangle(side, side)
{
  override def setWidth( s : Int ) = new Square(s)
  override def setHeight( s : Int ) = new Square(s)
}

```

LSP requires that each method of the subtype s must have contravariant input parameter(s) and a covariant output.

Contravariant means the variance is contrary to the direction of the inheritance, i.e. the type s_i , of each input parameter of each method of the subtype s , must be the same or a *supertype* of the type τ_i of the corresponding input parameter of the corresponding method of the supertype τ .

Covariance means the variance is in the same direction of the inheritance, i.e. the type s_o , of the output of each method of the subtype s , must be the same or a *subtype* of the type τ_o of the corresponding output of the corresponding method of the supertype τ .

This is because if the caller thinks it has a type τ , thinks it is calling a method of τ , then it supplies argument(s) of type τ_i and assigns the output to the type τ_o . When it is actually calling the corresponding method of s , then each τ_i input argument is assigned to a s_i input parameter, and the s_o output is assigned to the type τ_o . Thus if s_i were not contravariant w.r.t. to τ_i , then a subtype x_i — which would not be a subtype of s_i — could be assigned to τ_i .

Additionally, for languages (e.g. Scala or Ceylon) which have definition-site variance annotations on type polymorphism parameters (i.e. generics), the co- or contra-direction of the variance annotation for each type parameter of the type τ must be [opposite](#) or same

direction respectively to every input parameter or output (of every method of τ) that has the type of the type parameter.

Additionally, for each input parameter or output that has a function type, the variance direction required is reversed. This rule is applied recursively.

[Subtyping is appropriate](#) where the invariants can be enumerated.

There is much ongoing research on how to model invariants, so that they are enforced by the compiler.

[Typestate](#) (see page 3) declares and enforces state invariants orthogonal to type. Alternatively, invariants can be enforced by [converting assertions to types](#). For example, to assert that a file is open before closing it, then `File.open()` could return an `OpenFile` type, which contains a `close()` method that is not available in `File`. A [tic-tac-toe API](#) can be another example of employing typing to enforce invariants at compile-time. The type system may even be Turing-complete, e.g. [Scala](#). Dependently-typed languages and theorem provers formalize the models of higher-order typing.

Because of the need for semantics to [abstract over extension](#), I expect that employing typing to model invariants, i.e. unified higher-order denotational semantics, is superior to the Typestate. 'Extension' means the unbounded, permuted composition of uncoordinated, modular development. Because it seems to me to be the antithesis of unification and thus degrees-of-freedom, to have two mutually-dependent models (e.g. types and Typestate) for expressing the shared semantics, which can't be unified with each other for extensible composition. For example, [Expression Problem](#)-like extension was

unified in the subtyping, function overloading, and parametric typing domains.

My theoretical position is that for [knowledge to exist](#) (see section “Centralization is blind and unfit”), there will *never* be a general model that can enforce 100% coverage of all possible invariants in a Turing-complete computer language. For knowledge to exist, unexpected possibilities much exist, i.e. disorder and entropy must always be increasing. This is the entropic force. To prove all possible computations of a potential extension, is to compute a priori all possible extension.

This is why the Halting Theorem exists, i.e. it is undecidable whether every possible program in a Turing-complete programming language terminates. It can be proven that some specific program terminates (one which all possibilities have been defined and computed). But it is impossible to prove that all possible extension of that program terminates, unless the possibilities for extension of that program is not Turing complete (e.g. via dependent-typing). Since the fundamental requirement for Turing-completeness is [unbounded recursion](#), it is intuitive to understand how Gödel's incompleteness theorems and Russell's paradox apply to extension.

An interpretation of these theorems incorporates them in a generalized conceptual understanding of the entropic force:

- **Gödel's incompleteness theorems:** any formal theory, in which all arithmetic truths can be proved, is inconsistent.
- [Russell's paradox](#): every membership rule for a set that can contain a set, either enumerates the specific type of each member or contains itself. Thus sets either cannot be extended or they are unbounded recursion. For example, the set of everything that is not a teapot, includes itself, which includes itself,

which includes itself, etc.... Thus a rule is inconsistent if it (may contain a set and) does not enumerate the specific types (i.e. allows all unspecified types) and does not allow unbounded extension. This is the set of sets that are not members of themselves. This inability to be both consistent and completely enumerated over all possible extension, is Gödel's incompleteness theorems.

- **Liskov Substitution Principle:** generally it is an undecidable problem whether any set is the subset of another, i.e. inheritance is generally undecidable.
- **Linsky Referencing:** it is undecidable what the computation of something is, when it is described or perceived, i.e. perception (reality) has no absolute point of reference.
- **Coase's theorem:** there is no external reference point, thus any barrier to unbounded external possibilities will fail.
- **[Second law of thermodynamics](#):** the entire universe (a closed system, i.e. everything) trends to maximum disorder, i.e. maximum independent possibilities.

edited May 23 '17 at 12:26



Community ♦

1 1

answered Nov 26 '11 at 16:35



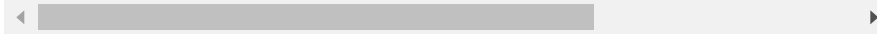
Shelby Moore III

5,199 1 24 27

- 16 @Shelbyby: You have mixed too many things. Things are not as confusing as you state them. Much of your theoretical assertions stand on flimsy grounds, like 'For knowledge to exist, unexpected possibilities much exist,' AND 'generally it is an undecidable problem whether any set is the subset of another, i.e. inheritance is generally undecidable'. You can start up a separate blog for each of these points. Anyways, your assertions and assumptions are highly

questionable. One must not use things which one is not aware of! – [aknon](#) Dec 27 '13 at 5:03

-
- 1 @aknon I [have a blog](#) that explains these matters in more depth. My TOE model of infinite spacetime is unbounded frequencies. It is not confusing to me that a recursive inductive function has a known start value with an infinite end bound, or a coinductive function has an unknown end value and a known start bound. Relativity is the problem once recursion is introduced. This is why [Turing complete is equivalent to unbounded recursion](#). – [Shelby Moore III](#) Mar 16 '14 at 8:38
-
- 4 @ShelbyMooreIII You are going in too many directions. This is not an answer. – [Soldalma](#) Dec 9 '16 at 14:48
-
- 1 @Soldalma it is an answer. Don't you see it in the Answer section. Yours is a comment because it is in the comment section. Psst. [Somebody is always wrong on the Internet](#), but diversity of opinion (and answers) is fundamentally what makes the Internet rich. – [Shelby Moore III](#) Dec 23 '16 at 12:54
-
- 1 Like your mixing with scala world! – [Ehsan M. Kermani](#) Jun 28 '17 at 18:38
-



20

The LSP is a rule about the contract of the classes: if a base class satisfies a contract, then by the LSP derived classes must also satisfy that contract.



In Pseudo-python

```
class Base:
    def Foo(self, arg):
        # *... do stuff*

class Derived(Base):
    def Foo(self, arg):
        # *... do stuff*
```

satisfies LSP if every time you call Foo on a Derived object, it gives exactly the same results as calling Foo on a

Base object, as long as arg is the same.

answered Nov 8 '08 at 17:53



Charlie Martin

92.9k 20 166 244

-
- 7 But ... if you always get the same behavior, then what is the point of having the derived class? – [Leonid](#) Jul 3 '12 at 17:14
-

You missed a point: it's the same *observed* behavior. You might, for example replace something with $O(n)$ performance with something functionally equivalent, but with $O(\lg n)$ performance. Or you might replace something that accesses data implemented with MySQL and replace it with an in-memory database. – [Charlie Martin](#) Jul 4 '12 at 18:06

@Charlie Martin, coding to an interface rather than an implementation - I dig that. This is not unique to OOP; functional languages such as Clojure promote that as well. Even in terms of Java or C#, I think that using an interface rather than using an abstract class plus class hierarchies would be natural for the examples that you provide. Python is not strongly typed and does not really have interfaces, at least not explicitly. My difficulty is that I have been doing OOP for several years without adhering to SOLID. Now that I came across it, it seems limiting and almost self-contradicting. – [Hamish Grubijan](#) Jul 6 '12 at 18:09

Well, you need to go back and check out Barbara's original paper. reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-156.ps It's not really stated in terms of interfaces, and it is a logical relation that holds (or doesn't) in any programming language that has some form of inheritance. – [Charlie Martin](#) Jul 7 '12 at 19:22

- 1 @HamishGrubijan I don't know who told you that Python is not strongly typed, but they were lying to you (and if you don't believe me, fire up a Python interpreter and try `2 + "2"`). Perhaps you confuse "strongly typed" with "statically typed"? – [asmeurer](#) Jan 20 '13 at 6:19
-



19

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

When I first read about LSP, I assumed that this was meant in a very strict sense, essentially equating it to interface implementation and type-safe casting. Which would mean that LSP is either ensured or not by the language itself. For example, in this strict sense, ThreeDBoard is certainly substitutable for Board, as far as the compiler is concerned.

After reading up more on the concept though I found that LSP is generally interpreted more broadly than that.

In short, what it means for client code to "know" that the object behind the pointer is of a derived type rather than the pointer type is not restricted to type-safety. Adherence to LSP is also testable through probing the objects actual behavior. That is, examining the impact of an object's state and method arguments on the results of the method calls, or the types of exceptions thrown from the object.

Going back to the example again, **in theory** the Board methods can be made to work just fine on ThreeDBoard. In practice however, it will be very difficult to prevent differences in behavior that client may not handle properly, without hobbling the functionality that ThreeDBoard is intended to add.

With this knowledge in hand, evaluating LSP adherence can be a great tool in determining when composition is the more appropriate mechanism for extending existing functionality, rather than inheritance.

edited Oct 7 '09 at 2:44



Peter Mortensen

14.1k 19 88 114

answered Sep 11 '08 at 16:04



Chris Ammerman

11k 7 34 41



There is a check list to determine whether or not you are violating Liskov.

18



- If you violate one of the following items -> you violate Liskov.
- If you dont violate any -> cant conclude anything.

Check list:

- **No new exceptions should be thrown in derived class:** If your base class threw `ArgumentNullException` then your sub classes were only allowed to throw exceptions of type `ArgumentNullException` or any exceptions derived from `ArgumentNullException`. Throwing `IndexOutOfRangeException` is a violation of Liskov.
- **Pre-conditions cannot be strengthened:** Assume your base class works with a member `int`. Now your sub-type requires that `int` to be positive. This is strengthened pre-conditions, and now any code that worked perfectly fine before with negative ints is broken.
- **Post-conditions cannot be weakened:** Assume your base class required all connections to database should be closed before the method returned. In your sub-class you overrode that method and leaved connection open for further reuse. You have weakened the post-conditions of that method.
- **Invariants must be preserved:** The most difficult and painful constraint to fulfill. Invariants are some time hidden in the base class and the only way to reveal them is to read the code of the base class. Basically

you have to be sure when you override a method anything unchangeable must remain unchanged after your overridden method executed. The best thing I can think of is to enforce this invariant constraints in the base class but that would not be easy.

- **History Constraint:** When overriding a method you are not allowed to modify an un-modifiable property in the base class. Take a look at these code and you can see Name is defined to be un-modifiable (private set) but SubType introduces new method that allows modifying it (through reflection):

```
public class SuperType
{
    public string Name { get; private set; }
    public SuperType(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
public class SubType : SuperType
{
    public void ChangeName(string newName)
    {
        var propertyType = base.GetType().GetProperty(
    }
}
```

There are 2 others items: **Contravariance of method arguments** and **Covariance of return types**. But it is not possible in C# (I'm a C# developer) so I don't care about them.

Reference:

- <http://www.ckode.dk/programming/solid-principles-part-3-liskovs-substitution-principle/>
- <https://softwareengineering.stackexchange.com/questions/187613/how-does-strengthening-of-pre-conditions-and-weakening-of-post-conditions-violat>

- <https://softwareengineering.stackexchange.com/questions/170189/how-to-verify-the-liskov-substitution-principle-in-an-inheritance-hierarchy>

edited Apr 12 '17 at 7:31



Community ♦

1 1

answered Aug 13 '16 at 14:26



Cù Đức Hiếu

3,610 3 20 34

I'm a C# developer also and I will tell your last statement isn't true as of Visual Studio 2010, with the .Net 4.0 framework. Covariance of return types allows for a more derived return type than what was defined by the interface. Example: IEnumerable<T> (T is covariant) IEnumerator<T> (T is covariant) IQueryable<T> (T is covariant) IGrouping<TKey, TElement> (TKey and TElement are covariant) IComparer<T> (T is contravariant) IEqualityComparer<T> (T is contravariant) IComparable<T> (T is contravariant) [msdn.microsoft.com/en-us/library/dd233059\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/dd233059(v=vs.100).aspx) – LCarter Sep 5 '17 at 3:48 ✎

- 1 Great and focused answer (although the original questions was about examples more than rules). – Mike Jun 12 '18 at 13:01



An important example of the *use* of LSP is in **software testing**.

17



If I have a class A that is an LSP-compliant subclass of B, then I can reuse the test suite of B to test A.

To fully test subclass A, I probably need to add a few more test cases, but at the minimum I can reuse all of superclass B's test cases.

A way to realize is this by building what McGregor calls a "Parallel hierarchy for testing": My `ATest` class will inherit from `BTest`. Some form of injection is then needed to ensure the test case works with objects of type A rather than of type B (a simple template method pattern will do).

Note that reusing the super-test suite for all subclass implementations is in fact a way to test that these subclass implementations are LSP-compliant. Thus, one can also argue that one *should* run the superclass test suite in the context of any subclass.

See also the answer to the Stackoverflow question "[Can I implement a series of reusable tests to test an interface's implementation?](#)"

edited May 23 '17 at 11:55



Community ♦

1 1

answered Mar 25 '13 at 21:26



avandeursen

6,754 2 32 45



15

I guess everyone kind of covered what LSP is technically:
You basically want to be able to abstract away from
subtype details and use supertypes safely.



So Liskov has 3 underlying rules:

1. Signature Rule : There should be a valid implementation of every operation of the supertype in the subtype syntactically. Something a compiler will be able to check for you. There is a little rule about throwing fewer exceptions and being at least as accessible as the supertype methods.

2. Methods Rule: The implementation of those operations is semantically sound.
 - Weaker Preconditions : The subtype functions should take at least what the supertype took as input, if not more.
 - Stronger Postconditions: They should produce a subset of the output the supertype methods produced.
3. Properties Rule : This goes beyond individual function calls.
 - Invariants : Things that are always true must remain true. Eg. a Set's size is never negative.
 - Evolutionary Properties : Usually something to do with immutability or the kind of states the object can be in. Or maybe the object only grows and never shrinks so the subtype methods shouldn't make it.

All these properties need to be preserved and the extra subtype functionality shouldn't violate supertype properties.

If these three things are taken care of , you have abstracted away from the underlying stuff and you are writing loosely coupled code.

Source: Program Development in Java - Barbara Liskov

answered Dec 18 '16 at 16:54



snagpaul

161 1 4



Long story short, let's leave rectangles rectangles and squares squares, practical example when extending a

parent class, you have to either **PRESERVE** the exact parent API or to **EXTEND** IT.

Let's say you have a **base** ItemsRepository.

```
class ItemsRepository
{
    /**
     * @return int Returns number of deleted rows
     */
    public function delete()
    {
        // perform a delete query
        $numberOfDeletedRows = 10;

        return $numberOfDeletedRows;
    }
}
```

And a sub class extending it:

```
class BadlyExtendedItemsRepository extends ItemsRepository
{
    /**
     * @return void Was suppose to return an INT like parent
     */
    public function delete()
    {
        // perform a delete query
        $numberOfDeletedRows = 10;

        // we broke the behaviour of the parent class
        return;
    }
}
```

Then you could have a **Client** working with the Base ItemsRepository API and relying on it.

```
/**
 * Class ItemsService is a client for public ItemsRepository
 * method).
 *
 * Technically, I am able to pass into a constructor a sub
```

```

* but if the sub-class won't abide the base class API, then
*/
class ItemService
{
    /**
     * @var ItemsRepository
     */
    private $itemsRepository;

    /**
     * @param ItemsRepository $itemsRepository
     */
    public function __construct(ItemsRepository $itemsRepository)
    {
        $this->itemsRepository = $itemsRepository;
    }

    /**
     * !!! Notice how this is suppose to return an int. My
the
     * ItemsRepository API in the constructor !!!
     *
     * @return int
     */
    public function delete()
    {
        return $this->itemsRepository->delete();
    }
}

```

The **LSP** is broken when **substituting** parent class with a **sub class breaks the API's contract**.

```

class ItemController
{
    /**
     * Valid delete action when using the base class.
     */
    public function validDeleteAction()
    {
        $itemService = new ItemService(new ItemsRepository);
        $numberOfDeletedItems = $itemService->delete();

        // $numberOfDeletedItems is an INT :)
    }

    /**

```

```

* Invalid delete action when using a subclass.
*/
public function brokenDeleteAction()
{
    $itemsService = new ItemsService(new BadlyExtended:
    $numberOfDeletedItems = $itemsService->delete();

    // $numberOfDeletedItems is a NULL :(
}
}

```

You can learn more about writing maintainable software in my course: <https://www.udemy.com/enterprise-php/>

edited Dec 18 '18 at 3:27



Pang

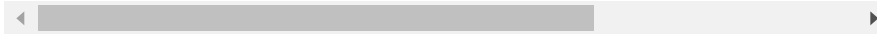
7,052 16 67 105

answered Oct 28 '17 at 23:10



BlocksByLukas

2,498 8 48 64



This formulation of the LSP is way too strong:

9



If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Which basically means that S is another, completely encapsulated implementation of the exact same thing as T. And I could be bold and decide that performance is part of the behavior of P...

So, basically, any use of late-binding violates the LSP. It's the whole point of OO to to obtain a different behavior when we substitute an object of one kind for one of another kind!

The formulation cited [by wikipedia](#) is better since the property depends on the context and does not necessarily include the whole behavior of the program.

answered Apr 3 '09 at 0:11



[Damien Pollet](#)

4,873 3 22 25

-
- 2 Erm, that formulation is Barbara Liskov's own. Barbara Liskov, "Data Abstraction and Hierarchy," SIGPLAN Notices, 23,5 (May, 1988). It is not "way too strong", it is "exactly right", and it does not have the implication you think it has. It is strong, but has just the right amount of strength. – [DrPizza](#) Nov 30 '09 at 7:08

Then, there are very few subtypes in the real life :) – [Damien Pollet](#) Dec 8 '09 at 22:26

-
- 3 "Behavior is unchanged" does not mean that a subtype will give you the exact same concrete result value(s). It means that the subtype's behavior matches what is expected in the base type. Example: base type Shape could have a draw() method and stipulate that this method should render the shape. Two subtypes of Shape (e.g. Square and Circle) would both implement the draw() method and the results would look different. But as long as the behavior (rendering the shape) matched the specified behavior of Shape, then Square and Circle would be subtypes of Shape in accordance with the LSP. – [SteveT](#) Oct 11 '12 at 19:38

▲ In a very simple sentence, we can say:

7

▼ The child class must not violate its base class characteristics. It must be capable with it. We can say it's same as subtyping.

edited Aug 16 '17 at 9:09



[dlmeetei](#)

4,301 1 17 29

answered Aug 16 '17 at 8:40



Alireza Rahmani Khalili

673 2 12 19



7



Some addendum:

I wonder why didn't anybody write about the Invariant , preconditions and post conditions of the base class that must be obeyed by the derived classes. For a derived class D to be completely substitutable by the Base class B, class D must obey certain conditions:

- In-variants of base class must be preserved by the derived class
- Pre-conditions of the base class must not be strengthened by the derived class
- Post-conditions of the base class must not be weakened by the derived class.

So the derived must be aware of the above three conditions imposed by the base class. Hence, the rules of subtyping are pre-decided. Which means, 'IS A' relationship shall be obeyed only when certain rules are obeyed by the subtype. These rules, in the form of invariants, preconditions and postcondition, should be decided by a formal '[design contract](#)'.

Further discussions on this available at my blog: [Liskov Substitution principle](#)

answered Dec 27 '13 at 5:20



aknon

863 1 14 22



I see rectangles and squares in every answer, and how to violate the LSP.

7

I'd like to show how the LSP can be conformed to with a real-world example :

```
<?php

interface Database
{
    public function selectQuery(string $sql): array;
}

class SQLiteDatabase implements Database
{
    public function selectQuery(string $sql): array
    {
        // sqlite specific code

        return $result;
    }
}

class MySQLDatabase implements Database
{
    public function selectQuery(string $sql): array
    {
        // mysql specific code

        return $result;
    }
}
```

This design conforms to the LSP because the behaviour remains unchanged regardless of the implementation we choose to use.

And yes, you can violate LSP in this configuration doing one simple change like so :

```
<?php

interface Database
{
    public function selectQuery(string $sql): array;
}

class SQLiteDatabase implements Database
```

```

{
    public function selectQuery(string $sql): array
    {
        // sqlite specific code

        return $result;
    }
}

class MySQLDatabase implements Database
{
    public function selectQuery(string $sql): array
    {
        // mysql specific code

        return ['result' => $result]; // This violates LSP
    }
}

```

Now the subtypes cannot be used the same way since they don't produce the same result anymore.

edited May 4 at 8:15

answered Feb 18 '18 at 19:07



Steve Chamillard

1,317 10 24

- 4 The example does not violate LSP only as long as we constrain the semantics of `Database::selectQuery` to support just the subset of SQL supported by **all** DB engines. That's hardly practical... That said, the example is still easier to grasp than most others used here. – [Palec](#) Feb 25 '18 at 13:02

I found this answer the easiest to grasp out of the rest. – [Malcolm Salvador](#) Feb 11 at 17:14



Liskov's Substitution Principle(LSP)

5

All the time we design a program module and we create some class hierarchies. Then we extend some classes creating some derived classes.

We must make sure that the new derived classes just extend without replacing the functionality of old classes. Otherwise, the new classes can produce undesired effects when they are used in existing program modules.

Liskov's Substitution Principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.

Example:

Below is the classic example for which the Liskov's Substitution Principle is violated. In the example, 2 classes are used: Rectangle and Square. Let's assume that the Rectangle object is used somewhere in the application. We extend the application and add the Square class. The square class is returned by a factory pattern, based on some conditions and we don't know the exact what type of object will be returned. But we know it's a Rectangle. We get the rectangle object, set the width to 5 and height to 10 and get the area. For a rectangle with width 5 and height 10, the area should be 50. Instead, the result will be 100

```
// Violation of Liskov's Substitution Principle
class Rectangle {
    protected int m_width;
    protected int m_height;

    public void setWidth(int width) {
        m_width = width;
    }

    public void setHeight(int height) {
        m_height = height;
    }
}
```

```
    }

    public int getWidth() {
        return m_width;
    }

    public int getHeight() {
        return m_height;
    }

    public int getArea() {
        return m_width * m_height;
    }
}

class Square extends Rectangle {
    public void setWidth(int width) {
        m_width = width;
        m_height = width;
    }

    public void setHeight(int height) {
        m_width = height;
        m_height = height;
    }
}

class LspTest {
    private static Rectangle getNewRectangle() {
        // it can be an object returned by some factory ..
        return new Square();
    }

    public static void main(String args[]) {
        Rectangle r = LspTest.getNewRectangle();

        r.setWidth(5);
        r.setHeight(10);
        // user knows that r it's a rectangle.
        // It assumes that he's able to set the width and l
        // class

        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100 :
    }
}
```

Conclusion:

This principle is just an extension of the Open Close Principle and it means that we must make sure that new derived classes are extending the base classes without changing their behavior.

See also: [Open Close Principle](#)

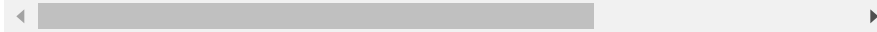
Some similar concepts for better structure: [Convention over configuration](#)

answered May 21 '18 at 9:01



[GauRang Omar](#)

499 5 14



4

A square is a rectangle where the width equals the height. If the square sets two different sizes for the width and height it violates the square invariant. This is worked around by introducing side effects. But if the rectangle had a `setSize(height, width)` with precondition $0 < \text{height}$ and $0 < \text{width}$. The derived subtype method requires `height == width`; a stronger precondition (and that violates lsp). This shows that though square is a rectangle it is not a valid subtype because the precondition is strengthened. The work around (in general a bad thing) cause a side effect and this weakens the post condition (which violates lsp). `setWidth` on the base has post condition $0 < \text{width}$. The derived weakens it with `height == width`.

Therefore a resizable square is not a resizable rectangle.

answered Jul 27 '15 at 21:56



[Wouter](#)

838 8 17



Would implementing ThreeDBoard in terms of an array of Board be that useful?

4



Perhaps you may want to treat slices of ThreeDBoard in various planes as a Board. In that case you may want to abstract out an interface (or abstract class) for Board to allow for multiple implementations.

In terms of external interface, you might want to factor out a Board interface for both TwoDBoard and ThreeDBoard (although none of the above methods fit).

answered Sep 11 '08 at 15:28



[Tom Hawtin - tackline](#)

128k 29 185 275

1 I think the example is simply to demonstrate that inheriting from board does not make sense with in the context of ThreeDBoard and all of the method signatures are meaningless with a Z axis. – [NotMyself](#) Sep 11 '08 at 15:32



Let's say we use a rectangle in our code

3



```
r = new Rectangle();  
// ...  
r.setDimensions(1,2);  
r.fill(colors.red());  
canvas.draw(r);
```

In our geometry class we learned that a square is a special type of rectangle because its width is the same length as its height. Let's make a `Square` class as well based on this info:

```
class Square extends Rectangle {  
    setDimensions(width, height){
```

```

        assert(width == height);
        super.setDimensions(width, height);
    }
}

```

If we replace the `Rectangle` with `Square` in our first code, then it will break:

```

r = new Square();
// ...
r.setDimensions(1,2); // assertion width == height failed
r.fill(colors.red());
canvas.draw(r);

```

This is because the `Square` has a new precondition we did not have in the `Rectangle` class: `width == height`. According to LSP the `Rectangle` instances should be substitutable with `Rectangle` subclass instances. This is because these instances pass the type check for `Rectangle` instances and so they will cause unexpected errors in your code.

This was an example for the *"preconditions cannot be strengthened in a subtype"* part in the [wiki article](#). So to sum up, violating LSP will probably cause errors in your code at some point.

answered Oct 9 '17 at 4:26



inf3rno

15.1k 7 78 133



Let's illustrate in Java:

3

```

class TrasportationDevice
{
    String name;
    String getName() { ... }
    void setName(String n) { ... }
}

```

```
double speed;
double getSpeed() { ... }
void setSpeed(double d) { ... }

Engine engine;
Engine getEngine() { ... }
void setEngine(Engine e) { ... }

void startEngine() { ... }
}

class Car extends TransportationDevice
{
    @Override
    void startEngine() { ... }
}
```

There is no problem here, right? A car is definitely a transportation device, and here we can see that it overrides the `startEngine()` method of its superclass.

Let's add another transportation device:

```
class Bicycle extends TransportationDevice
{
    @Override
    void startEngine() /*problem!*/
}
```

Everything isn't going as planned now! Yes, a bicycle is a transportation device, however, it does not have an engine and hence, the method `startEngine()` cannot be implemented.

These are the kinds of problems that violation of Liskov Substitution Principle leads to, and they can most usually be recognized by a method that does nothing, or even can't be implemented.

The solution to these problems is a correct inheritance hierarchy, and in our case we would solve the problem by

differentiating classes of transportation devices with and without engines. Even though a bicycle is a transportation device, it doesn't have an engine. In this example our definition of transportation device is wrong. It should not have an engine.

We can refactor our TransportationDevice class as follows:

```
class TransportationDevice
{
    String name;
    String getName() { ... }
    void setName(String n) { ... }

    double speed;
    double getSpeed() { ... }
    void setSpeed(double d) { ... }
}
```

Now we can extend TransportationDevice for non-motorized devices.

```
class DevicesWithoutEngines extends TransportationDevice
{
    void startMoving() { ... }
}
```

And extend TransportationDevice for motorized devices. Here is is more appropriate to add the Engine object.

```
class DevicesWithEngines extends TransportationDevice
{
    Engine engine;
    Engine getEngine() { ... }
    void setEngine(Engine e) { ... }

    void startEngine() { ... }
}
```

Thus our Car class becomes more specialized, while adhering to the Liskov Substitution Principle.

```
class Car extends DevicesWithEngines
{
    @Override
    void startEngine() { ... }
}
```

And our Bicycle class is also in compliance with the Liskov Substitution Principle.

```
class Bicycle extends DevicesWithoutEngines
{
    @Override
    void startMoving() { ... }
}
```

answered Feb 10 at 10:56



Khaled Qasem

407 1 16



2

▲ LISKOV SUBSTITUTION PRINCIPLE (From Mark Seemann book) states that we should be able to replace one implementation of an interface with another without breaking either client or implementation. It's this principle that enables to address requirements that occur in the future, even if we can't foresee them today.

▼

If we unplug the computer from the wall (Implementation), neither the wall outlet (Interface) nor the computer (Client) breaks down (in fact, if it's a laptop computer, it can even run on its batteries for a period of time). With software, however, a client often expects a service to be available. If the service was removed, we get a `NullPointerException`. To deal with this type of situation, we can create an implementation of an interface that does "nothing." This is a design pattern known as Null Object,[4] and it corresponds roughly to unplugging the computer

from the wall. Because we're using loose coupling, we can replace a real implementation with something that does nothing without causing trouble.

answered Aug 12 '16 at 17:16



Raghu Reddy Muttana

61 4

2

The clearest explanation for LSP I found so far has been "The Liskov Substitution Principle says that the object of a derived class should be able to replace an object of the base class without bringing any errors in the system or modifying the behavior of the base class" from [here](#). The article gives code example for violating LSP and fixing it.

answered May 3 '16 at 19:34



Prasa

31 3

1 Please provide the examples of code on stackoverflow. – [sebenalern](#) May 3 '16 at 19:51

2

I encourage you to read the article: [Violating Liskov Substitution Principle \(LSP\)](#).

You can find there an explanation what is the Liskov Substitution Principle, general clues helping you to guess if you have already violated it and an example of approach that will help you to make your class hierarchy be more safe.

answered Sep 9 '13 at 7:32



Ryszard Dżegan

17.5k 5 26 47

2

Liskov's Substitution Principle states that *if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.*

Intent - Derived types must be completely substitute able for their base types.

Example - Co-variant return types in java.

answered Sep 23 '17 at 19:26



Ishan Aggarwal

51 3

0

LSP says that "Objects should be replaceable by their subtypes". On the other hand, this principle points to

Child classes should never break the parent class's type definitions.

and the following example helps to have a better understanding of LSP.

Without LSP:

```
public interface CustomerLayout{

    public void render();

}

public FreeCustomer implements CustomerLayout {
    ...
    @Override
    public void render(){
        //code
    }
}
```

```
public PremiumCustomer implements CustomerLayout{
    ...
    @Override
    public void render(){
        if(!hasSeenAd)
            return; //it isn't rendered in this case
        //code
    }
}

public void renderView(CustomerLayout layout){
    layout.render();
}
```

Fixing by LSP:

```
public interface CustomerLayout{
    public void render();
}

public FreeCustomer implements CustomerLayout {
    ...
    @Override
    public void render(){
        //code
    }
}

public PremiumCustomer implements CustomerLayout{
    ...
    @Override
    public void render(){
        if(!hasSeenAd)
            showAd();//it has a specific behavior based on
        //code
    }
}

public void renderView(CustomerLayout layout){
    layout.render();
}
```

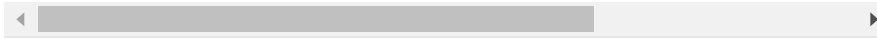
edited Apr 7 at 10:46

answered Apr 6 at 12:55



Zahra.HY

1,028 10 23



Let me try, consider an interface:

0

```
interface Planet{  
}
```



This is implemented by class:

```
class Earth implements Planet {  
    public $radius;  
    public function construct($radius) {  
        $this->radius = $radius;  
    }  
}
```

You will use Earth as:

```
$planet = new Earth(6371);  
$calc = new SurfaceAreaCalculator($planet);  
$calc->output();
```

Now consider one more class which extends Earth:

```
class LiveablePlanet extends Earth{  
    public function color(){  
    }  
}
```

Now according to LSP, you should be able to use LiveablePlanet in place of Earth and it should not break

your system. Like:

```
$planet = new LiveablePlanet(6371); // Earlier we were us:
$calc = new SurfaceAreaCalculator($planet);
$calc->output();
```

Examples taken from [here](#)

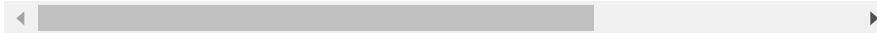
answered Apr 26 at 6:59



prady00

448 1 5 15

Poor example. Earth is an instance of planet, why would it get derived from it? – zar May 3 at 15:46



Here is an excerpt from [this post](#) that clarifies things nicely:

-1



[..] in order to comprehend some principles, it's important to realize when it's been violated. This is what I will do now.

What does the violation of this principle mean? It implies that an object doesn't fulfill the contract imposed by an abstraction expressed with an interface. In other words, it means that you identified your abstractions wrong.

Consider the following example:

```
interface Account
{
    /**
     * Withdraw $money amount from this account.
     *
     * @param Money $money
     * @return mixed
     */
    public function withdraw(Money $money);
}
```

```

class DefaultAccount implements Account
{
    private $balance;
    public function withdraw(Money $money)
    {
        if (!$this->enoughMoney($money)) {
            return;
        }
        $this->balance->subtract($money);
    }
}

```

Is this a violation of LSP? Yes. This is because the account's contract tells us that an account would be withdrawn, but this is not always the case. So, what should I do in order to fix it? I just modify the contract:

```

interface Account
{
    /**
     * Withdraw $money amount from this account if its bal:
     * Otherwise do nothing.
     *
     * @param Money $money
     * @return mixed
     */
    public function withdraw(Money $money);
}

```

Voilà, now the contract is satisfied.

This subtle violation often imposes a client with the ability to tell the difference between concrete objects employed. For example, given the first Account's contract, it could look like the following:

```

class Client
{
    public function go(Account $account, Money $money)
    {
        if ($account instanceof DefaultAccount && !$account
            return;
        }
        $account->withdraw($money);
    }
}

```

```
}  
}
```

And, this automatically violates the open-closed principle [that is, for money withdrawal requirement. Because you never know what happens if an object violating the contract doesn't have enough money. Probably it just returns nothing, probably an exception will be thrown. So you have to check if it `hasEnoughMoney()` -- which is not part of an interface. So this forced concrete-class-dependent check is an OCP violation].

This point also addresses a misconception that I encounter quite often about LSP violation. It says the "if a parent's behavior changed in a child, then, it violates LSP." However, it doesn't — as long as a child doesn't violate its parent's contract.

answered Aug 12 '18 at 17:10



[zapadlo](#)

1,598 1 30 57



protected by [revo](#) Jan 9 '18 at 12:27

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 [reputation](#) on this site (the [association bonus does not count](#)).

Would you like to answer one of these [unanswered questions](#) instead?