Does Liskov Substitution Principle also apply to classes implementing interfaces?

Ask Question

3

1

1) Does LSP also apply to interfaces, meaning that we should be able to use a class implementing a specific interface and still get the expected behavior?

2) If that is indeed the case, then why is programming to an interface considered a good thing (BTW- I know that programming to an interface increases loose coupling), if one of the main reasons against using inheritance is due to risk of not complying to LSP? Perhaps because:

a) benefits of loose coupling outweight the risks of not complying to LSP

b) compared to inheritance, chances that a class (implementing an interface) will not adher to LSP are much smaller

thank you

design-patterns liskov-substitution-principle

asked Sep 3 '12 at 18:06

user1483278

409 1 5 15

Home

PUBLIC

Stack Overflow

Tags

Users

3 Answers

6

> Does LSP also apply to interfaces, meaning that we should be able to use a class implementing a specific interface and still get the expected behavior?

LSP applies to the contract. The contract may be a class or an interface.

> If that is indeed the case, then why is programming to an interface considered a good thing (BTW- I know that programming to an interface increases loose coupling), if one of the main reasons against using inheritance is due to risk of not complying to LSP? Perhaps because:

It's not about an interface or a class. It's about a violation of the contract. Let's say that you have a `Break()` method in a `IVehicle` (or `VehicleBase`). Anyone calling it would expect the vehicle to break. Imagine the surprise if one of the implementations didn't break. That's what LSP is all about.

> a) benefits of loose coupling outweight the risks of not complying to LSP

ehh?

> b) compared to inheritance, chances that a class (implementing an interface) will not adher to LSP are much smaller

ehh?

You might want to read my SOLID article to understand the principle better: http://blog.gauffin.org/2012/05/solid-principles-with-real-

world-examples/

Update

> To elaborate - with inheritance virtual methods may consume private members, to which subclasses overriding these virtual methods don't have access to.

Yes. That's good. members (fields) should always be protected (= declared as private). Only the class that defined them really know what their values should be.

> Also, derived class inherits the context from parent and as such can be broken by future changes to parent class etc.

That's a violation of Open/closed principle. i.e. the class contract is changed by changing the behavior. Classes should be extended and not modified. Sure, it's not possible all the time, but changes should not make the class behave differently (other than bugfixes).

> Thus I feel it's more difficult to make subclass honour the contract than it is to make class implementing an interface honour it

There is a common reason to why extension through inheritance is hard. And that's because the relationship isn't a true `is-a` relationship, but that the developer just want to take advantage of the base class functionality.

That's wrong. Better to use composition then.

edited Sep 3 '12 at 19:13

answered Sep 3 '12 at 18:11

jgauffin

82k 36 196 320

1 "Anyone calling it would expect the vehicle to break. Imagine the surprise if one of the implementations didn't break. That's what LSP is all about." I understand that. – user1483278 Sep 3 '12 at 18:59

"ehh?" To elaborate - with inheritance virtual methods may consume private members, to which subclasses overriding these virtual methods don't have access to. Also, derived class inherits the context from parent and as such can be broken by future changes to parent class etc.Thus I feel it's more difficult to make subclass honour the contract than it is to make class implementing an interface honour it – user1483278 Sep 3 '12 at 19:00

1 @user1483278: Read my update. – jgauffin Sep 3 '12 at 19:13

much thanx for your help – user1483278 Sep 4 '12 at 18:53

"Anyone calling it would expect the vehicle to break. Imagine the surprise if one of the implementations didn't break. That's what LSP is all about." Then would you say that the null object pattern is an anti-pattern? ;) `IBreakStrategy`, `DiskBreakStrategy`, `NullBreakStrategy` where the null one is not breaking. – plalx Nov 7 '15 at 14:27

1

I'm comparing/contrasting my code practice with LSP at the moment. I think it must be all to do with expectation and deciding how to define what should be expected. I'm not convinced that substitution always means the same thing. For instance, if I defined

```
interface ICalculation
{
    double Calculate(double A, double B);
}
```

with the intention of defining `class Add : ICalculation`, `class Subtract : ICalculation`, `class Multiply : ICalculation` and `class Divide : ICalculation`, I believe that those classes should perform the respective `Calculation(A, B)`, however, I do not believe they should all pass the same unit tests. I like to use interfaces for extensibility, where each extension has a different function. In the case of a car braking, I would give classes the interface IBrakable and do this:

```
var myBrakableCar = MyCar as IBrakable;
if(myBrakableCar != null)
{
    myBrakableCar.Brake();
}
```

I believe that classes should be predictable in their use, but also useful. Useful comes top.

I'm just going to create a new concept - "Respective Substition": it does what it says on the tin.

edited Jan 27 '16 at 9:39

Robert Jørgensgaard Engdahl

2,488 16 25

answered Jan 25 '16 at 20:47

Phil

696 1 9 14

1 The `ICalculation` classes are never meant to substitute each other and of course they shouldn't pass the same unit tests either. However, all the unit tests that ICalculation pass, must also be passed by all its implementations. – Robert Jørgensgaard Engdahl Jan 27 '16 at 10:30

Thank you Robert. – Phil Jan 28 '16 at 10:15

0

Ad 1): Yes. However, it is hard to make interfaces enforce the contracts, and they really should.

Ad 2): Programming against an interface is a trade-off. As you point out interfaces encourages violation of LSP for all interfaces with a non-trivial contract.

I believe these are the answers to your questions, or at least my recognition that tese are unanswered questions that challenge the encouraged use of interfaces.

I use interfaces all the time (in C#) because I like doing TDD. I then just hope that my mocks and corresponding implementations don't violate LSP, because when they do my test suite is no longer sound (it claims something to work, but it doesn't).

Occasionally, I'll make an abstract base class instead of an interface. The abstract base class simply enforces the contract, and delegates to virtual protected methods that are supposed to define the actual implementation or being mocked during unit testing. This has proven to be useful in some applications, where errors in the unit tests themselves are discovered.

But then we run into the missing support for multiple inheritance in C#, and we are stuck with the interfaces anyway (or simply more classes, you can hide behind the SRP if you choose this approach)

edited Jan 27 '16 at 10:48

answered Jan 27 '16 at 7:37

Robert Jørgensgaard Engdahl
2,488 16 25