Liskov Substitution Principle C

Liskov substitution principle

The Liskov substitution principle (LSP) is a particular definition of a subtyping relation, called strong behavioral subtyping, that was initially introduced

The Liskov substitution principle (LSP) is a particular definition of a subtyping relation, called strong behavioral subtyping, that was initially introduced by Barbara Liskov in a 1987 conference keynote address titled Data abstraction and hierarchy. It is based on the concept of "substitutability" – a principle in object-oriented programming stating that an object (such as a class) may be replaced by a sub-object (such as a class that extends the first class) without breaking the program. It is a semantic rather than merely syntactic relation, because it intends to guarantee semantic interoperability of types in a hierarchy, object types in particular. Barbara Liskov and Jeannette Wing described the principle succinctly in a 1994 paper as follows:

Subtype Requirement: Let ?

?

(

x

)

{\displaystyle \phi (x)}

? be a property provable about objects ?

x

{\displaystyle x}

? of type T. Then ?

?

(

y

)

{\displaystyle \phi (y)}

? should be true for objects ?

y

{\displaystyle y}

? of type S where S is a subtype of T.

Symbolically:

S

?

T

?

(

?

x

:

T

.

?

(

x

)

?

?

y

:

S

.

?

(

y

)

)

{\displaystyle S\leq T\to (\forall x{:}T.\phi (x)\to \forall y{:}S.\phi (y))}

That is, if S subtypes T, what holds for T-objects holds for S-objects.

In the same paper, Liskov and Wing detailed their notion of behavioral subtyping in an extension of Hoare logic, which bears a certain resemblance to Bertrand Meyer's design by contract in that it considers the interaction of subtyping with preconditions, postconditions and invariants.

Barbara Liskov

abstract data types and the accompanying principle of data abstraction, along with the Liskov substitution principle, which applies these ideas to object-oriented

Barbara Liskov (born November 7, 1939, as Barbara Jane Huberman) is an American computer scientist who has made pioneering contributions to programming languages and distributed computing. Her notable work includes the introduction of abstract data types and the accompanying principle of data abstraction, along with the Liskov substitution principle, which applies these ideas to object-oriented programming, subtyping, and inheritance. Her work was recognized with the 2008 Turing Award, the highest distinction in computer science.

Liskov is one of the earliest women to have been granted a doctorate in computer science in the United States, and the second woman to receive the Turing award. She is currently an Institute Professor and Ford Professor of Engineering at the Massachusetts Institute of Technology.

SOLID

ISBN 978-0135974445. "Open/Closed Principle" (PDF). objectmentor.com. Archived from the original on 5 September 2015. "Liskov Substitution Principle" (PDF). objectmentor

In software programming, SOLID is a mnemonic acronym for five design principles intended to make object-oriented designs more understandable, flexible, and maintainable. Although the SOLID principles apply to any object-oriented design, they can also form a core philosophy for methodologies such as agile development or adaptive software development.

Software engineer and instructor Robert C. Martin introduced the basic principles of SOLID design in his 2000 paper Design Principles and Design Patterns about software rot. The SOLID acronym was coined around 2004 by Michael Feathers.

Circle–ellipse problem

programming (OOP). By definition, this problem is a violation of the Liskov substitution principle, one of the SOLID principles. The problem concerns which subtyping

The circle–ellipse problem in software development (sometimes called the square–rectangle problem) illustrates several pitfalls which can arise when using subtype polymorphism in object modelling. The issues are most commonly encountered when using object-oriented programming (OOP). By definition, this problem is a violation of the Liskov substitution principle, one of the SOLID principles.

The problem concerns which subtyping or inheritance relationship should exist between classes which represent circles and ellipses (or, similarly, squares and rectangles). More generally, the problem illustrates the difficulties which can occur when a base class contains methods which mutate an object in a manner which may invalidate a (stronger) invariant found in a derived class, causing the Liskov substitution principle to be violated.

The existence of the circle–ellipse problem is sometimes used to criticize object-oriented programming. It may also imply that hierarchical taxonomies are difficult to make universal, implying that situational classification systems may be more practical.

Single-responsibility principle

that requires a change in the module. Robert C. Martin, the originator of the term, expresses the principle as, "A class should have only one reason to

The single-responsibility principle (SRP) is a computer programming principle that states that "A module should be responsible to one, and only one, actor." The term actor refers to a group (consisting of one or more stakeholders or users) that requires a change in the module.

Robert C. Martin, the originator of the term, expresses the principle as, "A class should have only one reason to change". Because of confusion around the word "reason", he later clarified his meaning in a blog post titled "The Single Responsibility Principle", in which he mentioned Separation of Concerns and stated that "Another wording for the Single Responsibility Principle is: Gather together the things that change for the same reasons. Separate those things that change for different reasons." In some of his talks, he also argues that the principle is, in particular, about roles or actors. For example, while they might be the same person, the role of an accountant is different from a database administrator. Hence, each module should be responsible for each role.

Interface segregation principle

February 2021 "This principle was first defined by Robert C. Martin". Robert C. Martin,The Interface Segregation Principle, C++ Report, June 1996 Principles

In the field of software engineering, the interface segregation principle (ISP) states that no code should be forced to depend on methods it does not use. ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces. ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is one of the five SOLID principles of object-oriented design, similar to the High Cohesion Principle of GRASP. Beyond object-oriented design, ISP is also a key principle in the design of distributed systems in general and one of the six IDEALS principles for microservice design.

Open–closed principle

In object-oriented programming, the open–closed principle (OCP) states "software entities (classes, modules, functions, etc.) should be open for extension

In object-oriented programming, the open–closed principle (OCP) states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification";

that is, such an entity can allow its behaviour to be extended without modifying its source code.

The name open–closed principle has been used in two ways. Both ways use generalizations (for instance, inheritance or delegate functions) to resolve the apparent dilemma, but the goals, techniques, and results are different.

The open–closed principle is one of the five SOLID principles of object-oriented design.

Composition over inheritance

programmers do with inheritance in Java" Delegation pattern Liskov substitution principle Object-oriented design Object composition Role-oriented programming

Composition over inheritance (or composite reuse principle) in object-oriented programming (OOP) is the principle that classes should favor polymorphic behavior and code reuse by their composition (by containing

instances of other classes that implement the desired functionality) over inheritance from a base or parent class. Ideally all reuse can be achieved by assembling existing components, but in practice inheritance is often needed to make new ones. Therefore inheritance and object composition typically work hand-in-hand, as discussed in the book Design Patterns (1994).

Object-oriented programming

concept is known as behavioral subtyping, more specifically the Liskov substitution principle. However, this is often not true, especially in programming

Object-oriented programming (OOP) is a programming paradigm based on the object – a software entity that encapsulates data and function(s). An OOP computer program consists of objects that interact with one another. A programming language that provides OOP features is classified as an OOP language but as the set of features that contribute to OOP is contended, classifying a language as OOP and the degree to which it supports or is OOP, are debatable. As paradigms are not mutually exclusive, a language can be multi-paradigm; can be categorized as more than only OOP.

Sometimes, objects represent real-world things and processes in digital form. For example, a graphics program may have objects such as circle, square, and menu. An online shopping system might have objects such as shopping cart, customer, and product. Niklaus Wirth said, "This paradigm [OOP] closely reflects the structure of systems in the real world and is therefore well suited to model complex systems with complex behavior".

However, more often, objects represent abstract entities, like an open file or a unit converter. Not everyone agrees that OOP makes it easy to copy the real world exactly or that doing so is even necessary. Bob Martin suggests that because classes are software, their relationships don't match the real-world relationships they represent. Bertrand Meyer argues that a program is not a model of the world but a model of some part of the world; "Reality is a cousin twice removed". Steve Yegge noted that natural languages lack the OOP approach of naming a thing (object) before an action (method), as opposed to functional programming which does the reverse. This can make an OOP solution more complex than one written via procedural programming.

Notable languages with OOP support include Ada, ActionScript, C++, Common Lisp, C#, Dart, Eiffel, Fortran 2003, Haxe, Java, JavaScript, Kotlin, Logo, MATLAB, Objective-C, Object Pascal, Perl, PHP, Python, R, Raku, Ruby, Scala, SIMSCRIPT, Simula, Smalltalk, Swift, Vala and Visual Basic (.NET).

Covariant return type

types is usually one which allows substitution of the one type with the other, following the Liskov substitution principle. This usually implies that the

In object-oriented programming, a covariant return type of a method is one that can be replaced by a "narrower" (derived) type when the method is overridden in a subclass. A notable language in which this is a fairly common paradigm is C++.

C# supports return type covariance as of version 9.0. Covariant return types have been (partially) allowed in the Java language since the release of JDK5.0, so the following example wouldn't compile on a previous release:

More specifically, covariant (wide to narrower) or contravariant (narrow to wider) return type refers to a situation where the return type of the overriding method is changed to a type related to (but different from) the return type of the original overridden method. The relationship between the two covariant return types is usually one which allows substitution of the one type with the other, following the Liskov substitution principle. This usually implies that the return types of the overriding methods will be subtypes of the return type of the overridden method. The above example specifically illustrates such a case. If substitution is not

allowed, the return type is invariant and causes a compile error.

Another example of covariance with the help of built in Object and String class of Java: