The reading is also available as a PDF here:

Liskov.pdf

Classes

In object-oriented programming, developers are able to define their data structures and use them to encapsulate a set of attributes and methods. These data structures are known as "classes". They allow developers to create abstract representations of real world objects or theoretical concepts within a software system.

In general, objects and ideas do not exist in isolation, meaning that they are related to, composed of, part of, or associated with something else. So how are these relationships realized in object-oriented programming? By introducing object-oriented designs such as inheritance.

Class Inheritance

Inheritance is a concept that allows a class to take on and become "endowed" with the characteristics and behaviors of another class. In this relationship, the inherited class is known as the “base class”, and the inheriting class is the “subclass”. Inheritance is one of the pillars of object-oriented programming because it allows subclasses to obtain the same attributes and methods as the base class. In addition, inheriting classes are able to add their own characteristics and behaviors, allowing them to become more "specialized".

Since classes are user-defined data structures, it is implied that the type of a class is also user-defined. Therefore, inheritance allows the subclass to become polymorphic, because inheritance lets the subclass become a subtype of the base class.

Subtyping also allows subclasses to be used as substitutes for their base class. Substitution states that:

Any class, S, can be used to replace a class, B, if and only if, S is a subtype of B.

Simply put, any subclass can stand in for its base class. Because of inheritance, a subclass is expected to have the same characteristics and behave in the same way.

While this is easy to understand, it presents some difficulties when it comes to designing an object-oriented system. Developers need to be able to determine when it is appropriate to use inheritance. Applying inheritance incorrectly can cause classes to behave in an undesirable manner.

Liskov Substitution Principle

The base class is the more generalized class, and therefore, its attributes and behaviors should reflect it. The names given to the attributes and methods, as well as the implementation of each method must be broad enough that all subclasses can use them.

If inheritance is not used correctly, it can lead to a violation of the “Liskov Substitution Principle”. This principle uses substitution to determine whether or not inheritance has been properly used. The Liskov Substitution Principle states that:

If a class, S, is a subtype of a class, B, then S can be used to replace all instances of B without changing the behaviors of a program.

The logic behind this is straightforward. If S is a subtype of B, then it can be expected that S will have the same behaviours as B. Therefore, S can be used in place of B and it would not affect the software. This means that inheritance can be tested by applying substitution.

Inheritance Guidelines

There are a number of constraints that the Liskov Substitution Principle places on subclasses in order to enforce proper use of inheritance:

1. The condition used to determine if a base class should or should not invoke a method cannot be "strengthened" by a subclass. That is, a subclass cannot add more conditions to determine if a method should be called.
2. The condition of the program after the execution of a method cannot be "weakened" by a subclass. This means that the subclass should cause the state of the program to be in the same state as the base class after a method call. Subclasses are allowed to "strengthen" the postcondition of a program. For example, if the base class sets an alarm for a specific date, the subclass must do the same, but the result can be more precise by setting the specific hour as well.
3. Invariant conditions that exist in the base class, must also remain invariant in the subclass. Since invariant conditions are expected to be immutable, the subclass should not change them as it may cause a side effect in the behaviours of the base class or the program.
4. Immutable characteristics of a base class must not be changed by the subclass. Since classes can modify their own characteristics, a subclass can modify all the characteristics that it inherits from the base. However, the base class may encapsulate attributes that should be fixed values. These values are identifiable by observing whether or not they are changed in the

program, or by a method in the base class. If it is not changed, then these attributes are considered immutable. Subclasses can get around this problem by declaring and modifying their own attributes. The attributes of a subclass are not visible to the base class and therefore, do not affect the behaviour of the base class.

These rules are not programmatically enforced by any object-oriented language. In fact, overriding a base class's behaviors can have advantages. Subclasses can improve the performance of behaviours of its base class, without changing the expected results of said behavior.

As as an example, let's take a look at a class that is an abstraction of a department store. The base class may implement a naive searching algorithm that, in the worst case, iterates through the entire list of the items that the store sells. A subclass could override this method and provide a better search algorithm. Although the approach that the subclass takes to searching is different, the expected behavior and outcome is the same.

The Liskov Substitution Principle helps us determine if inheritance has been used correctly. If the expected behaviour between the subclass and the base class is different, then the principle has been violated.

Mark as completed