Instructor example

Web University by Harsha Vardhan

What does SOLID stand for? What are its principles?

S.O.L.I.D is an acronym for the first five object-oriented design (OOD) principles by Robert C. Martin.

S - Single-responsibility principle. A class should have one and only one reason to change, meaning that a class should have only one job.

O - Open-closed principle. Objects or entities should be open for extension, but closed for modification.

L - Liskov substitution principle. Instance of a child class must replace an instance of the parent class without affecting the results. That means both child class's instance and parent class's instance should offer same functionality of common methods.

I - Interface segregation principle. A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

D - Dependency Inversion Principle. Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

Benefits of SOLID principles
Accessibility: The SOLID Principle assures uncomplicated to control and access to object entities. The integrity of stable object-oriented applications gives easy access to objects, reducing the chances of unintended inheritance.

Ease of refactoring: Software changes with time. Thus, programmers are required to develop applications, considering the possibility of future changes. Software applications which are poorly constructed make it challenging to refactor, but it is quite easy to refactor your codebase with the SOLID principle.

Extensibility: The software goes through stages of improvement, including extra features. If the features of an application are not extended tactfully, this could affect prevailing functionalities and lead to unexpected problems. The

process of extensibility could be a tiring process since you require to design existing functionalities in the codebase, and if the existing functionalities are not rightly designed, this can make it even more challenging to include extra features. But the application of SOLID principles causes the extensibility procedure to be smoother.

Debugging: Debugging is an important aspect of the software development process. When software applications are not rightly devised, and the codebase gets clustered like spaghetti, it becomes difficult to debug applications. The SOLID principle embodies the feature of assuring that the software's debugging process is much more comfortable.

Readability: A well-designed codebase can be simple to comprehend and easy to read. Readability is also a crucial element in the software development process as it makes the refactoring and debugging operations easier, specifically in open-source projects. The SOLID principle method assures that your code is comparatively easier to read and interpret.

What is Bad Design?

If a system exhibits any or all of the following three traits then we have identified bad design:

The system is rigid: It's hard to change a part of the system without affecting too many other parts of the system

The system is fragile: When making a change, unexpected parts of the system break

The system or component is immobile: It is hard to reuse it in another application because it cannot be disentangled from the current application.

Describe the Single Responsibility Principle (SRP).

In Object Oriented Programming, Single Responsibility (SRP) ensures that every module or class should be responsible for single functionality supported by the software system. In other words, every class should have one and only reason to change it.

For example, In ASP.NET MVC HomeController class should be responsible related to Home Page functionality of software system. It should not involve in other aspects such as error handling or logging. Each of these functionalities should be accomplished by independent classes (say middleware, filters etc.).

Advantages of the Single Responsibility Principle

- You should modify your class more often, and every modification is more complex, has more side effects, and needs a lot more effort than

it should have. Thus, it's good to ward off these problems by ensuring that each class has just one responsibility.

- The class is simpler to comprehend. When the class just does "one thing", usually its interface has a few methods that are reasonably self-explanatory. It should also have a few member variables (less than seven).
- The class is effortless to manage. Changes are isolated, cutting down the probability of splitting other unrelated areas of the software. As programming flaws are inversely proportional to complexity, being simpler to figure out makes the code less inclined to bugs.
- The class is more reusable. If a class has different responsibilities, and just one of those is required in another area of the software, then the additional irrelevant responsibilities deter reusability. Owning a single responsibility implies the class should be reusable without modification.

Describe the Open Close Principle (OCP).

- The simplest approach to carry out the Open-Closed Principle in C# is to include the new functionalities by building new derived classes that need to be inherited from the original base class.
- Another step is to let the client access the original class with the help of an abstract interface.
- So, when there is a difference in need or any additional need comes, then rather than touching the prevailing functionality, it's always better proposed to design new derived classes and let the original class implementation be as it is.

Problems you'll get if you are not following the OCP in C#:

If you are not adhering to the OCP during the process of the application development process, then your application development may result in the following problems:

- If you let a function or class include new logic, then as a programmer you have to test the entire functionalities, including the old functionalities and the latest functionalities of the application.
- As a programmer, the onus is on you to inform the QA (Quality Assurance) team about the modifications beforehand so that they can, in advance, prepare themselves for regression testing along with testing the new features.
- If you are not adhering to the Open-Closed Principle, then it also splits the Single Responsibility Principle, since the module or class is going to achieve several responsibilities.
- If in a single class you are incorporating all the functionalities, then the maintenance of the class becomes challenging.

Explain Liskov Substitution Principle (LSP).

Liskov Substitution Principle (LSP) states that Objects in a program can be replaced by the instances of their subtypes without modifying the correctness of a program.

In other words, if A is subtype of B then instances of B may be replaced by the instances of A without altering the program correctness.

The LSP rules are:

- Subclass should incorporate all classes of the base class. This implies there should be no techniques that throw “NotImplementedException” or overriding existing methods of base class. Instead, you can create another method (with another name) with new / altered functionality.
- Overridden method of the parent class has to accept the exact parameters in the child class. For instance, you possess a method “CalculateSquare” in the parent class that accepts an int as a parameter. This implies you can’t point out the exact method in the child class, override it with the keyword new, and design an extra rule that will confirm that the number is not negative. If you attempt to pass -3 in the client code but utilize the child class rather than the superclass, the client code will have the exception. And that’s not the behavior that was proposed by the base class.

Advantages

- Stops code to break if someone by mistake has altered the base class with the derived class since its behaviour does not change
- Derived classes can readily throw exceptions for the method which are not managed by them.

Explain DIP (Dependency Inversion Principle)

Dependency Inversion Principle (DIP) is a design principle (guideline), which is a solution for the dependency problem.

- "The higher-level modules (clients) SHOULD NOT depend on low-level modules (dependencies). Both should depend on abstractions (interfaces or abstract class)."
- "Abstractions should not depend on details (both client and dependency). Details (both client and dependency) should depend on abstractions."

Advantage:

Low-level and high-level classes are loosely coupled

Problems if DIP is not implemented

- If DIP is not implemented, it indicates the high-level modules (classes that invoke services) and low-level modules (services) are tightly coupled.
- The developer of higher-level module SHOULD WAIT until the completion of development of lower-level module.
- Requires much code changes in to interchange an alternative lower-level module.
- Any changes made in the lower-level module effects changes in the higher-level module.
- Difficult to test a single module without effecting / testing the other module.

Explain Interface Segregation Principle

“Many client-specific interfaces are better than one general-purpose interface”.

ISP (interface segregation principle) demands that classes will only be capable of performing behaviors that are helpful in achieving their end functionality. Classes do not consist of behaviors they do not use.

Advantage:

The benefit of ISP is that it breaks large methods into minor, more precise methods. This makes the program simpler to debug for the below-mentioned reasons:

- There is limited code transferred between classes. Less code implies fewer bugs.
- A single method handles a smaller range of behaviors. If there is an issue with a behavior, you are only required to look over the minor methods.

Your submission

Cesar David Guerrero Regino

What does SOLID stand for? What are its principles?

What is Bad Design?

Describe the Single Responsibility Principle (SRP).

Describe the Open Close Principle (OCP).

Explain Liskov Substitution Principle (LSP).

Explain DIP (Dependency Inversion Principle)

Explain Interface Segregation Principle