Section Cheat Sheet (PPT)

Overview of SOLID Principles

"SOLID" is a set of five design patterns, whose main focus is to create loosely coupled, flexible, maintainabile code.

Broad goal of SOLID Principles:Reduce dependencies of various classes / modules of the application.

Single Responsibility Principle (SRP)

A software module or class should have one-and-only reason to change.

Liskov Subsitution Principle (LSP)

Subtypes must be substitutable for their base types.

Open-Closed Principle (OCP)

A class is closed for modifications; but open for extension.

Interface Segregation Principle (ISP)

No client class should be forced to depend on methods it does not use.

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend upon abstractions.

Dependency Inversion Principle (DIP)

Direct Dependency

Controller (Client)

```
public class MyController : Controller
{
  private readonly MyService _service;
  public MyController()
  {
    _service = new MyService(); //direct
  }
  public IActionResult ActionMethod()
```

```
  {
    _service.ServiceMethod();
  }
}
```

Service (Dependency)

```
public class MyService
{
  public void ServiceMethod()
  {
    …
  }
}
```

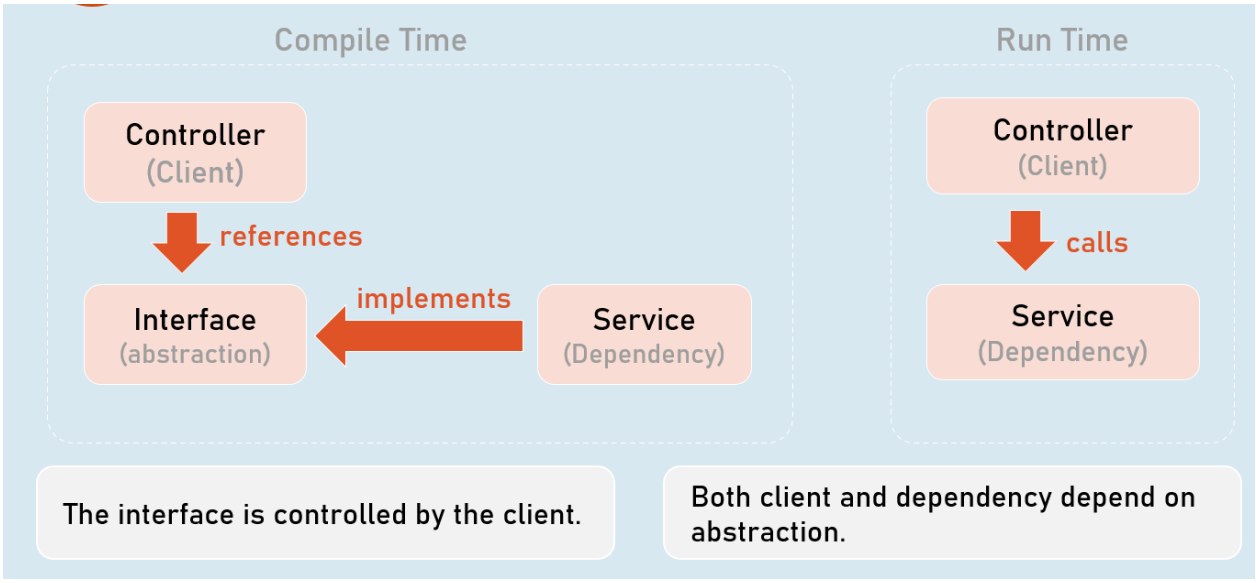
Dependency Problem

- Higher-level modules depend on lower-level modules.
- Means, both 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.

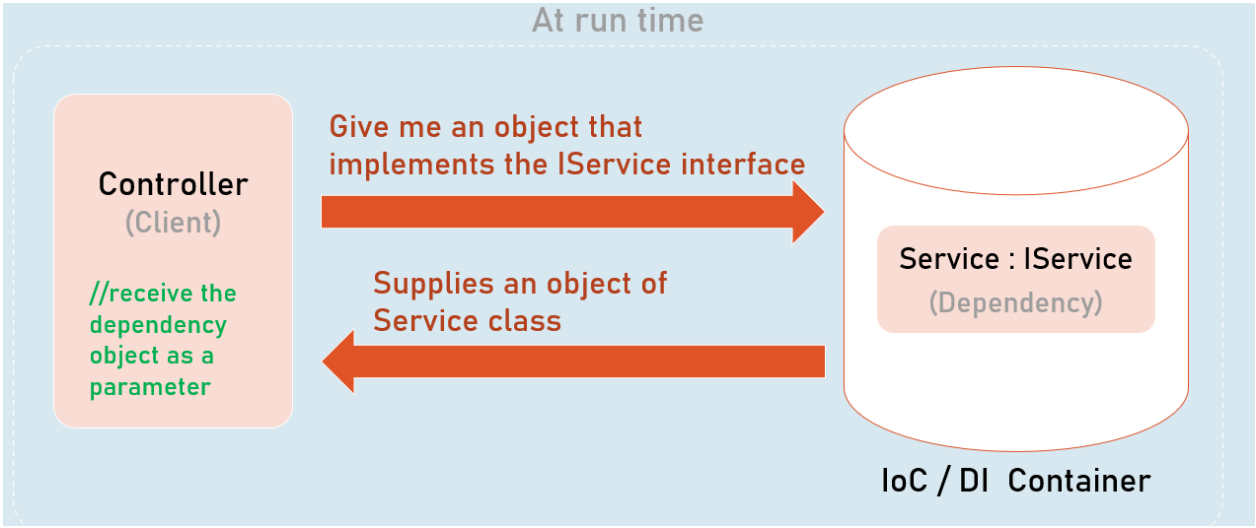
- Difficult to test a single module without effecting / testing the other module.

Dependency Inversion Principle

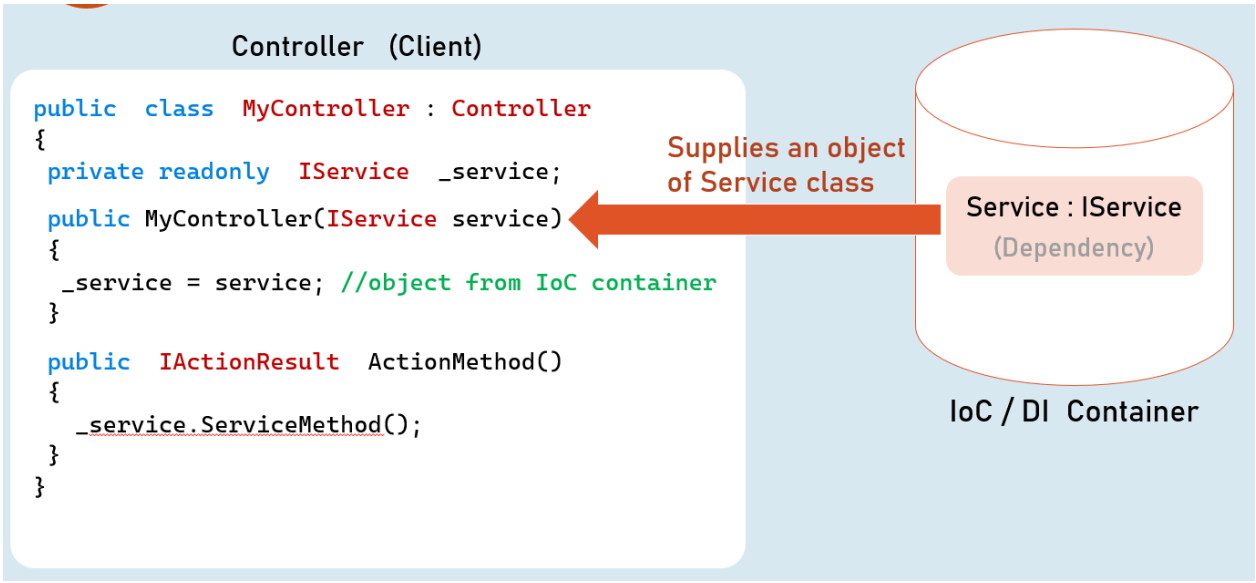
- Dependency Inversion Principle (DIP) is a design principle (guideline), which is a solution for the dependency problem.
- "The higher-level modules (clients) SHOULD NOT depend on low-level modules (dependencies).
- Both should depend on abstractions (interfaces or abstract class)."
- "Abstractions should not depend on details (both client and dependency).
- Details (both client and dependency) should depend on abstractions."



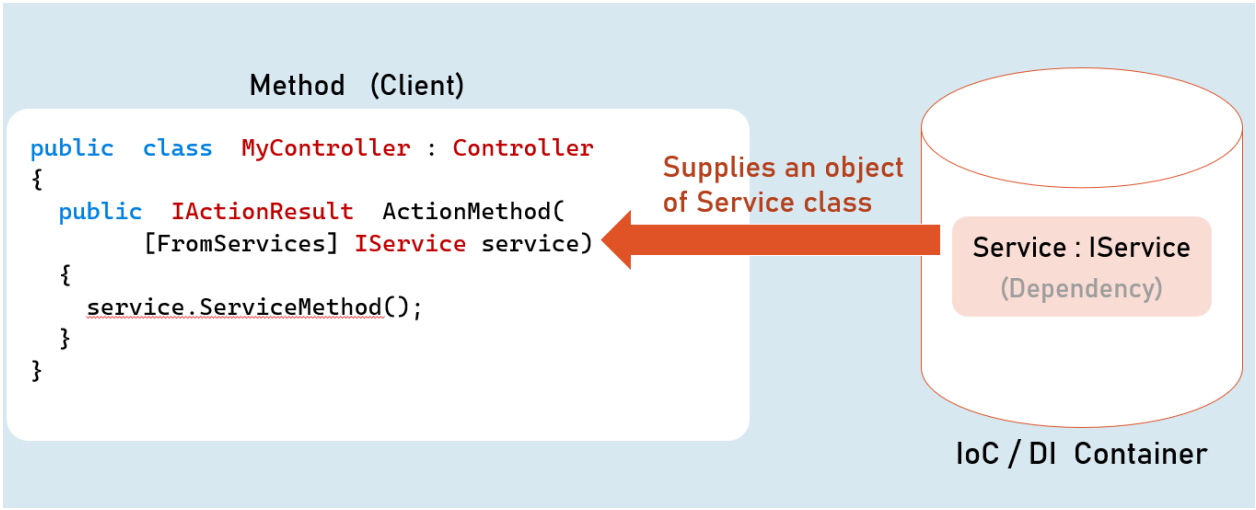
Dependency Injection



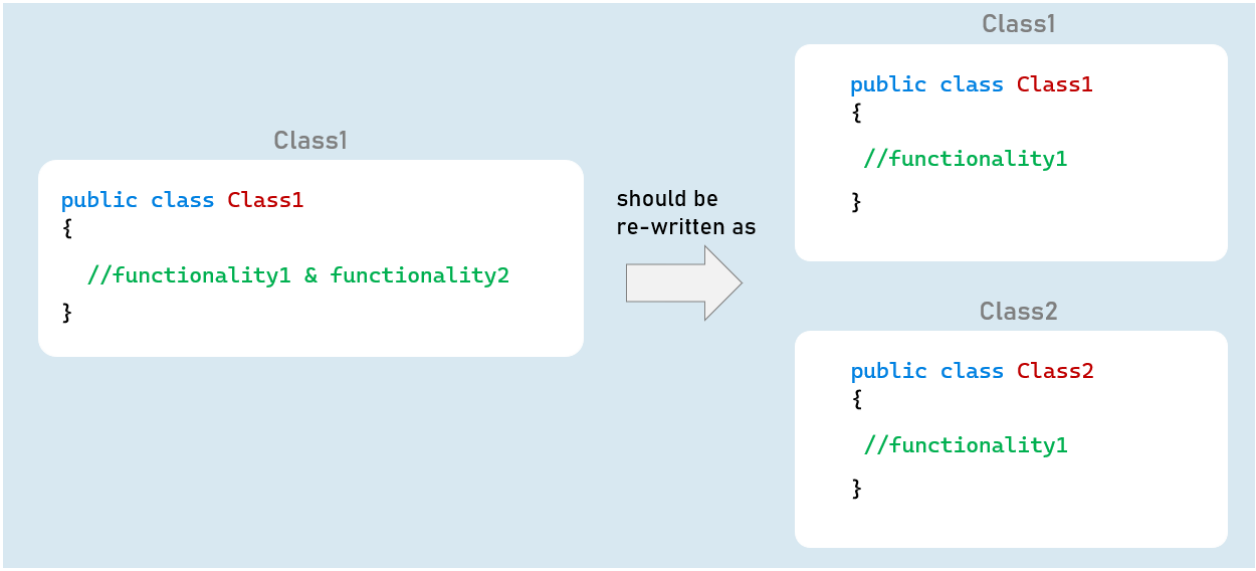
Constructor Injection



Method Injection



Single Responsibility Principle (SRP)



- A class should have one-and-only reason to change.
- A class should implement only one functionality.
- Avoid multiple / tightly coupled functionalities in a single class.
- Eg: A class that performs validation should only involve in validation.
- But it should not read configuration settings from a configuration file.
- But instead, it call a method of another class that reads configuration settings.

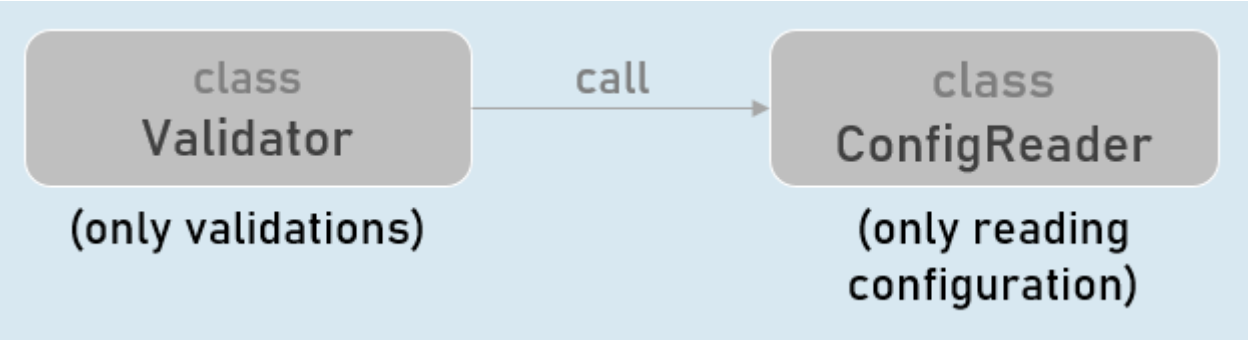
Interfaces

Create alternative implementation of the class by implementing the same interface.

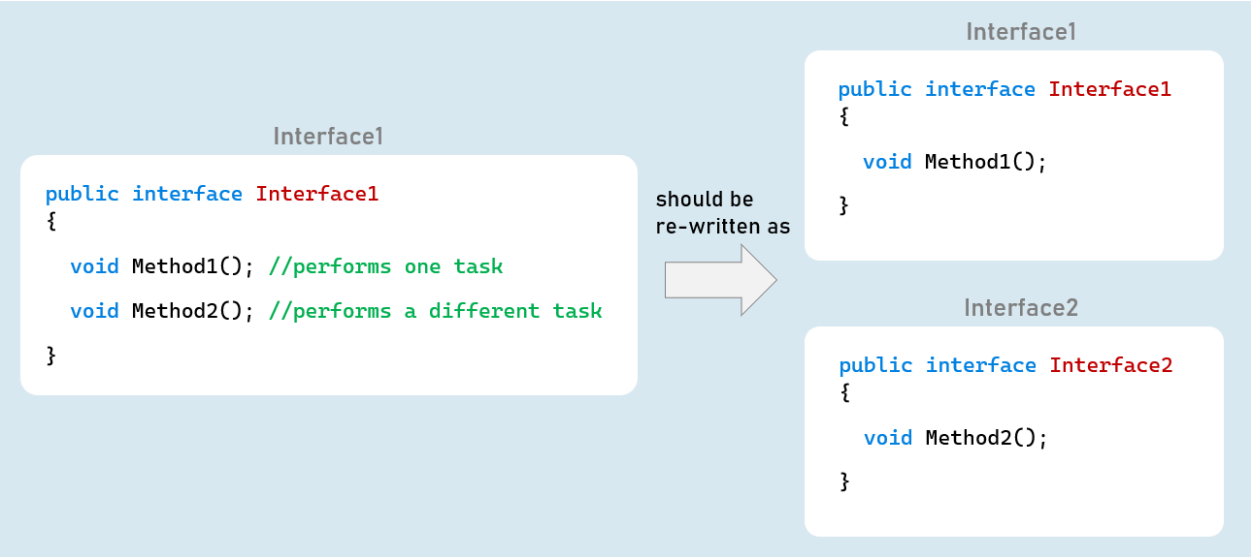
Benefit: Makes the class independent of other classes, in terms of its purpose / functionality.

So that, the classes become easier to design, write, debug, maintain and test.

Eg:



Interface Segregation Principle (ISP)



- No client class should be forced to depend on methods it doesn't use.
- We should prefer to make many smaller interfaces rather than one single big interface.
- The client classes may choose one or more interfaces to implement.
- Benefit: Makes it easy to create alternative implementation for a specific functionality, rather than recreating entire class.

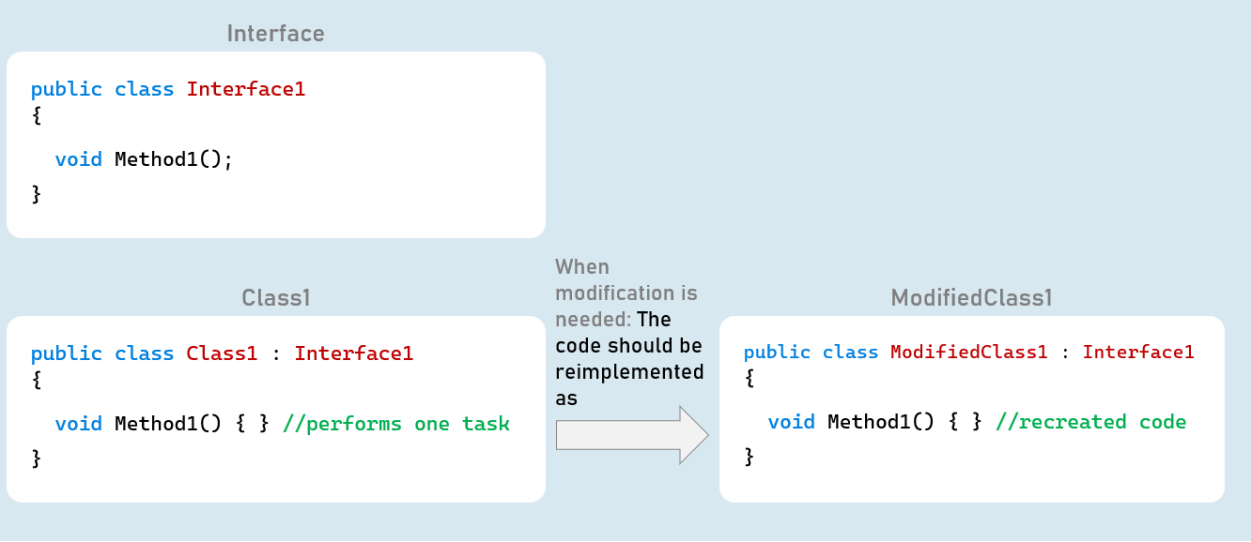
Eg:

Assume, a class has two methods: GetPersons() and AddPerson().

Instead of creating both methods in a single interface, create them as two different interfaces: IPersonGetter, IPersonAdder

1. interface IPersonsGetter (methods to get persons data)
2. interface IPersonsAdder(methods to create person)

Open/Closed Principle (OCP)



A class is closed for modifications; but open for extension.

You should treat each class as readonly for development means; unless for bug-fixing.

If you want to extend / modify the functionality of an existing class; you need to recreate it as a

separate & alternative implementation; rather than modifying existing code of the class.

Eg:

Assume, a class has a method: GetPersons().

The new requirement is to get list of sorted persons.

Instead of modifying existing GetPersons() method, you need to create an alternative class that gets sorted persons list.

Benefit: Not modifying existing code of a class doesn't introduce new bugs; and keeps the existing unit tests stay relavant and needs no changes.

```
class PersonGetter : IPersonGetter (GetPersons() method
       retrieves list of persons)
class SortedPersonGetter : IPersonGetter (GetPersons()
       method retrieves sorted list of persons)
```

Interfaces

Create alternative implementation of the class by implementing the same interface.

Inheritance

Create a child class of the existing class and override the required methods that needs changes.

Liskov Substitution Principle (LSP)

Parent Class

```
public class ParentClass
{
  public virtual int Calculate(int? a, int? b)
  {
    //if 'a' or 'b' is null, throw ArgumentNullException
    //return sum of 'a' and 'b'
  }
}
```

Child Class

```
public class ChildClass : ParentClass
{
  public override int Calculate(int? a, int b)
  {
    //if 'a' or 'b' is null, throw ArgumentNullException
    //if 'a' or 'b' is negative, throw ArgumentException
```

```
    //return product of 'a' and 'b'
  }
}
```

[Violates LSP]

Functions that use references of base classes must be able to use objects of derived classes without breaking / changing its functionality.

The child classes that override methods of base class, should provide same behavior.

Using object of parent class

ParentClass variable = new ParentClass();

variable.Method(); //executes ParentClass.Method

Using object of child class

ParentClass variable = new ChildClass();

variable.Method(); //executes ChildClass.Method

[Both methods should offer same functionality]

Functions that use references of base classes must be able to use objects of derived classes without breaking / changing its functionality.

The child classes that override methods of base class, should provide same behavior.

If a derived class overrides a method of base class; then the method of derived class should provide same behavior:

With same input, it should provide same output (return value).

The child class's method should not introduce (throw) any new exceptions than what were thrown in the base implementation.

The child class's method should not implement stricter rules than base class's implementation.

Benefit: Prevents code to break - if by mistake or wantedly, someone has replaced the derived class with its base class (or even vice versa), as its behavior doesn't change.