**SOLID Principles**

* Is a set of 5 design patterns, which mainly focus on creating loosely coupled, flexible and maintainable code = Reduces dependencies of various modules of application

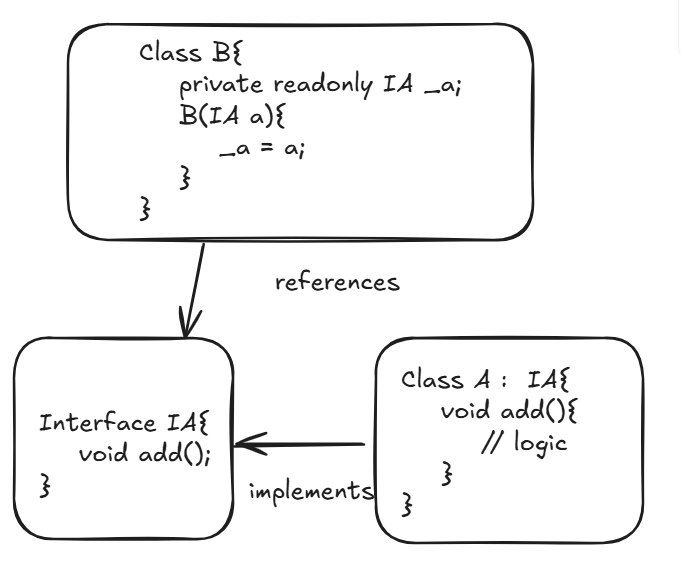
1. **S:** Single Responsibility Principle (SRP): A class should have one-and-one only reason to change
2. **O:** Open Closed Principle (OCP): A class should be closed for modifications(except for fixing bugs) but open for extensions
3. **L:** Liskov Substitution Principle (LSP): Subtypes should not override any existing methods of base class (in case if they override the existing functionality of base class should remain same method signature, return values, args, exceptions should remain same.)
4. **I:** Interface Segregation Principle: Instead of creating large interface with multiple methods, create smaller and meaningful interfaces like interface for adding, etc
5. **D:** Dependency Inversion Principle (DIP): Higher level modules should not depend on Lower-level modules instead both should depend on Abstractions(interfaces)

**Dependency Inversion Principle – DIP:**

* Whenever a class(B) calls another class(A) methods by directly creating object in it’s constructor as a result of this, higher level module (class B) depends on lower level module (class A) due to which:
  + Both classes becomes tightly coupled
  + Any effect in lower level class will effect higher level class
  + The developer of higher level module should wait until lower level code is completed
  + Difficult to test a single module without effecting/testing the other module

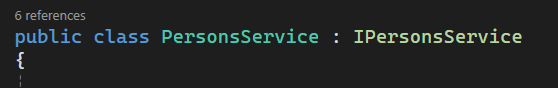
|  |  |
| --- | --- |
| Class A{ // lower level class  Void add(){  }  } | Class B{ // higher level class  B(){  A obj = new A(); // directly creating new object  }  } |

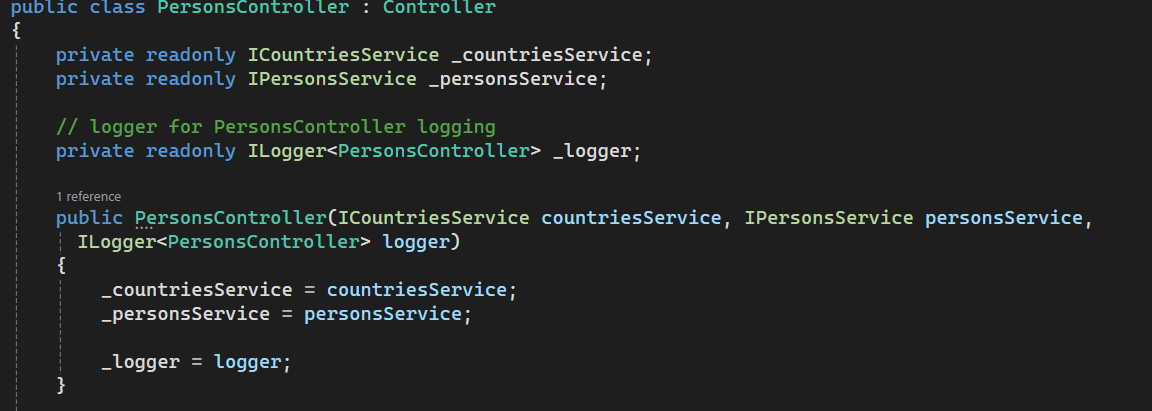
* DIP is a design pattern which solves this dependency problem
* We create an abstraction/interface and both higher level module and lower level module should depend on abstraction
  + The client(Higher level module) references the abstraction through injection
  + The Service(Lower level module) should implement the abstraction
  + Injection can be done in Constructor or method



* Here the object of class A in class B (higher module) is given by IOC container we need to register that service

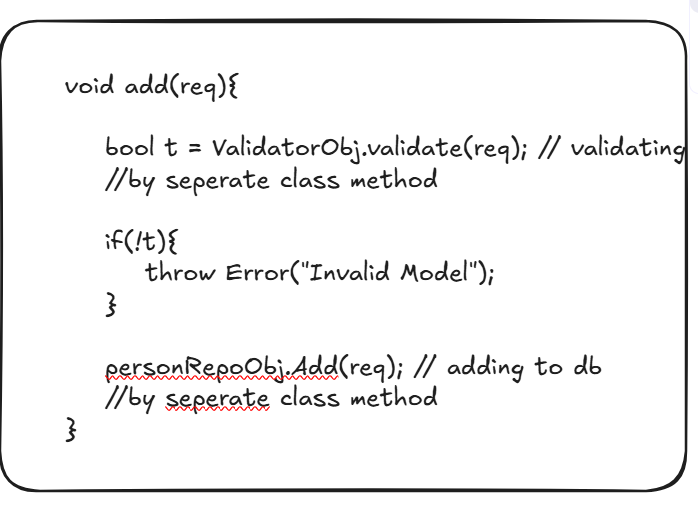






**Single Responsibility Principle:**

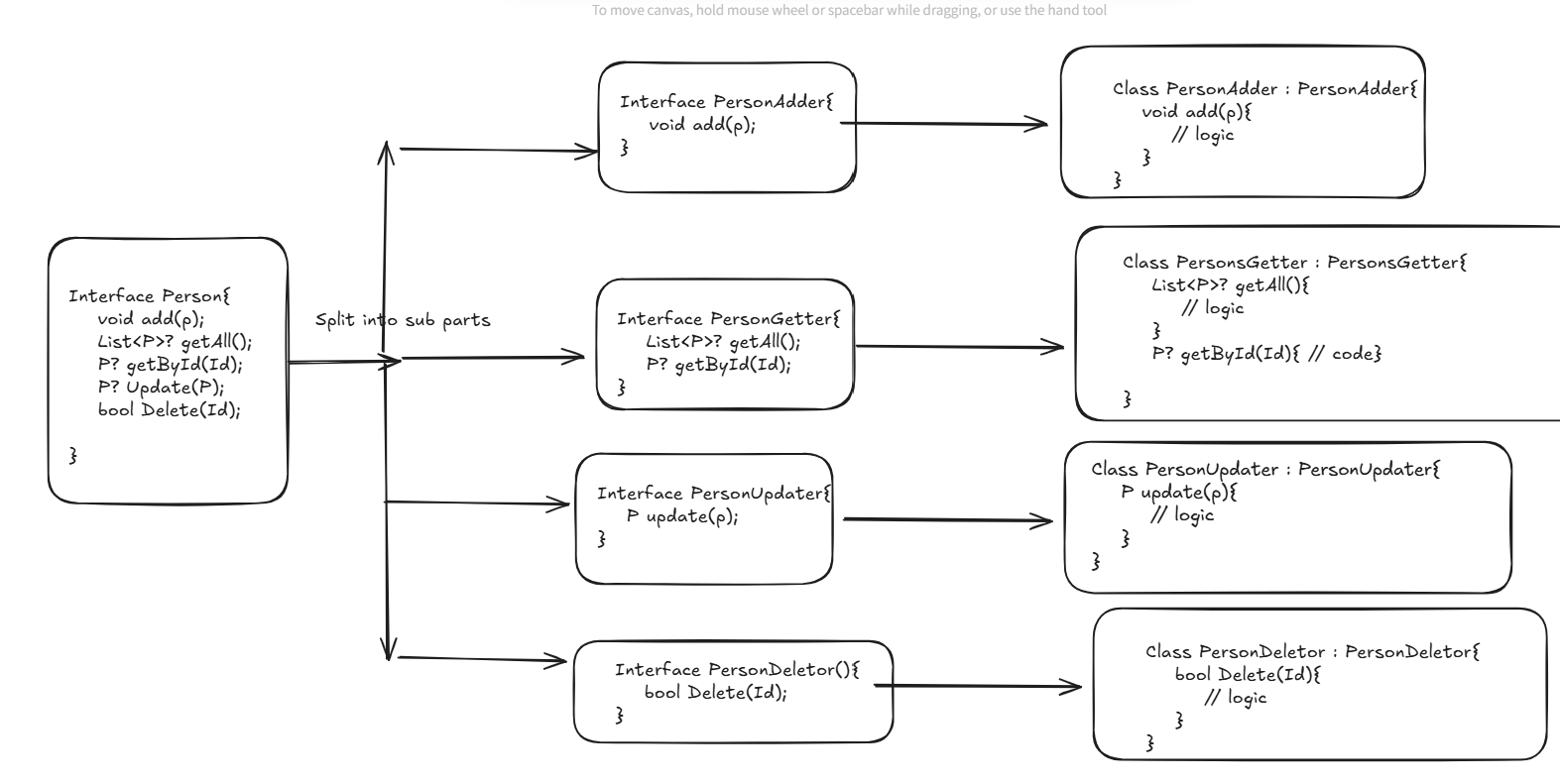
* Says that one class should provide only one functionality
* Eg: a class implementing Validation and then db access then adding doesnot follow SRP we can split to Validation Class, DB access class separately
* Disadvantes if SRP not followed: Updated code needs regression testing
  + Bugs identification is difficult



**Interface Segregation Principle: -**

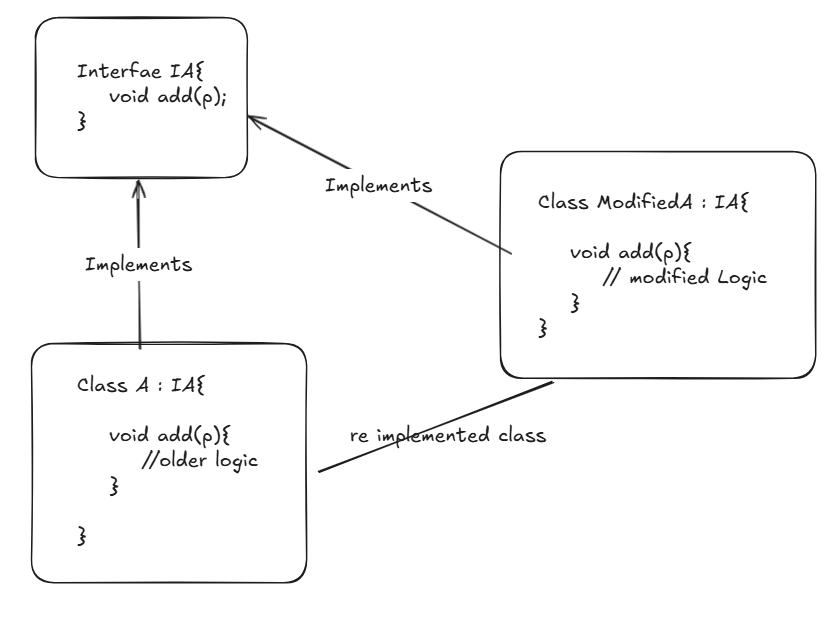
**No client class should be forced to depend on methods it doesn’t use. We should prefer to make many small interface rather than big one**

* Says that instead of loading all methods into single interface group them into separate interfaces like one interface for adding, interface for retrieving etc…
* Adv:
  + Easy to create alternative implementations for same method with different class that implements only new interface requirement
  + Client classes have choice to inject particular interface according to operation

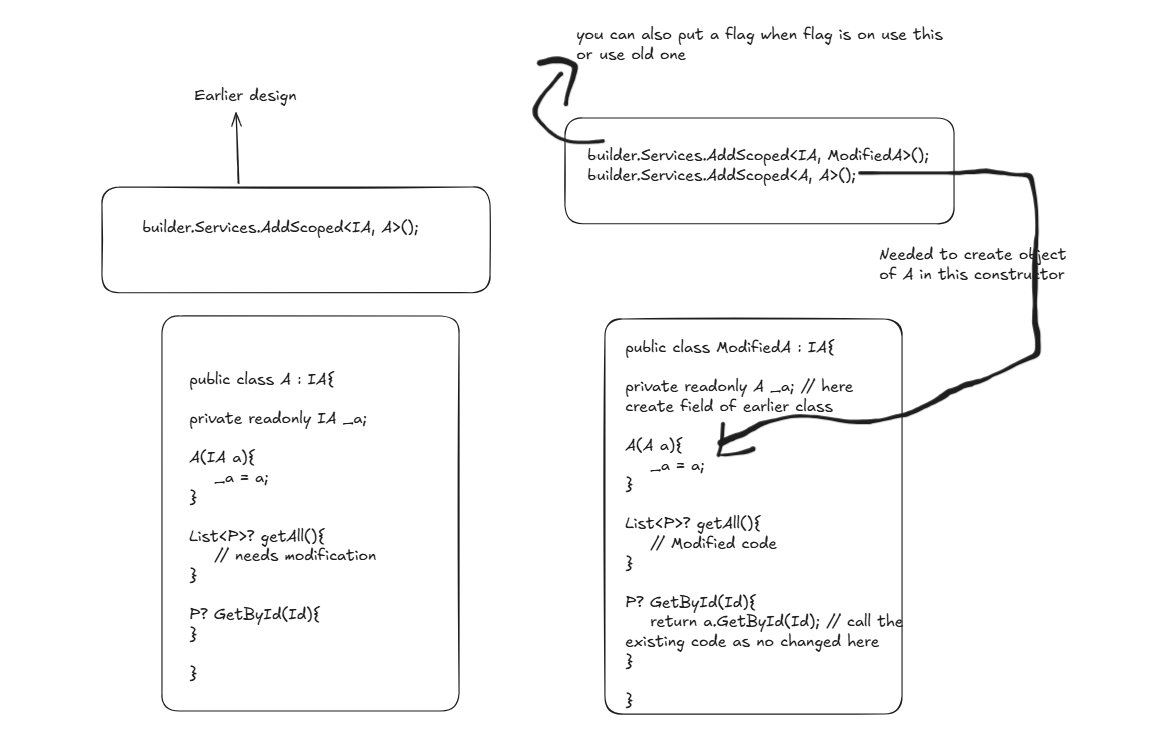


**Open Closed Principle – OCP**

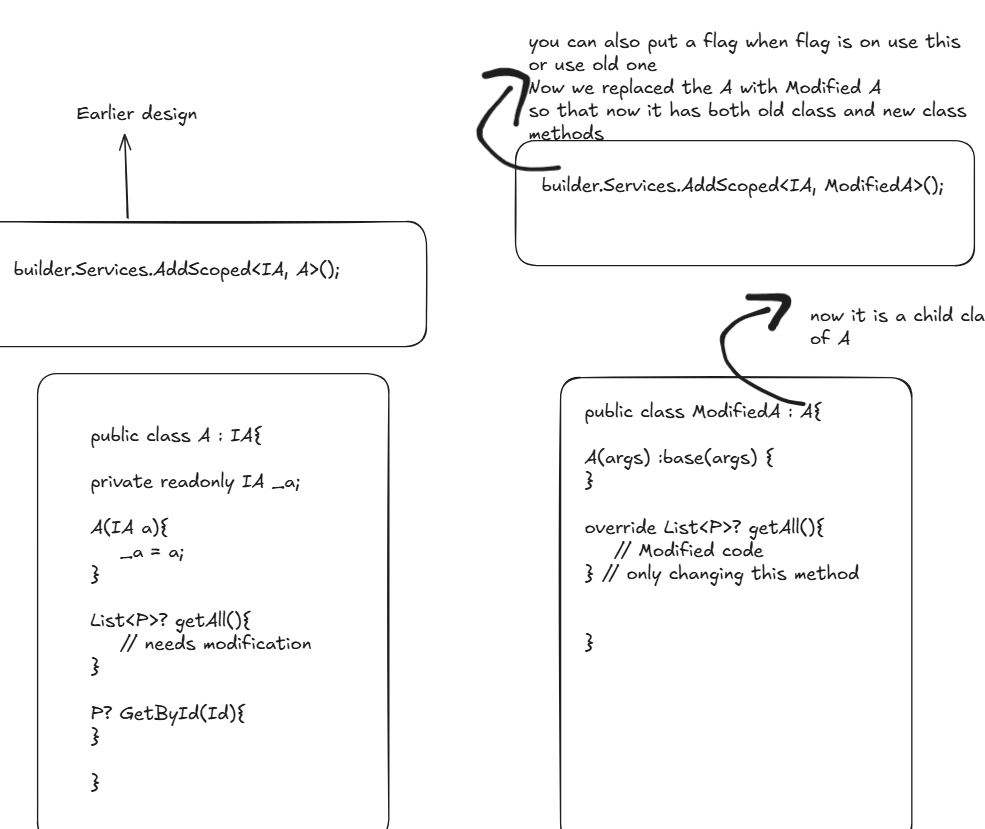
* Says that you should not make any changes in existing code base whenever you have requirement instead of altering you need to create a new class and reimplement that method
* Ie, Class should be closed for modifications but open for extensions
* If we edit in same code leads to:
  + Again testing
  + Chances of getting bugs
  + Old users who are using prev service gets affected



* Adv:
  + No need to test prev class we can add new tests to new classes
  + Both new users and old users can use the previous and new feature



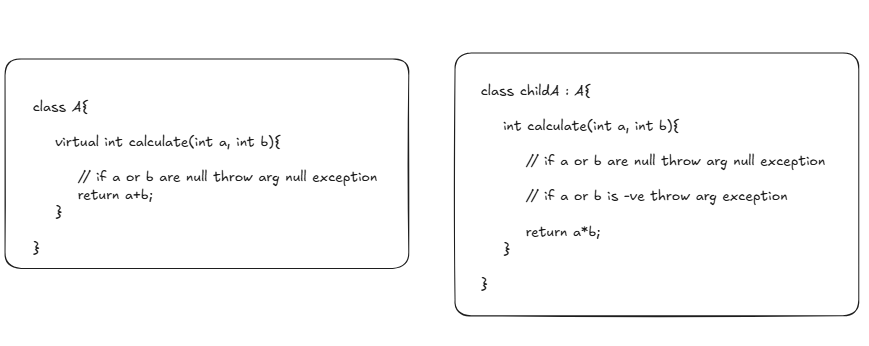
* There are 2 ways to implement OCP (interfaces and inheritance)
* By Interface: Create an alternate implementation of class by implementing same interface (See above Image) -- recommended
* By Inheritance: Create a child class of the existing class and override the required methods that needs changes
  + For the alternate implementation of class A, we create a child class and override that particular needed method
  + But you need to make all methods of class A to “virtual”
  + Here you may or may not override all methods you can choose to override your required method



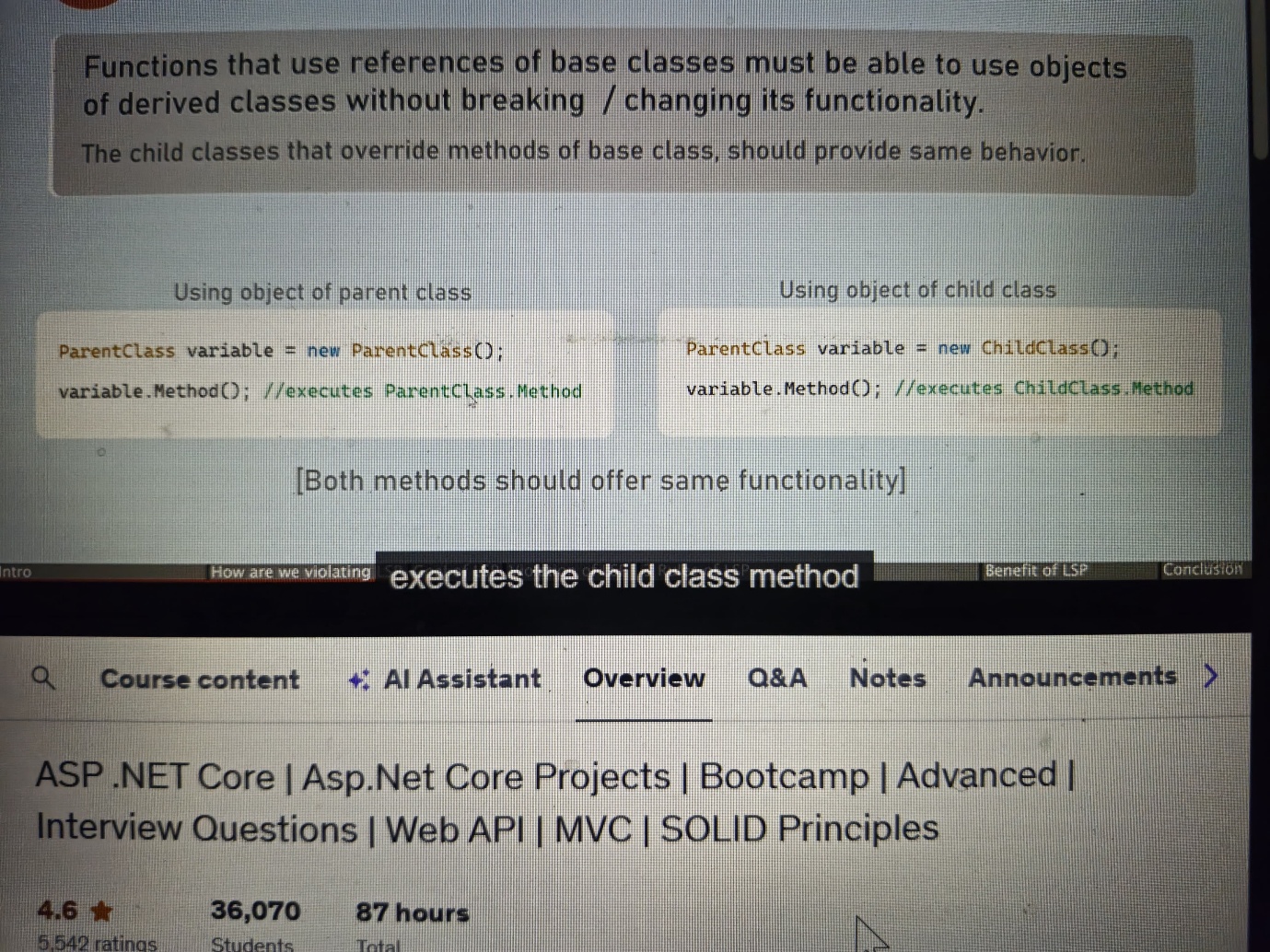
* Disadvantages with inheritance method : doesnot satisfy LSP sometimes

**Liskov Substitution Principle (LSP):**

* When you are overriding the parent class method in child class it should not introduce new exceptions, should have same method signature as in parents class and should not change the intention of parent class method
* In below example, it violates LSP as:
  + We introduced new exception in child class
  + The intention of method is to add but in child class it is changed to multiply



* Functions that use references of base classes must be able to use objects of derived class without breaking/changing its functionality ie, the child class that overrides methods of base class should provide the same behaviour.



* If a derived class overrides methods of base class, then method of derived class should provide same behaviour:
  + With same input it should provide same return value(output)
  + Child class shouldnot introduce new exceptions than what were thrown in base class
  + The child class methods should not implement strict rules than base class implementation
* Solution:
  + Instead of going with method 2 (inheritance) go with interface implementation(method 1) see in OCP
* Advantages:
  + Prevents code break: if by mistake, some one has replaced derived class with base class in configs as its behaviour doesnot change