IV. OBJECT-ORIENTED DESIGN PRINCIPLES

1. INTRODUCTION TO SOLID DESIGN PRINCIPLES

* Introduction
* Why use SOLID principles?
* Design principles.

I. Introduction :-

When creating software, we can follow good practices to avoid issues to make our code easier to understand, robust, and maintainable. Few of these practices are often termed as principles e.g., the SOLID principles refer to the best practices to be followed in OOD.

II. Why use SOLID principles?

If we don't adhere to the SOLID Principles,

1. The code may become tightly coupled with several components, which makes it difficult to integrate new features or bug fixes and sometimes leads to unidentified problems.
2. The code will be untestable, which effectively means that every change will need end-to-end testing.
3. The code may have a lot of duplication.
4. Fixing one issue results in additional errors.

If we adhere to the SOLID principles,

1. Reduce the tight coupling of the code, which reduces errors.
2. Reduces the code's complexity for future use.
3. Produce more extensible, maintainable, and understandable software code.
4. Produce the code that is modular, feature specific, and is extremely testable.

11. SOLID : SINGLE RESPONSIBILITY PRINCIPLE

* Introduction
* Real-life example.
* Book invoice application.
  - Violations.
* Conclusion.

1. Introduction :-

"A class should have only one reason to change."

The SRP helps us create simple classes that performs just one task. This helps in making modifications or adding extensions to the existing code much easier.

1. Real-life example :-

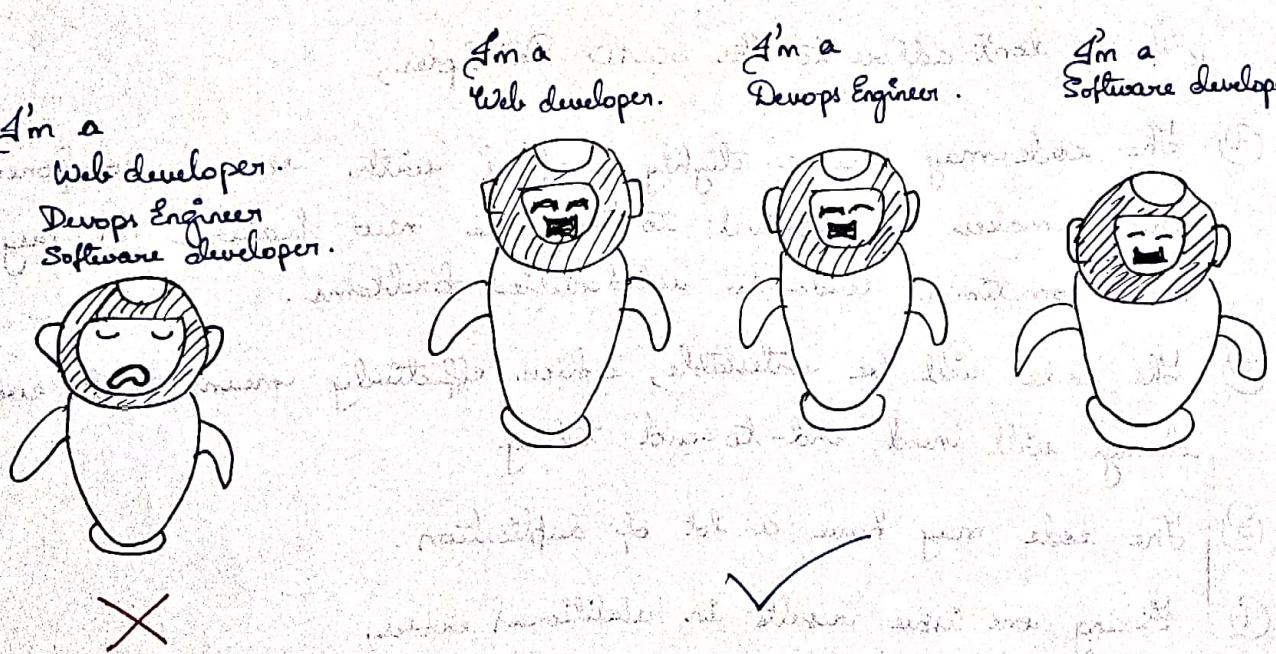


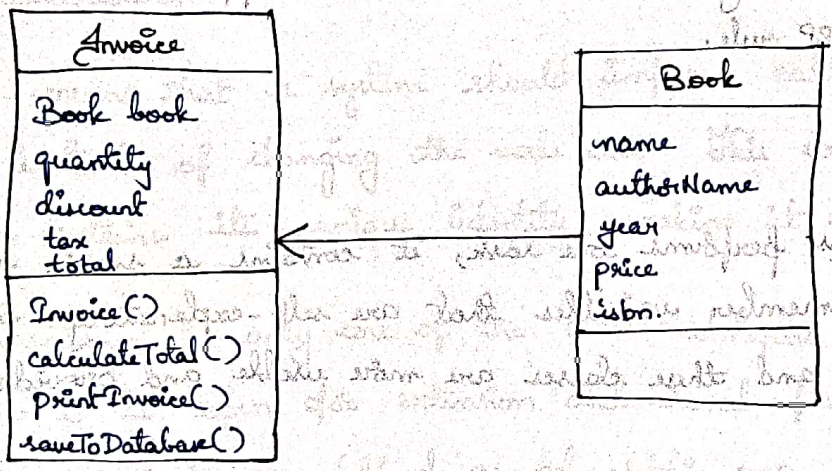
Fig., An example of SRP.

11. Book invoice application :-

Book invoice application has two classes:

1. Book - contains the data members related to the book.
2. Invoice - contains the following three functionalities:
   - > Calculating the price of the book.
   - > Printing the invoice.
   - > Saving the invoice into the database.

The following class diagram provides a blueprint of these classes:



Fig, The class diagram of the book invoice application without applying the SRP rule.

* Violations :-

1. The "Invoice" class is about invoices, but we have added print and storage functionality inside it. This breaks SRP rule.
2. If we want to change the logic of the printing or storage functionality in the future, we would need to change the class.

Instead of modifying the Invoice class for these uses, we can create two new classes for printing and persistence logic: "InvoicePrinter" and "InvoiceStorage", and move the methods accordingly, as shown below:

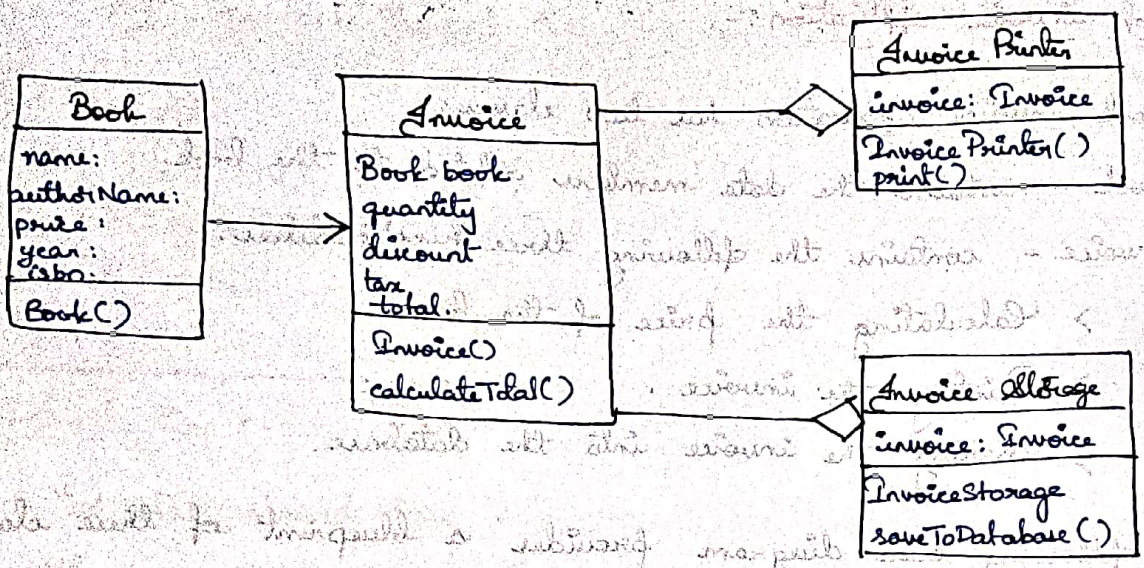


Fig., The class diagram of the book invoice application after applying the SRP rule.

IV. Conclusion:-

When a class performs one task, it contains a small number of methods and member variables that are self-explanatory. SRP helps to achieve this and these classes are more usable and provide easier maintenance.

III. SOLID: OPEN CLOSED PRINCIPLE

* Introduction
  → Example
    * Implementing the Open Closed Principle.
* → Conclusion.

1 Introduction :-

" A software artifact should be open for extension but closed for modification"

This means that a system should improve easily by adding new code instead of changing the code core. This way, the core code always retains its unique identity, making it reusable.

Note- "Inheritance is only one of the OCP techniques. We use interface because it is open for extension and closed for modification. Therefore, OCP is also defined as polymorphic OCP.

II. Example :-

Alex's cardboard business :-

①.

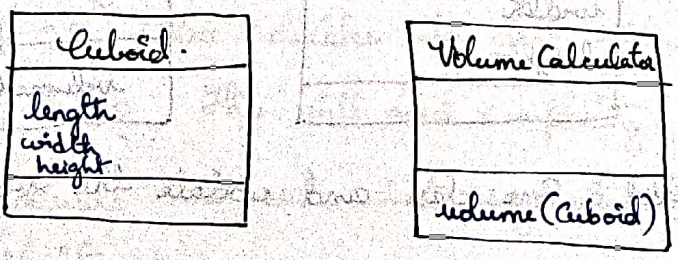


fig. The volume calculator class.

Initially Alex had only cuboid boxes.

The algorithm for volume(Cuboid) function is shown in flowchart below:

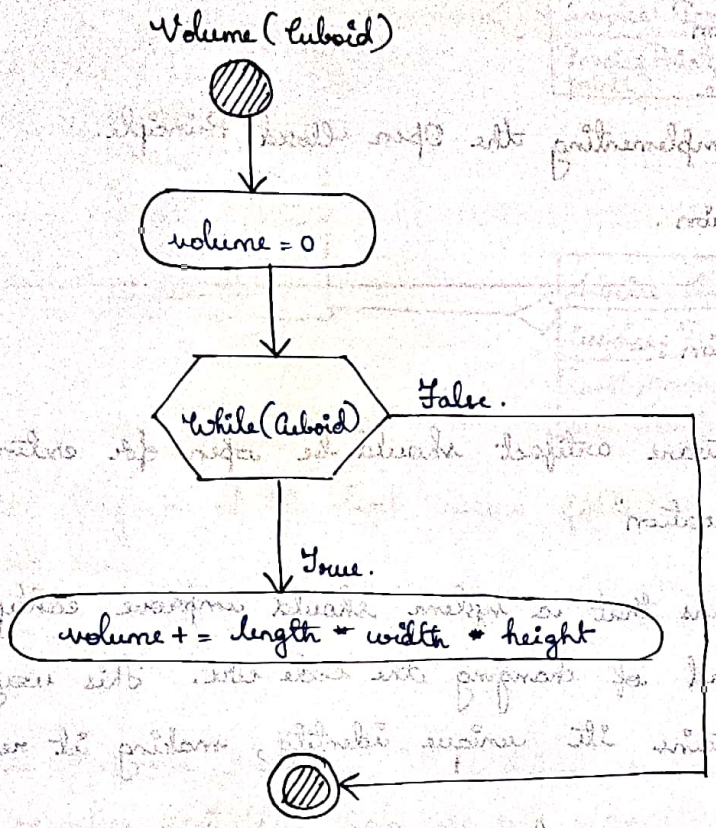


fig., The volume(Cuboid) function.

② As business grew, Alex also started selling cone-shaped boxes.

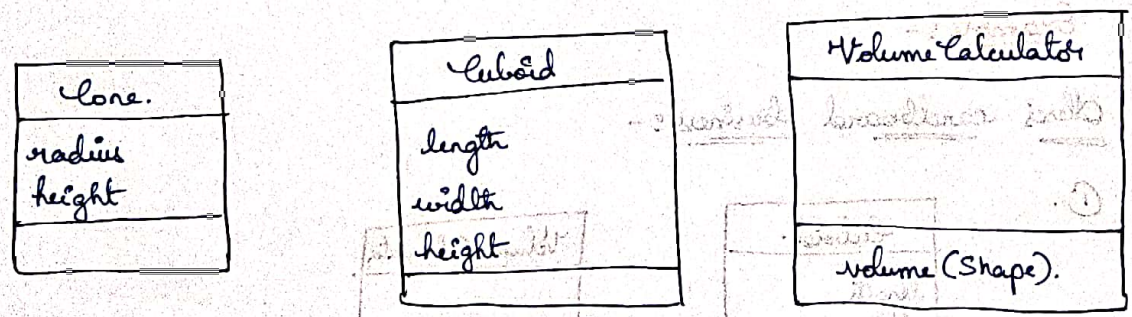
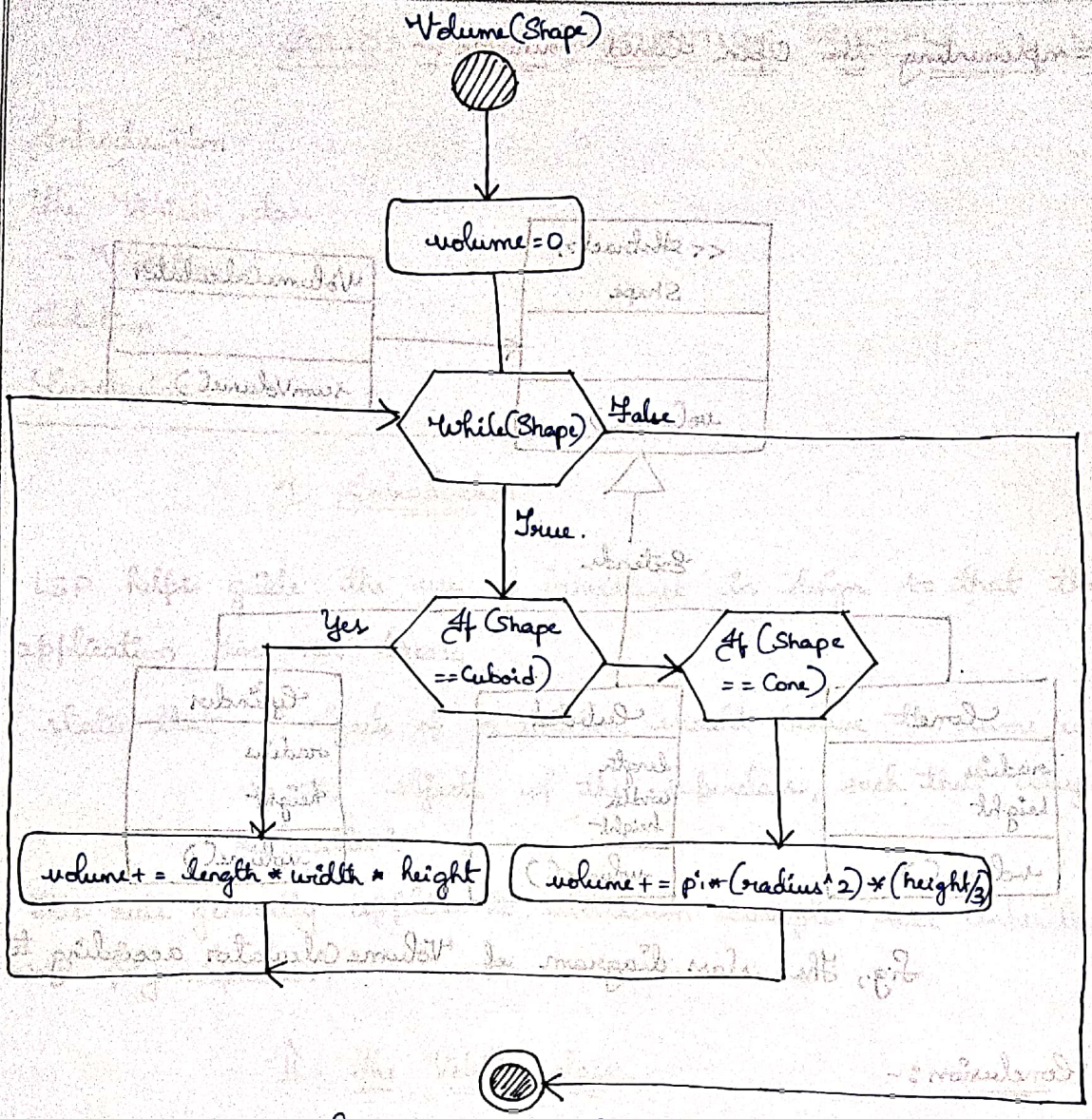


fig., Add a Cone class and update the volume method.

The algorithm for volume(Shape) function is shown in flowchart below:



Fig, The volume(Shape) function.

(3) With only two types of boxes, the class structure looks fine, but what if Alex decides to deal with more types of boxes, e.g, a cylinder box? This will add complexity to the volume(Shape). We will divide the code into segments using OCP to overcome this complexity.

III. Implementing the Open Closed Principle :-

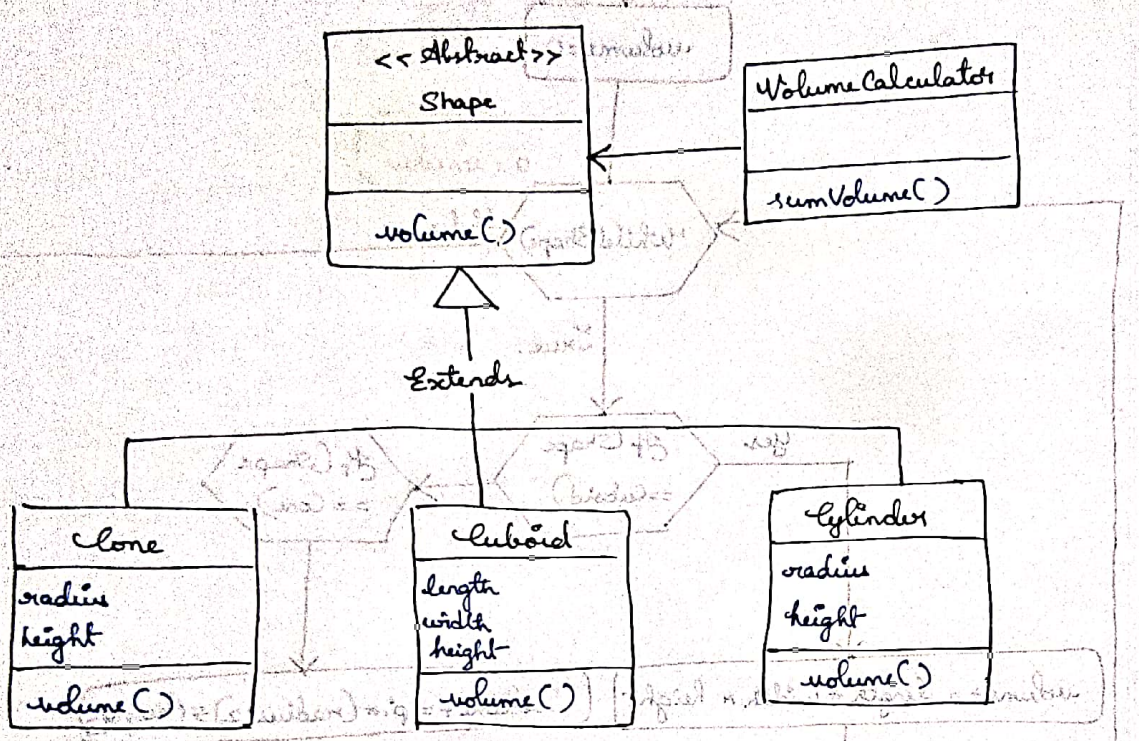


Fig., The class diagram of VolumeCalculator according to OCP.

IV. Conclusion :-

(1) A software system is easy to extend without the need for modification in the existing system. This is achieved by OCP.

(2) The system must be divided into small components, which are arranged, so that core code is always protected from new code.

IV. SOLID: LISKOV SUBSTITUTION PRINCIPLE.

* Introduction
* The Vehicle class.
  - Violation.
* Solution.
* Conclusion.

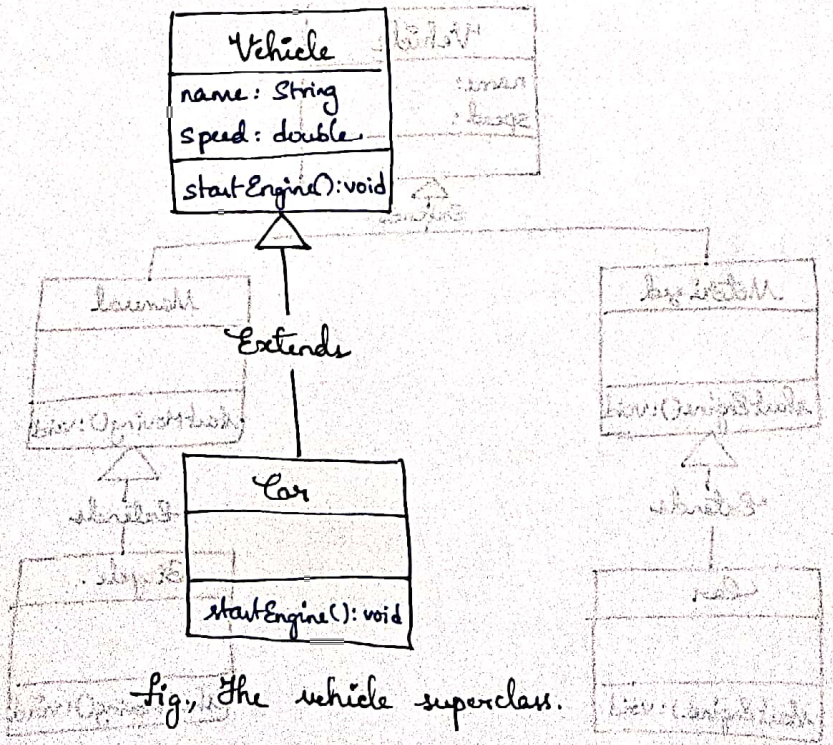
1. Introduction

→ LSP helps guide the use of inheritance in design so that the application does not break.

→ states that: "objects of a subclass should behave the same way as the objects of the superclass, such that they are replacable.".

→ This rule generally applies to abstraction concepts like inheritance and polymorphism.

2. The Vehicle class



Fig, The vehicle superclass.

- Violation :-

Let's add a Bicycle subclass in this system and see what happens:

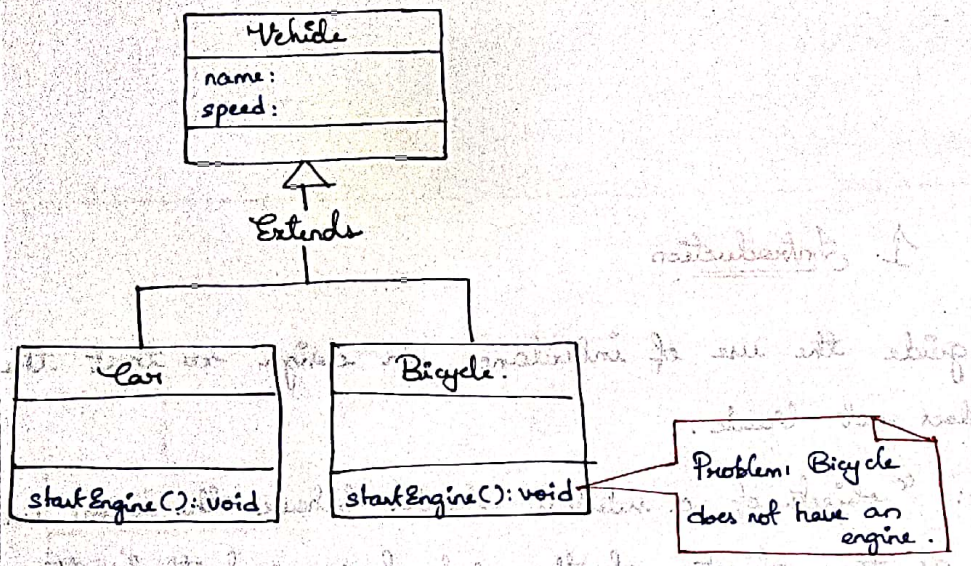


Fig., Violation of the LSP.

II. Solution:-

A possible fix would be to add two subclasses of Vehicle class that classify the vehicles as motorized vehicles and manual vehicles as follows:

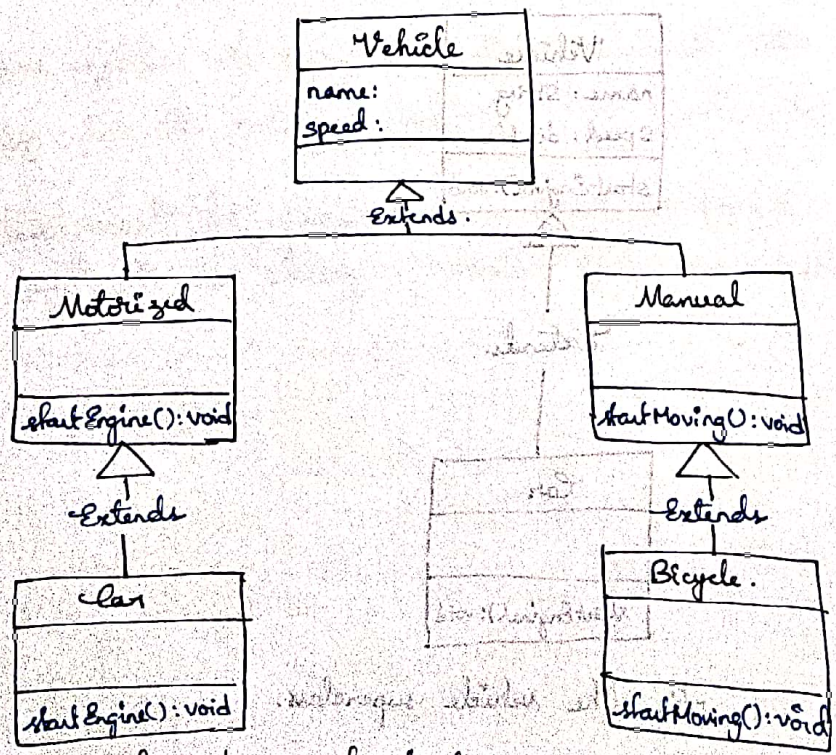


Fig., An example of the LSP implementation

* With this implementation, we have satisfied the LSP:

→ Car is substitutable with its superclass, Motorized, and Bicycle is substitutable with its superclass, Manual, without breaking the functionality.

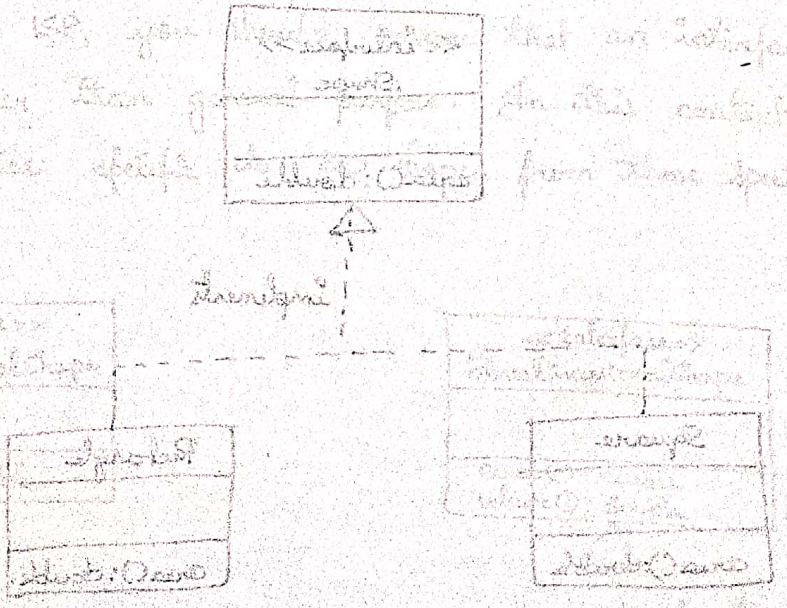
→ Their methods can also override the methods of the superclass.

IV. Conclusion :-

LSP is an important principle as small violation of substitutability of classes can cause the system to break down.

Benefits :-

1. It avoids the generalization of concepts that may not be needed in the future.
2. It makes the code maintainable and easier to upgrade.



4. INTERFACE SEGREGATION PRINCIPLE

* Introduction
* Example.
* Violation
* Solution
* Conclusion.

1. Introduction

* "does not recommend having methods that an interface would not use or require"
* The goal behind ISP is to have a code design that follows the correct abstraction guidelines and tends to be more flexible, which would help in making it more robust and reusable.
* This becomes the key when more and more features are added into the software, making it bloated and harder to maintain.

2. Example

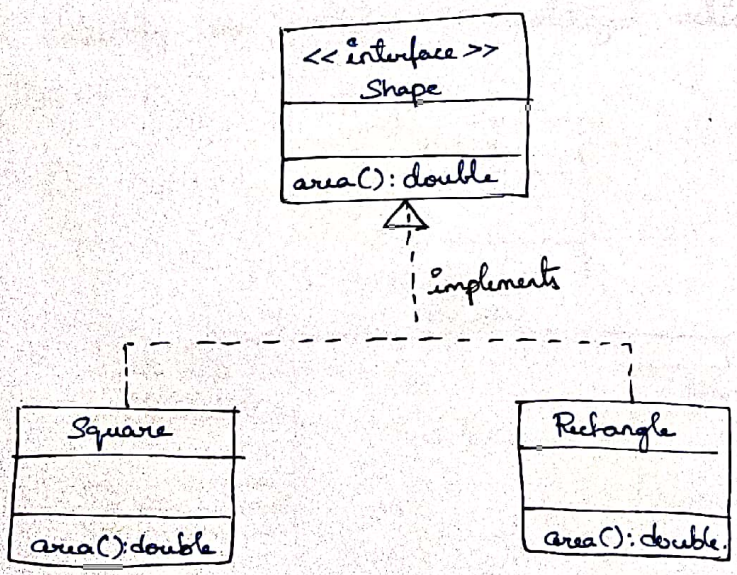


Fig., The Shape interface

III. Violation

while adding new feature volume().

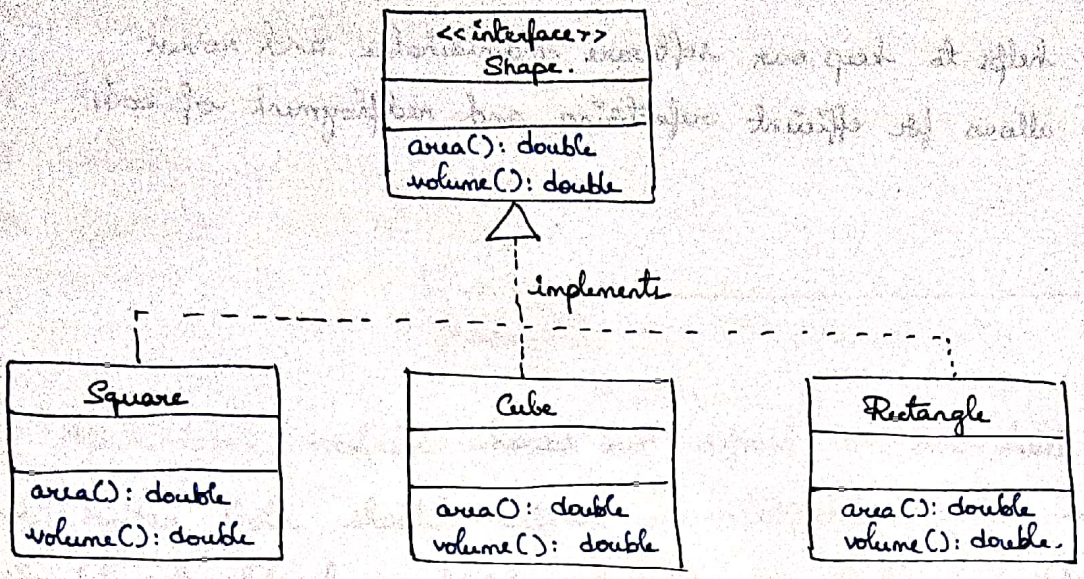


Fig., Violation of the ISP.

The violation leads to a problem. The 2-D shapes cannot have a volume, yet they're forced to implement the volume() method of Shape interface.

IV. Solution

To adhere to the ISP, you should ensure that an interface is client-specific rather than general-purpose. In this context, it means separating functionalities specific to 2D shapes from those specific to 3D shapes.

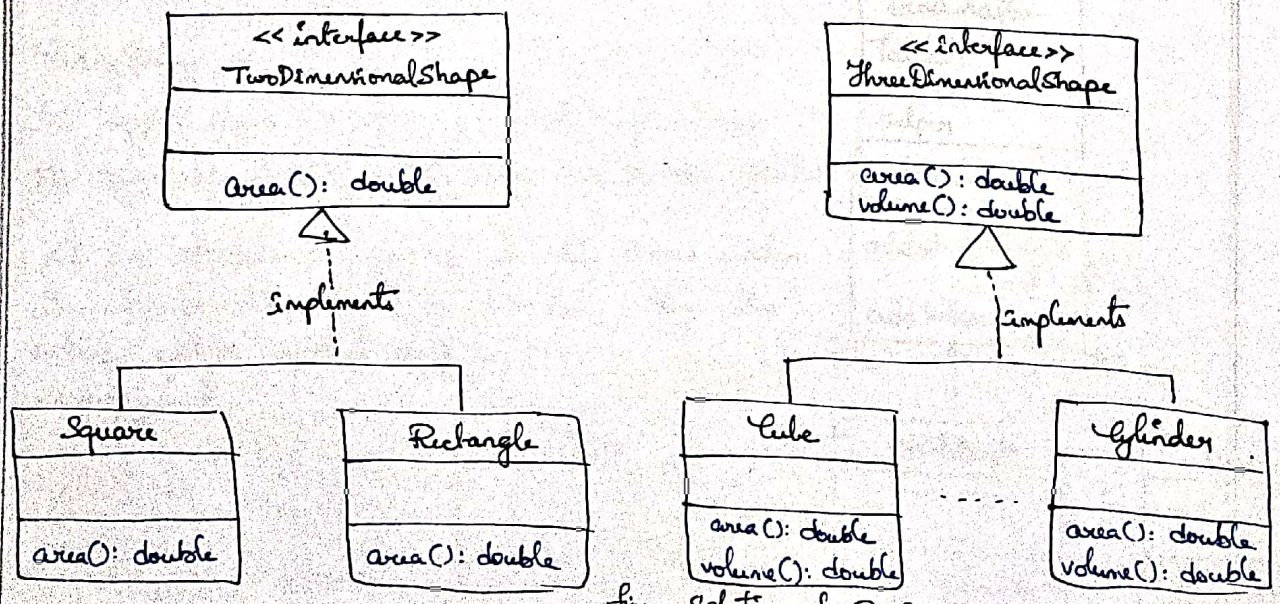


Fig., Solution of ISP.

5. Conclusion

* Benefits :-

① It helps to keep our software maintainable and robust.
② It allows for efficient refactoring and redeployment of code.

VI. SOLID: DEPENDENCY INVERSION PRINCIPLE

* Introduction
* Real-life example.
* Violation
* Solution
* Conclusion.

1. Introduction

* "High-level modules should not depend on low-level modules, but rather both should depend on abstractions. The abstractions should not depend on details. Instead, the details should depend on abstractions."

II. Real-life example

* School example:- Suppose there is a headmaster of a high school. Under the headmaster, there are faculty members such as teachers, assistants, and some helpers.

III. Violation

* The class diagram of this example:

→ Everything is exposed from the lower layer to upper layer & the headmaster must know the type of faculty that they can oversee beforehand.

→ If an additional type of faculty comes under the headmaster, such as secretary, then the entire system would need to be reconfigured.

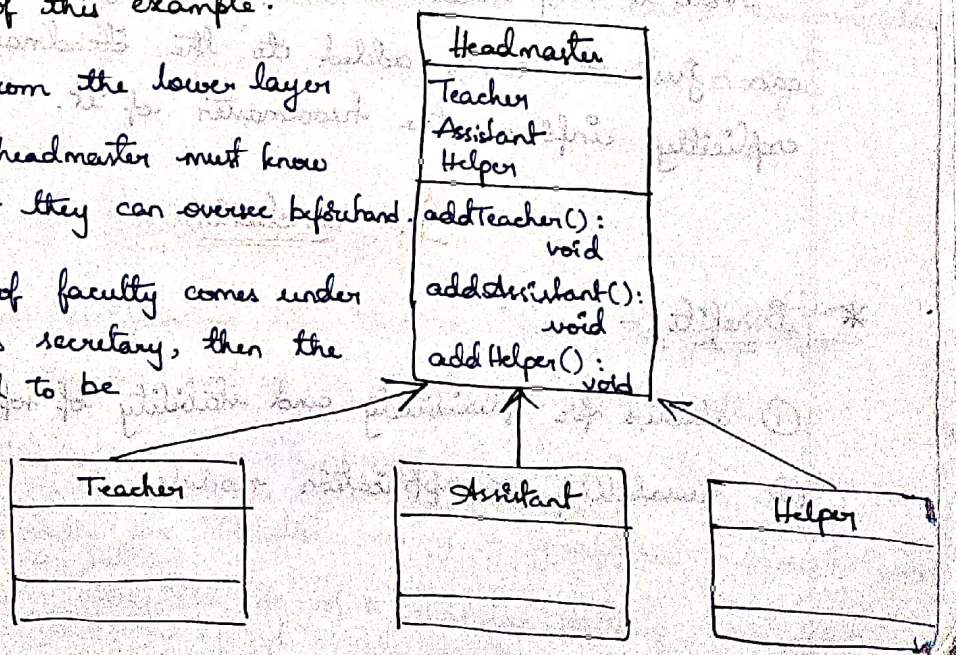


fig., Violation of DIP.

IV. Solution

Adding Faculty class would reduce the number of dependencies among modules and would make for an easily expandable system.

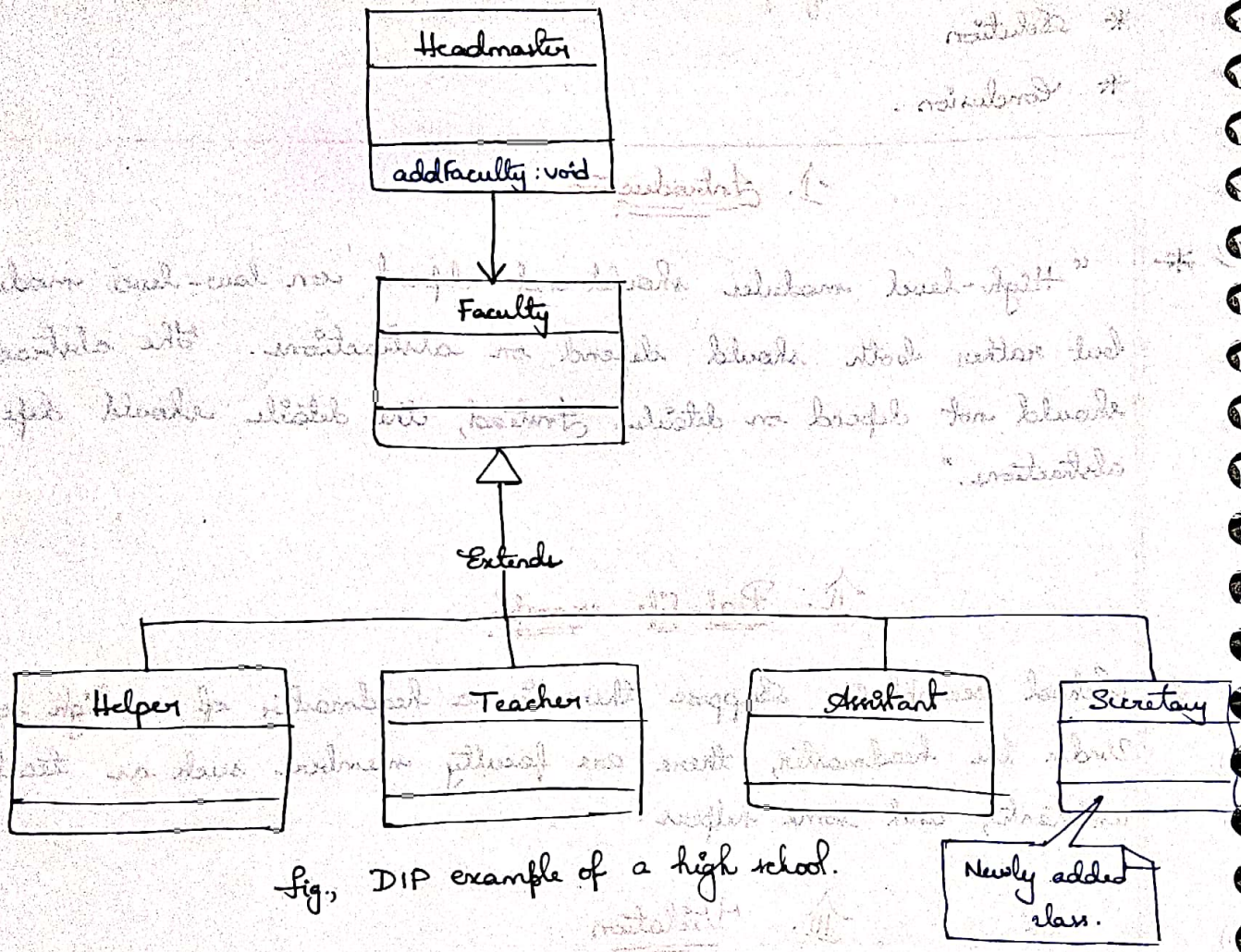


Fig., DIP example of a high school.

Now if any other kind of faculty is employed (eg, Secretary), they can just be easily added to the Headmaster without the need to explicitly inform the headmaster of it.

V. Conclusion

* Benefits :-

① allows for flexibility and stability of software.

② reusability of application modules