**S.O.L.I.D Principles**

**1) What are the S.O.L.I.D Principles?**

The SOLID principles are a set of five design principles that help in creating software that is easier to maintain, understand, and extend. These principles were originally defined by Robert C. Martin (also known as Uncle Bob) and are widely used in object-oriented programming languages like C++. Each principle focuses on a specific aspect of software design. Let's explore each of the SOLID principles:

1. Single Responsibility Principle
2. Open-closed principle
3. Liskov substitution principle
4. Interface Segregation Principle
5. Dependency Inversion Principle**.**

**2) What is the single Responsibility Principle?**

A class should have only one reason to change, meaning it should have only one responsibility or job. This principle promotes high cohesion and prevents classes from becoming too large and complex.

**Example:**

**#include <iostream>**

**using namespace std;**

**class Book {**

**public:**

**std::string name;**

**std::string authorName;**

**int year;**

**int price;**

**std::string isbn;**

**public:**

**Book(std::string name, std::string authorName, int year, int price, string isbn):name(name),authorName(authorName),**

**year(year),price(price),isbn(isbn){}**

**};**

**class Invoice**

**{**

**Book book;**

**int quantity;**

**double discountRate;**

**double taxRate;**

**double total;**

**public:**

**Invoice(Book book, int quantity, double discountRate, double taxRate):book(book),quantity(quantity),discountRate(discountRate),taxRate(taxRate)**

**{ total = calculateTotal();}**

**double calculateTotal()**

**{**

**double price = ((book.price - book.price \* discountRate) \* quantity);**

**double priceWithTaxes = price \* (1 + taxRate);**

**return priceWithTaxes;**

**}**

**void printInvoice() {**

**cout<< quantity << "x " << book.name << " " << book.price << "$"<<endl;**

**cout<<"Discount Rate: " << discountRate << endl;**

**cout<<"Tax Rate: " << taxRate<<endl;**

**cout<<"Total: " << total <<endl;**

**}**

**};**

**int main()**

**{**

**Book book("C++", "XYZ", 2021, 200, "1");**

**Invoice inv(book, 10, 10, 5);**

**inv.printInvoice();**

**return 0;**

**}**

**Here is our invoice class. It also contains some fields about invoicing and 3 methods:**

* **calculateTotal method, which calculates the total price,**
* **printInvoice method, which should print the invoice to the console, and**
* **saveToFile method, responsible for writing the invoice to a file.**

The first violation is the **printInvoice** method, which contains our printing logic. The SRP states that our class should only have a single reason to change, and that reason should be a change in the invoice calculation for our class.

But in this architecture, if we wanted to change the printing format, we would need to change the class. This is why we should not have printing logic –“  
mixed with business logic in the same class.

There is another method that violates the SRP in our class: the **saveToFile** method. It is also an extremely common mistake to mix persistence logic with business logic.

Don't just think in terms of writing to a file – it could be saving to a database, making an API call, or other stuff related to persistence.

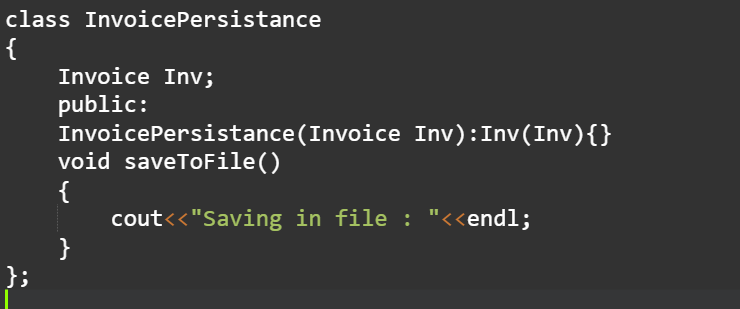
**So how can we fix this print function, you may ask?**

We can create new classes for our printing and persistence logic so we will no longer need to modify the invoice class for those purposes.

We create 2 classes, **InvoicePrinter** and **InvoicePersistence**, and move the methods.

A screenshot of a computer code

Description automatically generated with low confidence



**2) what is the Open-Close Principle?**

**Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that you should be able to add new functionality without modifying existing code**.

We should be able to add new functionality without touching the existing code for the class.

This is because whenever we modify the existing code, we are taking the risk of creating potential bugs.

So, we should avoid touching the tested and reliable production code if possible.

But how are we going to add new functionality without touching the class, you may ask? It is usually done.

with the help of interface and abstract classes.

Example: In the previous example, They want invoices to be saved to a database so that we can search

them easily. We think okay, this is easy peasy boss, just give me a second.

**We create the database, connect to it, and we add a save method to our**InvoicePersistence **class:**

A screenshot of a computer program

Description automatically generated with medium confidence

**Unfortunately, we, as the lazy developer for the bookstore, did not design the classes to be easily.**

**extendable in the future. So, in order to add this feature, we have modified the**InvoicePersistence **class.**

**If our class design obeyed the Open-Closed principle, we would not need to change this class.**

**So, as the lazy but clever developer for the bookstore, we see the design problem and decide to refactor**

**the code to obey the principle.**

**class InvoicePersistence**

**{**

**Public:**

**Virtual void save(Invoice invoice) = 0;**

**}**

**Class DatabasePersistence: public InvoicePersistence**

**{**

**Public:**

**Void save(Invoice invoice){ //save DB**

**}**

**}**

**Class FilePersistence: public InvoicePersistence**

**{**

**Public:**

**Void save(Invoice invoice){ //save to file }**

**}**

**3) what is the Liskov substitution principle?**

The Liskov Substitution Principle (LSP) is Subtypes must be substitutable for their base types without affecting the correctness of the program. In other words, objects of a superclass should be able to be replaced with objects of its subclasses without causing any errors or unexpected behavior.

**Example:**

**#include <iostream>**

**using namespace std;**

**class Rectangle {**

**protected:**

**int width;**

**int height;**

**public:**

**void setWidth(int w) {**

**width = w;**

**}**

**void setHeight(int h) {**

**height = h;**

**}**

**int area() {**

**return width \* height;**

**}**

**};**

**class Square : public Rectangle {**

**public:**

**void setWidth(int w) override {**

**width = w;**

**height = w;**

**}**

**void setHeight(int h) override {**

**width = h;**

**height = h;**

**}**

**};**

**void printArea(Rectangle& rect) {**

**rect.setWidth(5);**

**rect.setHeight(4);**

**std::cout << "Area: " << rect.area() << std::endl;**

**}**

**int main() {**

**Rectangle rect;**

**Square square;**

**printArea(rect); // Output: Area: 20**

**printArea(square); // Unexpected behavior: Output should be Area: 20, but it is Area: 16**

**return 0;**

**}**

* **In this example, we have a Rectangle base class and a Square derived class. According to the LSP, we should be able to substitute a Rectangle object with a Square object without affecting the behavior of the program. However, in this case, the Square class violates the LSP because it changes the behavior defined by the Rectangle class.**
* **When we call the printArea function with a Square object, the overridden setWidth and setHeight methods in the Square class modify both the width and height to be the same. This behavior is correct for a square but not for a rectangle. Consequently, the printArea function produces unexpected results when it computes the area of the Square object.of sides.**

**#include <iostream>**

**using namespace std;**

**class shape{**

**public:**

**virtual int area() = 0;**

**};**

**class Rectangle : public shape{**

**protected:**

**int width;**

**int height;**

**public:**

**void setWidth(int w) {**

**width = w;**

**}**

**void setHeight(int h) {**

**height = h;**

**}**

**int area() {**

**cout<<"Inside Rectangle area calculation : "<<endl;**

**return width \* height;**

**}**

**};**

**class Square : public shape {**

**int side;**

**public:**

**void setSide(int w) {**

**side = w;**

**}**

**int area()**

**{**

**cout<<"Inside squre area calculation : "<<endl;**

**return (side \* side);**

**}**

**};**

**void printArea(shape& Shape) {**

**std::cout << "Area: " << Shape.area() << std::endl;**

**}**

**int main() {**

**Rectangle rect;**

**rect.setWidth(10);**

**rect.setHeight(20);**

**printArea(rect); // Output: Area: 20**

**Square square;**

**square.setSide(20);**

**printArea(square); // Unexpected behavior: Output should be Area: 20, but it is Area: 16**

**return 0;**

**}**

**4) Interface Segregation Principle?**

**Segregation means keeping things separated, and the interface Segregation Principle is about separating.**

**the interfaces.**

* Clients should not be forced to depend upon interfaces that they don't use.
* The Interface Segregation Principle states that clients should not be forced to implement interfaces they don't use. Instead of one fat interface, many small interfaces are preferred based on the group of methods, each one serving one submodule.

**#include <iostream>**

**using namespace std;**

**class Printer {**

**public:**

**virtual void print() = 0;**

**virtual void scan() = 0;**

**virtual void fax() = 0;**

**};**

**class MultifunctionPrinter : public Printer {**

**public:**

**void print() override {**

**cout<<"MultifunctionPrinter::print : "<<endl;**

**}**

**void scan() override {**

**cout<<"MultifunctionPrinter::scan : "<<endl;**

**}**

**void fax() override {**

**cout<<"MultifunctionPrinter::fax : "<<endl;**

**}**

**};**

**class SimplePrinter : public Printer {**

**public:**

**void print() override {**

**cout<<"SimplePrinter::print : "<<endl;**

**}**

**void scan() override {**

**cout<<"SimplePrinter::scan : "<<endl;**

**}**

**void fax() override {**

**cout<<"SimplePrinter::fax : "<<endl;**

**}**

**};**

**int main()**

**{**

**SimplePrinter obj;**

**obj.print();**

**return 0;**

**}**

**Solution:**

**#include <iostream>**

**using namespace std;**

**class IPrinter {**

**public:**

**virtual void print() = 0;**

**};**

**class IScanner {**

**public:**

**virtual void scan() = 0;**

**};**

**class IFax**

**{**

**public:**

**virtual void fax() = 0;**

**};**

**class MultifunctionPrinter : public IPrinter, public IScanner, public IFax {**

**public:**

**void print() override {**

**cout<<"MultifunctionPrinter::print : "<<endl;**

**}**

**void scan() override {**

**cout<<"MultifunctionPrinter::scan : "<<endl;**

**}**

**void fax() override {**

**cout<<"MultifunctionPrinter::fax : "<<endl;**

**}**

**};**

**class SimplePrinter : public IPrinter {**

**public:**

**void print() override {**

**cout<<"SimplePrinter::print : "<<endl;**

**}**

**/\*void scan() override {**

**cout<<"SimplePrinter::scan : "<<endl;**

**}**

**void fax() override {**

**cout<<"SimplePrinter::fax : "<<endl;**

**}\*/**

**};**

**int main()**

**{**

**SimplePrinter obj;**

**obj.print();**

**return 0;**

**}**

In this revised example, **we have separate interfaces for printing, scanning, and faxing: IPrinter, IScanner, and IFax. Each interface defines only the methods specific to its functionality.**

The MultifunctionPrinter class implements all three interfaces since it supports printing, scanning, and faxing. On the other hand, the SimplePrinter class only implements the IPrinter interface because it is a simple printer without scanning or faxing capabilities.

By using specific interfaces, we ensure that clients can depend only on the interfaces they need. This promotes a more modular and flexible design, as clients are not burdened with unnecessary dependencies on methods they don't utilize.

**5) what is the Dependency inversion principle?**

The Dependency Inversion Principle (DIP) is a design principle in object-oriented programming that states that high-level modules should not depend on low-level modules. Instead, both should depend on abstractions. It promotes the decoupling of classes and the use of interfaces or abstract classes to define common behavior**.**

**Example:**

**// Example without DIP**

**class Database {**

**public:**

**void saveData(std::string data) {**

**// Code to save data to a specific database**

**}**

**};**

**class User {**

**private:**

**Database db;**

**public:**

**void saveUserData(std::string userData) {**

**db.saveData(userData);**

**}**

**};**

In this example, the User class depends directly on the Database class. The User class creates an instance of the Database class and uses it to save user data. This creates a tight coupling between the User class and the Database class, making it difficult to change or substitute the database implementation.

**To adhere to the Dependency Inversion Principle, we can introduce an abstraction or interface:**

**Solution:**

**// Example with DIP**

**class IDataStorage {**

**public:**

**virtual void saveData(std::string data) = 0;**

**};**

**class Database : public IDataStorage {**

**public:**

**void saveData(std::string data) override {**

**// Code to save data to a specific database**

**}**

**};**

**class User {**

**private:**

**IDataStorage& storage;**

**public:**

**User(IDataStorage& storage) : storage(storage) {}**

**void saveUserData(std::string userData) {**

**storage.saveData(userData);**

**}**

**};**

In this revised example, we define the IDataStorage interface, which declares the saveData method. The Database class implements this interface. The User class no longer creates an instance of the Database class directly but instead accepts an IDataStorage object through its constructor.

By introducing the interface and making the User class depend on the abstraction (IDataStorage) rather than a concrete implementation (Database), we achieve decoupling and inversion of dependencies. Now, the User class can work with any class that implements the IDataStorage interface, allowing for flexibility and easier substitution of different storage implementations. This promotes modularity and improves testability by facilitating the use of mock objects or alternative implementations during testing.