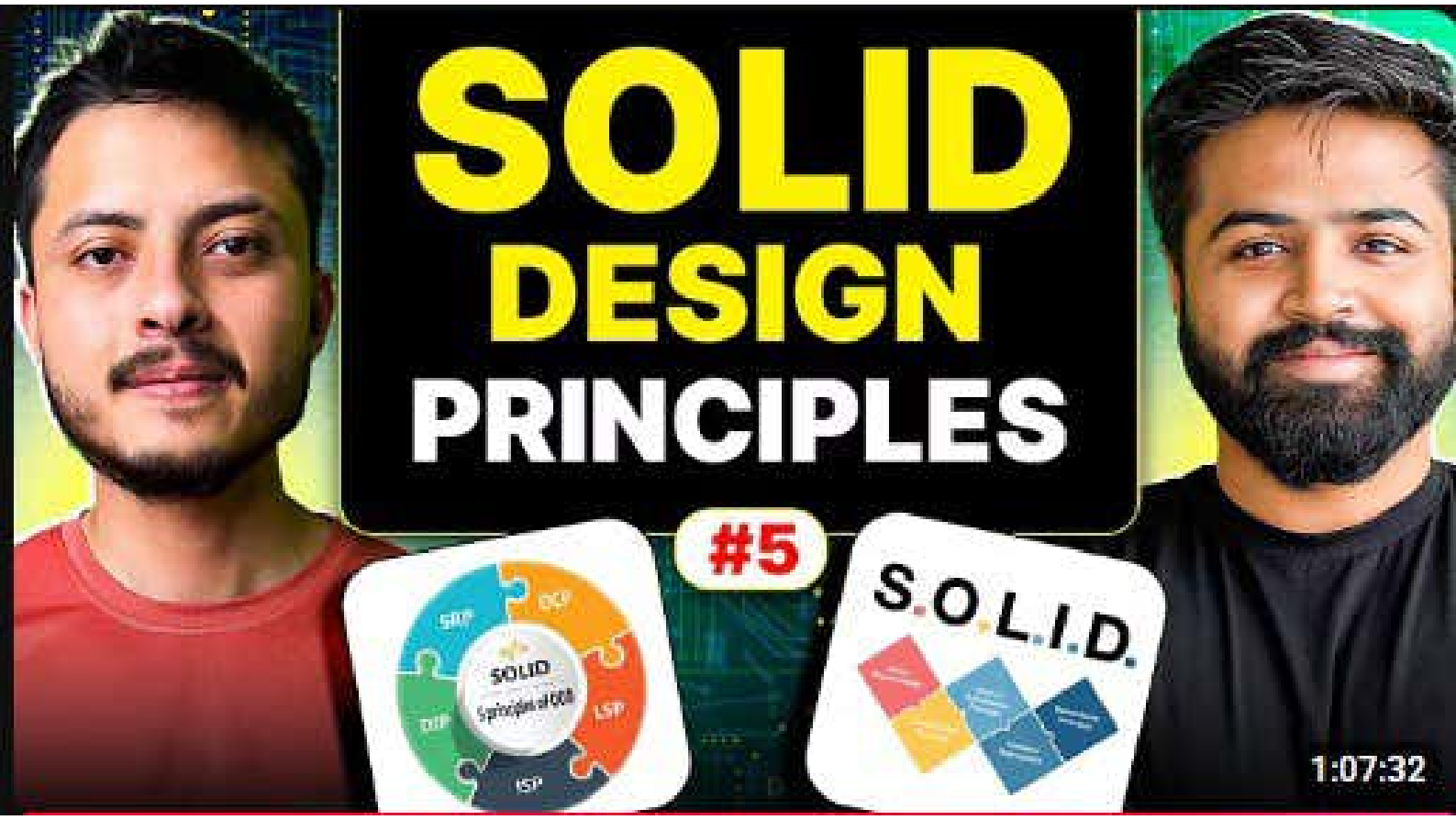

SOLID
DESIGN
PRINCIPLES
#5
SRP
OCP
LSP
ISP
DIP
SOLID
Principles
S.O.L.I.D.
1:07:32

May
16-05-2025
Friday

DAY-5
Lecture-5

Study now, be proud later.

SOLID Design Principles

* Introduction:

There will be many classes in real-world Project and to maintain these classes can be difficult. so there are some rules we should follow to manage classes. If we don't do this then there will be lots of bugs and code readability is not good.

Example: In a house there are lots of wires going through the same lane. So if one wire get fault then it will be difficult to found out that wire from that bunch. Because they are tightly coupled.

* Problems without design principles:

→ Maintainability
→ Readability
→ BUGS

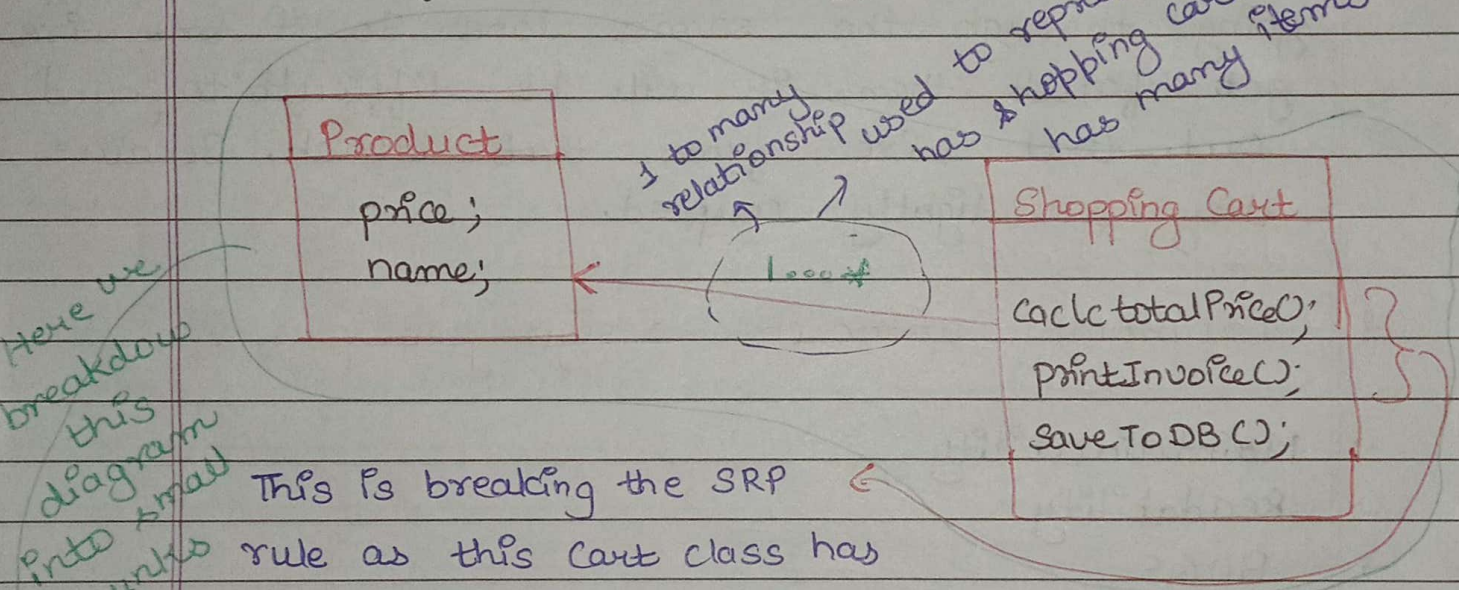
* Introduction To SOLID Design Principles

Robert C. Martin introduces these principles in 2000 papers

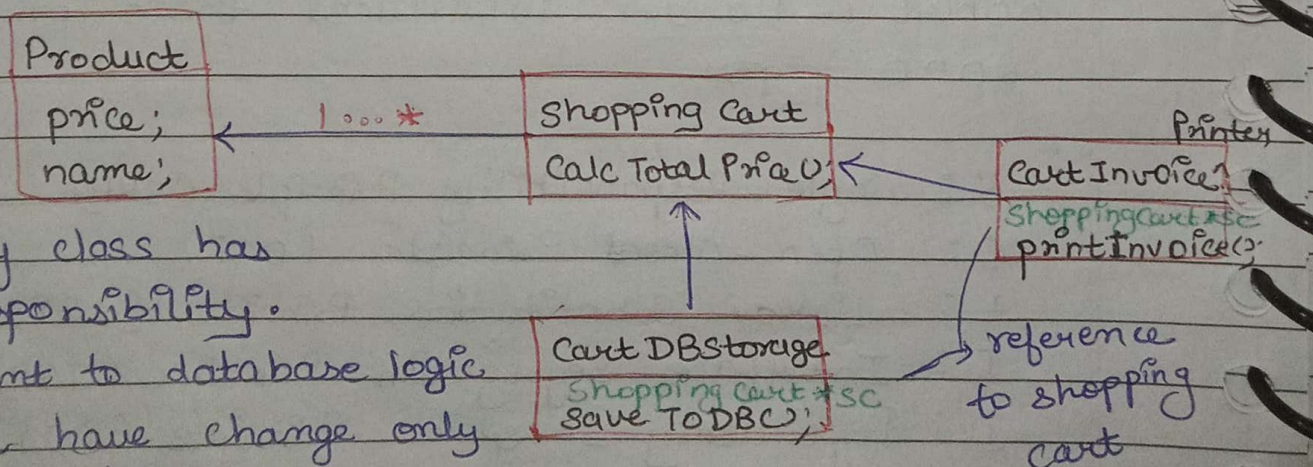
S: Single Responsibility Principle (SRP)
O: Open-Closed Principle (OCP)
L: LisKov Substitution Principle (LSP)
I: Interface Segregation Principle (ISP)
D: Dependency Inversion Principle (DIP)

* S : Single Responsibility Principle
  - A class should have only one reason to change
  - A class should do only one thing

Example: A T.V Remote all buttons are used to control TV. If it control fridge, fan then it will be complicated and ~~introduction~~ to integrate all features is difficult



This is breaking the SRP rule as this Cart class has to maintain three methods. If we want to change any one of this then we have to make changes in the same class.



Now every class has single responsibility.
If we want to database logic then we have change only in one class

```cpp
// SRP_Violated.cpp

#include <iostream>
#include <vector>

using namespace std;

// Product class representing any item of any ECommerce.
class Product {
public:
    string name;
    double price;

    Product(string name, double price) {
        this->name = name;
        this->price = price;
    }
};

// Violating SRP: ShoppingCart is handling multiple responsibilities
class ShoppingCart {
private:
    vector<Product*> products;
```

```
public:
    void addProduct(Product* p) {
        products.push_back(p);
    }

    const vector<Product*>& getProducts() {
        return products;
    }

    // 1. Calculates total price in cart.
    double calculateTotal() {
        double total = 0;
        for (auto p : products) {
            total += p->price;
        }
        return total;
    }

    // 2. Violating SRP - Prints invoice (Should be in a separate class)
    void printInvoice() {
        cout << "Shopping Cart Invoice:\n";
        for (auto p : products) {
            cout << p->name << " - Rs " << p->price << endl;
        }
        cout << "Total: Rs " << calculateTotal() << endl;
    }
```

```
            cout << p->name << " - Rs " << p->price << endl;
        }
        cout << "Total: Rs " << calculateTotal() << endl;
    }

    // 3. Violating SRP - Saves to DB (Should be in a separate class)
    void saveToDatabase() {
        cout << "Saving shopping cart to database..." << endl;
    }
};

int main() {
    ShoppingCart* cart = new ShoppingCart();

    cart->addProduct(new Product("Laptop", 50000));
    cart->addProduct(new Product("Mouse", 2000));

    cart->printInvoice();
    cart->saveToDatabase();

    return 0;
}
```

```cpp
// SRP_Followed.cpp

#include <iostream>
#include <vector>

using namespace std;

// Product class representing any item in eCommerce.
class Product {
public:
    string name;
    double price;

    Product(string name, double price) {
        this->name = name;
        this->price = price;
    }
};

//1. ShoppingCart: Only responsible for Cart related business logic.
class ShoppingCart {
private:
    vector<Product*> products; // Store heap-allocated products

public:
    void addProduct(Product* p) {
        products.push_back(p);
    }

    const vector<Product*>& getProducts() {
        return products;
    }
```

```
    //Calculates total price in cart.
    double calculateTotal() {
        double total = 0;
        for (auto p : products) {
            total += p->price;
        }
        return total;
    }
};

// 2. ShoppingCartPrinter: Only responsible for printing invoices
class ShoppingCartPrinter {
private:
    ShoppingCart* cart;

public:
    ShoppingCartPrinter(ShoppingCart* cart) {
        this->cart = cart;
    }

    void printInvoice() {
        cout << "Shopping Cart Invoice:\n";
        for (auto p : cart->getProducts()) {
            cout << p->name << " - Rs " << p->price << endl;
        }
        cout << "Total: Rs " << cart->calculateTotal() << endl;
    }
};

// 3. ShoppingCartStorage: Only responsible for saving cart to DB
class ShoppingCartStorage {
private:
    ShoppingCart* cart;

public:
    ShoppingCartStorage(ShoppingCart* cart) {
        this->cart = cart;
    }
```

```
72
73      void saveToDatabase() {
74          cout << "Saving shopping cart to database..." << endl;
75      }
76  };
77
78  int main() {
79      ShoppingCart* cart = new ShoppingCart();
80
81      cart->addProduct(new Product("Laptop", 50000));
82      cart->addProduct(new Product("Mouse", 2000));
83
84      ShoppingCartPrinter* printer = new ShoppingCartPrinter(cart);
85      printer->printInvoice();
86
87      ShoppingCartStorage* db = new ShoppingCartStorage(cart);
88      db->saveToDatabase();
89
90      return 0;
91  }
92
```

Conclusion - We might think that every class should have one Method. That is not the case.

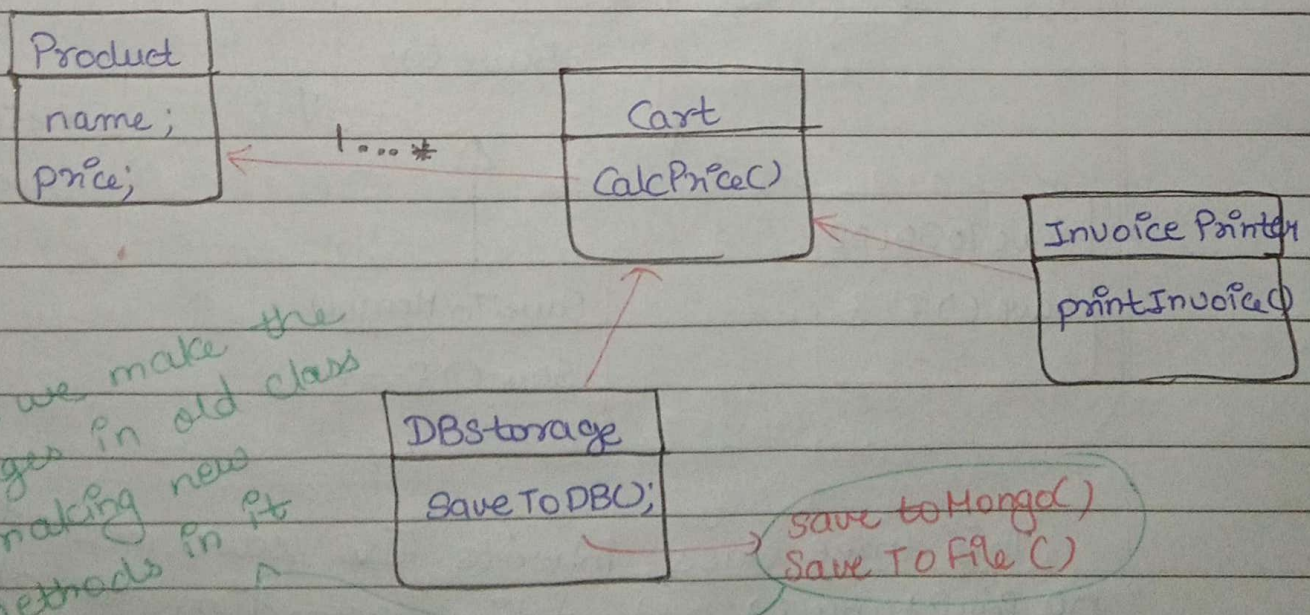
→ like in Shopping Cart, method is CalcTotalPrice() if we need some other methods like helper function then we can make it.

→ But all the methods should do only one work

* O: Open - Close Principle.
new feature addition
• A class should be open for extension but close for modification.
↳ changes in old code

Example: If we add new features then we should not changed the old code means — we should not create any new method in old class.

This feature is strict, might be idealistic, followed but we should try to follow them.



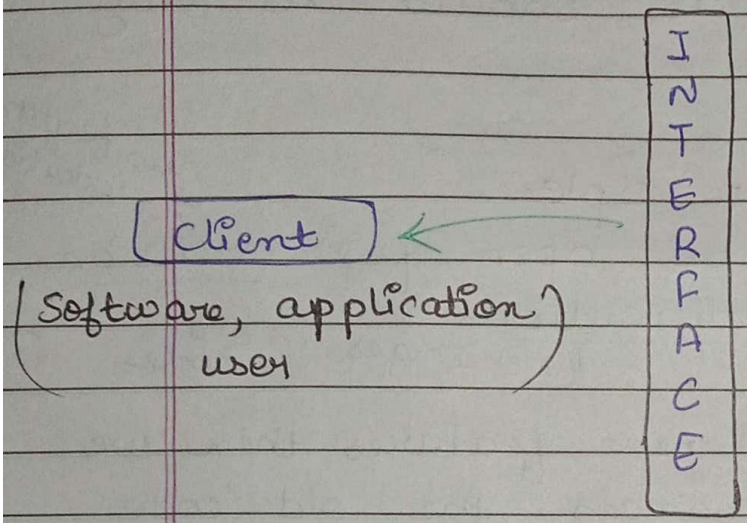
```cpp
// OCP_Violated.cpp

#include <iostream>
#include <vector>

using namespace std;

// Product class representing any item in eCommerce.
class Product {
public:
    string name;
    double price;

    Product(string name, double price) {
        this->name = name;
        this->price = price;
    }
};

//1. ShoppingCart: Only responsible for Cart related business logic.
class ShoppingCart {
private:
    vector<Product*> products;

public:
    void addProduct(Product* p) {
        products.push_back(p);
    }

    const vector<Product*>& getProducts() {
        return products;
    }

```

```
    //Calculates total price in cart.
    double calculateTotal() {
        double total = 0;
        for (auto p : products) {
            total += p->price;
        }
        return total;
    }
};

// 2. ShoppingCartPrinter: Only responsible for printing invoices
class ShoppingCartPrinter {
private:
    ShoppingCart* cart;

public:
    ShoppingCartPrinter(ShoppingCart* cart) {
        this->cart = cart;
    }

    void printInvoice() {
        cout << "Shopping Cart Invoice:\n";
        for (auto p : cart->getProducts()) {
            cout << p->name << " - Rs " << p->price << endl;
        }
        cout << "Total: Rs " << cart->calculateTotal() << endl;
    }
};

// 3. ShoppingCartStorage: Only responsible for saving cart to DB
class ShoppingCartStorage {
private:
    ShoppingCart* cart;

public:
    ShoppingCartStorage(ShoppingCart* cart) {
        this->cart = cart;
    }
```

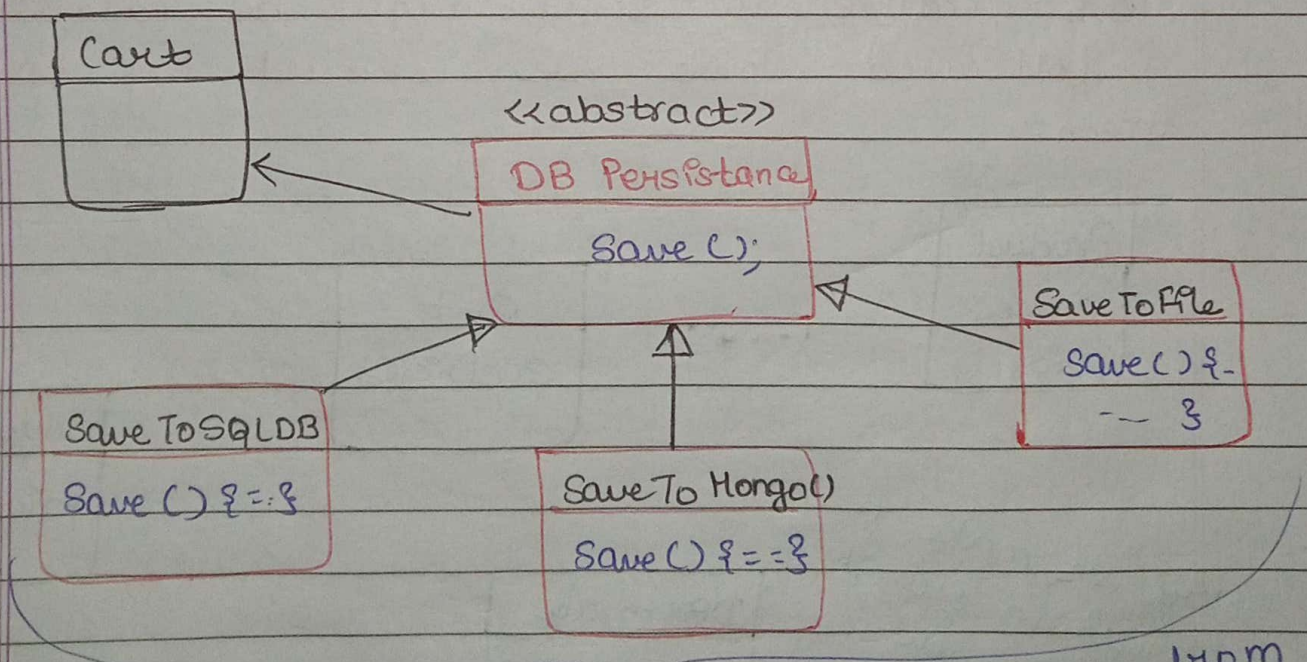
```
    void saveToSQLDatabase() {
        cout << "Saving shopping cart to SQL DB..." << endl;
    }

    void saveToMongoDatabase() {
        cout << "Saving shopping cart to Mongo DB..." << endl;
    }

    void saveToFile() {
        cout << "Saving shopping cart to File..." << endl;
    }
};

int main() {
    ShoppingCart* cart = new ShoppingCart();

    cart->addProduct(new Product("Laptop", 50000));
    cart->addProduct(new Product("Mouse", 2000));

    ShoppingCartPrinter* printer = new ShoppingCartPrinter(cart);
    printer->printInvoice();

    ShoppingCartStorage* db = new ShoppingCartStorage(cart);
    db->saveToSQLDatabase();

    return 0;
}
```

We will follow open - close principle by implementing Abstraction, Inheritance, Polymorphism.

We make DB as abstract class and other methods as concerete class



This interface tell the client what the class do without telling how it will do. Basically known as Abstraction



This three class inherits the save method from DB Persistance() and define according to their need. We can use method overriding

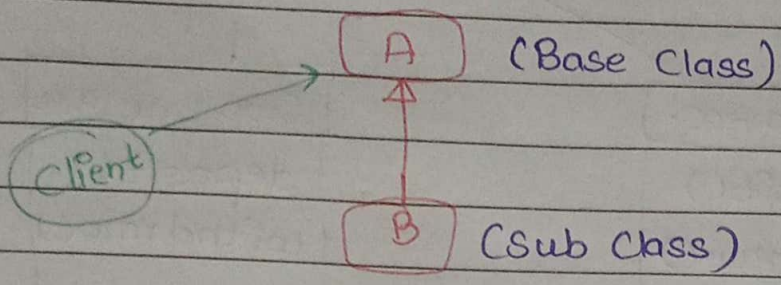
```cpp
// OCP_Followed.cpp

#include <iostream>
#include <vector>

using namespace std;

// Product class representing any item in eCommerce.
class Product {
public:
    string name;
    double price;

    Product(string name, double price) {
        this->name = name;
        this->price = price;
    }
};

//1. ShoppingCart: Only responsible for Cart related business logic.
class ShoppingCart {
private:
    vector<Product*> products; // Store heap-allocated products

public:
    void addProduct(Product* p) {
        products.push_back(p);
    }

    const vector<Product*>& getProducts() {
        return products;
    }
```

```
    //Calculates total price in cart.
    double calculateTotal() {
        double total = 0;
        for (auto p : products) {
            total += p->price;
        }
        return total;
    }
};

// 2. ShoppingCartPrinter: Only responsible for printing invoices
class ShoppingCartPrinter {
private:
    ShoppingCart* cart;

public:
    ShoppingCartPrinter(ShoppingCart* cart) {
        this->cart = cart;
    }

    void printInvoice() {
        cout << "Shopping Cart Invoice:\n";
        for (auto p : cart->getProducts()) {
            cout << p->name << " - Rs " << p->price << endl;
        }
        cout << "Total: Rs " << cart->calculateTotal() << endl;
    }
};

//Abstract class
class Persistence {
private:
    ShoppingCart* cart;

public:
    virtual void save(ShoppingCart* cart) = 0; // Pure virtual function
};

class SQLPersistence : public Persistence {
public:
    void save(ShoppingCart* cart) override {
        cout << "Saving shopping cart to SQL DB..." << endl;
    }
};
```

```
};

class MongoPersistence : public Persistence {
public:
    void save(ShoppingCart* cart) override {
        cout << "Saving shopping cart to MongoDB..." << endl;
    }
};

class FilePersistence : public Persistence {
public:
    void save(ShoppingCart* cart) override {
        cout << "Saving shopping cart to a file..." << endl;
    }
};

int main() {
    ShoppingCart* cart = new ShoppingCart();
    cart->addProduct(new Product("Laptop", 50000));
    cart->addProduct(new Product("Mouse", 2000));

    ShoppingCartPrinter* printer = new ShoppingCartPrinter(cart);
    printer->printInvoice();

    Persistence* db = new SQLPersistence();
    Persistence* mongo = new MongoPersistence();
    Persistence* file = new FilePersistence();

    db->save(cart);   // Save to SQL database
    mongo->save(cart); // Save to MongoDB
    file->save(cart);  // Save to File

    return 0;
}
```

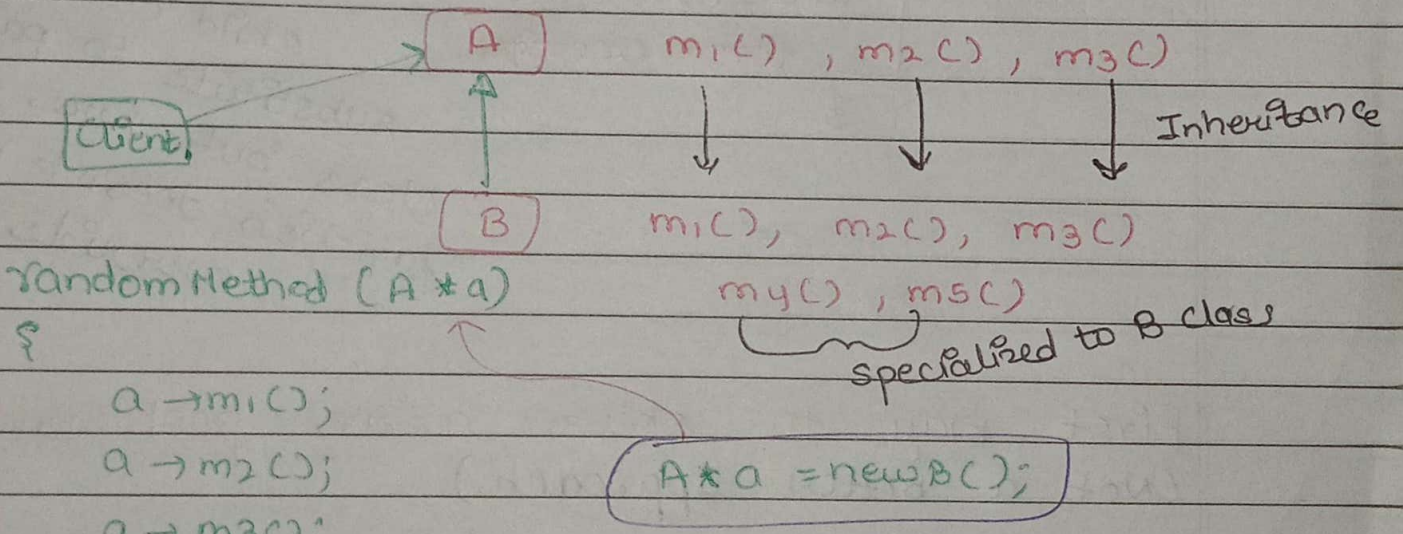
* L : Liskov Substitution Principle

Subclasses should be substituable for their base classes.



A client which is expecting method from class A. If we pass that method from B class then code should not break

Sub Class always extend methods from base class

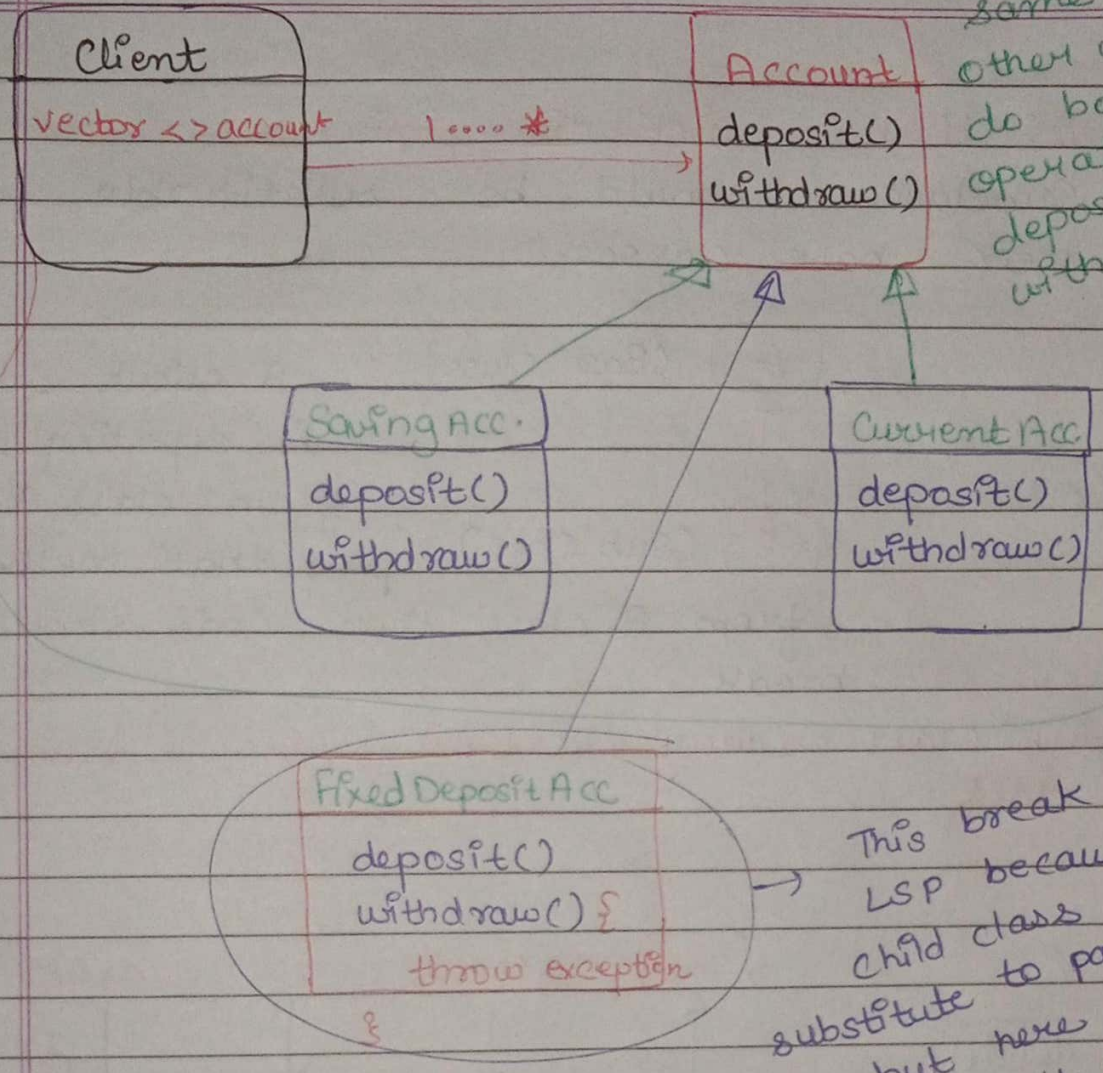


If client is talking to A. Client expect reference from A. Client can call method m1, m2, m3.

LSP says if we pass sub class B in reference then client can call method m1, m2, m3 because they are inherited.

This is simple principle but it break easily

Here all are working but after introducing fixed deposit Acc it breaking because Client assume that fixed Account is same as other we can do both the operation deposit and withdraw



→ This break the LSP because child class substitute to parent class but here withdraw function doesn't perform override

First way to solve above problem.
(Not a better approach.)

→ We change the code of Client like
→ if acc is fixed deposit then only call deposit method else call all method.
→ But here we tightly coupled the client with other three acc but client should only talk to interface
→ Client should be aware of all accounts.
→ It breaking OCP by changing code in client

```cpp
// LSP_Violated.cpp

#include <iostream>
#include <vector>
using namespace std;

class Account {
public:
    virtual void deposit(double amount) = 0;
    virtual void withdraw(double amount) = 0;
};

class SavingAccount : public Account {
private:
    double balance;

public:
    SavingAccount() {
        balance = 0;
    }

    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount << " in Savings Account. New Balance: " << balance << endl;
    }

    void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            cout << "Withdrawn: " << amount << " from Savings Account. New Balance: " << balance << endl;
        } else {
            cout << "Insufficient funds in Savings Account!\n";
        }
    }
};
```

```
};

class CurrentAccount : public Account {
private:
    double balance;

public:
    CurrentAccount() {
        balance = 0;
    }

    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount << " in Current Account. New Balance: " << balance << endl;
    }

    void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            cout << "Withdrawn: " << amount << " from Current Account. New Balance: " << balance << endl;
        } else {
            cout << "Insufficient funds in Current Account!\n";
        }
    }
};

class FixedTermAccount : public Account {
private:
    double balance;

public:
    FixedTermAccount() {
        balance = 0;
    }

    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount << " in Fixed Term Account. New Balance: " << balance << endl;
    }

    void withdraw(double amount) {
        throw logic_error("Withdrawal not allowed in Fixed Term Account!");
    }
};
```

```cpp
class BankClient {
private:
    vector<Account*> accounts;

public:
    BankClient(vector<Account*> accounts) {
        this->accounts = accounts;
    }

    void processTransactions() {
        for (Account* acc : accounts) {
            acc->deposit(1000);  //All accounts allow deposits

            //Assuming all accounts support withdrawal (LSP Violation)
            try {
                acc->withdraw(500);
            } catch (const logic_error& e) {
                cout << "Exception: " << e.what() << endl;
            }
        }
    }
};

int main() {
    vector<Account*> accounts;
    accounts.push_back(new SavingAccount());
    accounts.push_back(new CurrentAccount());
    accounts.push_back(new FixedTermAccount());

    BankClient* client = new BankClient(accounts);
    client->processTransactions(); //  Throws exception when withdrawing from FixedTermAccount

    return 0;
}
```

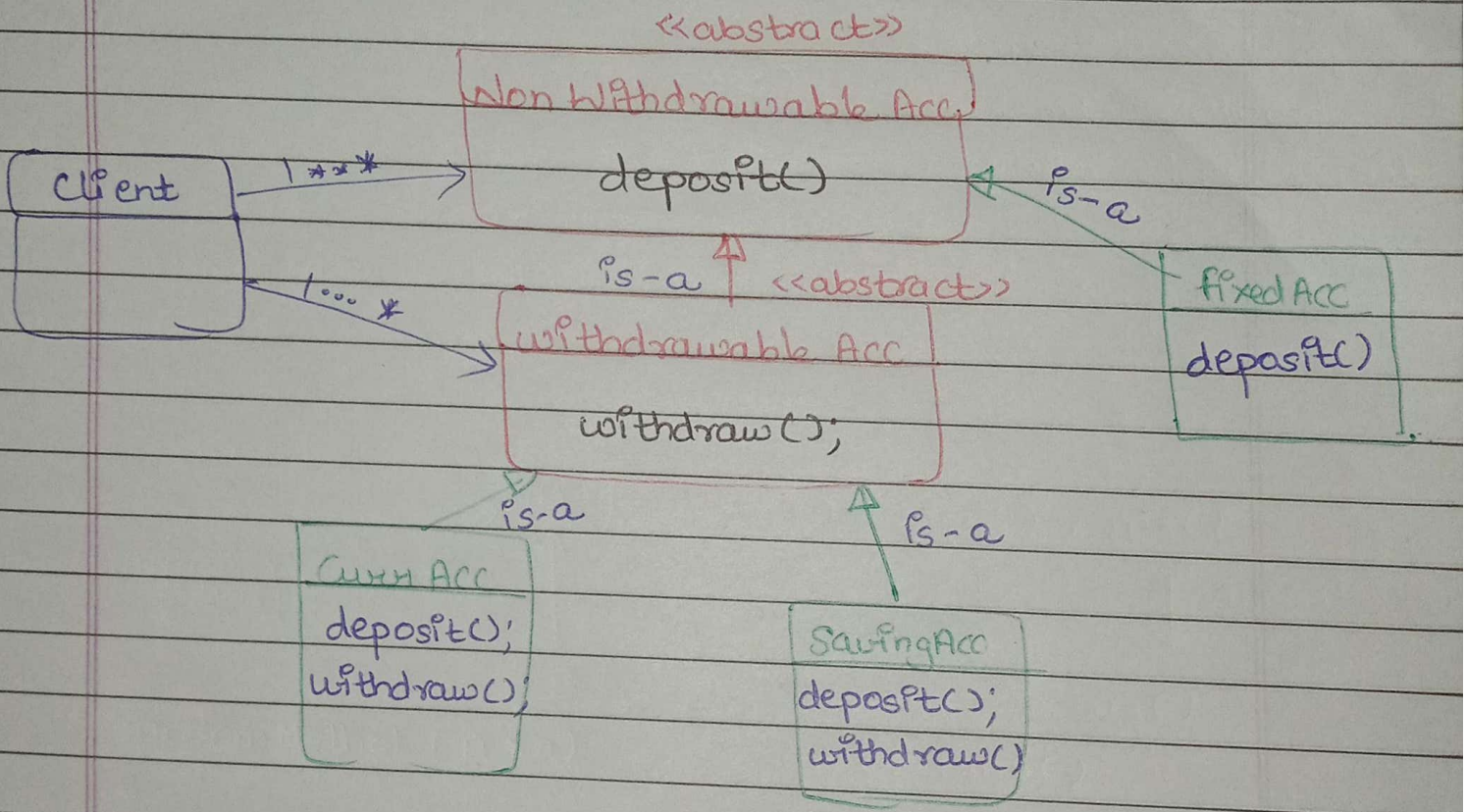
```cpp
// LSP_Followed_Wrongly.cpp

#include <iostream>
#include <vector>
#include <typeinfo>
#include <stdexcept>

using namespace std;

class Account {
public:
    virtual void deposit(double amount) = 0;
    virtual void withdraw(double amount) = 0;
};

class SavingAccount : public Account {
private:
    double balance;

public:
    SavingAccount() {
        balance = 0;
    }

    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount << " in Savings Account. New Balance: " << balance << endl;
    }

    void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            cout << "Withdrawn: " << amount << " from Savings Account. New Balance: " << balance << endl;
        } else {
            cout << "Insufficient funds in Savings Account!\n";
        }
    }
```

```cpp
class CurrentAccount : public Account {
private:
    double balance;

public:
    CurrentAccount() {
        balance = 0;
    }

    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount << " in Current Account. New Balance: " << balance << endl;
    }

    void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            cout << "Withdrawn: " << amount << " from Current Account. New Balance: " << balance << endl;
        } else {
            cout << "Insufficient funds in Current Account!\n";
        }
    }
};

class FixedTermAccount : public Account {
private:
    double balance;

public:
    FixedTermAccount() {
        balance = 0;
    }

    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount << " in Fixed Term Account. New Balance: " << balance << endl;
    }

    void withdraw(double amount) {
        throw logic_error("Withdrawal not allowed in Fixed Term Account!");
    }
};
```

```
    }
};

//Client class
class BankClient {
private:
    vector<Account*> accounts;

public:
    BankClient(vector<Account*> accounts) {
        this->accounts = accounts;
    }

    void processTransactions() {
        for (Account* acc : accounts) {
            acc->deposit(1000);

            //Checking account type explicitly
            if (typeid(*acc) == typeid(FixedTermAccount)) {
                cout << "Skipping withdrawal for Fixed Term Account.\n";
            } else {
                try {
                    acc->withdraw(500);
                } catch (const logic_error& e) {
                    cout << "Exception: " << e.what() << endl;
                }
            }
        }
    }
};

int main() {
    vector<Account*> accounts;
    accounts.push_back(new SavingAccount());
    accounts.push_back(new CurrentAccount());
    accounts.push_back(new FixedTermAccount());

    BankClient* client = new BankClient(accounts);
    client->processTransactions();

    return 0;
}
```

→ Second approach (Better way)

Note: FixeddepositAcc. should not the child of Account parent because it narrowing the methods of parent class. Child class should extend the method of parent class



→ We make two interface withdrawable and Non-withdrawable
→ Client will have both interfaces
→ Current and saving accounts call both method deposit() and withdraw().
→ fixed Acc. inherits only method from non withdrawable Account.

```cpp
// LSP_Followed.cpp

#include <iostream>
#include <vector>
#include <typeinfo>
#include <stdexcept>

using namespace std;


class DepositOnlyAccount {
public:
    virtual void deposit(double amount) = 0;
};

class WithdrawableAccount : public DepositOnlyAccount {
public:
    virtual void withdraw(double amount) = 0;
};

class SavingAccount : public WithdrawableAccount {
private:
    double balance;

public:
    SavingAccount() {
        balance = 0;
    }

    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount << " in Savings Account. New Balance: " << balance << endl;
    }

```

```
    void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            cout << "Withdrawn: " << amount << " from Savings Account. New Balance: " << balance << endl;
        } else {
            cout << "Insufficient funds in Savings Account!\n";
        }
    }
};

class CurrentAccount : public WithdrawableAccount {
    private:
        double balance;

    public:
        CurrentAccount() {
            balance = 0;
        }

        void deposit(double amount) {
            balance += amount;
            cout << "Deposited: " << amount << " in Current Account. New Balance: " << balance << endl;
        }

        void withdraw(double amount) {
            if (balance >= amount) {
                balance -= amount;
                cout << "Withdrawn: " << amount << " from Current Account. New Balance: " << balance << endl;
            } else {
                cout << "Insufficient funds in Current Account!\n";
            }
        }
    };
```

```
class FixedTermAccount : public DepositOnlyAccount {
private:
    double balance;

public:
    FixedTermAccount() {
        balance = 0;
    }

    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount << " in Fixed Term Account. New Balance: " << balance << endl;
    }
};

class BankClient {
private:
    vector<WithdrawableAccount*> withdrawableAccounts;
    vector<DepositOnlyAccount*> depositOnlyAccounts;

public:
    BankClient( vector<WithdrawableAccount*> withdrawableAccounts,
        vector<DepositOnlyAccount*> depositOnlyAccounts) {
        this->withdrawableAccounts = withdrawableAccounts;
        this->depositOnlyAccounts = depositOnlyAccounts;
    }

    void processTransactions() {
        for (WithdrawableAccount* acc : withdrawableAccounts) {
            acc->deposit(1000);
            acc->withdraw(500);
        }
        for (DepositOnlyAccount* acc : depositOnlyAccounts) {
            acc->deposit(5000);
        }
    }
};
```

```
            acc->deposit(5000);
        }
    }
};

int main() {
    vector<WithdrawableAccount*> withdrawableAccounts;
    withdrawableAccounts.push_back(new SavingAccount());
    withdrawableAccounts.push_back(new CurrentAccount());

    vector<DepositOnlyAccount*> depositOnlyAccounts;
    depositOnlyAccounts.push_back(new FixedTermAccount());

    BankClient* client = new BankClient (withdrawableAccounts, depositOnlyAccounts);
    client->processTransactions();

    return 0;
}
```