-Restricting access to certain attributes and/or methods is called **data hiding**.
-Combining the data and methods in the same entity is **encapsulation**
-**Scope:** private (inside class only), public (inside and outside class), protected (inside class and child classes)

Constructor is a physical piece of code that is used to construct and initialize objects. It is automatically invoked when a new object is created. There is no returned value, even a void. A class can have many constructors (overload). Name of the constructors must be the same as the
class name. If no constructor is implemented, the compiler will issue a default constructor.

Destructor: Invoked automatically, when the variable is removed from memory (e.g. goes out of scope). Each class can have at most one destructor. The destructor name is a name of a class preceded by a tilde sign (~). Destructor, the same as constructor, has no return type (even void). Destructor frees the resources used by the object (allocated memory, file descriptors, semaphores
etc.)

Copy constructor: Due to the bitwise copy of the default
constructor, it will cause a serious problem if the copying takes place when the object has a member pointer with a dynamic allocated memory.
Pointers of the source obj and the destination object will refer into the same memory.

```
Test::Test(const Test& src)
{
    iSize = src.iSize;
    ptr = new int [iSize];
    for (int i=0; i<iSize; ++i)
        ptr[i] = src.ptr[i];
}
```

Assignment operator: Clean up the allocated memory that the pointer member is pointing to before being allocated with a new memory. Remember to check for self-assignment

```
Test& Test::operator=(const Test& src)
{
    if (this != &src)
    {
        delete [] ptr;
        iSize = src.iSize;
        ptr = new int [iSize];
        for (int i=0; i<iSize; ++i)
            ptr[i] = src.ptr[i];
    }
    return *this;
}
```

Rule of three: If a class defines any of the following then it should explicitly define all three: (1) destructor, (2) copy constructor, (3) assignment operator
Shallow Copy: Perform bitwise copy from the source obj (S) to the destination obj (D), which means if S and D both have a pointer, they will point to the same memory space.
Deep Copy: Performing deep copy allocates new memory space for the copied data,
which means if S and D both have a pointer, they will point to different memory space.
Overloading:
Syntax:
<returned-type>
operator<op>(arguments)
E.g: bool FullName::**operator==**(const FullName& rhs)

Overloading insertion & extraction ops

```
ostream& operator<<(ostream&
os, const Fraction& x)
{
    os << abc;
    return os;
}
```

!!!Remember to add **friend** in the class for this situation.

Subscript operators

```
const A& operator[] (int index)
const;
A& operator[] (int index);
```

Prefix operator

```
Test& Test::operator++()
{
    *this += 1;
    return *this;
}
```

Postfix operator

```
Test Test::operator++(int)
{
    Test res = *this;
    *this += 1;
    return res;
}
```

friend:
With the keyword **friend**, you grant access to other functions or classes. Friend functions give a flexibility to the class. It doesn't violate the encapsulation of the class. Friendship is "directional". It means if class A considers class B as its friend, it doesn't mean that class B considers A as a friend.

Types of inheritance
public: public and protected of the base class become public and protected of the derived class.
protected: public and protected of the base class become protected of the derived class.
private: public and protected of the base class become private of the derived class

Constructors in inheritance
When a new object of a derived class is created:
-The constructor of the base class is invoked first.
-Then, the constructor of the derived class is invoked.
-In the constructor of the derived class, we can specify which constructor of the base class is
called. Otherwise, the default constructor of the base class will be invoked

```
class C1 {
public:
    C1() { cout
<< "C1 ctor;"; }
};
class C2 {
public:
    C2() { cout
<< "C2 ctor;"; }
};
class Base {
private:
    C1 c1;
public:
    Base() { cout
<< "Base ctor;";}
};
```

```
class Derived:
public Base {
private:
    C2 c2;
public:
    Derived():
Base() { cout <<
"Derived ctor;";}
};

int main() {
    Derived d;
}
```

The output will be:
C1 ctor;Base ctor;C2 ctor;Derived ctor
Destructor in inheritance
When an object of the derived class finishes its lifespan:
-The destructor of the derived class is invoked first.
-Then, the destructor of the base class is called later

Overloading – member methods of the same name but different parameters
Overriding – derived methods of the same name and parameters with the parent classes (same signature)

Assignment operator in inheritance
To implement the assignment operator for the derived class:
-Calling the assignment operator of the base class to assign data members of the base class part in the two objects first.
-Then, implement the assignment for data member of the derived class part.

```
X& X::operator=(const X& src) {
    if (this == &src) return *this;
    Y::operator=(src);
    //call the BASE
    //...
    return *this;
}
```

Virtual function: in the base class, member functions will become virtual functions if we add the virtual keyword in front of their declarations.
-For example: **virtual void** print();
-In the derived class, the definition of virtual member function can be re-define according to the requirement of that class.
-Note: it is allowed to have or not have the virtual keyword in front of the virtual functions of the derived class.

Dynamic binding
When using virtual functions, compiler will make sure which member function will be invoked according to which object is calling.
For example
print() is a virtual function

```
int main()
{
    A varA;
    B varB;
    C varC;
    A* var1, *var2;
    var1 = &varC;
    var2 = &varB;
    var1->print(); // print() of C
    var2->print(); // print() of B
}
```

Polymorphism is a core concept in object-oriented programming (OOP) that refers to the ability of different objects to respond to the same method or function call in different ways. It allows for flexibility and the reuse of code, making it more extensible and easier to maintain. There are two main types of polymorphism:

Compile-time Polymorphism (Static Binding):
This type of polymorphism is resolved during compile time. It includes:
Method Overloading: Multiple methods in the same class share the same name but have different parameters (different type, number, or both).
Operator Overloading: Specific operators can be overloaded to work with user-defined types (this is more common in languages like C++).

Runtime Polymorphism (Dynamic Binding):
This type of polymorphism is resolved during runtime. It includes:
Method Overriding: A subclass provides a specific implementation of a method that is already defined in its superclass.
Interfaces and Abstract Classes: Classes implement interfaces or extend abstract classes and provide the specific implementation of the methods.

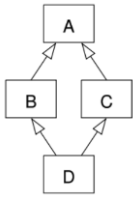
```
#include <iostream>
using namespace std;

class Animal {
public:
    virtual void sound() {
        cout << "Animal makes a sound" << endl;
    }
};

class Dog : public Animal {
public:
    void sound() override {
        cout << "Dog barks" << endl;
    }
};

int main() {
    Animal* myAnimal = new Dog();

    myAnimal->sound(); // Output: Dog barks

    delete myAnimal;
    return 0;
}
```

Why is Polymorphism Useful?
Flexibility: You can write code that works with different types of objects using a common interface.
Reusability: You can use the same method name for different types of actions, reducing code duplication.
Maintainability: It makes your code easier to manage and extend because you can add new types without changing existing code.

NO virtual constructor!
-Each constructor is used to initialize the class itself.
-Constructors are designed to run from Base to Derived classes.
Virtual destructor: YES!
The destructor should be virtual in order to free the memory/resource of the correct object.

Abstract class
-A pure virtual function that is not overriden in a derived class remains a pure virtual function, so the derived class is also an abstract class.
-An important use of abstract classes is to provide an interface without exposing any implementation details.
-Every class having at least one virtual function should have the virtual destructor

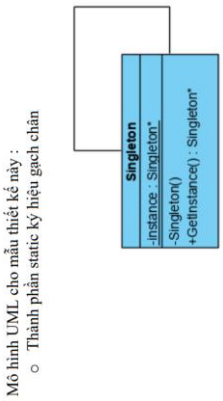
The Diamond Problem in OOP
The diamond problem is a common issue in object-oriented programming (OOP) that arises in multiple inheritance scenarios. It's called the diamond problem because the inheritance structure resembles a diamond shape.



Design Patterns:
In software development, a design pattern is a general and reusable solution to a commonly occurring problem.
-A design pattern can solve many problems by providing a framework for building an application.
-With design patterns, the design process is cleaner and more efficient.

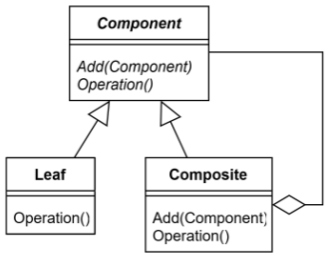
Singleton Pattern

```
class C {
private:
    static C* instance;
    C() {
        cout << "C created.\n";
    }
public:
    static C* getInstance() {
        if (instance == NULL) {
            instance = new C();
        }
        return instance;
    }
    static void delInstance() {
        if (instance) {
            delete instance;
            instance = NULL;
        }
    }
};
```

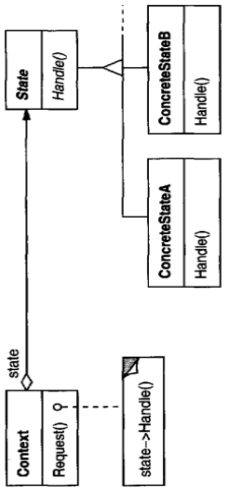


Mô hình UML cho mẫu thiết kế này :
- Thành phần static ký hiệu gạch chân

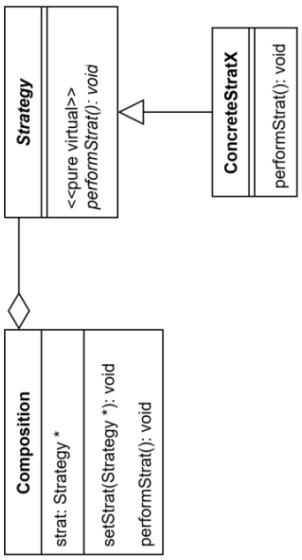
Composite Pattern:



State Pattern:



Strategy Pattern:



Code Composite:

```
class Circuit {
public:
    virtual double getResistance() = 0;
    virtual void addCircuit(Circuit* circuit) = 0;
};

class SingleCircuit : public Circuit {
private:
    double value;
public:
    SingleCircuit(double val) : value(val) {}
    double getResistance();
    virtual void addCircuit(Circuit* circuit){
    }
};

class SeriesCircuit : public Circuit {
private:
    vector<Circuit*> subCircuits;
public:
    void addCircuit(Circuit* circuit);
    double getResistance();
};

class ParallelCircuit : public Circuit {
private:
    vector<Circuit*> subCircuits;
public:
    void addCircuit(Circuit* circuit);
    double getResistance();
};
double SingleCircuit::getResistance(){
    return value;
}

void SeriesCircuit::addCircuit(Circuit* circuit) {
    subCircuits.push_back(circuit);
}
double SeriesCircuit::getResistance(){
    double totalResistance = 0.0;
    for (const auto& circuit : subCircuits) {
        totalResistance += circuit->getResistance();
    }
    return totalResistance;
}


void ParallelCircuit::addCircuit(Circuit* circuit) {
    subCircuits.push_back(circuit);
}
double ParallelCircuit::getResistance(){
    double inverseTotalResistance = 0.0;
    for (const auto& circuit : subCircuits) {
        inverseTotalResistance += 1.0 / circuit->getResistance();
    }
    return 1 / inverseTotalResistance;
}
```

Main:

```
Circuit* AB = new SeriesCircuit;
Circuit* R1 = new SingleCircuit(5.0);
Circuit* R2 = new SingleCircuit(6.0);

Circuit* pCircuit = new ParallelCircuit;
pCircuit->addCircuit(R1);
pCircuit->addCircuit(R2);

AB->addCircuit(R1);
AB->addCircuit(pCircuit);
AB->addCircuit(R2);
```

Code State Pattern

```
class Product {
    public:

    string ID;
    string Name;
    double price;
    int stockQuantity;

    Product(const string& id, const string& name, double cost, int quantity)
    : ID(id), Name(name), price(cost), stockQuantity(quantity) {}
};

class Payment {
    public:
    virtual void pay(double amount) = 0;
};

class Cash : public Payment {
    public:
        void pay(double amount);
};

class ATM : public Payment {
    public:
        void pay(double amount);
};

class ZaloPay: public Payment {
    public:
        void pay(double amount);
};

class Momo : public Payment {
    public:
        void pay(double payment);
};

class Order {
   private:
       string OrderId;
       string cusName;
       string cusPhone;
       string addr;
       vector<Product> productList;
       double totalPrice;
       Payment* method;

   public:
       Order(const string& id, const string& name, const string& phone, const string& add , Payment* PaymentMethod)
       : OrderId(id), cusName(name), cusPhone(phone), addr(add), method(PaymentMethod), totalPrice(0.0) {}
       void addProduct(const Product& product);
       void pay();
};
void Cash::pay(double amount) {
    cout << "Da thanh toan " << amount << "d bang tien mat" << endl;
}

void ATM::pay(double amount) {
    cout << "Da thanh toan " << amount << "d bang ATM" << endl;
}

void ZaloPay::pay(double amount) {
    cout << "Da thanh toan " << amount <<"d bang ZaloPay" << endl;
}

void Momo::pay(double amount) {
    cout << "Da thanh toan " << amount << "d bang Momo" << endl;
}


void Order::addProduct(const Product& product) {
    productList.push_back(product);
    totalPrice += product.price;
}

void Order::pay(){
    method->pay(totalPrice);
}
```

Main:

```
Product p1("001", "Laptop", 17000000, 1);
Product p2("002", "Mobile Phone", 36000000, 1);

int choice;
cout << "1.Cash\n2.Momo\n3.ZaloPay\n4.ATM\n";
cout << "Choose the method for payment (1-4): ";
cin >> choice;
Payment* method = nullptr;
switch (choice) {
    case 1: {
        method = new Cash();
        break;
    }
    case 2: {
        method = new Momo();
        break;
    }
    case 3: {
        method = new ZaloPay();
        break;
    }
    case 4: {
        method = new ATM();
        break;
    }
}
Order order("ORD001", "NGUYEN VAN A", "0123456789", "227 NVC Q5", method);
order.addProduct(p1);
order.addProduct(p2);

order.pay();

delete method;
```