1. Quan hệ giữa các phương thức:

- Shape::Description() và Circle::Description()
  - Đây là ghi đè (overriding) hàm, Circle::Description() override lại hàm Shape::Description() của lớp cha vì Description ở lớp Circle khác với lớp Shape nên ta không thể dùng lại phương thức của lớp cha hơn nữa đây lại là hàm ảo nên việc override này sẽ cho phép đối tượng Circle được quản lý bằng biến kiểu Shape * nhưng vẫn dùng phương thức Description của lớp Circle
- Ellipse::Scale(float) và Ellipse::Scale(float, float)

  - Đây là nạp chồng, quá tải (overload) hàm, cùng một tên hàm nhưng lại có tham số truyền vào khác nhau thường overload hàm sẽ giúp ta có nhiều cách để thực hiện cùng một công việc cho nhiều kiểu tham số khác nhau. Như bài này Scale(float) chỉ có 1 tham số vào và sẽ thay đổi theo cả phương cùng một tỉ lệ, còn Phương thức Scale(float, float) sẽ có thể thay đổi trục chính và trục phụ theo 2 tỉ lệ riêng biệt.
- Shape::InterfaceType() và Circle::InterfaceType()
  - Đây cũng là ghi đè hàm (overriding). Nhưng lần này 2 phương thức này không phải là hàm ảo nên biến có kiểu nào sẽ gọi đúng phương thức ở lớp đó được xác định lúc dịch chương trình.
- Circle::InterfaceType() và Ellipse::InterfaceType()
  - Không có quan hệ gì

2. Định nghĩa object và class

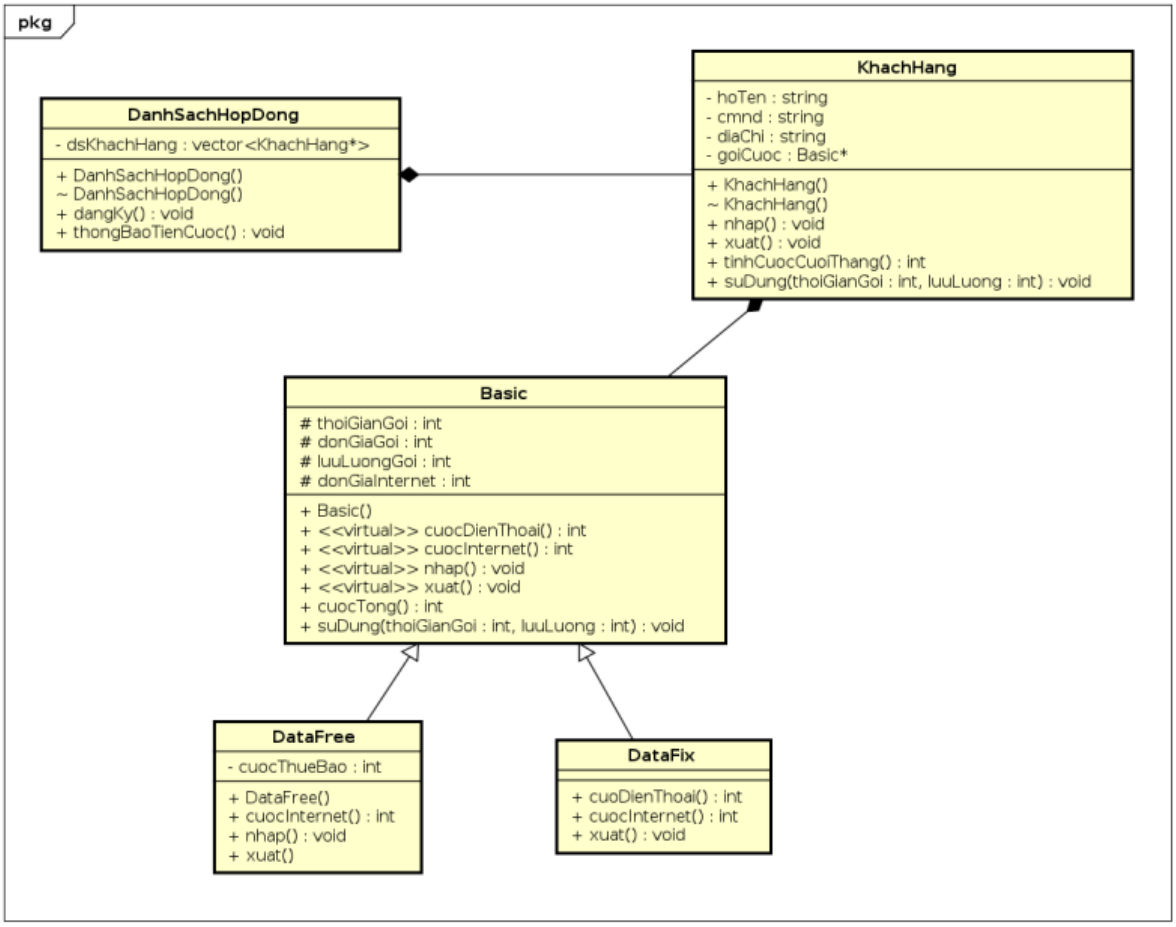
C.

Đối tượng là thể hiện của lớp. Một lớp có thể có nhiều đối tượng khác nhau. Mỗi đối tượng đều của 1 lớp sẽ có nhưng phương thức và thuộc tính mà lớp đó quy định nhưng giá trị của thuộc tính có thể khác nhau.

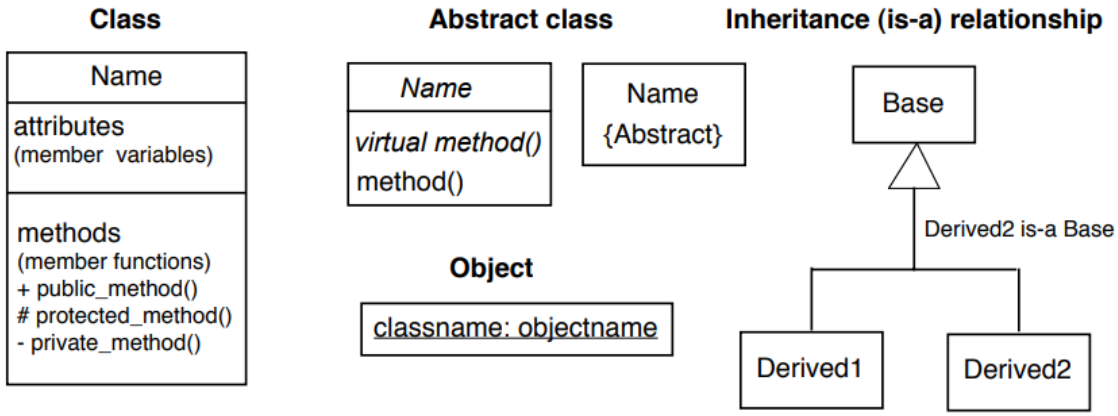
Lớp: Bike, EBike

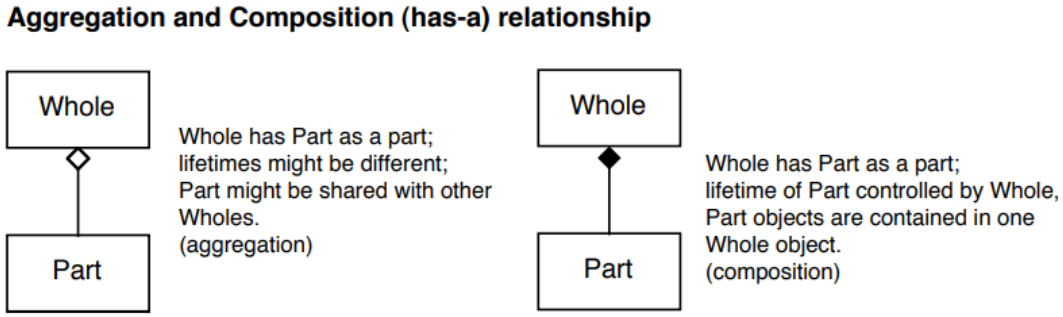
Đối tượng được lưu trong các biến b1, b2.

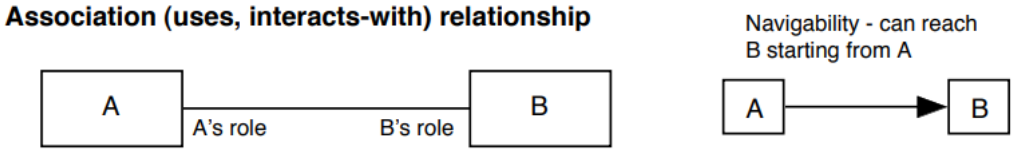
3. Ví dụ Sơ đồ

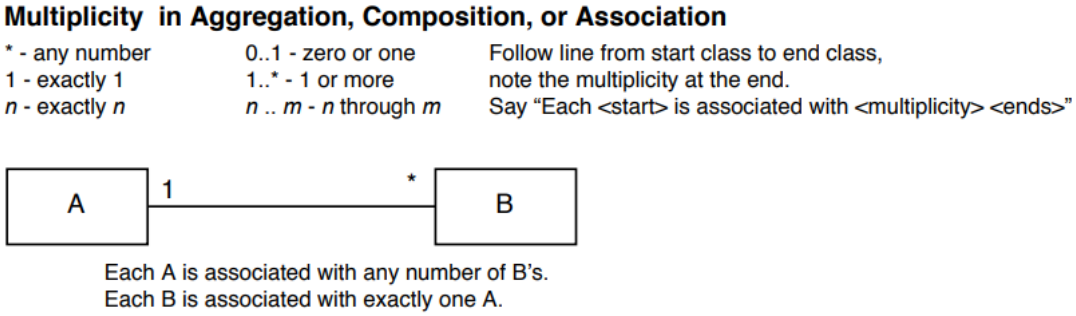


4. Các kí hiệu trong UML









5. Ví dụ ghi đè toán tử

một tham số giả để phân biệt toán tử ++ dạng hậu tố so với toán tử ++ dạng tiền tố.

Mã nguồn C++

```
1   #include "PhanSo.h"
2   PhanSo::PhanSo( ) {
3         tu=0;  mau=1;
4   }
5   PhanSo::PhanSo(int t, int m)  {
6         tu=t;  mau=m;
7   }
8
9   PhanSo& PhanSo::operator+=(const PhanSo& src)  {
10        tu = tu*src.mau + src.tu*mau;
11        mau = mau*src.mau;
12        return *this;
13  }
14  const PhanSo PhanSo::operator+(const PhanSo& src) const {
15        PhanSo temp;
16        temp.tu = tu*src.mau + src.tu*mau;
17        temp.mau = mau*src.mau;
18        return temp;
19        //toán tử có thể được cài đặt cách khác, dựa theo +=
20  }
21
22  bool PhanSo::operator==(const PhanSo& src) const {
23        //giả sử sau mỗi lần tính toán, phân số đã được rút gọn
24        return ( (tu==src.tu) && (mau==src.mau) );
25  }
26
27  PhanSo& PhanSo::operator++( )          // toán tử ++ dạng tiền tố
28  {
29        tu = tu + mau;
30        return *this;
31  }
32
33  PhanSo PhanSo::operator++(int)         // toán tử ++ dạng hậu tố
34  {
35        PhanSo temp = *this;
36        ++(*this);
37        return temp;
38  }
39
40  ostream& operator<<(ostream &out, const PhanSo& src) {
41        out << src.tu << "/" << src.mau;
42        return out;
43  }
```

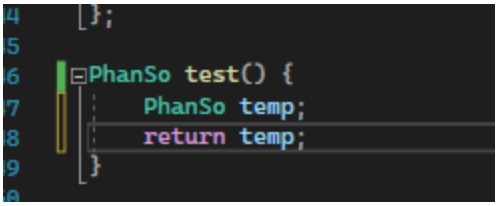
Bảng III-15. Mã nguồn cài đặt các toán tử của lớp PhanSo

6. **Destructor của cha luôn có virtual.**
7. **Quy luật 3 ông lớn**

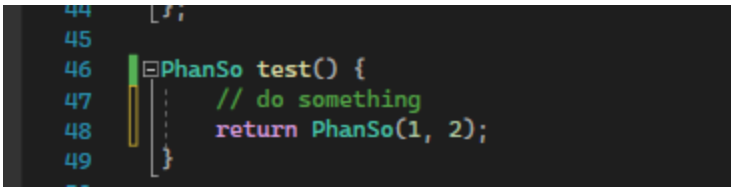
Bản chất của truyền object, return object.

Return object

Nếu bạn trả về một object của class thì có thể xảy ra chuyện gì nếu trong temp có chứa những attribute con trỏ. → đó là có thể gây lỗi bộ nhớ. Để tránh xảy ra điều này, ta cần phải cài đặt copy constructor. Bởi khi return temp thì nơi gọi hàm → nó sẽ tạo object mới bằng copy constructor với tham số là đối tượng temp.



Nếu bạn không muốn sử dụng copy constructor, bạn có thể chỉ định một constructor khác bằng cách:

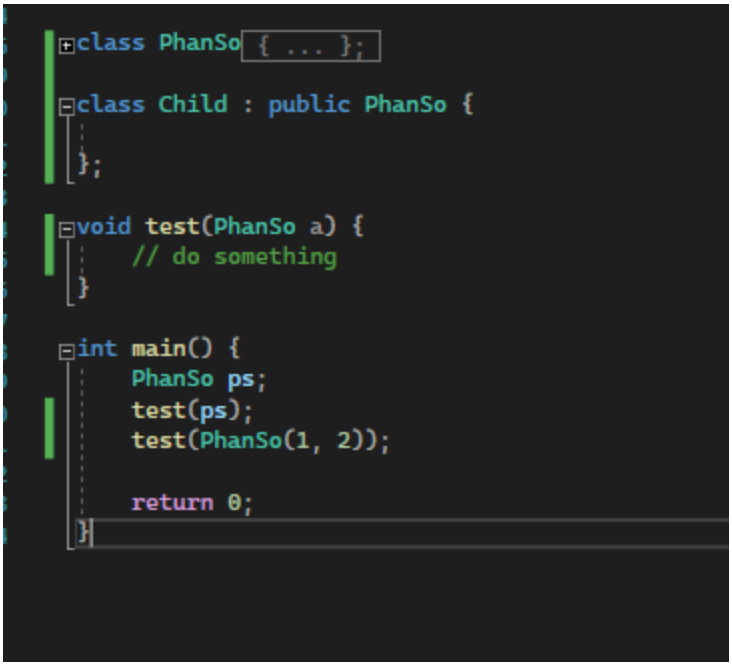


Ở đây, PhanSo(1,2) là chỉ định constructor sử dụng chứ nó ko phải tạo object ngay bên trong test.

Truyền object

Tương tự như Return, nếu truyền vào 1 object ⇒ sử dụng copy constructor; còn truyền vào PhanSo(1,2) → chỉ định constructor sử dụng để tạo object cho biến tham số a

(Không phải tạo object ở PhaSo(1,2) ở hàm main)



Nếu sử dụng tham chiếu thực hiện tạo một tên biến mới cho biến truyền vào nhưng chúng đều thao tác trên cùng 1 vùng nhớ (hay nói cách khác là chỉ định vùng nhớ mà biến tương tác) → nó sẽ ko thực hiện tạo object mới → nên không sử dụng constructor.

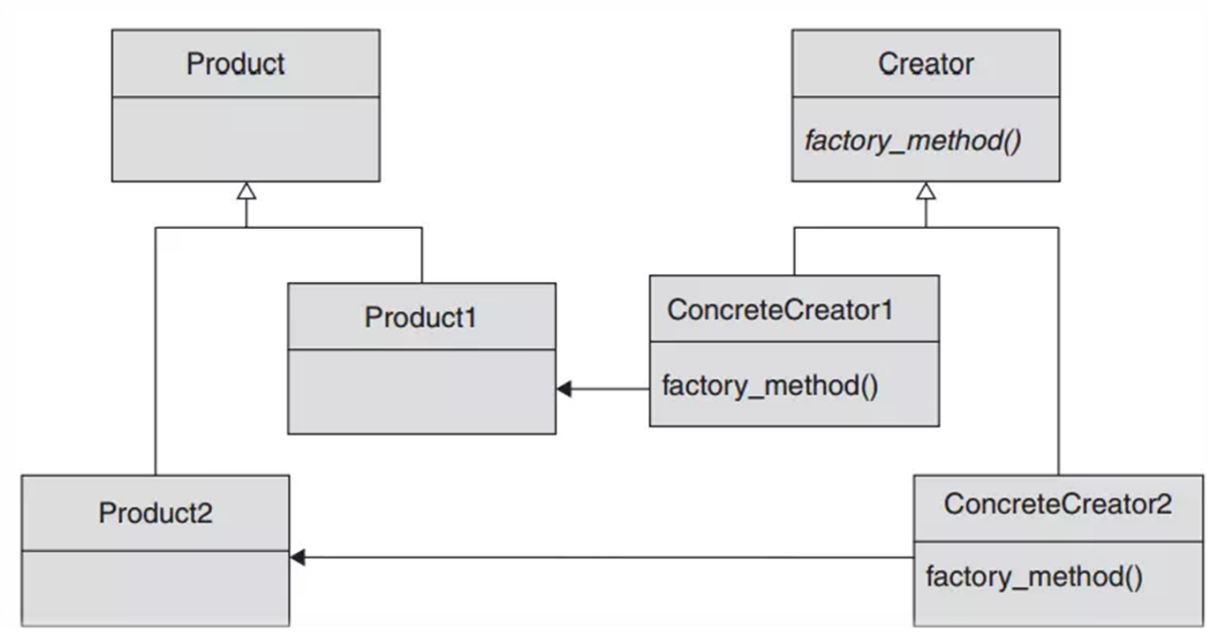
Factory

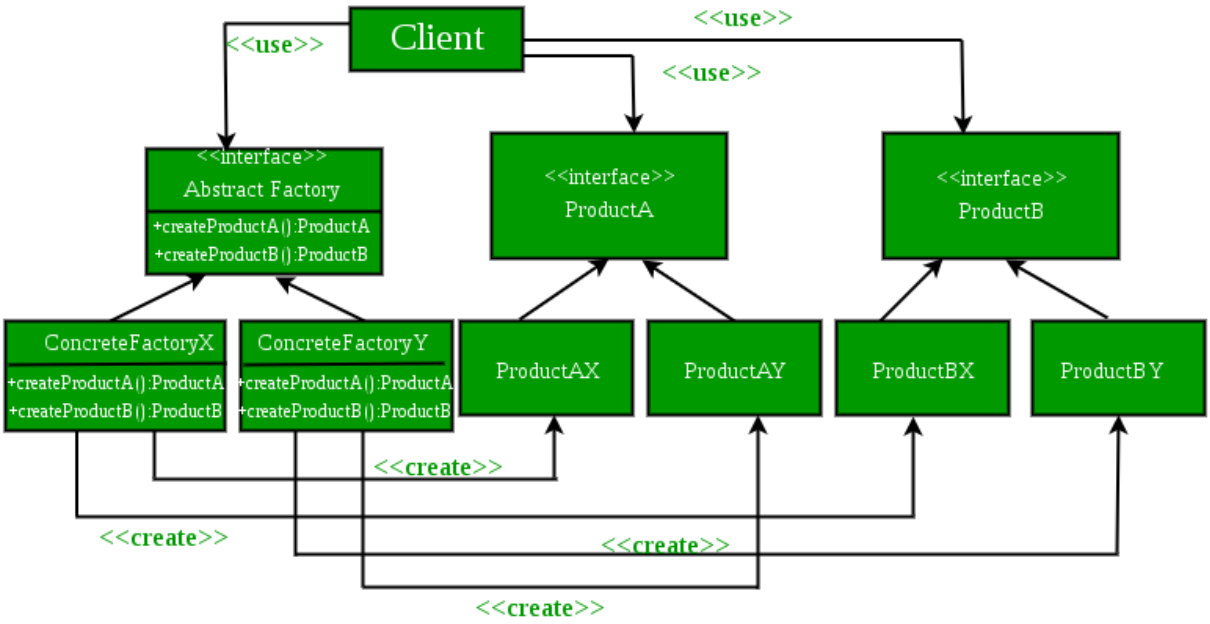
Định nghĩa

Là creational pattern

Dùng để tạo lập các đối tượng qua interface hoặc abstract class mà không cần chỉ chính xác lớp nào sẽ tạo.

Có 2 loại: Factory Method Design Pattern(ưu điểm: giải quyết vấn đề mở rộng, giấu logic; nhược điểm: chỉ tạo được object từ 1 class), Abstract Factory Metho Design Pattern (ưu điểm: mở rộng + gom nhóm, giấu logic code; nhược điểm: ko thể thêm số lượng class để tạo object từ class mới)





Cách sử dụng

Chủ chốt: thường sử dụng để tạo object ngay vào lúc chương trình đang chạy.

Liệt kê:

- Giấu logic khởi tạo khỏi class sử dụng
- Tạo object dựa trên dữ liệu, đầu vào đã cung cấp.
- Tạo object cùng loại nhưng khác thuộc tính.

Code

Simple Factory Method

```
#include <iostream>
#include <string>
#include <exception>

using namespace std;

class Animal{
public:
  virtual void talk() = 0;
};
```

```
class Dog:Animal {
  void talk() {
    cout << "Bark bark!!!" << endl;
  }
};

class Cat:Animal {
  void talk() {
    cout << "Meow meow!!!" << endl;
  }
};

class Factory {
public:
  static Animal* factoryMethod(string typeAnimal) {
    if (typeAnimal == "dog") return (Animal*)(new Dog());
    if (typeAnimal == "cat") return (Animal*)(new Cat());
    return nullptr;
  }
};


int main()
{
  try {
    string types[] = { "dog", "cat" };
    for (auto i : types) {
      Animal* a = Factory::factoryMethod(i);
      if (a) {
        a->talk();
        delete a;
        a = nullptr;
      }
      else {
        cout << "Invalid type" << endl;
      }
    }
  }
  catch (exception& ex) {
    cout << ex.what();
  }
  system("PAUSE");
  return 0;
}
```

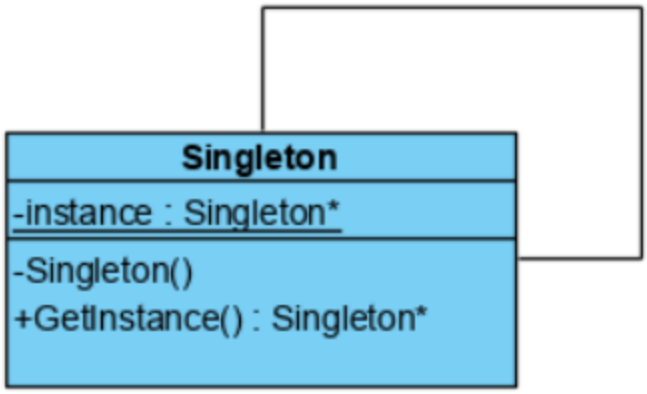
Factory Method

Abstract Factory Method

Singleton

Định nghĩa

- Singleton là mẫu thiết kế trong lập trình hướng đối tượng thuộc vào nhóm khởi tạo (Creational)
- Mục tiêu:
  - Mỗi lớp đảm bảo chỉ có một thể hiện duy nhất trong suốt chương trình chạy
  - Cung cấp cách thức truy xuất toàn cục



Cách sử dụng

- Sử dụng với những class chỉ cần tạo một object duy nhất trong chương trình
- Cung cấp khả năng truy xuất toàn cục

Code

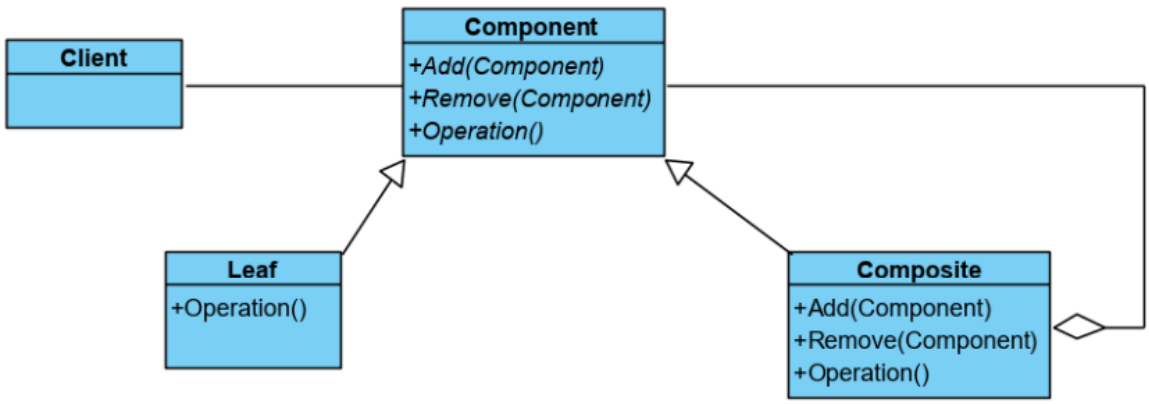
```
#include <iostream>

using namespace std;

class Singleton {
private:
  static Singleton* pInstance;
  Singleton() {}
public:
  ~Singleton() {
    delete pInstance;
    pInstance = nullptr;
  }

  static Singleton* getInstance() {
    if (!pInstance) {
      pInstance = new Singleton();
    }

    return pInstance;
  }

  static void deleteInstance() {
    if (pInstance) {
      delete pInstance;
    }

    pInstance = nullptr;
  }

  void display() {
    cout << "Hello" << endl;
  }
};

Singleton* Singleton::pInstance = nullptr;

int main()
{
  Singleton* object = Singleton::getInstance();
  object->display();
  Singleton::deleteInstance();
  system("PAUSE");
  return 0;
}
```

Composite

Trong các ứng dụng hiện đại, việc sử dụng smart pointers như `std::shared_ptr` hoặc `std::unique_ptr` được coi là phổ biến hơn và được khuyến nghị hơn so với việc sử dụng raw pointers. Đây là vì smart pointers cung cấp các lợi ích như.

Định nghĩa

- Composite là mẫu thiết kế trong lập trình hướng đối tượng thuộc vào nhóm cấu trúc (Structural).
- Mẫu thiết kế Composite dùng để tạo ra các đối tượng mang tính chất phức hợp (Composite).
  Một đối tượng phức hợp Composite được tạo thành từ một hay nhiều đối tượng có chức
  năng tương tự nhau
  - Thao tác trên một nhóm đối tượng theo cách thao tác trên một đối tượng



Lưu ý:

- Component có thể chứa Add, Remove hoặc không chưa. Component bắt buộc chứa những đặc điểm chung. Nếu nó chứa Add, Remove bởi vì: tránh phải ép kiểu phức tạp khi truy vấn dữ liệu từ cấu trúc cây.
- Có thể thêm một số hàm như: getNameClass, …

Cách sử dụng

- Nhận biết khi nào cần xài mẫu Composite:
  - Khi đối tượng cho phép chứa những loại đối tượng khác, bao gồm cả chính nó
  - Các thao tác của những đối tượng này có chức năng tương tự nhau
  - Mang tính chất đệ quy
- Mẫu Composite rất thích hợp cho việc xây dựng cấu trúc dạng phân lớp hay cấu trúc cây

Code

```
#include <iostream>
#include <vector>
#include <string>
#include <exception>

using namespace std;

class Component {
protected:
  string name;
  unsigned int size;
public:
  Component(string name) {
    this->name = name; this->size = size;
  }

  virtual string getName(){
    return name;
  }

  virtual unsigned int getSize() = 0;

  virtual void addComponent(Component* component) {}

  virtual void deleteComponet(Component* component) {}
```

```
};

class Composite : public Component{
private:
  vector<Component*> components;
public:
  Composite(string name): Component(name) {}

  unsigned int getSize() {
    unsigned int sum = 0;
    for (auto e : components) {
      sum += e->getSize();
    }

    return sum;
  }

  void addComponent(Component* component) {
    components.push_back(component);
  }

  void deleteComponet(Component* component) {
    for (int i = 0; i < components.size(); i++) {
      if (components[i] == component) {
        components.erase(components.begin() + i);
        break;
      }
    }
  }
};

class Leaf : public Component{
public:
  Leaf(string name, unsigned int size) : Component(name) {
    this->size = size;
  }

  unsigned int getSize() {
    return size;
  }
};

int main()
{
  try {
    Composite* folder1 = new Composite("ProgramFile");
    Leaf* file1 = new Leaf("Genshin.exe", 46);
    Composite* folder2 = new Composite("genshinFolder");
    Leaf* file2 = new Leaf("ControlPanel.exe", 2);
    folder2->addComponent(file1);
    folder1->addComponent(file2);
    folder1->addComponent(folder2);

    cout << "Size of folder1: " << folder1->getSize();
```

```
  }
  catch (exception& ex) {
    cout << ex.what();
  }


  system("PAUSE");
  return 0;
}
```

Prototype

Định nghĩa

- Creational pattern.
- Prototype quy định loại của các đối tượng cần tạo bằng cách dùng một đối tượng mẫu, tạo mới nhờ vào sao chép đối tượng mẫu này mà không làm cho code phụ thuộc vào các lớp của chúng.
- Prototype Pattern được dùng khi việc tạo một object tốn nhiều chi phí và thời gian trong khi bạn đã có một object tương tự tồn tại.
- Prototype Pattern cung cấp cơ chế để copy từ object ban đầu sang object mới và thay đổi giá trị một số thuộc tính nếu cần.

Cách sử dụng

- Cần giấu logic khởi tạo
- Nhân bản một object
- Giảm việc phân lớp (ví dụ như abstract method)
- Mẫu thiết kế Prototype không được sử dụng phổ biến trong việc xây dựng các ứng dụng nghiệp vụ (business application). Nó thường được sử dụng trong các kiểu ứng dụng xác định như đồ họa máy tính, CAD (Computer Assisted Drawing), GIS (Geographic Information Systems) và các trò chơi.

Code

```
#include <iostream>
#include <string>
#include <vector>
#include <exception>
```

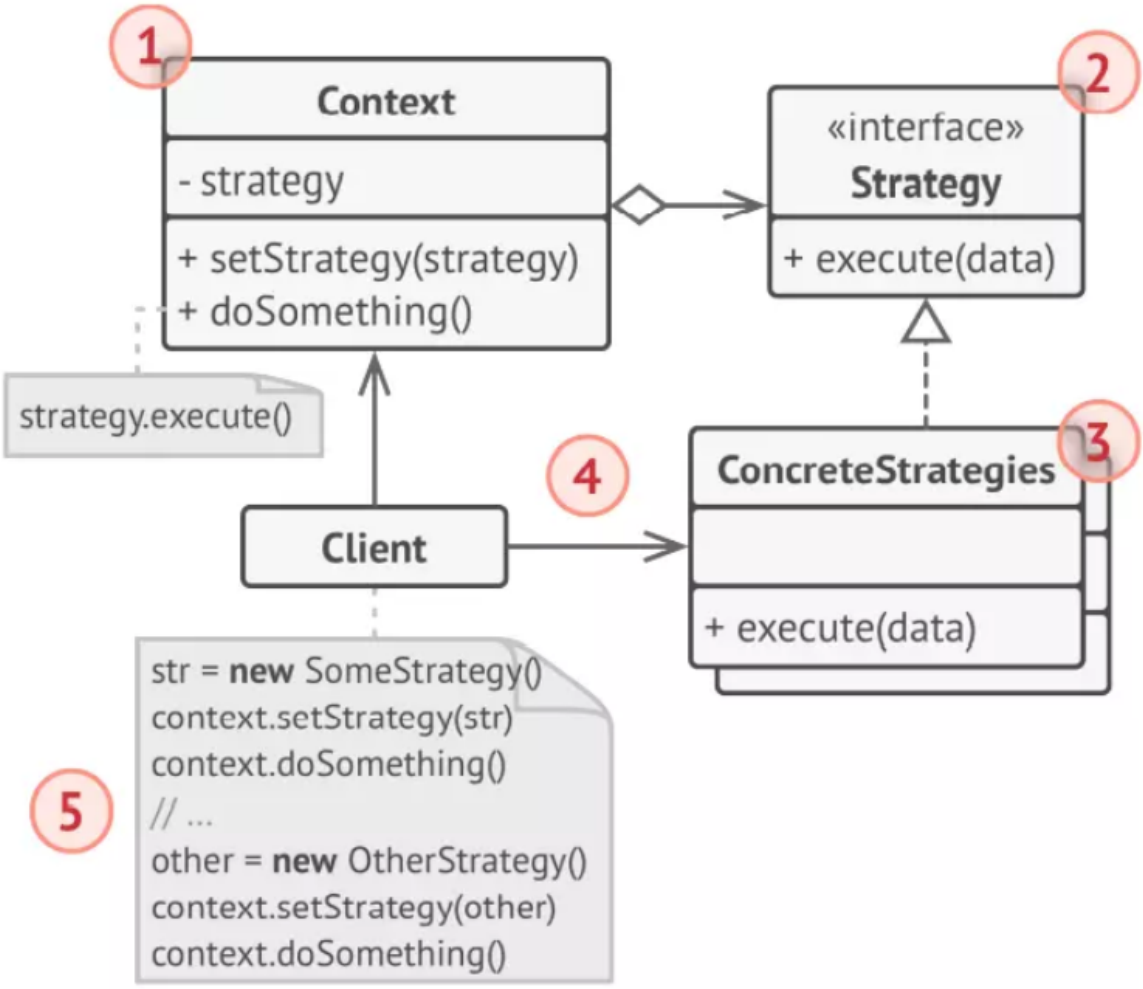
```
#include <unordered_map>

using namespace std;

class File {
protected:
  string name;
public:
  File(string name) {
    this->name = name;
  }

  File(const File* file) {
    this->name = name;
  }

  virtual File* clone() = 0;
  virtual string getNameClass() = 0;
  virtual void display() {
    cout << "Folder '" << name << "'" << endl;
  }
};

class FileWord :public File{
public:
  FileWord(string name) :File(name) {}

  FileWord(const File* file) :File(file) {}

  File* clone() {
    return new FileWord(this);
  }

  void display() {
    cout << "FileWord" << endl;
  }

  string getNameClass() {
    return "FileWord";
  }
};

class RegistryFile {
private:
  unordered_map<string, File*> map;
public:
  RegistryFile() {}

  void addRegistryList(File* file) {
    map.insert({ file->getNameClass(), file });
  }

  File* create(string nameClass) {
    return map[nameClass]->clone();
```

```
  }
};

int main()
{
  RegistryFile reg;
  FileWord* temp = new FileWord("BaiTap");
  reg.addRegistryList(temp);
  File* clone = reg.create("FileWord");
  clone->display();
  system("PAUSE");
  return 0;
}
```

Strategy

Định nghĩa

- Thuộc behavior pattern
- Mục đích:
  - Gom nhóm các thuật toán khác nhau có chung chức năng.
  - Dễ dàng thay thế nhau mà không phá vỡ cấu trúc chương trình.



Cách sử dụng

- Muốn sử dụng các biến thể khác nhau của một xử lý trong một đối tượng và có thể chuyển đổi giữa các xử lý trong runtime.
- Khi có nhiều lớp tương đương chỉ khác cách chúng thực thi một vài hành vi.
- Khi muốn tách biệt business logic của một lớp khỏi implementation details của các xử lý.
- Khi lớp có toán tử điều kiện lớn chuyển đổi giữa các biến thể của cùng một xử lý.

Code

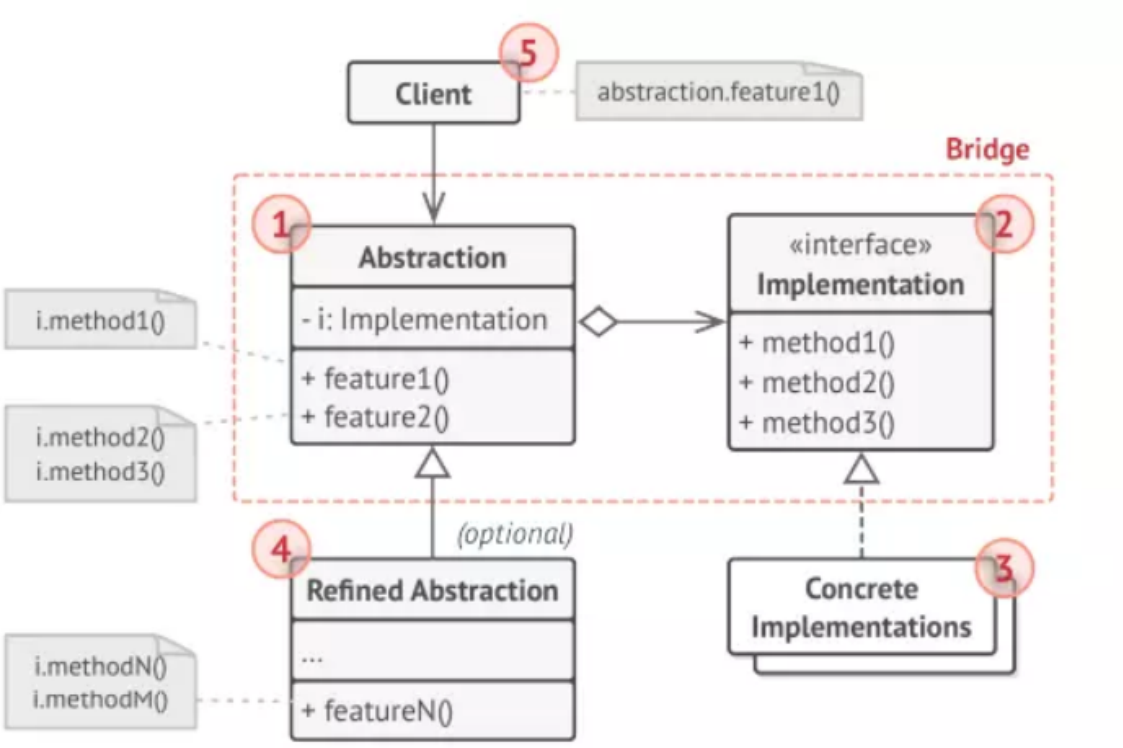
```
#include <iostream>
#include <string>
#include <exception>
#include <vector>

using namespace std;

class Strategy_Punch {
public:
  virtual void execute() = 0;
};

class ConcreteStrategy_StrongPuch : public Strategy_Punch {
public:
  void execute() {
    cout << "Punch: strong but slow" << endl;
  }
};

class ConcreteStrategy_WeakPuch : public Strategy_Punch {
public:
  void execute() {
    cout << "Punch: weak but fast" << endl;
  }
};

class Fighter {
  string name;
  Strategy_Punch* pPunch;
public:
  Fighter(string name, Strategy_Punch* pPunch) {
    this->name = name; this->pPunch = pPunch;
  }

  ~Fighter() {
    name = ""; pPunch = nullptr;
  }

  void setPuch(Strategy_Punch* punch) {
    this->pPunch = pPunch;
  }
  void punch() {
    pPunch->execute();
  }
};

int main() {
  Fighter girl("Girl", new ConcreteStrategy_WeakPuch());
  girl.punch();
```

```
  Fighter boy("Boy", new ConcreteStrategy_StrongPuch());
  boy.punch();
  system("Pause");
  return 0;
}
```

Bridge

Định nghĩa

- Bridge Pattern là một trong những Pattern thuộc nhóm Structural Pattern.
- Ý tưởng của nó là tách tính trừu tượng (abstraction) ra khỏi tính hiện thực (implementation) của nó. Từ đó có thể dễ dàng chỉnh sửa hoặc thay thế mà không làm ảnh hưởng đến những nơi có sử dụng lớp ban đầu



Cách sử dụng

- Khi muốn tách ràng buộc giữa Abstraction và Implementation, để có thể dễ dàng mở rộng độc lập nhau.
- Khi cả Abstraction và Implementation của chúng nên được mở rộng bằng subclass.

- Thay đổi trong thành phần được bổ sung thêm của một Abstraction mà không ảnh hưởng đối với các Client

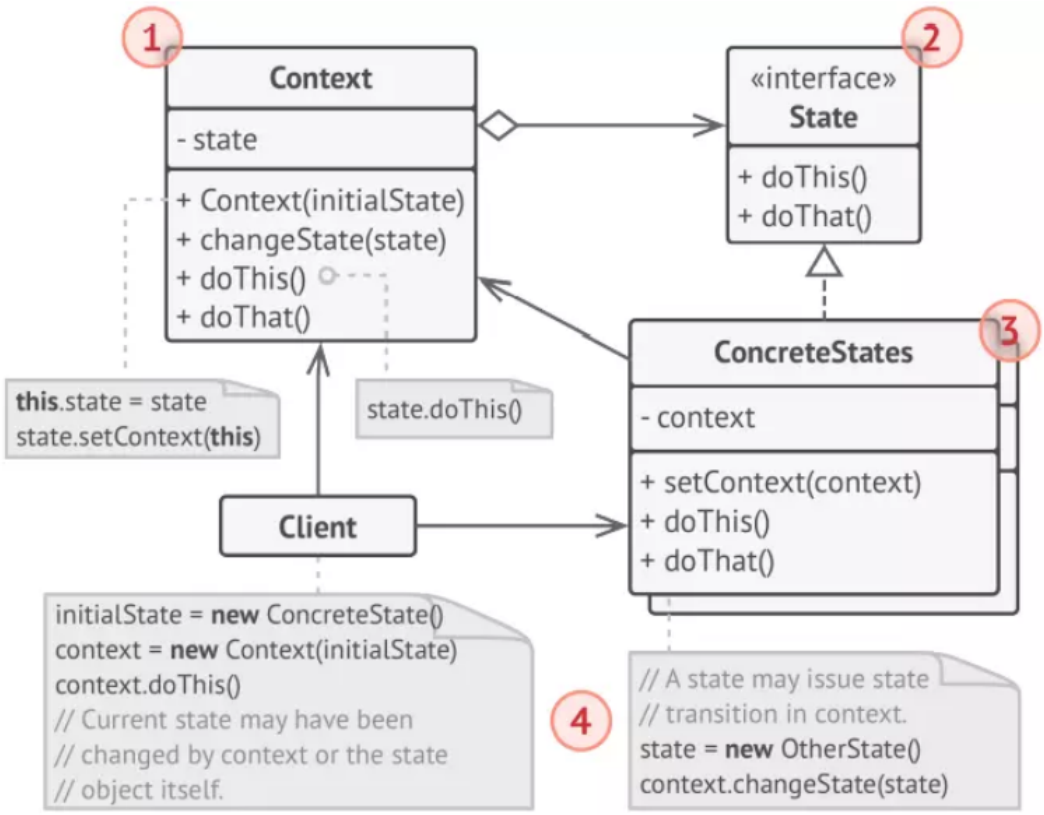
Code

```
#include <iostream>
#include <string>

using namespace std;

class Implementation {
public:
  virtual void work() = 0;
};

class WorkHard : public Implementation {
public:
  void work() {
    cout << "work fast" << endl;
  }
};

class WorkLazy : public Implementation {
public:
  void work() {
    cout << "work slowly" << endl;
  }
};

class Employee {
protected:
  string name;
  Implementation* implementation;

public:
  Employee(string name, Implementation* implementation) : name(name), implementation(implementation) {}

  void work() {
    implementation->work();
  }
};

int main()
{
  Employee anh("anh", new WorkHard());
  anh.work();
  system("PAUSE");
  return 0;
}
```

State

Định nghĩa

- State Pattern là một mẫu thiết kế thuộc nhóm Behavioral Pattern – những mẫu thiết kế xác định các mẫu giao tiếp chung giữa các object
- State Pattern là một mẫu thiết kế hành vi cho phép một object thay đổi hành vi của nó khi trạng thái bên trong của nó thay đổi.



Lưu ý: Lý do mà State chứa Context là vì ta cần phải thực hiện thay đổi một giá trị nào đó của context khi cài đặt hành vi.

Cách sử dụng

- Sử dụng State pattern khi bạn có một object hoạt động khác nhau tùy thuộc vào trạng thái hiện tại của nó, số lượng trạng thái là rất lớn và code của trạng thái cụ thể thường xuyên thay đổi.
- Sử dụng State pattern khi bạn có một lớp với nhiều các điều kiện lớn làm thay đổi cách class hoạt động theo các giá trị hiện tại của các trường của class.
- Sử dụng State Pattern khi bạn có nhiều code trùng lặp qua các trạng thái và chuyển đổi tương tự của State Pattern dựa trên điều kiện.
- Thay đổi hành vi object dựa trên trạng thái object
- Thay thế việc sử dụng rất nhiều điều kiện thay đổi cách lớp hành động dựa trên các giá trị của lớp

Code

```
#include <iostream>

using namespace std;

class WeatherState {
public:
  virtual void display() = 0;
};

class Rain : public WeatherState {
public:
  void display() {
    cout << "Weather: rain" << endl;
  }
};

class Sun : public WeatherState {
public:
  void display() {
    cout << "Weather: sun" << endl;
  }
};
```

```
class Game {
private:
  WeatherState* weatherState;
public:
  Game() {
    weatherState = new Sun();
  }

  ~Game() {
    delete weatherState;
    weatherState = nullptr;
  }

  void setState(WeatherState* weatherState) {
    delete this->weatherState;
    this->weatherState = weatherState;
  }

  void display() {
    weatherState->display();
  }
};

int main()
{
  Game game;
  game.display();
  game.setState(new Rain());
  game.display();

  system("PAUSE");
  return 0;
}
```