OOP

Gavrilut Dragos

Course 12

rev 1

Summary

- SOLID Principles
- Gang of Four (GoF)
- Singleton
- Multiton
- Factory
- Builder
- Adaptor
- Composite
- Visitor
- Observer
- Chain of Responsibility

SOLID Principles

SOLID Principles

Principle	Abbreviation
Single Responsibility Principle	SRP
Open-Close Principle	OCP
Liskov Substitution Principle	LSP
Interface Segregation Principle	ISP
Dependency Inversion Principle	DIP

Single Responsibility Principle

- A class should have only one reason to change, focusing on a single responsibility

Advantages:

- Easier to understand.
- Lower coupling between logic and infrastructure.
- Improved testability and reusability.

Single Responsibility Principle

- Let's analyze the following example that violates Single Responsibility Principle

App.cpp

```
#include <iostream>

class Car {
    string model;
    double fuelUsed;
    double distanceDriven;

public:
    Car(string m, double f, double d)
        : model(m), fuelUsed(f), distanceDriven(d) {}

    double compute_efficiency() const {
        return distanceDriven / fuelUsed;
    }

    void save(const string& filename) {
        std::cout << "Save: " << filename << std::endl;
    }

    void print_to_screen() const {
        std::cout << "Car: " << this->model << std::endl;
    }
};

int main() {
    Car c("Dacia", 100, 1.5);
    c.save("car.dat");
    c.print_to_screen();
}
```

Output:

```
Save: car.dat
Car: Dacia
```

Single Responsibility Principle

- Let's analyze the following example that violates Single Responsibility Principle

App.cpp

```
#include <iostream>

class Car {
    string model;
    double fuelUsed;
    double distanceDriven;

public:
    Car(string m, double f, double d)
        : model(m), fuelUsed(f), distanceDriven(d) {}

    double compute_efficiency() const {
        return distanceDriven / fuelUsed;
    }

    void save(const string& filename) {
        std::cout << "Save: " << filename << std::endl;
    }

    void print_to_screen() const {
        std::cout << "Car: " << this->model << std::endl;
    }
};

int main() {
    Car c("Dacia", 100, 1.5);
    c.save("car.dat");
    c.print_to_screen();
}
```

Output:

```
Save: car.dat
Car: Dacia
```

This methods are not relevant for a Car and imply additional tasks / responsibility for the Car class. The class has the following functionalities:

- **Managing state**
- **Compute efficiently**
- **Serialization support**
- **Print to screen**

Single Responsibility Principle

- Let's see the correct implementation of the previous problem so that the Single Responsibility Principle is respected:

App.cpp

```
#include <iostream>

class Car {
    string model;
    double fuelUsed;
    double distanceDriven;

public:
    Car(string m, double f, double d)
        : model(m), fuelUsed(f), distanceDriven(d) {}

    string get_model() const { return model; }
    double get_fuel_used() const { return fuelUsed; }
    double get_distance_driven() const {
           return distanceDriven;
    }
};
struct EfficiencyCalculator {
    double calculate(const Car& car) const {
        return car.get_distance_driven() /
               car.get_fuel_used();
    }
};
```

```
struct Serializer {
    void save(Car& car, string& filename) {
        std::cout << "Save : " << filename
                  << std::endl;
    }
};
```

```
struct Printer {
    void print(const Car& car) const {
        std::cout << "Car : "
                  << car.get_model()
                  << std::endl;
    }
};
```

```
int main() {
    Car c("Dacia", 100, 1.5);
    Serializer s;
    Printer p;
    s.save(c, « car.dat");
    p.print(c);
}
```

Output:

```
Save: car.dat
Car: Dacia
```

Each functionality is in its own class

Open-Close Principle

- “Software entities (such as modules, classes, functions, etc.) should be open for extension but closed for modification”

Advantages:

- Reduces the risk of breaking existing code
- Code reuse via polymorphism
- Easily to maintain
- Support for plug-ins

Open-Close Principle

- Let's analyze the following example that breaks open-close principle:

App.cpp

```
#include <iostream>

enum class ShapeType {
    Circle,
    Rectangle,
};

class Shape {
    ShapeType type;
    double dim1, dim2;
public:
    Shape(ShapeType t, double d1, double d2 = 0) :
                      type(t), dim1(d1), dim2(d2) {}
    double area() {
        switch (type) {
        case ShapeType::Circle:
            return 3.14 * dim1 * dim1;
        case ShapeType::Rectangle:
            return dim1 * dim2;
        default:
            return 0.0;
        }
    }
};

int main() {
    Shape c(ShapeType::Circle, 5);
    Shape r(ShapeType::Rectangle, 5, 4);
    std::cout << c.area() << std::endl;
    std::cout << r.area() << std::endl;
    return 0;
}
```

Output:

```
78.5
20
```

Open-Close Principle

- Let's analyze the following example that breaks open-close principle:

App.cpp

```
#include <iostream>

enum class ShapeType {
    Circle,
    Rectangle,
    Square
};
class Shape {
    …
public:
    Shape(ShapeType t, double d1, double d2 = 0) {…}
    double area() {
        switch (type) {
        case ShapeType::Circle:
            return 3.14 * dim1 * dim1;
        case ShapeType::Rectangle:
            return dim1 * dim2;
        default:
            return 0.0;
        }
    }
};

int main() {
    Shape c(ShapeType::Circle, 5);
    Shape r(ShapeType::Rectangle, 5, 4);
    std::cout << c.area() << std::endl;
    std::cout << r.area() << std::endl;
    return 0;
}
```

Let's assume that we want to add another shape: **Square**

Output:

```
78.5
20
```

Open-Close Principle

- Let's analyze the following example that breaks open-close principle:

App.cpp

```
#include <iostream>

enum class ShapeType {
    Circle,
    Rectangle,
    Square
};
class Shape {
    …
public:
    Shape(ShapeType t, double d1, double d2 = 0) {…}
    double area() {
        switch (type) {
        case ShapeType::Circle:
            return 3.14 * dim1 * dim1;
        case ShapeType::Rectangle:
            return dim1 * dim2;
        case ShapeType::Square:
            return dim1 * dim1;
        default:
            return 0.0;
        }
    }
};
```

```
int main() {
    Shape c(ShapeType::Circle, 5);
    Shape r(ShapeType::Rectangle, 5, 4);
    std::cout << c.area() << std::endl;
    std::cout << r.area() << std::endl;
    return 0;
}
```

Output:
78.5
20

We would also need to modify the ***area*** method to include the new shape, and this **breaks the open-close principle**

Open-Close Principle

- Let's see the correct implementation of the previous problem so that the Open-Close Principle is respected:

App.cpp

```
#include <iostream>

struct Shape {
    double virtual area() = 0;
};

struct Circle: public Shape {
    double ray;
    Circle(double r) : ray(r) {}
    double area() override {
        return 3.14 * ray * ray;
    }
};
struct Rectangle : public Shape {
    double lung, lat;
    Rectangle(double l1, double l2) :
            lung(l1), lat(l2) {}
    double area() override {
        return lung * lat;
    }
};
```

```
int main() {
    Circle c(5);
    Rectangle r(5, 4);
    std::cout << c.area() << std::endl;
    std::cout << r.area() << std::endl;
    return 0;
}
```

Output:
78.5
20

Notice the usage of polymorphism via the pure virtual method ***area*** from the interface / abstract class **Shape**

Liskov Substitution Principle

- Objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program.

or

- If class **S** is a subclass of class **T**, then objects of type **T** should be replaceable with objects of type **S** without breaking the logic expected of type **T**.

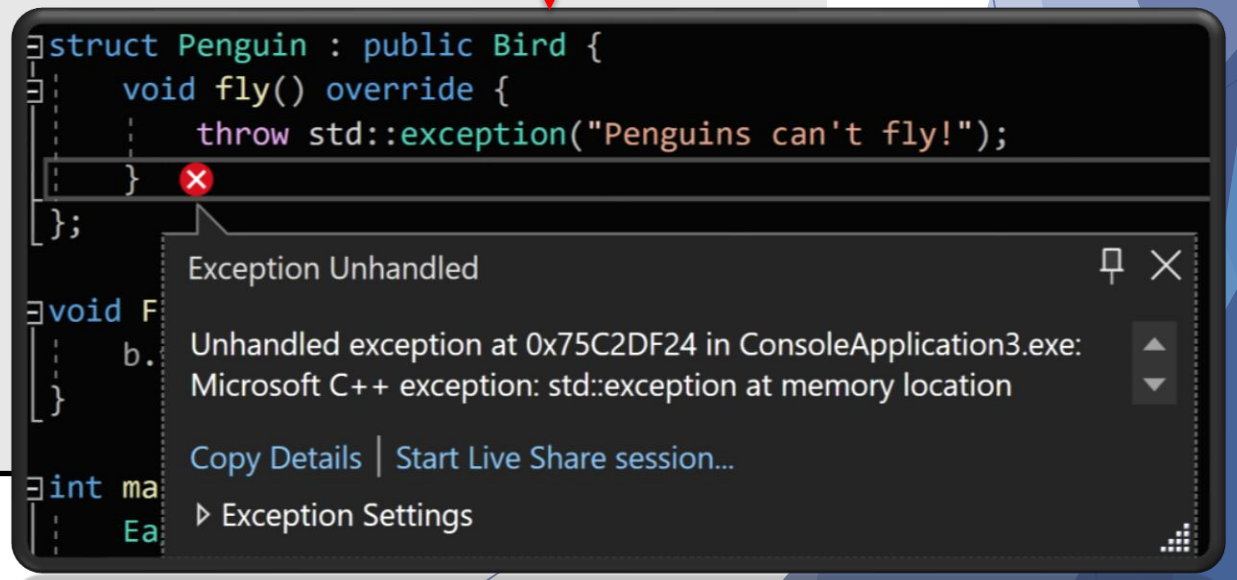
Liskov Substitution Principle

- Let's analyze the following example that breaks Liskov substitution principle:

App.cpp

```
#include <iostream>

struct Bird {
    virtual void fly() = 0;
};

struct Eagle : public Bird {
    void fly() override {
        std::cout << "Eagle::Fly()" << std::endl;
    }
};

struct Penguin : public Bird {
    void fly() override {
        throw std::exception("Penguins can't fly!");
    }
};

void Fly(Bird& b) {
    b.fly();
}

int main() {
    Eagle e;
    Penguin p;

    Fly(e);
    Fly(p);
}
```

Output:
Eagle::Fly



Liskov Substitution Principle

- Let's analyze the following example that breaks Liskov substitution principle:

App.cpp

```
#include <iostream>

struct Bird {
    virtual void fly() = 0;
};

struct Eagle : public Bird {
    void fly() override {
        std::cout << "Eagle::Fly()" << std::endl;
    }
};

struct Penguin : public Bird {
    void fly() override {
        throw std::exception("Penguins can't fly!");
    }
};

void Fly(Bird& b) {
    b.fly();
}

int main() {
    Eagle e;
    Penguin p;

    Fly(e);
    Fly(p);
}
```

The method **Fly** assumes that **each Bird can fly** (or in other words, every class derived from Bird has a valid ***fly method***)

Liskov Substitution Principle

- Let's analyze the following example that breaks Liskov substitution principle:

App.cpp

```
#include <iostream>

struct Bird {
    virtual void fly() = 0;
};

struct Eagle : public Bird {
    void fly() override {
        std::cout << "Eagle::Fly()" << std::endl;
    }
};

struct Penguin : public Bird {
    void fly() override {
        throw std::exception("Penguins can't fly!");
    }
};

void Fly(Bird& b) {
    b.fly();
}

int main() {
    Eagle e;
    Penguin p;

    Fly(e);
    Fly(p);
}
```

However, since Penguins can't fly, this implementation (while correct from a C++ standpoint of view), violates Liskov Substitution Principle

Liskov Substitution Principle

- Let's see the correct implementation of the previous problem so that the Liskov Substitution Principle is respected:

App.cpp

```
#include <iostream>

struct Bird {
};

struct BirdThatCanFly: public Bird {
    virtual void fly() = 0;
};

struct Eagle : public BirdThatCanFly {
    void fly() override {
        std::cout << "Eagle::Fly()" << std::endl;
    }
};

struct Penguin : public Bird {
};

void Fly(BirdThatCanFly& b) {
    b.fly();
}

int main() {
    Eagle e;
    Penguin p;

    Fly(e);
}
```

Output:
```
Eagle::Fly
```

Notice that we can no longer provide a Penguin parameter to function Fly !

Interface Segregation Principle

- Classes should not be forced to depend on interfaces they do not use.

or

- An interface (or abstract class) should contain only the methods that are relevant to the implementing class. It's better to have many small, specific interfaces than one large, all-purpose interface.

Interface Segregation Principle

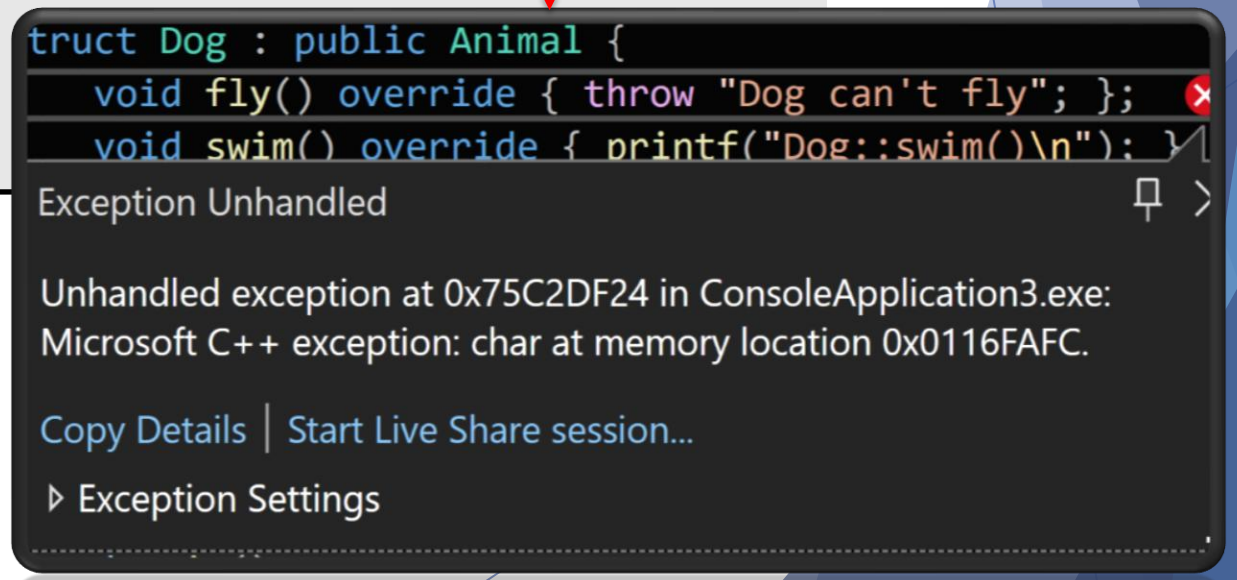
- Let's analyze the following example that breaks interface segregation principle:

App.cpp

```
#include <iostream>

struct Animal {
    virtual void fly() = 0;
    virtual void swim() = 0;
    virtual void eat() = 0;
};

struct Dog : public Animal {
    void fly() override { throw "Dog can't fly"; };
    void swim() override { printf("Dog::swim()\n"); };
    void eat() override { printf("Dog::eat()\n"); };
};

int main() {
    Dog d;
    d.eat();
    d.swim();
    d.fly();
}
```

Output:

```
Dog::eat()
Dog::swim()
```



Interface Segregation Principle

- Let's analyze the following example that breaks interface segregation principle:

App.cpp

```
#include <iostream>

struct Animal {
    virtual void fly() = 0;
    virtual void swim() = 0;
    virtual void eat() = 0;
};

struct Dog : public Animal {
    void fly() override { throw "Dog can't fly"; };
    void swim() override { printf("Dog::swim()\n"); };
    void eat() override { printf("Dog::eat()\n"); };
};

int main() {
    Dog d;
    d.eat();
    d.swim();
    d.f
}
```

The problem is that interface ***Animal*** implies that ***every animal should be able to fly, swim and eat*** (and this is not valid for most of the animals)

Interface Segregation Principle

- Let's analyze the following example that breaks interface segregation principle:

App.cpp

```
#include <iostream>

struct Animal {
    virtual void fly() = 0;
    virtual void swim() = 0;
    virtual void eat() = 0;
};

struct Dog : public Animal {
    void fly() override { throw "Dog can't fly"; };
    void swim() override { printf("Dog::swim()\n"); };
    void eat() override { printf("Dog::eat()\n"); };
};

int main() {
    Dog d;
```

The correct approach should split the initial interface into multiple ones (where we know that there aren't any animals that lack / can not implement all of the methods described in an interface).

Interface Segregation Principle

- Let's see the correct implementation of the previous problem so that the Interface Segregation Principle is respected:

App.cpp

```
#include <iostream>

struct Animal {
    virtual void eat() = 0;
};

struct AnimalThatCanSwim {
    virtual void swim() = 0;
};

struct AnimalThatCanFly {
    virtual void fly() = 0;
};

struct Dog : public Animal, public AnimalThatCanSwim {
    void swim() override { printf("Dog::swim()\n"); };
    void eat() override { printf("Dog::eat()\n"); };
};
struct Eagle : public Animal, public AnimalThatCanFly {
    void fly() override { printf("Eagle::fly()\n"); };
    void eat() override { printf("Eagle::eat()\n"); };
};

int main() {
    Dog d;
    d.eat();
    d.swim();
    Eagle e;
    e.eat();
    e.fly();
}
```

The original Animal interface was splitted in 3 different interfaces.

Output:

```
Dog::eat()
Dog::swim()
Eagle::eat()
Eagle::fly()
```

Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions

or

- High-level code (business logic) should not directly depend on concrete implementations (low-level details). Instead, both should depend on abstract interfaces, so you can swap out implementations (e.g., for testing, new behavior, or reuse).

- Dependency inversion principle promotes low-coupling between modules !

Dependency Inversion Principle

- Let's analyze the following example that breaks dependency inversion principle:

App.cpp

```
#include <iostream>

class Keyboard {
public:
    std::string get_input() {
        return "Key::Enter";
    }
};

class Computer
{
    Keyboard keyboard;
public:
    void process_input()
    {
        std::string input = keyboard.get_input();
        std::cout << "Input is : " << input
                  << std::endl;
    }
};

int main() {
    Computer c;
    c.process_input();
}
```

Output:

```
Input is: Key::Enter
```

Dependency Inversion Principle

- Let's analyze the following example that breaks dependency inversion principle:

App.cpp

```
#include <iostream>

class Keyboard {
public:
    std::string get_input() {
        return "Key::Enter";
    }
};

class Computer
{
    Keyboard keyboard;
public:
    void process_input()
    {
        std::string input = keyboard.get_input();
        std::cout << "Input is : " << input
                  << std::endl;
    }
};

int main() {
    Computer c;
    c.process_input();
}
```

Output:

```
Input is: Key::Enter
```

What if we want a different kind of input (e.g. ***mouse*** or ***touchscreen***) ?

Since **Computer** is the high level module it should not depend on lower level modules (such as keyboard) and in doing this it violates the Dependency inversion principle.

Whenever we would like to add a new input device, we would have to modify the high module to support it !!!

Dependency Inversion Principle

- Let's see the correct implementation of the previous problem so that the Dependency Inversion Principle is respected:

App.cpp

```
#include <iostream>

struct Input {
    virtual std::string get_input() = 0;
};

struct Keyboard: public Input {
    std::string get_input() override {
        return "Key::Enter";
    }
};

struct Mouse : public Input {
    std::string get_input() override {
        return "Mouse::Click";
    }
};

class Computer {
    Input *input;
public:
    Computer(Input* i) : input(i) {}
    void process_input() {
        std::string result = input->get_input();
        std::cout << "Input is : " << result << std::endl;
    }
};

int main() {
    Computer c1(new Keyboard());
    c1.process_input();
    Computer c2(new Mouse());
    c2.process_input();
    return 0;
}
```

Output:

```
Input is : Key::Enter
Input is : Mouse::Click
```

Dependency Inversion Principle

- Let's see the correct implementation of the previous problem so that the Dependency Inversion Principle is respected:

App.cpp

```
#include <iostream>

struct Input {
    virtual std::string get_input() = 0;
};

struct Keyboard: public Input {
    std::string get_input() override {
        return "Key::Enter";
    }
};

struct Mouse : public Input {
    std::string get_input() override {
        return "Mouse::Click";
    }
};

class Computer {
    Input *input;
public:
    Computer(Input* i) : input(i) {}
    void process_input() {
        std::string result = input->get_input();
        std::cout << "Input is : " << result << std::endl;
    }
};

int main() {
    Computer c1(new Keyboard());
    c1.process_input();
    Computer c2(new Mouse());
    c2.process_input();
    return 0;
}
```

Output:

```
Input is : Key::Enter
Input is : Mouse::Click
```

Notice that now we have two inputs (Keyboard and Mouse) both derived from the same interface (Input)

Dependency Inversion Principle

- Let's see the correct implementation of the previous problem so that the Dependency Inversion Principle is respected:

App.cpp

```
#include <iostream>

struct Input {
    virtual std::string get_input() = 0;
};

struct Keyboard: public Inp
    std::string get_input()
        return "Key::Enter"
    }
};

struct Mouse : public Input {
    std::string get_input() override {
        return "Mouse::Click";
    }
};
```

```
class Computer {
    Input *input;
public:
    Computer(In      i) : input(i) {}
    void proces      put() {
                                        endl;
                                    () {
    Computer c1(new Keyboard());
    c1.process_input();
    Computer c2(new Mouse());
    c2.process_input();
    return 0;
}
```

The ***Computer*** class does not store a Keyboard data member. Instead it stores a pointer to an Input interface (that could be anything). Further more, adding a new input will not change the Computer class as long as it is derived from the ***Input*** interface.

Output:

```
Input is : Key::Enter
Input is : Mouse::Click
```

- Gang of Four (GoF)

Gang of Four

GoF stands for **Gang of Four**, a nickname for the four authors of the influential software engineering book: ***Design Patterns: Elements of Reusable Object-Oriented Software*** **(1994)**

Authors (the "Gang of Four"):

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

Gang of Four

In their book they have introduced over 20 classic design patterns for object-oriented software development.

These patterns are grouped into the following categories:

Type	Purpose	Examples
Creational	Object creation mechanisms	Singleton, Factory, Builder
Structural	Composition of objects and classes	Adapter, Composite
Behavioral	Communication between objects	Observer, Visitor

Gang of Four

A **design pattern** is a **general, reusable solution to a common problem** that occurs in **software design**. It's **not a finished piece of code**, but rather a **template or best practice** that you can apply to solve a specific kind of problem in a particular context.

Characteristics:

- **Language-independent**: Patterns describe concepts, not syntax.
- **Reusable**: Once learned, a pattern can be used across many projects.
- **Proven**: Patterns are time-tested solutions used by experienced developers.
- **Documented structure**: Each pattern typically has a name, problem description, solution, and consequences.

Gang of Four

There are multiple designed patter described. However, for the purpose of this course we will discuss some of them for each category:

1. Creational → Singlton, Multiton, Factory and Builder
2. Structural → Adaptor and Composite
3. Behavioral → Visitor, Observer and Chain of Responsability

- Singleton

Singleton

- Problem: what if we want to model a class that can only have one instance ?
  The solution is to combine a private constructor with a static function:

App.cpp

```
class Object
{
    int value;
    static Object* instance;
    Object() { value = 0; }
public:
    static Object* GetInstance();
};
Object* Object::instance = nullptr;

Object* Object::GetInstance()
{
    if (instance == nullptr)
        instance = new Object();
    return instance;
}

void main()
{
    Object *obj = Object::GetInstance();
}
```

The default constructor is private – thus an object of this type can not be created !

As ***GetInstance*** is a static method of class Object, it can access any private constructors. However, as ***Object::instance*** is a static variable, the ***new*** operator will only be called once (when the first instance is requested ➔ therefor the name **Singleton**).

Singleton

- Problem: what if we want to model a class that can only have one instance ? The solution is to combine a private constructor with a static function:

App.cpp

```
class Object
{
    int value;
    static Object* instance;
    Object() { value = 0; }
public:
    static Object* GetInstance() { … }
};
Object* Object::instance = nullptr;

void main()
{
    Object *obj1 = Object::GetInstance();
    Object *obj2 = Object::GetInstance();
    Object *obj3 = Object::GetInstance();
}
```

```
Object* Object::GetInstance() {
    if (instance == nullptr)
        instance = new Object();
    return instance;
}
```

- Both ***obj1*** , ***obj2*** and ***obj3*** are in reality the same pointer. When ***obj1*** is first requested, Object::instance is first allocated, then it gets returned for ***obj2*** and ***obj3***.

Singleton

- Problem: what if we want to model a class that can only have one instance ? The solution is to combine a private constructor with a static function:

App.cpp

```
class Object
{
    int value;
    static Object* instance;
    Object() { value = 0; }
public:
    static Object* GetInstance() { … }
};
Object* Object::instance = nullptr;

void main()
{
    Object obj1;
    Object * obj = new Object();
}
```

```
Object* Object::GetInstance() {
    if (instance == nullptr)
        instance = new Object();
    return instance;
}
```

```
error C2248: 'Object::Object': cannot access private
member declared in class 'Object'
note: see declaration of 'Object::Object'
note: see declaration of 'Object'
```

- This code will not compile !

Singleton

- We can also modify a singleton to allow a maximum number of instances

App.cpp

```
#include <iostream>
#define MAX_INSTANCES 3
class Object
{
    int value;
    static Object* instances[MAX_INSTANCES];
    static int count;
    Object() { value = 0; }
public:
    static Object* CreateInstance();
    static void DeleteInstance(Object*);
};
Object* Object::instances[MAX_INSTANCES] = {};
int Object::count = 0;

Object* Object::CreateInstance()
{
    if (count >= MAX_INSTANCES) {
        return nullptr;
    }
    Object* o = new Object;
    instances[count] = o;
    count++;
    return o;
}
void Object::DeleteInstance(Object* o) {
    for (int tr = 0; tr < count; tr++) {
        if (instances[tr] == o) {
            delete o;
            instances[tr] = nullptr;
            for (int gr = tr + 1; gr < MAX_INSTANCES; gr++) {
                instances[gr - 1] = instances[gr];
            }
            count--;
            return;
        }
    }
}
void main()
{
    Object* o1 = Object::CreateInstance();
    Object* o2 = Object::CreateInstance();
    Object* o3 = Object::CreateInstance();
    Object* o4 = Object::CreateInstance();
    printf("%p,%p,%p,%p\n", o1, o2, o3, o4);
    Object::DeleteInstance(o2);
    Object* o5 = Object::CreateInstance();
    printf("%p", o5);
}
```

Output:

```
00C0CA58,00C0CA88,00C0CAB8,00000000
00C0CA88
```

Singleton

- We can also modify a singleton to allow a maximum number of instances

App.cpp

```
#include <iostream>
#define MAX_INSTANCES 3
class Object
{
    int value;
    static Object* instances[MAX_INSTANCES];
    static int count;
    Object() { value = 0; }
public:
    static Object* CreateI
    static void DeleteInst
};
Object* Object::instances[
int Object::count = 0;

Object* Object::CreateInstance()
{
    if (count >= MAX_INSTANCES) {
        return nullptr;
    }
    Object* o = new Object;
    instances[count] = o;
    count++;
    return o;
}
```

```
void Object::DeleteInstance(Object* o) {
    for (int tr = 0; tr < count; tr++) {
        if (instances[tr] == o) {
            delete o;
            instances[tr] = nullptr;
            for (int gr = tr + 1; gr < MAX_INSTANCES; gr++) {
                instances[gr - 1] = instances[gr];
            }
```

```
void main()
{
    Object* o1 = Object::CreateInstance();
    Object* o2 = Object::CreateInstance();
    Object* o3 = Object::CreateInstance();
    Object* o4 = Object::CreateInstance();
    printf("%p,%p,%p,%p\n", o1, o2, o3, o4);
    Object::DeleteInstance(o2);
    Object* o5 = Object::CreateInstance();
    printf("%p", o5);
}
```

The 4th instance was not instantiate as the Object only allows for 3 (MAX_INSTANCES)

Output:

```
00C0CA58,00C0CA88,00C0CAB8,00000000
00C0CA88
```

Singleton

- We can also modify a singleton to allow a maximum number of instances

App.cpp

```
#include <iostream>
#define MAX_INSTANCES 3
class Object
{
    int value;
    static Object* instances[MAX_INSTANCES];
    static int count;
    Object() { value = 0; }
public:
    static Object* CreateInstance();
    static void DeleteInstance(Object*);
};
Object* Object::instances[MAX_INSTANCES] = {};
int Object

Object* Ob
{
    if (co
        return nullptr;
    }
    Object* o = new Object;
    instances[count] = o;
    count++;
    return o;
}
```

```
void Object::DeleteInstance(Object* o) {
    for (int tr = 0; tr < count; tr++) {
        if (instances[tr] == o) {
            delete o;
            instances[tr] = nullptr;
            for (int gr = tr + 1; gr < MAX_INSTANCES; gr++) {
                instances[gr - 1] = instances[gr];
            }
            count--;
            return;
        }
    }
}

                                                    nce();
                                                ateInstance();
                                                ateInstance();
    Object* o4 = Object::CreateInstance();
    printf("%p,%p,%p,%p\n", o1, o2, o3, o4);
    Object::DeleteInstance(o2);
    Object* o5 = Object::CreateInstance();
    printf("%p", o5);
}
```

Output:

```
00C0CA58,00C0CA88,00C0CAB8,00000000
00C0CA88
```

Once one of the instances was deleted, we were able to create a new instance once again.

Singleton

- You can also use templates to build a singleton:

App.cpp

```
#include <iostream>

template <typename T>
class Singleton {
protected:
    static T* instance;
public:
    static T* GetInstance() {
        if (instance == nullptr) {
            instance = new T();
        }
        return instance;
    }
};

class Object : public Singleton<Object> {
    Object() : value(10) {}
public:
    friend class Singleton<Object>;
    int value;
};
Object* Object::instance = nullptr;

int main()
{
    Object* o = Object::GetInstance();
    printf("%d\n", o->value);
    return 0;
}
```

Output:
```
10
```

- But you still need to: **derive from the template**, make the new class **friend with the base class** and **create a static variable** that will hold the instance.

- Multiton

Multiton

- The **Multiton Pattern** is a **creational design pattern** that extends the Singleton pattern by allowing **controlled access to multiple instances**, each identified by a **unique key**.

It is often implemented using a **static map or dictionary**, where:

- The **key** is typically a string, enum, or ID.
- The **value** is a singleton-like instance corresponding to that key.

Multiton

- **A Multiton is often used for:**

1. Multi-lingual support (e.g. a Translator class can be implemented as a Multiton where each key is a language code (e.g., "en", "fr", "de"), and the corresponding instance holds the language-specific dictionary)

2. Named resources (e.g. when you have a fixed set of named resources that should each have a unique instance)

3. Using configuration set by the envinronment

Multiton

- Let’s see a multiton implementation for the previous Object class:

App.cpp

```
#include <iostream>
#include <unordered_map>

class Object {
    int value;
    static std::unordered_map<std::string, std::shared_ptr<Object>> instances;

    Object(int val) : value(val) {}

public:
    Object(const Object&) = delete;
    Object& operator=(const Object&) = delete;

    int getValue() const { return value; }

    static std::shared_ptr<Object> GetInstance(const std::string& key, int val = 0) {
        auto it = instances.find(key);
        if (it != instances.end())
            return it->second;
        auto instance = std::shared_ptr<Object>(new Object(val));
        instances[key] = instance;
        return instance;
    }
};
```

Multiton

- Let's see a multiton implementation for the previous Object class:

App.cpp

```
#include <iostream>
#include <unordered_map>

class Object {
    int value;
    static std::unordered_map<std::string, std::shared_ptr<Object>> instances;

    Object(int val) : value(val) {}

public:
    Object(const Object&) = delete;
    Object& operator=(const Object&) = delete;

    int getValue() const { return value; }

    static std::shared_ptr<Object> GetInstance(const std::string& key, int val = 0) {
        auto it = instances.find(key);
        if (it != instances.end())
            return it->second;
        auto instance = std::shared_ptr<Object>(new Object(val));
        instances[key] = instance;
        return instance;
    }
};
```

All instances of Object are stored in a map

Multiton

- Let's see a multiton implementation for the previous Object class:

App.cpp

```
#include <iostream>
#include <unordered_map>

class Object {
    int value;
    static std::unordered_map<std::string, std::shared_ptr<Object>> instances;

    Object(int val) : value(val) {}

pu
    int getValue() const { return value; }

    static std::shared_ptr<Object> GetInstance(const std::string& key, int val = 0) {
        auto it = instances.find(key);
        if (it != instances.end())
            return it->second;
        auto instance = std::shared_ptr<Object>(new Object(val));
        instances[key] = instance;
        return instance;
    }
};
```

When we request an instance, we need to provide a key for that instance. If the key exists we just return the instance, otherwise we create a new instance with the specified value ***val*** for Object initialization.

Multiton

- Let's see a multiton implementation for the previous Object class:

App.cpp

```
#include <iostream>
#include <unordered_map>

class Object {
    int value;
    static std::unordered_map<std::string, std::shared_ptr<Object>> instances;

    Object(int val) : value(val) {}

    int getValue() const { return value; }

    static std::shared_ptr<Object> GetInstance(const std::string& key, int val = 0) {
        auto it = instances.find(key);
        if (it != instances.end())
            return it->second;
        auto instance = std::shared_ptr<Object>(new Object(val));
        instances[key] = instance;
        return instance;
    }
};
```

Please note that we are using smart pointers (a ***shared_ptr***) to make sure that the object associated with a key is nit destroyed by mistake.

Multiton

- Let's see a multiton implementation for the previous Object class:

App.cpp

```
#include <iostream>
#include <unordered_map>

class Object {…};

std::unordered_map<std::string, std::shared_ptr<Object>> Object::instances;

int main() {
    auto a1 = Object::GetInstance("first", 42);
    auto a2 = Object::GetInstance("second", 100);
    auto a3 = Object::GetInstance("first");

    std::cout << "a1: " << a1->getValue() << "\n";
    std::cout << "a2: " << a2->getValue() << "\n";
    std::cout << "a3: " << a3->getValue() << "\n";

    return 0;
}
```

```
Output:
a1: 42
a2: 100
a3: 42
```

Notice that ***a1*** and ***a3*** are in fact the same variable (the one with the key "*first*")

- Factory

Factory

- "The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate.“
- Instead of creating objects directly with new, you use a factory (a method or class) that decides which object to create.
- This decouples object creation from the rest of the program and allows the programmer to maintain a higher logic (e.g. you can create a maximum number of elements at a specific period of time).

Factory

- Encapsulates the creation logic in one place.
- Enables easy substitution of object types without modifying client code.
- Promotes Open/Closed Principle: add new types without changing the factory interface.

Use Factory design pattern when:

- You want to **centralize and manage object creation**.
- The object to create depends on **input, configuration, or runtime logic**.
- You want to **abstract away the concrete classes** from the client.

Factory

- Factory example:

App.cpp

```
#include <iostream>

struct Animal {
    virtual void make_a_sound() = 0;
};

struct Dog : public Animal {
    void make_a_sound() override { std::cout << "Ham" << std::endl; }
private:
    Dog() {}
    friend class AnimalFactory;
};

struct Cat : public Animal {
    void make_a_sound() override { std::cout << "Miau" << std::endl; }
private:
    Cat() {}
    friend class AnimalFactory;
};
```

- Notice that each class derived from Animal has:
  - **A private constructor (so it can not be instantiated directly)**
  - **A friend class AnimalFactory that can access private members (e.g the constructor)**

Factory

- Factory example:

App.cpp

```
#include <iostream>

struct Animal {..};
struct Dog : public Animal {…};
struct Cat : public Animal {…};

struct AnimalFactory {
    static Animal* create_animal(const std::string& type) {
        if (type == "dog")
            return new Dog();
        else if (type == "cat")
            return new Cat();
        else
            return nullptr;
    }
};
int main() {
    auto a1 = AnimalFactory::create_animal("dog");
    auto a2 = AnimalFactory::create_animal("cat");
    a1->make_a_sound();
    a2->make_a_sound();
    return 0;

    return 0;
}
```

Output:

```
Ham
Miau
```

Factory

- Factory example:

App.cpp

```
#include <iostream>

struct Animal {..};
struct Dog : public Animal {…};
struct Cat : public Animal {…};

struct AnimalFactory {
    static Animal* create_animal(const std::string& type) {
        if (type == "dog")
            return new Dog();
        else if (type == "cat")
            return new Cat();
        else
            return nullptr;
    }
};
int main() {
    auto a1 = AnimalFactory::create_animal("dog");
    auto a2 = AnimalFactory::create_animal("cat");
    a1->make_a_sound();
    a2->make_a_sound();

    return 0;
}
```

Output:
Ham
Miau

Because ***AnimalFactory*** is a friend class of both **Dog** and **Cat** classes, it ca access their private constructor and as such it can create an instance of them.

Factory

- Factory example:

App.cpp

```
#include <iostream>

struct Animal {..};
struct Dog : public Animal {…};
struct Cat : public Animal {…};

struct AnimalFactory {..}

int main()
{
    auto dog = new Dog();

    dog->make_a_sound();

    return 0;
}
```

```
error C2248: 'Dog::Dog':
cannot access private member
declared in class 'Dog'
```

- Since Dog class has a private constructor, you can not create an object of type Dog without the ***AnimalFactory*** class.

Factory

- A factory is often used with an **enum** that specifies the type of object the factory should create.
- In our case, the ***AnimalKind*** enum specifies this !

App.cpp

```
#include <iostream>

struct Animal {..};
struct Dog : public Animal {…};
struct Cat : public Animal {…};

enum AnimalKind { Dog, Cat };

struct AnimalFactory {
    static Animal* create_animal(AnimalKind kind) { … }
}

int main()
{
    auto a1 = AnimalFactory::create_animal(AnimalKind::Dog);
    return 0;
}
```

Builder

Builder

- “The builder patter separate the construction of a complex object from its representation so that the same construction process can create different representations."
- The builder helps manage optional and mandatory parameters or various formats that are required for a parameter initialization
- Usually, a final .build() method is being called to create the actual object.

Builder

There are several scenarios when a Builder should be used:

- When a constructor would require too many parameters (or there are too many constructors for an object and its becoming complicated to create one)
- When an object requires multiple steps or conditions to be fully built
- When you want to reuse the construction process across different types or configurations.

Builder

- Let’s analyze the following example:

App.cpp

```
#include <iostream>
class Car {
    std::string name;
    std::string color;
    int speed;
    float fuel;
    int horsePower;
    bool manualSteering;
    std::string tireType;
public:
    Car(std::string name, std::string color, int speed, float fuel, int horsePower,
        bool manualSteering, std::string tireType) : name(name), color(color), speed(speed),
                                                     fuel(fuel), horsePower(horsePower),
                                                     manualSteering(manualSteering), tireType(tireType) {}
    void show() const {
        std::cout << "=== Car Information ===" << std::endl;
        std::cout << "Name: " << name << std::endl;
        std::cout << "Color: " << color << std::endl;
        std::cout << "Speed: " << speed << " km/h" << std::endl;
        std::cout << "Fuel: " << fuel << " liters" << std::endl;
        std::cout << "Horse Power: " << horsePower << " HP" << std::endl;
        std::cout << "Manual Steering: " << (manualSteering ? "Yes" : "No") << std::endl;
        std::cout << "Tire Type: " << tireType << std::endl;
    }
};
```

Notice that Car ctor has many parameters and you either need to use all of them, or you can make some of them default.
Either way, constructing a Car is not clear or readable !!!

Builder

- Let’s analyze the following example:

App.cpp

```
#include <iostream>
class Car {
…
public:
    Car(std::string name, std::string color, int speed, float fuel, int horsePower,
        bool manualSteering, std::string tireType) : name(name), color(color), speed(speed),
                                                     fuel(fuel), horsePower(horsePower),
                                                     manualSteering(manualSteering), tireType(tireType) {}
    void show() const {…}
};

int main() {
    Car c("Dacia", "Red", 100, 500, 110, false, "Winter");
    c.show();

    return 0;
}
```

Output:

```
=== Car Information ===
Name: Dacia
Color: Red
Speed: 100 km/h
Fuel: 500 liters
Horse Power: 110 HP
Manual Steering: No
Tire Type: Winter
```

- Looking on how “c” variable is constructed, is it clear what **100**, **500**, **110** and **false** parameters represent ?

Builder

- Let's see how we can create a Builder for the Car class:
  1 – modify the Car class to use a private constructor

App.cpp

```
class Car {
    std::string name;
    std::string color;
    int speed;
    float fuel;
    int horsePower;
    bool manualSteering;
    std::string tireType;

    Car(std::string name, std::string color, int speed, float fuel, int horsePower, bool manualSteering,
        std::string tireType)  : name(name), color(color), speed(speed), fuel(fuel), horsePower(horsePower),
                                 manualSteering(manualSteering), tireType(tireType) {}
public:
    void show() const {…}
};
```

Notice that the ***Car*** constructor is private (and can not be used directly)

Builder

- Let's see how we can create a Builder for the Car class:
  2 – create a builder class that has some default values and a method for each ***Car*** data member;

App.cpp

Notice that the ***Car*** constructor is private (and can not be used directly)

```
class Ca
    std:
    std:
    int speed = 0;
    float fuel = 0.0f;
    int horsePower = 0;
    bool manualSteering = true;
    std::string tireType = "All-season";
public:
    CarBuilder& SetName(const std::string& n) {
        name = n;
        return *this;
    }
    CarBuilder& SetColor(const std::string& c) {
        color = c;
        return *this;
    }
    CarBuilder& SetSpeed(int s) {
        speed = s;
        return *this;
    }
Builder& SetFuel(float f) {
    fuel = f;
    return *this;
}
CarBuilder& SetHorsePower(int hp) {
    horsePower = hp;
    return *this;
}
CarBuilder& SetManualSteering(bool manual) {
    manualSteering = manual;
    return *this;
}
CarBuilder& SetTireType(const std::string& type) {
    tireType = type;
    return *this;
}
Car build() const {
    return Car(name, color, speed, fuel, horsePower,
               manualSteering, tireType);
}
};
```

Builder

- Let's see how we can create a Builder for the Car class:
  2 – create a builder class that has some default values and a method for each ***Car*** data member;

App.cpp

```
class CarBuilder {
    std::string name = "Unnamed";
    std::string color = "Unpainted";
    int speed = 0;
    float fuel = 0.0f;
    int horsePower = 0;
    bool manualSteering = false;
    std::string tireType = "Standard";
pu
        name = n;
        return *this;
    }
    CarBuilder& SetColor(const std::string& c) {
        color = c;
        return *this;
    }
    CarBuilder& SetSpeed(int s) {
        speed = s;
        return *this;
    }
    CarBuilder& SetFuel(float f) {
        fuel = f;
        return *this;
    }
    CarBuilder& SetHorsePower(int hp) {
        horsePower = hp;
        return *this;
    }
    CarBuilder& SetManualSteering(bool manual) {
        manualSteering = manual;
        return *this;
    }
    CarBuilder& SetTireType(const std::string& type) {
        tireType = type;
        return *this;
    }
    Car build() const {
        return Car(name, color, speed, fuel, horsePower,
                   manualSteering, tireType);
    }
};
```

Notice that every method return a self reference (allowing the class to chain commands)

Builder

- Let's see how we can create a Builder for the Car class:
  4 – use the CarBuilder class to create a new Car

App.cpp

```
#include <iostream>

class Car { … };
class CarBuilder { … };

int main() {
    auto c = CarBuilder().SetColor("Red")
                         .SetFuel(300)
                         .SetHorsePower(150)
                         .SetManualSteering(true)
                         .SetSpeed(120)
                         .SetTireType("Summer")
                         .SetName("Audi")
                         .build();
    c.show();
    return 0;
}
```

Output:

```
=== Car Information ===
Name: Audi
Color: Red
Speed: 120 km/h
Fuel: 300 liters
Horse Power: 150 HP
Manual Steering: Yes
Tire Type: Summer
```

- Notice that it is much clearer right now how the car was constructed and with what parameters !

Adapter

Adapter

- “The adapter pattern convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces”.

Use an adapter when:

- You have an existing class whose interface does not match what a client expects.
- You want to reuse functionality from legacy or third-party code without modifying it.
- You need to integrate new components into a system that expects a fixed interface.

Adaptor

- Let’s analyze the following example:

App.cpp

```
#include <iostream>

struct Car {
    virtual void drive() = 0;
};
void run_a_test_drive(Car* car) {
    car->drive();
}
struct Dacia : public Car {
    void drive() override { std::cout << "Drive a Dacia !"; }
};

struct Bycicle {
    void move() { std::cout << "Bycicle::Move"; }
};

int main() {
    Dacia d;
    run_a_test_drive(&d);
}
```

Output:
```
Drive a Dacia !
```

What can we do to use the class **Bycicle** wth the **run_a_test_drive** function ?

Adaptor

- Let's analyze the following example:

App.cpp

```
#include <iostream>

struct Car {
    virtual void drive() = 0;
};
void run_a_test_drive(Car* car) {
    car->drive();
}
struct Dacia : public Car { … };

struct Bycicle {
    void move() { std::cout << "Bycicle::Move"; }
};
class BycicleAdapter : public Car {
    Bycicle* b;
public:
    BycicleAdapter(Bycicle* bycicle) : b(bycicle) {}
    void drive() override { b->move(); }
};

int main() {
    Bycicle b;
    auto adaptor = BycicleAdapter(&b);
    run_a_test_drive(&adaptor);
}
```

Output:
```
Bycicle::Move
```

The solution is to use an adaptor (a class that has an internal pointer of type Bycicle but also implements **Car** and as such can convert the ***move*** method from Bycicle to the drive method required by ***Car***.

- Composite

Composite

- "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions uniformly.“
- Composite object are in particular useful for representing Tree-like data (where a node could be a leaf or not)
- Mainly used in UX design (to represent various controls / widgets as well as containers).

Composite

- Let’s analyze the following example:

App.cpp

```
#include <iostream>
#include <vector>

struct Widget {
    virtual void Paint() = 0;
};
struct Button : public Widget {
    std::string name;
    Button(std::string name) : name(name) {}
    void Paint() override { printf("%s::Paint()\n",name.c_str()); }
};
struct Window : public Widget {
    std::vector<Widget*> children;
    std::string name;
    Window(std::string name) : name(name) {}
    void Add(Widget* w) { children.push_back(w); }
    void Paint() override {
        printf("%s::Paint()\n", name.c_str());
        for (auto w : children) {
            w->Paint();
        }
    }
};

int main() {
    Window w("QuestionWindow");
    w.Add(new Button("Ok"));
    w.Add(new Button("Cancel"));
    w.Add(new Button("Retry"));
    w.Paint();
}
```

Output:

```
QuestionWindow::Paint()
Ok::Paint()
Cancel::Paint()
Retry::Paint()
```

Composite

- Let's analyze the following example:

App.cpp

```
#include <iostream>
#include <vector>

struct Widget {
    virtual void Paint() = 0;
};
struct Button : public Widget {
    std::string name;
    Button(std::string name) : name(name) {}
    void Paint() override { printf("%s::Paint()\n",name.c_str()); }
};
struct Window : public Widget {
    std::vector<Widget*> children;
    std::string name;
    Window(std::string name) : name(name) {}
    void Add(Widget* w) { children.push_back(w); }
    void Paint() override {
        printf("%s::Paint()\n", name.c_str());
        for (auto w : children) {
            w->Paint();
        }
    }
};

int main() {
    Window w("QuestionWindow");
    w.Add(new Button("Ok"));
    w.Add(new Button("Cancel"));
    w.Add(new Button("Retry"));
    w.Paint();
}
```

Notice that the Window first draw itself and then calls the Paint method for all of its children.

Visitor

Visitor

- "Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.“

Use visitor when:

- You want to add operations to a class hierarchy without modifying it.
- You have a stable object structure but expect to add many operations.
- You want to separate behavior from the data structure.

Visitor

- Let’s analyze the following example:

App.cpp

```
#include <iostream>

struct BinaryOperation {
    int left, right;
    BinaryOperation(int l, int r) : left(l), right(r){}

    int add() const { return left + right; }
    int sub() const { return left - right; }
};

int main()
{
    BinaryOperation b(10, 5);
    std::cout << b.add() << std::endl;
    std::cout << b.sub() << std::endl;
}
```

Output:

```
15
5
```

- What if we want to add another operation to the BinaryOperation structure ?
- Can this be done without changing the BinaryOperation class ?

Visitor

- Let's analyze the following example:

App.cpp

```
#include <iostream>

struct BinaryOperation;
struct Visitor {
    virtual int visit(BinaryOperation* op) = 0;
};

struct BinaryOperation {
    int left, right;
    BinaryOperation(int l, int r) : left(l), right(r){}

    int add() const { return left + right; }
    int sub() const { return left - right; }

    int accept(Visitor* visitor) {
        return visitor->visit(this);
    }
};
struct MultiplyVisitor : public Visitor {
    int visit(BinaryOperation* op) override {
        return op->left * op->right;
    }
};

int main() {
    BinaryOperation b(10, 5);
    std::cout << b.add() << std::endl;
    std::cout << b.sub() << std::endl;

    MultiplyVisitor mv;
    std::cout << b.accept(&mv) << std::endl;
}
```

```
Output:
15
5
50
```

This is how a new functionality is being added to the BinaryOperation Class

Visitor

- Let’s analyze a slightly more complex example:

App.cpp

```
#include <iostream>

struct Animal;
struct Visitor;

struct Animal {
    virtual void accept(Visitor& v) = 0;
};

struct Dog : public Animal {
    void accept(Visitor& v) override;
    void bark() const {
        std::cout << "Ham, Ham :)" << std::endl;
    }
};

struct Cat : public Animal {
    void accept(Visitor& v) override;
    void miau() const {
        std::cout << "Miau" << std::endl;
    }
};

struct Visitor {
    virtual void visit(Dog& d) = 0;
    virtual void visit(Cat& c) = 0;
    virtual ~Visitor() = default;
};

struct SoundVisitor : public Visitor {
    void visit(Dog& d) override { d.bark(); }
    void visit(Cat& c) override { c.miau(); }
};

void Dog::accept(Visitor& v) { v.visit(*this); }
void Cat::accept(Visitor& v) { v.visit(*this); }

int main() {
    Dog dog;
    Cat cat;

    SoundVisitor sound;
    dog.accept(sound);
    cat.accept(sound);
}
```

Output:

```
Ham, Ham ☺
Miau
```

Visitor

- Let's analyze a slightly more complex example:

App.cpp

```
#include <iostream>

struct Animal;
struct Visitor;

struct Animal {
    virtual void accept(Visitor& v) = 0;
};

struct Cat : public Animal {
    void accept(Visitor& v) override;
    void miau() const {
        std::cout << "Miau" << std::endl;
    }
};

struct Visitor {
    virtual void visit(Dog& d) = 0;
    virtual void visit(Cat& c) = 0;
    virtual ~Visitor() = default;
};

struct SoundVisitor : public Visitor {
    void visit(Dog& d) override { d.bark(); }
    void visit(Cat& c) override { c.miau(); }
};

void Dog::accept(Visitor& v) { v.visit(*this); }
void Cat::accept(Visitor& v) { v.visit(*this); }

int main() {
    Dog dog;
    Cat cat;

    SoundVisitor sound;
    dog.accept(sound);
    cat.accept(sound);
}
```

Notice that the accept method in Dog and Cat classes calls the method visit with a parameter of type ***this** (that translated that for Dog it will use a Dog instance and for Cat a Cat instance)

Output:

```
Ham, Ham ☺
Miau
```

Visitor

- Let's analyze a slightly more complex example:

App.cpp

This means that the concrete visitor implementation (SoundVisitor) should implement different functionalities for each class: **Dog** and **Cat**

Output:
Ham, Ham ☺
Miau

```
#include <iostream>

struct Animal;
struct Visitor;

struct Animal {
    virtual void accept(Visitor& v) = 0;
};

struct Cat : public Animal {
    void accept(Visitor& v) override;
    void miau() const {
        std::cout << "Miau" << std::endl;
    }
};
```

```
    virtual void visit(Dog& d) = 0;
    virtual void visit(Cat& c) = 0;
    virtual ~Visitor() = default;
};

struct SoundVisitor : public Visitor {
    void visit(Dog& d) override { d.bark(); }
    void visit(Cat& c) override { c.miau(); }
};

void Dog::accept(Visitor& v) { v.visit(*this); }
void Cat::accept(Visitor& v) { v.visit(*this); }

int main() {
    Dog dog;
    Cat cat;

    SoundVisitor sound;
    dog.accept(sound);
    cat.accept(sound);
}
```

Notice that the accept method in Dog and Cat classes calls the method visit with a parameter of type ***this** (that translated that for Dog it will use a Dog instance and for Cat a Cat instance)

Visitor

- The visitor comes with a drawback:

1. Adding a new class (e.g. in our case a Bird) requires updating all visitors.
2. Can be overkill for small hierarchies or if objects change frequently.

Observer

Observer

- "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.“

- You can think about the Observer like a YouTube channel where you can subscribe. There are many subscribers. Once a new video is posted on that channel , all subscribers will receive a notification that the new video is available. A subscriber can decide to unsubscribe and from that moment if a new video will be present, he/she will not receive any notification about that video.

Observer

- Let’s build an observer for a YouTube channel:
  1 - First we will define the observer interface

App.cpp

```
#include <vector>
#include <algorithm>
#include <iostream>

struct Observer {
    virtual void notify(const std::string& channelName, const std::string& videoTitle) = 0;
};
```

Observer

- Let’s build an observer for a YouTube channel:
  2 – The YouTube channel class contains a vector of observers (subscribers)

App.cpp

```
#include <vector>
#include <algorithm>
#include <iostream>

struct Observer { … };

struct YouTubeChannel {
    std::string name;
    std::vector<Observer*> subscribers;

    YouTubeChannel(const std::string& channelName) : name(channelName) {}

    void subscribe(Observer* obs) {
        subscribers.push_back(obs);
    }

    void unsubscribe(Observer* obs) {
        subscribers.erase(std::remove(subscribers.begin(), subscribers.end(), obs), subscribers.end());
    }

    void upload_video(const std::string& title) {
        std::cout << "[Channel] '" << name << "' uploaded: " << title << std::endl;
        notifySubscribers(title);
    }
};
```

Observer

- Let's build an observer for a YouTube channel:
  3 – We create a notifySubscribers method that notifies all subscribers.

App.cpp

```
#include <vector>
#include <algorithm>
#include <iostream>

struct Observer { … };

struct YouTubeChannel {
    std::string name;
    std::vector<Observer*> subscribers;

    YouTubeChannel(const std::string& channelName) : name(channelName) {}
    void subscribe(Observer* obs) {… }
    void unsubscribe(Observer* obs) {…}
    void upload_video(const std::string& title) {…}

private:
    void notifySubscribers(const std::string& videoTitle) {
        for (auto* subscriber : subscribers) {
            subscriber->notify(name, videoTitle);
        }
    }
};
```

This method has to be ***private*** so that it can only be called from within the class when a specific event happens (e.g. a video was uploaded)

Observer

- Let’s build an observer for a YouTube channel:
  4 – Let’s create two different subscribers (one for a mobile app, and one for email)

App.cpp

```
#include <vector>
#include <algorithm>
#include <iostream>

struct Observer { … };

struct YouTubeChannel { … };

struct MobileSubscriber : public Observer
{
    void notify(const std::string& channelName, const std::string& videoTitle) override {
        std::cout << "[Mobile] New video from " << channelName << ": " << videoTitle << std::endl;
    }
};

struct EmailSubscriber : public Observer
{
    void notify(const std::string& channelName, const std::string& videoTitle) override {
        std::cout << "[Email] You've got a new video from " << channelName << ": " << videoTitle
                  << std::endl;
    }
};
```

Observer

- Let's build an observer for a YouTube channel:
  5 – The observer can be used in the following way

App.cpp

```
#include <vector>
#include <algorithm>
#include <iostream>

struct Observer { … };
struct YouTubeChannel { … };
struct MobileSubscriber : public Observer { … };
struct EmailSubscriber : public Observer { … };

int main() {
    YouTubeChannel tech("C++Exam");
    MobileSubscriber phone;
    EmailSubscriber mail;

    tech.subscribe(&phone);
    tech.subscribe(&mail);

    tech.upload_video("Using C++ 2023");
    tech.upload_video("C++ observer");

    tech.unsubscribe(&phone);

    tech.upload_video("Tips & tricks for the exam");
    return 0;
}
```

Output:

```
[Channel] 'C++Exam' uploaded: Using C++ 2023
[Mobile] New video from C++Exam: Using C++ 2023
[Email] You've got a new video from C++Exam: Using C++ 2023
[Channel] 'C++Exam' uploaded: C++ observer
[Mobile] New video from C++Exam: C++ observer
[Email] You've got a new video from C++Exam: C++ observer
[Channel] 'C++Exam' uploaded: Tips & tricks for the exam
[Email] You've got a new video from C++Exam: Tips & tricks for the exam
```

Chain of Responsibility

Chain of Responsibility

- “Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.”

- You have a request that needs to be handled. You create a chain of handlers, each of which can: ***Handle*** the request or **pass it along to the next handler**.

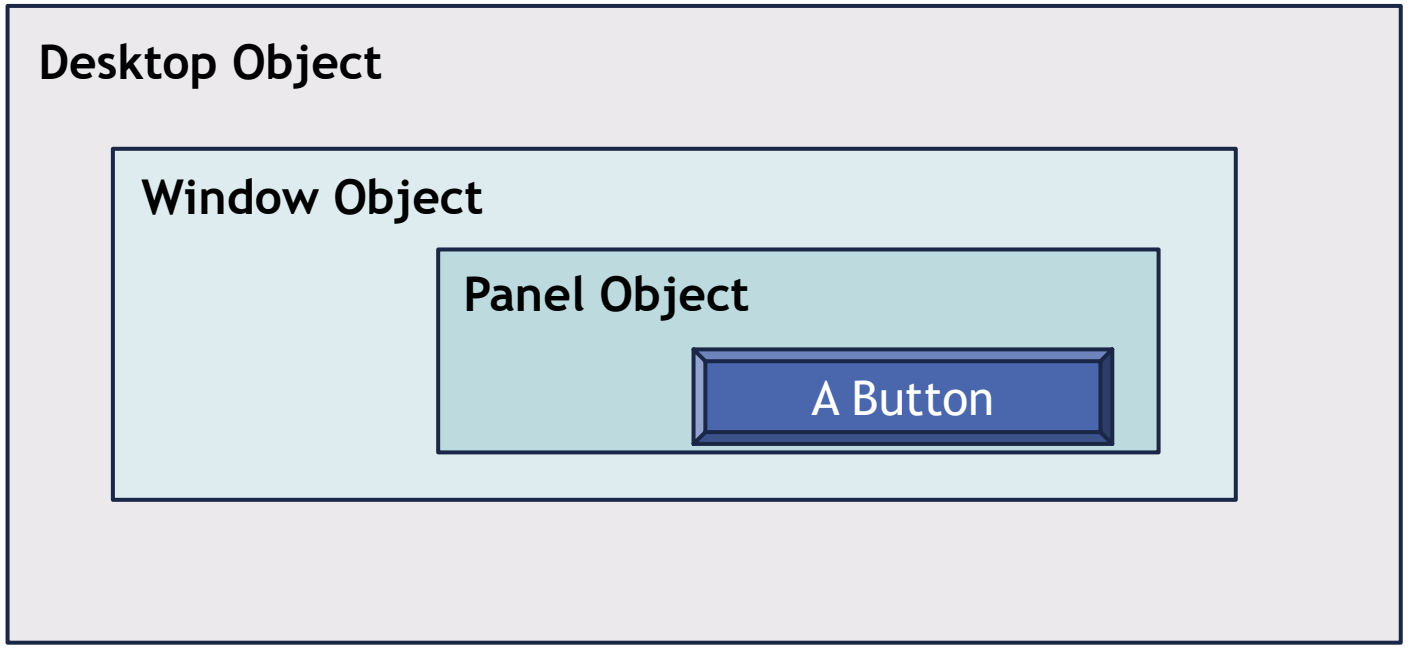
- Very useful for UX interface when processing input (keys, mouse, etc)

Chain of Responsibility

- “Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.”
- You have a request that needs to be handled. You create a chain of handlers, each of which can: ***Handle*** the request or **pass it along to the next handler**.
- Very useful for UX interface when processing input (keys, mouse, etc)

Chain of Responsibility

- Let’s define the following UX Architecture:



- Each of them know how to process a keyboard event (but if the keyboard event can not be processed they pass the event to their parent)

Chain of Responsibility

- Let's design a UX system that follows the chain of responsibility
  1 - First we will define a widget interface

App.cpp

```
#include <vector>
#include <iostream>

struct Widget {
    Widget* parent = nullptr;
    virtual void ProcessKey(std::string key) = 0;
    virtual void Paint() = 0;
    void SendKeyToParent(std::string key) {
        if (parent != nullptr)
            parent->ProcessKey(key);
    }
};
```

Both ProcessKey and Paint reflect characteristics of an Widget.
SendKeyToParent is a helper method that allows a widget to pass a key to its parent if it can not handle by itself.

Chain of Responsibility

- Let's design a UX system that follows the chain of responsibility
  2 – We will need a Container base class to handle basic operations such as Add.

App.cpp

```
#include <vector>
#include <iostream>

struct Widget {…}
struct Container: public Widget {
    std::vector<Widget*> children;
    void Add(Widget* w) {
        w->parent = this;
        children.push_back(w);
    }
    void PaintChildren() {
        for (auto c : children) {
            c->Paint();
        }
    }
};
```

- Method ***PaintChildren*** is a helper method that can be used by derivates classes to paint all children from a container.

Chain of Responsibility

- Let's design a UX system that follows the chain of responsibility
  3 – The desktop widget

App.cpp

```
#include <vector>
#include <iostream>

struct Widget {…}
struct Container: public Widget {…}
struct Desktop : public Container {
    virtual void ProcessKey(std::string key) {
        if (key == "Alt+F4") {
            std::cout << "Close application !" << std::endl;
        }
    };
    virtual void Paint() {
        std::cout << "Desktop paint" << std::endl;
        PaintChildren();
    };
};
```

- Notice that if the key is not processed by the Desktop, it is NOT passed to a different handler (it is assumed that the desktop is the last object in the chain).

Chain of Responsibility

- Let's design a UX system that follows the chain of responsibility
  4 – The window widget

App.cpp

```
#include <vector>
#include <iostream>

struct Widget {…}
struct Container: public Widget {…}
struct Desktop : public Container {…}
struct Window : public Container {
    virtual void ProcessKey(std::string key) {
        if (key == "Alt+F8") {
            std::cout << "Maximize window" << std::endl;
        }
        else {
            SendKeyToParent(key);
        }
    };
    virtual void Paint() {
        std::cout << "Window paint" << std::endl;
        PaintChildren();
    };
};
```

- Notice that in this case, if a key is not processed by the Window it is being passed to the parent via ***SendKeyToParent*** from Widget parent.

Chain of Responsibility

- Let's design a UX system that follows the chain of responsibility
  5 – The panel widget

App.cpp

```
#include <vector>
#include <iostream>

struct Widget {…}
struct Container: public Widget {…}
struct Desktop : public Container {…}
struct Window : public Container {…}
struct Panel : public Container {
    virtual void ProcessKey(std::string key) {
        if (key == "Shift+F10") {
            std::cout << "Hide panel" << std::endl;
        }
        else {
            SendKeyToParent(key);
        }
    };
    virtual void Paint() {
        std::cout << "Panel paint" << std::endl;
        PaintChildren();
    };
};
```

Chain of Responsibility

- Let's design a UX system that follows the chain of responsibility
  6 – The button widget

App.cpp

```
#include <vector>
#include <iostream>

struct Widget {…}
struct Container: public Widget {…}
struct Desktop : public Container {…}
struct Window : public Container {…}
struct Panel : public Container {…}
struct Button : public Widget {
    virtual void ProcessKey(std::string key) {
        if (key == "Enter") {
            std::cout << "Click on button" << std::endl;
        }
        else {
            SendKeyToParent(key);
        }
    };
    virtual void Paint() {
        std::cout << "Button paint" << std::endl;
    };
};
```

- Notice that in the case of Paint method, the ***PaintChildren*** from the Widget base class is not called anymore (a Button can not have children)

Chain of Responsibility

- Let's design a UX system that follows the chain of responsibility
  7 – Putting all of them together

App.cpp

```
#include <vector>
#include <iostream>

struct Widget {…}
struct Container: public Widget {…}
struct Desktop : public Container {…}
struct Window : public Container {…}
struct Panel : public Container {…}
struct Button : public Widget {…};

int main() {
    Desktop d;
    Button b;
    Panel p;
    Window w;
    p.Add(&b);
    w.Add(&p);
    d.Add(&w);
    b.ProcessKey("Enter");
    b.ProcessKey("Shift+F10");
    b.ProcessKey("Alt+F8");
    b.ProcessKey("Alt+F4");
    return 0;
}
```

```
Output:
Click on button
Hide panel
Maximize window
Close application !
```

Q & A