Creational Pattern

Kiến thức UML?

Hệ thống các mẫu Design pattern hiện có **23 mẫu** được định nghĩa trong cuốn "Design patterns Elements of Reusable Object Oriented Software" và được chia thành 3 nhóm:

- Creational Pattern (nhóm khởi tạo – 5 mẫu) gồm: **Factory Method, Abstract Factory**, Builder, Prototype, **Singleton**. Những Design pattern loại này cung cấp một giải pháp để tạo ra các object và che giấu được logic của việc tạo ra nó, thay vì tạo ra object một cách trực tiếp bằng cách sử dụng method new. Điều này giúp cho chương trình trở nên mềm dẻo hơn trong việc quyết định object nào cần được tạo ra trong những tình huống được đưa ra.
- Structural Pattern (nhóm cấu trúc – 7 mẫu) gồm: **Adapter**, Bridge, **Composite**, Decorator, **Facade**, Flyweight và Proxy. Những Design pattern loại này liên quan tới class và các thành phần của object. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.
- Behavioral Pattern (nhóm tương tác/ hành vi – 11 mẫu) gồm: Interpreter, **Template Method**, Chain of Responsibility, Command, Iterator, **Mediator**, Memento, **Observer**, **State**, **Strategy** và Visitor. Nhóm này dùng trong thực hiện các hành vi của đối tượng, sự giao tiếp giữa các object với nhau.

Factory Method

Factory method is a creational design pattern which solves the problem of creating product objects without specifying their concrete classes.

Factory Method defines a method, which should be used for creating objects instead of direct constructor call (`new` operator). Subclasses can override this method to change the class of objects that will be created.

Main: Connection *connection = DbConnectionFactory::createConnection(CONNECTION_TYPE_SQLSERVER);

// Class Factory, use ở hàm main:

```
class DbConnectionFactory
{
public:
   static Connection* createConnection(eConnectionType connectionType); // use static
};
// Connection là lớp base
Connection* DbConnectionFactory::createConnection(eConnectionType connectionType
{
   Connection *connection = nullptr;

   switch (connectionType)
   {
   case CONNECTION_TYPE_ORACLE:
      connection = new OracleConnection();
      break;
   case CONNECTION_TYPE_SQLSERVER:
      connection = new SqlServerConnection();
      break;
   case CONNECTION_TYPE_MYSQL:
      connection = new MySqlConnection();
      break;
   default:
      connection = new OracleConnection(); // default is Oracle
      break;
   }

   return connection;
}
```

Cách 2: sử dụng cho class Factory

```
// get instance use singleton
void DbConnectionFactory::createServiceModule() {
   m_oracleSvc = OracleConnection::getInstance();
   m_sqlServerSvc = SqlServerConnection::getInstance();
   m_mySqlSvc = MySqlConnection::getInstance();
}
```

Trong hàm createConnection: return m_oracleSvc, m_sqlServerSvc , m_mySqlSvc thay vì tạo thêm đối tượng của các class

// Class Base:

```
class Connection
{
   // define interfaces
   virtual string description() = 0;
        ...
};
```

// Define các class con kế thừa lớp base:

```
class OracleConnection : public Connection
{
   // implemement Connection's interfaces
   string description()
   {
      return "OracleConnection";
   };
};

class SqlServerConnection : public Connection
{
   // implemement Connection's interfaces
   string description()
   {
      return "SqlServerConnection";
   };
};
```

Singleton

It's pretty easy to implement a sloppy Singleton. You just need to hide the constructor and implement a static creation method.

The same class behaves incorrectly in a multithreaded environment. Multiple threads can call the creation method simultaneously and get several instances of Singleton class. (các luồng khác nhau có thể gọi phương thức tạo đồng thời và getInstances của Singleton class

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

Problem:

1. **Ensure that a class has just a single instance**
2. **Provide a global access point to that instance** (Cũng giống như một biến toàn cục, mẫu Singleton cho phép bạn truy cập một số đối tượng từ bất kỳ đâu trong chương trình)

Solution:

- Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
- Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object

How to Implement:

- **Add a private static field** to the class for storing the singleton instance.
- **Declare a public static creation method** for getting the singleton instance.
- **Implement "lazy initialization"** inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.
- **Make the constructor of the class private**. The static method of the class will still be able to call the constructor, but not the other objects.
- Go over the client code and replace all direct calls to the singleton's constructor with **calls to its static creation method**.

Example:

```
#include <iostream>
```

File .h:

```
class Singleton
{
private:
/* Here will be the instance stored. */
        static Singleton* instance;
/* Private constructor to prevent instancing. */
        Singleton();
public:
/* Static access method. */
        static Singleton* getInstance();
};
```

File .cpp:

```
/* Null, because instance will be initialized on demand. */
Singleton* Singleton::instance = 0; (or nullptr)

/* getInstance */
Singleton* Singleton::getInstance()
{
    if (instance == 0)
    {
        instance = new Singleton();
    }
    return instance;
}
//Contructor
Singleton::Singleton()
{}
```

Main

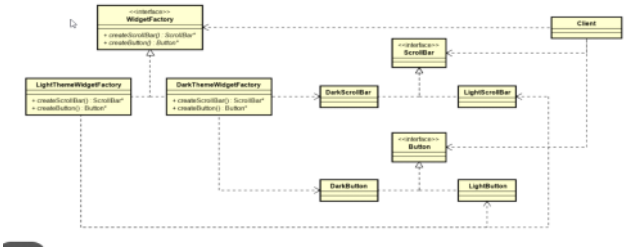
```
int main()
{
//new Singleton(); // Won't work
Singleton* s = Singleton::getInstance(); // Ok
Singleton* r = Singleton::getInstance();
/* The addresses will be the same. */
std::cout << s << std::endl;
std::cout << r << std::endl;
}
```

```
    // implemement Connection's interfaces
    string description()
    {
        return "SqlServerConnection";
    };
};

class MySqlConnection : public Connection
{
    // implemement Connection's interfaces
    string description()
    {
        return "MySqlConnection";
    };
};
```

Abstract Factory

- Xây dựng hệ thống hoặc ứng dụng sử dụng nhiều họ các products (products đây vd như ScrollBar, Button; còn họ ở đây vd như Dark, Light).
- Một họ các đối tượng products liên quan đến nhau, được thiết kế để được sử dụng cùng nhau một cách nhất quán.
- Muốn tách biệt độc lập phần code sử dụng các products với việc implement và tạo instance của chúng, nâng cao tính reusability và mantainability của hệ thống.



// Define Abstract Factory

```
class WidgetFactory
{
public:
    virtual ScrollBar* createScrollBar()=0;
    virtual Button* createButton()=0;
};
```

// LightScrollBar and DarkScrollBar kế thừa từ lớp base ScrollBar
// DarkButton and LightButton kế thừa từ lớp base Button
// LightThemeWidgetFactory và DarkThemeWidgetFactory kế thừa từ lớp Factory

```
class LightThemeWidgetFactory : public WidgetFactory
{
public:
    ScrollBar* createScrollBar() {
        return new LightScrollBar();
    }

    Button* createButton() {
        return new LightButton();
    }
};
class DarkThemeWidgetFactory : public WidgetFactory
{
public:
    ScrollBar* createScrollBar() {
        return new DarkScrollBar();
    }

    Button* createButton() {
        return new DarkButton();
    }
};
```

Main:

```
DarkThemeWidgetFactory widgetDarkFactory;
LightThemeWidgetFactory widgetLightFactory;
WidgetFactory *factoryPtr;
If(white)
{
        factoryPtr = &widgetLightFactory;
} else {
        factoryPtr = &widgetDarkFactory;
}
ScrollBar *scrollbar = factoryPtr->createScrollBar();
Button *button = factoryPtr->createButton();
```