Design Patterns

LSEG Technology

04th October 2023

LSEG

Agenda – Design Patterns

- What are design patterns?
- Benefits of design patterns
- Categorization of design patterns
- Examples

What is a design pattern?

- Typical solutions to commonly occurring problems in software design.
- Patterns are like pre-made blueprints that you can customize to solve a recurring problem in your code.
- Pattern is a general concept, not a specific piece of code.
- Does patterns and algorithms are same or different?
- The concept of design patterns originated in other fields but with the publication of “Elements of Reusable Object Oriented Software” by Gang of Four in 1994, concept has widely adopted by the developers.

What is a design pattern?

- Patterns are formally described and mostly consists with below information.
- **Pattern name**
- **The problem**: Explains the problem and the context
- **The solution**: Describes the elements that makeup the design, their relationships, responsibilities, and collaborations.
- **Consequences**: Results and the trade-offs of applying the pattern
- **Code example**

Why design pattern?

- Solutions defined in the design patterns are obtained by trial and error over a substantial period of time. So, you can use those without reinventing the wheel.
- Increases the readability, maintainability, and speed up the development process.
- It is a common language.

Categorization of Patterns

- Around 26 patterns are exist
- Can be categorized into 3 main types
  - Creational design patterns
    - About class instantiation
    - Eg: Singleton, Factory method, Abstract factory, Builder
  - Structural design patterns
    - About class and object composition
    - Eg: Adapter, Bridge, Composite, Decorator, Facade
  - Behavioral design patterns
    - About class’s objects communication
    - Eg: Chain of responsibility, Command, Interpreter, Mediator, Observer

Singleton (Creational)

- Only one instance of an object is created
- Make the default constructor private
- Create static creation method

```cpp
#include <stdio.h>
#include <iostream>
#include <string>

class Student
{
public:
    Student(const std::string& name) : m_name(name)
    {
    }
    ~Student() {}
private:
    const std::string m_name;
};

class StudentRegistry
{
public:
    static StudentRegistry& get() { return m_reg; }

    Student* find(const std::string name) { return nullptr; /*return student fond*/ }
    void add(Student* student) {}

private:
    StudentRegistry() {}
    ~StudentRegistry() {}

    static StudentRegistry m_reg;
};

StudentRegistry StudentRegistry::m_reg;

int main()
{
    auto& reg = StudentRegistry::get();
    auto student = reg.find("Student1");

    std::cin.get();
}
```

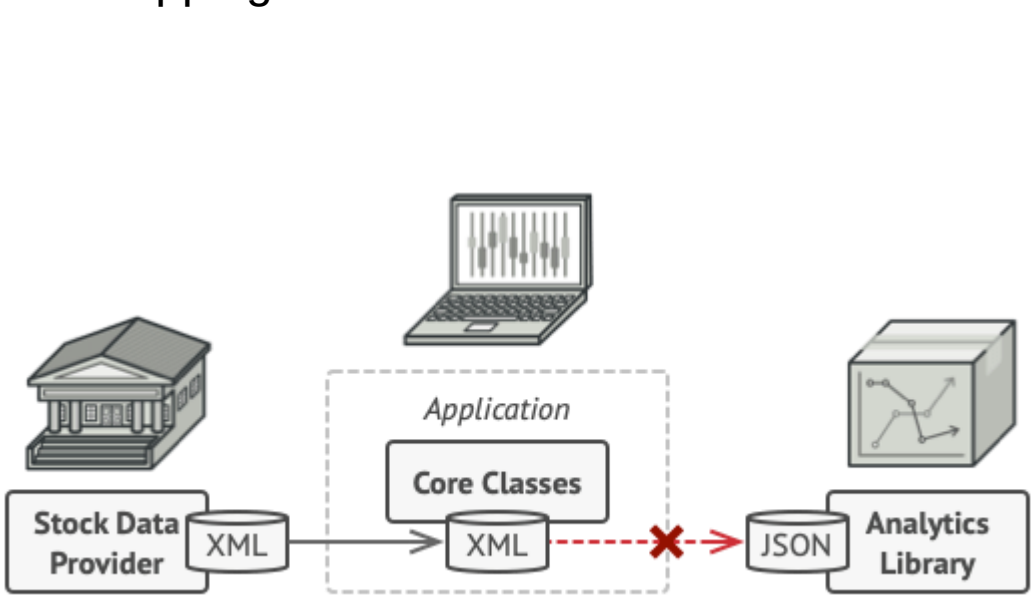
Factory (Creational)

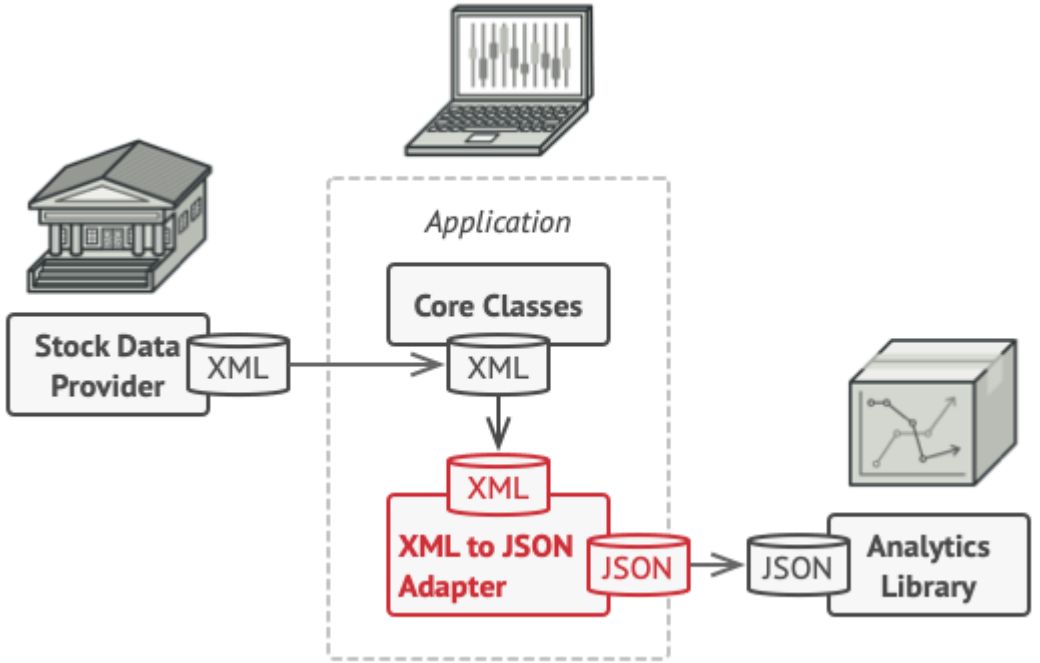
- Create objects without specifying the exact class to create

```cpp
#include <stdio.h>
#include <iostream>
#include <list>
#include <memory>

class Employee
{
public:
    Employee() {}
    virtual ~Employee() {}

    virtual void calculateSalary() = 0;
};

class FullTimeEmployee : public Employee
{
public:
    FullTimeEmployee() {}
    virtual ~FullTimeEmployee() {}

    void calculateSalary() override /*final also available*/
    {
        std::cout << "FullTimeEmployee - Salary calculation logic" << std::endl;
    }
};

class PartTimeEmployee : public Employee
{
public:
    PartTimeEmployee() {}
    virtual ~PartTimeEmployee() {}

    void calculateSalary() override /*final also available*/
    {
        std::cout << "PartTimeEmployee - Salary calculation logic" << std::endl;
    }
};

class BaseEmployeeFactory
{
public:
    virtual std::unique_ptr<Employee> createEmployee(std::string type) = 0;
};

class EmployeeFactory : public BaseEmployeeFactory
{
public:
    std::unique_ptr<Employee> createEmployee(std::string type) override
    {
        if (type == "FullTime")
        {
            return std::make_unique<FullTimeEmployee>();
        }
        else if (type == "PartTime")
        {
            return std::make_unique<PartTimeEmployee>();
        }
        else
        {
            std::cerr << "Unknown type" << std::endl;
            return nullptr;
        }
    }
};

int main()
{
    std::list<std::unique_ptr<Employee>> employees;
    std::unique_ptr<BaseEmployeeFactory> factory = std::make_unique<EmployeeFactory>();
    employees.push_back(std::move(factory->createEmployee("FullTime")));
    employees.push_back(std::move(factory->createEmployee("PartTime")));

    for (auto& employee : employees)
    {
        employee->calculateSalary();
    }

    std::cin.get();
}
```

Adapter (Structural)

- Allows for two incompatible classes to work together
- Wrapping an interface around one of the existing classes





Adapter (Structural)

```cpp
#include <stdio.h>
#include <iostream>
#include <memory>
#include <string>

struct Message
{
};

class Encoder
{
public:
    Encoder() {}
    virtual ~Encoder() {}
    virtual const std::string encode(Message& msg) = 0;
};

class JsonEncoder : public Encoder
{
public:
    JsonEncoder() {}
    virtual ~JsonEncoder() {}
    const std::string encode(Message& msg) override
    {
        std::cout << "JsonEncoder::encode" << std::endl;
        return "JsonEncoder - encoded buffer";
    }
};

class OldXMLEncoder
{
public:
    OldXMLEncoder() {}
    virtual ~OldXMLEncoder() {}
    const void encode(Message& msg, std::string& encoded)
    {
        std::cout << "OldXMLEncoder::encode" << std::endl;
        encoded = "OldXMLEncoder - encoded buffer";
    }
};
```

```cpp

class XMLEncoder : public Encoder
{
public:
    XMLEncoder() {}
    virtual ~XMLEncoder() {}

    const std::string encode(Message& msg) override
    {
        std::cout << "XMLEncoder::encode" << std::endl;
        std::string encodedBuffer;
        m_oldEncoder.encode(msg, encodedBuffer);
        return encodedBuffer;
    }

private:
    OldXMLEncoder m_oldEncoder;
};


int main()
{
    Message msg;
    std::unique_ptr<Encoder> e1 = std::make_unique<JsonEncoder>();
    std::cout << e1->encode(msg) << std::endl;

    std::cout << std::endl;
    std::unique_ptr<Encoder> e2 = std::make_unique<XMLEncoder>();
    std::cout << e2->encode(msg) << std::endl;

    std::cin.get();
}
```

```
JsonEncoder::encode
JsonEncoder - encoded buffer

XMLEncoder::encode
OldXMLEncoder::encode
OldXMLEncoder - encoded buffer
```

Observer (Behavior**)**

- Publish/subscribe pattern

```cpp
#include <stdio.h>
#include <iostream>
#include <memory>
#include <list>
#include <string>

class Socket;
class SocketObserver
{
public:
    SocketObserver() {}
    virtual ~SocketObserver() {}
    virtual void update(Socket& socket, const std::string data) = 0;
    //Instead of update can use meaningful name like onData, onRecvData, etc ...
};

class Socket
{
public:
    Socket() {}
    virtual ~Socket() {}
    void onData(const std::string data)
    {
        notifyObservers(data);
    }

    void registerObserver(std::shared_ptr<SocketObserver> observer)
    {
        m_observers.push_back(observer);
    }

private:
    void notifyObservers(const std::string data)
    {
        for (auto observer : m_observers)
        {
            observer->update(*this, data);
        }
    }
    std::list<std::shared_ptr<SocketObserver>> m_observers;
};

class WindowObserver : public SocketObserver
{
public:
    WindowObserver() {}
    virtual ~WindowObserver() {}
    void update(Socket& socket, const std::string data) override
    {
        std::cout << "Data received by WindowObserver. Data:" << data << std::endl;
    }
};

class ContainerObserver : public SocketObserver
{
public:
    ContainerObserver() {}
    virtual ~ContainerObserver() {}
    void update(Socket& socket, const std::string data) override
    {
        std::cout << "Data received by ContainerObserver. Data:" << data << std::endl;
    }
};

int main()
{
    std::unique_ptr<Socket> socket = std::make_unique<Socket>();
    std::shared_ptr<SocketObserver> observer1 = std::make_shared<WindowObserver>();
    std::shared_ptr<SocketObserver> observer2 = std::make_shared<ContainerObserver>();
    socket->registerObserver(observer1);
    socket->registerObserver(observer2);

    socket->onData("Test data");

    std::cin.get();
}
```

```
Data received by WindowObserver. Data:Test data
Data received by ContainerObserver. Data:Test data
```