The adapter pattern

START

Team:

Phan Huỳnh Anh Thư - 17095
Võ Công Minh - 10421040

Example:
Adapter & Traveling

Definition:

The observer pattern: a design pattern that allows two incompatible interfaces to work together. It is used when you have existing code that is not compatible with a new system. Rather than rewriting the existing code, you can use the Adapter Pattern to create an adapter that connects the old code to the new system. The adapter translates the interface of one object into another interface that a client expects.

.

When & How to use Apdater Pattern

When: Having two incompatible interfaces and you need to make them work together. Allowing to reuse of existing code instead of rewriting it. It also allows you to connect new systems to old systems that are not compatible with each other.

How: The Adapter Pattern works by creating an adapter that implements the interface of the new system, but internally uses the existing code to provide the necessary functionality. The adapter acts as a bridge between the two systems, translating the interface of the old system into the interface of the new system.

Types of Apdater Patterns

Class Adapter Pattern: The Class Adapter Pattern uses inheritance to adapt the interface of one class to another. The adapter class extends the existing class and implements the new interface. The adapter class uses the methods of the existing class to provide the required functionality for the new interface.

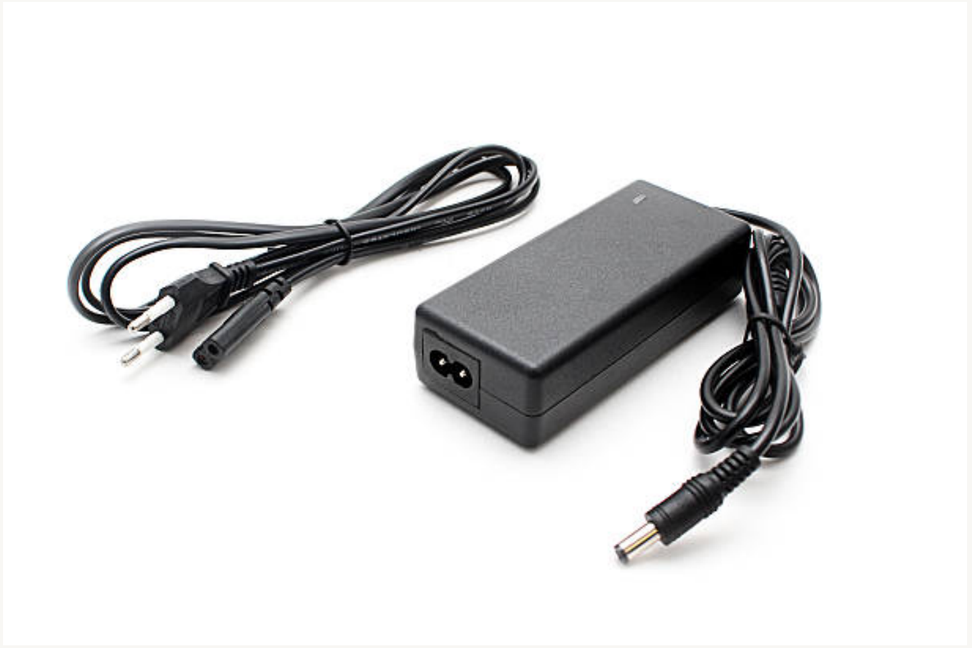
Object Adapter Pattern: The Object Adapter Pattern uses composition to adapt the interface of an object to another. The adapter class contains an instance of the existing class and implements the new interface. The adapter class uses the methods of the existing class to provide the required functionality for the new interface.

Implemetation

```cpp
#include <iostream>

// Existing interface
class Shape {
public:
    virtual void draw() = 0;
};

// Adaptee
class Rectangle {
public:
    void display() {
        std::cout << "Rectangle display function is called." << std::endl;
    }
};

// Adapter
class RectangleAdapter : public Shape {
public:
    RectangleAdapter(Rectangle* rect) {
        m_rect = rect;
    }

    void draw() override {
        m_rect->display();
    }

private:
    Rectangle* m_rect;
};

int main() {
    Rectangle rect;
    RectangleAdapter adapter(&rect);
    adapter.draw();
    return 0;
}
```

REAL-LIFE EXAMPLE

Power Adapter



Avantages

1. Allows the reuse of existing code
2. Enables incompatible systems' connection
3. Modularity and flexibility in software design
4. Maintainable and easier to understand.

Disavantages

1. Overhead
2. Complexity
3. Performance
4. Difficult to test

THANK YOU!