Paradygmaty programowania obiektowego

Paradygmaty programowania obiektowego, a wzorce projektowe

dr inż. Radosław Idzikowski

Definicja wzorca projektowego

Wzorzec projektowy

Wzorzec projektowy (ang. *design pattern*) – uniwersalne, sprawdzone w praktyce rozwiązanie często pojawiających się, powtarzalnych problemów projektowych. Pokazuje powiązania i zależności pomiędzy klasami oraz obiektami i ułatwia tworzenie, modyfikację oraz utrzymanie kodu źródłowego. Jest opisem rozwiązania, a nie jego implementacją.

- Wzorce projektowe dostarczają ogólnych rozwiązań dla konkretnych problemów projektowych.
- Są oparte na doświadczeniach programistów i architektów oprogramowania.
- Ułatwiają tworzenie skalowalnych, elastycznych i łatwo modyfikowalnych systemów.

Podział wzorców projektowych

Wzorce projektowe można podzielić na trzy główne kategorie:

- kreacyjne (ang. *Creational pattern*)
- strukturalne (ang. *Structural pattern*)
- behawioralne (ang. *Behavioral pattern*)

Wzorce kreacyjne

- Wzorce kreacyjne dotyczą procesu tworzenia obiektów.
- Służą do tworzenia obiektów w elastyczny sposób, niezależnie od konkretnej implementacji.
- Przykładowe wzorce kreacyjne to:
  - Fabryka,
  - Singleton,
  - Prototyp,
  - Budowniczy.

Fabryka

Fabryka (*Factory*)

Pozwala tworzyć rodziny spokrewnionych ze sobą obiektów bez określania ich konkretnych klas.

- Głównym celem wzorca "Fabryka" jest dostarczenie elastycznego i kontrolowanego sposobu tworzenia obiektów.
- Fabryka abstrahuje klienta od bezpośredniego tworzenia obiektów poprzez wykorzystanie metody fabrykującej.
- Metoda fabrykująca decyduje, jaki rodzaj obiektu zostanie stworzony i zwraca gotowy obiekt klientowi.

Fabryka

- Gwarancja, że produkty otrzymane stosując fabrykę są ze sobą kompatybilne.
- Zapobiegnięcie ścisłemu sprzęgnięciu konkretnych produktów z kodem.
- Zasada pojedynczej odpowiedzialności. Możesz zebrać kod kreacyjny produktów w jednym miejscu w programie, ułatwiając tym samym późniejsze utrzymanie kodu.
- Zasada otwarte/zamknięte. Możesz wprowadzać wsparcie dla nowych wariantów produktów bez psucia istniejącego kodu.

Singleton

Używany w celu zapewnienia, że istnieje tylko jedna instancja danej klasy w programie.

- Singleton gwarantuje, że klasa ma tylko jeden obiekt i zapewnia globalny dostęp do tego obiektu.
- Wzorzec Singleton jest przydatny w przypadkach, gdy potrzebujemy dostępu do wspólnego zasobu, np. bazy danych, konfiguracji aplikacji itp.
- Implementacja wzorca Singleton polega na ukryciu konstruktora klasy i udostępnieniu statycznej metody, która zwraca jedyną instancję klasy.

Singleton

```
class Singleton {
private:
    static Singleton* instance;
    Singleton() {}

public:
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }
};
Singleton* Singleton::instance = nullptr;
```

Singleton

- Gwarancja, że istnieje tylko jedna instancja klasy.
- Zyskujesz globalny dostęp do tej instancji.
- Obiekt singleton inicjalizowany jest dopiero wtedy, gdy jest po raz pierwszy potrzebny.

Prototyp (*Prototype*)

Umożliwia tworzenie nowych obiektów poprzez klonowanie istniejących obiektów.

- Ułatwianie tworzenia nowych obiektów, które różnią się tylko niektórymi parametrami.
- Wymaga zdefiniowania abstrakcyjnego interfejsu prototypu.

Prototyp

```
class Animal {
public:
    virtual Animal* clone() const = 0;
    virtual void toString() const = 0;
    virtual ~Animal() {}
};
class Dog : public Animal {
private:
    std::string name;
    int age;
public:
    Dog(const std::string& name, int age) : name(name), age(age) {}
    void setName(const std::string& name) { this->name = name; }
    void setAge(int age) { this->age = age; }
    void toString() const override {
        std::cout << "Dog: " << name << ", Age: " << age << std::endl;
    }
    Dog* clone() const override {
        return new Dog(*this);
    }
};
```

Prototyp

```
Animal* originalDog = new Dog("Buddy", 3);
originalDog->toString();
Animal* clonedDog = originalDog->clone();
clonedDog->toString();
Dog* clonedDogPtr = dynamic_cast<Dog*>(clonedDog);
if (clonedDogPtr) {
    clonedDogPtr->setName("Max");
    clonedDogPtr->setAge(2);
}
clonedDog->toString();
originalDog->toString();
delete originalDog;
delete clonedDog;

return 0;
```

Prototyp

- Możesz klonować obiekty bez konieczności sprzęgania ze szczegółami ich konkretnych klas.
- Możesz pozbyć się wielokrotnie powtarzanego kodu inicjalizacyjnego na rzecz klonowania prefabrykowanych prototypów.
- Dużo wygodniejsze produkowanie złożonych obiektów.
- Podejście to stanowi alternatywę do dziedziczenia w przypadku gdy mamy do czynienia z wcześniej zdefiniowanymi konfiguracjami złożonych obiektów.

Budowniczy

Budowniczy (*Bulider*)

Służy do konstrukcji obiektów krok po kroku. Oddziela proces konstrukcji od reprezentacji obiektu, umożliwiając różne sposoby tworzenia i reprezentowania obiektu.

- Pozwala na tworzenie złożonych obiektów krok po kroku.
- Oddziela proces konstrukcji od samego obiektu.
- Zapewnia różne reprezentacje obiektu.
- Ułatwia konstrukcję obiektów z takimi samymi składnikami, ale różnymi konfiguracjami.
- Definiowanie abstrakcyjnego budowniczego i produktu.
- Implementacja konkretnych budowniczych, które tworzą różne warianty produktów.
- Tworzenie dyrektora, który zarządza procesem konstrukcji.
- Konstruowanie obiektów za pomocą budowniczego z wybranymi składnikami.

Budowniczy

```
class DogBuilder {
public:
    virtual void buildName() = 0;
    virtual void buildAge() = 0;
    virtual Dog getDog() = 0;
};
class SimpleDogBuilder : public DogBuilder {
private:
    Dog dog;
public:
    void buildName() override {
        dog.setName("Max");
    }
    void buildAge() override {
        dog.setAge(2);
    }
    Dog getDog() override {
        return dog;
    }
};
```

Budowniczy

```
class Director {
private:
    DogBuilder* builder;
public:
    void setBuilder(DogBuilder* builder) {
        this->builder = builder;
    }
    Dog constructDog() {
        builder->buildName();
        builder->buildAge();
        return builder->getDog();
    }
};
int main() {
    Director director;
    SimpleDogBuilder builder;
    director.setBuilder(&builder);
    Dog dog = director.constructDog();
    dog.info();

    return 0;
}
```

Budowniczy

- Możesz konstruować obiekty etapami, odkładać niektóre etapy, lub wykonywać je rekursywnie.
- Możesz wykorzystać ponownie ten sam kod konstrukcyjny budując kolejne reprezentacje produktów.
- Zasada pojedynczej odpowiedzialności. Można odizolować skomplikowany kod konstrukcyjny od logiki biznesowej produktu.

Wzorce strukturalne

- Wzorce strukturalne dotyczą sposobu organizacji obiektów i klas.
- Służą do tworzenia większych struktur poprzez kompozycję obiektów.
- Przykładowe wzorce strukturalne to:
  - Dekorator,
  - Pełnomocnik.

Dekorator (*Decorator*)

Pozwala dynamicznie dodawać nowe funkcjonalności do istniejących obiektów, bez konieczności zmiany ich struktury.

- Umożliwienie elastycznego dodawania nowych zachowań lub właściwości do obiektów w czasie działania programu.
- Gdy chcemy dodać funkcjonalności do obiektu, ale nie chcemy modyfikować jego struktury.
- Gdy potrzebujemy wielu różnych kombinacji rozszerzeń funkcjonalności obiektu.
- Gdy chcemy, aby funkcjonalności obiektu były elastyczne i mogły być dodawane lub usuwane w czasie działania programu.

Dekorator

```
class Pizza {
public:
    virtual std::string getDescription() const = 0;
    virtual double getCost() const = 0;
};

class MargheritaPizza : public Pizza {
public:
    std::string getDescription() const override {
        return "Margherita␣Pizza";
    }

    double getCost() const override {
        return 5.99;
    }
};
```

Dekorator

```
class PizzaDecorator : public Pizza {
protected:
    Pizza* pizza;

public:
    PizzaDecorator(Pizza* pizza) : pizza(pizza) {}
};

class CheeseDecorator : public PizzaDecorator {
public:
    CheeseDecorator(Pizza* pizza) : PizzaDecorator(pizza) {}

    std::string getDescription() const override {
        return pizza->getDescription() + ", Extra Cheese";
    }

    double getCost() const override {
        return pizza->getCost() + 1.50;
    }
};
```

Dekorator

```
int main() {
    Pizza* margherita = new MargheritaPizza();
    Pizza* margheritaWithCheese = new CheeseDecorator(margherita);

    std::cout << "Order: " << margheritaWithCheese->getDescription() << std::endl;
    std::cout << "Cost: $" << margheritaWithCheese->getCost() << std::endl;

    delete margheritaWithCheese;
    delete margherita;

    return 0;
}
```

Dekorator

- Można rozszerzać zachowanie obiektu bez tworzenia podklasy.
- Można dodawać lub usuwać obowiązki obiektu w trakcie działania programu.
- Możliwe jest łączenie wielu zachowań poprzez nałożenie wielu dekoratorów na obiekt.
- Zasada pojedynczej odpowiedzialności. Można podzielić klasę monolityczną, która implementuje wiele wariantów zachowań, na mniejsze klasy.

Pełnomocnik (*Proxy*)

Pozwala na utworzenie obiektu zastępującego inny obiekt.

- Umożliwia dodanie dodatkowej warstwy kontroli lub funkcjonalności do istniejącego obiektu, bez zmiany jego interfejsu.
- Gdy chcemy ograniczyć dostęp do oryginalnego obiektu, na przykład w celu uwierzytelniania, autoryzacji lub kontroli dostępu.
- Gdy chcemy dodatkowo monitorować, zliczać lub rejestrować operacje wykonywane na oryginalnym obiekcie.
- Gdy tworzenie lub inicjalizacja oryginalnego obiektu jest kosztowna, a chcemy je opóźnić do momentu, gdy jest faktycznie potrzebny.

Pełnomocnik

- Można sterować obiektem usługi bez wiedzy klientów.
- Można zarządzać cyklem życia obiektu usługi, gdy klientów to nie interesuje.
- Pełnomocnik działa nawet wtedy, gdy obiekt udostępniający usługę nie jest jeszcze gotowy lub dostępny.
- Zasada otwarte/zamknięte. Można wprowadzać nowych pełnomocników do aplikacji bez modyfikowania usług lub klientów.

Wzorce behawioralne

- Wzorce behawioralne dotyczą interakcji między obiektami.
- Służą do opisywania sposobu komunikacji i zachowania obiektów.
- Przykładowe wzorce behawioralne to:
  - Iterator,
  - Obserwator,
  - Strategia.

Iterator

Umożliwia dostęp sekwencyjny do elementów kolekcji bez ujawniania jej wewnętrznej struktury.

- Zapewnia jednolity interfejs do iteracji po elementach kolekcji, niezależnie od jej konkretnej implementacji.
- Gdy chcemy zastosować ogólny sposób iteracji po elementach kolekcji bez konieczności zależności od konkretnej struktury kolekcji.
- Gdy chcemy umożliwić iterację po kolekcji w sposób sekwencyjny, bez konieczności dostępu do jej wewnętrznych elementów.
- Gdy chcemy umożliwić iterację w różnych kierunkach (np. od początku do końca, od końca do początku) bez zmiany kodu źródłowego.

Obserwator

Obserwator (*Observer*)

Definiuje relację jeden-do-wielu pomiędzy obiektami. Gdy stan jednego obiektu ulega zmianie, wszyscy zależni od niego obserwatorzy zostają poinformowani i automatycznie zaktualizowani.

- Umożliwia odseparowanie logiki biznesowej od warstwy prezentacji, umożliwiając reakcję na zmiany stanu obiektu bez bezpośredniej zależności od niego.
- Gdy potrzebujemy zapewnić synchronizację pomiędzy obiektami bez bezpośrednich zależności.
- Gdy chcemy umożliwić obiektom subskrypcję i reakcję na zdarzenia w innym obiekcie.
- Gdy zmiana stanu jednego obiektu może wpływać na wiele innych obiektów, które muszą być automatycznie aktualizowane.

Obserwator

```
class Observer {
public:
    virtual void update() = 0;
};

class ConcreteObserver : public Observer {
public:
    void update() override {
        std::cout << "ConcreteObserver: Otrzymalem powiadomienie o aktualizacji.\n";
    }
};
```

Obserwator

```
class Subject {
private:
    std::vector<Observer*> observers;

public:
    void attach(Observer* observer) {
        observers.push_back(observer);
    }

    void detach(Observer* observer) {
        auto it = std::find(observers.begin(), observers.end(), observer);
        if (it != observers.end()) {
            observers.erase(it);
        }
    }

    void notify() {
        for (auto observer : observers) {
            observer->update();
        }
    }
};
```

Obserwator

```
int main() {

    ConcreteObserver observer1;
    ConcreteObserver observer2;

    Subject subject;

    subject.attach(&observer1);
    subject.attach(&observer2);

    subject.notify();

    subject.detach(&observer1);
    subject.detach(&observer2);

    return 0;
}
```

Obserwator

- Zasada otwarte/zamknięte. Można wprowadzać do programu nowe klasy subskrybujące bez konieczności zmieniania kodu publikującego (i odwrotnie, jeśli istnieje interfejs publikujący).
- Można utworzyć związek pomiędzy obiektami w trakcie działania programu.

Strategia (*Strategy*)

Pozwala na zdefiniowanie wielu algorytmów lub strategii, które mogą być wymieniane w czasie działania programu. Algorytmy są enkapsulowane w oddzielnych klasach, co pozwala na ich niezależne użycie i zmianę bez wpływu na klientów.

- Umożliwia zmianę zachowania obiektu w trakcie działania programu, niezależnie od klientów. Pozwala na separację logiki biznesowej od jej implementacji.
- Gdy istnieje potrzeba zastosowania różnych algorytmów lub strategii w zależności od kontekstu.
- Gdy chcemy enkapsulować i ukryć różne implementacje algorytmów przed klientami.
- Gdy potrzebujemy elastyczności i możliwości zmiany algorytmu w czasie działania programu, bez wpływu na pozostałe komponenty.

Strategia

- Możesz zamieniać algorytmy stosowane w obrębie obiektu w trakcie działania programu.
- Możesz odizolować szczegóły implementacyjne algorytmu od kodu który z niego korzysta.
- Umożliwia zamianę dziedziczenia na kompozycję.
- Zasada otwarte/zamknięte. Możliwe jest wprowadzanie nowych strategii bez konieczności dokonywania zmian w kontekście.

Podsumowanie

- Wzorce projektowe to sprawdzone rozwiązania problemów projektowych w programowaniu.
- Wzorce projektowe pozwalają na tworzenie elastycznych, skalowalnych i łatwych w utrzymaniu systemów.
- Wzorce projektowe są ujęte w kategoriach, takich jak kreacyjne, strukturalne i behawioralne.
- Każdy wzorzec projektowy ma swoje specyficzne zastosowanie i korzyści.
- Wzorce projektowe można łączyć i adaptować do konkretnych potrzeb projektu.
- Zrozumienie i umiejętne stosowanie wzorców projektowych przyczynia się do poprawy jakości kodu i procesu tworzenia oprogramowania.
- Wzorce projektowe są powszechnie stosowane w różnych językach programowania, w tym w C++.

UWAGA!

Kod staje się bardziej skomplikowany, gdyż wdrożenie tego wzorca wiąże się z dodaniem wielu nowych klas!