

Wzorce projektowe

Zasady SOLID

SOLID jest akronimem od 5 zasad projektowania oprogramowania:

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single Responsibility Principle

Zasada jednej odpowiedzialności oznacza, że każda klasa odpowiada tylko za jedną odpowiedzialność. Przypuśćmy, że chcemy zrobić klasę Journal. Chcemy zapisywać do pliku. Dodajmy metodę:

```
1  void Journal::save(const string& filename)
2  {
3      ofstream ofs(filename);
4      for (auto& s : entries)
5          ofs << s << endl;
6  }
```

Single Responsibility Principle

Takie podejście jest problematyczne. Odpowiedzialnością klasy Journal jest trzymanie wpisów w czasopiśmie, nie zapis na dysku. Dodanie takiej lub podobnej funkcjonalności złamie zasadę jednej odpowiedzialności. Każda zmiana podejścia (np. zamiast zapisu na dysk to gromadzenie danych na serwerze) wymagałaby wymagałaby wielu drobnych zmian w każdej z danych klas.

Open-Closed Principle

Założmy, że mamy (całkowicie hipotetyczny) asortyment produktów w

Baza danych. Każdy produkt ma kolor i rozmiar i jest zdefiniowany jako:

```
1  enum class Color { Red, Green, Blue };
2  enum class Size { Small, Medium, Large };
3
4  struct Product
5  {
6      string name;
7      Color color;
8      Size size;
9  };
```

Teraz chcemy zapewnić pewne możliwości filtrowania dla danego zestawu produkty. Tworzymy filtr podobny do poniższego:

```
struct ProductFilter
{
    typedef vector Items;
};
```

Teraz, aby wesprzeć filtrowanie produktów według koloru, definiujemy członka funkcja, aby zrobić dokładnie to:

```
1 ProductFilter::Items ProductFilter::by_color(Items items, Color color)
2 {
3     Items result;
4     for (auto& i : items)
5         if (i->color == color)
6             result.push_back(i);
7     return result;
8 }
```

Nasze obecne podejście do filtrowania przedmiotów według koloru jest dobre. Nasz kod trafia do produkcji, ale niestety jakiś czas później szef przychodzi i prosi nas o zaimplementowanie filtrowania według rozmiaru. Więc odskakujemy z powrotem do `ProductFilter.cpp`, dodaj następujący kod i musimy przekompilować.

To brzmi jak zwykła duplikacja. Dlaczego po prostu nie napiszemy ogólnej metody, która przyjmuje predykat (pewną funkcję)? Cóż, jeden powód może być tak, że różne formy filtrowania można wykonać na różne sposoby: for na przykład niektóre typy rekordów mogą być indeksowane i należy je przeszukiwać określony sposób; niektóre typy danych można przeszukiwać na GPU, podczas gdy inni nie.

Nasz kod trafia do produkcji, ale po raz kolejny wraca szef i mówi nam, że teraz trzeba wyszukiwać zarówno według koloru, jak i rozmiaru. Więc co mamy zrobić, ale dodać kolejną funkcję?

```
1 ProductFilter::Items ProductFilter::by_color_and_size(Items
2 items, Size size, Color color)
3 {
4     Items result;
5     for (auto& i : items)
6         if (i->size == size && i->color == color)
7             result.push_back(i);
8     return result;
9 }
```

To, czego chcemy, z poprzedniego scenariusza, to wymuszenie otwarto-zamkniętej zasady, która mówi, że typ jest otwarty na rozszerzenie, ale zamknięty na modyfikacje. Innymi słowy, chcemy filtrowania, które jest rozszerzalne (być może w innej jednostce kompilacji) bez konieczności jej modyfikacji (i ponowne kompilowanie czegoś, co już działa i mogło zostać wysłane do klientów).

Jak możemy to osiągnąć?

Cóż, przede wszystkim rozdzielamy koncepcyjnie

(SRP!) nasz proces filtrowania na dwie części:

filtr (proces, który trwa wszystkie elementy i zwraca tylko niektóre)

specyfikację (predykat do zastosowania do elementu danych).

Możemy stworzyć bardzo prostą definicję interfejsu specyfikacji.

Ponownie, wszystko, co robimy, to określanie sygnatury funkcji o nazwie **filtr**, który pobiera wszystkie elementy i specyfikację i zwraca wszystkie elementy zgodne ze specyfikacją. Zakłada się, że przedmioty są przechowywane jako wektor $\langle T^* \rangle$, ale w rzeczywistości można również przekazać `filter()` dla iteratorów lub specjalnie zaprojektowany specjalnie zaprojektowany interfejs za przejście kolekcji. Na podstawie powyższego, implementacja ulepszonego filtra jest naprawdę prosta:

```
1 struct BetterFilter : Filter
2 {
3     vector filter(
4         vector items,
5         Specification& spec) override
6     {
7         vector result;
8         for (auto& p : items)
9             if (spec.is_satisfied(p))
10                 result.push_back(p);
11         return result;
12     }
13 };
```

Filtr kolorów

```
1 struct ColorSpecification : Specification
2 {
3     Color color;
4
5     explicit ColorSpecification(const Color color) : color{color} {}
6
7     bool is_satisfied(Product* item) override {
8         return item->color == color;
9     }
10};
```

Mając taką specyfikację i bazę możemy filtrować w następujący sposób:

```
1 Product apple{ "Apple", Color::Green, Size::Small };
2 Product tree{ "Tree", Color::Green, Size::Large };
3 Product house{ "House", Color::Blue, Size::Large };
4
5 vector all{ &apple, &tree, &house };
6
7 BetterFilter bf;
8 ColorSpecification green(Color::Green);
9
10 auto green_things = bf.filter(all, green);
11 for (auto& x : green_things)
12     cout << x->name << " is green" << endl;
```

Poprzednie daje nam „Jabłko” i „Drzewo”, ponieważ oba są zielone. Teraz jedyne, czego do tej pory nie zaimplementowaliśmy, to wyszukiwanie rozmiaru i kolor (lub rzeczywiście wyjaśniono, jak szukać rozmiaru lub koloru, lub mieszać różne kryteria). Odpowiedź brzmi: po prostu robisz kompozyt ‘specyfikacja’. Na przykład dla logicznego AND możemy to zrobić w następujący sposób:

```
1 template <typename T>struct AndSpecification : Specification<T>
2 {
3     Specification<T>& first;
4     Specification<T>& second;
5
6     AndSpecification(Specification<T>& first, Specification<T>& second)
7 : first{first}, second{second} {}
8
9 bool is_satisfied(T* item) override
10 {
11     return first.is_satisfied(item) && second.is_satisfied(item);
12 }
13 };
```

Teraz można swobodnie tworzyć złożone warunki na podstawie prostszej specyfikacji. Ponowne wykorzystanie zielonej specyfikacji, którą stworzyliśmy wcześniej, znalezienie czegoś zielonego i dużego jest teraz tak proste, jak

```
1 SizeSpecification large(Size::Large);
2 ColorSpecification green(Color::Green);
3 AndSpecification green_and_large{ large, green };
4
5 auto big_green_things = bf.filter(all, green_and_big);
6 for (auto& x : big_green_things)
7     cout << x->name << " is large and green" << endl;
8
9 // Tree is large and green
```

Liskov Substitution Principle

Nazwa pochodzi od Barbary Liskov, mówi, że jeśli interfejs przyjmuje obiekt typu Rodzic, to powinien on również przyjąć bez żadnego problemu obiekt typu Dziecko. Załóżmy, że mamy klasę Rectangle posiadającą wysokość, szerokość, getery, setery i funkcję do obliczania pola.

Przypuśćmy, że chcemy zrobić specjalny rodzaj Square, ta klasa dziedziczy po Rectangle settery co jest złym pomysłem

```

1  class Rectangle
2  {
3  protected:
4      int width, height;
5  public:
6      Rectangle(const int width, const int height)
7          : width{width}, height{height} { }
8
9      int get_width() const { return width; }
10     virtual void set_width(const int width) { this->wic
        width; }
11     int get_height() const { return height; }
12     virtual void set_height(const int height) { this->hei
        height; }
13
14     int area() const { return width * height; }
15 };

```

```

1  class Square : public Rectangle
2  {
3  public:
4      Square(int size): Rectangle(size,size) {}
5      void set_width(const int width) override {
6          this->width = height = width;
7      }
8      void set_height(const int height) override {
9          this->height = width = height;
10     }
11 };

```

Liskov Substitution Principle

Stwórzmy funkcję przyjmującą Rectangle, która nie zadziała przy Square

```
1  void process(Rectangle& r)
2  {
3      int w = r.get_width();
4      r.set_height(10);
5
6      cout << "expected area = " << (w * 10)
7          << ", got " << r.area() << endl;
8  }
```

Liskov Substitution Principle

Funkcja pobiera szerokość, ustawia wysokość i słusznie oczekuje, że produkt będzie równy obliczonemu obszarowi. Liczenie pola tą funkcją z obiektem klasy Square nie da poprawnego wyniku:

- 1 Square s{5};
- 2 process(s); // expected area = 50, got 25

Aby rozwiązać ten problem możemy użyć np. wzorca Fabryki.

Interface Segregation Principle

W dziedzinie inżynierii oprogramowania, zasada segregacji interfejsów (ISP) mówi, że żaden klient nie powinien być zmuszony do zależności od metod, których nie używa. ISP dzieli interfejsy, które są bardzo duże na mniejsze i bardziej szczegółowe tak, że klienci będą musieli wiedzieć tylko o tych metodach, które ich interesują. Takie zmniejszone interfejsy są również nazywane rolami interfejsy. ISP ma na celu utrzymanie systemu w stanie odsprężenia, a przez to łatwiejszego do refaktoryzacji, zmiany i ponownego wdrożenia. ISP jest jedną z pięciu zasad SOLID projektowania zorientowanego obiektowo, podobną do zasady wysokiej spójności GRASP

W projektowaniu zorientowanym obiektowo interfejsy zapewniają warstwy abstrakcji, które upraszczają kod i tworzą barierę zapobiegającą sprzężeniu z zależnościami. Według wielu ekspertów oprogramowania, którzy podpisali Manifesto for Software Craftsmanship, pisanie dobrze zaprojektowanego i zrozumiałego oprogramowania jest prawie tak samo ważne jak pisanie działającego oprogramowania. Używanie interfejsów do dalszego opisu intencji oprogramowania jest często dobrym pomysłem. System może stać się tak sprzężony na wielu poziomach, że nie jest już możliwe dokonanie zmiany w jednym miejscu bez konieczności wprowadzenia wielu dodatkowych zmian. Użycie interfejsu lub klasy abstrakcyjnej może zapobiec temu efektowi ubocznemu.

Dependency Inversion Principle

W projektowaniu zorientowanym obiektowo zasada odwrócenia zależności jest specyficzną formą luźno sprzężonych modułów oprogramowania. Stosując się do tej zasady, konwencjonalne relacje zależności ustanowione z modułów ustalających zasady wysokiego poziomu do modułów zależności niskiego poziomu są odwracane, w ten sposób uniezależniając moduły wysokiego poziomu od szczegółów implementacji modułu niskiego poziomu.

Według tej zasady:

1. Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Oba powinny zależeć od abstrakcji (np. interfejsów).
2. Abstrakcje nie powinny zależeć od szczegółów. Szczegóły (concrete implementations) powinny zależeć od abstrakcji.

W ten sposób, że zarówno obiekty wysokiego, jak i niskiego poziomu muszą zależeć od tej samej abstrakcji, taka zasada projektowania odwraca sposób, w jaki można myśleć o programowaniu obiektowym.

Ideą wymienionych punktów (1) i (2) w tejże zasadzie jest to, że projektując interakcję między modulem wysokiego poziomu a modulem niskiego poziomu, interakcję należy traktować jako abstrakcyjną interakcję między nimi. Ma to wpływ nie tylko na projekt modułu wysokopoziomowego, ale także na moduł niskopoziomowy:

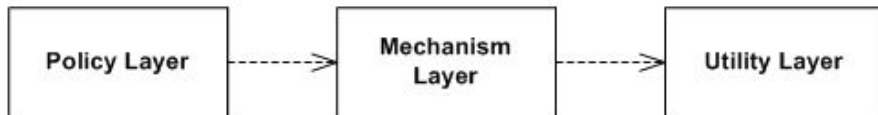
- moduł niskopoziomowy powinien być zaprojektowany z myślą o interakcji i może być konieczna zmiana jego interfejsu.

W wielu przypadkach myślenie o samej interakcji jako o abstrakcyjnym pojęciu pozwala na zredukowanie sprzężenia komponentów bez wprowadzania dodatkowych wzorców kodowania, pozwalając jedynie na lżejszy i mniej zależny od implementacji schemat interakcji.

Porównanie klasycznych “Traditional layer pattern” z “Dependency inversion pattern”

Traditional layer pattern

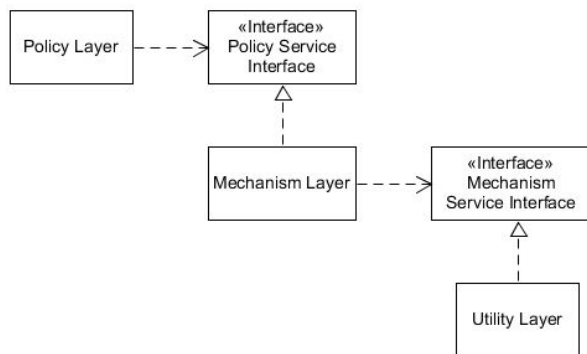
W konwencjonalnej architekturze aplikacji komponenty niższego poziomu (np. warstwa narzędziowa) są projektowane do wykorzystania przez komponenty wyższego poziomu (np. warstwa zasad), które umożliwiają budowanie coraz bardziej złożonych systemów. W tej kompozycji składniki wyższego poziomu zależą bezpośrednio od składników niższego poziomu, aby osiągnąć pewne zadanie. Ta zależność od składników niższego poziomu ogranicza możliwości ponownego wykorzystania składników wyższego poziomu.



Dependency inversion layer pattern

Wraz z dodaniem warstwy abstrakcyjnej zarówno warstwy wysokiego, jak i niższego poziomu zmniejszają tradycyjne zależności od góry do dołu. Niemniej jednak koncepcja „inwersji” nie oznacza, że warstwy niższego poziomu zależą bezpośrednio od warstw wyższego poziomu. Obie warstwy powinny zależeć od abstrakcji (interfejsów), które ujawniają zachowanie wymagane przez warstwy wyższego poziomu.

W bezpośrednim zastosowaniu odwracania zależności “abstrakty” (abstracts) należą do warstw wyższych. Ta architektura grupuje składniki wyższego poziomu i abstrakcje, które definiują niższe usługi w tym samym pakiecie. Warstwy niższego poziomu są tworzone przez dziedziczenie/implementację tych abstrakcyjnych klas lub interfejsów.



Wzorce projektowe

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 stosowane są w projektach wykorzystujących programowanie obiektowe.

Wzorce projektowe najczęściej tworzone są w oparciu o programowanie obiektowe . wzorce projektowe stanowią abstrakcyjny opis zależności pomiędzy klasami, co w efekcie wprowadza pewną standaryzację kodu oraz zwiększa jego zrozumiałość, wydajność i niezawodność. Wartość wzorców projektowych stanowi nie tylko samo rozwiązanie problemu, ale także dokumentacja, która wyjaśnia cel, działanie, zalety danego rozwiązania, co pomaga w łatwiejszym stosowaniu i adaptacji wzorców w danym zastosowaniu. Wzorce projektowe mogą przyspieszyć proces rozwoju oprogramowania przez dostarczenie wypróbowanych rozwiązań dla problemów, które mogą nie być oczywiste na początku procesu projektowego.

Elementy wzorca

Każdy wzorzec projektowy składa z czterech podstawowych elementów:

- nazwy wzorca;
- problemu – opisuje sposoby rozpoznawania sytuacji, w których możemy zastosować dany wzorzec oraz warunki jakie muszą zostać spełnione, by jego zastosowanie miało sens;
- rozwiązania – opisuje elementy rozwiązania: ich relacje, powiązania oraz obowiązki, zawiera także wskazówki implementacyjne dla różnych technologii;
- konsekwencji – zestawienie wad i zalet stosowania wzorca, uwzględniające informacje o jego brakach oraz kosztach rozwoju i utrzymania systemu wykorzystującego dany wzorzec.

Dokumentacja wzorca projektowego powinna zawierać informacje o rozwiązywanym problemie, kontekst w jakim należy go stosować oraz sugerowane rozwiązanie. Różni autorzy mogą stosować odmienne style tworzenia takiej dokumentacji, ale zwykle jej najważniejsze elementy są do siebie podobne. Dokumentacja może zawierać w sobie nazwę wzorca oraz klasyfikacji, przeznaczenie, motywację, stosowalność, struktura, uczestników, współpracę z innymi podmiotami, konsekwencję, implementację, przykładowy kod, przykład zastosowania, pokrewne wzorce. Jednym z najbardziej znanych układów opisu wzorca został zaproponowany przez Bandę Czterech.

Singleton

Singleton jest najbardziej nienawidzonym wzorcem projektowym, wyrósł z bardzo prostego pomysłu:

- powinien mieć tylko jedną instancję określonego komponentu.

Na przykład komponent, który ładuje bazę danych do pamięci i oferuje interfejs tylko do odczytu jest głównym kandydatem na Singletona, ponieważ nie ma sensu marnować pamięci na przechowywanie kilku identycznych zbiorów danych.

W rzeczywistości aplikacja może mieć ograniczenia takie, że dwie lub więcej instancji bazy danych po prostu nie zmieści się w pamięci lub spowoduje to brak pamięci, co spowoduje awarię programu.

Klasyczna implementacja

Umieszczamy statyczny licznik bezpośrednio w konstruktorze i rzucamy wyjątek, jeśli wartość zostanie kiedykolwiek zwiększona:

```
1 struct Database
2 {
3     Database()
4     {
5         static int instance_count{ 0 };
6         if (++instance_count > 1)
7             throw std::exception("Cannot make >1 database!");
8     }
9 };
```

Jest to szczególnie wrogie podejście do problemu: mimo że zapobiega tworzeniu więcej niż jednej instancji poprzez zgłoszenie wyjątku, nie przekazuje faktu, że nie chcemy, aby ktokolwiek wywołał konstruktor więcej niż jeden raz.

Bezpieczeństwo wątku /Thread Safety/

Inicjalizacja singletona we wskazany poprzednio sposób jest bezpieczna wątkowo od C++11, co oznacza, że jeśli dwa wątki chcą jednocześnie wywołać `get()`, nie spotkamy się z sytuacją, w której baza danych zostanie utworzona dwukrotnie.

Przed C++11 konstruowano singletona za pomocą podejścia zwanego “podwójnie sprawdzonym blokowaniem”.

Typowa implementacja wyglądałaby tak:

```
1 struct Database
2 {
3 // same members as before, but then...
4     static Database& instance();
5 private:
6     static boost::atomic instance;
7     static boost::mutex mtx;
8 };
9
10 Database& Database::instance()
11 {
12 Database* db = instance.load(boost::memory_order_consume);
13 if (!db)
14 {
15 boost::mutex::scoped_lock lock(mtx);
16 db = instance.load(boost::memory_order_consume);
17 if (!db)
18 {
19     db = new Database();
20     instance.store(db, boost::memory_order_release);
21 }
22 }
23 }
```

Ogólne podsumowanie singletona

Singletony nie są całkowicie złe, ale używane nieostrożnie mogą zepsuć testowalność i refaktoryzowalność aplikacji. Jeśli naprawdę musimy użyć singletona, spróbujmy unikać bezpośredniego używania go (jak w pisaniu `SomeComponent.getInstance().foo()`) i zamiast tego kontynuujmy określanie go jako zależności (np. argument konstruktora), gdzie wszystkie zależności są spełnione, z jednego miejsca w naszej aplikacji (np. inwersja kontenera kontrolnego).

Bibliografia

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides “Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku”
2. Dmitri Nesteruk “Design Patterns in Modern C++”
3. <https://gameprogrammingpatterns.com/>