Programowanie obiektowe

Laboratorium I

Wzorzec projektowy Singleton, zasada pojedynczej odpowiedzialności

Wzorzec projektowy Singleton

Wzorce projektowe (ang. *design patterns*) to typowe rozwiązania problemów często napotykanych podczas projektowania oprogramowania. Każdy z nich stanowi plan, który po odpowiednim dostosowaniu pomaga poradzić sobie z konkretnym problemem w projekcie. Nie można jednak wybrać wzorca i po prostu skopiować go do programu, jak bibliotekę czy funkcję zewnętrznego dostawcy. Wzorzec nie jest konkretnym fragmentem kodu, ale ogólną koncepcją pozwalającą rozwiązać dany problem. Postępując według wzorca można zaimplementować rozwiązanie, które będzie pasować do realiów konkretnego programu.

Wzorce często myli się z algorytmami, ponieważ obie koncepcje opisują typowe rozwiązanie jakiegoś znanego problemu. Algorytm jednak zawsze definiuje wyraźny zestaw czynności które prowadzą do celu, zaś wzorzec to wysokopoziomowy opis rozwiązania. Kod powstały na podstawie jednego wzorca może wyglądać zupełnie inaczej w różnych programach. Innymi słowy, można powiedzieć, że algorytm jest jak przepis kulinarny: ma wyraźnie określone etapy które trzeba wykonać w określonej kolejności by osiągnąć cel. Z kolei, wzorzec bardziej przypomina strategię: znany jest wynik i założenia, ale dokładna kolejność implementacji zależy od programisty.

Wzorce można skategoryzować według ich celu, bądź przeznaczenia. Mamy trzy główne grupy wzorców:

- **Wzorce kreacyjne** - wprowadzają elastyczniejsze mechanizmy tworzenia obiektów i pozwalają na ponowne wykorzystanie istniejącego kodu.
- **Wzorce strukturalne** - wyjaśniają jak składać obiekty i klasy w większe struktury, zachowując przy tym elastyczność i efektywność struktur.
- **Wzorce behawioralne** - zajmują się efektywną komunikacją i podziałem obowiązków pomiędzy obiektami.

*

Singleton jest kreacyjnym wzorcem projektowym, który pozwala zapewnić istnienie wyłącznie jednej instancji danej klasy. Ponadto daje globalny punkt dostępowy do tejże instancji. Wyobraź sobie, że masz już stworzony obiekt, ale po jakimś czasie potrzebujesz kolejnego. Zamiast otrzymać nowy, dostaniesz ten uprzednio stworzony. Zwróćmy uwagę na to, że takiego zachowania nie da się zaimplementować stosując zwykły konstruktor, ponieważ metody te z definicji muszą zawsze zwracać nowe obiekty.

Przykładem Singletona z innej dziedziny może być rząd jakiegoś państwa. Kraj może mieć wyłącznie jeden oficjalny rząd. Niezależnie od składu personalnego członków rządu, pojęcie “Rząd kraju X” jest uniwersalnym odwołaniem do organu władzy tego kraju.

Można postawić pytanie, dlaczego w ogóle ktokolwiek musiałby liczyć obiekty danej klasy? Otóż, najczęstszym powodem jest potrzeba kontroli dostępu do jakiegoś współdzielonego zasobu — na przykład bazy danych, lub pliku. Na przykład zmienne globalne, chociaż mogą być konieczne, to wiążą się też z poważnym ryzykiem przypadkowego nadpisania ich zawartości i tym samym awarii programu. Dla porównania, wzorzec Singleton pozwala skorzystać z jakiegoś obiektu w dowolnym miejscu programu ale dodatkowo zapewnia też ochronę tego obiektu przed działaniami innego kodu.

Wszystkie implementacje wzorca Singleton współdzielą poniższe dwa etapy:

- Ograniczenie dostępu do domyślnego konstruktora przez uczynienie go prywatnym, aby zapobiec stosowaniu operatora new w stosunku do klasy Singleton.
- Utworzenie statycznej metody kreacyjnej, która będzie pełniła rolę konstruktora. Za kulisami, metoda ta wywoła prywatny konstruktor, aby utworzyć instancję obiektu i umieści go w polu statycznym klasy. Wszystkie kolejne wywołania tej metody zwrócą już istniejący obiekt.

```
class Singleton
{
    public:
        static Singleton& getInstance()
        {
          // Inicjalizacja statycznego obiektu.
          // Obiekt zostanie utworzony przy pierwszym wywołaniu tej metody
          // i tylko wtedy nastąpi inicjalizacja przy pomocy konstruktora.
          // Każde następne wywołanie jedynie zwraca referencję tego obiektu.
          // Obiekt zostanie usunięty w momencie zakończenia działania programu
             static Singleton instance;
             return instance;
        }
    private:
        Singleton() = default;

    public:
        Singleton(const Singleton&) = delete;
        Singleton& operator=(const Singleton&) = delete;
        Singleton(Singleton&&) = delete;
        Singleton& operator=(Singleton&&) = delete;
};
```

W powyższej implementacji warto zwrócić uwagę na następujące rzeczy. Po pierwsze, poprzez przeniesienie do sekcji `private` oraz zastosowanie słowa kluczowego `delete` uniemożliwiono stworzenie w funkcji `main()` obiektu za pomocą konstruktora oraz wykonanie takich operacji jak przypisanie lub kopiowanie obiektu. Pod drugie, użycie statycznego pola w funkcji `GetInstance()` powoduje, że obiekt `Singleton` jest stworzony tylko jeden raz podczas pierwszego wywołania tej funkcji. Podczas kolejnych wywołań funkcji `GetInstance()` zostanie jedynie zwrócona referencja do istniejącego już obiektu. Gdy program ma dostęp do klasy Singleton, to będzie mógł wywołać jej statyczną metodę i tym samym za każdym razem gdy wywoła tą metodę to otrzyma ten sam obiekt.

Wzorzec Singleton można zastosować w następujących przypadkach. Bezpośredni dostęp do zasobów sprzętowych. W wielu takich przypadkach konieczne jest zapewnienie odpowiedniej sekwencji operacji na sprzęcie co można łatwo osiągnąć implementując tę logikę jako Singleton. Obsługa logów, plików konfiguracyjnych lub pamięci cache. Jeśli aplikacja potrzebuje zapisywać plik z logami to kod służący do zapisywanie komunikatów w logach dobrze zaimplementować w Singletonie co zapewnia poprawną strukturę logów. Podobnie jak w przypadku

logów, implementacja kodu odczytującego i zapisującego dane konfiguracyjne w postaci Singletona zapewnia utrzymanie spójności i poprawności danych w plikach konfiguracyjnych.

Zasada pojedynczej odpowiedzialności

Zasada pojedynczej odpowiedzialności (ang. *SRP - Single Responsibility Principle*) – można ją zdefiniować następująco: klasa powinna mieć jeden główny cel, być odpowiedzialna za jedną konkretną rzecz, jedną funkcjonalność. Dlaczego? Ponieważ łatwiej jest wtedy zrozumieć i znacznie łatwiej utrzymywać i naprawiać błędy w kodzie. Innymi słowy można powiedzieć: rób jedną rzecz i rób ją dobrze. Reguła ta zwiększa zazwyczaj liczbę klas w programie (co na pierwszy rzut oka wydaje się niekorzystne). Jednak korzyścią jest to, że eliminuje skomplikowane, rozbudowane klasy odpowiedzialne za wiele rzeczy.

Spójrzmy na klasę przedstawioną poniżej:

```
class BadEmployee
{
      string femail;
      int fsalary;
      string fname;
      string fsurname;

      public: BadEmployee (string name, string surname, string email, int salary)
      {
        fsalary = salary;
        femail = email;
        fname = name;
        fsurname = surname;
      }
}
```

Klasa ta opisuje pracownika. Zawiera imię, nazwisko, wiek oraz adres e-mail. Mając taką klasę, zapewne chcielibyśmy sprawdzać poprawność danych pracownika. Dodajmy zatem metody sprawdzające e-mail.

```
class BadEmployee
{
    string _email;
    int _salary;
    int _salaryThreshold = 2500;
    string _name;
    string _surname;

    public: BadEmployee (string name, string surname, string email, int salary)
    {
        fsalary = salary;
        femail = email;
        fname = name;
        fsurname = surname;
    }

    private: string ValidateEmail ()
    {
      // kod do walidacji email
    }
}
```

Zatem, mamy pracownika, potrafimy sprawdzić poprawność jego danych. Nie bez powodu klasa ta został nazwana `BadEmployee`, bo rzeczywiście jest to zły pracownik. Otwarcie łamie zasadę SRP, ponieważ klasa ta jednocześnie przechowuje dane pracownika oraz jest odpowiedzialna za sprawdzenie poprawności tych danych. Zgodnie z zasadą pojedynczej odpowiedzialności, walidacja powinna zostać oddelegowana do innej klasy, przez co odpowiedzialności zostaną jednoznacznie rozdzielone. Spójrzmy na kod poniżej:

```
class GoodEmployee
{
      string femail;
      int fsalary;
      ValidityTool fvalidityTool;
      string fname;
      string fsurname;
      public: GoodEmployee (string name, string surname, string email, byte
salary)
      {
        fsalary = salary;
        femail = email;
        fname = name;
        fsurname = surname;
      }

      private: string ValidateEmail ()
      {
        return fvalidityTool.ValidateEmail(femail);
      }

      bool IsAdult ()
      {
        return fvalidityTool.CheckSalary(salary);
      }
}

class ValidityTool
{
      int fsalaryThreshold = 2500;
      public: string ValidateEmail (string email)
      {
        // kod do walidacji email
      }
}
```

W powyższym kodzie, dodaliśmy klasę `ValidityTool`, która jest odpowiedzialna za sprawdzenie poprawności danych. W klasie dobrego pracownika oddelegowujemy zadanie walidacji do klasy `ValidityTool`. Dzięki temu zyskaliśmy podział odpowiedzialności – każda klasa ma swój jasno określony cel. Klasa `GoodEmployee` jest typowym przykładem klasy modelu. Tego typu klasy powinny tylko przechowywać dane, natomiast inne zadania (np. walidacja) powinny zostać oddelegowane do innych klas. Podsumowując, zgodnie z zasadą pojedynczej odpowiedzialności, należy pamiętać o tym, aby tworzone klasy miały jedno jasno określone zadanie.

Zadania[1]

Zadanie 1

1. Stwórz klasę `ComplexNumber`, która ma reprezentować liczbę zespoloną. Deklarację klasy umieść w pliku nagłówkowym .h, zaś implementację metod w pliku .cpp. W klasie należy umieścić następujące metody:

[1] Do każdego z zadań najlepiej stworzyć oddzielny projekt w VisualStudio.

a. publiczny konstruktor przyjmujący wartość rzeczywistą i urojoną,

b. konstruktor kopiujący (jego argumentem jest stała referencja do obiektu tej samej klasy),

c. gettery i settery dla części rzeczywistej i urojonej,

d. `ComplexNumber conjugate()` – zwracającą sprzężenie liczby,

e. `ComplexNumber abs()` – zwracającą moduł liczby (jako liczbę zespoloną)

f. operatory: `+`, `==`, operator wyjścia.

g. Następnie napisz, krótki program demonstrujący użycie tej klasy.

Zadanie 2

2. Wykorzystując wzorzec Singleton napisz klasę `FileLogger`, która będzie służyć do zapisywanie komunikatów w logach umieszczonych w pliku. Klasa powinna Zawierać następujące metody:
   a. `void set_target( const string& target_file )`, która ustawia docelowy plik,

   b. `void log_error( const string& message )`, która zapisuje do pliku log komunikat o błędzie,

   c. `void log_debug( const string& message )` - zapisuje informacje debugowania do pliku

   d. `void log_info( const string& message )` - zapisuje pozostałe informacje.

   W funkcji `main()` napisz program demonstrujący użycie klasy `FileLogger`. Informacja o błędzie powinna być zapisana w następującym formacie: `[error] bieżąca_data komunikat_o_błędzie znak_nowej_linii`. W podobnym formacie powinny zapisywać komunikaty metody `log_debug()` oraz `log_info()`. Należy zwrócić uwagę na to, iż każda operacja zapisu komunikatu do pliku log powinna dołączyć nowy komunikat do końca pliku a nie nadpisywać cały plik. Można to osiągnąć posługując się flagą `fstream::out | fstream::app` podczas otwierania pliku log. W C++, bieżącą datę można pobrać w następujący sposób: `time_t current_time=time(nullptr); string date=ctime(&current_time );`.

Zadanie 3

3. Napisz program, który obliczy sumę dwóch dodatnich wartości całkowitych. Suma do obliczenia jest podana jako a ciąg, np. “123 + 37”, “78+ 99” itd. Program powinien działać następująco:

   a. Poproś użytkownika o wprowadzenie ciągu;

   b. Wyeliminuj dodatkowe spacje z ciągu;

   c. Wyodrębnij dwa podciągi reprezentujące dwa argumenty, takie jak „123” i „37”;

   d. Oblicz wartości całkowite tych dwóch podciągów, więc „123” jest przekształcane na wartość 123 itd.

   e. Wydrukuj sumę dwóch wartości;

   f. Rozszerz swoje rozwiązanie o obsługę potencjalnych błędów.

Zadanie 4

4. Napisz program, który losuje liczbę całkowitą z przedziału od 0 do 100, a następnie prosi użytkownika o odgadnięcie wylosowanej liczby i wprowadzenie jej z klawiatury. Następnie program wyświetla komunikat,

czy wprowadzona liczba jest większa, czy mniejsza od wylosowanej. Gra kończy się w wypadku wprowadzenia poprawnej liczby lub przekroczenia maksymalnej liczby prób, która wynosi 5.

Zadanie 5

5. Napisz program, który wczytuje ze standardowego wejścia nieujemną liczbę całkowitą n ($n > 2$) i wypisuje na standardowym wyjściu największą liczbę k taką, że k dzieli n przy założeniu, że $k < n$. Algorytm wyszukiwania liczby k spełniającej powyższe warunki umieść w oddzielnej funkcji.

Zadanie 6

6. W oparciu o wzorzec Singleton, stwórz klasę, która przechowuje ustawienia w aplikacji. Klasa powinna umożliwiać dodanie, odczyt ustawień na zasadzie klucz wartość. Dodatkowo klasa umożliwia zapis i ładowanie ustawień do/z pliku.