Projektowanie obiektowe oprogramowania
Zestaw 5

Wzorce strukturalne

2024-03-19

Liczba punktów do zdobycia: **6/36**
Zestaw ważny do: 2024-04-09

1. (**1p**) (**Facade**) Dostarczyć implementacji dla fasady:

```
class SmtpFacade {
public void Send( string From, string To,
                      string Subject, string Body,
                      Stream Attachment, string AttachmentMimeType );
}
```

   W implementacji użyć klas z biblioteki standardowej, realizujących wysyłkę za pomocą SMTP.

2. (**1p**) (**Decorator**) Zaimplementować klasę strumienia `CaesarStream`, który będzie działał jak dekorator strumieni (wymaganym parametrem konstruktora będzie inny strumień, na którym operował będzie `CaesarStream`) przy czym odczytując i zapisując dane będzie aplikował przesunięcie cezariańskie poszczególnych bajtów danych. Przykład użycia:

```
FileStream fileToWrite = File.Create( ... );
CaesarSteram caeToWrite = new CaesarStream( fileToWrite, 5 );
// 5 to przesunięcie

caeToWrite.Write( ... );

FileStream fileToRead = File.Open( ... );
CaesarStream caeToRead = new CaesarStream( fileToRead, -5 );
// -5 znosi 5

caeToRead.Read( ... );
```

3. (**1p**) (**Adapter**) Dostarczyć adapter rozwiązujący zadanie:

   `https://www.wiktorzychla.com/2009/05/c-puzzle-no15-intermediate.html`

4. (**1p**) (**Bridge**) Dana jest klasa rejestru pracowników.

```
public class PersonRegistry
{
    /// <summary>
    /// Pierwszy stopień swobody - różne wczytywanie
    /// </summary>
    public List<Person> GetPersons()
    {
        return new List<Person>()
                    {
```

```
                    new Person(),
                        new Person()
                };
        }

        /// <summary>
        /// Drugi stopień swobody - różne użycie
        /// </summary>
        public void NotifyPersons()
        {
            foreach ( Person person in GetPersons() )
                Console.WriteLine( person );
        }
    }

    public class Person { }
```

Wiadomo też, że że w wyniku analizy funkcjonalnej zidentyfikowano dwa stopnie swobody tej klasy.

Jeden polega na możliwości wczytywania rejestru z różnych źródeł (XML, baza danych). Drugi polega na możliwości powiadamiania pracowników w różny sposób (mail, SMS).

W powyższym przykładzie, klasa sama sobie konstruuje dane i sama wie jak je prezentować (**WriteLine**). Gdyby stopień swobody był jeden, można by po prostu odpowiadającą mu metodę uczynić wirtualną i dostarczać podklas klasy **PersonRegistry**, które przeciążałyby stosowną metodę.

W sytuacji gdy stopnie swobody są dwa (lub więcej) rozwiązanie z dostarczaniem podklas nie sprawdzi się, ponieważ każda kombinacja implementacji szczegółowej dla niezależnych funkcjonalności prowadziłaby do innej klasy potomnej. Na przykład tu:

- implementacja która dane pobiera z XML a wyprowadza na mail
- implementacja która dane pobiera z XML a wyprowadza na SMS
- implementacja która dane pobiera z XML a wyprowadza na drukarkę
- implementacja która dane pobiera z bazy danych a wyprowadza na mail
- implementacja która dane pobiera z bazy danych a wyprowadza na SMS
- ...

Pokazać jak Bridge rozwiązuje ten problem przez wyniesienie jednej z odpowiedzialności na zewnątrz.

Przygotować **dwie** różne implementacje mostu, w których jeden ze zidentyfikowanych stopni swobody stanie się wstrzykiwaną implementacją, a drugi pozostanie jako możliwy do pokrycia w podklasie klasy abstrakcji (rejestru).

5. (**2p**) (**Proxy**) Na wykładzie wymieniono kilka rodzajów proxy

   - Proxy typu **Retry** – proxy z polityką automatycznego ponawiania kolejnych nieudanych wywołań
   - Proxy typu **Circuit Breaker** – czyli wykrywanie pewnej liczby nieudanych wywołań i blokowanie kolejnych nieudanych wywołań
   - Proxy typu **Rate Limiter** – czyli kontrolowanie liczby wywołań w czasie
   - Proxy typu **Fallback** – czyli obsługa wariantu awaryjnego w sytuacji gdy oryginalne wywołanie nie udaje się

- Proxy typu **Hedging** – czyli automatyczne przełączenie na wskazany wariant zapasowy gdy oryginalne wywołanie trwa zbyt długo

Dodatkowo, można wyobrazić sobie inne, bardziej specyficzne proxy, na przykład **proxy logujące**, które loguje wywołania metod (data, nazwa metody i wartości parametrów) oraz zakończenia wywołań (data, wartość zwracana z metody).

Przygotować obiekt jakiegoś typu (dowolny). Przygotować fabrykę tworzącą instancje zgodnego typu (ten sam interfejs lub podklasa wyjściowej klasy) które w rzeczywistości nie są obiektami wyjściowego typu, tylko są proxy do wyjściowego typu.

Zadanie jest za 2 punkty, bo należy wybrać sobie dwa dowolne wzmiankowane wyżej rodzaje proxy i przygotować dwie implementacje.

Uwaga. Zadanie może być łatwe lub trudne w zależności od tego które 2 rodzaje się wybierze i czy implementacja będzie samodzielna czy też użyje się którejś z wzmiankowanych na wykładzie gotowych implementacji (można ich użyć).

Uwaga 2. W przypadku samodzielnej implementacji, nie trzeba rozwiązywać problemu w ogólności, czyli fabryka proxy nie musi działać dla *dowolnego* typu wejściowego (nieznanego w trakcie przygotowywania fabryki), tylko może być dedykowana do jakiegoś jednego, konkretnego typu.

Wiktor Zychla