## Wzorce projektowe „Bandy czworga”



## Czym są wzorce projektowe „Bandy Czworga”?

- Wzorce projektowe „Bandy Czworga” to wzorce projektowe opisane przez czterech autorów (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides) w „Design Patterns Elements of Reusable Object-Oriented Software” w 1995 roku.
- Opracowanie zawiera 23 popularne wzorce projektowe oceniane jako niezwykle przydatne podczas projektowania obiektowego.
- Przyjmuje się, że w praktyce z 23 przedstawionych wzorców ok. 15 jest szeroko stosowanych.
- Wzorce projektowe podzielone są na następujące kategorie:
  - Wzorce kreacyjne (creational patterns): Factory, Builder, Factory Method, Prototype, Singleton,
  - Wzorce strukturalne (structural patterns): Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy,
  - Wzorce behawioralne (behavioral patterns): Chain of Responsability, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

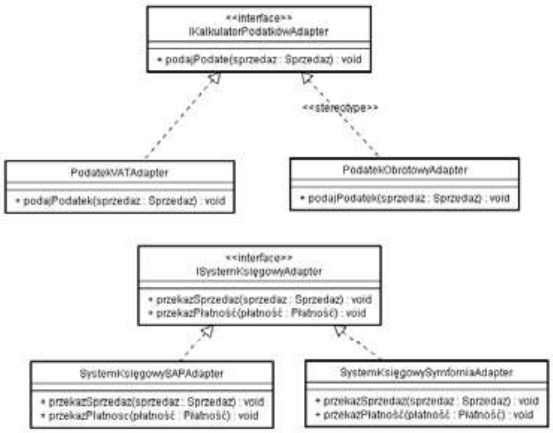
## Adapter: założenia

- **Problem:** W jaki sposób należy rozwiązać problem niekompatybilnych interfejsów (sprzęgów)? W jaki sposób należy udostępniać stabilny interfejs (sprzęg) do podobnych komponentów o różnych interfejsach.
- **Rozwiązanie:** Zalecane jest wprowadzenie nowego obiektu pośredniego (adapter), który konwertuje niekompatybilny interfejs w interfejs, który jest oczekiwany przez klienta.
- Wzorzec umożliwia współpracę dwóch klas w sytuacji, w której klient nie ma możliwości użycia wprost interfejsu danej klasy. Ma to miejsce w sytuacji w której klasa klient współpracuje z wieloma aplikacjami zewnętrznymi do których dostęp jest możliwy przy pomocy różnych (i niekompatybilnych) interfejsów (różne API).



## Adapter: przykład

- W programie wielozadaniowy POS zachodzi potrzeba współpracy z wieloma systemami (aplikacjami) zewnętrznymi: kalkulatory podatków, systemy autoryzacji kart kredytowych, system księgowy, itp. Każdy z tych systemów udostępnia różny interfejs w różnych technologiach (RMI, SOAP).



## Adapter: dyskusja

- Adaptery są budowane przy wykorzystaniu składniowego mechanizmu interfejsów (Java), bądź składniowego elementu polimorfizmu (C++, Java).
- Korzystanie z wzorca Adapter jest zalecane w następujących sytuacjach:
  - Niezbędne jest wykorzystanie istniejącej klasy, której interfejs nie może być wykorzystany przez klasę klienta,
  - Chcemy utworzyć klasę wykorzystywaną wielokrotnie (reusable), która współpracuje z niepowiązanymi oraz niedospecyfikowanymi klasami tzn. z klasami, które nie mają identycznych interfejsów.

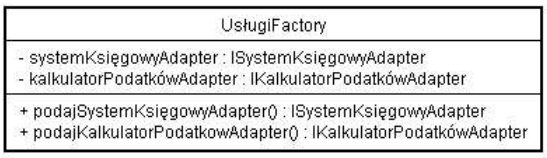


## Factory: założenia i przykład

- **Problem:** Kto powinien być odpowiedzialny za tworzenie obiektów w sytuacji w której należy uwzględnić dodatkowe ograniczenia np. duża złożoność tworzonych struktur.
- **Rozwiązanie:** Wprowadzenie obiektu Factory, który jest odpowiedzialny za tworzenie rozpatrywanych obiektów.
- Zalety z wprowadzonego rozwiązania:
  - Przekazanie odpowiedzialności tworzenia skomplikowanych struktur do specjalizowanych i spójnych obiektów,
  - Ukrywanie złożoności procesu tworzenia skomplikowanych struktur,
  - Możliwość wprowadzenia technik optymalizacji zarządzania pamięcią (cachowanie).

## Factory: przykład

- **Przykład**: W programie wielozadaniowy POS konieczne jest utworzenie adapterów *PodatekVATAdapter*, *SystemKsięgowySAPAdapter*. Kto powinien być odpowiedzialny za tworzenie tych obiektów?
- Przypisanie odpowiedzialności tworzenia obiektów do klas z modelu wiedzy dziedzinowej doprowadzi do przekroczenia zakresu odpowiedzialności tych klas (wykonywanie zadań związanych z rozpatrywaną dziedziną np. obliczanie podatków).
- Zaleca się rozdzielanie odpowiedzialności zgodnie z grupami tematycznymi i utworzenie obiektu *UsługiFactory* typu Factory .






## Singleton: założenia

- **Problem:** Konieczne jest, aby istniała dokładnie jedna instancja danej klasy („singleton”) oraz aby była ona globalnie dostępna. **Rozwiązanie:** Wprowadzenie metod statycznych do klasy, która zwraca instancje klasy „singleton”.
- Wzorzec Singleton zakłada, że klasa jest sama odpowiedzialna za dostarczanie swojej jedynej instancji! Klasa jest tak skonstruowana, że gwarantuje, że inne jej instancje nie będą utworzone.
- Wzorzec Singleton jest zalecany, gdy w programie ma być dostępna jedynie jedna instancja danej klasy i musi być ona dostępna dla klientów ze znanego wszystkim miejsca.
- Przykładem klasy tego typu może być np. printer spooler. W większości programów istnieje tylko jedna instancja tej klasy, która powinna być dostępna z całego programu.

## Singleton: realizacja

- Klasa wprowadza metodę *getInstance*, która jako jedyna może być wykorzystana do uzyskania instancji „singleton”. W większości języków programowania metoda ta będzie metodą statyczną (C++, Java). Jako statyczny wprowadzony zostaje również atrybut przechowujący instancję.
- Definicja klasy wygląda następująco:

```
public class Singleton {
    private static Singleton instance;
    public static Singleton getInstance() {
      if (instance == null)
                instance =new Singleton();
      return instance;
     }
}
```



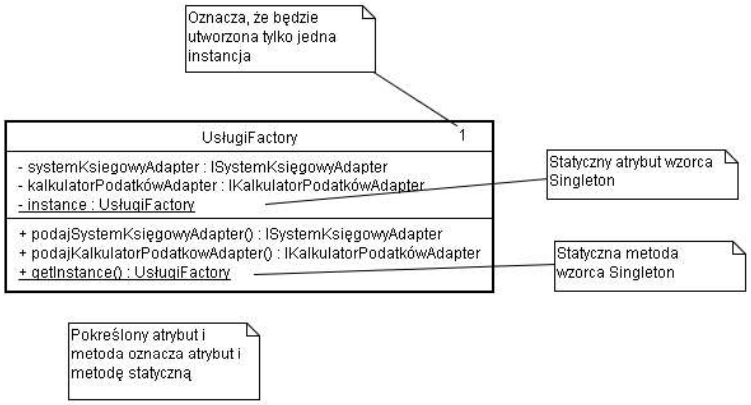
## Singleton: przykład (1)

```
public class Singleton {
    private static Singleton instance;
    private int Liczba = 0;

    public static Singleton getInstance() {
      if (instance == null)
                instance = new Singleton();
      return instance;
    }

    public int obliczeniaDowolne(){
       Liczba = Liczba +1;
       return Liczba;
    }
    }

class Start
{
   public static void main(String args[])
   {
    int i;
    i = Singleton.getInstance().obliczeniaDowolne();
    System.out.println(i);
     i = Singleton.getInstance().obliczeniaDowolne();
    System.out.println(i);
   }
}
```

Odwołanie się do jedynej instancji klasy

## Singleton: przykład (2)

- Wzorzec Singleton powinien być wykorzystywany do tworzenia obiektów typu Factory – w każdym programie powinien być jeden taki obiekt.
- W programie wielozadaniowy POS klasa *UsługiFactory* powinna być klasą utworzoną zgodnie z wzorcem Singleton:






## Singleton: dyskusja (1)

- Wprowadzenie wzorca Singleton jest lepszym rozwiązaniem niż wykorzystanie samej zmiennej globalnej, bądź obiektu statycznego:
  - Zmienne globalne są co prawda globalnie dostępna, ale nie można zapobiec wielokrotnemu ich tworzeniu,
  - Podczas tworzenia programu (faza inicjalizacji zmiennych globalnych) możemy nie posiadać pełnej informacji potrzebnej podczas tworzenia takiej instancji (singleton może wymagać danych, które zostaną obliczone w trakcie działania programu).
- Rozwiązania przyjęte we wzorcu (tzn. wprowadzenie metody statycznej zwracającej instancję klasy singleton) jest bardziej korzystne niż wprowadzenie wszystkich metod jako statyczne (np. uczynienie jako statyczną metodę *podajSystemKsiegowyAdapter()*:
  - Możliwe jest tworzenie podklas na bazie klasy singleton (gdyby wszystkie metody były statyczne nie jest to możliwe – metody statyczne w większości języków nie są polimorficzne,
  - Klasa nie zawsze jest klasą singleton we wszystkich kontekstach użycia; przyjęte rozwiązanie umożliwia proste wprowadzanie zmian.

## Singleton: dyskusja (2)

- We wzorcu został wykorzystany mechanizm tzw. późnej inicjalizacji (lazy initialization) w przeciwieństwie do możliwej tzw. pochopnej inicjalizacji (eager initialization):

**Późna inicjalizacja:**

```
public class Singleton {
      private static Singleton instance;
      public static Singleton getInstance() {
      if (instance = null)
                instance:=new Singleton;
      return instance;
       }
}
```

**Pochopna inicjalizacja:**

```
public class Singleton {
      private static Singleton instance =new Singleton;
      public static Singleton getInstance() {
                return instance;
       }
}
```

- Późna inicjalizacja jest preferowana ze względu na:
  - Instancja nie jest tworzona w sytuacji, w której nie jest wykorzystywana; unikamy zbędnej pracy tworzenia niepotrzebnych struktur,
  - Inicjalizacja w *getInstance* może zostać wykorzystana do zrealizowanie złożonych i warunkowych operacji.



## Strategy: założenia

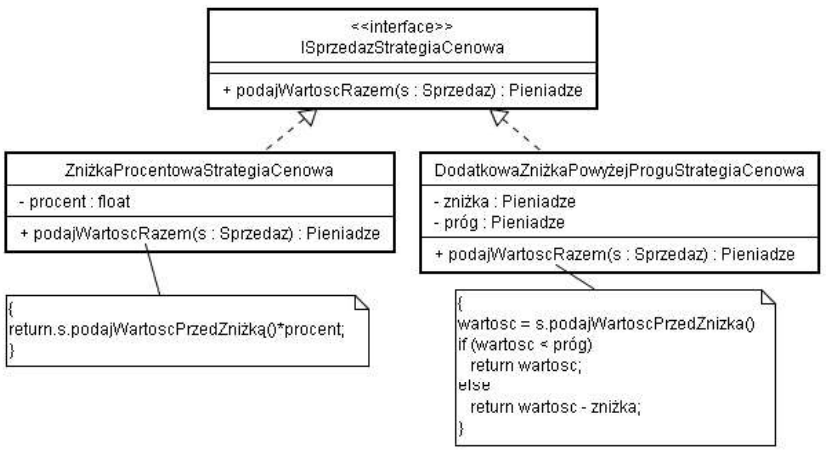
- **Problem:** W jaki sposób modelować klasy, które są powiązane, ale różnią się co do realizowanych algorytmów? W jaki sposób modelować, aby istniała możliwość elastycznej zmiany tych algorytmów?
- **Rozwiązanie:** Zdefiniować każdy z algorytmów w oddzielnych klasach, ale posiadających wspólny interfejs.
- Wzorzec Strategy znajduje zastosowanie gdy nie jest korzystne umieszczanie wszystkich algorytmów w jednej klasie:
  - Jeżeli umieścilibyśmy dany algorytm w klasie klienckiej (tej która z nich korzysta), klasa ta stałaby się większa i trudna w utrzymaniu, w szczególności wtedy gdy wykorzystywałaby wiele algorytmów,
  - Rożne algorytmy są wykorzystywane w różnym czasie - nie jest korzystne umieszczanie wszystkich algorytmów w jednej klasie w sytuacji w której z nich nie korzystamy,

## Strategy: przykład (1)

- W programie wielozadaniowy POS konieczne jest uwzględnienie obsługi złożonej polityki cenowej sklepu (zniżki dla różnych grup klientów). Polityka ta zmienia się w czasie (w jednym okresie zniżki mogą wynosic 5%, w innym, 15%).
- Rozwiązaniem problemu jest zastosowanie wzorca Strategy poprzez utworzenie licznych klas implementujących poszczególne algorytmy posiadających polimorficzną metodę *podajWartoscRazem*.
- Każda z metod *podajWartoscRazem* posiada jako parametr klasę *Sprzedaz*, aby przed obliczeniem zniżki mogła pobrać sumę sprzedaży przed zniżką (metoda *podajWartoscPrzedZnizka* z klasy *Sprzedaz*). Implementacja każdej z metod *podajWartoscRazem* będzie różna – uwzględniane będą różne algorytmy obliczania zniżek.
- Przyjęte rozwiązanie pozwala uniknąć sytuacji, w której metody z różnymi rodzajami zniżek są umieszczane wprost w klasie *Sprzedaż*!

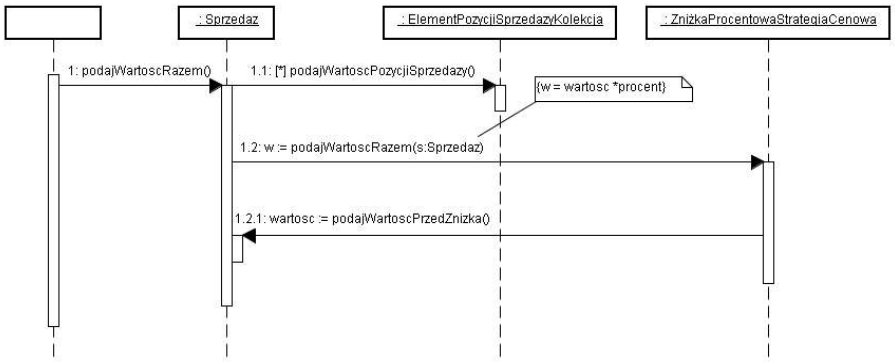


## Strategy: przykład (2)



## Strategy: dyskusja (1)

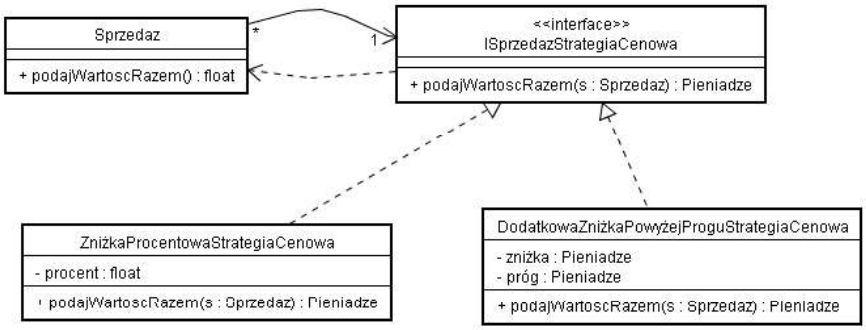
- Obiekt realizujący wzorzec Strategy jest zawsze powiązany z tzw. obiektem kontekstowym (context object), czyli obiektem klienckim.
- W rozpatrywanym przykładzie obiektem tym jest instancja klasy *Sprzedaż*. Obiekt ten deleguje część pracy na obiekt Strategy w sytuacji w której otrzymuje zadanie obliczenia sumy zakupów (metoda *podajWartoscRazem*).





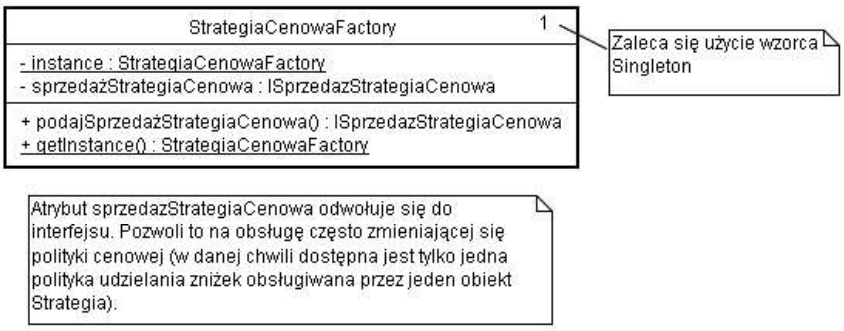

## Strategy: dyskusja (2)

- Rozwiązanie wykorzystane w przykładzie tzn. przekazanie instancji klasy *Sprzedaż* do obiektu Strategy jest rozwiązaniem często spotykanym. Oznacza to, że obiekt Strategy posiada widzialność parametryczną do obiektu kontekstowego. Ponadto obiekt kontekstowy posiada widoczność atrybutową do obiektu Strategy.



## Strategy: dyskusja (3)

- Wzorzec Factory zaleca, aby za tworzenie obiektów Strategy był odpowiedzialny obiekt Factory. Ze względu na wzorzec High Cohesion zaleca się, aby były tworzone różne obiekty Factory realizujące różne zadania.
- W związku z powyższym w rozpatrywanym przykładzie zaleca się utworzenie nowej klasy *StrategiaCenowaFactory* służąca do tworzenia wszystkich obiektów typu Strategy.






## Strategia: dyskusja (4)

- Podczas tworzenia instancji *Sprzedaz* kierowane będzie pytanie do instancji *StrategiaCenowaFactory*, która zwróci obiekt obsługujący aktualnie obowiązującą strategię cenową.