Wzorce projektowe „Bandy czworga” (c.d.)

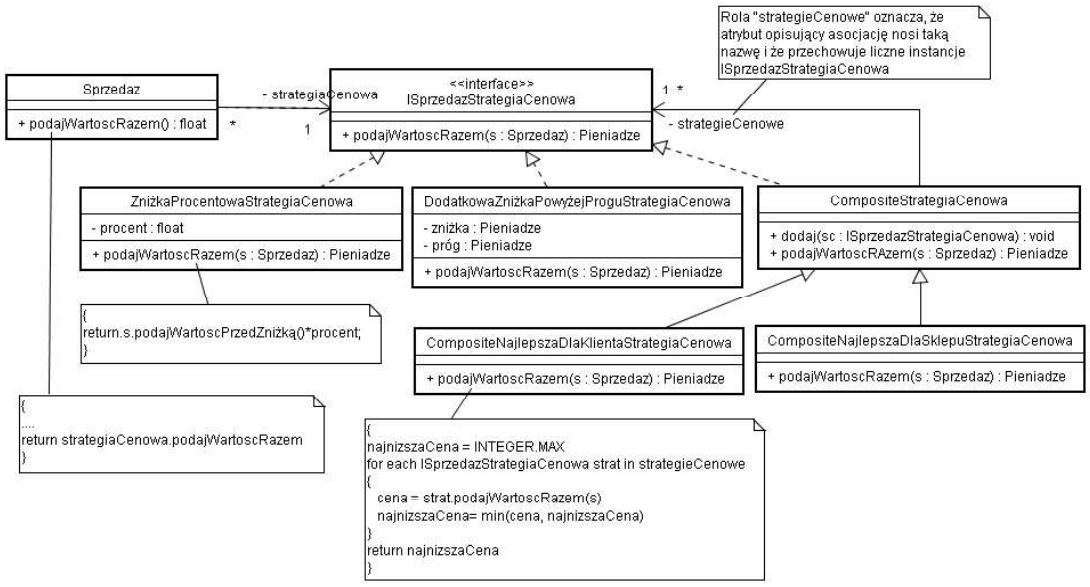
Composite: założenia

- **Problem:** W jaki sposób przetwarzać grupy obiektów (obiekty grupujące inne obiekty) w taki sam sposób jak obiekty występujące samodzielnie?
- **Rozwiązanie:** Zdefiniować klasy posiadające ten sam interfejs dla grup obiektów i obiektów występujących samodzielnie.
- Wzorzec Composite znajduje zastosowanie gdy:
  - Chcemy reprezentować hierarchie obiektów typu całość-część,
  - Chcemy aby klient nie postrzegał różnic pomiędzy obiektami grupującymi różne obiekty, a obiektami występującymi samodzielnie.
- Rozwiązanie jest korzystne w sytuacji, w której konieczne jest obsłużenie różnych strategii (algorytmów), które mogą pozostawać w sprzeczności ze sobą.

Composite: przykład (1)

- W programie wielozadaniowy POS konieczne jest obsłużenie sytuacji w której funkcjonuje jednocześnie wiele strategii cenowych mogących wykluczać się między sobą. Np. rozpatrujemy sytuacje, w której na cenę mogą wpływać jednocześnie trzy czynniki:
  - Dzień tygodnia (początek tygodnia tańczy, niż weekend),
  - Rodzaj klienta (klient stały ma zniżki),
  - Rodzaj produktu (system zniżek dla danego produktu).
- W jaki sposób mamy zaprojektować system, aby obiekt *Sprzedaż* nie musiał wiedzieć o tym, czy w danej chwili obowiązuje jedna czy wiele strategii cenowych?
- Rozwiązaniem jest utworzenie klasy *CompositeNajlepszaDlaKlienta StrategiaCenowa*, która implementuje interfejs *ISprzedażStrategia Cenowa* i sama zawiera również instancje klasy *ISprzedażStrategia Cenowa*!

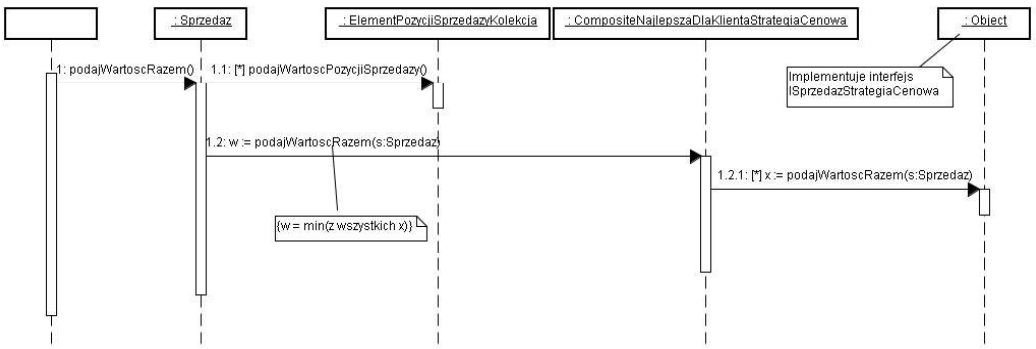
Composite: przykład (2)



Composite: przykład (3)

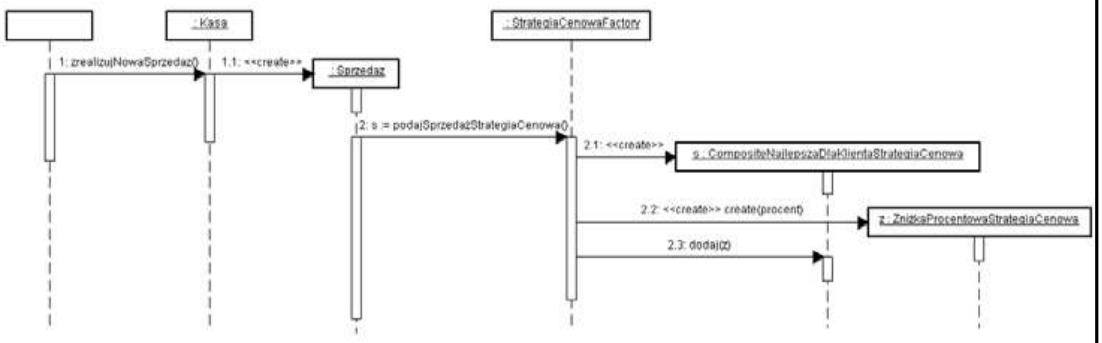
- Charakterystyczne dla wzorca jest to, że obiekt grupujący (zewnętrzny), jak i grupowany (wewnętrzny) implementują ten sam interfejs.
- Dzięki przyjętemu rozwiązaniu (*CompositeNajlepszaDlaKlienta StrategiaCenowa* posiada interfejs *ISprzedazStrategiaCenowa*) instancja klasy *Sprzedaz* może przetwarzać obiekty złożone (*CompositeNajlepszaDlaKlientaStrategiaCenowa*) grupujące inne obiekty, bądź obiekty indywidualne (*ZniżkaProcentowaStrategia Cenowa*). Dla instancji *Sprzedaz* nie jest istotne, czy strategia cenowa jest złożona, czy atomowa.

Composite: przykład (4)



Composite: przykład (5)

- Obiekty wewnętrzne dodawane są do obiektu typu *Composite* w momencie w którym realizowana jest sprzedaż (tworzona jest instancja klasy *Sprzedaż*)

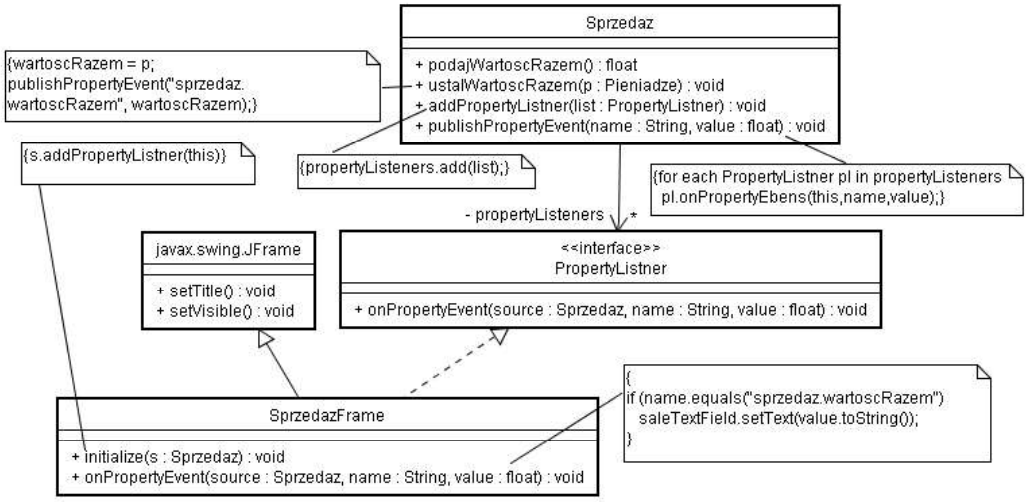


Observer: założenia

- **Problem:** W jaki sposób modelować sytuację, w której obiekty (słuchacze) zainteresowane śledzeniem zmian zachodzących w stanie pewnego obiektu (publikator) chcą reagować w różny sposób na zmianę generowaną przez ten obiekt.
- **Rozwiązanie:** Zalecane jest zdefiniowanie interfejsu „listener” (lub inaczej „subscriber”), który powinien być implementowany przez obiekty słuchacze. Publikator powinien mieć możliwość rejestrowania słuchaczy i informowania ich o zachodzących zmianach.
- Wzorzec znajduje zastosowanie do odświeżania zawartości GUI w odpowiedzi na zmianę określonych danych.
- Dzięki wzorcowi nie jest łamana zasada rozdzielenia Modelu i Widoku – nie ma potrzeby, aby klasy z modelu wysyłały komunikaty do widoku w odpowiedzi na zmianę wartości danych (w praktyce rozdzielenie Modelu i Widoku polega to na tym, że obiekty z modelu nie znają obiektów odpowiedzialnych za tworzenie interfejsu np. obiektów w Java Swing, C++ QT itp.)

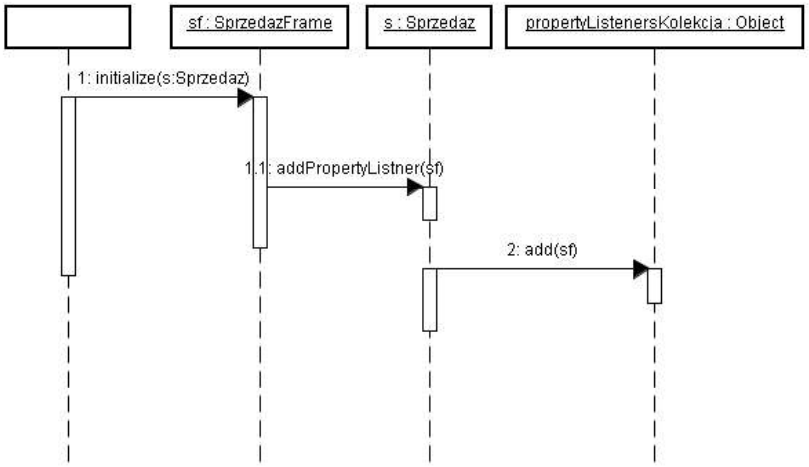
Observer: przykład (1)

- W programie wielozadaniowy POS wprowadzamy interfejs *PropertyListner*, który będzie wykorzystany do aktualizacji okien wyświetlających dane o wartości sprzedaży razem.



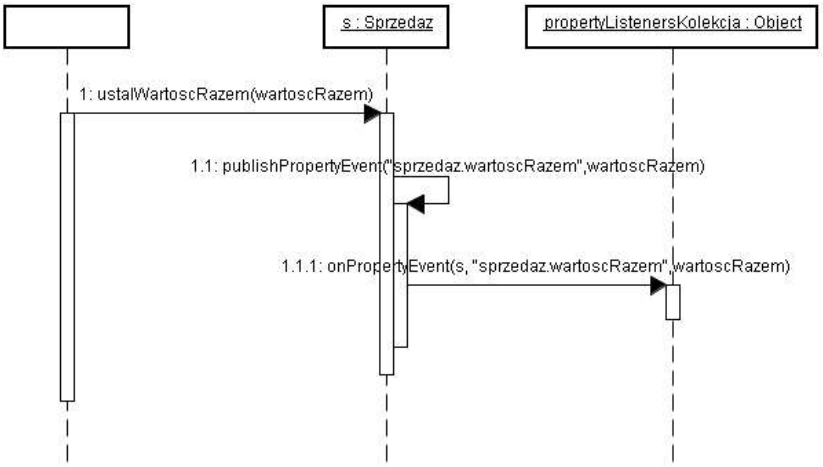
Observer: przykład (2)

- Po dodaniu obiektu słuchacza do publikatora obiekt *Sprzedaz* nie zna obiektu *SprzedazFrame*. Zachowany jest wzorzec Low Coupling.



Observer: przykład (3)

- W sytuacji, w której w obiekcie *Sprzedaz* następuje zmiana powiadamiane są wszystkie obiekty słuchacze.



Observer: dyskusja

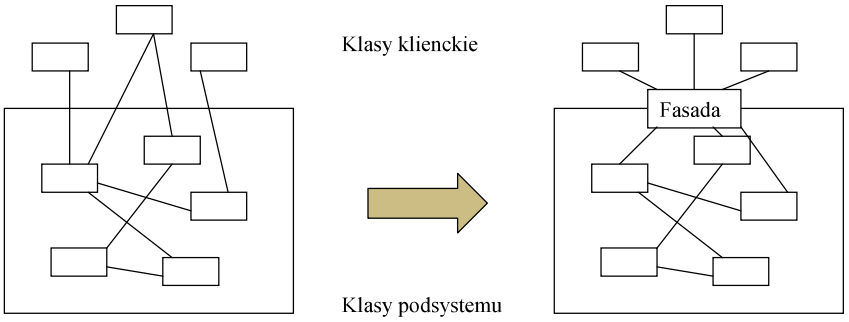
- Wzorzec projektowy Observer nosił również nazwę „publish-subscribe” i wywodzi się języka Smalltalk.
- Wzorzec ten znajduje szerokie zastosowanie w technologiach takich jak Java AWT, Java Swing, czy Mircosoft .NET.
- Model ten może być wykorzystany nie tylko do obsługi komunikacji z klasami GUI, ale również do modelowania wszystkich innych zagadnień w których konieczna jest komunikacja pomiędzy obiektami publikatorami, a słuchaczami.

Fasada: założenia (1)

- **Problem:** W jaki sposób udostępnić jednorodny interfejs do zbioru interfejsów udostępnianych przez złożony podsystem?
- **Rozwiązanie**: Wprowadzić jeden punkt dostępowy do podsystemu – fasadę, czyli obiekt który ukrywał będzie podsystem. Obiekt ten oferuje jeden spójny interfejs oraz jest odpowiedzialny za współpracę z obiektami składającymi się na podsystem.
- Fasada definiuje interfejs wyższego poziomu, który sprawia że korzystanie z podsystemu jest prostsze.
- Fasada jest jedynym punktem dostępowym do podsystemu! Komponenty podsystemu są prywatne i nie mogą być udostępnione komponentom zewnętrznym.

Fasada: założenia (2)

- Fasada redukuje złożoność komunikacyjną pomiędzy podsystemami:



Fasada: przykład (1)

- W systemie POS konieczna jest obsługa sytuacji, w których niektóre mechanizmy systemu zachowują się nieznacznie różnie od standardowego zachowania.
- Np. nietypowo może się zachowywać metoda *utwórzElementPozycjiSprzedazy* w klasie *Sprzedaż*, tak aby blokowana była możliwość wielokrotnego jej wywołania – dotyczy sytuacji realizacji płatności przez klienta przy pomocy specjalnych bonów zezwalających na zakup jedynie produktów określonego rodzaju.
- Aby problem obsłużyć poprawnie podczas tworzenia klasy *Sprzedaż* przeznaczonej do obsługi danej transakcji zostanie ona oznaczona jako wymagająca innego sposobu przetwarzania – dopuszczalne jest sprzedanie jedynie jednego produktu.
- Reguły rozstrzygające, czy produkt, który ma być w danej chwili zakupiony jest akceptowany do sprzedaży za okazany bon, czy też nie będą zapisane w dedykowanym podsystemie.

Fasada: przykład (2)

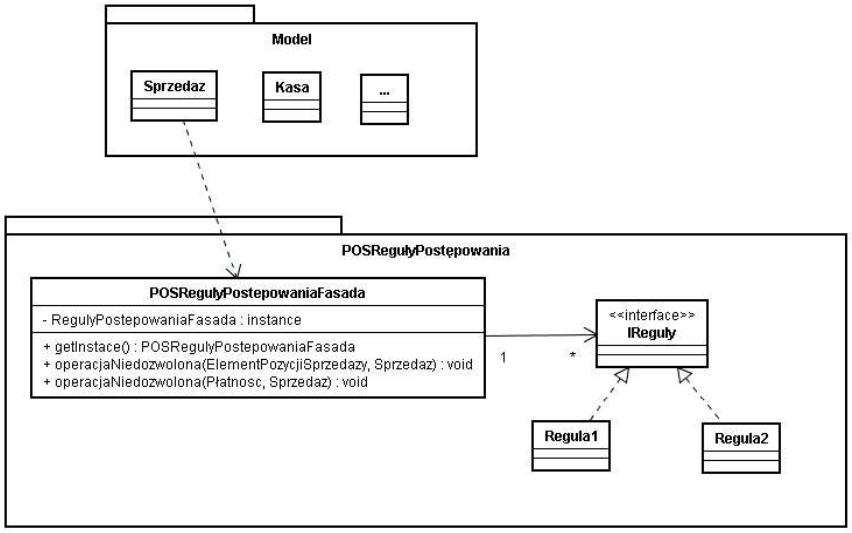
- Dostęp do podsystemu realizowany będzie poprzez fasadę *POSRegułyPostępowaniaFasada*
- Odwołanie się do fasady zapisane zostanie w metodzie, której dotyczy wariantowość:

```
public class Sprzedaż {
        ElementPozycjiSprzedaży eps = new ElementPozycjiSprzedazy(opis, ilosc);
        if (POSRegułyPostępowaniaFasada.getInstance.operacjaNiedozwolona(eps, this) )
                return;
        elementPozycjiSprzedazyKolekcja(eps);
        }
        ...
}
```

- Rozwiązanie umożliwia ukrycie złożoności reguł postępowania w podsystemie (być może bardzo złożonym), którego zasada działania nie musi być znana podczas tworzenia danej metody.
- Rozwiązanie przedstawione powyżej wykorzystuje również wzorzec Singleton.

Fasada: przykład (3)

- W UML podsystemy opisane są przy pomocy elementu ***package***.



Fasada: dyskusja

- Wzorzec fasada powinien być wykorzystany gdy:
  - Chcemy udostępnić prosty interfejs do skomplikowanego podsystemu. W takim rozwiązaniu fasada oferuje domyślny interfejs do podsystemu (w większości wypadków oznacza to również prosty interfejs). Jedynie klienci wymagający bardziej złożonych operacji będą korzystali wprost z podsystemu omijając fasadę.
  - Chcemy podzielić złożony system na logicznie różne części – fasady będą punktami dostępowymi dającymi nam takie abstrakcje.
  - Chcemy podzielić dany podsystem na warstwy. Fasady będą definiowały punkty wejścia do każdej warstwy podsystemu.
- Wzorzec Adapter jest Fasadą, która ukrywa zewnętrzne systemy różniące się interfejsami – wzorzec Fasada jest o wiele bardziej ogólny.