Inżynieria Oprogramowania

Wzorce

Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski
www.mimuw.edu.pl/~dabrowski

Motywacja

- Czy odpowiedzi na powtarzające się pytania można przedstawić w sposób ogólny
  - aby były pomocne w tworzeniu rozwiązań w różnych konkretnych kontekstach?
- Czy jakość projektu można opisać w kategoriach powtarzalnych rozwiązań
  - bez konieczności ciągłego „wynajdywania koła”?

Motywacja

- Co grozi programiście, który opanował podstawy i zaczyna pracować?
  - Będzie się zmagać z wieloma problemami, które ktoś z doświadczeniem mógłby uznać za podstawowe
- Pomysł używania nazywanych wzorców przy projektowaniu oprogramowania pochodzi od Kenta Becka (ten od Extreme Programming) i pojawił się w połowie lat 80.

Motywacja

- Wzorzec to sprawdzona koncepcja, która opisuje problem powtarzający się wielokrotnie w określonym kontekście, działające na niego siły, oraz podaje istotę jego rozwiązania w sposób abstrakcyjny.

  [Christopher Alexander, *A Pattern Language,* University of California, Berkeley]
- Niektóre analogie do budowania budynków są trafne.

W inżynierii oprogramowania

- Wzorce implementacyjne
  - Poziom języka oprogramowania
- Wzorce projektowe
  - Poziom interakcji między klasami
- Wzorce architektoniczne
  - Poziom integracji komponentów
- Wzorce analityczne
  - Poziom opisu rzeczywistości

Zasady / wzorce

- Wzorce
  - Posiadające nazwę i dobrze znane pary {problem/rozwiązania}, które można zastosować w nowym kontekście.
    - Zazwyczaj również informacje jak je dostosować oraz omówienie kompromisów, implementacji, odmian, itp.
    - Czasami mówi się również o antywzorcach
- Zbiory wzorców projektowych
  - Gang-of-Four (GoF)
    - Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides: *Wzorce projektowe*
    - coś co dla jednych jest wzorcem dla innych może być podstawowe
  - General Responsibility Assignment Patterns/Principles (GRASP)
    - Craig Larman: *Applying UML And Patterns*
    - prościej napisana; zwiera również wiele wzorców GoF
  - ...

Zasady / wzorce

- Wzorców projektowych jest bardzo dużo
  - W *Design Patterns* są opisane 23 wzorce
  - *Patterns Almanac 2000* wymienia ich 500
  - Ciągle pojawiają się nowe
    - a niektóre stare przestają być atrakcyjne
  - Większość wzorców można uznać za specjalizację pewnych zasad podstawowych

Zasady / wzorce

- Podstawowe wątpliwości -> zasady
  - Kto powinien być odpowiedzialny za tworzenie nowych instancji danej klasy?
    - Creator: zawiera (agreguje) / zapisuje / blisko używa / posiada dane inicjalizujące
  - Który obiekt (poza interfejsem użytkownika) jako pierwszy odbiera (i koordynuje) operacje systemowe?
    - Controller: reprezentuje system / scenariusz przypadku użycia
  - Według jakiej zasady poszczególnym obiektom przypisywane są odpowiedzialności (funkcje)?
    - Expert: posiada niezbędne informacje, minimalizuje zależności
  - Jak zmaksymalizować możliwość wprowadzania zmian, reużywalność, spójność?
    - High Cohesion: ograniczaj odpowiedzialności jednego bytu
    - Low Coupling: minimalizuj stopnie powiązań pomiędzy bytami
  - Czy wszystkie klasy w programie muszą być zainspirowane bytami z dziedziny?
    - Pure Fabrication: klasy nie posiadającej odpowiednika w dziedzinie problemu
  - Jak obsłużyć alternatywne rodzaje bytów? (instrukcje warunkowe)
    - Polymorphism: automatyczne rozdzielanie odpowiedzialności zgodnie z hierarchią
  - Gdzie przypisać odpowiedzialność aby uniknąć tworzenia bezpośrednich powiązań pomiędzy (dwoma lub więcej) bytami?
    - Indirection: powiązania z obiektem pośrednim a nie między sobą
  - Jak należy projektować obiekty / systemy / podsystemy aby zmiany wewnątrz tych obiektów nie miały wpływu na inne elementy?
    - Protected Variations: wokół niestabilnych miejsc (tam wariacje) stworzyć interfejsy

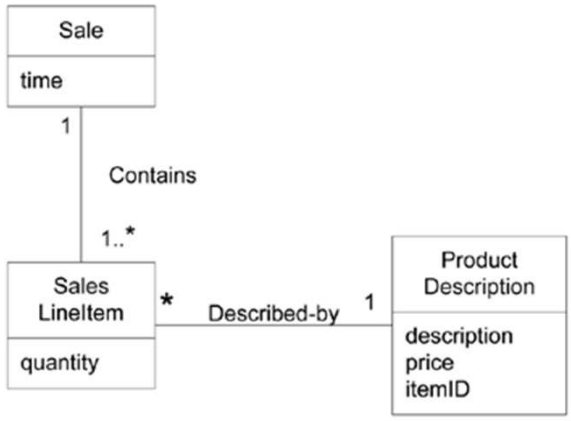
Zasady / wzorce

- Bardziej specyficzne wzorce
  - Adapter
  - Factory
  - Singleton
  - Strategy
  - Composite
  - Facade
  - Observer
  - ...

CREATOR - Problem

- Problem
  - Kto powinien być odpowiedzialny za tworzenie nowych instancji danej klasy?
    - (nie mówimy o konstruktorach, tylko o logice aplikacji)
  - Tworzenie obiektów jest jedną z częściej wykonywanych czynności w systemie obiektowym.
  - Wygodnie (dla programisty) jest mieć ogólną zasadę przypisywania odpowiedzialności za tworzenie obiektów.
  - Dobre zdefiniowanie odpowiedzialności zwiększa przejrzystość programu, zapewnia kapsułkowanie i reużywalność.

CREATOR - Przykład

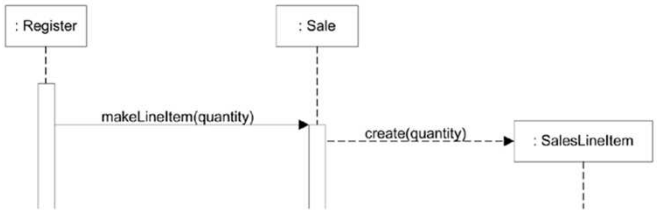


- Kto powinien być odpowiedzialny za tworzenie instancji SalesLineItem?

CREATOR - Rozwiązanie

- Rozwiązanie
  - Przypisz klasie B odpowiedzialność za tworzenie egzemplarzy klasy A jeżeli spełniony jest co najmniej jeden z warunków (im więcej, tym lepiej):
    - B zawiera (agreguje) A
    - B zapisuje A
    - B blisko używa A
    - B posiada dane inicjalizujące A

CREATOR – Przykład



- Zgodnie ze wzorcem szukamy klasy która agreguje, zawiera, itd. instancje SalesLineItem
- Ponieważ Sale zawiera (agreguje) wiele obiektów SalesLineItem, zatem jest według wzorca dobrym kandydatem na klasę odpowiedzialną za tworzenie instancji SalesLineItem

13

CREATOR - Dyskusja

- Często tworzenie nowych obiektów jest procesem złożonym
  - wykorzystanie reużywalnych obiektów ze wzlędów wydajnościowych
  - warunkowe tworzenie instancji w oparciu o klasę wybraną z rodziny klas (w oparciu o warunek zewnętrzny)
  - ...
- W takich przypadkach warto rozważyć delegowanie tworzenia obiektów do klasy pomocniczej (wzorzec Factory).

14

FACTORY - Problem

- Problem:
  - Kto odpowiada za tworzenie obiektów?
- Szczególne przypadki:
  - Logika tworzenia obiektu jest skomplikowana
  - Chcemy wydzielić logikę tworzenia obiektu (np. chcemy mieć możliwość wpływania na nią poprzez konfigurację systemu)
  - Stosujemy adaptery (zewnętrzne usługi mają zmieniające się interfejsy)
    - Kto tworzy obiekty będące adapterami?
    - W jaki sposób określana jest właściwa klasa adaptera (który powinien zostać utworzony)?
    - Jeśli adaptery są tworzone przez obiekt z dziedziny problemu, wówczas zakres odpowiedzialności takiego obiektu wykracza poza czystą logikę aplikacji.

FACTORY - Rozwiązanie

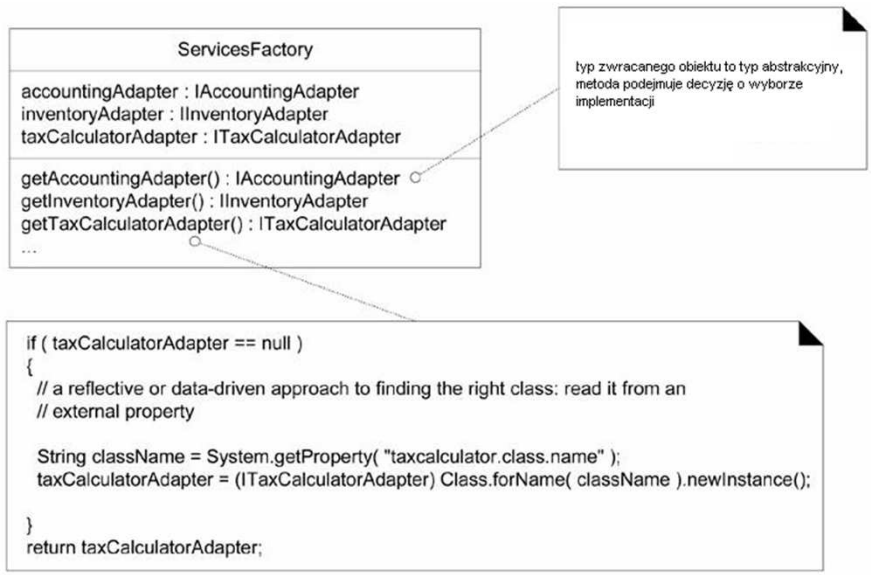
- Rozwiązanie
  - Powołujemy Fabrykę, która będzie odpowiadać za tworzenie innych obiektów
- Podobne
  - Dostęp do Fabryk jest często zapewniany z wykorzystaniem wzorca Singleton

FACTORY - Przykład

- Korzystamy z adapterów do usług oferowanych przez systemy zewnętrzne:
  - systemy autoryzacji kart płatniczych
  - systemy księgujące
  - systemy magazynowe
  - ...
- Zależy nam na utrzymaniu czystego podziału odpowiedzialności
  - która klasa za co odpowiada
  - chcemy zachować wysoką spójność projektu
- Powołujemy Fabrykę tworzącą te obiekty.
  - Fabryka korzysta z parametryzacji zewnętrznej dla systemu
    - np. zmienne środowiskowe
  - Ukrywa potencjalnie skomplikowaną logikę tworzenia obiektów
  - Pozwala na ukrycie nieeleganckich strategii
    - optymalizacja wykorzystania pamięci, reużywalne obiekty, cachowanie

17

FACTORY - Przykład



18

SINGLETON - Problem

- Problem:
  - Chcemy dopuścić istnienie dokładnie jednego (lub dokładnie kilku) egzemplarzy danej klasy
  - Egzemplarz ma być globalnie dostępny

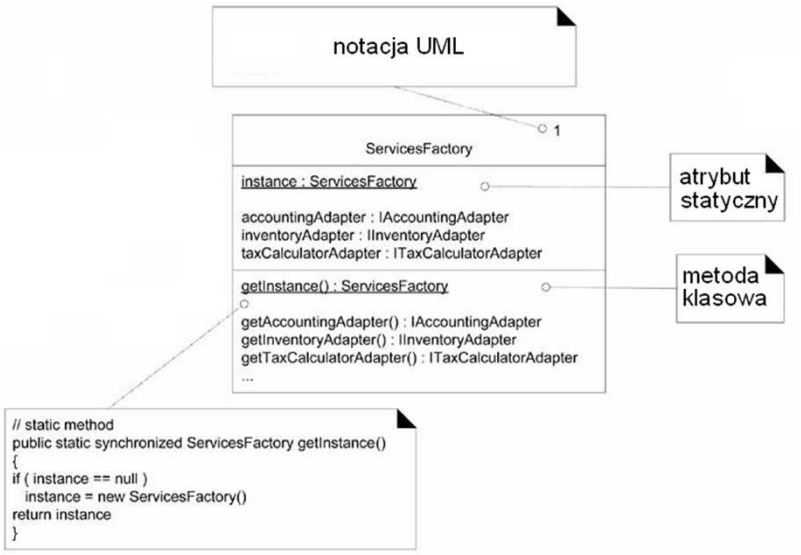
SINGLETON - Rozwiązanie

- Rozwiązanie:
  - Utwórz metodę klasową zwracającą jedyny egzemplarz
  - Zabezpiecz konstruktory przed publicznym dostępem

SINGLETON - Przykład

- W przypadku Fabryki (tworzącej obiekty) pojawia się problem:
  - Kto tworzy samą fabrykę?
  - Jak inne obiekty mają zapewniony do niej dostęp?
- Zazwyczaj wystarczy jedna instancja Fabryki obiektów
- Metody Fabryki muszą być zazwyczaj dostępne w wielu różnych miejscach programu
  - w rozważanym przypadku stosowanie adapterów do wywołania zewnętrznych usług (na zasadzie podprocedur)
- Czyli powstaje kolejny problem:
  - jak łatwo zapewnić widoczność tej jednej instancji Fabryki?
- Jedną z możliwości byłoby przekazywanie Fabryki (gdzie się tylko da) jako parametr lub inicjalizowanie Fabryką wszystkich obiektów (stała referencja)
- Lepiej w tym przypadku pozwolić na dostęp globalny („obiekt globalny")

SINGLETON - Przykład



SINGLETON - Dyskusja

- Uwagi:
  - W podobny sposób można zarządzać pulą zasobów
  - Dlaczego nie wszystkie metody statyczne?
    - w większości języków metod statycznych nie da się przesłonić w podklasie
    - większość mechanizmów zdalnego dostępu do obiektów, np. RMI, dotyczy jedynie metod egzemplarza
    - z czasem możemy zmienić zdanie i zapragnąć puli zamiast singletonu

SINGLETON - Dyskusja

- Uwagi:
  - Dlaczego nie wczesna inicjalizacja zamiast leniwej?
    - Czasami instancja nigdy nie powstaje
    - Logika tworzenia jest skomplikowana

```
public class ServicesFactory
{
// eager initialization
private static ServicesFactory instance =
   new ServicesFactory();
public static ServicesFactory getInstance()
{
   return instance;
}
// other methods...
}
```

SINGLETON - Dyskusja

- Uwagi
  - W przypadku aplikacji wielowątkowej – sekcja krytyczna

```
public static synchronized ServicesFactory getInstance()
{
  if ( instance == null )
  {
    // critical section if multi-threaded application
    instance = new ServicesFactory();
  }
  return instance;
}
```


CONTROLLER - Problem

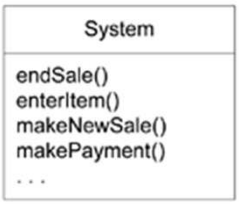
- Problem:
  - Który obiekt (poza interfejsem użytkownika) jako pierwszy odbiera (i koordynuje) operacje systemowe?
  - Operacje systemowe zostały zidentyfikowane podczas analizy. Wskazano główne zdarzenia wchodzące do systemu.
  - Kontroler jest pierwszym obiektem po warstwie UI która jest odpowiedzialna za odebranie i obsłużenie komunikatu operacyjnego.

CONTROLLER - Rozwiązanie

- Rozwiązanie:
  - Odpowiedzialność przypisz jednej z klas:
    - reprezentującej cały system, główny obiekt w systemie, urządzenie w ramach którego system działa lub główny podsystem
    - reprezentującej scenariusz przypadku użycia, w ramach którego zachodzi dana operacja systemowa
      - <UseCaseName>Handler, <UseCaseName>Coordinator, <UseCaseName>Session
- Na tej liście nie ma klas:
  - "window," "view," "document"
- Takie klasy nie powinny wypełniać zadań związanych z obsługą zdarzeń systemowych. Zazwyczaj jedynie odbierają zdarzenia i delegują je do kontrolera.

27

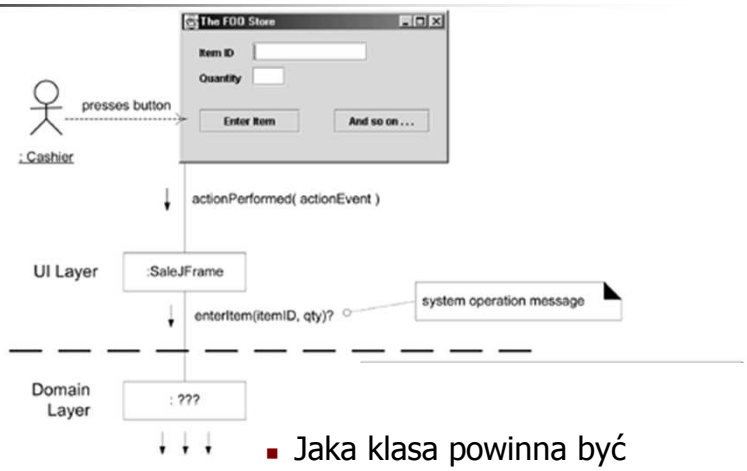
CONTROLLER - Przykład



- Podczas analizy zidentyfikowano operacje dostępne w systemie.

28

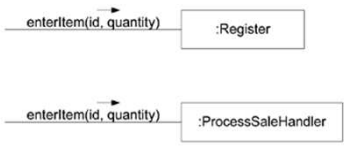
CONTROLLER - Przykład



- Jaka klasa powinna być odpowiedzialna za odebranie zdarzenia systemowego?

29

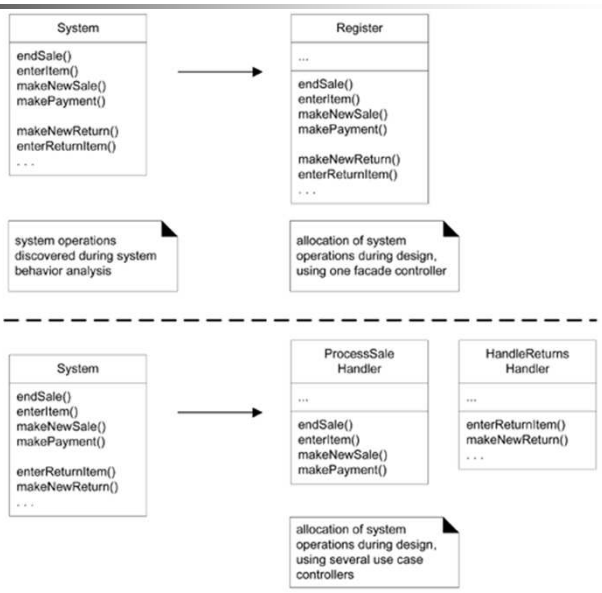
CONTROLLER - Przykład



- Kto powinien być kontrolerem dla zdarzeń systemowych takich jak enterItem lub endSale?
- Zgodnie z wzorcem CONTROLLER:
  - reprezentant całego „systemu”, „głównego obiektu”, urządzenia lub podsystemu
    - Register, POSSystem
  - reprezentant odbiorcy zdarzeń systemowych dla scenariuszy przypadku użycia
    - ProcessSaleHandler, ProcessSaleSession

30

CONTROLLER - Przykład



31

CONTROLLER - Dyskusja

- To jest wzorzec wykorzystujący delegację. Przyjmując założenie, że warstwa UI nie powinna zawierać logiki aplikacji, obiekty z warstwy UI musza delegować zadania do innej warstwy.
- Zazwyczaj system odbiera zewnętrzne zdarzenia wejściowe z wykorzystaniem GUI obsługiwanego przez osobę. Należy też uwzględnić obsługę zdarzeń z innych źródeł (wywołania procedur, sygnały, …). We wszystkich przypadkach trzeba wskazać miejsce obsługi tych zdarzeń.
- Często ta sama klasa będzie kontrolerem dla wszystkich zdarzeń z jednego przypadku użycia, żeby kontroler mógł kontrolować stan realizacji przypadku użycia. Można wtedy wychwytywać zdarzenia spoza scenariusza (sekwencji zdarzeń, np. dla rozważanego przykładu wykonanie makePayment przed wykonaniem endSale). Dla różnych przypadków użycia – różne kontrolery.
- Częstym błędem jest projektowanie kontrolerów posiadających przesadnie duży zakres odpowiedzialności.

32

CONTROLLER - Podsumowanie

- Podsumowanie
  - Pierwszą kategorią kontrolerów są fasady (FACADE) reprezentujące cały system, urządzenie, podsystem. Stanowią główne punkty obsługi zgłoszeń z warstwy UI, przekazują je do niższych warstw.
  - Fasady są czasami abstrakcjami bytów ze świata rzeczywistego (Kasa, Telefon, Robot, Centrala Telefoniczna, …) lub reprezentują cały system (PoSSystem, ChessGame, …).
  - Takie podejście sprawdza się głównie w przypadku, gdy nie ma zbyt wielu (oczywiście pojęcie „wiele” jest względne!) zdarzeń systemowych

CONTROLLER - Podsumowanie

- Podsumowanie
  - Drugą kategorią kontrolerów są klasy odpowiadające przypadkom użycia. Taki kontroler nie pochodzi z dziedziny problemu – wprowadza sztuczną warstwę / konstrukcję (PURE FABRICATION)
    - np. w rozważanym przykładzie ProcessSaleHandler do obsługi przypadków użycia Process Sale oraz Handle Returns
  - Konieczne do zastosowania jeśli
    - w systemie jest duża liczba obsługiwanych zdarzeń
    - kontroler fasadowy otrzymuje zbyt duży zakres odpowiedzialności
    - istnieje potrzeba śledzenia stanu realizacji scenariusza przypadku użycia

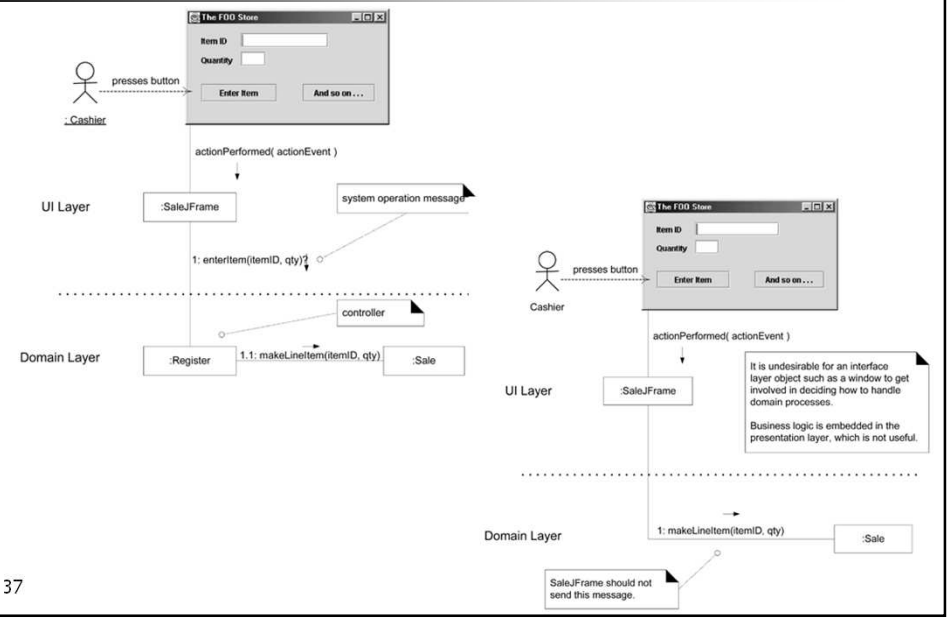
CONTROLLER – Podsumowanie

- Podsumowanie
  - W starszych metodykach występował podział na klasy:
    - boundary
    - control
    - entity
  - Ograniczenia (boundary) to abstrakcja interfejsów
  - Encje (entity) są bytami z dziedziny problemu, niezależnymi od aplikacji, zazwyczaj trwałymi (persistent)
  - Kontrolery (control) są obiektami tutaj opisywanymi

CONTROLLER – Podsumowanie

- Wytyczna:
  - Kontroler jedynie koordynuje wykonanie pracy – deleguje do innych obiektów właściwą pracę do wykonania, sam jej raczej nie wykonuje.
  - Obiekty z warstwy UI (okna, przyciski) same nie odpowiadają za obsługę zdarzeń systemowych.
  - Operacje systemowe są realizowane w warstwie logiki aplikacji / domenowej, a nie w warstwie interfejsu użytkownika

CONTROLLER – Przykład



ADAPTER - Problem

- Problem:
  - Jak ujednolicić niekompatybilne interfejsy?
  - Jak zapewnić stabilny interfejs do podobnych komponentów o różnych interfejsów?

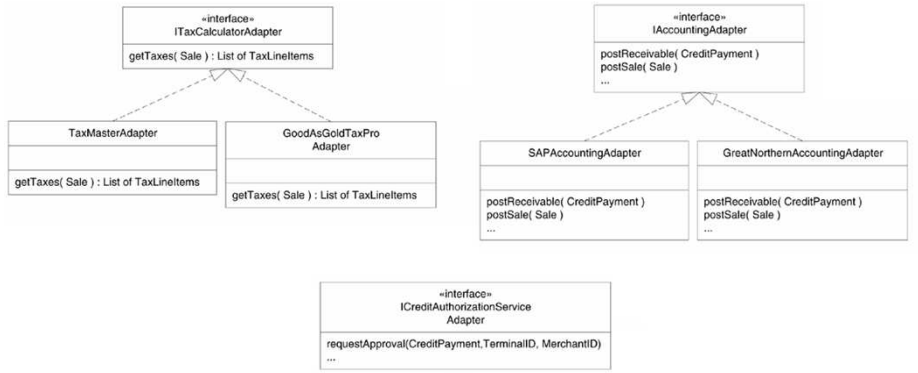
ADAPTER - Rozwiązanie

- Rozwiązanie:
  - Należy przekształcić (zasłonić) oryginalny interfejs komponentu na inny interfejs za pomocą dodatkowego obiektu – adaptera
- Podobne:
  - Adapter jest rodzajem Fasady, ale zastosowanej dla kilku systemów

ADAPTER - Przykład

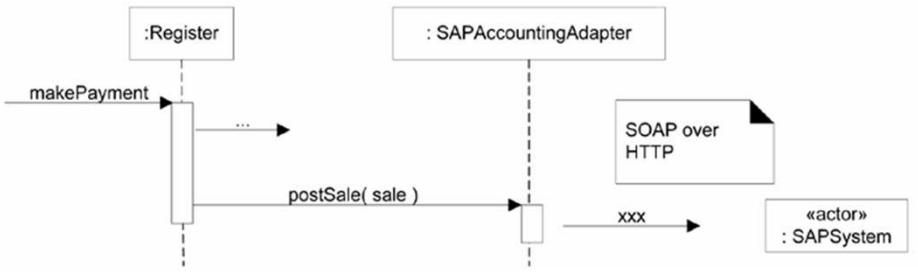
- Tworzymy system, który będzie wykorzystywać usługi oferowane przez systemy zewnętrzne:
  - systemy autoryzacji kart płatniczych
  - systemy księgujące
  - systemy magazynowe
- Systemy zewnętrzne istnieją, każdy posiada własne API (którego nie można zmienić)

ADAPTER - Przykład



itd.

ADAPTER - Przykład



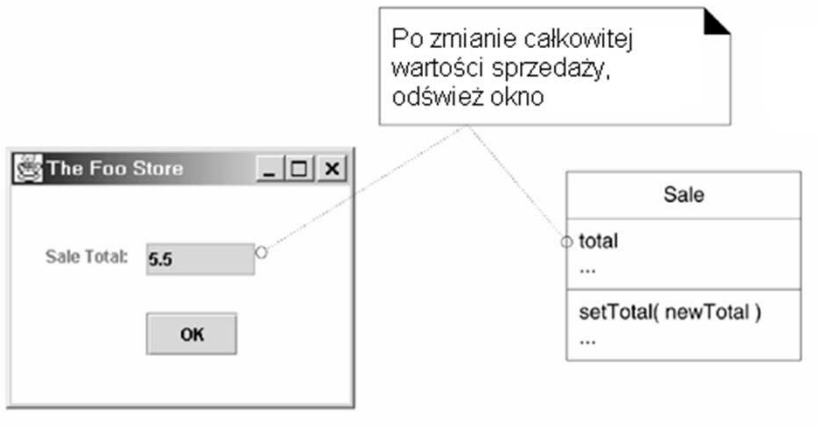
OBSERVER – Problem

- Problem:
  - Różne obiekty (obserwatorzy) są zainteresowane obserwowaniem zmian stanu obiektów (obserwowanych).
  - Chcą po swojemu reagować na te zdarzenia, jednocześnie zachowując luźne sprzężenie.

OBSERVER – Rozwiązanie

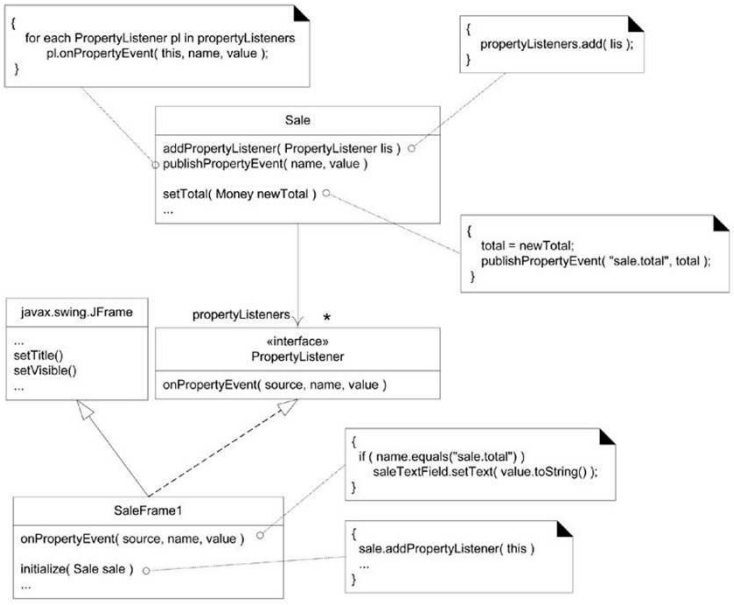
- Rozwiązanie:
  - Zdefiniuj interfejs np. *Listener* lub *Subscriber*
  - Obserwatorzy implementują ten interfejs, a obserwowani mogą dynamicznie rejestrować obserwatorów i powiadamiać ich o zdarzeniach.
  - Najczęściej stosowane dla kapsułkowania zmienności względem zmieniających się UI
    - separacja modelu i widoku
    - ale nie tylko – zdarzenia systemowe, zegary, …
  - Sprzężenie jest ograniczone do jednego interfejsu (obserwatorzy nie muszą być zawsze obecni i można ich dynamicznie dodawać/usuwać)
  - Często stosowane w bibliotekach widżetów
    - np. Swing, SWT, .Net

OBSERVER



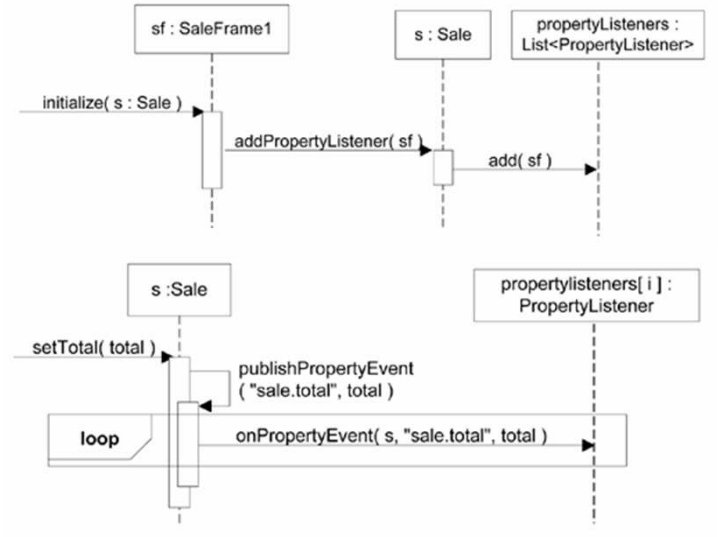
45

OBSERVER



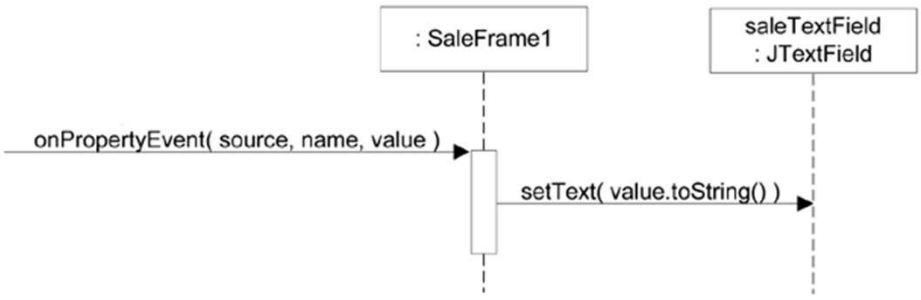
46

OBSERVER



47

OBSERVER



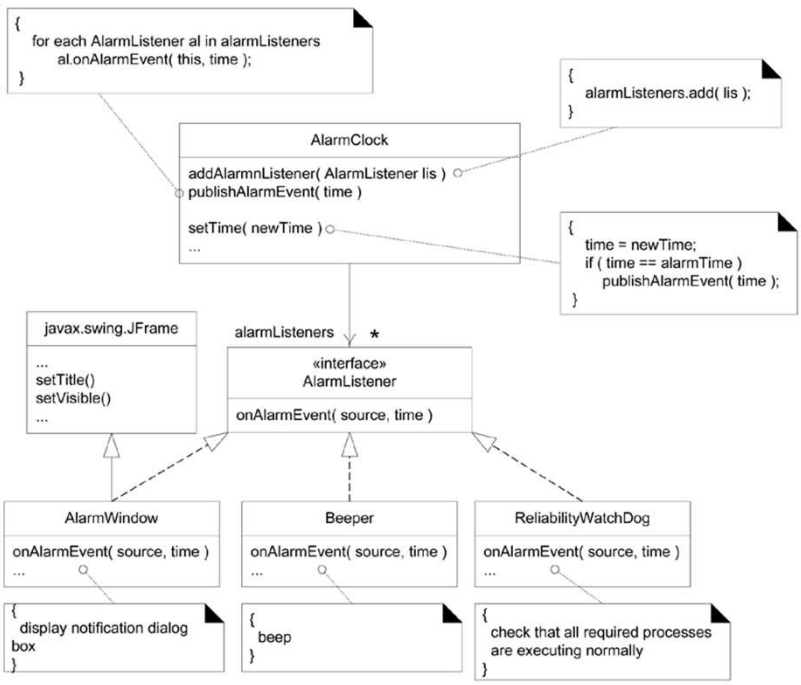
48

OBSERVER

```
class PropertyEvent extends Event
{
   private Object sourceOfEvent;
   private String propertyName;
   private Object oldValue;
   private Object newValue;
   //...
}

//...

class Sale
{
   private void publishPropertyEvent(
      String name, Object old, Object new )
   {
      PropertyEvent evt =
       new PropertyEvent( this, "sale.total", old, new);

      for each AlarmListener al in alarmListeners
       al.onPropertyEvent( evt );
   }

   //...
}
```

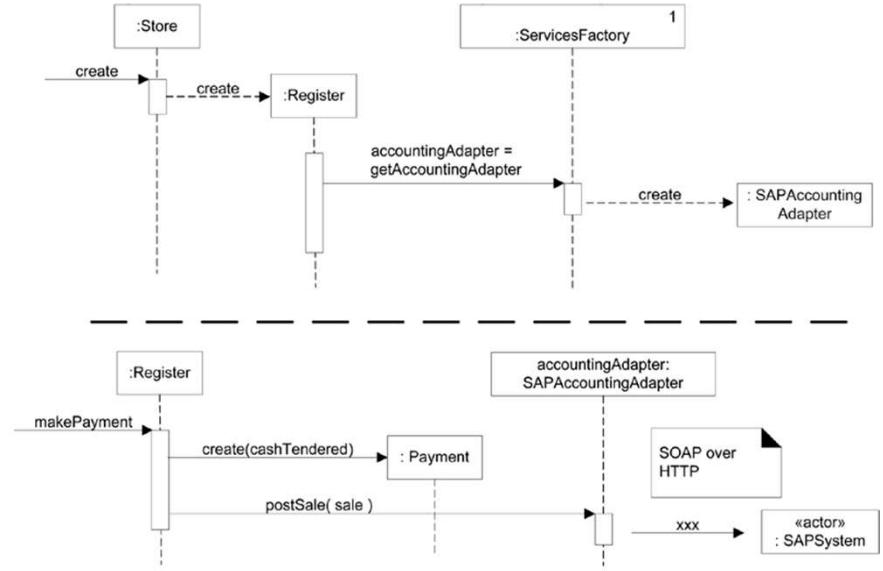
49

OBS



50

Krótkie podsumowanie dotychczasowych wzorców



STRATEGY - Problem

- Problem:
  - Jak w projekcie uwzględnić różnorodne, ale powiązane algorytmy lub strategie?
  - Jak zapewnić łatwą możliwość ich zmiany?

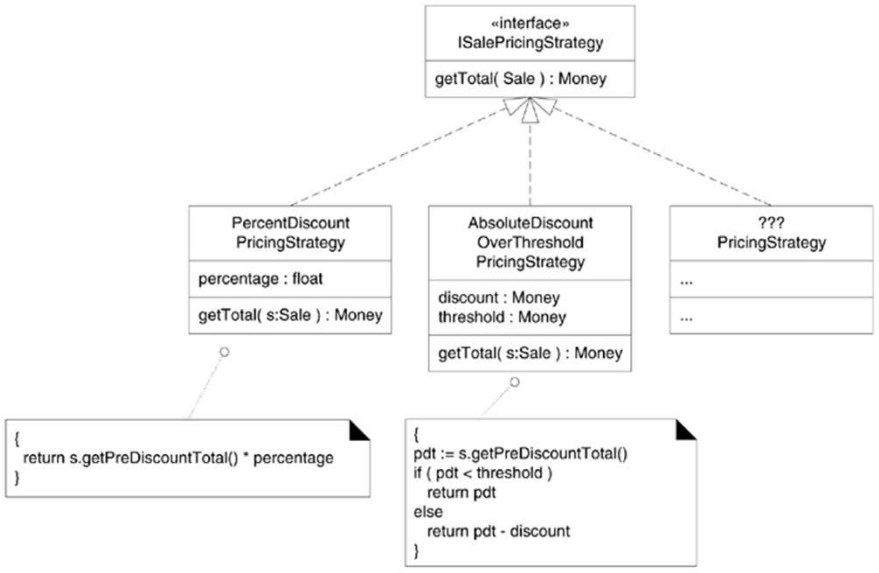
STRATEGY - Rozwiązanie

- Rozwiązanie:
  - Wszystkie algorytmy / strategie zdefiniuj w oddzielnych klasach o wspólnym interfejsie.
  - Obiekt Strategia zazwyczaj otrzymuje jako parametr jakiś obiekt wyznaczający kontekst, na którym pracuje
  - Strategie zazwyczaj tworzymy przy pomocy Singletonowej Fabryki
    - Fabryka może odczytywać wymagane dane z zewnętrznego źródła
    - użycie Fabryki pozwala łatwo zmieniać strategię w trakcie działania aplikacji

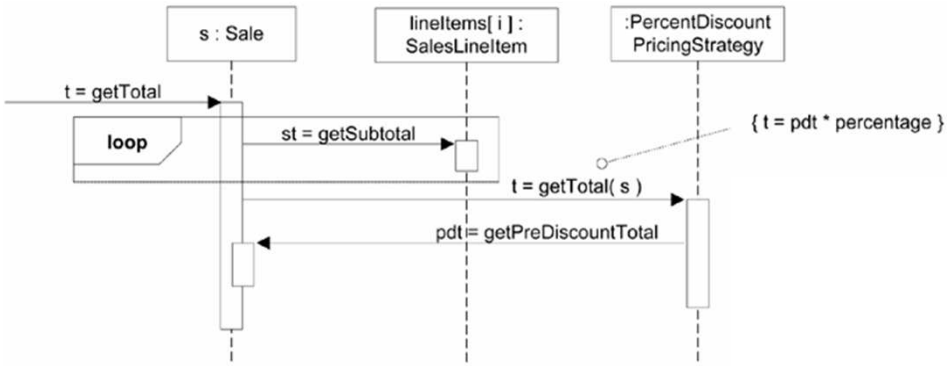
STRATEGY - Przykład

- W rozważanym systemie chcemy wykorzystywać różne strategie / algorytmy naliczania rabatów dla klienta

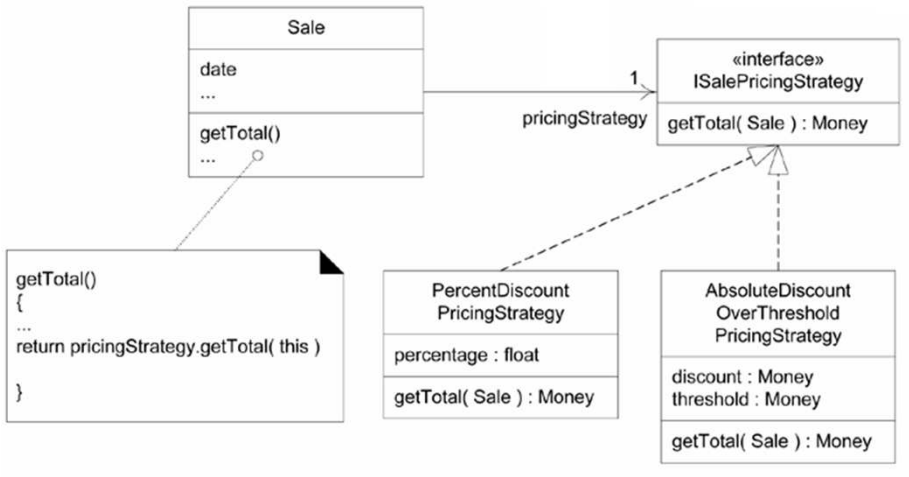
STRATEGY – Przykład



STRATEGY – Przykład

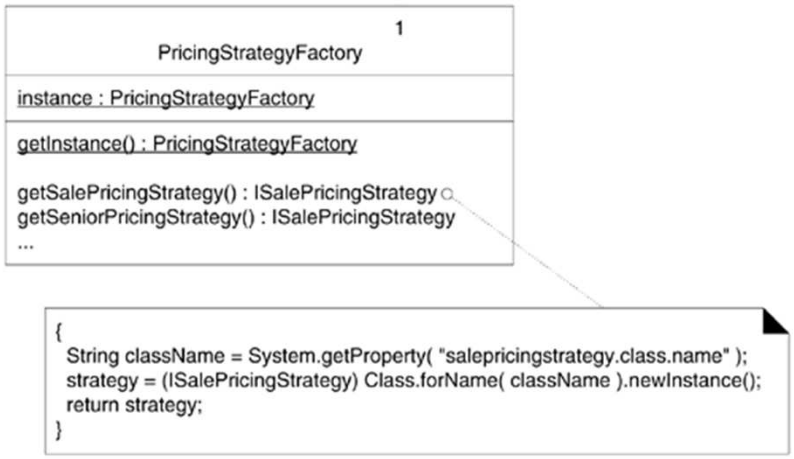


STRATEGY – Przykład



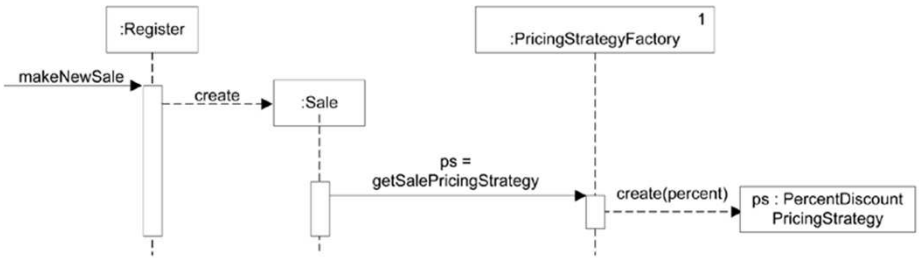
57

STRATEGY – Przykład



58

STRATEGY – Przykład



59

COMPOSITE – Problem

- Problem:
  - Jak traktować zgrupowanie lub kompozycję obiektów w taki sam sposób jak obiekt atomowy?

60

COMPOSITE – Rozwiązanie

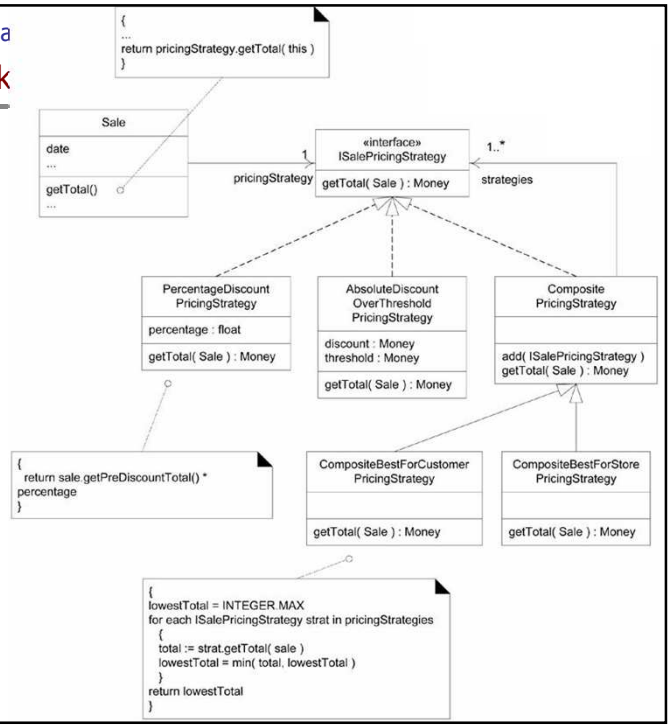
- Rozwiązanie:
  - Niech klasy reprezentujące kompozycję obiektów i obiekt atomowy posiadają ten sam interfejs

COMPOSITE – Przykład

- Przykład:
  - wiele strategii rabatu
  - polityka co zrobić jak klient pasuje do kilku strategii
    - i trzeba wybrać najlepszą dla Klienta
    - taki rabat może dotyczyć:
      - całego sklepu
        - dodajemy gdy tworzona jest sprzedaż
      - klienta
        - dodajemy gdy system zostaje poinformowany o typie klienta
      - produktu
        - dodajemy gdy produkt jest dodany do sprzedaży

Inżynieria oprogramowania

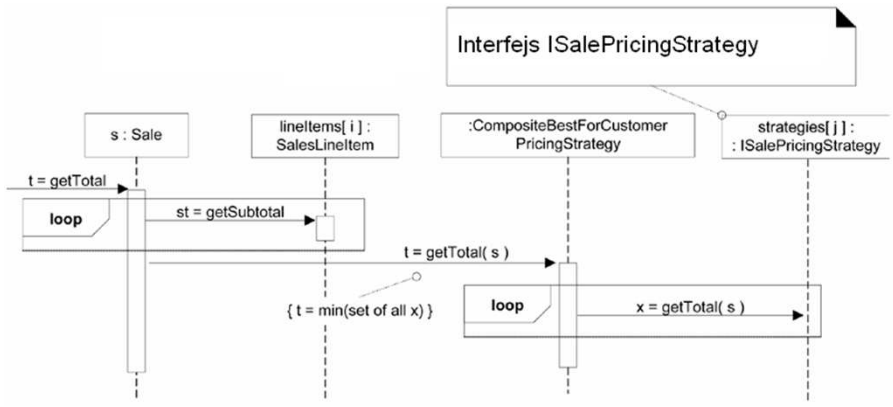
COMPOSITE – Przyk



63

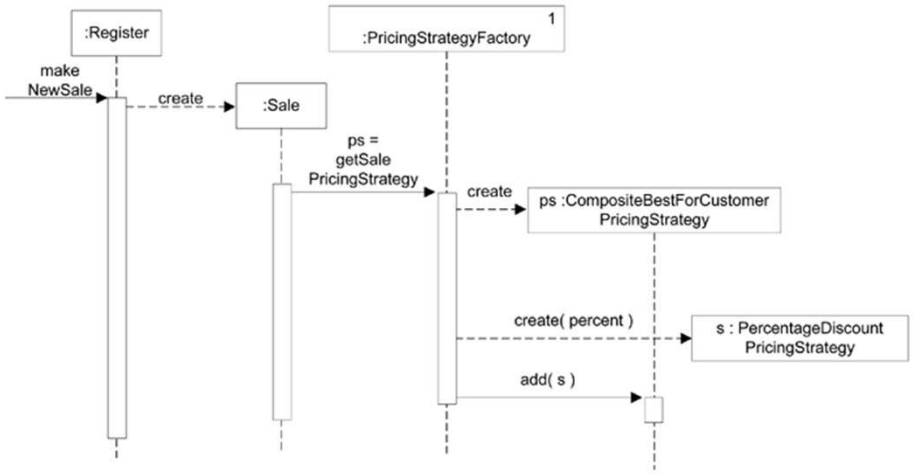
Inżynieria oprogramowania

COMPOSITE – Przykład



64

COMPOSITE – Przykład



65

COMPOSITE – Przykład

Use Case UC1: Process Sale

...

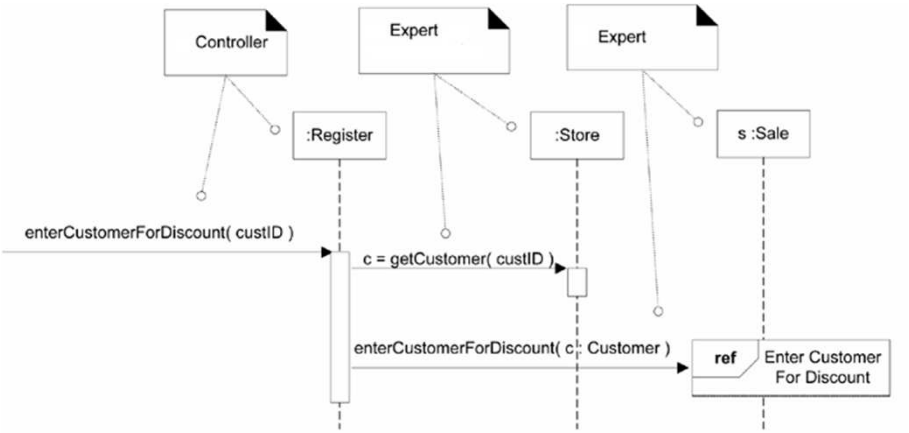
Extensions (or Alternative Flows):

5b. Customer says they are eligible for a discount (e.g., employee, preferred customer)

1. Cashier signals discount request.
2. Cashier enters Customer identification.
3. System presents discount total, based on discount rules.

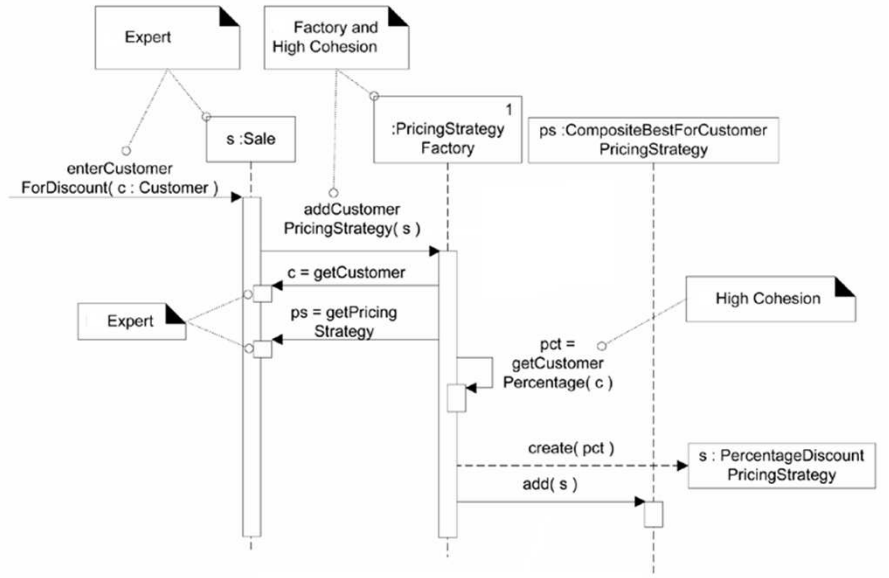
66

COMPOSITE – Przykład



67

COMPOSITE - Przykład



68

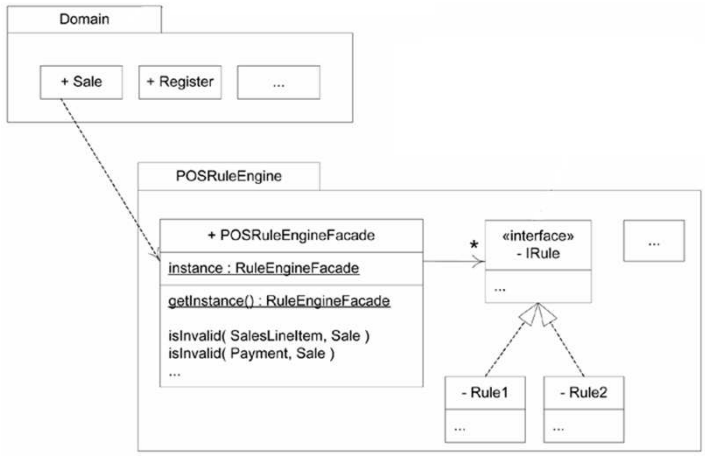
FACADE – Problem

- Problem:
  - Potrzebny jest wspólny, jednorodny interfejs do zbioru implementacji lub interfejsów
    - np. jeden interfejs do zbioru klas w ramach podsystemu
  - Chcemy ograniczyć niepożądane sprzężenie ze składnikami podsystemu oraz zabezpieczyć się przed jego zmianami.

FACADE - Rozwiązanie

- Rozwiązanie:
  - Zdefiniuj pojedynczy kontrakt dla całego podsystemu – obiekt fasadę, który opakowuje podsystem
  - Podobne do Adaptera
  - Kapsułkowanie zmienności względem zmian podsystemu.

FACADE – Przykład



Nadmiar wzorców

- Pattern Almanac 2000 wymienia ponad 500 wzorców projektowych
- Dociekliwy programista uczyłby się tylko wzorców nie mając czasu na programowanie!

Podstawowe zalecenia

- Doświadczony programista zna dobrze i stosuje 50+ najważniejszych (dla jej / jego pracy) wzorców projektowych
- Mało kto jest w stanie dobrze sklasyfikować większą ilość pojęć
- Większość wzorców jest uszczegółowieniem podstawowych zasad
  - poprzedni wykład
- Trzeba poznawać wzorce i rozumieć ich szczegóły, ale trzeba też poznać zasady ogólne
  - (i umieć tworzyć własne wzorce na nich oparte)