Studia Podyplomowe: Java EE – produkcja oprogramowania

PRODUKCJA I TESTOWANIE (PTE)
WZORCE PROJEKTOWE

(do użytku wewnętrznego)

Anna Derezińska
Instytut Informatyki
Politechnika Warszawska

Semestr zimowy 2020/21
Grupa: JA20Z (zesp. JA1-A, JA1-B) - semestr pierwszy

1

WZORCE PROJEKTOWE - LITERATURA

- E.Gamma, R.Helm, R.Johnson,J.Vlissides, Wzorce projektowe, WNT 2005, II wyd. WNT 2008, Helion 2010
- A. Shaloway, J.R. Trott: Projektowanie zorientowane obiektowo – wzorce projektowe, Helion 2001 (wyd. 1), Helion 2005 (wyd. 2)
- J.W.Cooper, Java wzorce projektowe, Helion 2001
  Kod z przykładami:
  ftp://ftp.helion.pl/przyklady/javawz.zip
- M. Yener, A. Theedom: Java EE. Zaawansowane wzorce projektowe, Helion 2015
- www.hillside.net/patterns – wzorce projektowe

2

GENEZA

„Każdy wzorzec opisuje problem, który ciągle pojawia się w naszej dziedzinie, a następnie określa zasadniczą część jego rozwiązania w taki sposób, by można było zastosować je nawet milion razy za każdym razem w nieco inny sposób"

C. Alexander, S. Ishikawa, M. Silverstein: A Pattern Language, Nowy York, Oxford University Press, 1977 (architektura)

3

BANDA CZWORGA - GANG OF FOUR

E.Gamma, R. Helm, R. Johnson, J.Vlissides: Design Patterns: Elements of Reusable Software, Reading, Mass., Addison-Wesley, 1995

- problemy, które rozwiązać można w podobny sposób przy tworzeniu programów
- wzorce w projektowaniu oprogramowania
- sposób katalogowania i opisu wzorców
- 23 wzorce
- zasady i strategie oparte na wzorcach projektowych w projektowaniu obiektowym

4

MOTYWACJA

- Możliwość wielokrotnego wykorzystania
- Przyspieszenie pracy
- Unikanie błędów
- Wspólny punkt odniesienia, terminologia, ułatwienie pracy i komunikacji w zespole
- Ogólna perspektywa bez wczesnego zgłębiania szczegółów
- Poprawa pielęgnowalności (zrozumienie kodu, możliwość modyfikacji, skalowalność)

Ale – możliwe wydłużenie kodu

5

KATEGORIE WZORCÓW PROJEKTOWYCH

Konstrukcyjne:
Do utworzenia obiektów:
fabryka, fabryka abstrakcji, singleton, budowniczy, prototyp, metoda produkcyjna/fabrykująca

Strukturalne :
Do powiązania istniejących obiektów: powiązania implementacji i abstrakcji, obsługi interfejsów.
fasada, adapter, most, dekorator, composite, pośrednik (proxy), waga piórkowa (flyweight).

Czynnościowe (zachowania, behawioralne):
Do manifestacji zmiennego zachowania: *strategia.*
stan, obserwator, iterator, metoda szablonu, mediator, wizytator, metoda szablonowa, łańcuch odpowiedzialności, polecenie, interpreter.

6

SINGLETON

Zapewnia, że istnieje tylko jedna instancja danej klasy oraz zapewnia globalny punkt dostępu do tej instancji.

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

Statyczna zmienna, prywatny konstruktor

Nie wołamy konstruktora ~~Singleton s = new Singleton();~~

Tworzymy instancję statyczną metodą, zawsze ma być ta sama instancja `Singleton s = Singleton.getInstance();`

7

FABRYKA ABSTRAKCJI

Intencja: Uzyskanie rodzin obiektów właściwych w określonym przypadku

Problem: Utworzenie odpowiednich rodzin obiektów

Rozwiązanie: Koordynacja tworzenia rodzin obiektów. Wydzielenie z obiektów użytkownika reguły tworzenia obiektów, które są przez nie używane.

Uczestnicy i współpracownicy: *Abs_ Fabryka* definiuje interfejs do tworzenia każdego z obiektów danej rodziny. Każda z rodzin obiektów posiada własną klasę *kon_fabryka*.

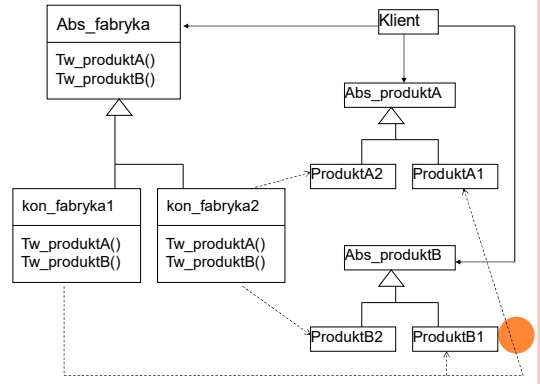
Konsekwencje: Izolacja reguł opisujących sposób wykorzystania obiektów od reguł decydujących o utworzeniu tych obiektów.

Implementacja: Klasa abstrakcyjna specyfikuje tworzone obiekty. Dla każdej z rodzin obiektów implementuje się klasę konkretną. Pliki konfiguracyjne lub tabela bazy danych do wyboru tworzonych obiektów

*Wzorzec ten jest stosowany m.in. w bibliotece **Java Swing** do reprezentacji tzw. skórek (mechanizmu umożliwiającego szybką zmianę wyglądu interfejsu użytkownika). Wszystkie implementacje okienek, przycisków, list i innych elementów GUI są produkowane przez wybraną fabrykę. Zmiana implementacji tej fabryki oznacza jednoczesną modyfikację zawartości ekranu.*

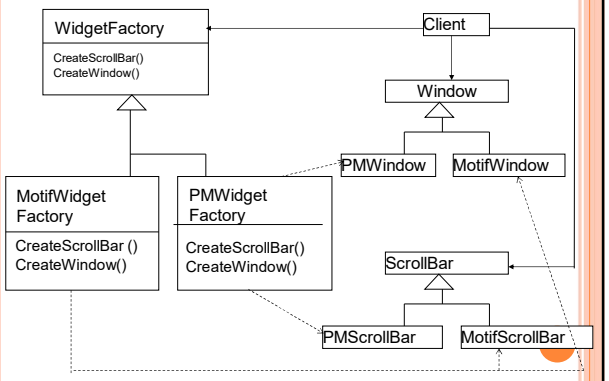
11

FABRYKA ABSTRAKCJI



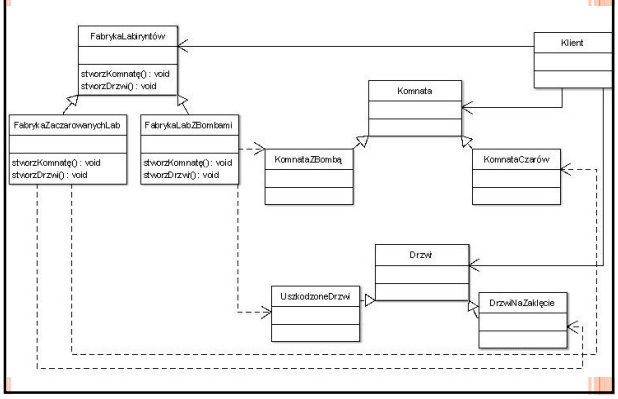
12

FABRYKA ABSTRAKCJI - PRZYKŁAD



13

FA – TWORZENIE LABIRYNTU DO GRY



14

FABRYKA ABSTRAKCJI - PRZYKŁAD

- Opis przykładu w pliku java-pte2-v3.pdf
- Kod przykładu w projekcie abstractFactory

- Przeanalizować kod, powiązania klas
- Uruchomić przykład
- Ewentualnie uruchomić przykład dla drugiej fabryki (drugiego systemu operacyjnego)

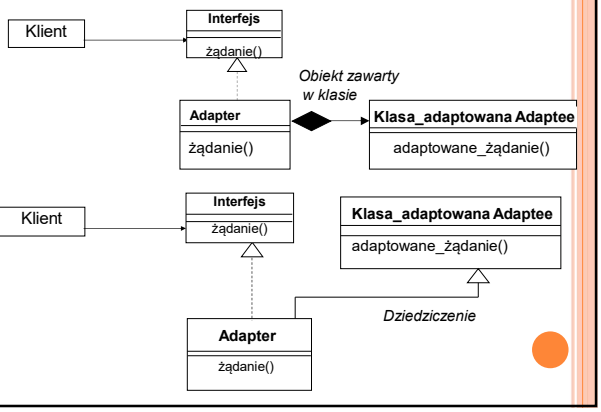
15

ADAPTER

Intencja: Dopasowanie istniejącego obiektu do sposobu wywołania.
Problem: Obiekt posiada nieodpowiedni interfejs.
Rozwiązanie: Adapter obudowuje obiekt pożądanym interfejsem
Uczestnicy i współpracownicy: Adapter dostosowuje interfejs klasy *Klasa-adaptowana* tak, by był zgodny z interfejsem klasy *Adapter*
Konsekwencje: Dopasowanie istniejących obiektów do tworzonych struktur klas
Uniknięcie ograniczeń związanych z ich interfejsem.
Implementacja: 1) Zawiera istniejąca klasę w nowej klasie, która posiada wymagany interfejs
2) Dziedziczy z istniejącej klasy

16

ADAPTER



17

ADAPTER

```
public class Adaptee {
        public void specialCall() {       }
}
public interface TargetInterface {
        void clientRequest();
}
public class Adapter implements TargetInterface {
        private Adaptee adaptee;
        public Adapter(Adaptee adaptee) {
                this.adaptee = adaptee; }
        @Override
        public void clientRequest() {
        adaptee.specialCall();     }
}
```

package pl.edu.pw.ii.pte.patterns.adapter

18

ADAPTER - ĆWICZENIE

- Uzupełnić metodę w klasie adptowanej *Adaptee* – żeby coś robiła (np. wypisała na ekran).
- Napisać klasę *AdpterDemo*, z metodą demonstrująca użycie adaptera

package pl.edu.pw.ii.pte.patterns.adapter

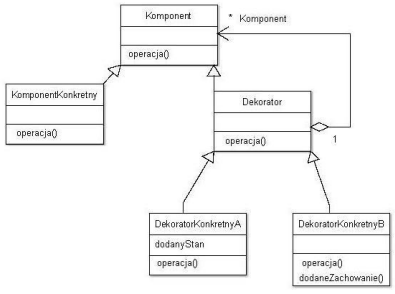
19

DEKORATOR

- Dodanie funkcjonalności istniejącemu obiektowi przez obudowanie go w dodatkowe możliwości
- Bez modyfikacji istniejącego kodu
- Obiekty dekorowany i dekorujące są tego samego typu
- Jeden obiekt dekorowany może być otoczony przez wiele dekorujących
- Przykład – obiekt posiada różne kombinacje rozszerzeń w zależności od potrzeb, można decydować o nich w trakcie działania programu

20

DEKORATOR



21

DECORATOR – PRZYKŁAD

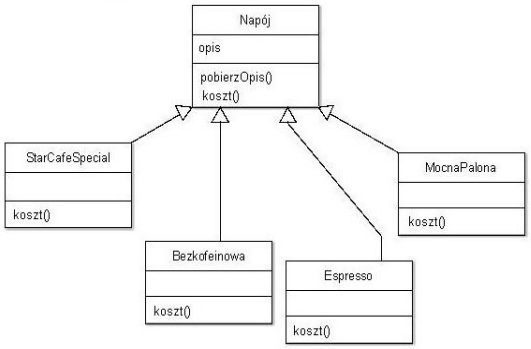
```
public abstract class AuthDecorator implements AuthenticationService {
        protected AuthDecorator(AuthenticationService s) {
                this.decoratedService = s;
        }
        @Override
        public boolean loginUser(String name, String authString) {
                return decoratedService.loginUser(name, authString);
        }
        private AuthenticationService decoratedService;
}
```

AuthDecorator ma dwie relacje do interfejsu *AuthenticationService* :
- implementacja interfejsu
- zawieranie

package pl.edu.pw.ii.pte.patterns.decorator.example

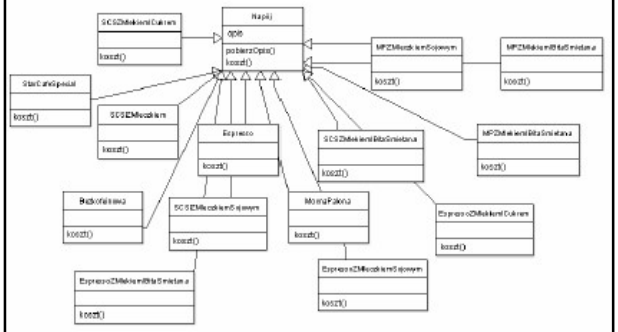
22

PRZYKŁAD – NAPOJE W KAWIARNI



23

ROZSZERZENIE OFERTY NAPOJÓW



24

AMBASADOR (PROXY), POŚREDNIK

Zastosowanie:
- nie jest konieczne stałe utrzymywanie zainicjowanego obiektu w systemie,
- utrzymywanie zainicjowanego obiektu powoduje duże zużycie zasobów

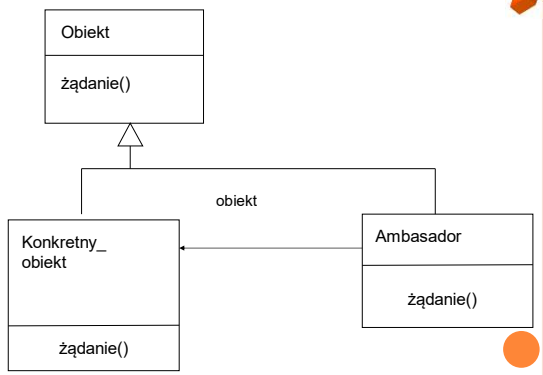
Identyczny interfejs jak reprezentowanego obiektu

Powołanie do życia konkretnego obiektu.

Rzeczywiste tworzenie obiektu opóźnione.

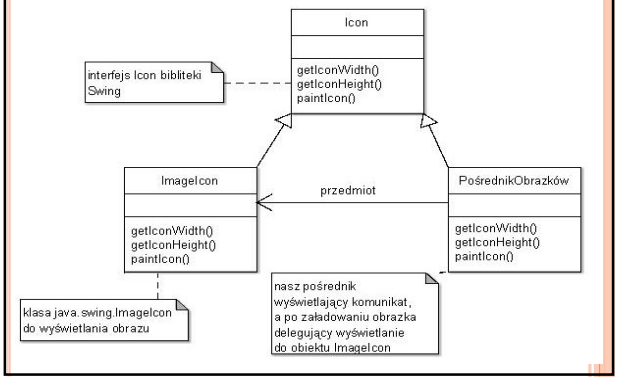
26

AMBASADOR



27

AMBASADOR – WYŚWIETLANIE OBRAZU



28

TYPOWE ZASTOSOWANIA AMBASADORA

- Zdalne – remote proxy

lokalny reprezentant w systemie rozproszonym, odwołania sieciowe przezroczyste dla klienta

- Wirtualne

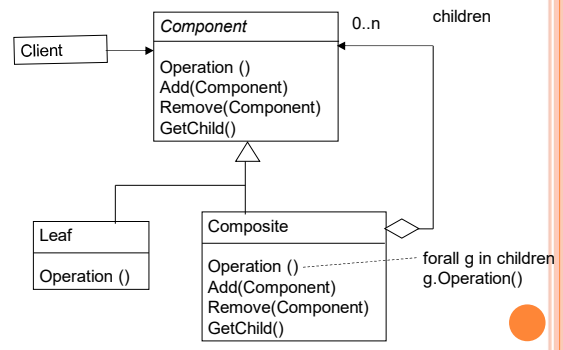
zamiennik dla obiektu wymagającego dużych zasobów (np. pamięci)

- Ochronne

udostępnienie obiektu tylko uprawnionym,
Klient nie ma bezpośredniego dostępu do rzeczywistego obiektu, a tylko do usług którymi zarządza proxy

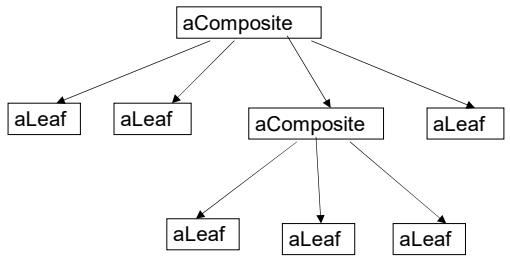
29

COMPOSITE



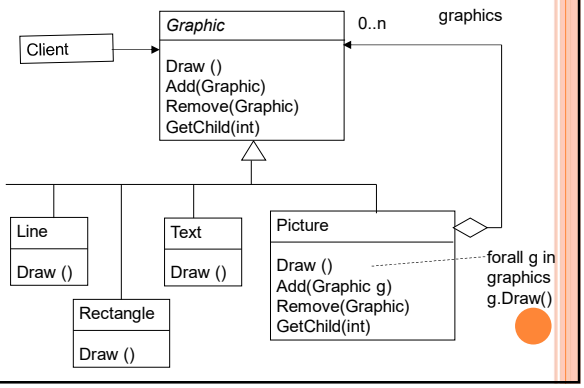
30

REKURENCYJNA STRUKTURA OBIEKTÓW



31

COMPOSITE - PRZYKŁAD



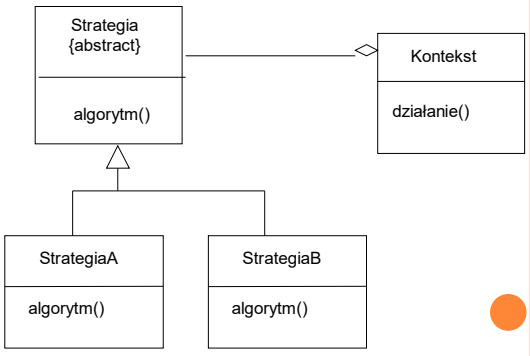
32

STRATEGIA

Intencja: Użycie różnych wersji algorytmu w zależności od kontekstu, Ukrycie szczegółów algorytmu przed klientem
Problem: Wybór algorytmu zależy od używającego go obiektu lub danych na których operuje.
Rozwiązanie: Separuje wybór algorytmu od jego implementacji.
Umożliwia wybór algorytmu na podstawie kontekstu.
Uczestnicy i współpracownicy:
strategia specyfikuje sposób użycia różnych algorytmów.
strategiaA, *strategia*B implementują konkretny algorytm.
Konsekwencje: Eliminacja instrukcji wyboru.
Algorytmy muszą być wywoływane w ten sam sposób.
Implementacja: Klasa *kontekst* zawiera klasę abstrakcyjną *strategia*.z metodą abstrakcyjną określającą sposób wywołania algorytmu. Klasy pochodne implementują algorytm. Gdy metody mają wspólną część, metoda ta nie musi być abstrakcyjna.

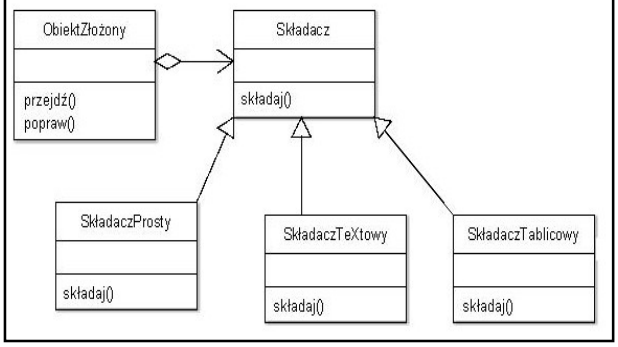
36

STRATEGIA



37

STRATEGIA – ALGORYTMY PODZIAŁU STRUMIENIA ZNAKÓW NA WIERSZE



38

STAN

Cel: Implementacja informacji z diagramu stanów klasy
Zmiana zachowania obiektu w zależności od stanu, w jakim obiekt się znajduje.

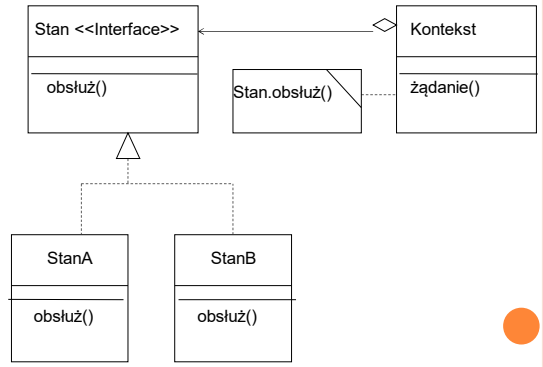
- Kontekst - posiada referencję do obiektu typu *Stan* reprezentującego bieżący stan
- Klasa abstrakcyjna *Stan* - definiuje interfejs pozwalający hermetyzować zachowanie związane z każdym stanem
- Klasy pochodne dla każdego stanu zawierają metody implementujące zachowanie specyficzne dla tego stanu

Metody obiektu *Stan* są polimorficzne, czyli wraz ze zmianą tego obiektu zmienia się też ich funkcjonalność.

Łatwe wprowadzanie zmian zachowania - zmiany prostych metod klas implementujących stany.
Wada - tworzenie rozbudowanej hierarchii klas.

39

STAN



40

PRZYKŁAD KONTA

```
public class Account {
   private int balance = 0;    //suma kredytów
   private String owner = null;          //właściciel
   private boolean isOpen = false;       //rachunek nieaktywny

public Account (String owner, int balance) {
   this.owner = owner;
   this.balance = balance;
   this.isOpen = true;                   //rachunek aktywny
   }
public void credit(int amount) {         //udzielenie kredytu zależy od stanu
   if (isOpen)
         {  balance += amount;   }
   else
         {  throw new IllegalStateException("Konto nieaktywne!");  }
   }
public void close() {          // zablokowanie konta
   this.isOpen = false;   }
.....
}
```

package pl.edu.pw.ii.pte.patterns.account_without_state

41

KONTO/STAN – ĆWICZENIE (WSTĘP)

- Dla konta (Account_without_state) utworzyć metodę dostępową getBalance()
- Dla konta (Account_without_state) utworzyć klasę z testami jednostkowymi testującymi przydział kredytu w obu stanach
- Przekopiwać getBalance() oraz klasę z testami do drugiej wersji konta (Account_with_state)

UWAGA Funkcjonalność i interfejsy metod dla zrefaktoryzowanego konta są identyczne – czyli powinny działać takie same testy.

42

STANY - KONTO AKTYWNE I NIEAKTYWNE

```
public interface AccountState {
   public void credit(Account acc, int amount);
}

public class AccountOpen implements AccountState {
   public void credit(Account acc, int amount) {
         acc.incBalance(amount);
   }
}

public class AccountClosed implements AccountState {
   public void credit(Account acc, int amount) {
         throw new IllegalStateException("Konto nieaktywne!");     }
}
```

43

KONTO KORZYSTAJĄCE Z WZORCA STAN

```
public class Account {
   private int balance = 0;    //saldo
   private String owner = null;          //właściciel
   private AccountState state = null;  //rachunek nieaktywny

public Account (String owner, int balance) {
   this.owner = owner;
   this.balance = balance;
   this.state = new AccountOpen();              //rachunek aktywny
   }
public void credit(int amount) {        //udzielenie kredytu zależy od stanu
   this.state.credit(this, amount);      //brak if -  else
   }
public void close() {          // zablokowanie konta
   this.state = new AccountClosed();             }
....
}                    package pl.edu.pw.ii.pte.patterns.account_with_state
```

44

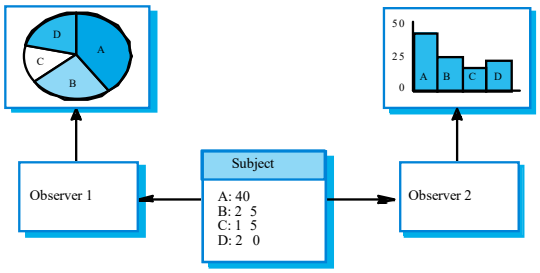
KONTO/STAN – ĆWICZENIE C.D.

- Dla konta korzystającego z wzorca projektowego Stan (*Account_with_state*) dodać:
  - nowy stan *Suspended*, w którym przydzielone jest tylko 10% kredytu,
  - konieczne operacje dla obsługi tego stanu w klasie *Account_with_state*,
  - testy dla nowego stanu.

package pl.edu.pw.ii.pte.patterns.account_with_state

45

WIELOKROTNE PREZENTACJE



Obserwator - projektowanie środowisk prezentacji danych.

46

OBSERWATOR (PUBLISH-SUBSCRIBE)

Intencja: Definicja „jeden-do-wielu" zależności między obiektami.
Zmiana stanu obiektu propaguje na związane z nim obiekty.

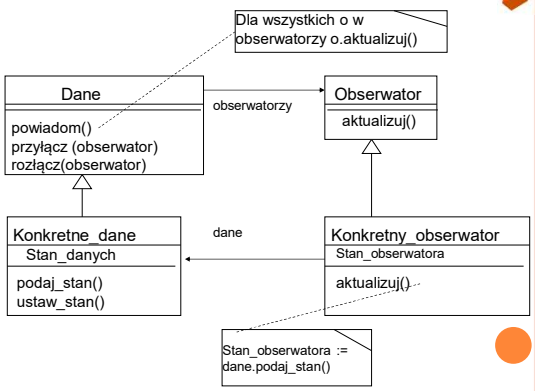
Problem: Powiadamianie zmieniającej się listy obiektów o zdarzeniu.

Rozwiązanie: Obiekty klasy *Obserwator* przekazują odpowiedzialność za monitorowane zdarzenia centralnemu obiektowi *dana*.

Konsekwencje: Klasa *Dane* może niepotrzebnie powiadamiać o zdarzeniu różne rodzaje obserwatorów

47

OBSERWATOR (PUBLISH-SUBSCRIBE)



48

OBSERWATOR

Uczestnicy i współpracownicy:

- Klasa ***Dane*** – zrejestruje obserwatorów – *przyłącz, rozłącz*. Metoda *powiadom* informuje obiekty o zmianach wewnątrz niej.
- Klasa ***Obserwator*** – definiuje interfejs dla obiektów powiadamianych o zmianach w danych.
  Po otrzymaniu informacji o konieczności aktualizacji wysyła żądanie do klasy *dane* z prośbą o podanie aktualnego stanu.

Powiązanie na poziomie klas konkretnych – komunikacja od klas obserwujących do danych.

- Klasa ***Konkretny_obserwator*** :
  - wskazanie na konkretne dane,
  - stan spójny z danymi.
- Klasa ***Konkretne_dane*** – powiadamia obserwatorów o zmianie stanu, przechowuje stan

49