Wzorce projektowe

Hello!

Tomasz Lisowski

Software developer, JIT Solutions

IT trainer

tomasz.lisowski@protonmail.ch

Agenda

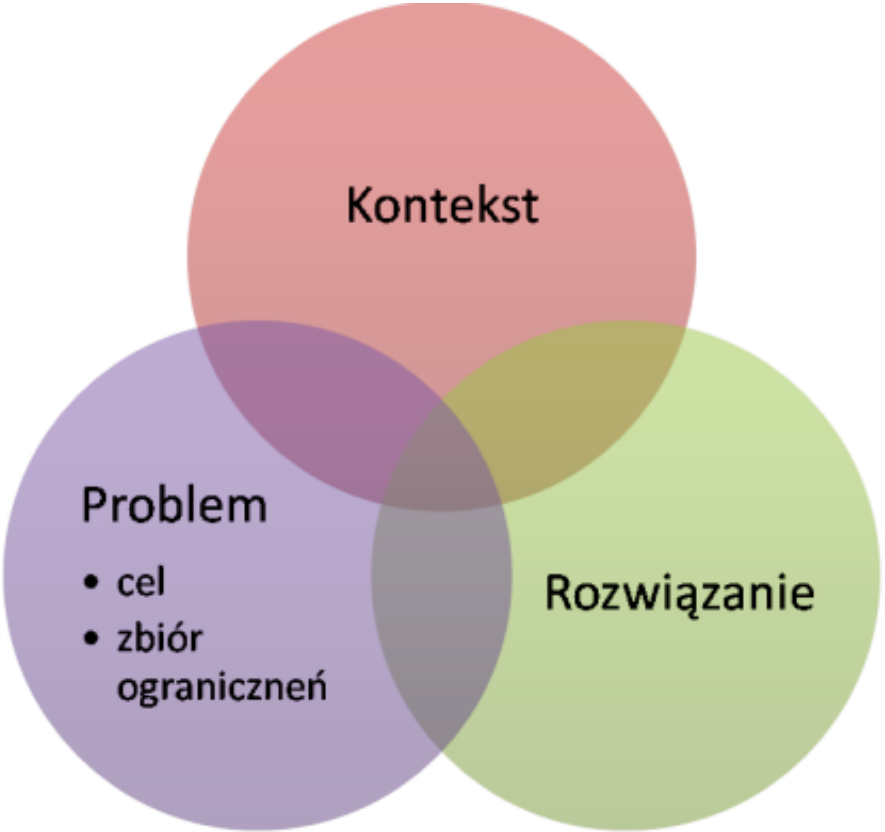
- powtórka
- omówienie
- anty-wzorce
- ćwiczenia

Wzorce
wprowadzenie

- uniwersalne, sprawdzone w praktyce rozwiązanie często pojawiających się powtarzalnych problemów
- opis lub szablon stworzony do rozwiązania problemu
- abstrakcyjny zbiór rozwiązań abstrakcyjnych problemów
- **nie są to gotowe rozwiązania (implementacje)**

Wzorce
wprowadzenie

- rozwiązanie problemu w danym kontekście



Podział wzorców

Wzorce projektowe

- **kreacyjne** - używane do tworzenia obiektów w taki sposób, aby były oddzielone od systemu, który je implementuje
- **strukturalne** – używane do tworzenia struktur między różnymi obiektami
- **behawioralne** – używane do zarządzania algorytmami, powiązaniami i odpowiedzialnościami

Wzorce projektowe

kreacyjne	strukturalne	behawioralne
singleton	*adapter*	*template method*
builder	*decorator*	*command*
factory method	*proxy*	*observer*
abstract factory	*facade*	*strategy*
prototype		*chain of responsibility*

Katalog wzorców - wzorce kreacyjne

Wzorce
kreacyjne

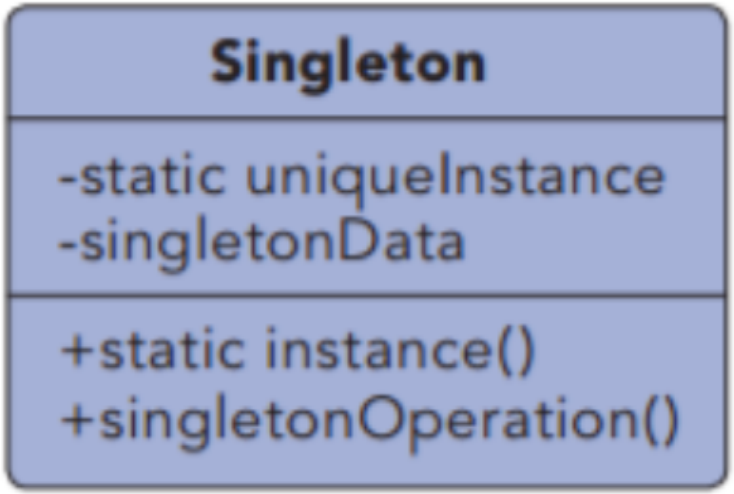
- prowadzą do utworzenia obiektu
- separują tworzenie obiektów od klienta, który je tworzy
- ułatwiają budowę systemu, który jest niezależny od sposobu tworzenia i składania obiektów

Wzorce
kreacyjne

- **Singleton** – zapewnia, że klasa ma tylko jeden egzemplarz i zapewnia globalny dostęp do niego
- **Builder** – rozdzielenie sposobu tworzenia obiektów od ich reprezentacji w taki sposób, że ten sam proces konstrukcji może tworzyć inne reprezentacje
- **Factory Method** – dostarcza interfejs do tworzenia obiektów, pozwalając podklasom tworzyć konkretne obiekty
- **Abstract Method** – dostarcza interfejs do tworzenia rodzin powiązanych lub zależnych obiektów bez podawania konkretnych klas
- **Prototype** – pozwala na tworzenie obiektów na podstawie szablonów istniejących obiektów poprzez klonowanie

Wzorce
singleton

- zapewnia, że klasa ma tylko jeden egzemplarz i zapewnia globalny dostęp do niego
- https://github.com/iluwatar/java-design-patterns/tree/master/singleton



Wzorce
singleton

Cel

- zapewnia, że klasa ma tylko jeden egzemplarz i zapewnia globalny dostęp do niego

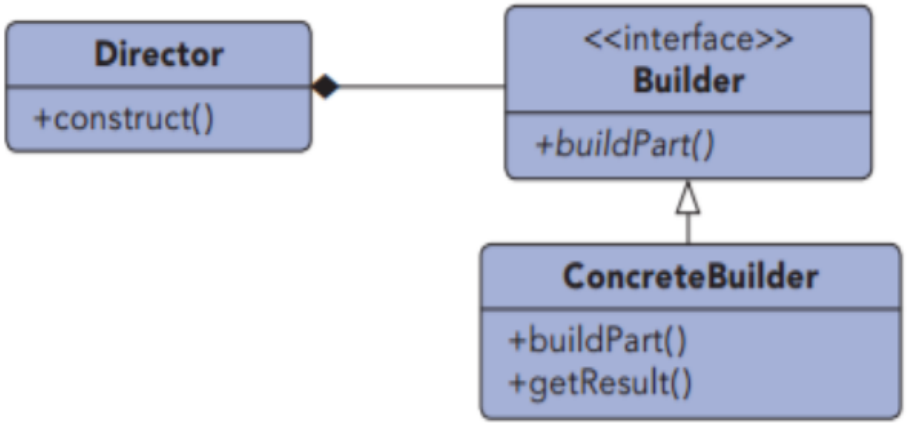
Konsekwencje

- klasa może kontrolować dostęp do swojej jedynej instancji
- wzorzec wprowadza zmienną globalną i usztywnia projekt

```
public enum EnumIvoryTower {
  INSTANCE;
}
```

Wzorce
builder

- zapewnia oddzielenie konstrukcji skomplikowanego obiektu od jego reprezentacji – ten sam proces może tworzyć różne instancje
- https://github.com/iluwatar/java-design-patterns/tree/master/builder



Wzorce
builder

Cel

- pozwala konstruować obiekty z komponentów

Konsekwencje

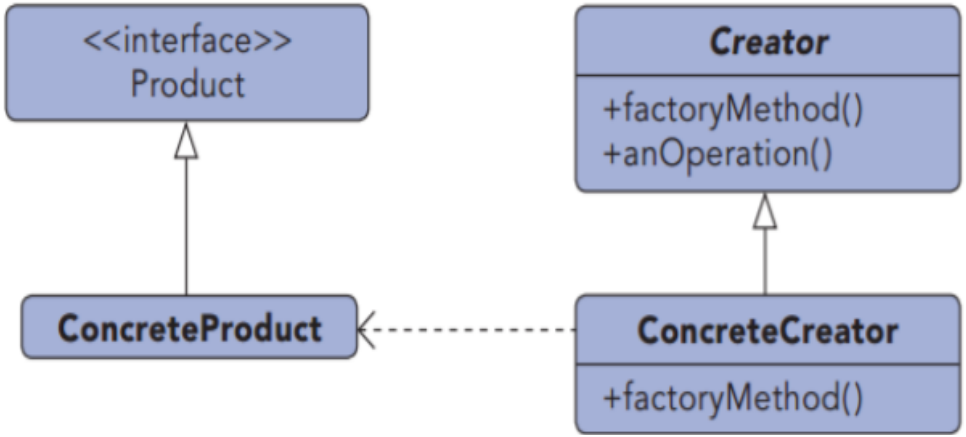
- oddziela kod służący konstruowaniu obiektu od jego wewnętrznej reprezentacji
- klient nie musi nic wiedzieć o wewnętrznej strukturze obiektu stworzonego przez builder

Zastosowanie

- skomplikowana struktura obiektowa tworzona w prosty sposób
- kreator tworzący obiekty w kilku etapach

Wzorce
factory method

- dostarcza interfejs do tworzenia obiektów pozwalając podklasom tworzyć konkretne obiekty
- https://github.com/iluwatar/java-design-patterns/tree/master/factory-method



Wzorce
factory method

Cel

- podejmuje decyzję o tym jaki obiekt należy utworzyć i tworzy go

Konsekwencje

- podklasy decydują jakiego typu obiekt zostanie utworzony

Zastosowanie

- wyodrębnienie tworzenia obiektów do osobnej dedykowanej klasy
- fabryka tworząca obiekt połączenia do bazy danych określonego typu

Katalog wzorców
- wzorce strukturalne

Wzorce strukturalne

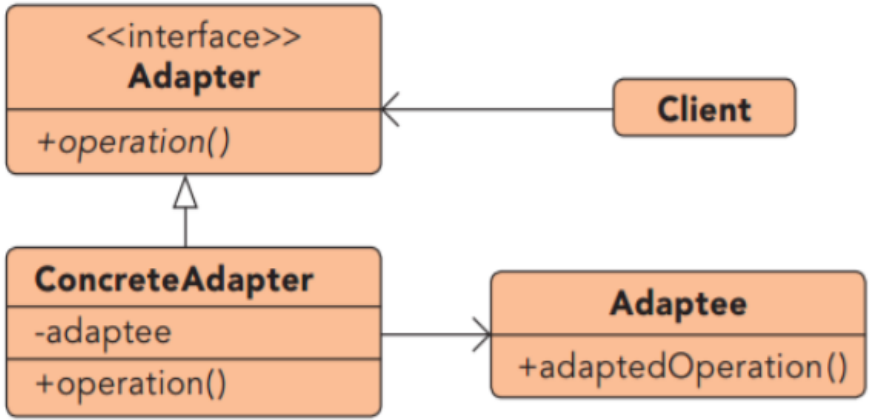
- używane do tworzenia struktur między różnymi obiektami
- opisują sposoby łączenia klas i obiektów w większe struktury

Wzorce
strukturalne

- **adapter** – umożliwia współpracę dwóm klasom o niekompatybilnych interfejsach
- **decorator** – polega na opakowaniu oryginalnej klasy w nową klasę “dekorującą”
- **facade** – ujednolica dostęp do złożonego systemu poprzez wystawienie uproszczonego, uporządkowanego interfejsu programistycznego
- **proxy** – utworzenie obiektu zastępującego inny obiekt
- **composite** – składanie obiektów w taki sposób, aby klient widział wiele z nich jako pojedynczy obiekt

Wzorce
adapter

- umożliwia współpracę dwóm klasom o niekompatybilnych interfejsach
- przekształca interfejs jednej klasy w interfejs drugiej
- https://github.com/iluwatar/java-design-patterns/tree/master/adapter



Wzorce
adapter

Cel

- przekształca interfejs klasy do postaci, której oczekują klienci

Konsekwencje

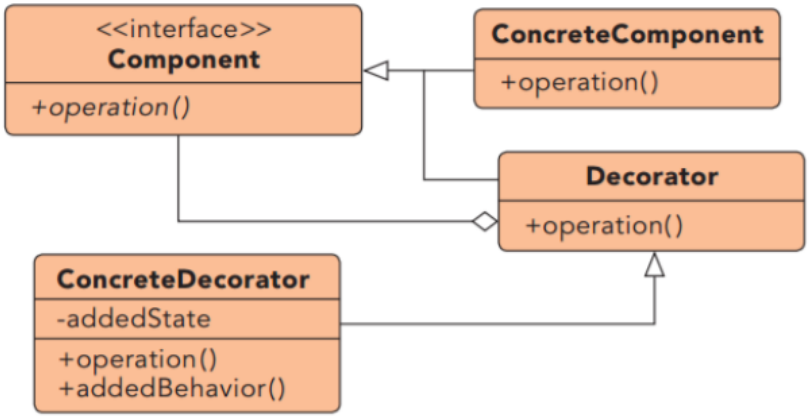
- może działać również z podklasami obiektu Adaptee
- zmiana zachowania obiektu Adaptee wymaga tworzenia podklas i odwoływania się obiektu Adapter bezpośrednio do nich
- możliwe pogorszenie wydajności obliczeniowej rozwiązania

Zastosowanie

- adaptowanie zewnętrznej biblioteki do własnych interfejsów

Wzorce
decorator

- polega na opakowaniu oryginalnej klasy w nową klasę “dekorującą”, zwykle przekazuje się oryginalny obiekt jako parametr konstruktora dekoratora
- https://github.com/iluwatar/java-design-patterns/tree/master/decorator



Wzorce
decorator

Cel

- opakowanie klasy, zmieniając dynamicznie jej zachowanie

Konsekwencje

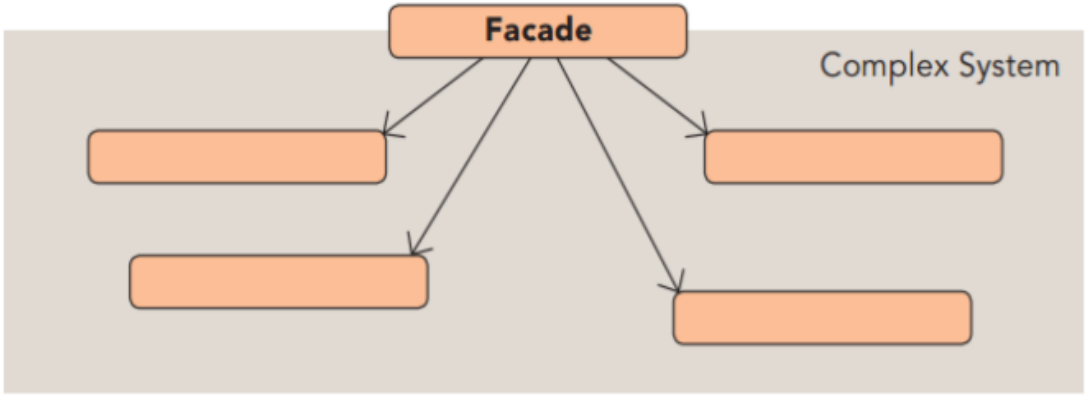
- dekorator dodaje funkcjonalności dynamicznie, co może utrudniać debuggowanie

Zastosowanie

- unikanie nadmiarowego tworzenia dziedziczenia w projekcie

Wzorce
facade

- dostarcza pojedynczy, uproszczony i uporządkowany interfejs do komunikacji z różnymi interfejsami złożonego systemu
- https://github.com/iluwatar/java-design-patterns/tree/master/facade



Wzorce
facade

Cel

- zapewnia jednolity interfejs dla podsystemu

Konsekwencje

- wzorzec zmniejsza ilość obiektów, z którymi klient podsystemu musi współpracować
- tworzy słabe powiązanie klienta z podsystemem, co umożliwia modyfikowanie podsystemu bez zmian w kliencie

Zastosowanie

- ułatwienie korzystania ze skomplikowanego podsystemu
- API biblioteki definiujące jej zunifikowany i uproszczony interfejs

Katalog wzorców
- wzorce behawioralne

Wzorce
behawioralne

- dotyczą interakcji pomiędzy klasami i obiektami
- omawiają sposoby podziału odpowiedzialności pomiędzy klasami

Wzorce
behawioralne

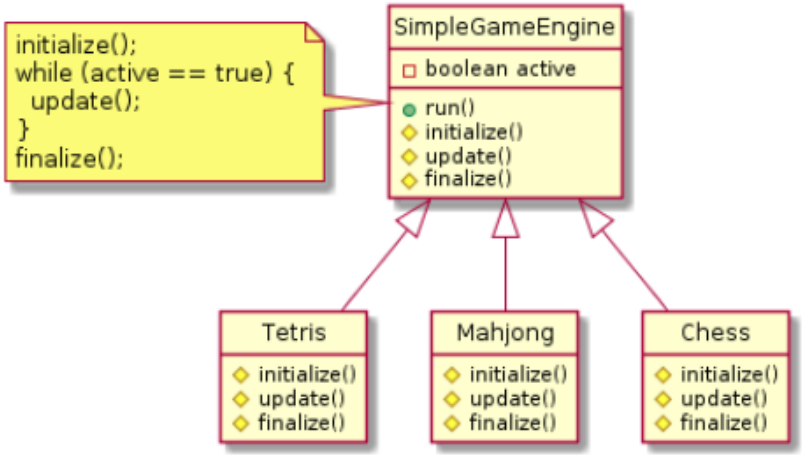
- **command** – traktuje żądanie wykonania określonej czynności jako obiekt, dzięki czemu mogą być parametryzowane w zależności od rodzaju odbiorcy, a także umieszczane w kolejkach
- **strategy** – definiuje rodzinę wymiennych algorytmów i „kapsułkuje” je w postaci klas; umożliwia wymienne stosowanie każdego z nich w trakcie działania aplikacji
- **observer** – używany do powiadamiania zainteresowanych obiektów o zmianie stanu pewnego innego obiektu
- **chain of responsibility** – umożliwia przetwarzanie żądania przez różne obiekty, w zależności od typu

Wzorce
behawioralne

- **template method** – definiuje metodę, będącą szkieletem algorytmu, algorytm może być następnie dokładnie definiowany w klasach pochodnych
- **iterator** – zapewnia sekwencyjny dostęp do podobiektów zgrupowanych w większym obiekcie
- **state** – umożliwia zmianę zachowania obiektu poprzez zmianę jego stanu wewnętrznego
- **memento** – zapamiętuje i udostępnia na zewnątrz wewnętrzny stan obiektu bez naruszania hermetyzacji, umożliwia to przywracanie zapamiętanego stanu obiektu

Wzorce
template method

- definiuje szkielet algorytmu, algorytm ten może być następnie dokładnie definiowany w klasach pochodnych
- https://github.com/iluwatar/java-design-patterns/tree/master/template-method



Wzorce
template method

Cel

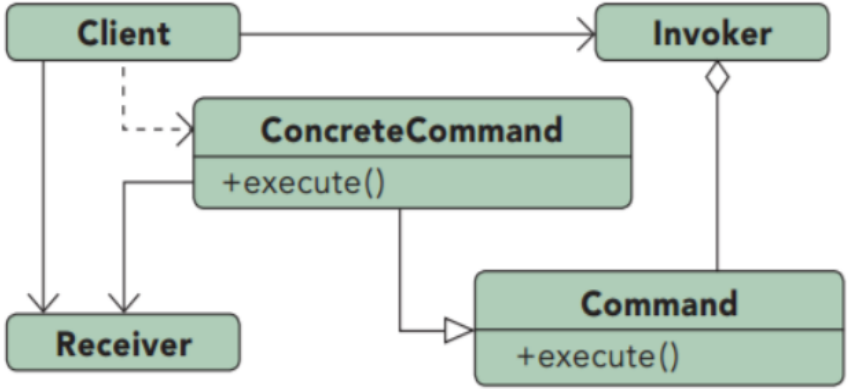
- umożliwia podklasom przedefiniowanie pewnych kroków algorytmu bez zmiany jego struktury

Konsekwencje

- ważne jest określenie operacji, które mogą być przedefiniowane, oraz tych, które muszą
- podklasy mogą rozszerzać działanie operacji z nadklasy poprzez przeciążanie jej i jawne wywołanie
- klasa nadrzędna definiuje wymagane kroki algorytmu i ich kolejność

Wzorce
command

- traktuje żądanie wykonania określonej czynności jako obiekt, dzięki czemu mogą być one parametryzowane lub umieszczane w kolejkach lub logach
- https://github.com/iluwatar/java-design-patterns/tree/master/command



Wzorce
command

Cel

- hermetyzuje żądania w postaci obiektów

Konsekwencje

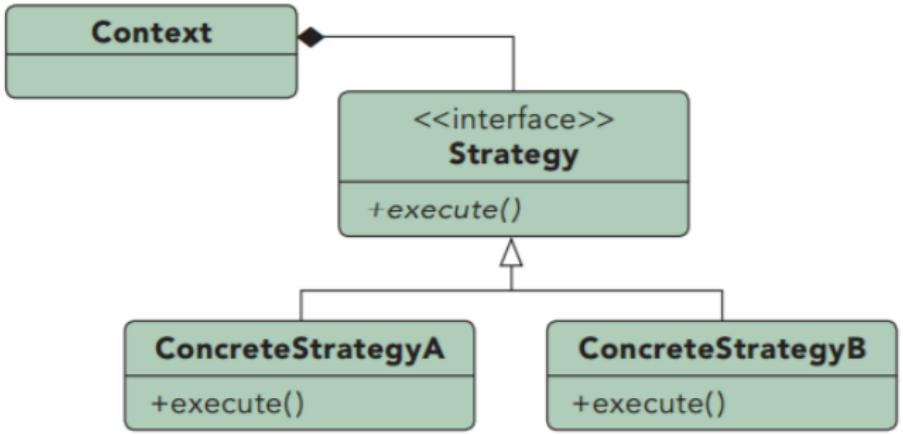
- separuje obiekt wywołujący polecenie od obiektu, który wie jak je zrealizować
- obiekty Command mogą być rozszerzane

Zastosowanie

- kolejkowanie zadań

Wzorce
strategy

- definiuje rodzinę wymiennych algorytmów i kapsułkuje je w postaci klas, umożliwia wymienne stosowanie każdego z nich w trakcie działania aplikacji, niezależnie od klientów
- https://github.com/iluwatar/java-design-patterns/tree/master/strategy



Wzorce
strategy

Cel

- tworzy rodzinę podobnych algorytmów i daje możliwość ich podmiany w trakcie działania programu

Zastosowanie

- implementacja jednego algorytmu na różne sposoby
- serializowanie danych do plików o różnych formatach
- zapis grafiki z wykorzystaniem różnych rodzajów algorytmów

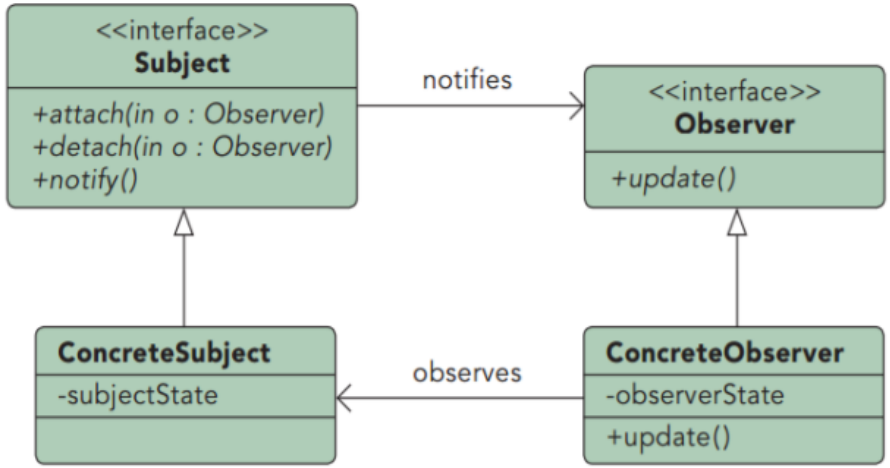
Wzorce
strategy

Konsekwencje

- definiuje rodzinę algorytmów powiązanych ze sobą
- wykorzystanie Strategy może być alternatywą dla korzystania z dziedziczenia w celu wyodrębnienia algorytmów
- hermetyzacja algorytmu w osobnych klasach ConcreteStrategy umożliwia jego modyfikowanie niezależnie od klasy Context
- eliminuje przeładowane instrukcje warunkowe
- może doprowadzić do powstania dużej ilości małych obiektów w systemie

Wzorce
observer

- używany do powiadamiania zainteresowanych obiektów o zmianie stanu obserwowanego obiektu
- https://github.com/iluwatar/java-design-patterns/tree/master/observer



Wzorce
observer

Cel

- umożliwia powiadamianie grupy obiektów o zmianie stanu obiektu obserwowanego

Konsekwencje

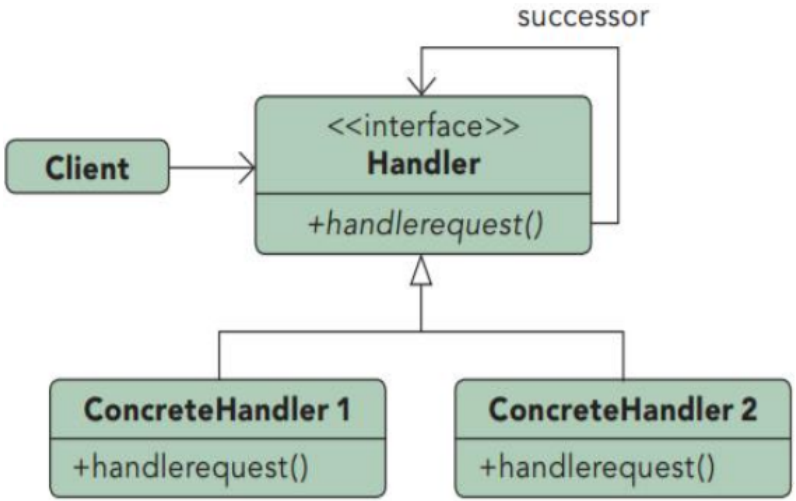
- zapewnia luźne powiązanie między obiektami
- umożliwia niezależne wymienianie obiektów obserwowanych i obserwatorów
- prosta operacja może wywołać kaskadę kosztownych uaktualnień

Zastosowanie

- informowanie o wydarzeniach zachodzących w systemie
- informowanie widoku i zmianach w modelu danych

Wzorce
chain of responsibility

- umożliwia wielu obiektom obsługę żądania poprzez połączenie tych obiektów
- https://github.com/iluwatar/java-design-patterns/tree/master/chain



Wzorce
chain of responsibility

Cel

- tworzy łańcuch odbiorców i przekazuje wzdłuż niego żądanie, aż któryś obiekt je obsłuży
- separuje nadawcę żądania od jego odbiorców

Konsekwencje

- nadawca i odbiorca żądania nie są ze sobą powiązani
- można dynamicznie dodawać/usuwać obiekty obsługujące
- zła konfiguracja łańcucha - żądanie może zostać nieobsłużone

Zastosowanie

- filtrowanie wyników skrzynki mailowej

Podsumowanie

Wzorce
podsumowanie

- jest wiele sposobów implementowania każdego wzorca
- diagram dołączany do opisu wzorca jest tylko przykładem, a nie specyfikacją
- wzorce są podstawą, którą można próbować rozszerzać
- jest wiele podobieństw między wzorcami
- wzorce występują grupowo, często współpracując ze sobą
  - *Strategy* może być tworzony przez *Factory Method*
  - *Decorator* może dynamicznie rozszerzać *Command*

Wzorce
różnice i podobieństwa

- **Strategy**
  - hermetyzuje algorytm
  - umożliwia jego dynamiczne podmienianie

- **Command**
  - hermetyzuje żądanie
  - umożliwia wykonanie żądania przez dowolnego klienta
  - umożliwia kolejkowanie żądań i wykonanie ich w późniejszym czasie

Wzorce
różnice i podobieństwa

- **Simple Factory**
  - tworzy gotowy obiekt
  - potrafi tworzyć różne obiekty na podstawie parametrów

- **Builder**
  - składa gotowy obiekt z części
  - tworzy obiekty podobne, różniące się elementami składowymi

Wzorce
współpracujące

- obiekty *Strategy* są często produkowane przez obiekty *Factory*
- wzorzec *Strategy* może korzystać z wzorca *Template Method*
- obiekt *Strategy* może korzystać ze wzorca *Observer*, aby informować inne obiekty np. o postępie w wykonaniu algorytmu
- *Chain of Responsibility* może używać komend do reprezentowania żądań w postaci obiektów
- obiekt *Factory* może posłużyć do dynamicznego rozszerzania odpowiedzialności obiektów *Command*

Anty-wzorce

Wzorce
anty wzorce

- antywzorzec pokazuje jak dojść od problemu do złego rozwiązania
  - dlaczego złe rozwiązanie wydaje się korzystne
  - długofalowe skutki takiego rozwiązania
  - wzorce, które mogą doprowadzić do dobrego rozwiązania
- http://sahandsaba.com/nine-anti-patterns-every-programmer-should-be-aware-of-with-examples.html

Wzorce
anty wzorce

- **Spaghetti Code**
  - kod staje się nieczytelny na skutek używania złożonych struktur
- **The Blob (God Class)**
  - jedna klasa implementuje zachowanie całej aplikacji, podczas gdy inne przechowują dane
- **Golden Hammer**
  - jedno narzędzie jest używane do rozwiązywania większości problemów

Thanks!

Q&A

tomasz.lisowski@protonmail.ch