Krzysztof Regulski
AGH, WIMiIP

WZORCE CZYNNOŚCIOWE

- PSK - projektowanie systemów komputerowych, notatki w Internecie, Beata Frączek, http://brasil.cel.agh.edu.pl/~09sbfraczek
- http://pl.wikipedia.org
- http://cpp0x.pl/kursy/Wzorce-projektowe/Wzorce-czynnosciowe/157

wzorce czynnościowe (behawioralne)

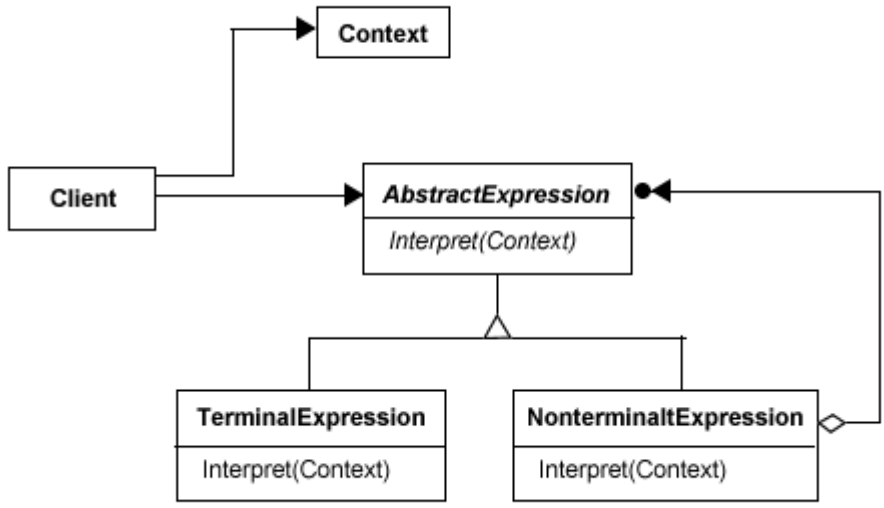
- opisują algorytmy i przydział odpowiedzialności
- charakteryzują sposób interakcji między obiektami
- *Interpreter, Metoda Szablonowa (Template Method), Iterator, Łańcuch Zobowiązań (Chain of Responsibility), Mediator, Obserwator (Observer), Odwiedziający (Visitor), Pamiątka (Memento), Polecenie (Command), Stan (State), Strategia (Strategy)*

1) Interpreter

Idea

- Celem jest zdefiniowanie opisu gramatyki pewnego języka interpretowalnego a także stworzenie dla niej interpretera, dzięki któremu będzie możliwe rozwiązanie opisanego problemu
- Jedna podklasa dla każdego symbolu w gramatyce formalnej
- Klasy te współdzielą jeden obiekt kontekstu, z którego pobierają dane wejściowe, i w którym zapisują wartości zmiennych
- Interpreter tworzy efektywny mechanizm przetwarzania danych wyjściowych

Struktura



Uczestnicy

- Client - tworzy abstrakcyjne drzewo składni przedstawiające zdanie w języku i uruchamia operację interpretacji.
- Context - przechowuje informacje globalne (np. wartości zmiennych).

- AbstractExpression - definiuje operację (lub operacje) "interpretowania", czyli pojedynczy węzeł w drzewie składni abstrakcyjnej.
- TerminalExpression - implementuje operację dla symbolu terminalnego (który występuje w przetwarzanych danych wewnętrznych).
- NonterminalExpression - implementuje operację (regułę gramatyczną) dla symbolu nieterminalnego.

Budowa (podstawowe elementy):

- Kontekst: Informacja globalna (np. wartości zmiennych).
- Wyrażenie abstrakcyjne: Definiuje operację (lub operacje) "interpretowania", czyli pojedynczy węzeł w drzewie składni abstrakcyjnej.
- Wyrażenie terminalne: Implementuje operację dla symbolu terminalnego (który występuje w przetwarzanych danych wewnętrznych).
- Wyrażenie nieterminalne: Implementuje operację (regułę gramatyczną) dla symbolu nieterminalnego.

Zastosowanie

Omawiany wzorzec projektowy można wykorzystać w sytuacjach, gdy zadania, zapisane w pewnym interpretowalnym języku, mogą być reprezentowane jako drzewa składniowe oraz istnieje prosta gramatyka opisująca ten język. Do przykładowych zastosowań tego wzorca należy interpretacja rzymskiego systemu liczbowego, interpretacja wyrażeń zapisanych w odwrotnej notacji polskiej oraz sprawdzanie poprawności pewnych reguł. Stosowany jest także w kompilatorach (np. kompilatorze języka Smalltalk).

Zalety

Modyfikowanie gotowej gramatyki jest stosunkowo proste - wystarczy utworzyć nowe klasy, które będą reprezentowały nowe produkcje. Przedstawianie każdej reguły gramatyki w klasie sprawia, że jest prosty w implementacji. W przypadku "oskryptowania" wielu elementów działania systemu unika się długiego procesu rekompilacji czy restartu systemu.

Wady

Jest niepraktyczny jeśli gramatyka składa się z więcej niż kilku produkcji (wzrasta wtedy liczebność klas)

2) Łańcuch zobowiązań. Łańcuch odpowiedzialności, ang. Chain of Responsibility

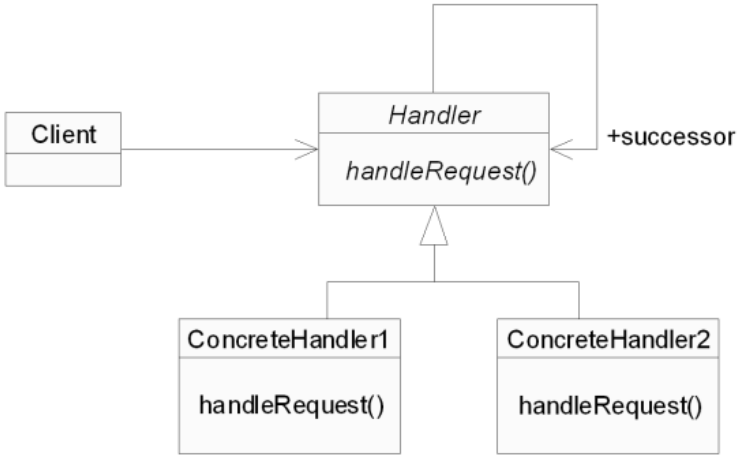
Idea

Łańcuch zobowiązań (ang. chain of responsibility) - wzorzec projektowy, który automatycznie przekazuje odpowiedzialność za wykonanie zadania do następnika, jeżeli obecny obiekt nie potrafi obsłużyć otrzymanego zadania. W przypadku gdy następnik nie istnieje - zadanie zostaje odrzucone. Łańcuch zobowiązań można wyobrazić sobie jako listę obiektów obsługujących, którą przechodzimy dopóki nie znajdziemy obiektu, który będzie w stanie obsłużyć żądane zadanie. Wzorzec znajduje zastosowanie wszędzie tam, gdzie mamy do czynienia z różnymi mechanizmami podobnych żądań, które można zaklasyfikować do różnych kategorii. Dodatkową motywacją do jego użycia są często zmieniające się wymagania.

Cel:

- Usunięcie powiązania pomiędzy nadawcą i odbiorcą żądania
- Umożliwienie wielu obiektom obsługi żądania
- Komunikat przekazywany jest pomiędzy obiektami należącymi do pewnego zbioru zgodnie z precyzyjnie wyznaczoną trasą i kolejnością.
- Odpowiedzialność za przetworzenie komunikatu spada na obiekt, który jest do tego zadania najlepiej przygotowany.

Struktura



Obiekty Handler tworzą listę jednokierunkową (łańcuch), wzdłuż której są przekazywane żądania. Struktura tego wzorca jest bardzo prosta: obiekty typu Handler są powiązane ze sobą w postaci jednokierunkowej kolejki (albo łańcucha). Nadchodzące od klienta żądanie jest przekazywane wzdłuż tego łańcucha, gdzie każdy obiekt typu Handler ma szansę na ich obsłużenie. Co ważne, obiekty typu Handler są od siebie niezależne, tzn. nie wiedzą o sobie nic (poza abstrakcyjnym wskazaniem na obiekt następnika).

Uczestnicy

- Handler - definiuje interfejs do obsługi żądań
- Concrete Handler - obsługuje jeden rodzaj żądania, pozostałe przekazuje do następnika w łańcuchu, posiada referencję typu Handler do następnika
- Client - inicjuje przetwarzanie, przekazując żądanie do pierwszego obiektu Handler w łańcuchu

Handler definiuje interfejs obsługi żądań. Zwykle jest to jedna metoda, która realizuje prosty algorytm: jeżeli dany obiekt ConcreteHandler jest w stanie obsłużyć żądanie, to obsługuje je; wprzeciwnym wypadku (bądź w sytuacji, gdy wiele obiektów typu Handler może obsłużyć jedno żądanie) - przekazuje je do swojego następnika w łańcuchu. Charakterystyczna dla wzorca jest dowolna konfigurowalność łańcucha: żaden jego element nie musi posiadać wiedzy o rodzaju żądań obsługiwanych przez kolejne elementy, dlatego zmiany w jego strukturze nie mają wpływu na zachowanie. Zadaniem klienta przy takiej strukturze jest przekazanie żądania pierwszemu elementowi łańcucha, który następnie dalej obsługuje żądanie.

Konsekwencje

- Ograniczone powiązania:

        - Klient i każdy obiekt Handler nie wiedzą, który z pozostałych obiektów Handler obsługuje dany typ żądania
        - nadawca i odbiorca żądania nie mają o sobie żadnej wiedzy
- Możliwość elastycznego przydziału odpowiedzialności do obiektów Handler
- Ułatwione testowanie
- Brak gwarancji obsłużenia żądania

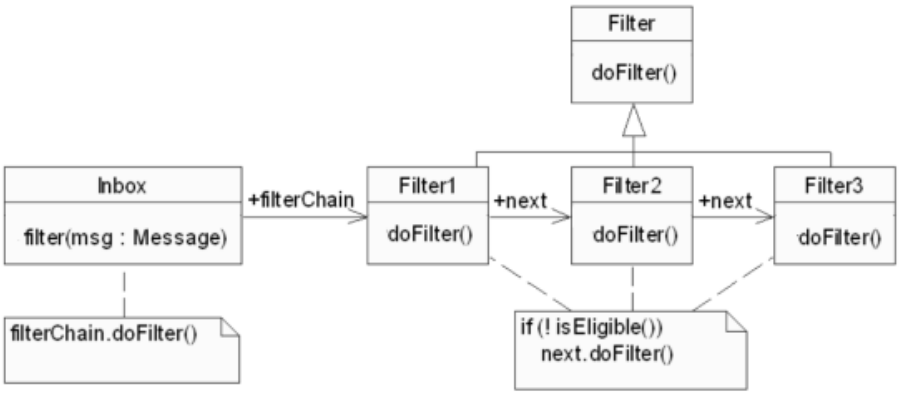
Zalety:

- elementy łańcucha mogą być dynamicznie dodawane i usuwane w trakcie działania programu,
- zmniejszenie liczby zależności między nadawcą, a odbiorcami,
- implementacja pojedynczej procedury nie musi znać struktury łańcucha oraz innych procedur.

Wady:

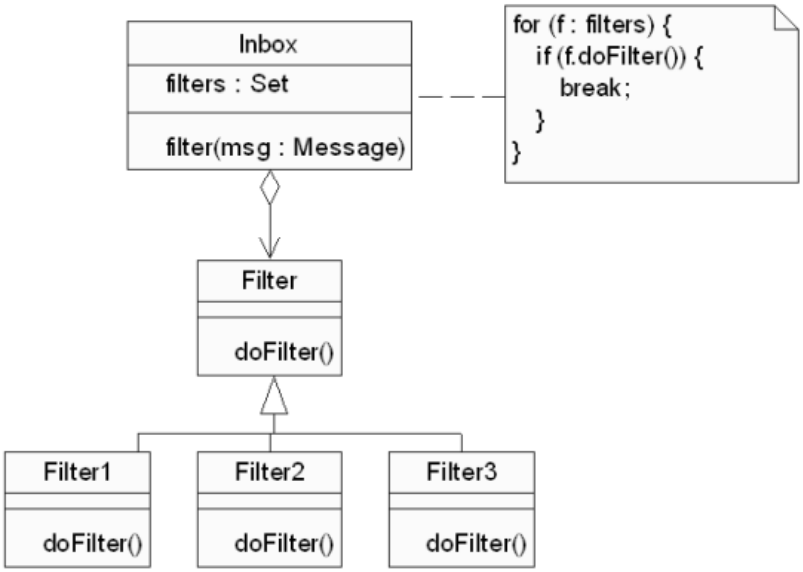
- wzorzec nie gwarantuje, że każde żądanie zostanie obsłużone,
- śledzenie i debugowanie pracy działania łańcucha może być trudne.

Przykład 1



Obiekt Inbox wywołuje pierwszy obiekt Filter w łańcuchu. Kolejne filtry przekazują sobie sterowanie Prostym przykładem tego wzorca jest np. mechanizm filtrów obecnych w większości klientów poczty elektronicznej. Wiadomość przychodząca do foldera Inbox jest przesyłana przez łańcuch zdefiniowanych przez użytkownika filtrów: każdy z nich może dokonać pewnej akcji na wiadomości, polegającej na przeniesieniu jej do innego foldera, zmianie jej priorytetu czy usunięciu jej. Każdy filtr podejmuje decyzję (poprzez wywołanie metody isEligible()), czy konkretna wiadomość powinna być przez niego obsłużona, i przekazuje sterowanie dalej.

Przykład 2



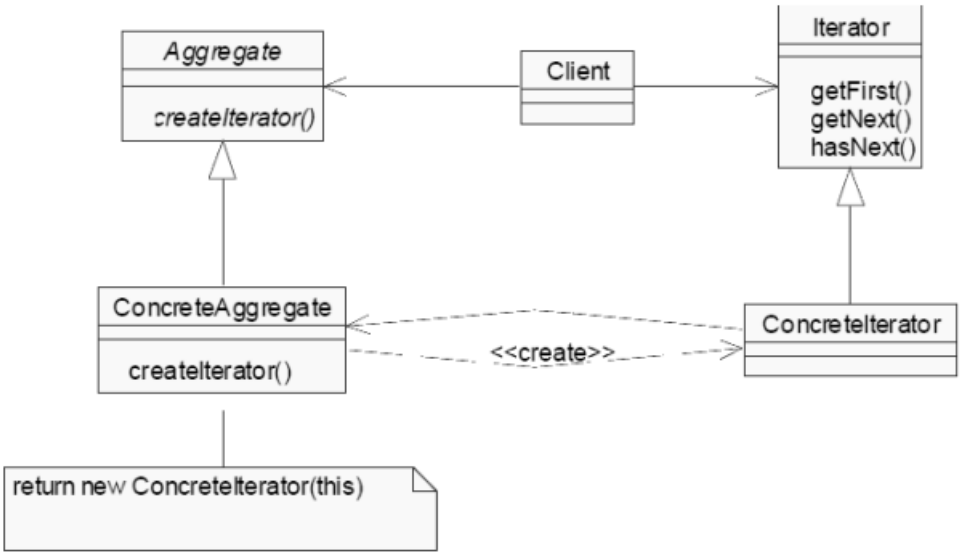
Obiekt Inbox wywołuje kolejno obiekty Filter. Nie występuje bezpośrenie przekazywanie sterowania z jednego filtra do drugiego. Z uwagi na wymienione wcześniej niedogodności, przede wszystkim możliwość przerwania łańcucha sterowania, możliwa jest także inna struktura przetwarzania, która nie posiada już topologii łańcucha. W tym rozwiązaniu pojawia się nowa rola: zarządcy, który posiada referencje do wszystkich filtrów. Zarządca (w tym przypadku jest nim także obiekt Inbox) wywołuje po kolei wszystkie filtry, które obsługują daną wiadomość lub nie. Jednak dzięki temu, że filtry nie przekazują sobie bezpośrednio sterowania, nie ma możliwości przerwania łańcucha, a ponadto informacja o nieobsłużeniu żądania może być w łatwy sposób przedstawiona klientowi przez zarządcę.

3) Iterator

Cel

Rożnorodność kolekcji obiektowych (listy, kolejki, stosy, zbiory, multizbiory, mapy etc.), zarówno funkcjonalna, jak i implementacyjna, powoduje, że wiele z nich wymaga specyficznej obsługi i stosowania zróżnicowanych metod dostępu do elementów. Wzorzec Iterator odpowiada na potrzebę zunifikowanego dostępu do elementów kolekcji, który pozwoli pominąć różnice w ich implementacji. Dzięki niemu, niezależnie od rodzaju kolekcji, jej elementy mogą być przetwarzane sekwencyjnie, z zachowaniem własności poszczególnych kolekcji. Celem jest *zapewnienie sekwencyjnego dostępu do podobiektów zgrupowanych w większym obiekcie*, w taki sposób, aby struktura obiektu pozostała nieznana dla klienta.

Struktura



Wzorzec Iterator składa się z dwoch klas abstrakcyjnych: Aggregate i Iterator, oraz dwóch klas konkretnych: ConcreteAggregate i ConcreteIterator. Wszystkie kolekcje są implementacją interfejsu Aggregate, tzn. posiadają metodę tworzącą iterator. Iterator, podobnie jak Aggregate, jest jedynie specyfikacją interfejsu, jaki każdy iterator musi posiadać. Klient, odwołując się do metody createIterator() w kolekcji, otrzymuje klasę implementującą interfejs Iterator. Dzięki temu klient nie zna konkretnej klasy implementacyjnej, a jedynie interfejs, do którego musi się odwoływać. Taka sytuacja ma miejsce np. w bibliotece Java Collections: każda kolekcji tworzy swój własny iterator, który jednak jest dostępny wyłącznie poprzez wspólny interfejs Iterator. W ten sposób mogą one być traktowane w jednolity sposób. Iterator posiada wewnętrzny wskaźnik, ktory wskazuje na aktualny element kolekcji. Iteratory definiują podstawowe operacje pozwalające na sekwencyjny dostęp do wszystkich elementów dowolnej kolekcji: getFirst() - ustawiająca wskaźnik iteratora na początek kolekcji, getNext() - zwracająca kolejny element, hasNext() - sprawdzająca, czy kolejny element istnieje. W niektórych implementacjach iterator pozwala także na modyfikacje kolekcji, np. dodawanie i usuwanie elementów. Kolekcje o specyficznej strukturze, np. listy mogą udostępniać itera tory wykorzystujące wiedzę o tej strukturze, np. udostępniającą możliwość swobodnego dostępu do elementów kolekcji, zmiany kierunku trawersu kolekcji etc. Z uwagi na konieczność dostępu do elementów kolekcji, iterator musi posiadać prawo odwołania się do nich. W praktyce jest on zatem zwykle klasą zaprzyjaźnioną lub wewnętrzną kolekcji.

Uczestnicy

We wzorcu uczestniczą dwie hierarchie obiektów: związanych z kolekcjami (Aggregate i jej klasy potomne) i związanych z iteracją (Iterator i jego podklasy). Obie hierarchie są powiązane ze sobą wyłącznie poprzez interfejsy. Warto zwrócić uwagę, że struktura wzorca i role pełnione przez poszczególne klasy są szczególnym przypadkiem struktury i rol zdefiniowanych we wzoru Factory Method. Tam również klient odwołuje się do abstrakcyjnej metody klasy-fabryki w celu otrzymania abstrakcyjnego produktu, a faktycznie wywołuje metody w implementacji klasy-fabryki i otrzymuje konkretny produkt zależny od użytej fabryki.

Konsekwencje

Iterator pozwala na oddzielenie kolekcji, czyli klasy związanej z przechowywaniem obiektów, od mechanizmu dostępu do tych obiektów. Dzięki temu klient odwołuje się do obiektów w sposób abstrakcyjny, niezależny od konkretnej implementacji kolekcji. Konstrukcja iteratora pozwala na jednoczesne współistnienie wielu niezależnych iteratorow, ponieważ każdy przechowuje wewnętrznie wskaźnik do aktualnie wskazywanego obiektu w kolekcji. Niektóre kolekcje mogą definiować kilka rożnych iteratorow, o zróżnicowanej funkcjonalności (w przypadku np. listy).

4) Mediator

Cel

- Uproszczenie komunikacji wielu obiektów
- Hermetyzacja mechanizmu wymiany komunikatów.

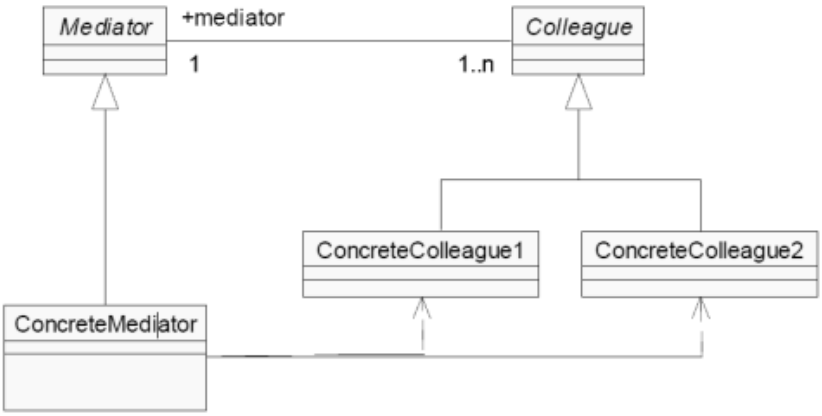
► Mediator znajduje zastosowanie w sytuacji, gdy wiele obiektów o wspólnym interfejsie musi komunikować się ze sobą w celu wykonania określonego zadania.

► Mediator zapewnia jednolity interfejs do różnych elementów danego podsystemu.

► Najprostszym, lecz trochę naiwnym rozwiązaniem jest powiązanie wszystkich obiektów ze sobą w topologii grafu pełnego. Takie rozwiązanie jest jednak słabo skalowalne: dołączenie kolejnego obiektu powoduje konieczność powiadomienia o zmianie wszystkich pozostałych, aby potrafili skomunikować się z nowym uczestnikiem interakcji. Ponadto powoduje, że mechanizm komunikacji jest rozproszony, co utrudnia jego modyfikację i dalszy rozwój.

► Wzorzec mediatora umożliwia zmniejszenie liczby powiązań między różnymi klasami, poprzez utworzenie mediatora będącego jedyną klasą, która dokładnie zna metody wszystkich innych klas, którymi zarządza. Nie muszą one nic o sobie wiedzieć, jedynie przekazują polecenia mediatorowi, a ten rozsyła je do odpowiednich obiektów.

Struktura



► Obiekty Colleague i obiekt Mediator tworzą topologię gwiazdy. Komunikaty między obiektami Colleague są przekazywane za pośrednictwem mediatora

► Wzorzec Mediator proponuje topologię gwiazdy, w której centrum znajduje się właśnie obiekt Mediator. Posiada on referencje do pozostałych obiektów (Colleague) i zna ich zakres odpowiedzialności. Komunikacja pomiędzy obiektami Colleague wymaga pośrednictwa Mediatora, który potrafi przekazać komunikat do właściwego odbiorcy.

Uczestnicy

- Mediator
  - definiuje interfejs dołączania i odłączania kolegów
- Concrete Mediator
  - implementuje mechanizm komunikacji pomiędzy obiektami Colleague
  - posiada referencje do zarejestrowanych obiektów Colleague
- Colleague
  - definiuje wspólny interfejs dla komunikujących się obiektów
  - posiada referencję do obiektu Mediator
  - komunikuje się z innymi obiektami za pośrednictwem obiektu Mediator

Mediator posiada metody służące do dołączania i odłączania obiektów Colleague. Ponadto jego zadaniem jest implementacja mechanizmu komunikacji, czyli podejmowanie decyzji który z obiektów Colleague powinien wykonać określone żądanie. Obiekty Colleague nie są obciążone zadaniem komunikacji z pozostałymi obiektami. Ich wiedza jest ograniczona do znajomości Mediatora. Także dołączenie i odłączenie obiektu Colleague wymaga jedynie powiadomienia Mediatora, a nie wszystkich obiektów. Struktura ta jest przybliżoną analogią do sieci komputerowych, w których komputery znajdujące się w różnych podsieciach komunikują się za pośrednictwem routera. Poszczególne Komputery nie muszą znać adresów wszystkich innych komputerów na świecie, a jedynie adres najbliższego routera.

Konsekwencje

- Centralizacja mechanizmu komunikacji
  - wyłączna odpowiedzialność obiektu Mediator
  - zmiana mechanizmu wymaga tylko zmiany Mediatora
  - prostota komunikacji vs. złożoność Mediatora
- Niezależność obiektów Colleague od siebie
- Uproszczenie protokołów obiektowych
  - Zamiana relacji wiele-wiele na relacje jeden-wiele

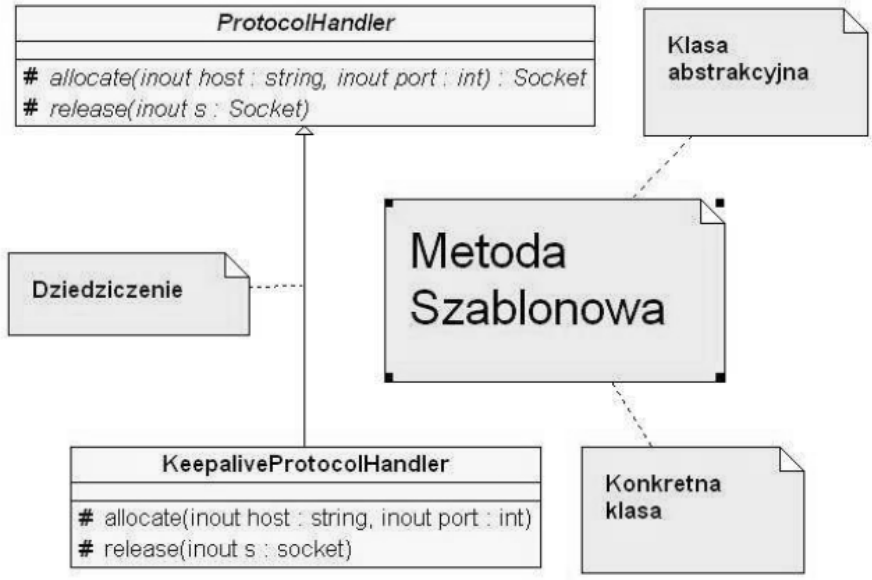
Mediator narzuca centralizację mechanizmu komunikacji. Odpowiedzialność za komunikację przejmuje w całości Mediator, co z jednej strony pozwala w łatwy sposób modyfikować go lub wymieniać, z drugiej jednak powoduje znaczny wzrost złożoności tego obiektu. Wydaje się jednak, że zamiana taka jest opłacalna, ponieważ uwalnia od problemów związanych z komunikacją resztę obiektów w systemie. Drugą ważną zaletą wzorca jest uniezależnienie obiektów Colleague od siebie: nie posiadają one o sobie żadnej wiedzy, co pozwala modyfikować ich liczbę i funkcjonalność.

5) Metoda Szablonowa

Cel

- Przenosi część algorytmu z poziomu nadklasy do podklasy w przypadku, gdy opracowanie ogólnej metody okazało się niemożliwe. Wzorzec ten definiuje szkielet programu dla implementacji algorytmu. Algorytm jest abstrakcyjny i dopiero klasy potomne realizują jego pełne działanie.
- Niezmienna część algorytmu zostaje opisana w tzw. metodzie szablonowej, której nie wolno przesłaniać. W metodzie tej wywoływane są inne metody, reprezentujące zmienne fragmenty algorytmu. Metody te mogą być abstrakcyjne lub definiować domyślne zachowania. Programista, który chce skorzystać z algorytmu, musi utworzyć podklasę klasy z metodą szablonową i przesłonić metody opisujące zmienne fragmenty.
- Najczęściej metoda szablonowa ma widoczność publiczną, a metoda do przesłonięcia – widoczność chronioną.

Struktura



Zalety:

Udostępnia puste "punkty zaczepienia" na poziomie nadklasy, aby programista mógł do niej wstawić swoją funkcjonalność za pośrednictwem odpowiednich relacji dziedziczenia.

Wady:

Stosowanie tego wzorca jest w większości przypadków nieuzasadnione, lepszym rozwiązaniem jest użycie wzorca strategii. W systemach obiektowych stosowanie mechanizmu dziedziczenia do modyfikowania zachowań nadklasy jest nie zalecane.

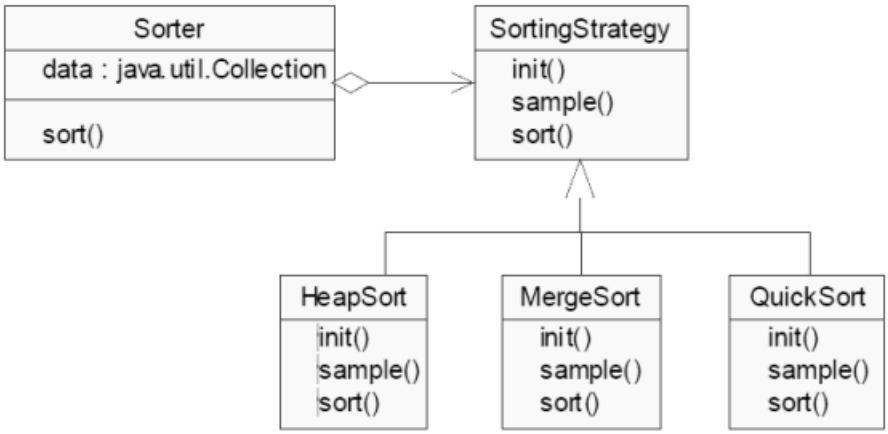
6) Strategia

- celem jest ukrycie implementacji grupy powiązanych ze sobą algorytmów oraz ułatwienia zmiany ich implementacji bez ingerencji w kod klienta
- dotyczy funkcjonalnej zmiany zachowania obiektu w trakcie wykonywania programu. W ten sposób pozornie obiekt ten zmienia klasę, do której należy.
- wzorzec może być wykorzystywany do takich rzeczy jak np. walidacja formularzy, filtrowanie danych czy też sortowanie danych.
- Wzorzec projektowy *strategy* stosuje się gdy jesteśmy w stanie wyłonić grupę algorytmów powiązanych ze sobą. Zastosowanie wzorca zapewnia możliwość łatwej rozbudowy aplikacji o nowe algorytmy bez konieczności zmiany jego architektury.
- algorytm obiektu nie może być zmieniany automatycznie np. w zależności od wartości zmiennych. Oznacza to, że algorytm musi być zmieniany spoza klasy.

Przykład zastosowania

- Przy projektowaniu architektury systemów informatycznych często natrafia się na miejsca, w których następuje potrzeba wybrania jednego z kilku sposobów wykonania jakiegoś cząstkowego zadania. Przykładowo:
  - aplikacja sklepu internetowego musi we właściwy sposób naliczyć podatek w zależności od kraju klienta
  - formularz zakładania konta musi w odpowiedni sposób (w zależności od typu oczekiwanych danych) walidować każde z pól
  - w zależności od polecenia użytkownika dane muszą być zaprezentowane jako strona WWW, dokument PDF lub obrazek PNG
- Najbardziej oczywistym sposobem rozwiązania tego problemu wydaje się zastosowanie którejś z instrukcji warunkowych wybierającej żądany algorytm. Jednak takie rozwiązanie jest mało elastyczne - zarówno warunki wyboru jak i lista algorytmów są trwale wpisane w program.
- W językach obiektowych lepszym wyjściem jest hermetyzacja zmienności algorytmów przez zdefiniowanie bazowej klasy abstrakcyjnej, z której dziedziczą konkretne klasy implementujące poszczególne algorytmy. Obiekt wykorzystujący algorytm zawiera wtedy pole przechowujące instancję klasy konkretnej (lub wręcz ciąg znaków identyfikujący nazwę klasy konkretnej jeśli są one przechowywanymi w puli singeltonami) i może dzięki polimorfizmowi wykonać go nie wiedząc zupełnie który konkretnie algorytm jest stosowany.
- W razie, gdyby istniała potrzeba dodania nowego algorytmu, zastosowanie tego wzorca zapewnia możliwość łatwej rozbudowy programu bez konieczności zmieniania jego architektury. Należy po prostu dodać kod nowej klasy i przekazać jej instancję do obiektu wykorzystującego algorytm.

Struktura

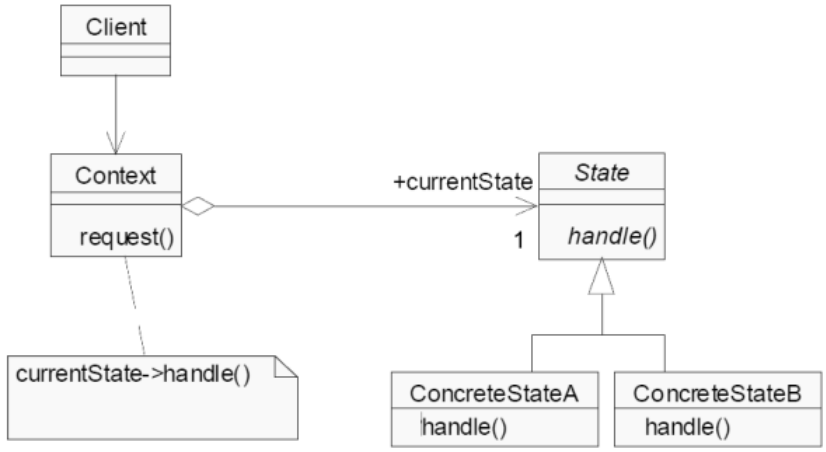


- Sorter zleca operację wybranej strategii sortowania. Każda strategia to jeden algorytm. Zmiana strategii nie wpływa na obiekt Sorter.
- Klasa Sorter wykonuje sortowanie wewnętrznej kolekcji. Ponieważ istnieją różne algorytmy sortowania, dlatego realizacja metody sort() jest delegowana do aktywnego algorytmu, stanowiącego implementację klasy SortingStrategy. Metody tej klasy to kroki algorytmu. Każdy algorytm może realizować je w charakterystyczny dla siebie sposób. Zmiana algorytmu sortowania jest realizowana wyłącznie przez zmianę obiektu reprezentującego ten algorytm: jest przezroczysta z punktu widzenia obiektu Sorter.

7) Stan

- Automatycznie kontroluje wewnętrzne zachowanie obiektu, nadając mu odpowiedni stan i funkcjonalność.
- na celu ułatwić rozbudowę obiektu o nowe stany. Wraz ze zmianą stanu następuje zmiana zachowania obiektu.
- Wszystkie stany dla danego obiektu muszą posiadać wspólny interfejs. Stany są obiektami tymczasowymi. Dane, które mają być trwałe muszą zostać zapisane do obiektu głównego. Dane tymczasowe powinny być trzymane w stanie do czasu uzyskania kompletnego zestawu danych, a następnie przekazane do obiektu głównego.
- stan jest wzorcem projektowym o takiej samej konstrukcji jak strategia. Istnieje jednak między nimi istotna różnica w zasadzie działania: Wzorzec projektowy state powinien posiadać następujące własności:
  - stan obiektu może ulec zmianie w wyniku wykonania jakiegoś zadania bądź upływu określonej ilości czasu;
  - stan obiektu nie powinien być ustawiany spoza klasy - ewentualne zmiany stanów powinny odbywać się za pomocą metod, które dbają o prawidłową modyfikację stanu obiektu.

Struktura



Stan jest obiektem. Zmiana stanu oznacza zmianę obiektu go reprezentującego. Delegowane do niego metody są wywoływane polimorficznie.
W przypadku wzorca State centralnym obiektem jest Context. Jego metody wywoływane przez klientów delegują żądania do skojarzonego z nim relacją kompozycji obiektu typu State, reprezentującego jego stan. Metody obiektu State są polimorficzne, czyli wraz ze zmianą tego obiektu zmienia się też ich funkcjonalność. W ten sposób, gdy zachodzi zmiana skojarzonego z obiektem Context obiektu State, zmieniają się też zachowanie metod kontekstu. Pozornie zatem obiekt Context zmienia klasę, do której należy. Wzorzec Strategy stosuje podobne rozwiązanie, tylko na nieco większą skalę. Obiekt Context realizuje pewien algorytm, którego poszczególne kroki mogą zmieniać się w zależności od wyboru konkretnego algorytmu. Z obiektem tym skojarzony jest (także za pomocą kompozycji) obiekt algorytmu, którego metody implementują zmieniające się kroki. Zmiana obiektu algorytmu powoduje zmianę zachowania obiektu Context. W obu przypadkach najważniejszą zaletą jest możliwość zmiany skojarzonego obiektu (stanu lub algorytmu) w trakcie działania programu, bez potrzeby jego rekompilacji.

Uczestnicy

- Context
  - posiada referencję do obiektu reprezentującego bieżący stan
- State
  - definiuje interfejs pozwalający hermetyzować zachowanie związane z każdym stanem
- Concrete State
  - definiuje własne metody implementujące zachowanie specyficzne dla tego stanu

Obiekt Context posiada referencję do obiektu typu State, wskazującą na bieżący stan. W obiekcie State zdefiniowane są wszystkie metody, których zachowanie zależy od stanu obiektu Context.

Konsekwencje

- Podział zachowania obiektu wg stanów
  - kod związany ze jednym stanem jest zapisany w jednym obiekcie
- zmiana stanu jest realizowana przez zmianę obiektu stanu na inny

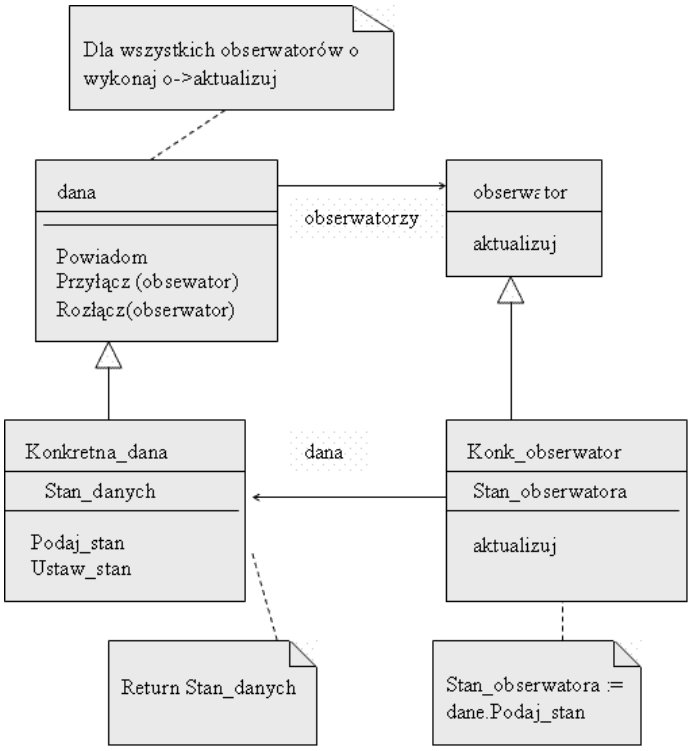
- ochrona przed stanem niespójnym
- możliwość współdzielenia obiektów State
  - obiekty State zwykle definiują tylko zachowanie
  - obiekty State zwykle są bezstanowe

Zastosowanie wzorca pozwala modyfikować zachowanie obiektów tak jakby zmieniała się ich klasa i to jest najważniejszy cel i konsekwencja tego wzorca. Istnieje natomiast grupa efektów pośrednich, ale o dość interesujących właściwościach. Hermetyzacja stanu w postaci niezależnych klas pozwala na jednorazową, niepodzielną zmianę tego stanu, bez wprowadzania stanów niespójnych czy nieoznaczonych. Jeżeli obiekty State nie przechowują informacji (w większości przypadków może ona być zapamiętana w obiekcie Context, ponieważ ona nie ulega zmianie), a jedynie definiują zachowanie, wówczas - paradoksalnie - obiekty te, reprezentujące stan, są bezstanowe i mogą być współdzielone między wiele obiektów Context.

8) Obserwator

- Wzorzec obserwator jest stosowany przy projektowaniu środowisk prezentacji danych.
- Używany jest do powiadamiania zainteresowanych obiektów o zmianie stanu pewnego innego obiektu.
- Obiekty klasy obserwator przekazują odpowiedzialność za monitorowane zdarzenia centralnemu obiektowi dana.
- Definiuje "jeden-do-wielu" zależności między obiektami. Zmiana stanu jednego obiektu automatycznie propaguje na związane z nim obiekty.

Struktura



Uczestnicy i współpracownicy:

Obiekt klasy `dana` wie, które obiekty klasy obserwator należy zawiadomić, ponieważ rejestrują się u niego. `Dana` zawiadamia obiekty klasy obserwator o wystąpieniu zdarzenia. Obiekty klasy `obserwator` są odpowiedzialne za zarejestrowanie się u obiektu `dana` i uzyskanie od niego potrzebnej informacji, gdy zostaną powiadomione o zdarzeniu.

Konsekwencje:

Obiekt klasy dana może niepotrzebnie powiadamiać o zdarzeniu różne rodzaje obserwatorów nawet wtedy, gdy są zainteresowane jego wystąpieniem tylko w pewnych przypadkach.

Implementacja:

Tworzy obiekty klasy obserwator, które mają być informowane o wystąpieniu pewnego zdarzenia i w tym celu rejestrują się u obiektu klasy dana. Kiedy zachodzi zdarzenie, dana powiadamia zarejestrowane obiekty klasy obserwator.

- Klasa dana - zna swoich obserwatorów, dowolna liczba obserwatorów może obserwować dane. Obserwatorzy są rejestrowani - przyłącz, rozłącz (wskazanie na klasę). Metoda powiadom informuje wszystkie zarejestrowane obiekty o zmianach wewnątrz niej.
- Klasa obserwator - definiuje interfejs dla obiektów, które mają być powiadamiane o zmianach w danych. Obserwator po otrzymaniu informacji o konieczności aktualizacji wysyła żądanie do klasy dana z prośbą o podanie aktualnego stanu. Powiązanie na poziomie klas konkretnych daje możliwość komunikacji od klas obserwujących do danych.
- Klasa Konk_obserwator - zawiera wskazanie na konkretne dane oraz stan, który powinien być spójny z danymi.
- Klasa Konkretna_dana - powiadamia obserwatorów o zmianie stanu, przechowuje stan

Zalety

- luźna zależność między obiektem obserwującym i obserwowanym. Ponieważ nie wiedzą one wiele o sobie nawzajem, mogą być niezależnie rozszerzane i rozbudowywane bez wpływu na drugą stronę,
- relacja między obiektem obserwowanym, a obserwatorem tworzona jest podczas wykonywania programu i może być dynamicznie zmieniana,
- Łatwość protokołu: obserwator tworzy regułę że jest on wyłącznie jedynym scentralizowanym punktem dostępu. Zależność jeden do wielu jest bardziej dogodna niż wiele do wielu między kolegami.
- domyślnie powiadomienie otrzymują wszystkie obiekty. Obiekt obserwowany jest zwolniony z zarządzania subskrypcją — o tym czy obsłużyć powiadomienie, decyduje sam obserwator.

Wady:

- obserwatorzy nie znają innych obserwatorów, co w pewnych sytuacjach może wywołać trudne do znalezienia skutki uboczne.
- Trudność rezygnacji: nie ma pewności że abonent nie zostanie powiadomiony o zdarzeniu już po swojej rezygnacji z abonamentu.

Przykłady

Konto bankowe - Rachunek klienta może być powiązany z innymi rachunkami pobocznymi. Na przykład, z głównym rachunkiem jest związany rachunek kredytu odnawialnego. Wysokość otwartej linii kredytowej zależy od środków na rachunku głównym. Zmiana wysokości salda na tym rachunku skutkuje podwyższeniem lub obniżeniem dostępnej kwoty kredytu.

Obserwator jest stosowany w aplikacjach z graficznym interfejsem użytkownika. Rozpatrzmy mechanizm kopiowania pliku oraz okienko graficzne obrazujące postęp prac. Mechanizm kopiujący jest niezależny od okienka i nie musi wiedzieć czy i w jaki sposób postępy są wyświetlane. Z drugiej strony, do poprawnego wyświetlania okienko potrzebuje informacji z mechanizmu kopiowania:

- ile bajtów danych już skopiowano,
- kiedy została skopiowana kolejna porcja danych.

Możemy zaimplementować w mechanizmie kopiowania interfejs Obserwowany, zaś w okienku - Obserwator i skomunikować je ze sobą. Mechanizm kopiowania będzie wywoływać metodę powiadomObserwatorow() po skopiowaniu bloku danych określonej wielkości, co spowoduje wysłanie powiadomienia do okienka i odświeżenie paska postępu.

Podsumowanie

Wzorzec obserwator wprowadza abstrakcyjne powiązania z podmiotem. Podmiot nie zna szczegółów działania żadnego z obserwatorów. Może więc się okazać, że wobec wystąpienia szeregu przyrostowych zmian danych podmiotu zostanie wysłana do obserwatora seria powtarzających się komunikatów, których obsługa wiązać się będzie ze zbyt dużym kosztem. Rozwiązaniem problemu będzie oczywiście wprowadzenie pewnej dodatkowej logiki, tak by informacje o zmianach nie były wysyłane zbyt wcześnie lub zbyt często. Inny problem występuje w przypadku, gdy zmiana danych podmiotu dokonywana jest przez pewne części kodu lub systemu zwane dalej klientami. Pojawia się wtedy pytanie, kto powinien inicjować wysłanie komunikatu o zmianach. Jeśli odpowiedzialny będzie za to, jak dotychczas, sam podmiot, to w przypadku wykonywania zmian przez kilku klientów znowu mogą pojawić się serie komunikatów o nieznacznych w istocie zmianach. Można ich uniknąć, jeśli to klient będzie informował podmiot, ze należy wysłać komunikat. Jeśli jednak któryś z klientów "zapomni" o poinformowaniu podmiotu, to program nie będzie już działał zgodnie z oczekiwaniami. Stosując wzorzec obserwator mona także zdefiniować kilka rodzajów komunikatów. W tym celu interfejs obserwatora może definiować kilka różnych metod powiadomienia. Dzięki temu w pewnych sytuacjach obserwator będzie mógł ignorować niektóre z nich. Przykładem zastosowania mogą być tu dane statystyczne oraz ich reprezentacje w postaci np. arkusza danych, wykresu słupkowego, wykresu kołowego, itp. .Przy zastosowaniu wzorca Obserwator oddzielamy logiczną strukturę danych od jej reprezentacji, unikając trwałych połączeń między klasami i umożliwiając tym samym ponowne wykorzystanie poszczególnych klas. Jednocześnie unikamy nieczytelnego i zbyt mocno rozbudowanego kodu w pojedynczej klasie.

9) Odwiedzający (wizytator)

Cel:

Umożliwia zmianę operacji wykonywanych na elementach klasy bez zmiany struktury tej klasy. Zadaniem jest odseparowanie algorytmu od struktury obiektowej na której operuje. Praktycznym rezultatem tego odseparowania jest możliwość dodawania nowych operacji do aktualnych struktur obiektów bez konieczności ich modyfikacji.

Umożliwia wydzielenie zadań powiązanych logicznie ze sobą do osobnej klasy zwanej *odwiedzającym*. Odwiedzający może posiadać wiele implementacji zapewniając tym samym możliwość wykonania wielu różnych zadań.

Zalety:

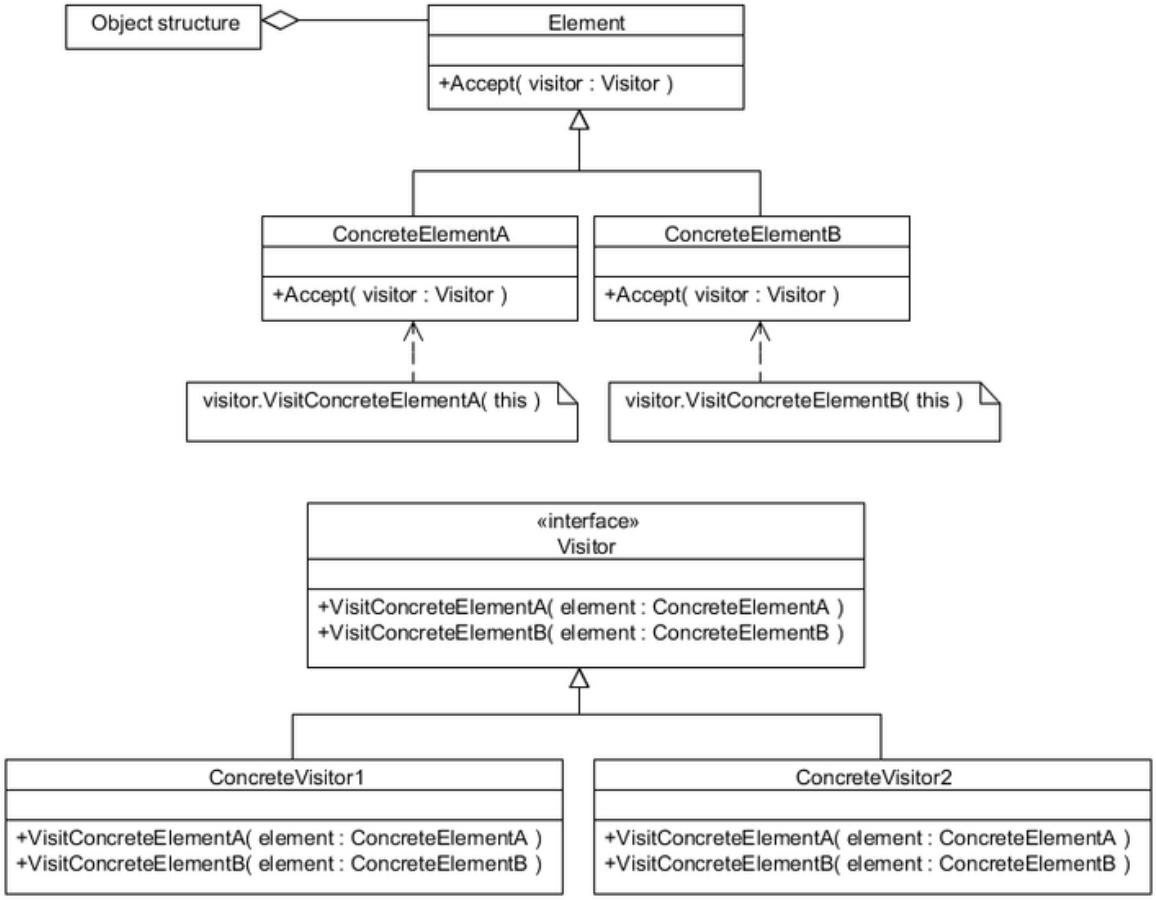
- Dodawanie nieplanowanych dotychczas operacji jest stosunkowo proste
- Klasy mogą być mniejsze, ponieważ rzadko wykonywane operacje można definiować w bytach zewnętrznych
- Obiekty wizytatorów mogą gromadzić informacje o stanie w czasie odwiedzania kolejnych elementów. Taki "mobilny agent" może odwiedzać zdalne obiekty (np. serwery baz danych) i budować złożony wynik na podstawie zawartości rozproszonej bazy danych
- Wymusza dodanie obsługi implementacji do wszystkich istniejących odwiedzających zapewniając tym samym obsługę elementu przez wszystkie zadania.
- Grupuje funkcjonalność logicznie powiązaną ze sobą w jedną klasę przez co dużo łatwiej jest ją rozwijać i utrzymywać.

Wady:

- Wewnętrzna struktura obiektu kompozytowego jest niekiedy ujawniana wizytatorowi, co z oczywistych względów narusza regułę hermetyzacji obiektów.
- Dodanie nowego elementu, który ma być obsługiwany przez wzorzec wymusza dodanie implementacji obsługi do wszystkich istniejących odwiedzających.
- Następuje rozhermetyzowanie obiektów zwanych elementami, bowiem każdy odwiedzający powinien mieć dostęp do składowych elementu.

Struktura

- Idea wzorca polega na stworzeniu interfejsu odwiedzającego (Visitor) zawierającego metody wirtualne Visit, dedykowane dla każdej z implementacji elementów (dziedziczących po klasie Element) w zadanej strukturze obiektów.
- Każdy odwiedzający jest "przyjmowany" przez dany element poprzez metodę Accept - dla poszczególnych implementacji obiektów Element, wołane są odpowiednie metody Visit w interfejsie Visitor odwiedzającego.
- Różne implementacje interfejsu Visitor mogą zawierać (hermetyzować) różne funkcjonalności dla całych struktur danych (składających się z obiektów Element). Obiekty tego typu reprezentują algorytmy wykonujące zadane czynności na każdym obiekcie osobno.
- Dla zbioru obiektów odwiedzanych, metoda Accept powinna być wywoływana w odpowiedniej kolejności, gwarantując, iż każdy element zostanie odwiedzony w odpowiednim momencie. Przykładowo, wizytator odwiedzający węzły w drzewie, powinien być akceptowany w kolejnych potomkach każdego z węzłów, zaś wizytator odwiedzający listę, może być wołany kolejno dla poszczególnych elementów.



Przykładowe zastosowanie

- Wzorzec wizytatora może być zastosowany przy implementacji drzewa wyprowadzenia w parserach lub kompilatorach. Niech analizator składniowy zwróci strukturę danych, będącą drzewem wyprowadzenia danego na wejście wyrażenia matematycznego. Drzewo to symbolizuje budowę semantyczną pewnej formuły matematycznej i składa się z dwóch rodzajów węzłów:
  - ArgumentNode - stała wartość numeryczna w wyrażeniu, np. "1.0".
  - OperatorNode - operator binarny w wyrażeniu; przyjmuje za argumenty prawy i lewy węzeł potomny, np. "1.0 + 2.0".
- Aby wykonać szereg operacji na całym drzewie, takich jak np. translacja wyrażenia do odwrotnej notacji polskiej lub kalkulacja wyrażenia, trzeba dla każdego z węzłów drzewa zaimplementować metody wykonujące powyższe zadania. Utrzymanie i stworzenie tak rozwiniętego kodu dla każdej z klas reprezentujących węzeł w drzewie może być dość skomplikowane. Aby uniknąć takich sytuacji, można wykorzystać wzorzec wizytatora.
- Niech obiekty będące węzłami w drzewie, będą zawierały metodę Accept, w treści której odwiedzający będzie przyjmowany kolejno w lewym i prawym dziecku oraz w samym węźle. Cała funkcjonalność wykonywana na drzewie, może być zaimplementowana w różnych wizytatorach (implementujących interfejs Visitor):
  - PostfixPrintVisitor - wizytator odpowiedzialny za przetłumaczenie wyrażenia matematycznego do postaci Odwrotnej Notacji Polskiej.

- CalculationVisitor - wizytator odpowiedzialny za obliczenie wyrażenia matematycznego. Wizytator używa stosu zawierającego wartości numeryczne; na stosie odkładane są obliczone wyniki dla każdego z węzłów reprezentującego operacje matematyczną.

10) Pamiątka (ang. Momento)

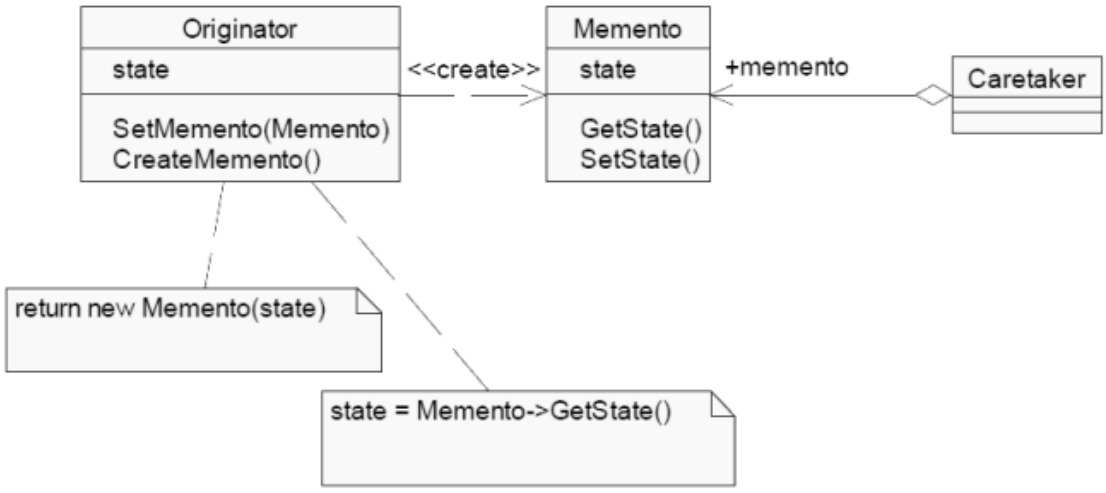
Cel:

- Umożliwienie zachowania stanu obiektu na zewnątrz w celu jego późniejszego odtworzenia
- Zachowanie hermetyzacji tego obiektu

Wzorzec memento pozwala na zapamiętanie, przechowywanie oraz odtworzenie stanu obiektu. Potrzeba zaimplementowania takiej funkcjonalności pojawia się dość często. Memento nie ma na celu zarządzania stanem ale tylko umożliwienie bezpiecznego dostępu do niego i równie bezpiecznego odtworzenia.

Wzorzec projektowy pamiątki w pierwotnej formie jest bardzo niewygodny w użyciu. Zazwyczaj wzorzec pamiątki implementuje się w postaci szablonu, pozbywając się jednocześnie potrzeby tworzenia struktury do przechowywania informacji.

Struktura



- ► Originator zapisuje i odtwarza swój stan w postaci obiektu Memento. Obiekt Caretaker przechowuje obiekty Memento, ale nie ma dostępu do ich danych
- ► Obiektem, którego stan należy przechować, jest Originator. Posiada on metody służące do utworzenia migawki stanu (createMemento()) oraz jej odczytania w celu przywrócenia wcześniejszego stanu (setMemento()). Obiekty-migawki stanu (Memento) przechowują stan obiektu Originator w postaci niezależnych instancji obiektu. Obiekty Memento posiadają metody getState() i setState(), służące do odczytania i zapisania stanu wewnątrz niego. Zarządzaniem kolejnymi migawkami stanu zajmuje się dedykowany obiekt Caretaker. Jednak istotą wzorca nie jest sama możliwość tworzenia migawek stanu, ale zapewnienie im właściwego poziomu bezpieczeństwa. Wzorzec Memento pozwala na dostęp do stanu zapisanego w migawce wyłącznie jego właścicielowi, czyli obiektowi

Originator, natomiast inne obiekty (w tym Caretaker) mogą tylko odwoływać się do całych obiektów, a metody setState() i getState() są dla nich niewidoczne.

Uczestnicy

- Memento
  - przechowuje zapisany stan obiektu Originator
  - uniemożliwia dostęp do tego stanu obiektowi Caretaker
- Originator
  - tworzy obiekt Memento ze swoim aktualnym stanem
  - odtwarza stan na podstawie obiektu Memento
- Caretaker
  - przechowuje obiekty Memento
  - nie ma dostępu do ich zawartości

Szczególną rolę we wzorcu odgrywają dwie klasy: Originator, który jest twórcą i właścicielem wszystkich migawek stanu, oraz Memento, której obiekty przechowują stan Originatora. Obiekt Originator musi posiadać możliwość utworzenia obiektu Memento oraz odczytania jego zawartości w celu przywrócenia na tej podstawie poprzedniego stanu. Memento przechowuje stan obiektu Originator zapisany w dowolnym momencie; pozwala też na dostęp do niego obiektowi Originator, natomiast uniemożliwia operacje na migawce wszelkim innym obiektom. Przykładem jest obiekt Caretaker, który zarządza utworzonymi migawkami, natomiast nie ma dostępu do ich zawartości.

Konsekwencje

- Zachowanie hermetyzacji obiektu Memento
- Uproszczenie obiektu Originator
  - odpowiedzialność za zapis stanu przeniesiona na Memento
- Podwójny interfejs obiektu Memento
  - wąski: dla obiektu Caretaker
  - szeroki: dla obiektu Originator
- Potencjalny wzrost złożoności pamięciowej
  - stan może być obszerny
  - Caretaker nie zna tego rozmiaru i nie może optymalizować sposobu zarządzania nim

11) Polecenie (komenda, ang. Command)

Cel

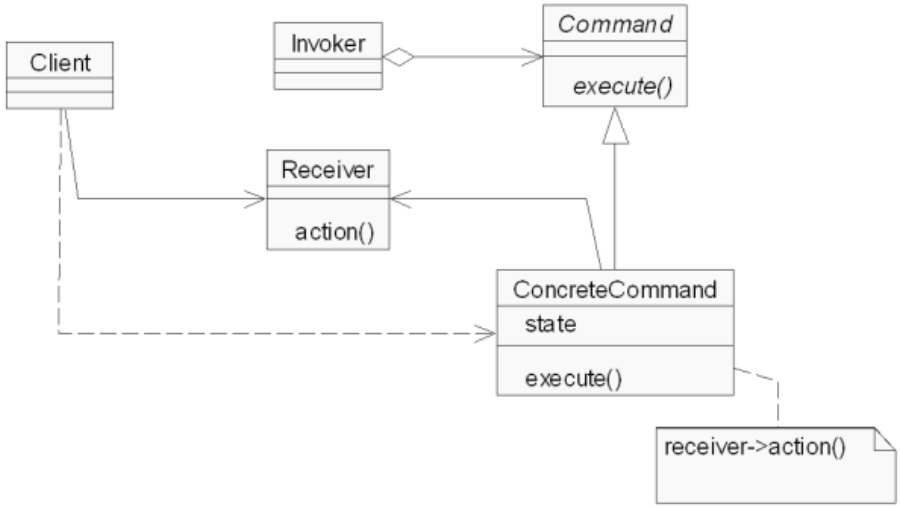
- Hermetyzacja poleceń do wykonania w postaci obiektów, umożliwienie parametryzacji klientów obiektami poleceń, wsparcie dla poleceń odwracalnych
- Polega na reprezentowaniu czynności do wykonania przez specjalne obiekty polecenia enkapsulujące czynność do wykonania i jej parametry.
- Na przykład biblioteka do obsługi drukarki może zawierać klasę PolecenieDrukowania. Użytkownik biblioteki tworzy nowy obiekt klasy PolecenieDrukowania, ustawia jego parametry (zawartość dokumentu, ilość kopii itp.) i na końcu wywołuje metodę przekazującą zadanie do drukarki.
- W tym wypadku wystarczyłoby jedynie zrobić pojedynczą funkcję WyślijPolecenieDoDrukarki(), która przyjmowałaby odpowiednie parametry. Zalety tworzenia osobnych klas dla poleceń to:
  - Możliwe uproszczenie API. Czasami kod używający obiektów poleceń jest krótszy i bardziej przejrzysty niż kod wywołujący procedurę z wieloma parametrami, zwłaszcza gdy zwykle ustawia się tylko kilka parametrów a resztę zostawia w domyślnym stanie.
  - Obiekt polecenia zapewnia wygodne miejsce do tymczasowego przechowywania parametrów polecenia.
  - Klasa umożliwia wygodne zgrupowanie całego kodu i danych dotyczących danego polecenia. Obiekt polecenia może przechowywać informacje o poleceniu, jak na przykład jego nazwę, identyfikator użytkownika zlecającego polecenia, może udzielać dodatkowych informacji, jak na przykład przewidywany czas zakończenia.
  - Traktowanie poleceń jako obiekty pozwala na tworzenie struktur danych przechowujących wiele poleceń. Skomplikowane polecenie może być traktowane jako drzewo lub graf obiektów poleceń.
  - Traktowanie poleceń jako obiekty pozwala na dowolne cofanie działania poleceń (undo), jeżeli tylko obiekty po ich wykonaniu się gdzieś zachowa (np. na stosie) oraz polecenie potrafi przywrócić stan sprzed jego wykonania.

Zastosowania wzorca polecenia

- Wielopoziomowe cofanie zmian. Jeżeli wszystkie działania użytkownika programu są zapisane jako obiekty poleceń, program może przechowywać na stosie wszystkie ostatnio wykonane działania. Jeżeli użytkownik chce cofnąć polecenie, program po prostu ściąga ostatnie polecenie ze stosu i uruchamia jego metodę cofnij().
- Transakcje. Potrzebne jest też czasem automatyczne cofanie zmian jeżeli działanie operacji przerwie się w połowie, na przykład w instalatorach czy bazach danych.
- Paski postępu. Jeżeli program wykonuje sekwencję poleceń i każde z nich ma metodę oszacujCzasDoKońca(), można łatwo oszacować całkowity czas wszystkich operacji i pokazać dokładny pasek postępu.
- Kreatory. Kreatory często pokazują wiele stron konfiguracji dla pojedynczej akcji, która jest realizowana dopiero, gdy użytkownik klika przycisk "Zakończ" na ostatniej stronie. Można zapisać akcję kreatora jako obiekt polecenia. Jest on tworzony, gdy okno kreatora jest po raz pierwszy wyświetlane. Każdy krok kreatora ustawia parametry w obiekcie polecenia. Przycisk "Zakończ" jedynie wywołuję metodę wykonaj(). W ten sposób klasa polecenia nie zawiera żadnego kodu interfejsu użytkownika.

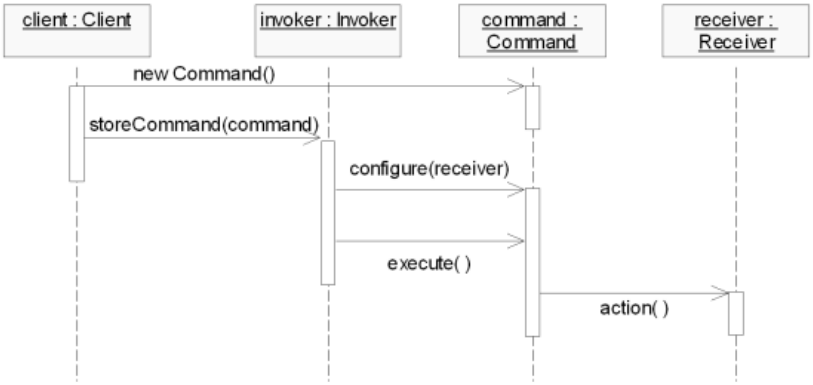
- Pule wątków. Typowa klasa puli wątków może mieć publiczną metodę dodajZadanie(), dodającą obiekty poleceń do wewnętrznej kolejki zadań do wykonania, a następnie ograniczona liczba wątków wykonuje po kolei te zadania. Te obiekty poleceń zwykle wypełniają wspólny interfejs jak na przykład java.lang.Runnable, który pozwala puli wątków wykonywać zadania bez jakiejkolwiek wiedzy o ich działaniu.
- Nagrywanie makr. Jeżeli wszystkie działania użytkownika są reprezentowane przez obiekty poleceń, program może zapisać sekwencję akcji jako listę obiektów poleceń w miarę jak są wykonywane. Można później "odegrać" te same akcje, wykonując z powrotem po kolei te obiekty poleceń. Jeżeli program ma wbudowany silnik skryptów, każdy obiekt może implementować metodę doSkryptu() i akcje użytkowników mogą być w łatwy sposób zapisane jako skrypty.
- Sieć. Można przesyłać przez sieć całe obiekty poleceń i wykonywać je na innych komputerach, na przykład akcje wszystkich graczy w grach komputerowych.

Struktura



Podstawowym elementem wzorca jest interfejs Command, deklarujący metodę execute(). Jest to polimorficzna metoda reprezentująca polecenie do wykonania. Metoda ta jest implementowana w klasach ConcreteCommand w postaci polecenia wykonania określonej akcji na obiekcie-przedmiocie Receiver. Klient nie jest bezpośrednio związany ani z obiektem Command, ani z obiektem inicjującym jego wywołanie, czyli Invoker. Widzi jedynie odbiorcę wyników operacji - obiekt Receiver.

Interakcje



Szczegółowy przepływ sterowania przedstawia diagram sekwencji. Inicjatorem przetwarzania jest obiekt Invoker, który zarządza obiektami typu Command. W momencie nadejścia żądania wykonania określonej operacji Invoker parametryzuje skojarzony z nią obiekt Command właściwym odbiorcą ich działań, czyli obiektem Receiver. Następnie wywołuje metodę execute() w tym obiekcie, powodując określone skutki w obiekcie Receiver, widoczne dla Klienta.

Uczestnicy

- Command - definiuje interfejs obiektu reprezentującego polecenie
- Concrete Command - jest powiązany z właściwym obiektem Receiver, implementuje akcję w postaci metody execute()
- Client - tworzy Concrete Command
- Invoker - ustala odbiorcę akcji każdego obiektu Command, wywołuje metodę execute()
- Receiver - jest przedmiotem akcji wykonanej przez Command

W aplikacji okienkowej polecenia znajdujące się w menu są zdefiniowane w postaci obiektów typu Command. Każde polecenie jest inną implementacją tego interfejsu, i posiada innego odbiorcę, ustalanego w momencie wykonywania akcji (np. polecenie zamknięcia okna działa na aktualnie aktywne okno). W momencie kliknięcia na wybranej pozycji menu (czyli obiektu Invoker), wykonuje ona metodę execute() skojarzonego z nią polecenia typu Command, ustalając jego odbiorcę. Efekt, w postaci np. zamknięcia okna, jest widoczny dla klienta.

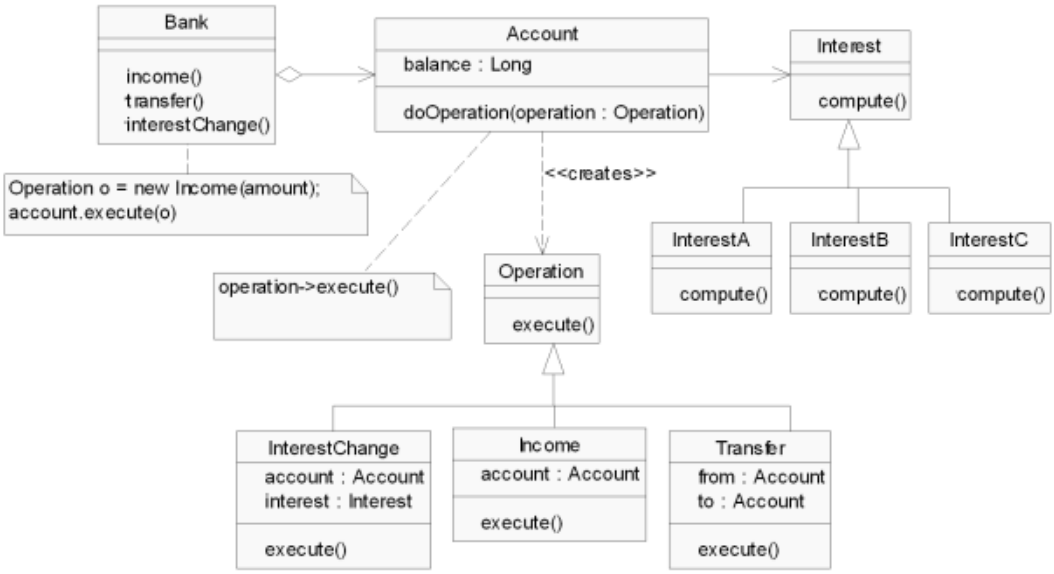
Konsekwencje

- Usunięcie powiązania między nadawcą i przedmiotem polecenia
- Łatwe dodawanie kolejnych obiektów Command
- Możliwość manipulacji obiektami Command - polecenia złożone: wzorzec Composite
- Polecenia mogą być odwracalne - zapamiętanie stanu przez Concrete Command, wykorzystanie wzorca Memento

Istotną korzyścią płynącą z zastosowania wzorca jest rozdzielenie zależności pomiędzy nadawcą (Klientem) i odbiorcą (obiektem Receiver) komunikatu. Zastosowanie polimorfizmu pozwala traktować poszczególne polecenia abstrakcyjnie, a co za tym idzie - dodawać nowe typy poleceń bez konieczności zmiany struktury

systemu. Poszczególne obiekty Command mogą być dowolnie złożone, także w postaci kompozytów innych poleceń. Dodatkową zaletą użycia obiektu do hermetyzacji poleceń jest możliwość utworzenia w typie Command przeciwstawnej metody, która odwraca efekt wykonania polecenia. W takiej sytuacji obiekt ConcreteCommand musi zapamiętać stan obiektu Receiver sprzed wykonania operacji lub np. skorzystać z wzorca Memento.

Przykład



Bank zarządza grupą obiektów Account reprezentujących rachunki bankowe. Operacje bankowe, wykonywane na rachunkach, są implementacjami interfejsu Operation, posiadającego metodę execute(). Jej implementacja zależy od rodzaju operacji, dlatego w przypadku obiektu InterestChange będzie ona zmieniała stopę procentową, a w przypadku obiektu Transfer - dokonywała przelewu. Ponieważ każda operacja wymaga innych parametrów, dlatego są one przekazywane w konstruktorze poszczególnej klasy, a nie bezpośrednio w metodzie execute(). W tym przykładzie rolę obiektu Invoker pełni bank, ponieważ on wykonuje metodę execute(), a rolę przedmiotu polecenia (obiektu Receiver) - obiekt Account.