Wzorce projektowe w technologii .NET

Paweł Biesiada
https://www.linkedin.com/in/pawelbiesiada/

- Architekt, software developer, ex. Scrum Master, IT Trainer
- .NET developer since 11.2011
- IT Trainer since 06.2018
- *pawel.piotr.biesiada@gmail.com*

Plan zajęć

- SOLID
- Wzorce interfejsów
  - adapter, facade, composite, bridge
- Wzorce konstrukcyjne
  - builder, factory method, abstract factory, prototype, memento
- Wzorce odpowiedzialności
  - singleton, observer, mediator, proxy, chain-of-responsibility, flyweight pattern
- Wzorce operacji
  - template method, state, strategy, command, interpreter
- Wzorce rozszerzeń
  - Decorator, iterator, visitor

SOLID

- S – Single responsibility principle
- O – Open/Closed principle
- L – Liskov substitution principle
- I – Interface segregation principle
- D – dependency inversion principle

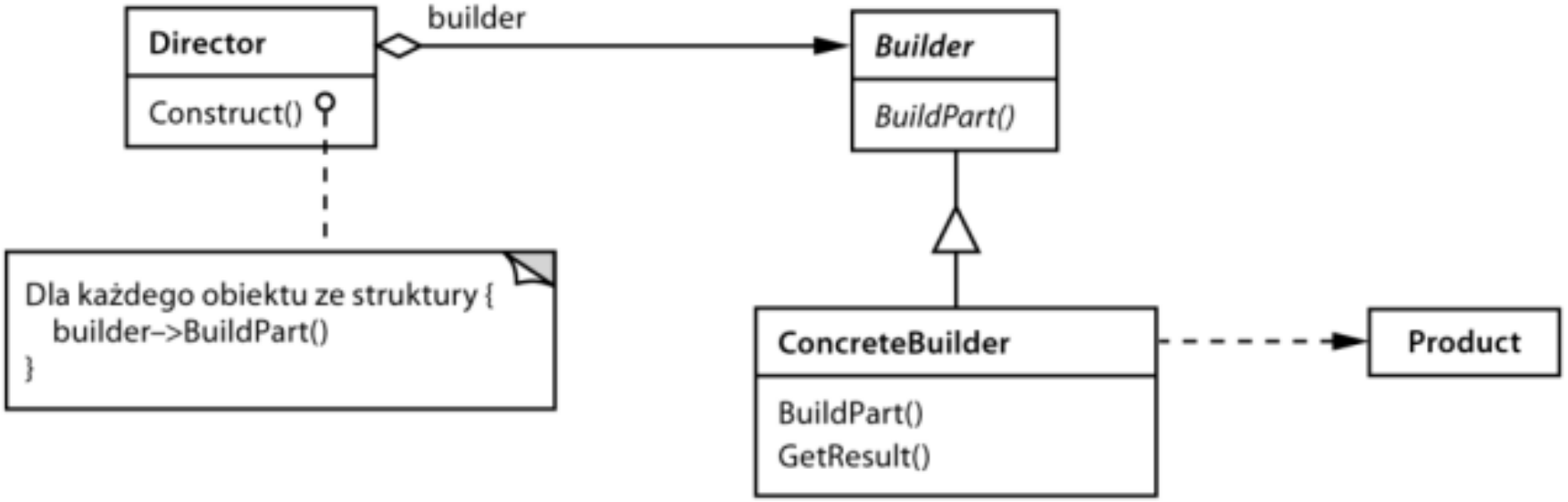
Wzorce projektowe

- Kreacyjne (Konstrukcyjne)
  - Singleton, Abstract factory, Builder
- Strukturalne
  - Facade, Proxy
- Operacyjne (Czynnościowe)
  - Observer, State, Strategy, Command, Iterator

Wzorce projektowe

- Typowe rozwiązania typowych problemów
- Standardowe słownictwo projektowe
- Pomoc w dokumentowaniu i rozumieniu kodu

Builder - Budowniczy (Kreacyjny)



- Oddziela tworzenie złożonego obiektu od jego reprezentacji, dzięki czemu ten sam proces kontrukcji może prowadzić do powstawania różnych reprezentacji

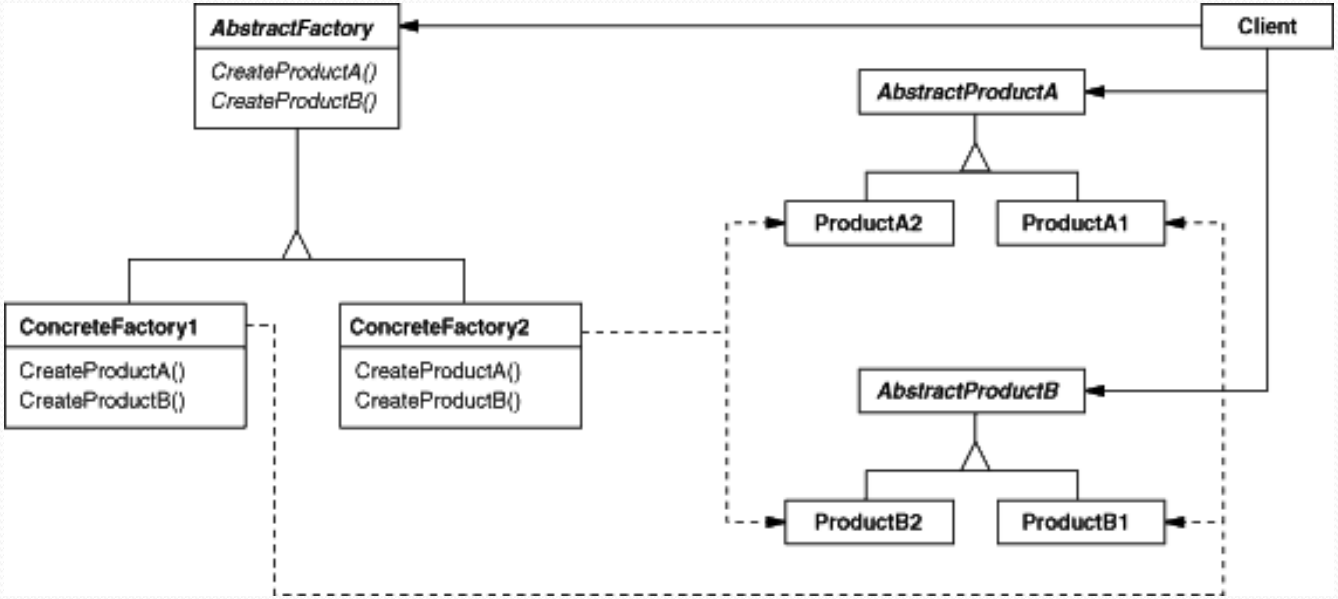
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Builder - Budowniczy (Kreacyjny)

- **Stosowanie**
  - Jeśli algorytm tworzenia obiektu złożonego powinien być niezależny od składników tego obiektu i sposobu ich łączenia
  - Kiedy proces konstrukcji musi umożliwiać tworzenie różnych reprezentacji generowanego obiektu

- **Konsekwencje**
  - Możliwość modyfikowania reprezentacji obiektu
  - Odizolowanie reprezentacji od kodu tworzącego produkt
  - Większa kontrola nad procesem tworzenia

Abstract Factory – Fabryka abstrakcyjna (Kreacyjny)



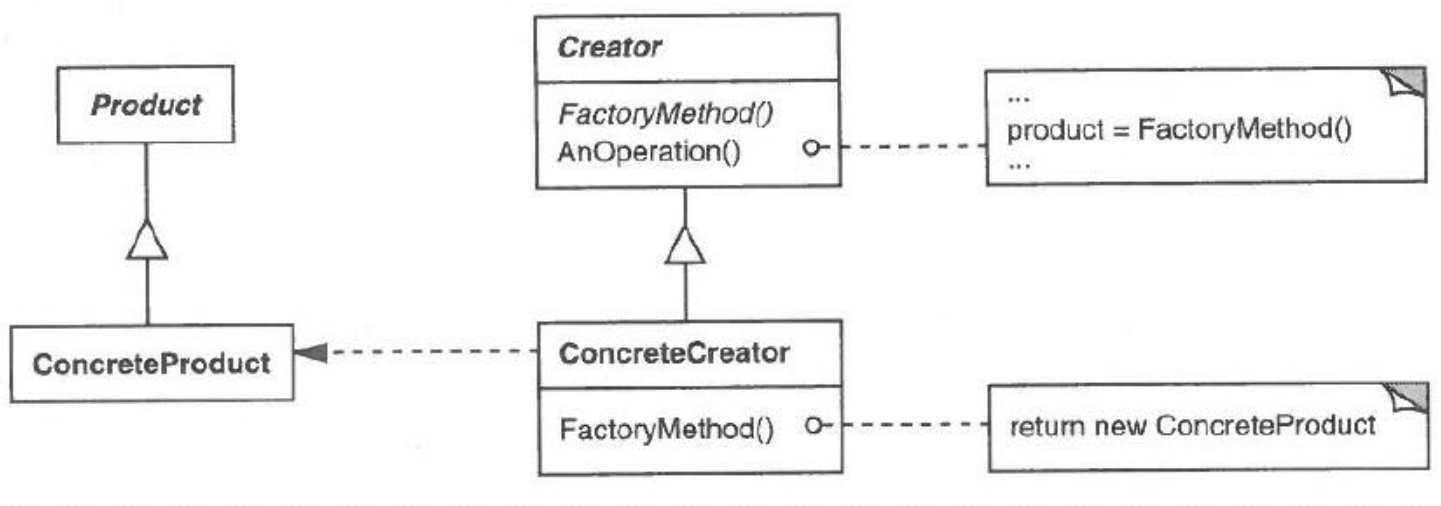
- Udostępnia interfejs do tworzenia pewnego podzbioru podobnych obiektów
- To konkretna implementacja Fabryki dostarcza konkretne implementacje obiektów

Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Abstract Factory – Fabryka abstrakcyjna (Kreacyjny)

- **Stosowanie**
  - Jeśli system należy skonfigurować za pomocą jednej z wielu rodzin produktów
  - Jeśli powiązane obiekty z jednej rodziny są zaprojektowane do wspólnego użytku i należy z nich korzystać jednocześnie
- **Konsekwencje**
  - Izoluje klasy konkretne
  - Ułatwia zastępowanie rodzin produktów
  - Ułatwia zachowanie spójności między produktami
  - Utrudnia dodawanie obsługi nowego rodzaju produktów

Factory Method – metoda wytwórcza (Kreacyjny)



- Określa interfejs do tworzenia obiektów
- To konkretna implementacja metody wytwórczej decyduje, którą implementację interfejsu zwrócić

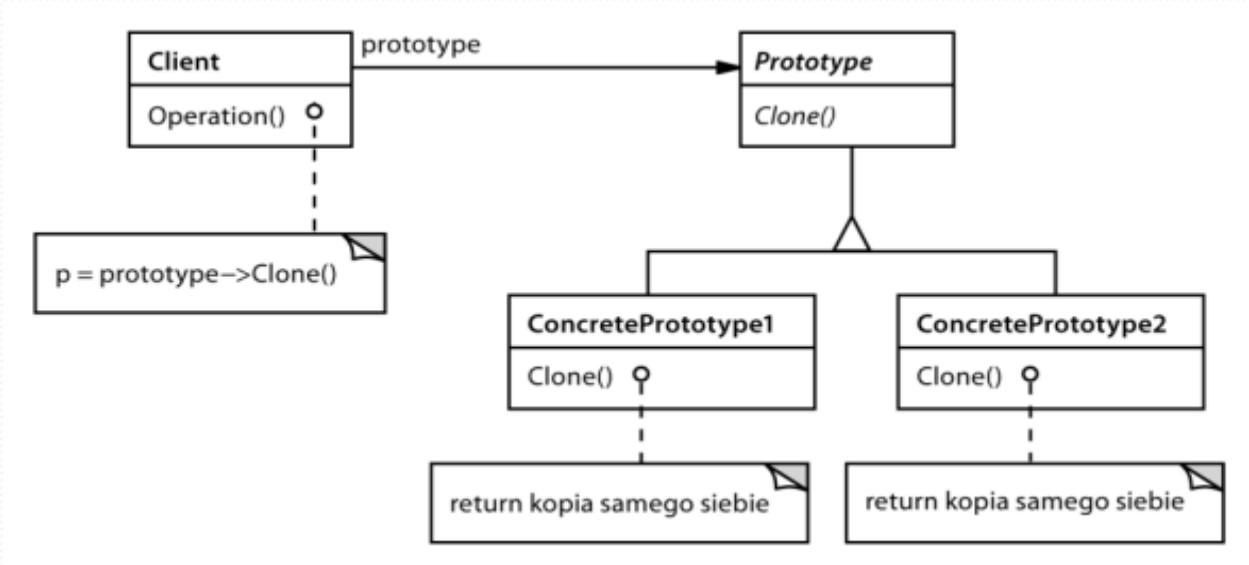
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Factory Method – metoda wytwórcza (Kreacyjny)

- **Stosowanie**
  - Kiedy nie można, lub nie chcemy z góry ustalić klasy obiektów, które trzeba utworzyć
  - Jeśli to programista chce, aby to podklasy danej klasy określały tworzone przez nią obiekty

- **Konsekwencje**
  - Połączenie równoległych hierarchii klas
  - Zapewnienie punktów zaczepienia dla podklas – pozwala na elastyczne tworzenie obiektów niż bezpośrednio poprzez konstruktor

Prototype – Prototyp (Kreacyjny)



- Określa na podstawie prototypowanego egzemplarza rodzaje tworzonych obiektów i generuje nowe obiekty przez kopiowanie tego prototypu

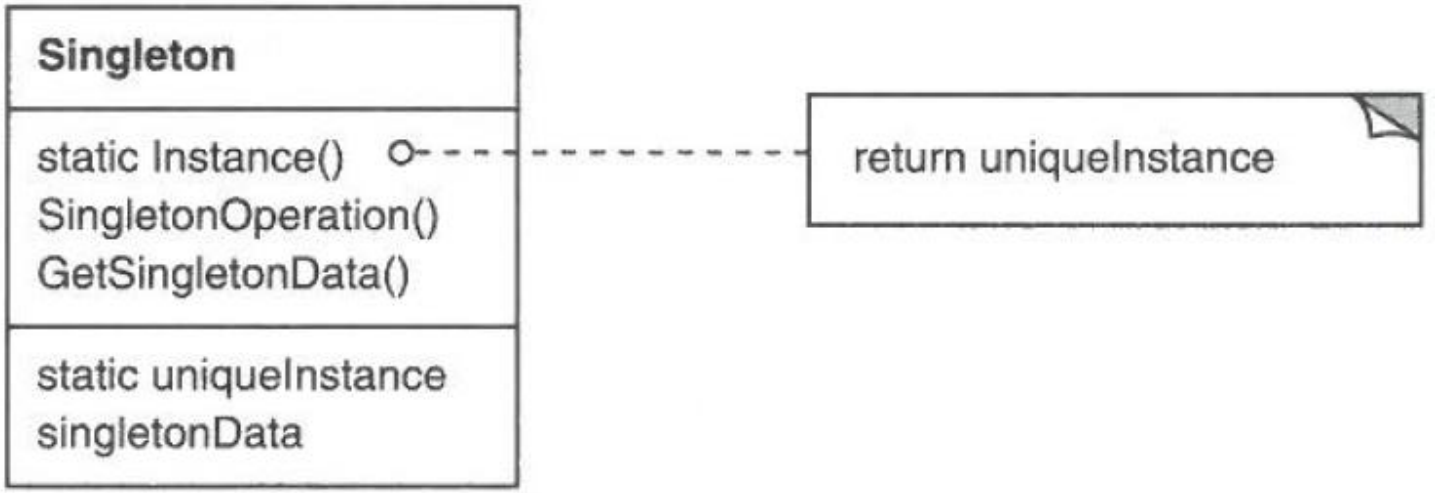
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Prototype – Prototyp (Kreacyjny)

- **Stosowanie**
  - Jeśli system powinien być niezależny od sposobu tworzenia, składania i reprezentowania produktów
  - Kiedy klasy tworzonych obiektów są określany w czasie wykonywania programu

- **Konsekwencje**
  - Możliwość dodawania i usuwania produktów w czasie wykonywania programu
  - Zmniejszenie liczby podklas poprzez spłaszczenie hierarchii
  - Możliwość dynamicznego konfigurowania aplikacji za pomocą klas (używając menadżera prototypów)

Singleton - Singleton (Kreacyjny)



- Gwarantuje istnienie tylko jednej instancja klasy i zapewnia do niej globalny dostęp

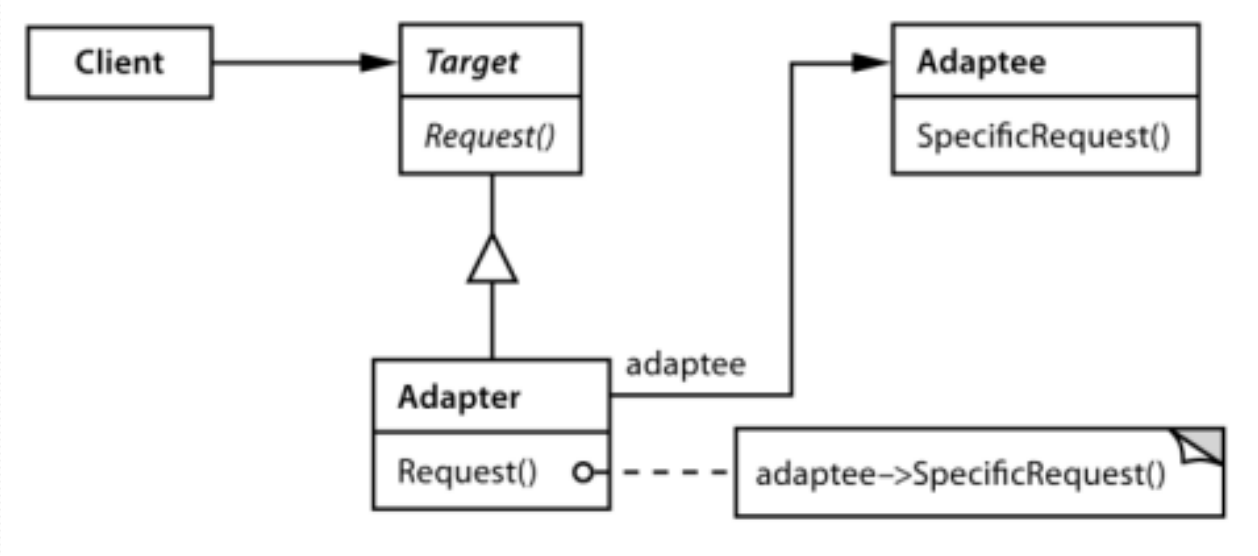
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Singleton - singleton (Kreacyjny)

- **Stosowanie**
  - Jeśli powinien istnieć dokładnie jeden egzemplarz klasy
  - Kiedy potrzebna jest możliwość rozszerzania jedynego egzemplarza przez tworzenie podklas

- **Konsekwencje**
  - Zapewnia kontrolę dostępu do jedynego egzemplarza
  - Umożliwia dopracowywanie operacji i reprezentacji
  - Umożliwia określenie dowolnego limitu liczby egzemplarzy (więcej niż domyślnie jednego)
  - Jest bardziej elastyczny od operacji statycznych

Adapter – adapter (Strukturalny)



- Przekształca interfejs klasy na inny, oczekiwany przez klienta
- Umożliwia współdziałanie klasom, które ze względu na interfejs nie mogą działać razem

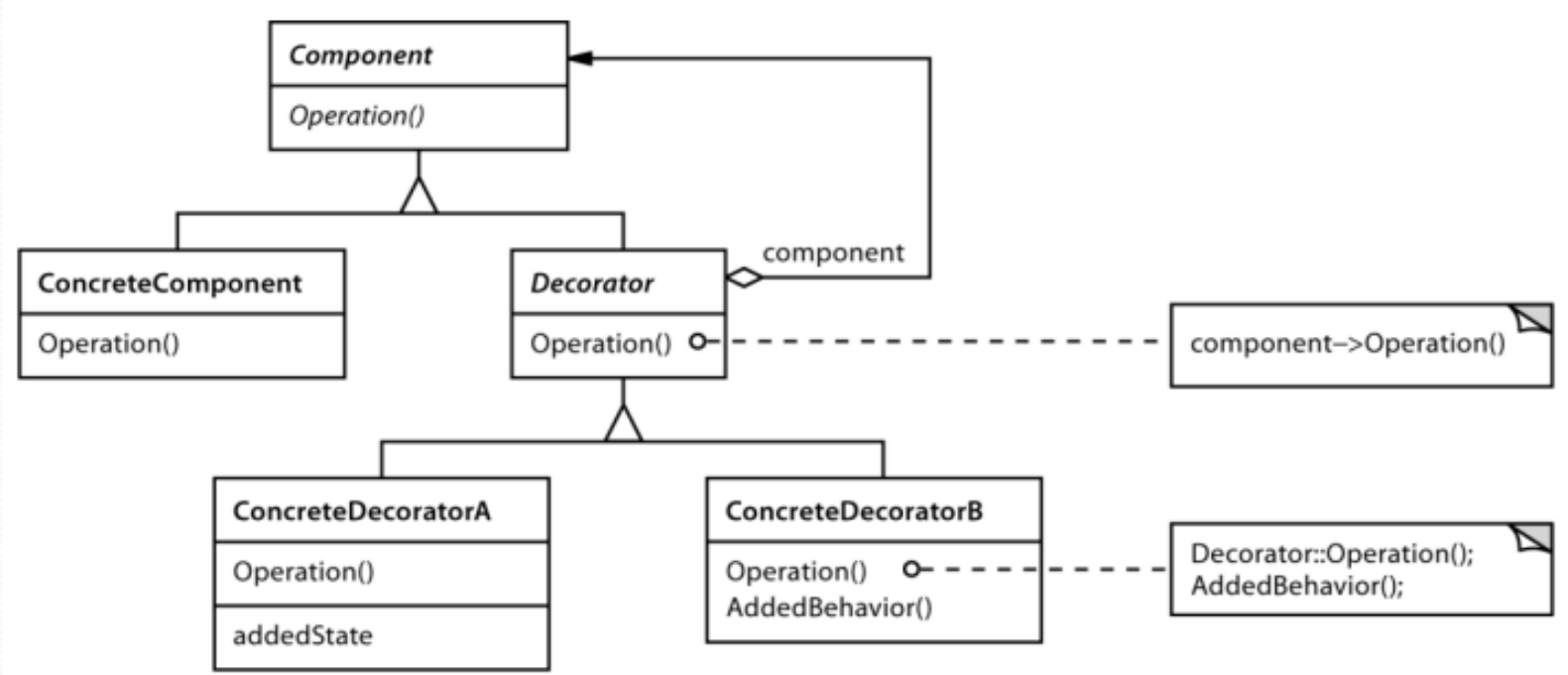
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Adapter – adapter (Strukturalny)

- **Stosowanie**
  - Jeśli chcesz wykorzystać istniejącą klasę, ale jej interfejs nie pasuje do tego, który jest Tobie potrzebny
  - Kiedy chcesz utworzyć klasę do wielokrotnego użytku współdziałającą z niepowiązanymi lub nieznanymi klasami

- **Konsekwencje**
  - Umożliwia współdziałanie jednej klasy *Adapter* z wieloma klasami *Adaptee*
  - Utrudnia przesłanianie zachowań z klasy Adaptee

Decorator – Dekorator (Strukturalny)



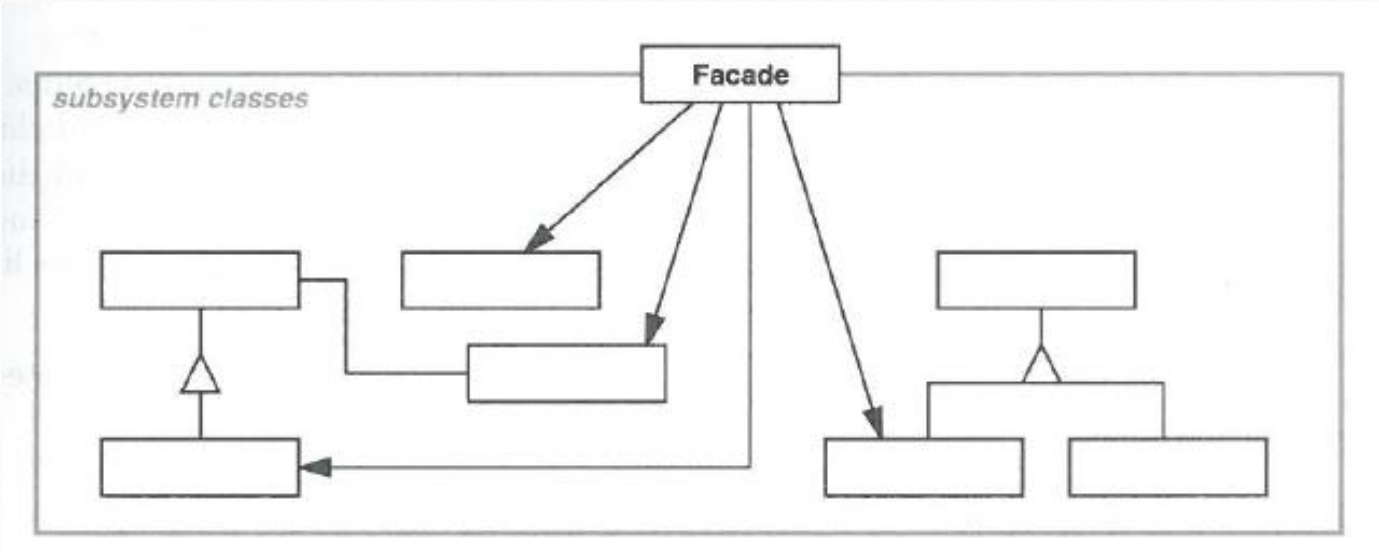
- Dynamicznie dołącza dodatkowe obowiązki obiektu.
- Wzorzec ten udostępnia alternatywny, elastyczny sposób tworzenia podklas o wzbogaconych funkcjach

Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Decorator – Dekorator (Strukturalny)

- **Stosowanie**
  - Kiedy trzeba dodawać zadania do poszczególnych obiektów w dynamiczny sposób bez wpływu na inne obiekty
  - Jeśli potrzebny jest mechanizm do obsługi zadań, które można cofnąć
  - Jeśli rozszerzanie przez tworzenie podklas jest niepraktyczne
- **Konsekwencje**
  - Zapewnia większą elastyczność niż dziedziczenie
  - Pozwala uniknąć budowania nadmiernej hierarchii klas
  - Dekorator i powiązany z nim komponent nie są identyczne
  - Powstawanie wielu małych obiektów

Facade - Fasada (Strukturalny)



- Fasada tworzy jednolity interfejs dla zbioru interfejsów z podsystemu do komunikacji ze środowiskiem zewnętrznym
- Nowy interfejs ułatwia korzystanie z podsystemów

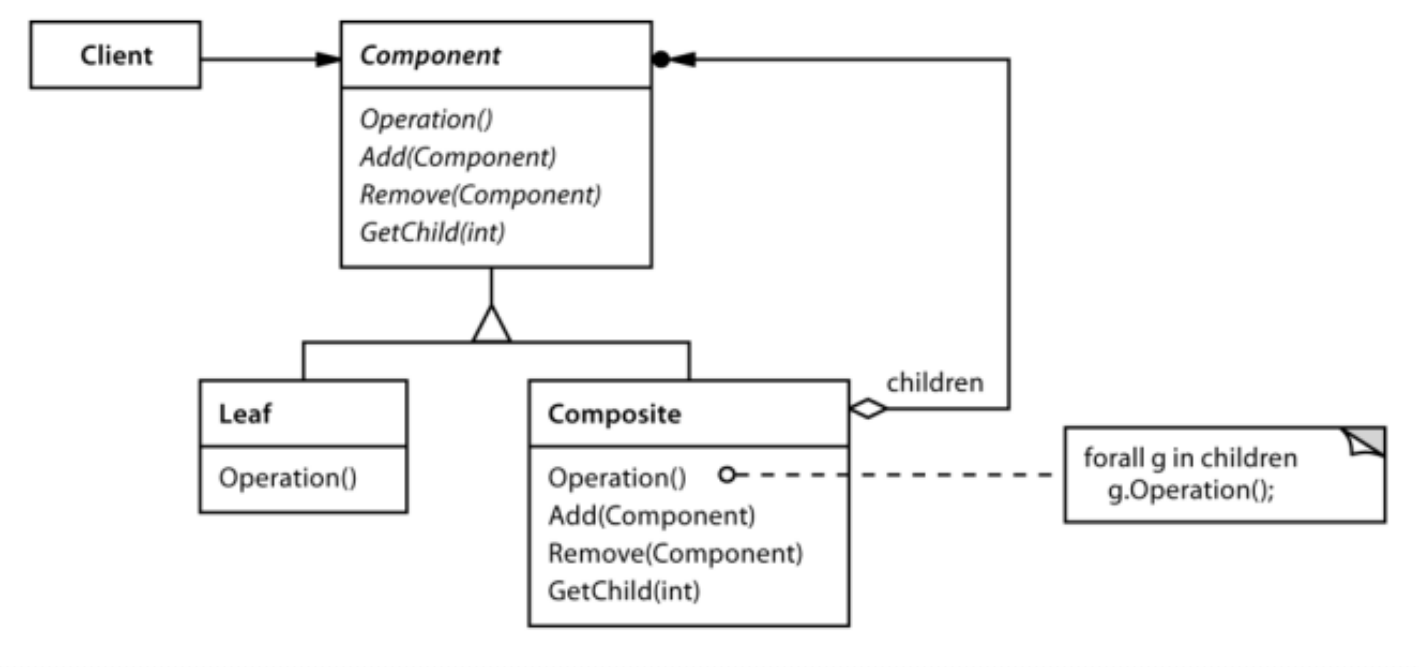
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Facade - Fasada (Strukturalny)

- **Stosowanie**
  - Programista chce udostępnić prosty interfejs do złożonego podsystemu
  - Występowanie wielu zależności między klientami i klasami z implementacją abstrakcji

- **Konsekwencje**
  - Oddziela klientów od komponentów podsystemów
  - Pomaga zachować luźne powiązanie pomiędzy podsystemem, a klientami
  - Nie uniemożliwia aplikacjom korzystania z klas podsystemów

Composite – Kompozyt (Strukturalny)



- Składa obiekty w struktury drzewiaste odzwierciedlające hierarchię typu część-całość.
- Umożliwia klientom traktowanie poszczególnych obiektów w taki sam sposób

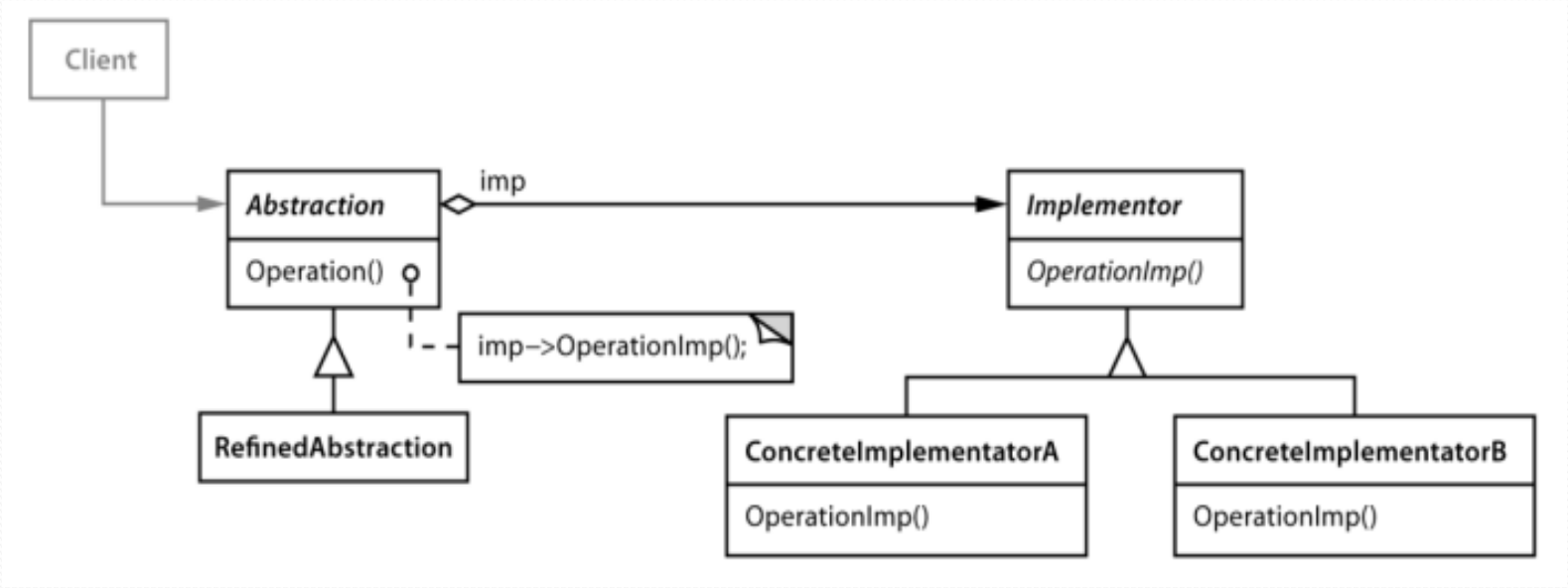
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Composite – Kompozyt (Strukturalny)

- **Stosowanie**
  - Jeżeli chcesz przedstawić hierarchię obiektów typu część-całość
  - Kiedy chcesz, aby w klientach można było zignorować różnicę pomiędzy złożeniami obiektów i pojedynczymi obiektami

- **Konsekwencje**
  - Umożliwia definiowanie hierarchii składających się obiektów prostych i złożonych
  - Upraszcza kod klientów
  - Ułatwia dodawanie komponentów nowego rodzaju
  - Może sprawić, że projekt stanie się zanadto ogólny

Bridge - Most (Strukturalny)



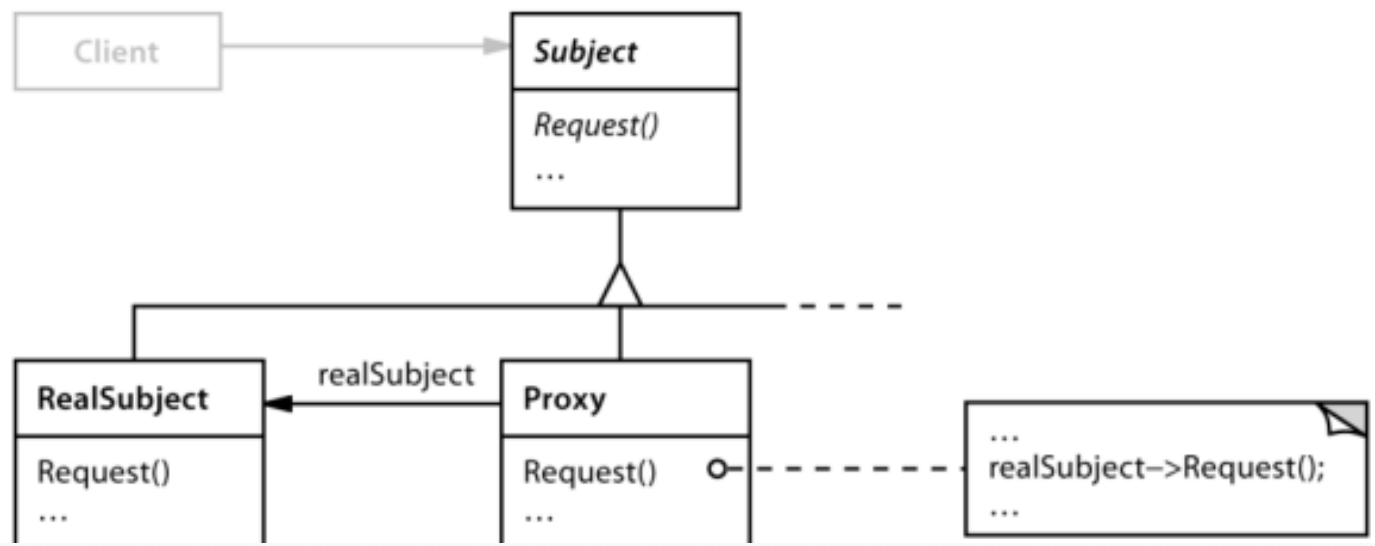
- Oddziela abstrakcję od jej implementacji, dzięki czemu można modyfikować te dwa elementy niezależnie od siebie

Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Bridge - Most (Strukturalny)

- **Stosowanie**
  - Jeśli chcesz uniknąć trwałego powiązania abstrakcji i jej implementacji.
  - Kiedy rozszerzanie przez tworzenie podklas powinno być możliwe zarówna dla abstrakcji jaki i dal implementacji
  - Jeżeli zmiany w implementacji abstrakcji nie powinny mieć wpływu na klientów
- **Konsekwencje**
  - Oddzielenie interfejsu od implementacji
  - Łatwiejsze rozszerzanie
  - Ukrycie szczegółów implementacji przed klientami

Proxy – Pełnomocnik (Strukturalny)



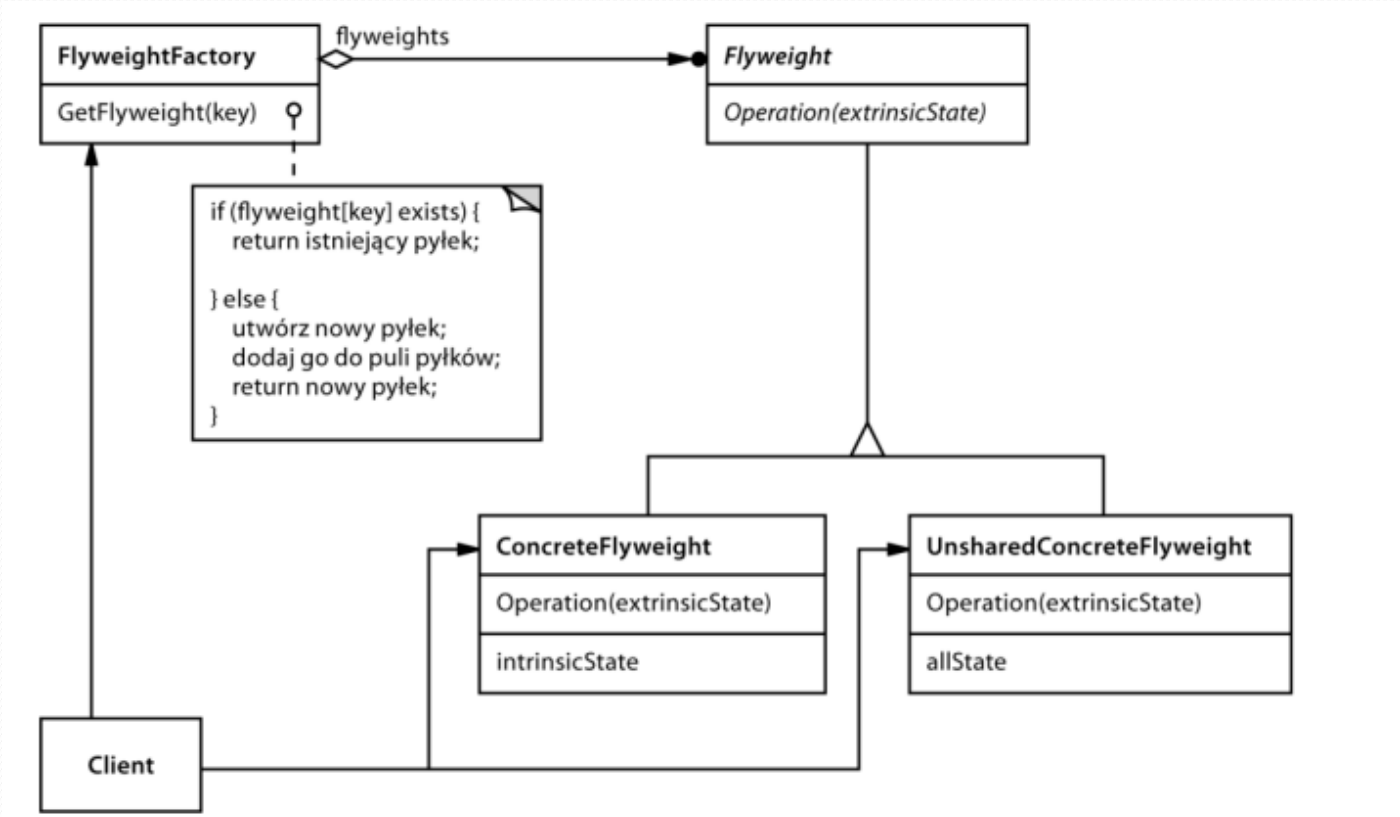
- Udostępnia zastępnik, lub reprezentanta innego obiektu w celu kontrolowania dostępu do niego

Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Proxy – Pełnomocnik (Strukturalny)

- **Stosowanie**
  - Kiedy chcesz oddelegować zarządzanie obiektem do innego obiektu
  - Kiedy chcesz oddelegować czas inicjalizacji docelowego obiektu (np. lazy loading)
  - Kiedy chcesz ograniczyć kontrolę do docelowego obiektu
- **Konsekwencje**
  - Wprowadza nowy poziom pośredniości przy dostępie do obiektu
  - Może ukrywać fakt, że obiekt znajduje się w innej przestrzeni adresowej
  - Umożliwia wykonywanie dodatkowych operacji przy dostępie do obiektu

Flyweight – Pyłek (Strukturalny)



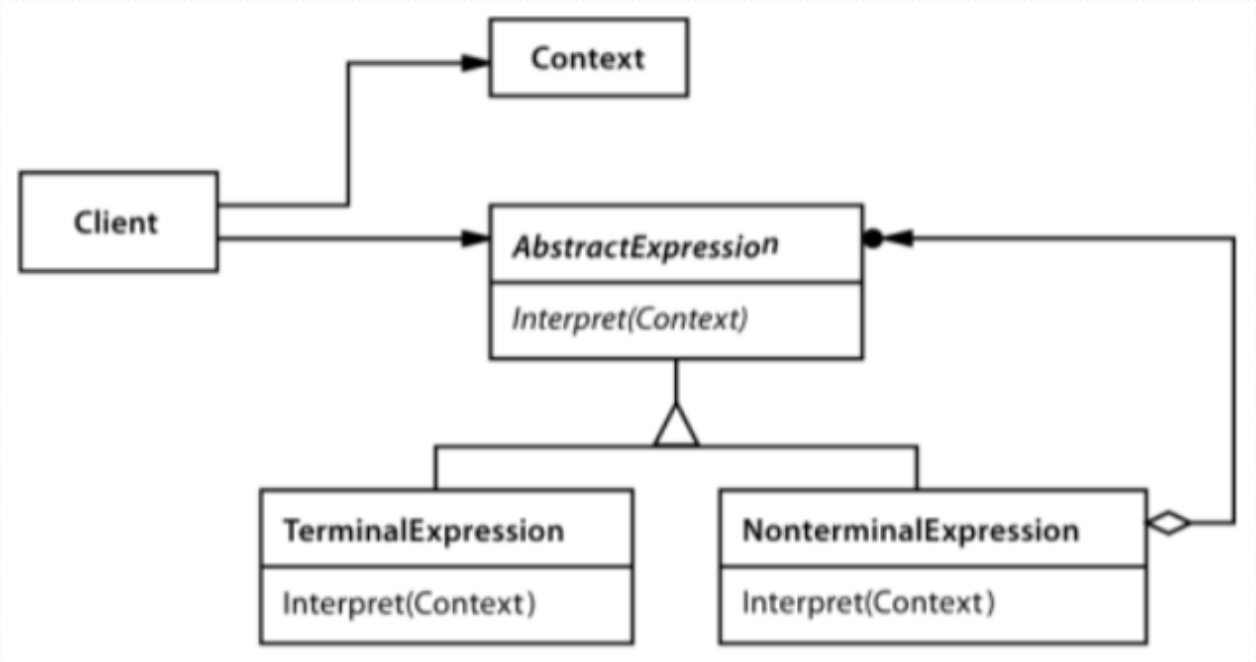
- Wykorzystuje współdzielenie do wydajnej obsługi dużej liczby małych obiektów

Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Flyweight – Pyłek (Strukturalny)

- **Stosowanie**
  - Kiedy aplikacja korzysta z dużej liczby obiektów
  - Koszty przechowywania obiektów są wysokie
  - Większość stanu obiektów można zapisać poza nimi
  - Po przeniesieniu stanu na zewnątrz wiele grup obiektów można zastąpić stosunkowo nielicznymi obiektami
  - Aplikacja nie jest zależna od tożsamości obiektów
- **Konsekwencje**
  - Zmniejsza koszty przechowywania obiektów
  - Zwiększa koszty związane z przenoszeniem, wyszukiwaniem i obliczaniem stanu (stan można zapisać) obiektów

Interpreter – Interpreter (Operacyjny)



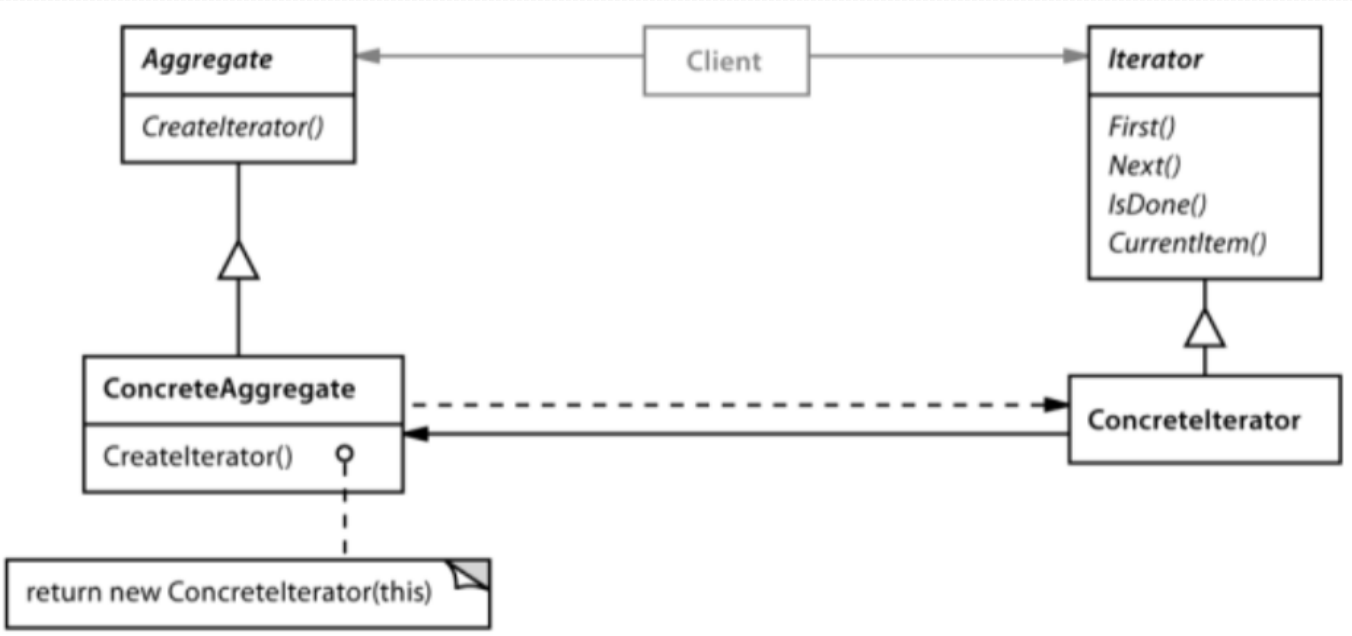
- Określa reprezentację gramatyki języka oraz interpreter, który wykorzystuje tę reprezentację do interpretowania zdań z danego języka

Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Interpreter – Interpreter (Operacyjny)

- **Stosowanie**
  - Jeśli istnieje interpretowany język, a zdania tego języka można przedstawić za pomocą drzewa składni abstrakcyjnej
  - Gramatyka jest prosta (w przeciwnym wypadku lepszy jest parser)
- **Konsekwencje**
  - Implementowanie, modyfikowanie i rozszerzanie gramatyki jest łatwe
  - Konserwowanie złożonych gramatyk jest trudne

Iterator – Iterator (Operacyjny)



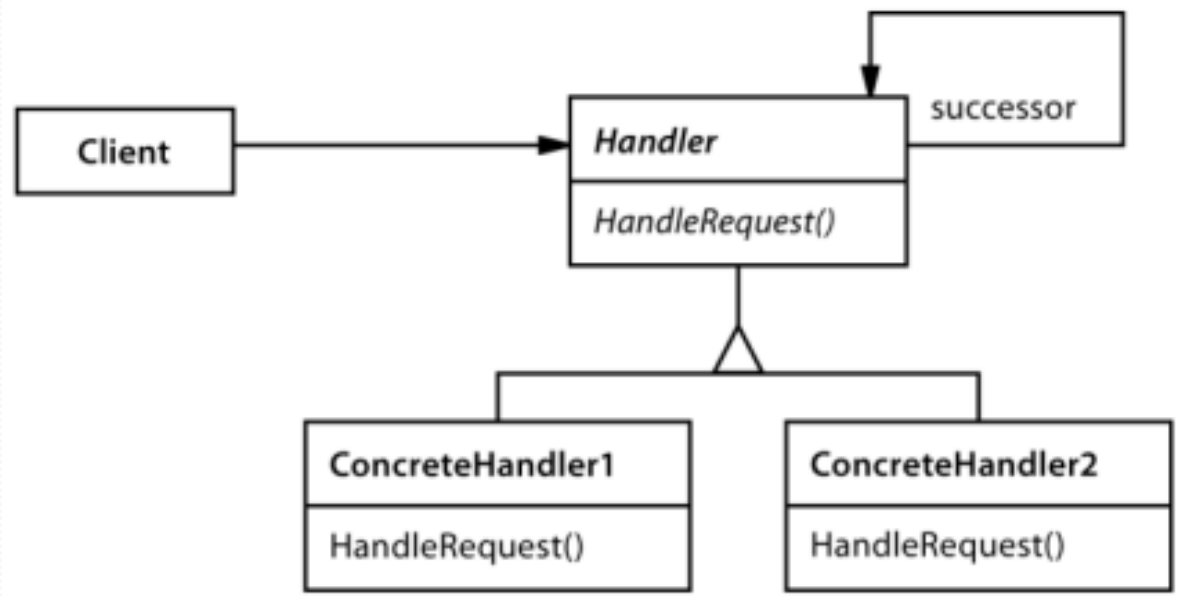
- Zapewnia sekwencyjny dostęp do elementów obiektu złożonego bez ujawniania jego wewnętrznej reprezentacji

Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Iterator – Iterator (Operacyjny)

- **Stosowanie**
  - Kiedy chcesz uzyskać dostęp do zawartości obiektu zagregowanego bez ujawniania jego wewnętrznej reprezentacji
  - Jeśli chcesz umożliwić jednoczesne działanie wielu procesom przechodzenie po obiektach zagregowanych
- **Konsekwencje**
  - Umożliwienie zmiany sposobu przechodzenia po agregacie
  - Uproszczenie interfejsu klasy *Aggregate*
  - Możliwość jednoczesnego działania więcej niż jednego procesu przechodzenia po agregacie

Chain of Responsibility – Łańcuch zobowiązań (Operacyjny)



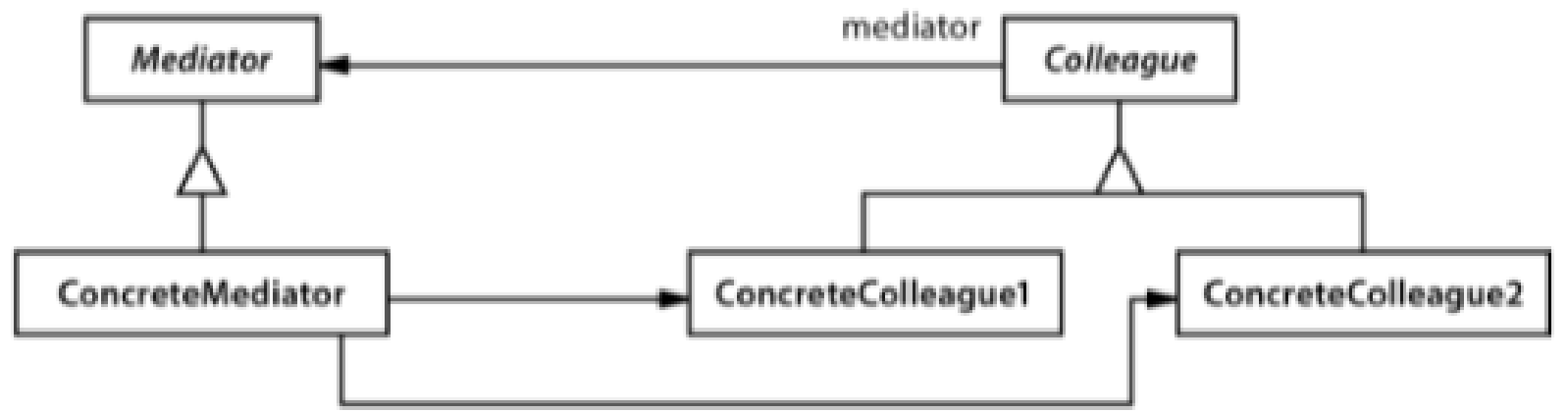
- Pozwala uniknąć wiązania nadawcy z jego odbiorcą
- Łączy w łańcuch obiekty odbiorcze i przekazuje między nimi żądanie do momentu obsłużenia go

Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Chain of Responsibility – Łańcuch zobowiązań (Operacyjny)

- **Stosowanie**
  - Kiedy więcej niż jeden obiekt może obsłużyć żądanie, a nie wiadomo z góry, który z nich to zrobi
  - Jeśli zbiór obiektów, które mogą obsłużyć żądanie, należy określać dynamicznie
- **Konsekwencje**
  - Zapewnia mniejsze powiązanie
  - Zwiększa elastyczność w zakresie przypisywania zadań obiektom
  - Nie zapewnia gwarancji odbioru żądania

Mediator - Mediator (Operacyjny)



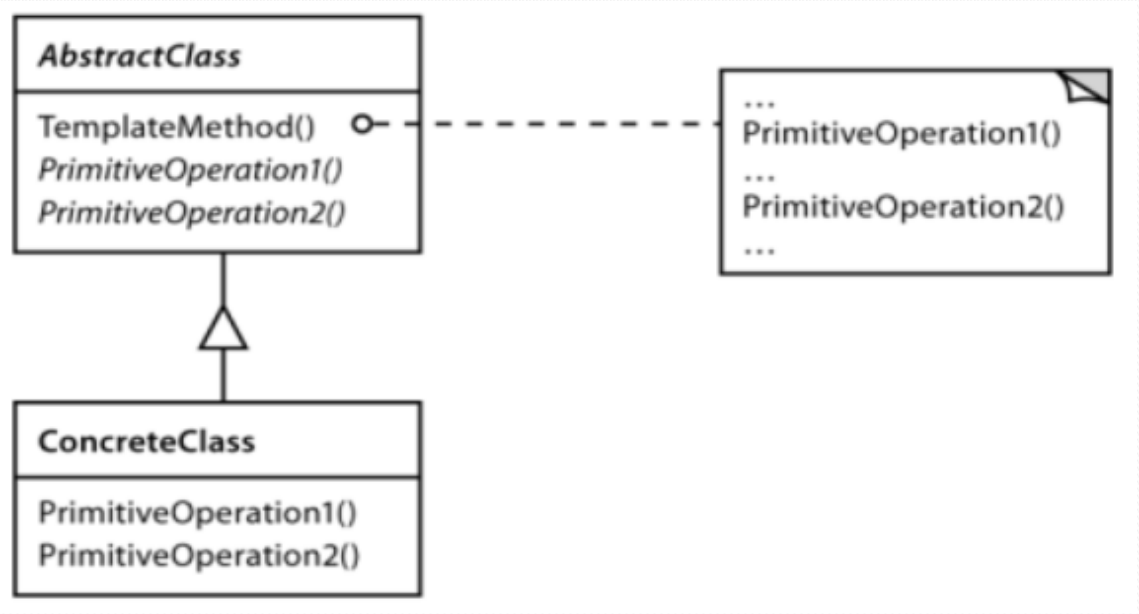
- Określa obiekt enkapsulujący informacje o interakcji między obiektami z danego zbioru.

Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Mediator - Mediator (Operacyjny)

- **Stosowanie**
  - Kiedy zestaw obiektów komunikuje się w dobrze zdefiniowany, ale skomplikowany sposób
  - Powtórne wykorzystanie obiektu jest trudne, ponieważ odwołuje się do wielu innych obiektów i komunikuje się z nimi
  - Dostosowanie zachowania rozproszonego po kilku klasach nie powinno wymagać tworzenia wielu podklas
- **Konsekwencje**
  - Ogranicza tworzenie podklas
  - Rozdziela obiekty współpracujące
  - Rozdziela protokołu obiektów
  - Centralizuje sterowanie

Template method - Metoda szablonowa (Operacyjny)



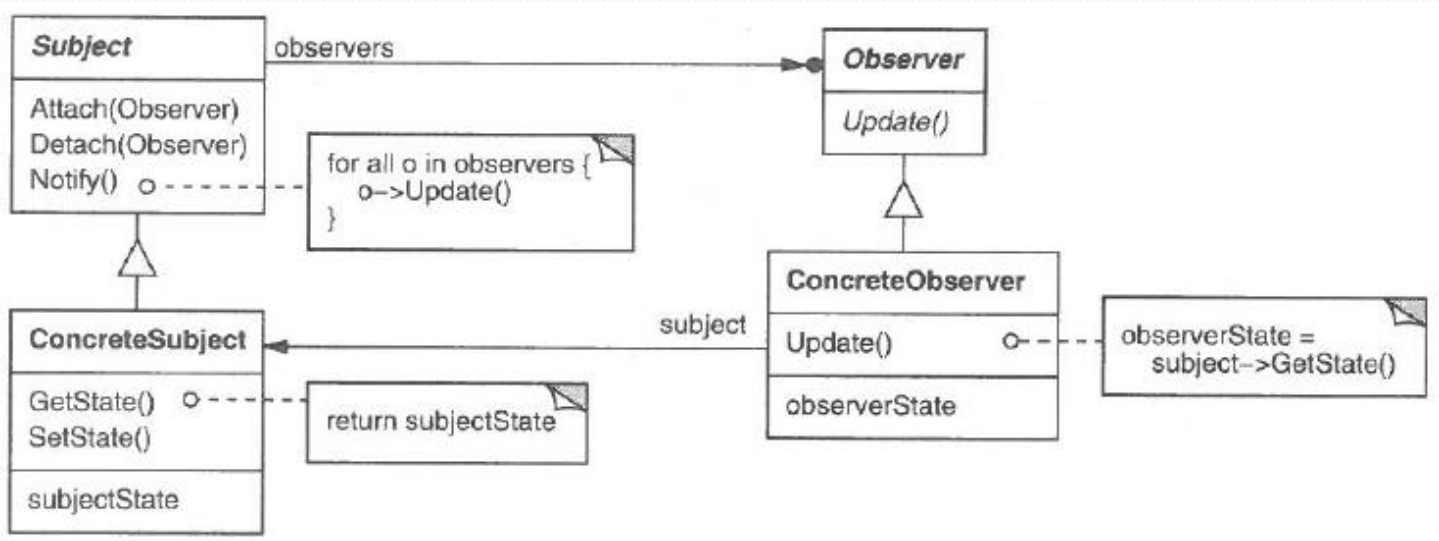
- Określa szkielet algorytmu i pozostawia doprecyzowanie niektórych jego kroków podklasom
- Umożliwia modyfikację niektórych etapów algorytmu w podklasach bez zmiany jego struktury

Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Template method - Metoda szablonowa (Operacyjny)

- **Stosowanie**
  - Do jednorazowego implementowania niezmiennych części algorytmu i umożliwienia implementowania zmieniających się zachowań w podklasach
  - Do kontrolowania rozszerzania podklas
- **Konsekwencje**
  - Odwracają strukturę sterowania wywołując metody przed ich implementacją

Observer - Obserwator (Operacyjny)



- Określa zależność jeden do wielu między obiektami
- Kiedy zmieni się stan jednego z obiektów, wszystkie obiekty zależne od niego są o tym powiadamiane

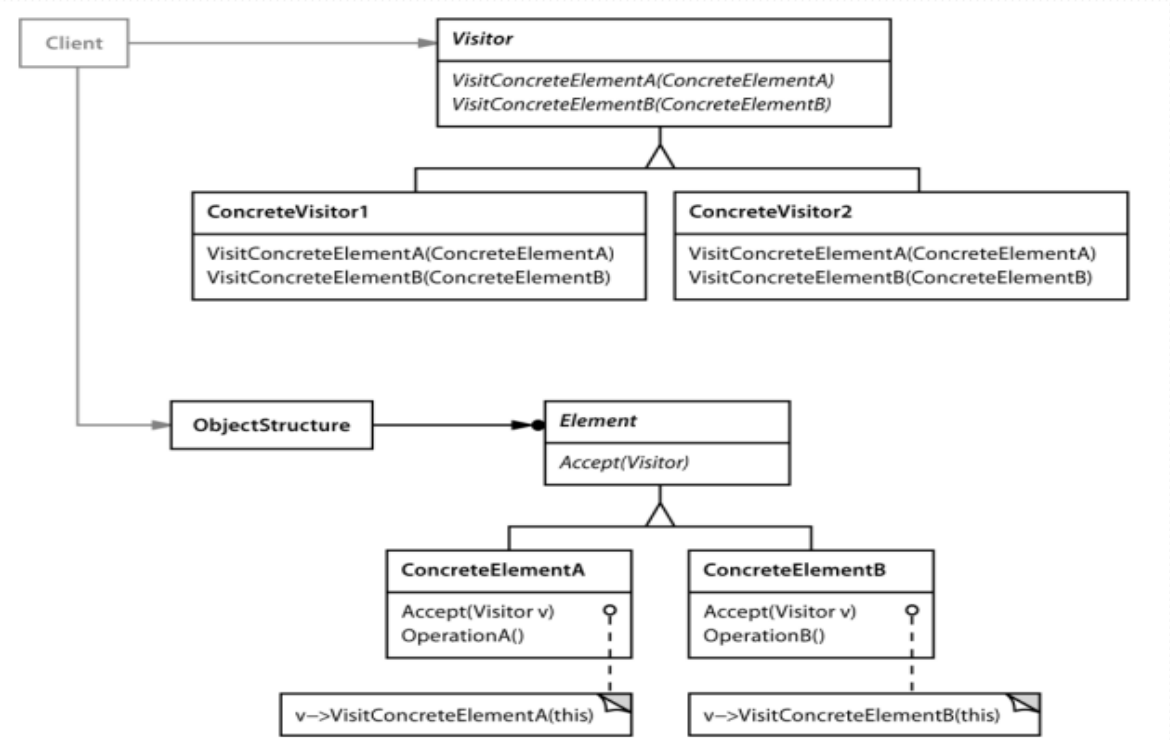
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Observer - Obserwator (Operacyjny)

- **Stosowanie**
  - Kiedy abstrakcja ma dwa aspekty, a jeden z nich zależy od drugiego
  - Jeśli zmiana w jednym obiekcie wymaga zmodyfikowania drugiego, a nie wiadomo, ile obiektów trzeba przekształcić
  - Jeżeli obiekt powinien mów powiadamiać inne bez określania ich rodzaju

- **Konsekwencje**
  - Umożliwia niezależne modyfikowanie podmiotów i obserwatorów
  - Abstrakcyjne powiązanie między obiektami *Subject* i *Observer*
  - Obsługa rozsyłania grupowego komunikatów

Visitor – Odwiedzający (Operacyjny)



- Reprezentuje operację wykonywaną na elementach struktury obiektów.
- Umożliwia zdefiniowanie nowej operacji bez zmieniania klas elementów na której działa

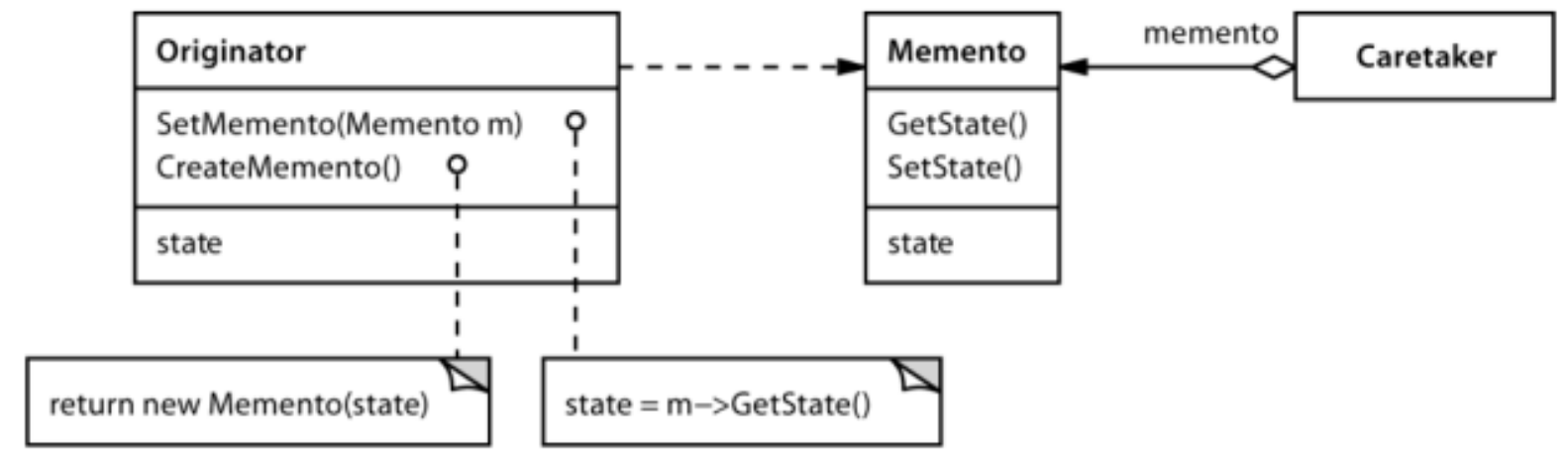
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Visitor – Odwiedzający (Operacyjny)

- **Stosowanie**
  - Jeżeli struktura obiektów obejmuje wiele klas o różnych interfejsach, a chcesz wykonać na tych obiektach operacje zależne od ich klas konkretnych
  - Kiedy na obiektach trzeba wykonać wiele różnych operacji, a chcesz uniknąć „zaśmiecania” klas tymi operacjami

- **Konsekwencje**
  - Ułatwia dodawanie nowych operacji
  - Umożliwia połączenia powiązanych operacji i wyodrębniania niepowiązanych
  - Utrudnia dodawanie nowych klas *ConcreteElement*

Memento – Pamiątka (Operacyjny)



- Bez naruszania kapsułkowania rejestruje i zapisuje w zewnętrznej jednostce stan wewnętrzny obiektu, co umożliwia późniejsze przywrócenie obiektu według zapisanego stanu

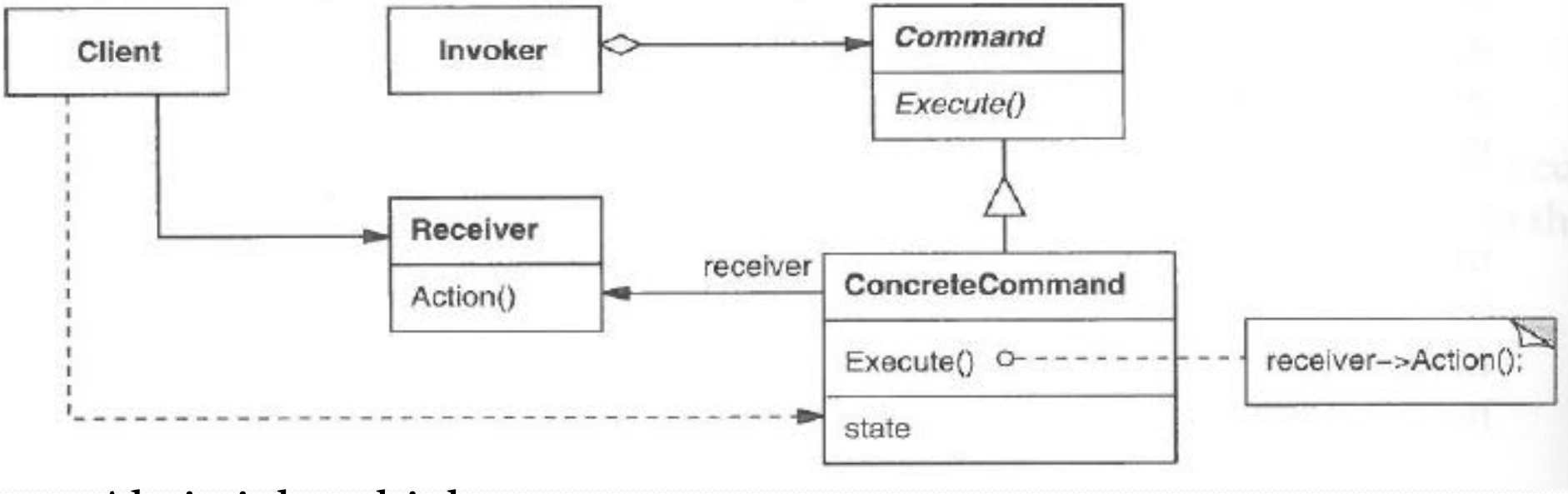
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Memento – Pamiątka (Operacyjny)

- **Stosowanie**
  - Kiedy trzeba zachować obraz stanu obiektu w celu jego późniejszego odtworzenia w tym stanie
  - Bezpośredni interfejs do pobierania stanu spowodowałby ujawnienie szczegółów implementacyjnych obiektu

- **Konsekwencje**
  - Zachowanie granic enkapsulacji
  - Korzystanie z pamiątek może być kosztowne
  - Definiowanie zawężonego i pełnego interfejsu
  - Ukryte koszty zarządzania pamiątkami

Command - Polecenie (Operacyjny)



- Akcja jako obiekt
- Umożliwia parametryzację klienta przy użyciu różncych żądań (akcji) oraz umieszczanie ich w kolejkach i dziennikach

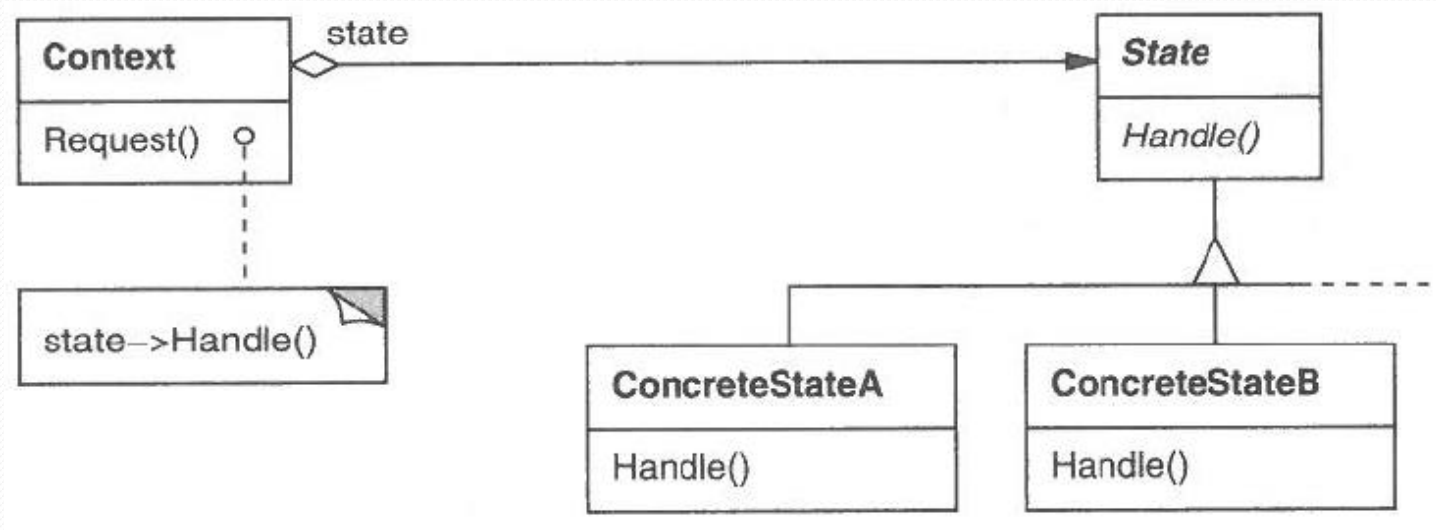
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Command - Polecenie (Operacyjny)

- **Stosowanie**
  - Kiedy chcesz sparametryzować obiekty za pomocą wykonywanych działać
  - Kiedy chcesz umożliwić cofanie zmian przez operacje
  - Kiedy chcesz rejestrować zmiany

- **Konsekwencje**
  - Obiekt *Command* oddziela obiekt wywołujący operację od tego, który potrafi ją wykonać
  - Polecenia można połączyć w polecenia zbiorowe
  - Dodawanie nowych obiektów *Command* jest proste, ponieważ nie wymaga modyfikowania istniejących klas

State – Stan (Operacyjny)



- Umożliwia obiektowi modyfikację zachowania w wyniku zmiany jego stanu wewnętrznego

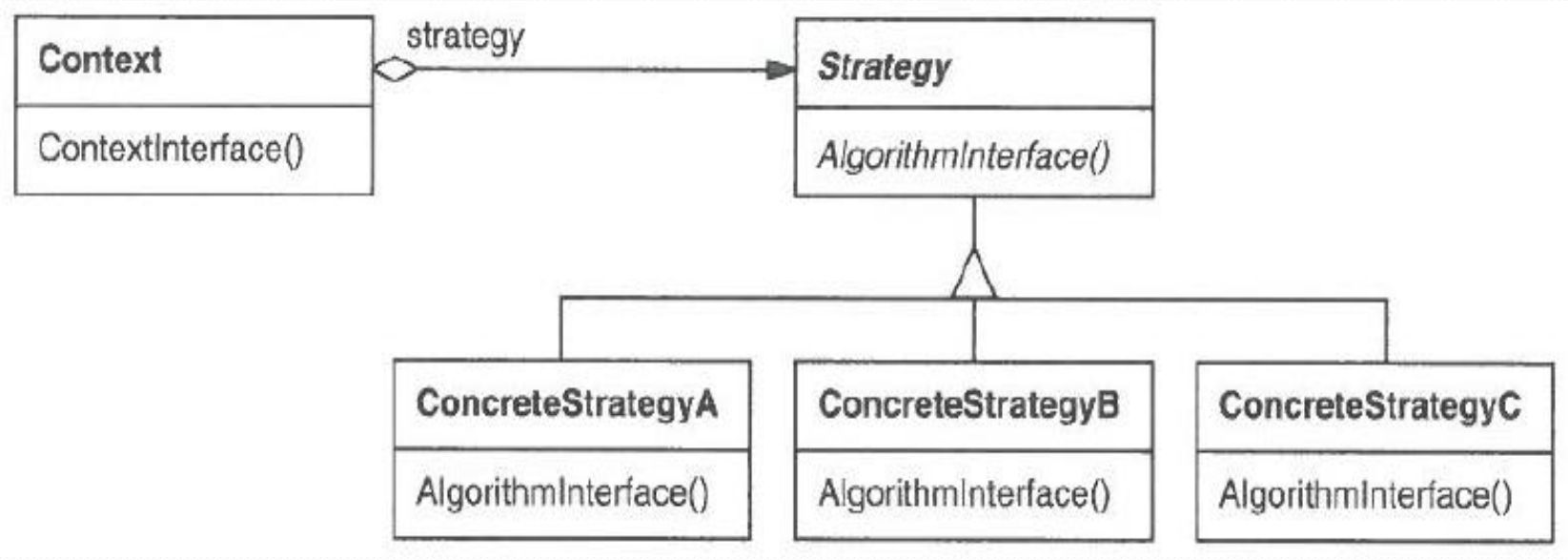
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

State – Stan (Operacyjny)

- **Stosowanie**
  - Zachowanie obiektu zależy od jego stanu
  - Operacje obejmują długie wieloczęściowe instrukcje warunkowe zależne od stanu obiektu

- **Konsekwencje**
  - Pozwala umieścić w jednym miejscu zachowanie specyficzne dla stanu i rozdzielić zachowania powiązane z różnymi stanami
  - Powoduje, że zmiany stanu są jawne
  - Stan obiektów można współużytkować (wtedy jest też Pyłkiem)

Strategy – Strategia (Operacyjny)



- Określa rodzinę algorytmów, enkapsuluje każdy z nich i umożliwia ich zamienne stosowanie

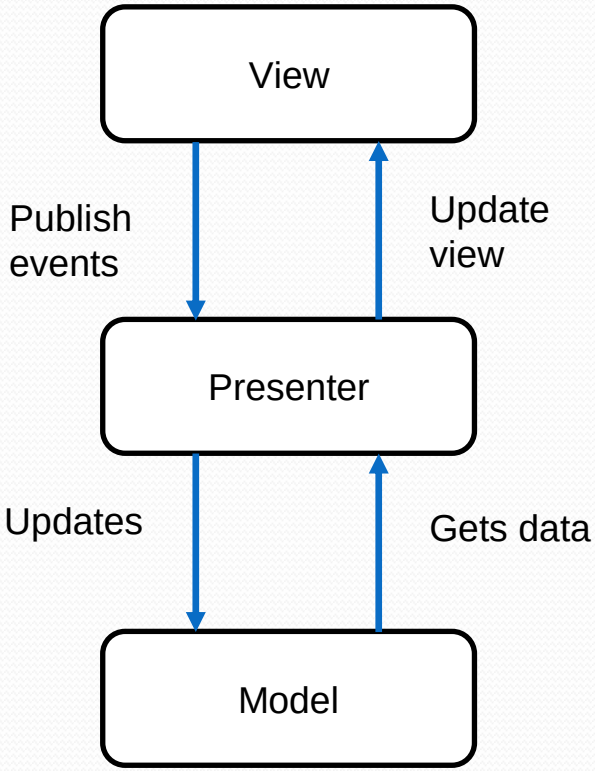
Źródło: “Design Patterns, Elements of Reusable Object Oriented Software” E.Gamma, R. Helm

Strategy – Strategia (Operacyjny)

- **Stosowanie**
  - Kiedy wiele powiązanych klas różni się tylko zachowaniem
  - Jeśli potrzebne są różne wersje algorytmu
  - Jeżeli algorytm korzysta z danych o których klienci nie powinni wiedzieć
- **Konsekwencje**
  - Powoduje powstawanie rodzin powiązanych algorytmów
  - Zapewnia alternatywę dla tworzenia podklas
  - Pozwala wyeliminować instrukcje warunkowe
  - Klienci muszą znać różne strategie
  - Powoduje koszty komunikacji między obiektami *Strategy* i *Context*

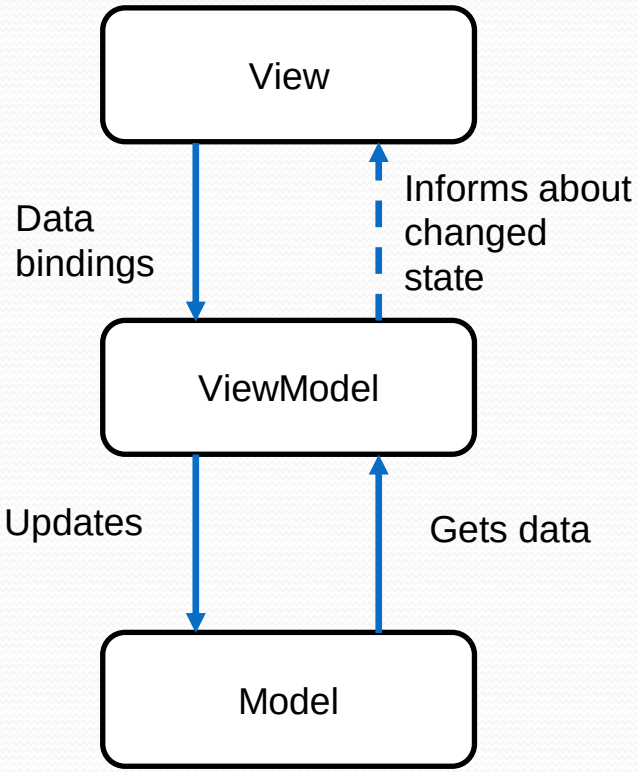
Model-View-Presenter

- Widok
  - Deleguje zdarzenia do Presentera
  - Odświeża wygląd po zmianie Presentera
- Model
  - Odzwierciedla strukturę danych
- Presenter
  - Wykonuje akcje wywołane na widoku
  - Modyfikuje model
  - Zmienia widok



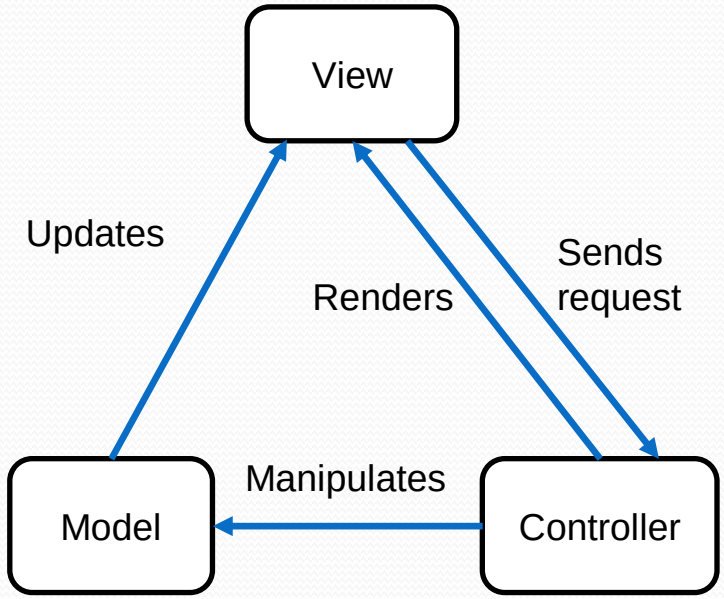
Model-View-ViewModel

- Widok
  - Odczytuje interakcje użytkownika i wysyła zdarzenia do ViewModela za pomocą mechanizmu data binding
  - Odświeża wygląd po zmianie
- Model
  - Odzwierciedla strukturę danych
- ViewModel
  - Przy pomocy bindera odczytuje gesty użytkownika na widoku i wykonuje akcje
  - Modyfikuje stan modelu
  - Za pomocą bindera do publicznych właściwości oraz komend modyfikuje widok



Model-View-Controller

- Widok
  - Wywołuje akcje (requests) na Controllerze
  - Wyświetla Dane zawarte w modelu
- Model
  - Odzwierciedla strukturę danych
  - Przekazuje dane do widoku
- Kontroler
  - Wykonuje akcje żądane przez Widok
  - Procesuje całą logikę i zmienia Model
  - Podpina model do wybranego Widoku i przesyła do użytkownika



Podsumowanie