https://www.youtube.com/playlist?list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc

Reguła projektowania

Zidentyfikuje fragmenty aplikacji, które się zmieniają i oddziel je od tych, które pozostają stałe.

STRATEGIA
(symulator kaczki)

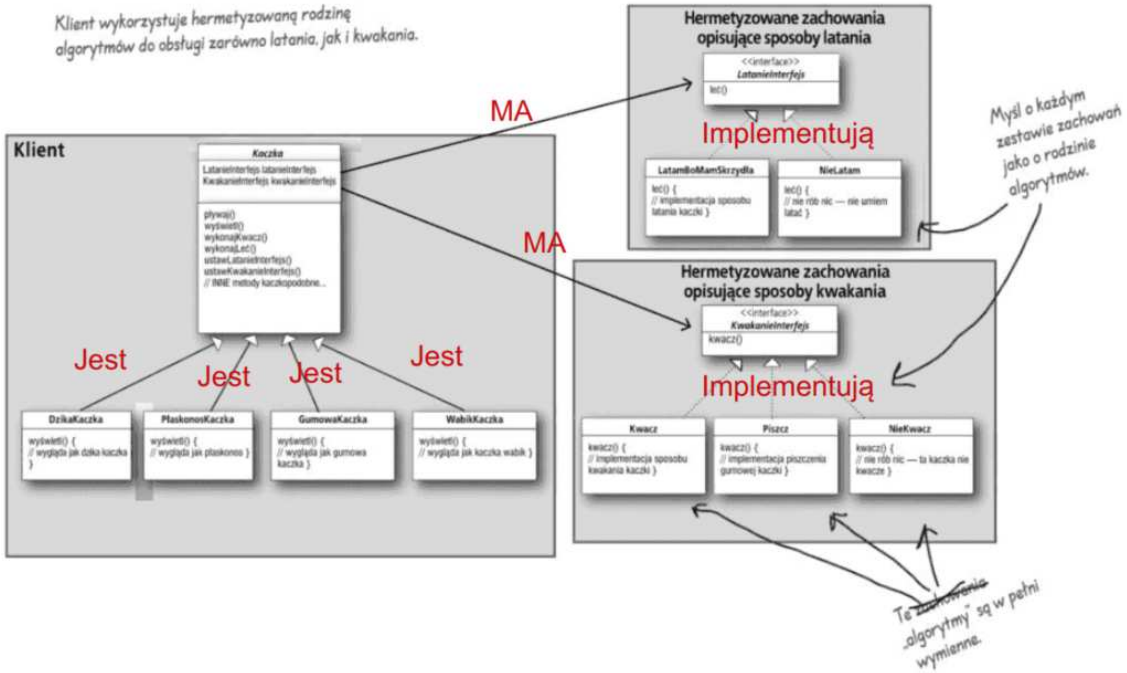
Reguła projektowania

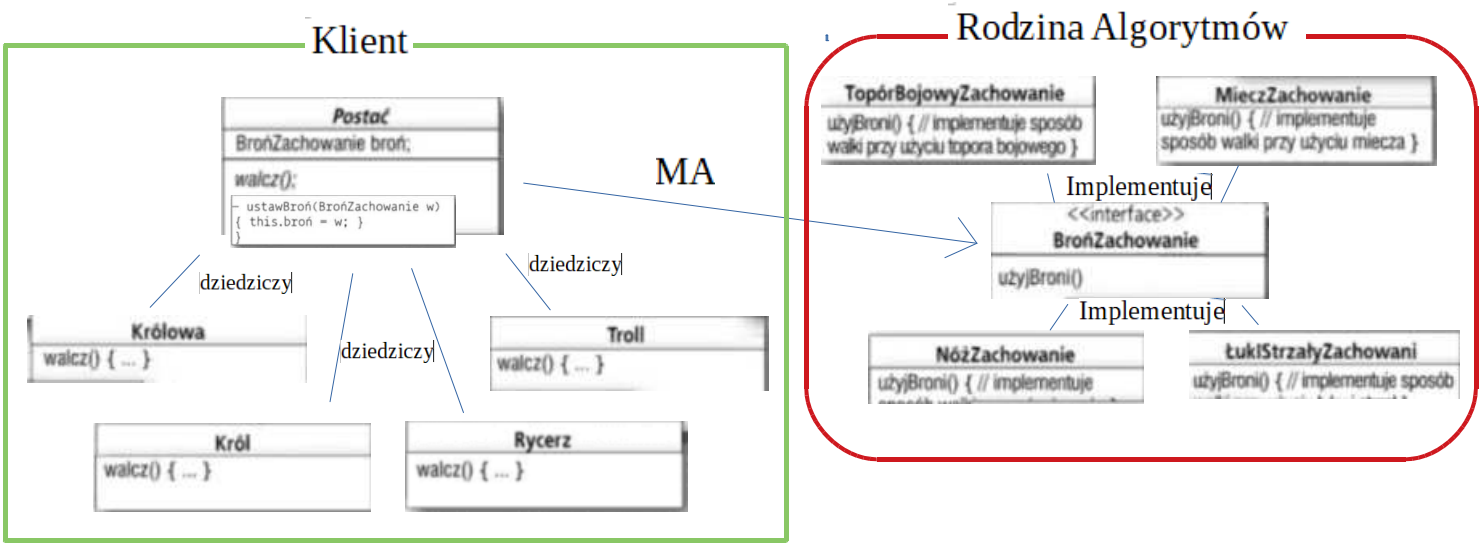
Skoncentruj się na tworzeniu interfejsów, a nie implementacji.

Reguła projektowania

Przedkładaj kompozycję nad dziedziczenie.

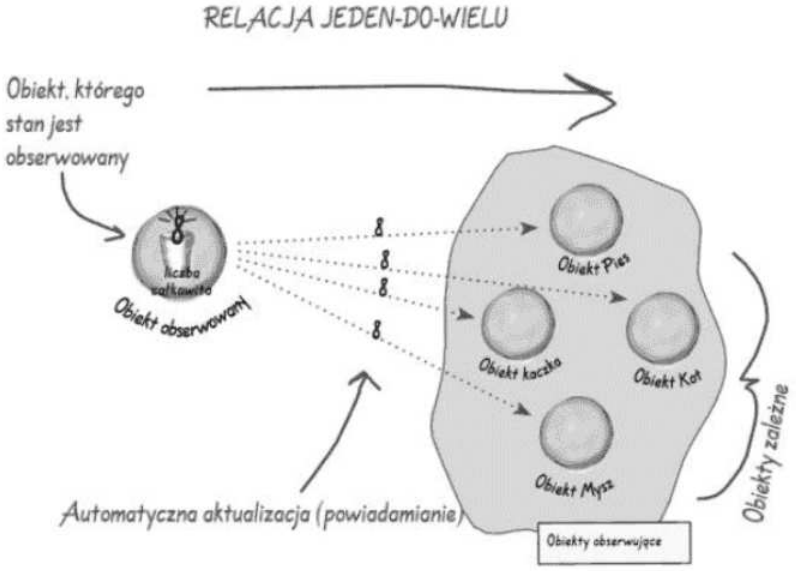
Wzorzec Strategia definiuje rodzinę algorytmów, pakuje je jako osobne klasy i powoduje, że są one w pełni wymienne. Zastosowanie tego wzorca pozwala na to, aby zmiany w implementacji algorytmów przetwarzania były całkowicie niezależne od strony klienta, który z nich korzysta.





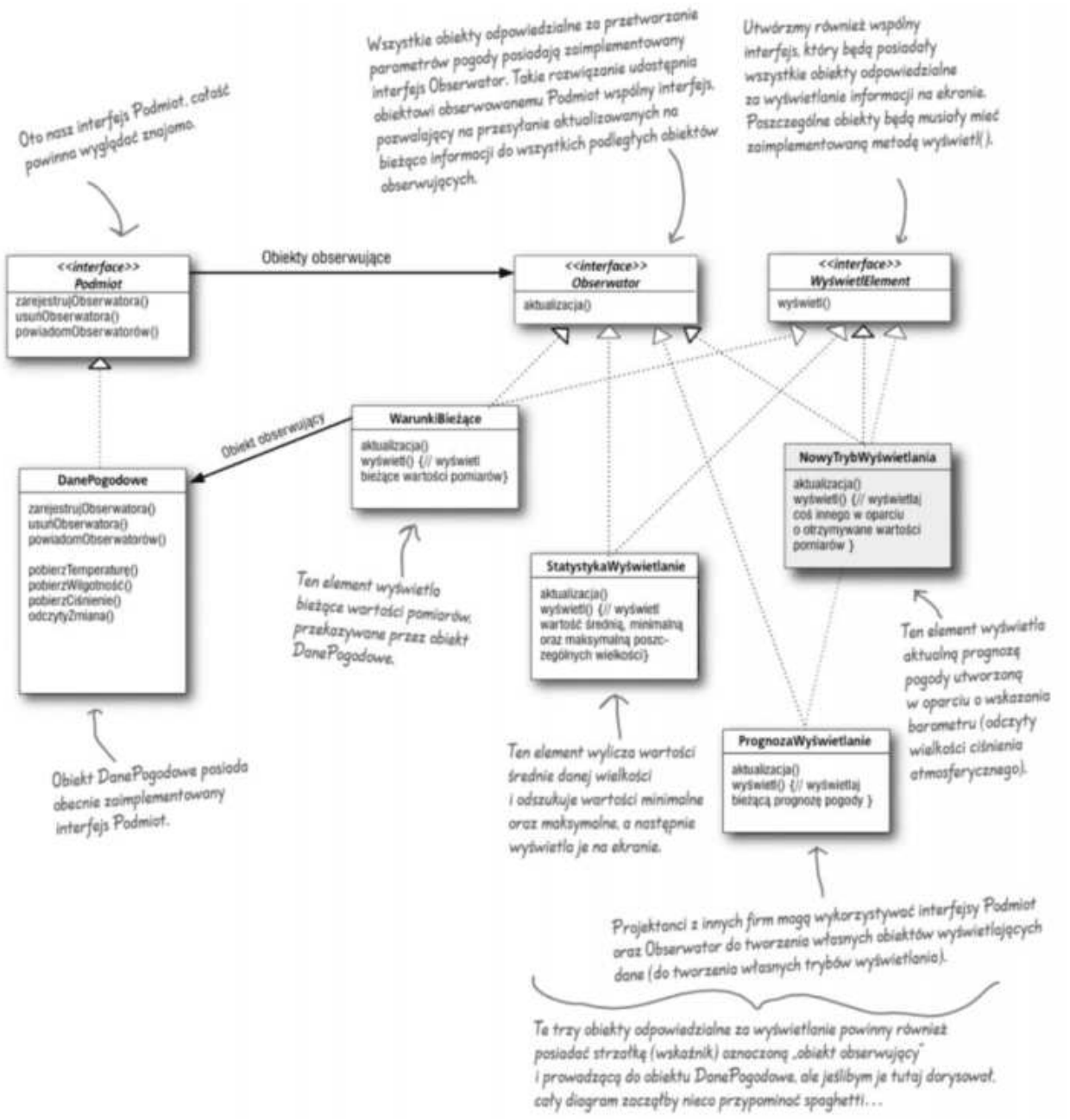
OBSERWATOR

Wzorzec Obserwator definiuje pomiędzy obiektami relację jeden-do-wielu w taki sposób, że kiedy wybrany obiekt zmienia swój stan, to wszystkie jego obiekty zależne zostają o tym powiadomione i automatycznie zaktualizowane.



Reguła projektowania

Staraj się tworzyć projekty, w których obiekty są ze sobą luźno powiązane i, o ile to możliwe, nie oddziałują na siebie wzajemnie.

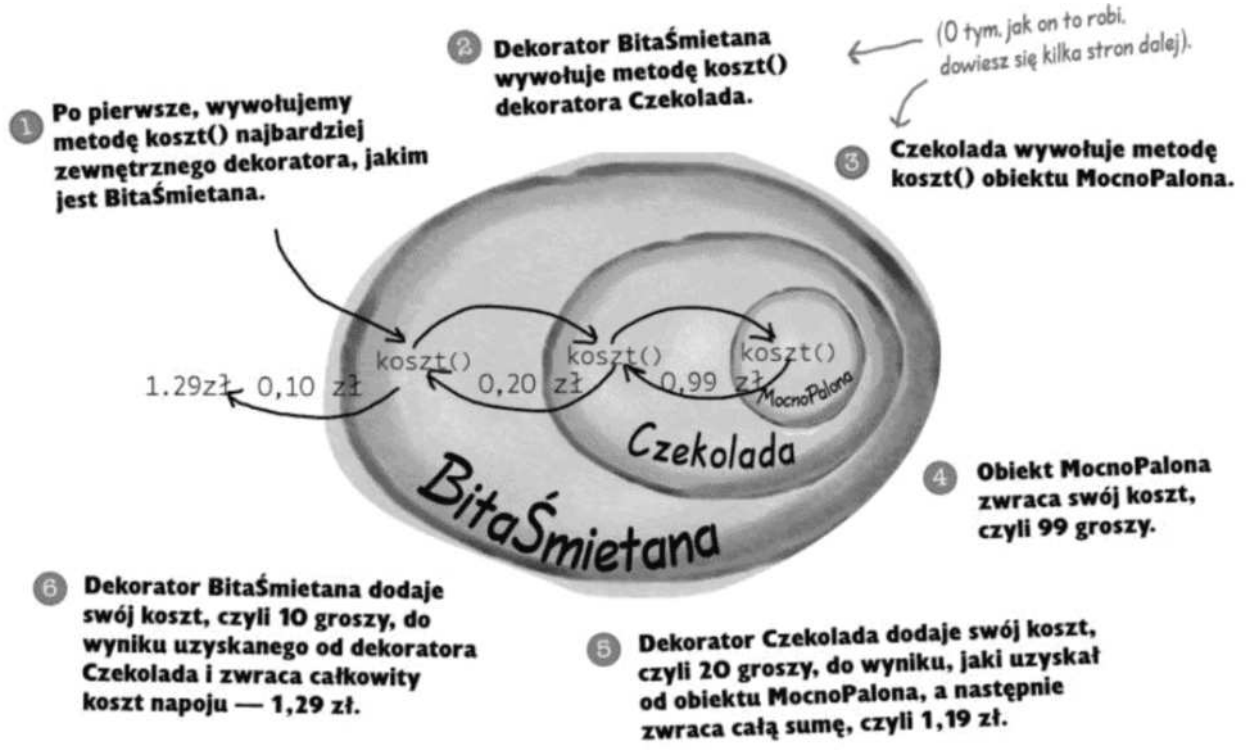


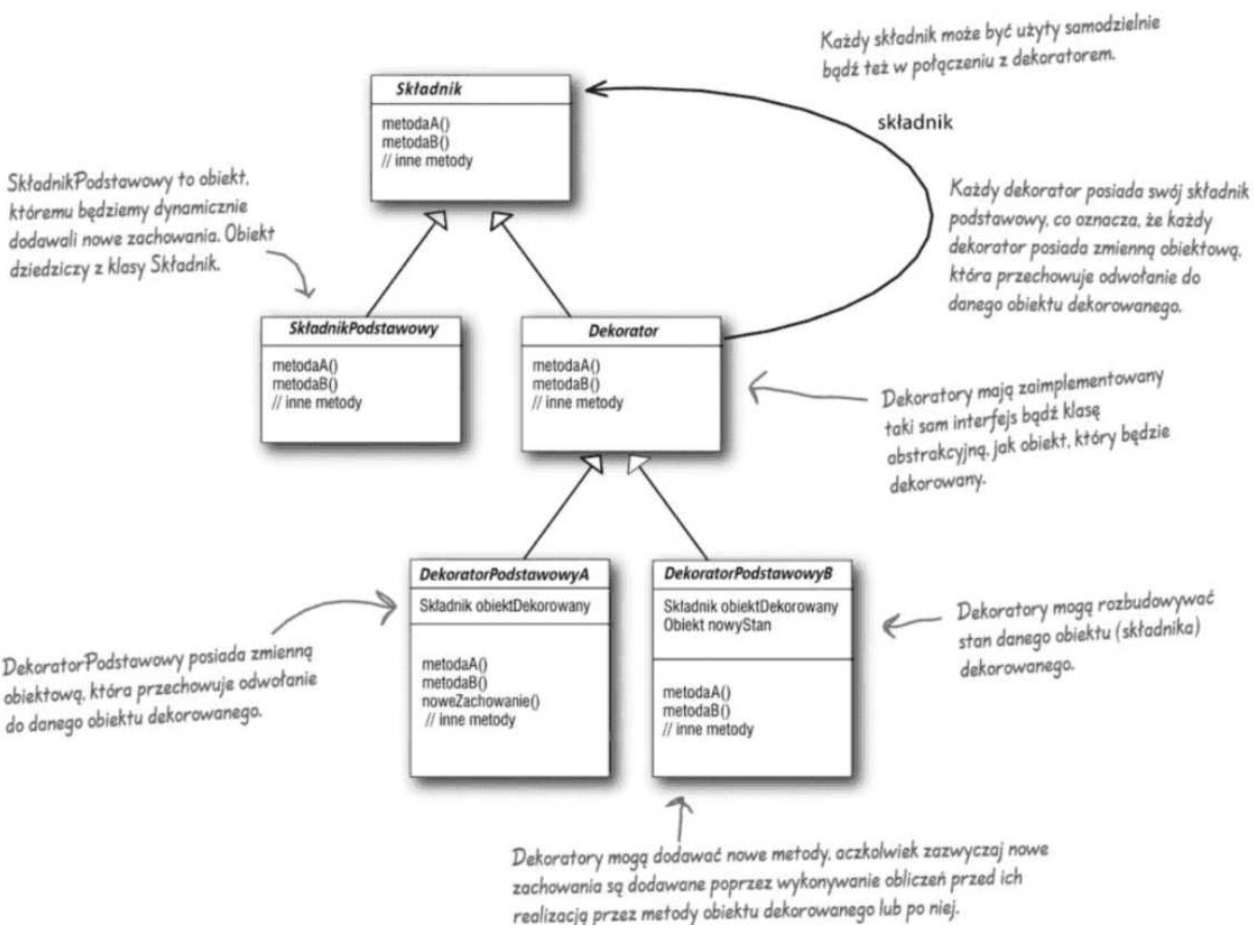
DEKORATOR

Reguła projektowania

Klasy powinny być otwarte na rozbudowę, ale zamknięte na modyfikacje.

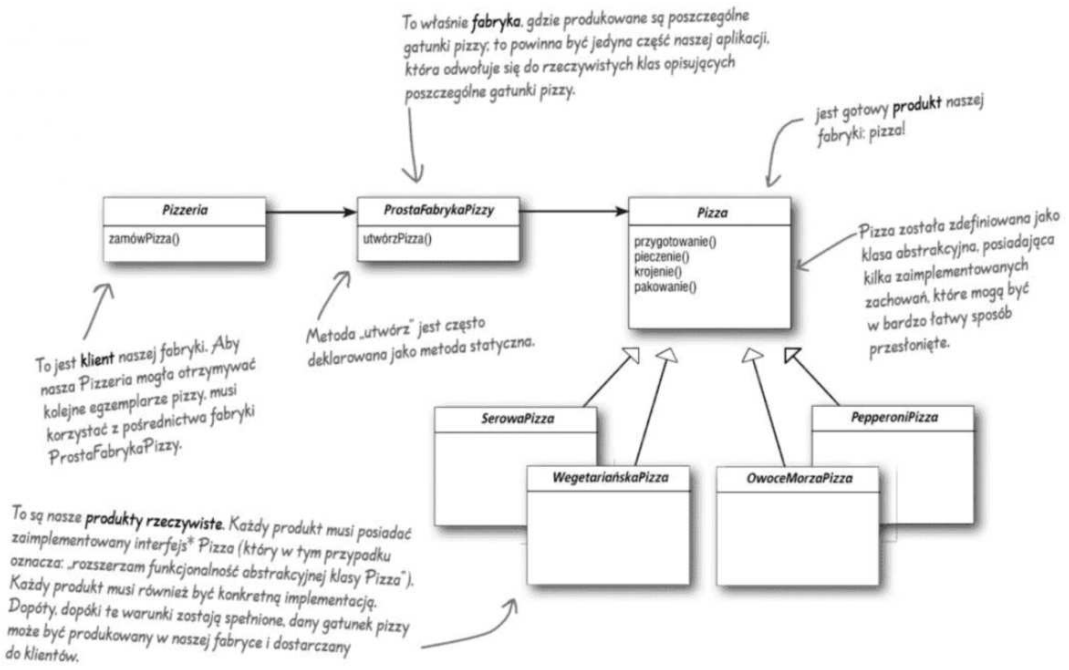
Wzorzec Dekorator pozwala na dynamiczne przydzielanie danemu obiektowi nowych zachowań. Dekoratory dają elastyczność podobną do tej, jaką daje dziedziczenie, oferując jednak w zamian znacznie rozszerzoną funkcjonalność.

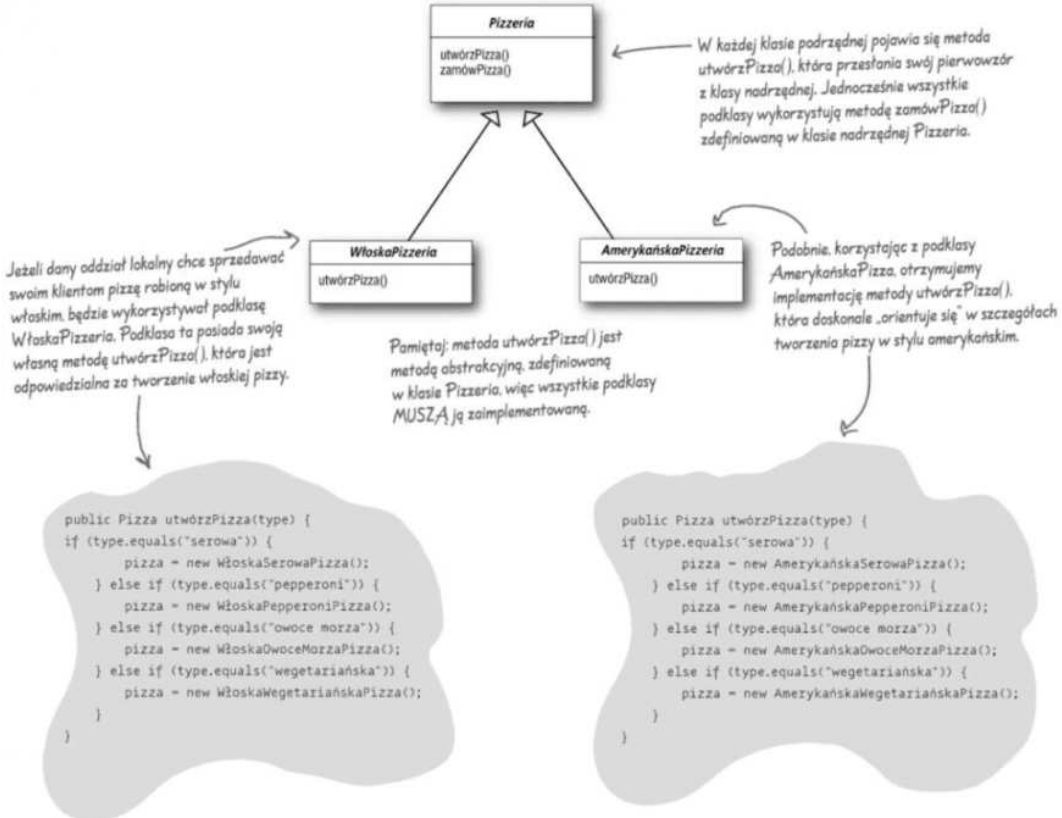

1 Po pierwsze, wywołujemy metodę koszt() najbardziej zewnętrznego dekoratora, jakim jest BitaŚmietana.
2 Dekorator BitaŚmietana wywołuje metodę koszt() dekoratora Czekolada.
(O tym, jak on to robi, dowiesz się kilka stron dalej).
3 Czekolada wywołuje metodę koszt() obiektu MocnoPalona.
koszt()
koszt()
koszt()
1.29zł
0,10 zł
0,20 zł
0,99 zł
MocnoPalona
Czekolada
BitaŚmietana
4 Obiekt MocnoPalona zwraca swój koszt, czyli 99 groszy.
5 Dekorator Czekolada dodaje swój koszt, czyli 20 groszy, do wyniku, jaki uzyskał od obiektu MocnoPalona, a następnie zwraca całą sumę, czyli 1,19 zł.
6 Dekorator BitaŚmietana dodaje swój koszt, czyli 10 groszy, do wyniku uzyskanego od dekoratora Czekolada i zwraca całkowity koszt napoju — 1,29 zł.


Każdy składnik może być użyty samodzielnie bądź też w połączeniu z dekoratorem.
Składnik
metodaA()
metodaB()
// inne metody
składnik
SkładnikPodstawowy to obiekt, któremu będziemy dynamicznie dodawali nowe zachowania. Obiekt dziedziczy z klasy Składnik.
Każdy dekorator posiada swój składnik podstawowy, co oznacza, że każdy dekorator posiada zmienną obiektową, która przechowuje odwołanie do danego obiektu dekorowanego.
SkładnikPodstawowy
metodaA()
metodaB()
// inne metody
Dekorator
metodaA()
metodaB()
// inne metody
Dekoratory mają zaimplementowany taki sam interfejs bądź klasę abstrakcyjną, jak obiekt, który będzie dekorowany.
DekoratorPodstawowyA
Składnik obiektDekorowany
metodaA()
metodaB()
noweZachowanie()
// inne metody
DekoratorPodstawowyB
Składnik obiektDekorowany
Obiekt nowyStan
metodaA()
metodaB()
// inne metody
DekoratorPodstawowy posiada zmienną obiektową, która przechowuje odwołanie do danego obiektu dekorowanego.
Dekoratory mogą rozbudowywać stan danego obiektu (składnika) dekorowanego.
Dekoratory mogą dodawać nowe metody, aczkolwiek zazwyczaj nowe zachowania są dodawane poprzez wykonywanie obliczeń przed ich realizacją przez metody obiektu dekorowanego lub po niej.

FABRYKA

„Prosta fabryka”

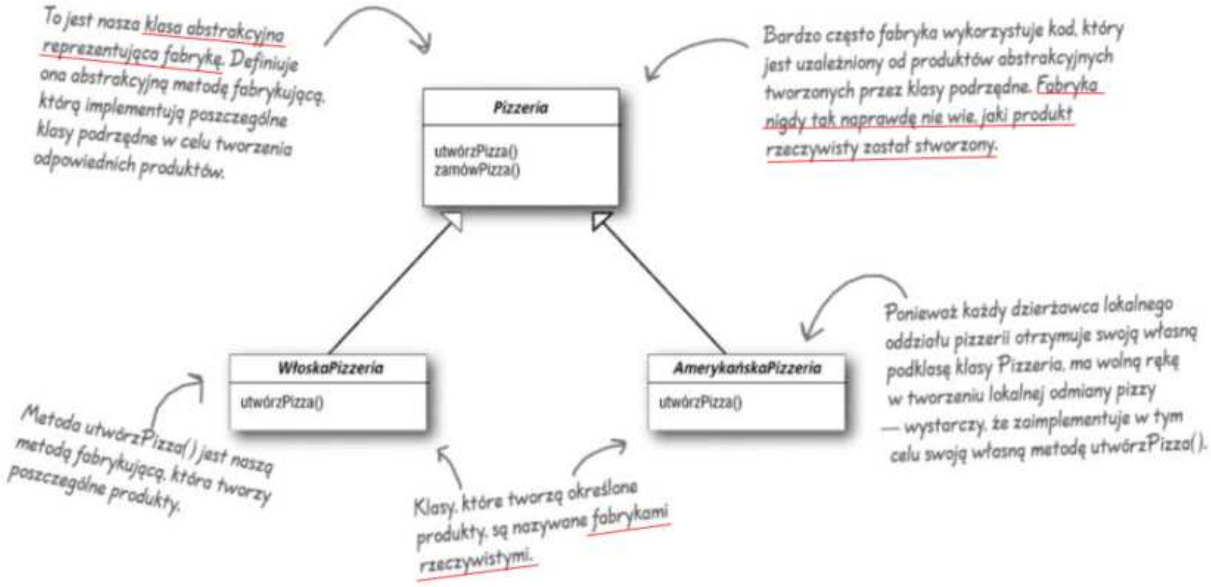




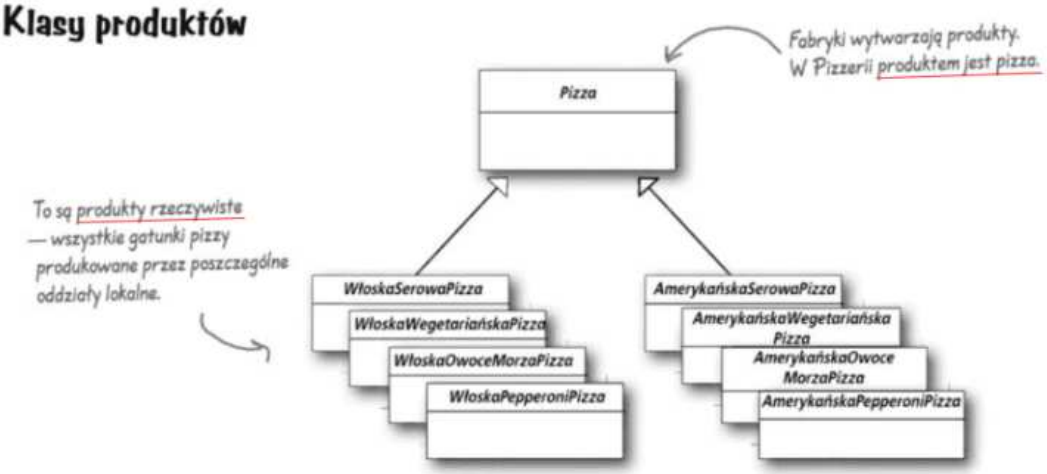
Wzorzec Metoda Fabrykująca definiuje interfejs pozwalający na tworzenie obiektów, ale pozwala klasom podrzędnym decydować, jakiej klasy obiekt zostanie utworzony. Wzorzec Metoda Fabrykująca przekazuje odpowiedzialność za tworzenie obiektów do klas podrzędnych.

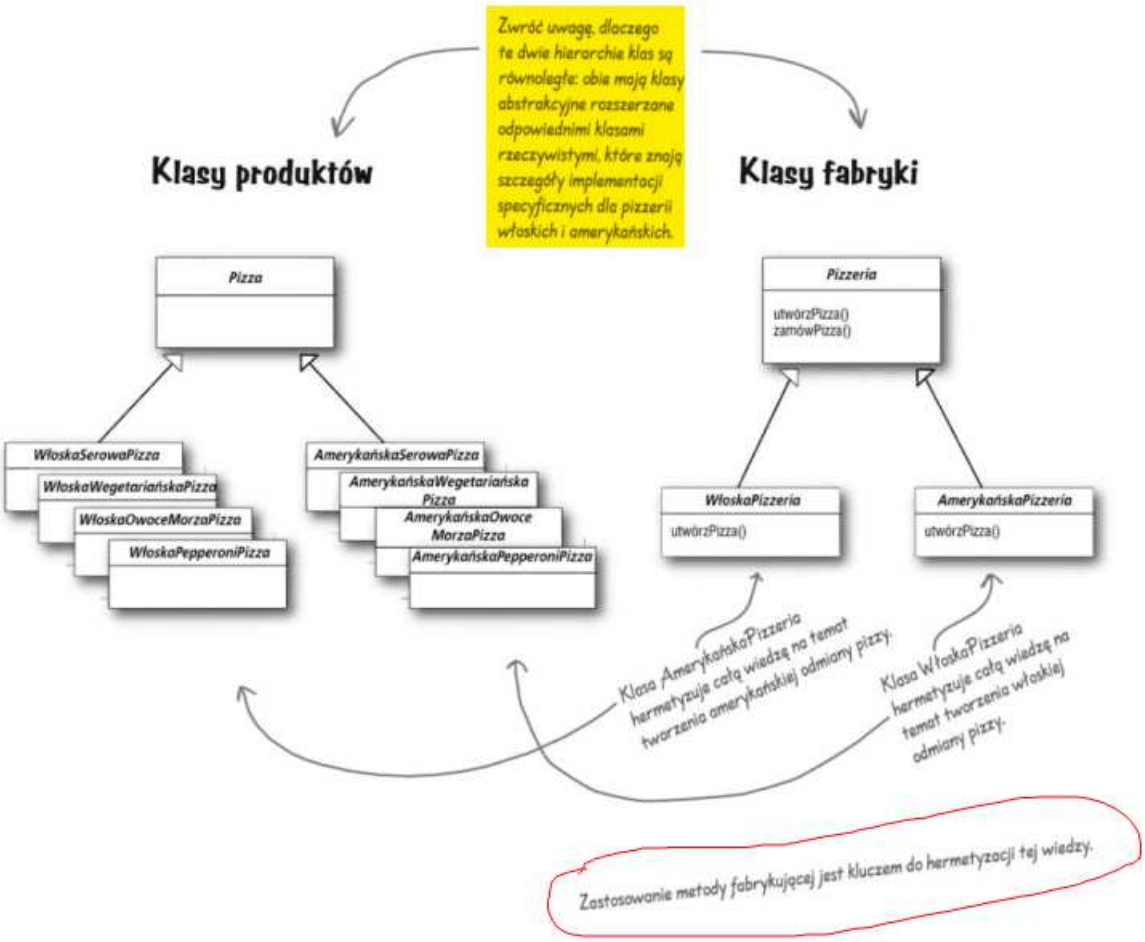
Klasy fabryki

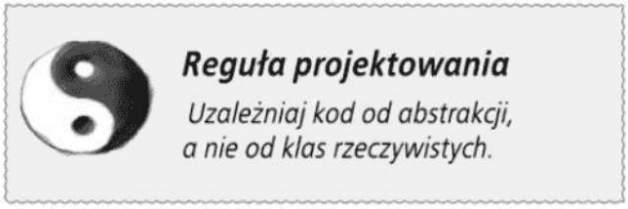
Jest to pewna odmiana metody szablonowej



Klasy produktów

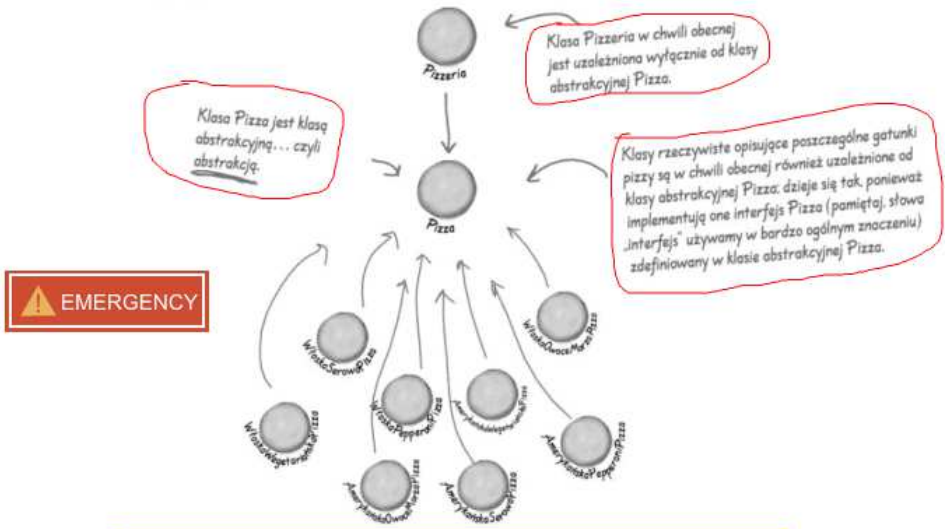




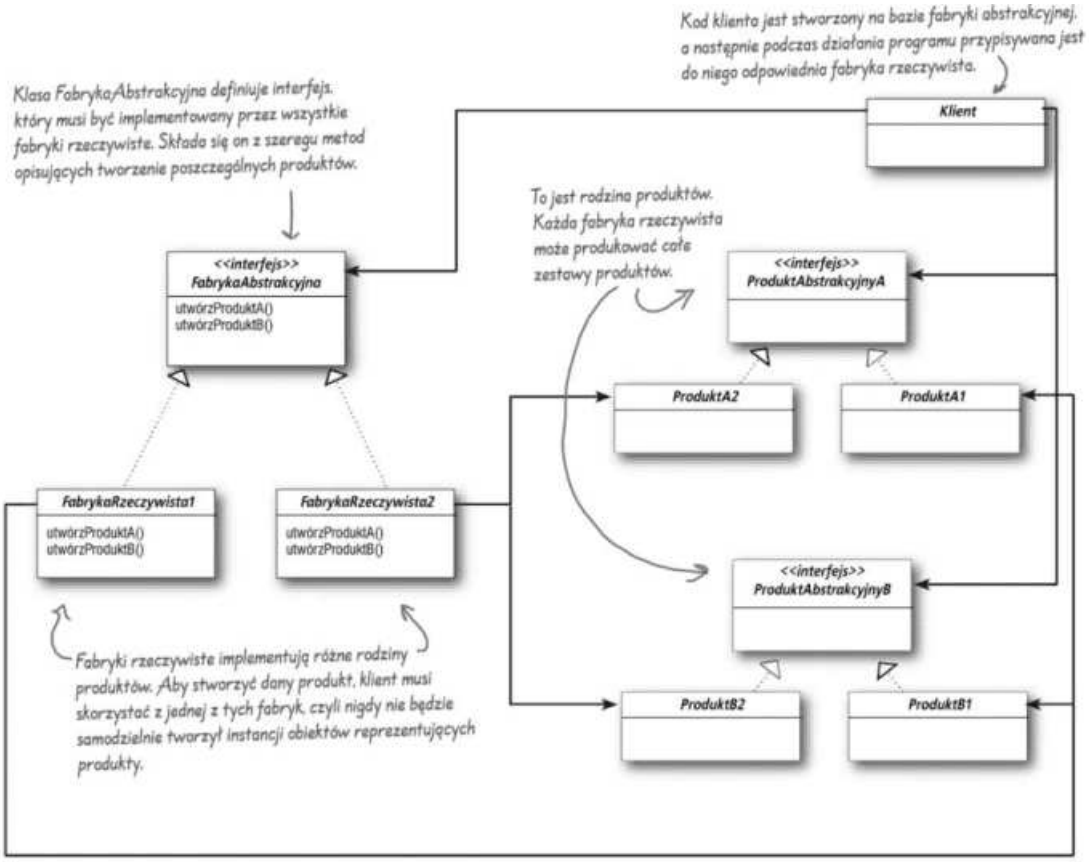


Reguła projektowania

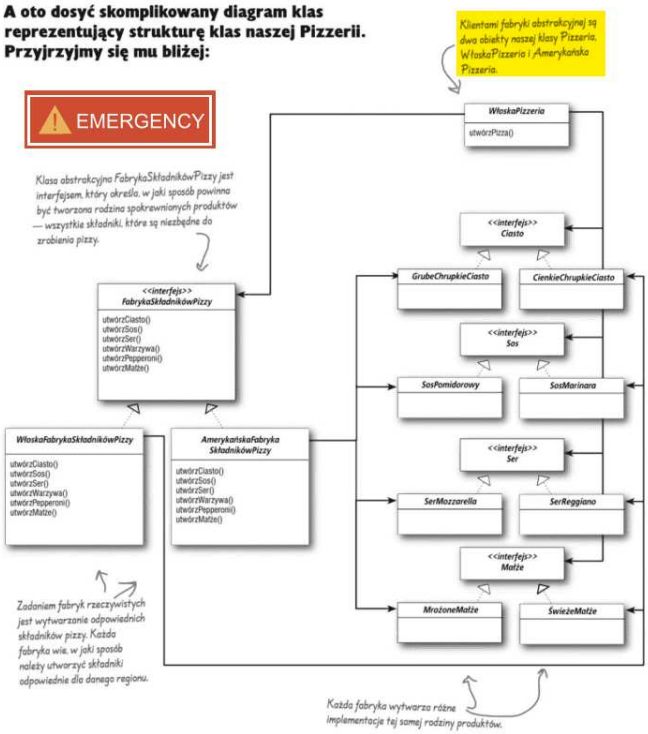
Uzależniaj kod od abstrakcji, a nie od klas rzeczywistych.



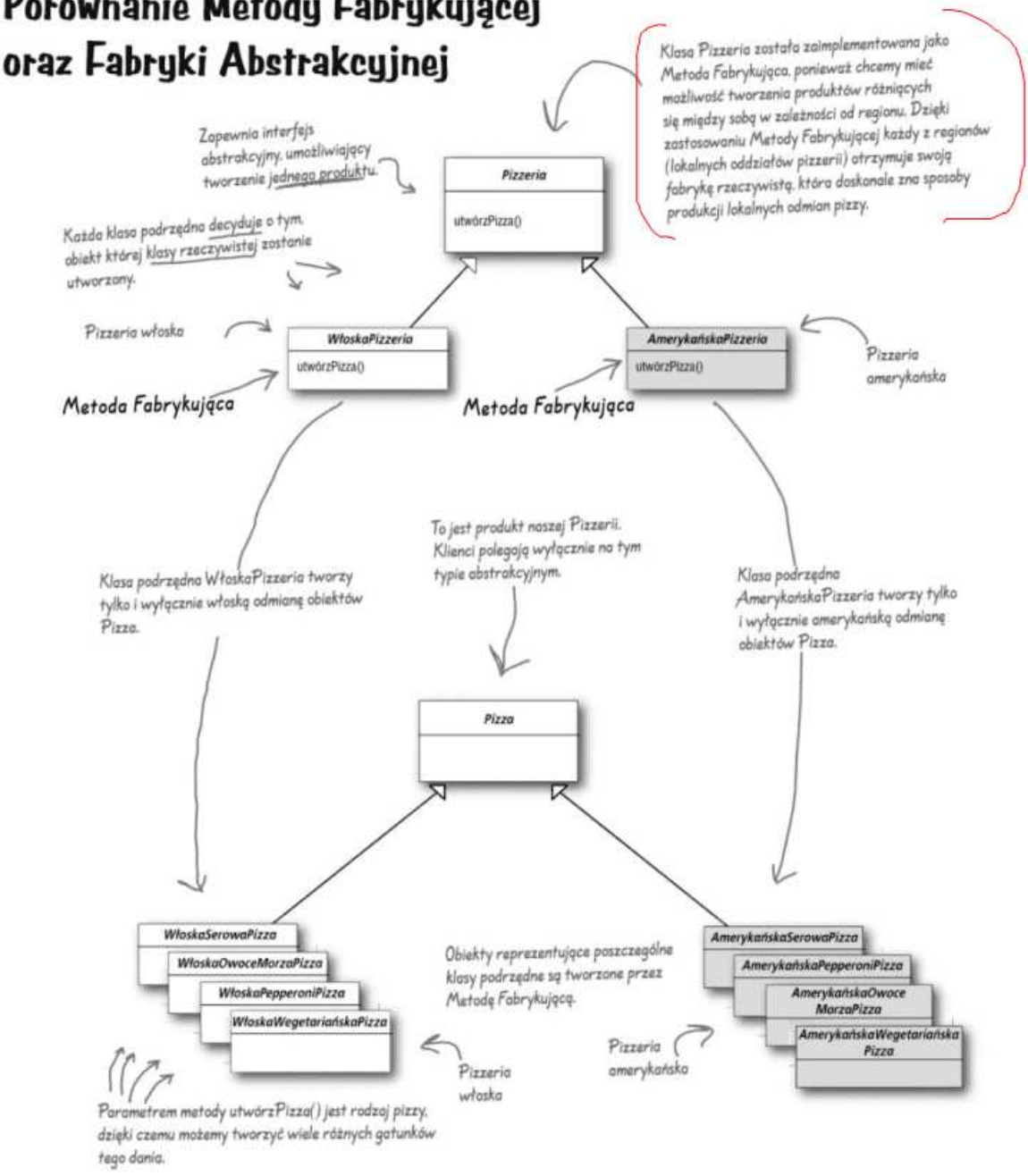
Wzorzec Fabryka Abstrakcyjna dostarcza interfejs do tworzenia całych rodzin spokrewnionych lub zależnych od siebie obiektów bez konieczności określania ich klas rzeczywistych.

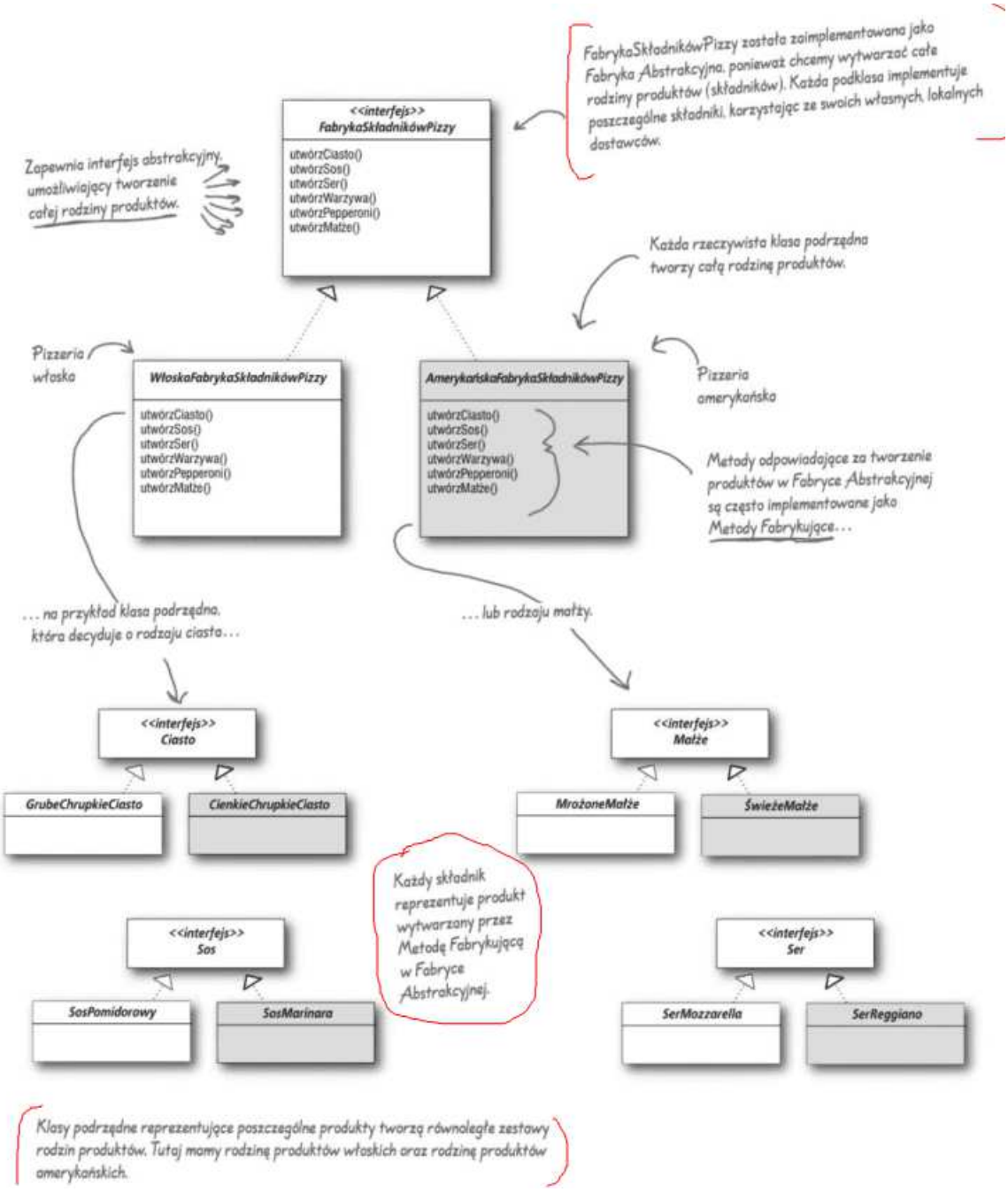


A oto dosyć skomplikowany diagram klas reprezentujący strukturę klas naszej Pizzerii. Przyjrzyjmy się mu bliżej:



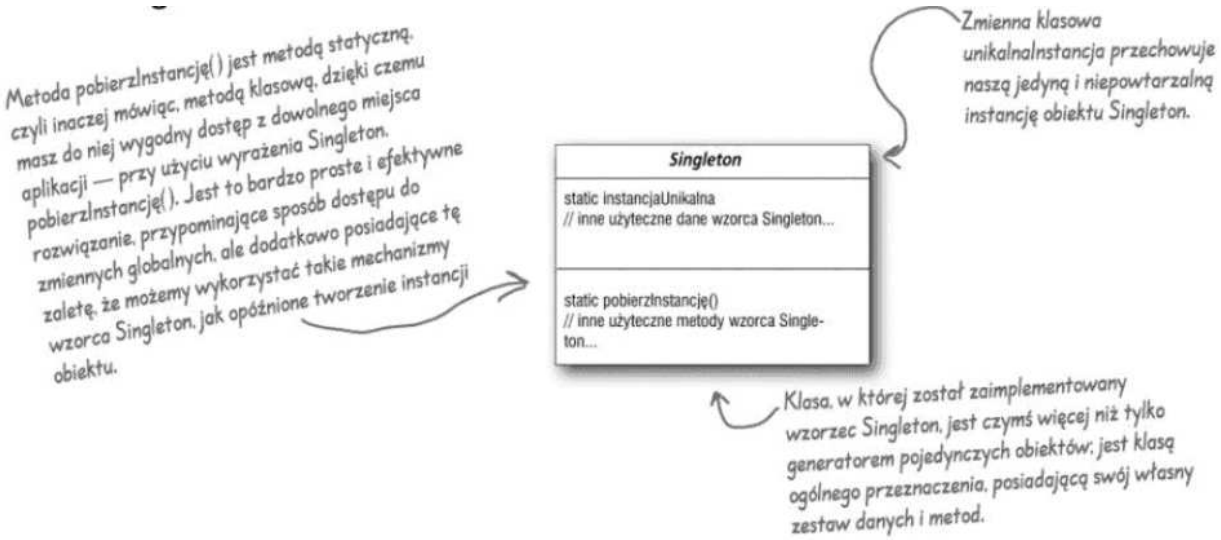
Porównanie Metody Fabrykującej oraz Fabryki Abstrakcyjnej





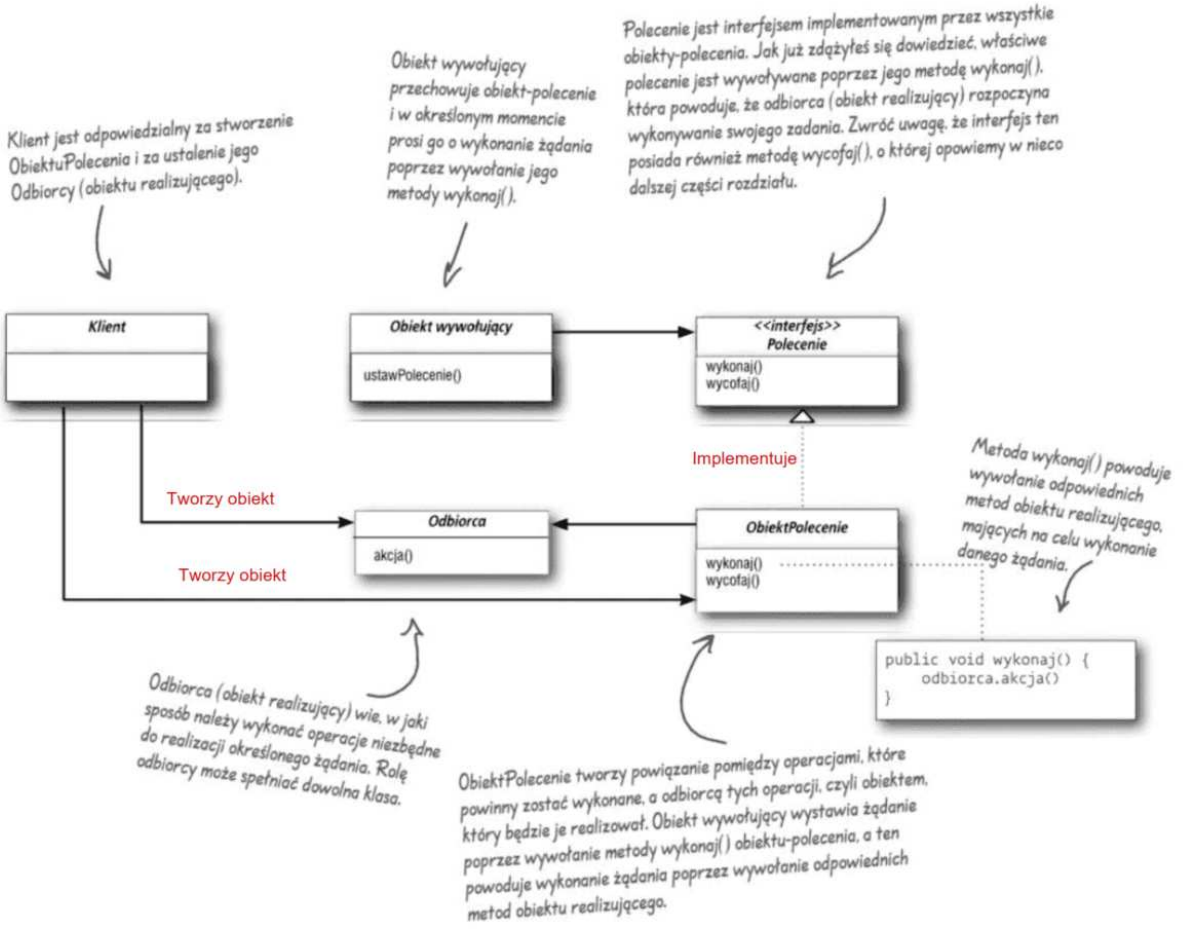
SINGLETON

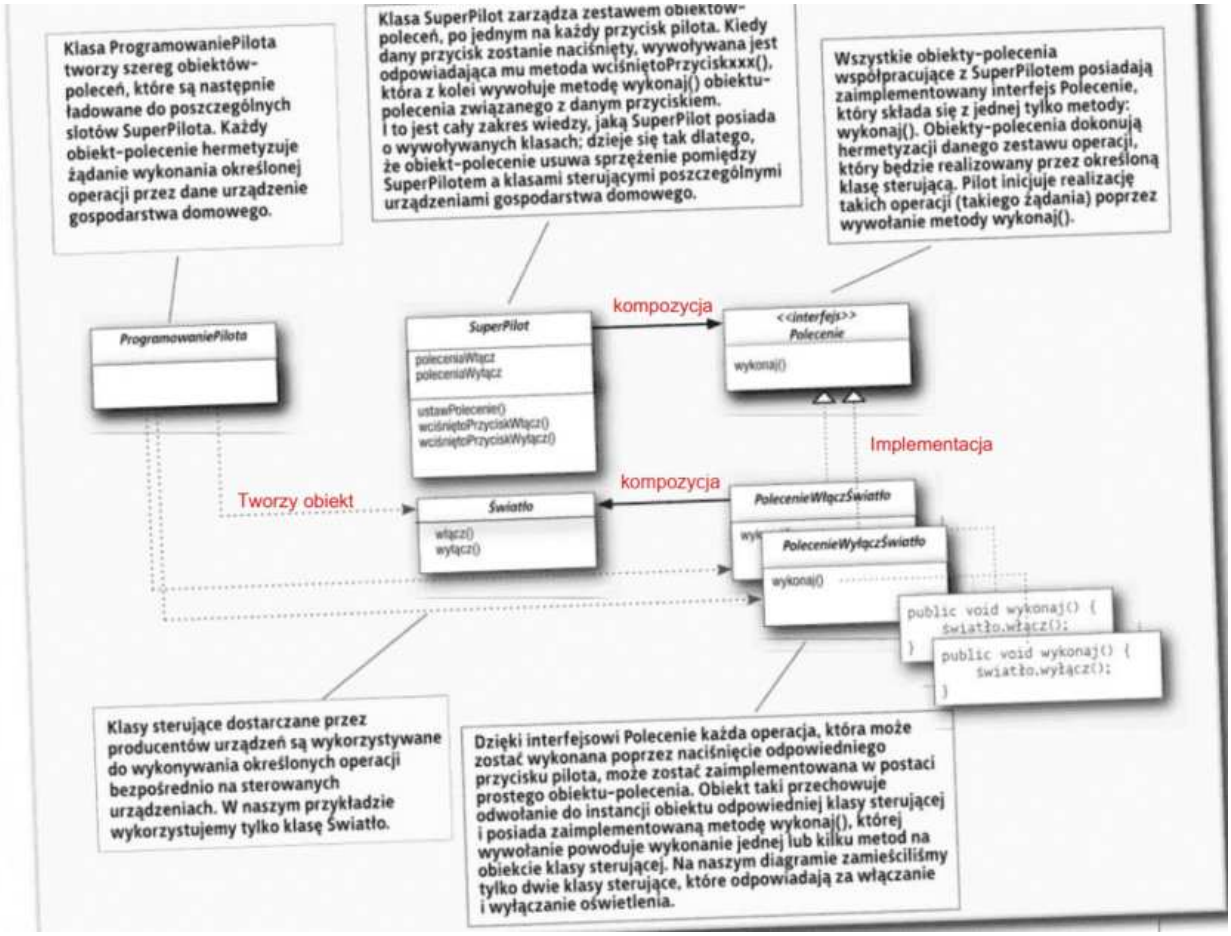
Wzorzec Singleton zapewnia, że dana klasa będzie miała tylko i wyłącznie jedną instancję obiektu, i zapewnia globalny punkt dostępu do tej instancji.



COMMAND

Wzorzec Polecenie hermetyzuje żądania w postaci obiektów, co umożliwia parametryzowanie różnych obiektów zróżnicowanymi żądaniami (takimi jak np. żądania kolejkowania lub rejestracji) oraz obsługiwanie operacji, które można wycofać.





(NULL OBJECT)

Honorowy Członek Loży Wzorców Projektowych Rusz głową!

Obiekt BrakPolecenia jest znakomitym przykładem *obiektu pustego* (ang. *null object*). Obiekt pusty jest bardzo przydatny w sytuacji, kiedy nie jesteś w stanie zwrócić żadnego innego obiektu, a jednocześnie nie chcesz składać odpowiedzialności za obsługę wartości null na klienta. Przykładowo, w aplikacji obsługującej naszego pilota nie mieliśmy żadnego znaczącego obiektu, który mógłby zostać początkowo przypisany do poszczególnych slotów, więc utworzyliśmy obiekt BrakPolecenia, który odgrywa rolę namiastki obiektu właściwego i którego metoda wykonaj() nie wykonuje po wywołaniu żadnych operacji.

Jak się niebawem przekonasz, obiekty puste są często używane w połączeniu z wzorcami projektowymi, a czasami nawet uda Ci się zobaczyć pozycję Null Object na liście wzorców projektowych.

ADAPTER

Wzorzec Adapter dokonuje konwersji interfejsu danej klasy do postaci innego interfejsu, zgodnego z oczekiwaniami klienta. Adapter pozwala na wzajemną współpracę klas, które ze względu na niekompatybilne interfejsy wcześniej nie mogły ze sobą współpracować.

Diagram klas adaptera obiektów:

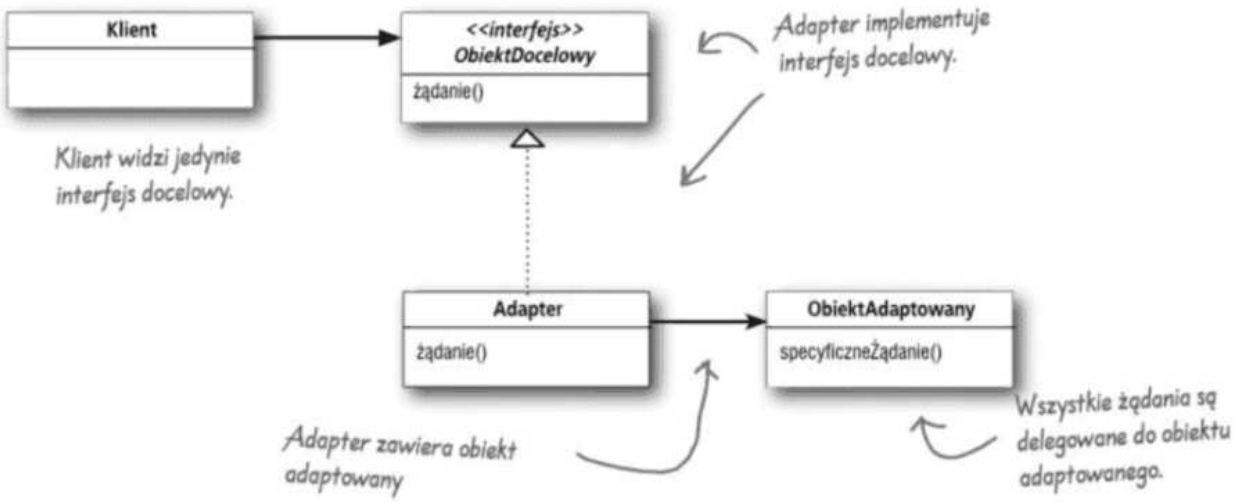
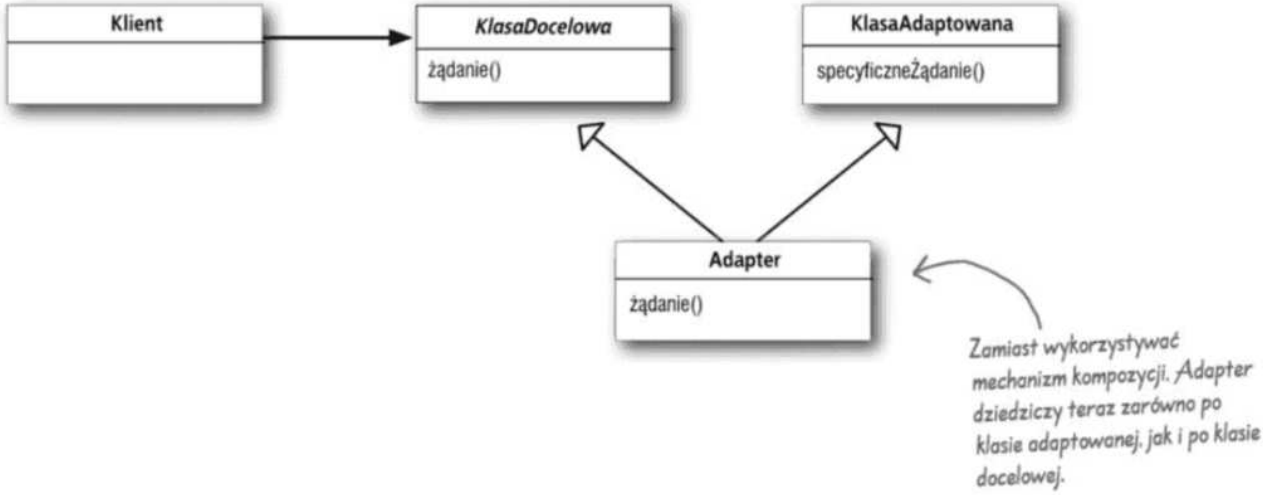
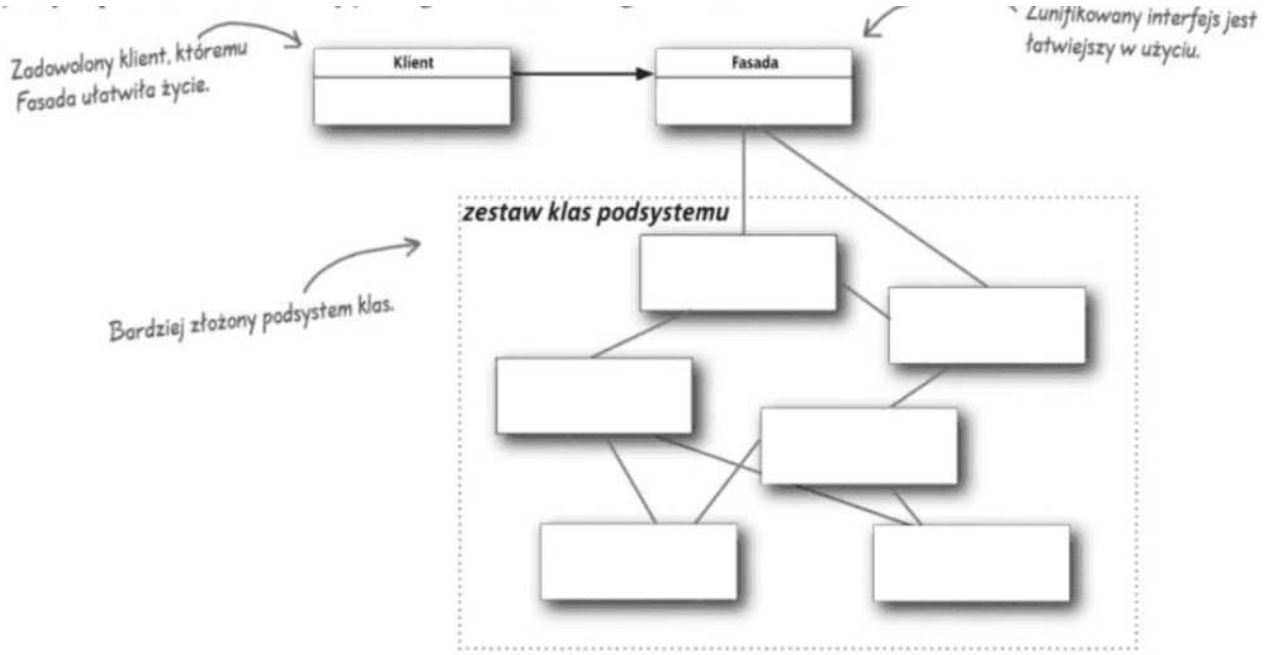


Diagram klas adaptera klas (wykorzytujący dziedziczenie wielokrotne):



FASADA

Wzorzec Fasada zapewnia jeden, zunifikowany interfejs dla całego zestawu interfejsów określonego podsystemu. Fasada tworzy nowy interfejs wysokiego poziomu, który powoduje, że korzystanie z całego podsystemu staje się zdecydowanie łatwiejsze.



Reguła projektowania

Reguła ograniczania interakcji — rozmawiaj tylko z najbliższymi przyjaciółmi.

daje kilka wskazówek: dla dowolnego obiektu powinniśmy wywoływać tylko takie metody, które należą do:

- samego obiektu
- obiektów przekazywanych jako argumenty metody
- dowolnego obiektu, który tworzy dana metoda
- dowolnego składnika danego obiektu

(inaczej Prawo Demeter)

```
public class Samochód {
    Silnik silnik;
    // tutaj umieść inne zmienne obiektowe

    public Samochód() {
        // rozruch silnika i inne tego typu sprawy...
    }

    public void uruchomSamochód(Kluczyk kluczyk) {
        Drzwi drzwi = new Drzwi();

        boolean uprawniony = kluczyk.obrótWStacyjce();

        if (uprawniony) {
            silnik.uruchom();
            aktualizacjaWskaźnikówKokpitu();
            drzwi.zamknijZamek();
        }
    }

public void aktualizacjaWskaźnikówKokpitu() {
    // aktualizacja informacji wyświetlanych na desce
rozdzielczej
    }
}
```

Tutaj mamy składnik tej klasy. Jego metody możemy wywoływać bez żadnych ograniczeń.

Tutaj tworzymy nowy obiekt. jego metody są całkowicie legalne.

Możesz wywoływać metody na obiekcie. który jest przekazywany jako argument.

Możesz wywoływać metody na składnikach obiektu.

Możesz wywoływać lokalne metody wewnątrz obiektu.

Możesz wywoływać metody na obiekcie, który sam tworzysz.

Niezgodnie z regułą

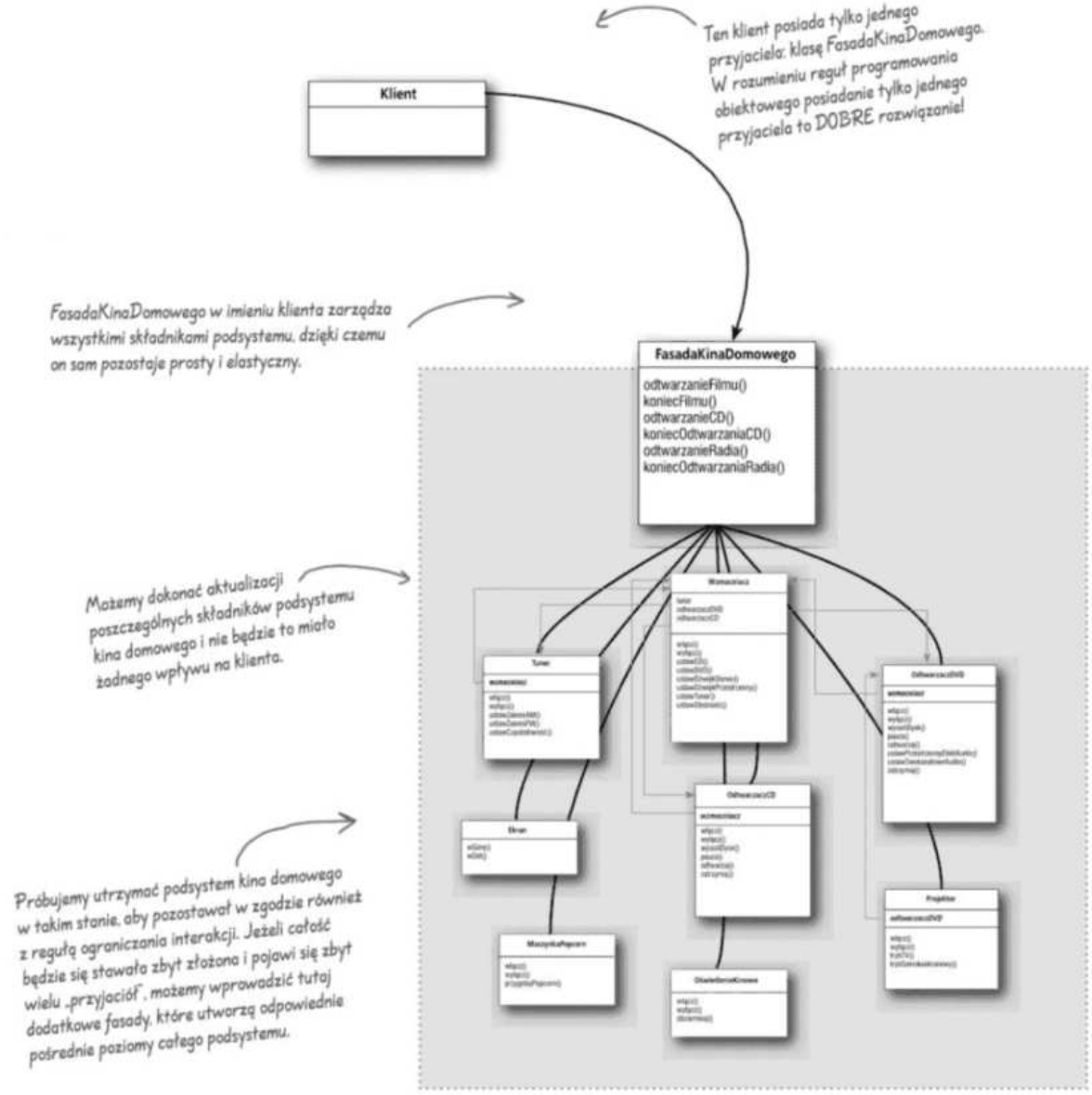
```
public float pobierzTemperaturę() {
    Termometr termometr = stacjaMeteo.pobierzTermometr();
    return termometr.pobierzTemperaturę();
}
```

Tutaj otrzymujemy ze stacji meteo obiekt reprezentujący termometr i sami wywołujemy jego metodę pobierzTemperaturę().

Zgodnie z regułą

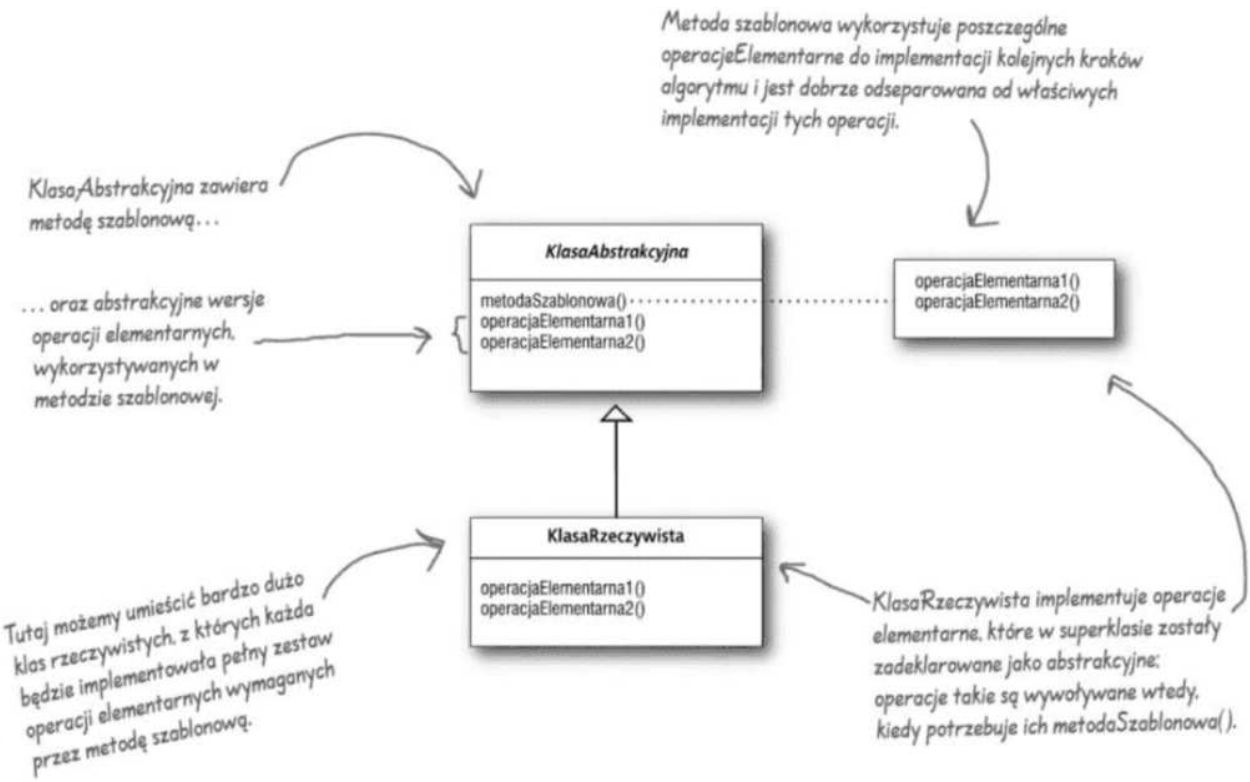
```
public float pobierzTemperaturę() {
    return stacjaMeteo.pobierzTemperaturę();
}
```

Kiedy postępujemy zgodnie z regułą, dodajemy metodę do klasy stacjaMeteo, a ona dokonuje za nas tego wywołania. W ten sposób redukujemy ilość klas, od których jesteśmy uzależnieni.



TEMPLATE METHOD (METODA SZABLONOWA)

Wzorzec Metoda Szablonowa definiuje szkielet danego algorytmu w określonej metodzie, przekazując realizację niektórych jego kroków do klas podrzędnych. Wzorzec ten pozwala klasom podrzędnym na redefiniowanie pewnych kroków algorytmu, ale jednocześnie uniemożliwia zmianę jego struktury.

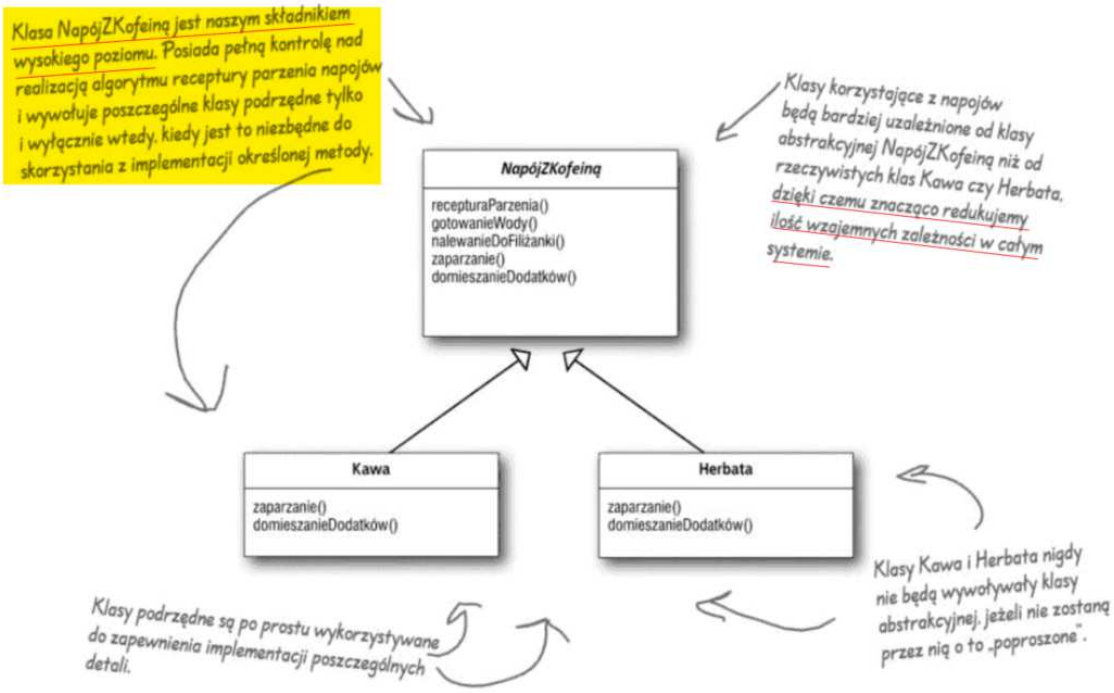


Reguła Hollywood

Nie dzwoń do nas, my zadzwonimy do Ciebie.

Reguła Hollywood pozwala nam uniknąć zjawiska niezbyt elegancko nazywanego „butwieniem drzewa zależności". Występuje ono wtedy, kiedy w systemie pojawiają się składniki wysokiego poziomu, uzależnione od określonych, innych składników wysokiego poziomu, które z kolei są uzależnione od pewnych składników bocznych poziomów, zależnych z kolei od jeszcze innych składników na niskim poziomie i tak dalej. Kiedy rozpoczyna się proces butwienia, nikt już nie jest w stanie łatwo zorientować się, jak właściwie dana aplikacja została zaprojektowana.

Wdrażając w życie regułę Hollywood, pozwalamy składnikom niskiego poziomu egzystować gdzieś w systemie, ale to składniki wysokiego poziomu decydują o tym, kiedy i w jaki sposób z nich korzystają. Innymi słowy, składniki wysokiego poziomu traktują składniki niskiego poziomu tak, jak to często ma miejsce w Hollywood: „Nie dzwoń do nas, to my zadzwonimy do Ciebie".

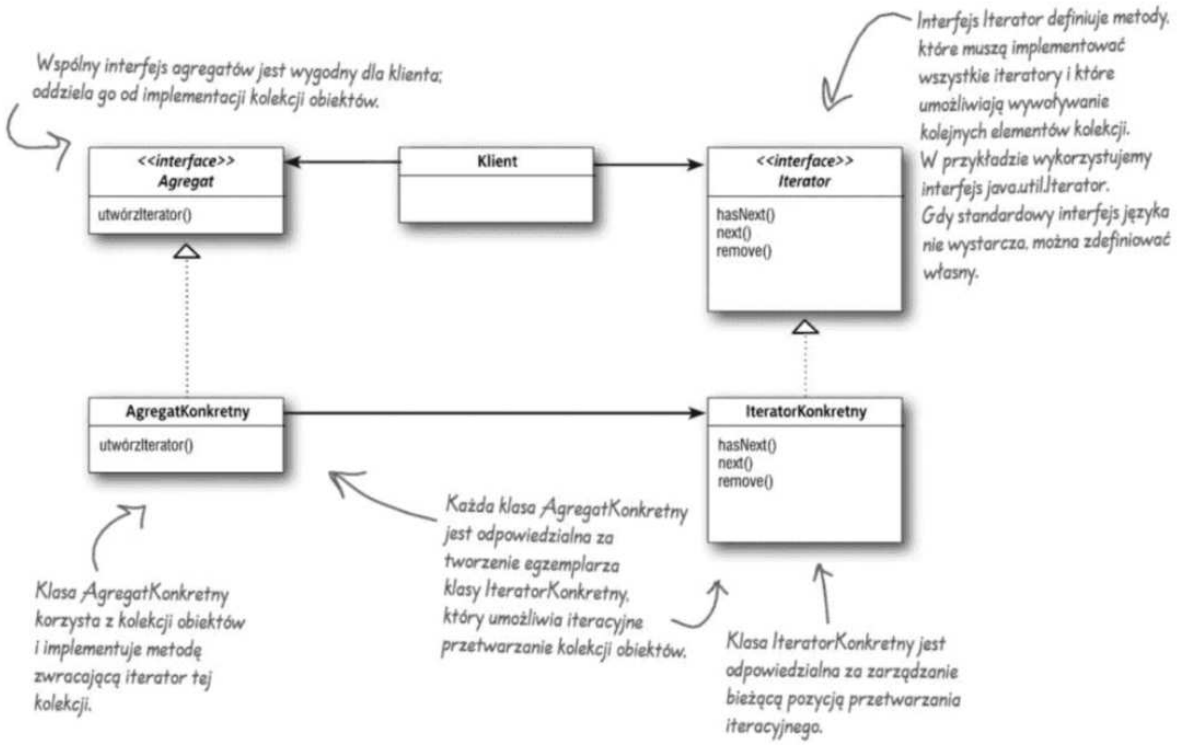


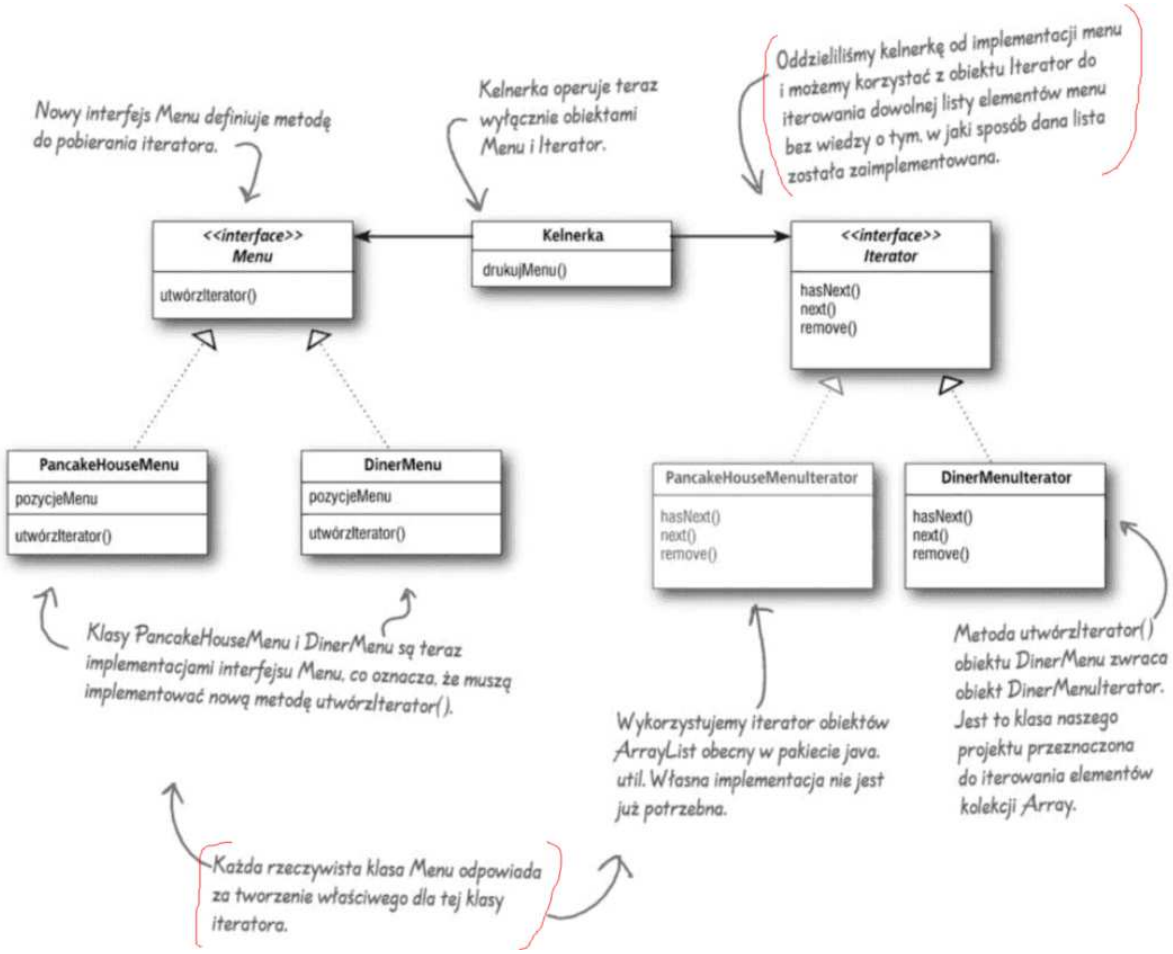
ITERATOR

Wzorzec Iterator zapewnia metodę dostępu sekwencyjnego do elementów obiektu zagregowanego bez ujawniania jego reprezentacji wewnętrznej.

Wzorzec Iterator umożliwia przechodzenie między elementami agregatu bez ujawniania ich implementacji.

Dodatkowo przenosi implementację przechodzenia między elementami z agregatu do iteratora, co upraszcza interfejs agregatu i implementację oraz prowadzi do właściwego przypisania odpowiedzialności.


Wspólny interfejs agregatów jest wygodny dla klienta; oddziela go od implementacji kolekcji obiektów.
<<interface>>
Agregat
utwórzIterator()
Klient
<<interface>>
Iterator
hasNext()
next()
remove()
Interfejs Iterator definiuje metody, które muszą implementować wszystkie iteratory i które umożliwiają wywoływanie kolejnych elementów kolekcji. W przykładzie wykorzystujemy interfejs java.util.Iterator. Gdy standardowy interfejs języka nie wystarcza, można zdefiniować własny.
AgregatKonkretny
utwórzIterator()
IteratorKonkretny
hasNext()
next()
remove()
Klasa AgregatKonkretny korzysta z kolekcji obiektów i implementuje metodę zwracającą iterator tej kolekcji.
Każda klasa AgregatKonkretny jest odpowiedzialna za tworzenie egzemplarza klasy IteratorKonkretny, który umożliwia iteracyjne przetwarzanie kolekcji obiektów.
Klasa IteratorKonkretny jest odpowiedzialna za zarządzanie bieżącą pozycją przetwarzania iteracyjnego.


Nowy interfejs Menu definiuje metodę do pobierania iteratora.
Kelnerka operuje teraz wyłącznie obiektami Menu i Iterator.
Oddzieliliśmy kelnerkę od implementacji menu i możemy korzystać z obiektu Iterator do iterowania dowolnej listy elementów menu bez wiedzy o tym, w jaki sposób dana lista została zaimplementowana.
<<interface>>
Menu
utwórzIterator()
Kelnerka
drukujMenu()
<<interface>>
Iterator
hasNext()
next()
remove()
PancakeHouseMenu
pozycjeMenu
utwórzIterator()
DinerMenu
pozycjeMenu
utwórzIterator()
PancakeHouseMenuIterator
hasNext()
next()
remove()
DinerMenuIterator
hasNext()
next()
remove()
Klasy PancakeHouseMenu i DinerMenu są teraz implementacjami interfejsu Menu, co oznacza, że muszą implementować nową metodę utwórzIterator().
Wykorzystujemy iterator obiektów ArrayList obecny w pakiecie java.util. Własna implementacja nie jest już potrzebna.
Metoda utwórzIterator() obiektu DinerMenu zwraca obiekt DinerMenuIterator. Jest to klasa naszego projektu przeznaczona do iterowania elementów kolekcji Array.
Każda rzeczywista klasa Menu odpowiada za tworzenie właściwego dla tej klasy iteratora.

Reguła projektowania

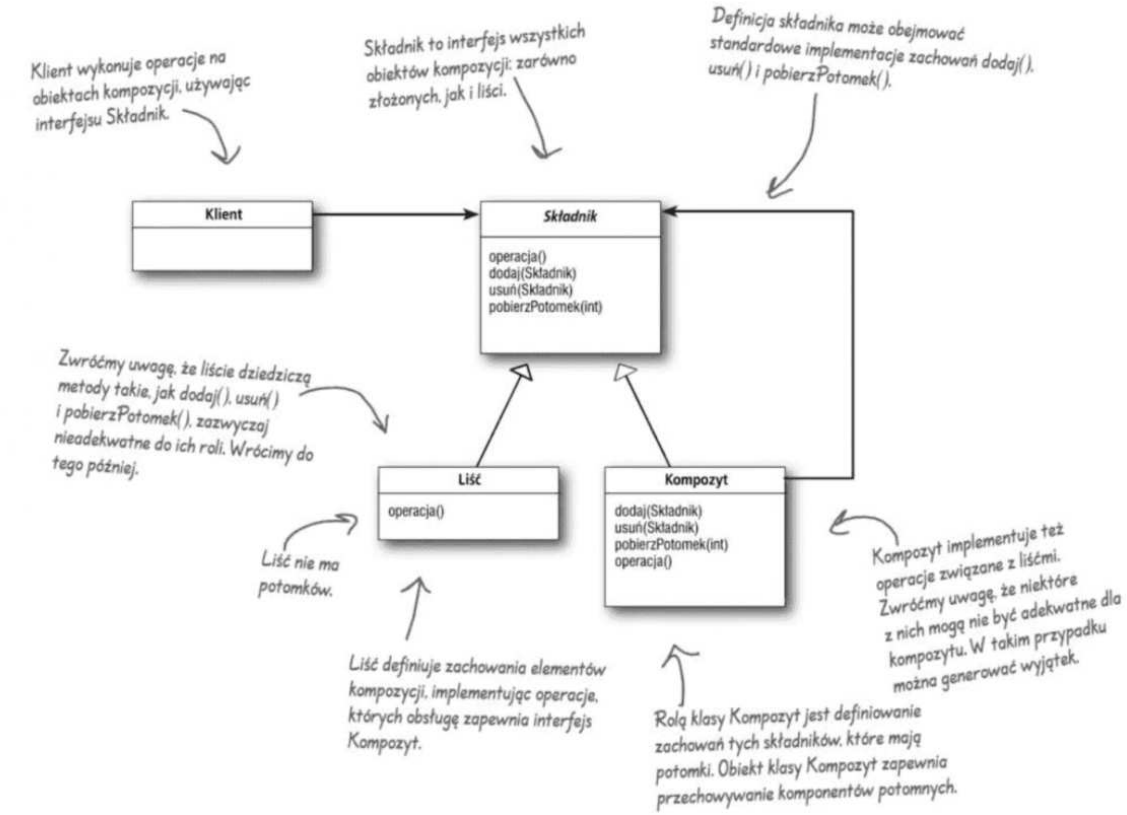
Klasa powinna mieć tylko jeden powód do zmian.

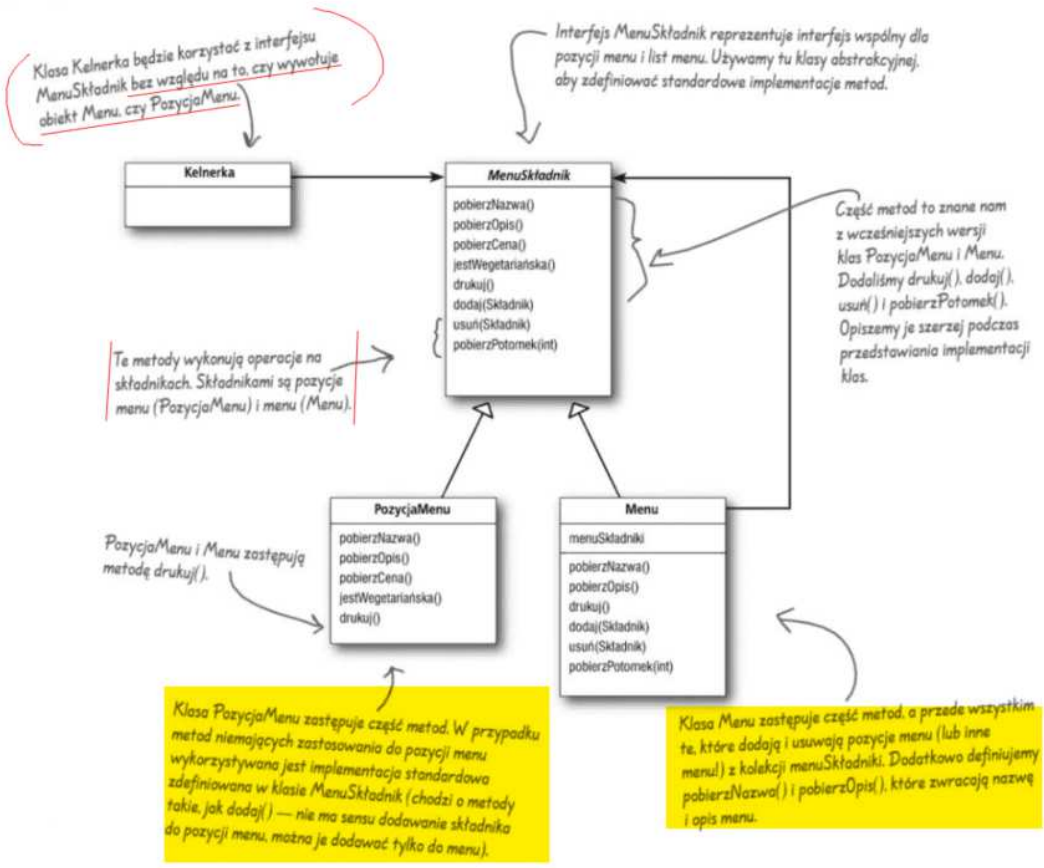
Każdy zakres odpowiedzialności klasy to obszar potencjalnych zmian. Większy zakres odpowiedzialności poszerza pole do zmian. Każda klasa powinna mieć jeden obszar odpowiedzialności.

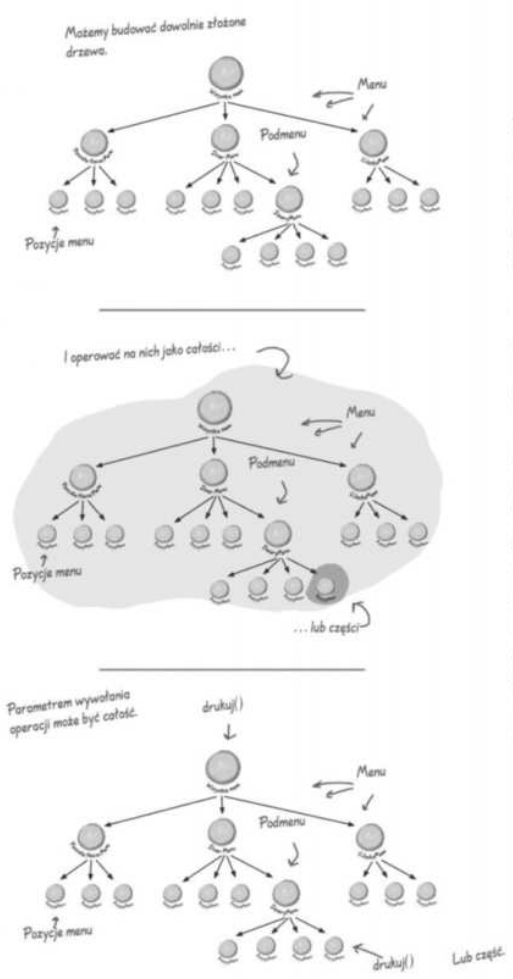
KOMPOZYT

Wzorzec Kompozyt (ang. composite, obiekt złożony) pozwala łączyć obiekty w struktury drzewiaste, które reprezentują hierarchie część-całość. Jego zastosowanie pozwala klientom jednolicie obsługiwać zarówno pojedyncze obiekty, jak i ich kompozycje.

Diagram klas wzorca Kompozyt





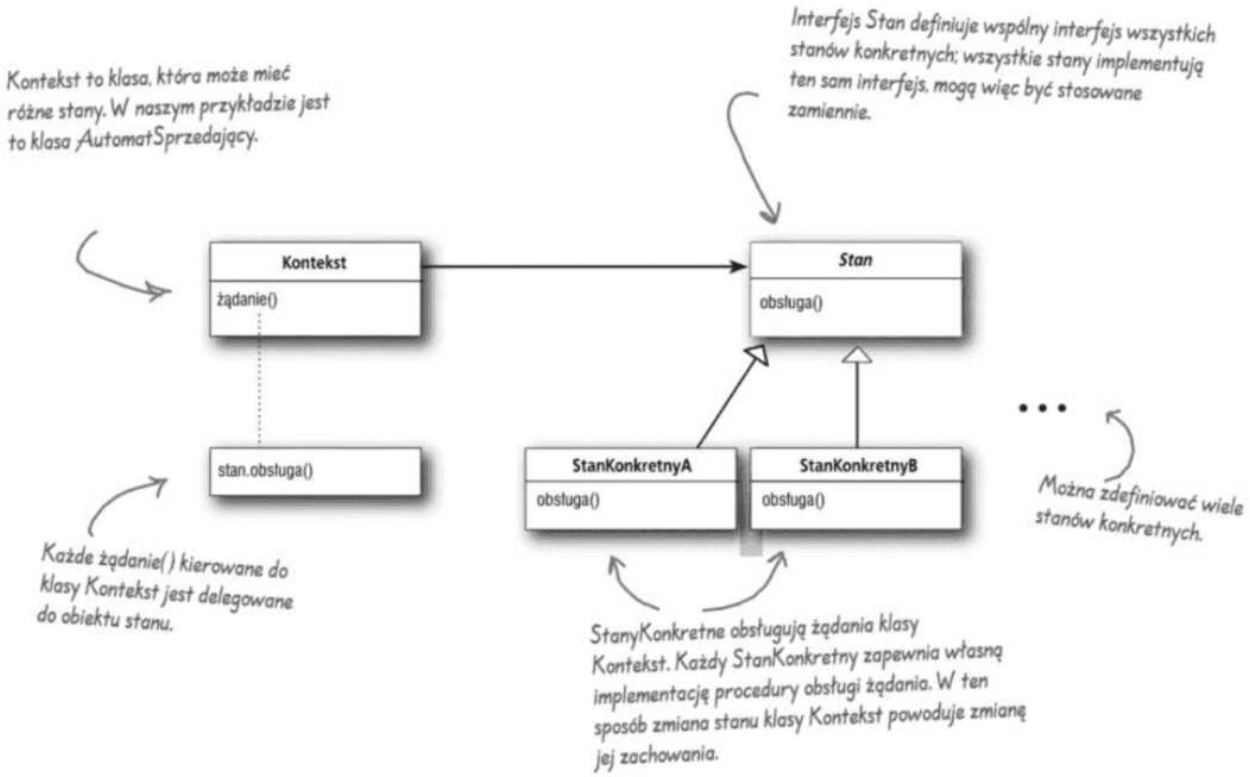


Wzorzec Kompozyt umożliwia budowanie struktur obiektów będących drzewami, których węzłami są zarówno kompozycje obiektów, jak i pojedyncze elementy.

Używając takiej struktury kompozytowej, możemy wykonywać operacje zarówno na całości, jak i na pojedynczych obiektach. Innymi słowy, w większości przypadków możemy zignorować różnice między kompozycjami obiektów a indywidualnymi obiektami.

STAN

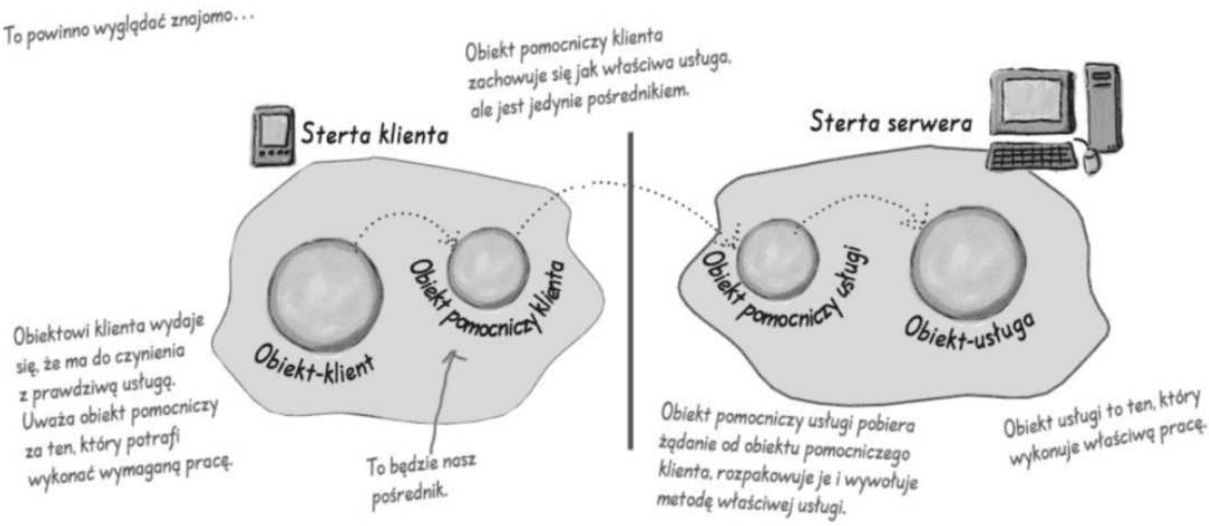
> ***Wzorzec Stan*** *umożliwia obiektowi zmianę zachowania wraz ze zmianą jego wewnętrznego stanu. Po takiej zmianie funkcjonuje on jak inna klasa.*

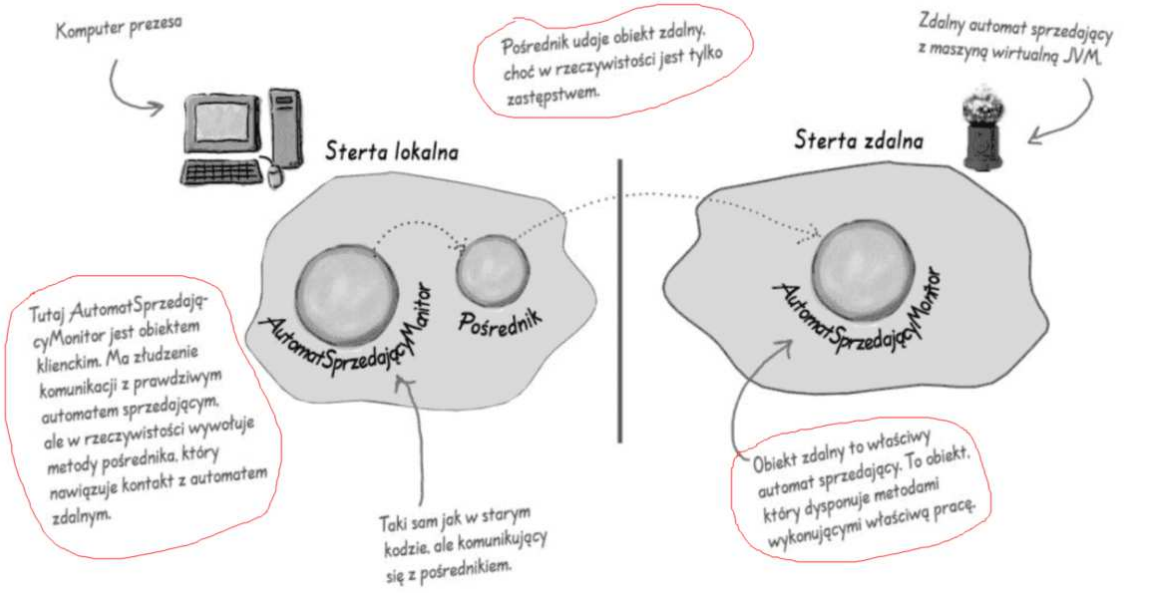


PROXY

> **Wzorzec Proxy** prowadzi do utworzenia obiektu pośredniczącego, który kontroluje dostęp do innego obiektu.

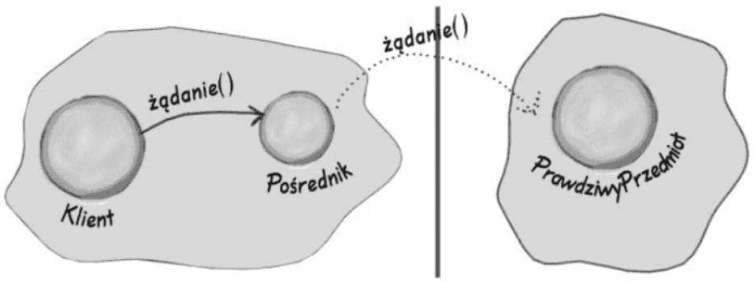
Wzorca Proxy używamy do stworzenia obiektu zastępczego, który kontroluje dostęp do innego obiektu, kiedy mamy do czynienia z takim obiektem zdalnym, którego stworzenie wiąże się z dużym kosztem lub który wymaga zabezpieczeń.


To powinno wyglądać znajomo...
Obiekt pomocniczy klienta zachowuje się jak właściwa usługa, ale jest jedynie pośrednikiem.
Sterta klienta
Sterta serwera
Obiektowi klienta wydaje się, że ma do czynienia z prawdziwą usługą. Uważa obiekt pomocniczy za ten, który potrafi wykonać wymaganą pracę.
Obiekt-klient
Obiekt pomocniczy klienta
Obiekt pomocniczy usługi
Obiekt-usługa
To będzie nasz pośrednik.
Obiekt pomocniczy usługi pobiera żądanie od obiektu pomocniczego klienta, rozpakowuje je i wywołuje metodę właściwej usługi.
Obiekt usługi to ten, który wykonuje właściwą pracę.

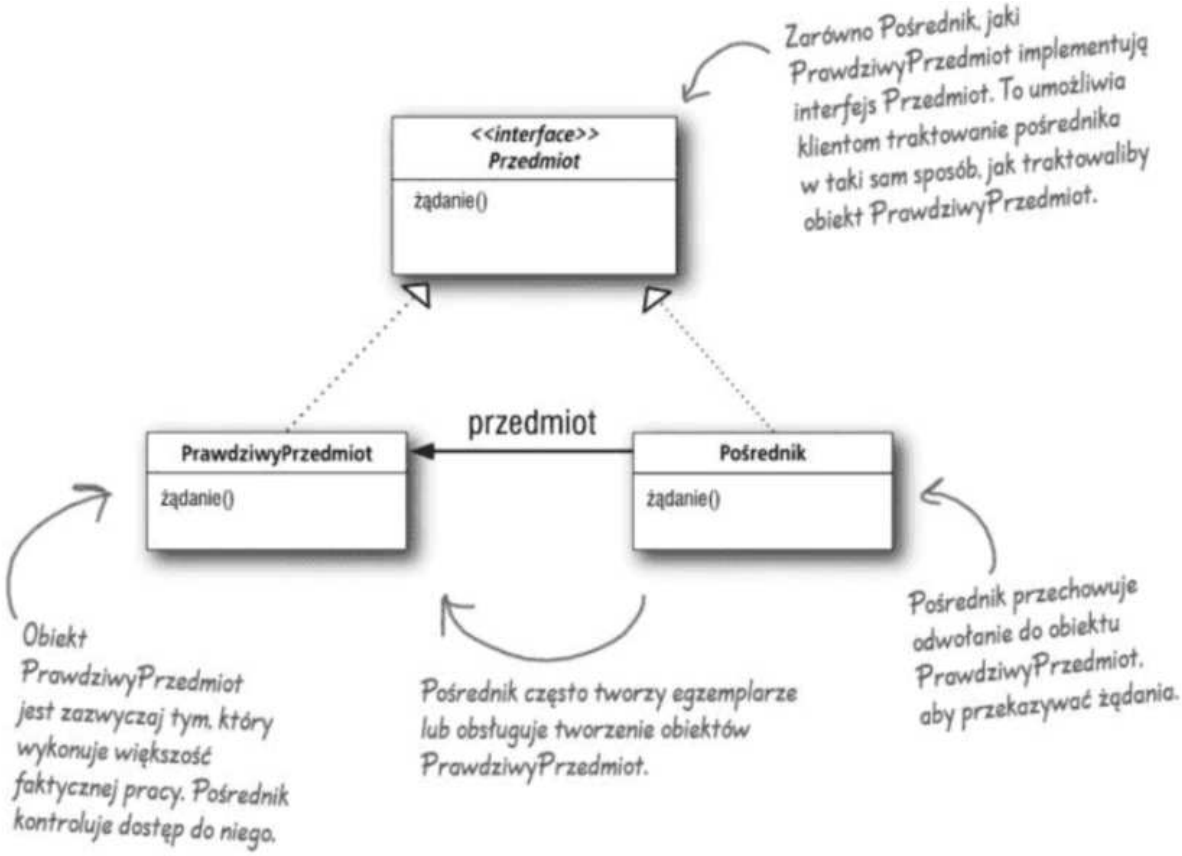

Komputer prezesa
Pośrednik udaje obiekt zdalny, choć w rzeczywistości jest tylko zastępstwem.
Zdalny automat sprzedający z maszyną wirtualną JVM
Sterta lokalna
Sterta zdalna
Tutaj AutomatSprzedającyMonitor jest obiektem klienckim. Ma złudzenie komunikacji z prawdziwym automatem sprzedającym, ale w rzeczywistości wywołuje metody pośrednika, który nawiązuje kontakt z automatem zdalnym.
AutomatSprzedającyMonitor
Pośrednik
AutomatSprzedającyMonitor
Taki sam jak w starym kodzie, ale komunikujący się z pośrednikiem.
Obiekt zdalny to właściwy automat sprzedający. To obiekt, który dysponuje metodami wykonującymi właściwą pracę.

Remote Proxy — pośrednik zdalny

W schemacie Remote Proxy pośrednik jest lokalnym przedstawicielem obiektu pracującego w innej maszynie wirtualnej. Wywołanie metody pośrednika prowadzi do wywołania metody obiektu zdalnego. Wynik jest zwracany poprzez sieć i pośrednik do klienta.

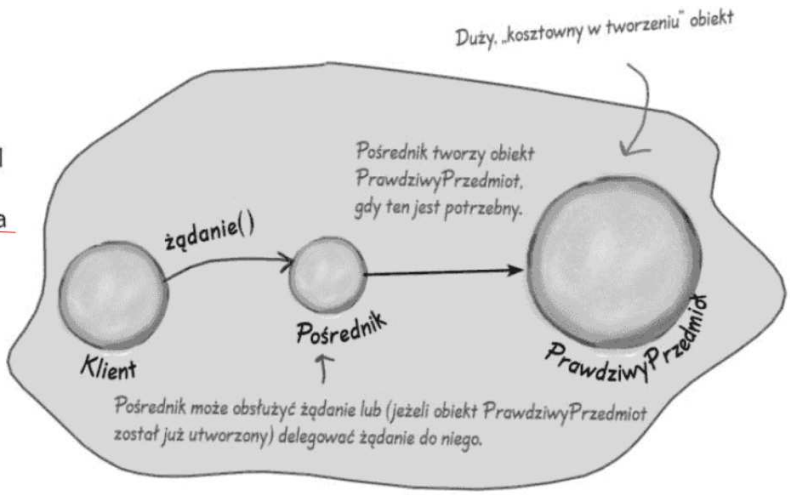


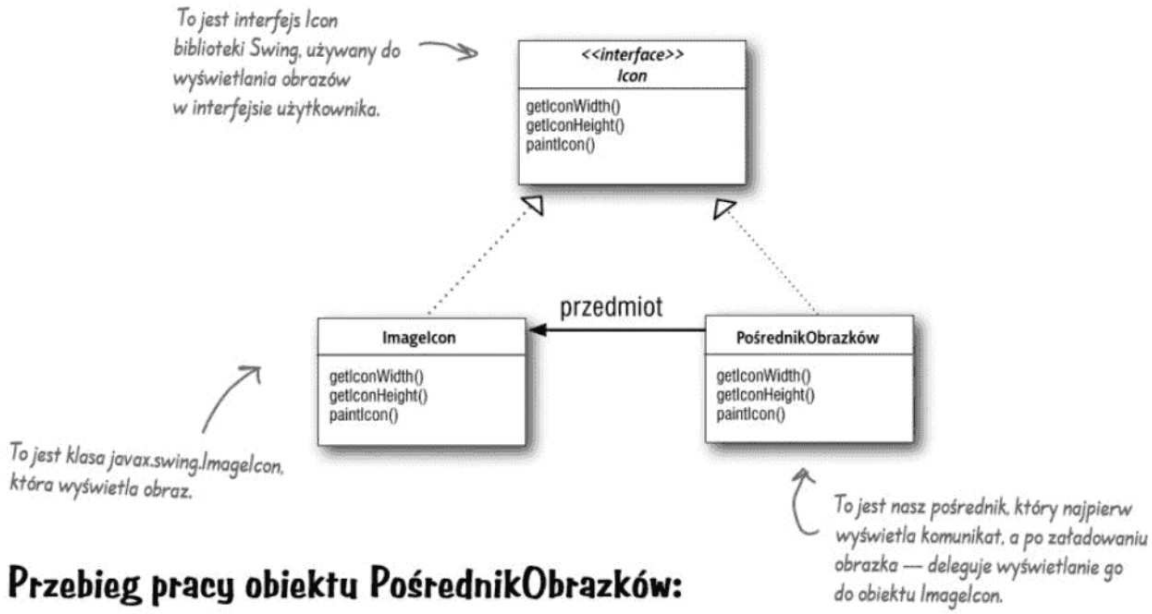
Ten diagram znamy już dość dobrze...



Virtual Proxy — pośrednik wirtualny

Pośrednik wirtualny to przedstawiciel obiektu, którego utworzenie jest w pewien sposób kosztowne. Pozwala to opóźnić utworzenie tego obiektu do momentu, gdy staje się on faktycznie niezbędny. Pośrednik zastępuje właściwy obiekt przed utworzeniem egzemplarza i w trakcie jego tworzenia. Później deleguje żądania bezpośrednio do obiektu PrawdziwyPrzedmiot.

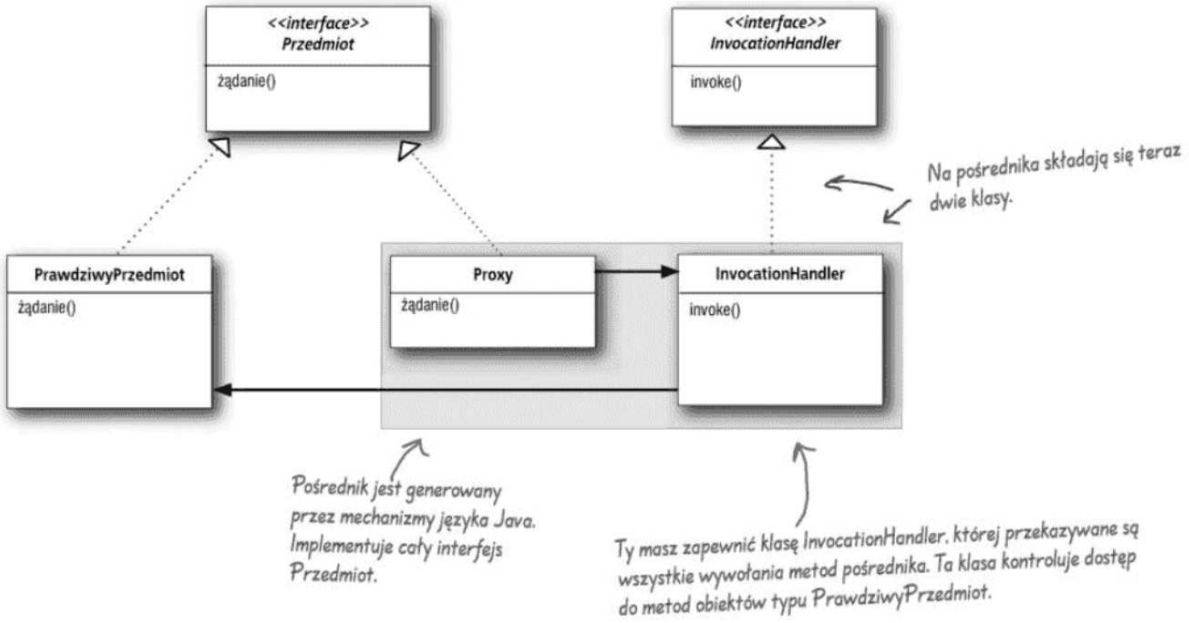




Przebieg pracy obiektu PośrednikObrazków:

https://www.youtube.com/watch?v=NwaabHqPHeM&t=1946s

pośrednik dynamiczny (ang. *dynamic proxy*).



COMPOUND PATTERN (łączenie)

Rozpoczęliśmy od kilku obiektów Kwacząca...

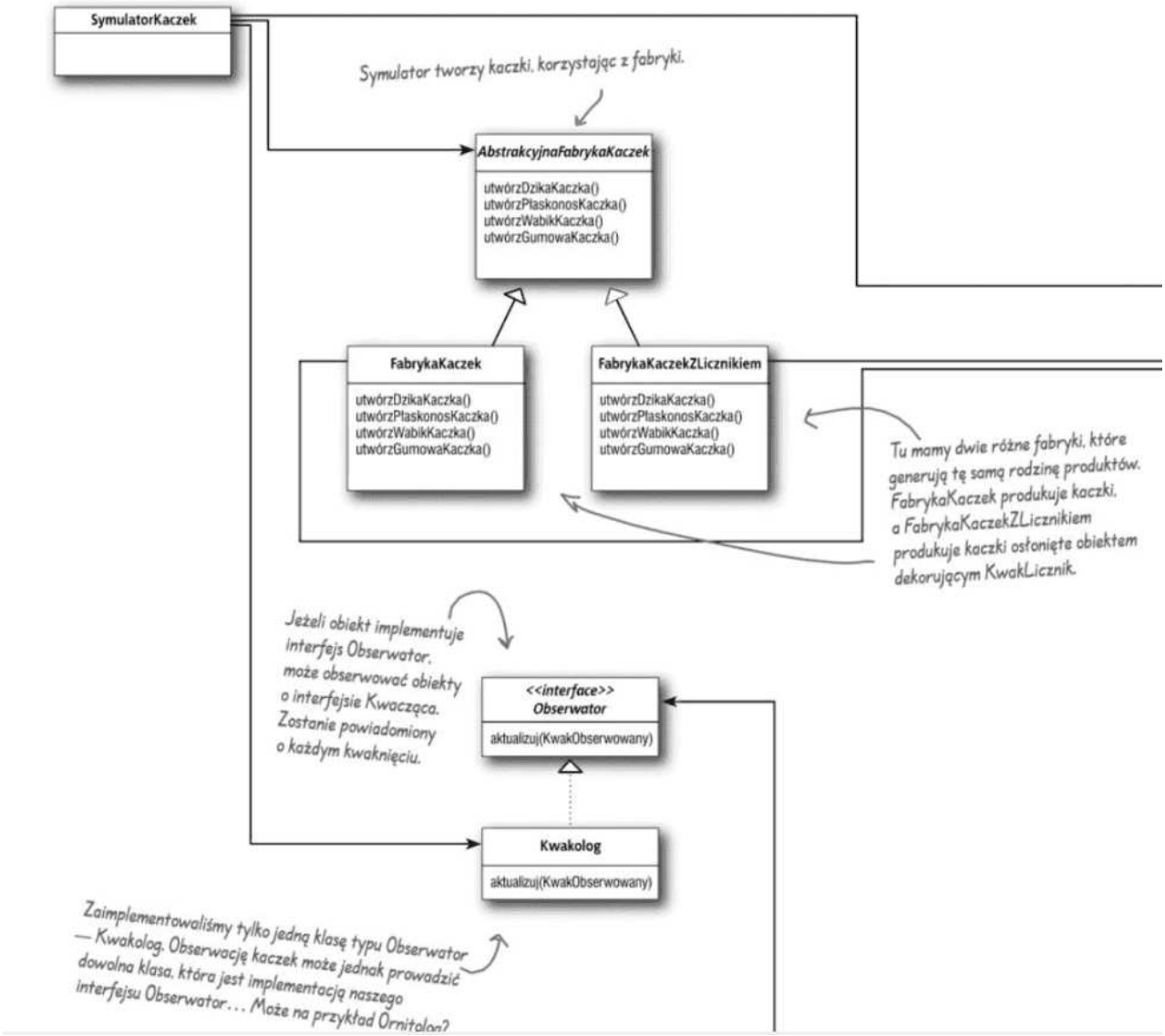
Potem pojawiła się Gęś i też chciała być Kwacząca. Użyliśmy wzorca *Adapter*, aby zaadaptować ją do interfejsu Kwacząca. Dzięki temu mogliśmy wywoływać metodę kwacz() osłony gęsi (adaptera) i uzyskiwaliśmy gęganie!

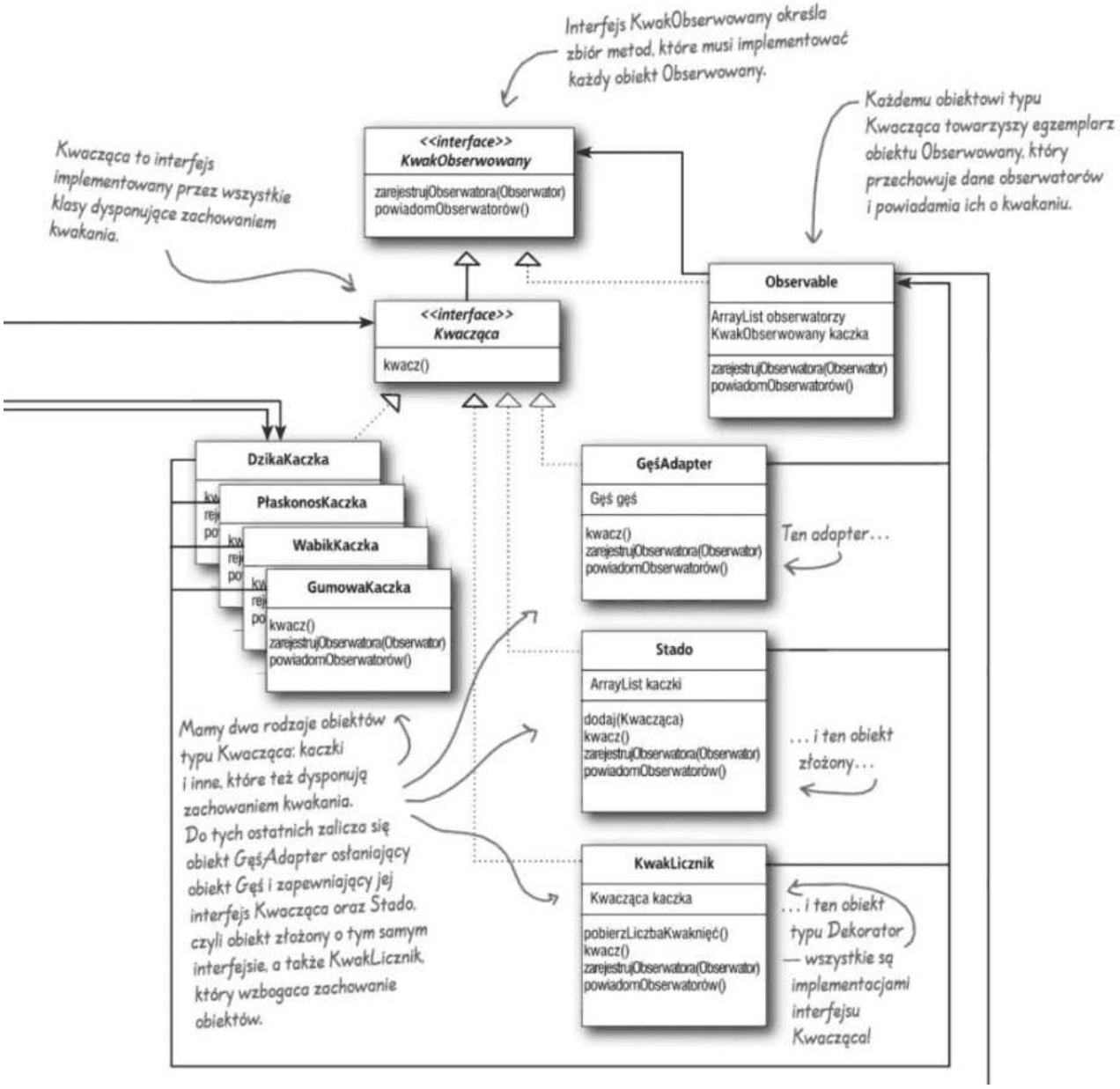
Potem kwakolodzy zdecydowali, że wymagają liczenia kwaknięć. Użyliśmy więc wzorca *Dekorator* i stworzyliśmy klasę KwakLicznik, która zlicza wywołania metody kwacz() i deleguje jej wykonanie do osłanianego obiektu Kwacząca.

Kwakolodzy ciągle zapominali o dodawaniu osłony zliczającej kwaknięcia. Użyliśmy więc wzorca *Fabryka Abstrakcyjna*, aby ukryć operacje tworzenia ptaków z osłonami. Po takiej zmianie, jeśli kiedykolwiek potrzebna jest kaczka, pobieramy ją z fabryki. Fabryka zwraca obiekt odpowiednio osłonięty. Wciąż możemy korzystać z innej fabryki, zwracającej kaczki, których kwaknięcia nie są zliczane.

Pojawiły się problemy w zarządzaniu wszystkimi kaczkami, gęsiami i obiektami Kwacząca. Użyliśmy więc wzorca *Kompozyt*, aby połączyć ptaki w stada. Ten wzorzec umożliwił dodatkowo utworzenie rodzin kaczek, czyli mniejszych grup w stadzie. Użyliśmy też wzorca *Iterator*, pobierając standardowy iterator kolekcji ArrayList.

Kwakolodzy wymagali też powiadamiania o każdym kwaknięciu. Użyliśmy wzorca *Obserwator*, aby umożliwić rejestrowanie obiektów Kwakolog (jako obiektów Obserwator) w obiektach Kwacząca. Dzięki temu Kwakolog jest powiadamiany o każdym kwaknięciu. Ponownie skorzystaliśmy z iteratora. Kwakolog może również skorzystać z możliwości jednorazowej rejestracji w obiekcie złożonym (stadzie).


SymulatorKaczek
Symulator tworzy kaczki, korzystając z fabryki.
AbstrakcyjnaFabrykaKaczek
utwórzDzikaKaczka()
utwórzPłaskonosKaczka()
utwórzWabikKaczka()
utwórzGumowaKaczka()
FabrykaKaczek
utwórzDzikaKaczka()
utwórzPłaskonosKaczka()
utwórzWabikKaczka()
utwórzGumowaKaczka()
FabrykaKaczekZLicznikiem
utwórzDzikaKaczka()
utwórzPłaskonosKaczka()
utwórzWabikKaczka()
utwórzGumowaKaczka()
Tu mamy dwie różne fabryki, które generują tę samą rodzinę produktów. FabrykaKaczek produkuje kaczki, a FabrykaKaczekZLicznikiem produkuje kaczki osłonięte obiektem dekorującym KwakLicznik.
Jeżeli obiekt implementuje interfejs Obserwator, może obserwować obiekty o interfejsie Kwaczące. Zostanie powiadomiony o każdym kwaknięciu.
<<interface>>
Obserwator
aktualizuj(KwakObserwowany)
Kwakolog
aktualizuj(KwakObserwowany)
Zaimplementowaliśmy tylko jedną klasę typu Obserwator — Kwakolog. Obserwację kaczek może jednak prowadzić dowolna klasa, która jest implementacją naszego interfejsu Obserwator... Może na przykład Ornitolog?



Wzorzec projektowy to rozwiązanie problemu w pewnym kontekście.

Przykład: masz kolekcję obiektów.

Kontekst to powtarzająca się sytuacja, w której można wzorzec zastosować.

Problem to cel, który chcemy osiągnąć w danym kontekście; w tym pojęciu mieszczą się również narzucane przez kontekst ograniczenia.

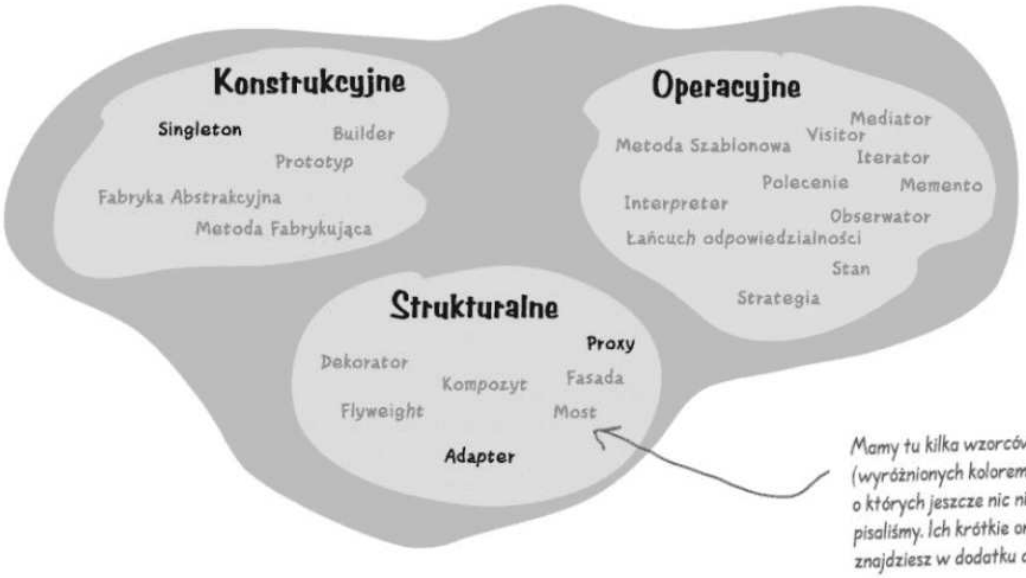
Potrzebujesz iterowania obiektów bez ujawniania implementacji tej kolekcji.

Rozwiązanie to jest to, czego szukamy: uogólniony projekt, który może zastosować każdy, kto dąży do tego samego celu, przy tych samych ograniczeniach.

Hermetyzujesz iterowanie w osobnej klasie.

Wzorce konstrukcyjne *dotyczą interakcji klas i obiektów oraz podziału odpowiedzialności.*

Wzorce operacyjne *dotyczą interakcji klas i obiektów oraz podziału odpowiedzialności.*



Mamy tu kilka wzorców (wyróżnionych kolorem szarym), o których jeszcze nic nie pisaliśmy. Ich krótkie omówienie znajdziesz w dodatku do książki.

Wzorce strukturalne *pozwalają łączyć klasy i obiekty w większe struktury.*

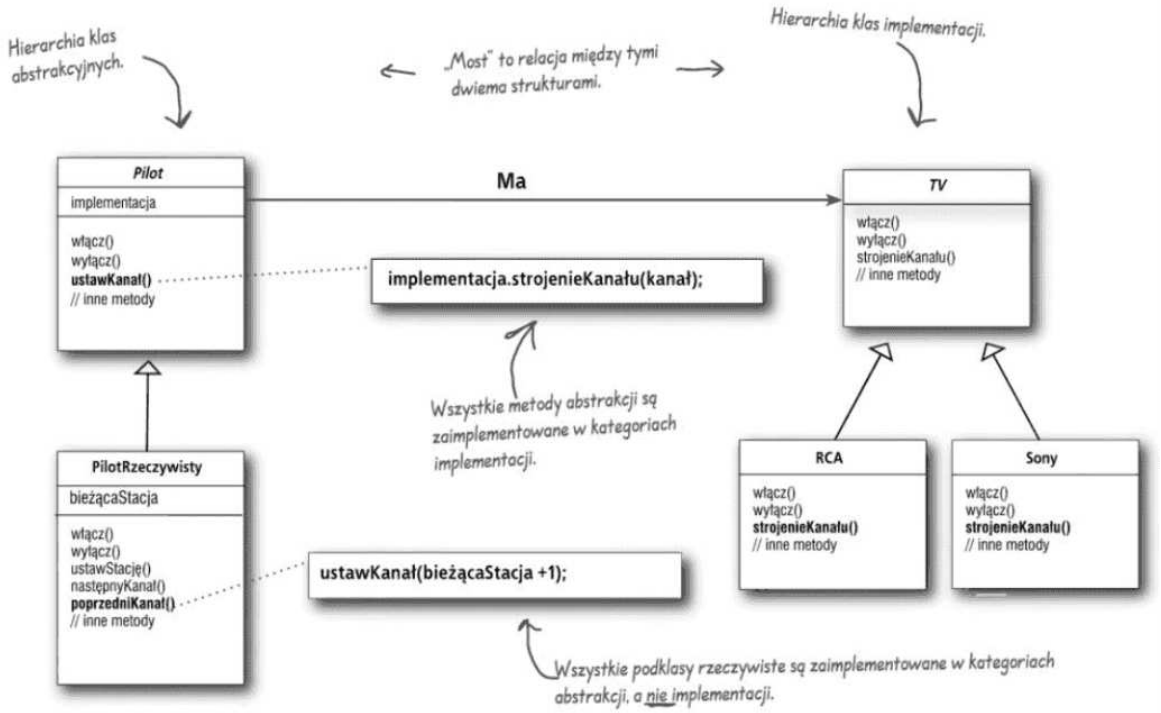
Wzorce klas *opisują, w jaki sposób dziedziczenie definiuje związki między klasami. Relacje wzorców klas ustanawiane są w czasie kompilowania kodu.*

Wzorce obiektów *opisują związki między obiektami i są definiowane głównie przez kompozycję. Relacje opisane we wzorcach obiektów powstają najczęściej w czasie wykonywania programu. Są bardziej dynamiczne i elastyczne.*

Zwróć uwagę, że jest znacznie więcej wzorców obiektów niż klas!

MOST

Wzorzec Most umożliwia różnicowanie implementacji i abstrakcji poprzez umieszczenie obu elementów w osobnych hierarchiach klas.



Masz teraz dwie hierarchie, jedną dla pilotów i drugą dla tych implementacji telewizorów, które są specyficzne dla modelu. Most umożliwia modyfikowanie każdej ze stron niezależnie.

Zalety wzorca Most

- Separuje implementację, usuwając jej trwałe powiązanie z interfejsem.
- Abstrakcja i implementacja mogą być rozszerzane niezależnie.
- Zmiany w klasach rzeczywistych abstrakcji nie wpływają na klienta.

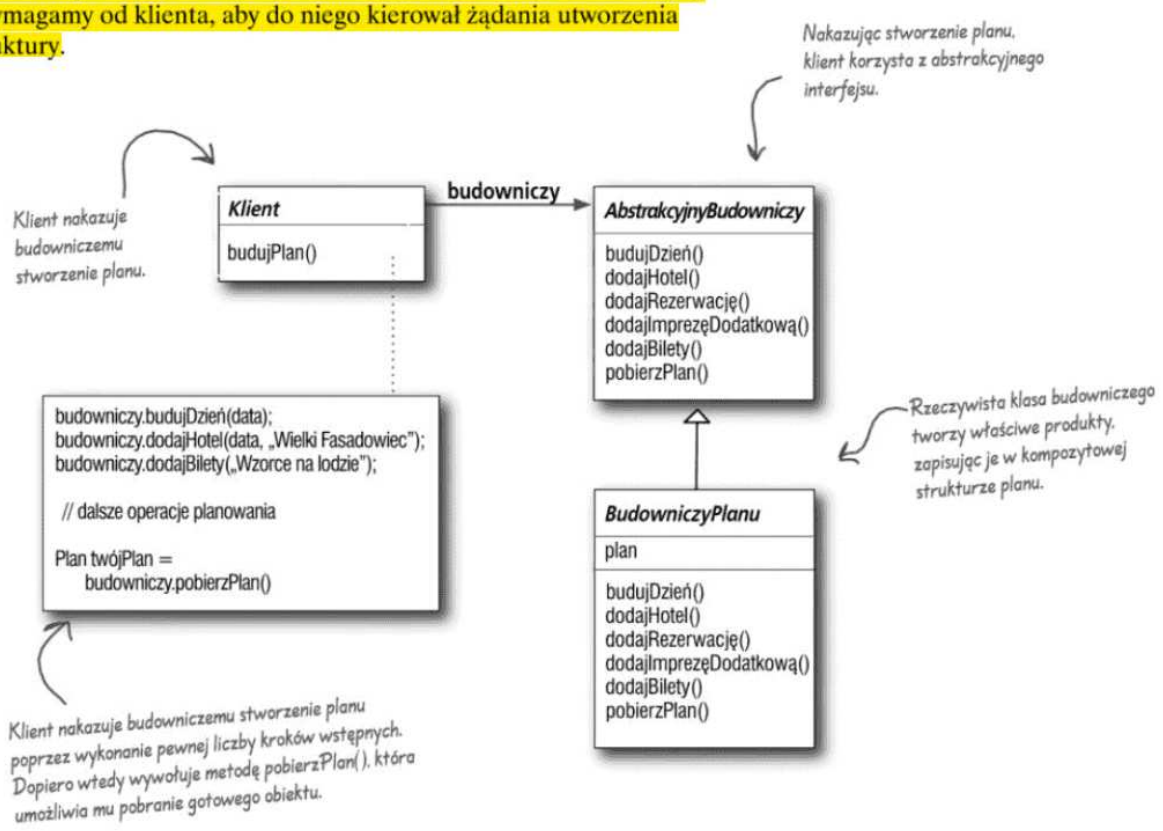
Zastosowania i wady wzorca Most

- Przydatny w systemach graficznych i okienkowych, które muszą pracować na różnych platformach.
- Przydatny w każdej sytuacji, gdy oczekujemy innych zmian w interfejsie i innych w implementacji.
- Zwiększa złożoność.

BUILDER

Wzorca Builder używamy do hermetyzowania tworzenia produktu i w celu umożliwienia jego wieloetapowego inicjowania.

klientem. Podobny pomysł pojawia się w tym przypadku: hermetyzujemy tworzenie planu wycieczki w obiekcie (będzie to nasz „budowniczy") i wymagamy od klienta, aby do niego kierował żądania utworzenia struktury.



Zalety wzorca Builder

- Hermetyzuje operacje niezbędne do stworzenia złożonego obiektu.
- Umożliwia tworzenie obiektów w procedurze wielokrokowej i nie narzuca tej procedury (w przeciwieństwie do jednokrokowych wzorców z grupy Fabryka).
- Ukrywa wewnętrzną reprezentację produktu przed klientem.
- Implementacje produktów mogą być wymieniane, ponieważ klient korzysta wyłącznie z abstrakcyjnego interfejsu.

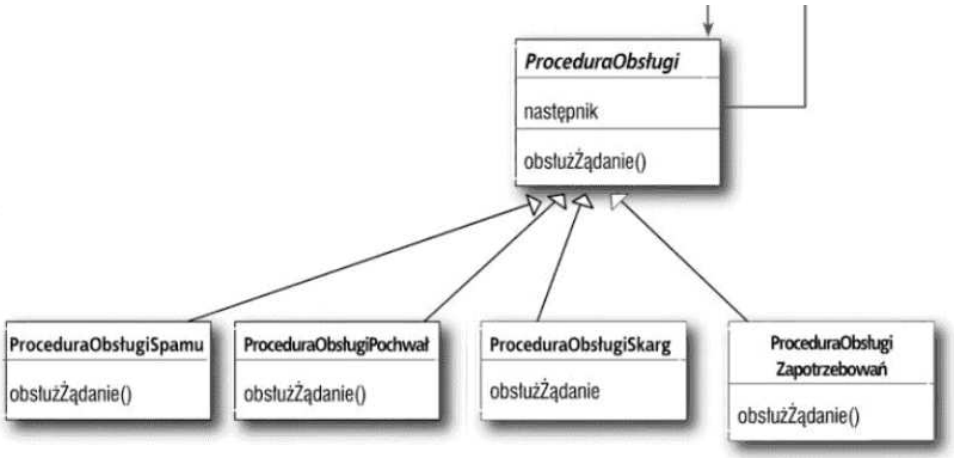
Zastosowania i wady wzorca Builder

- Często stosowany do budowania struktur kompozytowych.
- Tworzenie obiektów wymaga od klienta większej wiedzy z zakresu dziedziny problemu niż w przypadku stosowania wzorca Fabryka.

ŁAŃCUCH ODPOWIEDZIALNOŚCI (CHAIN OF RESPONSIBILITY)

Wzorzec Łańcuch Odpowiedzialności stosujemy wtedy, gdy kolejne żądania mają być obsługiwane przez różne obiekty.

Każdy obiekt w łańcuchu działa jak procedura obsługi żądania i ma pewien następnik. Jeżeli dany obiekt potrafi obsłużyć żądanie, przetwarzanie zostaje zakończone. W innych przypadkach przekazuje żądanie następnikowi.



Po odebraniu list jest przekazywany do pierwszej procedury obsługi — tej, która radzi sobie ze spamem. Jeżeli nie pasuje ona do złożonego żądania, jego przetwarzanie jest przekazywane do procedury obsługi pochwał. I tak dalej…

List nie zostanie odpowiednio potraktowany, jeżeli żaden z obiektów nie zapewni jego obsługi. Jednak ostatni element łańcucha może inicjować pewnego rodzaju przetwarzanie niezależnie od charakterystyki żądania.

Każdy list jest przekazywany do pierwszej procedury.

Zalety wzorca Łańcuch Odpowiedzialności

- Separuje nadawcę żądania od jego odbiorców.
- Upraszcza obiekt, ponieważ nie wymaga utrzymywania informacji o strukturze łańcucha i operowania bezpośrednimi odwołaniami do jego elementów.
- Umożliwia dynamiczne dodawanie i usuwanie procedur obsługi poprzez zmianę elementów lub kolejności łańcucha.

Zastosowania i wady wzorca Łańcuch Odpowiedzialności...

- Powszechnie używany w systemach okienkowych do obsługi zdarzeń takich, jak kliknięcia myszy i wciśnięcia klawiszy.
- Wykonanie żądania nie jest gwarantowane. Jeżeli żaden obiekt nie zapewni jego obsługi, żądanie opuści łańcuch, nie inicjując żadnych operacji (co może być tak wadą, jak i zaletą).
- Obserwowanie pracy łańcucha w czasie wykonywania aplikacji oraz jego debugowanie mogą być trudne.

FLYWEIGHT

Wzorzec Flyweight stosujemy wtedy, gdy jeden egzemplarz klasy może zostać wykorzystany do stworzenia wielu „egzemplarzy wirtualnych”.

Co by się stało, jeżeli mógłbyś wprowadzić taką zmianę w projekcie, aby w miejsce tysięcy obiektów Drzewo tworzyć tylko jeden taki obiekt? Niezbędnym uzupełnieniem będzie obiekt kliencki, który przechowuje stan WSZYSTKICH drzew. To właśnie wzorzec Flyweight!

Zalety wzorca Flyweight

- Ogranicza liczbę egzemplarzy obiektów w czasie wykonywania programu, oszczędzając w ten sposób pamięć aplikacji.
- Skupia składowanie danych stanu „wirtualnych” obiektów w jednej lokalizacji.

Zastosowania i wady wzorca Flyweight

- Wzorzec Flyweight stosuje się tam, gdzie klasa ma wiele egzemplarzy, a wszystkie mogą być sterowane w identyczny sposób.
- Wadą wzorca Flyweight jest to, że po jego zaimplementowaniu pojedyncze, logiczne egzemplarze klasy nie będą mogły dysponować zachowaniami niezależnymi od pozostałych egzemplarzy.

INTERPRETER

Wzorca Interpreter używamy do budowania interpretera pewnego języka.

Teraz, mając w pamięci zasady budowania gramatyk formalnych z jednego z kursów wprowadzających do programowania, rozpisujesz gramatykę tego języka:

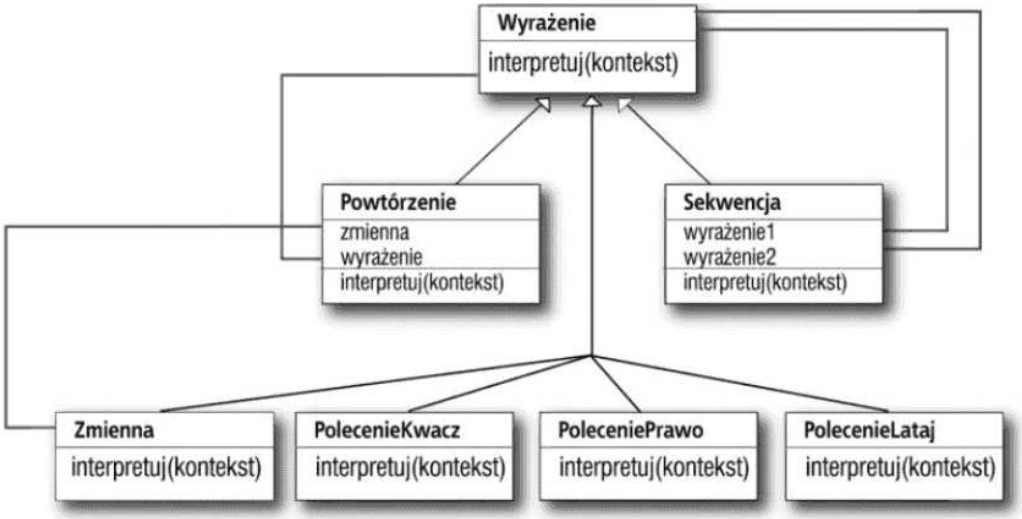
Program to wyrażenie składające się z sekwencji poleceń i powtórzeń, czyli instrukcji takich jak „dopóki" („while").

```
wyrażenie ::= <polecenie> | <sekwencja> | <powtórzenie>
sekwencja ::= <wyrażenie> ';' <wyrażenie>
polecenie ::= prawo | kwacz | lataj
powtórzenie ::= dopóki '(' <zmienna> ')' <wyrażenie>
zmienna ::= [A-z,a-z]+
```

Sekwencja to zbiór wyrażeń rozdzielanych średnikami.

Mamy trzy polecenia: prawo, kwacz i lataj.

Instrukcja „dopóki" obejmuje zmienną, która określa warunek, oraz wyrażenie.



Zalety wzorca Interpreter

- Reprezentowanie każdej reguły gramatycznej jako klasy znacznie ułatwia implementowanie języka.
- Ponieważ gramatykę reprezentują klasy, język można łatwo modyfikować i rozszerzać.
- Dodając nowe metody klas, możesz wprowadzać nowe zachowania wykraczające poza samą interpretację, jak ładne wydruki lub wyszukane sprawdzanie poprawności programu.

Zastosowania i wady wzorca Interpreter

- Wzorca Interpreter używamy tam, gdzie pojawia się potrzeba zaimplementowania prostego języka.
- Wzorzec jest odpowiedni tam, gdzie gramatyka jest prosta i przejrzystość rozwiązania jest ważniejsza od wydajności.
- Używany do obsługi języków skryptowych i innych języków programowania.
- Wzorzec może stać się nieporęczny, gdy liczba reguł gramatycznych jest znaczna. W takich przypadkach generator kompilatorów będzie lepszym rozwiązaniem.

MEDIATOR

Wzorca Mediator używamy do skupiania złożonych procedur komunikacji i sterowania w środowisku powiązanych obiektów.

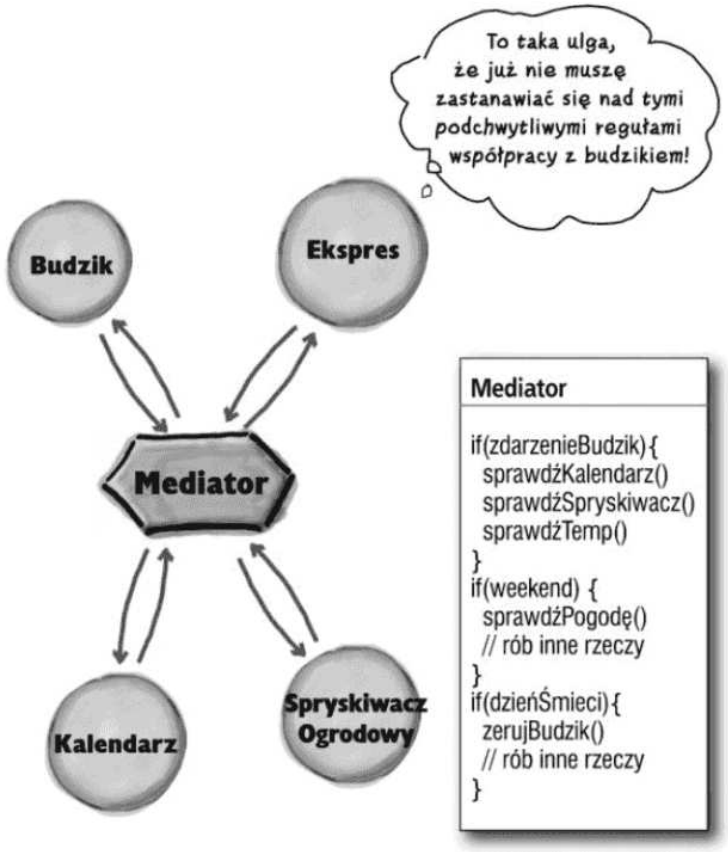
Mediator w akcji

Po wprowadzeniu do systemu wzorca Mediator wszystkie obiekty reprezentujące urządzenia mogą być znacznie uproszczone:

- Informują one mediatora o zmianach stanu.
- Odpowiadają na żądania mediatora.

Przed wprowadzeniem wzorca do projektu wszystkie obiektu urządzeń musiały wiedzieć o sobie nawzajem… obowiązywała zasada ścisłych powiązań. Mediator zapewnia im *całkowitą separację*.

Mediator zawiera całość logiki sterowania systemu. Gdy jedno z urządzeń wymaga nowej reguły albo pojawia się całkiem nowe urządzenie, jest jasne, że wszelka logika zostanie zaimplementowana w klasie mediatora.



Zalety wzorca Mediator

- Zwiększa szanse ponownego wykorzystania obiektów, z którymi współpracuje mediator, bo zapewnia ich separację od systemu.
- Upraszcza rozwijanie systemu, bo skupia w jednym miejscu logikę sterowania.
- Upraszcza i zmniejsza liczbę komunikatów przesyłanych pomiędzy obiektami w systemie.

Zastosowania i wady wzorca Mediator

- Wzorzec Mediator jest powszechnie stosowany do koordynowania powiązanych składników graficznego interfejsu użytkownika.
- Wadą wzorca jest to, że gdy projekt nie jest dopracowany, sam obiekt mediatora może stać się za bardzo skomplikowany.

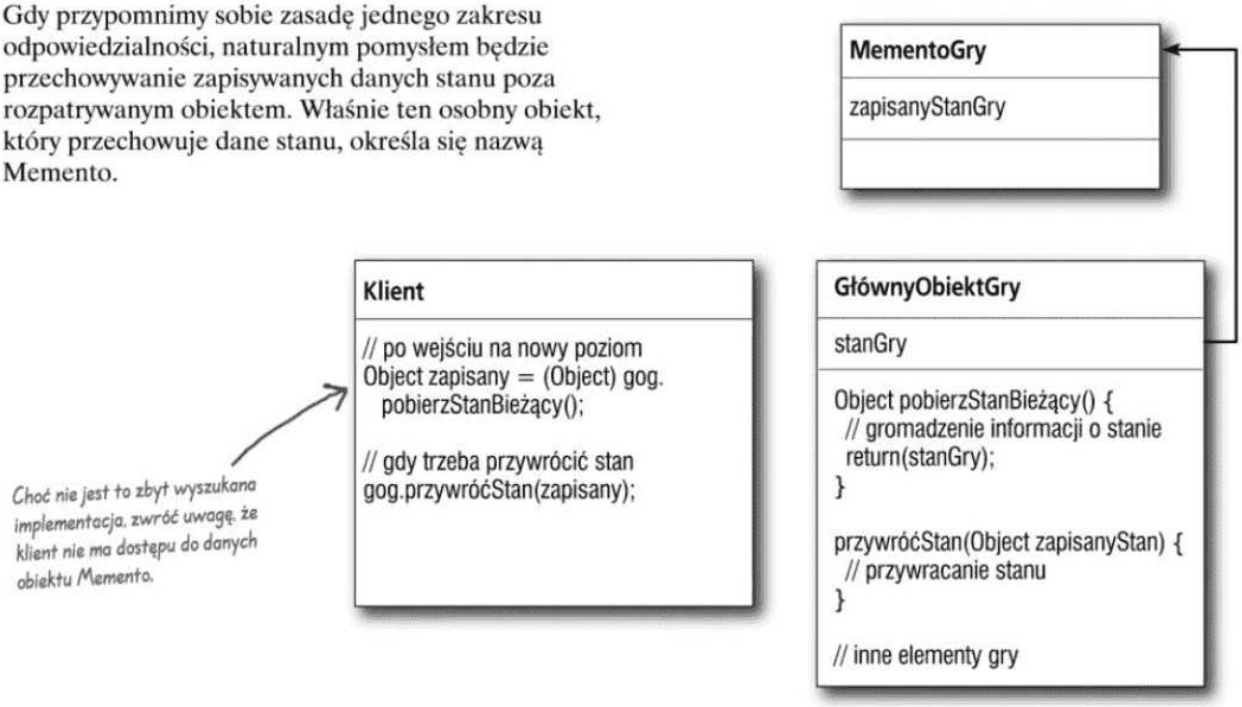
MEMENTO

Wzorca Memento używamy wtedy, gdy potrzebujemy możliwości przywrócenia obiektu do jednego z wcześniejszych stanów. Przykładem może być operacja użytkownika „Cofnij”.

Wzorzec Memento ma dwa cele:

- Zapisanie istotnych danych stanu kluczowego obiektu systemu.
- Hermetyzacja tego obiektu.

Gdy przypomnimy sobie zasadę jednego zakresu odpowiedzialności, naturalnym pomysłem będzie przechowywanie zapisywanych danych stanu poza rozpatrywanym obiektem. Właśnie ten osobny obiekt, który przechowuje dane stanu, określa się nazwą Memento.



Zalety wzorca Memento

- Pozostawienie zapisywanego stanu poza głównym obiektem pomaga zachować spójność.
- Zapewnia hermetyzację danych głównego obiektu.
- Zapewnia funkcję przywracania, która jest stosunkowo łatwa do implementacji.

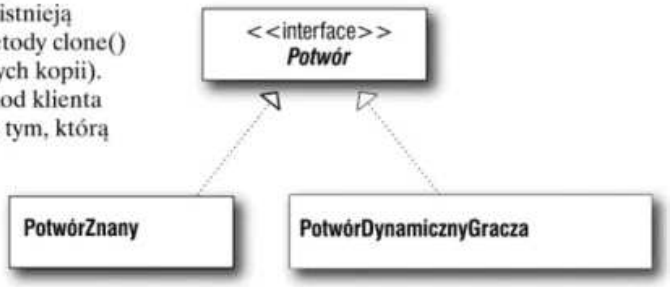
Zastosowania i wady wzorca Memento

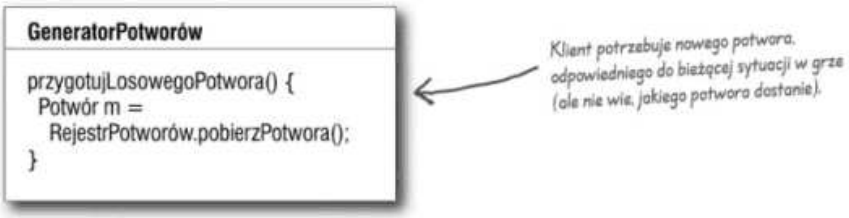
- Wzorzec Memento służy do zapisywania stanu.
- Wadą wzorca Memento jest to, że zapisywanie i przywracanie danych może być czasochłonne.
- W systemach opartych na języku Java warto rozważyć zastosowanie serializacji do zapisywania stanu systemu.

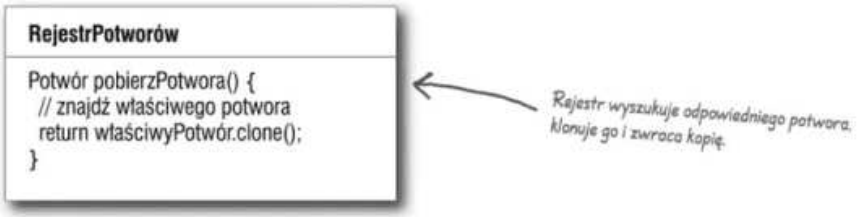
PROTOTYP

Wzorca Prototyp używamy wtedy, gdy tworzenie egzemplarza danej klasy jest w pewien sposób kosztowne.

Wzorzec Prototyp umożliwia tworzenie nowych egzemplarzy przez kopiowanie tych, które już istnieją (w języku Java oznacza to zazwyczaj użycie metody clone() lub deserializację, gdy potrzebujemy pełniejszych kopii). Kluczowym aspektem tego wzorca jest to, że kod klienta może tworzyć nowe egzemplarze bez wiedzy o tym, którą klasą rzeczywistą operuje.







Zalety wzorca Prototyp

- Ukrywa złożoność operacji tworzenia nowych egzemplarzy przed klientem.
- Zapewnia klientowi możliwość generowania obiektów, których typ nie jest znany.
- W pewnych warunkach kopiowanie obiektów może być wydajniejsze niż generowanie nowych.

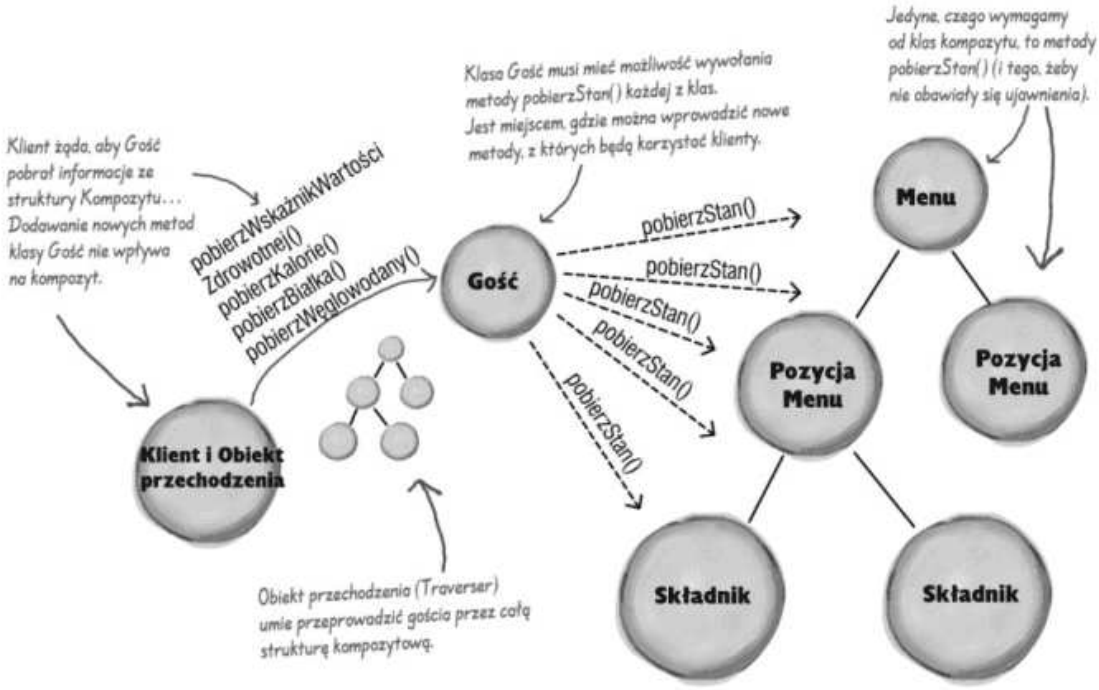
Zastosowania i wady wzorca Prototyp

- Warto rozważyć zastosowanie wzorca Prototp, gdy system wymaga tworzenia nowych obiektów różnych typów w złożonej hierarchii klas.
- Wadą wzorca jest to, że utworzenie kopii obiektu może być niekiedy bardzo skomplikowane.

VISITOR

Wzorca Visitor używamy, aby rozbudować zakres funkcji obiektów kompozytowych, gdy hermetyzacja nie jest ważna.

Obiekt Visitor („gość") odwiedza każdy element kompozytu. Ta funkcja jest częścią obiektu Traverser, czyli „obiektu przechodzenia". Zgodnie z jego nakazem obiekt gościa gromadzi dane stanu wszystkich elementów kompozytu. Po wykonaniu takiej operacji klient może żądać od klasy gościa wykonywania różnorodnych operacji na pobranych danych. Gdy pojawia się potrzeba dodania nowych funkcji, jest to jedyna modyfikowana klasa.



Zalety wzorca Visitor

- Umożliwia dodawanie nowych operacji do struktury kompozytowej bez modyfikowania samej struktury.
- Dodawanie nowych operacji jest stosunkowo proste.
- Kod operacji wykonywanych przez Gościa jest skupiony w jednym miejscu.

Zastosowania i wady wzorca Visitor

- Wprowadzenie wzorca Visitor łamie hermetyzację uzyskaną dzięki zastosowaniu struktury kompozytowej.
- Ponieważ wykorzystywana jest funkcja przechodzenia między elementami kompozytu, wprowadzanie w niej zmian jest utrudnione.