Zaawansowane programowanie obiektowe i funkcyjne

Wykład 10: Wzorce projektowe

dr inż. Marcin Luckner
mluckner@mini.pw.edu.pl

Wydział Matematyki i Nauk Informacyjnych

Wersja 1.1
4 marca 2021

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Rzeczpospolita
Polska

Politechnika
Warszawska

Wzorce projektowe

- Wzorce projektowe *Design Patterns* są powszechnie stosowanym sposobem organizacji obiektów w programach.
- Ułatwiają pisanie programów i utrzymywanie kodu.
- Ułatwiają komunikację między programistami umożliwiając opisywanie kodu przy pomocy wspólnego słownictwa.

Definicja wzorca

Definicja naukowa

Wzorzec identyfikuje i specyfikuje pewną abstrakcję, której poziom znajduje się powyżej poziomu abstrakcji pojedynczej klasy, instancji lub komponentu.

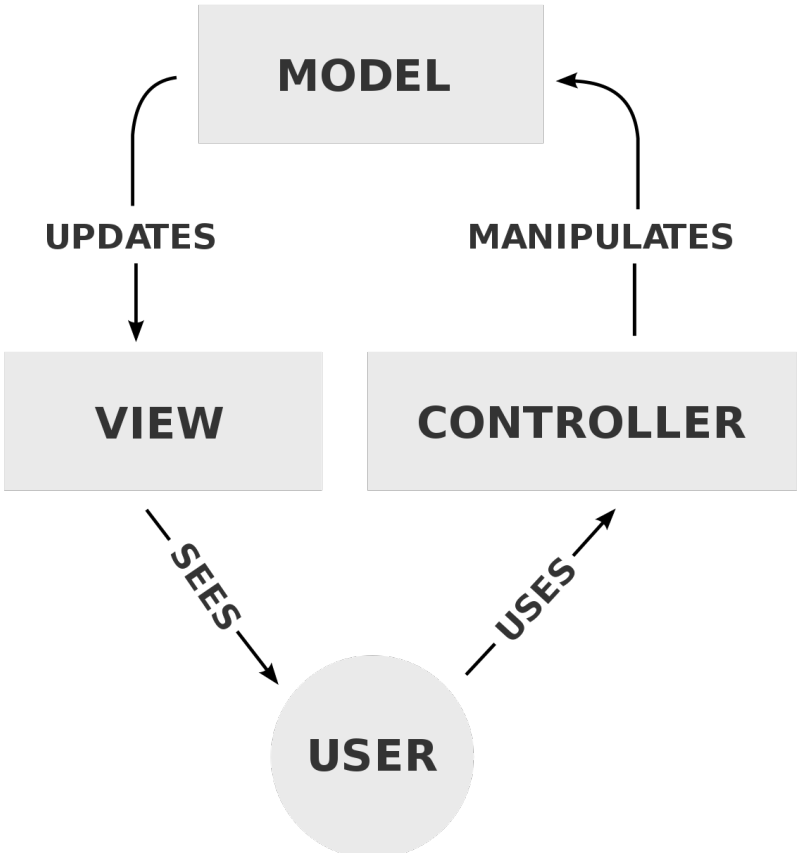
[Gamma et al., 1993]

Definicja biznesowa

Wzorce projektowe stanowią powtarzalne rozwiazanie zagadnień projektowych z którymi wciąż się spotykamy.

[Alpert et al., 1998]

Typy wzorców projektowych



Rysunek 1: *Model-Widok-Kontroler [Wikipedia, 2019]*

- Popularnym i jednym z najstarszych wzorców jest *Model-View-Controler* [Krasner and Pope, 1988].
- Wzorzec rozdziela interfejs użytkownika na trzy części
  - Model odpowiadający za przetrzymywanie danych.
  - Widok prezentujący interfejs użytkownika.
  - Kontroler pośredniczący między użytkownikiem a widokiem.

Typy wzorców projektowych

- Wzorce konstrukcyjne
  - Tworzą obiekty.
- Wzorce strukturalne
  - Łączą obiekty w struktury.
- Wzorce behawioralne
  - Określają komunikację i przepływ danych.

Wzorce konstrukcyjne

- Wzorce konstrukcyjne służą do pozyskiwania obiektów.
- Mają zastąpić stosowane w Javie tworzenie instancji przy pomocy operatora `new`.
- Pozwalają na pisanie bardziej ogólnego i elastycznego kodu.

Typy wzorców projektowych

Factory Method Metoda fabrykująca - klasa która w zależności od otrzymanych danych zwraca obiekt jednej z wielu pochodnych klasy podstawowej.

Abstract Factory Fabryka abstrakcji - dostarcza interfejs umożliwiający tworzenie rodzin spokrewnionych lub zależnych od siebie obiektów.

Builder Budowniczy - oddziela konstruowanie złożonych obiektów od ich reprezentacji.

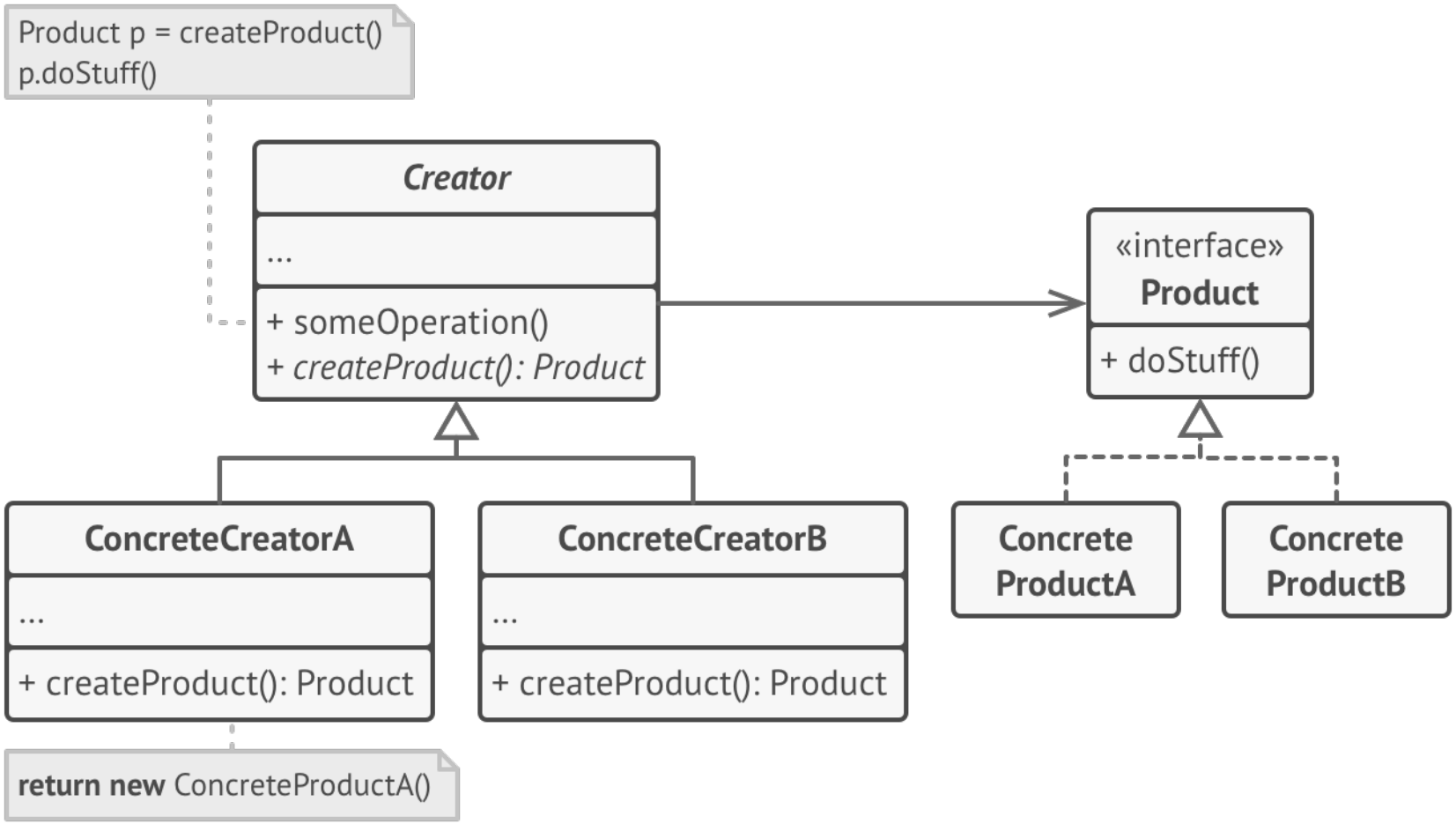
Prototype Prototyp - pozwala na klonowanie istniejących instancji klasy i ich późniejsze modyfikowanie.

Singleton Klasa zezwalająca na utworzenie tylko jednego obiektu.

Metoda fabrykująca

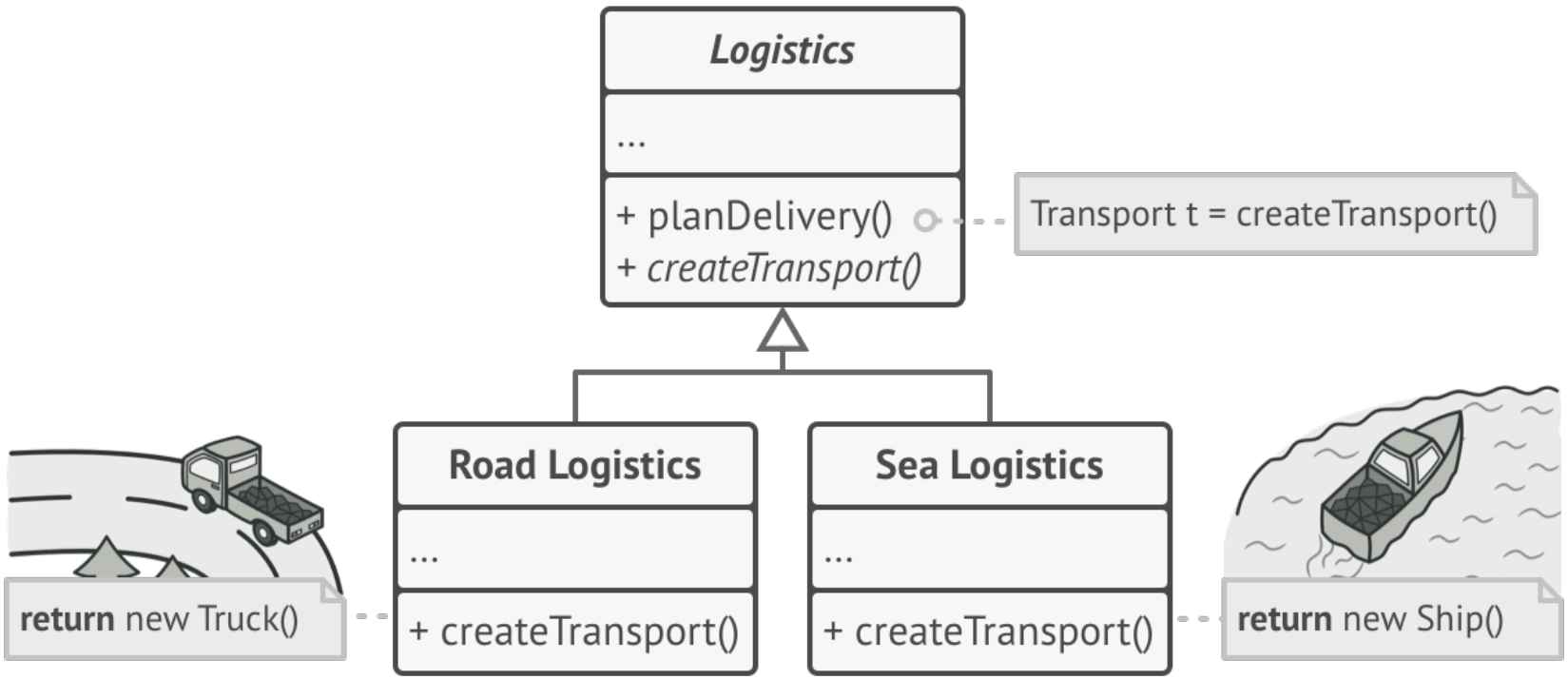
- Metoda fabrykująca pozwala na utworzenie instancji klasy w zależności od parametrów.
- Klasa kreatora (*Creator*) przekazuje poprzez metodę jedną z instancji klas podrzędnych.
- Powstaje produkt (*Product*) będący abstrakcją implementujących go klas, odzwierciedlających strukturę metody fabrykującej.

Schemat Metody fabrykującej



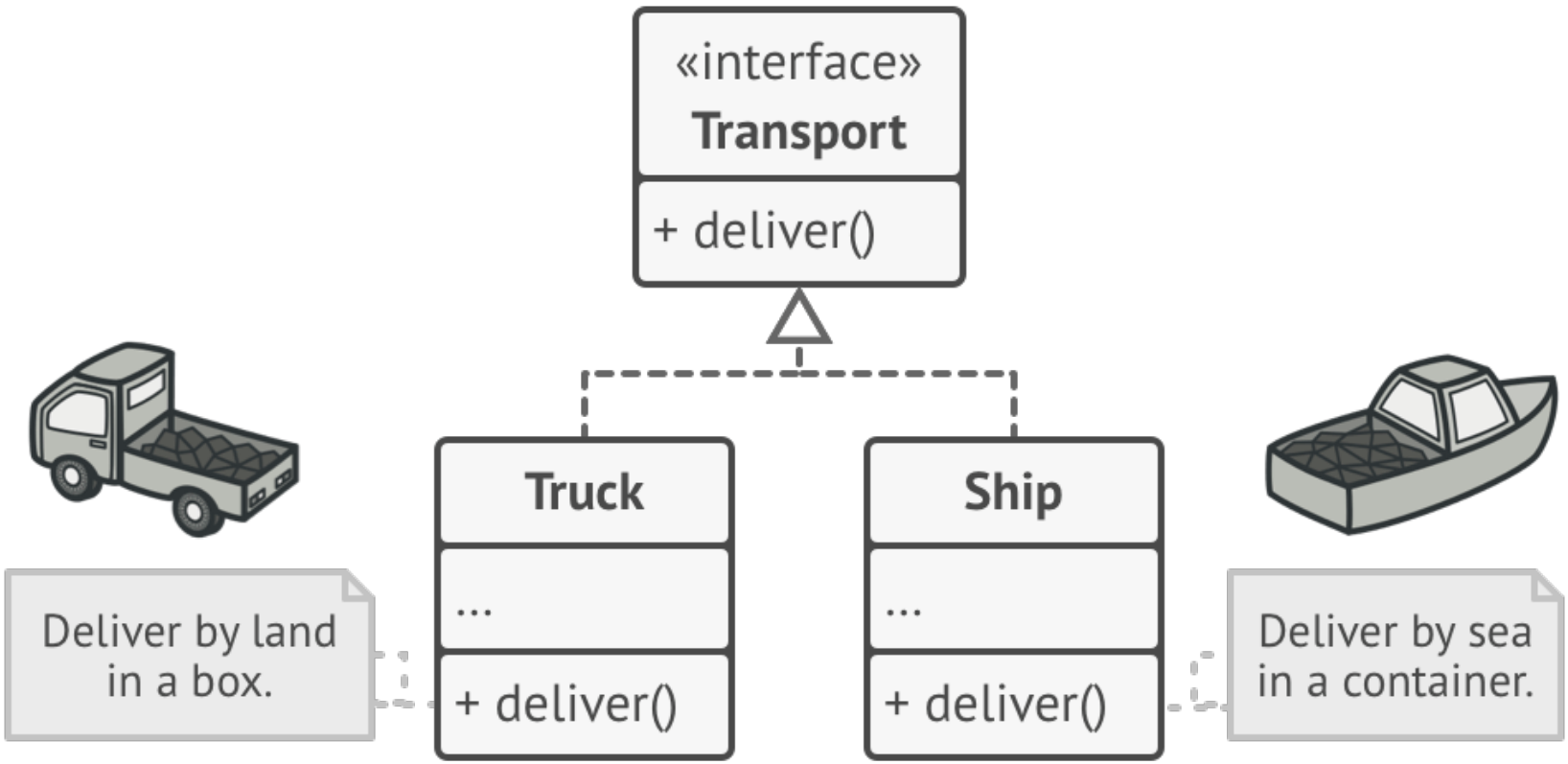
Rysunek 2: *Schemat Metody fabrykującej [Refactoring.Guru, 2019]*

Przykład produkcji



Rysunek 3: *Produkcja środków transportu [Refactoring.Guru, 2019]*

Przykład produktów



Rysunek 4: *Środki transportu [Refactoring.Guru, 2019]*

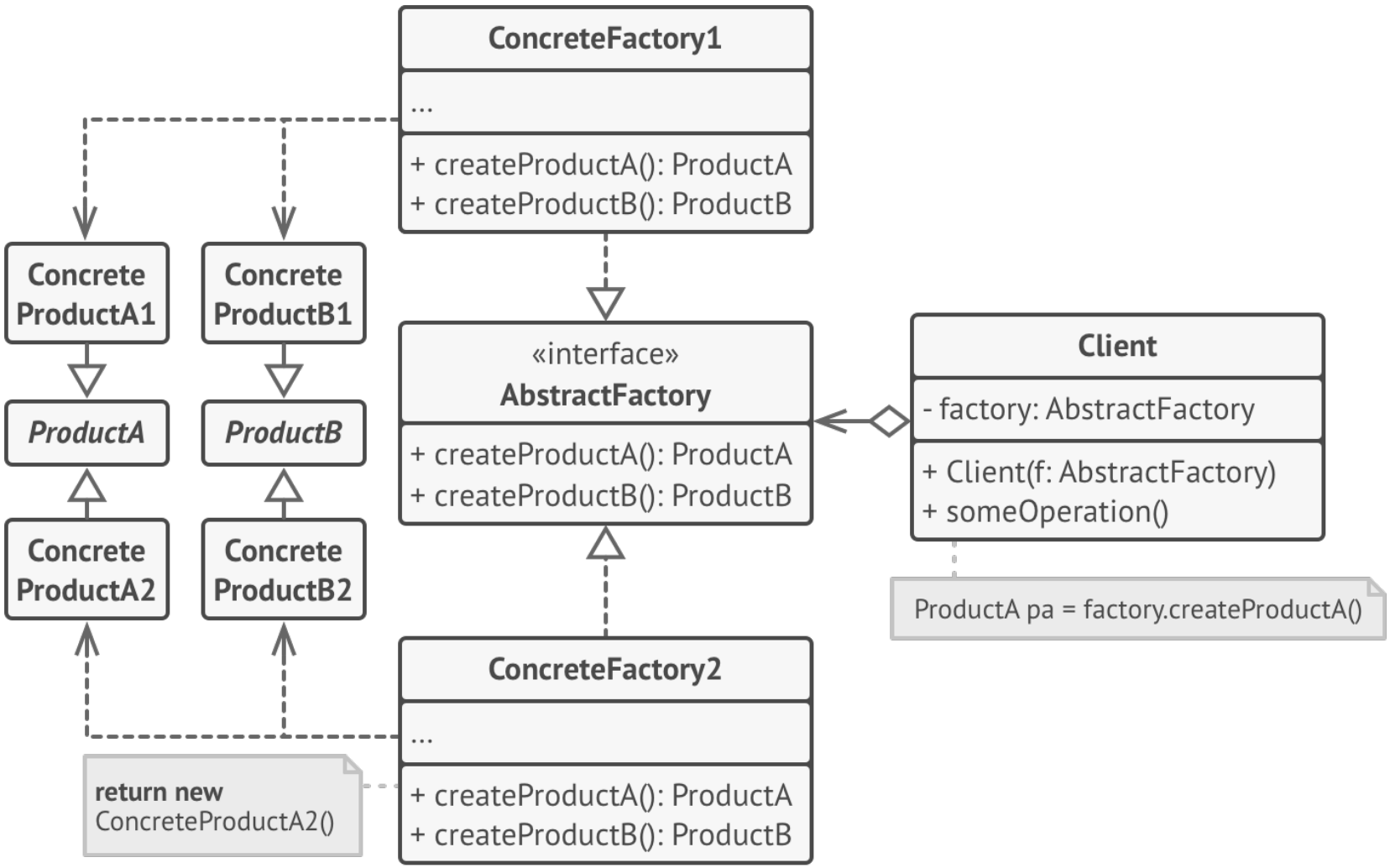
Stosowanie Metody fabrykującej

- Należy stosować jeżeli klasa mająca stworzyć nowy obiekt nie ma wiedzy jaki rodzaj obiektu należy utworzyć.
- Rozdziela tworzenie obiektów od ich dostarczania co ułatwia wsparcie kodu.
- Może powodować nadmierne rozbudowanie hierarchii klas o liczne klasy pochodne.

Fabryka abstrakcji

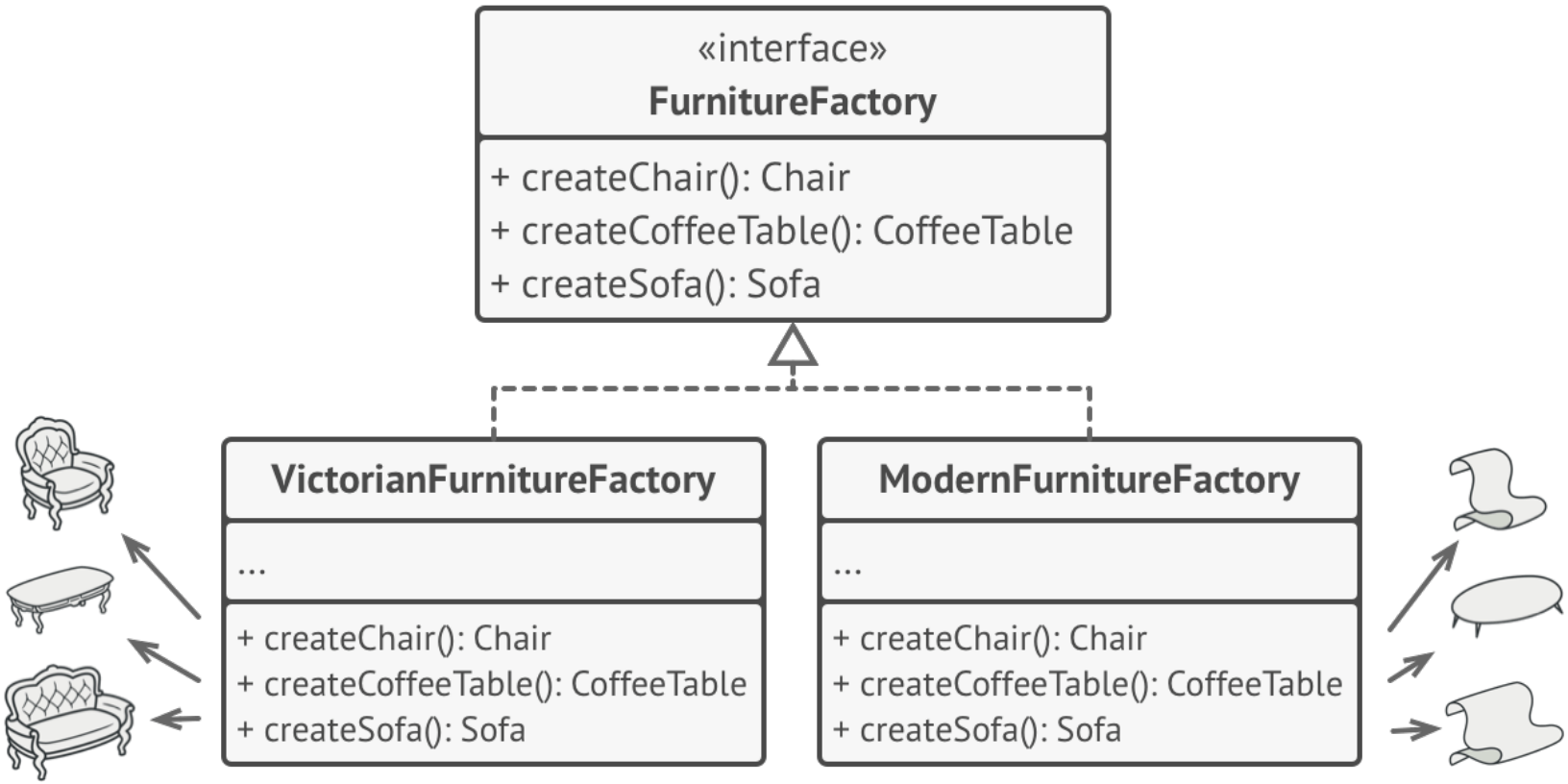
- Fabryka abstrakcji dostarcza schemat fabryki dostarczającej różne produkty.
- Dostarczane produkty są ze sobą powiązane tworząc rodziny/serie produktów.
- Za klasą fabryki abstrakcyjnej kryją się różne rodzaje fabryk, z których każda dostarcza produkty z innej rodziny/serii.

Schemat Fabryki abstrakcji



Rysunek 5: *Schemat Fabryki abstrakcji [Refactoring.Guru, 2019]*

Przykład produkcji



Rysunek 6: *Produkcja mebli [Refactoring.Guru, 2019]*

Stosowanie Fabryki abstrakcji

- Fabryka abstrakcji jest stosowana w celu odizolowania i ukrycia w fabryce tworzonych klas.
- Izolacja pozwala zmieniać rodziny produkowanych klas.
- Wzorzec zapewnia, że elementy nie zostaną wymieszane między rodzinami.

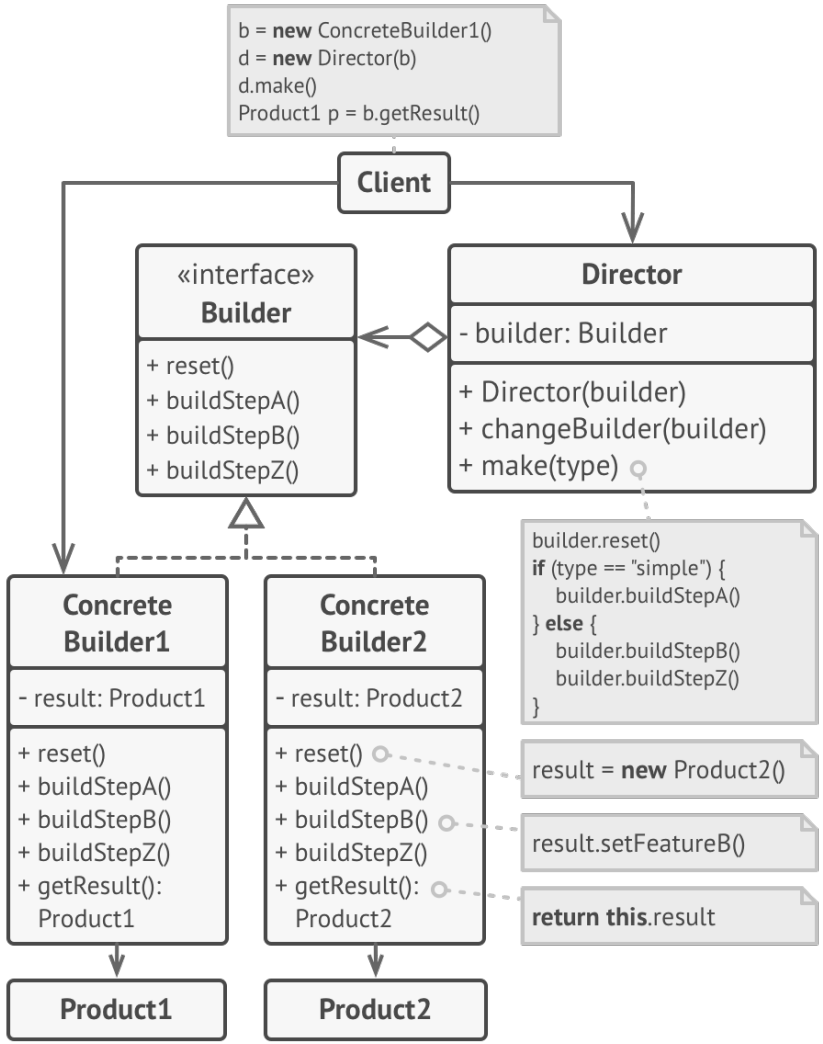
Stosowanie Fabryki abstrakcji w Javie

- Fabryka abstrakcji jest stosowana w Javie do tworzenia odmian interfejsów użytkownika.
- Fabryki budują kontrolki różnych typów (przyciski, suwaki, listy itp...).
- Każda z fabryk produkuje te same kontrolki ale o wyglądzie dostosowanym do systemu operacyjnego albo wyglądu przenośnego Javy.

Budowniczy

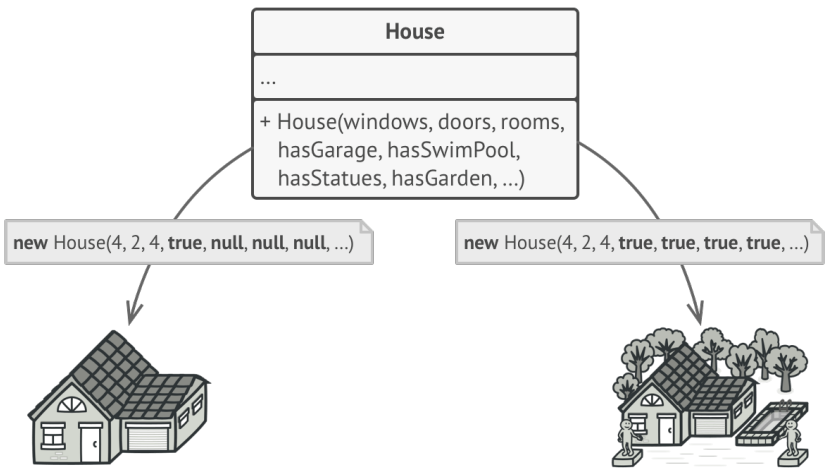
- Budowniczy tworzy schemat budowy obiektu z komponentów.
- Korzystają z jednej z dostępnych implementacji tworzenia obiektu przekazujemy dodatkowe informacje na temat ich wykorzystania.
- Dodatkowe informacje są przekazywane przez klasę zarządcza *Director*.
- Zarówno klasy implementujące, jak i klasa zarządcza opierają się na schemacie dostarczanym przez Budowniczego.

Schemat Budowniczego

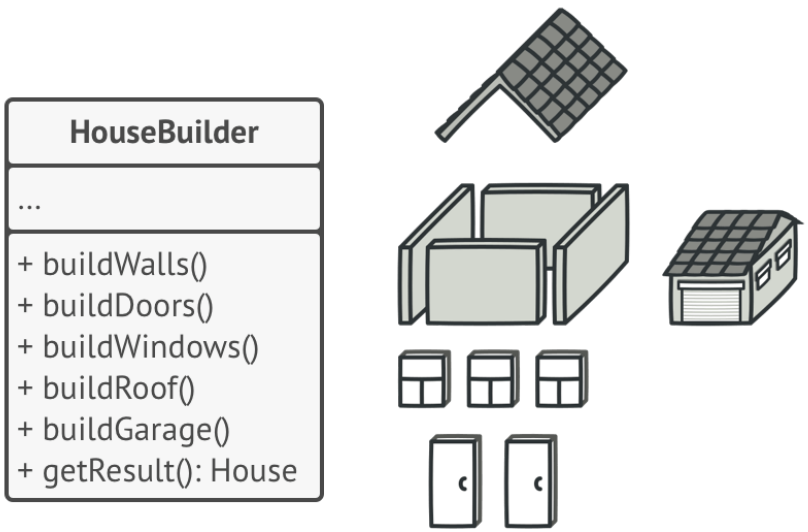


Rysunek 7: *Schemat Budowniczego [Refactoring.Guru, 2019]*

Zastosowanie Budowniczego



Rysunek 8: *Konstruktor parametryczny [Refactoring.Guru, 2019]*



Rysunek 9: *Budowniczy [Refactoring.Guru, 2019]*

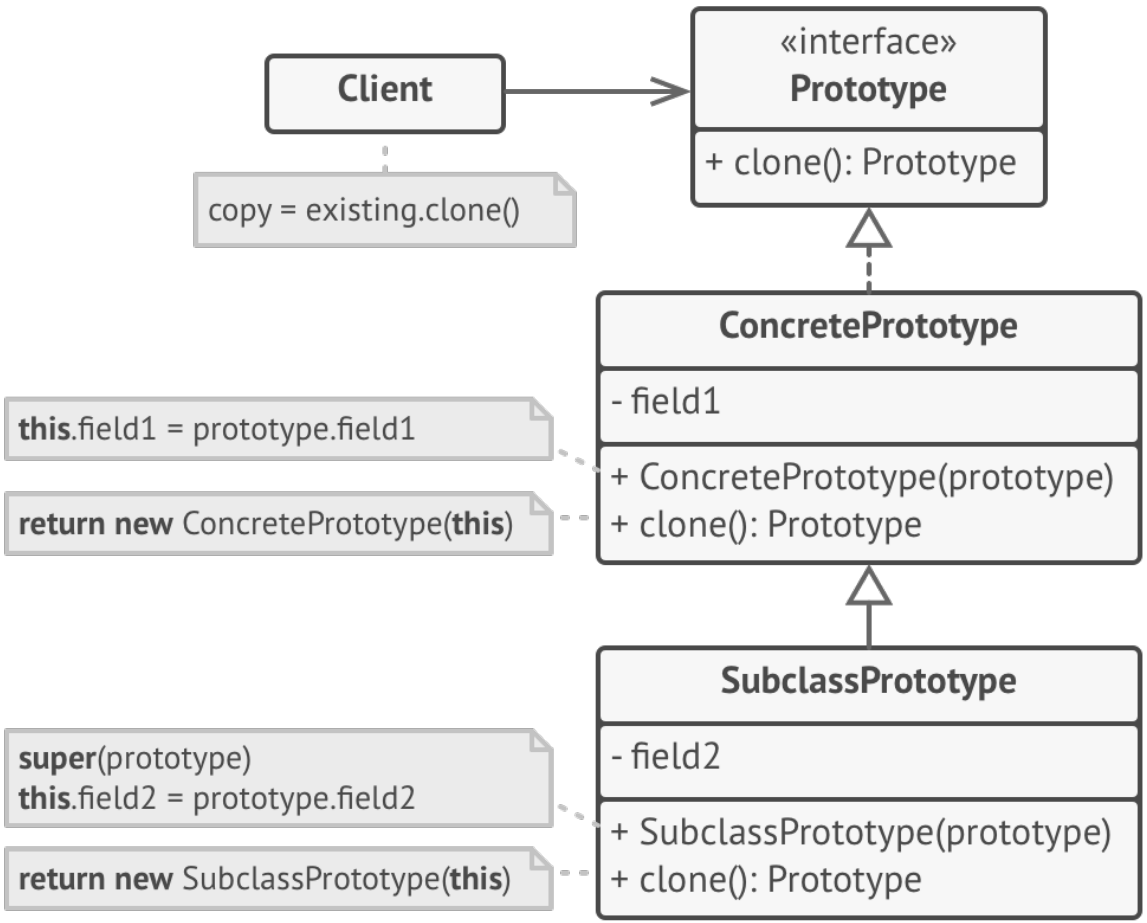
Stosowanie Budowniczego

- Wzorzec Budowniczego pozwala wyeliminować tworzenie wielu konstruktorów parametrycznych.
- Pozwala tworzyć różne reprezentacje takich samych produktów.
- Pozwala tworzyć produkty krok po kroku, rekurencyjnie lub odraczając wytwarzanie.

Prototyp

- Wzorzec Prototypu pozwala na skopiowanie a następnie na modyfikację istniejącego obiektu.
- W wypadku skomplikowanych obiektów utworzenie kopi i jej modyfikacja może być tańsze niż stworzenie nowego obiektu.
- W szczególności może to dotyczyć obiektów zawierających wyniki zapytań do baz danych.

Schemat Prototypu



Rysunek 10: *Schemat Prototypu [Refactoring.Guru, 2019]*

Klonowanie w Javie

- W Javie można tworzyć kopię obiektu za pomocą `clone`

```
Sheep dollyClone = (Sheep) dolly.clone();
```

- Funkcja `clone` domyślnie zwraca typ `Object` dlatego wymagane jest rzutowanie wyniku.
- Metoda jest składową chronioną `protected` i może być wywoływana jedynie z tej samej klasy lub klasy pochodnej.
- Można klonować tylko obiekty klasy, która implementuje interfejs `Cloneable`.
- Obiekty, których klasy nie implementują interfejsu `Cloneable` zgłaszają wyjątek `CloneNotSupportedException`.

Prototyp w Javie

- Ze względu na chroniony charakter należy obudować metodę clone metodą publiczną.

```
public abstract class queryData implements Cloneable{
  public Object cloneMe() throws
      CloneNotSupportedException{
    return super.clone();
  }
}
```

- Należy pamiętać, że clone tworzy płytką kopię obiektów osadzonych, czyli przekazuje do kopi ich referencje.
- Chcąc stworzyć głęboką kopię należy ręcznie utworzyć nowe instancje elementów.

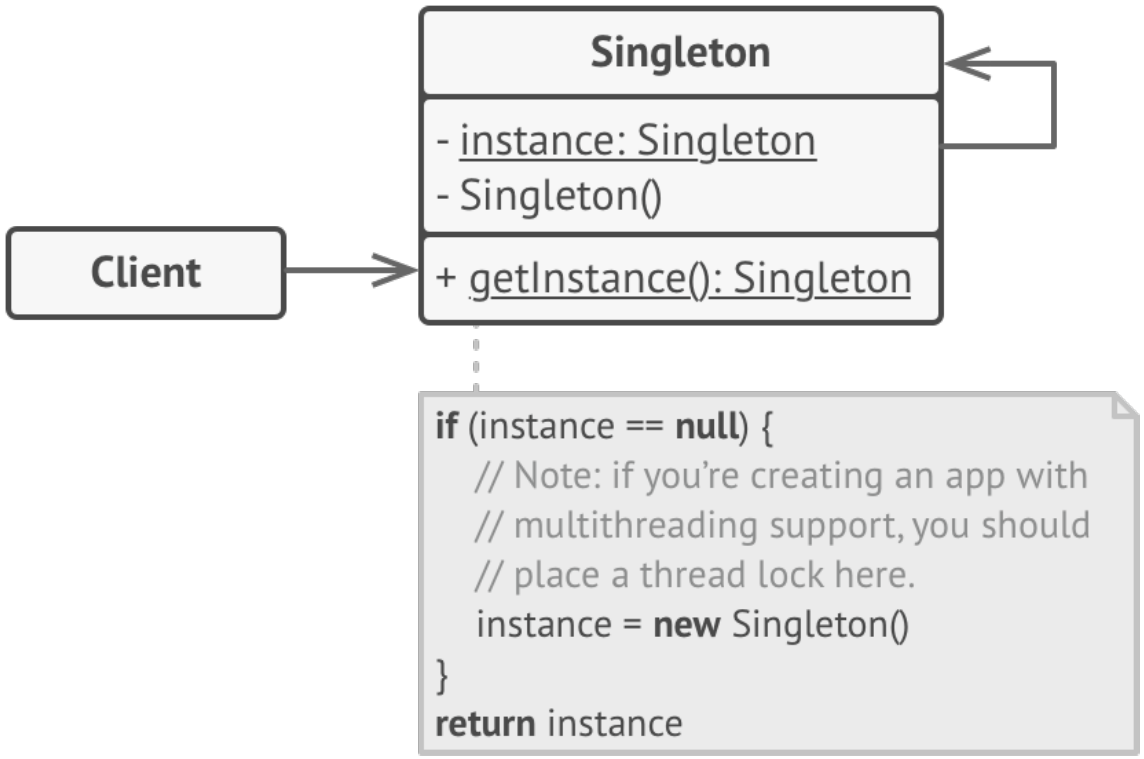
Stosowanie Prototypu

- Pozwala na klonowanie i modyfikację obiektów bez tworzenia nowych podklas.
- Pozwala na uniknięcie powtarzania kodu inicjującego.
- Daje alternatywę dla narzędzi dziedziczenia.

Singleton

- Singleton zapewnia istnieje tylko jednej kopi obiektu z danej klasy.
- Kopia powinna być dostępna dla wszystkich pozostałych klas.
- Jest stosowany gdy reprezentacja danej klasy musi się ograniczyć do pojedynczego wystąpienia.
- Najczęściej jest to związane z ograniczeniami sprzętowymi i systemowymi.

Schemat Singletonu



Rysunek 11: *Schemat Singletonu [Refactoring.Guru, 2019]*

Singleton w Javie

```
public class PrintSpooler{
  private static PrintSpooler;

  private PrintSpooler(){
    if(spooler == null){
      spooler = new PrintSpooler();
    }
    return spooler;
  }
}
```

- Problemem związanym z implementacją w Javie jest brak zmiennych globalnych.
- Utrudnia to udostępnianie tego obiektu pozostałym klasom.

Wzorce strukturalne

- Wzorce strukturalne opisują sposoby łączenia klas i obiektów w większe struktury.
- Wzorce klas opisują jak dziedziczenie może być używane w celu dostarczenia bardziej użytecznych interfejsów programistycznych.
- Wzorce obiektowe opisuje jak obiekty mogą być łączone w większe struktury poprzez zawieranie się jednych obiektów w innych.

Typy wzorców strukturalnych

Adapter Tworzy nowy interfejs dla klasy, aby dostosować go do interfejsu innej klasy.

Composite Kompozyt - tworzy obiekt będący kompozycją obiektów.

Proxy Pośrednik - tworzy prosty obiekt zastępujący obiekt złożony, który będzie dostępny w późniejszym czasie.

Flyweight Pyłek - pozwala na unikanie wielokrotnego tworzenia takich samych obiektów.

Facade Fasada - tworzy pojedynczy interfejs dla całego systemu.

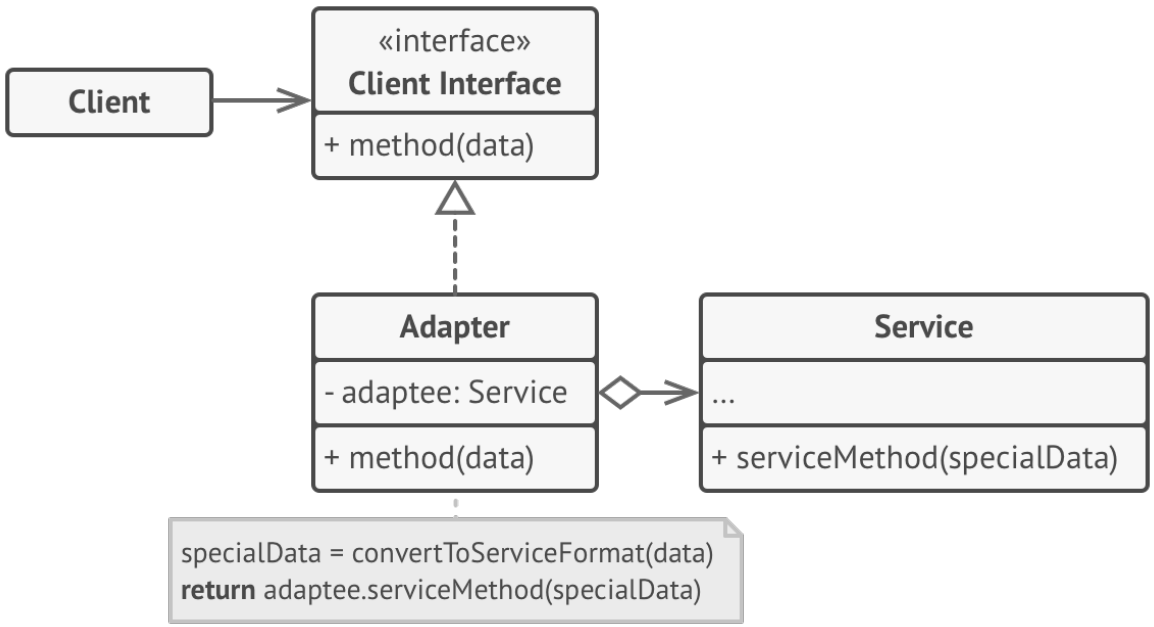
Bridge Most - oddziela interfejs od implementacji umożliwiając ich niezależne modyfikowanie.

Decorator Dekorator - pozwala dynamicznie dodawać nowe cechy do obiektu.

Adapter

- Adapter umożliwia współpracę niekompatybilnym klasom.
- Implementacja sprowadza się do napisania klasy implementującej wymagany interfejs i zapewnienie jej komunikacji z klasą, która ma inny interfejs.
- Możliwa jest implementacja przez dziedziczenie lub kompozycję.
- W implementacji przez dziedziczenie wywodzimy klasę pochodną i dodajemy implementację interfejsu.
- W implementacji poprzez kompozycję zawieramy pierwotną klasę wewnątrz nowej i tworzymy metody interfejsu wywołujące metody klasy wewnętrznej.

Schemat Adapter



Rysunek 12: *Schemat Adeptera [Refactoring.Guru, 2019]*

Adapter w Javie

- Java stosuje Adaptery do implementacji klas nasłuchujących.
- Przykładowo, klasa `WindowAdapter` implementuje puste metody interfejsu `WindowListener` pozwalając użytkownikowi na nadpisanie tylko zmienianej metody.
- Wykorzystanie mechanizmu refleksji pozwala także na dynamiczne tworzenie Adapterów poprzez odczyt jakie metody wymagają implementacji `getClass().getMethod()`

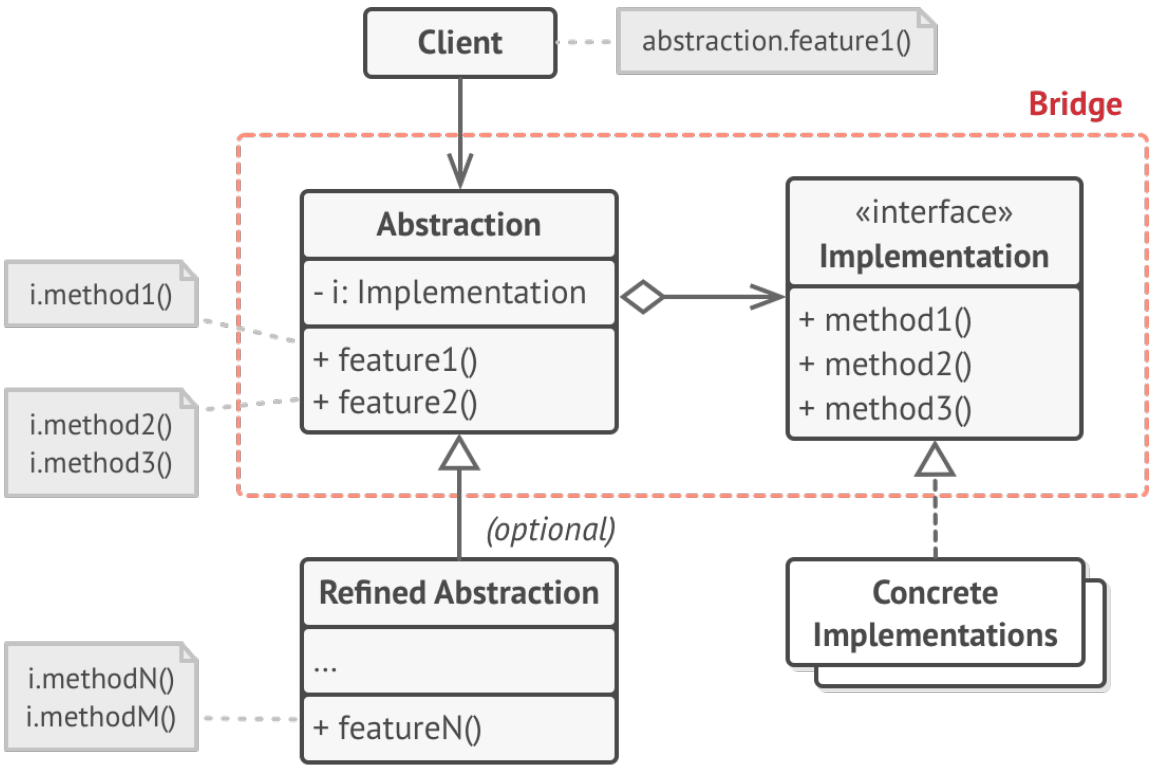
Most

- Most pozwala podzielić dużą klasę lub zbiór spokrewnionych klas na dwie osobne hierarchie.
- Wyodrębnione hierarchie to:
  - Hierarchia abstrakcji.
  - Hierarchia implementacji.
- Pozwala to na rozwój każdej hierarchii z osobna.

Elementy Mostu

- Na most składają się.
  - Abstrakcja definiująca interfejs Mostu.
  - Implementator definiujący interfejs klasy implementującej.
  - Konkretne implementatory.
  - Abstrakcja pierwotna.

Schemat Mostu

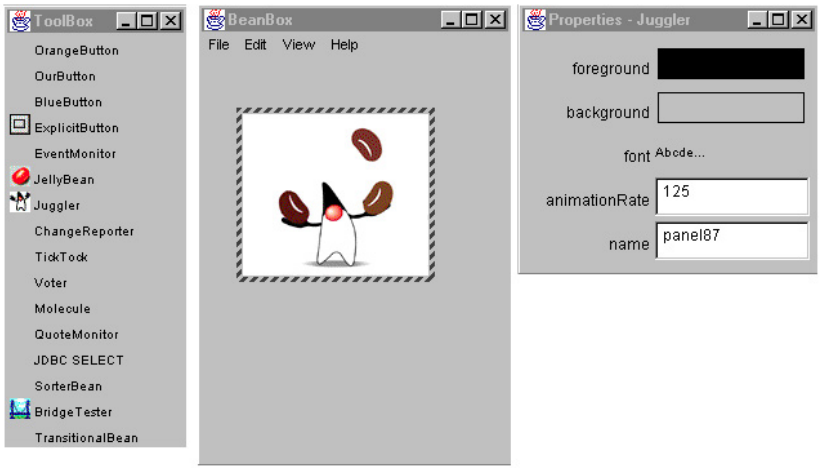


Rysunek 13: *Schemat Mostu [Refactoring.Guru, 2019]*

Stosowanie Mostu

- Most pozwala zachować stały interfejs dla programu klienta podczas wprowadzania zmian w klasach implementujących.
- Można niezależnie rozszerzać klasy implementujące i klasę mostu.
- Pozwala na zmianę sposobu implementacji w czasie działania programu.

Stosowanie Mostu w Javie



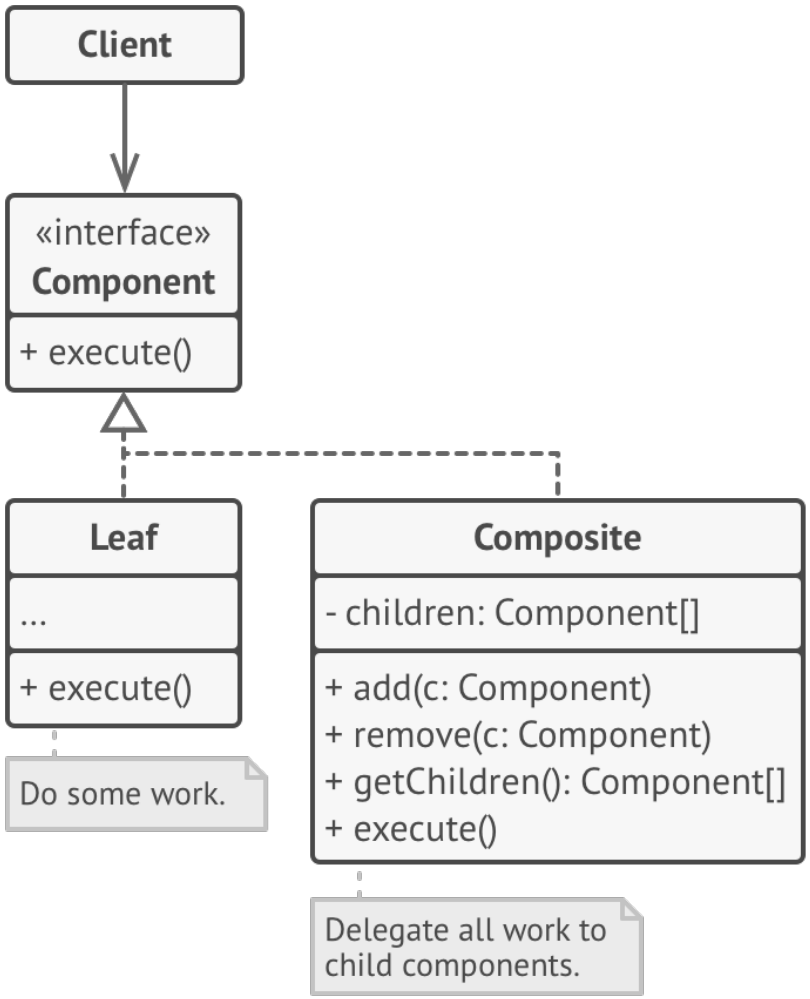
Rysunek 14: *Zastosowanie JavaBeans*

- Most jest podstawą JavaBeans - komponentu wizualnego, który można modyfikować online.
- Stosuje się go do budowy GUI i programowania wizualnego.

Kompozyt

- Kompozyt jest wzorcem porządkującym obiekty w strukturze drzewiastej.
- Następnie można pracować z daną strukturą tak jakby była pojedynczym obiektem.
- Nadal istnieje dostęp do poszczególnych komponentów.
- Co więcej można rozróżnić węzły wewnętrzne drzewa od liści.

Schemat Kompozytu



Rysunek 15: *Schemat Kompozytu [Refactoring.Guru, 2019]*

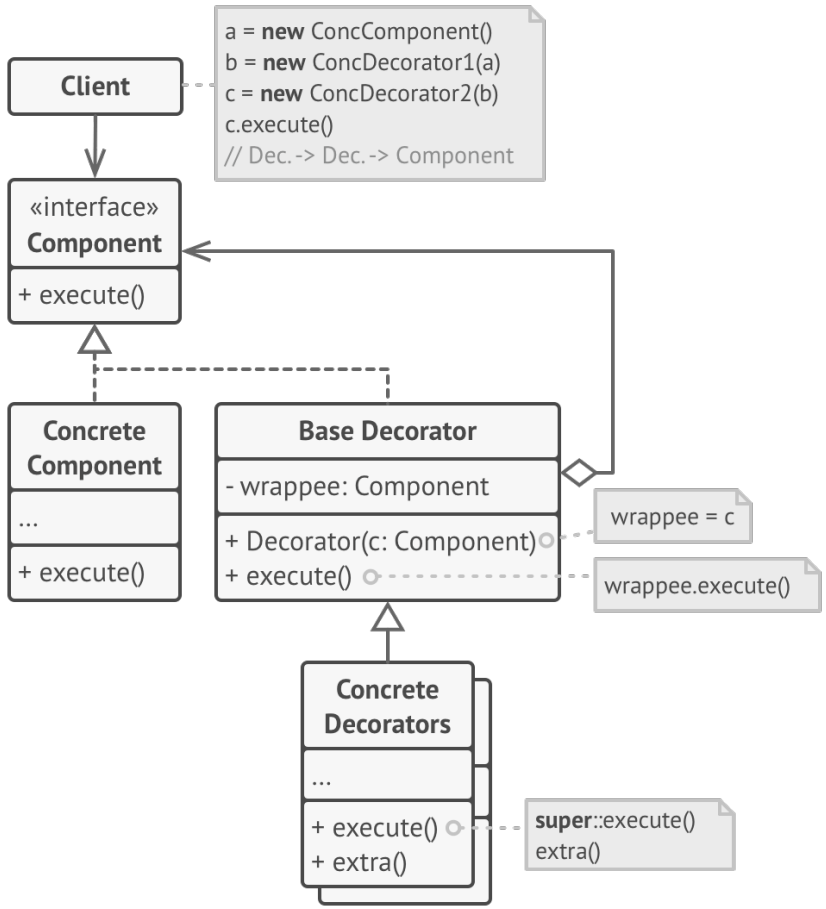
Stosowanie Kompozytu w Javie

- Kompozyt jest wykorzystywany do odwzorowania struktury hierarchicznej.
- Rozróżnienie między liściem a węzłem wewnętrznym odbywa się przy pomocy trybu `Enumerate` informującego czy istnieje podzbiór potomków.
- Klasa `JTree` pozwala na budowanie wzorca z kompozytów `DefaultMutableTreeNode`.
- Wzorzec Kompozytu opisuje hierarchię takich klas jak `JFrame`, `JPanel` i `JComponents` wykorzystywanych do budowy GUI.

Dekorator

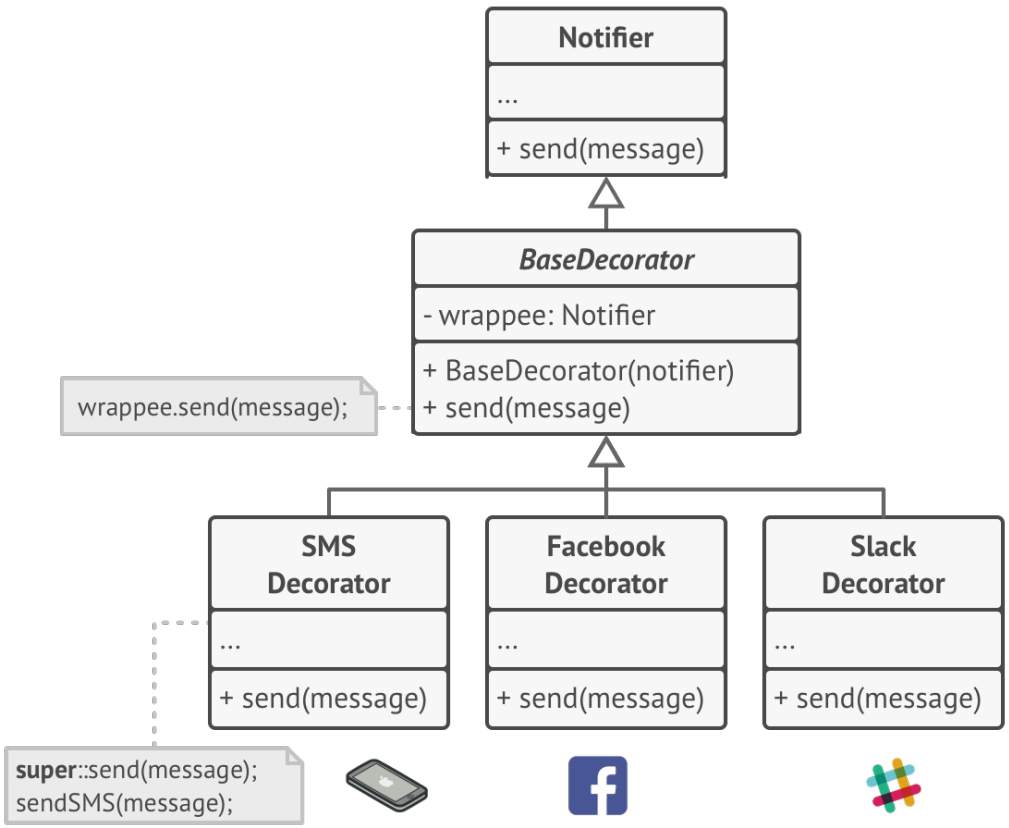
- Dekorator jest wzorcem pozwalającym na dołączenie nowego zachowania do obiektu.
- Nowe zachowanie dołączane jest poprzez włożenie obiektu do opakowania (*wrapper*).
- Tworzymy klasę dekoratora, który zapewnia nowe zachowanie.
- Następnie implementacje dekoratora zapewniają różne jego użycie.
- Wzorzec jest użyteczny szczególnie gdy mamy różne funkcjonalności i chcemy dołączać do obiektów ich różne podzbiory.

Schemat Dekoratora



Rysunek 16: *Schemat Dekoratora [Refactoring.Guru, 2019]*

Przykład Dekoratora

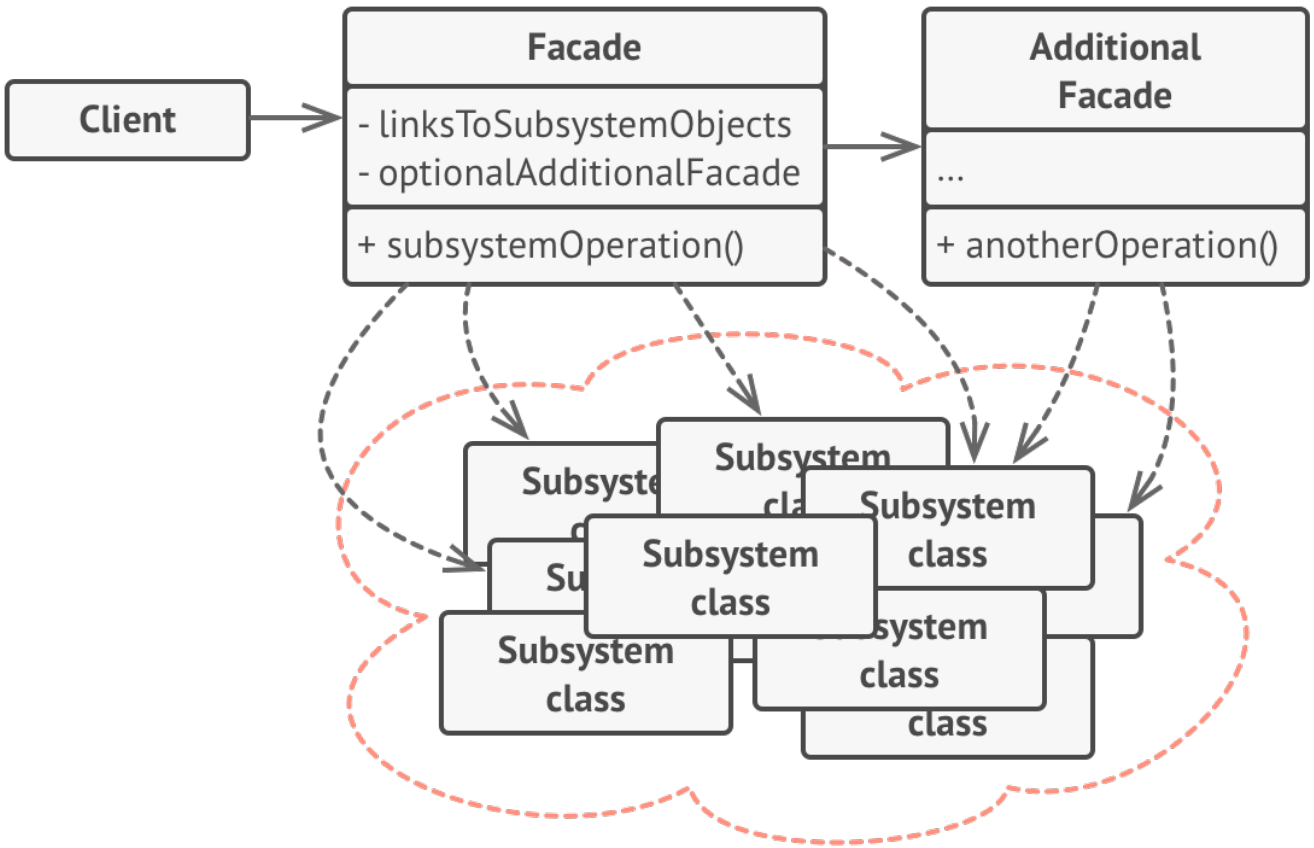


Rysunek 17: *Dekorator wysyłający notyfikacje [Refactoring.Guru, 2019]*

Fasada

- Fasada jest interfejsem umożliwiającym dostęp do wszystkich funkcji systemu.
- W szczególności jest wykorzystywana do udostępnienia zewnętrznego API.
- W Java EE Fasada jest implementowana jako `SessionBean` dający dostęp do chronionej zawartości bazy danych.

Schemat Fasady

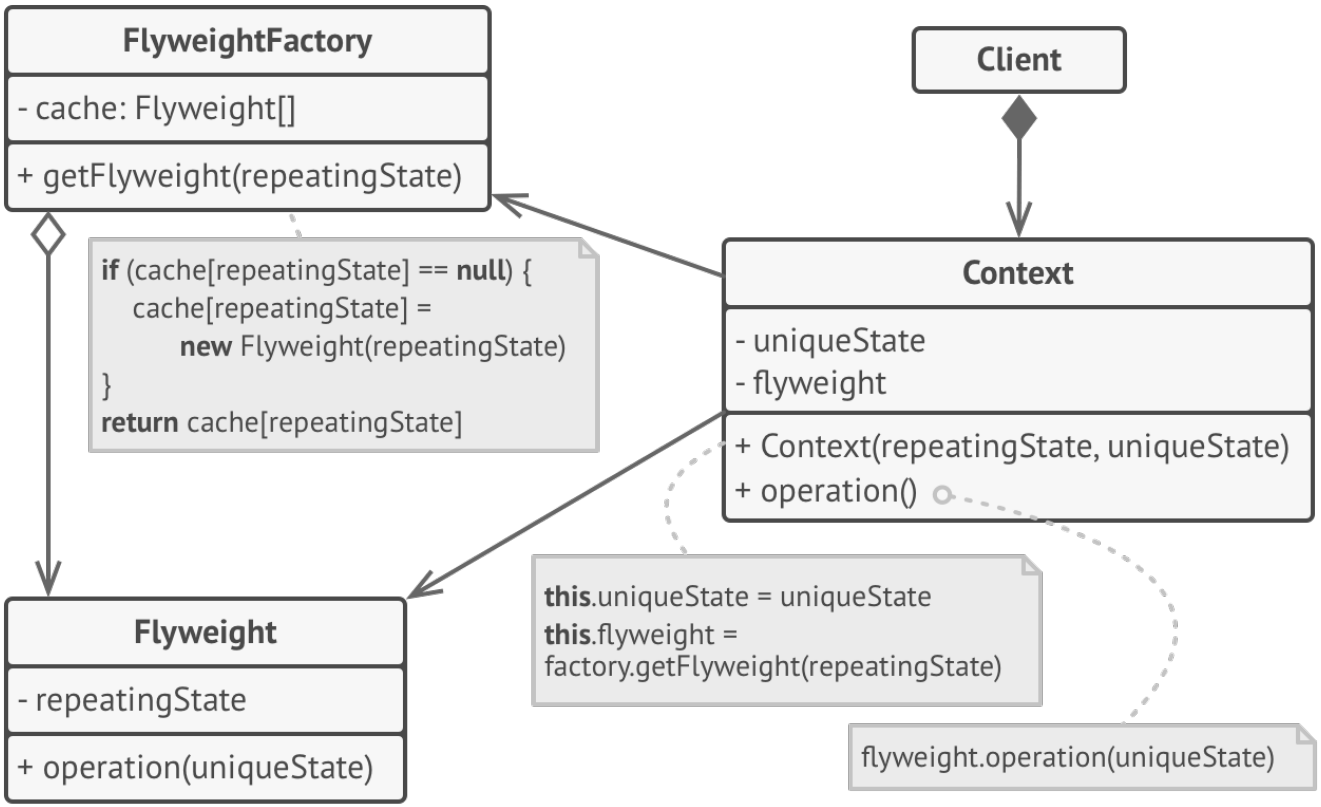


Rysunek 18: *Schemat Fasady [Refactoring.Guru, 2019]*

Pyłek

- Pyłek jest wzorcem pozwalającym na przetrzymywanie większej liczby obiektów w dostępnej pamięci RAM.
- Realizuje to poprzez przechowywanie współdzielonej części stanu obiektów zamiast zapisywać go wielokrotnie w wielu instancjach.
- Zazwyczaj używany na niższym poziomie niż na poziomie języka programowania.

Schemat Pyłka

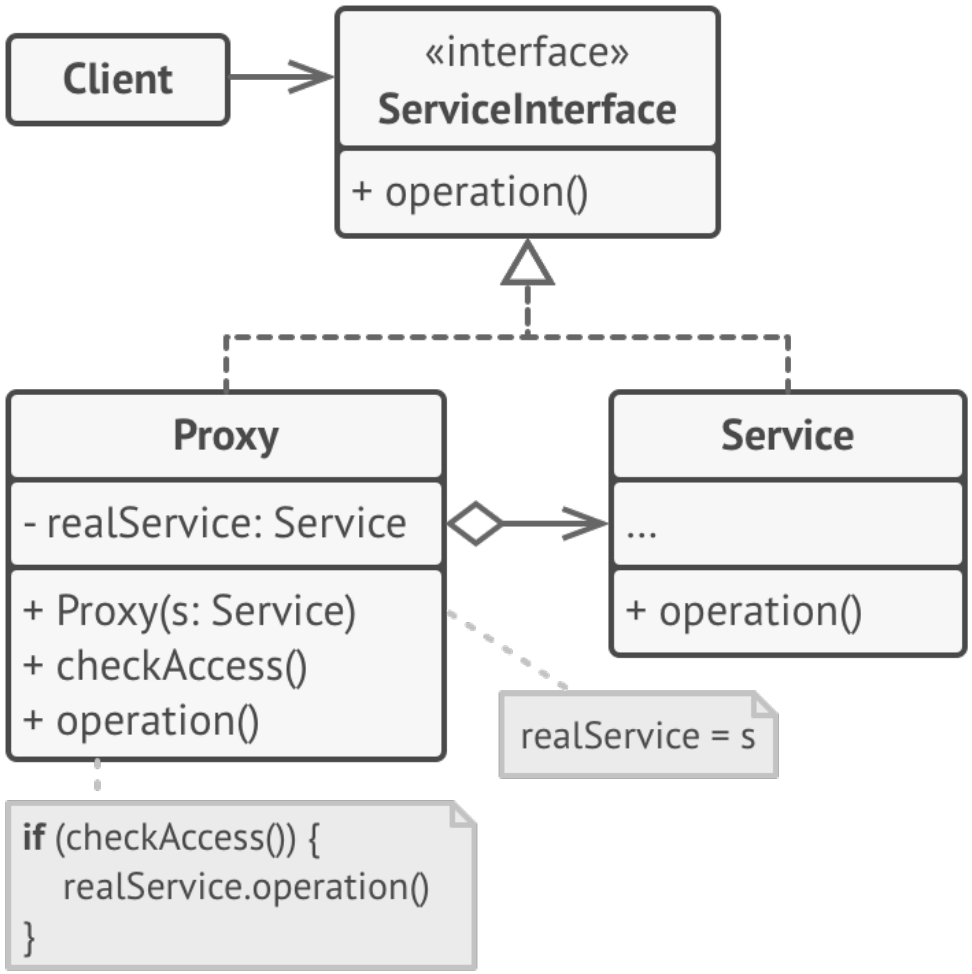


Rysunek 19: *Schemat Pyłka [Refactoring.Guru, 2019]*

Pośrednik

- Pośrednik (*proxy*) jest wzorcem dostarczającym zastępnika dla innego obiektu.
- W szczególności jest on stosowany gdy należy zastąpić obiekt którego powstawanie jest długotrwałe.
- Może też być używany do ograniczania dostępu do zasobów.
- Może być wykorzystywany do przechowywania lokalnej kopi najczęściej używanych zasobów pobieranych z serwera.

Schemat Pośrednika



Rysunek 20: *Schemat Pośrednika [Refactoring.Guru, 2019]*

Wzorce behawioralne

- Wzorce behawioralne są wzorcami dotyczącymi komunikowania się obiektów pomiędzy sobą.

Typy wzorców behawioralnych I

Chain of Responsibility Łańcuch odpowiedzialności - przekazywanie żądania do łańcucha powiązanych obiektów, aby zostało obsłużone przez jeden z nich.

Command Polecenie - nadaje żądaniu formę obiektu, pozwala na zrealizowanie dziennika poleceń i wycofanie wykonanych operacji.

Interpreter definiuje, w jaki sposób wprowadzić do programu składowe języka na podstawie jego gramatyki i opisów semantycznych.

Iterator formalizuje sposób poruszania się po dowolnej kolekcji danych.

Mediator definiuje, w jaki sposób można uprościć komunikację pomiędzy obiektami z użyciem osobnego obiektu, zapobiegając jawnym odwołaniom między nimi.

Typy wzorców behawioralnych II

Observer Obserwator - definiuje sposób powiadamiania o zmianach stanu obiektu.

State Stan - umożliwia obiektowi zmianę jego zachowania w wyniku jego stanu wewnętrznego.

Strategy Strategia - definiuje rodzinę obudowanych algorytmów i umożliwia ich wymianę.

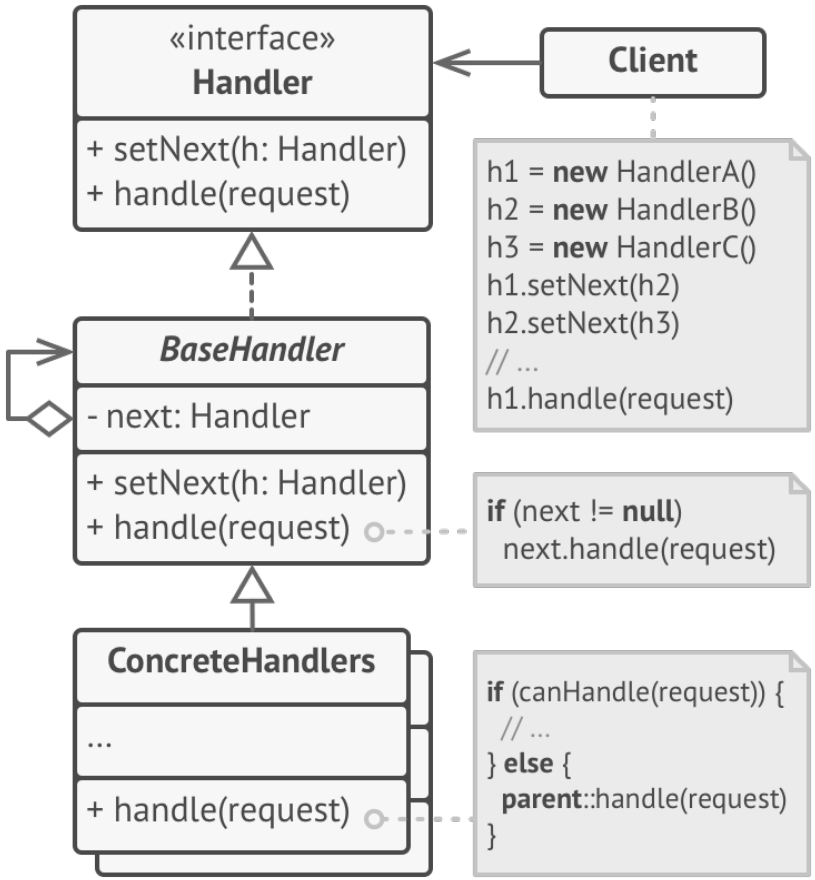
Template Method Metoda szablonowa - dostarcza abstrakcyjnej definicji algorytmu.

Visitor Wizytator - pozwala na zdefiniowanie nowej operacji bez zmian w klasach elementów, do których operacja się odnosi.

Łańcuch odpowiedzialności

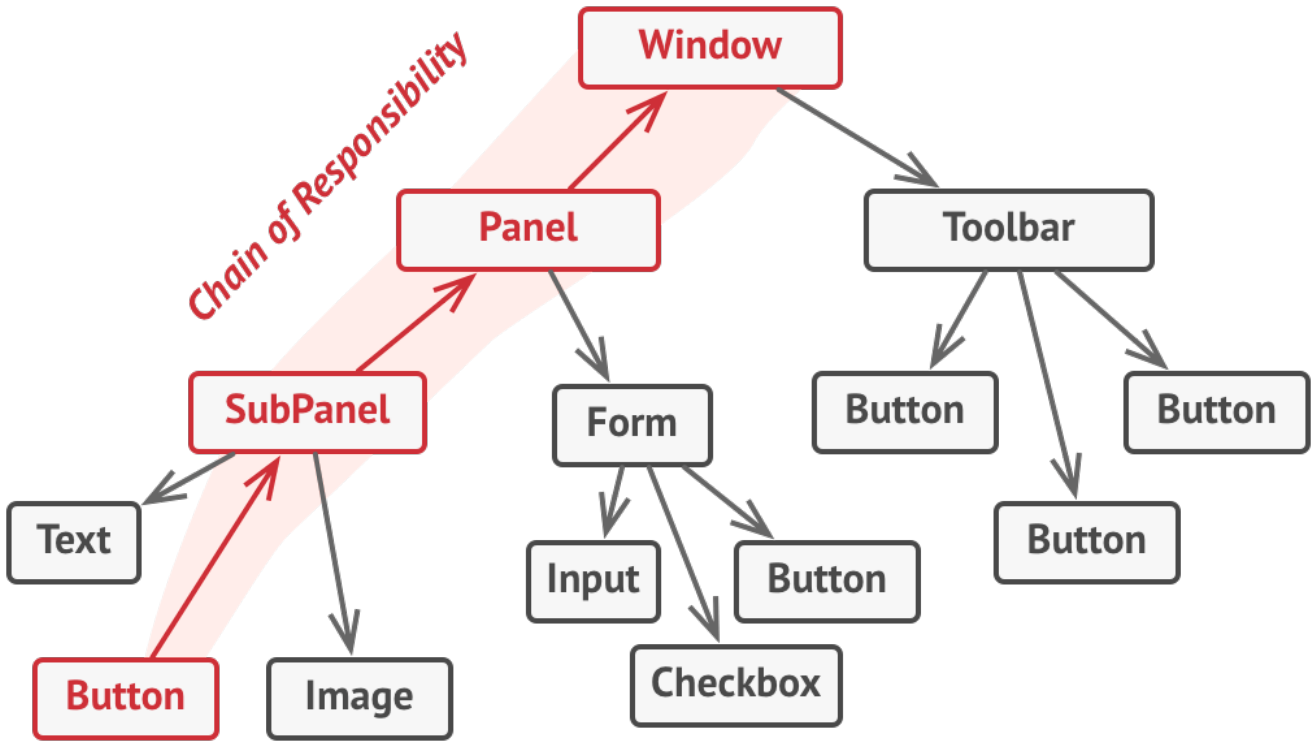
- Łańcuch odpowiedzialności pozwala przekazać żądanie wzdłuż łańcucha połączonych elementów.
- Po otrzymaniu żądania obiekt decyduje czy zrealizować żądanie czy przekazać je do następnego elementu w łańcuchu.
- Wzorzec stosowany przy przekazywaniu zdarzeń w GUI.

Schemat Łańcucha odpowiedzialności



Rysunek 21: *Schemat Łańcucha odpowiedzialności [Refactoring.Guru, 2019]*

Przykład Łańcucha odpowiedzialności

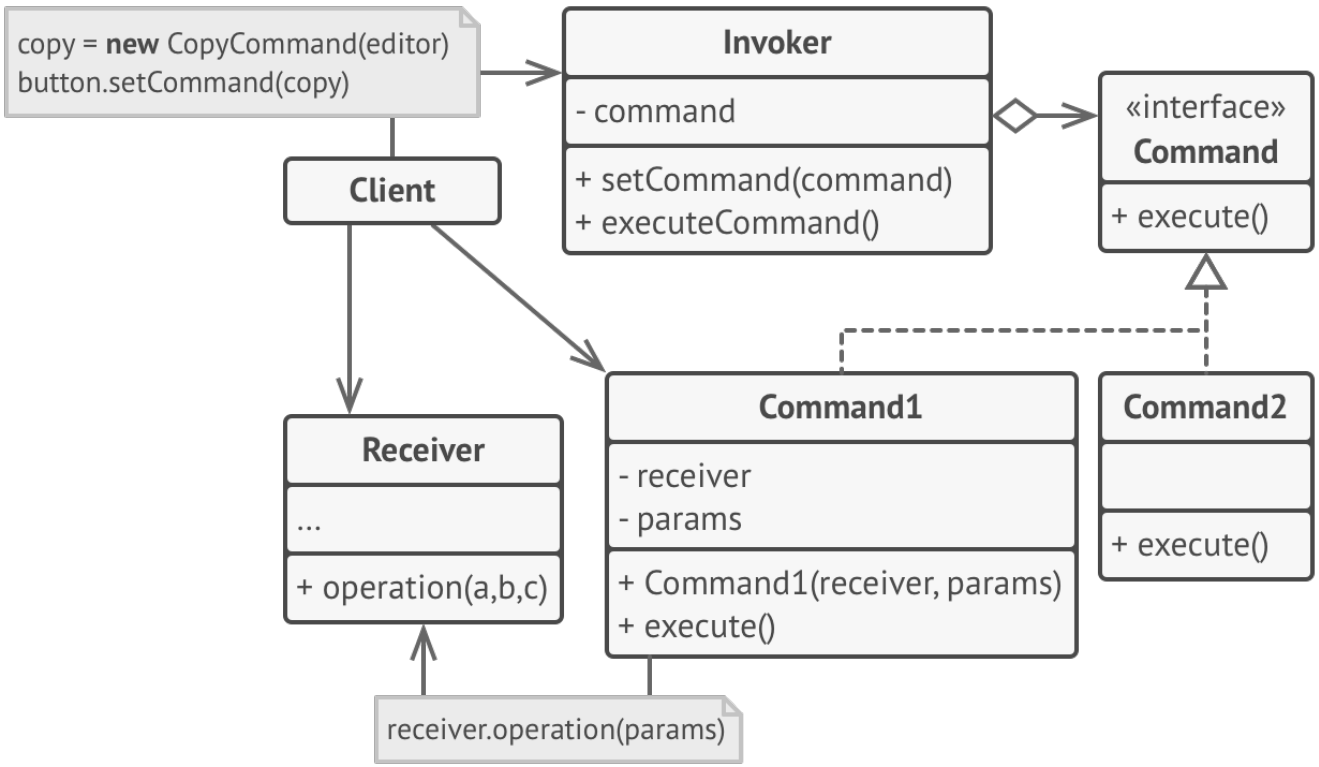


Rysunek 22: *Przykład Łańcucha odpowiedzialności [Refactoring.Guru, 2019]*

Polecenie

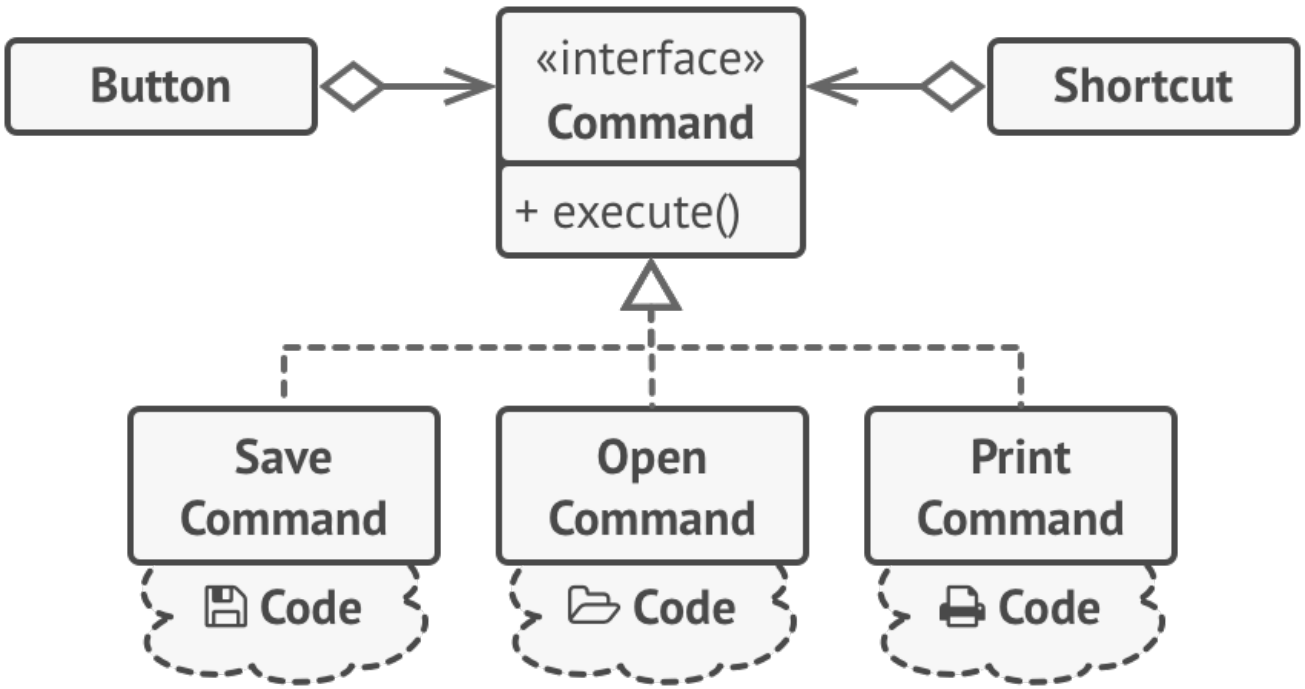
- Polecenie jest wzorcem przekształcającym żądanie w autonomiczny obiekt.
- Stworzony obiekt zawiera komplet informacji o żądaniu.
- Transformacja pozwala na parametryzowanie metod różnymi żądaniami, opóźniać i kolejkować realizację żądań i wspiera odwracalność operacji.
- W Javie wzorzec jest realizowany przez klasę `Action`.
- Implementacja pozwala na przypisanie do wielu działań wyzwalających tego samego zachowania oraz na ujednolicenie sposobu prezentacji.

Schemat Polecenia



Rysunek 23: *Schemat Polecenia [Refactoring.Guru, 2019]*

Przykład Polecenia

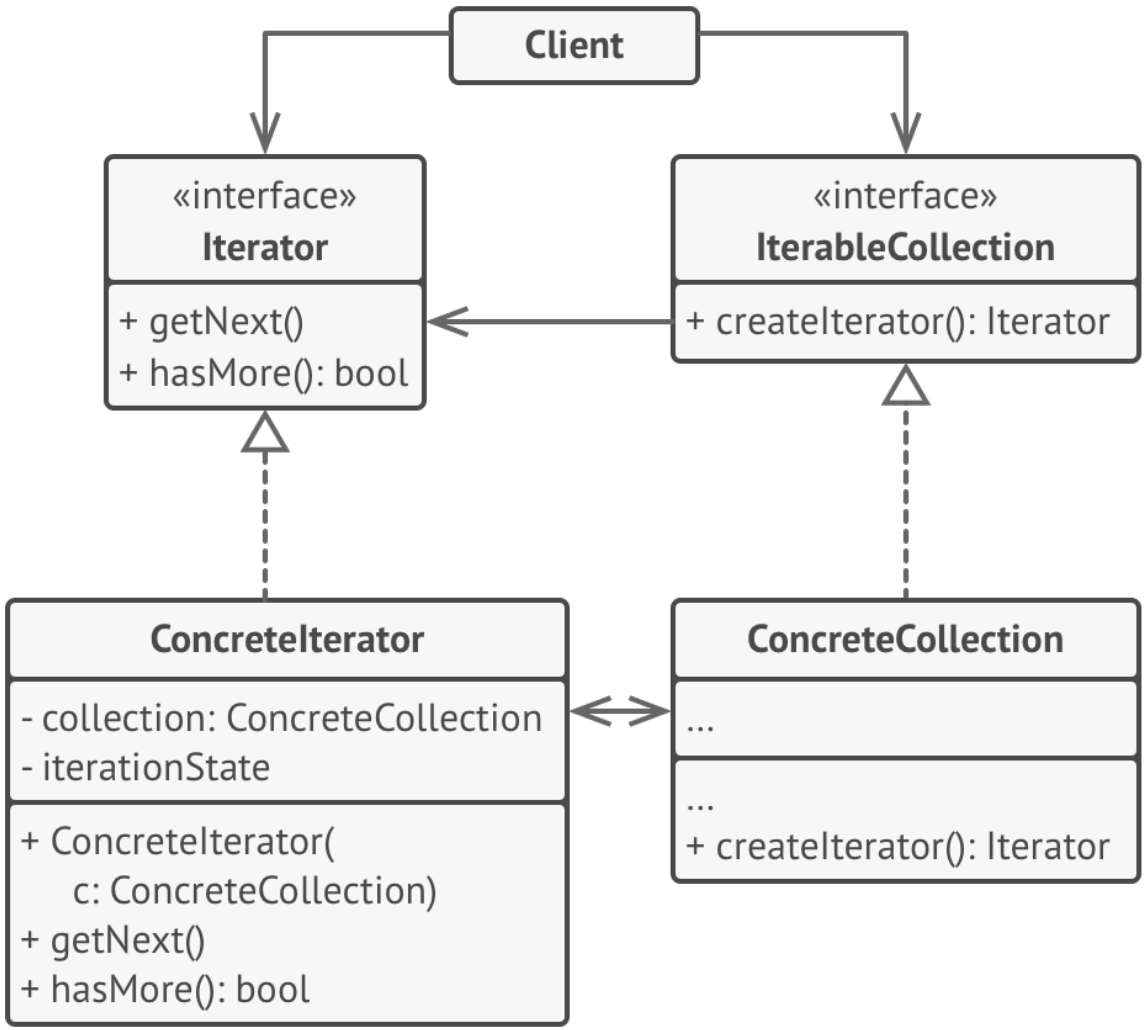


Rysunek 24: *Przykład Polecenia [Refactoring.Guru, 2019]*

Iterator

- Iterator jest wzorcem umożliwiającym przechodzenie po kolekcjach obiektów.
- Działa niezależnie od struktury danych, może być stosowany zarówno do list jak i do drzew czy stosów.
- Może pozwolić przeglądać struktury na różne sposoby np Deep-first, Breadth-first.
- W Javie wszystkie kolekcje `Collection` są wyposażone w iteratory.

Schemat Iteratora

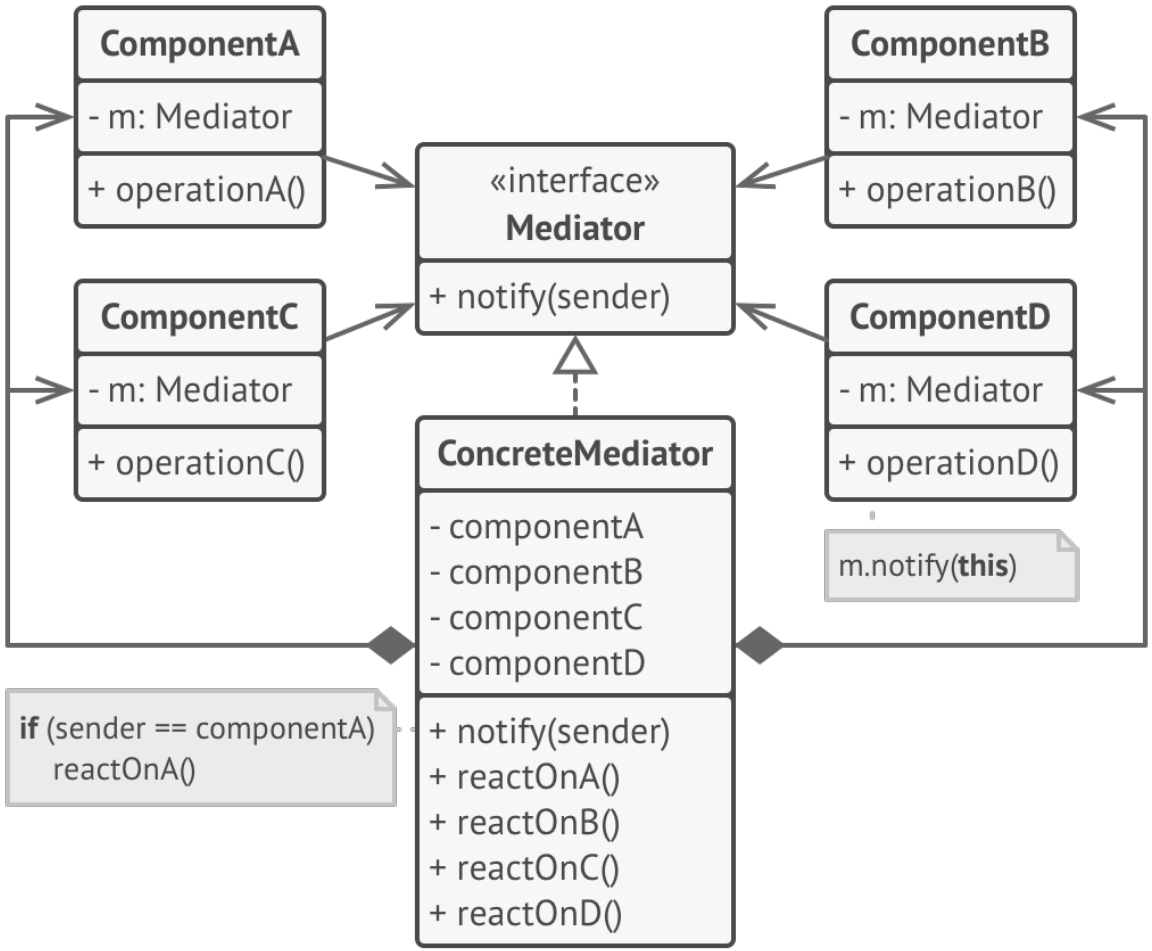


Rysunek 25: *Schemat Iteratora [Refactoring.Guru, 2019]*

Mediator

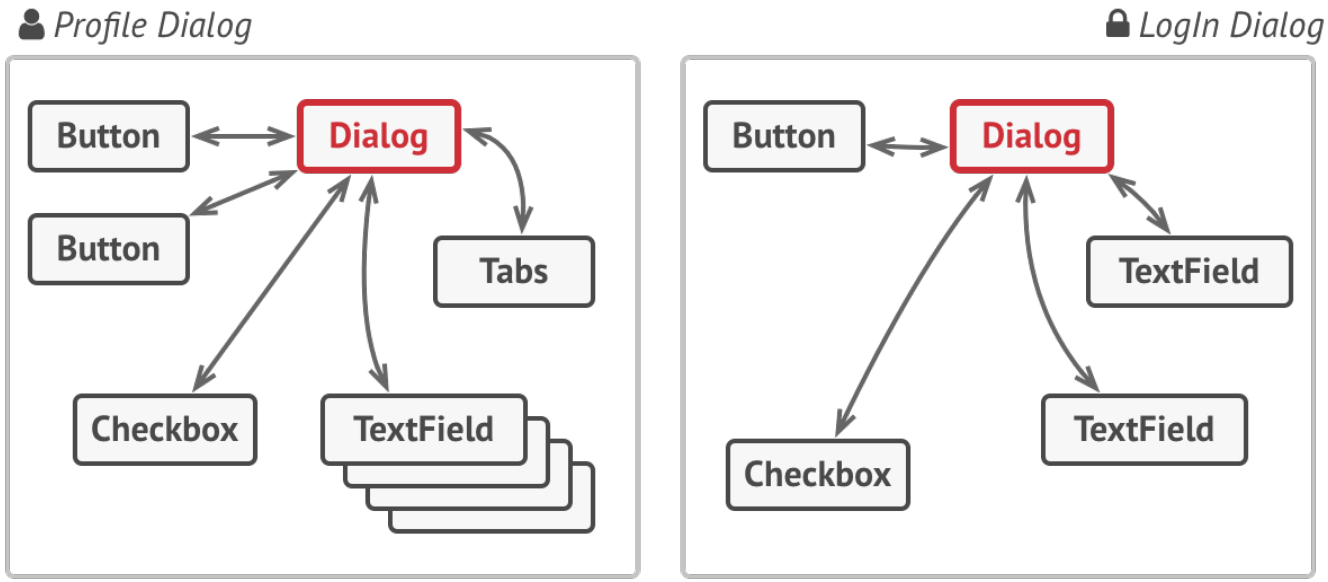
- Mediator ma na celu skanalizowanie komunikacji między obiektami.
- Zabrania bezpośredniej komunikacji między obiektami.
- Wymusza współpracę między obiektami tylko i wyłącznie za pomocą obiektu Mediatora.
- W Javie tworzymy Mediatora korzystając z implementacji interfejsu `ActionListener`.
- W przypadku GUI rolę Mediatora dla kontrolek pełnią kontenery np. `JDialog`

Schemat Mediatora



Rysunek 26: *Schemat Mediatora [Refactoring.Guru, 2019]*

Przykład Mediatora

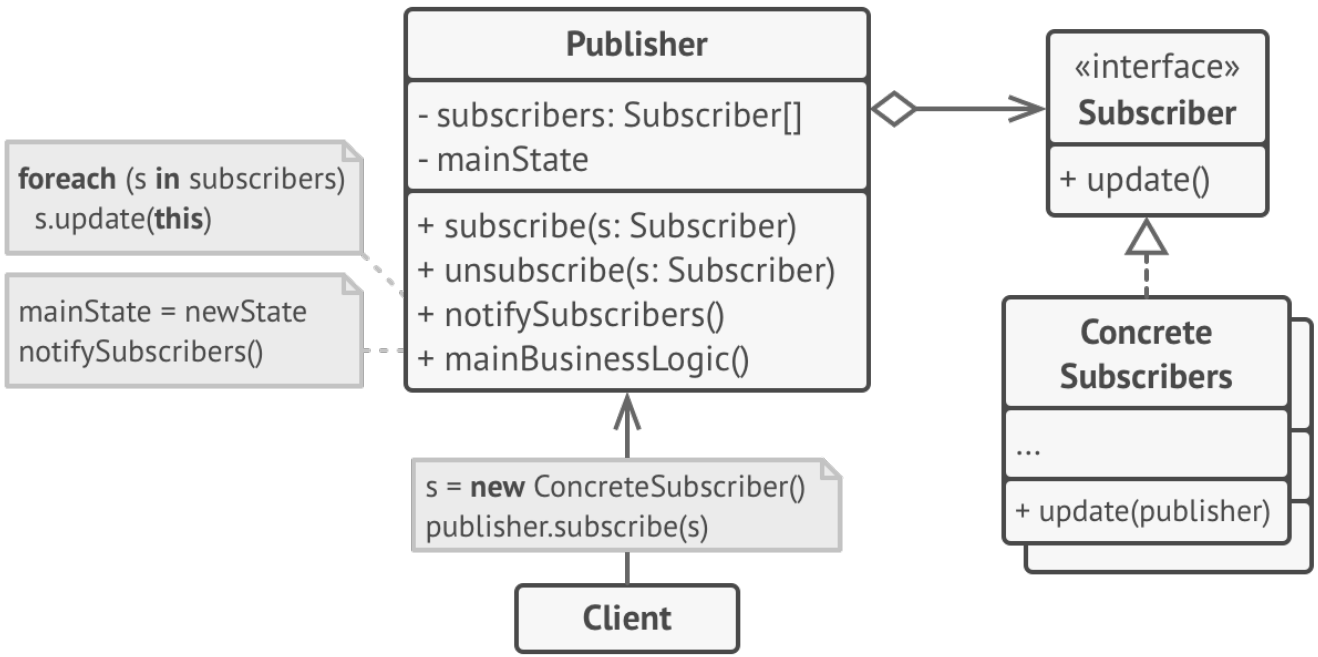


Rysunek 27: *Przykład Mediatora [Refactoring.Guru, 2019]*

Obserwator

- Wzorzec Obserwatora pozwala stworzyć mechanizm subskrybcji.
- Mechanizm pozwala na notyfikację wielu obiektów o zdarzeniach, które obserwują.
- W Javie mechanizm Obserwatora jest realizowany przez mechanizmy `wait` i `notify` klasy `Object`.
- Niezależnie od tego mechanizmu istnieje klasa `Observable`, która pozwala na powiadomienie wszystkich obserwujących.

Schemat Obserwatora

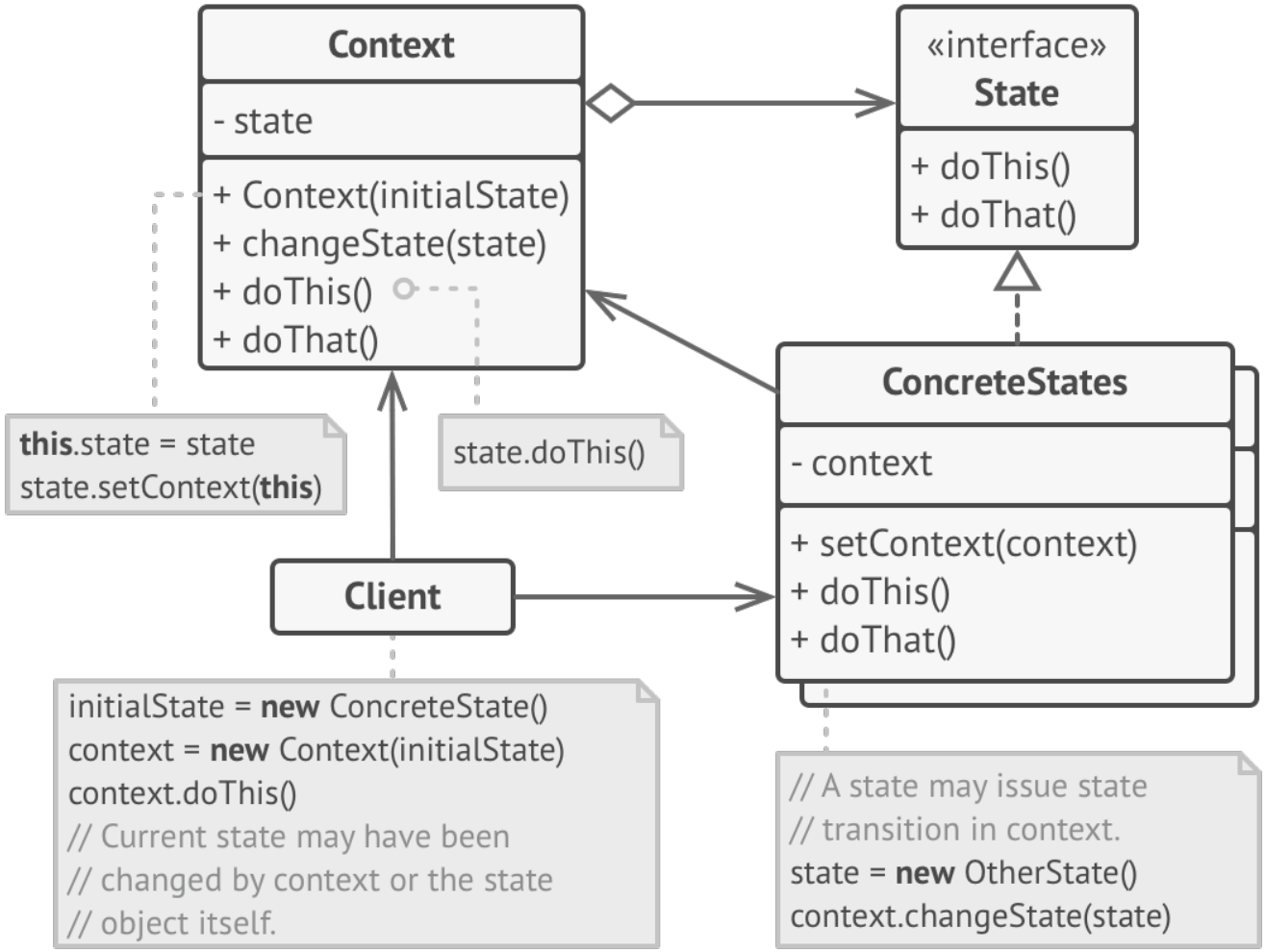


Rysunek 28: *Schemat Obserwatora [Refactoring.Guru, 2019]*

Stan

- Wzorzec Stanu pozwala obiektowi obserwować swój własny stan.
- W momencie zmiany stanu obiekt może reagować i zmieniać swoje zachowanie w zależności od stanu.
- Może być to użyte do symulacji zmiany klasy obiektu.
- W Javie wzorzec pozwala na wyeliminowanie stosowania instrukcji warunkowych w klasie, aby zróżnicować jej zachowanie w zależności od argumentów.

Schemat Stanu

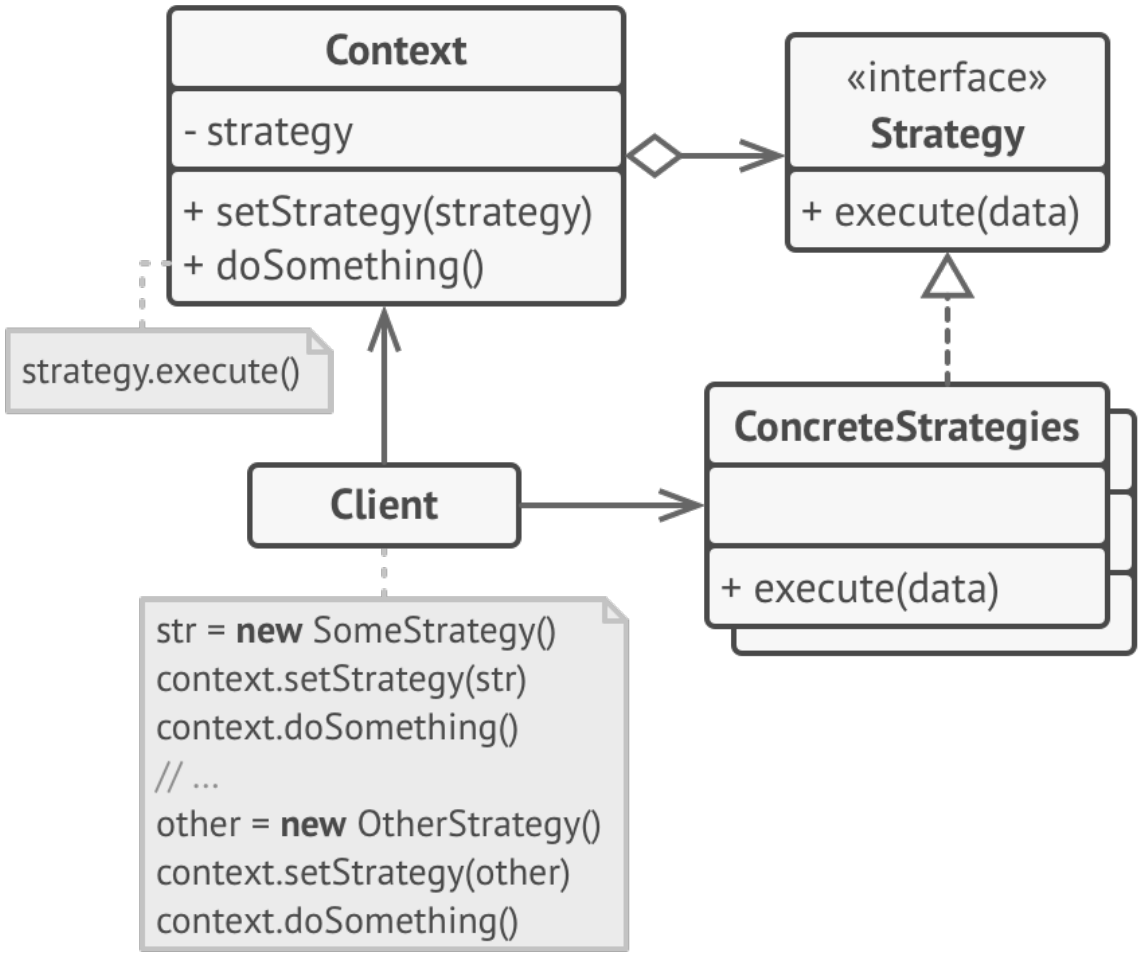


Rysunek 29: *Schemat Stanu [Refactoring.Guru, 2019]*

Strategia

- Strategia pozwala zdefiniować rodzinę algorytmów.
- Wstawia każdy z nich do osobnej klasy i traktuje ich obiekty wymiennie.

Schemat Strategii



Rysunek 30: *Schemat Strategii [Refactoring.Guru, 2019]*

Strategia w Javie 8

- W Javie 8 możemy użyć predefiniowanych interfejsów do opisania różnych algorytmów.

```
interface Computation<T> {
  public T compute(T n, T m);
}
```

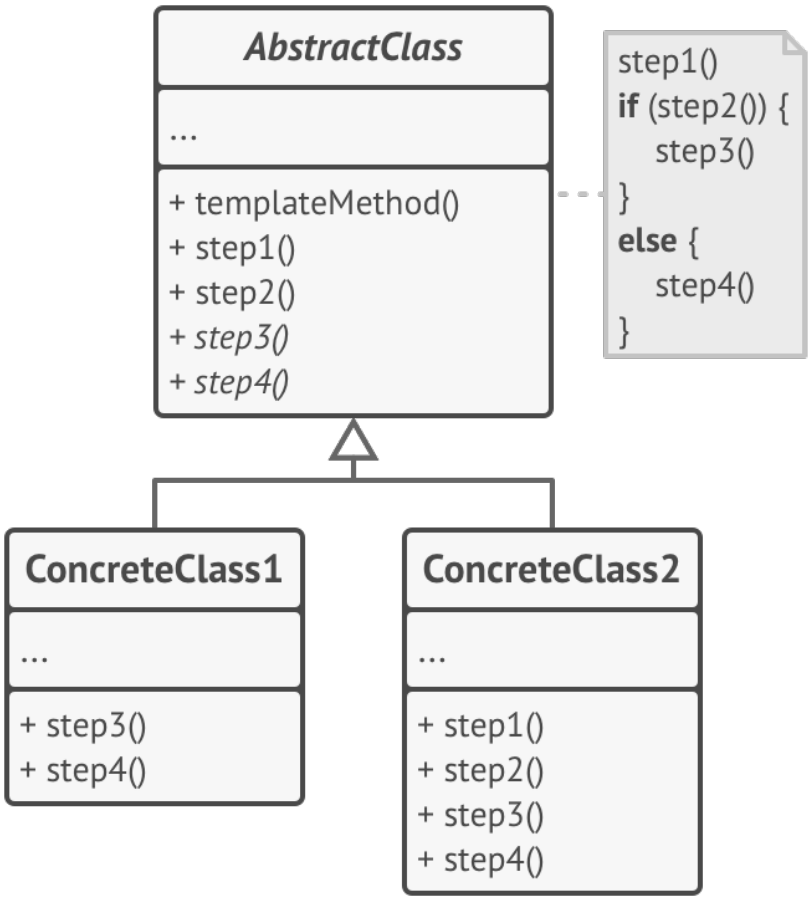
- Ze zbioru algorytmów `ArrayList<Computation> algoritms` możemy utworzyć strumień, który będzie procesowany element po elemencie

```
algoritms.stream().reduce(v -> v,
    Computation::compute)
```

Metoda szablonowa

- Metoda szablonowa definiuje szkielet algorytmu.
- Definicja zawarta jest klasie nadrzędnej.
- Klasy podrzędne mają możliwość nadpisania poszczególnych kroków algorytmu bez zmiany jego struktury.

Schemat Metody szablonowej



Rysunek 31: *Schemat Metody szablonowej [Refactoring.Guru, 2019]*

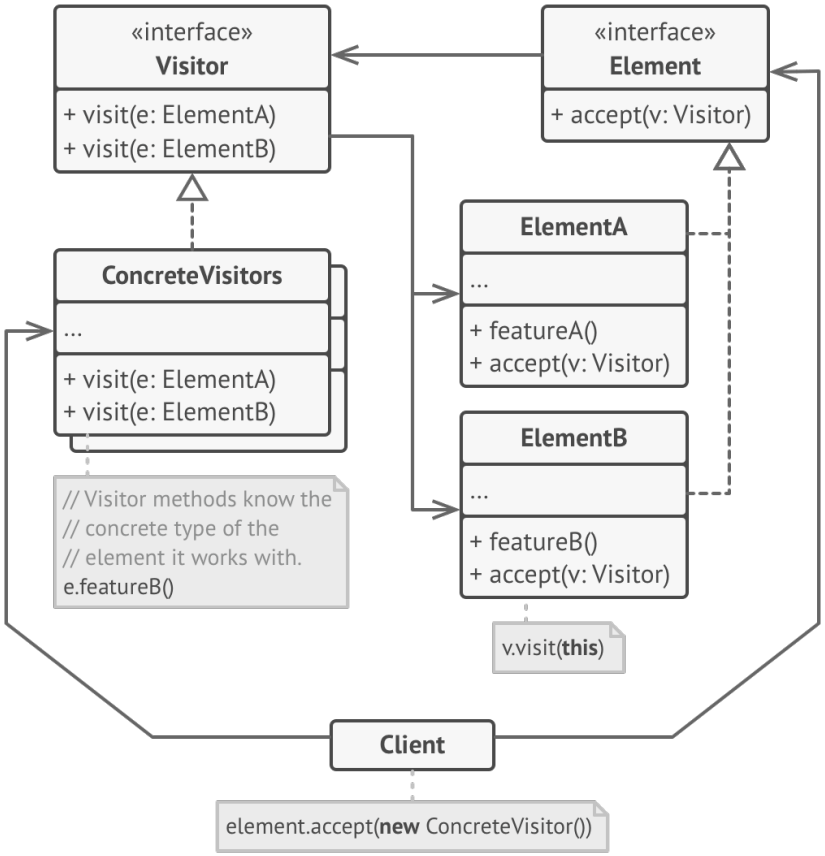
Metody szablonowe w Javie

- Java w szerokim zakresie wykorzystuje metodę szablonową.
- Wiele hierarchii klas zaczyna się od klasy abstrakcyjnej, której klasy pochodne implementują szablon.
- Przykładami są klasy opisujące modele danych `AbstractTableModel` i `AbstractListModel`.

Wizytator

- Wizytator pozwala na odseparowanie algorytmu od obiektów na których operuje.
- Ponieważ nie ma możliwości analizy prywatnych danych przez klasę zewnętrzną klasy wizytowane muszą implementować metodę `accept`, która akceptuje działanie wizytatora

Schemat Wizytatora



Rysunek 32: *Schemat Wizytatora [Refactoring.Guru, 2019]*

Bibliografia

[Alpert et al., 1998] Alpert, S. R., Brown, K., and Woolf, B. (1998).
The Design Patterns Smalltalk Companion.
Addison-Wesley.

[Gamma et al., 1993] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1993).
Design Patterns: Abstraction and Reuse of Object-Oriented Design, pages 361–388.
Springer Berlin Heidelberg, Berlin, Heidelberg.

[Krasner and Pope, 1988] Krasner, G. E. and Pope, S. T. (1988).
A cookbook for using the model-view controller user interface paradigm in Smalltalk-80.
Journal of Object Oriented Programming, 1(3):26–49.

[Refactoring.Guru, 2019] Refactoring.Guru (2019).
Design patterns.

[Wikipedia, 2019] Wikipedia (2019).
Model-view-controller.