Wzorce projektowe
Opracowanie

Spis treści

1 Na podstawie **5**

2 Podział wzorców projektowych **6**

3 Podstawy diagramów klas UML **7**
3.1 Klasa . . . 7
3.1.1 Widoczność atrybutów klasy . . . 7
3.1.2 Atrybuty statyczne i typy atrybutów . . . 8
3.1.3 Widoczność metod klasy . . . 8
3.1.4 Metody statyczne, typy argumentów i typ zwracany . . . 8
3.2 Interfejsy . . . 9
3.3 Relacje . . . 9
3.3.1 Dziedziczenie (generalizacja) . . . 9
3.3.2 Agregacja . . . 10
3.3.3 Kompozycja . . . 10
3.3.4 Realizacja . . . 10
3.3.5 Asocjacja . . . 11

4 Wzorce kreacyjne **12**
4.1 Builder . . . 12
4.1.1 Charakterystyka . . . 12
4.1.2 Zastosowanie . . . 12
4.1.3 UML . . . 12
4.1.4 Kod . . . 12
4.1.5 Zalety i wady . . . 12
4.1.6 Porównanie do innych wzorców TODO . . . 13
4.2 Prototype . . . 14
4.2.1 Charakterystyka . . . 14
4.2.2 Zastosowanie . . . 14
4.2.3 UML . . . 14
4.2.4 Kod . . . 14
4.2.5 Zalety i wady . . . 14
4.2.6 Porównanie do innych wzorców TODO . . . 15
4.3 Factory Method . . . 16
4.3.1 Charakterystyka . . . 16
4.3.2 Zastosowanie . . . 16
4.3.3 UML . . . 16
4.3.4 Kod . . . 16
4.3.5 Zalety i wady . . . 16
4.3.6 Porównanie do innych wzorców TODO . . . 17
4.4 Abstract Factory . . . 18
4.4.1 Charakterystyka . . . 18
4.4.2 Zastosowanie . . . 18

4.4.3 UML . . . 18
4.4.4 Kod . . . 18
4.4.5 Zalety i wady . . . 18
4.4.6 Porównanie do innych wzorców TODO . . . 19

5 Wzorce strukturalne **20**
5.1 Adapter . . . 20
5.1.1 Charakterystyka . . . 20
5.1.2 Zastosowanie . . . 20
5.1.3 UML . . . 20
5.1.4 Kod . . . 21
5.1.5 Zalety i wady . . . 21
5.1.6 Porównanie do innych wzorców TODO . . . 21
5.2 Bridge . . . 22
5.2.1 Charakterystyka . . . 22
5.2.2 Zastosowanie . . . 22
5.2.3 UML . . . 22
5.2.4 Kod . . . 22
5.2.5 Zalety i wady . . . 22
5.2.6 Porównanie do innych wzorców TODO . . . 23
5.3 Composite . . . 24
5.3.1 Charakterystyka . . . 24
5.3.2 Zastosowanie . . . 24
5.3.3 UML . . . 24
5.3.4 Kod . . . 24
5.3.5 Zalety i wady . . . 24
5.3.6 Porównanie do innych wzorców TODO . . . 24
5.4 Flyweight . . . 25
5.4.1 Charakterystyka . . . 25
5.4.2 Zastosowanie . . . 25
5.4.3 UML . . . 25
5.4.4 Kod . . . 25
5.4.5 Zalety i wady . . . 25
5.4.6 Porównanie do innych wzorców TODO . . . 25
5.5 Proxy . . . 26
5.5.1 Charakterystyka . . . 26
5.5.2 Zastosowanie . . . 26
5.5.3 UML . . . 26
5.5.4 Kod . . . 26
5.5.5 Zalety i wady . . . 26
5.5.6 Porównanie do innych wzorców TODO . . . 27
5.6 Decorator . . . 28
5.6.1 Charakterystyka . . . 28
5.6.2 Zastosowanie . . . 28
5.6.3 UML . . . 28
5.6.4 Kod . . . 28
5.6.5 Zalety i wady . . . 28
5.6.6 Porównanie do innych wzorców TODO . . . 29
5.7 Facade . . . 30
5.7.1 Charakterystyka . . . 30

5.7.2 Zastosowanie . . . 30
5.7.3 UML . . . 30
5.7.4 Kod . . . 30
5.7.5 Zalety i wady . . . 30
5.7.6 Porównanie do innych wzorców TODO . . . 31

6 Wzorce behawioralne **32**
6.1 Obserwator (Observer) . . . 32
6.1.1 Charakterystyka . . . 32
6.1.2 Zastosowanie . . . 32
6.1.3 UML . . . 32
6.1.4 Kod . . . 32
6.1.5 Zalety i wady . . . 32
6.1.6 Porównanie do innych wzorców . . . 33
6.2 Chain of responsibility . . . 34
6.2.1 Charakterystyka . . . 34
6.2.2 Zastosowanie . . . 34
6.2.3 UML . . . 34
6.2.4 Kod . . . 34
6.2.5 Zalety i wady . . . 34
6.2.6 Porównanie do innych wzorców . . . 35
6.3 Command . . . 36
6.3.1 Charakterystyka . . . 36
6.3.2 Zastosowanie . . . 36
6.3.3 UML . . . 37
6.3.4 Kod . . . 37
6.3.5 Zalety i wady . . . 37
6.3.6 Porównanie do innych wzorców . . . 37
6.4 Strategy . . . 39
6.4.1 Charakterystyka . . . 39
6.4.2 Zastosowanie . . . 39
6.4.3 UML . . . 39
6.4.4 Kod . . . 39
6.4.5 Zalety i wady . . . 39
6.4.6 Porównanie do innych wzorców . . . 40
6.5 Template Method . . . 41
6.5.1 Charakterystyka . . . 41
6.5.2 Zastosowanie . . . 41
6.5.3 UML . . . 41
6.5.4 Kod . . . 41
6.5.5 Zalety i wady . . . 41
6.5.6 Porównanie do innych wzorców . . . 42
6.6 State (State machine) . . . 43
6.6.1 Charakterystyka . . . 43
6.6.2 Zastosowanie . . . 43
6.6.3 UML . . . 43
6.6.4 Kod . . . 43
6.6.5 Zalety i wady . . . 43
6.6.6 Porównanie do innych wzorców TODO . . . 44
6.7 Visitor . . . 45

6.7.1 Charakterystyka 45
6.7.2 Zastosowanie 45
6.7.3 UML 45
6.7.4 Kod 45
6.7.5 Zalety i wady 46
6.7.6 Porównanie do innych wzorców TODO 46

7 Szablon na opracowanie pojedynczego wzorca **47**
7.1 Nazwa wzorca 47
7.1.1 Charakterystyka 47
7.1.2 Zastosowanie 47
7.1.3 UML 47
7.1.4 Kod 47
7.1.5 Zalety i wady 47
7.1.6 Porównanie do innych wzorców TODO 47

1 Na podstawie

- https://refactoring.guru/design-patterns/ - najlepsze znalezione źródło do wzorców projektowych
- wzorce projektowe, opracowanie 2019 z wiki - napisane na podstawie powyższego linka
- Książka - Designed Patterns exaplined simply
- https://www.journaldev.com/1827/java-design-patterns-example-tutorial
- Książka - Head first Design Patterns

2 Podział wzorców projektowych

Pytanie

Na jakie rodzaje dzielimy wzorce projektowe ?

Wzorce projektowe dzielimy na:

- Kreacyjne - opisują, w jaki sposób obiekty są tworzone, zapewniają sposoby na instancjację obiektów w najlepszy możliwy sposób w danej sytuacji
- Behawioralne - opisują zachowanie obiektów, w jaki sposób obiekty komunikują się ze sobą
- Strukturalne - opisują sposób, w jaki obiekty są zbudowane, definiują jak klasy i interfejsy mają być zbudowane w celu realizacji pewnych działań
- Architektoniczne - opisują oprogramowanie na bardziej abstrakcyjnym poziomie, skupiają się na architekturze rozwiązań raczej niż na ich szczegółach

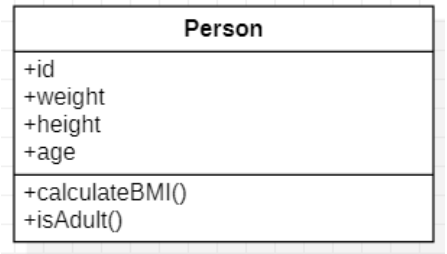
3 Podstawy diagramów klas UML

Przy rozważaniach o wzorcach często stosujemy notację UML w celu przedstawienia jak dany wzorzec jest implementowany, konieczne jest więc abyśmy poznali przynajmniej jakieś podstawy tej notacji. Ograniczymy się do niezbędnego minimum z diagramów klas.

Podstawowe komponenty z których budujemy nasz diagram klas to klasy i interfejsy, które rozumiemy w takim sensie jak są rozumiane w językach programowania.

3.1. Klasa

Klasa składa się z trzech obszarów, pierwszy z nich jest przeznaczony na nazwę klasy, środkowy to miejsce na atrybuty klasy, a na dole umieszczamy metody klasy. Przykładowo:



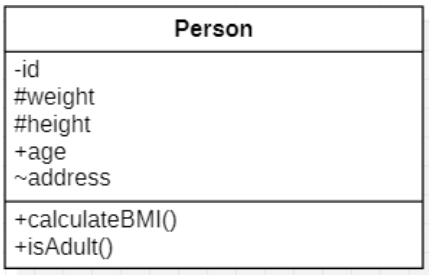
Rysunek 1: Przykładowa reprezentacja klasy w UML

3.1.1. Widoczność atrybutów klasy

Na diagramie klas możemy uwzględnić widoczność atrybutów i metod (private, protected, itd.). Widoczność ta jest oznaczana symbolem poprzedzającym nazwę atrybutu w następujący sposób:

- + (public)
- - (private)
- # (protected)
- ~(default, package)
- / (do reprezentacji atrybutu, który został odziedziczony)

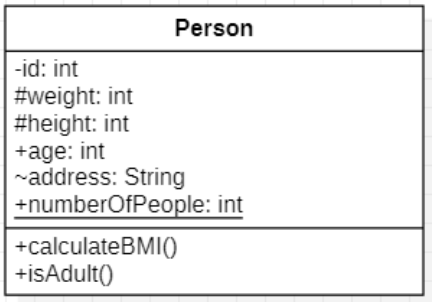
Pozmieniajmy więc widoczność paru pól w naszej klasie Person:



Rysunek 2: Uwzględnienie widoczności atrybutów

3.1.2. Atrybuty statyczne i typy atrybutów

Na diagramie klas możemy także uwzględnić atrybuty statyczne, reprezentujemy je przez podkreślenie atrybutu, a także typy atrybutów, te reprezentujemy po nazwie atrybutu i dwukropku. Typy mogą się oczywiście różnić pomiędzy językami, możemy wybrać ten w którym aktualnie kodujemy lub przyjąć jakieś ogólnie rozumiane typy. Dodajemy więc atrybut statyczny oraz typy do naszej klasy Person.

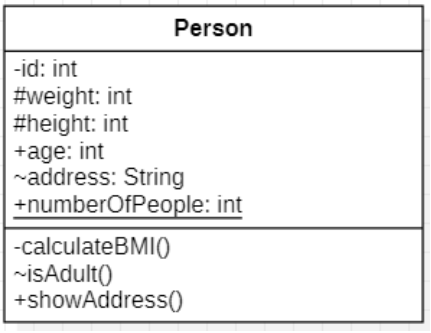


Rysunek 3: Uwzględnienie atrybutów statycznych oraz typów atrybutów.

3.1.3. Widoczność metod klasy

W analogiczny sposób do atrybutów możemy dodawać informacje o widoczności do naszych metod:

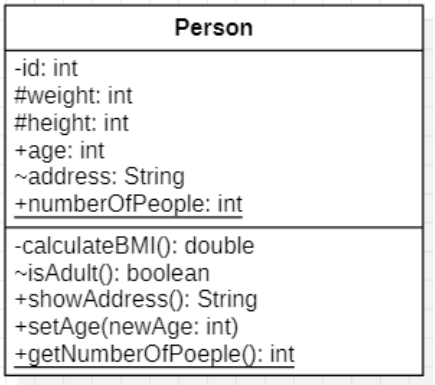
- + (public)
- - (private)
- # (protected)
- ~(default, package)



Rysunek 4: Uwzględnienie atrybutów statycznych oraz typów atrybutów.

3.1.4. Metody statyczne, typy argumentów i typ zwracany

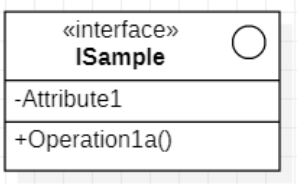
Analogicznie jak dla atrybutów możemy oznaczyć w klasie metodę statyczną poprzez jej podkreślenie. Typy argumentów wejściowych zaznaczamy tak samo jak typy atrybutów, tj. po dwukropku. Typ zwracany przez metodę zapisujemy po dwukropku po nazwie metody, jeśli typem tym jest void to nic nie zapisujemy. Dodajmy więc nowe możliwości do naszej klasy Person:



Rysunek 5: Uwzględnienie metod statycznych, typów argumentów i typu zwracanego

3.2. Interfejsy

Interfejsy różnią się od klas jedynie tym, że w pierwszej sekcji zawierają dodatkowo przed nazwą wyrażenie «interface» oznaczające, że dany komponent jest interfejsem. Poza tym, metody i atrybuty definiujemy analogicznie jak dla klas. W programie STAR UML interfejs ma dodatkowo w reprezentacji graficznej kółko jak widać na obrazku poniżej, jednak w ogólności nie jest ono zawierane w interfejsach na diagramach UML.



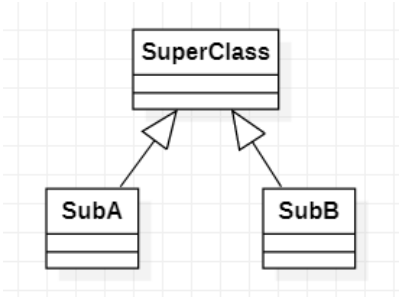
Rysunek 6: Przykład interfejsu

3.3. Relacje

Relacje służą nam do opisu zależności pomiędzy interfejsami i klasami. Możemy wyrażać wszystkie relacje, które typowo stosujemy w programowaniu, tj. agregacje, implementacje, dependencje itd.

3.3.1. Dziedziczenie (generalizacja)

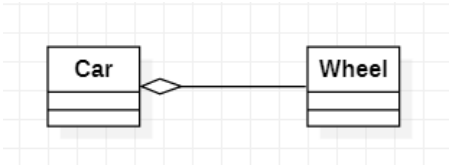
Dziedziczenie jest relacją, którą znamy z programowania obiektowego. Na diagramie UML jest reprezentowane strzałką w pustym grotem, strzałka zawsze wskazuje na superklase.



Rysunek 7: Relacja dziedziczenia na diagramie UML

3.3.2. Agregacja

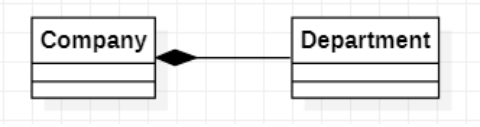
To relacja, która informuje nas, że dana klasa jest częścią innej klasy, ale również może istnieć niezależnie od tej klasy. Przykładowo koło od samochodu może istnieć zanim jeszcze samochód zostanie wyprodukowany. Można więc powiedzieć, że koło istnieje niezależnie od samochodu, pomimo, że każdy samochód ma koła. Taką relacje modelujemy przy pomocy agregacji, czyli strzałki z grotem w kształcie pustego rombu. Grot ten jest zawsze skierowany w stronę klasy, która zawiera inne klasy.



Rysunek 8: Przykład agregacji

3.3.3. Kompozycja

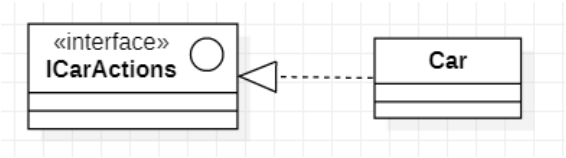
To relacja bardzo podobna do agregacji z tą różnicą, że obiekt który jest częścią innego nie może istnieć niezależnie, tj. jeśli obiekt, który zawiera inne obiekty zostaje zniszczony, również wszystkie obiekty w nim zawarte zostają zniszczone. Dodatkowo, każdy z obiektów stanowiących części innego ma przypisany dokładnie jeden taki obiekt. Przykładem takiej relacji może być firma i jej działy. Każda firma ma wiele działów, jednak nie ma sensu mówić o działach bez kontekstu firmy. Relację taką reprezentujemy tak jak relację agregacji z tą różnicą, że tym razem romb jest wypełniony.



Rysunek 9: Przykład kompozycji

3.3.4. Realizacja

To relacja, która odpowiada relacji implementacji interfejsu jaką znamy z języków programowania. Relacja tą oznaczamy strzałką przerywaną z pustym grotem (podobnie jak dziedziczenie, ale tutaj strzałka jest przerywana, a nie ciągła). Strzałka zawsze skierowana jest w stronę interfejsu, który implementujemy.



Rysunek 10: Przykład realizacji

3.3.5. Asocjacja

To relacja, która informuje nas, że dana klasa korzysta z funkcjonalności innej klasy lub komunikuje się z nią. Może ona być jednokierunkowa (strzałka z cienkim grotem) lub dwukierunkowa (prosta linia). Asocjacja może też wyrażać, że dana klasa zawiera inną klasę (tak jak agregacja czy kompozycja). Jaka jest więc różnica pomiędzy asocjacją, a agregacją i kompozycją ? Otóż asocjacja jest terminem bardziej ogólnym, może wyrażać więcej niż tylko zawieranie, ale także komunikacje, korzystanie z funkcjonalności itp. Dlatego do zawierania się klas w klasach raczej należy stosować agregację lub kompozycje.

Rysunek 11: Asocjacja dwukierunkowa

4 Wzorce kreacyjne

4.1. Builder

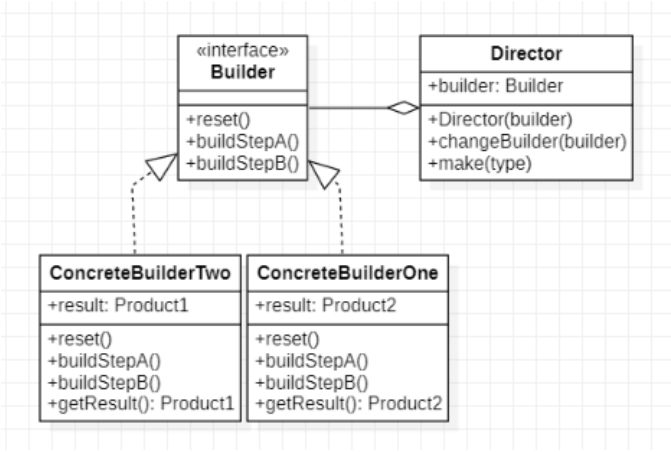
4.1.1. Charakterystyka

To wzorzec kreacyjny, który pozwala na konstruowanie złożonych obiektów krok po kroku. Wzorzec ten pozwala na tworzenie różnych reprezentacji obiektów tej samej klasy w zależności od potrzeb klienta.

4.1.2. Zastosowanie

- Zastąpienie w kodzie konstruktorów z bardzo dużą liczbą parametrów lub takich, które wywołują inne konstruktory
- W sytuacji gdy chcemy w kodzie tworzyć różne reprezentacje obiektów tej samej klasy
- W przypadku gdy tworzymy w kodzie bardzo złożone obiekty - o wielu polach lub o specyficznej strukturze

4.1.3. UML



Rysunek 12: Wzorzec Builder

4.1.4. Kod

Github

4.1.5. Zalety i wady

- Zalety
  - Obiekty tworzone są krok po kroku, kroki mogą być pomijane lub wywoływane rekurencyjnie
  - Odizolowanie skomplikowanej logiki tworzenia obiektów od logiki działania obiektu
- Wady
  - Wymaga utworzenia wielu nowych klas i interfejsów

4.1.6. Porównanie do innych wzorców TODO

-

4.2. Prototype

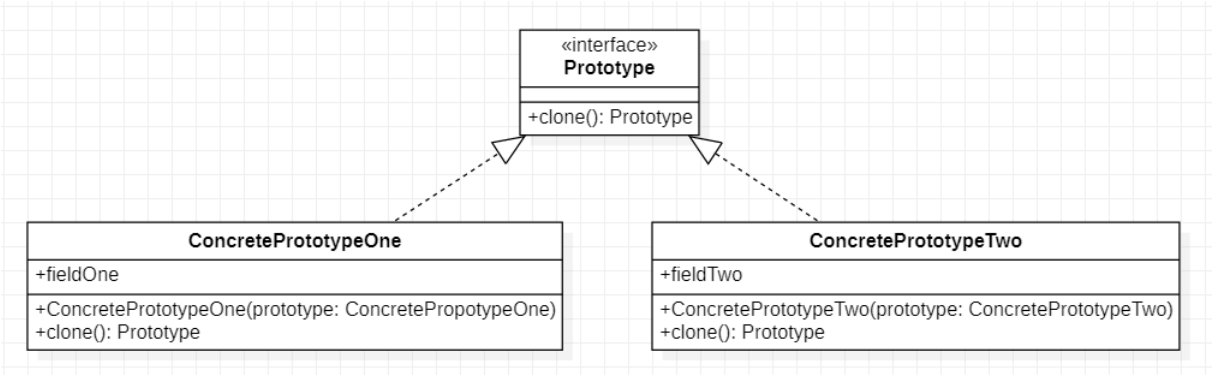
4.2.1. Charakterystyka

To kreacyjny wzorzec projektowy, który pozwala na klonowanie obiektów przy braku zależności od klas tych obiektów. Wzorzec ten przerzuca odpowiedzialność za kopiowanie obiektów na obiekty, które są kopiowane.

4.2.2. Zastosowanie

- W przypadku gdy chcemy móc kopiować obiekty bez bycia zależnym od klas tych obiektów
- Zamiast tworzyć podklasy, które różnią się tylko wartościami pól, możemy stworzyć obiekty o danej konfiguracji, a potem je klonować - klonowanie zamiast dziedziczenia
- Mechanizm implementowany przez ten wzorzec często jest implementowany w językach programowania do klonowania obiektów

4.2.3. UML



Rysunek 13: Wzorzec Prototype

4.2.4. Kod

Github

4.2.5. Zalety i wady

- Zalety
  - Możliwość klonowania obiektów bez wiedzy o ich konkretnych klasach
  - Pozbycie się powtarzalnego kodu inicjalizacji obiektów, zamiast pisać to samo można klonować
  - Tworzenie złożonych obiektów w wygodny sposób poprzez *clone*, zamiast ustawiać referencje
  - Alternatywa dla dziedziczenia dla obiektów o różnych konfiguracjach
- Wady
  - Klonowanie obiektów, które zawierają pętle referencji może być bardzo trudne

4.2.6. Porównanie do innych wzorców TODO

-

4.3. Factory Method

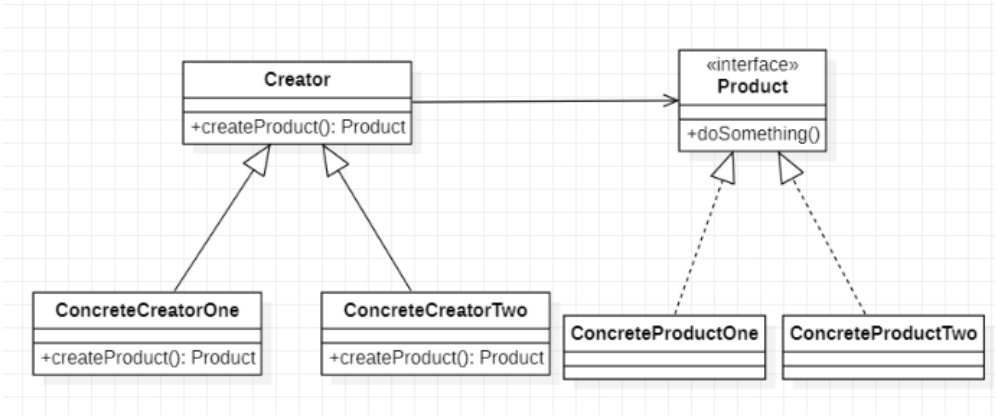
4.3.1. Charakterystyka

Factory Method to kreacyjny wzorzec projektowy, który w nadklasie definiuje interfejs metody do tworzenia obiektów, ale pozwala podklasom na określenie typu obiektów, które będą tworzone przez tą metodę.

4.3.2. Zastosowanie

- W przypadku gdy nie znamy wszystkich typów i zależności obiektów z którymi ma współpracować nasz kod
- W przypadku gdy chcemy dać możliwość klientom biblioteki rozszerzania jej wewnętrznych komponentów poprzez nadpisanie factoryMethod z klasy bazowej, która jest w bibliotece
- W przypadku gdy chcemy zaoszczędzić zasoby systemowe poprzez użycie istniejących instancji obiektów zamiast tworzenia nowych, logikę zwracania istniejących instancji zawieramy w factory method

4.3.3. UML



Rysunek 14: Wzorzec Factory Method

4.3.4. Kod

Github

4.3.5. Zalety i wady

- Zalety
  - Brak zależności od konkretnego typu obiektu
  - Kod tworzenia obiektów znajduje się w jednym miejscu
  - Łatwe wprowadzanie nowych typów obiektów do istniejącego kodu
- Wady
  - Wymaga wprowadzania do kodu wielu dodatkowych klas

4.3.6. Porównanie do innych wzorców TODO

-

4.4. Abstract Factory

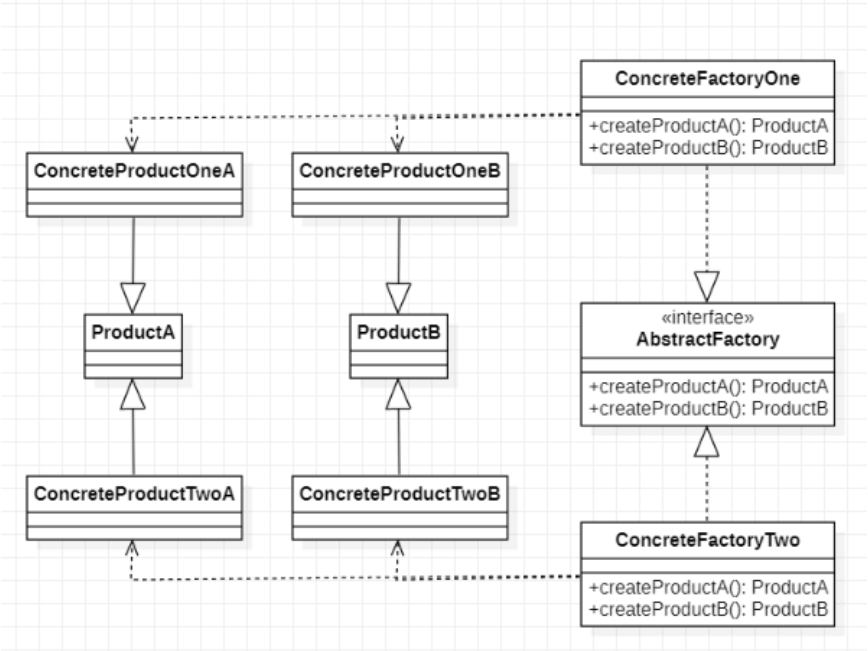
4.4.1. Charakterystyka

Fabryka abstrakcyjna to wzorzec kreacyjny, który pozwala na tworzenie grup powiązanych ze sobą pewnymi atrybutami obiektów z możliwością zmiany grupy, którą aktualnie tworzy fabryka. Każda z takich grup składa się z klas, które dziedziczą po pewnych ogólnych nadklasach. Dla każdej z takich grup tworzona jest fabryka, która implementuje pewien przyjęty interfejs zdefiniowany do tworzenia obiektów klas bazowych.

4.4.2. Zastosowanie

- W przypadku gdy w programie pracujemy z różnymi grupami powiązanych ze sobą obiektów, ale nie chcemy być zależni od konkretnych typów z danej grupy
- W przypadku gdy chcemy mieć w przyszłości możliwość rozszerzenia kodu o nowe grupy obiektów

4.4.3. UML



Rysunek 15: Wzorzec Abstract Factory

4.4.4. Kod

Github

4.4.5. Zalety i wady

- Zalety
  - Niezależność od konkretnych typów obiektów

  - Zgodność typów produkowanych przez fabryki
  - Przeniesienie kodu odpowiedzialnego za tworzenia obiektów do jednego miejsca
  - Możliwość dodawania nowych fabryk w przyszłości
- Wady
  - Konieczność wprowadzenia do kodu wielu dodatkowych klas i interfejsów

4.4.6. Porównanie do innych wzorców TODO

-

5 Wzorce strukturalne

5.1. Adapter

5.1.1. Charakterystyka

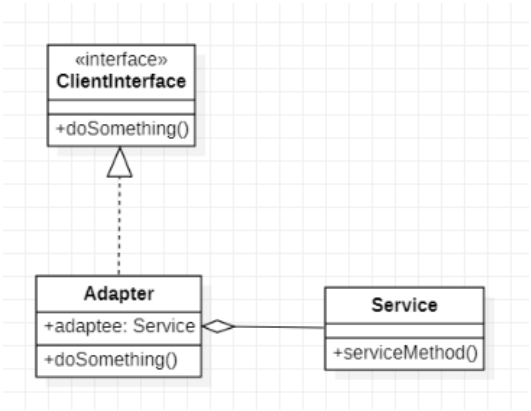
Adapter to strukturalny wzorzec projektowy, który pozwala obiektom o niekompatybilnych interfejsach na kolaborację poprzez dostosowanie jednego z interfejsów do pozostałych.

5.1.2. Zastosowanie

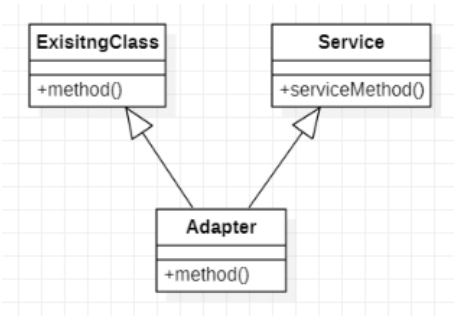
- W przypadku gdy chcemy użyć jakiejś istniejącej klasy, ale jej interfejs nie jest kompatybilny z resztą naszego kodu
- W przypadku gdy chcemy użyć wspólnego interfejsu dla wielu nadklas, którym brakuje pewnej wspólnej funkcjonalności

5.1.3. UML

Wzorzec adaptera może być zrealizowany przez kompozycję (Object Adapter) lub przez dziedziczenie (Class Adapter).



Rysunek 16: Wzorzec Object Adapter (Adapter Obiektowy)



Rysunek 17: Wzorzec Class Adapter (Adapter Klasowy)

5.1.4. Kod

Github

5.1.5. Zalety i wady

- Zalety
  - Separacja logiki biznesowej aplikacji od kodu konwersji (Single responsibility)
  - Możliwość wprowadzania do kodu nowych adapterów bez psucia istniejącego kodu
- Wady
  - Wzrasta złożoność kodu, ponieważ konieczne jest wprowadzenie dodatkowych klas i interfejsów

5.1.6. Porównanie do innych wzorców TODO

-

5.2. Bridge

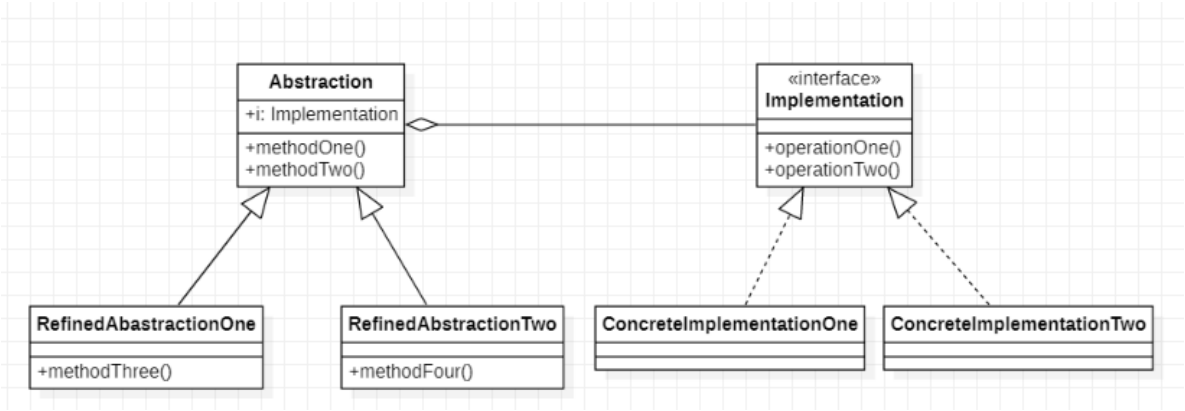
5.2.1. Charakterystyka

Bridge to strukturalny wzorzec projektowy, który pozwala na podzielenie hierarchii klas lub jednej dużej klasy na dwie oddzielne hierarchie. Jedna z tych hierarchii reprezentuje abstrakcję, a druga implementację. Każda z hierarchii może być rozwijana niezależnie.

5.2.2. Zastosowanie

- W przypadku gdy chcemy rozwinąć hierarchie klasy biorąc pod uwagę więcej niż jedną jej cechę
- W przypadku gdy chcemy móc zmieniać implementację zachowania klasy w czasie działania programu
- W przypadku gdy chcemy podzielić jedną klasę, która ma wiele wariantów wykonywania tej samej funkcjonalności

5.2.3. UML



Rysunek 18: Wzorzec Bridge

5.2.4. Kod

Github

5.2.5. Zalety i wady

- Zalety
  - Można dodawać nowe abstrakcje i implementacje bez potrzeby modyfikacji istniejącego kodu
  - Różne typy zachowań są odseparowane od abstrakcji tych zachowań
  - Można rozwijać abstrakcje i implementacje niezależnie
- Wady
  - Wprowadzenie wzorca wymaga dodatkowych klas i interfejsów

5.2.6. Porównanie do innych wzorców TODO

-

5.3. Composite

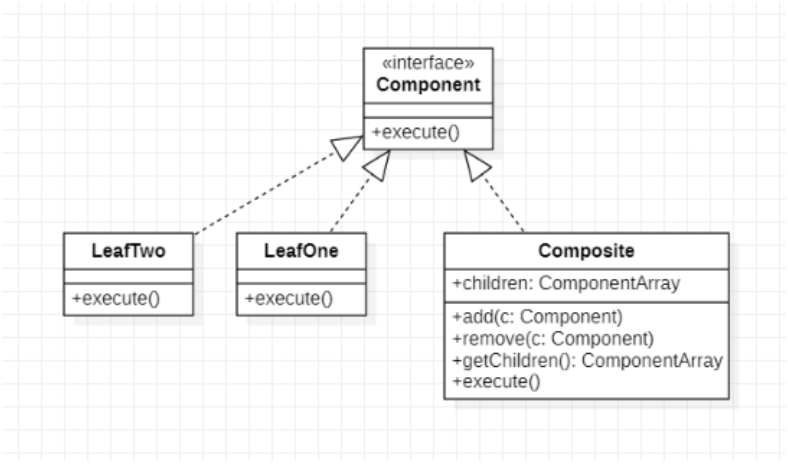
5.3.1. Charakterystyka

Kompozyt to strukturalny wzorzec projektowy, który służy do implementacji w programie struktur drzewiastych, a następnie na pracę z tymi strukturami tak jakby były pojedynczymi obiektami.

5.3.2. Zastosowanie

- W przypadku gdy w programie korzystamy z modelu danych, który ma strukturę drzewa
- W przypadku gdy chcemy aby obiekty proste oraz złożone były traktowane tak samo

5.3.3. UML



Rysunek 19: Wzorzec Composite

5.3.4. Kod

Github

5.3.5. Zalety i wady

- Zalety
  - Wygodna praca z rekursywnymi strukturami danych
  - Można wprowadzać do aplikacji nowe typy węzłów bez konieczności modyfikacji istniejącego kodu
- Wady
  - Konieczne jest dopasowanie każdej z klas reprezentujących węzeł do wspólnego interfejsu, co w przypadku wielu klas może być trudne

5.3.6. Porównanie do innych wzorców TODO

-

5.4. Flyweight

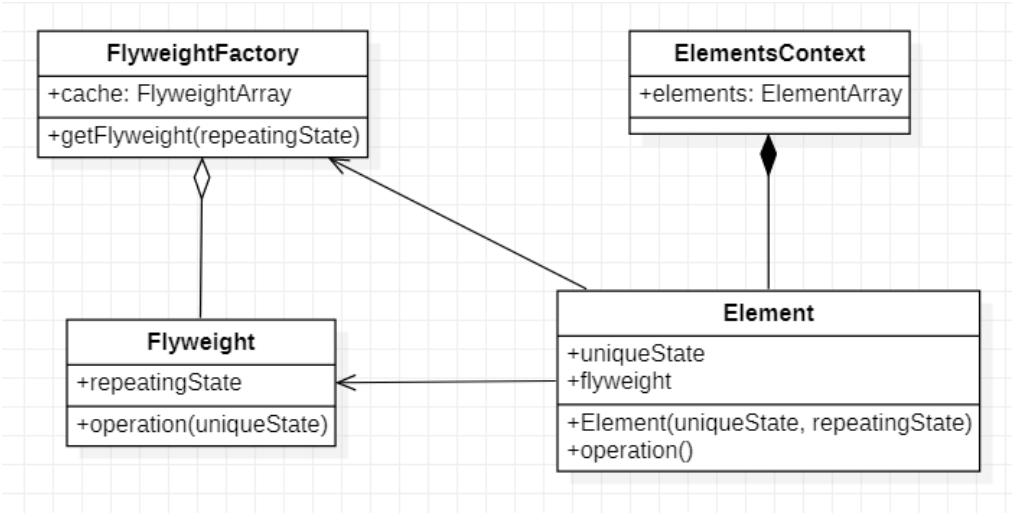
5.4.1. Charakterystyka

Flyweight to strukturalny wzorzec projektowy, który pozwala na zmieszczenie większej liczby obiektów pamięci RAM poprzez przeniesienie atrybutów o powtarzających się wartościach do osobnych klas.

5.4.2. Zastosowanie

- W przypadku gdy program ma obsługiwać ogromną liczbę obiektów, które nie mieszczą się w pamięci RAM

5.4.3. UML



Rysunek 20: Wzorzec Flyweight

5.4.4. Kod

Github

5.4.5. Zalety i wady

- Zalety
  - Oszczędność pamięci przy założeniu ogromnej liczby obiektów
- Wady
  - Kod staje się bardziej skomplikowany
  - Mniejsze zużycie RAM może być zyskane większym zużyciem cyklów procesora przy wywołaniach metod pyłka

5.4.6. Porównanie do innych wzorców TODO

-

5.5. Proxy

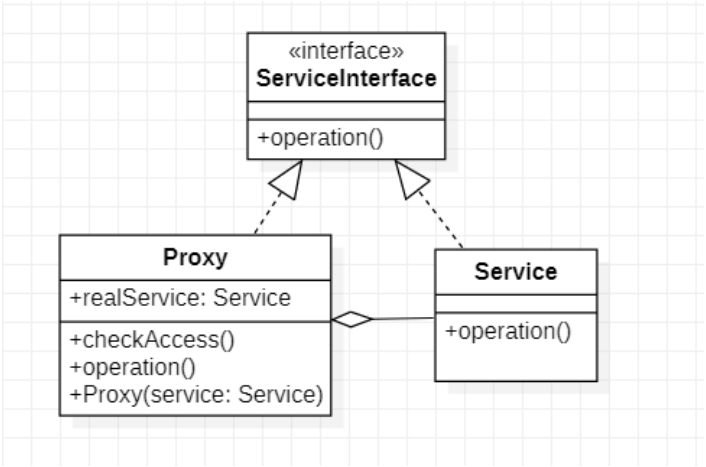
5.5.1. Charakterystyka

Proxy to strukturalny wzorzec projektowy, który kontroluje dostęp do obiektu na którym rzeczywiście wykonywane są operacje. Proxy pozwala na wykonanie jakiś operacji przed lub po tym jak właściwe zapytanie zostanie przekazane do prawdziwego obiektu.

5.5.2. Zastosowanie

- Lokalne wywołanie metody obiektu, który znajduje się na zdalnym serwerze (tzw. remote proxy).
- Przechowywanie rezultatów zapytania w pamięci programu, zwłaszcza gdy wyniki są bardzo duże (tzw. caching proxy)
- W przypadku kontroli dostępu do wywołania metod obiektu (protection proxy)
- W celu implementacji lazy loading dla dużego obiektu, opóźnienie inicjalizacji obiektu do momentu kiedy jest potrzebny w programie (virtual proxy)

5.5.3. UML



Rysunek 21: Wzorzec Proxy

5.5.4. Kod

Github

5.5.5. Zalety i wady

- Zalety
  - Możliwość wywoływania operacji na obiektach ’bez wiedzy klienta’
  - Możliwość zarządzania cyklem życia rzeczywistego obiektu
  - Możliwość dodawania nowych Proxy bez zmiany kodu klienta oraz serwisu

- Wady
  - Odpowiedź od rzeczywistego obiektu może zostać opóźniona
  - Konieczne wprowadzenie dodatkowych klas i interfejsów

5.5.6. Porównanie do innych wzorców TODO

-

5.6. Decorator

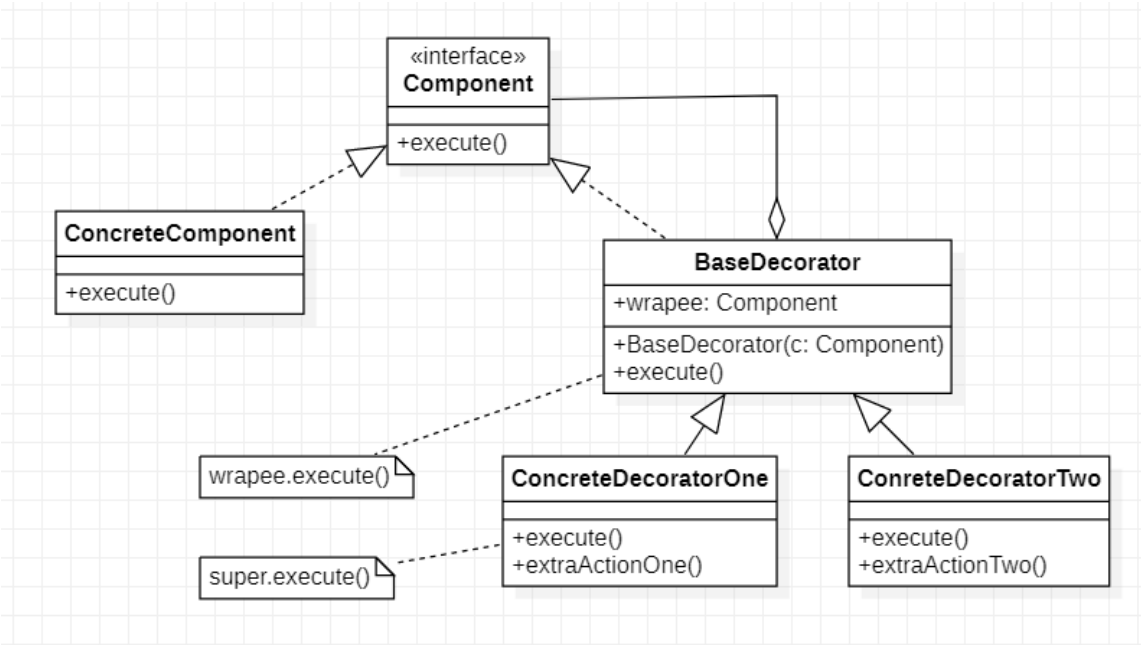
5.6.1. Charakterystyka

Decorator to strukturalny wzorzec projektowy, który pozwala na dodanie do klas zachowań. Realizowane jest to poprzez kompozycje obiektów w specjalnych obiektach (dekoratorach), które zawierają dodatkowe zachowania.

5.6.2. Zastosowanie

- W przypadku gdy chcemy dodać do klasy dodatkowe zachowania w czasie wykonywania programu bez popsucia istniejącego kodu
- W przypadku gdy nie mamy możliwości rozszerzenia klasy przez dziedziczenia, a chcemy do niej dodać dodatkowe zachowania

5.6.3. UML



Rysunek 22: Wzorzec Decorator

5.6.4. Kod

Github

5.6.5. Zalety i wady

- Zalety
  - Możliwość rozszerzenia klasy o nowe zachowania bez konieczności tworzenia podklasy
  - Możliwość dodawania i usuwania zachowań w czasie wykonywania programu

  - Możliwość łączenia nowych zachowań poprzez tworzenie stosu dekoratorów

- Wady
  - Trudno jest usunąć określone dekorator ze stosu dekoratorów
  - Zachowanie zależne od kolejności w stosie dekoratorów

5.6.6. Porównanie do innych wzorców TODO

-

5.7. Facade

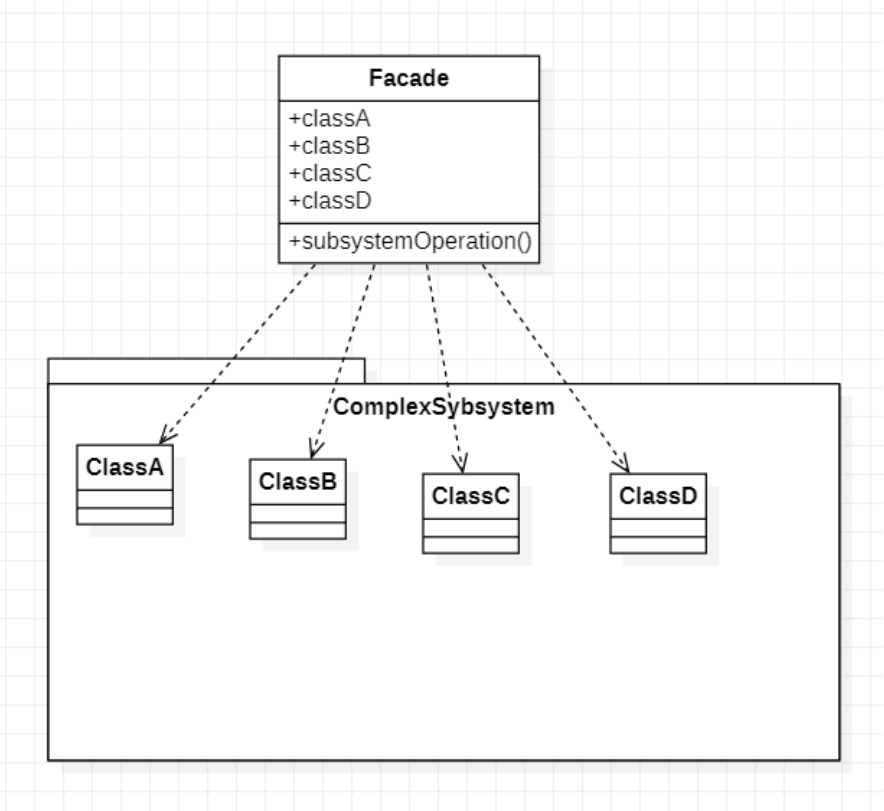
5.7.1. Charakterystyka

Facade to strukturalny wzorzec projektowy, który zapewnia uproszczony interfejs do złożonego zbioru klas, biblioteki lub frameworka.

5.7.2. Zastosowanie

- W przypadku gdy chcemy mieć ograniczony, ale prosty interfejs do złożonego podsystemu
- W przypadku gdy chcemy podzielić podsystem na warstwy

5.7.3. UML



Rysunek 23: Wzorzec Facade

5.7.4. Kod

Github

5.7.5. Zalety i wady

- Zalety

  - Izoluje kod od skomplikowanego podsystemu

- Wady
  - Fasada może stać się 'obiektem bogiem', który zawiera bardzo dużo dependencji
  - Fasada może dawać mniejsze możliwości niż korzystanie bezpośrednio z podsystemu

5.7.6. Porównanie do innych wzorców TODO

-

6 Wzorce behawioralne

6.1. Obserwator (Observer)

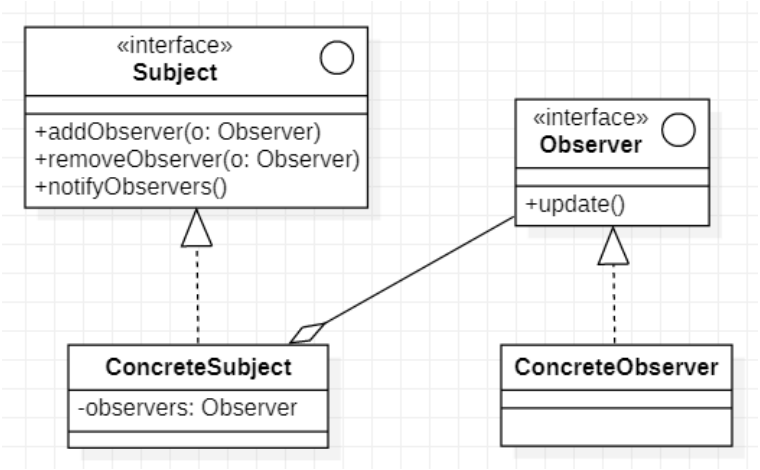
6.1.1. Charakterystyka

Obserwator to behawioralny wzorzec projektowy, który pozwala zdefiniować mechanizm subskrypcji w celu powiadamiania wielu obiektów o zdarzeniach zachodzących w obiekcie, który te obiekty obserwują.

6.1.2. Zastosowanie

- Kiedy należy powiadomić zmieniającą się listę obiektów o pewnym zdarzeniu, np. naciśnięciu elementu GUI
- Gdy zmiany stanu jednego obiektu mogą wymagać zmiany innych obiektów, a rzeczywisty zestaw obiektów jest wcześniej nieznany lub zmienia się dynamicznie
- Gdy niektóre obiekty w aplikacji muszą obserwować inne, ale tylko przez ograniczony czas lub w określonych przypadkach.

6.1.3. UML



Rysunek 24: Asocjacja dwukierunkowa

6.1.4. Kod

Github.

6.1.5. Zalety i wady

- Zalety
  - Możesz wprowadzić nowe klasy subskrybentów bez konieczności zmiany kodu wydawcy (i na odwrót, jeśli istnieje interfejs wydawcy).
  - Możesz ustanowić relacje miedzy obiektami w czasie wykonywania.

- Wady
  - Subskrybenci są powiadamiani w kolejności losowej
  - Ciężko jest śledzić flow aplikacji

6.1.6. Porównanie do innych wzorców

Chain of responsibility, command i obserwer poruszają różne sposoby łączenia nadawców i odbiorców:

- Chain of responsibility przekazuje zadanie sekwencyjnie wzdłuż dynamicznego łańcucha potencjalnych odbiorców, dopóki jeden z nich nie zajmie się nim, w przeciwieństwie do Observera wzorzec ten nie wymaga zawierani w sobie referencji klas komunikujących się,
- Podobnie jak Chain of responsibility i Command wzorzec Observer może być modyfikowany w czasie wykonywania programu
- Command ustanawia jednokierunkowe połaczenia miedzy nadawcami i odbiorcami

6.2. Chain of responsibility

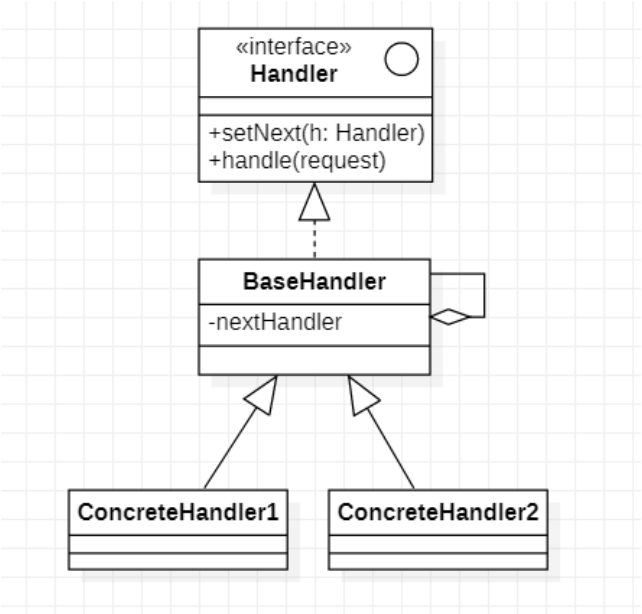
6.2.1. Charakterystyka

Pozwala przekazywać zadania wzdłuż łańcucha handlerów. Po otrzymaniu zadania, każdy handler postanawia albo przetworzyć zadanie, albo przekazać je do następnego handlera w łańcuchu.

6.2.2. Zastosowanie

- Obsługa żądań, które wymagają walidacji danych i autoryzacji
- Obsługa zdarzeń w GUI, przechodząc po drzewie elementów GUI szukamy elementu mogącego obsłużyć zdarzenie
- Wszystkie sytuacje gdzie konieczne jest wykonanie kilku procedur obsługi w określonej kolejności

6.2.3. UML



Rysunek 25: Wzorzec Chain of responsibility

6.2.4. Kod

Github

6.2.5. Zalety i wady

- Zalety
  - Rozdzielenie klas wysyłających komunikaty od odbierających komunikaty
  - Możliwość modyfikacji bez naruszenia istniejącego kodu

  - Możliwość zmian w czasie wykonywania programu (np. dodanie nowego handlera, zmiana kolejności)

- Wady
  - Brak gwarancji obsługi żądania przez łańcuch

6.2.6. Porównanie do innych wzorców

Chain of responsibility, command i obserwator poruszają różne sposoby łączenia nadawców i odbiorców:

- Chain of responsibility przekazuje zadanie sekwencyjnie wzdłuż dynamicznego łańcucha potencjalnych odbiorców, dopóki jeden z nich nie zajmie się nim, w przeciwieństwie do Observera wzorzec ten nie wymaga zawierania w sobie referencji klas komunikujących się
- Podobnie jak Observer oraz Chain of responsibility wzorzec ten może być modyfikowany w czasie wykonywania programu (modyfikacja handlerów i ich kolejności)
- Handlery w Chain of responsibility mogą być implementowane jako Command. W takim przypadku można wykonać wiele różnych operacji w tym samym obiekcie kontekstu, reprezentowanym przez zadanie.

6.3. Command

6.3.1. Charakterystyka

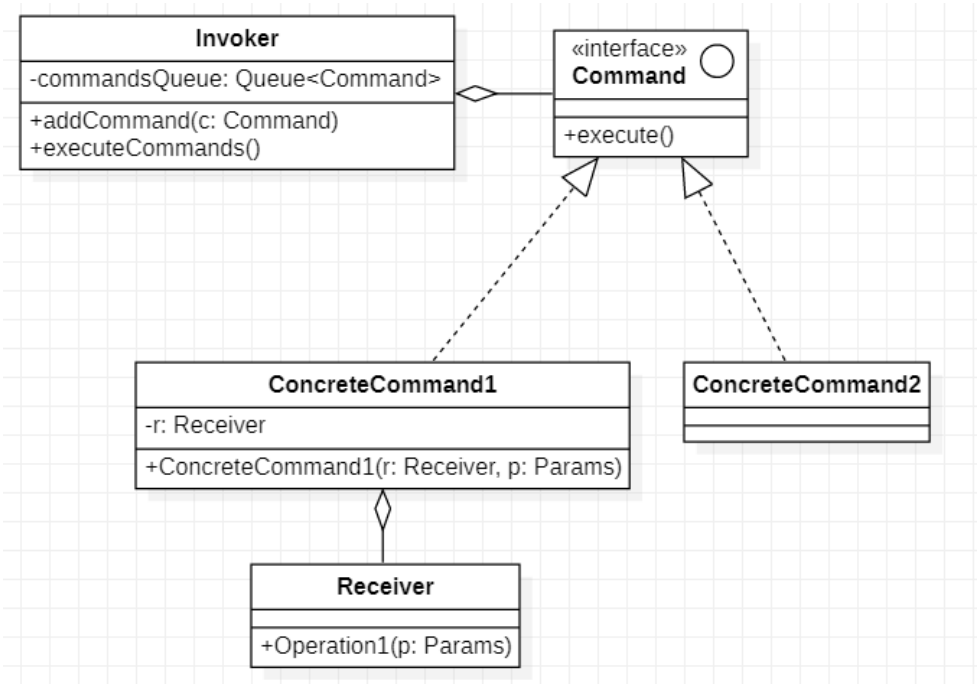
Command to behawioralny wzorzec projektowy, który zmienia zadanie do wykonania w autonomiczny obiekt zawierający wszystkie informacje o zadaniu. Zadania te są zlecane do wykonania innym obiektom (tzw. Invoker), które na podstawie obiektu zadań wykonują powierzone im zadanie. Obiekt Invoker może zarządzać otrzymanymi obiektami zadań, np. je kolejkować, zapisywać historię wykonania, opóźniać itd. Umożliwia to:

- parametryzowanie żądań
- opóźnienie (ustalenie czasu) wykonania zadania
- kolejkowanie wykonania zadań
- obsługę operacji, które można cofnąć

6.3.2. Zastosowanie

- Kolejkowanie zadań, np. zarządzanie wątkami
- Parametryzowanie żądań
- Wykonywanie operacji odwracalnych, np. program do rysowania może mieć opcję UNDO
- Planowanie wykonywania zapytań w czasie, np. opóźnienie zapytań do zdalnej usługi
- Zdalnie wywoływanie metod

6.3.3. UML



Rysunek 26: Wzorzec Command

6.3.4. Kod

Github

6.3.5. Zalety i wady

- Zalety
  - Oddzielenie klas wykonujących operacje od klas na których są one wykonywane
  - Możliwość modyfikacji istniejącego kodu bez popsucia go (np. dodanie Command)
  - Możliwość modyfikacji wykonywanych zadań w czasie wykonywania programu (np. kolejkowanie Command do wykonania)
- Wady
  - Każda nowa akcja zwiększa liczbę klas do utrzymywania, ponieważ wymaga stworzenia nowej klasy implementującej interfejs Command

6.3.6. Porównanie do innych wzorców

Chain of responsibility, command i observer poruszają różne sposoby łączenia nadawców i odbiorców:

- Command podobnie jak Chain of responsibility nie wymaga zawierania w sobie referencji klas komunikujących się, a także może być modyfikowany w czasie wykonywania programu (modyfikacja handlerów i ich kolejności)

- Command nie definiuje określonej kolejności wykonywania akcji jak jest to w Chain of responsibility
- Command ustanawia jednokierunkowe połączenia miedzy nadawcami i odbiorcami w którym korzysta z wprowadzenia dodatkowej warstwy, takiej warstwy nie ma w Observer ani w Chain of responsibility
- Handlery w Chain of responsibility mogą być implementowane jako Command. W takim przypadku można wykonać wiele różnych operacji w tym samym obiekcie kontekstu, reprezentowanym przez zadanie.

6.4. Strategy

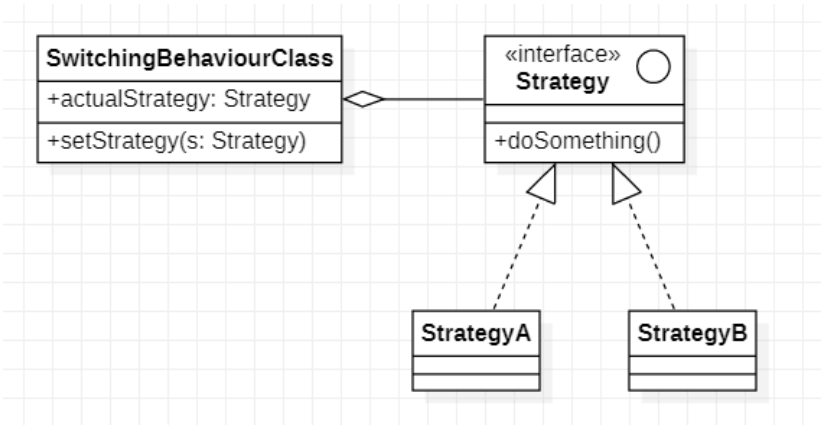
6.4.1. Charakterystyka

Wzorzec ten pozwala na wyodrębnienie z klasy operacji, która może być realizowana na wiele sposobów i utworzenia zbioru klas, które będą implementowały każdy z tych sposobów. Klasy te służą następnie do konfigurowania klasy wykonującej operacje, konfigurujemy, która strategia ma zostać wybrana do zastosowania.

6.4.2. Zastosowanie

- W celu użycia różnych wariantów algorytmu w obiekcie i przełączania się z jednego algorytmu na inny w czasie wykonywania
- W sytuacji gdy mamy w kodzie wiele podobnych klas, które różnią się tylko sposobem wykonywania niektórych zachowań, możemy wtedy stworzyć jedną klasę, która będzie miała zaimplementowany wybór jednego z tych zachowań
- W celu izolacji logiki biznesowej klasy od szczegółów implementacji algorytmów
- W celu zastąpienia istniejącego w kodzie dużego operatora warunkowego, który przełącza między różnymi wariantami tego samego algorytmu

6.4.3. UML



Rysunek 27: Wzorzec Strategy

6.4.4. Kod

Github

6.4.5. Zalety i wady

- Zalety
  - Możliwe jest wprowadzanie nowych zmian bez zmiany kontekstu w którym są używane
  - Możliwe jest konfigurowanie strategii w czasie wykonywania programu

  - Izolowanie logiki algorytmu od jego kontekstu

- Wady

  - Wprowadza dodatkowy narzut związany z koniecznością implementowania wielu klas, dla małej liczby wariantów algorytmu może być to nieopłacalne
  - Klient musi być świadomy różnic pomiędzy strategiami

6.4.6. Porównanie do innych wzorców

- Command i strategia mogą wyglądać podobnie, ponieważ każdego z nich można użyć do sparametryzowania obiektu przy pomocy jakiegoś działania, używane są one jednak w różnych celach.

  - Command używamy aby zamienić dowolną operacje na obiekt. Parametry operacji staja się polami tego obiektu. Taka konwersja pozwala opóźnić wykonanie operacji, umieścić ją w kolejce, zapisać historie poleceń czy wysłać polecenia do usług zdalnych. Obiekt konkretnego Command może mieć własne obiekty (Receiver) na których wykonywana jest operacja (w strategii działamy w ramach jednej klasy).
  - Natomiast wzorzec Strategy zazwyczaj opisuje różne sposoby robienia tego samego, pozwalając na zamianę sposobu wykonywania danego zadania w ramach jednej klasy.

6.5. Template Method

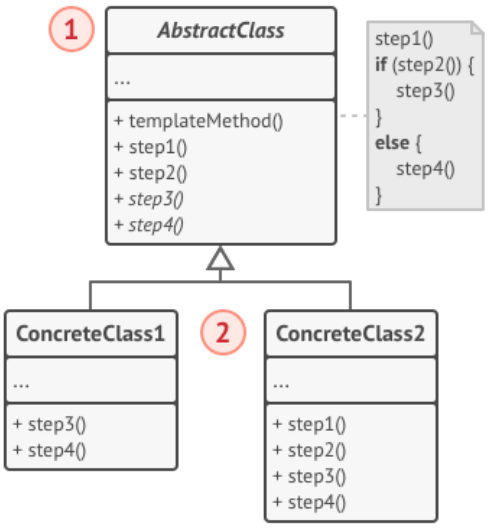
6.5.1. Charakterystyka

Definiuje szkielet algorytmu w nadklasie (operacje i kolejność ich wykonywania), ale pozwala podklasom zastąpić określone kroki algorytmu (poprzez zaimplementowanie metody abstrakcyjnej lub nadpisanie metody) bez zmiany jego struktury.

6.5.2. Zastosowanie

- Używamy wzorca gdy chcemy aby klienci rozszerzali tylko określone kroki algorytmu, ale nie cały algorytm lub jego strukturę.
- Używamy wzorca gdy mamy kilka klas, które zawierają prawie identyczne algorytmy z niewielkimi różnicami. W rezultacie może być konieczne zmodyfikowanie obu klas po zmianie algorytmu.

6.5.3. UML



Rysunek 28: Wzorzec Template Method

6.5.4. Kod

Github

6.5.5. Zalety i wady

- Zalety
  - Kod, który się nie zmienia można przechowywać w superklasie
  - Można pozwolić klientom na modyfikowanie tylko niektórych części algorytmu
- Wady

  - Klienci mogą być ograniczeni przez strukturę algorytmu
  - Dla algorytmów składających się z wielu kroków metody szablonowe mogą być trudne w utrzymywaniu

6.5.6. Porównanie do innych wzorców

- Metoda szablonowa może być stosowana razem ze Strategy w celu utworzenia struktury algorytmu, który jest definiowany przez Strategy (patrz przykład na github). Możemy także zdefiniować dla jednej strategii wiele template method, tym samym otrzymujemy wiele struktur algorytmu opisywanego przez strategię.
- Metoda szablonowa opiera się na dziedziczeniu: pozwala na zmianę części algorytmu poprzez rozszerzenie tych części w podklasach. Strategy oparta jest na zawieraniu(composition): możesz zmieniać części zachowania obiektu, dostarczając mu różne strategie, które odpowiadają temu zachowaniu. Metoda szablonów działa na poziomie klasy, wiec jest statyczna. Strategia działa na poziomie obiektu, umożliwiając zmianę zachowań w czasie wykonywania.

6.6. State (State machine)

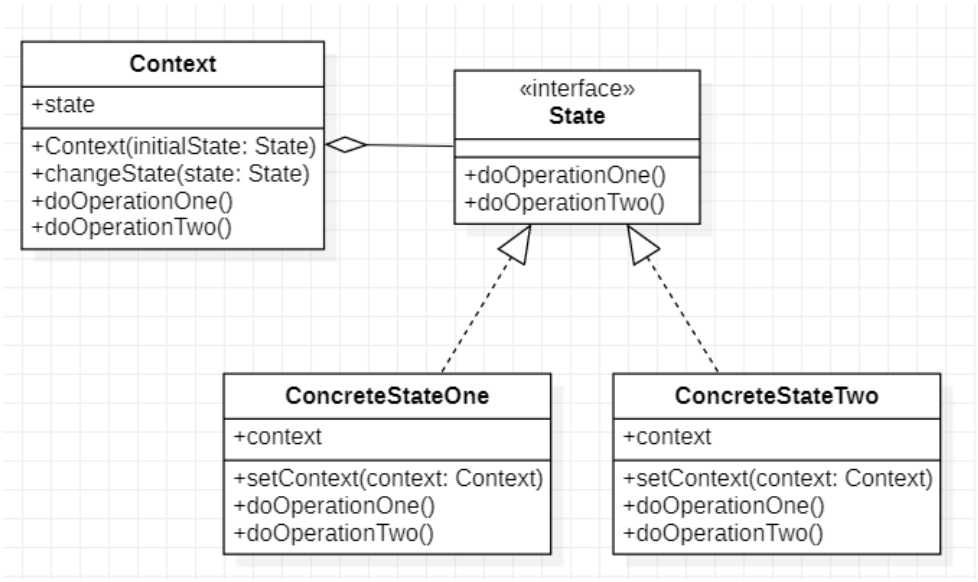
6.6.1. Charakterystyka

To behawioralny wzorzec projektowy, który pozwala na zmianę zachowania klasy w zależności od jej wewnętrznego stanu. Działanie State przypomina działanie skończonej maszyny stanowej - w zależności od stanu w którym się znajdujemy, efekty wykonywania działań mają inne rezultaty.

6.6.2. Zastosowanie

- Używamy wzorca gdy klasa zachowuje się inaczej w zależności od stanu w którym się znajduje, istnieje duża liczba możliwych stanów oraz stany często się zmieniają
- Używamy wzorca gdy mamy klasę, która zawiera metody zawierające ogromną liczbę instrukcji *switch* oraz *if*
- Używamy wzorca gdy mamy dużo zduplikowanego kodu, który wykonywany jest w w instrukcjach warunkowych

6.6.3. UML



Rysunek 29: Wzorzec State

6.6.4. Kod

Github

6.6.5. Zalety i wady

- Zalety
  - Zachowuje zasadę pojedynczej odpowiedzialności - każdy stan to inna klasa

  - Możliwość wprowadzania nowych stanów bez konieczności modyfikacji kodu stanów istniejących
  - Eliminuje dużą liczbę wyrażeń warunkowych, o skutkuje lepszą organizacją kodu

- Wady
  - Stosowanie wzorca jest ograniczone do wielkiej ilości stanów, które często się zmieniają, dla małej liczby lub rzadko zmieniających się stanów wzorzec ten może być zbytnim skomplikowaniem

6.6.6. Porównanie do innych wzorców TODO

-

6.7. Visitor

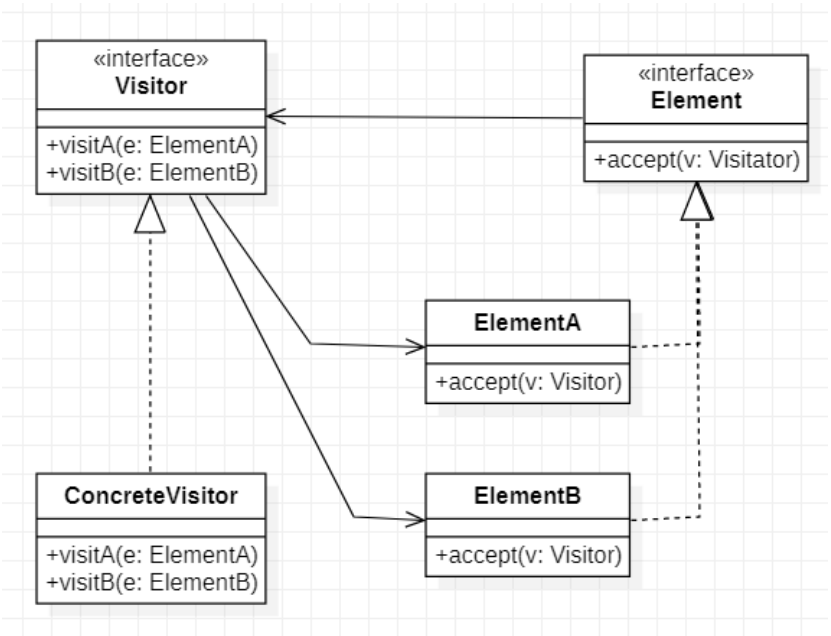
6.7.1. Charakterystyka

Wizytator to behawioralny wzorzec projektowy, który pozwala na oddzielenie obiektów od algorytmów, które są na nich wykonywane. Wzorzec ten pozwala wykonać odpowiedni algorytm na odpowiednim typie obiektu bez konieczności stosowanie złożonych instrukcji warunkowych. To obiekt posiadający implementacje algorytmów odwiedza obiekty i wykonuje na nich odpowiedni algorytm w zależności od typu obiektu, który odwiedza.

6.7.2. Zastosowanie

- Stosujemy wzorzec gdy dany algorytm może być wykonywany na dużej liczbie klas i każda z implementacji jest inna
- Stosujemy wzorzec aby przenieść podobne zachowania wielu klas do jednej klasy, co daje lepszą organizację kodu
- Często przydatny w sytuacji gdy chcemy przejść po skomplikowanej strukturze obiektów o różnym typie, np. drzewie

6.7.3. UML



Rysunek 30: Wzorzec Visitor

6.7.4. Kod

Github

6.7.5. Zalety i wady

- Zalety
  - Łatwe wprowadzanie nowego zachowania dla klas bez modyfikacji tych klas
  - Możliwość przeniesienia metod o takim samym działaniu z wielu klas w jedno miejsce
  - Podczas przechodzenia do strukturach wizytator może zbierać użyteczne dla programu informacje, np. liczba odwiedzonych węzłów
- Wady
  - Wizytator nie ma dostępu do prywatnych składowych obiektów klas z którymi współpracuje
  - Konieczność aktualizacji kodu wizytatora w przypadku dodania klasy z którą ma współpracować

6.7.6. Porównanie do innych wzorców TODO

-

7 Szablon na opracowanie pojedynczego wzorca

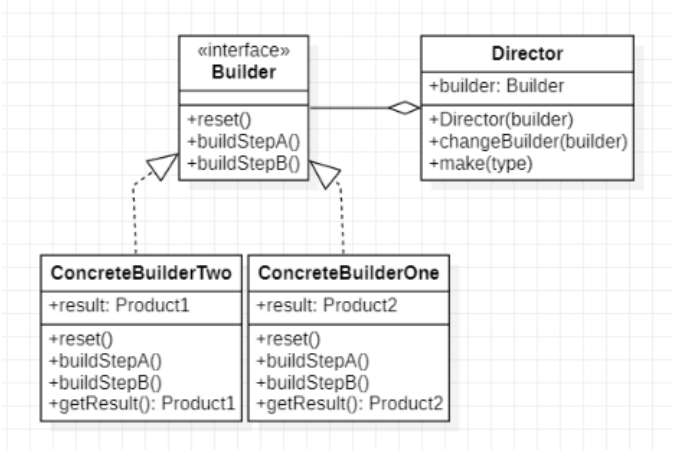
7.1. Nazwa wzorca

7.1.1. Charakterystyka

7.1.2. Zastosowanie

-

7.1.3. UML



Rysunek 31: Wzorzec

7.1.4. Kod

Github

7.1.5. Zalety i wady

- Zalety
  -
- Wady
  -

7.1.6. Porównanie do innych wzorców TODO

-