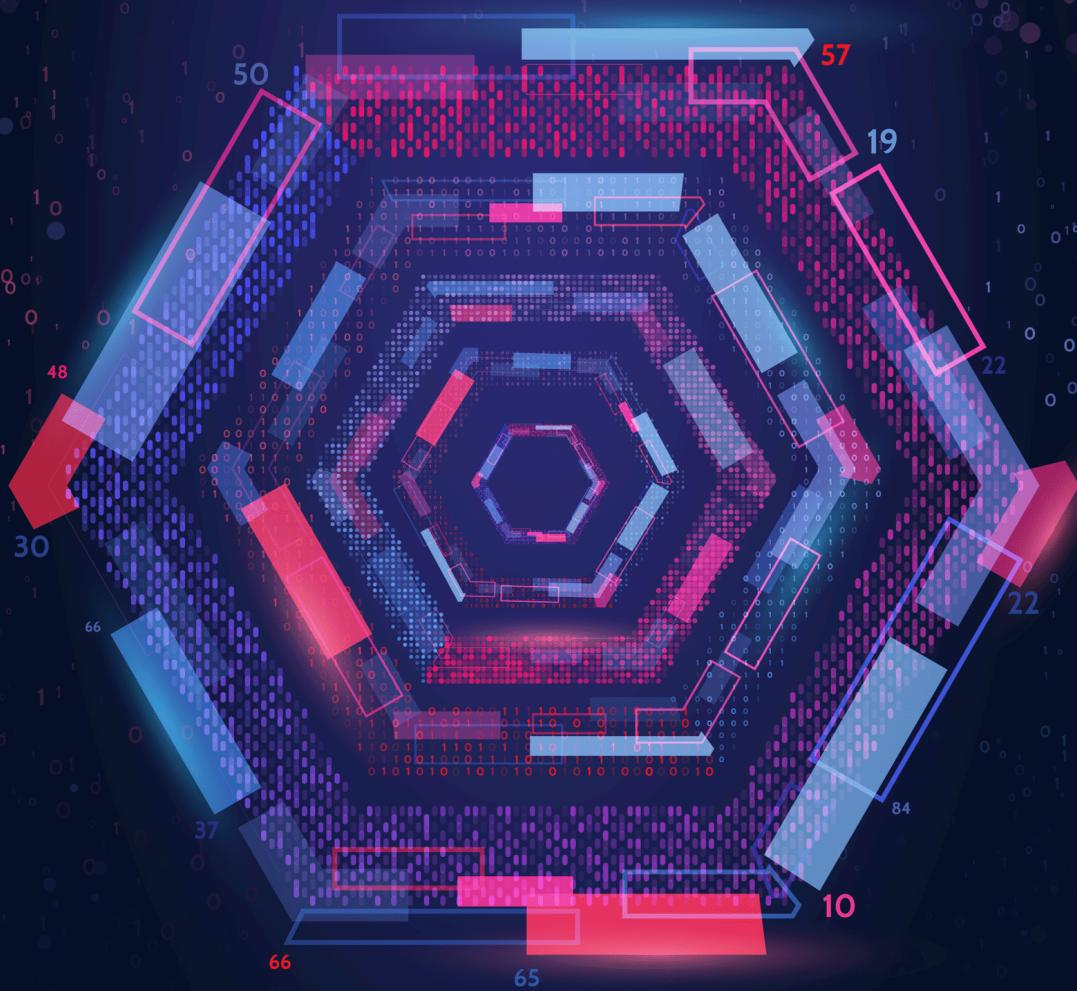


# **WZORCE PROJEKTOWE**

**NOWOCZESNY PODRĘCZNIK**



Aleksander Shvets

# **WZORCE PROJEKTOWE**

## **NOWOCZESNY PODRĘCZNIK**

v2022-1.15

Kupujący Michał Lasia  
[michal.lasia3@gmail.com \(#78893\)](mailto:michal.lasia3@gmail.com)

# Parę słów o prawach autorskich

Witajcie! Nazywam się Aleksander Shvets. Jestem autorem książki *Wzorce projektowe. Nowoczesny podręcznik*<sup>1</sup> oraz kursu online *Zanurzenie w refaktoryzacji*<sup>2</sup>.



Ta książka jest przeznaczona do użytku osobistego. Proszę o nieudostępnianie swojego egzemplarza nikomu spoza swojej rodziny. Jeśli chcesz podzielić się tą książką z przyjacielem lub współpracownikiem – kup kolejny egzemplarz i prześlij go w prezencie. Można również wykupić licencję dla całego zespołu lub całej firmy.

Wszystkie zyski ze sprzedaży moich książek i kursów przeznaczam na rozwijanie **Refactoring.Guru**. Każdy zakupiony egzemplarz wspomaga projekt i nieco przybliża moment wydania kolejnej książki.

© Aleksander Shvets, Refactoring.Guru, 2022  
✉ [support@refactoring.guru](mailto:support@refactoring.guru)

📷 Ilustracje: Dmitry Zhart  
📄 Przekład: Michał Rzepka  
📝 Redakcja: Maciej Włodarczak

- 
1. *Wzorce projektowe. Nowoczesny podręcznik:*  
<https://refactoring.guru/pl/design-patterns/book>
  2. *Zanurzenie w refaktoryzacji:*  
<https://refactoring.guru/pl/refactoring/course>

*Dedykuję tę książkę mojej żonie Marii.  
Gdyby nie ona, zapewne ukończyłbym pisanie  
jakieś 30 lat później.*

# Spis treści

<b>Spis treści .....</b>	<b>4</b>
<b>Jak czytać tę książkę .....</b>	<b>6</b>
<b>WPROWADZENIE DO PROGRAMOWANIA OBIEKTOWEGO .....</b>	<b>7</b>
Podstawy programowania obiektowego.....	8
Filary programowania obiektowego .....	13
Relacje pomiędzy obiektami .....	20
<b>WPROWADZENIE DO WZORCÓW PROJEKTOWYCH .....</b>	<b>26</b>
Czym jest wzorzec projektowy? .....	27
Dlaczego mam poznawać wzorce? .....	31
<b>ZASADY PROJEKTOWANIA OPROGRAMOWANIA .....</b>	<b>32</b>
Cechy dobrego projektu.....	33
Zasady projektowania.....	38
§ Hermetyzuj to, co się różni .....	39
§ Programuj pod interfejs, nie implementację .....	43
§ Preferuj kompozycję ponad dziedziczenie.....	48
Zasady SOLID .....	52
§ S: Zasada pojedynczej odpowiedzialności.....	53
§ O: Zasada otwarte/zamknięte .....	55
§ L: Zasada podstawienia Liskov .....	59
§ I: Zasada segregacji interfejsów .....	66
§ D: Zasada odwrócenia zależności .....	69

<b>KATALOG WZORCÓW PROJEKTOWYCH .....</b>	<b>73</b>
<b>Wzorce kreacyjne.....</b>	<b>74</b>
§ Metoda wytwórcza / <i>Factory Method</i> .....	76
§ Fabryka abstrakcyjna / <i>Abstract Factory</i> .....	94
§ Budowniczy / <i>Builder</i> .....	110
§ Prototyp / <i>Prototype</i> .....	131
§ Singleton / <i>Singleton</i> .....	146
<b>Wzorce strukturalne.....</b>	<b>156</b>
§ Adapter / <i>Adapter</i> .....	159
§ Most / <i>Bridge</i> .....	173
§ Kompozyt / <i>Composite</i> .....	189
§ Dekorator / <i>Decorator</i> .....	203
§ Fasada / <i>Facade</i> .....	223
§ Pyłek / <i>Flyweight</i> .....	233
§ Pełnomocnik / <i>Proxy</i> .....	248
<b>Wzorce behawioralne.....</b>	<b>261</b>
§ Łańcuch zobowiązań / <i>Chain of Responsibility</i> .....	265
§ Polecenie / <i>Command</i> .....	284
§ Iterator / <i>Iterator</i> .....	305
§ Mediator / <i>Mediator</i> .....	321
§ Pamiątka / <i>Memento</i> .....	337
§ Obserwator / <i>Observer</i> .....	353
§ Stan / <i>State</i> .....	369
§ Strategia / <i>Strategy</i> .....	386
§ Metoda szablonowa / <i>Template Method</i> .....	401
§ Odwiedzający / <i>Visitor</i> .....	415
<b>Zakończenie .....</b>	<b>432</b>

## Jak czytać tę książkę

Niniejsza książka zawiera opisy 22 klasycznych wzorców projektowych sformułowanych przez “Bandę Czworga” (w skrócie GoF – ang. Gang of Four) w 1994 roku.

Każdy rozdział eksploruje któryś ze wzorców. Dlatego możliwe jest czytanie od deski do deski, albo wybierając tylko te wzorce, które cię interesują.

Wiele wzorców jest ze sobą powiązanych, dlatego możliwe jest łatwe przeskakiwanie między tematami za pomocą licznych zakładek. Na końcu każdego rozdziału znajdziesz listę powiązań bieżącego wzorca z pozostałymi. Jeśli zobaczysz nazwę wzorca którego jeszcze nie znasz – czytaj dalej, pojawi się w którymś z kolejnych rozdziałów.

Wzorce projektowe są uniwersalne. Dlatego też wszystkie przykłady kodu w tej książce są napisane w pseudokodzie który nie podlega ograniczeniom któregokolwiek z języków programowania.

Zanim zaczniesz uczyć się o wzorcach, możesz odświeżyć pamięć przeglądając **kluczowe pojęcia związane z programowaniem obiektowym**. Rozdział ten wyjaśnia także podstawy diagramów UML, a to się przyda, gdyż w książce znajdziesz ich wiele. Oczywiście jeśli już znasz to wszystko na pamięć, zapraszam od razu do **nauki wzorców**.

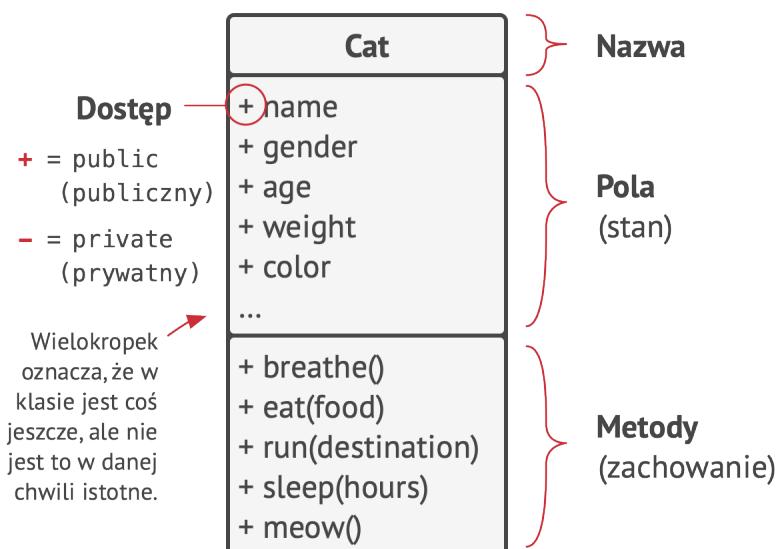
# **WPROWADZENIE DO PROGRAMOWANIA ZORIENTOWANEGO OBIEKTOWO**

# Podstawy programowania obiektowego

Programowanie zorientowane obiektowo (ang. *OOP* – **Object-oriented programming**) to paradymat opierający się na koncepcji opakowywania fragmentów danych i czynności z nimi związanych w specjalne pakiety zwane **obiektami**, konstruowanymi na podstawie “planów” zdefiniowanych przez programistę, zwanych **klasami**.

## Obiekty, klasy

Lubisz koty? Mam nadzieję, że tak, bo zamierzam spróbować wyjaśnić koncepcje związane z programowaniem obiektowym na ich przykładzie.



Oto diagram klasy w UML. Napotkasz ich mnóstwo w niniejszej książce.

Powiedzmy, że masz kota imieniem Leo. Leo jest obiektem – instancją klasy `Kot`. Każdy kot posiada mnóstwo standardowych atrybutów: imię, płeć, wiek, waga, barwa, ulubione jedzenie, itd. Są to *pola* klasy.

Pozostawienie nazw klas i ich składowych na diagramach w języku angielskim to standardowa praktyka, podobnie jak w prawdziwym kodzie. Komentarze i adnotacje można jednak pisać też po polsku.

Wszystkie koty zachowują się podobnie: oddychają, jedzą, biegają, śpią i miauczą. Są to *metody* klasy. Pola oraz metody łącznie nazywamy *składowymi* klasy.

Dane przechowywane w polach obiektu często zwane są jego *stanem*, a wszystkie metody definiują jego *zachowanie*.



Leo: Cat

```
name      = "Leo"
sex       = "męski"
age       = 3
weight    = 7
color     = brązowy
texture   = smugowaty
```

Luna: Cat

```
name      = "Luna"
sex       = "żeński"
age       = 2
weight    = 5
color     = szary
texture   = zwyczajny
```

*Obiekty są instancjami klas.*

Luna, kot twojego znajomego, również jest instancją klasy `Kot`. Można wyróżnić u niej te same atrybuty, co u Leo. Różnicą są wartości tych atrybutów: inna płeć, inne futro i waga ciała.

*Klasa* jest więc planem definiującym strukturę *obiektów*, a te z kolei są konkretnymi instancjami klasy.

## Hierarchie klas

Wszystko ładnie pięknie gdy mówimy o jednej klasie. Prawdziwy program będzie jednak posiadał większą ich ilość. Niektóre klasy mogą być zorganizowane w **hierarchie klas**. Dowiedzmy się, co to oznacza.

Powiedzmy, że twój sąsiad ma psa o imieniu Azor. Okazuje się, że psy i koty mają sporo wspólnego: imię, płeć, wiek, barwa futra – to atrybuty zarówno psa jak i kota. Psy oddychają, śpią, biegają – zupełnie jak koty. Wygląda więc na to, że możemy zdefiniować klasę **Zwierzę** która zatrzyma ich wspólne atrybuty i zachowanie.

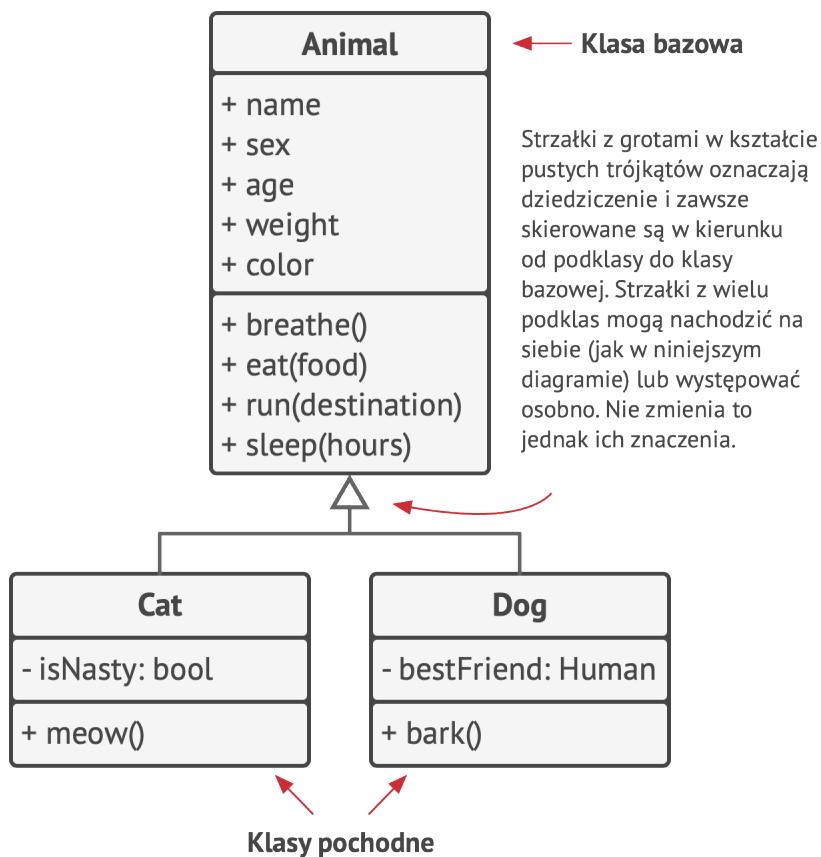
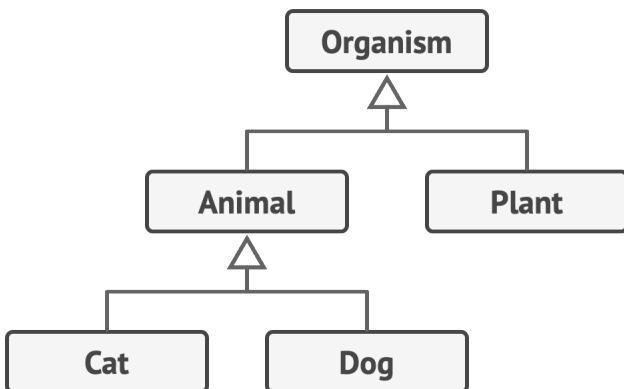


Diagram UML prostej hierarchii klas. Wszystkie klasy w tym diagramie są częścią hierarchii klas **Zwierzę**.

Klasa-rodzic, jak ta którą przed chwilą zdefiniowaliśmy, nazywa się **klassą bazową**. Jej “dzieci” to **podklasy**. Podklasy dziedziczą stan i zachowanie po rodzicu, definiując tylko atrybuty i zachowanie które je od nich odróżnia. Dlatego też klasa `Kot` miałaby metodę `miau`, a klasa `Pies` metodę `hau`.

Zakładając, że mamy podobne wymaganie biznesowe, możemy pójść o krok dalej i wyekstrahować jeszcze ogólniejszą klasę reprezentującą wszystkie żywe `Organizmy` która będzie stanowić klasę bazową dla `Zwierząt` i `Roślin`. Taka piramida klas jest **hierarchią**. W takiej hierarchii, klasa `Kot` dziedziczy po klasach `Zwierzę` i `Organizm`.



*Klasy na diagramie UML można uprościć, jeśli ukazanie relacji między nimi jest istotniejsze niż ich zawartość.*

Podklasy mogą nadpisywać zachowanie metod które dziedziczą po klasie nadzędnej. Podklasa może albo całkowicie zmienić domyślne zachowanie, albo tylko je rozbudować.

# Filary programowania obiektowego

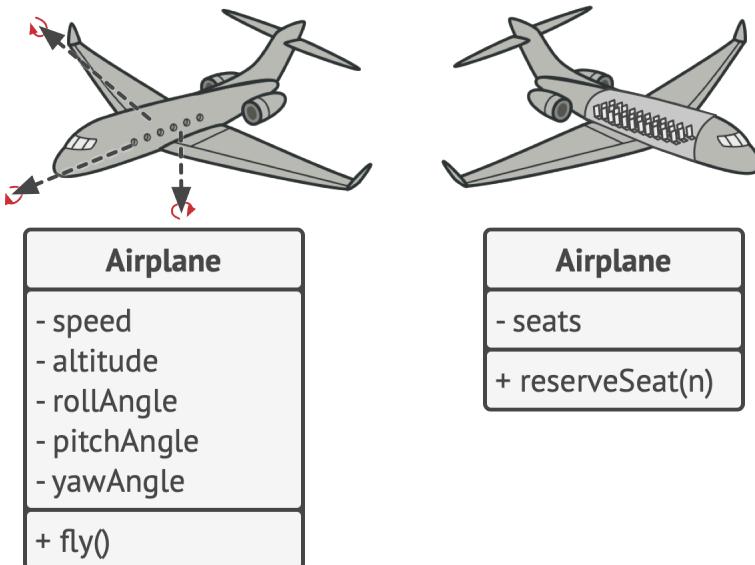
Programowanie zorientowane obiektowo opiera się na czterech filarach, koncepcjach które odróżniają je od innych paradymatów programowania.



## Abstrakcja

Zazwyczaj gdy piszesz program zgodnie z paradymatem obiektowym, kształtujesz obiekty programu na wzór prawdziwych obiektów. Obiekty programu nie reprezentują jednak swoich rzeczywistych pierwowzorów w pełni (i rzadko muszą). Zamiast tego, *modelują* one atrybuty i zachowanie prawdziwych obiektów w pewnym kontekście, ignorując resztę.

Przykładowo, klasa `Samolot` mogłaby istnieć zarówno w symulatorze lotu jak i aplikacji do rezerwacji lotów. W pierwszym przypadku przechowywałaby szczegóły związane z faktycznym lotem, a w drugim jedynie rozkład miejsc siedzących.



Różne modele tego samego rzeczywistego obiektu.

*Abstrakcja* jest modelem rzeczywistego obiektu lub zjawiska, ograniczonego do pewnego kontekstu, przez co dokładnie przedstawia szczegóły związane z tym kontekstem, ale pomija pozostałe.

## Hermetyzacja

Aby uruchomić silnik samochodu, musisz jedynie przekręcić klucz w stacyjce lub wcisnąć przycisk. Nie musisz zwierać przewodów pod maską ani kręcić korbowodem by zainicjować cykl pracy silnika. Zamiast tego masz uproszczony interfejs: przycisk, kierownicę i pedały. W ten sposób można opisać **interfejs** – publicznie dostępną część obiektu, nastawioną na interakcję z innymi obiektami.

*Hermetyzacja* to zdolność obiektu do ukrywania części swojego stanu i zachowania przed innymi obiektami, eksponując reszcie programu tylko ograniczony interfejs.

*Hermetyzować* coś oznacza uczynić to coś prywatnym, a tym samym dostępnym wyłącznie w obrębie metod swojej klasy. Istnieje nieco mniej restrykcyjny poziom dostępu zwany chronionym, który czyni składową klasy dostępną także podklasom.

Interfejsy i klasy/metody abstrakcyjne większości języków programowania oparte są na koncepcji abstrakcji i hermetyzacji. We współczesnych językach programowania zorientowanych obiektowo, mechanizm interfejsu (zwykle deklarowany poprzez słowo kluczowe `interface` lub `protocol`) pozwala zdefiniować kontrakty interakcji pomiędzy obiektami. Jest to jeden z powodów dla którego interfejsy interesują się tylko zachowaniem obiektu i nie można w nich zadeklarować pola prywatnego.

Fakt, że słowo *interfejs* oznacza publicznie dostępną część obiektu, a dodatkowo wiele języków programowania posiada typ `interface` jest bardzo mylący. Też tak uważam.

Wyobraź sobie, że masz interfejs `TransportPowietrzny` z metodą `leć(źródło, cel, pasażerowie)`. Projektując symulator transportu lotniczego, możnaby ograniczyć klasę `Lotnisko`, by

współpracowała jedynie z obiektami implementującymi interfejs `TransportPowietrzny`. Dzięki temu można mieć pewność, że każdy obiekt przekazany obiektowi lotnisko, czy to będzie `Samolot`, `Helikopter`, czy nawet `OswojonyGryf`, będzie w stanie lądować i startować z tego typu lotniska.

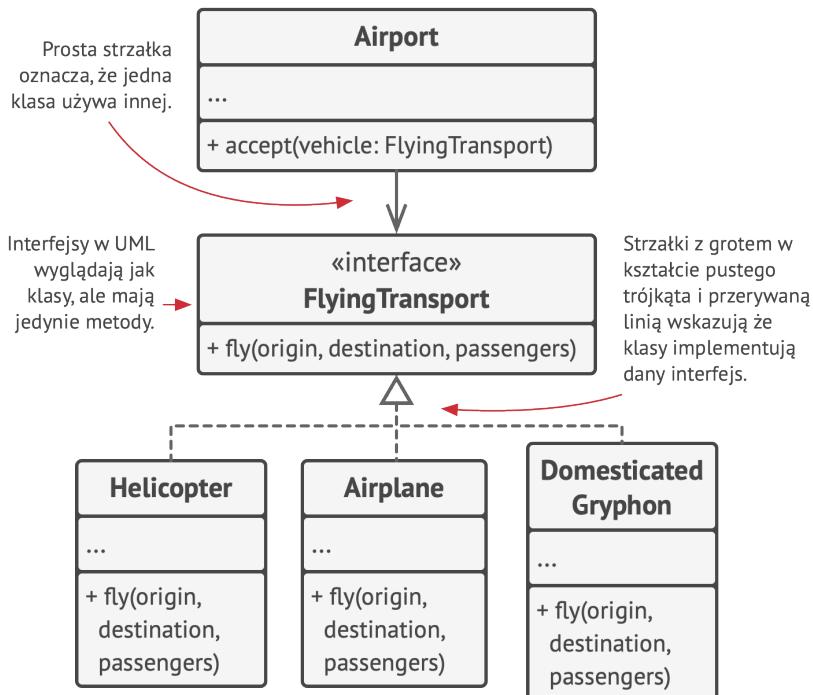


Diagram UML przedstawiający kilka klas implementujących interfejs.

Możemy zmienić implementację metody `leć` w tych klasach w dowolny sposób. O ile sygnatura metody pozostanie zgodna z deklaracją w interfejsie, wszystkie instancje `Lotniska` mogą współpracować z danym statkiem powietrznym.

## Dziedziczenie

*Dziedziczenie* to możliwość tworzenia nowych klas na bazie istniejących. Główną zaletą dziedziczenia jest możliwość ponownego wykorzystania kodu. Jeśli potrzebujesz stworzyć klasę nieco inną od istniejącej, nie trzeba powtarzać tego samego kodu. Wystarczy rozszerzyć istniejącą i umieścić dodatkową funkcjonalność w tak powstałej podklasie. Podklasa przejmie pola i metody klasy bazowej.

Konsekwencją dziedziczenia jest przejęcie przez podklasy interfejsu klasy-rodzica. Nie można ukryć metody w podklasie jeśli zadeklarowano ją w klasie bazowej. Musisz ponadto zaimplementować wszystkie metody zadeklarowane jako abstrakcyjne, nawet jeśli nie mają racji bytu w twojej podklasie.

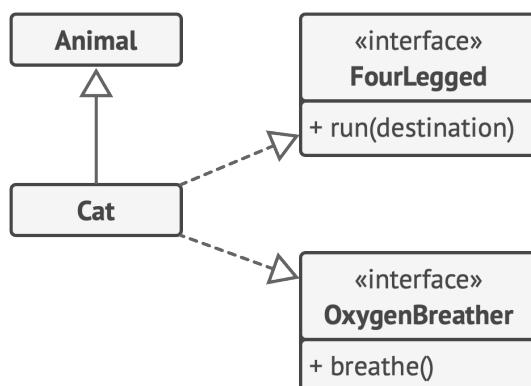


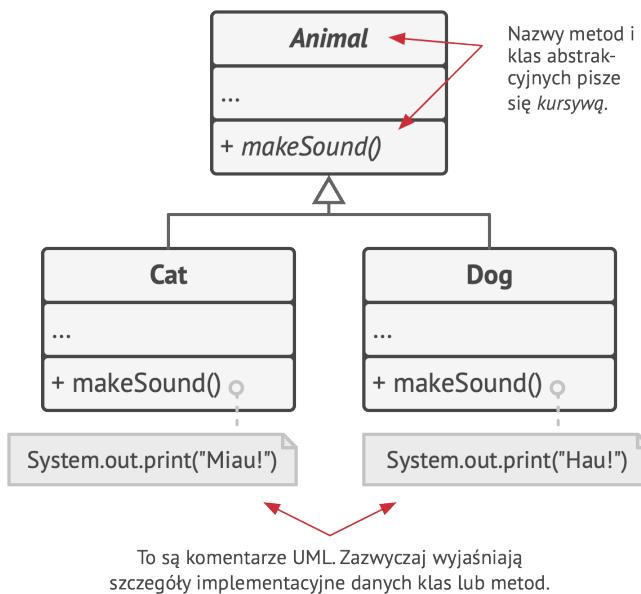
Diagram UML przedstawiający rozszerzenie pojedynczej klasy oraz implementację wielu interfejsów jednocześnie.

W większości języków programowania, podklasa może rozszerzać tylko jedną klasę bazową. Z drugiej strony jednak, każda

klasa może implementować jednocześnie wiele interfejsów. Jak wspomniałem wyżej, jeśli klasa bazowa implementuje interfejs, wszystkie podklasy muszą zrobić to samo.

## Polimorfizm

Spójrzmy na przykłady oparte na zwierzętach. Większość **Zwierząt** wydaje odgłosy. Spodziewać się można więc że wszystkie podklasy będą musiały nadpisać bazową metodę **wydajOdglos** by móc wydawać odpowiednie odgłosy. Możemy więc zadeklarować tę metodę od razu jako *abstrakcyjną*. Pozwoli nam to pominąć tworzenie domyślnej implementacji metody w klasie bazowej, zmuszając podklasy by zrealizowały swoją.



Wyobraź sobie, że umieściliśmy wiele kotów i psów w pojedynczej torbie. Następnie, z zamkniętymi oczami, wyciągamy zwierzaki jedno po drugim. Wyjmując zwierzę z torby, początkowo nie wiemy które wyjęliśmy. Ale jeśli przytulimy je dostatecznie czule, wyda ono odgłos zadowolenia zależny od swej konkretnej klasy.

```
1 bag = [new Cat(), new Dog()];
2
3 foreach (Animal a : bag)
4     a.makeSound()
5
6 // Miau!
7 // Hau!
```

Program nie zna konkretnego typu obiektu zawartego w zmiennej `a`, ale dzięki specjalnemu mechanizmowi zwanemu *polimorfizmem* program może wyśledzić podklasę obiektu którego metodę uruchomiono i wykonać właściwą.

*Polimorfizm* to zdolność programu do wykrywania faktycznej klasy obiektu i wywoływania jego implementacji nawet gdy typ jest nieznany w danym kontekście.

Można również myśleć o polimorfizmie jako o zdolności obiektu do “udawania” czegoś innego, zazwyczaj klasy którą rozszerza lub interfejsu który implementuje. W naszym przykładzie, psy i koty ukryte w torbie udawały zwierzęta ogólne.

# Relacje pomiędzy obiektami

Oprócz *dziedziczenia* i *implementacji* które już poznaliśmy, istnieją inne typy relacji pomiędzy obiektami których nie poruszyliśmy.

## Zależność



*Zależności w UML. Profesor zależy od treści przedmiotu.*

*Zależność* to najbardziej podstawowy i najsłabszy rodzaj zależności pomiędzy klasami. Zależność istnieje pomiędzy dwiema klasami, jeśli pewne zmiany definicji jednej z nich mogą skutkować modyfikacjami drugiej. Zależność zwykle zachodzi gdy w kodzie znajdują się nazwy konkretnych klas. Na przykład określając typy w sygnaturach metod, tworząc instancje obiektów w konstruktorze, itd. Można osłabić zależność stosując interfejsy i klasy abstrakcyjne zamiast konkretnych klas.

Zazwyczaj, diagram UML nie pokazuje wszystkich zależności – jest ich zbyt wiele w prawdziwym kodzie. Zamiast zaśmiecać diagram zależnościami, należy wybiórczo pokazać tylko te które są najistotniejsze dla idei którą usiłujemy przekazać.

## Asocjacja



*Asocjacja w UML. Profesor komunikuje się ze studentami.*

Asocjacja jest relacją w której jeden obiekt korzysta z innego lub wchodzi w interakcję z innym obiektem. Na diagramach UML, relacja asocjacji ukazywana jest prostą strzałką, która prowadzi od obiektu korzystającego w kierunku obiektu wykorzystywanego. Przy okazji warto wspomnieć, że dwukierunkowa asocjacja jest jak najbardziej normalna. W takim przykładzie linia ma dwa groty po obu końcach. Asocjację można traktować jako rodzaj zależności, w której obiekt zawsze ma dostęp do obiektu z którym wchodzi w interakcję, zaś prosta zależność nie powoduje związania obiektów na stałe.

Asocjację powinno się na ogół stosować do przedstawienia takich elementów jak na przykład pole klasy. Połączenie zawsze istnieje i w każdej chwili można złożyć zamówienie dla jej klienta. Ale nie musi to być tylko pole klasy. Jeśli modelujesz klasy z perspektywy interfejsu, asocjacja może wskazywać na istnienie metody zwracającej klienta danego zamówienia.

Aby utrwały różnicę pomiędzy asocjacją a zależnością, spójrzmy na przykład łączący oba typy relacji. Wyobraź sobie klasę Profesor :

```
1 class Professor is
2   field Student student
3   // ...
4   method teach(Course c) is
5     // ...
6     this.student.remember(c.getKnowledge())
```

Spójrz na metodę `nauczaj`. Pobiera w charakterze argumentu obiekt klasy `Przedmiot`, który potem jest użyty w ciele metody. Jeśli ktoś zmieni metodę `pozyskajWiedzę` klasy `Przedmiot` (zmieni nazwę lub doda obowiązkowe parametry, itp.) to kod się popsuje. To właśnie nazywamy zależnością.

Spójrzmy teraz na pole `student` i jak jest wykorzystywane w metodzie `nauczaj`. Wiemy na pewno, że klasa `Student` jest także zależnością `Profesora`: jeśli zmieni się metoda `pamiętaj`, kod `Profesora` również się popsuje. Jednakże skoro pole `student` jest zawsze dostępne dla każdej metody klasy `Profesor`, to relacja z klasą `Student` jest nie tylko zależnością, ale też asocjacją.

## Agregacja



*Agregacja w UML. Wydział posiada profesorów.*

*Agregacja* to wyspecjalizowany rodzaj asocjacji reprezentujący relacje “jeden do wielu”, “wiele do wielu” lub “całość-część” pomiędzy wieloma obiektami.

Zazwyczaj, w przypadku agregacji, obiekt “posiada” zestaw innych obiektów i służy za kontener lub kolekcję. Komponent może istnieć bez kontenera i może być połączony z wieloma kontenerami jednocześnie. W UML, relacja agregacji przedstawiana jest linią zakońzoną grotem w kształcie pustego diamentu po stronie kontenera i strzałką wskazującą na komponent po przeciwej stronie.

Mówiąc o relacjach pomiędzy obiektami, warto pamiętać, że UML przedstawia relację pomiędzy *klasami*. Oznacza to, że obiekt typu uniwersytet może składać się z wielu wydziałów mimo że widzimy tylko jeden “blok” na każdy podmiot na diagramie. Notacja UML może przedstawiać ilości po obu stronach relacji, ale można je pominąć, jeśli ilości da się łatwo wywnioskować z kontekstu.

## Kompozycja



*Kompozycja UML. Uniwersytet składa się z wydziałów.*

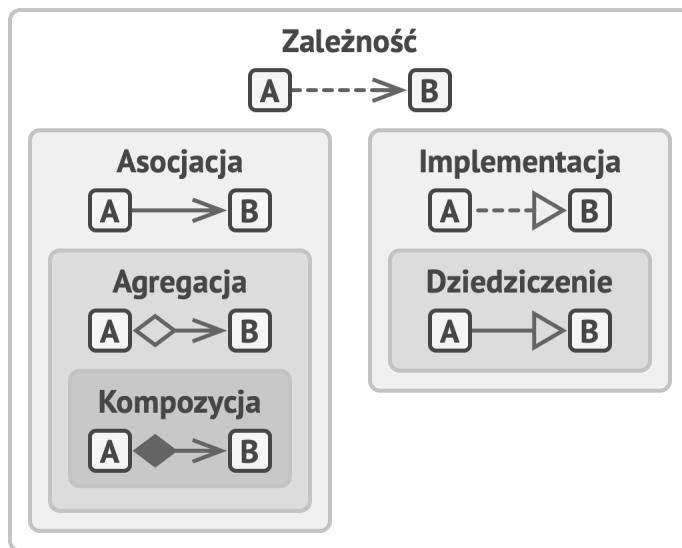
*Kompozycja* to szczególny rodzaj agregacji, w którym jeden obiekt składa się z jednej lub wielu instancji innego. Różnicą pomiędzy tą relacją a innymi jest to, że komponent może istnieć wyłącznie jako część kontenera. W UML kompozycję przedstawia się tak jak agregację, ale diamentowy grot jest tu wypełniony.

Zwróć uwagę, że wiele osób stosuje termin “kompozycja” gdy tak naprawdę mają na myśli zarówno agregację, jak i kompozycję. Niesławnym przykładem tego jest słynna zasada “preferuj kompozycję ponad dziedziczenie”. Nie wynika to z ignorancji, ale raczej z faktu że słowo “kompozycja” (np. “kompozycja obiektu”) brzmi bardziej naturalnie.

## Szersza perspektywa

Teraz gdy znamy wszystkie typy relacji pomiędzy obiektami, sprawdźmy jak się łączą. Mam nadzieję, że rozwieje to wątpliwości dotyczące np. różnic pomiędzy agregacją a kompozycją, albo czy dziedziczenie jest rodzajem zależności.

- **Zależność:** Klasa A może odczuć zmiany dokonywane w klasie B.
- **Asocjacja:** Obiekt A wie o obiekcie B. Klasa A jest zależna od klasy B.
- **Agregacja:** Obiekt A wie o obiekcie B i składa się z B. Klasa A jest zależna od B.
- **Kompozycja:** Obiekt A wie o obiekcie B, składa się z B i zarządza cyklem życia B. Klasa A jest zależna od B.
- **Implementacja:** Klasa A definiuje metody zadeklarowane w interfejsie B. Obiekty A można traktować jak B. Klasa A jest zależna od B.
- **Dziedziczenie:** Klasa A dziedziczy interfejs i implementację po klasie B ale może go rozszerzyć. Obiekt A można traktować jak B. Klasa A jest zależna od B.



*Relacje pomiędzy obiektami i klasami: od najsłabszej do najsilniejszej.*

# **WPROWADZENIE DO WZORCÓW PROJEKTOWYCH**

# Czym jest wzorzec projektowy?

**Wzorce projektowe** to typowe rozwiązania problemów często napotykanych przy projektowaniu oprogramowania. Stanowią coś na kształt gotowych planów które można dostosować, aby rozwiązać powtarzający się problem w kodzie.

Nie można jednak wybrać wzorca i po prostu skopiować go do programu, jak bibliotekę czy funkcję zewnętrznego dostawcy. Wzorzec nie jest konkretnym fragmentem kodu, ale ogólną koncepcją pozwalającą rozwiązać dany problem. Postępując według wzorca możesz zaimplementować rozwiązanie które będzie pasować do realiów twojego programu.

Wzorce często myli się z algorytmami, ponieważ obie koncepcje opisują typowe rozwiązanie jakiegoś znanego problemu. Algorytm jednak zawsze definiuje wyraźny zestaw czynności które prowadzą do celu, zaś wzorzec to wysokopoziomowy opis rozwiązania. Kod powstały na podstawie jednego wzorca może wyglądać zupełnie inaczej w różnych programach.

Algorytm jest jak przepis kulinarny: oba mają wyraźnie określone etapy które trzeba wykonać w określonej kolejności by osiągnąć cel. Wzorzec bardziej przypomina strategię: znany jest wynik i założenia, ale dokładna kolejność implementacji należy do ciebie.

## ⬇️ Co składa się na wzorzec?

Większość wzorców posiada formalny opis, dzięki czemu każdy może odtworzyć ich ideę w różnych kontekstach. Oto sekcje na które zwykle dzieli się opis wzorca:

- **Cel** pobicieżnie opisuje zarówno problem, jak i rozwiązanie.
- **Motywacja** rozszerza opis problemu i rozwiązania jakie umożliwia dany wzorzec.
- **Struktura** klas ukazuje poszczególne części wzorca i jak są ze sobą powiązane.
- **Przykład kodu** w którymś z popularnych języków programowania pomaga zrozumieć ideę wzorca.

Niektóre katalogi wzorców wymieniają inne użyteczne szczegóły, jak typowe zastosowanie wzorca, etapy implementacji i powiązania z innymi wzorcami.

## ⌚ Klasyfikacja wzorców

Wzorce projektowe różnią się między sobą złożonością, poziomem szczegółowości i skalą w jakiej da się je zaimplementować w projektowanym systemie. Podoba mi się analogia do budowy dróg: można uczynić skrzyżowanie bezpieczniejszym montując tylko sygnalizację świetlną, albo tworząc wielopoziomowe rozjazdy z podziemnymi przejściami dla pieszych.

Najbardziej podstawowe i niskopoziomowe wzorce są często nazywane *idiomami*. Zazwyczaj stanowią funkcjonalność jakieśgo języka programowania.

Najbardziej uniwersalnymi i wysokopoziomowymi wzorcami są wzorce *architektoniczne*. Deweloperzy mogą implementować je w niemal każdym języku. W przeciwnieństwie do innych wzorców, mogą służyć zaprojektowaniu architektury całej aplikacji.

Ponadto, wszystkie wzorce można skategoryzować według ich *celu*, bądź przeznaczenia. Niniejsza książka dotyczy trzech głównych grup wzorców:

- **Wzorce kreacyjne** wprowadzają elastyczniejsze mechanizmy tworzenia obiektów i pozwalają na ponowne wykorzystanie istniejącego kodu.
- **Wzorce strukturalne** wyjaśniają jak składać obiekty i klasy w większe struktury, zachowując przy tym elastyczność i efektywność struktur.
- **Wzorce behavioralne** które zajmują się efektywną komunikacją i podziałem obowiązków pomiędzy obiektami.

## Kto wynalazł wzorce?

Dobre, ale niezbyt dokładne pytanie. Wzorce projektowe nie są tajemną, skomplikowaną nauką – wręcz przeciwnie. Są typowymi rozwiązaniami typowych problemów właściwych projek-

towaniu obiektowemu. Gdy jakieś rozwiązanie stosowane jest raz za razem w kolejnych projektach, ktoś w pewnym momencie nada mu nazwę i opisze je szczegółowo. Tak “odkrywa się” wzorce.

Koncepcja wzorców została po raz pierwszy opisana przez Christophera Alexandra w książce *Język wzorców. Miasta, budyńki, konstrukcja*<sup>1</sup>. Książka opisuje “język” służący projektowaniu środowiska miejskiego. Jednostkami tego języka są wzorce. Mogą one opisywać wysokość okien, ilość pięter, powierzchnię terenów zielonych w dzielnicach i tak dalej.

Ideę podchwycili czterej autorzy: Erich Gamma, John Vlissides, Ralph Johnson i Richard Helm. W roku 1995 opublikowali książkę *Wzorce projektowe: Elementy oprogramowania obiektowego wielokrotnego użytku*<sup>2</sup>, w której opisano 23 wzorce pozwalające poradzić sobie z różnymi problemami związanymi z projektowaniem obiektowym. Pozycja szybko stała się best-sellerem. Z racji długiego tytułu, ludzie zaczęli skracać go do “książka bandy czworga”, a następnie do “Książka GoF”.

Od tamtego czasu opisano wiele innych wzorców obiektowych. “Podejście wzorcowe” stało się bardzo popularne w innych obszarach programowania, więc wiele innych wzorców pojawiło się również poza areną projektowania obiektowego.

- 
1. *Język wzorców. Miasta, budyńki, konstrukcja:* <https://refactoring.guru/pl/pattern-language-book>
  2. *Wzorce projektowe: Elementy oprogramowania obiektowego wielokrotnego użytku:* <https://refactoring.guru/pl/gof-book>

# Dlaczego mam poznawać wzorce?

Prawda jest taka, że da się pracować jako programista przez wiele lat bez znajomości jakiegokolwiek wzorca. Jest wiele takich osób. Nawet wtedy jednak zdarza się, że zaimplementuje się któryś ze wzorców zupełnie nieświadomie. Po co więc się ich uczyć?

- Wzorce projektowe są zestawem narzędziowym składającym się z **wypróbowanych rozwiązań** typowych problemów pojawiających się podczas projektowania oprogramowania. Nawet jeśli nigdy się na te problemy nie natknęło, znajomość wzorców nadal się przydaje, bo uczy jak poradzić sobie z bardzo wieloma różnymi sytuacjami przy pomocy zasad projektowania obiektowego.
- Wzorce projektowe definiują wspólny język za pomocą którego możesz sprawniej komunikować się ze współpracownikami. Możesz na przykład powiedzieć: "O, tu możesz użyć Singletona" i wszyscy będą wiedzieć co masz na myśli. Nie trzeba tłumaczyć całej koncepcji, jeśli wszyscy znają wzorce z nazwy.

# ZASADY PROJEKTOWANIA OPROGRAMOWANIA

# Cechy dobrego projektu

Zanim przejdziemy do wzorców, pomówmy o projektowaniu architektury oprogramowania: do czego dążyć i czego się wystrzegać.

## Ponowne użycie kodu

Koszt oraz czas to dwie najważniejsze miary gdy tworzy się jakiekolwiek oprogramowanie. Mniej czasu poświęconego na produkcję oznacza wejście na rynek przed konkurencją. Mniejsze koszty produkcji pozwalają zainwestować więcej w marketing, co przysporzy więcej potencjalnych klientów.

**Ponowne wykorzystanie kodu** to jeden z najpowszechniejszych sposobów obniżenia kosztu tworzenia. Cel jest jasny: zamiast budować coś wciąż od nowa, dlaczego by nie wykorzystać istniejącego kodu w kolejnych projektach?

Pomysł wygląda dobrze na papierze, ale okazuje się, że aby uczynić istniejący kod nadającym się do wykorzystania w nowym kontekście, trzeba się sporo naprawić. Ścisłe spręganie pomiędzy komponentami, zależności od konkretnych klas zamiast interfejsów, programowanie operacji “na sztywno” – wszystko to redukuje elastyczność kodu i utrudnia ponowne wykorzystanie.

Stosowanie wzorców projektowych jest jednym ze sposobów na zwiększenie elastyczności komponentów oprogramowania i ułatwienie ponownego wykorzystania, ale czasem odbywa się to kosztem większego poziomu skomplikowania komponentów.

Oto słowa mądrości od Ericha Gamma<sup>1</sup>, jednego z ojców koncepcji wzorców, na temat roli wzorców projektowych w ponownym wykorzystywaniu kodu:

“

Wyróżniam trzy poziomy ponownego użycia.

Na najniższym poziomie, wykorzystuje się ponownie klasy: biblioteki klas, kontenery, może też “zespoły” wzorców jak kontener/iterator.

Najwyższym poziomem są frameworki. Bardzo usilnie próbują wydestylować podejmowane decyzje projektowe. Określały kluczowe abstrakcje rozwiązania problemu, prezentują je jako klasy i opisują związki między nimi. JUnit jest na przykład małym frameworkiem. Stanowi “Hello world!” frameworków. Ma zdefiniowane `Test`, `TestCase`, `TestSuite` i relacje.

Framework dotyczy na ogół elementów większych niż pojedyncze klasy. Ponadto, aby podpiąć się pod framework, zazwyczaj tworzy się podkласę jakiejś jego części. Stosuje się znaną z Hollywood zasadę “nie dzwoń do nas, my zadzwonimy do ciebie”.

Framework pozwala określić zachowanie i daje znać w którym

- 
1. Erich Gamma o elastyczności i ponowym wykorzystaniu:

<https://refactoring.guru/gamma-interview>

momencie musisz przejąć stery. Tak samo jak z JUnit, prawda? Informuje cię gdy chce wykonać test, ale reszta dzieje się we frameworku.

Jest też warstwa pośrednia. Tam właśnie widzę wzorce. Wzorce projektowe są i mniejsze i bardziej abstrakcyjne niż framework. W zasadzie opisują jak parę klas może być powiązanych i jak mogą współdziałać. Potencjał na ponowne wykorzystanie zwiększa się wraz z przejściem od klas do wzorców i wreszcie do frameworków.

Zaletą tej warstwy pośredniej jest fakt, że wzorce pozwalają ponownie korzystać z kodu w sposób mniej ryzykowny niż frameworki. Budowanie frameworku to ryzykowna i duża inwestycja. Wzorce zaś pozwalają na recykling idei projektowych i koncepcji niezależnie od konkretnego kodu.

”

## Rozszerzalność

**Zmiana** jest jedyną stałą w życiu programisty.

- Wydaliśmy grę na Windows, ale ludzie proszą o wersję na macOS.
- Stworzyliśmy framework graficznego interfejsu użytkownika z prostokątnymi przyciskami, ale wiele miesięcy później modne zrobiły się okrągłe.
- Mamy świetną autorską architekturę witryny e-commerce, ale miesiąc później klienci proszą o możliwość przyjmowania zamówień przez telefon komórkowy.

Każdy twórca oprogramowania może wymienić wiele takich przykładów. Dzieje się tak z kilku powodów.

Po pierwsze, lepsze rozumienie problemu przychodzi wraz z rozwiązywaniem go. Często kończąc pierwszą wersję aplikacji jest się gotowym pisać ją od nowa, bo widzi się więcej aspektów problemu. Z czasem przychodzi też rozwój umiejętności, a wtedy nasz dawny kod zaczyna wyglądać kiepsko.

Być może zmienił się jakiś czynnik na który nie mamy wpływu. Dlatego wielu deweloperów często przestawia się ze swoich pierwotnych planów na nowe. Wszyscy którzy dawniej tworzyli swoje aplikacje pod Flash, musieli zacząć migrację, gdy przeglądarka po przeglądarce kończyły wsparcie dla Flash.

Trzeci powód jest taki, że żaden cel nie jest ostateczny – klient był zachwycony aktualną wersją aplikacji, ale teraz zaczął zauważać kolejne obszary warte usprawnienia. Mało tego, czasem usprawnienia dotyczą funkcji o których nawet nie było mowy na etapie planowania. Pamiętaj, twój klient nie jest lekkomyślny: po prostu stworzyłeś coś, co otworzyło mu oczy na nowe możliwości.

Jest i pozytywna strona takich sytuacji: jeśli ktoś prosi o zmiany w twojej aplikacji, oznacza to, że jeszcze go ona obchodzi.

Dlatego też każdy doświadczony deweloper projektujący architekturę oprogramowania, stara się aby możliwe było dokonywanie zmian w przyszłości.

# Zasady projektowania

Czym jest dobre projektowanie oprogramowania? Jak można je zmierzyć? Jakich praktyk należy się trzymać aby to osiągnąć? Jak możesz uczynić architekturę elastyczną, stabilną i zrozumiałą?

To dobre pytania, ale niestety odpowiedzi zależą od rodzaju aplikacji jaką tworzysz. Niemniej jednak, istnieje wiele uniwersalnych zasad projektowania oprogramowania które mogą pomóc odpowiedzieć na te pytania w kontekście twojego projektu. Większość wzorców projektowych opisanych w niniejszej książce opiera się na tych zasadach.

# Hermetyzuj to, co się różni

Identyfikuj te aspekty aplikacji, które ulegają zmianom i rozdziel je od tego, co stałe.

Ta zasada pozwala zminimalizować skutki uboczne zmian.

Wyobraź sobie, że twój program jest okrętem narażonym na zderzenie z podwodną miną, która jest w stanie zatopić statek.

Wiedząc o tym, możesz podzielić kadłub na osobne przedziały które można bezpiecznie zapieczętować, ograniczając tym samym uszkodzenie do pojedynczego przedziału. A więc gdy statek natrafi na minę, utrzyma się na powierzchni.

W ten sam sposób można odizolować od siebie te obszary programu które różnią się, tworząc niezależne moduły i ochronić resztę kodu od niekorzystnych skutków zmian. W wyniku tego spędzisz mniej czasu na przywracanie funkcjonalności programu, implementowanie i testowanie zmian. Im mniej czasu poświęcasz na dokonywanie zmian, tym więcej pozostanie go na implementację nowych funkcji.

## Hermetyzacja na poziomie metody

Powiedzmy, że tworzysz witrynę e-commerce. Gdzieś w kodzie znajduje się metoda `pobierzSumęZamówienia` obliczająca całą

kowią wartość składanego zamówienia, wraz z opodatkowaniem.

Można się spodziewać, że kod związany z opodatkowaniem może ulec zmianie w przyszłości. Wysokość podatku zależy od kraju, regionu czy nawet miasta w którym mieszka klient. Jest też różnie naaliczana – zależnie od zmieniającego się prawa. Musisz więc często zmieniać metodę `pobierzSumęZamówienia`. Ale nawet patrząc na nazwę metody można wywnioskować, że nie obchodzi jej *jak* naaliczany jest podatek.

```
1 method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4         total += item.price * item.quantity
5
6     if (order.country == "US")
7         total += total * 0.07 // Podatek obrotowy w USA
8     else if (order.country == "EU"):
9         total += total * 0.20 // Europejski podatek VAT
10
11    return total
```

*PRZED: Kod naliczania podatku jest zmieszany z resztą kodu metody.*

Można wyekstrahować logikę naliczania podatku do odrębnej metody, ukrywając ją tym samym przed metodą pierwotną.

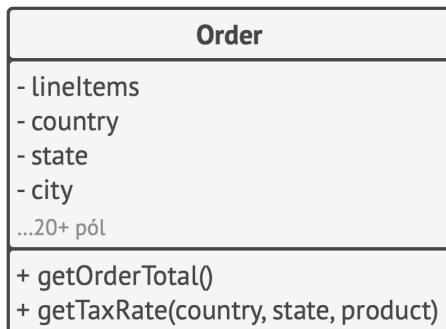
```
1 method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4         total += item.price * item.quantity
5
6     total += total * getTaxRate(order.country)
7
8     return total
9
10 method getTaxRate(country) is
11     if (country == "US")
12         return 0.07 // Podatek obrotowy w USA
13     else if (country == "EU")
14         return 0.20 // Europejski podatek VAT
15     else
16         return 0
```

*PO: możesz pozyskać wysokość podatku wywołując odpowiednią metodę.*

Zmiany dotyczące opodatkowania zostają odizolowane w pojedynczej metodzie. Co więcej, jeśli logika naliczania podatku stanie się zbyt skomplikowana, łatwiej będzie przenieść ją do osobnej klasy.

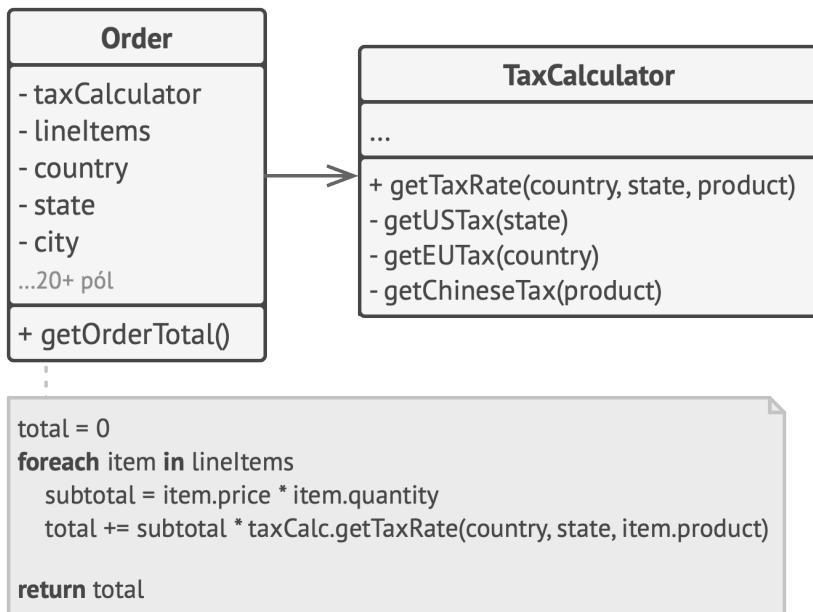
## Hermetyzacja na poziomie klasy

Z czasem możesz nadać więcej i więcej zadań metodzie która miała wykonywać jedno proste zadanie. Takie dodane zachowanie zwykle niesie ze sobą dodatkowe pomocnicze pola i metody, a główna odpowiedzialność klasy staje się mniej wyraźna. Wyekstrahowanie wszystkiego do nowej klasy może wszystko uprościć.



*PRZED: obliczanie podatku w klasie `Zamówienie`.*

Obiekty klasy `Zamówienie` delegują zadania związane z podatkami do obiektu wyspecjalizowanego w tym kierunku.



*PO: naliczanie podatku jest ukryte przed klasą `Zamówienie`.*

# Programuj pod interfejs, nie implementację

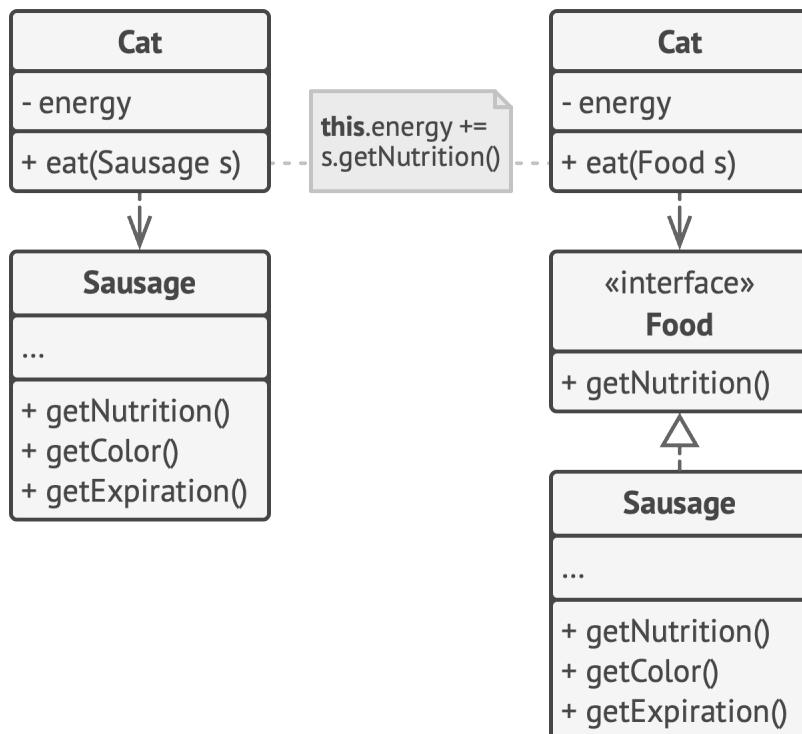
Programuj pod interfejs, a nie implementację. Opieraj się na abstrakcjach zamiast na konkretnych klasach.

Można stwierdzić, że projekt jest wystarczająco elastyczny jeśli w prosty sposób możesz go rozszerzyć i nie zepsuć przy tym istniejącego kodu. Upewnijmy się, że to twierdzenie jest prawdziwe na kolejnym kocim przykładzie. Kot wszystkożerny jest bardziej elastyczny niż taki który je wyłącznie parówki. Pierwszy z nich też może jeść parówki, bo są podzbiorem “wszystkiego”, ale możliwe jest rozszerzenie menu tego kota tak, by jadł dowolne inne jedzenie.

Jeśli chcesz pozwolić dwóm klasom na współpracę, możesz zacząć od uniezależnienia jednej od drugiej. Ja również od tego zaczynam. Ale istnieje inne, bardziej elastyczne podejście do ustanowienia współpracy obiektów:

1. Określ czego konkretnie potrzebuje jeden obiekt od drugiego: którą metodę wywołuje?
2. Opisz te metody w nowym interfejsie lub klasie abstrakcyjnej.
3. Spraw, by klasa stanowiąca zależność implementowała powyższy interfejs.

4. Uczę się drugą klasę zależną od tego interfejsu, a nie od konkretnej klasy. Może ona nadal współdziałać z obiektami pierwotnej klasy, ale połączenie jest teraz elastyczniejsze.

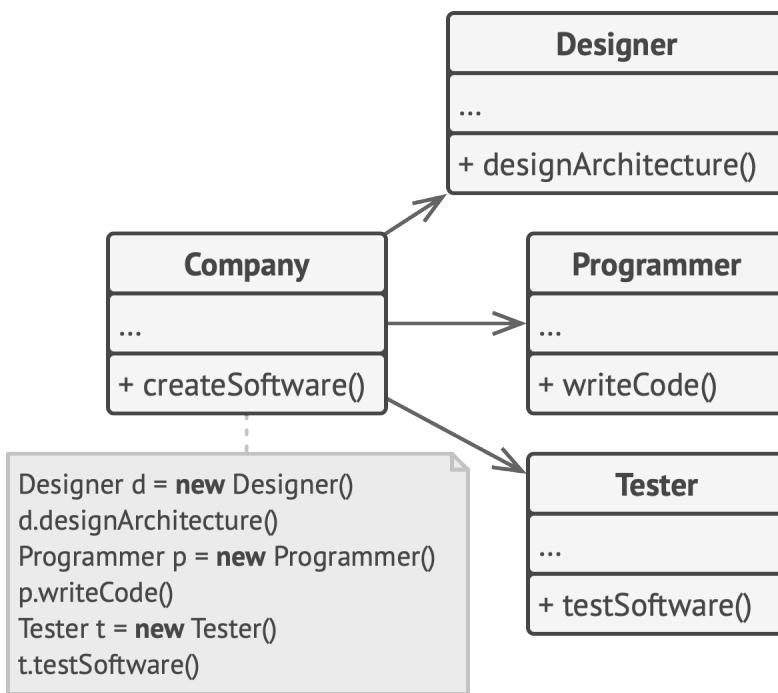


*Przed i po ekstrakcji interfejsu. Kod po prawej stronie jest elastyczniejszy niż ten po lewej, ale również bardziej skomplikowany.*

Po dokonaniu tej zmiany być może nie odczujesz różnicy od razu. Mało tego, kod stał się bardziej skomplikowany. Jednak warto to zrobić, jeśli wydaje ci się, że może to być dobry punkt zaczepienia dla ewentualnego poszerzenia funkcjonalności, albo że ktoś inny, korzystając z tego kodu, zechce go rozbudowywać od tego miejsca.

## Przykład

Popatrzmy na kolejny przykład który pokazuje w jaki sposób praca z obiektami za pośrednictwem interfejsów może być korzystniejsza niż opieranie się na konkretnych klasach. Wyobraź sobie, że tworzysz symulator firmy tworzącej oprogramowanie. Masz różne klasy, reprezentujące różne typy pracowników.

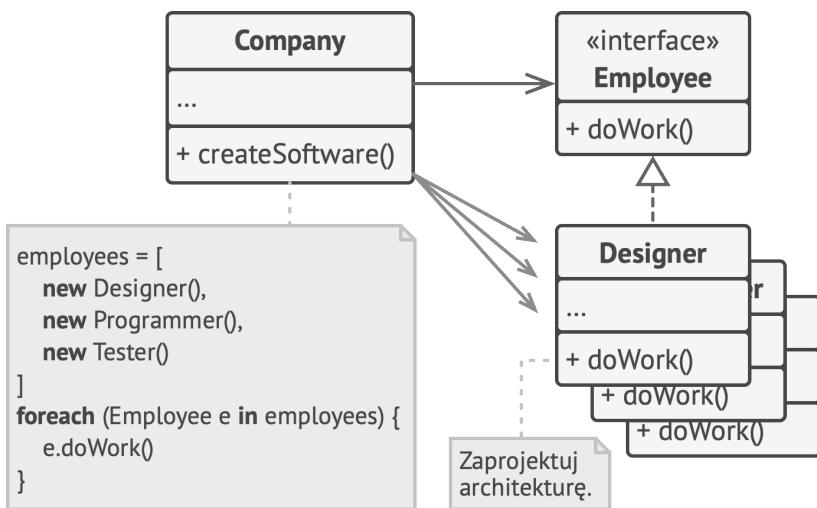


*PRZED: wszystkie klasy są ze sobą ścisłe sprzężone.*

Początkowo, klasa `Firma` jest ścisłe spręgnięta z konkretnymi klasami pracowników. Jednakże, mimo różnicy w ich implementacjach, możemy uogólnić pewne metody dotyczące

pracy, a następnie wyekstrahować wspólny interfejs dla wszystkich klas pracowników.

Zrobiwszy to, możemy zastosować w obrębie klasy `Firma` polimorfizm i korzystać z wszystkich obiektów odpowiadających pracownikom za pośrednictwem interfejsu `Pracownik`.

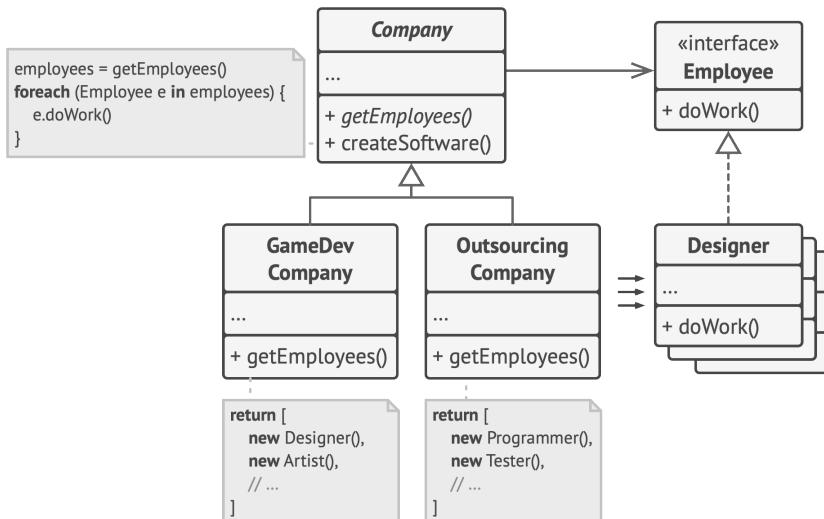


*LEPIEJ: polimorfizm pozwolił nam uprościć kod, ale reszta klasy `Firma` nadal jest zależna od konkretnych klas pracowników.*

Klasa `Firma` pozostaje sprzęgnięta z klasami pracowników. To niedobrze, bo jeśli wprowadzimy nowe typy firm, współpracujące z innymi typami pracowników, będzie trzeba nadpisać większość klasy `Firma` zamiast wykorzystać jej kod ponownie.

Aby rozwiązać ten problem, możemy zadeklarować metody pobierania danych pracowników jako *abstrakcyjne*. Każda kon-

kretna firma zaimplementuje tę metodę w inny sposób, tworząc wyłącznie potrzebnych jej pracowników.



*PO: główna metoda klasy `Firma` jest niezależna od konkretnych klas pracowników. Obiekty przedstawiające pracowników tworzone są w konkretnych podklasach firm.*

Po tej zmianie, klasa `Firma` stała się niezależna od różnorakich klas pracowników. Można teraz rozszerzyć tę klasę i wprowadzić nowe typy firm oraz pracowników, ponownie wykorzystując część kodu bazowej klasy firmy. Rozszerzenie bazowej klasy firmy nie zepsuje innego kodu, który już od niej zależy.

Jest to przy okazji pokaz wprowadzania wzorca projektowego. Konkretnie, zastosowaliśmy wzorzec *Metody wytwórczej*. Nie martw się: omówię ją szczegółowo w dalszej części.

# Preferuj kompozycję ponad dziedziczenie

Dziedziczenie jest bodaj najbardziej oczywistym i prostym sposobem na ponowne wykorzystanie kodu klas. Masz dwie klasy z tym samym kodem. Stwórz dla nich wspólną klasę bazową i przenieś do niej podobny kod. Pestka!

Niestety dziedziczenie niesie ze sobą także obciążenia które stają się oczywiste dopiero gdy twój program ma już mnóstwo klas a zmiana czegokolwiek staje się trudna. Oto lista tych problemów.

- **Podklasa nie zredukuje interfejsu klasy bazowej.** Musisz zaimplementować wszystkie abstrakcyjne metody klasy nadzędnej nawet jeśli nie zamierzasz z nich korzystać.
- **Nadpisując metody trzeba mieć pewność że nowe zachowanie jest kompatybilne z tym w klasie bazowej.** Jest to istotne, gdyż obiekty podklas mogą zostać przekazane do jakiegokolwiek kodu który oczekuje obiektów klasy bazowej i nie chcemy tego kodu zepsuć.
- **Dziedziczenie psuje hermetyzację klasy bazowej** ponieważ wewnętrzne szczegóły klasy-rodzica stają się dostępne dla podklasy. Może pojawić się także odwrotna sytuacja, w której programista czyni szczegóły podklasy widocznymi dla klasy bazowej w celu ułatwienia dalszej rozbudowy.

- **Podklasy są ściśle sprzęgnięte z klasami bazowymi.** Każda zmiana w klasie bazowej może zepsuć funkcjonalność podklas.
- **Próba ponownego wykorzystania kodu poprzez dziedziczenie może doprowadzić do utworzenia równoległych hierarchii dziedziczenia.** Dziedziczenie zazwyczaj odbywa się w jednym wymiarze. Gdy zaczyna odbywać się w kilku płaszczyznach, trzeba tworzyć mnóstwo kombinacji klas, rozbudowując ich hierarchię do absurdalnych rozmiarów.

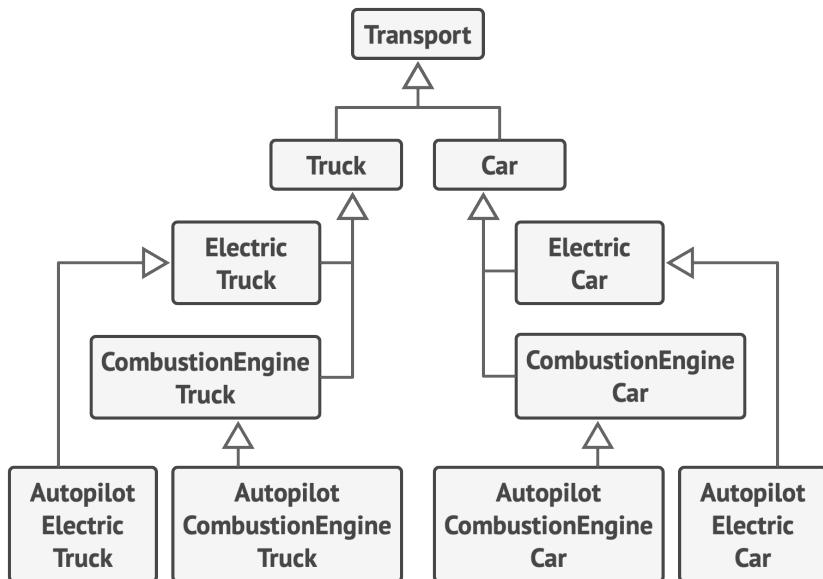
Istnieje alternatywa dla dziedziczenia zwana *kompozycją*. Podczas gdy dziedziczenie reprezentuje relację “jest czymś” pomiędzy klasami (samochód *jest* środkiem transportu), kompozycja przedstawia relację “posiada coś” (samochód *osiada* silnik).

Warto wspomnieć, że ta zasada dotyczy również agregacji – czyli luźniejszej formy kompozycji, w której jeden obiekt może przechowywać odniesienie do innego, ale nie zarządza jego cyklem życia. Przykładowo: samochód *osiada* kierowcę, ale on lub ona mogą skorzystać z innego samochodu albo pójść piechotą – *bez samochodu*.

## Przykład

Wyobraź sobie, że musisz stworzyć aplikację katalogową dla producenta samochodów. Firma produkuje zarówno auta osobowe, jak i ciężarowe. Mogą być elektryczne lub na paliwo i

wszystkie modele mają albo sterowanie ręczne, albo autopilota.

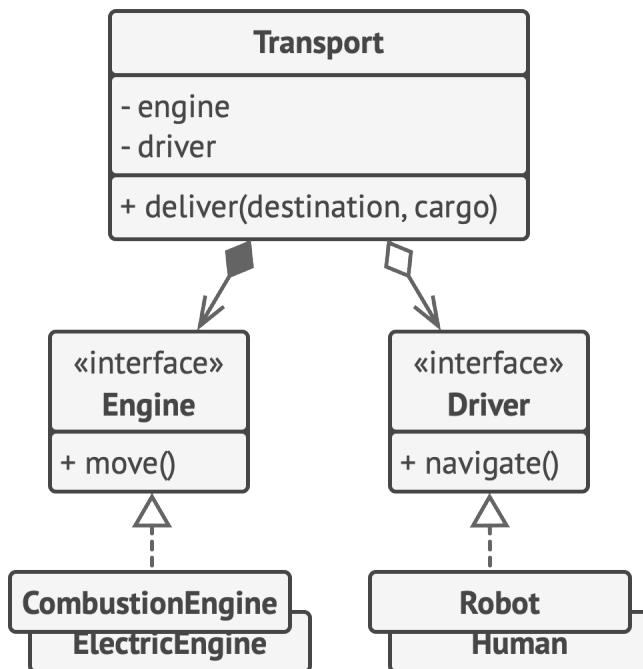


*DZIEDZICZENIE:* Rozszerzanie klasy w kilku wymiarach (typ ładunku × typ silnika × typ sterowania) może prowadzić do kombinatorycznej eksplozji podklas.

Jak widać, każdy dodatkowy parametr powoduje rozmnożenie liczby podklas. Jest też mnóstwo powtarzającego się kodu pomiędzy podklasami, ponieważ podklaśa nie może dziedziczyć po dwóch klasach bazowych jednocześnie.

Mözesz rozwiązać ten problem stosując kompozycję. Obiekty-samochody zamiast implementować zachowanie samodzielnie, mogą je delektować innym obiektom.

Dodatkowym pozytkiem z tego jest możliwość zamiany zachowania obiektu w trakcie działania programu. Na przykład można dokonać wymiany obiektu silnika powiązanego z autem, poprzez przypisanie samochodowi innego obiektu.



*KOMPOZYCJA: wiele “wymiarów” funkcjonalności wyekstrahowanych do odrębnych hierarchii klas.*

Taka struktura klas przywołuje na myśl wzorzec *Strategia*, który poznamy w dalszej części książki.

# Zasady SOLID

Teraz, gdy znasz podstawowe zasady projektowania, spójrzmy na pięć z nich które powszechnie znane są jako zasady SOLID. Robert Martin zaprezentował je w książce *Zwinne wytwarzanie oprogramowania. Najlepsze zasady, wzorce i praktyki*<sup>1</sup>.

*SOLID* to mnemonik opisujący pięć zasad projektowania mające ułatwić zrozumienie projektu oprogramowania oraz uczynić projekt elastycznym i łatwym w utrzymaniu.

Jak ze wszystkim co w życiu dobre, bezmyślne stosowanie tych zasad może czynić więcej szkody, niż pożytku. Koszt wprowadzenia tych zasad do architektury programu może uczynić ją bardziej skomplikowaną niż trzeba. Wątpię, czy istnieje jakieś znane oprogramowanie które byłoby zgodne ze wszystkimi zasadami jednocześnie. Warto wprawdzie do nich dążyć, ale wykaż się pragmatyzmem i nie traktuj wszystkiego co tu jest napisane jako dogmat.

---

1. *Zwinne wytwarzanie oprogramowania. Najlepsze zasady, wzorce i praktyki:*  
<https://refactoring.guru/pl/principles-book>

# S

## Zasada pojedynczej odpowiedzialności Single Responsibility Principle

Klasa powinna podlegać zmianie tylko z jednego powodu.

Spróbuj uczynić każdą klasę odpowiedzialną za jedną tylko część funkcjonalności oferowanej przez oprogramowanie i niech ta odpowiedzialność będzie całkowicie hermetyzowana (inaczej mówiąc, *ukryta*) przez klasę.

Głównym celem tej zasady jest redukcja złożoności. Nie trzeba wynajdywać skomplikowanego projektu dla programu który składa się tylko z 200 linii kodu. Wystarczy tuzin ładnie napisanych metod.

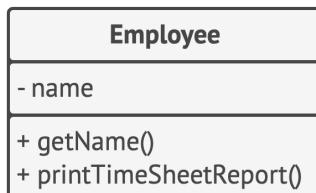
Problemy przychodzą gdy program zaczyna się rozrastać i ulegać zmianom. W którymś momencie klasy rosną tak, że trudno zapamiętać ich szczegóły. Nawigacja w kodzie staje się czasochłonna i musisz przeglądać klasy albo i cały kod by znaleźć jakiś szczególny element. Liczba podmiotów w programie powoduje w twojej głowie przepełnienie stosu i tracisz kontrolę nad kodem.

Poza tym, jeśli klasa zaczyna odpowiadać za zbyt wiele rzeczy, trzeba ją modyfikować za każdym razem gdy coś ulegnie zmianie. Modyfikując wprowadzasz ryzyko popsucia innych części klasy, które miały pozostać bez zmian.

Jeśli wydaje ci się, że coraz trudniej jest ci skupić się na konkretnych aspektach programu, przypomnij sobie o zasadzie pojedynczej odpowiedzialności i rozważ podział klas na kilka części.

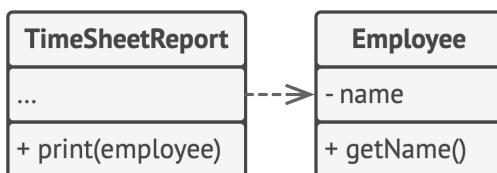
## Przykład

Klasa `Pracownik` może wymagać modyfikacji z wielu powodów. Pierwszy z nich to główne zadanie tej klasy: zarządzanie danymi o pracownikach. Jest też kolejny powód: format raportu ewidencji czasu pracy może ulec zmianie, a wtedy musisz zmienić także kod klasy.



*PRZED:* klasa zawiera wiele różnych zachowań.

Problem można rozwiązać przenosząc zachowanie odnoszące się do drukowania raportów do odrębnych klas. Taka zmiana pozwoli przenieść też inne fragmenty dotyczące sprawozdawczości do nowej klasy.



*PO:* Dodatkowe zachowanie w swojej własnej klasie.

# O Zasada otwarte/zamknięte pen/Closed Principle

Klasy powinny być otwarte na rozszerzenie ale zamknięte dla modyfikacji.

Główną ideą tej zasady jest zapobieżenie popuszcia istniejącego kodu gdy implementuje się nową funkcjonalność.

O klasie mówimy, że jest *otwarta*, jeśli możemy ją rozszerzyć, stworzyć jej podklasę i swobodnie dodawać pola lub metody, nadpisywać zachowanie klasy bazowej, itp. Niektóre języki programowania pozwalają na ograniczenie dalszego rozszerzania klasy za pomocą specjalnych słów kluczowych, jak `final`. Klasa wówczas przestaje być otwartą. Klasa staje się zaś *zamknięta* (inaczej mówiąc *zakończona*) i jest w stu procentach gotowa do użycia przez inne klasy, gdy jej interfejs jest wyraźnie określony i nie ulegnie zmianie w przyszłości.

Gdy pierwszy raz dowiedziałem się o tej zasadzie, było to dla mnie niejasne, bo słowa *otwarte* & *zamknięte* wzajemnie się wykluczają. Ale w kontekście tej zasady, klasa może być jednocześnie otwarta (dla rozszerzania) oraz zamknięta (dla modyfikacji).

Gdy jakaś klasa jest już opracowana, przetestowana, opisana i zawarta w jakimś frameworku lub w inny sposób w aplikacji, to zmiana jej kodu niesie ryzyko. Zamiast zmieniać kod klasy bez-

pośrednio, tworzy się podklasę i nadpisuje te elementy pierwotnej klasy, których zachowanie chce się zmienić. Osiągnie się więc cel i przy okazji nie zepsuje istniejących klientów pierwotnej klasy.

Tej zasady nie stosuje się jednak wobec każdej zmiany klasy. Jeśli wiesz o bugu w klasie, to go napraw, zamiast tworzyć podklasę. Klasa pochodna nie powinna być odpowiedzialna za problemy klasy-rodzica.

## Przykład

Masz aplikację e-commerce zawierającą klasę `Zamówienie`, która oblicza koszty wysyłki, a metody wysyłki są zaprogramowane “na sztywno” w obrębie klasy. Jeśli chcesz dodać nowy sposób wysyłki, musisz zmienić kod klasy `Zamówienie`, ryzykując jej zepsucie.

Order
- lineItems
- shipping

+ getTotal()
+ getTotalWeight()
+ setShippingType(st)
+ getShippingCost() ○
+ getShippingDate()

```

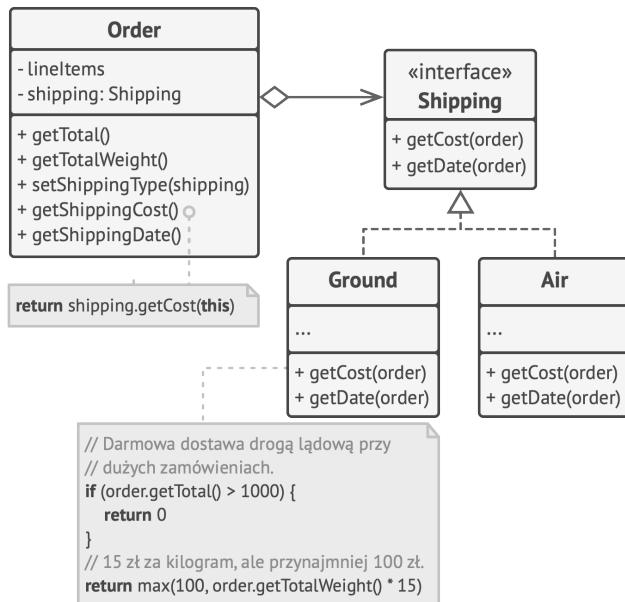
if (shipping == "ground") {
    // Darmowa dostawa drogą lądową przy
    // dużych zamówieniach.
    if (getTotal() > 1000) {
        return 0
    }
    // 15 zł za kilogram, ale przynajmniej 100 zł.
    return max(100, getTotalWeight() * 15)
}

if (shipping == "air") {
    // 30 zł za kilogram, ale przynajmniej 200 zł.
    return max(200, getTotalWeight() * 30)
}

```

*PRZED: trzeba dokonywać zmian w klasie `Zamówienie` za każdym razem gdy dodaje się do aplikacji nowy sposób wysyłki.*

Możesz rozwiązać ten problem stosując wzorzec *Strategia*. Začnij od wyekstrahowania metod dotyczących wysyłki do oddzielnych klas ze wspólnym interfejsem.



*PO: dodanie nowego sposobu wysyłki nie wymaga zmian istniejących klas.*

Teraz, gdy musisz zaimplementować nowy sposób wysyłki, możesz wydzielić nową klasę z interfejsu `Wysyłka` bez konieczności ruszania kodu klasy `Zamówienie`. Kod kliencki klasy `Zamówienie` będzie łączył zamówienia z obiektami metod wysyłki zgodnie z wyborem przez użytkownika (dokonanym przez GUI).

Dodatkowo, to rozwiązanie pozwala przenieść obliczanie czasu dostarczenia do bardziej odpowiednich klas, zgodnie z *zasadą pojedynczej odpowiedzialności*.

**L**

## Zasada podstawienia Liskov **liskov Substitution Principle<sup>1</sup>**

Rozszerzając klasę, trzeba pamiętać, aby było możliwe przekazywanie obiektów nowej podklasy w miejsce obiektów klasy bazowej bez psucia kodu klienta.

Oznacza to, że podklasa powinna pozostać kompatybilna z zachowaniem klasy bazowej. Nadpisując metodę, należy rozszerzać zachowanie klasy bazowej zamiast zamieniać je na coś zupełnie innego.

Zasada podstawienia jest zestawem czynności sprawdzających które pomagają przewidzieć kompatybilność podklasy z kodem który był w stanie współdziałać z obiektami klasy bazowej. Ta koncepcja jest kluczowa gdy tworzy się biblioteki oraz frameworki, gdyż twoje klasy będą używane przez inne osoby, a na ich kod nie ma się bezpośredniego wpływu.

W przeciwieństwie do innych zasad projektowania, które pozostawiają sporo przestrzeni na własną interpretację, zasada podstawienia określa zestaw formalnych wymagań wobec podklas, a konkretnie wobec metod podklas. Przyjrzymy się tym wymaganiom.

- 
1. Zasadę tę sformułowała w 1987 roku Barbara Liskov w swojej pracy *Hierarchia i abstrakcja danych* (ang. *Data abstraction and hierarchy*): <https://refactoring.guru/liskov/dah>

- **Typy parametrów metody podklasy powinny być zgodne z lub bardziej abstrakcyjne niż typy parametrów metody klasy bazowej.** Zagmatwane? Spójrzmy na przykład.
  - Założmy że istnieje klasa z metodą służącą karmieniu kotów: `nakarm(Kot c)`. Kod klienta zawsze przekazuje tej metodzie obiekty-koty.
  - **Poprawnie:** Powiedzmy, że stworzyłeś podkласę nadpisującą tę metodę w taki sposób, aby mogła wykarmić dowolne zwierzę (klasę bazową kotów): `nakarm(Zwierzę c)`. Gdy teraz przekaże się kodowi klienckiemu obiekt podklasy zamiast obiektu klasy bazowej – wszystko będzie nadal działać poprawnie. Ta metoda może nakarmić każde zwierzę, więc nakarmi i kota.
  - **Niepoprawnie:** Stworzyłeś kolejną podklasę i ograniczyłeś metodę karmienia tak, że przyjmuje wyłącznie koty bengalskie (podklasę kotów): `nakarm(KotBengalski c)`. Co się stanie z kodem klienckim jeśli powiążesz go z takim obiektem zamiast z obiektem pierwotnej klasy? Skoro metoda służy karmieniu wyłącznie konkretnej rasy kota, to nie nakarmi innych kotów, tym samym psując powiązaną funkcjonalność.
- **Typ obiektu zwracanego przez metody podklasy powinien być zgodny z lub być podtypem obiektu zwracanego przez metodę klasy bazowej.** Jak widzisz, wymagania co do typu zwracanego są odwrotne do wymagań dotyczących typów parametrów.

- Założmy, że masz klasę z metodą `kupKota(): Kot`. Kod kliencki oczekuje przekazania mu dowolnego kota w wyniku uruchomienia tej metody.
- **Poprawnie:** Podklasa nadpisuje metodę w następujący sposób: `kupKota(): KotBengalski`. Klient otrzymuje kota bengalskiego, który też jest kotem, więc wszystko jest w porządku.
- **Niepoprawnie:** Podklasa nadpisuje metodę w następujący sposób: `kupKota(): Zwierze`. W tej sytuacji kod kliencki zepsuje się, gdyż otrzyma nieznane mu generyczne zwierzę (aligatora? niedźwiedzia?) które nie pasuje do struktury tworzonej z myślą o kotach.

Kolejny anty-przykład pochodzi ze świata języków programowania o typowaniu dynamicznym: metoda klasy bazowej zwraca łańcuch znaków, ale nadpisana metoda zwraca liczbę.

- **Metoda w podklasie nie powinna rzucać wyjątków o typie którego nie rzuca metoda klasy bazowej.** Innymi słowy, typy wyjątków powinny być *zgodne z* lub być *podtypami* wyjątków które może rzucać metoda klasy bazowej. Ta zasada bierze się z faktu, że bloki kodu `try-catch` w kodzie klienckim oczekują konkretnych typów wyjątków, które zwykle rzuca metoda klasy bazowej. Dlatego wyjątek nieoczekiwanej typu może nie zostać przechwycony przez kod defensywny klienta i zakłócić wykonywanie aplikacji.

W większości nowoczesnych języków programowania, szczególnie typowanych statycznie (Java, C# i inne) te zasady są częścią samego języka. Nie da się nawet skompilować programu który ich nie przestrzega.

- **Podklasa nie powinna wzmacniać warunków wstępnych.** Przyjmijmy, że metoda klasy bazowej przyjmuje parametr typu `int`. Jeśli podklasa nadpisze tę metodę i będzie wymagać zawsze wartości dodatnich (rzuci wyjątek w przypadku ujemnej wartości) to wzmacnia warunek wstępny. Kod kliencki który wcześniej działał poprawnie przekazując do metody ujemne wartości, teraz się popsuje współpracując z obiektem podklasy.
- **Podklasa nie powinna osłabiać warunków końcowych.** Założymy, że masz klasę z metodą działającą na bazie danych. Metoda tej klasy powinna zawsze zamykać wszystkie aktywne połączenia z bazą po zwróceniu wartości.

Stworzyłeś podkласę i zmieniłeś ją tak, że połączenia z bazą pozostają otwarte umożliwiając ich ponowne użycie. Ale klient może nie wiedzieć nic o twoich intencjach. Ponieważ spodziewa się zamknięcia połączenia przez metody, może zwyczajnie przerwać wykonywanie programu po wywołaniu metody, zasmiecając system osieroconymi połączonymi z bazą.

- **Niezmienniki klasy bazowej muszą zostać zachowane.** Jest to bodaj najmniej formalna reguła. *Niezmienniki* to warunki w ja-

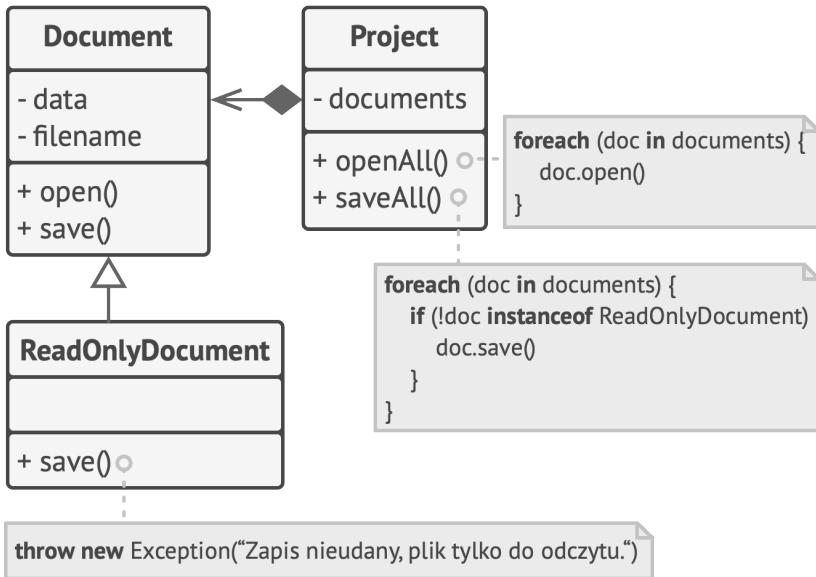
kich obiekt ma sens. Na przykład niezmiennikami kota są: ilość łap, ogon, umiejętność miauczenia, itp. Mylącą częścią niezmienników jest to, że chociaż mogą być wyraźnie zdefiniowane w formie interfejsów lub założeń w metodach, to mogą też być implikowane przez niektóre testy jednostkowe lub oczekiwania kodu klienckiego.

Zasadę niezmienników najłatwiej złamać wskutek błędnego zrozumienia lub niezrealizowania wszystkich niezmienników złożonej klasy. Dlatego też najbezpieczniejszym sposobem rozszerzenia klasy jest wprowadzenie nowych pól i metod i nie ruzszanie istniejących składowych klasy bazowej. Oczywiście nie zawsze jest to możliwe.

- **Podklasa nie powinna zmieniać wartości prywatnych pól klasy bazowej. Co takiego? Jak to w ogóle możliwe?** Okazuje się, że niektóre języki programowania pozwalają na dostęp do prywatnych składowych klasy poprzez mechanizm refleksji. Inne języki (Python, JavaScript) w ogóle nie chronią prywatnych składowych.

## Przykład

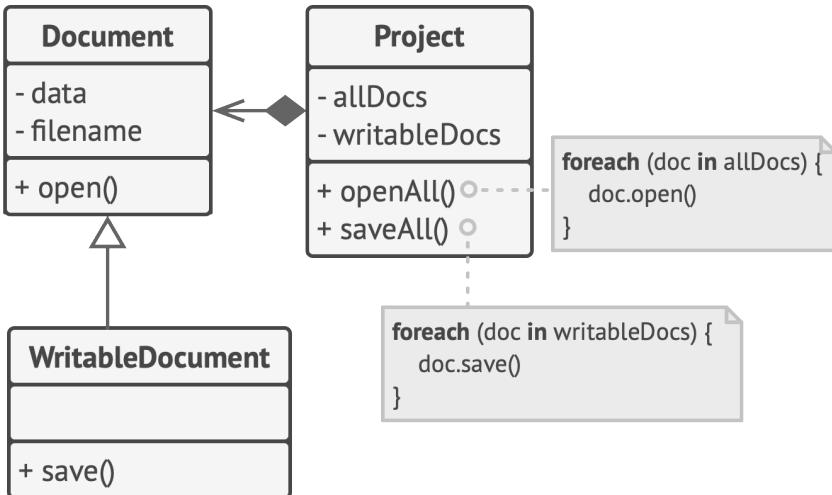
Popatrzmy na przykład hierarchii klas dokumentów która łamie zasadę podstawienia.



*PRZED: zapisywanie nie ma sensu w przypadku dokumentu tylko do odczytu, więc podkласa próbuje to rozwiązać zmieniając zachowanie klasy bazowej w nadpisywanej metodzie.*

Metoda `zapisz` w podklasie `DokumentTylkoDoOdczytu` rzuca wyjątek gdy ktoś próbuje ją wywołać. Metoda klasy bazowej nie ma tego ograniczenia. Oznacza to, że kod kliencki nie będzie działał poprawnie jeśli nie wprowadzi się sprawdzania typu dokumentu przed próbą zapisu.

Tak powstały kod łamie też zasadę otwarte/zamknięte, ponieważ kod kliencki staje się zależny od konkretnych klas dokumentów. Jeśli wprowadzi się nową podkласę dokumentu, trzeba zmienić kod kliencki tak, aby ją obsługiwał.



*PO: problem rozwiązywany poprzez uczynienie klasy dokumentu tylko do odczytu, klasą bazową całej hierarchii.*

Można rozwiązać ten problem przeprojektowując hierarchię klas: podklasa powinna rozszerzać zachowanie klasy bazowej, więc dokument tylko do odczytu stanie się klasą bazową hierarchii. Dokument obsługujący zapis staje się podkąską rozszerzającą klasę bazową o funkcjonalność zapisu.

## I Zasada segregacji interfejsów Interface Segregation Principle

Klientom nie powinno się narzucać zależności od nieużywanych metod.

Staraj się tworzyć interfejsy na tyle wąsko wyspecjalizowane, żeby klienci nie musieli implementować zachowań których nie potrzebują.

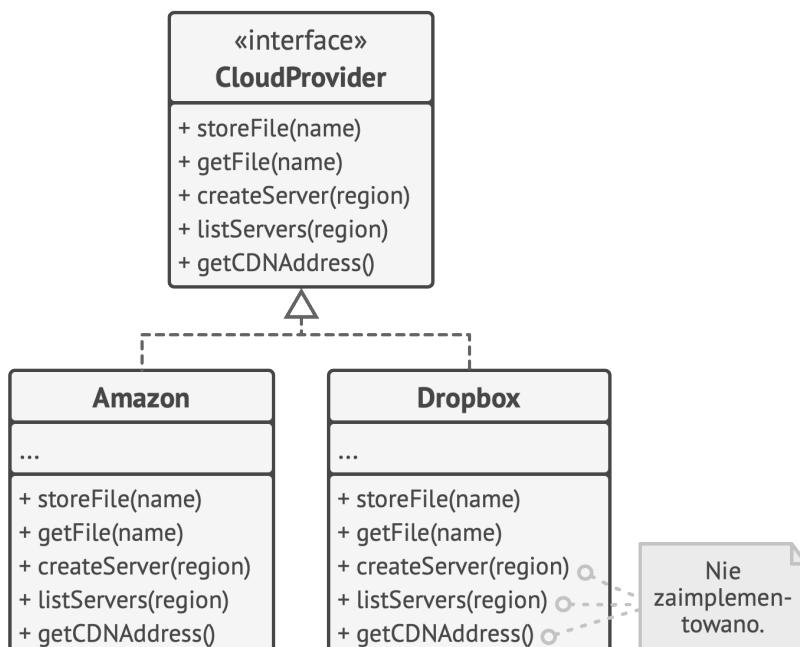
Zgodnie z zasadą segregacji interfejsów, powinno się najpierw rozłożyć “opasłe” interfejsy na drobniejsze i bardziej specjalistyczne. Klienci powinni implementować tylko te metody, które są im faktycznie potrzebne. W przeciwnym wypadku zmiana “opasłego” interfejsu popsułaby nawet ten kod kliencki, który nie korzysta ze zmienianych metod.

Dziedziczenie pozwala danej klasie mieć tylko jedną klasę bazową, ale nie ogranicza ilości interfejsów które klasa może naraz zaimplementować. Dlatego nie ma potrzeby umieszczania masy niezwiązańzych ze sobą metod w jednym interfejsie. Podziel je na mniejsze, bardziej szczegółowe interfejsy – nadal będzie możliwa zaimplementowanie je wszystkie w jednej klasie gdy zajdzie potrzeba.

## Przykład

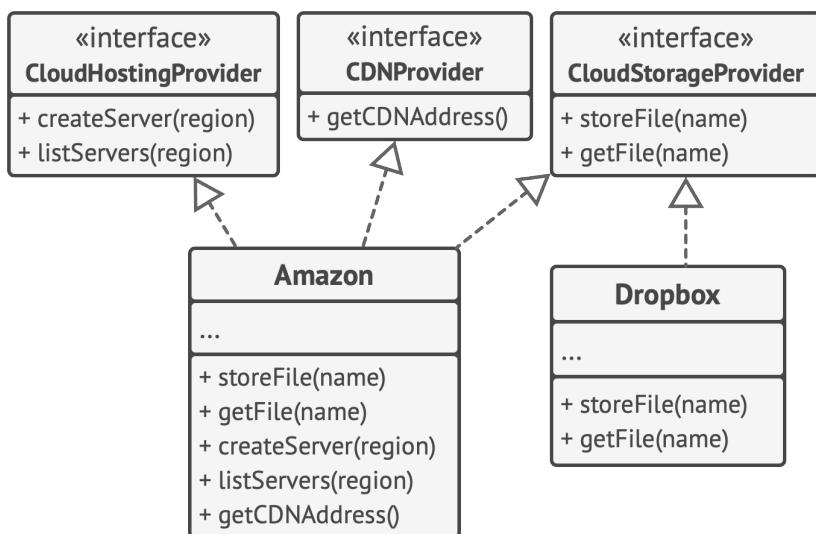
Wyobraźmy sobie, że stworzyliśmy bibliotekę ułatwiającą integrację aplikacji z różnymi dostawcami chmur obliczeniowych. Wstępna wersja wspierała tylko Amazon Cloud, ale za to obsługiwała pełen zestaw jej usług i funkcjonalności.

W owym czasie zakładaliśmy, że wszyscy dostawcy oferują tak samo szerokie spektrum funkcjonalności jak Amazon. Ale gdy przyszło implementować wsparcie innych dostawców, okazało się że większość interfejsów biblioteki jest zbyt szeroka. Niektóre metody opisywały funkcje których inni dostawcy po prostu nie mieli w ofercie.



*PRZED: nie każdy klient jest w stanie sprostać wymaganiom implementacji rozbuchanego interfejsu.*

Można wprawdzie zaimplementować te metody w bardzo ograniczonej postaci, jednak nie byłoby to eleganckie rozwiązanie. Lepszym wyjściem jest rozbicie interfejsu na mniejsze części. Klasy które są w stanie zaimplementować pierwotny interfejs mogą teraz po prostu zaimplementować wiele konkretniejszych interfejsów. Inne klasy zaś mogą implementować tylko te interfejsy, których metody są dla nich sensowne.



*PO: jeden obszerny interfejs rozbito na zestaw mniejszych, bardziej wyspecjalizowanych interfejsów.*

Jak w przypadku innych zasad – i tu można przesadzić. Nie warto dzielić interfejsu który już i tak jest dość specjalistyczny. Pamiętaj, że im więcej stworzysz interfejsów, tym bardziej skomplikuje się kod. Dąż do równowagi.

# D Zasada odwrócenia zależności Dependency Inversion Principle

Wysokopoziomowe klasy nie powinny być zależne od niskopoziomowych. Obie grupy powinny być zależne od abstrakcji. Abstrakcje nie powinny być zależne od szczegółów. Szczegóły z kolei powinny zależeć od abstrakcji.

Zazwyczaj projektując oprogramowanie można rozróżnić dwa poziomy klas.

- **Klasy niskopoziomowe** implementują podstawowe operacje takie jak współpraca z dyskiem, przesyłanie danych przez sieć, łączenie z bazą danych, itp.
- **Klasy wysokopoziomowe** zawierają złożoną logikę biznesową która zleca niskopoziomowym klasom wykonywanie działań.

Czasami projektuje się niskopoziomowe klasy jako pierwsze i dopiero potem rozpoczyna się pracę nad wysokopoziomowymi. Jest to powszechne podejście gdy zaczyna się pracę nad prototypem nowego systemu i nie jest się nawet pewnym co właściwie będzie możliwe na wyższym poziomie, bo szczegóły są jeszcze niezaimplementowane i niejasne. Na ogół przy takim podejściu klasy logiki biznesowej stają się zależne od prymitywnych klas niskopoziomowych.

Zasada odwracania zależności każe zmienić kierunek tej zależności.

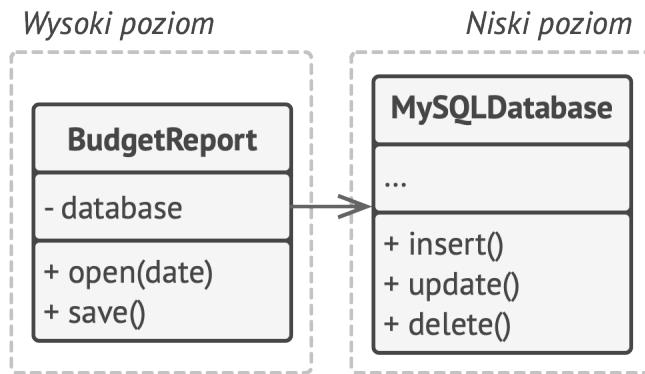
1. Na początek trzeba opisać interfejsy niskopoziomowych operacji od których będą zależeć wysokopoziomowe klasy, najlepiej stosując terminologię biznesową. Na przykład, logika biznesowa powinna wywoływać metodę `otworzRaport(plik)` zamiast ciąg prostszych poleceń `otworzPlik(x)`, `wczytajBajty(n)`, `zamknijPlik(x)`. Takie interfejsy stanowią wysoki poziom.
2. Następnie można uczynić klasy wysokopoziomowe zależnymi od tych interfejsów, zamiast od konkretnych klas niskiego poziomu. Taka zależność będzie dużo bardziej subtelna niż pierwotna.
3. Gdy klasy niskopoziomowe zaimplementują te interfejsy, stają się zależne od logiki biznesowej, odwracając tym samym pierwotny kierunek zależności.

Zasada odwrócenia zależności często idzie w parze z *zasadą otwarcie/zamknięcie*: można rozszerzać niskopoziomowe klasy by korzystać z nich w innych klasach logiki biznesowej, bez psucia istniejących klas.

## Przykład

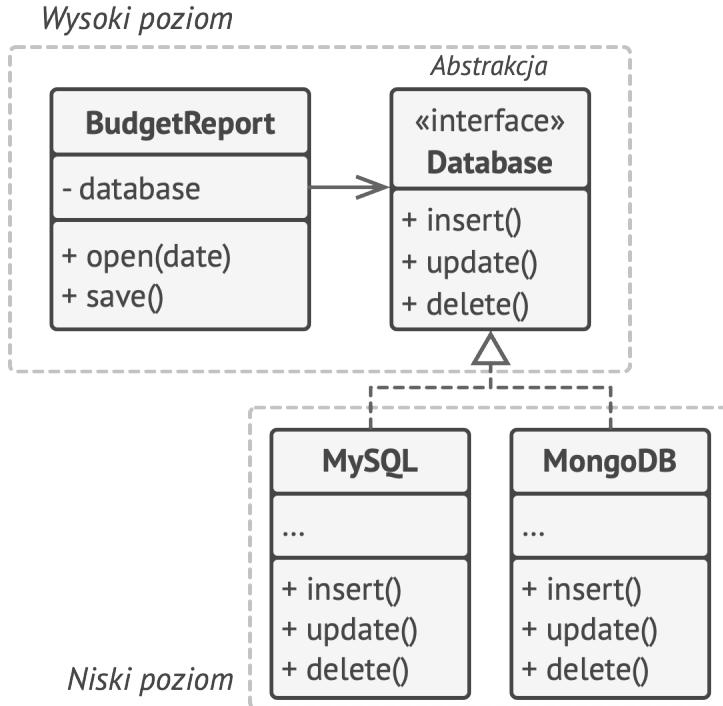
W poniższym przykładzie, wysokopoziomowa klasa sprawozdawczości budżetowej korzysta z niskopoziomowej klasy bazodanowej by wczytywać i zachowywać swoje dane. Oznacza to, że każda zmiana w klasie niskiego poziomu, jak dostosowanie jej do nowej wersji serwera baz danych, może wpłynąć na

klasę wyższego poziomu, której nie powinny obchodzić szczegóły przechowywania danych.



*PRZED: klasa wysokiego poziomu jest zależna od klasy niskiego poziomu.*

Można tę sytuację naprawić tworząc wysokopoziomowy interfejs który opisuje operacje odczytu/zapisu i prezentując go klasie odpowiedzialnej za raporty. Po dokonaniu tego możliwe będzie zmienianie lub rozszerzenie pierwotnej klasy niskiego poziomu aby implementowała interfejs odczytu/zapisu zadeklarowany przez logikę biznesową.



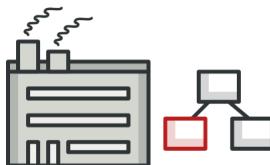
PO: klasy niskopoziomowe zależą od wysokopoziomowej abstrakcji.

Rezultatem jest odwrócenie pierwotnej zależności: teraz klasy niskopoziomowe są zależne od wysokopoziomowej abstrakcji.

# KATALOG WZORCÓW PROJEKTOWYCH

# Wzorce kreacyjne

Wzorce kreacyjne są źródłem różnych mechanizmów tworzenia obiektów, zwiększających elastyczność i ułatwiających ponowne użycie kodu.



## Metoda wytwórcza

Factory Method

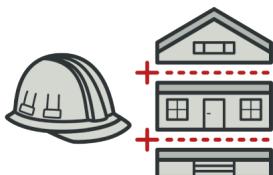
Udostępnia interfejs do tworzenia obiektów w klasie bazowej, ale pozwala podklasom zmieniać typ tworzonych obiektów.



## Fabryka abstrakcyjna

Abstract Factory

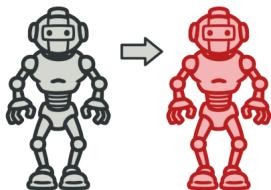
Umożliwia tworzenie rodzin spokrewnionych ze sobą obiektów bez określania ich konkretnych klas.



## Budowniczy

Builder

Daje możliwość konstruowania złożonych obiektów krok po kroku. Wzorzec ten pozwala produkować różne typy oraz reprezentacje obiektu używając tego samego kodu konstrukcyjnego.



## Prototyp

Prototype

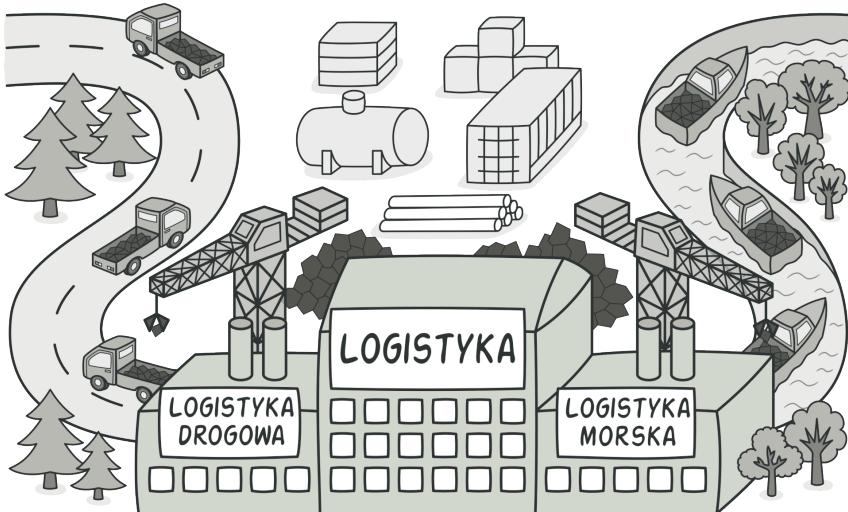
Umożliwia kopowanie istniejących obiektów bez tworzenia zależności pomiędzy twoim kodem, a ich klasami.



## Singleton

Singleton

Pozwala zachować pewność, że istnieje wyłącznie jedna instancja danej klasy oraz istnieje dostęp do niej w przestrzeni globalnej.



# METODA WYTÓRCZA

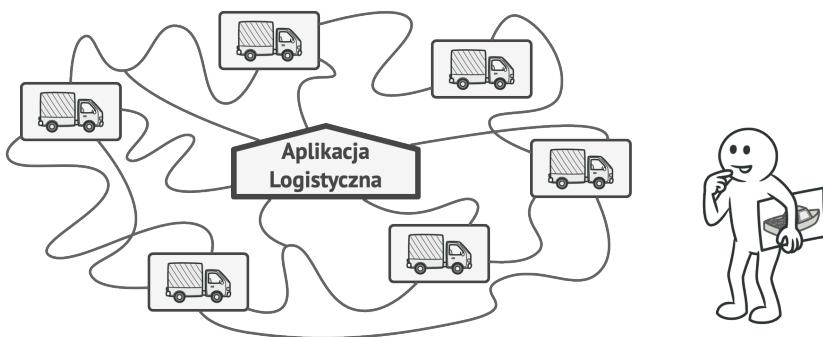
Znany też jako: *Konstruktor wirtualny, Virtual constructor, Factory Method*

**Metoda wytwórcza** jest kreacyjnym wzorcem projektowym, który udostępnia interfejs do tworzenia obiektów w ramach klasy bazowej, ale pozwala podklasom zmieniać typ tworzonych obiektów.

## Problem

Wyobraź sobie, że tworzysz aplikację do zarządzania logistyką. Pierwsza wersja twojej aplikacji pozwala jedynie na obsługę transportu za pośrednictwem ciężarówek, więc większość kodu znajduje się wewnątrz klasy `Ciążarówka`.

Po jakimś czasie Twoja aplikacja staje się całkiem popularna. Codziennie otrzymujesz tuzin prośb od firm realizujących spedycję morską, abyś dodał stosowną funkcjonalność do swojej aplikacji.



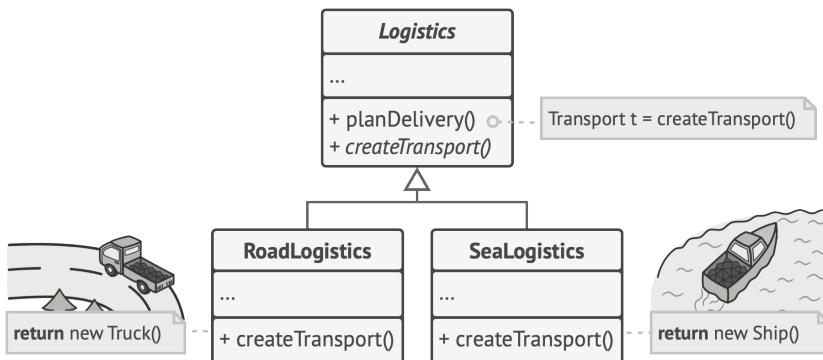
*Dodanie nowej klasy do programu nie jest takie proste, jeśli reszta kodu jest już związana z istniejącymi klasami.*

Świetna wiadomość, prawda? Ale co z kodem? W tej chwili większość Twojego kodu jest powiązana z klasą `Ciążarówka`. Dodanie do aplikacji klasy `Statki` wymagałoby dokonania zmian w całym kodzie. Co więcej, jeśli później zdecydujesz się dodać kolejny rodzaj transportu, zapewne będziesz musiał dokonać tych zmian jeszcze jeden raz.

Rezultatem powyższych działań będzie brzydki kod, pełen instrukcji warunkowych, których zadaniem będzie dostosowanie zachowania aplikacji zależnie od klasy transportu.

## 😊 Rozwiążanie

Wzorzec projektowy Metody wytwórczej proponuje zamianę bezpośrednich wywołań konstruktora obiektów (wykorzystujących operator `new`) na wywołania specjalnej metody *wytwórczej*. Jednak nie przejmuj się tym: obiekty nadal powstają za pośrednictwem operatora `new`, ale teraz dokonuje się to za kulisami – z wnętrza metody wytwórczej. Obiekty zwarcane przez metodę wytwórczą często są nazywane *produktami*.

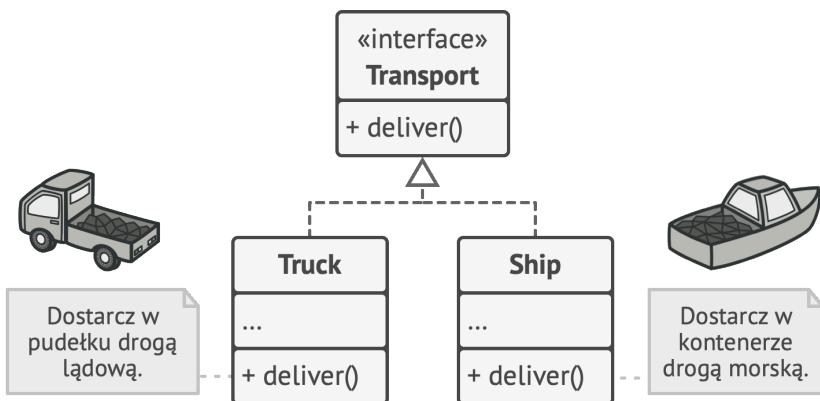


*Podklasy mogą zmieniać klasę obiektów zwracanych przez metodę wytwórczą.*

Na pierwszy rzut oka zmiana ta może wydawać się bezcelowa. Przecież przenieśliśmy jedynie wywołanie konstruktora z jednej części programu do drugiej. Ale zwróć uwagę, że teraz mo-

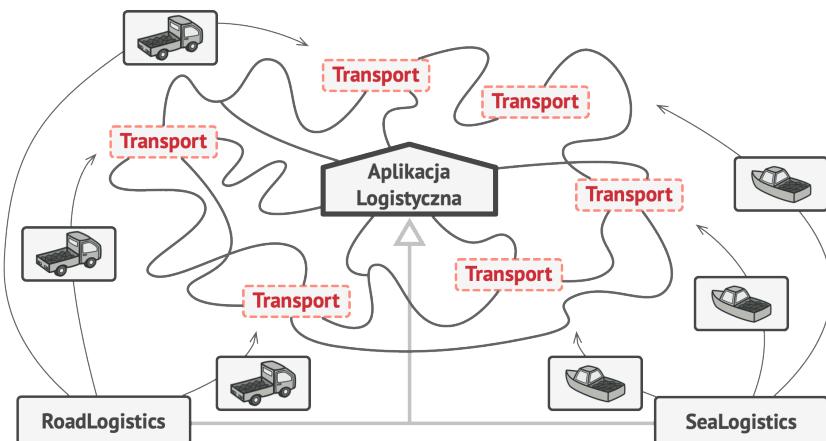
żesz nadpisać metodę wytwórczą w podklasie, a tym samym zmienić klasę produktów zwracanych przez metodę.

Istnieje jednak małe ograniczenie: podklasy mogą zwracać różne typy produktów tylko wtedy, gdy produkty te mają wspólną klasę bazową lub wspólny interfejs. Ponadto, zwracany typ metody wytwórczej w klasie bazowej powinien być zgodny z tym interfejsem.



*Wszystkie produkty muszą być zgodne z tym samym interfejsem.*

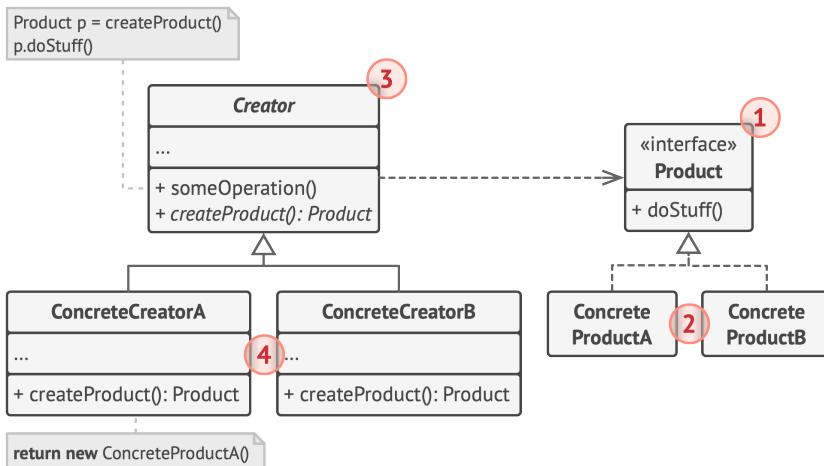
Na przykład zarówno klasy `Ciążarówka`, jak i `Statek` powinny implementować interfejs `Transport`, który z kolei deklaruje metodę `dostarczaj`. Każda klasa różnie implementuje tę metodę: ciężarówki dostarczają towar drogą lądową, statki drogą morską. Metoda wytwórcza znajdująca się w klasie `LogistykaDrogowa` zwraca obiekty `Ciążarówka`, zaś metoda wytwórcza w klasie `LogistykaMorska` zwraca `Statki`.



O ile wszystkie klasy produktów implementują wspólny interfejs, możesz przekazywać ich obiekty do kodu klienckiego bez obawy o jego zepsucie.

Kod, który wykorzystuje metodę wytwarzającą (zwany często kodem *klienckim*) nie widzi różnic pomiędzy faktycznymi produktami zwróconymi przez różne podklasy. Klient traktuje wszystkie produkty jako abstrakcyjnie pojęty `Transport`. Klient wie także, że wszystkie obiekty transportowe posiadają metodę `dostarczaj`, ale szczegóły jej działania nie są dla niego istotne.

# Struktura



1. **Produkt** deklaruje interfejs, który jest wspólny dla wszystkich obiektów zwracanych przez twórcę oraz jego podklasy.
2. **Konkrete Produkty** są różnymi implementacjami interfejsu produktów.
3. Klasa **Twórca** deklaruje metodę wytwórczą, która zwraca nowe obiekty-produkty. Istotne jest, że typ zwracany przez tę metodę jest zgodny z interfejsem produktu.

Możesz zadeklarować metodę wytwórczą jako abstrakcyjną. Wówczas każda podklasa będzie musiała zaimplementować swoją jej wersję. Innym sposobem jest sprawienie, aby bazowa metoda wytwórcza zwracała jakiś domyślny typ produktu.

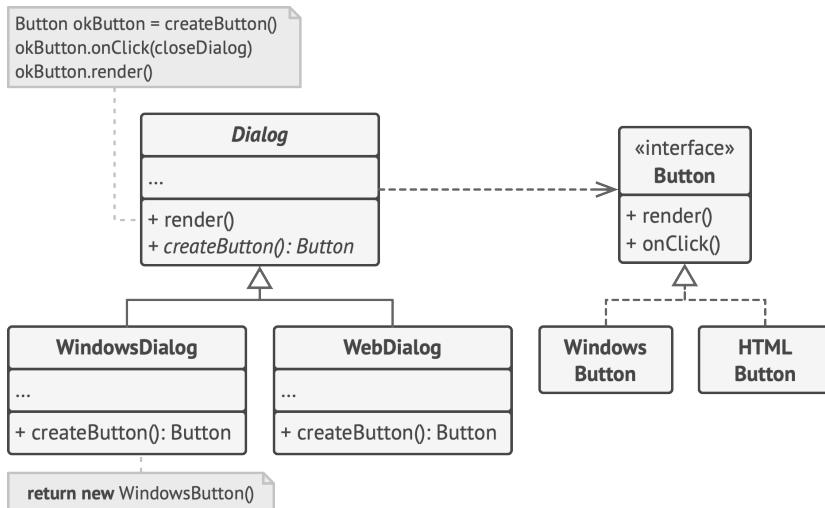
Weź jednak pod uwagę, że wbrew swojej nazwie, tworzenie produktów **nie** jest główną odpowiedzialnością Klasy Twórcy. Zazwyczaj klasa kreacyjna zawiera już jakąś ważną logikę biznesową związaną z produktami. Metoda twórcza pomaga rozprzegnąć tę logikę i konkretne klasy produktów. Oto analogia: duża firma tworząca oprogramowanie może posiadać swój dział szkoleniowy dla programistów. Ale głównym zadaniem firmy jako całości jest nadal tworzenie kodu, a nie tworzenie programistów.

4. **Konkretni Twórcy** nadpisują bazową metodę twórczą, co sprawia, że zwraca ona inny typ obiektu.

Tu jednak uwaga: Metoda twórcza nie musi wciąż **tworzyć** nowych instancji. Może też zwrócić istniejący już obiekt z pamięci podręcznej, puli obiektów lub z innego źródła.

## # Pseudokod

Przykład ten ilustruje, jak **Metoda Twórcza** może służyć tworzeniu elementów interfejsu użytkownika (UI) które będą przenośne między platformami. Nie występuje tu spręgnięcie kodu klienckiego z konkretnymi klasami interfejsu użytkownika.



*Przykład międzyplatformowego okna dialogowego.*

Bazowa klasa okna dialogowego wykorzystuje różne elementy interfejsu użytkownika rysując na ekranie okno. W różnych systemach operacyjnych, elementy te mogą różnić się nieco wyglądem, ale powinny zachowywać się w sposób spójny. Przycisk w Windows powinien być wciąż przyciskiem również w Linux.

Wraz z pojawieniem się metody wytwórczej, znika konieczność przepisywania logiki okna dialogowego dla poszczególnych systemów operacyjnych. Jeśli zadeklarujemy metodę wytwórczą, która tworzy przyciski w ramach klasy bazowej okna dialogowego, możemy później stworzyć dodatkową podkласę okna dialogowego, która będzie zwracała przyciski w stylu Windows z poziomu metody wytwórczej. Podklasa odziedziczy wówczas większość kodu okna z klasy bazowej, ale dzięki

metodzie wytwórczej, będzie renderowała przyciski w stylu Windows.

Aby ten wzorzec zadziałał, klasa bazowa okna dialogowego musi działać korzystając z przycisków abstrakcyjnych klasy bazowej lub interfejsu, zgodnie z którym powstaną wszystkie konkretne przyciski. Dzięki temu, kod okna dialogowego pozostanie funkcjonalny niezależnie od tego, jaki będzie rodzaj przycisku.

Oczywiście możesz zastosować to podejście również w stosunku do innych elementów interfejsu użytkownika. Jednak wraz z dodaniem do okna dialogowego każdej kolejnej metody wytwórczej, zblizasz się do wzorca projektowego zwanego **Fabryka abstrakcyjna**. Ale nie obawiaj się, później zajmiemy się również tym problemem.

```
1 // Klasa kreacyjna deklaruje metodę wytwórczą która musi zwracać
2 // obiekt klasy produktu. Poszczególne podklasy kreatora na ogół
3 // implementują tę metodę.
4 class Dialog is
5     // Kreator może również posiadać jakąś domyślną
6     // implementację metody wytwórczej.
7     abstract method createButton():Button
8
9     // Zwróć uwagę, że pomimo swojej nazwy, głównym zadaniem
10    // kreatora nie jest tworzenie produktów. Zamiast tego
11    // zawiera jakąś kluczową logikę biznesową która jest
12    // zależna od obiektów-produktów zwróconych przez metodę
```

```
13 // wytwórczą. Podklasy mogą pośrednio zmieniać tę logikę
14 // biznesową poprzez nadpisywanie metody wytwórczej i
15 // zwracanie innych typów produktów.
16 method render() is
17     // Wywołanie metody wytwórczej w celu stworzenia
18     // obiektu-produktu.
19     Button okButton = createButton()
20     // A następnie użycie produktu.
21     okButton.onClick(closeDialog)
22     okButton.render()
23
24 // Konkretni kreatorzy nadpisują metodę wytwórczą w celu zmiany
25 // zwracanego typu produktu.
26 class WindowsDialog extends Dialog is
27     method createButton():Button is
28         return new WindowsButton()
29
30 class WebDialog extends Dialog is
31     method createButton():Button is
32         return new HTMLButton()
33
34 // Interfejs produktu deklaruje wszystkie działania które
35 // konkretne produkty muszą zaimplementować.
36 interface Button is
37     method render()
38     method onClick(f)
39
40 // Konkretne produkty posiadają różne implementacje interfejsu
41 // produktu.
42 class WindowsButton implements Button is
43     method render(a, b) is
44         // Renderuj przycisk w stylu Windows.
```

```
45  method onClick(f) is
46      // Powiąż z wbudowanym w system operacyjny zdarzeniem
47      // kliknięcia
48
49  class HTMLButton implements Button is
50      method render(a, b) is
51          // Zwróć wersję HTML przycisku.
52      method onClick(f) is
53          // Powiąż z wbudowanym w przeglądarkę zdarzeniem
54          // kliknięcia
55
56
57  class Application is
58      field dialog: Dialog
59
60      // Aplikacja wybiera typ kreatora na podstawie bieżącej
61      // konfiguracji lub zmiennych środowiskowych.
62      method initialize() is
63          config = readApplicationConfigFile()
64
65          if (config.OS == "Windows") then
66              dialog = new WindowsDialog()
67          else if (config.OS == "Web") then
68              dialog = new WebDialog()
69          else
70              throw new Exception("Error! Unknown operating system.")
71
72      // Kod kliencki współpracuje z instancją konkretnego twórcy
73      // za pośrednictwem interfejsu bazowego. Tak długo jak
74      // klient będzie współpracował z kreatorem za pośrednictwem
75      // interfejsu bazowego, można będzie mu przekazywać dowolną
76      // podkласę twórcy.
```

```
77 method main() is
78     this.initialize()
79     dialog.render()
```

## 💡 Zastosowanie

- ⚡ **Stosuj Metodę Wytwórczą gdy nie wiesz z góry jakie typy obiektów pojawią się w twoim programie i jakie będą między nimi zależności.**
- ⚡ Metoda Wytwórcza oddziela kod konstruujący produkty od kodu który faktycznie z tych produktów korzysta. Dlatego też łatwiej jest rozszerzać kod konstruujący produkty bez konieczności ingerencji w resztę kodu.

Przykładowo, aby dodać nowy typ produktu do aplikacji, będziesz musiał utworzyć jedynie podklassę kreacyjną i nadpisać jej metodę wytwórczą.
- ⚡ **Korzystaj z Metody Wytwórczej gdy zamierzasz pozwolić użytkującym twą bibliotekę lub framework rozbudowywać jej wewnętrzne komponenty.**
- ⚡ Dziedziczenie jest prawdopodobnie najłatwiejszym sposobem rozszerzania domyślnego zachowania się biblioteki lub frameworku. Ale skąd framework wiedziałby o konieczności zastosowania twojej podklasy, zamiast standardowego komponentu?

Rozwiązaniem jest zredukowanie kodu konstruującego komponenty na przestrzeni framework do pojedynczej metody wytwórczej. Trzeba też umożliwić nadpisywanie tej metody, a nie tylko rozbudowywanie samego komponentu.

Sprawdźmy, jak by to wyglądało. Wyobraź sobie, że piszesz aplikację korzystając z open source'owego frameworku interfejsu użytkownika (UI). Twój aplikacja ma posiadać okrągłe przyciski, ale framework oferuje jedynie prostokątne. Rozszerzasz więc standardową klasę `Przycisk` o podkласę `OkrągłyPrzycisk`. Ale teraz trzeba również sprawić, by główna klasa `UIFramework` korzystała z nowo utworzonej podklasy, zamiast z domyślnej.

Aby to osiągnąć, tworzysz podkласę `UIZ0KołaPrzyciskami` z bazowej klasy framework i nadpisujesz jej metodę `stwórzPrzycisk`. Metoda ta będzie zwracać obiekty klasy `Przycisk` w klasie bazowej, zaś twoja jej podklasa zwróci obiekty `OkrągłyPrzycisk`. Od teraz skorzystasz z klasy `UIZ0KołaPrzyciskami` zamiast klasy `UIFramework`. I to tyle!

- ⚡ **Korzystaj z Metody wytwórczej gdy chcesz oszczędniej wykorzystać zasoby systemowe poprzez ponowne wykorzystanie już istniejących obiektów, zamiast odbudowywać je raz za razem.**
- ⚡ Powyższa sytuacja może wyłonić się na pierwszy plan, gdy mamy do czynienia z dużymi obiektami, wymagającymi sporej

ilości zasobów. Mogą do nich należeć połączenia do bazy danych, systemy plików oraz zasoby sieciowe.

Zastanówmy się, co musi się stać, aby istniejący obiekt mógł zostać wykorzystany ponownie:

1. Najpierw musisz stworzyć jakiś magazyn, który będzie pamiętał wszystkie utworzone obiekty.
2. Gdy ktoś zgłosi zapotrzebowanie na obiekt, program powinien odszukać wolny obiekt spośród tych już istniejących w puli.
3. ... i wreszcie zwrócić go kodowi klienckiemu.
4. Jeśli nie ma żadnych wolnych obiektów, program powinien utworzyć nowy (i dodać go do puli magazynowej).

To strasznie dużo kodu! I na dodatek cały musi się znaleźć w jednym miejscu, aby uniknąć rozrzucania jego kopii po całym programie.

Prawdopodobnie, najbardziej oczywistym i odpowiednim miejscem na umieszczenie tego nowego kodu jest konstruktor tej klasy, której obiekty chcemy ponownie wykorzystywać. Ale konstruktor musi z definicji zawsze zwracać **nowe obiekty**. Nie może zwracać istniejących jego instancji.

Dlatego też będzie potrzebna zwykła metoda, która zdolna jest zarówno tworzyć nowe obiekty, jak i pozwolić na wykorzystanie już istniejących. A to już brzmi jak metoda wytwórcza.

## Jak implementować

1. Wszystkie produkty powinny być zgodne z tym samym interfejsem. Interfejs ten powinien deklarować metody, które są sensowne dla każdego produktu.
2. Dodaj pustą metodę wytwórczą do klasy kreacyjnej. Zwracany typ tej metody powinien być zgodny z interfejsem wspólnym dla produktów.
3. W kodzie kreacyjnym należy odnaleźć wszystkie odniesienia do konstruktorów produktów. Jeden po drugim, trzeba zamienić je na wywołania metody wytwórczej, ekstrahując przy tym kod kreacyjny produktów, aby umieścić go w metodzie wytwórczej.

Możliwe, że konieczne okaże się ustanowienie tymczasowego parametru w metodzie wytwórczej, aby kontrolować jaki typ produktu będzie zwrócony.

Na tym etapie, kod metody wytwórczej może brzydko wyglądać. Może się na przykład okazać, że wewnątrz znajduje się wielki operator `switch` wybierający stosowną klasę produktu, jaką należy powołać do istnienia. Ale nie martw się, niedługo się tym zajmiemy.

4. Teraz stwórz zestaw podklas kreacyjnych dla każdego typu produktu wymienionego w metodzie wytwórczej. Nadpisz metodę wytwórczą w każdej z podklas i wyekstrahuj stosowne fragmenty kodu konstruującego z metody bazowej.

5. Jeśli mamy zbyt wiele typów produktów i nie ma sensu tworzyć podklasy dla każdego z nich, możesz wykorzystać ponownie parametr kontrolny klasy bazowej w podklasach.

Na przykład wyobraź sobie, że masz następującą hierarchię klas. Istnieje klasa bazowa `Poczta` z kilkoma podklasami: `PocztaLotnicza` oraz `PocztaLądowa`. Klasa `Transport` to: `Samolot`, `Ciężarówka` oraz `Pociąg`. Klasa `PocztaLotnicza` używa jedynie obiektów klasy `Samolot`, ale `PocztaLądowa` zarówno obiektów klasy `Ciężarówka`, jak i `Pociąg`. Można więc stworzyć kolejną podkласę (powiedzmy, że `PocztaKolejowa`) aby obsłużyć oba przypadki, ale istnieje lepszy sposób. Klientki może bowiem przekazać parametr metodzie wytwórczej znajdującej się w klasie `PocztaLądowa` który zdecyduje o typie produktów, jakie są potrzebne.

6. Jeśli po dokonaniu wszystkich ekstrakcji, bazowa metoda wytwórcza została pusta, możesz uczynić ją abstrakcyjną. Jeśli jednak coś w niej pozostało, możesz pozostawić to jako domyślne zachowanie się metody.

## ΔΔ Zalety i wady

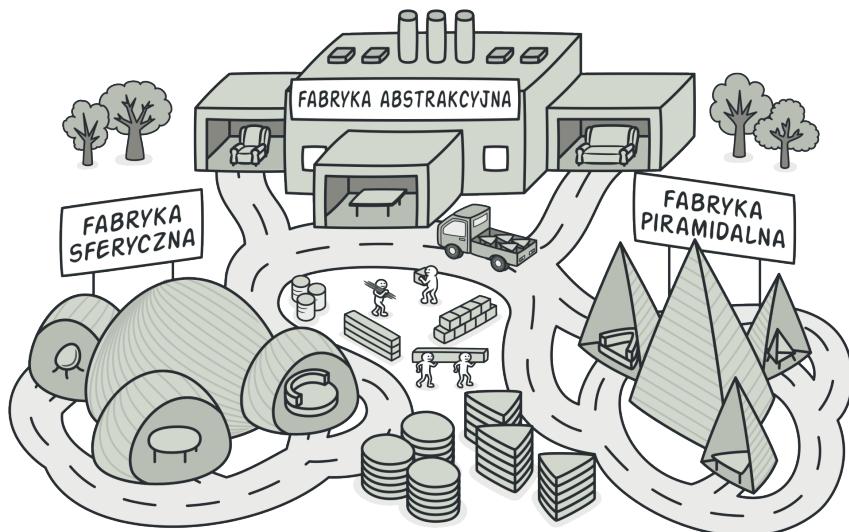
- ✓ Unikasz ścisłego sprzęgnięcia pomiędzy twórcą a konkretnymi produktami.
- ✓ *Zasada pojedynczej odpowiedzialności.* Możesz przenieść kod kreacyjny produktów w jedno miejsce programu, ułatwiając tym samym utrzymanie kodu.

- ✓ *Zasada otwarte/zamknięte.* Możesz wprowadzić do programu nowe typy produktów bez psucia istniejącego kodu klienckiego.
- ✗ Kod może się skomplikować, ponieważ aby zaimplementować wzorzec, musisz utworzyć liczne podklasy. W najlepszej sytuacji wprowadzisz ów wzorzec projektowy do już istniejącej hierarchii klas kreacyjnych.

## ↔ Powiązania z innymi wzorcami

- Wiele projektów zaczyna się od zastosowania **Metody wytwórczej** (mniej skomplikowanej i dającej się dostosować poprzez tworzenie podklas). Projekty następnie ewoluują stopniowo w **Fabrykę abstrakcyjną**, **Prototyp** lub **Budowniczego** (bardziej elastyczne, ale i bardziej skomplikowane wzorce).
- Klasa **Fabryka abstrakcyjna** często wywodzi się z zestawu **Metod wytwórczych**, ale można także użyć **Prototypu** do skomponowania metod w tych klasach.
- Możesz zastosować **Metodę wytwórczą** wraz z **Iteratorem** aby pozwolić podklasom kolekcji zwracać różne typy iteratorów kompatybilnych z kolekcją.
- **Prototyp** nie bazuje na dziedziczeniu, więc nie posiada właściwych temu podejścia wad. Z drugiej strony jednak *Prototyp* wymaga skomplikowanej inicjalizacji klonowanego obiektu. **Metoda wytwórcza** bazuje na dziedziczeniu, ale nie wymaga etapu inicjalizacji.

- **Metoda wytwórcza** to wyspecjalizowana **Metoda szablonowa**. Może stanowić także jeden z etapów większej *Metody szablonowej*.



# FABRYKA ABSTRAKCYJNA

Znany też jako: *Abstract Factory*

**Fabryka abstrakcyjna** jest kreacyjnym wzorcem projektowym, który pozwala tworzyć rodziny spokrewnionych ze sobą obiektów bez określania ich konkretnych klas.

## Problem

Wyobraź sobie, że tworzysz symulator sklepu meblowego. Twój kod składa się z klas, które reprezentują:

1. Rodzinę spokrewnionych produktów, powiedzmy: `Fotel` + `Sofa` + `StolikKawowy`.
2. Różne warianty w ramach powyższej rodziny. Na przykład, produkty `Fotel` + `Sofa` są dostępne w wariantach: `Nowoczesny`, `Wiktoriański`, `ArtDeco`.



*Rodziny produktów i ich warianty.*

Trzeba produkować poszczególne meble w taki sposób, aby do siebie pasowały. Klienci nie cierpią bowiem otrzymywać mebli w zupełnie różnych stylizacjach.



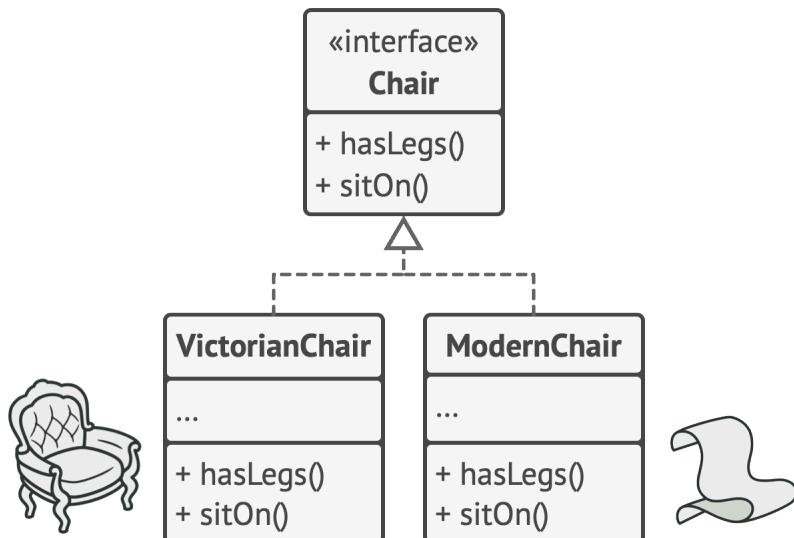
*Sofa zaprojektowana w stylu nowoczesnym nie pasuje do foteli wiktoriańskich.*

Ponadto, nie chciałbyś zmieniać istniejącego kodu tylko po to, aby dodać nowy produkt lub rodzinę produktów do programu. Producenci mebli dość często wypuszczają nowe katalogi i nie chciałbyś zmieniać głównej części kodu za każdym razem gdy tak się stanie.

## 😊 Rozwiążanie

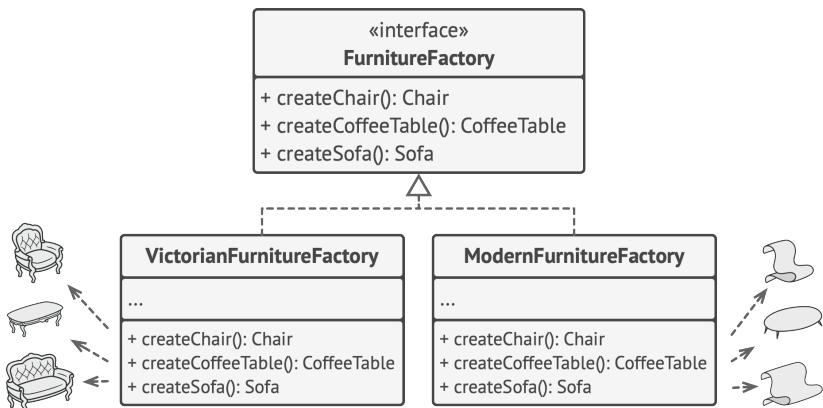
Pierwszą rzeczą, jaką proponuje wzorzec projektowy Fabryka abstrakcyjna, jest wyraźne określenie interfejsów dla każdego konkretnego produktu z jakiejś rodziny (np. fotel, sofa, stolik kawowy). Następnie trzeba sprawić, aby wszystkie warianty produktów były zgodne z tymi interfejsami. Na przykład

wszystkie fotele implementują interfejs `Fotel`, wszystkie stołki kawowe implementują interfejs `StolikKawowy` i tak dalej.



*Wszystkie warianty tego samego obiektu muszą się znaleźć w jednej hierarchii klasowej.*

Kolejnym krokiem jest deklaracja interfejsu *Fabryka abstrakcyjna*, który zatrzyma listę metod kreacyjnych wszystkich produktów w ramach jednej rodziny (na przykład, `stwórzFotel`, `stwórzSofę`, `stwórzStolikKawowy`). Metody te muszą zwracać wyłącznie **abstrakcyjne** typy produktów, reprezentowane uprzednio określonymi interfejsami: `Fotel`, `Sofa`, `StolikKawowy` i tak dalej.



Każda konkretna fabryka odpowiada konkretnemu wariantowi produktu.

A co z poszczególnymi wariantami produktów? Otóż, dla każdego wariantu rodziny produktów tworzymy osobną klasę fabryczną na podstawie interfejsu `FabrykaAbstrakcyjna`. Klasa fabryczna to taka klasa, która zwraca produkty danego rodzaju. A więc, `FabrykaNowoczesnychMebli` może zwracać wyłącznie obiekty: `NowoczesneFotele`, `NowoczesneSofy` oraz `NowoczesneStolikiKawowe`.

Kod kliencki będzie korzystał z fabryk oraz produktów za pośrednictwem ich interfejsów abstrakcyjnych. Dzięki temu będzie można zmienić typ fabryki przekazywanej kodowi klienckiemu oraz zmienić wariant produktu jaki otrzyma kod kliencki i to wszystko bez ryzyka popsucia samego kodu klienckiego.

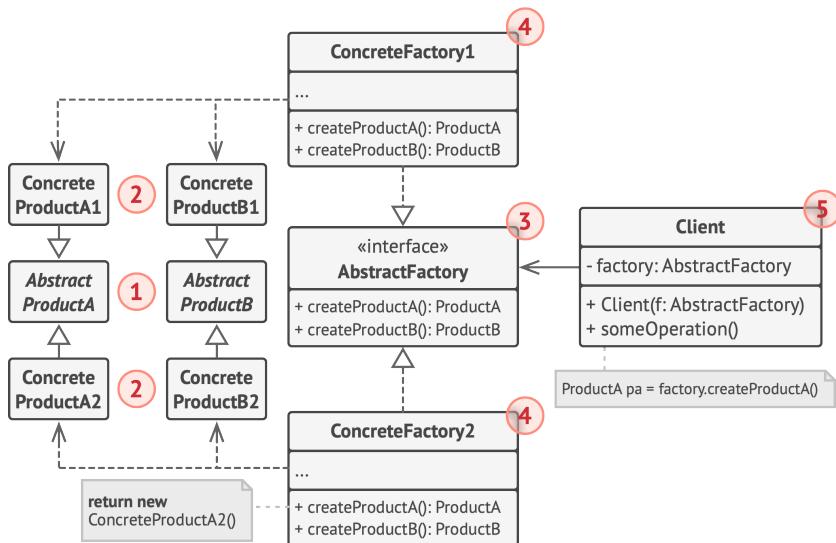


*Klienta nie powinno obchodzić to, z jaką konkretnie klasą fabryczną ma do czynienia.*

Załóżmy, że klient potrzebuje fabrykę do stworzenia fotela. Nie powinien musieć być świadom klasy tej fabryki, ani martwić się o rodzaj fotelu z jakim przyjdzie mu działać. Czy będzie to fotel nowoczesny, czy też wiktoriański, klient powinien traktować wszystkie w taki sam sposób, za pośrednictwem interfejsu abstrakcyjnego `Fotel`. Dzięki temu podejściu, klient wie tylko tyle, że fotele implementują jakąś formę metody `usiądźNa`. Ponadto, niezależnie od wariantu zwracanego fotelu, zawsze będą one pasowały do sof lub stolików kawowych jakie produkuje dany obiekt fabryczny.

Pozostaje do wyjaśnienia jeszcze jedna sprawa: jeśli klient ma do czynienia wyłącznie z interfejsami abstrakcyjnymi, to co właściwie tworzy rzeczywiste obiekty fabryczne? Na ogół aplikacja tworzy konkretny obiekt fabryczny na etapie inicjalizacji. Tuż przed tym wybiera stosowny typ fabryki zależnie od konfiguracji lub środowiska.

## Struktura

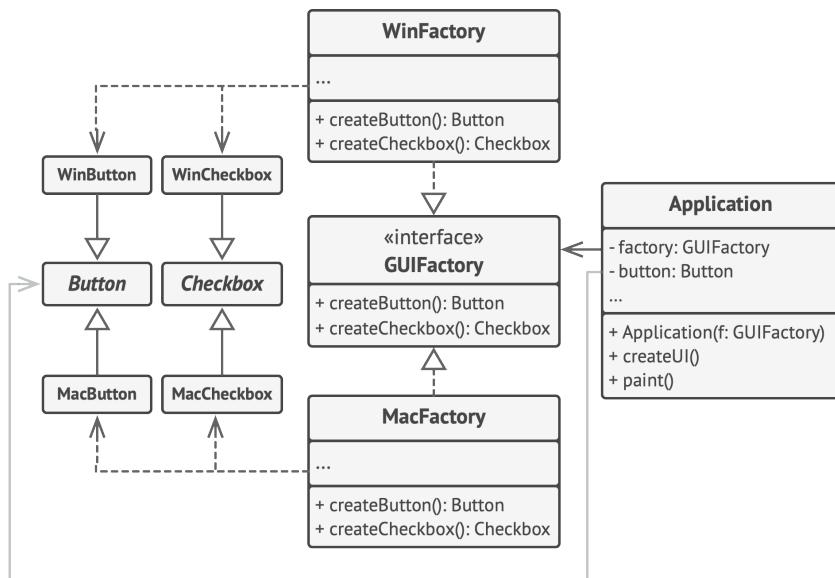


1. **Produkty Abstrakcyjne** deklarują interfejsy odmiennych produktów, które składają się na wspólną rodzinę.
2. **Konkretnie Produkty** to różnorakie implementacje abstrakcyjnych produktów, pogrupowane według wariantów. Każdy abstrakcyjny produkt (fotel/sofa) musi być zaimplementowany we wszystkich zadanych wariantach (Wiktoriański/Nowoczesny).
3. Interfejs **Fabryki Abstrakcyjnej** deklaruje zestaw metod służących tworzeniu każdego z abstrakcyjnych produktów.
4. **Konkrete Fabryki** implementują metody kreacyjne fabryki abstrakcyjnej. Każda konkretna fabryka jest związana z jakimś określonym wariantem produktu i produkuje wyłącznie meble w tym stylu.
5. Mimo, że konkretne fabryki tworzą konkretne egzemplarze produktu, sygnatury ich metod kreacyjnych muszą zwracać stosowne

*abstrakcyjne* produkty. Dzięki temu kod klienki, który korzysta z fabryki, nie zostanie sprzęgnięty z jakimś konkretnym wariantem produktu, jaki otrzymuje z fabryki. **Klient** może działać na dowolnym konkretnym wariantie fabryki/produkту, o ile będzie korzystał z interfejsów abstrakcyjnych ich obiektów.

## # Pseudokod

Poniższy przykład ilustruje zastosowanie **Fabryki abstrakcyjnej** do tworzenia międzyplatformowych elementów interfejsu użytkownika (UI), unikając tym samym sprzężenia kodu klienta z konkretnymi klasami interfejsu użytkownika oraz zachowując zgodność tworzonych elementów UI z danym systemem operacyjnym.



Przykład międzyplatformowych klas UI.

Elementy interfejsu użytkownika tego samego typu powinny zachowywać się podobnie niezależnie od platformy, ale mogą wyglądać nieco inaczej na różnych systemach operacyjnych. Co więcej, to twoim zadaniem jest zapewnić zgodność wizualnego stylu elementów UI ze stylem konkretnego systemu operacyjnego. Nie chcemy przecież wyświetlać kontrolek w stylu macOS w programie uruchamianym pod Windows.

Interfejs fabryki abstrakcyjnej deklaruje pewien zestaw metod kreacyjnych, dzięki którym kod kliencki może tworzyć różne typy elementów interfejsu użytkownika. Konkretne fabryki odpowiadają poszczególnym systemom operacyjnym i tworzą zgodne z nimi wizualnie elementy UI.

Działa to tak: kiedy aplikacja jest uruchamiana, sprawdza pod jakim pracuje systemem operacyjnym. Mając tę wiedzę, aplikacja tworzy obiekt fabryczny z klasy odpowiadającej danemu systemowi. Reszta kodu z kolei korzysta z tej fabryki przy tworzeniu elementów interfejsu użytkownika. Dzięki temu zapobiega się tworzeniu niewłaściwych kontrolek.

Dzięki takiemu podejściu, kod kliencki nie jest zależny od konkretnych klas fabryk oraz elementów UI, pod warunkiem, że będzie korzystał z ich interfejsów abstrakcyjnych. Pozwoli to także kodowi klienckiemu zachować zgodność z fabrykami lub elementami UI mogącymi pojawić się w przyszłości.

Wynikiem takiego projektowania, uniknąć można zmian w kodzie klienckim w razie dodania obsługi nowych stylów wizu-

alnych kontrolek. Wystarczy stworzyć nową klasę fabryczną, która wytwarzać będzie te elementy oraz nieco zmodyfikować kod inicjalizujący aplikacji, aby mógł obrać tę klasę.

```
1 // Interfejs fabryki abstrakcyjnej deklaruje pewien zestaw metod
2 // które zwracają różne produkty abstrakcyjne. Produkty te
3 // łącznie nazywa się rodziną i łączy je jakiś wysokopoziomowy
4 // wspólny motyw lub koncepcja. Produkty z jednej rodziny są
5 // zwykle zdolne do współpracy z sobą nawzajem. Rodzina
6 // produktów może mieć wiele wariantów, ale produkty jednego
7 // wariantu są niekompatybilne z produktami z rodziny innego
8 // wariantu.
9 interface GUIFactory is
10    method Button createButton()
11    method Checkbox createCheckbox()
12
13 // Konkretne fabryki tworzą produkty należące do jednego
14 // wariantu jednej rodziny. Fabryka gwarantuje że powstałe
15 // produkty będą kompatybilne. Zwracane typy w sygnaturach metod
16 // wytwórczych konkretnej fabryki określone są jako produkty
17 // abstrakcyjne, zaś konkretna instancja produktu powstaje
18 // wewnątrz metody.
19 class WinFactory implements GUIFactory {
20    method Button createButton() {
21        return new WinButton()
22    }
23    method Checkbox createCheckbox() {
24        return new WinCheckbox()
25    }
26
27 // Każda konkretna fabryka ma swój wariant produktu.
28 class MacFactory implements GUIFactory {
29    method Button createButton() {
```

```
28     return new MacButton()
29     method createCheckbox():Checkbox is
30         return new MacCheckbox()
31
32 // Każdy odrębny produkt z rodziny powinien mieć interfejs
33 // bazowy. Wszystkie warianty produktu muszą zaimplementować ten
34 // interfejs.
35 interface Button is
36     method paint()
37
38 // Konkretne fabryki mają swoje konkretne produkty.
39 class WinButton implements Button is
40     method paint() is
41         // Renderuj przycisk w stylu Windows.
42
43 class MacButton implements Button is
44     method paint() is
45         // Renderuj przycisk w stylu macOS.
46
47 // Oto interfejs bazowy innego produktu. Wszystkie produkty mogą
48 // ze sobą współpracować, ale właściwa interakcja możliwa jest
49 // wyłącznie pomiędzy produktami jednego konkretnego wariantu.
50 interface Checkbox is
51     method paint()
52
53 class WinCheckbox implements Checkbox is
54     method paint() is
55         // Renderuj pole wyboru w stylu Windows.
56
57 class MacCheckbox implements Checkbox is
58     method paint() is
59         // Renderuj pole wyboru w stylu macOS.
```

```
60
61 // Kod kliencki współpracuje z fabrykami i produktami wyłącznie
62 // stosując typy abstrakcyjne: GUIFactory, Button i Checkbox.
63 // Pozwala to przekazywać klientowi dowolną podklasę produktu
64 // lub fabryki.
65 class Application is
66     private field factory: GUIFactory
67     private field button: Button
68     constructor Application(factory: GUIFactory) is
69         this.factory = factory
70     method createUI() is
71         this.button = factory.createButton()
72     method paint() is
73         button.paint()
74
75 // Aplikacja wybiera typ fabryki na podstawie bieżącej
76 // konfiguracji lub zmiennych środowiskowych i tworzy jej
77 // instancję w trakcie działania programu (zazwyczaj na etapie
78 // inicjalizacji).
79 class ApplicationConfigurator is
80     method main() is
81         config = readApplicationConfigFile()
82
83         if (config.OS == "Windows") then
84             factory = new WinFactory()
85         else if (config.OS == "Mac") then
86             factory = new MacFactory()
87         else
88             throw new Exception("Error! Unknown operating system.")
89
90         Application app = new Application(factory)
```

## Zastosowanie

-  **Stosuj Fabrykę abstrakcyjną, gdy twój kod ma działać na produktach z różnych rodzin, ale jednocześnie nie chcesz, aby ścisłe zależało od konkretnych klas produktów. Mogą one bowiem być nieznane na wcześniejszym etapie tworzenia programu, albo chcesz umożliwić przyszłą rozszerzalność aplikacji.**
  -  Fabryka abstrakcyjna dostarcza ci interfejs służący tworzeniu obiektów z różnych klas danej rodziny produktów. O ile twój kod będzie kreował obiekty za pośrednictwem tego interfejsu – nie musisz się martwić stworzeniem produktu w niezgodnym z innymi wariancie.
- 
- Przemyśl ewentualną implementację wzorca Fabryki abstrakcyjnej, gdy masz do czynienia z klasą posiadającą zestaw **Metod wytwórczych** które zbytnio przyjmiewają główną odpowiedzialność tej klasy.
  - W prawidłowo zaprojektowanym programie *każda klasa jest odpowiedzialna za jedną rzecz*. Gdy zaś klasa ma do czynienia z wieloma typami produktów, warto być może zebrać jej metody wytwórcze i umieścić je w osobnej klasie fabrycznej, albo nawet w pełni zaimplementować Fabrykę abstrakcyjną z ich pomocą.

## Jak zaimplementować

1. Stwórz mapę poszczególnych typów produktów z uwzględnieniem wariantów w jakich mogą one być dostępne.
2. Dla każdego typu produktu zaimplementuj abstrakcyjny interfejs. Niech wszystkie konkretne klasy produktów implementują powyższe interfejsy.
3. Zadeklaruj interfejs fabryki abstrakcyjnej zawierający zestaw metod kreacyjnych wszystkich produktów abstrakcyjnych.
4. Zaimplementuj zestaw konkretnych klas fabrycznych – po jednym dla każdego wariantu produktu.
5. Gdzieś w programie umieść kod inicjalizujący fabrykę. Kod ten powinien powołać do życia obiekt jednej z konkretnych klas fabrycznych – zależnie od konfiguracji programu, czy też środowiska, w jakim został uruchomiony. Przekaż następnie ten obiekt fabryczny każdej klasie, której zadaniem jest konstrukcja produktów.
6. Przejrzyj kod aplikacji, wyszukując wszelkie bezpośrednie wywołania konstruktorów produktów. Zamień te wywołania na takie, które odnoszą się do stosownych metod kreacyjnych obiektu fabrycznego.

## Δ Δ Zalety i wady

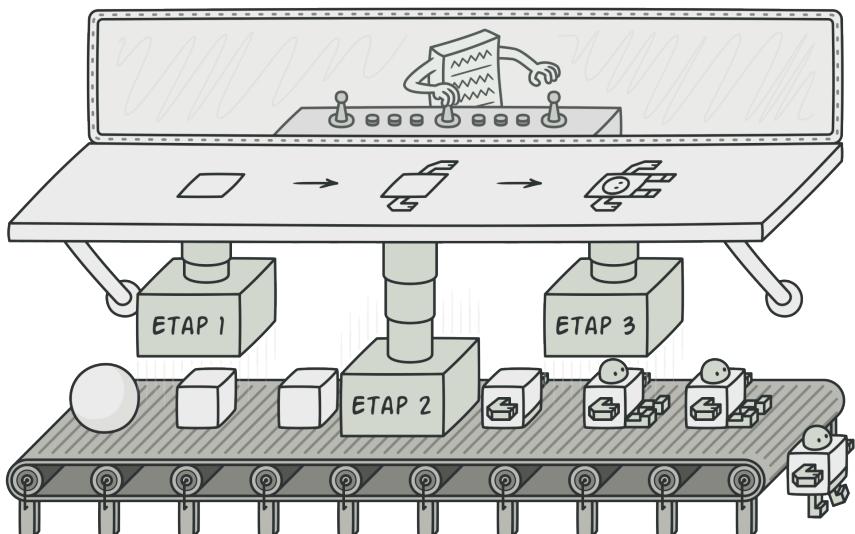
- ✓ Zyskujesz pewność, że produkty, jakie otrzymujesz stosując fabrykę, są ze sobą kompatybilne.
- ✓ Zapobiegasz ścisłemu sprzęgnięciu konkretnych produktów z kodem klienckim.
- ✓ *Zasada pojedynczej odpowiedzialności.* Możesz zebrać kod kreacyjny produktów w jednym miejscu w programie, ułatwiając tym samym późniejsze utrzymanie kodu.
- ✓ *Zasada otwarte/zamknięte.* Możesz wprowadzać wsparcie dla nowych wariantów produktów bez psucia istniejącego kodu klienckiego.
- ✗ Kod może stać się bardziej skomplikowany, niż powinien. Wynika to z konieczności wprowadzenia wielu nowych interfejsów i klas w toku wdrażania tego wzorca projektowego.

## ↔ Powiązania z innymi wzorcami

- Wiele projektów zaczyna się od zastosowania **Metody wytwarzającej** (mniej skomplikowanej i dającej się dostosować poprzez tworzenie podklas). Projekty następnie ewoluują stopniowo w **Fabrykę abstrakcyjną**, **Prototyp** lub **Budowniczego** (bardziej elastyczne, ale i bardziej skomplikowane wzorce).
- **Budowniczy** koncentruje się na konstruowaniu złożonych obiektów krok po kroku. **Fabryka abstrakcyjna** specjalizuje się w tworzeniu rodzin spokrewnionych ze sobą obiektów. Fa-

*bryka abstrakcyjna* zwraca produkt natychmiast, zaś *Budowniczy* pozwala dołączyć dodatkowe etapy konstrukcji zanim będzie można pobrać finalny produkt.

- Klasy **Fabryka abstrakcyjna** często wywodzą się z zestawu **Metod wytwórczych**, ale można także użyć **Prototypu** do skomponowania metod w tych klasach.
- **Fabryka abstrakcyjna** może służyć jako alternatywa do **Fasady** gdy jedyne co chcesz zrobić, to ukrycie przed kodem klienckim procesu tworzenia obiektów podsystemu.
- **Fabryka abstrakcyjna** może być stosowana wraz z **Mostem**. Takie sparowanie jest użyteczne gdy niektóre abstrakcje zdefiniowane przez *Most* mogą współdziałać wyłącznie z określonymi implementacjami. W tym przypadku, *Fabryka abstrakcyjna* może hermetyzować te relacje i ukryć zawiłości przed kodem klienckim.
- **Fabryki abstrakcyjne, Budowniczych** oraz **Prototypy** można zaimplementować jako **Singletony**.



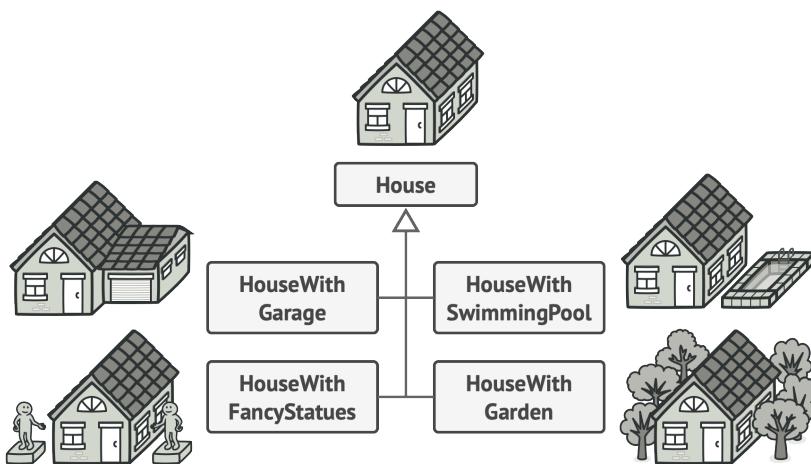
# BUDOWNICZY

Znany też jako: *Builder*

**Budowniczy** jest kreacyjnym wzorcem projektowym, który daje możliwość tworzenia złożonych obiektów etapami, krok po kroku. Wzorzec ten pozwala produkować różne typy oraz reprezentacje obiektu używając tego samego kodu konstrukcyjnego.

## (:() Problem

Wyobraź sobie jakiś skomplikowany obiekt, którego inicjalizacja jest pracochłonnym, wieloetapowym procesem obejmującym wiele pól i obiektów zagnieżdżonych. Taki kod inicjalizacyjny jest często wrzucany do wielgachnego konstruktora, przyjmującego mnóstwo parametrów. Albo jeszcze gorzej: kod taki rozrzucono po całym kodzie klienckim.

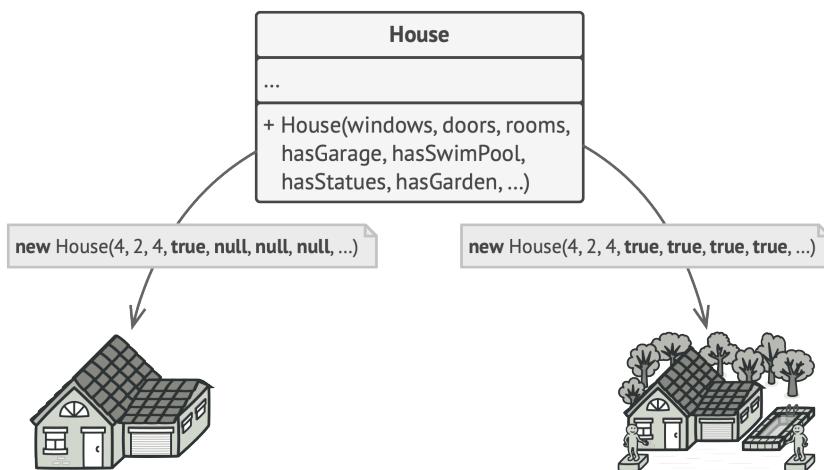


*Program może stać się nadmiernie skomplikowany, jeśli każda możliwa konfiguracja oznacza dodanie nowej podklasy.*

Na przykład pomyślmy, jak stworzyć obiekt Dom. Do zbudowania najprostszego domu wystarczą cztery ściany i podłoga. Do tego drzwi, parę okien i dach. Ale co, jeśli chcesz większy, jaśniejszy dom z podwórkiem i innymi dodatkami (ogrzewanie, kanalizacja, elektryczność)?

Najprostsze rozwiązanie to rozszerzenie klasy bazowej Dom i stworzenie zestawu podklas, które spełniałyby każdy możliwy zestaw wymogów. Ale takie podejście doprowadzi do wielkiej liczby podklas. Dodanie kolejnego parametru, jak styl werandy, jeszcze bardziej rozbuduje tę hierarchię.

Istnieje jednak inne rozwiązanie, które nie wiąże się z mnożeniem podklas. Można stworzyć jeden wielki konstruktor w klasie bazowej Dom, uwzględniający wszystkie możliwe parametry, które sterują obiektem typu dom. W ten sposób nie mnożymy liczby klas, ale tworzymy nieco inny problem.



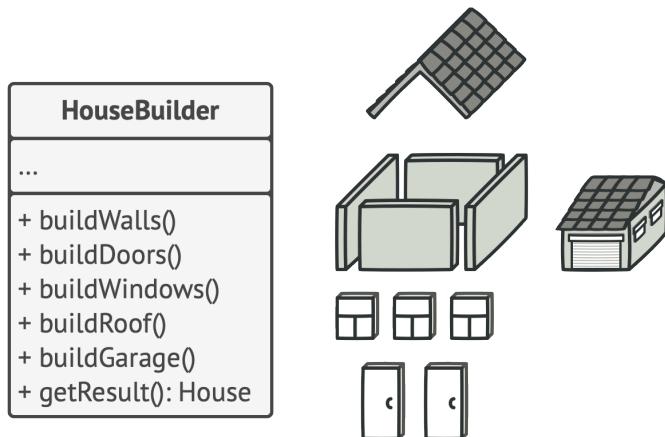
*Konstruktor przyjmujący mnóstwo parametrów ma swoją wadę: nie wszystkie parametry będą potrzebne za każdym razem.*

W większości przypadków parametry pozostaną nieużyte, a **wywołania konstruktora będą wyglądać niechlujnie**. Na przykład tylko niektóre domy mają basen, więc parametry do-

tyczące basenu w dziewięciu na dziesięć przypadków będą niepotrzebne.

## 😊 Rozwiążanie

Wzorzec projektowy Budowniczy proponuje ekstrakcję kodu konstrukcyjnego obiektu z jego klasy i umieszczenie go w osobnych obiektach zwanych *budowniczymi*.



*Wzorzec Budowniczy pozwala konstruować złożone obiekty krok po kroku. Budowniczy ponadto nie pozwala na dostęp do nich innym obiektom, dopóki nie zostaną ukończone.*

Ten wzorzec projektowy dzieli konstrukcję obiektu na pewne etapy (`budujŚciany`, `wstawDrzwi`, itd.). Aby powołać do życia obiekt, wykonuje się ciąg takich etapów za pośrednictwem obiektu-budowniczego. Istotne jest to, że nie musisz wywoływać wszystkich etapów. Możesz bowiem ograniczyć się

tylko do tych kroków, które są niezbędne do określenia potrzebnej nam konfiguracji obiektu.

Niektóre etapy konstrukcji mogą wymagać odmiennych implementacji, zależnie od potrzebnej w danej chwili reprezentacji produktu. Na przykład, ściany leśnej chatki mogą być drewniane, ale mury zamku warownego – kamienne.

W takim przypadku, można utworzyć wiele różnych klas budowniczych które implementują te same etapy konstrukcji, ale w różny sposób. Można następnie korzystać z tych budowniczych podczas procesu konstrukcji (np. odpowiednia kolejność wywołań etapów budowy) aby wytworzyć różne rodzaje obiektów.



*Różni budowniczowie wykonują to samo zadanie w różny sposób.*

Przykładowo, wyobraź sobie budowniczego, który konstruuje wyłącznie z drewna i szkła, drugi zaś stosuje tylko kamień i żelazo, a trzeci – złoto i diamenty. Wywołując te same etapy,

uzyskasz zwykły dom autorstwa pierwszego budowniczego, drugi z nich zbuduje mały zamek, a trzeci – pałac. Jednakże, to zadziała tylko pod warunkiem, że kod kliencki, który wywołuje etapy budowy, jest w stanie komunikować się z budowniczymi za pośrednictwem wspólnego interfejsu.

## Kierownik

Można pójść jeszcze dalej i przenieść kolejkę bezpośrednich wywołań budowniczego do osobnej klasy, zwanej *kierownikiem*. Kierownik określa kolejność etapów jaką musi zachować budowniczy, który z kolei implementuje te etapy konstrukcji obiektu.



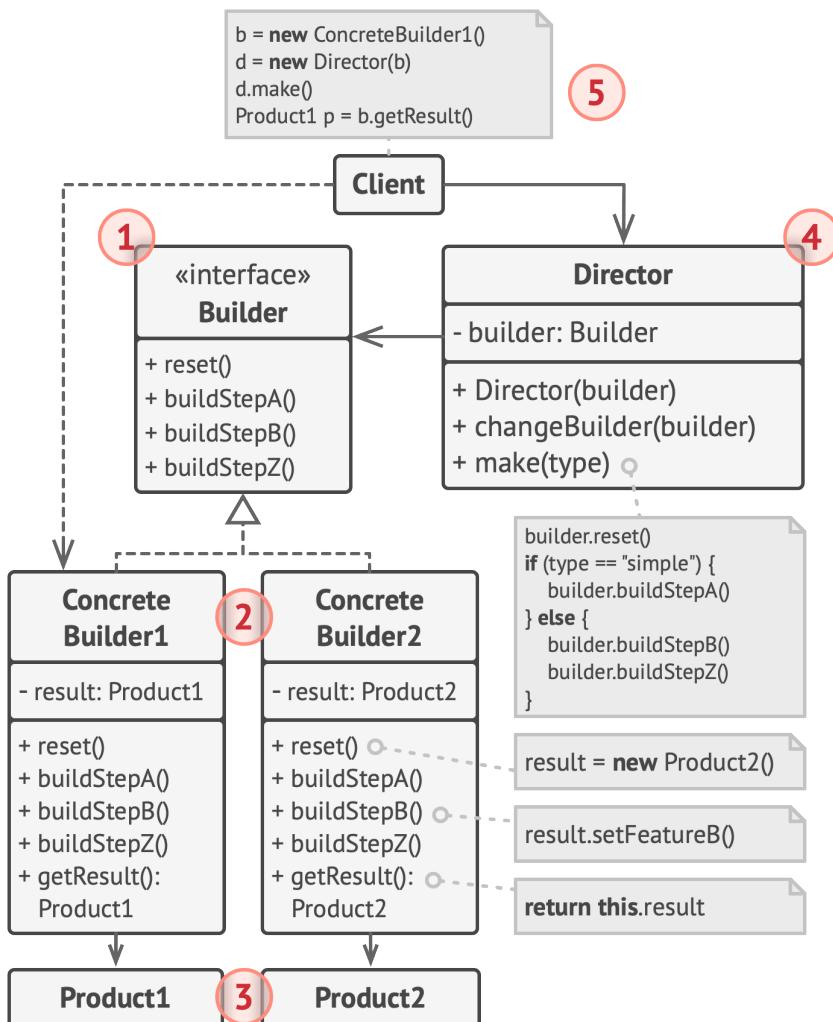
*Kierownik wie jakie kroki należy wykonać, aby otrzymać działający produkt.*

Posiadanie w programie klasy kierownika nie jest niezbędne. Można bowiem zawsze wywoływać etapy konstrukcji w odpowiedniej kolejności z poziomu kodu klienckiego. Jednakże,

kierownik może okazać się dobrym miejscem na umieszczenie czynności konstrukcyjnych, potrzebnych w innych miejscach programu.

Dodatkowo, klasa kierownik ukrywa szczegóły konstrukcji produktu przed kodem klienckim. Klient musi tylko skojarzyć budowniczego z kierownikiem, wywołać proces budowy za pośrednictwem tego pierwszego, a następnie odebrać wynik pracy od drugiego.

## Struktura

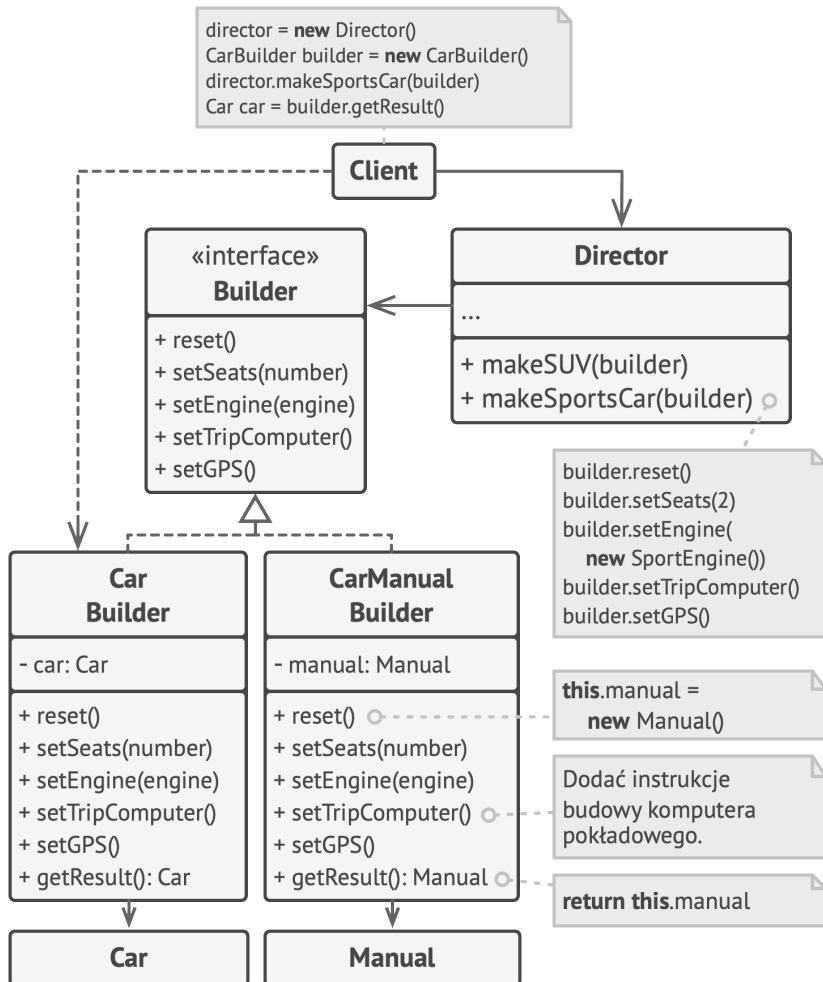


1. Interfejs **Budowniczego** deklaruje etapy konstrukcji produktu wspólnie dla wszystkich typów budowniczych.

2. **Konkretni Budowniczowie** zapewniają różne implementacje etapów konstrukcji. Konkretni budowniczowie mogą tworzyć produkty które nie mają wspólnego interfejsu.
3. **Produkty** to powstałe obiekty. Produkty konstruowane przez różnych budowniczych nie muszą należeć do tej samej hierarchii klas, czy interfejsu.
4. Klasa **Kierownik** definiuje kolejność w jakiej należy wywołać etapy konstrukcyjne, aby móc stworzyć i następnie użyć ponownie określone konfiguracje produktów.
5. **Klient** musi dopasować jeden z obiektów budowniczych do kierownika. Zazwyczaj robi się to tylko raz, za pośrednictwem parametru przekazywanego do konstruktora kierownika. Następnie kierownik za pomocą obiektu budowniczego wykonuje dalszą konstrukcję. Jednakże istnieje alternatywne podejście w przypadku przekazania obiektu budowniczego metodzie produkcyjnej kierownika. W takim przypadku kierownik może skorzystać z różnych budowniczych.

## # Pseudokod

Poniższy przykład użycia wzorca projektowego **Budowniczy** pokazuje, jak można ponownie wykorzystać ten sam kod konstrukcyjny obiektu budując różne typy produktów, takie jak samochody oraz odpowiednie do nich instrukcje obsługi.



Przykład kilkuetapowej konstrukcji samochodów oraz instrukcji obsługi ich modeli.

Samochód jest skomplikowanym obiektem, który można skonstruować na setki sposobów. Zamiast obciążać klasę Samochód olbrzymim konstruktorem, wyekstrahowaliśmy kod montażu auta do osobnej klasy budowniczego samochodu. Klasa ta

ma zestaw metod pozwalających skonfigurować dowolną część auta.

Jeśli kod kliencki musi utworzyć specjalny model samochodu na zamówienie, może skorzystać bezpośrednio z budowniczego. Z drugiej strony, klient może oddelegować montaż klasie kierownika, która wie jak za pomocą budowniczego skonstruować wiele najpopularniejszych modeli aut.

Może cię to zaskoczyć, ale do każdego auta powinna istnieć instrukcja obsługi (poważnie? ktoś je czyta?). Instrukcje opisują cechy i wyposażenie samochodów, więc ich zawartość będzie się różnić pomiędzy modelami. Dlatego warto ponownie wykorzystywać istniejący proces konstrukcyjny zarówno dla aut, jak i dla ich instrukcji. Oczywiście tworzenie instrukcji to nie to samo, co montaż auta i dlatego musimy dodać kolejną klasę budowniczego specjalizującą się w tworzeniu instrukcji. Klasa ta implementuje takie same metody budowy, co jej montująca auta krewna, ale zamiast montować – opisuje. Poprzez przekazanie tych budowniczych do tego samego obiektu kierownika, konstruujemy albo pojazd, albo instrukcję obsługi.

Ostatni etap to pobranie nowo utworzonego obiektu. Metalowy samochód i papierowa instrukcja obsługi, to jednak bardzo różne rzeczy, choć ze sobą związane. Nie możemy umieścić metody pobierającej wynik pracy w kierowniku, zanim nie powiążemy go z konkretną klasą produktów. Dlatego też odbieramy wynik u budowniczego, który jest jego autorem.

```
1 // Stosowanie wzorca Budowniczy ma sens tylko gdy produkty w
2 // programie są złożone i wymagają kompleksowej konfiguracji.
3 // Poniższe dwa produkty są powiązane, ale nie mają wspólnego
4 // interfejsu.
5 class Car is
6     // Samochód może mieć GPS, komputer pokładowy i pewną liczbę
7     // siedzeń. Różne modele samochodów (sportowe, SUV-y,
8     // kabriolety) mogą mieć różne opcjonalne funkcjonalności.
9
10 class Manual is
11     // Do każdego samochodu powinna istnieć instrukcja obsługi,
12     // która opisuje jego konfigurację i funkcje.
13
14
15 // Interfejs budowniczy określa metody służące tworzeniu
16 // poszczególnych części obiektów-produktów.
17 interface Builder is
18     method reset()
19     method setSeats(...)
20     method setEngine(...)
21     method setTripComputer(...)
22     method setGPS(...)
23
24 // Konkretne klasy budownicze są zgodne pod względem interfejsu
25 // i zawierają szczegółowe implementacje etapów budowania. Twój
26 // program może mieć wiele różnie zaimplementowanych wariacji
27 // budowniczych.
28 class CarBuilder implements Builder is
29     private field car:Car
30
31     // Nowo utworzona instancja budowniczego powinna zawierać
32     // pusty obiekt-produkt, który będzie podstawą do dalszej
```

```
33 // budowy.  
34 constructor CarBuilder() is  
35     this.reset()  
36  
37 // Metoda resetująca zeruje budowany obiekt.  
38 method reset() is  
39     this.car = new Car()  
40  
41 // Wszystkie etapy produkcji działają na tej samej instancji  
42 // produktu.  
43 method setSeats(...) is  
44     // Ustaw ilość siedzeń w aucie.  
45  
46 method setEngine(...) is  
47     // Zamontuj dany silnik.  
48  
49 method setTripComputer(...) is  
50     // Zamontuj komputer pokładowy.  
51  
52 method setGPS(...) is  
53     // Zamontuj nawigację GPS.  
54  
55 // Konkretni budowniczy powinni posiadać własne metody  
56 // pozyskiwania wyników działań, gdyż różni budowniczowie  
57 // tworzą różne produkty i nie zawsze zgodne pod względem  
58 // interfejsu. Dlatego też nie można zadeklarować takich  
59 // metod w interfejsie budowniczego (a przynajmniej nie  
60 // można tego dokonać w statycznie typowanym języku  
61 // programowania).  
62 //  
63 // Zazwyczaj po zwróceniu wyniku działania klientowi,  
64 // instancja budowniczego powinna być gotowa na rozpoczęcie
```

```
65 // produkcji od nowa. Dlatego typową praktyką jest wywołanie
66 // metody resetującej na końcu kodu metody `getProduct`. Nie
67 // jest to jednak obowiązkowe i można odroczyć zerowanie do
68 // momentu gdy klient prześle stosowne polecenie.
69 method getProduct():Car is
70     product = this.car
71     this.reset()
72     return product
73
74 // W przeciwieństwie do innych wzorców kreacyjnych, budowniczy
75 // pozwala konstruować produkty które nie mają wspólnego
76 // interfejsu.
77 class CarManualBuilder implements Builder is
78     private field manual:Manual
79
80     constructor CarManualBuilder() is
81         this.reset()
82
83     method reset() is
84         this.manual = new Manual()
85
86     method setSeats(...) is
87         // Udokumentuj specyfikację siedzeń.
88
89     method setEngine(...) is
90         // Dodaj instrukcję silnika.
91
92     method setTripComputer(...) is
93         // Dodaj instrukcję komputera pokładowego.
94
95     method setGPS(...) is
96         // Dodaj instrukcję nawigacji GPS.
```

```
97
98     method getProduct():Manual is
99         // Zwróć instrukcję obsługi i zresetuj budowniczego.
100
101
102    // Kierownik jest odpowiedzialny tylko za wywoływanie etapów
103    // budowy w odpowiedniej kolejności. Przydaje się to gdy
104    // tworzymy produkty według określonego porządku lub
105    // konfiguracji. Dopełczając – klasa Kierownik jest
106    // opcjonalna, ponieważ klient może kontrolować budowniczych
107    // bezpośrednio.
108    class Director is
109        // Kierownik może współdziałać z dowolną instancją
110        // budowniczego jaką klient mu przekaże. Dzięki temu kod
111        // klienta może zmienić ostateczny typ nowo utworzonego
112        // produktu. Kierownik może skonstruować wiele wariacji
113        // danego produktu postępując według tych samych etapów
114        // budowy.
115    method constructSportsCar(builder: Builder) is
116        builder.reset()
117        builder.setSeats(2)
118        builder.setEngine(new SportEngine())
119        builder.setTripComputer(true)
120        builder.setGPS(true)
121
122    method constructSUV(builder: Builder) is
123        ...
124
125
126    // Kod kliencki tworzy obiekt budowniczego, przekazuje go
127    // kierownikowi a następnie inicjuje proces konstrukcji.
128    // Ostateczny wynik działania pobiera się od obiektu
```

```
129 // budowniczego.  
130 class Application is  
131  
132 method makeCar() is  
133     director = new Director()  
134  
135     CarBuilder builder = new CarBuilder()  
136     director.constructSportsCar(builder)  
137     Car car = builder.getProduct()  
138  
139     CarManualBuilder builder = new CarManualBuilder()  
140     director.constructSportsCar(builder)  
141  
142     // Finalny produkt zwykle pobiera się od obiektu  
143     // budowniczego, ponieważ kierownik nic o nim nie wie i  
144     // nie jest zależny od konkretnych budowniczych czy  
145     // konkretnych produktów.  
146     Manual manual = builder.getProduct()
```

## 💡 Zastosowanie

💡 **Stosuj wzorzec Budowniczy, aby pozbyć się “teleskopowych konstruktorów”.**

⚡ Założmy, że masz konstruktor, przyjmujący 10 opcjonalnych parametrów. Wywołanie takiego potwora jest co najmniej nie-wygodne, dlatego przeciążamy konstruktor i tworzymy wiele jego krótszych wersji, wymagających mniej parametrów. Będą one wciąż odwoływać się do głównego konstruktora, prze-

kazując jakieś domyślne wartości w miejsce pominiętych argumentów.

```

1 class Pizza {
2     Pizza(int size) { ... }
3     Pizza(int size, boolean cheese) { ... }
4     Pizza(int size, boolean cheese, boolean pepperoni) { ... }
5     // ...

```

*Tworzenie takich potworów jest możliwe jedynie w językach obsługujących przeciążanie metod, a więc na przykład w C# oraz Javie.*

Wzorzec Budowniczy pozwala konstruować obiekty krok po kroku, w miarę jak staje się to w programie potrzebne. Po za-implementowaniu tego wzorca, nie musisz przekazywać konstruktorowi tuzina parametrów.

- ⚡ **Stosuj wzorzec Budowniczy, gdy potrzebujesz możliwości tworzenia różnych reprezentacji jakiegoś produktu (na przykład, domy z kamienia i domy z drewna).**
- ⚡ Wzorzec Budowniczy można użyć gdy konstruowanie różnorakich reprezentacji produktu obejmuje podobne etapy, które różnią się jedynie szczegółami.

Bazowy interfejs budowniczego definiuje wszelkie możliwe etapy konstrukcji, a konkretni budowniczy implementują te kroki by móc tworzyć poszczególne reprezentacje obiektów.

Natomiast klasa kierownik pilnuje właściwego porządku konstruowania.

 **Stosuj ten wzorzec do konstruowania drzew Kompozytowych lub innych złożonych obiektów.**

 Wzorzec budowniczego umożliwia konstrukcję w etapach. Niektóre z nich możemy odroczyć bez szkody dla finalnego produktu. Możemy nawet wywoływać etapy rekursywnie, co przydaje się przy budowie drzewa obiektów.

Budowniczy uniemożliwia dostęp do nieskończonego produktu przez okres jego konstrukcji. Zapobiega to pozyskiwaniu niekompletnych wyników przez kod kliencki.

## Jak zaimplementować

1. Upewnij się, że jesteś w stanie zdefiniować konkretne, wspólne etapy, wykonywane przy tworzeniu wszystkich dostępnych reprezentacji produktu. Bez tego nie uda się wdrożyć tego wzorca.
2. Zadeklaruj te etapy w interfejsie bazowego budowniczego.
3. Stwórz konkretną klasę budowniczego dla każdej reprezentacji produktu i zaimplementuj specyficzne dla nich etapy konstrukcyjne.

Nie zapomnij zaimplementować metodę pobierającą wynik konstrukcji. Powodem, dla którego taka metoda nie może być zadeklarowana w ramach interfejsu budowniczego jest to, że różni budowniczowie mogą tworzyć obiekty które nie posiadają wspólnego interfejsu. Dlatego też nie byłoby wiadomo jaki typ obiektu zwracałby taka metoda. Jednakże, jeśli masz do czynienia wyłącznie z produktami wchodząymi w skład jednej hierarchii, metodę taką można bezpiecznie dodać do interfejsu bazowego.

4. Rozważ stworzenie klasy kierownika. Może ona zawrzeć różne sposoby konstruowania produktu przy pomocy tego samego obiektu budowniczego.
5. Kod kliencki tworzy zarówno obiekty budowniczego, jak i kierownika. Przed rozpoczęciem konstrukcji, klient musi przekazać kierownikowi obiekt budowniczego. Zazwyczaj klient musi to zrobić jednorazowo, za pośrednictwem parametrów konstruktora kierownika. Kierownik potem wykonuje wszelkie prace konstrukcyjne za pomocą tego budowniczego. Jest też inny sposób, w którym budowniczy jest przekazywany bezpośrednio metodzie konstrukcyjnej kierownika.
6. Wynik konstrukcji może być odebrany bezpośrednio od kierownika tylko wtedy, gdy wszystkie produkty współdzielą taki sam interfejs. W przeciwnym razie, klient powinien pobrać wynik od budowniczego.

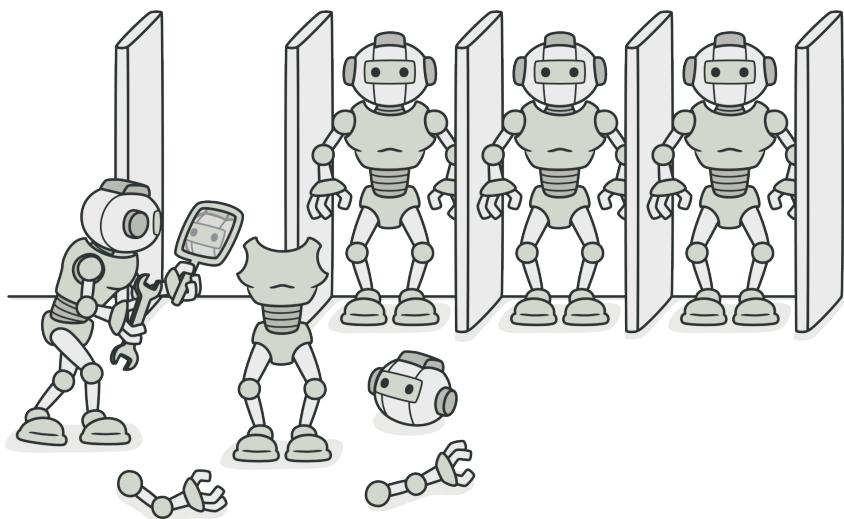
## Zalety i wady

- ✓ Możesz konstruować obiekty etapami, odkładać niektóre etapy, lub wykonywać je rekursywnie.
- ✓ Możesz wykorzystać ponownie ten sam kod konstrukcyjny budując kolejne reprezentacje produktów.
- ✓ *Zasada pojedynczej odpowiedzialności.* Można odizolować skomplikowany kod konstrukcyjny od logiki biznesowej produktu.
- ✗ Kod staje się bardziej skomplikowany, gdyż wdrożenie tego wzorca wiąże się z dodaniem wielu nowych klas.

## Powiązania z innymi wzorcami

- Wiele projektów zaczyna się od zastosowania **Metody wytwarzającej** (mniej skomplikowanej i dającej się dostosować poprzez tworzenie podklas). Projekty następnie ewoluują stopniowo w **Fabrykę abstrakcyjną**, **Prototyp** lub **Budowniczego** (bardziej elastyczne, ale i bardziej skomplikowane wzorce).
- **Budowniczy** koncentruje się na konstruowaniu złożonych obiektów krok po kroku. **Fabryka abstrakcyjna** specjalizuje się w tworzeniu rodzin spokrewnionych ze sobą obiektów. *Fabryka abstrakcyjna* zwraca produkt natychmiast, zaś *Budowniczy* pozwala dołączyć dodatkowe etapy konstrukcji zanim będzie można pobrać finalny produkt.

- Możesz zastosować wzorzec **Budowniczy** by tworzyć złożone drzewa **Kompozytowe** dzięki możliwości zaprogramowania ich etapów konstrukcji tak, aby odbywały się rekurencyjnie.
- Możliwe jest połączenie wzorców **Budowniczy** i **Most**: klasa *kierownik* pełni rolę abstrakcji, zaś poszczególni *budowniczy* stanowią *implementacje*.
- **Fabryki abstrakcyjne**, **Budowniczych** oraz **Prototypy** można zaimplementować jako **Singletony**.



# PROTOTYP

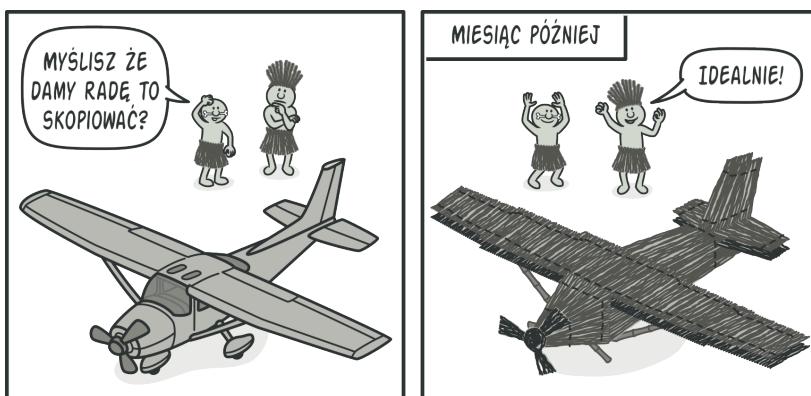
*Znany też jako: Klon, Clone, Prototype*

**Prototyp** to kreacyjny wzorzec projektowy, który umożliwia kopiowanie już istniejących obiektów bez tworzenia zależności pomiędzy twoim kodem, a klasami obiektów.

## (:() Problem

Przypuśćmy, że mamy jakiś obiekt i potrzebujemy jego dokładnej kopii. Jak można tego dokonać? Na początek, trzeba stworzyć nowy obiekt tej samej klasy. Potem zaś skopiować zawartość pól oryginału do pól kopii.

Fajnie! Ale jest i haczyk. Nie wszystkie obiekty można w ten sposób kopiować, bo nie wszystkie pola obiektu są ogólnodostępne – mogą być prywatne i tym samym niedostępne spoza samego obiektu.



*Kopiowanie obiektu oglądając go tylko z zewnątrz nie zawsze jest możliwe.*

Jest jeszcze jeden kłopot z takim bezpośrednim podejściem. Skoro musisz wiedzieć, jakiej klasy jest obiekt który chcesz skopiować, to twój kod staje się zależny od tej klasy. Jeśli to nie wygląda na kłopot, to weź pod uwagę, że czasem znamy tylko interfejs obiektu, a nie jego konkretną klasę. Przykładem

tego są metody przyjmujące w charakterze parametru obiekty zgodne z jakimś wspólnym interfejsem.

## Rozwiążanie

Wzorzec projektowy Prototyp deleguje proces klonowania samym obiektom, które mają być sklonowane. We wzorcu tym deklarowany jest wspólny interfejs dla wszystkich obiektów wspierających funkcjonalność klonowania. Interfejs taki pozwala klonować obiekty bez konieczności sprzęgania kodu z klasą obiektu. Zazwyczaj taki interfejs ogranicza się tylko do pojedynczej metody `klonuj`.

Implementacja metody `klonuj` jest bardzo podobna w poszczególnych klasach. Metoda tworzy obiekt swojej klasy i kopiuje doń wartości jej pól. Można wówczas skopiować także pola prywatne, gdyż większość języków programowania pozwala obiektom na dostęp do prywatnych pól obiektów tej samej klasy.

Obiekt, który posiada funkcjonalność klonowania zwany jest *prototypem*. Gdy obiekty z którymi masz do czynienia mają mnóstwo pól i setki możliwych konfiguracji, klonowanie ich może okazać się korzystną alternatywą do tworzenia podklaś.

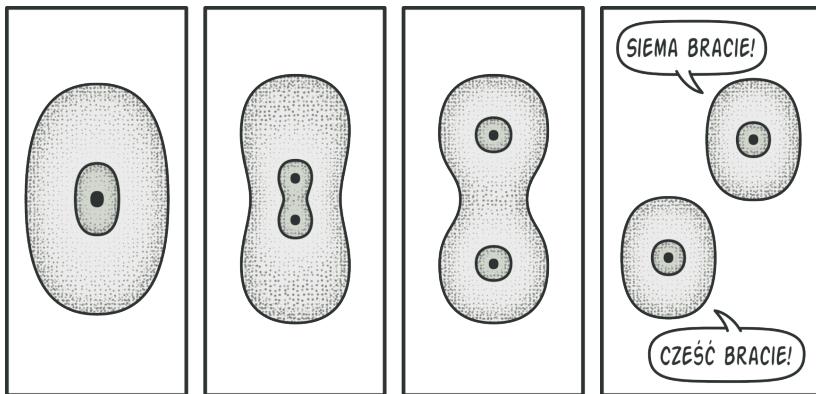


Prefabrykowane prototypy mogą być alternatywą do tworzenia podklas.

Działa to tak: tworzymy zestaw obiektów, każdy skonfigurowany na swój sposób. Jeśli potrzebujesz obiektu, który ma taką samą konfigurację jak już istniejący, klonujesz prototyp – zamiast konstruować nowy obiekt od podstaw.

## 🚗 Analogia do prawdziwego życia

W prawdziwym życiu prototypy stosuje się w testach, zanim rozpoczęta zostanie seryjna produkcja wyrobu. Tu jednak prototypy nie biorą udziału w faktycznej produkcji, lecz pełnią w niej rolę pasywną.

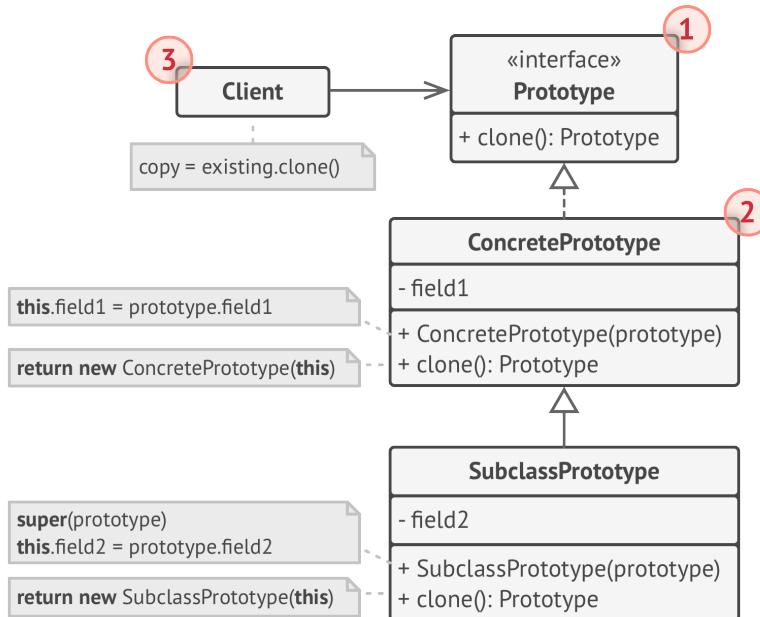


*Podział żywnej komórki.*

Ponieważ prototypy w przemyśle nie potrafią kopiować siebie, bliższą analogią do prawdziwego życia będzie mitotyczny podział żywej komórki (biologia, pamiętasz?). Po podziale mitotycznym powstaje para identycznych komórek. Pierwotna komórka stanowi prototyp i jednocześnie pełni aktywną rolę w duplikacji.

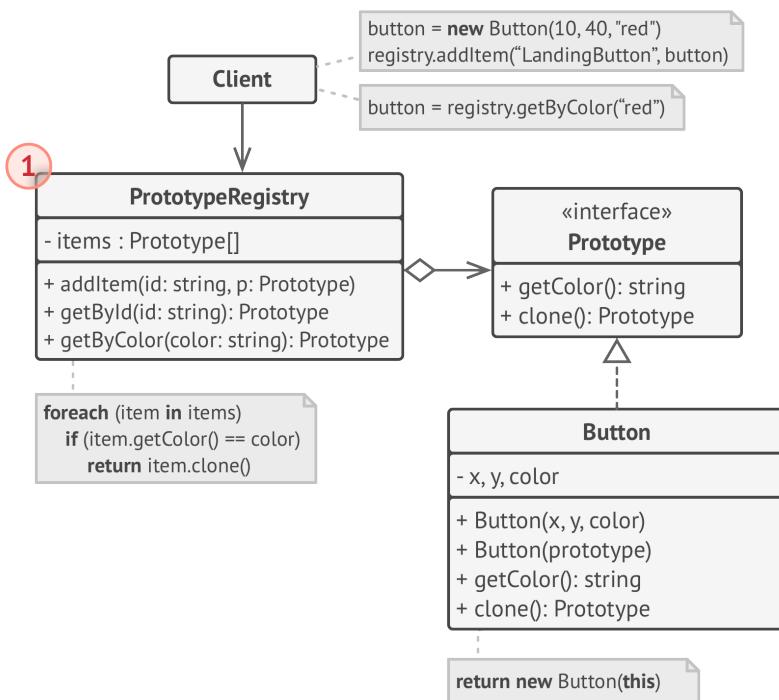
# Struktura

## Podstawowa implementacja



1. Interfejs **Prototyp** deklaruje metody klonujące. W większości przypadków jest to pojedyncza metoda `klonuj`.
2. Klasa **Konkretny Prototyp** implementuje metodę klonującą. Oprócz kopiowania danych pierwotnego obiektu do nowo powstającego, metoda ta może również rozwiązywać kwestie związane z klonowaniem obiektów powiązanych, czy też z zależnościami rekursywnymi.
3. **Klient** może stworzyć kopię dowolnego obiektu który jest zgodny z interfejsem prototypu.

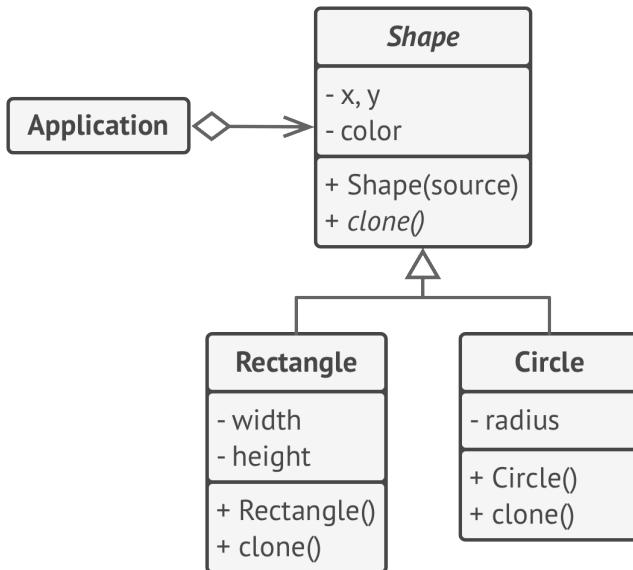
## Implementacja rejestru prototypów



1. **Rejestr prototypów** udostępnia łatwy dostęp do często używanych prototypów. Przechowuje zestaw prefabrykowanych obiektów, które są gotowe do skopiowania. Najprostszym rejestrzem prototypów będzie tablica asocjacyjna nazwa → prototyp. Jednakże, jeśli potrzebne są bardziej szczegółowe kryteria wyszukiwania, aniżeli tylko nazwa, można zbudować bardziej złożoną wersję rejestru.

## # Pseudokod

W tym przykładzie, wzorzec **Prototyp** pozwala tworzyć dokładne kopie figur geometrycznych bez sprzągania kodu z klasami figur.



*Klonowanie obiektów należących do jednej hierarchii klasowej.*

Wszystkie klasy figur są zgodne pod względem interfejsu, który z kolei posiada metodę klonującą. Podklasa może wywołać metodę klonującą klasy-rodzica przed skopiowaniem wartości swoich pól do obiektu wynikowego.

```

1 // Bazowy prototyp.
2 abstract class Shape is
3     field X: int

```

```
4   field Y: int
5   field color: string
6
7   // Zwyczajny konstruktor.
8   constructor Shape() is
9     // ...
10
11  // Konstruktor prototypu. Nowy obiekt jest inicjalizowany
12  // wartościami istniejącego obiektu.
13  constructor Shape(source: Shape) is
14    this()
15    this.X = source.X
16    this.Y = source.Y
17    this.color = source.color
18
19  // Klonowanie zwraca jedną z podklas Shape.
20  abstract method clone():Shape
21
22
23  // Konkretny prototyp. Metoda klonująca tworzy świeży obiekt i
24  // przekazuje go do konstruktora. Dopóki konstruktor nie skończy
25  // pracy, posiada jako jedyny odniesienie do świeżego klona.
26  // Gwarantuje to spójność klonowania.
27  class Rectangle extends Shape is
28    field width: int
29    field height: int
30
31  constructor Rectangle(source: Rectangle) is
32    // Trzeba wywołać konstruktor nadklasy w celu
33    // skopiowania pól zdefiniowanych w nadklasie.
34    super(source)
35    this.width = source.width
```

```
36     this.height = source.height
37
38     method clone():Shape is
39         return new Rectangle(this)
40
41
42     class Circle extends Shape is
43         field radius: int
44
45         constructor Circle(source: Circle) is
46             super(source)
47             this.radius = source.radius
48
49         method clone():Shape is
50             return new Circle(this)
51
52
53 // Gdzieś w kodzie klienckim.
54 class Application is
55     field shapes: array of Shape
56
57     constructor Application() is
58         Circle circle = new Circle()
59         circle.X = 10
60         circle.Y = 10
61         circle.radius = 20
62         shapes.add(circle)
63
64         Circle anotherCircle = circle.clone()
65         shapes.add(anotherCircle)
66         // Zmienna `anotherCircle` zawiera dokładną kopię
67         // obiektu `circle`.
```

```
68
69     Rectangle rectangle = new Rectangle()
70     rectangle.width = 10
71     rectangle.height = 20
72     shapes.add(rectangle)
73
74 method businessLogic() is
75     // Wzorzec Prototyp rządzi, bo pozwala stworzyć kopię
76     // obiektu bez wiedzy o jego typie.
77     Array shapesCopy = new Array of Shapes.
78
79     // Na przykład – nie musimy znać dokładnych elementów w
80     // tablicy figur (shapes). Wiemy tylko, że wszystkie są
81     // figurami. Ale dzięki polimorfizmowi, gdy wywołujemy
82     // metodę `klonuj` na figurze, program sprawdza jej
83     // klasę i uruchamia odpowiednią metodę klonującą
84     // zdefiniowaną w tejże klasie. Dlatego też otrzymujemy
85     // odpowiednie obiekty-klony, a nie ogólne obiekty
86     // Shape.
87     foreach (s in shapes) do
88         shapesCopy.add(s.clone())
89
90     // Tablica `shapesCopy` zawiera wierne kopie podklas z
91     // tablicy `shape`.
```

## 💡 Zastosowanie

💡 **Stosuj wzorzec Prototyp gdy chcesz aby twój kod nie był zależny od konkretnej klasy kopiowanego obiektu.**

- ⚡ Powyższa sytuacja zdarza się często, gdy twój kod pracuje na obiektach przekazanych z zewnętrznego źródła poprzez jakiś interfejs. Nieznana jest wówczas konkretna klasa takich obiektów.

Wzorzec Prototyp pozwala kodowi klienckiemu na pracę ze wszystkimi obiektami wspierającymi klonowanie za pomocą uogólnionego interfejsu. Interfejs czyni kod kliencki niezależnym od konkretnych klas klonowanych obiektów.

- ⚠ **Stosuj ten wzorzec gdy chcesz zredukować ilość podklas różniących się jedynie sposobem inicjalizacji swych obiektów. Ktoś inny bowiem mógł stworzyć takie podklasy tylko w celu tworzenia obiektów o określonej konfiguracji.**
- ⚡ Wzorzec Prototyp pozwala korzystać z zestawu prefabrykowanych obiektów w różnorakich konfiguracjach, stanowiących prototypy.

Zamiast tworzyć instancję podklasy zgodnej z jakąś konfiguracją, klient może po prostu wyszukać i sklonować odpowiedni prototyp.

## 📝 Jak zaimplementować

1. Stwórz interfejs prototypu i zadeklaruj w nim metodę `klonuj`. Albo po prostu dodaj tę metodę do wszystkich klas należących do istniejącej hierarchii, o ile taką masz.

2. Klasa prototypowa musi definiować alternatywny konstruktor, taki, który akceptuje jako argument obiekt swojej klasy. Taki konstruktor powinien skopiować cechy przekazanego obiektu do nowo utworzonej instancji. Jeśli modyfikujemy podklasę, musimy wywołać konstruktor klasy-rodzica, aby to nadklasa dokonała klonowania pól zdefiniowanych jako prywatne.

Jeśli używany przez ciebie język programowania nie wspiera przeciążania metod, możesz zdefiniować specjalną metodę służącą kopiowaniu danych obiektu. Konstruktor jest na nią dogodniejszym miejscem, bo zwraca nowo utworzony obiekt od razu po wywołaniu operatora `new`.

3. Metoda klonująca zazwyczaj składa się tylko z jednej linii kodu: wywołanie operatora `new` z prototypową wersją konstruktora. Pamiętać trzeba, że każda klasa musi wyraźnie nadpisać metodę klonującą, by stosowała nazwę własnej klasy wraz z operatorem `new`. W przeciwnym razie metoda klonująca może stworzyć obiekt klasy nadrzędnej.
4. Opcjonalnie, można stworzyć scentralizowany rejestr prototypów przechowujący katalog najczęściej używanych.

Możesz zaimplementować rejestr w formie nowej klasy fabrycznej, albo umieścić go w bazowej klasie prototypowej ze statyczną metodą służącą pobieraniu prototypu. Metoda ta powinna szukać prototypu na podstawie określonych przez klienta, przekazanych jej kryteriów. Kryteriami takimi może być albo prosty łańcuch tekstowy, albo też bardziej skomplikowany ze-

staw parametrów wyszukiwania. Po znalezieniu stosownego prototypu, rejestr powinien sklonować go i zwrócić kopię klientowi.

Na końcu należy zamienić bezpośrednie wywołania konstruktorów podklas na wywołania metody fabrycznej rejestru prototypów.

## ⚠ Zalety i wady

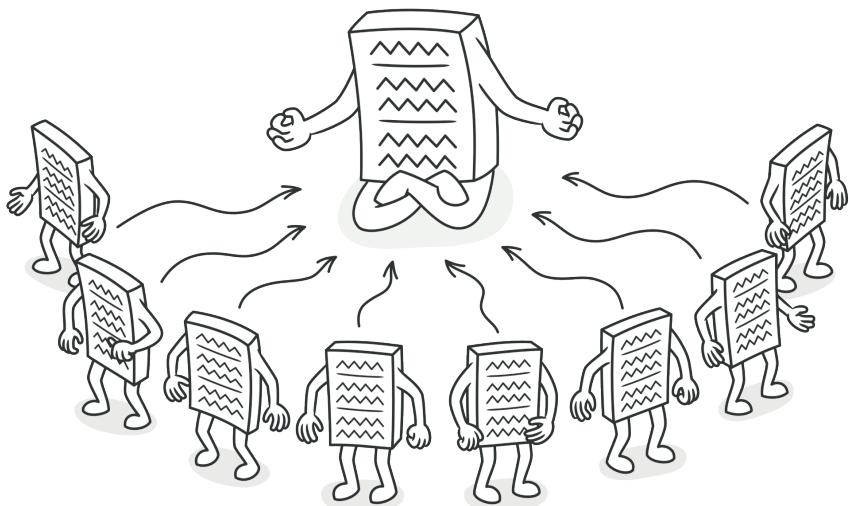
- ✓ Możesz klonować obiekty bez konieczności sprzęgania ze szczegółami ich konkretnych klas.
- ✓ Możesz pozbyć się wielokrotnie powtarzanego kodu inicjalizacyjnego na rzecz klonowania prefabrykowanych prototypów.
- ✓ Dużo wygodniejsze produkowanie złożonych obiektów.
- ✓ Podejście to stanowi alternatywę do dziedziczenia w przypadku gdy mamy do czynienia z wcześniej zdefiniowanymi konfiguracjami złożonych obiektów.
- ✗ Klonowanie złożonych obiektów, które mają odniesienia cykliczne, może być trudne.

## ➡ Powiązania z innymi wzorcami

- Wiele projektów zaczyna się od zastosowania **Metody wytwarzającej** (mniej skomplikowanej i dającej się dostosować poprzez tworzenie podklas). Projekty następnie ewoluują stopniowo w

**Fabrykę abstrakcyjną, Prototyp lub Budowniczego** (bardziej elastyczne, ale i bardziej skomplikowane wzorce).

- Klasa **Fabryka abstrakcyjna** często wywodzą się z zestawu **Metod wytwórczych**, ale można także użyć **Prototypu** do skomponowania metod w tych klasach.
- **Prototyp** może pomóc stworzyć historię, zapisując kopie **Poleceń**.
- Projekty intensywnie korzystające ze wzorców **Kompozyt** i **De-korator** mogą skorzystać również na zastosowaniu **Prototypu**. Zastosowanie tego wzorca pozwala klonować złożone struktury zamiast konstruować je ponownie od zera.
- **Prototyp** nie bazuje na dziedziczeniu, więc nie posiada właściwych temu podejścia wad. Z drugiej strony jednak *Prototyp* wymaga skomplikowanej inicjalizacji klonowanego obiektu. **Metoda wytwórcza** bazuje na dziedziczeniu, ale nie wymaga etapu inicjalizacji.
- Czasem **Prototyp** może być prostszą alternatywą dla **Pamiątki**. Jest to możliwe jeśli obiekt, którego stan chcesz przechować w historii, jest w miarę nieskomplikowany. Obiekt taki nie powinien też mieć powiązań z zewnętrznymi zasobami, albo muszą one być łatwe do ponownego nawiązania.
- **Fabryki abstrakcyjne, Budowniczych** oraz **Prototypy** można zaimplementować jako **Singletony**.



# SINGLETON

**Singleton** jest kreacyjnym wzorcem projektowym, który pozwala zapewnić istnienie wyłącznie jednej instancji danej klasy. Ponadto daje globalny punkt dostępowy do tejże instancji.

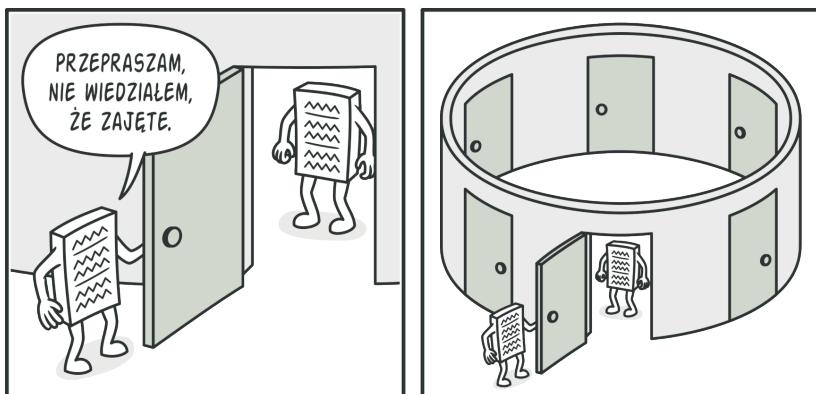
## (:() Problem

Wzorzec Singleton rozwiązuje jednocześnie dwa problemy, ale jednocześnie łamie *Zasadę pojedynczej odpowiedzialności*:

1. **Zapewnia istnienie wyłącznie jednej instancji danej klasy.** Dlaczego w ogóle ktokolwiek musiałby liczyć obiekty danej klasy? Otóż, najczęstszym powodem jest potrzeba kontroli dostępu do jakiegoś współdzielonego zasobu – na przykład bazy danych, lub pliku.

Działa to tak: wyobraź sobie, że masz już stworzony obiekt, ale po jakimś czasie potrzebujesz kolejnego. Zamiast otrzymać nowy, dostaniesz ten uprzednio stworzony.

Zwróć uwagę, że takiego zachowania nie da się zaimplementować stosując zwykły konstruktor, ponieważ metody te z definicji **muszą** zawsze zwracać nowe obiekty.



*Klienci mogą nawet nie zauważyc, że cały czas korzystają z tego samego obiektu.*

## 2. Pozwala na dostęp do tej instancji w przestrzeni globalnej.

Pamiętasz te globalne zmienne, z których korzystaliśmy (no dobra, ja korzystałem) do przechowywania istotnych obiektów? Chociaż są one bardzo poręczne, to wiążą się też z poważnym ryzykiem nadpisania ich zawartości i tym samym awarii programu.

Zupełnie, jak w przypadku zmiennej globalnej, wzorzec Singleton pozwala skorzystać z jakiegoś obiektu w dowolnym miejscu programu. Jednakże, zapewnia też ochronę tego obiektu przed działaniami innego kodu.

Jest też inna strona tego problemu: nie chcemy, aby kod, który pozwala nam rozwiązać pierwszy z powyższych problemów był porozrzucany po całym programie. Dużo lepiej jest trzymać go w jednej klasie, szczególnie, jeśli reszta kodu już od niego zależy.

Wzorzec Singleton stał się obecnie tak popularny, że czasem nazywa się Singletonem rozwiązania odnoszące się tylko do jednego z powyższych problemów.

## Rozwiązanie

Wszystkie implementacje wzorca Singleton współdzierają poniższe dwa etapy:

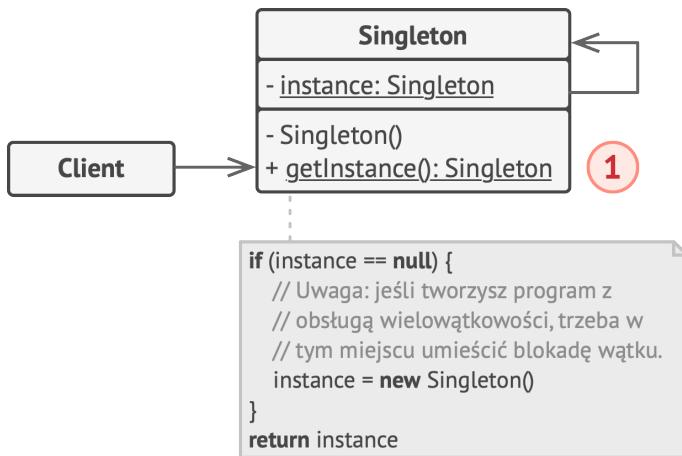
- Ograniczenie dostępu do domyślnego konstruktora przez uczy-nienie go prywatnym, aby zapobiec stosowaniu operatora `new` w stosunku do klasy Singleton.
- Utworzenie statycznej metody kreacyjnej, która będzie pełni-ła rolę konstruktora. Za kulisami, metoda ta wywoła prywatny konstruktor, aby utworzyć instancję obiektu i umieści go w polu statycznym klasy. Wszystkie kolejne wywołania tej meto-dy zwróci już istniejący obiekt.

Jeżeli twój kod ma dostęp do klasy Singleton, to będzie mógł wywołać jej statyczną metodę i tym samym za każdym razem otrzyma ten sam obiekt.

## Analogia do prawdziwego życia

Rząd jest doskonałym przykładem wzorca Singleton. Kraj może mieć wyłącznie jeden oficjalny rząd. Niezależnie od składu per-sonalnego członków rządu, pojęcie “Rząd kraju X” jest uniwer-salnym odwołaniem do organu władzy kraju.

## Struktura



1. Klasa **Singleton** deklaruje statyczną metodę `getInstance`, która zawsze zwraca tę samą instancję swojej klasy.

Konstruktor klasy Singleton musi być ukryty przed kodem klienckim. Tylko i wyłącznie wywołanie metody `getInstance` powinno dawać dostęp do takiego obiektu.

## # Pseudokod

W poniższym przykładzie, połączenie z bazą danych zrealizowane jest jako **Singleton**. Klasa ta nie ma publicznie dostępnego konstruktora, więc aby uzyskać dostęp do jej instancji, trzeba wywołać metodę `getInstance`. Metoda ta zachowuje pierwszy stworzony egzemplarz klasy i zwraca go przy każdym kolejnym wywołaniu.

```
1 // Klasa Database definiuje metodę `getInstance` która pozwala
2 // klientom na dostęp do tej samej instancji połączenia
3 // bazodanowego z dowolnego miejsca w programie.
4 class Database is
5     // Pole służące przechowywaniu instancji singleton powinno
6     // być zadeklarowane jako statyczne.
7     private static field instance: Database
8
9     // Konstruktor singletona powinien zawsze być zadeklarowany
10    // jako prywatny, aby uniemożliwić wywoływanie na klasie
11    // operatora `new`.
12    private constructor Database() is
13        // Jakiś kod inicjalizacyjny, na przykład faktyczne
14        // nawiązanie połączenia z serwerem bazy danych.
15        // ...
16
17    // Statyczna metoda kontrolująca dostęp do instancji
18    // singletona.
19    public static method getInstance() is
20        if (Database.instance == null) then
21            acquireThreadLock() and then
22                // Trzeba się upewnić, że instancja nie została
23                // już zainicjalizowana przez inny wątek w
24                // czasie gdy ten wątek oczekiwali na zwolnienie
25                // blokady.
26                if (Database.instance == null) then
27                    Database.instance = new Database()
28
29        return Database.instance
30
31    // Wreszcie – każdy singleton powinien definiować jakąś
32    // logikę biznesową którą można wywołać z instancji
33    // singletona.
```

```
33 public method query(sql) is
34     // Przykładowo, wszystkie zapytania do bazy danych w
35     // całej aplikacji muszą przejść przez tę metodę. Możesz
36     // tu więc umieścić kod pamięci podręcznej lub kontroli
37     // przepustowości.
38     // ...
39
40 class Application is
41     method main() is
42         Database foo = Database.getInstance()
43         foo.query("SELECT ...")
44         // ...
45         Database bar = Database.getInstance()
46         bar.query("SELECT ...")
47         // Zmienna `bar` będzie zawierała ten sam obiekt, co
48         // zmienna `foo`.
```

## 💡 Zastosowanie

- 💡 Korzystaj z wzorca Singleton, gdy w twoim programie ma prawo istnieć wyłącznie jeden ogólnodostępny obiekt danej klasy. Przykładem może być połączenie z bazą danych, którego używa wiele fragmentów programu.
- ⚡ Wzorzec projektowy Singleton uniemożliwia tworzenie obiektów danej klasy inaczej, niż przez stosowną metodę kreacyjną. Ta z kolei zwróci albo nowy obiekt, albo wcześniej stworzony.

## **Stosuj wzorzec Singleton gdy potrzebujesz ścisłej kontroli nad zmiennymi globalnymi.**

 W przeciwnieństwie do zmiennych globalnych, wzorzec Singleton gwarantuje istnienie tylko jednego obiektu danej klasy. Nic, oprócz samej klasy, nie jest w stanie zamienić tego obiektu.

Zwróć uwagę, że zawsze można zmienić ograniczenie dotyczące ilości i pozwolić na jakąś inną maksymalną liczbę jej instancji. Wystarczy zmienić jeden fragment kodu – metodę `getInstance`.

## **Jak zaimplementować**

1. Dodaj prywatne, statyczne pole klasy celem przechowywania w nim instancji singleton.
2. Zadeklaruj publicznie dostępną, statyczną metodę kreacyjną która daje dostęp do instancji klasy singleton.
3. Zaimplementuj w tej metodzie statycznej “leniwą inicjalizację”. Oznacza to, że nowy obiekt powinien być tworzony tylko przy pierwszym wywołaniu metody i zdeponowany w statycznym polu klasy. Przy kolejnych wywołaniach, metoda powinna zwracać już istniejący egzemplarz.

4. Uczęń konstruktor klasy prywatnym. Statyczna metoda klasy będzie miała do niego dostęp, ale obiekty innych klas – już nie.
5. Przejrzyj kod kliencki i zamień wszystkie bezpośrednie wywoływanie konstruktora klasy singleton na wywołania jej statycznej metody kreacyjnej.

## ⚠ Zalety i wady

- ✓ Masz pewność, że istnieje tylko jedna instancja klasy.
- ✓ Zyskujesz globalny dostęp do tej instancji.
- ✓ Obiekt singleton inicjalizowany jest dopiero wtedy, gdy jest po raz pierwszy potrzebny.
- ✗ Łamie *Zasadę pojedynczej odpowiedzialności*. Wzorzec rozwiązuje bowiem dwa różne problemy jednocześnie.
- ✗ Zastosowanie wzorca Singleton może zamaskować niewłaściwe projektowanie. Można na przykład doprowadzić do sytuacji, w której komponenty programu wiedzą zbyt wiele o sobie nawzajem.
- ✗ Wzorzec ten wymaga specjalnej uwagi w środowisku wielowątkowym, w którym trzeba unikać tworzenia wielu instancji singletona przez wiele wątków.
- ✗ Utrudnieniu mogą ulec testy jednostkowe kodu klienckiego klasy singleton, ponieważ wiele frameworków testujących polega na dziedziczeniu przy produkcji atrap obiektów. Ponieważ konstruktor klasy Singleton jest prywatny, a nadpisywanie sta-

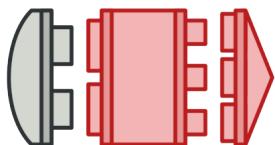
tycznych metod jest niemożliwe w większości języków programowania, będzie trzeba wykazać się kreatywnością i znaleźć jakiś inny sposób tworzenia atrapy singletona. Albo nie pisać testów, albo zrezygnować z tego wzorca.

## ↔ Powiązania z innymi wzorcami

- Klasa **Fasada** może często być przekształcona w **Singleton**, ponieważ pojedynczy obiekt fasady jest w większości przypadków wystarczający.
- **Pyłek** mógłby przypominać **Singleton**, gdybyśmy zdołali zredukować wszystkie współdzielone stany obiektów do tylko jednego obiektu-pyłka. Ale są jeszcze dwie fundamentalne różnice między tymi wzorcami:
  1. Powinna istnieć tylko jedna instancja interfejsu **Singleton**, zaś instancji **Pyłka** będzie wiele, o różnym stanie wewnętrznym.
  2. Obiekt **Singleton** może być zmienny. Pyłki są zaś niezmienne.
- **Fabryki abstrakcyjne, Budowniczych** oraz **Prototypy** można zaimplementować jako **Singletony**.

# Wzorce strukturalne

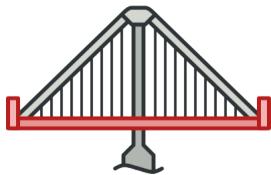
Wzorce strukturalne wyjaśniają w jaki sposób można składać obiekty i klasy w większe struktury zachowując przy tym elastyczność i efektywność tych struktur.



## Adapter

Adapter

Pozwala na współdziałanie obiektów o niekompatybilnych interfejsach.



## Most

Bridge

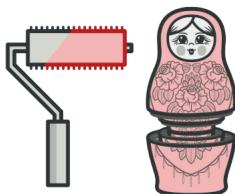
Pozwala podzielić dużą klasę lub blisko spokrewnione ze sobą klasy na dwie hierarchie: abstrakcję oraz implementację, nad którymi można pracować niezależnie od siebie.



## Kompozyt

Composite

Pozwala komponować obiekty w struktury drzewiaste, a potem traktować je tak, jakby były one osobnymi obiektami.



## Dekorator

Decorator

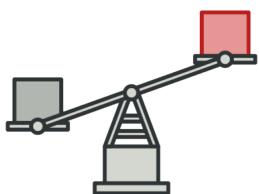
Pozwala nadać dodatkowe obowiązki obiektom poprzez umieszczenie tych obiektów w specjalnych obiektach opakowujących, które zawierają odpowiednie zachowania.



## Fasada

Facade

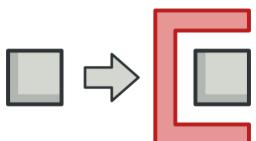
Wyposaży bibliotekę, framework lub inny złożony zestaw klas w uproszczony interfejs.



## Pyłek

Flyweight

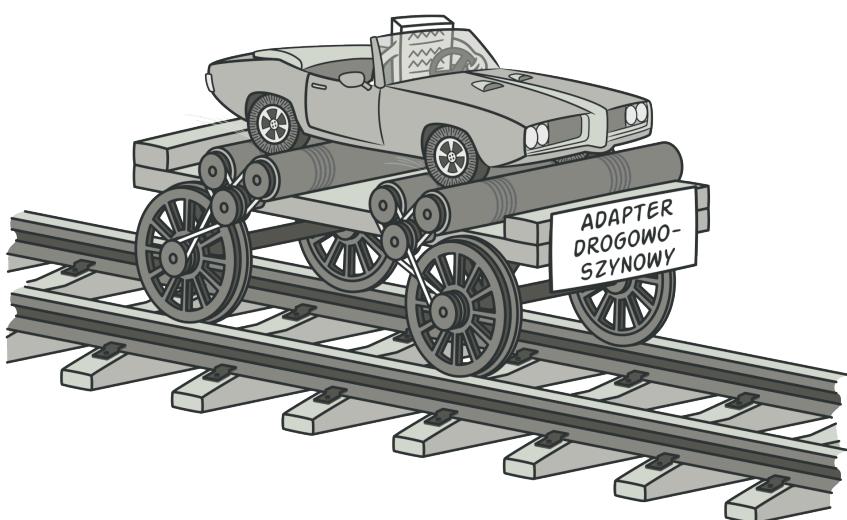
Pozwala zmieścić większą liczbę obiektów w dostępnej pamięci RAM poprzez współdzielenie elementów opisujących stan obiektu. Część opisu stanu jest wspólna dla wielu obiektów, więc nie muszą one zawierać kopii wszystkich danych.



## Pełnomocnik

Proxy

Pozwala tworzyć zastępcę dla innego obiektu. Pełnomocnik nadzoruje dostęp do pierwotnego obiektu, pozwalając na wykonanie jakiejś czynności przed lub po przekazaniu do niego żądania.



# ADAPTER

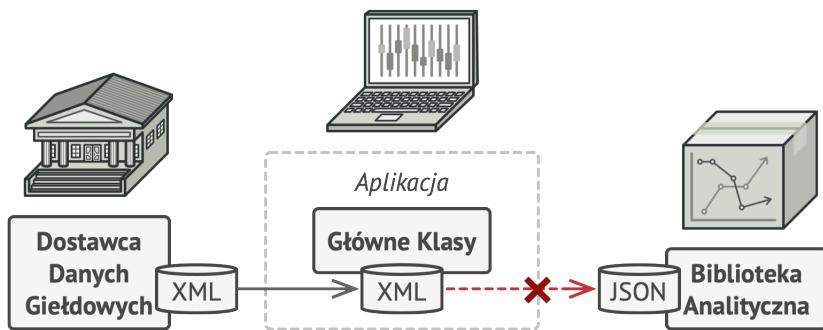
*Znany też jako: Opakowanie, Nakładka, Wrapper*

**Adapter** jest strukturalnym wzorcem projektowym pozwalającym na współdziałanie ze sobą obiektów o niekompatybilnych interfejsach.

## (:() Problem

Wyobraź sobie, że tworzysz aplikację monitorującą giełdę. Pobiera ona dane rynkowe z wielu źródeł w formacie XML, a następnie wyświetla ładnie wyglądające wykresy i diagramy.

Na jakimś etapie postanawiasz wzbogacić aplikację poprzez dodanie inteligentnej biblioteki analitycznej innego producenta. Ale jest haczyk: biblioteka ta działa wyłącznie z danymi w formacie JSON.



*Nie można zastosować biblioteki analitycznej od razu, bo wymaga ona przedstawiania danych w formacie który jest niekompatybilny z twoją aplikacją.*

Można przerobić bibliotekę tak, aby obsługiwała XML. To jednak może naruszyć działanie istniejącego kodu korzystającego z tej biblioteki. Co gorsza, możesz nie mieć dostępu do kodu źródłowego biblioteki, co czyni ten plan niewykonalnym.

## Rozwiążanie

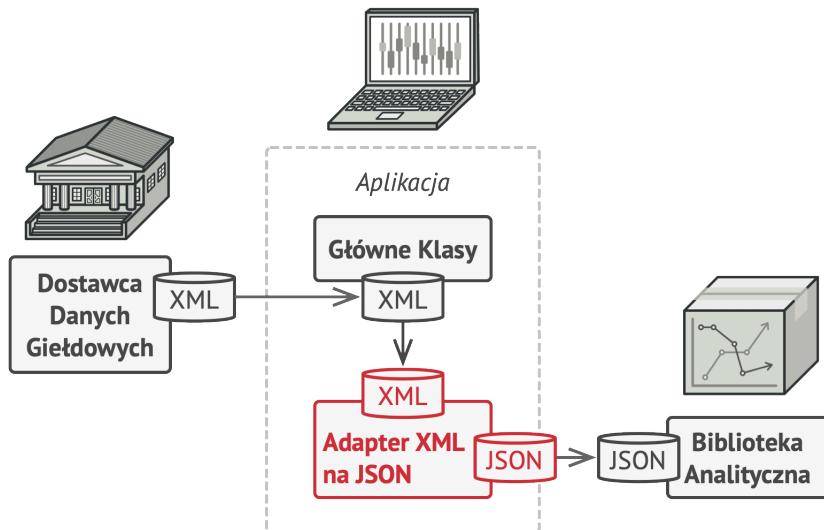
Możesz stworzyć *adapter*. Jest to specjalny obiekt konwertujący interfejs jednego z obiektów w taki sposób, że drugi obiekt go rozumie.

Adapter stanowi swego rodzaju opakowanie dla obiektu, ukrywając szczegóły konwersji jakie odbywają się za kulisami. Obiekt opakowywany może nawet nie wiedzieć o istnieniu adaptera. Można na przykład opakować obiekt korzystający z jednostek kilometr i metr w adapter konwertujący te dane na jednostki imperialne, takie jak stopy i mile.

Adaptery mogą nie tylko konwertować dane pomiędzy formatami, ale również pozwolić na współpracę obiektów o różnych interfejsach. Działa to tak:

1. Adapter uzyskuje interfejs kompatybilny z interfejsem jednego z obiektów.
2. Za pomocą tego interfejsu, istniejący obiekt może śmiało wywoływać metody adaptera.
3. Otrzymawszy wywołanie, adapter przekazuje je dalej, ale już w formacie obsługiwany przez opakowany obiekt.

Czasami jest nawet możliwe stworzenie adaptera dwukierunkowego, potrafiącego konwertować wywołania w obu kierunkach.

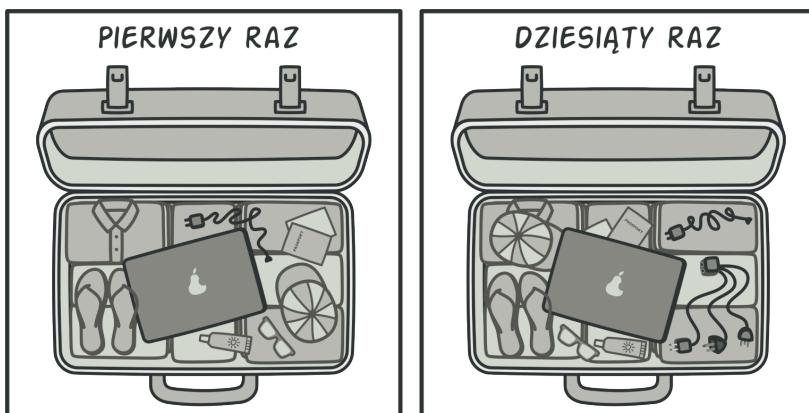


Wróćmy do naszej aplikacji giełdowej. Aby rozwiązać dylemat niezgodności formatów, można stworzyć adaptery XML-do-JSON dla każdej klasy biblioteki analitycznej, która jest używana bezpośrednio przez nasz kod. Potem zaś możemy dostosować kod tak, aby komunikował się z biblioteką wyłącznie za pomocą adapterów. Gdy adapter otrzyma wywołanie, tłumaczy przychodzące dane XML na strukturę JSON i przekazuje wywołanie dalej, do odpowiedniej metody opakowywanego obiektu analitycznego.

## 🚗 Analogia do prawdziwego życia

Gdy pierwszy raz podróżujesz z Polski do Wielkiej Brytanii, możesz zdziwić się próbując naładować laptop. Standardy wtyczek i gniazd są różne.

## PODRÓŻOWANIE ZA GRANICĘ



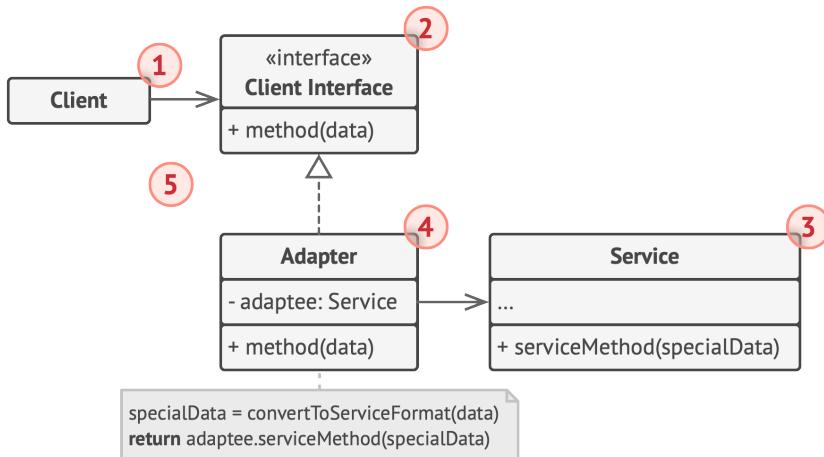
*Walizka przed i po zagranicznej podróży.*

Twoja wtyczka nie będzie pasować do brytyjskiego gniazdka. Problem można rozwiązać za pomocą przejściówka posiadającej gniazdo typu europejskiego oraz wtyk typu brytyjskiego.

## █ Struktura

### Adapter obiektu

Ta implementacja stosuje zasadę kompozycji obiektu: adapter implementuje interfejs jednego z obiektów i opakowuje drugi obiekt. Można to zaimplementować we wszystkich popularnych językach programowania.

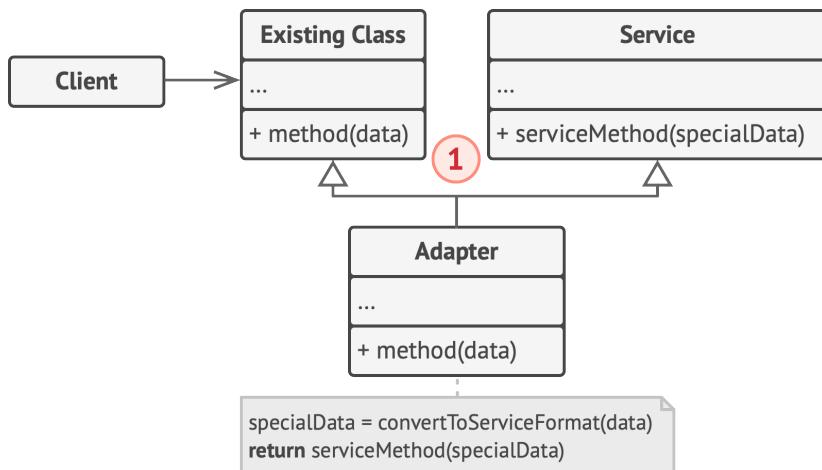


1. **Klient** jest klasą zawierającą istniejącą logikę biznesową programu.
2. **Interfejs Klienta** opisuje protokół którego muszą się trzymać pozostałe klasy by móc współdziałać z kodem klienckim.
3. **Usługa** to jakąś użyteczna klasa (na ogół od innego producenta lub starsza). Klient nie jest w stanie użyć jej bezpośrednio z powodu niekompatybilnego interfejsu.
4. **Adapter** to klasa która jest w stanie współdziałać zarówno z klientem jak i z usługą: implementuje interfejs klienta, opakowując obiekt-usługę. Adapter otrzymuje wywołania od klienta za pośrednictwem interfejsu adaptera i tłumaczy je na wywołania których format zrozumie obiekt udostępniający usługę.

- Kod kliencki nie ulegnie sprzęgnięciu z konkretną klasą adaptera, o ile może współpracować z adapterem za pośrednictwem interfejsu klienta. Dzięki temu można wprowadzać nowe typy adapterów do programu bez psucia istniejącego kodu klienckiego. To przydatne, gdy interfejs klasy udostępniającej usługę ulegnie zmianie, bo wówczas wystarczy dodać nową klasę adapter bez konieczności zmiany kodu klienckiego.

## Adapter klasy

Ta implementacja stosuje dziedziczenie: adapter dziedziczy interfejsy od obu obiektów jednocześnie. Ten sposób można zaimplementować tylko w językach programowania obsługujących wielokrotne dziedziczenie – np. C++.

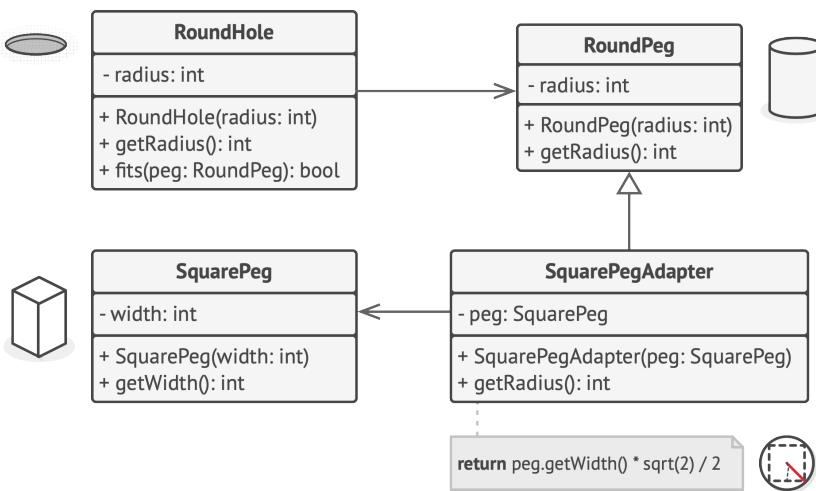


- Adapter Klasy** nie musi opakowywać żadnych obiektów, ponieważ dziedziczy zachowanie zarówno po kliencie, jak i po usłudze. Adaptacja odbywa się wewnątrz przeciążonych metod.

Otrzymany w ten sposób adapter może być użyty w miejsce istniejącej klasy klienckiej.

## # Pseudokod

Ten przykład użycia **AdAPTERA** bazuje na klasycznym konflikcie pomiędzy kwadratowym klockiem i okrągłym otworem.



*Adaptacja kwadratowych klocków do okrągłych otworów.*

Adapter udaje, że jest okrągłym klockiem o promieniu równym połowie przekątnej kwadratu (innymi słowy, promienia najmniejszego okręgu w jakim zmieści się kwadratowy klocek).

```

1 // Powiedzmy że masz dwie klasy o kompatybilnych interfejsach:
2 // RoundHole i RoundPeg.
3 class RoundHole is
4   constructor RoundHole(radius) { ... }
  
```

```
5
6   method getRadius() is
7     // Zwraca promień otworu.
8
9   method fits(peg: RoundPeg) is
10    return this.getRadius() >= peg.getRadius()
11
12 class RoundPeg is
13   constructor RoundPeg(radius) { ... }
14
15   method getRadius() is
16     // Zwraca promień klocka (ang. peg).
17
18
19 // Mamy jednak niekompatybilną klasę: SquarePeg.
20 class SquarePeg is
21   constructor SquarePeg(width) { ... }
22
23   method getWidth() is
24     // Zwrócić długość boku kwadratowego klocka.
25
26 // Klasa adapter pozwala zmieścić kwadratowy klocek w okrągłym
27 // otworze. Rozszerza klasę RoundPeg pozwalając obiektom-
28 // adapterem zachowywać się jak okrągłe klocki.
29 class SquarePegAdapter extends RoundPeg is
30   // W rzeczywistości, adapter zawiera instancję klasy
31   // SquarePeg.
32   private field peg: SquarePeg
33
34   constructor SquarePegAdapter(peg: SquarePeg) is
35     this.peg = peg
36
```

```
37 method getRadius() is
38     // Ten adapter udaje okrągły klocek o promieniu
39     // pozwalającym zmieścić w sobie opakowywany kwadratowy
40     // klocek.
41     return peg.getWidth() * Math.sqrt(2) / 2
42
43
44 // Gdzieś w kodzie klienta.
45 hole = new RoundHole(5)
46 rpeg = new RoundPeg(5)
47 hole.fits(rpeg) // prawda
48
49 small_sqpeg = new SquarePeg(5)
50 large_sqpeg = new SquarePeg(10)
51 hole.fits(small_sqpeg) // to się nie skompiluje (niezgodne typy)
52
53 small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
54 large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
55 hole.fits(small_sqpeg_adapter) // prawda
56 hole.fits(large_sqpeg_adapter) // fałsz
```

## 💡 Zastosowanie

- 💡 Stosuj klasę Adapter gdy chcesz wykorzystać jakąś istniejącą klasę, ale jej interfejs nie jest kompatybilny z resztą twojego programu.
- ⚡ Wzorzec Adapter pozwala utworzyć klasę która stanowi warstwę pośredniczącą pomiędzy twoim kodem, a klasą pocho-

dzającą z zewnątrz, lub inną, posiadającą jakiś nietypowy interfejs.

 **Stosuj ten wzorzec gdy chcesz wykorzystać ponownie wiele istniejących podklas którym brakuje jakiejś wspólnej funkcjonalności, niedającej się dodać do ich nadklasy.**

 Możesz rozszerzyć każdą podkласę i umieścić potrzebną funkcjonalność w nowych klasach pochodnych. Jednak wtedy trzeba by było duplikować kod i umieścić go we wszystkich nowych klasach, a to **psuje zapach kodu**.

Znacznie bardziej eleganckim rozwiązaniem jest umieszczenie brakującej funkcjonalności w klasie adapter i opakowanie nią obiektów pozbawionych potrzebnych funkcji. Aby to zadziałało, klasy docelowe muszą mieć wspólny interfejs, a pole adaptora musi być z nim zgodne. Podejście to bardzo przypomina wzorzec **Dekorator**.

## Jak zaimplementować

1. Upewnij się, że masz przynajmniej dwie klasy o niekompatybilnych interfejsach:
  - Jakąś użyteczną klasę *usługową* której nie możesz zmienić (innego producenta, przestarzałą, albo ze zbyt wieloma istniejącymi zależnościami)

- Jedną lub wiele klas *klienckich* które zyskałyby na możliwości skorzystania z powyższej usługi.
2. Zadeklaruj interfejs klienta i opisz jak ma wyglądać komunikacja klientów z usługą.
  3. Stwórz klasę adapter zgodną z interfejsem klienckim. Póki co, pozostaw metody pustymi.
  4. Dodaj pole do klasy adapter, które przechowa odniesienie do obiektu usługi. Typową praktyką jest inicjalizacja tego pola za pośrednictwem konstruktora, ale czasem wygodniej jest przekazać usługę adapterowi wywołując jego metody.
  5. Jeden po drugim, zaimplementuj wszystkie metody interfejsu klienta w klasie adapter. Adapter powinien delegować większość faktycznej pracy obiektem oferującemu usługę i zajmować się wyłącznie pośrednictwem lub konwersją danych.
  6. Klienci powinni stosować adapter za pośrednictwem interfejsu klienta. Pozwoli to zmieniać lub rozszerzać adaptery bez wpływu na kodu kliencki.

## Zalety i wady

- ✓ *Zasada pojedynczej odpowiedzialności.* Można oddzielić interfejs lub kod konwertujący dane od głównej logiki biznesowej programu.

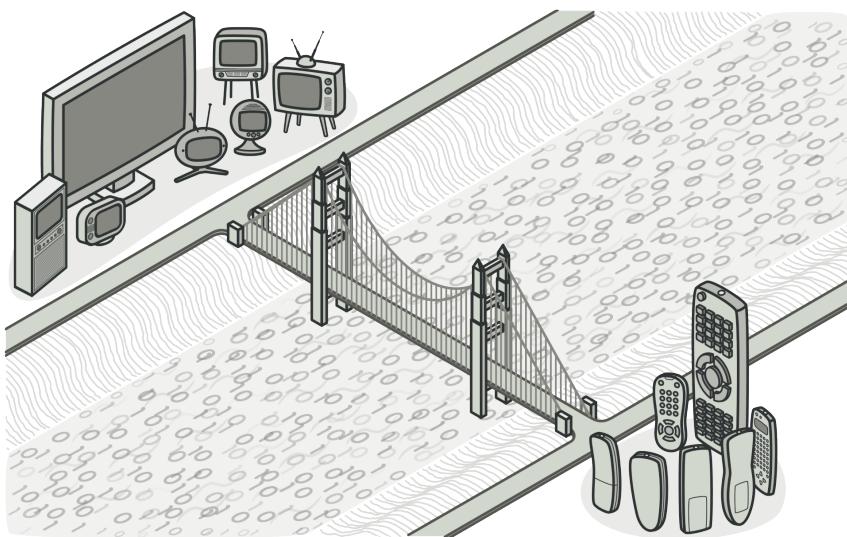
- ✓ *Zasada otwarte/zamknięte.* Można wprowadzać do programu nowe typy adapterów bez psucia istniejącego kodu klienckiego, o ile będzie on korzystał z adapterów poprzez interfejs kliencki.
- ✗ Ogólna złożoność kodu zwiększa się, ponieważ trzeba wprowadzić zestaw nowych interfejsów i klas. Czasem łatwiej zmienić klasę udostępniającą jakąś potrzebną usługę, aby pasowała do reszty kodu.

## ↔ Powiązania z innymi wzorcami

- **Most** zazwyczaj wykorzystuje się od początku projektu, by pozwolić na niezależną pracę nad poszczególnymi częściami aplikacji. Z drugiej strony, **Adapter** jest rozwiązaniem stosowanym w istniejącej aplikacji w celu umożliwienia współpracy pomiędzy niekompatybilnymi klasami.
- **Adapter** zmienia interfejs istniejącego obiektu, zaś **Dekorator** rozszerza go bez zmiany interfejsu. Ponadto *Dekorator* wspiera rekursywną kompozycję, co nie jest możliwe gdy zastosuje się *Adapter*.
- **Adapter** wyposaża “opakowywany” obiekt w inny interfejs, **Pet-nomocnik** w taki sam, zaś **Dekorator** wprowadza rozszerzony interfejs.
- **Fasada** definiuje nowy interfejs istniejącym obiektom, zaś **Adapter** zakłada zwiększenie użyteczności zastanego interfejsu.

*Adapter* na ogół opakowuje pojedynczy obiekt, zaś *Fasada* obejmuje cały podsystem obiektów.

- **Most, Stan, Strategia** (i w pewnym stopniu **Adapter**) mają podobną strukturę. Wszystkie oparte są na kompozycji, co oznacza delegowanie zadań innym obiektom. Jednak każdy z tych wzorców rozwiązuje inne problemy. Wzorzec nie jest bowiem tylko receptą na ustrukturyzowanie kodu w pewien sposób, lecz także informacją dla innych deweloperów o charakterze rozwiązywanego problemu.



# MOST

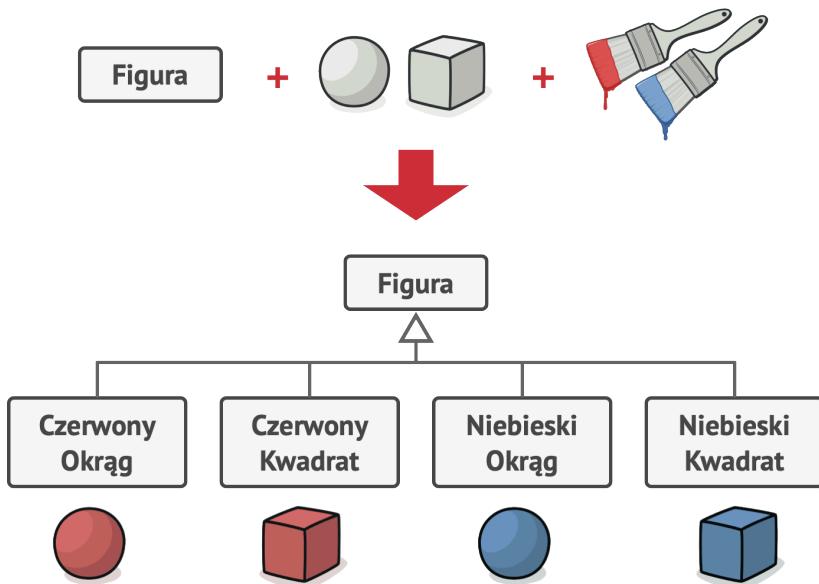
*Znany też jako: Bridge*

**Most** jest strukturalnym wzorcem projektowym pozwalającym na rozdzielenie dużej klasy lub zestawu spokrewnionych klas na dwie hierarchie – abstrakcję oraz implementację. Nad obiema można wówczas pracować niezależnie.

## Problem

*Abstrakcja? Implementacja?* – brzmi strasznie? Spokojnie, zaczniemy od prostego przykładu.

Załóżmy, że mamy klasę `Figura` wraz z dwiema podkласami: `Okrąg` i `Kwadrat`. Chcesz rozszerzyć tę hierarchię klasową, aby zawierała kolory, więc zamierzysz stworzyć podklasy dla figur `Czerwonych` i `Niebieskich`. Jednak ponieważ już są dwie podklasy, konieczne byłoby stworzenie czterech kombinacji, między innymi `NiebieskiOkrąg` i `CzerwonyKwadrat`.



*Ilość kombinacji klas wzrasta w postępie geometrycznym.*

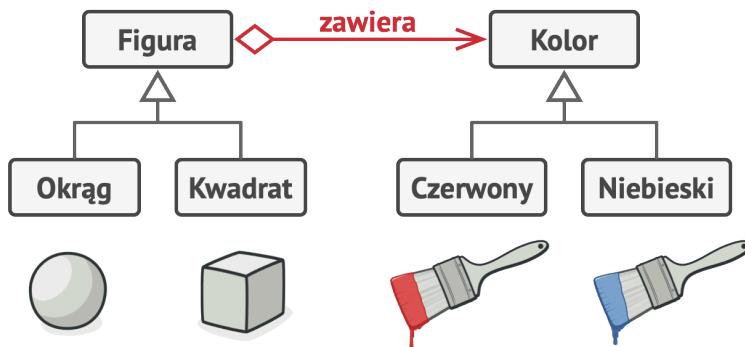
Dodawanie do hierarchii nowych typów figur oraz kolorów spowoduje jej wykładniczy wzrost. Przykładowo, aby dodać fi-

gurę trójkąt, musisz dodać dwie podklasy – po jednej na każdy kolor. Dodanie nowego koloru wymagałoby stworzenia trzech podklas – po jednej na każdą figurę. Im dalej, tym gorzej.

## Rozwiążanie

Problem ten powstaje, ponieważ próbujemy poszerzyć klasy figur w dwóch niezależnych wymiarach: kształt oraz kolor. Jest to bardzo częsty problem przy dziedziczeniu klas.

Wzorzec Most próbuje rozwiązać ten problem poprzez przestawienie się z dziedziczenia na kompozycję obiektów. Oznacza to, że ekstrahujemy jeden z tych wymiarów i tworzymy osobną hierarchię klas, przez co pierwotne klasy będą posiadały odniesienie do obiektów z nowej hierarchii, zamiast przechowywać wszystkie swoje stany i zachowanie wewnętrz klasy.



*Możesz zapobiec eksplozji hierarchii klas do wielkich rozmiarów poprzez przekształcenie jej na kilka powiązanych hierarchii.*

Stosując to podejście, możemy zebrać kod dotyczący kolorów i umieścić go w swojej własnej klasie z dwiema podklasami: `Czerwony` i `Niebieski`. Klasa `Figura` następnie zyskuje pole przechowujące odniesienie do jednego z obiektów-kolorów. W efekcie figura geometryczna może oddelegować wszelkie działania związane z kolorami do odpowiedniego obiektu-koloru. Odniesienie to pełni rolę mostu pomiędzy klasami `Figura` a `Kolor`. Od teraz, dodawanie nowych kolorów nie będzie wymagało zmiany hierarchii figur – i odwrotnie.

## Abstrakcja i implementacja

Książka GoF<sup>1</sup> wprowadza terminy *Abstrakcji* i *Implementacji* jako elementy definicji mostu. Moim zdaniem, pojęcia te brzmią zbyt fachowo i tworzą wrażenie, że wzorzec ten jest przesadnie skomplikowany. Przeczytawszy nasz prosty przykład o figurach i kolorach, spróbujmy rozszyfrować znaczenie niepokojącej terminologii.

*Abstrakcja* (zwana też *interfejsem*) stanowi wysokopoziomową warstwę umożliwiającą kontrolę nad czymś. Nie ma ona wykonywać konkretnych prac, ale delegować zadania do warstwy *implementacyjnej* (zwanej też *platformą*).

- 
1. “Gang of Four – Banda Czterech” to przezwisko nadane czterem autorom znanej książki przedstawiającej wzorce projektowe: *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku* <https://refactoring.guru/gof-book>.

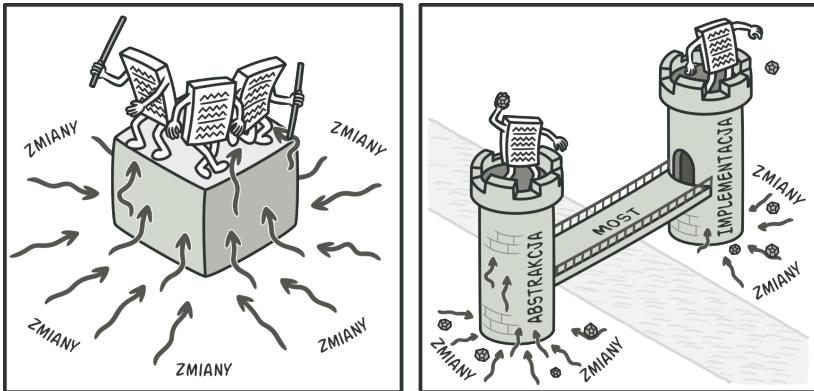
Zwróć uwagę, że nie mówimy tu o znanych ci z języków programowania *interfejsach i klasach abstrakcyjnych* – chodzi o coś innego.

Mówiąc o prawdziwych aplikacjach, za abstrakcję można uznać graficzny interfejs użytkownika (GUI), zaś implementację stanowi znajdujący się poniżej interfejs programowania aplikacji (API). Użytkownik za pomocą interfejsu użytkownika wydaje polecenia niższej warstwie.

Ogólnie mówiąc, taką aplikację można rozwijać w dwóch niezależnych kierunkach:

- Posiadać wiele różnych interfejsów użytkownika (wersje dla zwykłych użytkowników oraz dla administratora)
- Współpracować z wieloma różnymi API (możliwość uruchomienia na systemie Windows, Linux oraz macOS).

W najgorszym przypadku aplikacja będzie przypominała spaghetti, gdzie setki instrukcji warunkowych łączą różne interfejsy użytkownika z różnymi interfejsami programowania aplikacji.

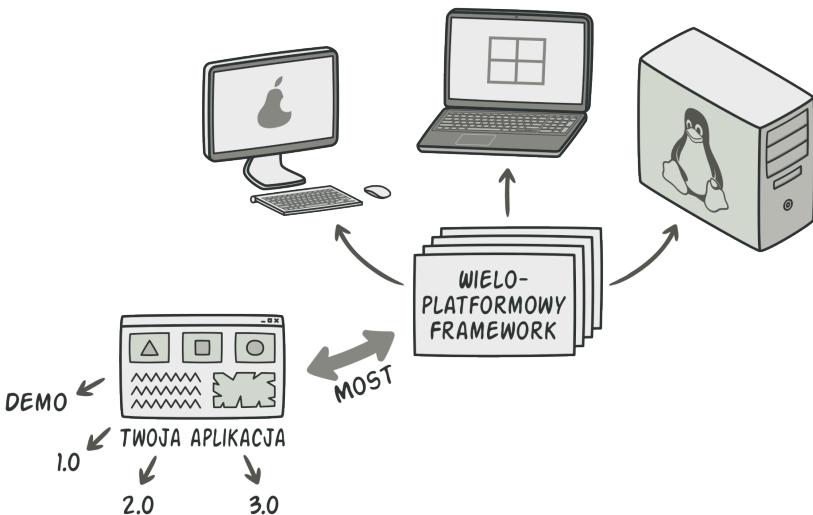


*Dokonywanie nawet najmniejszych zmian w kodzie monolitycznym jest trudne, ponieważ konieczne jest dobre rozumienie całości. Wprowadzanie zmian w małych, dobrze określonych modułach jest znacznie prostsze.*

Można zaprowadzić trochę ładu wśród tego chaosu ekstrahuując kod związanego z kombinacjami interfejsów-platforma i umieszczając go w osobnych klasach. Wkrótce jednak odkryjesz, że takich klas powstanie *mnóstwo*. Hierarchia klasowa rozrośnie się wykładniczo, ponieważ dodanie obsługi nowego GUI lub wsparcia dla nowego API będzie wymagać tworzenia wciąż to nowych klas.

Spróbujmy rozwiązać problem stosując wzorzec Most. Według jego założeń, dzielimy klasy na dwie hierarchie:

- Abstrakcja: warstwa graficznego interfejsu użytkownika aplikacji.
- Implementacja: interfejs programowania aplikacji systemu operacyjnego.

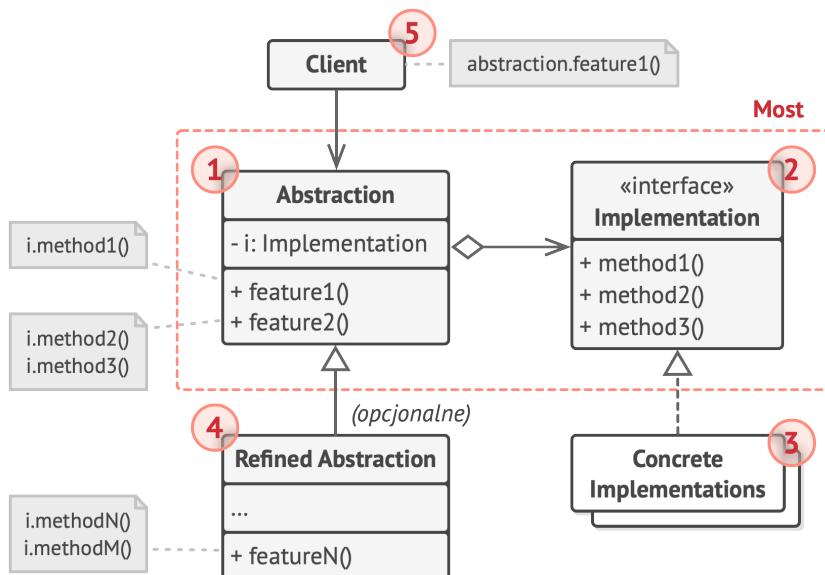


*Jeden ze sposobów ustrukturyzowania aplikacji wieloplatformowej.*

Obiekt abstrakcyjny steruje wyglądem aplikacji, delegując faktyczne zadania do powiązanego z nim obiektu implementacyjnego. Różne implementacje są wymienne, o ile zachowują zgodność ze wspólnym interfejsem, umożliwiając w ten sposób stworzenie jednolitego graficznego interfejsu użytkownika i pod Windows i pod Linux.

W rezultacie, możemy zmieniać klasy GUI bez konieczności modyfikacji klas odnoszących się do API. Co więcej, dodanie obsługi kolejnego systemu operacyjnego wymaga jedynie utworzenia podklasy w hierarchii implementacyjnej.

## Struktura



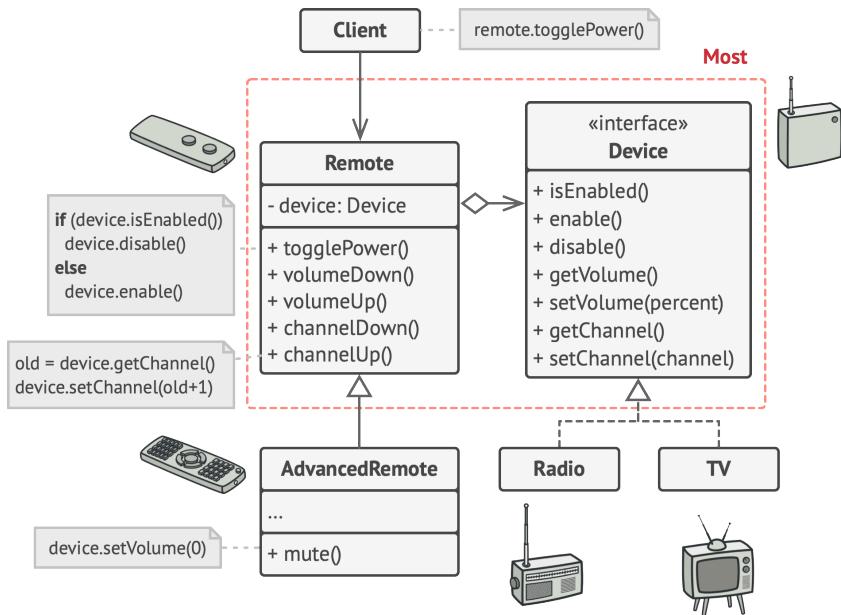
1. **Abstrakcja** obejmuje logikę kontrolną wysokiego poziomu. Potrzebuje obiektu implementacyjnego by wykonywać faktyczne działania.
2. **Implementacja** deklaruje interfejs, który jest wspólny dla wszystkich konkretnych implementacji. Abstrakcja może komunikować się z obiektem implementacyjnym wyłącznie za pomocą zadeklarowanych tu metod.

Abstrakcja może zawierać listę tych samych metod, co implementacja, ale zazwyczaj deklaruje bardziej złożone działania, na które składa się wiele prostszych działań deklarowanych przez implementację.

3. **Konkretnie implementacje** zawierają kod specyficzny dla danej platformy.
4. **Wzbogacona Abstrakcja** udostępnia warianty logiki kontrolnej. Tak jak ich klasa-rodzic, współpracują z różnymi implementacjami poprzez ogólny interfejs implementacyjny.
5. Zazwyczaj, **Klienta** interesuje tylko współpraca z abstrakcją. Ale to zadaniem klienta jest połączenie obiektu abstrakcji z jednym z obiektów implementacji.

## # Pseudokod

Poniższy przykład ilustruje jak wzorzec **Most** może pomóc podzielić monolityczny kod aplikacji zarządzającej urządzeniami i pilotami do nich. Klasy `Urządzenie` stanowią implementację, natomiast `Piloty` – abstrakcję.



Pierwotna hierarchia klas podzielona na dwie części: urządzenia i piloty zdalnego sterowania.

Bazowa klasa pilota deklaruje pole z odniesieniem do obiektu urządzenia. Wszystkie piloty sterują urządzeniami za pośrednictwem uogólnionego interfejsu, który pozwala jednemu pilotowi obsługiwać wiele typów urządzeń.

Można pracować nad klasą pilota zdalnego sterowania niezależnie od klas urządzeń. Potrzeba jedynie nowej podklasy pilota. Na przykład, podstawowy pilot miałby tylko dwa przyciski, ale można by go było wzbogacić o dodatkowe funkcje, jak dodatkowa bateria, lub ekran dotykowy.

Klient dobiera pożądany typ pilota z konkretnym obiektem urządzenia za pośrednictwem konstruktora pilota.

```
1 // "Abstrakcja" definiuje interfejs dla części "kontrolującej"
2 // obu hierarchii klas. Posiada odniesienie do obiektu z
3 // hierarchii "implementacyjnej" i deleguje temu obiekowi
4 // faktyczną pracę.
5 class RemoteControl is
6     protected field device: Device
7     constructor RemoteControl(device: Device) is
8         this.device = device
9     method togglePower() is
10        if (device.isEnabled()) then
11            device.disable()
12        else
13            device.enable()
14    method volumeDown() is
15        device.setVolume(device.getVolume() - 10)
16    method volumeUp() is
17        device.setVolume(device.getVolume() + 10)
18    method channelDown() is
19        device.setChannel(device.getChannel() - 1)
20    method channelUp() is
21        device.setChannel(device.getChannel() + 1)
22
23
24 // Można rozszerzać klasy należące do hierarchii abstrakcyjnej
25 // niezależnie od klas urządzeń.
26 class AdvancedRemoteControl extends RemoteControl is
27     method mute() is
28         device.setVolume(0)
29
30
31 // Interfejs "implementacji" deklaruje metody wspólne dla
32 // wszystkich konkretnych klas implementacji. Nie musi zgadzać
```

```
33 // się z interfejsem abstrakcją. Co więcej, oba interfejsy mogą
34 // być zupełnie różne. Zazwyczaj interfejs implementacji posiada
35 // tylko proste działania, podczas gdy abstrakcja definiuje
36 // działania wysokopoziomowe oparte na tych podstawowych.
37 interface Device is
38     method isEnabled()
39     method enable()
40     method disable()
41     method getVolume()
42     method setVolume(percent)
43     method getChannel()
44     method setChannel(channel)
45
46
47 // Wszystkie urządzenia są zgodne pod względem interfejsu.
48 class Tv implements Device is
49     // ...
50
51 class Radio implements Device is
52     // ...
53
54
55 // Gdzieś w kodzie klienckim.
56 tv = new Tv()
57 remote = new RemoteControl(tv)
58 remote.togglePower()
59
60 radio = new Radio()
61 remote = new AdvancedRemoteControl(radio)
```

## Zastosowanie

-  **Stosuj wzorzec Most gdy chcesz rozdzielić i przeorganizować monolityczną klasę posiadającą wiele wariantów takiej samej funkcjonalności (na przykład, jeśli klasa ma współpracować z wieloma serwerami bazodanowymi).**
  -  Im większa staje się klasa, tym ciężej zrozumieć jej działanie, a dodawanie kolejnych zmian staje się coraz bardziej czasochłonne. Zmiana jednego wariantu funkcjonalności może wymagać dokonania zmian na przestrzeni całej klasy, a to z kolei wiąże się z wprowadzaniem błędów lub przegapieniem jakichś krytycznych efektów ubocznych.
- Wzorzec Most pozwala rozdzielić monolityczną klasę w wiele hierarchii klas. Możemy wówczas zmieniać klasy w jednej z hierarchii niezależnie od klas w drugiej. Podejście to upraszcza utrzymanie kodu i pozwala zminimalizować ryzyko zepsucia istniejącego kodu.
-  **Użyj tego wzorca gdy chcesz rozszerzyć klasę na kilku niezależnych płaszczyznach.**
  -  Most proponuje ekstrakcję osobnej hierarchii klas dla każdej z takich płaszczyzn rozbudowy. Pierwotna klasa deleguje pracę obiektom należącym do tych hierarchii zamiast wykonywać ją sama.

 **Most pozwala spełnić wymóg możliwości wyboru implementacji w trakcie działania programu.**

 Chociaż jest to opcjonalne, Most umożliwia zamianę obiektu implementacji znajdującego się w abstrakcji. Jest to tak proste, jak przypisanie polu klasy nowej wartości.

*Tak przy okazji, ostatnia cecha Mostu jest głównym powodem, dla którego wiele ludzi myli Most ze wzorcem **Strategia**. Pamiętaj, że wzorzec to więcej niż pewien sposób strukturyzowania klas. Może bowiem też sugerować pewien zamiar i wskazać rozwiązanie jakiegoś problemu.*

## Jak zaimplementować

1. Określ płaszczyznę swoich klas. Takie niezależne koncepcje to na przykład: abstrakcja/platforma, domena/infrastruktura, front-end/back-end, interfejs/implementacja.
2. Określ jakie operacje są klientowi potrzebne i zdefiniuj je w bazowej klasie abstrakcji.
3. Określ zakres operacji dostępnych na wszystkich platformach. Zadeklaruj te, których abstrakcja potrzebuje w ogólnym interfejsie implementacji.
4. Stwórz konkretne klasy implementacji dla wszystkich obsługiwanych platform. Upewnij się jednak, aby wszystkie były zgodne z interfejsem implementacji.

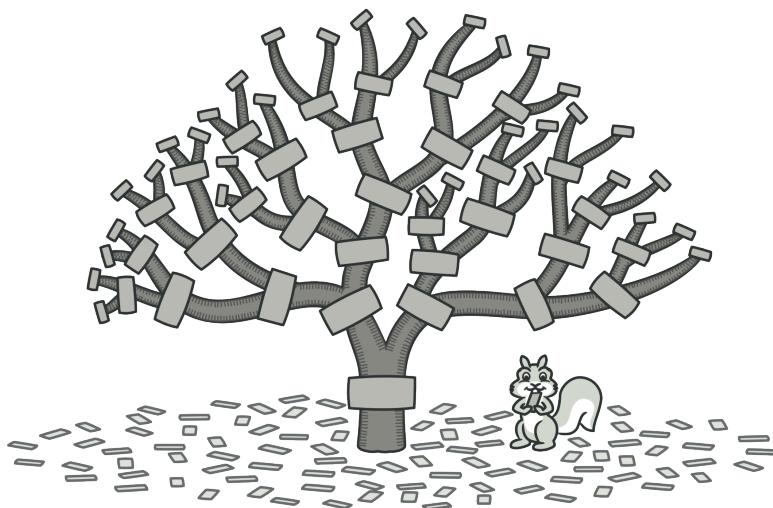
5. Wewnątrz klasy abstrakcji, dodaj pole odnoszące się do typu implementacji. Abstrakcja będzie delegować większość zadań temu obiektowi implementacji.
6. Jeśli masz wiele wariantów wysokopoziomowej logiki, stwórz wzbogacone abstrakcje dla każdego z wariantów poprzez rozszerzenie bazowej klasy abstrakcji.
7. Kod kliencki powinien przekazywać obiekt implementacji konstruktorowi abstrakcji, aby skojarzyć obie hierarchie ze sobą. Po tym, klient może zapomnieć o implementacji i korzystać tylko z obiektu abstrakcji.

## Zalety i wady

- ✓ Możesz tworzyć niezależne od platformy klasy i aplikacje.
- ✓ Kod klienta działa na wyższym poziomie abstrakcji. Nie musi mieć do czynienia ze szczegółami platformy.
- ✓ *Zasada otwarte/zamknięte.* Możesz wprowadzać nowe abstrakcje i implementacje niezależnie od siebie.
- ✓ *Zasada pojedynczej odpowiedzialności.* W abstrakcji możesz skupić się na wysokopoziomowej logice, zaś w implementacji na szczegółach platformy.
- ✗ Kod może stać się bardziej skomplikowany gdy zastosuje się ten wzorzec w przypadku wysoce zwartej klasy.

## ↔ Powiązania z innymi wzorcami

- **Most** zazwyczaj wykorzystuje się od początku projektu, by pozwolić na niezależną pracę nad poszczególnymi częściami aplikacji. Z drugiej strony, **Adapter** jest rozwiązaniem stosowanym w istniejącej aplikacji w celu umożliwienia współpracy pomiędzy niekompatybilnymi klasami.
- **Most, Stan, Strategia** (i w pewnym stopniu **Adapter**) mają podobną strukturę. Wszystkie oparte są na kompozycji, co oznacza delegowanie zadań innym obiektom. Jednak każdy z tych wzorców rozwiązuje inne problemy. Wzorzec nie jest bowiem tylko receptą na ustrukturyzowanie kodu w pewien sposób, lecz także informacją dla innych deweloperów o charakterze rozwiązywanego problemu.
- **Fabryka abstrakcyjna** może być stosowana wraz z **Mostem**. Takie sparowanie jest użyteczne gdy niektóre abstrakcje zdefiniowane przez *Most* mogą współdziałać wyłącznie z określonymi implementacjami. W tym przypadku, *Fabryka abstrakcyjna* może hermetyzować te relacje i ukryć zawiłości przed kodem klienckim.
- Możliwe jest połączenie wzorców **Budowniczy** i **Most**: klasa *kierownik* pełni rolę abstrakcji, zaś poszczególni *budowniczy* stanowią *implementacje*.



# KOMPOZYT

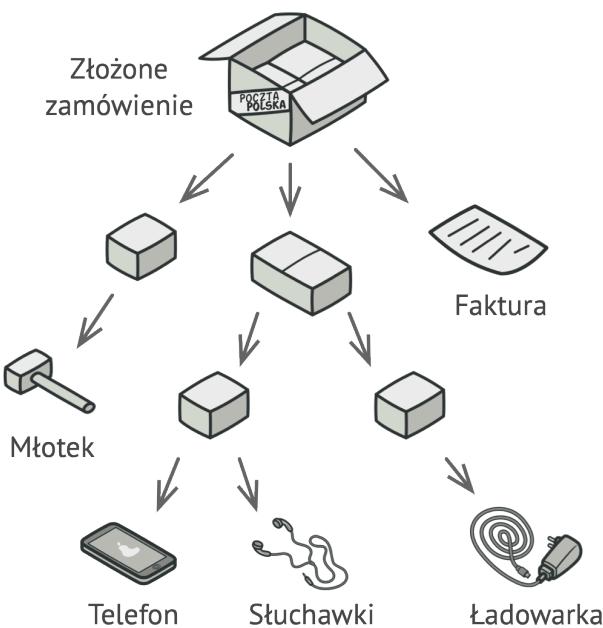
Znany też jako: Drzewo obiektów, Object Tree, Composite

**Kompozyt** to strukturalny wzorzec projektowy pozwalający komponować obiekty w struktury drzewiaste, a następnie traktować te struktury jakby były osobnymi obiektami.

## (:() Problem

Stosowanie wzorca Kompozyt ma sens tylko w przypadku, gdy główny model twojej aplikacji można przedstawić w formie drzewa.

Na przykład, wyobraź sobie, że masz dwa typy obiektów: **Produkty** i **Opakowania**. **Opakowanie** może zawierać wiele **Produktów**, a także pewną liczbę mniejszych **Opakowań**. Te małe **Opakowania** także mogą przechowywać zarówno **Produkty**, jak i jeszcze mniejsze **Opakowania** i tak dalej.



*Zamówienie może obejmować różnorakie produkty opakowane w pudełka, które z kolei są zapakowane w większych pudełkach, i tak dalej. Cała struktura przypomina odwrócone drzewo.*

Założymy, że postanowisz stworzyć system zamawiania, który wykorzystuje jakieś klasy. Zamówienia mogą obejmować proste produkty nie posiadające opakowań, a także pudła wypełnione produktami i... innymi pudełkami. Jak wówczas określić całkowitą wartość pieniężną takiego zamówienia?

Możesz spróbować bezpośredniego podejścia: rozpakuj wszystkie pudełka, przejrzyj ich zawartość i oblicz sumę. W prawdziwym życiu jest to wykonalne, ale w programie nie jest to niestety tak proste, jak działanie w pętli. Musisz z góry znać klasy `Produktów` i `Opakowań` jakie przeglądasz, poziom zagnieżdżenia opakowań i inne kłopotliwe szczegóły. Wszystko to sprawia, że bezpośrednie podejście byłoby problematyczne, albo wręcz niemożliwe.

## Rozwiązanie

Wzorzec projektowy Kompozyt zakłada rozwiązanie w którym zarówno z `Produktami`, jak i `Opakowaniami` pracujemy poprzez wspólny interfejs, deklarujący metodę obliczania całej sumy.

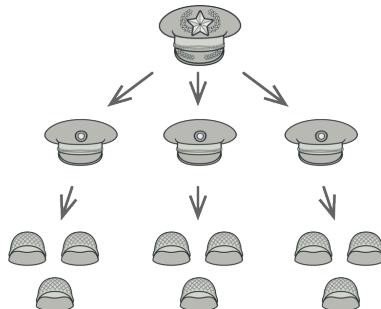
Jak by to działało? W przypadku produktu, metoda zwracałaby jego cenę. W przypadku pudła, przejrzałaby zawartość przedmiot po przedmiocie, pytając o cenę każdego z nich, a na końcu zwróciła całkowitą wartość opakowania. Jeśli któryś z przedmiotów w pudle okazałby się mniejszym pudełkiem, również przejrzałaby jego zawartość i tak aż do obliczenia wartości wszystkich obiektów wewnętrz. Samo opakowanie mogłoby nawet dodawać swoją wartość do ostatecznej ceny.



*Wzorzec Kompozyt pozwala wykonywać działania rekursywne – po wszystkich komponentach drzewa.*

Największą zaletą tego podejścia jest to, że nie musimy się przejmować konkretną klasą obiektów składających się na drzewo. Nie musimy wiedzieć, czy obiekt jest prostym produktem, czy też złożonym kontenerem. Traktujemy wszystko w taki sam sposób, za pomocą takiego samego interfejsu. Gdy wywołasz metodę, same obiekty przekażą ją sobie dalej.

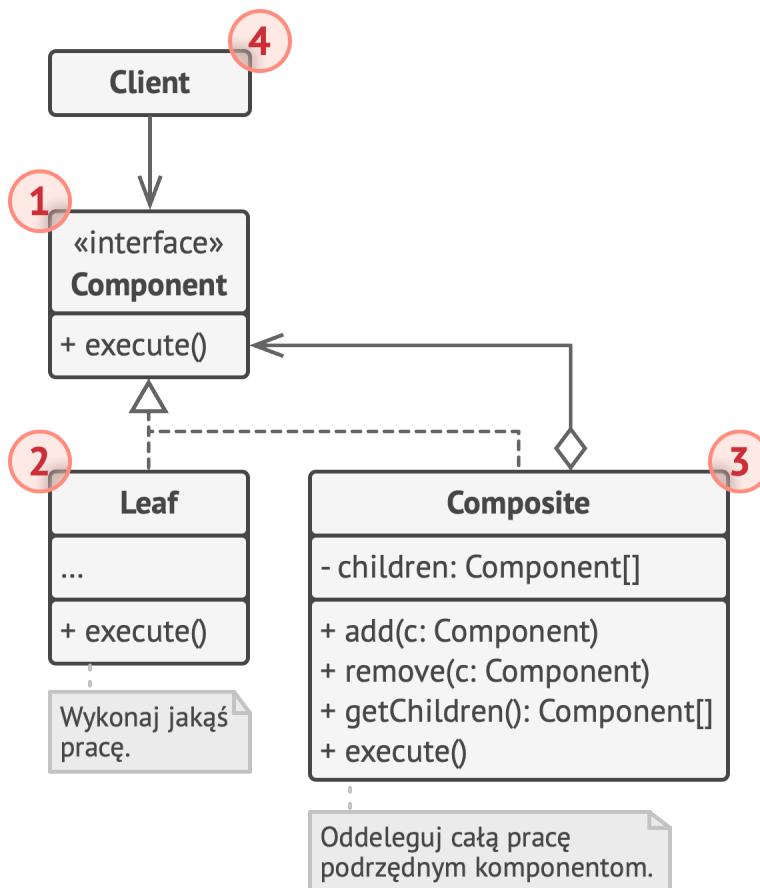
## 💡 Analogia do prawdziwego życia



*Przykład struktury w wojsku.*

Armie większości państw mają strukturę hierarchiczną. Armia składa się z wielu dywizji; dywizje dzielą się na brygady, a brygady składają się z drużyn. Rozkazy wydaje się odgórnie, po czym są one przekazywane w dół, po każdym z poziomów, aż każdy żołnierz będzie wiedział co jest do zrobienia.

## STRUCTURE



1. Interfejs **Komponentu** opisuje operacje wspólne zarówno dla prostych, jak i złożonych elementów drzewa.
2. **Liść** jest podstawowym elementem drzewa i nie posiada elementów podrzędnych.

Zazwyczaj, to właśnie komponenty-liście wykonują większość faktycznej pracy, ponieważ nie mają komu jej zlecić.

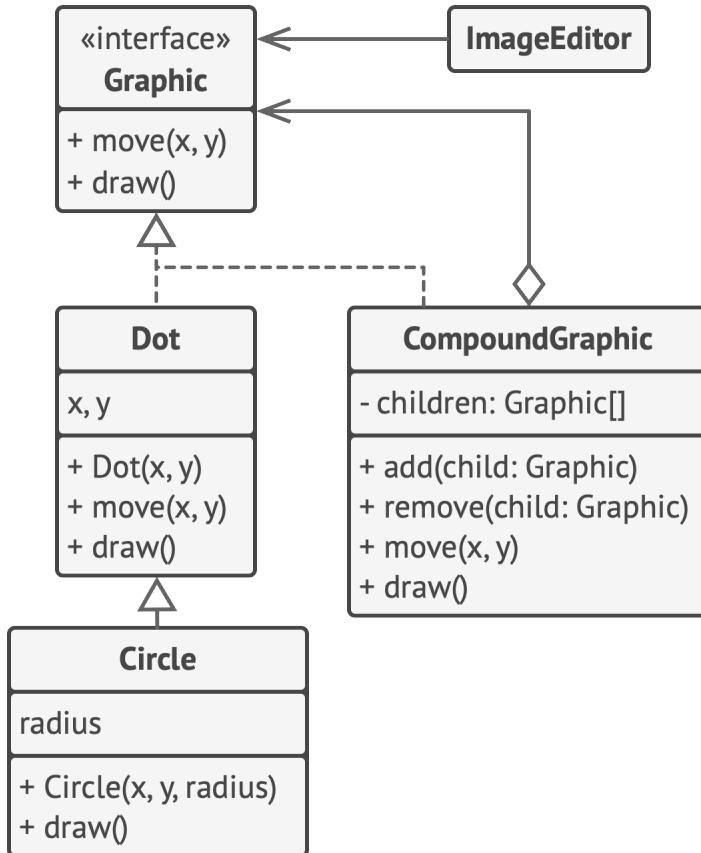
3. **Kontener** (zwany też *kompozytem*) jest elementem posiadającym elementy podrzędne: liście, lub inne kontenery. Kontener nie zna konkretnych klas swojej zawartości. Komunikuje się ze wszystkimi elementami podrzędnymi tylko poprzez interfejs komponentu.

Otrzymawszy żądanie, kontener deleguje pracę do swoich podelementów, przetwarza wyniki pośrednie i zwraca ostateczny wynik klientowi.

4. **Klient** współpracuje ze wszystkimi elementami za pośrednictwem interfejsu komponentu. W wyniku tego klient może działać w taki sam sposób zarówno na prostych, jak i złożonych elementach drzewa.

## # Pseudokod

W tym przykładzie, wzorzec **Kompozyt** pozwala zaimplementować układanie figur geometrycznych w stos w programie graficznym.



*Przykład edytora figur geometrycznych.*

Klasa **ZłożonaGrafika** jest kontenerem, na który może składać się każda liczba podrzędnych figur, w tym inne figury złożone. Figura złożona ma takie same metody jak pojedyncza figura. Jednakże, zamiast robić coś samodzielnie, figura złożona przekazuje żądanie rekursywnie wszystkim swoim elementom, a sama “zsumowuje” wynik.

Kod kliencki współpracuje ze wszystkimi figurami za pomocą interfejsu który jest jednakowy dla wszystkich klas figur. Tym

samym klient nie jest świadomy, czy ma do czynienia z prostym kształtem, czy złożonym. Klient może pracować z bardzo złożonymi strukturami bez wiązania się z konkretnymi klasami tworzącymi strukturę.

```
1 // Interfejs komponentu deklaruje działania wspólne zarówno dla
2 // prostych jak i skomplikowanych obiektów struktury.
3 interface Graphic is
4     method move(x, y)
5     method draw()
6
7 // Klasa liść reprezentuje końcowe elementy struktury. Liść nie
8 // posiada pod-obiektów. Na ogół to właśnie te obiekty wykonują
9 // faktyczne działania, zaś obiekty złożone jedynie je delegują
10 // swoim pod-komponentom.
11 class Dot implements Graphic is
12     field x, y
13
14     constructor Dot(x, y) { ... }
15
16     method move(x, y) is
17         this.x += x, this.y += y
18
19     method draw() is
20         // Narysuj kropkę w miejscu o współrzędnych X i Y.
21
22 // Wszystkie klasy komponentów mogą rozszerzać inne komponenty.
23 class Circle extends Dot is
24     field radius
25
26     constructor Circle(x, y, radius) { ... }
```

```
27
28     method draw() is
29         // Narysuj okrąg w punkcie X i Y o promieniu R.
30
31     // Klasa kompozyt reprezentuje komponenty złożone które mogą
32     // posiadać potomstwo. Obiekty kompozytowe zazwyczaj delegują
33     // faktyczną pracę swoim dzieciom a następnie "podsumowują"
34     // otrzymane wyniki.
35     class CompoundGraphic implements Graphic is
36         field children: array of Graphic
37
38         // Obiekt będący kompozytem może dodawać lub usuwać inne
39         // komponenty (zarówno proste jak i złożone) do/ze swojej
40         // listy obiektów-dzieci.
41         method add(child: Graphic) is
42             // Dodaj obiekt potomny do tablicy potomstwa.
43
44         method remove(child: Graphic) is
45             // Usuń obiekt-dziecko z tablicy potomstwa.
46
47         method move(x, y) is
48             foreach (child in children) do
49                 child.move(x, y)
50
51         // Kompozyt wykonuje swoje podstawowe zadania w konkretny
52         // sposób. Przechodzi rekurencyjnie przez wszystkie obiekty
53         // podrzędne, zbierając od nich wyniki i je sumując. Skoro
54         // obiekty-dzieci kompozytu przekazują te wywołania swoim
55         // obiektom-dzieciom i tak dalej, całe drzewo obiektów
56         // zostaje przejrzane.
57         method draw() is
58             // 1. Dla każdego komponentu-dziecka:
```

```
59      //      – Narysuj komponent.  
60      //      – Zaktualizuj otaczający prostokąt.  
61      // 2. Narysuj prostokąt przerywaną linią korzystając ze  
62      // współrzędnych granicy.  
63  
64  
65  // Kod klienta współpracuje ze wszystkimi komponentami za  
66  // pośrednictwem ich interfejsu bazowego. Dzięki temu kod  
67  // kliencki posiada wsparcie zarówno prostych obiektów-liści,  
68  // jak i złożonych kompozytów.  
69  class ImageEditor is  
70      field all: CompoundGraphic  
71  
72  method load() is  
73      all = new CompoundGraphic()  
74      all.add(new Dot(1, 2))  
75      all.add(new Circle(5, 3, 10))  
76      // ...  
77  
78  // Połącz wybrane komponenty w jeden złożony komponent  
79  // kompozytowy.  
80  method groupSelected(components: array of Graphic) is  
81      group = new CompoundGraphic()  
82      foreach (component in components) do  
83          group.add(component)  
84          all.remove(component)  
85      all.add(group)  
86      // Wszystkie komponenty zostaną narysowane.  
87      all.draw()
```

## Zastosowanie

-  **Stosuj wzorzec Kompozyt gdy musisz zaimplementować drzewistą strukturę obiektów.**
-  Wzorzec Kompozyt określa dwa podstawowe typy elementów współdzielących jednakowy interfejs: proste liście oraz złożone kontenery. Kontener może być złożony zarówno z liści, jak i z innych kontenerów. Pozwala to skonstruować zagnieżdzoną, rekurencyjną strukturę obiektów przypominającą drzewo.
-  **Stosuj ten wzorzec gdy chcesz, aby kod kliencki traktował zarówno proste, jak i złożone elementy jednakowo.**
-  Wszystkie elementy zdefiniowane przez wzorzec Kompozyt współdzielą jeden interfejs. Dzięki temu, klient nie musi martwić się konkretną klasą obiektów z jakimi ma do czynienia.

## Jak zaimplementować

1. Upewnij się, że główny model twojej aplikacji można przedstawić w formie struktury drzewistej. Spróbuj rozdzielić go na proste elementy i kontenery. Pamiętaj, że kontenery muszą móc zawierać w sobie zarówno proste elementy, jak i inne kontenery.
2. Zadeklaruj interfejs komponentu z listą metod które mają sens zarówno w przypadku prostych, jak i złożonych komponentów.

3. Stwórz klasę-liść reprezentującą proste elementy. Program może posiadać wiele różnych klas-liści.
4. Stwórz klasę-kontener, reprezentującą złożone elementy. W tej klasie umieść pole tablicowe, które przechowywać będzie odniesienia do elementów podrzędnych. Tablica musi być w stanie przechowywać zarówno liście, jak i kontenery, więc zadeklaruj jej typ jako interfejs komponentu.

Implementując metody interfejsu komponentu, pamiętaj, że kontener ma delegować większość swych obowiązków elementom podrzędnym.

5. Na koniec zdefiniuj metody pozwalające dodawać i usuwać elementy podrzędne w kontenerze.

Pamiętaj, że powyższe operacje można zadeklarować w interfejsie komponentu. Łamie to *Zasadę segregacji interfejsów*, ponieważ metody będą puste w klasie-liść. W zamian, klient będzie w stanie traktować wszystkie elementy jednakowo, nawet komponując drzewo.

## Zalety i wady

- ✓ Można pracować ze skomplikowanymi strukturami drzewiastymi w wygodny sposób: wykorzystaj na swoją korzyść polimorfizm i rekursję.

- ✓ *Zasada otwarte/zamknięte.* Możesz wprowadzać do programu obsługę nowych typów elementów bez psucia istniejącego kodu, gdyż pracuje on teraz z drzewem różnych obiektów.
- ✗ Ustalenie wspólnego interfejsu dla klas o diametralnie różnych funkcjonalnościach może okazać się trudne. W pewnych przypadkach trzeba przesadnie uogólnić interfejs komponentu, co uczyni go trudniejszym do zrozumienia.

## ↔ Powiązania z innymi wzorcami

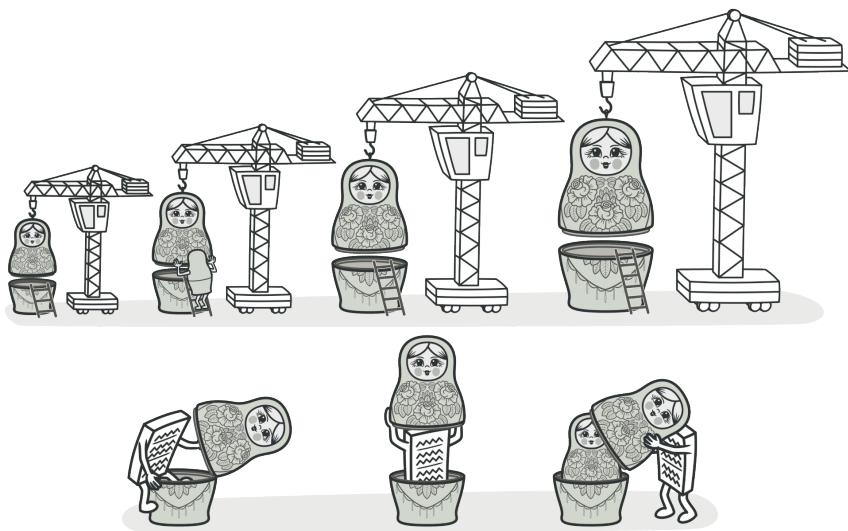
- Możesz zastosować wzorzec **Budowniczy** by tworzyć złożone drzewa **Kompozytowe** dzięki możliwości zaprogramowania ich etapów konstrukcji tak, aby odbywały się rekurencyjnie.
- **Łańcuch zobowiązań** często stosuje się w połączeniu z **Kompozytem**. W takim przypadku, gdy komponent-liść otrzymuje żądanie, może je przekazać poprzez łańcuch nadzędnych komponentów aż do korzenia drzewa obiektów.
- **Iteratory** służą do sekwencyjnego przemieszczania się po drzewie **Kompozytowym** element po elemencie.
- **Odwiedzający** może wykonać działanie na całym drzewie **Kompozytowym**.
- Węzły będące liśćmi drzewa **Kompozytowego** można zaimplementować jako **Pyłki** by zaoszczędzić nieco pamięci RAM.

- **Kompozyt i Dekorator** mają podobne diagramy struktur ponieważ oba bazują na rekursywnej kompozycji w celu zorganizowania nieokreślonej liczby obiektów.

*Dekorator* przypomina *Kompozyt*, ale posiada tylko jeden element podrzędny. Ponadto, kolejną różnicą jest to, że *Dekorator* przypisuje dodatkowe obowiązki opakowanemu obiekowi, zaś *Kompozyt* jedynie “sumuje” wyniki otrzymane od elementów podrzędnych.

Wzorce mogą też współpracować: *Dekorator* może służyć rozszerzeniu zachowania określonego obiektu w drzewie *Kompozytowym*

- Projekty intensywnie korzystające ze wzorców **Kompozyt i Dekorator** mogą skorzystać również na zastosowaniu **Prototypu**. Zastosowanie tego wzorca pozwala klonować złożone struktury zamiast konstruować je ponownie od zera.



# DEKORATOR

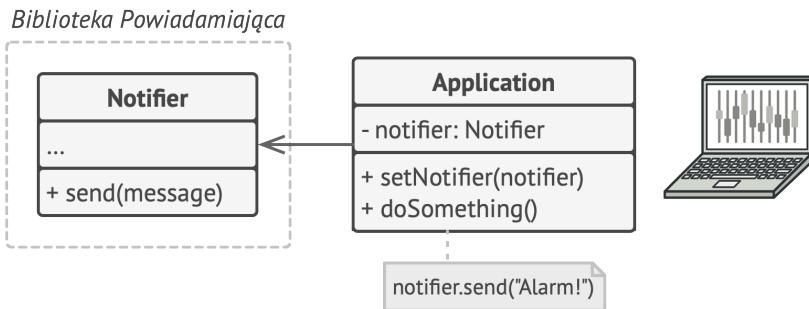
Znany też jako: *Nakładka, Wrapper, Decorator*

**Dekorator** to strukturalny wzorzec projektowy pozwalający dodawać nowe obowiązki obiektom poprzez umieszczanie tych obiektów w specjalnych obiektach opakowujących, które zawierają odpowiednie zachowania.

## Problem

Wyobraź sobie, że pracujesz nad biblioteką powiadamiającą, która pozwala innym programom informować użytkowników o istotnych zdarzeniach.

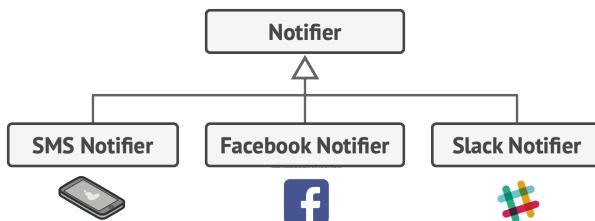
Wstępna wersja biblioteki bazowała na klasie `Powiadamiacz`, która miała tylko parę pól, konstruktor oraz jedną metodę – `wyslij`. Metoda przyjmowała wiadomość jako argument od klienta i przesyłała tę wiadomość do listy emailów, które z kolei przekazywano powiadamiaczowi przez jego konstruktor. Aplikacja innego producenta, pełniąca rolę klienta, miała za zadanie stworzyć i skonfigurować obiekt powiadamiacza jednorazowo, potem zaś korzystać z niego w razie istotnych zdarzeń.



*Program może korzystać z klasy powiadamiającej by wysyłać powiadomienia o ważnych wydarzeniach na określony zestaw adresów email.*

W jakimś momencie zauważasz, że użytkownicy biblioteki oczekują więcej, niż tylko przesyłania maila. Wielu chce otrzy-

mywać SMS gdy zdarzy się coś bardzo ważnego. Inni z kolei chcą być powiadomiani poprzez Facebooka, a użytkownicy korporacyjni byliby zachwyceni powiadomieniami Slack.

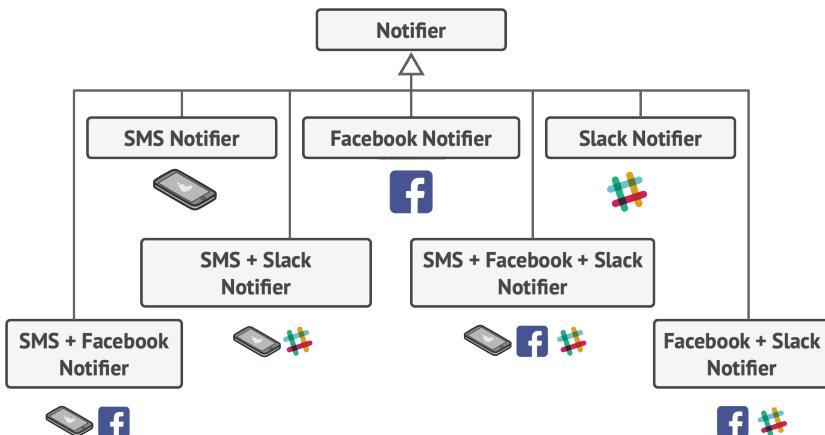


*Każdy rodzaj powiadomienia zaimplementowano w osobnej podklasie powiadamiača.*

Czy to takie trudne? Wystarczy rozszerzyć klasę `Powiadamiač` i umieścić dodatkowe metody powiadamiania w nowych podklasach. Teraz klient powinien stworzyć instancję potrzebnej mu klasy powiadomień i może korzystać do woli.

Wszystko w porządku do momentu, aż ktoś zada całkiem sensowne pytanie: “Czemu nie da się przesyłać powiadomienia wieloma drogami naraz? Przecież jeśli w twoim domu wybuchnie pożar, warto zastosować wszelkie możliwe opcje powiadamiania”.

Próbowiesz więc spełnić to wymaganie tworząc specjalne podklasy łączące różne metody powiadamiania w jednej klasie. Jednakże, szybko okazuje się oczywiste, że wskutek tego podejścia kod strasznie spuchł, i to nie tylko po stronie biblioteki, ale i klienta.



*Kombinatoryczna eksplozja podklas.*

Trzeba znaleźć jakiś inny pomysł na strukturę klas powiadomień, aby uniknąć wpisania do księgi rekordów Guinessa przy dodawaniu kolejnych możliwości powiadamiania.

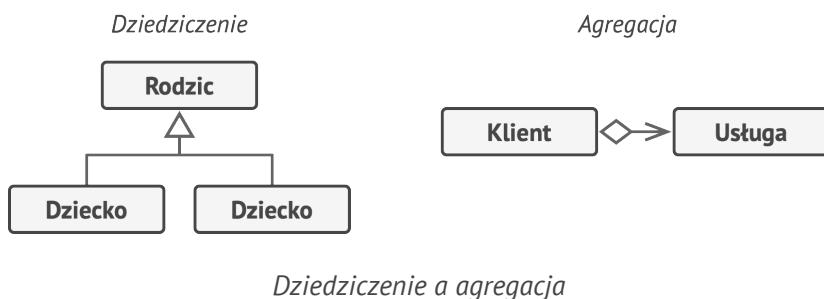
## 😊 Rozwiążanie

Rozszerzenie klasy jest pierwszym sposobem jaki przychodzi do głowy, gdy stajemy wobec konieczności zmiany zachowania się obiektu. Jednakże dziedziczenie wiąże się z wieloma obciążeniami, o których trzeba pamiętać.

- Dziedziczenie jest statyczne. Nie da się zmienić zachowania istniejącego obiektu po uruchomieniu programu. Można tylko zastąpić cały obiekt innym, stworzonym z innej podkłasy.
- Podklasy mogą mieć tylko jedną klasę-rodzica. W większości języków nie można odziedziczyć zachowania wielu klas jednocześnie.

Jednym ze sposobów uniknięcia tych ograniczeń jest zastosowanie *Agregacji* lub *Kompozycji*<sup>1</sup> zamiast *Dziedziczenia*. Obie alternatywy działają prawie tak samo: jeden z obiektów *posiada* odniesienie do innego i deleguje mu jakąś pracę, zaś w przypadku dziedziczenia, obiekt *sam jest* w stanie wykonać tę pracę, dziedzicząc zachowanie od swej nadklasy.

Dzięki temu nowemu sposobowi można łatwo zamienić obiekt, z którym istnieje połączenie, na inny, tym samym zmieniając zachowanie kontenera w czasie działania programu. Obiekt może korzystać z zachowań różnych klas, mając odniesienia do wielu obiektów i delegując im różne rodzaje zadań. Agregacja/kompozycja jest kluczową koncepcją wielu wzorców projektowych. Nie inaczej jest z Dekoratorem. Wróćmy więc do omówienia wzorca.



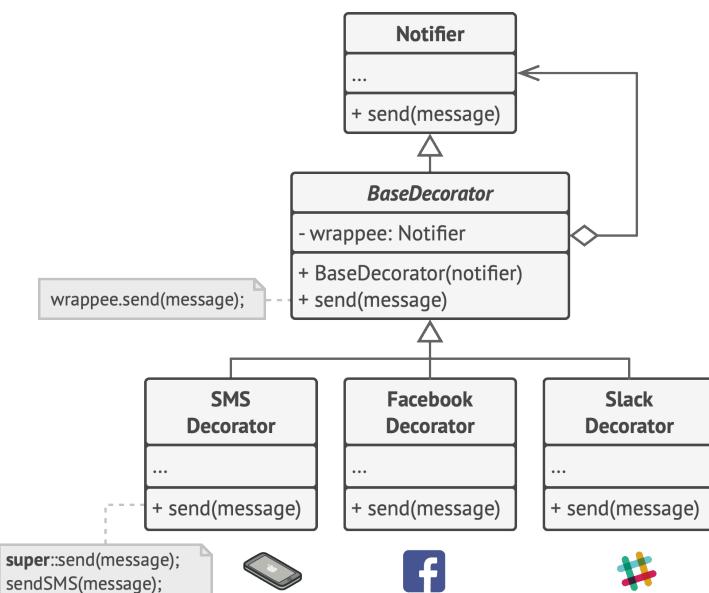
Dekorator znany jest też pod nazwą “Nakładka”. To słowo dobrze wyraża główną ideę tego wzorca. *Nakładka* jest obiektem który może być połączony z jakimś *docelowym* obiektem. Nakładka za-

- 
1. *Agregacja*: obiekt A zawiera obiekty B; B może istnieć bez A.  
*Kompozycja*: obiekt A składa się z obiektów B; A zarządza cyklem życia B; B nie może istnieć samodzielnie.

wiera ten sam zestaw metod jak obiekt docelowy i deleguje mu wszelkie otrzymywane żądania. Jednak nakładka może wpłynąć na rezultat, wykonując coś albo przed, albo po przekazaniu żądania.

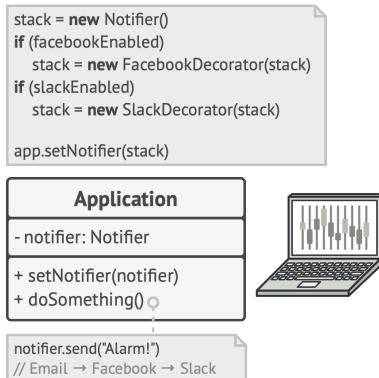
Kiedy więc prosta nakładka staje się prawdziwym dekoratorem? Jak wspomniałem, nakładka implementuje ten sam interfejs co “opakowywany” obiekt. Dlatego też z punktu widzenia klienta te obiekty są identyczne. Niech pole referencyjne nakładki przyjmie każdy obiekt zgodny z tym interfejsem, pozwoli to wówczas “przykryć” obiekt wieloma warstwami, sumując tym samym zachowania każdej z nich.

W naszym przykładzie z powiadomieniami, zostawmy proste powiadomienie mailowe w klasie bazowej `Powiadamiacz`, ale zmieńmy inne metody powiadamiania w dekoratory.



*Różne metody powiadamiania stały się dekoratorami.*

Kod kliencki musiałby opakować podstawowy obiekt powiadacza w zestaw dekoratorów stosowny do preferencji klienta. Wynikowy obiekt będzie miał strukturę stosu.

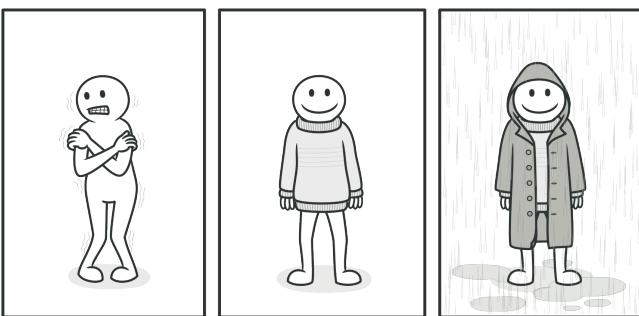


*Aplikacje mogą tworzyć złożone konfiguracje stosów dekoratorów powiadamiaczy.*

Ostatnim dekoratorem w stosie będzie obiekt, z którym klient faktycznie pracuje. Skoro wszystkie dekoratory implementują ten sam interfejs co powiadamiaż bazowy, reszta kodu klienta nie będzie musiała wiedzieć, czy pracuje na “czystym” obiekcie powiadamiača, czy “udekorowanym”.

Moglibyśmy zastosować to samo podejście wobec innych obwiązków, jak formatowanie wiadomości lub komponowanie listy odbiorców. Klient może udekortować obiekt dowolnymi dekoratorami, o ile będą one zgodne ze sobą co do interfejsu.

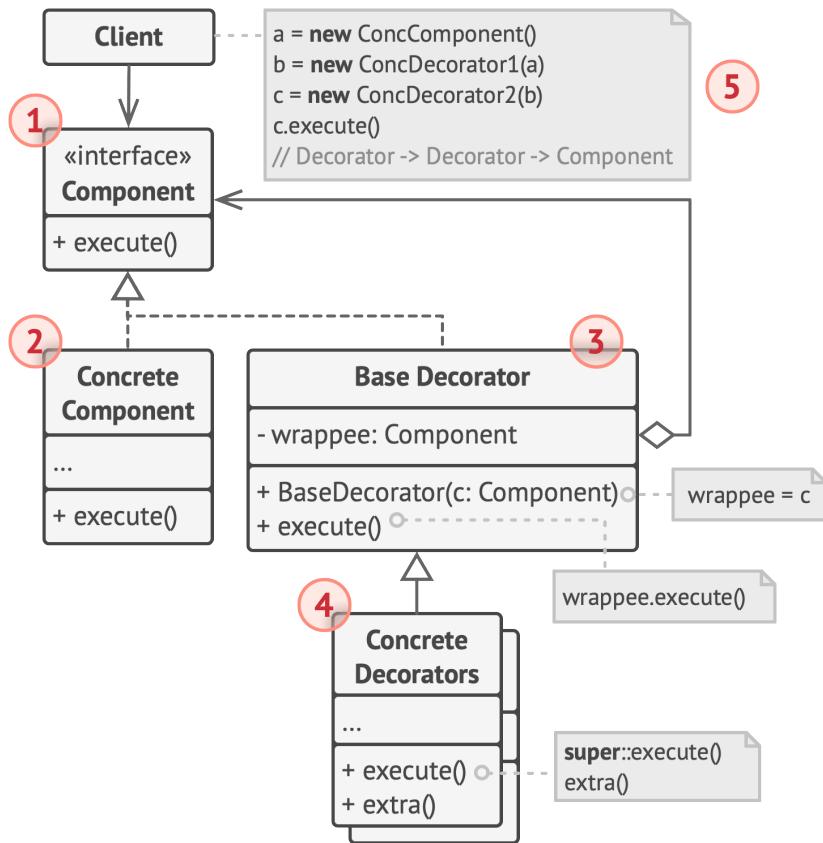
## Analoga do prawdziwego życia



Zyskujesz połączony efekt poszczególnych elementów ubioru.

Noszenie ubrań jest przykładem stosowania dekoratorów. Gdy ci zimno, zakładasz sweter. Jeśli dalej ci zimno, zakładasz jeszcze kurtkę. A jeśli do tego pada deszcz, możesz założyć płaszcz przeciwdeszczowy. Wszystkie te elementy ubioru "rozszerzają" twoje domyślne zachowanie, ale nie są częścią ciebie i możesz pozbyć się każdego z nich gdy nie jest ci akurat potrzebny.

## Struktura

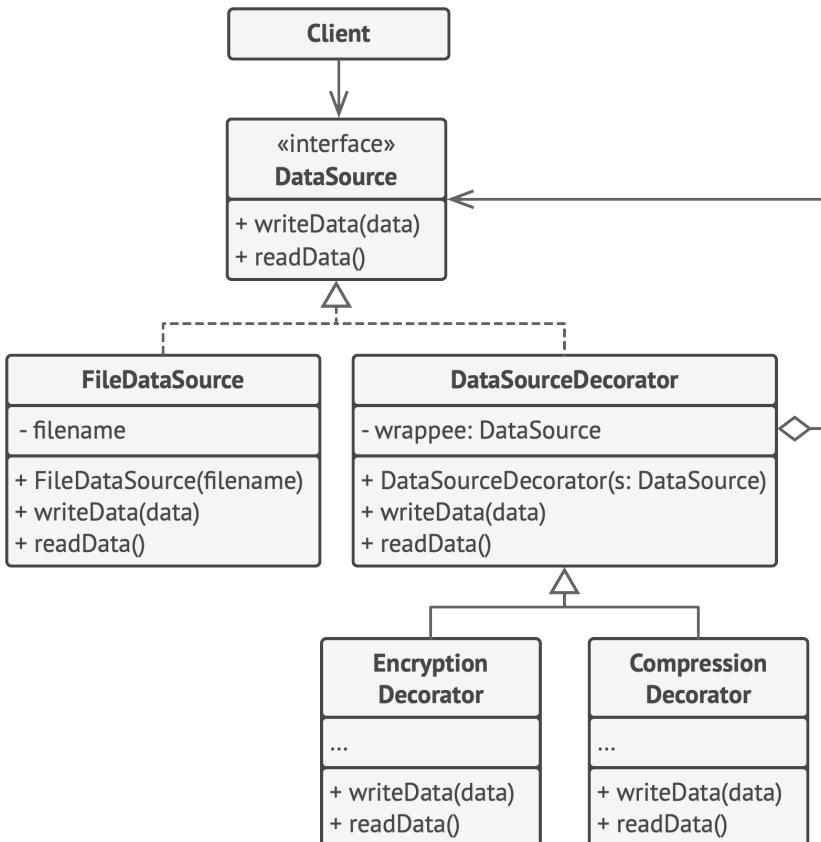


1. **Komponent** deklaruje interfejs wspólny zarówno dla nakładek, jak i opakowywanych obiektów.
2. **Konkretny Komponent** to klasa opakowywanych obiektów. Definiuje ona podstawowe zachowanie, które następnie można zmieniać za pomocą dekoratorów.

3. Klasa **Bazowy Dekorator** posiada pole przeznaczone na referencję do opakowywanego obiektu. Typ pola powinien być zadeklarowany jako interfejs komponentu, aby mogło przechować zarówno konkretne komponenty, jak i inne dekoratory. Dekorator bazowy deleguje wszystkie działania opakowanemu obiekowi.
4. **Konkretni Dekoratorzy** definiują dodatkowe zachowania które można przypisać do komponentów dynamicznie. Konkretni dekoratorzy nadpisują metody dekoratora bazowego i wykonują swoje działania albo przed, albo po wywołaniu metody klasycznego.
5. **Klient** może opakowywać komponenty w wiele warstw dekoratorów, o ile działa na wszystkich obiektach poprzez interfejs komponentu.

## # Pseudokod

W tym przykładzie, wzorzec **Dekorator** pozwala skompresować i zaszyfrować wrażliwe dane niezależnie od kodu który faktycznie korzysta z tych danych.



*Przykład dekoratorów kompresujących i szyfrujących.*

Aplikacja opakowuje źródło danych w parę dekoratorów. Obie nakładki zmieniają sposób, w jaki dane są zapisywane na i odczytywane z dysku:

- Tuż przed **zapisaniem na dysk** danych, dekoratory szyfrują i kompresują je. Pierwotna klasa zapisuje do pliku dane już zaszyfrowane i skompresowane – bez wiedzy o dokonanej obróbce.

- Tuż po **odczytaniu z dysku** danych, przechodzą one przez te same dekoratory, które je dekompresują i deszyfrują.

Dekoratory i klasa źródła danych implementują ten sam interfejs, co czyni je wymienialnymi w kodzie klienta.

```
1 // Interfejs komponentu definiuje działania które można
2 // modyfikować za pomocą dekoratorów.
3 interface DataSource is
4     method writeData(data)
5     method data readData():data
6
7 // Konkretne komponenty dostarczają domyślnych implementacji
8 // działań. Może istnieć wiele odmian tych klas w całym
9 // programie.
10 class FileDataSource implements DataSource is
11     constructor FileDataSource(filename) { ... }
12
13     method writeData(data) is
14         // Zapisz dane do pliku.
15
16     method data readData():data is
17         // Wczytaj dane z pliku.
18
19 // Bazowa klasa dekorator ma taki sam interfejs jak inne
20 // komponenty. Głównym celem tej klasy jest zdefiniowanie
21 // interfejsu, który będzie opakowywał wszystkie konkretne
22 // dekoratory. Domyślona implementacja kodu opakowującego może
23 // zawierać pole służące przechowywaniu opakowanego komponentu
24 // oraz narzędzia służące jego inicjalizacji.
25 class DataSourceDecorator implements DataSource is
```

```
26 protected field wrappee: DataSource
27
28 constructor DataSourceDecorator(source: DataSource) is
29     wrappee = source
30
31 // Dekorator bazowy po prostu deleguje całą pracę
32 // opakowanemu komponentowi. Dodatkową funkcjonalność
33 // można dodać w formie kolejnych konkretnych dekoratorów.
34 method writeData(data) is
35     wrappee.writeData(data)
36
37 // Konkretni dekoratorzy mogą wywoływać implementacje
38 // działania z nadklasy zamiast bezpośrednio z opakowanego
39 // obiektu. To podejście upraszcza rozszerzanie klas
40 // dekoratorów.
41 method readData():data is
42     return wrappee.readData()
43
44 // Konkretne dekoratory wywołują metody opakowanego obiektu,
45 // ale mogą wzbogacać ich funkcjonalność. Dekoratory mogą
46 // wykonywać swoje działania albo przed, albo po wywołaniu
47 // metody opakowanego obiektu.
48 class EncryptionDecorator extends DataSourceDecorator is
49     method writeData(data) is
50         // 1. Zaszyfruj przekazane dane.
51         // 2. Przekaż zaszyfrowane dane metodzie writeData
52         // obiektu opakowanego.
53
54     method readData():data is
55         // 1. Pobierz dane od metody readData obiektu
56         // opakowanego.
57         // 2. Spróbuj odszyfrować dane jeśli są zaszyfrowane.
```

```
58     // 3. Zwróć wynik.
59
60 // Można opakowywać obiekty wieloma warstwami dekoratorów.
61 class CompressionDecorator extends DataSourceDecorator is
62     method writeData(data) is
63         // 1. Skompresuj przekazane dane.
64         // 2. Przekąż skompresowane dane metodzie writeData
65         // opakowywanego obiektu.
66
67     method readData():data is
68         // 1. Pobierz dane od metody readData opakowywanego
69         // obiektu.
70         // 2. Spróbuj je zdekompresować jeśli są skompresowane.
71         // 3. Zwróć wynik.
72
73
74 // Opcja 1. Prosty przykład zestawu dekoratorów.
75 class Application is
76     method dumbUsageExample() is
77         source = new FileDataSource("somefile.dat")
78         source.writeData(salaryRecords)
79         // Docelowy plik wypełniono danymi w formie otwartego
80         // tekstu.
81
82         source = new CompressionDecorator(source)
83         source.writeData(salaryRecords)
84         // Plik docelowy wypełniono skompresowanymi danymi.
85
86         source = new EncryptionDecorator(source)
87         // Zmienna source zawiera teraz:
88         // Szyfrowanie > Kompresja > FileDataSource
89         source.writeData(salaryRecords)
```

```
90      // Plik zapisano zaszyfrowanymi i skompresowanymi
91      // danymi.
92
93
94  // Opcja 2. Kod kliencki korzystający z zewnętrznego źródła
95  // danych. Obiekty SalaryManager nie znajdują szczegółów
96  // magazynowania danych. Pracuję z już skonfigurowanym źródłem
97  // danych które otrzymały od konfiguratora aplikacji.
98 class SalaryManager is
99   field source: DataSource
100
101  constructor SalaryManager(source: DataSource) { ... }
102
103  method load() is
104    return source.readData()
105
106  method save() is
107    source.writeData(salaryRecords)
108  // ...Inne przydatne metody...
109
110
111 // W aplikacji można złożyć różne stosy dekoratorów w trakcie
112 // działania programu – zależnie od konfiguracji lub środowiska
113 // uruchomieniowego.
114 class ApplicationConfigurator is
115   method configurationExample() is
116     source = new FileDataSource("salary.dat")
117     if (enabledEncryption)
118       source = new EncryptionDecorator(source)
119     if (enabledCompression)
120       source = new CompressionDecorator(source)
121
```

```
122     logger = new SalaryManager(source)
123     salary = logger.load()
124     // ...
```

## 💡 Zastosowanie

- ⚡ **Stosuj wzorzec Dekorator gdy chcesz przypisywać dodatkowe obowiązki obiektom w trakcie działania programu, bez psucia kodu, który z tych obiektów korzysta.**
- ⚡ Dekorator pozwala ustrukturyzować logikę biznesową w formie warstw, tworząc dekorator dla każdej warstwy i składać obiekty z różnymi kombinacjami tej logiki w czasie działania programu. Kod klienta może traktować wszystkie obiekty w taki sam sposób, ponieważ wszystkie są zgodne pod względem wspólnego interfejsu.
- ⚡ **Stosuj ten wzorzec gdy rozszerzenie zakresu obowiązków obiektu za pomocą dziedziczenia byłoby niepraktyczne, lub niemożliwe.**
- ⚡ Wiele języków programowania posiada słowo kluczowe `final`, za pomocą którego uniemożliwia się dalsze rozszerzanie klasy. W przypadku klasy finalnej, jedynym sposobem na ponowne wykorzystanie istniejącego zachowania jest opakowanie jej nakładkami swojego autorstwa – zgodnie ze wzorcem Dekorator.

## Jak zaimplementować

1. Upewnij się, że Twoja domena biznesowa może zostać przedstawiona w formie podstawowego komponentu z nałożonymi nań wieloma opcjonalnymi warstwami.
2. Ustal jakie metody są wspólne zarówno dla podstawowego komponentu, jak i warstw opcjonalnych. Stwórz interfejs komponentu i zadeklaruj tam owe wspólne metody.
3. Stwórz klasę konkretnego komponentu i zdefiniuj w niej podstawowe zachowanie.
4. Stwórz bazową klasę dekoratora. Powinna ona zawierać pole do przechowywania odniesienia do opakowywanego obiektu. Pole takie powinno być zadeklarowane jako typ interfejsu komponenta, aby umożliwić wiązanie z konkretnymi komponentami oraz dekoratorami. Dekorator bazowy musi delegować pracę obiekowi opakowywanemu.
5. Upewnij się, że wszystkie klasy implementują interfejs komponentu.
6. Stwórz konkretne dekoratory poprzez rozszerzanie dekoratora bazowego. Konkretny dekorator musi wykonywać swoje zadania przed lub po wywołaniu metody rodzica (który zawsze deleguje opakowywanemu obiekowi).

7. Kod kliencki musi być odpowiedzialny za tworzenie dekoratorów oraz składanie ich wedle swoich potrzeb.

## ⚠️ Zalety i wady

- ✓ Można rozszerzać zachowanie obiektu bez tworzenia podklasy.
- ✓ Można dodawać lub usuwać obowiązki obiektu w trakcie działania programu.
- ✓ Możliwe jest łączenie wielu zachowań poprzez nałożenie wielu dekoratorów na obiekt.
- ✓ *Zasada pojedynczej odpowiedzialności.* Można podzielić klasę monolityczną, która implementuje wiele wariantów zachowań, na mniejsze klasy.
- ✗ Zabranie jednej konkretnej nakładki ze środka stosu nakładek jest trudne.
- ✗ Trudno jest zaimplementować dekorator w taki sposób, aby jego zachowanie nie zależało od kolejności ułożenia nakładek na stosie.
- ✗ Kod wstępnie konfigurujący warstwy może wyglądać brzydko.

## ↔ Powiązania z innymi wzorcami

- **Adapter** zmienia interfejs istniejącego obiektu, zaś **Dekorator** rozszerza go bez zmiany interfejsu. Ponadto *Dekorator* wspiera rekursywną kompozycję, co nie jest możliwe gdy zastosuje się *Adapter*.

- **Adapter** wyposaża “opakowywany” obiekt w inny interfejs, **Pet-nomocnik** w taki sam, zaś **Dekorator** wprowadza rozszerzony interfejs.
- **Łańcuch zobowiązań** i **Dekorator** mają bardzo podobne struktury klas. Oba wzorce bazują na rekursywnej kompozycji w celu przekazania obowiązku wykonania przez ciąg obiektów. Istnieją jednak kluczowe różnice.

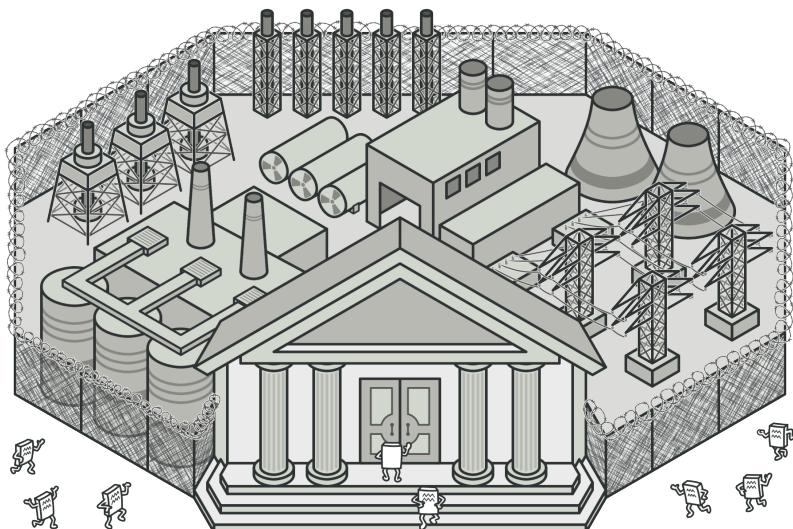
Obsługujący *Łańcucha zobowiązań* mogą wykonywać działania niezależnie od siebie. Mogą również zatrzymać dalsze przekazywanie żądania na dowolnym etapie. Z drugiej strony, różne *Dekoratory* mogą rozszerzać obowiązki obiektu zachowując zgodność z interfejsem bazowym. Dodatkowo, dekoratory nie mają możliwości przerwania przepływu żądania.

- **Kompozyt** i **Dekorator** mają podobne diagramy struktur ponieważ oba bazują na rekursywnej kompozycji w celu zorganizowania nieokreślonej liczby obiektów.

*Dekorator* przypomina *Kompozyt*, ale posiada tylko jeden element podrzędny. Ponadto, kolejną różnicą jest to, że *Dekorator* przypisuje dodatkowe obowiązki opakowywanemu obiekowi, zaś *Kompozyt* jedynie “sumuje” wyniki otrzymane od elementów podrzędnych.

Wzorce mogą też współpracować: *Dekorator* może służyć rozszerzeniu zachowania określonego obiektu w drzewie *Kompozytowym*

- Projekty intensywnie korzystające ze wzorców **Kompozyt** i **Dekorator** mogą skorzystać również na zastosowaniu **Prototypu**. Zastosowanie tego wzorca pozwala klonować złożone struktury zamiast konstruować je ponownie od zera.
- **Dekorator** pozwala zmienić otoczkę obiektu, zaś **Strategia** jej wnętrze.
- **Dekorator** i **Pełnomocnik** mają podobne struktury, ale inne cele. Oba wzorce bazują na zasadzie kompozycji – jeden obiekt deleguje część zadań innemu. **Pełnomocnik** dodatkowo zarządza cyklem życia obiektu udostępniającego jakąś usługę, zaś komponowanie *Dekoratorów* leży w gestii klienta.



# FASADA

*Znany też jako: Facade*

**Fasada** jest strukturalnym wzorcem projektowym, który wyposaża bibliotekę, framework lub inny złożony zestaw klas w uproszczony interfejs.

## Problem

Wyobraź sobie, że twój kod musi współdziałać z szerokim zestawem obiektów należących do jakiejś skomplikowanej biblioteki lub frameworku. Zazwyczaj należy zainicjalizować wszystkie te obiekty, śledzić zależności, wywoływać metody w odpowiedniej kolejności i tak dalej.

W rezultacie doszłoby do ścisłego sprzęgnięcia logiki biznesowej twoich klas ze szczegółami implementacji klas innych dostawców, co utrudniłoby utrzymanie i zrozumienie kodu.

## Rozwiązanie

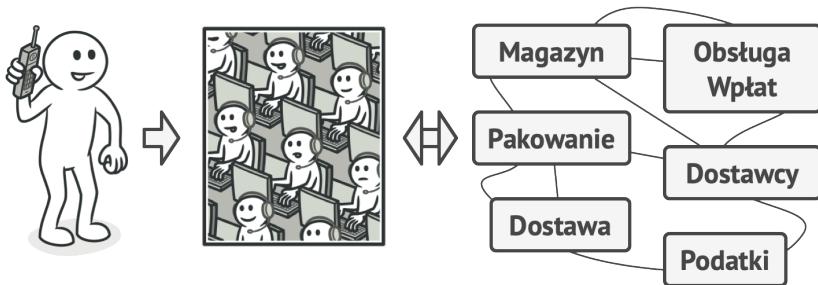
Fasada to klasa stanowiąca prosty interfejs dla złożonego podsystemu, zawierającego mnóstwo ruchomych części. Fasada może dawać ograniczoną funkcjonalność, w porównaniu z korzystaniem z elementów podsystemu bezpośrednio, ale za to eksponuje tylko te możliwości, których klient naprawdę potrzebuje.

Stworzenie fasady jest wygodnym sposobem integracji twej aplikacji ze skomplikowaną biblioteką posiadającą wiele funkcji, gdy potrzebujesz tylko wąskiego zakresu jej funkcji.

Na przykład, aplikacja która publikuje krótkie zabawne filmy z kotami w mediach społecznościowych, może korzystać z profesjonalnej biblioteki konwersji wideo. Z całej biblioteki potrzebuje jednak tylko klasy z jedną metodą –

`zakoduj(nazwa_pliku, format)`. Po stworzeniu takiej klasy i podłączeniu jej do biblioteki konwersji wideo otrzymamy naszą pierwszą fasadę.

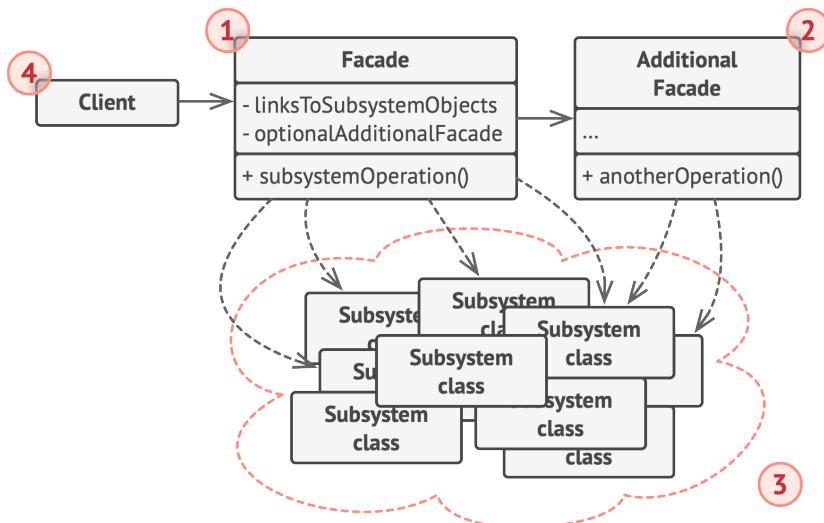
## Analoga do prawdziwego życia



*Składanie zamówienia przez telefon.*

Gdy dzwonisz do sklepu aby złożyć zamówienie, biuro jest twoją fasadą dla wszystkich usług i oddziałów tego sklepu. Pracownik sklepu, czy automat zgłoszeniowy, stanowią prosty interfejs głosowy do systemu zamawiania, płacenia i różnych usług dostawczych.

## Struktura



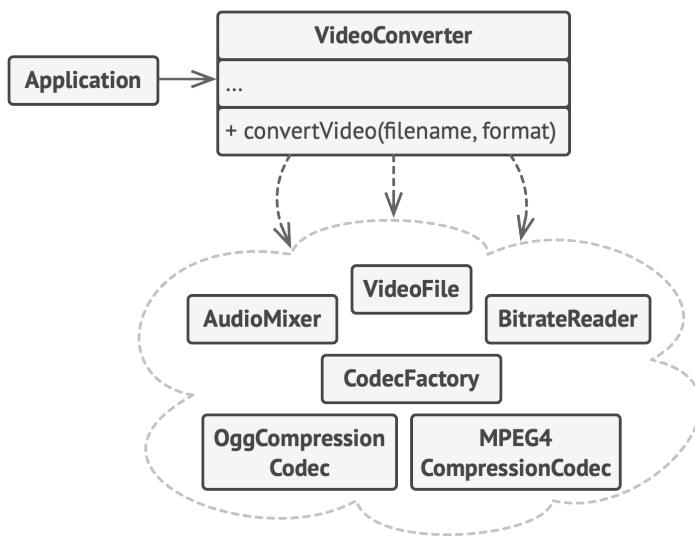
1. **Fasada** daje wygodny dostęp do pewnego zakresu funkcjonalności podsystemu. Wie dokąd przekierować żądanie klienta i jak pokierować wszystkimi szczegółami.
2. Można stworzyć klasę **Dodatkowa Fasada**, by zapobiec zaśmieceniu pojedynczej fasady niepotrzebnymi funkcjami, które ponownie ograniczyłyby prostotę używania. Dodatkowe fasady mogą być wykorzystane zarówno przez klienta, jak i inne fasady.
3. **Złożony podsystem** składa się z wielu różnych obiektów. Aby mogły one wszystkie wykonać coś pożytecznego, trzeba zagłębić się w szczegóły implementacyjne podsystemu – inicjalizację obiektów we właściwej kolejności i przekazanie im danych w odpowiednim formacie.

Klasy podsystemu nie są świadome istnienia fasady. Działają wewnątrz systemu i współpracują ze sobą bezpośrednio.

4. **Klient** korzysta z fasady zamiast wywoływać obiekty podsystemu bezpośrednio.

## # Pseudokod

W poniższym przykładzie, wzorzec **Fasada** upraszcza interakcję ze złożonym frameworkm do konwersji wideo.



*Przykład izolowania wielu zależności w pojedynczej klasie fasada.*

Zamiast wiązać bezpośrednio kod z wieloma klasami składowymi frameworku, tworzymy klasę fasady która hermetyzuje funkcjonalność i ukrywa ją przed resztą kodu. Ta struktura pozwala też zminimalizować wysiłek związany z aktualizacją fra-

meworku do nowszej wersji lub wręcz wymiany na inny. Jedyna rzecz, jaką trzeba będzie wówczas zmienić, to implementacja metod fasady.

```
1 // Oto klasy złożonego frameworku od zewnętrznego dostawcy
2 // służącego konwersji wideo. Nie mamy wpływu na ten kod, więc
3 // nie możemy go uproszczyć.
4
5 class VideoFile
6 // ...
7
8 class OggCompressionCodec
9 // ...
10
11 class MPEG4CompressionCodec
12 // ...
13
14 class CodecFactory
15 // ...
16
17 class BitrateReader
18 // ...
19
20 class AudioMixer
21 // ...
22
23
24 // Tworzymy klasę fasada która ukryje złożoność frameworku za
25 // prostym interfejsem. Takie podejście jest kompromisem
26 // pomiędzy funkcjonalnością, a łatwością użycia.
27 class VideoConverter is
```

```

28 method convert(filename, format):File is
29     file = new VideoFile(filename)
30     sourceCodec = new CodecFactory.extract(file)
31     if (format == "mp4")
32         destinationCodec = new MPEG4CompressionCodec()
33     else
34         destinationCodec = new OggCompressionCodec()
35     buffer = BitrateReader.read(filename, sourceCodec)
36     result = BitrateReader.convert(buffer, destinationCodec)
37     result = (new AudioMixer()).fix(result)
38     return new File(result)
39
40 // Klasa aplikacji nie są zależne od masy klas wchodzących w
41 // skład frameworku. Ponadto, w przypadku konieczności wymiany
42 // frameworku na inny, trzeba będzie zmodyfikować jedynie klasę
43 // fasada.
44 class Application is
45     method main() is
46         convertor = new VideoConverter()
47         mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
48         mp4.save()

```

## Zastosowanie

-  **Użyj wzorca Fasada gdy potrzebujesz ograniczonego, ale łatwego w użyciu interfejsu do złożonego podsystemu.**
-  Zazwyczaj podsystemy stają się coraz bardziej złożone z biegiem czasu. Nawet stosowanie wzorców projektowych prowadzi do przyrostu liczby klas. Podsystem może wprawdzie

stać się elastyczniejszym i łatwiejszym do ponownego użycia w różnych kontekstach, ale ilość kodu konfigurującego i przygotowawczego wymagana od klienta wzrośnie. Fasada jest sposobem rozwiązywania tego problemu poprzez udostępnienie skrótów do najczęściej używanych funkcji podsystemu, zgodnie z wymaganiami klienta.

 **Stosuj Fasadę gdy chcesz ustrukturyzować podsystem w warstwy.**

 Twórz fasady by zdefiniować punkty wejścia do każdego poziomu podsystemu. Możesz ograniczyć sprzęgnięcie pomiędzy wieloma podsystemami, zmuszając je do komunikacji ze sobą wyłącznie poprzez fasady.

Dla przykładu, wróćmy do naszego framework'u konwersji wideo. Można go podzielić na dwie warstwy: związaną z obrazem i związaną z dźwiękiem. Dla każdej z warstw możesz utworzyć fasadę i sprawić, by klasy każdej warstwy komunikowały się ze sobą za pośrednictwem tych fasad. Takie podejście przypomina wzorzec **Mediator**.

## Jak zaimplementować

1. Sprawdź czy możliwe jest utworzenie prostszego interfejsu w porównaniu z tym, jaki dostarcza istniejący podsystem. Jeśli planowany interfejs sprawiłby, że kod kliencki będzie niezależny od wielu klas podsystemu – jesteś na właściwej drodze.

2. Zadeklaruj i zaimplementuj ten interfejs jako nową klasę fasada. Fasada powinna przekierowywać wywołania pochodzące od kodu klienckiego do stosownych obiektów podsystemu. Ponadto, fasada powinna być odpowiedzialna za inicjalizację podsystemu i zarządzanie jego dalszym cyklem życia, chyba, że kod kliencki już pełni tę rolę.
3. Aby w pełni skorzystać na tym wzorcu, spraw, aby kod kliencki komunikował się z podsystemem wyłącznie poprzez fasadę. W ten sposób kod kliencki pozostanie chroniony przed zmianami dokonywanymi w kodzie podsystemu. Na przykład, aktualizacja podsystemu będzie wymagała jedynie zmian w kodzie fasady.
4. Jeśli fasada stanie się **zbyt duża**, rozważ ekstrakcję części jej obowiązków do nowej, doskonalszej klasy fasady.

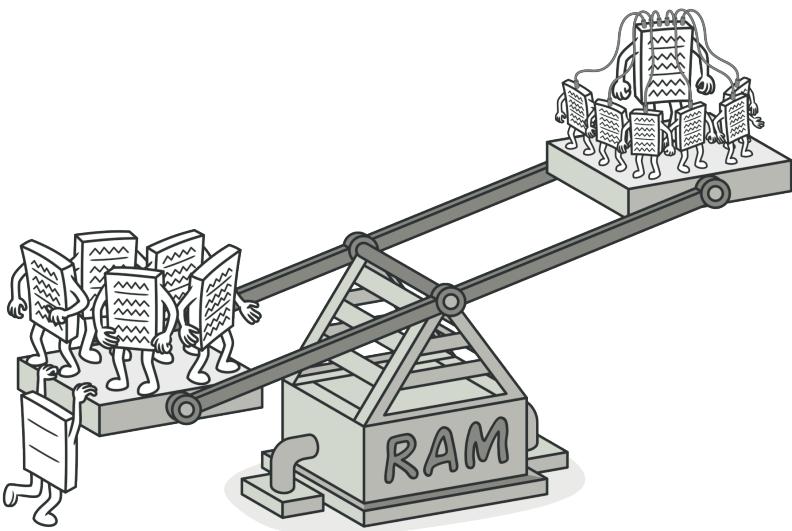
## ΔΔ Zalety i wady

- ✓ Można odizolować kod od złożoności podsystemu.
- ✗ Fasada może stać się **boskim obiektem** sprzężonym ze wszystkimi klasami aplikacji.

## ↔ Powiązania z innymi wzorcami

- **Fasada** definiuje nowy interfejs istniejącym obiektom, zaś **Adapter** zakłada zwiększenie użyteczności zastanego interfejsu. *Adapter* na ogół opakowuje pojedynczy obiekt, zaś *Fasada* obejmuje cały podsystem obiektów.

- **Fabryka abstrakcyjna** może służyć jako alternatywa do **Fasady** gdy jedyne co chcesz zrobić, to ukrycie przed kodem klienckim procesu tworzenia obiektów podsystemu.
- **Pyłek** przedstawia sposób na stworzenie wielkiej liczby małych obiektów, zaś **Fasada** na stworzenie pojedynczego obiektu reprezentującego cały podsystem.
- **Fasada** i **Mediator** mają podobne zadania: służą zorganizowaniu współpracy pomiędzy wieloma ściśle sprzęgniętymi klasami.
  - *Fasada* definiuje uproszczony interfejs podsystemu obiektów, ale nie wprowadza nowej funkcjonalności. Podsystem jest nieświadomy istnienia fasady. Obiekty w obrębie podsystemu mogą komunikować się bezpośrednio.
  - *Mediator* centralizuje komunikację pomiędzy komponentami podsystemu. Komponenty wiedzą tylko o obiekcie mediator i nie komunikują się ze sobą bezpośrednio.
- Klasa **Fasada** może często być przekształcona w **Singleton**, ponieważ pojedynczy obiekt fasady jest w większości przypadków wystarczający.
- **Fasada** przypomina wzorzec **Pełnomocnik** w tym sensie, że oba buforują złożony podmiot i inicjalizują go samodzielnie. W przeciwieństwie do *Fasady*, *Pełnomocnik* ma taki sam interfejs jak obiekt udostępniający usługę który ją reprezentuje, co czyni je wymienialnymi.



# PYŁEK

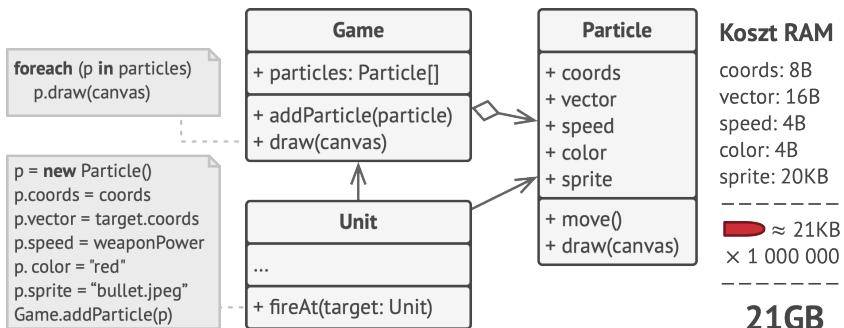
Znany też jako: *Cache, Flyweight*

**Pyłek** jest strukturalnym wzorcem projektowym pozwalającym zmieścić więcej obiektów w danej przestrzeni pamięci RAM poprzez współdzielenie części opisu ich stanów.

## (:() Problem

Aby rozerwać się nieco po pracy, postanawiasz stworzyć prostą grę komputerową: gracze poruszają się po mapie i strzelają do siebie. Chcesz zaimplementować realistyczny system cząstek i uczynić z niego wyróżniającą się zaletę gry. Niech wielkie ilości kul, rakiet i odłamków fruwają po całej mapie, dostarczając ekscytującej rozrywki.

Ukończywszy pracę, wykonujesz ostatni commit, komplujesz grę i wysydasz znajomemu na próbę. Chociaż gra chodzi płynnie na twoim komputerze, kolega nie może dłużej pograć. Po paru minutach gra się wiesza. Po wielogodzinnym poszukiwaniu przyczyn w dziennikach debugowych, zauważasz, że grze zabrakło pamięci RAM. Okazało się bowiem, że komputer kolegi jest słabszy niż twój i dlatego problem objawił się u niego tak szybko.

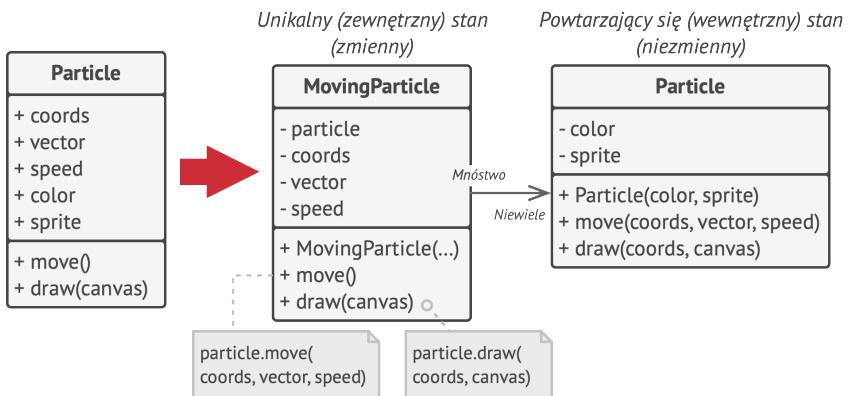


Źródłem problemu był system cząstek. Każda cząstka, jak kula, rakieta czy odłamek, reprezentowany był jako osobny obiekt zawierający mnóstwo danych. W którymś momencie, w czasie

renderowania strzelaniny, nowo utworzone cząstki nie mieściły się w pamięci operacyjnej i gra kończyła działanie.

## Rozwiążanie

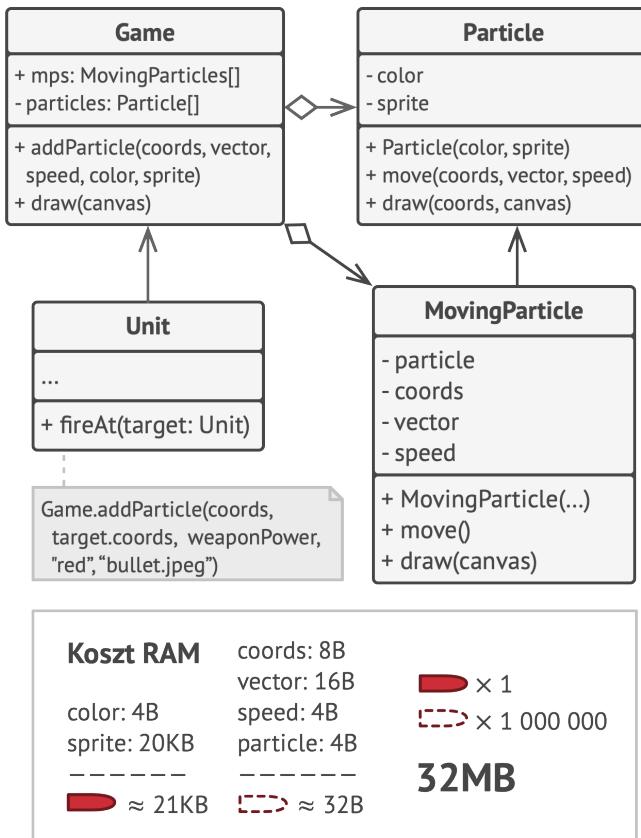
Przy dokładniejszej inspekcji klasy `Cząstka` zauważamy, że kolor i sprite każdej cząstki zużywają znacznie więcej pamięci, niż inne pola obiektu. Co gorsza, te dwa pola przechowują nienormalne identyczne dane we wszystkich cząstkach. Na przykład – wszystkie kule mają tę samą barwę i sprite.



Inne elementy opisujące stan cząstki, jak współrzędne, wektor ruchu i prędkość są unikalne dla każdej z nich. Bo przecież te wartości ulegają ciągłej zmianie. Dane te reprezentują wciąż zmieniający się kontekst, w jakim cząstka się znajduje, zaś kolor i sprite pozostają jednakowe dla każdej z nich.

Dane niezmienne, opisujące obiekt, nazywa się *stanem wewnętrznym*. Opisany jest on w każdym z obiektów, zaś inne obiekty mają do niego tylko prawo odczytu. Reszta stanu

obiektu, często zmienianym “z zewnątrz” przez inne obiekty, zwana jest *stanem zewnętrznym*.



Wzorzec Pyłek proponuje rezygnację z przechowywania stanu zewnętrznego w obiekcie. Zamiast tego należy przekazywać ten stan konkretnym metodom które go potrzebują. Tylko stan wewnętrzny powinien pozostać zapisany w obrębie obiektu, pozwalając na użycie go ponownie w innych kontekstach. Dzięki temu potrzebujemy mniej tych obiektów, ponieważ róż-

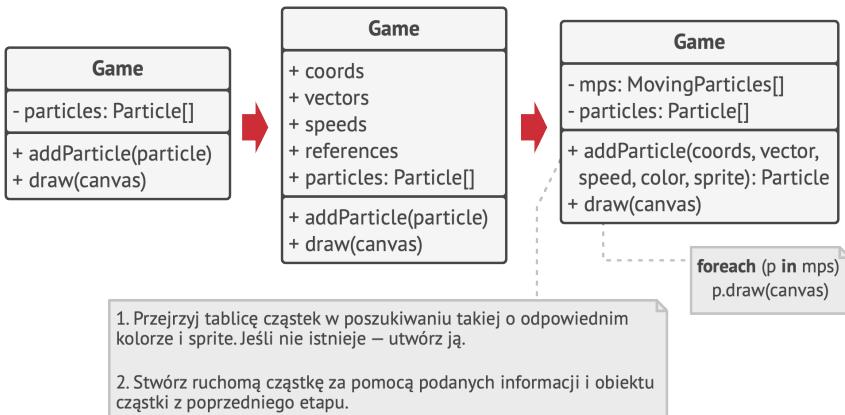
nią się tylko pod względem wewnętrznego stanu, którego możliwych kombinacji jest znacznie mniej.

Wróćmy do naszej gry. Zakładając, że wyekstrahowaliśmy stan zewnętrzny z naszej klasy-cząstki, wystarczą zaledwie 3 obiekty, aby reprezentować wszystkie cząstki w grze: kulę, rakietę i odłamek. Jak zapewne już się domyślasz, obiekt przechowujący tylko stan wewnętrzny nazywa się Pyłkiem.

### Przechowywanie danych zewnętrznych

Dokąd przenieść zewnętrzny stan? Jakaś klasa powinna go przechowywać, prawda? W większości przypadków, przenosi się go do obiektu kontenerowego, który agreguje obiekty zanim zastosujemy wzorzec.

W naszym przypadku to główny obiekt `Gra` przechowuje wszystkie cząstki w polu `cząstki`. By przenieść zewnętrzne stany do tej klasy, musisz stworzyć wiele pól tablicowych do przechowywania współrzędnych, wektorów i prędkości każdej cząstki. Ale to nie wszystko – potrzebujesz jeszcze jednej tablicy w celu przechowania referencji do konkretnego pyłku reprezentującego cząstkę. Te dwie tablice muszą być zsynchronizowane, aby można było pobrać wszystkie dane cząstki stosując ten sam indeks.



Bardziej eleganckim rozwiązaniem jest utworzenie osobnej klasy kontekstowej która przechowuje zewnętrzny stan wraz z odniesieniem do obiektu pyłek. W takiej sytuacji potrzebna jest tylko jedna tablica w klasie kontenerowej.

Ale chwileczkę! Czy czasem nie będzie nam potrzebne tyle takich obiektów kontekstowych, ile mieliśmy na samym początku? W zasadzie tak, ale te obiekty są dużo mniejsze niż wcześniej. Pola zajmujące najwięcej pamięci przeniesiono do kilku obiektów pyłków. Teraz tysiąc małych obiektów kontekstowych może wykorzystać ponownie pojedynczy, duży obiekt pyłek, zamiast przechowywać tysiąc kopii ich danych.

## Pyłek a niezmienność

Skoro ten sam obiekt pyłek może być wykorzystany w różnych kontekstach, musisz się upewnić, że jego stan nie może być zmieniony. Pyłek powinien inicjalizować swój stan tylko jednorazowo, za pośrednictwem parametrów konstruktora. Nie po-

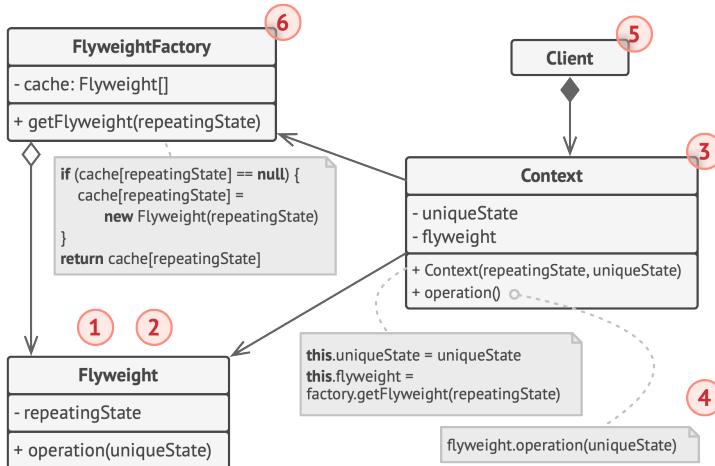
winien eksponować innym obiektom żadnych setterów ani pól publicznych.

## Fabryka pyłków

Stworzenie metody wytwórczej zarządzającej pulą istniejących obiektów pyłków daje nam wygodniejszy dostęp do różnych częstek. Metoda przyjmuje pożądany przez klienta opis stanu wewnętrznego, poszukuje istniejącego obiektu o takim stanie i go zwraca. Jeśli go nie znajdzie – tworzy nowy i dodaje go do puli.

Istnieje wiele miejsc, gdzie można umieścić taką metodę. Najbardziej oczywistym jest kontener pyłków. Innym sposobem jest stworzenie nowej klasy fabrycznej. Można też uczynić metodę wytwórczą statyczną i umieścić ją w faktycznej klasie pyłek.

# Struktura



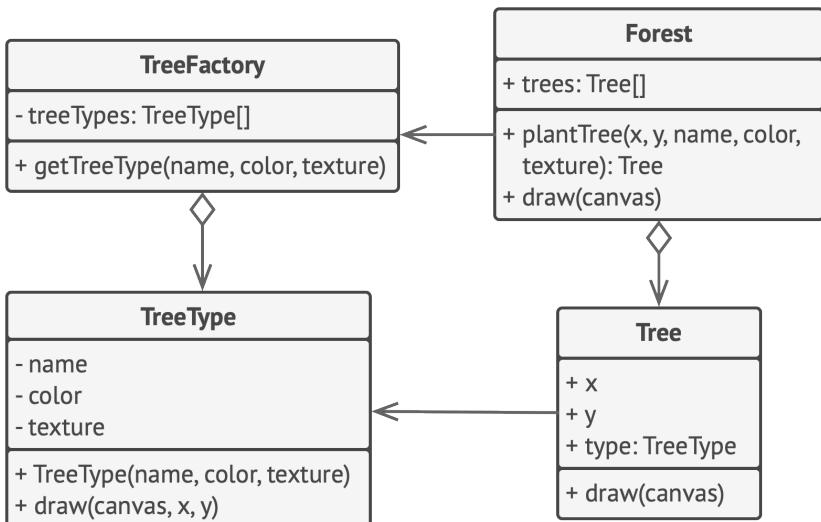
1. Wzorzec Pyłek jest jedynie optymalizacją. Przed zastosowaniem go, upewnij się, że twój program ma potencjalny problem z wyczerpywaniem pamięci RAM wskutek istnienia jednocześnie wielkiej liczby podobnych obiektów. Odpowiedz sobie na pytanie, czy nie da się takiego problemu rozwiązać w inny sposób.
2. Klasa **Pyłek** zawiera tę porcję stanu pierwotnego obiektu, która może być współdzielona pomiędzy wieloma instancjami. Ten sam obiekt-pyłek może być wykorzystany w wielu kontekstach. Stan przechowywany w pyłku nazywa się *wewnętrzny*. Stan przekazywany metodą pyłka to dane *zewnętrzne*.
3. Klasa **Kontekst** zawiera opis zewnętrznego stanu, unikalny dla każdego z pierwotnych obiektów. Gdy kontekst skojarzy się z

jednym z obiektów-pyłków, otrzymuje się reprezentację pełnego stanu pierwotnego obiektu.

4. Zazwyczaj obowiązki pierwotnego obiektu pozostają w klasie pyłek. W takim przypadku, w momencie wywołania metody pyłka, trzeba przekazać jej również odpowiednie elementy stanu zewnętrznego. Z drugiej strony, obowiązki można przemieścić do klasy kontekstowej, która korzysta ze skojarzonego pyłku tylko jako obiektu danych.
5. **Klient** oblicza lub przechowuje zewnętrzny stan pyłków. Z punktu widzenia klienta, pyłek to obiekt szablonowy który może być skonfigurowany w trakcie działania programu poprzez przekazanie jakichś danych kontekstowych w charakterze parametrów jego metod.
6. **Fabryka Pyłków** zarządza pulą istniejących pyłków. Dzięki fabryce, klienci nie tworzą pyłków w sposób bezpośredni. Zamiast tego wywołują fabrykę, przekazując jej fragmenty danych o wewnętrznym stanie pożądanego pyłka. Fabryka przegląda poprzednio stworzone pyłki i albo zwraca odpowiedni, albo go tworzy.

## # Pseudokod

W poniższym przykładzie, wzorzec **Pyłek** pomaga zredukować zużycie pamięci podczas renderowania milionów obiektów-drzew na ekranie.



Działając według wzorca, ekstrahuje się powtarzający, wewnętrzny stan z głównej klasy Drzewo i przenosi do klasy pyłek o nazwie TypDrzewa.

Teraz, zamiast przechowywać te same dane w wielu obiektach, znajdują się one tylko w kilku obiektach-pyłkach, skojarzonych ze stosownymi obiektami Drzewo które służą za kontekst. Kod klienta tworzy nowe drzewa za pośrednictwem fabryki pyłków, która hermetyzuje złożoność poszukiwania odpowiedniego obiektu i jego ewentualnego ponownego użycia.

```

1 // Klasa pyłek zawiera część stanu drzewa. Pola te przechowują
2 // wartości które są unikalne dla każdego drzewa. Przykładowo
3 // nie znajdziemy tu współrzędnych drzewa, ale teksturę oraz
4 // wspólne barwy – owszem. Ponieważ te dane są zazwyczaj
5 // WIELKIE, zmarnowałibyśmy bardzo dużo pamięci operacyjnej,
  
```

```
6 // przechowując ich kopie w obrębie każdego z obiektów-drzew.  
7 // Ekstrahujemy więc tekstury, barwy i inne powtarzające się  
8 // dane do odrębnego obiektu. Wszystkie drzewa będą posiadać  
9 // odniesienie do nowego obiektu.  
10 class TreeType is  
11     field name  
12     field color  
13     field texture  
14     constructor TreeType(name, color, texture) { ... }  
15     method draw(canvas, x, y) is  
16         // 1. Utwórz mapę bitową o danym typie, kolorze i  
17         // tekstuze.  
18         // 2. Narysuj mapę bitową na ekranie w punkcie o  
19         // współrzędnych X i Y.  
20  
21     // Fabryka pyłków podejmuje decyzję o ponownym użyciu  
22     // istniejącego obiektu-pyłka lub utworzeniu nowego.  
23 class TreeFactory is  
24     static field treeTypes: collection of tree types  
25     static method getTreeType(name, color, texture) is  
26         type = treeTypes.find(name, color, texture)  
27         if (type == null)  
28             type = new TreeType(name, color, texture)  
29             treeTypes.add(type)  
30         return type  
31  
32     // Obiekt-kontekst zawiera zewnętrzne elementy stanu drzewa.  
33     // Aplikacja może stworzyć miliardy drzew, bo są one bardzo  
34     // małe: opisują je dwie liczby całkowite oznaczające  
35     // współrzędne i jedno pole przechowujące odniesienie do obiektu  
36     // zawierającego opis stanu zewnętrznego.  
37 class Tree is
```

```

38   field x,y
39   field type: TreeType
40   constructor Tree(x, y, type) { ... }
41   method draw(canvas) is
42     type.draw(canvas, this.x, this.y)
43
44 // Klasa Tree i Forest są klientami pyłku. Możesz je połączyć,
45 // jeśli nie zamierzasz dalej rozwijać klasy Tree.
46 class Forest is
47   field trees: collection of Trees
48
49   method plantTree(x, y, name, color, texture) is
50     type = TreeFactory.getType(name, color, texture)
51     tree = new Tree(x, y, type)
52     trees.add(tree)
53
54   method draw(canvas) is
55     foreach (tree in trees) do
56       tree.draw(canvas)

```

## Zastosowanie

-  **Stosuj wzorzec Pyłek gdy twój program musi pracować z wielką ilością obiektów, które ledwo mieścią się w dostępnej pamięci RAM.**
-  Zyski z wprowadzenia tego wzorca zależą od tego jak i gdzie się go zastosuje. Największy pożytek uzyskuje się gdy:
  - aplikacja musi tworzyć wielką ilość podobnych obiektów,

- powyższa sytuacja poważnie obciąża dostępną pamięć RAM urządzenia,
- obiekty zawierają wielokrotnie powtarzające się opisy stanów, dające się wyekstrahować i pozwoli się na współdzielienie ich pomiędzy wieloma obiektami.

## Jak zaimplementować

1. Podziel na dwie części pola klasy z których powstanie pyłek:
  - stan wewnętrzny: pola, które przechowują niezmienne dane, powtarzające się w wielu obiektach
  - stan zewnętrzny: pola, które przechowują dane kontekstowe, unikalne dla każdego obiektu
2. Pozostaw pola reprezentujące wewnętrzny stan w klasie, ale upewnij się, że nie mogą być zmieniane. Powinny one przyjmować swój stan początkowy wyłącznie w konstruktorze.
3. Przejrzyj metody korzystające z pól zewnętrznego stanu. Dla każdego pola użytego w metodzie, dodaj nowy parametr i używaj go zamiast pola.
4. Opcjonalnie, utwórz klasę fabryczną służącą zarządzaniu pulą pyłków. Powinna ona poszukać istniejącego pyłka przed utworzeniem nowego. Gdy fabryka jest już gotowa, klienci powinni wnioskować o pyłki wyłącznie przez nią, przekazując opis stanu wewnętrznego żądanego obiektu.

5. Klient musi przechowywać lub wyliczać wartości opisujące stan zewnętrzny (kontekst), by mógł wywoływać metody obiektów-pyłków. Dla wygody, zewnętrzny stan wraz z polem odnoszącym się do pyłka można przenieść do osobnej klasy kontekstowej.

## ⚠ Zalety i wady

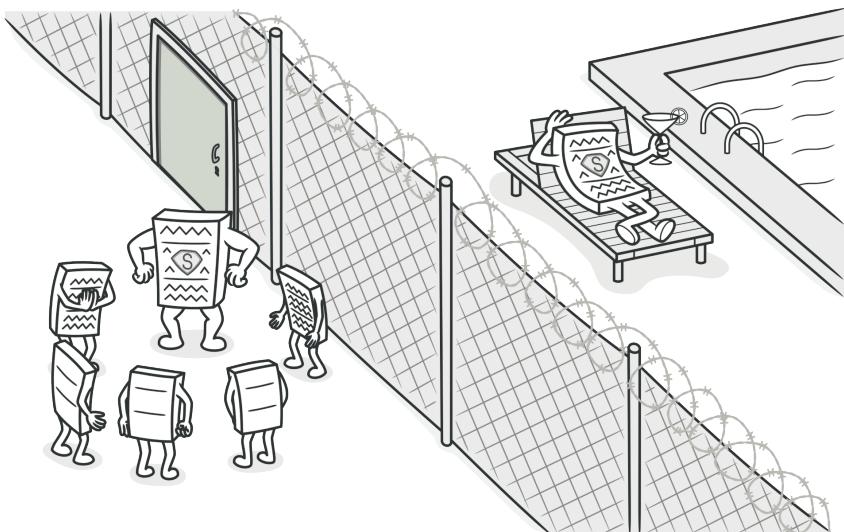
- ✓ Możesz zaoszczędzić mnóstwo pamięci RAM, o ile twój program tworzy mnóstwo podobnych obiektów.
- ✗ Może się zdarzyć, że oszczędność pamięci odbędzie się kosztem czasu procesora, gdyż część danych kontekstowych musi być wyliczana przy każdym wywołaniu metody pyłka.
- ✗ Kod staje się dużo bardziej skomplikowany. Nowi członkowie zespołu z pewnością będą się zastanawiać dlaczego stan czegoś został odseparowany.

## ↔ Powiązania z innymi wzorcami

- Węzły będące liśćmi drzewa **Kompozytowego** można zaimplementować jako **Pyłki** by zaoszczędzić nieco pamięci RAM.
- **Pyłek** przedstawia sposób na stworzenie wielkiej liczby małych obiektów, zaś **Fasada** na stworzenie pojedynczego obiektu reprezentującego cały podsystem.
- **Pyłek** mógłby przypominać **Singleton**, gdybyśmy zdołali zredukować wszystkie współdzielone stany obiektów do tylko

jednego obiektu-pyłka. Ale są jeszcze dwie fundamentalne różnice między tymi wzorcami:

1. Powinna istnieć tylko jedna instancja interfejsu Singleton, zaś instancji *Pyłka* będzie wiele, o różnym stanie wewnętrznym.
2. Obiekt *Singleton* może być zmienny. Pyłki są zaś niezmienne.



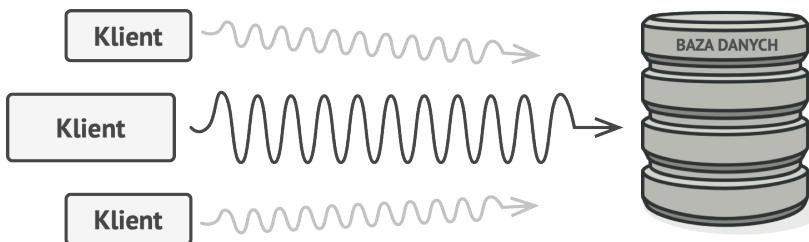
# PEŁNOMOCNIK

Znany też jako: *Proxy*

**Pełnomocnik** to strukturalny wzorzec projektowy pozwalający stworzyć obiekt zastępczy w miejsce innego obiektu. Pełnomocnik nadzoruje dostęp do pierwotnego obiektu, pozwalając na wykonanie jakieś czynności przed lub po przekazaniu do niego żądania.

## :( Problem

Po co właściwie kontrolować dostęp do obiektu? Założmy, że mamy duży obiekt, który zużywa dużo zasobów. Potrzebujesz go co jakiś czas, ale nie ciągle.



*Zapytania bazodanowe mogą być bardzo powolne.*

Moglibyśmy zastosować leniwyą inicjalizację: tworzyć ten obiekt tylko gdy staje się faktycznie potrzebny. Wszystkie jego klienci musiałyby wykonać jakiś opóźniony kod inicjalizujący. Niestety doprowadziłoby to do powielania kodu.

W idealnym świecie, chcielibyśmy umieścić ten kod w klasie obiektu, ale nie zawsze jest to możliwe. Klasa może być bowiem częścią zamkniętej biblioteki innego dostawcy.

## :D Rozwiązanie

Wzorzec Pełnomocnik zakłada stworzenie nowej klasy pośredniczącej, o takim samym interfejsie co pierwotny obiekt udostępniający usługę. Następnie aktualizujemy nasz program tak, aby przekazywał obiekt pełnomocnika wszystkim klientom pierwotne-

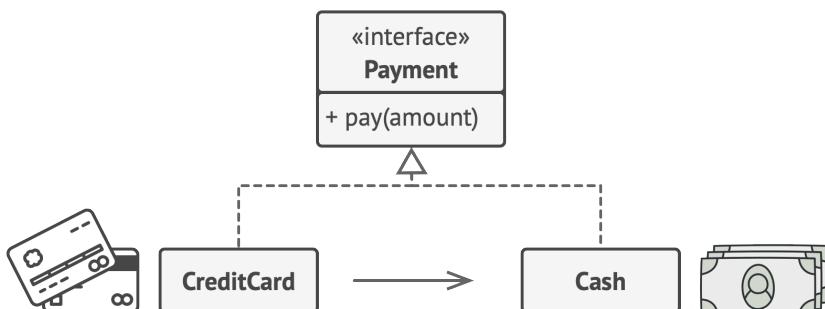
go obiektu. Otrzymawszy żądanie od klienta, pełnomocnik tworzy prawdziwy obiekt usługi i deleguje mu całą pracę.



*Pełnomocnik udaje obiekt bazodanowy. Może zająć się leniwą inicjalizacją i przechowywaniem wyników bez wiedzy klienta i samej bazy danych.*

Ale co z tego mamy? Jeśli musisz uruchomić coś albo przed, albo po głównej logice klasy, pełnomocnik pozwala uczynić to bez modyfikowania tej klasy. Ponieważ pełnomocnik implementuje ten sam interfejs, co pierwotna klasa, może być przekazany do dowolnego klienta wymagającego prawdziwego obiektu udostępniającego usługę.

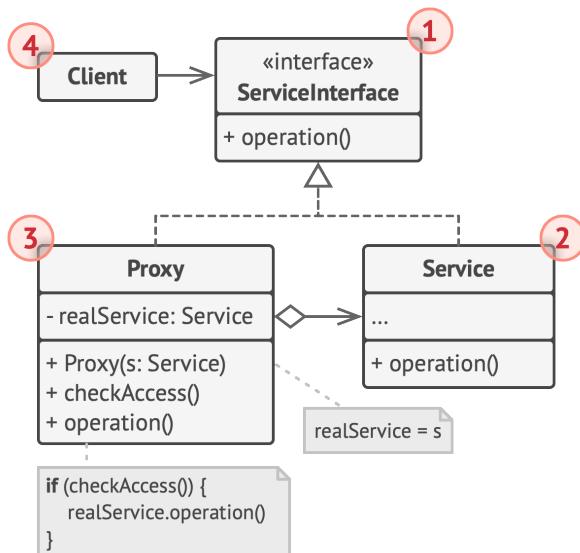
## 🚗 Analogia do prawdziwego życia



*Płacić można zarówno kartą kredytową, jak i gotówką.*

Karta kredytowa jest pełnomocnikiem konta bankowego, które z kolei jest pełnomocnikiem gotówki. Oba implementują taki sam interfejs: można za ich pomocą płacić. Klient jest zadowolony, gdyż nie musi nosić przy sobie gotówki. Sklepikarz również jest zadowolony, bo przychody z transakcji są elektronicznie dodawane do konta bankowego sklepu bez ryzyka zagubienia czy kradzieży utargu.

## STRUCTURE



1. **Interfejs Usługi** deklaruje interfejs z którym Pełnomocnik musi być zgodny, aby móc udawać obiekt usługi.
2. **Usługa** to klasa udostępniająca jakąś użyteczną logikę biznesową.

3. Klasa **Pełnomocnik** zawiera pole z odniesieniem do konkretnego obiektu udostępniającego usługę. Po wykonaniu swoich zadań (leniwa inicjalizacja, zapis w dzienniku, kontrola dostępu, przechowanie w pamięci podręcznej, itp.) Pełnomocnik przekazuje żądanie do tego obiektu.

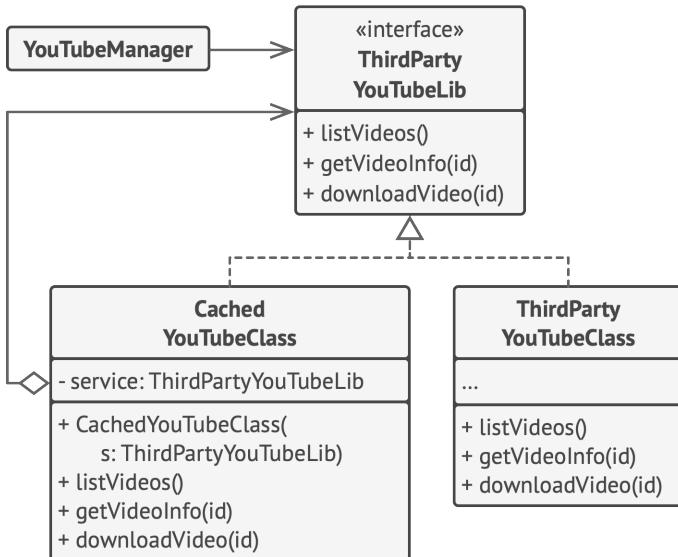
Pośrednicy zazwyczaj zarządzają całym cyklem życia swych obiektów usługowych.

4. **Klient** powinien móc pracować zarówno z usługami, jak i pełnomocnikami za pośrednictwem takiego samego interfejsu. Dzięki temu można umieścić pełnomocnika w dowolnym kacie, który ma współdziałać z obiektem usługowym.

## # Pseudokod

Poniższy przykład ilustruje jak wzorzec **Pełnomocnik** może pomóc dodać obsługę leniwej inicjalizacji i pamięci podręcznej do biblioteki innego producenta, odpowiedzialnej za integrację z YouTube.

Biblioteka udostępnia klasę do pobierania filmów wideo. Jest jednak bardzo nieefektywna. W przypadku otrzymania żądania pobrania drugi raz tego samego filmu, biblioteka pobierze go od nowa, zamiast buforować dane pozyskane podczas poprzedniego pobrania.



*Przechowywanie w pamięci podręcznej danych uzyskanych od usługi za pomocą pełnomocnika.*

Klasa pośrednicząca implementuje ten sam interfejs, co pierwotne narzędzie do pobierania i deleguje mu całą pracę. Zapamiętuje jednak pobrane pliki i zwraca dane z pamięci podręcznej, jeśli otrzyma żądanie pobrania tego samego filmu kolejny raz.

```

1 // Interfejs zdalnej usługi.
2 interface ThirdPartyYouTubeLib is
3     method listVideos()
4     method getVideoInfo(id)
5     method downloadVideo(id)
6
7 // Konkretna implementacja łącza do usługi. Metody tej klasy
8 // mogą żądać informacji z YouTube. Szybkość realizacji żądania

```

```
9 // zależy od połączenia internetowego użytkownika oraz od samego
10 // YouTube. Aplikacja będzie działać wolniej, jeśli wiele żądań
11 // zostanie wysłanych jednocześnie, nawet jeśli wszystkie
12 // żądania dotyczą tych samych danych.
13 class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is
14     method listVideos() is
15         // Wyślij żądanie do interfejsu programowania aplikacji
16         // (API) YouTube.
17
18     method getVideoInfo(id) is
19         // Pobierz metadane pliku wideo.
20
21     method downloadVideo(id) is
22         // Pobierz plik wideo z YouTube.
23
24 // Aby zaoszczędzić nieco przepustowości, możemy stworzyć pamięć
25 // podręczną do przechowywania pobranych danych i przechowywać
26 // je jakiś czas. Jednak umieszczenie takiego kodu bezpośrednio
27 // w klasie usługi może być niemożliwe, jeśli na przykład
28 // stanowi część biblioteki od zewnętrznego dostawcy i/lub jest
29 // zdefiniowana jako `final`. Dlatego też umieszczamy kod
30 // pamięci podręcznej w odrębnej klasie pośredniczącej która
31 // implementuje taki sam interfejs co klasa usługi. Nowo
32 // powstała klasa deleguje żądania faktycznemu obiekowi usługi
33 // tylko wtedy gdy faktycznie trzeba przesłać żądanie przez
34 // sieć.
35 class CachedYouTubeClass implements ThirdPartyYouTubeLib is
36     private field service: ThirdPartyYouTubeLib
37     private field listCache, videoCache
38     field needReset
39
40     constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib) is
```

```
41     this.service = service
42
43     method listVideos() is
44         if (listCache == null || needReset)
45             listCache = service.listVideos()
46
47         return listCache
48
49     method getVideoInfo(id) is
50         if (videoCache == null || needReset)
51             videoCache = service.getVideoInfo(id)
52
53         return videoCache
54
55     method downloadVideo(id) is
56         if (!downloadExists(id) || needReset)
57             service.downloadVideo(id)
58
59 // Klasa GUI która dotychczas współpracowała bezpośrednio z
60 // obiektem oferującym usługę pozostaje niezmieniona, o ile
61 // będzie współpracować z obiektem usługi poprzez interfejs.
62 // Możemy śmiało przekazać obiekt pośrednika zamiast obiektu
63 // usługi ponieważ implementują one ten sam interfejs.
64
65 class YouTubeManager is
66     protected field service: ThirdPartyYouTubeLib
67
68     constructor YouTubeManager(service: ThirdPartyYouTubeLib) is
69         this.service = service
70
71     method renderVideoPage(id) is
72         info = service.getVideoInfo(id)
73         // Renderuj stronę z filmem.
74
75     method renderListPanel() is
```

```
73     list = service.listVideos()  
74     // Renderuj listę miniaturek plików wideo.  
75  
76     method reactOnUserInput() is  
77         renderVideoPage()  
78         renderListPanel()  
79  
80     // Aplikacja może konfigurować pośredników w trakcie działania.  
81     class Application is  
82         method init() is  
83             aYouTubeService = new ThirdPartyYouTubeClass()  
84             aYouTubeProxy = new CachedYouTubeClass(aYouTubeService)  
85             manager = new YouTubeManager(aYouTubeProxy)  
86             manager.reactOnUserInput()
```

## ⌚ Zastosowanie

Wzorzec Pełnomocnik może przydać się w wielu przypadkach. Przejrzyjmy najpopularniejsze przypadki użycia tej koncepcji.

- ⌚ **Leniwa inicjalizacja (wirtualny pełnomocnik). Gdy masz do czynienia z zasobozernym obiektem usługi, którego potrzebujesz jedynie co jakiś czas.**
- ⚡ Zamiast tworzyć obiekt podczas uruchamiania aplikacji, możesz opóźnić inicjalizację obiektu do momentu gdy faktycznie staje się potrzebny.

- ⚡ Kontrola dostępu (pełnomocnik ochronny). Przydatne, gdy chcesz pozwolić tylko niektórym klientom na korzystanie z obiektu usługi. Na przykład, gdy usługi stanowią kluczową część systemu operacyjnego, a klienci to różne uruchamiane aplikacje (również te szkodliwe).
- ⚡ Pełnomocnik może przekazać żądanie obiekowi usługi tylko wtedy, gdy klient przedstawi odpowiednie poświadczania.
- ⚡ Lokalne uruchamianie zdalnej usługi (pełnomocnik zdalny). Użyteczne, gdy obiekt udostępniający usługę znajduje się na zdalnym serwerze.
  - ⚡ W takim przypadku, pełnomocnik przekazuje żądania klienta przez sieć, biorąc na siebie kłopotliwe szczegóły przesyłu.
- ⚡ Prowadzenie dziennika żądań (pełnomocnik prowadzący dziennik). Pozwala prowadzić rejestr żądań przesyłanych do obiektu usługi.
  - ⚡ Pełnomocnik może zapisywać do dziennika każde żądanie przed przekazaniem go usłudze.
- ⚡ Przechowywanie w pamięci podręcznej wyników działań (pełnomocnik z pamięcią podręczną). Pozwala przechować wyniki przekazywanych żądań i zarządzać cyklem życia pamięci podręcznej. Szczególnie ważne przy dużych wielkościach danych wynikowych.

- ⚡ Pełnomocnik może implementować pamięć podręczną często powtarzających się żądań dających ten sam wynik. Można wykorzystać parametry żądania w charakterze kluczy identyfikujących odpowiedni obszar pamięci podręcznej.
- ⚡ **Sprytne referencje. Można likwidować zasobozerny obiekt, gdy nie ma klientów którzy go potrzebują.**
- ⚡ Pełnomocnik może pozwolić na śledzenie klientów którzy otrzymali referencję do obiektu usługi lub wyników jego pracy. Co jakiś czas pełnomocnik może przejrzeć listę klientów, sprawdzając czy wciąż są aktywni. Jeśli lista klientów okazuje się pusta, pełnomocnik może zlikwidować obiekt usługi i tym samym zwolnić zasoby systemowe.

Pełnomocnik może też pamiętać, że klient zmodyfikował obiekt-usługę. Dzięki temu niezmienione obiekty mogą być ponownie wykorzystane przez innych klientów.

## ⌚ Jak zaimplementować

1. Jeśli nie ma istniejącego interfejsu usługi, stwórz go, aby uczyć się pełnomocnika i obiekt usługi wymiennymi. Ekstrakcja interfejsu z klasy usługi nie zawsze jest możliwa, ponieważ trzeba by było zmienić wszystkich klientów usługi tak, by komunikowali się przez ten interfejs. Plan B to stworzenie pełnomocnika w formie podklasy klasy usługi. Dzięki temu pełnomocnik odziedziczy interfejs usługi.

2. Stwórz klasę pełnomocnika. Powinna posiadać pole służące do przechowywania odniesienia do usługi. Zazwyczaj pośrednicy zarządzają całym cyklem życia usługodawców, włącznie z tworzeniem ich obiektów. W pewnych przypadkach obiekt usługi jest przekazywany przez klienta pełnomocnikowi za pośrednictwem konstruktora.
3. Zaimplementuj metody pełnomocnika zgodnie z ich przeznaczeniem. W większości przypadków, po wykonaniu jakieś części pracy, pełnomocnik powinien oddelegować jej resztę obiekowi usługi.
4. Rozważ utworzenie metody kreacyjnej która decyduje o tym, czy klient otrzyma obiekt pełnomocnika, czy faktyczny obiekt usługi. Może to być prosta, statyczna metoda w klasie pełnomocnika, albo w pełni rozwinięta metoda twórcza.
5. Przemyśl implementację leniwej inicjalizacji obiektu usługi.

## ΔΔ Zalety i wady

- ✓ Można sterować obiektem usługi bez wiedzy klientów.
- ✓ Można zarządzać cyklem życia obiektu usługi, gdy klientów to nie interesuje.
- ✓ Pełnomocnik działa nawet wtedy, gdy obiekt udostępniający usługę nie jest jeszcze gotowy lub dostępny.
- ✓ *Zasada otwarte/zamknięte.* Można wprowadzać nowych pełnomocników do aplikacji bez modyfikowania usług lub klientów.

- ✗ Kod może ulec skomplikowaniu, ponieważ trzeba wprowadzić wiele nowych klas.
- ✗ Odpowiedzi ze strony usługi mogą ulec opóźnieniu.

## ↔ Powiązania z innymi wzorcami

- Adapter wyposaża “opakowywany” obiekt w inny interfejs, Pełnomocnik w taki sam, zaś Dekorator wprowadza rozszerzony interfejs.
- Fasada przypomina wzorzec Pełnomocnik w tym sensie, że oba buforują złożony podmiot i inicjalizują go samodzielnie. W przeciwieństwie do *Fasady*, Pełnomocnik ma taki sam interfejs jak obiekt udostępniający usługę który ją reprezentuje, co czyni je wymienialnymi.
- Dekorator i Pełnomocnik mają podobne struktury, ale inne cele. Oba wzorce bazują na zasadzie kompozycji – jeden obiekt deleguje część zadań innemu. Pełnomocnik dodatkowo zarządza cyklem życia obiektu udostępniającego jakąś usługę, zaś komponowanie *Dekoratorów* leży w gestii klienta.

# Wzorce behawioralne

Wzorce behawioralne dotyczą algorytmów i rozdzielania odpowiedzialności pomiędzy obiektami.



## Łańcuch zobowiązań

Chain of Responsibility

Pozwala przekazywać żądania według łańcucha obiektów obsługujących. Otrzymawszy żądanie, każdy z obiektów obsługujących decyduje o zrealizowaniu żądania lub przekazaniu go do swojego następnika w łańcuchu.



## Polecenie

Command

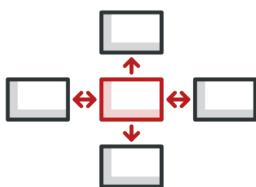
Zmienia żądanie w samodzielny obiekt zawierający wszystkie informacje o tym żądaniu. Taka transformacja pozwala na parametryzowanie metod przy przy użyciu różnych żądań. Oprócz tego umożliwia opóźnianie lub kolejkowanie wykonywania żądań oraz pozwala na cofanie operacji.



## Iterator

Iterator

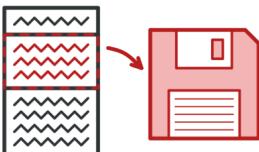
Pozwala przechodzić sekwencyjnie po elementach zbioru bez konieczności eksponowania jego formy (lista, stos, drzewo, itp.).



## Mediator

Mediator

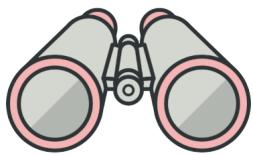
Pozwala zredukować chaos zależności pomiędzy obiektami. Wzorzec ogranicza bezpośrednią komunikację pomiędzy obiektami i zmusza je do współpracy wyłącznie za pośrednictwem obiektu mediatora.



## Pamiątka

Memento

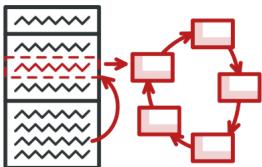
Pozwala zapisywać i przywracać wcześniejszy stan obiektu bez ujawniania szczegółów jego implementacji.



## Obserwator

Observer

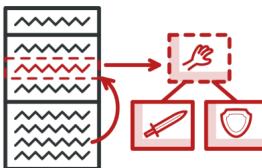
Pozwala zdefiniować mechanizm subskrypcji by powiadamiać wiele obiektów o zdarzeniach odbywających się w obserwowanym obiekcie.



## Stan

State

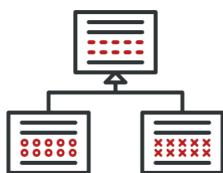
Pozwala obiektem zmienić swoje zachowanie gdy zmieni się jego wewnętrzny stan. Wygląda to tak, jakby obiekt zmienił swoją klasę.



## Strategia

Strategy

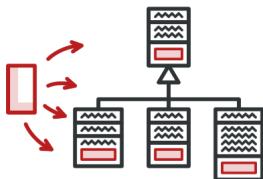
Pozwala zdefiniować rodzinę algorytmów, umieścić je w osobnych klasach i uczynić obiekty tych klas wymienialnymi.



## Metoda szablonowa

Template Method

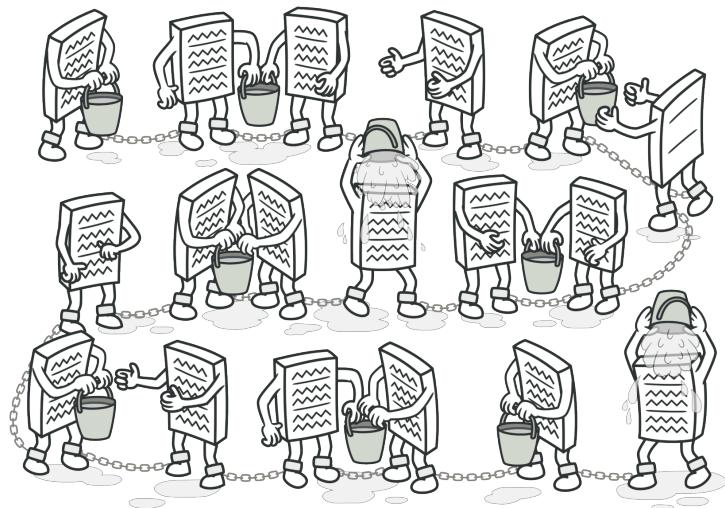
Definiuje szkielet algorytmu w klasie bazowej, ale umożliwia podklasom nadpisanie poszczególnych etapów algorytmu bez konieczności zmiany jego struktury.



## Odwiedzający

Visitor

Pozwala oddzielić algorytmy od obiektów na których pracują.



# ŁAŃCUCH ZOBOWIĄZAŃ

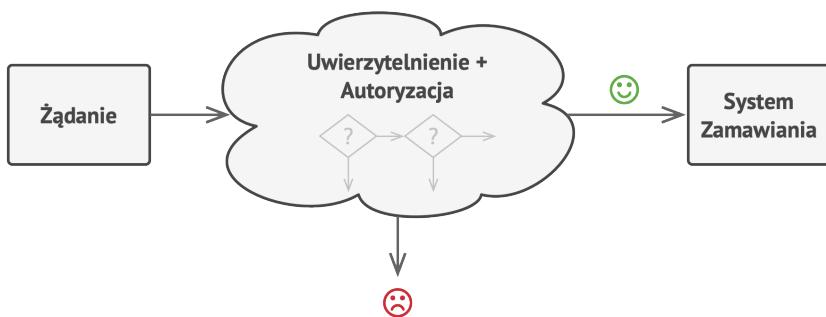
Znany też jako: *Chain of command*, *Chain of Responsibility*

**Łańcuch zobowiązań** jest behawioralnym wzorcem projektowym, który pozwala przekazywać żądania wzdłuż łańcucha obiektów obsługujących. Otrzymawszy żądanie, każdy z obiektów obsługujących decyduje o przetworzeniu żądania lub przekazaniu go do kolejnego obiektu obsługującego w łańcuchu.

## (:() Problem

Wyobraź sobie, że pracujesz nad systemem zamawiania online. Chcesz ograniczyć dostęp do systemu, by wyłącznie użytkownicy uwierzytelnieni mogli składać zamówienia. Ponadto użytkownicy z uprawnieniami administracyjnymi powinni mieć pełen dostęp do wszystkich zamówień.

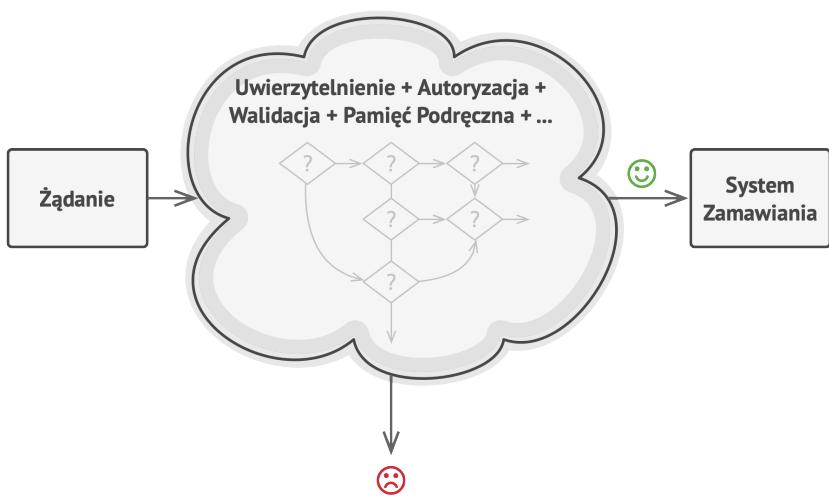
Po obmyśleniu planu, zdajesz sobie sprawę, że takie sprawdzenia powinno się wykonywać sekwencyjnie. Aplikacja może spróbować uwierzytelnić użytkownika otrzymawszy żądanie zawierające poświadczania użytkownika. Jednak jeśli poświadczania nie są prawidłowe i uwierzytelnienie nie powiedzie się, nie ma powodu dokonywać dalszych sprawdzeń.



*Żądanie musi przejść serię sprawdzeń zanim system zamawiania będzie mógł go obsłużyć.*

W kolejnych miesiącach, implementujesz wiele takich sekwencyjnych sprawdzeń.

- Jeden z twoich współpracowników zauważa, że przekazywanie surowych danych przez system zamawiania nie jest bezpieczne. Dodajesz więc etap walidacyjny, czyszczący dane zawarte w żądaniu.
- Później ktoś zauważa, że system jest podatny na łamanie haseł metodą brute force. Aby się przed tym uchronić, dodajesz sprawdzenie odrzucające wielokrotne nieskuteczne próby uwierzytelnienia przychodzące z tego samego adresu IP.
- Ktoś inny zaś zasugerował, że można przyspieszyć działanie systemu, gdyby zwracał on przechowane w pamięci podręcznej wyniki żądań zawierające te same dane. Dodajesz więc kolejne sprawdzenie, pozwalające żądaniu przejść dalej tylko jeśli nie ma już stosownej odpowiedzi zapisanej w pamięci podręcznej.



*Im bardziej kod urósł, tym bardziej stał się zabałaganiony.*

Kod sprawdzeń, który już na początku wyglądał pogmatwane, spuchł jeszcze bardziej wraz z dodawaniem funkcjonalności. Zmiana jednego sprawdzenia czasem wpływała na inne. A co najgorsze, próba ponownego użycia sprawdzeń w zabezpieczeniu innych komponentów systemu spowodowała duplikację części kodu ponieważ niektóre komponenty potrzebowaly sprawdzeń, ale inne nie.

System stał się trudny do zrozumienia i kosztowny w utrzymaniu. Po okresie trudzenia się postanawiasz dokonać refaktoryzacji całości.

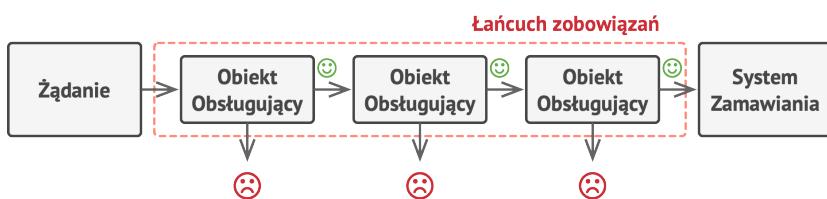
## Rozwiązańe

Jak wiele innych wzorców behawioralnych, **Łańcuch Zobowiązań** zakłada przekształcenie pewnych obowiązków w samodzielne obiekty zwane *obiektami obsługującymi*. W naszym przypadku, każde sprawdzenie powinno się wyekstrahować do osobnej klasy posiadającej jedną metodę dokonującą sprawdzenia. Żądanie wraz z towarzyszącymi mu danymi przekazywane jest jako argument tej metody.

Wzorzec sugeruje połączenie tych obiektów obsługujących w łańcuch. Każdy obiekt obsługujący stanowiący ognisko łańcucha posiada pole przechowujące odniesienie do następnego obiektu w łańcuchu. Poza przetworzeniem żądania, obiekty przekazują je dalej. Żądanie biegnie wzdłuż łańcucha, by wszystkie ogniska miały okazję je obsłużyć.

A co najlepsze, obiekt obsługujący może zdecydować o nie-przekazaniu żądania dalej i tym samym kończy proces.

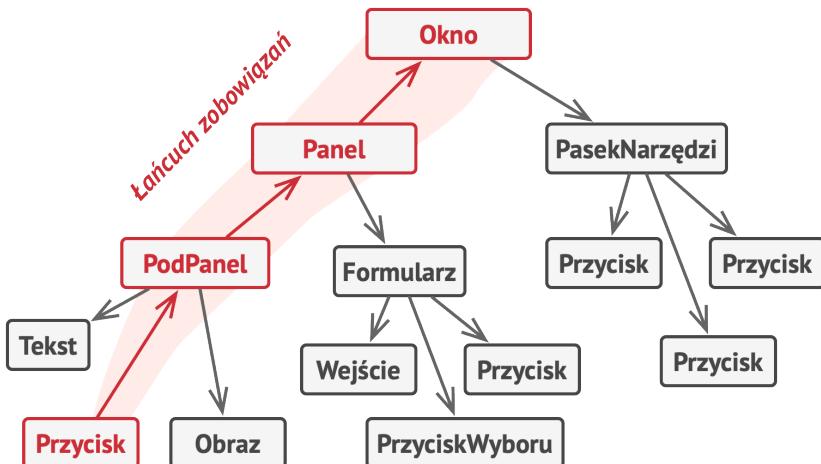
W naszym przykładzie systemu zamawiającego, obiekt obsługujący dokonuje przetwarzania żądania i decyduje o przekazaniu go dalej, lub nie. Zakładając, że żądanie zawiera właściwe dane, obiekty obsługujące mogą wykonywać swoje obowiązki, takie jak uwierzytelnianie czy zapis w pamięci podręcznej.



*Obiekty obsługujące są ułożone jeden obok drugiego, tworząc łańcuch.*

Istnieje jednak nieco inne podejście (które weszło do kanonu), według którego obiekt obsługujący otrzymawszy żądanie decyduje czy może je obsłużyć i jeśli tak, to nie przekazuje go dalej. Więc albo tylko jeden obiekt obsługuje jedno żądanie, albo żaden. Podejście to jest bardzo powszechnie w przypadku stosu zdarzeń w obrębie graficznego interfejsu użytkownika.

Na przykład, gdy użytkownik kliknie przycisk, zdarzenie rozpropaguje się wzduż łańcucha elementów UI, zaczynając od przycisku, poprzez jego kontenery (formatki lub panele) i dociera do głównego okna aplikacji. Zdarzenie jest przetwarzane przez pierwszy element w łańcuchu który jest w stanie je obsłużyć. Ten przykład jest też godny uwagi, bo pokazuje jak z każdego drzewa obiektów można wyekstrahować łańcuch.



Z gałęzi drzewa obiektów można uformować łańcuch.

Istotnym jest, że wszystkie klasy obiektów obsługujących implementują ten sam interfejs. Każdy konkretny obiekt obsługujący powinien wiedzieć tylko o następnym, posiadającym metodę `wykonaj`. W ten sposób można komponować łańcuchy w trakcie działania programu, stosując różne obiekty obsługujące bez sprzągania kodu z ich konkretnymi klasami.

## 🚗 Analogia do prawdziwego życia

Właśnie kupiłeś sobie i zainstalowałeś jakąś część do komputera. Ponieważ jesteś geekiem, na komputerze jest kilka systemów operacyjnych. Uruchamiasz więc jeden po drugim, sprawdzając czy urządzenie jest obsługiwane. Windows wykrywa i włącza urządzenie automatycznie. Jednak twoja ukochana dystrybucja Linuksa odmawia współpracy. W nikłym przebłysku nadziei, dzwonisz na numer pomocy technicznej podany na opakowaniu.

Pierwsze, co słyszysz, to sztucznie brzmiący głos automatu zgłoszeniowego. Sugeruje on dziewięć typowych rozwiązań różnych problemów, ale żaden z nich nie dotyczy twoego przypadku. Po jakimś czasie, automat poddaje się i łączy cię z żywym człowiekiem.

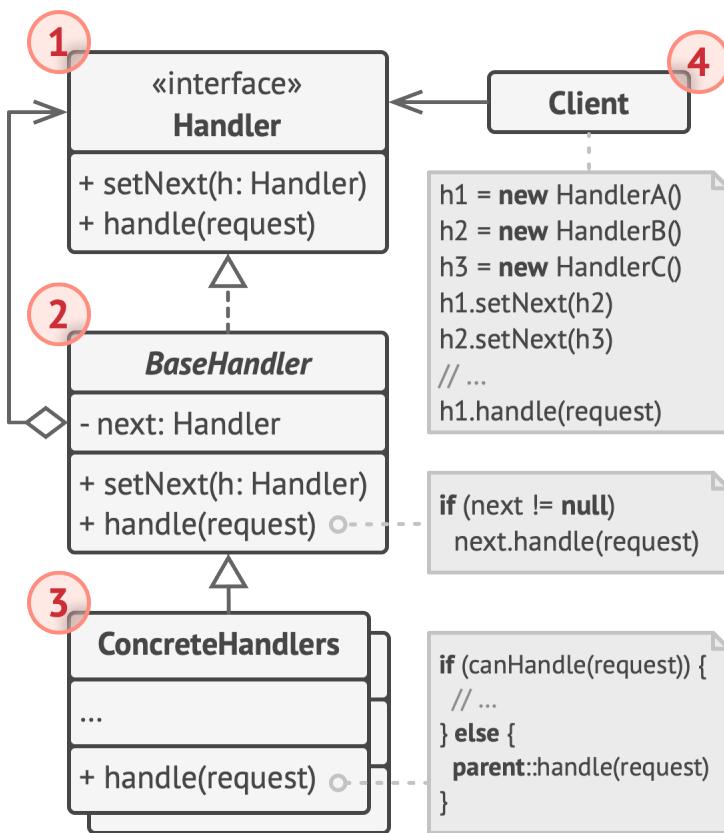


*Połączenie z pomocą techniczną może być przekazane kolejnym jej pracownikom.*

Ale żywy pracownik również nie jest w stanie zasugerować nic pożytecznego. Cytuje długie ustępy instrukcji obsługi i nie słucha twoich uwag. Po usłyszeniu dziesiąty raz sugestii “proszę spróbować wyłączyć i włączyć ponownie komputer”, żądasz połączenia z prawdziwym inżynierem.

Ostatecznie łączy cię z jednym z inżynierów, który zapewne od wielu godzin teskni za rozmową z żywym człowiekiem, siedząc w swojej odosobnionej serwerowni gdzieś w ciemnej piwnicy. Inżynier podaje ci link do odpowiednich sterowników do urządzenia i tłumaczy jak je zainstalować pod Linuksem. Wreszcie – rozwiązanie! Rozłączasz się pełen radości.

## Struktura



1. **Obiekt Obsługujący** deklaruje wspólny dla wszystkich obiektów obsługujących interfejs. Zazwyczaj posiada on tylko jedną metodę do obsługi żądań, ale czasem może zawierać też drugą, służącą do wybierania kolejnego obiektu w łańcuchu.
2. **Bazowy Obiekt Obsługujący** to opcjonalna klasa, gdzie można umieścić kod przygotowawczy, wspólny dla wszystkich klas obsługujących.

Na ogół klasa ta definiuje pole służące przechowywaniu odniesienia do kolejnego obiektu obsługującego. Klienci mogą sformować łańcuch przekazując obiekt obsługujący konstruktorowi lub metodzie setter poprzedniego obiektu. Klasa może też implementować domyślną obsługę: przekazać wykonanie kolejnemu obiekowi obsługującemu, sprawdziwszy, czy taki istnieje.

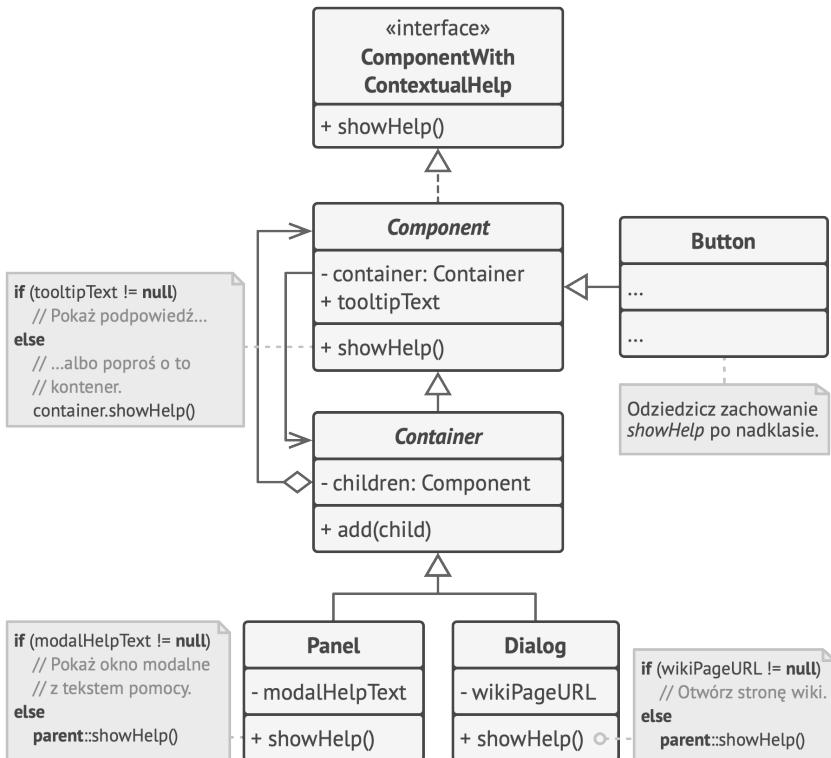
3. **Konkretne Obiekty Obsługujące** zawierają faktyczny kod służący obsłudze żądań. Otrzymawszy żądanie, każdy obiekt musi zdecydować, czy je obsłużyć i czy przekazać je dalej.

Obiekty obsługujące są zazwyczaj samodzielne i niezmienne, akceptują wszystkie konieczne dane jednorazowo za pośrednictwem konstruktora.

4. **Klient** może skomponować łańcuch raz, albo robić to dynamicznie, zależnie od logiki aplikacji. Warto pamiętać, że żądanie może być przekazane dowolnemu ogniwu łańcucha – niekoniecznie pierwszemu.

## # Pseudokod

W poniższym przykładzie, wzorzec **Łańcuch Zobowiązań** jest odpowiedzialny za wyświetlanie pomocy kontekstowej dotyczącej aktywnych elementów interfejsu użytkownika.

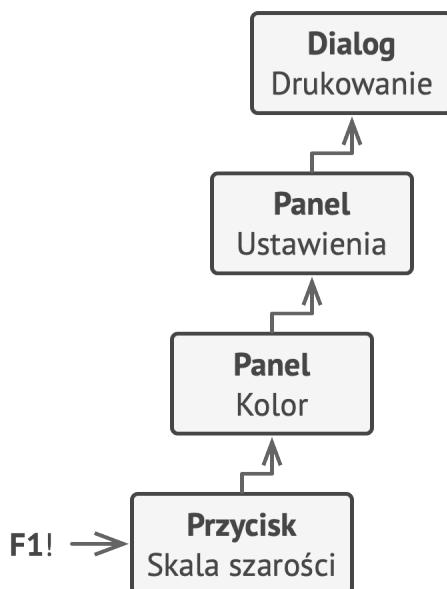


*Klasy interfejsu użytkownika (GUI) zbudowano według wzorca Kompozyt.*

*Każdy element jest powiązany ze swoim kontenerem. W dowolnym momencie można stworzyć łańcuch elementów, zaczynający się od samego elementu i prowadzący przez wszystkie elementy kontenera.*

Interfejs użytkownika aplikacji zazwyczaj jest ustrukturyzowany w formie drzewa obiektów. Na przykład klasa **Dialog**, która renderuje główne okno aplikacji byłaby korzeniem drzewa obiektów. **Dialog** zawiera **Panele**, które mogą z kolei zawierać inne panele lub proste niskopoziomowe elementy jak **Przyciski** i **PolaTekstowe**.

Prosty komponent może pokazywać krótkie kontekstowe podpowiedzi, o ile ma przypisaną mu jakąś treść pomocy. Ale bardziej złożone komponenty definiują swoje sposoby na wyświetlanie pomocy kontekstowej, jak prezentacja stosownego fragmentu instrukcji obsługi lub otwarcie strony internetowej w przeglądarce.



*Sposób w jaki żądanie pomocy przechodzi przez kolejne obiekty GUI.*

Gdy użytkownik wskaże kursem jakiś element i wciśnie klawisz **F1**, aplikacja sprawdza jaki komponent znajduje się pod kursem i wysyła mu żądanie wyświetlenia pomocy. Żądanie przechodzi przez wszystkie kontenery elementu, aż dotrze do tego, który jest w stanie obsłużyć żądanie.

```
1 // Interfejs obiektu obsługującego deklaruje metodę wykonującą
2 // żądanie.
3 interface ComponentWithContextualHelp is
4     method showHelp()
5
6
7 // Klasa bazowa prostych komponentów.
8 abstract class Component implements ComponentWithContextualHelp is
9     field tooltipText: string
10
11 // Kontener komponentu pełni rolę kolejnego ogniska łańcucha
12 // obiektów obsługujących żądanie.
13 protected field container: Container
14
15 // Komponent pokazuje podpowiedź jeśli przypisano mu jakiś
16 // tekst pomocy. W przeciwnym razie przekazuje wywołanie
17 // kontenerowi, o ile takowy istnieje.
18 method showHelp() is
19     if (tooltipText != null)
20         // Pokaż podpowiedź.
21     else
22         container.showHelp()
23
24
25 // Kontenery mogą zawierać zarówno proste komponenty, jak i inne
26 // kontenery podrzędne. Tu ustala się relacje łańcucha. Klasa
27 // dziedziczy zachowanie showHelp od klasy-rodzica.
28 abstract class Container extends Component is
29     protected field children: array of Component
30
31     method add(child) is
32         children.add(child)
```

```
33     child.container = this
34
35
36 // Prymitywne komponenty mogą posiadać tylko domyślną
37 // implementację funkcji pomoc...
38 class Button extends Component is
39     // ...
40
41 // Ale złożone komponenty mogą nadpisywać domyślną
42 // implementację. Jeśli nie ma innej treści pomocy, komponent
43 // może zawsze wywołać implementację z klasy bazowej (patrz
44 // klasa Component).
45 class Panel extends Container is
46     field modalHelpText: string
47
48     method showHelp() is
49         if (modalHelpText != null)
50             // Wyświetl modalne okno dialogowe z treścią pomocy.
51         else
52             super.showHelp()
53
54 // ...jak wyżej...
55 class Dialog extends Container is
56     field wikiPageURL: string
57
58     method showHelp() is
59         if (wikiPageURL != null)
60             // Otwórz stronę z pomocą na wiki.
61         else
62             super.showHelp()
```

```

65 // Kod klienta.
66 class Application is
67 // Każda aplikacja inaczej konfiguruje łańcuch.
68 method createUI() is
69     dialog = new Dialog("Budget Reports")
70     dialog.wikiPageURL = "http://..."
71     panel = new Panel(0, 0, 400, 800)
72     panel.modalHelpText = "This panel does..."
73     ok = new Button(250, 760, 50, 20, "OK")
74     ok.tooltipText = "This is an OK button that..."
75     cancel = new Button(320, 760, 50, 20, "Cancel")
76     // ...
77     panel.add(ok)
78     panel.add(cancel)
79     dialog.add(panel)
80
81 // Wyobraź sobie co tu się może dziać.
82 method onF1KeyPress() is
83     component = this.getComponentAtMouseCoords()
84     component.showHelp()

```

## ⌚ Zastosowanie

- ⌚ Stosuj wzorzec Łańcuch zobowiązań gdy twój program ma obsługiwać różne rodzaje żądań na różne sposoby, ale dokładne typy żądań i ich sekwencji nie są wcześniej znane.
- ⚡ Wzorzec pozwala połączyć wiele obiektów obsługujących w jeden łańcuch i otrzymawszy żądanie “odpytać” każde ognisko

czy jest w stanie je obsłużyć. W ten sposób wszystkie obiekty obsługujące mają okazję przetworzyć żądanie.

 **Stosuj ten wzorzec gdy istotne jest uruchomienie wielu obiektów obsługujących w pewnej kolejności.**

 Skoro można połączyć obiekty obsługujące w dowolnej kolejności, wszystkie żądania przejdą przez łańcuch w takim porządku, jaki zaplanowano.

 **Łańcuch zobowiązań pozwala ustawić obiekty obsługujące i ich kolejność w czasie działania programu.**

 Jeśli eksponujesz w klasie obsługującej metodę setter, ustawiające pole przechowujące odniesienie, będzie można wstawiać, usuwać lub zmieniać kolejność ogniw łańcucha dynamicznie.

## Jak zaimplementować

1. Zadeklaruj interfejs obiektu obsługującego i opisz sygnaturę metody obsługującej żądania.

Zdecyduj jak klient będzie przekazywał dane żądań do metody. Najbardziej elastycznym sposobem jest konwersja żądania na obiekt i przekazywanie go metodzie obsługującej w charakterze argumentu.

2. Aby wyeliminować powtarzający się kod przygotowawczy w konkretnych obiektach obsługujących, być może warto utworzyć abstrakcyjną bazową klasę obiektu obsługującego, wywołującą się z interfejsu obiektu obsługującego.

Klasa taka powinna zawierać pole przechowujące odniesienie do kolejnego ogniska łańcucha. Rozważ użycie tej klasy niezmienialnej. Jednak jeśli planujesz modyfikować łańcuch w czasie działania programu, musisz też zdefiniować setter zmieniający wartość pola z odniesieniem.

Można także zaimplementować wygodne domyślne zachowanie metody obsługującej, która przekieruje żądanie do kolejnego obiektu o ile takowy istnieje. Konkretne obiekty obsługujące będą w stanie skorzystać z tego zachowania wywołując metodę nadklasy.

3. Jeden po drugim twórz podklasy obiektów obsługujących i zaimplementuj im metody obsługujące. Każdy obiekt obsługujący powinien podjąć dwie decyzje otrzymany żądanie:
  - Czy przetworzyć żądanie.
  - Czy przekazać żądanie dalej wzdłuż łańcucha.
4. Klient może albo złożyć łańcuch samodzielnie, albo otrzymać wcześniej przygotowany od innego obiektu. W drugim przypadku, trzeba zaimplementować jakieś klasy fabryczne do budowy łańcuchów zgodnie z konfiguracją lub ustawieniami środowiska.

5. Klient może uruchomić kolejny obiekt w łańcuchu, niekoniecznie pierwszy. Żądanie zostanie przekazane dalej wzdłuż łańcucha, aż jakiś obiekt obsługujący odmówi przekazania go dalej, albo nie będzie już komu je przekazać.
6. W związku z dynamiczną naturą łańcucha, klient powinien być gotów obsłużyć następujące scenariusze:
  - łańcuch zawiera tylko jedno ognisko.
  - Niektóre żądania mogą nie dotrzeć do końca łańcucha.
  - Inne żądania mogą dotrzeć do końca łańcucha i nie zostać obsłużone.

## Zalety i wady

- ✓ Można ustalać porządek obsługi żądania.
- ✓ *Zasada pojedynczej odpowiedzialności.* Można rozprzegnać klasy wywołujące działania klas od klas wykonujących działania.
- ✓ *Zasada otwarte/zamknięte.* Można wprowadzać do programu nowe obiekty obsługujące bez psucia istniejącego kodu klienta.
- ✗ Niektóre żądania mogą wcale nie zostać obsłużone.

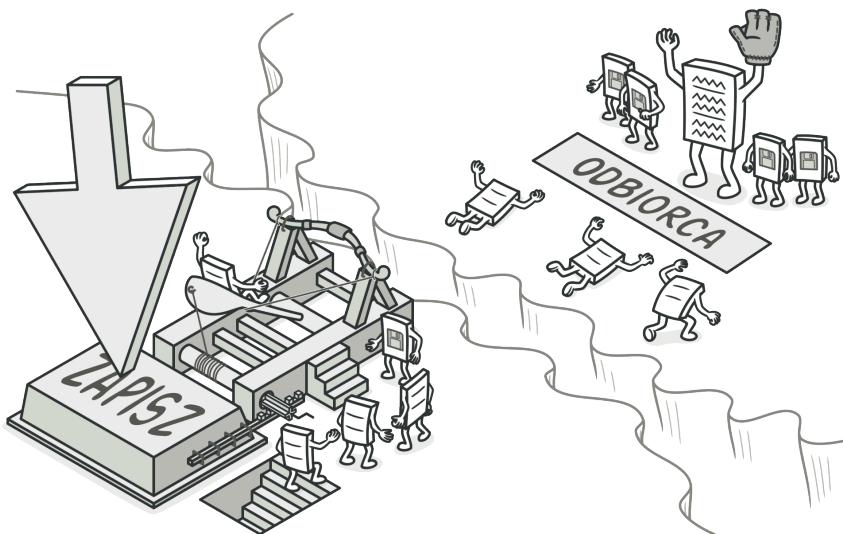
## ↔ Powiązania z innymi wzorcami

- Wzorce **Łańcuch zobowiązań**, **Polecenie**, **Mediator** i **Obserwator** dotyczą różnych sposobów na łączenie nadawców z odbiorcami żądań:
  - *Łańcuch zobowiązań* przekazuje żądanie sekwencyjnie wzdłuż dynamicznego łańcucha potencjalnych odbiorców, aż któryś z nich je obsłужy.
  - *Polecenie* pozwala nawiązywać jednokierunkowe połączenia pomiędzy nadawcami i odbiorcami.
  - *Mediator* eliminuje bezpośrednie połączenia pomiędzy nadawcami a odbiorcami, zmuszając ich do komunikacji za pośrednictwem obiektu mediator.
  - *Obserwator* pozwala odbiorcom dynamicznie zasubskrybować się i zrezygnować z subskrypcji żądań.
- **Łańcuch zobowiązań** często stosuje się w połączeniu z **Kompozytem**. W takim przypadku, gdy komponent-liść otrzymuje żądanie, może je przekazać poprzez łańcuch nadzędnych komponentów aż do korzenia drzewa obiektów.
- Obsługujący w **Łańcuchu zobowiązań** mogą być zaimplementowani jako **Polecenia**. Można wówczas wykonać wiele różnych działań reprezentowanych jako żądania na tym samym obiekcie-kontekście.

Istnieje jednak jeszcze jedno podejście, według którego samo żądanie jest obiektem **Polecenie**. W takim przypadku możesz wykonać to samo działanie na łańcuchu różnych kontekstów.

- **Łańcuch zobowiązań** i **Dekorator** mają bardzo podobne struktury klas. Oba wzorce bazują na rekursywnej kompozycji w celu przekazania obowiązku wykonania przez ciąg obiektów. Istnieją jednak kluczowe różnice.

Obsługujący *Łańcucha zobowiązań* mogą wykonywać działania niezależnie od siebie. Mogą również zatrzymać dalsze przekazywanie żądania na dowolnym etapie. Z drugiej strony, różne *Dekoratory* mogą rozszerzać obowiązki obiektu zachowując zgodność z interfejsem bazowym. Dodatkowo, dekoratory nie mają możliwości przerwania przepływu żądania.



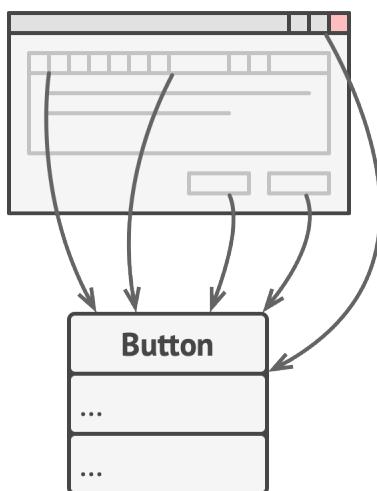
# POLECENIE

Znany też jako: *Action, Transaction, Command*

**Polecenie** jest behawioralnym wzorcem projektowym który zmienia żądanie w samodzielny obiekt zawierający wszystkie informacje o tym żądaniu. Taka transformacja pozwala na parametryzowanie metod przy użyciu różnych żądań. Oprócz tego umożliwia opóźnianie lub kolejkowanie wykonywania żądań oraz pozwala na cofanie operacji.

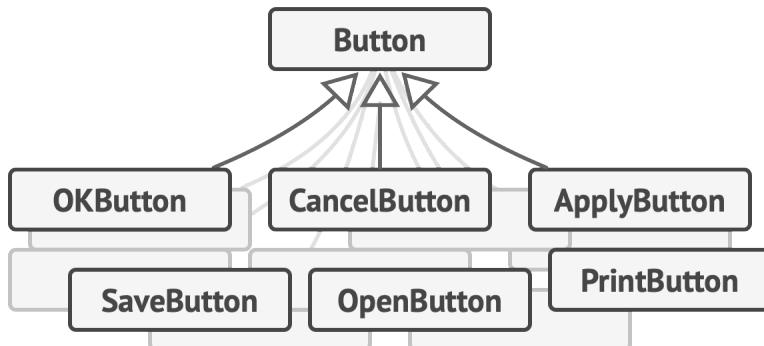
## (:() Problem

Wyobraź sobie, że pracujesz nad nowym edytorem tekstu. Tworzysz pasek narzędziowy z przyciskami wywołującymi różne działania edytora. Masz już elegancką klasę `Przycisk` która może być używana zarówno do przycisków paska, a także jako ogólne przyciski w różnych oknach dialogowych.



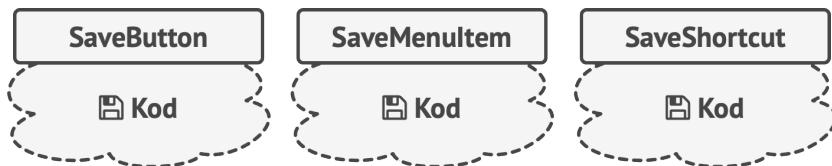
*Wszystkie przyciski w aplikacji wywodzą się z tej samej klasy.*

Mimo, że wszystkie te przyciski wyglądają podobnie, to mają wywoływać różne działania. Gdzie więc umieścić kod obiektów obsługujących kliknięcie przycisków? Najprostszym rozwiązańiem jest stworzenie wielu podklas dla każdego przypadku użycia przycisku. Takie podklasy zawierałyby kod wykonywany po wcisnięciu przycisku.



*Mnóstwo podklas przycisku. Co może pójść nie tak?*

Szybko zauważasz, że to podejście jest wadliwe. Powstanie wielka liczba podklas, co byłoby akceptowalne, gdybyśmy przy okazji nie ryzykowali popsucia kodu tych podklas przy każdej zmianie klasy bazowej **Przycisk**. Kod twojego interfejsu użytkownika byłby zależny od zmiennego kodu logiki biznesowej.



*Wiele klas implementuje tę samą funkcjonalność.*

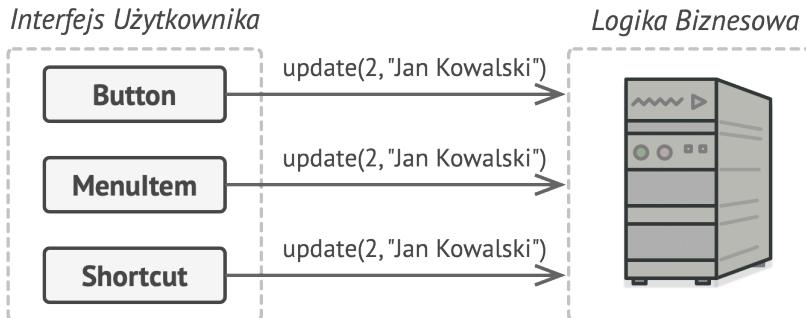
Ale to jeszcze nie wszystko. Niektóre operacje, jak kopiowanie/wklejanie tekstu, powinny być dostępne z wielu miejsc: po kliknięciu na mały przycisk “Kopiuj” na pasku narzędziowym, wybraniu z menu kontekstowego, czy też po wcisnięciu skrótu klawiszowego **Ctrl+C**.

Na początku, gdy nasza aplikacja posiadała tylko pasek narzędziowy, umieszczenie implementacji operacji w podklasach przycisku miało sens. Innymi słowy, trzymanie kodu służącego do kopiowania tekstu w obrębie podklasy `PrzyciskKopiuj` było w porządku. Jednak po zaimplementowaniu menu kontekstowego, skrótów i innych – trzeba duplikować kod operacji w wielu klasach lub uczynić menu zależnymi od przycisków, co jest jeszcze gorszą opcją.

## Rozwiążanie

Dobre projektowanie oprogramowania często bazuje na *zasadzie separacji odpowiedzialności*, co zwykle skutkuje podziałem aplikacji na warstwy. Najczęstszy przykład: warstwa graficznego interfejsu użytkownika i warstwa logiki biznesowej. Pierwsza jest odpowiedzialna za renderowanie pięknego obrazu na ekranie, przechwytywanie sygnałów na wejściu i wyświetlanie efektów pracy użytkownika i aplikacji. Jednak gdy chodzi o wykonywanie ważnych zadań, jak obliczanie trajektorii księżyca, lub generowanie rocznego bilansu, warstwa interfejsu użytkownika deleguje te zadania warstwie logiki biznesowej.

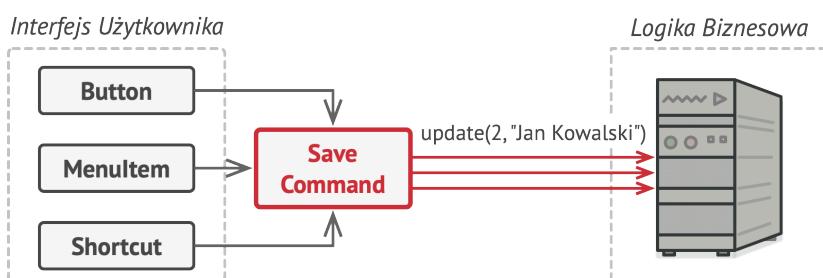
W kodzie wyglądałoby to na przykład tak: obiekt graficznego interfejsu użytkownika wywołuje metodę obiektu logiki biznesowej, przekazując jej jakieś argumenty. Proces ten zwykle można opisać jako przesłanie *żądania* przez jeden obiekt drugiemu obiekowi.



*Obiekty graficznego interfejsu użytkownika mogą mieć bezpośredni dostęp do obiektów warstwy logiki biznesowej.*

Według wzorca Polecenie, obiekty GUI nie powinny wysyłać żądań bezpośrednio. Zamiast tego należy wyekstrahować szczegóły żądania, takie jak obiekt docelowy, nazwę metody i listę argumentów do osobnej klasy *polecenie* posiadającej tylko jedną metodę – wywołującą to żądanie.

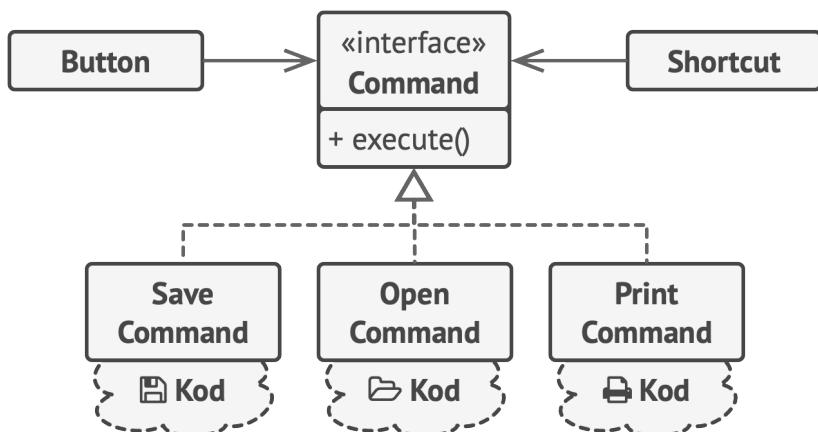
Obiekty polecenie stanowią łącza pomiędzy obiektami interfejsu użytkownika i logiki biznesowej. Od teraz, obiekt GUI nie musi wiedzieć który obiekt logiki biznesowej otrzyma żądanie i jak je obsłuży. Obiekt interfejsu użytkownika jedynie wywołuje polecenie, a ono samo zajmuje się szczegółami.



*Dostęp do warstwy logiki biznesowej za pośrednictwem polecenia.*

Kolejnym etapem jest zaimplementowanie wszystkim poleceniom jednakowego interfejsu. Zazwyczaj posiada on jedną tylko metodę wywołującą działanie, która nie przyjmuje parametrów. Taki interfejs pozwala jednemu nadawcy wywoływać wiele różnych poleceń bez konieczności sprzęgania go z konkretnymi klasami poleceń. Dodatkowo można teraz wymieniać obiekty-polecenia powiązane z nadawcą, a tym samym zmieniać jego zachowanie w trakcie działania programu.

Brakuje jeszcze jednego elementu układanki – parametrów żądania. Obiekt graficznego interfejsu użytkownika mógł dostarczyć obiekutowi warstwy logiki biznesowej jakieś parametry. Skoro wykonanie polecenia nie przyjmuje żadnych parametrów, to jak przekazać odbiorcy szczegóły żądania? Otóż albo te dane powinny być wcześniej skonfigurowane w poleceniu, albo polecenie powinno móc pozyskać je samodzielnie.



*Obiekty graficznego interfejsu użytkownika delegują pracę poleceniom.*

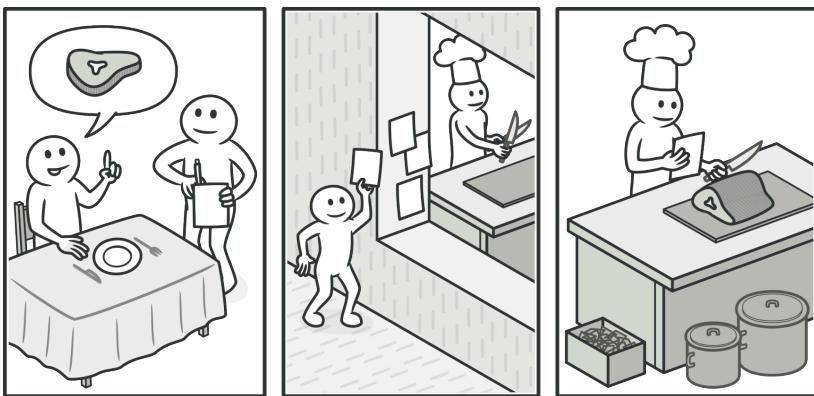
Wróćmy do naszego edytora tekstu. Po zastosowaniu wzorca **Polecenie**, nie potrzebujemy tych wszystkich podklas przycisku, by zaimplementować różne reakcje na kliknięcie. Wystarczy umieścić w klasie bazowej `Przycisk` jedno pole przechowujące odniesienie do obiektu typu polecenie i sprawić, by kliknięcie powodowało uruchomienie tego polecenia.

Należy zaimplementować kilka klas polecenie dla każdej możliwej operacji i połączyć je z konkretnymi przyciskami, zależnie od planowanej reakcji na wciskanie ich.

Inne elementy GUI, jak menu, skróty czy całe okna dialogowe można zaimplementować w taki sam sposób: powiązać je z poleceniem które będzie uruchamiane w odpowiedzi na interakcję użytkownika z danym elementem. Jak być może się już domyślasz, elementy związane z tymi samymi działaniami będą połączone z tymi samymi poleceniami, zapobiegając tym samym duplikacji kodu.

W rezultacie polecenia stają się poręczną warstwą pośrednią redukującą sprzężenie pomiędzy graficznym elementem użytkownika i warstwami logiki biznesowej. A to tylko część zysków płynących z użycia wzorca **Polecenie**!

## Analoga do prawdziwego życia

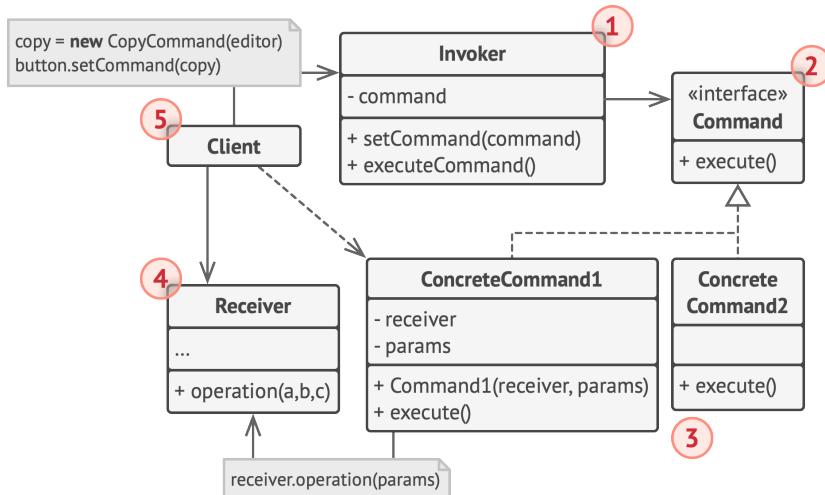


*Składanie zamówienia w restauracji.*

Podczas długiego spaceru po mieście, docierasz do miłej restauracji i siadasz przy oknie. Przyjazny kelner szybko przyjmuje zamówienie, spisując je na małym kawałku papieru. Następnie kelner idzie do kuchni i przykleja kartkę na ścianie. Po jakimś czasie zamówienie dociera do szefa kuchni, który przygotowuje danie, a następnie umieszcza posiłek na tacce wraz z zamówieniem. Kelner znajduje tacę, sprawdza zgodność z zamówieniem i zanosi ją do stolika.

Zamówienie na papierze stanowi polecenie. Trafia do kolejki, do momentu aż szef kuchni je przygotuje. Zamówienie zawiera wszystkie niezbędne informacje wymagane do przygotowania posiłku. Umożliwia to kucharzowi rozpoczęcie gotowania od razu, zamiast ustalać szczegóły z klientem na własną rękę.

## Struktura



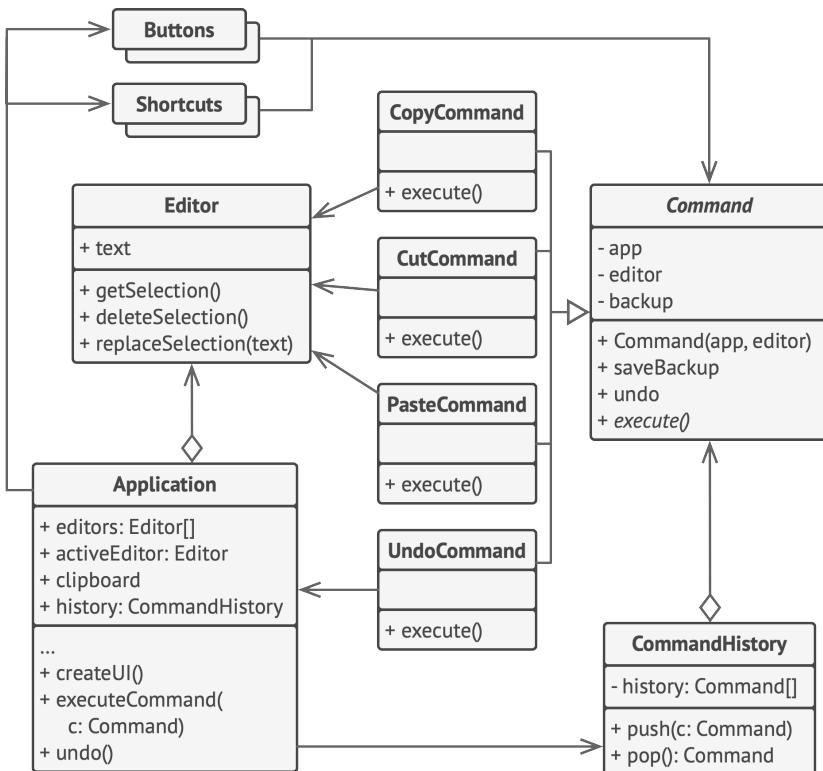
1. Klasa **Nadawca** (lub *wywołującą*) jest odpowiedzialna za inicjowanie żądań. Musi ona zawierać pole przechowujące odniesienia do obiektu polecenia. Nadawca uruchamia polecenie zamiast przesyłać żądanie bezpośrednio do odbiorcy. Zauważ, że nadawca nie jest odpowiedzialny za tworzenie obiektu polecenie. Zazwyczaj otrzymuje wcześniej przygotowane polecenie od klienta za pośrednictwem konstruktora.
2. Interfejs **Polecenie** zwykle deklaruje pojedynczą metodę służącą wykonaniu polecenia.
3. **Konkretne polecenia** implementują różne rodzaje żądań. Konkretne polecenie nie powinno wykonywać pracy samodzielnie, lecz przekazać je do jednego z obiektów logiki biznesowej. Jednak dla uproszczenia kodu, klasy te można złączyć.

Parametry potrzebne do uruchomienia metody na obiekcie odbiorcy można zadeklarować w formie pól konkretnego polecenia. Obiekty poleceń można uczynić niezmienialnymi, zezwalając na inicjalizację tych pól wyłącznie za pośrednictwem konstruktora.

4. Klasa **Odbiorca** zawiera jakąś logikę biznesową. Prawie każdy obiekt może pełnić rolę odbiorcy. Większość poleceń obsługuje tylko szczegóły przekazania żądania do odbiorcy, zaś faktyczną pracę wykonuje ten ostatni.
5. **Klient** tworzy i konfiguruje konkretne obiekty żądań. Klient musi przekazać wszystkie parametry żądania, włącznie z instancją odbiorcy, do konstruktora polecenia. Następnie otrzymane polecenie można skojarzyć z jednym lub wieloma nadawcami.

## # Pseudokod

W poniższym przykładzie, wzorzec **Polecenie** pozwala śledzić historię wykonanych działań i umożliwia cofnięcie danej operacji jeśli zaistnieje potrzeba.



*Odwracalne działania w edytorze tekstu.*

Polecenia skutkujące zmianą stanu edytora (na przykład wycinanie i wklejanie tekstu) wykonują kopię stanu edytora zanim wywołają działanie skojarzone z tym poleceniem. Po wykonaniu polecenia jest ono umieszczane w historii poleceń (stos obiektów polecenie) wraz z kopią zapasową stanu edytora na tamten moment. Jeśli użytkownik zechce cofnąć jakieś действие, aplikacja może pobrać ostatnie polecenie z historii, odczytać skojarzoną z nim kopię zapasową stanu edytora i przywrócić ją.

Kod klienta (elementy GUI, historia poleceń, itd.) nie jest sprzężony z konkretnymi klasami poleceń ponieważ współpracuje z poleceniami za pośrednictwem interfejsu polecenia. Takie podejście pozwala wdrożyć nowe polecenia do aplikacji bez psucia istniejącego kodu.

```
1 // Bazowa klasa polecenie definiuje wspólny interfejs wszystkich
2 // konkretnych poleceń.
3 abstract class Command is
4     protected field app: Application
5     protected field editor: Editor
6     protected field backup: text
7
8     constructor Command(app: Application, editor: Editor) is
9         this.app = app
10        this.editor = editor
11
12    // Zrób kopię zapasową stanu edytora.
13    method saveBackup() is
14        backup = editor.text
15
16    // Przywróć stan edytora.
17    method undo() is
18        editor.text = backup
19
20    // Metoda wykonująca zadeklarowana jest jako abstrakcyjna,
21    // aby zmusić wszystkie konkretne polecenia do
22    // zaimplementowania jej we własnym zakresie. Metoda musi
23    // zwracać prawdę lub fałsz zależnie od tego, czy polecenie
24    // zmienia stan edytora lub pozostawia stan bez zmian.
25    abstract method execute()
```

```
26
27
28 // Tu umieszczane są konkretne polecenia.
29 class CopyCommand extends Command is
30     // Polecenie kopiuj nie jest zapisywane w historii ponieważ
31     // nie zmienia stanu edytora.
32     method execute() is
33         app.clipboard = editor.getSelection()
34         return false
35
36 class CutCommand extends Command is
37     // Polecenie wytnij zmienia stan edytora, dlatego trzeba
38     // zapisać stan w historii. Stan będzie zapisywany zawsze
39     // gdy metoda zwróci prawdę.
40     method execute() is
41         saveBackup()
42         app.clipboard = editor.getSelection()
43         editor.deleteSelection()
44         return true
45
46 class PasteCommand extends Command is
47     method execute() is
48         saveBackup()
49         editor.replaceSelection(app.clipboard)
50         return true
51
52 // Funkcja cofania operacji również jest polecienniem.
53 class UndoCommand extends Command is
54     method execute() is
55         app.undo()
56         return false
57
```

```
58
59 // Globalna historia poleceń jest po prostu stosem.
60 class CommandHistory is
61     private field history: array of Command
62
63     // Ostatni na wejściu...
64     method push(c: Command) is
65         // Wepchnij polecenie na koniec tablicy historii.
66
67     // ...pierwszy na wyjściu.
68     method pop():Command is
69         // Pobierz najświeższe polecenie z historii.
70
71
72 // Klasa edytora posiada narzędzia do edycji tekstu. Pełni rolę
73 // odbiorcy: wszystkie polecenia delegują faktyczne wykonanie
74 // metodom edytora.
75 class Editor is
76     field text: string
77
78     method getSelection() is
79         // Zwróć zaznaczony tekst.
80
81     method deleteSelection() is
82         // Skasuj zaznaczony tekst.
83
84     method replaceSelection(text) is
85         // Wstaw zawartość schowka w bieżącym miejscu.
86
87
88 // Klasa aplikacji ustanawia relacje między obiektami. Pełni
89 // rolę nadawcy: gdy trzeba coś zrobić, tworzy obiekt polecenia
```

```
90 // i uruchamia go.
91 class Application is
92     field clipboard: string
93     field editors: array of Editors
94     field activeEditor: Editor
95     field history: CommandHistory
96
97 // Kod przypisujący polecenia do obiektów interfejsu
98 // użytkownika może wyglądać w sposób następujący.
99 method createUI() is
100    // ...
101    copy = function() { executeCommand(
102        new CopyCommand(this, activeEditor)) }
103    copyButton.setCommand(copy)
104    shortcuts.onKeyPress("Ctrl+C", copy)
105
106    cut = function() { executeCommand(
107        new CutCommand(this, activeEditor)) }
108    cutButton.setCommand(cut)
109    shortcuts.onKeyPress("Ctrl+X", cut)
110
111    paste = function() { executeCommand(
112        new PasteCommand(this, activeEditor)) }
113    pasteButton.setCommand(paste)
114    shortcuts.onKeyPress("Ctrl+V", paste)
115
116    undo = function() { executeCommand(
117        new UndoCommand(this, activeEditor)) }
118    undoButton.setCommand(undo)
119    shortcuts.onKeyPress("Ctrl+Z", undo)
120
121 // Uruchom polecenie i sprawdź czy trzeba je zapisać w
```

```
122 // historii.  
123 method executeCommand(command) is  
124     if (command.execute)  
125         history.push(command)  
126  
127     // Weź najświeższe polecenie z historii i uruchom jego  
128     // metodę wycofującą. Zwróć uwagę, że nie znamy klasy tego  
129     // polecenia i nie musimy znać, bo polecenie samo wie jak  
130     // cofnąć rezultat swojego działania.  
131 method undo() is  
132     command = history.pop()  
133     if (command != null)  
134         command.undo()
```

## 💡 Zastosowanie

 **Zastosuj wzorzec Polecenie gdy chcesz parametryzować obiekty za pomocą działań.**

 Wzorzec Polecenie pozwala przekształcić wywołanie metody w samodzielny obiekt. Zmiana taka otwiera wiele ciekawych zastosowań: można przekazywać polecenia jako argumenty metody, przechowywać je w innych obiektach, zamieniać powiązane polecenia w trakcie działania programu, itp.

Oto przykład: pracujesz nad komponentem graficznego interfejsu użytkownika takim jak menu kontekstowe i chcesz aby użytkownicy mogli konfigurować elementy menu odpowiadające działaniom.

 **Wzorzec Polecenie pozwala układać kolejki zadań, ustalać harmonogram ich wykonania bądź uruchamiać je zdalnie.**

 Jak każdy inny obiekt, polecenie można serializować, co oznacza przekształcenie go w łańcuch znaków dający się łatwo zapisać w pliku lub bazie danych. Można później taki łańcuch znaków przywrócić do formy pierwotnego obiektu polecenia. Dzięki temu można opóźniać i ustalać harmonogram wykonywania poleceń. Co więcej, w taki sam sposób można kolejkować, notować w dzienniku lub wysyłać polecenia przez sieć.

 **Stosuj wzorzec Polecenie gdy chcesz zaimplementować operacje odwracalne.**

 Chociaż istnieje wiele sposobów na implementację funkcjonalności cofnij/ponów, wzorzec Polecenie jest prawdopodobnie najpopularniejszym.

Aby móc wycofywać działania, trzeba zaimplementować historię wykonanych działań. Historia poleceń jest stosem zawierającym wszystkie obiekty wykonanych poleceń wraz ze skojarzonymi z nimi kopiami zapasowymi stanu aplikacji.

Ta metoda ma dwie wady. Po pierwsze, zapisanie stanu aplikacji może nie być tak proste, gdyż część jej danych może być prywatna. Problem ten można obejść stosując wzorzec Pamiątka.

Po drugie, kopie zapasowe stanów mogą zużywać sporo pamięci RAM. Dlatego czasem można uciec się do alternatywnej implementacji: zamiast przywracać przeszły stan, można wykonać polecenie odwrotne. Takie polecenie również jednak miałoby swoją cenę: może okazać się trudne lub wręcz niemożliwe do zaimplementowania.

## Jak zaimplementować

1. Zadeklaruj interfejs polecenia z pojedynczą metodą uruchamiającą.
2. Dokonaj ekstrakcji żądań do konkretnych, odrębnych klas poleceń które implementują interfejs polecenia. Każda klasa powinna mieć zestaw pól służących przechowywaniu argumentów żądania wraz z odniesieniem do faktycznego obiektu odbiorcy. Wszystkie te wartości muszą być inicjalizowane za pośrednictwem konstruktora polecenia.
3. Zidentyfikuj klasy które będą pełnić rolę *nadawców*. Dodaj tym klasom pola służące przechowywaniu poleceń. Nadawcy powinni komunikować się z poleceniami wyłącznie za pośrednictwem interfejsu polecenia. Nadawcy na ogół nie tworzą obiektów polecenie sami, lecz otrzymują je od strony kodu klienta.
4. Zmień nadawców w taki sposób, aby uruchamiali polecenie zamiast wysyłania żądania bezpośrednio do odbiorcy.

5. Klient powinien inicjalizować obiekty w następującej kolejności:

- Tworzyć odbiorców.
- Tworzyć polecenia i kojarzyć je z odpowiednimi odbiorcami, jeśli istnieje potrzeba,
- Tworzyć nadawców i kojarzyć ich z konkretnymi poleceniami.

## Zalety i wady

- ✓ *Zasada pojedynczej odpowiedzialności.* Można rozprzegnać klasy wywołujące polecenia od klas faktycznie je wykonujących.
- ✓ *Zasada otwarte/zamknięte.* Można wprowadzić nowe polecenia do aplikacji bez psucia istniejącego kodu klienta.
- ✓ Pozwala zaimplementować cofnij/ponów.
- ✓ Pozwala zaimplementować opóźnione wykonywanie działań.
- ✓ Można złożyć zestaw prostszych poleceń w jedno skomplikowane.
- ✗ Kod może stać się bardziej skomplikowany gdyż wprowadzamy całą nową warstwę pomiędzy nadawcami a odbiorcami.

## ↔ Powiązania z innymi wzorcami

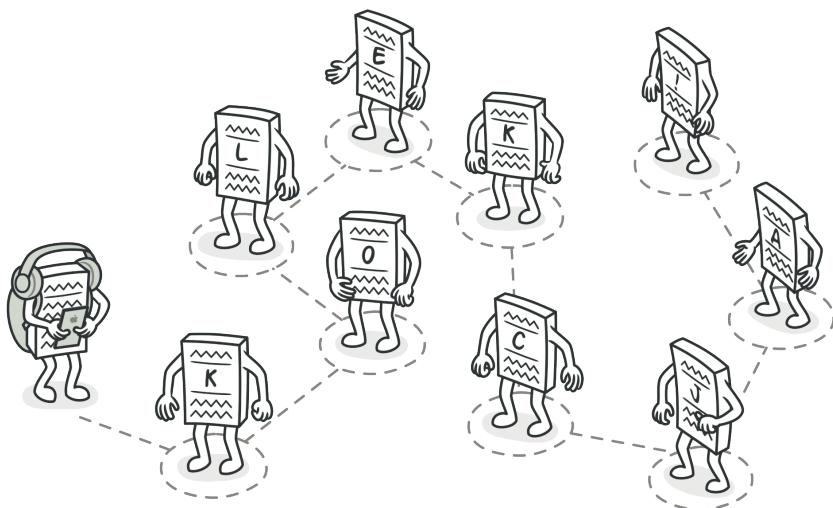
- Wzorce Łańcuch zobowiązań, Polecenie, Mediator i Obserwator dotyczą różnych sposobów na łączenie nadawców z odbiorcami żądań:
  - *Łańcuch zobowiązań* przekazuje żądanie sekwencyjnie wzdłuż dynamicznego łańcucha potencjalnych odbiorców, aż któryś z nich je obsłужy.
  - *Polecenie* pozwala nawiązywać jednokierunkowe połączenia pomiędzy nadawcami i odbiorcami.
  - *Mediator* eliminuje bezpośrednie połączenia pomiędzy nadawcami a odbiorcami, zmuszając ich do komunikacji za pośrednictwem obiektu mediator.
  - *Obserwator* pozwala odbiorcom dynamicznie zasubskrybować się i zrezygnować z subskrypcji żądań.
- Obsługujący w Łańcuchu zobowiązań mogą być zaimplementowani jako Polecenia. Można wówczas wykonać wiele różnych działań reprezentowanych jako żądania na tym samym obiekcie-kontekście.

Istnieje jednak jeszcze jedno podejście, według którego samo żądanie jest obiektem Polecenie. W takim przypadku możesz wykonać to samo działanie na łańcuchu różnych kontekstów.

- Można stosować Polecenie i Pamiątkę jednocześnie – implementując funkcjonalność “cofnij”. W takim przypadku, polecenie

nia są odpowiedzialne za wykonywanie różnych działań na obiekcie docelowym, zaś pamiętki służą zapamiętaniu stanu obiektu tuż przed wykonaniem polecenia.

- **Polecenie** i **Strategia** mogą wydawać się podobne, ponieważ oba mogą służyć parametryzacji obiektu jakimś działaniem. Mają jednak inne cele.
  - Za pomocą *Polecenia* można konwertować dowolne działanie na obiekt. Parametry działania stają się polami tego obiektu. Konwersja zaś pozwala odroczyć wykonanie działania, kolejkować je i przechowywać historię wykonanych działań, a także wysyłać polecenia zdalnym usługom, itd.
  - Z drugiej strony, *Strategia* zazwyczaj opisuje różne sposoby wykonywania danej czynności, pozwalając zamieniać algorytmy w ramach jednej klasy kontekstu.
- **Prototyp** może pomóc stworzyć historię, zapisując kopie **Polecen**.
- Wzorzec **Odwiedzający** można traktować jak późniejszą wersję **Polecenia**. Jego obiekty mogą wykonywać różne polecenia na obiektach różnych klas.



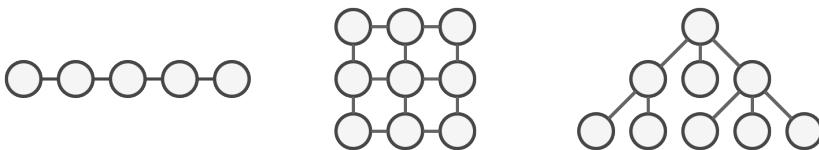
# ITERATOR

*Znany też jako: Kursor*

**Iterator** to behawioralny wzorzec projektowy, pozwalający sekwencyjnie przechodzić od elementu do elementu jakiegoś zbioru bez konieczności eksponowania jego formy (lista, stos, drzewo, itp.).

## (:() Problem

Kolekcje są jednym z najpopularniejszych typów danych wśród programistów, mimo że są to po prostu kontenery przechowujące grupy obiektów.



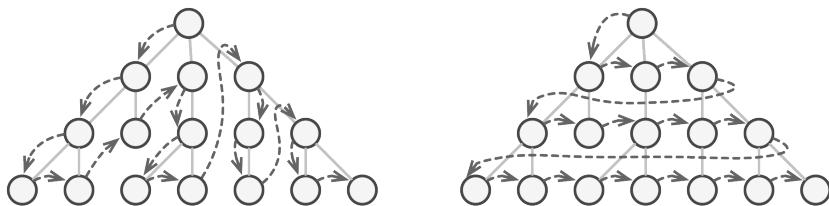
*Różne rodzaje kolekcji.*

Większość kolekcji przechowuje swoje elementy w prostych listach, ale niektóre są zbudowane w oparciu o stosy, drzewa, grafy i inne złożone struktury.

Jednak niezależnie od struktury kolekcji musi ona udostępniać dostęp do swoich elementów tak, aby można było używać ich gdzie indziej w programie. Powinien istnieć jakiś sposób przejścia po każdym elemencie kolekcji bez konieczności powtórzonego dostępu do jakiegoś elementu.

Wydaje się to łatwe, jeśli mamy do czynienia z kolekcją ustrukturowaną w formie listy. Tworzymy pętlę przechodzącą po elementach i gotowe. Ale jak przejść sekwencyjnie, element po elemencie, przez złożoną strukturę, jak np. drzewo? Na początku być może wystarczy poruszanie się w głąb, ale kolejnego dnia pojawi się wymóg przechodzenia wszerz. Jeszcze później

okazuje się, że przydałaby się możliwość losowego dostępu do dowolnego elementu drzewa.



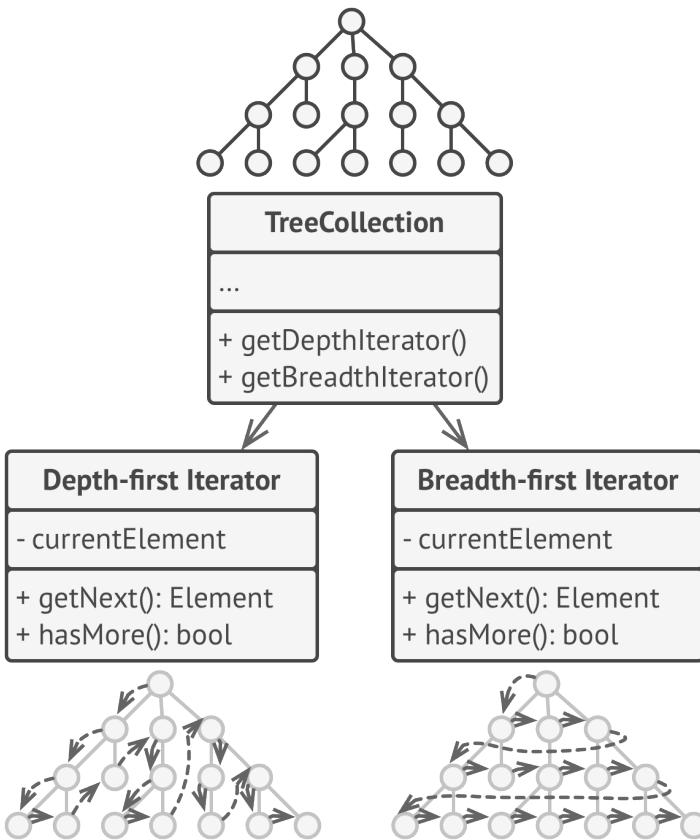
*Zawartość tej samej kolekcji można przejrzeć na wiele różnych sposobów.*

Dodawanie kolejnych algorytmów przechodzenia przez kolekcję stopniowo przyjmiewa jej główne zadanie, którym jest efektywne przechowywanie danych. Ponadto niektóre algorytmy mogą być zoptymalizowane pod kątem konkretnego zastosowania, więc włączanie ich do uogólnionej klasy kolekcji byłoby dziwne.

Z drugiej strony, kod klienta który ma za zadanie działać na różnych kolekcjach może nawet nie być zainteresowany sposobem w jaki przechowują one elementy. Jednak skoro wszystkie kolekcje udostępniają różne sposoby dostępu do swoich elementów, to nie ma innego wyjścia, niż związać swój kod z klasą konkretnej kolekcji.

## 😊 Rozwiążanie

Główną ideą wzorca Iterator jest ekstrakcja zadań związanych z przechodzeniem przez elementy kolekcji do osobnego obiektu zwanego *iteratorem*.



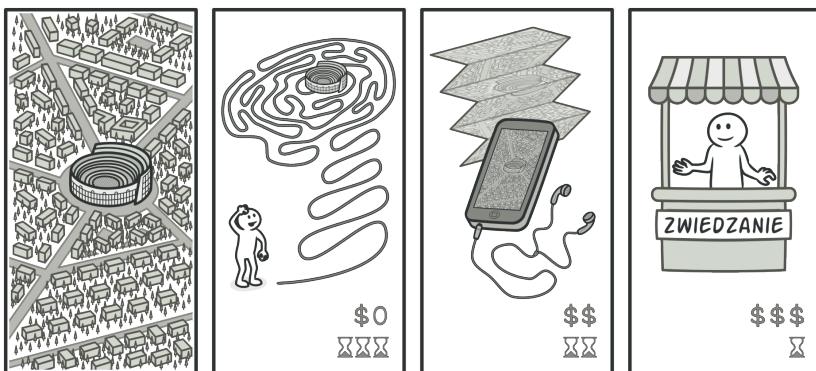
*Iteratory implementują różne algorytmy sekwencyjnego dostępu do kolejnych elementów. Wiele obiektów iteratora może przeskakiwać po elementach jednej kolekcji jednocześnie.*

Oprócz implementowania samego algorytmu, obiekt iteratora hermetyzuje wszystkie szczegóły sposobu przechodzenia przez kolejne elementy, jak bieżąca pozycja, czy ilość pozostałych elementów. Dzięki temu wiele iteratorów może jednocześnie przeglądać tę samą kolekcję, niezależnie od siebie.

Zazwyczaj, iteratory udostępniają jedną główną metodę pobierającą elementy kolekcji. Klient może wywoływać ją raz za razem aż przestanie ona zwracać kolejne obiekty, co oznacza osiągnięcie końca zbioru.

Wszystkie iteratory muszą implementować ten sam interfejs. Czyni to kod klienta kompatybilnym z dowolną kolekcją czy algorytmem przechodzenia o ile istnieje odpowiedni iterator. Jeśli potrzebny jest specjalny sposób przeglądania kolekcji, można stworzyć nową klasę iteratora, bez konieczności dokonywania zmian w kolekcji lub w kodzie klienta.

## 🚗 Analogia do prawdziwego życia



Różne sposoby zwiedzania Rzymu.

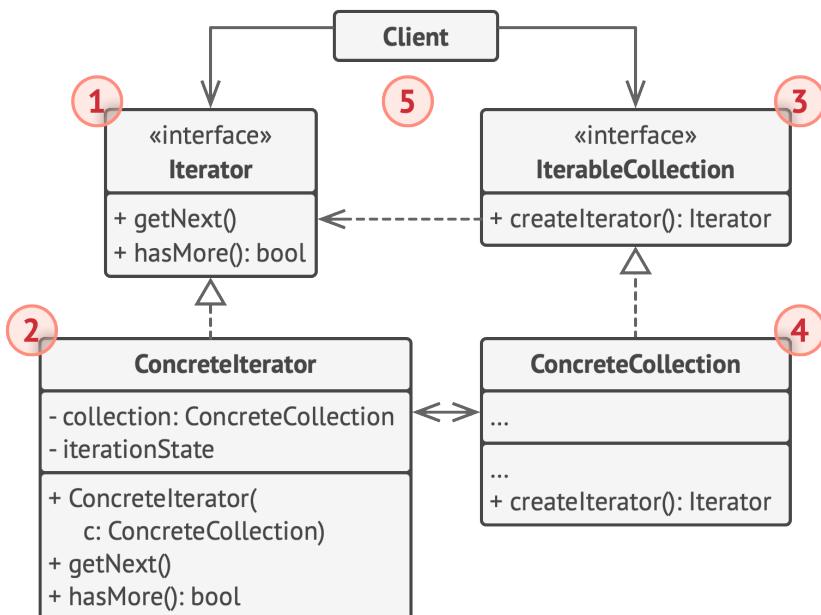
Zamierzasz odwiedzić Rzym na parę dni i zwiedzić wszystkie najważniejsze miejsca i atrakcje turystyczne. Ale na miejscu łatwo zmarnować sporo czasu chodząc w kółko, nie mogąc znaleźć nawet Koloseum.

Z drugiej strony, można kupić aplikację wirtualnego przewodnika na smartfon i nawigować z jej pomocą. Sprytne i niedrogie, a dodatkowo można zatrzymać się przy dowolnym miejscu na ile się chce.

Trzecia alternatywa to poświęcenie części swojego wycieczkowego budżetu na wynajęcie przewodnika który zna miasto jak własną kieszeń. Przewodnik mógłby dostosować trasę wycieczki do twoich preferencji, pokazać wszystko co najciekawsze i opowiedzieć wiele ciekawych historii. Byłoby miło, ale również i drożej.

Wszystkie te opcje – losowo obrane kierunki, smartfonowy przewodnik i wynajęty przewodnik – stanowią iteratory pozwalające na dostęp do wielkiej kolekcji widoków i atrakcji Rzymu.

## Struktura



1. Interfejs **Iterator** deklaruje działania niezbędne do sekwencyjnego przechodzenia przez elementy kolekcji: pobieranie kolejnego elementu, ustalenie bieżącej pozycji, powrót do pierwszego elementu, itp.
2. **Konkretne Iteratory** implementują specyficzne algorytmy przeglądania kolekcji. Obiekt iterator powinien samodzielnie śledzić postęp tego procesu. Pozwala to wielu iteratorom na jednoczesne przeglądanie tej samej kolekcji.
3. Interfejs **Kolekcja** deklaruje jedną lub więcej metod służących kompatybilności kolekcji z iteratorami. Zwracany typ metody

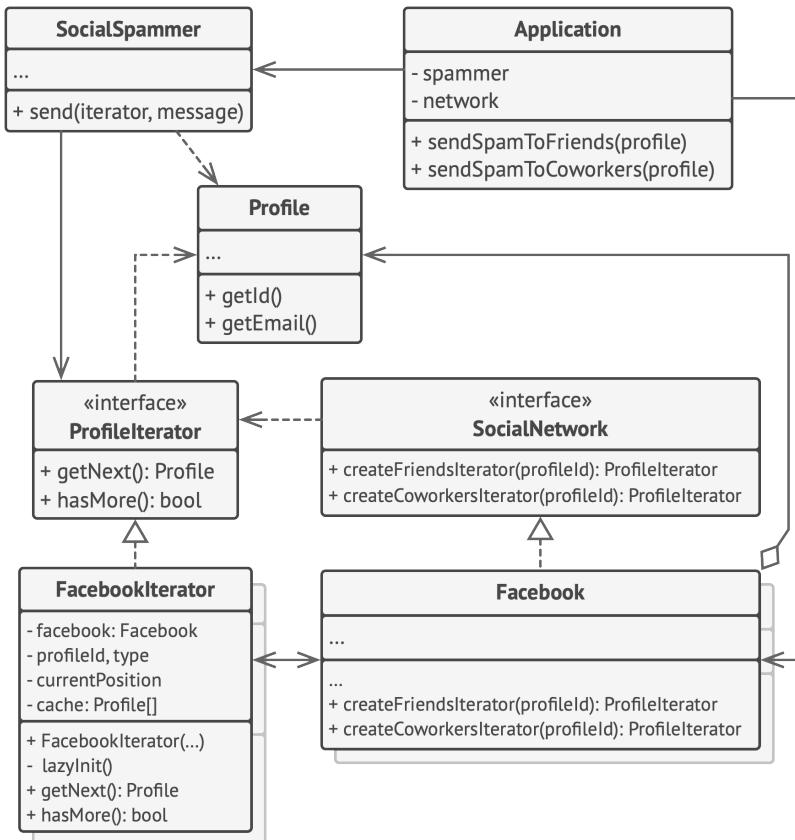
musi być zadeklarowany jako interfejs iterator, aby konkretne kolekcje mogły zwracać różne rodzaje iteratorów.

4. **Konkretnie Kolekcje** zwracają nowe instancje konkretnych klas iteratorów za każdym razem gdy klient ich zażąda. Pewnie ciekawi cię gdzie jest reszta kodu kolekcji? Nie martw się, powinien znajdować się w tej samej klasie. Po prostu te szczegółы nie są istotne dla konkretnego wzorca, więc je pomijamy.
5. **Klient** współpracuje zarówno z kolekcjami jak i iteratorami za pośrednictwem ich interfejsów. Dzięki temu nie jest sprzężony z konkretnymi klasami, pozwalając na pracę z różnymi kolekcjami i operatorami w tym samym kodzie klienta.

Zazwyczaj klienci nie tworzą iteratorów sami, lecz otrzymują je od kolekcji. Ale w pewnych przypadkach klient może stworzyć iterator bezpośrednio – na przykład gdy sam definiuje swój specjalny iterator.

## # Pseudokod

W poniższym przykładzie, wzorzec **Iterator** służy przejęciu przez specjalny typ kolekcji, który hermetyzuje dostęp do diagramu relacji pomiędzy profilami na Facebooku. Kolekcja udostępnia wiele iteratorów które mogą przeglądać kolejne profile na różne sposoby.



Przykład iteracji po kolejnych profilach użytkowników.

Iterator `przyjaciele` może służyć przeglądaniu zaprzyjaźnionych profili danej osoby. `Współpracownicy` robi to samo, ale pomija osoby które nie pracują wraz ze wskazaną osobą. Oba iteratory implementują wspólny interfejs który pozwala klientom pobierać profile bez zagłębiania się w szczegóły implementacyjne takie jak uwierzytelnianie, czy wysyłanie żądań REST.

Kod klienta nie jest sprzężony z konkretnymi klasami, ponieważ współpracuje z kolekcjami i iteratorami wyłącznie za pośrednictwem interfejsów. Jeśli postanowisz połączyć swoją aplikację z nowym portalem społecznościowym, musisz jedynie dostarczyć odpowiednie klasy kolekcji i iteratora, bez konieczności dokonywania zmian istniejącego kodu.

```
1 // Interfejs kolekcja musi deklarować metodę wytwarzającą do
2 // produkowania iteratorów. Można zadeklarować wiele metod jeśli
3 // potrzebujesz kilku różnych sposobów iterowania.
4 interface SocialNetwork is
5     method ProfileIterator createFriendsIterator(profileId)
6     method ProfileIterator createCoworkersIterator(profileId)
7
8
9 // Każda konkretna kolekcja jest powiązana z zestawem
10 // konkretnych klas iteratorów jakie zwraca. Ale klient nie jest
11 // z nimi powiązany, bo sygnatury tych metod zwracają typ
12 // interfejs iteratora.
13 class Facebook implements SocialNetwork is
14     // ... Większość kodu kolekcji powinna znaleźć się tutaj ...
15
16     // Kod kreacyjny iteratora.
17     method ProfileIterator createFriendsIterator(profileId) is
18         return new FacebookIterator(this, profileId, "friends")
19     method ProfileIterator createCoworkersIterator(profileId) is
20         return new FacebookIterator(this, profileId, "coworkers")
21
22
23 // Wspólny interfejs wszystkich iteratorów.
24 interface ProfileIterator is
```

```
25  method getNext():Profile
26  method hasMore():bool
27
28
29 // Konkretna klasa iteratora.
30 class FacebookIterator implements ProfileIterator is
31     // Iterator potrzebuje odniesienia do kolekcji po elementach
32     // której ma przechodzić.
33     private field facebook: Facebook
34     private field profileId, type: string
35
36     // Obiekt iteratora przechodzi po elementach kolekcji
37     // niezależnie od innych iteratorów. Dlatego musi
38     // przechowywać stan iteracji.
39     private field currentPosition
40     private field cache: array of Profile
41
42 constructor FacebookIterator(facebook, profileId, type) is
43     this.facebook = facebook
44     this.profileId = profileId
45     this.type = type
46
47 private method lazyInit() is
48     if (cache == null)
49         cache = facebook.socialGraphRequest(profileId, type)
50
51 // Każda konkretna klasa iteratora posiada własną
52 // implementację wspólnego interfejsu iteratora.
53 method getNext() is
54     if (hasMore())
55         currentPosition++
56         return cache[currentPosition]
```

```
57
58     method hasMore() is
59         lazyInit()
60         return currentPosition < cache.length
61
62
63 // Oto kolejna użyteczna sztuczka: możesz przekazać iterator
64 // klasie klienckiej zamiast dawać jej dostęp do całej kolekcji.
65 // Dzięki temu nie trzeba eksponować całej kolekcji klientowi.
66 //
67 // Kolejna zaleta takiego podejścia to możliwość zmiany sposobu
68 // w jaki klient współpracuje z kolekcją w trakcie działania
69 // programu poprzez przekazanie klientowi innego iteratora. Jest
70 // to możliwe ponieważ kod klienta nie jest sprzęgnięty z
71 // konkretnymi klasami iteratora.
72 class SocialSpammer is
73     method send(iterator: ProfileIterator, message: string) is
74         while (iterator.hasMore())
75             profile = iterator.getNext()
76             System.sendEmail(profile.getEmail(), message)
77
78
79 // Klasa aplikacji konfiguruje kolekcje i iteratory, a następnie
80 // przekazuje je kodowi klienta.
81 class Application is
82     field network: SocialNetwork
83     field spammer: SocialSpammer
84
85     method config() is
86         if working with Facebook
87             this.network = new Facebook()
88         if working with LinkedIn
```

```
89     this.network = new LinkedIn()
90     this.spammer = new SocialSpammer()
91
92     method sendSpamToFriends(profile) is
93         iterator = network.createFriendsIterator(profile.getId())
94         spammer.send(iterator, "Very important message")
95
96     method sendSpamToCoworkers(profile) is
97         iterator = network.createCoworkersIterator(profile.getId())
98         spammer.send(iterator, "Very important message")
```

## 💡 Zastosowanie

- ⚡ Stosuj wzorzec Iterator gdy kolekcja z którą masz do czynienia posiada skomplikowaną strukturę, ale zależy ci na ukryciu jej przed klientem (dla wygody, lub dla bezpieczeństwa).
- ⚡ Iterator hermetyzuje szczegóły współpracy ze złożonymi strukturami danych, dając klientowi pewną liczbę prostych metod służących dostępowi do elementów kolekcji. To podejście jest dla klienta wygodne, ale również chroni kolekcję przed nieuwaznym lub zlosliwym dzialaniem, ktorego ryzyko istnieje przy bezporedniej pracy ze strukturą.
- ⚡ Stosuj wzorzec w celu redukcji duplikowania kodu przeglądzania elementów zbiorów na przestrzeni całego programu.
- ⚡ Kod nietrywialnych algorytmów iteracji bywa obszerny. Gdy umieści się go w ramach logiki biznesowej aplikacji, zwykle

zaciera główną odpowiedzialność pierwotnego kodu i czyni go trudniejszym do utrzymania. Przeniesienie kodu przeglądania elementów do stosownych iteratorów pomaga uczynić kod aplikacji czystszy i prostszy.

 **Stosuj Iterator gdy chcesz, aby twój kod był w stanie przeglądać elementy różnych struktur danych, lub gdy nie znasz z góry szczegółów ich struktury.**

 Wzorzec Iterator udostępnia parę ogólnych interfejsów zarówno kolekcji, jak i iteratorów. Skoro twój kod korzysta z tych interfejsów, to będzie nadal działał, nawet gdy przekażesz mu różne rodzaje kolekcji i iteratorów, o ile implementują te interfejsy.

## Jak zaimplementować

1. Zadeklaruj interfejs iteratora. W najprostszym przypadku musi posiadać metodę pobierającą kolejny element kolekcji. Ale dla wygody możesz dodać parę innych metod, np. do pobierania poprzedniego elementu, śledzenia bieżącej pozycji i ustalenia końca procesu iteracji.
2. Zadeklaruj interfejs kolekcji i opisz metodę pobierającą iteratory. Typ zwracany przez metodę powinien być zgodny z interfejsem iteratora. Możesz zadeklarować podobne metody, jeśli zamierzasz mieć wiele różnych grup iteratorów.

3. Zaimplementuj konkretne klasy iterator dla kolekcji które powinny być możliwe do przeglądania za pomocą iteratorów. Obiekt iterator musi być powiązany z jedną instancją kolekcji. Zazwyczaj tworzy się takie powiązanie w konstruktorze iteratora.
4. Zaimplementuj interfejs kolekcji w swoich klasach kolekcji. Główną ideą jest udostępnienie klientowi skrótu do tworzenia iteratorów optymalnych dla danej klasy kolekcji. Obiekt kolekcji musi przekazywać siebie samego do konstruktora iteratora aby stworzyć między nimi powiązanie.
5. Przejrzyj swój kod klienta i zamień cały kod przeglądania kolekcji na wykorzystujący iteratory. Klient pobiera nowy obiekt iteratora za każdym razem gdy chce przejrzeć zawartość kolekcji.

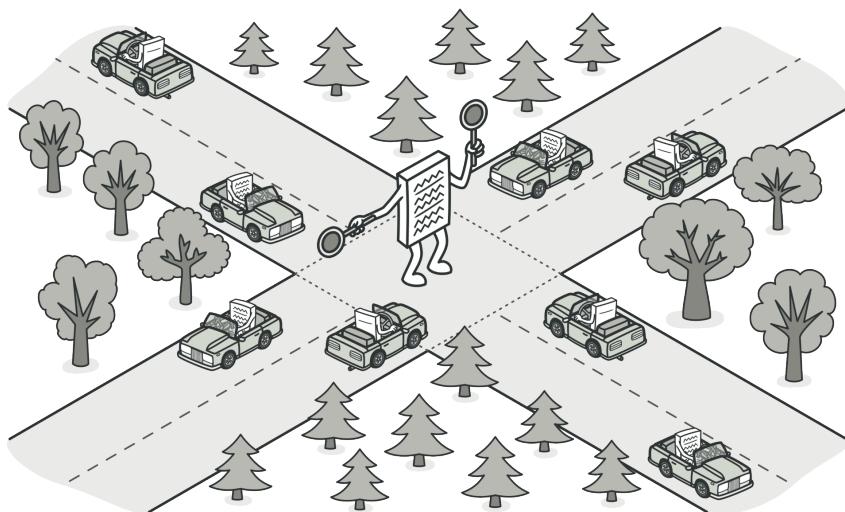
## Zalety i wady

- ✓ *Zasada pojedynczej odpowiedzialności.* Można uprzątnąć kod klienta i kolekcje, ekstrahując obszerny kod przeglądania do osobnych klas.
- ✓ *Zasada otwarte/zamknięte.* Można zaimplementować nowe typy kolekcji i iteratorów oraz przekazywać je do istniejącego kodu bez psucia czegokolwiek.
- ✓ Można przeglądać tę samą kolekcję równolegle wieloma iteratorami, gdyż każdy z nich przechowuje informacje o swoim stanie.

- ✓ Z powyższego powodu można opóźniać iterację i kontynuować ją gdy zachodzi taka potrzeba.
- ✗ Zastosowanie tego wzorca będzie przesadą jeśli Twoja aplikacja korzysta wyłącznie z prostych kolekcji.
- ✗ Używanie iteratora może być mniej efektywne niż bezpośrednie przejście po elementach jakiejś wyspecjalizowanej kolekcji.

## ↔ Powiązania z innymi wzorcami

- **Iteratory** służą do sekwencyjnego przemieszczania się po drzewie **Kompozytowym** element po elemencie.
- Możesz zastosować **Metodę wytwórczą** wraz z **Iteratorem** aby pozwolić podklasom kolekcji zwracać różne typy iteratorów kompatybilnych z kolekcją.
- Można zastosować **Pamiątkę** wraz z **Iteratorem** by zapisać bieżący stan iteracji, co pozwoli w razie potrzeby do niego powrócić.
- Połączenie **Odwiedzającego** z **Iteratorem** może służyć sekwenциальнemu przeglądowi elementów złożonej struktury danych i wykonaniu na nich jakieś działania, nawet jeśli te elementy są obiektami różnych klas.



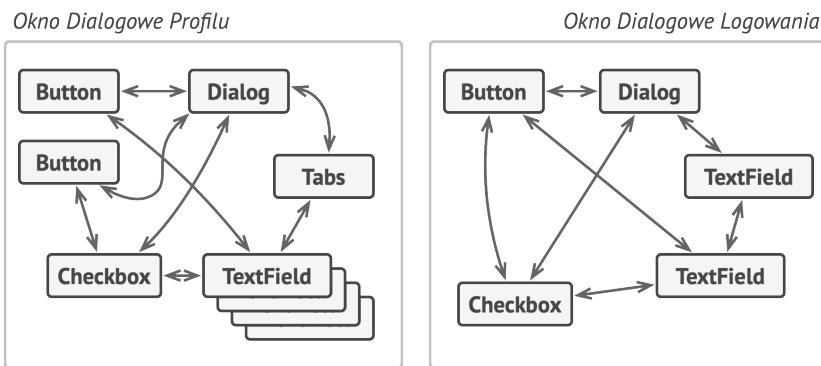
# MEDIATOR

Znany też jako: *Intermediary, Controller*

**Mediator** to behawioralny wzorzec projektowy pozwalający zredukować chaos zależności pomiędzy obiektami. Wzorzec ten ogranicza bezpośrednią komunikację pomiędzy obiektami i zmusza je do współpracy wyłącznie za pośrednictwem obiektu mediatora

## :( Problem

Załóżmy, że masz okno dialogowe służące tworzeniu i edycji profili klientów. Składa się z różnych kontrolek, takich jak pola tekstowe, pola wyboru, przyciski, itd.



*Wraz z ewolucją aplikacji, powiązania między elementami interfejsu użytkownika mogą stać się coraz bardziej chaotyczne.*

Niektóre elementy formularza mogą współdziałać z innymi. Przykładowo, zaznaczenie pola wyboru “Mam psa” spowoduje pojawienie się ukrytego wcześniej pola tekstowego do wpisywania imienia. Inny przykład to przycisk “Wyślij” który musi dokonać walidacji danych w polach zanim je zapisze.



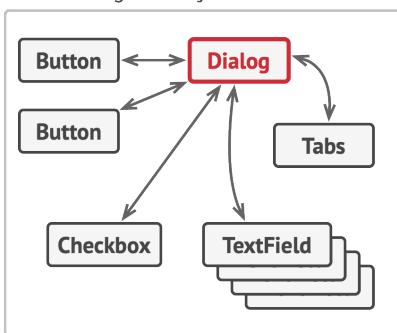
*Elementy mogą mieć wiele relacji z innymi. W związku z tym zmiany jednych wpływają też na inne.*

Implementacja tej logiki bezpośrednio w kodzie elementów formularza sprawi, że klasy elementów będzie trudno ponownie użyć w innych formularzach. Przykładowo, nie będzie można użyć klasy pola wyboru w innym formularzu, ponieważ jest sprzężony z polem tekstowym imienia psa. Można albo użyć wszystkie klasy biorące udział w renderowaniu formularza profilu, albo nie używać żadnych.

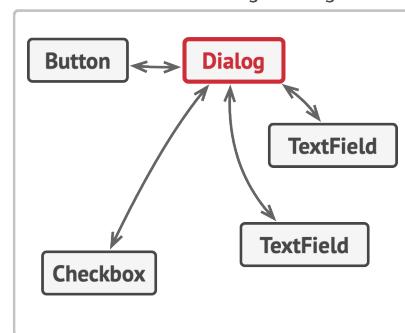
## Rozwiążanie

Wzorzec Mediator sugeruje przerwanie bezpośredniej komunikacji między komponentami które mają być niezależne. W zamian, komponenty te muszą współpracować pośrednio, wywołując specjalny obiekt mediatora, który przekierowuje wywołania do odpowiednich komponentów. W wyniku tego komponenty zależą tylko od pojedynczej klasy mediatora, zamiast sprzężenia ze sobą nawzajem.

Okno Dialogowe Profilu



Okno Dialogowe Logowania



*Elementy interfejsu użytkownika powinny komunikować się pośrednio, poprzez obiekt mediator.*

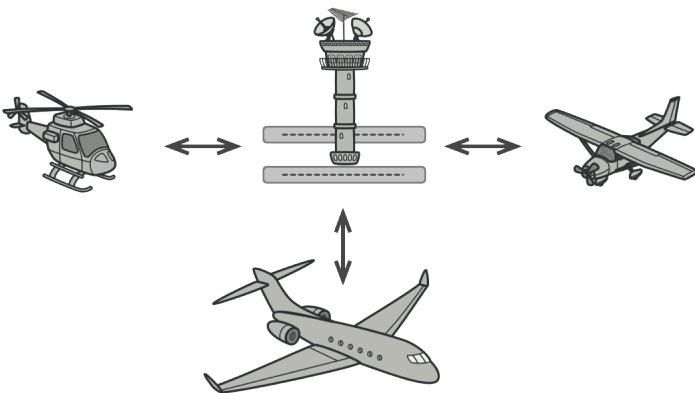
W naszym przykładzie z formularzem edycji profilu, klasa okna dialogowego może pełnić rolę mediatora. Prawdopodobnie klasa ta wie już o swoich podelementach, więc nie trzeba nawet wprowadzać nowych zależności do tej klasy.

Najistotniejsza zmiana dotyczy samych elementów formularza. Rozważmy przycisk wysyłania. Wcześniej, za każdym razem gdy kliknięto przycisk, musiał on dokonać walidacji wartości we wszystkich elementach formularza. Teraz jego jedynym zadaniem jest powiadomienie okna dialogowego o kliknięciu. Otrzymawszy powiadomienie, okno dialogowe dokonuje walidacji lub przekazuje to zadanie poszczególnym elementom formularza. Tym samym, zamiast sprzężenia z wieloma elementami, przycisk zależy jedynie od klasy okna dialogowego.

Można pójść o krok dalej i rozluźnić powiązanie jeszcze bardziej, ekstrahując wspólny interfejs dla wszystkich typów okien dialogowych. Taki interfejs deklarowałby metodę powiadamiania, za pomocą której wszystkie elementy formularza mogłyby powiadamiać okno dialogowe o zdarzeniach z nimi związanych. Dzięki temu przycisk wysyłania będzie mógł współdziałać z dowolnym oknem dialogowym implementującym ten interfejs.

Jest to sposób w jaki wzorzec Mediator pozwala hermetyzować złożone płatniny relacji pomiędzy obiektami w jednym obiekcie. Im mniej zależności ma klasa, tym łatwiej ją modyfikować, rozszerzać lub użyć ponownie.

## Analoga do prawdziwego życia

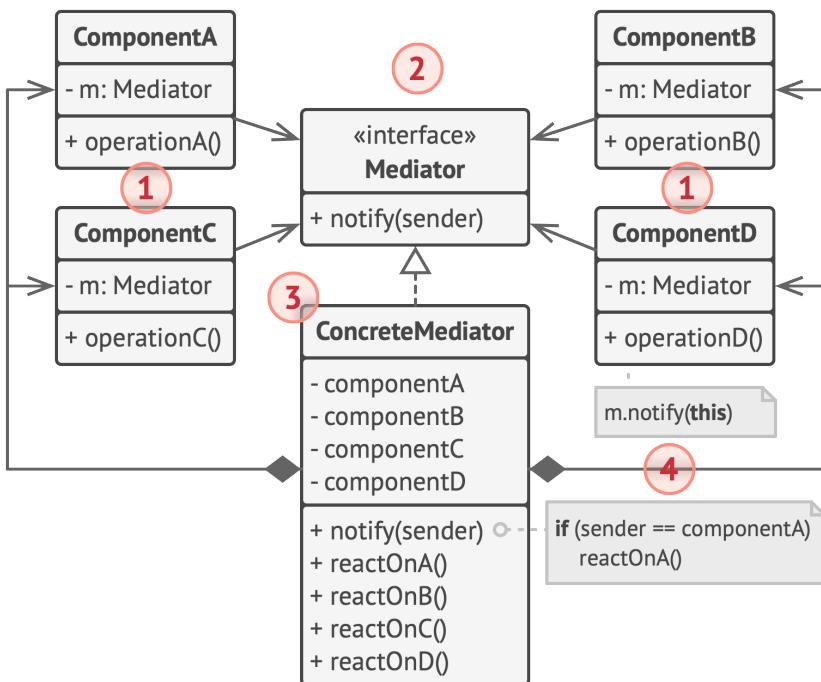


*Piloci statków powietrznych nie rozmawiają ze sobą nawzajem bezpośrednio, gdy ustalają kto następny będzie lądował. Cała komunikacja odbywa się za pośrednictwem wieży kontroli lotów.*

Piloci statków powietrznych zbliżający się do lotniska lub je opuszczający nie rozmawiają ze sobą nawzajem, lecz za pośrednictwem kontrolera lotów, siedzącego w wieży z widokiem na lądowisko. Bez kontrolera, piloci musieliby wiedzieć o każdym samolocie lub śmigłowcu w okolicy lotniska, dyskutować na temat pierwszeństwa lądowania. Mogłoby to niekorzystnie wpłynąć na statystyki bezpieczeństwa...

Wieża nie musi kontrolować całego lotu. Jej funkcja służy nakładaniu ograniczeń w obszarze terminala aby żaden z pilotów nie był przytłoczony dużą ilością aktorów biorących udział w funkcjonowaniu lotniska.

## Struktura



1. **Komponenty** to różne klasy zawierające jakąś logikę biznesową. Każdy komponent posiada odniesienie do mediatora, zadeklarowany jako typ interfejsu mediatora. Komponent ten nie jest świadom faktycznej klasy obiektu mediatora, więc można go używać ponownie w innych programach, łącząc z innym mediatorem.
2. Interfejs **Mediator** deklaruje metody komunikacji z komponentami, które na ogół ograniczają się do jednej metody powiadamiania. Komponenty mogą przekazywać dowolny kontekst jako argument tej metody, włącznie z samymi sobą, ale wyłącz-

nie w sposób który nie spowoduje sprzęgnięcia komponentu otrzymującego z klasą nadawcy.

3. **Konkretni Mediatorzy** hermetyzują relacje pomiędzy różnora-kimi komponentami. Konkretni mediatorzy często przechowu-ją odniesienia do wszystkich komponentów jakimi zarządzają i czasem nawet zarządzają ich cyklem życia.
4. Komponenty nie mogą być świadome innych komponentów. Jeśli coś istotnego zdarzy się w innym komponencie, musi on powiadamiać wyłącznie mediatora. Gdy zaś ten otrzyma po-wiadomienie, może w prosty sposób zidentyfikować nadawcę i to czasem wystarcza, by zdecydować jaki komponent urucho-mić w odpowiedzi.

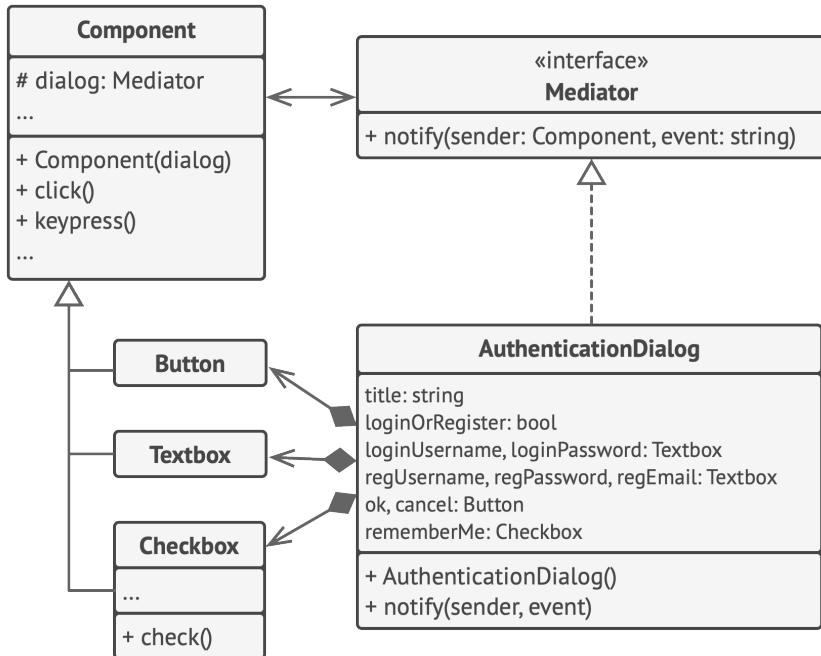
Z perspektywy komponentu, wszystko przypomina czarną skrzynkę. Nadawca nie wie kto ostatecznie obsłuży jego żąda-nie, a odbiorca nie wie kto je nadał.

## # Pseudokod

W poniższym przykładzie, wzorzec **Mediator** pomaga wyeli-minować wzajemne zależności pomiędzy różnymi klasami UI: przyciskami, polami wyboru i etykietami tekstowymi.

Element, uruchomiony przez użytkownika, nie komunikuje się bezpośrednio z innymi elementami, nawet jeśli wydaje się, że mógłby. W zamian element musi dać znać o zdarzeniu tylko

swojemu mediatorowi, przekazując ewentualne dane kontekstowe wraz z powiadomieniem.



Struktura klas okna dialogowego.

W tym przykładzie, całe okno dialogowe uwierzytelniania pełni rolę mediatora. Wie jak konkretne elementy powinny współpracować i zapewnia im pośrednią komunikację ze sobą. Otrzymawszy powiadomienie o zdarzeniu, okno dialogowe decyduje który element powinien zająć się jego obsługą i odpowiednio przekierowuje wywołanie.

```
1 // Interfejs mediatora deklaruje metodę za pomocą której
2 // komponenty powiadamiają go o różnych zdarzeniach. Mediator
3 // może zareagować na te zdarzenia i przekazać wykonanie innym
4 // komponentom.
5 interface Mediator is
6     method notify(sender: Component, event: string)
7
8
9 // Konkretna klasa mediator. Splątana sieć połączeń pomiędzy
10 // poszczególnymi komponentami została uporządkowana i
11 // przeniesiona do mediatora.
12 class AuthenticationDialog implements Mediator is
13     private field title: string
14     private field loginOrRegisterChkBx: Checkbox
15     private field loginUsername, loginPassword: Textbox
16     private field registrationUsername, registrationPassword,
17             registrationEmail: Textbox
18     private field okBtn, cancelBtn: Button
19
20 constructor AuthenticationDialog() is
21     // Utwórz wszystkie obiekty-komponenty i przekaż bieżący
22     // mediator ich konstruktorom by ustawić połączenia.
23
24     // Gdy coś się zdarzy komponentowi, poinformuje on o tym
25     // mediatora. Mediator otrzymawszy powiadomienie może coś
26     // zrobić, lub przekazać żądanie innemu komponentowi.
27 method notify(sender, event) is
28     if (sender == loginOrRegisterChkBx and event == "check")
29         if (loginOrRegisterChkBx.checked)
30             title = "Log in"
31             // 1. Pokaż komponenty formularza logowania.
32             // 2. Ukryj komponenty formularza rejestracji.
```

```
33     else
34         title = "Register"
35         // 1. Pokaż komponenty formularza rejestracji.
36         // 2. Ukryj komponenty formularza logowania.
37
38     if (sender == okBtn && event == "click")
39         if (loginOrRegister.checked)
40             // Spróbuj znaleźć użytkownika po
41             // poświadczeniach.
42         if (!found)
43             // Pokaż komunikat błędu nad polem nazwy
44             // użytkownika.
45     else
46         // 1. Utwórz konto użytkownika korzystając z
47         // danych w polach formularza rejestracji.
48         // 2. Zaloguj użytkownika.
49         // ...
50
51 // Komponenty współpracują z mediatorem za pośrednictwem
52 // interfejsu mediatora. Dzięki temu można używać tych samych
53 // komponentów w różnych kontekstach poprzez łączenie ich z
54 // różnymi obiektami mediator.
55 class Component is
56     field dialog: Mediator
57
58 constructor Component(dialog) is
59     this.dialog = dialog
60
61 method click() is
62     dialog.notify(this, "click")
63
64 method keypress() is
```

```
65     dialog.notify(this, "keypress")
66
67 // Konkretne komponenty nie komunikują się ze sobą. Mają tylko
68 // jeden kanał komunikacyjny, którym przesyłają powiadomienia
69 // mediatorowi.
70 class Button extends Component is
71     // ...
72
73 class Textbox extends Component is
74     // ...
75
76 class Checkbox extends Component is
77     method check() is
78         dialog.notify(this, "check")
79     // ...
```

## 💡 Zastosowanie

- ⚡ **Stosuj wzorzec Mediator gdy zmiana jakichś klas jest trudna z powodu ścisłego sprzegnięcia z innymi klasami.**
- ⚡ **Wzorzec pozwala wyekstrahować wszystkie relacje pomiędzy klasami do osobnej klasy, izolując ewentualne zmiany określonego komponentu od reszty komponentów.**
- ⚡ **Stosuj ten wzorzec gdy nie możesz ponownie użyć jakiegoś komponentu w innym programie, z powodu zbytniej jego zależności od innych komponentów.**

- ⚡ Po zastosowaniu wzorca Mediator, pojedyncze komponenty stają się nieświadome innych komponentów. Nadal mogą komunikować się ze sobą, ale pośrednio – poprzez obiekt mediator. Aby ponownie użyć komponent w innej aplikacji, musisz zapewnić mu nową klasę mediator.
- ⚠ **Stosuj wzorzec Mediator gdy zauważysz, że tworzysz mnóstwo podklas komponentu tylko aby móc ponownie użyć jakieś zachowanie w innych kontekstach.**
- ⚡ Skoro wszystkie relacje pomiędzy komponentami zawierają się w obrębie mediatora, łatwo zdefiniować całkowicie nowe metody współpracy tych komponentów wprowadzając nowe klasy mediatorów, bez konieczności zmian w samych komponentach.

## Jak zaimplementować

1. Zidentyfikuj grupę ścisłe sprężonych klas które zyskałyby na niezależności (np. dla łatwiejszego utrzymania lub ponownego użycia).
2. Zadeklaruj interfejs mediatora i określ potrzebny protokół komunikacji pomiędzy mediatorami i innymi komponentami. W większości przypadków, wystarczy pojedyncza metoda otrzymywania powiadomień od komponentów.

Taki interfejs jest kluczowy gdy chcemy ponownie wykorzystać klasy komponentów w innych kontekstach. O ile komponent

współpracuje ze swoim mediatorem za pośrednictwem ogólnego interfejsu, można łączyć komponent z innymi implementacjami mediatora.

3. Zaimplementuj konkretną klasę mediator. Najlepiej, gdyby klasa ta przechowywała odniesienia do wszystkich komponentów jakimi zarządza.
4. Można nawet pójść o krok dalej i uczynić mediatora odpowiedzialnym za tworzenie i niszczenie obiektów komponentów. Wówczas mediator zacznie przypominać **fabrykę** lub **fasadę**.
5. Komponenty powinny przechowywać odniesienie do obiektu mediatora. Połączenie zwykle nawija się w konstruktorze komponentu, do którego obiekt mediatora przekazywany jest w roli argumentu.
6. Zmień kod komponentów tak, aby wywoływały metodę powiadamiania mediatora zamiast metod powiadamiania innych komponentów. Wyekstrahuj kod zawierający wywołania do innych komponentów do klasy mediatora. Wykonuj ten kod gdy mediator otrzyma powiadomienie od komponentu.

## ΔΔ Zalety i wady

- ✓ *Zasada pojedynczej odpowiedzialności.* Możesz wyekstrahować komunikację pomiędzy różnymi komponentami w jedno miejsce, czyniąc ją łatwiejszą do zrozumienia i utrzymania.

- ✓ *Zasada otwarte/zamknięte.* Można wprowadzać kolejnych mediatorów bez konieczności zmiany samych komponentów.
  - ✓ Można zredukować sprzężenie pomiędzy różnymi komponentami programu.
  - ✓ Można ułatwić ponowne wykorzystanie komponentów.
- ✗ Z czasem mediator może ewoluować do postaci **Boskiego Obiektu**.

## ↔ Powiązania z innymi wzorcami

- Wzorce **Łańcuch zobowiązań**, **Polecenie**, **Mediator** i **Obserwator** dotyczą różnych sposobów na łączenie nadawców z odbiorcami żądań:
  - *Łańcuch zobowiązań* przekazuje żądanie sekwencyjnie wzdłuż dynamicznego łańcucha potencjalnych odbiorców, aż któryś z nich je obsłужy.
  - *Polecenie* pozwala nawiązywać jednokierunkowe połączenia pomiędzy nadawcami i odbiorcami.
  - *Mediator* eliminuje bezpośrednie połączenia pomiędzy nadawcami a odbiorcami, zmuszając ich do komunikacji za pośrednictwem obiektu mediator.
  - *Obserwator* pozwala odbiorcom dynamicznie zasubskrybować się i zrezygnować z subskrypcji żądań.

- **Fasada** i **Mediator** mają podobne zadania: służą zorganizowaniu współpracy pomiędzy wieloma ściśle spręgniętymi klasami.
    - *Fasada* definiuje uproszczony interfejs podsystemu obiektów, ale nie wprowadza nowej funkcjonalności. Podsystem jest nieświadomy istnienia fasady. Obiekty w obrębie podsystemu mogą komunikować się bezpośrednio.
    - *Mediator* centralizuje komunikację pomiędzy komponentami podsystemu. Komponenty wiedzą tylko o obiekcie mediator i nie komunikują się ze sobą bezpośrednio.
  - Różnica pomiędzy **Mediatorem** a **Obserwatorem** jest często trudna do uchwycenia. W większości przypadków można implementować je zamiennie, a czasem jednocześnie. Zobaczmy, jak by to wyglądało.
- Głównym celem *Mediatora* jest eliminacja wzajemnych zależności pomiędzy zestawem komponentów systemu. Zamiast tego uzależnia się te komponenty od jednego obiektu-mediatora. Celem *Obserwatora* jest ustanowienie dynamicznych, jednokierunkowych połączeń między obiektami, których część jest podległa innym.

Istnieje popularna implementacja wzorca Mediator która bazuje na *Obserwatorze*. Obiekt mediatora pełni w niej rolę publikującego, zaś komponenty są subskrybentami mogącymi “prenumerować” zdarzenia które nadaje mediator. Gdy *Media-*

*tor* jest zaimplementowany w ten sposób, może przypominać *Obserwatora*.

Jeśli nie jest to zrozumiałe, warto przypomnieć sobie, że można zaimplementować wzorzec Mediator na inne sposoby. Na przykład można na stałe powiązać wszystkie komponenty z tym samym obiektem mediator. Taka implementacja nie będzie przypominać *Obserwatora*, ale nadal będzie instancją wzorca Mediator.

A teraz wyobraźmy sobie program w którym wszystkie komponenty stały się publikującymi, pozwalając na dynamiczne połączenia pomiędzy sobą. Nie będzie wówczas skoncentrowanego obiektu mediatora, a tylko rozproszony zestaw obserwatorów.



# PAMIĄTKA

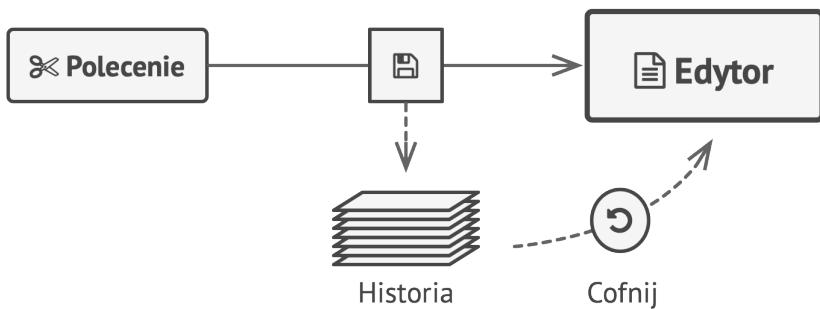
Znany też jako: *Snapshot, Memento*

**Pamiątka** to behawioralny wzorzec projektowy pozwalający zapisywać i przywracać wcześniejszy stan obiektu bez ujawniania szczegółów jego implementacji.

## Problem

Wyobraź sobie, że tworzysz edytor tekstu. Poza zwykłym edytowaniem treści, edytor może ją formatować, wstawiać obrazki, itd.

W jakimś momencie postanawiasz pozwolić użytkownikom cofać dowolną operację wykonaną na tekście. Funkcja taka stała się przez lata popularna, a użytkownicy oczekują jej w każdej aplikacji. W celu implementacji obierasz podejście bezpośrednie. Przed wykonaniem dowolnego działania, aplikacja zapamiętuje stan wszystkich obiektów i zapisuje go w jakimś magazynie. Gdy użytkownik zechce wycofać jakąś zmianę, aplikacja pobiera ostatnią migawkę z historii i za jej pomocą przywraca wcześniejszy stan wszystkich obiektów.

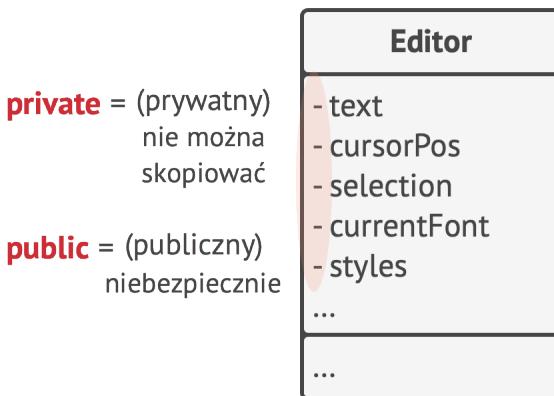


*Przed wykonaniem działania, aplikacja zapisuje migawkę stanu obiektów. Pozwoli ona później przywrócić obiekty do ich poprzedniego stanu.*

Pomyślmy o tych migawkach. Jak dokładnie je tworzyć? Prawdopodobnie trzeba byłoby przejrzeć wszystkie pola obiektu, skopiować ich wartości i zapisać w jakiś magazynie. Jed-

nak to zadziałałoby tylko jeśli obiekt ma stosunkowo luźne ograniczenia dostępu do swojej zawartości. Niestety, większość prawdziwych obiektów nie pozwoli obcym tak łatwo zajrzeć w swoją zawartość i najistotniejsze dane będą ukryte w polach prywatnych.

Na razie zignorujmy jednak ten problem i założmy, że nasze obiekty są niczym hipisi: wolą otwarte związki i nie ukrywają swojej natury. Chociaż podejście takie pomija powyższe ograniczenie i pozwala wykonać migawkę stanu obiektów, to tworzy przy okazji inne problemy. W przeszłości bowiem możemy zdecydować o refaktoryzacji niektórych klas edytora, lub dodać, bądź usunąć pola. Brzmi łatwo, ale wymagałoby przy okazji zmiany klas odpowiedzialnych za kopiowanie stanów zmienianych obiektów.



*Jak skopiować prywatną część stanu obiektu?*

Ale jest coś jeszcze. Rozważmy samą “migawkę” stanu edytora. Jakie dane zawierałaby? W najprostszej formie: tekst, współ-

rzędne kursora, bieżącą pozycję przewijania, itd. Aby wykonać migawkę, trzeba zebrać te wartości i umieścić je w jakimś kontenerze.

Najprawdopodobniej będzie trzeba przechowywać mnóstwo takich obiektów kontenerowych w formie listy reprezentującej historię. Dlatego też kontenery zapewne byłyby obiektami jednej klasy. Klasa ta nie miałaby prawie żadnej metody, ale za to wiele pól, odzwierciedlających stan edytora. Aby pozwolić innym obiektom zapisywać i odczytywać dane migawki, trzeba by uczynić jej pola publicznymi. Ale to ujawniłoby wszystkie stany edytora, także te prywatne. Inne klasy stałyby się zależne od najdrobniejszej zmiany klasy migawki, które w przeciwnym wypadku działałyby się w obrębie jej prywatnych pól i metod bez wpływu na zewnętrzne klasy.

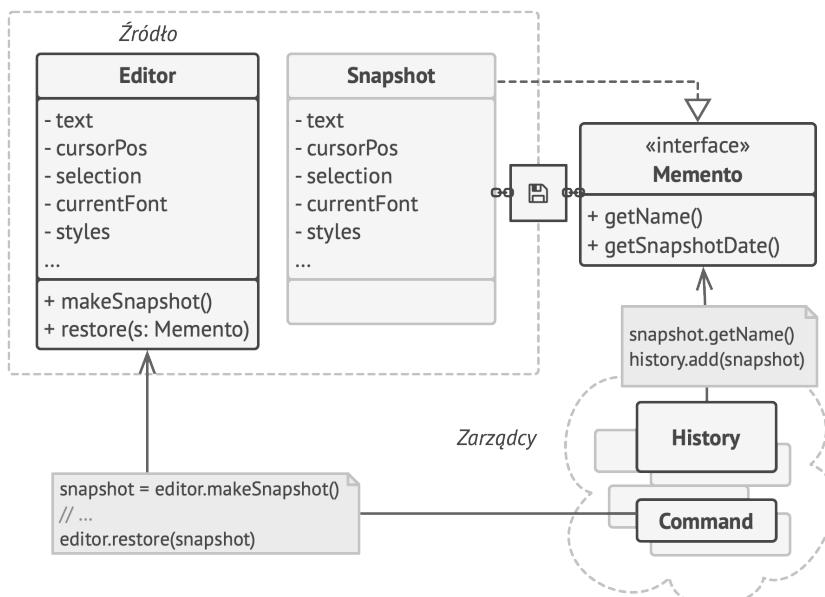
Wygląda na to, że trafiliśmy w ślepą uliczkę: można albo eksponować wszystkie wewnętrzne szczegóły klas, czyniąc je delikatnymi, albo ograniczyć dostęp do ich stanu, uniemożliwiając tym samym wykonywanie migawek. Czy jest jakiś inny sposób na implementację "cofnij"?

## Rozwiążanie

Wszystkie problemy na jakie się natknęliśmy są spowodowane niewłaściwą hermetyzacją. Niektóre obiekty próbują robić więcej niż powinny. Aby zbierać dane w celu wykonania jakiegoś zadania, wkradają się w prywatną przestrzeń innych obiektów, zamiast pozwolić im na samodzielne wykonanie tego zadania.

Wzorzec Pamiątka deleguje tworzenie migawki stanu samemu właścielowi stanu – obiektowi **źródło**. Dlatego też, zamiast pozwalać innym obiektom próbować skopiować stan edytora “z zewnątrz”, sama klasa edytora może wykonać migawkę siebie, gdyż ma pełny dostęp do swojego stanu.

Wzorzec proponuje przechowywanie kopii stanu w specjalnym obiekcie zwany *pamiątką*. Zawartość pamiętki nie jest dostępna innym obiektom, oprócz jej twórcy. Inne obiekty muszą komunikować się z pamięcią za pośrednictwem ograniczonego interfejsu, który pozwala na pobieranie metadanych migawki (czas utworzenia, nazwa wykonanej operacji, itd.), ale nie stanu pierwotnego obiektu zawartego w migawce.



*Źródło ma pełen dostęp do pamiętki, zaś zarządcy tylko do jej metadanych.*

Tak restrykcyjna polityka pozwala przechowywać pamiątki w obrębie innych obiektów, zwykle zwanych *zarządcami*. Skoro zarządca współpracuje z pamiątką tylko za pośrednictwem ograniczonego interfejsu, to nie jest w stanie naruszyć stanu w niej przechowywanego. Ponadto, źródło ma pełen dostęp do wszystkich pól pamiętki, co pozwala na przywracanie poprzednich stanów wedle potrzeb.

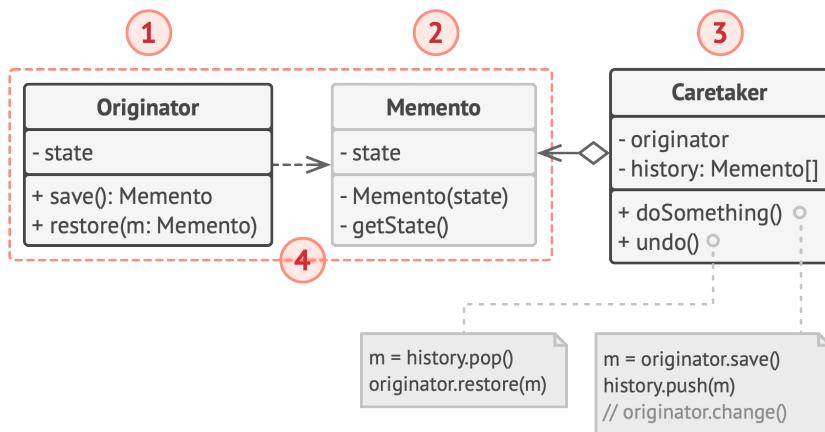
W naszym przykładzie edytora tekstowego możemy stworzyć osobną klasę historii która przyjmie rolę zarządcy. Stos pamiętek przechowany w zarządcy powiększy się przed każdą operacją edytora. Można nawet przedstawić reprezentację tego stosu w graficznym interfejsie użytkownika aplikacji, wyświetlając historię wcześniej wykonanych operacji.

Gdy użytkownik wywoła wycofanie operacji, historia pobierze najnowszą pamiątkę ze stosu i przekaże ją edytorowi z żądaniem cofnięcia. Edytor ma pełny dostęp do pamiętki, więc zmieni swój stan w oparciu o wartości w niej zawarte.

# Struktura

## Implementacja w oparciu o zagnieżdżone klasy

Klasyczna implementacja wzorca polega na zagnieżdżaniu klas, które jest możliwe w wielu popularnych językach programowania (jak C++, C# i Java).



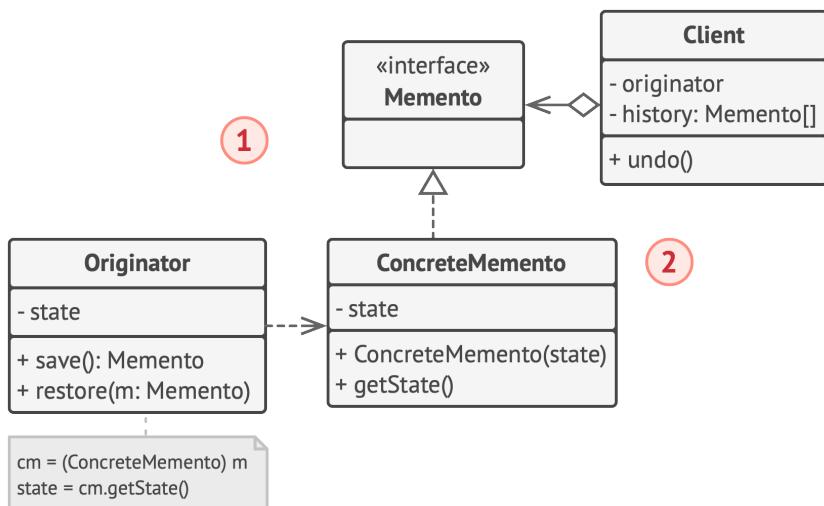
1. Klasa **Źródło** może tworzyć migawki swego stanu, a także przywracać wcześniejszy stan z migawek, gdy zachodzi taka potrzeba.
2. **Pamiątka** to obiekt wartości pełniący rolę pamiętki stanu źródła. Popularną praktyką jest czynienie pamiętki niezmienialną i ustawianie jej danych jednorazowo – poprzez konstruktor.
3. **Zarządca** wie nie tylko “kiedy” i “po co” rejestrować stan zarządcy, ale także kiedy należy przywrócić wcześniejszy stan.

Zarządcy może śledzić historię źródła przechowując stos pamiętek. Gdy źródło musi wrócić się w przeszłość, zarządcy pobiera najnowszą pamiętkę ze stosu i przekazuje ją metodzie przywracającej źródła.

- W tej implementacji klasa pamiętka jest zagnieźdzona wewnątrz klasy źródła. Pozwala to źródłu mieć dostęp do pól i metod pamiętki, mimo że są zadeklarowane jako prywatne. Z drugiej strony, zarządcy ma bardzo ograniczony dostęp do pól i metod pamiętek, co pozwala mu przechowywać je w formie stosu ale bez możliwości zmiany ich stanu.

### Implementacja na podstawie interfejsu pośredniego

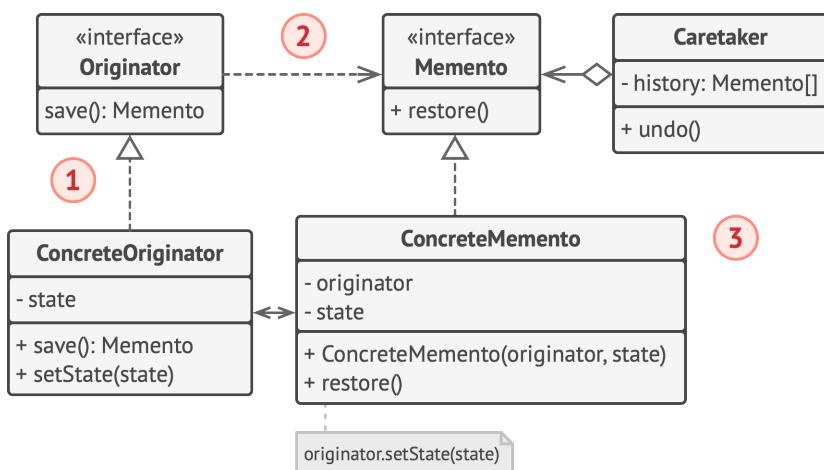
Istnieje alternatywna implementacja, odpowiednia w przypadku języków programowania które nie wspierają zagnieżdżania klas (mówiąc o sobie, PHP!).



- Wobec niemożności zagnieźdzania klas można ograniczyć dostęp do pól pamiętki stosując konwencję według której zarządcy współdziałają z pamiętkami wyłącznie za pośrednictwem jasno zadeklarowanego interfejsu pośredniego. Taki interfejs deklarowałby tylko metody związane z metadanymi pamiętki.
- Z drugiej strony, źródła mogą współpracować z pamiętkami bezpośrednio, poprzez dostęp do pól i metod w nich zadeklarowanych. Wadą tego podejścia jest konieczność deklaracji wszystkich składowych pamiętki jako publiczne.

### Implementacja z jeszcze ścisłejszą hermetyzacją

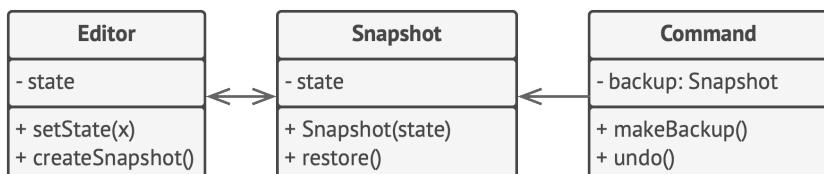
Kolejna implementacja jest użyteczna, gdy nie chcemy pozostawić nawet najmniejszego ryzyka, że obce klasy uzyskają dostęp do stanu źródła za pośrednictwem pamiętki.



1. Ta implementacja pozwala mieć wiele typów źródeł i pamiętek. Każde źródło współdziała z odpowiednią dla niego klasą pamiętki. Ani źródła, ani pamiętki nie eksponują swojego stanu komukolwiek.
2. Zarządcom uniemożliwiono teraz zmianę stanu przechowywanego w pamiętkach. Co więcej, klasa zarządcy staje się niezależna od źródła, ponieważ metoda przywracająca stan jest teraz zdefiniowana w klasie pamiętki.
3. Każda pamiętka staje się powiązana ze źródłem które ją utworzyło. Źródło przekazuje samo siebie do konstruktora pamiętki wraz z wartościami opisującymi jego stan. Dzięki bliskiemu związku pomiędzy tymi klasami, pamiętka może przywrócić stan źródła, gdyż ten drugi ma zdefiniowane stosowne metody setter.

## # Pseudokod

W poniższym przykładzie zastosowano wzorce Pamiątka oraz **Polecenie** w celu przechowywania migawek stanu rozbudowanego edytora tekstu i przywracania wcześniejszego stanu z owych migawek gdy zajdzie potrzeba.



*Zapisywanie migawek stanu edytora tekstu.*

Obiekty typu polecenie pełnią rolę zarządców. Pobierają pamiątkę edytora przed wykonaniem działań związanych z poleciением. Gdy użytkownik spróbuje cofnąć ostatnio wykonane polecenie, edytor może skorzystać z pamiątki przechowanej w tym poleceniu aby przywrócić wcześniejszy stan siebie.

Klasa pamiątka nie deklaruje żadnych pól publicznych, getterów, ani setterów. Dzięki temu żaden obiekt nie może zmienić zawartości pamiątki. Pamiątki są powiązane z tym obiektem edytora, który je utworzył. Pozwala to pamiętce przywrócić stan związanego z nią edytora przekazując przechowywane dane obiektowi edytora za pośrednictwem funkcji setter. Ponieważ pamiątki są połączone z konkretnymi obiektami edytorów, aplikacja mogłaby wspierać wiele niezależnych okien edycji ze scentralizowanym stosem historii.

```
1 // Źródło posiada jakieś istotne dane które mogą się z czasem
2 // zmieniać. Definiuje także metodę służącą zapisywaniu swojego
3 // stanu wewnątrz pamiątki i inną metodę do przywracania swojego
4 // stanu na podstawie pamiątki.
5 class Editor is
6     private field text, curX, curY, selectionWidth
7
8     method setText(text) is
9         this.text = text
10
11    method setCursor(x, y) is
12        this.curX = x
13        this.curY = y
14
```

```
15  method setSelectionWidth(width) is
16      this.selectionWidth = width
17
18 // Zapisuje bieżący stan w pamiętce.
19 method createSnapshot():Snapshot is
20     // Pamiątka to niezmienialny obiekt i dlatego źródło
21     // przekazuje swój stan pamiętce w formie parametru
22     // konstruktora.
23     return new Snapshot(this, text, curX, curY, selectionWidth)
24
25 // Pamiątka przechowuje przeszły stan edytora.
26 class Snapshot is
27     private field editor: Editor
28     private field text, curX, curY, selectionWidth
29
30 constructor Snapshot(editor, text, curX, curY, selectionWidth) is
31     this.editor = editor
32     this.text = text
33     this.curX = x
34     this.curY = y
35     this.selectionWidth = selectionWidth
36
37 // W którymś momencie będzie można przywrócić poprzedni stan
38 // edytora korzystając z obiektu pamiętki.
39 method restore() is
40     editor.setText(text)
41     editor.setCursor(curX, curY)
42     editor.setSelectionWidth(selectionWidth)
43
44 // Obiekt polecenie może pełnić rolę nadzorcy. W takim przypadku
45 // polecenie zapisuje w sobie pamiętkę zanim zmieni stan źródła.
46 // Gdy otrzyma rozkaz wycofania działania, przywraca stan źródła
```

```
47 // na podstawie zachowanej pamiątki.  
48 class Command is  
49     private field backup: Snapshot  
50  
51     method makeBackup() is  
52         backup = editor.createSnapshot()  
53  
54     method undo() is  
55         if (backup != null)  
56             backup.restore()  
57     // ...
```

## 💡 Zastosowanie

- ⚡ **Stosuj wzorzec Pamiątka gdy chcesz tworzyć migawki stanu obiektu i móc przywracać poprzedni jego stan.**
- ⚡ Wzorzec Pamiątka pozwala tworzyć pełne kopie stanu obiektu, wraz z jego danymi prywatnymi i przechowywać je poza obiektem. Chociaż większość kojarzy ten wzorzec z przypadkiem użycia “cofnij”, to jest on również nieodzowny w przypadku transakcji (np. gdy chcesz cofnąć wykonywane działanie w razie pojawienia się błędu).
- ⚡ **Stosuj ten wzorzec gdy bezpośredni dostęp do pól/getterów/setterów obiektu psuje hermetyzację.**

 Pamiątka czyni obiekt odpowiedzialnym za tworzenie migawek swojego stanu. Żaden inny obiekt nie ma prawa odczytać migawki, co zabezpiecza stan pierwotnego obiektu.

## Jak zaimplementować

1. Określ która klasa będzie pełniła rolę źródła. Ważne jest ustalenie czy program będzie miał jeden centralny obiekt tego typu, czy parę mniejszych.
2. Stwórz klasę pamiętki. Jeden po drugim zadeklaruj zestaw pól odzwierciedlających pola zadeklarowane w klasie źródła.
3. Uczyn klasę pamiętki niezmienialną. Pamiątka powinna przyjmować dane tylko raz – za pośrednictwem konstruktora. Klasa nie powinna mieć metod setter.
4. Jeśli stosowany język programowania wspiera zagnieżdżane klasy, zagnieźdź pamiętkę w obrębie klasy źródła. Jeśli nie wspiera, wyekstrahuj pusty interfejs z klasy pamiętka i spraw, by inne obiekty korzystały z niego w kontakcie z pamięcią. Można dodać do takiego interfejsu funkcje związane z metadanymi, ale żadnych funkcji eksponujących stan źródła.
5. Dodaj metodę tworzącą pamiętki do klasy źródła. Źródło powinno przekazywać swój stan do pamiętki za pośrednictwem jednego lub wielu argumentów konstruktora pamiętki.

Typem zwracanym przez metodę powinien być interfejs wyekstrahowany w poprzednim etapie (zakładając, że w ogóle się go wyekstrahowało). Za kulisami, metoda tworząca pamiątkę powinna współdziałać bezpośrednio z klasą pamiątki.

6. Dodaj do klasy źródła metodę służącą przywracaniu stanu. Powinna przyjmować w charakterze argumentu obiekt typu pamiątka. Jeśli wyekstrahowano interfejs w poprzednim etapie, uczyń go typem parametru. W tym przypadku, trzeba rzutować obiekt przyjmowany na klasę pamiątki, ponieważ źródło musi mieć pełen dostęp do tego obiektu.
7. Zarządcy, niezależnie od tego czy reprezentuje obiekt poleceń, historię lub coś jeszcze innego, powinien wiedzieć kiedy żądać nowych pamiątek od źródła, jak je przechowywać i kiedy przywracać stan źródła za pomocą jakiejś pamiątki.
8. Połączenie między zarządcami a źródłami można przenieść do klasy pamiątka. W tym przypadku, każda pamiątka musi być połączona ze źródłem które je utworzyło. Metoda przywracająca również trafiłaby do klasy pamiątka. To wszystko jednak ma sens tylko jeśli klasa pamiątka jest zagnieżdżona wewnątrz klasy źródła lub jeśli klasa źródła udostępnia funkcje setter służące nadpisywaniu jej stanu.

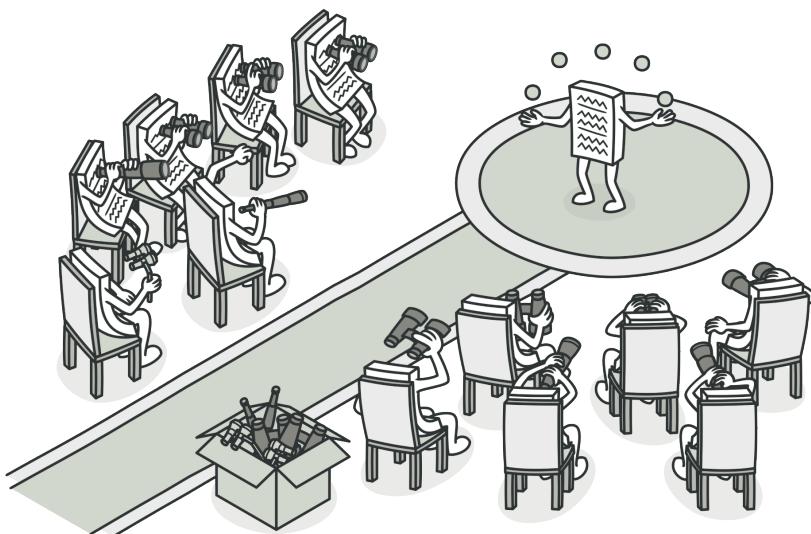
## Zalety i wady

- ✓ Można tworzyć migawki stanu obiektów bez naruszania ich hermetyzacji.

- ✓ Można uprościć kod źródła, pozwalając zarządcy śledzić historię stanu źródła.
- ✗ Aplikacja może wymagać dużej ilości pamięci RAM jeśli klienci zbyt często będą tworzyć pamiątki.
- ✗ Zarządcy powinni śledzić cykl życia źródła, aby być w stanie kasować zbędne pamiątki.
- ✗ Większość dynamicznych języków programowania, jak PHP, Python i JavaScript nie daje gwarancji niezmienialności stanu pamiątki.

## ↔ Powiązania z innymi wzorcami

- Można stosować **Polecenie** i **Pamiątkę** jednocześnie – implementując funkcjonalność “cofnij”. W takim przypadku, polecenia są odpowiedzialne za wykonywanie różnych działań na obiekcie docelowym, zaś pamiątki służą zapamiętaniu stanu obiektu tuż przed wykonaniem polecenia.
- Można zastosować **Pamiątkę** wraz z **Iteratorem** by zapisać bieżący stan iteracji, co pozwoli w razie potrzeby do niego powrócić.
- Czasem **Prototyp** może być prostszą alternatywą dla **Pamiątki**. Jest to możliwe jeśli obiekt, którego stan chcesz przechować w historii, jest w miarę nieskomplikowany. Obiekt taki nie powinien też mieć powiązań z zewnętrznymi zasobami, albo muszą one być łatwe do ponownego nawiązania.



# OBSERWATOR

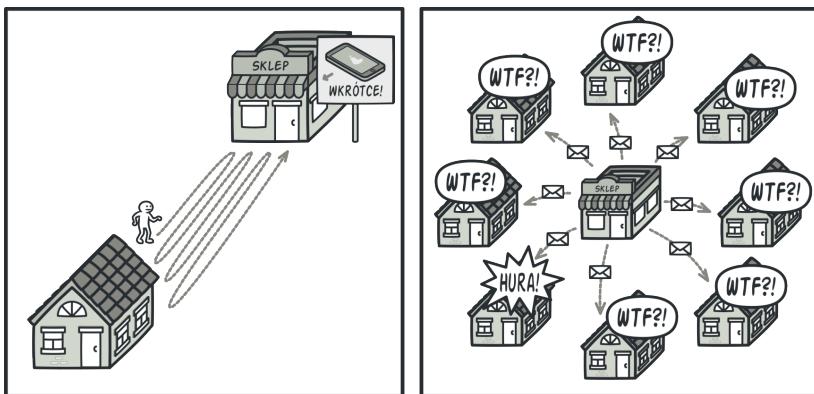
Znany też jako: *Event-Subscriber, Listener, Observer*

**Obserwator** to czynnościowy (behawioralny) wzorzec projektowy pozwalający zdefiniować mechanizm subskrypcji w celu powiadamiania wielu obiektów o zdarzeniach dzierżących się w obserwowanym obiekcie.

## Problem

Wyobraź sobie, że masz dwa typy obiektów: `Klient` oraz `Sklep`. Klient jest zainteresowany jakąś szczególną marką produktu, na przykład nowym modelem iPhone, który ma się niedługo pojawić w sklepie.

Klient mógłby odwiedzać sklep codziennie i sprawdzać dostępność produktu, ale dopóki produkt jest w drodze, większość tych spacerów będzie bezcelowa.



*Odwiedzanie sklepu a rozsyłanie spamu.*

Z drugiej strony, sklep mógłby rozesłać mnóstwo emaili do wszystkich klientów jak tylko pojawi się nowy produkt, ale to może zostać odebrane jako spamowanie. Zaoszczędziłoby wprawdzie niektórym klientom zbędnych podróży do sklepu, ale jednocześnie niektórzy by się zdenerwowali otrzymując nieistotną dla nich wiadomość.

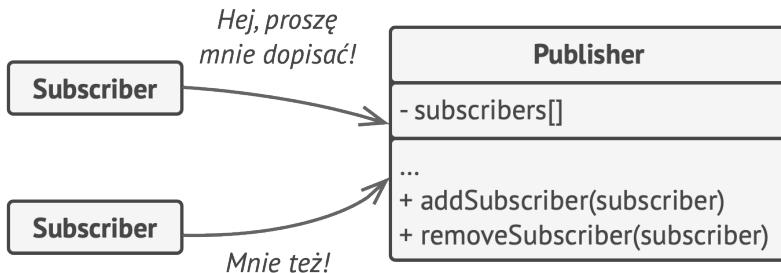
Wygląda na to, że mamy tu konflikt. Albo klient traci czas sprawdzając dostępność produktu, albo sklep ponosi koszty powiadamiając niewłaściwych klientów.

## Rozwiążanie

Obiekt który posiada jakiś interesujący stan nazywa się zwykle *podmiotem*, ale skoro będzie powiadaniał inne obiekty o zmianach swojego stanu, można nazwać go *publikującym*. Wszystkie pozostałe obiekty, które chcą śledzić zmiany stanu nadawcy nazywa się *subskrybentami*.

Wzorzec Obserwator proponuje dodanie mechanizmu subskrypcji do klasy publikującego, aby pojedyncze obiekty mogły subskrybować lub przerwać subskrypcję strumienia zdarzeń publikującego. Na szczęście nie jest to tak skomplikowane jak brzmi. Tak naprawdę mechanizm ten składa się z 1) pola tablicowego służącego przechowywaniu listy odniesień do subskrybentów oraz 2) wielu metod publicznych pozwalających dodawać i usuwać wpisy tej listy.

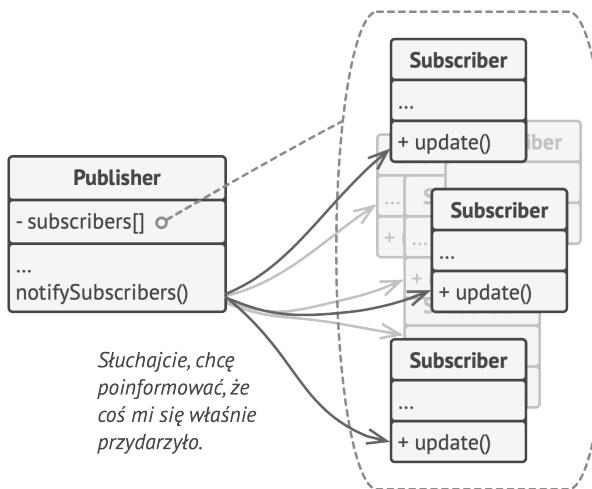
Za każdym razem, gdy wydarzy się coś ważnego publikującemu, może on przejrzeć swoją listę subskrybentów i wywołać odpowiednią metodę powiadamiania ich obiektów.



*Mechanizm subskrypcji pozwala pojedynczym obiektom subskrybować powiadomienia o zdarzeniach.*

Prawdziwe aplikacje mogą mieć tuziny różnych klas subskrybentów które są zainteresowane śledzeniem zdarzeń jednej klasy publikującego. Nie chcielibyśmy spręgać nadawcy z tymi wszystkimi klasami. Poza tym możesz nawet z góry nic o nich nie wiedzieć, jeśli klasę publikującą zamierzasz udostępnić innym ludziom.

Dlatego właśnie ważnym jest, aby wszyscy subskrybenci implementowali ten sam interfejs i żeby publikujący komunikował się z nimi wyłącznie poprzez ten interfejs. Powinien on deklarować metodę powiadamiania wraz z zestawem parametrów za pomocą których publikujący może przekazać dodatkowe dane kontekstowe wraz z powiadomieniem.



*Publikujący powiadamia subskrybentów wywołując ich odpowiednie metody powiadamiania.*

Jeśli Twoja aplikacja ma wiele różnych typów nadawców i chcesz uczynić swoich subskrybentów kompatybilnymi z każdym z typów, możesz pójść o krok dalej i zmusić publikujących do korzystania z tego samego interfejsu. Taki interfejs musiałby opisywać tylko kilka metod subskrybowania. Interfejs umożliwiłby subskrybentom obserwację stanów obiektu publikującego bez konieczności sprzągania z ich konkretnymi klasami.

## 🚘 Analogia do prawdziwego życia

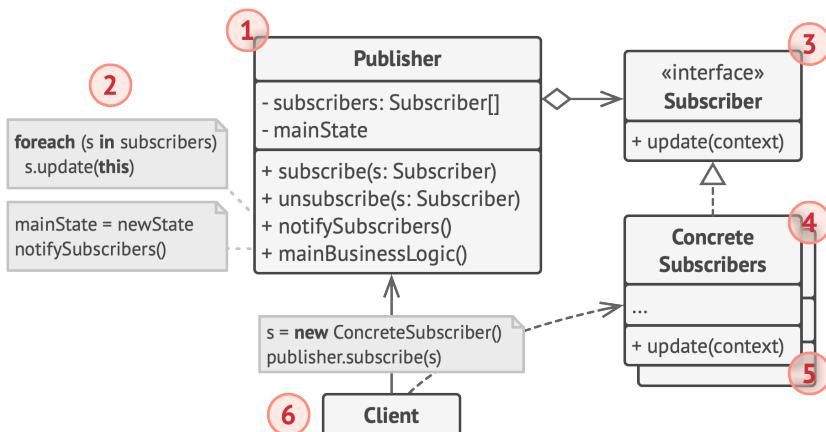
Jeśli subskrybujesz czasopismo lub gazetę, nie musisz więcej chodzić do sklepu by sprawdzić czy jest już nowy numer. Zamiast tego, wydawca przesyła ci nowe egzemplarze pocztą od razu po opublikowaniu, a czasem nawet trochę wcześniej.



*Subskrypcje czasopism i gazet.*

Wydawca zarządza listą subskrybentów i wie, które czasopisma kogo interesują. Subskrybenci mogą wypisać się z listy, kiedy nie chcą już otrzymywać kolejnych edycji.

## Struktura



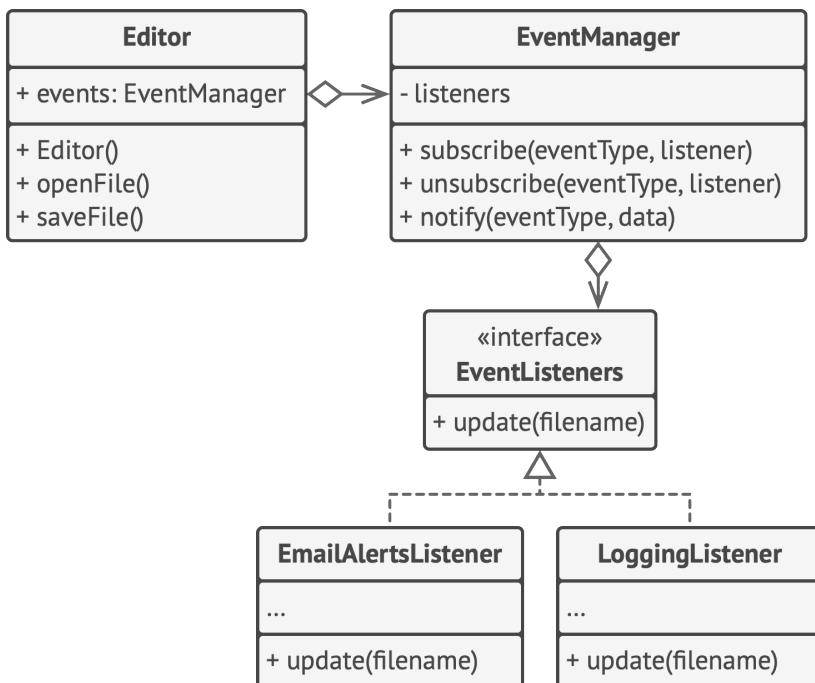
1. **Publikujący** rozsyła zdarzenia interesujące inne obiekty. Zdarzenia te zachodzą, gdy publikujący zmienia swój stan lub wy-

kona jakieś obowiązki. Publikujący posiadają infrastrukturę dającą możliwość subskrybowania ich zdarzeń lub przerwania subskrypcji.

2. Gdy nastąpi nowe zdarzenie, nadawca przegląda listę subskrybentów i wywołuje metody powiadamiania zadeklarowane w ich interfejsach.
3. Interfejs **Subskrybenta** deklaruje interfejs powiadamiania. W większości przypadków składa się on z jednej metody aktualizuj. Metoda ta może przyjmować wiele parametrów pozwalających publikującemu przekazać za ich pomocą szczegóły dotyczące aktualizacji.
4. **Konkretni Subskrybenci** wykonują jakieś czynności w odpowiedzi na powiadomienia wysłane przez publikującego. Wszystkie te klasy muszą implementować ten sam interfejs, aby nadawca nie musiał być sprzęgnięty z ich konkretnymi klasami.
5. Zazwyczaj, subskrybenci potrzebują jakichś kontekstowych informacji aby poprawnie obsłużyć aktualizacje. Dlatego publikujący na ogół przekazują dane kontekstowe jako argumenty metody powiadamiania. Publikujący może przekazać samego siebie jako argument, umożliwiając subskrybentom pobranie sobie potrzebnych danych bezpośrednio.
6. **Klient** tworzy obiekty publikujące i subskrybujące osobno, po czym rejestruje subskrybentów by mogli otrzymywać aktualizacje od publikującego.

## # Pseudokod

W poniższym przykładzie, wzorzec **Obserwator** pozwala obiektovi będącemu edytorem tekstu powiadamać inne obiekty usługowe o zmianach swojego stanu.



*Powiadamianie obiektów o zdarzeniach dotyczących innych obiektów.*

Lista subskrybentów jest kompilowana dynamicznie: obiekty mogą zacząć lub przestać oczekiwania na powiadomienia w trakcie działania programu, zależnie od potrzebnego zachowania twojej aplikacji.

W tej implementacji, klasa edytora sama nie zarządza listą subskrybentów. Deleguje to zadanie specjalnemu obiektowi obsługującemu właśnie tę czynność. Można ulepszyć ten obiekt, aby służył jako scentralizowany dyspozytor, pozwalając każdemu obiekowi pełnić rolę publikującego.

Dodawanie nowych subskrybentów do programu nie wymaga zmian w istniejących klasach publikujących, o ile będą one współpracować ze wszystkimi subskrybentami za pośrednictwem tego samego interfejsu.

```
1 // Bazowa klasa publikującego zawiera kod zarządzania
2 // subskrypcją i metody powiadamiania.
3 class EventManager is
4     // Tablica asocjacyjna typów zdarzeń i słuchaczy.
5     private field listeners: hash map
6
7     method subscribe(eventType, listener) is
8         listeners.add(eventType, listener)
9
10    method unsubscribe(eventType, listener) is
11        listeners.remove(eventType, listener)
12
13    method notify(eventType, data) is
14        foreach (listener in listeners.of(eventType)) do
15            listener.update(data)
16
17    // Konkretny publikujący zawiera prawdziwą logikę biznesową
18    // która jest istotna z punktu widzenia niektórych
19    // subskrybentów. Moglibyśmy pozyskać tę klasę z klasy bazowej
```

```
20 // publikującego, ale nie jest to zawsze możliwe w prawdziwym
21 // świecie, ponieważ konkretny publikujący może już być
22 // podklassą. W takim przypadku można wkomponować logikę
23 // subskrypcji w sposób pokazany poniżej.
24 class Editor is
25     public field events: EventManager
26     private field file: File
27
28     constructor Editor() is
29         events = new EventManager()
30
31     // Metody logiki biznesowej mogą powiadamiać subskrybentów o
32     // zmianach.
33     method openFile(path) is
34         this.file = new File(path)
35         events.notify("open", file.name)
36
37     method saveFile() is
38         file.write()
39         events.notify("save", file.name)
40
41     // ...
42
43
44 // Oto interfejs subskrybenta. Jeśli język programowania którego
45 // używasz obsługuje typy funkcyjne, możesz wymienić całą
46 // hierarchię subskrypcji zestawem funkcji.
47 interface EventListener is
48     method update(filename)
49
50 // Konkretni subskrybenci reagują na aktualizacje emitowane
51 // przez obiekt publikujący do którego są przywiązane.
```

```
52 class LoggingListener implements EventListener is
53     private field log: File
54     private field message: string
55
56     constructor LoggingListener(log_filename, message) is
57         this.log = new File(log_filename)
58         this.message = message
59
60     method update(filename) is
61         log.write(replace('%s',filename,message))
62
63 class EmailAlertsListener implements EventListener is
64     private field email: string
65     private field message: string
66
67     constructor EmailAlertsListener(email, message) is
68         this.email = email
69         this.message = message
70
71     method update(filename) is
72         system.email(email, replace('%s',filename,message))
73
74
75 // Aplikacja może konfigurować publikujących i subskrybentów w
76 // trakcie działania programu.
77 class Application is
78     method config() is
79         editor = new Editor()
80
81         logger = new LoggingListener(
82             "/path/to/log.txt",
83             "Someone has opened the file: %s")
```

```
84     editor.events.subscribe("open", logger)
85
86     emailAlerts = new EmailAlertsListener(
87         "admin@example.com",
88         "Someone has changed the file: %s")
89     editor.events.subscribe("save", emailAlerts)
```

## 💡 Zastosowanie

💡 **Stosuj wzorzec Obserwator gdy zmiany stanu jednego obiektu mogą wymagać zmiany w innych obiektach, a konkretny zestaw obiektów nie jest zawsze znany lub ulega zmianom dynamicznie.**

⚡ Można często natknąć się na ten problem pracując z klasami graficznego interfejsu użytkownika. Przykładowo, stworzyliśmy własne klasy przycisków i chcemy aby klienci mogli podpiąć jakiś własny kod do twoich przycisków, aby uruchamiał się po ich naciśnięciu.

Wzorzec Obserwator pozwala każdemu obiekowi implementującemu interfejs subskrypcji otrzymywać powiadomienia o zdarzeniach w obiektach publikujących. Można dodać mechanizm subskrypcji do swoich przycisków, pozwalając klientom na podłączenie do przycisków ich kodu za pośrednictwem własnych klas subskrybentów.

-  **Stosuj ten wzorzec gdy jakieś obiekty w twojej aplikacji muszą obserwować inne, ale tylko przez jakiś czas lub w niektórych przypadkach.**
-  Lista subskrybentów jest dynamiczna, więc subskrybenci mogą zapisać się lub wypisać z listy kiedy chcą.

## Jak zaimplementować

1. Przejrzyj logikę biznesową swojego programu i spróbuj podzielić ją na dwie części. Główną funkcjonalność, która jest niezależna od innego kodu, uczynimy obiektem publikującym. Reszta natomiast zostanie przekształcona na zestaw klas subskrybujących.
2. Zadeklaruj interfejs subskrybenta. W najprostszej postaci powinien deklarować pojedynczą metodę `aktualizuj`.
3. Zadeklaruj interfejs publikujący i opisz metody służące do dodawania i usuwania subskrybentów z listy. Pamiętaj, że obiekty publikujące muszą współdziałać z subskrybentami wyłącznie poprzez interfejs subskrybenta.
4. Zdecyduj gdzie umieścić faktyczną listę subskrybentów oraz implementację metod zarządzających nią. Zazwyczaj taki kod wygląda jednakowo dla wszystkich typów obiektów publikujących, więc najbardziej oczywistym miejscem wydaje się klasa abstrakcyjna wywodząca się bezpośrednio z interfejsu publiku-

jącego. Konkretni publikujący rozszerzają tę klasę, dziedzicząc funkcjonalność subskrypcji.

Jednak jeśli stosujesz ten wzorzec w kontekście istniejącej hierarchii klas, rozważ podejście bazujące na kompozycji: umieść logikę subskrypcji w osobnym obiekcie i pozwól by wszyscy publikujący z niej korzystali.

5. Stwórz konkretne klasy publikujące. Za każdym razem gdy zdarzy się coś ważnego w obiekcie publikującym, musi on powiadomić o tym swoich subskrybentów.
6. Zaimplementuj metody powiadamiania o aktualizacji w konkretnych klasach subskrybentów. Większość subskrybentów będzie potrzebować jakichś danych kontekstowych o zdarzeniu. Można je przekazać w formie argumentu metody powiadamiania.

Istnieje jednak jeszcze jedna opcja. Otrzymawszy powiadomienie, subskrybent może pobrać dane bezpośrednio od powiadomienia. W takim przypadku, obiekt publikujący musi przekazać samego siebie za pośrednictwem metody aktualizacji. Mniej elastyczną opcją jest powiązanie obiektu publikującego z subskrybentem na stałe za pośrednictwem konstruktora.

7. Klient musi stworzyć wszystkich niezbędnych subskrybentów i zarejestrować ich u odpowiednich publikujących.

## Zalety i wady

- ✓ *Zasada otwarte/zamknięte.* Można wprowadzać do programu nowe klasy subskrybujące bez konieczności zmieniania kodu publikującego (i odwrotnie, jeśli istnieje interfejs publikujący).
- ✓ Można utworzyć związek pomiędzy obiektami w trakcie działania programu.
- ✗ Subskrybenci powiadamiani są w przypadkowej kolejności.

## Powiązania z innymi wzorcami

- Wzorce Łańcuch zobowiązań, Polecenie, Mediator i Obserwator dotyczą różnych sposobów na łączenie nadawców z odbiorcami żądań:
  - *Łańcuch zobowiązań* przekazuje żądanie sekwencyjnie wzdłuż dynamicznego łańcucha potencjalnych odbiorców, aż któryś z nich je obsługuje.
  - *Polecenie* pozwala nawiązywać jednokierunkowe połączenia pomiędzy nadawcami i odbiorcami.
  - *Mediator* eliminuje bezpośrednie połączenia pomiędzy nadawcami a odbiorcami, zmuszając ich do komunikacji za pośrednictwem obiektu mediatora.
  - *Obserwator* pozwala odbiorcom dynamicznie zasubskrybować się i zrezygnować z subskrypcji żądań.

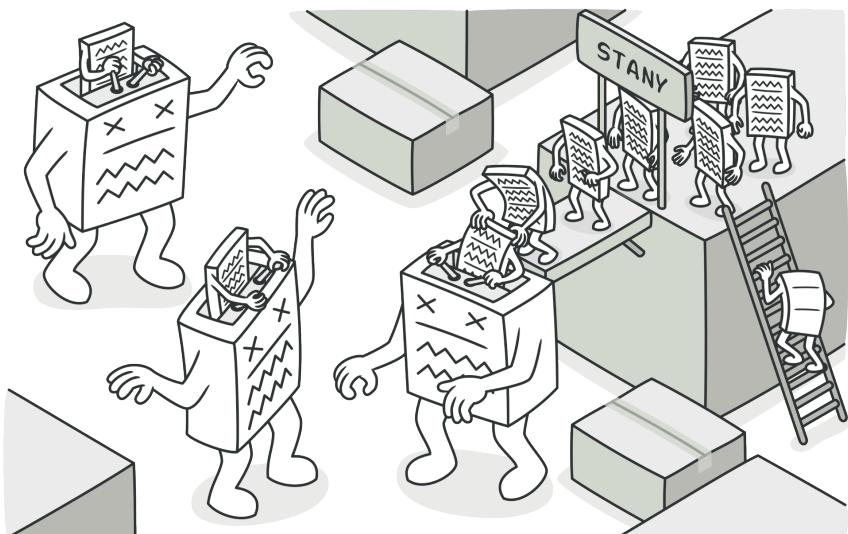
- Różnica pomiędzy **Mediatorem** a **Obserwatorem** jest często trudna do uchwycenia. W większości przypadków można implementować je zamiennie, a czasem jednocześnie. Zobaczmy, jak by to wyglądało.

Głównym celem *Mediatora* jest eliminacja wzajemnych zależności pomiędzy zestawem komponentów systemu. Zamiast tego uzależnia się te komponenty od jednego obiektu-mediatora. Celem *Obserwatora* jest ustanowienie dynamicznych, jednokierunkowych połączeń między obiekty, których część jest podległa innym.

Istnieje popularna implementacja wzorca Mediator która bazuje na *Obserwatorze*. Obiekt mediatora pełni w niej rolę publikującego, zaś komponenty są subskrybentami mogącymi "prenumerować" zdarzenia które nadaje mediator. Gdy *Mediator* jest zaimplementowany w ten sposób, może przypominać *Obserwatora*.

Jeśli nie jest to zrozumiałe, warto przypomnieć sobie, że można zaimplementować wzorzec Mediator na inne sposoby. Na przykład można na stałe powiązać wszystkie komponenty z tym samym obiektem mediator. Taka implementacja nie będzie przypominać *Obserwatora*, ale nadal będzie instancją wzorca Mediator.

A teraz wyobraźmy sobie program w którym wszystkie komponenty stały się publikującymi, pozwalając na dynamiczne połączenia pomiędzy sobą. Nie będzie wówczas skoncentrowanego obiektu mediatora, a tylko rozproszony zestaw obserwatorów.



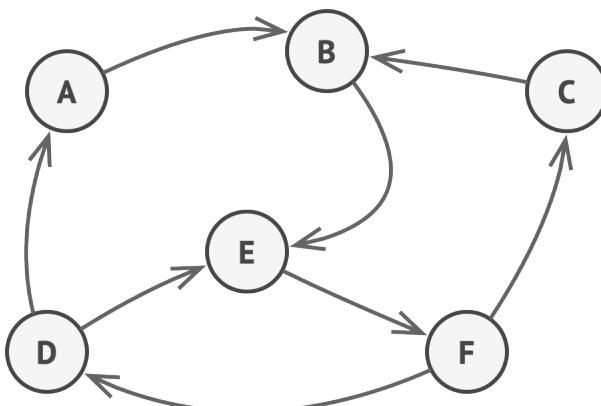
# STAN

Znany też jako: *State*

**Stan** to behawioralny wzorzec projektowy pozwalający obiektowi zmienić swoje zachowanie gdy zmieni się jego stan wewnętrzny. Wygląda to tak, jakby obiekt zmienił swoją klasę.

## :( Problem

Wzorzec Stan jest powiązany z koncepcją *Automatu skończonego*<sup>1</sup>.



Automat skończony.

Sednem tej koncepcji jest to, że w dowolnym momencie istnieje skończona liczba stanów w których program może się znajdować. W każdym z nich program zachowuje się różnie i może zostać przełączony z jednego stanu w drugi natychmiastowo. Możliwe stany, w jakich obiekt może się znaleźć, zależą od bieżącego stanu. Liczba reguł przełączeń, zwanych *przejściami* również jest skończona i są one z góry określone.

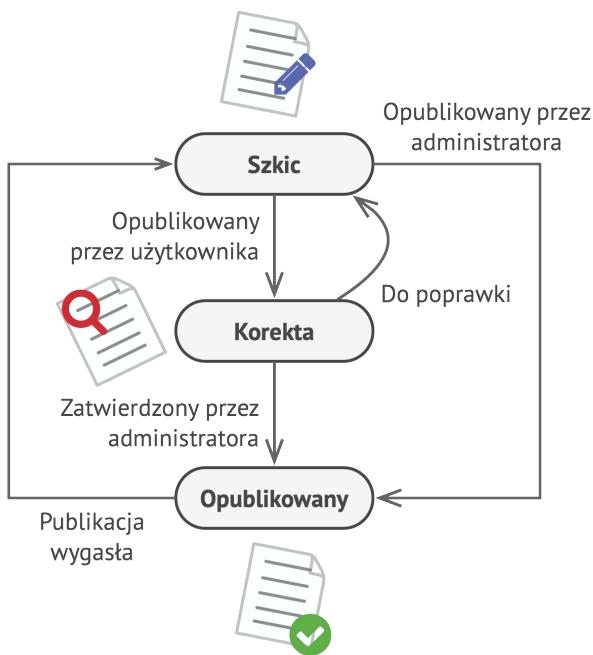
Można też zastosować to podejście wobec obiektów. Wyobraźmy sobie klasę Dokument. Dokument może znajdować się w jednym z trzech stanów: Szkic, Korekta i Opublikowany. Me-

---

1. Automat skończony: <https://refactoring.guru/pl/fsm>

toda `publikuj` dokumentu działa nieco inaczej zależnie od jego stanu:

- W przypadku `Szkicu` przenosi dokument do moderacji.
- W stanie `Korekta` czyni dokument dostępnym publicznie, ale tylko jeśli bieżący użytkownik jest administratorem.
- W stanie `Opublikowany` nie robi nic.



*Możliwe stany i przejścia między stanami obiektu dokument.*

Maszyny stanów są zwykle implementowane za pomocą wielu struktur warunkowych (`if` lub `switch`) które wybierają odpowiednie zachowanie zależnie od bieżącego stanu dokumentu. Zazwyczaj “stan” jest tylko zestawem wartości pól obiektu.

Nawet jeśli nie wiesz nic o automatach skończonych, prawdopodobnie przynajmniej raz implementowałeś stan. Czy poniższy kawałek kodu coś ci przypomina?

```
1 class Document is
2   field state: string
3   // ...
4   method publish() is
5     switch (state)
6       "draft":
7         state = "moderation"
8         break
9       "moderation":
10      if (currentUser.role == "admin")
11        state = "published"
12        break
13      "published":
14      // Nie rób nic.
15      break
16    // ...
```

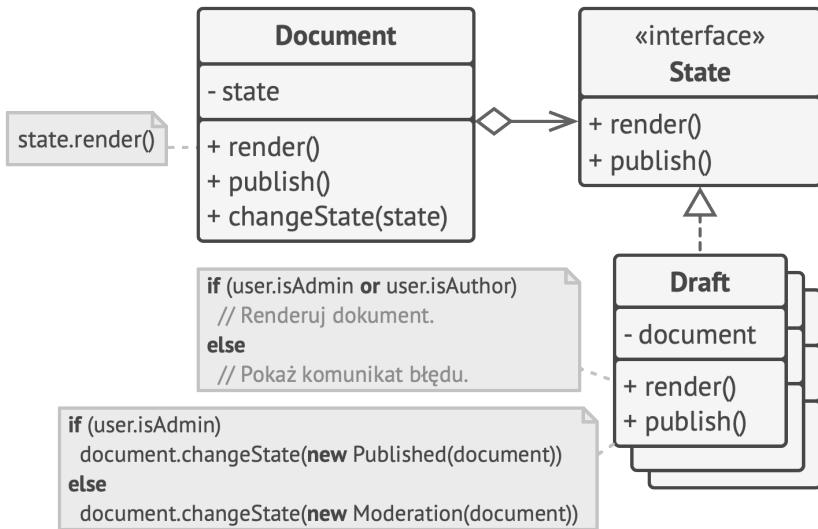
Największa słabość maszyny stanów opartej na instrukcjach warunkowych ujawnia się w miarę dodawania kolejnych stanów oraz zachowań zależnych od tych stanów do klasy `Dokument`. Większość metod będzie zawierała olbrzymie instrukcje warunkowe wybierające stosowne zachowanie się metody zależnie od bieżącego stanu. Taki kod jest trudny w utrzymaniu, ponieważ każda zmiana logiki przechodzenia między stanami może wymagać zmian instrukcji warunkowych w każdej z metod.

Problem narasta wraz z rozbudową projektu. Trudno jest przewidzieć wszystkie możliwe stany i przejścia między nimi na etapie projektowania. Dlatego też prosta maszyna stanów, zbudowana z ograniczonej liczby instrukcji warunkowych, może z czasem spuchnąć do niebotycznych rozmiarów.

## Rozwiążanie

Wzorzec Stan proponuje stworzenie nowych klas dla każdego z możliwych stanów obiektu oraz ekstrakcję wszystkich zachowań zależnych od stanu do tychże klas.

Zamiast implementować wszystkie zachowania samodzielnie, pierwotny obiekt, zwany *kontekstem*, przechowuje odniesienie do jednego z obiektów-stanów który w danej chwili reprezentuje jego bieżący stan i deleguje mu zadania związane z tym stanem.



*Dokument deleguje pracę obiektowi stanu.*

Aby przełączyć kontekst do innego stanu, zamieniamy aktywny obiekt stanu na inny obiekt reprezentujący nowy stan. Jest to możliwe wyłącznie gdy wszystkie klasy stanu są zgodne pod względem interfejsu, zaś kontekst współpracuje z tymi obiektami tylko poprzez ów interfejs.

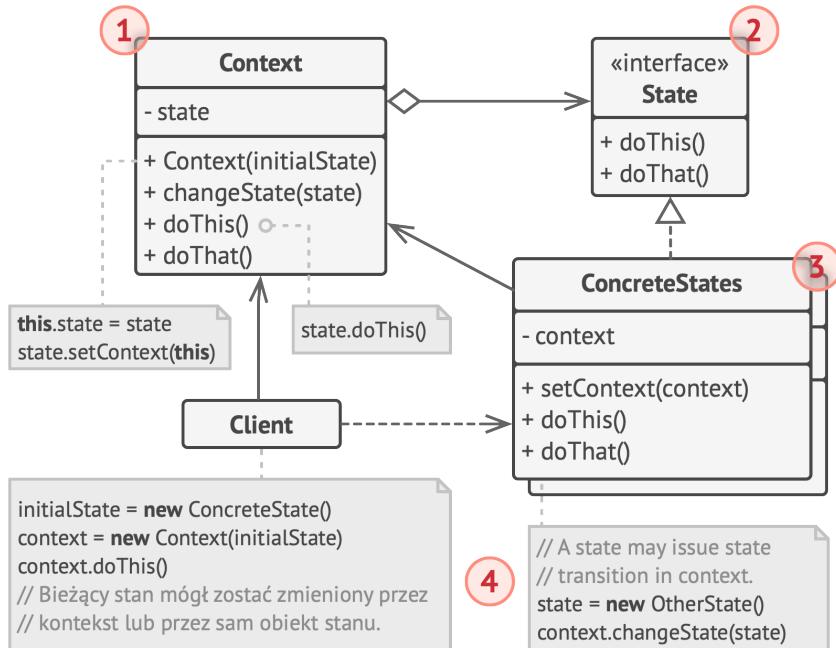
Taka struktura może przypominać wzorzec **Strategia**, ale z jedną kluczową różnicą. W przypadku wzorca Stan, poszczególne stany mogą być świadome siebie nawzajem i inicjować przejścia z jednego stanu w drugi, zaś strategie prawie nigdy nie wiedzą nic o sobie.

## 🚗 Analogia do prawdziwego życia

Przyciski i przełączniki w twoim smartfonie zachowują się w różny sposób, w zależności od bieżącego stanu urządzenia:

- Gdy telefon jest odblokowany, wciskanie przycisków wywołuje różne funkcje.
- Gdy telefon jest zablokowany, wciskanie przycisków wyświetli ekran służący odblokowaniu.
- Gdy bateria telefonu jest na wyczerpaniu, wciśnięcie dowolnego przycisku wyświetli ekran ładowania.

## 📱 Struktura



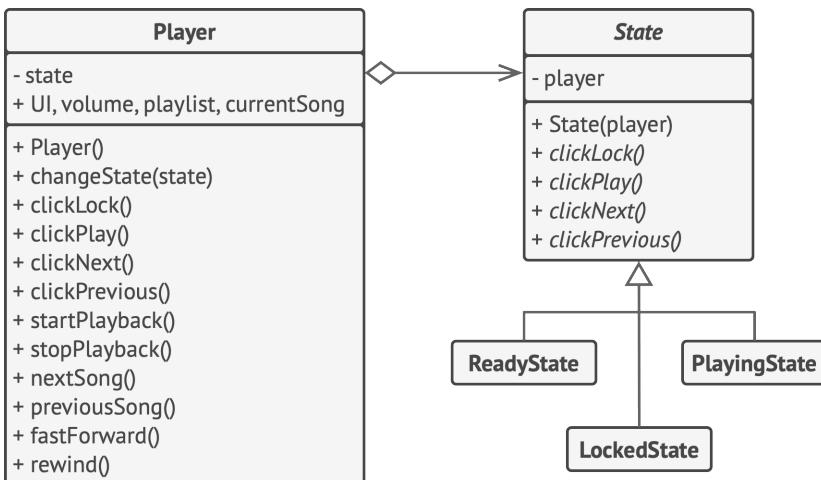
1. **Kontekst** przechowuje odniesienie do jednego z konkretnych obiektów-stanów i deleguje mu zadania specyficzne dla danego stanu. Kontekst porozumiewa się z obiektem stanu za pośrednictwem interfejsu stanu. Kontekst eksponuje metodę setter, przez którą przekazuje się obiekt nowego stanu dla kontekstu.
2. Interfejs **Stanu** deklaruje metody specyficzne dla stanu. Metody te powinny być sensowne dla konkretnych stanów, ponieważ nie chcemy, aby któreś ze stanów posiadały bezużyteczne metody które nie zostaną nigdy wywołane.
3. **Konkretne Stany** dostarczają swoje implementacje metod specyficznych dla poszczególnych stanów. Aby uniknąć powtórzeń podobnego kodu w wielu stanach, można utworzyć pośrednie klasy abstrakcyjne które hermetyzują jakieś wspólne zachowania.

Obiekty stanu mogą przechowywać referencje wsteczne do obiektu kontekst. Za pośrednictwem tego odniesienia stan może pobrać dowolne informacje z obiektu kontekst, a także zainicjować zmianę stanu.

4. Zarówno kontekst, jak i konkretne stany mogą ustawić kolejny stan kontekstu i wykonać samą zmianę stanu poprzez zamianę obiektu stanu powiązanego z kontekstem na inny.

## # Pseudokod

W tym przykładzie, wzorzec **Stan** pozwala tym samym kontrolkom odtwarzacza multimedialnego zachowywać się nieco inaczej, zależnie od stanu odtwarzania.



*Przykład zmiany zachowania się obiektu zależnie od obiektu stanu.*

Główny obiekt odtwarzacza jest zawsze powiązany z obiektem stanu, który wykonuje większość zadań odtwarzacza. Niektóre czynności zamieniają bieżący obiekt stanu na inny, co przy okazji zmienia reakcje odtwarzacza na polecenia od użytkownika.

```

1 // Klasa AudioPlayer pełni rolę kontekstu. Posiada także
2 // odniesienie do instancji jednej z klas-stanów reprezentującej
3 // bieżący stan odtwarzacza audio.
4 class AudioPlayer is
  
```

```
5   field state: State
6   field UI, volume, playlist, currentSong
7
8   constructor AudioPlayer() is
9     this.state = new ReadyState(this)
10
11    // Kontekst deleguje obsługę danych wejściowych
12    // użytkownika obiektowi stanu. Oczywiście wynik zależy
13    // od tego jaki stan jest aktualnie aktywny, ponieważ
14    // każdy ze stanów może obsługiwać dane wejściowe nieco
15    // inaczej.
16    UI = new UserInterface()
17    UI.lockButton.onClick(this.clickLock)
18    UI.playButton.onClick(this.clickPlay)
19    UI.nextButton.onClick(this.clickNext)
20    UI.prevButton.onClick(this.clickPrevious)
21
22    // Inne obiekty muszą być w stanie przełączyć aktywny stan
23    // odtwarzacza audio.
24  method changeState(state: State) is
25    this.state = state
26
27    // Metody interfejsu użytkownika delegują wykonanie
28    // aktywnemu stanowi.
29  method clickLock() is
30    state.clickLock()
31  method clickPlay() is
32    state.clickPlay()
33  method clickNext() is
34    state.clickNext()
35  method clickPrevious() is
36    state.clickPrevious()
```

```
37
38 // Stan może wywoływać jakieś metody-usługi kontekstu.
39 method startPlayback() is
40     // ...
41 method stopPlayback() is
42     // ...
43 method nextSong() is
44     // ...
45 method previousSong() is
46     // ...
47 method fastForward(time) is
48     // ...
49 method rewind(time) is
50     // ...
51
52
53 // Klasa bazowa stanu deklaruje metody które muszą być
54 // zaimplementowane przez wszystkie konkretne stany. Posiada też
55 // referencję zwrotną do obiektu-kontekstu skojarzonego ze
56 // stanem. Stany mogą za pomocą tej referencji zwrotnej wywołać
57 // przejście kontekstu z jednego stanu w inny.
58 abstract class State is
59     protected field player: AudioPlayer
60
61 // Kontekst przekazuje siebie do konstruktora stanu. Pomoże
62 // to stanowi pozyskiwać różne użyteczne dane kontekstu
63 // jeśli zaistnieje potrzeba.
64 constructor State(player) is
65     this.player = player
66
67 abstract method clickLock()
68 abstract method clickPlay()
```

```
69  abstract method clickNext()
70  abstract method clickPrevious()
71
72
73 // Konkretne stany implementują różnorakie zachowania związane z
74 // danym stanem kontekstu.
75 class LockedState extends State is
76
77 // Gdy odblokuje się zablokowany odtwarzacz, może on znaleźć
78 // się w którymś z dwóch stanów.
79 method clickLock() is
80     if (player.playing)
81         player.changeState(new PlayingState(player))
82     else
83         player.changeState(new ReadyState(player))
84
85 method clickPlay() is
86     // Zablokowany, więc nie rób nic.
87
88 method clickNext() is
89     // Zablokowany, więc nie rób nic.
90
91 method clickPrevious() is
92     // Zablokowany, więc nie rób nic.
93
94
95 // Konkretne stany mogą też wyzwolić przejście kontekstu z
96 // jednego stanu w inny.
97 class ReadyState extends State is
98     method clickLock() is
99         player.changeState(new LockedState(player))
100
```

```
101 method clickPlay() is
102     player.startPlayback()
103     player.changeState(new PlayingState(player))
104
105 method clickNext() is
106     player.nextSong()
107
108 method clickPrevious() is
109     player.previousSong()
110
111
112 class PlayingState extends State is
113     method clickLock() is
114         player.changeState(new LockedState(player))
115
116     method clickPlay() is
117         player.stopPlayback()
118         player.changeState(new ReadyState(player))
119
120     method clickNext() is
121         if (event.doubleclick)
122             player.nextSong()
123         else
124             player.fastForward(5)
125
126     method clickPrevious() is
127         if (event.doubleclick)
128             player.previous()
129         else
130             player.rewind(5)
```

## Zastosowanie

-  **Stosuj wzorzec Stan gdy masz do czynienia z obiektem którego zachowanie jest zależne od jego stanu, liczba możliwych stanów jest wielka, a kod specyficzny dla danego stanu często ulega zmianom.**
-  Wzorzec proponuje ekstrakcję całego kodu właściwego poszczególnym stanom do zestawu osobnych klas. W wyniku tego można będzie dodawać nowe stany lub zmieniać istniejące niezależnie od siebie, zmniejszając koszty utrzymania.
-  **Stosuj ten wzorzec gdy masz klasę zaśmieconą rozbudowanymi instrukcjami warunkowymi zmieniającymi zachowanie klasy zależnie od wartości jej pól.**
-  Wzorzec Stan pozwala wyekstrahować rozgałęzienia tych instrukcji warunkowych do metod które znajdą się w klasach reprezentujących poszczególne stany. W ten sposób uprzątać można przy okazji tymczasowe pola i metody pomocnicze związane z kodem odnoszącym się do stanów.
-  **Wzorzec Stan pomaga poradzić sobie z dużą ilością kodu który się powtarza w wielu stanach i przejściach między stanami automatu skońzonego, bazującego na instrukcjach warunkowych.**

⚡ Wzorzec Stan pozwala komponować hierarchie klas stanów i zmniejszyć ilość powtórzeń kodu poprzez ekstrakcję wspólnego kodu do abstrakcyjnych klas bazowych.

## Jak zaimplementować

1. Zdecyduj która klasa będzie pełniła rolę kontekstu. Może to być istniejąca klasa zawierająca już jakiś kod zależny od stanu obiektu, ale może to być także nowa klasa, jeśli kod specyficzny dla stanów jest rozrzucony po wielu klasach.
2. Zadeklaruj interfejs stanu. Mimo że może on odzwierciedlać wszystkie metody zadeklarowane w kontekście, skup się tylko na tych, które dotyczą zachowania specyficznego dla danego stanu.
3. Dla każdego faktycznego stanu stwórz klasę wywodzącą się z interfejsu stanu. Następnie przejrzyj metody kontekstu i wyekstrahuj cały kod dotyczący tego stanu do nowo utworzonej klasy.

Przenosząc kod do klasy stanu możesz zauważyć, że zależy on od prywatnych składowych klasy kontekstu. Można sobie z tym poradzić w następujący sposób:

- Uczyń te pola lub metody publicznymi.
- Zmień ekstrahowane zachowanie na publiczną metodę kontekstu i wywołuj ją z klasy stanu. To brzydkie, ale szybkie rozwiązanie, które można później naprawić.

- Zagnieźdź klasy stanów w klasie kontekstu, ale tylko jeśli używany język programowania obsługuje zagnieżdżanie klas.
4. W klasie kontekstu dodaj pole przechowujące odniesienie do interfejsu stanu i publicznie dostępną metodę setter, która umożliwia nadpisanie wartości tego pola.
  5. Przejrzyj metodę kontekstu raz jeszcze i zamień puste instrukcje warunkowe dotyczące stanu na wywołania stosownych metod obiektu stanu.
  6. Aby przełączyć stan kontekstu, utwórz instancję jednej z klas stanu i przekaż ją kontekstowi. Można tego dokonać w ramach samego kontekstu, w którymś ze stanów, bądź po stronie klienta. Za każdym razem, gdy to się dzieje, klasa staje się zależna od konkretnej klasy stanu której instancja powstaje.

## ⚠ Zalety i wady

- ✓ *Zasada pojedynczej odpowiedzialności.* Zorganizuj kod związany z konkretnymi stanami w osobne klasy.
- ✓ *Zasada otwarte/zamknięte.* Można wprowadzać nowe stany bez zmiany istniejących klas stanu lub kontekstu.
- ✓ Upraszczka kod kontekstu eliminując obszerne instrukcje warunkowe automatu skońzonego.
- ✗ Zastosowanie tego wzorca może być przesadą jeśli mamy do czynienia zaledwie z kilkoma stanami i rzadkimi zmianami.

## ↔ Powiązania z innymi wzorcami

- **Most, Stan, Strategia** (i w pewnym stopniu **Adapter**) mają podobną strukturę. Wszystkie oparte są na kompozycji, co oznacza delegowanie zadań innym obiektom. Jednak każdy z tych wzorców rozwiązuje inne problemy. Wzorzec nie jest bowiem tylko receptą na ustrukturyzowanie kodu w pewien sposób, lecz także informacją dla innych deweloperów o charakterze rozwiązywanego problemu.
- **Stan** można uważać za rozszerzenie **Strategii**. Oba wzorce oparte są o kompozycję: zmieniają zachowanie kontekstu przez delegowanie części zadań obiektom pomocniczym. *Strategia* czyni te obiekty całkowicie niezależnymi i nieświadomymi siebie nawzajem. Jednakże *Stan* nie ogranicza zależności pomiędzy konkretnymi stanami i pozwala im zmieniać stan kontekstu według uznania.



# STRATEGIA

*Znany też jako: Strategy*

**Strategia** to behawioralny wzorzec projektowy pozwalający zdefiniować rodzinę algorytmów, umieścić je w osobnych klasach i uczynić obiekty tych klas wymienialnymi.

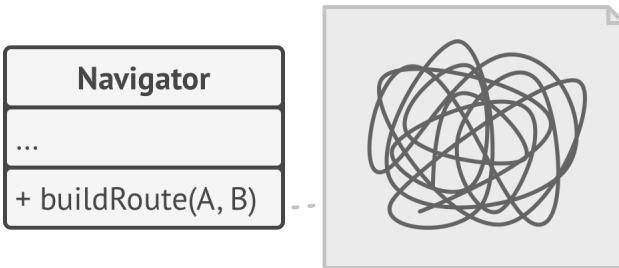
## Problem

Pewnego dnia postanawiasz stworzyć aplikację służącą nawigacji dla turystów. Aplikacja jest zbudowana w oparciu o piękną mapę ułatwiającą użytkownikom szybko zorientować się w topografii miasta.

Jedną z funkcji, o którą użytkownicy prosili najczęściej, było automatyczne planowanie trasy przez aplikację. Po wprowadzeniu adresu, na mapie pokazałaby się najszybsza trasa do tego miejsca.

Pierwsza wersja aplikacji potrafiła planować trasy po drogach. Osoby podróżujące autami były więc wniebowzięte. Okazało się jednak, że nie wszyscy lubią jeździć na urlop autem. Dlatego kolejna aktualizacja wprowadziła możliwość generowania szlaków pieszych. Niedługo po tym aplikacja zyskała opcję wyznaczania tras w oparciu o komunikację miejską.

To był jednak dopiero początek. W planach pojawiło się bowiem dodanie obsługi tras dogodnych dla rowerzystów. A jeszcze później zaplanowano tworzenie tras uwzględniających wszystkie atrakcje turystyczne miasta.



*Kod nawigacji stał się zagmatwany.*

Chociaż z perspektywy biznesowej aplikacja odniosła sukces, aspekty techniczne stały się dla ciebie zmorą. Po każdym dodaniu kolejnego algorytmu wytyczania trasy, główna klasa nawigatora dwukrotnie się powiększała. W pewnym momencie okiełznanie tej bestii stało się zbyt trudne.

Każda zmiana któregoś z algorytmów – usunięcie usterki, czy też dostrajanie punktacji kolejnych odcinków trasy wpływalo na całą klasę, zwiększając tym samym ryzyko błędu w już działającym kodzie.

Na dodatek ucierpiała praca zespołowa. Twoi współpracownicy, zatrudnieni po udanym wprowadzeniu programu na rynek, zaczęli narzekać, że marnują zbyt wiele czasu na rozwiązywanie konfliktów scalania. Implementacja każdej nowej funkcji wymaga zmiany tej samej rozbudowanej klasy, nad którą jednocześnie pracują inni.

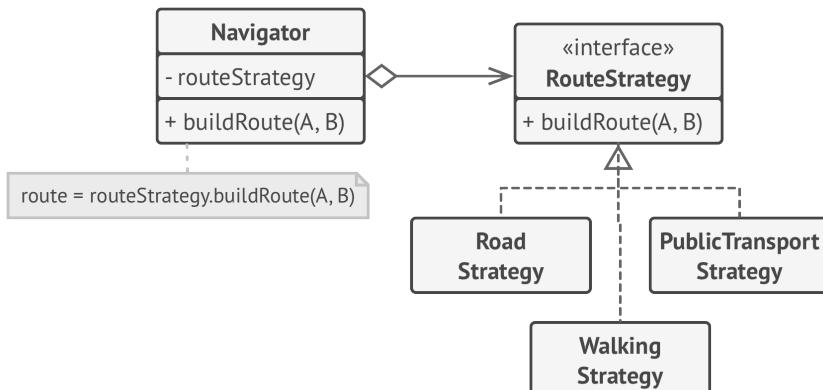
## Rozwiążanie

Wzorzec Strategia proponuje ekstrakcję poszczególnych algorytmów wykonujących dane zadanie na różne sposoby i umieszczenie ich w odrębnych klasach, zwanych *strategiami*.

Pierwotna klasa, zwana *kontekstem*, musi zawierać pole służące przechowywaniu odniesienia do którejś ze strategii. Kontekst deleguje pracę powiązanemu obiektyowi typu strategia, zamiast wykonywać ją samodzielnie.

Kontekst nie jest odpowiedzialny za wybór stosownego algorytmu dla danego zadania. To klient przekazuje żądaną strategię kontekstowi. Co więcej, kontekst nie wie zbyt wiele o strategiach. Współpracuje ze wszystkimi strategiami za pośrednictwem tego samego, ogólnego interfejsu, który eksponuje pojedynczą metodę uruchamiającą algorytm ukryty w danej strategii.

Tym sposobem kontekst staje się niezależny od konkretnych strategii, więc można dodawać kolejne algorytmy, lub modyfikować istniejące, bez zmianiania kodu kontekstu lub kodu innych strategii.

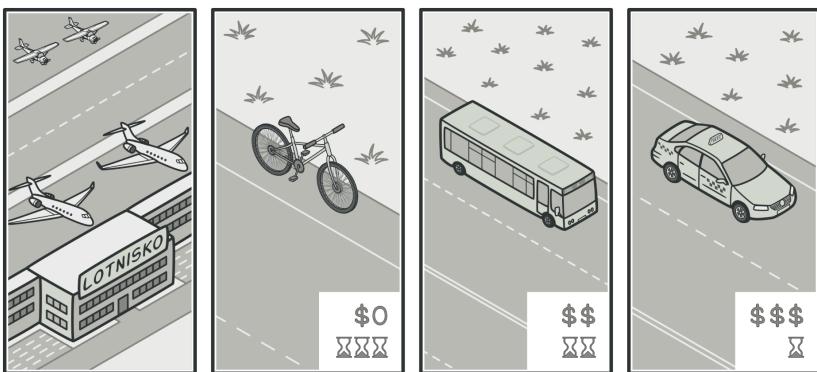


*Strategie planowania trasy.*

W naszej aplikacji nawigacyjnej, każdy algorytm wyznaczania tras może zostać wyekstrahowany do swojej własnej, odrębnej klasy, posiadającej jedną metodę `stwórzTrasę`. Metoda przyjmuje informacje o punkcie początkowym i docelowym, a zwraca zestaw kluczowych etapów wyznaczonej trasy.

Każda klasa wyznaczająca trasę może wytyczyć inną trasę w oparciu o te same argumenty, a główna klasa nawigator nie musi wiedzieć o obranym algorytmie, gdyż zajmuje się jedynie przedstawianiem trasy na mapie. Klasa posiada metodę umożliwiającą zmianę aktywnej strategii planowania trasy, dzięki czemu jej klienci, jak na przykład przyciski interfejsu użytkownika, mogą zmienić bieżący sposób wyznaczania trasy na inny.

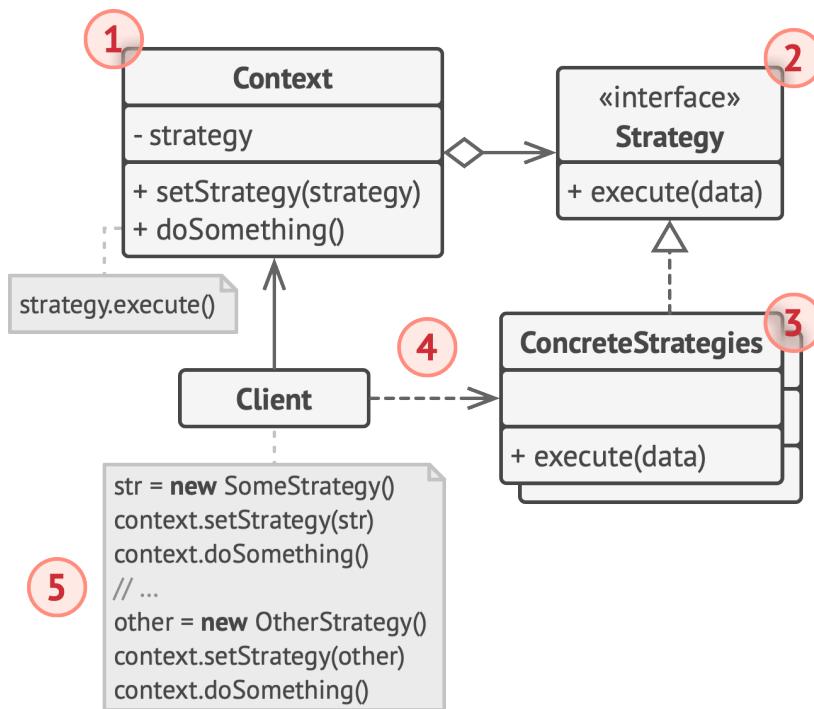
## 🚗 Analogia do prawdziwego życia



Różne strategie dotarcia na lotnisko.

Wyobraź sobie, że musisz dotrzeć na lotnisko. Możesz złapać autobus, zamówić taksówkę lub pojechać rowerem. To są twoje strategie przejazdu. Możesz wybrać jedną z nich zależnie od czynników takich jak budżet lub ograniczenia czasowe.

## Struktura



1. **Kontekst** przechowuje odniesienie do jednej z konkretnych strategii i komunikuje się z jej obiektem za pośrednictwem interfejsu strategia.
2. Interfejs **Strategia** jest wspólny dla wszystkich konkretnych strategii. Deklaruje metodę za pomocą której kontekst uruchamia daną strategię.
3. **Konkretne Strategie** implementują różne warianty algorytmu z którego korzysta kontekst.

4. Kontekst wywołuje metodę uruchamiającą eksponowaną przez powiązany z nim obiekt strategii za każdym razem gdy chce uruchomić algorytm. Kontekst nie wie z jaką strategią ma do czynienia, lub jak działa algorytm.
5. **Klient** tworzy określony obiekt strategii i przekazuje go kontekstowi. Kontekst eksponuje metodę setter pozwalającą klientom zamienić strategię skojarzoną z tym kontekstem w trakcie działania programu.

## # Pseudokod

W poniższym przykładzie, kontekst wykorzystuje kilka **strategii** by wykonać różne działania arytmetyczne.

```
1 // Interfejs strategii deklaruje działania wspólne dla
2 // wszystkich wspieranych wersji jakiegoś algorytmu. Kontekst
3 // korzysta z tego interfejsu w celu wywoływania algorytmów
4 // zdefiniowanych przez konkretne strategie.
5 interface Strategy is
6     method execute(a, b)
7
8     // Konkretne strategie implementują algorytm według poniższego
9     // interfejsu bazowej strategii. Interfejs sprawia, że są one
10    // wymienialne w kontekście.
11 class ConcreteStrategyAdd implements Strategy is
12     method execute(a, b) is
13         return a + b
14
15 class ConcreteStrategySubtract implements Strategy is
```

```
16  method execute(a, b) is
17      return a - b
18
19 class ConcreteStrategyMultiply implements Strategy is
20     method execute(a, b) is
21         return a * b
22
23 // Kontekst definiuje interfejs stanowiący przedmiot
24 // zainteresowania klientów.
25 class Context is
26     // Kontekst posiada odniesienie do jednego z obiektów-
27     // strategii. Kontekst nie zna konkretnej klasy strategii.
28     // Powinien współpracować ze wszystkimi strategiami za
29     // pośrednictwem wspólnego interfejsu strategii.
30     private strategy: Strategy
31
32     // Zazwyczaj kontekst przyjmuje strategię poprzez
33     // konstruktor. Udostępnia także funkcję setter
34     // umożliwiającą wymianę strategii na inną w trakcie
35     // działania programu.
36     method setStrategy(Strategy strategy) is
37         this.strategy = strategy
38
39     // Kontekst deleguje jakąś pracę obiektowi-strategii zamiast
40     // samodzielnie implementować wiele wersji algorytmu.
41     method executeStrategy(int a, int b) is
42         return strategy.execute(a, b)
43
44
45 // Kod klienta wybiera jakąś konkretną strategię i przekazuje ją
46 // kontekstowi. Klient powinien być świadom różnic pomiędzy
47 // poszczególnymi strategiami aby móc podjąć właściwy wybór.
```

```
48 class ExampleApplication is
49     method main() is
50         Create context object.
51
52         Read first number.
53         Read last number.
54         Read the desired action from user input.
55
56         if (action == addition) then
57             context.setStrategy(new ConcreteStrategyAdd())
58
59         if (action == subtraction) then
60             context.setStrategy(new ConcreteStrategySubtract())
61
62         if (action == multiplication) then
63             context.setStrategy(new ConcreteStrategyMultiply())
64
65         result = context.executeStrategy(First number, Second number)
66
67         Print result.
```

## 💡 Zastosowanie

- 💡 Stosuj wzorzec Strategia gdy chcesz używać różnych wariantów jednego algorytmu w obrębie obiektu i zyskać możliwość zmiany wyboru wariantu w trakcie działania programu.
- ⚡ Wzorzec Strategia pozwala pośrednio zmienić zachowanie obiektu w trakcie działania programu poprzez przypisywanie

temu obiektowi różnych podobiektów wykonujących określone podziałania na różne sposoby.

- ⚡ **Warto stosować ten wzorzec gdy masz w programie wiele podobnych klas, różniących się jedynie sposobem wykonywania jakichś zadań.**
- ⚡ Wzorzec Strategia umożliwia ekstrakcję różniących się zachowań do odrębnej hierarchii klas i połączenie pierwotnych klas w jedną, redukując tym samym powtórzenia kodu.
- ⚡ Strategia pozwala odizolować logikę biznesową klasy od szczegółów implementacyjnych algorytmów, które nie są istotne w kontekście tej logiki.
- ⚡ Strategia umożliwia odizolowanie kodu różnych algorytmów, ich danych wewnętrznych oraz zależności od reszty kodu. Klienci otrzymują prosty interfejs umożliwiający uruchamianie algorytmów i wymiany ich na inne w trakcie działania programu.
- ⚡ **Stosuj ten wzorzec gdy Twoja klasa zawiera duży operator warunkowy, którego zadaniem jest wybór odpowiedniego wariantu tego samego algorytmu.**
- ⚡ Wzorzec Strategia pozwala pozbyć się wyżej wymienionych kawałków kodu poprzez ekstrakcję algorytmów do odrębnych klas implementujących taki sam interfejs. Pierwotny obiekt de-

leguje uruchamianie jednemu z tych obiektów, zamiast samodzielnie implementować wszystkie warianty algorytmu.

## Jak zaimplementować

1. W klasie kontekstu zidentyfikuj algorytm który często może ulegać zmianom. Może to być też obszerna instrukcja warunkowa wybierająca i uruchamiająca wariant tego samego algorytmu w trakcie działania programu.
2. Zadeklaruj interfejs strategii który będzie wspólny dla wszystkich wariantów algorytmu.
3. Jeden po drugim ekstrahuj wszystkie algorytmy do odrębnych klas implementujących interfejs strategii.
4. W klasie kontekstu, dodaj pole służące przechowywaniu odniesienia do obiektu strategii. Przygotuj metodę setter służącą zmianie wartości tego pola. Kontekst powinien współpracować z obiektem strategii wyłącznie przez interfejs strategii. Kontekst może definiować interfejs dający strategii dostęp do swoich danych.
5. Klienci kontekstu muszą skojarzyć go ze stosowną strategią, która odpowiada sposobowi w jaki kontekst ma wykonać swoje zadanie.

## ⚠ Zalety i wady

- ✓ Możesz zamieniać algorytmy stosowane w obrębie obiektu w trakcie działania programu.
- ✓ Możesz odizolować szczegóły implementacyjne algorytmu od kodu który z niego korzysta.
- ✓ Umożliwia zamianę dziedziczenia na kompozycję.
- ✓ *Zasada otwarte/zamknięte.* Możliwe jest wprowadzanie nowych strategii bez konieczności dokonywania zmian w kontekście.
- ✗ Jeśli masz zaledwie kilka algorytmów i rzadko ulegają one zmianie, nie ma wyraźnej potrzeby nadmiernego komplikowania programu przez dodawanie nowych klas i interfejsów związanych z tym wzorcem.
- ✗ Klienci muszą być świadomi różnic pomiędzy poszczególnymi strategiami, aby mogli wybrać właściwą.
- ✗ Wiele nowoczesnych języków programowania posiada wsparcie dla typów funkcyjnych pozwalających zaimplementować różne wersje algorytmu wewnątrz zestawu anonimowych funkcji. Można następnie korzystać z tych funkcji dokładnie tak jak z obiektów strategia, ale bez konieczności rozbudowy kodu o kolejne klasy i interfejsy.

## ↔ Powiązania z innymi wzorcami

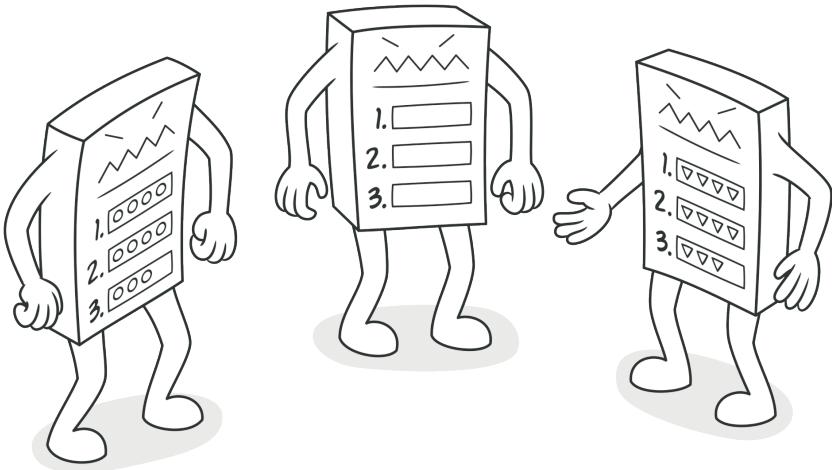
- **Most, Stan, Strategia** (i w pewnym stopniu **Adapter**) mają podobną strukturę. Wszystkie oparte są na kompozycji, co oznacza delegowanie zadań innym obiektom. Jednak każdy z tych

wzorców rozwiązuje inne problemy. Wzorzec nie jest bowiem tylko receptą na ustrukturyzowanie kodu w pewien sposób, lecz także informacją dla innych deweloperów o charakterze rozwiązywanego problemu.

- **Polecenie i Strategia** mogą wydawać się podobne, ponieważ oba mogą służyć parametryzacji obiektu jakimś działaniem. Mają jednak inne cele.
  - Za pomocą *Polecenia* można konwertować dowolne działanie na obiekt. Parametry działania stają się polami tego obiektu. Konwersja zaś pozwala odroczyć wykonanie działania, kolejkować je i przechowywać historię wykonanych działań, a także wysyłać polecenia zdalnym usługom, itd.
  - Z drugiej strony, *Strategia* zazwyczaj opisuje różne sposoby wykonywania danej czynności, pozwalając zamieniać algorytmy w ramach jednej klasy kontekstu.
- **Dekorator** pozwala zmienić otoczkę obiektu, zaś **Strategia** jej wnętrze.
- **Metoda szablonowa** polega na mechanizmie dziedziczenia: pozwala zmieniać części algorytmu rozszerzając je w podklasach. **Strategia** bazuje na kompozycji: można zmienić część zachowania obiektu poprzez nadanie mu różnych strategii odpowiadających temu zachowaniu. *Metoda szablonowa* działa na poziomie klasy, więc jest statyczna. *Strategia* działa na poziomie obiektu.

mie obiektu, więc pozwala przełączać zachowania w trakcie działania programu.

- Stan można uważać za rozszerzenie Strategii. Oba wzorce oparte są o kompozycję: zmieniają zachowanie kontekstu przez delegowanie części zadań obiektom pomocniczym. *Strategia* czyni te obiekty całkowicie niezależnymi i nieświadomymi siebie nawzajem. Jednakże *Stan* nie ogranicza zależności pomiędzy konkretnymi stanami i pozwala im zmieniać stan kontekstu według uznania.



# METODA SZABLONOWA

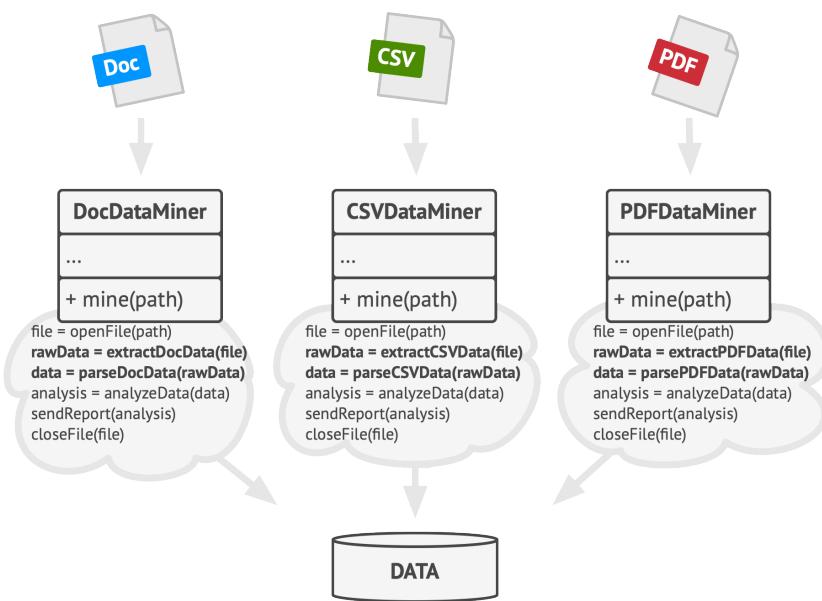
Znany też jako: *Template Method*

**Metoda szablonowa** to behawioralny wzorzec projektowy definiujący szkielet algorytmu w klasie bazowej, ale pozwalający podklasom nadpisać pewne etapy tego algorytmu bez konieczności zmiany jego struktury.

## :( Problem

Wyobraź sobie, że tworzysz aplikację zbierającą dane, która analizuje dokumenty w korporacji. Użytkownicy przesyłają do niej dokumenty w różnych formatach (PDF, DOC, CSV), a ta próbuje wydobyć z nich istotne dane i przedstawić je w jednym formacie.

Pierwsza wersja aplikacji mogła współpracować tylko z plikami DOC, kolejna wersja dodała obsługę plików CSV. Miesiąc później aplikacja “umiała” już ekstrahować dane z PDF.



*Klasy eksploracji danych zawierały sporo powtarzającego się kodu.*

W jakimś momencie zauważasz, że wszystkie trzy klasy mają sporo podobnego kodu. Fragmenty odpowiedzialne za pracę z

różnymi formatami danych są bardzo odmienne, ale kod odpowiedzialny za obróbkę i analizę danych jest niemal identyczny. Śivetnie byłoby się tych powtórzeń pozbyć nie naruszając przy tym struktury algorytmów, prawda?

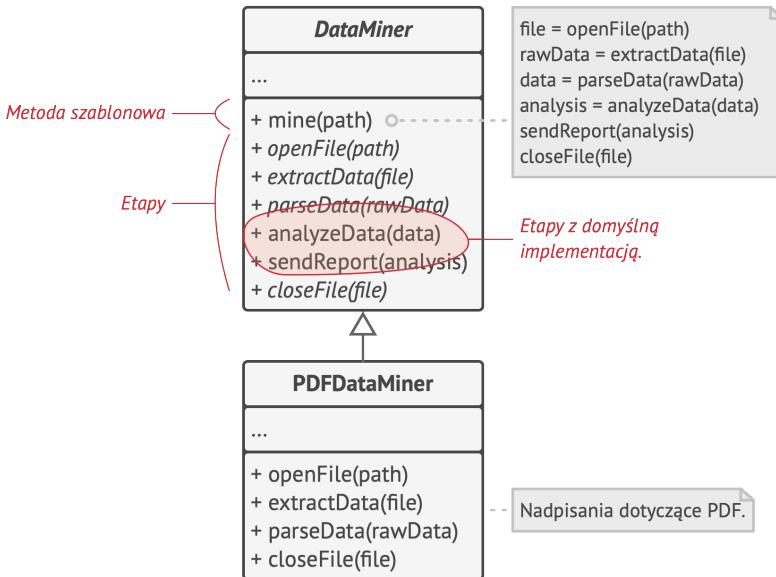
Istniał jeszcze jeden problem, dotyczący kodu klienckiego który korzystał z tych klas. Miał pełno instrukcji warunkowych służących wybieraniu odpowiedniego sposobu działania zależnie od klasy obiektu przetwarzającego. Jeśli wszystkie trzy klasy przetwarzające miałyby jeden wspólny interfejs lub wspólną klasę bazową, byłoby możliwe usunięcie instrukcji warunkowych w kodzie klienckim i zastosowanie polimorfizmu podczas wywoływania metod obiektu przetwarzającego.

## Rozwiążanie

Rozwiążanie zawarte we wzorcu Metody szablonowej zakłada rozdzielenie algorytmu na kolejne etapy, utworzenie z tych etapów metod i umieszczenie ciągu wywołań poszczególnych metod w jednej *metodzie szablonowej*. Etapy mogą być albo abstrakcyjne, albo posiadać jakąś domyślną implementację. Aby skorzystać z algorytmu, klient powinien dostarczyć swoją podklasę implementującą wszystkie etapy abstrakcyjne i nadpisać opcjonalne etapy jeśli jest taka potrzeba (oprócz samej metody szablonowej).

Zobaczmy jak się to sprawdzi w naszej aplikacji do eksploracji danych. Możemy stworzyć klasę bazową dla wszystkich trzech algorytmów parsowania. Ta klasa definiuje metodę szablono-

wą składającą się z ciągu wywołań różnych etapów przetwarzania dokumentu.



*Metoda szablonowa dzieli algorytm na etapy, umożliwiając podklasom ich nadpisywanie, ale nie samą metodę szablonową.*

Na początek możemy zadeklarować wszystkie etapy jako abstrakcyjne, zmuszając podklasy do ich zaimplementowania. W naszym przypadku podklasy już posiadają konieczne implementacje, więc jedyną rzeczą jaka pozostaje do zrobienia jest dostosowanie sygnatur metod tak, aby zgadzały się z metodami klasy bazowej.

A teraz zobaczymy, co da się zrobić by zlikwidować duplikacje kodu. Kod otwierania/zamykania pliku oraz ekstrakcji/parbowania wydaje się różnić pomiędzy formatami danych, więc nie ma sensu się tymi metodami zajmować. Jednakże imple-

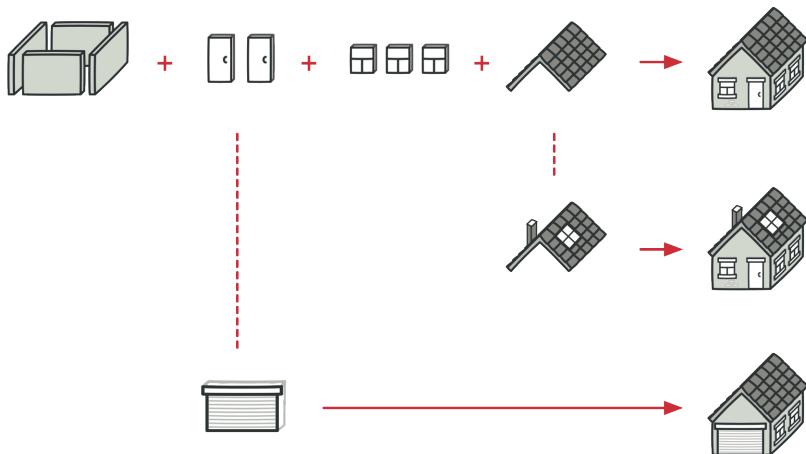
mentacja pozostałych etapów, jak analiza surowych danych i układania raportów, jest bardzo podobna. Można więc przeńieść te fragmenty do klasy bazowej, dzięki czemu podklasy będą mogły je współdzielić.

Jak widać, mamy dwa rodzaje etapów:

- *etapy abstrakcyjne*, które trzeba zaimplementować w każdej podklasie
- *etapy opcjonalne*, które mają już jakąś domyślną implementację, ale nadal mogą być nadpisane, jeśli zaistnieje taka potrzeba

Istnieje jednak jeszcze jeden rodzaj etapów, zwane *hookami*. Hook to opcjonalny, pusty etap. Metoda szablonowa będzie działać nawet jeśli nie nadpisze się hooków. Zazwyczaj umieszcza się je przed i po istotnych etapach algorytmu, dając tym samym podklasom dogodne punkty zaczepienia, mogące służyć rozbudowie algorytmu.

## 🚗 Analogia do prawdziwego życia

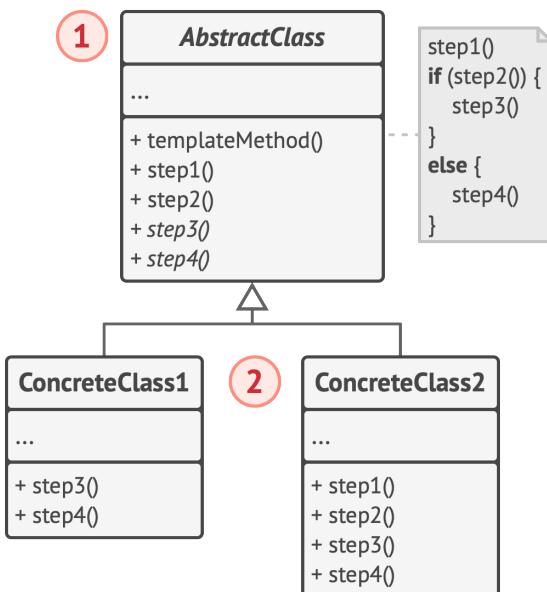


*Typowy projekt architektoniczny można nieco zmienić by zaspokoić potrzeby klienta.*

Opisywane tu podejście można zastosować w seryjnej budowie domów. Projekt architektoniczny standardowego domu może mieć wiele punktów zaczepienia, które pozwolą potencjalnemu właścicielowi dostosować finalną budowlę do swoich potrzeb.

Każdy etap budowy, jak kładzenie fundamentów, szkielet, wznoszenie ścian, instalację wodno-kanalizacyjną i elektryczną, itd., można nieco zmodyfikować, czyniąc dom odmiennym od reszty.

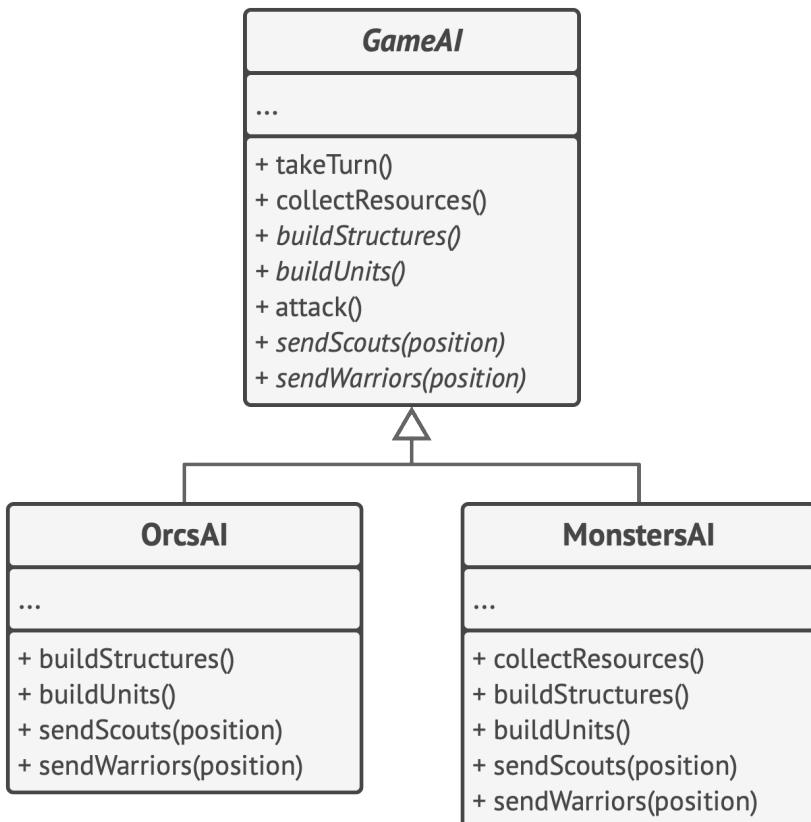
## Struktura



1. **Klasa Abstrakcyjna** deklaruje metody stanowiące etapy algorytmu oraz faktyczną metodę szablonową która wywołuje poszczególne etapy w określonej kolejności. Etapy mogą być zadeklarowane jako `abstrakcyjne`, albo posiadać domyślną implementację.
2. **Konkretne Klasy** mogą nadpisać każdy z etapów, oprócz samej metody szablonowej.

## # Pseudokod

W poniższym przykładzie, wzorzec **Metoda Szablonowa** daje “szkielet” dla różnorakich gałęzi sztucznej inteligencji w prostej grze strategicznej.



*Klasy SI prostej gry komputerowej.*

Wszystkie rasy mają niemal takie same typy jednostek i budynków. Możesz więc ponownie wykorzystać tę samą strukturę SI w poszczególnych rasach, nadpisując pewne szczegóły. Dzięki

takiemu podejściu SI orków czyni je agresywniejszymi, zaś ludzie charakteryzują się bardziej defensywną postawą. Ponadto, potwory nie potrafią wznosić budynków. Dodanie kolejnej rasy do gry wymagałoby stworzenia nowej podklasy SI i nadpisania domyślnych metod zadeklarowanych w klasie bazowej SI.

```
1 // Klasa abstrakcyjna definiuje metodę szablonową która zawiera
2 // szkielet jakiegoś algorytmu skomponowany w formie zestawu
3 // wywołań prostych, abstrakcyjnych działań. Konkretnie podklasy
4 // implementują te działania, ale pozostawiają samą metodę
5 // szablonową nietkniętą.
6 class GameAI is
7     // Metoda szablonowa definiuje szkielet algorytmu.
8     method turn() is
9         collectResources()
10        buildStructures()
11        buildUnits()
12        attack()
13
14     // Niektóre etapy mogą zostać zaimplementowane już w klasie
15     // bazowej.
16     method collectResources() is
17         foreach (s in this.builtStructures) do
18             s.collect()
19
20     // A inne mogą być zdefiniowane jako abstrakcyjne.
21     abstract method buildStructures()
22     abstract method buildUnits()
23
24     // Klasa może mieć wiele metod szablonowych.
```

```
25  method attack() is
26      enemy = closestEnemy()
27      if (enemy == null)
28          sendScouts(map.center)
29      else
30          sendWarriors(enemy.position)
31
32  abstract method sendScouts(position)
33  abstract method sendWarriors(position)
34
35 // Konkretne klasy muszą zaimplementować wszystkie abstrakcyjne
36 // działania klasy bazowej, ale nie mogą nadpisać samej metody
37 // szablonowej.
38 class OrcsAI extends GameAI is
39     method buildStructures() is
40         if (there are some resources) then
41             // Buduj farmy, potem koszary, potem twierdzę.
42
43     method buildUnits() is
44         if (there are plenty of resources) then
45             if (there are no scouts)
46                 // Buduj piechura, dodaj go do grupy zwiadowców.
47             else
48                 // Buduj żołnierza, dodaj go do grupy
49                 // wojowników.
50
51     // ...
52
53     method sendScouts(position) is
54         if (scouts.length > 0) then
55             // Wyślij zwiadowców na pozycję.
56
```

```
57 method sendWarriors(position) is
58     if (warriors.length > 5) then
59         // Wyślij wojowników na pozycję.
60
61 // Podklasy mogą też nadpisać niektóre działania domyślną
62 // implementacją.
63 class MonstersAI extends GameAI is
64     method collectResources() is
65         // Potwory nie zbierają zasobów.
66
67     method buildStructures() is
68         // Potwory nie wznoszą budowli.
69
70     method buildUnits() is
71         // Potwory nie budują jednostek.
```

## 💡 Zastosowanie

- ⚡ **Stosuj wzorzec Metoda szablonowa gdy chcesz pozwolić klientom na rozszerzanie niektórych tylko etapów algorytmu, ale nie całego, ani też jego struktury.**
- ⚡ **Metoda szablonowa pozwala zmienić monolityczny algorytm w ciąg pojedynczych etapów, które można następnie łatwo rozszerzać w podklasach bez naruszania struktury opisanej w klasie bazowej.**
- ⚡ **Wzorzec ten jest przydatny gdy masz wiele klas zawierających niemal identyczne algorytmy różniące się jedynie szczegółami.**

**W takiej sytuacji bowiem konieczność modyfikacji algorytmu skutkuje koniecznością modyfikacji wszystkich klas.**

- ⚡ Zmieniając taki algorytm w metodę szablonową możesz także przenieść jego etapy o podobnej implementacji do klasy bazowej, eliminując tym samym duplikację kodu. Kod który różni się pomiędzy podklasami, może w nich pozostać.

## Jak zaimplementować

1. Przeanalizuj algorytm docelowy pod kątem możliwego podziału na etapy. Rozważ które z nich są wspólne dla wszystkich podklas, a które zawsze będą unikalne.
2. Stwórz abstrakcyjną klasę bazową i zadeklaruj metodę szablonową oraz zestaw abstrakcyjnych metod reprezentujących etapy algorytmu. Nakreśl strukturę algorytmu w metodzie szablonowej poprzez uruchamianie odpowiednich etapów. Rozważ zastosowanie słowa kluczowego `final` w stosunku do metody szablonowej aby zapobiec nadpisaniu jej przez podklasy.
3. Może się zdarzyć, że wszystkie etapy pozostaną abstrakcyjnymi. Jednak niektóre z nich skorzystałyby na posiadaniu domyślnej implementacji. Podklasy nie muszą implementować tych metod.
4. Zastanów się nad dodaniem hooków pomiędzy kluczowymi etapami algorytmu.

5. Dla każdego wariantu algorytmu stwórz nową konkretną podklasę. *Musi* ona implementować wszystkie abstrakcyjne etapy, ale *może* także nadpisać część opcjonalnych.

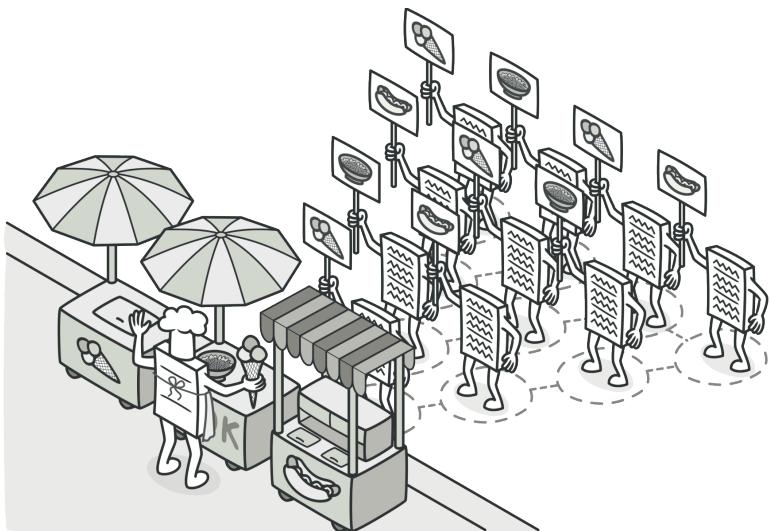
## Zalety i wady

- ✓ Można pozwolić klientom nadpisać tylko niektóre partie dużego algorytmu czyniąc go odporniejszym na szkody wskutek zmian poszczególnych jego części.
- ✓ Można przenieść powtarzający się kod do klasy bazowej.
- ✗ Dla niektórych klientów przygotowany szkielet algorytmu może stanowić ograniczenie.
- ✗ Może prowadzić do naruszenia *Zasady podstawienia Liskov* wskutek stłumienia domyślnych implementacji etapów w podklasach.
- ✗ Metody szablonowe zwykle trudniej utrzymywać w miarę jak przybywa etapów.

## Powiązania z innymi wzorcami

- **Metoda wytwórcza** to wyspecjalizowana **Metoda szablonowa**. Może stanowić także jeden z etapów większej *Metody szablonowej*.
- **Metoda szablonowa** polega na mechanizmie dziedziczenia: pozwala zmieniać części algorytmu rozszerzając je w podklasach. **Strategia** bazuje na kompozycji: można zmienić część zachowania.

wania obiektu poprzez nadanie mu różnych strategii odpowiadających temu zachowaniu. *Metoda szablonowa* działa na poziomie klasy, więc jest statyczna. *Strategia* działa na poziomie obiektu, więc pozwala przełączać zachowania w trakcie działania programu.



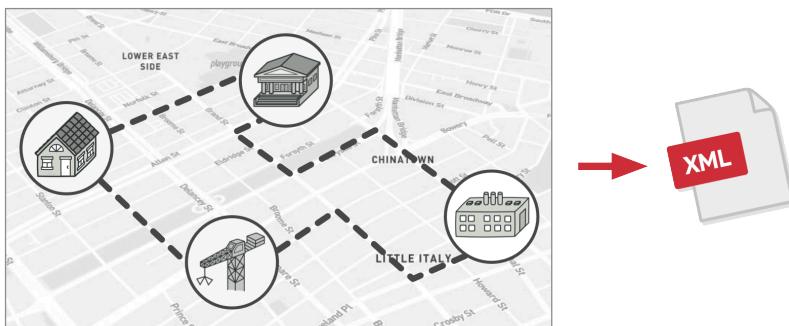
# ODWIEDZAJĄCY

*Znany też jako: Visitor*

**Odwiedzający** to behawioralny wzorzec projektowy pozwalający oddzielić algorytmy od obiektów na których pracują.

## (:() Problem

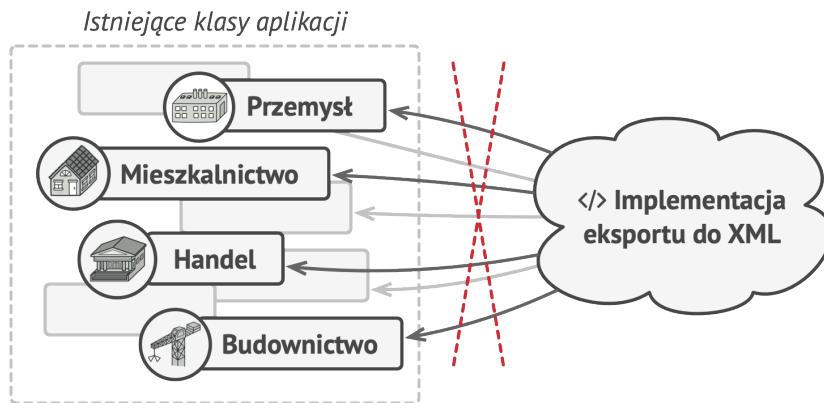
Wyobraź sobie, że twój zespół opracowuje aplikację korzystającą z danych geograficznych ustrukturyzowanych w jeden wielki graf. Każdy węzeł grafu odpowiada złożonemu podmiotowi jak miasto, ale również bardziej szczegółowym elementom, takim jak obiekty przemysłowe, atrakcje turystyczne, itp. Węzły są połączone ze sobą jeśli istnieje droga pomiędzy faktycznymi obiektami jakie reprezentują. Za kulisami każdy typ węzła reprezentowany jest przez osobną klasę, zaś poszczególne węzły to ich obiekty.



*Eksportowanie grafu do XML.*

W którymś momencie otrzymujesz zadanie implementacji eksportu grafu do formatu XML. Początkowo zadanie wydało ci się proste. Planujesz dodanie metody eksportującej do każdej klasy węzła, a następnie rekursywne wywołanie jej w każdym z węzłów. Rozwiążanie wydaje się proste i eleganckie: dzięki polimorfizmowi nie doszło do spręgnięcia kodu wywołującego metodę eksportującą z konkretnymi klasami węzłów.

Niestety architekt systemu odmówił zgody na modyfikację istniejących klas węzłów. Stwierdził, że kod jest już na etapie produkcji i nie może sobie pozwolić na ryzyko związane z wprowadzeniem ewentualnego błędu wraz z twoimi zmianami.



*Do wszystkich klas węzłów trzeba było dodać metodę eksportu do XML. Z wdrażaniem zmian wiązało się ryzyko wprowadzenia błędu do aplikacji.*

Architekt systemu miał również wątpliwości co do sensu umieszczania kodu eksportu do XML w klasach węzłów. Przecież głównym zadaniem tych klas jest działanie na danych geograficznych, a obecność kodu eksportu do XML wyglądały nie na miejscu.

Istniał też jeszcze jeden powód odmowy. Było bowiem bardzo możliwe, że po implementacji tej funkcjonalności, ktoś z działu marketingu poprosiłby o dodanie możliwości eksportu do jakiegoś innego formatu, lub o inne dziwactwa. Zmusiłoby to

ciebie do ponownych zmian w tych drogocennych, delikatnych klasach.

## 😊 Rozwiążanie

Wzorzec projektowy Odwiedzający proponuje umieszczenie nowych obowiązków w osobnej klasie zwanej *odwiedzającym*, zamiast próbować zintegrować je z istniejącymi klasami. Pierwotny obiekt, który miał wykonywać te obowiązki, teraz jest przekazywany do jednej z metod odwiedzającego w charakterze argumentu. Daje to metodzie dostęp do wszystkich potrzebnych danych znajdujących się w obiekcie.

Ale co jeśli te czynności można wykonać także na obiektach-węzłach innych klas? W naszym przykładzie z eksportem do XML, faktyczna implementacja zapewne będzie się nieco różnić pomiędzy poszczególnymi klasami węzłów. Dlatego też klasa odwiedzający może definiować nie jedną, ale cały zestaw metod, z których każda przyjmuje argumenty różnych typów, jak na poniższym przykładzie:

```
1 class ExportVisitor implements Visitor is
2     method doForCity(City c) { ... }
3     method doForIndustry(Industry f) { ... }
4     method doForSightSeeing(SightSeeing ss) { ... }
5     // ...
```

Ale jak właściwie wywoływalibyśmy te metody, zwłaszcza mając do czynienia z całym grafem? Metody z zestawu mają różne sygnatury, więc nie możemy zastosować polimorfizmu. Aby wybrać taką metodę odwiedzającego, która będzie w stanie obsłużyć dany obiekt, trzeba znać jego klasę. Brzmi koszmarnie, prawda?

```
1 foreach (Node node in graph)
2     if (node instanceof City)
3         exportVisitor.doForCity((City) node)
4     if (node instanceof Industry)
5         exportVisitor.doForIndustry((Industry) node)
6     // ...
7 }
```

Być może zastanawiasz się, dlaczego nie zastosujemy w tym miejscu przeciążenia metod? Polegałoby to na nadaniu wszystkim metodom tej samej nazwy, mimo że przyjmują inne parametry. Niestety, nawet zakładając że stosowany język programowania posiada tę funkcjonalność (jak Java i C#), w niczym nam to nie pomoże. Klasa konkretnego obiektu węzła jest zawsze nieznana, więc mechanizm przeciążania nie będzie w stanie określić właściwej metody którą trzeba wywołać. Domyślnym działaniem w takim przypadku będzie wywołanie metody która przyjmuje obiekt klasy bazowej **Węzeł**.

Wzorzec projektowy Odwiedzający odwołuje się właśnie do takiego problemu. Stosuje technikę zwaną **Podwójną dyspozycją**, która pozwala wywołać odpowiednią metodę obiektu bez

uciekania się do instrukcji warunkowych. Zamiast pozwalać klientowi wybrać odpowiednią wersję metody, można oddelegować ten wybór samym obiektom przekazywanym odwiedzającemu w charakterze argumentu.

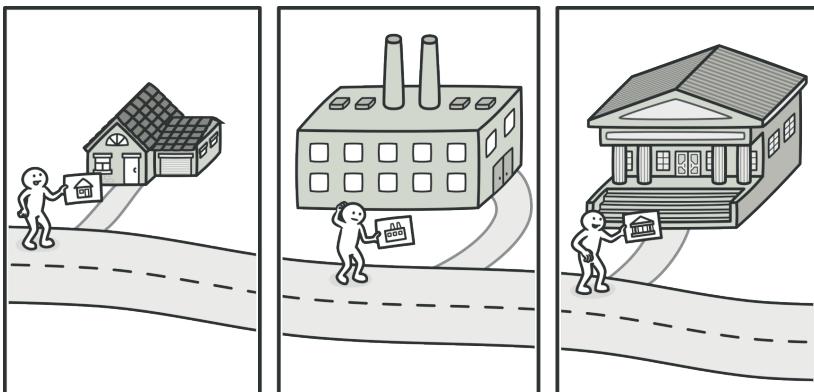
Skoro obiekty wiedzą jakie są klasy, będą w stanie wybrać właściwą metodę odwiedzającego w naturalniejszy sposób. "Przyjmują" one odwiedzającego i informują którą jego metodę należy wywołać.

```
1 // Kod Klienta
2 foreach (Node node in graph)
3     node.accept(exportVisitor)
4
5 // Miasto
6 class City is
7     method accept(Visitor v) is
8         v.doForCity(this)
9         // ...
10
11 // Przemysł
12 class Industry is
13     method accept(Visitor v) is
14         v.doForIndustry(this)
15     // ...
```

Przyznaję – jednak musielibyśmy zmienić klasy węzłów. Ale za to zmiana jest trywialna i umożliwia dalsze ewentualne dodawanie obowiązków już bez konieczności ponownej zmiany klas.

Jeśli uda się wyekstrahować wspólny interfejs wszystkich odwiedzających, wszystkie istniejące węzły będą mogły współpracować z dowolnym odwiedzającym jakiego dodamy do aplikacji. Jeśli będzie trzeba wprowadzić jakieś nowe czynności związane z węzłami, wystarczy zaimplementować nową klasę odwiedzającego.

## 🚗 Analogia do prawdziwego życia



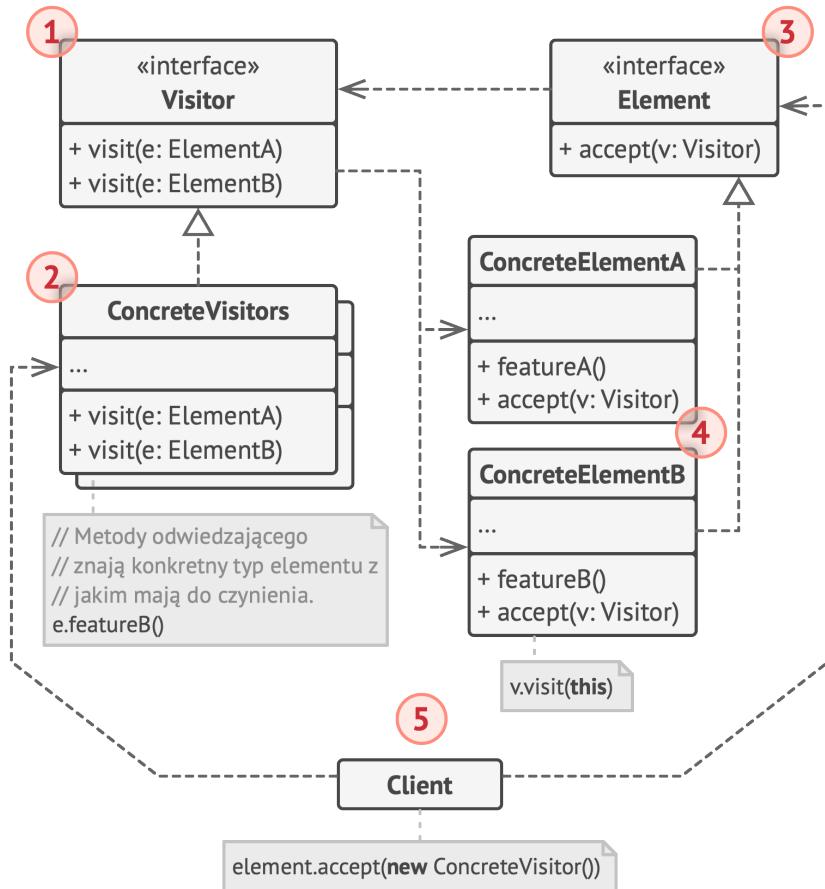
*Dobry agent ubezpieczeniowy jest gotów zawsze zaoferować polisę odpowiednią do danej organizacji.*

Wyobraźmy sobie doświadczonego agenta ubezpieczeniowego który chce pozyskać nowych klientów. Może odwiedzić wszystkie budynki danej dzielnicy, próbując sprzedać polisy każdemu kogo napotka. Zależnie od rodzaju organizacji znajdującej się w budynku, może zaoferować specjalistyczne polisy:

- Jeśli jest to obiekt mieszkalny, sprzedaje ubezpieczenie zdrowotne.

- Jeśli jest to bank, sprzedaje ubezpieczenie od kradzieży.
- Jeśli jest to kawiarnia, sprzedaje ubezpieczenie od ognia i wody.

## Struktura



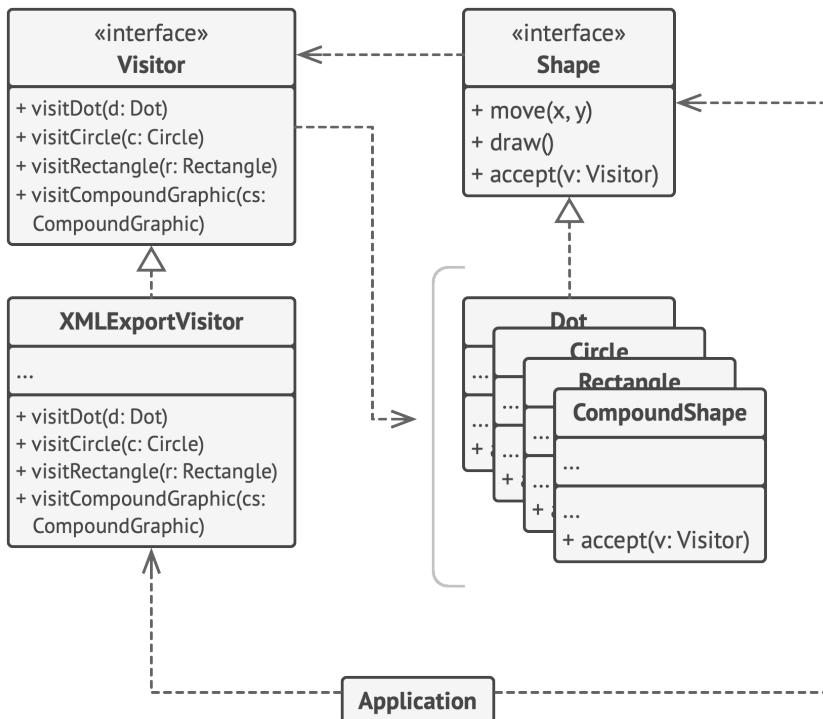
1. Interfejs **Odwiedzający** deklaruje zestaw metod odwiedzania które przyjmują w charakterze argumentów konkretne elementy struktury obiektu. Metody te mogą mieć takie same nazwy

jeśli stosowany jest język programowania obsługujący przeciążanie, ale typ parametrów musi być różny.

2. Każdy **Konkretny Odwiedzający** implementuje wiele wersji tego samego zachowania, dostosowane do różnych konkretnych klas elementów.
3. Interfejs **Element** deklaruje metodę służącą “przyjmowaniu” odwiedzających. Typem przyjmowanego parametru takiej metody powinien być interfejs odwiedzającego.
4. Każdy **Konkretny Element** musi implementować metodę przyjmowania. Zadaniem tej metody jest przekierowanie wywołania do właściwej metody odwiedzającego, odpowiadającej bieżącej klasie elementu. Trzeba pamiętać, że nawet jeśli bazowa klasa elementu implementuje tę metodę, wszystkie podklasy muszą ją nadpisywać w swoich klasach i wywoływać stosowną metodę obiektu odwiedzającego.
5. **Klient** to na ogół kolekcja lub inny złożony obiekt (na przykład drzewo **Kompozytowe**). Na ogół klienci nie są świadomi wszystkich konkretnych klas swoich elementów, gdyż współpracują z nimi za pośrednictwem jakiegoś abstrakcyjnego interfejsu.

## # Pseudokod

W poniższym przykładzie, wzorzec **Odwiedzający** służy wypo- sażaniu hierarchii klas figur geometrycznych we wsparcie eks- portu do XML.



*Eksport różnych typów obiektów do formatu XML za pomocą obiektu odwiedzającego.*

```

1 // Interfejs elementu deklaruje metodę `accept` która przyjmuje
2 // argument typu interfejs bazowy odwiedzającego.
3 interface Shape is
4     method move(x, y)
  
```

```
5   method draw()
6   method accept(v: Visitor)
7
8 // Każda konkretna klasa elementu musi implementować metodę
9 // `accept` w taki sposób, aby wywoływała tę metodę
10 // odwiedzającego, która odpowiada klasie elementu.
11 class Dot implements Shape is
12   // ...
13
14 // Zwróć uwagę, że wywołujemy `visitDot`, a więc metodę
15 // zgodną z nazwą bieżącej klasy. Dzięki temu informujemy
16 // odwiedzającego o klasie elementu z jakim współpracuje.
17 method accept(v: Visitor) is
18   v.visitDot(this)
19
20 class Circle implements Shape is
21   // ...
22   method accept(v: Visitor) is
23     v.visitCircle(this)
24
25 class Rectangle implements Shape is
26   // ...
27   method accept(v: Visitor) is
28     v.visitRectangle(this)
29
30 class CompoundShape implements Shape is
31   // ...
32   method accept(v: Visitor) is
33     v.visitCompoundShape(this)
34
35
36 // Interfejs odwiedzającego deklaruje zestaw metod służących
```

```
37 // odwiedzaniu, które odpowiadają poszczególnym klasom
38 // elementów. Sygnatura metody odwiedzającej pozwala
39 // odwiedzającemu określić dokładną klasę elementu z jakim ma do
40 // czynienia.
41 interface Visitor is
42     method visitDot(d: Dot)
43     method visitCircle(c: Circle)
44     method visitRectangle(r: Rectangle)
45     method visitCompoundShape(cs: CompoundShape)
46
47 // Konkretni odwiedzający implementują wiele wersji tego samego
48 // algorytmu, które mogą działać ze wszystkimi konkretnymi
49 // klasami elementów.
50 //
51 // Zaobserwuje się najwięcej korzyści ze stosowania wzorca
52 // Odwiedzający, gdy ma się do czynienia ze złożoną strukturą
53 // obiektów, taką jak drzewo kompozytowe. W takim przypadku
54 // pomocne może być przechowanie jakiegoś pośredniego stanu
55 // algorytmu w czasie uruchamiania metod odwiedzającego na
56 // kolejnych obiektach struktury.
57 class XMLExportVisitor implements Visitor is
58     method visitDot(d: Dot) is
59         // Eksportuj ID i współrzędne środka kropki.
60
61     method visitCircle(c: Circle) is
62         // Eksportuj ID okręgu, współrzędne środka i promień.
63
64     method visitRectangle(r: Rectangle) is
65         // Eksportuj ID prostokąta, współrzędne lewego górnego
66         // wierzchołka, szerokość i wysokość.
67
68     method visitCompoundShape(cs: CompoundShape) is
```

```
69     // Eksportuj ID figury oraz listę ID składających się na
70     // nią.
71
72
73 // Kod klienta może uruchamiać działania odwiedzającego na
74 // dowolnym zestawie elementów bez konieczności ustalania ich
75 // konkretnych klas. Operacja przyjmująca kieruje wywołanie do
76 // odpowiedniego działania obiektu odwiedzającego.
77 class Application is
78     field allShapes: array of Shapes
79
80     method export() is
81         exportVisitor = new XMLExportVisitor()
82
83         foreach (shape in allShapes) do
84             shape.accept(exportVisitor)
```

Jeśli zastanawiasz się nad celowością metody `przyjmującej` w tym przykładzie, mój artykuł [Odwiedzający i podwójna dyspozycja](#) szczegółowo tłumaczy tę kwestię.

## 💡 Zastosowanie

- ⚡ Stosuj wzorzec Odwiedzający gdy istnieje potrzeba wykonywania jakiegoś działania na wszystkich elementach złożonej struktury obiektów (jak drzewo obiektów).
- ⚡ Wzorzec Odwiedzający pozwala wykonać jakieś działanie na zestawie obiektów różnych klas dzięki istnieniu obiektu odwiedzającego. On z kolei implementuje wiele wariantów tego

działania, odpowiadających poszczególnym klasom docelowym.

 **Stosowanie Odwiedzającego pozwala uprątnać logikę biznesową czynności pomocniczych.**

 Wzorzec Odwiedzający daje możliwość ograniczenia zakresu obowiązków głównych klas aplikacji tylko do tych najważniejszych poprzez ekstrakcję wszystkich innych obowiązków do zestawu klas odwiedzających.

 **Warto stosować ten wzorzec gdy jakieś zachowanie ma sens tylko w kontekście niektórych klas wchodzących w skład hierarchii klas, ale nie wszystkich.**

 Możesz wyekstrahować główne obowiązki do osobnej klasy odwiedzający i zaimplementować tylko te metody odwiedzania, które przyjmują obiekty istotnych klas, zaś resztę metod pozostawić pustą.

## Jak zaimplementować

1. Zadeklaruj interfejs odwiedzający z zestawem metod “odwiedzania”, po jednej na każdą konkretną klasę elementu istniejącą w programie.
2. Zadeklaruj interfejs elementu. Jeśli pracujesz z istniejącym elementem hierarchii klas, dodaj abstrakcyjną metodę “przyjmu-

jącą” do klasy bazowej hierarchii. Metodzie tej będzie się przekazywać w charakterze argumentu obiekt odwiedzającego.

3. Zaimplementuj metody przyjmujące we wszystkich konkretnych klasach elementów. Metody te muszą przekierowywać wywołanie do tej metody odwiedzania otrzymanego obiektu odwiedzającego, która odpowiada klasie bieżącego elementu.
4. Klasa elementów powinny współpracować z odwiedzającymi wyłącznie za pośrednictwem interfejsu odwiedzającego. Odwiedzający jednak muszą być świadomi wszystkich klas konkretnych elementów, do których odnoszą się typy parametrów metod odwiedzających.
5. Dla każdego zachowania którego nie da się zaimplementować w obrębie hierarchii elementów, należy utworzyć nową konkretną klasę odwiedzającego i zaimplementować wszystkie metody odwiedzania.

Możesz natknąć się na sytuację w której odwiedzający będzie potrzebował dostępu do jakichś prywatnych pól klasy elementu. W takim przypadku można albo uczynić te pola i metody publicznymi, psując jednak tym samym hermetyzację elementu, albo zagnieździć klasę odwiedzającego w klasie elementu. To drugie możliwe jest tylko wtedy gdy (szczęśliwie) masz do czynienia z językiem programowania obsługującym zagnieżdżanie klas.

6. Klient musi tworzyć obiekty odwiedzającego i przekazywać je elementom za pośrednictwem metod “przyjmujących”.

## ΔΔ Zalety i wady

- ✓ *Zasada otwarte/zamknięte.* Pozwala wprowadzać nowe zachowanie odnoszące się do obiektów różnych klas bez konieczności zmiany tych klas.
- ✓ *Zasada pojedynczej odpowiedzialności.* Można przenieść kilka wersji danego zachowania do jednej klasy.
- ✓ Obiekt odwiedzający może zebrać użyteczne informacje współpracując z różnymi obiektami. Może się to przydać, gdy zastanieje potrzeba przejrzenia złożonej struktury danych element po elemencie (takiej jak drzewo obiektów) i zastosowania odwiedzającego do każdego obiektu struktury.
- ✗ Trzeba zaktualizować wszystkich odwiedzających za każdym razem gdy hierarchia elementów zyskuje nową klasę lub ktoś traci.
- ✗ Odwiedzający mogą nie mieć dostępu do prywatnych pól i metod elementów z którymi mają współpracować.

## ↔ Powiązania z innymi wzorcami

- Wzorzec **Odwiedzający** można traktować jak późniejszą wersję **Polecenia**. Jego obiekty mogą wykonywać różne poleceńia na obiektach różnych klas.

- **Odwiedzający** może wykonać działanie na całym drzewie **Kompozytowym**.
- Połączenie **Odwiedzającego** z **Iteratorem** może służyć sekwenциальнemu przeglądowi elementów złożonej struktury danych i wykonaniu na nich jakiegoś działania, nawet jeśli te elementy są obiektami różnych klas.

# Zakończenie

**Brawo! Udało ci się dotrzeć do końca książki!**

Na świecie istnieje jednak jeszcze wiele innych wzorców projektowych. Mam nadzieję, że niniejsza książka będzie dobrym punktem wyjścia dla dalszej nauki wzorców i rozwijania programistycznych supermocy projektowania.

Poniższa lista zawiera propozycje dalszego działania.

- </> Nie zapomnij, że masz dostęp do archiwum próbek kodu do pobrania w różnych językach programowania.
- ☰ Przeczytaj “Refaktoryzację do wzorców projektowych” Joshua Kerievskiego.
- 🎓 Nie wiesz nic o refaktoryzacji? Mam dla ciebie kurs.
- ☰ Wydrukuj sobie te ściągawki ze wzorców projektowych i zawsze miej je pod ręką.
- 💬 Zostaw komentarz do książki. Chętnie poznam twoje zdanie, nawet jeśli będzie wysoce krytyczne 😊