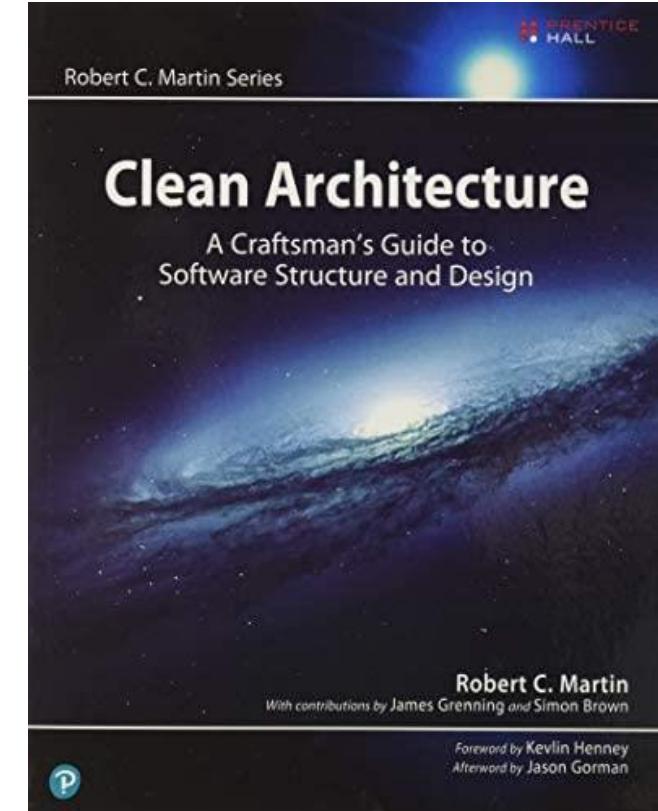


SOLID Principles of Object Oriented Design

Robert C. Martin (2000 r)

Rober C. Martin (Unkle Bob)

Opublikowana w 2017



SOLID Principles of Object Oriented Design

(sformułowane w 2000r. w artykule Design Principles and Design Patterns)

S

Reguła jednej odpowiedzialności

SRP(Single Responsibility Principle)

S – Single Responsibility Principle



Każdy moduł powinien mieć jedną i tylko jedną przyczynę zmiany.

Klasa powinna mieć tylko jedną odpowiedzialność, czyli albo gotowanie, albo zmywanie, ale nigdy to i to.

SOLID Principles of Object Oriented Design

S

Reguła jednej odpowiedzialności

SRP(Single Responsibility Principle)

O

Reguła otwarте-zamknięte

OCP(Open-Close Principle)

O - Open Close Principle

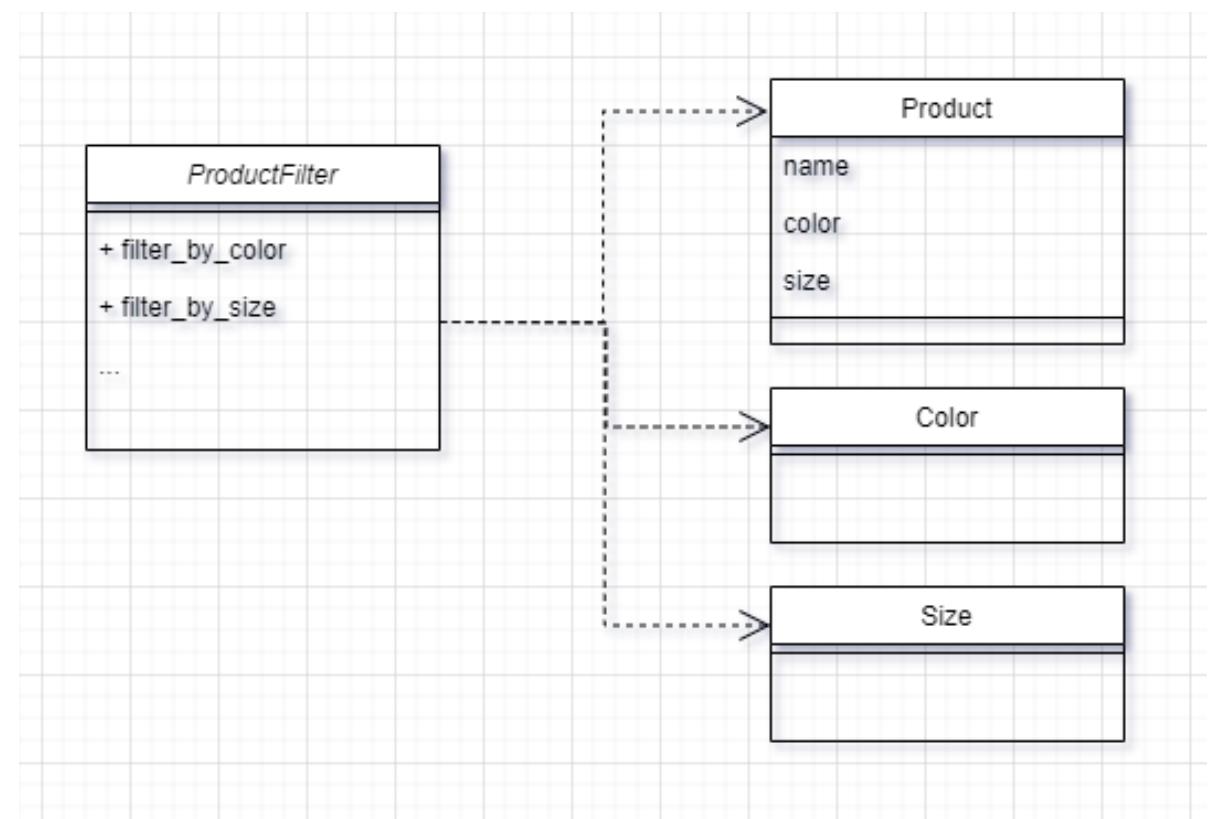


Element oprogramowania powinien być otwarty na rozbudowę, a zamknięty na modyfikacje.

Klasa powinna być otwarta na rozbudowę (przeważnie poprzez zastosowanie dziedziczenia), ale zamknięta na modyfikacje.

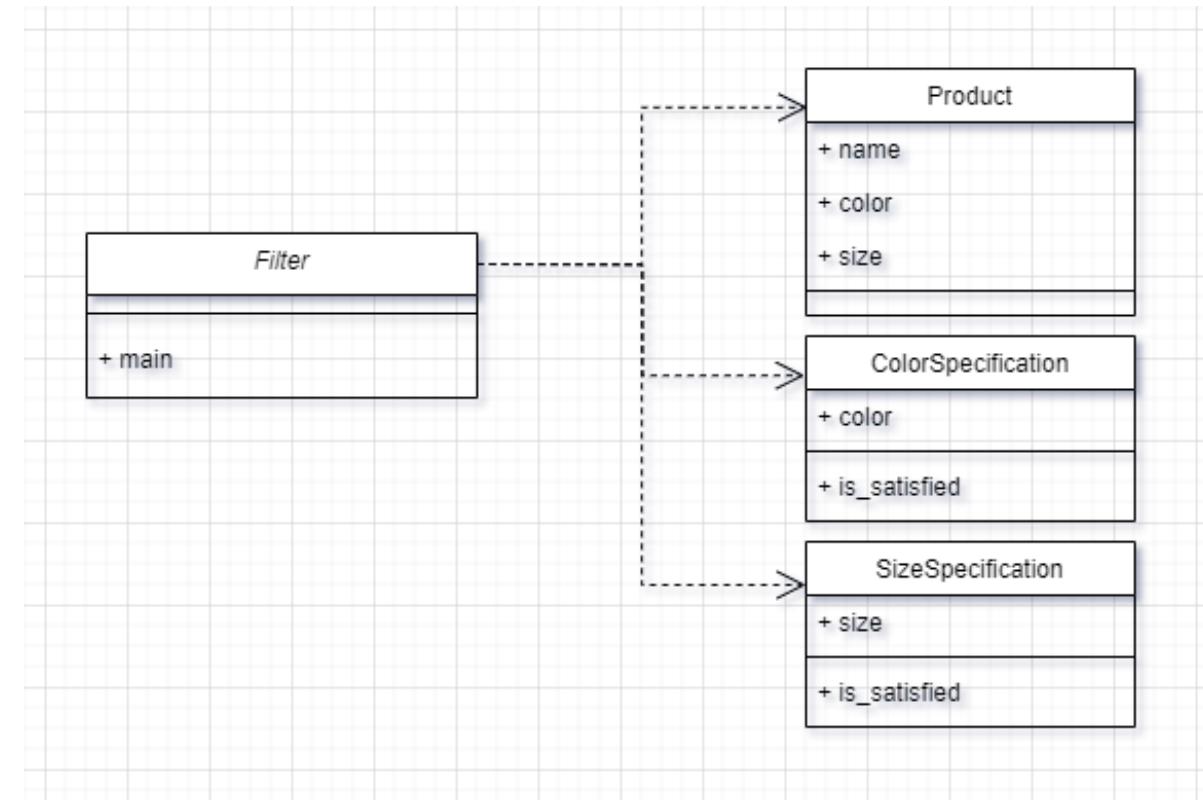
OPC

Open Close Principle - before



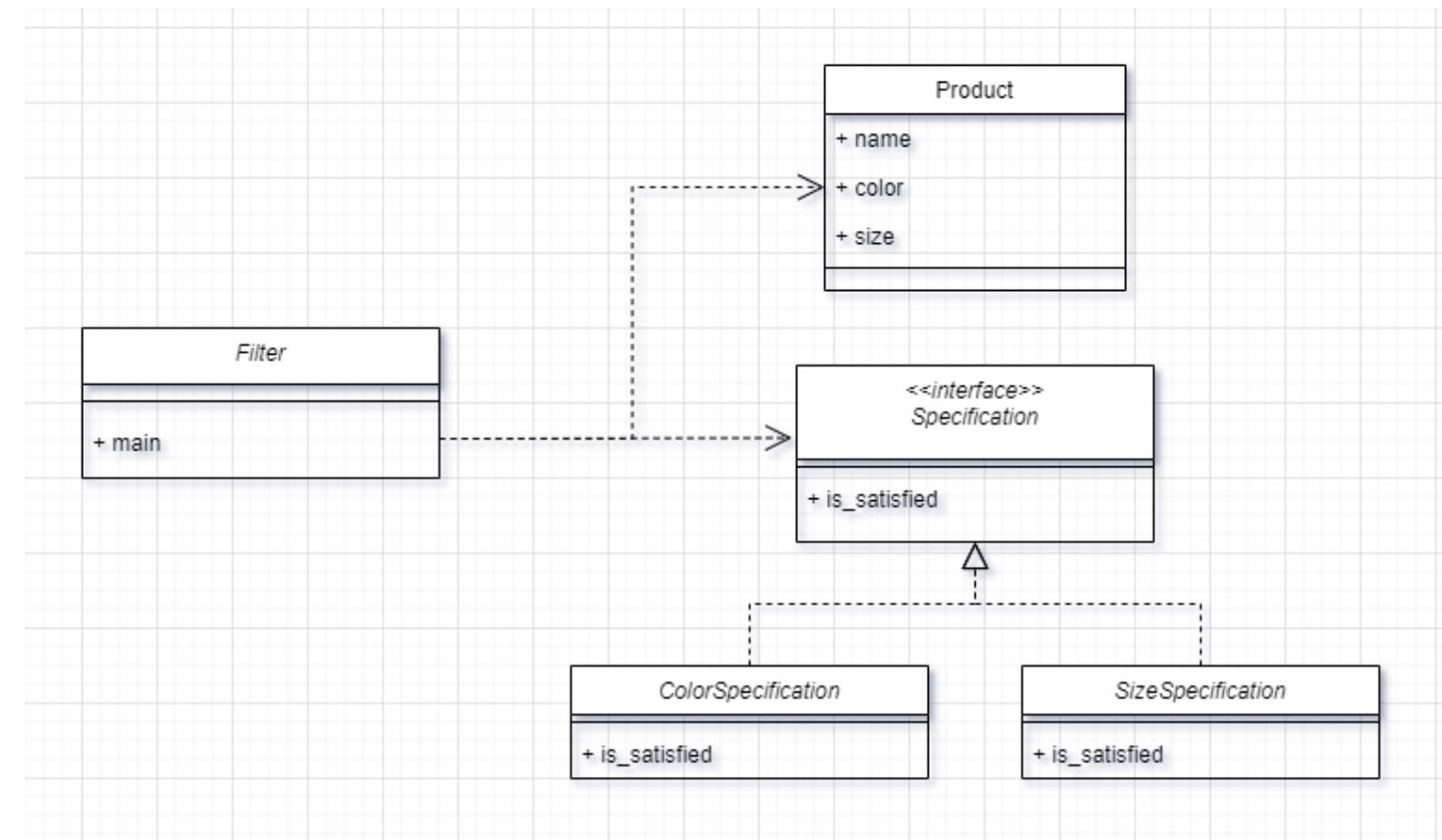
OPC

Open Close Principle - after



OPC

Open Close Principle – after 2



SOLID Principles of Object Oriented Design

S

Reguła jednej odpowiedzialności

SRP(Single Responsibility Principle)

O

Reguła otwarте-zamknięte

OCP(Open-Close Principle)

L

Zasada podstawień Barbary
Liskov

LSP (Liskov Substitution Principle)

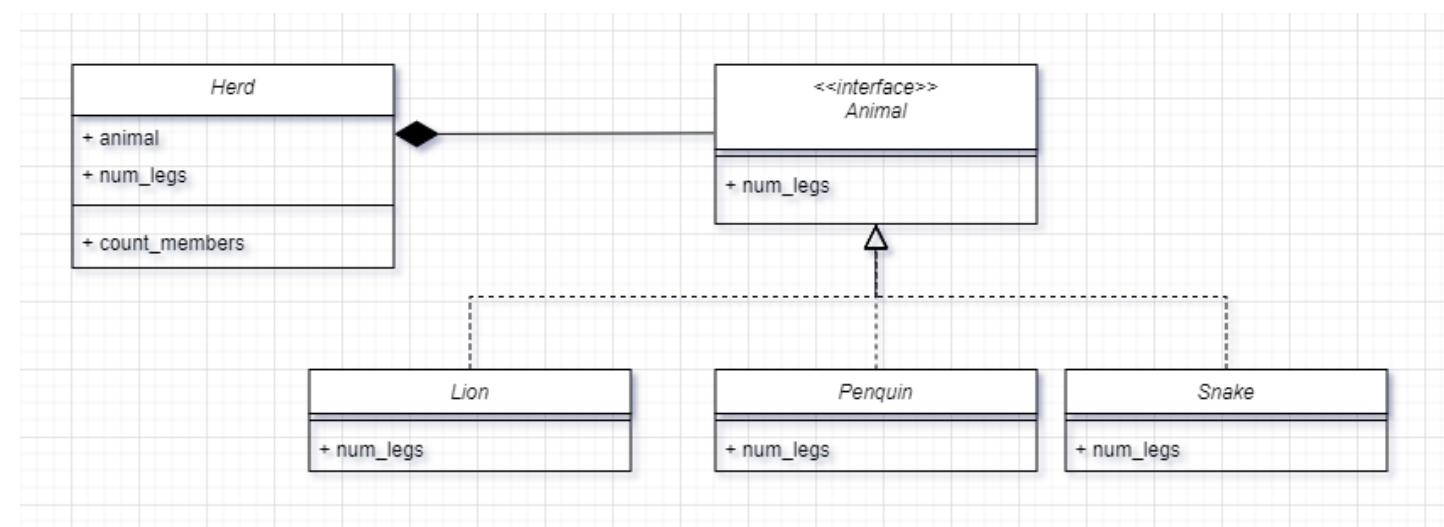
L – Liskov Substitution Principle



Podklasy powinny być w stanie zastąpić swoich rodziców bez wywoływania błędu w programie.

LSP

Liskov Substitution Principle



SOLID Principles of Object Oriented Design

S

Reguła jednej odpowiedzialności

SRP (Single Responsibility Principle)

O

Reguła otwarте-zamknięte

OCP (Open-Close Principle)

L

Zasada podstawień Barbary Liskov

LSP (Liskov Substitution Principle)

I

Zasada rozdzielenia interfejsów

ISP (Interface Segregation Principle)

I – Interface Segregation Principle



Wiele różnych interfejsów jest lepsze niż jeden interfejs typu do-it-all

SOLID Principles of Object Oriented Design

S

Reguła jednej odpowiedzialności

SRP (Single Responsibility Principle)

O

Reguła otwarте-zamknięte

OCP (Open-Close Principle)

L

Zasada podstawień Barbary Liskov

LSP (Liskov Substitution Principle)

I

Zasada rozdzielenia interfejsów

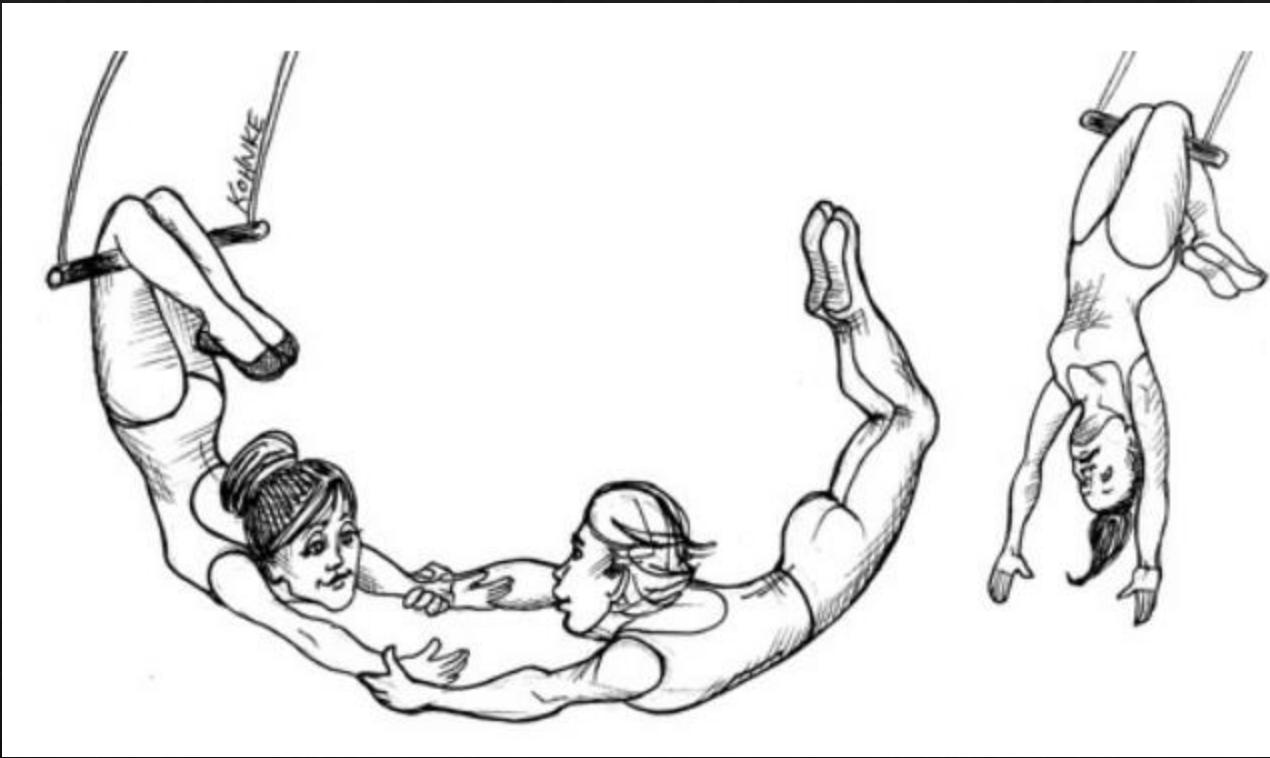
ISP (Interface Segregation Principle)

D

Zasada odwrócenia zależności

DIP (Dependency Inversion Principle)

D – Dependency Inversion Principle



Powinniśmy programować za pomocą abstrakcji, nie implementacji. Implementacje mogą się zmieniać, abstrakcje nie powinny

DIP

Dependency Inversion Principle - before

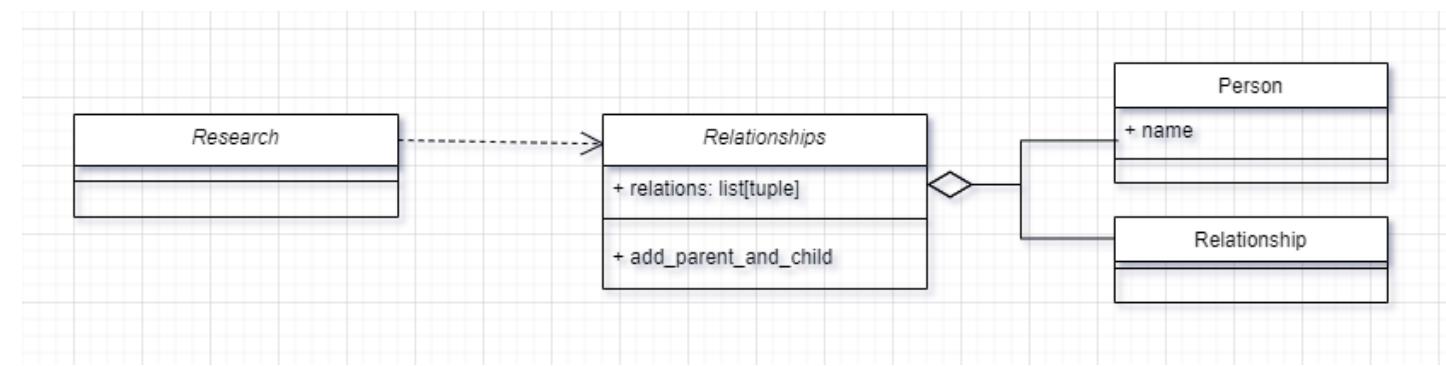
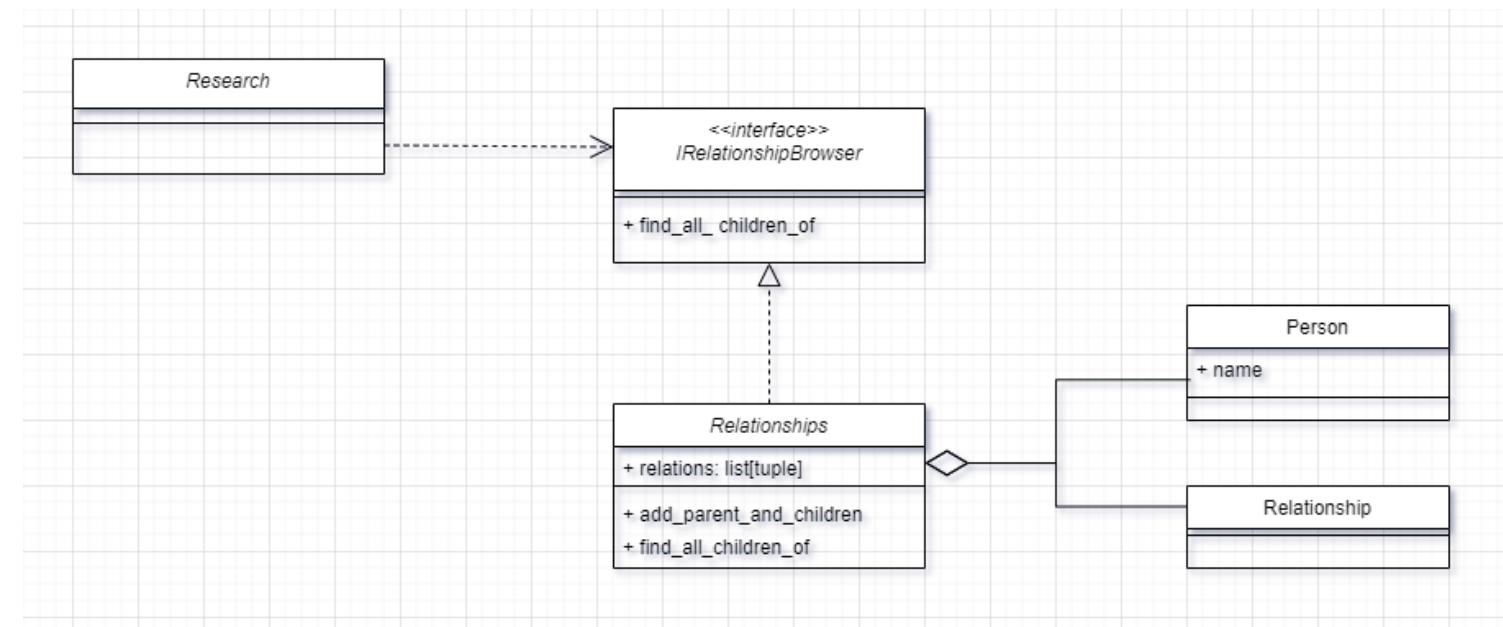


Diagram klas - relacje

Asocjacja *ang. association*



SOLID Principles of Object Oriented Design

S

Reguła jednej odpowiedzialności

SRP (Single Responsibility Principle)

O

Reguła otwarте-zamknięte

OCP (Open-Close Principle)

L

Zasada podstawień Barbary Liskov

LSP (Liskov Substitution Principle)

I

Zasada rozdzielenia interfejsów

ISP (Interface Segregation Principle)

D

Zasada odwrócenia zależności

DIP (Dependency Inversion Principle)

STUPID - antySOLID

??? (??? r)

STUPID

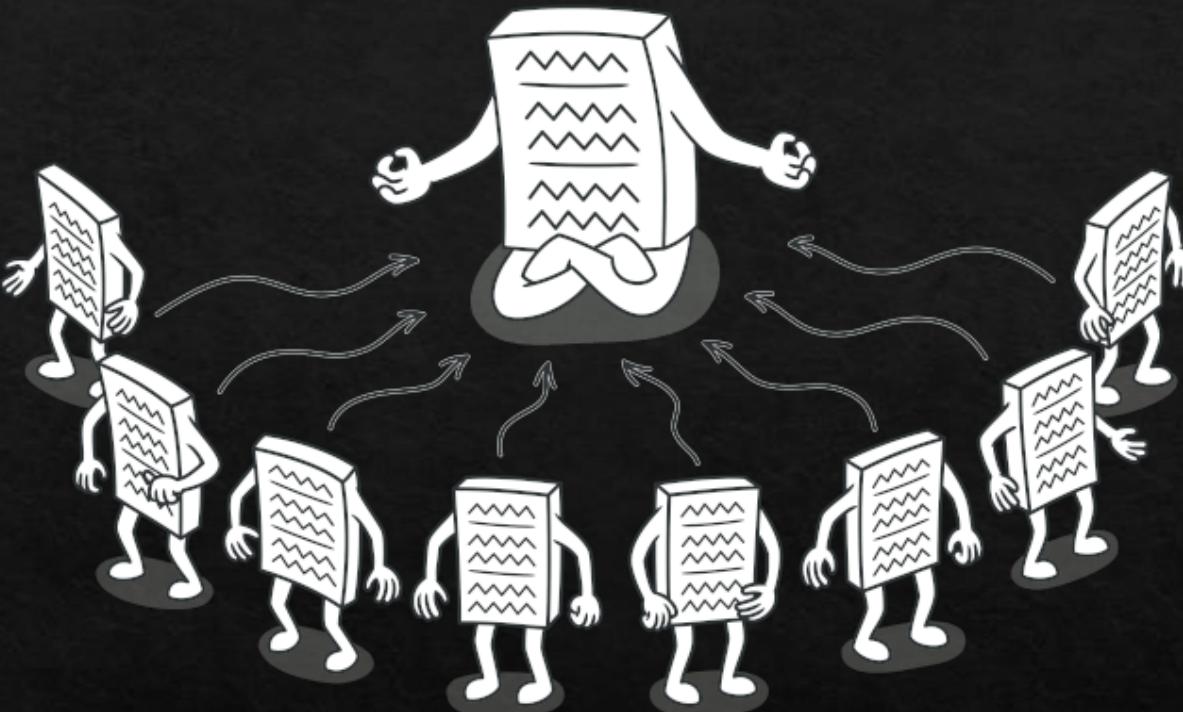
Zestaw 5 antywzorców na wzór SOLID

- **S**ingleton
- **T**tight coupling
- **U**ntestability
- **P**remature optimizations
- **I**ndescriptive naming

STUPID, czasami przedstawiany jest jako zestaw antywzorców przed, którym chroni stosowanie reguł SOLID.

Chociaż takie twierdzenie nie jest bezpodstawne, w ogólności jest nadużyciem.

S – Singleton



Stany o zasięgu globalnym przeważnie są trudne do przetestowania.

T- Tight coupling (silne sprzężenia)



Za dużo sprzężeń

U – Untestability



Testowanie nie powinn być trudne

P – Premature optimizations

(przedwczesne optymalizacje)



"Premature optimization is the root of all evil"

Donald Knuth

I – Indescriptive naming



Używaj precyzyjnych, samodefinujących nazw zmiennych/funkcji/klas. Na ile to możliwe startaj się unikac skrótowców.

D – Duplication



Don't Repeat Yourself!
Keep it simple, Stupid!

Be lazy the right way – write code only once

CUPID

Dan North (2017 r)

SOLID spotkał się ze sporą krytyką w ostatnich latach. Najczęściej podnoszonym zarzutem jest wiek tych zasad. 30 lat temu programiści mierzyli się z zupełnie innymi zagadnieniami niż dzisiaj.

Dan North zakwestionował zasady SOLID w 2017 roku poprzez zaproponowanie nowych zasad - CUPID

CUPID

Zestaw 5 wzorców stworzonych jako krytyka SOLID

- **Composable** – plays well with others
- **Unix philosophy** – does one thing well
- **Predictable** – does what you expect
- **Idiomatic** – feels natural
- **Domain-based**

Composable

Plays well with others

Composable code has classes and functions that are easily combined, assembled in different combinations, to address specific requirements.

Unix philosophy

Does one thing well

Unix is a longstanding operating system (or family of systems) that led to today's Linux. The Unix philosophy encourages coding components that work well together, each doing one thing and doing it well.

Predictable

Does what you expect

The goal here is code that is robust, reliable, and resilient. It should also behave as expected.

Idiomatic

Feels natural

With CUPID, the code should be understandable by someone else, not just you, the original coder.

Domain-based

The solution domain models the problem domain in language and structure

The code should be compatible with its domain, using the problem domain's language to avoid the need for future developers to 'translations' or cognitive leaps in order to understand it.

Wzorce projektowe



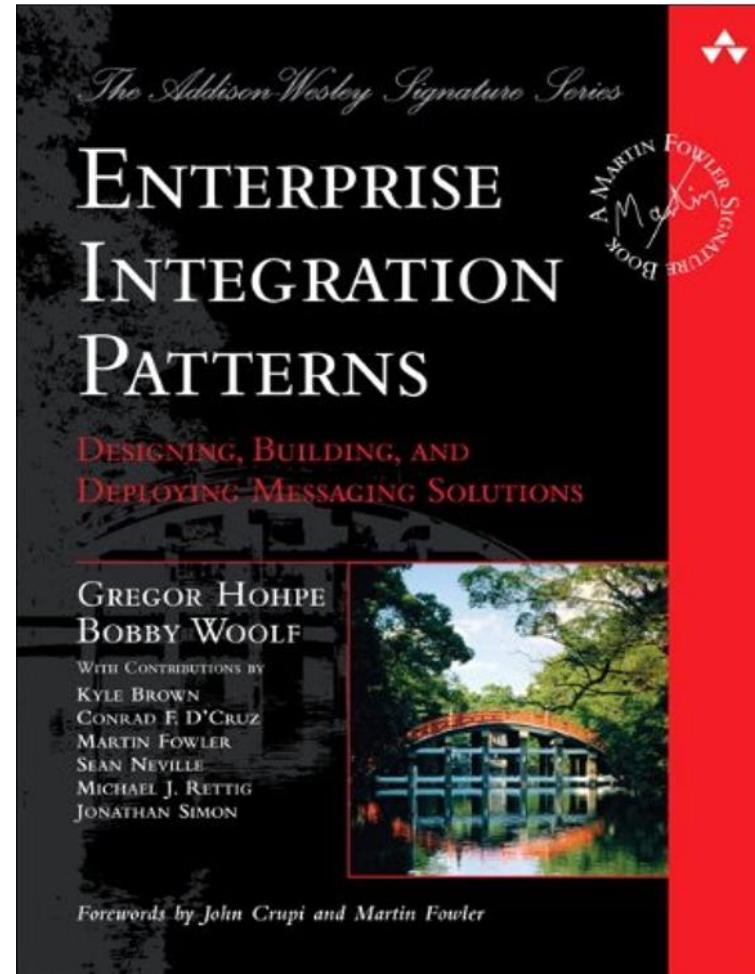
Wzorce projektowe

- **GoF (Gang of Four)**
Klasyczny zestaw zawierający 23 wzorce projektowe
- **EIP (Enterprise Integration Patterns)**
Zestaw 64 wzorców projektowych związanych z integracją systemów i integracją międzykomponentową
- **PoEAA (Patterns of Enterprise Application Architecture)**
Zestaw około 40 wzorców projektowych (liczba może się różnić pomiędzy wydaniami) opisanych przez Martina Flowera
- **Cloud Desing Patterns**
Wzorce projektowe dostosowane do tworzenia aplikacji w chmurze
- **Concurrency Patterns**
Wzorce projektowe dostosowane do tworzenia aplikacji współbieżnych
- **Security Patterns**
Wzorce projektowe związane z aspektami bezpieczeństwa
- **Game Programming Patterns**
Wzorce projektowe związane z programowaniem gier komputerowych
- **Test Design Patterns**
Wzorce projektowe związane z projektowaniem testów oprogramowania
- **J2EE Patterns**
Wzorce projektowe stosowane w ramach platformy Java 2 Enterprise Edition
- **JSP Patterns**
Zestaw wzorców projektowych związanych z językiem JavaScript

Gregor Hohpe, Bobby Woolf

Opublikowana w 2003

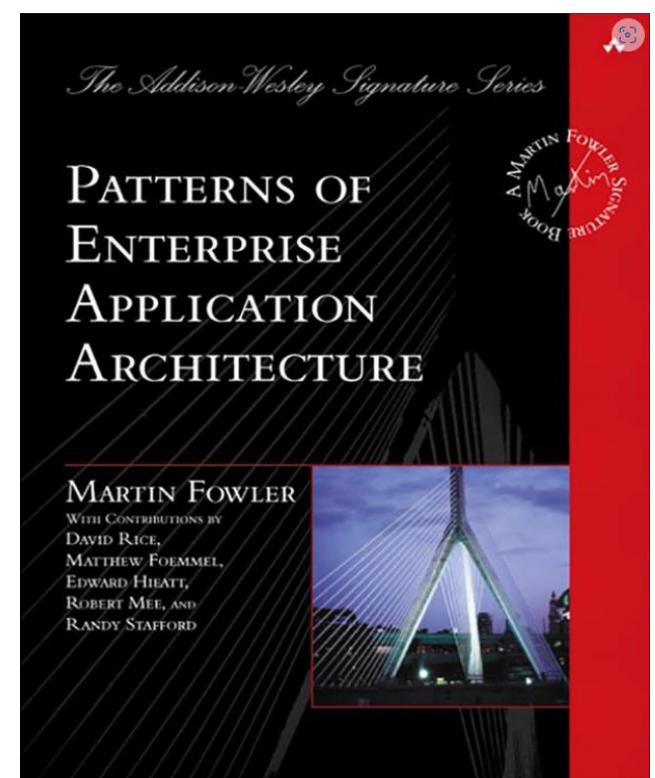
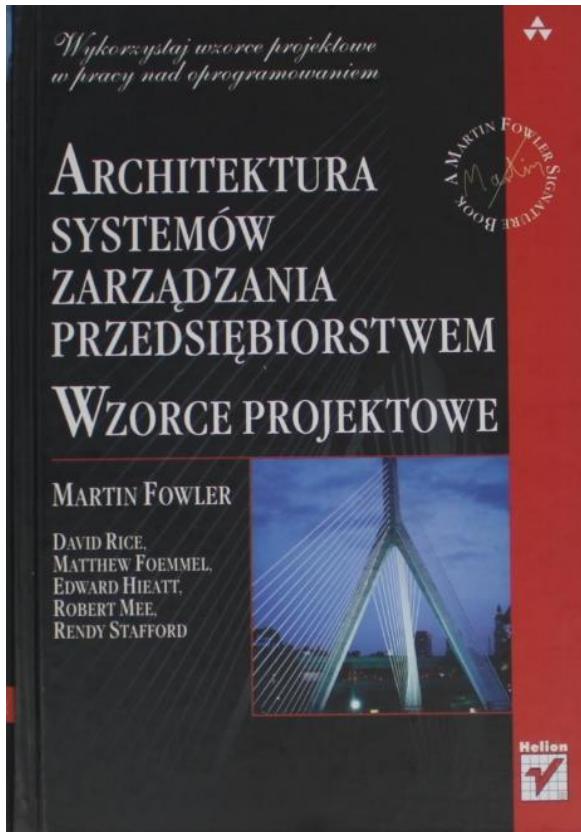
Zawiera zestaw 65 wzorców
pogrupowanych w 9 kategorii.



Martin Fowler

Opublikowana w 2003

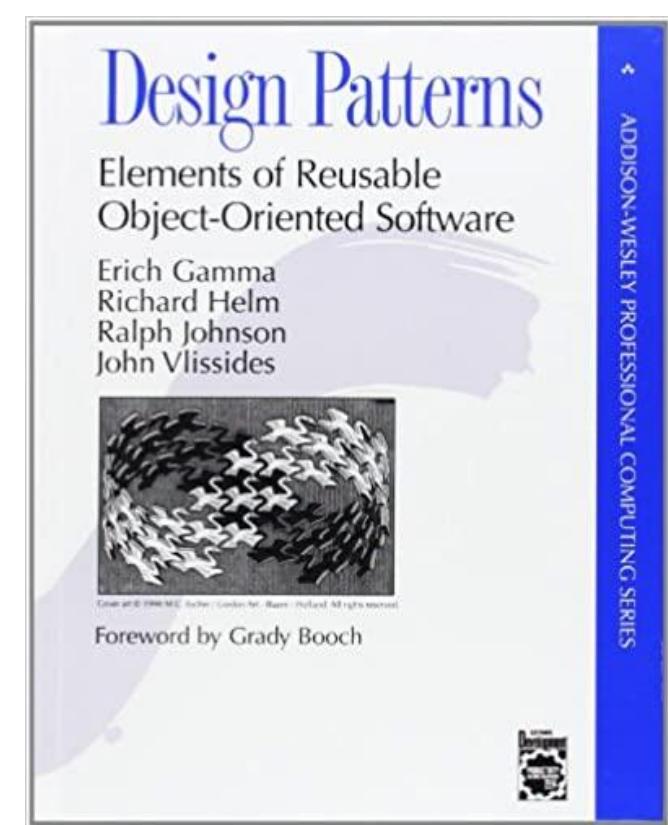
Stanowi ogólny przegląd dziedziny. W treści nawiązuje i analizuje wiele różnych wzorców bez prób ich kategoryzowania.



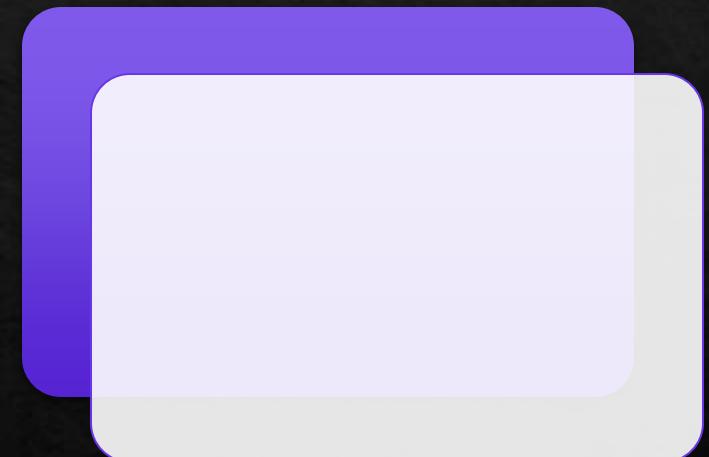
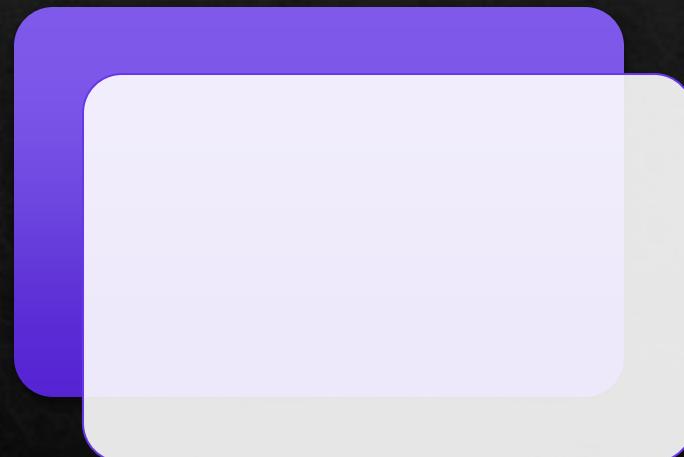
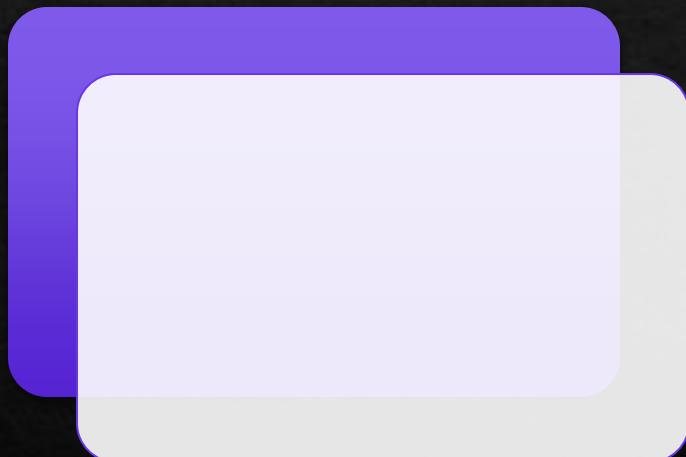
GAMMA, HELM, JOHNSON, VLISSIDES (GoF – Gang of Four)

Opublikowana w 1995

Pierwsza publikacja dotycząca wzorców projektowych zawierająca całościowe podejście do zagadnienia. Zawiera zestaw 23 wzorców pogrupowanych w 3 kategorie.



Podstawowa klasyfikacja wzorców projektowych (kategoryzacja Gamma)



Podstawowa klasyfikacja wzorców projektowych (kategoryzacja Gamma)

Kreacyjne

Tworzenie obiektów

Podstawowa klasyfikacja wzorców projektowych (kategoryzacja Gamma)

Kreacyjne

Tworzenie obiektów

Strukturalne

Kompozycja obiektów

Podstawowa klasyfikacja wzorców projektowych (kategoryzacja Gamma)

Kreacyjne

Tworzenie obiektów

Strukturalne

Kompozycja obiektów

Behawioralne (czynnościowe)

Interakcja
pomiędzy obiektyami,
odpowiedzialności

Alternatywna klasyfikacja wzorców projektowych



Alternatywna klasyfikacja wzorców projektowych

Klasowe

Opisujące statyczne
związki pomiędzy klasami

Alternatywna klasyfikacja wzorców projektowych

Klasowe

Opisujące statyczne
związki pomiędzy klasami

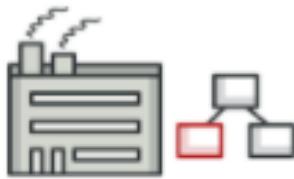
Obiektowe

Opisujące dynamiczne
związki pomiędzy obiektami

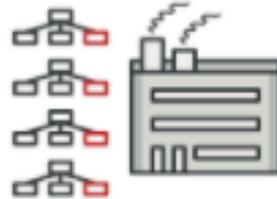
Wzorce konstrukcyjne	Wzorce strukturalne	Wzorce behawioralne
<ul style="list-style-type: none"> • Metoda wytwarzca • Fabryka abstrakcyjna • Budowniczy • Prototyp • Singleton 	<ul style="list-style-type: none"> • Adapter • Most • Kompozyt • Dekorator • Fasada • Pylek 	<ul style="list-style-type: none"> • Łańcuch zobowiązań • Polecenie • Iterator • Mediator • Pamiątka • Obserwator • Stan • Strategia • Metoda szablonowa • Odwiedzający

Wzorce kreacyjne

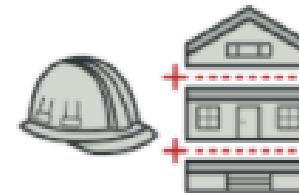
(Creational Patterns)



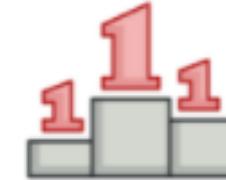
**Metoda
wytwórcza**
Factory Method



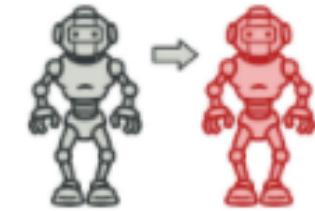
**Fabryka
abstrakcyjna**
Abstract Factory



Budowniczy
Builder



Singleton
Singleton

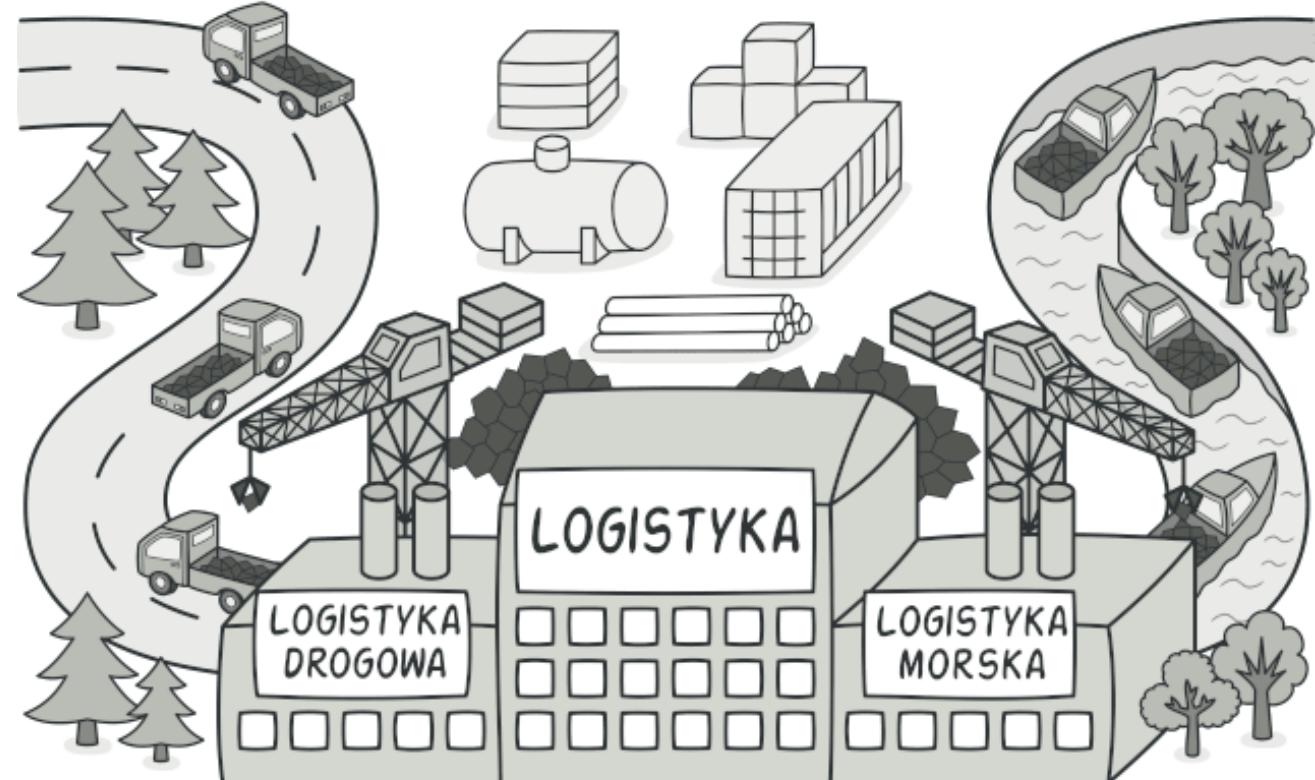


Prototyp
Prototype

Metoda wytwórcza (Factory method)

Kreacyjny wzorzec projektowy, który udostępnia interfejs do tworzenia obiektów w ramach klasy bazowej, ale pozwala podklasom zmieniać typ tworzonych obiektów.

Złożoność ★★★
Popularność ★★★★



Motywacja

Wyobraź sobie, że tworzysz aplikacje do zarządzania logistiką. Na razie aplikacja obsługuje tylko transport kołowy, a dokładniej ciężarówki. Aplikacja staje się popularna i pojawia się coraz więcej zgłoszeń od firm spedycyjnych, że przydałaby się również obsługa transportu wodnego.

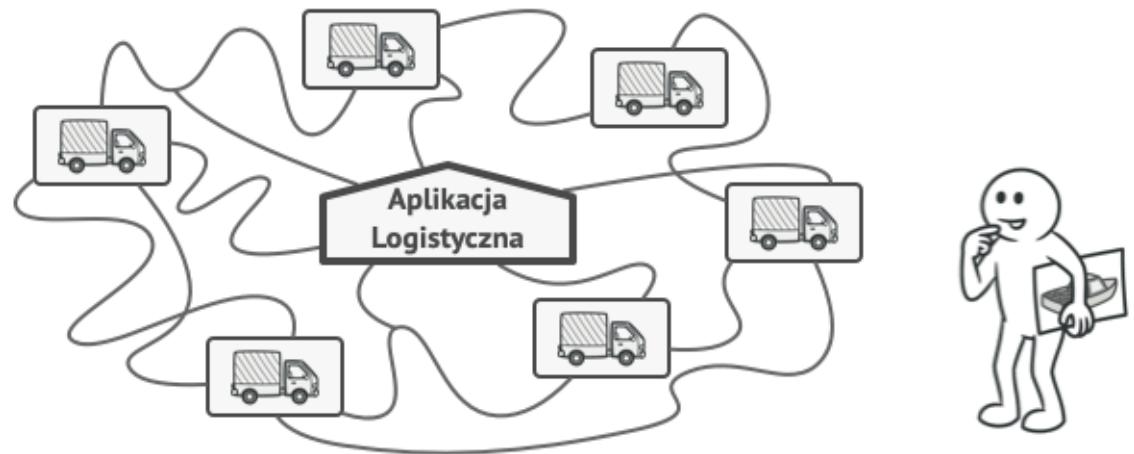


Dodanie nowej klasy do programu nie jest takie proste, jeśli reszta kodu jest już silnie powiązana z istniejącą klasą/klasami.



Rozwiązanie

Wyobraź sobie, że tworzysz aplikacje do zarządzania logistiką. Na razie aplikacja obsługuje tylko transport kołowy, a dokładniej ciężarówki. Aplikacja staje się popularna i pojawia się coraz więcej zgłoszeń od firm spedycyjnych, że przydałaby się również obsługa transportu wodnego.



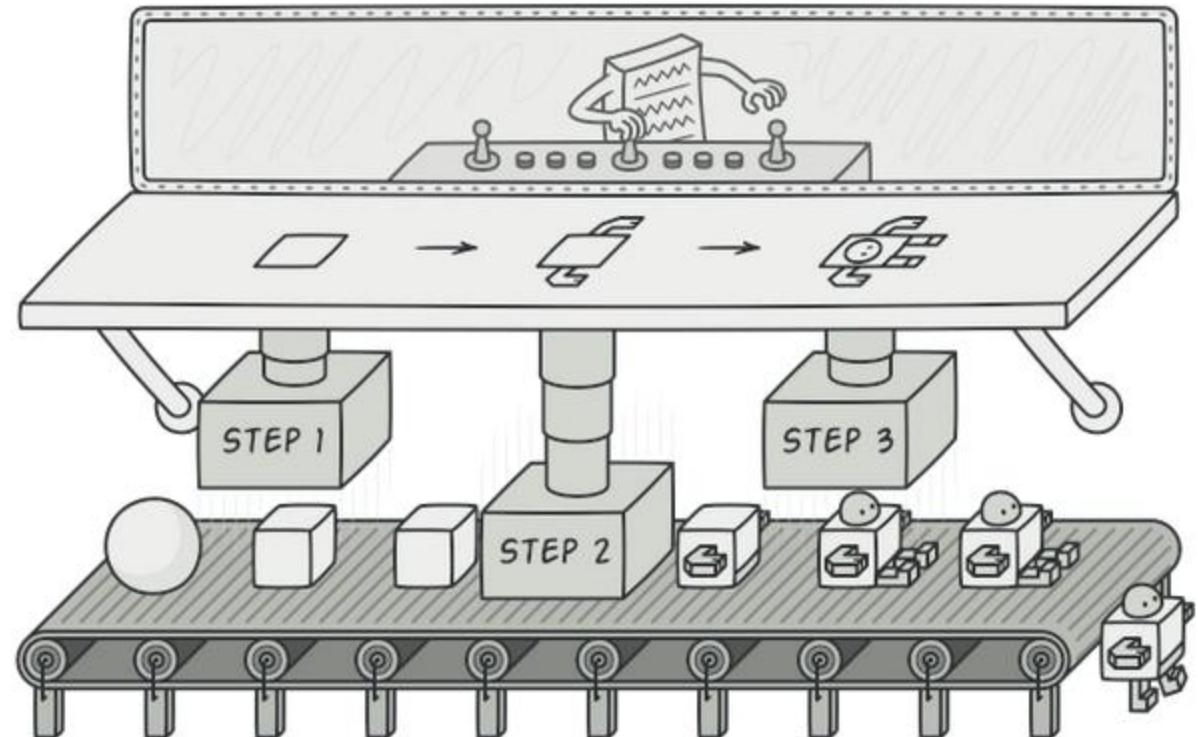
Dodanie nowej klasy do programu nie jest takie proste, jeśli reszta kodu jest już silnie powiązana z istniejącą klasą/klasami.



Fabryka abstrakcyjna (Builder)

Kreacyjny wzorzec projektowy, którego celem jest rozdzielenie sposobu tworzenia obiektu od jego reprezentacji.

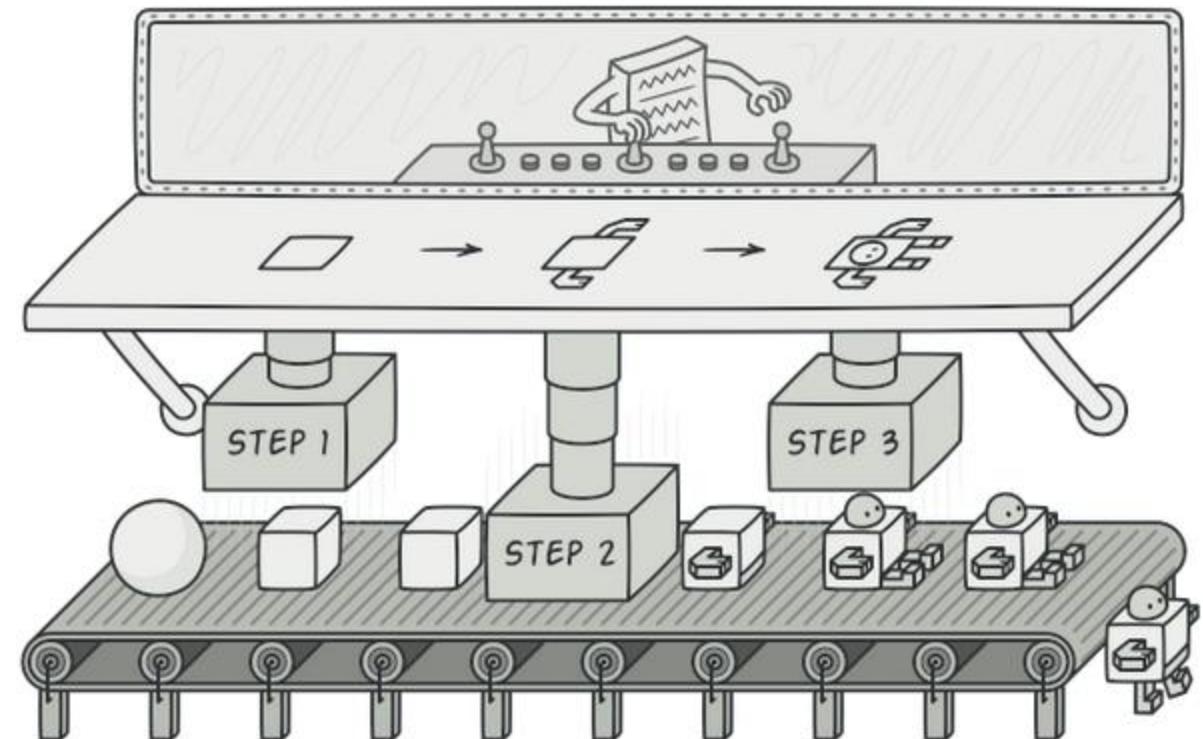
Złożoność ★★★
Popularność ★★★★



Złożoność ★★★
Popularność ★★★

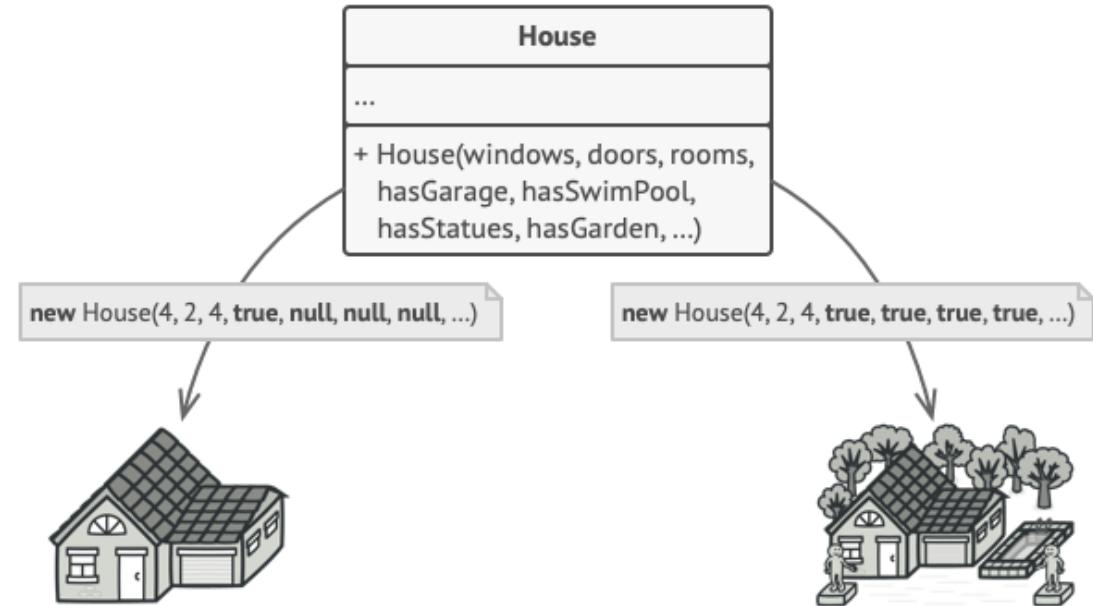
Budowniczy (Builder)

Kreacyjny wzorzec projektowy, którego celem jest rozdzielenie sposobu tworzenia obiektu od jego reprezentacji.



Motywacja

Wyobraź sobie jakiś skomplikowany obiekt (np. dom), którego inicjalizacja jest pracochłonnym, wieloetapowym procesem obejmującym wiele pól i obiektów zagnieżdżonych.

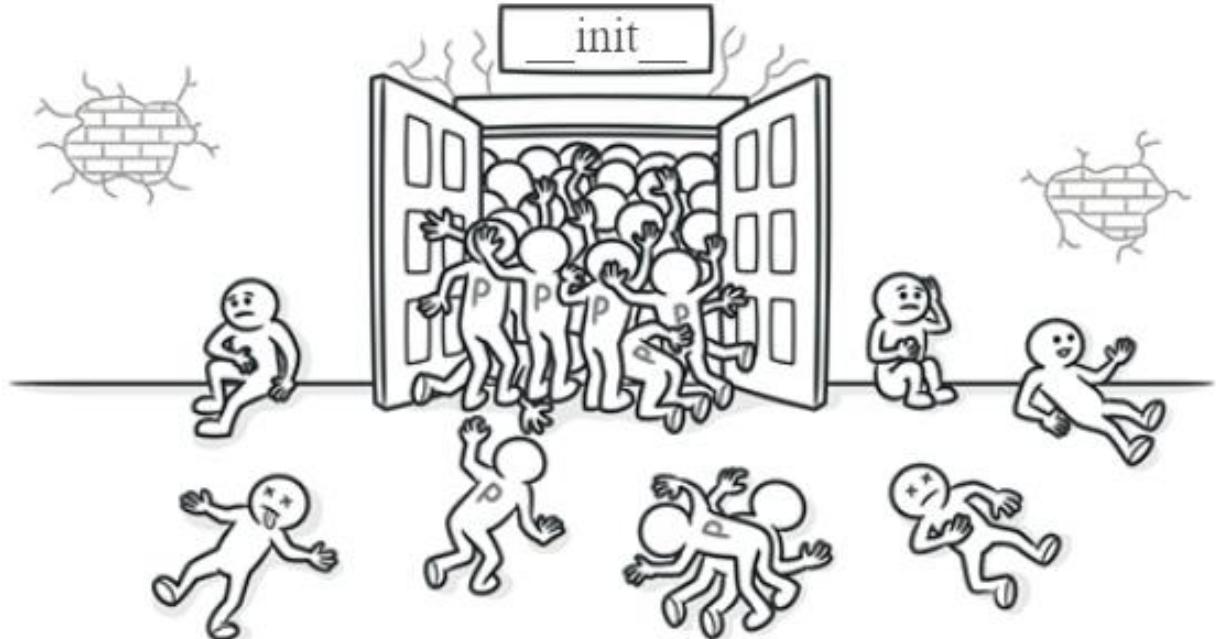


Konstruktor przyjmujący mnóstwo parametrów ma swoją wadę: nie wszystkie parametry będą potrzebne za każdym razem.



Trudności

Taki kod inicjalizacyjny jest często wrzucany do wielgachnego konstruktora, przyjmującego mnóstwo parametrów. Albo jeszcze gorzej, kod taki bywa rozrzucony po całym kodzie klienckim.

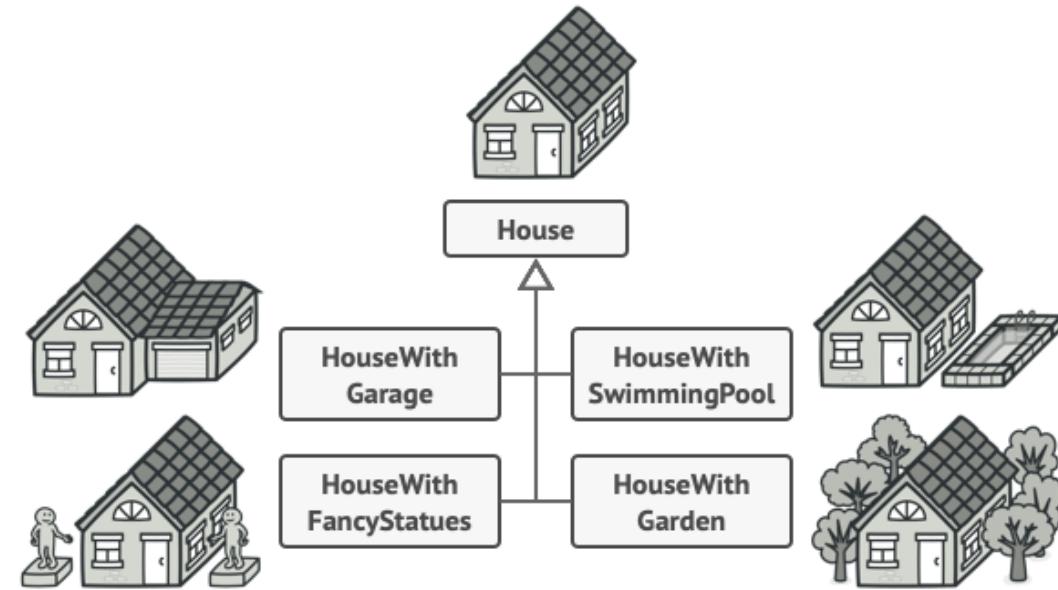


Konstruktor przyjmujący mnóstwo parametrów ma swoją wadę: nie wszystkie parametry będą potrzebne za każdym razem.



Próba uporządkowania

Możemy rozszerzyć klasę bazową Dom i stworzyć system 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.



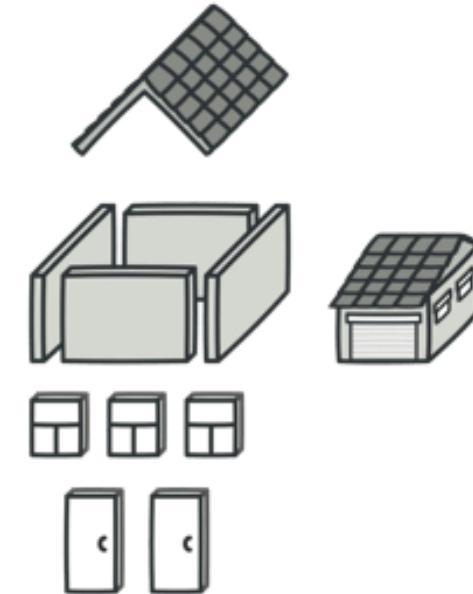
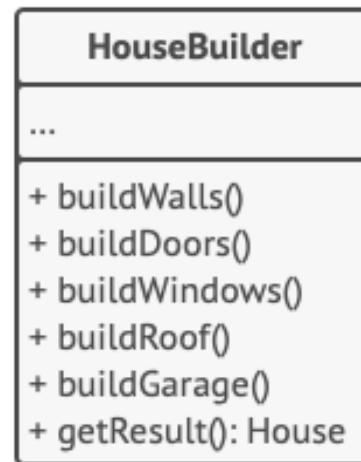
Program może stać się nadmiernie skomplikowany, jeśli każda możliwa konfiguracja oznacza dodanie nowej podklasy.



Rozwiązanie

Wzorzec projektowy Budowniczy proponuje wydzielenie kodu konstrukcyjnego obiektu z jego klasy i umieszczenie go w osobnym obiekcie zwany *budowniczym*.

Następnie wydzielony kod konstrukcyjny można podzielić na etapy (budujŚciany, wstawDrzwi, itp) i wywoływać tylko te etapy, które są niezbędne dla określonej konfiguracji obiektu. W ten sposób unikamy dużej liczby podklas.



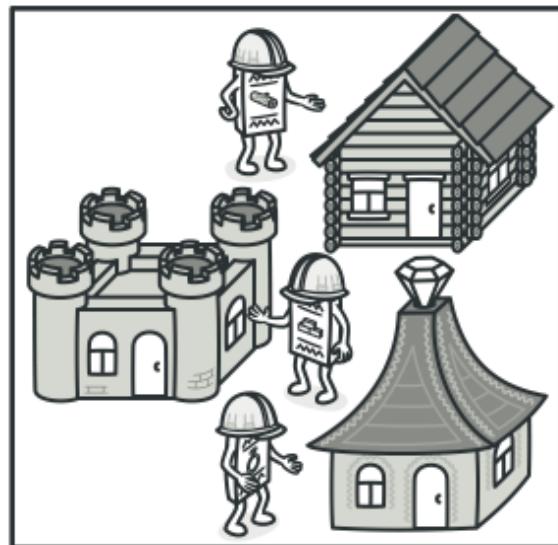
Wzorzec Budowniczy pozwala konstruować złożone obiekty krok po kroku



Interfejs budowniczego

Niektóre etapy konstrukcji mogą wymagać odmiennych implementacji 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. Część etapów można wynieść do wspólnego interfejsu.



Różni budowniczowie wykonują to samo zadanie w różny sposób



Kierownik (director)

Poszczególne etapy konstrukcji można wywoływać w odpowiedniej kolejności z poziomu kodu klienckiego. Można też przenieść te wywołania do osobnej klasy, zwanej *kierownikiem*.

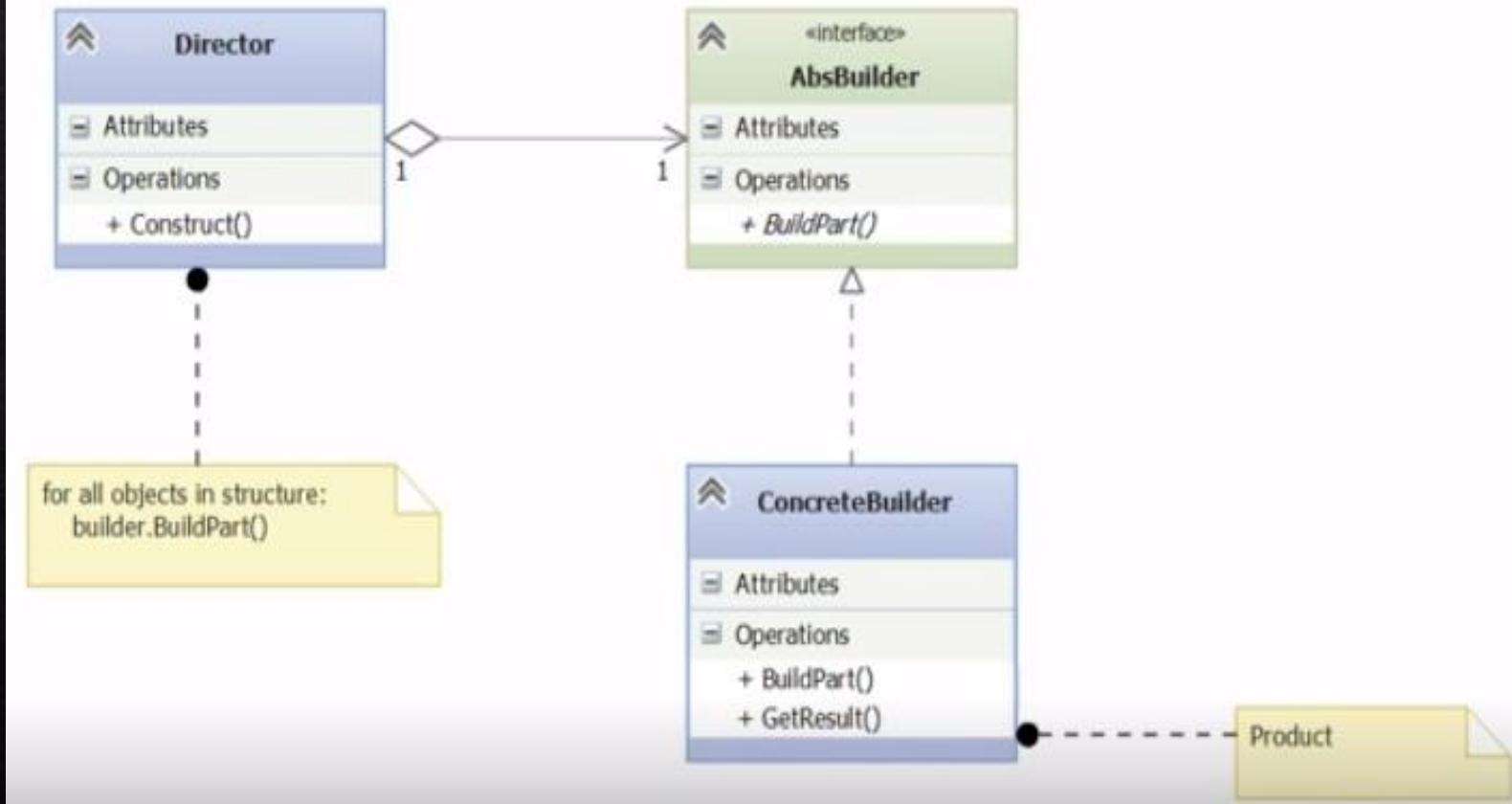
Kierownik określa kolejność etapów jaką musi zachować budowniczy. Wtedy kod kliencki musi tylko skojarzyć budowniczego z kierownikiem, a następnie odebrać wynik pracy od kierownika.



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

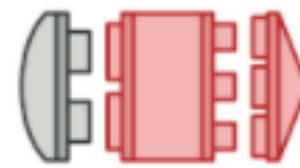


Struktura wzorca Budowniczy (Builder)



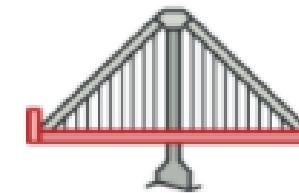
Wzorce strukturalne

(Structural Patterns)



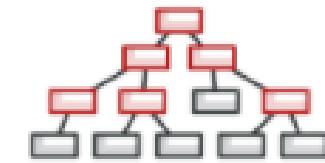
Adapter

Adapter



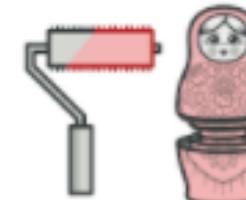
Most

Bridge



Kompozyt

Composite



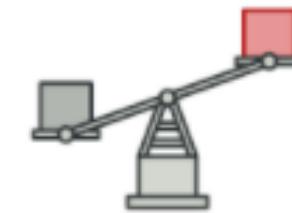
Dekorator

Decorator



Fasada

Facade



Pyłek

Flyweight

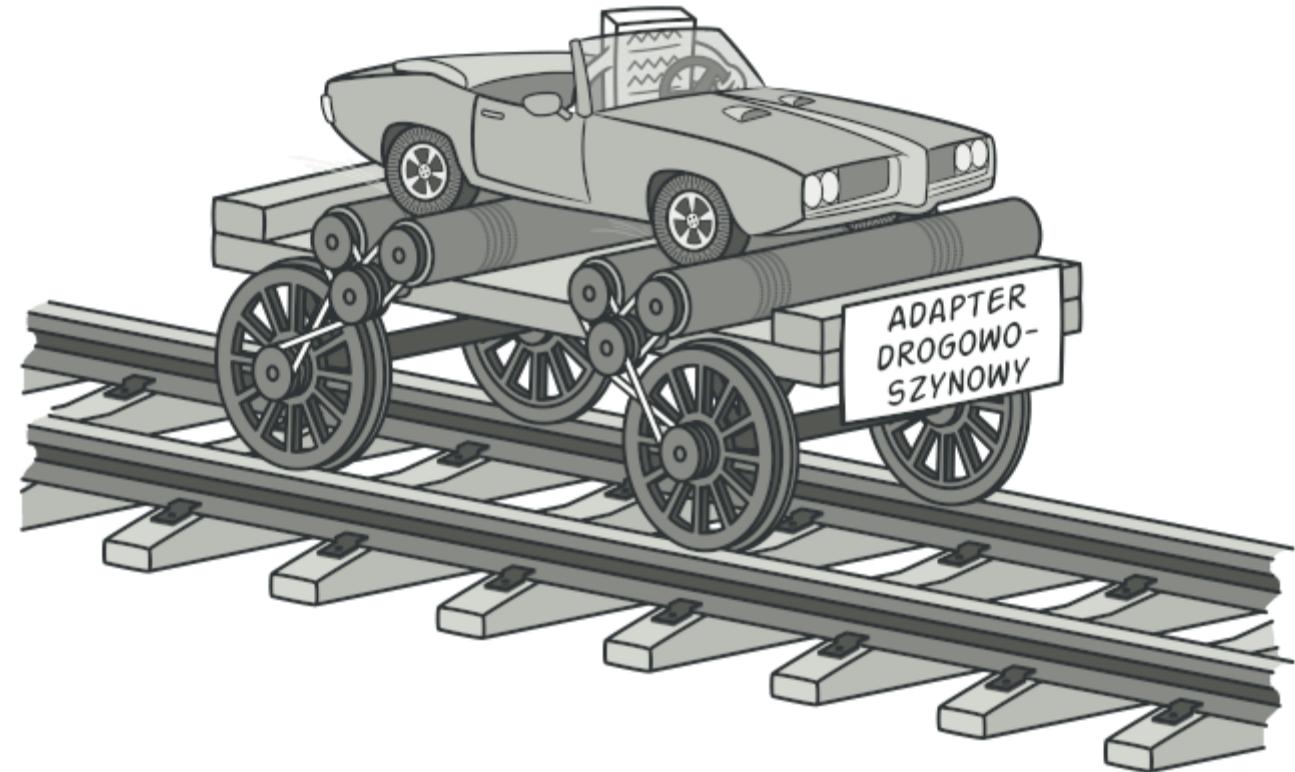
Adapter

Strukturalny wzorzec projektowy pozwalający na współdziałanie ze sobą obiektów o niekompatybilnych interfejsach.

Złożoność



Popularność



Adapter

Adapters in Real Life



Wall wart

Adapter

Adapters in Real Life



Wall wart



Pipe adapter

Adapter

Adapters in Real Life



Wall wart

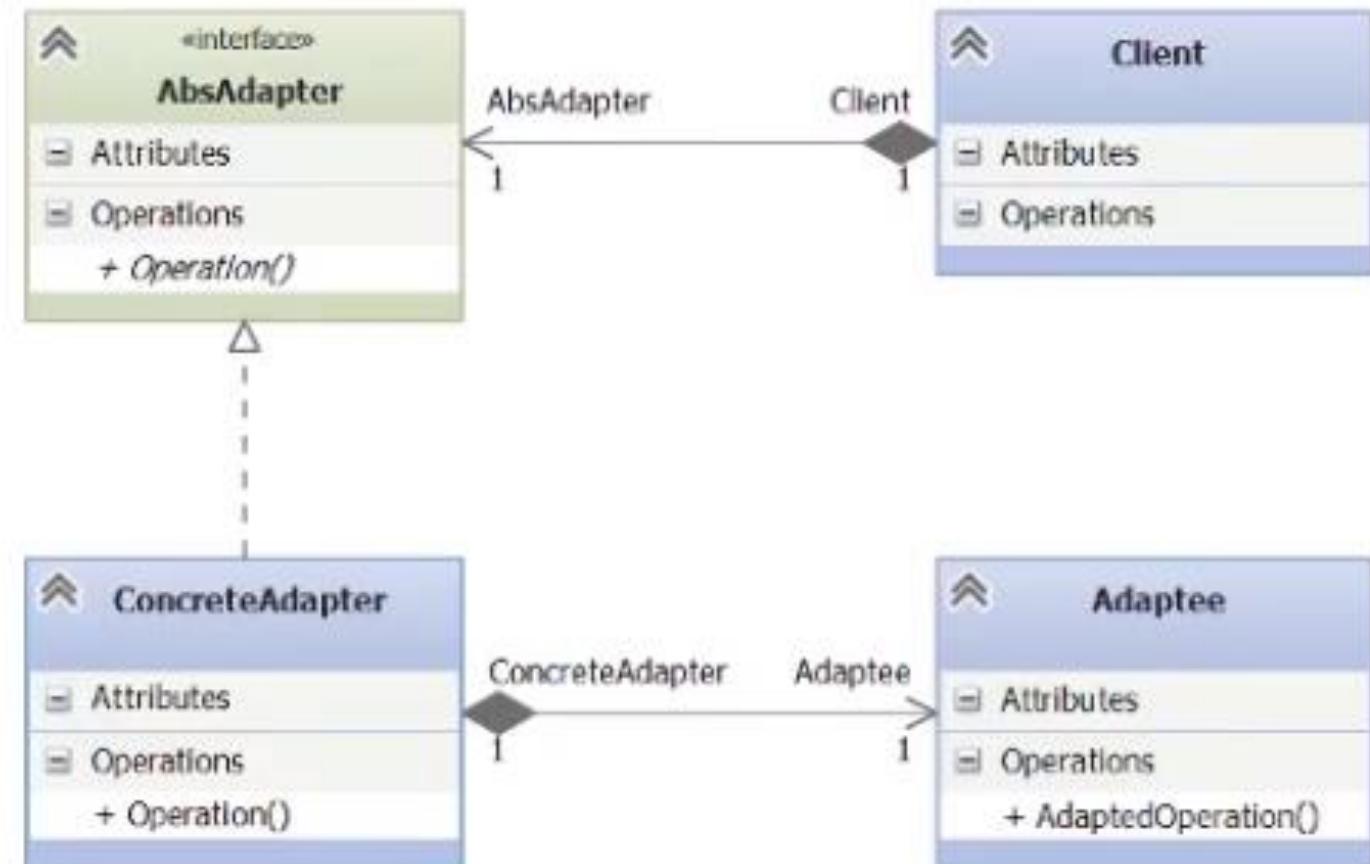


Pipe adapter



Don't try this at home!

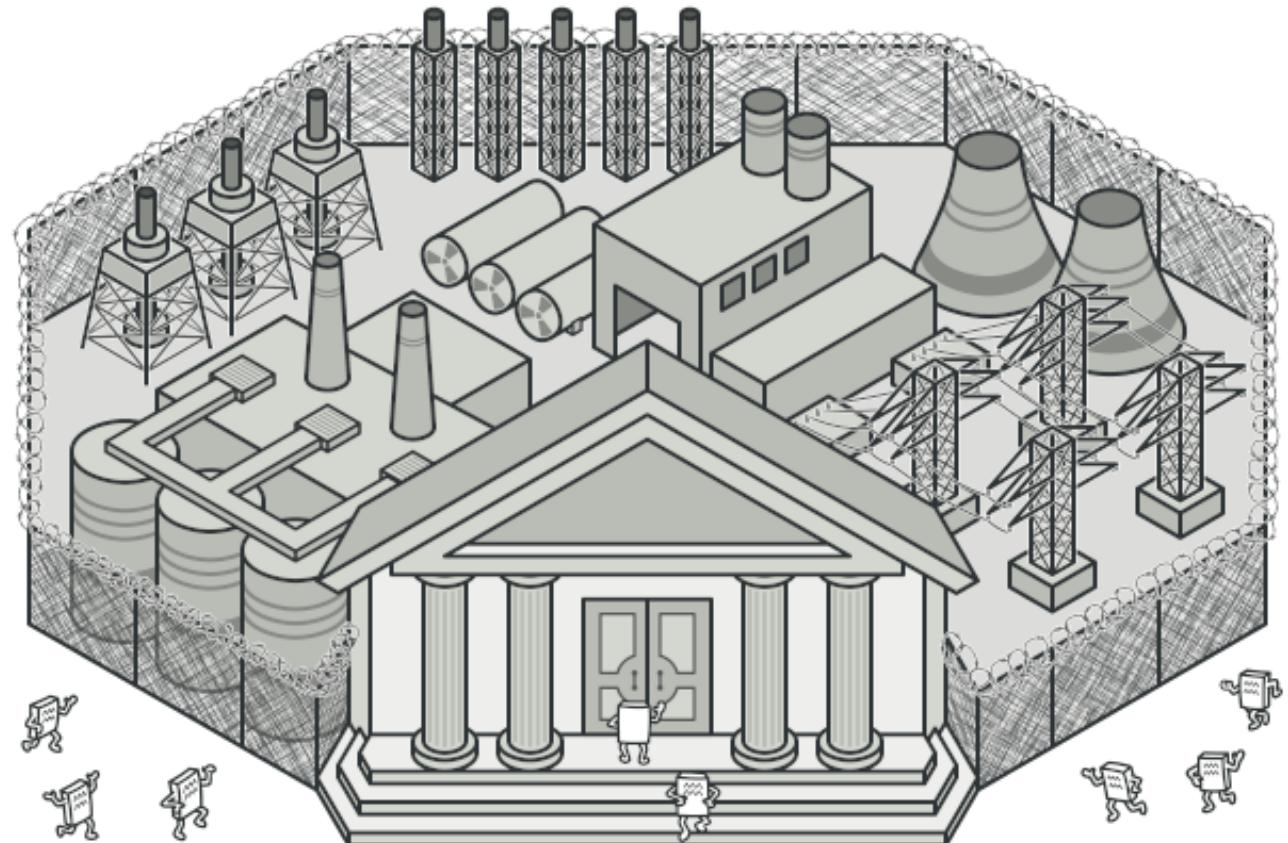
Struktura wzorca adapter (Adapter aka Wrapper)



Fasada (Facade)

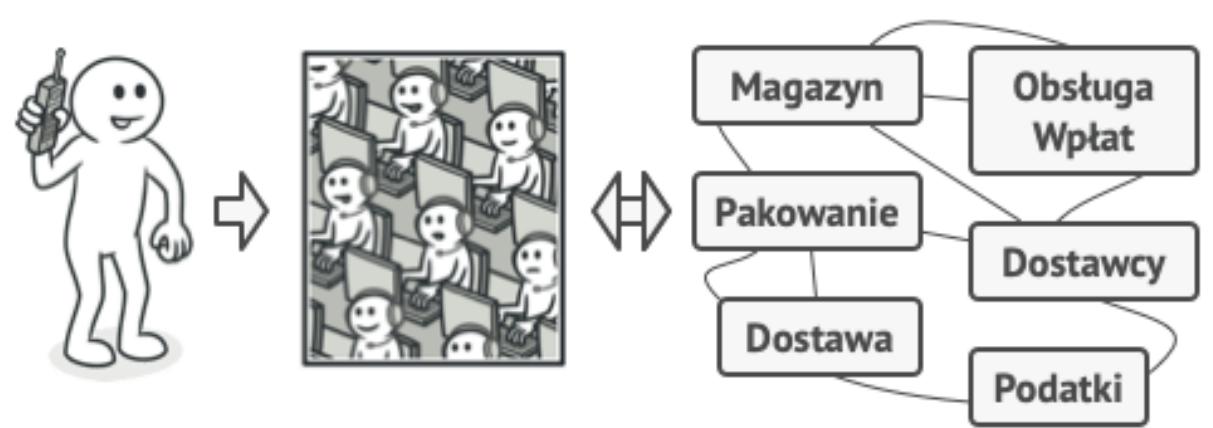
Strukturalny wzorzec projektowy,
który wyposaża bibliotekę,
framework lub inny złożony
zestaw klas w uproszczony interfejs

Złożoność ★ ★ ★
Popularność ★ ★ ★



Analogia z życia

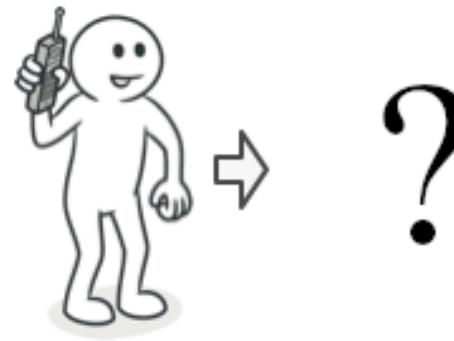
Gdy dzwonisz do sklepu aby złożyć zamówienie, biuro jest twoją fasadą dla wszystkich usług i oddziałów tego sklepu. Pracownik sklepu, czy automat zgłoszeniowy, stanowią prosty interfejs głosowy do systemu zamawiania, płacenia i różnych usług dostawczych.



Składanie zamówienia przez telefon



A jak by to wyglądało bez fasady ?



Składanie zamówienia przez telefon



A jak by to wyglądało bez fasady ?



Składanie zamówienia przez telefon



Motywacja

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

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



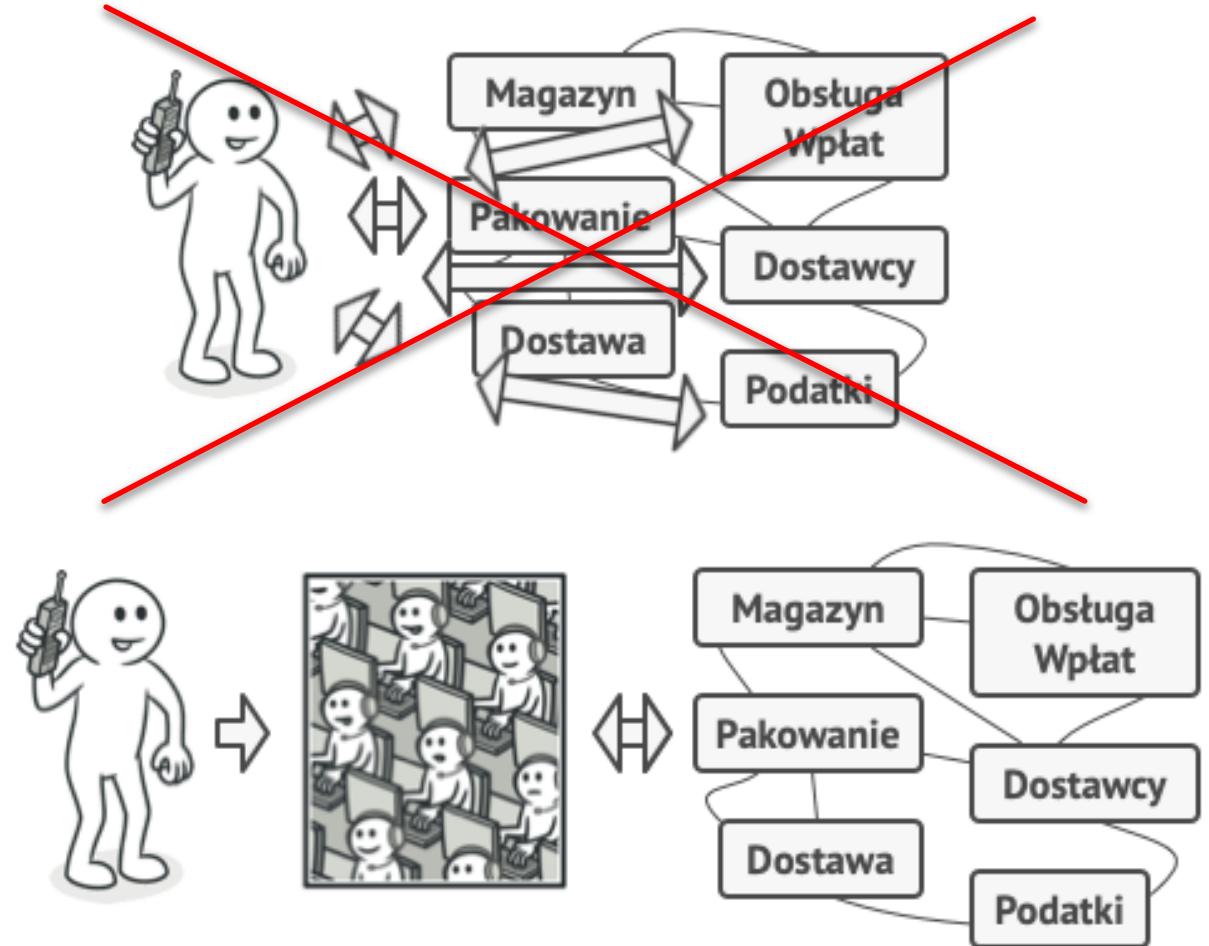
Składanie zamówienia przez telefon



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.

Klient korzysta z fasady zamiast wywoływać obiekty systemu bezpośrednio.



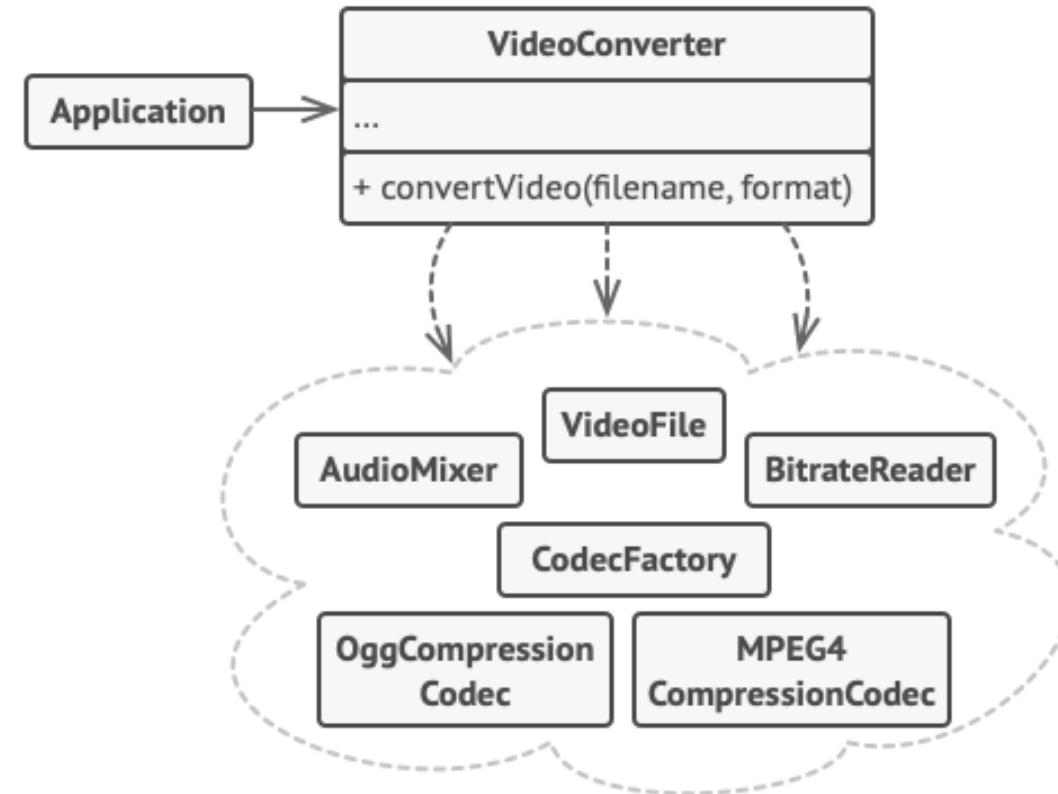
Składanie zamówienia przez telefon



Przykład

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

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

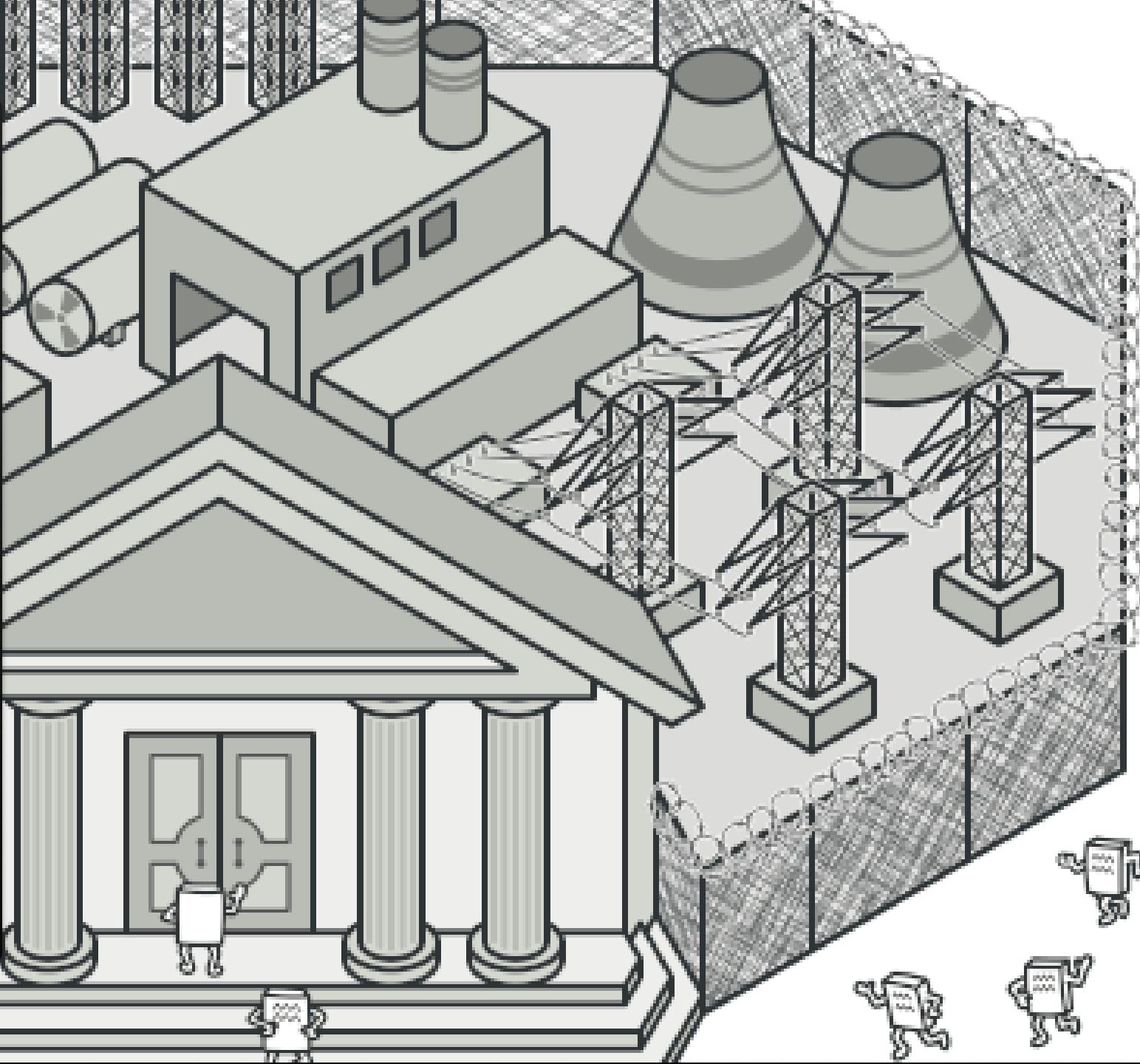


Przykład izolowania wielu zależności w pojedynczej klasie fasady



Zastosowanie

- Użyj wzorca Fasada, gdy potrzebujesz ograniczonego, ale łatwego w użyciu interfejsu do złożonego podsystemu
- Stosuj Fasadę, gdy chcesz ustrukturyzować podsystem w warstwy.



Kiedy użyć

- Użyj wzorca Fasada, gdy potrzebujesz ograniczonego, ale łatwego w użyciu interfejsu do złożonego podsystemu
- Stosuj Fasadę, gdy chcesz ustrukturyzować podsystem w warstwy.

Jak nie używać

- Uważaj, żeby Fasada nie stała się boskim obiektem, sprężonym ze wszystkimi klasami aplikacji.



Wzorce behawioralne

(Behavioral Patterns)



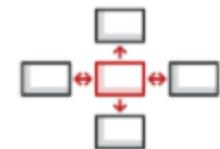
**Łańcuch
zobowiązań**
Chain of
Responsibility



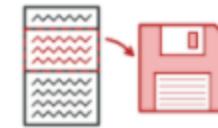
Polecenie
Command



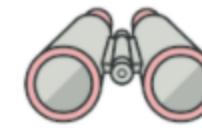
Iterator
Iterator



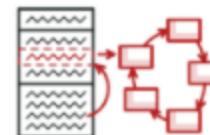
Mediator
Mediator



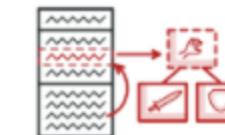
Pamiątka
Memento



Obserwator
Observer



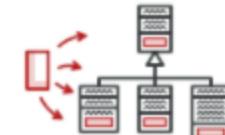
Stan
State



Strategia
Strategy



**Metoda
szablonowa**
Template Method

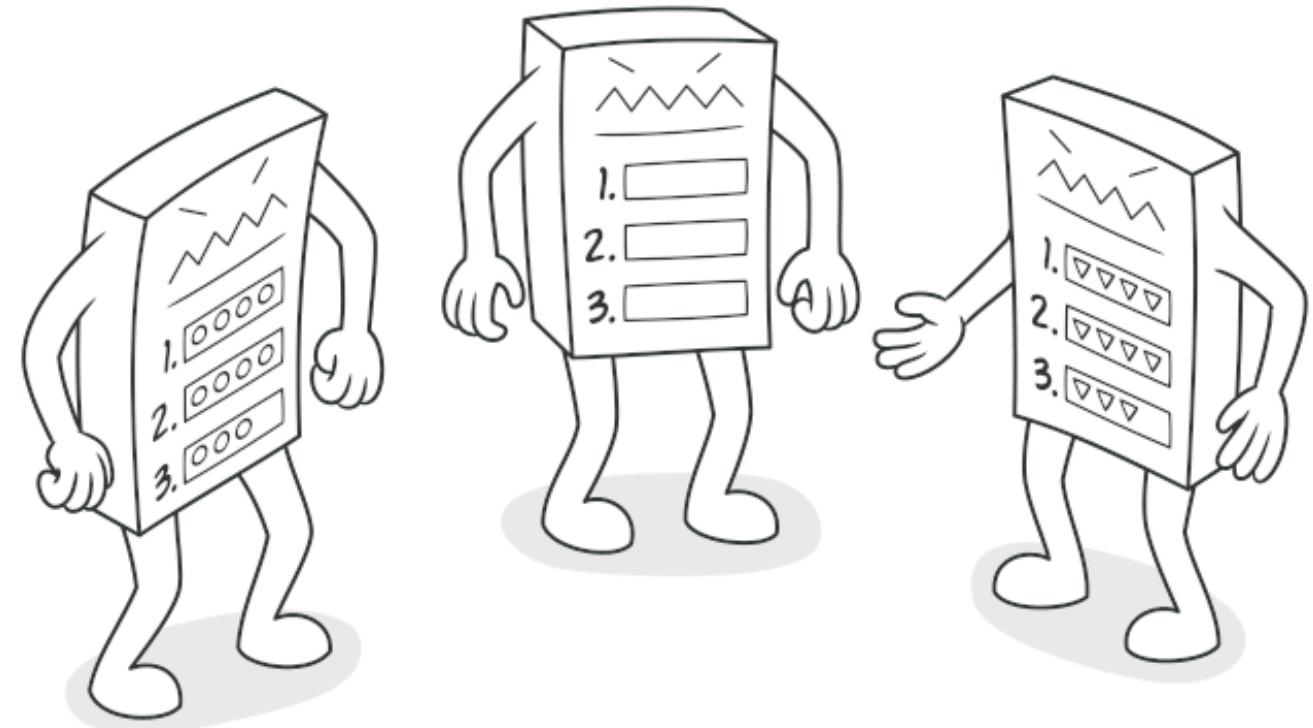


Odwiedzający
Visitor

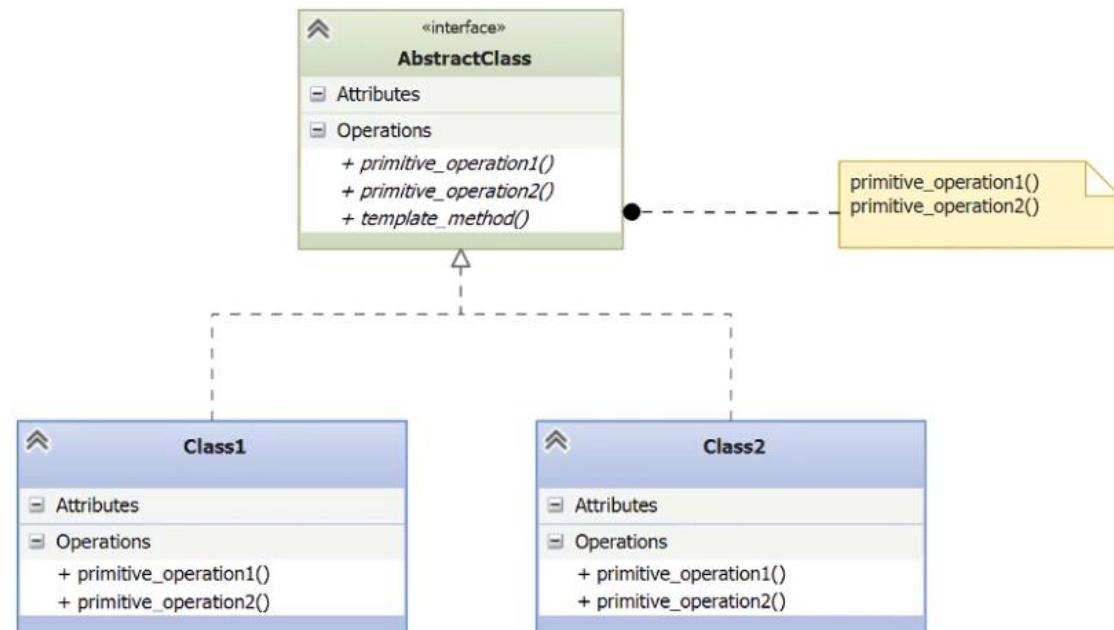
Metoda szablonowa (Template)

Czynnościowy wzorzec projektowy definiujący szkielet algorytmu w klasie bazowej, ale pozwalający podklasom nadpisać pewne etapy tego algorytmu bez konieczności zmiany jego struktury.

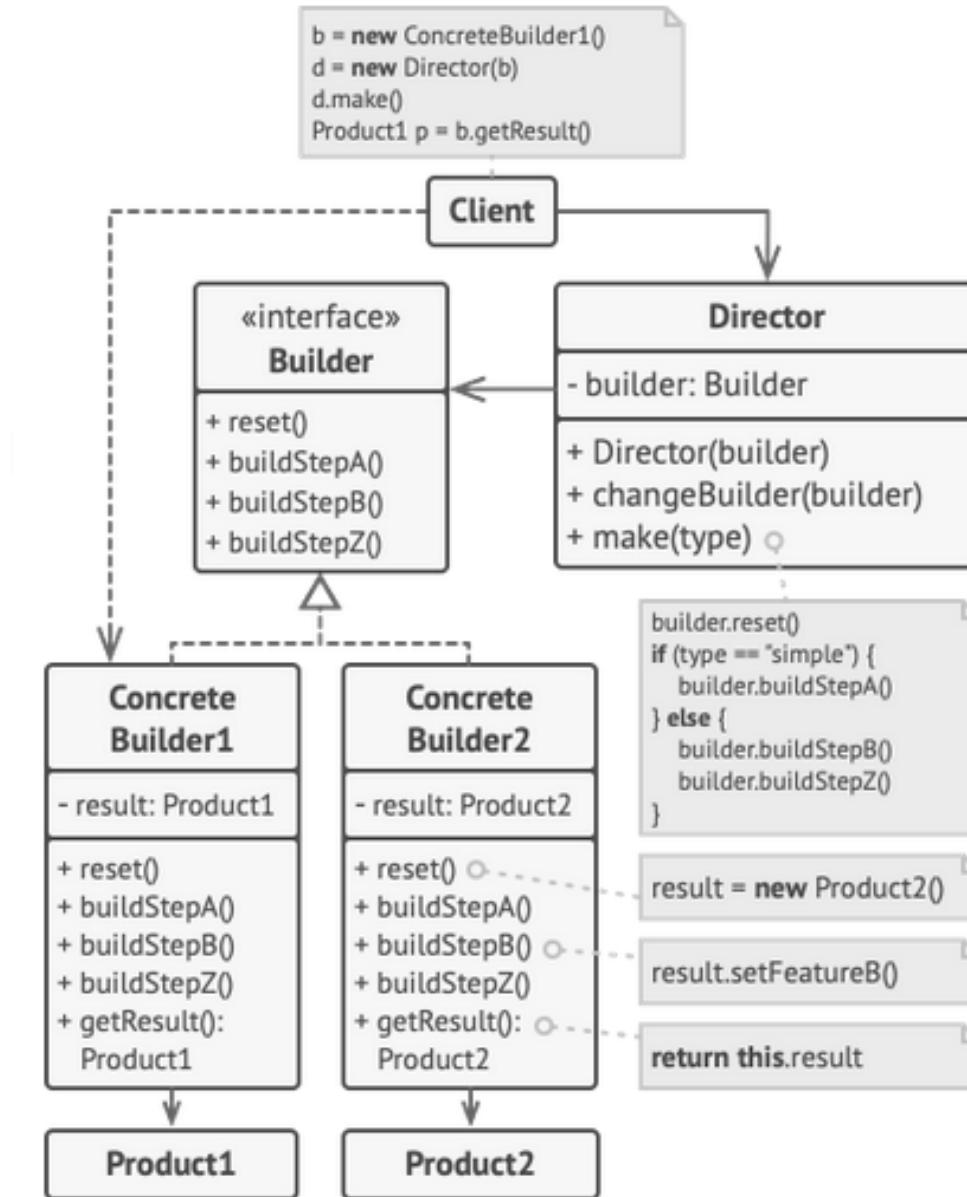
Złożoność ★ ★ ★
Popularność ★ ★ ★



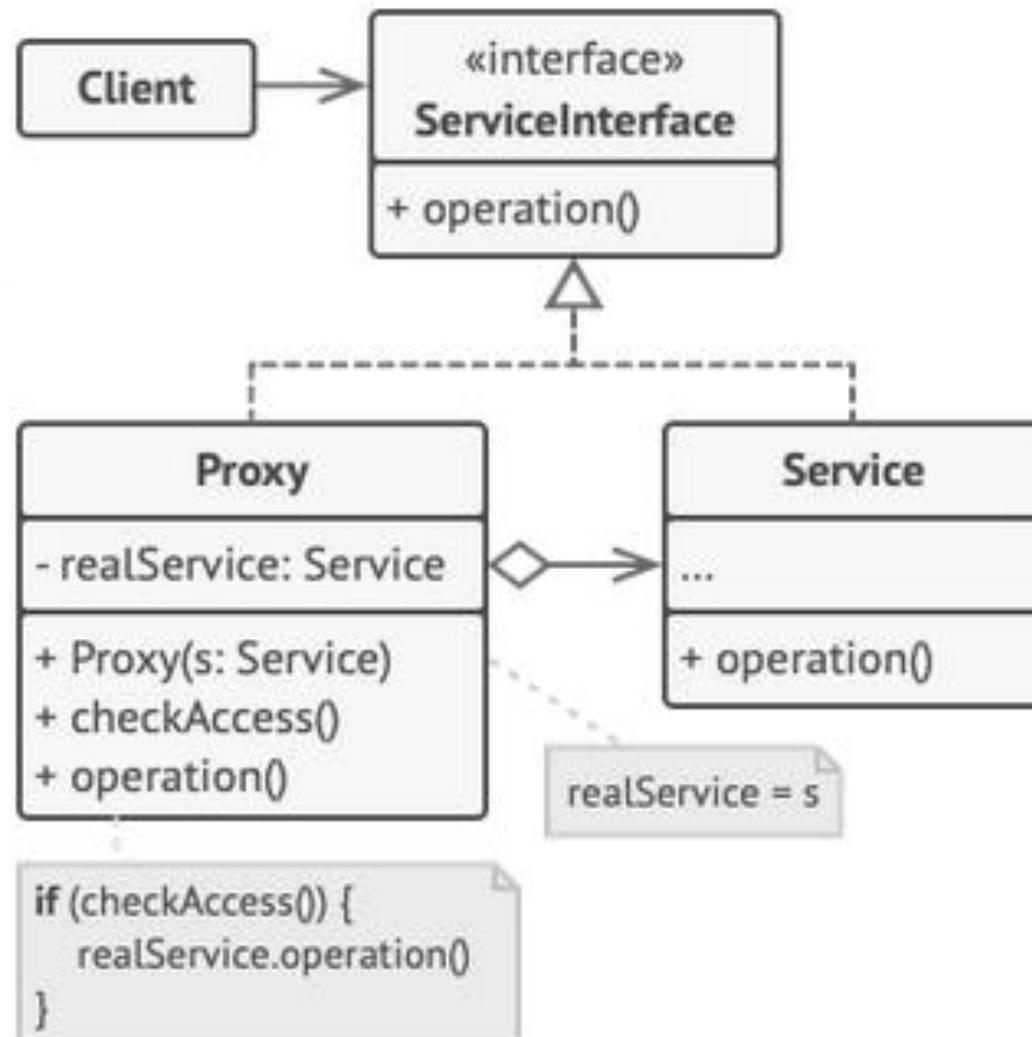
Struktura wzorca metoda szablonowa (Template)



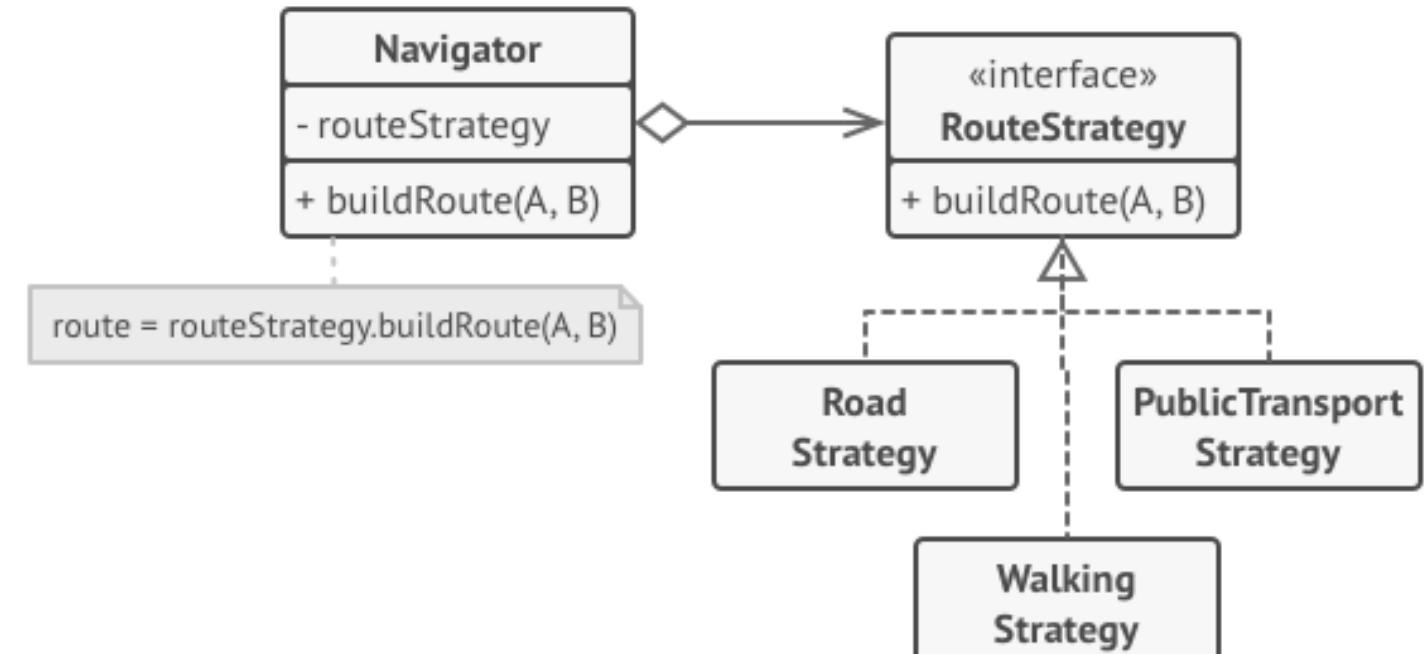
Budowniczy



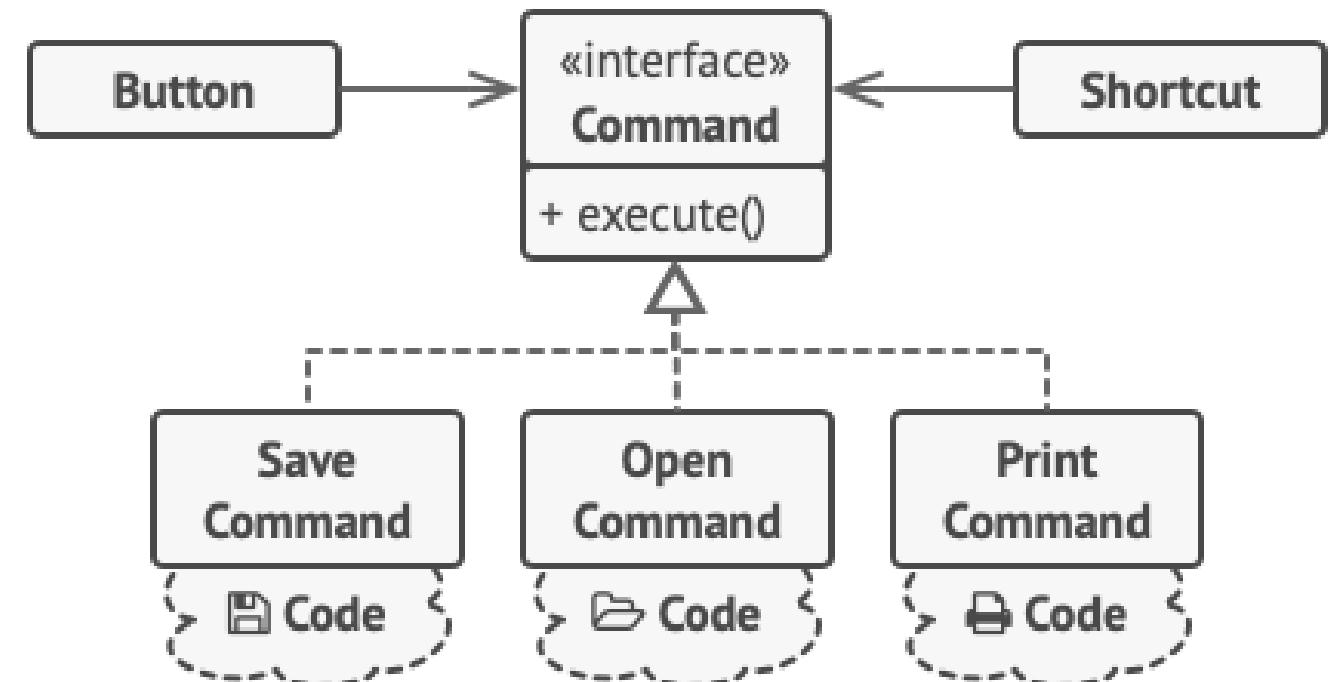
Proxy



Strategia

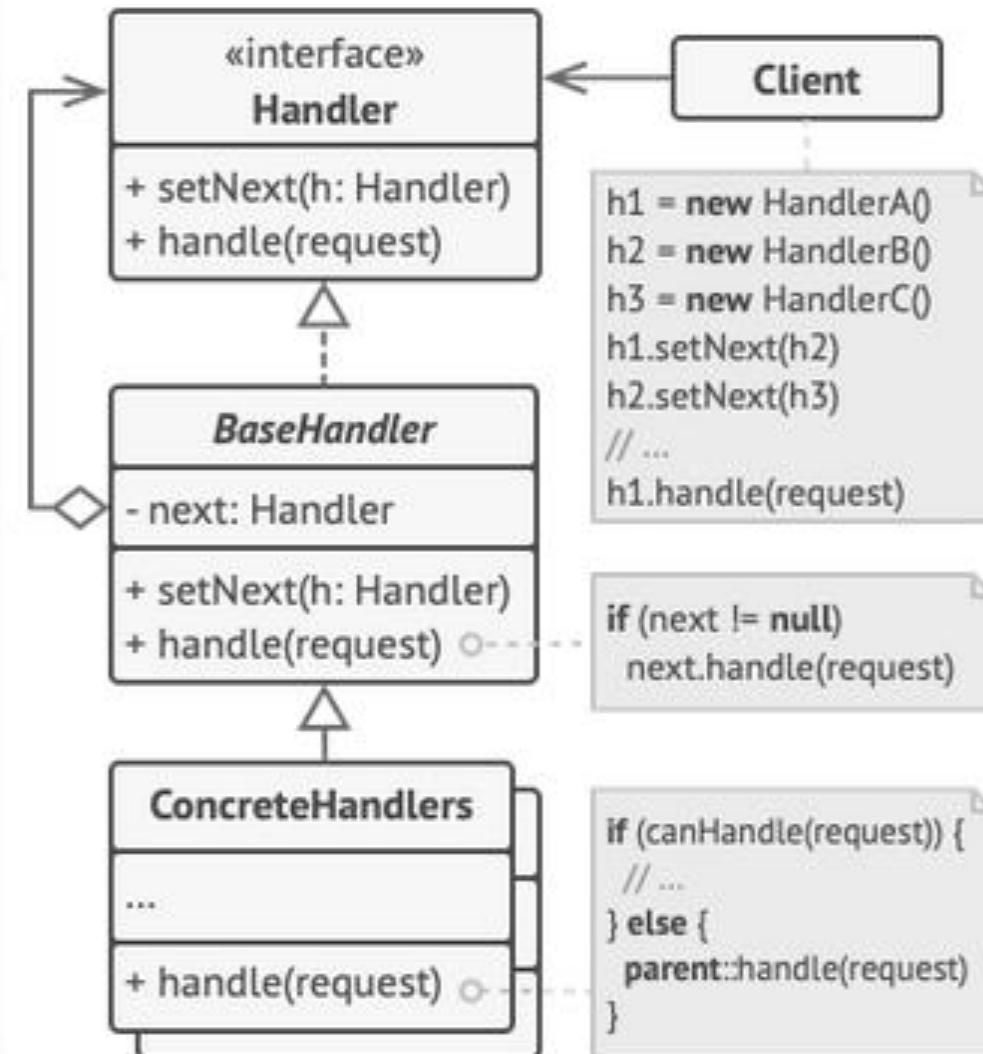


Komenda

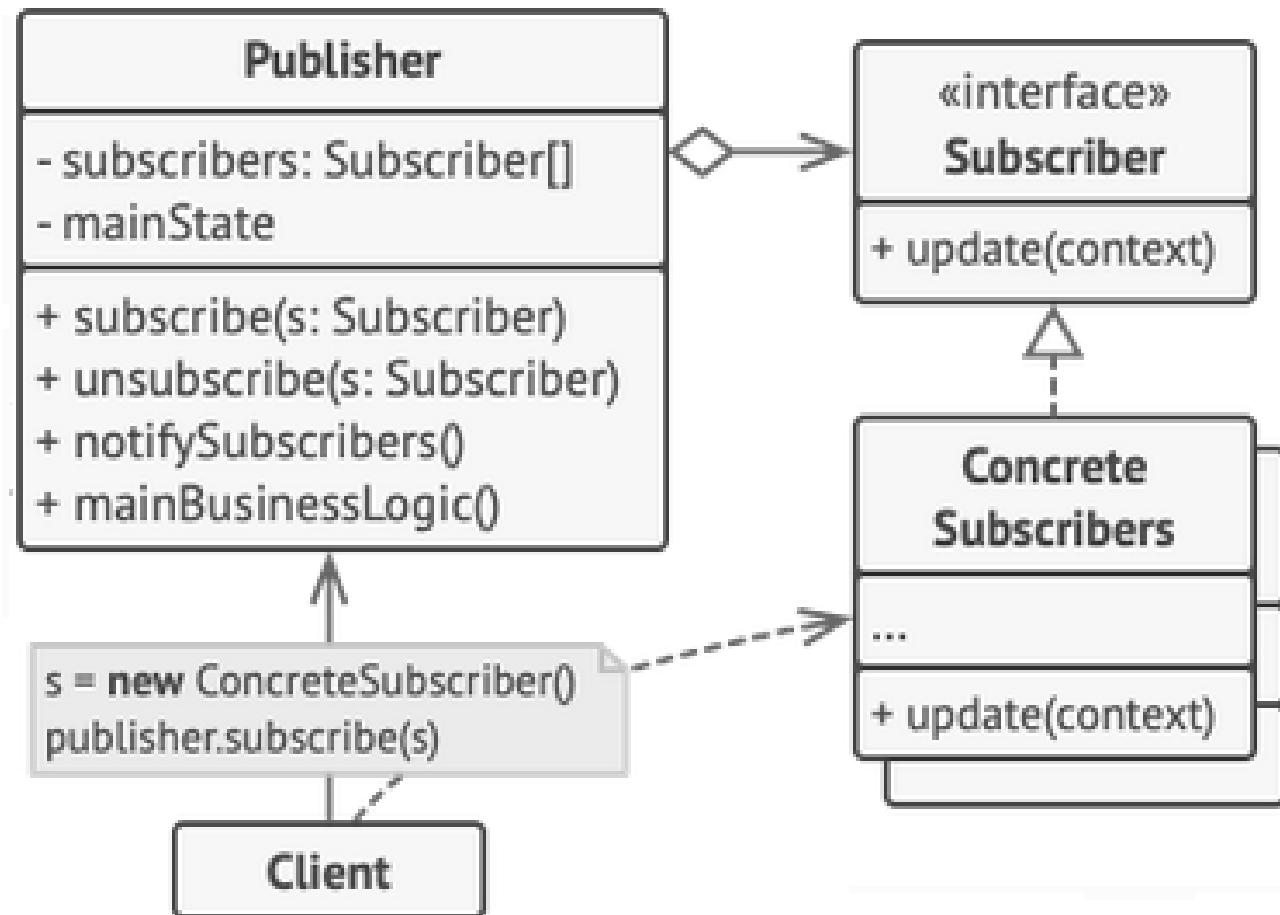


The GUI objects delegate the work to commands.

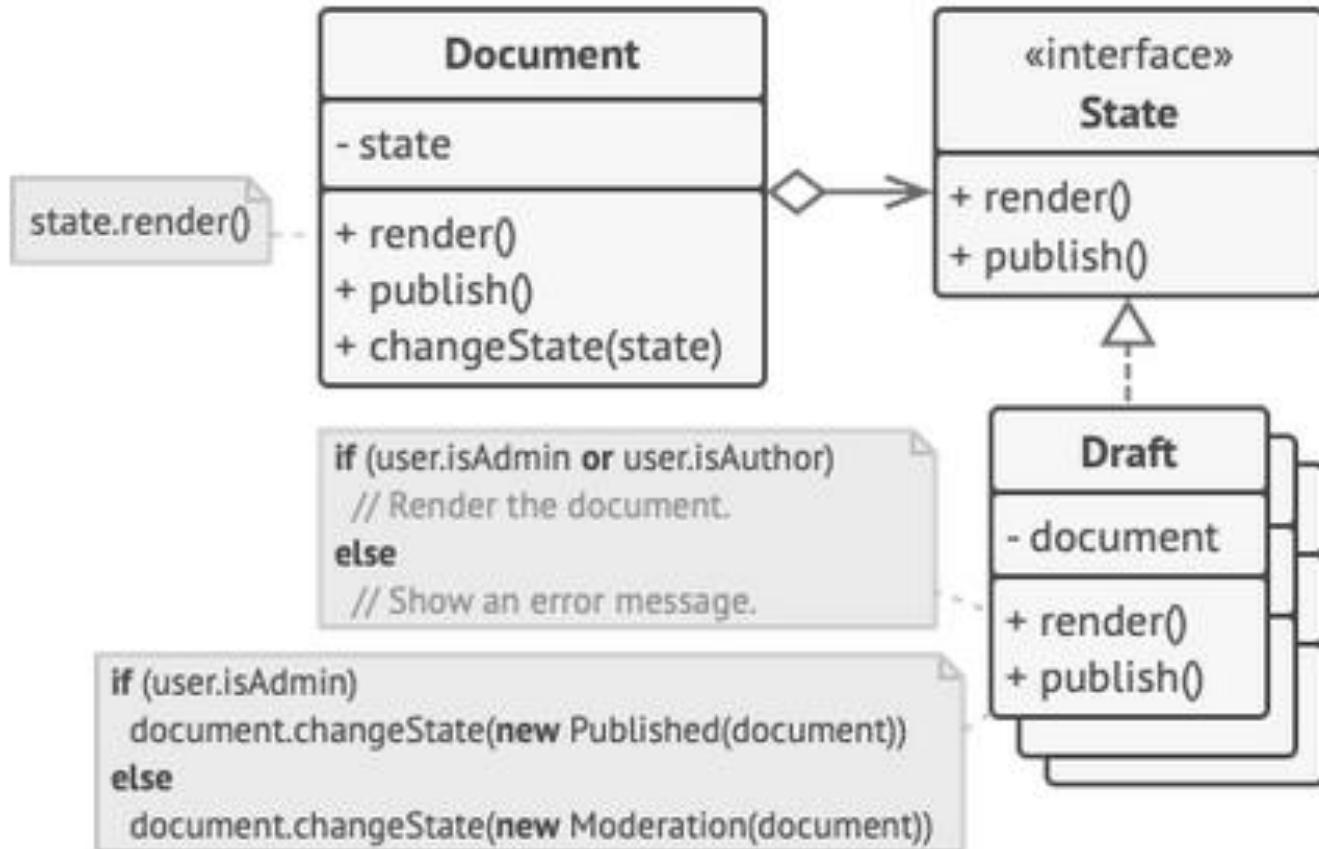
Łańcuch odpowiedzialności



Obserwator



Stan



Document delegates the work to a state object.