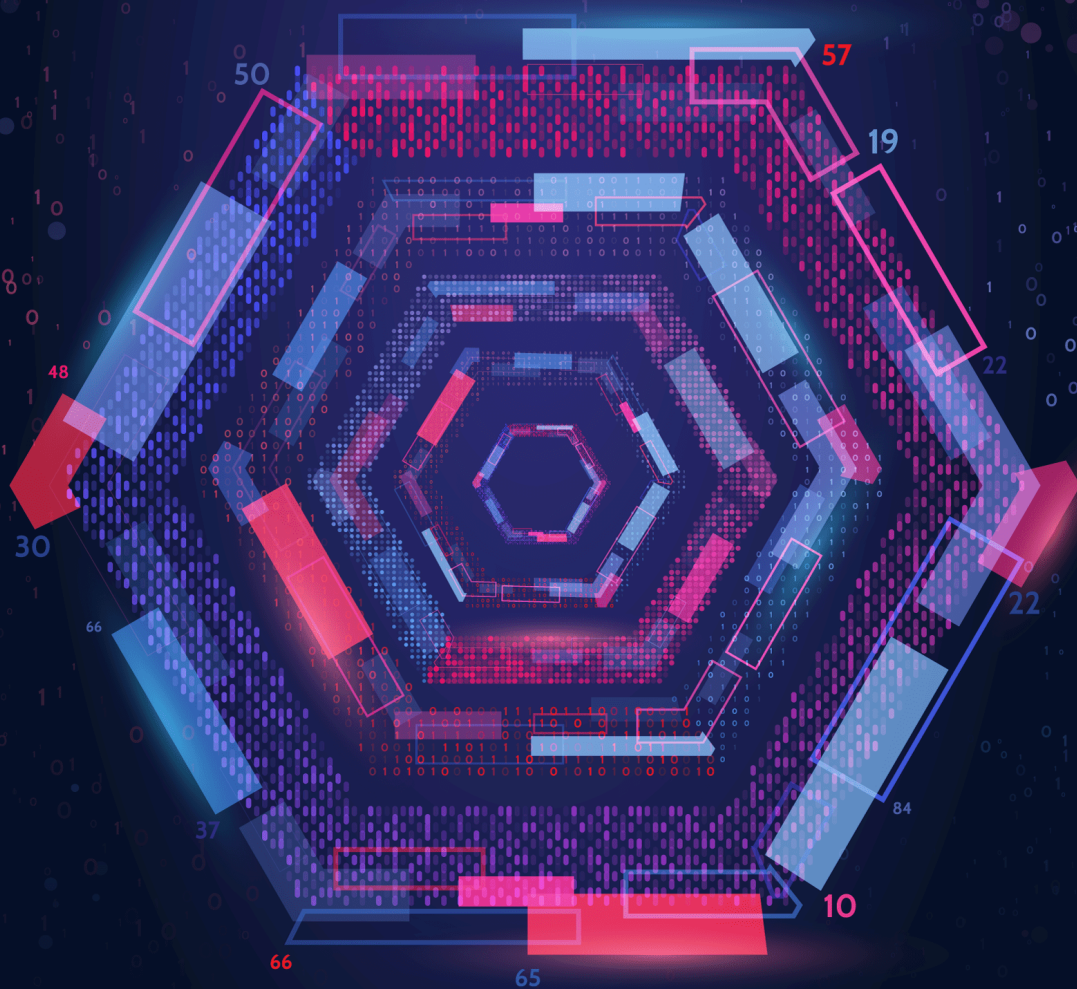


WZORCE PROJEKTOWE

NOWOCZESNY PODRĘCZNIK



Aleksander Shvets

WZORCE PROJEKTOWE

NOWOCZESNY PODRĘCZNIK

v2021-1.7

WERSJA DEMONSTRACYJNA

Kup książkę

<https://refactoring.guru/pl/design-patterns/book>

Parę słów o prawach autorskich

Witajcie! Nazywam się Aleksander Shvets. Jestem autorem książki Wzorce projektowe. Nowoczesny podręcznik oraz kursu online Zanurzenie w refaktoryzacji.



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, 2021

✉ support@refactoring.guru

🖼️ Ilustracje: Dmitry Zhart

🇵🇱 Przekład: Michał Rzepka

✍️ Redakcja: Maciej Włodarczak

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

*Pragnę także wyrazić moją wdzięczność dla
wszystkich dobrych ludzi dzięki którym udało
się wydać polską wersję tej książki.*

Rafał Bera

Filip Borak

Marcin Jagieła

Kamil Kuryś

Bartosz Prokop

Wiesław Ruszowski

Paweł Ryniawiec

Spis treści

| | |
|--|-----------|
| Spis treści | 5 |
| Jak czytać tę książkę | 7 |
| WPROWADZENIE DO PROGRAMOWANIA OBIEKTOWEGO | 8 |
| Podstawy programowania obiektowego..... | 9 |
| Filary programowania obiektowego | 14 |
| Relacje pomiędzy obiektami | 21 |
| WPROWADZENIE DO WZORCÓW PROJEKTOWYCH | 27 |
| Czym jest wzorzec projektowy? | 28 |
| Dlaczego mam poznawać wzorce? | 32 |
| ZASADY PROJEKTOWANIA OPROGRAMOWANIA | 33 |
| Cechy dobrego projektu..... | 34 |
| Zasady projektowania | 39 |
| § Hermetyzuj to, co się różni | 40 |
| § Programuj pod interfejs, nie implementację | 44 |
| § Preferuj kompozycję ponad dziedziczenie..... | 49 |
| Zasady SOLID | 53 |
| § S: Zasada pojedynczej odpowiedzialności..... | 54 |
| § O: Zasada otwarte/zamknięte | 56 |
| § L: Zasada podstawienia Liskov | 60 |
| § I: Zasada segregacji interfejsów | 67 |
| § D: Zasada odwrócenia zależności | 70 |

| | |
|---|------------|
| KATALOG WZORCÓW PROJEKTOWYCH | 74 |
| Wzorce kreacyjne..... | 75 |
| § Metoda wytwórcza | 77 |
| § Fabryka abstrakcyjna..... | 95 |
| § Budowniczy..... | 111 |
| § Prototyp..... | 132 |
| § Singleton | 147 |
| Wzorce strukturalne..... | 157 |
| § Adapter..... | 160 |
| § Most..... | 174 |
| § Kompozyt..... | 190 |
| § Dekorator..... | 204 |
| § Fasada..... | 224 |
| § Pyłek..... | 234 |
| § Pełnomocnik..... | 249 |
| Wzorce behawioralne | 262 |
| § Łańcuch zobowiązań | 266 |
| § Polecenie | 285 |
| § Iterator..... | 306 |
| § Mediator..... | 322 |
| § Pamiątka | 338 |
| § Obserwator | 354 |
| § Stan | 371 |
| § Strategia | 388 |
| § Metoda szablonowa..... | 403 |
| § Odwiedzający..... | 417 |
| Zakończenie | 434 |

Jak czytać tę książkę

Niniejsza książka zawiera opisy 22 klasycznych wzorców projektowych sformułowanych przez “Bandę Czworka” (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 któregośkolwiek 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**.

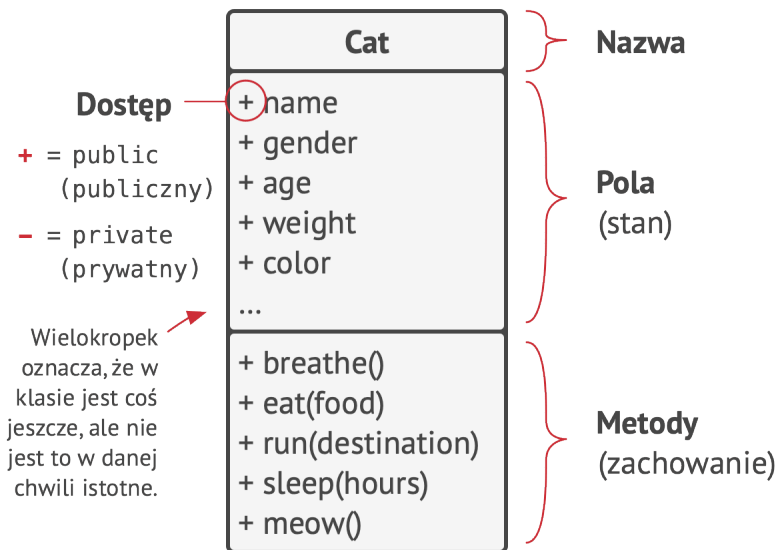
**WPROWADZENIE
DO PROGRAMOWANIA
ZORIENTOWANEGO
OBIEKTOWO**

Podstawy programowania obiektowego

Programowanie zorientowane obiektowo (ang. *OOP* – *Object-oriented programming*) to paradygmat 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 obiektywnym na ich przykładzie.



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

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.

W niniejszej książce odwołuję się do nazw klas poprzez ich polskie nazwy, chociaż na diagramach i w kodzie widnieją po angielsku (jak w przypadku klasy `Kot`). Chciałbym bowiem, aby książkę dało się czytać jak koleżeńską rozmowę, bez wplatania obcojęzycznych słów gdy jest mowa o jakiejś klasie.

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 *pole* klasy.

Wszystkie koty zachowują się podobnie: oddychają, jedzą, biegają, śpią i miauczą. Są to *metody* klasy. Pole 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 zawrze ich wspólne atrybuty i zachowanie.

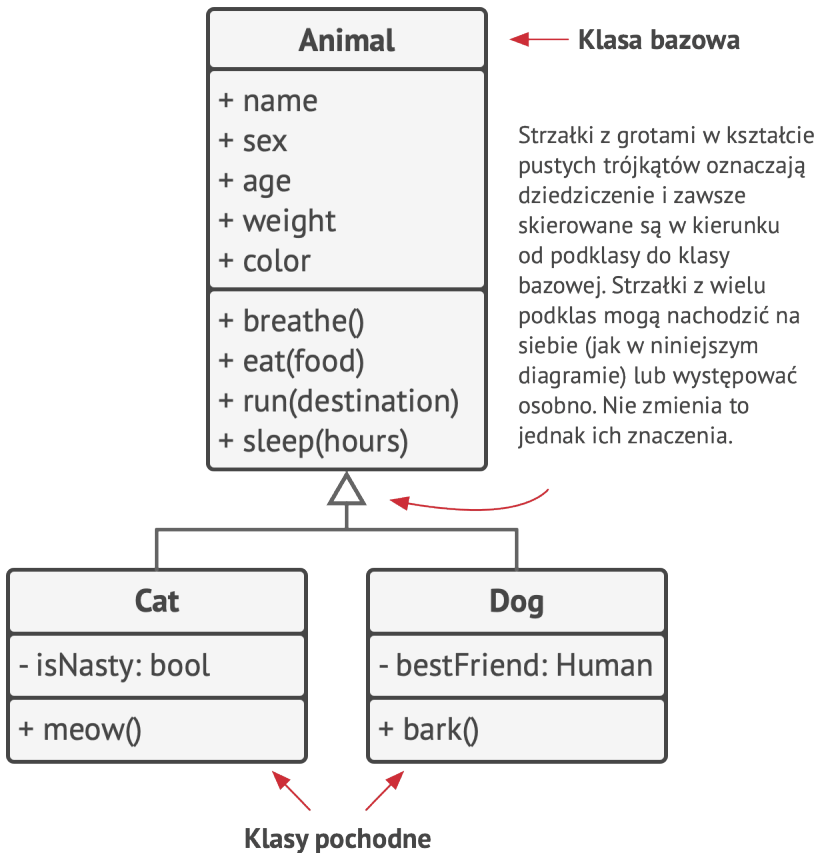
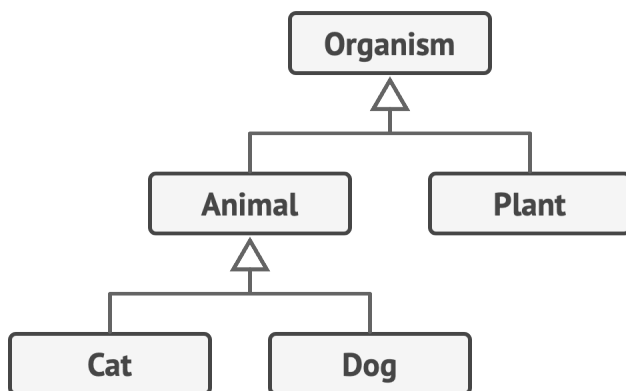


Diagram UML prostej hierarchii klas. Wszystkie klasy w tym diagramie są częścią hierarchii klas `Zwierzę`.

Klasa-rodzic, jak ta którą przed chwilą zdefiniowaliśmy, nazywa się **klasą 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 nadrzędnej. Podklasa może albo całkowicie zamienić domyślne zachowanie, albo tylko je rozbudować.

Liczba stron pominiętych w wersji demonstracyjnej:

19

**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 jakiegokolwiek 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 napracować. Ścisłe sprzęgnięcia 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 Gammy¹, 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. Określają 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 podklasę jakiejś jego części. Stosuje się znaną z Hollywood zasadę “nie dzwoń do nas, my zadzwonimy do ciebie”.

1. Erich Gamma o elastyczności i ponownym wykorzystaniu:

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

Framework pozwala określić zachowanie i daje znać w którym 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 lekomyślny: 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 cał-

kowitą 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 naliczana – zależnie od zmieniającego się prawa. Musisz więc często zmieniać metodę `pobierzSumęZamówienia`. Ale nawet patrząc na nazwę metody można wywnioskować, że nie obchodzi jej *jak* naliczany 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

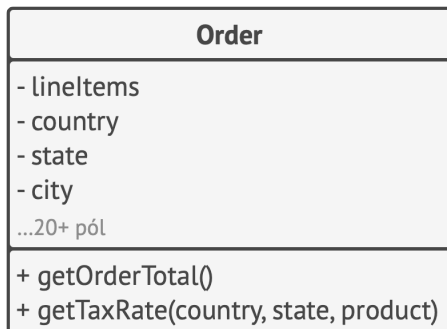
```

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.

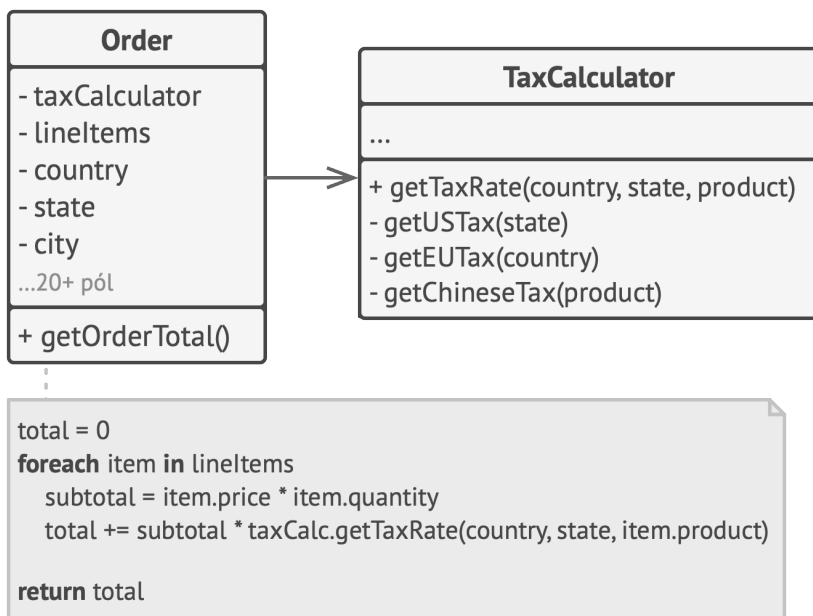
Hermetyzacja na poziomie klasy

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`.

Liczba stron pominiętych w wersji demonstracyjnej:

30

**KATALOG
WZORCÓW
PROJEKTOWYCH**

Wzorce kreacyjne

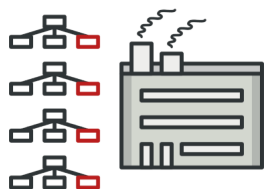
Wzorce kreacyjne są źródłem różnych mechanizmów tworzenia obiektów, zwiększających elastyczność i ułatwiających ponowne użycie kodu.



Metoda wytwórcza

Factory Method

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



Fabryka abstrakcyjna

Abstract Factory

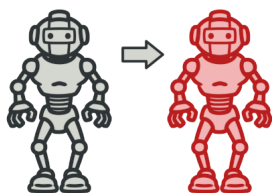
Umożliwia tworzenie rodzin spokrewnionych ze sobą obiektów bez określania ich konkretnych klas.



Budowniczy

Builder

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



Prototyp

Prototype

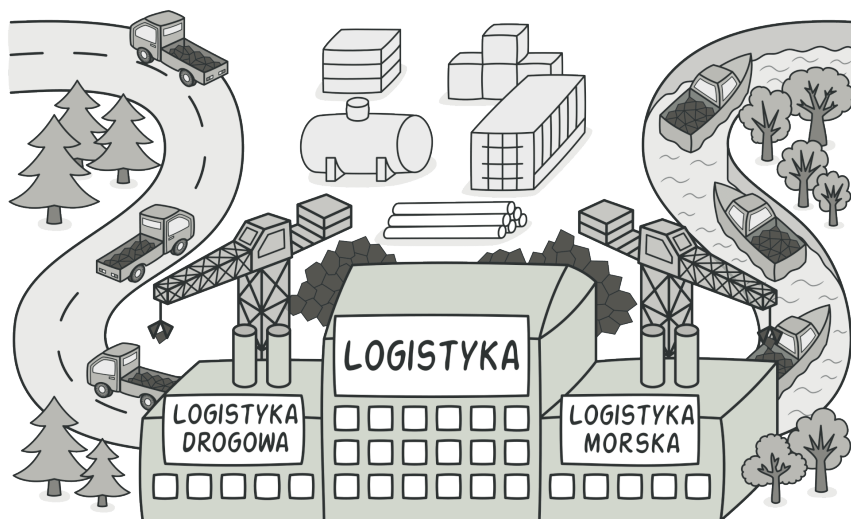
Umożliwia kopiowanie istniejących obiektów bez tworzenia zależności pomiędzy twoim kodem, a ich klasami.



Singleton

Singleton

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



METODA WYTWÓRCZA

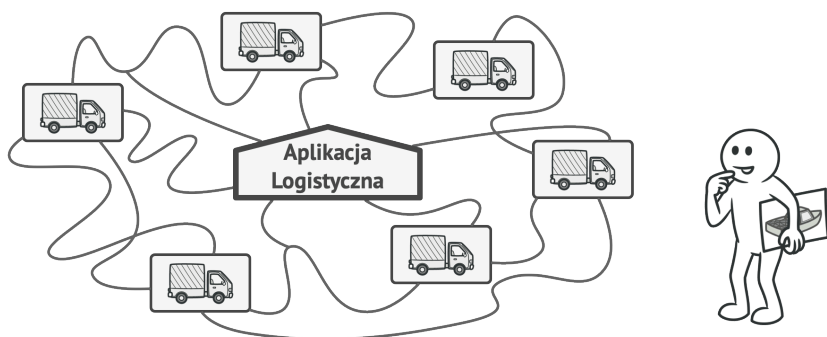
Znany też jako: Konstruktor wirtualny, Virtual constructor, Factory Method

Metoda wytwórcza jest kreatywnym 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 `Ciężarówka`.

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



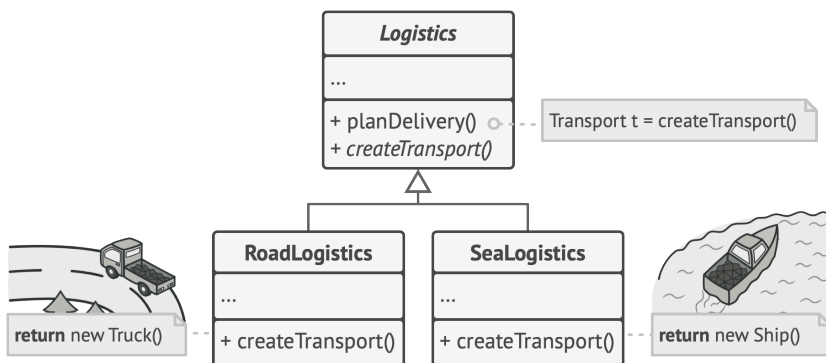
Dodanie nowej klasy do programu nie jest takie proste, jeśli reszta kodu jest już związana z istniejącymi klasami.

Świetna wiadomość, prawda? Ale co z kodem? W tej chwili większość twojego kodu jest powiązana z klasą `Ciężarówka`. 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.

😊 Rozwiązanie

Wzorec projektowy Metody wytwórczej proponuje zamianę bezpośrednich wywołań konstruktorów 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 zwracane przez metodę wytwórczą często są nazywane *produktami*.

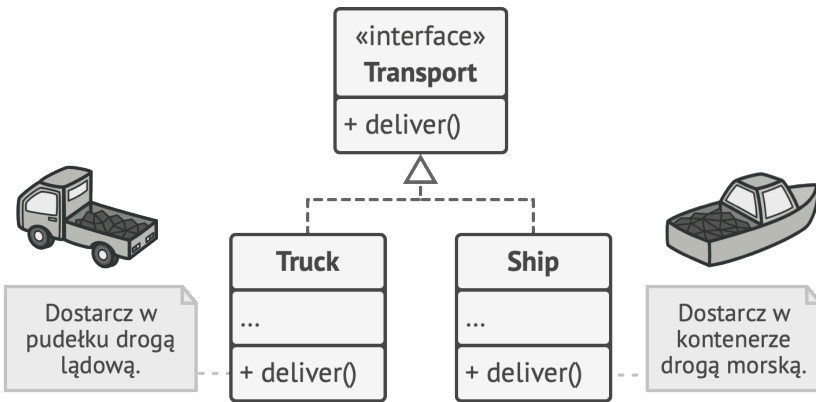


Podklasy mogą zmieniać klasę obiektów zwracanych przez metodę wytwórczą.

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-

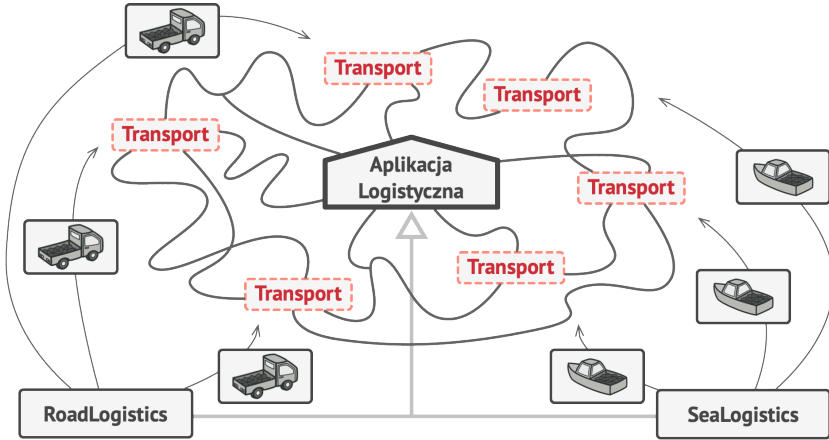
żesz nadpisać metodę wytwórczą w podklasie, a tym samym zmienić klasę produktów zwracanych przez metodę.

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.

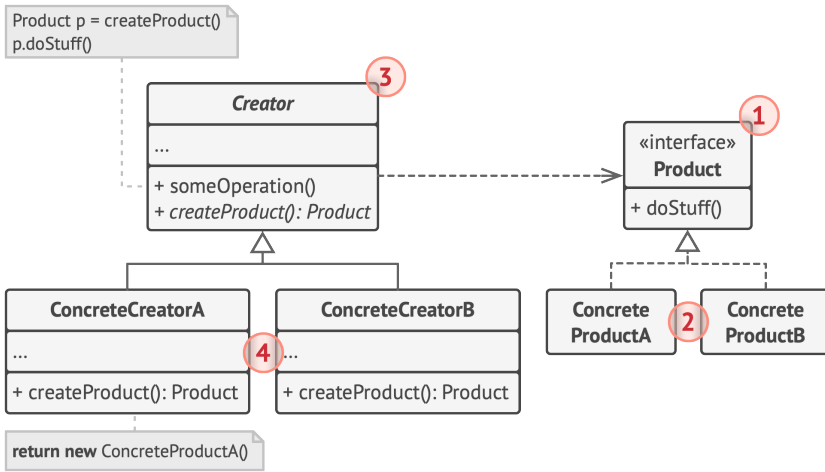
Na przykład zarówno klasy `Ciężarówka`, jak i `Statek` powinny implementować interfejs `Transport`, który z kolei deklaruje metodę `dostarczaj`. Każda klasa różnie implementuje tę metodę: ciężarówki dostarczają towar drogą lądową, statki drogą morską. Metoda wytwórcza znajdująca się w klasie `LogistykaDrogowa` zwraca obiekty `Ciężarówka`, 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 wytwórczą (zwany często kodem *klienckim*) nie widzi różnicy 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. **Konkretne 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.

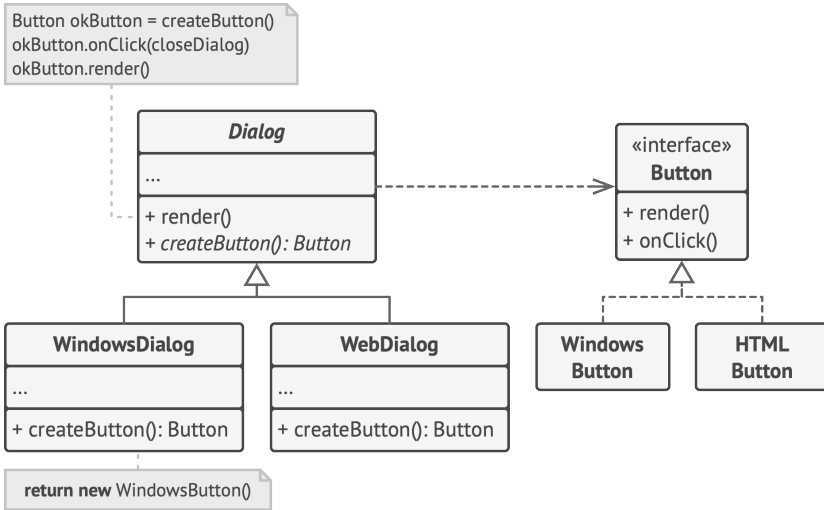
Weź jednak pod uwagę, że wbrew swojej nazwie, tworzenie produktów **nie** jest główną odpowiedzialnością Klasy Twórcy. Zazwyczaj klasa kreatywna zawiera już jakąś ważną logikę biznesową związaną z produktami. Metoda wytwórcza pomaga rozpręgnąć tę logikę i konkretne klasy produktów. Oto analogia: duża 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 wytwórczą, co sprawia, że zwraca ona inny typ obiektu.

Tu jednak uwaga: Metoda wytwórcza 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 Wytwórcza** może służyć tworzeniu elementów interfejsu użytkownika (UI) które będą przenośne między platformami. Nie występuje tu sprzęgnięcie 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 podklasę 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, zbliżasz 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     // podklasę 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 podklasę 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 konstruuującego komponenty na przestrzeni frameworku 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). Twoja aplikacja ma posiadać okrągłe przyciski, ale framework oferuje jedynie prostokątne. Rozszerzasz więc standardową klasę `Przycisk` o podklasę `OkrągłyPrzycisk`. Ale teraz trzeba również sprawić, by główna klasa `UIFramework` korzystała z nowo utworzonej podklasy, zamiast z domyślnej.

Aby to osiągnąć, tworzysz podklasę `UIZ0krągłymiPrzyciskami` z bazowej klasy frameworku 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 `UIZ0krągłymiPrzyciskami` 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`. Klasy `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 podklasę (powiedzmy, że `PocztaKolejowa`) aby obsłużyć oba przypadki, ale istnieje lepszy sposób. Kod kliencki 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.

Zalety i wady

- ✓ Unikasz ścisłego sprzężenia 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).
- Klasy **Fabryka abstrakcyjna** często wywodzą 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 podejściu 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*.

Liczba stron pominiętych w wersji demonstracyjnej:
340