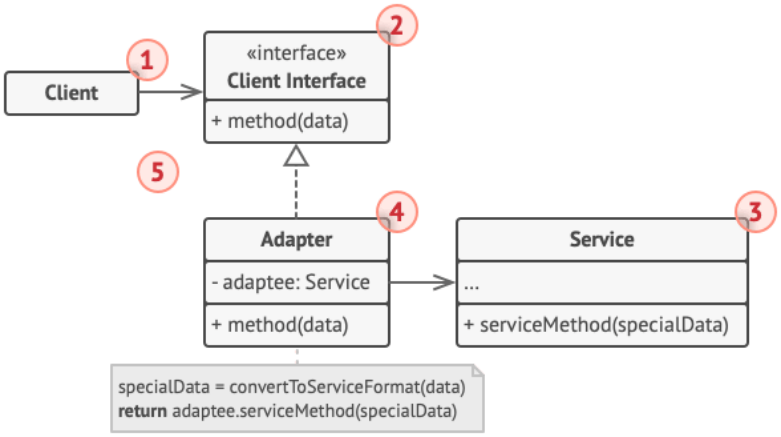
1. Perspektywa programowania obiektowego
    a. Część wspólna problemu- interfejs (klasa abstrakcyjna)
    b. Część zmienna- zmienność do zidentyfikowana i ukrycia wewnątrz części wspólnej (określa klasy pochodne)
    c. Specyfikacja- interfejs klasy abstrakcyjnej
2. Wzorzec architektoniczny
    a. Rozwiązanie często pojawiających się, powtarzalnych problemów z architektury
    b. Ogólna struktura systemu, elementy składowe (funkcjonalności i komunikacja między nimi)
    c. Problemy z tworzeniem systemu, organizacją systemu oraz jego modułów (wzorce projektowe dotyczą problemów na poziomie obiektów i klas)
3. Wzorce konstrukcyjne- tworzenie obiektów
4. Wzorce strukturalne- składanie klas i obiektów w większe struktury zachowując elastyczność
5. Wzorce czynnościowe- komunikacja oraz podział obowiązków między obiektami

1. Fasada
    a. Wzorzec **strukturalny**
    b. https://github.com/iluwatar/java-design-patterns/tree/master/facade
    c. https://refactoring.guru/pl/design-patterns/facade
    d. Wyposaża złożony system klas, bibliotekę lub framework w uproszczony interfejs
    e. Ukrycie logiki biznesowej- prosty interfejs dla złożonego podsystemu
    f. Interfejs deleguje swoje zadania innym klasom
    g. Zastosowanie:
        i. Wiele klas może być ciężkich do zrozumienia, dlatego łatwo stworzyć fasadę
        ii. Fasada dostarcza prosty widok podsystemu
        iii. Można zrobić fasadę jako punkt wejścia do każdej warstwy podsystemu. Jeżeli podsystemy zależą od siebie, można spowodować, by warstwy komunikowały się przez swoje fasady.

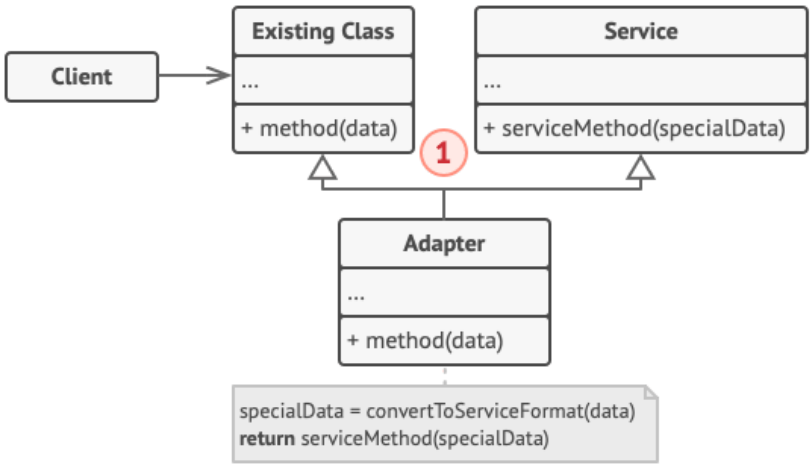
```
public void startEngine() {
    fuelInjector.on();
    airFlowController.takeAir();
    fuelInjector.on();
    fuelInjector.inject();
    starter.start();
    coolingController.setTemperatureUpperLimit(DEFAULT_COOLING_TEMP);
    coolingController.run();
    catalyticConverter.on();
}

public void stopEngine() {
    fuelInjector.off();
    catalyticConverter.off();
    coolingController.cool(MAX_ALLOWED_TEMP);
    coolingController.stop();
    airFlowController.off();
}
```

2. Adapter
    a. Wzorzec **strukturalny**
    b. https://github.com/iluwatar/java-design-patterns/tree/master/adapter
    c. https://refactoring.guru/pl/design-patterns/adapter
    d. Przekształcenie interfejsu klasy na interfejs którego klient oczekuje. Umożliwia współpracę klas o niekompatybilnych interfejsach.
    e. Współdziałanie ze sobą elementów o niekompatybilnych interfejsach
    f. Aplikacja pobiera dane XML, ale musi je eksportować do biblioteki JSON
        i. Adapter uzyskuje interfejs jednego z obiektów
        ii. Istniejący obiekt może za pomocą tego interfejsu wywoływać metody adaptera
        iii. Otrzymawszy wywołanie, adapter przekazuje je dalej, ale już w formacie obsługiwanym przez opakowany obiekt
    g. Adapter implementuje interfejs jednego z obiektów i opakowuje drugi obiekt
    h. Adapter obiektu



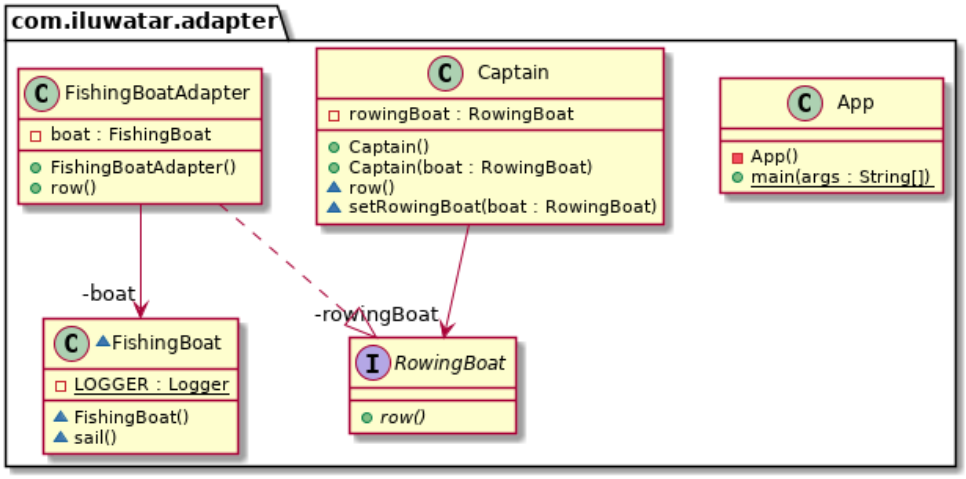
        i. (1) Client zawiera logikę biznesową systemu. (2) Interfejs klienta to kontrakt którego muszą się trzymać klasy z nim współpracujące. (3) Service to użyteczna klasa z niekompatybilnym interfejsem. (4) Adapter implementuje interfejs klienta, opakowując Service. Otrzymuje wywołania, konwertuje je i przekazuje do Service.
    i. Zastosowanie:
        i. Można wprowadzać nowe typy adapterów, bo Client nie jest sprzężony z konkretnym adapterem.
        ii. Stosujemy kiedy chcemy wykorzystać istniejącą klasę, ale jej interfejs nie jest kompatybilny

j. Adapter klasy- dla języków z wielodziedzieczeniem

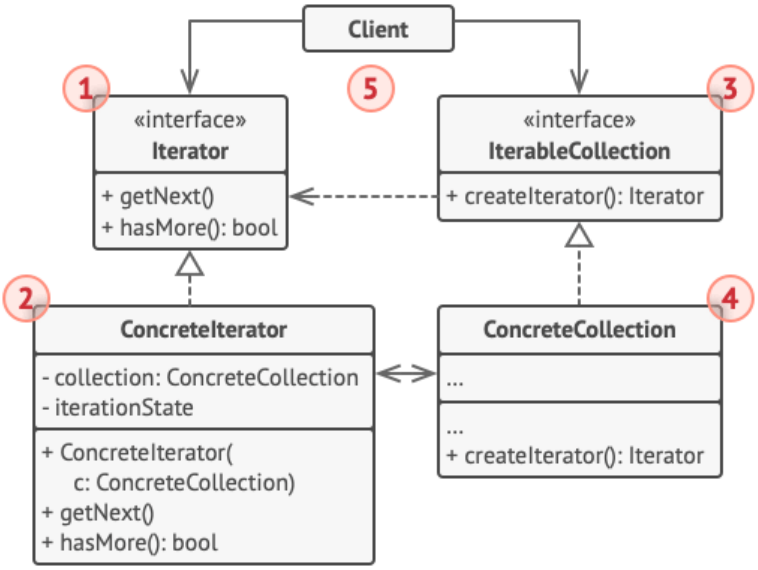


i. Adapter dziedziczy interfejsy obu obiektów jednocześnie.
ii. Adapter nie opakowuje obiektów, bo dziedziczy zachowanie klienta oraz usługi. Adaptacja odbywa się w nadpisanych metodach. Adapter może być użyty w miejsce klasy klienckiej.

k.

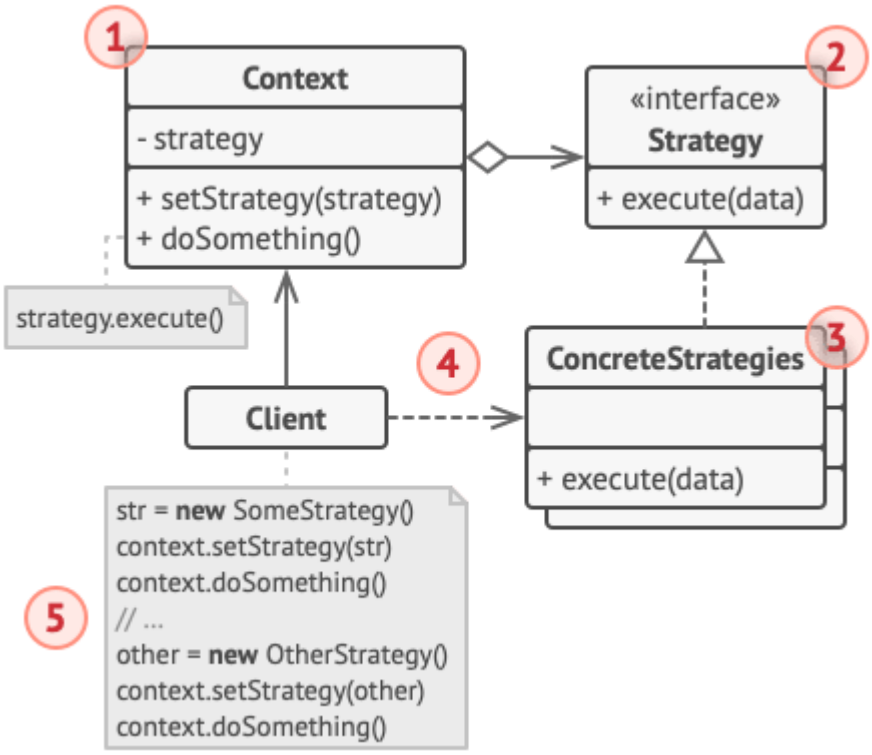


3. Iterator
    a. Wzorzec **behawioralny**
    b. https://refactoring.guru/pl/design-patterns/iterator
    c. https://github.com/iluwatar/java-design-patterns/tree/master/iterator
    d. Sekwencyjne przechodzenie od elementu do elementu bez konieczności eksponowania formy zbioru.
    e. Ekstrakcja zadań związanych z przechodzeniem przez elementy kolekcji do osobnego obiektu zwanego iteratorem.
    f. 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.
    g. Struktura



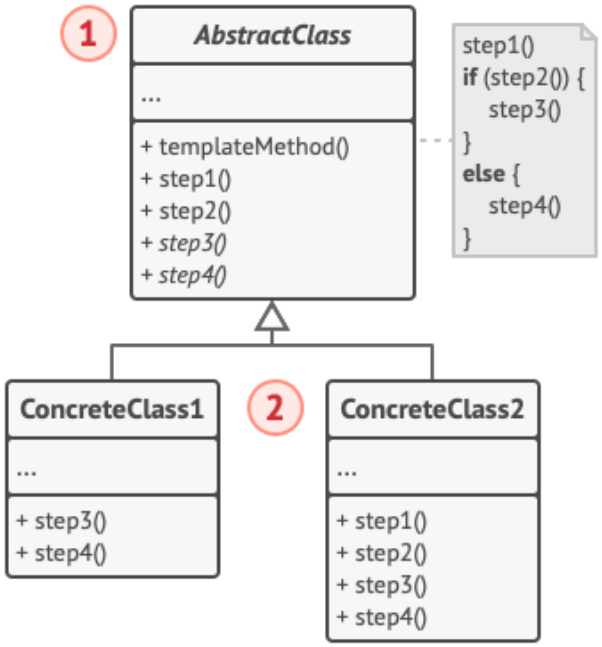
    h. (1) Interfejs Iterator deklaruje niezbędne działania do przechodzenia po kolekcji.
    i. (2) Konkretny Iterator implementuje specyficzny algorytm przeglądania kolekcji. Sam samodzielnie śledzi ten proces.
    j. (3) Interfejs kolekcja deklaruje metody do otrzymywania iteratora. Metoda zwraca interfejs iteratora.
    k. (4) Konkretne kolekcje zwracają nowe instancje konkretnych klas iteratorów, gdy klient ich zażąda.
    l. (5) Klient współpracuje z kolekcjami jak i iteratorami za pomocą interfejsów. Umożliwia to pracę z różnymi kolekcjami i operatorami. Klienci nie tworzą iteratorów sami, lecz otrzymują je od kolekcji.
    m. Zastosowanie:
        i. Używamy gdy chcemy przeglądać strukturę bez ujawnienia jej wewnętrznej reprezentacji.
        ii. Unikanie duplikowania kodu przy przeglądaniu elementów.
        iii. Używany gdy mamy skomplikowaną strukturę którą ciężko przeglądać.
        iv. By kod był w stanie przeglądać różne struktury, nie znając ich szczegółów. Można przekazać różne rodzaje kolekcji i iteratorów, byleby implementowały ten interfejs.

1. Strategia
    a. Wzorzec **behawioralny**
    b. https://refactoring.guru/pl/design-patterns/strategy
    c. https://github.com/iluwatar/java-design-patterns/tree/master/strategy
    d. Zdefiniowanie rodziny algorytmów i ekstrakcja poszczególnych algorytmów jako klasy- strategie
    e. Pozwala zmieniać algorytmy niezależnie od wykorzystujących je obiektów
    f. Używanie kilku wariantów jednego algorytmu i ich zmiany podczas działania programu
    g. Warto używać gdy mamy wiele podobnych klas, różniących się sposobem wykonywania zadań
    h. Kontekst zawiera pole odnoszące się do strategii. Kontekst deleguje pracę strategii. Klient przekazuje strategię kontekstowi.
    i. Struktura



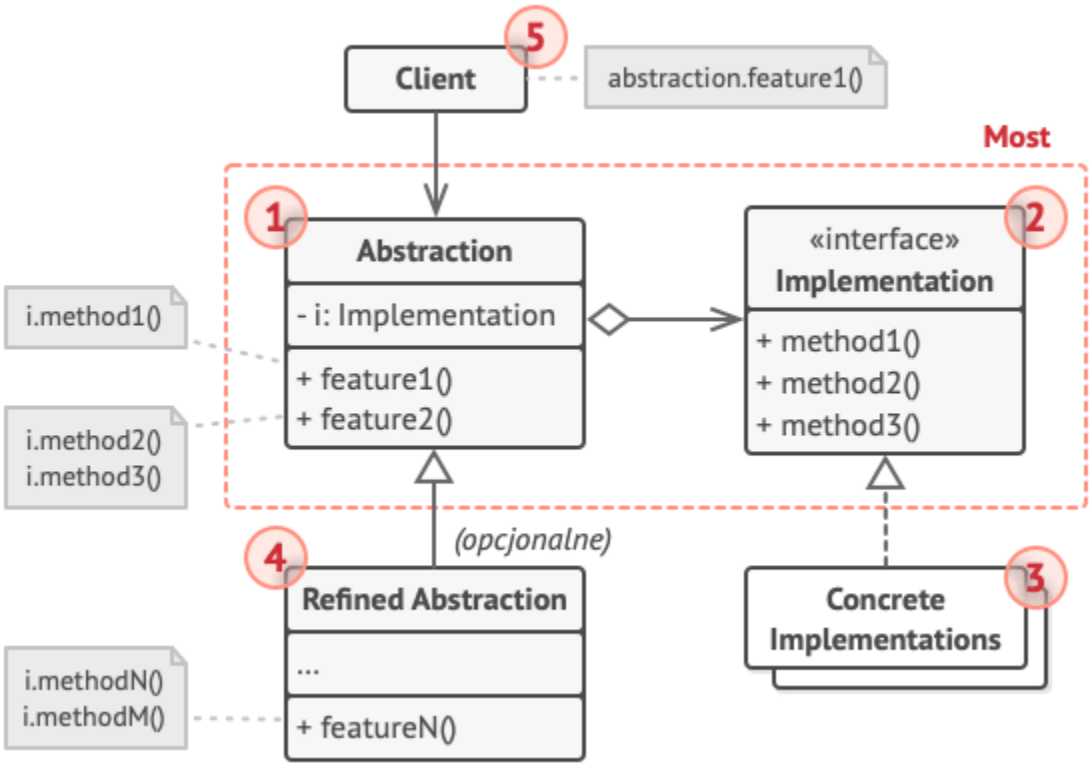
    j. (1) Kontekst ma odniesienie do jednej strategii za pomocą interfejsu.
    k. (2) (3) Różne strategie implementują interfejs strategia.
    l. (4) Kontekst wywołuje metodę ze strategii, za każdym razem jak klient chce uruchomić algorytm.
    m. (5) Klient przekazuje obiekty strategii kontekstowi. Kontekst pozwala zmieniać strategię podczas działania programu.
    n. Zastosowanie:
        i. Używanie różnych wariantów jednego algorytmu.
        ii. Gdy mamy wiele podobnych klas różniących się sposobem wykonywania zadań.
        iii. Izolacja logiki biznesowej od szczegółów implementacyjnych.
        iv. Gdy mamy duży if wybierający wariant algorytmu.

2. Template method
    a. Wzorzec **behawioralny**
    b. https://refactoring.guru/pl/design-patterns/template-method
    c. https://github.com/iluwatar/java-design-patterns/tree/master/template-method
    d. Definiuje szkielet algorytmu w klasie bazowej, ale pozwala podklasom nadpisać pewne etapy algorytmu.
    e. Rozdzielenie algorytmu na etapy, utworzenie metod z etapów i umieszczenie ciągu wywołań jako metodę szablonową
    f. Etapy mogą mieć domyślną implementację
    g. Struktura



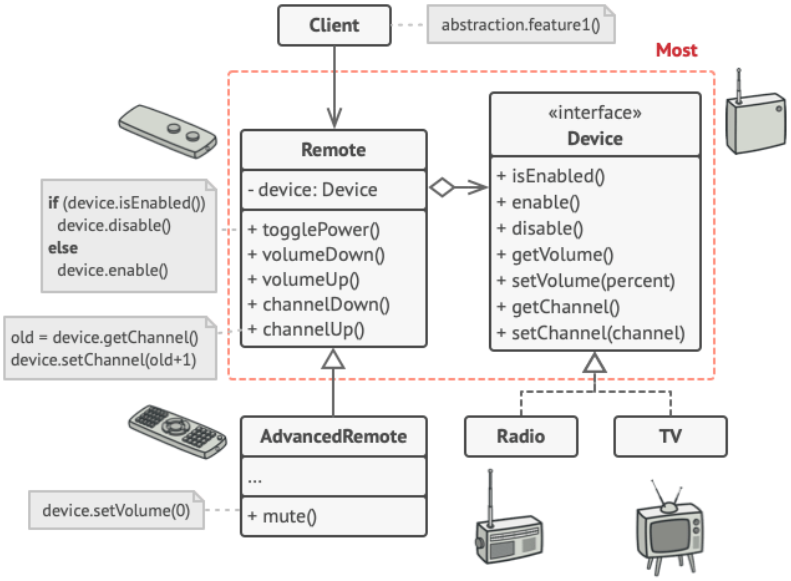
    h. (1) Klasa abstrakcyjna deklaruje metody będące etapami algorytmu oraz metodę szablonową, która wywołuje poszczególne etapy w poszczególnej kolejności. Etapy mogą być abstrakcyjne lub posiadające domyślną implementację
    i. (2) Konkretne klasy rozszerzające klasę abstrakcyjną mogą nadpisać każdy z etapów **oprócz metody szablonowej**
    j. Zastosowanie:
        i. Używany gdy chcemy rozszerzyć niektóre etapy algorytmu, bez zmiany jego struktury
        ii. Przydatny, gdy mamy wiele klas z podobnymi algorytmami różniącymi się tylko szczegółami

1. Most
    a. Wzorzec **strukturalny**
    b. https://refactoring.guru/pl/design-patterns/bridge
    c. https://github.com/iluwatar/java-design-patterns/tree/master/bridge
    d. Próbujemy poszerzyć klasy w dwóch niezależnych wymiarach- kształt oraz kolor
    e. Rozdzielamy dużą klasę lub zestaw klas na dwie hierarchie- abstrakcję i implementację
    f. Kompozycja ponad dziedziczeniem. Szczegóły implementacji są przeniesione z hierarchii do innego obiektu, mającego własną hierarchię.
    g. Zmieniamy dziedziczenie na kompozycję-wyodrębniamy jeden z wymiarów, tworząc osobną hierarchię klas. Pierwotne obiekty mają odniesienie do nowej hierarchii, zamiast przechowywać wszystkie stany wewnątrz klasy.
    h. Kod kolorów będzie w osobnej klasie, z podklasami Czerwony i Niebieski. Klasa Figura mam pole przechowujące kolor.
    i. Abstrakcja stanowi wysokopoziomową warstwę umożliwiając kontrolę nad implementacją. Abstrakcja to GUI, a implementacja to API.
    j. Aplikację taką można rozwijać w dwóch kierunkach. Posiadać wiele różnych GUI oraz API.
    k. Struktura



    l. (1) Abstrakcja ma logikę wysokiego poziomu. Potrzebuje implementacji by wykonywać działania
    m. (2) Implementacja deklaruje interfejs, wspólny dla wszystkich implementacji. Abstrakcja komunikuje się z implementacją tylko za pomocą tych metod. Abstrakcja może zawierać listę tych samych metod co implementacja, ale zawiera bardziej złożone działania, na które składa się wiele prostszych działań z implementacji.
    n. (3) Konkretne implementacje zawierają kod specyficzny dla danej platformy.
    o. (4) Wzbogacona abstrakcja udostępnia warianty logiki kontrolnej

p. (5) Klienta interesuje współpraca z abstrakcją. Klient łączy obiekt abstrakcji z obiektem implementacji.



q. Zastosowanie

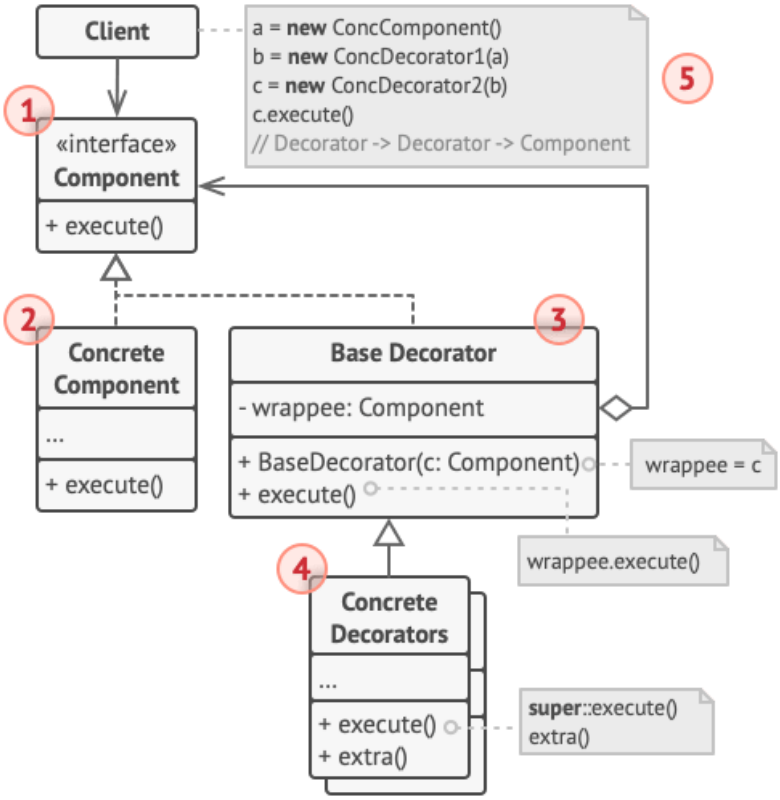
i. By przeorganizować monilityczną klasę mającą wiele wariantów tej samej funkcjonalności (wiele baz danych). Wzorzec Most pozwala rozdzielić monolityczną klasę w wiele hierarchii klas. Możemy wówczas zmieniać klasy w jednej z hierarchii niezależnie od klas w drugiej. Podejście to upraszcza utrzymanie kodu i pozwala zminimalizować ryzyko zepsucia istniejącego kodu

ii. Rozszerzenie klasy na kilku niezależnych płaszczyznach- rozszerzamy abstrakcję i implementację.

iii. Chcemy uniknąć łączenia między abstrakcją a implementacją.

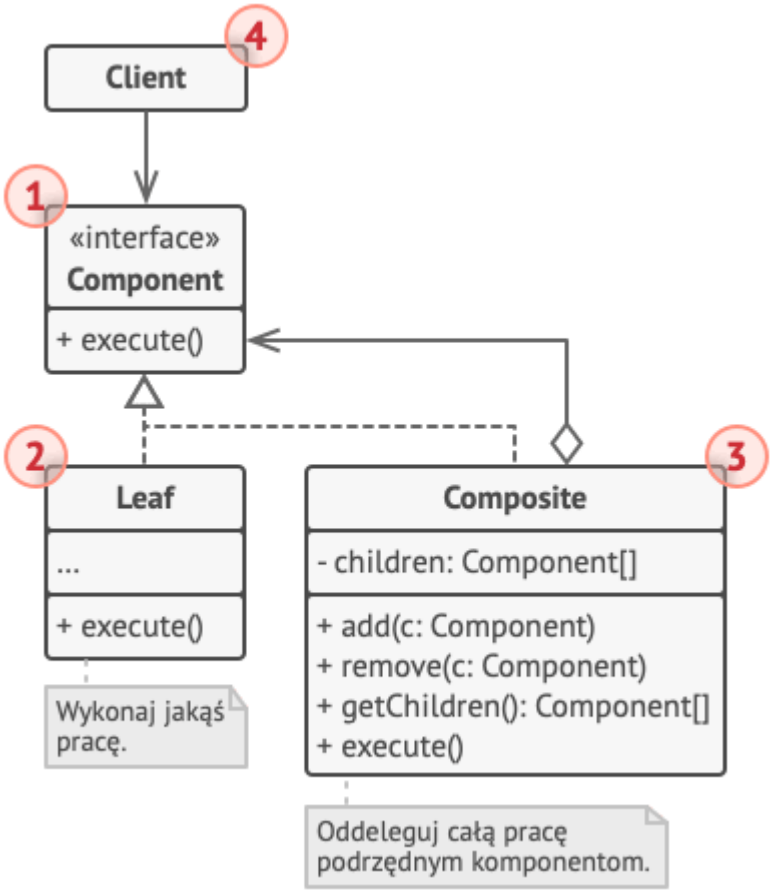
iv. Możliwość rozszerzania

1. Dekorator
    a. Wzorzec **strukturalny**
    b. https://refactoring.guru/pl/design-patterns/decorator
    c. https://github.com/iluwatar/java-design-patterns/tree/master/decorator
    d. Dodawanie nowych obowiązków obiektom, przez umieszczenie ich w obiektach opakowujących
    e. Musimy zmienić zachowanie obiektu. Dekorator to nakładka, zawiera ten sam zestaw metod co obiekt docelowy- implementuje ten sam interfejs. Może jednak wykonać coś przed lub po przekazaniu żądania.
    f. Dekorator implementuje ten sam interfejs co klasa bazowa, to klient nie będzie musiał wiedzieć czy pracuje na czystym obiekcie czy na udekorowanym
    g. Struktura



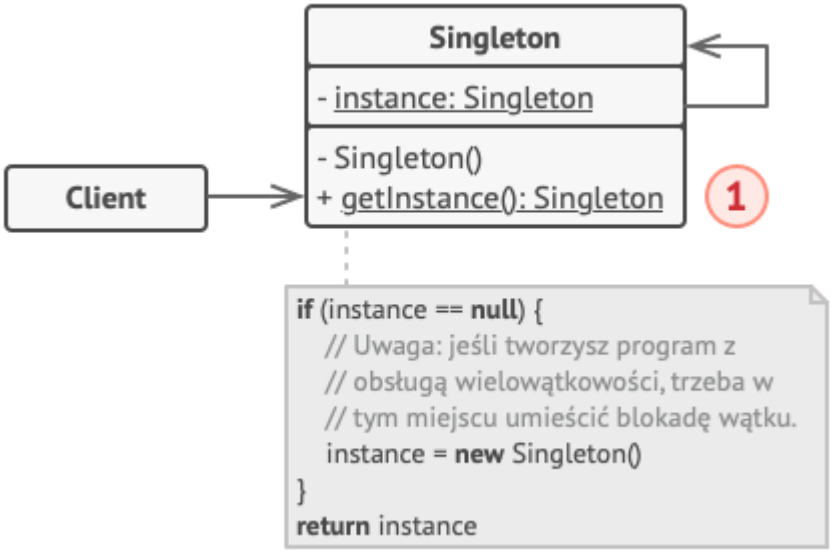
    h. (1) Komponent deklaruje interfejs wspólny dla obiektów jak i dekoratorów
    i. (2) Konkretny komponent to klasa z podstawowym zachowaniem
    j. (3) Bazowy dekorator posiada referencję do opakowania obiektu- pole interfejs, żeby mógł przechowywać zarówno komponenty jak i dekoratory. Dekorator deleguje działania opakowywanemu obiektowi.
    k. (4) Konkretni dekoratorzy definiują dodatkowe zachowania, nadpisują metody dekoratora bazowego
    l. (5) Klient opakowuje komponenty w wiele warstw dekoratorów, działając przez interfejs komponentu.
    m. Zastosowanie:
        i. Użyć gdy rozszerzenie zakresu za pomocą dziedziczenia jest niepraktyczne lub niemożliwe.
        ii. Gdy chcemy przypisać dodatkowe obowiązki w trakcie działania programu.

2. Kompozyt
    a. Wzorzec **strukturalny**
    b. https://refactoring.guru/pl/design-patterns/composite
    c. https://github.com/iluwatar/java-design-patterns/tree/master/composite
    d. Komponowanie obiektów w **struktury drzewiaste**, traktowanie prostych jak i złożonych struktur jednakowo
    e. Tworzenie interfejsu, deklarującego metodę obliczenia sumy
    f. Struktura



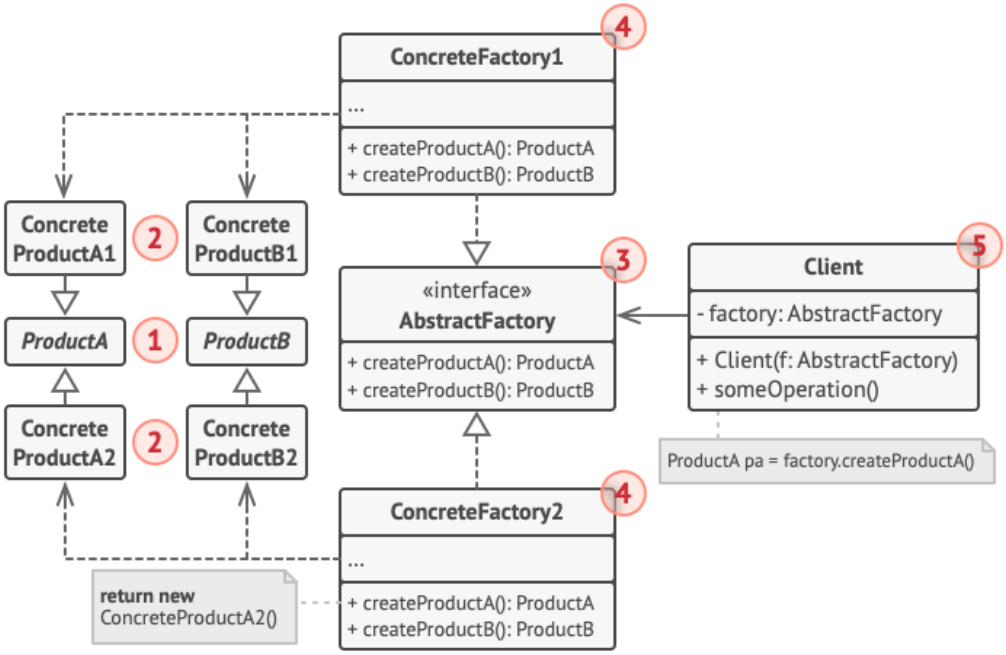
    g. (1) Interfejs Component zawiera metody wspólne dla prostych jak i złożonych elementów drzewa
    h. (2) Liść nie ma elementów podrzędnych, wykonuje pracę
    i. (3) Kompozyt- kontener posiada elementy podrzędne. Komunikuje się z elementami podrzędnymi przez interfejs komponentu. Kiedy otrzymuje żądanie, kontener deleguje pracę do swoich podelementów, przetwarza wyniki i zwraca ostateczny wynik
    j. (4) Klient współpracuje z elementami za pomocą interfejsu Component. Klient może działać tak samo na prostych jak i złożonych elementach drzewa.
    k. Zastosowanie
        i. Drzewiasta struktura obiektów
        ii. Kod kliencki ma traktować proste jak i złożone elementy jednakowo

3. Singleton
    a. Wzorzec **kreacyjny**
    b. https://refactoring.guru/pl/design-patterns/singleton
    c. Struktura

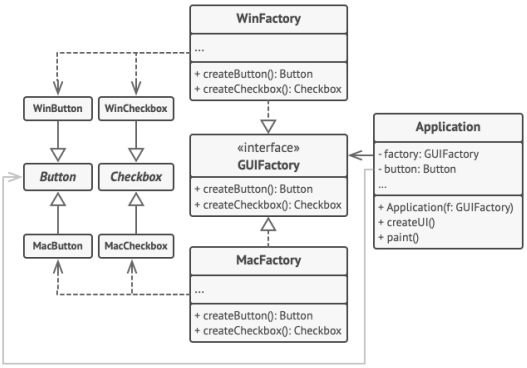


    d. (1) mamy statyczną metodę getInstance która zawsze zwraca tę samą instancję danej klasy. Konstruktor jest prywatny.
    e. Zastosowanie
        i. Jedna ogólnodostępna instancja danej klasy
        ii. Połączenie z bazą danych
        iii. Kontrola nad zmiennymi globalnymi

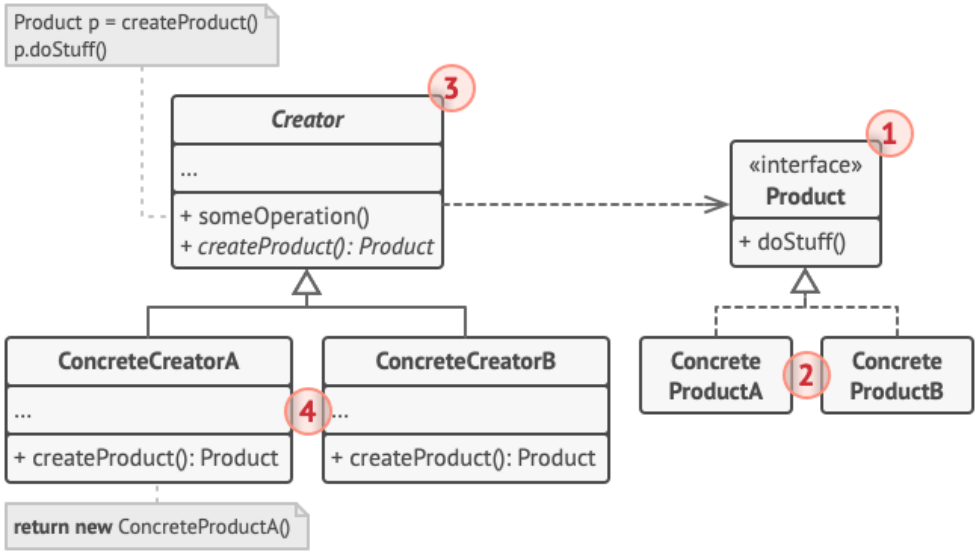
1. Fabryka abstrakcyjna
    a. Wzorzec **kreacyjny**
    b. https://refactoring.guru/pl/design-patterns/abstract-factory
    c. https://github.com/iluwatar/java-design-patterns/tree/master/abstract-factory
    d. Tworzenie rodzin spokrewnionych ze sobą obiektów bez określania konkretnych klas
    e. Tworzenie interfejsów dla typu produktu (Fotel, Sofa) oraz interfejsu Fabryki abstrakcyjnej, którego implementacje zwracają obiekty
    f. Struktura



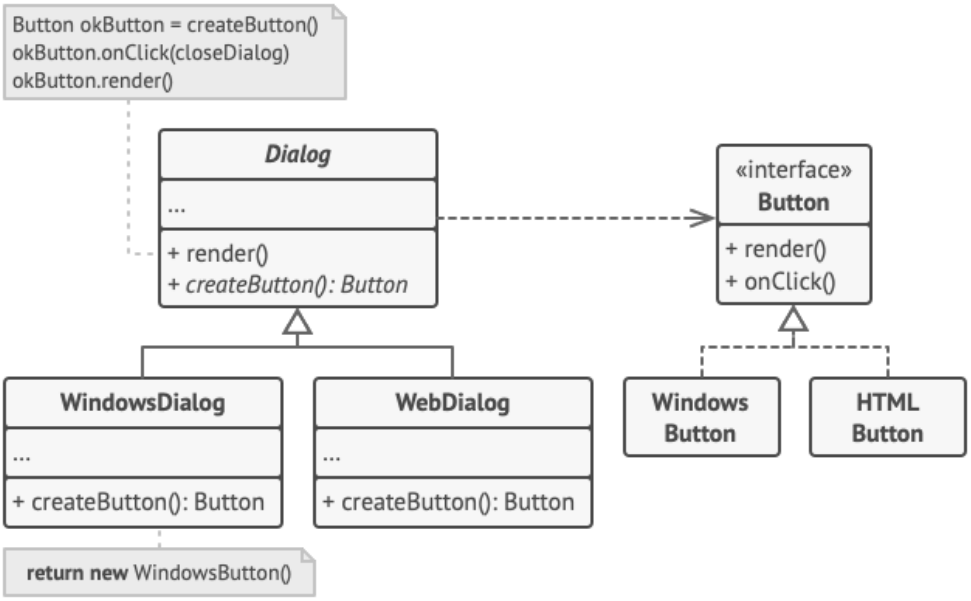
    g. (1) Mamy interfejsy zwracanych odmiennych produktów, składających się na rodzinę
    h. (2) Konkretne produkty to implementacje produktów. Każdy produkt musi być zaimplementowany we wszystkich wariantach.
    i. (3) Interfejs fabryki abstrakcyjnej deklaruje zestaw metod do tworzenia obiektów.
    j. (4) Konkretne fabryki implementują metody kreacyjne. Każda fabryka jest związana z jakimś wariantem produktu i produkuje produkty z tego wariantu.
    k. (5) Konkretne fabryki tworzą konkretne egzemplarze produktu, ale ich zwracany typ to interfejs. Dzięki temu kod korzystający z fabryki nie jest sprzęgnięty z konkretnym wariantem produktu.
    l. Zastosowanie
        i. Kod ma działać na produktach z różnych rodzin. Kod nie będzie zależał od konkretnych klas produktów, mogą one być jeszcze nieznane lub do potencjalnego rozszerzenia.
    m.



2. Factory method
    a. Wzorzec **kreacyjny**
    b. https://refactoring.guru/pl/design-patterns/factory-method
    c. https://github.com/iluwatar/java-design-patterns/tree/master/factory-method
    d. Wzorzec udostępnia interfejs do tworzenia obiektów w ramach klasy bazowej. Pozwala podklasom zmieniać typ tworzonych obiektów.
    e. Factory method powoduje zmianę wywołań konstruktorów obiektów na wywołanie metody wytwórczej. Obiekty są tworzone wewnątrz metody wytwórczej.
    f. Kod kliencki, korzystający z metody wytwórczej nie widzi różnicy między produktami zwróconymi, wszystkie produkty są traktowane przez pryzmat interfejsu
    g. Struktura



    h. (1) Product deklaruje wspólny interfejs dla obiektów zwracanych przez Creator i jego podklasy
    i. (2) Product ma swoje implementacje
    j. (3) Klasa Creator ma metodę wytwórczą, zwracającą typ interfejsu. Metoda wytwórcza może być abstrakcyjna, lub zwracać domyślny typ produktu, będzie ona nadpisywana. Klasa Creator ma **logikę biznesową** związaną z produktami, metoda wytwórcza pomaga rozprzęgnąć tą logikę i konkretne klasy produktów.



k.

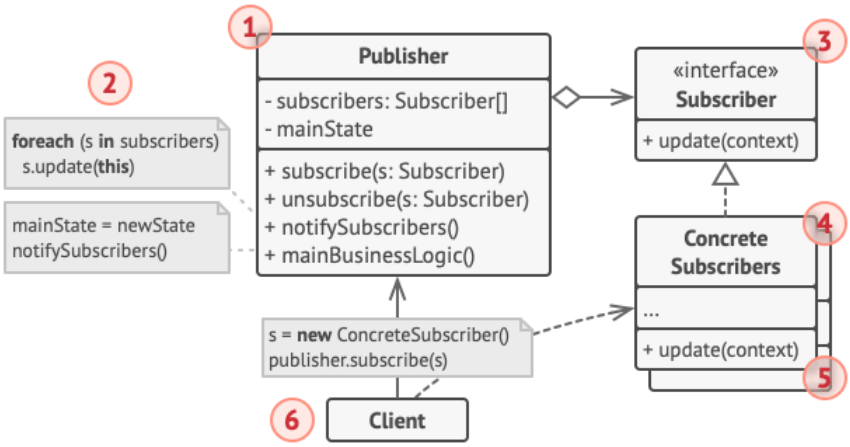
l. Zastosowanie:

   i. Użyć kiedy nie wiadomo jakie typy obiektów pojawią się w programie. Oddziela kod tworzący produkty od kodu z niego korzystających. Łatwiej rozszerzać kod konstruujący produkty bez ingerencji w resztę kodu.

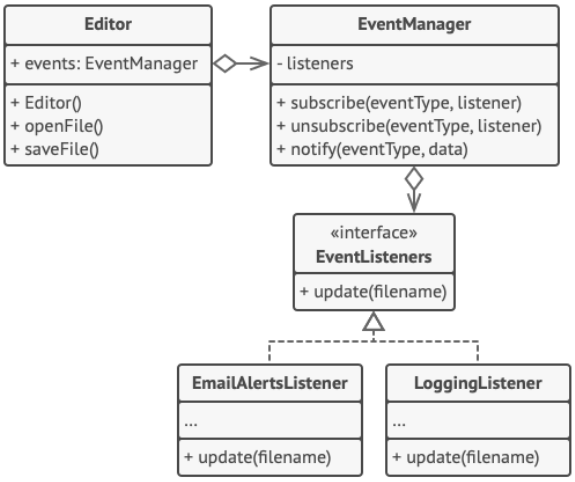
   ii. Używać by pozwolić użytkownikom biblioteki na rozbudowę komponentów. Aby dodać nowy typ produktu, trzeba utworzyć klasę produktu i napisać klasę kreacyjną wraz z metodą wytwórczą.

   iii. Oszczędzać zasoby systemowe, przez ponowne wykorzystanie istniejących obiektów. Konstruktor musi zwracać tylko nowe obiekty.

3. Observer
    a. Wzorzec **behawioralny**
    b. https://refactoring.guru/pl/design-patterns/observer
    c. https://github.com/iluwatar/java-design-patterns/tree/master/observer
    d. Publikujący powiadamia subskrybentów o swoich zmianach stanu.
    e. Klasa publikującego ma mechanizm subskrypcji. Obiekty mogą subskrybować. Za każdym razem gdy się coś zdarzy, publikujący przegląda listę subskrybentów i powiadamia ich.
    f. Subskrybenci implementują ten sam interfejs i publikujący komunikuje się z nimi przez niego.
    g. Schemat

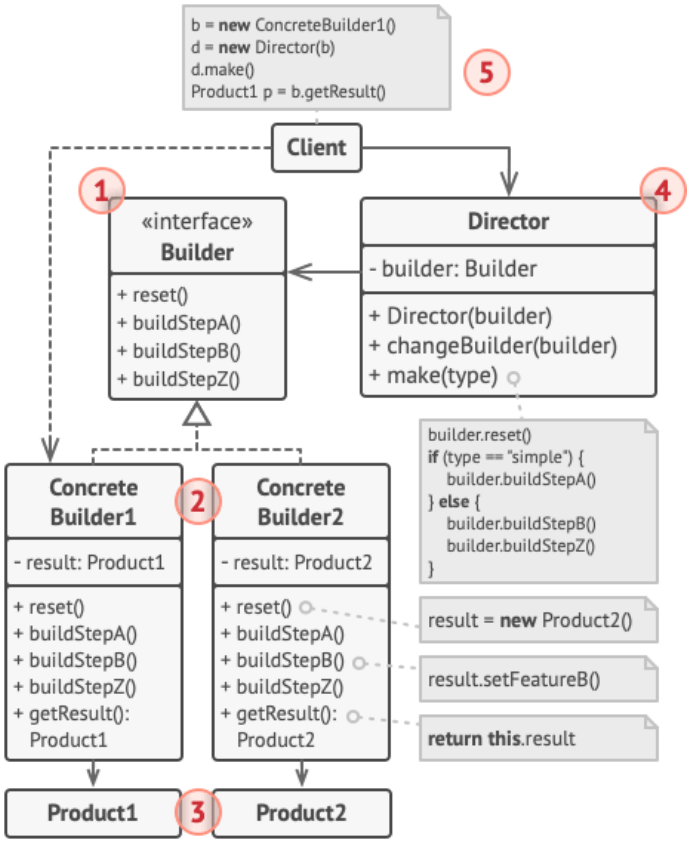


    h. (1) Publikujący rozsyła zdarzenia interesujące inne obiekty. Zachodzą one gdy zmienia swój stan. Posiada on infrastrukturę dającą możliwość subskrybowania go.
    i. (2) Gdy nastąpi nowe zdarzenie nadawca przegląda listę subskrybentów i wywołuje metody powiadamiania
    j. (3) Interfejs Subscriber to interfejs powiadamiania.
    k. (4) Konkretni Subskrybenci wykonują czynność w odpowiedzi na powiadomienie.
    l. (5) Klient tworzy obiekty publikujące i subskrybujące osobno, po czym rejestruje subskrybentów by mogli otrzymywać aktualizacje

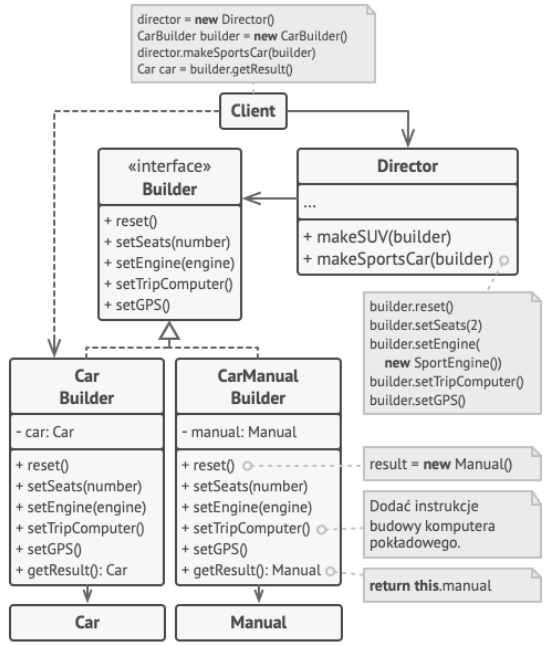


    m.
    n. Zastosowanie
        i. Użycie gdy zmiany w jednym obiekcie mogą spowodować zmiany w innych obiektach, zestaw tych obiektów jest dynamiczny.
        ii. Obiekty musza obserwować inne, ale tylko w określonych przypadkach

1. Budowniczy
    a. Wzorzec **kreacyjny**
    b. https://refactoring.guru/pl/design-patterns/builder
    c. https://github.com/iluwatar/java-design-patterns/tree/master/builder
    d. Tworzenie złożonych obiektów krok po kroku
    e. Tworzenie różnych typów i reprezentacji używając tego samego kodu
    f. Mamy duży obiekt, z wieloma polami, gdzie niektóre nie są użyte i ogromny konstruktor.
    g. Struktura

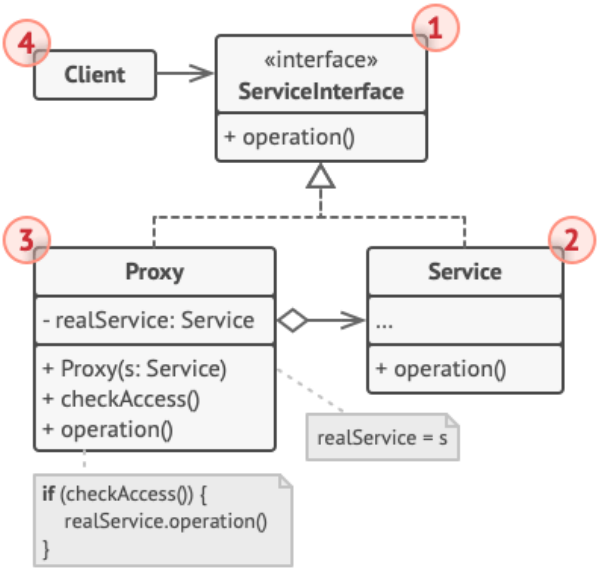


    h. (1) Interfejs Builder ma wspólne etapy konstrukcji.
    i. (2) Implementacje interfejsu Builder zapewniają implementacje etapów konstrukcji.
    j. (3) Produkty- powstałe obiekty, nie muszą mieć wspólnego interfejsu, **mogą być różnego typu**
    k. (4) Kierownik definiuje kolejność wykonywania etapów konstrukcji.
    l. (5) Klient dopasowuje budowniczego do kierownika. Kierownik za pomocą budowniczego wykonuje konstrukcję.

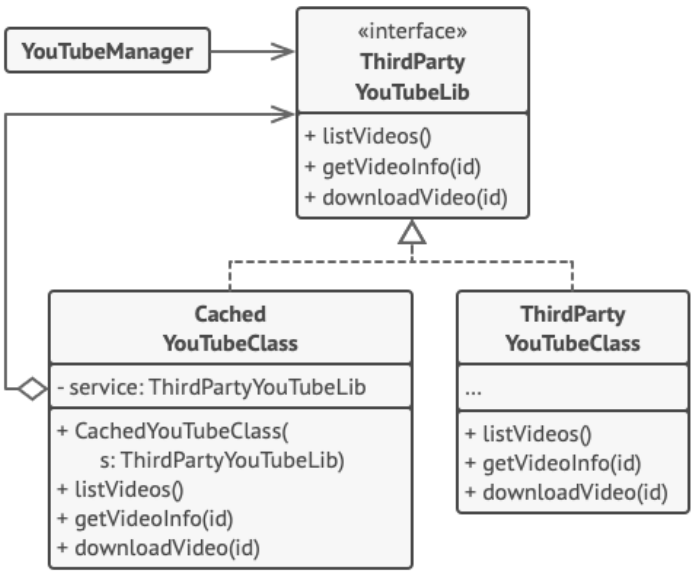
m. 

n. Zastosowanie
    i. Gdy mamy ogromne konstruktory których mamy kilka wersji, bo nie potrzebujemy wszystkich parametrów.
    ii. Kiedy trzeba stworzyć różne reprezentacje danego produktu. Mamy podobne etapy konstrukcji różniące się szczegółami.
    iii. Budowa rekurencyjna drzewa kompozytowego.
    iv. `var mage = new Hero.Builder(Profession.MAGE, "Riobard").withHairColor(HairColor.BLACK).withWeapon(Weapon.DAGGER).build();`

2. Proxy
    a. Wzorzec **strukturalny**
    b. https://refactoring.guru/pl/design-patterns/proxy
    c. https://github.com/iluwatar/java-design-patterns/tree/master/proxy
    d. Pozwala stworzyć obiekt zastępczy w miejsce innego obiektu. Proxy nadzoruje dostęp do pierwotnego obiektu, pozwalając na wykonanie czynności przed lub po przekazaniu żądania.
    e. Mamy duży obiekt, potrzebny raz na jakiś czas.
    f. Stworzenie klasy pośredniczącej, o tym samym interfejsie co pierwotny obiekt udostępniający usługę.
    g. Obiekt Proxy jest przekazywany wszystkim klientom pierwotnego obiektu
    h. Pełnomocnik otrzymawszy żądanie, tworzy prawdziwy obiekt usługi i przekazuje mu żądanie.
    i. Jeśli musisz uruchomić coś albo przed, albo po głównej logice klasy, pełnomocnik pozwala uczynić to bez modyfikowania tej klasy.
    j. Struktura



    k. (1) Interfejs usługi to interfejs z którym usługa i Proxy muszą być zgodni
    l. (2) Service ma logikę biznesową
    m. (3) Proxy zawiera pole z odniesieniem do usługi. Po wykonaniu zadań (leniwa inicjalizacja, zapis w dzienniku, kontrola dostępu, przechowanie w pamięci podręcznej) Proxy przekazuje żądanie do obiektu.
    n. (4) Klient może pracować z Service jak i Proxy za pomocą tego samego interfejsu. Proxy można umieścić w dowolnym kodzie działającym na obiekcie usługowym.
    o. Różnice między Proxy a dekoratorem:
        i. **A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.**
        ii. **A Decorator is always passed its delegatee. A Proxy might create it himself, or he might have it injected.**



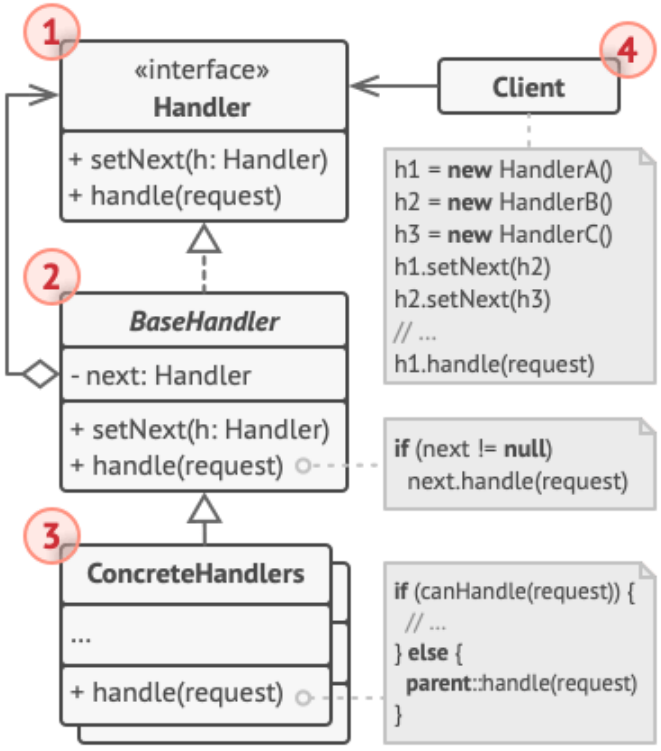
p.

q. Biblioteka po otrzymaniu żądania pobrania drugi raz tego samego filmu pobierze go od nowa, zamiast buforować. Klasa Cached-Proxy zapamiętuje pliki i zwraca z pamięci podręcznej jeżeli otrzyma żądanie pobrania drugi raz.

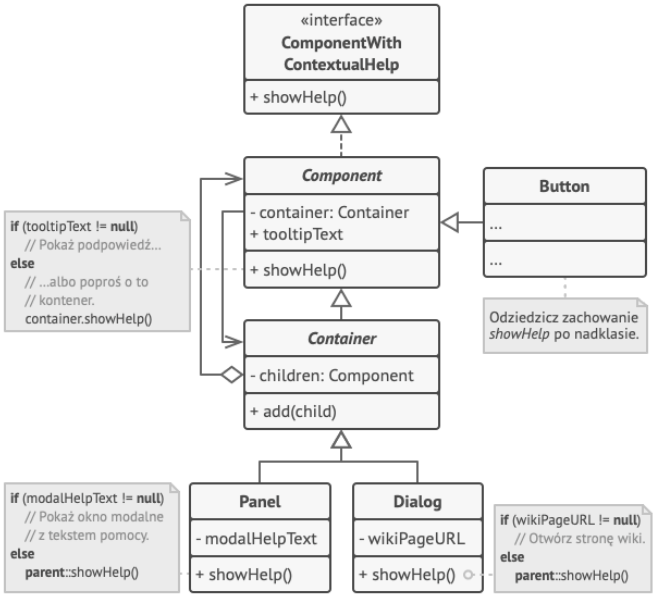
r. Zastosowanie
   i. Leniwa inicjalizacja. Mamy do czynienia z zasobożernym obiektem usługi, którego potrzeba co jakiś czas.
   ii. Kontrola dostępu- tylko niektórzy klienci mogą korzystać z usługi.
   iii. Lokalne uruchamianie zdalnej usługi- obiekt udostępniający usługę jest na innym serwerze.
   iv. Prowadzenie dziennika żądań przesyłanych do usługi.
   v. Przechowywanie w pamięci podręcznej wyników działań.
   vi. Śledzenie klientów mających referencję do obiektu

1. Chain of responsibility
    a. Wzorzec **behawioralny**
    b. https://refactoring.guru/pl/design-patterns/chain-of-responsibility
    c. https://github.com/iluwatar/java-design-patterns/tree/master/chain
    d. Przekazywanie żądań, każdy obiekt decyduje o przetworzeniu żądania lub przekazaniu go dalej
    e. Chcemy zrobić jakiś łańcuch autoryzacji
    f. Przekształcamy obowiązki w samodzielne obiekty obsługujące. Każde sprawdzenie powinno być osobną klasą mającą metodę sprawdzającą. Żądanie przekazywane jest dalej po łańcuchu lub obiekt decyduje o nie przekazaniu żądania dalej i zakończeniu procesu.

    g. Struktura



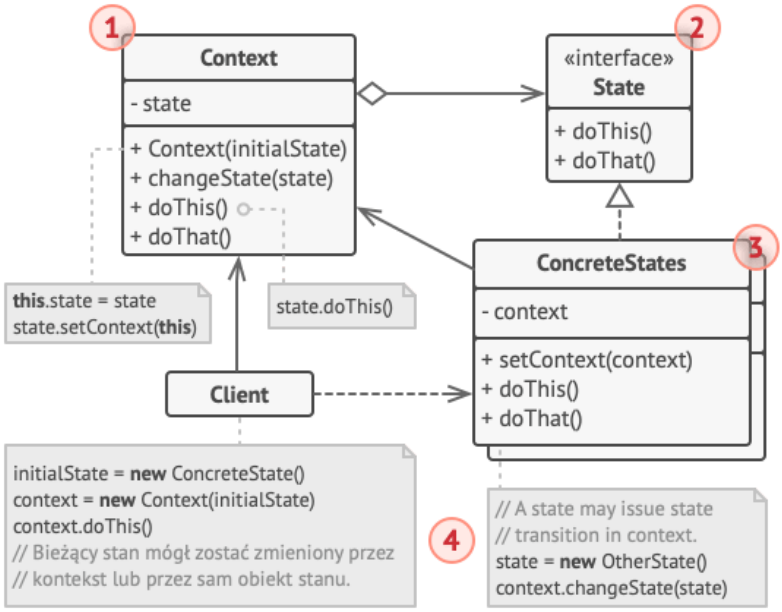
    h. (1) Mamy interfejs Handler, wspólny dla wszystkich obiektów obsługujących. Ma on metodę przetwarzania żądania.
    i. (2) Bazowy Handler zawiera odniesienie do następnego obiektu obsługującego. Jest on wspólny dla wszystkich klas obsługujących.
    j. (3) Konkretny Handler zawiera kod obsługi żądań. Musi zadecydować czy obsłużyć żądanie, czy przekazać je dalej.
    k. (4) Klient komponuje łańcuch. Żądanie można przekazać dowolnemu ogniwu łańcucha.



l.

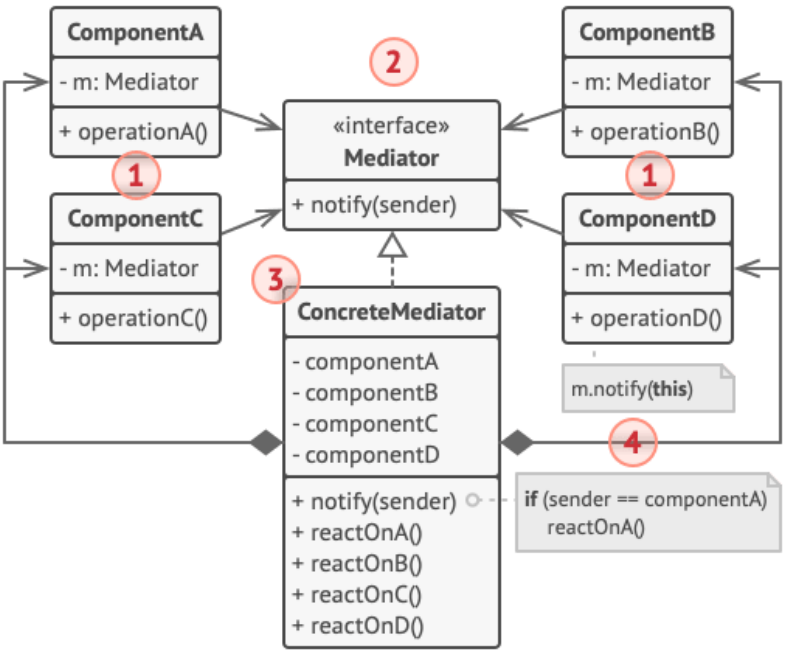
m. Zastosowanie:
    i. Program ma obsługiwać różne rodzaje żądań na różne sposoby, dokładny typ i sekwencja nie są zdane
    ii. Chcemy uruchomić wiele obiektów obsługujących w pewnej kolejności.
    iii. Chcemy ustawić obiekty obsługujące i ich kolejność w czasie działania programu

2. State
    a. **Behawioralny-** zmiana zachowania w zależności od stanu
    b. https://refactoring.guru/pl/design-patterns/state
    c. https://github.com/iluwatar/java-design-patterns/tree/master/state
    d. Zmiana zachowania obiektu w zależności od jego stanu wewnętrznego. Kiedy mamy automat skończony- istnieje skończona liczba stanów w jakim obiekt się może znajdować.
    e. Używając tradycyjnego podejścia, gdy dodamy jakiś stan będziemy mieć duże ify
    f. Struktura



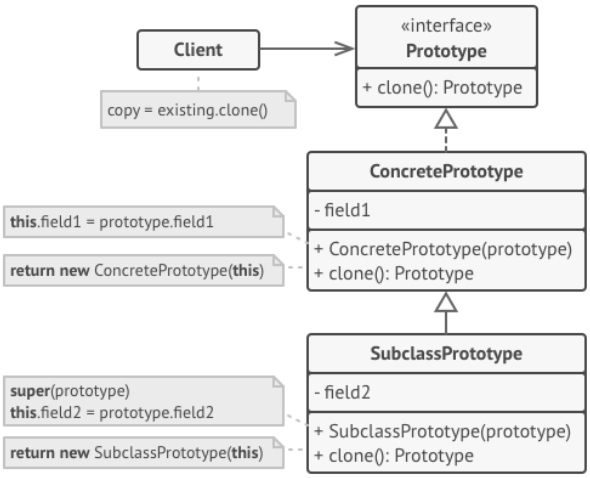
    g. (1) Kontekst ma odniesienie do stanów i deleguje mu zadania- wywołania metod.
    h. (2) Interfejs stanu ma metody specyficzne dla stanu
    i. (3) Konkretne stany mają implementację metod dla poszczególnych stanów. Można użyć klas abstrakcyjnych.
    j. (4) Kontekst jak i stany mogą ustawiać kolejny stan i wykonywać zmianę stanu.
    k. Poszczególne stany mogą być siebie świadome i inicjować przejścia z jednego stanu w drugi, w przeciwieństwie do strategii, które nie wiedzą o sobie.
    l. Zastosowanie:
        i. Użyć gdy zachowanie obiektu zależy od jego stanu, mamy dużo tych stanów
        ii. Duże ify zmieniające zachowanie klasy w zależności od jej pól
        iii. Kiedy mamy dużo powtarzającego się kodu w wielu stanach, używanie przejść między stanami za pomocą ifów

3. Mediator
    a. Wzorzec **behawioralny**- redukuje chaos zależności między obiektami. Zmusza je do współpracy przez mediatora.
    b. https://refactoring.guru/pl/design-patterns/mediator
    c. https://github.com/iluwatar/java-design-patterns/tree/master/mediator
    d. Przerwanie bezpośredniej komunikacji między komponentami które mają być niezależne. Współpracują one pośrednio, wywołując mediator który przekierowuje wywołania do komponentów. Komponenty zależą od Mediatora, nie są sprzężone ze sobą nawzajem.
    e. Struktura



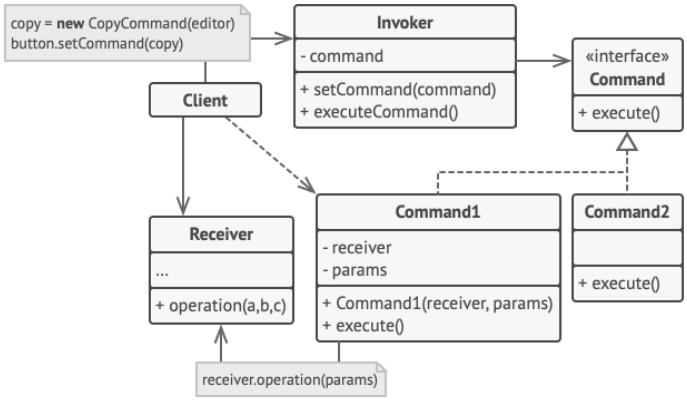
    f. (1) Komponenty to klasy zawierające różną logikę biznesową. Każdy komponent ma odniesienie do mediatora za pomocą jego interfejsu.
    g. (2) Interfejs mediator ma metody komunikacji z komponentami, jest to metoda powiadamiania. Komponenty przekazują kontekst jako argument metody
    h. (3) Konkretni mediatorzy hermetyzują relacje pomiędzy komponentami. Często mają odniesienia do komponentów którymi zarządzają.
    i. (4) Komponenty nie mogą być świadome istnienia innych komponentów. Komponent może powiadamiać tylko mediatora. Mediator identyfikuje nadawcę i wie jaki komponent uruchomić w odpowiedzi.
    j. Zastosowanie
        i. Gdy zmiana klas jest trudna z powodu sprzęgnięcia między klasami
        ii. Gdy nie można użyć ponownie komponentu, bo jest zależny od innych komponentów
        iii. Gdy jest tworzone dużo podklas, by móc ponownie użyć zachowanie w innych kontekstach

1. Prototyp
    a. Wzorzec **kreacyjny**
    b. https://refactoring.guru/pl/design-patterns/prototype
    c. https://github.com/iluwatar/java-design-patterns/tree/master/prototype
    d. Kopiowanie istniejących obiektów
    e. Pola prywatne nie są ogólnodostępne
    f. Musimy wiedzieć jaką klasę chcemy skopiować
    g. Proces kopiowania oddany obiektom które chcemy sklonować- dodanie metody klonuj dla obiektu
    h. Struktura



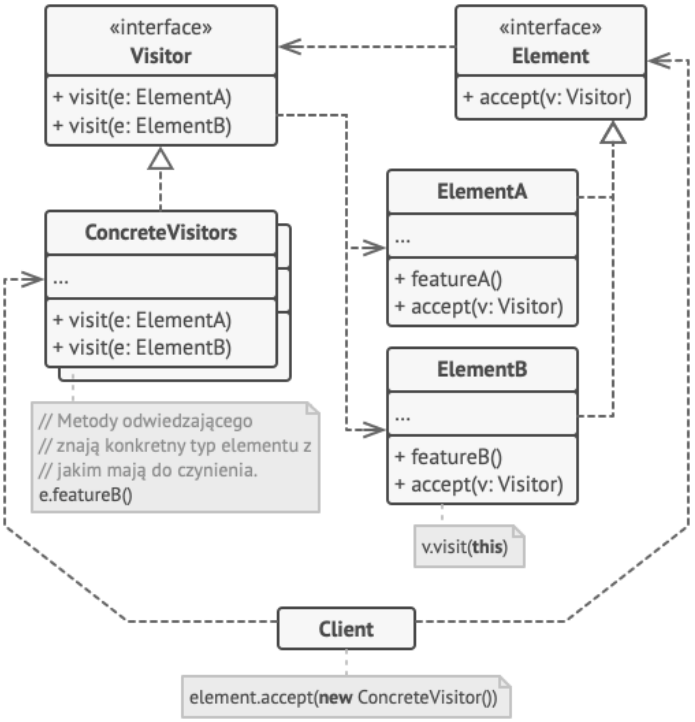
    i. (1) Interfejs Prototype ma metodę clone()
    j. (2) Klasa ConcretePrototype implementuje metodę clone, gdzie kopiuje dane do obiektu nowo powstałego, może klonować obiekty powiązane.
    k. (3) Klient tworzy kopię obiektu implementującego interfejs Prototype
    l. Zastosowanie:
        i. Używamy tego wzorca, gdy chcemy by nasz kod nie był zależny od konkretnej klasy kopiowanego obiektu. Wzorzec pozwala na klonowanie wszystkich obiektów implementujących interfejs.
        ii. By zredukować ilość podklas różniących się sposobem inicjalizacji obiektów. Mogą one być tworzone by utworzyć obiekt o danej konfiguracji. Zamiast tworzyć podklasę, klient może sklonować prototyp.

2. Command
    a. Wzorzec **behawioralny**
    b. https://refactoring.guru/pl/design-patterns/command
    c. https://github.com/iluwatar/java-design-patterns/tree/master/command
    d. Zmienia żądanie w samodzielny obiekt zawierające wszystkie informacje żądaniu. Pozwala na parametryzowanie metod przy użyciu wielu żądań. Umożliwia /kolejkowanie wykonywania żądań.
    e. Obiekty nie powinny wysyłać żądań bezpośrednio. Zamiast tego szczegóły żądania powinny się zawierać w klasie command.
    f. Schemat



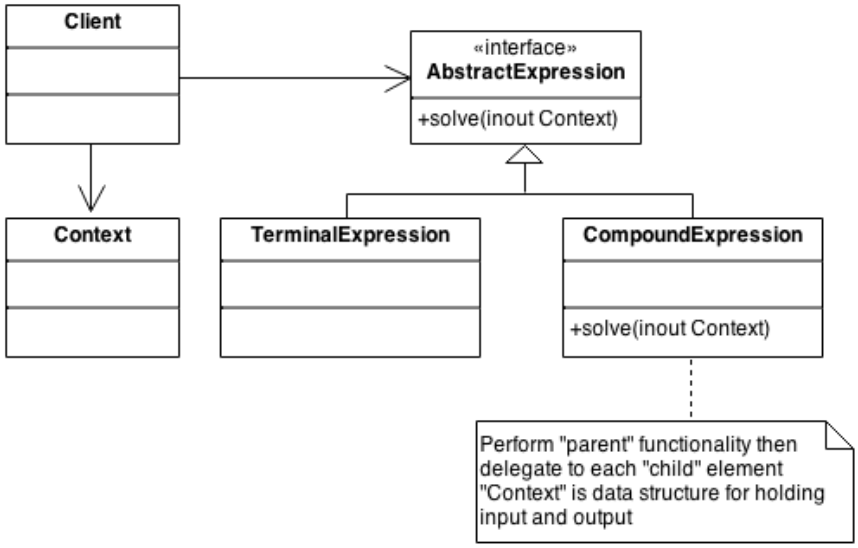
    g. (1) klasa Invoker inicjalizuje zadania. Ma ono pole obiektu polecenia. Invoker uruchamia polecenie
    h. (2) Interfejs Command deklaruje metodę wykonania polecenia
    i. (3) Konkretne polecenia implementują rodzaje żądań. Nie wykonuje pracy samodzielnie lecz przekazuje je do logiki biznesowej. Parametry to pola w klasie polecenie.
    j. (4) Klasa Receiver zawiera logikę biznesową. Command obsługuje szczegóły przekazania żądania do obiorcy, pracę wykonuje Receiver.
    k. (5) Klient tworzy i konfiguruje obiekty żądań. Klient przekazuje wszystkie parametry żądania wraz z instancją odbiorcy do konstruktora polecenia.
    l. Zastosowanie:
        i. Parametryzowanie obiektów za pomocą działań. Przekazywanie polecenia jako argumentów metody, przechwycenia ich.
        ii. Układanie kolejki zadań, ustawiania ich harmonogramu
        iii. Implementacja operacji odwracalnych. Historia będzie stosem zawierającym obiekty wykonanych poleceń wraz z kopiami stanu aplikacji.

1. Visitor
    a. Wzorzec **behawioralny**
    b. https://refactoring.guru/pl/design-patterns/visitor
    c. https://github.com/iluwatar/java-design-patterns/tree/master/visitor
    d. Oddzielenie algorytmów od obiektów na których pracują
    e. Umieszczenie nowych obowiązków w nowej klasie- odwiedzający. Pierwotny obiekt przekazywany jest jako argument metody.
    f. Wywołujemy odpowiednią metodę bez użycia instrukcji warunkowych. Zamiast pozwolić klientowi wybrać odpowiednią wersję metody, delegujemy ten wybór obiektom przekazywanym jako argument.
    g. Struktura



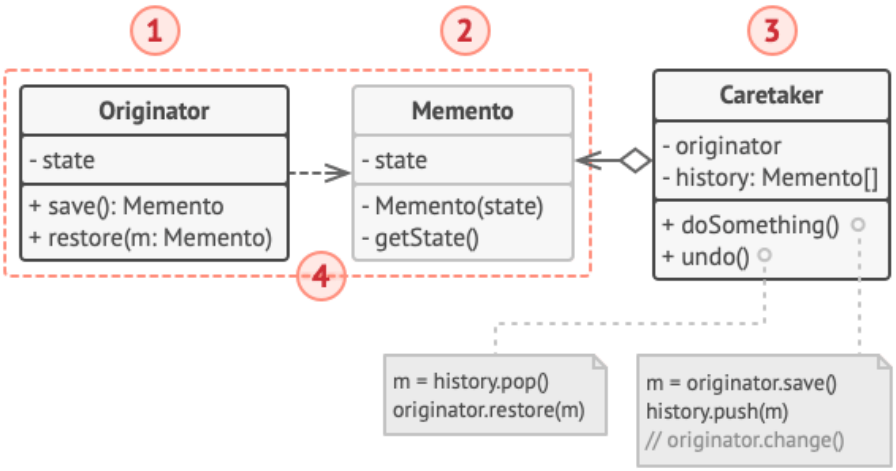
    h. (1) Interfejs Visitor deklaruje zestaw metod odwiedzania, które przyjmują jako argument różne typy obiektów.
    i. (2) Konkretny odwiedzający implementuje wiele wersji tego samego zachowania, dostosowanych do różnych konkretnych klas.
    j. (3) Interfejs Element deklaruje metodę przyjmowania odwiedzających. Parametrem jest interfejs odwiedzającego.
    k. (4) Konkretny element implementuje metodę przyjmowania. Przekierowuje wywołania do właściwej metody odwiedzającego (visitora), odpowiadającej klasie konkretnego elementu.
    l. (5) Klient to kolekcja lub złożony obiekt. Klienci nie są świadomi wszystkich konkretnych klas swoich elementów, współpracują z nimi na podstawie interfejsu.
    m. Zastosowanie:
        i. Do wykonania jakiegoś działania na wszystkich elementach złożonej struktury obiektów (na drzewie obiektów)
        ii. Do uprzątnięcia logiki biznesowej czynności pomocniczych- ekstrakcja pobocznych obowiązków klas do klas odwiedzających
        iii. Gdy zachowanie chcemy zastosować tylko dla niektórych klas w hierarchii- niektóre metody będą puste

2. Interpreter
    a. Wzorzec **behawioralny**
    b. https://sourcemaking.com/design_patterns/interpreter
    c. https://github.com/iluwatar/java-design-patterns/tree/master/interpreter
    d. Zdefiniowanie reprezentacji gramatyki języka oraz interpretera, który używa tej reprezentacji do interpretacji zdań tego języka.
    e. Posiadanie klasy dla każdego symbolu (kończącego lub niekończącego).
    f. Zdania języka są przedstawione jako drzewa składni.
    g. Interpreter używa klas do interpretacji zdań języka.
    h. Struktura



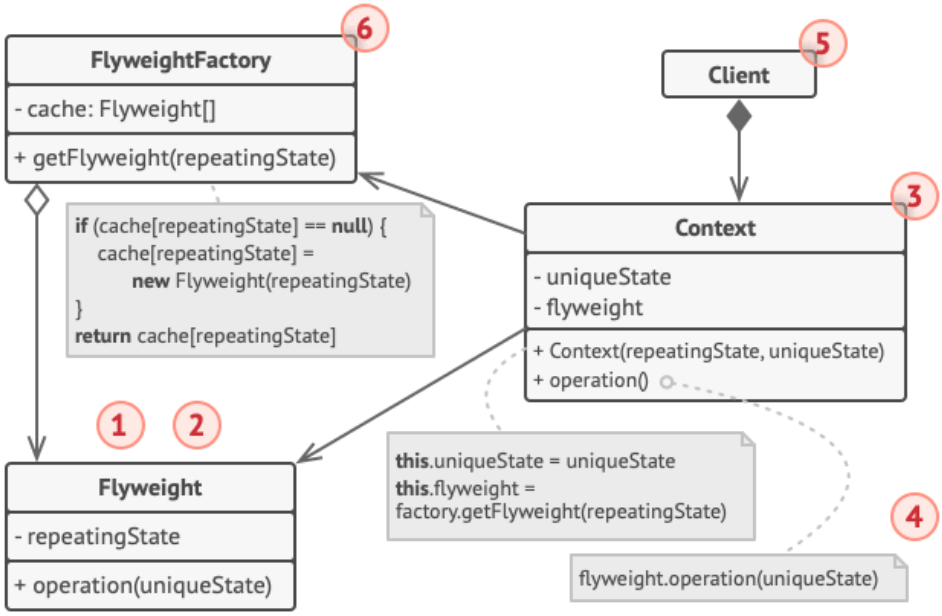
    i. (1) AbstractExpression-deklaruje metodę interpret (solve) która jest wspólna dla wszystkich węzłów.
    j. (2) TerminalExpression- ma operację interpret() oraz dla każdego symbolu terminalnego języka trzeba utworzyć egzemplarz tej klasy.
    k. (3) NonterminalExpression- wyrażenie złożone. Dla każdej produkcji wymagana jest odrębna klasa. Implementuje metodę interpret(), wywołując rekurencyjnie samą siebie na zmiennych.
    l. (4) Client- tworzy drzewo składni reprezentujące zdanie, wywołuje operację interpret()
    m. Zastosowanie:
        i. Interpreter pozwala na modelowanie domeny używając rekursywnej gramatyki. Każda reguła gramatyki jest kompozytem lub terminalem (liściem). Interpreter używa rekursywnego przejścia po drzewie używając wzorca kompozytu do interpretacji zdań, które ma przeprocesować.

3. Memento
    a. Wzorzec **behawioralny**
    b. https://refactoring.guru/pl/design-patterns/memento
    c. https://github.com/iluwatar/java-design-patterns/tree/master/memento
    d. Pozwala zapisywać i przywracać wcześniejszy stan obiektu, bez ujawnienia szczegółów jego implementacji
    e. Do tworzenia migawek stanu obiektu i przywracania poprzedniego stanu
    f. Wzorzec Memento deklaruje tworzenie migawki samemu obiektowi.
    g. Kopia stanu przechowywana jest w klasie Memento. Zawartość snapshota dostępna jest tylko jego twórcy.
    h. Memento są przechowywane w obrębie innych obiektów- zarządców. Zarządca ma dostęp tylko do metadanych migawki.
    i. Struktura



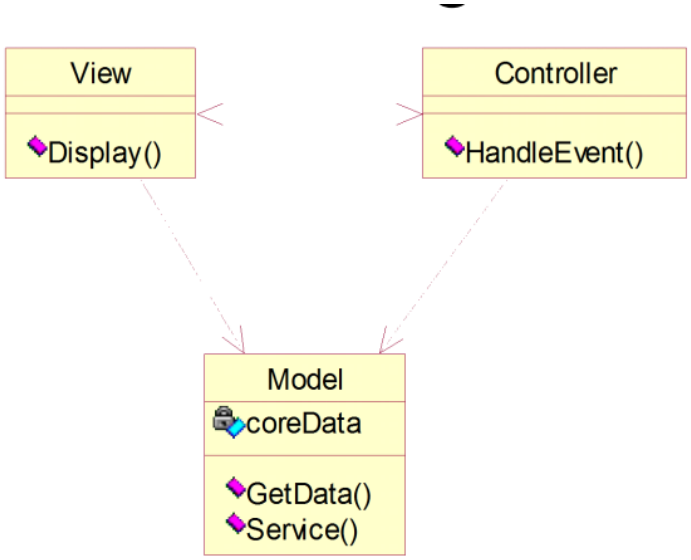
    j. (1) Klasa Origin (źródło) tworzy migawki swojego stanu, może przywracać wcześniejszy stan z migawek.
    k. (2) Memento to obiekt będący pamiątką stanu źródła. Memento jest niezmieniane, tworzone tylko przez konstruktor.
    l. (3) Caretaker (zarządca) wie kiedy rejestrować stan Originatora, ale również przywrócić poprzedni stan. Zarządca trzyma stos pamiątek (Memento). Gdy cofamy operację, zarządca pobiera pamiątkę ze stosu i przekazuje ją metodzie przywracającej klasy Originator.
    m. (4) Klasa Memento jest zagnieżdżona wewnątrz klasy Originator (źródło). Pozwala to źródłu mieć dostęp do pól pamiątki, mimo że są prywatne. Zarządca ma ograniczony dostęp do pól pamiątki, może je przechowywać ale nie może zmieniać ich stanu.
    n. Zastosowanie:
        i. Stosujemy kiedy chcemy stworzyć migawkę stanu obiektu i móc przywracać jego poprzedni stan.
        ii. Kiedy dostęp do pól getterów/ seterów obiektu psuje hermetyzację. Obiekt tworzy migawki swojego stanu, inny obiekt nie ma prawa odczytać migawki, co ją zabezpiecza.

4. Flyweight
    a. Wzorzec **strukturalny**
    b. https://refactoring.guru/pl/design-patterns/flyweight
    c. https://github.com/iluwatar/java-design-patterns/tree/master/flyweight
    d. Zmieszczenie większej ilości obiektów w RAM przez współdzielenie części opisu ich stanów.
    e. Stan wewnętrzny to dane niezmienne, opisujące obiekt, wspólne dla wszystkich obiektów.
    f. Stan zewnętrzny- dane ulegające ciągłej zmianie, jest to zmieniający się kontekst. Stan zewnętrzny nie jest przechowywany w obiekcie. Zamiast tego jest przekazywany konkretnym metodom które go potrzebują.
    g. W obrębie obiektu znajduje się stan wewnętrzny, pozwalając na użycie go w innych kontekstach. Dzięki temu potrzebujemy mniej obiektów, bo różnią się tylko pod względem wewnętrznego stanu.
    h. Struktura



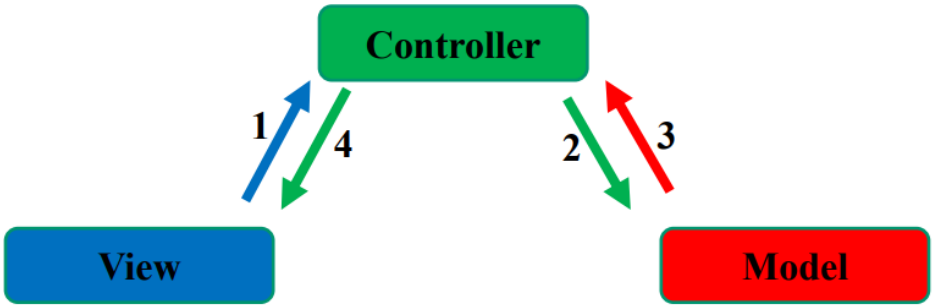
    i. (1) wzorzec ten jest optymalizacją. Stosujemy go gdy tworzymy wielką ilość podobnych obiektów.
    j. (2) Klasa Flyweight zawiera stan wewnętrzny obiektu, który może być współdzielony między wieloma instancjami. Ten sam obiekt- pyłek może być wykorzystany w wielu kontekstach. Stan przekazywany metodom pyłka to zmieniający się stan zewnętrzny.
    k. (3) Klasa Context zawiera opis zewnętrznego stanu, unikalny dla każdego z pierwotnych obiektów (kierunek wystrzelenia naboju). Kontekst kojarzy opis z pyłkiem, otrzymując pełną reprezentację obiektu.
    l. (4) Obowiązki pierwotnego obiektu pozostają w klasie Flyweight- pyłek. W przypadku wywołania metody pyłka, trzeba przekazać mu stan zewnętrzny.
    m. (5) Klient przechowuje stan zewnętrzny pyłków. Z jego punktu widzenia, pyłek to szablon konfigurowany przez przekazanie danych zewnętrznych.
    n. (6) Fabryka pyłków zarządza pulą istniejących pyłków. Klienci nie tworzą nowych pyłków bezpośrednio, lecz odwołują się do fabryki. Ta sprawdza czy taki stan wewnętrzny istnieje, jeżeli nie to go tworzy lub zwraca już istniejący.
    o. Zastosowanie:
        i. Duża ilość obiektów, bardzo pamięciożernych

1. **Wzorce architektoniczne**:
2. MVC
    a. Model- View- Controller
    b. Podział aplikacji na 3 komponenty
    c. Brak zależności modelu od widoków
    d. Łatwa rozbudowa widoków
    e. Model
        i. zawiera logikę biznesową
        ii. może rejestrować widoki i kontrolery
    f. Widok
        i. prezentuje informacje użytkownikowi
        ii. tworzy swój kontroler
        iii. pobiera dane z modelu
    g. Kontroler
        i. Pobiera wejście od użytkownika
        ii. Wywołuje operacje modelu lub zleca wyświetlanie widoków
    h. Struktura:

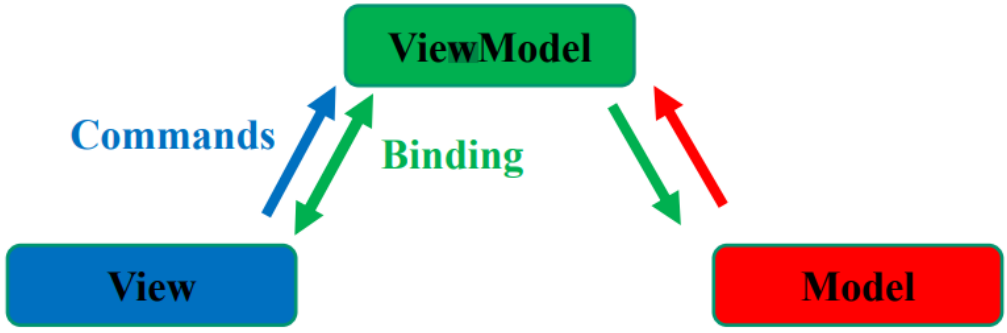


    i. Zastosowanie:
        i. Ta sama informacja jest prezentowa w różnych elementach UI
        ii. Zmiana kodu UI nie narusza kodu aplikacji

3. MVP
    a. Model- View- Presenter
    b. Kontroler staje się prezenterem
    c. Model ma podstawowe operacje na danych
    d. Prezenter odpowiada za komunikację między widokiem a modelem. Prezenter pobiera dane z modelu, przetwarza je i przesyła do widoku.
    e. Struktura



4. MVVM
    a. Model- View- ViewModel
    b. ViewModel jest odpowiedzialny za reagowanie na interakcję użytkownika, zarządzanie stanem widoku oraz konwersję danych. Nie ma zależności między View a ModelView
    c. Struktura



    d. Model- logika biznesowa
    e. Controller/ Presenter/ ViewModel- reakcja na interakcję z użytkownikiem
    f. View- interfejs użytkownika
    g. W MVC interakcja przekazywana bezpośrednio do Controller
    h. W MVP, MVVM interakcja jest przechwytywana przez View
    i. MVVP- stan widoku jest przechowywany w Controller (z wykładu, jest wgl Controller w MVVP? XD)

Klasyfikacja wzorców

		RODZAJ		
		KONSTRUKCYJNE	**STRUKTURALNE**	**OPERACYJNE/CZYNNOŚCIOWE**
ZASIĘG	**KLASOWE**	FACTORY METHOD	ADAPTER (KLASOWY)	INTERPRETER TEMPLATE METHOD
	OBIEKTOWE	ABSTRACT FACTORY BUILDER PROTOTYPE SINGLETON	ADAPTER (OBIEKTOWY) BRIDGE COMPOSITE DECORATOR FACADE FLYWEIGHT PROXY	CHAIN OF RESPONSIBILITY COMMAND ITERATOR MEDIATOR MEMENTO OBSERVER STATE STRATEGY VISITOR
	ARCHITEKTONICZNE	MVC MVP MVVM Layers		