Jak dobierać i stosować wzorce projektowe

Tomasz Sadza
Tech Lead @ Sii

Czym są wzorce projektowe

- **sprawdzone rozwiązania projektowe** dla typowych problemów
- **przemyślane wytyczne** jak podejść do określonego problemu
- **efektywne w użyciu szablony** rozwiązań

Jakie znaczenie mają wzorce projektowe

- **ułatwiają komunikację** w dyskusji o projektach i architekturze
- **oszczędzają czas**, stanowiąc wcześniej opracowane metody rozwiązań
- **stanowią źródło wiedzy** jak tworzyć skalowalne i łatwe w utrzymaniu oprogramowanie

JAK PODZIELIĆ WZORCE

KREACYJNE / KONSTRUKCYJNE

Wzorce konstrukcyjne koncentrują się na **tworzeniu obiektów w sposób, który jest odpowiedni dla danego kontekstu i celu**. Te wzorce pomagają abstrahować proces tworzenia obiektów, ukrywając jego złożoność oraz umożliwiając większą elastyczność i kontrolę nad tworzonymi obiektami.

STRUKTURALNE

Wzorce strukturalne koncentrują się na organizacji klas i obiektów w celu stworzenia większych struktur. Pozwalają na elastyczne **tworzenie złożonych struktur** i ułatwiają **zarządzanie relacjami między obiektami** oraz klasami.

BEHAWIORALNE

Behawioralne wzorce projektowe **koncentrują się na sposobach efektywnej komunikacji** między obiektami w systemie.
Pozwalają na **efektywne zarządzanie interakcjami** między obiektami oraz elastyczne **rozwiązywanie problemów komunikacji i koordynacji działań**.

WZORCE KREACYJNE / KONSTRUKCYJNE

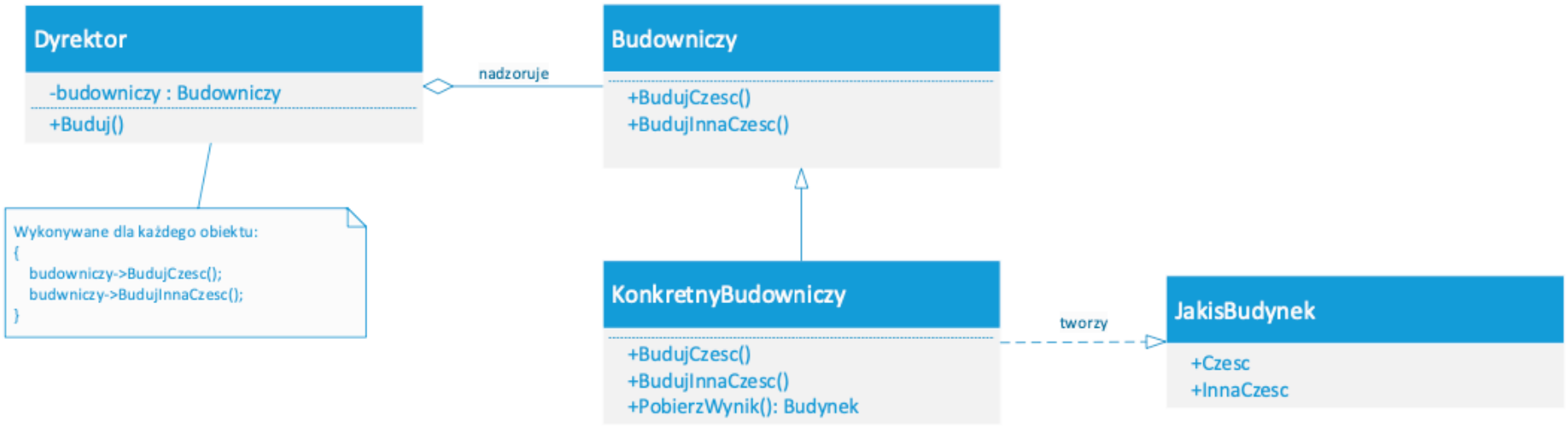
Dotyczą sposobów tworzenia obiektów
i skupiają się na efektywnym i elastycznym tworzeniu instancji klas,
zwiększają elastyczność systemu.

- **Chcesz izolować kod od informacji o tym, jakie konkretne klasy tworzy**
- **Chcesz ukryć skomplikowane procesy tworzenia i reprezentować cały proces jako pojedynczą operację**
- **Chcesz system, który jest niezależny od tego, jak jego obiekty są tworzone, komponowane i reprezentowane.**

WZORCE KREACYJNE / KONSTRUKCYJNE

BUILDER

- umożliwia hermetyzowanie wielu operacji tworzenia złożonego obiektu
- stosowany w przypadkach dużej listy argumentów
- oddziela proces konstrukcji od samego obiektu



Schemat kroków do zastosowania wzorca BUILDER:

1. Zdefiniuj interfejs **Builder**, który będzie zawierał metody do konfiguracji różnych składników i opcji obiektu.
2. Utwórz klasę **ConcreteBuilder**, która implementuje interfejs Builder. Ta klasa będzie odpowiedzialna za konfigurację obiektu, dodawanie składników i opcji.
3. Utwórz klasę **Director**, która zawiera metody do konstruowania obiektu przy użyciu klasy ConcreteBuilder. Klasa Director zarządza procesem tworzenia obiektu, korzystając z metod klasy ConcreteBuilder.

4. Utwórz klasę docelową, która reprezentuje docelowy złożony obiekt, powinna mieć metody do pobierania składników i opcji obiektu.

5. W kodzie klienta, użyj klasy Director i klasy ConcreteBuilder do konstruowania obiektu krok po kroku, korzystając z metod Buildera do konfiguracji składników i opcji.

Zastosowanie:

- generowanie różnych formatów dokumentów przez dodawanie różnych sekcji, tabel, wykresów, itp za pomocą tego samego procesu budowy

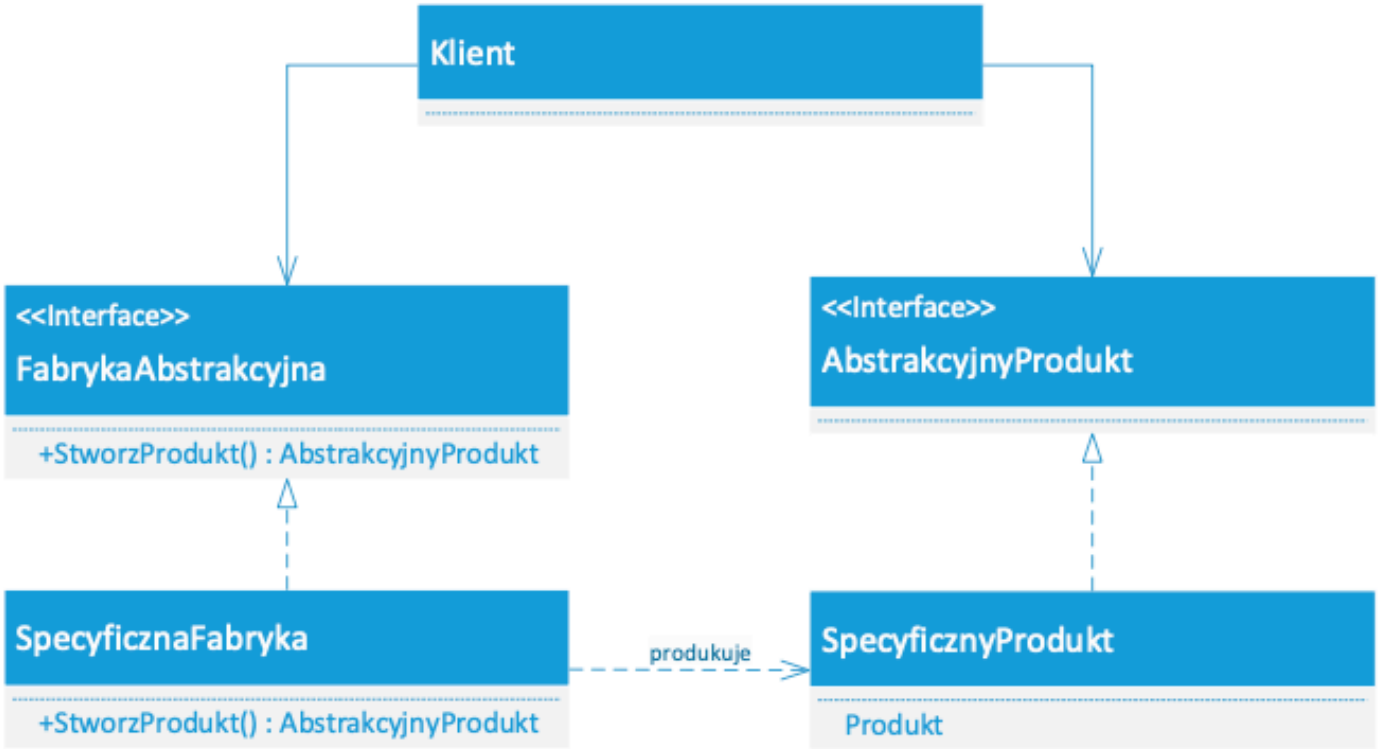
- konfiguracja obiektów z różnymi opcjami wyposażenia, np. spersonalizowany pojazd, lub spersonalizowany posiłek, lub zestaw komputerowy

- tworzenie złożonych zapytań SQL (opcjonalne warunki, sortowanie, grupowanie)

WZORCE KREACYJNE / KONSTRUKCYJNE

ABSTRACT FACTORY

- umożliwia tworzenie spójnych rodzin obiektów bez określania ich klas rzeczywistych
- gdy potrzebna jest abstrakcja nad procesem tworzenia różnych rodzajów produktów
- powinniśmy zdefiniować interfejs fabryki oraz konkretne fabryki tworzące różne typy zamówień



Kroki do zastosowania wzorca ABSTRACT FACTORY:

1. Zdefiniuj **interfejsy lub klasę abstrakcyjną** dla każdego typu produktu, który chcesz wytworzyć.
2. Utwórz **konkretne klasy** dla każdego typu produktu implementujące interfejsy tych typów.

3. Utwórz interfejs AbstractFactory, który zawiera metody do tworzenia każdego typu produktu.
4. Utwórz konkretne klasy fabryki dla każdego typu, implementując interfejs AbstractFactory. Ta klasa powinna zawierać logikę decyzyjną dotyczącą tego, którą konkretną implementację produktu utworzyć.

5. W swoim kodzie klienta użyj fabryk do tworzenia produktów. Kod powinien być niezależny od konkretnej implementacji fabryki. Jeśli chcesz zmienić rodziny produktów w kodzie klienta, to wystarczy zmienić typ konkretnej fabryki.

Zastosowanie:

- tworzenie różnych typów połączeń z bazami danych (MySQL, PostgreSQL, SQLite ..)
  bez konieczności zmiany kodu aplikacji

- generowanie mocków lub stubów dla różnych środowisk testowych,
  umożliwiając testowanie komponentów w izolacji

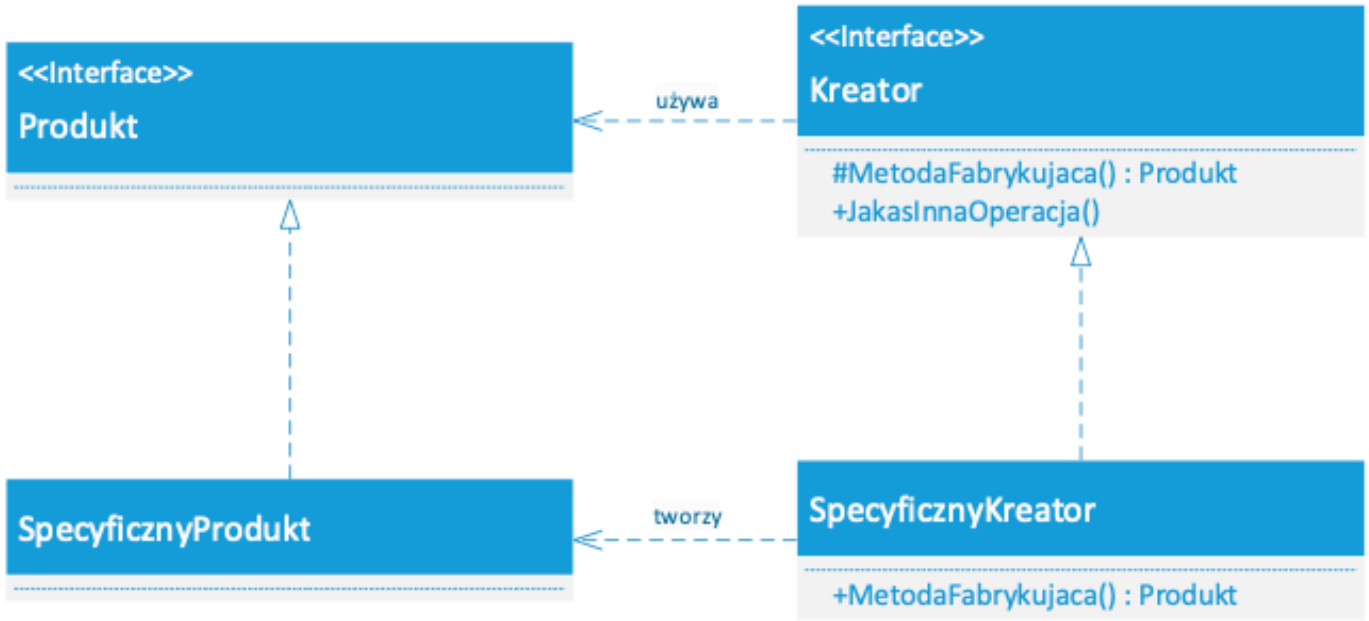
- generowanie różnych zasobów językowych (komunikaty, labelki, teksty, ..)
  w zależności od wybranego języka użytkownika

- tworzenie różnych zestawów produktów w jednolity sposób,
  ułatwiając zarządzanie różnorodnymi ofertami w e-commerce

WZORCE KREACYJNE / KONSTRUKCYJNE

FACTORY METHOD

- zakłada użycie metody fabrykującej, która jest zdefiniowana w abstrakcyjnej klasie lub interfejsie i implementowana przez konkretne klasy fabryczne
- klasy podrzędne decydują o tym, która klasa rzeczywista zostanie użyta do utworzenia obiektu
- stosujemy, gdy chcemy oddzielić kod konstruujący od korzystającego



Jak zastosować wzorzec FACTORY METHOD ?

1. Zdefiniuj **klasę bazową** (lub interfejs) dla wszystkich obiektów, które chcesz tworzyć. Klasa ta powinna określać interfejs, który będą musiały zaimplementować wszystkie konkretyzacje (tzw. "produkty").

2. Zdefiniuj klasy **dla każdego konkretnego typu obiektu**, które dziedziczą po klasie bazowej lub implementują zdefiniowany interfejs.

3. Zdefiniuj klasę "twórcy" (**Factory**), która ma metodę odpowiedzialną za tworzenie obiektów. Ta metoda powinna zwracać obiekty jako instancje klasy bazowej lub interfejsu, co pozwala na zmienność typu produktu bez wpływu na kod kliencki.

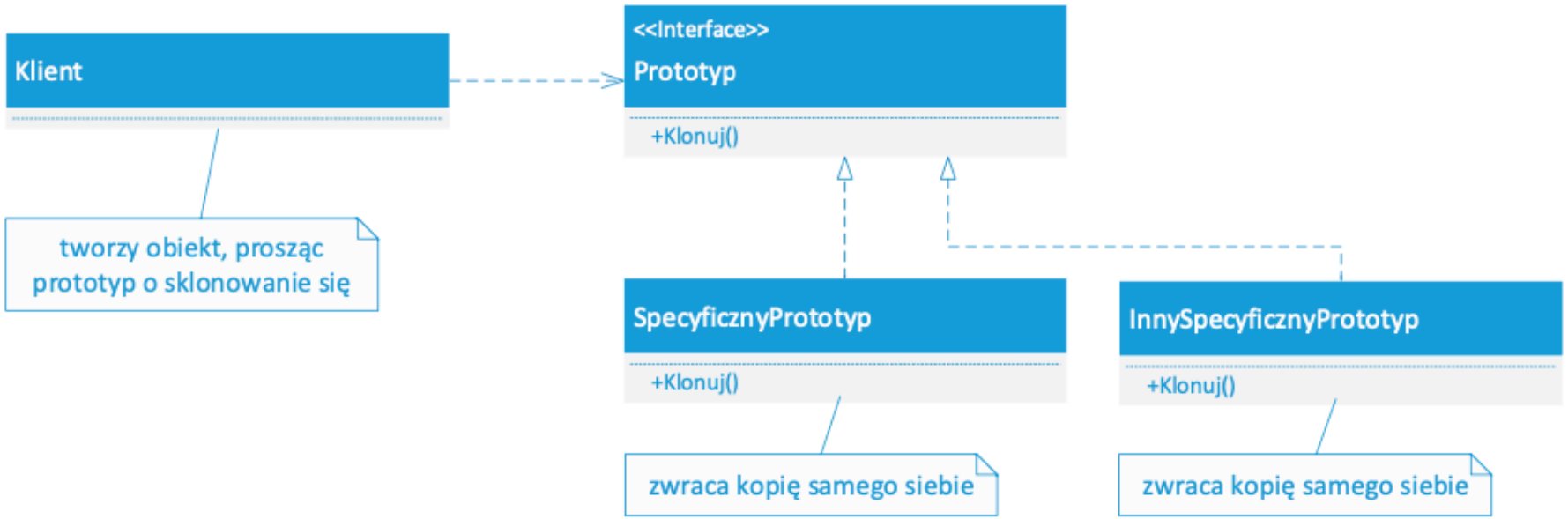
4. Używaj metody fabryki, aby tworzyć nowe obiekty, zamiast bezpośrednio korzystać z konstruktorów klas konkretnych. Dzięki temu kod kliencki jest odseparowany od konkretnych klas produktów i jest gotowy do obsługi nowych typów produktów, które mogą zostać wprowadzone w przyszłości.

Zastosowanie:

- tworzenie typów dokumentów w zależności od wybranego formatu
- tworzenie różnych typów obiektów, gdzie każdy typ może mieć inną logikę tworzenia
- w systemach obsługi logów, gdzie są różne mechanizmy zapisywania logów
- przy tworzeniu różnych widoków w zależności od danych lub kontekstu

PROTOTYPE

- gdy tworzenie egzemplarza danej klasy jest kosztowne, lub złożone
- umożliwia tworzenie nowych obiektów przez klonowanie instancji



Schemat kroków do zastosowania wzorca Prototype:

1. Stwórz interfejs lub abstrakcyjną klasę, która będzie zawierać **metodę klonowania** (np. metoda `clone`).
2. Implementuj interfejs lub dziedzicz po abstrakcyjnej klasie w klasie, której obiekty będą klonowane.

3. W implementacji metody klonowania, stwórz nowy obiekt i przekopiuj wartości wszystkich właściwości **z oryginalnego obiektu do nowego obiektu**.

4. W odpowiednich miejscach w kodzie, gdzie chcesz tworzyć nowe obiekty na podstawie istniejących, użyj metody klonowania na istniejącym obiekcie, aby utworzyć jego kopię.

Zastosowanie:

- tworzenie dokumentów lub ich fragmentów przez kopiowanie istniejących lub ich części

- klonowanie postaci, obiektów, poziomów gry

- tworzenie konfiguracji przez klonowanie istniejących

- tworzenie nowych złożonych produktów przez kopiowanie prototypów, szczególnie przydatne przy produkcjach seryjnych (batch processing)

WZORCE KREACYJNE / KONSTRUKCYJNE

SINGLETON

- zapewnia utworzenie dokładnie jednego egzemplarza obiektu (jedna instancja danej klasy)
- daje globalny punkt dostępowy do tej instancji

- stosuj gdy musi być tylko jeden obiekt danej klasy
- stosuj gdy potrzebujesz ścisłej kontroli nad zmiennymi globalnymi

Schemat kroków do zastosowania wzorca Singleton:

1. Zadeklaruj **konstruktor** klasy jako `private`, żeby zapobiec tworzeniu obiektów z zewnątrz.

2. Zadeklaruj statyczną zmienną, która przechowuje **instancję singleton**.

3. Udostępnij publiczną i statyczną metodę, która sprawdza, czy instancja singleton już istnieje, a jeśli nie - tworzy ją.

4. Zapewniaj dostęp do tej jednej instancji poprzez tę publiczną metodę.

WZORCE STRUKTURALNE

Dotyczą organizacji różnych elementów systemu w większe struktury.

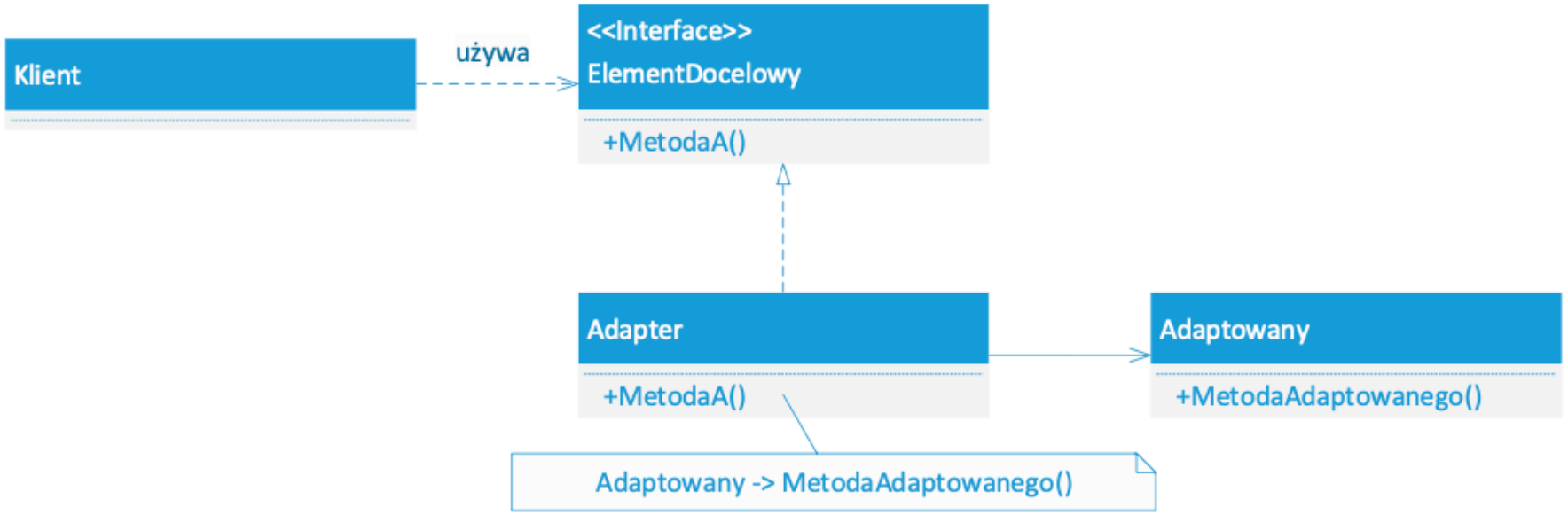
Koncentrują się na związku między obiektami i tym jak one są ze sobą powiązane.

- **Chcesz zapewnić, że pewne części systemu są zbudowane w określony sposób**
- **Chcesz, aby system był łatwo skalowalny**
- **Chcesz, aby system był elastyczny w tworzeniu i modyfikacji struktur**
- **Chcesz mieć kontrolę dostępu do pewnych części obiektu**

WZORCE STRUKTURALNE

ADAPTER

- opakowuje inny obiekt i udostępnia inny interfejs
- gdy chcemy użyć jakiegoś interfejsu, który jest niezgodny z interfejsem oczekiwanym przez naszą klasę. Pozwala to na integrację z nowymi systemami lub zastąpienie starych systemów, minimalizując wpływ na istniejący kod.



Schemat kroków do zastosowania wzorca Adapter:

1. Zidentyfikuj interfejs, który potrzebujesz (**InterfejsCelu**).
2. Zidentyfikuj interfejs, który jest dostępny (**InterfejsKlasyAdaptowanej**).

3. Stwórz klasę **Adapter** implementującą InterfejsCelu.
4. Wewnątrz klasy Adaptera dodaj referencję na obiekt **KlasyAdaptowanej**.

5. Wewnątrz klasy Adaptera zaimplementuj metody interfejsu celu tak, aby korzystały z metody/mi KlasyAdaptowanej.

6. Klient powinien używać obiektu Adaptera poprzez InterfejsCelu.

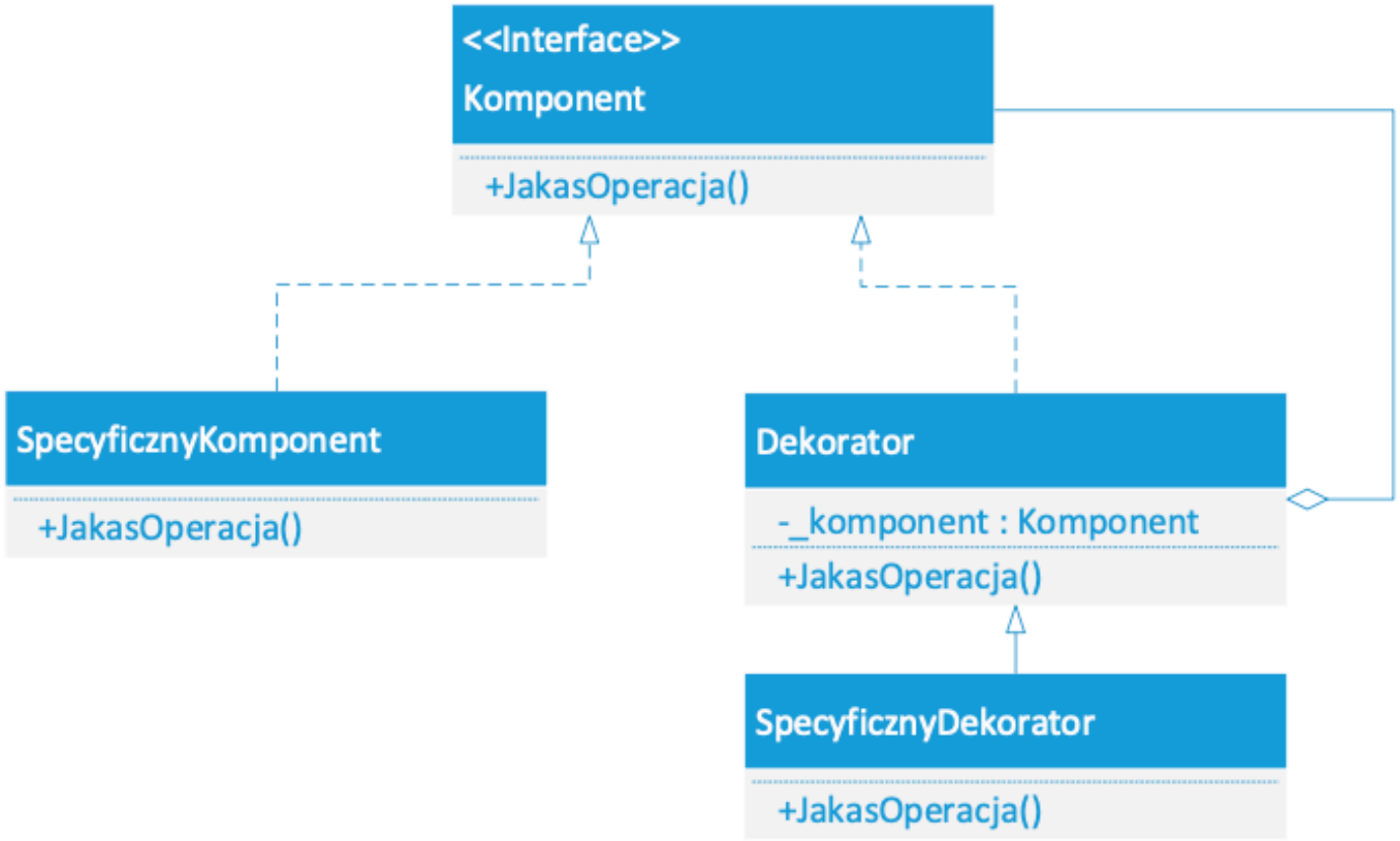
Zastosowanie:

- integracja nowych komponentów z kodem legacy (starsze libki)
- współpraca z zewnętrznymi usługami, które mają różne interfejsy
- adaptacja interfejsów bibliotek zewnętrznych
- łączenie różnych bramek płatności z różnymi interfejsami
- przekształcanie formatów danych w jednolity format
- adaptacja interfejsów różnych urządzeń peryferyjnych
- migracja danych między różnymi systemami bazodanowymi

WZORCE STRUKTURALNE

DECORATOR

- opakowuje inny obiekt i zapewnia dodatkowe zachowania
- stosujemy, gdy chcemy dodać nowe funkcjonalności do obiektów bez modyfikacji ich klas
- gdy potrzebujemy kombinacje dodatkowych funkcjonalności



Schemat kroków do zastosowania wzorca Dekorator:

1. Zdefiniuj **interfejs** komponentu, który określa zachowania, które mogą być rozszerzone przez dekoratory.

2. Zaimplementuj konkretny komponent, który oferuje podstawową funkcjonalność.

3. Zdefiniuj interfejs lub klasę bazową dekoratora, który zawiera referencję do obiektu komponentu.

4. Zaimplementuj jeden lub więcej konkretnych dekoratorów, dodając dodatkowe zachowania do obiektu komponentu.

5. Użyj dekoratorów w kodzie klienta, owijając komponenty w dekoratory w sposób dynamiczny, zależny od stanu programu lub preferencji użytkownika.

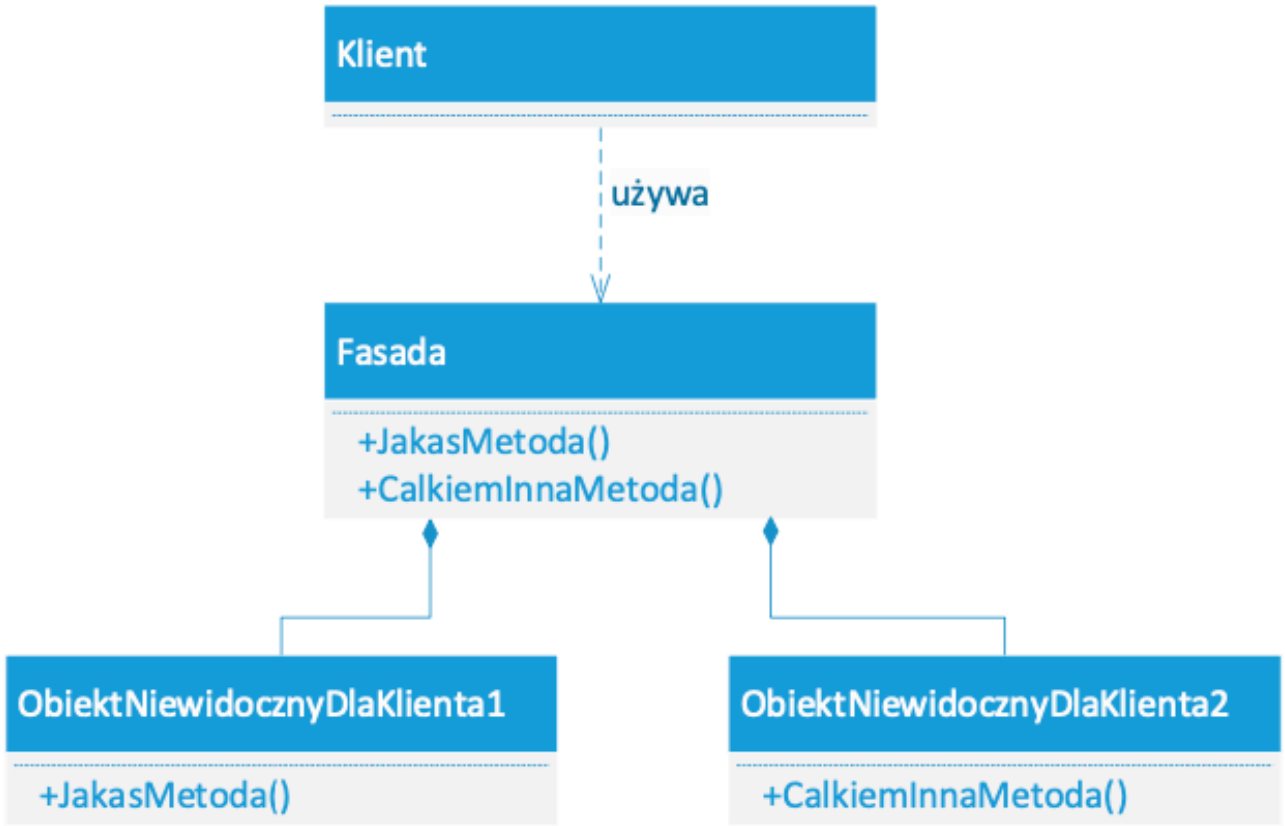
Zastosowanie:

- dynamiczne dodawanie logowania do obiektów, bez zmiany ich kodu
- dodawanie funkcji np. sortowania, filtrowania do kolekcji danych
- dodawanie formatowania do obiektów przetwarzających dane
- dawanie opcji dostawy, pakowania, promocji do produktów w e-commerce
- dodawanie funkcji szyfrowania, kompresji, autoryzacji

WZORCE STRUKTURALNE

FACADE

- upraszcza interfejs grupy klas (ukrycie złożoności)



Schemat kroków do zastosowania wzorca Facade:

1. Zidentyfikuj zestaw podsystemów, które są trudne do zrozumienia lub skomplikowane do używania.

2. Utwórz klasę "Facade", która zawiera metody wysokiego poziomu do interakcji z tymi podsystemami.

3. Implementuj metody "Facade" tak, aby przekazywały wywołania do odpowiednich metod w podsystemach.

4. Klient powinien używać tylko klasy "Facade" do interakcji z podsystemami.

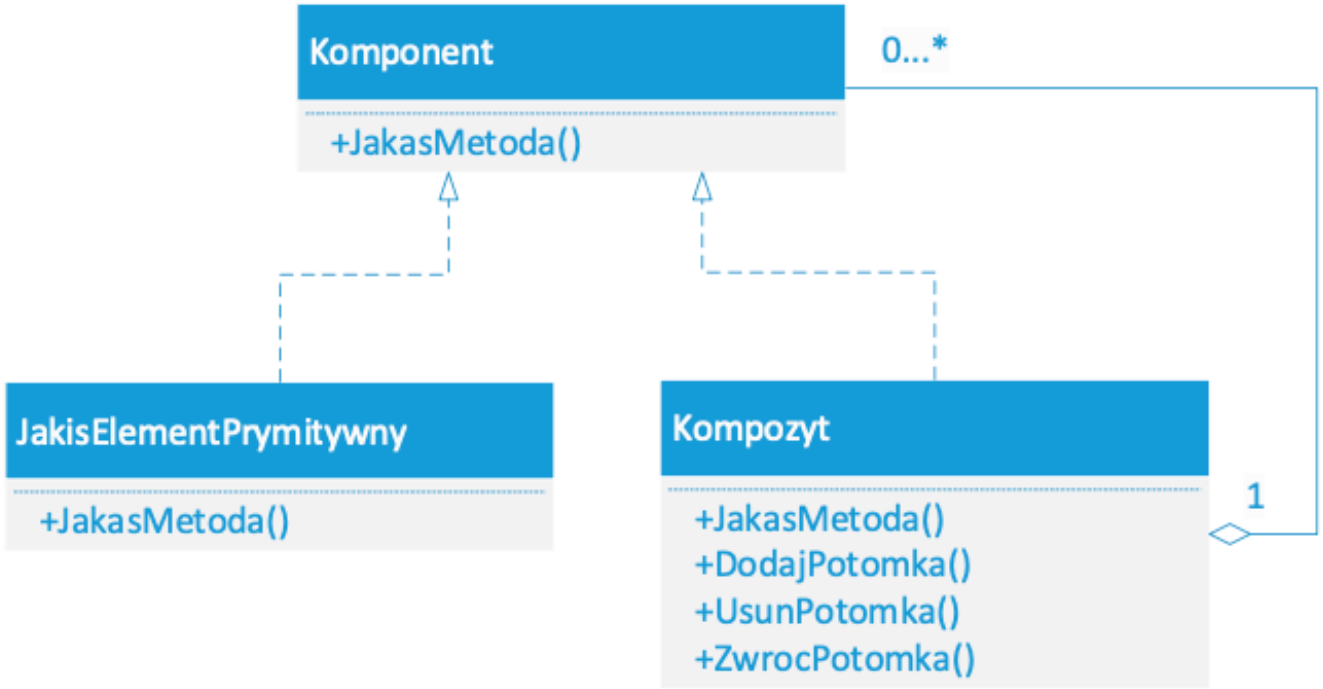
Zastosowanie:

- uproszczenie interakcji z dużymi systemami z wieloma podsystemami
- dla ukrycia złożoności biblioteki zewn
- dla luźnych powiązań między modułami, w celu zdefiniowania jednego punktu wyjścia
- dla łączenia wielu usług w jeden interfejs
- logi

WZORCE STRUKTURALNE

COMPOSITE

- klienci dysponują ujednoliconym dostępem do kolekcji jak do pojedynczych obiektów
- gdy istnieje potrzeba zarządzania i manipulacji grupą obiektów tak samo, jak pojedynczymi obiektami



Schemat kroków do zastosowania wzorca:

1. Zdefiniuj **interfejs** komponentu - powinien definiować operacje, które muszą być wspólne dla wszystkich elementów w hierarchii.
2. Stwórz klasę **liścia**, która implementuje interfejs komponentu i reprezentuje pojedyncze obiekty w hierarchii.
3. Stwórz klasę **kompozytu**, która implementuje interfejs komponentu, ale ma dodatkowe metody do zarządzania dziećmi. Przechowuje **podkomponenty** (mogą to być zarówno liście, jak i inne kompozyty) i zaimplementuje operacje interfejsu komponentu w sposób, który jest sensowny dla struktur zawierających wiele elementów.

4. Używaj interfejsu komponentu do interakcji z obiektami: Bez względu na to, czy obiekt jest pojedynczym elementem (liściem) czy złożonym kompozytem, powinien być manipulowany przez interfejs komponentu. Dzięki temu klient kodu może ignorować różnice między obiektami pojedynczymi i złożonymi.

Zastosowanie:

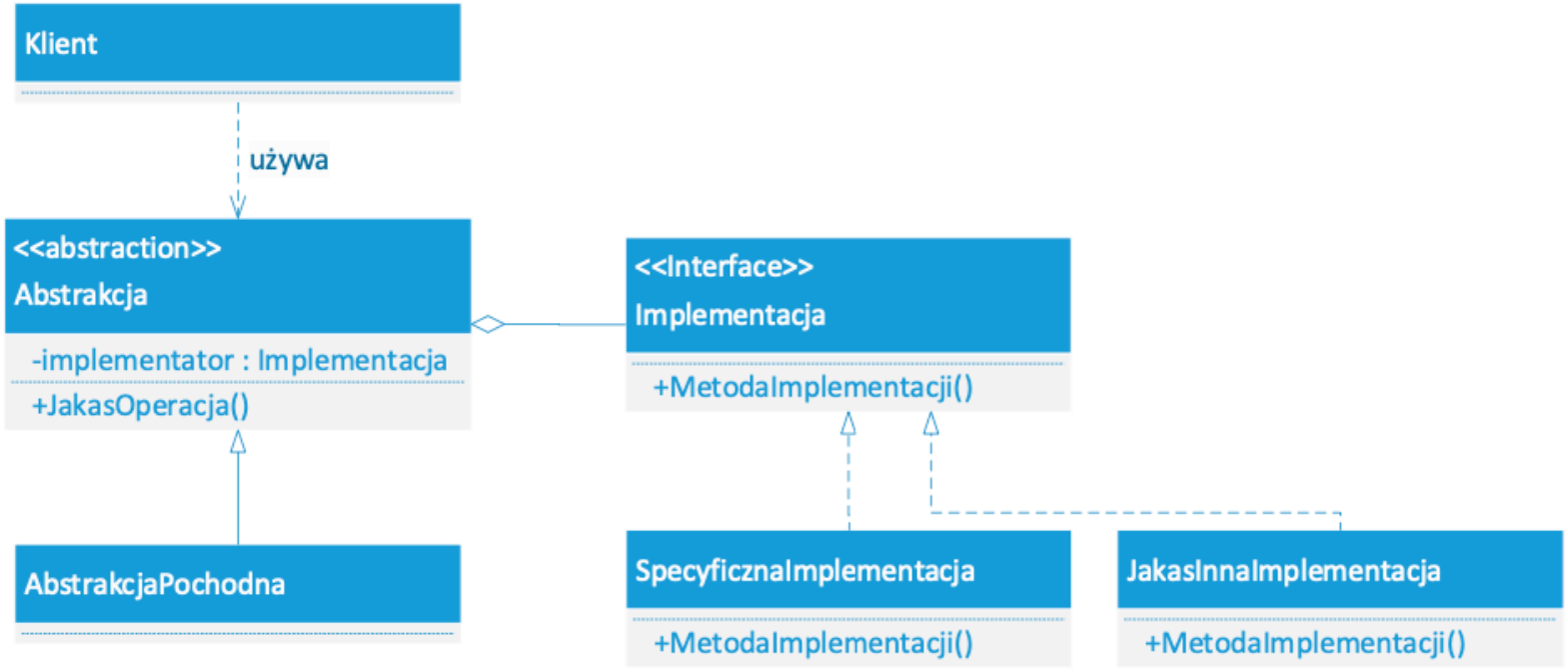
- reprezentowanie struktur hierarchicznych, jak drzewa katalogów, menu aplikacji, itp
- modelowanie dokumentów składających się z różnych elementów jako jednolitej struktury
- budowanie dynamicznych formularzy, gdzie każde pole jest traktowane jak część większej struktury
- umożliwia operacje takie jak wyszukiwanie, sortowanie na strukturach złożonych z obiektów i ich kolekcji

WZORCE STRUKTURALNE

BRIDGE

- zapewnia możliwość modyfikowania nie tylko implementacji, ale i stosowanych abstrakcji poprzez umieszczenie obu elementów w osobnych hierarchiach

- budujemy Most, czyli relację pomiędzy strukturą abstrakcji i strukturą implementacji w przypadkach, gdy konieczne jest modyfikowanie każdej ze stron niezależnie



Kroki do zastosowania wzorca Bridge:

1. **Abstrakcję** można zidentyfikować jako kluczowy interfejs używany przez klienta, natomiast **implementację** identyfikujemy jako różne warianty działania abstrakcji.

2. Podziel abstrakcję i implementację w odrębne klasy. Mogą one istnieć jako hierarchie klasy dla różnych grup abstrakcji i implementacji.

3. Utwórz **interfejs implementacji**, który definiuje metody, które musi dostarczyć konkretna implementacja. Wszystkie klasy implementacji powinny ten interfejs implementować.

4. W klasie abstrakcji dodaj "most" - referencję do obiektu typu interfejs implementacji. Wtedy abstrakcja będzie mogła delegować niektóre (lub wszystkie) swoje zadania do obiektu implementacji.

5. Klient powinien przekazywać obiekt implementacji do konstruktora klasy abstrakcji podczas inicjalizacji i dalej korzystać wyłącznie z obiektu abstrakcji.

Zastosowanie:

- platformy wielosystemowe

- oddzielenie logiki UI od sposobu jego wyświetlania, co pozwala na łatwe zmiany wyglądu bez modyfikacji logiki

- abstrakcja nad protokołami komunikacji

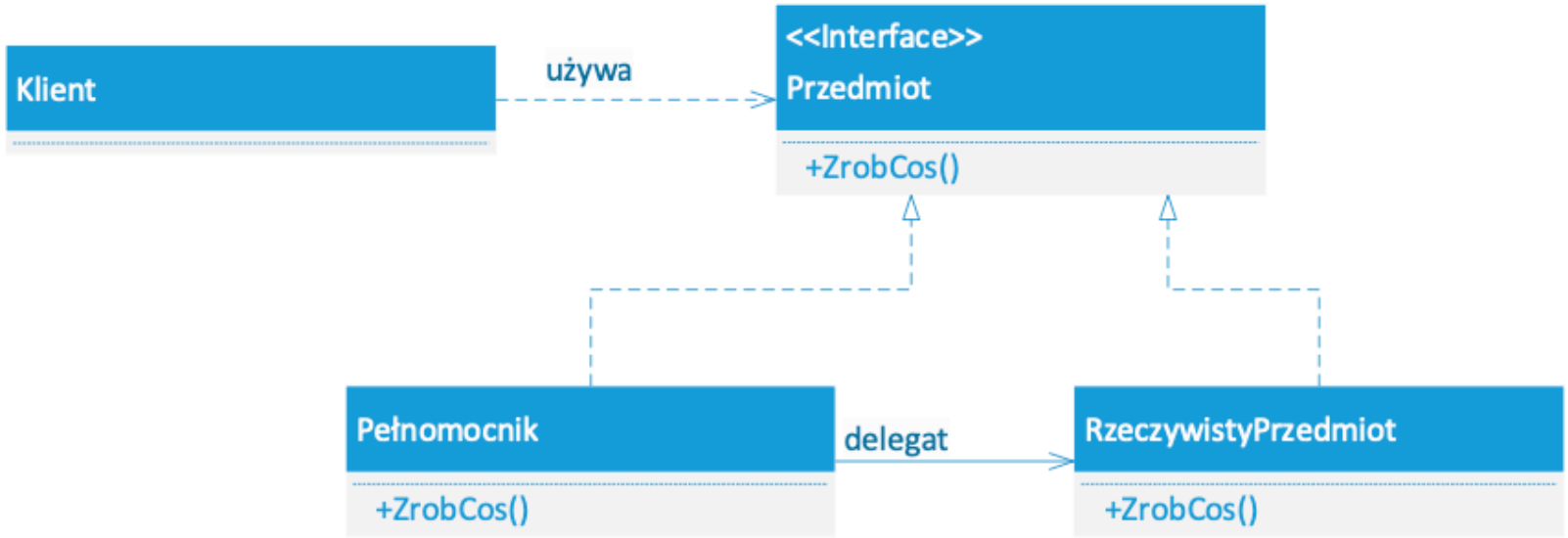
- elastyczne łączenie różnych struktur danych z ich sposobami przetwarzania lub prezentacji

- tworzenie aplikacji, które mogą działać na różnych urządzeniach, z oddzielną logiką i implementacją dla różnej platformy

WZORCE STRUKTURALNE

PROXY

- opakowuje inny obiekt, aby kontrolować dostęp



Kroki do zastosowania wzorca Proxy:

1. Zdefiniuj **interfejs**, który będzie wspólny dla rzeczywistego obiektu i jego proxy. Wszystkie metody, które klient powinien móc wywoływać, powinny być zdefiniowane w tym interfejsie.

2. Utwórz klasę rzeczywistego obiektu, która implementuje powyższy interfejs i zawiera rzeczywistą logikę biznesową.

3. Utwórz klasę Proxy która również implementuje ten sam interfejs. Zawiera referencję do rzeczywistego obiektu i kontroluje dostęp do niego. Może też implementować dodatkową logikę, taką jak ładowanie opóźnione, kontrola dostępu, logowanie i inne.

4. Klient powinien używać proxy tak, jakby to był rzeczywisty obiekt. Proxy przekierowuje wywołania do rzeczywistego obiektu, kiedy to konieczne.

Zastosowanie:

- kontrola dostępu do zasobów przez sprawdzenie uprawnień użytkownika przed wykonywaniem operacji

- opóźnia tworzenie ciężkich obiektów do momentu kiedy będą potrzebne

- logowanie operacji przed przekazaniem ich do obiektu (śledzenie)

- buforowanie

WZORCE STRUKTURALNE

FLYWEIGHT

Stosowany do tworzenia dużej liczby identycznych lub niewiele różniących się obiektów aby móc się nimi posługiwać w jednolity sposób.

Wykorzystuje współdzielenie, przez co zmniejsza zapotrzebowanie aplikacji na pamięć, jednocześnie mogąc zmniejszyć wydajność.
Stosujemy tam, gdzie klasa ma wiele egzemplarzy, a wszystkie mogą być sterowane z identyczny sposób.

Kroki do zastosowania wzorca Flyweight:

1. Rozdzielenie stanu na zewnętrzny i wewnętrzny. Stan wewnętrzny to dane, które są wspólne dla różnych obiektów i nie ulegają zmianie, natomiast stan zewnętrzny to dane, które są specyficzne dla każdego obiektu i mogą ulec zmianie.

2. Implementacja klasy **Flyweight** - stan wewnętrzny i metody do jego manipulowania.
3. Implementacja klasy **Factory** odpowiedzialnej za tworzenie i zarządzanie obiektami Flyweight. Powinna zawierać mechanizm, który sprawdza, czy już istnieje obiekt o danym stanie wewnętrznym, zanim utworzy nowy obiekt. Jeśli tak, powinna zwrócić istniejący obiekt zamiast tworzyć nowy.

4. Używanie obiektów Flyweight: Klienci powinni korzystać z klasy Factory do pobierania obiektów Flyweight, a następnie przekazywać stan zewnętrzny do metod obiektu Flyweight.

Zastosowanie:

- zarządzanie drobnymi obiektami graficznymi

- współdzielenie obiektów reprezentujących pojedyncze znaki w edytorach tekstu dla zmniejszenia zużycia pamięci

- optymalizacja zarządzania sesjami użytkowników przez współdzielenie danych sesji

- efektywne zarządzanie wieloma obiektami które mają wspólne atrybuty

- współdzielenie elementów dokumentu, takich jak formatowanie, aby zmniejszyć zużycie pamięci

WZORCE STRUKTURALNE

REPOSITORY

- **skupia się na organizacji struktury danych, zarządzaniu dostępem do danych i dostarczaniu jednolitego interfejsu do operacji na tych danych**
- **pozwala na łatwą wymianę sposobu przechowywania danych bez konieczności zmiany kodu aplikacji**
- **stosujemy do oddzielenia warstwy dostępu do danych od reszty aplikacji**

Schemat kroków do zastosowania wzorca Repository

1. Utwórz interfejs reprezentujący operacje, które chcesz wykonywać na twoich obiektach domenowych (np. `UserRepositoryInterface`).
2. Utwórz klasę implementującą ten interfejs, która obsługuje szczegóły dostępu do danych (np. `UserRepository`).
3. W miejscach, gdzie potrzebujesz manipulować danymi, użyj obiektu Repository zamiast bezpośredniego dostępu do bazy danych.
4. Dostarcz obiektu Repository do klas, które go potrzebują, np. poprzez wstrzykiwanie zależności.

Zastosowanie:

- abstrakcja warstwy dostępu do danych od logiki biznesowej
- ułatwia testowanie przez możliwość podmiany na mocki lub stuby
- zapewnia jedno miejsce, gdzie zdefiniowane są operacje na danych
- ułatwia migrację między różnymi systemami bazodanowymi

WZORCE BEHAWIORALNE / CZYNNOŚCIOWE

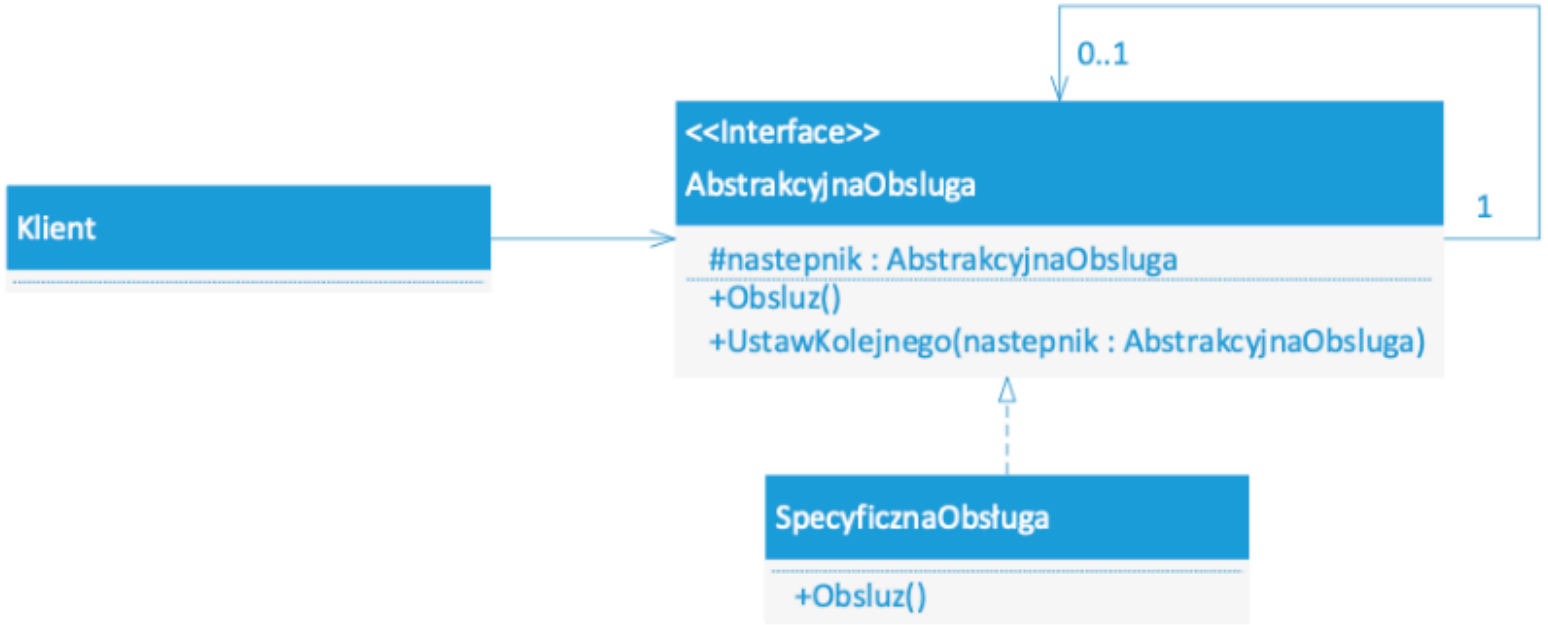
opisują zachowania obiektów,
komunikację pomiędzy nimi i ich odpowiedzialność

- **Chcesz zapewnić klarowną komunikację i odpowiedzialność między obiektami**
- **Chcesz oferować rozwiązania, efektywne i bezpieczne, i które wspierają obiekty w realizacji ich zadań**
- **Chcesz modelować skomplikowane przepływy, sekwencje lub algorytmy**

WZORCE BEHAWIORALNE / CZYNNOŚCIOWE

CHAIN OF RESPONSIBILITY

- stosujemy, gdy kolejne żądania mają być obsługiwane przez różne obiekty
- gdy istnieje wiele różnych typów zadań, które mają specyficzną logikę, istnieje sekwencja lub hierarchia - kolejność
- gdy nie musimy znać szczegółów - decyzje kto ma obsłużyć podejmuje łańcuch



Każdy obiekt w łańcuchu działa jak procedura obsługi żądania i ma pewien następnik, co upraszcza obiekt, bo nie wymaga przechowywania informacji o całej strukturze łańcucha.

Schemat kroków do zastosowania wzorca Chain Of Responsibility:

1. Utwórz abstrakcyjną klasę lub interfejs (zwany "Handler"), który definiuje metodę do obsługi zadań (często nazywaną "handle" lub podobnie) oraz metodę do ustawiania następnego obiektu w łańcuchu (często nazywaną "setNext" lub podobnie).

2. Utwórz konkretne klasy, które implementują ten interfejs "Handler". Każda z tych klas powinna mieć własną implementację metody "handle", która obsługuje odpowiednie zadanie lub przekazuje je dalej do następnego obiektu w łańcuchu.

3. Zbuduj łańcuch obiektów "Handler", ustawiając dla każdego obiektu jego następnika za pomocą metody „setNext".

4. Wywołaj metodę "handle" na pierwszym obiekcie w łańcuchu, przekazując do niej zadanie do obsługi. Ten obiekt albo obsłuży zadanie, albo przekaże je do następnego obiektu, aż zadanie zostanie obsłużone lub dojdzie do końca łańcuchu.

Zastosowanie:

- przetwarzanie zgłoszeń serwisowych w różnych poziomach wsparcia

- przetwarzanie przez szereg filtrów w aplikacjach

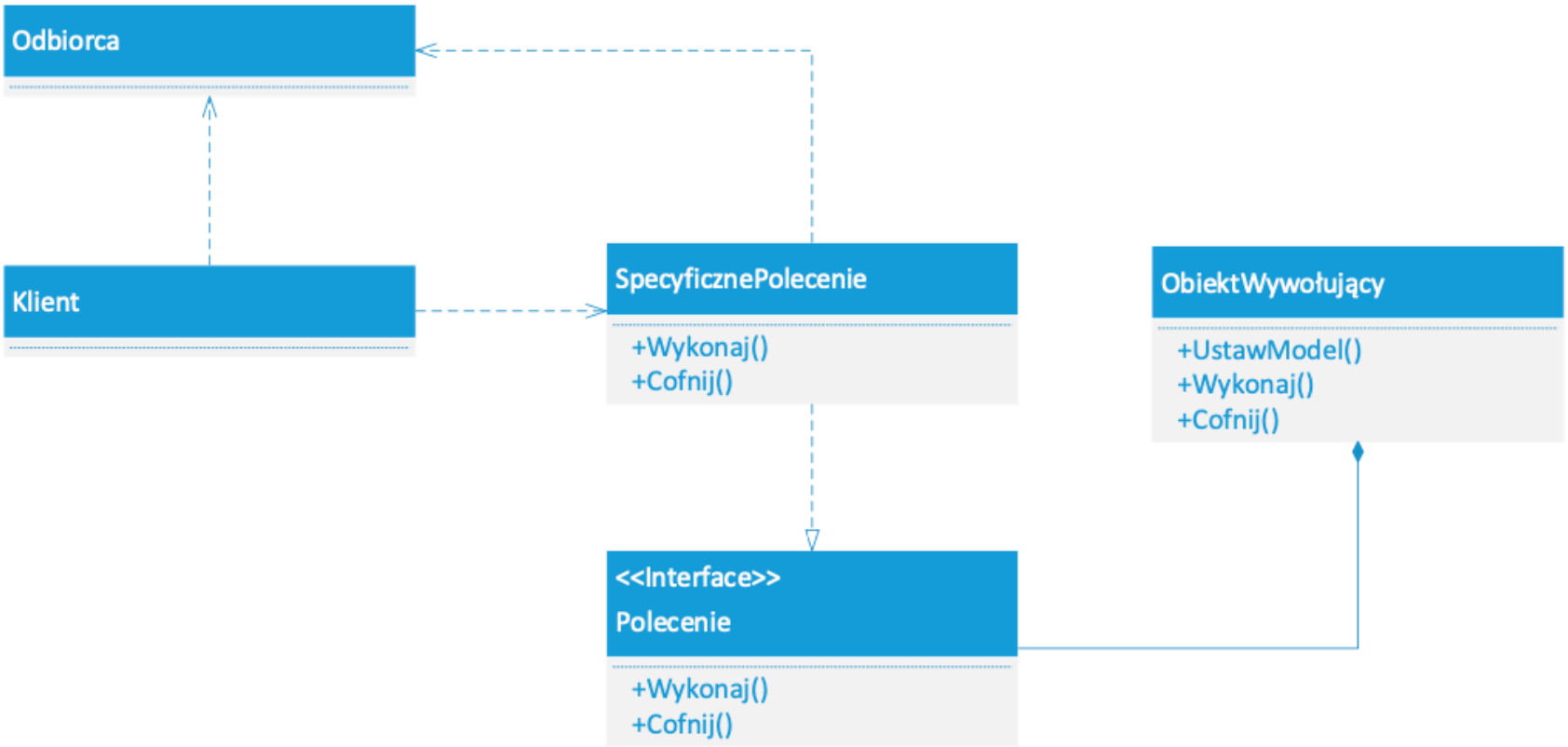
- sekwencyjne sprawdzanie poprawności danych przez różne reguły walidacji

- delegowanie sprawdzania uprawnień

WZORCE BEHAWIORALNE / CZYNNOŚCIOWE

COMMAND

Hermetyzuje żądanie w postaci obiektu



Kroki do zastosowania wzorca Command:

1. Zdefiniuj **interfejs** `Command`, który deklaruje metodę `execute()` i opcjonalnie `undo()`.
2. Dla każdej operacji zdefiniuj klasę, która implementuje interfejs `Command` i realizuje odpowiednią operację.

3. Zdefiniuj klasę `Invoker`, która przyjmuje obiekty `Command` i wywołuje na nich metodę `execute()` w odpowiednim momencie.

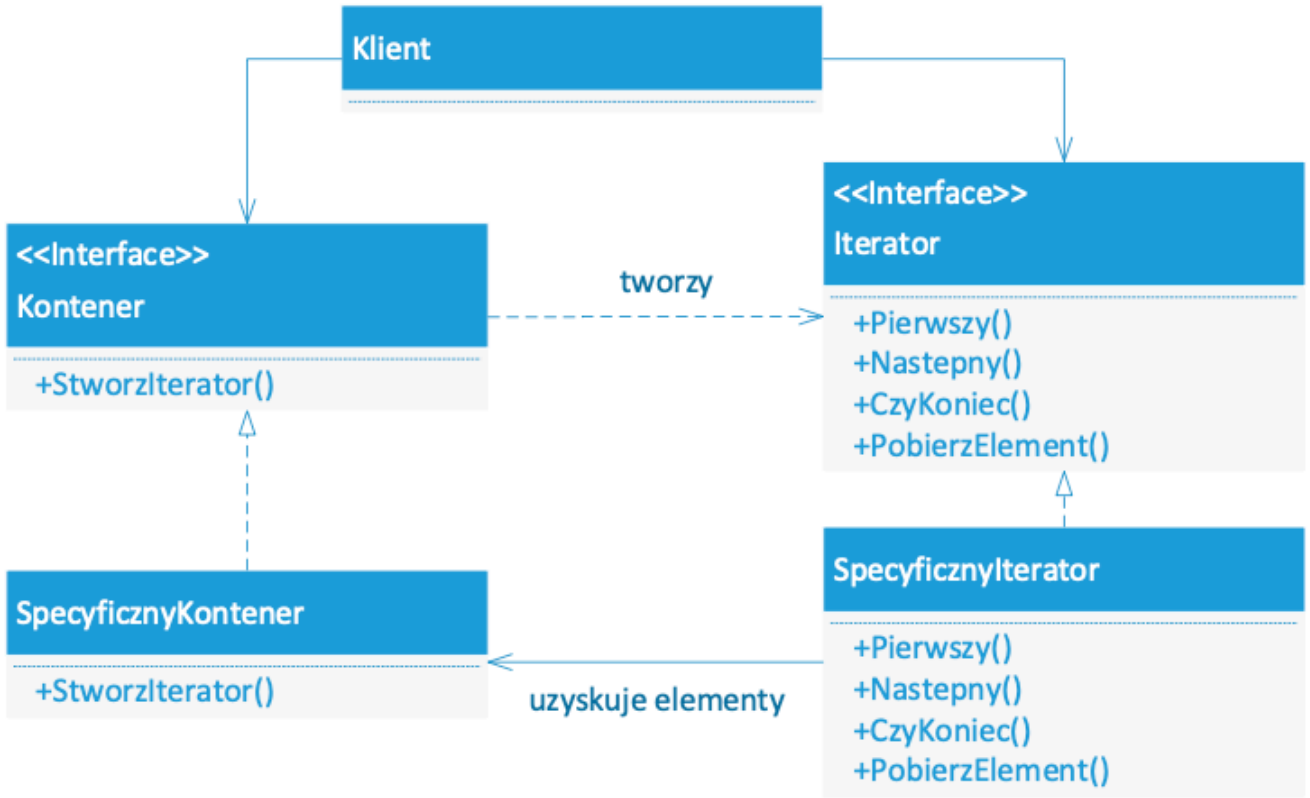
4. Jeżeli potrzebujesz obsługi operacji undo, zadbaj o przechowywanie historii wykonanych operacji w `Invoker`.

5. Stwórz konkretne obiekty `Command` w odpowiednim momencie (np. na podstawie interakcji użytkownika) i przekazuj je do obiektu `Invoker`.

WZORCE BEHAWIORALNE / CZYNNOŚCIOWE

ITERATOR

Zapewnia możliwość przeglądania kolekcji obiektów bez wiedzy o implementacji



Schemat kroków do zastosowania wzorca:

1. Zdefiniuj **interfejs Iteratora**, powinien zawierać metody, które umożliwią klientom przeglądanie elementów kolekcji. Typowe metody to `current()`, `next()`, `key()`, `valid()` i `rewind()`.

2. Zaimplementuj **klasę Iteratora** - powinna implementować zdefiniowany interfejs Iteratora. Powinna zawierać logikę dostępu do elementów kolekcji i przechodzenia przez nie.

3. Użyj Iteratora w klasie kolekcji - klasa, która reprezentuje kolekcję elementów, powinna oferować metodę, która zwraca instancję Iteratora.
   Klienci będą korzystać z tego iteratora do przeglądania elementów kolekcji.

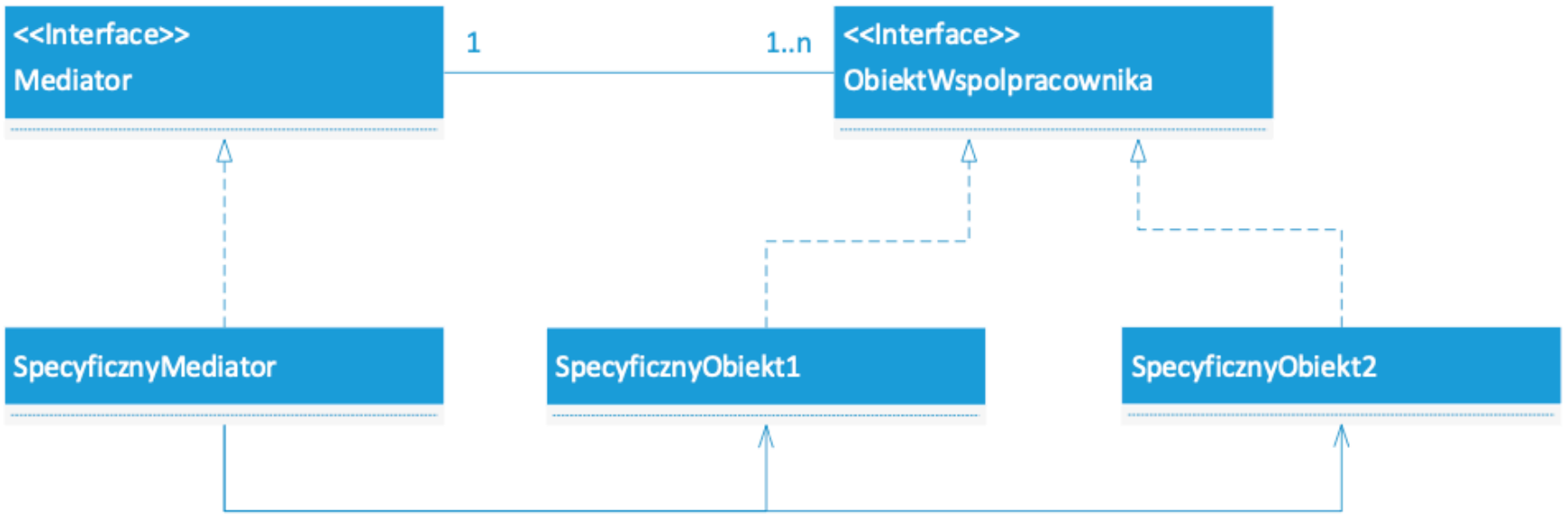
4. Klienci powinni korzystać z Iteratora do przeglądania elementów kolekcji, np. foreach

WZORCE BEHAWIORALNE / CZYNNOŚCIOWE

MEDIATOR

Stosowany do centralizacji złożonych procedur komunikacji i sterowania w środowisku powiązanych obiektów

Wzorzec sugeruje przerwanie bezpośredniej komunikacji między komponentami, które mają być niezależne. W zamian, komponenty te muszą współpracować pośrednio, wywołując specjalny obiekt mediatora, który przekierowuje wywołania do odpowiednich komponentów.



ed

Schemat kroków do zastosowania wzorca Mediator

1. Zdefiniuj interfejs Mediatora, powinien zawierać metody, które pozwalają na komunikację między obiektami. Zwykle zawiera to jedną lub więcej metod "**notify**" lub "**send**".
2. Klasa Mediatora implementuje interfejs Mediatora i zarządza komunikacją między obiektami. Powinna mieć referencje do wszystkich obiektów, z którymi musi komunikować się.

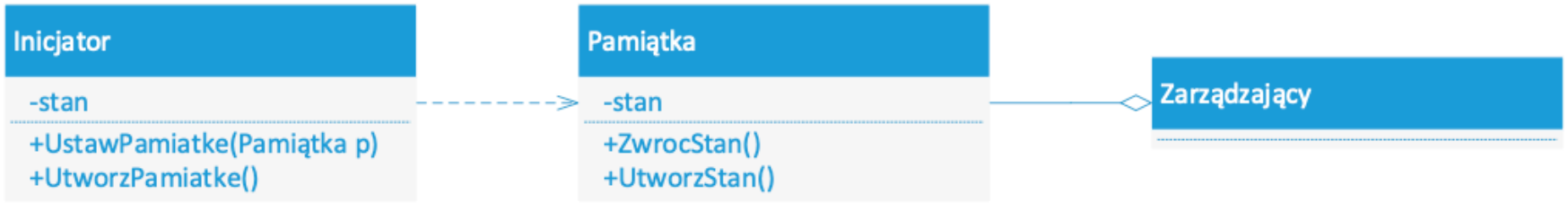
3. Zmodyfikuj klasy obiektów: Klasy obiektów, które będą komunikować się przez Mediatora, powinny być zmodyfikowane, aby używać Mediatora do komunikacji zamiast komunikować się bezpośrednio z innymi obiektami. Powinny mieć referencje do Mediatora i używać go do wysyłania i odbierania wiadomości.

4. Używaj Mediatora w miejscu, gdzie obiekty są tworzone i używane. Następnie można używać Mediatora do zarządzania komunikacją między obiektami.

WZORCE BEHAWIORALNE / CZYNNOŚCIOWE

MEMENTO

- gdy potrzebujemy możliwości przywrócenia obiektu do jednego z wcześniejszych stanów, jak w operacji „Cofnij”



Schemat kroków do zastosowania wzorca Memento:

1. Zdefiniuj `Memento`, **klasę przechowującą stan obiektu**. Ta klasa powinna być **niemutowalna**, tzn. jej stan nie powinien zmieniać się po utworzeniu.

2. Zdefiniuj `Originator`, klasę, której stan jest zapisywany. Ta klasa powinna mieć **metody umożliwiające zapisywanie jej stanu do obiektu memento oraz przywracanie stanu z obiektu memento.**

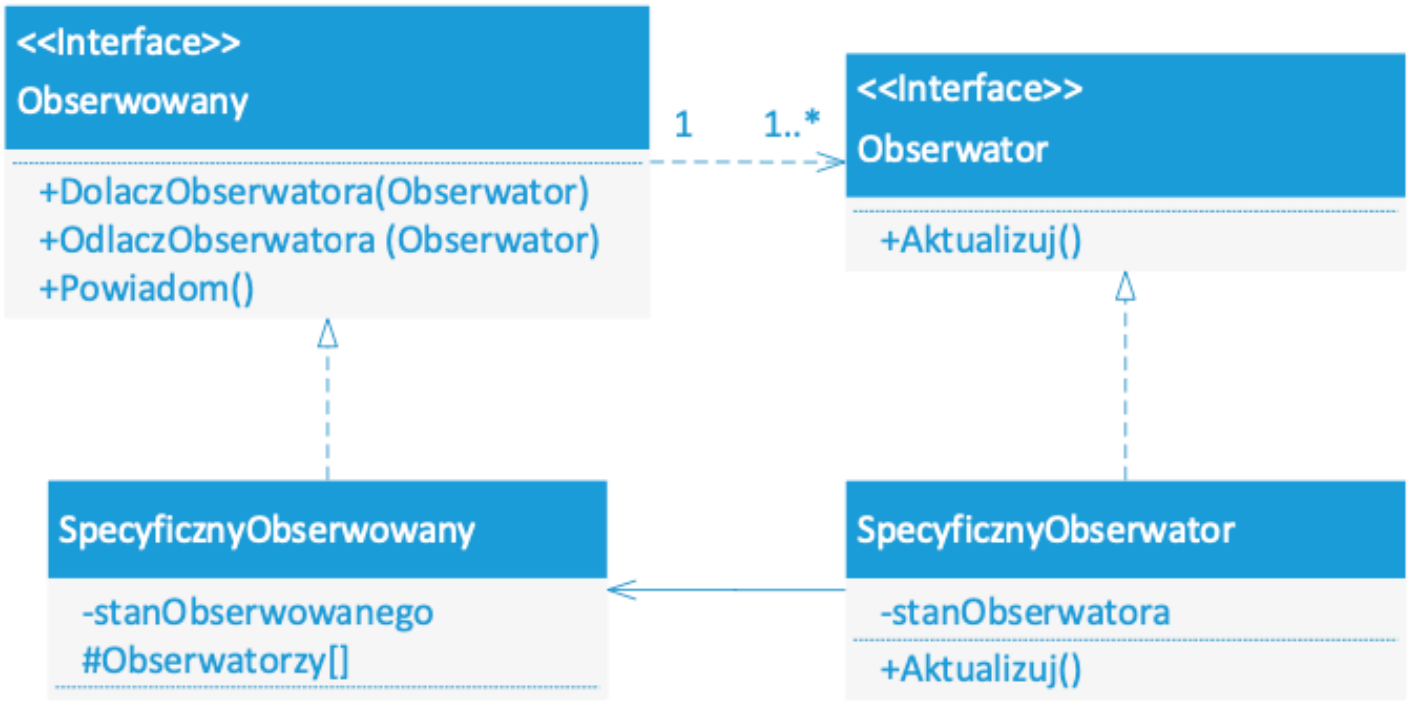
3. Zdefiniuj `Caretaker`, klasę odpowiedzialną za **przechowywanie i zarządzanie** obiektami memento. Ta klasa może mieć metody do dodawania mementos, usuwania mementos i, co najważniejsze, przywracania stanu originatora do stanu zapisanego w memento.

4. Użyj `Caretaker` do zapisywania stanów `Originator` i przywracania stanów z obiektów `Memento` kiedy to potrzebne. Pamiętaj, że Originator nie powinien wiedzieć o istnieniu Caretaker.

WZORCE BEHAWIORALNE / CZYNNOŚCIOWE

OBSERVER

- umożliwia powiadamianie grupy obiektów o zmianie stanu



Schemat kroków do zastosowania wzorca Observer:

1. Zidentyfikuj **obiekt obserwowany** (**subject**), którego zmiany stanu mają być monitorowane przez obserwatorów.
2. Zdefiniuj **interfejs obserwatora** (**observer**), powinien zawierać metodę lub metody, które obserwatorzy będą implementować w celu otrzymywania powiadomień o zmianach.

3. Zaimplementuj obiekty obserwatorów, które implementują interfejs obserwatora.
   Te klasy reprezentują różne obserwatory, które będą monitorować obiekt obserwowany.

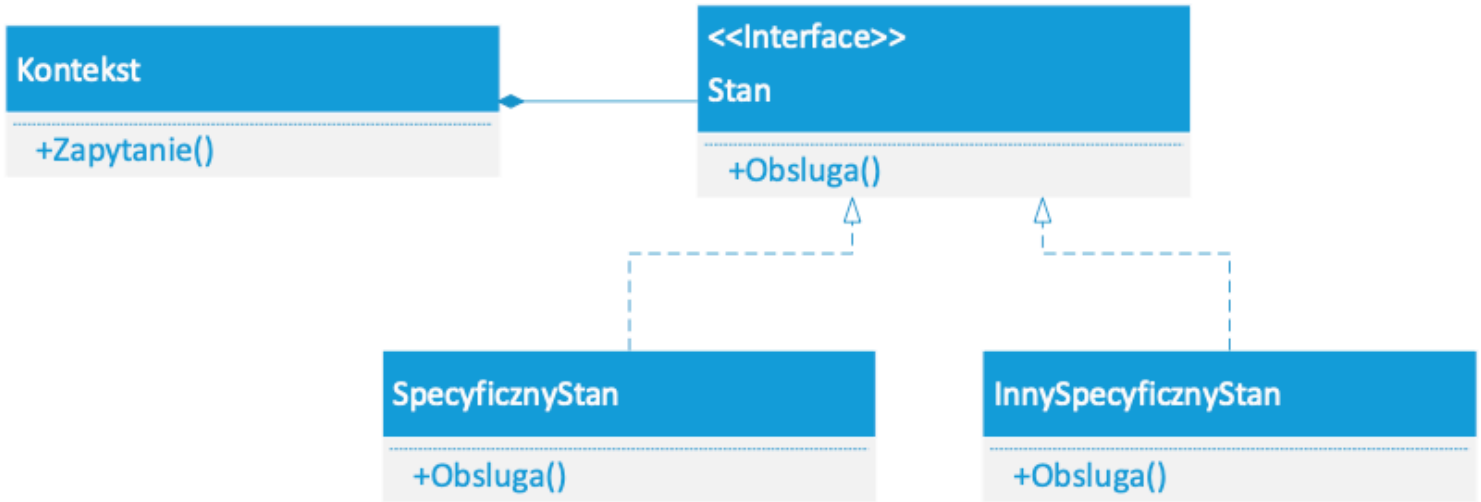
4. W obiekcie obserwowanym utwórz listę obserwatorów.
5. Zdefiniuj metody w obiekcie obserwowanym, które pozwalają obiektowi obserwowanemu na dodawanie, usuwanie i powiadamianie obserwatorów o zmianach stanu.

6. W odpowiednich miejscach w obiekcie obserwowanym, wywołuj metody obserwatorów w celu powiadomienia ich o zmianach stanu.

WZORCE BEHAWIORALNE / CZYNNOŚCIOWE

STATE

- pozwala obiektowi zmieniać swoje zachowanie w zależności od stanu, w którym się znajduje



Schemat kroków do zastosowania tego wzorca:

1. Zdefiniuj interfejs (lub klasę abstrakcyjną), który określa metody, które muszą być zaimplementowane przez wszystkie stany. Te metody powinny odzwierciedlać operacje, które mogą zmieniać wewnętrzny stan obiektu.

2. Utwórz osobną klasę dla każdego stanu, implementując interfejs stanu. Każda klasa stanu powinna hermetyzować logikę związaną z danym stanem.

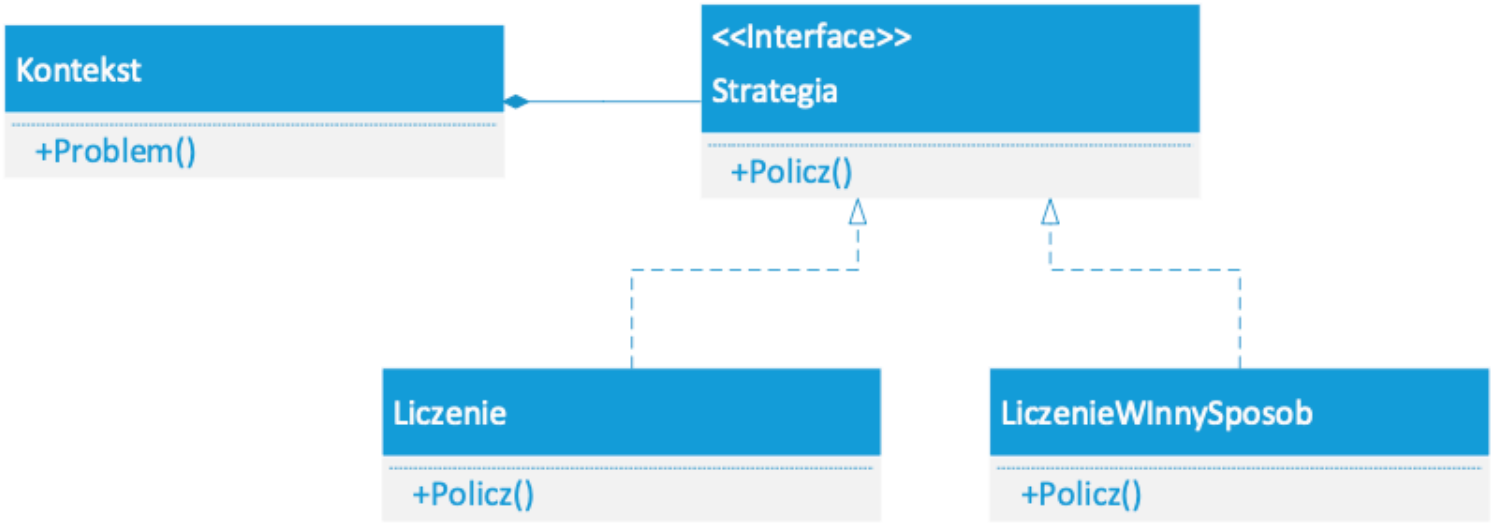
3. W klasie kontekstowej (która przechowuje obiekt stanu) utwórz metodę umożliwiającą zmianę obiektu stanu. Ta metoda może być publiczna, jeśli chcesz zezwolić na dowolną zmianę stanu, lub prywatna, jeśli chcesz, aby tylko obiekty stanu mogły zmienić stan.

4. Wywołaj odpowiednią metodę na obiekcie stanu w odpowiedniej metodzie klasy kontekstowej. Metoda ta powinna delegować wywołanie do obiektu stanu.

WZORCE BEHAWIORALNE / CZYNNOŚCIOWE

STRATEGY

- pomaga w tworzeniu rodziny algorytmów, hermetyzując każdy z nich i sprawia, aby były wymienne



Schemat kroków do zastosowania wzorca strategii:

1. Pierwszym krokiem w zastosowaniu wzorca strategii jest zidentyfikowanie części kodu, które **mogą ulec zmianie lub różnić się w zależności od kontekstu**.
2. Następnie zdefiniuj **interfejs strategii**, który będą implementować wszystkie strategie.

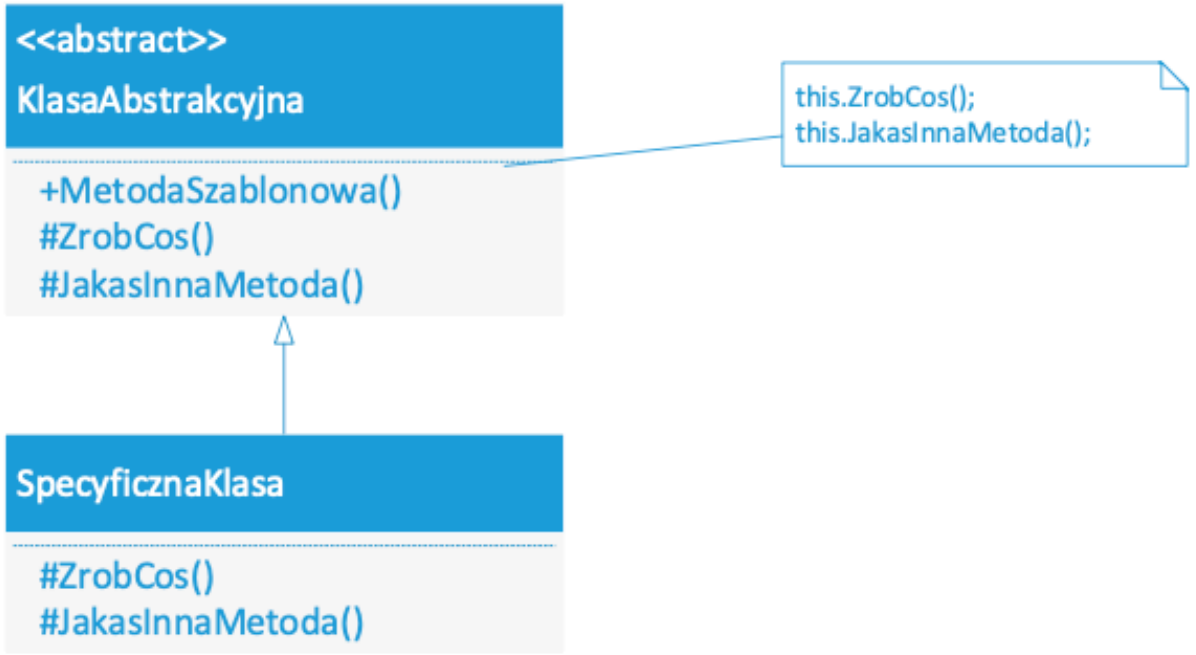
3. Dla każdego zachowania, które może ulec zmianie, zaimplementuj klasę strategii, która implementuje interfejs strategii.

4. Zintegruj strategię z kontekstem - W klasie, która używa różnych strategii (tzw. kontekście), **dodaj metodę, która pozwala na ustawienie strategii**. Ta metoda powinna przyjmować obiekt implementujący interfejs strategii. Kontekst powinien wywoływać metody strategii bez bezpośredniego odwołania do konkretnej klasy strategii.

5. Zmień strategię w czasie wykonywania: w czasie wykonywania programu, możesz zmienić strategię, którą używa kontekst, poprzez wywołanie metody ustawiającej strategię z nową strategią.

TEMPLATE METHOD

- definiowanie szkieletu algorytmu w metodzie, delegując niektóre kroki do podklas
- klasy podrzędne decydują o tym, w jaki sposób zostaną zaimplementowane poszczególne kroki algorytmu



Schemat kroków do zastosowania wzorca Template Method:

1. Zdefiniuj abstrakcyjną klasę bazową, która zawiera szkielet algorytmu i deklaruje metody abstrakcyjne lub wirtualne, które będą implementowane przez konkretne klasy.

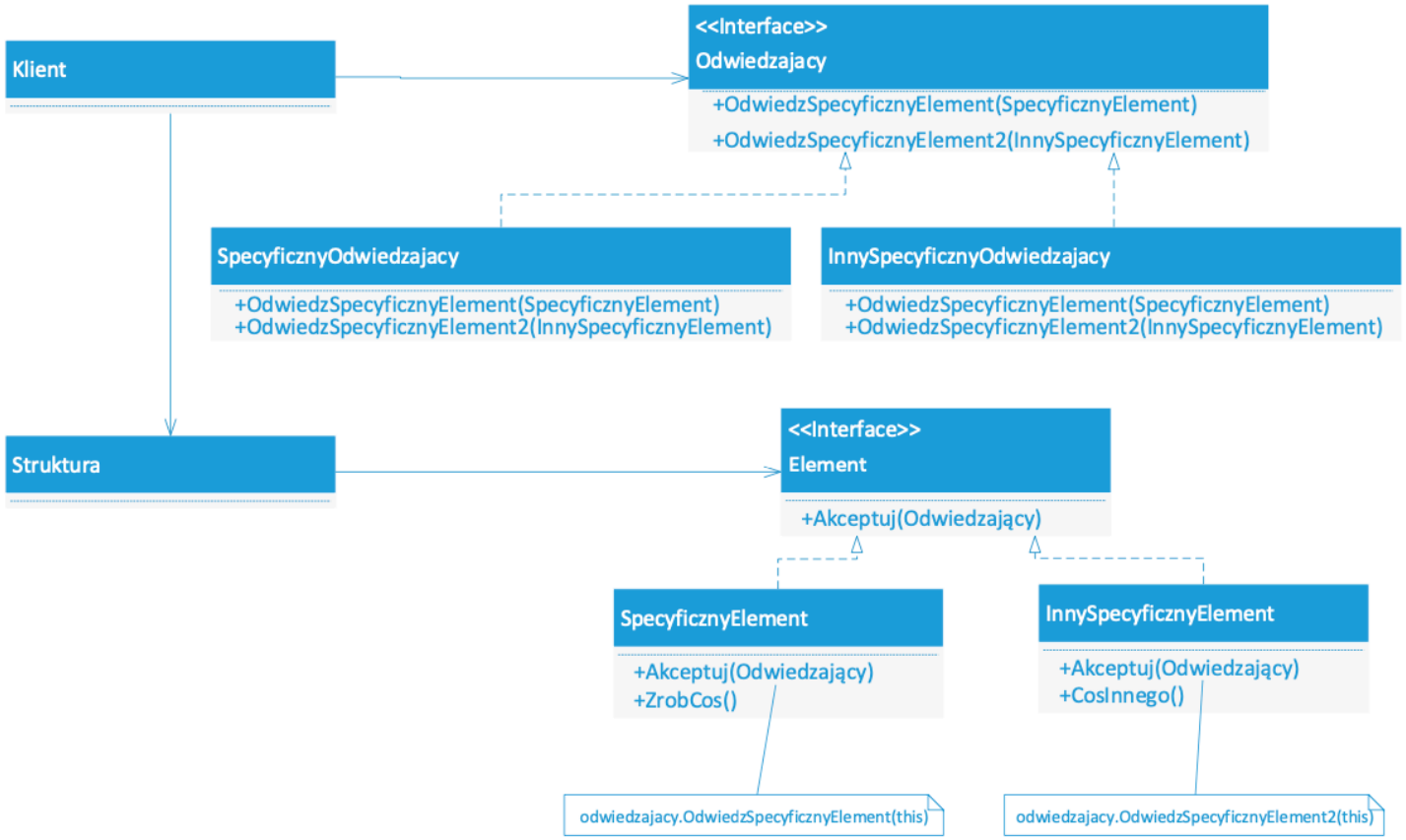
2. Stwórz konkretne klasy dziedziczące po klasie bazowej i zaimplementuj metody abstrakcyjne lub wirtualne, dostosowując je do specyficznych potrzeb. Nadpisz tylko te kroki, które się różnią, a pozostałe pozostaw niezmienione.

3. W głównym kodzie aplikacji utwórz instancję konkretnej klasy i wywołaj jej metody, które wykonają zdefiniowany algorytm, uwzględniając nadpisane kroki w klasach pochodnych.

WZORCE BEHAWIORALNE / CZYNNOŚCIOWE

VISITOR

- oddziela algorytm od struktury obiektu z którym wchodzi w interakcję
- tzn. gdy chcemy dodać nową operację do zestawu podobnych obiektów nie zmieniając ich struktury
- gdy mamy hierarchię klas obiektów i chcemy wykonać różne operacje na tych obiektach w zależności od ich typu.



Schemat kroków do zastosowania wzorca Visitor:

1. Zdefiniuj interfejs **Visitor**, powinien zawierać metody wizytujące dla każdego typu obiektu, na którym chcemy wykonać operacje. Każda metoda wizytująca przyjmuje obiekt konkretnego typu jako argument.
2. Dla każdego typu obiektu utwórz implementację **wizytatora**: Implementacja wizytatora powinna implementować interfejs Visitor i definiować operacje, które mają być wykonane na konkretnym typie obiektu. Każda metoda wizytująca zawiera logikę odpowiednią dla danego typu obiektu.
3. Zdefiniuj interfejs **Element**: Interfejs Element deklaruje metodę **accept (visitor)**, która przyjmuje obiekt Visitor jako argument. Ta metoda będzie wywoływana na obiekcie, który ma zostać odwiedzony.
4. Dla każdego typu obiektu utwórz klasę implementującą interfejs Element, z implementacją metody **accept (visitor)**, która wywołuje odpowiednią metodę **visit()** na obiekcie Visitor.
5. W miejscu, gdzie chcesz wykonać operacje na obiektach, utwórz obiekt odpowiedniego typu **Visitor** i przekaż go do obiektu, który ma zostać odwiedzony, za pomocą metody **accept (visitor)**. Tym samym wywołanie metody **accept** spowoduje wywołanie odpowiedniej metody **visit** na obiekcie Visitor.

Zastosowanie:

- przeglądanie i wykonywanie operacji na strukturach drzewiastych
- serializacja obiektów do różnych formatów
- sprawdzanie poprawności składni przez przeglądanie drzew składniowych
- generowanie raportów przez przeglądanie obiektów biznesowych i zbieranie danych do raportu