Wzorce Projektowe

Spis treści

Podstawowe pojęcia projektowania obiektowego

Hermetyzacja (enkapsulacja) to zasada programowania obiektowego, która polega na ukrywaniu wewnętrznych szczegółów klasy i udostępnianiu dostępu do danych tylko poprzez wyznaczone metody. Dzięki temu chronimy dane przed nieautoryzowaną modyfikacją, zapewniając bezpieczne i kontrolowane operacje na obiekcie.

Abstrakcja to koncepcja w programowaniu obiektowym polegająca na wyodrębnieniu istotnych cech i zachowań obiektu, pomijając szczegóły implementacyjne. Abstrakcyjna klasa definiuje ogólny model i może zawierać metody abstrakcyjne, które muszą być zaimplementowane w klasach pochodnych. **Nie można tworzyć instancji klasy abstrakcyjnej**, co odróżnia ją od zwykłego dziedziczenia, gdzie klasy bazowe mogą być instancjonowane.

Polimorfizm to zasada programowania obiektowego, która pozwala na traktowanie obiektów różnych klas w taki sam sposób, jeśli dzielą wspólny interfejs lub klasę bazową. Dzięki polimorfizmowi można wywoływać te same metody na różnych obiektach, uzyskując odmienne rezultaty zależnie od klasy konkretnego obiektu.

Dziedziczenie to mechanizm w programowaniu obiektowym, który pozwala na tworzenie nowych klas (klas pochodnych) na podstawie już istniejących klas (klas bazowych). Klasy pochodne dziedziczą cechy (pola i metody) klas bazowych, co umożliwia ponowne wykorzystanie kodu, a także wprowadzenie nowych funkcjonalności. Dzięki dziedziczeniu możliwe jest tworzenie hierarchii klas oraz stosowanie polimorfizmu, co zwiększa elastyczność i organizację kodu.

Zasady projektowanie obiektowego

SOLID to akronim pięciu zasad projektowania, które pomagają tworzyć elastyczny, czytelny i łatwy w utrzymaniu kod. Oto rozwinięcie akronimu:

- **S**: Single Responsibility Principle (Zasada pojedynczej odpowiedzialności) Każda klasa powinna mieć tylko jedną odpowiedzialność lub cel. Ułatwia to zrozumienie i utrzymanie klasy, a także sprzyja ponownemu użyciu.
- **O**: Open/Closed Principle (Zasada otwarte/zamknięte) Klasy powinny być otwarte na rozszerzenie, ale zamknięte na modyfikacje. Oznacza to, że powinno być możliwe dodawanie nowych funkcji bez modyfikowania istniejącego kodu.
- **L**: Liskov Substitution Principle (Zasada substytucji Liskov) Obiekty klasy pochodnej powinny być w stanie zastąpić obiekty klasy bazowej, nie zmieniając poprawności programu. Oznacza to, że klasy pochodne muszą implementować wszystkie metody klas bazowych.

- **I**: Interface Segregation Principle (Zasada segregacji interfejsów) Klient nie powinien być zmuszony do zależności od interfejsów, których nie używa. Zamiast jednego dużego interfejsu, lepiej jest mieć wiele mniejszych, bardziej wyspecjalizowanych interfejsów.
- **D**: Dependency Inversion Principle (Zasada odwrócenia zależności) Moduły wyższego poziomu nie powinny zależeć od modułów niższego poziomu, ale od abstrakcji. Oznacza to, że powinno się unikać bezpośrednich zależności pomiędzy klasami.

DRY (Don't Repeat Yourself) to zasada, która promuje unikanie powtarzania kodu. Główna idea polega na tym, że każdy fragment wiedzy lub logiki powinien mieć jedną, jednoznaczną reprezentację w systemie. Dzięki zastosowaniu zasady DRY:

- Zmiany są łatwiejsze do wprowadzenia, ponieważ modyfikacje w jednym miejscu automatycznie odnoszą się do wszystkich części kodu, które tego fragmentu używają.
- Zmniejsza się ryzyko błędów, które mogą wystąpić przy aktualizacji wielu powtarzających się fragmentów.

KISS (Keep It Simple, Stupid) to zasada, która promuje prostotę w projektowaniu i implementacji systemów. Jej główna idea to unikanie nadmiernej złożoności. Kluczowe punkty KISS to:

- Proste rozwiązania są łatwiejsze do zrozumienia, wdrożenia i utrzymania.
- Skupienie się na najważniejszych funkcjonalnościach, bez dodawania zbędnych komplikacji.
- Często prostszy kod prowadzi do lepszej wydajności i mniejszej liczby błędów.

Wzorce Projektowe

Wzorce kreacyjne

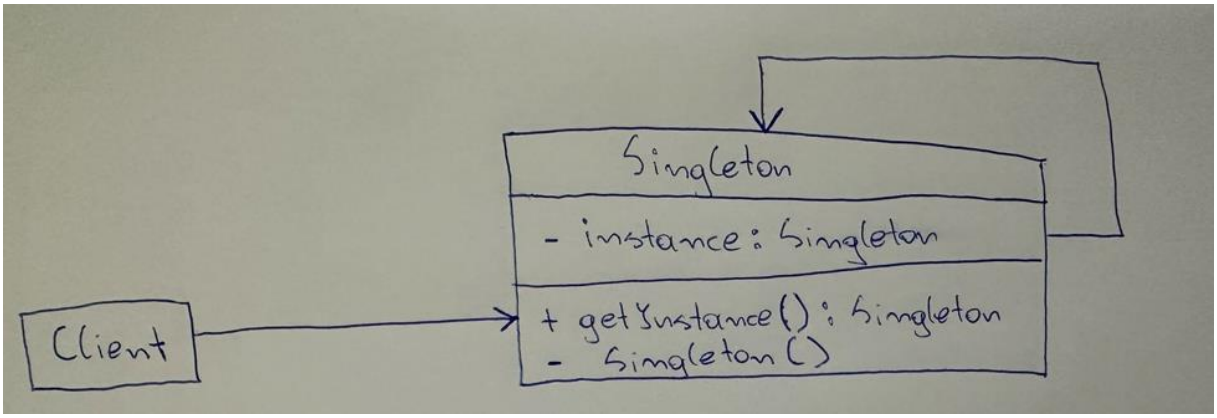
Wzorce kreacyjne koncentrują się na sposobach tworzenia obiektów. Celem tych wzorców jest dostarczenie mechanizmów do zarządzania tworzeniem obiektów w sposób, który zwiększa elastyczność i ponowne użycie kodu. Wzorce te pomagają w ukryciu logiki tworzenia obiektów i zarządzaniu cyklem życia obiektów.

Singleton

Wzorzec Singleton zapewnia, że dana klasa będzie miała tylko i wyłącznie jedną instancję obiektu, i zapewnia globalny punkt dostępu do tej instancji

Analogia do prawdziwego życia

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



1. Ograniczenie dostępu do domyślnego konstruktora przez uczynienie go prywatnym, aby zapobiec stosowaniu operatora **new** w stosunku do klasy Singleton.
2. Utworzenie statycznej metody kreacyjnej, która będzie pełniła rolę konstruktora. Za kulisami, metoda ta wywoła prywatny konstruktor, aby utworzyć instancję obiektu i umieści go w polu statycznym klasy. Wszystkie kolejne wywołania tej metody zwrócą już istniejący obiekt.

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

Zastosowanie

Korzystaj z wzorca Singleton, gdy w twoim programie ma prawo istnieć wyłącznie jeden ogólnodostępny obiekt danej klasy. Przykładem może być połączenie z bazą danych, którego używa wiele fragmentów programu.

Wzorzec projektowy Singleton uniemożliwia tworzenie obiektów danej klasy inaczej, niż przez stosowną metodę kreacyjną. Ta z kolei zwróci albo nowy obiekt, albo wcześniej stworzony.

Stosuj wzorzec Singleton gdy potrzebujesz ściślejszej kontroli nad zmiennymi globalnymi.

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

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

Jak zaimplementować

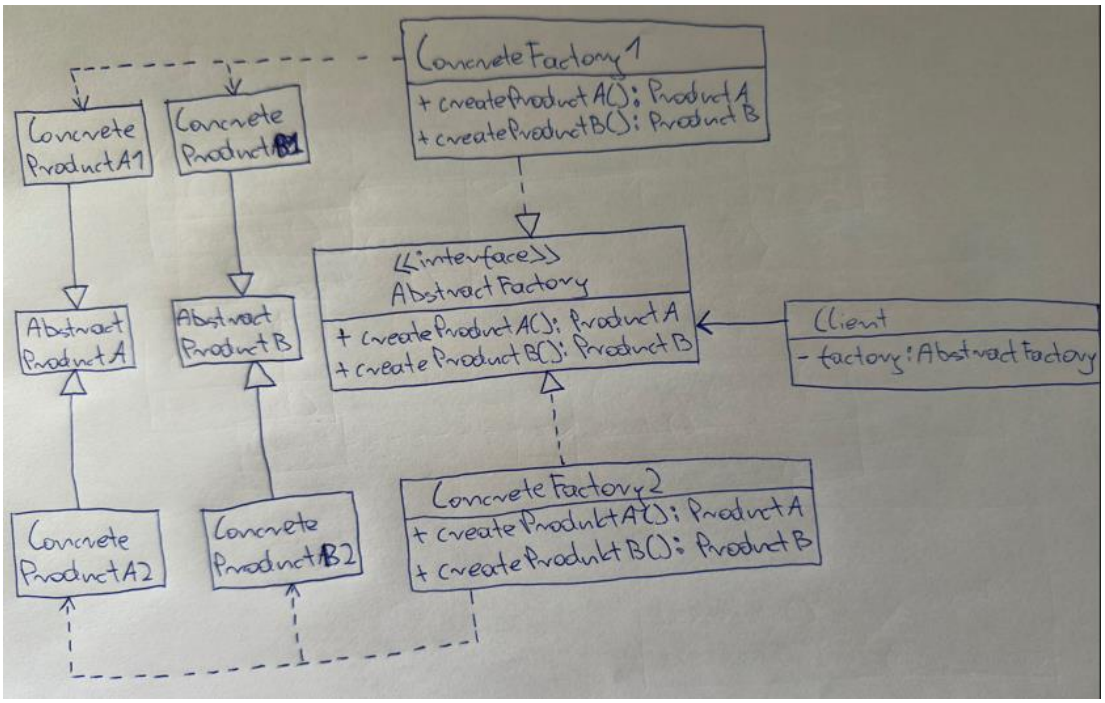
1. Dodaj prywatne, statyczne pole klasy celem przechowywania w nim instancji singleton.
2. Zadeklaruj publicznie dostępną, statyczną metodę kreacyjną która daje dostęp do instancji klasy singleton.
3. Zaimplementuj w tej metodzie statycznej “leniwą inicjalizację”. Oznacza to, że nowy obiekt powinien być tworzony tylko przy pierwszym wywołaniu metody i zdeponowany w statycznym polu klasy. Przy kolejnych wywołaniach, metoda powinna zwracać już istniejący egzemplarz.
4. Uczyń konstruktor klasy prywatnym. Statyczna metoda klasy będzie miała do niego dostęp, ale obiekty innych klas — już nie.
5. Przejrzyj kod kliencki i zamień wszystkie bezpośrednie wywołania konstruktora klasy singleton na wywołania jej statycznej metody kreacyjnej.

Fabryka abstrakcyjna

Fabryka Abstrakcyjna jest wzorcem projektowym, który dostarcza interfejs do tworzenia rodzin powiązanych lub zależnych obiektów bez określania ich konkretnych klas. Fabryka abstrakcyjna pozwala klientowi tworzyć produkty z tej samej rodziny (np. nowoczesne lub klasyczne meble), gwarantując, że wszystkie obiekty będą współpracować, bez ujawniania konkretnego typu produktu.

Analogia do prawdziwego życia

Fabryka Abstrakcyjna w prawdziwym życiu może być **fabryka mebli**, która produkuje całe zestawy różnych typów mebli w jednym stylu. Wyobraźmy sobie, że mamy dwie fabryki mebli: jedna produkuje meble nowoczesne, a druga meble klasyczne. Każda fabryka wytwarza zestawy składające się z krzesła, stołu i sofy, ale wszystkie te elementy będą różnić się wyglądem w zależności od stylu fabryki.



1. **Produkty Abstrakcyjne** deklarują interfejsy odmiennych produktów, które składają się na wspólną rodzinę.
2. **Konkretne Produkty** to różnorakie implementacje abstrakcyjnych produktów, pogrupowane według wariantów. Każdy abstrakcyjny produkt (fotel/sofa) musi być zaimplementowany we wszystkich zadanych wariantach (Wiktoriański/Nowoczesny).
3. Interfejs **Fabryki Abstrakcyjnej** deklaruje zestaw metod służących tworzeniu każdego z abstrakcyjnych produktów.
4. **Konkretne Fabryki** implementują metody kreacyjne fabryki abstrakcyjnej. Każda konkretna fabryka jest związana z jakimś określonym wariantem produktu i produkuje wyłącznie meble w tym stylu.
5. Mimo, że konkretne fabryki tworzą konkretne egzemplarze produktu, sygnatury ich metod kreacyjnych muszą zwracać stosowne *abstrakcyjne* produkty. Dzięki temu kod kliencki, który korzysta z fabryki, nie zostanie sprzęgnięty z jakimś konkretnym wariantem produktu, jaki otrzymuje z fabryki. **Klient** może działać na dowolnym konkretnym wariancie fabryki/produktu, o ile będzie korzystał z interfejsów abstrakcyjnych ich obiektów.

Zastosowanie

Stosuj Fabrykę abstrakcyjną, gdy twój kod ma działać na produktach z różnych rodzin, ale jednocześnie nie chcesz, aby ściśle zależał od konkretnych klas produktów. Mogą one bowiem być nieznane na wcześniejszym etapie tworzenia programu, albo chcesz umożliwić przyszłą rozszerzalność aplikacji.

Fabryka abstrakcyjna dostarcza ci interfejs służący tworzeniu obiektów z różnych klas danej rodziny produktów. O ile twój kod będzie kreował obiekty za pośrednictwem tego interfejsu — nie musisz się martwić stworzeniem produktu w niezgodnym z innymi wariancie.

Przemyśl ewentualną implementację wzorca Fabryki abstrakcyjnej, gdy masz do czynienia z klasą posiadającą zestaw Metod wytwórczych które zbytnio przyćmiewają główną odpowiedzialność tej klasy.

W prawidłowo zaprojektowanym programie *każda klasa jest odpowiedzialna za jedną rzecz*. Gdy zaś klasa ma do czynienia z wieloma typami produktów, warto być może zebrać jej metody wytwórcze i umieścić je w osobnej klasie fabrycznej, albo nawet w pełni zaimplementować Fabrykę abstrakcyjną z ich pomocą.

Jak zaimplementować

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

Wzorce strukturalne

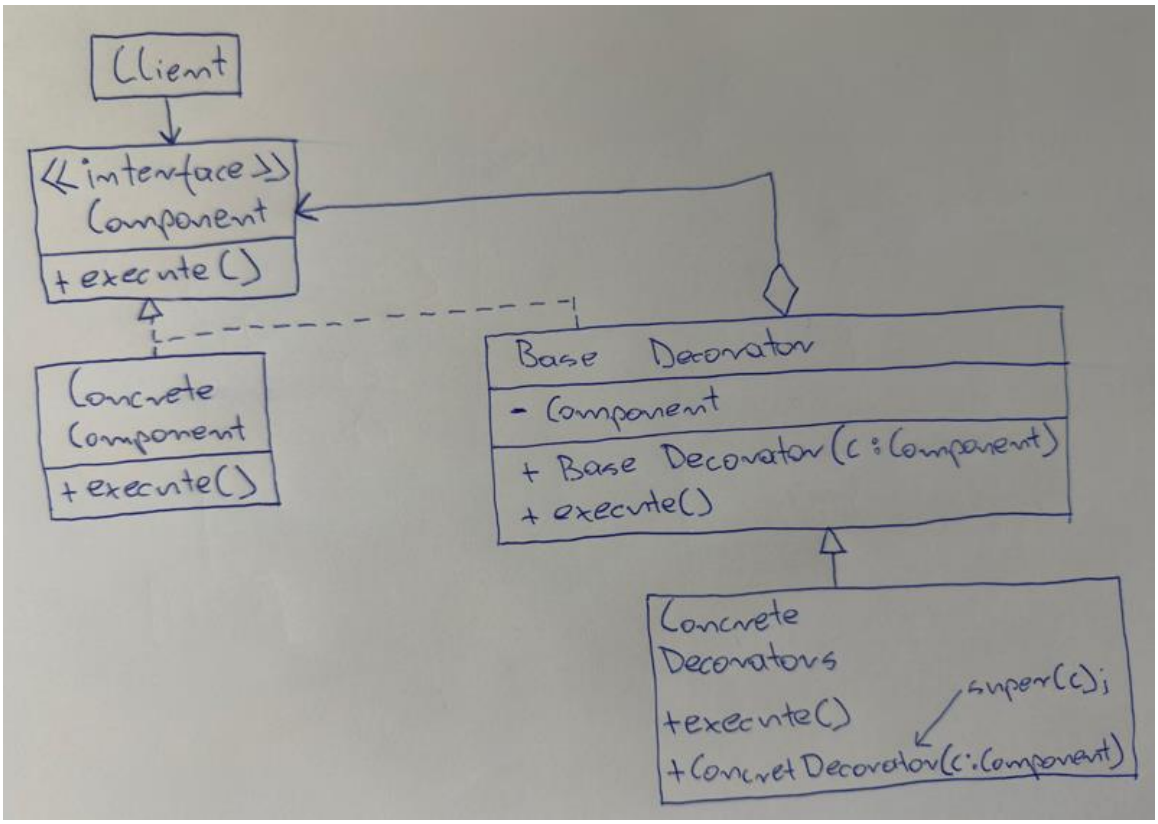
Wzorce strukturalne dotyczą organizacji klas i obiektów w większe struktury. Ich celem jest zapewnienie, że jeśli jedna część systemu zmienia się, inne części nie muszą się zmieniać. Wzorce te koncentrują się na relacjach między obiektami i tym, jak współpracują.

Dekorator

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

Analogia do prawdziwego życia

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



- **Komponent** deklaruje interfejs wspólny zarówno dla nakładek, jak i opakowywanych obiektów.
- **Konkretny Komponent** to klasa opakowywanych obiektów. Definiuje ona podstawowe zachowanie, które następnie można zmieniać za pomocą dekoratorów.
- Klasa **Bazowy Dekorator** posiada pole przeznaczone na referencję do opakowywanego obiektu. Typ pola powinien być zadeklarowany jako interfejs komponentu, aby mogło przechować zarówno konkretne komponenty, jak i inne dekoratory. Dekorator bazowy deleguje wszystkie działania opakowywanemu obiektowi.
- **Konkretni Dekoratorzy** definiują dodatkowe zachowania które można przypisać do komponentów dynamicznie. Konkretni dekoratorzy nadpisują metody dekoratora bazowego i wykonują swoje działania albo przed, albo po wywołaniu metody klasy-rodzica.
- **Klient** może opakowywać komponenty w wiele warstw dekoratorów, o ile działa na wszystkich obiektach poprzez interfejs komponentu.

Zastosowanie

Stosuj wzorzec Dekorator gdy chcesz przypisywać dodatkowe obowiązki obiektom w trakcie działania programu, bez psucia kodu, który z tych obiektów korzysta.

Dekorator pozwala ustrukturyzować logikę biznesową w formie warstw, tworząc dekorator dla każdej warstwy i składać obiekty z różnymi kombinacjami tej logiki w czasie działania programu.

Kod klienta może traktować wszystkie obiekty w taki sam sposób, ponieważ wszystkie są zgodne pod względem wspólnego interfejsu.

Stosuj ten wzorzec gdy rozszerzenie zakresu obowiązków obiektu za pomocą dziedziczenia byłoby niepraktyczne, lub niemożliwe.

Wiele języków programowania posiada słowo kluczowe **final**, za pomocą którego uniemożliwia się dalsze rozszerzanie klasy. W przypadku klasy finalnej, jedynym sposobem na ponowne wykorzystanie istniejącego zachowania jest opakowanie jej nakładkami swojego autorstwa — zgodnie ze wzorcem Dekorator.

Jak zaimplementować

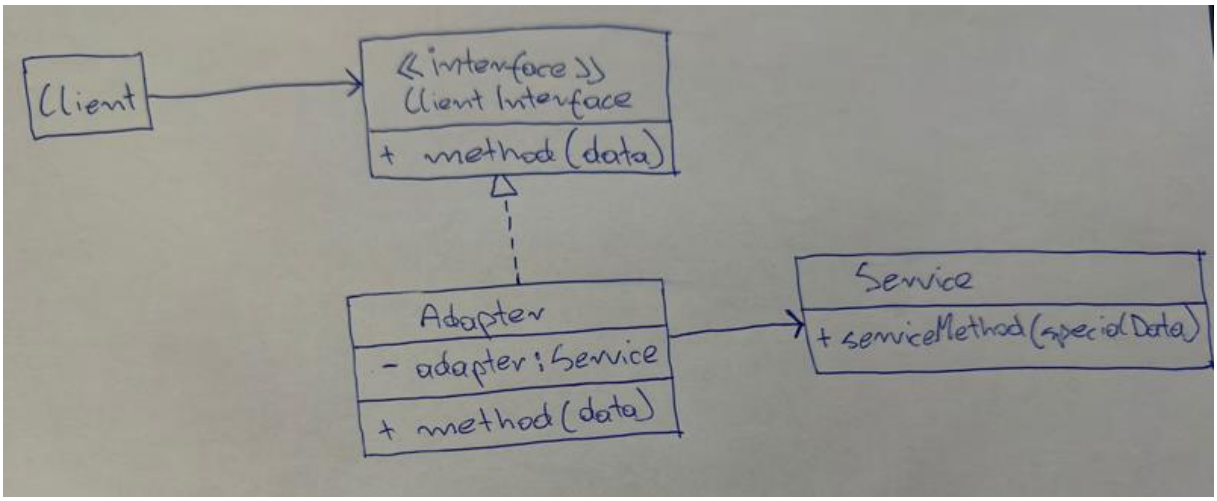
1. Upewnij się, że twoja domena biznesowa może zostać przedstawiona w formie podstawowego komponentu z nałożonymi nań wieloma opcjonalnymi warstwami.
2. Ustal jakie metody są wspólne zarówno dla podstawowego komponentu, jak i warstw opcjonalnych. Stwórz interfejs komponentu i zadeklaruj tam owe wspólne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

Adapter

Adapter jest wzorcem projektowym, który umożliwia współpracę między interfejsami, które normalnie nie są ze sobą zgodne. Działa jak most, przekształcając interfejs jednego obiektu na interfejs, który jest oczekiwany przez klienta. Dzięki temu można używać istniejących klas bez konieczności ich modyfikowania.

Analogia do prawdziwego życia

Gdy pierwszy raz podróżujesz z Polski do Wielkiej Brytanii, możesz zdziwić się próbując naładować laptop. Standardy wtyczek i gniazd są różne. Twoja wtyczka nie będzie pasować do brytyjskiego gniazdka. Problem można rozwiązać za pomocą przejściówki posiadającej gniazdo typu europejskiego oraz wtyk typu brytyjskiego.



1. **Klient** jest klasą zawierającą istniejącą logikę biznesową programu.
2. **Interfejs Klienta** opisuje protokół którego muszą się trzymać pozostałe klasy by móc współdziałać z kodem klienckim.
3. **Usługa** to jakaś użyteczna klasa (na ogół od innego producenta lub starsza). Klient nie jest w stanie użyć jej bezpośrednio z powodu niekompatybilnego interfejsu.
4. **Adapter** to klasa która jest w stanie współdziałać zarówno z klientem jak i z usługą: implementuje interfejs klienta, opakowując obiekt-usługę. Adapter otrzymuje wywołania od klienta za pośrednictwem interfejsu klienta i tłumaczy je na wywołania których format zrozumie obiekt udostępniający usługę.
5. Kod kliencki nie ulegnie sprzęgnięciu z konkretną klasą adaptera, o ile może współpracować z adapterem za pośrednictwem interfejsu klienta. Dzięki temu można wprowadzać nowe typy adapterów do programu bez psucia istniejącego kodu klienckiego. To przydatne, gdy interfejs klasy udostępniającej usługę ulegnie zmianie, bo wówczas wystarczy dodać nową klasę adapter bez konieczności zmiany kodu klienckiego.

Zastosowanie

Stosuj klasę Adapter gdy chcesz wykorzystać jakąś istniejącą klasę, ale jej interfejs nie jest kompatybilny z resztą twojego programu.

Wzorzec Adapter pozwala utworzyć klasę która stanowi warstwę pośredniczącą pomiędzy twoim kodem, a klasą pochodzącą z zewnątrz, lub inną, posiadającą jakiś nietypowy interfejs.

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

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

Znacznie bardziej eleganckim rozwiązaniem jest umieszczenie brakującej funkcjonalności w klasie adapter i opakowanie nią obiektów pozbawionych potrzebnych funkcji. Aby to zadziałało,

klasy docelowe muszą mieć wspólny interfejs, a pole adaptera musi być z nim zgodne. Podejście to bardzo przypomina wzorzec Dekorator.

Jak zaimplementować

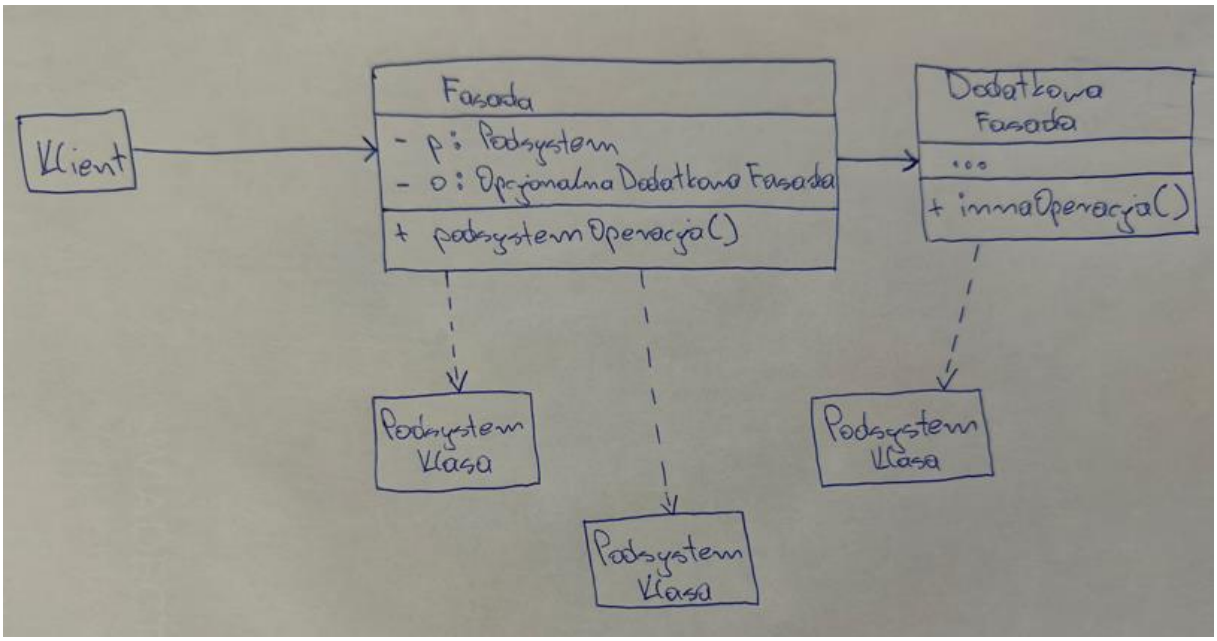
1. Upewnij się, że masz przynajmniej dwie klasy o niekompatybilnych interfejsach:
    - Jakąś użyteczną klasę *usługową* której nie możesz zmienić (innego producenta, przestarzałą, albo ze zbyt wieloma istniejącymi zależnościami)
    - Jedną lub wiele klas *klienckich* które zyskałyby na możliwości skorzystania z powyższej usługi.
2. Zadeklaruj interfejs klienta i opisz jak ma wyglądać komunikacja klientów z usługą.
3. Stwórz klasę adapter zgodną z interfejsem klienckim. Póki co, pozostaw metody pustymi.
4. Dodaj pole do klasy adapter, które przechowa odniesienie do obiektu usługi. Typową praktyką jest inicjalizacja tego pola za pośrednictwem konstruktora, ale czasem wygodniej jest przekazać usługę adapterowi wywołując jego metody.
5. Jeden po drugim, zaimplementuj wszystkie metody interfejsu klienta w klasie adapter. Adapter powinien delegować większość faktycznej pracy obiektowi oferującemu usługę i zajmować się wyłącznie pośrednictwem lub konwersją danych.
6. Klienci powinni stosować adapter za pośrednictwem interfejsu klienta. Pozwoli to zmieniać lub rozszerzać adaptery bez wpływania na kodu kliencki.

Fasada

Fasada służy do ujednolicenia dostępu do złożonego systemu poprzez wystawienie uproszczonego, uporządkowanego interfejsu programistycznego, który ułatwia jego użycie.

Analogia do prawdziwego życia

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



1. **Fasada** daje wygodny dostęp do pewnego zakresu funkcjonalności podsystemu. Wie dokąd przekierować żądanie klienta i jak pokierować wszystkimi szczegółami.
2. Można stworzyć klasę **Dodatkowa Fasada**, by zapobiec zaśmieceniu pojedynczej fasady niepotrzebnymi funkcjami, które ponownie ograniczyłyby prostotę używania. Dodatkowe fasady mogą być wykorzystane zarówno przez klienta, jak i inne fasady.
3. **Złożony podsystem** składa się z wielu różnych obiektów. Aby mogły one wszystkie wykonać coś pożytecznego, trzeba zagłębić się w szczegóły implementacyjne podsystemu — inicjalizację obiektów we właściwej kolejności i przekazanie im danych w odpowiednim formacie.
    - Klasy podsystemu nie są świadome istnienia fasady. Działają wewnątrz systemu i współpracują ze sobą bezpośrednio.
4. **Klient** korzysta z fasady zamiast wywoływać obiekty podsystemu bezpośrednio.

Zastosowanie

Użyj wzorca Fasada gdy potrzebujesz ograniczonego, ale łatwego w użyciu interfejsu do złożonego podsystemu.

Zazwyczaj podsystemy stają się coraz bardziej złożone z biegiem czasu. Nawet stosowanie wzorców projektowych prowadzi do przyrostu liczby klas. Podsystem może wprawdzie stać się elastyczniejszym i łatwiejszym do ponownego użycia w różnych kontekstach, ale ilość kodu konfigurującego i przygotowawczego wymagana od klienta wzrośnie. Fasada jest sposobem rozwiązania tego problemu poprzez udostępnienie skrótów do najczęściej używanych funkcji podsystemu, zgodnie z wymaganiami klienta.

Stosuj Fasadę gdy chcesz ustrukturyzować podsystem w warstwy.

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

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

Jak zaimplementować

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

2. Zadeklaruj i zaimplementuj ten interfejs jako nową klasę fasada. Fasada powinna przekierowywać wywołania pochodzące od kodu klienckiego do stosownych obiektów podsystemu. Ponadto, fasada powinna być odpowiedzialna za inicjalizację podsystemu i zarządzanie jego dalszym cyklem życia, chyba, że kod kliencki już pełni tę rolę.

3. Aby w pełni skorzystać na tym wzorcu, spraw, aby kod kliencki komunikował się z podsystemem wyłącznie poprzez fasadę. W ten sposób kod kliencki pozostanie chroniony przed zmianami dokonywanymi w kodzie podsystemu. Na przykład, aktualizacja podsystemu będzie wymagała jedynie zmian w kodzie fasady.

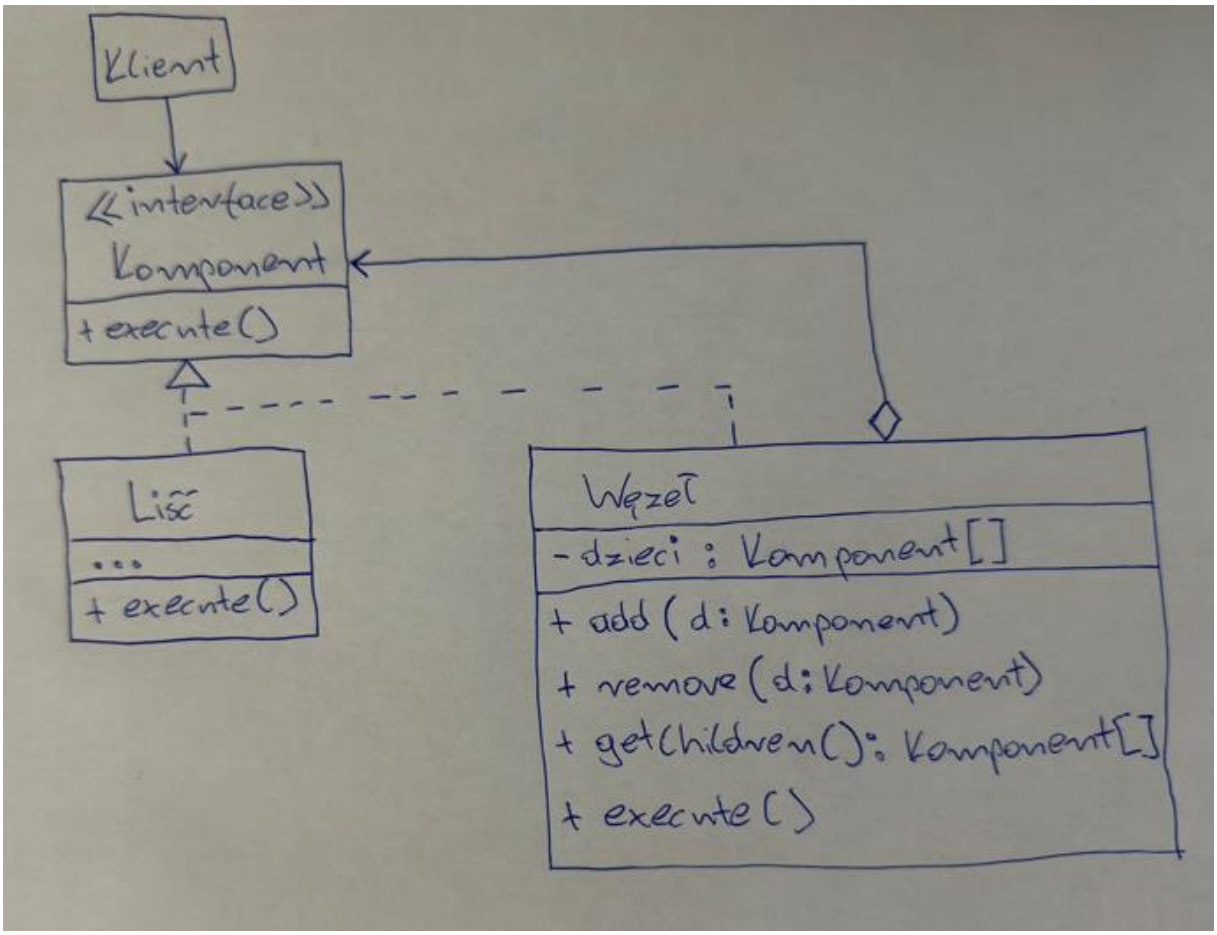
4. Jeśli fasada stanie się zbyt duża, rozważ ekstrakcję części jej obowiązków do nowej, doskonalszej klasy fasady.

Kompozyt

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

Analogia do prawdziwego życia

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



1. Interfejs **Komponentu** opisuje operacje wspólne zarówno dla prostych, jak i złożonych elementów drzewa.
2. **Liść** jest podstawowym elementem drzewa i nie posiada elementów podrzędnych.
3. Zazwyczaj, to właśnie komponenty-liście wykonują większość faktycznej pracy, ponieważ nie mają komu jej zlecić.
4. **Kontener** (zwany też *kompozytem*) jest elementem posiadającym elementy podrzędne: liście, lub inne kontenery. Kontener nie zna konkretnych klas swojej zawartości. Komunikuje się ze wszystkimi elementami podrzędnymi tylko poprzez interfejs komponentu.
5. Otrzymawszy żądanie, kontener deleguje pracę do swoich podelementów, przetwarza wyniki pośrednie i zwraca ostateczny wynik klientowi.
6. **Klient** współpracuje ze wszystkimi elementami za pośrednictwem interfejsu komponentu. W wyniku tego klient może działać w taki sam sposób zarówno na prostych, jak i złożonych elementach drzewa.

Zastosowanie

Stosuj wzorzec Kompozyt gdy musisz zaimplementować drzewiastą strukturę obiektów.

Wzorzec Kompozyt określa dwa podstawowe typy elementów współdzielących jednakowy interfejs: proste liście oraz złożone kontenery. Kontener może być złożony zarówno z liści, jak i z innych kontenerów. Pozwala to skonstruować zagnieżdżoną, rekurencyjną strukturę obiektów przypominającą drzewo.

Stosuj ten wzorzec gdy chcesz, aby kod kliencki traktował zarówno proste, jak i złożone elementy jednakowo.

Wszystkie elementy zdefiniowane przez wzorzec Kompozyt współdzielą jeden interfejs. Dzięki temu, klient nie musi martwić się konkretną klasą obiektów z jakimi ma do czynienia.

Jak zaimplementować

1. Upewnij się, że główny model twojej aplikacji można przedstawić w formie struktury drzewiastej. Spróbuj rozdzielić go na proste elementy i kontenery. Pamiętaj, że kontenery muszą móc zawierać w sobie zarówno proste elementy, jak i inne kontenery.
2. Zadeklaruj interfejs komponentu z listą metod które mają sens zarówno w przypadku prostych, jak i złożonych komponentów.
3. Stwórz klasę-liść reprezentującą proste elementy. Program może posiadać wiele różnych klas-liści.
4. Stwórz klasę-kontener, reprezentującą złożone elementy. W tej klasie umieść pole tablicowe, które przechowywać będzie odniesienia do elementów podrzędnych. Tablica musi być w stanie przechowywać zarówno liście, jak i kontenery, więc zadeklaruj jej typ jako interfejs komponentu.

   Implementując metody interfejsu komponentu, pamiętaj, że kontener ma delegować większość swych obowiązków elementom podrzędnym.
5. Na koniec zdefiniuj metody pozwalające dodawać i usuwać elementy podrzędne w kontenerze.

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

Wzorce behawioralne

Wzorce behawioralne koncentrują się na interakcji i odpowiedzialności między obiektami. Ich celem jest umożliwienie komunikacji między obiektami w sposób, który zmniejsza ich zależność i zwiększa elastyczność. Wzorce te pomagają w definiowaniu, jak obiekty powinny współdziałać oraz jak zlecać odpowiedzialności między nimi.

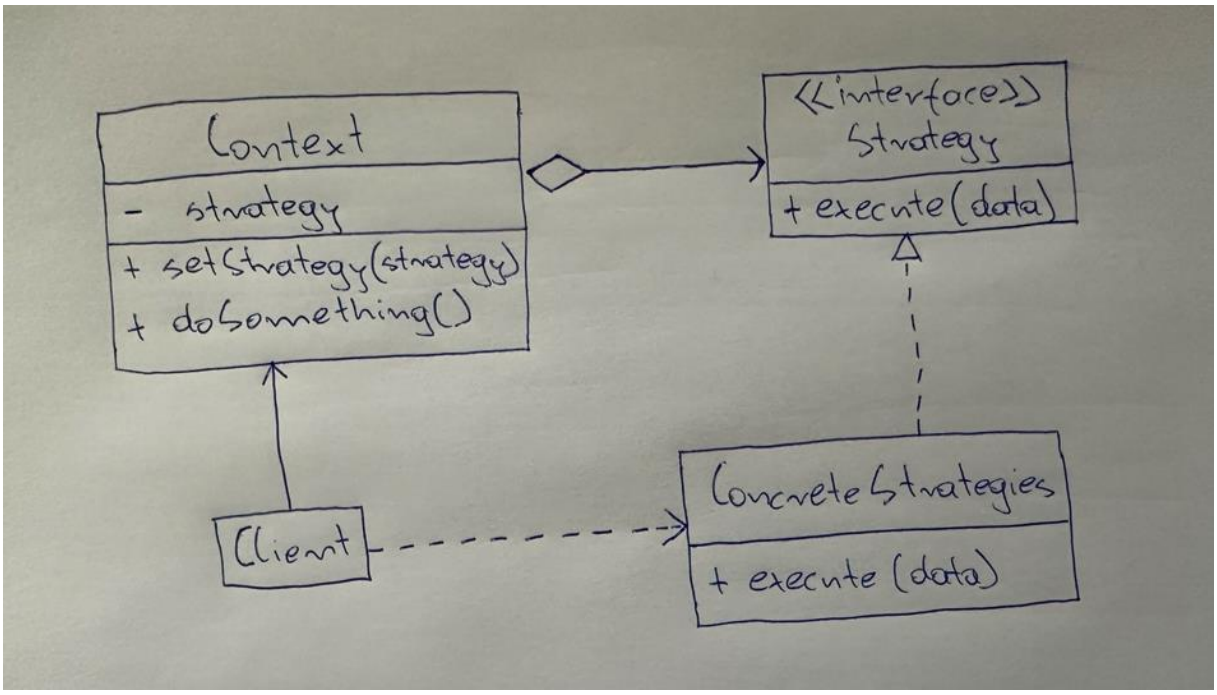
Strategia

Strategia umożliwia definiowanie różnych algorytmów i zachowań w oddzielnych klasach (strategiach) oraz umożliwia dynamiczne zmienianie tych algorytmów w trakcie działania programu. Dzięki temu wzorcowi kontekst (klasa używająca strategii) nie musi znać szczegółów działania konkretnego algorytmu, co zwiększa elastyczność i ułatwia rozbudowę kodu.

W skrócie: Strategia pozwala zamienić algorytmy na wymienne moduły, które mogą być przypisane do obiektu kontekstu i zmieniane bez modyfikacji jego kodu.

Analogia do prawdziwego życia

Wyobraź sobie, że musisz dotrzeć na lotnisko. Możesz złapać autobus, zamówić taksówkę lub pojechać rowerem. To są twoje strategie przejazdu. Możesz wybrać jedną z nich zależnie od czynników takich jak budżet lub ograniczenia czasowe.



1. **Kontekst** przechowuje odniesienie do jednej z konkretnych strategii i komunikuje się z jej obiektem za pośrednictwem interfejsu strategia.
2. Interfejs **Strategia** jest wspólny dla wszystkich konkretnych strategii. Deklaruje metodę za pomocą której kontekst uruchamia daną strategię.
3. **Konkretne Strategie** implementują różne warianty algorytmu z którego korzysta kontekst.
4. **Kontekst** wywołuje metodę uruchamiającą eksponowaną przez powiązany z nim obiekt strategii za każdym razem gdy chce uruchomić algorytm. Kontekst nie wie z jaką strategią ma do czynienia, lub jak działa algorytm.
5. **Klient** tworzy określony obiekt strategii i przekazuje go kontekstowi. Kontekst eksponuje metodę setter pozwalającą klientom zamienić strategię skojarzoną z tym kontekstem w trakcie działania programu.

Zastosowanie

Stosuj wzorzec Strategia gdy chcesz używać różnych wariantów jednego algorytmu w obrębie obiektu i zyskać możliwość zmiany wyboru wariantu w trakcie działania programu.

Wzorzec Strategia pozwala pośrednio zmienić zachowanie obiektu w trakcie działania programu poprzez przypisywanie temu obiektowi różnych podobiektów wykonujących określone poddziałania na różne sposoby.

Warto stosować ten wzorzec gdy masz w programie wiele podobnych klas, różniących się jedynie sposobem wykonywania jakichś zadań.

Wzorzec Strategia umożliwia ekstrakcję różniących się zachowań do odrębnej hierarchii klas i połączenie pierwotnych klas w jedną, redukując tym samym powtórzenia kodu.

Strategia pozwala odizolować logikę biznesową klasy od szczegółów implementacyjnych algorytmów, które nie są istotne w kontekście tej logiki.

Strategia umożliwia odizolowanie kodu różnych algorytmów, ich danych wewnętrznych oraz zależności od reszty kodu. Klienci otrzymują prosty interfejs umożliwiający uruchamianie algorytmów i wymiany ich na inne w trakcie działania programu.

Stosuj ten wzorzec gdy twoja klasa zawiera duży operator warunkowy, którego zadaniem jest wybór odpowiedniego wariantu tego samego algorytmu.

Wzorzec Strategia pozwala pozbyć się wyżej wymienionych kawałków kodu poprzez ekstrakcję algorytmów do odrębnych klas implementujących taki sam interfejs. Pierwotny obiekt deleguje uruchamianie jednemu z tych obiektów, zamiast samodzielnie implementować wszystkie warianty algorytmu.

Jak zaimplementować

1. W klasie kontekstu zidentyfikuj algorytm który często może ulegać zmianom. Może to być też obszerna instrukcja warunkowa wybierająca i uruchamiająca wariant tego samego algorytmu w trakcie działania programu.
2. Zadeklaruj interfejs strategii który będzie wspólny dla wszystkich wariantów algorytmu.
3. Jeden po drugim ekstrahuj wszystkie algorytmy do odrębnych klas implementujących interfejs strategii.
4. W klasie kontekstu, dodaj pole służące przechowywaniu odniesienia do obiektu strategii. Przygotuj metodę setter służącą zmianie wartości tego pola. Kontekst powinien współpracować z obiektem strategii wyłącznie przez interfejs strategii. Kontekst może definiować interfejs dający strategii dostęp do swoich danych.
5. Klienci kontekstu muszą skojarzyć go ze stosowną strategią, która odpowiada sposobowi w jaki kontekst ma wykonać swoje zadanie.

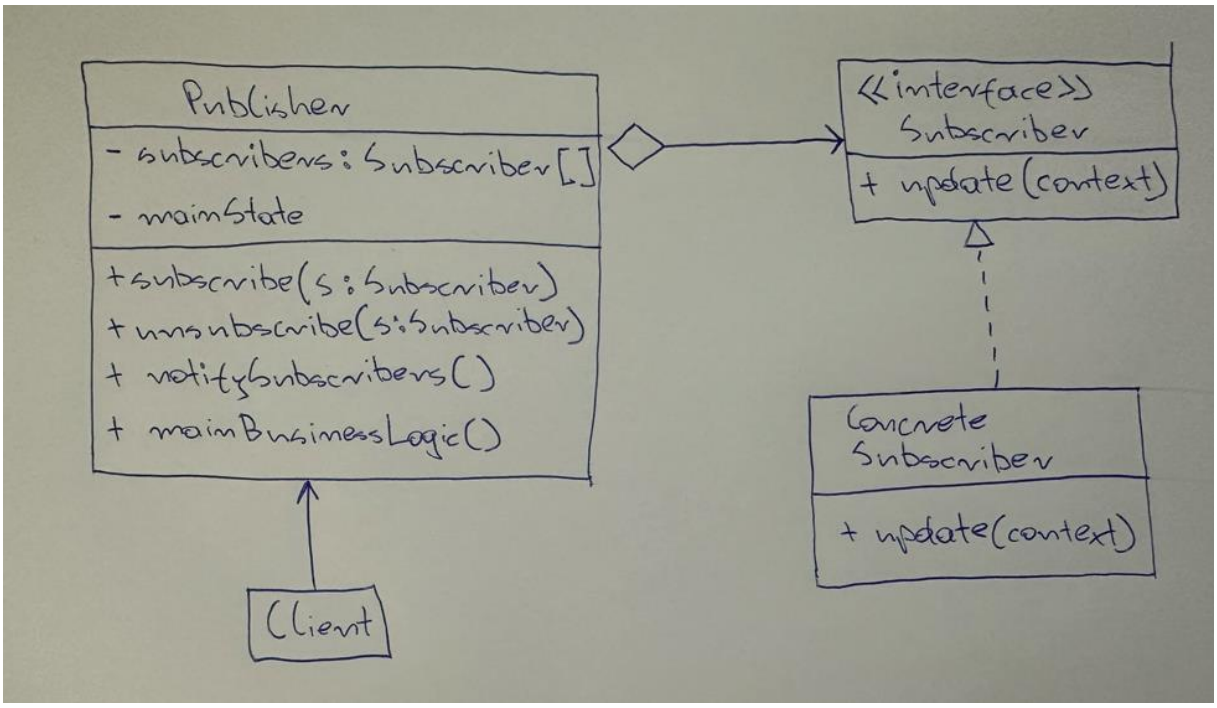
Obserwator

Obserwator to czynnościowy (behawioralny) wzorzec projektowy pozwalający zdefiniować mechanizm subskrypcji w celu powiadamiania wielu obiektów o zdarzeniach dziejących się w obserwowanym obiekcie.

Analogia do prawdziwego życia

Jeśli subskrybujesz czasopismo lub gazetę, nie musisz więcej chodzić do sklepu by sprawdzić czy jest już nowy numer. Zamiast tego, wydawca przysyła ci nowe egzemplarze pocztą od razu po opublikowaniu, a czasem nawet trochę wcześniej.

Wydawca zarządza listą subskrybentów i wie które czasopisma kogo interesują. Subskrybenci mogą wypisać się z listy kiedy nie chcą już otrzymywać kolejnych edycji.



1. **Publikujący** rozsyła zdarzenia interesujące inne obiekty. Zdarzenia te zachodzą gdy publikujący zmienia swój stan lub wykona jakieś obowiązki. Publikujący posiadają infrastrukturę dającą możliwość subskrybowania ich zdarzeń lub przerwania subskrypcji.
2. Gdy nastąpi nowe zdarzenie, nadawca przegląda listę subskrybentów i wywołuje metody powiadamiania zadeklarowane w ich interfejsach.
3. Interfejs **Subskrybenta** deklaruje interfejs powiadamiania. W większości przypadków składa się on z jednej metody aktualizuj. Metoda ta może przyjmować wiele parametrów pozwalających publikującemu przekazać za ich pomocą szczegóły dotyczące aktualizacji.
4. **Konkretni Subskrybenci** wykonują jakieś czynności w odpowiedzi na powiadomienia wysłane przez publikującego. Wszystkie te klasy muszą implementować ten sam interfejs, aby nadawca nie musiał być sprzęgnięty z ich konkretnymi klasami.

5. Zazwyczaj, subskrybenci potrzebują jakichś kontekstowych informacji aby poprawnie obsłużyć aktualizacje. Dlatego publikujący na ogół przekazują dane kontekstowe jako argumenty metody powiadamiania. Publikujący może przekazać samego siebie jako argument, umożliwiając subskrybentom pobranie sobie potrzebnych danych bezpośrednio.
6. **Klient** tworzy obiekty publikujące i subskrybujące osobno, po czym rejestruje subskrybentów by mogli otrzymywać aktualizacje od publikującego.

Zastosowanie

Stosuj wzorzec Obserwator gdy zmiany stanu jednego obiektu mogą wymagać zmiany w innych obiektach, a konkretny zestaw obiektów nie jest zawczasu znany lub ulega zmianom dynamicznie.

Można często natknąć się na ten problem pracując z klasami graficznego interfejsu użytkownika. Przykładowo, stworzyliśmy własne klasy przycisków i chcemy aby klienci mogli podpiąć jakiś własny kod do twoich przycisków, aby uruchamiał się po ich naciśnięciu.

Wzorzec Obserwator pozwala każdemu obiektowi implementującemu interfejs subskrypcji otrzymywać powiadomienia o zdarzeniach w obiektach publikujących. Można dodać mechanizm subskrypcji do swoich przycisków, pozwalając klientom na podłączenie do przycisków ich kodu za pośrednictwem własnych klas subskrybentów.

Stosuj ten wzorzec gdy jakieś obiekty w twojej aplikacji muszą obserwować inne, ale tylko przez jakiś czas lub w niektórych przypadkach.

Lista subskrybentów jest dynamiczna, więc subskrybenci mogą zapisać się lub wypisać z listy kiedy chcą.

Jak zaimplementować

1. Przejrzyj logikę biznesową swojego programu i spróbuj podzielić ją na dwie części. Główną funkcjonalność, która jest niezależna od innego kodu, uczynimy obiektem publikującym. Reszta natomiast zostanie przekształcona na zestaw klas subskrybujących.
2. Zadeklaruj interfejs subskrybenta. W najprostszej postaci powinien deklarować pojedynczą metodę aktualizuj.
3. Zadeklaruj interfejs publikujący i opisz metody służące do dodawania i usuwania subskrybentów z listy. Pamiętaj, że obiekty publikujące muszą współdziałać z subskrybentami wyłącznie poprzez interfejs subskrybenta.
4. Zdecyduj gdzie umieścić faktyczną listę subskrybentów oraz implementację metod zarządzających nią. Zazwyczaj taki kod wygląda jednakowo dla wszystkich typów obiektów publikujących, więc najbardziej oczywistym miejscem wydaje się klasa abstrakcyjna wywodząca się bezpośrednio z interfejsu publikującego. Konkretni publikujący rozszerzają tę klasę, dziedzicząc funkcjonalność subskrypcji.

Jednak jeśli stosujesz ten wzorzec w kontekście istniejącej hierarchii klas, rozważ podejście bazujące na kompozycji: umieść logikę subskrypcji w osobnym obiekcie i pozwól by wszyscy publikujący z niej korzystali.

5. Stwórz konkretne klasy publikujące. Za każdym razem gdy zdarzy się coś ważnego w obiekcie publikującym, musi on powiadomić o tym swoich subskrybentów.

6. Zaimplementuj metody powiadamiania o aktualizacji w konkretnych klasach subskrybentów. Większość subskrybentów będzie potrzebować jakichś danych kontekstowych o zdarzeniu. Można je przekazać w formie argumentu metody powiadamiania.

   Istnieje jednak jeszcze jedna opcja. Otrzymawszy powiadomienie, subskrybent może pobrać dane bezpośrednio od powiadomienia. W takim przypadku, obiekt publikujący musi przekazać samego siebie za pośrednictwem metody aktualizacji. Mniej elastyczną opcją jest powiązanie obiektu publikującego z subskrybentem na stałe za pośrednictwem konstruktora.

7. Klient musi stworzyć wszystkich niezbędnych subskrybentów i zarejestrować ich u odpowiednich publikujących.

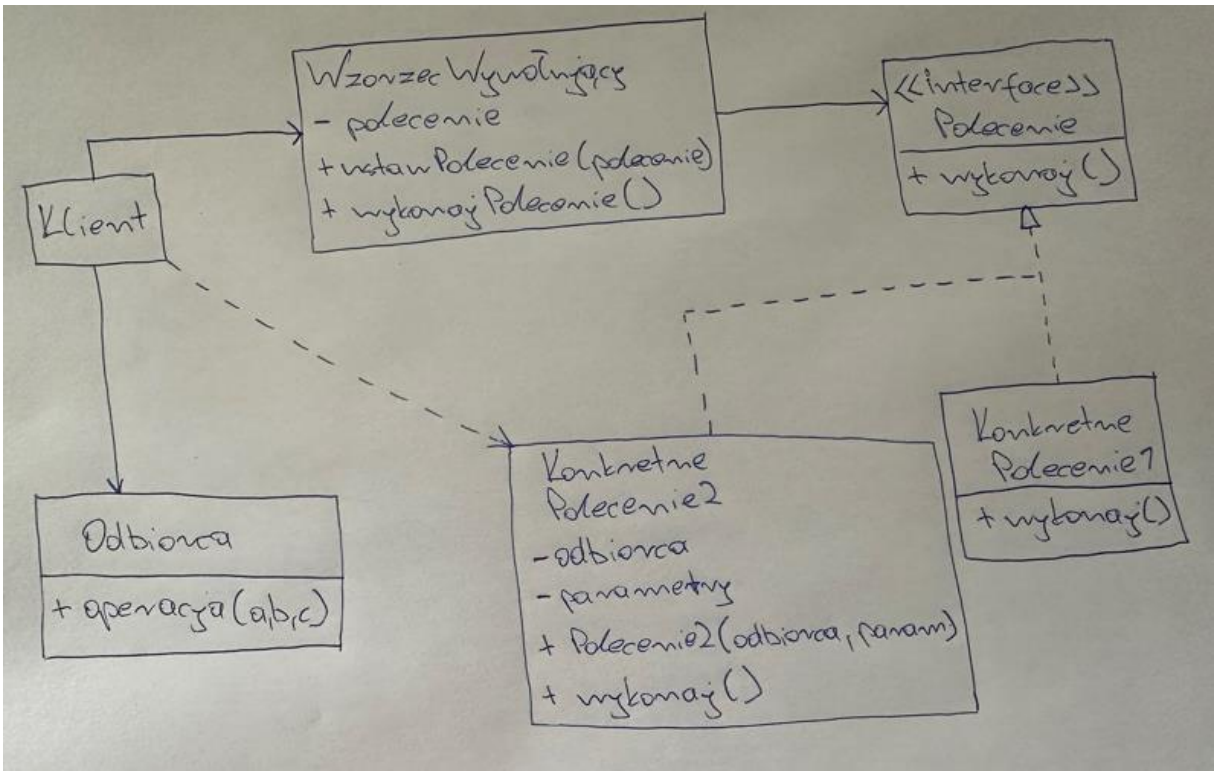
Polecenie

Polecenie (Command) służy do przekształcenia żądań w obiekty, dzięki czemu możemy oddzielić obiekt, który wydaje polecenie, od obiektu, który to polecenie wykonuje. Pozwala to m.in. na przechowywanie, kolejkowanie, logowanie poleceń oraz ich cofanie i ponowne wykonywanie.

Analogia do prawdziwego życia

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

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



1. Klasa **Nadawca** (lub *wywołująca*) jest odpowiedzialna za inicjowanie żądań. Musi ona zawierać pole przechowujące odniesienia do obiektu polecenia. Nadawca uruchamia polecenie zamiast przesyłać żądanie bezpośrednio do odbiorcy. Zauważ, że nadawca nie jest odpowiedzialny za tworzenie obiektu polecenie. Zazwyczaj otrzymuje wcześniej przygotowane polecenie od klienta za pośrednictwem konstruktora.
2. Interfejs **Polecenie** zwykle deklaruje pojedynczą metodę służącą wykonaniu polecenia.
3. **Konkretne polecenia** implementują różne rodzaje żądań. Konkretne polecenie nie powinno wykonywać pracy samodzielnie, lecz przekazać je do jednego z obiektów logiki biznesowej. Jednak dla uproszczenia kodu, klasy te można złączyć.
4. Parametry potrzebne do uruchomienia metody na obiekcie odbiorcy można zadeklarować w formie pól konkretnego polecenia. Obiekty poleceń można uczynić niezmienialnymi, zezwalając na inicjalizację tych pól wyłącznie za pośrednictwem konstruktora.
5. Klasa **Odbiorca** zawiera jakąś logikę biznesową. Prawie każdy obiekt może pełnić rolę odbiorcy. Większość poleceń obsługuje tylko szczegóły przekazania żądania do odbiorcy, zaś faktyczną pracę wykonuje ten ostatni.
6. **Klient** tworzy i konfiguruje konkretne obiekty żądań. Klient musi przekazać wszystkie parametry żądania, włącznie z instancją odbiorcy, do konstruktora polecenia. Następnie otrzymane polecenie można skojarzyć z jednym lub wieloma nadawcami.

Zastosowanie

Zastosuj wzorzec Polecenie gdy chcesz parametryzować obiekty za pomocą działań.

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

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

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

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

Stosuj wzorzec Polecenie gdy chcesz zaimplementować operacje odwracalne.

Chociaż istnieje wiele sposobów na implementację funkcjonalności cofnij/ponów, wzorzec Polecenie jest prawdopodobnie najpopularniejszym.

Aby móc wycofywać działania, trzeba zaimplementować historię wykonanych działań. Historia poleceń jest stosem zawierającym wszystkie obiekty wykonanych poleceń wraz ze skojarzonymi z nimi kopiami zapasowymi stanu aplikacji.

Ta metoda ma dwie wady. Po pierwsze, zapisanie stanu aplikacji może nie być tak proste, gdyż część jej danych może być prywatna. Problem ten można obejść stosując wzorzec Pamiątka.

Po drugie, kopie zapasowe stanów mogą zużywać sporo pamięci RAM. Dlatego czasem można uciec się do alternatywnej implementacji: zamiast przywracać przeszły stan, można wykonać polecenie odwrotne. Takie polecenie również jednak miałoby swoją cenę: może okazać się trudne lub wręcz niemożliwe do zaimplementowania.

Jak zaimplementować

1. Zadeklaruj interfejs polecenia z pojedynczą metodą uruchamiającą.
2. Dokonaj ekstrakcji żądań do konkretnych, odrębnych klas poleceń które implementują interfejs polecenia. Każda klasa powinna mieć zestaw pól służących przechowywaniu argumentów żądania wraz z odniesieniem do faktycznego obiektu odbiorcy. Wszystkie te wartości muszą być inicjalizowane za pośrednictwem konstruktora polecenia.
3. Zidentyfikuj klasy które będą pełnić rolę *nadawców*. Dodaj tym klasom pola służące przechowywaniu poleceń. Nadawcy powinni komunikować się z poleceniami wyłącznie za pośrednictwem interfejsu polecenia. Nadawcy na ogół nie tworzą obiektów polecenie sami, lecz otrzymują je od strony kodu klienta.

4. Zmień nadawców w taki sposób, aby uruchamiali polecenie zamiast wysyłania żądania bezpośrednio do odbiorcy.
5. Klient powinien inicjalizować obiekty w następującej kolejności:
    - Tworzyć odbiorców.
    - Tworzyć polecenia i kojarzyć je z odpowiednimi odbiorcami, jeśli istnieje potrzeba,
    - Tworzyć nadawców i kojarzyć ich z konkretnymi poleceniami.

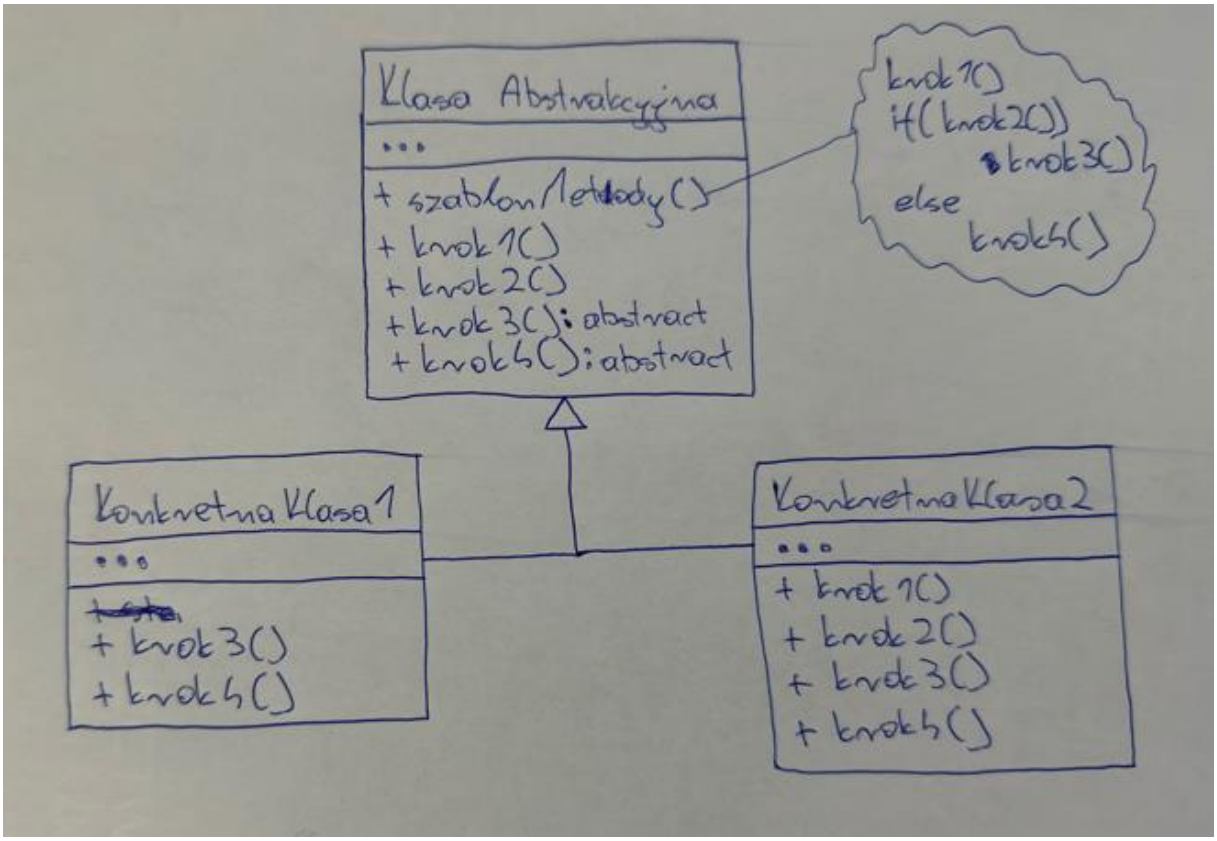
Metoda szablonowa

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

Analogia do prawdziwego życia

Projekt architektoniczny standardowego domu może mieć wiele punktów zaczepienia, które pozwolą potencjalnemu właścicielowi dostosować finalną budowlę do swoich potrzeb.

Każdy etap budowy, jak kładzenie fundamentów, szkielet, wznoszenie ścian, instalację wodno-kanalizacyjna i elektryczną, itd., można nieco zmodyfikować, czyniąc dom odmiennym od reszty.



1. **Klasa Abstrakcyjna** deklaruje metody stanowiące etapy algorytmu oraz faktyczną metodę szablonową która wywołuje poszczególne etapy w określonej kolejności. Etapy mogą być zadeklarowane jako abstrakcyjne, albo posiadać domyślną implementację.
2. **Konkretne Klasy** mogą nadpisać każdy z etapów, oprócz samej metody szablonowej.

Zastosowanie

Stosuj wzorzec Metoda szablonowa gdy chcesz pozwolić klientom na rozszerzanie niektórych tylko etapów algorytmu, ale nie całego, ani też jego struktury.

Metoda szablonowa pozwala zmienić monolityczny algorytm w ciąg pojedynczych etapów, które można następnie łatwo rozszerzać w podklasach bez naruszania struktury opisanej w klasie bazowej.

Wzorzec ten jest przydatny gdy masz wiele klas zawierających niemal identyczne algorytmy różniące się jedynie szczegółami. W takiej sytuacji bowiem konieczność modyfikacji algorytmu skutkuje koniecznością modyfikacji wszystkich klas.

Zmieniając taki algorytm w metodę szablonową możesz także przenieść jego etapy o podobnej implementacji do klasy bazowej, eliminując tym samym duplikację kodu. Kod który różni się pomiędzy podklasami, może w nich pozostać.

Jak zaimplementować

1. Przeanalizuj algorytm docelowy pod kątem możliwego podziału na etapy. Rozważ które z nich są wspólne dla wszystkich podklas, a które zawsze będą unikalne.
2. Stwórz abstrakcyjną klasę bazową i zadeklaruj metodę szablonową oraz zestaw abstrakcyjnych metod reprezentujących etapy algorytmu. Nakreśl strukturę algorytmu w metodzie szablonowej poprzez uruchamianie odpowiednich etapów. Rozważ zastosowanie słowa kluczowego final w stosunku do metody szablonowej aby zapobiec nadpisaniu jej przez podklasy.
3. Może się zdarzyć, że wszystkie etapy pozostaną abstrakcyjnymi. Jednak niektóre z nich skorzystałyby na posiadaniu domyślnej implementacji. Podklasy nie muszą implementować tych metod.
4. Zastanów się nad dodaniem hooków pomiędzy kluczowymi etapami algorytmu.
5. Dla każdego wariantu algorytmu stwórz nową konkretną podklasę. *Musi* ona implementować wszystkie abstrakcyjne etapy, ale *może* także nadpisać część opcjonalnych.

Iterator

Iterator to jeden z fundamentalnych wzorców projektowych, który pozwala na sekwencyjne przechodzenie przez elementy kolekcji (np. listy, zbioru, tablicy) bez ujawniania wewnętrznej struktury danych. Dzięki temu wzorcowi można oddzielić logikę iteracji od samej kolekcji.

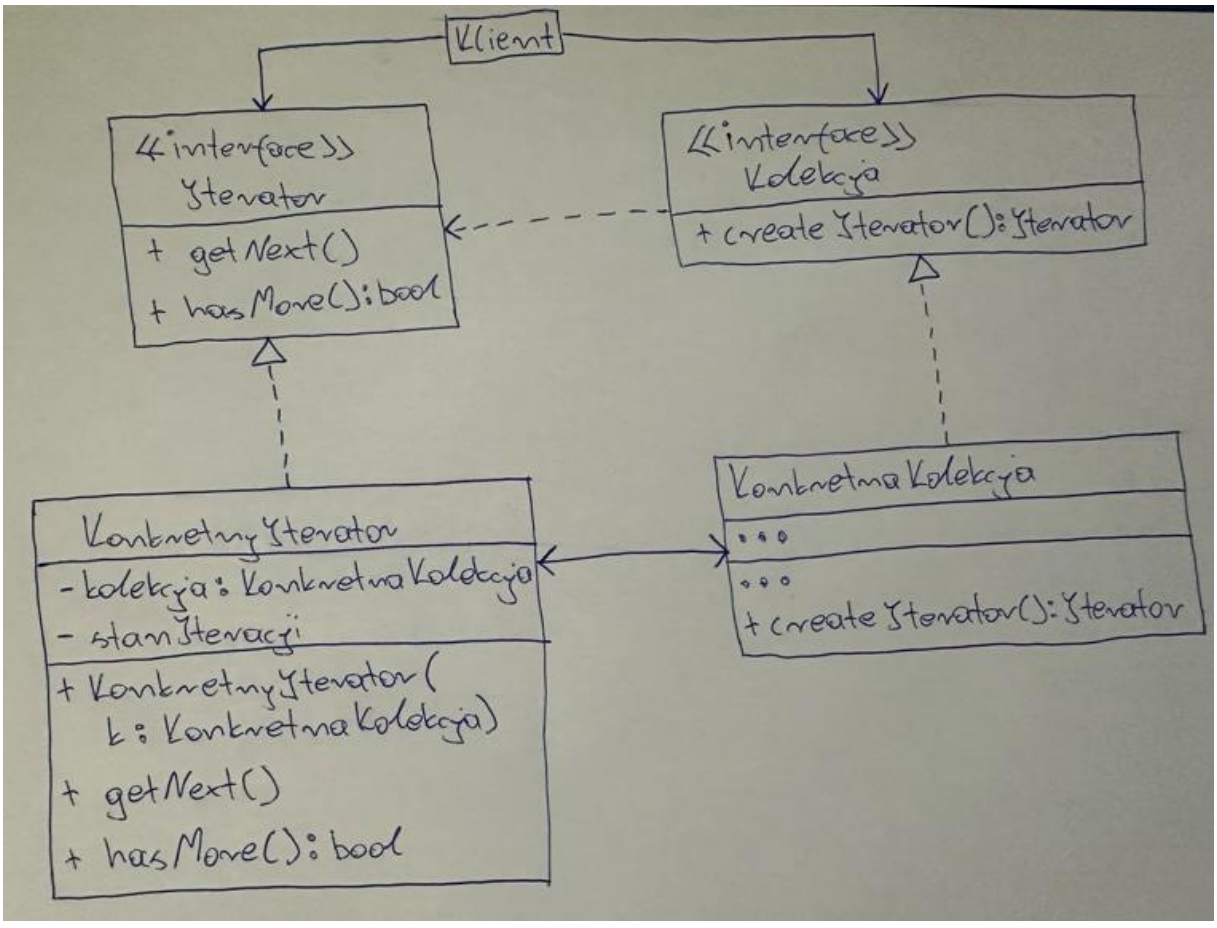
Analogia do prawdziwego życia

Zamierzasz odwiedzić Rzym na parę dni i zwiedzić wszystkie najważniejsze miejsca i atrakcje turystyczne. Ale na miejscu łatwo zmarnować sporo czasu chodząc w kółko, nie mogąc znaleźć nawet Koloseum.

Z drugiej strony, można kupić aplikację wirtualnego przewodnika na smartfon i nawigować z jej pomocą. Sprytne i niedrogie, a dodatkowo można zatrzymać się przy dowolnym miejscu na ile się chce.

Trzecia alternatywa to poświęcenie części swojego wycieczkowego budżetu na wynajęcie przewodnika który zna miasto jak własną kieszeń. Przewodnik mógłby dostosować trasę wycieczki do twoich preferencji, pokazać wszystko co najciekawsze i opowiedzieć wiele ciekawych historii. Byłoby miło, ale również i drożej.

Wszystkie te opcje — losowo obrane kierunki, smartfonowy przewodnik i wynajęty przewodnik — stanowią iteratory pozwalające na dostęp do wielkiej kolekcji widoków i atrakcji Rzymu.



1. Interfejs **Iterator** deklaruje działania niezbędne do sekwencyjnego przechodzenia przez elementy kolekcji: pobieranie kolejnego elementu, ustalenie bieżącej pozycji, powrót do pierwszego elementu, itp.
2. **Konkretne Iteratory** implementują specyficzne algorytmy przeglądania kolekcji. Obiekt iterator powinien samodzielnie śledzić postęp tego procesu. Pozwala to wielu iteratorom na jednoczesne przeglądanie tej samej kolekcji.

3. Interfejs **Kolekcja** deklaruje jedną lub więcej metod służących kompatybilności kolekcji z iteratorami. Zwracany typ metody musi być zadeklarowany jako interfejs iterator, aby konkretne kolekcje mogły zwracać różne rodzaje iteratorów.

4. **Konkretne Kolekcje** zwracają nowe instancje konkretnych klas iteratorów za każdym razem gdy klient ich zażąda. Pewnie ciekawi cię gdzie jest reszta kodu kolekcji? Nie martw się, powinien znajdować się w tej samej klasie. Po prostu te szczegóły nie są istotne dla konkretnego wzorca, więc je pomijamy.

5. **Klient** współpracuje zarówno z kolekcjami jak i iteratorami za pośrednictwem ich interfejsów. Dzięki temu nie jest sprzężony z konkretnymi klasami, pozwalając na pracę z różnymi kolekcjami i operatorami w tym samym kodzie klienta.

6. Zazwyczaj klienci nie tworzą iteratorów sami, lecz otrzymują je od kolekcji. Ale w pewnych przypadkach klient może stworzyć iterator bezpośrednio — na przykład gdy sam definiuje swój specjalny iterator.

Zastosowanie

Stosuj wzorzec Iterator gdy kolekcja z którą masz do czynienia posiada skomplikowaną strukturę, ale zależy ci na ukryciu jej przed klientem (dla wygody, lub dla bezpieczeństwa).

Iterator hermetyzuje szczegóły współpracy ze złożonymi strukturami danych, dając klientowi pewną liczbę prostych metod służących dostępowi do elementów kolekcji. To podejście jest dla klienta wygodne, ale również chroni kolekcję przed nieuważnym lub złośliwym działaniem, którego ryzyko istnieje przy bezpośredniej pracy ze strukturą.

Stosuj wzorzec w celu redukcji duplikowania kodu przeglądania elementów zbiorów na przestrzeni całego programu.

Kod nietrywialnych algorytmów iteracji bywa obszerny. Gdy umieści się go w ramach logiki biznesowej aplikacji, zwykle zaciera główną odpowiedzialność pierwotnego kodu i czyni go trudniejszym do utrzymania. Przeniesienie kodu przeglądania elementów do stosownych iteratorów pomaga uczynić kod aplikacji czystszym i prostszym.

Stosuj Iterator gdy chcesz, aby twój kod był w stanie przeglądać elementy różnych struktur danych, lub gdy nie znasz z góry szczegółów ich struktury.

Wzorzec Iterator udostępnia parę ogólnych interfejsów zarówno kolekcji, jak i iteratorów. Skoro twój kod korzysta z tych interfejsów, to będzie nadal działał, nawet gdy przekażesz mu różne rodzaje kolekcji i iteratorów, o ile implementują te interfejsy.

Jak zaimplementować

1. Zadeklaruj interfejs iteratora. W najprostszym przypadku musi posiadać metodę pobierającą kolejny element kolekcji. Ale dla wygody możesz dodać parę innych metod, np. do pobierania poprzedniego elementu, śledzenia bieżącej pozycji i ustalenia końca procesu iteracji.

2. Zadeklaruj interfejs kolekcji i opisz metodę pobierającą iteratory. Typ zwracany przez metodę powinien być zgodny z interfejsem iteratora. Możesz zadeklarować podobne metody, jeśli zamierzasz mieć wiele różnych grup iteratorów.
3. Zaimplementuj konkretne klasy iterator dla kolekcji które powinny być możliwe do przeglądania za pomocą iteratorów. Obiekt iterator musi być powiązany z jedną instancją kolekcji. Zazwyczaj tworzy się takie powiązanie w konstruktorze iteratora.
4. Zaimplementuj interfejs kolekcji w swoich klasach kolekcji. Główną ideą jest udostępnienie klientowi skrótu do tworzenia iteratorów optymalnych dla danej klasy kolekcji. Obiekt kolekcji musi przekazywać siebie samego do konstruktora iteratora aby stworzyć między nimi powiązanie.
5. Przejrzyj swój kod klienta i zamień cały kod przeglądania kolekcji na wykorzystujący iteratory. Klient pobiera nowy obiekt iteratora za każdym razem gdy chce przejrzeć zawartość kolekcji.

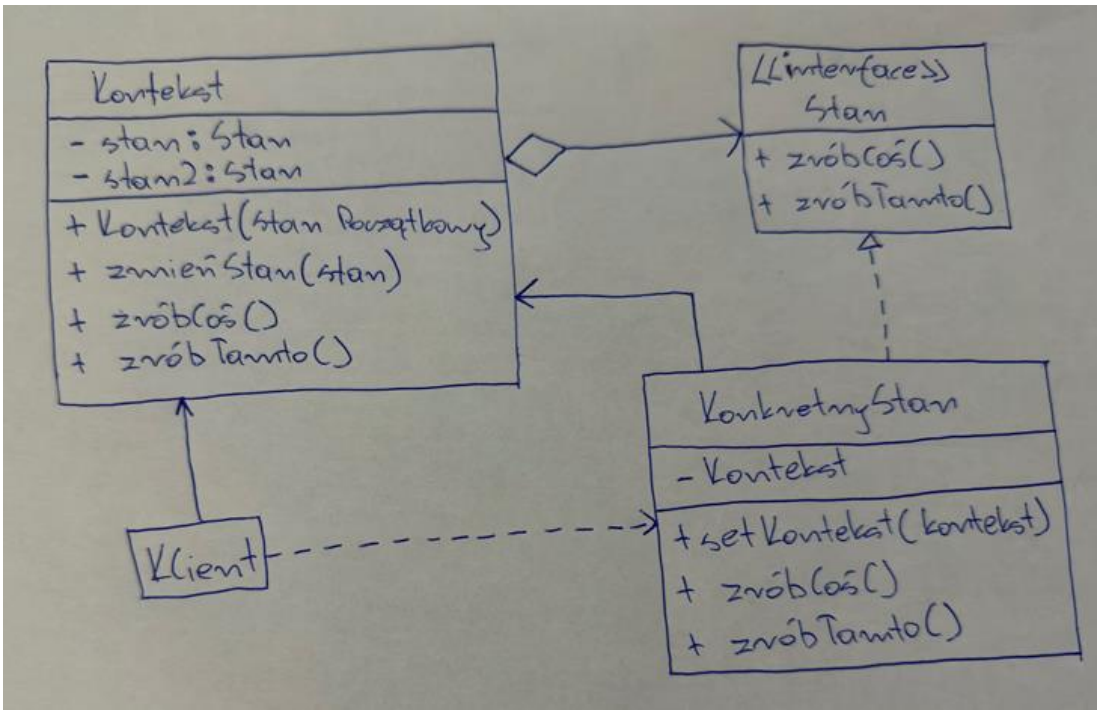
Stan

Stan pozwala obiektowi zmieniać swoje zachowanie, gdy zmienia się jego stan wewnętrzny, co sprawia, że obiekt zachowuje się jakby należał do innej klasy. Dzięki temu, zamiast stosować rozbudowane instrukcje warunkowe zależne od aktualnego stanu obiektu, możemy przełączać się pomiędzy różnymi stanami reprezentowanymi jako osobne obiekty, z których każdy realizuje określone zachowanie.

Analogia do prawdziwego życia

Przyciski i przełączniki w twoim smartfonie zachowują się w różny sposób, w zależności od bieżącego stanu urządzenia:

- Gdy telefon jest odblokowany, wciskanie przycisków wywołuje różne funkcje.
- Gdy telefon jest zablokowany, wciskanie przycisków wyświetli ekran służący odblokowaniu.
- Gdy bateria telefonu jest na wyczerpaniu, wciśnięcie dowolnego przycisku wyświetli ekran ładowania.



1. **Kontekst** przechowuje odniesienie do jednego z konkretnych obiektów-stanów i deleguje mu zadania specyficzne dla danego stanu. Kontekst porozumiewa się z obiektem stanu za pośrednictwem interfejsu stanu. Kontekst eksponuje metodę setter, przez którą przekazuje się obiekt nowego stanu dla kontekstu.
2. Interfejs **Stanu** deklaruje metody specyficzne dla stanu. Metody te powinny być sensowne dla konkretnych stanów, ponieważ nie chcemy, aby któreś ze stanów posiadały bezużyteczne metody które nie zostaną nigdy wywołane.
3. **Konkretne Stany** dostarczają swoje implementacje metod specyficznych dla poszczególnych stanów. Aby uniknąć powtórzeń podobnego kodu w wielu stanach, można utworzyć pośrednie klasy abstrakcyjne które hermetyzują jakieś wspólne zachowania.
4. Obiekty stanu mogą przechowywać referencje wsteczne do obiektu kontekst. Za pośrednictwem tego odniesienia stan może pobrać dowolne informacje z obiektu kontekst, a także zainicjować zmianę stanu.
5. Zarówno kontekst, jak i konkretne stany mogą ustawić kolejny stan kontekstu i wykonać samą zmianę stanu poprzez zamianę obiektu stanu powiązanego z kontekstem na inny.

Zastosowanie

Stosuj wzorzec Stan gdy masz do czynienia z obiektem którego zachowanie jest zależne od jego stanu, liczba możliwych stanów jest wielka, a kod specyficzny dla danego stanu często ulega zmianom.

Wzorzec proponuje ekstrakcję całego kodu właściwego poszczególnym stanom do zestawu osobnych klas. W wyniku tego można będzie dodawać nowe stany lub zmieniać istniejące niezależnie od siebie, zmniejszając koszty utrzymania.

Stosuj ten wzorzec gdy masz klasę zaśmieconą rozbudowanymi instrukcjami warunkowymi zmieniającymi zachowanie klasy zależnie od wartości jej pól.

Wzorzec Stan pozwala wyekstrahować rozgałęzienia tych instrukcji warunkowych do metod które znajdą się w klasach reprezentujących poszczególne stany. W ten sposób uprzątnąć można przy okazji tymczasowe pola i metody pomocnicze związane z kodem odnoszącym się do stanów.

Wzorzec Stan pomaga poradzić sobie z dużą ilością kodu który się powtarza w wielu stanach i przejściach między stanami automatu skończonego, bazującego na instrukcjach warunkowych.

Wzorzec Stan pozwala komponować hierarchie klas stanów i zmniejszyć ilość powtórzeń kodu poprzez ekstrakcję wspólnego kodu do abstrakcyjnych klas bazowych.

Jak zaimplementować

1. Zdecyduj która klasa będzie pełniła rolę kontekstu. Może to być istniejąca klasa zawierająca już jakiś kod zależny od stanu obiektu, ale może to być także nowa klasa, jeśli kod specyficzny dla stanów jest rozrzucony po wielu klasach.
2. Zadeklaruj interfejs stanu. Mimo że może on odzwierciedlać wszystkie metody zadeklarowane w kontekście, skup się tylko na tych, które dotyczą zachowania specyficznego dla danego stanu.
3. Dla każdego faktycznego stanu stwórz klasę wywodzącą się z interfejsu stanu. Następnie przejrzyj metody kontekstu i wyekstrahuj cały kod dotyczący tego stanu do nowo utworzonej klasy.

Przenosząc kod do klasy stanu możesz zauważyć, że zależy on od prywatnych składowych klasy kontekstu. Można sobie z tym poradzić w następujący sposób:

   - Uczyń te pola lub metody publicznymi.
   - Zmień ekstrahowane zachowanie na publiczną metodę kontekstu i wywołuj ją z klasy stanu. To brzydkie, ale szybkie rozwiązanie, które można później naprawić.
   - Zagnieźdź klasy stanów w klasie kontekstu, ale tylko jeśli używany język programowania obsługuje zagnieżdżanie klas.

4. W klasie kontekstu dodaj pole przechowujące odniesienie do interfejsu stanu i publicznie dostępną metodę setter, która umożliwia nadpisanie wartości tego pola.
5. Przejrzyj metodę kontekstu raz jeszcze i zamień puste instrukcje warunkowe dotyczące stanu na wywołania stosownych metod obiektu stanu.
6. Aby przełączyć stan kontekstu, utwórz instancję jednej z klas stanu i przekaż ją kontekstowi. Można tego dokonać w ramach samego kontekstu, w którymś ze stanów, bądź po stronie klienta. Za każdym razem, gdy to się dzieje, klasa staje się zależna od konkretnej klasy stanu której instancja powstaje.