Proszę opisać wzorce projektowe i napisać jakie mają zastosowanie w .NET Core: 2 pkt.

Kompozyt

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

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.

Zalety

- Można pracować ze skomplikowanymi strukturami drzewiastymi w wygodny sposób: wykorzystaj na swoją korzyść polimorfizm i rekursję.
- *Zasada otwarte/zamknięte*. Możesz wprowadzać do programu obsługę nowych typów elementów bez psucia istniejącego kodu, gdyż pracuje on teraz z drzewem różnych obiektów.

Wady

- Ustalenie wspólnego interfejsu dla klas o diametralnie różnych funkcjonalnościach może okazać się trudne. W pewnych przypadkach trzeba przesadnie uogólnić interfejs komponentu, co uczyni go trudniejszym do zrozumienia.

Obserwator

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

Obiekt który posiada jakiś interesujący stan nazywa się zwykle *podmiotem*, ale skoro będzie powiadamiał inne obiekty o zmianach swojego stanu, można nazwać go *publikującym*. Wszystkie pozostałe obiekty, które chcą śledzić zmiany stanu nadawcy nazywa się *subskrybentami*.

Wzorzec Obserwator proponuje dodanie mechanizmu subskrypcji do klasy publikującego, aby pojedyncze obiekty mogły subskrybować lub przerwać subskrypcję strumienia zdarzeń publikującego. Na szczęście nie jest to tak skomplikowane jak brzmi. Tak naprawdę mechanizm ten składa się z 1) pola tablicowego służącego przechowywaniu listy odniesień do subskrybentów oraz 2) wielu metod publicznych pozwalających dodawać i usuwać wpisy tej listy.

Za każdym razem, gdy wydarzy się coś ważnego publikującemu, może on przejrzeć swoją listę subskrybentów i wywołać odpowiednią metodę powiadamiania ich obiektów.

Prawdziwe aplikacje mogą mieć tuziny różnych klas subskrybentów które są zainteresowane śledzeniem zdarzeń jednej klasy publikującego. Nie chcielibyśmy sprzęgać nadawcy z tymi wszystkimi klasami. Poza tym możesz nawet z góry nic o nich nie wiedzieć, jeśli klasę publikującą zamierzasz udostępnić innym ludziom.

Dlatego właśnie ważnym jest, aby wszyscy subskrybenci implementowali ten sam interfejs i żeby publikujący komunikował się z nimi wyłącznie poprzez ten interfejs. Powinien on deklarować metodę powiadamiania wraz z zestawem parametrów za pomocą których publikujący może przekazać dodatkowe dane kontekstowe wraz z powiadomieniem.

Jeśli twoja aplikacja ma wiele różnych typów nadawców i chcesz uczynić swoich subskrybentów kompatybilnymi z każdym z typów, możesz pójść o krok dalej i zmusić publikujących do korzystania z tego samego interfejsu. Taki interfejs musiałby opisywać tylko kilka metod subskrybowania. Interfejs umożliwiłby subskrybentom obserwację stanów obiektu publikującego bez konieczności sprzęgania z ich konkretnymi klasami

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.

ZALETY

- *zasada otwarte/zamknięte*. Można wprowadzać do programu nowe klasy subskrybujące bez konieczności zmieniania kodu publikującego (i odwrotnie, jeśli istnieje interfejs publikujący).
- Można utworzyć związek pomiędzy obiektami w trakcie działania programu.

WADY

Subskrybenci powiadamiani są w przypadkowej kolejności.

Strategia

Strategia to behawioralny wzorzec projektowy pozwalający zdefiniować rodzinę algorytmów, umieścić je w osobnych klasach i uczynić obiekty tych klas wymienialnymi.

Wzorzec Strategia proponuje ekstrakcję poszczególnych algorytmów wykonujących dane zadanie na różne sposoby i umieszczenie ich w odrębnych klasach, zwanych *strategiami*.

Pierwotna klasa, zwana *kontekstem*, musi zawierać pole służące przechowywaniu odniesienia do którejś ze strategii. Kontekst deleguje pracę powiązanemu obiektowi typu strategia, zamiast wykonywać ją samodzielnie.

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.

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.

ZALETY

- Możesz zamieniać algorytmy stosowane w obrębie obiektu w trakcie działania programu.
- Możesz odizolować szczegóły implementacyjne algorytmu od kodu który z niego korzysta.
- Umożliwia zamianę dziedziczenia na kompozycję.
- *Zasada otwarte/zamknięte*. Możliwe jest wprowadzanie nowych strategii bez konieczności dokonywania zmian w kontekście.

WADY

- Jeśli masz zaledwie kilka algorytmów i rzadko ulegają one zmianie, nie ma wyraźnej potrzeby nadmiernego komplikowania programu przez dodawanie nowych klas i interfejsów związanych z tym wzorcem.
- Klienci muszą być świadomi różnic pomiędzy poszczególnymi strategiami, aby mogli wybrać właściwą.
- Wiele nowoczesnych języków programowania posiada wsparcie dla typów funkcyjnych pozwalających zaimplementować różne wersje algorytmu wewnątrz zestawu anonimowych funkcji. Można następnie korzystać z tych funkcji dokładnie tak jak z obiektów strategia, ale bez konieczności rozbudowy kodu o kolejne klasy i interfejsy.

Metoda Wytwórcza

Metoda wytwórcza jest kreacyjnym wzorcem projektowym, który udostępnia interfejs do tworzenia obiektów w ramach klasy bazowej, ale pozwala podklasom zmieniać typ tworzonych obiektów.

Wzorzec projektowy Metody wytwórczej proponuje zamianę bezpośrednich wywołań konstruktorów obiektów (wykorzystujących operator `new`) na wywołania specjalnej metody *wytwórczej*. Jednak nie przejmuj się tym: obiekty nadal powstają za pośrednictwem operatora `new`, ale teraz dokonuje się to za kulisami — z wnętrza metody wytwórczej. Obiekty zwracane przez metodę wytwórczą często są nazywane *produktami*.

Na pierwszy rzut oka zmiana ta może wydawać się bezcelowa. Przecież przenieśliśmy jedynie wywołanie konstruktora z jednej części programu do drugiej. Ale zwróć uwagę, że teraz możesz nadpisać metodę wytwórczą w podklasie, a tym samym zmienić klasę produktów zwracanych przez metodę.

Istnieje jednak małe ograniczenie: podklasy mogą zwracać różne typy produktów tylko wtedy, gdy produkty te mają wspólną klasę bazową lub wspólny interfejs. Ponadto, zwracany typ metody wytwórczej w klasie bazowej powinien być zgodny z tym interfejsem.

Kod, który wykorzystuje metodę wytwórczą (zwany często kodem *klienckim*) nie widzi różnicy pomiędzy faktycznymi produktami zwróconymi przez różne podklasy. Klient traktuje wszystkie produkty jako abstrakcyjnie pojęty `Transport`. Klient wie także, że wszystkie obiekty transportowe posiadają metodę `dostarczaj`, ale szczegóły jej działania nie są dla niego istotne.

Stosuj Metodę Wytwórczą gdy nie wiesz z góry jakie typy obiektów pojawią się w twoim programie i jakie będą między nimi zależności.
Metoda Wytwórcza oddziela kod konstruujący produkty od kodu który faktycznie z tych produktów korzysta. Dlatego też łatwiej jest rozszerzać kod konstruujący produkty bez konieczności ingerencji w resztę kodu.

Przykładowo, aby dodać nowy typ produktu do aplikacji, będziesz musiał utworzyć jedynie podklasę kreacyjną i nadpisać jej metodę wytwórczą.

Korzystaj z Metody Wytwórczej gdy zamierzasz pozwolić użytkującym twą bibliotekę lub framework rozbudowywać jej wewnętrzne komponenty.
Dziedziczenie jest prawdopodobnie najłatwiejszym sposobem rozszerzania domyślnego zachowania się biblioteki lub frameworku. Ale skąd framework wiedziałby o konieczności zastosowania twojej podklasy, zamiast standardowego komponentu?

Rozwiązaniem jest zredukowanie kodu konstruującego komponenty na przestrzeni frameworku do pojedynczej metody wytwórczej. Trzeba też umożliwić nadpisywanie tej metody, a nie tylko rozbudowywanie samego komponentu.

Sprawdźmy, jak by to wyglądało. Wyobraź sobie, że piszesz aplikację korzystając z open source'owego frameworku interfejsu użytkownika (UI). Twoja aplikacja ma posiadać okrągłe przyciski, ale framework oferuje jedynie prostokątne. Rozszerzasz więc standardową klasę `Przycisk` o podklasę `OkrągłyPrzycisk`. Ale teraz trzeba również sprawić, by główna klasa `UIFramework` korzystała z nowo utworzonej podklasy, zamiast z domyślnej.

Aby to osiągnąć, tworzysz podklasę `UIZOkrągłymiPrzyciskami` z bazowej klasy frameworku i nadpisujesz jej metodę `stwórzPrzycisk`. Metoda ta będzie zwracać obiekty klasy `Przycisk` w klasie bazowej, zaś twoja jej podklasa zwróci obiekty `OkrągłyPrzycisk`. Od teraz skorzystasz z klasy `UIZOkrągłymiPrzyciskami` zamiast klasy `UIFramework`. I to tyle!

Korzystaj z Metody wytwórczej gdy chcesz oszczędniej wykorzystać zasoby systemowe poprzez ponowne wykorzystanie już istniejących obiektów, zamiast odbudowywać je raz za razem.
Powyższa sytuacja może wyłonić się na pierwszy plan, gdy mamy do czynienia z dużymi obiektami, wymagającymi sporej ilości zasobów. Mogą do nich należeć połączenia do bazy danych, systemy plików oraz zasoby sieciowe.

ZALETY

- Unikasz ścisłego sprzęgnięcia pomiędzy twórcą a konkretnymi produktami.
- *Zasada pojedynczej odpowiedzialności*. Możesz przenieść kod kreacyjny produktów w jedno miejsce programu, ułatwiając tym samym utrzymanie kodu.
- *Zasada otwarte/zamknięte*. Możesz wprowadzić do programu nowe typy produktów bez psucia istniejącego kodu klienckiego.

WADY

Kod może się skomplikować, ponieważ aby zaimplementować wzorzec, musisz utworzyć liczne podklasy. W najlepszej sytuacji wprowadzisz ów wzorzec projektowy do już istniejącej hierarchii klas kreacyjnych.

Dekorator

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

1. **Komponent** deklaruje interfejs wspólny zarówno dla nakładek, jak i opakowywanych obiektów.
2. **Konkretny Komponent** to klasa opakowywanych obiektów. Definiuje ona podstawowe zachowanie, które następnie można zmieniać za pomocą dekoratorów.
3. 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.
4. **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.
5. **Klient** może opakowywać komponenty w wiele warstw dekoratorów, o ile działa na wszystkich obiektach poprzez interfejs komponentu.

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.

Zalety

- Można rozszerzać zachowanie obiektu bez tworzenia podklasy.
- Można dodawać lub usuwać obowiązki obiektu w trakcie działania programu.
- Możliwe jest łączenie wielu zachowań poprzez nałożenie wielu dekoratorów na obiekt.
- *Zasada pojedynczej odpowiedzialności*. Można podzielić klasę monolityczną, która implementuje wiele wariantów zachowań, na mniejsze klasy.

Wady

- Zabranie jednej konkretnej nakładki ze środka stosu nakładek jest trudne.
- Trudno jest zaimplementować dekorator w taki sposób, aby jego zachowanie nie zależało od kolejności ułożenia nakładek na stosie.
- Kod wstępnie konfigurujący warstwy może wyglądać brzydko.