Code Review

Wzorzec Command – analiza zastosowania

W projekcie wzorzec Command został użyty, aby każda akcja z menu była zamknięta w osobnej klasie. Obiekty komend są tworzone jednorazowo podczas inicjalizacji programu i przechowywane w strukturze słownika w klasie invokera. Każda komenda reprezentuje konkretną akcję użytkownika, np. dodanie zadania, dodanie notatki lub wyświetlenie listy danych.

W trakcie działania programu nie są tworzone nowe obiekty Command dla kolejnych wywołań danej akcji. Wybranie tej samej opcji menu powoduje wielokrotne wywołanie metody Execute() na tym samym obiekcie komendy. Oznacza to, że liczba obiektów Command w systemie jest stała i nie zwiększa się wraz z kolejnymi operacjami użytkownika.

Zastosowana implementacja odpowiada uproszczonej wersji wzorca Command. Komendy nie przechowują informacji o efektach swojego wykonania, a jedynie posiadają referencje do obiektów realizujących logikę aplikacji, takich jak repozytorium czy builder. Dane tworzone podczas wykonywania komendy są obiektami domenowymi i są przechowywane wyłącznie w repozytorium, natomiast sama komenda nie zapamiętuje ich stanu.

Konsekwencją takiego podejścia jest brak mechanizmu cofania operacji (undo/redo). Ponieważ komendy nie zapisują stanu obiektu przed i po wykonaniu, ani nie są odkładane do historii operacji, nie ma możliwości odtworzenia lub cofnięcia wcześniejszych działań użytkownika. Aby umożliwić undo, konieczne byłoby tworzenie nowych obiektów Command dla każdej konkretnej operacji oraz przechowywanie ich w strukturze historii, wraz z implementacją metody Undo().

Zastosowanie wzorca Command w obecnej formie spełnia swoją podstawową rolę, tj. oddzielenie logiki wyboru akcji w menu od jej realizacji, jednak ogranicza potencjał dalszej rozbudowy aplikacji o funkcje związane z historią operacji.

Wzorzec Builder – analiza zastosowania

W projekcie wzorzec Builder został wykorzystany do konstruowania obiektów domenowych TaskItem oraz NoteItem, które posiadają wiele pól, w tym pola opcjonalne, oraz wymagają walidacji przed utworzeniem. Builder odpowiada za zebranie danych krok po kroku i kontrolę poprawności danych wejściowych, oddzielając proces budowy obiektu od pozostałej logiki aplikacji.

Instancje builderów (TaskBuilder, NoteBuilder) są tworzone jednorazowo podczas inicjalizacji programu i wielokrotnie wykorzystywane przy kolejnych operacjach dodawania danych. Przed każdym użyciem builder jest resetowany poprzez wywołanie metody Reset(), co pozwala usunąć dane pozostałe po poprzednim procesie budowania. Następnie komenda ustawia kolejne pola obiektu za pomocą metod udostępnianych przez Builder, takich jak Title(), Category(), Tags().

Dopiero wywołanie metody Build() kończy proces budowy obiektu. W jej trakcie builder wykonuje walidację zgromadzonych danych, a następnie deleguje faktyczne utworzenie obiektu domenowego do wzorca Factory. Builder nie przechowuje referencji do utworzonego obiektu i po zakończeniu operacji może zostać ponownie użyty.

Zastosowana implementacja odpowiada uproszczonej, stanowej wersji wzorca Builder. Builder nie jest tworzony per operacja budowania, lecz współdzielony pomiędzy wieloma wywołaniami, co wymaga zachowania dyscypliny w postaci ręcznego resetowania stanu. Takie podejście jest poprawne funkcjonalnie, jednak w przypadku rozbudowy aplikacji może prowadzić do błędów związanych z pozostawieniem nieaktualnych danych w builderze.

Zastosowanie wzorca Builder w tej formie spełnia swoją podstawową rolę, tj. uporządkowanie procesu tworzenia złożonych obiektów i centralizację walidacji, jednak ogranicza bezpieczeństwo i elastyczność rozwiązania w porównaniu do wariantu, w którym builder tworzony jest jako nowa instancja dla każdej operacji budowania.

Wzorzec Factory – analiza zastosowania

W projekcie wzorzec Factory został użyty w celu odseparowania logiki fizycznego tworzenia instancji obiektów od klas odpowiedzialnych za ich konfigurację. Zdefiniowano interfejs IItemFactory, który dostarcza metody do wytwarzania rodziny powiązanych produktów: zadań (TaskItem) oraz notatek (NoteItem). Implementacja fabryki (ItemFactory) jest dostarczana do klas budowniczych (TaskBuilder, NoteBuilder) poprzez konstruktor, co realizuje zasadę odwrócenia zależności.

W trakcie działania programu fabryka funkcjonuje jako element nieprzechowujący żadnych danych. Metody fabryczne, takie jak CreateTask, przyjmują kompletny zestaw parametrów niezbędnych do utworzenia obiektu i natychmiast zwracają nową instancję. Fabryka nie posiada wewnętrznych pól pamięci ani nie zarządza dalszym cyklem życia wytworzonych obiektów. Jej zadanie kończy się w momencie przekazania gotowego produktu do buildera.

Problemy:

1. Metody interfejsu zwracają konkretne typy klas (TaskItem, NoteItem), a nie ich ogólne interfejsy. Wewnątrz metod implementacji następuje bezpośrednie wywołanie konstruktora, bez stosowania złożonej logiki wyboru wariantu produktu. Skutkiem takiego podejścia jest ograniczenie elastyczności rozwiązania. Ponieważ metody zwracają konkretne typy, kod korzystający z fabryki pozostaje powiązany z jedną, specyficzną implementacją modelu danych. Utrudnia to ewentualne wprowadzenie w przyszłości alternatywnych wersji produktów bez konieczności modyfikacji kodu interfejsu.

Sugestie poprawy:

1. Wprowadzenie abstrakcji produktów, czyli dodanie ITaskItem oraz INoteItem i wykorzystanie ich jako typów zwracanych w metodach fabryki.

Zastosowanie wzorca Factory w obecnej formie spełnia swoją kluczową rolę. Pozwala on na wyizolowanie operatora ‘new’ z logiki biznesowej oraz znacząco ułatwia testowanie builderów. Dzięki takiemu podejściu możliwe jest podstawienie w testach sztucznej implementacji fabryki, co porządkuje architekturę aplikacji.

Wzorzec Observer – analiza zastosowania

W projekcie wzorzec Observer został wykorzystany w celu stworzenia reaktywnego modelu komunikacji, w którym warstwa przechowywania danych (InMemoryRepository) automatycznie informuje niezależne moduły o zmianach swojego stanu. Obiekty obserwatorów (takie jak statystyki, powiadomienia czy generator raportów) są tworzone jednorazowo podczas inicjalizacji systemu i rejestrowane w obiekcie obserwowanym.

Działanie mechanizmu opiera się na modelu "Push". W momencie modyfikacji danych (np. dodanie zadania), repozytorium wywołuje metodę Notify(), która iteruje po liście zarejestrowanych obserwatorów i uruchamia ich metodę Update(). Dzięki temu repozytorium nie posiada sztywnej zależności od konkretnych klas logiki nie "wie", kto i w jaki sposób przetworzy informację o nowym zadaniu, co realizuje zasadę luźnych powiązań.

Konkretny obserwator, np. StatisticsObserver, utrzymuje własne, wewnętrzne liczniki (np. totalTasks), które są aktualizowane przy każdym zdarzeniu. Pozwala to na bieżące śledzenie metryk bez konieczności każdorazowego przeliczania całej bazy danych. Przekazywanie danych odbywa się w sposób generyczny metoda Update przyjmuje typ zdarzenia jako ciąg znaków (string eventType) oraz opcjonalny ładunek danych jako object.

Obecna forma wzorca skutecznie realizuje wymaganie otwarte-zamknięte dodanie nowej reakcji na zdarzenie wymaga jedynie stworzenia nowej klasy implementującej IObserver i jej rejestracji, bez konieczności modyfikacji kodu repozytorium.

Wzorzec Composite – analiza zastosowania

W projekcie wzorzec Composite został zastosowany w celu ujednolicenia sposobu traktowania pojedynczych elementów domenowych (zadań i notatek) oraz ich grup (kategorii). Dzięki wprowadzeniu wspólnego interfejsu (np. ITaskComponent), kod kliencki – w tym przypadku komendy listujące – może operować na całej hierarchii obiektów w sposób polimorficzny, nie rozróżniając, czy w danym momencie przetwarza pojedynczy liść (TaskComponent), czy złożony węzeł (TaskCategoryComposite).

Kluczowym aspektem implementacji jest wykorzystanie mechanizmu rekurencji do realizacji operacji na strukturze drzewiastej. Metody takie jak Display(int depth) czy Search(string keyword) są propagowane w dół hierarchii. W przypadku kompozytu (kategorii), metoda iteruje po wszystkich swoich elementach podrzędnych, przekazując im sterowanie, natomiast w przypadku liścia (zadania) – wykonuje właściwą logikę biznesową lub prezentacyjną. Zastosowanie parametru depth pozwala na dynamiczne wizualizowanie poziomu zagnieżdżenia w konsoli poprzez odpowiednie formatowanie wcięć.

Charakterystycznym elementem tej implementacji jest metoda GetContent(), która pełni funkcję mostu między strukturą hierarchiczną a płaskim modelem danych wymaganym przez niektóre algorytmy (np. statystyki). Metoda ta pozwala na rekurencyjne pobranie wszystkich obiektów domenowych z drzewa i zwrócenie ich w formie standardowej listy List<TaskItem>. Dzięki temu repozytorium może korzystać z zalet struktury drzewiastej przy prezentacji danych, zachowując jednocześnie możliwość łatwego filtrowania i przetwarzania danych za pomocą LINQ, jak w klasycznych kolekcjach płaskich.

Zastosowanie wzorca Composite w tej formie znacząco upraszcza kod warstwy prezentacji. Dodanie nowego typu elementu struktury (np. „Podprojekt” lub „Archiwum”) wymaga jedynie implementacji interfejsu komponentu, bez konieczności modyfikacji pętli wyświetlających w klasach komend. Rozwiązanie to zapewnia elastyczność w zarządzaniu strukturą danych, przenosząc odpowiedzialność za sposób traversowania i wyświetlania drzewa na same obiekty składowe.

Obsługa wyjątków – analiza i problemy

W projekcie obsługa wyjątków została zrealizowana w bardzo uproszczony sposób. W praktyce aplikacja wykorzystuje jeden ogólny wyjątek, który sygnalizuje różne rodzaje błędów występujących podczas działania programu, takie jak niepoprawne dane wejściowe, błędy walidacji czy nieudane operacje logiczne.

Takie podejście powoduje, że:

- różne sytuacje błędne są traktowane w identyczny sposób,
- kod wywołujący nie ma możliwości rozróżnienia rodzaju błędu,
- reakcja aplikacji na błąd jest mało precyzyjna.

Brak rozróżnienia wyjątków utrudnia zarówno debugowanie, jak i dalszą rozbudowę systemu. W szczególności nie jest możliwe podjęcie różnych działań w zależności od rodzaju błędu, np. innego komunikatu dla użytkownika przy błędzie walidacji danych, a innego przy błędzie logicznym.

Lepszym rozwiązaniem byłoby wprowadzenie bardziej szczegółowej hierarchii wyjątków

- wyjątki walidacyjne (np. brak wymaganego pola),
- wyjątki infrastrukturalne (np. problem z repozytorium).

Pozwoliłoby to na:

- czytelniejszy kod,
- lepszą kontrolę nad przepływem błędów,
- bardziej precyzyjne komunikaty dla użytkownika,
- łatwiejsze testowanie i debugowanie.

Organizacja kodu – zbyt wiele klas w pojedynczych plikach

W projekcie część plików zawiera zbyt wiele klas w jednym miejscu. Powoduje to obniżoną czytelność kodu i utrudnia szybkie odnajdywanie konkretnej funkcjonalności. W szczególności problem dotyczy plików, które grupują kilka różnych typów klas jednocześnie, np. kilka observerów, kilka komend lub kilka elementów infrastruktury.

Takie podejście ma następujące konsekwencje:

- utrudniona nawigacja po projekcie, szczególnie przy rozbudowie,
- zwiększone ryzyko konfliktów podczas pracy zespołowej (częstsze zmiany w tym samym pliku),
- trudniejsze utrzymanie zasady jednej odpowiedzialności na poziomie pliku, nawet jeśli pojedyncze klasy są poprawnie rozdzielone,
- większa podatność na „rozrost" plików wraz z dodawaniem nowych funkcji.

Lepszą praktyką byłoby rozdzielenie kodu zgodnie z zasadą:

- jedna klasa, jeden plik (szczególnie dla klas publicznych),
- grupowanie w folderach według roli, np. Commands/, Observers/, Builders/, Factories/, Models/, Repository/.

Dzięki temu projekt byłby łatwiejszy do rozwijania i modyfikowania, a odnalezienie konkretnej klasy lub wzorca projektowego zajmowałoby mniej czasu.

Mieszanie warstwy prezentacji (UI) z logiką aplikacji

W projekcie logika aplikacji jest mocno połączona z interfejsem użytkownika, w szczególności z konsolą. W wielu miejscach kod, który powinien zajmować się logiką programu, bezpośrednio korzysta z Console.WriteLine i Console.ReadLine, przez co trudno wyraźnie oddzielić logikę aplikacji od warstwy prezentacji.

Przykładem takiego podejścia są:

- komendy, które oprócz wywoływania logiki aplikacji bezpośrednio obsługują wejście i wyjście użytkownika,
- obserwatorzy, którzy reagując na zdarzenia wypisują komunikaty na konsolę,
- elementy struktur danych (np. komponenty Composite), które posiadają metody odpowiedzialne za wyświetlanie danych.

Takie rozwiązanie działa w prostych aplikacjach konsolowych, jednak w dłuższej perspektywie prowadzi do kilku problemów:

- utrudnione testowanie kodu, ponieważ logika jest zależna od konsoli,
- brak możliwości łatwej zmiany interfejsu użytkownika bez modyfikowania logiki aplikacji,
- trudniejsza analiza i utrzymanie kodu.

Lepszym podejściem byłoby wyraźne oddzielenie warstwy prezentacji od logiki aplikacji, np. poprzez:

- wprowadzenie abstrakcji odpowiedzialnej za komunikację z użytkownikiem (np. interfejs IUserIO),
- przeniesienie odpowiedzialności za formatowanie i wyświetlanie danych do warstwy UI,
- pozostawienie w logice wyłącznie operacji na danych i reagowania na zdarzenia.

Uporządkowany proces tworzenia obiektów dzięki połączeniu Builder i Factory

W projekcie proces tworzenia obiektów domenowych został zaprojektowany w sposób uporządkowany poprzez połączenie wzorców Builder oraz Factory. Builder odpowiada za stopniowe zbieranie danych wejściowych, ich wstępną walidację oraz kontrolę kompletności informacji, natomiast Factory przejmuje odpowiedzialność za faktyczne utworzenie instancji obiektu domenowego.

Takie rozdzielenie ról przynosi kilka istotnych korzyści architektonicznych. Po pierwsze, eliminuje konieczność stosowania rozbudowanych konstruktorów z wieloma parametrami, co znacząco poprawia czytelność kodu i zmniejsza ryzyko błędów przy tworzeniu obiektów. Po drugie, walidacja danych znajduje się w jednym, jasno określonym miejscu, dzięki czemu zasady poprawności obiektów są spójne i łatwe do modyfikacji.

Delegowanie samego tworzenia obiektów do Factory powoduje, że zmiana sposobu instancjonowania modeli domenowych, na przykład dodanie wartości domyślnych, dodatkowej konfiguracji lub nowych typów obiektów, nie wymaga modyfikowania kodu komend ani logiki aplikacji. Builder pozostaje interfejsem procesu budowy, a Factory centralnym punktem odpowiedzialnym za tworzenie instancji.

Takie rozwiązanie zwiększa elastyczność systemu i dobrze przygotowuje go na dalszą rozbudowę. W przypadku rozszerzenia modeli domenowych o nowe pola lub wprowadzenia nowych typów zadań i notatek, zmiany mogą zostać ograniczone do buildera i fabryki, bez naruszania pozostałych elementów aplikacji. Jest to przykład poprawnego zastosowania wzorców projektowych w celu zwiększenia czytelności, modularności i odporności kodu na zmiany.