Notatki do 3. tygodnia kursu Dotneta

Maciek Mielczarek

20 sierpnia 2020

Spis treści

1 Wstęp. Stan aplikacji po 2. tygodniu

U mnie jak i u przynajmniej kilkorga innych kursantów aplikacja ma przynajmniej 1 plik który ma kilkaset linii. Prawdopodobnie można to uznać za wyznacznik bałaganu, potrzeby refactoru i powtarzającego się kodu. Aplikacje jednak w jakimś stopniu działają.

2 Konstruktory

Stworzenie jakiegokolwiek konstruktora powoduje nie tworzenie konstruktora domyślnego.

Słowo this jest do wołania innych konstruktorów tej samej klasy. Wpisujemy je po dwukropku, jak wywołanie funkcji z parametrami, między listą parametrów a ciałem konstruktora. W tym miejscu możemy zamiast this użyć base, by wywołać konkretny konstruktor klasy bazowej.

Można zainicjalizować pola zaraz za wywołaniem konstruktora domyślnego, w nawiasach klamrowych.

3 Przeciążanie metod

W różnych wariantach metody o danej nazwie zwracamy ten sam typ.

Pojawiła się wzmianka o podpowiedziach IDE na temat przeciążonych metod. To całkiem przydatna funkcja, szczególnie gdy nie jesteśmy pewni jakich dokładnie danych potrzebuje dana funkcja, najczęściej napisana przez kogoś innego (np. twórców bibliotek standardowych).

4 Dziedziczenie

O dziedziczenie często pyta się na rozmowach rekrutacyjnych.

Przy dziedziczeniu klasa pochodna musi mieć konstruktory takie jak w klasie bazowej. Inne metody też mogą wołać metody klasy bazowej, w swoich ciałach

poprzez base.Metoda().

W C# każda klasa może dziedziczyć tylko po 1 klasie.

W programowaniu sieciowym często spotyka się klasę bazową AuditableModel z właściwościami mówiącymi kto i kiedy ją stworzył i zmodyfikował. Po tej klasie dziedziczy wszystko co chcemy zapisywać do bazy danych.

5 Polimorfizm

Statyczny – przeciążanie funkcji i wybieranie wprost liczbą i typem argumentów która ma być wołana. Program wybiera funkcję na etapie kompilacji.

Dynamiczny – decyzja który kod zostanie wykonany jest odkładana do momentu wykonania kodu. Do nadpisywania metod z klas bazowych używamy modyfikatorów virtual – przy metodzie (w klasie bazowej) do nadpisania i override przy metodzie (w klasie potomnej) nadpisującej.

Jak to włożyć do swojego projektu? Zrobić klasę bazową z metodą wirtualną i 2 klasy dziedziczące po niej, nadpisujące tą metodę. Następnie trzymać obiekty typów potomnych w zmiennej (prawdopodobnie tablicy zmiennych) zadeklarowanej jako bazowa. Potem wołać tą wspólną metodę i użyć faktu, że dla obiektów każdej z klas potomnych będzie wołana jej wersja funkcji. Czyli jeśli mam w projekcie jakąś metodą wirtualną, to prawdopodobnie używam tego konceptu.

Aha, nie jestem pewien czy było to powiedziane wprost, ale obiekty typów pochodnych można trzymać w zmiennej typu bazowego.

new zamiast override mówi, że to jest nowa metoda, nie związana z metodą w klasie bazowej. To znaczy jeśli mamy obiekt typu pochodnego w zmiennej typu bazowego i wołamy tą "wspólną" metodę, wtedy, przy braku powiązania metody bazowej z pochodną, zostanie wywołana metoda z klasy bazowej.

6 Hermetyzacja

Niektóre pola (albo tylko ich settery) i metody są wewnętrznymi sprawami klasy bądź projektu, więc nie powinny być dostępne z zewnątrz. Hermetyzacja albo inaczej enkapsulacja polega właśnie na takim ustawieniu modyfikatorów dostępu, żeby sprawy wewnętrzne pozostawały wewnątrz.

7 Klasy abstrakcyjne

Modyfikator abstract przy klasie mówi, że nie da się tworzyć obiektów danej klasy. Służą one do tego, żeby po nich dziedziczyć, a więc po to żeby trzymać tam pola, właściwości i metody wspólne dla klas potomnych.

Modyfikator abstract przy metodzie mówi, że tej metody nie da się używać, służy ona tylko do tego, żeby ją nadpisywać w klasach potomnych. Czyli chcemy, żeby wszystkie klasy potomne miały metodę o danej nazwie i sygnaturze (czyli typach przyjmowanych i zwracanych), ale każda z nich może być inna.

Metoda abstrakcyjna nie może mieć ciała.

Nie nadpisanie metody abstrakcyjnej w klasie pochodnej zostanie wychwycone przez kompilator jako błąd.

Modyfikator sealed przy klasie mówi, że nie można po niej dziedziczyć. Ta możliwość jest stosowana bardzo rzadko.

8 Interfejsy

Interfejs (wstawiamy słowo interface zamiast class) to coś podobnego do klas abstrakcyjnych, tyle że:

- Można dziedziczyć po 1 klasie naraz, a implementować wiele interfejsów.
- Interfejsy nie mają konstruktorów.
- Ani pól.
- To nie do końca tak, że interfejs może mieć właściwości. Interfejs może mieć, wyglądające jak właściwości, zapisy mówiące, że klasa implementująca ten interfejs musi mieć dane właściwości. Implementacja takiej "publicznej właściwości" też musi być publiczna.
- Do C#7 włącznie metody interfejsu nie mogą mieć ciał, a od C#8 wprawdzie mogą, ale nie jest jasne czy powinny.
- Domyślnym modyfikatorem dostępu wewnątrz klasy jest private, a wewnątrz interfejsu – public.
- Do C#7 włącznie nie można dawać modyfikatorów dostępu niczemu co jest wewnątrz interfejsu. Od C#8 można, ale nie jest jasne czy należy.

Różnice między interfejsem a klasą abstrakcyjną to częste pytanie na rozmowach rekrutacyjnych.

W Klasach i Interfejsach najpierw deklarujemy właściwości i pola, potem metody.

Standardowo najpierw piszemy interfejsy, a potem klasy.

Można implementować w klasie metodę wymaganą przez interfejs jako wirtualną.

Nazwy interfejsów zaczynamy od "I".

9 Typy generyczne

Znane w C++ jako szablony (templates), np List<JakiśTyp>. Często wykorzystywane w operacjach typu CRUD (create, read, update, delete).

Definiujemy jak inne klasy, tylko z "<T>" od razu za nazwą klasy. W środku naszej klasy generycznej tego T możemy używać jako typu.

Aby ograniczyć w naszej klasie generycznej możliwe typy T, piszemy zaraz za nazwą klasy (i za <T>) "where T jakiś warunek", np. "where T: IFoo", czyli akceptujemy tylko typy implementujące interfejs IFoo albo "where T: class", czyli T musi być klasą (a nie typem prostym).

10 Refaktor

Do zrobienia (jeśli ma to sens w mojej aplikacji):

1. Podział rozwiązania na więcej niż 1 projekt.
2. Dodanie interfejsów do serwisów.
3. Dodanie serwisu i modelu bazowego.

W moim projekcie jest raczej wystarczający bałagan, żeby chwilowo robienie nowych funkcjonalności odłożyć na później.

10.1 Ad.1

Dodatkowe projekty to prawdopodobnie powinny byś biblioteki (Class Library (.Net Standard)). Po stworzeniu projektu usunąć przykładową klasę, dodać zależność projektu głównego od biblioteki i prawdopodobnie zostawić w spokoju "Target Framework" w nowej bibliotece (w pliku csproj). W szczególności nie ustawiać Frameworku bibliotek na ten sam co w projekcie głównym, ponieważ Framework do bibliotek ma nieco inną nazwę i numerację.

Jedna z bibliotek (App) do trzymania serwisów i sterowania logiką aplikacji. W środku folder Abstract do interfejsów i Concrete do implementujących je klas. Miejsce metody inicjalizującej jakiś serwis jest w tym serwisie. Folder App.Managers i w środku Managery (Kontrolery) do komunikacji z użytkownikiem, wołania metod serwisów i decydowania co się dzieje w programie. W programowaniu internetowym serwisów będzie więcej.

Jedna z bibliotek (Domain) do trzymania Modeli, czyli wszystkich klas reprezentujących obiekty, które mogą w przyszłości zostać wrzucone do lub pobrane z bazy danych.

Przy przenoszeniu plików między projektami pamiętać żeby zmienić nazwę przestrzeni nazw.

Folder Helpers zostaje w głównej części projektu.

10.2 Ad.3

Typy bazowe robimy, w miarę możliwości, generyczne.

Folder App.Common do trzymania serwisu bazowego.

Domain.Common do trzymania modelu bazowego (BaseEntity) i AuditableModel z poprzednich lekcji.

Domain.Entity do trzymania reszty modeli.

11 Mój Refaktor

Zacząłem od wydzielenia części kodu do refaktoryzacji (klasa z tekstami). Następnie zaplanowałem co powinno się stać ze wszystkimi częściami kodu z tej klasy. Potem chciałem dodać bibliotekę Domain w folderze obok programu i wtedy okazało się, że program mam w tym samym folderze co rozwiązanie, więc samą refaktoryzację zacząłem od poprawienia tego ustawienia.

Następnie, nadal przy dodawaniu bibliotek, przekonałem się, że Frameworki używane przez biblioteki (netstandard2.1) i aplikacje konsolowe (netcoreapp3.1) mają nieco inne nazwy i osobne numeracje. MonoDevelop, którego używałem wcześniej nie wiedział co to "netstandard3.1" i zanim zorientowałem się że czegoś takiego rzeczywiście nie ma, to zmieniłem IDE na Ridera.

Przy rozbijaniu klasy Texts okazało się, że dziedziczenie enumów działa nieco inaczej niż się spodziewałem, więc potrzebne było korygowanie planów na bieżąco. Przy przenoszeniu kodu w inne miejsca w innej formie bardzo pomocne okazały się występujące w Riderze zakładki (Ctrl + Shift + cyfra żeby stworzyć i Ctrl + cyfra by przywołać), wskazane całej grupie przez Maćka Kukuczkę.

Inne klasy były bardziej rozbite, więc właściwie wystarczyło je poprzenosić w odpowiednie miejsca. Oczywiście po tym przenoszeniu mam wrażenie że nie wszystko jest tam gdzie być powinno, ale przynajmniej nie mam już wszystkiego w jednym miejscu.

Jeszcze jedno miejsce w którym mam duże nagromadzenie kodu to sama klasa Program. Wcześniej powydzielałem z Maina dużo funkcji typu MenuTakie, MenuInne, więc wystarczy stworzyć dla każdej z nich odpowiednie miejsce.