TEZA:

Podział aplikacji na mniejsze projekty (UI, Core, Shared), zgodnie z ideą Clean Architecture znacząco ułatwi podmianę poszczególnych elementów aplikacji, np. całego frontendu.

Zmiany wprowadzone w solucji w ramach podmiany interfejsu aplikacji z WinForms na WPF.

|  |  |
| --- | --- |
|  | Jak widać zmiany nawet ciężko nazwać zmianami, bowiem został jedynie dodany nowy projekt WPF. Nie były wymagane zmiany w pozostałych projektach.  Warto mieć na uwadze, że mowa tutaj o przejściu z WinForms na WPF, mowa więc praktycznie o zmianie całego frameworka. Jest to zmiana duża. |

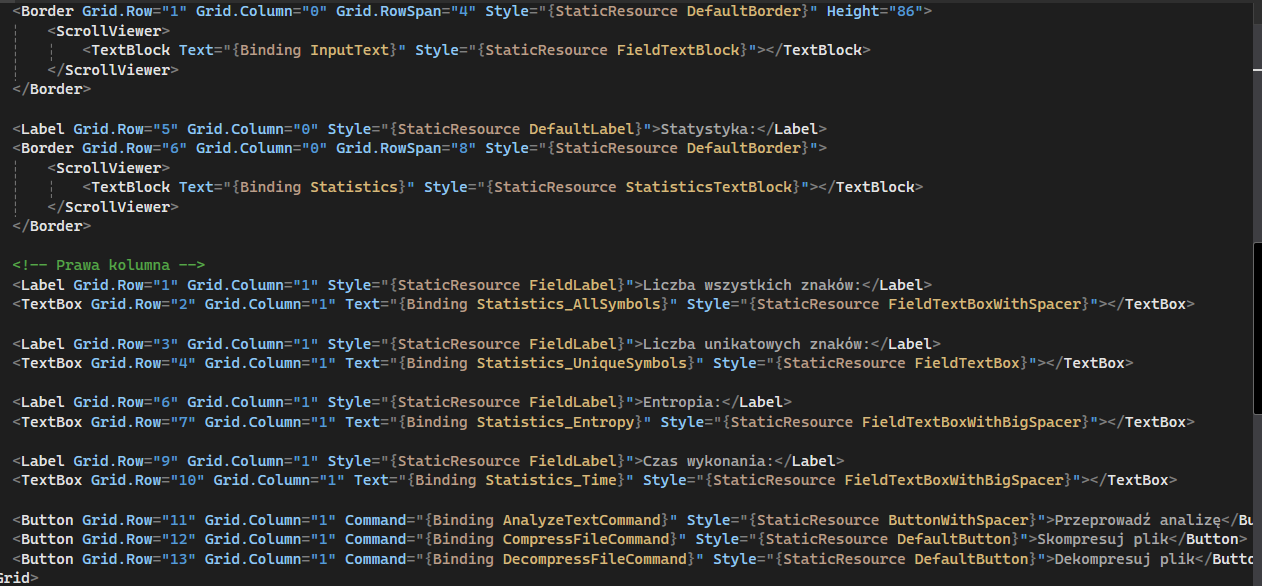
Aplikacja WPF

|  |  |
| --- | --- |
|  |  |

Struktura aplikacji bardzo podobna:

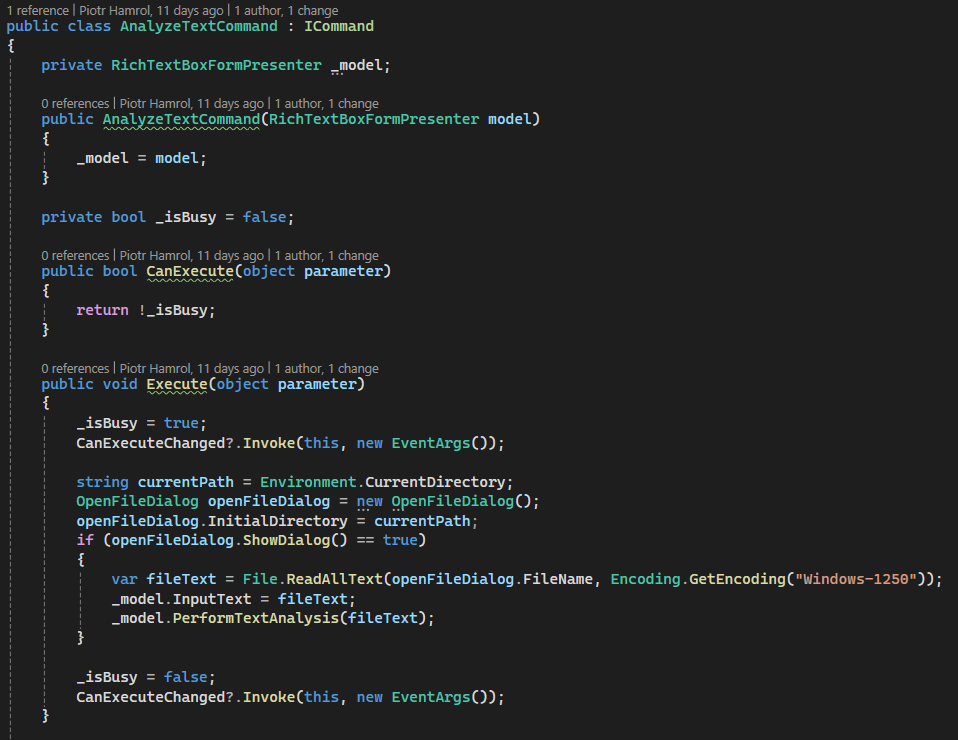
* Okna aplikacji w plikach xaml
* Frontend zrealizowany przy pomocy poleceń Command
* Klasy pośredniczące pomiędzy interfejsem a logiką aplikacji podobnie jak wcześniej umieszczone w folderze Presenters
* Klasy Presenter podpięte do widoków jako ViewModel

Główne okno aplikacji



Wartości wyświetlane na ekranie aplikacji za pośrednictwem mechanizmu Binding

Obsługa ekranu aplikacji zrealizowana nie poprzez zwykłe eventy a za pośrednictwem poleceń Command

Przykład zrealizowania obsługi zdarzeń za pośrednictwem tradycyjnych Command

Akcja wywołana na interfejsie aplikacji powoduje uruchomienie metody Execute w Command.

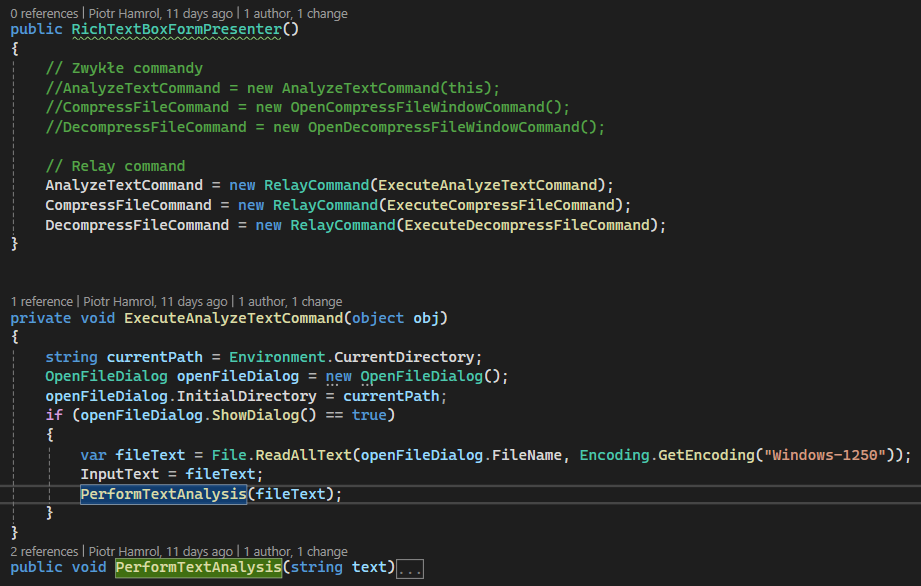
Metoda ta wykonuje dodatkowe kroki, na przykład otwarcie okna OpenFileDialog. Docelowo zadaniem Commanda jest wywołanie odpowiedniej metody we ViewModelu, czyli w klasie Presenter.

Możliwe jest zastosowanie Commandów w postaci Relay Command.

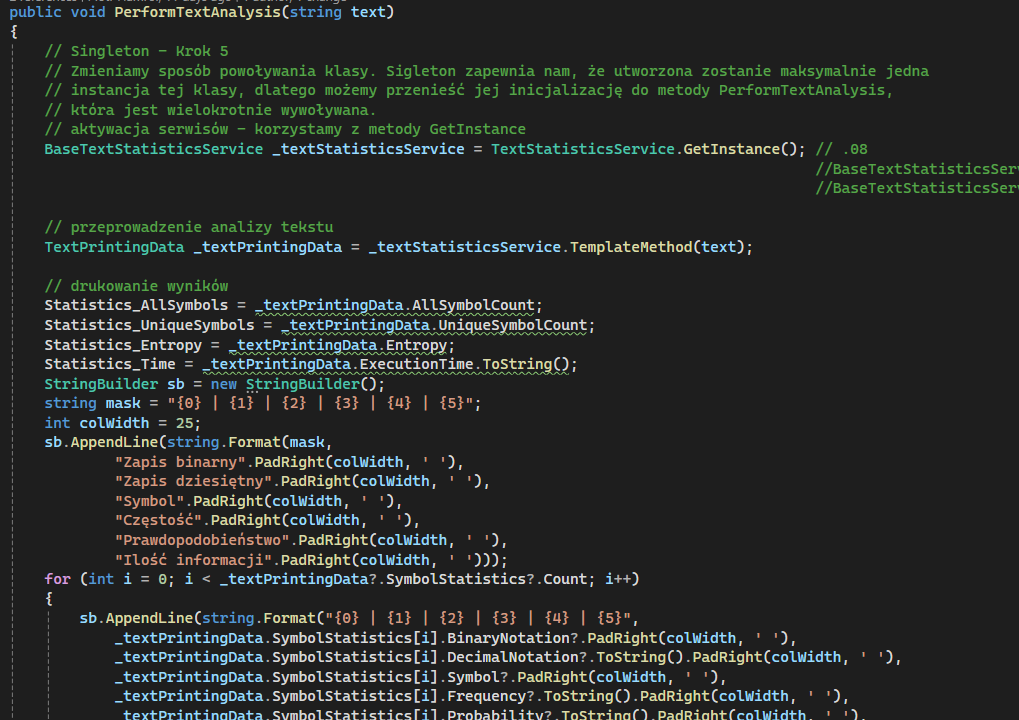
Znacząco zmniejsza to ilość dodatkowych plików.

Nadal zadaniem Commanda jest wywołanie metody w klasie Presenter.

Podobnie działo się w obsłudze zdarzeń OnClick w aplikacji WinForms.



Presenter w projekcie WPF



Zadaniem klasy Presenter było pośredniczenie pomiędzy interfejsem aplikacji a logiką biznesową.

Klasa ta znajduje się w projekcie z fortendem, posiada dostęp do kontrolek na ekranie aplikacji (w tym przypadku za pośrednictwem interfejsu INotifyPropertyChanged), jak również jest odpowiedzialna za zainicjalizowanie poszczególnych serwisów, ma zatem dostęp do logiki biznesowej.

Zasada działania klasy jest identyczna jak w przypadku projektu WinForms.

Dependency Injection

Aplikacja działa. Jak widać przejście z WinForms na technologię WPF wymagało minimum pracy.

Jedyne działania programistyczne wykonane w tym celu polegały na napisaniu nowego frontendu.

Podobnie sprawy wyglądałyby, gdybyśmy przerabiali aplikację z desktopowej na WWW, co więcej, nic nie stoi na przeszkodzie by równocześnie utrzymywać aplikację w formie Desktopowej oraz WWW, co zmniejsza nam ilość potrzebnych prac (wspólny backend).

Łatwość podmiany frontendu nie jest jedyną zaletą takiego podziału kodu.

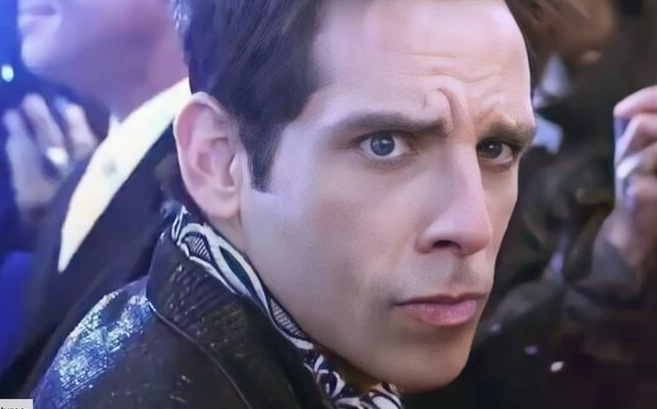
Jak zostało wspomniane jedyne zmiany w kodzie jakie nastąpiły, to dodanie nowego projektu z nowym forntendem. Jako, że nie były wymagane żadne zmiany w backendzie mamy pewność, że wszystko działa tak samo. Nie ma teoretycznie potrzeby testowania backendu. Jedyne co musimy sprawdzić to poprawność działania frontendu.

Z projektem jest jednak pewien problem.

Klasa Presenter mimo, iż jej zadanie jest dość jasne i naturalne to klasa ta wydaje się dziwna.

Jej istnienie powoduje, że projekt wygląda mało profesjonalnie.

Nie jest to jednak jedyny problem. Klasa ta łamie… **zasadę odwrócenia zależności**!



S.O.L.I.D.

**(S)**INGLE RESPONSIBILITY PRINCIPLE

ZASADA POJEDYNCZEJ ODPOWIEDZIALNOŚCI

**(O)**PEN CLOSE PRINCIPLE

ZASADA OTWARTE-ZAMKNIĘTE

**(L)**ISKOV SUBSTITUTION PRINCIPLE

ZASADA PODSTAWIENIA LISKOV

**(I)**NTERFACE SEGREGATION PRINCIPLE

ZASADA SEGREGACJI INTERFEJSÓW

**(D)**EPENDENCY INVERSION PRINCIPLE

ZASADA ODWRÓCENIA ZALEŻNOŚCI

S – SRP (SINGLE RESPONSIBILITY PRINCIPLE – ZASADA POJEDYNCZEJ ODPOWIEDZIALNOŚCI)

* klasa powinna mieć tylko jeden powód do zmiany, co oznacza, że powinna być odpowiedzialna tylko za jedną funkcjonalność lub aspekt działania systemu
* każda klasa w programie powinna realizować tylko jeden zestaw spójnych zadań, koncentrując się na jednej konkretnej odpowiedzialności
* przestrzeganie SRP ułatwia lokalizowanie i izolowanie błędów
* zastosowanie SRP przyczynia się do modułowości kodu
* sprzyja lepszemu ponownemu wykorzystaniu komponentów

O – OCP (OPEN CLOSE PRINCIPLE – ZASADA OTWARTE-ZAMKNIĘTE)

* oprogramowanie powinno być otwarte na rozszerzenia, ale zamknięte na modyfikacje
* powinno być możliwe dodawanie nowych funkcjonalności do systemu bez konieczności modyfikowania istniejącego kodu
* zachęca do stosowania abstrakcji i polimorfizmu
* polega na definiowaniu ogólnych interfejsów lub klas bazowych, które określają wymagane zachowania, deweloperzy mogą następnie rozszerzać te abstrakcje za pomocą nowych klas, które implementują te zachowania w specyficzny sposób
* przestrzeganie tej zasady może znacząco zmniejszyć ryzyko wprowadzania błędów podczas rozbudowy systemu
* nowe funkcjonalności są dodawane jako nowe klasy, nie zakłócając działania istniejących komponentów

L – LSP (LISKOV SUBSTITUTION PRINCIPLE – ZASADA PODSTAWIENIA LISKOV)

* odgrywa kluczową rolę w utrzymaniu spójności i niezawodności w hierarchiach dziedziczenia oprogramowania
* obiekty w programie powinny być zastępowalne przez instancje ich podtypów bez wpływu na poprawność działania programu
* dzięki przestrzeganiu LSP systemy stają się bardziej modułowe, ponieważ komponenty mogą być łatwiej zastąpione lub rozszerzone przez nowe implementacje bez ryzyka nieoczekiwanych efektów ubocznych
* deweloperzy muszą zapewnić, że każda klasa dziedzicząca rozszerza funkcjonalność klasy bazowej, nie zmieniając jej pierwotnego zachowania
* zakłada unikanie sytuacji, w których metody dziedziczone są nadpisywane w sposób, który zmienia ich działanie, lub wprowadzanie nowych wymagań dla danych wejściowych

I – ISP (INTERFACE SEGREGATION PRINCIPLE – ZASADA SEGREGACJI INTERFEJSÓW)

* projektowanie czystych interfejsów, które promują spójność i minimalizm
* zamiast jednego dużego interfejsu oferującego wiele różnych funkcji, powinno się tworzyć wiele mniejszych, bardziej specyficznych interfejsów
* interfejsy projektowanie z myślą o modularności i elastyczności
* prowadzi to do efektu, w którym klasy nie są obarczane nadmiarowymi metodami, które nie są dla nich istotne

D – DIP (DEPENDENCY INVERSION PRINCIPLE – ZASADA ODWRÓCENIA ZALEŻNOŚCI)

* odgrywa kluczową rolę w tworzeniu zdecentralizowanych systemów
* moduły wysokiego poziomu nie powinny zależeć od modułów niskiego poziomu
* oba rodzaje modułów powinny zależeć od abstrakcji
* projektowanie systemów powinno odbywać się od ogółu do szczegółu, promując zależności od interfejsów lub klas abstrakcyjnych, a nie od konkretnych implementacji
* sprzyja osiągnięciu luźnego sprzężenia między komponentami systemu, co zwiększa modułowość i elastyczność kodu
* umożliwia stosowanie mocków lub stubów w testach jednostkowych

Zasady SOLID nie są zestawem gotowych rozwiązań. Stanowią drogowskaz i wyznaczają kierunek w rozwoju oprogramowania.

Każda z tych zasad ma na celu promowanie określonych dobrych praktyk, które prowadzą do tworzenia bardziej modułowego, łatwiejszego do zarządzania i rozszerzania kodu.

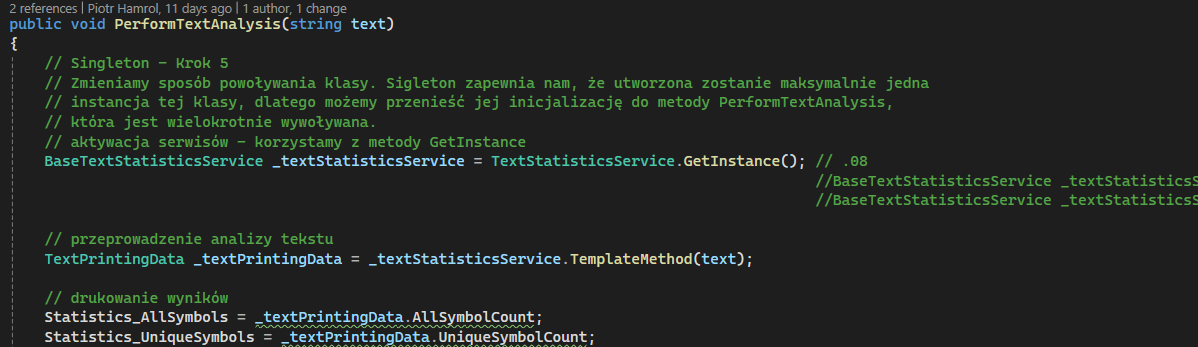
Programowanie zgodne z zasadami SOLID jest uznawane za jedno z najlepszych podejść do projektowania oprogramowania, zwłaszcza w kontekście programowania obiektowego.

Rygorystyczne stosowanie każdej z zasad w każdym aspekcie projektu może być wyzwaniem, zazwyczaj warto dążyć do ich przestrzegania, aby maksymalizować korzyści płynące z czystego i dobrze zaprojektowanego kodu.

Odwrócenie zależności

|  |  |
| --- | --- |
| Klasa MyLogicClass przyjmuje parametr typu MyServiceClass.  Można więc powiedzieć, że klasa MyLogicClass zmusza każdego, kto będzie chciał z niej skorzystać do użycia konkretnej implementacji MyServiceClass.  Kierunek wiązania jest ze środka na zewnątrz. Jest to ścisłe powiązanie. | Klasa MyLogicClass przyjmuje parametr typu IServiceClass.  Można powiedzieć, że klasa MyLogicClass daje wolną rękę każdemu, kto będzie chciał z niej skorzystać do użycia dowolnej implementacji, tak długo jak spełni interfejs IServiceClass.  Kierunek wiązania jest z zewnątrz do środka. Klasa MyLogicClass będzie współpracowała z tym, co jej się poda. |
|  |  |

Gdzie łamana jest zasada DIP?



Klasa Presenter jawnie korzysta z klasy TextStatisticsService. Instancja klasy podstawiana jest co prawda do zmiennej typu BaseTextStatisticsService, ale metoda GetInstance pochodzi z konkretnie wskazanej implementacji TextStatisticsService.

Dependency Injection

Jednym ze sposobów zapewnienia zgodności z piątą zasadą SOLID jest stosowanie mechanizmów wstrzykiwania zależności.

Mechanizmy te stosują najczęściej jeden z trzech sposobów wstrzykiwania zależności:

* wstrzyknięcie poprzez konstruktor
* wstrzyknięcie poprzez propercję
* wstrzyknięcie poprzez metodę

Najczęściej wykorzystuje się do tego gotowe biblioteki, takie jak:

* Microsoft Dependency Injection
* Autofac
* NInject
* Unity
* Lamar

Jak i po co?

Kontenery IoC zwalniają programistę z obowiązku zarządzania procesem powoływania instancji obiektów.

Kontener pozwala na zarejestrowanie obiektów różnego typu, przypisania ich do abstrakcji a następnie umożliwia dostęp do nich i stosowanie w całym cyklu życia aplikacji i w różnych jej obszarach.

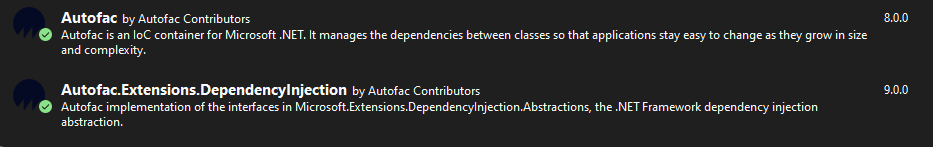
Większość kontenerów IoC przy okazji rejestrowania obiektów pozwala na skonfigurowanie zasad tworzenia i przechowywania instancji obiektów.

Większość frameworków pozwala na skonfigurowanie trzech podstawowych trybów:

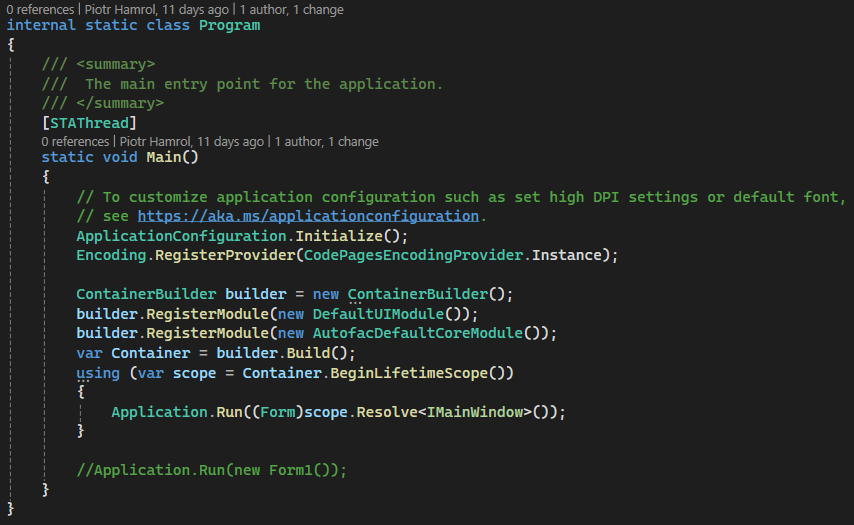
* Scoped - każdy request HTTP dostaje nową instancję obiektu, ale w trakcie przetwarzania tego samego requestu zwracana jest ta sama instancja obiektu. Pozwala zachować stan w ramach jednego requesta.
* Transient - nowa instancja obiektu tworzona za każdym razem, gdy jest żądana
* Singleton - klasyczna implementacja wzorca Singleton

WinForms – Autofac

WinForms domyślnie nie posiada frameworków IoC. Aby zaimplementować Dependency Injection w WinForms należy pobrać, któryś z dostępnych frameworków z biblioteki Nuget. Bardzo popularną biblioteką jest Autofac



Inicjalizacja mechanizmu odbywa się w głównym pliku programu, czyli w Program.cs



* Stworzenie kontenera
* Rejestracja modułów
* Zbudowanie kontenera
* Uruchomienie aplikacji w ramach jednego Scope
  + Już w tym miejscu następuje wstrzyknięcie zarejestrowanego głównego okna aplikacji

Rejestracja modułów mogłaby odbyć się w pliku Program.cs, jednak mając aplikację podzieloną na różne projekty/warstwy, również dobrze jest podzielić rejestracje modułów z poszczególnych warstw na osobne pliki.

Dlatego w powyższym kodzie widać dwie linie builder.RegisterModule.

|  |  |
| --- | --- |
|  |  |

Obiekty można rejestrować bezpośrednio lub przypisując je do abstrakcji.

Jeżeli chcemy przypisać klasę do abstrakcji należy skorzystać z operacji „As<T>()”

W tym miejscu następuje też konfiguracja zasad inicjalizacji obiektów. Na powyższym przykładzie skorzystano z implementacji klasycznego wzorca Singleton – „SingleInstance()”

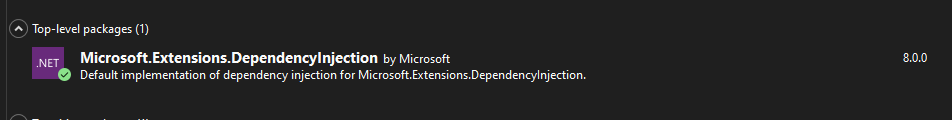
Jak widać stosowanie tego mechanizmu pozwala całościowo zarządzać w jednym miejscu zasadami inicjalizacji obiektów oraz decydowania, która konkretnie implementacja obiektu będzie rozwiązywana w procesie inicjalizacji.

Uproszczenie kodu

|  |  |
| --- | --- |
|  |  |
| Struktura projektu frontendowego znacząco uprościła się.  Zniknął folder Presenters.  W konstruktorze umieszczono obiekty, potrzebne do wykonania operacji. Nastąpiła ich automatyczna inicjalizacja. |  |

WPF – Microsoft Dependency Injection

Podobnie jak Autofac, rozwiązanie Microsoftu wymaga zainstalowania dodatkowej biblioteki z Nugeta.



Inicjalizacja mechanizmu odbywa się w głównym pliku programu, czyli w App.xaml.cs

|  |  |
| --- | --- |
|  | Inicjalizacja odbywa się w konstruktorze.  Podobnie jak w przypadku Autofac:   * Stworzenie kontenera (ServiceCollection) * Rejestracja modułów * Zbudowanie kontenera   Jako że główne okno aplikacji przestało być jawnie wskazane, a jest rozwiązywane w ramach DI:   * W procedurze OnStartup * Pozyskanie głównego okna, wyświetlenie   Stosowanie DI w WPF powoduje pewne problemy z wyłączeniem aplikacji. Stąd potrzeba oprogramowania OnMainWindowClosing. |

Rejestracja modułów mogłaby odbyć się w pliku App.xaml.cs, jednak mając aplikację podzieloną na różne projekty/warstwy, również dobrze jest podzielić rejestracje modułów z poszczególnych warstw na osobne pliki.

Dlatego w powyższym kodzie widać dwie linie […].Register(services);.

|  |  |
| --- | --- |
|  |  |

Obsługa mechanizmu jest niemal identyczna jak w przypadku Autofac. Różni się jedynie składnią.

Obiekty można rejestrować bezpośrednio lub przypisując je do abstrakcji.

Jeżeli chcemy przypisać klasę do abstrakcji należy skorzystać funkcji generycznych, np. „AddSingleton<T>()”

W tym miejscu następuje też konfiguracja zasad inicjalizacji obiektów. Na powyższym przykładzie skorzystano z implementacji klasycznego wzorca Singleton – „SingleInstance()”

Dla okien aplikacji skorzystano z tryby Transient.

Jak widać stosowanie tego mechanizmu pozwala całościowo zarządzać w jednym miejscu zasadami inicjalizacji obiektów oraz decydowania, która konkretnie implementacja obiektu będzie rozwiązywana w procesie inicjalizacji.

Uproszczenie kodu

|  |  |
| --- | --- |
|  |  |
| W przypadku WPF aplikacja nie zmniejszyła aż tak bardzo ilości plików.  Pliki, które zostały są ściśle powiązane z WPF (Commandsy, ViewModele) |  |

ViewModele nadal zostały ale praca w nich bardzo się uprościła.

Podobnie jak w przypadku Autofac w konstruktorze klasy wypisujemy parametry/obiekty, z których mamy zamiar korzystać.

Następuje automatyczna inicjalizacja obiektów zgodnie ze skonfigurowanymi zasadami.

W przypadku WPF stosowanie DI może utrudnić korzystanie z designerów okien. Wynika to z faktu, że „normlanie” podpinany ViewModel wskazywany jest jawnie poprzez silne wiązanie. Designer korzysta z tego faktu i na przykład podpowiada składnię, podkreśla błędy, itp.

Model inicjalizowany poprzez mechanizm DI jest luźno powiązany, przez co Designer nie może w locie sprawdzać zależności. Tracimy dostęp do Intelisense, większość kodu podkreślona jest jako niepewna.

Rozwiązaniem tego problemu jest zmiana sposobu powiązywania ViewModeli:



Ewolucja

Zaobserwować można ewolucje i stopniowe przystosowywanie projektów do korzystania z mechanizmów DI.

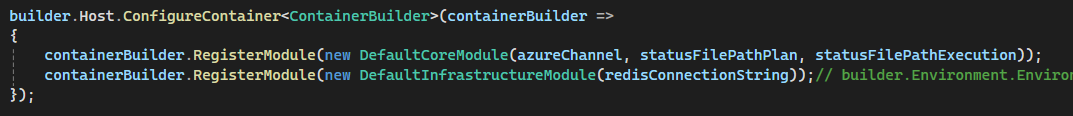
WinForms domyślnie nie posiada w ogóle wsparcia.

WPF dużo lepiej współpracuje z DI.

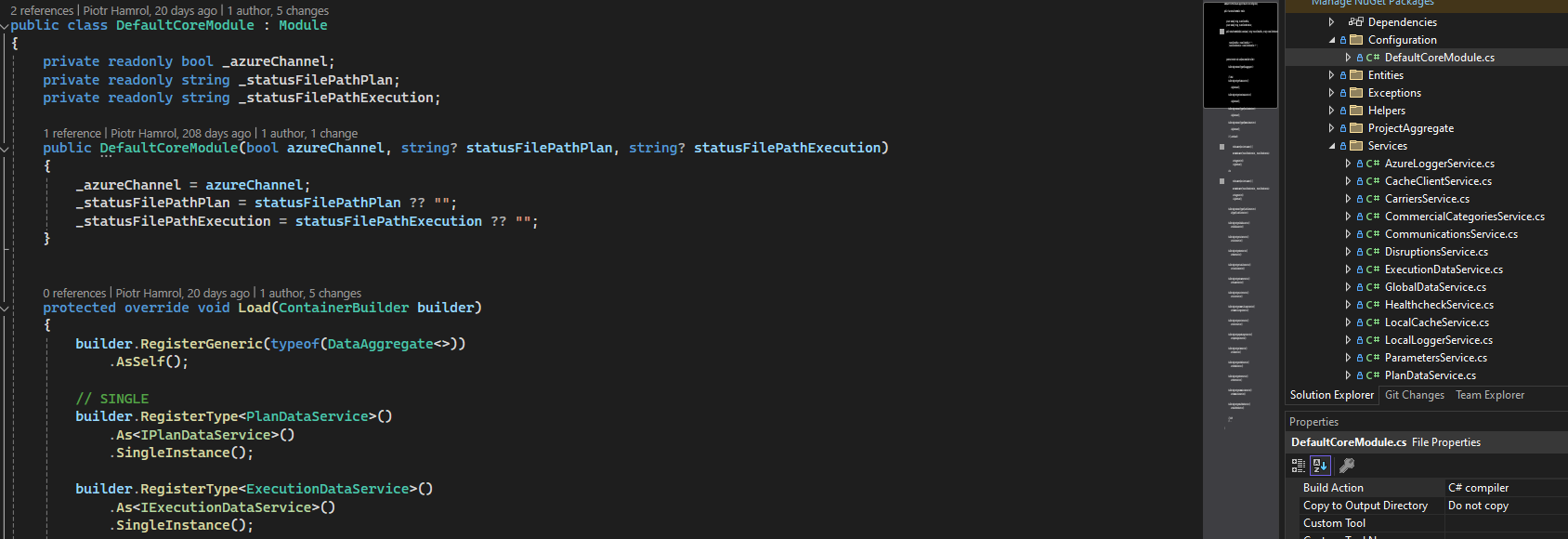
Projekty webowe (ASP.NET) posiadają bardzo dobre wsparcie dla mechanizmów DI.

Po doinstalowaniu odpowiedniego Nugeta, w pliku Program.cs można praktycznie bez problemu rozpocząć konfigurowanie i korzystanie z DI:

* rejestracja modułów:



* konfiguracja



* inicjalizacja i wstrzykiwanie obiektów poprzez konstruktor

