


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.

Message:

Dodanie fontendu WPF, dodanie DependencyInjection

 Piotr Hamrol

01.03.2024

▲ Changes (77)


...

▲ Semestr 1\Projekt


▲ Etap_09\Rozwiazanie\Projekt_01_Etap_09_Rozwi...

▸ Commands


▸ Presenters

 App.xaml


A

 App.xaml.cs


A

 AssemblyInfo.cs


A

 CompressForm.xaml


A

 CompressForm.xaml.cs


A

 DecompressForm.xaml


A

 DecompressForm.xaml.cs


A

 MainWindow.xaml

A


 MainWindow.xaml.cs

A

 Projekt_01_Etap_09_Rozwiazanie.WPF.csproj

A

▸ Etap_10_DependencyInjection\Rozwiazanie

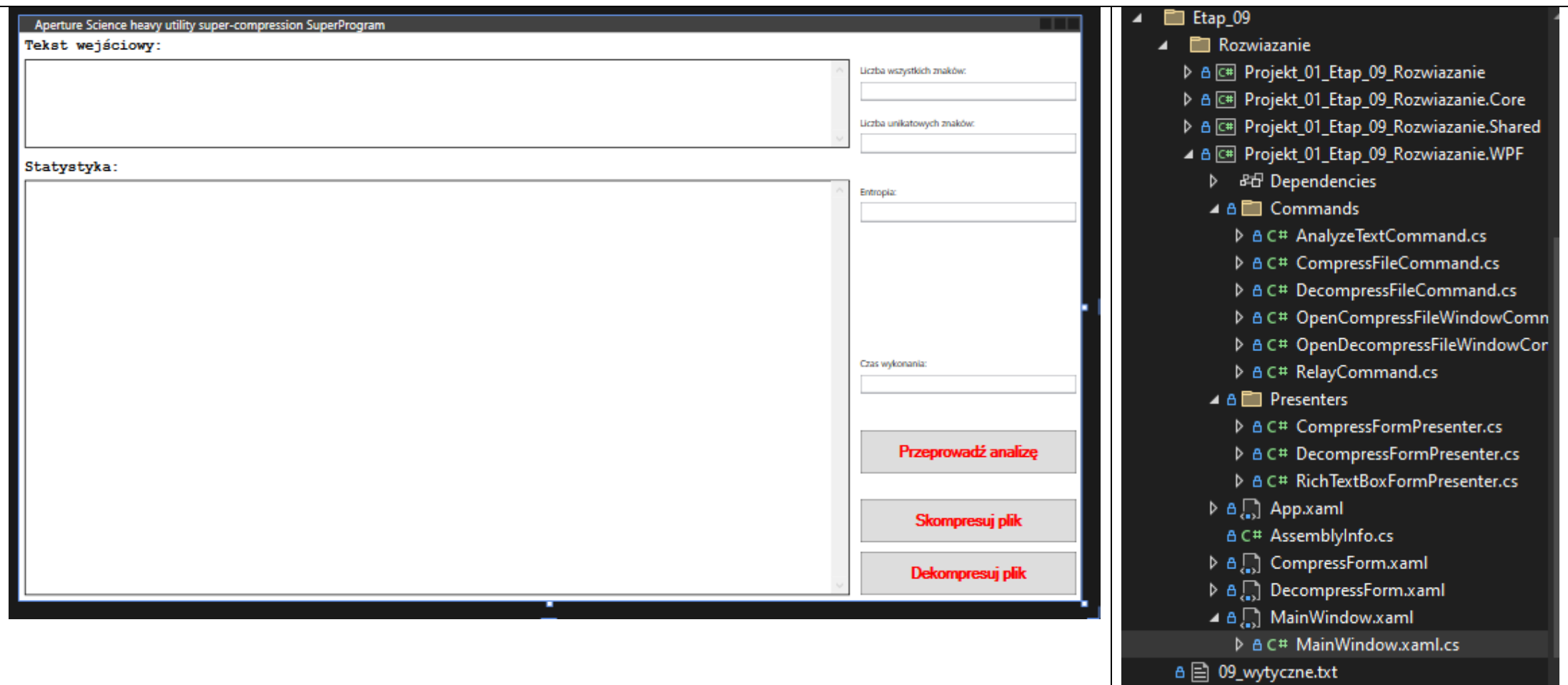
 zsk_pad_lekcje.sln

M

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

```
<Border Grid.Row="1" Grid.Column="0" Grid.RowSpan="4" Style="{StaticResource DefaultBorder}" Height="86">
    <ScrollView>
        <TextBlock Text="{Binding InputText}" Style="{StaticResource FieldTextBlock}"></TextBlock>
    </ScrollView>
</Border>

<Label Grid.Row="5" Grid.Column="0" Style="{StaticResource DefaultLabel}">Statystyka:</Label>
<Border Grid.Row="6" Grid.Column="0" Grid.RowSpan="8" Style="{StaticResource DefaultBorder}">
    <ScrollView>
        <TextBlock Text="{Binding Statistics}" Style="{StaticResource StatisticsTextBlock}"></TextBlock>
    </ScrollView>
</Border>

<!-- Prawa kolumna -->
<Label Grid.Row="1" Grid.Column="1" Style="{StaticResource FieldLabel}">Liczba wszystkich znaków:</Label>
<TextBox Grid.Row="2" Grid.Column="1" Text="{Binding Statistics_AllSymbols}" Style="{StaticResource FieldTextBoxWithSpacer}"></TextBox>

<Label Grid.Row="3" Grid.Column="1" Style="{StaticResource FieldLabel}">Liczba unikatowych znaków:</Label>
<TextBox Grid.Row="4" Grid.Column="1" Text="{Binding Statistics_UniqueSymbols}" Style="{StaticResource FieldTextBox}"></TextBox>

<Label Grid.Row="6" Grid.Column="1" Style="{StaticResource FieldLabel}">Entropia:</Label>
<TextBox Grid.Row="7" Grid.Column="1" Text="{Binding Statistics_Entropy}" Style="{StaticResource FieldTextBoxWithBigSpacer}"></TextBox>

<Label Grid.Row="9" Grid.Column="1" Style="{StaticResource FieldLabel}">Czas wykonania:</Label>
<TextBox Grid.Row="10" Grid.Column="1" Text="{Binding Statistics_Time}" Style="{StaticResource FieldTextBoxWithBigSpacer}"></TextBox>

<Button Grid.Row="11" Grid.Column="1" Command="{Binding AnalyzeTextCommand}" Style="{StaticResource ButtonWithSpacer}">Przeprowadź analizę</Button>
<Button Grid.Row="12" Grid.Column="1" Command="{Binding CompressFileCommand}" Style="{StaticResource DefaultButton}">Skompresuj plik</Button>
<Button Grid.Row="13" Grid.Column="1" Command="{Binding DecompressFileCommand}" Style="{StaticResource DefaultButton}">Dekompresuj plik</Button>
</Grid>
```

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

1 reference | Piotr Hamrol, 11 days ago | 1 author, 1 change

```
public class AnalyzeTextCommand : ICommand
```

```
{
```

```
    private RichTextBoxFormPresenter _model;
```

0 references | Piotr Hamrol, 11 days ago | 1 author, 1 change

```
    public AnalyzeTextCommand(RichTextBoxFormPresenter model)
```

```
    {
```

```
        _model = model;
```

```
    }
```

```
    private bool _isBusy = false;
```

0 references | Piotr Hamrol, 11 days ago | 1 author, 1 change

```
    public bool CanExecute(object parameter)
```

```
    {
```

```
        return !_isBusy;
```

```
    }
```

0 references | Piotr Hamrol, 11 days ago | 1 author, 1 change

```
    public void Execute(object parameter)
```

```
    {
```

```
        _isBusy = true;
```

```
        CanExecuteChanged?.Invoke(this, new EventArgs());
```

```
        string currentPath = Environment.CurrentDirectory;
```

```
        OpenFileDialog openFileDialog = new OpenFileDialog();
```

```
        openFileDialog.InitialDirectory = currentPath;
```

```
        if (openFileDialog.ShowDialog() == true)
```

```
        {
```

```
            var fileText = File.ReadAllText(openFileDialog.FileName, Encoding.GetEncoding("Windows-1250"));
```

```
            _model.InputText = fileText;
```

```
            _model.PerformTextAnalysis(fileText);
```

```
        }
```

```
        _isBusy = false;
```

```
        CanExecuteChanged?.Invoke(this, new EventArgs());
```

```
    }
```

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 działa się w obsłudze zdarzeń OnClick w aplikacji WinForms.

0 references | Piotr Hamrol, 11 days ago | 1 author, 1 change

```
public RichTextBoxFormPresenter()
{
    // Zwykłe commandy
    //AnalyzeTextCommand = new AnalyzeTextCommand(this);
    //CompressFileCommand = new OpenCompressFileWindowCommand();
    //DecompressFileCommand = new OpenDecompressFileWindowCommand();

    // Relay command
    AnalyzeTextCommand = new RelayCommand(ExecuteAnalyzeTextCommand);
    CompressFileCommand = new RelayCommand(ExecuteCompressFileCommand);
    DecompressFileCommand = new RelayCommand(ExecuteDecompressFileCommand);
}
```

1 reference | Piotr Hamrol, 11 days ago | 1 author, 1 change

```
private void ExecuteAnalyzeTextCommand(object obj)
{
    string currentPath = Environment.CurrentDirectory;
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.InitialDirectory = currentPath;
    if (openFileDialog.ShowDialog() == true)
    {
        var fileText = File.ReadAllText(openFileDialog.FileName, Encoding.GetEncoding("Windows-1250"));
        InputText = fileText;
        PerformTextAnalysis(fileText);
    }
}
```

2 references | Piotr Hamrol, 11 days ago | 1 author, 1 change

```
public void PerformTextAnalysis(string text)...
```


Prezenter w projekcie WPF

```
public void PerformTextAnalysis(string text)
{
    // Singleton - Krok 5
    // Zmieniamy sposób powoływania klasy. Singleton zapewnia nam, że utworzona zostanie maksymalnie jedna
    // instancja tej klasy, dlatego możemy przenieść jej inicjalizację do metody PerformTextAnalysis,
    // która jest wielokrotnie wywoływana.
    // aktywacja serwisów - korzystamy z metody GetInstance
    BaseTextStatisticsService _textStatisticsService = TextStatisticsService.GetInstance(); // .08
                                                    //BaseTextStatisticsSer
                                                    //BaseTextStatisticsSer

    // przeprowadzenie analizy tekstu
    TextPrintingData _textPrintingData = _textStatisticsService.TemplateMethod(text);

    // drukowanie wyników
    Statistics_AllSymbols = _textPrintingData.AllSymbolCount;
    Statistics_UniqueSymbols = _textPrintingData.UniqueSymbolCount;
    Statistics_Entropy = _textPrintingData.Entropy;
    Statistics_Time = _textPrintingData.ExecutionTime.ToString();
    StringBuilder sb = new StringBuilder();
    string mask = "{0} | {1} | {2} | {3} | {4} | {5}";
    int colWidth = 25;
    sb.AppendLine(string.Format(mask,
        "Zapis binarny".PadRight(colWidth, ' '),
        "Zapis dziesiętny".PadRight(colWidth, ' '),
        "Symbol".PadRight(colWidth, ' '),
        "Częstość".PadRight(colWidth, ' '),
        "Prawdopodobieństwo".PadRight(colWidth, ' '),
        "Ilość informacji".PadRight(colWidth, ' ')));
    for (int i = 0; i < _textPrintingData?.SymbolStatistics?.Count; i++)
    {
        sb.AppendLine(string.Format("{0} | {1} | {2} | {3} | {4} | {5}",
            _textPrintingData.SymbolStatistics[i].BinaryNotation?.PadRight(colWidth, ' '),
            _textPrintingData.SymbolStatistics[i].DecimalNotation?.ToString().PadRight(colWidth, ' '),
            _textPrintingData.SymbolStatistics[i].Symbol?.PadRight(colWidth, ' '),
            _textPrintingData.SymbolStatistics[i].Frequency?.ToString().PadRight(colWidth, ' '),
            _textPrintingData.SymbolStatistics[i].Probability?.ToString().PadRight(colWidth, ' '),
            _textPrintingData.SymbolStatistics[i].Symbol?.ToString().PadRight(colWidth, ' ')));
    }
}
```

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 frontendem. 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.

```
0 references | 0 changes | 0 authors, 0 changes
public MyLogicClass(MyServiceClass service)
{
    //...
}
```

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.

```
0 references | 0 changes | 0 authors, 0 changes
public MyLogicClass(IServiceClass service)
{
    //...
}
```


Gdzie łamana jest zasada DIP?

2 references | Piotr Hamrol, 11 days ago | 1 author, 1 change

```
public void PerformTextAnalysis(string text)
```

```
{
```

```
    // Singleton - Krok 5
```

```
    // Zmieniamy sposób powoływania klasy. Singleton zapewnia nam, że utworzona zostanie maksymalnie jedna
```

```
    // instancja tej klasy, dlatego możemy przenieść jej inicjalizację do metody PerformTextAnalysis,
```

```
    // która jest wielokrotnie wywoływana.
```

```
    // aktywacja serwisów - korzystamy z metody GetInstance
```

```
    BaseTextStatisticsService _textStatisticsService = TextStatisticsService.GetInstance(); // .08
```

```
    //BaseTextStatisticsService _textStatisticsS
```

```
    //BaseTextStatisticsService _textStatisticsS
```

```
    // przeprowadzenie analizy tekstu
```

```
    TextPrintingData _textPrintingData = _textStatisticsService.TemplateMethod(text);
```

```
    // drukowanie wyników
```

```
    Statistics_AllSymbols = _textPrintingData.AllSymbolCount;
```

```
    Statistics_UniqueSymbols = _textPrintingData.UniqueSymbolCount;
```

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 właściwość
- 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



Autofac by Autofac Contributors

8.0.0

Autofac is an IoC container for Microsoft .NET. It manages the dependencies between classes so that applications stay easy to change as they grow in size and complexity.



Autofac.Extensions.DependencyInjection by Autofac Contributors

9.0.0

Autofac implementation of the interfaces in Microsoft.Extensions.DependencyInjection.Abstractions, the .NET Framework dependency injection abstraction.

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

```
0 references | Piotr Hamrol, 11 days ago | 1 author, 1 change
internal static class Program
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    0 references | Piotr Hamrol, 11 days ago | 1 author, 1 change
    static void Main()
    {
        // To customize application configuration such as set high DPI settings or default font,
        // see https://aka.ms/applicationconfiguration.
        ApplicationConfiguration.Initialize();
        Encoding.RegisterProvider(CodePagesEncodingProvider.Instance);

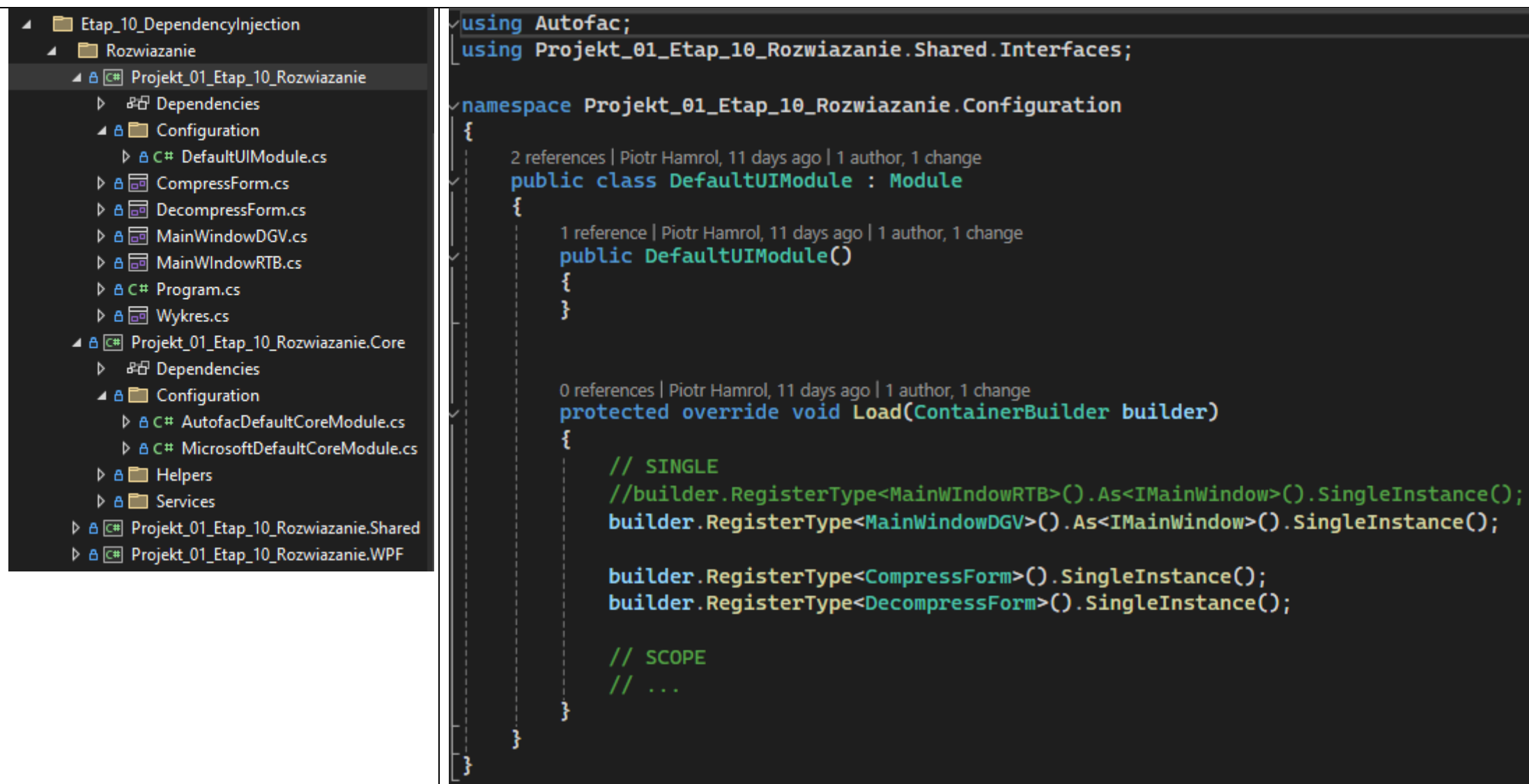
        ContainerBuilder builder = new ContainerBuilder();
        builder.RegisterModule(new DefaultUIModule());
        builder.RegisterModule(new AutofacDefaultCoreModule());
        var Container = builder.Build();
        using (var scope = Container.BeginLifetimeScope())
        {
            Application.Run((Form)scope.Resolve<IMainWindow>());
        }

        //Application.Run(new Form1());
    }
}
```

- 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`.



The image shows a Visual Studio interface with a project explorer on the left and a code editor on the right.

Project Explorer (Left):

- Etap_10_DependencyInjection
 - Rozwiazanie
 - Projekt_01_Etap_10_Rozwiazanie
 - Dependencies
 - Configuration
 - DefaultUIModule.cs
 - CompressForm.cs
 - DecompressForm.cs
 - MainWindowDGV.cs
 - MainWindowRTB.cs
 - Program.cs
 - Wykres.cs
 - Projekt_01_Etap_10_Rozwiazanie.Core
 - Dependencies
 - Configuration
 - AutofacDefaultCoreModule.cs
 - MicrosoftDefaultCoreModule.cs
 - Helpers
 - Services
 - Projekt_01_Etap_10_Rozwiazanie.Shared
 - Projekt_01_Etap_10_Rozwiazanie.WPF

Code Editor (Right):

```
using Autofac;
using Projekt_01_Etap_10_Rozwiazanie.Shared.Interfaces;

namespace Projekt_01_Etap_10_Rozwiazanie.Configuration
{
    2 references | Piotr Hamrol, 11 days ago | 1 author, 1 change
    public class DefaultUIModule : Module
    {
        1 reference | Piotr Hamrol, 11 days ago | 1 author, 1 change
        public DefaultUIModule()
        {
        }

        0 references | Piotr Hamrol, 11 days ago | 1 author, 1 change
        protected override void Load(ContainerBuilder builder)
        {
            // SINGLE
            //builder.RegisterType<MainWIndowRTB>().As<IMainWindow>().SingleInstance();
            builder.RegisterType<MainWIndowDGV>().As<IMainWindow>().SingleInstance();

            builder.RegisterType<CompressForm>().SingleInstance();
            builder.RegisterType<DecompressForm>().SingleInstance();

            // SCOPE
            // ...
        }
    }
}
```

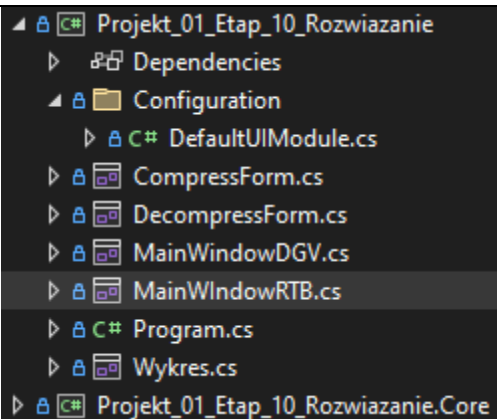

Obiekty można rejestrować bezpośrednio lub przypisując je do abstrakcji.

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 – „Singleton()”

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



```
public partial class MainWindowRTB : Form, IMainWindow
{
    private readonly ITextStatisticsService _textStatisticsService;
    private readonly CompressForm _compressForm;
    private readonly DecompressForm _decompressForm;

    0 references | Piotr Hamrol, 11 days ago | 1 author, 1 change
    public MainWindowRTB(ITextStatisticsService textStatisticsService, CompressForm compressForm, DecompressForm decompressForm)
    {
        _textStatisticsService = textStatisticsService;
        _compressForm = compressForm;
        _decompressForm = decompressForm;

        InitializeComponent();
    }
}
```

Struktura projektu
frontendowego
znacząco uproszczyła się.
Zniknął folder
Presenters.

W konstruktorze
umieszczono obiekty,
potrzebne do
wykonania operacji.
Nastąpiła ich
automatyczna
inicjalizacja.

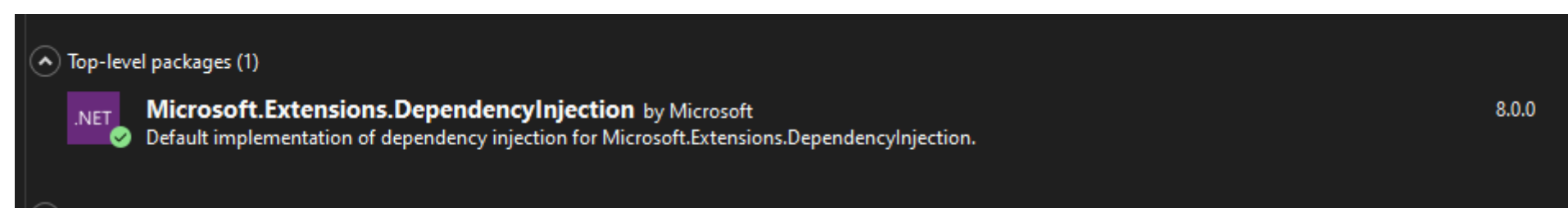
```
1 reference | Piotr Hamrol, 11 days ago | 1 author, 1 change
private void bt_PrzeprowadzAnalyze_Click(object sender, EventArgs e)
{
    string currentPath = Environment.CurrentDirectory;
    openFileDialog1.InitialDirectory = currentPath;
    //openFileDialog1.Filter = "Pliki tekstowe|*.txt";
    var result = openFileDialog1.ShowDialog(this);
    if (result == DialogResult.OK)
    {
        var fileText = File.ReadAllText(openFileDialog1.FileName, Encoding.GetEncoding("Windows-1250"));
        rtb_Statystyka_TekstWejscowy.Text = fileText;

        Stopwatch sw = new Stopwatch();
        sw.Start();
        TextStatisticsData statisticsData = _textStatisticsService.CountStatistics(fileText);
        sw.Stop();
        TextPrintingData printingData = _textStatisticsService.FillPrintingData(statisticsData);
        printingData.ExecutionTime = sw.Elapsed;

        // drukowanie wyników
        ShowStats(printingData);
    }
}
```

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

```
4 references | Piotr Hamrol, 11 days ago | 1 author, 1 change
public partial class App : Application
{
    private ServiceProvider serviceProvider;

    1 reference | Piotr Hamrol, 11 days ago | 1 author, 1 change
    public App()
    {
        ServiceCollection services = new ServiceCollection();
        new DefaultUIModule().Register(services);
        new MicrosoftDefaultCoreModule().Register(services);
        serviceProvider = services.BuildServiceProvider();
    }

    0 references | Piotr Hamrol, 11 days ago | 1 author, 1 change
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);

        Encoding.RegisterProvider(CodePagesEncodingProvider.Instance);

        var mainWindow = serviceProvider.GetService<IMainWindow>() as Window;
        if (mainWindow != null)
        {
            mainWindow.Closed += OnMainWindowClosing;
            mainWindow.Show();
        }
    }

    1 reference | Piotr Hamrol, 11 days ago | 1 author, 1 change
    private void OnMainWindowClosing(object? sender, EventArgs e)
    {
        Application.Current.Shutdown();
    }
}
```

Inicjalizacja odbywa się w konstruktorze.

Podobnie jak w przypadku Autofac:

- Stworzenie kontenera (ServiceCollection)
- Rejestracja modułów
- Zbudowanie kontenera

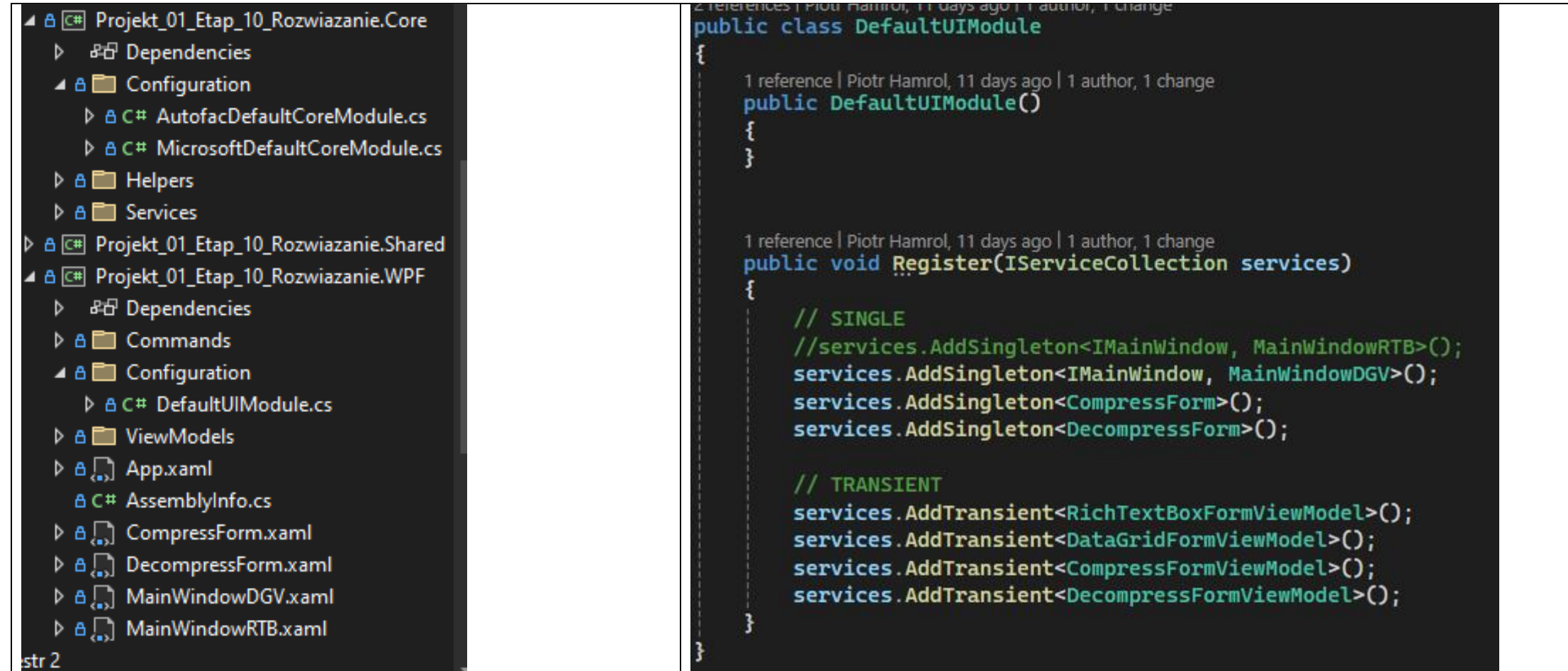
Jako że główne okno aplikacji przestało być jawnie wskazane, a jest rozwiązywane w ramach DI:

- 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);.



The image shows a Visual Studio interface with a Solution Explorer on the left and a Code Editor on the right. The Solution Explorer displays a project named 'Projekt_01_Etap_10_Rozwiazanie' with the following structure:

- Projekt_01_Etap_10_Rozwiazanie.Core
 - Dependencies
 - Configuration
 - AutofacDefaultCoreModule.cs
 - MicrosoftDefaultCoreModule.cs
 - Helpers
 - Services
- Projekt_01_Etap_10_Rozwiazanie.Shared
- Projekt_01_Etap_10_Rozwiazanie.WPF
 - Dependencies
 - Commands
 - Configuration
 - DefaultUIModule.cs
 - ViewModels
 - App.xaml
 - AssemblyInfo.cs
 - CompressForm.xaml
 - DecompressForm.xaml
 - MainWindowDGV.xaml
 - MainWindowRTB.xaml

The Code Editor shows the content of 'DefaultUIModule.cs' in the 'Projekt_01_Etap_10_Rozwiazanie.WPF' project. The code is as follows:

```
2 references | Piotr Hamrol, 11 days ago | 1 author, 1 change
public class DefaultUIModule
{
    1 reference | Piotr Hamrol, 11 days ago | 1 author, 1 change
    public DefaultUIModule()
    {
    }

    1 reference | Piotr Hamrol, 11 days ago | 1 author, 1 change
    public void Register(IServiceCollection services)
    {
        // SINGLE
        //services.AddSingleton<IMainWindow, MainWindowRTB>();
        services.AddSingleton<IMainWindow, MainWindowDGV>();
        services.AddSingleton<CompressForm>();
        services.AddSingleton<DecompressForm>();

        // TRANSIENT
        services.AddTransient<RichTextBoxFormViewModel>();
        services.AddTransient<DataGridFormViewModel>();
        services.AddTransient<CompressFormViewModel>();
        services.AddTransient<DecompressFormViewModel>();
    }
}
```

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

Projekt_01_Etap_10_Rozwiazanie.WPF

Dependencies

Commands

RelayCommand.cs

Configuration

DefaultUIModule.cs

ViewModels

CompressFormViewModel.cs

DataGridFormViewModel.cs

DecompressFormViewModel.cs

RichTextBoxFormViewModel.cs

App.xaml

AssemblyInfo.cs

CompressForm.xaml

DecompressForm.xaml

MainWindowDGV.xaml

MainWindowRTB.xaml

W przypadku WPF aplikacja nie zmniejszyła aż tak bardzo ilości plików.

Pliki, które zostały są ściśle powiązane z WPF (Commands, ViewModele)

```
public class RichTextBoxFormViewModel : INotifyPropertyChanged
{
    private readonly ITextStatisticsService _textStatisticsService;
    private readonly CompressForm _compressForm;
    private readonly DecompressForm _decompressForm;

    public event PropertyChangedEventHandler? PropertyChanged;
    6 references | Piotr Hamrol, 11 days ago | 1 author, 1 change
    protected void OnPropertyChanged(string name = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
    }

    Polecenia

    Pola tekstowe

    0 references | Piotr Hamrol, 11 days ago | 1 author, 1 change
    public RichTextBoxFormViewModel(ITextStatisticsService textStatisticsService, CompressForm compressForm, DecompressForm decompressForm)
    {
        _textStatisticsService = textStatisticsService;
        _compressForm = compressForm;
        _decompressForm = decompressForm;

        AnalyzeTextCommand = new RelayCommand(ExecuteAnalyzeTextCommand);
        CompressFileCommand = new RelayCommand(ExecuteCompressFileCommand);
        DecompressFileCommand = new RelayCommand(ExecuteDecompressFileCommand);
    }

    public void PerformTextAnalysis(string fileText)
    {
        Stopwatch sw = new Stopwatch();
        sw.Start();
        TextStatisticsData statisticsData = _textStatisticsService.CountStatistics(fileText);
        sw.Stop();
        TextPrintingData printingData = _textStatisticsService.FillPrintingData(statisticsData);
        printingData.ExecutionTime = sw.Elapsed;

        // drukowanie wyników
        Statistics_AllSymbols = printingData.AllSymbolCount;
        Statistics_UniqueSymbols = printingData.UniqueSymbolCount;
        Statistics_Entropy = printingData.Entropy;
        Statistics_Time = printingData.ExecutionTime.ToString();
        StringBuilder sb = new StringBuilder();
        string mask = "{0} | {1} | {2} | {3} | {4} | {5}";
        int colWidth = 25;
```

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 Intellisense, większość kodu podkreślona jest jako niepewna.

Rozwiązaniem tego problemu jest zmiana sposobu powiązywania ViewModeli:

```
d:DataContext="{d:DesignInstance viewModels:DataGridFormViewModel, IsDesignTimeCreatable=True}"  
Title="Aperture Science heavy utility super-compressor SuperDreadnought" Height="450" Width="1023">
```

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:

```
builder.Host.ConfigureContainer<ContainerBuilder>(containerBuilder =>
{
    containerBuilder.RegisterModule(new DefaultCoreModule(azureChannel, statusFilePathPlan, statusFilePathExecution));
    containerBuilder.RegisterModule(new DefaultInfrastructureModule(redisConnectionString)); // builder.Environment.EnvironmentName switch
});
```

- konfiguracja

The screenshot displays the Visual Studio IDE. The main editor shows the code for `DefaultCoreModule`. It includes private readonly fields for `_azureChannel`, `_statusFilePathPlan`, and `_statusFilePathExecution`. The constructor `DefaultCoreModule` takes three parameters and assigns them to these fields. The `Load` method, which is protected and overrides a base method, uses `builder.RegisterGeneric` to register `DataAggregate` as a self-service. It also registers `PlanDataService` and `ExecutionDataService` as singletons using `builder.RegisterType` and `.As<I...>().SingleInstance()`.

The Solution Explorer on the right shows the project structure. Under the `Configuration` folder, `DefaultCoreModule.cs` is listed. Below it, the `Services` folder contains several service classes: `AzureLoggerService.cs`, `CacheClientService.cs`, `CarriersService.cs`, `CommercialCategoriesService.cs`, `CommunicationsService.cs`, `DisruptionsService.cs`, `ExecutionDataService.cs`, `GlobalDataService.cs`, `HealthcheckService.cs`, `LocalCacheService.cs`, `LocalLoggerService.cs`, `ParametersService.cs`, and `PlanDataService.cs`.

The Properties window at the bottom right shows the file properties for `DefaultCoreModule.cs`. It indicates that the build action is `C# compiler`, the copy to output directory is `Do not copy`, and it is not a custom tool.

- inicjalizacja i wstrzykiwanie obiektów poprzez konstruktor

The screenshot shows the code for `PlanDataController` in the Visual Studio IDE. The class inherits from `BaseApiController`. It has three private readonly fields: `_configuration` of type `IConfiguration`, `_loggerService` of type `ILoggerService`, and `_planDataService` of type `IPlanDataService`. The constructor `PlanDataController` takes three parameters: `IConfiguration configuration`, `ILoggerService logowanieService`, and `IPlanDataService planDataService`. Inside the constructor, these parameters are assigned to the corresponding private fields.