1. Tworzymy aplikację WPF. Dodajemy do niej biblioteki z poprzednich zajęć Interfaces i Cars.DB_1.dll. Na poprzednich zajęciach pokazano jak wczytywać dynamicznie bibliotekę „schowaną" za interfejsem. Z braku czasu rezygnujemy częściowo z tego rozwiązania – dodajemy biblioteki, ale cały czas wszystkie obiekty „widzimy" przez interfejsy (czyli jeśli jest taka potrzeba nie używamy obiektu Car tylko interfejsu ICar).

2. W oknie głównym zmieniamy komponent dodany do głównego okna z Grid na DockPanel. Dodajemy do DockPanelu ListBox i nadajemy jej nazwę (Name). Następnie w pliku cs, w konstruktorze okna, po wywołaniu metody InitializeComponent() tworzymy obiekt Parking
   ```
   IParking parking = new Cars.DB_1.Parking();
   ```
   i do właściwości ItemsSource listy przypisujemy parking.GetAllCars();
   Dla listy, właściwość DisplayMemberPath ustawiamy na Name (tak nazywa się właściwość w ICar).

3. Tworzymy komponent do edycji/wyświetlania pojedynczego elementu.
   Do DockPanelu dodajemy Grid, dla którego tworzymy dwie kolumny i siedem wierszy (POD1). Do pierwszej kolumny wstawiamy etykiety (Label) z nazwą parametru, a do drugiego wstawiamy TextBox z wiązaniem danych do odpowiedniej właściwości (POD2).
   Właściwość DataContext komponentu Grid ustawiamy na:
   ```
   {Binding ElementName=lista,Path=SelectedItem}
   ```
   Na tym etapie w aplikacji można z listy wybrać samochód, po czym jego parametry wyświetlą się w interfejsie.

4. Tworzymy wrapper dla klasy Car (ICar) implementujący interfejs INotifyPropertyChanged. Nazwijmy tę klasę CarViewModel i umieśćmy w katalogu ViewModels.
   Implementujemy interfejs INotifyPropertyChanged:
   public event PropertyChangedEventHandler PropertyChanged;
   i dodajemy prywatne pole:
   ```
   private Interfaces.ICar car;
   ```
   Dodajemy konstruktor:
   ```
   public CarViewModel(Interfaces.ICar _car)
   ```
   i jego implementację (proszę się domyślić :))
   Żeby poprawnie reagować na zmiany właściwości i obsłużyć interfejs INotifyPropertyChanged, dodajemy implementację metody:
   ```
   private void RaisePropertyChanged(string propertyName)
   {
               if (PropertyChanged != null)
                   PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
   }
   ```
   Dla każdej z właściwości tworzymy nową właściwość na bazie poniższego schematu:

   ```
   public string Name
   {
               get => car.Name;
               set
               {
                   car.Name = value;
                   RaisePropertyChanged(nameof(Name));
               }
   ```

```
}
```

5. Zmieniamy wcześniejsze przypisanie do właściwości ItemsSource naszej listy jako: parking.GetAllCars() na:
   ```
   lista.ItemsSource = new ObservableCollection<CarViewModel>();
   ```
   tutaj wymagane będzie dodanie:
   ```
   using System.Collections.ObjectModel;
   ```
   Następnie do listy dodajemy nowe elementy CarViewModel utworzone na bazie obiektów pobranych z GetAllCars obiektu Parking.

6. Utworzenie szablonu wyświetlania danych dla listy – proszę pamiętać, że to koliduje z obiema właściwościami z Path w tytule, więc albo-albo.
   Szablon powinien wyglądać mniej więcej tak:
   ```
   <ListBox>
           <ListBox.ItemTemplate>
                   <DataTemplate>
                           <StackPanel>
                                   <TextBlock Text="{Binding Name}"/>
                           </StackPanel>
                   </DataTemplate>
           </ListBox.ItemTemplate>
   </ListBox>
   ```
   Oczywiście w StackPanelu można dodać inne elementy kontrolne typu Label, Id, Name, itd. I powiązać z innymi właściwościami – proszę zobaczyć jakie właściwości ma klasa Car.

7. Zmieniamy sposób wyświetlania rodzaju skrzyni biegów. Jest to typ wyliczeniowy, więc wykorzystamy do tego komponent typu ComboBox i metodę statyczną GetValues dla klasy System.Enum. W pierwszej kolejności musimy mieć odniesienie do biblioteki w której zdefiniowano ten typ wyliczeniowy. U nas jest to Interfaces (w projekcie zaliczeniowym powinno to być w Core!).

   Dodajemy w XAML-u dla obiektu Window
   ```
   xmlns:interfaces="clr-namespace:Cars.Interfaces;assembly=Interfaces"
   ```
   oraz dostęp do klasy Enum – jest ona w przestrzeni nazw System w klasie netsatandard.
   ```
   xmlns:system="clr-namespace:System;assembly=netstandard"
   ```
   Następnie tworzymy zasoby dla naszego okna (czyli zaraz za znacznikiem Window dodajemy:
   ```
   <Window.Resources>
    <ObjectDataProvider x:Key="TransmissionType"
                        ObjectType="{x:Type system:Enum}"
                        MethodName="GetValues">
        <ObjectDataProvider.MethodParameters>
            <x:Type Type="{x:Type interfaces:TransmissionType}"/>
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
   </Window.Resources>
   ```
   Proszę zauważyć, że tutaj tworzymy w XAML-u obiekt, który pobiera dane bezpośrednio z naszego typu wyliczeniowego. Proszę również zwrócić uwagę na sposób przekazania parametru.
   Zamiast TextBox-a wstawiamy ComboBox z dodanym następującym parametrem:

   ```
   ItemsSource="{Binding Source={StaticResource TransmissionType}}"
   ```

8. Tworzymy ogólną klasę dla komend - RelayCommand

```
internal class RelayCommand : ICommand
{
    public event EventHandler? CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }

    public bool CanExecute(object? parameter)
    {
        return canExecute == null ? true: canExecute(parameter);
    }

    public void Execute(object? parameter)
    {
        execute(parameter);
    }

    private readonly Action<object> execute;
    private readonly Predicate<object> canExecute;

    public RelayCommand(Action<object> execute, Predicate<object> canExecute)
    {
        this.execute = execute;
        this.canExecute = canExecute;
    }

    public RelayCommand(Action<object> _execute): this( _execute, null)
    {

    }
}
```

Proszę zobaczyć jak implementujemy taką ogólną klasę. Metodę, którą będziemy wywoływać przy wykonaniu komendy przechowujemy w delegacie Action<object>. Metoda CanExecute jest opcjonalna i będzie zwracała true jeśli wywołanie metody będzie możliwe. Jeśli nie została ta metoda ustawiona zawsze będziemy zwracać true. Pamiętajmy, że metodą CanExecute sterujemy „wykonywalnością" komendy, a co za tym idzie dostępnością komponentów kontrolnych związanych z komendą.

9. Tworzymy nową klasę CarListViewModel, która też implementuje interfejs INotifyPropertyChanged. Ponownie implementujemy ten interfejs (dodajemy zdarzenie i metodę RaisePropertyChanged). Do tej klasy przenosimy utworzenie ObservableCollection z listą samochodów. Następnie tworzymy właściwość (publiczną) Cars i udostępniamy nasze ObservableCollection z samochodami. Następnie w głównym oknie (MainWindow.xaml.cs) przypisujemy obiekt CarListViewModel do DataContext okna.

10. W CarListViewModel tworzymy nową komendę AddNewCar. W pierwszej kolejności tworzymy metodę prywatną z dodaną nową funkcjonalnością tworzenia nowego samochodu i od razu dodania go do listy (funkcjonalność sprawdzenia poprawności danych zrobimy później). Następnie tworzymy pole typu RelayCommand i właściwość publiczną (tylko get), która ten obiekt udostępnia.
    W oknie głównym dodajemy nowy komponent (np. StackPanel) na dole naszego okna, dodajemy do niego nowy Button i ustawiamy jego właściwość Command na wiązanie do udostępnionej komendy.

11. Nowo utworzony samochód powinien automatycznie stać się tym, który jest edytowany w komponencie Grid. W tym celu tworzymy nową właściwość publiczną SelectedCar typu CarViewModel z „pełną” obsługą właściwości (czyli RaisePropertyChanged). Dalej, tworzymy dwukierunkowe łącze z SelectedItem naszej listy do SelectedCar. Tę samą właściwość wykorzystujemy jako DataContext w komponencie Grid w którym mamy edytor parametrów. W ostatnim kroku, po utworzeniu nowego samochodu przypisujemy go do właściwości SelectedCar.

12. Tworzymy funkcjonalność związaną z filtrowaniem listy. W CarListViewModel dodajemy pole i właściwość FilterValue oba typu string. Dodajemy również nową komendę (znowu z metodą, obiektem RelayCommand i właściwością). Dodajemy prywatne pole typu ListCollectionView o nazwie view. W konstruktorze dodajemy powiązanie:

    ```
    view = (ListCollectionView)CollectionViewSource.GetDefaultView(cars);
    ```
    gdzie cars jest naszą ObservableCollection samochodów.
    Załóżmy, że nasze filtrowanie ma dotyczyć nazwy samochodu. W metodzie dla komendy wpisujemy:
    ```
    if (!string.IsNullOrWhiteSpace(filterValue))
    {
        view.Filter = (c) => ((CarViewModel)c).Name.Contains(filterValue);

    }
    else
    {
        view.Filter = null;
    }
    ```
    Do interfejsu (może być obok Buttonu AddNewCar) dodajemy pole TextBox i nowy Button. Właściwość TextBox łączymy z udostępnioną właściwością FilterValue, a komendę przycisku przypisujemy komendę filtrowania.

Podpowiedzi:

POD1:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
</Grid.ColumnDefinitions>
```

POD2:

```
<Label Grid.Row="0" Grid.Column="0">Id</Label>
<TextBox Grid.Row="0" Grid.Column="1" Text="{Binding Id}"></TextBox>
```