JAK NIE DAĆ SIĘ ZWARIOWAĆ, CZYLI O ZDROWIU PSYCHICZNYM W IT | QUO VADIS, EMBEDDED?

INDEX: 285358

www.programistamag.pl

Magazyn programistów i liderów zespołów IT

programista

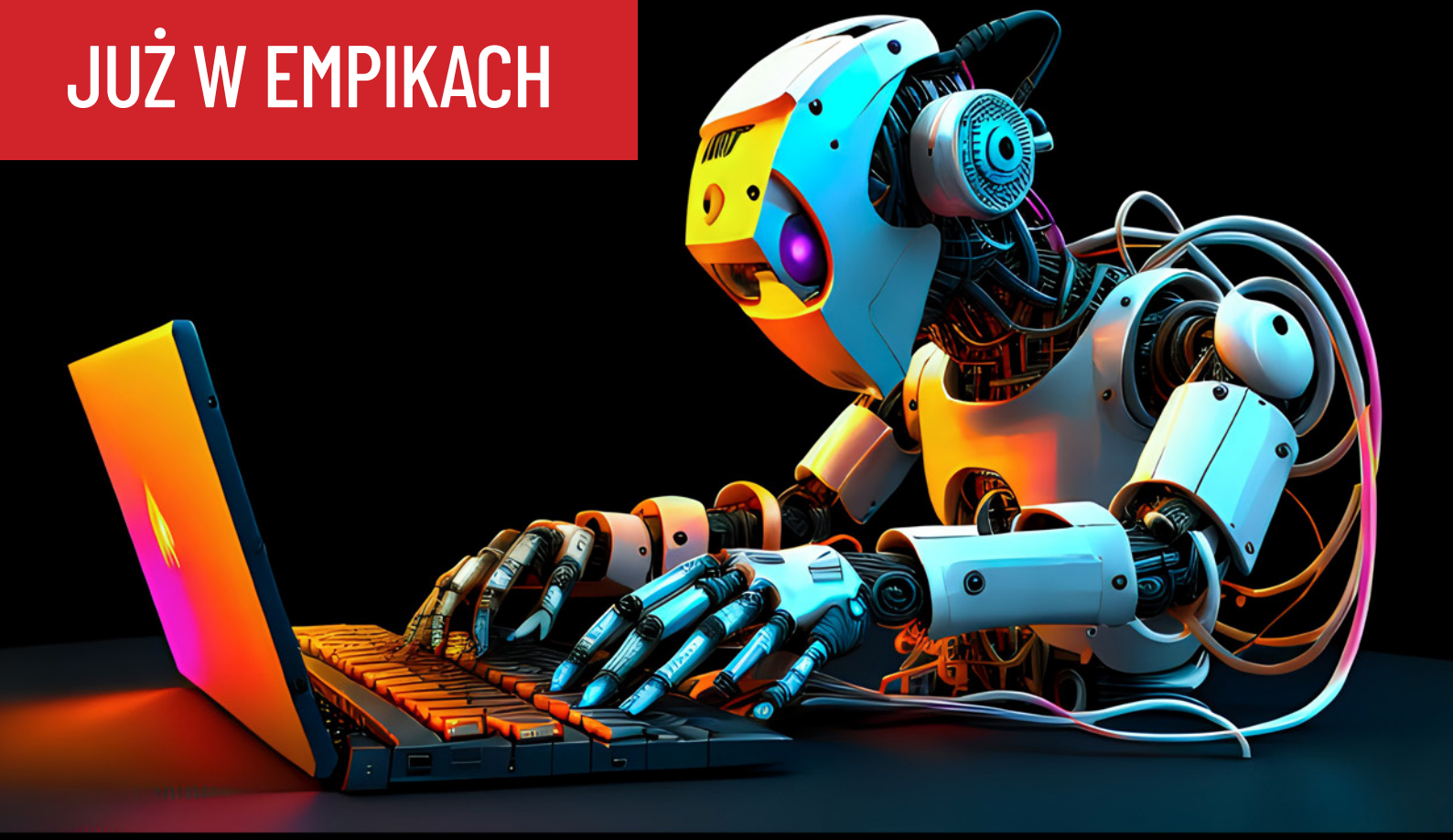
5/2023 (110)

Cena 28,90 zł (w tym VAT 8%)

EGZAMIN DLA MASZYNY

LLMY VS PROGRAMOWANIE

JUŻ W EMPIKACH



Wyszukiwanie zależności do obiektów bazodanowych w SQL Server

Konsola PlayDate okiem programisty

Jak bezpiecznie korzystać z HttpClient w .NET

Renderowanie animacji wektorowych

Wzorce projektowe

Część 2. Wzorce strukturalne i behawioralne

Czasami można mieć wrażenie, że takie zasady, jak wzorce projektowe, SOLID, KISS, DRY, YAGNI i jeszcze wiele innych, istnieją głównie po to, żeby móc w łatwy sposób odfiltrować kandydatów na rozmowach kwalifikacyjnych. Okazuje się jednak, że znajdują one zastosowanie również w rzeczywistych projektach, co starałem się pokazać w poprzednim artykule, przedstawiając pierwsze dziesięć z dwudziestu dwóch podstawowych wzorców. Dziś zobaczymy, jak wygląda sytuacja z pozostałą dwunastką.

WZORCE STRUKTURALNE

Wzorce strukturalne są najliczniejszą podgrupą wzorców projektowych. Dla przypomnienia, regulują one sposób, w jaki współdziałają ze sobą różne klasy w ramach aplikacji. Naszą podróż kontynuujemy, biorąc na warsztat dwa ostatnie wzorce strukturalne: Pyłek i Fasada.

Pyłek (Flyweight)

Mam wrażenie, że wzorzec Pyłek jest przykładem słowa za pięć dolarów opisującego koncepcję za pięć centów. Ogólnie rzecz biorąc, jego istotę można streścić w jednym zdaniu: niezmieniające się zasoby, które zajmują dużo pamięci, powinny być współdzielone, a nie kopiowane.

Wzorzec Pyłek znajduje zastosowanie, gdy operujemy na (relatywnie) dużej liczbie elementów, z których każdy można podzielić na dwie logiczne części: jedną, która pozostaje niezmienna, oraz drugą, która w czasie pracy aplikacji się zmienia. Jeżeli okaże się dodatkowo, że nie zmieniające się zasoby mogą być przez elementy współdzielone, otrzymujemy kanoniczny przypadek problemu rozwiązywanego przez opisywany wzorzec.

Prosty przykład implementacji wzorca strukturalnego Pyłek przedstawiono w Listingu 1.

Listing 1. Przykładowa implementacja wzorca Pyłek

```
public class FixedState
{

}

public class Flyweight
{
    private readonly FixedState fixedState;

    public Flyweight(FixedState fixedState)
    {
        this.fixedState = fixedState;
    }

    public bool MutableState { get; set; }
}
```

Przykład z życia

Jednym z moich projektów jest aplikacja opakowująca niektóre funkcjonalności TFS w wygodną w użyciu aplikację desktopową. Użytkownik może między innymi obejrzeć tablicę dla bieżącego sprintu zawierającą więcej informacji niż ta dostępna przez webowy interfejs lub skorzystać z menadżera zadania, który automatyzuje całą masę procesów związanych z implementowaniem zadań (na przykład tworzenie gałęzi git, przełączanie na nią, tworzenie pull requestów, zlecanie buildów i tak dalej).

Część funkcjonalności wymaga wyświetlenia informacji o użytkowniku – avatara oraz imienia i nazwiska. Aby uniknąć powielania (relatywnie ciężkich) obrazków, skorzystałem ze wzorca Pyłek. Wszystkie obrazy w postaci gotowego do użycia w WPF obiektu `ImageSource` trzymane są wewnątrz serwisu współpracującego z samym TFSem, zaś za każdym razem, gdy potrzebne są informacje o użytkowniku, zwracana jest tylko referencja do obiektu już zcache'owanego w pamięci.

Pod metodą `EnsureUsersCacheFor` znajduje się przykład klasy, która implementuje wzorzec Pyłek. Niemutowalną, „ciężką" częścią klasy jest pobierany z cache'u `ProjectUserModel`, zaś mutowalną i „lekką" jest informacja, czy jest on zaznaczony, czy też nie.

Listing 2. Aktualizowanie cache'u użytkowników

```
private async Task EnsureUsersCacheFor(string projectId)
{
    if (!usersCache.ContainsKey(projectId))
    {
        await semaphore.WaitAsync();
        List<WebApiTeam> teams;
        try
        {
            teams = await teamClient.GetAllTeamsAsync();
        }
        finally
        {
            semaphore.Release();
        }

        var members = new List<ProjectUserModel>();

        foreach (var team in teams)
        {
            IEnumerable<TeamMember> teamMembers;

            try
            {
                await semaphore.WaitAsync();
                try
                {
                    teamMembers = await teamClient
.GetTeamMembersWithExtendedPropertiesAsync(
                        projectId,
                        team.Id.ToString(),
                        null,
                        null,
                        null,
```

```
                    CancellationToken.None);
                }
                finally
                {
                    semaphore.Release();
                }

                foreach (var teamMember in teamMembers)
                {
                    if (!members.Any(m =>
                        m.Identity.Id ==
                            teamMember.Identity.Id))
                    {
                        ImageSource avatar =
                            GetAvatarFor(teamMember.Identity);
                        var user = new ProjectUserModel(
                            teamMember.Identity,
                            avatar);
                        members.Add(user);
                    }
                }
            }
            catch
            {
                // You may not have access to all team members
            }
        }

        members = members
            .OrderBy(m => m.Identity.DisplayName)
            .ToList();

        usersCache[projectId] = members;
    }
}

// W innym miejscu aplikacji

public class ReviewerViewModel : BaseViewModel
{
    private ProjectUserModel user;
    private bool isSelected;

    public ReviewerViewModel(ProjectUserModel user)
    {
        this.user = user;
    }

    public bool IsSelected
    {
        get => isSelected;
        set => Set(ref isSelected, value);
    }

    public string Display => user.Identity.DisplayName;
    public IdentityRef Identity => user.Identity;
    public ImageSource Avatar => user.Avatar;
}
```

Fasada (Facade)

Czym jest w rzeczywistym świecie fasada? Ogólnie rzecz biorąc, stanowi ona estetyczne pokrycie czegoś, co pod spodem jest surowe, toporne lub brzydkie. I tak samo jest w świecie wzorców projektowych: Fasady używamy w sytuacji, w której mamy do czynienia z jakimś trudnym lub rozwlekłym w użyciu API, aby dostarczyć interfejs bardziej estetyczny i łatwiejszy w użyciu.

Przykładową implementację wzorca zaprezentowano w Listingu 3. Mamy jakiś skomplikowany proces, który chcemy uprościć. W tym celu powołujemy do życia klasę `Facade`, która realizuje cały proces dla nas i zamyka go w pojedynczej, łatwej do wywołania metodzie.

Listing 3. Przykład implementacji wzorca Fasada

```
public class ComplicatedProcess
{
    public void Stage1()
    {
        Console.WriteLine("Stage 1");
    }

    public void Stage2()
    {
        Console.WriteLine("Stage 2");
    }

    public void Stage3()
    {
        Console.WriteLine("Stage 3");
    }
}

public class Facade
{
    private readonly ComplicatedProcess complicatedProcess;

    public Facade()
    {
        this.complicatedProcess = new ComplicatedProcess();
    }

    public void RunProcess()
    {
        complicatedProcess.Stage1();
        complicatedProcess.Stage2();
        complicatedProcess.Stage3();
    }
}

class Program
{
    static void Main(string[] args)
    {
        var facade = new Facade();
        facade.RunProcess();

        Console.ReadKey();
    }
}
```

Przykład z życia

Fasada jest chyba jednym z najłatwiejszych wzorców do odnalezienia wśród projektów z życia wziętych. Na dobrą sprawę każdy ORM jest pewnego rodzaju fasadą – odcina nas od SQLa, który jest na pewno bardziej skomplikowanym mechanizmem od zapytań pisanych bezpośrednio w kodzie – niezależnie od tego, czy musimy wywołać szereg metod, czy – jak w C# – użyć LINQ.

Wzorcowym – nomen omen – przykładem wzorca Fasada jest dostępna jako pakiet Nuget biblioteka ClosedXML. Stanowi ona opakowanie bibliotek Office Open XML dostarczanych przez Microsoft, a pozwalających na generowanie i przetwarzanie dokumentów pakietu Office: tekstowych, arkuszy i prezentacji. Te ostatnie okazują się bowiem stanowić zaledwie proste opakowanie XML i nie dają nawet gwarancji generowania prawidłowych dokumentów (trzeba samodzielnie dbać m.in. o kolejność elementów dodawanych do różnych list). ClosedXML wypełnia tę lukę, dostarczając znacznie bardziej wysokopoziomowego API do pracy z dokumentami, przez co proces ten staje się znacznie prostszy.

Również i mnie zdarzyło się napisać kawałek kodu, który idealnie pasuje do wzorca Fasada. Gdy kompilatory zaczęły powoli wspierać standard C++11, który przy pomocy typu `std::function` umożliwiał wreszcie wygodne zrealizowanie mechanizmu zdarzeń, zacząłem pisać autorski framework GUI mający być w założeniu czymś w rodzaju znanego z Delphi VCL lub znanego z C# Windows Forms w wersji dla C++. Projekt o nazwie Eleven – nawiązującej do nazwy nowego standardu C++ – pozostał niestety tylko w fazie prototypu, ale nawet w tym niewielkim zakresie, w jakim został zaimplementowany, pozwalał na radykalne uproszczenie pracy z WinAPI podczas projektowania interfejsu użytkownika.

Listing 4. Eleven – przykład użycia

```
#include <Windows.h>
#include <Application.h>
#include <Form.h>

using namespace Eleven;

void formMouseDown(Form* form,
    int x,
    int y,
    MouseButton button,
    ShiftState shiftState)
{
    MessageBox(form->GetHandle(),
        L"Hello, world!",
        L"Eleven",
        MB_OK | MB_ICONINFORMATION);
}

void formClose(Form* form, bool& doClose)
{
    Application::Terminate();
}

int WINAPI winMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    PWSTR pCmdLine,
    int nCmdShow)
{
    Form form(hInstance);
    form.SetText(L"Eleven");

    form.SetCloseHandler(formClose);
    form.SetMouseDownHandler(formMouseDown);

    Application::Run(form);
}
```

WZORCE BEHAWIORALNE

Wzorcem Fasada zakończyliśmy analizę wzorców strukturalnych. Ponieważ wzorce kreacyjne przerobiliśmy już w poprzedniej części artykułu, przechodzimy teraz do ostatniej kategorii, czyli wzorców behawioralnych.

Wzorce behawioralne dotyczą sytuacji, w której musimy wykonać jakąś operację, przeważnie związaną z określoną strukturą danych. O ile więc wzorce kreacyjne dotyczyły konstruowania obiektów, a strukturalne – określania pomiędzy tymi obiektami zależności, to wzorce behawioralne obracają się wokół algorytmów, które na tych obiektach wykonujemy.

Łańcuch zobowiązań (Chain of Responsibility)

To chyba jeden z moich ulubionych wzorców. Przypomina on taśmę produkcyjną, na którą trafia nieobrobiony element. Na każdym z etapów element ten może zostać poddany obróbce, jeżeli nie ma takiej konieczności – pozostawiony w spokoju, lub w ekstremalnym przypadku – na przykład gdy okaże się, że jest on uszkodzony – zdjęty z taśmy. Bardzo podobnie działa wzorzec Łańcucha zobowiązań.

Łańcuch zobowiązań warto zastosować wówczas, gdy scenariusz, który oprogramowujemy, przypomina właśnie taśmę produkcyjną. Otrzymujemy jakiś element na wejściu, a na każdym etapie możemy go przetworzyć, zignorować albo zatrzymać jego dalsze przetwarzanie.

Przykładowa implementacja Łańcucha zobowiązań może wyglądać następująco:

Listing 5. Wzorzec Łańcuch zobowiązań

```
public abstract class BaseAction
{
    public abstract void Process(ref int value);
}

public class AddAction : BaseAction
{
    private readonly int addedValue;

    public AddAction(int addedValue)
    {
        this.addedValue = addedValue;
    }

    public override void Process(ref int value)
    {
        value += addedValue;
    }
}

public class MultiplyAction : BaseAction
{
    private readonly int multipliedValue;

    public MultiplyAction(int multipliedValue)
    {
        this.multipliedValue = multipliedValue;
    }

    public override void Process(ref int value)
    {
        value *= multipliedValue;
    }
}

public static class Program
{
    public static void Main(string[] args)
    {
        int value = 12;

        List<BaseAction> actions = new()
        {
            new AddAction(5),
            new MultiplyAction(20),
            new AddAction(-10)
        };

        foreach (var action in actions)
            action.Process(ref value);

        Console.WriteLine(value);
        Console.ReadKey();
    }
}
```

Przykład z życia

W poprzedniej części artykułu wspomniałem o napisanym przeze mnie symulatorze kredytów. Oprócz wzorca Metoda kreacyjna znalazło się tam miejsce również dla Łańcucha zobowiązań.

Symulator generuje listę rat do spłacenia według zadanych mu kryteriów, ale oprócz tego pozwala zaaplikować do kredytu w czasie jego trwania różne akcje. Na chwilę obecną zaimplementowałem trzy: zmianę oprocentowania, wakacje kredytowe oraz nadpłatę kapitału.

Każda z akcji ma zdefiniowaną – w zależności od rodzaju – datę wystąpienia lub zakres dat, pomiędzy którymi akcja jest aktywna. Może się więc zdarzyć, że kilka akcji wystąpi naraz – w ekstremalnym przypadku mogą to być wszystkie trzy z nich. Co więcej, parametry kredytu zmieniają się pod działaniem akcji, co również należy uwzględnić. Na przykład w danym miesiącu może zmienić się oprocentowanie, co wpłynie na wysokość raty, ale jednocześnie kredytobiorca może skorzystać z wakacji kredytowych, co (niezależnie od poprzedniej zmiany) zmniejszy wysokość raty do zera. I właśnie tutaj znalazło się miejsce dla Łańcucha zobowiązań (z minimalną modyfikacją).

Każda z akcji dziedziczy po bazowej klasie `BaseAction`:

Listing 6. Klasa BaseAction

```
public abstract class BaseAction
{
    public BaseAction(int financialId)
    {
        FinancialId = financialId;
    }

    public abstract bool CheckIfApplies(int year, int month);

    public abstract void Apply(LoanModel loan,
        ref LoanDataModel loanData,
        int year,
        int month,
        List<string> notes);

    public int FinancialId { get; }
    public abstract int Priority { get; }
}
```

Dwiema kluczowymi metodami są `CheckIfApplies` oraz `Apply`. Zadaniem pierwszej jest sprawdzenie, czy akcja powinna zostać zaaplikowana w danym miesiącu, natomiast druga wykonuje akcję na kredycie. Bieżący stan na dany miesiąc znajduje się w klasie `LoanModelData`, która może zostać zmodyfikowana lub nawet zastąpiona inną instancją, jeżeli zajdzie taka potrzeba. Wreszcie każda z akcji może też dodać notatkę, która zostanie potem wyświetlona użytkownikowi w opisie danego miesiąca.

Aplikowanie akcji odbywa się w serwisie realizującym symulację.

Listing 7. Aplikowanie akcji do kredytu

```
private LoanStatusModel ApplyLoan(LoanModel loan,
    List<BaseAction>? actions,
    int year,
    int month)
{
    if (loan.Months <= 0 || loan.Amount <= 0)
        throw new ArgumentException(nameof(loan));

    LoanDataModel loanData =
        LoanCalculator.EvalLoanData(loan, year, month);

    List<string> notes = new();

    // Apply actions for this month
    if (actions != null)
    {
        foreach (var action in actions
            .Where(a => a.FinancialId == loan.Id &&
                a.CheckIfApplies(year, month))
            .OrderBy(a => a.Priority))
        {
            action.Apply(loan,
                ref loanData,
                year,
                month,
                notes);
        }
    }

    loan.Amount -= loanData.Capital;
    loan.Months -= 1;

    return new LoanStatusModel(loan.Id,
        loan.Name,
        loanData.Capital,
        loanData.Interest,
        loan.Amount,
        loan.MonthlyCost,
        notes);
}
```

Jedyną różnicą wobec „czystego" wzorca Łańcuch zobowiązań jest fakt, że lista akcji budowana jest dla każdego miesiąca na nowo: akcje, które nie pasują do tego okresu czasu, są odrzucane, a pozostałe sortowane są według ich priorytetów. Jest to istotne, bo akcje muszą być wykonywane w odpowiedniej kolejności (na przykład jeżeli po wakacjach kredytowych wykonamy zmianę oprocentowania, przywróciłaby ona niezerową wartość raty na dany miesiąc).

Polecenie (Command)

Wszystkie aplikacje mające interfejs użytkownika muszą zawierać jakąś koncepcję komend, czyli operacji, które użytkownik może im zlecać. Pojęcie komendy należy traktować bardzo szeroko, bo komendą może być operacja wykonana po wciśnięciu przycisku, ale równie dobrze i wyzwolona mechanizmem *drag&drop*, lub nawet przemieszczeniem kursora myszy ponad jakimś obszarem.

Często zdarza się, że ta sama komenda – być może z niewielkimi zmianami – wywoływana jest z poziomu kilku miejsc w aplikacji. Dobrym przykładem mogą być operacje kopiowania, wycinania i wklejania, które dostępne są z głównego menu, paska narzędzi lub wstążki i menu kontekstowego, a wywołane mogą być również przy pomocy skrótu klawiaturowego. Jeśli nie przyłożymy się do zaprojektowania odpowiedniego mechanizmu komend, możemy skończyć z tym samym kodem występującym w wielu miejscach, czasem tylko z niewielkimi zmianami (co jest chyba jeszcze gorsze).

Naturalnym rozwiązaniem powyższego problemu jest zastosowanie wzorca Polecenie. Polega on na opakowaniu komendy wraz ze wszystkimi informacjami potrzebnymi do jej wywołania w klasę. Gdy wówczas użytkownik wywołuje taką komendę, stworzony obiekt powinien zawierać wszystkie informacje, które są potrzebne do zrealizowania danego polecenia.

Wzorzec ten jest obecny w środowisku .NET w postaci interfejsu `ICommand` reprezentującego taką właśnie abstrakcyjną komendę. Jego implementacja może wyglądać następująco (zaprezentowany kod jest fragmentem biblioteki Spooksoft.VisualStateManager):

Listing 8. Przykładowa implementacja interfejsu ICommand

```
public class AppCommand : ICommand
{
    private readonly BaseCondition condition;
    private readonly Action<object> action;

    private void HandleConditionPropertyChanged(object sender,
        PropertyChangedEventArgs e)
    {
        if (e.PropertyName == nameof(BaseCondition.Value))
            CanExecuteChanged?.Invoke(this, EventArgs.Empty);
    }

    public AppCommand(Action<object> action,
        BaseCondition condition = null)
    {
        this.action = action;
        this.condition = condition;
        if (condition != null)
            condition.PropertyChanged +=
                HandleConditionPropertyChanged;
    }

    public bool CanExecute(object parameter)
    {
        return condition?.Value ?? true;
    }

    public void Execute(object parameter)
    {
        action(parameter);
    }

    public event EventHandler CanExecuteChanged;
}
```

Wzorzec Polecenie wymaga tylko, by obiekt komendy zawierał metodę `Execute`. Interfejs `ICommand` znany ze środowiska .NET wprowadza jednak dwa dodatkowe elementy. Po pierwsze, komendzie z poziomu definicji interfejsu użytkownika można przekazać parametr. Gdy na przykład piszemy aplikację kalkulatora, w której cyfry wprowadzane są przy pomocy przycisków, nie ma konieczności definiowania dziesięciu osobnych komend; wystarczy tylko jedna, otrzymująca poprzez parametr informację, który z przycisków został wciśnięty.

Druga zmiana polega na wprowadzeniu metody `CanExecute` oraz zdarzenia `CanExecuteChanged`. Pozwalają one na stwierdzenie, czy komendę można w danym momencie wywołać, oraz powiadamiają framework o sytuacji, gdy dostępność ta się zmieniła.

Przykład z życia

Ze wzorca Polecenie korzystam za każdym razem, gdy piszę aplikację w WPF. Poniżej zamieszczam przykład skorzystania z tej samej komendy w kilku miejscach w interfejsie użytkownika – jest to fragment mojej otwartoźródłowej aplikcji Dev.Editor.

Listing 9. Korzystamy ze wzorca Polecenie

```
<f:RibbonWindow>

  <f:RibbonWindow.InputBindings>
    (...)
    <KeyBinding Key="S" Modifiers="Ctrl"
                Command="{Binding SaveCommand}" />
    (...)
  </f:RibbonWindow.InputBindings>

  <!-- Main content -->

  <DockPanel Style="{StaticResource RootDockPanel}">

    <!-- Ribbon -->

    <f:Ribbon DockPanel.Dock="Top" Focusable="False"
         FocusManager.IsFocusScope="True">
        <f:Ribbon.Menu>
        <f:ApplicationMenu>
          (...)
          <f:MenuItem
            Header="{x:Static p:Strings.Ribbon_Home_File_Save}"
            Command="{Binding SaveCommand}" />
          (...)
        </f:ApplicationMenu>
      </f:Ribbon.Menu>

      <!-- Home tab -->

      <f:RibbonTabItem
        Header="{x:Static p:Strings.Ribbon_Tab_Home}"
        ReduceOrder="grHomeEdit, grHomeFile">

        <!-- File -->

        <f:RibbonGroupBox x:Name="grHomeFile">
          (...)
          <f:Button
            Header="{x:Static p:Strings.Ribbon_Home_File_Save}"
            Command="{Binding SaveCommand}" />
          (...)
        </f:RibbonGroupBox>
      (...)
    </f:Ribbon>
  (...)
  </DockPanel>
</f:RibbonWindow>
```

Dodajmy jeszcze, że komendę trzeba traktować bardzo ogólnie. Na przykład w środowisku webowym (powiedzmy ASP.NET) rolę komendy pełnić będzie request, zaś w aplikacji konsolowej może to być komenda wpisana przez użytkownika.

Iterator (Iterator)

Jeżeli ktoś programuje w .NET, implementację tego wzorca ogląda zapewne kilkanaście albo kilkadziesiąt razy dziennie.

Koncepcja jest bardzo prosta: mamy strukturę danych i chcemy przetworzyć wszystkie jej elementy. Problem polega na tym, że mechanizm ten ma działać niezależnie od tego, z jaką konkretną strukturą mamy do czynienia: listą bazowaną na tablicy, listą pojedynczo lub podwójnie dowiązaną, słownikiem, tablicą haszującą, drzewem, grafem – możliwości jest wiele.

Wzorzec Iterator rozwiązuje ten problem poprzez wprowadzenie jednolitego interfejsu umożliwiającego przechodzenie pomiędzy kolejnymi elementami danej struktury. Każda zaś ze struktur, która chce być zgodna z tym wzorcem, musi wówczas zaimplementować ów interfejs, dostarczając odpowiedniego dla niej rozwiązania.

Interfejsem takim w .NET jest oczywiście `IEnumerable` oraz jego generyczna wersja `IEnumerable<T>`. Interfejs ten wymusza na klasie dostarczenie instancji specjalnego obiektu iteratora: przy pomocy tego właśnie obiektu można pobrać bieżący element struktury oraz poprosić o przejście do kolejnego. Przy okazji tej ostatniej operacji iterator informuje też, czy został już osiągnięty koniec struktury.

Przy pomocy wzorca Iterator możemy również dostarczyć kilka różnych implementacji, jeżeli mają one sens dla danej struktury. Na przykład w przypadku drzewa i grafu możemy dostarczyć osobne implementacje iteratorów, które wykonują przeszukiwanie w głąb i wszerz.

Na potrzeby przykładu możemy zaprezentować implementację wzorca Iterator dla macierzy – jej elementy zwracane będą wiersz po wierszu.

Listing 10. Przykładowa implementacja wzorca Iterator

```
public class Matrix : IEnumerable<int>
{
    private class MatrixEnumerator : IEnumerator<int>
    {
        private int row, col;
        private int[,] data;

        public MatrixEnumerator(int[,] data)
        {
            this.data = data;
            row = 0;
            col = -1;
        }

        public int Current => data[col, row];

        object IEnumerator.Current => data[col, row];

        public void Dispose()
        {

        }

        public bool MoveNext()
        {
            col += 1;
            if (col >= data.GetLength(0))
            {
                row += 1;
                col = 0;
            }

            return row < data.GetLength(1);
        }

        public void Reset()
        {
            row = 0;
            col = -1;
        }
    }
```

```
    private int[,] data;

    public Matrix(int[,] data)
    {
        this.data = data;
    }

    public IEnumerator<int> GetEnumerator()
    {
        return new MatrixEnumerator(data);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return new MatrixEnumerator(data);
    }
}

public static class Program
{
    public static void Main(string[] args)
    {
        int[,] data = new int[3, 3];

        for (int col = 0; col < 3; col++)
            for (int row = 0; row < 3; row++)
                data[col, row] = 3 * row + col;

        var matrix = new Matrix(data);

        foreach (var entry in matrix)
        {
            Console.Write($"{entry} ");
        }

        Console.WriteLine();
        Console.ReadKey();
    }
}
```

Ujednolicenie dostępu do elementów dowolnej struktury danych stanowi bardzo potężne narzędzie, bo umożliwia również wykonywanie na ich danych standardowych operacji: filtrowania, konwersji, sortowania, pomijania i tak dalej. W środowisku .NET dostępne są nawet gotowe implementacje takich operacji, obecne jako metody rozszerzające interfejs `IEnumerable` (ang. *extension methods*).

Mediator (Mediator)

W momencie, gdy pisany przez nas program zaczyna się powoli rozrastać, coraz częściej dochodzi do sytuacji, w której różne jego komponenty potrzebują ze sobą współpracować. Nietrudno doprowadzić wtedy do sytuacji, w której pomiędzy różnymi klasami zaczynają pojawiać się chaotyczne zależności.

Wzorzec Mediator rozwiązuje ten problem poprzez wprowadzenie scentralizowanego mechanizmu, który pośredniczy w komunikacji pomiędzy różnymi komponentami. Zamiast więc komunikować się ze sobą nawzajem, komponenty korzystają do tego celu z Mediatora.

Przykładową implementację wzorca Mediator zaprezentowano w Listingu 11. Zwróćmy uwagę, że wzorzec opisuje tylko zależności pomiędzy komponentami, a nie logikę działania samego mediatora – ta zależna jest bowiem od konkretnego przypadku.

Listing 11. Przykładowa implementacja wzorca Mediator

```
public interface IComponent
{
    void React();
}

public class ComponentA : IComponent
{
    private Mediator mediator;

    public ComponentA(Mediator mediator)
    {
        this.mediator = mediator;
    }

    public void React()
    {
        Console.WriteLine("Reaction from component A");
    }

    public void Initiate()
    {
        mediator.Notify(this);
    }
}

public class ComponentB : IComponent
{
    private Mediator mediator;

    public ComponentB(Mediator mediator)
    {
        this.mediator = mediator;
    }

    public void React()
    {
        Console.WriteLine("Reaction from component B");
    }

    public void Initiate()
    {
        mediator.Notify(this);
    }
}

public class Mediator
{
    public void Notify(IComponent initiator)
    {
        foreach (var component in Components)
        {
            if (component != initiator)
            {
                component.React();
            }
        }
    }

    public List<IComponent> Components { get; } = new();
}

public static class Program
{
    public static void Main(string[] args)
    {
        var mediator = new Mediator();

        ComponentA componentA = new(mediator);
        ComponentB componentB = new(mediator);

        mediator.Components.Add(componentA);
        mediator.Components.Add(componentB);

        componentA.Initiate();

        Console.ReadKey();
    }
}
```

Przykład z życia

W moim edytorze programistycznym Dev.Editor stanąłem w pewnym momencie przed potrzebą rozwiązania problemu globalnych zdarzeń. Na przykład, gdy użytkownik zapisze parametry wyszukiwania lub zamiany, musi na to zareagować zarówno okno wyszukiwania, jak i główne okno aplikacji. Podobnie w momencie, gdy aplikacja staje się aktywna, kilka wizualnych komponentów musi odświeżyć wyświetlane informacje.

Oczywiście mógłbym powiązać wszystkie komponenty ze sobą i bezpośrednio powiadamiać je wzajemnie o zaszłych sytuacjach. Jednak to doprowadziłoby do ich silnego powiązania, a tego chciałem

uniknąć. Z tego względu wprowadziłem mechanizm globalnych zdarzeń, a klasa, która nimi zarządza – `EventBus` – stanowi właśnie formę mediatora pomiędzy wszystkimi zainteresowanymi komponentami. Gdy jeden z nich zarejestruje zajście jakiegoś zdarzenia, które może zainteresować innych, informuje o tym klasę `EventBus`. Ta z kolei rozsyła informację o zdarzeniu wszystkim pozostałym komponentom. Co ważne jednak, komponenty nie mają pojęcia o sobie nawzajem – komunikują się przez Mediatora.

Listing 12. Implementacja klasy EventBus pełniącej rolę Mediatora pomiędzy różnymi komponentami aplikacji (fragmenty)

```
internal class EventBus : IEventBus
{
    (...)

    private readonly Dictionary<Type, object> listeners =
        new Dictionary<Type, object>();

    (...)

    public void Send<T>(
        object sender,
        T @event)
        where T : BaseEvent
    {
        Type t = typeof(T);

        while (t != typeof(object))
        {
            if (listeners.ContainsKey(t))
            {
                List<BaseListenerInfo> eventListeners =
                    listeners[t] as List<BaseListenerInfo>;

                foreach (var info in eventListeners)
                {
                    if (info.NotifySelf ||
                        !info.ContainsListener(sender))
                        info.Receive(@event);
                }
            }

            t = t.BaseType;
        }
    }
}
```

Moja implementacja szyny zdarzeń stanowi Mediatora bardzo generycznego, ślepo przekazującego informacje z jednego komponentu do innych. W mojej sytuacji było to konieczne, nic nie stoi jednak na przeszkodzie, by Mediator był nieco bardziej inteligentny i przekazywał informacje wszystkim konkretnym typom komponentów lub wręcz do konkretnych, znanych przez niego instancji.

Pamiątka (Memento)

Operacja „Cofnij" jest chyba jedną z najbardziej pożytecznych funkcji dostępnych w każdym programie. Naturą ludzką jest popełnianie błędów, więc miło jest mieć do dyspozycji mechanizm, który pomaga błędy te szybko i bez wysiłku wycofywać.

Implementacja mechanizmu wycofywania wprowadzonych zmian nie należy jednak do najprostszych. Przede wszystkim musimy bardzo starannie śledzić, jakie zmiany zostały wprowadzone, by móc je potem na życzenie użytkownika w odwrotnej kolejności anulować. Pociąga to za sobą konieczność przechowywania dużych ilości informacji – jeżeli bowiem na przykład akcją użytkownika było usunięcie jakiegoś elementu, jej wycofanie wymaga przywrócenia go w takim stanie, w jakim znajdował się przed usunięciem.

Wzorzec Pamiątka ma na celu rozwiązanie tego problemu. Koncepcja polega na tym, by w dowolnym momencie móc wykonać zrzut całego edytowanego modelu, aby można było go potem w takim samym stanie odtworzyć. Nie jest to może zbyt odkrywcze, ale diabeł tkwi w szczegółach: wzorzec Pamiątka kładzie duży nacisk na to, by zawartość obiektu pamiątki – czyli takiego właśnie zrzutu – była dostępna tylko dla obiektu, który zrzuty takie tworzy i aplikuje.

Listing 13. Przykład zastosowania wzorca Pamiątka

```
public class Memento
{
    internal int Value { get; set; }
}

public class Container
{
    public int Value { get; set; }

    public Memento Store()
    {
        return new Memento
        {
            Value = Value
        };
    }

    public void Restore(Memento memento)
    {
        Value = memento.Value;
    }
}

public static class Program
{
    public static void Main(string[] args)
    {
        Container container = new Container();
        container.Value = 123;

        var memento = container.Store();
        // Tu nie powinniśmy mieć dostępu
        // do memento.Value

        container.Value = 456;
        container.Restore(memento);

        Console.WriteLine(container.Value);
        Console.ReadKey();
    }
}
```

W powyższym przykładzie izolacja wewnętrznych danych pamiątki odbywa się poprzez zastosowanie akcesora `internal`. W domyśle użytkownicy klas `Container` i `Memento` powinni znajdować się w osobnym zestawie (ang. *assembly*), co uniemożliwiłoby (bezpośrednią) ingerencję w dane przechowane w Pamiątce.

Przykład z życia

Jakiś czas temu na tych łamach pisałem o koncepcji Smart Modelu (Programista 3/2022 (102)). Mowa o mechanizmie, który automatycznie rejestruje i przechowuje wszystkie zmiany, które w nim zachodzą. Udostępnia on również możliwość wycofywania i ponawiania zmian. Mechanizm zaimplementowany wewnątrz Smart Modelu stanowi formę implementacji wzorca Pamiątka, choć z kilkoma zmianami.

Po pierwsze, obiekty pamiątek są całkowicie izolowane od użytkowników klasy. Mogą oni wycofywać zmiany i ponawiać je, ale nie mają bezpośredniej możliwości przechowywania poszczególnych zrzutów. W zamian przechowywane są one wszystkie w klasie o nazwie `History`, która dba o to, aby operacje na historii nie doprowa-

dziły do niespójnego stanu modelu (na przykład takiego, w którym tylko część zmian została zaaplikowana lub przywrócona).

Po drugie, obiekt historii przechowuje informacje tylko o zmienionych elementach, podczas gdy wzorzec Pamiątka sugeruje zrobienie kompletnego zrzutu całego modelu. Moje rozwiązanie ma jednak tę zaletę, że znacząco zmniejsza zużycie pamięci, szczególnie w przypadkach, gdy model jest duży, a zmiany małe.

Jak wspomniałem, o Smart Modelu pisałem bardziej szczegółowo w innym artykule, więc przytoczę tylko, w jaki sposób można użyć mechanizmu implementującego wzorzec Pamiątka:

Listing 14. Korzystamy ze wzorca Pamiątka do wycofywania i przywracania zmian

```
[TestMethod, TestCategory("Simple scope tests")]
public void ScopeTest1()
{
    // Arrange
    var obj = new SimpleDocument();
    obj.IntProp = 10;
    obj.StringProp = "Test";
    var history = new History();
    obj.History = history;

    // Act
    using (history.StartUndoScope())
    {
        obj.IntProp = 20;
        obj.StringProp = "Test 2";
    }

    // Assert
    history.Undo();
    Assert.AreEqual(10, obj.IntProp);
    Assert.AreEqual("Test", obj.StringProp);

    history.Dispose();
}
```

Obserwator (Observer)

Obserwator jest mechanizmem pozwalającym na wprowadzenie ujednoliconego powiadamiania o zmianach, które zaszły w danym obiekcie. Wprowadza on dwa byty: producenta i subskrybenta. Producent jest klasą, która generuje jakieś dane oraz pozwala rejestrować się subskrybentom implementującym odpowiedni interfejs. Ci ostatni natomiast stanowią konsumentów danych generowanych przez producenta.

Programiści .NET i Delphi są w tej kwestii jak pączki w maśle, bo ich języki mają wbudowaną implementację tego wzorca w postaci zdarzeń. Do tych ostatnich można podwieszać metody, które wywoływane są w momencie, gdy zajdą określone okoliczności. A jeżeli konieczna jest bardziej granularna kontrola nad publikowaniem i konsumowaniem danych, zawsze można skorzystać z interfejsów `IObservable` oraz `IObserver`. Albo też, oczywiście, napisać inne, dedykowane rozwiązanie.

Przykładową implementację wzorca Obserwator przy pomocy interfejsów dostępnych w .NET możemy zobaczyć poniżej.

Listing 15. Przykładowa implementacja wzorca Obserwator

```
public class Producer : IObservable<int>
{
    private class Unsubscriber : IDisposable
    {
        private readonly IObserver<int> observer;
        private readonly List<IObserver<int>> observers;

        public Unsubscriber(IObserver<int> observer,
            List<IObserver<int>> observers)
        {
            this.observer = observer;
            this.observers = observers;
        }

        public void Dispose()
        {
            observers.Remove(observer);
        }
    }

    private readonly List<IObserver<int>> observers =
        new List<IObserver<int>>();

    public IDisposable Subscribe(IObserver<int> observer)
    {
        observers.Add(observer);
        return new Unsubscriber(observer, observers);
    }

    public void Produce(int count)
    {
        Random random = new Random();

        for (int i = 0; i < count; i++)
        {
            var value = random.Next(100);
            foreach (var observer in observers)
            {
                observer.OnNext(value);
            }
        }
    }
}

public class Subscriber : IObserver<int>
{
    public void OnCompleted()
    {
        Console.WriteLine("Sequence completed!");
    }

    public void OnError(Exception error)
    {
        Console.WriteLine($"Error: {error}");
    }

    public void OnNext(int value)
    {
        Console.WriteLine($"Received: {value}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Producer producer = new Producer();
        Subscriber subscriber = new Subscriber();

        using (var subscription = producer
            .Subscribe(subscriber))
        {
            producer.Produce(10);
        }

        producer.Produce(10);

        Console.ReadKey();
    }
}
```

Przykład z życia

Tym razem pójdzie bardzo łatwo. Pamiętacie klasę `EventBus`, o której pisałem w kontekście wzorca Mediator? Sama klasa pełni faktycznie rolę mediatora, ale jej interakcja z komponentami nasłuchującymi globalnych zdarzeń stanowi właśnie implementację wzorca Obserwator. Zobaczmy:

Listing 16. Rejestracja obserwatora

```
public ExplorerToolViewModel(
    IFileIconProvider fileIconProvider,
    IImageResources imageResources,
    IConfigurationService configurationService,
    IExplorerHandler handler,
    IEventBus eventBus,
    IPlatformService platformService)
    : base(handler)
{
    this.fileIconProvider = fileIconProvider;
    this.imageResources = imageResources;
    this.configurationService = configurationService;
    this.explorerHandler = handler;
    this.eventBus = eventBus;
    this.platformService = platformService;

    eventBus.Register(
        (IEventListener<ApplicationActivatedEvent>)this);

    // (...)
}
```

Listing 17. Odbieranie zdarzenia

```
void IEventListener<ApplicationActivatedEvent>.Receive(
    ApplicationActivatedEvent @event)
{
    RefreshFolders();
}
```

Stan (State)

Wzorzec Stan należy do grupy wzorców bardzo niszowych, ponieważ muszą zajść bardzo specyficzne okoliczności, by zaistniała potrzeba jego zastosowania. Dotyczy on bowiem tych sytuacji, w których mamy do czynienia z jakimś rodzajem maszyny stanu.

Maszyna stanu to bardzo prosty koncept programistyczny; mówimy o obiekcie, który ma zdefiniowany pewien zbiór możliwych stanów, z których tylko jeden naraz może być aktywny. Dodatkowo określone są również ścisłe reguły, które definiują, w jakich okolicznościach możliwa jest zmiana stanu na inny. Każda z tych reguł ma zawsze postać: „jeżeli bieżący stan to X oraz zaszło zdarzenie Y, stan musi zostać zmieniony na Z".

Wzorzec Stan proponuje, żeby każdy ze stanów opakować w osobną klasę, która implementować będzie wszystkie te reguły dla danego stanu. Przykładową implementację zaprezentowano w Listingu 18.

Listing 18. Przykład implementacji wzorca Stan

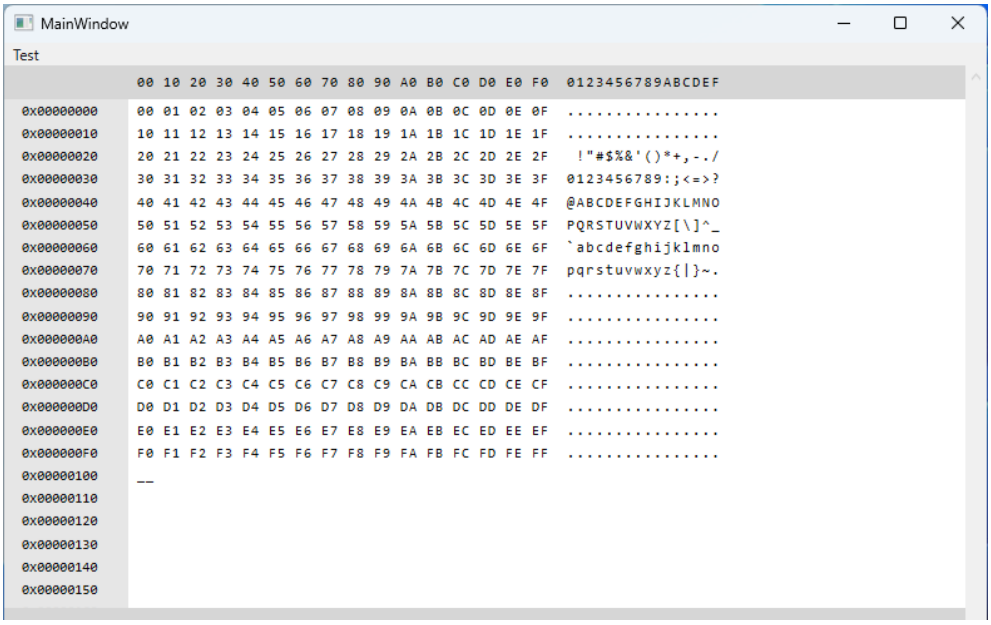
```
public abstract class ButtonState
{
    protected readonly Button button;

    public ButtonState(Button button)
    {
        this.button = button;
    }

    public abstract void MouseDown();
    public abstract void MouseUp();
}

public class NormalState : ButtonState
{
    public NormalState(Button button)
        : base(button)
    {

    }

    public override void MouseDown()
    {
        button.State = new PressedState(button);
    }

    public override void MouseUp()
    {
        // Nothing happens
    }
}

public class PressedState : ButtonState
{
    public PressedState(Button button)
        : base(button)
    {

    }

    public override void MouseDown()
    {
        // Nothing happens
    }

    public override void MouseUp()
    {
        Console.WriteLine("Button pressed!");
        button.State = new NormalState(button);
    }
}

public class Button
{
    private ButtonState state;

    public Button()
    {
        state = new NormalState(this);
    }

    public void MouseDown()
    {
        state.MouseDown();
    }

    public void MouseUp()
    {
        state.MouseUp();
    }

    public ButtonState State
    {
        get => state;
        set => state = value;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Button button = new Button();
        button.MouseDown();
        button.MouseUp();

        Console.ReadKey();
    }
}
```

Przykład z życia

Jednymi z najbardziej oczywistych maszyn stanu są elementy interfejsu użytkownika. Na przykład przycisk może być w stanie normalnym, nieaktywnym, wciśniętym oraz wciśniętym, gdy mysz opuściła jego obszar (puszczenie przycisku myszy nie spowoduje wtedy wywołania akcji przycisku). Zdarzeniami wpływającymi na zmianę stanu jest wciskanie i puszczanie przycisków myszy oraz jej przemieszczanie po ekranie.

W życiu miałem okazję zaimplementować całkiem sporo kontrolek użytkownika i praktycznie w każdym przypadku były to maszyny stanu. Ostatnią napisaną przeze mnie kontrolką jest hexedytor dla WPF, którego potrzebowałem podczas pisania Dev.Editora (Rysunek 1).

Jeżeli rozważymy, jakie stany może przyjmować hexedytor z perspektywy operowania na nim myszą, dojdziemy do wniosku, że mamy trzy możliwości:



Rysunek 1. Kontrolka hexedytora dla WPF

- Bezczynny (*de facto* każda kontrolka ma taki stan).
- Zaznaczanie wartości heksadecymalnych.
- Zaznaczanie znaków.

Jeżeli na przykład hexedytor jest w stanie bezczynnym, a my wciśniemy lewy przycisk myszy nad obszarem wartości heksadecymalnych, kontrolka przejdzie w stan zaznaczania wartości heksadecymalnych. Puszczenie przycisku myszy powoduje powrót do stanu bezczynnego.

W Listingu 19 przedstawiono kluczowe fragmenty implementacji wzorca Stan w kontrolce hexedytora. Pominąłem część implementacji, ponieważ zawiera ona obliczenia związane z położeniem kursora myszy oraz logikę związaną z obsługą zdarzeń. Link do repozytorium zawierającego kompletny kod kontrolki znajduje się na końcu artykułu.

Listing 19. Implementacja wzorca Stan w kontrolce hexedytora (istotne fragmenty)

```
private abstract class MouseState
{
    protected BaseMouseHitInfo GetMouseHit(
        HexEditorDisplay display,
        PixelPoint point)
    {
        // (...)
    }

    public virtual void MouseDown(
        HexEditorDisplay control,
        Point point)
    {

    }

    public virtual void MouseMove(
        HexEditorDisplay control,
        Point point)
    {

    }

    public virtual void MouseUp(
        HexEditorDisplay control,
        Point point)
    {

    }

    public static MouseState Idle { get; } =
        new IdleState();
    public static MouseState HexSelection { get; } =
        new HexSelectionState();
    public static MouseState CharSelection { get; } =
        new CharSelectionState();
}

private class IdleState : MouseState
{
    public override void MouseDown(
        HexEditorDisplay control,
        Point point)
    {
        // (...)
    }
}

private class HexSelectionState : MouseState
{
    public override void MouseMove(
        HexEditorDisplay control,
        Point point)
    {
        // (...)
    }

    public override void MouseUp(
        HexEditorDisplay control,
        Point point)
    {
        control.mouseState = MouseState.Idle;
        control.mouseData = null;

        control.InvalidateVisual();
    }
}

private class CharSelectionState : MouseState
{
    public override void MouseMove(
        HexEditorDisplay control,
        Point point)
    {
        // (...)
    }

    public override void MouseUp(
        HexEditorDisplay control,
        Point point)
    {
        control.mouseState = MouseState.Idle;
        control.mouseData = null;

        control.InvalidateVisual();
    }
}
```

W odróżnieniu od kanonicznej wersji wzorca zdecydowałem się przechowywać wszystkie dostępne stany w osobnym repozytorium i używać je wielokrotnie, przekazując instancję kontrolki do wywołań zamiast za każdym razem tworzyć obiekty stanów na nowo. Stany w kontrolce zmieniają się dosyć często, więc alokowanie pamięci za każdym razem wpłynęłoby negatywnie na jej wydajność. Wszelkie dane dotyczące stanu znajdują się w polu `mouseData` i instancjonowane są tylko wtedy, gdy zajdzie taka potrzeba (na przykład stan `Idle` nie potrzebuje żadnych danych).

Metoda szablonowa (Template method)

Metoda szablonowa jest stosunkowo łatwym do zaimplementowania wzorcem, o którym warto pamiętać, gdy implementujemy kilka dużych, ale niewiele się od siebie różniących procesów. Koncepcja Metody szablonowej polega na tym, by w klasie bazowej zaimplementować większą część procesu, pozostawiając niektóre metody jako wirtualne lub abstrakcyjne: odpowiadają one za te fragmenty procesu, które są zmienne. Możemy wtedy odziedziczyć po tej klasie bazowej, wymieniając w razie potrzeby implementację tychże metod na inną.

Przykładowa implementacja Metody szablonowej może wyglądać tak:

Listing 20. Implementacja Metody szablonowej

```
public abstract class BaseSorter
{
    protected abstract int Compare(int a, int b);

    public void Sort(List<int> list)
    {
        for (int i = 0; i < list.Count - 1; i++)
            for (int j = i; j < list.Count; j++)
            {
                if (Compare(list[i], list[j]) < 0)
                {
                    (list[i], list[j]) =
                        (list[j], list[i]);
                }
            }
    }
}

public class AscendingSorter : BaseSorter
{
    protected override int Compare(int a, int b)
    {
        return b - a;
    }
}

class Program
{
    static void Main(string[] args)
    {
        List<int> data = new List<int>
        {
            9, 4, 5, 1, 8, 2, 6, 3, 7, 0
        };

        AscendingSorter sorter = new AscendingSorter();
        sorter.Sort(data);

        Console.WriteLine(String.Join(", ", data));

        Console.ReadKey();
    }
}
```

Przykład z życia

Wśród moich prywatnych projektów jednym z bodaj najbardziej funkcjonalnych jest Board. Stanowi on uproszczony klon serwisu Trello: pozwala konstruować tablice, w nich kolumny, a w tych z kolei notatki o dowolnej treści. Notatkom można przyporządkowywać daty, tagi, dodawać opisy formatowane przy pomocy Markdown czy wreszcie pozostawiać pod nimi komentarze. Różnica w stosunku do popularnego serwisu polega na tym, że wszystkie dane zapisywane są w lokalnej bazie SQLite, co pozwala na przykład swobodnie przechowywać tam wrażliwe dane.

Zależało mi bardzo na zaimplementowaniu mechanizmu zmiany kolejności kart w obrębie kolumny. Pojawił się jednak problem – w jaki sposób kolejność tę przechować również w bazie danych?

Aby go rozwiązać, zbudowałem mechanizm, który przydziela wszystkim kartom wartość porządkową, według której są one potem sortowane. Gdy jedna z kart zostanie przeniesiona w inne miejsce, mechanizm ten wykonuje szereg operacji, aby przywrócić odpowiednie wartości pozycji (w najgorszym przypadku wykonując ponownie numerację wszystkich kart w tej kolumnie od nowa).

Haczyk polegał na tym, że oprócz kolejności kart, chciałem móc również zmieniać kolejność kolumn oraz całych tablic. Mechanizm pozostawał więc ten sam, ale z oczywistych powodów za każdym razem operowałem na innych modelach. Klasę opiekującą się wartościami pozycji zaimplementowałem więc przy pomocy wzorca Metoda szablonowa: algorytm pozostał w klasie bazowej, zaś klasy pochodne musiały dostarczyć implementacji dostosowanej do konkretnego modelu, na którym numeracja się odbywała.

Listing 21. Klasa bazowa poprawiająca numerację modeli

```
public abstract class BaseEntityOrdering<TModel>
    where TModel : IOrderedModel
{
    private const long ORDER_STEP = 1024;
    private const int REORDER_BATCH = 50;

    /// <summary>Returns order value of
    /// last item ( = greatest order value)</summary>
    protected abstract long GetLastOrderValue(int groupId);

    /// <summary>Returns order value of
    /// first item ( = smallest order value)</summary>
    protected abstract long GetFirstOrderValue(int groupId);

    /// <summary>Returns count of ordered items</summary>
    protected abstract int GetModelCount(int groupId);

    /// <summary>Returns a list of models ordered by
    /// the order field</summary>
    protected abstract List<TModel> GetOrderedModels(
        int groupId,
        int skip,
        int take);

    /// <summary>Returns model at index
    /// <paramref name="index"/> and its
    /// immediate successor.</summary>
    protected abstract (TModel indexModel,
        TModel nextModel) GetModelWithSuccessor(
        int groupId,
        int index);

    /// <summary>Persists changes made to given
    /// list of models.</summary>
    protected abstract void UpdateItems(
        int groupId,
        List<TModel> updatedItems);

    /// <summary>
    /// Sets order field of given model. If necessary,
    /// reorders existing items so that new order value
    /// can be properly generated.
    /// </summary>
    public void SetNewOrder(
        TModel model,
        int desiredPosition,
        int groupId)
    {
        // Tu znajduje się implementacja algorytmu
    }
}
```

Algorytm jest dosyć skomplikowany, ale nie odgrywa większej roli w implementacji samego wzorca, więc go pominąłem. Aplikacja Board jest jednak otwartoźródłowa, zatem zainteresowanych odsyłam do jej repozytorium (link znajduje się na końcu artykułu).

Strategia (Strategy)

Na swój sposób wzorzec Strategia stanowi rozszerzenie Metody szablonowej. O ile w Metodzie szablonowej algorytm był stały, a zmieniały się tylko jego fragmenty, to tym razem mamy możliwość wymieniać całe algorytmy pracujące w pewnym kontekście.

Wzorzec Strategia składa się z dwóch kluczowych elementów: Interfejsu strategii, który reprezentuje realizację jakiejś podstawowej funkcjonalności, oraz obiektu kontekstu, któremu przekazywane są instancje klas implementujących wspomniany wcześniej interfejs. Gdy użytkownik poprosi kontekst o zrealizowanie jakiegoś zadania, ten będzie mógł w trakcie tego procesu wykorzystać przekazane mu instancje do realizacji poszczególnych jego etapów.

Kluczowym aspektem wzorca Strategia jest fakt, iż implementacje strategii, którymi operuje kontekst, można wymieniać w trakcie działania aplikacji, co pozwala precyzyjnie sterować przebiegiem całego procesu.

Jeżeli powyższy opis wydaje się być zawiły, spróbujmy zobaczyć, jak może wyglądać przykładowa implementacja wzorca Strategia:

Listing 22. Przykładowa implementacja wzorca Strategia

```
public interface IGeneratorStrategy
{
    int GenerateValue();
}

public interface IOutputStrategy
{
    void Output(int value);
}

public class RandomGeneratorStrategy
    : IGeneratorStrategy
{
    private Random random = new Random();

    public int GenerateValue()
    {
        return random.Next();
    }
}

public class ConsoleOutputStrategy
    : IOutputStrategy
{
    public void Output(int value)
    {
        Console.WriteLine(value);
    }
}

public class Context
{
    public IGeneratorStrategy Generator
    {
        get;
        set;
    }

    public IOutputStrategy Output
    {
        get;
        set;
    }
    public void Process()
    {
        int value = Generator.GenerateValue();
        Output.Output(value);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var context = new Context();
        context.Generator =
            new RandomGeneratorStrategy();
        context.Output =
            new ConsoleOutputStrategy();

        context.Process();

        Console.ReadKey();
    }
}
```

Przykład z życia

Byłem przekonany, że nigdzie w moich projektach nie znajdę implementacji tego wzorca. Ba! Podczas pracy nad niniejszym artykułem zostawiłem go sobie nawet na koniec, przekonany, że takiej implementacji nie znajdę i będę musiał poszukać jakiegoś przykładu z zewnętrznych projektów. A tymczasem okazało się, że i Strategię miałem już okazję zaimplementować, a co zabawniejsze – w projekcie, którego używam chyba najczęściej.

Mowa o Z, aplikacji-wyrzutni, zastępującej systemowe menu Start (Rysunek 2). Sposób jej działania jest prosty: użytkownik wpisuje w wyskakującym okienku dowolne wyrażenie, a aplikacja stara się zgadnąć, co użytkownik próbuje znaleźć – i wyświetla mu pasujące wyniki. Obszarów przeszukiwania jest sporo: aplikacja potrafi przeglądać system plików, aplikacje w menu Start i na pulpicie, wpisy Panelu Sterowania, podkatalogi w wyznaczonych katalogach. Oprócz tego przeszukuje również uruchomione procesy i pozwala je zamykać, ma wbudowany kalkulator, obsługuje skróty użytkownika (można na przykład ustalić słowo kluczowe i przy jego pomocy otworzyć stronę internetową) i jeszcze kilka innych, mniej lub bardziej przydatnych funkcji.

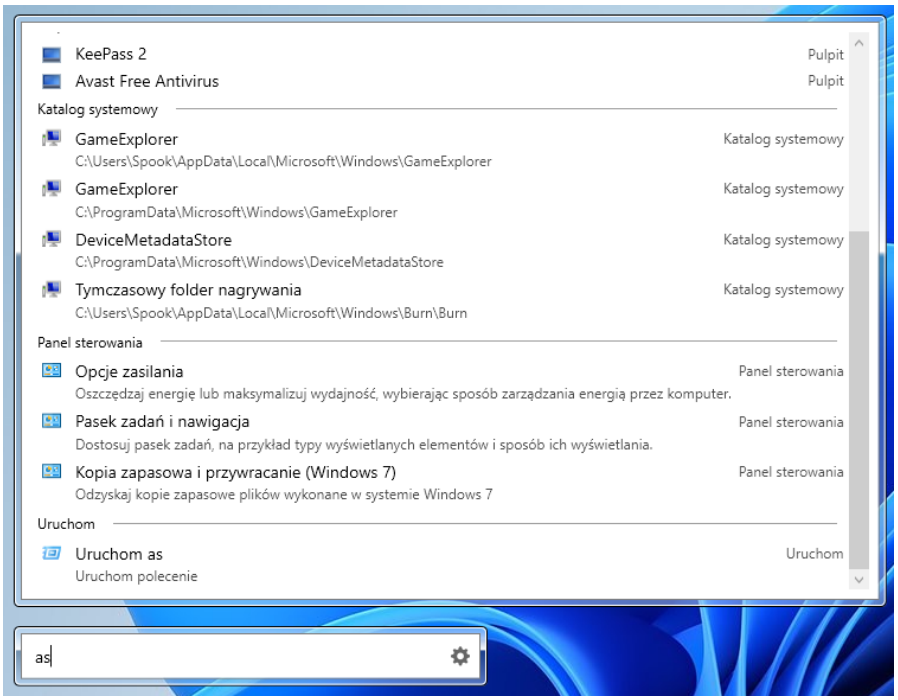
Jak można się łatwo domyślić, algorytmy realizujące wyszukiwanie w konkretnych obszarach dostarczane są aplikacji w postaci wtyczek. Sama aplikacja niczego nie umie szukać, wszystko zleca zarejestrowanym podczas jej uruchamiania modułom. I tu właśnie widzimy zastosowanie wzorca Strategia. Z jedną tylko modyfikacją – moja aplikacja nie korzysta z jednej, ale z całego zbioru strategii, uruchamiając je jedną po drugiej.

Listing 23. Implementacja wzorca Strategia w wyrzutni Z

```
class ModuleService : IModuleService,
    IEventListener<ShuttingDownEvent>
{
    // (...)

    private void InitDefaultModules()
    {
        AddModule(new WebSearchModule.Module());
        AddModule(new Filesystem.Module());
        AddModule(new ControlPanelModule.Module());
        AddModule(new StartMenuModule.Module());
        AddModule(new ProCalcModule.Module());
        AddModule(new PowerModule.Module());
        AddModule(new DesktopModule.Module());
        AddModule(new CustomCommandsModule.Module());
        AddModule(new ProjectsModule.Module());
        AddModule(new ShellFoldersModule.Module());
        AddModule(new ProcessModule.Module());
        AddModule(new HashModule.Module());
        AddModule(new RunModule.Module());
        AddModule(new FavoritesModule.Module());
```

```
    AddModule(new MorseModule.Module());
}

// (...)

protected virtual void OnModulesChanged()
{
    eventBus.Send(new ModulesChangedEvent());
}

// (...)

public IZModule GetModule(string internalName)
{
    return modules
        .SingleOrDefault(m => m.Name == internalName);
}

public ModuleService(
    IPathService pathService,
    IEventBus eventBus)
{
    this.pathService = pathService;
    this.eventBus = eventBus;

    eventBus.Register(
        (IEventListener<ShuttingDownEvent>)this);

    modules = new List<IZModule>();
    InitDefaultModules();
    LoadPluginModules();
}

// (...)

public List<SuggestionData> GetSuggestionsFor(
    string text,
    KeywordData keyword,
    bool perfectMatchesOnly = false)
{
    var suggestions = new List<SuggestionData>();

    var exclusiveModule = modules
        .FirstOrDefault(m =>
            m is IZExclusiveSuggestions &&
            ((IZExclusiveSuggestions) m)
                .IsExclusiveText(text));

    if (keyword != null)
    {
        // (...)
    }
    else
    {
        var source = exclusiveModule != null ?
            new List<IZModule> { exclusiveModule } :
                modules;

        foreach (var module in source)
        {
            using (var collector =
                new SuggestionCollector(
                    suggestions,
                    module))
            {
                module.CollectSuggestions(
                    text,
                    null,
                    perfectMatchesOnly,
                    collector);
            }
        }
    }

    return suggestions;
}

public void AddModule(IZModule module)
{
    // (...)

    modules.Add(module);

    if (module is IZInitializable)
        ((IZInitializable) module)
            .Initialize(new ModuleContext(
                pathService, module.Name));

    OnModulesChanged();
}

public int ModuleCount => modules.Count;
}
```

Oczywiście musiałem nieco skrócić zaprezentowany przykład, ponieważ zawiera dużo kodu infrastrukturalnego. Widzimy jednak metodę `AddModule`, która rejestruje kolejną strategię (w nomenklaturze aplikacji – moduł), oraz `GetSuggestionsFor`, która realizuje wyszukiwanie, wywołując zarejestrowane strategie.



Rysunek 2. Wyrzutnia Z

Odwiedzający (Visitor)

Wśród różnych struktur danych i, bardziej ogólnie, źródeł danych istnieją takie, których trawersowanie jest sporym problemem. Prostym przykładem może być graf: gdy chcemy przejść po jego elementach, już na początku stajemy przed wyborem sposobu – wszerz, w głąb lub jeszcze innego, zależnego od sytuacji. Innym przykładem może być też system plików: mamy tu dyski lokalne, sieciowe, przenośne, podłączone urządzenia, które nie mają swojej litery dysku (na przykład telefony), i wreszcie lokalizacje wirtualne, które wyświetlają się jako dysk, ale tak naprawdę nim nie są.

Kiedy chcemy przetworzyć dane znajdujące się w takim skomplikowanym źródle danych, stajemy zwykle przed koniecznością zaimplementowania dwóch skomplikowanych mechanizmów: jednego, który trawersuje dane, i drugiego, który je przetwarza. Wzorzec Odwiedzający pozwala na odseparowanie obu mechanizmów, przeciwdziałając powstaniu trudnego do rozwikłania kodu-spaghetti.

Kanoniczne rozwiązanie polega na wykonaniu trzech kroków. Po pierwsze, przygotowujemy interfejs odwiedzającego, w którym znajdują się metody odpowiadające danym, które chcemy przetwarzać. Następnym krokiem jest napisanie klasy, której zadaniem jest trawersowanie źródła danych. Metoda inicjująca przechodzenie po elementach otrzymuje instancję klasy implementującej przygotowany wcześniej interfejs i wywołuje jego metody na każdym odwiedzanym elemencie. Na koniec musimy napisać klasę implementującą interfejs odwiedzającego, w której nastąpi faktyczne przetworzenie analizowanych danych.

Pewną wariacją na temat jest niewielkie uproszczenie powyższego mechanizmu polegające na scaleniu klasy trawersującej źródło danych i interfejsu odwiedzającego. W takim przypadku metody odwiedzające poszczególne elementy dostarczane są wtedy jako wirtualne lub abstrakcyjne, zaś mechanizm przetwarzający dane musi po tej klasie odziedziczyć.

Listing 24. Przykładowa implementacja wzorca Odwiedzający

```
public interface IVisitor
{
    void VisitInt(int value);
    void VisitDouble(double value);
}

public class ListTraverser
{
    public void TraverseList(List<object> list,
        IVisitor visitor)
    {
        foreach (object item in list)
        {
            if (item is int iItem)
                visitor.VisitInt(iItem);
            else if (item is double dItem)
                visitor.VisitDouble(dItem);
        }
    }
}

public class DisplayVisitor : IVisitor
{
    private int intSum = 0;
    private double doubleSum = 0;

    public void VisitInt(int value)
    {
        intSum += value;
    }

    public void VisitDouble(double value)
    {
        doubleSum += value;
    }

    public int IntSum => intSum;
    public double DoubleSum => doubleSum;
}

public static class Program
{
    public static void Main(string[] args)
    {
        List<object> data = new List<object>()
        {
            1, 2.0, 3, 4.0, 5, "Ala ma kota"
        };

        var traverser = new ListTraverser();
        var visitor = new DisplayVisitor();

        traverser.TraverseList(data, visitor);

        Console.WriteLine(
            $"Sum of ints: {visitor.IntSum}");
        Console.WriteLine(
            $"Sum of doubles: {visitor.DoubleSum}");

        Console.ReadKey();
    }
}
```

Przykład z życia

Wzorzec Odwiedzający obecny jest między innymi wśród dostarczonych wraz z .NET klas ułatwiających pracę z wyrażeniami. Skorzystałem z niego podczas implementowania biblioteki Spooksoft. VisualStateManager, o której pisałem w jednym z poprzednich artykułów (Programista 4/2022 (103)).

Udostępniam tam klasy `LambdaCondition` oraz `ChainedLambdaCondition`, które pozwalają na zdefiniowanie warunku wywołania jakiejś komendy w postaci wyrażenia lambda. Wzorca Odwiedzający używam tam do sprawdzenia, czy wyrażenie nie zawiera niewspieranych konstrukcji, oraz do wyłuskania wszystkich wyrażeń typu `ParameterExpression` oraz `MemberExpression`.

Listing 25. Przetwarzanie wyrażeń lambda przy pomocy wzorca Odwiedzający

```
private class Visitor : BaseMemberAccessVisitor
{
    private readonly ExpressionType[]
        availableExpressions = new[] {
        ExpressionType.Add,
        ExpressionType.AddChecked,
        ExpressionType.And,
        ExpressionType.AndAlso,
        ExpressionType.ArrayLength,
        ExpressionType.ArrayIndex,
        ExpressionType.Constant,
        ExpressionType.Convert,
        ExpressionType.Divide,
        ExpressionType.Equal,
        ExpressionType.ExclusiveOr,
        ExpressionType.GreaterThan,
        ExpressionType.GreaterThanOrEqual,
        ExpressionType.Lambda,
        ExpressionType.LeftShift,
        ExpressionType.LessThan,
        ExpressionType.LessThanOrEqual,
        ExpressionType.MemberAccess,
        ExpressionType.Modulo,
        ExpressionType.Multiply,
        ExpressionType.MultiplyChecked,
        ExpressionType.Negate,
        ExpressionType.UnaryPlus,
        ExpressionType.NegateChecked,
        ExpressionType.Not,
        ExpressionType.NotEqual,
        ExpressionType.Or,
        ExpressionType.OrElse,
```

```
            ExpressionType.Parameter,
            ExpressionType.Power,
            ExpressionType.Quote,
            ExpressionType.RightShift,
            ExpressionType.Subtract,
            ExpressionType.SubtractChecked,
            ExpressionType.TypeIs,
            ExpressionType.Default,
            ExpressionType.Unbox,
            ExpressionType.TypeEqual,
            ExpressionType.OnesComplement,
            ExpressionType.IsTrue,
            ExpressionType.IsFalse
        };

        private readonly List<Expression> expressions =
            new List<Expression>();

        public override Expression Visit(Expression node)
        {
            if (!availableExpressions.Contains(node.NodeType))
                throw new ArgumentException(
                    "Expression contains unsupported " +
                    $"operation: {node.NodeType}");

            return base.Visit(node);
        }

        protected override Expression VisitParameter(
            ParameterExpression node)
        {
            expressions.Add(node);
            return base.VisitParameter(node);
        }

        protected override Expression VisitMember(
            MemberExpression node)
        {
            expressions.Add(node);
            return base.VisitMember(node);
        }

        public override IReadOnlyList<Expression> Expressions
            => expressions;
    }
```

NA KONIEC

Jeżeli mam być szczery, obie części artykułu były dla mnie sporym wyzwaniem. Postawiłem sobie za punkt honoru nie tylko pokazać przypadki użycia wzorców projektowych z realnych, konkretnych projektów, ale podniosłem sobie jeszcze trochę poprzeczkę, ograniczając się na końcu tylko do projektów, które sam pisałem kiedyś w wolnym czasie, nie mając prawie żadnego pojęcia o wzorcach projektowych.

Tak jest, dobrze przeczytaliście. O wzorcach projektowych usłyszałem dawno temu, raz czy dwa rzuciłem na nie okiem, znam te najbardziej oczywiste (Singleton, Iterator), ale pozostałe nie były mi bliżej znane. Tematem zająłem się głębiej dopiero przy okazji uczestniczenia w rozmowach kwalifikacyjnych, bo jest to jedno z najpopularniejszych poruszanych tam zagadnień. Jak to jest więc możliwe, że udało mi się zaprezentować przykłady zastosowania każdego ze wzorców w moich projektach? Czy to jakaś forma wrodzonego geniuszu? Zbieg okoliczności? Szczęśliwy przypadek?

Nic z tych rzeczy.

Wyobraźmy sobie, że dla każdego możliwego rozwiązania jakiegoś problemu wprowadzamy punktację. Za każde złamanie zasady pojedynczej odpowiedzialności, powielenie kodu, wprowadzenie do niego „ifologii", nadmiernych zależności pomiędzy klasami – słowem, za każdy programistyczny grzech popełniony podczas jego implementowania przydzielamy punkty karne. Naturalnie zależy nam zawsze na tym, by podczas pisania programów punktacja naszych rozwiązań była w miarę możliwości minimalna. I gdy tak wędrujemy sobie po tej wielowymiarowej funkcji, co odnajdziemy w jej globalnym minimum? Oczywiście wzorzec projektowy.

Wzorce projektowe nie są magicznym panaceum na wszelkie zło, programistyczną biblią zawierającą dwadzieścia dwa nieprzekraczalne przykazania, ani też miernikiem tego, jak dobrym ktoś jest programistą. Tak naprawdę stanowią one wypadkową stosowania wszystkich dobrych praktyk programowania: z dużą dozą prawdopodobieństwa każde alternatywne do nich rozwiązanie będzie obarczone jakąś wadą. Oczywiście możemy te wady usuwać, ale w efekcie po prostu wrócimy do punktu wyjścia, czyli do naszego wzorca.

Długie lata programowania wykształciły we mnie dużo dobrych nawyków, a co najzabawniejsze, większość z nich wypracowałem, pisząc projekty prywatne, bo poświęcałem na nie mój cenny wolny czas i najzwyczajniej w świecie nie chciałem marnować go potem na poprawianie błędów lub robienie długich i skomplikowanych refaktoryzacji. I prawdopodobnie właśnie takie podejście sprawiło właśnie, że w moich programach zaczęły pojawiać się wzorce projektowe.

Czy więc warto je znać? Pewnie – szkoda czasu wyważać otwarte drzwi, jeśli wiemy, jakie rozwiązanie jest dla danego problemu najlepsze. Ale czy koniecznie trzeba je znać? Nie, uważam, że można być świetnym programistą, nie znając wcale wzorców projektowych. Dlatego też myślę, że na rozmowach kwalifikacyjnych zamiast oczekiwać od kandydata recytowania z pamięci, co oznacza dany wzorzec, więcej sensu ma postawienie mu jakiegoś standardowego problemu i sprawdzenie, czy udało mu się osiągnąć to metaforyczne globalne minimum (lub przynajmniej zbliżyć się do niego na zadowalającą odległość).

To powiedziawszy, mam nadzieję, że moje zestawienie pomoże przyswoić sobie zagadnienie wzorców projektowych, aby każdy następny napisany przez czytelnika projekt był choć trochę lepszy od poprzedniego.

W sieci

» Repozytorium kontrolki HexEditor dla WPF: https://gitlab.com/spook/hexeditor/
» Repozytorium aplikacji Board: https://gitlab.com/spook/Board

WOJCIECH SURA

wojciechsura@gmail.com

Programuje 30 lat, z czego 15 komercyjnie; ma na koncie aplikacje desktopowe, webowe, mobilne i wbudowane – pisane w C#, C++, Javie, Delphi, PHP, JavaScript i w jeszcze kilku innych językach. Obecnie pracuje w SII – największym w Polsce dostawcy usług doradztwa technologicznego, transformacji cyfrowej, Business Process Outsourcing i inżynierii.