Wzorce projektowe a praktyka

Każdy programista wcześniej czy później zderza się z tematem wzorców projektowych. Najczęściej widnieją one jako jedno z wymagań w ofertach pracy, czasem pojawiają się na rozmowach kwalifikacyjnych, a chyba jeszcze rzadziej rozmawia się o nich w czasie realizacji programistycznych projektów. Czym są? Czy warto zaprzątać sobie nimi głowę po zakończeniu procesu rekrutacyjnego? Czy nadają się tylko do akademickiej dysputy, czy też może da się zastosować je w prawdziwych scenariuszach? Spróbujmy zmierzyć się z tym ciekawym tematem.

ALE O CO KAMAN?

Wzorce projektowe stanowią tak naprawdę garść gotowych rozwiązań dla znanych, ogólnych problemów programistycznych. Tak samo jak architekt zapytany o możliwość przerzucenia mostu pomiędzy dwoma punktami, prawdopodobnie bardzo szybko zdecyduje się na most wolnopodparty, wspornikowy, łukowy, wantowy lub wiszący, tak obeznany ze wzorcami projektowymi programista skonfrontowany z koniecznością – powiedzmy – kontrolowanego powoływania do życia obiektów może zdecydować się na skorzystanie ze wzorca Fabryki abstrakcyjnej, Budowniczego lub Prototypu.

Tak zdefiniowane pojęcie wzorców projektowych może kreować wizję cudownego panaceum na każde zło, więc żeby zachować zdrowy umiar, czuję się w obowiązku dorzucić do tej beczki miodu łyżkę dziegciu.

Przede wszystkim wzorce projektowe istnieją tylko dla określonej grupy problemów. Oznacza to, że istnieją sytuacje, w których i tak konieczne okaże się zaprojektowanie rozwiązania skrojonego na miarę. Ale to nie wszystko: dotyczy to również i tych przypadków, w których problem wprawdzie pasuje do jednego ze wzorców, ale w grę wchodzi również dodatkowe ograniczenie, jak na przykład konieczność zminimalizowania rozmiaru pliku wykonywalnego, ilości zużytej pamięci lub przygotowania maksymalnie szybko działającego kodu. Również i w takiej sytuacji konieczne może okazać się zaprojektowanie niestandardowego rozwiązania.

Należy mieć też na uwadze, że chociaż znamy dokładnie wady, zalety i przypadki zastosowania wzorców projektowych, to nie oznacza to wcale, że są w każdej sytuacji najlepszym wyborem. Uważam, że do każdego problemu warto podchodzić z otwartym umysłem i nie odrzucać z marszu możliwości zaprojektowania czegoś zupełnie od nowa – o ile tylko uczciwie i starannie podejdziemy potem do procesu ewaluacji takiego rozwiązania. Historia niejednokrotnie pchana była do przodu przez ludzi, którzy najpierw wbrew publicznej opinii zabierali się za wynajdywanie koła na nowo, a potem kończyli z metaforycznym odpowiednikiem lewitującej deskorolki.

I jeszcze jedna istotna kwestia. O ile zdecydowanie uważam, że wzorce projektowe warto znać, to jednocześnie jestem zdania, że ich znajomość nie jest wcale wyznacznikiem dobrego programisty. W praktyce bowiem każdy ze wzorców jest często po prostu logiczną konsekwencją zaaplikowania do danego problemu bardziej ogólnych zasad, takich jak enkapsulacja, zwięzłość, separacja warstw, zasada pojedynczej odpowiedzi, SOLID i tak dalej.

W Internecie można oczywiście odnaleźć całą masę artykułów, a nawet serwisów poświęconych wzorcom projektowym. W każdym przypadku brakuje mi jednak zaprezentowania jakiegoś wziętego z życia przykładu zastosowania danego wzorca. Niniejszym artykułem postaram się więc tę lukę wypełnić, uciekając od akademickich przykładów opartych na pojazdach, samochodach, zwierzętach i kotach, a starając się w miarę możliwości pokazać realny przypadek użycia danego wzorca, który zaowocował rozwiązaniem jakiegoś konkretnego problemu.

RODZAJE WZORCÓW

Pierwotnie wzorce podzielone zostały na trzy podstawowe kategorie: wzorce kreacyjne, strukturalne i behawioralne. Z czasem pojawiło się więcej kategorii, jak na przykład wzorce wielowątkowe, ale zostawimy je sobie na później.

Wzorce kreacyjne dotyczą sytuacji, w których musimy powołać do życia nowy obiekt. Oczywiście samo w sobie nie jest to zbyt skomplikowane zadanie, mówimy tu jednak o podejściu nieco bardziej wysokopoziomowym, biorącym pod uwagę okoliczności, w których obiekt jest tworzony.

Wzorce strukturalne dotyczą sposobu interakcji pomiędzy różnymi klasami. Każda z nich pełni zwykle jakąś rolę, a cały ich zbiór może wykreować konkretny przypadek, w którym zasadne może okazać się zastosowanie określonego wzorca.

Wzorce behawioralne natomiast dotyczą zwykle tych sytuacji, w których konieczne jest oprogramowanie różnego rodzaju algorytmów, często powiązanych z określoną strukturą lub strukturami danych.

WZORCE KREACYJNE

Naszą podróż przez świat wzorców projektowych rozpoczniemy od wzorców kreacyjnych. Kanonicznie jest ich pięć: Metoda wytwórcza, Fabryka abstrakcyjna, Budowniczy, Prototyp oraz Singleton.

Metoda wytwórcza (Factory method)

Metoda wytwórcza to nic innego, jak abstrakcyjna metoda w klasie bazowej, której zadaniem jest powołanie do życia jakiegoś obiektu – „produktu". Podczas jej deklarowania ustalamy jednak tylko klasę bazową takiego produktu, pozwalając klasom pochodnym zwrócić instancje dziedziczące po tej klasie bazowej (Listing 1).

Listing 1. Prosta implementacja wzorca metody wytwórczej

```
public abstract class BaseProduct
{
    public override string ToString() => "Base product";
}

public abstract class BaseFactory
{
    public abstract BaseProduct CreateProduct();
}

public class Product : BaseProduct
{
    public override string ToString() => "Product";
}

public class Factory : BaseFactory
{
    public override BaseProduct CreateProduct()
    {
        return new Product();
    }
}

public static class Program
{
    public static void Main(string[] args)
    {
        BaseFactory factory = new Factory();
        BaseProduct product = factory.CreateProduct();
        Console.WriteLine(product);
        Console.ReadKey();
    }
}
```

Przykład z życia

Rok 2022 zapisze się w historii bodaj rekordowym wzrostem stóp procentowych. Przeszło dwukrotne zwiększenie rat kredytów skłoniło mnie do napisania aplikacji, która pomogłaby oszacować, jakie działania pomogłyby ograniczyć koszty związane ze spłacaniem kredytów. Na Rysunku 1 widać przykładową symulację, w której pieniądze oszczędzone dzięki (niesławnym) wakacjom kredytowym przeznaczone są na nadpłacenie drugiego kredytu.

Pierwsze dwie kolumny zawierają konfigurację kredytów oraz podejmowanych akcji. Zgodnie ze wzorcem MVVM konfiguracja przechowywana jest w viewmodelach poszczególnych wpisów. Przed uruchomieniem symulacji konieczne jest jednak przekształcenie ich w odpowiednie modele, które przekazywane są potem serwisowi zajmującemu się obliczeniami.

Proces generowania modeli zrealizowany jest właśnie przy pomocy wzorca metody wytwórczej. Bazowa klasa viewmodelu akcji finansowej wygląda następująco:

Listing 2. Bazowa klasa viewmodelu akcji finansowej (fragmenty)

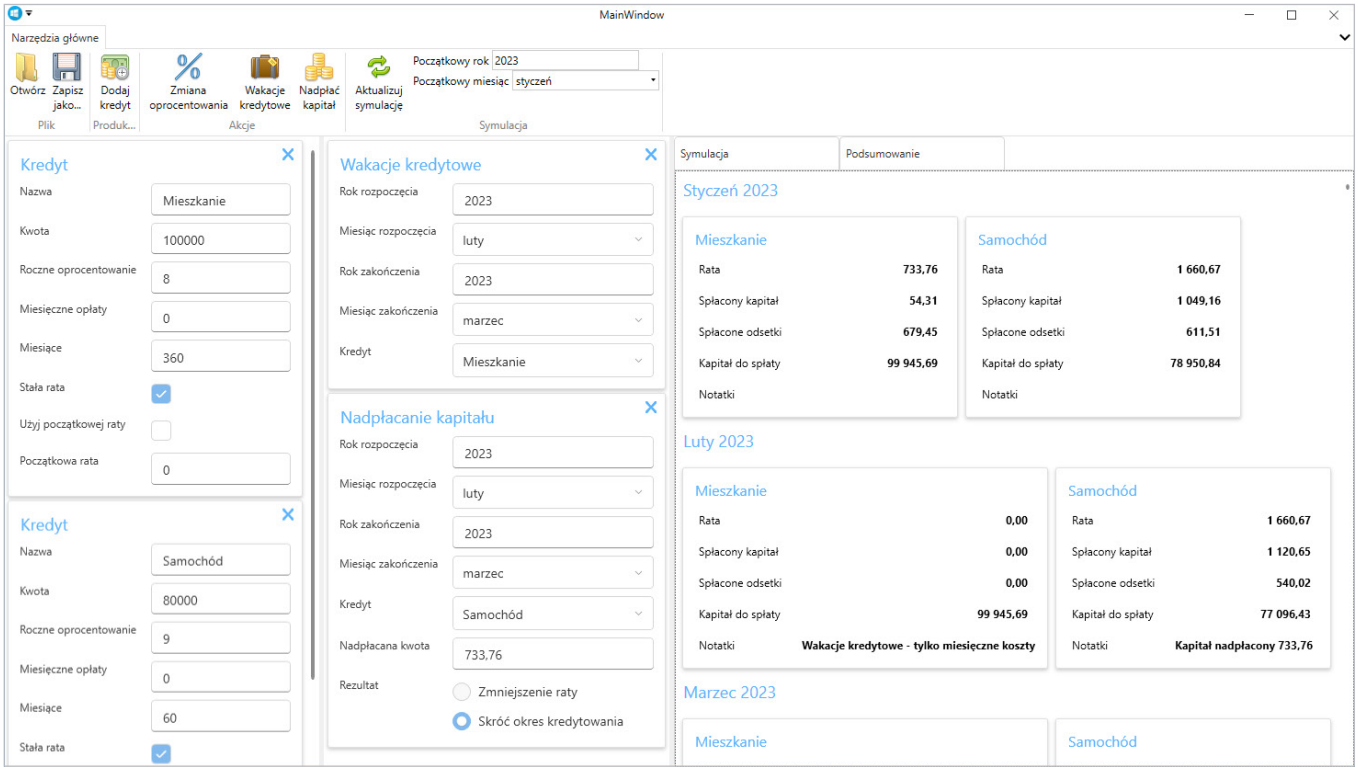
```
public abstract class BaseActionViewModel<T> :
    BaseValidateableViewModel<T>,
    IActionViewModel
    where T : BaseActionViewModel<T>
{
    protected int simulationStartYear;
    protected int simulationStartMonth;

    protected readonly IFinancialsProvider financialsProvider;
    protected readonly IActionViewModelHandler handler;

    // (...)

    public abstract BaseActionModel ToModel();

    // (...)
}
```



Rysunek 1. Symulator kredytów

W ten sposób viewmodel konkretnej akcji (na przykład nadpłacenia kapitału innego kredytu) może wyglądać następująco:

Listing 3. Klasa viewmodelu akcji nadpłacenia kredytu (fragmenty)

```
public class CapitalOverpaymentActionViewModel :
    BaseRecurringActionViewModel
        <CapitalOverpaymentActionViewModel>
{
    public const int ActionCode = 0;

    private double overpaymentAmount;
    private OverpaymentOutcome overpaymentOutcome;
    private IFinancialViewModel selectedLoan;

    // (...)

    public override BaseActionModel ToModel()
    {
        return new CapitalOverpaymentActionModel(
            selectedLoan.Id,
            StartYear,
            StartMonth.MonthId + 1,
            EndYear,
            EndMonth.MonthId + 1,
            overpaymentAmount,
            overpaymentOutcome);
    }

    // (...)
}
```

Co zyskujemy dzięki zastosowaniu wzorca Metody wytwórczej? Przede wszystkim przesuwamy decyzję dotyczącą stworzenia instancji konkretnej klasy do klasy pochodnej. Zrealizowanie tej funkcjonalności w klasie bazowej byłoby trochę dziwne (ogólnie klasa bazowa nie powinna wiedzieć nic o klasach pochodnych), a gdybyśmy z kolei przenieśli ten kod jeszcze wyżej – do viewmodelu głównego okna, skończylibyśmy z koniecznością napisania serii instrukcji warunkowych, które konstruowałyby odpowiednie obiekty, w zależności od tego, z jakim rodzajem viewmodelu akcji miałyby do czynienia.

Zastąpienie „ifologii" polimorfizmem ma jedną kluczową zaletę, o której warto pamiętać. Jeżeli dodamy nowy rodzaj akcji, ale zapomnimy zaimplementować kod generujący odpowiedni model, w pierwszym przypadku program się uruchomi i w najgorszym razie nie zgłosi nawet żadnego błędu (nowa akcja, nieobsługiwana przez odpowiedni warunek będzie po prostu ignorowana). W drugim przypadku jednak niezaimplementowanie metody `ToModel` skończy się błędem kompilacji: uruchomienie programu z takim błędem nie będzie w ogóle możliwe.

Fabryka abstrakcyjna (Abstract factory)

Wzorzec Fabryki abstrakcyjnej jest tak naprawdę rozwinięciem koncepcji Metody wytwórczej. Zamiast delegować do klasy pochodnej wytworzenie konkretnego, pojedynczego obiektu, delegujemy teraz wytworzenie całego zbioru potencjalnie zależnych od siebie obiektów. Ponieważ różnica z poprzednim wzorcem jest niewielka, ograniczę się tym razem tylko do zaprezentowania przykładowego kodu:

Listing 4. Przykładowa implementacja wzorca Fabryki abstrakcyjnej

```
public abstract class BaseProduct1
{
    public override string ToString()
        => "Base product 1";
}

public abstract class BaseProduct2
{
    public override string ToString()
        => "Base product 2";
}

public abstract class BaseFactory
{
    public abstract BaseProduct1 CreateProduct1();
    public abstract BaseProduct2 CreateProduct2();
}

public class Product1 : BaseProduct1
{
    public override string ToString()
        => "Product 1";
}

public class Product2 : BaseProduct2
{
    public override string ToString()
        => "Product 2";
}

public class Factory : BaseFactory
{
    public override BaseProduct1 CreateProduct1()
        => new Product1();

    public override BaseProduct2 CreateProduct2()
        => new Product2();
}

public static class Program
{
    public static void Main(string[] args)
    {
        BaseFactory factory = new Factory();
        BaseProduct1 product1 = factory.CreateProduct1();
        BaseProduct2 product2 = factory.CreateProduct2();
        Console.WriteLine($"{product1}, {product2}");
        Console.ReadKey();
    }
}
```

Budowniczy (Builder)

Wzorzec Budowniczego jest tak popularny, że w świecie .NET trudno się obrócić, by się na niego nie nadziać – bardzo często jest istotnym składnikiem składni określanej mianem „Fluent".

Załóżmy, że chcemy uruchomić jakiś proces, którego konfiguracja jest bardzo złożona. W najprostszym scenariuszu wszystkie konieczne dane możemy przekazać po prostu przy pomocy parametrów metody, ale szybko może okazać się, że jest ich tak dużo, iż konieczne jest kilkukrotne zawijanie linii, by w ogóle zobaczyć wszystkie na jednym ekranie.

Pierwszym pomysłem jest zwykle przygotowanie kilku wariantów wywoływanej metody, ale często i to nie pomaga w uporządkowaniu dziesiątków parametrów, a poza tym generuje często długą listę przeciążeń, które też nie wpływają pozytywnie na czytelność kodu. Naturalnym rozwiązaniem często staje się wtedy opakowanie parametrów w klasę i przekazywanie jej do metody – wtedy konfiguracja staje się nieco prostsza.

Ale to nie koniec problemów – czasem okazuje się, że parametry są od siebie zależne i rządzi nimi pewna logika. Na przykład jeżeli w jakimś scenariuszu dostarczymy datę początkową, konieczne może być dostarczenie również daty końcowej, która dodatkowo musi być późniejsza od tej pierwszej. Oczywiście przygotowaną konfigurację możemy zawsze zwalidować, ale dobra praktyka programistyczna każe przesuwać wystąpienia błędów możliwie jak najwcześniej – znacznie lepiej jest, by błąd wystąpił podczas kompilacji niż podczas działania programu.

Z pomocą przychodzi wtedy wzorzec Budowniczego. W jego ramach proces przygotowywania konfiguracji (lub, uogólniając, bu-