Zaawansowane Programowanie Internetowe

Łukasz Bartczuk

October 16, 2024

Zajęcia 1

Architektura Hexagonalna w C#

Na tych zajęciach zapoznacie się z przygotowaniem aplikacji zgodnie z architekturą hexagonalna.

> **Uwaga:**
>
> W przypadku jakichkolwiek pytań bądź problemów z odtworzeniem przykładów z niniejszego opracowania proszę o kontakt pod adresem: `lukasz.bartczuk@pcz.pl`. W mailu proszę opisać problem oraz przesłać swój kod do weryfikacji.

1.1 Wprowadzenie

Logika biznesowa aplikacji jest najważniejszą częścią tworzonego oprogramowania. Aplikacja nawet z najlepiej przygotowanym interfejsem użytkownika, która nie będzie spełniać wymagań biznesowych klienta, będzie bezużyteczna. Oczywiście, logika biznesowa musi współpracować z innymi elementami aplikacji, jak na przykład mechanizmami przechowywania danych lub interfejsem użytkownika w celu pobierania oraz zapisywania danych i prezentowania wyników. Jednak musimy pamiętać, że elementy te pełnią rolę (bardzo ważnych) dodatków dla logiki biznesowej.

Aby mieć pewność, że aplikacja spełnia wymagania klienta powinna być ona dobrze przetestowana. Cały (lub prawie cały) kod przygotowany przez nas powinien być sprawdzany za pomocą testów automatycznych jednostkowych, integracyjnych, testów UI, czy testów E2E. [1]. Aby to jednak było możliwe i odbyło się w miarę bezboleśnie aplikacja powinna być przygotowana w odpowiedni sposób.

Najczęściej aplikację dzieli się na mniejsze elementy, określane mianem komponentów[2], które powinny ze sobą współpracować. Przygotowanie takiego podziału nie jest jednak zadaniem trywialnym. Aby je uprościć zostały zaproponowane różne wzorce architektoniczne, których zadaniem jest określenie zasad organizacji kodu oraz współpracy między poszczególnymi jego częściami. Do najczęściej wykorzystywanych wzorców należą:

1. Architektura warstwowa,
2. Architektura hexadecymalna,

[1] Niestety temat testowania aplikacji wykracza poza ramy tego przedmiotu

[2] Pisząc komponenty mam na myśli fragmenty kodu tworzące logicznie spójną całość. Nie należy tutaj mylić tego pojęcia z komponentami interfejsu użytkownika budowanymi w Angular czy React.

3. CQRS

W dalszej części bliżej zapoznamy się z architekturą hexadecymalną (z małym udziałem CQRS) i sposobem jej implementacji w aplikacji C#.

1.2 Architektura hexagonalna

Architektura hexagonalna (zwana również Porty i Adaptery, Architekturą czystą czy Architekturą cebuli) jest sposobem organizacji kodu aplikacji, w centrum której znajduje się domena aplikacji, co jest przedstawione na rysunku 1.1.

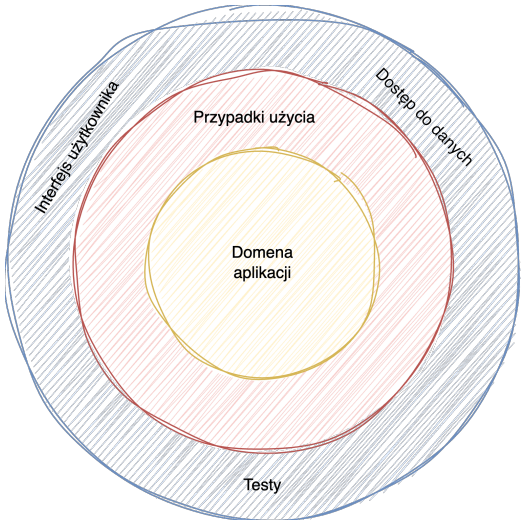

Figure 1.1: Architektura hexagonalna

Domena aplikacji jest to zestaw klas, które przede wszystkim definiują stan aplikacji (w postaci pól i właściwości) oraz metody, implementujące reguły modyfikujące ten stan. To w tej części powinna znaleźć się cała logika biznesowa.

Jeżeli przyjmiemy, że każda warstwa może zależeć, tylko od warstwy znajdującej się pod nią okaże się, że domena nie może zależeć od żadnego innego elementu aplikacji, natomiast wszystkie inne elementy zależą od niej. Oznacza to, że podczas programowania domeny, powinniśmy korzystać tylko i wyłącznie z czystych mechanizmów języka, bez żadnych dodatkowych bibliotek[3]. Warstwa ta będzie również zawierać usługi domeny, czyli fragmenty kodu, których celem jest realizowanie dodatkowych zdań na rzecz domeny (najczęściej pozwalają wpływać na jej stan), a które nie pasują do umieszczenia ich w innych klasach modelu domeny.

Bezpośrednio nad domeną znajduje się **warstwa przypadków użycia** (usług aplikacji), w której określamy działania jakie udostępnia nasza aplikacja. Te dwie warstwy stanowią rdzeń naszej aplikacji.

W zewnętrznej warstwie umieszczamy kod interfejsu użytkownika oraz kod dostępu do danych. Elementy te nazywane adapterami, komunikują się z rdzeniem aplikacji za pomocą portów. Te najczęściej są implementowane jako interfejsy (jeżeli język programowania wspiera tę konstrukcję programistyczną.)

[3]Moim zadaniem, należy zachować jednak pragmatyzm w programowaniu i pozwolić sobie na przygotowanie i korzystanie z biblioteki, która definiuje podstawowe typy danych, z których będziemy korzystać w różnych aplikacjach. Przykładem takich typów mogą być Option i Result znane wam z paradygmatów programowania.

Rysunek 1.2 przedstawia inny sposób reprezentacji architektury hexagonalnej, na którym jawnie zaznaczyłem porty.

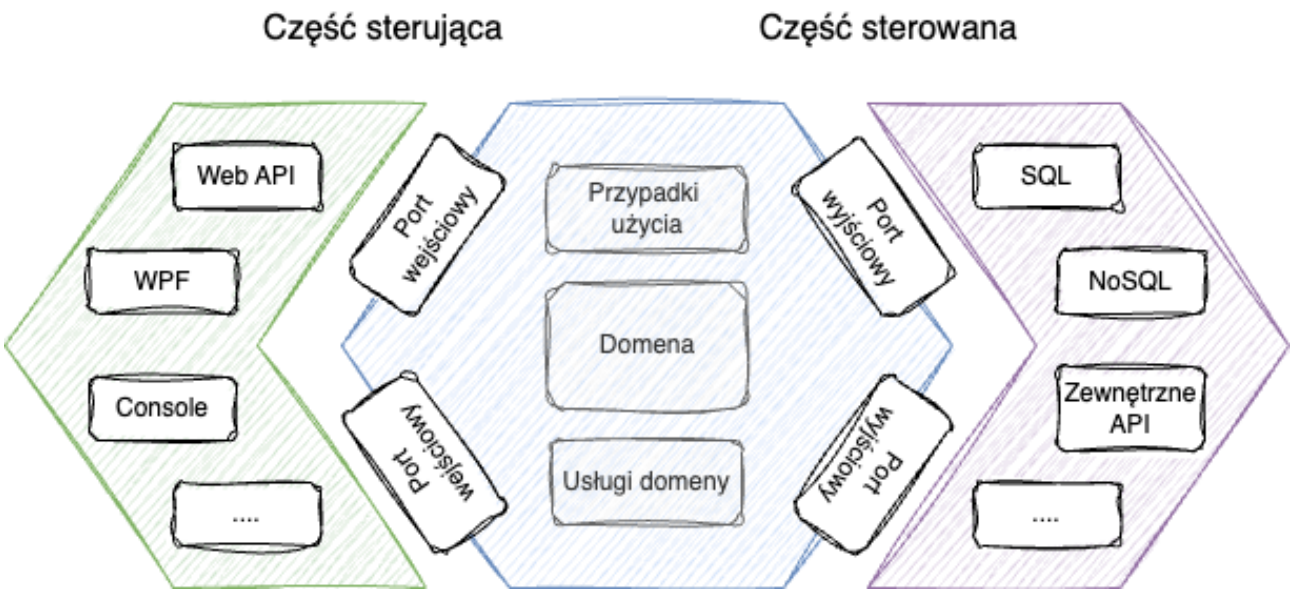

Figure 1.2: Architektura hexagonalna

Porty dzieli się na wejściowe i wyjściowe. Wejściowe znajdują się w części sterującej, czyli zawierającej kod, który uruchamia operacje w naszej aplikacji (kod ten steruje aplikacją). Najczęściej są to interfejsy, definiujące przypadki użycia.

Porty wyjściowe znajdują się w części sterowanej, czyli zawierającej kod uruchamiany przez domenę. Przykładowo mogą to być interfejsy repozytoriów umożliwiających wykonywanie operacji na bazie danych.

Dzięki takiemu przygotowaniu aplikacji uzyskujemy korzyści takie jak:

- izolacja rdzenia – rdzeń aplikacji (domena) jest niezależny od wykorzystanych technologii. Dzięki czemu możemy je bez większych problemów podmienić, jeżeli będzie taka potrzeba.
- rozszerzalność – Dzieki portom i adapterom bardzo łatwo zintegrować aplikację z nowymi systemami i technologiami.
- testowalność – domena aplikacji jest bardzo prosta do przetestowania (w miejsce konkretnych technologii możemy zastosować atrapy – mocki lub stuby.)

W dalszej części opracowania zobaczymy sposób implementacji tej architektury w języku C#.

1.3 Architektura hexagonalna w C#

W tej części stworzymy prosty projekt pokazujący przykładową implementację opisywanej architekturze w C#.

Będzie się on składał z czterech elementów:

1. Biblioteki DLL (`Application`) zawierającej rdzeń naszej aplikacji (typy definiujące domenę oraz poszczególne przypadki użycia),

2. Biblioteki DLL (`Data`) zawierającej kod adapterów wyjściowych dla operacji bazodanowych, współpracujący z bazą SQLite,

3. Aplikacji konsolowej (`ConsoleApp`) – umożliwiającej nam sprawdzenie w dniu dzisiejszym, czy podstawowe funkcje naszej aplikacji działają poprawnie,

4. Aplikacji ASP.NET Web API - docelowej aplikacji Web API, którą stworzymy na kolejnych zajęciach.

Działania nasze rozpoczynamy od stworzenia katalogu w którym umieścimy naszą aplikację (np. `Contacts`).

Następnie stwórzmy w nim szablon naszego projektu. W tym celu wydajmy następujące komendy:

Konsola

```
dotnet new sln --name Contacts
dotnet new classlib --output Application
dotnet new classlib --output Data
dotnet new console --output ConsoleApp
dotnet sln add Application Data ConsoleApp
dotnet add Data/Data.csproj reference Application/Application.csproj
dotnet add ConsoleApp/ConsoleApp.csproj reference Application/Application.csproj
dotnet add ConsoleApp/ConsoleApp.csproj reference Data/Data.csproj
```

Pierwsze polecenie tworzy plik rozwiązania platformy .NET dla naszej aplikacji. Następne dwa tworzą szablony wspomnianych wcześniej bibliotek DLL. Czwarte polecenie tworzy testową aplikację konsolową. Piąte dodaje wszystkie utworzone projekty do rozwiązania platformy .NET. Ostatnie trzy polecenia tworzą referencje pomiędzy projektami.

Po wydaniu powyższych poleceń w projektach typu class library utworzone zostaną pliki `Class1.cs`. Aby nam nie przeszkadzały, pliki te możemy usunąć z projektu.

1.3.1 Utworzenie modelu domeny

Kodowanie rozpoczniemy od rdzenia naszej aplikacji, czyli projektu `Application`. Przejdźmy do katalogu tego projektu i stworzmy w nim podkatalogi dla poszczególnych elementów projektu:

Konsola

```
cd Application
mkdir Domain Queries Repositories Services UseCases
```

Utworzone katalogi będą zawierały następujące elementy:

1. `Domain` - model domeny naszej aplikacji.

2. `Repositories` - porty wyjściowe aplikacji (interfejsy koniecznych repozytoriów).

3. `Services` - porty wejściowe aplikacji (interfejsy przypadków użycia (komend) naszej aplikacji).

4. `UseCases` - implementacje przypadków użycia – komend – naszej aplikacji.
5. `Queries` - implementacje zapytań wydawanych do danych w naszej aplikacji.

> **Uwaga:**
>
> Rozdzielając komendy i zapytania w naszej aplikacji postępujemy zgodnie z wzorcem **CQRS** (ang. *Command Query Responsibility Segregation*). **Komendy** mają na celu zmianę stanu naszej aplikacji, czyli w naszym przypadku np. dodawanie nowych kontaktów i modyfikowanie istniejących. Najczęściej nie zwracają one żadnych informacji (oprócz informacji o sposobie jej zakończenia). Z kolei zapytania zwracają dane w aplikacji bez modyfikowania stanu aplikacji. Rozdzielenie tych dwóch elementów pozwala na osobną pracę nad nimi i np. wdrażanie różnych technik optymalizacyjnych dla jednych i drugich. W naszej prostej aplikacji ograniczymy się do kodu na różne klasy, jednak w rzeczywistych implementacjach moglibyśmy zastosować różne aplikacje i różne typy baz danych.

Nasza aplikacja będzie stosunkowo prosta więc model również nie będzie szczególnie rozbudowany. Przedstawiony on jest na rys. 1.3

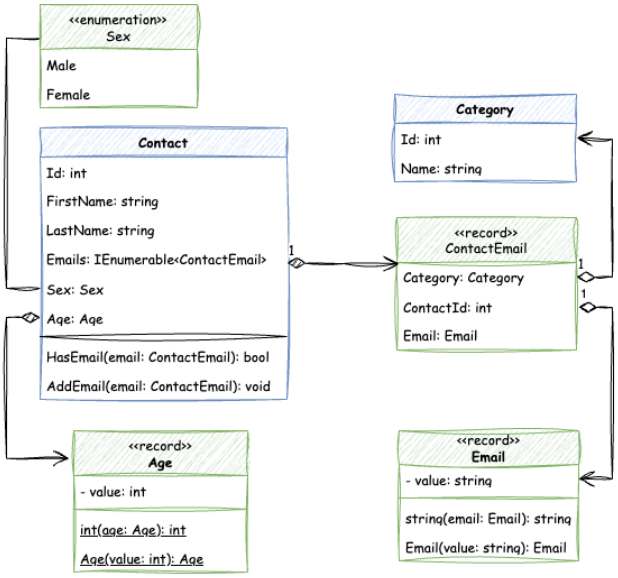

Figure 1.3: Diagram klas modelu domeny naszej aplikacji

Elementy pokazane na diagramie 1.3 możemy podzielić na dwie części (oznaczone na diagramie kolorami niebieskim i zielonym). Pierwsza to jednostki (ang. *Entities*). Są to obiekty, których tożsamość jest określona poprzez identyfikator (w naszym przypadku właściwość `Id`). Oznacza to, że dwa obiekty będą uznawane za takie same jeżeli będą miały taki sam identyfikator. Określają one obiekty, które mogą ewoluować w czasie (zmieniać wartości swoich właściwości).

Drugie to obiekty wartości (ang. `Value objects`). Ich celem jest opisanie pojedynczej wartości (często złożonej) oraz zdefiniowanie warunków jakie musi spełniać, aby mogłaby być uznana za poprawną. Obiekty te są niemutowalne, czyli nie mogą zmienić wartości swoich składowych. Możemy jedynie stworzyć nowy obiekt wartości i podmienić go z istniejącym. Ponadto są one porównywane strukturalnie tzn. dwa obiekty są traktowane za identyczne jeżeli wszystkie składowe tych obiektów są takie same.

Obiekty wartości najwygodniej zaimplementować jako rekordy C#. Dzięki temu nie będziemy musieli implementować równości strukturalnej pomiedzy obiektami (rekordy dają nam ją automatycznie). Jednostki implementuje się jako klasy. Powinniśmy w nich nadpisać metody `Equals` i `GetHashCode`, aby zapewnić ich równość identyfikatora [4].

Kod na listing-u 1.1 przedstawia przykładową implementację obiektu wartości `Age`.

1.1: Domain/Age.cs

```
namespace Contacts.Application.Domain;

public record Age {
    public int Value {get;}
    public Age(int value) {
        if(value < 18 && 120 < value)
            throw new ArgumentOutOfRangeException(nameof(value));
        Value = value;
    }

    public static implicit operator int(Age age)=>age.Value;
    public static implicit operator Age(int value) => new(value);
}
```

Definicja rekordu jest bardzo prosta zawiera jedną właściwość, konstruktor i dwa operatory rzutowania. Konstruktor sprawdza, czy ustawiany wiek ma wartość z określonego zakresu i jeżeli nie to rzucany jest odpowiedni wyjątek. Dzieki temu mamy pewność, że nasz obiekt jest zawsze w poprawnym stanie. Operatory rzutowania z i na wartość `int` ułatwią nam korzystanie z tego obiektu w innych częściach aplikacji. Dzieki nim (z punktu widzenia domeny naszej aplikacji) publiczną właściwość moglibyśmy zamienić na prywatne pole klasy. Nie będziemy tego robili ponieważ w naszej prostej aplikacji model domeny będzie jednocześnie modelem bazy danych musimy pogodzić się z pewnymi ustępstwami. Jedną z nich jest konieczność istnienia publicznych właściwości dzięki, którym Entity Framework będzie mógł stworzyć odpowiednie pola w tabeli bazy danych.

Typ Sex jest reprezentowany jako prosta enumeracja więc jego implementacja jest bardzo prosta i przedstawiona na listing-u 1.2.

1.2: Domain/Sex.cs

```
namespace Contacts.Application.Domain;

public enum Sex {Male,Female}
```

Kolejnym obiektem wartości występującym w naszej aplikacji jest `EmailAddress`. Przedstawiony on jest na listing-u 1.3. Jest to bardzo prosty rekord nie zawierający żadnej logiki [5]. Dzieki temu podejściu nie musimy w nim definiować właściwości oraz jawnego konstruktora. Dzieki zastosowanemu

[4] Najczęściej to zachowanie implementowane jest jako klasa bazowa np. `Entity`, która definiuje właściwość `Id` i nadpisuje wspomniane metody. My w naszej prostej aplikacji nie będziemy tego jednak robić.

[5] Dodanie jej będzie jednym z zadań do samodzielnego wykonania

tzw. konstruktorowi podstawowemu kompilator generuje odpowiednie właściwości na podstawie jego parametrów (nazwy właściwości są dokładnie takie jak nazwy parametrów. Z tego powodu nazwa parametru zapisana jest dużą literą – pomimo, że jest to złamanie konwencji nazewniczych C#).

1.3: Domain/EmailAddress.cs

```
namespace Contacts.Application.Domain;

public record EmailAddress(string Address) {

    public static implicit operator string(EmailAddress address)
        => address.Address;
    public static implicit operator EmailAddress(string text)
        => new (text);
}
```

W domenie mamy zdefiniowane dwie jednostki, które są reprezentowane jako klasy `Email`, `Contact` i `Category` przedstawionych na listing-ach 1.4 - 1.6.

1.4: Domain/Email.cs

```
namespace Contacts.Application.Domain;

public class Email
{
    public int Id {get; private set;}
    public EmailAddress Address {get; private set; } = null!;
    public Contact Contact {get; private set;} = null!;
    public int ContactId {get; private set;}
    public int CategoryId {get; private set;}
    public Category Category {get; private set;} = null!;

    private Email()
    {
    }

    public Email(int id, Contact contact, Category category, EmailAddress email)
    {
        Id = id;
        CategoryId = category.Id;
        ContactId = contact.Id;
        Contact = contact;
        Category = category;
        Address = email;
    }
}
```

1.5: Domain/Contact.cs

```
namespace Contacts.Application.Domain;

public class Contact {
    public int Id {get; private set;}
    public string FirstName {get; private set;}  = string.Empty;
    public string LastName {get; private set;} = string.Empty;
    public Sex Sex {get; private set;}
    public ICollection<Email> Emails {get; private set;} = null!;
    public Age Age {get; private set;} = null!;

    private Contact() {}

    public Contact(int id, string firstName, string lastName,
                Sex sex, ICollection<Email> emails, Age age) {
        Id = id;
        FirstName =
            string.IsNullOrWhiteSpace(firstName)
            ? throw new ArgumentException(nameof(firstName))
            : firstName;
        LastName =
            string.IsNullOrWhiteSpace(lastName)
            ? throw new ArgumentException(nameof(lastName))
            : lastName;
        Sex = sex;
        Emails = emails ?? throw new ArgumentNullException(nameof(emails));
        Age = age ?? throw new ArgumentNullException(nameof(age));
    }
        public bool HasEmail(Email email) {
            return Emails.Any(e => e.Address == email.Address);
        }

        public void AddEmail(Email email)
        {
            if (!HasEmail(email))
            {
                Emails.Add(email);
            }
        }
}
```

Zauważmy, że właściwości`Emails` i `Age` są sprawdzane tylko pod kątem pustej referencji. Jest to możliwe ponieważ obiekty tych typów tworzone są wcześniej i tam odpowiednie warunki są sprawdzane. Mamy więc pewność, że jeżeli dostaniemy odpowiedni obiekt, to na pewno będzie on poprawny.

Takiej pewności nie mamy jednak w przypadku właściwości `FirstName` i `LastName` i dlatego musimy sprawdzić czy użytkownik podał odpowiednią wartość [6].

[6] W naszym przypadku sprawdzamy tylko, czy użytkownik nie podał samych spacji, ale łatwo możemy wyobrazić sobie również inne warunki

Zwróćmy również uwagę na prywatny konstruktor domyślny, który nic nie robi. Z punktu widzenia domeny jego istnienie jest zupełnie nie potrzebne. Jednak ponieważ (ze względu na ograniczenie ilości koniecznego na napisania kodu) postanowiłem wykorzystać nasz model domeny również jako model danych Entity Framework, jest on niezbędny [7]

Poniżej mamy definicję jednostki `Category`, korzystającej z nowej składni konstruktora podstawowego:

1.6: Domain/Category.cs

```
namespace Contacts.Application.Domain;

public class Category(string name, int id)
{
    public Category(string name)
        : this(name, 0)
    {
    }

    public int Id {get; private set;} = id;
    public string Name {get; private set;} = name;
}
```

Uwaga:

Zwróćmy uwagę, że klasa `Category` nie zawiera prywatnego konstruktora domyślnego. Nie jest on w tym przypadku potrzebny ponieważ wszystkie parametry konstruktora podstawowego są typów wbudowanych w C#, więc EntityFramework nie ma problemu z jego wywołaniem.

1.3.2 Definicja portów wyjściowych

Jak wspomniałem wcześniej porty wyjściowe służą do komunikacji z zewnętrznymi źródłami danych (bazą danych, czy zewnętrznymi usługami). Ponieważ rdzeń naszej aplikacji nie powinien w żaden sposób zależeć od źródeł danych, wewnątrz niego zdefiniowane zostaną tylko interfejsy definiujące metody jakie będą potrzebne aplikacji, aby poprawnie wykonywać swoje działania. W naszym przypadku interfejsy te będą definiowały repozytoria dla jednostek `Category` i `Contact` oraz jednostkę pracy (ang. *Unit of Work*). Ten ostatni będzie nam służył do zatwiedzania zapisu zmian w bazie danych. Poszczególne interfejsy przedstawione są na listingach 1.7 - 1.9.

1.7: Repositories/ICategoryServices.cs

```
using Contacts.Application.Domain;

namespace Contacts.Application.Repositories;
```

[7] Jego brak zakończyłby się rzuceniem przez EntityFramework wyjątku, że biblioteka nie jest w stanie dokonać bindowania do parametrów konstruktora `emails` i `age`.

```
public interface ICategoryRepository
{
    IEnumerable<Category> GetCategories();
    Category? GetCategory(int categoryId);
    Category? GetCategoryByName(string categoryName);
    void Add(Category category);
    void RemoveCategory(Category category);
}
```

1.8: IUnitOfWork.cs

```
namespace Contacts.Application;

public interface IUnitOfWork
{
    void Save();
}
```

Ponieważ implementacja tych interfejsów jest zależna od wybranego sposobu dostępu do bazy danych nie należy ona do rdzenia aplikacji. Będzie ona dostępna jako adapter i zaimplementowana później w bibliotece DLL `Data`.

1.3.3 Definicja portów wejściowych

Porty wejściowe sterują aplikacją. Możemy je podzielić na dwie części: zapytania i komendy (co było już opisywane wcześniej). W obu przypadkach definicja będzie składała się z dwóch elementów: interfejsu, z którego będzie korzystał użytkownik oraz implementacji. Komendy definiują poszczególne przypadki użycia, których celem jest modyfikacja stanu aplikacji. Ich wejściem powinny być wszystkie informacje potrzebne do zrealizowania danej operacji. Komendy nie mają wyjścia (chyba, że wyjście określa sposób zakończenia komendy - poprawnie czy błędnie). Na listing-ach 1.10 - 1.12 mamy pokazane przykładowe implementacje przypadków użycia dla komend dotyczących kategorii.

Listing 1.10 przedstawia interfejs dla przypadków użycia dotyczących kategorii.

1.9: ICategoryUseCases.cs

```
using Contacts.Application.UseCases.DTOs;

namespace Contacts.Application.UseCases;

public interface ICategoryUseCases
{
    AddCategoryResult AddCategory(AddCategoryDTO addCategoryDTO);
    void RemoveCategory(int categoryId);
}
```

Zawiera on dwie proste metody. Pierwsza pozwala dodać nową kategorię do bazy danych, druga

pozwala ją usunąć.

Metoda `AddCategory`, przyjmuje – jako argument – obiekt typu `AddCategoryDTO`. Jest to tzw. Data Transfer Object. W tym przypadku zawiera on wszystkie informacje jakie są potrzebne do wykonania zadania, choć nie zawsze tak musi być. Jego definicja przedstawiona jest na listingu 1.11. Dodatkowo, aby ułatwić zamianę obiektów `AddCategoryDTO` na obiekty `Category` do klasy `AddCategoryDTO` zaimplementowałem niejawny operator konwersji.

1.10: UseCases/DTOs/AddCategoryDTO.cs

```
using Contacts.Application.Domain;

namespace Contacts.Application.UseCases.DTOs;

public class AddCategoryDTO
{
    public string Name { get; set; } = string.Empty;

    public static implicit operator Category(AddCategoryDTO dto)
        => new (dto.Name);
}
```

Metoda `AddCategory`, powinna być taktowana jak komenda – zmienia ona stan systemu – i jako taka nie powinna zwracać, żadnego wyniku (poza informacje czy zakończyła się poprawnie, czy nie). Ponieważ jednak będziemy korzystać z EntityFramework, a na następnych zajęciach również z ASP.NET WebAPI zrobimy małe odstępstwo od tej reguły i zwrócimy informacje o utworzonym obiekcie (który później będzie można przesłać do aplikacji uruchomionej w przeglądarce internetowej). Jednak nie zwracamy bezpośrednio obiektu domeny, tylko znów obiekt DTO. Jego definicja przedstawiona jest na listingu 1.11.

1.11: UseCases/DTOs/AddCategoryResult.cs

```
using Contacts.Application.Domain;

namespace Contacts.Application.UseCases.DTOs;

public class AddCategoryResult
{
    public int Id {get; init;}
    public string Name { get; init;} = string.Empty;

    public static implicit operator AddCategoryResult(Category category)
        => new AddCategoryResult{Id = category.Id, Name = category.Name};
}
```

Zwróćmy uwagę, że właściwości klasy `AddCategoryResult` są tylko do odczytu. Jest to spowodowane tym, że nie chcemy, aby można było zmodyfikować rezultat naszych działań po jego utworzeniu.

Implementacja interfejsu `ICategoryUseCases` przedstawiona jest na listingu 1.12. Klasa `CategoryServices` ma zdefiniowany konstruktor podstawowy przyjmujący dwa parametry. Pierwszym jest repozytorium, umożliwiające kontakt z źródłem danych przechowującym wymagane informacje, a drugim jednostka praca zatwierdzająca modyfikację danych i ewentualnie obsługująca transakcje (o ile będziemy z nich korzystać w naszej aplikacji).

1.12: Services/CategoryServices.cs

```
using Contacts.Application.UseCases;
using Contacts.Application.UseCases.DTOs;
using Contacts.Application.Repositories;

namespace Contacts.Application.Services;

public class CategoryServices(
    ICategoryRepository categoryRepository,
    IUnitOfWork unitOfWork

) : ICategoryUseCases
{
    public AddCategoryResult AddCategory(AddCategoryDTO categoryDTO)
    {
        if (categoryRepository.GetCategoryByName(categoryDTO.Name) == null)
        {
            var category = (Category)categoryDTO;
            categoryRepository.Add(category);
            unitOfWork.Save();
            return category;
        }
        else
        {
            throw new ServiceException("Kategoria o tej nazwie już istnieje");
        }
    }

    public void RemoveCategory(int categoryID)
    {
        var category = categoryRepository.GetCategory(categoryID);
        if (category != null)
        {
            categoryRepository.RemoveCategory(category);
            unitOfWork.Save();
        }
        else
        {
            throw new ServiceException(
                $"Kategoria o id: {categoryID} nie istnieje");
        }
    }
```

```
}
```

W przypadku wystąpienia błędu metody klasy `CategoryServices` rzucają wyjątek `ServiceException`. Wyjątek ten jest zaimplementowany jako bezpośrednie dziedziczenie po klasie `ApplicationException`, bez żadnej dodatkowej logiki. Dla porządku jego kod jest pokazany na listingu 1.13.

1.13: Services/ServiceException.cs

```
namespace Contacts.Application.Services;

public class ServiceException(string message)
    : ApplicationException(message)
{
}
```

To na razie kończy prace nad biblioteką `Application`. W następnym punkcie zobaczymy implementację wymaganych repozytoriów i jednostki pracy.

1.3.4 Implementacja repozytoriów i jednostki pracy

Dostęp do danych uzyskamy za pomocą znanej wam biblioteki Entity Framework Core. W tym celu do projektu `Data` konieczne będzie dołączenie kilku dodatkowych bibliotek [8]:

```
dotnet add Data package Microsoft.EntityFrameworkCore
dotnet add Data package Microsoft.EntityFrameworkCore.Design
dotnet add Data package Microsoft.EntityFrameworkCore.Sqlite
```

Pierwsza biblioteka zawiera kod biblioteki EntityFramework. Druga ułatwia wykonywanie zadań podczas tworzenia aplikacji np. przygotowuje kod dla migracji. Ostatnia stanowi adapter umożliwiający połączenie kontekstu z bazą danych SQLite.

> **Uwaga:**
>
> Na tych zajęciach będziemy korzystać z SQLite. Jest to bardzo prosta relacyjna baza danych, wykorzystywana głównie do celów deweloperskich.

Dodatkowo aby móc wykonywać polecenia EntityFramework z poziomu konsoli powinniśmy zainstalować rozszerzenie polecenia `dotnet` dla biblioteki `EntityFramework`:

```
dotnet tool install --global dotnet-ef
```

po zainstalowaniu powyższych bibliotek musimy przygotować klasę kontekstu, która będzie definiowała mapowanie relacyjno-obiektowe. W tym celu w katalogu `Data` stwórzmy plik `AppDbContext.cs` i umieśćmy w nim następujący kod:

[8] Polecenia te są przygotowane do wydania z konsoli z poziomu katalogu, w którym znajduje się plik `Contacts.sln`. Jeżeli będziecie wewnątrz katalogu projektu `Data`, można pominąć nazwę projektu. Polecenie domyślnie przeszukuje bieżący katalog w poszukiwaniu pliku .scproj

1.14: Data/AppDbContext.cs

```
using Microsoft.EntityFrameworkCore;
using Contacts.Application.Domain;

namespace Contacts.Data;

public class AppDbContext : DbContext {

    public DbSet<Contact> Contacts => Set<Contact>();
    public DbSet<Category> Categories => Set<Category>();
    public DbSet<Email> Emails => Set<Email>();

    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    {}

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Contact>(contactBuilder=>{
            contactBuilder
                .HasKey(contact=>contact.Id);
            contactBuilder
                .Property(contact => contact.FirstName)
                .IsRequired();
            contactBuilder
                .Property(contact=>contact.LastName)
                .IsRequired();
        });

        modelBuilder.Entity<Category>(categoryBuilder => {
            categoryBuilder
                .HasKey(category => category.Id);
            categoryBuilder
                .Property(category => category.Name)
                .IsRequired();
        });

        modelBuilder.Entity<Email>(emailBuilder => {
            emailBuilder
                .HasKey(email => email.Id);
        });

        modelBuilder.Entity<Email>().OwnsOne(email => email.Address);

        modelBuilder.Entity<Contact>()
            .HasMany(c => c.Emails)
            .WithOne(e => e.Contact)
            .HasForeignKey(e => e.ContactId);
```

```
        modelBuilder.Entity<Contact>().OwnsOne(
            contact => contact.Age,
            ageBuilder => ageBuilder.Property(age => age.Value)
        );
    }
}
```

Jak widzimy, dla każdej klasy definiującą jednostkę mamy w EntityFramework zdefiniowaną osobą właściwość typu `DBSet<>` (a co za tym idzie, osobną tabelę w bazie danych). Takich właściwości nie definiujemy dla obiektów wartości, gdyż te są zarządzane przez jednostki (co jest ustawione poprzez wywołanie metod `OwnsOne` podczas tworzenia modelu w metodzie `OnModelCreating`).

Następnie powinniśmy przygotować definicję portów wyjściowych, czyli repozytoria i jednostkę pracy. W katalogu `Data` stwórzmy kolejne pliki `CategoryRepository.cs`, `CategoryQueries.cs` oraz `UnitOfWork.cs`. Ich kod jest przedstawiony na listeningach 1.15 - 1.17. Ich kod jest bardzo prosty więc nie powinniście mieć problemów ze zrozumieniem go.

1.15: Data/CategoryRepository.cs

```
using Contacts.Application.Domain;
using Contacts.Application.Repositories;

namespace Contacts.Data;

public class CategoryRepository(AppDbContext context) : ICategoryRepository
{
    public void Add(Category category)
    {
        context.Categories.Add(category);
    }

    public IEnumerable<Category> GetCategories()
        => context.Categories;

    public Category? GetCategory(int categoryId)
        => context.Categories.Find(categoryId);

    public Category? GetCategoryByName(string categoryName)
        => context.Categories.SingleOrDefault(
            category => category.Name == categoryName);

    public void RemoveCategory(Category category)
        => context.Remove(category);
}
```

1.16: Data/CategoryQueries.cs

```
using Contacts.Application.Queries;
using Contacts.Application.Queries.DTOs;

namespace Contacts.Data;

public class CategoriesQueries(AppDbContext context) : ICategoryQueries
{
    public IEnumerable<CategoryDTO> GetCategories()
    {
        return context.Categories.ToList().Select(
            category => (CategoryDTO)category);
    }

    public CategoryDTO? GetCategory(int categoryId)
    {
        return context.Categories.Find(categoryId);
    }
}
```

1.17: Data/UnitOfWork.cs

```
using Contacts.Application;

namespace Contacts.Data;

public class UnitOfWork(AppDbContext context) : IUnitOfWork
{
    public void Save()
        => context.SaveChanges();
}
```

1.3.5 Przygotowanie aplikacji konsolowej

Musimy mieć możliwość sprawdzenia, czy to co przygotowaliśmy działa poprawnie. Powinniśmy to zrobić pisząc odpowiednie testy (jednostkowe i integracyjne). Niestety temat ten wykracza poza ramy tego przedmiotu. Na kolejnych zajęciach przygotujemy aplikację WebAPI, dziś jednak musimy poprzestać na prostej aplikacji konsolowej.

Zawartość pliku `Program.cs` jest w sumie bardzo prosta. Składa się z dwóch głównych części: konfiguracyjnej (wczytanie pliku konfiguracyjnego oraz skonfigurowanie połączenia z bazą danych) oraz właściwej aplikacji (utworzenie kontekstu i uruchomienie jednego z przypadków użycia). Kod tego pliku przedstawiony jest na listingu 1.18:

1.18: Program.cs

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Contacts.Application.Domain;
using Contacts.Application.Services;
using Contacts.Application.UseCases.DTOs;
using Contacts.Data;

// Wczytanie pliku konfiguracyjnego
var builder = new ConfigurationBuilder();
builder.SetBasePath(AppContext.BaseDirectory)
        .AddJsonFile(
            "appsettings.json",
            optional: false,
            reloadOnChange: true);

// Skonfigurowanie połączenia z bazą danych
IConfiguration config = builder.Build();
var dbOptionsBuilder = new DbContextOptionsBuilder<AppDbContext>();
dbOptionsBuilder.UseSqlite(config.GetConnectionString("default"));

// Stworzenie kontekstu
using var context = new AppDbContext(dbOptionsBuilder.Options);
context.Database.Migrate();
context.Database.EnsureCreated();

// Uruchomienie aplikacji
var categoryRepository = new CategoryRepository(context);
var unitOfWork = new UnitOfWork(context);
var categoryServices = new CategoryServices(categoryRepository, unitOfWork);

categoryServices.RemoveCategory(1);

foreach(var category in context.Categories)
{
    Console.WriteLine(category.Name);
}
```

W celu wykonania pierwszego kroku (wczytanie pliku konfiguracyjnego) musimy do aplikacji dołączyć bibliotekę, `Microsoft.Extensions.Configuration.json` która to umożliwi. Robimy to poleceniem:

```
dotnet add ConsoleApp package Microsoft.Extensions.Configuration.json
```

W kodzie tym wywołujemy metodę `Migrate`, która jest odpowiedzialna za wykonanie migracji w trakcie pierwszego uruchomienia aplikacji. Metoda `EnsureCreated` dodatkowo sprawdza, czy na pewno baza danych została utworzona.

W tej aplikacji wszystkie elementy musimy stworzyć ręcznie (na kolejnych zajęciach skorzystamy z mechanizmu DependencyInjection). Daje nam to możliwość zobaczenia faktycznych zależności pomiędzy

poszczególnymi obiektami.

Do uruchomienia aplikacji będziemy potrzebować pliku konfiguracyjnego, którego kod przedstawiony jest poniżej:

1.19: appsettings.json

```
{
    "ConnectionStrings": {
        "default": "Data Source=LocalDatabase.db"
    }
}
```

Najlepiej go utworzyć w katalogu `ConsoleApp`, gdzie znajduje się plik `Program.cs`. Jednak finalnie powinien on się znaleźć w katalogu, w którym będzie uruchamiana aplikacja (podczas debugowania aplikacji powinno to być np.: `/bin/Debug/net8.0`). Możemy go tam przekopiować ręcznie, jednak łatwo można zapomnieć o tym kroku podczas przygotowywania aplikacji. Lepszym rozwiązaniem jest zmusić środowisko (w naszym przypadku polecenie `dotnet`), aby automatycznie wykonało tę pracę za nas. W tym celu musimy zmodyfikować plik `ConsoleApp.csproj` dołączając tam wpis informujący, że `appsettings.json` ma być kopiowany do odpowiedniego folderu podczas budowania aplikacji o ile tam nie istnieje lub jego wersja jest nowsza:

1.20: ConsoleApp.csproj

```
<Project Sdk="Microsoft.NET.Sdk">

    <ItemGroup>
        <ProjectReference Include="..\Application\Application.csproj" />
        <ProjectReference Include="..\Data\Data.csproj" />
    </ItemGroup>

    <ItemGroup>
        <PackageReference Include="Microsoft.Extensions.Configuration.json"
            Version="8.0.0" />
    </ItemGroup>

    <ItemGroup>
        <None Update="appsettings.json">
            <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
        </None>
    </ItemGroup>

    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net8.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>
    </PropertyGroup>
```

```
</Project>
```

Zanim będziemy mogli uruchomić aplikację, musimy stworzyć migracje, aby na ich podstawie można było zbudować strukturę bazy danych. Aby to zrobić musimy zainstalować do projektu `ConsoleApp` bibliotekę `Microsoft.EntityFrameworkCore.Design` [9]:

```
dotnet add ConsoleApp package Microsoft.EntityFrameworkCore.Design
```

Robimy to za pomocą polecenia: [10]:

```
dotnet ef migrations add InitialCreate --project Data --startup-project ConsoleApp
```

Polecenie to najpierw sprawdzi, czy aplikacja się kompiluje, a następnie będzie starało się utworzyć migracje. Jednak nawet jeżeli udało nam się napisać aplikacje bezbłędnie polecenie to nie zakończy się poprawnie. Przyczyną tego jest barak możliwości stworzenia przez polecenie `dotnet ef` kontekstu bazy danych, który jest dla niego niezbędny do przygotowania migracji. Naprawa tej sytuacji polega na stworzeniu pliku np. o nazwie `ApplicationDbContextFactory.cs`, którego kod przedstawiony jest na listingu 1.21. Zawiera on klasę implementującą interfejs `IDesignTimeDbContextFactory<>`, który zawiera metodę `CreateDbContext`, której zadaniem jest utworzenie brakującego kontekstu [11].

1.21: ApplicationDbContextFactory.cs

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.Extensions.Configuration;
using Contacts.Data;

internal class ApplicationDbContextFactory
    : IDesignTimeDbContextFactory<AppDbContext>
{
    AppDbContext IDesignTimeDbContextFactory<AppDbContext>.CreateDbContext(
        string[] args)
    {
        IConfigurationRoot configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json")
            .Build();

        var builder = new DbContextOptionsBuilder<AppDbContext>();
        var connectionString = configuration
            .GetConnectionString("DefaultConnection");

        builder.UseSqlite(connectionString);

        return new AppDbContext(builder.Options);
```

[9] Bez dołączenia tej biblioteki aplikacja będzie się kompilowała, ale podczas tworzenia migracji pojawi się błąd

[10] Na zajęciach wszystkie polecenia kompilacji, uruchamiania aplikacji lub tworzenia migracji wydajemy w konsoli, w której bieżącym katalogiem jest katalog gdzie znajduje się plik `Contacts.sln`

[11] Krok ten potrzebny jest tylko w aplikacji konsolowej i w projekcie WebAPI nie będzie potrzebny

```
    }
}
```

Jeżeli teraz ponownie wydamy polecenie tworzenia migracji, powinno zakończyć się ono sukcesem. Możemy teraz wydać polecenie uruchomienia przygotowanej aplikacji:

```
dotnet run --project ConsoleApp
```

1.4 Zadania do samodzielnego wykonania

Zadanie 1.
Klasa `Contact` ma właściwości `FirstName` i `LastName` typu `string`. Zaproponuj dla nich odpowiednie obiekty wartości i przenieś do nich sprawdzanie warunków poprawności.

Zadanie 2.
W rekordzie `EmailAddress` brakuje sprawdzania poprawności formatu adresu email. Dodaj go. Wykorzystaj wyrażenia regularne.

Zadanie 3.
Rozbuduj aplikację o możliwość modyfikowania kategorii.

Zadanie 4.
Zmodyfikuj operację usuwania kategorii. Zastanów się nad warunkami jakie powinny być spełnione, aby ta operacja mogła zakończyć się poprawnie. Przyjmij założenie, że adres email musi mieć zawsze przypisaną kategorię.

Zadanie 5.
Rozbuduj aplikację o możliwości dodawania, edycji i usuwania nowego kontaktu.

Zadanie 6.
Rozbuduj aplikację o możliwości dołączania i usuwania adresu email z kontaktu.

Zadanie 7.
Rozbuduj aplikację o możliwości dołączania i usuwania numeru telefonu do kontaktu. Zastanów się, jakich obiektów wartości i jednostek będziesz do tego potrzebował. Zaimplementuj odpowiednie przypadki użycia.

Zadanie 8.
Rozbuduj aplikację o możliwości zapisywania informacji o ważnych datach związanych z kontaktem. Zastanów się, jakich obiektów wartości i jednostek będziesz do tego potrzebował. Zaimplementuj odpowiednie przypadki użycia.

Biblioteka

Zadanie 9.
Chciałbyś stworzyć system do obsługi biblioteki. Aplikacja powinna pozwalać na zarządzanie katalogiem książek, katalogiem czytelników, wypożyczeniami i rezerwacjami. Zastanów się jakie reguły biznesowe mogą sterować tymi operacjami.

Zadanie 10.
Zaproponuj jakie przypadki użycia mogłby być zdefiniowane dla tego projektu. Jak będą wyglądały obiekty DTO stanowiące ich dane wejściowe oraz ich rezultaty.

Zadanie 11.
Jakie jednostki kodu i jakie obiekty wartości powinny być w tym projekcie zastosowane.

Zadanie 12.
Zaimplementuj te projekt na podstawie rezultatów zadań 9–11.