Technologie Obiektowe – Projekt	
Temat 12: Obiektowe bazy danych	Kaja Wieczorek
	1ID22A

1. Cel i początek projektu

Celem projektu było zapoznanie się oraz stworzenie obiektowej bazy danych, a następnie porównanie jej z relacyjną bazą danych.

2. Obiektowa baza danych

Obiektowe systemy zarządzania bazą danych zapewniają tradycyjną funkcjonalność baz danych, lecz bazują na modelu obiektowym. Ich atutem jest udostępnianie danych w postaci obiektowej, czyli takiej samej w jakiej dane są przechowywane w programach napisanych w obiektowych językach programowania. Znika konieczność odwzorowania między modelem obiektowym a modelem relacyjnym jak to ma miejsce w przypadku użycia relacyjnej bazy danych.

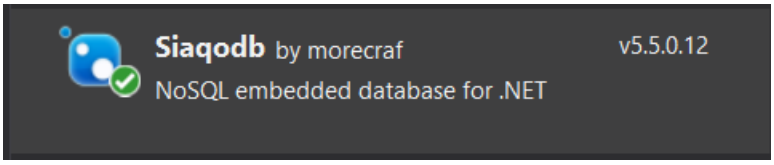
Obiekty są jednoznacznie identyfikowane za pomocą systemowego atrybutu nazywanego identyfikatorem obiektu lub w skrócie - OID. Wartości tego atrybutu są unikalne i niezmienne.

2.1. Wybór bazy danych

Do wykonania obiektowej bazy danych użyłam SiaqoDB. Jest to baza NoSQL dla obiektów i dokumentów, która jest obsługiwana przez .NET, MonoMac, Xamarin.iOS, Xamarin.Android, Xamarin.Mac i Unity3D. Może być synchronicowana z chmurą lub serwerem baz takich jak CouchDB, MongoDB lub Azure Storage.

2.2. Instalacja

Instalacja SiaqoDB jest bardzo prosta i polega na dodaniu do projektu pakietu „SiaqoDB” oraz dodać `using Sqo;`. Nie wymaga ona więcej konfiguracji.



2.3. Stworzenie bazy danych

Do stworzenia bazy danych, używana jest zmienna typu Siaqodb, na której w późniejszych etapach wykonywane są zapytania oraz inne operacje.

```
var db = new Siaqodb(dbPath);
```

Gdzie `dbPath` jest ścieżką do folderu, w którym chcemy utworzyć bazę danych.

2.4. Stworzenie obiektów i relacji

W obiektowej bazie danych obiekty tworzy się nie inaczej niż w zwykłym programie, zaczynając od stworzenia klasy. Musi ona posiadać pole OID – object id, którego nie przypisujemy. Jest ono automatycznie nadawane przy dodawaniu obiektu do bazy. Przykładowa klasa:

```
public class Home
{
    public int OID { get; set; }
    public List<Owner> Owners { get; set; }
    public List<Room> Rooms { get; set; }
    public Address Address { get; set; }
    public List<Occupant> Occupants { get; set; }
}
```

Aby stworzyć relację jeden do wielu, np. pomiędzy Domem a Właścicielami, należy w Domu dać listę typu Właściciel `List<Owner> Owners.` Podobnie jest, gdy chcemy, aby Dom miał jeden adres, wystarczy, aby miał pole typu Adres.

Tworzymy obiekt o pożądanych wartościach i dodajemy do bazy.

2.5. Operacje CRUD

- create – aby dodać obiekt do bazy, należy użyć metody StoreObject(objToAdd)

```
db.StoreObject(homeAddress);
```

- retrive – do pobrania danych z bazy służy metoda LoadAll<>

```
IObjectList<Address> clients = db.LoadAll<Address>();
```

- update – aby zmodyfikować obiekt, modyfikujemy pożądane pola obiektu w programie, a następnie wywołujemy StoreObject(), np.

```
homeAddress.Street = "Zmieniona";
homeAddress.ApartmentNumber = "900";
db.StoreObject(homeAddress);
```

- delete – usuwanie obiektu odbywa się poprzez metodę Delete(), a do usunięcia wszystkich obiektów danego typu DropType<>()

```
db.Delete(homeAddress);
db.DropType<Address>();
```

2.6. Zapytania

W programie zapytania można tworzyć na dwa sposoby. Poprzez kod SQL, np.

```
var query = from Address a in db select a;
```

lub przy wykorzystaniu LINQ:

```
var query = db.Query<Address>().Where(c => c.City == "Kielce");
```

W moim projekcie, zarówno dla obiektowej jak i relacyjnej użyłam właśnie LINQ, ponieważ są wygodniejsze w użyciu.

3. Relacyjna baza danych

Do projektu wybrałam bazę SQLite, ze względu na prostotę instalacji oraz użycia. Do mapowania obiektowo-relacyjnego użyłam Entity Framework Core, w skrócie EF.

3.1. Entity Framework Core

Modelowanie zaczyna się od klasy MyDbContext, dziedziczącej z DbContext. Tu tworzone są tabele z relacjami.

Pierwszym krokiem jest stworzenie tabeli. Używany jest tu DbSet<>

```
public DbSet<Home> Homes { get; set; }
```

Następnie w metodzie `OnModelCreating` są określane klucze główne oraz relacje.

```
modelBuilder.Entity<Home>().ToTable("Home", "DB");
modelBuilder.Entity<Home>(entity =>
{
    entity.HasKey(e => e.OID);
    entity.HasOne(e => e.Address);
    entity.HasMany(e => e.Owners);
    entity.HasMany(e => e.Rooms);
    entity.HasMany(e => e.Occupants);
});
```

`HasKey` – określa pole, które będzie kluczem głównym,

`HasOne` – określa pole z relacją jeden do jednego, tworzy klucz obcy,

`HasMany` – określa relację jeden do wielu, np. `entity.HasMany(e => e.Owners)` stworzy klucz obcy w tabeli `Owners` klucz obcy do `Home`.

3.2. Operacje CRUD

- create – aby dodać obiekt do bazy, należy użyć metody Add(objToAdd), lub użyta przeze mnie AddRange()

```
dbContext.People.AddRange(people);
```

- retrive – do pobrania danych z bazy wystarczy stworzyć DbSet<> i wybrać z dbContext pożądaną tabelę

```
var people = dbContext.People;
```

- update – aby zmodyfikować obiekt, modyfikujemy pożądane pola obiektu w programie, a następnie wywołujemy Update()

```
dbContext.Update(person);
```

- delete – usuwanie obiektu odbywa się poprzez metodę Remove()

```
dbContext.Remove(person);
```

Ważnym jest, aby po każdej zmianie dokonanej na obiekcie dbContext, wykonać polecenie `dbContext.SaveChanges();`

Bez niego zmiany nie zostaną zapisane do bazy.

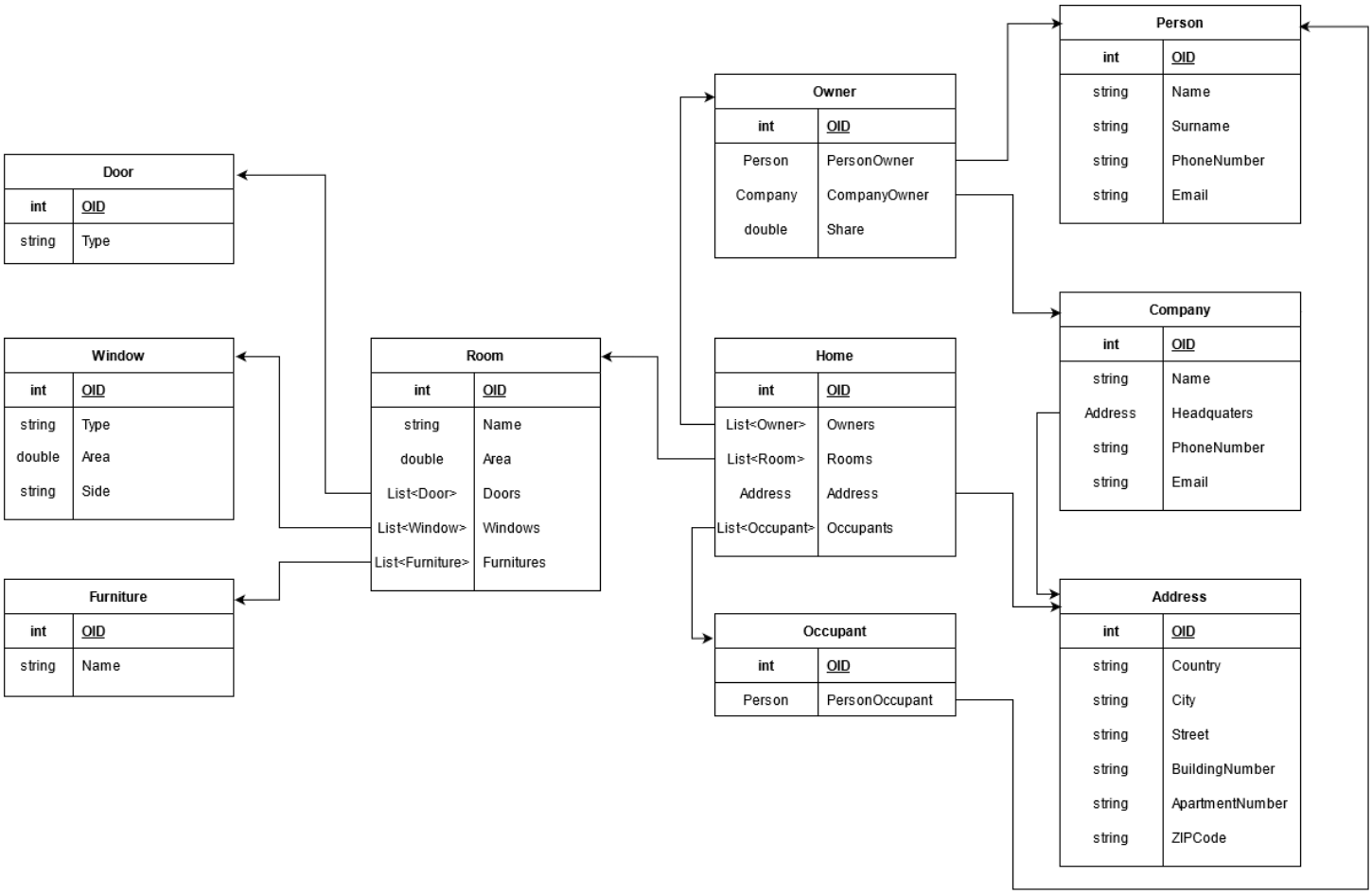
2.6. Zapytania

```
var query = db.Address.Where(c => c.City == "Kielce");
```

Jak wspomniano wcześniej, zapytania do bazy poprzez Entity Framework Core realizowane są poprzez LINQ. Przez program są one tłumaczone na zapytanie SQL.

4. Realizacja projektu

Do pokazania różnicy w typach baz danych, stworzyłam bazę z wieloma kluczami obcymi, prezentowaną na poniższym schemacie.



Schemat przedstawia stworzone klasy. Tak samo wyglądają one dla SiaqoDB jak w EF dla SQLite.

4.1. Generowanie danych

Najdłuższą częścią realizacji projektu było napisanie generatora danych tak, aby wygenerowane dane miały sens. Przykładowa metoda generująca pokój:

```
1 reference
public Room RandomRoom(Home home)
{

    var room = new Room();
    var random = new Random();
    var types = new List<string> { "kuchnia", "łazienka", "toaleta", "salon", "sypialnia", "gościnny", "garderoba", "spiżarnia", "pralnia", "garaż" };
    var mustHave = new List<string> { "kuchnia", "łazienka", "salon", "sypialnia" };
    int rooms = 0;
    if (home.Rooms != null)
    {
        rooms = home.Rooms.ToList().Count();
    }
    foreach (var type in types.ToList())
    {
        types = CheckRoomToRemove(types, type, home, 2);
    }
    if (rooms <= mustHave.Count())
        foreach (var type in mustHave.ToList())
        {
            types = CheckRoomToAdd(mustHave, type, home);
        }

    room.Name = types[random.Next(0, types.Count() - 1)];
    room.Area = random.Next(5, 25);
    room.Doors = new List<Door>() { RandomDoor() };
    if (random.Next(0, 2) == 0) { room.Doors.Add(RandomDoor()); }
    room.Windows = new List<Window>();
    room.Windows.AddRange(RandomWindows(room.Area));
    room.Furniture = new List<Furniture>();
    room.Furniture.AddRange(RandomFurnitures(room));
    return room;
}
```

Zapewnia ona, że w każdym domu będzie przynajmniej jedna kuchnia, łazienka, salon i sypialnia oraz że nie będzie więcej niż 2 pokoje tego samego typu. Zwraca obiekt pokoju już ze wszystkimi oknami, drzwiami i meblami.

Innymi założeniami to m. in. powierzchnia okien stanowi 1/9 powierzchni pokoju, meble są zależne od typu pokoju. Właścicielem domu może być zarówno firma jak i osoba prywatna, z losowymi udziałami własności. Dodatkowe założenie to to, że w domu mogą mieszkać od jednej do trzech osób, a jedna osoba nie może zamieszkiwać więcej niż trzy domy. Dla ułatwienia, wszystkie adresy znajdują się w różnych miastach Stanów Zjednoczonych.

4.2. Załadowane dane

Do obu baz zostało załadowane po 100 osób, 100 domów, 64 firmy, a reszta danych jest w losowych ilościach. Od 400 do 1800 pokoi, posiadających jedne lub dwie pary drzwi, z różną powierzchnią okien wychodzących na różne strony.

4.3. Porównanie baz

Dla porównania baz, wykonałam zapytania o coraz większej ilości złączeń i zestawiłam ze sobą czasy wykonania. Zapytania w LINQ wyglądają tak samo, za wyjątkiem początku pobrania danych. dla relacyjnej: `dbContext.Homes,` a dla obiektowej: `db.LoadAll<Home>().`

Każde zapytanie wykonane zostało po 10 razy.

Baza	Obiektowa	Relacyjna

Zapytanie	`.Where(x => x.Rooms`
	`.Count(r => r.Name == "sypialnia") > 1)`
	`.ToList();`	

Czas[ms] pierwsze	129	903
Czas[ms] średnia	130	95

Baza	Obiektowa	Relacyjna

Zapytanie	`.Where(x => x.Occupants`
	`.Any(x => x.PersonOccupant.Name == "James")`
	`&& x.Rooms.Count(r => r.Name == "sypialnia") > 1)`
	`.ToList();`	

Czas[ms] pierwsze	134	1276
Czas[ms] średnia	145	129

Baza	Obiektowa	Relacyjna

Zapytanie	`.Where(x => x.Occupants`
	`.Any(x => x.PersonOccupant.Name == "James")`
	`&& x.Rooms.Count(r => r.Name == "sypialnia") > 1`
	`&& x.Rooms.Any(f => f.Furniture.Count() > 3`
	`&& f.Name == "sypialnia"))`
	`.ToList();`	

Czas[ms] pierwsze	59	1031
Czas[ms] średnia	65	109

Baza	Obiektowa	Relacyjna

Zapytanie	`.Where(x => x.Owners`
	`.Any(o => o.CompanyOwner.Headquaters.City.Contains("Lake"))`
	`&& x.Rooms.Any(w => w.Windows.Any(s => s.Side == "wschód")`
	`&& x.Occupants.Count() >= 2)).ToList();`	

Czas[ms] pierwsze	51	1210
Czas[ms] średnia	56	126

Baza	Obiektowa	Relacyjna

Zapytanie	`.Where(x => x.Owners`
	`.Any(o => o.CompanyOwner.Headquaters.City.Contains("Lake")`
	`&& o.Share > 30)`
	`&& x.Rooms.Any(w => w.Windows.Any(s => s.Side == "wschód")`
	`&& w.Furniture.Any(t => t.Name == "komoda")`
	`&& x.Occupants.Count() >= 2)`
	`&& x.Address.City.Contains("Lake"))`
	`.ToList();`	

Czas[ms] pierwsze	60	1016
Czas[ms] średnia	61	108

5. Podsumowanie i wnioski

Dzięki realizacji projektu zaznajomiłam się z tematem obiektowych baz danych. Stosuje się je głównie do schematów z dużą ilością połączeń, kluczy obcych między tabelami, gdzie w relacyjnej bazie danych najbardziej kosztowne są złączenia JOIN.

Pierwszy raz miałam również styczność z mapowaniem obiektowo-relacyjnym z pomocą Entity Framework Core i jest to niesamowicie pomocne i wygodne narzędzie do pracy z relacyjną bazą danych.

Z przeprowadzonych testów jasno widać, że im więcej złączeń, tym obiektowa baza danych szybciej realizuje zapytania, kiedy to relacyjna zwalnia. Na przedstawione wyniki ma również wpływ cachowanie danych, dlatego dla Entity Framework widać dużą różnicę w czasie pierwszego zapytania, a średniej z czasów wykonania zapytania 10 razy.