Listes polymorphes et jointures dans Siaqodb

Introduction

Siaqodb est un logiciel de gestion d'objets persistants (base de données d'objets) simple à mettre en œuvre, destiné à gérer des ensembles de taille moyenne d'objets. Siaqodb est destiné à de nombres plateformes utilisant le langage C#. Il est donc principalement utilisé sur la plateforme .NET de Microsoft. Siaqodb supporte le langage de manipulation de données Linq intégré dans C#.

Ce document étudie un aspect de la modélisation par objets, à savoir les listes polymorphes, et un aspect des bases de données issu du modèle relationnel, à savoir les jointures, dans le but de faciliter le développement d'applications avec Siaqodb.

La version de Siaqodb utilisée est la version 5.5.0.8.

Ces différents aspects sont étudiés au travers d'un exemple de gestion d'employés, de clients et de sociétés. Le schéma de la base de données correspondante est formé par un ensemble de classes C# dont les codes sources sont fournis ci-dessous.

Dans les classes **Customer** et **Employee**, les données relatives à la ville sont dupliquées dans le but de montrer les différentes techniques utilisées. On trouve donc l'information dans les attributs **City** et **Country** et une seconde fois dans le lien **CityRef**. Il convient de noter qu'un lien inverse existe entre les instances de la classe **City** et les instances de **Customer**, d'une part, et entre les instances de la classe **City** et les instances de **Employee**, d'autre part.

La classe **HitechCompany** est une sous-classe de **Company** dont nous verrons l'utilité par la suite. La classe **CompanyList** est un singleton destiné à contenir la propriété **TheCompanyList** qui est une liste dont les éléments sont typés par **Company**.

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Sqo.Attributes;

namespace TestProfilingSiaqodb
{
    public class Customer
    {
        public int OID { get; set; }
        public string LastName { get; set; }
        public string FirstName { get; set; }
        public int Age { get; set; }
        public string City { get; set; }
        public string Country { get; set; }
        public City CityRef { get; set; }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Sqo.Attributes;
```

```
namespace TestProfilingSiaqodb
{
    public class Employee
    {
        public int OID { get; set; }
        public Company Employer { get; set; }
        public string LastName { get; set; }
        public string FirstName { get; set; }
        public int Age { get; set; }
        public DateTime HireDate { get; set; }
        public string City { get; set; }
        public string Country { get; set; }
        public City CityRef { get; set; }
    }

}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Sqo.Attributes;

namespace TestProfilingSiaqodb
{
    public class City
    {
        public int OID { get; set; }
        public string Name { get; set; }
        public string Country { get; set; }
         public List<Employee> Employees { get => employees; set => employees = value; }
        protected List<Employee> employees = new List<Employee>();
         public List<Customer> Customers { get => customers; set => customers = value; }
        protected List<Customer> customers = new List<Customer>();

    }
}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestProfilingSiaqodb
{
    public class Company
    {
        public int OID { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }
        public string Phone { get; set; }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestProfilingSiaqodb
{
    class HitechCompany : Company
    {
        public string Domain { get; set; }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace TestProfilingSiaqodb
{
    class CompanyList
    {
        public int OID { get; set; }
        public List<Company> TheCompanyList { get => theCompanyList;
                                               set => theCompanyList = value; }
        protected List<Company> theCompanyList = new List<Company>();
    }
}
```

Le peuplement de la base de données de test s'effectue par le code suivant. Il crée 100 instances de **City**, 50 instances directes de **Company** et 50 instances de **HitechCompany**. Il crée ensuite 90 instances de **Customer** (pour j variant de 0 à 89) et 90 instances de **Employee** (pour j variant de 10 à 99). Chaque instance de **Customer** et de **Employee** est liée à une ville (instance de **City**) et le lien inverse est construit. Chaque instance de **Employee** est liée à la société (instance directe de **Company** ou instance de **HitechCompany**) qui a été créée après la création de la ville.

Il convient enfin de noter la création d'une instance de **CompanyList** qui gère la liste de toutes les sociétés créées, quelle que soit leur classe d'instanciation.

Cette liste est également stockée dans la base de données (dernière ligne du code qui suit).

```
            Siaqodb siaqodb = new Siaqodb(@"c:\SiaqodbTest");

            siaqodb.DropType<Customer>();
            siaqodb.DropType<Employee>();
            siaqodb.DropType<Company>();
            siaqodb.DropType<HitechCompany>();
            siaqodb.DropType<CompanyList>();
            siaqodb.DropType<City>();

            CompanyList myCompanyList = new CompanyList();

            for (int j = 0; j < 100; j++)
            {
                City paris = new City();
                paris.Name = "Paris" + j;
                paris.Country = "France" + j;
                Company company;
                if (j % 2 == 0)
                    company = new Company();
                else
                {
                    HitechCompany hitechCompany = new HitechCompany();
                    hitechCompany.Domain = "Information Technology" + j;
                    company = hitechCompany;
                }
                company.Name = "The Company" + j;
                siaqodb.StoreObject(company);
                myCompanyList.TheCompanyList.Add(company);
                if (j >= 10)
                {
                    Employee employee1 = new Employee();
                    employee1.Employer = company;
                    employee1.FirstName = "Olivier" + j;
                    employee1.LastName = "Dupond" + j;
                    employee1.City = "Paris" + j;
                    employee1.Country = "France" + j;
                    employee1.CityRef = paris;
                    paris.Employees.Add(employee1);
                    siaqodb.StoreObject(employee1);
                }
                if (j < 90)
                {
```

```
                Customer customer1 = new Customer();
                customer1.FirstName = "Jean" + j;
                customer1.LastName = "Martin" + j;
                customer1.City = "Paris" + j;
                customer1.Country = "France" + j;
                customer1.CityRef = paris;
                paris.Customers.Add(customer1);
                siaqodb.StoreObject(customer1);
                siaqodb.StoreObject(paris);
            }
        }

        siaqodb.StoreObject(myCompanyList);
```

Gestion des listes polymorphes

Une requête Linq du type « from…in…select… » ne se base que sur les instances **directes** de la classe qui suit le from et non pas l'ensemble de ses instances, à savoir ses instances directes et les instances de ses sous-classes. Il ne s'agit donc pas d'une liste polymorphe.

Nous présentons deux possibilités pour obtenir l'ensemble des instances :

- Soit en utilisant l'opérateur Union de Linq pour fusionner l'ensemble des instances directes avec l'ensemble des instances directes de la sous-classe. (Nous faisons ici l'hypothèse que la classe n'a qu'une seule sous-classe, le problème général est plus complexe car sa solution consiste à parcourir l'arbre des sous-classes).

- Soit en gérant l'ensemble de toutes les instances de la classe dans un autre objet. Cet autre objet pourrait, par exemple, être référencé par une instance de classe. Il pourrait également être l'unique instance d'une classe implantant le pattern Singleton.

Dans notre exemple, la première possibilité est réalisée par le code suivant :

```
        var query0 = ((from Company c in siaqodb
             select new { Name = c.Name }).
             Union(from HitechCompany c in siaqodb
             select new { Name = c.Name })).ToList();

        foreach (var o in query0)
        {
            Console.WriteLine(o.Name);
        }
```

Le résultat de la requête tel qu'il est affiché est illustré à la suite. Les instances directes de **Company** apparaissent bien avant les instances de sa sous-classe de **HitechCompany**.

```
The Company0
The Company2
The Company4
The Company6
The Company8
The Company10
…
The Company90
The Company92
The Company94
The Company96
The Company98
The Company1
The Company3
The Company5
```

```
The Company7
The Company9
…
The Company91
The Company93
The Company95
The Company97
The Company99
```

Quant à la seconde possibilité, son étude commence par celle du code de peuplement de la base où après chaque création et initialisation d'une société, la ligne suivante est exécutée :

```
                myCompanyList.TheCompanyList.Add(company);
```

L'attribut **TheCompanyList** est typé par **List<Company>**. C'est une liste polymorphe. L'objet référencé par **myCompanyList** a été sauvegardé dans la base de données, il peut donc être restauré. Ensuite, il est possible d'afficher toutes les instances qu'il contient comme illustré dans le code suivant (à noter que nous l'avons fait en utilisant Linq mais pour nous aurions pu appliquer directement une boucle foreach sur cette list).

```
            List<Company> companyList = (from CompanyList cl in siaqodb
             select cl.TheCompanyList).ToList()[0];
            var query01 = (from Company c in companyList
             select new { Name = c.Name }).ToList();

            foreach (var o in query01)
            {
                Console.WriteLine(o.Name);
            }
```

Le résultat de la requête tel qu'il est affiché est illustré à la suite. L'ordre de l'affichage correspond bien à l'ordre dans lequel les instances (directes ou non) de la classe **Company** ont été insérées dans l'objet **myCompanyList** lors du peuplement de la base de données.

```
The Company0
The Company1
The Company2
The Company3
The Company4
The Company5
The Company6
The Company7
The Company8
The Company9
The Company10
…
The Company90
The Company91
The Company92
The Company93
The Company94
The Company95
The Company96
The Company97
The Company98
The Company99
```

Il est intéressant d'observer ici que Siaqodb gère parfaitement des listes polymorphes quand elles sont constituées sous la forme d'un attribut de type

List<T>. Cette possibilité de Siaqodb est mise en œuvre ici pour implanter et gérer la liste de toutes les instances de la classe **Company**.

Jointures classiques

Nous nous intéressons maintenant à une technique issue des bases de données relationnelles : les jointures. Dans un premier temps, nous nous limitons aux jointures classiques, celles qu'il est possible de définir dans la clause **where** du langage SQL.

Reprenons notre exemple ; nous cherchons toutes les couples employé/client qui habitent dans la même ville. Nous voulons retrouver les quatre informations suivantes :

- Le nom de la ville ;
- Le nom du pays ;
- Le nom de l'employé ;
- Le nom du client.

Siaqodb implante les jointures classiques (à noter par ailleurs que le mot-clé **join** ne semble pas être implanté dans la version testée de Siaqodb). Par conséquent, nous pouvons écrire le code de la requête tel qu'il est décrit à la suite. Il convient de remarquer que nous faisons le test d'égalité entre deux villes non seulement en comparant leur nom mais également en comparant le nom de leur pays. En effet, il existe des villes de même nom dans des pays différents.

```
            var query3 = (from Customer c in siaqodb
                          from Employee emp in siaqodb
                          where (c.City == emp.City) && (c.Country == emp.Country)

                          select new { CityName = c.City, CountryName = c.Country,
                                CName = c.LastName, EName = emp.LastName }).ToList();}
            foreach (var o in query3)
            {
                    Console.WriteLine(o.CityName + " " + o.CountryName + " " + o.CName
                                      + " " + o.EName);
            }
```

Le résultat affiché est inclus ci-dessous. Il commence avec la ville de Paris10 et se termine avec celle de Paris89. En effet, il n'existe pas d'employé habitant dans les villes de Paris0 à Paris9 et il n'existe pas de client habitant dans les villes de Paris90 à Paris99. Il s'agit donc bien d'une jointure classique.

```
Paris10 France10 Martin10 Dupond10
Paris11 France11 Martin11 Dupond11
Paris12 France12 Martin12 Dupond12
Paris13 France13 Martin13 Dupond13
Paris14 France14 Martin14 Dupond14
Paris15 France15 Martin15 Dupond15
Paris16 France16 Martin16 Dupond16
Paris17 France17 Martin17 Dupond17
Paris18 France18 Martin18 Dupond18
Paris19 France19 Martin19 Dupond19
Paris20 France20 Martin20 Dupond20
…
Paris80 France80 Martin80 Dupond80
Paris81 France81 Martin81 Dupond81
Paris82 France82 Martin82 Dupond82
Paris83 France83 Martin83 Dupond83
Paris84 France84 Martin84 Dupond84
```

```
Paris85 France85 Martin85 Dupond85
Paris86 France86 Martin86 Dupond86
Paris87 France87 Martin87 Dupond87
Paris88 France88 Martin88 Dupond88
Paris89 France89 Martin89 Dupond89
```

Sur la machine utilisée pour tester l'application, le temps d'exécution de cette jointure fut de 547 millisecondes.

Une autre possibilité de programmer cette jointure est de comparer non plus le nom de la ville et le nom du pays mais d'utiliser la propriété **CityRef** qui référence un objet de type **City**. La comparaison de référence ne fonctionne pas directement dans une requête Linq pour Siaqodb, aussi devons nous comparer la propriété **OID** de chaque objet référencé pour savoir s'ils sont ou non identiques. Pour rappel, la propriété **OID** est un identifiant unique attribué par le moteur de Siaqodb à chaque objet lorsqu'il devient persistant. La requête s'écrit alors ainsi :

```
            var query3bis = (from Customer cust in siaqodb
                             from Employee emp in siaqodb
                             where cust.CityRef.OID == emp.CityRef.OID
                             select new { CityName = cust.CityRef.Name,
                                CountryName = cust.CityRef.Country, CName = cust.LastName,
                             EName = emp.LastName }).ToList();
            foreach (var o in query3bis)
            {
                    Console.WriteLine(o.CityName + " " + o.CountryName + " " + o.CName +
                                      " " + o.EName);
            }
```

Le résultat obtenu est exactement le même que celui obtenu lors de l'exécution de la requête précédente. En revanche, le temps d'exécution est légèrement inférieur. Il fut de 468 millisecondes sur la même machine mais cette différence n'est pas vraiment significative.

Nous présentons maintenant une 3^{e} solution qui n'est plus basée sur la jointure mais sur une approche basée sur le modèle « objet ». Celle-ci consiste non plus à se baser sur les classes **Employee** et **Customer** mais sur la classe **City** : elle consiste à parcourir toutes les instances de la classe **City** et pour chaque instance, à considérer chaque client habitant dans cette ville puis pour chaque client, chaque employé habitant dans cette ville. Ainsi il n'y a plus qu'une seule requête à Siaqodb pour retrouver chaque instance de **City**. Le code Linq est écrit ci-dessous.

```
            var query3ter = (from City city in siaqodb
                             from Customer cust in city.Customers
                             from Employee emp in city.Employees
                             select new { CityName = city.Name,
                                 CountryName = city.Country, CName = cust.LastName,
                                 EName = emp.LastName }
                            ).ToList();
            foreach (var o in query3ter)
            {

                    Console.WriteLine(o.CityName + " " + o.CountryName + " " + o.CName +
                                      " " + o.EName);
            }
```

Le résultat est le même mais le temps d'exécution est très inférieur : 16 millisecondes en utilisant toujours la même machine ! Mais ici, nous n'avons plus effectué une jointure mais nous avons, en quelque sorte, préparé cette requête lors de l'écriture des classes **Employee**, **Customer** et **City** en créant les références de

City vers **Employee** et de **City** vers **Customer**. Lors du peuplement de la base de données, les pointeurs peuplant ces références ont été introduits. L'utilisation de ces pointeurs a considérablement réduit le temps d'exécution.

La modélisation par objets n'implique évidemment pas de rechercher les requêtes qui seront mises en œuvre sur le schéma. Mais d'une part, il est préférable de réifier et de factoriser un maximum de propriétés pour évider les redondances. Le fait que les classes **Employee** et **Customer** détiennent un attribut **City** et un attribut **Country** (deux chaînes de caractères) ne constitue pas une bonne approche car les valeurs de ces attributs seront souvent les mêmes dans plusieurs employés et dans plusieurs clients distincts. Il est donc préférable d'introduire des instances de la classe **City**, instances référencées par les instances de **Employeee** et de **Customer**. Pour chaque ville de la base de données, il existe alors une et une seule instance de la classe **City**.

D'autre part, il est naturel en modélisation par objets d'introduire des associations bidirectionnelles (voir, à ce sujet, la notion d'association introduite dans le langage UML).

Jointures internes et externes

La jointure interne est équivalente à la jointure classique.

La jointure externe existe sous trois formes :

- Jointure externe gauche ;
- Jointure externe droite ;
- Jointure externe totale.

Pour rappel la jointure externe retrouve un élément d'une classe même s'il n'est pas associé à un élément de l'autre classe. La valeur des attributs issue de l'élément inexistant est alors null.

Nous reprenons la 3e solution de la jointure classique et l'adaptons pour en faire une jointure externe gauche (nous considérons **Customer** comme l'opérande de gauche).

```
var query3ter = ((from City city in siaqodb
                  from Customer cust in city.Customers
                  from Employee emp in city.Employees
                  select new { CityName = city.Name, CountryName = city.Country,
                    CName = cust.LastName, EName = emp.LastName }
                ).Union(
                  from City city in siaqodb
                  where city.Employees.Count() == 0
                  from Customer cust in city.Customers
                  select new { CityName = city.Name, CountryName = city.Country,
                    CName = cust.LastName, EName = (String)null }
                )).OrderBy(x => x.CountryName).ThenBy(x => x.CityName).ToList();

            foreach (var o in query3ter)
            {

                Console.WriteLine(o.CityName   +   "   "   +   o.CountryName   +   "   "   +
                   (o.CName ?? "Null") + " " + (o.EName ?? "Null"));
            }
```

L'analyse du code source de la requête montre qu'elle est constituée de l'union de la requête de la jointure classique et d'une requête qui renvoie les clients se trouvant dans les villes où il n'y a pas d'employé. Le nom de l'employé est **null** dans ces cas-là. Le résultat de l'exécution du code ci-dessus est le suivant.

```
Paris0 France0 Martin0 Null
Paris1 France1 Martin1 Null
Paris10 France10 Martin10 Dupond10
…
Paris2 France2 Martin2 Null
Paris20 France20 Martin20 Dupond20
…
Paris89 France89 Martin89 Dupond89
Paris9 France9 Martin9 Null
```

Les dix clients Martin0 à Martin9 apparaissent maintenant dans le résultat alors qu'ils habitent dans une ville où aucun employé n'habite.

Remarque : nous avons utilisé un tri par ville (nom du pays suivi du nom de la ville) pour que le résultat soit plus facile à comprendre. Sans ce tri, les dix clients Martin0 à Martin9 auraient été stockés à la fin du résultat.

Il convient aussi de remarquer que nous avons utilisé une coercion de type pour affecter la valeur de l'attribut **EName** à **null** dans la 2e requête de l'union, ce qui a donné le code suivant :

```
                    select new { CityName = city.Name, CountryName = city.Country,
                      CName = cust.LastName, EName = (String)null }
```

Cette coercion de **null** à **(String)null** est imposée par le compilateur de C# pour qu'il puisse déterminer le type de l'attribut **EName**.

Le code de la jointure externe droite apparait à la suite. Il s'agit maintenant d'afficher les employés qui habitent dans une ville où aucun client n'habite, à savoir les employés Dupond90 à Dupond99.

```
var query3ter = ((from City city in siaqodb
                  from Customer cust in city.Customers
                  from Employee emp in city.Employees
                  select new { CityName = city.Name, CountryName = city.Country,
                               CName = cust.LastName, EName = emp.LastName }
              ).Union(
                  from City city in siaqodb
                  where city.Customers.Count() == 0
                  from Employee emp in city.Employees
                    select new { CityName = city.Name, CountryName = city.Country,
                      CName = (String)null, EName = emp.LastName }
              )).OrderBy(x => x.CountryName).ThenBy(x => x.CityName).ToList();

          foreach (var o in query3ter)
          {

              Console.WriteLine(o.CityName + " " + o.CountryName + " " +
                 (o.CName ?? "Null") + " " + (o.EName ?? "Null"));
          }
```

Le résultat de l'exécution se trouve à la suite. Les employés Dupond90 à Dupond99 se trouve à la fin du résultat.

```
Paris10 France10 Martin10 Dupond10
…
Paris89 France89 Martin89 Dupond89
Paris90 France90 Null Dupond90
Paris91 France91 Null Dupond91
Paris92 France92 Null Dupond92
Paris93 France93 Null Dupond93
Paris94 France94 Null Dupond94
Paris95 France95 Null Dupond95
Paris96 France96 Null Dupond96
Paris97 France97 Null Dupond97
Paris98 France98 Null Dupond98
Paris99 France99 Null Dupond99
```

La jointure externe totale consiste à retrouver tous les employés et tous les clients qu'ils habitent ou non dans une ville où habite une instance de l'autre classe.

Le code de cette jointure qui consiste à prendre l'union des trois requêtes qui apparaissent dans la jointure externe gauche et dans la jointure externe droite est le suivant :

```
var query3ter = ((from City city in siaqodb
                  from Customer cust in city.Customers
                  from Employee emp in city.Employees
                  select new { CityName = city.Name, CountryName = city.Country,
                               CName = cust.LastName, EName = emp.LastName }
                ).Union(
                 from City city in siaqodb
                 where city.Employees.Count() == 0
                 from Customer cust in city.Customers
                 select new { CityName = city.Name, CountryName = city.Country,
                   CName = cust.LastName, EName = (String)null }
                ).Union(
                  from City city in siaqodb
                  where city.Customers.Count() == 0
                  from Employee emp in city.Employees
                    select new { CityName = city.Name, CountryName = city.Country,
                      CName = (String)null, EName = emp.LastName }
                )).OrderBy(x => x.CountryName).ThenBy(x => x.CityName).ToList();

            foreach (var o in query3ter)
            {

                Console.WriteLine(o.CityName + " " + o.CountryName + " " +
                   (o.CName ?? "Null") + " " + (o.EName ?? "Null"));
            }
```

Le résultat de la jointure externe totale contient notamment les clients Martin0 à Martin9 et les employés Dupond90 à Dupond99.

```
Paris0 France0 Martin0 Null
Paris1 France1 Martin1 Null
Paris10 France10 Martin10 Dupond10
Paris11 France11 Martin11 Dupond11
Paris12 France12 Martin12 Dupond12
Paris13 France13 Martin13 Dupond13
Paris14 France14 Martin14 Dupond14
Paris15 France15 Martin15 Dupond15
Paris16 France16 Martin16 Dupond16
Paris17 France17 Martin17 Dupond17
Paris18 France18 Martin18 Dupond18
Paris19 France19 Martin19 Dupond19
…
Paris89 France89 Martin89 Dupond89
Paris9 France9 Martin9 Null
Paris90 France90 Null Dupond90
Paris91 France91 Null Dupond91
```

```
Paris92 France92 Null Dupond92
Paris93 France93 Null Dupond93
Paris94 France94 Null Dupond94
Paris95 France95 Null Dupond95
Paris96 France96 Null Dupond96
Paris97 France97 Null Dupond97
Paris98 France98 Null Dupond98
Paris99 France99 Null Dupond99
```