Projektowanie obiektowe oprogramowania
Wykład 9
Wzorce architektury aplikacji (1)
Wiktor Zychla 2013

---

1 Automated code generation

To bardziej technika wspomagająca niż wzorzec, ale wykorzystywana w praktyce bardzo często. Chodzi o generowanie kodu przez automat, ale w taki sposób, żeby „kod który generuje kod" nie był tylko i wyłącznie kodem imperatywnym, ale możliwie jak najbardziej deklaratywnym.

Realizowane przez wiele narzędzi, np. **Text Template Transformation Toolkit** (T4).

Przykład: szablon generowania kodu klasy.

Imperatywnie zadany model klasy:

```
    public partial class PreTextTemplate1
    {
        public string ClassName { get; set; }
        public List<string> FieldNames { get; set; }
    }
```

Deklaratywnie napisany generator:

```
<#@ template language="C#" #>
public class <#= this.ClassName #>
{
   <# foreach ( var field in this.FieldNames ) { #>

   public string <#= field #>;

   <# } #>
}
```

To co należy zapamiętać to to, że generatory kodu powinny udostępniać strategię generowania jedno i dwustopniowego.

W generowaniu **jednostopniowym** (**bezpośrednim**) efektem ewaluacji szablonu jest dokument wynikowy. W powyższym przypadku dla przykładowego modelu wynikiem ewaluacji szablonu byłby tekst:

```
public class Foo
{
   public string Bar;
}
```

W generowaniu **dwustopniowym** efektem ewaluacji szablonu jest kod imperatywny, który trzeba skompilować i wykonać i dopiero wynikiem wykonania tego kodu jest docelowy dokument. Zaletą podejścia dwustopniowego jest możliwość automatyzacji scenariuszy automatycznego generowania (łączenia wielu szablonów, generowania tego samego szablonu wielokrotnie z różnymi parametrami, itd.). W powyższym przypadku wynikiem pierwszego etapu dwustopniowej ewaluacji byłby (raczej nieczytelny, bo wygenerowany przez automat) kod:

```
  #line 1 "ConsoleApplication83\PreTextTemplate1.tt"
    [System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.VisualStudio.TextTe
mplating", "10.0.0.0")]
    public partial class PreTextTemplate1 : PreTextTemplate1Base
    {
        public virtual string TransformText()
        {
            this.Write("\r\npublic class ");

            #line 3 "ConsoleApplication83\PreTextTemplate1.tt"
            this.Write(this.ToStringHelper.ToStringWithCulture(this.ClassName));

            #line default
            #line hidden
            this.Write("\r\n{\r\n    ");

            #line 5 "ConsoleApplication83\PreTextTemplate1.tt"
            foreach ( var field in this.FieldNames ) {

              #line default
              #line hidden
              this.Write("\r\n    public string ");

              #line 7 "ConsoleApplication83\PreTextTemplate1.tt"
              this.Write(this.ToStringHelper.ToStringWithCulture(field));

              #line default
              #line hidden
              this.Write("\r\n\r\n    ");

              #line 9 "ConsoleApplication83\PreTextTemplate1.tt"
            }

            #line default
            #line hidden
            this.Write("}\r\n");
            return this.GenerationEnvironment.ToString();
        }
    }
```

2 Object-relational mapping

Uwaga! Ilustracje pochodzą z podręcznika Patterns of Enterprise Application Architecture.

Wzorzec mapowania obiektowo-relacyjnego jest jednym z możliwych podejść do problemu niezgodności świata obiektowego i świata relacyjnego (relacyjnych baz danych). Wzorzec ORM jest interesujący w kontekście podwzorców.

2.1 Lazy loading

Obiekt nie zawiera danych relacyjnych, ale wie jak je pozyskać.

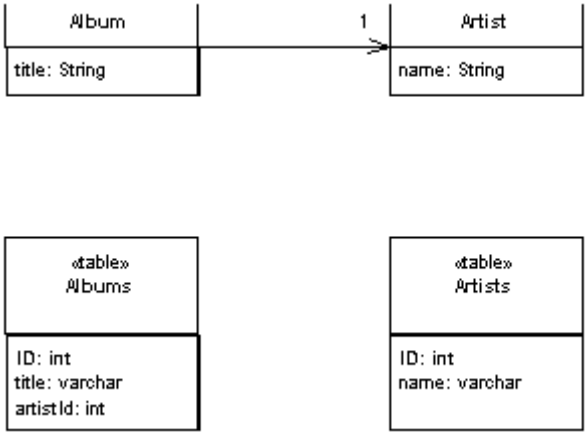
W praktyce Lazy loading realizowany jest na jeden z trzech sposobów:

1. **Lazy initialization** – leniwe właściwości (propercje) zawierają normalny, imperatywny kod dostępu do danych i flagę boole'owską, która działa jak strażnik, dzięki któremu kod dostępu do danych wykonuje się tylko za pierwszym razem
2. **Virtual proxy** – silnik mapowania obiektowo relacyjnego automatycznie tworzy wirtualne proxy do zwracanych obiektów, które to proxy mają automatycznie dołączony kod implementujący Lazy initialization (różnica jest taka, że nie trzeba tego kodu implementować bezpośrednio w klasie dziedzinowej)
3. **Value holder** – implementacja leniwych składowych deleguje pozyskiwanie wartości do zewnętrznych obiektów, które zarządzają dostępem do danych

W praktyce wszystkie trzy metody są wykorzystywane często, z przewagą 2. i 3.

2.2 Navigation properties (aka Foreign key mapping)

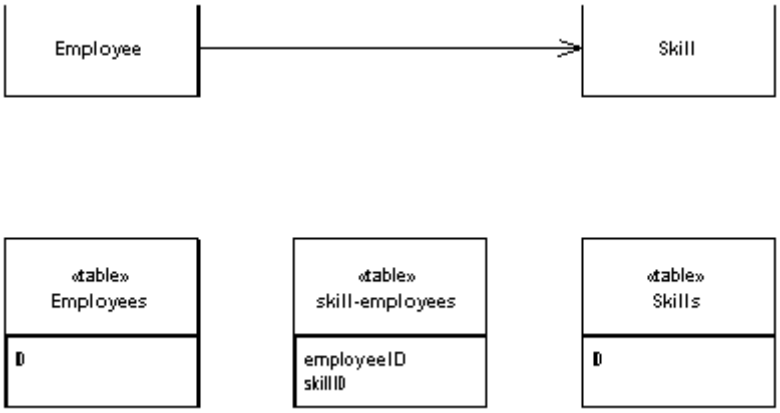
Mapowanie kluczy obcych na relacje między klasami.



To bardzo naturalne oczekiwanie. W modelu z diagramu powyżej oczekiwalibyśmy właściwości nawigacyjnych (*navigation properties*) w obie strony – klasa **Album** powinna mieć właściwość typu **Artist**, a klasa **Artist** właściwość typu **IEnumerable<Album>**.

2.3 Many-to-many (aka Association table mapping)

Automatycznie modeluj asocjację relację wiele-do-wiele między obiektami jako pomocniczą tabelę

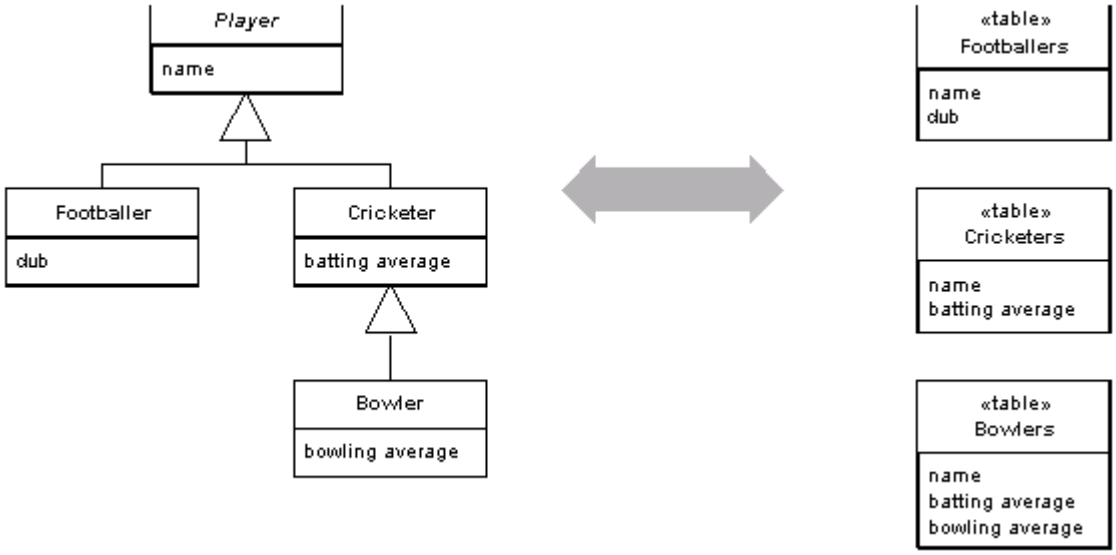


W prawidłowej implementacji tego podwzorca ważne jest to, żeby silnik mógł samodzielnie zarządzać tabelą odwzorowującą relację. W praktyce spotyka się często implementacje niepełne – za pomocą podwzorca Navigation modelowane są jawnie relacje między obiema klasami a istniejącą jawnie klasą-modelem dla tabeli odwzorowującej relację.

2.4 Concrete table/Single table/Class table Inheritance

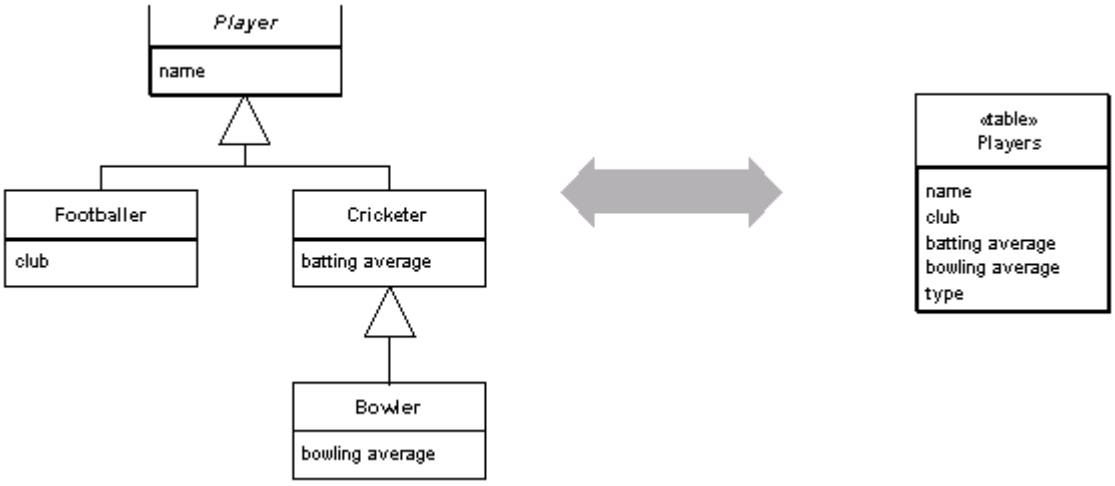
Trzy możliwe strategie odwzorowania w strukturze relacyjnej relacji dziedziczenia.

1. **Concrete table**



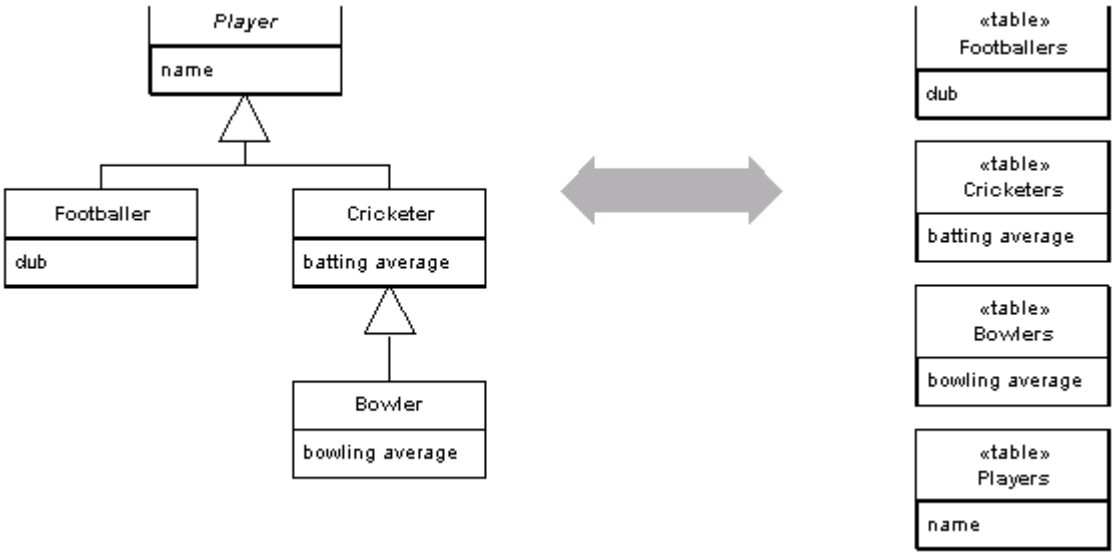
W tym podejściu hierarchia mapuje się na osobne tabele dla każdej z klas.

2. **Single table**



W tym podejściu hierarchia mapuje się na jedną tabelę zawierającą sumę kolumn z całej hierarchii oraz dodatkową kolumnę **type**, dzięki której silnik może zorientować się jakiego typu jest konkretny wiersz tabeli.

3. **Class table**



W tym podejściu hierarchia mapuje się na tabele, ale każda z tabel odwzorowuje tylko te kolumny, które są właściwe dla klasy, którą modeluje.

2.5 1st level cache (aka Identity map)

1st level cache - zapytania o obiekty o konkretnych identyfikatorach powinny być cache'owane. Identity map to jedna z technik implementacji, w której wewnętrznie wykorzystuje się kolekcję asocjacyjną, mapującą wartość identyfikatora na obiekt.

2.6 2nd level cache

2nd level cache – wybrane zapytania powinny być możliwe do cache'owania. Powinien istnieć jawny mechanizm do zarządzania cache. Implementacja podsystemu cache'owania powinna być wymienna.

2.7 Query language

Silnik powinien udostępniać obiektowy język zadawania zapytań, **inny** niż SQL. Dobrym przykładem języka zapytań jest **Linq**, inne przykłady to HQL, Criteria API, JPQL.

2.8 Global filter

Query language powinien dawać możliwość określenia „globalnego filtra” w taki sposób żeby był konsekwentnie automatycznie aplikowany do wszystkich poziomów zagnieżdżonych zapytań.

2.9 Metadata mapping

Silnik mapowania obiektowo-relacyjnego powinien udostępniać różne strategie definiowania metadanych:

1. Metadane są częścią definicji klas (atrybuty, określona hierarcha dziedziczenia)
2. Metadane są zewnętrzne w stosunku do definicji klas (definiowane deklaratywnie w XML lub imperatywnie za pomocą API)

2.10 Model first vs Code first

Różne strategie definiowania metadanych mogą przełożyć się na różną filozofię logistyki orm:

1. Model first – najpierw modelowana jest struktura relacyjna, a model obiektowy ją odwzorowuje (często jest generowany przez automat; patrz **Automated code generation**)
2. Code first – najpierw modelowana jest struktura obiektowa, a struktura relacyjna tylko ją odwzorowuje (często jest generowana i zarządzana przez automat; przykład z wykładu Entity Framework Code First + Migrations)