Projektowanie obiektowe oprogramowania
Wykład 9
Wzorce architektury aplikacji (1)
Wiktor Zychla 2021

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). Inne przykładowe narzędzia na tej liście.

Przykład: Szablon generowania kodu klasy.

```
<#@ template debug="false" hostspecific="false" language="C#" #>
<#@ assembly name="System.Core" #>
<#@ output extension=".cs" #>

<#
    string ClassName = "Foo";
    string[] FieldNames = { "Bar", "Qux" };
#>

public class <#= ClassName #>
{
   <# foreach ( var field in FieldNames ) { #>

   public string <#= field #>;

   <# } #>
}
```

Proszę zwrócić uwagę na język szablonu:

- elementy imperatywne są umieszczane w blokach: tu <# …. #> - te elementy są podczas ewaluacji szablonu wykonywane jako kod. Interpreter szablonu zwykle wymaga kodu w jakimś konkretnym języku, tu dla T4 wymagany jest jakiś język środowiska .NET, za pomocą dyrektywy **<#@ template language #>** określa się który język będzie używany we fragmentach imperatywnych
- elementy deklaratywne są umieszczane poza blokami – te elementy są kopiowane do wyjścia bez zmiany, nawet jeśli są elementami języka programowania

Wykonanie szablonu którego elementy imperatywne są napisane w jakimś jezyku (tu: C#) może więc „wygenerować” plik tekstowy, który jest kodem dowolnego innego języka imperatywnego (C#, Java, C++, itd.) lub deklaratywnego (XML, HTML, SQL etc.)

```
"c:\Program Files (x86)\Microsoft Visual
Studio\2019\Enterprise\Common7\IDE\TextTransform.exe" template.t4
```

spowoduje wygenerowanie kodu:

```
public class Foo
{
    public string Bar;

    public string Qux;
}
```

Szablon może pozyskać dane w dowolny sposób – skoro jego część imperatywna może korzystać z biblioteki standardowej, to nie ma problemu żeby odczytywać dane z pliku, z bazy danych itd.

Można więc wyobrazić sobie szereg praktycznych scenariuszy:

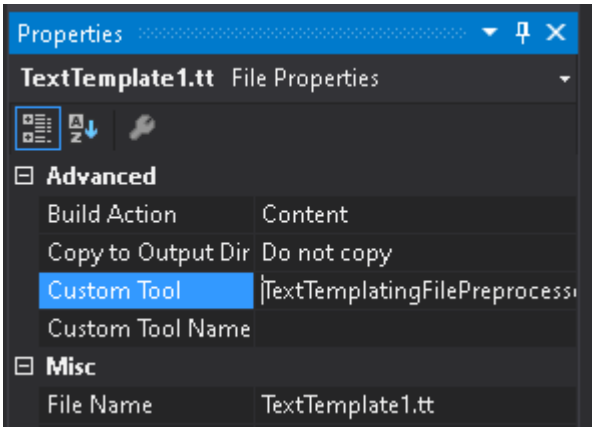
- odczytywanie danych z pliku XML, na tej podstawie budowanie klas modeli (C#, Java)
- odczytywanie danych z bazy danych, na tej podstawie budowanie klas modeli (C#, Java)
- odczytywanie danych z bazy danych, na tej podstawie budowanie plików XML
- odczytywanie danych z pliku XLS, na tej podstawie budowanie kwerend SQL
- itd.

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 (jak wyżej).

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).

W przypadku T4 przy uruchamianiu szablonu z z poziomu Visual Studio należy we właściwościach zmienić **TextTemplatingFileGenerator** na **TextTemplatingFilePreprocessor:**



Zrefaktoryzujmy powyższy przykład.

Krok 1. Określenie parametrów szablonu na zewnątrz, w definicji klasy częściowej:

```
    public partial class TextTemplate1
    {
        public string ClassName;
        public string[] FieldNames;
    }
```

Krok 2. Korekta szablonu w taki sposób żeby odwoływał się do pól-parametrów:

```
<#@ template debug="false" hostspecific="false" language="C#" #>
<#@ assembly name="System.Core" #>
<#@ output extension=".cs" #>

public class <#= this.ClassName #>
{
   <# foreach ( var field in this.FieldNames ) { #>

   public string <#= field #>;

   <# } #>
}
```

Krok 3. Zamiana strategii generowania szablonu na **TextTemplatingFilePreprocessor**. Efektem generacji będzie tym razem plik **TextTemplate1.cs** zawierający kod imperatywny klasy **TextTemplate1**, która będzie posiadać metodę **TransformText**, wykonującą proces ewaluacji szablonu (metoda zwraca obiekt typu **string**). Taką klasę można włączyć do innej aplikacji a metodę wywołać w dowolnym kontekście.

Krok 4. Automatyzacja szablonu:

```
        static void Main( string[] args )
        {
            TextTemplate1 tpl = new TextTemplate1();
            tpl.ClassName = "Foo";
            tpl.FieldNames = new [] { "Bar", "Qux" };

            Console.WriteLine( tpl.TransformText() );
            Console.ReadLine();
        }
```

Taki dwustopniowy model ewaluacji jest ogólniejszy – automatyzacja szablonu może odbywać się w kontekście złożonego procesu przetwarzania danych wejściowych i generowania wielu szablonów na podstawie jednego zbioru danych.

Na przykład w ten sposób można zrealizować scenariusz w którym z **jednej** bazy danych produkowane jest **wiele** plików z modelami kodu – osobny plik dla każdej tabeli z bazy.

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). To co daje podsystem ORM to możliwość używania **obiektów** w kodzie obiektowym i translacji tychże na **rekordy w tabelach** w bazach danych.

Wzorzec ORM jest jednym z częściej wykorzystywanych wzorców architektury warstwy dostępu do danych. Istniejące implementacje dostępne są dla wszystkich głównych platform wytwarzania aplikacji:

- Hibernate/nHibernate – Java/.NET
- Entity Framework - .NET
- JPA – Java
- Sequelize – node.JS
- Active Record – Ruby
- Itd.

Wzorzec ORM jest interesujący w kontekście podwzorców. Zachęcam więc Państwa do zapoznania się z wybraną przez siebie implementacją i ocenę jej pod kątem dostępności niżej opisanych funkcjonalności.

2.1 Database first vs Model first vs Code first

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

1. **Database** first – najpierw modelowana jest struktura relacyjna, a model obiektowy ją odwzorowuje (często jest generowany przez automat; patrz **Automated code generation**)
2. **Model** first – najpierw modelowana jest abstrakcyjna struktura modelu mapowania bazy danych na kod klas. Z modelu mapowania wynika zarówno struktura bazy danych jak i kodu klas.
3. **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)

2.2 Metadata mapping

Silnik mapowania obiektowo-relacyjnego powinien udostępniać różne strategie definiowania metadanych, czyli informacji na temat szczegółów implementacyjnych utrwalania elementów modelu obiektowego w strukturze relacyjnej (m.in. nazwy tabel, kolumn, typy kolumn, długości kolumn literalnych, oznaczenia co do wymagalności, sposobu kodowania itd. – wszystko to co nie wynika wprost z modelu obiektowego).

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)

Niektóre technologie mapowania obiektowo relacyjnego dostarczają dodatkowego mechanizmu zarządzania tzw. **migracjami** czyli wersjonowaniem modelu relacyjnego.

Przykład:

Mapowanie metadanych Hibernate:

```
<?xml version="1.0"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" assembly="Eg"
    namespace="Eg">

    <class name="Cat" table="CATS" discriminator-value="C">
        <id name="Id" column="uid" type="Int64">
            <generator class="hilo"/>
        </id>
        <discriminator column="subclass" type="Char"/>
        <property name="BirthDate" type="Date"/>
        <property name="Color" not-null="true"/>
        <property name="Sex" not-null="true" update="false"/>
        <property name="Weight"/>
        <many-to-one name="Mate" column="mate_id"/>
        <set name="Kittens">
            <key column="mother_id"/>
            <one-to-many class="Cat"/>
        </set>
        <subclass name="DomesticCat" discriminator-value="D">
            <property name="Name" type="String"/>
        </subclass>
    </class>

    <class name="Dog">
        <!-- mapping for Dog could go here -->
    </class>

</hibernate-mapping>
```

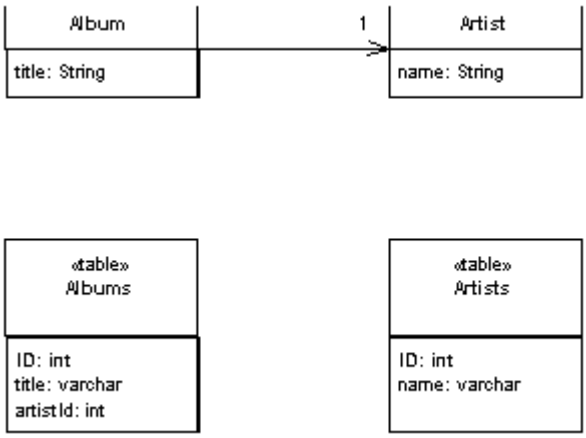
Mapowanie metadanych Entity Framework:

```
public class O_ZawodMap : EntityTypeConfiguration<O_Zawod>
{
    public O_ZawodMap(bool DeleteSoftly)
    {
        // Primary Key
        this.HasKey(t => t.OID);

        // Table & Column Mappings
        this.ToTable("O_Zawod");
        this.Property(t => t.OID).HasColumnName("OID");
        this.Property(t => t.ID_JEDNOSTKA_SKLADOWA).HasColumnName("ID_JEDNOSTKA_SK
LADOWA");

        this.Property(t => t.DataOd).HasColumnName("DataOd");
        this.Property(t => t.DataDo).HasColumnName("DataDo");
```

```
        // Relationships
        this.HasRequired(t => t.Jednostka_Skladowa)
            .WithMany(t => t.Zawody)
            .HasForeignKey(d => d.ID_JEDNOSTKA_SKLADOWA);
        this.HasRequired(t => t.S31)
            .WithMany(t => t.Zawody)
            .HasForeignKey(d => d.ID_S31);

    }
}
```

2.3 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 **Album[]** (tablica lub jakikolwiek inny typ zbiorowy)

```
public class Artist
{
    public Album[] Albums;

    ...
}

public class Album
{
    public long ID_Artist;

    public Artist Artist;

    ...
}
```

2.4 Lazy loading

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

Lazy loading dotyczy zwykle procesu ładowania zawartości właściwości nawigacyjnych. Jeżeli klasa **Artist** z poprzedniego diagramu zawiera właściwość typu **Album[]**, to inicjowanie tej właściwości za każdym razem podczas materializacji obiektu typu **Artist** mogłoby być nieefektywne.

Wystarczy wyobrazić sobie zapytanie

```
SELECT * FROM ARTISTS WHERE ID = 1
```

którego intencją jest zmaterializowanie obiektu typu **Artist** o identyfikatorze 1, tyle że dodatkowo materializuje ono zawartość właściwości **Albums** i tak się niefortunnie składa, że pewien artysta wydał już wiele set albumów, a z kolei każdy album ma właściwość nawigacyjną wskazującą na wydawcę, wydawca na siedzibę, siedziba na adres itd.

Gorliwe materializowanie takiego drzewa obiektów połączonych przez właściwości nawigacyjne nie tylko byłoby nieefektywne – zwykle też nie ma wręcz potrzeby materializowania go, bo kod żądający materializacji początkowego obiektu **Artist** w ogóle mógłby nawet nie odczytywać wartości żadnej z właściwości nawigacyjnych.

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

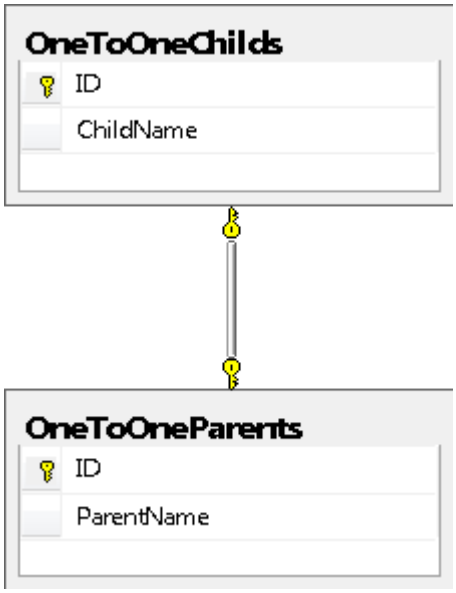
1. **Lazy initialization** – leniwe właściwości (propercje) zawierają normalny, imperatywny kod dostępu do danych i flagę boole'owską, która działa jak strażnik, dzięki któremu kod dostępu do danych wykonuje się tylko za pierwszym razem
2. **Virtual proxy** – silnik mapowania obiektowo relacyjnego automatycznie 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.

2.5 One-to-one

Pozwól wybrać model dla relacji jeden-do-jeden między tabelami

Model relacyjny w którym występuje mapowanie jeden-do-jeden realizowany jest zwyczajowo w taki sposób, że jedna z tabel (nadrzędna) ma kolumnę ID oznaczoną jako klucz główny, a druga z tabel (podrzędna) kolumnę ID oznaczoną jako klucz główny i równocześnie klucz obcy do pierwszej tabeli.



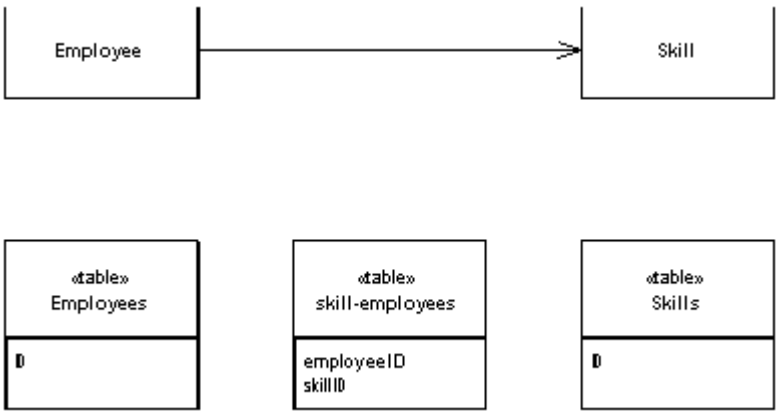
W ten sposób powiązane ze sobą rekordy mają **tę samą** wartość klucza ID.

Silnik mapowania obiektowo-relacyjnego powinien pozwalać na dwa sposoby zamodelowania takiej struktury relacyjnej po stronie modelu obiektowego:

- Jedna klasa mapowana na dwie tabele (aka **split entity**) – z punktu widzenia modelu obiektowego mamy jedną klasę, fizycznie odwzorowywaną w dwóch tabelach
- Dwie osobne klasy, połączone relacją (i prawdopodobnie właściwościami nawigacyjnymi w obie strony relacji)

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

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

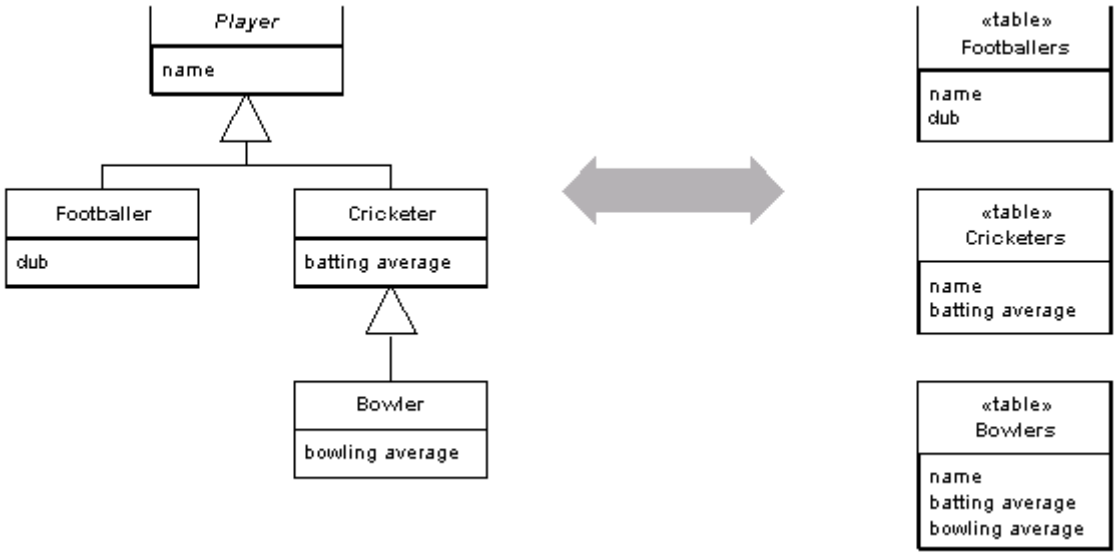


W prawidłowej implementacji tego podwzorca ważne jest to, żeby silnik mógł samodzielnie zarządzać tabelą odwzorowującą relację. W praktyce spotyka się często implementacje niepełne – za pomocą podwzorca Navigation modelowane są jawnie relacje między obiema klasami a istniejącą jawnie klasą-modelem dla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

2.7 Concrete table/Single table/Class table Inheritance

Trzy możliwe strategie odwzorowania dziedziczenia w strukturze relacyjnej.

1. **Concrete table (aka Table per Concrete Type, TPC)**



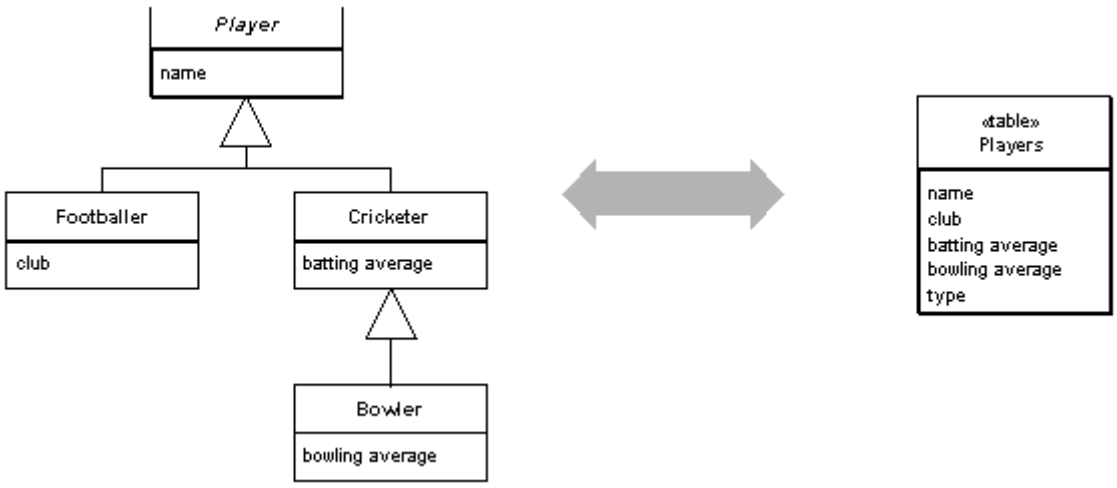
W tym podejściu hierarchia mapuje się na osobne tabele dla każdej z konkretnych klas. Struktura relacyjna nie jest świadoma relacji dziedziczenia w modelu obiektowym.
Plusy:

- to najprostsze możliwe podejście do mapowania dziedziczenia

Minusy:

- problematyczne zarządzanie identycznościami (technicznie baza danych wstawi do różnych przecież tabel obiekty o tym samym identyfikatorze – a przecież modelujemy jedną i tę samą hierarchię obiektów, więc identyfikatory powinny być różne!)
- problematyczne mapowanie relacji do zewnętrznych klas (relacje musiałyby dotyczyć każdej z tabel z osobna!)
- utrudnione zarządzanie strukturą relacyjną
- zapytania o obiekty mają z konieczności często postać UNION

2. **Single table (aka Table per Hierarchy, TPH)**



W tym podejściu hierarchia mapuje się na jedną tabelę zawierającą sumę kolumn z całej hierarchii oraz dodatkową kolumnę **dyskryminatora**, która określa rzeczywisty typ obiektu reprezentowanego przez wiersze tabeli.

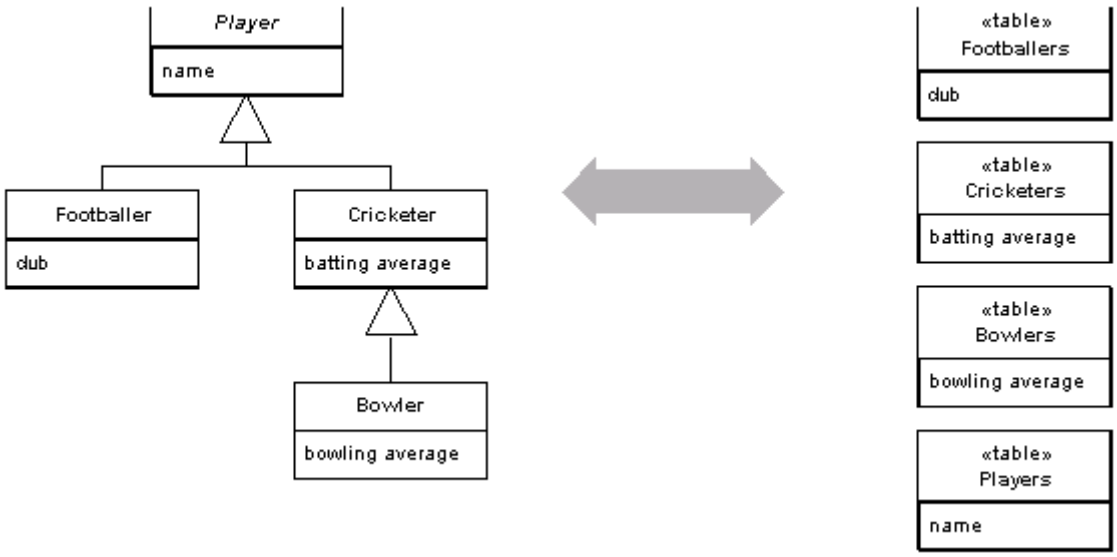
Plusy:

- Jedna tabela na hierarchię oznacza, że zapytania są wykonywane efektywnie

Minusy:

- Wszystkie te kolumny z klas potomnych, które nie występują w typie bazowym, muszą zezwalać w strukturze relacyjnej na przechowanie wartości NULL
- Łamie trzecią postać normalną (kolumna dyskryminatora determinuje funkcjonalnie wartości dodatkowych kolumn, ale nie jest ona równocześnie częścią klucza głównego)

3. **Class table (aka Table per Type, TPT)**



W tym podejściu hierarchia mapuje się na tabele, ale każda z tabel odwzorowuje tylko te kolumny, które są właściwe dla klasy, którą modeluje. Dodatkowo, każda z tabel odwzorowujących klasy potomne zawiera wskazanie (klucz obcy) do tabeli odwzorowującej klasę bazową.

Plusy:

- Struktura relacyjne jest poprawnie znormalizowana
- Zarządzanie odwzorowaniem hierachii typów na strukturę relacyjną jest oczywiste

Minusy:

- zapytania do skomplikowanych hierachii są złożone i często nieefektywne (konieczność generowania złączeń typu INNER JOIN)

Więcej:

http://weblogs.asp.net/manavi/archive/2010/12/24/inheritance-mapping-strategies-with-entity-framework-code-first-ctp5-part-1-table-per-hierarchy-tph.aspx

http://weblogs.asp.net/manavi/archive/2010/12/28/inheritance-mapping-strategies-with-entity-framework-code-first-ctp5-part-2-table-per-type-tpt.aspx

http://weblogs.asp.net/manavi/archive/2011/01/03/inheritance-mapping-strategies-with-entity-framework-code-first-ctp5-part-3-table-per-concrete-type-tpc-and-choosing-strategy-guidelines.aspx

2.8 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.

2.9 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.

2.10 Query language

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

Przykłady języków zapytań:

```
// LINQ
unitOfWork.User
    .Where(u => u.Name.StartsWith("K") && u.Age > 18)
    .OrderBy(u => u.Age);
```

```
// HQL
select order.id, sum(price.amount), count(item)
from Order as order
join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all(
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount)desc
```

```
// Hibernate criteria query API
var cats = sess.CreateCriteria<Cat>()
    .Add(Expression.Like("Name", "Fritz%"))
    .Add(Expression.Or(
        Expression.Eq("Age", 0),
```

```
        Expression.IsNull("Age")
    ))
    .List<Cat>();
```

2.11 Global filter

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

Przykład:

Metadane:

```
<filter-def name = "effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>
```

i potem w kodzie

```
ISession session = ...;

session.EnableFilter("effectiveDate").SetParameter("asOfDate", DateTime.Today);

var results = session.CreateQuery("from Employee as e where e.Salary > :targetSala
ry")
          .SetInt64("targetSalary", 1000000L)
          .List<Employee>();
```

2.12 Soft Delete (logical delete)

Filozofia odwzorowywania danych w bazie danych, w której nigdy nie usuwa się fizycznie rekordów, a zamiast tego – znakuje się je jako usunięte w kolumnie dodatkowego dyskryminatora.

Wymaga wsparcia w języku zapytań (Query language), które najczęściej realizuje się jako globalny filtr (Global filter). Dodatkowo, wymaga wsparcia w warstwie mapowania (wykonanie usunięcia powinno zostać odwzorowane w strukturze relacyjnej jako oznakowanie).

Zalety:

- Często znakowanie (UPDATE) jest szybsze niż usuwanie (DELETE)
- Istnieje możliwość audytu (w tym przywrócenia!) usuniętych danych

Wady:

- Dodatkowa kolumna dyskryminatora musi być uwzględniana w zapytaniach (SELECT)

3 Przykład dla Entity Framework 6.1

3.1 Klasy modeli

```
public class ManyToManyLeft
 {
     public ManyToManyLeft()
     {
         this.Rights = new List<ManyToManyRight>();
     }

     public int ID { get; set; }
     public string TheLeftProperty { get; set; }

     public ICollection<ManyToManyRight> Rights { get; set; }
 }
```

```
public class ManyToManyRight
{
    public ManyToManyRight()
    {
        this.Lefts = new List<ManyToManyLeft>();
    }

    public int ID { get; set; }
    public string TheRightProperty { get; set; }

    public ICollection<ManyToManyLeft> Lefts { get; set; }
}
```

```
public class OneToOneChild
{
    public int ID { get; set; }
    public string ChildName { get; set; }

    public virtual OneToOneParent Parent { get; set; }
}
```

```
public class OneToOneParent
{
    public int ID { get; set; }
    public string ParentName { get; set; }

    public virtual OneToOneChild Child { get; set; }
}
```

```
public class SplitEntity
{
    public int ID { get; set; }

    public string Property1 { get; set; }
    public string Property2 { get; set; }
}
```

```
public abstract class TPHBase
{
    public int ID { get; set; }

    public string CommonProperty { get; set; }
}

public class TPHChild1 : TPHBase
{
    public string Child1Property { get; set; }
}

public class TPHChild2 : TPHBase
{
    public int? Child2Property { get; set; }
}
```

```
public abstract class TPTBase
{
    public int ID { get; set; }

    public string CommonProperty { get; set; }
}

public class TPTChild1 : TPTBase
{
    public string Child1Property { get; set; }
}

public class TPTChild2 : TPTBase
{
    public int? Child2Property { get; set; }
}
```

3.2 Klasa kontekstu

```
public class ExampleContext : DbContext
{
    // table-per-hierarhy
    public IDbSet<TPHBase> TPH { get; set; }
    // table-per-type
    public IDbSet<TPTBase> TPT { get; set; }

    // one to one
    public IDbSet<OneToOneParent>  OneToOneParent { get; set; }
    public IDbSet<OneToOneChild>   OneToOneChild { get; set; }

    // split entity
    public IDbSet<SplitEntity>     SplitEntity { get; set; }

    // many to many
    public IDbSet<ManyToManyLeft>  Lefts { get; set; }
    public IDbSet<ManyToManyRight> Rights { get; set; }

    protected override void OnModelCreating( DbModelBuilder modelBuilder )
    {
        // tph
```

```
        // domyślnie - nic nie trzeba konfigurować

        // zmiana dyskryminatora
        modelBuilder.Entity<TPHBase>()
            .ToTable( "TPHBase" )
            .Map<TPHChild1>( m => m.Requires( "Dyskryminator" ).HasValue( "c1" ) )
            .Map<TPHChild2>( m => m.Requires( "Dyskryminator" ).HasValue( "c2" ) )
;

        // tpt
        modelBuilder.Entity<TPTBase>().ToTable( "TPTBase" );
        modelBuilder.Entity<TPTChild1>().ToTable( "TPTChild1" );
        modelBuilder.Entity<TPTChild2>().ToTable( "TPTChild2" );

        // one-to-one
        modelBuilder.Entity<OneToOneChild>()
            .HasRequired( e => e.Parent )
            .WithOptional( e => e.Child );

        // entity split
        modelBuilder.Entity<SplitEntity>()
            .Map( m =>
            {
                m.Properties( p =>
                    new
                    {
                        p.ID,
                        p.Property1
                    } );
                m.ToTable( "SplitEntity" );
            } )
            .Map( m =>
            {
                m.Properties( p =>
                    new
                    {
                        p.Property2
                    } );
                m.ToTable( "SplitEntityDetails" );
            } );

        // many to many
        modelBuilder.Entity<ManyToManyLeft>()
            .HasMany<ManyToManyRight>( s => s.Rights )
            .WithMany( c => c.Lefts )
            .Map( cs =>
            {
                cs.MapLeftKey( "LeftId" );
                cs.MapRightKey( "RightId" );
                cs.ToTable( "ManyToManyLeftRight" );
            } );

        base.OnModelCreating( modelBuilder );

        // tpc -
 nie pokazujemy przykładu, bo trzebaby ręcznie zarządzać identycznościami
    }
}
```

3.3 Orkiestracja

```
class Program
{
    static void Main( string[] args )
    {
        Console.WriteLine( "starting" );

        Database.SetInitializer<ExampleContext>( new DropCreateDatabaseIfModelChan
ges<ExampleContext>() );

        // tph
        Console.WriteLine( "tph" );
        using ( var ctx = new ExampleContext() )
        {
            var c1            = new TPHChild1();
            c1.Child1Property = "child1property";
            c1.CommonProperty = "commonproperty1";

            ctx.TPH.Add( c1 );

            var c2            = new TPHChild2();
            c2.Child2Property = 5;
            c2.CommonProperty = "commonproperty2";

            ctx.TPH.Add( c2 );

            ctx.SaveChanges();

            foreach ( var c in ctx.TPH )
            {
                Console.WriteLine( c.GetType() );
            }
        }

        // tpt
        Console.WriteLine( "tpt" );
        using ( var ctx = new ExampleContext() )
        {
            var c1            = new TPTChild1();
            c1.Child1Property = "child1property";
            c1.CommonProperty = "commonproperty1";

            ctx.TPT.Add( c1 );

            var c2            = new TPTChild2();
            c2.Child2Property = 5;
            c2.CommonProperty = "commonproperty2";

            ctx.TPT.Add( c2 );

            ctx.SaveChanges();

            foreach ( var c in ctx.TPT )
            {
```

```
            Console.WriteLine( c.GetType() );
        }
    }

    // one to zero-or-one
    Console.WriteLine( "one to zero-or-one" );
    using ( var ctx = new ExampleContext() )
    {
        var p1        = new OneToOneParent();
        p1.ParentName = "parent1" + DateTime.Now;

        ctx.OneToOneParent.Add( p1 );

        var p2        = new OneToOneParent();
        p2.ParentName = "parent2" + DateTime.Now;
        var c2        = new OneToOneChild();
        c2.ChildName  = "child2" + DateTime.Now;
        c2.Parent     = p2;

        ctx.OneToOneChild.Add( c2 );

        ctx.SaveChanges();

        foreach ( var p in ctx.OneToOneParent.ToList() )
        {
            Console.WriteLine( "{0} - {1}",
            p.ParentName, p.Child != null ? p.Child.ChildName : "[brak]" );
        }
    }

    // split entity
    Console.WriteLine( "split entity" );
    using ( var ctx = new ExampleContext() )
    {
        SplitEntity entity = new SplitEntity();
        entity.Property1   = "foo1" + DateTime.Now;
        entity.Property2   = "foo2" + DateTime.Now;

        ctx.SplitEntity.Add( entity );

        ctx.SaveChanges();

        foreach ( var e in ctx.SplitEntity )
        {
            Console.WriteLine( "{0} {1}", e.Property1, e.Property2 );
        }
    }

    // many to many
    Console.WriteLine( "many to many" );
    using (var ctx = new ExampleContext())
    {
        ManyToManyLeft l1 = new ManyToManyLeft()
        { TheLeftProperty = "l1" + DateTime.Now };

        ManyToManyRight r1 = new ManyToManyRight()
        { TheRightProperty = "r1" + DateTime.Now };
        ManyToManyRight r2 = new ManyToManyRight()
        { TheRightProperty = "r2" + DateTime.Now };
```

```
            l1.Rights.Add( r1 );
            l1.Rights.Add( r2 );

            ctx.Lefts.Add( l1 );

            ctx.SaveChanges();

            l1 = ctx.Lefts.FirstOrDefault( l => l.ID == l1.ID );
            foreach ( var right in l1.Rights )
            {
                Console.WriteLine( "{0}", right.TheRightProperty );
            }
        }

        Console.WriteLine( "finished" );

        Console.ReadLine();
    }
}
```