Rudi van den Belt

Domain Driven Software Factory

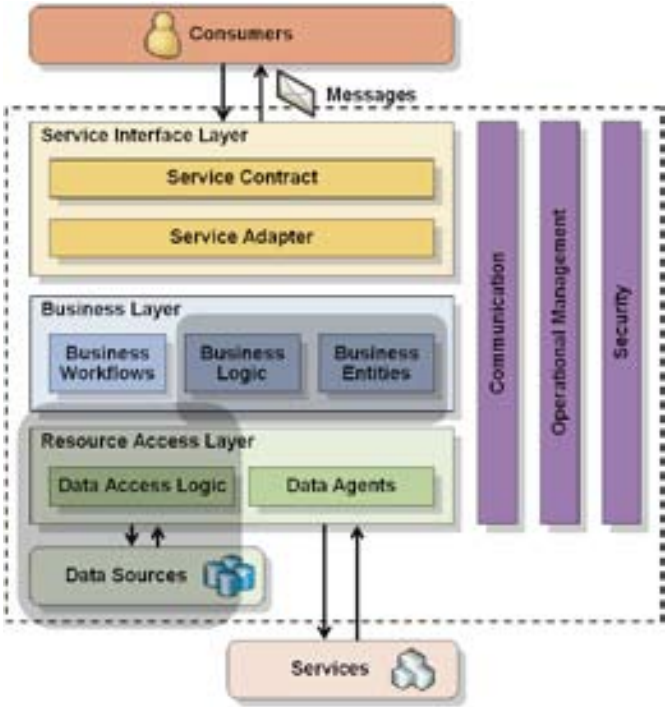
DOMAIN DRIVEN DESIGN MET DE WEB SERVICE SOFTWARE FACTORY

Serviceoriëntatie en domain driven design zijn onderwerpen die de laatste tijd vaak de revue passeren. In dit artikel wordt beschreven hoe je de Web Service Software Factory van Microsoft met NHibernate kunt combineren.

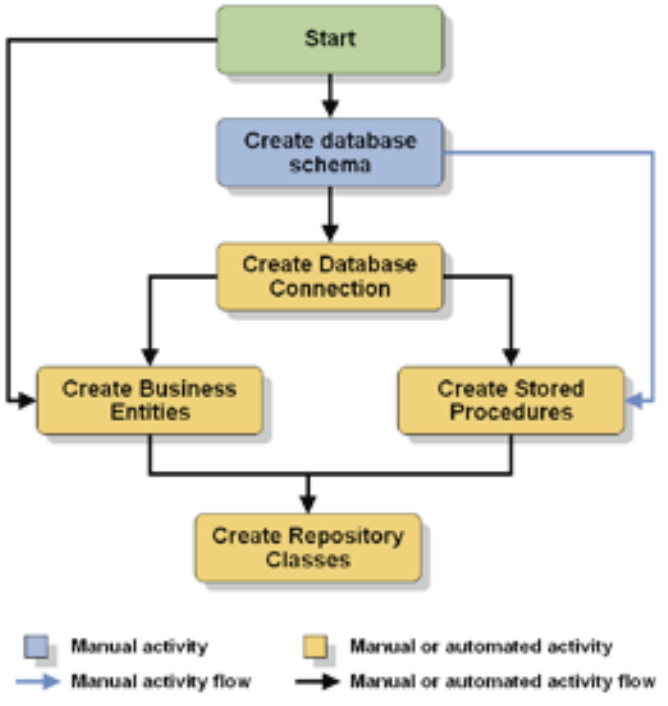
Om de zoveel jaar, zeg 4 à 5, vindt er een verschuiving plaats in technologie, ook wel een 'paradigm shift' genoemd. In dit geval de technologie van gedistribueerde applicaties. We kennen allemaal nog wel de tijd dat we het alleen over client-server hadden. Dit bleek niet de heilige graal te zijn, zodat we tegenwoordig ons heil zoeken in service oriented architecture, SOA, na onder meer een DCOM-poging met gedistribueerde objecten. Los van hoe we de software distribueren, is de manier van opslag gelijk gebleven, namelijk relationeel. Tegelijkertijd heeft object-oriented ontwikkelen (OO) zich de afgelopen jaren bewezen als de manier om software te ontwerpen. SOA and OO zijn niet tegenstrijdig. Je kunt SOA zien als een wrapper om OO heen. Deze wrapper maakt de software geschikt om te distribueren. Deze architectuur is terug te vinden in de documentatie van de Web Service Software Factory van Microsoft [WSSF] in afbeelding 1.
Dit artikel beperkt zich tot de donker gemaakte gebieden in de businesslayer en de resource access-layer en data-sources.De business-layer in afbeelding 1 is nog steeds object-oriented. Uitgangspunt is dat we een complexe applicatie aan willen kunnen. Volgens Fowler [Fow03] hebben we dan te maken met een rich *domain model*. Een dergelijk domain model moet vertaald kunnen worden naar de database volgens het *data mapper pattern*. Microsoft maakt deze mapping mogelijk door het inzetten van de resource access-layer. Deze laag heeft als doel om het domein en de databron gescheiden te houden. De toegang tot deze laag vindt plaats via het repository-pattern. Voor elk domain-object (Business Entity) is er een repository die weet hoe hij die objecten kan creëren, opslaan, zoeken en verwijderen. Met de Data Access Guidance Package van Microsoft is het mogelijk om dit soort repositories aan te maken. Helaas zit hier een OO-discrepantie; de visie van de factory is dat er eerst een database dient te zijn, zie afbeelding 2.
Dit is in tegenstrijd met hoe we willen ontwikkelen, namelijk vanuit de domeingedachte, domain driven design waarbij het domeinmodel zo 'schoon' mogelijk is (Plain Old CLR Object) [NILSON]. Om de pragmatici hiervan te overtuigen het volgende;. Stel dat je op deze manier werkt en er vindt een wijziging plaats in een tabel, laten we zeggen Customer; er komt een attribuut bij. Ten eerste moeten we de tabel aanpassen. Vervolgens alle relevante stored procedures aanpassen (als je geluk hoef je alleen opnieuw te genereren). De business entity opnieuw genereren/aanpassen. Kortom een behoorlijk aantal stappen, dat foutgevoelig is. Zou het niet mooi zijn als we in ons domeinmodel, in dit geval de class Customer, alleen een property hoeven toe te voegen en dat de rest vanzelf werkt? Nou dat kan! (Dit is een eenvoudig voorbeeld van de mismatch, voor meer nadelen zie [ARJRN]).

NHibernate++

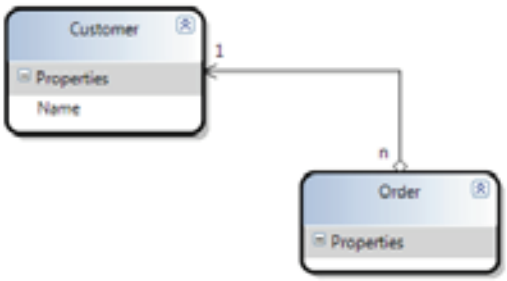
In .NET magazine #15 is al een artikel gepubliceerd over NHibernate [NHIBERNATE]. Zeker nu Microsoft ADO.NET Entity Framework heeft uitgesteld tot 2008 is het zeker interessant om NHibernate als alternatief te nemen. Een gedachte die belangrijk is om te onthou-



Afbeelding 1. Architectuur van Web Service Software Factory



Afbeelding 2. Data Access Guidance Package-werkwijze



Afbeelding 3. Domeinmodel met behulp van ActiveWriter DSL

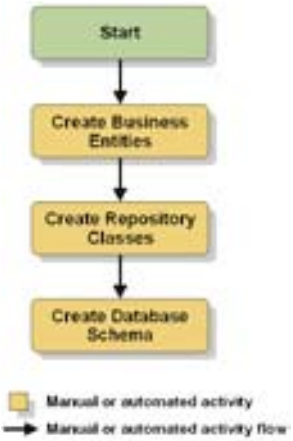
den is dat de technologie die je kiest voor de implementatie van de repositories zo veel mogelijk gescheiden moet zijn van jouw domeinmodel. (Dan kun je later gemakkelijker over naar Entity Framework bijvoorbeeld). Een nadeel van NHibernate is dat XML-configuratiefiles moeten worden bijgehouden. Conceptueel weer een prima gedachte, maar in de praktijk erg vervelend. Dat moeten de ontwikkelaars van ActiveRecord (www.castleproject.org) ook gedacht hebben. Zij hebben een wrapper om NHibernate heen gemaakt die het mogelijk maakt jouw businessentiteiten te decoreren met attributen. (Als je naar twee verschillende bestaande databases moet mappen, dan is dit niet toepasbaar). Deze attributen zijn de vervangers van de xml-files. De naam ActiveRecord zegt het al, ze implementeren het *ActiveRecord*-pattern. Dit pattern is vooral geschikt voor simpele domeinen. We hadden als doel een complex domein. Gelukkig is dit simpel op te lossen zonder verlies aan functionaliteit zoals we zullen zien. Hoe ziet de code er nu uit. We hebben ons domeinmodel: Customer. Zie codevoorbeeld 1 voor de definitie van de Business Entity Customer.
Origineel dient de Customer-class ook over te erven van de ActiveRecordBase<T> of ActiveRecordBase-class, zodat de persistentiefunctionaliteit beschikbaar komt via de klasse zelf zoals het ActiveRecord-pattern ook beschrijft. Door deze overerving weg te laten en door de repository te laten plaatsvinden, kiezen we voor het Repository-pattern om complexe domeinen aan te kunnen. (We hebben ActiveRecord-methodes van static naar instance methods aangepast overloading mogelijk te maken, we gebruiken de laatste versie van ActiveRecord uit hun SVN voor support voor generics). Hier zien we de ActiveRecord-attributen op de Customer-class. De attributen hebben argumenten, maar als er geen afwijking tussen de namen van de objecten en de databasetabellen is, dan wordt de naam en het type afgeleid. Vervolgens dienen we per business entity een repository aan maken; zie codevoorbeeld 2. (In de resource access-layer van afbeelding 1).

```
[ActiveRecord("Customer")]
 public partial class Customer
{
  private string _Name;
  [Property("Name", ColumnType="String")]
  public virtual string Name
  {
    get
    {
      return this._Name;
    }
    set
    {
      this._Name = value;
    }
  }
}
```

Codevoorbeeld 1. Definitie van Customer Business Entity.

```
/// <summary>
/// Repository for the Customer entity.
/// </summary>
public class CustomerRepository : ActiveRecordBase<Customer>
{
}
```

Codevoorbeeld 2. Definitie van de Customer Repository



Afbeelding 4. Domain Driven-design werkwijze

Deze repository heeft via overerving toegang tot Save Delete, Find-methodes waarbij ook de query (HQL) functionaliteit van NHibernate tot onze beschikking staat. Elke entity krijgt zijn eigen repository die overerft van een generic repository, zodat er altijd functionaliteit aan toegevoegd kan worden.
Het is ook noodzakelijk voor objecten die zelf niet persistent zijn, maar samengesteld zijn uit andere entiteiten; bijvoorbeeld Order bestaat uit Order en Orderlines. Er zal alleen een OrderRepository bestaan die de verantwoordelijkheid heeft om een Order (altijd bestaande uit een Order met OrderLines) op te slaan, op te halen, et cetera. De aanroep van de action in codevoorbeeld 3 (Architecture Web Service Software Factory) ziet er uit zoals in codevoorbeeld 3.
Het is nu mogelijk om op basis van de attributen in het domeinmodel een database te genereren. Ook is het mogelijk een SQL-file te genereren (Zie ActiveRecord-documentatie). Een mooi startpunt. Mocht je echter geen controle over de database hebben, dan moet je de mapping (op basis van attributen) laten aansluiten. Natuurlijk is het ook mogelijk om relaties en overerving te gebruiken met ActiveRecord. Dan moet je echter wel de nodige kennis hebben van het gebruik van de attributen.

DSL

Tegenwoordig hebben we met DSL in Visual Studio de mogelijkheid om bepaalde functionaliteit grafisch weer te geven. Het gratis product ActiveWriter [ACTIW] is een dergelijke DSL die is geënt op het gebruik van ActiveRecord. We kunnen dus nu grafisch ons domeinmodel maken zoals in afbeelding 3 is te zien. ActiveWriter genereert voor ons de Business Entity-classes met de attributen voor de properties en de relaties. Ook bevat het de mogelijkheid tot overerving van een eigen base-class. Deze DSL is een 'one-way' tool, van visueel model naar code en niet vice versa. De code die we toevoegen, doen we dan ook in een partial class waarin de extra functionaliteit zit (behavior). De wijziging zoals we in het begin van het artikel zagen, bestaat nu uit de volgende stappen: voeg

```
public class GetCustomerAction
{
  public Execute(int customerId)
  {
    CustomerRepository repository = new CustomerRepository();
    return repository.FindByPrimaryKey(customerId);
  }
}
```

Codevoorbeeld 3.

```
/// <summary>
/// Behavior to set session handling on operations.
/// </summary>
[AttributeUsage(AttributeTargets.Method,
Inherited = false, AllowMultiple = false)]
public sealed class SessionOperationBehaviorAttribute : Attribute, IOpe
rationBehavior
{
   #region IOperationBehavior Members

   public void AddBindingParameters(OperationDescription
   operationDescription, System.ServiceModel.Channels.
   BindingParameterCollection bindingParameters)
   {
      return;
   }

   public void ApplyClientBehavior(OperationDescription
   operationDescription, System.ServiceModel.Dispatcher.
   ClientOperation clientOperation)
   {
      throw new NotImplementedException("this operation behavior
      is meant for the server side");
   }

   public void ApplyDispatchBehavior(OperationDescription
   operationDescription,  System.ServiceModel.Dispatcher.
   DispatchOperation dispatchOperation)
   {
      dispatchOperation.ParameterInspectors.Add(new SessionOperationHandler());
   }

   public void Validate(OperationDescription operationDescription)
   {
      return;
   }

   #endregion
}

class SessionOperationHandler : IParameterInspector
{
   #region IParameterInspector Members
   /// <summary>
   /// After the call.
   /// </summary>
   /// <param name="operationName">Name of the operation.</param>
   /// <param name="outputs">The outputs.</param>
   /// <param name="returnValue">The return value.</param>
   /// <param name="correlationState">State of the correlation.</param>
   /// <remarks> Flushed an ActiveRecord Session.</remarks>
   public void AfterCall(string operationName, object[] outputs,
   object returnValue, object  correlationState)
   {
      ISessionScope scope = SessionScope.Current;
      if (scope != null)
      {
         scope.Flush();
      }
   }

   /// <summary>
   /// </summary>
   /// Before the call.
   /// <param name="operationName">Name of the operation.</param>
   /// <param name="inputs">The inputs.</param>
   /// <returns> Starts an ActiveRecord Session.</returns>
   public object BeforeCall(string operationName, object[] inputs)
   {
      SessionScope scope = new SessionScope();
      return null;
   }

   #endregion
}
```

Codevoorbeeld 5. Custom OperationBehavior

property toe aan de Customer-entity in de DSL. Sla de DSL op. Op dit moment worden de classes geschreven met de attributen. Compileer het project. Omdat wij de database opnieuw generen (de databasescripts zijn immers ook codefiles met een bepaalde versie), is de wijziging meteen verwerkt in de database. Alle logica om op te slaan, te wijzigen, te zoeken, et cetera, wordt runtime bepaald en behoeft dus geen aanpassing. De procedure is veel korter en veel minder foutgevoelig. Deze werkwijze is te zien in afbeelding 4.

```
/// <summary>
/// custom ServiceHostFactory to do activerecord initialization.
/// </summary>
public class CustomHostFactory : ServiceHostFactory
{
   protected override System.ServiceModel.ServiceHost
CreateServiceHost(Type serviceType, Uri[]  baseAddresses)
   {
      ServiceHost serviceHost = base.CreateServiceHost(
      serviceType,baseAddresses);
      serviceHost.Opening += new EventHandler(serviceHost_Opening);
      return serviceHost;
   }

   /// <summary>
   /// Handles the Opening event of the serviceHost control.
   /// </summary>
   /// <param name="sender">The source of the event.</param>
   /// <param name="e">The <see cref="System.EventArgs"/>
   /// instance containing the event data.</param>
   /// <remarks>when opening the service ActiveRecord has to be
   /// initialized.</remarks>
   void serviceHost_Opening(object sender, EventArgs e)
   {
      IConfigurationSource config = ActiveRecordSectionHandler.Instance;
        Assembly[]activeRecordAssembly = Assembly.Load("
        ReferenceImplemention.BusinessEntities") ();
        ActiveRecordStarter.Initialize(activeRecordAssembly, config);
   }
}
```

Codevoorbeeld 4. ActiveRecord-initialisatie via custom ServiceHostFactory

Integratie Web Service Software Factory en ActiveRecord

ActiveRecord dient voor gebruik geïnitialiseerd te worden. Ook dient er, indien gebruikgemaakt wordt van Lazy Load-functionaliteit, eerst een zogenaamde sessie geïnitialiseerd te worden. De uitbreidbaarheid van Windows Communication Foundation is hier prima voor te gebruiken. Door een custom ServiceHostFactory te maken, kunnen we bij het openen van de service, ActiveRecord initialiseren, zie codevoorbeeld 4. Voor meer informatie over deze extensibility van WCF zie het artikel "WCF servicemodel internals & extensibility" in .NET magazine #16.

Sessiemanagement wordt opgelost door een operation behavior te maken. Deze plaatsen we door een attribuut op de operatie die daardoor binnen een ActiveRecord-sessie zal draaien.

Abstracter en toch beter begrijpbaar

Door uit te gaan van het domain-model en de mapping naar de database te laten regelen door tooling, kan er op een hoger abstractieniveau, Domain Driven, geprogrammeerd worden. Het domeinmodel blijft begrijpbaar voor de business en de ontwikkelaar. Wijzigingen kunnen sneller en met minder fouten doorgevoerd worden.

Rudi van den Belt is senior software developer bij Reasult, www.reasult.com, hij is te bereiken op rvandenbelt@reasult.com.

Referenties

[ACTIVERECORD], www.castleproject.org

[ACTIW], http://www.altinoren.com/activewriter/

[ARJRN] Michael Pizzo. Application-Oriented Data Model. The Architecture Journal, Journal 12, www.architecturejournal.net.

[Fow03] M.Fowler. Patterns of Enterpise Application Architecture. Addison-Wesley, 2003.

[NILSON] J.Nilsson. Applying Domain-Driven Design and Patterns. Addison-Wesley. 2006.

[NHIBERNATE] www.nhibernate.org

[WSSF] Web Service Software Factory, http://msdn2.microsoft.com/en-us/library/aa480534.aspx