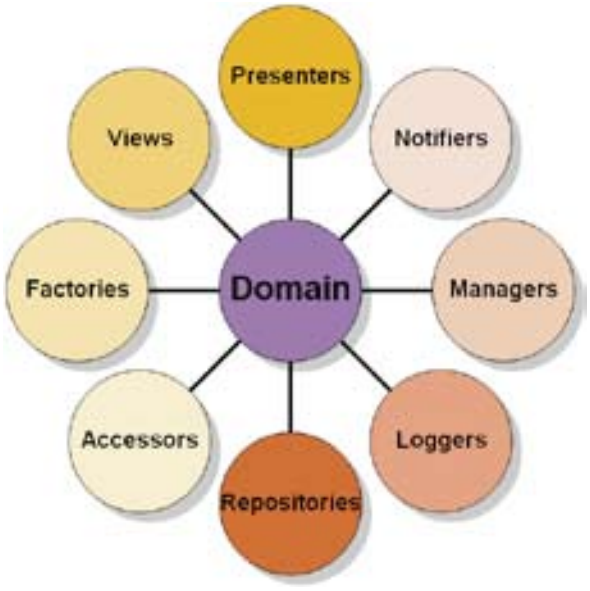
Arjan Keene

Domain Driven Design met attributes-smaak

DECLARATIEVE DECORATIE VAN EEN DOMEINMODEL

Domain Driven Design is een aanpak waarin een domeinmodel centraal wordt gesteld en alle ondersteunende services van dit model gebruikmaken. In dit artikel wordt een softwarearchitectuur voor domeinmodellen beschreven, waarin veelvuldig wordt geleund op custom attributes, reflectie en onderdelen van het componentmodel.

Domain Driven Design (DDD) geniet een groeiende belangstelling. De belangrijkste evangelisten hiervan zijn Eric Evans en Jimmy Nilsson, met Martin Fowler als prominente co-promotor. Domain Driven Design is in feite niet iets nieuws, het wordt ook niet gepositioneerd als een nieuwe methodologie of als een andere programmeerwijze, ook voegt het geen nieuwe taalstructuren toe. Het biedt wel een manier van denken over softwareprojecten en een structurele aanpak van softwareontwerp waarin het onderhavige domein en de domeinlogica centraal wordt gepositioneerd. Common sense, feitelijk, want het domein – kennisdomein, klantdomein, bedrijfsdomein, 'domain of discourse' of hoe je het ook wilt noemen - daar gaat het in essentie uiteindelijk om. Al het andere dient hier als satellieten omheen te cirkelen en diensten te leveren die dit domein ontsluiten. Domain Driven Design fungeert hier als kapstok, maar de terminologie en voorbeelden in dit artikel zijn niet naar de letter in overeenstemming met deze aanpak. Ook zijn er overeenkomsten met Domain Specific Languages en komen er zaken naar voren die sinds enige tijd deel uitmaken van de Enterprise Library. Het gaat echter niet om verschillen of overeenkomsten, maar meer om een algemene richting. De architectuur die in dit artikel wordt beschreven, is in projectverband succesvol toegepast.



Afbeelding 1. Domeinmodel

Principes voor een domeinmodel

Er zijn meer principes en uitgangspunten waaraan een domeinmodel moet voldoen. We noemen voor ons doel de drie meest relevante. Een eerste principe is dat je een domeinmodel onderbrengt in een verzameling classes die bij elkaar horen en waarin businesslogica een plaats heeft. Daarbij luistert het nauw hoe je de entiteiten modelleert, welke properties je waarbij onderbrengt, hoe en waarom je subclasses aanbrengt, enzovoort. Anders gezegd, het is geen open deur om te benadrukken dat een domeinmodel zeer gedegen gemodelleerd moet worden. Een tweede uitgangspunt is om het domeinmodel voldoende zuiver te houden, zodat het principe van 'separation of concerns' tot zijn recht komt. Persistentie bijvoorbeeld, is op zichzelf geen onderdeel van het domeinmodel. Logging evenmin. Object factories, repositories en andere dienstverlenende managers horen er ook niet in thuis. En dit gaat ook op voor bijvoorbeeld views, presenters, controllers, notifiers en subscribers; zie afbeelding 1. Het domeinmodel wordt door al deze ondersteunende services gebruikt, maar ze maken er geen deel van uit. Ten derde is het van groot belang dat een domeinmodel vervangbaar is door een ander model. Het moet ook mogelijk zijn meer modellen naast elkaar te ondersteunen. De softwarearchitectuur moet dit toestaan, zodanig dat een vervanging of toevoeging van domein-assemblies in runtime mogelijk is. Bij voorkeur is een domeinmodel dan ook ondergebracht in één assembly.

Gemeenschappelijke 'kelder'

Om aan bovengenoemde principes te kunnen voldoen, hebben alle entity-classes in het domeinmodel een gemeenschappelijke base class, die we hier BasementObject noemen; zie afbeelding 2. Deze base class en de hiermee samenhangende types worden ondergebracht in een assembly separaat van het domeinmodel. Deze assembly heeft geen referenties naar het domeinmodel, en dus geen inhoudelijke kennis van de entity-classes van een domeinmodel. Wel bevat het voorzieningen voor objectparameters (properties), attributes, validatie, en een aantal managers voor objecten, repositories, subscriptions, et cetera. Deze basement-assembly is in feite een mini-framework voor een domeinmodel, waarin met reflectie informatie aan een domeinmodel wordt onttrokken. De genoemde zaken zullen we nu in meer detail bespreken.

Properties zonder eigenschappen

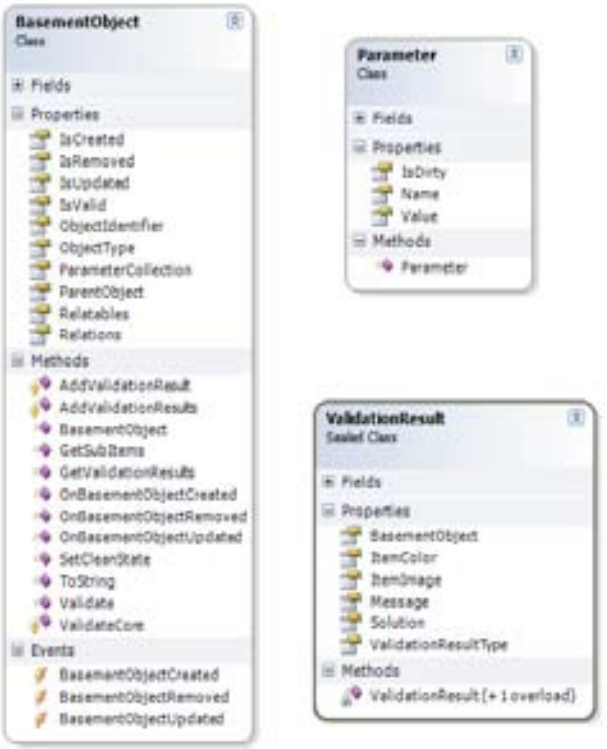
De properties van objecten van het domeinmodel moeten gepersisteerd kunnen worden. Ook moet er een generieke manier zijn om deze properties door de base-assembly te laten lezen, bijvoorbeeld om validatie te kunnen toepassen. Daarom heeft de BasementObject-class een ParameterCollection die instanties van

het type Parameter bevat. Deze Parameter heeft een name-value-pair-samenstelling, waarbij een value-type als string wordt weggeschreven. De base class heeft dus op het eerste gezicht geen enkele kennis van wat er met die properties wordt bedoeld, kent zelfs niet eens de datatypen ervan. In codevoorbeeld 1 is de opbouw van een klasse in het domeinmodel weergegeven, in dit voorbeeld een klasse 'Server'. De constructor van de klasse voegt in de vorm van string-literals de properties toe aan de ParameterCollection van de instantie. In de getters en setters van de properties worden deze literals gebruikt, en nergens anders. Door de encapsulatie is het gebruik van de properties typesafe, en de vereenvoudigde 'verstringing' van het name-value-pair in de Parameter wordt hier verrijkt met een expliciet data-type in de return-type van de property. Een voorbeeld is de enum ServerType als return-type van de gelijknamige property.

Attributes: domeinkennis tussen blokhaakjes

De hoeksteen van een domeinmodel zoals hier beschreven, is het rijke gebruik van attributes. Alle meta-informatie over entity-classes in het domeinmodel wordt uitgedrukt met behulp van attribute-decoratie. Het voordeel van attributes is dat ze een declaratieve manier bieden om definities en andere meta-informatie vast te leggen. Op deze manier is ook businesslogica op te nemen in het bestek van een domeinmodel. Hierdoor wordt het gewenste gedrag van objecten vastgelegd, zonder dat het gebruik van deze declaraties wordt gespecificeerd. Dit biedt flexibiliteit om de attributes op verschillende manieren te kunnen gebruiken voor diverse doeleinden. Bovendien waarborgt deze aanpak het principe van 'separation of concerns'.

In de eerste plaats worden er standaard attributes gebruikt, zoals op class-niveau naast Serializable bijvoorbeeld DefaultProperty, zoals in codevoorbeeld 1 is te zien. Op property-niveau kunnen Category, Description, Browsable, DefaultValue, RefreshProperties, DisplayName, enzovoort, worden gebruikt. Ook bieden de System.ComponentModel- en System.Drawing.Design-namespaces diverse attributes als TypeConverter en (UIType)Editor, waar custom implementaties voor geschreven kunnen worden om de uitdrukkingskracht ervan te benutten. Gedrag dat niet standaard beschikbaar is in bestaande attributes, kunnen we beschrijven in custom attributes. Voorbeelden hiervan zijn:



Afbeelding 2. BasementObject

```
/// <summary>
/// This class contains the definition of a server.
/// </summary>
[Serializable, DisplayName("Cyber Pizza"), DefaultProperty("HostName")]
[Relatable(BasementObjectType = typeof(Reservation), Cardinality = 0,
  CardinalityType = CardinalityType.CardinalityOrMore), Linkable]
public class Server : BasementObject
{
  const ServerType _defaultServerType = Model.ServerType.WEBSVR;

  public Server()
  {
    ParameterCollection.Add(new Parameter(this, "ServerType"));
    ParameterCollection.Add(new Parameter(this, "HostName"));
    ParameterCollection.Add(new Parameter(this, "IPAddress"));
  }

  [Category("Identification")]
  [Mandatory]
  [Description("The hostname of the server.")]
  [RegularExpression(RegexExpressions.HostNameExclusionFormat,
    RegexMatchType.MatchExclusions, RegexOptions.Compiled,
    RegexExpressions.HostNameExclusionFormatHelp)]
  [RefreshProperties(RefreshProperties.All)]
  [StringLength(Min = 4, Max = 40)]
  public string HostName
  {
    get { return ParameterCollection["HostName"].Value; }
    set { ParameterCollection["HostName"].Value = value; }
  }

  [Category("Identification")]
  [Description("Internet Protocol Address")]
  [RegularExpression(RegexExpressions.IPAddressFormat,
    RegexMatchType.MatchFormat, RegexOptions.Compiled,
    RegexExpressions.IPAddressFormatHelp)]
  public string IPAddress
  {
    get { return ParameterCollection["IPAddress"].Value; }
    set { ParameterCollection["IPAddress"].Value = value; }
  }

  [Category("Server Type"), DisplayName("Main Server Usage")]
  [DefaultValue(_defaultServerType)]
  [TypeConverter(typeof(EnumDescriptionConverter))]
  [Editor(typeof(ServerTypeEditor), typeof(UITypeEditor))]
  public ServerType ServerType
  {
    get
    {
      string val = ParameterCollection["ServerType"].Value;
      if (val != null && Enum.IsDefined(typeof(ServerType), val))
        return (ServerType)Enum.Parse(typeof(ServerType), val);
      else
        return _defaultServerType;
    }
    set
    {
      ParameterCollection["ServerType"].Value = value.ToString();
    }
  }

  /// <summary>
  /// ValidateCore
  /// </summary>
  protected override void ValidateCore()
  {
    // specific business logic validation here
    base.ValidateCore();
  }

  public override string ToString()
  {
    return (!String.IsNullOrEmpty(HostName) ? HostName :
        base.ToString());
  }
}
```

Codevoorbeeld 1.

- Mandatory. Indicatie dat een property verplicht is, eventueel afhankelijk van de waarde van andere properties.
- ConditionalDependency. Hierbij is de waarde van de ene property afhankelijk van de (drempel)waarde van een andere property.
- Range. Het waardebereik van een property.
- StringLength. Het lengtebereik van een stringwaarde.
- UnitOfMeasure. De meeteenheid van een property.
- RegularExpression. Een property moet voldoen aan een Regex-pattern, of een inverse variant hiervan om bijvoorbeeld niet-toegestane karakters uit te sluiten.
- Sequence. Properties moeten in een specifieke volgorde worden behandeld of getoond. Ook toepasbaar op enum-fields om de volgorde te reguleren.
- Column. Mapping op een Identifier en of Name van een veld in een databasetabel.

De mogelijkheden zijn grenzeloos en er is een grote variëteit aan AttributeTarget-types. Ook het toepassen op classes biedt expressiemogelijkheden, voor persistentie bijvoorbeeld een Table-attribute dat een mapping op een databasetabel aangeeft, in combinatie met Column-attributes voor properties. Voor mappings die complexer zijn en voor bestaande datastores zal dit niet altijd kunnen en komt een O/R-mapper in beeld. Linq kan hier uiteraard goed benut worden, of een provider-gebaseerd bindingmodel. Maar let wel, databaseschema's dienen ondergeschikt te zijn aan domeinmodellen, en niet andersom.

Relaties: (no) strings attached

Een andere toepassing die ook in het voorbeeld van de Server-class is opgenomen, betreft relaties tussen objecten. Custom attributes die hier een rol spelen zijn bijvoorbeeld:

- Relatable. Welke classes zijn toegestaan in relaties? Relevante parameters zijn bijvoorbeeld cardinaliteit (als integer), bijbehorende qualifier (exact, meer dan, minder dan), de soort relatie; de richting (child, parent, association, enzovoort).
- Copyable. Mag er een kopie (create like/clone) gemaakt worden van de instantie?
- Linkable. Zijn instanties van de class als referentie te 'linken'? Dit is feitelijk een domeinmodelweergave van een foreign key-relatie, maar met een veel lossere koppeling en niet beperkt tot het strikte relationele model.

Codevoorbeeld 2 toont de property BasementObject.Relatables die de types retourneert die in RelatableAttribute-declaraties zijn opgenomen. De toegestane types waar objectinstanties relaties mee kunnen hebben, is hiermee op een eenduidige manier te sturen bij het aanbrengen van relaties. Hier zijn veel toepassingen voor. Een voorbeeld is het generiek kunnen samenstellen van contextmenu's om de toegestane types aan te bieden die aan een object kunnen worden gekoppeld.

```
[Browsable(false)]
public Collection<Type> Relatables
{
  get
  {
    Collection<Type> relatableTypes = new Collection<Type>();
    if (GetType().IsDefined(typeof(RelatableAttribute), true))
    {
      RelatableAttribute[] attributes =
        GetType().GetCustomAttributes(
          typeof(RelatableAttribute), true)
            as RelatableAttribute[];

      if (attributes != null)
      {
        foreach (RelatableAttribute attribute in attributes)
        {
          relatableTypes.Add(attribute.BasementObjectType);
        }
      }
    }
    return relatableTypes;
  }
}
```

Codevoorbeeld 2.

Waar de property Relatables de te relateren types betreft, geeft de property Relations de instanties van relaties weer. Omdat we de declaratie van relaties uit de classes hebben gehaald, is het niet meer mogelijk tegen dit model te coderen met een aanroep als:

```
Customer.Addresses.Add(new Address(…)).
```

Dit is op het eerste gezicht wellicht een nadeel, maar er staat tegenover dat er een veel rijkere expressiekracht voor in de plaats komt als deze 'geëxternaliseerde' relaties met reflectie worden ontsloten. Bovendien dwingt dit principe impliciet het gebruik van object-factories af. In codevoorbeeld 3 staat een voorbeeld van een ObjectManager.CreateObject-method. De customer-snippet hierboven zou bij rechtstreeks gebruik in code in dit geval als volgt worden:

```
if (customer.Relatables.Contains(typeof(Address)))
  customer.Relations.Add(new Address(...));
```

Het simpele 'Contains' speelt hier feitelijk de rol van een 'IsDefined'. Om ook rekening te houden met verschillende soorten relaties moet hiervoor een genuanceerde implementatie in de plaats komen.

Validation Pattern

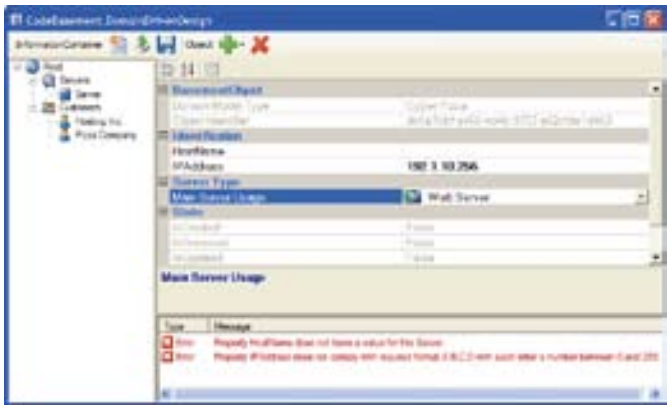
Gegeven het voorgaande is een generieke validatie zeer eenvoudig te implementeren, met wat we hier het Validation Pattern kunnen noemen. Het BasementObject heeft een Validate-method, waarin een ValidationResults-collection wordt gevuld. Dit gebeurt deels volledig generiek door validators aan te roepen die met reflectie de attributes afhandelen met betrekking tot classes, properties, relaties, enzovoort. Daarna wordt de ValidateCore-method aangeroepen, die overridable is in afgeleide domeinmodel-classes; zie codevoorbeeld 4.

```
public static BasementObject CreateObject(
          Type type,
          BasementObject parentObject)
{
  BasementObject newObject =
    Activator.CreateInstance(type) as BasementObject;

  if (newObject != null)
  {
    newObject.ObjectIdentifier = Guid.NewGuid();
    newObject.OnBasementObjectCreated();
  }

  if (parentObject != null)
  {
    if (parentObject.Relatables.Contains(type))
      parentObject.Relations.Add(newObject);
    else
      throw new InvalidOperationException(
        String.Format("Type {0} not relatable to object {1}",
          type, parentObject));
  }
  return newObject;
}
```

Codevoorbeeld 3.



Afbeelding 3. Voorbeeld van een typische developer/designer-UI

De ValidateCore is dan ook de plek in een domeinmodel-class waar specifiek gecodeerde validatielogica kan worden geïmplementeerd. De resultaten worden wederom aan de ValidationResults van de instantie toegevoegd. Result-instanties kunnen worden uitgerust met een kenmerk (error, warning, enzovoort.) en eventueel bijbehorende visualisatie. Dit is een krachtig patroon voor zowel preventieve als correctieve diagnostiek van domeinmodelobjecten.

PropertyGrid en custom designers

Attributes hebben een bijkomend voordeel dat designer-controls 'attribute-aware' zijn. Een out-of-the-box voorbeeld hiervan is natuurlijk het PropertyGrid. In algemene zin is het gedrag dat in attributes is vastgelegd elegant te benutten door controls en controllers, en uit te bouwen met eigen implementaties van diverse descriptors. De diamanten die verborgen zitten in het System.ComponentModel, zoals (custom) type descriptors, designers, property descriptors en type description providers, vergen heel wat meer arbeid om te ontginnen en verdienen een behandeling op zich. Een fraai voorbeeld hiervan is het SmartPropertyGrid van VisualHint (zie referenties).
Nu is het PropertyGrid niet de eerste kandidaat voor een control waar je aan denkt voor presentaties. Toch leert de ervaring dat deze typische developer/designer-UI ook door eindgebruikers wordt gewaardeerd. Het biedt namelijk wel de mogelijkheid een zeer consistente look-and-feel te garanderen, al is de standaard implementatie wel wat Spartaans; zie afbeelding 3. Aan de andere kant, niet iedereen zit te wachten op roterende buttons en gerenderde transparante visuals als dit geen core value toevoegt aan het onderhavige domein. Met behulp van type-converters en type-editors is er al voor een deel tegemoet te komen aan een betere presentatie, en dit mes snijdt aan twee kanten.

```
/// <summary>
/// Validates this instance.
/// </summary>
public void Validate()
{
  if (_validationResults != null)
    _validationResults.Clear();
  else
    _validationResults = new ValidationResultCollection();
  AddValidationResults(new ObjectRelationValidator().Validate(this));
  AddValidationResults(new ObjectTypeValidator().Validate(this));
  AddValidationResults(new ObjectPropertyValidator().Validate(this));
  ValidateCore();
}

/// <summary>
/// ValidateCore : overridable in derived classes
/// </summary>
protected virtual void ValidateCore()
{
}
```

Codevoorbeeld 4.

```
internal class ServerTypeEditor : UITypeEditor
{
  public override bool GetPaintValueSupported(
          ITypeDescriptorContext context)
  {
    return true;
  }

  public static string GetIdentifier(Enum value)
  {
    FieldInfo fi = value.GetType().GetField(value.ToString());
    ImageAttribute[] attributes =
      (ImageAttribute[])fi.GetCustomAttributes(
      typeof(ImageAttribute), false);
    return (attributes.Length > 0) ?
        attributes[0].ImageIdentifier : value.ToString();
  }

  public override void PaintValue(PaintValueEventArgs pe)
  {
    string id = GetIdentifier((Enum)pe.Value);

    Bitmap img = (Bitmap)ModelImages.ImageList.
    Images[ModelImages.GetImageIndex(id)];

    if (img != null)
      {
        pe.Graphics.DrawImage(img, pe.Bounds);
        img.Dispose();
      }
  }
}
```

Codevoorbeeld 5.

In codevoorbeeld 1 is bij de property ServerType een EnumDescriptionConverter opgenomen als type-converter voor de property. Het resultaat hiervan is dat de waarde van een enum-field bij de presentatie wordt vervangen door een omschrijving die in een Description-attribute is opgenomen. En andersom. Dit houdt dus in dat conversie van 'codes' naar omschrijvingen automatisch wordt opgelost; typisch zullen de 'codes' worden gepersisteerd.
Ook is een UITypeEditor opgenomen (zie codevoorbeeld 5) die de property ServerType van een waardeafhankelijk image voorziet. En uiteraard is de referentie aan de image in dit geval als custom attribute op de velden van de enum gedecoreerd.

Ten slotte

Met bovenstaande toevoeging is duidelijk dat we sommige presentatieaspecten in het domeinmodel als onderdeel van het model beschouwen, en ook ankers opnemen die je normaliter wellicht in een datalaag zou verwachten. Maar dat is juist een belangrijke eigenschap van een domeinmodel als hier beschreven: het voorziet in een rijk objectmodel, waarin validaties en business-rules zijn opgenomen, en waarin alle meta-informatie over het domein een plek heeft. Het domeinmodel wordt met reflectie ontsloten door een onderliggende 'basement assembly' dat als fundament dient.
Rest mij u een smakelijke maaltijd te wensen, met als toetje een gelaagde vlaflip gedecoreerd met fantasierijke attributen.

Arjan Keene is freelance software- en information engineer. Zijn specialismen zijn domeinmodellering en de ontwikkeling van object model frameworks. Hij is bereikbaar op apkeene@cs.com.

Referenties

Domain Driven Design: www.domaindrivendesign.org
Type converters/editors: www.codeproject.com
Regex patterns: www.regexlib.com
Smart PropertyGrid: www.visualhint.com