Sander Hoogendoorn

Value Objects implementeren

QUICK WINS IN DOMAIN DRIVEN DEVELOPMENT

Regelmatig beland ik in gesprekken met klanten over het bedrijfsdomein en de bedrijfslogica van hun applicaties. Steevast rollen de domeinobjecten en hun properties over tafel en worden er – vaak tussen de regels door – allerlei wensen en eisen aan deze objecten gesteld. “De klant kan een boek natuurlijk ook vinden door het ISBN in te typen” of “we registeren de website van de klant ook wel eens door gewoon de url over te nemen uit de browser”. Als ik niet allang ben afgehaakt, spits ik nu toch even de oren. Hier is iets bijzonders aan de hand.

Alhoewel in bovenstaand voorbeeld de properties **Isbn** van het domeinobject **Book** en **Website** van **Customer** op het eerste gezicht strings lijken te zijn, weten we eigenlijk van zowel **Isbn** als **Url** veel meer. Er moet in beide gevallen aan allerhande voorwaarden worden voldaan. Niet alle strings vertegenwoordigen een geldige ISBN of een geldige url. Hoewel deze twee voorbeelden overbekend zijn, komen er in de de spelonken van bedrijfsdomeinen veel meer van dit soort typen voor in, die je al gauw over het hoofd ziet als je het specifieke domein van de klant niet goed kent. Voorbeelden zijn een werknemernummer dat aan iedere nieuwe medewerker wordt toegekend of een voorgetypeerd wachtwoord. Toen ik onlangs in Amsterdam een training over UML verzorgde, verklaarde een softwarearchitect van een grote financiële instelling dat daar vrijwel nooit base types zoals **int**, **string** of **DateTime** worden gebruikt, maar dat voor bijna alle properties dit soort speciale typen zijn gedefinieerd. Als reden voerde hij aan dat de kennis over vrijwel alle properties van domeinobjecten verder reikt dan dat ze een string of een getal zijn.

Gelijkheid

Typen zoals **Isbn** en **Postcode** worden in de literatuur wel *value objects* genoemd, zoals in het werk van Martin Fowler of Eric Evans. Deze value-objecten onderscheiden zich van domeinobjecten zoals **Customer** of **Product**. Het nut van value-objecten moge evident zijn; alle logica die bij **Postcode** of **Sofinummer** hoort, bevindt zich maar op één plek in de code van de applicatie, hoe vaak ze ook worden gebruikt. Hoewel **Customer** en **Url** duidelijke voorbeelden zijn, is het onderscheid tussen value-objecten en domeinobjecten in de praktijk niet altijd eenvoudig. Er is technisch gezien geen strikt onderscheid. Beide typen hebben properties en methoden, en zijn gelijkaardig te implementeren. In de regel wordt wel gesteld dat domeinobjecten identiteit hebben en value-objecten niet. Bovendien is het zo dat value-objecten meestal klein blijven, met één of enkele properties en domeinobjecten groeien afhankelijk van het aantal properties dat nodig is om een **Customer** of **Product** te typeren. Maar daar schieten we nog niet veel mee op.

Het belangrijkste verschil tussen domeinobjecten en value-objecten is de manier waarop ze omgaan met gelijkheid. Een domeinobject bezit identiteit, ongeacht hoe deze is geïmplementeerd. Het kan zijn dat wordt verwezen naar een object in het geheugen, maar vaker wordt gerefereerd aan een **Id** property, zeker wanneer het domeinobject nog moeten worden gepersisteerd. Bij een value-object wordt gelijkheid echter uitsluitend gevalideerd op basis van de waarden van de properties. Twee verschillende instanties van een value-object **Email** zijn gelijk als de waarden van hun properties overeenkomen. In codevoorbeeld 1 is een deel **Email** geïmplementeerd.

```
private string value;

public Email(string address)
{
   value = null;

   if (string.IsNullOrEmpty(address)) return;

   if (!IsValidEmail(address))
      throw new FormatException("{0} is not an email");

   value = address;
}
```

Codevoorbeeld 1.

In codevoorbeeld 2 worden twee instanties **first** en **second** van **Email** gecreëerd. Op basis van het feit dat beide instanties de waarde **aahoogendoorn@gmail. com** retourneren, worden ze als gelijk beschouwd.

Hoe implementeren?

Als ik een value-object realiseer in .NET doemt al snel het eerste keuzemoment op: implementeer ik mijn value-object als value type (een struct) of als reference type (een class)? Als ik Email implementeer als een struct zijn de instanties first en second uit codevoorbeeld 2 automatisch gelijk. Wanneer ik Email echter als een class bouw, zijn twee instanties alleen gelijk als ze verwijzen naar hetzelfde object in het geheugen. Wat zijn de afwegingen? Over het algemeen bouw ik een value-object als struct wanneer onderstaande geldt:

- Het value-object representeert een enkelvoudige waarde, zoals **Email**, **PostCode** of **Isbn**.
- De instanties van het value-object zijn ‘immutable’.
- De instanties van het value-object hebben een kleine footprint (in het ideale geval 16 bytes of minder).
- De instanties zijn een kort leven beschoren.

```
Email first = new Email ("aahoogendoorn@gmail.com");
Email second = new Email ("aahoogendoorn@gmail.com");

if (first.Equals(second)) MessageBox.Show ("Email addresses are equal");
```

Codevoorbeeld 2

- De instanties worden voornamelijk als property in andere objecten gebruikt of als argument of return-type van methodes.
- De instanties vereisen geen herhaaldelijke boxing en unboxing.

In alle andere gevallen is het verstandiger het value-object te implementeren als **class** en niet als **struct.** Met andere woorden: implementeer enkelvoudige value-objecten, zoals **PostCode** of **Isbn**, als **struct** en bouw complexere types, zoals **Money** (dat valuta en verschillende numerieke indelingen bevat) als **class.**

Recept

Voor het bouwen van een value-object hanteer ik graag het onderstaande recept:

- Bepaal welk value-object je nodig hebt. Meestal komt dit voort uit het domeinmodel. Geef het nieuwe value-object een geschikte naam, zoals **Email**, **Password** of **SubscriptionNumber.**
- Bepaal of het nieuwe type een value type of een reference type wordt.
- Bedenk hoe de interne waarde van het value-object het beste kan worden voorgesteld.
- Programmeer de vereiste validaties en controles om de geldigheid van nieuwe waarden te valideren.
- Bepaal of het value-object leeg of zelfs null mag zijn.
- Implementeer de juiste interfaces voor het value-object, zoals **IComparable,** en de generieke interfaces **IComparable<T>** om de instanties van het value-object te vergelijken. Implementeer **IEquatable<T>** om te bekijken of twee instanties gelijk zijn.
- Implementeer alle aanvullende accessors die voor het value-object vereist zijn, inclusief het **TryParse()** pattern.
- Zorg er voor dat het value-object immutable is.

Het value-object definiëren

Tijdens analyse en ontwerp van een project duiken allerlei value-objecten op, ongeacht of een agile of watervalmethodiek wordt toegepast. Mijn ervaring is dat de value-objecten vooral naar voren komen tijdens informele gesprekken met gebruikers, zoals de eerder genoemde voorbeelden **Isbn** en **Url.** Hoewel deze twee voorbeelden voor weinig complicaties zorgen, komen veel meer van deze typen voor, al herken je ze misschien niet direct als zodanig. Een leuk voorbeeld is dat ondanks dat ik al jaren met value-objecten werk, ik er pas kort geleden door een collega op werd geattendeerd dat persoonsnamen geen strings zijn. Veel tekens, zoals &, @ en #, zijn immers niet toegestaan. Waarom ben ik daar niet eerder op gekomen?

```
private static Regex Expression =
   new Regex (@"^(?=.-{13}$)\d{1,5}([-])\d{1,7}\l\d{1,6}\l(\d|X)$");

public static bool IsValidIsbn(string newvalue)
{
   return !Expression.IsMatch(newvalue) ;
}

public Isbn(string newvalue)
{
   value = null;

   if (string.IsNullOrEmpty (newvalue) ) return;

   if (IsValidIsbn(newvalue) )
   (
      throw new FormatException(String.Format("{0} is no ISBN",
      newvalue));
   }

   value = newvalue;
}
```

Codevoorbeeld 3.

Tip Geef je value-object een duidelijke en toekomstvaste naam die ook voor de klant duidelijk herkenbaar is.

Voor value-objecten is het nog essentiëler dan voor andere typen objecten om een geschikte naam te kiezen, omdat deze lichtgewichten gemakkelijk opnieuw zijn te gebruiken in andere, toekomstige projecten.

De interne waarde bepalen

Value-objecten worden vergeleken op basis van de interne waarde(n) die ze beschrijven, niet op basis van het feit dat ze naar eenzelfde adres wijzen. Stel daarom nauwkeurig vast welke base types je gebruikt voor het vastleggen van deze interne waarden. De ervaring leert dat de meeste value-objecten, zoals **Email**, **PostCode** en zelfs **SocialSecurityNumber** gemakkelijk zijn uit te drukken als **string**. Hoewel ze natuurlijk ook zijn uit te drukken als **object**, werk ik liever met **string**, omdat dit meer gemak biedt, vooral als ik twee instanties van mijn value-object moet vergelijken. Sommige value-objecten krijgen echter een ander type interne waarde, zoals **int** of **DateTime**. Zo ben ik weleens een value-object **DateTimeInPast** tegengekomen, waarvan het gedrag zich gemakkelijk laat raden. **DateTimeInPast** representeerde zijn interne waarde als **DateTime**. De klasse **DateTime** – zelf ook een value-object – maakt overigens weer gebruik van een **ulong** om de eigen interne waarde (in tikken) te beschrijven.

Tip Kenmerk de interne waarde van een value-object als **private**, zodat niemand je value-object van buitenaf kan beschadigen.

Geldige waarden valideren

Vervolgens moet worden geverifieerd of een nieuw toe te kennen waarde wel echt betrekking heeft op een bankrekeningnummer, url of e-mailadres. Afhankelijk van de context zijn er vaak verschillende aanpakken mogelijk. Het valideren van een Nederlandse postcode lijkt eenvoudig: de code bestaat uit een viercijferig getal, waarvan het eerste geen nul mag zijn, gevolgd door twee alfabetische tekens. Je kunt dit gemakkelijk checken met een regular expression. Maar let op: een regular expression is weliswaar eenvoudig, maar vertelt niet of de postcode die je aan de constructor doorgeeft ook een echte postcode is. De syntax van de expressie is misschien wel correct, maar je hebt nog geen zekerheid over of er wel een straat bestaat met deze postcode. Alleen indien je over een volledige postcodelijst beschikt, kun dit valideren.

```
public bool IsEmpty
{
   get { return string.IsNullOrEmpty(value); }
}
```

Codevoorbeeld 4.

```
public override bool Equals(object obj)
{
   if (obj == null) return false;

   if (obj is Email)
   {
      return Equals((Email) obj);
   }

   return false;
}
```

Codevoorbeeld 5.

```
public bool Equals(Email other)
{
   return (value == other.value);
}
```

Codevoorbeeld 6.

De wenselijkheid van kostbare validaties voor een value-object is niet altijd evident en is sterk afhankelijk van de context waarin het wordt gebruikt. Een value-object moet immers klein en snel zijn. Zo geredeneerd kom je al snel tot eenvoudige en kosteneffectieve validaties.
Wanneer je je aanpak hebt bepaald, kun je deze inbouwen in het value-object. In de meeste situaties kies ik ervoor een afzonderlijke methode aan de validatie te wijden. Hierdoor kan ik de validatie vanuit verschillende locaties in de code aanroepen en weet ik zeker dat de validatie altijd op dezelfde wijze plaatsvindt.

Tip Neem de validatiecode op in een afzonderlijke methode, zodat je deze vanuit verschillende locaties kunt aanroepen, wellicht zelfs buiten het value-object zelf.

In codevoorbeeld 3 voor het value-object **Isbn** doet een static method met de naam **IsValidIsbn()** het zware werk. In dit voorbeeld wordt deze method aangeroepen vanuit de constructor voor **Isbn**. Doordat ik de methode **public** heb gemaakt, is ie ook van buiten **Isbn** benaderbaar.

Mag het value-object leeg zijn?

Een extra punt van overweging is de vraag of lege waarden, of misschien zelfs null, zijn toegestaan. Neem als voorbeeld een domeinobject **ContactPerson** met de property **Email** (van het type **Email**). Dit type **Email** is geïmplementeerd als een value-object. De vraag is echter of we van alle contactpersonen de e-mailadressen kennen. Waarschijnlijk niet. Hier is het dus nodig ook lege waarden voor **Email** toe te staan.

Tip Bepaal of lege waarden zijn toegestaan voor het value-object. Gebruik van een lege waarde verwijst meestal naar onbekende waarden, zoals bij nullable types.

In het codevoorbeeld 3 voor **Isbn** is een lege waarde toegestaan. Bovendien wordt de interne waarde standaard ingesteld op **null**. Als nu een ongeldige waarde wordt doorgegeven aan de constructor, wordt er een exception gegooid. Een lege string is wel toegestaan. Hier gebruik ik de lege waarde vaak om aan te geven dat het ISBN van een boek nog onbekend is. Als het value-object lege waarden mag bevatten, moet je er rekening mee houden dat wordt gecontroleerd of een bepaalde instantie leeg is. Hiervoor implementeer ik in het value-object een eenvoudige property genaamd **IsEmpty,** die verifieert of het object daadwerkelijk leeg is, zoals in codevoorbeeld 4.

Gelijkheid

Instanties van een value-object worden als gelijk beschouwd als de waarden van hun desbetreffende gelijk zijn. Om hier achter te komen, maakt de standaardimplementatie van de **Equals()** method van **object** in het .NET Framework gebruik van reflection. Voor value-objecten kan dit kostbaar zijn. Je kunt dan het best een eigen methode **Equals()** implementeren, en zo de efficiëntie van het algoritme verhogen. Zie codevoorbeeld 5 voor **Email**.
Ter aanvulling kun je overwegen de generieke interface **IEquatable<T>** te implementeren, die equality-checks uitvoert voor het specifieke value-object. Het implementeren van deze interface voorkomt boxing en unboxing tijdens het vergelijken van twee instanties van het value-object. Merk op dat in codevoorbeeld 5 voor **Equals()** gebruikgemaakt wordt van deze generieke implementatie om de vergelijking uit te voeren. Wanneer je bovendien je value-object realiseert als **class**, zijn twee instanties ook gelijk wanneer ze naar dezelfde referentie wijzen. In dat geval kan het geen kwaad eerst de reference equality te valideren. Hiermee verbeter je de efficiëntie van je code nog verder, zoals in codevoorbeeld 6.

```
public static bool operator ==(Email i, Email j)
{
   return i.Equals(j);
}

public static bool operator !=(Url i, Url j)
{
   return !i.Equals(j);
}
```

Codevoorbeeld 7.

Bepaal of je ook de **==** en **!=** operators van het value-object wilt overschrijven, wanneer je **IEquatable<T>** implementeert. Als je dit doet, let er op dat beide implementaties verwijzen naar de generieke **Equals()** methode, zodat verschillende validaties dezelfde resultaten retourneren.

Tip Zorg dat alle equality-checks in een value-object gebruikmaken van één en dezelfde implementatie.

Pas overigens op dat je niet slechts één van deze operators overschrijft. Overschrijf of beide operators, of geen van beide, zoals in codevoorbeeld 7.
Wanneer je gelijkheid implementeert voor jouw value-object, moet je met een aantal factoren rekening houden:

Als je **object. Equals()** wilt overschrijven, moet je ook **GetHashCode()** overschrijven. Hierdoor wordt gewaarborgd dat twee instanties die als gelijk worden beschouwd, ook dezelfde hashcode retourneren. In de meeste gevallen volstaat het de hashcode van de interne waarde terug te geven.
Gebruik deze methodes niet om excepties te gooien. Als argumenten de waarde **null** hebben of van het verkeerde type zijn, laat de methodes dan gewoonweg de waarde **false** retourneren.
Implementeer de methoden van de interfaces niet met hun volledige naam (inclusief de naam van de interface). Doe je dit wel, dan leidt het vergelijken van instanties automatisch tot boxing en unboxing. Dit geldt overigens ook voor de comparison interfaces, die hierna worden beschreven.

Vergelijkingen

Sommige value-objecten moeten gesorteerd kunnen worden; andere niet. Ik kan me voorstellen dat **Money** en **PostCode** sorteerbaar zijn, terwijl dit voor **SocialSecurityNumber** en **Url** minder waarschijnlijk is. Hoewel het nooit helemaal zeker is dat een lijst van value-objecten niet hoeft te worden gesorteerd, is dit meestal af te leiden uit de context van het project. De beide interfaces **IComparable** en **IComparable<T>** die vergelijkingen van verschillende instanties mogelijk maken, definiëren slechts één enkele methode, met de naam **CompareTo()**. Beide methoden retourneren overigens een **int.** Deze kan de volgende waarden hebben:

```
public int CompareTo(object obj)
{
   if (obj == null)
      return 1;

   if (!(obj is Email))
   (
      throw new ArgumentException("Argument is not a email address");
   )

   return CompareTo((Email) obj);
}

public int CompareTo(Email other)
{
   return value.CompareTo(other.value);
}
```

Codevoorbeeld 8.

```
public DateTime AddMinutes(int months)
{
   ...
}

public DateTime AddMonths(int months)
{
   ...
}
```
Codevoorbeeld 9.

```
DateTime start;

if (!DateTime.TryParse(txtContractStartDate, out start))
{
   MessageBox.Show("Value for start date is not a valid date")
}
```
Codevoorbeeld 10.

```
public static bool TryParse(string s, out Isbn result)
{
   result = Empty;

   if (string.IsNullOrEmpty(s))
   {
      return true;
   }

   if (!IsValidIsbn(s))
   {
      return false;
   }

   result = new Isbn(s);
   return true;
}
```
Codevoorbeeld 11.

- Lager dan nul, als de huidige instantie kleiner is dan de instantie die als argument is meegegeven.
- Nul, als de huidige instantie en de als argument doorgegeven instantie gelijk zijn. Voor value-objecten betekent dit value equality.
- Groter dan nul, als de huidige instantie groter is dan het meegegeven argument.

Normaal gesproken komt het implementeren van een vergelijking voor een value-object er op neer dat je de interne waarden van twee instanties vergelijkt, zoals wordt getoond in codevoorbeeld 8 voor **Email.** In dit geval heb ik de generieke **CompareTo()**-method gebruikt om de vergelijking van de interne waarde **value** van **Email** uit te voeren. Ik heb hier de efficiëntie van de code iets vergroot door een null-check op te nemen in de niet-generieke **ComparoTo()**-method, hoewel de methode **CompareTo()** van **string** (**value** is geïmplementeerd als **string**) dit ook doet.

Als je beslist dat sorteren een vereiste is voor je value-object, moet je ook de volgende consequenties incalculeren:

Bepaal of je de comparison operators, zoals <, >, <= en >= wilt overschrijven. Vergeet niet dat ook deze implementaties betrekking hebben op de vergelijking van de interne waarde van je value-object, terwijl de equality operators al de gelijkheid hiervan valideren. Zorg ervoor dat al deze implementaties gebruikmaken van één en dezelfde methode. Meestal is dit de generieke implementatie van CompareTo().
Gebruik voor de methoden van de interfaces weer geen expliciete namen. Ook hier geldt dat als je dit wel doet, dit tot onnodige boxing en unboxing leidt.

Toegang tot het value-object

Value-objecten worden vaak als argumenten gebruikt in methoden of als retourwaarden van methoden. Als een value-object is gebouwd als **struct,** worden ze nu niet op basis van hun reference, maar op basis van hun value doorgegeven. Daarmee heb je als ontwikkelaar niet altijd in de hand wanneer een nieuwe kopie van jouw value-object wordt gecreëerd of wanneer instanties worden bewerkt. Om verwarring te voorkomen, is het een goed gebruik er voor te zorgen dat jouw value-object immutable is. In dat geval bevat het geen publiek beschikbare methoden of properties die de interne waarde kunnen wijzigen. In plaats van een huidige instantie aan te passen, is het verstandiger een nieuwe kopie aan te maken, met daarin de nieuwe of bewerkte waarde.
Tip Zorg dat je value-object immutable is.

Het .NET Framework bevat zeer goede voorbeelden van deze techniek. Bekijk het value-object **DateTime** maar eens. Het kent een groot aantal methoden, zoals **AddHours()**, **AddMinutes()** en **AddMonths()**, die de interne waarde van **DateTime** lijken aan te passen. In werkelijkheid doen ze echter iets heel anders; zie codevoorbeeld 9.

Bij elk van deze accessors wordt als eerste de interne waarde van **DateTime** in tikken gedisassembleerd om zo de berekening van de nieuwe waarde mogelijk te maken. Nadat deze is uitgevoerd, wordt een nieuwe kopie van DateTime met de zojuist berekende nieuwe interne waarde geretourneerd.
Een ander voorbeeld. Een van de handige patronen die in het .NET Framework worden toegepast, is TryParse(); zie hiervoor codevoorbeeld 10. Met dit patroon, dat voor de meeste base types is gebruikt, kun je checken of een doorgegeven argument van het specifieke type is. **TryParse()** accepteert een argument dat je wilt omzetten en probeert dit naar het specifieke value-object om te zetten. Vervolgens wordt de waarde **true** of **false** geretourneerd om aan te duiden of dit is gelukt. Als dit het geval is, retourneert de methode een nieuwe instantie als output. Wat dit patroon bijzonder maakt, is dat **TryParse()** gebruikersvriendelijk is. Als het omzetten niet mogelijk is, wordt er geen exception gegooid, maar wordt alleen false geretourneerd.

Je kunt overwegen een of meer van deze **TryParse()**-methoden te implementeren. Een voor ieder type van waaruit je vaak converteert naar het desbetreffende value-object. Meestal komt het er op neer dat je wilt converteren vanuit **string** of een van de andere base types. Implementatie van een additionele **TryParse()** is een goed idee wanneer je zo vaak van een specifiek type naar je value-object omzet dat het voor de performance beter is om de eventuele excepties die het gevolg kunnen zijn van gewone **Parse()**-methoden te voorkomen. Codevoorbeeld 11 voor **Isbn** toont een implementatie van **TryParse()**.

Tip Implementeer eventueel additionele **TryParse()** methoden voor het value-object.

Pragmatisch te werk

Door onderscheid te maken tussen referentieobjecten (in de vorm van domeinobjecten) en andere typen – niet alleen value-objecten, maar ook enumeraties, descriptors en referentietabellen – kun je een stabieler en vakkundiger domeinmodel opbouwen. In value-objecten kun je al een deel van de bedrijfslogica kwijt. Value-objecten zijn vooral handig voor het valideren van eenvoudige semantiek en type safety. Het implementeren van value-objecten brengt echter wel een aantal vragen met zich mee, zoals: implementeer ik het value-object als **class** of als **struct**? Hoe kan ik het best equality en comparison implementeren? Hoe leg ik de interne waarde van het ding vast? Op grond van mijn eigen ervaringen raad ik aan

hierbij vooral pragmatisch te werk te gaan bij het implementeren van value-objecten. Overdrijf de mogelijkheden nooit. Houdt value-objecten klein en ongecompliceerd en vermijd zo veel mogelijk het gebruik van extra functionaliteit. Een duidelijk voorbeeld is comparison en de operator overloads die hiervan het gevolg kunnen zijn. Hierbij sluipen snel fouten in de betrouwbare code van jouw value-object. Implementeer alleen extra functionaliteit als deze essentieel is binnen de context van het domein. Dit in ogenschouw nemend, beveel ik het toepassen van value-objecten van harte aan. Als je ze maar lichtgewicht implementeert, vormen ze een verrijking van de bedrijfslogica van jouw project en zijn ze gemakkelijk te porten naar andere projecten. Een gegarandeerde quick win.

Sander Hoogendoorn is Principal Technology Officer bij Capgemini. In deze rol houdt Sander zich bezig met innovatie op het gebied van softwareontwikkeling. Daarnaast coacht hij een organisaties en projecten op het gebied van agile development, software architecture en patterns, modelering, model driven software development en .NET. Sander is een veelgevraagd spreker op internationale conferenties en seminars en maakt deel uit van Microsoft's Partner Advidsory Council voor .NET. Hij heeft talrijke artikelen en columns geschreven en twee boeken gepubliceerd, over UML en agile softwareontwikkeling. www.sanderhoogendoorn.org en blog.sanderhoogendoorn.org