Sander Hoogendoorn is Partner bij Ordina

Patterns voor het gebruik van referenties in bedrijfsklassen

GOED GEBRUIK VAN REFERENTIES

In de dagen dat applicaties voornamelijk bestonden uit een aantal schermen en een database, was het gebruik van referenties naar gegevens nog eenvoudig. Men hoefde slechts een referentietabel te maken en hiernaar te verwijzen vanuit de hoofdtabellen. Kijk maar eens hoe sentimenteel databaseprogrammeurs kunnen worden over land- of valutatabellen. In moderne objectgeoriënteerde omgevingen, zoals Java en .NET, waarin de bedrijfslogica centraal staat, zijn er echter meer alternatieven voor het omgaan met referenties.

In een klassendiagram worden klassen beschreven in termen van attributen, operaties en relaties met andere klassen. Wanneer een dergelijk diagram wordt omgezet in code, verschijnen de attributen en relaties als properties in de klassen. Properties kunnen nu worden onderverdeeld in verschillende typen. Globaal zijn hierin de volgende typen te onderscheiden:

- **Basic types.** Eenvoudige properties als `FirstName`, `StartDate` of `Limit` hebben eenvoudige typen als `string`, `datetime` of `int`.
- **Value types.** Bij properties als `Email`, `ZIP` of `CreditCard` ligt het iets ingewikkelder. Bij dergelijke properties wordt vaak validatie afgedwongen van de mogelijke property-waarden, door bijvoorbeeld regular expressions en range-checks. Daarom worden dergelijke property-typen beschreven als value types, zoals `BankAccount`, `ZIP` of `CreditCard`.
- **Business class types.** Properties van klassen die een relatie met een andere klasse implementeren, hebben meestal het type van de desbetreffende klasse en vertegenwoordigen een enkele instantie of een verzameling instanties van de gerelateerde klasse. Zo is het waarschijnlijk dat een klasse `Project` een property met de naam `Manager` kent die van het type `Employee` is
- **Reference types.** Als de waarde van een property uit een beperkte collectie waarden komt, wordt in het type van de desbetreffende property gebruikgemaakt van een referentietypen. Er zijn verschillende implementaties van referentietypen mogelijk, die uiteenlopen van enumeraties zoals `CourseLevel` in een class `Course`, tot kleine referentieklassen, als een klasse `Country` of `Currencies`.

Een enumeratie of een referentieklasse lijken de meest voor de hand liggende patronen voor de implementatie van referentietypen, maar er zijn ook andere technieken - die je overigens weer naar keuze kunt toepassen. Dit artikel beschrijft vijf van deze patronen en toont de toepassing en het nut aan. Elk patroon wordt beschouwd aan de hand van een aantal criteria:

- **Type safety.** Is een bepaalde property waarnaar wordt verwezen type safe? Kunnen alleen waarden uit de beperkte collectie worden gebruikt, wanneer het type van de property wordt doorgegeven als argument in een operatie? Is het mogelijk om het type te controleren in code, zonder dat typefouten je code in de war gooien?

Bijvoorbeeld

```
if (MyCourse.CourseLevel == CourseLevels.Expert) { ... }
```

is niet echt hetzelfde als

```
if (MyCourse.CourseLevel == "Expert") { ... }
```

Het tweede voorbeeld is niet erg safe. Hier geldt de wet van Murphy; uiteindelijk zal iemand hier een typefout in maken.

- **Collectie van waarden.** Bij het ontwerpen van de userinterface moeten vaak de mogelijke waarden uit de collectie van het type van de property worden weergegeven (bijvoorbeeld in dropdownlists of in radiobuttons). Hoeveel programmeerwerk vergt het ophalen van alle mogelijke waarden voor een bepaald referentietype?
- **Display-waarde.** In bepaalde gevallen kan het beter zijn andere waarden weer te geven dan de technische waarden die in code worden gebruikt. Zo kan in een enumeration `CourseLevel` de waarde `VeryHigh` zijn gedefinieerd, terwijl die op het scherm liever wordt weergegeven als Very high (met spatie dus).
- **Flexibiliteit.** Kunnen de waarden van de elementen in de collectie worden veranderd zonder dat de code hierdoor moet worden gewijzigd? Staat het aantal elementen vast of kunnen nieuwe elementen worden toegevoegd zonder dat hierdoor de applicatie opnieuw moet worden gecompileerd?
- **Uitbreidbaarheid van referentietypen.** In sommige gevallen kan het nuttig zijn om een referentietype te definiëren waarvan de collectie waarden later kan worden uitgebreid. Stel dat je een framework gebruikt met een referentietype `Actions` dat alle acties bevat die het framework kent. Het kan voorkomen dat ontwikkelaars extra acties nodig hebben, terwijl ze toch de functionaliteit willen blijven gebruiken die door het framework wordt geboden. Dit kan alleen worden bereikt door het uitbreiden van `Actions` met eigen acties.

Tijd om een aantal verschillende patronen te bekijken voor het omgaan met referenties en ze te beschouwen met betrekking tot de voornoemde criteria.

```
public enum CourseLevels
{
   Beginner,
   Advanced,
   VeryHigh,
   Expert
}
```
Codevoorbeeld 1.

```
public class Course
{
   ...
   private CourseLevels level = CourseLevels.Beginner;
   public CourseLevels Level
   {
      get { return level; }
      set { level = value; }
   }
}
```
Codevoorbeeld 2.

```
if (MyCourse.Level == CourseLevels.Beginner) { ... }
```
Codevoorbeeld 3.

```
public bool HasLevel(CourseLevels level)
{
   return (Level == level);
}
```
Codevoorbeeld 4.

Enumeration

Wanneer je beschikt over een vast aantal elementen waaruit moet worden gekozen, is een enumeration een geschikte methode voor het implementeren van referenties. Een enumeration is een hard gecodeerde collectie waarden. De enumeration `CourseLevels` in codevoorbeeld 1 is hiervan een uitstekend voorbeeld. Het gebruik van een enumeration is eenvoudig, zoals in codevoorbeeld 2 wordt aangetoond. Een enumeration is een prettig patroon als de waarde van de property in de code moet worden gevalideerd, zoals je in codevoorbeeld 3 kunt zien. Argumenten van methoden voor de klasse `Course` kunnen op een elegante manier worden opgegeven zoals te zien in codevoorbeeld 4.

Bovendien kunnen alle mogelijke waarden van een enumeration eenvoudig worden opgehaald. In C# volstaat een eenvoudige aanroep van `CourseLevel.GetNames()` of `CourseLevel.GetValues()`. Hoewel het gebruik van een enumeration heel eenduidig is, moet je met dit patroon wel opletten. Als de waarden moeten worden weergegeven op het scherm, is het niet mogelijk andere waarden te gebruiken dan de waarden die in de enumeration zijn gedefinieerd. Dit betekent dat `VeryHigh` in je dropdownlist zal worden weergegeven als *VeryHigh*. Bovendien kunnen enumerations niet eenvoudig worden aangepast. Als je de waarde van een element wilt wijzigen of nieuwe elementen wilt toevoegen, kan dit alleen als de applicatie opnieuw wordt gecompileerd. Het derde punt is dat enumerations in .NET worden geïmplementeerd als value type, hetgeen betekent dat er niet van kan worden geërfd.

Constantencollectie

Om dit laatste probleem te omzeilen, maken ontwikkelaars vaak gebruik van een tweede patroon, namelijk een collectie met constanten. Hierbij wordt de collectie met mogelijke waarden geïmplementeerd als een klasse die attributen bevat waarin constanten zijn gedefinieerd. In codevoorbeeld 5 zie je hoe dit eruitziet. Ook van een dergelijke klasse kunnen gemakkelijk waarden worden gebruikt.

Ook het eerste probleem dat bij de enumerations bestaat, wordt in de constantencollectie enigszins opgelost. Bij het weergeven van waarden op het scherm kunnen de eigenlijke waarden van de constanten worden gebruikt. Zodoende kan `VeryHigh` op het scherm worden weergegeven als *Very high*. De constantencollectie kent echter ook enkele problemen. Allereerst kunnen de mogelijke waarden alleen worden opgehaald via reflection. Dit is wellicht altijd wenselijk. Maar het grootste probleem is dat constantencollecties niet altijd type safe zijn, ook al lijkt dit op het eerste gezicht wel het geval. Omdat constanten als statische waarden zijn gedefinieerd, lijkt de constantencollectie op het eerste gezicht type safe, zoals te zien is bij de validatie in codevoorbeeld 6.

Maar wees voorzichtig! Wanneer in een argument van een methode voor de klasse `Course` een cursusniveau moet worden opgegeven, heeft deze het type `string`. Zie codevoorbeeld 7 en vergelijk dit maar eens met codevoorbeeld 4. Er is dus geen garantie dat alleen type safe waarden worden doorgegeven aan `HasLevel()`, zoals bij een enumeration. Alle drie de statements in codevoorbeeld 8 geven `true` als resultaat als `MyCourse` het niveau `Advanced` heeft. Het ergst is nog dat het derde statement niet eens een exception oplevert; wat gewenst zou zijn, aangezien `Easy` geen geldige waarde is.

Descriptor

In het derde patroon worden de best practices uit de twee vorige patronen gecombineerd. Dit patroon wordt het Descriptor Pattern genoemd. Een descriptor is een klasse die als properties een collectie instanties van zichzelf kent. Ook met dit patroon kan de klasse `CourseLevels` worden geïmplementeerd; zie codevoorbeeld 9. Omdat de instanties statisch zijn, is een descriptor (net als een enumeration) type safe, zowel bij eenvoudige controls als in operation signatures. Dit wordt duidelijk uit codevoorbeeld 10.

Ook kunnen argumenten van methoden voor de class `Course` eenvoudig worden gekenmerkt als `CourseLevel`; zie codevoorbeeld 11. Daarnaast kan er van een descriptor worden geërfd, net als van

```
public class CourseLevels
{
   public const string Beginner = "Beginner";
   public const string Advanced = "Advanced";
   public const string VeryHigh = "Very high";
   public const string Expert = "Expert";
}
```
Codevoorbeeld 5.

```
if (MyCourse.Level == CourseLevels.Beginner) { ... }
```
Codevoorbeeld 6.

```
public bool HasLevel(string level)
{
   return (Level == level);
}
```
Codevoorbeeld 7.

```
bool hasLevel = MyCourse.HasLevel(CourseLevels.Advanced);
bool hasLevel = MyCourse.HasLevel("Advanced");
bool hasLevel = MyCourse.HasLevel("Easy");
```
Codevoorbeeld 8.

```
public class CourseLevels
{
   public static CourseLevels Beginner = new CourseLevels();
   public static CourseLevels Advanced = new CourseLevels();
   public static CourseLevels VeryHigh = new CourseLevels();
   public static CourseLevels Expert = new CourseLevels();
}
```
Codevoorbeeld 9.

```
if (MyCourse.Level == CourseLevels.Beginner) { ... }
```
Codevoorbeeld 10.

```
public bool HasLevel(CourseLevels level)
{
   return (Level == level);
}
```
Codevoorbeeld 11.

```
public class CourseLevels
{
   public static string DisplayValue;
   public CourseLevels(string displayvalue)
   {
      DisplayValue = display;
   }
   public static CourseLevels Beginner = new CourseLevels("Beginner");
   public static CourseLevels Advanced = new CourseLevels("Advanced");
   public static CourseLevels VeryHigh = new CourseLevels("Very high");
   public static CourseLevels Expert = new CourseLevels("Expert");
}
```
Codevoorbeeld 12.

```
public static CourseLevels[] GetValues()
{
   ArrayList list = new ArrayList();
   foreach (FieldInfo pi in type.GetFields())
   {
      list.Add((CourseLevels) pi.GetValue(type));
   }
   return (CourseLevels[]) list.ToArray(typeof(CourseLevels));
}
```
Codevoorbeeld 13.

een constantencollectie. In de eenvoudige implementatie van codevoorbeeld 11 kunnen echter geen andere waarden worden weergegeven dan de gedefinieerde namen van de instanties. Gelukkig kan dit probleem desgewenst worden opgelost met een eenvoudige uitbreiding; zie codevoorbeeld 12. Wel is het zo dat ook een descriptor een vast aantal elementen heeft, net als een enumeration en een constantencollectie. En net als bij het vorige patroon moet de reeks van mogelijke waarden worden opgehaald via reflection. In C# pakken we dit op met een methode als te zien is in codevoorbeeld 13.

Type safe of flexibel?

De tot nog toe beschreven patterns zijn vooral nuttig wanneer het op type safety aankomt. Hiervoor zijn de specifieke elementen hard gecodeerd. Er kunnen zich echter situaties voordoen waarbij type safety minder belangrijk is dan de flexibiliteit om elementen te kunnen wijzigen en/of toe te voegen zonder dat de code daarvoor moet worden gewijzigd. Een eenvoudig voorbeeld hiervan zijn referentietypen als `Countries` of `Currencies`. Het aantal landen hoeft weliswaar niet vaak te worden gewijzigd, maar als er een nieuw land ontstaat moet dit meteen kunnen worden toegevoegd aan de collectie elementen. Zo doemt een tweede groep van referentietypen op waarin flexibiliteit de hoogste prioriteit heeft.

Small business class

Stel dat je een klasse Location hebt, waarmee een locatie wordt opgegeven waar cursussen worden gehouden. Een dergelijke klasse zou een property Country kunnen hebben die verwijst naar een specifieke instantie van een referentietype Countries, zoals in codevoorbeeld 14.

Bij deze implementatie van de klasse `Location` kan de property `Country` worden ingesteld op elke geldige instantie van de klasse `Countries`. De class `Countries` is een small business class. Er worden zelden nieuwe elementen toegevoegd en bestaande elementen worden zelden gewijzigd. Small business classes worden in applicaties vooral gebruikt als referenties. In de meeste gevallen worden de gegevens van instances van small business classes opgeslagen in een database, waarbij elke small business class een gelijknamige tabel kent in de database. Even los van hoe de gegevens worden opgehaald uit de database, ziet een small business class er ongeveer zo uit als in codevoorbeeld 15. Het spreekt vanzelf dat de waarden die worden weergegeven in de user interface voor een small business class kunnen worden gecombineerd uit alle beschikbare properties. een makkelijke manier om dit eenduidig te doen is om de methode `ToString()` te overschrijven en de methode te gebruiken als die wordt getoond in codevoorbeeld 16.

In de database heeft elke record in de tabel `Location` een foreign key naar een bijbehorende primary key uit de tabel `Countries`. Met een small business class wordt een hoge mate van flexibiliteit bereikt. Er kunnen gemakkelijk elementen worden gewijzigd of toegevoegd zonder dat de applicatie opnieuw hoeft te worden gecompileerd. Een ander voordeel van het gebruik van een small business class is dat het ophalen van de collectie van alle instanties erg eenvoudig is. Waarschijnlijk bevat een small business class een statische property *All* (of een vergelijkbare methode) waarmee de verzameling landen als array van de instanties van *Countries* wordt geretourneerd. Daarnaast kan van een small business class gemakkelijk worden geërfd en kan zo extra functionaliteit worden toegevoegd.
Aan het gebruik van een small business class kleven ook enkele

```
public class Location
{
` public string Name;
   public string Address;
   public string City;
   public Countries Country;
}
```
Codevoorbeeld 14.

```
public class Countries
{
   public string Name;
   public string Description;
   public string CountryCode;
   ...
   public static Countries New()
   { ... }
   public static Countries Get(ID id)
   { ... }
   public bool Delete()
   { ... }
   public bool Update()
   { ... }
   public static Countries[] All
   {
      get { ... }
   }
}
```
Codevoorbeeld 15.

```
public class Countries
{
...
   public string ToString()
   {
      return Name + " (" + Code + ")" ;
   }
}
```
Codevoorbeeld 16.

```
public class Location
{
...
   private Countries country;
   public Countries Country
   {
      get { return country; }
      set { if (value.CountryCode != "NL") country = value; }
   }
}
```
Codevoorbeeld 17.

```
public class Countries
{
   ...
   public static Countries Netherlands = Countries.Get("NL");
}
```
Codevoorbeeld 18.

```
public class Location
{
   ...
   private Countries country;
   public Countries Country
   {
      get { return country; }
      set { if (value != Countries.Netherlands) country = value; }
   }
}
```
Codevoorbeeld 19.

nadelen, vooral wanneer type safe validaties gedaan moeten worden op bepaalde velden van de class. Dit wordt duidelijk aan de hand van codevoorbeeld 17 van de class `Location`. In dit voorbeeld wordt een `string` gebruikt om de validatie te implementeren. Ook hier bestaat kans op typefouten (denk aan de constantencollectie). Als type safety vereist is, kan de klasse `Countries` worden uitgebreid met een aantal statische instanties die kunnen worden gebruikt in validaties. In codevoorbeeld 18 is de instantie Netherlands vooraf gedefinieerd. Validatie kan nu worden uitgevoerd met deze statische instantie. De syntax van de validatie is nu gelijk aan die van validatie met een enumeration of descriptor; zie codevoorbeeld 19. Aangezien de instantie `Netherlands` statisch gedefinieerd is, wordt het bijbehorende record slechts éénmaal opgehaald uit de database - op het moment dat de `Netherlands` voor het eerst wordt benaderd. Alhoewel deze techniek meer mogelijkheden biedt voor statische controle, vermindert tegelijkertijd wel de flexibiliteit van de applicatie, omdat deze statische instanties nooit kunnen wijzigen of worden verwijderd uit de onderliggende tabel. Er is altijd sprake van een compromis tussen bruikbaarheid (in code) en flexibiliteit (in gebruik). Bij een keuze uit deze patronen zal deze afweging altijd eerst moeten worden gemaakt.

Smart reference

In het volgende en laatste patroon wordt een eveneens een enkele klasse gedefinieerd. Alleen wordt nu vanuit deze klasse verwezen naar elk referentietype dat kan worden uitgedrukt in de properties van ervan. Meestal heeft een dergelijke klasse, die een Smart Reference wordt genoemd, properties als `Name` en `Description`. De belangrijkste property van een smart reference is echter de property `Type`, waarin wordt gedefinieerd welk specifiek referentietype wordt bedoeld. De property `Type` kan het best worden uitgedrukt met een enumeration, bijvoorbeeld met de naam `ReferenceTypes`, zoals getoond in codevoorbeeld 20. Met deze property kunnen bewerkingen als `All()` en `Default()` worden gedefinieerd om voor een bepaald type alle instanties op te halen van `SmartReference`, of zelfs de default instantie. Dergelijke methoden kunnen zowel statisch als niet-statisch worden gedefinieerd, waarbij ofwel het type wordt opgegeven, of wordt gebruikgemaakt van de property `Type` van de instance.

Dit laatste patroon is zeer flexibel. Alle landen en valuta zijn nu op te vragen zoals is getoond in codevoorbeeld 21.

Er kunnen gemakkelijk nieuwe elementen worden toegevoegd aan elk type in de enumeration `ReferenceTypes`. Het grootste voordeel van het gebruik van een smart reference ten opzichte van het onderhouden van een aantal small business classes is dat hiermee een hoop programmeerwerk bespaard blijft. De onderhoudsfunctionaliteit voor al je referentietypen bestaat uit slechts één scherm, waarin de gebruiker een referentietype kan selecteren en vervolgens de elementen van dat type kan onderhouden.

Deze onderhoudsflexibiliteit heeft echter een klein nadeel. Het gebruik van een smart reference biedt geen automatische garantie voor type safe validatie. Een property van het type `SmartReference` zal extra referentietype validatie moeten uitvoeren, zoals in codevoorbeeld 22 wordt aangetoond. Ook is de wijze waarop een `SmartReference` wordt gerepresenteerd in de user interface gelimiteerd tot de standaard properties die zijn gedefinieerd op de klasse `SmartReference`. Ook hier verdient het aanbeveling de methode `ToString()` te gebruiken om de weergavenwaarde zodanig in te stellen dat deze gebruikt kan worden, zoals in codevoorbeeld 16.

Wanneer gebruik ik welk patroon?

Het gebruik van referenties is onvermijdelijk bij het implementeren van een business layer. Er zijn echter veel talrijke implementaties, in de meeste gevallen variaties of uitbreidingen op een van de vijf in de tabel vermelde patronen. De grote vraag is wanneer je welk

```
public enum ReferenceTypes
{
   Countries,
   Currencies
}

public class SmartReference
{
   public string Name;
   public string Description;
   public ReferenceTypes Type;
   ...
   public SmartReference[] All()
   { ... }

   public static SmartReference[] All(ReferenceTypes rt)
   { ... }
}
```
Codevoorbeeld 20.

```
SmartReference[] countries = SmartReference.All(ReferenceType.Countries);
SmartReference[] currencies = SmartReference.All(ReferenceType.Currencies);
```
Codevoorbeeld 21.

```
public class Location
   {
   ...
   private SmartReference country;
   public SmartReference Country
   {
      get { return country; }
      set { if (value.Type == ReferenceTypes.Country) country = value; }
   }
}
```
Codevoorbeeld 22.

	Type safety	Collectie van waarden	Weergave van waarden	Flexibiliteit	Overerving
Enumeration	Ja	Ja	Beperkt tot enumerated waarden	Vast, wijziging van code vereist	Nee
Constantencollectie	Niet bij gebruik als parameter	Ja, met reflection	Beperkt tot gedefinieerde waarden	Vast, wijziging van code vereist	Ja
Descriptor	Ja	Ja, met reflection	Ja	Vast, wijziging van code vereist	Ja
Small business class	Ja, bij definitie van statische instanties	Ja, gemakkelijk opgehaald uit database	Elke willekeurige combinatie van de velden	Flexibel, toegewezen tabel wijzigen	Ja
Smart reference	Nee, anoniem	Ja, gemakkelijk opgehaald uit database	Elke willekeurige combinatie van de velden	Flexibel, één tabel wijzigen	Ja

pattern het beste kunt gebruiken. Onderstaande tabel biedt een overzicht van de karakteristieken van de vijf patronen aan de hand van de eerder vermelde criteria.

Samenvattend kan worden gesteld dat de eerste drie patronen het meest geschikt zijn in situaties waarin een beperkt en vast aantal elementen wordt verwacht. De verschillen in de drie liggen voornamelijk in de mogelijkheid om andere waarden weer te geven en de behoefte aan uitbreidbaarheid via overerving. De laatste twee patronen zijn vooral geschikt voor situaties waarin sprake is van een grote collectie elementen en waarbij het niet waarschijnlijk is dat deze collectie vaker moet worden gewijzigd dan de code wordt gecompileerd. Hiervoor worden de elementen extern opgeslagen, in plaats van hard gecodeerd. Meestal wordt hiervoor een database gebruikt, soms XML. Samen bieden deze vijf patronen en de talloze variaties en uitbreidingen daarop prettig houvast bij het gebruik van de referentietypen in projecten.

Sander Hoogendoorn Sander Hoogendoorn (www.sanderhoogendoorn.com) is partner bij Ordina. Zijn specialisatie is moderne softwareontwikkeling, en omvat methodieken, objectoriëntatie, UML, component based development, service oriented architectures, Java en .NET. Sander is onder meer verantwoordelijk voor het DaVinci Platform, Ordina's ontwikkelstaat voor .NET. Hij heeft talrijke artikelen geschreven en schrijft columns voor Software Release Magazine en SDN Magazine. Daarnaast is zijn boek 'Pragmatisch modelleren met UML 2.0' is in 2004 gepubliceerd bij Addison Wesley.