Enterprise

Domain Driven Design (DDD) neemt steeds meer in populariteit toe. DDD is een ontwerpstijl waarbij het business domein centraal gesteld wordt en daar rondom services geplaatst worden die het domein faciliteren. DDD is een stijl die dicht aanhangt tegen Object Orientatie (OO) en veel van deze OO-design principes kunnen dan ook in DDD toegepast worden.

DDD en DSL: een mooie combinatie!

Hoe DSL ingezet kan worden om complexiteit in software ontwikkeling te reduceren

Edwin van Dillen
is principal consultant bij Sogyo en bereikbaar via evdillen@sogyo.nl.

Andre Boonzaaijer
is senior consultant bij Sogyo en bereikbaar via aboonzaaijer@sogyo.nl.

Het verleden leert ons dat goed OO-design en implementatie in de praktijk complex bevonden wordt. Domain Specific Languages (DSL) zouden toegepast kunnen worden om deze complexiteit te abstraheren. In dit artikel wordt ook duidelijk gemaakt hoe DSL's in de praktijk ingezet kunnen worden.

Complexiteit van een DDD[1] implementatie

In DDD wordt het business domein centraal gesteld. De grondlegger van deze benadering, Eric Evans[1], beschrijft in zijn werk hoe enitities, aggrates en services ingezet kunnen worden om dit te bereiken. In onze praktijk gaan we een stapje verder door te stellen dat het business domein altijd onafhankelijk moet zijn van de techniek. Dit wil zeggen dat het business domein centraal geplaatst wordt en de technische infrastructurele componenten als services erbuiten. Deze infrastructurele services zorgen ervoor dat het domein in zijn levenscyclus ondersteund wordt.

Voorbeelden van services die het businessdomein omringen zijn presentatie en persistentie services. De presentatieservice zorgt er voor dat de businessobjecten getoond kunnen worden aan de gebruiker. In- en uitvoerschermen worden in deze service geïmplementeerd. De persistentieservice zorgt ervoor dat de businessobjecten kunnen worden opgeslagen en teruggevonden.

Deze ontwikkelstijl kent meerdere vormen van complexiteit die geabstraheerd kunnen worden:

- *Business domeinen*
  In het business domein komen de objecten terug die daadwerkelijk in de businessprocessen een rol spelen. De uitdaging daarbij is het onderkennen van de juiste objecten en hun onderlinge interacties. Welke termen worden in het domein gebruikt en wat zijn de bijbehorende verantwoordelijkheden? Het vinden van de

DSL nieuw?

Domain Specific Languages, of kortweg DSL's, genieten veel aandacht. DSL's representeren een specifieke taal voor een bepaald *domein*. Ondanks het feit dat ze nu veel aandacht genieten zijn ze verre van nieuw. Het was David Parnas die de ideeën van DSL in zijn werk "*On the design and development of program families*" al in 1976 beschreef. Dat het nu zo populair is komt vooral omdat mensen als Martin Fowler er ook over spreken. Hij noemt het overigens "*Language Workbenches*". Daarnaast bieden steeds meer softwareleveranciers goede tools om DSL's te implementeren. Zo heeft Microsoft sinds de indiensttreding van Steve Cook en Jack Greenfield een hele set aan DSL-tools ontwikkeld. In Eclipse zijn er initiatieven als GMT die invulling geven aan DSL-tools.

Soorten DSL's

DSL's komen in verschillende soorten voor. Martin Fowler maakt hierin onderscheid tussen externe DSL's en interne DSL's:

- *Interne DSL*
  Een interne DSL is een ingebedde taal in een general purpose taal. Voorbeelden hiervan zijn Rails als embedded DSL in Ruby, maar ook JMock is een framework dat binnen Java zijn specifieke toepassing heeft; het omschrijven van expectations van Mock objecten. JUnit voorziet in eenzelfde soort interface voor het omschrijven van Unit-tests.
- *Externe DSL*
  Externe DSL's zijn vaak talen die meer los staan van een gebruikte general-purpose taal, en die vaak een eigen syntax en grammatica hebben. Deze externe DSL's lossen de problemen op door een codegeneratie-slag te maken naar de gebruikte general purpose taal. AspectJ zou gezien kunnen worden als zo'n externe DSL. De vele configuratie XML files die tegenwoordig gemaakt worden – neem bijvoorbeel Hibernate mappings – zijn ook voorbeelden van externe DSL's.

  Naast de opdeling in interne en externe DSL's maakt Steve Cook onderscheid in horizontale en verticale DSL's [3].

- *Horizontale DSL*
  Horizontale DSL's zijn gericht op technische aspecten binnen software ontwikkeling. Wordt een lagenmodel gebruikt dan richt een horizontale DSL zich op een specifieke laag.
- *Verticale DSL*
  Verticale DSL's zijn meer businessgeoriënteerd. Ze beschrijven een specifiek (onderdeel uit het) businessdomein.

Als we de praktische toepassing van deze twee verschillende doorsneden op vormen van DSL's in een schema weergeven en verschillende onderdelen in moderne software ontwikkelingstrajecten toewijzen ontstaat het onderstaande beeld.

DSL	Horizontaal	Verticaal
Intern	Validation rules Change notification	Domein model
Extern	SQL XPath (N)Hibernate mappings	(Business) rule repository

juiste business domeinobjecten is een lastige aangelegenheid.

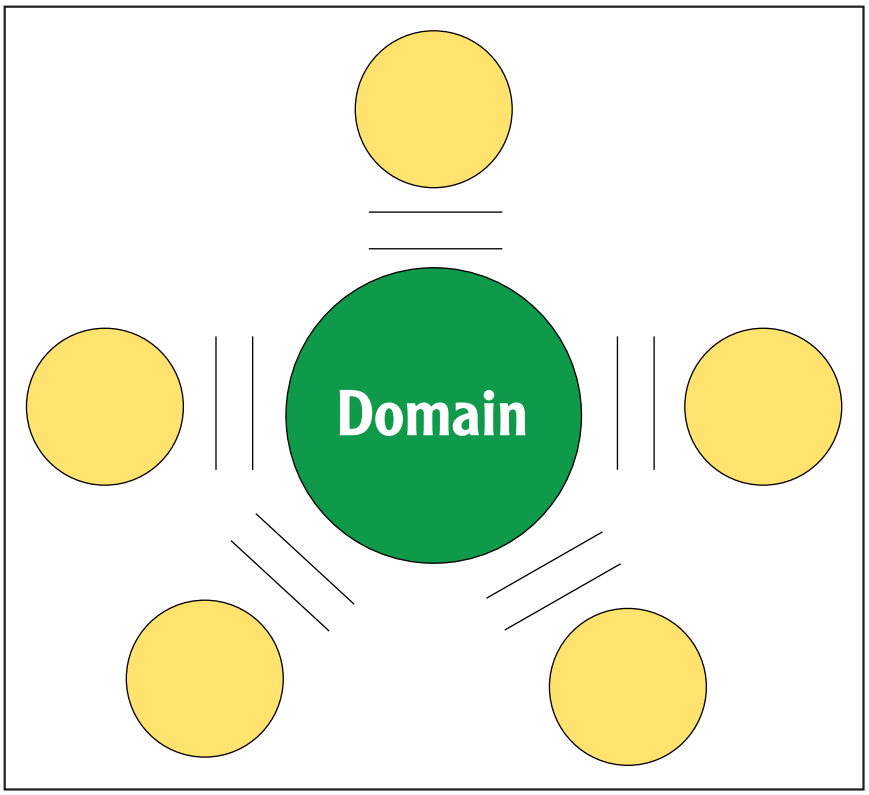
- *Faciliterende Services*
  De faciliterende services zijn van technische aard. Ze hebben allen hun specifieke doelstellingen en eigenschappen. Een veel genoemd voorbeeld is persistentie in de vorm van wegschrijven naar en ophalen van een relationele database. De persistentieservice is dan verantwoordelijk voor vertaling van en naar een relationele structuur. Vaak blijft het daar echter niet bij: deze service kan ook voorzien in transactie ondersteuning. Daarmee kan het domein ook weer gefaciliteerd worden in concurrency uitdagingen. Dit zijn slechts enkele voorbeelden van verantwoordelijkheden die deze service vervult. In de praktijk betekent dit dat iemand die volgens deze stijl ontwikkelt kennis moet hebben van al de eigenschappen van deze services en ook van de manier waarop deze eigenschappen geconfigureerd of toegepast moeten worden.

Nu we de verschillende aspecten van de domeingedreven ontwikkelstijl kort hebben belicht zullen we wat verder ingaan op DSL's en hoe deze twee samenhangen.

Wat zijn DSL's?

DSL's, *Domain Specific Languages*, kunnen gedefinieerd worden als: *beperkte, kleine en zeer context-afhankelijke 'taaltjes' die de omschrijving van een bepaald 'domein' faciliteren*. De term Domain in DDD en DSL is ietwat verwarrend: de betekenis is namelijk niet helemaal gelijk. Waar

Figuur: Zonnebloemmodel



Definities zijn prachtig, maar wie zit te wachten op wederom een nieuwe manier van software ontwikkelen?

we in DDD Domain definiëren als het bedrijfsspecifieke deel van een systeem kan Domain in DSL een andere betekenis hebben. Welke verschillende betekenissen dat zijn zal verderop duidelijk worden.

Wat DSL's allemaal gemeen hebben is een grote afhankelijkheid van context en een zeer beperkte scope van toepassing, veel meer dan bijvoorbeeld een algemeen toepasbare taal als Java. DSL's hebben, meer dan bijvoorbeeld bij een API het geval is, een gerichte focus op de taal die ze omvatten. Bij het maken van een DSL is het gebruikelijk om de taal voor de gebruiker zo vloeiend mogelijk te maken, door Martin Fowler[2] ook wel 'Fluent Interfaces' genoemd.

Veel gebruikte voorbeelden van DSL's zijn de verschillende Shell-scripttalen die in Unix te vinden zijn, SQL, XPath of HTML. In een meer businessgeoriënteerde context wordt de Starbuck's DSL [4] vaak genoemd (Iced Latte Grande) of denk aan een bestelling bij de chinees (3 x 110 met rijst, 2 x 112 met bami en extra sambal). In sidebox 2: *Soorten DSL* worden verschillende soorten DSL's die onderkend kunnen worden verder beschreven.

Naast het onderscheid in verschillende soorten DSL's hebben ze ook verschillende verschijningsvormen. Ze kunnen grafisch gerepresenteerd worden, denk bijvoorbeeld aan een klassendiagram in UML. XPath is een voorbeeld van een tekstuele DSL. In de sidebox 3: *verschijningsvormen van DSL* wordt dit onderscheid verder toegelicht.

Doelen van een DSL

Definities zijn prachtig, maar wie zit te wachten op wederom een nieuwe manier van software ontwikkelen? Is na structured programming, OO, SOA weer een nieuw soort paradigma nodig?
Het is belangrijk om aan te merken is dat hier *geen* sprake is van een nieuw paradigma. (zie sidebox 1: DSL nieuw?) DSL's werken prima samen met bestaande ontwikkelstijlen als OO en SOA. Wat zijn dan de kenmerken ervan en belangrijker, wat is de meerwaarde?

Met het toepassen van DSL's kan het abstractieniveau van implementaties in software ontwikkeling verhoogd worden. Abstraheren betekent dat complexiteit verborgen kan worden door code op een hoger en vaak meer natuurlijk niveau te beschrijven. Kort samengevat is dit het hoofddoel van DSL's.

Door complexe en repeterende onderdelen van de software te abstraheren kunnen ontwikkelaars implementaties op dit hogere abstractieniveau beschrijven. Deze hogere abstractie kan in de DSL vastgelegd worden. Vervolgens kan vanuit de DSL de code op het lagere abstractieniveau gegeneerd worden. Met DSL's kan de complexiteit voor ontwikkelaars verborgen worden en daarmee de ontwikkeltijd voor software verkort worden.

Een goed voorbeeld is SQL. Door de komst van deze DSL kan de ontwikkelaar in iedere implementatie van een relationele bron queries afvuren. De ontwikkelaar hoeft zich geen zorgen meer te maken over de onderliggende database: SQL verbergt de implementatiespecifieke complexiteit, in dit geval de concrete database. Dit neemt overigens niet weg dat je iemand nodig hebt in je team die wel kennis heeft van de database, in staat is deze in te richten en te tunen. Het gebruik van DSL's heeft echter tot gevolg dat slechts één persoon deze detailkennis nodig heeft.

DDD en DSL

De grote vraag die nu rest is hoe DSL's kunnen bijdragen aan de toepassing van DDD. Om dit vast te stellen worden de twee grote uitdagingen van DDD toepassingen belicht.

DSL en het Business domein.
In moderne platformen als .NET en Java wordt het business domein uiteindelijk geïmplementeerd als een verzameling van klassen in de taal (*POCO's* of *POJO's*). Een ontwikkelaar zou deze implementaties rechtstreeks kunnen schrijven. Echter een van de grootste uitdagingen van het vastleggen van een business domein is het komen tot de juiste termen, ofwel het beschrijven van het jargon. Daarnaast speelt de samenwerking tussen

Verschijningsvormen van DSL

De complexiteit kan in een DSL op twee manieren worden gerepresenteerd:

- *Grafische DSL*
  De meest bekende grafische DSL is misschien wel UML. In de UML worden visuele weergaven van klassen gedefinieerd. Door deze visualisatie kan in een relatief compact plaatje inzicht gegeven worden in een totaal klassendiagram.
  Grafische DSL's vereisen een omgeving waarin ze gevisualiseerd kunnen worden. Microsoft DSL-tools is zo'n omgeving. Daar is expliciet gekozen om eerst alleen de grafische DSL's te ondersteunen in de Visual Studio omgeving.
- *Tekstuele DSL*
  Van de tekstuele DSL's is XSLT een mooi voorbeeld. XSLT templates kunnen in een simpele teksteditor worden geschreven. Vervolgens aan een parser worden aangeboden die de template interpreteert.

de objecten een belangrijke rol. Het visualiseren van de relaties tussen de businessobjecten kan dan een grote hulp zijn.
In eerste instantie hoeft hier nog geen nieuwe DSL voor ontwikkeld te worden. UML klassendiagrammen kunnen hier een grote rol spelen. Feit is wel dat klassendiagrammen slechts ondersteuning bieden op generiek niveau. Ze zeggen iets over klassen en relaties en hoe die gevisualiseerd kunnen worden. Om ze ook specifiek tot een domein te betrekken zouden UML-Profielen toegepast moeten worden. Dan wordt het mogelijk een rijkere businessdomein-specifieke taal te ontwikkelen.

In ontwikkelomgevingen waar gebruik gemaakt wordt van Java technologie wordt deze stijl al wel toegepast. In de Microsoft omgeving wordt UML over het algemeen als schets tool gebruikt en zien we nog weinig UML-profielen toegepast worden. In deze omgeving kan op basis van DSL-tools een visuele implementatie van de klassendiagrammen ontwikkeld worden.

Het toepassen van een generieke DSL als het UML klassendiagram voor een business domein heeft zichzelf in de praktijk al bewezen en is van grote waarde. Het vereenvoudigt de communicatie tussen teamleden en helpt daarbij bij het voorkomen van misinterpretaties.

Het wordt zinvol om een DSL met businessdomein specifieke elementen te ontwikkelen als het businessdomein frequent wijzigt dan wel als veel verschillende mensen in het domeinwerken. Dat rechtvaardigt het om de specifieke business termen in de DSL zelf te verwerken. Concreet betekent dit dat een UML profiel gebruikt kan worden in de Java wereld en een DSL in DSL-tools in de Microsoft wereld.

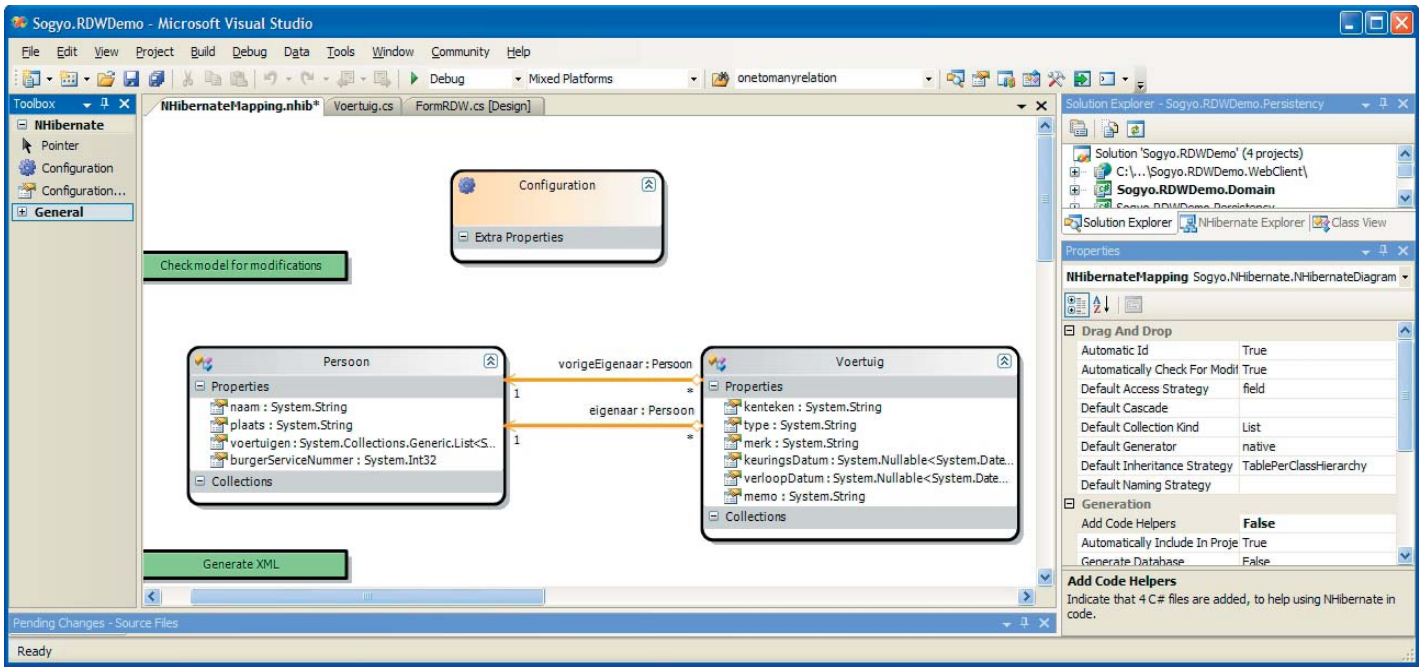
Om te illustreren hoe een domeinontwerp baat kan hebben bij een eigen domeinspecifieke taal en hoe deze geïmplementeerd kan worden nemen we het voorbeeld van bestellingen in een snackbar. Bijgevoegde figuur biedt de structuur voor implementatie van deze snackbar DSL. Het ontwerp bevat de klassen *Snack* en *Saus* waarmee we snackbar bestellingen kunnen samenstellen. Door handig te schuiven met returnwaardes en een aantal static factory methoden, kunnen klassen gebruikt worden door in code *bestellingen* van de volgende vorm te doen:

```
new Bestelling().
    Bestel(Patat.Groot().Speciaal()).
    Bestel(Frikandel.Gebakken().Met());
```

We zien hier in code dus een domeinspecifieke taal ontstaan die zeer goed aansluit bij de natuurlijke taal. Uiteraard is dit voorbeeld vrij star en simpel uitgewerkt en is het domein eigenlijk bij iedereen wel bekend. Denk echter eens na over de kracht van een dergelijke ingebedde taal in de ontwikkeling van zeer complexe systemen waarbij domeinkennis bij de programmeur verre van vanzelfsprekend is.

DSL en faciliterende Services

De faciliterende services zijn van technische aard. Neem de persistentie service. Deze is verantwoordelijk voor het opslaan en kunnen terugvinden van domeinobjecten. Als basis voor deze service kan een object-relational mapper als (N)Hibernate gebruikt worden. Om dit framework toe te passen



moeten mapping geschreven worden. Dit zijn XML documenten waarin beschreven wordt hoe een object opgeslagen moet worden. Het schrijven van zo'n mapping is voor een enkele klas zeer eenvoudig. Een domeinmodel is in de praktijk veel uitgebreider, dus zo ook de mapping die gemaakt moeten worden.

In deze situatie kan een visuele DSL een grote bijdrage leveren. De domeinklassen kunnen op de DSL worden gesleept. De DSL kan op basis van reflectie de klasseneigenschappen bepalen en deze visualiseren. Op deze manier kan een basis van de mapping gegenereerd worden. Vervolgens kan de gebruiker van de DSL meta-informatie toevoegen. Ook dit kan plaatsvinden door eigenschappen van de gevisualiseerde objecten te wijzigen.

In de afbeelding is een voorbeeld gegeven van een, door Sogyo ontwikkelde, NHibernate DSL die in Microsoft Visual Studio wordt gebruikt. Van de twee domeinklassen worden specifiek de properties, collecties en associaties weergegeven. Deze informatie wordt gebruikt om de mapping mee te genereren. Naast de domeinklassen is nog een bijzondere klas weergegeven: Configuration. Deze vertegenwoordigt het NHibernate configuratie bestand. In dit bestand kan het gedrag van NHibernate worden vastgelegd.

Middels deze DSL wordt een deel van de implementatie van de persistentie service verborgen voor de gebruiker. Daarmee hoeft de gebruiker van de DSL minder gedetailleerde kennis van de service paraat te hebben. Het meest interessante is nog wel dat het repeterende handelingen voorkomt. Wie zit er nu te wachten om alle mappingen met de hand te moeten schrijven.

Conclusie

DSL's kunnen de repeteerde handelingen in softwareontwikkeling automatiseren. Door specifieke DSL's te ontwikkelen voor de DDD ontwikkelstijl wordt de complexiteit deels verborgen. Daarmee wordt deze ontwikkelstijl zelf toegankelijker.
Het ontwikkelen van een DSL is geen sinecure. DSL's voor het businessdomein zullen specifiek per branch/organisatie ontwikkeld moeten worden. De DSL's voor faciliterende services zullen ook door de IT-industrie zelf ontwikkeld worden. Houd hier de leveranciers en open source wereld dus goed voor in de gaten.

Resources

[1] "Domain Driven Design: tackling complexity at the hart of software" Eric Evans.
[2] "Language Workbenches: The Killer-App for Domain Specific Languages?" Martin Fowler http://www.martinfowler.com/articles/languageWorkbench.html
[3] "Domain-Specific Languages" Steve Cook
[4] "Language oriented programming" Martin Fowler, Neal Ford http://www.nealford.com/downloads/conferences/Neal_Ford_Martin_Fowler-Language_Oriented_Programming-keynote.pdf

Noten

1 Voor een uitgebreide omschrijving van DDD wordt verwezen naar het artikel "Domain Driven Design: achtergronden en ervaringen uit de praktijk " van Edwin van Dillen, Ralf Wolter en Thomas Zeeman welke April 2007 in dit blad verscheen.