

# Domain Driven Design in de Praktijk

## Een case met Lego Mindstorms

*Domain Driven Design (DDD), Domain Oriented Architecture (DOA) en domeingeoriënteerde applicatiearchitectuur zijn verschillende namen voor een stijl waarmee applicaties ontwikkeld kunnen worden. Steeds vaker worden deze termen genoemd, maar wat is het eigenlijk?*

*Het is géén nieuw concept. Sterker nog, het zijn ideeën die door mensen als Alistair Cockburn, Ivar Jacobson en Rob Vens al langer onderkend worden. Het is een denkstijl waarin het business domein dat de software moet ondersteunen, centraal gesteld wordt. Het domein dat de business logica representeert wordt volledig los van de technische infrastructuur gerealiseerd.*

In dit artikel laten we zien hoe je een DDD implementatie kunt maken om een Lego Mindstorms robot aan te sturen. Hiermee geven we inzicht hoe een domein wordt ontworpen en opgebouwd. Daarnaast wordt een aantal manieren genoemd hoe de services die het domein faciliteren, geïmplementeerd kunnen worden en hoe vervolgens de services en het domein met elkaar worden verbonden. Daarmee geven we een overzicht van de basis ingrediënten om een DDD implementatie te maken.

In dit artikel is gekozen om een Lego Mindstorms robot als voorbeeld domein te nemen. Een DDD implementatie kan natuurlijk betrekking hebben op ieder willekeurig domein. De keuze voor de robot is simpel:

1. het spreekt tot de verbeelding;
2. in dit domein moeten meerdere beslissingen worden genomen en er moet iets gedaan worden met historie.

Houd dus tijdens het lezen van het artikel continu in het achterhoofd dat het ook op je eigen business domein betrekking kan hebben.

## Domain Driven Design

Alvorens direct van start te gaan met het modelleren van een domeinmodel, staan we even stil bij het grotere plaatje. Wat we met domain driven design willen bereiken is het business domein centraal stellen, volledig ontdaan van zaken die niet direct met het betreffende geautomatiseerde bedrijfsproces te maken hebben. Alles wat niet direct met de kern van deze bedrijfslogica te maken heeft, maar een meer faciliterende rol heeft in het geheel, plaatsen we buiten dit domein en duiden we aan met de term services.

Zoals gezegd bevat het domeinmodel enkel alle eigenschappen en verantwoordelijkheden die in het business domein aanwezig zijn. De services hebben de eigenschappen en verantwoordelijkheden om het domeinmodel in zijn bestaan te faciliteren. Voorbeelden van services zijn het opslaan van het domeinmodel (persistentie services) of het ontsluiten van het domeinmodel naar het web (presentatie services). Dit geheel wordt in onderstaande figuur afgebeeld, wat we ook wel aanduiden met het “**zonnebloemmodel**”. [1]

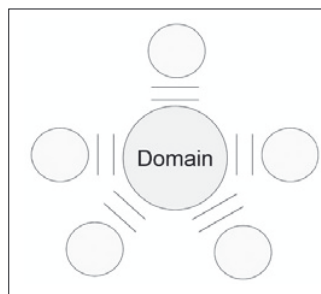


Fig. 1: Zonnebloemmodel

Uiteraard is domain driven design geen doel op zich, maar een middel. Het doel dat we met deze manier van ontwerpen willen bereiken is betere onderhoudbaarheid en snellere ontwikkeling van complexe software. Doormiddel van scheiding van de feitelijke

kern van de applicatie en de faciliterende onderdelen kan een hoge mate van flexibiliteit bereikt worden. Dit is vooral wenselijk als er sprake is van een dynamische bedrijfsomgeving. Bijkomend voordeel is een hoge mate van herbruikbaarheid van de onderdelen.

Theoretisch is dit een mooie gedachte en er is al veel over gezegd en geschreven [2]. Vaak worden echter de zeer praktische zaken als koppeling tussen services en domein achterwege gelaten. We zullen in de rest van dit artikel een volledig uitgewerkt voorbeeld illustreren voor ontwerp van het domein, implementatie van verschillende services en koppeling tussen deze verschillende onderdelen.

## Domein

Om de domeingedreven ontwerpmethode te illustreren definiëren we een probleem in de context van een Lego Mindstorms™ robot.



Fig. 2: Lego Mindstorms robot

We richten ons in eerste instantie op het kunnen volgen van een parcours dat is afgezet met een zwarte lijn. We zullen later zien dat we met de domeingedreven aanpak dit gedrag ook op een intuïtieve manier kunnen aanpassen. De robot heeft twee wielen die aangestuurd kunnen worden om alle kanten mee uit te rijden. Op de voorkant heeft de robot een lichtsensor, een druksensor en een ultrasone sensor. Het gedrag dat de robot in eerste instantie moet vertonen is dat deze zelfstandig een parcours kan afleggen door een zwarte lijn te volgen. Op [www.sogyo.nl/dddrobot](http://www.sogyo.nl/dddrobot) kun je een filmpje bekijken met de robot in actie.

In dit domein klinkt het logisch om te beginnen met een robot klasse. De robot klasse moet ons het gedrag kunnen leveren om het parcours te volgen. Om het parcours te kunnen volgen zal de robot periodiek een waarneming moeten doen om daaropvolgend actie te ondernemen. Het doen van de feitelijke waarneming heeft een statusverandering van de robot tot gevolg en zijn dientengevolge als methoden van de robot klasse opgenomen. De feitelijke waarnemende actoren – die deze methoden aanroepen – vallen buiten het domein en komen later aan bod. Om prikkels van alle sensoren te kunnen ontvangen implementeert de robotklasse de methodes `Seeing()`, `Feeling()`, `Hearing()` en `DetectingLine()`.

Om deze waarnemingen te kunnen interpreteren introduceren we een periodieke “hartslag” voor de robot. Deze hartslag kan dan aangesloten worden op een timer – later wordt duidelijk hoe we dat regelen. Voor het domein is het alleen van belang dat de robot een methode `HeartBeat()` implementeert om op de timerprikkels te kunnen reageren.

Op iedere hartslag kan de robot bepalen wat zijn volgende actie gaat worden. Het bepalen van de actie halen we uit de robot klasse en wordt een specifieke klasse. Door de actie uit de robot klasse te halen kunnen we actie de eigenschap geven dat deze zijn voorgaande actie kent. Dus iedere keer dat de robot een hartslag heeft, wordt een actie aangemaakt. Deze actie kent zijn voorgaande actie. De laatste actie kan vervolgens bepalen wat de meest logische richting is die de robot moet rijden. Deze ketting van acties is in feite het geheugen van de robot.

Een voorbeeld: stel de robot rijdt niet op de lijn. Dat betekent dat hij de lijn moet gaan zoeken. In het meest simpele geval gaan we standaard naar links om de lijn te zoeken. Als we bij de volgende hartslag de lijn nog niet gevonden hebben, dan blijven we naar links zoeken. Dit doen we vier keer achter elkaar, hebben we dan nog steeds de lijn niet gevonden dan gaan we acht keer naar rechts net zolang tot dat we de lijn gevonden hebben.

Dit is een vrij simpel algoritme om de lijn te zoeken, maar wel eentje die werkt. Doordat de actie zijn voorgaande

actie kent, kan deze eenvoudig bepalen welke kant de robot moet uitrijden. Overigens is het belangrijk om op te merken dat een actie alleen zijn voorgaande actie kent. Het is dus een keten van acties en de keten lost het vraagstuk op welke kant we opgaan (zie “Chain of Responsibility” pattern [3]).

Het komt erop neer dat de robot klasse het bepalen van zijn actie overlaat aan een helper klasse die we in dit geval ook gewoon actie noemen. Nadat de robot zijn actie heeft bepaald, moet die in beweging komen. De beweging plaatsen we ook in een eigen klasse. Door aan een bewegingsobject de laatste actie mee te geven kan de beweging aan de actie vragen welke richting de robot uit moet gaan. Het bewegingsobject zal vervolgens een richting aannemen. Op het moment dat een bewegingsobject een richting aanneemt zal deze ook een change event genereren. Het object vertelt daarmee aan de buitenwereld dat hij veranderd is. In de buitenwereld zal dit event dus door relevante actoren moeten kunnen worden afgevangen zodat ze erop kunnen reageren. Deze externe actoren zullen we verder aanduiden als services en de koppeling tussen deze services en het domein zullen we ook nader toelichten.

Om een en ander te verduidelijken is hieronder een klassediagram opgenomen van het domeinmodel. Mocht je de implementatie van het domein willen zien ga dan naar [www.sogyo.nl/dddrobot](http://www.sogyo.nl/dddrobot).

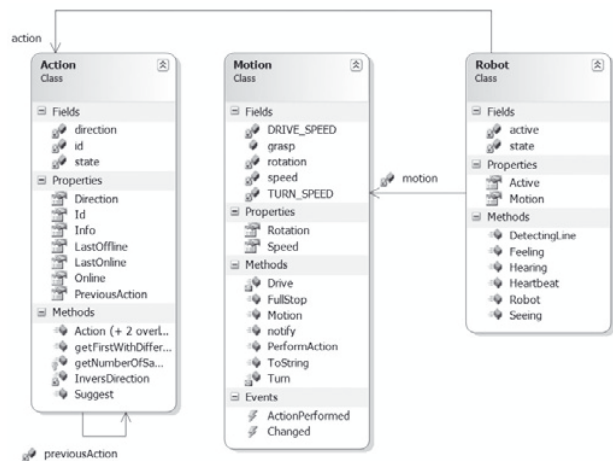


Fig. 3: Domeinmodel

### Services

Zoals eerder gemeld bij de toelichting op ons robotdomein hebben we externe actoren ondergebracht in services. We spreken van een service zodra de functionaliteit die geboden wordt niet relevant is voor het domein. Hoewel dit suggereert dat services dus niets met een domein te maken hebben, kan het wel degelijk mogelijk zijn dat een service zelf ook domein gedreven opgezet is. Voor onze robot hebben we de hieronder beschreven services onderscheiden.

## MindstormsService

Deze service verzorgt alle communicatie van en naar de Lego NXT brick. De interactie met de legosteen is gebouwd bovenop de NXT# library [4]. Deze service biedt methodes om de motoren van de NXT aan te sturen. De service wordt als singleton aangeboden via de GetInstance() methode, dit om meerdere instanties te voorkomen.

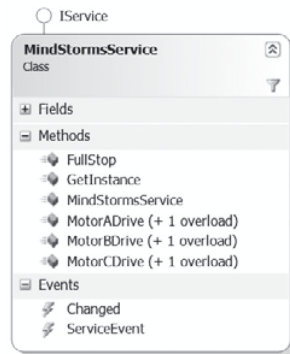


Fig. 4: MindStormsService

Naast methoden voor het aansturen van de methoden definieert deze service een Changed event dat veranderde sensorwaarden doorgeeft. Voor deze sensorwaarden zijn een aantal data klassen gedefinieerd (Sound, Vision, Touch), waar we hier verder niet op in zullen gaan.

## TimerService

De timerservice is een zeer eenvoudige klasse die enkel een timer bevat en op een instelbaar interval zijn Changed event afvuurt. Deze service voorziet het systeem van een klok zodat er op bepaalde tijdstippen actie kan worden ondernomen. Deze service zal dus aangesloten moeten worden op de eerder besproken HeartBeat() methode van de robot klasse.

## PersistencyService

De PersistencyService is een klasse die action objecten uit het domein kan persisteren. Zoals al bleek uit het domeinmodel bevat de motion klasse een tweede event dat daar niet verder omschreven is, te weten ActionPerformed. Dit event geeft de betreffende actie mee aan de afhandelende methode, zodat bij correcte aansluiting deze persistency service via zijn Update() methode het betreffende object kan persisteren.

## GUIService

De volledig puristische aanpak van domeingedrevenheid stelt dat ook de gebruikersinterface(s) als service aangeboden dient te worden. De bootstrapping (die we verderop zullen bespreken) wordt gedaan vanuit een eigen interface. Ter illustratie hebben we echter wel een volledig losstaande applicatie gemaakt die een grafische representatie van de robot faciliteert. Deze applicatie is ontsloten d.m.v. een WCF (Windows Communication Foundation) Service.

## Adaptoren

Tot zover hebben we het gehad over scheiding tussen actieve delen van een applicatie in domein en services. De kern van de applicatie, het logische beslissingscentrum, ligt in het domein. Services worden volledig los gezien

van het domein en hebben als zodanig geen directe communicatie met de domeinobjecten.

Uiteraard is deze communicatie wel noodzakelijk. Er bestaat immers vrijwel geen software die op geen enkele manier reageert op prikkels van buitenaf ofwel signalen afgeeft aan de buitenwereld. Deze communicatie en het generiek faciliteren ervan biedt een uitdaging voor onze hier beschreven benadering van DDD.

The Gang of Four [2] doet een voorstel voor het implementeren van een universele interface tussen verschillende objecten via het *Adapter pattern*. Een adapter verzorgt de vertaling van communicatie tussen verschillende (niet compatibele) objectsoorten.

We hebben eerder gesteld dat objecten binnen het domein enkel signalen (events) afgeven naar de buitenwereld. Deze events kunnen afgevangen worden door adaptoren om vervolgens een service aan te spreken. Deze relatie is afgebeeld in onderstaande figuur.

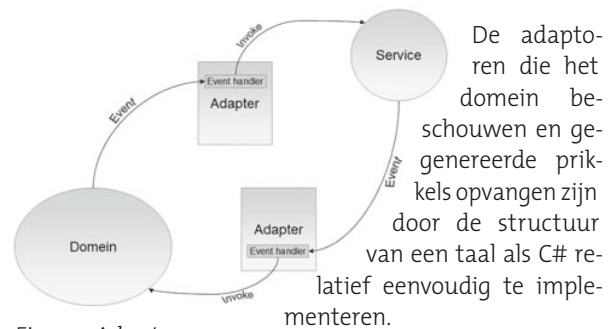


Fig. 4a: Adaptoren

We hebben voor onze robotimplementatie gekozen om het domein op één manier naar de buitenwereld toe te ontsluiten: via events. Door globaal bekende delegates te declareren kan iedere adapter relevante - vanuit het domein gegenereerde - events afhandelen.

Ter illustratie zijn in de robot referentieapplicatie adaptoren gemaakt voor aansturen van de motoren (op basis van events uit de motion klasse) en voor het aansturen van een persistency service op basis van NHibernate.

Communicatie van services richting domein verloopt precies in tegenovergestelde richting. We hebben gesteld dat het domein geen afhankelijkheden of kennis mag hebben van de buitenwereld. We kunnen dus geen events van buitenaf in het domein afhandelen. Domeinspecifieke functionaliteit zal door adaptoren aangeroepen moeten worden om prikkels het domein in te krijgen. We gaan hier even uit van een gelijksoortige manier van communiceren van services met de buitenwereld: deze genereren dus ook relevante events.

De meest simpele vorm om dit te implementeren is door de adaptoren zelf de verantwoordelijkheid te geven om events van een bepaalde service af te handelen. De event-handler die reageert op een service-event doet dit dan

door rechtstreeks aanroepen van methoden op domein- objecten. Op deze manier heeft de adapter echter veel verantwoordelijkheden (het abonneren op het event, het filteren van de events, het aanroepen van de domein- functionaliteit) en is er tevens sprake van een vrij hoge graad van koppeling.

## Domeinspecifieke functionaliteit zal door adaptoren aangeroepen moeten worden om prikkels het domein in te krijgen

Een alternatief dat we hebben uitgewerkt gaat via een Command pattern implementatie. Alle mogelijke domein- specifieke invokes worden aangemaakt als het type ICommand en deze commands worden via een singleton Commandrunner uitgevoerd. Onze implementatie ondersteunt commands die methoden in het domein kunnen aanroepen (InvokeCommand) en methoden die properties in het domein kunnen wijzigen (SetCommand). Beide commands worden uitgevoerd met behulp van reflectie.

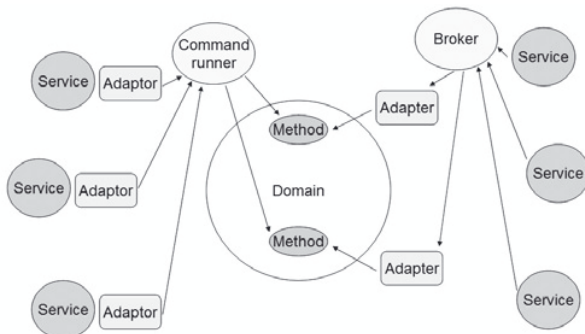


Fig. 4b: Command en Broker

De adaptoren worden op deze manier losgekoppeld van het domein; ze zijn echter nog wel verantwoordelijk voor filtering van de service events. Voor een overzicht van de implementatie van de command-based afhandeling zie onderstaande figuur.

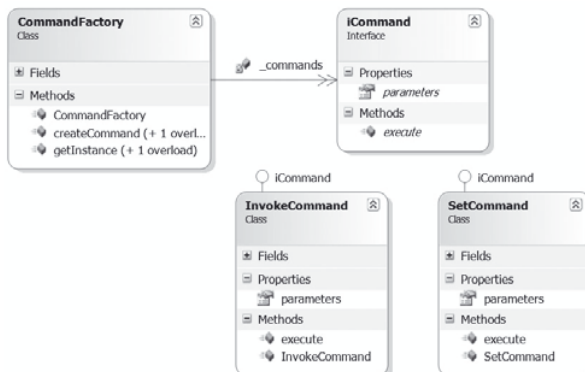


Fig. 5: CommandAdapter

Een laatste vorm die we hebben uitgewerkt is event af- handeling via een centrale Broker. Deze entiteit vangt – in onze implementatie – alle service-events af. Vervolgens worden deze events gedistribueerd naar alle geabon- neerde adaptoren. Het voordeel hiervan is dat de event fil- tering ook gecentraliseerd is en dus niet alle events meer naar alle adaptoren gestuurd worden zoals in de beide eerder genoemde alternatieve implementaties. Een na- deel is dat (in onze huidige implementatie) de services allemaal een IService interface moeten implementeren om door de Broker ondersteund te worden (wat dus weer een ongewenste koppeling introduceert). In onderstaan- de figuur is de broker structuur weergegeven.

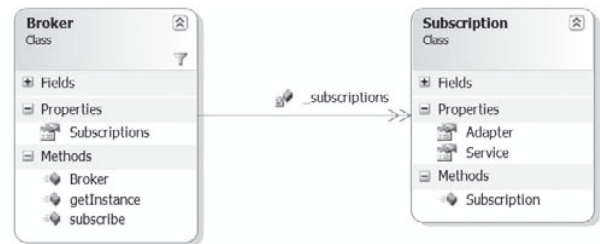


Fig. 6: Broker

Ter illustratie zijn in de referentieapplicatie adaptoren gemaakt in deze drie smaken voor events vanuit de Nxt Service van respectievelijk de sonarsensor, de lichtsensor, de druksensor en de geluidssensor.

### Bootstrapping & Lijm

De bak met onderdelen is compleet. We hebben een domein, omringende services en I/O adaptoren geïmple- menteerd. Wat rest is de boel aan elkaar te lijmen en de initialisatie te realiseren. Afhankelijk van de gekozen adapter implementatie moet in de bootstrap de broker of commandrunner worden geïnitialiseerd. Daarnaast kan hier afhankelijk van de gewenste flexibiliteit confi- guratie van de adaptoren ondergebracht worden.

Een groot bijkomend voordeel is dat tijdens deze boots- trap volledig dynamisch services en adaptoren toege- voegd dan wel verwijderd kunnen worden. De aangeslo- ten services op het domein zijn dus tijdens de bootstrap – op runtime – volledig dynamisch toe te voegen en te verwijderen, eventueel via configuratie settings. In de praktijk betekent dit dat we nieuwe presentatie services of bijvoorbeeld communicatie services eenvoudig kun- nen toevoegen of verwijderen

Ter illustratie zijn in de referentieapplicatie drie vormen van bootstrapping opgenomen om de drie adapterim- plementaties te kunnen instantiëren.

### Discussie

Domein en services zijn nu compleet en gekoppeld. Uiter- aard is het nu tijd om te verifiëren of de stelling dat DDD een grotere flexibiliteit in de software heeft gebracht dan we gewend zijn, ook waar is. Zoals eerder gesteld is het mogelijk om op een intuïtieve plaats het gedrag dat

de robot vertoont (nu: het volgen van een lijn) te wijzigen. Stel dat we een obstakel willen omzeilen dat binnen gezichtsafstand van de robot opduikt. Op welke delen van het totaalsysteem zou deze wijziging impact hebben? Het antwoord laat zich raden: het domein. Het domein is de plaats waar de beslissingen genomen worden voor het definiëren van de nieuwe actie, te weten de suggest() methode van de action klasse.

## Het domein is de plaats waar de beslissingen genomen worden voor het definiëren van de nieuwe actie

De flexibiliteit is één aspect, maar hoe zit het met de complexiteit? Ons domein bestaat uit drie klassen, wat het feitelijke gedrag van de robot tamelijk eenvoudig omschrijft. Daarnaast is echter een veelvoud aan classes gedefinieerd om enerzijds de losstaande services te implementeren, alsmede de lijnlaag tussen de services en domein te faciliteren. Als we gelijkwaardig gedrag implementeren en de boel in elkaar schuiven zijn we voor dit probleem sneller klaar en met minder code.

Er zullen niet veel lezers zijn na het doornemen van dit artikel die niet gedacht hebben aan SOA (Service Oriented Architectures). We willen echter benadrukken dat de hier beschreven ontwerpmethodode volledig losstaat van SOA. We spreken dan wel over services die het domein faciliteren, maar dit heeft niets met SOA te maken. SOA zou echter wel een manier kunnen zijn om services en domeinen te koppelen (we hebben in de referentieapplicatie al een aanzet gegeven door de WPF GUI te ontsluiten met een WCF Service). Het verdiepen van de relatie van DDD en SOA benadering bewaren we echter voor een andere dag.

### Conclusie

Nu het hele proces de revue gepasseerd is van ontwerp en implementatie van domein tot en met de aansluiting tussen domein en services kunnen we stellen dat voor een relatief simpele applicatie als het aansturen van een robot een domeingedreven aanpak een tamelijk lijvige implementatie tot gevolg heeft. De echte meerwaarde ligt dan ook niet in speelprojecten als robotjes die over lijnen heen rijden, maar specifieke complexe bedrijfsprocessen die aan verandering onderhevig zijn over de tijd. Daarnaast biedt het dynamisch kunnen aan- en afkoppelen van services en adaptoren een groot pluspunt. Er is in het recente verleden veel geschreven over domeingedrevenheid en wat ons betreft is de koppeling van domein en buitenwereld vaak onderbelicht gebleven. We hebben drie mogelijke strategieën aangedragen om deze koppeling te realiseren en ieder van deze mogelijkheden heeft zijn eigen voor- en nadelen.

# Agenda 2007

CodeCamp - <a href="http://www.code-camp.nl">www.code-camp.nl</a>	12 mei
SDN: Software Developer Event	1 juni
Microsoft Tech Ed - USA <i>Orlando (USA)</i>	4 – 8 juni
Microsoft Developer Days <i>RAI, Amsterdam</i>	13 & 14 juni
SDN Magazine Nr. 94	17 augustus



**SDN: Software Developer Conference 2007**  
*Papendal, Arnhem* 17 & 18 September

SQL PASS, <i>Denver (USA)</i>	18 – 21 september
EKon - Entwickler Konferenz, <i>Frankfurt (D)</i>	24 – 28 september
Microsoft PDC 2007, <i>Los Angeles (USA)</i>	2 – 5 oktober
Visual FoxPro Konferenz, <i>Frankfurt (D)</i>	8-10 november
Microsoft TechEd Developers, <i>Barcelona (E)</i>	5 – 9 november
Microsoft TechEd IT Forum, <i>Barcelona (E)</i>	12 – 16 november
SDN Magazine Nr. 95	16 november
SDN: Software Developer Event	14 december

*Genoemde data onder voorbehoud.*

### Literatuur

1. Vens, R., <http://www.sepher.nl/downloads/Domain%20Centered%20Architecture.zip>
2. Evans, Eric, Domain Driven Design, Tackling Complexity in the Heart of Software  
Jimmy Nilsson, Applying Domain Driven Design and Patterns
3. Gamma, E. et al., Design Patterns, Addison-Wesley (1995)
4. Fokke, B., <http://nxtsharp.fokke.net/>



**André Boonzaaijer** is consultant / software engineer bij Sogyo. Zijn specialiteiten liggen in de Microsoft .NET hoek waar hij veel met onder andere SOA bezig is. Samen met een aantal enthousiaste collega's probeert André de theorie in de praktijk toe te passen met name op het gebied van OO en DDD.



**Edwin van Dillen** is principal consultant bij Sogyo. Hij is oprichter en eindverantwoordelijk voor het IT expert center. Naast het geven van advies over software ontwikkeling spreekt hij op bijeenkomsten en geeft hij trainingen over software architectuur. Tot zijn klantenkring behoren o.a. ANVA, Fortis ASR, ING, Postbank, ABN AMRO, Noad, Credit EuropeBank.