Domain-Driven Design

EEN PRAGMATISCHE GIDS

Software ontwikkeling met een focus op business

Inhoud

INTRODUCTIE 3

DOMAIN-DRIVEN DESIGN - WAT MAG JE VERWACHTEN? 5

SOFTWARE OP MAAT - DE TRADITIONELE PROBLEMATIEKEN 9

DE BASIS - DDD IN EEN NOTENDOP 15

- DOMAIN-DRIVEN DESIGN DRAAIT OM MODELERING 16
- DOMAIN-DRIVEN DESIGN VERTREKT NIET VANUIT TECHNISCHE SOFTWARE ARCHITECTUUR 16
- STRATEGISCH DESIGN BINNEN DDD 16
- LITERATUUR OVER DOMAIN-DRIVEN DESIGN 22

PRAKTISCHE TIPS - AAN DE SLAG MET DOMAIN-DRIVEN DESIGN 23

- WEET WAAROM JE VOOR DOMAIN-DRIVEN DESIGN KIEST 23
- BEGIN BIJ DE BASICS: BOUW DE NODIGE DDD KENNIS OP IN JE TEAM 23
- VOCABULARY (UBIQUITOUS LANGUAGE) IS JE BELANGRIJKSTE ASSET 25
- WAAK OVER EEN CORRECTE TOEPASSING VAN DOMAIN-DRIVEN DESIGN 26
- LAAT DDD NIET LOS TIJDENS BOUWEN EN TESTEN 26
- DURF REFACTOREREN BIJ NIEUWE INZICHTEN IN HET DOMEIN 27
- VERMIJD “ANALYSE TRACKS” 27
- WERK VISUEEL 28

DE WERELD ROND DDD - WAT IS ER NOG? 30

- HEXAGONALE ARCHITECTUUR 30
- EVENT STORMING 31
- CQRS 34
- EVENT SOURCING 36

CONCLUSIE 37

Introductie

Domain-Driven Design (DDD) is geen nieuw onderwerp. Al jaren zijn er fervente pleitbezorgers én critici binnen de software-ontwikkelwereld. Toch is DDD duidelijk een grote sprong voorwaarts in de eeuwige zoektocht naar betere manieren om niet-tastbare business concepten in nog meer ongrijpbare software-concepten te vertalen. Dit zorgt ervoor dat gebruikers, ontwikkelaars, beheerders en zelfs investeerders op korte én op lange termijn tevreden zullen zijn.

Domain-Driven Design is geen evident onderwerp. Er zijn heel wat mogelijke valkuilen bij toepassing in de praktijk. We hebben onze inzichten hierrond dan ook gebundeld, met de nodige zin voor realisme en pragmatisme. Het is geenszins onze bedoeling om een herkauwde versie van de theorie van DDD naar voor te schuiven.

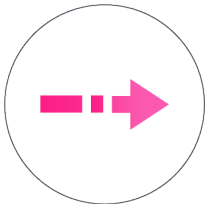
We hebben hier vooral een stukje ervaring en nuttige tips proberen te verzamelen die het verschil kunnen maken als je zelf aan Domain-Driven ontwikkelprojecten werkt of van plan bent dat te gaan doen. Hopelijk inspireert het je om DDD verder in detail te verkennen, of helpt het je om de complexe materie beter in de praktijk vertaald te krijgen.

Hoofdstuk 1

De verwachtingen

Domain-Driven Design - wat mag je verwachten?

Laten we met een beeld van het ideale eindresultaat voor ogen beginnen: wat mag je verwachten als je een geslaagd DDD project realiseert m.a.w. wat is de belofte van DDD die de investering in een nieuwe werkwijze voor software ontwikkelingsprojecten de moeite waard maakt?

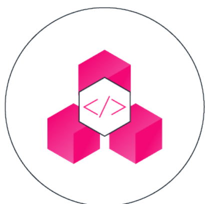


DDD maakt een voorspelbaar proces van functioneel en technisch design van software.

De eerste twee vragen die bij elk ontwikkelingsproject gesteld worden, zijn de moeilijkste: “Wat gaat het kosten?”, en “Wanneer is het klaar?”. DDD biedt uiteraard geen magische oplossing, wel laat het toe om op zeer korte termijn en met zeer beperkte investering een goed beeld te krijgen op het gehele traject. Daarbij niet onbelangrijk je ervan te vergewissen dat iedereen hetzelfde beeld in z’n hoofd heeft.

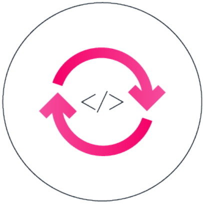
DDD is architectuur- en technologie-agnostisch.

De oplossingen zijn transparanter voor business én beter bestand tegen digitale erosie. De meerwaarde hiervan is niet te onderschatten: bij het initiëren en uitwerken van het project kan je de focus leggen op de business kant in plaats van jezelf te verliezen in eindeloze technische discussies en avonturen. Meer nog: je kan een behoorlijk eind opschieten in de realisatie zonder al definitieve (en eventueel in vroeg stadium niet te onderbouwen) architectuur- en technologie-keuzes te moeten maken. Op lange termijn is het resultaat een stuk software dat zich veel makkelijker laat onderhouden, kan evolueren en zelfs migreren naar nieuwe technologie-stacks dan wat je in een klassiek project gewoon bent.



DDD vertrekt vanuit een modellering van business processen en niet vanuit technologie perspectief.

De focus moet liggen op de primaire activiteiten zoals het opstellen van een domein model en decompostitie van de functionele architectuur. Dit zorgt voor een structurering van de software op hoog (architectuur) en laag niveau (design) dat helemaal gealigneerd is met de functionele business kant. Daardoor is het onder andere ook makkelijker om ownership van elk onderdeel te bepalen, wat bij wijzigingen aan business kant een lokalere en voorspelbare impact op de software genereert.



DDD laat een snelle en transparante ontwikkeling toe.

Het leidt onder meer tot vlot onderhoud en refactoring in het business domein met een voorspelbare en behapbare impact bij elke wijziging. Doordat de essentie van de software technologie- en framework-agnostisch wordt gerealiseerd, is de development cycle zeer optimaal: zonder een te zware rugzak aan technologie-dependencies kan je concepten snel implementeren en in eenvoudige unittesten de werking van het domein model demonstreren, aftoetsen en documenteren. Pas nadien zullen alle technologische integraties (databases, messaging, API's) en aspecten (transacties, deployment, packaging, …) toegevoegd worden aan de reeds gevalideerde basis.

DDD biedt een fundamenteel inzicht in het businessdomein aan het hele team vanuit groepsleren, met een hoge betrokkenheid van alle teamleden.

Het domein modeleren is niet alleen een activiteit die een tastbaar eindresultaat oplevert maar ook, en vooral een leerproces. Hoe zit de business nu echt in elkaar? Hoe verhouden concepten zich? Welke logica zit erachter? We herkennen het allemaal: business vertelt aan functioneel analyst, functioneel analyst specifieert naar development, development spreekt af met operations. Onvermijdelijk botsen we op communicatiefouten en mis-interpretaties door ontbrekend inzicht in de overkoepelende materie. Door DDD toe te passen ga je hier in groep het hele domein verkennen en iteratief corrigeren met nieuwe inzichten. Dit levert een gigantisch voordeel op gaandeweg het project en nadien in onderhoud en evolutie. DDD maakt een voorspelbaar proces van functioneel en technisch design van software.

De eerste twee vragen die bij elk ontwikkelingsproject gesteld worden, zijn de moeilijkste: “Wat gaat het kosten?”, en “Wanneer is het klaar?”. DDD biedt uiteraard geen magische oplossing, wel laat het toe om op zeer korte termijn en met zeer beperkte investering een goed beeld te krijgen op het gehele traject. Daarbij niet onbelangrijk je ervan te vergewissen dat iedereen hetzelfde beeld in z’n hoofd heeft.

Is DDD dan een “silver bullet” oplossing voor alle problemen en risico’s verbonden aan software ontwikkeling? Uiteraard niet, er zijn een aantal klassieke struikelblokken waar teams al eens tegenaan durven lopen.

- Zo bestaat er vaak veel **verwarring rond de concepten die aan de basis van DDD liggen**, zeker in de Java-wereld waar nagenoeg elk DDD concept al een andere betekenis of invulling heeft (domein model,repository,entity,value object,business event etc). Dit maakt dat uitgerekend in de Java-wereld DDD vaak fout loopt, vanwege misinterpretatie van deze concepten, hoe er mee om te gaan, en hoe ze te implementeren. Een fundamenteel begrip van DDD en z’n concepten bij alle teamleden is onontbeerlijk voor een succesvol project.

- Er wordt ook vaak **op basis van enkele aspecten geclaimd om DDD te werken**, terwijl de meest fundamentele onderdelen van DDD niet aangeraakt worden. Vaak gaat het dan om de concrete en zeer tastbare details van tactisch design, terwijl het o zo belangrijke strategisch design over het hoofd gezien wordt. Dit maakt dat men nooit de meerwaarde zal realiseren die DDD zou kunnen leveren, en mogelijk zelfs een complexer traject tegemoet gaat dan in een klassieke benadering.

- Tenslotte zie je ook wel **individuele efforts rond DDD** die - hoe nobel of visionair ze ook zijn - gedoemd zijn om te mislukken. Dit komt net omdat DDD een allesomvattend werkkader biedt dat toelaat aan alle betrokkenen om op basis van een aantal welomlijnde concepten aan de slag te gaan en te communiceren. Indien dit beperkt blijft tot enkele mensen binnen een team dan blijft het resultaat gegarandeerd ondermaats.

Hoofdstuk 2

Op maat gemaakt

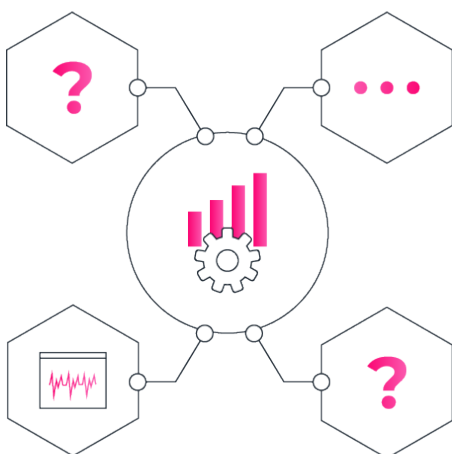
Software op maat - de traditionele problematieken

Met een leeg blad beginnen en je eigen software opbouwen in plaats van een off-the-shelf pakket implementeren, redenen genoeg te bedenken waarom dat een goed idee is. Vooral op de domeinen waar je je net iets anders wil organiseren dan de andere spelers in de markt, of op domeinen die nog niet de maturiteit hebben om de commodities in een standaard pakketje te vinden.

Er zijn oneindig veel mogelijkheden: wat je kan bedenken, kan je ook bouwen. Er zijn altijd voldoende opportuniteiten rond de nieuwste technologieën en het is meestal geen enkel probleem om hiervoor een enthousiaste ploeg bij elkaar te krijgen, klaar om er in te vliegen.

Weinig activiteiten zijn echter complexer dan het bouwen van software op maat en de valkuilen die je project, je product of je business case kunnen doen falen, zijn legio. Al te vaak is de perceptie dat software ontwikkeling traag en duur is, daar getuige de vele horrorstories rond giga-projecten die maanden of zelfs jaren te laat opleveren en meermaals over budget gaan. En terwijl iedereen heel gefocust naar een go-live datum toewerkt, bevindt het grootste deel van de levenscyclus van een stuk software zich na de go-live in support en evolutie, een moment waar je pas echt de vruchten plukt - of de rekening gepresenteerd krijgt - van beslissingen die veel vroegen genomen werden.

We zetten enkele symptomen van een problematisch ontwikkeltraject op een rijtje en lichten ook al een tip van de sluier hoe DDD deze problematieken vermijdt. Elk van deze symptomen zijn een teken aan de wand dat je project misschien de verkeerde richting uit gaat, of minstens toch moeilijk onder controle te houden zal zijn. Allicht herken je er zelf wel enkele van?



Een “Sprint nul” die geen business waarde oplevert.

De typische sprint nul, die de start van het project vormt, moet zich focussen op team- en projectwerking. Maar al te vaak zien we dat men zich direct al richt op IT- technische opzet. In de praktijk levert dit nog geen business value op - in tegenstelling tot elke oprechte agile-ambitie en MVP-gerichte projectwerking. Teveel project zorgen letterlijk op dag één voor een kloof tussen business en development, die verderop in de projectwerking slechts met heel veel moeite verder beperkt zal kunnen worden.

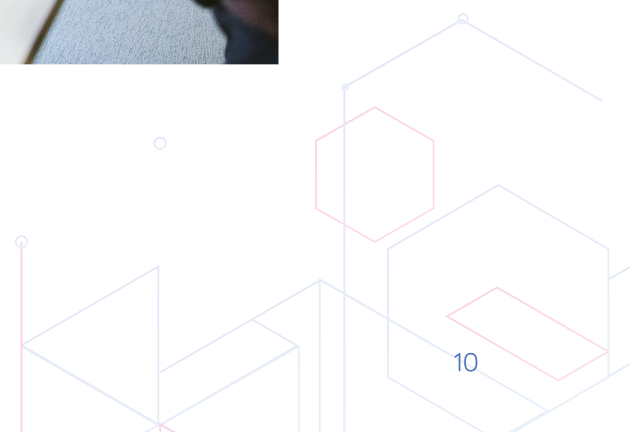
Domain-Driven Design schrijft een extreem business-gerichte focus voor, welke toelaat om de opstart van je project enkel op de pure business value toe te spitsen. Geen technologie-setup, maar een eerste versie van het business domein en een oplossing voor de vraagstukken die daarin relevant zijn.

Een vertaalslag tussen de business-taal van het project en de development-taal.

Een allicht herkenbaar problematisch fenomeen in software ontwikkeling is de verschillende set termen en concepten die gehanteerd worden bij bespreking met business en tijdens development. Developers maken als het ware een vertaalslag naar een technische wereld, die het onmogelijk maakt voor niet-technische stakeholders om de activiteiten van de ontwikkelploeg echt te volgen en nuttige input, correcties of inzichten aan te dragen.

De vertaalslag van analyses, user stories, functionele testen etc. naar technologische concepten door het development team maken dat er een gap ontstaat tussen de code en terminologie die daar gehanteerd wordt tegenover het business jargon dat gangbaar is. Niet enkel de bijhorende vertaalslag is problematisch, maar zeker ook het feit dat deze het onmogelijk maakt om de impact van wijzigingen te voorspellen.

Domain-Driven Design met z'n technologie-agnostische concepten vermijden dat deze vertaalslag nodig is en houdt iedereen aangesloten tijdens het hele projectverloop. Dit garandeert dat iedereen goed kan begrijpen welke beslissingen genomen worden, inschatten welke gevolgen die hebben en daarop heldere input te geven.

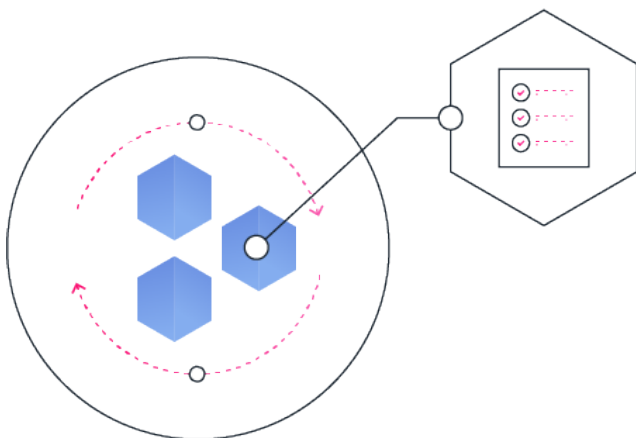


Het databasemodel van de toepassing wordt “set in stone” en een echte handicap naar evolutie toe.

Uiteraard is een gepast informatiemodel - met de nodige concepten, relaties, attributen, datatypes en dergelijke onontbeerlijk voor een kwalitatieve implementatie. Vaak is het ook de enige houvast die door alle betrokkenen begrepen en herkend wordt. Het beperkt de ambities om het systeem vorm te geven of evoluties later bespreekbaar te maken.

Een te verregaande rol voor een, vaak zeer statisch model, schuift de vormgeving van de werkelijke business waarde en processen, functionaliteiten en logica vooruit. Of het informatiemodel hiervoor de gepaste vorm heeft, zal pas veel later blijken onder de vorm van complexe en trage ontwikkelingen.
Uiteindelijk is de eindsituatie te vaak dat het informatiemodel niet langer de (immer wijzigende) business reflecteert, maar de ooit geschikt geachte informatie-structuur van het digitaal product, die verdere ontwikkeling en onderhoud, eerder complex maakt dan vlot ondersteunt.

Domain-Driven Design vermijdt het modeleren van een puur informatie-gericht model, wat ook wel eens een “anemic domain model” wordt genoemd, maar vertrekt van de functionaliteiten die - uiteraard - ondersteund worden door informatiestructuren. Deze worden ook in hun eigen levenscyclus gespecifieerd en krijgen uiteindelijk hun rol in het gehele model.



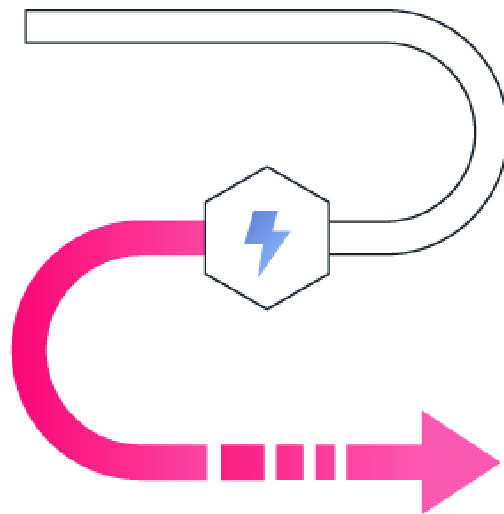
Paniek voor wijzigingen en zelfs weerstand naar verandering toe

Dit is nooit een probleem in de beginfase, maar vaak een blokkerende factor verderop in de bouw en tijdens onderhoud: “zo werkt het systeem niet”, of “als we dat moeten aanpassen, moeten we het halve systeem herbouwen”.

Business evolueert, software dus ook, en indien die daar niet op voorzien is, wordt het een moeilijke en risicovolle onderneming. Er ontstaat veel druk op en onbegrip naar de ontwikkelploeg toe. Deze staat dan ineens heel ver af van het enthousiasme waarmee het project ooit begon. Developers vertrekken, kennis loopt buiten en de “mental picture” van het systeem blijft niet achter in de organisatie. Een nieuwe legacy applicatie is ontstaan.

Domain-Driven Design zorgt dat iedereen dezelfde mental picture rond het project heeft en dicteert een strikte implementatie van dit gedeelde beeld. Enerzijds zorgt dit voor meer transparantie en inzicht in de materie, waaruit dan wederzijds begrip ontstaat.

Anderzijds genereert het vooral een voorspelbare impact bij wijzigingen. Grote wijzigingen in functionaliteiten zullen nog altijd een stevige impact op de gebouwde oplossing hebben, maar dit zal logisch en proportioneel zijn. Kleine wijzigingen in functionaliteiten zullen kleinere gevolgen hebben. Dit alles zorgt ervoor dat wijzigingen ook effectief zullen gebeuren, in plaats van een snelle en slim ogende hack die het probleem telkens alleen maar groter maakt.

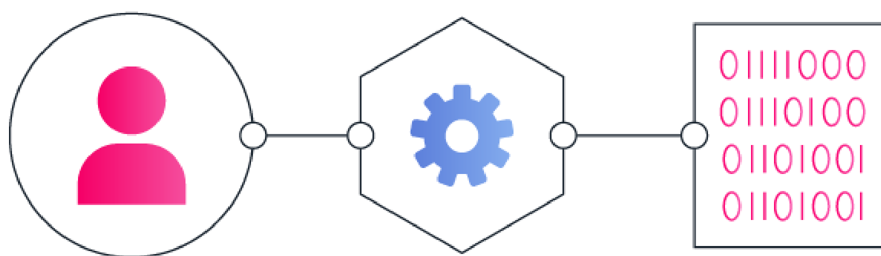


Moeilijke team retentie en onboarding

In de ideale wereld is onderhoud aan software een beperkte en eenvoudige opdracht. In praktijk is te vaak het omgekeerde waar. Steeds complexer wordende projecten vereisen teamleden met heel wat ervaring, die van alle markten thuis zijn en ontwikkelaars met 101 skills zijn nodig om aan de applicatie te kunnen werken. De zoektocht naar het spreekwoordelijke vijfpotige schaap wordt steeds moeilijker en duurder. Als deze dan gevonden wordt, volgt er een te lange inwerktijd alvorens verbeteringen kunnen worden opgeleverd.

Domain-Driven Design gaat uit van een (technisch alvast) eenvoudige representatie van het business domein in applicatie-code, die dan ook nog eens één-op-één het gemeenschappelijk begrip (domein model) representeert. Deze “functionele” code wordt in een technische omgeving gebracht die verantwoordelijk is voor aansluiting en integratie met andere componenten, opslag en messaging etc.

Door deze mooie scheiding is het perfect mogelijk dat sommige mensen zich inwerken in het domein an sich, terwijl andere zich - zelfs zeer tijdelijk - toeleggen op specifieke technologische integraties, zonder zich in de volledige functionele draagwijdte van het systeem te moeten inwerken.



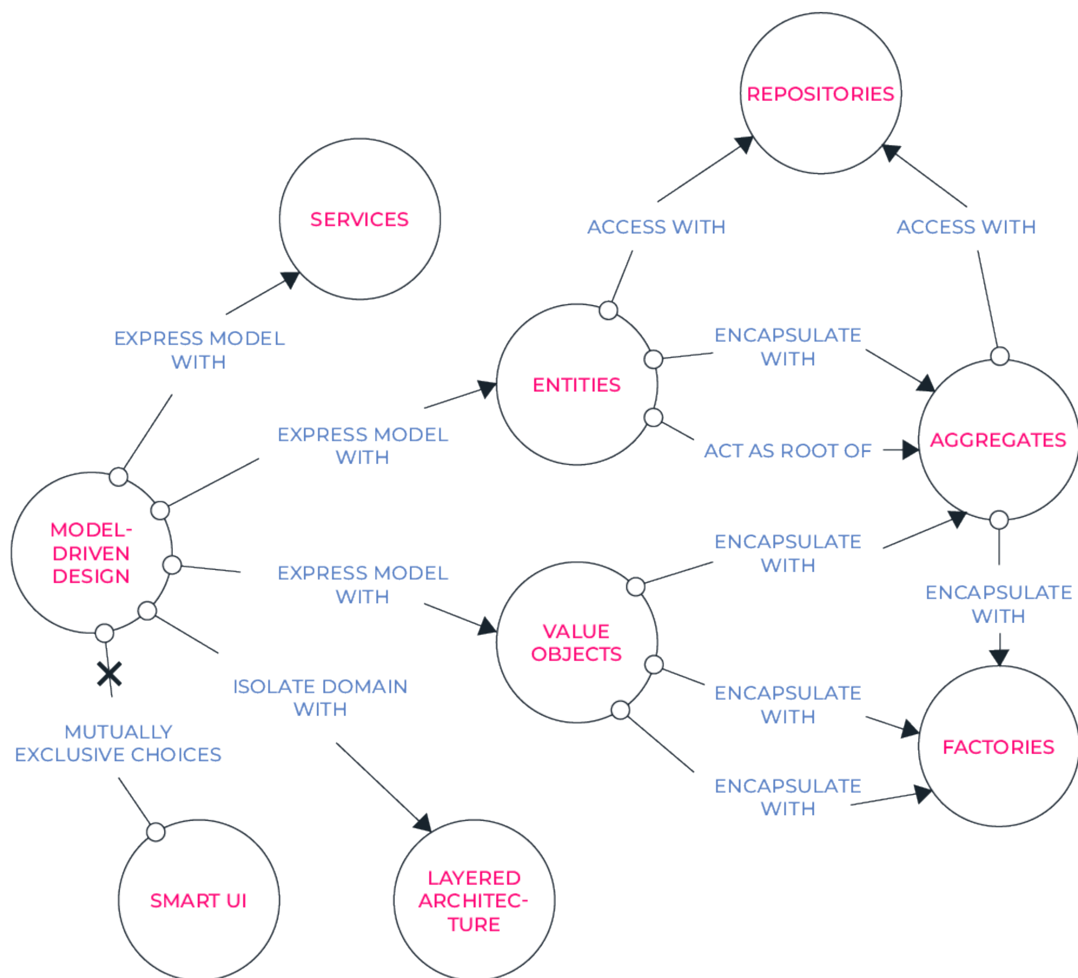
Hoofdstuk 3

De basis

De basis - DDD in een notendop

Domain-Driven Design is op het eerste zicht een evidente materie, maar vergt in praktijk toch vaak meerdere pogingen, gesprekken, inzichten en verduidelijkingen alvorens je de hele filosofie juist aanvoelt én in de praktijk omgezet krijgt op een manier dat je er effectief de vruchten van plukt.

Als dit je eerste kennismaking met Domain-Driven Design is, hopen we je op basis van onze eigen inzichten een sneller parcours te bieden om de finesses van DDD te doorgronden. Onderaan dit hoofdstuk verwijzen we je naar de belangrijkste bronnen om je in te lezen. Indien je de basisterminologie al beheerst of al praktijkervaring hebt met DDD, hopen we hier enkele extra elementen aan te reiken die je begrip rond DDD verder kunnen verrijken.



Alvorens een overzicht te geven van de onderdelen en werking van DDD met strategisch en tactisch design, willen we twee factoren onder de aandacht brengen.

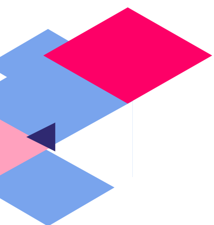
Domain-Driven Design draait om modelering

Een eerste belangrijke factor om rekening mee te houden is dat de **juiste terminologie en concepten zeer centraal staan én zeer belangrijk zijn in Domain-Driven Design**. De concepten vormen als het ware een niet-technisch vocabularium om software in uit te drukken, wat dan ook goed gekend moet zijn in je team dat er gebruik van gaat maken. **DDD draait om modelering**, wat betekent dat iets niet per se 'juist' of 'fout' is, maar dat de ene benadering of modelering allicht 'beter geschikt' is dan de andere (indachtig de quote van George Box - "All models are wrong, but some are useful").

Het is de kunst om op zoek gaan naar het meest geschikte model voor de software die je voor ogen hebt, niet te simplistisch, maar zeker ook geen overengineering.

Domain-Driven Design vertrekt niet vanuit technische software architectuur

Een tweede factor is dat software architectuur uiteraard nodig is, maar in Domain-Driven Design een heel andere (en minder prominente) plaats krijgt in de aanpak. **Eerst wordt het model als het ware als een digital twin van de te digitaliseren business uitgewerkt en daarna wordt deze in een IT-technische software architectuur gebracht.** Dit maakt dat units of deployment e.d. bijna puur operationele beslissingen worden die weinig tot geen invloed hebben op de opbouw van de codebase. DDD vormt op deze manier de basis voor zowel eenvoudige als large-scale architecturen én garandeert een vlotter migratiepad van het eerste naar het tweede, met beperkte impact.



Strategisch design binnen DDD

Wellicht het belangrijkste maar vaak onderbelichte alsook minst makkelijk begrepen deel van Domain-Driven Design is het "Strategisch Design".

Strategisch Design focust op de decompositite van het hele systeem in kleinere onderdelen, met een duidelijke samenhang tussen deze onderdelen. Elke projectaanpak voor een softwareontwikkeling van enig formaat gaat uiteraard op zoek naar een opdeling van het grotere probleemdomein in kleinere onderdelen. Deze vormen de basis voor inzichtelijkheid, parallelle ontwikkeling, en heel wat andere kwalitatieve factoren.

CONTINUOUS INTEGRATION

KEEP MODEL UNIFIED BY

SHARED KERNEL

BOUNDED CONTEXT

OVERLAP ALLIED CONTEXTS THROUGH

NAMES ENTER

CUSTOMER / SUPPLIER TEAMS

ASSESS / OVERVIEW RELATIONSHIPS WITH

ISOLATE RELATE ALLIED CONTEXTS ASDOMAIN WITH

UBIQUITOUS LANGUAGE

CONTEXT MAP

OVERLAP UNILATERALLY AS

CONFORMIST

SUPPORT MULTIPLE CLIENTS THROUGH

TRANSLATE AND INSULATE UNILATERALLY WITH

FREE TEAMS TO GO

OPEN HOST SERVICE

FORMALIZE AS

PUBLISHED LANGUAGE

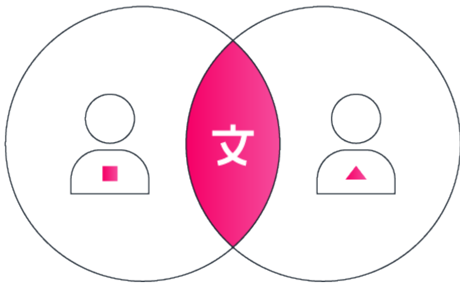
ANTICORRUP-TION LAYER

SEPARATE WAYS

Strategisch Design kan je in feite zien als de ‘juiste’ versie van functionele + technische architectuur. Het expliciteren van strategisch design dwingt je in een vroege fase strategische vragen te beantwoorden vanuit een business redenering, die bepalend zullen zijn voor de verdere uitwerking.

In een klassieke aanpak wordt deze beslissingen uiteraard ook genomen, maar vaak onbewust, te laat of op basis van de foute factoren.

We lichten kort de belangrijkste concepten uit het strategisch design toe:

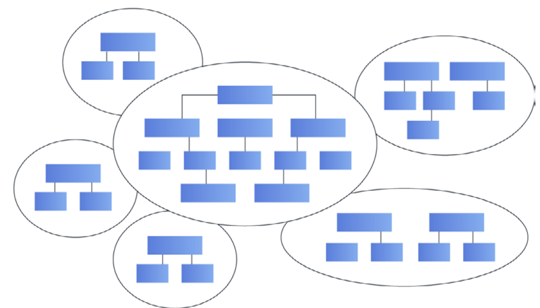


Ubiquitous Language

Fundamenteel in een DDD project is dat er een eenduidige, door iedereen begrepen én gehanteerde taal ontstaat, in lijn met de gangbare business spoke. Deze Ubiquitous Language zorgt ervoor dat kennis van het business domein (en dus ook de software) diep opgebouwd wordt in het hele team. Het vermijdt dat er een vertaalslag naar development dient te gebeuren. Bij nieuwe inzichten of correcties op de Ubiquitous Language volgt sowieso een refactoring om de codebase dit nieuwe inzicht te laten weerspiegelen.

Subdomains

Een subdomain is een onderdeel van het grotere systeem, maar niet op de manier die je misschien kent vanuit microservices of een andere IT-technische architectuur. Wel is het een natuurlijke opdeling bepaald door hoe de business werkt (typisch in lijn met communicatiestructuren). Deze decompositie zorgt voor een verregaande alignment tussen business en IT die op basis van een klassieke technische architectuur-decompositie nooit bereikt wordt. Business evolutie veroorzaakt op deze manier een voorspelbare impact op de software en geeft duidelijk aan waar de prioriteiten en business value liggen.



Subdomains worden ook expliciet benoemd als “core”, “generic” of “support” domains, wat ineens weergeeft of ze key zijn in de business aanpak en een competitief voordeel kunnen opleveren. Door deze benoeming kunnen we elk domein de passende prioriteit en aandacht geven. Tijdens deze benoeming gaat men zorgvuldig een handvol coredomains bepalen. Door deze types expliciet toe te kennen aan de subdomains, in overleg met alle stakeholders, wordt de projectfocus kraakhelder. Typisch zien we dat hoge budgettten en developers met veel ervaring toegekent worden aan de coredomains. Elk coredomain wordt steeds volledig op maat gemaakt.

Daarnaast wordt ook duidelijk wat competitive advantage en commodity is én kunnen beslissingen rond bouwen versus. huren, configureren of kopen objectiever genomen worden.

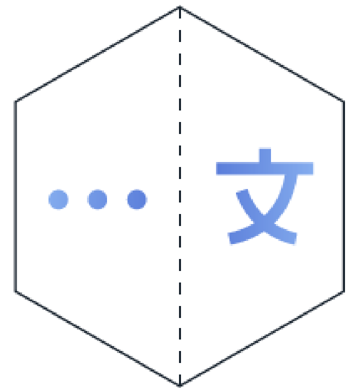
Bounded Context

Een belangrijk maar vaak minder begrepen concept is de Bounded Context. Elk subdomein kan 1 of meer van zulke bounded contexts bevatten, maar een bounded context overspant zelden meerdere subdomains. **Deze verdere opdeling van subdomains in Bounded Contexts is subtiel maar belangrijk naar de Ubiquitous Language toe, aangezien een Ubiquituous Language altijd geldig is binnen één bounded context.**

Dit betekent dus dat een woord een eenduidige betekenis heeft binnen één Bounded Context.

Als de boekhouder spreekt over een 'klant' gaat het allicht om een partij waar al ooit een factuur werd naar opgemaakt, terwijl de verkoper na een beloftevol prospectiegesprek al durft spreken over een 'klant' en de helpdesk over een 'klant' spreekt zolang er een lopend servicecontract is.

Deze aanpak met een terminologie per bounded context is één van de sleutelverschillen tussen DDD en een geïntegreerde modelering (database model, klassiek domein model, ...), waar een concept vaak overladen wordt met meerdere aspecten en rollen, wat voor een conceptuele monoliet zorgt waarbij de impact van wijzigingen vaak onnodig groot is en het model onnodig complex.



In DDD vormt de bounded context voor een beperkte scope van werkzaamheden en begrip - dit garandeert een makkelijkere onboarding van nieuwe teamleden, minder technische dependencies binnen de software zelf en meer mogelijkheid tot subteams die echt parallel development uitvoeren.

Context Maps

Het laatste belangrijke concept binnen strategisch design is de **Context Map, die de relaties tussen bounded contexts en domeinen onderling expliciteert.**

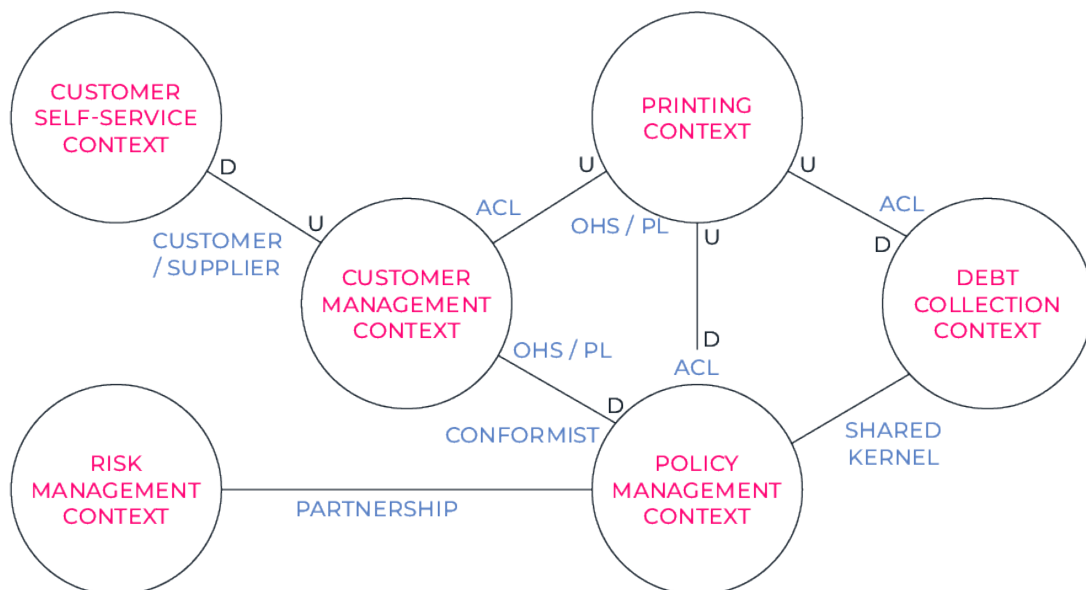
DDD voorziet een hele reeks mogelijkheden, die verbazend bekend zullen voorkomen als integratie-patronen, maar bijna nooit op zo'n niet-technisch strategisch niveau benoemd en beslist worden.

Bijvoorbeeld de 'shared kernel' linkt twee domeinen door het gebruik van gedeelde concepten (en technisch straks dus allicht een shared library met deze concepten erin), wat hergebruik optimaliseert, maar de twee domeinen wel zeer sterk aan elkaar verbindt en wijzigingen quasi altijd beide laat impacteren. Een ander belangrijk concept bij het opstellen van een relatie tussen twee contexten is de richting: 'upstream' of 'downstream'.

Een vaak voorkomende combinatie in bijvoorbeeld een microservice architectuur is 'Open Host Service' als upstream in combinatie met een "Anti-corruption layer" waarbij "published language" zorgt voor een gedocumenteerde en gedeelde taal. De Open host service zorgt hierbij voor een generieke set van één of meerdere services die aan de downstream kant vrij geïmplementeerd kunnen worden. De anti-correption layer isoleert het client model van andere modellen door middel van een extra vertaalslag. Dit zorgt voor een mooie scheiding tussen een integratie laag of adapter en de bounded contexts van het model.

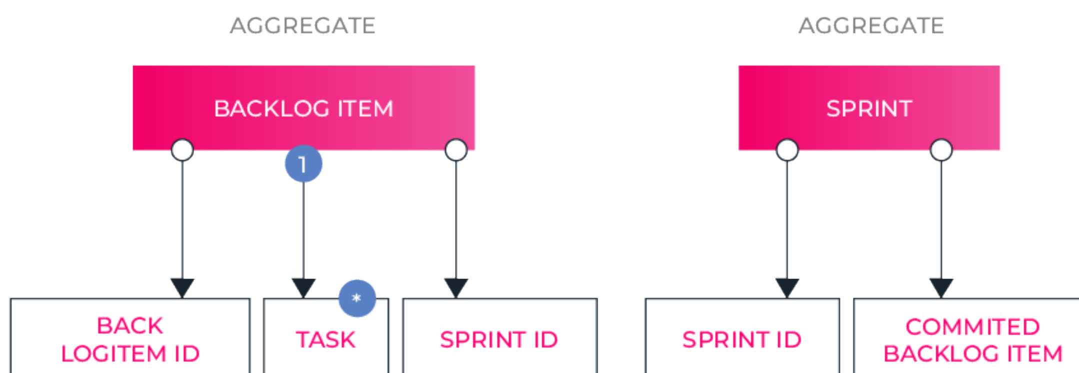
Een Context Map is dan ook de ‘juiste’ manier om integratie-patronen correct te kiezen op basis van business afwegingen en niet op basis van techniciteiten. Deze beslissingen zullen ook de mogelijkheden rond systeemarchitectuur verder bepalen: opsplitsing vanmogelijke microservices of expliciete keuze rond shared libraries.

Strategisch Design zorgt binnen Domain-Driven Design voor belangrijke kwalitatieve aspecten zoals ontkoppeling en onafhankelijkheid, om de nodige souplesse in de architectuur te houden waar gewenst.



Tactisch Design binnen DDD

Het Tactisch Design is vaak een stuk bevattelijker én bekender dan Strategisch Design. Tactisch Design bepaalt het interne ontwerp of de opbouw van één bounded context. Je zou het kunnen zien als de ‘juiste’ manier om een technisch design te maken, maar dan uiteraard op de DDD-manier: verder bouwend op de beslissingen van het Strategisch Design én in complete samenwerking en transparantie vanuit het multi-disciplinaire projectteam.



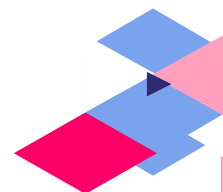
Tactisch Design bepaalt de interne structuur van de software per bounded context, maar doet dit op een technologie-agnostische manier. Dit laat toe om technologie-keuzes uit te stellen tot het zeer duidelijk is wat er gebouwd zal worden en om welke redenen. Bijvoorbeeld om een functionele proof-of-concept te bouwen zonder technologie-overhead waarbij we later een vlottere technologie-migratie met beperkte impact kunnen uitvoeren.

DDD bepaalt een beperkt aantal bouwstenen waarin het Tactisch Design van een bounded context wordt uitgedrukt: **Entities, Value Objects, Domain Services, Repositories, Factories en Aggregates.**

Deze bouwstenen van het tactisch design zijn heel concreet en vaak bevattelijker dan die uit het strategisch design, maar meestal niet zo evident om fundamenteel te begrijpen en juist in te zetten. Het merendeel van deze termen heeft reeds een beladen terminologie: al deze woorden hebben namelijk ook een (gerelateerde maar minstens genuanceerde) andere betekenis in de Java wereld. Denk maar aan een “@Service” uit Spring Framework, een “Entity” uit JPA, een “Repository” uit Spring Data, etc. De veronderstelling dat deze concepten dezelfde betekenis, nuances en toepassingsgebied hebben, is dan ook de grootste valkuil voor een beginnend DDD ontwikkelaar of team.

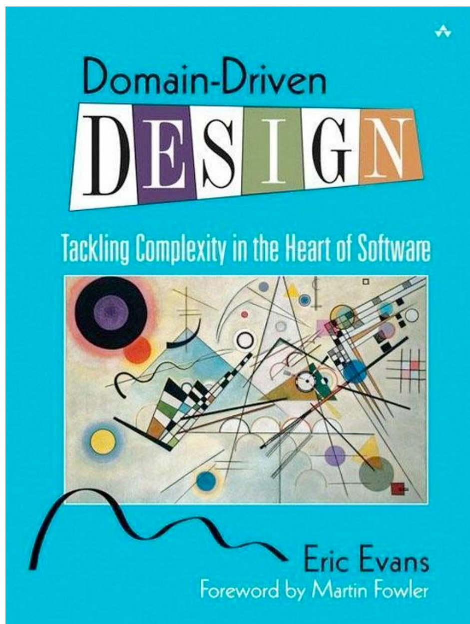
Het allicht belangrijkste inzicht om rond Tactisch Design mee te geven is dat het misschien het meest bevattelijke en begrijpbare deel van DDD is, maar nooit het begin van toepassing in een project mag zijn!

Over tactisch design is veel info te vinden, maar het wordt vaak te snel het onderwerp waar een team te veel en te vroeg op gaat focussen, terwijl de kwalitatieve aspecten op lange termijn vooral voortkomen uit strategisch design. Hoe goed tactisch design ook wordt uitgevoerd, het zal weinig meerwaarde bieden in een fout gedefinieerde bounded context.



Literatuur over Domain-Driven Design

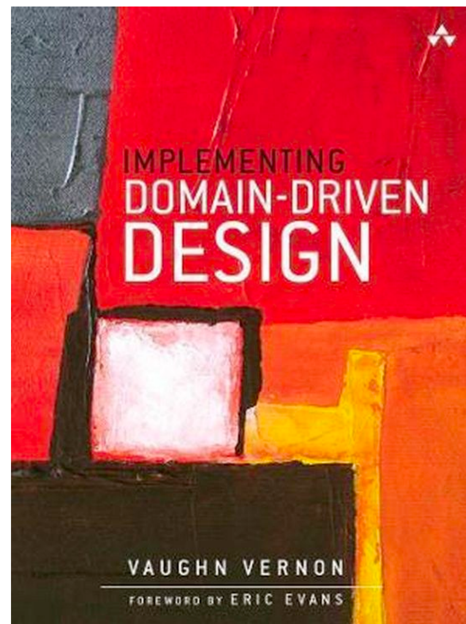
Aangezien het absoluut niet onze ambitie is om hier een compleet overzicht in de diepte te geven rond DDD, verwijzen we graag door naar de vele online resources die hierrond te vinden zijn, alsook uiteraard de twee zeer aan te bevelen referentie-boeken rond dit topic:



Domain-Driven Design

Eric Evans

Het originele werk dat het DDD topic op de kaart zette.



Implementing Domain-Driven Design

Vaughn Vernon

Wat een praktische vertaalslag naar een technische implementatie beschrijft, alsook enkele nieuwere concepten zoals 'Business Events' toevoegt.

Praktische tips - Aan de slag met Domain-Driven Design

Graag geven we een aantal praktische tips mee voor als je zelf aan het experimenteren slaat met Domain-Driven Design. Waar moet je op letten om het maximum uit je project en je team te halen?

Weet waarom je voor Domain-Driven Design kiest

Gebruik DDD om de juiste redenen en in de juiste omstandigheden:

- DDD is geen Rapid Application Development (RAD) alternatief.
- DDD is zeker ook geen toveroplossing voor een software engineering probleem. DDD toepassen vereist commitment, discipline en het moet een bewuste keuze zijn om zo te werken, gedragen door alle stakeholders.
- DDD is ook geen afterthought, het kan niet halverwege een projectverloop als oplossing ingezet worden om het een en ander vlotter te laten lopen. Uiteraard kan een bestaand project wel richting DDD bijgestuurd worden, maar DDD is zeker geen quick-fix want ook het strategisch design zal bijvoorbeeld moeten gebeuren, met mogelijk een stevige nieuwe visie op de projectopbouw en organisatie tot gevolg. Een investering in de nodige refactoring zal nodig zijn om reeds gebouwde puzzelstukken in de uiteindelijke bounded contexts in te passen.

Begin bij de basics: bouw de nodige DDD kennis op in je team

DDD is een aanpak die de input van het hele multidisciplinaire team vereist, inclusief en niet in het minst ook business! Zorg er dus voor dat iedereen op z'n minst de nodige basiskennis kan opbouwen rond de concepten en principes van Domain-Driven Design, zij het in zelfstudie of door voldoende tijd in de projectwerking en workshops te voorzien om alle concepten en de redeneringen daarachter uit te leggen en te laten inoefenen. Een projectwerking waarin slechts enkele mensen in het team DDD denken, werkt niet.

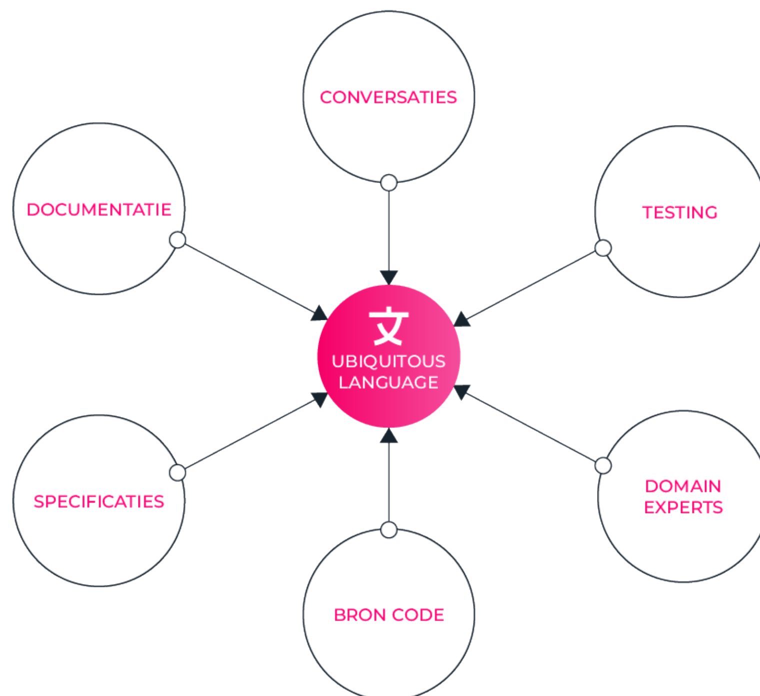
Start dus altijd met de huidige kennis rond DDD van het team te peilen en bij te spijkeren waar en indien nodig. Geef het team de tijd en ruimte om zich voldoende vertrouwd te maken met de materie. De eerste event storming workshops en modeleringen kunnen hiervoor perfect als praktijkvoorbeelden dienen, voorzie gewoon wat extra tijd voor briefing, reflectie, en eventueel herhalen van de oefening om met nieuwe inzichten het resultaat te herbekijken en te verbeteren.

Vocabulary (ubiquitous language) is je belangrijkste asset

Een uitgebreide collectie schema's of andere deliverables zijn niet het primaire asset van een DDD project, maar wél de gedeelde taal: vocabularium dat duidelijk begrepen wordt door het hele team, en dat de ruggengraat vormt van alle verdere conversaties, verkenning en discussies, en uiteindelijk ook broncode.

Duidelijke concepten en terminologie die als referentiekader of bouwstenen echt toelaten om business én IT te laten spreken over dezelfde materie, en die toelaat om in een zeer vroeg stadium fouten of verwarring te identificeren, makkelijker zekerheid te krijgen of het juiste gebouwd wordt én sneller bij te sturen indien dat niet zo blijkt te zijn.

Deze ubiquitous language wordt gehanteerd vanuit alle disciplines. Ook vanuit QA, die binnen een bounded context gaat afchecken of de software er mee in lijn is. Zij bewaken de kwaliteitsborging rond de heldere definitie, het consistente en correcte gebruik van de ubiquitous language.



Indien tactisch design goed uitgevoerd wordt, gebeurt documenteren automatisch als onderdeel van het coderen; glossary-pages e.d. zijn binnen DDD eigenlijk een anti-pattern omdat daarbij het risico wordt geïntroduceerd van een gap tussen (het recentste begrip van) de language en de software zelf.

Let dan ook altijd goed op correct gebruik van deze ubiquitous language, verbeter elkaar: praat altijd "in context". Werp vragen op als anderen alternatieve of net dubbele naamgevingen of synoniemen in of over contexts heen gebruiken, vaak zijn dit sleutelmomenten naar een dieper begrip van het domein!

Waak over een correcte toepassing van Domain-Driven Design

De meest gangbare fouten die gemaakt worden bij een oprechte poging om DDD te werken, ga je al vrij snel leren herkennen: wees er dan ook aandachtig voor en werp die telkens op als een project-issue en leermoment voor het ganse team:

- Modelering is de sleutel tot een goede software ontwikkeling, de rest is bijna bijzaak of volgt daar uit. Als je ziet dat er te weinig tijd naar het itereren en valideren van een uniform domein model gaat en dat er te snel naar de technische oplossing wordt gekeken, bestaat het risico dat het domein onvoldoende doordacht of begrepen is hierdoor gaat de gebouwde oplossing helemaal niet stand zal houden.

- Wees alert voor een overdreven focus op enkel het tactical design, maar ook voor een correct strategisch design dat landt bij development team waarin te weinig DDD kennis aanwezig is. Zo'n team zal allicht een IT-technische vertaalslag geven aan alle output uit het strategisch design die de DDD aanpak ondergraaft. Succes zit hem in het bewaken van een evenwicht tussen tactisch en strategisch design.

- Gebruik elk concept altijd in de juiste context: een 'klant' voor de boekhouding is helemaal iets anders dan een 'klant' voor sales. Dit kan zeer moeilijk zijn voor mensen die over bounded contexts heen moeten werken. Deel eventueel je teams op in lijn met je bounded contexts om te vermijden dat mensen in hun hoofd een mentale "context switch" moeten maken.

- Wees dubbel voorzichtig in een Java team met de veronderstelling dat iedereen mee is, DDD landt namelijk net iets moeilijker in de Java wereld wegens heel wat 'beladen' woorden: Repository, Entity, … die daar al een andere betekenis hebben. DDD is als het ware OOP++ (terug naar de originele ambities van object oriëntatie, iets wat gaandeweg scheefgelopen is of gekaapt is door frameworks/technologie die een te naïeve 1-op-1 mapping van OOP concepten op hun tech propageerden)

Laat DDD niet los tijdens bouwen en testen

Eens aangekomen in de echte comfort zone van het development team zal blijken hoe doordrongen het team is van Domain-Driven Design, en hoe goed het inzicht is in het domein dat gedigitaliseerd wordt.

Wat je praktisch wil zien is een duidelijke prioriteit op het uitwerken van het core model, inclusief unit testen, maar volledig out-of-environment, zonder teveel aandacht voor frameworks, integratie-problematieken, transacties, persistentie, en alle andere aspecten die in een klassieke benadering zeer snel met de focus van het team opeisen. Een grote indicator om even halt te houden en bij te sturen is elke vorm van **sprint 0** die 'de technische opzet' moet gaan realiseren.

Een juiste aanpak resulteert dan in een zeer snelle functionele realisatie, met snelle builds, weinig overhead en laten toe om makkelijk te valideren of je op juiste spoor zit.

Later, in volgende iteraties volgt een integratie in technische opzet (db, api's, transacties, ...) die wordt uitgewerkt als een schil rond de implementatie van domein logica. Deze taken kunnen zelfs gebeuren door mensen met de nodige technische expertise die niet ingewerkt zijn in de domein logica. Dit zorgt voor een hele verademing in allocatie en planning van activiteiten en experten. Scheid gerust de activiteiten en indien mogelijk ook de rollen tussen domein programming en integratie development, hou de scheidingslijn aan en dwing ze ook af in CI pipelines én vermijd frameworks en technologie dependencies.
Technische specialisten kunnen op die manier in-en uit-faseren waar en wanneer nodig waardoor projectteams typisch kleiner en dus efficiënter kunnen zijn. Een groot verschil met de stereotiepe horror-sprintplanning- en grooming-meetings met grote groepen, met bijhorende communicatie-overhead.

Natuurlijk blijven de klassieke best practices zoals automation-of-everything etc. ook tijdens de build van een DDD project gelden.

Durf refactoreren bij nieuwe inzichten in het domein

Om op lange termijn te blijven profiteren van de voordelen die DDD biedt, mag er absoluut geen gap ontstaan tussen de code van het project en je begrip van het domein. Elke afwijking hierop moet je beschouwen als belangrijke technical debt.

Durf refactoren en hernoemen waar nodig om 100% in lijn met de modelering te blijven, geen tradeoffs hier!

Vermijd "analyse tracks"

Een makkelijk te detecteren maar soms moeilijk te vermijden anti-pattern voor een DDD project ontstaat als iemand komt aandraven met een (deel)analyse van het project. Domain-Driven Design stoelt op 'leren als groep' en zo de nodige kennis en visie verankeren in het hele team. Beter nog start je met een leeg blad en bouw je stapsgewijs met het hele team de nodige domeinkennis op dan te vervallen in de stereotiepe aanpak van de "spec over de muur". Analyse gebeurt door het team en niet door een rol binnen het team.

Hou ook goed in de gaten dat een domeinmodel niet hetzelfde is als een informatiemodel, een subtiel verschil om te bewaken, zeker omdat beide vaak gevisualiseerd worden aan de hand van UML diagrammen. Zowel de werkmethode als het resultaat van modelering zijn fundamenteel verschillend van een niet-DDD aanpak.

Werk visueel

Domain-Driven Design gaat uit van het creëren van een gedeelde "mental picture" van een domein, en dus ook de bijhorende software binnen een team. Dit lukt een stuk vlotter als mensen conversaties, discussies en deliverables visueel kunnen ondersteunen. Helemaal in lijn met alle voordelen van visual facilitation geldt ook voor Domain-Driven Design dat je best visueel te werk gaat, samen opgebouwde visuals uithangt in de projectruimte of digitaal ter beschikking stelt zodat iedereen letterlijk vanuit het laatste beeld van het systeem redeneert en werkt.

- Maak visuele modellen waarin alle concepten en relaties daartussen benoemd worden.

- Deze modellen horen aan de muur, of als dat niet kan door de projectorganisatie in een digitale documentatie tool. De modellen zijn een werkinstrument, een levend iets met een cruciale rol in de projectwerking. Verstop ze niet als naslagwerken of duffe documenten ergens in een kast.

- Modellen moeten ook live ge-update worden bij nieuwe inzichten, ze moeten de weerspiegeling zijn van het begrip dat het team heeft rond het domein. Belangrijk is ook dat effectieve aanpassingen in groep gebeuren, niet als het resultaat van een actie van één of enkele teamleden wat het gedeelde begrip van de modellen en het bijhorende domein zou ondergraven.

De wereld rond DDD

De wereld rond DDD - wat is er nog?

Eens je op verkenning gaat in de wereld rond Domain-Driven Design merk je dat een aantal andere onderwerpen op regelmatige basis tegenkomt.

Domain-Driven Design staat niet op zichzelf, het is omgeven met andere practices, methodieken, technologieën om als team samen kennis op te bouwen en een goede domein modelering uit te voeren, keuzes te maken en om de DDD concepten een concrete vertaling naar een codebase te geven.

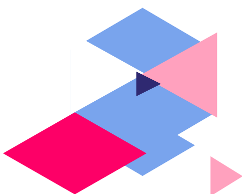
We zetten hier een aantal methodologische en technologische onderwerpen op een rijtje die heel complementair zijn met Domain-Driven Design en zeker nuttig om je verder in te verdiepen.

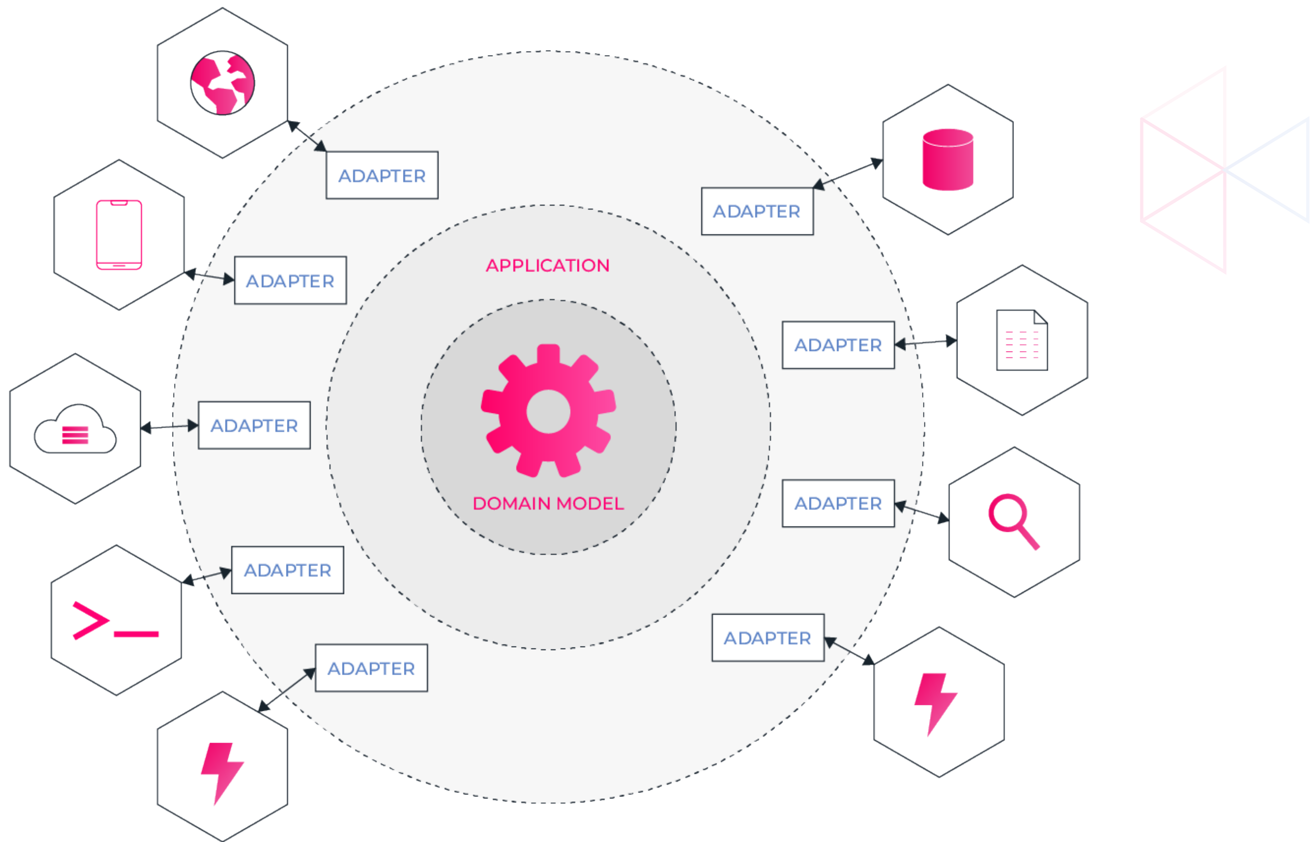
Hexagonale architectuur

Zoals al enkele keren aangehaald, is Domain-Driven Design technologie-agnostisch en bepaalt het geenszins met welke technologie, framework of zelfs applicatie-architectuur de ontwikkeling dient te gebeuren, of hoe exact elk DDD-concept in de codebase verwerkt wordt.
Een heel logische synergie is er echter te vinden met het idee van een 'Hexagonale Architectuur 'omdat dit toelaat om ook op code-niveau business eerst te plaatsen.

De meeste applicaties worden gebouwd op een technisch raamwerk: een set van technische constructies die als het ware de bedrading of het fundament van de applicatie vormen, waarin de business code dan geplaatst wordt. Dit veroorzaakt natuurlijk een design dat vaak vertrekt vanuit technische constructies "We maken een REST Controller die dan een service aanspreekt die een bericht op het queuing systeem zet." Business code geraakt echter versnipperd over de hele setup, wijzigen van technologie wordt een moeizaam proces en de logica testen lukt niet zonder veel techniciteiten mee te zeulen of artificieel uit te schakelen a.d.h.v. mocking.

Een hexagonale architectuur vertrekt technisch vanuit hetzelfde idee als DDD conceptueel: de kern van de zaak is een puur domein model dat de business reflecteert. Daarrond worden dan technische ontsluitingen voorzien (input, output, persistentie, messaging, ...) maar deze vormen geen afhankelijkheid voor het domein model en impacteren dit niet.





Dit laat toe om het concrete domein model te bouwen en testen zonder enige afhankelijkheid naar technologie, framework, of technische setup. Dit alleen garandeert dat er een ongelooflijke focus én snelheid kan gezet worden op het bouwen en valideren van domein functionaliteiten, dat builds en automatische testen zeer snel (out of environment) werken, en dat het domeinmodel gevrijwaard blijft van enige framework-afhankelijkheid, wat de conceptuele integriteit op lange termijn garandeert.

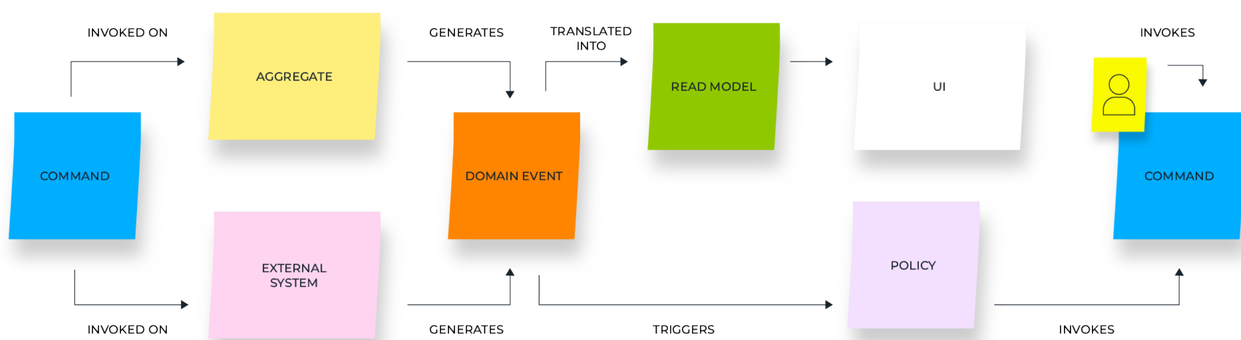
Event Storming

Domain Events komen in het originele, eerder theoretisch georiënteerde boek van Eric Evans helemaal niet als een primair DDD concept naar voren. In de loop der jaren zijn business events, onder de invloed van event-driven architecturen, event sourcing en dergelijke, een steeds prominentere plaats gaan krijgen in software design. In feite kan je stellen dat een business transactie is doorgegaan als er een Domain Event rond gegenereerd is, en dat alle daarvan afgeleide zaken, zoals het updaten van databases ondergeschikt zijn geworden. De event stream is de single-source-of-truth geworden.

In de praktische realisatie van DDD, en dus ook in het boek van Vaughn Vernon hierrond, zijn domain events (net zoals in de brede software engineering wereld trouwens) een volwaardig primair DDD concept geworden.

De grote vraag is dan: Hoe bepalen we die Domain Events, die vaak een fundamentele rol spelen in ons begrip van het domein? Hoe identificeren we die ietwat abstracte events als een concept met hulp van de mensen die de business het best kennen?

Event Storming, een techniek beschreven door Alberto Brandolini, plaatst net dat centraal: in een groepsworkshop die het geconsolideerde inzicht en kennis in een domein moet expliciteren worden Business Events als eerste en meest primaire concept centraal gesteld. Door het gebruik van post-its met specifieke kleuren per type concept ontstaat er snel een vlotte samenwerking van een potentieel groot team naar een (vormelijk) voorspelbaar resultaat.



Niet alleen worden de domain events daardoor geïdentificeerd (de beruchte oranje post-its), maar door de andere DDD concepten mee te verwerken in de workshop worden ze ineens gebruikt als basis om de Bounded Contexts te bepalen. Sleutelmomenten in een business process komen boven als events, wat een eerste indicatie geeft naar functionele opdeling én worden gelieerd aan 'Aggregates' die verantwoordelijk zijn voor het raisen van bepaalde domain events.

Belangrijk in DDD is om de specificaties business-gedreven te maken, en het hele team van dezelfde mental picture rond het domein te voorzien. Event Storming werkt zeer sterk op die twee punten. Door van het modeleren een visueel iets te maken, levert het een geconsolideerd inzicht op dat iedereen mee tot stand heeft zien komen, en dat op ieders netvlies gebrand staat. Dit vereenvoudigd verdere communicatie tussen alle betrokkenen. Daarnaast wordt door de aanpak ook vermeden dat er een upfront of eenzijdige analyse vanuit één of enkele betrokkenen ontstaat die niet de juiste basis vormt voor het DDD model.

Enkele uurtjes Event Stormen levert heel wat nieuwe inzichten en gemeenschappelijke kennis in het team op, waar je maanden nadien nog de vruchten van plukt. Wil je graag aan de slag met Event Storming? Lees dan zeker het e-book van Roberto Brandolini, te vinden op EventStorming. Voor elke situatie en doelstelling is er wel een specifieke aanpak beschreven. Zowel rond het vormelijke (benodigd materiaal, kleuren, werkwijze), met de nodige aandacht naar voorbereiding toe alsook briefing van de deelnemers en moderatie van een succesvolle Event Storming sessie.

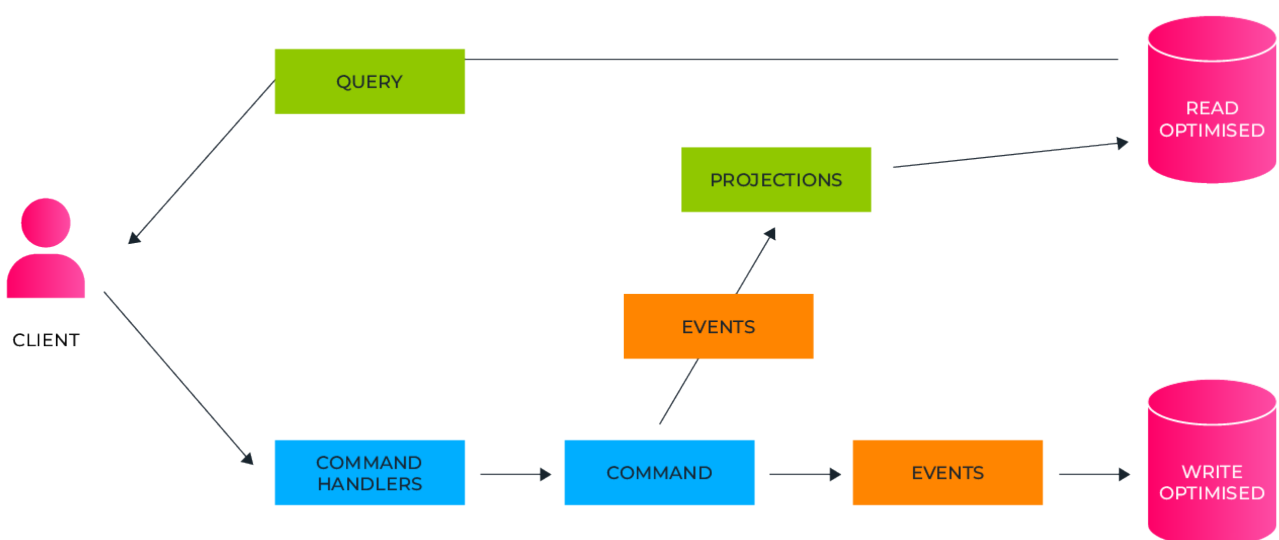
CQRS

CQRS, voluit Command/Query Responsibility Segregation, is een patroon in softwaredesign dat het scheiden van het lees- en schrijf-model in de software hanteert als alternatief voor één model dat zowel voor lezen als schrijven wordt gebruikt. Door het sturen van Commands (C) naar het model kunnen business transacties uitgevoerd worden. Het resultaat en de huidige toestand van het systeem kunnen dan via Queries (Q) bevraagd worden. Echter, de informatie-structuren en mogelijkheden van beide worden doelbewust verschillend ingericht.

Het toepassen van CQRS en Domain-Driven Design gaan vaak hand in hand, omdat in een DDD project het design vanuit het domeinbeeld gebeurt, en niet vanuit specifieke use cases van een gebruiker. Dit neemt natuurlijk niet weg dat elke usecase z'n unieke inkijk in het systeem nodig heeft, en een specifieke samengestelde informatie de nuttigste context oplevert voor de eindgebruiker.

De usecase-specifieke overwegingen worden in een klassieke aanpak in het achterhoofd meegenomen bij het opbouwen van een informatiemodel en wegen daar op het eindresultaat. Dit model wordt dan zowel voor het verwerken van business transacties (write) als het consulteren en aanbieden van die usecase-specifieke context (read) gebruikt, en moet voor beiden bruikbaar zijn. Het resultaat is dan ook dat het voor geen van beide optimaal is.

CQRS splitst deze twee aspecten, het lezen en schrijven in twee aparte modellen: het model dient om business transacties te verwerken (command). Voor het verstrekken van informatie aan gebruikers of aansluitende systemen wordt een projectie op een leesmodel voorzien (query). Dit laat niet enkel toe om het leesmodel helemaal voor te kauwen op de manier zoals het zal bevraagd en gepresenteerd worden, maar ook om dit bij wijzigende behoeften hierrond te laten evolueren zonder impact op het core model of zonder deze verantwoordelijkheden het model te laten beïnvloeden.



De projectie naar een leesmodel kan synchroon gebeuren, maar vaak wordt het ook asynchroon ingericht door de stream van domain Events geproduceerd door het model te verwerken en het leesmodel bij te werken. Asynchroon opbouwen van het leesmodel brengt natuurlijk wel wat complexiteit rond eventual consistency mee: een transactie kan verwerkt zijn, maar het leesmodel nog net niet geüpdate, wat zonder de nodige voorzieningen een bevreemdende tot onbruikbare user interface kan opleveren.

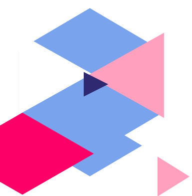
Eens CQRS toegepast zijn de voordelen legio:

- Het domein model zelf blijft gevrijwaard van elke usecase- of visualisatie-specifieke vereisten en daardoor compacter en onderhoudbaarder.

- Leesmodellen kunnen vlot mee evolueren met de behoeften en gebruikerswensen op applicatief niveau.

  Schrijf- en lees-modellen kunnen op een andere technologie gebaseerd worden, bv een klassieke relationele database aanpak voor de schrijfmodellen, met een projectie naar een ElasticSearch zoekindex om aan de leeszijde vlot en performant data ter beschikking te stellen.

- Indien er een onevenwicht is tussen schrijf- en lees-operaties (bv beperkte schrijfoperaties maar gigantisch veel consulterende gebruikers, of net een zeer hoog aantal concurrent updaters) kan de Command- of net de Query-kant van de implementatie geschaald worden zonder impact op de andere zijde.



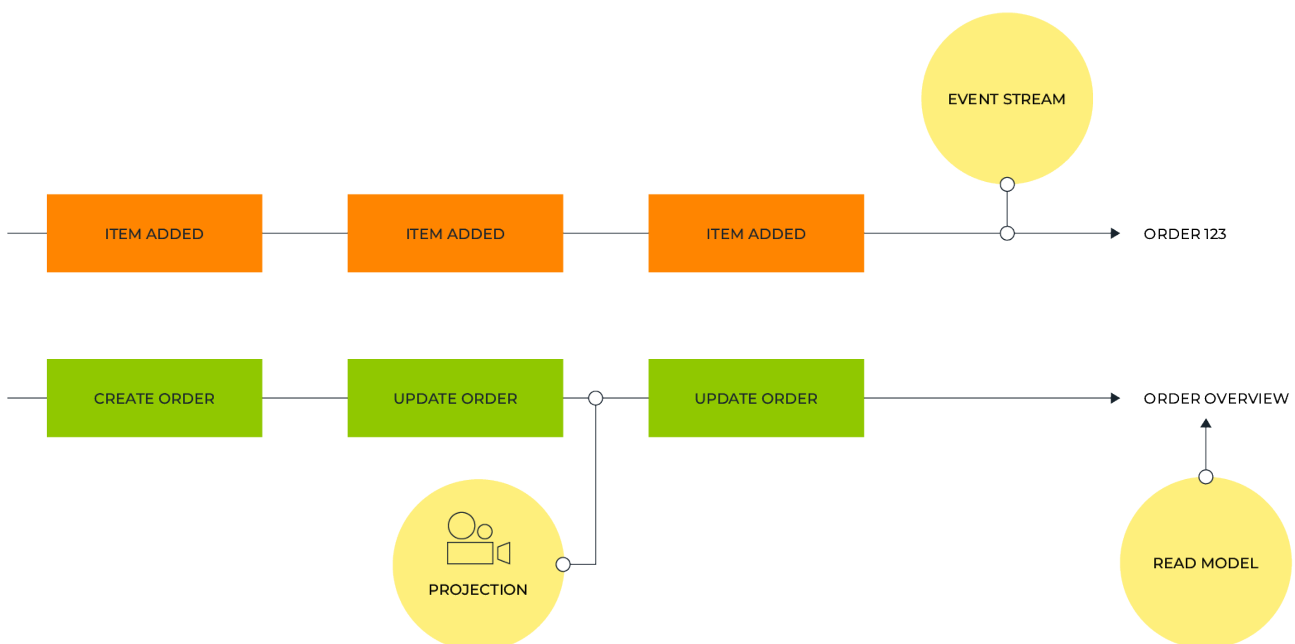
Event Sourcing

In Domain-Driven Design staat een domein model centraal, maar dit domein model is geen persistence model of het hoeft er in elk geval niet mee overeen te komen.

Wijzigingen in het domein kunnen genoteerd worden als domain events, indien dit een vertaalslag moet krijgen naar een relationeel of niet-relationeel model voor data opslag kan dat zeker, maar dat is geen conditio sine qua non.

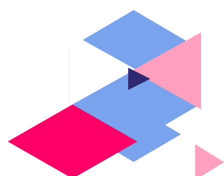
Eén van de heel interessante technische mogelijkheden die vaak aan een Domain-Driven model gekoppeld worden, is persistentie door de stream van domain events te gaan bijhouden in plaats van de huidige of laatst bekende toestand van alle data daarin.

Bij Event Sourcing, in meer detail toegelicht in onze blog post (Wat is Event Sourcing?) wordt de persistentie van het model (een DDD Aggregate om precies te zijn) voorzien door alle betrokken domain events in sequentie op te slaan. Om de laatste toestand terug te bepalen kan eenvoudigweg de volledige stroom van domain events terug afgespeeld en verwerkt worden.



Dit brengt heel wat voordelen zoals de herafspeelbaarheid om makkelijk problemen te identificeren, hoge schrijfperformantie indien dat nodig is, makkelijke inpassing in een event driven architectuur, asynchrone leesmodellen (CQRS), …

Hoe interessant Event Sourcing ook is, ervaring leert dat het ook wel wat complexiteit meebrengt, zowel in ontwikkeling maar zeker ook operationeel. Technieken als snapshots zullen ingezet moeten worden om eventuele performantie-problemen bij zeer grote event streams te ondervangen. Het feit dat elk historisch domain event - in de vorm zoals het ooit bedacht werd - tot in de eeuwigheid verwerkt zal moeten kunnen worden door elke toekomstige versie van de software zijn lange-termijn commitments die toch best degelijk overdacht worden.



Conclusie

Waarom zijn we bij XTi zo enthousiast over Domain-Driven Design? Wat mag je verwachten van een software project met Domain-Driven Design?

- DDD zorgt voor een betere teamdynamiek, met een veel hogere mate van betrokkenheid en een focus op communicatie binnen het multidisciplinaire team.

- Dit garandeert een sterke verankering van kennis en inzicht in zowel het business domein dat wordt gedigitaliseerd alsook in het resulterende systeem dat wordt opgeleverd.

- Daardoor werpt DDD niet enkel z'n vruchten af tijdens de bouwfase, waar je op een voorspelbaarder en rationeler ontwikkelproces mag rekenen, maar des te meer in een betere onderhoudbaarheid en evolueerbaarheid tijdens de 90% van de levenscyclus van de software die zich na de initiële go-live situeert.

- Daarbovenop garandeert DDD een sterk gereduceerd risico naar technologie lock-in, en duidelijke en realistische migratiepaden met telkens zeer beperkte impact en risico naar de totaaloplossing toe.

Neem contact met ons op!

Merelbeke

Guldensporenpark Gebouw i
4de verdieping
9820 Merelbeke

Sam Waegeman

Kontich

Prins Boudewijnlaan 24E
1e verdieping
2550 Kontich
+32 3 871 99 60

Dennis De Reyer

Hasselt

Kempische Steenweg 311
Corda Campus, Gebouw 1
3500 Hasselt

Bert Roex

GRAAG OVER JOUW DDD SPREKEN?

Contacteer ons