ONDERZOEKSVERSLAG
DOMAIN DRIVEN DESIGN

Onderzoek naar Domain Driven Design in de context van een CRM applicatie

Auteurs: OOSE-team Thompson, studenten HBO-ICT Opleiding HAN
Opdrachtgever: JDI

Inhoudsopgave

1. Abstract

Dit onderzoek gaat over het gebruik van Domain Driven Design (DDD) in een CRM applicatie. Deze doelstelling is opgesteld omdat JDI meer te weten wilt komen over DDD. Dit onderzoek is uitgevoerd door eerst literatuuronderzoek te doen naar DDD en dan een implementatie te maken van het onderzochte onderdeel van DDD. Uit deze implementatie is te merken dat de voordelen van DDD niet grootschalig genoeg aanwezig zijn in een kleine applicatie en de nadelen dus niet overtreffen. Dit betekend dat DDD relevant is voor grootschalige projecten en moet de opdrachtgever goed kijken of het project groot genoeg is om effectief gebruik te maken van DDD.

Dit onderzoek gaat over het gebruik van Domain Driven Design (DDD) in een CRM applicatie. Deze doelstelling is opgesteld omdat JDI meer te weten wilt komen over DDD. Hiervoor is een onderzoek uitgevoerd en een kleine demo-applicatie ontwikkeld.

Voor het onderzoek is de hoofdvraag "Wanneer is DDD een effectieve methode van ontwikkelen" opgesteld en bevat de volgende deelvragen:

1. Wat is DDD?
2. Wat zijn de voor- en nadelen van DDD?
3. Wat is een CRM?
4. Hoe ziet DDD er uit in de context van een CRM?

Tijdens dit onderzoek is er literatuur- en praktijkonderzoek gedaan. Door literatuuronderzoek te doen is er een demoapplicatie opgesteld waarop de bevindingen zijn gebaseerd.

In dit onderzoek is ondervonden dat DDD de complexiteit van de architectuur verhoogd en helpt onderdelen gescheiden te houden. De voordelen van die scheiding is niet echt gemerkt in de demoapplicatie omdat die te kleinschalig en eenvoudig was om er echt gebruik van te maken.

2. Inleiding

Dit onderzoek is opgesteld naar aanleiding van een gesprek met JDI. In dit gesprek gaf JDI aan meer te weten te willen komen over de voor en nadelen van Domain Driven Design (DDD) en hoe men deze techniek kan toepassen in hun applicaties. Hiervoor is de hoofdvraag "Wanneer is DDD een effectieve methode van ontwikkelen" opgesteld en bevat de volgende deelvragen:

1. Wat is DDD?
2. Wat zijn de voor- en nadelen van DDD?
3. Wat is een CRM?
4. Hoe ziet DDD er uit in de context van een CRM?

Na deze vragen te hebben beantwoord wordt aan de hand van door JDI uitgevoerde usecases advies gegeven wanneer een programma effectief DDD kan gebruiken.

Voor dit onderzoek is er een kleine demo applicatie gemaakt die gebruik maakt van Domain Driven Design. Deze code is te vinden op deze bitbucket repository.

3. Materialen en methoden

Dit hoofdstuk bevat de materialen en onderzoeksmethoden die gebruikt zijn in dit onderzoek.

3.1. Literatuuronderzoek

Voor dit onderzoek wordt er gebruik gemaakt van literatuuronderzoek. De literatuur bestaat veelal uit online bronnen. Alle bronnen worden streng gecontroleerd op de volgende punten:

- Actualiteit
- Auteur
- Meningen
- Onpartijdigheid van de auteur

Elke gebruikte bron is terug te vinden in de Literatuurlijst aan het einde van dit onderzoeksrapport. Alle bronnen worden gecontroleerd op bovenstaande punten en deze resultaten komen in het hoofdstuk Discussie aan bod. De genoemde punten worden gemeten op de volgende voorwaarde(n):

Actualiteit

Een bron mag niet ouder zijn dan 2 jaar op het moment van raadplegen *of* de informatie in de bron moet nog steeds relevant zijn, d.w.z. dat hetgeen wat in de bron staat in de loop van tijd

niet veranderd kan zijn. Dit laatste moet verantwoord (bewezen) worden in het hoofdstuk Discussie.

Auteur

De auteur moet voldoende kennis hebben over hetgeen waar hij/zij over schrijft. Dat wil zeggen dat de auteur gediplomeerd is in het vakgebied waar de bron over gaat *of* de auteur is minimaal 2 jaar werkzaam in het vakgebied waar de bron over gaat. De kennis van de auteur moet worden verantwoord in het hoofdstuk Discussie.

Meningen

Alle bronnen moeten gecontroleerd worden op meningen. Bronnen moeten objectief zijn. Bronnen mogen alleen een vorm van mening bevatten wanneer er wordt gesproken over voor- en nadelen, omdat dit altijd een persoonlijke mening is. Echter moeten er bij dit soort bronnen minimaal 2 andere bronnen hebben, die ook voldoen aan alle andere beoordelingscriteria, die dezelfde mening ondersteunen. Gebruik van meningen moet duidelijk (d.w.z. concreet in woorden vermeld aan het begin van een alinea) worden vermeld.

Onpartijdigheid van de auteur

De auteur mag geen direct relatie hebben met een partij die belang heeft bij een bron. Een voorbeeld: een bron over de manier van werken bij Tesla mag niet geschreven zijn door de teamleider van Tesla, omdat hij/zij een directe relatie heeft met de inhoud van de bron, waardoor al snel een subjectieve en/of verheerlijkende bron ontstaat. De relatie van de auteur (of de organisatie waar de auteur voor werkt) met de inhoud van de bron moet worden onderzocht en verantwoord worden in het hoofdstuk Discussie.

3.2. Praktijkonderzoek

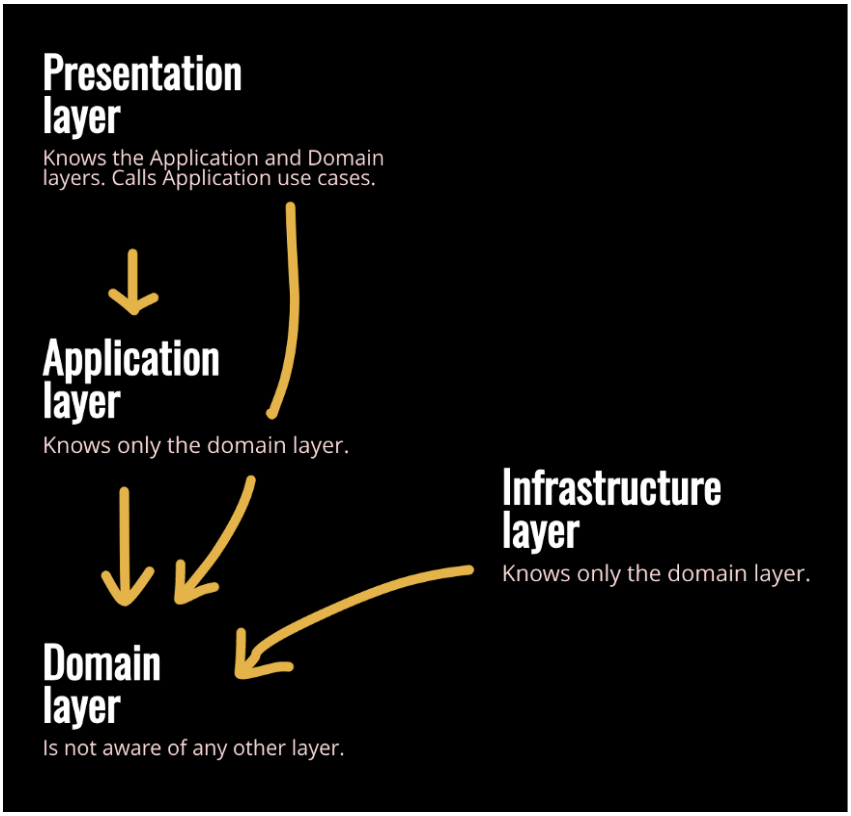
Voor het onderzoek wordt ook gebruik gemaakt van praktijkonderzoek. Dit gebeurd door middel van het toepassen van de techniek in een voorbeeld CRM-applicatie. Deze onderzoekstechniek wordt toegepast om te kijken of Domain Driven Design daadwerkelijk een goede toevoeging is in de praktijk, om zo bij te dragen aan het doel van het overkoepelende onderzoek CRM-JDI (Plan van Aanpak - Hoofdstuk 3).

4. Wat is Domain Driven Design?

Domain Driven Design, benaamd in het gelijknamige boek van Eric Evans uit 2003, is een manier om software te ontwerpen waarin domeinen gescheiden worden van technische details.

4.1. Het Principe

Domain Driven Design scheidt de applicatie in vier verschillende layers, de Presentation layer, de Application layer, de Domain layer en de Infrastructure Layer. Wat Domain Driven Design anders maakt van Layered Architecture is dat Domain Driven Design naast de "horizontale" scheiding van de layers een "verticale" scheiding heeft tussen de domeinen. Deze domeinen weten van elkaar in de Domain layer. Hierdoor zit bijna alle logica in de Domain layer en sturen de andere layers alleen aan wat in de Domain layer gebeurt. Het overzicht tussen de layers is te zien in de volgende afbeelding.



Figuur 1: Overzicht van de lagen in DDD.

4.2. De layers

Zoals te zien in *figuur 1* kent de Presentation layer, ook wel bekend als de user interface layer, de Application layer en de Domain layer. Deze laag is, zoals de naam verklapt, de user interface en is verantwoordelijk om de informatie te laten zien en de acties van de gebruiker te interpreteren. De gebruiker kan ook een ander computersysteem zijn.
De Application layer kent de Domain Layer. De application layer is er verantwoordelijk voor om de juiste functies in de Domain layer aan te roepen, verder bevat het geen logica.
De Infrastructure layer kent de Domain layer. De Infrastructure layer omvangt technische capaciteiten, zoals een database connectie.
De Domain layer kent geen andere layer. Hierin bevindt zich de belangrijkste logica en is de kern van de applicatie.

Behalve de Presentation layer kent iedere layer services. Hoewel ze allemaal services heten, is het doel per layer anders.
De services in de domain layer omvatten functies die niet bij een enkele entity passen en functies die de zogeheten aggregates als argumenten nemen. Deze staan altijd buiten aggregates. Aggregates en entities worden in het volgende hoofdstuk besproken.
De services in de app layer krijgen data van buiten het systeem en zetten het om naar data waar het systeem gebruik van kan maken en roepen de benodigde functies uit de domain layer aan.
De services in de infrastructure layer dienen om de afhankelijkheid van een extern systeem los te koppelen, zodat men naar een ander systeem kan wisselen zonder de bestaande code te hoeven aanpassen. Deze services implementeren een interface uit de domain layer. Als een service over verbinding met een database gaat, heet dit een repository.

4.3. Aggregates.

Aggregates omvatten twee verschillende elementen, ook wel modules genoemd:

1. Value objects.
2. Entities.

Value objects zijn meestal simpele objecten die dingen omschrijven zonder een specifieke identiteit. Dit is bijvoorbeeld geld, waar in het systeem de hoeveelheid van belang is en niet het specifieke briefje of muntje. Deze objecten worden meestal meegegeven als argument voor een functie.

Entities zijn objecten die een specifieke identiteit hebben, wat bepaalde attributen bepaald. Dit kan bijvoorbeeld een portemonnee zijn, wiens identiteit bepaald hoeveel geld er in zit. Deze

objecten bevatten regelmatig functionaliteit specifiek voor het object, zoals geld uit de portemonnee halen.

Groeperingen van Value Objects en Entities worden aggregates genoemd. Deze aggregates zorgen voor een overzicht en grenzen tussen en in Domains. Aggregates worden via één Entity aangeroepen, de zogeheten root. Deze root bepaald de waarden en de identiteiten van de Value Objects en de Entities in de aggregate en laat alle logica in de aggregate uitvoeren. Verschillende aggregates kunnen kennis hebben over dezelfde Value Objects. Omdat value objects geen specifieke identiteit hebben, kan aggregate A niet bij de implementatie van de value object van aggregate B. Omdat entities wel een specifieke identiteit hebben, is dat wel mogelijk en worden dus niet gedeeld tussen aggregates. Hierdoor kunnen de aggregates zogeheten invariant business rules afdwingen. De invariants zijn business rules die niet overschreden mogen worden en omdat de aggregates altijd door de root worden aangeroepen kan men de assumptie nemen dat de aggregate zich nooit in een incorrecte staat bevind.

Als een aggregate van een andere aggregate af weet of een functie twee aggregates beïnvloed, wordt er gebruik gemaakt van een event handler. Deze event handler wordt vanuit een aggregate root aangeroepen en kan meerdere aggregates aanroepen om enige veranderingen door te voeren.

5. Voor en nadelen DDD

Hierin worden de bevonden voor en nadelen van Domain Driven Design besproken.

5.1. Goed voor complexe domeinen en grote systemen.

Doordat DDD de focus legt op domeinen, is er veel ruimte om deze uit te werken. Omdat de programmeurs in domeinen werken, kunnen ze makkelijker werken en overleggen met de domeinexperts. Hierdoor kunnen complexere domeinen, of domeinen die met andere domeinen te maken hebben, dieper worden uitgewerkt.
Ook zorgt de loskoppeling van de technische details en de domeinen er voor dat er makkelijker onderhoud kan worden gepleegd. Als een extern systeem moet worden vervangen, omdat het systeem verouderd is of men een beter systeem heeft gevonden, hoeft men alleen de service implementatie voor het nieuwe systeem aan te maken zonder in de domeinen code te veranderen. Hierdoor is het ook makkelijk te testen. Doordat een domein niet een directe afhankelijkheid heeft met een ander domein of een extern systeem, is deze makkelijk te testen zonder dat iets buiten de scope van de test invloed heeft op het resultaat.

5.2. DDD is complex.

DDD voegt een extra laag complexiteit toe aan het layer pattern, wat voor teams die niet bekend zijn met DDD voor een hoop leerwerk zorgt. Door de extra complexiteit wordt het ontwerpen en managen van de architectuur lastiger en langer, waardoor het langer duurt het product te bouwen. Omdat de voordelen alleen te merken zijn bij grotere en/of complexe systemen, is het niet aan te raden voor kleine projecten om DDD te gebruiken.

6. Demo-applicatie

Om een goed beeld te krijgen hoe een Domain Driven Design programma werkt, is er besloten om een demoprogramma te maken voor dit onderzoek. De basisfunctionaliteiten voor een CRM applicatie zijn geïmplementeerd om te kijken hoe Domain Driven Design werkt voor de soort applicaties die JDI ontwikkeld. Om de applicatie simpel te houden en de focus te behouden op het onderzoek van DDD is er besloten om alleen de API endpoints te maken om met swagger aan te roepen.

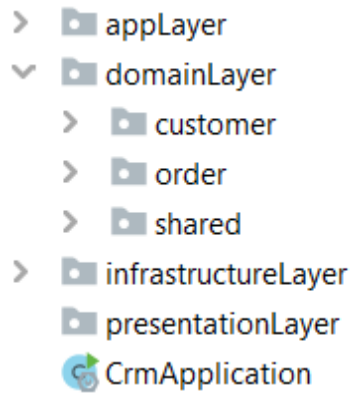
6.1. Wat is een CRM?

Customer relationship management *oftewel* ***CRM*** *is een Engelstalige benaming voor klantrelatiebeheer, soms ook relatiemarketing of verkoopbeheersysteem genoemd. Het is een werkwijze alsmede een technologie waarbij klantgegevens worden geanalyseerd om de zakelijke relatie met klanten te verbeteren, met als doel hen aan het bedrijf of de organisatie te binden en zo uiteindelijk de inkomsten te verhogen.* [2]

Omdat dit project de focus legt op Domain Driven Design en niet de CRM zelf hebben we in overleg met JDI gekozen om de implementatie zo simpel mogelijk te houden. De applicatie bevat een CRUD **(create, read, update, delete)** endpoint voor een klant **(customer).**

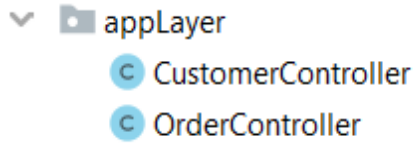
6.2. Applicatiestructuur

De structuur van de applicatie is te zien in figuur 2. Zo als te zien is, is de applicatie opgedeeld in de verschillende lagen die genoemd zijn in hoofdstuk drie. Deze groepering is gedaan met folders, waardoor het makkelijk te zien is waar welk gedeelte van de code te vinden is. Omdat besloten is niet een UI te maken voor dit demoproject is de presentation layer leeg. Ook is te zien dat de domainLayer verder is opgedeeld in de verschillende domeinen en een aparte klasse genaamd CrmApplication. Dit is de klasse die het programma draait door gebruik te maken van het SPRING framework. Omdat je mogelijk van framework wilt wisselen, zou deze klasse in de infrastructure layer horen, maar omdat de klassen in de application layer ook gebruik maken van SPRING zou het misschien beter daar bij passen. Hierdoor weten we nog niet zeker waar het moet staan en hebben we het nog even buiten de groepering gehouden.



Figuur 2: De structuur van het demoprogramma.

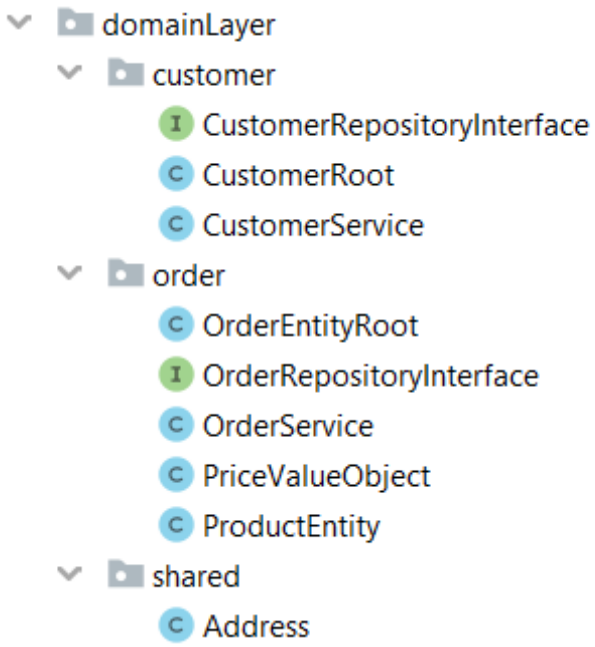
6.2.1. De application layer



Figuur 3: De structuur van de applicationLayer in het demoprogramma.

Dit is de appLayer van het demoprogramma. Het bevat de REST-endpoints, geïmplementeerd met het SPRING framework, waarmee de applicatie wordt aangeroepen. Ieder domein heeft nu een eigen controller, maar wanneer het aantal endpoints groeit zullen de controllers worden opgesplitst en gegroepeerd worden in een folder, met de naam van het domein.

6.2.2. De domain layer



Figuur 4: De structuur van de domainLayer in het demoprogramma.

Hier zijn de twee verschillende domeinen te zien, met een gedeelde value object die apart staat. Ieder domein bevat een aggregate, een service en een repository interface. Via de service roept de application layer het domein aan en de service kan via de repository de aggregates ophalen. De aggregates worden via de root aangeroepen. Hierdoor moet bijvoorbeeld de orderservice eerst via de repository alle order aggregates ophalen om dan via iedere OrderEntityRoot de productEntites opvragen om te kijken of er een bepaald product bestaat.

6.2.2.1. Het customer domein

De aggregate omvat de CustomerRoot en de Address.

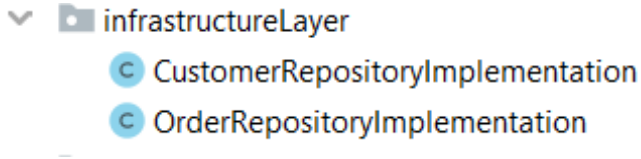
6.2.2.2. Het order domein

De aggregate omvat de OrderEntityRoot, de ProductEntity, de PriceValueObject en Address. De OrderEntityRoot en de ProductEntity maken beide gebruik van de PriceValueObject. Met event sourcing zal OrderEntityRoot bij kunnen houden hoe de prijs van de order is veranderd en hoeft dan mogelijk niet meer gebruik te maken van PriceValueObject.

6.2.2.3. Het gedeelde domein

De shared folder bevat de value object Address. Dit value object wordt door de customer en de order domeinen gebruikt, waardoor het niet bij een van de twee past.

6.2.3. De infrastructure layer



Figuur 5: De structuur van de infrasctructureLayer in het demoprogramma.

In de infrastructure layer zijn de repository interfaces van de domain layer geïmplementeerd. Deze slaan gegevens nu in memory op, zodat men niet een aparte kleine database moet aanmaken om de applicatie uit te kunnen voeren. In deze laag zal ook de connectie met de mogelijke database komen.

7. Conclusie

Uit het onderzoek is gebleken dat Domain Driven Development niet bijzonder veel voordelen heeft op kleinschalige en eenvoudige projecten zoals het demoproject. De extra complexiteit zorgt er voor dat men tijd moet steken in het leren van DDD, en om de architectuur te behouden kost tijd. Maar doordat het gescheiden is in domeinen, kan een verandering van de code in één domein niet zomaar fouten opleveren in een ander domein. Verder maar het veel gebruik van dependency injection, waardoor je makkelijk de specifieke implementatie kan vervangen zonder code aan te hoeven passen. Hierdoor is het goed geschikt voor grotere projecten, waar men zo min mogelijk wilt aanpassen aan de al bestaande code.

Er is niet een concreet punt waarop je kan zeggen wanneer DDD geschikt is voor een project. Een project kan grootschalig zijn maar als het alleen bestaand uit eenvoudige CRUD functionaliteiten is DDD onnodig, en een project kan kleinschalig zijn maar in DDD zes complexe aggregrates krijgen en er veel nut van verkrijgen. Om de hoofdvraag, "Wanneer is DDD een effectieve methode van ontwikkelen", te beantwoorden: Er is geen definitief punt, maar het lijkt ons dat wanneer er vier aggregates zijn die ieder vier entities en/of value objects bevatten en minstens één functionaliteit bevatten, de voordelen van DDD zowiezo meer naar voren gaan komen dan de nadelen.

8. Advies

In dit hoofdstuk willen we als team Thompson concreet advies geven binnen welke projecten Domain Driven design toepasbaar zou kunnen zijn binnen JDI. Er is gekeken naar verschillende cases die JDI in het verleden heeft opgepakt en voor deze cases zijn de hypothetische voor en nadelen opgesteld.

8.1. Usecase 1: DOS-applicatie slim nagebouwd

https://www.jdi.nl/cases/dos-applicatie-nagebouwd-tot-webapplicatie

Beschrijving van de case op de website van JDI: *Een bedrijfskritische DOS-applicatie waarmee Circulus-Berkel registreert hoeveel klein chemisch afval (KCA) er wordt ingeleverd, wordt niet meer ondersteind door haar leverancier. Om de organisatie zo min mogelijk te belasten en processen minimaal te laten wijzigen, is besloten om de applicatie na te bouwen. Door de focus te leggen op de gebruikers mag het eindresultaat er zijn: een gebruiksvriendelijke KCA webapplicatie met alleen functionaliteiten die er echt toe doen.*

Dit project lijkt ons net te kleinschalig om echt effectief gebruik te kunnen maken van Domain Driven Development. De domeinen die we uit de omschrijving kunnen halen zijn de registraties, de werknemers/gebruikers en de brandweer. Ook kunnen we de mogelijke domeinen van de debiteuren en de chemische stoffen vinden, hoewel we niet zeker weten in hoeverre deze inbegrepen zijn in de registraties maar voor dit voorbeeld zien we ze als aparte domeinen.

De registraties maken waarschijnlijk goed gebruik van de voordelen van DDD. Een registratie weet alleen de IDs van de debiteurs en chemische stoffen en zal er dan niet direct bij kunnen, waardoor er met domain events moet worden gewerkt en ongewenste veranderingen zo kunnen worden voorkomen. Denk bijvoorbeeld aan dat de kwantiteit van een chemische stof wordt veranderd, dit is waarschijnlijk opgeslagen in het domein van de registratie en kan de naam van de stof niet per ongeluk aanpassen voor iedere registratie met die stof. Of als een spelfout in de naam van een contactpersoon van een debiteur moet worden veranderd, moet met een domain event eerst specifiek van deze registratie naar de enigste bekende debiteur, dus via de root entity, voordat de contactpersoon kan worden veranderd. Hierdoor haal je de mogelijkheid weg dat je de verkeerde debiteur hebt.

De gebruiker is het andere domein dat logica bevat, met het inloggen en mogelijk aanmaken en aanpassen van registraties. Hierdoor hebben alleen de registraties en mogelijk de gebruiker het voordeel dat ze alleen de specifieke implementaties van andere domeinen waarvan ze afweten via domain event kunnen aanroepen, waardoor een belangrijk voordeel van DDD mogelijk niet veel wordt benut. Hierdoor denken wij ook dat het grote nadeel van DDD, de langere ontwikkelingstijd door de extra complexiteit in de architectuur, toch nog zwaarder weegt dan het voordeel en raden het bij deze usecase net niet aan.

Team Thompson adviseert bij usecases met vergelijkbare scope kritisch te kijken naar hoeveel domeinen kunnen worden beïnvloed en op hoeveel manieren. Als vier domeinen kunnen worden beïnvloed of één domein op iets van zes verschillende manieren kan worden beïnvloed, moet men beginnen te overwegen DDD te gebruiken.

8.2. Usecase 2: Software voor volledig geautomatiseerde project calculaties

https://www.jdi.nl/cases/software-voor-volledig-geautomatiseerde-project-calculaties

Beschrijving van deze usecase: *Deze case gaat over een producent, een wereldmarktleider, die sterk is in het ontwikkelen van nieuwe producten en systemen voor de Agri Sector. Deze producent had de wens om het verkoopproces voor haar klanten te vereenvoudigen en de drempels te verlagen, dit door het ontwikkelen van een product configurator.*

Dit project lijkt ons wel geschikt voor Domain Driven Design. Het Minimum Viable Product is wel een klein puntje als men nog niet bekend is met DDD. Het MVP moet zo snel mogelijk worden uitgerold en als men nog bekend moet raken met DDD, levert dit vertragingen op.

De producent heeft de configurator die zij voor ogen hadden uitgebreid uitgewerkt. Aan ons was het de taak om dit te vertalen naar een geautomatiseerd online process, gevangen in een webapplicatie.
In een nauwe samenwerking is de online product configurator ontwikkeld. Een project wat vooral R&D (Research and Development) gedreven was. De grootste uitdaging zat in de 2D tekentool, welke samen moest werken met de hoeveelheid formules en de business rules in de formules.

Zoals te lezen heeft de klant het project al uitgebreid uitgewerkt en had nauwe communicatie. DDD kan dit bevorderen doordat je in domeinen werkt en dus niet de domeinen die de klant heeft opgesteld hoeft te vertalen. Ook zijn de business rules voor de formules af te dwingen in de domeinen met het gebruik van invarianten, of een corrective policy als de business rules overschreden mogen worden.
Ook lijkt het ons dat dit programma veel domeinen zou bevatten, met alle verschillende producten, de formules en mogelijk een zeer complex domein voor de tool zelf. Hierdoor denken we dat het programma wel goed gebruik kan maken van de voordelen van DDD, zonder dat de ontwikkeling te lang wordt uitgestrekt door de extra complexiteit.

Team Thompson adviseert bij usecases met vergelijkbare scope Domain Driven Design te gebruiken.

9. Discussie

In dit hoofdstuk wordt besproken wat beter kon in het onderzoek en worden de bronnen beoordeeld.

9.1. Verbeterpunten onderzoek

9.1.1. Beter focus leggen op coderen

Tijdens het onderzoek is er een voorbeeldprogramma ontwikkeld. Er is geen grote focus op het programmeren gelegd, waardoor sommige functionaliteiten van DDD over het hoofd zijn gezien. Denk aan de functies in de entities, de domeinevents en de invarianten. Deze zijn tijdens het schrijven van het verslag gevonden en de functionaliteiten zijn uiteindelijk wel geïmplementeerd, maar de rest zijn dan achterwegen gelaten omdat er gewerkt moest worden aan het hoofdprogramma.

Verder is dit programma niet complex, wat niet ten goede gaat met DDD. Hierdoor kunnen we niet goed oordelen hoe DDD werkt bij de soort programma's waarvoor het is bedoeld.

9.2. Beoordeling bronnen

Wikipedia schept een goed beginnend overzicht van wat DDD is, maar gaat er niet diep genoeg op in om achter alles te komen. Hier mist bijvoorbeeld de verschillende lagen van het systeem en moet je hierover lezen in een verwezen pagina. Ditzelfde geld ook voor de wikipedia pagina over het CRM. Het schept een goed beginnend overzicht, maar het mist diepgang om het echt goed te begrijpen.

Medium is een zeer goede bron. Hoewel het meer dan twee jaar is geschreven, is DDD 20 jaar oud en staat vast in de definitie. Het is geschreven door Laurent Grima, een product ontwerper die met DDD heeft gewerkt. Er wordt een duidelijk overzicht geschept over waarvoor DDD is bedoeld, hoe het is opgebouwd en hoe ieder onderdeel werkt. In deze bron worden geen meningen verwerkt en de auteur is onpartijdig.

Mirkosertic is een goede bron. Hoewel het is geschreven in 2013, is de informatie nog steeds relevant. De auteur is Mirko Sertic, een java programmeur en systeemarchitect die ervaring heeft met DDD. Het volgt een voorbeeld project die wordt omgezet naar DDD, wat een overzicht geeft waarom je DDD zou gebruiken en hoe het in elkaar zit. Er zit een hoop informatie in, wat soms te veel is en je het opnieuw moet lezen om te begrijpen. Hoewel er meningen worden gegeven, zijn deze over hoe interessant sommige onderdelen zijn en niet op het vlak of iets beter is. De auteur is onpartijdig.

De Microsoft bron is een goede bron. Het gaat over een techniek die niet specifiek voor DDD is maar er wel wordt gebruikt en gaat zeer diep in hoe dit werkt en wordt geïmplementeerd. De 13 auteurs hebben geen ruimte gelaten om meningen toe te voegen en omdat het een techniek is, heeft niemand er baat bij om dit te promoten en is het artikel onpartijdig.

Developer20 is een matige bron. Het verdiept zich in de services en de informatie komt overeen met de rest, maar de auteur heeft niks waaruit blijkt hoeveel ervaring hij heeft met DDD of andere programmeertalen. Verder is de website waarop het is gehost ook niks noemenswaardig, waardoor je daarop ook niet kan beoordelen hoe goed de informatie is. De auteur is wel onpartijdig en heeft geen meningen is het stuk verwerkt.

Domaincentric is wel een goede bron. De website is helemaal gericht op Domain Driven Design en de auteur zou 10+ jaar ervaring hebben in de IT, waarin hij veel heeft gewerkt met DDD. Het stuk is recent en volledig onpartijdig geschreven.

10. Bronnen

[1] Wikipedia contributors. (2021, 22 november). *Domain-driven design*. Wikipedia. Geraadpleegd op 3 december 2021, van https://en.wikipedia.org/wiki/Domain-driven_design

[2] Wikipedia-bijdragers. (2021, 3 mei). *Customer relationship management*. Wikipedia. Geraadpleegd op 10 december 2021, van https://nl.wikipedia.org/wiki/Customer_relationship_management

[3] Grima, L. (2020, 1 juni). *An introduction to Domain-Driven Design - inato*. Medium. Geraadpleegd op 3 december 2021, van https://medium.com/inato/an-introduction-to-domain-driven-design-386754392465

[4] Sertic, M. (2013, 22 april). *Mirko Sertic*. Mirkosertic. Geraadpleegd op 3 december 2021, van https://www.mirkosertic.de/blog/2013/04/domain-driven-design-example/

[5] N. (2021, 15 september). *Domain events. design and implementation*. Microsoft Docs. Geraadpleegd op 3 december 2021, van https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation

[6] *Services in DDD finally explained*. (2018, 15 juli). Developer20. Geraadpleegd op 3 december 2021, van https://developer20.com/services-in-ddd-finally-explained/

[7] Gunia, K. (2020, 28 februari). *Modelling Aggregates: Invariants vs Corrective Policies*. Domain Centric. Geraadpleegd op 8 december 2021, van

https://domaincentric.net/blog/modelling-business-rules-invariants-vs-corrective-policies