Methodology

Domain Driven Design (DDD) is gebaseerd op ideeën die in de objectgeoriënteerde wereld al langer leven. Waar gaat DDD eigenlijk over? Hoe kan het in de praktijk toegepast worden? Wat proberen we ermee te bereiken en wat zijn de voor- en nadelen van deze benadering in de praktijk? In dit artikel geven Ralf Wolter, Thomas Zeeman en Edwin van Dillen vanuit hun ervaring met Domain Driven Design antwoord op bovenstaande vragen.

Domain Driven Design

Achtergronden en ervaringen uit de praktijk

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

Ralf Wolter en **Thomas Zeeman**
zijn senior consultant bij Sogyo en bereikbaar via rwolter@sogyo.nl en tzeeman@sogyo.nl

DDD is sinds het werk van Eric Evans (zie [3] bij kader 'Literatuur' op pagina 11) bekend geworden bij een breed publiek. Ook hebben onder andere Martin Fowler [1] en Alistair Cockburn [9] al verschillende malen hun licht over dit onderwerp laten schijnen. Ook op dit moment geniet DDD veel aandacht binnen de wereld van de software-ontwikkeling. Met deze ontwerp- en implementatiestijl wordt getracht de onderhoudbaarheid van software te verhogen, waardoor deze makkelijker aan te passen zou zijn. Is dit ook daadwerkelijk het geval?

Goede aanpasbaarheid

Veel applicaties worden gedurende hun levensduur veelvuldig aangepast, vaak door veranderende requirements [7]. Al tijdens de projectfase van een applicatie kunnen wijzigingen een belangrijke rol spelen: volgens in Larman [8] aangehaalde onderzoeken blijkt dat rond 65% van de requirements die aan het begin van een project benoemd worden, aan het eind al niet meer relevant zijn. Software is dus continu aan verandering onderhevig. De kosten van deze wijzigingen lopen volgens sommige onderzoeken [6] uiteen van 50% tot meer dan 90% van de totale kosten van een applicatie gedurende zijn levensduur. Kortom: met een vergroting van de aanpasbaarheid van een applicatie is veel voordeel te behalen.

Eén van de factoren die de aanpasbaarheid van software bepaalt, is de mate van complexiteit: de complexiteit in de software zelf en de complexiteit in de businessprocessen die de software ondersteunen. Deze twee vormen van complexiteit worden door Brooks [2] aangeduid met achtereenvolgens 'accidental complexity' en 'essential complexity' (zie kader).

Essential complexity - de complexiteit die in de businessprocessen aanwezig is - kan over het algemeen alleen verlaagd worden door het aanpassen van deze processen zelf. Met softwareontwikkeling kan hier immers weinig aan veranderd worden. Accidental complexity –bepaald door de manier waarop de software ontworpen en gecodeerd wordt- probeert men in de praktijk te reduceren door lagen in de software aan te brengen. Op zichzelf is dit een goed uitgangspunt: door het definiëren van lagen wordt aangegeven welke verantwoordelijkheid waar komt te liggen. De nadruk ligt hierbij echter op de opdeling van de lagen zelf, terwijl niet alleen deze opdeling centraal moet staan, maar daarnaast ook de manier waarop de lagen met elkaar zullen samenwerken en daarmee de afhankelijkheden die ontstaan. Zodoende wordt de focus inmiddels steeds meer gelegd op het opsplitsten van verantwoordelijkheden (of, naar Dijkstra[1]: *Separation of Concerns* - zie kader), in plaats van alleen op het lagenmodel. Hoe geeft DDD invulling aan deze separation of concerns? Om deze vraag te beantwoorden, volgt nu een uitleg over DDD. Vervolgens kijken we hoe het werkt in de praktijk.

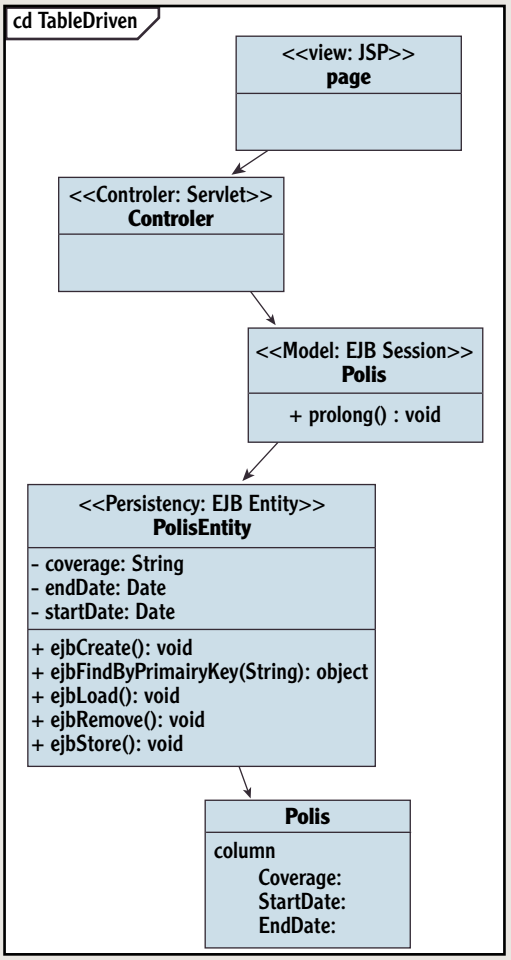
DDD en *Separation of Concerns*

Bij DDD is het uitgangspunt het feit dat het businessdomein ondersteund moet worden. Dit businessdomein wordt gemodelleerd, het model dat overblijft representeert het gedrag dat het systeem moet vertonen. In het businessdomein wordt dus de kern van het systeem uitgedrukt. Vanuit een technisch oogpunt bezien, bestaat het systeem dan slechts uit een paar classes (zie kader 'Table driven versus domain driven').

Er is nog geen GUI of database aan te pas gekomen. Natuurlijk kan een systeem doorgaans niet

Table-driven versus domain-driven

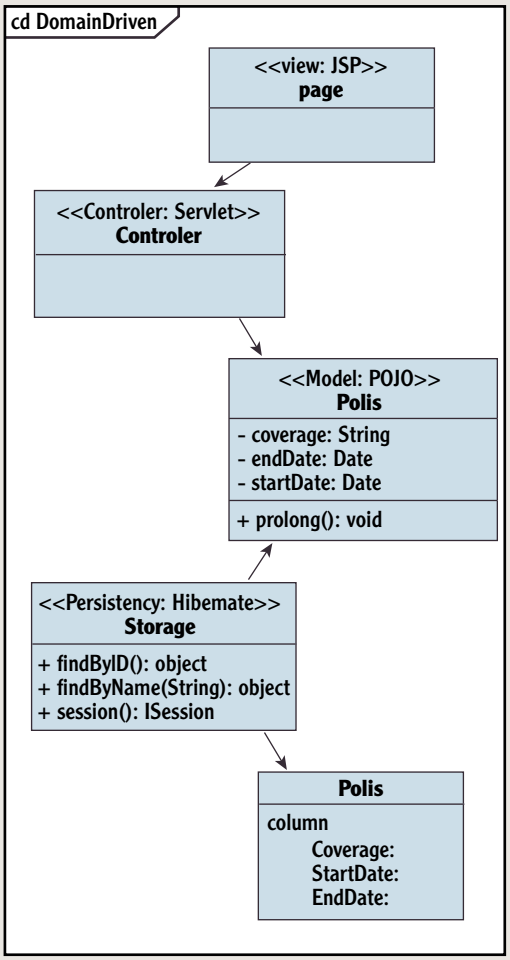
Martin Fowler[1] onderkent in zijn werk drie verschillende applicatie-ontwikkelstijlen: transactiescript, table driven en domain driven. Transactiescript is een stijl die zeer procedureel georiënteerd is en zal hier verder niet besproken worden. Het is interessant om het verschil tussen de table driven en domain driven benadering te bekijken aan de hand van een implementatie.



Afbeelding 1. Table driven.

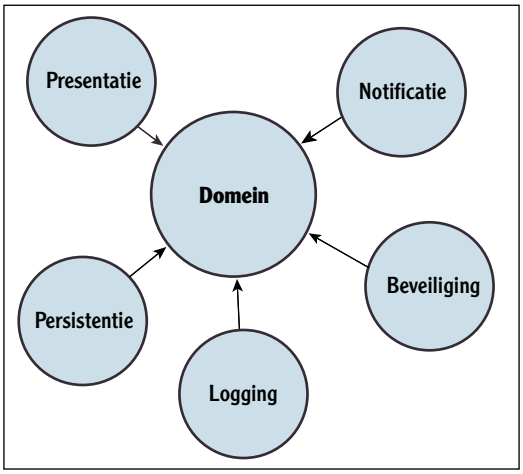
In de table driven-ontwikkelstijl staan tabellen (entiteiten) centraal. Rondom de entiteit wordt het systeem opgebouwd, een werkwijze die ook terugkomt in de Sun-blueprint voor EJB-implementaties. Stel dat een webapplicatie geïmplementeerd wordt. Dan kan van JSP's gebruikgemaakt worden om een view (denk aan MVC) te implementeren. De controler wordt vervolgens in een Servlet geïmplementeerd. Hier kunnen natuurlijk ook uitgebreide frameworks als Struts of JSF voor gebruikt worden.

In de volgende stap wordt vaak een businesscomponent geïmplementeerd door een Session Bean. Het betreft een Polis (zie afbeelding 1). In de Session Bean wordt vervolgens de businesslogica geïmplementeerd die betrekking heeft op een polis (in dit eenvoudige voorbeeld de functionaliteit die de polis kan verlengen). Het feit dat de polisgegevens in een database worden opgeslagen, betekent hier dat een Entity Bean gebruikt wordt om deze data in een objectvorm te representeren. De Polis kan nu via de Entity Bean de polis data benaderen. Wat opvalt in deze ontwikkelstijl, is dat de actie iedere keer een laag naar beneden wordt doorgegeven tot deze in de database belandt. Strikt gezien is de database dus de enige onafhankelijke laag. De overige lagen zijn allemaal afhankelijk van de onderliggende laag.



Afbeelding 2. Domain driven.

In de domeingedreven ontwikkelstijl wordt ook gebruik gemaakt van een goede MVC-implementatie, dus de JSP voor de view en Servlet voor de controler, of ook hier een uitgebreid framework als Struts of JSF. Het grote verschil met de tabelgedreven benadering zit in het feit dat de businesslogica geïmplementeerd wordt in een POJO: een simpele Java-class die zowel de businesslogica als de data bevat. Deze zijn dus altijd verenigd in een ding (domeinobject). Dit object kan relaties hebben met andere domeinobjecten. In het voorbeeld: een relatie of een verzekerd object. Dit laatste is erg interessant. Zoals je ziet zijn er alleen afhankelijkheden vanuit de omringende 'lagen' naar het domeinmodel en niet vanuit het domeinmodel naar buiten.



Afbeelding 3. De GUI en database worden dus buiten het domein geplaatst en faciliteren slechts.

zonder een GUI of database. Deze onderdelen worden in losse services geplaatst. De GUI en database worden dus buiten het domein geplaatst en faciliteren slechts, zie het model in afbeelding 3.
Uit dit model blijkt dat het domein geheel op zichzelf staat. De services daarentegen mogen het domein wel kennen, waardoor er dus een beperkte afhankelijkheid bestaat. Het grote voordeel hiervan is dat het domein geheel onveranderd blijft als er een service verandert of als er een nieuwe service wordt toegevoegd. Wat dit verder betekent, kan het makkelijkst worden verduidelijkt met een voorbeeld:

Bouwmarkt

Neem een systeem waarin klantinformatie voor een bouwmarkt bijgehouden wordt. Een klant wordt vastgelegd en diens gegevens kunnen veranderd worden. Stel bijvoorbeeld dat de klant verhuist. In het domein zal dan de volgende methode terug te vinden zijn: "`klant.verhuis();`". Het is dus de klant die verantwoordelijk is om zichzelf te verhuizen. Aangezien dit gedrag in het domein zit, kan dit gedrag ook getest worden. Let wel, het domein is nog niet gekoppeld aan een database of een presentatie. Dat is pas de volgende stap.
Zoals al eerder is benadrukt, mogen de services het domein wel kennen. Als het domein dus vorm heeft gekregen, kan de persistentieservice (een database- of file-structuur) aan het domein gekoppeld worden. Hetzelfde geldt voor de presentatieservice, denk aan een Windows- of webpresentatie. Daarnaast kan ook worden gedacht aan autorisatieservices, logging of communicatieservices. Hierbij is van belang, dat het domein in geen geval weet mag hebben van de services om zich heen.
Een mogelijke businessregel voor ons voorbeeldsysteem is bijvoorbeeld dat klanten die verhuizen een kortingsbon toegestuurd krijgen ter besteding bij die bouwmarkt. Deze regel inpassen in het systeem is bij een domeingedreven aanpak niet meer dan een regel toevoegen aan de implementatie van de verhuismethode. Een verandering van het gedrag is dus beperkt tot een verandering in het domein. De businessregel ligt daarmee centraal vast en is snel inzichtelijk te maken.
Terugkomende op de *Separation of Concerns* gaat het Bij DDD gaat het om de scheiding tussen businessregels en de faciliterende services daaromheen. Wat betekent dit in de praktijk? Hieronder volgt een beschrijving van de verschillende stadia van DDD in de praktijk.

Modelleren

Domain Driven Design is een designmethode waarbij de functionaliteit van een applicatie gevangen wordt in één enkel centraal onderdeel. Experts op het gebied van het domein beschrijven allereerst de functionaliteit ervan in natuurlijke taal. Deze taal is erg belangrijk, het bevat immers de concepten en het gedrag van het domein. Modelleren is de kunst van het creëren van een model dat de taal van de expert weergeeft. Terugkoppeling hiervan geeft meteen een eerste controle over de correctheid van het domeinmodel: als de expert bepaalde constructies of benamingen onlogisch vindt klinken, is de kans groot dat het model elementen bevat die niet in het domein thuishoren of niet correct gemodelleerd zijn.
Het domeinmodel bevat geen servicelogica zoals persistentie, of presentatielogica zoals de Graphical User Interface. Het is puur een representatie van de beoogde functionaliteit. Omdat het de verzamelplaats van functionele kennis is, is het belangrijk dat het correct en compleet is. Met 'correct' wordt bedoeld dat de domeinexpert zich kan vinden in de functionaliteit, 'compleet' betekent dat de functionaliteit van de applicatie

Separation of Concerns

Separation of Concerns (of SoC) is een begrip dat in de informatica al in de jaren '70 door Edsger Dijkstra is benoemd [5]. Het is een begrip dat niet beperkt is tot de IT en beschrijft het opdelen van een probleem in subproblemen. Dit opdelen gebeurt zodanig dat de afzonderlijke subproblemen zo min mogelijk overlap met elkaar hebben en makkelijker op te lossen zijn dan het oorspronkelijke, ongedeelde probleem.
Bij het oplossen moet ingezoomd worden op enkel en alleen de verschillende subproblemen. Volgens Dijkstra wil dit echter niet zeggen dat de overige problemen genegeerd mogen worden, ze zijn slechts irrelevant voor het dan behandelde subprobleem. In de IT zijn de technische middelen om dit mogelijk te maken al geïmplementeerd. Denk daarbij aan de mogelijkheid tot modulariseren van de code en gebruik van packages of namespaces, classes en methodes.

volledig in het domeinmodel zit. Businessfunctionaliteit mag daarmee dus niet in services als presentatie, controller of workflow worden geïmplementeerd. 'Compleet' wil echter niet zeggen dat alle businessfunctionaliteit vanaf het begin gemodelleerd moet zijn. Tijdens de ontwikkeling groeit het domeinmodel en komt er steeds meer functionaliteit in. Een domeinmodel is dus niet statisch, het verandert mee met de wijzigingen die in de businessprocessen gewenst zijn.

Programmeren

Tijdens het ontwikkelproces is het domein het onderdeel dat het meest verandert. Het bevat immers het daadwerkelijke gedrag dat de applicatie moet gaan vertonen. Dit centrale karakter van het domein kan tot gevolg hebben dat veranderingen een grote impact hebben op de rest van de applicatie. Tijdens het implementeren van het design is het noodzakelijk deze impact te minimaliseren, waarvoor een aantal technieken beschikbaar zijn.

De eerste techniek is de meest fundamentele, maar gaat ook vaak fout: Object Oriented Programming. De belangrijkste eigenschap van deze manier van programmeren is het encapsuleren van state en gedrag in een object. Dit leidt ertoe dat dit object zelf de verantwoordelijkheid heeft over hoe een functionaliteit tot stand komt en welke gevolgen dat heeft. Neem het verlengen van een abonnement. Vaak gebeurt dit door bijvoorbeeld de einddatum een jaar op te schuiven: `subscription.setEndDate(newEndDate)`. In puristisch OO moet het gedrag echter geëncapsuleerd worden door de functionaliteit aan te bieden: `subscription.renew()`. Dit heeft tot gevolg dat een verandering in het gedrag van verlengen geen invloed heeft op services die van deze functionaliteit gebruik maken. Het effect van de verandering blijft daardoor beperkt.

Na een goed objectgeoriënteerd ontwerp, is het van belang om dit zo te houden. Veranderingen in functionaliteit moeten weer volgens de OO-principes ingevoegd worden, dus de verantwoordelijkheden in het juiste object en voldoende geëncapsuleerd. Code die hier niet aan voldoet, moet zo snel mogelijk gerefactored worden. Dit heeft de hoogste prioriteit van het hele ontwikkelproces. Hou de code schoon! Refactoring heeft een slechte naam in de softwareontwikkeling, omdat projectleiders vrezen dat daarmee het ontwikkelen van functionaliteit tot stilstand komt en ontwikkelaars kostbare tijd verkwanselen aan het herzien van code. Dit is echter onterecht, het is zaak ontwikkelaars in de verschillende technieken voor refactoring te trainen (zie het boek *Refactoring* [4] van Martin Fowler). Voor ontwikkelaars die deze technieken goed kennen is refactoring namelijk geen speciale fase meer, maar iets dat continu, bijna onopgemerkt gebeurt. Een methode die te lang wordt, wordt opgesplitst in twee methodes, twee stukken code die erg op elkaar lijken, maken gebruik van een iets generiekere methode die deze code bevat. Het is belangrijk om de ontwikkelaar hierin te trainen zodat het een gewoonte wordt, niet te onderscheiden van normale ontwikkelactiviteiten.

Een andere methode om de gevolgen van verandering te beperken, is het nastreven van eenvoud in het ontwerp. Dit klinkt tegenstrijdig. Binnen veel projectteams heerst de mening dat elke verandering een lastige aangelegenheid is die veel werk met zich meebrengt. Daarom kan het maar beter in één keer goed, waarbij men schermt met termen als *future-proof, flexibel en configureerbaar*. Vervolgens wordt een model ontworpen waarin gepoogd wordt overal rekening mee te houden. Het model groeit. De ontwikkelaars overzien het geheel niet meer en vervolgens gebeurt wat iedereen al verwachtte: doordat niemand het geheel meer kan overzien, hebben kleine veranderingen grote gevolgen… Introduceer daarom niet meer complexiteit dan nodig is. Als een bepaalde functionaliteit in een latere versie misschien nodig is, vergeet deze functionaliteit dan tot dat latere moment is aangebroken en je zeker weet dat de functionaliteit nodig is. Als een functionaliteit in een andere iteratie gepland staat, houdt er dan geen rekening mee in de huidige iteratie. Houd het zo eenvoudig mogelijk, elke toevoeging maakt het geheel complexer en lastiger om te onderhouden.

Unit testen

Veel traditionele technieken zijn erop gericht veranderingen tijdens het proces zoveel mogelijk te beperken. Vooraf wordt uitgebreid geanalyseerd om de requirements vast te stellen en deze worden zo compleet mogelijk ontworpen, zoals in de vorige paragraaf beschreven. Object Oriented Design en Domain Driven Design proberen deze veranderingen echter te omarmen.

Accidental verus Essential Complexity

In zijn artikel 'No Silver Bullet' [2] beschrijft Frederick Brooks hoe enkele vindingen in de afgelopen 50 jaar het ontwikkelen van software met ordes van grootte hebben weten te versnellen en dat er waarschijnlijk geen andere vindingen te verwachten zijn die dat nog eens voor elkaar gaan krijgen. De argumentatie is gebaseerd op het idee van Aristoteles dat dingen zijn opgebouwd uit *essence*, de onderdelen die iets maken tot wat het daadwerkelijk is, en *accidents*, de onderdelen die weggelaten kunnen worden zonder het wezen te raken, de franje. De orde van grootte van de versnelling in software ontwikkeling is bereikt door accidental complexity uit het proces weg te nemen.

Volgens deze visie vormen veranderingen de natuurlijke evolutie en ontwikkeling van een applicatie.

Ondanks de technieken om de impact van deze verandering te beperken, zal elke verandering nog steeds een bepaalde invloed hebben. Dit is gedeeltelijk ook de bedoeling: een verandering hoort een bepaald effect op de applicatie te hebben, anders zou er geen reden voor het doorvoeren van de verandering bestaan. Dit effect kan op functioneel gebied zijn, maar ook op andere, minder tastbare onderdelen, zoals performance en beveiliging. Zelfs een refactoring heeft een bepaald effect. Het heeft immers tot doel de onderhoudbaarheid van de applicatie te vergroten en daarmee een impact erop. De effecten die echter zoveel mogelijk voorkomen moeten worden, zijn de neveneffecten.

Hierboven hebben we technieken omschreven die tot doel hebben de applicatie inzichtelijk te houden en neveneffecten te voorkomen. Toch kunnen neveneffecten niet altijd voorkomen worden. Daarnaast is een probleem dat neveneffecten niet altijd onderkend worden door de ontwikkelaar. In een relatief kleine applicatie is het al lastig om alle effecten van een verandering te kunnen overzien, in een grotere applicatie is het vrijwel onmogelijk. Deze effecten vinden is het doel van testen.

Unit tests kunnen de bestaande functionaliteit bewaken en de ontwikkelaar behoeden en waarschuwen voor neveneffecten. Unit tests zouden dan ook in geen enkel softwareproject mogen ontbreken. Deze unit tests testen de applicatie op het laagste niveau, namelijk het niveau van de individuele methodes en businessfunctionaliteit van een object. Dit levert vaak een probleem op in projecten omdat deze manier van testen ervan uitgaat dat de geteste functionaliteit op zichzelf staat. Elke koppeling die de functionaliteit met een ander onderdeel heeft, moet daarom aanwezig zijn om de test te laten werken. Externe koppelingen leveren een extra groot probleem op, wat tot gevolg heeft dat er een grote verzameling uitbreidingen op de markt is die allerhande externe koppelingen realiseren of simuleren, zoals databases, mock objects, enterprise java beans, en zelfs PDF-documenten.

Binnen Domain Driven Design zijn deze extensies overbodig. De drager van alle functionaliteit is het domein en het domein heeft geen afhankelijkheden naar externe onderdelen. Een goed objectgeoriënteerd ontwerp zorgt er bovendien ook voor dat alle functionaliteit afgeschermd is, zodat ook binnen een domein de afhankelijkheden beperkt blijven. Dit maakt een domein bij uitstek geschikt is om te unit testen.

Niet alleen bewaakt het unit testen daarom de functionaliteit tegen neveneffecten, maar daarnaast geeft het schrijven van een test inzicht in de afscherming en ontkoppeling van de geschreven code.

Conclusie

Iedere ontwikkelaar kan beamen dat aanpassingen aan applicaties een tijdrovende aangelegenheid zijn. Het vergroten van de aanpasbaarheid van een applicatie kan daarom zeer gewenst zijn. Biedt DDD nu inderdaad de oplossing voor dit probleem? We hebben in dit artikel betoogd dat aan te tonen. We hebben laten zien dat DDD de verantwoordelijkheden opsplitst door de businessregels centraal te stellen. Daarmee worden ze geheel onafhankelijk van de faciliterende services geïmplementeerd. Ze zijn geheel testbaar en veranderingen kunnen sneller en eenvoudiger worden doorgevoerd. De *separation of concerns* in DDD is dus gericht op de kern van software, de businessregels. Helaas is ook DDD niet de oplossing voor alle problemen, geen 'silver bullet' zoals Brooks[2] het zou zeggen. Wat duidelijk is geworden, is dat DDD meer grip geeft op de businessregels en de aanpasbaarheid daarvan dan meer gangbare ontwikkelstijlen. In de praktijk is de grootste uitdaging de manier waarop het domeinmodel met de services gekoppeld wordt. In een volgend artikel meer hierover.

Literatuur

[1] "Patterns of Enterprise Application Architecture", Martin Fowler, www.martinfowler.com

[2] "No Silver Bullet", Frederick Brooks, http://www-inst.eecs.berkeley.edu/~maratb/readings/NoSilverBullet.html

[3] "Domain-Driven Design: Tackling Complexity in the Heart of Software", Eric Evans, Addison-Wesley

[4] "Refactoring: Improving the Design of Existing Code", Martin Fowler, Addison-Wesley Professional, 1999

[5] "On the Role of Scientific Thought", Edsger Dijkstra, http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

[6] "Software Maintenance cost", Jussi Koskinen, http://www.cs.jyu.fi/~koskinen/smcosts.htm

[7] "Software Engineering, 3rd edition", Hans van Vliet, Wiley

[8] "Applying UML and Patterns, 3rd edition", Craig Larman, Prentice Hall

[9] "Hexagonal architecture" Alistair Cockburn http://alistair.cockburn.us/index.php/Hexagonal_architecture