Functional Design Patterns
INF/SCR-04-20

Jeroen Snijders

13 augustus 2004

Samenvatting

Bij de ontwikkeling van software komt een groot aantal problemen in veel verschillende applicaties terug. Wanneer dit gaat om technische problemen kan er in veel situaties gebruikt gemaakt worden van *(technical) design patterns* waarin een technische oplossing voor een algemeen voorkomend probleem geschetst wordt.
Op functioneel gebied wordt er veel minder gebruik gemaakt van algemene oplossingen. Dit terwijl een systeem in de eerste instantie juist wordt ontworpen voor zijn functionaliteit en niet voor zijn technische eigenschappen.
Een manier waarop gemeenschappelijke functionaliteit beschreven kan worden is door middel van *functional design patterns*. Functional design patterns proberen op een gestructureerde manier domeinkennis vat te leggen en als basis te dienen voor generieke oplossingen.
Hoe dit nieuwe conceptuele patroon-type beschreven en gebruikt kan worden en hoe het zich verhoudt tot andere technieken zal, in deze scriptie beschreven worden.

Over dit document

Dit document bevat het openbare gedeelte van mijn onderzoek naar *functional design patterns voor pensioenberekeningen* dat van december 2003 tot en met juni 2004 heeft plaatsgevonden bij *Quinity B.V.* te Utrecht.
Naast dit document zijn er twee vertrouwelijke documenten waarin informatie over dit onderzoek te vinden is:

- *functional design patterns voor Pensioenberekeningen*[Sni04a]
  Dit document bevat de functionele patronen voor pensioenberekeningen waar in *Sectie Patronen voor Pensioenberekeningen*{*60*} naar verwezen wordt (zie *Sectie Structuur beschreven patronen*{*62*}).
- *Implementatie functional design patterns voor Pensioenberekeningen*[Sni04b]
  Dit document bevat gedetaileerde beschrijvingen van de concrete oplossingen waar in *Sectie Patronen voor Pensioenberekeningen*{*60*} (zie naar verwezen wordt (zie *Sectie Toepassing pensioenpatronen*{*63*}).

Deze vertrouwelijke documenten zijn op verzoek bij *Quinity B.V.* door onderwijscommissies in te zien.

Naam Jeroen Snijders

Opleiding Algemene Informatica

Specialisatie Software Technologie

Instelling Universiteit Utrecht

Afstudeerperiode december 2003 t/m juni 2004

Begeleiders Universiteit Jurriaan Hage en Doaitse Swiersta

Begeleiders Quinity Robert Guitink en Lonneke Baas

Inhoudsopgave

1 Inleiding **1**
- Context . . . 1
- Software Ontwerp . . . 2
  - Datamodel . . . 2
  - Functioneel Ontwerp . . . 3
  - Technisch Ontwerp . . . 3
- Webapplicaties . . . 4
  - Verspreiding applicatie . . . 4
  - Functionaliteit . . . 4
  - Ontwikkeling webapplicatie . . . 4
  - Analyse . . . 6

2 Patterns **7**
- Inleiding Patterns . . . 7
  - Anti-pattern . . . 8
  - Pattern omschrijving . . . 8
  - Pattern Language . . . 9
  - Schrijven Patronen . . . 10
- Design Patterns . . . 11
  - Process Patterns . . . 12
  - Analysis Patterns . . . 14
  - Technical Design Patterns . . . 16
  - Interaction or User Interface Design Patterns . . . 18

3 Functional Design Patterns **20**
- Waarom een nieuw patroontype? . . . 20
- Een nieuw patroontype . . . 21
  - Eisen aan het patroon . . . 21
- Toegevoegde waarde . . . 23
- Functional Design Patterns . . . 24
  - Verschillende niveaus . . . 24
  - Omschrijving FDP . . . 26
  - Intentie . . . 26
  - Grafische weergave verbanden patronen . . . 27
  - Afleiden functional design patterns . . . 28
- Voorbeeldpatronen . . . 30
  - Overzicht en Detail . . . 31
  - EntiteitenOverzicht . . . 33

EntiteitenBeheer . . . 35
EntiteitBekijken . . . 38
EntiteitWijzigen . . . 40
Detailscherm . . . 43

4 Toepassing patronen **45**
Concrete Oplossing . . . 45
Generieke Technieken . . . 46
Bestaande Technieken . . . 47
Nieuwe Technieken . . . 49

5 Patterns binnen Quinity **55**
Generieke oplossingen . . . 55
Applicatie-onafhankelijke oplossingen . . . 56
Applicatie/data-specifieke oplossingen . . . 56

6 Patronen voor Pensioenberekeningen **60**
Inleiding Pensioenberekeningen . . . 60
Doel . . . 61
Domeinkennnis . . . 61
Hergebruik Implementatie . . . 62
Structuur beschreven patronen . . . 62
Toepassing pensioenpatronen . . . 63
Procedurele implementatie . . . 64
Object-georiënteerde implementatie . . . 65
Grafische specificatie . . . 66
Vertaling grafische specificatie naar code . . . 67
1. Specificeren van berekening in een model . . . 67
2. Opslaan van berekenings-model in een tussenformaat . . . 69
3. De relevante gegevens uit het tussenformaat halen . . . 70
4. Op basis van de relevante gegevens code voor de berekening genereren . . . 71
Zin van patronen voor (pensioen)berekeningen . . . 74

7 Conclusie **78**
Toekomst . . . 79

A Model Driven Design **80**
Platform . . . 80
Viewpoints . . . 81
Mappings . . . 82
Toepasbaarheid MDA . . . 82

B Domain-Driven Design **85**
Traditionele technieken . . . 85
Domain-Driven Design . . . 86
Ubiquitous Language . . . 87
Model . . . 87
Rol van objecten . . . 88
Toepasbaarheid . . . 90

Ontwikkelmethodes 90
Domain-Driven Design en Model-Driven Architecture 90
Domain Driven Design en patterns 91

C Feature Oriented Programming **93**
Features 93
Feature Modellen 94
Features en FDPs 95

Bibliografie **96**

Hoofdstuk 1

Inleiding

Context

Veel administratieve taken zijn tegenwoordig geautomatiseerd.
Er bestaan veel standaard administratieve pakketten voor deze taken. Deze applicaties zijn bedoeld om gebruikt te worden door verschillende bedrijven. Het gevolg hiervan is dat oplossingen vaak te generiek zijn en niet precies aansluiten op de behoeften van het bedrijf.
In plaats van een standaardpakket kan een bedrijf er voor kiezen om een maatwerkpakket te laten ontwikkelen dat precies voldoet aan de eisen die het heeft. Het voordeel van een applicatie op maat is dat de klant invloed heeft op:

- de functionaliteit van het systeem,
- het uiterlijk van het systeem,
- de prioriteit van bepaalde aspecten binnen een systeem,
- de gebruikte technieken, en
- integratie van het nieuwe pakket met de bestaande infrastructuur.

Een nieuw maatwerkpakket heeft echter ook grote nadelen:

Kosten De ontwikkelkosten van het systeem kunnen niet worden verdeeld over meerdere bedrijven. Dit betekent dat een maatwerk-applicatie over het algemeen een stuk duurder is dan een licentie voor een standaard pakket. Ook onderhouds/support-kosten zouden hoger kunnen zijn omdat kennis over het systeem specifieker is.

Tijd Het ontwikkelen van een nieuw maatwerksysteem duurt veel langer dan het aanschaffen van een bestaand pakket, dat bijna direct beschikbaar is.

Fouten De kans op fouten en bugs is, over het algemeen, in een nieuw systeem groter dan bij een systeem wat al langer wordt gebruikt. Dit komt doordat de grootste/meest optredende fouten vaak al in een vroeg stadium tijdens het gebruik opgemerkt en opgelost zullen worden.

Veel maatwerk oplossingen binnen een bepaald domein bieden min of meer dezelfde functionaliteit. Bijvoorbeeld: het beheren en onderhouden van producten en relaties, het bijhouden van mutaties, het uitvoeren van bepaalde berekeningen en het definiëren van vragenlijsten komt terug in verschillende administratieve

applicaties. Pas op detailniveau zullen deze functies van elkaar verschillen.
Een alternatief voor het ontwikkelen van een geheel nieuw maatwerkproduct zou zijn het samenstellen van een systeem uit standaard oplossingen en deze 'op maat' te maken voor een klant. Dit maakt het mogelijk om de voordelen van een standaardpakket te combineren met die van een maatwerkpakket. Een systeem op maat zal op deze manier goedkoper, sneller en met minder fouten ontwikkeld kunnen worden.
Om standaard (of gegenereerde) oplossingen te kunnen hergebruiken is het noodzakelijk om terugkerende patronen binnen applicaties vast te leggen.
Dit onderzoeksproject vindt plaats bij *Quinity B.V.* te Utrecht (Nederland). Dit bedrijf ontwikkelt maatwerkinternetapplicaties, hoofdzakelijk voor administratieve toepassingen en internetbankieren.
Binnen dit bedrijf wordt geprobeerd om zoveel mogelijk functionaliteit te hergebruiken tussen verschillende systemen, om op deze manier goedkopere en betere maatwerkoplossingen aan te kunnen bieden.

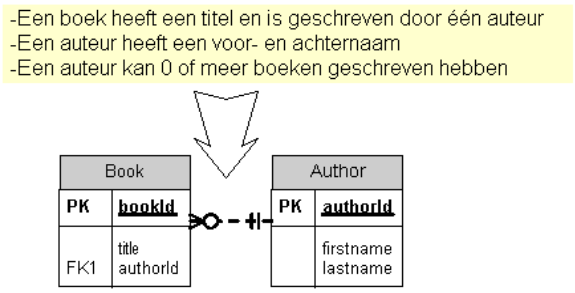
Software Ontwerp

Een belangrijke fase in de ontwikkeling van een nieuwe applicatie is het vertalen van de behoeften van een klant naar een ontwerp voor het systeem.

Datamodel

In de meeste administratieve toepassingen speelt de database een zeer belangrijke rol. Bijna alle acties die de gebruiker uitvoert zullen invloed hebben op de toestand van de database. De structuur van de database is van groot belang voor een correct en efficiënt gedrag van het systeem. De lay-out van de database wordt bepaald door het datamodel. Het datamodel geeft aan welke tabellen er zijn, hoe deze tabellen gerelateerd zijn en wat er in deze tabellen wordt opgeslagen.
Er zijn twee belangrijke technieken om een goed datamodel te ontwerpen [Bla90]:

Top-down Gegevens voor het datamodel worden verzameld tijdens interviews met de klant. De klant geeft aan welke entiteiten er straks binnen zijn systeem zijn, wat de eigenschappen van deze entiteiten zijn en aan welke eisen zij moeten voldoen. Aan de hand van deze gegevens wordt een *Entity Relationship diagram* gecreëerd (zie *Figuur 1.1*).



Figuur 1.1: ER-diagram

Bottom-up De benodigde informatie voor het opstellen van een datamodel wordt verzameld aan de hand van bestaande systemen. Aan de hand van schermoverzichten van formulieren en rapporten kunnen bubble-charts worden getekend waarin de afhankelijkheden tussen de verschillende data aangegeven worden. Deze bubble-charts kunnen vervolgens worden gebruikt om een *Entity Relationship diagram* voor het onderliggende database te maken. Het is ook mogelijk om de lay-out van bestaande databases te gebruiken. Het gevaar hierbij is echter dat naïef hergebruik hiervan kan leiden tot het overnemen van ontwerpfouten, bijvoorbeeld als van bepaalde velden niet duidelijk is wat zij precies doen.

In beide gevallen kunnen normalisatie technieken toegepast worden om het datamodel de juiste structuur te geven.

Het grote gevaar bij de top-down methode is dat de klant vergeet bepaalde concepten te noemen, of omdat hij ze zelf niet gebruikt, of omdat ze voor hem zo vanzelfsprekend zijn dat hij er niet aan denkt.
De beste aanpak is dan ook om beide methodes te combineren. Door dit te doen kan de ontwerper zijn kennis over het oude systeem gebruiken om de eisen van het nieuwe systeem tijdens interviews met de klant precies te bepalen. Op het moment dat de klant iets essentieels vergeet te noemen is de kans groot dat de ontwerper hier toch aan de hand van de bottom-up-analyse achterkomt.

Functioneel Ontwerp

De functionele eisen van het nieuwe systeem worden verzameld in het functioneel ontwerp. Het is heel belangrijk dat klant en ontwerper/ontwikkelaar tijdens het opstellen van de de requirements en functioneel ontwerp hetzelfde conceptuele model van de applicatie hebben. Als dit niet het geval is, is de kans groot dat de applicatie uiteindelijk niet aan de eisen/verwachtingen van de klant zal voldoen.
Informatiefuncties kunnen beschreven worden in tekst of met behulp van modelleer technieken als use-case-diagrammen in UML (*Unified Modeling Language* [OMG]). Jammergenoeg is UML niet expressief genoeg om alle mogelijke functionaliteit op een inzichtelijke manier te modelleren. Een bijkomend nadeel is dat niet alle klanten UML begrijpen.
De functionele beschrijving kan worden aangevuld met voorbeeldschermen van de te ontwikkelen applicatie.
Het is verstandig om al in een vroeg stadium van het ontwerpproces een prototype (zonder databasefunctionaliteit) te ontwikkelen (Building a Low Fidelity Prototype[SA03]). Dit prototype geeft een ondubbelzinnig beeld van het functioneel ontwerp en zorgt er voor dat klant en ontwikkelaar op één lijn zitten.

Technisch Ontwerp

In het algemeen wordt er na het maken van het functioneel ontwerp een technisch ontwerp gemaakt. In het technisch ontwerp wordt, vaak met behulp van UML-diagrammen (component-model-diagrammen, sequentie-diagrammen et cetera) en class-diagrammen, aangegeven hoe het systeem uiteindelijk geïmplementeerd moet worden. Het technisch ontwerp bevat alle informatie die relevant is voor

de ontwikkelaars om het systeem uiteindelijk te kunnen bouwen. Dus ook informatie over gebruikte technieken, eisen aan de infrastructuur et cetera.
Het gebruik van een basis framework dat alle low-level functionaliteit zoals databasetoegang en request-afhandeling afhandelt, maakt het mogelijk om applicaties te bouwen vanuit een hoger abstractie niveau. Low-level technische details zitten verborgen in het framework, hetgeen het maken van een technisch ontwerp vergemakkelijkt. Op het moment dat alle implementatiedetails afgeleid kunnen worden vanuit het functioneel ontwerp is een uitgebreid technisch ontwerp zelfs niet eens nodig.

Webapplicaties

De laatste jaren worden veel informatiesystemen als webapplicatie geïmplementeerd. Het gaat hierbij vooral om administratieve applicaties waarbij het belangrijk is dat er gebruik gemaakt wordt van één gedeelde databron. Bij een webapplicatie beschikt de gebruiker niet over een client-versie van de applicatie op zijn eigen computer. Gebruikers maken via hun web-browser, over het internet, contact met een centrale server waar de applicatie draait. Via webpagina's communiceert de gebruiker vervolgens met het systeem.

Verspreiding applicatie

Het grote voordeel van webapplicaties is dat op een eenvoudige manier een databron gedeeld kan worden en dat gebruikers geen aparte client-applicatie nodig hebben voor het systeem. De gebruiker zal op het moment dat hij het systeem wil gebruiken altijd de meest recente versie van de interface, in de vorm van webpagina's krijgen van de server.
Dit is met name belangrijk voor publieke systemen voor doeleinden als internetbankieren en internetshopping, waarbij het belangrijk is dat iedereen van de applicatie gebruik kan maken zonder speciale software te hoeven installeren.

Functionaliteit

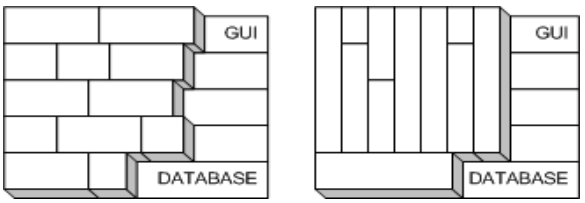
De meeste webapplicaties werken volgens het 'thin-client' principe; vrijwel alle logica wordt hierbij overgelaten aan de server-applicatie. In het geval van datageoriënteerde systemen -waarvoor webapplicaties hoofdzakelijk gebruikt worden- is dit vaak een zeer geschikte oplossing. Dit aangezien de server snellere toegang heeft tot de database en er relatief weinig rekenwerk nodig is. De gebruikerskant van het systeem, de webpagina's in de browser, dient enkel als interface tussen de gebruiker en de applicatie op de server.

Ontwikkeling webapplicatie

Webapplicaties worden over het algemeen anders ontwikkeld dan de 'normale' softwaresystemen [WM01, Pre98].

> *"I fear that poor design practices could introduce latent defects that will make Y2K look like child's play"*
>
> Watts Humprey over webapplicaties [Pre98]

Bij de ontwikkeling van webapplicaties wordt in de meeste gevallen geen gebruik gemaakt van een black-box principe zoals we dat kennen bij de ontwikkeling van 'normale', componentgebaseerde software.
Bij het black-box principe worden componenten onafhankelijk door de subteams ontwikkeld. Iedere component behandeld een afgebakend technisch gebied van de applicatie en communiceert via een dunne interface met de andere componenten. De subteams hoeven zich hierdoor maar te specialiseren op een klein gedeelte van het systeem. Zo kunnen er subteams zijn voor de interface met de gebruiker, data-uitwisseling met oude systemen, printfunctionaliteit, databasecommunicatie, client-server-communicatie, hulpfunctionaliteit, specifieke bedrijfsfuncties et cetera. Deze componenten kunnen vervolgens laag voor laag gecombineerd worden om het systeem op te bouwen (zie linkerkant *Figuur 1.2*).



Figuur 1.2: 'Black-box' en 'breedte' ontwikkeling

Tijdens de ontwikkeling van webapplicaties wordt het systeem meestal in de breedte opgebouwd [WM01]. In plaats van met 'technische blokken' wordt er veelal met 'functionele blokken' gewerkt. Vanuit een (werkende) basisversie breiden de subteams het systeem uit met nieuwe functionaliteit. Bij het toevoegen van een functionaliteit zullen over het algemeen wijzigingen moeten worden aangebracht over alle lagen van het systeem; van data uit de database tot de representatie naar de gebruiker.
De scheiding tussen de verschillende disciplines die de subteams uitvoeren bij een functionele opdeling is minder duidelijk dan bij een technische opdeling (zie rechterkant *Figuur 1.2*). Het is over het algemeen lastig om overlap te voorkomen in functionele blokken. De kans is dan ook groot dat teams soortgelijke problemen op zullen moeten lossen voor verschillende (functionele) aspecten van het systeem.
Dit laatste is niet specifiek een probleem bij webapplicaties, maar bij alle applicaties die in functionele blokken opgebouwd worden.
Traditionele ontwerp/ontwikkelmethodes worden relatief weinig gebruikt bij de ontwikkeling van webapplicaties. Dit heeft te maken met aantal belangrijke punten [WM01]:

- *Snelle oplevering*
  De periode van aanvraag tot oplevering is bij webapplicaties over het algemeen veel korter dan bij 'normale' applicaties (drie tot zes maanden). Daarnaast wordt er meestal gewerkt door kleine ontwikkelteams (minder dan tien personen).
- *Testen*
  Het testen en het inbouwen van functionaliteit gebeurt vaak tegelijkertijd. Door de beperkte tijd is er vaak geen aparte testperiode mogelijk.
- *Documentatie*

Het documenteren van een systeem is essentieel om het te kunnen onderhouden of in de toekomst uit te breiden. Toch wordt documentatie vaak niet als een belangrijk onderdeel gezien tijdens de ontwikkeling van een webapplicatie. Mede door de tijdsdruk wordt er vooral gekeken naar de snelheid van de oplevering zelf en daarbij heeft het maken van documentatie enkel een vertragende werking.

- *Analyseface*
  Bij veel projecten is er geen aparte analysefase waarin alle eisen van de klant en gebruikers zorgvuldig worden vastgelegd om daarna pas aan de implementatie te beginnen. De analyse/ontwerp en implementatiefase zijn vaak sterk verweven. Tijdens de ontwikkeling (en tevens testfase) worden dan ook vaak nog aanpassingen aan het ontwerp gemaakt.

Analyse

Naast het niet halen van een deadline, hetgeen ook bij reguliere projecten applicaties voorkomt, zorgen vooral het gebrek aan een goede analyse en het ontbreken van een goed functioneel ontwerp ervoor dat veel projecten falen. Achteraf blijkt vaak dat het systeem toch niet precies doet wat de klant en, nog belangrijker, de eindgebruikers precies voor ogen hadden.
De term 'User Centered Design' komt vaak naar voren binnen de (web)informatica [BBBR01]. Toch blijkt dat er in de praktijk vooral naar de eisen van de klant gekeken te worden in plaats van de eindgebruiker. De beschikbare tijd en (domein)kennis van de ontwerpers zijn vaak niet voldoende om een zorgvuldige analyse uit te kunnen voeren.
Bij het ontbreken van een zorgvuldige analyse zal er vertrouwd moeten worden op de kennis en ervaring van de ontwikkelaar; of kunnen er *patterns* gebruikt worden waarin ervaring uit voorgaande projecten is vastgelegd, om ervoor te zorgen dat zij toch de benodigde kennis hebben.

Hoofdstuk 2

Patterns

Inleiding Patterns

Patterns, oftewel patronen, proberen op een inzichtvolle manier een vaak terugkerend probleem en zijn bewezen oplossing te beschrijven. Dit kan een probleem zijn in verschillende contexten. Een pattern geeft naast instructieve informatie over de oplossing van het probleem zelf ook aan waarom de oplossing nodig is.

> *"I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself."*
>
> Jim Coplien [Cop96]

Of in het Nederlands: Je kan iemand uitleggen hoe hij een jurk moet maken door met kniplengtes en hoeken precies aan te geven hoe hij de stof moet knippen. Die persoon zal echter geen idee hebben waarmee hij precies bezig is, wat hij aan het maken is en of het eindresultaat uiteindelijk is wat hij wil.
In plaats daarvan kan je hem ook een patroon geven; naast hoe de jurk gemaakt moet worden beschrijft het patroon in feite ook de jurk zelf.
Bij het lezen van patronen wordt enig inzicht verwacht van de lezer. De lezer moet zelf bepalen *of* en *hoe* hij het patroon gebruikt. Patronen liggen niet vast; het kunnen improviseren op patterns is één van de grote voordelen. Het pattern zal duidelijk moeten maken waarom dit een oplosing is voor het probleem zodat potentiële gebruikers kunnen zien in hoeverre het patroon aan te passen is om te gebruiken binnen hun context.
Het oudste en misschien wel bekendste gebruik van patterns is op het gebied van architectuur [ea97]. Deze patronen beschrijven standaarden voor ruimtelijke ordening, gebouwontwerp en oplossingen voor (bouwtechnische) constructieproblemen [Jac]. De informatica probeert op soortgelijke manier, dus middels het definiëren van patronen, binnen software ontwerp het proces van softwareontwikkeling te stroomlijnen.

Anti-pattern

Niet alleen bewezen oplosingen in de vorm van patterns zijn nuttig; In veel gevallen kan juist het tegenovergestelde van groot belang zijn. Het tegenovergestelde van een pattern is een anti-pattern [Bro99, Ant98].

> *"An anti-pattern is something that looks like a good idea, but which backfires badly when applied."*
>
> Jim Coplien

Of in het Nederlands: een anti-pattern is iets dat op het eerste gezicht een goed idee lijkt maar als het daadwerkelijk wordt gebruikt toch niet zo goed blijkt.
Een anti-pattern beschrijft hoe je van een probleem naar een slechte oplossing kan gaan, of van een slechte naar een goede oplossing. Anti-patterns kunnen zeer bruikbaar zijn om slechte ontwerpbeslissingen te herkennen en betere oplossingen te verzinnen.
Een eenvoudig voorbeeld van een anti-pattern is het BoatAnchor[Bro99]-patroon.
Tijdens een project gebeurt het vaak dat (van bovenaf) wordt aangegeven dat een bepaald stuk software/hardware gebruikt moet worden omdat dit in een voorgaand project goed werkte of omdat er veel geld voor betaald is. In de huidige project hoeft dit echter helemaal geen slimme keuze te zijn.
Dit is te vergelijken met het anker van een schip: Een anker is in een boot heel nuttig maar tijdens bijvoorbeeld een bergwandeling zal het enkel tot last zijn.
Het doel van anti-patterns is net als bij normale patterns om ervaringen vast te leggen; in plaats van om goede ervaringen gaat het in dit geval juist om slechte ervaringen. Deze kunnen echter net zo belangrijk zijn.

Pattern omschrijving

Potentiële gebruikers moeten in staat zijn om op een eenvoudige manier te zien of een patroon in hun situatie toepasbaar is. Om dit voor elkaar te krijgen is het verstandig om probleem/oplossing volgens een vast patroon te beschrijven. Een veelgebruikte vorm ([MD98]) bevat de volgende secties:

Name (naam) Een patroon moet een duidelijke naam hebben om ernaar te kunnen refereren.

Context Situatie waarin het probleem het voordoet.

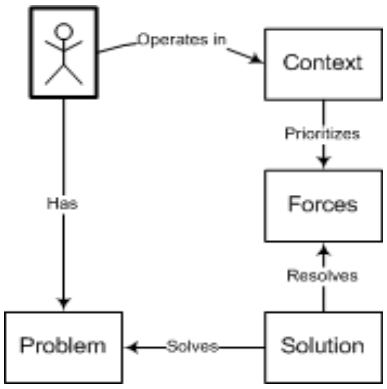
Forces (overwegingen) Welke overwegenigen zullen er gemaakt moeten worden voordat het patroon wordt toegepast.

Problem (probleem) Het op te lossen probleem.

Solution (oplossing) De oplossing voor het probleem.

De relaties tussen de verschillende secties is grafisch weergegeven in *Figuur 2.1*.
Het boek van Eric Gamma et al.[GHJV95] gebruikt een sjabloon met de volgende secties (ook wel "GoF Form"):

- Pattern Name and Classification (naam en classificatie)
- Intent (intentie)



Figuur 2.1: Relaties tussen secties

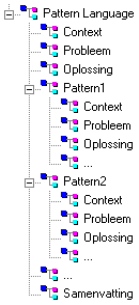
- Also Known As (ook bekend als)
- Applicability (toepasbaarheid)
- Structure (structuur)
- Participants (deelnemers)
- Collaborations (samenwerkingen)
- Consequences (consequenties)
- Implementation (implementatie)
- Sample Code (voorbeeld code)
- Known Uses (gebruiksvoorbeelden)
- Related Patterns (gerelateerde patterns)

In deze schrijfwijze is het probleem/oplossing-paar minder duidelijk aanwezig, maar kan op een informelere manier de hele intentie en motivatie van het patroon beschreven worden.
Het gebruik van een standaard formaat structureert de beschrijving en maakt het makkelijker voor een gebruiker om patronen die niet van toepassing zijn op het probleem over te slaan. Het is niet verboden om af te wijken van deze regels maar voor de herbruikbaarheid is het wel verstandig om binnen een pattern-type ervoor te zorgen dat de beschrijvingen consequent zijn.

Pattern Language

Een probleem/oplossing kan worden beschreven in een aparte pattern-definitie maar in veel gevallen staat een pattern niet op zichzelf; patterns kunnen een partiële oplossing voor een groter probleem bieden. In dit geval is de relatie ten opzichte van andere patterns die van toepassing zijn op het probleem heel belangrijk. Het is mogelijk om meerdere losse patronen te definiëren en de relaties tussen deze patronen te beschrijven in de pattern-definities zelf (in de 'gerelateerde patterns'-sectie).
Dit is in veel gevallen geen handige oplossing: door de platte structuur wordt de lezer gedwongen om alle patronen door te lezen voordat hij een goed beeld kan vormen van de relaties tussen de patronen, het probleem dat ze oplossen en

hoe ze dit oplossen.
Een betere oplossing is om een pattern language te definiëren die de losse patronen omvat en een samenvatting geeft van de patronen binnen de taal, het probleem dat ze samen oplossen en hoe ze dit doen (zie *Figuur 2.2*). Door het gebruik van een pattern language worden de relaties tussen de verschillende patronen gestructureerd, maar blijft het nog steeds mogelijk om individuele patronen te hergebruiken (hetgeen bij het definiëren van één groot patroon niet mogelijk is).



Figuur 2.2: Structuur Pattern Language

Schrijven Patronen

Het schrijven van goede herbruikbare patronen is erg lastig. Het gebruik van een standaardformaat kan ervoor zorgen dat het patroon voldoet aan een algemene structuur en 'overzichtelijk' is. Daarentegen kan het niet garanderen dat het patroon ook begrijpelijk is voor de lezer. Het conceptuele model van de schrijver moet op een dusdanige manier worden opgeschreven dat het:

- voor de lezer, die de specifieke kennis niet heeft, begrijpelijk is,
- alle essentiële kenmerken van het patroon beschrijft, en
- alleen de essentiële kenmerken van het patroon beschrijft.

Niet iedere goede expert is een goede pattern-schrijver. Er zijn wat dat betreft drie soorten mensen [Beck96]:

- Personen die patronen om zich heen niet zien.
- Personen die de patronen wel zien maar niet in staat zijn ze vast te leggen.
- Personen die overal patronen herkennen en in staat zijn ze duidelijk te beschrijven.

Alleen experts uit de laatste categorie zullen in staat zijn om patronen op een duidelijke manier over te brengen op gebruikers. Maar ook bij deze categorie is het een illusie om te denken dat zij patronen in één keer goed op papier kunnen zetten.
Het beschrijven van een patroon is vaak een (interactief) iteratief proces tussen de schrijver en lezers. Lezers geven aan wat zij niet direct begrijpen, en antwoorden op deze vragen kunnen vervolgens verwerkt worden in de patroonomschrijving.

Design Patterns

Design patterns zijn een populair onderzoeksgebied binnen de informatica sinds begin jaren negentig. Een design pattern beschrijft een vaak voorkomend (software) ontwerp patroon en zijn oplossing. Deze patterns zijn erg nuttig omdat zij het mogelijk maken om bestaande oplossingen te hergebruiken tijdens de ontwikkeling van nieuwe software. Het definiëren van design patterns zorgt ervoor dat het wiel maar één keer uitgevonden hoeft te worden. Dit scheelt veel ontwikkeltijd en het maakt applicaties minder foutgevoelig aangezien zij gebruik kunnen maken van bewezen technieken.
Een ander groot voordeel van het gebruik van design patterns ligt op het gebied van communicatie: patronen introduceren een extra abstractieniveau tussen het probleem en de oplossing. Dit maakt het mogelijk om de patroonnaam te gebruiken om een probleem en zijn oplossing te beschrijven. De communicatie tussen ontwerpers en ontwikkelaars wordt op deze manier vergemakkelijkt. De patronen maken het ontwerp ook begrijpelijker omdat de (lagere niveau) oplossing verborgen wordt.
Een informatiesysteem kan vanaf verschillende perspectieven bekeken worden. Er zijn dan ook patterns vanuit verschillende perspectieven.
In de volgende paragrafen zullen verschillende patroontypes in de informatica beschreven worden volgens een standaard patroonbeschrijvingsmethode (in feite een meta-pattern).

Process Patterns

Context

Procedures tijdens de levenscyclus van een applicatie.

Probleem

Hoe standaardiseer je procedures in het ontwerp/ontwikkel/onderhoudproces van een applicatie?

Overwegingen

- Goede ontwerp/ontwikkelgewoontes kunnen het ontwikkelproces structureren, hetgeen kan leiden tot sneller en beter resultaat.
- Zonder vastgestelde procedures zullen ontwerpers/ontwikkelaars volgens hun eigen gewoontes werken. In veel gevallen zal dit leiden tot projecten met een grote variatie aan stijlenl die een grotere kans va falen hebben.

Oplossing

Definiëer *process patterns* (PP's) waarin goede standaardwerkwijzen worden vastgelegd.

Rationale

Process patterns zijn geen echte *design patterns* aangezien zij niet expliciet het ontwerp/implementatie van een applicatie zelf beínvloeden. PP's beïnvloeden alleen het *proces* van ontwerp naar en realisatie van een systeem. Desalniettemin spelen PP's een grote rol bij het succes of falen van een systeem.

Speciale Beschrijvings Technieken

Sommige process patterns kunnen duidelijk gemaakt worden met behulp van flowcharts die een aantal stappen van het ontwerp/realisatie op een begrijpelijke manier proberen te visualiseren.

Voorbeeld

Building a Low Fidelity Prototype[SA03]
Dit patroon beschrijft de zin van het maken van een lichtgewicht prototype van een applicatie in een vroeg stadium van het ontwerp.
Het prototype bevat een eenvoudige versie van de user-interface maar mist de eigenlijke functionaliteit. Dit prototype kan worden gebruikt door de klant en ontwerper/ontwikkelaar om overeenstemming te bereiken over de functionaliteit van een applicatie en hoe deze in de user-interface verwerkt is. Het praten over het systeem wordt versimpeld omdat er al een concrete implementatie op tafel ligt die ervoor zorgt dat alle partijen hetzelfde beeld van de applicatie hebben.

Gebruik

Veel bedrijven gebruiken process patterns om het pad van ontwerp naar realisatie en standaardprocedures binnen het bedrijf te beschrijven. Quinity gebruikt bijvoorbeeld de DSDM-techniek [DSD] tijdens het ontwerpproces, inclusief het maken van een Building a Low Fidelity Prototype[SA03] in de functionele ontwerpfase.

Onderzoek

IBM en universiteiten zoals Concordia University (Montreal Canada) doen veel onderzoek naar User-Centered-Design [BBBR01, SA03]. Binnen UCD worden veel process patterns beschreven voor richtlijnen tijdens het ontwerp van software.

Analysis Patterns

Context

Concrete concepten binnen het (bedrijfs)leven moeten gerepresenteerd worden in een informatiesysteem.

Probleem

Hoe definieer je standaarden voor de virtuele representaties van bedrijfsconcepten?

Overwegingen

- Voor sommige concrete bedrijfsconcepten die verschijnen in veel systemen is het lastig om een eenvoudige/deterministische representatie te vinden in software.
- Representaties van bedrijfsconcepten kunnen (her)gebruikt worden in veel situaties.
- Het hebben van standaardrepresentaties vergemakkelijkt het praten over software en het ontwerp ervan.

Oplossing

Definieer *analysis patterns* (AP's) om te beschrijven hoe algemene constructies binnen het (bedrijfs)leven gerepresenteerd kunnen worden. Bespreek de voordelen en nadelen van deze representatie zodat ontwerpers/ontwikkelaars kunnen zien of deze representatie voldoet aan hun eisen.
AP's beschrijven hoe de objectrepresentaties van 'iets' uit de realiteit, bijvoorbeeld een artikel, gebruiker of mutatie, eruit moet zien. In een applicatie zijn deze concepten meestal verpakt in *business-objecten*. Business-objecten zijn in feite implementaties van de onderliggende analysis patterns.

Rationale

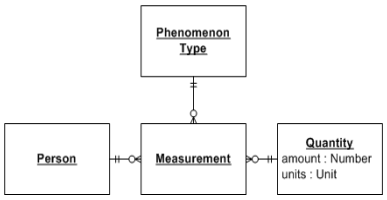
Door het definiëren van analysis patterns voor bedrijfsconcepten is het mogelijk om verschillende business-objecten voor hetzelfde concept af te leiden. De patroonomschrijving documenteert tevens de afgeleide business-objecten.

Speciale Beschrijvings Technieken

Analysis patterns hebben vaak direct betrekking op gegevensstructuren; een gebruikelijke notatie hiervoor zijn ER-diagrammen.

Voorbeeld

Measurement[Fow96]
Dit patroon beschrijft het los vastleggen van eigenschappen die bij een persoon horen.
Dit patroon kan gebruikt worden om metingen van een persoon te representeren (bijvoorbeeld in een ziekenhuis). Als er weinig verschillende metingen mogelijk



Figuur 2.3: ER Measurement Patroon

zijn is het mogelijk om deze eigenschappen als attributen voor de persoon-entiteit bij te houden, maar als dit te ingewikkeld wordt is het beter om het eigenschappen apart op te slaan en deze vervolgens te linken met de bijbehorende persoon (zie *Figuur 2.3*). Dit patroon is flexibeler en makkelijker uit te breiden dan wanneer er met vaste attributen van een entiteit gewerkt wordt.

Gebruik

Veel bedrijven gebruiken standaard mappings tussen concrete producten en hun virtuele representaties in de vorm van business objecten. De term analysis patterns is hiervoor echter vrij onbekend en veel van hen hebben dan ook geen flauw idee dat ze feitelijk een soort van AP's gebruiken. In het algemeen is het vrijwel onmogelijk om deze ad-hoc mapping te hergebruiken buiten het bedrijf (hetgeen vaak ook niet de bedoeling is).

Onderzoek

Martin Fowler wordt gezien als de uitvinder van AP's. Zijn werk [Fow96] heeft veel informatici genspireerd eind jaren negentig. Tegenwoordig is er vrij veel onderzoek naar analysis patterns op universiteiten die betrokken zijn bij de PLoP conferenties (Pattern Languages of Programs) zoals Wenen [HGS02], Illinois en Florida. Maar wereldwijd gezien zijn AP's toch vrij onbekend en is er naast [Fow96] weinig literatuur met voorbeelden van analysis patterns of over hoe ze gebruikt worden.

Technical Design Patterns

Context

Technische implementatieproblemen die verschijnen in veel applicaties.

Probleem

Hoe kun je oplossingen beschrijven voor veelvoorkomende (technische) problemen.

Overwegingen

- Het vinden van oplossingen voor problemen tijdens de implementatie van een systeem kan veel tijd kosten.
- Niet elke programmeur heeft genoeg kennis om specifieke problemen op een juiste manier op te lossen.
- Veel technische problemen kunnen op een soortgelijke manier opgelost worden.

Oplossing

Definieer een *technical design pattern* (TDP) dat het algemene technische probleem en oplossing beschrijft en (her)gebruikt kan worden in situaties dat het probleem optreed.
In principe is het de bedoeling dat deze beschrijving programmeertaalonafhankelijk is, in de praktijk gaan technische patronen meestal uit van een object-georiënteerde taal.

Rationale

Technical design patterns leveren oplossingen op architectuur/object/code-niveau voor problemen binnen het technisch ontwerp. Het kennen van deze patronen maakt het leven van een programmeur een stuk minder ingewikkeld.

Speciale Beschrijvings Technieken

De gebruikelijke manier om de structuur van technische weer te geven is met behulp van de modelleertaal UML [OMG].
De meeste TDP's-omschrijvingen bevatten enkele *code-voorbeelden* om duidelijk te maken hoe het patroon toegepast kan worden. Deze concrete voorbeelden kunnen ook worden gebruikt om taalspecifieke implementatieproblemen naar voren te laten komen. Soms is het echter beter om dit soort details te verbergen en in plaats van een concrete programmeertaal pseudo-code te gebruiken.

Voorbeeld

Singleton[GHJV95]
Dit patroon zorgt ervoor dat er maar één instantie van een klasse tegelijk aanwezig kan zijn binnen een systeem.

Listing 2.1: Singleton klasse

```
public class Singleton {

   static private Singleton instance = null;

   private Singleton() {}

   static public Singleton getInstance() {
      if(null == instance) {
         instance = new Singleton();
      }
      return instance;
   }
}
```

Door een klasse geen `public` constructor te geven maar een `static` methode die altijd een referentie naar dezelfde instantie van de klasse retourneert (zie *Code 2.1*) kan worden voorkomen dat er tijdens de executie van een programma meer dan één instantie van een klasse aangemaakt kan worden. Het patroon zorgt er daarnaast voor dat verschillende delen van een programma één instantie van een klasse op een eenvoudige manier kunnen delen.

Gerelateerde Patronen

Vanwege de grote hoeveelheid technische patronen worden TDP's vaak onderverdeeld in een aantal categoriën[Dev02]:

Creational patterns creatie/configuratie van objecten (SINGLETON[GHJV95])

Structural patterns het loskoppelen van interface en hun implementatie in classes (MODEL-VIEW-CONTROLLER[Cop96])

Behavioral patterns interactie tussen objecten (OBSERVER[GHJV95])

Gebruik

TDP's zijn de bekendste design patterns. Wanneer iemand het heeft over 'een design pattern' dan zal hij waarschijnlijk een technical design pattern bedoelen. De belangrijkste oorzaak hiervan is het boek 'Design Patterns' door de *Gang of Four*[GHJV95] wat door veel programmeurs wordt gezien als de 'Bible of Patterns'. Technische patronen worden dan ook in veel bedrijven gebruikt [ea96]. n hogere programmeertalen zijn veel (low-level) technische patronen taalconstructies geworden; voorbeeld hiervan zijn overerving, interfaces en synchronisatietechnieken.

Onderzoek

Naar het voorbeeld van het werk van *the Gang of Four* doen veel onderzoeksgroepen onderzoek naar TDP's op verschillende vlakken en op verschillende niveaus. Ondanks het bestaan van enorme collecties TDP's leiden nieuwe programmeertechnieken tot nieuwe mogelijkheden voor patronen.

Interaction or User Interface Design Patterns

Context

De gebruiker bestuurt en heeft interactie met een informatiesysteem met behulp van een user interface.

Probleem

Hoe kun je standaarden vastleggen voor de interactie van de gebruiker met een systeem.

Overwegingen

- Alle interactie tussen het systeem en de gebruiker gaat via de user-interface, het moet dan ook in de UI duidelijk zijn voor de gebruiker wat hij precies kan doen met het systeem.
- Een user-interface die niet duidelijk of intutief is zal het leerproces vertragen en *kan* de productiviteit van een gebruiker verlagen.
- Een vertrouwd reagerende/uitziende UI zal de gebruiker aansporen het systeem te gebruiken.

Oplossing

Definieer *user interface patterns* (UDP's) voor bewezen interactietechnieken tussen de gebruiker en het systeem.

Rationale

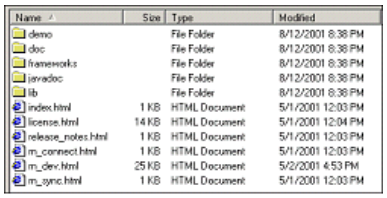
Door het gebruik van patronen voor de interactie tussen gebruiker en systeem, zijn (graphical) user-interfaces binnen en tussen verschillende applicaties consistenter. Consistentie vergemakkelijkt het leerproces aangezien gebruikers geen nieuwe (navigatie) technieken, toetscombinaties, iconen et cetera, hoeven te leren om overweg te kunnen met een nieuw systeem. Over het algemeen verhoogt dit de productiviteit.UDP's maken het mogelijk om over een user-interface te praten zonder dat het in eerste instantie nodig is om lay-out schetsen te maken op papier.

Speciale Beschrijvings Technieken

Aangezien de meeste user-interfaces tegenwoordig grafisch zijn, is het gebruikelijk om UDP's te illustreren met een screenshot van een geval waarin het patroon is toegepast.

Voorbeeld

Sortable Table[Tid]
Dit patroon beschrijft een tabel waar de gebruiker door op de kolomkop te klikken de rijen kan sorteren op de betreffende kolom. Een extra keer klikken zorgt ervoor dat de volgorde omgekeerd wordt.



Figuur 2.4: Sortable Table

Het screenshot van MS Windows bestandsbeheer (zie *Figuur 2.4*) laat zien hoe dit patroon er binnen een applicatie uit zou kunnen zien. Het driehoekje geeft hierin aan op welke kolom er is gesorteerd en hoe: aflopend of oplopend.

Gebruik

Kijkend naar de user-interfaces van applicaties en websites zou je kunnen zeggen dat bijna iedereen gebruik maakt van UDP's. Bijvoorbeeld de meeste Microsoft producten delen dezelfde 'look-and-feel', veel websites gebruiken een menu aan de linkerkant enzovoort. Maar de vraag is uiteraard of zij bewust patronen gebruiken of dat de ontwikkelaar simpelweg doet wat hem makkelijkst lijkt.

Figuur 2.5: Onoverzichtelijke layout

Het is over het algemeen makkelijker om UI's te herkennen waarin patronen duidelijk niet op de juiste manier zijn gebruikt (zie *Sectie Anti-pattern*{*8*}). Zoals bijvoorbeeld in *Figuur 2.5* te zien is; deze website (`http://www.southparkx.net`) heeft zeer onoverzichtelijk lay-out waar het menu erg lastig te herkennen is.

Onderzoek

Onderzoek omtrent UDP's (en web design patterns) gebeurt vooral op VU (Vrije Universiteit) in Amsterdam en andere onderzoeksgroepen gespecialiseerd in gebruikersinteractie zoals Concordia University (Montreal, Canada).

Hoofdstuk 3

Functional Design Patterns

Waarom een nieuw patroontype?

Design patterns worden binnen de informatietechnologie met succes toegepast. Bij het ontwikkelen van systemen en het specificeren van frameworks zorgt dit gebruik voor grote tijdwinst en vooral 'betere' software. Het gebruik van patronen heeft een groot aantal voordelen, zoals:

- Leggen (domein/technische) kennis en ervaring vast.
- Maken oplossingen herbruikbaar.
- Introduceren abstractie binnen software ontwerp.
- Vergemakkelijken communicatie.
- Vergemakkelijken het ontwerpproces.
- Versnellen het ontwikkelproces.

Op dit moment worden voornamelijk de technische design patterns veel gebruikt; zoals die door de 'Gang of Four' [GHJV95] en de 'Gang of Five' [BMR+96, SSRB00] zijn opgesteld. Het grote nadeel van technische patronen is dat zij alleen inzicht verschaffen op technisch niveau. Domein/applicatie specifieke informatie kan over het algemeen niet verwerkt worden in deze patronen.
Een patrooncategorie waarin domeinkennis wél benut wordt zijn conceptuele patronen. Een conceptueel patroon is een patroon waarvan zijn vorm is beschreven in termen van concepten binnen het applicatiedomein [App00]. Conceptuele patronen, zoals analysis patterns (zie *Sectie Analysis Patterns*{*14*}, worden nog relatief weinig gebruikt. Dat analysis patterns weinig gebruikt worden heeft een aantal oorzaken, waaronder:

- *Nalaten analyse*
  Analysis patterns kunnen gebruikt worden bij het opzetten van een functioneel ontwerp. Het schrijven van een goed functioneel ontwerp en een grondige domeinanalyse is echter iets dat in veel projecten wordt nagelaten.
- *Onbekendheid*
  Ontwikkelaars hebben vaak geen weet van het bestaan van analysis patterns omdat er relatief weinig publicaties over zijn.

- *Beperkt aanbod*
  In vergelijking met technische patronen zijn er maar weinig analysis patterns beschikbaar.
- *Luiheid ontwerper/ontwikkelaar*
  Analysis patterns vertalen (concrete) concepten naar een (object)model. Meestal zal dit model niet direct toepasbaar zijn maar aangepast moeten worden aan een situatie. Het combineren van deze vertaling met andere patronen zou als 'te lastig' kunnen worden ervaren en daarom maar niet gebruikt.

De focus ligt bij analysis patterns op hoe concepten uit het domein gemodelleerd kunnen worden in programmatuur; niet op de interactie met en tussen deze concepten binnen een informatiesysteem, oftewel de functionaliteit.
Hoewel er patronen zijn op verschillende abstractieniveaus is er tussen deze patronen vaak niet direct een relatie zichtbaar. De relaties tussen patterns van verschillende patroontypes is onduidelijk; het is dan ook vaak lastig om patterns van verschillende types met elkaar te combineren.

Een nieuw patroontype

De bestaande patroontypes bieden geen mogelijkheid om terugkerende functionaliteit van applicaties binnen een bepaald domein te beschrijven. Dit terwijl dit toch een belangrijke rol zou kunnen spelen tijdens het ontwerp van een applicatie. Dit is de reden dat binnen dit onderzoek een nieuw, conceptueel, patroontype wordt gentroduceerd: *functional design pattern*.

> Functional design patterns zijn patronen die terugkerende functionele aspecten van (domeinspecifieke) applicaties beschrijven.

Een *functional design pattern*(FDP) heeft betrekking op de relaties tussen de verschillende concepten en de functionaliteit die een applicatie voor het desbetreffende domein moet bieden.
FDP's moeten ertoe bijdragen dat functionaliteit op een voorspelbare en herhaalbare manier gemplementeerd kan worden. Hiervoor geeft het de structuur aan waarin het probleem, een functionele eis, in een systeem opgelost kan worden. Daarbij wordt, als het mogelijk is, gebruikt gemaakt van andere patronen. Vooral wanneer er tegelijk door verschillende lagen in het systeem wordt ontwikkeld, zoals bij webapplicaties vaak gebeurt, is het belangrijk om het verband tussen het domein, functionaliteit, techniek en representatie vast te kunnen leggen. Op het moment dat dit verband bekend is zou het makkelijker moeten zijn om functionaliteit op een inzichtelijke en voorspelbare manier toe te voegen aan een systeem.
In de komende paragrafen zal er dieper worden ingegaan op wat functional design patterns precies beschrijven en hoe.

Eisen aan het patroon

Alvorens het *wat* en *hoe* van FDP's te beschrijven is het goed om te bekijken waar een goed pattern volgens patroonschrijvers aan zou moeten voldoen. Dit geeft gelijk inzicht over de eisen die aan het FDP-patroontype gesteld moeten

worden.
Volgens Coplien [App00] moet een goed pattern aan de volgende eisen voldoen:

- **Het lost een probleem op:**
  Patterns beschrijven oplossingen, niet alleen abstracte regels of strategieën
- **Het is een bewezen concept:**
  Patterns beschrijven toegepaste oplossingen, geen theorieën of speculaties.
- **De oplossing is niet triviaal:**
  Veel probleemoplossende technieken (zoals software-ontwerpparadigma's of methodes) proberen oplossingen af te leiden vanuit de beginselen. De beste patronen genereren een indirect oplossing voor een probleem.
- **Het beschrijft een relatie:**
  Patterns beschrijven niet alleen modules, maar diepere systeemstructuren en mechanismen.
- **Het pattern heeft een significante menselijke component:**
  Alle software dient het menselijk comfort of levenskwaliteit; de beste patronen beroepen zich expliciet op esthetica en bruikbaarheid.

Om een goed functional design pattern te kunnen formuleren is het noodzakelijk dat er (enigszins) aan deze eisen voldaan kan worden.
In hoeverre kunnen *goede* FDP's aan deze eisen voldoen?

- **Het lost een probleem op:**
  Functional design patterns beschrijven hoe een functionele eigenschap verwerkt kan worden binnen een systeem. Ze doen dit door de abstracte structuur te beschrijven waarin de functionaliteit terug komt binnen een applicatie.
  In plaats van het expliciet geven van de hele oplossing zal er worden verwezen naar patronen en mechanismen op een lager niveau. Hoewel deze oplossing minder concreet is dan bij de meeste technische patronen zal de lezer na het lezen van het patroon een duidelijk idee moeten hebben hoe hij de functionele eigenschap kan verwerken.
- **Het is een bewezen concept:**
  Het is de bedoeling dat FDP's worden vastgelegd aan de hand van verschillende (liefst drie of meer [1]) applicaties waarin een zelfde probleem wordt opgelost. Hiermee wordt voldaan aan de eis dat een patroon een daadwerkelijk toegepaste oplossing moet beschrijven.
  Het speculeren over FDP's kan nuttig zijn maar pas op het moment dat een (potentieel) patroon zich meerdere malen bewezen heeft in de praktijk kan bepaald worden of het ook daadwerkelijk goed is.
- **De oplossing is niet triviaal:**
  FDP's zijn vooral bedoeld om functionaliteit van applicaties binnen een bepaald domein te beschrijven. Om dit te kunnen doen zullen factoren binnen dit domein afgewogen moeten worden. Hoewel een oplossing uiteindelijk misschien triviaal mag lijken zal er in de meeste gevallen toch de benodigde domeinanalyse hebben plaatsgevonden om tot de conclusie te komen dat de gestelde eenvoudige oplossing correct is.

[1] De regel van drie: het eerste voorkomen is een voorval, het tweede voorkomen is toeval en het derde voorkomen zou een patroon kunnen zijn [Ant98].

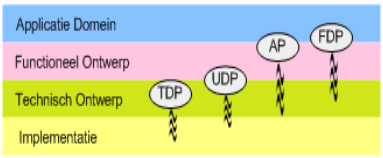
- **Het beschrijft een relatie:**
  FDP's beschrijven geen modules maar structuren binnen het domein. Ook in de oplossing wordt enkel de structuur beschreven waarin het probleem aangepakt kan worden. Hierbij zal er niet direct verwezen worden naar softwaremodules, maar naar diepere structuren (onderliggende patronen) en mechanismen.
- **Het pattern heeft een significante menselijke component:**
  De problemen die beschreven zullen worden met FDP's bevinden zich op functioneel/applicatie-niveau. De ervaring van de gebruiker speelt in deze een zeer belangrijke rol en zal dan ook nadrukkelijk naar voren komen. Het expliciet maken van relaties tussen FDP's en user-interface design patterns maakt het mogelijk om in te gaan op de manier waarop een functionaliteit zich presenteert naar de gebruiker.

Het moet dus mogelijk zijn om, volgens deze richtlijnen, een goed functional design pattern te beschrijven.

Toegevoegde waarde

Naast de expliciet genoemde patroontypes worden er binnen de literatuur nog veel meer verschillende patroontypes onderscheiden. Het blijkt echter in de praktijk dat deze types meestal maar weinig toegevoegde waarde hebben of als subtype onder één van de genoemde types vallen.
Functional design patterns hebben wel degelijk een toegevoegde waarde op de bestaande patronen:

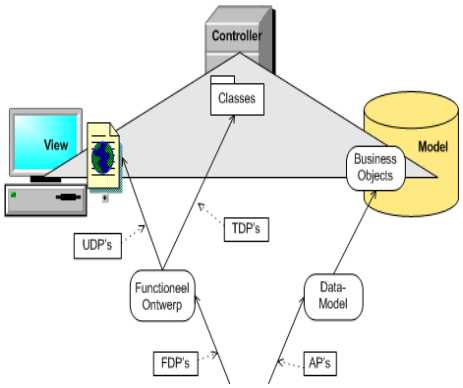
- Het FDP-type maakt het mogelijk om patronen binnen het applicatiedomein te beschrijven, en op een hoger niveau dan bij analysis patterns.
  In *Figuur 3.1* is te zien vanuit welk niveau de verschillende patronen (Technical Design Pattern, User-interface Design Patterns, Analysis Patterns en Functional Design Patterns) over het algemeen werken.



Figuur 3.1: Patronen op verschillende abstractielagen

- De (domein)kennis die vastgelegd is door FDP's kan gebruikt worden tijdens de analyse, het opstellen van de requirements en het maken van het functioneel ontwerp. Daarmee versnellen zij het (functionele) ontwerpproces.
- In de structuur van de oplossing wordt niet verwezen naar objecten of modulen maar naar oplossingen op hoger niveau zoals sub-FDP's of technische patronen. Dit maakt het makkelijker om patterns te combineren tot één applicatie.
  Hoe de verschillende patroontypes gebruikt kunnen worden is schematisch

weergegeven in het onderstaande diagram; FDP's geven aan wat er op representatie en aansturings-niveau nodig is om een functionele eigenschap in een applicatie te verwerken. Voor details over de implementatie hiervan kan gebruik worden gemaakt van de genoemde UDP's (voor representatie) en TDP's (voor aansturing). Analysis Patterns kunnen worden gebruikt tijdens het modeleren van de datarepresentatie van de domeinconcepten.



Figuur 3.2: Relaties tussen verschillende patroontypes

Functional Design Patterns

Functional design patterns beschrijven terugkerende (functionele) eigenschappen van applicaties. Het belangrijkste doel van FDP's is om domeinkennis vast te leggen zodat ontwikkelaars deze kunnen hergebruiken op het moment dat zij een systeem moeten bouwen wat dezelfde eigenschappen bevat.
Functional design patterns geven geen hapklare oplossing voor het probleem zoals veel technische patronen dat doen. Ze geven enkel de structuur aan waarbinnen een functionele eigenschap, binnen het applicatiedomein, opgelost kan worden.
Het zal bij functional design patterns soms gaan om patronen die sterk gerelateerd zijn aan de representatie voor de gebruiker. Het kan echter ook om patronen zijn die meer verborgen zitten in de structuur van het domein en uiteindelijk de applicatie.
Echter, uiteindelijk zullen de effecten van een functional design pattern altijd zichtbaar zijn voor de gebruiker.

Verschillende niveaus

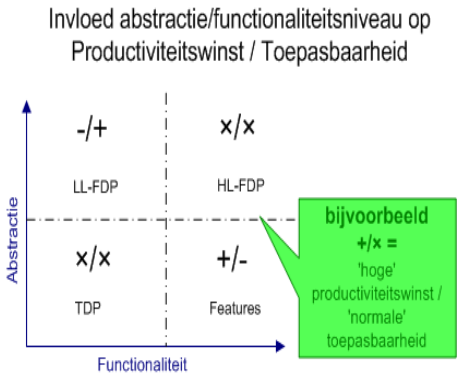
Functional design patterns zijn er op verschillende niveaus: sommige patronen beschrijven vrij algemene, eenvoudige basisfuncties, terwijl andere complexere, meer domeinspecifieke, functionaliteiten beschrijven. Men zou deze patronen op kunnen delen in respectievelijk low-level en high-level patterns, waarbij een high-level pattern zijn taken naar meerdere low-level patterns delegeert. Een harde scheiding hiertussen is lastig te definiëren; wat binnen het ene domein als high-

level gezien wordt kan binnen een algemener domein juist als basisfunctionaliteit gezien worden.
Het functionaliteitsniveau van een patroon is samen met het abstractieniveau van de oplossing bepalend voor:

- de herbruikbaarheid: in hoeveel gevallen is het patroon toepasbaar
- de productiviteitswinst: hoeveel werk wordt er uitgespaard door toepassen van het patroon (ontwerp, ontwikkeling én onderhoud)

In het algemeen geldt (zie ook *Figuur 3.3*):

- *hoog functionaliteitniveau en laag abstractieniveau -¿ grote productiviteitswinst maar lage herbruikbaarheid.*
  Patronen met een hoog functionaliteitniveau en laag abstractieniveau beschrijven specifieke functionele eigenschappen op een concrete manier. Als het patroon gebruikt kan worden, hetgeen alleen in specifieke situaties het geval is, dan levert het veel tijdwinst op aangezien de oplossing zeer gedetailleerd beschreven is; eventueel met direct bruikbare code of een template voor code.
- *laag functionaliteitniveau en hoog abstractieniveau -¿ lage productiviteitswinst maar hoge herbruikhaarbeid*
  Patronen met een laag functionaliteitniveau en hoog abstractieniveau beschrijven algemene patronen op een abstracte manier. Het patroon zal in veel gevallen te gebruiken zijn maar aangezien de oplossing vrij oppervlakkig is zal de ontwikkelaar nog veel zelf moeten doen en zal de productiviteitswinst relatief laag zijn.



Figuur 3.3: Invloed abstractie/functionaliteitsniveau

Binnen één applicatiedomein is het wel mogelijk om onderscheid te maken tussen lower-level en higher-level patterns.
Bijvoorbeeld bij administratieve applicaties:

- *Lower-level patterns* betreffen veelal eenvoudige basisfunctionaliteit die voor alle administratieve toepassingen hetzelfde is.
- *Higher-level patterns* betreffen specifieke functionaliteit die voor een bepaald probleemgebied van toepassing is.

Omschrijving FDP

fdp-omschrijving Zoals al eerder naar voren is gekomen (zie *Sectie Pattern omschrijving*{*8*}) is het verstandig om een standaard formaat te definiëren waarmee FDP's omschreven kunnen worden. Het maken van een expliciete scheiding tussen probleem, context, overwegingen en oplossing zal bij veel functional design patterns vrij lastig zijn. Het is dan ook niet verstandig om een patroonvorm te kiezen waarin deze secties expliciet terugkomen. Een sjabloon waarbij het patroon op een iets minder formele manier beschreven kan worden, zoals de GoF-form, sluit beter aan op de ideeën achter FDP's. De volgende alinea's geven aan wat in de verschillende secties van de omschrijving van een functional design pattern beschreven kan worden.

Naam & Classificatie

De *naam* van een patroon is zeer belangrijk. Deze zal gebruikt gaan worden om naar het patroon te refereren. Het is dan ook belangrijk dat de naam van het patroon zijn essentie weergeeft. Een goede naam van een functioneel patroon zal al een indicatie moeten wat het gaat doen en waarmee. De classificatie geeft aan in welke categorie het patroon past, dit zou gebruikt kunnen worden om patronen te categoriseren.

Domein

Het *domein* geeft het applicatiedomein aan waarvoor het patroon ontworpen is. Het patroon zou ook in andere domeinen nuttig kunnen zijn, maar het geeft de lezer een indicatie over de domeinkennis die verwerkt zal zitten in de patroonomschrijving.
Het aangeven van het domein kan ook nuttig zijn als de lezer zich vóór of tijdens de analysefase wil verdiepen in problemen binnen een specifiek applicatiedomein.

Intentie

De *intentie* van het patroon geeft aan wat het patroon probeert te bereiken; welke (functionele) eigenschap uit het applicatiedomein wordt beschreven door het patroon. Dit is te vergelijken met het probleem dat het patroon probeert op te lossen.
De intentie geeft in feite een korte samenvatting van het patroon.

Motivatie

De *motivatie* schetst een situatie waarin het patroon toegepast wordt en laat zien dat de functionele eigenschap op de beschreven manier op een juiste manier in het systeem verwerkt kan worden. Aan de hand van deze omschrijving zal de lezer al een idee moeten krijgen hoe de structuur van het patroon in elkaar zit.

Toepasbaarheid

De *toepasbaarheid* geeft aan, aan welke voorwaarden voldaan moet worden om de voorgestelde structuur te kunnen gebruiken. Het geeft aan in welke situaties het patroon toegepast kan worden en in welke situaties het beter is dit niet te doen; indien mogelijk kan er een naar een alternatief patroon verwezen worden.

Structuur

De *structuur* beschrijft hoe de functionele eigenschap verwerkt kan worden in de applicatie. Als er hierbij gebruik gemaakt kan worden van technische en/of user-interface patronen zal hiernaar verwezen worden en uitgelegd worden hoe zij gebruikt kunnen worden in combinatie met de andere patronen. Deze relaties kunnen grafisch worden verduidelijkt met behulp van een structuurdiagram. Daarnaast kan binnen de structuur-sectie worden beschreven hoe applicatiespecifieke kenmerken verwerkt zouden kunnen worden in de uiteindelijke oplossing. Vaak is dit echter niet nodig aangezien low-level details verborgen zijn in de onderliggende patronen.

Consequenties

De *consequenties*-sectie geeft een korte conclusie aan de hand van het gebruik van het patroon. Het beschrijft hoe het patroon voldoet aan de eisen en bespreekt de voor- en nadelen van het gebruik van het patroon.

Gebruik

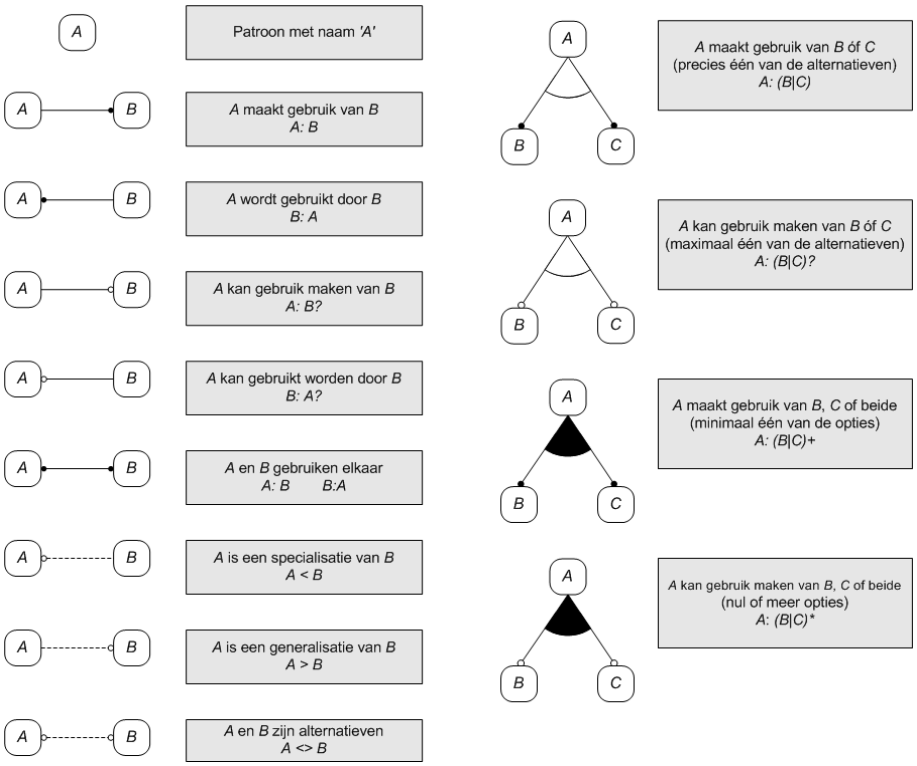
De sectie *gebruik* geeft voorbeelden aan van programma's waarin het patroon gebruikt wordt en kan daarnaast een voorbeeld geven van hoe het patroon er in de praktijk uit kan zien voor de gebruiker. Deze visuele representatie zal vaak al beschreven worden door user-interface design patterns. Toch kan het voor de gebruiker zeer verhelderend zijn om, al binnen het functionele patroon, te laten zien hoe de functionaliteit in het systeem verwerkt kan worden.

Gerelateerde patronen

In *gerelateerde patronen* kunnen patterns genoemd worden die op de één of andere manier gerelateerd zijn aan het beschreven patroon. Het kan hierbij gaan om specialisaties, generalisaties of alternatieven van het patroon of patronen die gebruikt worden binnen de structuur van de oplossing. Aangezien deze laatste patronen ook al ter sprake zijn gekomen bij het bespreken van de structuur van de oplossing is het noemen hiervan eigenlijk dubbel. Het kan echter voor de duidelijkheid goed zijn om deze toch te noemen hier.

Grafische weergave verbanden patronen

Het grafisch weergeven van verbanden tussen patronen helpt dat lezers een conceptueel model te vormen van het patroon: de lezer zal sneller inzicht krijgen in de relaties tussen de gebruikte patronen en de werking ervan. Er zijn een aantal verschillende soorten relaties die er tussen patronen kunnen voorkomen.
De in *Figuur 3.4* getoonde notatie kan gebruikt worden om alle relaties tussen patronen binnen een applicatiedomein aan te geven met behulp. Deze notatie is gebaseerd op de notatie zoals deze ook in featuremodellen gebruikt wordt (zie *Sectie Feature Oriented Programming*{*93*}). Het voordeel van deze notatie is dat er met een relatief eenvoudige syntax een duidelijke structuur aangegeven kan worden waarbij alternatieven expliciet naar voren komen; in een modelleertaal als uml is dit een stuk lastiger. Daarbij komt, dat het gebruik van uml de



Figuur 3.4: Notatie verbanden tussen patronen

indruk zou wekken dat er wordt gesproken over het ontwerp van software en de verbanden tussen klassen/componenten; hetgeen niet het geval is.
Bij het beschrijven van de structuur van functional design patterns is het belangrijk om een juist detailniveau te kiezen: Op het moment dat alle verbanden tussen de verschillende functionele, technische en user-interface patronen expliciet zouden worden gemaakt zou het geheel zeer onoverzichtelijk worden. De essentie van het functionele patroon zou verloren gaan in de wirwar van relaties; bijvoorbeeld tussen technische en user-interface-patronen. Het is daarom verstandig om bij het beschrijven van de relaties van een functioneel patroon vooral te focussen op de omliggende functionele patronen en relaties tussen het patroon zelf en de technische en user-interface patronen.

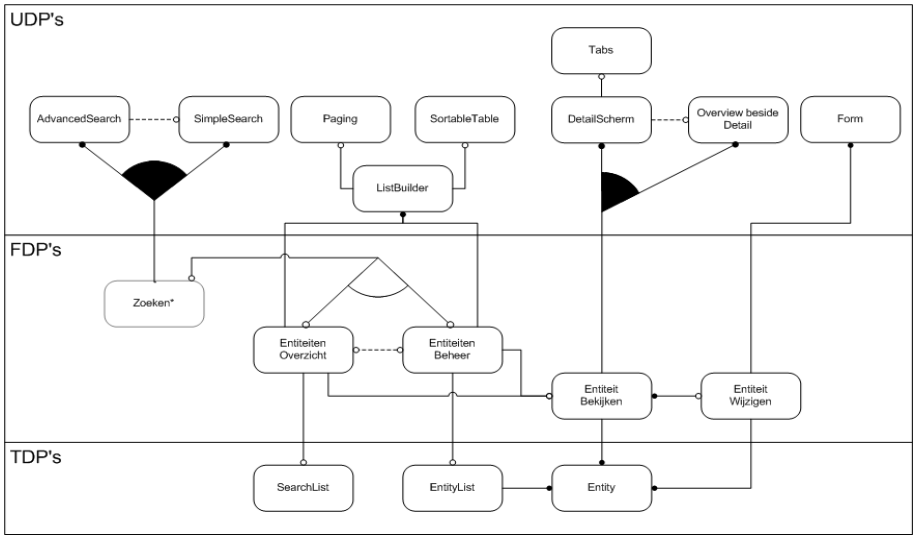
Afleiden functional design patterns

Er zijn verschillende manieren om functionele patronen af te leiden.
Belangrijk is om in ieder geval uit kunnen gaan van een aantal (succesvolle) applicaties waarin het patroon voorkomt. Het heeft geen zin om een patroon te schrijven voor een functionaliteit die specifiek is voor één applicatie. Op de eerste plaats beschrijft dit niets terugkerends en is het feitelijk geen patroon. Daarnaast zou het ook niet nuttig zijn aangezien er geen situatie is waarin het patroon opnieuw gebruikt kan worden.
Een goede methode om functional design patterns af te leiden is door middel

van de *functionele decomposities* van een aantal soortgelijke systemen. Een functionele decompositie verdeeld het systeem in functionele blokken; blokken die in verschillende systemen terugkomen kunnen als uitgangspunt dienen voor functionele patronen. Bij deze methode worden patronen opgesteld op basis van terugkerende functionele aspecten in een aantal bestaande systemen. Deze patronen kunnen vervolgens in toekomstige systemen gebruikt worden wanneer de zelfde functionaliteit geboden moet worden.
In de praktijk zullen patronen vaak juist opgesteld worden naar aanleiding van een nieuw te ontwikkelen systeem waarvan de functionaliteit sterk overeenkomt met bestaande systemen. Er zal gekeken worden welke functionele aspecten van de nieuwe applicatie ook in bestaande applicaties voorkomen en hiervoor zullen patronen geschreven worden. Deze patronen kunnen vervolgens direct getest worden bij de ontwikkeling van het nieuwe systeem.

Voorbeeldpatronen

In de komende paragrafen zullen enkele voorbeelden gegeven worden van (lower-level) functionele patronen in administratieve toepassingen. Dit zijn patronen die binnen *Quinity B.V.* gebruikt worden
Bij deze patronen draait het om het weergeven van *entiteiten* binnen een systeem. Een *entiteit* is hierbij een wezenlijk 'iets' dat opgeslagen wordt binnen het systeem; meestal in een tabel in een database.
Aangezien het hier gaat om voorbeeldpatronen en niet om een complete pattern-language voor administratieve toepassingen wordt er op enkele plaatsen gerefereerd naar (nog) niet-bestaande patronen, welke gemarkeerd zijn met een *.
Door de afkortingen F, U en T voor de patroonnaam zal worden aangegeven of het hierbij gaat om respectievelijk een functional, user-interface of technical design pattern. Deze patronen dienen enkel ter verduidelijking van de structuur en om aan te geven op welk terreinen er extra mogelijkheden liggen voor (functionele) patronen.
In *Figuur 3.5* is een globaal overzicht te zien van de relaties tussen de verschillende voorbeeldpatronen; vanuit functioneel oogpunt[2].

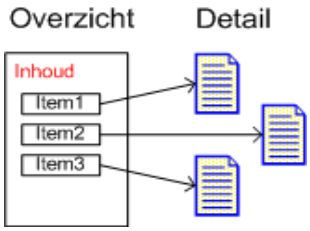


Figuur 3.5: Verbanden tussen voorbeeldpatronen

[2]Het Overzicht en Detail-patroon is in bovenstaand diagram weggelaten omdat dit patroon, vanwege zijn abstractie, het geheel minder overzichtelijk zou maken.

Overzicht en Detail

Domein: Alle applicaties met (grafische) user-interface
Intentie
Informatie/functionaliteit op een gestructureerde manier bereikbaar maken
Motivatie
Je wilt veel informatie beschikbaar stellen voor de gebruiker maar om dit allemaal vanuit één scherm te doen is zeer onoverzichtelijk en vaak zelfs onmogelijk. Anders dan bij een boek, is een computersysteem meestal niet lineair. Het geven van een overzicht waarvanuit de gebruiker naar het voor hem relevante gedeelte van het systeem kan gaan, zonder dat hij met de details van irrelevante secties geconfronteerd wordt, vergroot het gebruiksgemak.
Toepasbaarheid
Het is binnen systemen met veel functionaliteit of veel informatie die gepresenteerd wordt aan de gebruiker bijna altijd verstandig om (sub)systemen hiërarchisch te structureren; behalve als er sprake is van grote interne samenhang of als er sprake is van een lineair systeem (bijvoorbeeld een WIZARD[vW]).
Structuur
Een systeem of een gedeelte van een systeem wordt opgedeeld in verschillende (onafhankelijke) gedeeltes. Vanuit een overzicht kan er, door middel van referenties in de vorm van namen of korte beschrijvingen, naar de verschillende detailgedeeltes gegaan worden.



Figuur 3.6: Overzicht en detail patroon

Vanuit het detailgedeelte moet het mogelijk zijn om weer terug te gaan naar het overzicht.
Als het (lineaire) overzicht gelijksoortige detailgedeeltes bevat, bijvoorbeeld bij een foto-browser, is het verstandig om bladerfuncties, volgende en vorige, te bieden. Zo kan de gebruiker op een eenvoudige manier door de detailgedeeltes browsen.
Het is mogelijk dat ook de detailgedeeltes zelf ook weer overzichten bevatten waardoor een boomstructuur ontstaat.
Consequenties
Door het toepassen van het patroon ontstaat er een expliciete hiërarchische structuur, welke het systeem over het algemeen overzichtelijker maakt. Echter als de indeling, vanuit het oogpunt van de gebruikers, niet logisch is zal het navigeren door het systeem alleen maar lastiger worden. Soms is het niet mogelijk om het systeem volgens een strikte boomstructuur in te delen. Verwijzingen tussen deelsystemen zijn dan onvermijdelijk.
Een zoekfunctie (F.ZOEKEN*) kan de gebruiker ondersteuning geven in het vinden van het voor hem relevante gedeelte van de applicatie.

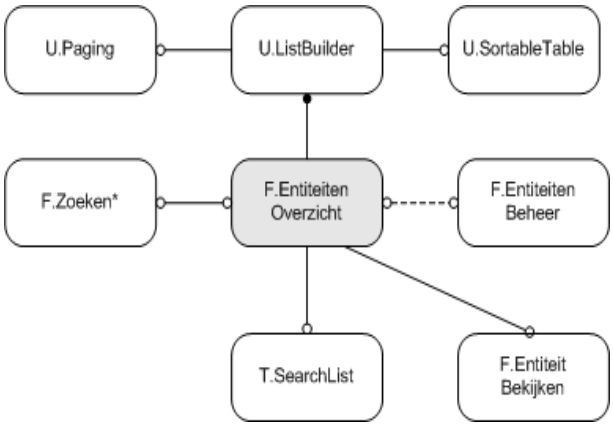
Gebruik
Het gebruik van overzichten waarin verwezen wordt naar detailgedeeltes komt op zeer veel plaatsen terug:

- Menu's
- Lijsten
- Navigatie door webpagina's

Gerelateerde Patronen
In plaats van, of ter verduidelijking van, een boomstructuur kan ook een zoekfunctie (F.ZOEKEN*) gebruikt worden.

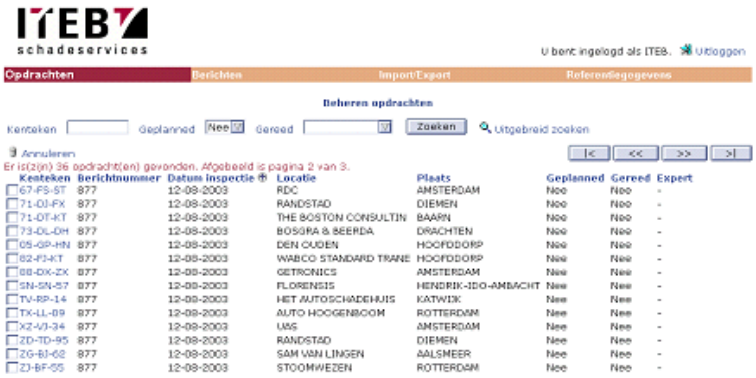
EntiteitenOverzicht

Domein: (administratieve) applicaties
Intentie
Het geven van een overzicht van entiteiten in lijsten binnen een systeem.
Motivatie
De gebruiker wil kunnen zien welke entiteiten, bijvoorbeeld producten, zich in een systeem bevinden en deze entiteiten doorzoeken. Het zou hierbij zeer onhandig zijn als de gebruiker enkel de mogelijkheid heeft om gegevens per entiteit te kunnen bekijken.
Toepasbaarheid
EntiteitenOverzicht kan toegepast worden als de gebruiker de mogelijkheid moet hebben om door *gelijksoortige* entiteiten in de database te 'browsen'. Het patroon is alleen nuttig als hij niet meerdere entiteiten in de tabel tegelijk hoeft te wijzigen, in dit geval is het EntiteitenBeheer-patroon beter.
Structuur
De structuur waarin de eigenschap om entiteiten te beheren terug kan op de volgende manier terugkomen in een applicatie.
De lijst, of een gedeelte daarvan welke bijvoorbeeld door middel van de aanwezige zoekfunctie (F.Zoeken*) verkregen is, kan met behulp van een lijst (Listbuilder[vW]) gerepresenteerd worden aan de gebruiker.
Het is belangrijk om de juiste attributen te kiezen om te laten zien in de lijst; de gebruiker wil voldoende informatie hebben over de entiteit, maar niet te veel anders zal hij het overzicht kwijt raken. Bij het kiezen van de te tonenen velden moet worden uitgegaan van het doel van de gebruikers.
Bijvoorbeeld in het geval van een adresboek voor telefonistes is het verstandig om in ieder geval naam en telefoonnummer te tonen in het overzicht; emailadres, bankrekeningnummer, pasfoto's en dergelijke zijn in eerste instantie minder relevant.
Lange lijsten kunnen verdeeld worden over meerdere schermen voor een beter overzicht (Paging[vW]).
Eventueel kan het overzicht de gebruiker de extra mogelijkheid geven om de lijst te sorteren door middel van het klikken op kolomkoppen (Sortable Table[Tid]).
Door het EntiteitenOverzicht-patroon te combineren met het subpatroon EntiteitBekijken is het mogelijk om vanuit het overzicht (bijvoorbeeld door middel van het klikken op item) naar een individuele entiteit te gaan en de details hiervan te bekijken.
In de gegeven voorbeelden is het mogelijk om, bijvoorbeeld door middel van checkboxen, meerdere items te selecteren om hier vervolgens in één keer iets mee te doen, zoals kopiëren of verwijderen.
Daarnaast kan er de mogelijkheid worden geboden om vanuit het overzicht naar een invoerscherm te gaan om een nieuw item in te voeren.
In de applicatie kan de lijst met entiteiten geïmplementeerd worden als T.SearchList*.
Dit patroon zorgt ervoor dat een lijst op een efficiënte manier uitgelezen kan worden uit de database en kan worden gerepresenteerd in het geheugen.
De relaties van het EntiteitenOverzicht-patroon met andere patronen is te zien in *Figuur 3.7*.
Consequenties
Het EntiteitenOverzicht-patroon zorgt er voor dat een gebruiker op een



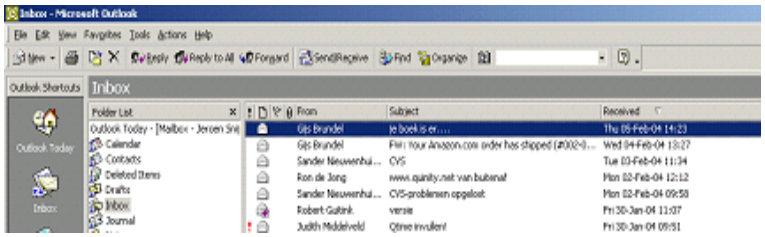
Figuur 3.7: Relaties met andere patronen

overzichtelijke manier kan zien welke gegevens in de database zitten. Vanuit de overzichtspagina kan de gebruiker op een eenvoudige manier naar zoek en onderhoudsschermen. Als de gebruiker geen overzicht heeft over de database zal het voor hem onduidelijk zijn welke entiteiten zich in de database bevinden.
Gebruik
Het patroon wordt gebruikt in veel administratieve systemen voor het beheren van gegevens in een database (zie *Figuur 3.8*) of bijvoorbeeld in een email-programma om een overzicht te tonen van ontvangen berichten (zie *Figuur 3.9*).



Figuur 3.8: Opdrachtenoverzicht ITEB

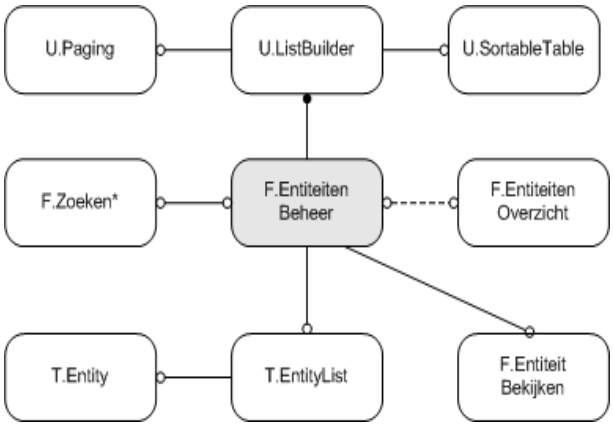
Gerelateerde Patronen
Het EntiteitenOverzicht-patroon vormt samen met EntiteitBekijken een instantiëring van Overzicht en Detail. Het maakt gebruik van verschillende andere patronen:
FDP's: EntiteitBekijken
TDP's: SearchList*
UDP's: ListBuilder[vW], Sortable Table[Tid], Paging[vW]
Alternatief: EntiteitenBeheer.



Figuur 3.9: Berichtenoverzicht MS Outlook

EntiteitenBeheer

Domein: (administratieve) applicaties waarbij gegevens beheerd moeten worden
Intentie
Het bekijken en tegelijk wijzigen van entiteiten in lijsten in een systeem.
Motivatie
In sommige situaties wil de gebruiker vanuit een lijst meerdere records tegelijk wijzigen; bijvoorbeeld bij het invoeren van cijfers voor een tentamen. Net als bij het EntiteitenOverzicht wil de gebruiker hierbij een overzicht hebben van de data in de database. *Toepasbaarheid*
Het EntiteitenBeheer-patroon kan worden gebruikt bij (korte) lijsten van records, van hetzelfde type.
Structuur
De structuur waarin de eigenschap om entiteiten te beheren terug kan komen in de applicatie is de volgende.
De te wijzigen lijst kan met behulp van een ListBuilder[vW] gepresenteerd worden aan de gebruiker. Het gebruik van verschillende pagina's (Paging[vW]) is meestal niet verstandig aangezien de gebruiker zo het overzicht kan verliezen over de gewijzigde records.
In het overzicht zullen mogelijkheden moeten zijn voor de gebruiker om relevante gegevens te wijzigen, bijvoorbeeld door invoervelden of comboboxen.
Door middel van het subpatroon EntiteitWijzigen zou de mogelijkheid kunnen worden geboden om vanuit het overzicht naar een individuele entiteit te gaan (door middel van het klikken op item) en de overige attributen van deze entiteit te wijzigen.
Binnen het systeem kan de lijst met entiteiten geïmplementeerd worden met behulp van het T.EntityList*-patroon dat er voor zorgt dat er meerdere entiteiten van hetzelfde type tegelijk ingeladen n gewijzigd kunnen worden.
De relaties van het EntiteitenBeheer-patroon met andere patronen is te zien in *Figuur 3.10*.
Consequenties
Het EntiteitenBeheer-patroon zorgt er voor dat een gebruiker vanuit één scherm meerdere entiteiten tegelijk kan wijzigen. Dit is vooral nuttig als er relatief weinig veranderd hoeft te worden bij veel entiteiten of er een relatie bestaat tussen de verschillende wijzigingen.
Het toepassen van het patroon vergemakkelijkt de invoer maar is technisch gezien ingewikkelder en vereist meer rekenwerk dan het één voor één wijzigen van records aangezien er nu niet één maar meerdere records tegelijk gewijzigd kunnen worden. Vanwege de extra overhead is het verstandig om dit patroon

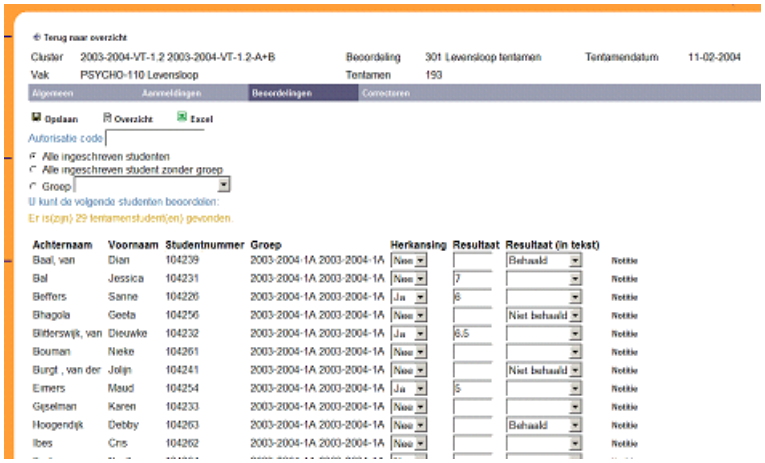


Figuur 3.10: Relaties met andere patronen

alleen toe te passen als het belangrijk is om meerdere entiteiten tegelijkertijd te wijzigen. Als dit niet nodig is kan er voor het EntiteitenOverzicht-patroon gekozen worden.

Gebruik

Het patroon wordt veel gebruikt bij het invullen/wijzigen van lijsten waarbij er een duidelijk verband bestaat tussen de verschillende regels; bijvoorbeeld de regels van een order of de cijfers voor één tentamen van verschillende leerlingen(zie *Figuur 3.11*).



Figuur 3.11: Cijferinvoer MMC

Gerelateerde Patronen

Net als het alternatief, EntiteitenOverzicht, is het EntiteitenBeheer-patroon samen met EntiteitBekijken een voorbeeld van het Overzicht en Detail-patroon.

EntiteitenBeheer maakt gebruik van verschillende andere patronen:

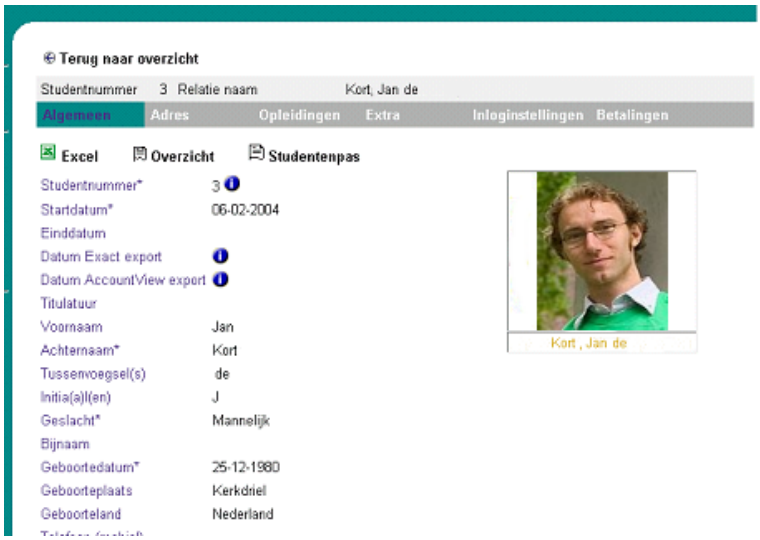
FDP's: EntiteitBekijken

TDP's: T.SearchList*

UDP's: ListBuilder[vW], Sortable Table[Tid], Paging[vW]

EntiteitBekijken

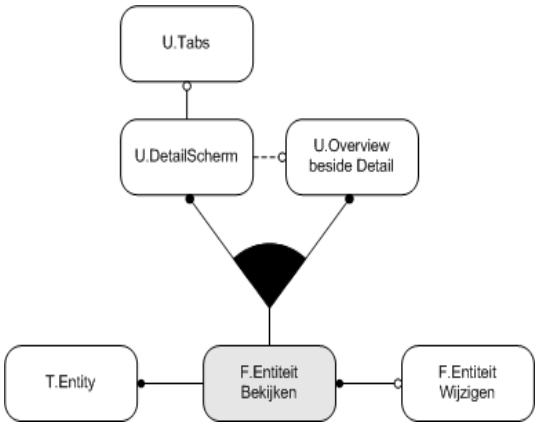
Domein: (administratieve) applicaties
Intentie
Details van een entiteit binnen een systeem bekijken.
Motivatie
Gebruikers willen detailgegevens van een entiteit in het systeem kunnen zien. Bijvoorbeeld de naam en adresgegevens van een klant in een klantenadministratie.
In een overzicht is het meestal wel mogelijk om de belangrijkste gegevens van een entiteit te zien maar het is meestal onhandig om alle gegevens te tonen. Het is daarom verstandig om gebruikers daarnaast de mogelijkheid te geven om in n keer alle gegevens van een entiteit te benaderen. *Voorbeeld*



Figuur 3.12: Bekijken studentgegevens, MMCStudenten-administratie

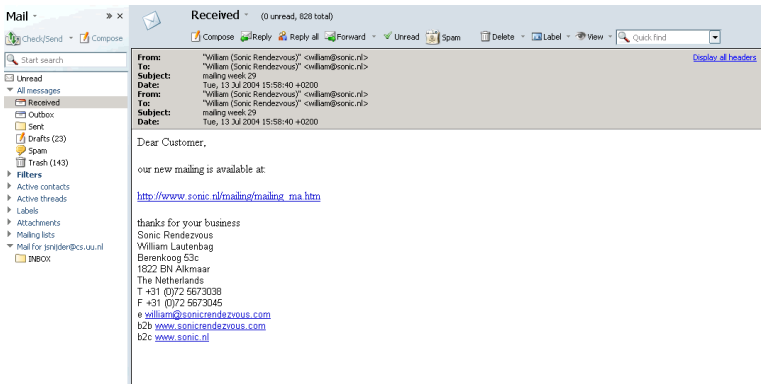
Toepasbaarheid
Het EntiteitBekijken-patroon kan gebruikt worden in applicaties waarbij de gebruiker de mogelijkheid moet/mag hebben om details van entiteiten te bekijken.
Structuur
De eigenschap om details van entiteiten te bekijken kan op verschillende manieren binnen een systeem verwerkt worden.Een mogelijkheid is om vanuit een lijstoverzicht (EntiteitenOverzicht, EntiteitenBeheer) in een nieuw scherm (Detailscherm), of een gereserveerd gedeelte van het scherm (Overview beside detail[Laa]), alle details van een entiteit te tonen. Om eenvoudig door de details van entiteiten in deze lijst te kunnen browsen kunnen bladerfuncties geboden worden om terug naar het overzicht en de volgende en vorige entiteit te gaan.
Het direct bekijken van detailinformatie van een entiteit door middel van een code kan in sommige gevallen nuttig zijn, denk aan het benaderen van productinformatie op basis van een ingescande streepjescode.

Een entiteit kan attributieve entiteiten hebben die naar hem refereren. Als deze entiteiten belangrijke informatie bevatten is het essentieel dat deze details ook getoond worden aan de gebruiker. Om ervoor te zorgen dat dit overzichtelijk blijft kunnen tabbladen gebruikt worden (Tabs[vW]).De entiteit kan binnen het systeem door middel van het T.Entity*-patroon gerepresenteerd worden. Dit patroon bied ook de mogelijkheid om extra gegevens over het object, bijvoorbeeld de attributieve entiteiten, te verzamelen.
De relaties van het EntiteitBekijken-patroon met andere patronen is te zien in *Figuur 3.13*.



Figuur 3.13: Relaties met andere patronen

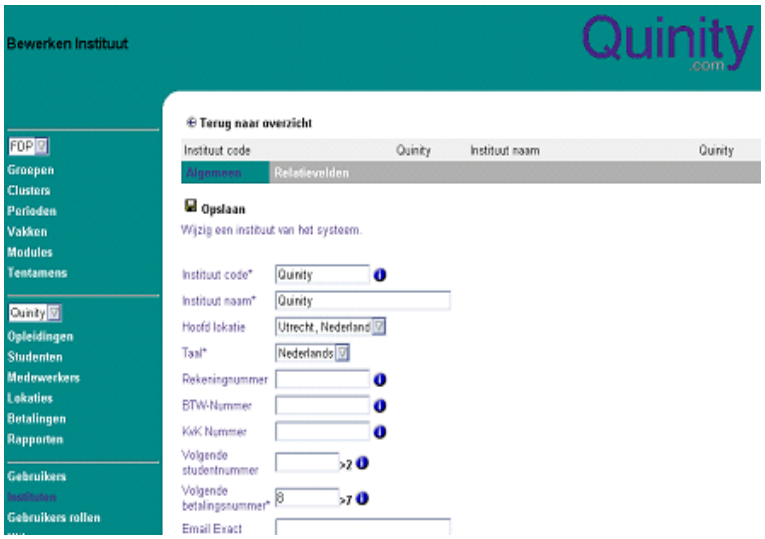
Consequenties
Een overzicht zal in veel gevallen niet alle details van een entiteit kunnen tonen. In een aparte detailweergave voor entiteiten kan dit wel zonder de applicatie minder overzichtelijk te maken. Het is verstandig om op het moment dat de gebruiker ook in staat is om nieuwe entiteiten toe te voegen via het systeem alle in te voeren details te tonen of duidelijk te maken welke, en waarom, gegevens verborgen zijn. Op deze manier kan er geen verwarring ontstaan over 'verborgen details' in detailschermen.
Gebruik
In vrijwel iedere (administratieve) applicatie moet het mogelijk zijn om de details van gegevens binnen de database te bekijken. Bijvoorbeeld in een studentenadminstratie waar de persoonsgegevens van studenten bekeken kunnen worden (*Figuur 3.12*) en emailapplicaties waarin alle data van een ontvangen bericht bekeken kan worden (*Figuur 3.14*).
Gerelateerde Patronen
Het EntiteitBekijken-patroon is een specialisatie van de detailweergave uit het Overzicht en Detail-patroon.
Gerelateerde patronen:
FDP: EntiteitWijzigen
TDP: T.Entity*
UDP: Detailscherm



Figuur 3.14: Bekijken van emailbericht Opera

EntiteitWijzigen

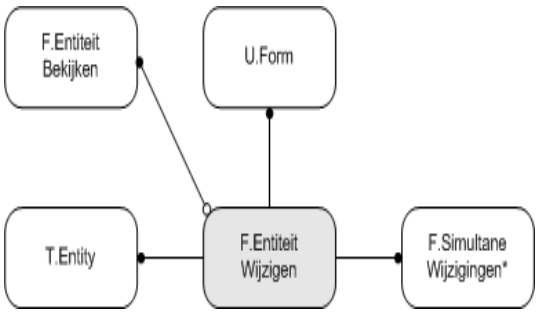
Domein: (administratieve) applicaties
Intentie
Details van een entiteit in het systeem wijzigen.
Voorbeeld



Figuur 3.15: Wijzigen instituutgegevens, MMCStudenten-administratie

Motivatie
Je wilt dat gebruikers (beheerders) de gegevens van een entiteit in de database kunnen wijzigen. Bijvoorbeeld de verkoopprijs of omschrijving van een product in een winkeladministratie.
Toepasbaarheid
Het EntiteitWijzigen-patroon kan gebruikt worden in applicaties waarbij de gebruiker de mogelijkheid moet/mag hebben om details van entiteiten te

wijzigen.
Structuur
Naast het bekijken van details zullen beheerders of geauthoriseerde gebruikers de mogelijkheid moeten hebben om gegevens van entiteiten te kunnen wijzigen. Hiervoor kan het 'EntiteitWijzigen'-patroon gebruikt worden. Het is verstandig om het wijzigen direct benaderbaar te maken vanuit het detailoverzicht zodat de gebruiker de getoonde gegevens direct kan wijzigen.
Het invoeren van nieuwe data kan door middel van een formulier (FORM[vW]). In enkele gevallen is het belangrijk dat de gebruiker de oude waarde van het veld niet kan zien; dit is bijvoorbeeld het geval bij het wijzigen van een wachtwoord. In alle overige gevallen zal de gebruiker wél willen kunnen zien wat de oude waarden zijn van de velden die hij wil wijzigen. Dit kan door het bekijken (ENTITEITBEKIJKEN) en wijzigen van details te combineren tot één scherm waarbij de oude en nieuwe waarden naast elkaar te zien zijn.
Een andere manier is om een scherm te tonen waarin de oude waarden ingevuld zijn in het formulier. Een nadeel hiervan is dat tijdens het wijzigen van een veld de oude waarden niet meer te zien zijn.
Over het algemeen zullen niet alle gebruikers de mogelijkheid moeten hebben om gegevens te kunnen wijzigen. Het is bijvoorbeeld niet verstandig om klanten de mogelijkheid te bieden de verkoopprijs van een product te wijzigen. Autorisatie, F.AUTORISATIE*, zal dan ook vereist zijn om ervoor te zorgen dat alleen gebruikers met de juiste rechten deze mogelijkheid hebben.
Op het moment dat gebruikers gegevens kunnen wijzigen in het systeem zullen er maatregelen genomen moeten worden tegen *concurrency*-problemen. Concurrency treed op als meerdere gebruikers tegelijkertijd gegevens van n entiteit willen wijzigen. Dit kan gedaan worden door gebruik te maken van het patroon F.SIMULTANE-WIJZIGINGEN*.
De relaties van het ENTITEITWIJZIGEN-patroon met andere patronen is te zien in *Figuur 3.16*.



Figuur 3.16: Relaties met andere patronen

Consequenties
De mogelijkheid bieden voor gebruikers om via het systeem gegevens te kunnen wijzigen is in veel applicaties cruciaal: gegevens veranderen en het is van groot belang dat de gegevens in het systeem de juiste waarde hebben. Het alternatief: het toevoegen van een nieuwe entiteit en eventueel het verwijderen van de oude op het moment dat waarden niet meer geldig zijn, is in veel gevallen geen optie; verwijzingen naar oude entiteiten zullen ongeldig worden en aangepast moeten

worden. Daarnaast is het vaak conceptueel niet juist zijn om een entiteit te verwijderen en een nieuwe hiervoor toe te voegen op het moment dat waarden veranderd zijn; in het echt wordt bijvoorbeeld een persoon ook niet 'verwijderd' en opnieuw gecreëerd op het moment dat hij verhuisd.

Gebruik

Het wijzigen van gegevens komt in vrijwel ieder administratief systeem voor. Het patroon kan dan ook in veel applicaties gebruikt worden.

Figuur 3.17: Het wijzigen van een mutatieaanvraag

Gerelateerde Patronen

Gerelateerde patronen:

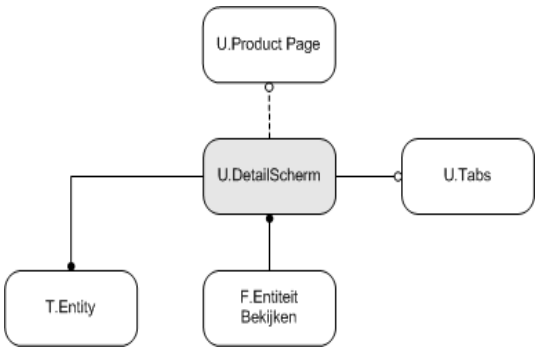
FDP: EntiteitBekijken, F.Autorisatie*, F.Simultane-Wijzigingen*

TDP: T.Entity*

UDP: Form[vW]

Detailscherm

Domein: (administratieve) applicaties
Intentie
Het tonen van de details van ëën record binnen het systeem aan de gebruiker.
Motivatie
De gebruiker wil op een simpele manier kunnen zien welke gegevens over bijvoorbeeld een product of persoon in het systeem bekend zijn.
Toepasbaarheid
Een DETAILSCHERM kan gebruikt worden in applicaties waarbij de gebruiker de mogelijkheid moet/mag hebben om details van entiteiten binnen het systeem te bekijken en het weergeven van meerdere entiteiten tegelijk (bijvoorbeeld in een overzicht) niet overzichtelijk zou zijn.
Structuur
Een detailscherm kan simpelweg alle gegevens die over een object, zijn velden en de waarden specifiek voor dit object, tonen aan de gebruiker.
Gebruikers zijn echter vaak maar geïnteresseerd in een aantal van deze velden. Het is de taak van de ontwerper om ervoor te zorgen dat de gebruiker de informatie die hij zoekt op een zo makkelijk mogelijke manier kan vinden. De volgorde en stijl waarop de gegevens gepresenteerd worden speelt hierin een belangrijke rol. Over het algemeen is het verstandig om belangrijke data opvallend, groot of vet, bovenaan (of juist helemaal onderaan) het scherm te plaatsen zodat de gebruiker deze snel ziet.
Soms is er zoveel informatie over een object beschikbaar binnen het systeem dat dit niet op een overzichtelijk manier op n scherm past. Het gebruik van een navigatietechniek als tabbladen (TABS[vW]) of een onderverdeling van details in categorieën kan dan uitkomst bieden.
De relaties van het DETAILSCHERM-patroon met andere patronen is te zien in *Figuur 3.18*.



Figuur 3.18: Relaties met andere patronen

Consequenties
Door alle gegevens over een gegeven in het systeem beschikbaar te maken vanuit één pagina 'verdwalen' gebruikers minder snel in het systeem terwijl zij wel bij alle gegevens over een entiteit kunnen.
Gebruik
Vrijwel ieder systeem kent detailschermen waarin specifieke gegevens over een object weergegeven kunnen worden.

Figuur 3.19: Tonen Productdetails (http://www.mp3spelers.nl/)

Gerelateerde Patronen
Het DETAILSCHERM is een generalisatie van het PRODUCT PAGE[vW]-patroon en kan gebruik maken van verschillende patronen voor 'navigatietechnieken' [vW].

Hoofdstuk 4

Toepassing patronen

Concrete Oplossing

Hoewel het beschrijven van een patroon en de feedback van gebruikers kan leiden tot betere inzichten is het beschrijven zelf niet het doel van functionale design patterns. Het doel van patronen is deze te gebruiken bij de ontwikkeling van software.
Functional design patterns beschrijven geen concrete oplossingen in de vorm van code(templates) die direct gebruikt zouden kunnen worden bij de implementatie. In plaats daarvan wordt op een abstracte manier aangegeven hoe een functionaliteit verwerkt zou kunnen worden in een systeem.
Een patroon is niet implementatiegebonden. Om een oplossing binnen een systeem te kunnen hergebruiken zal er dus vanuit het patroon een implementatie geconstrueerd moeten worden. Tussen verschillende systemen binnen een bedrijf zal deze oplossing echter vaak grote gelijkenissen vertonen. Het is dan ook verstandig om naast het patroon ook een implementatiegebonden oplossing te geven die beschrijft hoe het patroon in de praktijk wordt toegepast.
Deze *concrete oplossing* zou beschreven kunnen worden in de 'gebruik'-sectie van het patroon, maar deze sectie is eigenlijk meer bedoeld om voorbeelden te geven in plaats van een precies gebruik af te dwingen. Wanneer de *concrete oplossing* een oplossing beschrijft voor een heel algemeen probleem zou deze ook beschreven kunnen worden in een technisch patroon (zie *Sectie Technical Design Patterns*{*16*}).
Het los beschrijven van het patroon en de concrete oplossing heeft een aantal voordelen:

- Het patroon is niet afhankelijk van de implementatie en dus
  - ook buiten bedrijf te gebruiken, en
  - bij verandering technieken blijft het patroon geldig.
- Het patroon vormt documentatie en verantwoording bij implementatie.
- Een concrete oplossing verhoogt de productiviteitswinst.

Hoe de concrete oplossing gebruikt kan worden bij het ontwikkelen van een systeem hangt af van de vorm van deze oplossing:

indirecte oplossing:
: Een indirecte oplossing geeft een voorbeeld-implementatie die als referentie kan dienen bij het implementeren van het patroon.

directe oplossing:
: Een directe oplossing geeft een implementatie die direct (her)gebruikt kan worden om het patroon toe te passen.

Wanneer een *indirecte oplossing* gegeven wordt zal een programmeur die de functionaliteit wil implementeren zelf de oplossing aan moeten passen op zijn situatie. Er is niet direct sprake van hergebruik van code maar enkel van methode/techniek. Het is belangrijk dat een referentie-implementatie alle belangrijkste details van het patroon behandeld maar ook niet te veel. In een te gedetaileerde voorbeeld-implementatie zou het lastig zijn om het onderscheid tussen algemene structuren en details specifiek voor het voorbeeld niet.
Bij een *directe oplossing* kan de concrete oplossing zelf gebruikt worden voor de implementatie van een nieuw systeem. De oplossing geeft in dit geval een her te gebruiken *generieke oplossing*.
De voorkeur van een programmeur zal in de eerste instantie uitgaan naar een directe generieke oplossing aangezien hier door hergebruik van code veel tijd bespaard zou kunnen worden. Of een directe oplossing mogelijk is hangt vooral af van de afhankelijkheden van de implementatie:
Het kan zijn dat de manier waarop een oplossing in een systeem verwerkt wordt alleen afhangt van de gebruikte technieken, zoals programmeertaal, framework, operating system et cetera. De meeste van deze technieken zullen binnen een productlijn gelijk zijn. Een concrete implementatie voor een dergelijk patroon zal dan ook in veel gevallen direct gegeven kunnen worden.
Een oplossing kan naast de gebruikte technieken ook afhankelijk zijn van applicatie-specifieke kenmerken, zoals datamodellen, de data zelf, specifieke wensen van de opdrachtgever et cetera.
Een eenduidige concrete implementatie zal in dergelijke gevallen niet mogelijk zijn. Het is soms wel mogelijk om een directe oplossing te geven voor het generieke gedeelte van het probleem en 'gaten' open te houden voor applicatie-specieke eigenschappen. Hoe applicatiespecifieke details vervolgens in deze generieke oplossing gespecificeerd en verwerkt kunnen worden hangt af van de gekozen techniek. Verschillende technieken zullen besproken worden in *Sectie Generieke Technieken*{*46*}.
Het bieden van een generieke oplossing is alleen zinnig als dit een programmeur werk bespaard of de implementatie hierdoor inzichtelijker maakt. Wanneer het niet mogelijk is om een generieke oplossing op een inzichtelijke wijze te parameteriseren met applicatie-specifieke keuzes zal een referentie-implementatie geschikter zijn. Door methodisch aan te geven hoe een ontwikkelaar/ontwerper deze aan kan passen aan applicatie-specifieke kenmerken kan er toch voor gezorgd worden dat een functionaliteit volgens een vaste strategie geïmplementeerd wordt.

Generieke Technieken

Functional design patterns vormen een goede uitgangspositie voor het maken van een generieke oplossing voor een productlijn. Hoe deze generieke oplossing

eruit zal zien is voor een groot deel afhankelijk van de manier waarop de eigenschap uiteindelijk door het systeem verweven zal zitten; Autorisatie zal op een hele andere manier in het systeem terugkomen dan het tonen van een bepaalde entiteit.
Door deze grote verschillen is het lastig om één techniek aan te wijzen voor generieke oplossingen. Losse modules, eventueel gegenereerd met behulp van generatieve programmeertechnieken ([CE00]), zijn alleen geschikt voor eigenschappen die niet sterk verweven zijn in het systeem.
Voor eigenschappen die sterk verweven zijn met het systeem is het over het algemeen lastig om generieke oplossingen te geven. Er zijn een aantal programmeer methodieken die dit proberen te ondervangen, waaronder *Aspect Oriented Programming* ([Lee02]).
In de volgende secties zullen een aantal bekende en relatief nieuwe technieken besproken worden om kennis te hergebruiken in (object-georienteerde) code. Oftewel manieren om een generieke oplossing voor een probleem binnen een bepaald domein te (her)gebruiken.

Bestaande Technieken

Het kunnen hergebruiken van code is een probleem dat al heel lang bekend is binnen de informatica. In vrijwel alle programmeertalen is er directe ondersteuning voor het hergebruiken van code. Deze mechanismen kunnen gebruikt worden bij het bieden van een generieke oplossing voor een probleem.
De 'standaard' manier om code te hergebruiken is door het definiëren van libraries en deze te *importeren* binnen een nieuwe applicatie. Deze techniek is fundamenteel in vrijwel alle moderne programmeertalen.
Het idee is om een los library functies en objecten te definiëren welke na het importeren van het library direct in de nieuwe applicatie gebruikt kunnen worden. Met behulp van parameters in deze functies en objecten kunnen applicatiespecifieke waarden meegegeven worden. De functies en objecten zelf zullen zo ontworpen moeten zijn dat zij binnen zoveel mogelijk applicaties gebruikt kunnen worden;
Voor methodes en objecten met een duidelijke losstaande functionaliteit die in principe voor elke applicatie gelijk is is dit eenvoudig; voorbeelden hiervan zijn bijvoorbeeld functies als `cos` en `abs` in het Math-llibrary in Java:

```
import java.lang.Math;

class A{
  ...
    int x    = ... ;
    int cosx = Math.cos(x);
  ...
}
```

Voor generieke oplossingen waarbij de werking altijd gelijk is zijn losse functielibraries een goede oplossing.
Voor methodes en componenten waarbij de werking enigzins afhangt van de context (de applicatie) waarin zij gebruikt worden is dit een stuk lastiger. Alle eigenschappen die van invloed kunnen zijn op de werking zullen als parameter meegegeven moeten kunnen worden.

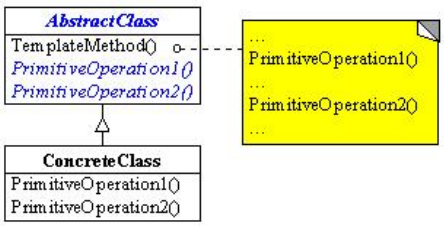
Het risico bestaat bij een generieke oplossing dat het doorgeven van, en het rekening houden met, context-eigenschappen onnodig veel overhead geeft, en voor een ondoorzichtig monolitisch systeem zorgt. Een oplossing voor dit probleem zou kunnen zijn om een opdeling te maken in kleinere functies/modules die vervolgens door ontwikkelaar zelf gecombineerd kunnen worden tot de door hem gewenste functionaliteit.
Soms is het niet voldoende om gebruik te maken van basisfuncties maar zal een ontwikkelaar invloed moeten kunnen hebben op de werking van het object of de component zelf. Een mechanisme wat hierbij gebruikt kan worden is *overerving*.

Overerving

Overerving maakt het mogelijk om alle eigenschappen van een object over te nemen in een subklasse en eventueel nieuwe eigenschappen toe te voegen of oude methodes te overschrijven. Deze subklasse kan vervolgens gebruikt worden op elke plek waar een object van de hoofdklasse gevraagd wordt.
Een generieke oplossing zou een klasse kunnen definiëren met een algemene implementatie die de basisfunctionaliteit voor alle toepassingen levert. Vervolgens kan deze klasse 'extend' worden waarbij applicatiespecifieke eigenschappen in een subklasse door een ontwikkelaar kunnen worden toegevoegd bijvoorbeeld door middel van een template methode.

Template method

Wanneer een generieke klasse niet gebruikt kan/mag worden zonder dat een ontwikkelaar zelf toepassing-specifieke code toevoegt kan een zogenaamde Template Method[GHJV95] gebruikt worden.
De template-methode definiëert een skelet van een algoritme waarin wordt verwezen naar substappen in subklassen. Het skelet wordt gevormd door een abstracte klasse met daarin de functies met verwijzigen naar specifieke operaties waarvan de implementatie in het skelet ontbreekt. Om de klasse te gebruiken zal er een subklasse aangemaakt moeten worden waarin de missende operaties gedefiniëerd worden (zie voorbeeldcode en structuur in *Code 4.1* en *Figuur 4.1*).



Figuur 4.1: Structuur template method

Listing 4.1: Template Method

```
abstract class AbstractClass{
  public void foo1(){
    ...
    primitiveOperation1();
    primitiveOperation2();
    ...
  }
  abstract primitiveOperation1();
```

```
  abstract primitiveOperation2();
}

public class ConcreteClass{
  public primitiveOperation1(){
    ...
  }
  public primitiveOperation2(){
    ...
  }
}
```

Het gebruik van abstracte template-klassen is zeer geschikt voor het bieden van een generieke oplossing. Zo kunnen generieke code en applicatie-specifieke code op een nette manier gescheiden worden.

Voor en Nadelen

Wanneer op een nette manier met objecten wordt omgegaan, bijvoorbeeld door middel van template methoden, bieden de oude technieken een redelijke ondersteuning voor generieke oplossingen. Het gaat hierbij vooral om generieke oplossingen waarbij de applicatie-specifieke kenmerken op code-niveau makkelijk te scheiden zijn van gemeenschappelijke functionaliteit.
Jammergenoeg zorgt een generieke oplossing op code-niveau niet voor een extra abstractie die in veel gevallen wel gewenst is. Een ontwikkelaar zal nog steeds op code-niveau aan moeten geven hoe de generieke klassen gebruikt moeten worden.
In combinatie met functionele patronen zou het mooier zijn om een techniek te kunnen gebruiken die het voor ontwikkelaars mogelijk maakt om applicatie-specifieke kenmerken toe te voegen vanaf een hoger abstractie niveau.

Nieuwe Technieken

Conceptuele (functionele) patronen bekijken software vanaf een hoger abstractie-niveau; wat zijn de verbanden tussen concepten binnen het domein, hoe kunnen deze concepten verwerkt worden binnen software en hoe moeten deze samenwerken om een bepaalde functionaliteit te kunnen bieden.
Wanneer een ontwerper specificeert hoe een generieke oplossing gebruikt moet worden zou het mooi zijn als hij dit kan doen op een abstractie-niveau dat niet te ver afligt van het conceptuele niveau.
Abstraheren is een van de belangrijkste begrippen binnen de informatica. Onderzoek naar technieken die het mogelijk maken om software vanaf een hoger abstractie-niveau te ontwerpen is iets dat altijd al belangrijk is geweest en zal blijven binnen de informatica.
Bekende ontwikkelingen hierin zijn bijvoorbeeld geweest: de ontwikkeling van procedurele programmeertalen als abstractie over machine-instructies en functionele/logische talen over procedurele talen.
Door de toenemende grootte en complexiteit van software-systemen wordt het steeds lastiger om applicaties puur op code-niveau te ontwikkelen. Nieuwe technieken probren hierop in te springen door een extra abstractie te bieden boven

op de gebruikelijke object-structen in een object-georiënteerde programmeertaal als Java.

Aspect Oriented Programming

Aspect oriented programming(AOP) maakt het mogelijk om binnen een object-georiënteerde structuur te abstraheren over methoden/klassen [Lee02]. Dit kan door het definiëren van *crosscuts* (dwarsdoorsnedes van het systeem) en *aspects* (functionele aspecten) van deze crosscuts.
Met AOP kan een functioneel aspect, dat normaal gesproken op veel verschillende plaatsen binnen een systeem voorkomt, op één plaats beschreven worden. Een eenvoudig voorbeeld hiervan is om alle methodes binnen een programma in één keer te selecteren en hier logfunctionaliteit aan toe te voegen:

```
aspect Logger {
  pointcut log(): call(public * Obj.*(..));

  before(): log() {
    System.out.println("before method call");
  }
  after(): log() {
    System.out.println("after method call\n");
  }
}
```

Feitelijk geven crosscuts en aspecten de ontwikkelaar de mogelijkheid om vanuit een ander perspectief naar de programmacode te kijken. Het voordeel hiervan is, dat hierdoor globale functionaliteit op een eenvoudige lokale manier aan een (bestaande) objectstructuur kan worden toegevoegd. Dit betekent echter ook dat de code opeens vanuit twee perspectieven bekeken kan worden. Bij het hergebruik van een functioneel aspect is dit een voordeel; door de *crosscut* aan te passen kan een *aspect* op een ander punt binnen een systeem werken.
Het hebben van twee *views* kan echter ook een nadeel zijn wanneer de samenwerking tussen beide niet duidelijk is.
AOP wordt altijd gebruikt in combinatie met een object-georiënteerde taal. Alle mogelijkheden die deze taal heeft zullen dan ook nog steeds gebruikt kunnen worden in combinatie met AOP. Jammergenoeg brengt AOP het ontwerp en de implementatie van een systeem niet veel dichter bij het applicatiedomein; een systeem zal nog steeds op code-niveau gespecificeerd moeten worden; hoewel aspecten er wel voor kunnen zorgen dat functionaliteit op een mooiere manier binnen deze code verwerkt kan worden.

Feature Oriented Programming

Feature Oriented Programming(FOP) is een manier van programmeren waarbij uitgegaan wordt van een aantal vooraf gedefinieerde functionele eigenschappen voor producten binnen een productlijn: *features*; een uitgebreidere beschrijving van FOP is te vinden in *Sectie Feature Oriented Programming*{*93*}.
In een featuremodel wordt gemodelleerd welke features er binnen een productlijn zijn en wat de relatie tussen deze features is (in *Figuur C.1* is een voorbeeld hiervan te zien). Door het selecteren van features kan vervolgens een specifiek product samengesteld worden [Bat03a].

Hoe een selectie van features, *feature-set*, vervolgens omgezet wordt naar een product staat los van deze specificatie. Hiervoor zal een mechanisme bedacht moeten worden dat het mogelijk maakt om op een eenvoudige manier features te combineren tot een product. De complexiteit van dit mechanisme zal afhangen van de mogelijkheden binnen het featuremodel.
Het verwerken van applicatie-specifieke kenmerken, zoals een gegevensmodel, is iets waar niet direct rekening gehouden wordt bij FOP. Er wordt vanuit gegaan dat een product (vrijwel) volledig te specificeren is door aan te geven wat zijn functionele eigenschappen zijn.
Het ontwerpen van een product op basis van functionele eigenschappen zorgt voor een grote abstractie; het definiëren van een feature-set zit op vrijwel hetzelfde niveau als het opstellen van de *requirements* van een product.
Het voordeel van *feature oriented programming* is dat van te voren al vastligt welke features of combinaties van features er gekozen kunnen worden. Hier kan bij de implementatie op worden ingespeeld. Dit kan het ontwikkelen van bijvoorbeeld losse componenten voor een bepaalde functionaliteit een stuk eenvoudiger maken.

Domain-specific Languages

Domeinspecieke talen(DSL) zijn bedoeld voor het schrijven van software voor een specifiek domein.
De syntax van een DSL zal aangepast zijn op concepten die binnen dit domein voorkomen. De taal hoeft niet alle mogelijkheden te hebben van een *general-purpose language*(GPL); de context waarbinnen de taal gebruikt zal worden is van te voren bekend en enkel eisen binnen die context zijn relevant. Zo zal het binnen een taal voor mathematische expressies niet direct nodig zijn om ondersteuning te bieden voor 3D-geluidseffecten.

Een voorbeeld van een bekende en algemeen-gebruikte domeinspecifieke taal is *SQL*. SQL is een taal speciaal bedoeld om acties uit te voren op een (relationeel) database:

- data opvragen (`SELECT`), toevoegen (`INSERT`), aanpassen (`UPDATE`), en verwijderen (`DELETE`),
- data selecteren aan de hand van voorwaarden (`SELECT .. WHERE`),
- tabellen combineren (`JOIN`),
- resultaten groeperen (`GROUP`) en sorteren (`SORT`),
- ...

Deze acties komen allemaal op een intuïtieve manier terug in de syntax.
Het voordeel van een domeinspecifieke taal is dat het gat tussen *'wat de programmeur wil bereiken'* en *'hoe hij dit moet programmeren'* zo klein mogelijk gehouden kan worden. Syntax en eventuele foutmeldingen kunnen binnen de context van het domein gehouden worden.
Het ontwerpen van een domeinspecifieke programmeertaal voor een domein is zeker niet triviaal.
Allereerst zal er een syntax bedacht moeten worden die concepten binnen het domein op een inzichtelijke manier representeerd; daarbij is het belangrijk dat alles wat de programmeur zou willen binnen het applicatiedomein vanuit deze

syntax te doen is.
Vervolgens zal er gekeken moeten worden in hoeverre het nodig is om parsers/-compilers te schrijven voor de taal: kan er gebruik gemaakt worden van een andere taal en de DSL hierop baseren of zal er een complete compiler geschreven moeten worden die compileert naar machine-instructies?
In een taal als *Haskell* is het mogelijk om een domein-specifieke taal of combinator-library[1] te specificeren binnen *Haskell* zelf. Op deze manier kan er gebruik gemaakt worden van de bestaande technische architectuur terwijl wel het voordeel van de hogere abstractie geboden kan worden. Voorbeelden van hiervan zijn parser/pretty-printing-libraries en combinators voor financiële producten [ea00].

Model-Driven Architecture

Een programmeerparadigma dat primair ontworpen is voor het bieden van een extra abstractie boven (OO)code is *Model Driven Architecture*(MDA); een uitgebreidere beschrijving van MDA is te vinden in *Sectie Model Driven Design*{*80*}.
Het idee achter MDA is om software te ontwerpen vanuit een (platform-onafhankelijk) visueel model. Door middel van (platformspecifieke) mappings zou dit model uiteindelijk vertaald kunnen worden naar een concrete implementatie.
Het grafische model, dat gebruikt wordt om het systeem te ontwerpen, zou een inzichtelijke kijk moeten kunnen bieden op het hele systeem. In het model zal een ontwikkelaar genoeg details aan moeten kunnen geven om elk gewenst systeem te kunnen specificeren. Hierbij zal er wel voor gezorgd moeten worden dat de details de inzichtelijkheid van het model niet verstoren. Het platform waarnaar gemapt wordt of de mappings zullen dan ook genoeg functionaliteit moeten bieden om dit te voorkomen.
Voor algemene systemen zal het erg lastig, zo niet onmogelijk, zijn om aan deze eisen te voldoen. Het gevaar is dat, wanneer er teveel details in het model zitten, het voordeel van het gebruik van een visueel model verdwijnt: het overzicht, de structuur, vervaagd.
Het vinden van de juiste abstracties, syntax en mappings binnen het model is zeer complex en het op grote schaal toepassen van MDA voor een complete applicatie zal op korte termijn (vrijwel) niet haalbaar zijn.
Binnen een kleiner domein kan het wel realistisch zijn om het idee achter MDA te gebruiken, een voorbeeld hiervan voor pensioenberekeningen is uitgewerkt in *Sectie Grafische specificatie*{*66*}.
Uiteraard zullen er wel tools nodig zijn waarin het grafische model gemodelleerd kan worden.

Domain-driven design

Domain-driven design(DDD) is een ontwikkelmethode om met de standaard object-georiënteerde technieken een systeem te ontwerpen vanuit het domein; een uitgebreide beschrijving is te lezen in *Sectie Domain-Driven Design*{*85*}.
Bij DDD wordt geprobeerd het domein op een conceptueel zo natuurlijk mogelijke manier terug te laten komen in de software. Dit kan door relatief eenvoudige dingen zoals duidelijke naamgeving, goede objectindeling. De basis hiervoor

[1] Een combinator-library is een verzameling van (domein-specifieke) operatoren die gebruikt kunnen worden om grotere taalconstructies, expressies, samen te stellen.

wordt gevormd door het hebben van een juist conceptueel beeld van het domein door iedereen die werkt aan het project.
In het algemeen wordt software ontworpen vanuit de techniek: *hoe is het, technisch gezien, het best/snelst/handigst om dit te implementeren.*
Bij DDD wordt gekeken hoe iets conceptueel het beste binnen het systeem zou passen: hoe zijn de relaties van concepten binnen het domein en hoe kunnen deze binnen een systeem representeerd worden.
Door een sterkere koppeling tussen domein en systeem zou het systeem beter aan moeten sluiten op de wensen van de gebruikers. Het idee is dat belanghebbenden (opdrachtgever, gebruikers, domeinexperts) ook mee kunnen denken tijdens de ontwikkeling van het systeem om ervoor te zorgen dat het domein op de juiste manier gerepresenteerd wordt.
Hoewel een ontwikkelaar op hetzelfde abstractie-niveau een systeem zal blijven ontwikkelen zal het uiteindelijke systeem dichter bij het domein en het functioneel ontwerp liggen. De winst van het gebruik van domain-driven design ligt niet op het gebied van programmeren. Het programmeren zelf zal door DDD niet sneller gaan; waarschijnlijk zelfs langzamer aangezien nu niet alleen rekening gehouden hoeft te worden met de techniek maar ook met het domein, hetgeen vaak lastiger is. De winst ligt vooral op het gebied van analyse; met DDD ontwikkelde software zou beter aan de eisen van de gebruikers moeten voldoen. Hetgeen uiteindelijk toch bepaled zal zijn voor een project.

Samenvatting

Het ideale programmeerparadigma, in combinatie met functionele patronen, zou de volgende eigenschappen moeten hebben:

- inzichtelijke specificatie, dicht bij het domein,
- oplossingen eenvoudig herbruikbaar,
- minder werk voor ontwikkelaar, maar
- wel alle noodzakelijke mogelijkheden.

Er is niet direct één techniek aan te wijzen die zomaar aan al deze punten voldoet.
Aspect-oriented programming is een techniek die op code-niveau voor een extra abstractie kan zorgen en hiermee werk kan besparen van de ontwikkelaar; ook de herbruikbaarheid van oplossingen wordt vergroot. AOP zorgt echter niet direct voor een hechtere band met het domein.
Feature-oriented programming zorgt voor een extra abstractie binnen een productlijn door middel van featuremodellen waaruit de ontwikkelaar een product kan assembleren. Functionele patronen zijn goed te gebruiken in combinatie hiermee. Om FOP te kunnen gebruiken voor het concreet samenstellen van een product zal een techniek gebruikt moeten worden die het mogelijk maakt om *features* eenvoudig met elkaar te combineren. Dit zou bijvoorbeeld kunnen met behulp van voorgedefinieerde componenten in combinatie met AOP.
Het gebruik van FOP zal vrijwel alleen zinnig zijn wanneer er veel gelijksoortige producten in een productlijn ontwikkeld moeten worden en waarbij de features op zichzelf altijd gelijk zijn.

Model-driven architecture zou aan al de hierboven gestelde eisen *kunnen* voldoen. De vraag is echter in hoeverre het mogelijk is om om een complete applicatie te ontwikkelen vanuit een (visueel) model dat op zichzelf nog inzichtelijk is. Voor een afgeschermd domein is het vaak wel mogelijk om een inzichtelijke syntax te ontwerpen en de bijbehorende mappings. Dit is vergelijkbaar met het ontwerpen van een domeinspecifieke taal, alleen dan grafisch.
Bij beiden, MDA en een DSL, is het nodig om een onderliggende techniek te hebben die voor de vertaling zorgt tussen (domeinspecifieke) syntax naar 'normale' programmacode of assembly.
Domain-driven design is een *methode* die ervoor kan zorgen dat het ontwerp van een systeem een hechte koppeling heeft met het applicatiedomein. Door DDD te combineren met een techniek als MDA of een domein specifieke taal kan ervoor gezorgd worden dat naast de winst op analyse-gebied ook op het gebied van daadwerkelijke implementatie winst behaald kan worden. Functionele patronen kunnen een belangrijke rol spelen hierbij aangezien zij de vertaling van domein naar software vastleggen; hetgeen in DDD juist belangrijk is.
Welke techniek er ook gekozen wordt; functionele patronen kunnen helpen om vast te stellen wat de belangrijkste functionele eigenschappen zijn maar er zal altijd gekeken moeten worden in hoeverre applicatiespecifieke kenmerken een rol spelen.

Hoofdstuk 5

Patterns binnen Quinity

Binnen *Quinity B.V.* wordt er veel gebruik gemaakt van design patterns.
Het gaat hierbij veelal om de bekende technische patronen [GHJV95, BMR+96, SSRB00] die gebruikt worden bij de implementatie van systemen en analysis patterns [Fow96] voor de representatie van gegevens binnen een gegevensmodel.
Daarnaast wordt er gebruik gemaakt van zelf beschreven functional design patterns op het gebied van algemene functionaliteit in administratieve toepassingen.
De meeste van deze patronen zijn geschreven naar aanleiding van een nieuw systeem waarin een functionaliteit nodig was die ook al in eerder ontwikkelde systemen aanwezig is. Door deze functionaliteit te beschrijven in functional design patterns kan deze later eenvoudiger hergebruikt worden.
Tot nu toe zijn er vooral *low-level* functional design patterns beschreven voor hele algemene problemen die terugkomen in vrijwel alle administratieve systemen die door *Quinity B.V.* worden ontwikkeld.
Dat er weinig *higher-level* patronen zijn voor een specifieker domein heeft een aantal oorzaken; allereerst heeft het schrijven van patronen alleen zin als een functionaliteit vaker voorkomt: de kans dat een domein-specifieke funtionaliteit in meerdere applicaties terugkomt is kleiner dan bij algemene functionaliteit.
Daarnaast is er voor het schrijven van patronen voor een specifiek domein meer domeinkennis vereist: er zullen minder mensen zijn met deze kennis en daarnaast kost het helder vastleggen hiervan vaak veel tijd.
Toch kan het erg nuttig zijn om ook functional design patterns te hebben specifiek voor één domein, juist om ervoor te zorgen dat domeinkennis op een gestructureerde manier beschikbaar is voor de mensen die aan een project werken.

Generieke oplossingen

Voor veel van deze functional design patterns zijn er binnen *Quinity B.V.* generieke oplossingen aanwezig.
Deze oplossingen kunnen grofweg verdeeld worden in twee categorieën:

- applicatie-onafhankelijke oplossingen, en
- applicatie/data-specifieke oplossingen.

Applicatie-onafhankelijke oplossingen

Bij *data-onafhankelijke oplossingen* gaat het om concrete oplossingen die voor iedere administratieve toepassing nagenoeg gelijk is; onafhankelijk van de structuur van de data die door het systeem wordt verwerkt. Het kan hierbij bijvoorbeeld gaan om het contact leggen met het systeem, autorisatie, het weergeven van foutmeldingen et cetera.
Aangezien deze functionaliteit in vrijwel elke applicatie nodig is, is deze ingebouwd in het *Quinity-framework*. Dit framework vormt de basis voor vrijwel elke (web)applicatie die gebouwd wordt binnen *Quinity B.V.* . Het framework verzorgt alle standaardfunctionaliteit die nodig is binnen een webapplicatie: het afhandelen van requests van gebruikers, het versturen van pagina's, het verbinden met een onderliggend database, sessiebeheer et cetera.
Applicatie-specifieke functionaliteit kan later door ontwikkelaars worden toegevoegd als laag boven op dit framework door middel van het aanroepen van methodes binnen het framework en overerving van basisklassen binnen het framework.
Het kan verstandig zijn om de generieke oplossing voor een functioneel patroon te integreren in het framework als één of meer van de volgende punten geldt.

- Als de functionaliteit enkel afhangt van de gebruikte techniek, niet van datamodel of veel applicatiespecifieke kenmerken.
- Als de functionaliteit in vrijwel alle applicaties nodig is.
- Als het toevoegen van de functie geen invloed heeft op systemen die deze functionaliteit niet gebruiken.
- Als de functionaliteit conceptueel binnen het framework past.
- Als een los library lastig te combineren zou zijn met het framework .

Wanneer een functionaliteit enkel gebruikt wordt binnen een specifiek domein, bijvoorbeeld pensioenberekeningen, en bij andere applicaties volstrekt overbodig is, is het mooier om gebruik te maken van losse modules die per domein toegevoegd kunnen worden aan het framework.

Applicatie/data-specifieke oplossingen

Bij *applicatie/data-specifieke oplossingen* gaat het om generieke oplossingen waarvan de concrete implementatie binnen een systeem afhangt van applicatie specifieke eigenschappen of een onderliggend datamodel.
Vrijwel alle administratieve applicaties binnen *Quinity B.V.* worden gebouwd rondom de gegevens die worden verwerkt. Het data-model speelt hierin een cruciale rol aangezien deze de structuur aangeeft van deze data. Veel patronen zijn direct afhankelijk af dit datamodel voor het opslaan/representeren/wijzigen van gegevens (zie ook *Sectie Voorbeeldpatronen*{*30*}).

Het datamodel wordt gebruikt om een concrete implementatie voor een aantal van de basisfunctionaliteiten te genereren: De generieke oplossing definieert een template met daarin de algemene implementatie voor de functionaliteit;
Bijvoorbeeld het tonen van de details van een entiteit (EntiteitBekijken):
De naam en de velden van de entiteit die bekeken wordt zijn toepassingspecifiek en binnen de generieke oplossing niet bekend. Hiervoor zal verwezen moeten

worden naar een later toe te kennen datamodel, waarbij elke op zichzelf staande tabel de EntiteitBekijken-functionaliteit krijgt:

1. Er wordt een subklasse aangemaakt van de `StdEnitity`-klasse specifiek voor deze entiteit. De `StdEnitity`-klasse is een onderdeel van het framework en beschikt standaard over de basiskennis om data binnen het systeem te representeren op te slaan, op te vragen, te wijzigen et cetera. Echter de velden die moeten worden opgeslagen zijn per entiteit verschillend en code (SQL) zal dan ook per entiteit gegenereerd moeten worden binnen de subklasse.
   Ook worden er getters en setters toegevoegd zodat de velden van de entiteit eenvoudig benaderd kunnen worden.

2. Er wordt een detailscherm (Detailscherm) aangemaakt met daarop alle attributen van de entiteit. De types van de waarden van deze velden kunnen uit het datamodel gehaald worden om op de juiste manier gerepresenteerd te worden.

3. Het detailscherm en de onderliggende representatie van de entiteit (uit eerste stap) worden aangestuurd door gegenereerde subklassen van `StdHandler` en `StdController` die ervoor zorgen dat de gebruiker het detailscherm kan benaderen en de waarden in het detailscherm worden gezet.

Een soortgelijk generatieproces kan ook gebruikt worden bij lijsten van entiteiten (EntiteitenOverzicht) en zoekschermen.

Aanpassingen Gegenereerde Code

Door de combinatie van codegeneratie vanuit het datamodel en overerving van generieke klassen binnen het framework kan veel gelijksoortig programmeerwerk bespaard worden.
Enkel het applicatie-specifieke gedrag zal later door een programmeur moeten worden toegevoegd aan de gegenereerde code of *aangepast*. Het nadeel van aanpassen is dat wanneer dit niet netjes gebeurt, en gegenereerde code en aangepaste code door elkaar komen, het later onmogelijk is om code opnieuw te genereren zonder dat de aanpassingen van de programmeur verloren gaan. Ook is het onderhoud erg lastig wanneer de plaatsen waar aanpassingen gedaan zijn aan generieke code niet duidelijk te herkennen zijn.
Het is daarom verstandig om een methode te kiezen waardoor gegenereerde code en aangepaste code goed te scheiden zijn. Dit kan bijvoorbeeld door middel van overerving of voorgedefinieerde annpassingspunten (zie *Sectie Bestaande Technieken*{*47*} of een mogelijkheid te bieden om applicatie-specifieke kenmerken al binnen de specificatie te kunnen specificeren.

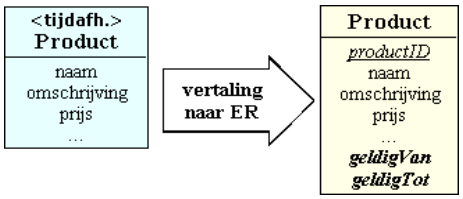
Beperkingen ER-model als specificatietaal

Op dit moment wordt code binnen *Quinity B.V.* gegenereerd vanuit een standaard ER-diagram welke in dit geval als specificatie-taal wordt gebruikt. Het voordeel van een standaard specificatie-taal als een ER-diagram is dat er volop tools beschikbaar zijn waarin dergelijke diagrammen gecreëerd kunnen worden. Het gebruik van ER heeft echter ook grote nadelen;

Het beschreven datamodelafhankelijke generatie proces gaat uit van een standaard ER-datamodel dat precies aangeeft welke tabellen er in een database aanwezig zijn en welke attributen deze entiteiten hebben. Patronen als EntiteitBekijken opereren in feite vanaf dit niveau. Er zijn echter ook functional design patterns en analysis patterns op een hoger niveau zoals het bijhouden van mutaties en rekening houden met tijdsafhankelijkheid.
Deze patronen opereren op een niveau boven het feitelijke database-model;
Bijvoorbeeld:
Wanneer een entiteit tijdsafhankelijk is zal er per record in de tabel moeten worden bijgehouden vanaf en tot wanneer de waarden van dit record geldig zijn. Voor elke tijdsafhankelijke entiteit is dit gelijk. Het mooiste zou dan ook zijn om van een entiteit expliciet aan te kunnen geven dat hij tijdsafhankelijk is in plaats van handmatig deze velden bij elke tijdsafhankelijke entiteit toe te voegen. Hetzelfde geldt voor mutatiesmechanismen, databaselog-functionaliteit, attributieve entiteiten en andere patronen die rechtstreeks invloed hebben op de structuur van het database-model.
Een ander nadeel is dat ER-diagrammen voor een doel gebruikt worden waarvoor het eigenlijk niet ontworpen is; de specificatie van een systeem. De expressiviteit van ER is dan ook niet groot genoeg om extra, (applicatie-specifieke) details aan te kunnen geven.

Database-gerelateerde specificatietaal

In plaats van het directe gebruik van ER-modellen als specificatie voor de data in een systeem zou er ook gebruik gemaakt kunnen worden van een specificatiemodel dat abstraheert van het feitelijke datamodel. Binnen dit model zouden vervolgens eigenschappen van gegevens aangegeven worden worden.
Een voorbeeld hoe een, als *tijdafhankelijk* gemarkeerde, entiteit vertaald kan worden naar een databasetabel is te zien in *Figuur 5.1*.



Figuur 5.1: Data-specificatie naar ER

Het abstraheren over het feitelijke datamodel heeft als voordeel dat eigenschappen van data op een eenvoudigere manier hergebruikt kunnen worden.
Er zal wel een duidelijke keus gemaakt moeten worden of de specificatietaal enkel als hulpmiddel gebruikt wordt om een ER-datamodel te genereren of dat het in de specificatietaal gedefinieerde gegevensmodel als uitgangspunt dient voor de gehele applicatie. Dit laatste zou het mooiste zijn maar heeft als nadeel dat de ontwerper alleen via een indirectie invloed heeft op het feitelijke datamodel. Dit kan een probleem vormen wanneer de programmeur binnen de software de databron wel direct moet kunnen benaderen en de structuur hiervan moet kennen, bijvoorbeeld bij het opvragen van gegevens uit het database. Naast de vertaling tussen specificatietaal naar ER zal er dan ook een omgekeerde vertaling moeten

komen vanaf feitelijke data naar gespecificeerde gegevensstructuur.

Hoofdstuk 6

Patronen voor Pensioenberekeningen

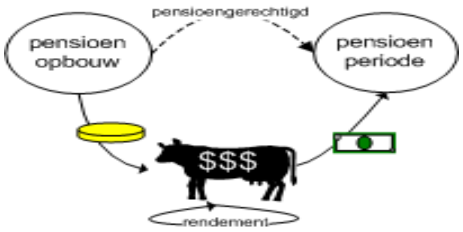
Conceptuele patronen -analysis patterns (zie *Sectie Analysis Patterns{14}*) en functional design patterns- binnen *Quinity B.V.* - worden vooral gebruikt voor algemene patronen voor adminstratieve toepassingen:

- Het representeren van producten en hun eigenschappen
- Het omgaan met mutaties en tijdafhankelijkheid
- Het weergeven en zoeken van gegevens door de gebruiker van het systeem
- ...

Binnen een specifieker domein komt het echter ook vaak voor dat een bepaalde functionaliteit of een specifieke berekening in meerdere systemen terugkomt.
Zo komen in veel applicaties voor pensioenverzekeraars pensioenberekeningen voor die veel gelijkenis met elkaar vertonen. Door de algemene structuur van deze berekeningen te beschrijven en de betekenis van subberekeningen binnen deze structuur zou het mogelijk moeten zijn om oplossingen en kennis over pensioenberekeningen te kunnen hergebruiken.
Pensioenberekeningen dienen in dit geval als voorbeeld voor een analyse naar het nut van functional design patterns op het gebied van complexe domeinspecifieke berekeningen.

Inleiding Pensioenberekeningen

Bij (ouderdoms)pensioenberekeningen gaat het om de berekening van de hoogte van de pensioenuitkering die mogelijk is op basis van *beschikbare premie*.
Tijdens de pensioenopbouw betaald de verzekerde per periode (bijvoorbeeld een jaar) premie aan de pensioeninstelling. De pensioeninstelling beheert dit geld voor de deelnemer en keert, als vergoeding voor het beschikbaar stellen van het geld, rendement uit aan de deelnemer. De betaalde premie en het rendement zorgen ervoor dat het depot van de deelnemer groeit. Als de deelnemer stopt met werken, en de pensioengerechtigde leeftijd heeft bereikt, zal het opgebouwde depot gebruikt worden om hem van een uitkering te voorzien tot aan zijn dood (zie *Figuur 6.1*).



Figuur 6.1: Pensioen Opbouw

Naast de voorwaardse pensioenberekening -het bepalen van de mogelijke uitkering op basis van een bekende premie- is vaak ook de *omgekeerde pensioenberekening* van belang. Bij de omgekeerde pensioenberekening wordt bepaald hoeveel premie er betaald moet worden voor een gewensde uitkering.

Doel

Domeinkennnis

Het doel van functionele patronen voor pensioenberekeningen is in eerste instantie om domeinkennis op het gebied van pensioenberekeningen vast te leggen en te structureren.
Een ontwerper/ontwikkelaar van een nieuw systeem zal over het algemeen niet over voldoende domeinkennis beschikken om een dergelijke complexe berekening in één keer te snappen wanneer deze wordt voorgeschoteld door de opdrachtgever.
Naast het gebrek aan basiskennis op het gebied van pensioenberekeningen zijn er een aantal factoren die het implementeren van een berekening compliceren:

- Totale berekening kan zeer complex zijn.
- Het onderscheid tussen algemene structuur en (systeem-specifieke) details is vaak lastig te herkennen.
- Het materiaal wat aangeleverd wordt door de opdrachtgever is niet altijd even inzichtelijk.

Door een algemene structuur te schetsen waarin vrijwel elke pensioenberekening gegoten kan worden zou een ontwerper/ontwikkelaar eerder in staat moeten zijn de eisen en wensen van de klant te bevatten. Het gaat hierbij niet alleen om het begrijpen van de requirements die een klant opstelt maar ook om eventueel foute of onhandige keuzes van een klant te herkennen. Het is de vraag of de contactpersoon van het bedrijf zelf wel precies weet wat hij wil dat het systeem doet of dat hij enkel bestaande berekeningen heeft meegenomen met het idee: *"deze berekeningen werken dus in het nieuwe systeem willen we deze op precies dezelfde manier"*. Hoewel het beschreven doel veel lijkt op simpelweg 'het overdragen van domeinkennis' is de koppeling met software voor een ontwikkelaar essentieel.
In het geval van een berekening lijkt deze relatie minder sterk aanwezig;
Wiskunde en Informatica zijn echter twee vakgebieden die sterk met elkaar verbonden zijn en het beschrijven van een berekening lijkt dan ook vanzelf veel op

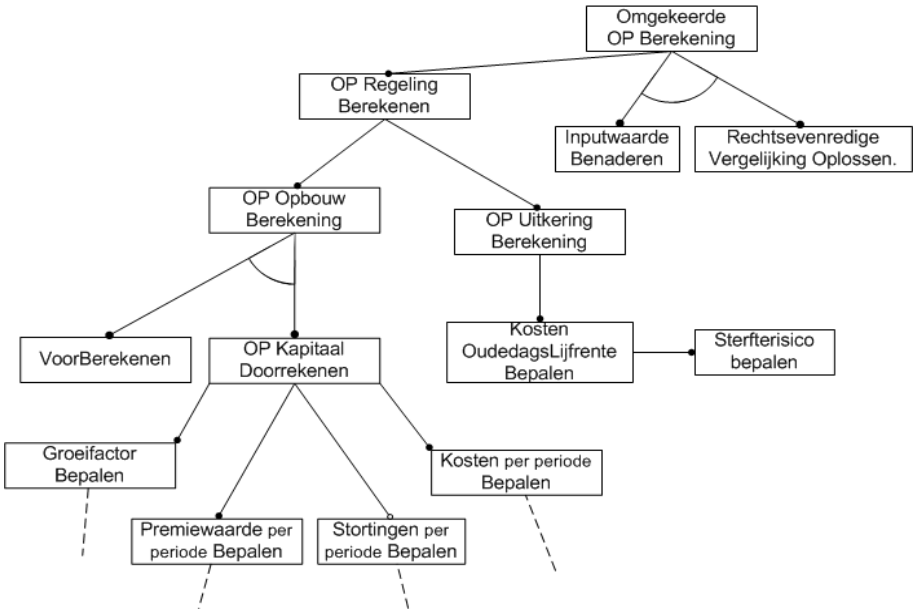
het beschrijven van een programma (of onderdeel van). Door een berekening in een structuur te gieten die eenvoudig te vertalen is naar software zou het eenvoudiger moeten zijn een calculatie op te zetten voor een nieuw systeem.

Hergebruik Implementatie

Naast het beschrijven van domeinkennis kunnen functionele patronen voor pensioenberekeningen ook de herbruikbaarheid van implementaties van berekeningen vergroten.
Op de eerste plaats kunnen de patronen ervoor zorgen dat berekeningen een standaardstructuur hebben binnen een systeem. Het voordeel hiervan is dat de implementaties tussen verschillende systemen eenvoudiger uitgewisseld kunnen worden. Op de tweede plaats zou een (generieke) concrete oplossing geboden kunnen worden op basis van de beschreven patronen. Het zou hierbij kunnen gaan om een indirecte (methodische) beschrijving hoe een dergelijke berekening geïmplementeerd zou kunnen worden, een directe generieke oplossing of een combinatie hiervan.

Structuur beschreven patronen

De beschreven functional design patterns beschrijven een algemene structuur voor pensioenberekeningen en daarnaast het *'wat'* ,*'hoe'* en *'waarom'* van de (sub)berekening per term.
Deze patronen zijn opgesteld aan de hand van een aantal projecten binnen *Quinity B.V.* waarin pensioenberekeningen voorkwamen.
De structuur van de berekening komt terug in de relaties tussen de verschillende functionele patronen; in *Figuur 6.2* is een klein gedeelte van deze structuur te zien. Een compleet overzicht van alle patronen is te vinden in [Sni04a].
Het globale idee achter de pensioenberekening is om de berekening op te delen in twee gedeeltes (Ouderdomspensioen Kapitaal Doorrekenen[Sni04a]): een *opbouwfase* en een *uitkeringsfase*. Deze harde grens is mogelijk aangezien bij een oudersdomspensioen al van te voren vastligt op welke leeftijd iemand stopt met werken.
In de opbouwende fase (Ouderdomspensioen Opbouw Berekening[Sni04a]) wordt er doorgerekend met de betaalde premie en kosten per periode tussen het afsluiten van de regeling tot de pensioenleeftijd van de deelnemer (Ouderdomspensioen Kapitaal Doorrekenen[Sni04a]).
In de uitkeringsfase (Ouderdomspensioen Uitkering Berekening[Sni04a]) wordt er gekeken hoe hoog de levenslange uitkering is die aangekocht kan worden met het geld dat is opgebouwd in de opbouwende fase. Hiervoor wordt er gebruik gemaakt van een actuariële berekening die rekening houdt met de kans dat een persoon een bepaalde leeftijd behaald (Kosten Oudedagslijfrente Bepalen[Sni04a]).
De details van de verschillende subberekeningen en omgekeerde pensioenberekening worden besproken in [Sni04a].



Figuur 6.2: Structuur Pensioenpatronen

Toepassing pensioenpatronen

Om de herbruikbaarheid van functional design patterns te vergroten is het zinnig na te gaan in hoeverrre deze patronen omgezet kunnen worden naar een concrete, liefst generieke, oplossing voor het probleem.
In het geval van de patronen voor pensioenberekeningen is hierbij gekozen voor een combinatie van een directe concrete oplossing in de vorm van libraries voor een aantal basisberekeningen en een indirecte oplossing in de vorm van een aantal referentie-implementaties voor de overkoepelende berekening; een uitgebreide beschrijving van deze implementatie is te vinden in [Sni04b].

Libraries

De actuariële basisberekeningen die binnen een pensioenberekening, of andere financiële berekening, gebruikt worden, zullen er vrijwel altijd precies hetzelfde uitzien. Het gebruik van deze functies zal echter wel verschillen en zorgt er dan ook voor dat financiële producten van elkaar verschillen. Berekeningen die beschreven kunnen worden in standaard-libraries zijn:

- berekeningen op het gebied van groei:
  - groeifactor (Groeifactor Bepalen[Sni04a]),
  - termijnfactor (Termijnfactor Bepalen[Sni04a]),
  - ...
- levenskansen (Overlevingskans/sterfterisico bepalen[Sni04a]), en
- kosten voor (levenslange) uitkeringen (Kosten Oudedagslijfrente Bepalen[Sni04a]).

Daarnaast is het door een wiskundige benadering van de omgekeerde terugre-

kening ook mogelijk om algoritmes hiervoor op te nemen in losse libraries die gebruikt kunnen worden in een willekeurige (pensioen)berekening.
De beschreven actuariële functies berekenen enkel een eindwaarde en hebben geen invloed op de rest van het programa (*side-effect-free*); deze functies kunnen dan ook vanuit een statische context gebruikt worden.
Voor de terugrekening is dit anders: hier zal gebruik gemaakt worden van de implementatie van de normale pensioenberekening. Door ervoor te zorgen dat deze berekening op een nette manier door te geven is aan het library kan er voor gezorgd worden dat er bij het gebruik zo weinig mogelijk code door de ontwikkelaar toegevoegd hoeft te worden om gebruik te kunnen maken van de generieke terugrekening.
De functionele patronen kunnen in dit geval dienen als (uitgebreide) documentatie van de libraries. Waarin wordt aangegeven wat de betreffende functies berekenen, hoe/waarom ze dit doen en wanneer/hoe ze te gebruiken zijn.
Referentie implementaties
De globale structuur van verschillende pensioenberekeningen is vrijwel gelijk echter op detailniveau zijn er vaak nogal wat verschillen die het erg lastig maken om een algemene generieke oplossing te ontwerpen die altijd te gebruiken is. In plaats daarvan is het wel mogelijk om een aantal voorbeeld-implementaties te ontwerpen die als referentie kunnen dienen bij de implementatie van een nieuw systeem.
Binnen deze implementatie kan voor de ingewikkelde actuariële berekeningen gebruik gemaakt worden van de standaard libraries. De precieze, maar op codeniveau vaak eenvoudige, structuur van de (sub)berekeningen zal door de ontwikkelaars zelf toegevoegd moeten worden. De referentie-implementaties laten zien hoe ervoor gezorgd kan worden dat er over de verschillende perioden doorgerekend kan worden, variabelen op de juiste manier doorgegeven worden, hoe de terugrekening gebruikt kan worden et cetera.
Als implementatie-voorbeelden zijn twee algemene strategieën uitgewerkt: een *periode-gebaseerde* procedurele aanpak en een *term-gebaseerde* object-georiënteerde aanpak.

Procedurele implementatie

Bij een procedurele aanpak wordt er gefocust op de complete berekening die binnen een periode van een pensioenberekening wordt uitgevoerd. De verschillende termen binnen de berekening worden beschouwd als variabelen zonder eigen logica. De programmeur zal expliciet aan moeten geven in welke volgorde de waarde van de verschillende termen berekend zal moeten worden. De complete berekening zal dan ook stap voor stap terug te vinden zijn in de implementatie.
De globale structuur van een procedurele implementatie bestaat uit:

- Voor de aansturing: een object van de hoofdklasse.
- Voor elke periode: een periode-object van de periodeklasse.

In de hoofdklasse zit een methode die in een lus voor elke periode van de berekening een periode-object aanmaakt en de berekening start. In het periode-object worden de waarden van alle termen opgeslagen voor de betreffende periode. Aan het eind van de berekening kunnen deze periodeobjecten gebruikt worden om het verloop van een waarde (bijvoorbeeld het kapitaal) te tonen.

Er zijn verschillende mogelijkheden waar de werkelijke berekening van de verschillende termen plaatsvindt en hoe waarden doorgegeven worden tussen verschillende perioden. In [Sni04b] is een viertal mogelijkheden uitgewerkt. Een voorbeeld van één van deze mogelijkheden, de *recursieve berekening*, wordt in *Sectie Grafische specificatie*{*66*} gebruikt.
Het voordeel van een procedurele implementatie is dat de complete berekening voor een periode vaak op een eenvoudige, directe, manier terugkomt in de implementatie. Er is weinig overhead van ingewikkelde klassenstructuren.
Een nadeel is wel dat de abstracte structuur van de berekening en subberekeningen niet erg duidelijk naar voren komt: verbanden tussen termen zijn niet altijd even duidelijk. Functionele patronen zijn dan ook lastiger te herkennen in de uiteindelijke implementatie door het gebrek aan structuur.

Object-georiënteerde implementatie

Bij de procedurele implementatie wordt de berekening stap voor stap voor elke periode opgebouwd. De waarde van een term wordt hierbij op periode-niveau opgeslagen.
Het is ook mogelijk om vanuit de termen zelf te werken:

1. Van welke andere termen hangt de waarde van deze term af?
2. Hoe wordt de waarde van deze term bepaald?
3. Blijft de waarde van een term de hele berekening geldig of zal deze elke periode opnieuw bepaald moeten worden?

De basis van een term-gebaseerde, object-georiënteerde berekening wordt gevormd door een objectstructuur waarin elke term in de berekening gerepresenteerd wordt door een eigen object.
Elk term-object bevat de logica die nodig is om zijn waarde te bepalen. Door middel van referenties naar andere term-objecten kan het object de waarden die hij tijdens zijn berekening nodig heeft opvragen. Wanneer de waarde voor een bepaalde periode is berekend zal dit in het object worden opgeslagen zodat deze de volgende keer wanneer deze wordt opgevraagd niet meer berekend hoeft te worden.

De volgorde waarin termen berekend moeten worden hoeft niet expliciet aangegeven te worden; de objecten zorgen er zelf voor dat wanneer hun waarde wordt opgevraagd voor een periode ze deze uitrekenen of de opgeslagen waarde retourneren. Dit werkt recursief, bijvoorbeeld:
Term A gebruikt term B en term B heeft term C nodig tijdens zijn berekening. Als de waarde van term A gevraagd wordt, zal object A, omdat hij voor zijn berekening term B nodig heeft, de waarde term B opvragen. Term B zal vervolgens de waarde van term C opvragen. Als de waarde van C is bepaald zal deze gebruikt worden om B te berekenen, enzovoort.
Hoewel enkel de waarde van term A wordt gevraagd zullen impliciet ook B en C berekend worden.
Het grote voordeel van de object-georiënteerde berekening is dat deze uitgaat van de berekening per term en de algemene aansturing impliciet gebeurt. Er is bijvoorbeeld geen lus nodig die ervoor zorgt dat alle perioden berekend worden:

het *eindkapitaal*-object vraagt de waarde van het *beginkapitaal*-object, welke vervolgens weer het *eindkapitaal* van de vorige periode opvraagt, enzovoort (zie voorbeeld in *Sectie Grafische specificatie*{*66*}).
Een term-gebaseerde implementatie heeft daarnaast als voordeel dat dit goed te combineren is met functionele patronen waarin wordt beschreven *hoe* een term precies wordt berekend. Patronen komen expliciet terug in de object-structuur.

Grafische specificatie

Afhankelijkheden tussen verschillende waarden binnen een tekstuele (procedurele) weergave van een (pensioen)berekening zijn soms lastig te herkennen. Diagrammen waarbij deze afhankelijkheden expliciet worden weergegeven kunnen de structuur van een berekening een stuk duidelijker maken. Een grafische weergave van de structuur van een (pensioen)berekening zou naast het verduidelijken van de berekening ook gebruikt kunnen worden om een berekening te specificeren. Deze specificatie zou vervolgens gebruikt kunnen worden om code te genereren voor deze berekening. Belangrijk is dan dat alle details van de berekening die nodig zijn voor de implementatie -het liefst op een inzichtelijke manier- kunnen worden uitgedrukt in dit model. Een aantal essentiele punten hierbij zijn:

- onderscheid tussen berekende waarden en invoerwaarden,
- onderscheid tussen variabele en constante waarden,
- expressies van (sub)berekeningen aan kunnen geven, en
- zichtbare verbanden tussen termen.

Tools om een grafisch model van een berekening te specificeren zijn grofweg in te delen in twee categorieën:

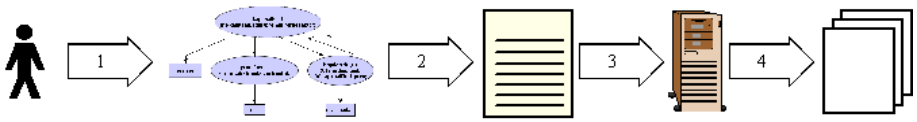
- *Specifieke tools:*
  Specifieke tools om berekeningen te visualiseren begrijpen de semantiek van de gebruikte notatie en zouden hierdoor zinnige dingen over de berekening kunnen zeggen.
  Zo zou een tool kunnen controleren of een berekening correct is, welke semantische verbanden er zijn tussen termen, et cetera: op basis van expressies in subberekeningen zouden de onderlinge verbanden tussen termen gevisualiseerd kunnen worden.
- *Algemene tools:*
  Algemene tools voor het tekenen van diagrammen kunnen gebruikt worden om een berekening te visualiseren, maar zij zullen de semantiek van de berekening niet begrijpen. Het zal niet mogelijk zijn voor de tool om iets zinnigs te zeggen over de correctheid van een berekening.
  In sommige applicaties is het mogelijk om constraints te definiëren; constraints kunnen gebruikt worden om bepaalde voorwaarden aan een diagram te controleren. Op deze manier kan een vorm van semantiek worden toegevoegd aan een algemene tool.

De keuze van het soort tool is van groot belang wanneer een grafisch model van een berekening vertaald wordt naar een concrete implementatie. Zo zal een gespecialiseerde tool weten welke informatie er essentieel is en zou de vertaling

zelfs verwerkt kunnen worden in de tool zelf. Bij een algemene tool zal de relevante informatie uit de algemene representatie van het diagram gehaald moeten worden.
In de volgende paragraaf wordt geschetst hoe een (pensioen)berekening met behulp van een algemene graaf-teken-tool gespecificeerd kan worden en vervolgens vertaald kan worden naar een concrete implementatie.

Vertaling grafische specificatie naar code

Het specificeren van een (pensioen)berekening in een algemene tekentool en de vertaling van deze specificatie naar een concrete implementatie vereist een aantal stappen (zie *Figuur 6.3*):



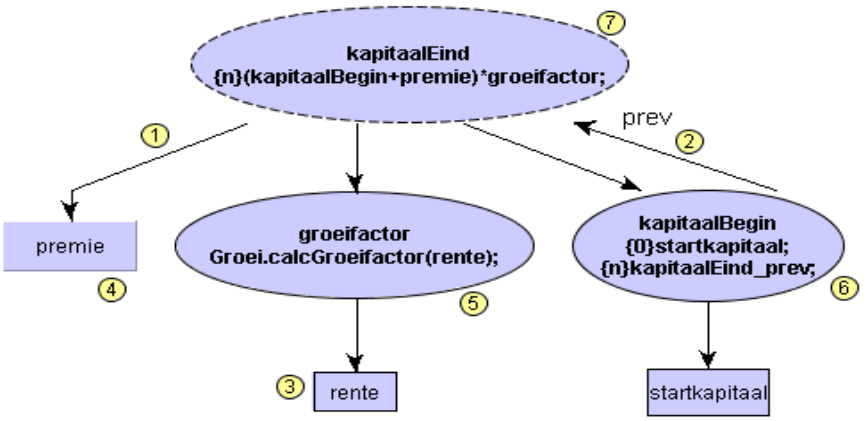
Figuur 6.3: Vertaling grafische specificatie

1. Specificeren van berekening in een model.
2. Opslaan van berekenings-model in een tussenformaat.
3. De relevante gegevens uit het tussenformaat halen.
4. Op basis van de relevante gegevens code voor de berekening genereren.

1. Specificeren van berekening in een model

Om een specifiek diagram, als de specificatie van een (pensioen)berekening, te kunnen verwerken binnen een standaard tool is het nodig om een manier te kiezen waarop relevante gegevens binnen de standaardnotatie 'gecodeerd' kunnen worden.
Hierbij is het belangrijk om een goede afweging te maken tussen begrijpelijkheid en expressiviteit.
Het visuele model in *Figuur 6.4* geeft een eenvoudige periodieke berekening weer waarbij elke periode premie wordt betaald en het kapitaal met een vaste groeifactor groeit.
Bij het kiezen van deze representatie is uitgegaan van de mogelijkheden binnen bestaande (algemene) tools om grafen te kunnen tekenen (in dit geval *yEd*[yWo]).
De gebruikte notatie:

1. Een **pijl zonder label** geeft een afhankelijkheid tussen twee termen in dezelfde periode aan.
   Het eindkapitaal in een periode is afhankelijk van de premie die in die periode betaald is.
2. Een **pijl gelabeld met prev** geeft een afhankelijkheid weer waarbij een term afhankelijk is van de waarde van een term uit de vorige periode.



Figuur 6.4: Grafische specificatie van kapitaalberekening

Het kapitaal aan het begin van een periode is afhankelijk van het kapitaal aan het eind van de vorige periode.

3. Een **rechthoek** geeft een constante invoerwaarde weer: de rente waarmee gerekend wordt aan het begin van de berekening gekozen en is voor alle perioden gelijk.

4. Een **3D-rechthoek** geeft een invoerwaarde aan die per periode kan verschillen.
   De premie kan per periode veranderen en zal elke periode opnieuw ingevoerd moeten worden; het bepalen van de hoogte hiervan wordt buiten de berekening gelaten.

5. Een **ellips** geeft een subberekening aan; oftewel een term die berekend moet worden. Onder de naam van de term kan de berekening gegeven worden die moet worden uitgevoerd (in dit geval in Java-code). Door $\{n\}$ voor deze berekening te zetten wordt aangegeven dat deze berekening elke periode moet worden uitgevoerd.
   Wanneer de term alleen afhankelijk is van constante (invoer)termen hoeft deze maar één keer (voor alle termen tegelijk) uitgevoerd te worden. Dat een berekende term constant is kan worden aangegeven door niets voor de berekening te zetten (dit zou eventueel ook afgeleid kunnen worden uit de afhankelijkheden). De groeifactor kan op basis van de (constante) rente bepaald worden met de standaardfunctie calcGroeifactor.

6. Bij sommige subberekeningen (**ellips**) moet de eerste periode iets anders gedaan worden dan in de andere periodes. Dit kan worden aangegeven door $\{0\}$ voor de berekening van de eerste periode te zetten en n voor de berekening die in de volgende periodes moet worden uitgevoerd.
   Het kapitaal aan het begin van een periode is afhankelijk van de hoogte van het eindkapitaal van de vorige periode. Bij de eerste periode is er geen vorige periode; in plaats daarvan kan een startkapitaal gegeven worden.

7. Door een subberekening te omlijnen met een **doorbroken lijn** kan expliciet aangegeven worden dat de waarde van deze berekening voor elke periode opvraagbaar moet zijn van buitenaf.
   Het is belangrijk dat het kapitaal aan het eind van een periode van buiten de berekening op te vragen is. Anders heeft de berekening weinig zin.

Aangezien een algemene tool niets weet over de semantiek van de gekozen notatie is het, zonder voorwaarden/restricties, niet direct mogelijk om te controleren of de beschreven berekening wel consistent is:

- Zijn de expressies die beschreven zijn in de labels wel correct?
- Zijn er geen circulaire afhankelijkheden binnen de berekening?
- Zijn alle verbanden wel expliciet aangegeven?
- ...

Dit betekent dat het voor een gebruiker niet direct te zien is of de door hem gemodelleerde berekening wel correct is.

2. Opslaan van berekenings-model in een tussenformaat

Wanneer een specificatie van een berekening wordt opgeslagen in een tussenformaat is het belangrijk dat alle relevante informatie behouden blijft en het liefst in een vorm waarop deze eenvoudig uit het tussenformaat te halen is. Zo zou het bijvoorbeeld niet handig zijn om een diagram als bitmaptekening op te slaan aangezien het terughalen van informatie heel lastig is; enkel de visuele representatie wordt bewaard in een bitmaptekening niet de structuur.
Over het algemeen is het gunstig om een tool te kiezen dat een diagram in een open formaat als *gml*[Grab] of *graphml*[Graa] opslaat.
Graphml is een xml-formaat bedoeld om de structuur van een graaf in nodes en edges op een standaard manier op te slaan zodat grafen uitgewisseld kunnen worden tussen verschillende programma's. Daarnaast kan het formaat worden uitgebreid om (tool-specifieke) eigenschappen van nodes/edges op te slaan; bijvoorbeeld hoe en waar een knoop getekend moet worden.
Het voordeel van een formaat gebaseerd op xml is de goede ondersteuning: het bestand is met behulp van standaardparsers in te lezen (*sax*/*dom*) of te transformeren (*xslt*).
De uitbreidbaarheid van graphml zou als nadeel gezien kunnen worden; tools kunnen het extensie-mechanisme gebruiken om informatie over de knopen zelf -vorm, label, positie et cetera- op hun eigen manier op te slaan. Dit verhoogt de bruikbaarheid maar maakt het uitwisselen lastig wanneer relevante gegevens opgeslagen zijn in een tool-specifieke extensie.
Als graphml wordt gebruikt om een specificatie van een berekening op te slaan zal, wanneer relevante informatie in een extensie wordt opgeslagen, er per tool moeten worden vastgelegd hoe deze relevante gegevens uit het xml-bestand gehaald kunnen worden.
Figuur 6.7 laat een gedeelte zien van het graphml-bestand voor het diagram uit de vorige paragraaf. Informatie over labels en vormen van nodes wordt gedefinieerd in een *yEd*-specifieke extensie (namespace: *y*). Een aantal van de relevante gegevens valt binnen deze extensie en zal dan ook op een *yEd*-specifieke manier uit het graphml-bestand gehaald moeten worden.

3. De relevante gegevens uit het tussenformaat halen

Voordat er een concrete implementatie vanuit een specificatie gemaakt kan worden zullen de relevante gegevens uit het tussenbestand gehaald moeten worden. Allereerst zal het tussenbestand ingelezen moeten worden. Om dit te kunnen doen zal de programmatuur het formaat van het tussenformaat moeten kunnen parsen. Wanneer hiervoor een xml-bestand gebruikt wordt zal dit inlezen geen probleem moeten zijn.
Vervolgens zal de benodigde informatie uit het ingelezen bestand gehaald moeten worden; Hiervoor zal bekend moeten zijn waar de relevante gegevens zich in het bestand bevinden. Dit zal in het geval van een graphml-extensie afhangen van de gebruikte graaf-teken-tool.
De tool-afhankelijkheid van een tussenformaat kan lastig zijn wanneer de vertaling van tussenformaat naar implementatie gebeurt in een applicatie die niet eenvoudig aan te passen is. Het kan daarom verstandig zijn om een tweede tussenformaat te definiëren -het specificatieformaat- dat gebruikt wordt als invoer voor de code-generator. Dit specificatieformaat hoeft enkel de voor de generator relevante gegevens te bevatten.
Het voordeel van een dergelijke tussenstap is dat de codegenerator niet aangepast hoeft te worden wanneer een tussenformaat veranderd, zolang het specificatieformaat maar gelijk blijft. Een tool specifiek voor het (visueel) specificeren van een berekening zou een model direct in het specificatieformaat op kunnen slaan. Het is verstandig om een specificatieformaat te kiezen wat eenvoudig te interpreteren is door de codegenerator, een xml-formaat is daarvoor in veel gevallen erg geschikt.
Het graphml-bestand in *Figuur 6.7* kan vrij eenvoudig vertaald worden naar het volgende specificatieformaat:

```
<berekeningsModel>
  <invoerNode type="const" id="rente"/>
  <calcNode type="const" id="groeifactor">
    <calc>Groei.calcGroeifactor(rente);</calc>
  </calcNode>
  <invoerNode type="const" id="startkapitaal"/>
  <invoerNode type="var" id="premie"/>
  <calcNode type="var" id="kapitaalBegin">
    <calc_0>startkapitaal;</calc_0>
    <calc_n>kapitaalEind_prev;</calc_n>
  </calcNode>
  <calcNode type="var" access="public" id="kapitaalEind">
    <calc_n>(kapitaalBegin+premie)*groeifactor;</calc_n>
  </calcNode>
  <edge type="normal" from="groeifactor"   to="rente"/>
  <edge type="normal" from="kapitaalBegin" to="startkapitaal"/
      >
  <edge type="prev"   from="kapitaalBegin" to="kapitaalEind"/>
  <edge type="normal" from="kapitaalEind"  to="kapitaalBegin"/
      >
  <edge type="normal" from="kapitaalEind"  to="premie"/>
  <edge type="normal" from="kapitaalEind"  to="groeifactor"/>
</berekeningsModel>
```

De enige non-triviale stappen tijdens deze vertaling zijn:

- Het splitsen van labels in de naam van de termen en de berekening(en).
- Het substitueren van de namen van de termen in de edges.

Edges zijn eigenlijk overbodig aangezien zij ook kunnen worden afgeleid uit de berekeningen. Het expliciet geven van de edges heeft als voordeel dat de afhankelijkheid van een term bekend is zonder de expressie(s) te hoeven ontleden.

4. Op basis van de relevante gegevens code voor de berekening genereren

Het omzetten van het specificatiebestand naar een concrete implementatie van een berekening is de lastigste stap van het proces van grafisch model naar implementatie. De complexiteit van deze stap hangt voor een groot gedeelte af van de structuur van de implementatie. Als gekozen wordt voor een object-georiënteerde implementatie, waarbij elke term als object gerepresenteerd wordt, zal er met andere dingen rekening gehouden moeten worden dan wanneer voor een procedurele benadering gekozen wordt; Bij een *procedurele benadering* is de volgorde waarin de termen berekend worden van groot belang; een term kan pas gebruikt worden binnen een expressie als zijn waarde voor de te berekenen periode al bekend is. Bij een *object-georiënteerde aanpak* is deze volgorde impliciet; als de waarde van een term wordt opgevraagd zal het object er zelf voor zorgen dat zijn waarde (indien onbekend) berekend wordt.. Om dit voor elkaar te krijgen zal er wel moeten worden gezorgd dat het object referenties heeft naar alle andere termen waarvan hij afhankelijk is. Bij beide benaderingen zal er rekening gehouden moeten worden met de aansturing van de berekening en het beschikbaar kunnen stellen van de invoerwaarden.

Vertalen naar object-georiënteerde implementatie

Het vertalen van een specificatie in een object-georienteeerde implementatie kan vrij direct. Elke node in de visuele specificatie zal als object terugkomen in de concrete implementatie. Binnen elk object wordt een lijst bijgehouden met de waarden van de term voor de verschillende perioden in de berekening. Afhankelijkheden tussen termen komen terug als referenties tussen de verschillende objecten. De object-structuur en de structuur van de specificatie zullen er dan ook vrijwel gelijk uit zien.
De belangrijkste stappen in de vertaling tussen specificatie en implementatie zijn:

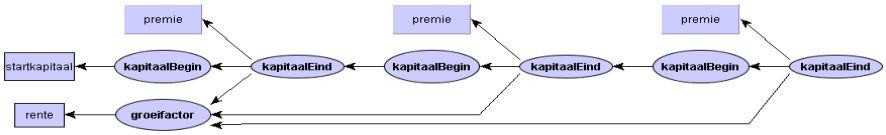
- Er wordt een hoofdklasse gemaakt die instanties van de objecten creëert en beschikbaar stelt zodat objecten elkaar kunnen benaderen.
- Voor elke term wordt een object gedefiniëerd met referenties naar de termen waarvan hij afhankelijk is.
- Binnen elk object wordt een `calculateValue(periode)`-methode gecreëerd waarmee de waarde van een term in een periode berekend kan worden wanneer deze met getValue(periode) opgevraagd wordt en nog niet bekend is. De `calculateValue(periode)`-methode wordt gedefinieerd op basis van de gespecificeerde expressie. In de gegeven expressie worden verwijzingen naar andere termen vervangen door een aanroep van de `getValue(periode)`

voor het betreffende object; `getValue(periode-1)` wanneer het om de waarde in de vorige periode gaat.

Om de invoerwaarde-objecten te vullen met de juiste waarden zullen deze waarden vanuit de aanroepende klasse in de objecten moeten worden gezet. Deze aanroep kan niet volledig gegenereerd worden aangezien van te voren niet bekend is waar deze invoerwaarden vandaan komen. De `setInputValues()`-methode zal dan ook later moeten worden toegevoegd door de programmeur.
In *Figuur 6.8* is een concrete implementatie te zien van de voorbeeld-berekening welke op basis van een specificatie eenvoudig gegenereerd zou kunnen worden. De logica voor het updaten van variabelen in de objecten is generiek geïmplementeerd in de basisklassen `OPConstInputValue`, `OPVarInputValue`, `OPConstCalcValue` en `OPVarCalcValue` waarvan de verschillende term-objecten afstammen [Sni04b].
In de implementatie worden `initialize`-methoden gedefinieerd om de structuur expliciet op te bouwen. In feite zijn deze in de gekozen implementatie niet nodig (`createStructure()` is ook niet nodig); anonieme klassen kunnen sowieso bij de variabelen van de klasse waarin zij gedefinieerd worden; het expliciet opbouwen van de structuur is dan ook niet nodig. Als een externe klasse voor elke object gegenereerd wordt is deze losse initialisatie-stap wel noodzakelijk.

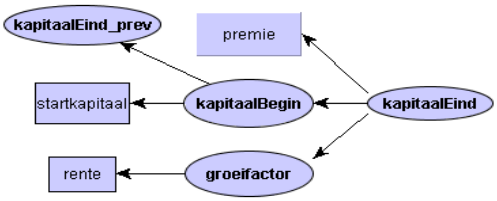
Vertalen naar procedurele implementatie

De vertaling van de term-gebaseerde specificatie van een berekening naar een concrete procedurele implementatie van deze berekening is minder direct dan naar een object-georiënteerde benadering. In plaats van het gedrag per term wordt bij een procedurele benadering gekeken wat er per periode met de waarden, in het bijzonder het eindresultaat, gebeurt.
De volgorde waarin tussenberekeningen worden uitgevoerd is van groot belang bij een procedurele berekening. Deze volgorde kan bepaald worden door de termen op basis van de onderlinge afhankelijkheden topologisch te sorteren. Het diagram in *Figuur 6.5* geeft een topologische sortering weer van het specificatievoorbeeld.



Figuur 6.5: Topologische sortering kapitaalberekening

In het diagram in *Figuur 6.6* is deze sortering per periode gegeven. Het eindkapitaal van de vorige periode (`kapitaalEind_prev`) is hierin als losse waarde meegenomen.
De afleiding van deze laatste sortering uit de verbanden binnen de specificatie is triviaal.
Uit deze sortering is op te maken dat het eindkapitaal pas bepaald kan worden als het beginkapitaal, de groeifactor en de premie bekend zijn. Dit betekent dat de berekening van het eindkapitaal pas als laatste kan plaatsvinden.



Figuur 6.6: Topologische sortering kapitaalberekening per periode

De concrete procedurele implementatie van de berekening gaat op basis van een *recursieve aanpak*[Sni04b]. Bij deze implementatie zijn er in principe twee verschillende klassen: een hoofdklasse waarvandaan de berekening wordt opgestart en per periode een instantie van een periode-klasse waarin de berekening voor één periode wordt uitgevoerd.
Bij het berekenen van de waarden binnen een periode wordt de vorige periode meegegeven. Constante waarden worden uit deze vorige periode gehaald en gebruikt tijdens de berekeningen binnen de nieuwe periode. In de eerste periode is dit niet mogelijk en zullen de inputwaarden geleverd moeten worden door de aansturende klasse. Inputwaarden waarvan de hoogte per periode kan verschillen zullen elke periode worden opgevraagd via de aansturende klasse.
De belangrijkste stappen van het vertalen van een specificatie in een procedurele implementatie zijn:

- Voor de berekening in de eerste periode wordt in de periode-klasse een methode `calculate0(inputwaarden)` gegenereerd, waarin:
  1. alle constante invoerwaarden worden opgevraagd met een `getVar()`-methode,
  2. alle constante berekeningen worden uitgevoerd op basis van de gegeven expressies,
  3. alle variabele invoerwaarden worden opgevraagd met een `getVar(periode)`-methode, en
  4. alle overige (variabele) berekeningen worden uitgevoerd, in de goede volgorde (bepaald door topologische sortering), op basis van de gegeven expressies.
- Voor de berekening in de overige perioden wordt in de periode-klasse een methode `calculateN(inputwaarden, periode_prev)` gegenereerd, waarin:
  1. alle constante invoerwaarden en berekeningen, die voor alle perioden gelijk zijn, worden gehaald uit het periode-object van de vorige periode,
  2. alle variabele invoerwaarden worden opgevraagd met een `getVar(periode)`-methode in de hoofdklasse, en
  3. alle overige (variabele) berekeningen worden uitgevoerd, in de goede volgorde (bepaald door topologische sortering), op basis van de gegeven expressies. Verwijzingen naar waarden van de vorige periode ("prev") worden uit het object van de vorige periode gehaald.

- Om de waarden van de publieke velden in een periode-object op te kunnen vragen moeten er `get`-methoden in de periode-klasse gegenereerd worden.

Naast de periode-klasse zal er een hoofdklasse gegenereerd moeten worden die de berekening aanstuurd door middel van een eenvoudige lus waarin steeds het vorige periode-object wordt doorgegeven.
Daarnaast zullen in hoofdklasse `get...()`- en `get...(periode)`-methoden aanwezig moeten zijn waarmee de periode-objecten de invoerwaarden kunnen opvragen. Deze kunnen niet gegenereerd worden op basis van de gegeven specificatie aangezien hierin niet wordt aangegeven waar inputwaarden vandaan moeten komen. De programmeur zal deze methoden dan nog moeten implementeren. Door middel van een interface of abstracte hoofdklasse met daarin de benodigde `get`-methodes kan gegarandeerd worden dat een programmeur de juiste methoden er bij programmeert.
In *Figuur 6.9* is een procedurele implementatie te zien die op basis van de voorbeeld-specificatie gegenereerd zou kunnen worden.

Zin van patronen voor (pensioen)berekeningen

Hoewel de absolute grootte van de implementatie van een pensioenberekening een stuk kleiner is dan de patronen die deze berekening beschrijven, zijn deze functionele patronen zeker zinnig. Wanneer deze patronen eenmaal zijn vastgelegd is er, op het moment dat er een nieuw project start waarin pensioenberekeningen geimplementeerd moeten worden, al veel kennis over de structuur van de berekening én de uiteindelijke implementatie aanwezig. Het nieuwe systeem zal dan ook sneller en waarschijnlijk beter ontworpen/ontwikkeld kunnen worden.
Naast een snellere ontwikkeling wordt ook de onderhoudbaarheid vergroot op het moment dat er gebruik gemaakt wordt van vaste patronen; vergelijkbare producten zullen vergelijkbare code bevatten en de betekenis/intentie van code zal dan ook eenvoudiger te achterhalen zijn.
De concrete oplossingen zorgen ervoor dat er naast de kennis over de berekeningen ook implementatie hergebruikt kan worden. Hoewel het hier niet gaat om volledig herbruikbare oplossingen krijgt de ontwikkelaar toch extra steun door de beschreven referentie-oplossingen.
Een implementatie van de grafische specificatie zou er daarnaast voor kunnen zorgen dat (pensioen)berekeningen vanaf een hoger, conceptueel, niveau ontworpen kunnen worden. De structuur en verbanden binnen een berekening kunnen op een inzichtelijke manier worden gevisualiseerd. Op deze manier zou zelfs iemand zonder programmeerkennis de specificatie begrijpoen en hier feedback op geven. Dit zou de ontwerper/ontwikkelaar de mogelijkheid bieden om de opdrachtgever mee te laten helpen in de ontwikkeling van het systeem (zie ook *Sectie Domain-Driven Design*{*85*}).

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This file was written by the JAVA GraphML Library.-->
<graphml xmlns="http://graphml.graphdrawing.org/xmlns/graphml"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns/graphml
http://www.yworks.com/xml/schema/graphml/1.0/ygraphml.xsd"
xmlns:y="http://www.yworks.com/xml/graphml">
  <key id="d0" for="node" yfiles.type="nodegraphics"/>
  <key id="d1" for="edge" yfiles.type="edgegraphics"/>
  <graph id="G" edgedefault="directed">
    <node id="n0">
      <data key="d0" >
        <y:ShapeNode>
          <y:Geometry  x="-33.5" y="56.5" width="49.0" height="29.0"/>
          <y:Fill color="#CCCCFF"  transparent="false"/>
          <y:BorderStyle type="line" width="1.0" color="#000000" />
          <y:NodeLabel x="8.5" y="5" ... >
             rente
          </y:NodeLabel>
          <y:Shape type="rectangle"/>
        </y:ShapeNode>
      </data>
    </node>
    <node id="n1">
      <data key="d0" >
        <y:ShapeNode>
          <y:Geometry  x="-110.0" y="-60.0" width="202.0" height="73.0"/>
          <y:Fill color="#CCCCFF"  transparent="false"/>
          <y:BorderStyle type="line" width="1.0" color="#000000" />
          <y:NodeLabel x="18.0" y="20.0" ... >
             groeifactor
             Groei.calcGroeifactor(rente);
          </y:NodeLabel>
          <y:Shape type="ellipse"/>
        </y:ShapeNode>
      </data>
    </node>
    <node id="n4">
      <data key="d0" >
        <y:ShapeNode>
          <y:Geometry  x="-102.0" y="-181.0" width="248.0" height="79.0"/>
          <y:Fill color="#CCCCFF"  transparent="false"/>
          <y:BorderStyle type="dashed" width="1.0" color="#000000" />
          <y:NodeLabel x="12.0" y="23.0" ... >
             kapitaalEind
             {n}(kapitaalBegin+premie)*groeifactor;
          </y:NodeLabel>
          <y:Shape type="ellipse"/>
        </y:ShapeNode>
      </data>
    </node>
...
    <edge id="e1" source="n4" target="n1">
      <data key="d1" >
        <y:PolyLineEdge>
          <y:Path sx="-31.08" sy="39.5" tx="0.0" ty="-36.5"/>
          <y:LineStyle type="line" width="1.0" color="#000000" />
          <y:Arrows source="none" target="standard"/>
          <y:BendStyle smoothed="false"/>
        </y:PolyLineEdge>
      </data>
    </edge>
...
```

Naam van term

Geeft aan dat *rente* een constante invoerwaarde is

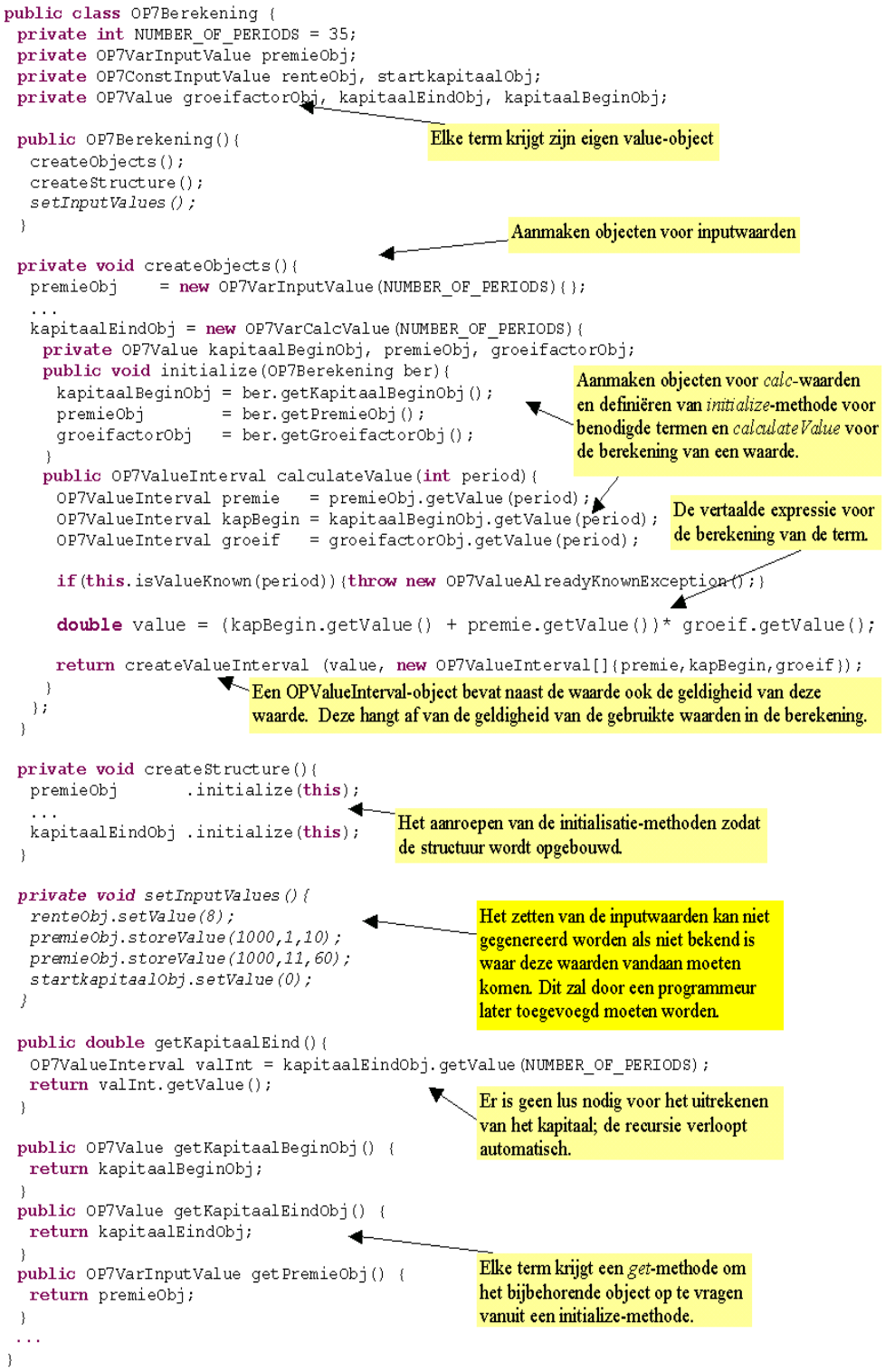
Berekening om eenmalig uit te voeren

Geeft aan dat *groeifactor* een subberekening is

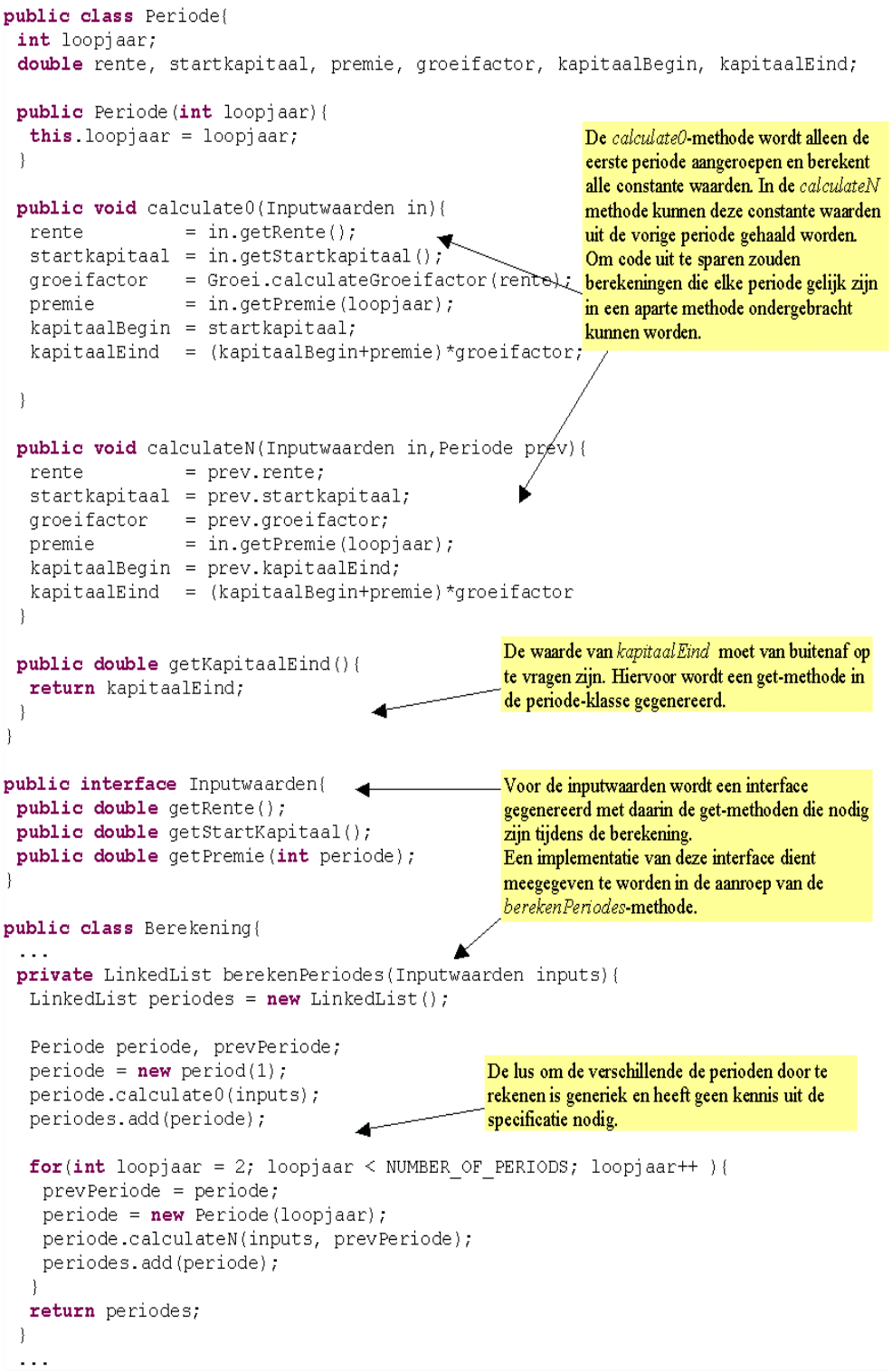
Geeft aan dat *kapitaalEind* opvraagbaar moet zijn

Verbanden tussen nodes op basis van id (niet op label)

Figuur 6.7: Grafische specificatie opgeslagen in *graphml*



Figuur 6.8: Gegenereerde object-georiënteerde implementatie



Figuur 6.9: Gegenereerde procedurele implementatie

Hoofdstuk 7

Conclusie

Design patterns bewijzen al jaren hun waarde bij het ontwikkelen van software. Tot op heden ging het hierbij vooral om *technische* oplossingen die in verschillende systemen worden hergebruikt. Patronen op conceptueel gebied, zoals *analysis patterns*, zijn nooit echt populair geworden. Misschien is het de beschikbaardheid, onbekendheid of het feit dat de kwaliteit/productiviteitwinst minder duidelijk zichtbaar is dan bij technische patronen.
Dat het gebruik van conceptuele patronen wel degelijk kan leiden tot kwaliteit/-productiviteitwinst is iets dat zich al enige jaren bewijst binnen *Quinity B.V.* .
Waar analysis patterns zich vooral richten op de representatie van domeinconcepten binnen software, richten *functional design patterns* zich op de manier waarop functionaliteit in software terugkomt. Functional design patterns vormen hierbij een belangrijke basis voor het hergebruik van functionaliteit binnen systemen.
Functional design patterns werken vanaf twee kanten: allereerst structureren zij domeinkennis en zorgen er hierdoor voor dat het domein dichter bij de software komt te liggen. Daarnaast vormen zij een goed uitgangspunt voor een concrete oplossing. Deze concrete oplossing moet ervoor zorgen dat functionaliteit eenvoudiger hergebruikt kan worden; de software komt als het ware dichter bij het domein te liggen.
Dit zou ervoor moeten zorgen dat door het gebruik van functional design patterns het gat tussen applicatiedomein en de applicatie makkelijker overbrugbaar is.

Pensioenberekeningen

Dat functional design patterns goed gebruikt kunnen worden voor het herbruikbaar maken van algemene 'low-level' functionaliteit is iets dat zich al bewezen heeft. Binnen dit onderzoek is geprobeerd om aan te tonen dat functional design patterns ook voor domeinspecifiekere 'higher-level' patronen gebruikt kunnen worden.
Berekeningen op het gebied van pensioenen vertonen een complexe structuur die voor een ontwerper/ontwikkelaar, die niet over kennis op dit gebied beschikt, erg lastig te begrijpen is. Door de berekening te structuren in verschillende functionele patronen zou het eenvoudiger moeten worden om deze te begrijpen en te vertalen naar software. Of de functionele patronen voor pensioenberekeningen er daadwerkelijk voor zorgen dat pensioenberekeningen eenvoudiger vertaald kunnen worden naar software, is iets dat zich in de praktijk zal moeten

bewijzen.
Bij de implementatie kan er in ieder geval gebruik gemaakt worden van de concrete oplossing die op basis van de patronen ontworpen is. Deze beschreven concrete oplossing kan als uitgangspunt gebruikt worden voor een applicatie waarin deze patronen terugkomen. Wanneer de beschreven vertaling van grafisch specificatie naar concrete implementatie geïmplementeerd zou worden, is het zelfs mogelijk om vanaf een heel hoog (domein)niveau een berekening te modelleren.

Toekomst

Er is nog veel functionaliteit die in veel applicaties voorkomt en waarvoor het nuttig zou zijn als deze beschreven zou worden in functional design patterns. Wanneer dit gebeurt tijdens een lopend project, op basis van kennis opgedaan in voorgaande projecten, zou de overhead van het beschrijven van de patronen niet al te groot moeten zijn.
Vervolgens kan bij de concrete implementatie van het systeem rekening gehouden worden met deze patronen en gekeken worden in hoeverre een generieke oplossing geboden kan worden.
Een belangrijk onderzoeksgebied is het gebruik van technieken waarmee het mogelijk is om vanuit een hoger abstractieniveau software te onwerpen. Het feitelijke ontwerp van de software komt hierbij dichter bij het domein te liggen, waardoor een ontwerper zich meer kan focussen op de functionaliteit van het systeem dan op de techniek hierachter.
Conceptuele patronen kunnen als basis dienen bij het ontwerp van een (eventueel grafische) domeinspecifieke specificatietaal; zoals ook voor pensioenberekeningen is beschreven.
Zo zou het in de toekomst mogelijk zijn om een specificatietaal voor data te ontwerpers die abstraheert over het feitelijke datamodel om hierin patronen als tijdsafhankelijkheid, mutaties en het tonen en van entiteiten te verwerken.
Onderzoek naar de technieken die hierbij gebruikt kunnen worden is van groot beland; met name voor de vertaling van de specificatie op hoger niveau naar een concreet programma.
Beschikbaarheid
De beschikbaarheid van design patterns is een belangrijk punt: er kunnen nog zoveel nuttige design patterns beschreven worden, wanneer potentiële gebruikers niets afweten van het bestaan van deze patronen zullen zij er ook geen gebruik van kunnen maken.
Patronen zullen dan ook op een bekende (centrale) plaats beschikbaar moeten zijn zodat iedereen die er gebruik wil, en mag, maken dit ook kan doen.
Het hebben van een centrale plaats voor patronen is echter niet voldoende: een gebruiker moet ook het voor hem/haar bruikbare patroon tussen alle andere patronen kunnen vinden. Hierbij is een goede ordening naar categorieën en/of zoekfunctionaliteit essentiëel.
Daarnaast is het verstandig patroonbeschrijvingen in te delen volgens een standaardstructuur (zoals ook in deze scriptie beschreven is). Zo hoeft een gebruiker niet het complete patroon door te lezen voordat hij erachter komt dat het patroon niet bruikbaar is in zijn situatie.

Bijlage A

Model Driven Design

Model-Driven Architecture is een term die in 2001 door de Object Management Group (OMG) geïntroduceerd is als de nieuwe ontwerpstandaard voor software [MM01].
Bij MDA staat een model van het systeem centraal. Het zal hierbij vrijwel altijd gaan om een *grafisch* model, bijvoorbeeld in UML.
Het idee van MDA is om het systeem met een 'platformonafhankelijk' model te specificeren, om vervolgens door middel van transformaties, een platform-specifieke oplossing te genereren. Het ontwerp zelf zou op deze manier niet of nauwelijks aangepast hoeven te worden aan veranderende technieken.

Platform

Een belangrijk begrip binnen MDA is een 'platform'.
Met een *platform* wordt een abstractielaag bedoeld die door een (sub)systeem via interfaces kan worden benaderd. Deze laag biedt een aantal mogelijkheden waarvan de technische details voor het bovengelegen systeem verborgen blijven. Voorbeelden van platforms zijn:

- middleware technologie,
- Corba Component Model,
- Enterprise Java Beans,
- object-georinteerde programmeertalen,
- Java,
- Java Virtual Machine,
- besturingssystemen,
- processor,
- ...

'Platformonafhankelijkheid' is een relatief begrip. Om een systeem op een 'platformonafhankelijke' manier te kunnen beschrijven, zal er toch uitgegaan moeten worden van een bepaalde basisdunctionaliteit die een platform biedt.
Bijvoorbeeld voor een model van een gedistribueerd systeem: het gebruikte platform zal mogelijkheden moeten bieden om met processen op andere computers

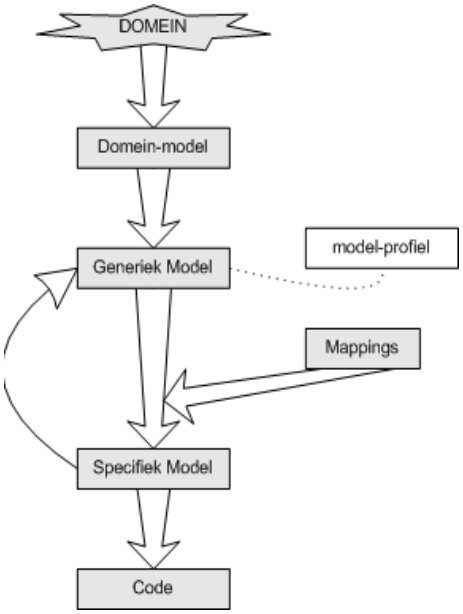
te kunnen communiceren.
Daarnaast kan 'platformonafhankelijk' in de ene context juist 'platformspecifiek' zijn in een andere context en vice versa.
Bijvoorbeeld:
Voor de ontwerper van een virtuele machine is de virtuele machine platformonafhankelijk en zijn besturingssystemen platformspecifiek. Voor de ontwerper van een besturingssysteem voor verschillende machineconfiguraties zullen deze configuraties specifiek zijn terwijl het besturingssysteem platformonafhankelijk is.
De platforms waarvoor een model bedoeld is bepalen het detailniveau van het model.

Viewpoints

Bij model-driven architecture wordt er vanuit drie perspectieven naar een systeem gekeken.
Computation Independent Viewpoint
Het *computation independent viewpoint* kijkt naar het domein van het systeem, op een manier onafhankelijk van de berekeningen en structuur zoals deze uiteindelijk in een informatiesysteem zou komen. Dit beeld wordt vastgelegd in een *computation independent model* (CIM). Dit domein-model zou het gat moeten overbruggen tussen de domein-experts en de software-ontwerpers.
Platform Independent Viewpoint
Het *platform independent viewpoint* kijkt, op een platformonafhankelijke manier, naar de algemene structuur en gedrag van het informatiesysteem. Hierbij wordt gegeneraliseerd over de verschillende platforms die gebruikt kunnen worden en worden technische details voor een specifiek platform weggelaten.
Een veel gebruikte techniek om systemen op een platformonafhankelijke manier te ontwerpen is door gebruik te maken van een virtuele machine welke platformspecifieke eisen verbergt voor de ontwikkelaar.
Zo werkt de Java Runtime Environment als virtuele machine voor Java-programma's en is de werking van een Java-programma onafhankelijk van het besturingssysteem.
Het *platformonafhankelijke model* (PIM) is een formele specificatie van de structuur en functionaliteit/gedrag van het systeem. Dit model kan uitgedrukt worden in domeinspecifieke notatie of een generieke modelleertechniek als UML. Als er gebruik gemaakt wordt van UML zal er, door middel van een profiel, aangegeven moeten worden welke structuren binnen het model mogelijk zijn, hetgeen afhankelijk is van de ondersteunde platforms.
Platform Specific Viewpoint
Het *platform specific viewpoint* vult het platformonafhankelijke beeld van het platform independent viewpoint aan met, technische, details die horen bij het gebruikte platform. Het *platformspecifieke model* (PSM) geeft aan hoe concepten uit het platformonafhankelijke model binnen het gebruikte platform uitgewerkt kunnen worden; dit zou al een concrete implementatie voor het gebruikte platform kunnen zijn.

Mappings

Een essentieel onderdeel van model-driven architecture zijn de transformaties tussen de verschillende modellen, oftewel *mappings*. Mappings worden gebruikt om vanuit een generiek model naar een steeds platformspecifiekere oplossing te gaan (zie *Figuur A.1*).



Figuur A.1: Overzicht van het vertalen van modellen

Om het transformatieproces te sturen kunnen modellen gemarkeerd worden. Deze markeringen kunnen gebruikt worden als parameters voor de mappings.
Bijvoorbeeld:
Een object zou in het platformonafhankelijke model gemarkeerd kunnen worden met `<<Entity>>` om aan te geven dat het hier om een entiteit gaat. Hoe entiteiten vervolgens binnen het platformspecifieke model gerepresenteerd worden, wordt bepaald door de mappings.
Om transformaties mogelijk te maken zullen de beschreven modellen precies genoeg moeten zijn om alle gegevens die nodig zijn binnen de mappings te bevatten.

Toepasbaarheid MDA

Model driven architecture brengt een scheiding aan tussen het platformonafhankelijk definieren van de structuur van een systeem en het verwerken van platformspecifieke technische details. Het idee om met behulp van een visueel model een systeem te specificeren is erg aantrekkelijk. in plaats van het low-level programmeren van functionaliteit.
Op dit moment is de specificatie van MDA, bij OMG, nog niet compleet en zijn er nog geen concrete implementaties van de techniek.

De specificatie van MDA specificeert (nog) niet direct hoe een PIM eruit zou moeten komen te zien, hoe transformatieregels eruit zouden moeten zien et cetera. Ook is het onduidelijk of MDA enkel gebruikt kan worden in combinatie met code/model-generatie; OMG ([MM01, MM03]) vindt van niet, maar andere literatuur spreekt dit tegen:

> *"The MDA process may look suspiciously much like traditional development. However, there is a crucial difference. Traditionally, the transformations from model to model, or from model to code, are done mainly by hand. Many tools can generate some code from a model, but that usually goes no further than the generation of some template code, where most of the work still has to be filled in by hand. In contrast, MDA transformations are always executed by tools."*
>
> [KWB03]

Als alle transformaties automatisch gaan en de laatste transformatie executeerbare code genereerd, zou dit betekenen dat alleen het platformonafhankelijke model door de ontwikkelaar gemaakt hoeft te worden om vervolgens een compleet programma op te leveren. Het modelprofiel en de mappings zouden in feite fungeren als 'programmeer'taal en compiler. Hierbij zou het platformonafhankelijke model als input dienen om uiteindelijk platformspecifieke code te 'compileren'. Dit is erg ambitieus en de vraag is in hoeverre het mogelijk is om een platformonafhankelijk model op een zodanige manier te definiëren dan een ontwikkelaar al zijn specifieke ontwerpkeuzes hierin kan verwerken.
Daarnaast nog een aantal kritiekpunten:

- Het strikt gebruik van MDA is alleen zinnig als er platformonafhankelijkheid vereist is. Het ontwerpen van een platformspecifiek model is in de meeste gevallen eenvoudiger. Daarnaast kan hierbij al meer kennis over de gebruikte technieken verwerkt worden tijdens het ontwerp.
- De OMG beschrijft naast platformonafhankelijke en platformspecifieke modellen ook een soort van domein-model; het CIM. Het CIM vormt een extra scheidingslaag tussen het domein en het model van het systeem. Het gaat in feite uit van de traditionele gefaseerde ontwikkelmethode, de waterval-methode, waarbij domein-analyse en systeemontwerp strikt gescheiden zijn. Echter zal het (domein)model van een systeem vaak wel afhankelijk zijn van de gebruikte technieken.
  Het grote voordeel van MDA is dat software vanaf een hoger abstractieniveau gespecificeerd kan worden. Het platformononafhankelijke model zou hierbij gebruikt kunnen worden in combinatie met de domein-analyse om ervoor te zorgen dat de systeem-specificatie zo dicht mogelijk bij het domein ligt. De voordelen van een hechte relatie tussen ontwerp en domein worden uitvoerig besproken bij Domain-Driven Design [Eva03].
- De beschikbare transformaties bepalen en beperken de mogelijkheden binnen een platform; Voor elke constructie zullen voor elk mogelijk platform mappings gemaakt moeten worden naar het desbetreffende platform.
- Hoe, generieke, oplossingen opnieuw te gebruiken zijn is niet direct duidelijk. Er zou gebruikt gemaakt kunnen worden van markeringen in combinatie met vooraf gedefinieerde templates (patronen). De vraag is echter

hoe dit gedaan kan worden bij patronen die niet eenvoudig geparametriseerd kunnen worden.

- MDA is sterk gebaseerd op Corba en het Corba Component Model hetgeen niet erg populair is aangezien er (nog) geen grote partijen zijn die dit ondersteunen. Op het moment dat ook MDA genegeert wordt door de grote partijen is de kans erg klein dat model-driven architecture ooit een succes wordt. Een voordeel is wel dat er zich inmiddels tools ontwikkelen voor de populaire opensource ontwikkelomgeving Eclipse.
  Bijkomend nadeel is dat bijvoorbeeld Intentional Programming van Microsoft een veelbelovend alternatief zou kunnen zijn; Intentional Programming is nog sterker afhankelijk van tools dan MDA, maar zij heeft het voordeel dat Microsoft met Visual Studio een veelgebruikte ontwikkelomgeving in handen heeft.

Al met al, MDA is een veelbelovende techniek, maar het zal nog enige jaren duren voordat MDA op grote schaal in de praktijk toepasbaar is, als het ooit toepasbaar wordt...
De bruikbaarheid van MDA is voor een heel groot deel afhankelijk van toolondersteuning en beschikbare transformaties. Zolang deze niet beschikbaar zijn zal het commercieel gebruik van MDA nog een te groot risico vormen.
Hoewel het gebruik van MDA, zoals de OMG voorstelt, op dit moment nog niet realistisch is voor complete systemen, zouden de technieken wel gebruikt kunnen worden voor kleinere (sub)problemen binnen een specifiek domein.
Als het domein van het probleem bekend is, is het makkelijker om een strikt model-profiel te definiëren en de mappings die hiervoor nodig zijn. Het model kan concepten binnen het domein op een natuurlijke manier reflecteren zodat ontwerper/ontwikkelaar én domein-expert/opdrachtgever samen een juist model kunnen ontwerpen voor het probleem; zie ook *Sectie Domain-Driven Design*{*85*}.

Bijlage B

Domain-Driven Design

Een van de belangrijkste onderdelen van het ontwikkelproces van een nieuw informatiesysteem zijn het ontwerp van het systeem en de domeinanalyse. Een ontwerp kan nog zo goed zijn geïmplementeerd, als het niet aansluit op de wensen van de eindgebruiker zal het project onherroepelijk falen.
Bij Domain Driven Design ([Eva03]) worden domeinanalyse en ontwerp verenigd zodat opdrachtgever/domeinexpert en ontwerper/ontwikkelaar samen tot een goed ontwerp kunnen komen dat aansluit op het domein én de te gebruiken technieken.

Traditionele technieken

De manier waarop opdrachtgevers of domein-experts betrokken worden bij het ontwikkelproces verschilt per ontwikkelmethode:

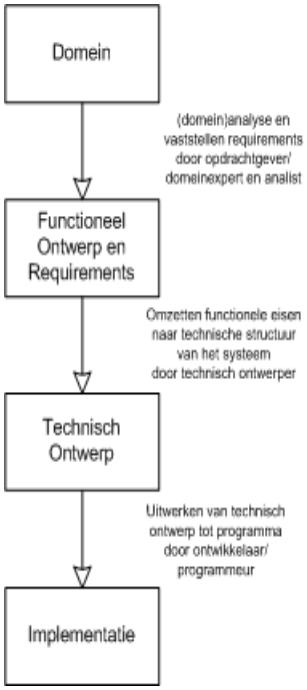
- *Watervalmethode:*
  Bij de watervalmethode worden alle eisen/wensen voor het systeem aan het begin van het ontwikkeltraject, de analyse fase, vastgelegd. De opdrachtgever heeft in principe na deze fase geen controle meer over de uitvoering van het project.
- *Iteratieve methode:*
  Bij de iteratieve ontwikkelmethode wordt het systeem in meerdere stappen ontwikkeld. Het idee is om de applicatie vanuit een basissysteem stapsgewijs uit te breiden naar het uiteindelijke systeem. Elk van deze stappen heeft een analyse, ontwerp, implementatie en test-fase. Als de testfase van een stap is afgerond wordt aan de volgende stap begonnen. De klant zou in principe na elke stap het systeem kunnen testen en zijn wensen voor de volgende stap kenbaar kunnen maken.
- *DSDM:*
  DSDM (Dynamic system development method) beschrijft, naast een projectmanagementmethodiek, een ontwikkelmethode waarbij de scheiding tussen de verschillende stappen minder duidelijk aanwezig is dan bij de iteratieve methode en de watervalmethode. De ontwikkeling volgens DSDM is dynamischer, mede doordat het implementeren en testen tegelijkertijd

plaats vindt. De klant vervult hierbij een actieve rol door tijdens de ontwikkeling al te testen en feedback te geven op het systeem.

Hoewel de rol van opdrachtgever binnen de genoemde ontwikkelmethoden nogal verschilt zullen opmerkingen als de volgende in alle methoden voor komen:

- "het systeem moet ... kunnen"
- "dit bedoelde ik niet *zo*"
- "dit lijkt erop, maar in de praktijk ..."
- "is het zo lastig om dit toe te voegen?"

De gebruikelijke manier om software te ontwikkelen is om wensen van een opdrachtgever, in één of meerdere stappen, te vertalen naar een ontwerp voor de implementatie: een analist zal door vraaggesprekken met de opdrachtgever, gebruiker of domeindeskundige proberen te achterhalen aan welke eisen het systeem moet voldoen. Een ontwerper/ontwikkelaar vertaalt vervolgens deze eisen naar een ontwerp van het systeem of wijziging hierin. De opdrachtgever kan vervolgens bij de implementatie, het eindresultaat en bij tussentijdse evaluaties (iteratieve methode en DSDM) controleren of de gemaakte aannames juist bleken (zie ook *Figuur B.1*).

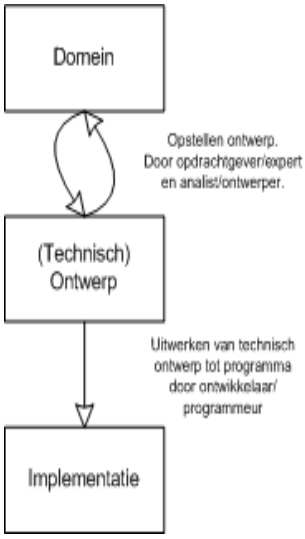


Figuur B.1: Stappen Softwareontwikkeling

Domain-Driven Design

Bij de traditionele methodes wordt de opdrachtgever/domeinexpert niet actief betrokken bij het ontwerp van het techinsche gedeelte van het systeem. Concep-

ten uit het domein worden door de analist en ontwerper vertaald naar concepten binnen het ontwerp van het informatiesysteem, maar niet andersom. De klant zal geen idee hebben hoe de structuur van het systeem eruit ziet. Het is dan ook niet mogelijk om te controleren of het conceptueel model van de ontwerper/programmeur wel overeenkomt met de werkelijkheid.
Bij Domain-driven design wordt er geprobeerd software vanuit het domein te ontwerpen. Op deze manier moet het mogelijk worden om opdrachtgevers, gebruikers en domeinexperts te betrekken in het ontwerpproces: het is de bedoeling dat de klant al tijdens vraaggesprekken met de analist én ontwerper (deze rollen zijn verenigd) feedback kan geven op het ontwerp van het systeem (zie *Figuur B.2*).
Dit kan alleen als de experts/gebruikers begrijpen hoe de ontwerper domeinconcepten in het systeem wil modelleren en de ontwerper begrijpt wat er precies gemodelleerd moet worden.



Figuur B.2: Stappen Softwareontwikkeling met domain-driven design

Ubiquitous Language

Een belangrijke rol bij de communicatie tussen opdrachtgever en ontwikkelaar is weggelegd voor de *ubiquitous language*, oftewel de gemeenschappelijke taal. Deze taal beschrijft wat concepten binnen het domein en het systeem voorstellen en wat hun functie is. Het hebben van een gemeenschappelijke taal heeft een grote voordelen: het voorkomt vertaalfouten en ambibuïteit: iedereen weet precies waarover er gesproken wordt. Uiteindelijk zal het ertoe bijdragen dat iedereen die betrokken is bij het project hetzelfde conceptuele model van het systeem in gedachten heeft.

Model

Om tot een systeem te komen dat het domein op een juiste manier representeert wordt er een model gemaakt. Dit model beschrijft de structuur van het

uiteindelijke systeem en beschrijft wat objecten voorstellen en wat hun functie is. Het wordt ontworpen vanuit het domein én de software. Hierbij worden implementatie-issues op een zo natuurlijk mogelijke manier verwerkt binnen dit model zodat software, domein en model conceptueel met elkaar overeenkomen. Op het moment dat er tijdens het ontwerpen van het model geen rekening gehouden zou worden met de implementatie, dan zou het uiteindelijke systeem onherroepelijk gaan afwijken van het model.
Bij het woord 'model' wordt vaak in eerste instantie gedacht aan een diagram waarin de relaties tussen de verschillende objecten te zien is. Een model is echter meer; een model is een verzameling concepten die bepaalde eigenschappen, gedrag en onderlinge relaties hebben. Hoewel diagrammen erg bruikbaar zijn bij het communiceren over de verbanden tussen objecten zijn ze vaak niet expressief genoeg om ook de betekenis en functie van een object op een duidelijke manier vast te leggen.
Ook al zou het mogelijk zijn om alle details van een model te beschrijven binnen één diagram is het vaak niet verstandig dit te doen. Het grote voordeel van diagrammen is dat de lezer op een snelle manier een idee krijgt van de globale structuur. Door een diagram te 'vervuilen' met details zal het lastiger zijn voor de lezer om onderscheid te maken tussen de algemene structuur en details.
Ondersteunende tekst is dan ook vaak onmisbaar om, naast diagrammen, duidelijk te maken wat de verschillende concepten beschrijven. Het beschrijven en het gebruik van de *ubiquitous language* is hierbij de eerste stap.
Het gebeurt vaak dat een voorgesteld ontwerp en daadwerkelijke implementatie uit elkaar groeien. Het is dan lastig om achteraf te achterhalen wat de overwegingen van de programmeur zijn geweest om het systeem op de gebruikte manier te implementeren.
Om dit te voorkomen is het belangrijk dat veranderingen in begrip van het domein ook doorgevoerd worden in de taal, het model én het systeem. Het doorvoeren van deze veranderingen kan tot gevolg hebben dat complete objectstructuren herschikt moeten worden. Dit zal in eerste instantie veel tijd kosten, maar de voordelen hiervan zijn groot. Het zorgt ervoor dat het besproken model en systeem consistent blijven hetgeen de begrijpelijkheid (en onderhoudbaarheid) van het systeem aanzienlijk bevorderd. Ook is er beter te anticiperen op uitbreidingen van het systeem; het systeem ligt dicht bij het domein zodat uitbreidingen over het algemeen op een natuurlijke wijze geïntregreerd zullen kunnen worden.

Rol van objecten

Het is belangrijk dat de rol van objecten en methoden binnen een systeem duidelijk is. Er zijn een aantal technieken die gebruikt kunnen worden om de rol van objecten en methoden binnen een systeem te verduidelijken:

- *Side-effect-free functies*
  Het is verstandig om zoveel mogelijk gebruik te maken van side-effect-vrije functies. De aanroep van side-effect-vrije functies veranderd de toestand van het systeem niet en retourneert enkel het resultaat van de functie (te vergelijken met wiskundige functies). Het gedrag van een applicatie is hierdoor veel eenvoudiger te voorspellen.

- *Naamgeving*
  Door de intentie van objecten en methoden naar voren te laten komen in de naamgeving ervan is het makkelijker om een systeem te begrijpen. Ook verduidelijkt dit de koppeling tussen domein en systeem.
- *Regels/voorwaarden expliciet maken*
  Voorwaarden van methode-aanroepen zitten vaak diep verborgen in de implementatie. Het grote nadeel hiervan is dat deze in het model niet of nauwelijks te zien zijn terwijl ze van groot belang kunnen zijn op de correctheid van een systeem. Door deze voorwaarden expliciet in het model op te nemen, bijvoorbeeld door middel van specificaties (Specification-pattern [Evan03]) is de correctheid van een systeem makkelijker te verifieren.
- *Goede indeling modules en objecten*
  Verantwoordelijkheden moeten op de juiste manier toegewezen worden aan de verschillende modules en objecten binnen het systeem (er vanuit gaande dat het om een objectgeoriënteerd systeem gaat). Door uit te gaan van *'zwakke koppeling, sterke verbondenheid'* zal er *binnen* modules/objecten een sterke verbondenheid zijn terwijl er *tussen* verschillende modules/-objecten een zwakke koppeling is (interfaces). Bij het samenstellen van objecten/modules moet er rekening gehouden worden met conceptuele én technische verbanden. Als dit niet tegelijk mogelijk is, moeten de conceptuele verbanden voorrang krijgen. Anders zou de relatie met het domein onduidelijk worden.
  Het is niet verstandig om concepten over meerdere lagen te verspreiden: Groepeer functionaliteit/gedrag/waarden van een concept in één object of desnoods één module. Op deze manier komen concepten maar op één plaats terug binnen het systeem.
- *Domeinobjecten*
  In veel projecten is het gebruikelijk om bijvoorbeeld entiteiten te splitsen in meerdere objecten voor verschillende verantwoordelijkheden; eentje voor database opslag, eentje om de waarden vast te houden, eentje voor objectspecifieke functies enzovoort. Hoewel het hebben van deze verschillende objecten vanuit technisch oogpunt te rechtvaardigen is, is het vanuit het domein gekeken niet logisch: de entiteit is één ding en zou ook zo behandeld moeten worden in een applicatie. Het gedrag en de toestand van domeinobjecten moeten dan ook gebundeld zijn.
- *Onderscheid maken tussen verschillende objecten.*
  Door een duidelijke scheiding te maken tussen de verschillende objectsoorten worden de rollen binnen het systeem benadrukt:

  Entiteiten op zichzelf staande objecten waarvan de identiteit belangrijk is. Eigenschappen van een entiteit kunnen veranderen, maar het object behoudt zijn identiteit.
  Bijvoorbeeld persoon of order.

  Waarde objecten objecten die op zichzelf geen identiteit hebben maar die eigenschappen van iets beschrijven.
  Bijvoorbeeld kleur of diersoort.

  Services operaties die niet specifiek tot één ding behoren, maar een functionaliteit voor het systeem leveren.
  Bijvoorbeeld 'geld overschrijf service' of 'verzend notificatie service'

Vooral het gebruik van domeinobjecten is belangrijk. Objecten moeten dezelfde verantwoordelijkheden hebben als de concepten die ze representeren uit het domein. Dit is de enige manier om ervoor te zorgen dat probleemdomein en applicatie met elkaar overeenkomen.

Toepasbaarheid

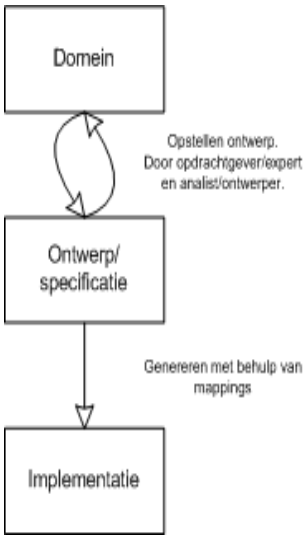
Principes uit domain-driven design zijn in veel situaties toepasbaar om ervoor te zorgen dat een domein op een 'intuitieve' manier in een informatiesysteem verwerkt wordt. Belangrijk hierbij is de integratie van het maken van een technisch ontwerp en de domeinanalyse. Dit zou er toe moeten leiden dat informatiesystemen conceptueel beter overeenkomen met het domein waarvoor ze bedoeld zijn.

Ontwikkelmethodes

Domain-driven design houdt veel mogelijkheden over voor het combineren met bestaande ontwikkelmethodes. Het zou echter niet verstandig zijn om deze methode te combineren met bijvoorbeeld de watervalmethode; om goede feedback te kunnen krijgen over een model kan het verstandig zijn om de opdrachtgever tussentijds resultaten te laten zien van het systeem zoals bij de iteratieve methode en DSDM. Bij de iteratieve methode wordt in duidelijke stappen gewerkt; het omgooien van een al geimplementeerd gedeelte van het model omdat dit beter past bij het domein is dan ook duidelijk niet gewenst bij deze methode. Het zou het stappenplan danig verstoren en het idee van tussentijdse correcte producten is weg.
Het idee van DSDM sluit beter aan op de principes achter domain-driven design. DSDM beschrijft een dynamische ontwikkelmethode waarin de opdrachtgever veel invloed kan uitoefenen. Domain-driven design gaat hierin nog verder en leidt tot ontwikkeling van software vanuit het domein van de opdrachtgever en betrekt daarbij de gebruikers en experts in het ontwerpproces.

Domain-Driven Design en Model-Driven Architecture

Net als bij model-driven architecture staat het model centraal bij domain-driven design. Echter, er is een verschil tussen de betekenis en het gebruik van deze modellen:
Bij MDA is het model -beschreven in een diagram, met eventueel constraints- een specificatie van het systeem. Bij DDD is het model -beschreven met de taal, diagrammen en tekst- een ontwerp van het systeem.
Een complete specificatie is in feite het ultieme ontwerp en de ideale situatie zou dan ook zijn om, met behulp van technieken uit DDD, een model te ontwerpen dat gelijk een complete specificatie van het systeem is. Vervolgens zou deze specificatie, in termen van het domein, gebruikt kunnen worden om een systeem te genereren (zie ook *Figuur B.3*).



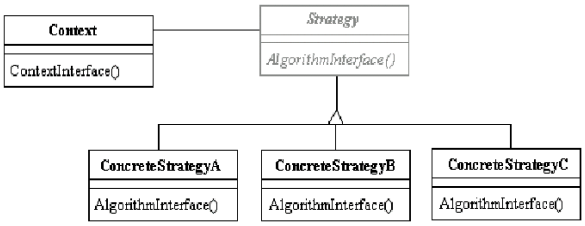
Figuur B.3: domain-driven design i.c.m. MDA

Domain Driven Design en patterns

Conceptuele patronen zoals analysis patterns en functional design patterns zijn zeer geschikt om gebruikt te worden in combinatie met domain-driven design. Conceptuele patronen beschrijven geen hapklare technische oplossingen maar verschaffen inzicht in een probleem binnen een domein; ze beschrijven bewezen structuren en de kennis en overwegingen die tot deze oplossingen hebben geleid. Dat is precies wat nodig is om tot een goed model te komen.
Hoewel het doel van technische patronen in eerste instantie is om oplossingen te bieden voor technische problemen kunnen veel van deze patronen ook op een hoger conceptueel niveau toegepast worden.
Een goed voorbeeld hiervan is het STRATEGY[GHJV95]-patroon:

> *"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."*

zie ook *Figuur B.4*



Figuur B.4: Structuur Strategy-pattern

Op technisch niveau zorgt het strategy patroon ervoor dat alternatieve algoritmes uitwisselbaar zijn (zie klassendiagram hiernaast).
Op conceptueel niveau biedt het strategy patroon een oplossing voor het geval er verschillende manieren zijn om iets af te handelen (strategieën), maar dat deze op een zo transparant mogelijke manier in het systeem verwerkt moeten zijn.
Bijvoorbeeld bij een routeplanner:
Je hebt een algoritme om de kortste route uit te rekenen en eentje voor de snelste route. Het liefst wil je deze berekeningen op dezelfde manier kunnen aanspreken zonder overal in het systeem onderscheid hiertussen te moeten maken.
Niet alle technische patronen zeggen iets over het conceptuele domein; Een interpreter bijvoorbeeld is lastig voor te stellen als een concept binnen een domein. Toch verschaffen veel technische patronen naast hun technische inzicht vaak een conceptueel inzicht dat binnen domain-driven design gebruikt kan worden.
Al met al kan DDD goed gecombineerd worden met patronen. De grootste winst is te bepalen met conceptuele patronen, hiervan zijn er echter jammergenoeg (nog) maar vrij weinig vastgelegd.

Bijlage C

Feature Oriented Programming

Feature-Oriented Programming(FOP) is een programmeerparadigma wat als doel heeft om aparte producten te kunnen specificeren binnen een productlijn [Bat03b]. Dit gebeurt op basis van functionele eigenschappen van een systeem, oftewel *features*.

Features

Het begrip *'feature'* is een begrip dat onder andere wordt gebruikt bij *Feature-Oriented Domain Analysis*(FODA) [KCH+90].
Bij FODA wordt er geanalyseerd welke overeenkomsten en verschillen er zijn tussen systemen binnen een domein. Deze eigenschappen worden *features* genoemd. Definitie features:

> *"Een prominent of onderscheidend en voor de gebruiker zichtbaar aspect, kwaliteit, of karakteristiek van een software systeem of systemen."*
>
> [KCH+90]

Of algemener:

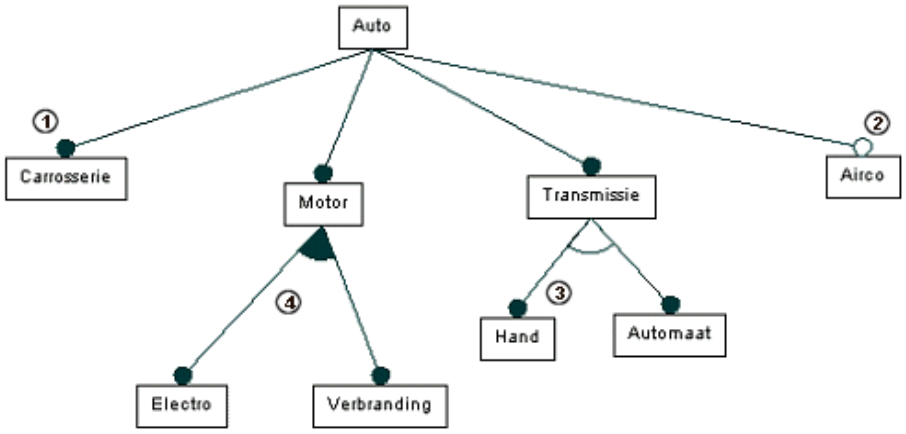
> *"Eigenschap van een domein-concept, welke relevant is voor een belanghebbende van het domein en wordt gebruikt om onderscheid te maken tussen verschillende instanties van een concept"*
>
> [CE00]

Features kunnen worden gebruikt om op een eenduidige manier kennis over het domein, in de vorm van relaties tussen eigenschappen, vast te leggen zodat deze hergebruikt kan worden bij het ontwikkelen van producten.
Het doel van het gebruik van features is uiteindelijk dat een ontwikkelaar aan de hand van een specificatie van gekozen eigenschappen, een compleet product kan

samenstellen. Deze methode van programmeren wordt Feature-Oriented Programming [CE97, Jan02] genoemd. Met behulp van technieken als Generative Programming [CE97, CE00] zou dit er toe moeten leiden dat het specificeren van een *featureset* genoeg is om code voor een (vrijwel) compleet product te kunnen genereren.
Het detail-niveau waarop features beschreven worden is bepalend voor het aantal keuzes voor een product. De vraag of een concept als een losse feature beschouwt moet worden is over het algemeen simpel te beantwoorden door de klant te vragen of hij bereid is ervoor te betalen [Rie03].

Feature Modellen

De relaties tussen verschillende features worden vastgelegd in een featuremodel. In *Figuur C.1* is een featuremodel te zien voor een auto.
De schrijfwijze[KCH+90][1] is vergelijkbaar met de notatie van de relaties tussen verschillende patronen (zie *Sectie Grafische weergave verbanden patronen*{*27*}).



Figuur C.1: Featuremodel van een auto

1. *Verplichte features:*
   Verplichte features zijn features, die als de ouder bestaat ook aanwezig moeten zijn (`REQUIRED`). Zoals: elke 'Auto' moet een 'Carrosserie' hebben.

2. *Optionele features:*
   Optionele features zijn features die aanwezig kunnen zijn. Zoals: een 'Auto' *kan* 'Airco' hebben. (`OPTION`).

3. *Alternatieve features:*
   Alternatieve features zijn twee of meer features waarvan er *één* gekozen *moet* worden (`XOR`). Zoals: de transmissie van een auto is handmatig of automatisch maar niet beide.

[1] alternatieve schrijfwijzen zijn te vinden in [CE00, GFd98, Rie03]

4. *Of-features:*
   Of-features zijn twee of meer features waarvan er *minstens één* gekozen *moet* worden (`OR`). Zoals: de motor van een auto is een verbrandingsmotor, elektrisch of een combinatie van beide (hybride).

Bij het modelleren van features wordt gekeken naar overeenkomstige en variërende structuren/concepten in verschillende producten binnen een bepaald domein. De overeenkomsten worden vastgelegd als verplichte features. De optionele, alternatieve en or-features worden bepaald aan de hand van verschillen tussen de systemen.
De hierarchie waarin de features verschijnen in het featuremodel kan op verschillende manieren opgebouwd worden:

- Er kan vanuit een technisch oogpunt gekeken worden naar de impact die een bepaalde feature-keuze heeft voor de implementatie van het product. Keuzes met een grote impact zullen dan zo hoog mogelijk in de hierarchie geplaatst moeten worden.
- Er kan vanuit gebruikersoogpunt gekeken worden wat 'logische' eigenschappen en subeigenschappen zijn.

Het kan nodig zijn om bepaalde eisen aan combinaties te stellen. Bijvoorbeeld dat een auto met een elektromotor altijd een automatische transmissie moet hebben (*requires*), of dat een auto met airco een elektromotor uitsluit (*excludes*).
Uit het featuremodel en deze regels is een aantal geldige combinaties van features (*featureset*) af te leiden. Elke geldige featureset specificeert een specifiek product. Andersom is het ook mogelijk om aan de hand van het featuremodel en zijn beperkingen te controleren of een featureset wel geldig is.
Er zijn verschillende technieken om featuremodellen en regels te normaliseren, combineren, valideren et cetera [vDK01, CE00].

Features en FDPs

Featuremodellen en functional design patterns hebben een gemeenschappelijk doel: het vullen van het gat tussen de requirements en de oplossing. *Hoe* en *hoe ver* zij dit gat opvullen verschilt echter:
Features zijn pragmatischer dan FDP's en uiteindelijk bedoeld om direct gebruikt te worden bij het assembleren van een product. Featuremodellen worden specifiek voor een productlijn gemaakt en het is in de eerste instantie dan ook niet de bedoeling dat deze kennis vrij wordt hergebruikt door verschillende ontwikkelaars. Functional design patterns zijn algemener dan featuremodellen en bedoeld om algemene (functionele) patronen binnen een domein vast te leggen. Bij functionele patronen wordt er vooral op de overeenkomsten terwijl bij features juist de verschillen belangrijk zijn. Aangezien dit eigenschappen zijn die de producten uniek maken.
Het is mogelijk om FDP's te gebruiken bij het definiëren van featuremodellen. Hierbij is wel applicatiespecifieke kennis nodig omdat er tevens op de verschillen tussen applicaties moet worden ingegaan en niet alleen op de overeenkomsten.
Omgekeerd is het ook mogelijk om op basis van veelvoorkomende features, functionele patronen te definiëren.

Bibliografie

[Ant98] AntiPatterns. The software patterns criteria. http://www.antipatterns.com/whatisapattern/, 1998.

[App00] Brad Appleton. Patterns and software: Essential concepts and terminology, 2000.

[Bat03a] Don Batory. The road to utopia: A future for generative programming. 2003.

[Bat03b] Don Batory. A tutorial on fop and product-lines. 2003.

[BBBR01] Didier Bardo, Dick Berry, Carolyn Bjerke, and Dave Roberts. Crafting the compelling user experience. 2001.

[Bla90] H.M. Blanken. *Informatie-analyse en database ontwerp volgens Information Engineering*. Academic Service, 1990.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons, 1996.

[Bro99] William J Brown. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York: John Wiley & Sons, 1999.

[CE97] Krysztof Czarnecki and Ulrich Eisenecker. Beyond object: Generative programming. 1997.

[CE00] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000.

[Cop96] James Coplien. Software patterns. 1996.

[DSD] DSDM.org. Dynamic systems development method. http://www.dsdm.org/.

[ea96] Kent Beck et al. Industrial experience with design patterns. 1996.

[ea97] Christopher Alexander et al. *A Pattern Language: Towns, Buildings Construction*. Oxford University Press, 1997.

[ea00] Simon Peyton Jones et al. Composing contracts: an adventure in financial engineering. 2000.

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Pub Co, 2003.

[Fow96] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional, 1996.

[GFd98] Martin Griss, John Favaro, and Massimo d'Allesandro. Integrating feature modelling with the rseb. 1998.

[GHJV95] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[Graa] Graphdrawing.org. The graphml file format. http://graphml.graphdrawing.org/.

[Grab] Graphlet. The gml file format. http://www.infosun.fmi.uni-passau.de/Graphlet/GML/.

[HGS02] Michael Hashler and Andreas Geyer-Schulz. Software reuse with analysis patterns. 2002.

[Jac] Jacana. A pattern language. http://www.jacana.org.uk/pattern/.

[Jan02] Anton Jansen. Feature based composition, 2002.

[KCH+90] Kyo Kang, Sholom Cohn, James Hess, William Novak, and Spencer Person. Feature-oriented domain analysis (foda) feasability study. 1990.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. Mda explained: the model driven architecture: Practice and promise. 2003.

[Laa] Sari Laakso. User interface design patterns. http://www.cs.helsinki.fi/u/salaakso/patterns/.

[Lee02] Ken Wing Kuen Lee. An introduction to aspect-oriented programming. 2002.

[MD98] Gerard Meszaros and Jim Doble. A pattern language for pattern writing. 1998.

[MM01] Joaquin Miller and Jishnu Mukerji. Model driven architecture (mda). 2001.

[MM03] Joaquin Miller and Jishnu Mukerji. Mda guide version 1.01. 2003.

[OMG] OMG. Uml resource page. http://www.omg.org/uml/.

[Pre98] Roger Pressman. Can internet-based applications be engineered? 1998.

[Rie03] Matthias Riebisch. Towards a more precise definition of feature models. 2003.

[SA03] Ahmed Seffah and Alina Andreevskaia. Empowering software engineers in human-centered design. 2003.

[Sni04a] Jeroen Snijders. functional design patterns voor pensioenberekeningen. Technical report, Quinity.com, 2004. dit is een vertrouwelijk document van Quinity.com en enkel op aanvraag beschikbaar.

[Sni04b] Jeroen Snijders. Implementatie functional design patterns voor pensioenberekeningen. Technical report, Quinity.com, 2004. dit is een vertrouwelijk document van Quinity.com en enkel op aanvraag beschikbaar.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[Tid] Jenifer Tidwell. Ui patterns and techniques. http://time-tripper.com/uipatterns/.

[vDK01] Arie van Deursen and Paul Klint. Dsl design requires feature description. 2001.

[vW] Martijn van Welie. Patterns in interaction design. http://www.welie.com.

[WM01] Ray Welland and Andrew McDonald. Web engineering in practice. 2001.

[yWo] yWorks. yed - javatm graph editor. http://www.yworks.com/en/products_yed_about.htm.