Longbow
SAMENSTELLING VAN COMPONENTEN IN EEN DEPENDENCY GRAPH

PHILIP VAN OOSTEN

1 juni 2007

Abstract

Longbow is a dynamic model for building software. It consists of a framework that enables to compose many kinds of components in a controlled fashion in a dependency graph. It relates to object oriented programming. Design Patterns can be applied. The interface between the components is important. The data representation in the dependency graph strongly influences that interface. There are well separated user roles. The proposed formal model can be used as a starting point for a new implementation. It refers to the Semantic Web to try to enable automization of software development in the future. Various optimization techniques, such as multiprocessing, can be applied. Longbow can be extended to be a scalable system that can be secured.

Keywords

object orientation, optimization, Semantic Web, software development, Component Composition, dependency graph, JavaBeans, Design Patterns, data processing, formal model

Samenvatting

Longbow is een dynamisch model voor het opbouwen van software. Het bestaat uit een framework waarmee allerlei soorten componenten op een gecontroleerde manier kunnen samengesteld worden in een dependency graph. Het houdt verband met objectgeoriënteerd programmeren. Design Patterns kunnen toegepast worden. De interface tussen de componenten is belangrijk. De voorstelling van de gegevens in de dependency graph beïnvloedt die interface sterk. Er zijn duidelijk afgelijnde gebruikersrollen. Het voorgestelde formeel model kan als vertrekpunt voor een nieuwe implementatie dienen. De link naar het Semantic Web wordt gelegd om te trachten automatisering van softwareontwikkeling in de toekomst mogelijk te maken. Er kunnen diverse optimalisatietechnieken toegepast worden, waaronder multiprocessing. Longbow kan uitgebouwd worden tot een schaalbaar systeem dat kan beveiligd worden.

Sleutelwoorden

objectoriëntatie, optimalisatie, Semantic Web, softwareontwikkeling, Component Composition, dependency graph, JavaBeans, Design Patterns, gegevensverwerking, formeel model

Woord Vooraf

Graag wil ik iedereen bedanken die heeft geholpen deze thesis tot een goed einde te brengen.

Mijn dank gaat uit naar mijn promotors, die mij op weg hielpen. Dirk Vandycke en Wijnand Schepens bedank ik voor hun goede ideeën en hun enthousiasme die zeker hebben bijgedragen tot het eindresultaat.

Voor het lezen en corrigeren van de tekst ben ik een aantal mensen heel erg dankbaar: Liselotte Anckaert, Anne Loddo, Marian Verbeke, Bart van Oosten en Liesbeth Vandepitte. Zij gaven hun vrije tijd op om mij te steunen.

Nogmaals wil ik mijn dank betuigen aan mijn moeder, die mij al die jaren gesteund heeft. Ik dank ook mijn vader, die in mijn gedachten bij me is geweest.

Ook mijn vriendin wil ik nog een keer bedanken voor haar geduld, haar goede zorgen en haar luisterend oor.

Ik richt ook een woord van dank aan Steven Schockaert, voor de gesprekken over mijn thesis en voor het aanbrengen van nieuwe ideeën.

Mijn medestudenten “van de vrijdag” maakten het werk aangenamer dankzij hun gezelschap.

Verder bedank ik mijn vrienden en al de mensen die aan mij dachten, en af en toe vroegen hoe het vlotte met mijn thesis.

Philip van Oosten
Roeselare, 31 mei 2007

Inhoudsopgave

Lijst van figuren

Inleiding

In de industrie worden producten niet alleen *en masse* geproduceerd, ze worden ook efficiënt ontworpen. Bestaande bouwstenen worden terug gebruikt om nieuwe producten te ontwikkelen. Het ontwerp van bestaande producten kan aangepast worden, door de bouwstenen ervan te vervangen, weg te laten of door er nieuwe aan toe te voegen.

Het opbouwen van een ontwerp uit bestaande elementen versnelt de totstandkoming van een nieuw product aanzienlijk. Bovendien kan de kwaliteit van het eindresultaat beter zijn dan bij het van de grond af bedenken van een creatie. De deugdelijkheid van elke component afzonderlijk kan immers al dikwijls bewezen zijn.

De fabricage moet niet op één plaats gebeuren. Verschillende leveranciers kunnen diverse bouwstenen leveren voor nieuwe producten. Veel mensen kunnen samenwerken aan één ontwerp, zelfs totaal onafhankelijk van elkaar.

Een voordeel van een volledig nieuw design, is dat er geen problemen kunnen voorkomen doordat bepaalde componenten niet goed bij elkaar passen. Alles kan ook op één plaats geproduceerd worden, zodat kwaliteitsafspraken tussen verschillende fabrikanten niet nodig zijn. Toch kunnen ook nieuwe ontwerpen bestaan uit componenten, die allemaal door dezelfde fabrikant gemaakt worden.

De software-industrie is een buitenbeentje, omdat de productie zelf dikwijls enkel bestaat uit het beschikbaar stellen van een download op het www. In de ontwerpfase kan er echter nog veel veranderen.

Voor het ontwerpen van software worden er al vaak modules hergebruikt. Daarvoor moeten duidelijke afspraken gemaakt worden. De ontwikkelaar van een component moet de karakteristieken van een component specificeren: de mogelijke raakvlakken tussen de verschillende bouwstenen moeten duidelijk zijn.

Er bestaan raamwerken die bepalen hoe zulke raakvlakken er precies uitzien. Zowel het specificeren van een raakvlak als het herkennen ervan door een programma zijn geen probleem. Met behulp van zo'n omkadering kan een *programmeur* doelbewust componenten samenstellen.

Telkens er een nieuwe toepassing wordt gemaakt op basis van bestaande componenten, is er toch nog programmeerwerk nodig om een vloeiende samenwerking te bekomen. Na compileren wordt een werkende toepassing verkregen.

Om dat programmeerwerk te vermijden kan er een programma ontwikkeld worden dat ervoor zorgt dat de bouwstenen enkel op een correcte manier kunnen samengevoegd worden. De raakvlakken moeten hiervoor duidelijker afgelijnd worden. De componenten moeten eenvoudiger kunnen voorgesteld worden en het aantal mogelijke acties om ze te verbinden beperkt.

Longbow heeft de ambitie een dergelijke samenstelling van componenten mogelijk te maken, voor een beperkte categorie van bouwstenen.

Overzicht van de scriptie

Literatuur en technologie is een overzicht van interessante literatuur en technologie. Dit deel kan los van de rest gelezen worden.

In het tweede deel, *Beschrijvende voorstelling van* **Longbow**, wordt bondig uitgelegd wat ***Longbow*** eigenlijk is. Er wordt een poging gedaan om dit voor een ruim publiek duidelijk te maken. Soms is wat programmeerervaring echter nuttig.

Hoofdstuk 3 geeft een kort overzicht van de belangrijkste concepten van ***Longbow***. Het vierde hoofdstuk vergelijkt het met enkele programma's en programmeermodellen.

Het derde deel, *Ontwikkeling van* **Longbow**, bespreekt hoe het resultaat van dit project tot stand is gekomen. Hoofdstuk 5 behandelt de manier van werken, terwijl het zesde hoofdstuk handelt over enkele van de ideeën die tijdens de ontwikkeling naar boven kwamen.

In de *formele beschrijving* wordt dieper ingegaan op een ontwerp op basis van de verworven ideeën. Op basis van dit deel kan een ontwerp gemaakt worden van een raamwerk voor het maken van software door het samenstellen van kleinere bouwstenen.

Hoofdstuk 7 behandelt enkele begrippen die aan bod komen in de daarop volgende hoofdstukken. Het daarop volgende hoofdstuk beschrijft in formele taal hoe de basis kan gelegd worden voor het samenstellen van componenten in een raamwerk. Het negende hoofdstuk behandelt tenslotte enkele uitbreidings- en toepassingsmogelijkheden, die voortbouwen op het raamwerk uit hoofdstuk 8.

DEEL I

LITERATUUR EN TECHNOLOGIE

Hoofdstuk 1

Literatuurstudie

Alle wijsheid staat in boeken, pleegt men soms te zeggen. Hier volgt een verslag van mijn pogingen om het warm water niet opnieuw uit te vinden.

1.1 Boeken

1.1.1 Head First Design Patterns

De bedoeling van dit boek[1] is *design patterns* "rechtstreeks in het brein van de lezer te pompen". *Head First* boeken zitten speels in elkaar, maar achter de luchtige stijl zit een didactische methode verborgen, die ervoor zorgt dat de lezer echt begrijpt waar het over gaat.

Het boek beschrijft een selectie van *design patterns* en legt klaar en duidelijk uit wat er achter zit en op welke principes de *patterns* steunen.

Een aanrader voor wie nog niet met *design patterns* bekend is, of wie er een oppervlakkige kennis over heeft. Goed om te leren over *design patterns*, maar minder als referentiewerk.

1.1.2 Hardcore Java

Dit boek[2] is niet bedoeld voor beginnende programmeurs, maar eerder voor mensen die al een beetje ervaring met Java hebben. Naar verluidt zouden ook ervaren programmeurs nog het één en ander kunnen bijleren van dit boek.

Het behandelt een selectie van onderwerpen, waaronder een volledige bespreking van het gebruik van het sleutelwoord *final*, *immutable* objecten, reflection, verschillende soorten referenties, ... Er wordt dikwijls diep op de onderwerpen ingegaan.

Het is een goed boek om meer geavanceerde aspecten van het Java Platform te leren kennen. De ideale manier om dit boek te lezen is eerst het boek helemaal lezen en er daarna af en toe een hoofdstuk uitpikken om de stof dieper te doorgronden.

Een inleiding tot Java 1.5 kan men er ook in terugvinden, hoewel die geschreven is nog voordat deze definitief vorm had gekregen. Voor meer recente informatie over Java kan beter andere literatuur geraadpleegd worden.

---

1. [Freeman et al., 2004]
2. [Robert Simmons, 2004]

1.1.3 Effective Java

Dit boek[3] is van de hand van Joshua Bloch, een van de ontwikkelaars van de Java API. Het is onderverdeeld in een aantal items. Elk item bespreekt een bepaald aspect waar een Javaontwikkelaar rekening mee zou moeten houden. In zijn geheel is het een schat aan informatie over allerlei onderwerpen. De items kunnen los van elkaar gelezen worden.

Bloch duidt in dit boek op enkele onvolkomenheden van de Java API. Wie bijvoorbeeld de copy constructor van de klasse `String` gebruikt, heeft dit boek waarschijnlijk niet gelezen.

Effective Java is een onmisbaar referentiewerk voor wie Java API's schrijft, zoniet onmisbaar voor alle Java-programmeurs.

1.1.4 Java Puzzlers

Dit boek[4] is een aangename manier om een expert te worden op het vlak van Java. Je krijgt stukjes code voorgeschoteld die schijnbaar geen probleem opleveren, of waarvan de functie niet duidelijk is. Daarna volgt telkens de uitleg over hoe de problemen kunnen voorkomen worden. Het zijn stuk voor stuk doordenkers.

De meeste vraagstukken staan los van elkaar. Je kan met dit boek dus een Java-expert worden in een kwartier per dag.

1.2 Andere literatuur

Ik heb mij, behalve in boeken, ook verdiept in enkele papers en ik heb een aantal websites bezocht. Het resultaat van deze speurtocht komt ten gepasten tijde ter sprake.

De onderwerpen waarin ik mij op deze manier heb verdiept zijn onder meer *Semantic Web* en het *Easycomp* project.

3. [Bloch, 2001]
4. [Bloch and Gafter, 2005]

Hoofdstuk 2

Technologie

Toen ik begon aan deze thesis wist ik nog niet volledig waar het naartoe zou gaan. Ik wist dus ook niet welke technologie ik zou nodig hebben. Ik heb heel wat uren gespendeerd aan opzoekwerk over welke software of programma's ik zou kunnen (her)gebruiken om efficiënter te kunnen werken. Het meeste van wat ik op mijn zoektocht ben tegengekomen, heb ik uiteindelijk niet gebruikt in de laatste versie van de software. Meestal omdat bleek dat de technologie niet voldeed aan mijn verwachtingen.

2.1 JavaBeans, BeanContext en BeanContextServices

JavaBeans is de componententechnologie van Java. Een *JavaBean*, of kortweg *bean* is een klasse met een standaardconstructor. Beans worden verondersteld in een *multithreaded* omgeving te kunnen werken en zijn over het algemeen serialiseerbaar. Serialisatie van beans kan op verschillende manieren gebeuren: met kortetermijnserialisatie wordt de objectserialisatie van Java bedoeld. Langetermijnserialisatie is serialisatie in XML, op basis van de publieke en protected eigenschappen van beans.

Er bestaan vier soorten eigenschappen ('properties') van beans:

simple properties zijn eenvoudige eigenschappen van beans. Ze worden meestal voorgesteld door een publieke *getter* en een publieke *setter*.

bound properties zijn eigenschappen die kunnen wijzigen. Telkens als er een wijziging optreedt, wordt er een `PropertyChangeEvent` gegenereerd. `PropertyChangeListener`s kunnen dus op de hoogte gebracht worden van wijzigingen van een bound property. Een `PropertyChangeListener` kan zich registreren bij een bean om op de hoogte gehouden te worden van wijzigingen van een welbepaalde eigenschap.

constrained properties zijn eigenschappen waarvan de waarde kan wijzigen. `VetoableChangeListeners` kunnen hun veto stellen tegen de 'lopende' wijziging. In dat geval moet die wijziging ongedaan gemaakt worden.

indexed properties zijn eigenschappen die geïndexeerd zijn met een `int`.

Sommige eigenschappen kunnen alleen gelezen of alleen geschreven worden, andere kunnen beide ondergaan. Dit kan bekomen worden door respectievelijk een accessor of een modificator of beide toe te voegen aan de bean.

Publieke methoden van een beanklasse worden gezien als methoden van de JavaBeanscomponent. Die methoden kunnen toegankelijk zijn voor de buitenwereld, bijvoorbeeld voor een scripttaal, als de bean in een componentenframework geplaatst wordt.

Er kunnen *events* gedeclareerd worden voor JavaBeans. Deze worden op dezelfde manier gedeclareerd als *events* in Swing, aangezien swingcomponenten ook beans zijn.

De instantiëring van beans kan op verschillende manieren gebeuren. De klasse `Beans` bevat methoden om instanties van een bean te maken. Daarnaast kunnen beans ook aangemaakt worden door de standaardconstructor of een andere constructor op te roepen, door ze te klonen of te deserialiseren. `Beans` deserialiseert indien mogelijk een instantie van een bean, of gebruikt anders de standaardconstructor. Het is ook mogelijk een aangepaste `ClassLoader` mee te geven aan de methode `Beans.instantiate()`, waardoor beans bijvoorbeeld rechtstreeks van het internet kunnen gehaald worden.

2.1.1 Introspector en BeanInfo

De interfaces van het JavaBeans-framework bevinden zich in het pakket `java.beans`. Dat pakket bevat onder meer de klassen `Introspector` en `BeanInfo`. Aan de hand van een `BeanInfo` kan alle informatie over een object, als zijnde een bean, achterhaald worden. Er kan opgevraagd worden welke *events* ondersteund worden en welke methoden en eigenschappen de bean heeft. `Introspector` heeft een statische methode waarmee een `BeanInfo`-object kan verkregen worden voor een bean.

`Introspector` kan die informatie op twee manieren achterhalen: ofwel doordat er gebruik is gemaakt van *bean design patterns*, ofwel omdat er expliciet een `BeanInfo` is geïmplementeerd voor de bean. *Bean design patterns* zijn afspraken waar beans aan moeten voldoen opdat er dynamisch een `BeanInfo` zou kunnen aangemaakt worden.

JavaBeans zijn niet zomaar klassen met enkele extra afspraken. Ze kunnen samenwerken met andere componentenarchitecturen, zoals *ActiveX* en *CORBA*.

2.1.2 BeanContext

BeanContext is een specificatie die toelaat JavaBeans in een container onder te brengen. Een beancontext is op zich ook een bean en laat dus toe een hiërarchie van componenten samen te stellen. De context zorgt voor allerlei taken, zoals synchronisatie en serialisatie.

2.1.3 BeanContextServices

In sommige gevallen is het nodig dat er binnen een bepaalde context diensten moeten verleend worden aan sommige componenten. Een bean kan bijvoorbeeld gemaakt zijn om bepaalde taken uit te voeren met een databank. Een verbinding met die databank (of een *connection-pool*) kan aan die bean geleverd worden als een dienst. Het volstaat om een instantie van de bean in een andere context onder te brengen, om dezelfde taken uit te voeren met een andere databank.

2.1.4 Meer informatie

Meer informatie is op het web beschikbaar. De JavaBeans en BeanContext specificaties zijn als pdf beschikbaar op de JavaBeans-website[1].

2.2 JUnit

JUnit is een testframework voor Java. Unit tests zijn herbruikbare tests die in principe gemaakt worden vóór hetgeen ermee getest wordt (*test-first programming*). Het is een uitvinding van

1. [JavaBeans, 2007]

Kent Beck, de grondlegger van eXtreme Programming. JUnit is vrij gemakkelijk te leren. IDE's zoals NetBeans en Eclipse hebben er voorzieningen voor. Er bestaan tal van uitbreidingen en specialisaties. De meeste daarvan zijn gratis te verkrijgen op het web.

JUnit is niet alleen handig als tool voor *test-first programming*, maar kan ook een grote hulp zijn voor het localiseren van fouten en het vermijden van dezelfde fouten in de toekomst. Doordat JUnit tests telkens opnieuw kunnen gedraaid worden, kan ermee vermeden worden dat door het oplossen van een fout nieuwe fouten geïntroduceerd worden.

Het is natuurlijk geen wondermiddel. Het is praktisch onmogelijk ervoor te zorgen dat software geen enkele fout bevat. Niettemin is het een handige tool die de kwaliteit van software kan verhogen als het goed gebruikt wordt.

JUnit Pocket Guide[2] is een goede inleiding tot JUnit en eigenlijk meer dan genoeg om ermee aan de slag te kunnen.

2.3 Subversion

Subversion (SVN)[3] is een versiebeheersysteem zoals CVS, dat onder meer toelaat dat meerdere programmeurs samenwerken aan dezelfde broncode. Er zijn verschillende clients beschikbaar. TortoiseSVN[4] is een client voor Windows die geïntegreerd is in de desktop. Kdesvn[5] is ongeveer dezelfde software, maar dan voor KDE. Er is ook een client die geïntegreerd is in NetBeans. Daarvoor moet wel een aparte module gedownload worden. Alle vermelde clients maken gebruik van de standaard SVN-client die werkt op de commandolijn.

Programmeurs die alleen werken kunnen ook voordeel halen uit het gebruik van versiebeheersytemen. Als er een fout gemaakt wordt, heeft men de mogelijkheid om terug te keren naar de laatste correct werkende versie.

Subversion is dus niet alleen interessant voor als er met meerdere mensen aan dezelfde broncode wordt gewerkt, maar is ook nuttig als hulpmiddel voor het maken van incrementele backups.

2.4 JGraph

JGraph[6] is een Swingcomponent waarmee een graaf kan worden weergegeven. Het heeft een vrij ingewikkelde structuur. Quasi alles wat een grafisch programma zou moeten kunnen met een graaf, is ermee mogelijk. Aan één model kunnen meerdere views gekoppeld worden. De verschillende views kunnen verschillende onderdelen van het model grafisch weergeven. Delen van een graaf kunnen gegroepeerd worden, zodat de groep altijd als één geheel beweegt. En zo zijn er nog veel meer snufjes.

De basisversie van JGraph is *open source*. Het kunnen beschikken over de broncode heb ik ervaren als een groot voordeel. Naast een goede handleiding biedt de broncode een uitstekende documentatie. De Javadoc is minder volledig.

Het *Command Pattern* is zodanig sterk verweven met de code van JGraph, dat het onhandelbaar wordt als je JGraph niet als basis van een applicatie, maar enkel als view wil gebruiken. Het

2. [Beck, 2004]
3. [Subversion, 2007]
4. [Tortoise, 2007]
5. [kdesvn, 2007]
6. [JGraph, 2006]

aanpassen van het model vereist kennis over een groot deel van de JGraphimplementatie. De mogelijkheden voor ongedaan maken en een ongedaan gemaakte actie hernemen zijn interessant voor eenvoudige toepassingen op basis van JGraph, maar maken het aanpassen van het model voor meer geavanceerde toepassingen ingewikkeld. De oplossing die door de makers van JGraph wordt voorgesteld, is om – tegen de MVC principes in – niet het model aan te passen, maar via de views knopen en verbindingen toe te voegen, zelfs programmatorisch. Achteraf gezien was dit met de gegeven implementatie van JGraph inderdaad de gemakkelijkste oplossing geweest.

Uiteindelijk vind ik JGraph goede software. Het doet over het algemeen wat het moet doen. Jammer genoeg is het model ervan is te verschillend van dat van ***Longbow*** om ten volle bruikbaar te zijn.

2.5 JFreeChart

JFreeChart[7] is een Java2D-bibliotheek voor het weergeven van grafieken. Er zijn er een heleboel soorten beschikbaar. Voor elke soort zijn er verschillende mogelijke modellen om de data voor te stellen. Voor de meeste toepassingen waarin grafieken moeten worden weergegeven, zal deze bibliotheek voldoen aan de behoeften.

Het gebruik ervan is eenvoudig, evenals de integratie met Swing.

2.6 Java 1.6 Mustang

De nieuwste versie van Java, met de codenaam Mustang, werd eind vorig jaar officieel uitgebracht. Zoals het een nieuwe versie betaamt, zijn de mogelijkheden van het Java Platform een stuk uitgebreid. Sommige *features* die vroeger alleen voor de serverversie van Java beschikbaar waren, zijn nu ook geïntegreerd in de desktopversie. Andere *features* zijn volledig nieuw. Een handig overzicht van de nieuwigheden van Java 1.6 is te lezen in *Java 6 Platform Revealed* [8]. Het is aan te raden eerst meer te weten te komen over de nieuwe *features* van Java 1.5, zoals Generics en Annotations voor je naar dit boek grijpt.

In de nieuwe versie is er aandacht besteed aan de performantie, o.a. van synchronisatie. In enkele fora op het web wordt beweerd dat de performantie in sommige gevallen spectaculair verbeterd is. Objectieve meetresultaten heb ik niet gezien.

Ook is er een nieuwe versie beschikbaar van JDBC. Het huidige rugnummer ervan is 4.0. Een van de verbeteringen is dat het inladen van een driver kan geconfigureerd worden en dus niet meer moet gebeuren in de broncode. Er zijn ook *annotations* toegevoegd, specifiek voor JDBC. Die annotaties maken het mogelijk het schrijven van ellenlange *boilerplate code* te vermijden.

Boilerplate code is broncode die enkel dient om applicatielogica te kunnen toepassen in de context van een bepaald framework of door gebruik te maken van een bepaalde library. Om een SELECT-query uit te voeren in een databank met behulp van JDBC 3.0 moet bijvoorbeeld altijd ongeveer hetzelfde stramien gevolgd worden: een driver inladen, een verbinding maken, de query versturen, het resultaat verkrijgen, uit dat resultaat de gewenste waarde halen en de verbinding sluiten.

Mustang geeft betere ondersteuning voor I/O. Er is bijvoorbeeld een methode om de vrije ruimte op een volume te bepalen. AWT en Swing hebben enkele wijzigingen ondergaan. Er is

7. [JFreeChart, 2007]
8. [Zukowski, 2006]

een nieuwe klasse, SwingWorker, die het makkelijker maakt om een GUI op het resultaat van een berekening in een andere *thread* te laten wachten, zonder onhandelbaar te worden. Er is ook betere ondersteuning voor het maken van *webservices*.

Javaprogramma's hebben nu toegang tot de Java-compiler. Door middel van een eenvoudige methodeaanroep kan een reeks bronbestanden gecompileerd worden, op bijna dezelfde manier als dat gebeurt op de commandolijn.

2.6.1 Scripting API

Er zijn gestandaardiseerde voorzieningen voor het besturen van Java-applicaties met behulp van scripttalen. Een JavaScript-engine wordt standaard meegeleverd met de Mustang JRE. Er kan een hele rits aan scripttalen ondersteund worden, waaronder Ruby en Python. Scripts kunnen gecompileerd worden als dat ondersteund wordt door de bewuste implementatie van de Scripting API. De visie van Sun is dat in de toekomst software uit lossere modules kan bestaan, die op een flexibele manier aan elkaar kunnen gelijmd worden met behulp van een scripttaal.

2.6.2 JavaDB

Bij de JDK wordt nu een relationele databank meegeleverd. JavaDB is volledig geschreven in Java en is gebaseerd op Apache Derby, een project dat oorspronkelijk bij IBM werd ontwikkeld en later werd afgestaan aan de Apache open source community.

2.6.3 Annotaties

Annotations zijn een zeer krachtig instrument en werden ingevoerd in Java 1.5 (Tiger). Ze kunnen tijdens compilatietijd of looptijd doen wat *reflection* (RTTI) tijdens de looptijd kan doen. Ze zorgen ervoor dat een programmeur zich niet moet verdiepen in de interne werking van een framework voordat hij er gebruik van kan maken. Sterker nog: gebruik maken van annotations kan ervoor zorgen dat de interface van een framework nog niet volledig vast moet liggen op het ogenblik dat er al toepassingen voor dat framework gemaakt worden. Aan de hand van annotaties kan *boilerplate code* gegenereerd worden. Hercompileren van geannoteerde broncode kan volstaan om te voldoen aan een gewijzigde interface van een framework.

In Java 1.5 moesten annotaties verwerkt worden met de aparte tool `apt`. Dat programma is nu geïntegreerd in de Java-compiler. Er kan voor het verwerken van annotaties gebruik gemaakt worden van de *Annotation Processing API* en van het *Java Language Model*. Er kunnen nieuwe annotaties gemaakt worden en ze kunnen een concrete betekenis krijgen door middel van de *Annotation Processing API*. Hier en daar wordt beweerd dat een "gewone" programmeur nooit nieuwe annotaties zou moeten schrijven. Ik denk dat wie een framework schrijft, meestal voordeel kan halen uit het declareren van nieuwe annotaties.

Annotaties zorgen ervoor dat er sneller software kan geschreven worden die bovendien minder fouten bevat en van hogere kwaliteit is. Er kan in bepaalde gevallen voor gezorgd worden dat de gegenereerde code performanter is dan ad-hocbroncode die zou moeten geschreven worden als annotaties niet gebruikt werden. Annotaties zorgen voor een soort hergebruik dat nog sterker is dan hergebruik van code.

2.6.4 JAXB 2.0

Op het gebied van XML is de API een stuk uitgebreid in Mustang. Een van de nieuwe bibliotheken is JAXB 2.0, die voor *marshalling* en *unmarshalling* zorgt.

De tweede versie van JAXB werkt met annotaties, wat als gevolg heeft dat het makkelijker is om mee te werken. Vergeleken met de eerste versie moet er een pak minder code geschreven worden.

Voor een uitgebreide, praktische inleiding tot JAXB en meer algemeen tot de combinatie van Java en XML verwijs ik naar *Java & XML*[9]. Het is hierbij aan te raden te kiezen voor de meest recente uitgave.

2.7 Jakarta

Jakarta is de Java-afdeling van de Apache-community. Jakarta Commons[10] zijn een aantal softwarebibliotheken die als basis dienen voor andere Jakarta-projecten. Commons Collections zijn collecties die volgens veel ontwikkelaars bij de Java Development Kit van Sun Microsystems hadden mogen horen.

Jakarta Commons maakt geen gebruik van de nieuwe *features*[11] van Java 1.5, dus ook niet van Generics. Nochtans zijn deze een belangrijke verbetering aan Java die toelaten sommige fouten door de compiler te laten ontdekken die anders slechts tijdens het uitvoeren aan het licht zouden komen. Door gebruik te maken van Generics heb ik op bepaalde punten belangrijke inzichten verworven. Het ontbreken ervan in Jakarta Commons was een reden om hier geen gebruik van te maken.

Een tweede reden waarom ik deze bibliotheek niet heb gebruikt is omdat ik ze eigenlijk niet echt nodig heb gehad. Ik heb zelf enkele klassen geïmplementeerd die ook wel uit Commons Collections had kunnen halen, maar ik vond het wat overdreven om voor die enkele losstaande klassen mijn software afhankelijk te maken van een externe bibliotheek.

2.8 Xfig en Jfig

Om digitale grafische voorstellingen te maken van mijn denkbeelden over wat ik aan het ontwerpen was, dacht ik een eenvoudig tekenprogramma nodig te hebben. Het tekenprogramma dat ik in gedachten had zou enkele dingen moeten kunnen:

- Werken met vectorafbeeldingen
- Eenvoudige tekeningen maken met eenvoudige vormen zoals lijnen, rechthoeken en ellipsen
- Combinaties van eenvoudige vormen opslaan als klasse in een bibliotheek
- Objecten, instanties van klassen uit de bibliotheek kunnen hergebruiken in dezelfde en andere tekeningen
- Objecten zouden ook andere objecten moeten kunnen bevatten
- Objecten zouden met pijlen moeten kunnen verbonden worden, zodanig dat de verbindingen meegaan met de objecten, als die verplaatst worden

Met zo'n tekenprogramma is het mogelijk om relatief snel nieuwe concepten grafisch voor te stellen. Vooral voor wat ik moest ontwerpen zou een programma waar ik snel tekeningen mee kon maken ideaal geweest zijn. Een tekening zegt meer dan duizend woorden en één snel gemaakte vectorafbeelding is waardevoller dan een heleboel met potlood getekende prentjes.

9. [McLaughlin and Edelson, 2006]
10. [Jakarta Commons, 2007]
11. [Flanagan and McLaughlin, 2004]

Verschrikkelijk moeilijk kan het niet zijn om een dergelijk programma te maken. Hedendaagse tekenprogramma's kunnen meestal veel meer dan wat ik hier beschreven heb. Toch heb ik lang tevergeefs gezocht naar een programma dat voldeed aan mijn behoeften.

Ik dacht dat Xfig[12] zou kunnen wat ik wou, later kwam ik Jfig[13] tegen op een zoektocht naar een gelijkaardig programma. Jfig is een kloon van Xfig, geïmplementeerd in Java. Xfig en Jfig crashten jammer genoeg beiden herhaaldelijk binnen enkele seconden op mijn pc.

2.9 Scalable Vector Graphics

De volgende denkpiste voor het maken van tekeningen was nagaan hoe SVG in elkaar zat. SVG is een W3C-aanbeveling[14] voor het weergeven en doorsturen van tweedimensionale statische en bewegende afbeeldingen. Het is een XML-taal waarin scripttalen kunnen ingebed worden en waarin hyperlinks kunnen gedefinieerd worden met behulp van XLink en XPointer. Interessant zijn ook de *Java Language Bindings*, waardoor Java kan gebruikt worden in plaats van een scripttaal.

Een SVG-document heeft een een bibliotheek. Ik heb echter geen (gratis) programma gevonden dat daar gebruik van maakt op de manier die ik hierboven heb beschreven.

Een minpunt van SVG, zoals ik het heb ervaren, is dat er geen standaard is voor de gebruikte lettertypes. Weergave van tekst is daardoor nog altijd systeemafhankelijk. Een ander nadeel van SVG is dat de specificatie laks omspringt met het vertalen van entiteiten. De bibliotheek en de rest van een SVG-document horen samen. Nochtans zou de mogelijkheid om deze van elkaar los te koppelen voordelen kunnen bieden. Het derde en belangrijkste minpunt van SVG is echter de gebrekkige implementatie in de praktijk. De aanbeveling van het W3C is veelbelovend. Hopelijk blijft het geen dode letter.

De combinatie van XML en vectorafbeeldingen biedt mogelijkheden voor het ontwerpen van een nieuwe desktopinfrastructuur. In principe is het mogelijk om vrij veel van de functionaliteit te bieden van de grafische desktopomgevingen die vandaag bestaan. Daarvoor is enkel een SVG renderer en een netwerkverbinding nodig. Toepassingen als VNC worden in veel gevallen overbodig, als ze worden vervangen door een combinatie van Ajax en SVG. GUI's zouden systeemonafhankelijk kunnen gemaakt worden. Een desktopomgeving die wordt getekend door SVG te renderen biedt veel meer aanpassingsmogelijkheden dan huidige omgevingen.

2.9.1 Inkscape

Inkscape[15] is een betrekkelijk eenvoudig tekenprogramma. Het gebruikt SVG als interne voorstelling. Het werd gebruikt om de tekeningen die deze tekst vergezellen te maken.

2.10 JavaCC

JavaCC[16] is voor Java wat lex en yacc zijn voor C. Het is open source, en wordt gebruikt door een groot aantal script interpreters die in Java zijn geschreven.

12. [xfig, 2007]
13. [Jfig, 2006]
14. [Scalable Vector Graphics (SVG) 1.1 Specification, 2003]
15. [Inkscape, 2007]
16. [JavaCC, 2006]

2.11 Omniscient Debugger

Omniscient debugging[17] kan letterlijk (vrij) vertaald worden als "alleswetend ongedierte verwijderen". Een betere verwoording zou eigenlijk zijn "alles onthoudend fouten opsporen". Een *omniscient debugger* voert een programma uit zoals een debugger dat gewoonlijk doet, met dat verschil dat de loop van het programma wordt opgeslagen in een boomstructuur.

De *omniscient debugger* voert het programma dat gedebugd wordt uit. Tijdens de uitvoering wordt de structuur van het programma onthouden. Een alleswetende debugger houdt dus niet alleen de huidige stack bij, maar slaat die telkens ook op, waardoor een boomstructuur ontstaat.

Breakpoints worden hier niet gebruikt, want ze zijn niet nodig. Nadat de loop van een programma is opgenomen, kan naar gelijk welk ogenblik dat het programma liep, teruggekeerd worden. Er kan ook in alle *threads* gekeken worden wat er gebeurde. Als er dus op een gegeven ogenblik iets fout gelopen is in het programma, kan er teruggekeerd worden, om de oorzaak van de fout te achterhalen.

Bij een debugger die met breakpoints werkt is dat niet het geval. Opdat een fout zichtbaar wordt op de plaats waar hij wordt veroorzaakt, moet er al geluk in het spel zijn. Er moet als het ware gegokt worden waar breakpoints moeten gezet worden om de oorzaak van de fout te vinden. Voor ingewikkelde applicaties kan dat een tijdrovende klus zijn. Dagen na elkaar debuggen is zelfs voor doorgewinterde programmeurs geen grote uitzondering.

De ontwikkelaar van de *omniscient debugger* heeft een lovenswaardig product gemaakt. De eerder povere aanblik van de GUI maakt wel duidelijk dat het bovenstaande product bestemd is om gebruikt te worden door softwareontwikkelaars die weten dat je geen 3D-effecten nodig hebt om goede software te schrijven.

De ontwikkelaar van deze beweert dat hij tegenwoordig veel fouten maakt door zo snel mogelijk code te schrijven omdat hij plezier schept in het debuggen van zijn programma's. Deze ervaring heb ik met hem nog niet kunnen delen, omdat de *omniscient debugger* niet werkt als er nieuwe *features* van Java 1.5 gebruikt worden. Niettemin is dit een interessante technologie, waar zeker toekomst in zit.[18]

2.12 Kwaliteitscontrole: PMD en FindBugs

PMD[19] en FindBugs[20] zijn programma's die een softwareontwikkelaar kunnen helpen fouten of onregelmatigheden op te sporen. Beide systemen zijn gelijkaardig in die zin dat ze software controleren aan de hand van een aantal door de programmeur gekozen regels. Als er problemen vastgesteld worden, wordt de programmeur daarvan op de hoogte gesteld.

2.12.1 PMD

Achter de naam van deze tool moet niet veel gezocht worden. De makers hebben gewoon gezocht naar een combinatie van drie letters die goed klinkt.

---

17. [Omniscient Debugger, 2007]
18. Op het moment van dit schrijven is er reeds een nieuwe versie van de *Omniscient Debugger* uitgekomen, die compatibel is met Java 1.5. Volgens de ontwikkelaar is deze versie nagelnieuw en nog niet grondig getest.
19. [PMD, 2007]
20. [FindBugs, 2007]

PMD zoekt in een verzameling opgegeven bronbestanden naar patronen die als een fout herkend worden. De bedoeling is ervoor te zorgen dat de code voldoet aan algemeen aanvaarde standaarden, voor zover die bestaan, waaraan broncode moet voldoen om leesbaar te zijn. Het is een geautomatiseerde visuele inspectie van de broncode. Nieuwe regels maken voor PMD is volgens de ontwikkelaars vrij eenvoudig.

De keuze van een goede set regels is belangrijk, want dezelfde eisen kunnen niet aan alle software gesteld worden. Bovendien zijn er regels die elkaar tegenspreken, zodat niet alle regels *kunnen* toegepast worden. Met PMD worden er bijvoorbeeld twee regels meegeleverd, waarvan er een stelt dat een klasse minstens één constructor heeft. Een andere regel eist dat een standaardconstructor niet expliciet mag geschreven worden.

PMD heeft meermaals mijn aandacht gevestigd op problemen. In dat opzicht heeft het zeker zijn doel niet gemist.

2.12.2 FindBugs

In tegenstelling tot PMD, laat de naam van deze tool weinig tot de verbeelding over. FindBugs doorzoekt niet de broncode, maar de bytecode naar problemen. De bugs worden overzichtelijk weergegeven. De gebruiker kan kiezen tussen verschillende indelingen: per pakket, per foutcategorie, ...

Sommige van de regels zijn verouderd, wat vrij vervelend kan zijn als je dat niet weet. *Double-checked locking* is bijvoorbeeld gegarandeerd *thread-safe* sinds Java 1.5, omdat het sleutelwoord `volatile` sinds dan duidelijker gespecificeerd is. Toch herkent FindBugs *double-checked locks* als een fout.

Omzichtige configuratie is een must. De gebruiker van FindBugs dient ervoor te zorgen dat de 'juiste' bugs gezocht worden. Het kan gebeuren dat er geen enkele fout gevonden wordt als de configuratie niet afgesteld is op de problemen die op dat moment moeten opgespoord worden.

Over het algemeen ben ik tevreden over de tool. FindBugs heeft me toegelaten sommige problemen vroegtijdig te ontdekken.

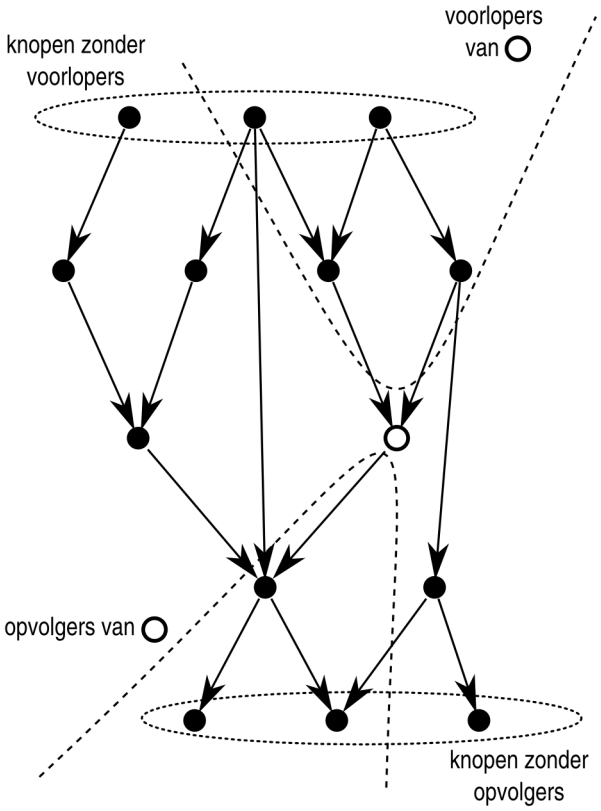
DEEL II

BESCHRIJVENDE VOORSTELLING VAN *Longbow*

Hoofdstuk 3

Longbow in vogelvlucht

Dit hoofdstuk is een snelle inleiding tot ***Longbow***. Er wordt een idee gegeven van verschillende concepten die in deze tekst gebruikt worden. Een meer accurate beschrijving wordt in deel IV gegeven.



Figuur 3.1: Voorbeeld van een *dependency graph*

Longbow is software voor het samenstellen van kleine stukjes software tot een groter geheel. De basisstructuur ervan is een *dependency graph* (zie figuur 3.1). Er zijn twee soorten knopen: *transformaties* en *dataknopen*.

Transformaties zijn knopen die kunnen uitgevoerd worden. Ze kunnen ingangen en uitgangen hebben. Als ze uitgevoerd worden, halen ze meestal eerst data van een ingang op, dan verwerken ze die data en daarna sturen ze de verwerkte gegevens naar de uitgangen. Transformaties worden in de tekeningen voorgesteld als ellipsen. Soms wordt er in de ellips vermeld waar ze voor dienen.

Dataknopen zijn knopen die data kunnen bevatten. Samen met de transformaties vormen ze de knopen van het*graafmodel*. Verbindingen kunnen gevormd worden van uitgangen van transformaties naar dataknopen, en van dataknopen naar ingangen.

Ingangen en uitgangen zijn in de tekeningen die deze tekst vergezellen meestal niet weergegeven. Telkens een verbinding wordt weergegeven, wordt verondersteld dat die verbonden is met een ingang of een uitgang. Verbindingen kunnen alleen gemaakt worden tussen dataknopen en ingangen of uitgangen die met hetzelfde 'soort' gegevens overweg kunnen.

Om ten volle gebruik te maken van ***Longbow***, moeten er transformaties bijgemaakt worden. Daarvoor kan een programmeur die over de broncode beschikt een afgeleide klasse maken van `AbstractTransformatie`. Een basiskennis Java is vereist.

Mark-and-sweep is de manier waarop de volgorde van het uitvoeren van transformaties bepaald wordt. Het is de bedoeling dat transformaties alleen uitgevoerd worden wanneer een resultaat van die uitvoering ergens anders nodig is. Knopen kunnen geldig of ongeldig zijn. Om geldig te kunnen zijn, moeten alle voorlopers van een knoop ook geldig zijn. Bijgevolg worden alle opvolgers van een knoop ongeldig als die knoop ongeldig wordt.

Programmeurs zijn mensen die nieuwe transformaties maken.

Ontwerpers zijn gebruikers die bestaande transformaties toevoegen aan een graafmodel en verbindingen maken, om zo een *dependency graph* te vormen. Het graafmodel is hetgeen waar alle transformaties inzitten. Ontwerpers werken in *designtime*. Uitvoeren van het graafmodel gebeurt in *runtime*.

Inkapselen van transformaties is het maken van nieuwe transformaties uit reeds bestaande. Dat is met behulp van de GUI die het huidige prototype vergezelt niet mogelijk.

De interface van een transformatie bepaalt met welke dataknopen de ingangen en uitgangen ervan kunnen verbonden worden. De interface bestaat gedeeltelijk uit een beschrijving van de ingangen en uitgangen.

Callbacks treden op wanneer een transformatie ongeldig wordt door het uitvoeren van een van zijn opvolgers. De mogelijkheid dat *callbacks* optreden is onmogelijk te vermijden.

Hoofdstuk 4

Vergelijking met bekende concepten

4.1 Rekenbladen

Rekenbladen of spreadsheets (zoals MS Excel, OO.o Calc, ...) kunnen een mark-and-sweep-strategie toepassen. In een rekenblad kunnen er rechtstreeks waarden ingevuld worden in cellen. Ze kunnen ook berekend worden door formules te gebruiken. De gebruiker moet weinig verschil ondervinden met het feit of in een cel een waarde of een formule is ingevuld.

Stel dat er in een rekenbladprogramma een document geopend is met twee rekenbladen en een grafiek als derde blad. Er is op elk ogenblik slechts een klein deel van beide rekenbladen zichtbaar op het scherm. Het is slechts het zichtbare gedeelte dat correcte gegevens moet bevatten. Stel dat het eerste rekenblad geen formules bevat, enkel rechtstreeks ingevoerde data. Het tweede rekenblad bevat enkel formules die kunnen afhangen van data van het eerste rekenblad en van de uitkomst van andere formules in het tweede rekenblad. De grafiek op het derde rekenblad geeft grafisch iets weer dat gebruik maakt van de gegevens van de eerste twee rekenbladen. Op het ogenblik dat een gebruiker cel X van het eerste rekenblad wijzigt, kunnen de cellen die van X afhankelijk zijn geïnvalideerd worden (*mark*). Op dat ogenblik wordt het resultaat van de formules die van X afhankelijk zijn nog niet aangepast, laat staan dat de grafiek al wordt aangepast. Het enige wat gebeurt is dat alles wat afhankelijk is van X als ongeldig gemarkeerd wordt, inclusief de grafiek. Als de gebruiker de grafiek wil weergeven, worden eerst alle nodige waarden uit het tweede rekenblad herberekend. Daarna wordt de grafiek hertekend. De waarde van de meeste cellen moet niet herberekend worden. Het mark-and-sweepmechanisme zorgt dus voor meer efficiëntie.

Mark-and-sweep is niet de enige gelijkenis tussen ***Longbow*** en spreadsheets. Het achterliggende model voor een spreadsheet zou tot op zekere hoogte een *dependency graph* kunnen zijn. In een rekenblad kunnen geen lussen gemaakt worden. Wanneer geprobeerd wordt om circulaire referenties in te voeren in een rekenblad, zal het programma dat detecteren en de gebruiker daarop attent maken. In dat opzicht zou de tabelvorm van een spreadsheetprogramma als een van de mogelijke tabelweergaven van de graaf kunnen gezien worden. Een cel waarin de waarde rechtstreeks is ingevuld kan in ***Longbow*** voorgesteld worden als een dataknoop zonder voorlopers, een cel met een formule zoals een transformatie met één uitgang. Het aantal ingangen van die transformatie is onbeperkt, want een formule kan afhankelijk zijn van een onbeperkt aantal cellen van meerdere rekenbladen.

Er zijn ook enkele verschillen. Het meest opvallende is natuurlijk dat ***Longbow*** geen gebruik maakt van een tabelvorm om de gegevens voor te stellen. Een tweede verschil is dat er een duidelijk onderscheid wordt gemaakt tussen transformaties en dataknopen. Een cel in een rekenblad heeft één enkel resultaat, terwijl een transformatie meerdere resultaten kan hebben. Bovendien

kunnen er in combinatie met spreadsheets slechts enkele datatypes gebruikt worden. In het huidige prototype van ***Longbow*** kunnen alle Java-types gebruikt worden[1]. In rekenbladen kan een bepaald bereik bepaald worden. In ***Longbow*** worden reeksen gegevens voorgesteld als een tabel of collectie, dus als Java-object.

Meer algemeen is mark-and-sweep een optimalisatietechniek die in heel veel toepassingen zou kunnen gebruikt worden. GUI's zijn een typevoorbeeld.

Oorspronkelijk werd mark-and-sweep uitgevonden door Dijkstra. Meer geavanceerde vormen van het algoritme worden gebruikt in *garbage collectors* in objectgeoriënteerde talen.

4.2 LabVIEW

LabVIEW[2] is een programma dat oorspronkelijk bedoeld was om meetgegevens van experimenten te verwerken. Het bestaat uit een *dependency graph*, waar componenten in de knopen kunnen gezet worden. Qua opzet lijkt het dus goed op ***Longbow***, maar toch zijn er belangrijke verschillen.

Componenten die voor LabVIEW kunnen gebruikt worden, zijn er speciaal voor gemaakt. Er is geen specificatie bekend waarmee iemand zijn eigen componenten kan maken. De bedoeling van ***Longbow*** is het tegenovergestelde: niet een uitgebreide bibliotheek voorzien, maar wel iedereen in staat te stellen zijn eigen transformaties te maken. Voor ***Longbow*** ligt de nadruk op het kunnen samenstellen van een zo algemeen mogelijke vorm van componenten, terwijl er voor het maken van LabVIEWcomponenten intern waarschijnlijk een strikte specificatie bestaat.

Het doel van LabVIEW is het maken van GUI's door grafisch te programmeren. Het doel van ***Longbow*** lijkt daar in eerste instantie sterk op, maar uiteindelijk moet het mogelijk zijn om automatisch componenten samen te stellen, met behulp van artificiële intelligentie. Grafisch programmeren is voor ***Longbow*** eigenlijk maar een stap in de ontwikkeling.

LabVIEW is een systeem dat voor een specifiek doel is ontworpen. Het is vooral interessant om snel een GUI te ontwikkelen waarmee meetresultaten kunnen geëvalueerd worden. Het wordt veel gebruikt, en is waarschijnlijk goed in wat het doet. ***Longbow*** daarentegen heeft de ambitie vrij algemeen bruikbaar te zijn als concept voor het programmeren door samenstelling van componenten. Het kan een schaalbaar, performant en distribueerbaar systeem worden dat de kwaliteit van software in het algemeen kan verhogen. Het is eerder de bedoeling om meer mogelijkheden te bieden om bestaande software te hergebruiken en beter te laten samenwerken, dan om ze te vervangen.

4.3 Procedurale talen

C wordt hier als voorbeeld van een procedurale taal genomen. Een programma ontwikkelen in een procedurale taal komt er op neer dat functies en procedures worden samengesteld. Er is echter een wereld van verschil tussen het samenstellen van procedures en functies en het samenstellen van transformaties en dataknopen.

---

1. inclusief het type dat het graafmodel zelf voorstelt, wat interessante gevolgen heeft
2. [Instruments, 2007]

4.3.1 Stack

Uitvoeren van een C-programma betekent dat er een stack van procedureaanroepen bestaat. De stack bevat alle actieve procedures. Over de volledige tijd dat het programma loopt, kan dit gezien worden als een boom die wordt overlopen. Dat kan altijd met een stack, gezien over een bepaalde tijd. Elke knoop van de boom stelt een actieve kopie van een procedure voor en elke tak een procedureaanroep vanuit de ouder. Zo'n – behalve bij omniscient debugging fictieve – boom vormt de geschienis van de loop van een programma. Als een proceduraal programma in meerdere threads wordt uitgevoerd, kan er voor elke thread zo'n boom opgebouwd worden en de bomen kunnen in elkaar verweven zitten waar de threads met elkaar communiceren.

Longbow wordt uiteindelijk ook als een proceduraal programma uitgevoerd, maar het is interessanter om het op een hoger niveau te bekijken. De transformaties van ***Longbow*** zijn het best te vergelijken met de procedures van een procedurale taal. In ***Longbow*** is vanaf de start van een programman bekend welke instanties van transformaties er bestaan, dit in tegenstelling tot C. Tijdens de looptijd kunnen er geen nieuwe transformaties toegevoegd worden. Dit staat in schril contrast met procedurale talen, waar actieve kopieën van procedures worden gemaakt en op de stapel gelegd op het ogenblik dat ze nodig zijn. In ***Longbow*** is er geen stack.

In procedurale talen wordt een instantie van een procedure aangemaakt en op de stack geplaatst, waarna die procedure actief wordt. Na de beëindiging wordt de instantie van de procedure verwijderd. Transformaties zijn herbruikbaar en kunnen meermaals uitgevoerd worden.

Recursie

Als gevolg van het ontbreken van een stack is recursie niet mogelijk in ***Longbow***. Transformaties worden niet aangemaakt op het ogenblik dat ze nodig zijn, maar vooraf. Bij recursieve algoritmen kan niet op voorhand bepaald worden hoeveel instanties van een bepaalde procedure nodig zullen zijn. Om niettemin een krachtig instrument aan te bieden, dat toch inherent aanwezig is, kunnen er in ***Longbow*** *callbacks* toegepast worden.

4.3.2 Het programmeren

Proceduraal

In C moet een programmeur stap voor stap aan een computer duidelijk maken wat er moet gebeuren. Hij kan daarbij gebruik maken van bibliotheken die door andere programmeurs geschreven zijn. Programmeren in een procedurale taal vergt veel organisatie en technische kennis over de programmeeromgeving. Kennis over de compiler of interpreter is meestal niet strikt nodig, maar kan goed van pas komen. Hoe beter de programmeur op de hoogte is van de interne werking van de taal en van het platform waarop het programma zal uitgevoerd worden, hoe meer aandacht kan besteed worden aan *performance tuning* en aan een betere samenwerking met dat platform. De leercurve voor procedurale talen kan steil zijn voor mensen die er nog niet mee gewerkt hebben.

Scheiding van verantwoordelijkheden

> Where ignorance is bliss, 'Tis folly to be wise.
>
> Thomas Gray (1716-1771)

De verantwoordelijkheid voor het maken van een werkend programma in ***Longbow*** wordt gescheiden tussen twee gebruikersrollen: *programmeurs* en *ontwerpers*. De communicatie tussen mensen die de verschillende rollen vervullen bestaat uit een *blackbox*beschrijving van transformaties.

Programmeurs moeten kennis hebben van een procedurale taal (m.n. Java) om transformaties te kunnen maken voor ***Longbow***. Het voordeel voor programmeurs van het werken met ***Longbow*** is dat ze geen weet moeten hebben van de globale structuur van een applicatie, maar slechts van een beperkt deel ervan. Hun werk wordt makkelijker te overzien, wat hun meer mogelijkheden biedt om de kwaliteit ervan te verhogen.

Longbowprogrammeurs kunnen beter niets weten over het latere gebruik van hun transformaties. Die wetenschap zou hen ertoe kunnen verleiden ongeoorloofde veronderstellingen te maken, die de door hun gemaakte componenten beperken in hun nut. Wat programmeurs moeten doen, is het opvullen van een zwarte doos. In principe werkt modulair ontwerp altijd zo, maar in de praktijk kan het er anders aan toe gaan. Vooraleer een module hergebruikt wordt in een nieuwe omgeving, moet hij getest worden. ***Longbow*** tracht een hulpmiddel te zijn om de noodzaak om opnieuw te testen wanneer een transformatie in een nieuwe omgeving – een nieuw graafmodel – terechtkomt, tot een minimum te beperken.

De belangrijkste manier om dat te verwezenlijken is te eisen dat een programmeur een transformatie ontwerpt zoals de implementatie van een zwarte doos, zonder veronderstellingen te maken over de omgeving waarin de transformatie kan terechtkomen. In het Engels wordt dat op een poëtische manier in drie woorden uitgedrukt als *"Ignorance is bliss"*.

Het is de taak van de programmeurs om ervoor te zorgen dat een transformatie, die door een ontwerper in een graafmodel wordt geplugd, effectief datgene doet waarvoor het gemaakt is in dat graafmodel. Een ontwerper moet kunnen vertrouwen op het werk van de programmeur.

Ontwerpers puzzelen de transformaties die programmeurs gemaakt hebben aan elkaar, zodat ze een werkende applicatie bekomen, die doet wat het moet doen. Bij het samenstellen van een graafmodel, moeten ontwerpers rekening houden met de mogelijkheden en beperkingen van de transformaties, die door de programmeurs zijn opgelegd. ***Longbow*** zorgt ervoor dat ontwerpers daar niet onderuit kunnen. ***Longbow*** stelt bovendien bijkomende voorwaarden, zoals bijvoorbeeld de eis dat de transformaties moeten afgewisseld worden met dataknopen en samen daarmee een *dependency graph* moeten vormen.

Longbow zorgt er steeds voor dat fouten van de ontwerper die de uitvoering van een programma belemmeren relatief gemakkelijk en nauwkeurig door een computerprogramma kunnen opgespoord worden. Dit staat in schril contrast met de vaak voorkomende compilatiefouten in procedurale talen. Compilatiefouten zijn waarschijnlijk het eerste waar een beginnende programmeur mee in aanraking komt in de minuten voordat *"Hello World!"* op zijn scherm verschijnt. De oorzaak ervan ligt vaak niet voor de hand. Opsporen van de oorzaak van de fout kan in procedurale talen vaak alleen door menselijke tussenkomst.

De gemakkelijke opspoorbaarheid van fouten van de ontwerper, is te verklaren door het beperkt aantal vrijheidsgraden van het ontwerpen in ***Longbow***, ten opzichte van het programmeren in een procedurale taal. Door de beperkingen zal het werk van de ontwerper waarschijnlijk als relatief gemakkelijk ervaren worden. Voor de ontwerper in spe gaat het aanleren ook gemakkelijker. De leercurve is heel wat minder steil dan die voor proceduraal programmeren. Als een ontwerper alle nodige transformaties ter beschikking heeft, kan het samenstellen ervan eenvoudig worden.

Wat voor de programmeur geldt, geldt dus op een andere manier ook voor de ontwerper: *ignorance is bliss*.

4.3.3 Performantie

De performantie kan in C soms opgekrikt worden door programma's te schrijven die aangepast zijn voor een bepaald besturingssysteem of hardwareplatform. Het is zelden makkelijk om de performantie op te drijven door gebruik te maken van meerdere processoren, of van meerdere computers.

In ***Longbow*** is aanpassingen maken voor een bepaald hardwareplatform of besturingssysteem onmogelijk. Er kan echter wel gebruik gemaakt worden van *multithreading* en van gedistribueerde verwerking, zonder dat er een programmeur daar speciale maatregelen voor moet treffen.

4.4 Objectgeoriënteerde talen

4.4.1 Objectgeoriënteerde principes

Abstractie

Abstractie is de mogelijkheid te werken met objecten van een specifiekere of minder specifieke vorm naargelang de situatie daarom vraagt.

Een manier om abstractie te maken van een transformatie is om slechts rekening te houden met de verbindingsmogelijkheden. Van twee transformaties met dezelfde soorten ingangen en uitgangen zou kunnen gezegd worden dat ze aan dezelfde interface voldoen. Er kan bovendien ook rekening gehouden worden met eigenschappen van de transformatie zelf.

Inkapseling

Inkapseling is het verbergen van de interne structuur van een klasse Met het doel dat de ingewikkelde structuur van de objecten niet opvalt. Objecten kunnen zodoende als een zwarte doos bekeken worden.

Dit is het principe van objectgeoriënteerd programmeren dat het duidelijkst naar boven komt in ***Longbow***. De inkapseling van een transformatie is veel sterker dan die van een object. De interne structuur van een transformatie kan opgebouwd zijn uit andere transformaties, of geprogrammeerd zijn. De interne structuur is transparant voor de ontwerper.

Overerving

Overerving betekent dat een klasse een meer gespecialiseerde variant kan hebben. Dat is bij transformaties niet het geval. Een transformatie is op zich een klasse, wat betekent dat een subklasse mogelijk is; maar een subklasse van een transformatie kan niet als een specialisatie of uitbreiding van die transformatie beschouwd worden.

Overerving structureert de types van objecten in een boom, of in een *dependency graph* wanneer meervoudige overerving mogelijk is. Een dergelijke structuur is voor bepaalde categorieën van transformaties mogelijk. De achterliggende klassenstructuur komt echter niet noodzakelijk overeen met de structuur van de transformaties.

Polymorfisme

Polymorfisme is de mogelijkheid dat een object om zich in verschillende omstandigheden op een andere manier gedraagt. Dit concept hangt sterk samen met overerving. Een object kan tegelijk

een `Hond`, een `Zoogdier` en een `Dier` zijn. Alle honden kunnen blaffen, terwijl niet alle zoogdieren dat kunnen. Alle zoogdieren kunnen ademen, terwijl niet alle dieren dat kunnen.

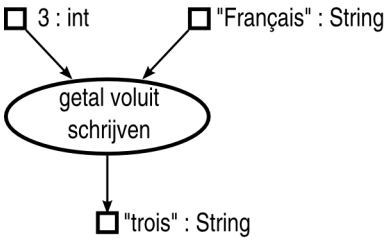
Voor dezelfde categorieën als waarvoor overerving mogelijk is, is polymorfisme dat ook voor transformaties. Er is opnieuw geen direct verband met de klassen die de transformaties voorstellen.

4.4.2 Design patterns

Sommige *design patterns* zijn zonder meer bruikbaar, zij het in een iets gewijzigde vorm als hun objectgeoriënteerde varianten. De definities van de *design patterns* werden overgeschreven uit [Freeman et al., 2004]. Voor meer achtergrondinformatie over *design patterns* wordt verwezen naar de uitgebreide literatuur.

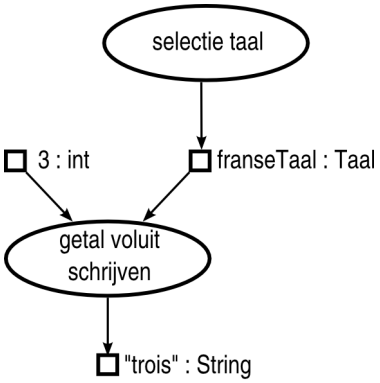
Strategy

> The *Strategy Pattern* defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use them.



Figuur 4.1: Voluit schrijven van een getal in een bepaalde taal zonder *Strategy*

Het gedrag van een transformatie kan bepaald worden door parameters, die door de logica van de transformatie geïnterpreteerd worden. Neem bijvoorbeeld een transformatie die een getal omzet in een `String`, zoals in figuur 4.1, met twee ingangen en een uitgang die dat getal weergeeft in een bepaalde taal. Als de ingang *getal* het geheel getal 4 bevat en de ingang *taal* bevat het woord `Italiano`, dan is het resultaat van de transformatie aan de uitgang de string `quattro`.



Figuur 4.2: Voluit schrijven van een getal in een taal met *Strategy*

Als het *Strategy Pattern* wordt toegepast, kan elke taal als een apart algoritme gebruikt worden, zoal in figuur 4.2. Een taal bepaalt nu zelf hoe een getal in een woord wordt omgezet en is daar-

voor niet meer afhankelijk van de logica in de transformatie. Er kunnen extra talen bijgemaakt worden zonder dat de vertaaltransformatie daarvoor moet veranderen.

Het *Strategy Pattern* kan toegepast worden doordat in ***Longbow*** alle Java-objecten kunnen worden doorgegeven. Qua implementatie is er bijna geen verschil tussen het toepassen van *Strategy* in objectgeoriënteerde talen en in ***Longbow***.

Observer

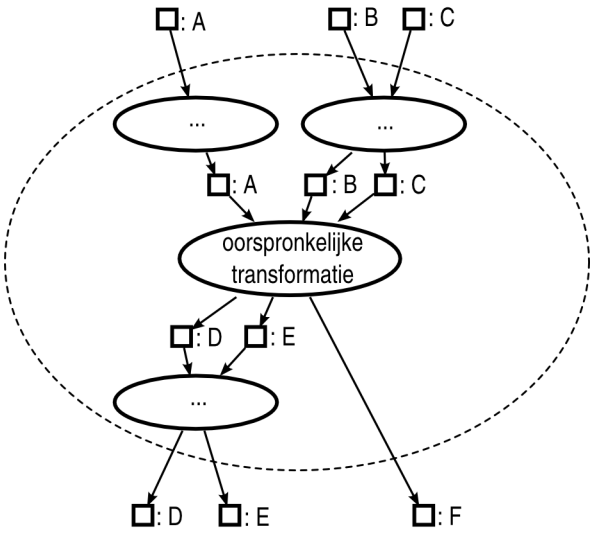
> The *Observer Pattern* defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Het *Observer Pattern* kan gebruikt worden om ervoor te zorgen dat modules minder sterk van elkaar afhankelijk zijn. Een *Observable* object maakt weinig veronderstellingen over zijn *Observers*. Componententechnologieën voorzien over het algemeen een mechanisme om het *Observer Pattern* te implementeren (*events*). Voor JavaBeans gebeurt dat meestal door *bean design patterns* te gebruiken, voor ActiveX componenten meestal door middel van *delegates*.

Er zijn twee manieren om in ***Longbow*** gebruik te maken van het *Observer Pattern*. De gemakkelijkste manier is ervoor te zorgen dat alle te verwittigen knopen opvolgers zijn van de knoop die kan veranderen. Observer zit immers stevig ingebakken in het mark-and-sweepmechanisme van ***Longbow***. Om een voorloper te verwittigen van de wijziging van een opvolger moet een callbackmechanisme worden gebruikt.

Decorator

> The *Decorator Pattern* attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Figuur 4.3: Implementatie van het *Decorator Pattern*

Decorator is makkelijk uit te voeren, door het verbinden van ingangen en uitgangen met transformaties die bijkomende functionaliteit bieden, zoals weergegeven in figuur 4.3. Door het inkapselen van de transformaties kan een ontwerper nieuwe transformaties maken. Zodoende wordt de toepassing van het *Decorator Pattern* voor andere ontwerpers transparant.

Adapter

> The *Adapter Pattern* converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

In Objectgeoriënteerde talen bestaat het Adapter Pattern eigenlijk alleen maar om verschillende modules, die oorspronkelijk niet gemaakt waren om samen te werken, aan elkaar te lijmen. De noodzaak om dit te doen schuilt vooral in het feit dat interfaces kunnen herkend worden doordat ze *geïdentificeerd* en niet *beschreven* worden. ***Longbow*** tracht het beschrijven van een interface, zonder de interface te identificeren, mogelijk te maken. Klassieke objectgerichte talen beschrijven een (virtuele) wereld op een statische manier, terwijl ***Longbow*** een dynamisch model is.

De signatuur van een interface in objectgeoriënteerde talen is zo accuraat mogelijk bepaald als het type van invoervariabelen zo ruim mogelijk en van uitvoervariabelen zo specifiek mogelijk is. Als een interface op die manier bepaald is, dan is er een maximale kans dat de interface bruikbaar is in zoveel mogelijk toepassingen. Meer algemeen is het in ***Longbow*** belangrijk dat een interface zo accuraat mogelijk wordt beschreven. Hoe accurater beschreven is in welke gevallen een transformatie bruikbaar is, hoe meer gevallen er zijn waarin die transformatie ook effectief bruikbaar is. Een nauwkeurige beschrijving van een interface van een transformatie is altijd te verkiezen boven het gebruik van een Adapter.

Controle of een ingang en een uitgang aan elkaar kunnen gekoppeld worden gebeurt niet door een compiler maar door een framework. Dit biedt veel ruimere mogelijkheden voor het specificeren van interfaces. Ook contextgevoelige interfaces worden mogelijk.

Het *Adapter Pattern* moet zoveel mogelijk preventief vermeden worden in ***Longbow***. Het gebruik ervan zal vaak duiden op slecht ontwerp van de interface van componenten. Niettemin blijft het bruikbaar, op bijna dezelfde manier als zijn objectgeoriënteerde variant. Een *Adapter* zal nodig zijn wanneer de interface van een transformatie niet accuraat genoeg is ontworpen.

Factory patterns

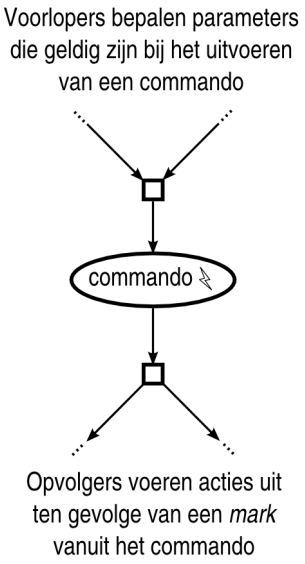
Met *factory patterns* worden het *Simple Factory* paradigma, het *Factory Method Pattern* en het *Abstract Factory Pattern* bedoeld. Het doel van deze paradigma's is ervoor te zorgen dat objecten die voldoen aan een bepaalde interface kunnen gemaakt worden, zonder dat moet bekend zijn op welke manier de objecten worden aangemaakt. M.a.w. het aanmaken van de objecten wordt ingekapseld.

Inkapselen van het aanmaken van transformaties in ***Longbow*** lukt niet. De kern van het probleem om *factory patterns* toe te passen in ***Longbow*** zit in het feit dat transformaties niet worden aangemaakt op het ogenblik dat ze nodig zijn om het graafmodel uit te voeren, maar geselecteerd moeten worden vóór de uitvoering.

Creational design patterns hebben dus over het algemeen geen analoge voorstelling in ***Longbow***. Voor *behavioral design patterns* ligt dat anders. *Creational design patterns* kunnen achter de schermen gebruikt worden, terwijl *behavioral design patterns* duidelijk naar voor komen.

Command Pattern

> The *Command Pattern* encapsulates a request as an object, threby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.



Figuur 4.4: Implementatie van het *Command Pattern*

Het implementeren van het *Command Pattern* in ***Longbow*** heeft meer invloed op de structuur van het graafmodel dan andere *design patterns*. Een *Command object* kan gezien worden als een transformatie in het graafmodel, die reageert op een bericht van buitenaf. Het bijhouden van de volgorde van uitvoeren van de commando's kan als een dienst van het graafmodel aan de commando's beschouwd worden. Het onthouden van de volgorde is belangrijk voor het ongedaan maken of loggen van acties. Figuur 4.4 geeft de implementatie van een commando ***Longbow*** weer.

Facade

> The *Facade Pattern* provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

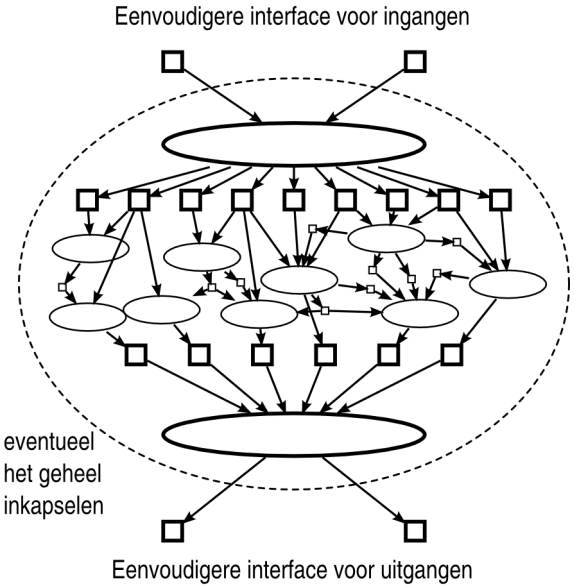
Het *Facade Pattern* kan toegepast worden door transformaties aan elkaar te hangen, zoals in figuur 4.5. In tegenstelling tot de objectgeoriënteerde variant is het na het toepassen van een Facade niet meer mogelijk om het oorspronkelijke subsysteem (die als één transformatie kan voorgesteld worden) via zijn ingangen te benaderen en de Facade te omzeilen.

State Pattern

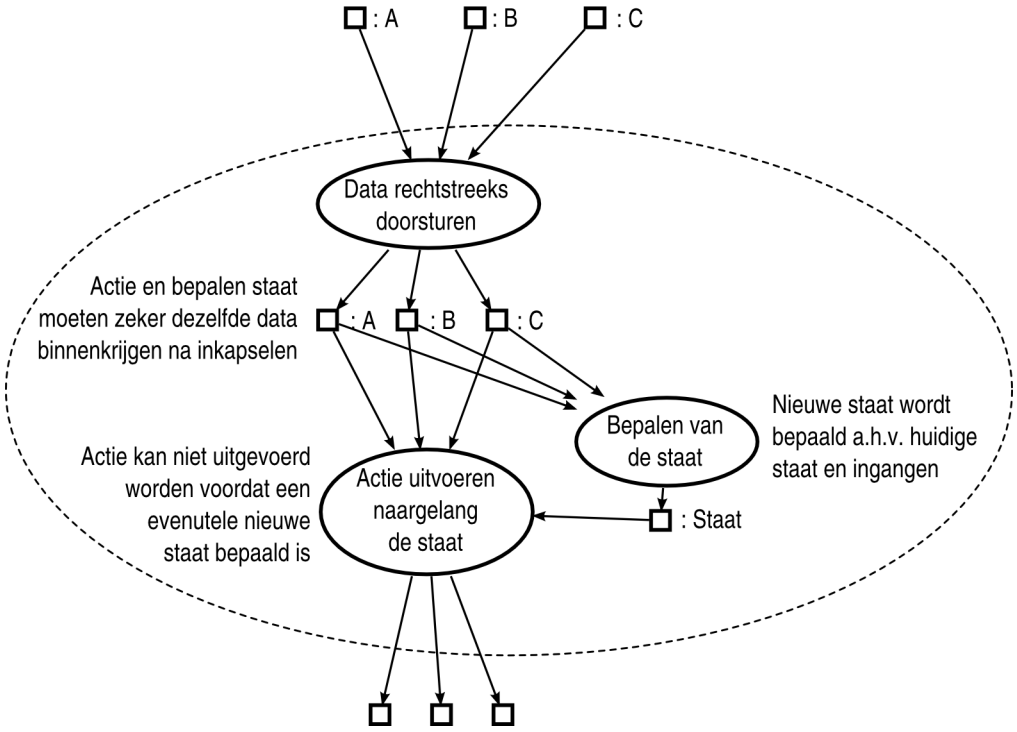
> The *State Pattern* allows an object to alter its behaviour when its internal state changes. The object will appear to change its class.

Het *State Pattern* kan verwezenlijkt worden op bijna dezelfde manier als het *Strategy Pattern*. Het verschil zit erin dat de strategie nu niet kan gekozen worden zoals de ontwerper het wil, maar bepaald wordt door de omstandigheden en de huidige staat van de transformaties die het *State Pattern* implementeren. Figuur 4.6 geeft dit grafisch weer.

De combinatie van de twee transformaties kan ingekapseld worden, zodat het gebruik van het *State Pattern* transparant kan gemaakt worden voor ontwerpers.



Figuur 4.5: Implementatie van het *Facade Pattern*



Figuur 4.6: Implementatie van het *State Pattern*

DEEL III

ONTWIKKELING VAN *Longbow*

Hoofdstuk 5

Werkwijze en evolutie

5.1 Algemeen

Het maken van ***Longbow*** was, gezien de beperkte tijd van een jaar, schipperen tussen maken van een formeel model en proof-of-concept. Hoe meer ervan was geprogrammeerd, hoe meer ik te weten kwam over hoe een formeel model in elkaar moest zitten. Het opstellen van deze laatste gaat echter veel sneller en de interne werking ervan is makkelijker te begrijpen dan die van broncode.

Een afwisseling tussen onderzoek, ontwerp en implementatie vormde de rode draad doorheen het tot stand komen van dit eindwerk. In de beginfase was dit vooral onderzoek, zonder meteen te weten waar achtergrondinformatie te vinden was. Ik heb een evolutie doorgemaakt van het mezelf moeilijk maken, door alles als een apart concept te bekijken, tot te proberen alles zo eenvoudig en uniform mogelijk voor te stellen. Het resultaat is een implementatie die werkt en van goede kwaliteit is en een formeel model dat kan dienen als basis voor een betere implementatie. Eerder bedachte begrippen komen terug en zijn eenvoudiger geworden. De uiteindelijke implementatie is veel eenvoudiger dan wat ik oorspronkelijk voor ogen had.

5.2 De beginfase

Er waren slechts enkele punten van houvast. Het systeem moest data kunnen verwerken en dat moest gebeuren met behulp van een *dependency graph*. Een brainstormperiode had enkele goede ideeën als resultaat.

Daarna begon het programmeren. Top-down: eerst een globaal overzicht van wat allemaal moest gemaakt worden, dat stuk voor stuk uitdiepen en beetje bij beetje implementeren. Deze werkwijze bleek te groots opgevat om op korte termijn bruikbare resultaten neer te zetten.

Toch zijn van de ideeën die in deze fase zijn ontstaan, een aantal bewaard gebleven. Ik heb goed leren werken met JUnit in deze periode, wat belangrijk is geweest in het verdere verloop van de thesis. Daarnaast heb veel opgezocht over technologie die ik zou kunnen gebruiken.

Een besluit dat ik uit deze fase kan trekken, is dat te veel structuur belemmerend werkt. Flexibiliteit is belangrijk voor een project dat alle kanten kan uitgaan. Te veel planning en analyse en te weinig werkende prototypes staan vooruitgang in de weg en werken demotiverend. Daarmee wil ik niet gezegd hebben dat planning en structuur niet van belang zijn als je precies weet waar je naartoe wil.

5.3 Eerste semester

Op een gegeven moment stootte ik op te veel problemen, in die mate dat ik de ontwikkeling van het eerste prototype heb stilgelegd. De onderdelen van de grote applicatie pasten niet goed samen. Ik vond geen goede manier om een GUI te ontwerpen. Dat is niet het belangrijkste onderdeel, maar er moet kunnen aangetoond worden dat het *mogelijk* is om er een te maken.

Het idee om *mark-and-sweep* te gebruiken was een doorbraak. Er vielen een aantal puzzelstukjes in elkaar. Door gebruik te maken van *mark-and-sweep* werd het GUI-probleem bijna automatisch opgelost. Er is veel nagedacht over de timing van het uitvoeren van transformaties. Er was al een – zij het ingewikkelde – oplossing voor gevonden. *Mark-and-sweep* maakte het heel wat eenvoudiger.

Een volgende vraag was hoe data konden voorgesteld worden. Ik maakte een werkend prototype dat gebruik maakte van een eenvoudig datamodel en *mark-and-sweep*.

Toen nieuwe ontwerpkeuzes opdoken, was het duidelijk dat ik een stap verder was.

5.4 Tweede semester

Vanaf dat ogenblik zou ik bottom-up beginnen ontwikkelen. Ik begon met de interne voorstelling van het graafmodel, zorgde voor een werkend prototype en begon stukje bij beetje aanpassingen te doen, tot het geworden is wat het nu is.

Hoe meer het werk vorderde, hoe eenvoudiger ik alles zag worden. Af en toe drongen er zich keuzes op waar ik op voorhand nooit bij had stilgestaan. Een van de grootste uitdagingen bleek het ontwerpen van een goede interface tussen de transformaties.

Ik ontwikkelde de GUI eerst min of meer onafhankelijk van het model dat het afbeeldde. Na een tijd waren beide modules rijp genoeg om ze aan elkaar te koppelen. De bibliotheek werd een eenvoudige `JList`, waaruit transformaties kunnen gesleept worden en neergezet op het graafmodel.

5.5 De eindfase

Gezien vanuit een toekomstgerichte visie, bleek het interessanter om sommige van mijn ideeën uit te werken, dan om te proberen zo veel mogelijk te implementeren. De mogelijkheden van deze concepten zijn namelijk uitgebreider dan ik in het begin had kunnen denken.

Ik ben ervan overtuigd dat het in deze fase nuttiger is om verder te analyseren dan om verder te werken aan de software. Het aantal geïmplementeerde transformaties is beperkt gebleven en de GUI is voor verbetering vatbaar, maar wat wel geïmplementeerd is, is van goede kwaliteit.

Over het algemeen kan deze thesis als een uitgebreide behoefteanalyse gezien worden. Er kan zonder probleem op verder gebouwd worden. De structuur die ik miste in het begin, is nu opgebouwd, zodat er gericht kan verder gewerkt worden aan ***Longbow***.

Hoofdstuk 6

Evolutie van de ontwikkeling

Dit is een beperkt overzicht van de ideeën waarmee ik mij mee heb beziggehouden. Een volledig overzicht van alle ideeën van het begin tot het einde van het project is minder interessant en vraagt meer opzoekwerk. Ik beperk mij hier tot het noteren van ideeën die inzicht moeten verschaffen in de evolutie van het ontwerp in de loop van het project.

Ik heb ervoor gekozen in dit hoofdstuk voornamelijk de laatst gebruikte namen te gebruiken en niet de oorspronkelijke namen die ik aan nieuwe concepten heb gegeven. Aangezien deze tekst waarschijnlijk door niemand als een literair hoogstandje zal bekeken worden, denk ik dat het weinig zin heeft een etymologisch overzicht erin te verwerken.

6.1 Doelstellingen

Oorspronkelijk was het vrij onduidelijk waar het naartoe zou gaan met dit project. In de loop van de ontwikkeling heb ik een bepaalde tijd een lijstje bijgehouden van eisen die ik stelde aan het eindresultaat. Dat lijstje heb ik verschillende keren bijgewerkt. In de volgende twee secties geef ik twee verschillende versies van de doelstellingen die ik had voor dit project.

6.1.1 Oorspronkelijke doelstellingen

Wat hier volgt is niet de allereerste versie van de doelstellingen, maar de eerste versie die vrij duidelijk vorm had gekregen.

Functionaliteit

- Er kan met verschillende soorten gegevens gewerkt worden: getallen, vectoren, matrices en eventueel andere soorten gegevens.
- Instroom en uitstroom van gegevens moeten eruitzien als andere bewerkingen, zodat ze zo flexibel mogelijk kunnen toegepast worden
- Er kunnen globale constanten gedefinieerd worden, die in heel het graafmodel bruikbaar zijn
- De gebruiker kan een strategie bepalen waarmee de data moeten verwerkt worden. Als een bepaalde hoedanigheid zich voordoet in de data, moet het model zichzelf automatisch kunnen aanpassen.

  Hiermee bedoelde ik dat er tijdens het verwerken van data door het graafmodel keuzes moeten kunnen gemaakt worden over hoe de resterende data verwerkt moeten worden, dit aan de hand van voorlopige resultaten.

- Uitvoer van een transformatie moet door meerdere andere transformaties kunnen hergebruikt worden.
- Het model moet in staat zijn selectief data op te halen aan de hand van referenties in de data, m.a.w. data kunnen metadata zijn.

  De bedoeling hiervan was dat, als bijvoorbeeld zou kunnen voorspeld worden dat slechts een beperkte hoeveelheid gegevens uit een databank nodig is en als dat tijdens de verwerking zou kunnen bepaald worden, dat ook implementeerbaar zou moeten zijn in een graafmodel.
- Data moeten kunnen ingedeeld worden in categorieën en per categorie verwerkt worden.
- Categorieën moeten kunnen overlappen.

  De bedoeling van deze eis was duidelijk te maken dat sequentieel verwerken van data niet altijd zou volstaan. Dezelfde statistieken moeten bijvoorbeeld kunnen bepaald worden voor de Belgen, de Nederlanders, de Belgische en Nederlandse mannen en vrouwen.

Foutbestendigheid

- Recursie op niveau van het graafmodel moet vermeden worden, om te vermijden dat de gebruiker zijn data niet kan verwerken ten gevolge van een `StackOverflowError`.
- Er moet een systeem zijn waardoor het model stopt met wachten op bijkomende data, als zeker is dat er geen data meer volgen.

  Eigenlijk waren deze regels vrij onduidelijk. Het komt erop neer dat het systeem moet kunnen gebruikt worden door mensen die niet kunnen programmeren en bijgevolg niet kunnen omgaan met programmeerfouten, zoals bijvoorbeeld een stack die overloopt. Iemand die van programmeren geen kaas heeft gegeten, heeft waarschijnlijk geen idee wat een stack is, laat staan te weten waarom hij overloopt.

Gebruiksvriendelijkheid

In deze tekst wordt met een *programmeur* een persoon bedoeld, die de mogelijkheden van het model uitbreidt door componenten te ontwikkelen die in het model passen, zoals het ontworpen is. Met een *ontwerper* wordt een gebruiker van het datamodel bedoeld, die met behulp van een script het model opbouwt en bestuurt, waarbij elementen kunnen gebruikt worden die toegevoegd zijn door een programmeur.

> In die tijd had ik nog geen duidelijk omschreven gebruikersrollen voorzien. Waar nu “ontwerper” staat, stond toen het minder duidelijke “gebruiker”. Ik was toen van plan om een scripttaal te maken waarmee het graafmodel kon opgebouwd worden. Later zou dat plan te ambitieus blijken, gezien de dingen die ik effectief voor elkaar kreeg.

- Het moet voor de ontwerper eenvoudig zijn om met het graafmodel te werken.
- Implementatie van transformaties door een programmeur moet eenvoudig zijn, hooguit enkele methoden.

  Deze eisen gelden nog altijd en zijn in grote mate verwezenlijkt.

Geen prioriteit

Volgende onderwerpen zijn wel leuk, maar zijn niet nodig om het model correct te laten werken.

- Het moet mogelijk zijn bewerkingen te laten uitvoeren door externe software.

Hiermee bedoelde ik dat transformaties software moeten kunnen aansturen, bijvoorbeeld een computeralgebrapakket voor het uitvoeren van bepaalde berekeningen. Dat is zonder meer mogelijk, want een transformatie is een stuk Javaprogrammatuur, waarmee bijgevolg programma's kunnen aangestuurd worden.

- Het model moet op verschillende computers tegelijk kunnen draaien (distributed computing).

  Dat klonk best leuk, maar ik had er in die tijd absoluut geen idee van hoe ik het zou kunnen verwezenlijken. Nu is dat even anders.
- De invoer van de gebruiker voor het model kan geoptimaliseerd worden door het datamodel op een gepaste manier te herschikken.

  Dit gaat over optimaliseren door herschikken van de transformaties in het graafmodel. Daar is nog altijd geen oplossing voor gevonden.

6.1.2 Een tijdje later...

Wat moet de applicatie doen?

- Gegevens verwerken volgens een door een ontwerper gemaakt graafmodel
- Verwerkte gegevens grafisch weergeven
- Tijdens verwerking automatisch annotaties genereren

  Met annotaties worden bijkomende datasets bedoeld, die aanduiden welke data "interessant" zijn, of die bijkomende informatie geven over de data. In een grafiek zouden bijvoorbeeld enkel interessante gebieden kunnen worden weergegeven. Oorspronkelijk werd gedacht dat dit een apart concept wat, dat losstaat van de eigenlijke data, maar uiteindelijk zijn metadata ook data en kunnen ze als dusdanig behandeld worden.
- De gebruiker de mogelijkheid bieden annotaties toe te voegen in een view.

  Samen met de opmerking bij het vorige punt betekent dit dat views interactief moeten kunnen zijn. Eigenlijk is deze doelstelling dubbelzinnig. Het is niet duidelijk of de interactief toegevoegde annotaties deel uitmaken van de data van het graafmodel.

Bovendien zijn belangrijke voorwaarden:

- Een graafmodel opstellen moet eenvoudig zijn,

  Zoals in de vorige versie van de doelstellingen ook al geëist werd.
- Verwerkingsmogelijkheden dienen makkelijk uitgebreid te worden.

  Het werk van programmeurs moet dus eenvoudig zijn, net zoals de vorige versie al vermeldde.
- Werkende implementatie (niet louter modellering) moet toegepast worden.

  Het is niet de bedoeling van de thesis om enkel een modellering te maken van een theoretisch concept, zonder dat concreet kan aangetoond worden dat het mogelijk is het te implementeren.

Voor wie is het systeem bedoeld?

Doelgroepen

1. niet-programmeurs (ontwerpers)
2. ontwikkelaars van applicaties die het systeem gebruiken
3. ontwikkelaars van *front-end*
4. ontwikkelaars die het systeem willen uitbreiden (programmeurs)

Mogelijkheden

1. Uitbreiden van het systeem door transformaties toe te voegen
   Er moeten nieuwe transformaties kunnen gemaakt worden. De bewoordingen die gebruikt werden impliceren eigenlijk dat transformaties integraal deel uitmaken van het framework. Dat hoeft niet zo te zijn.
2. Uitbreiden van het systeem door instroommogelijkheden toe te voegen
   In de huidige versie wordt instroom niet meer als iets bijzonders gezien. Invoer van buitenaf gebeurt nu via een transformatie. Het feit dat een transformatie bijvoorbeeld dient om een bestand te kunnen inlezen, betekent niet dat die geen ingangen mag hebben. De bestandsnaam zou kunnen geleverd worden door een ingang van de transformatie. Op het ogenblik dat ik deze eis neerschreef, had ik dat waarschijnlijk nog niet goed door, want in een bepaald opzicht beperkt het bekijken van instroom als iets bijzonders de mogelijkheden van het systeem.
3. Maken van een niewe *front-end*
   Aangezien dit in de praktijk alleen maar betekent dat er een publieke interface moet zijn waardoor *front-ends* het graafmodel kunnen opbouwen en manipuleren, zou even goed kunnen gezegd worden dat de publieke interface van het model goed moet ontworpen worden. Aangezien het tegengestelde beweren onzinnig is, is deze eis overbodig.
4. Maken van een nieuwe view
   In de huidige versie is dit weeral niets speciaals, zolang de view niet interactief is. Aan de andere kant zijn het niet alleen views die interactief kunnen zijn, dus is dit opnieuw eerder een beperking dan een goede eis.
5. Het systeem gebruiken voor verwerking van gegevens in een nieuwe applicatie
   Hiermee bedoelde ik dat het systeem niet alleen als framework dienst moet kunnen doen, maar ook als component moet kunnen gebruikt worden in nieuwe software.
6. Gebruik maken van een *front-end*

(. . .)

6.2 Masterplans

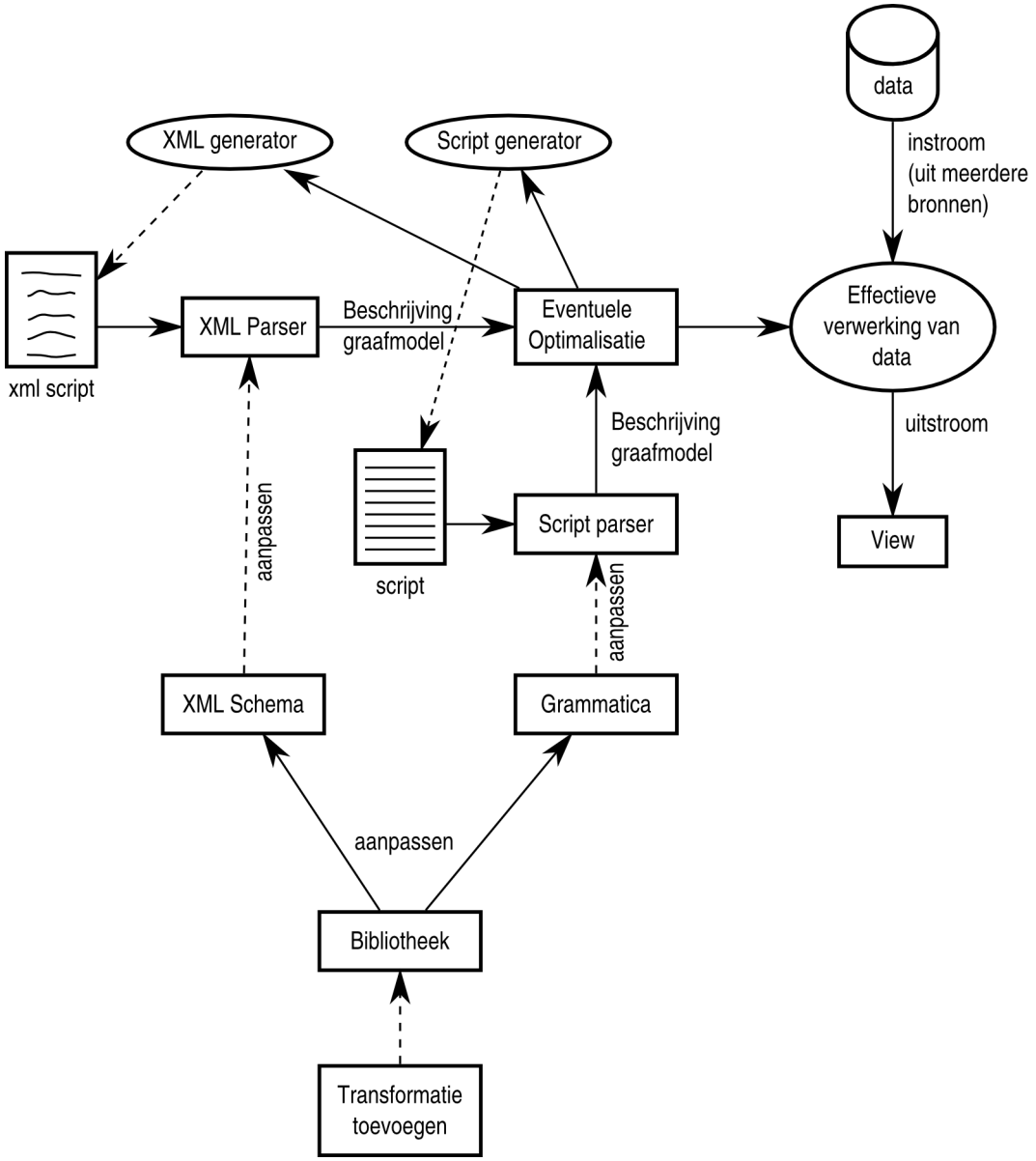
Longbow evolueerde van een heel erg uitgebreide applicatie, niet realiseerbaar door één persoon in één jaar tijd, tot een heel wat bescheidener ontwerp. Uiteindelijk blijkt bescheidenheid een goede deugd.

6.2.1 Meest uitgebreid masterplan

Dit wordt schematisch weergegeven in figuur 6.1.

Ik sprak van *front-ends* en *back-ends*. *Front-ends* zouden uitwisselbaar zijn. Ze beschrijven het graafmodel, door een aantal interfaces te implementeren. Het graafmodel zou dan in staat zijn om zichzelf op te bouwen aan de hand van die beschrijvingen. Ik had drie specifieke gevallen in gedachten:

- Een script-parser, geïmplementeerd met behulp van JavaCC, op basis van een eenvoudige grammatica die ik zou moeten ontwerpen.



Figuur 6.1: Een van de eerste versies van het top-downontwerp

- Een XML-parser, die samen met een XML Schema in staat zou zijn om een XML-bestand om te zetten in opdrachten waarmee het graafmodel kan opgebouwd worden.
- *Ad-hocfront-ends* zouden implementaties zijn die op zich het graafmodel beschrijven (hard gecodeerd).

Ik had ook verschillende *back-ends*. Die zouden met elke soort *front-end* kunnen samenwerken.

- Het graafmodel, dat uiteindelijk de data zou moeten verwerken, was de belangrijkste *back-end*.
- Een XML-generator, die een beschrijving kon omzetten in een XML-voorstelling van een graafmodel.

Tenslotte waren er ook nog modules waaraan zowel een *front-end* als een *back-end* aan konden gekoppeld worden. Zo was er bijvoorbeeld een module die de beschrijving van het graafmodel kon optimaliseren, door de transformaties en de verbindingen ertussen te herschikken. Zowel de invoer als de uitvoer van die module waren een beschrijving van een graafmodel.

Met dit systeem waren er belangrijke problemen. Ik wou onder meer een GUI voorzien waarmee makkelijker een graafmodel zou kunnen opgebouwd worden dan door het intypen van een script of een XML-bestand. Het systeem dat ik voorstelde leek een beetje op een compiler. Er bestaan GUI's waarmee compilers kunnen geconfigureerd worden, maar een GUI die als *front-end* voor een compiler dient bestaat waarschijnlijk niet. De opbouw die ik toen voor ogen had, leende zich er niet voor om het MVC principe toe te passen.

Een ander probleem was hoe nieuwe transformaties in het systeem zouden passen. Een *front-end* zou het graafmodel alleen maar beschrijven, dus moest er voor elke *front-end* op een andere manier beschreven worden welke transformaties er gebruikt kunnen worden. Het bijhouden van een bibliotheek die alle mogelijke transformaties bevat, wordt dus moeilijk en het dynamisch aanpassen van de bibliotheek bijna onmogelijk. Het toevoegen van een transformatie zou betekenen dat zowel de bibliotheek, de script-parser, de XML-parser en eventueel nog andere *front-ends* en *back-ends* moesten aangepast worden. Als een wijziging in een programma niet op één enkele plaats kan gebeuren, zou dat een belletje moeten laten rinkelen.

6.2.2 DOM en motor

Na een tijd heb ik stevig gesnoeid in mijn ontwerp. De verschillende *front-ends* waren voorlopig niet nodig. Ik moest een bruikbare GUI hebben. Het moest volstaan om een transformatie toe te voegen aan de bibliotheek om hem te kunnen gebruiken, zonder dat er bijkomende ingewikkelde acties moesten gebeuren. De kern van alles was een soort van XML DOM.

Functies van het DOM

Als basis voor heel het systeem wilde ik een *beschrijving* gebruiken van hoe transformaties samengesteld worden. Die beschrijving zou bestaan uit een hiërarchische structuur, die makkelijk om te zetten is in XML en zou kunnen dienen als basis voor verschillende toepassingen:

- Interactie met NetBeans platform (zie 6.6): zowel GUI als beschrijving worden up to date gehouden.
- Interactie met JGraph: een deel van het DOM zou een beschrijving van een graaf zijn, die zou kunnen voorgesteld worden met JGraph. Een JGraph-component zou gebruikt worden om het graafmodel weer te geven in de GUI.
- Aan de hand van de beschrijving, zou een programma voor een eindgebruiker kunnen opgebouwd worden. Hierin ligt de kiem van het idee van de motor [1].

1. zie volgend prototype

- Bestuurbaar door JavaScript. Door een hiërarchische structuur op te bouwen met JavaBeans, kan die structuur zowel door een scripttaal als JavaScript bestuurd worden, als makkelijk opgeslagen worden in een DOM.
- Serialisatie in XML is eenvoudig.

DOM is opdracht voor motor

Ik dacht dat het een goed idee was om de beschrijving en het uitvoeren van het graafmodel door twee verschillende modules te laten gebeuren: beschrijving door het DOM en en uitvoeren door de *motor*.

De motor staat los van de rest van de applicatie. Om de motor te laten werken, wordt een DOM er aan doorgegeven. De motor kan aan de hand van dat DOM, dat een formele beschrijving is van wat de motor moet doen, zichzelf klaarmaken, controleren en uitvoeren.

Voordelen:

- Opbouw van beide modules is eenvoudig: zowel motor als DOM zijn eenvoudig.
- Motor kan makkelijker geoptimaliseerd worden voor prestaties en DOM voor gebruiksgemak.
- Motor moet niet identiek dezelfde structuur hebben als DOM. Er kunnen knopen toegevoegd worden in de motor die er in het DOM niet expliciet zijn.
- Er kan van motor gewisseld worden, of de functionaliteit van de motor kan aangepast worden zonder dat de gebruiker daar veel van merkt. OCP (Open-Closed Principle) is daardoor makkelijker te bekomen.
- Taken zijn duidelijker verdeeld tussen verschillende delen van de software (Single Responsibility Principle, op niveau van softwaremodules)
- Sommige aspecten, zoals bijvoorbeeld parallelle verwerking van gegevens, kunnen volledig transparant voor de ontwerper afgehandeld worden. De motor kan optimalisaties toepassen, waardoor een functioneel equivalent graafmodel wordt bekomen, maar performanter dan wat beschreven wordt door het DOM.
- Optioneel kan de motor op zich als een motor geïmplementeerd worden (*bootstrapping*). Dat wordt een stuk eenvoudiger wanneer motor-code niet vermengd is met DOM-code.
- Rechtstreeks runnen vanuit ontwerpermodus, zonder een omweg te moeten maken via eindgebruikermodus is nu mogelijk.

Nadelen:

- Interactie met de gebruiker is moeilijk tijdens het runnen van het DPS. Een bijkomende structuur is nodig voor interactie tijdens het draaien van de motor: MVC met de motor als model.
- De eindgebruiker het een en het ander laten aanpassen voordat de motor effectief gedraaid wordt, is lastiger. De eindgebruiker werkt niet puur met de motor, maar moet ook een deel (en alleen dat deel) van het DOM kunnen aanpassen. Het DOM blijft dus twee staten behouden, hoewel deze nu wel makkelijker van elkaar te onderscheiden zijn (aanpassen door ontwerper / door eindgebruiker).
- Programmeur moet grondigere kennis hebben van ***Longbow***, omdat de componenten die hij maakt zowel voor de motor als voor het DOM moeten werken.
- Het maken van een nieuwe component voor de motor kan vereisen dat een nieuwe overeenkomstige component voor het DOM moet gemaakt worden, om de motorcomponent te kunnen gebruiken of aanpassen. Aan de andere kant is er geen 1 op 1 relatie tussen soorten motorcomponenten en soorten DOM-componenten. Kort samengevat: DRY (Don't Repeat Yourself) principe is moeilijker te realiseren.

Besluit over DOM en motor

Blijkbaar was dit model nog niet eenvoudig genoeg. Ik maakte nog altijd een onderscheid tussen de beschrijving en de uitvoering van een graafmodel. Daar zag ik voordelen in, maar uiteindelijk ben ik tot de conclusie gekomen dat ik niet wist hoe ik iets moest beschrijven zonder te weten hoe het eruit zag.

6.2.3 Het huidige prototype

Uiteindelijk ben ik begonnen met bottom-up te ontwerpen, omdat mijn vorige pogingen weinig vruchten hadden afgeworpen. Vanaf het ogenblik dat ik op die manier werkte, raakte ik sneller vooruit. Ik kon aan den lijve ondervinden of iets makkelijk te implementeren is en wat voor problemen een bepaald ontwerp oplevert.

Het onderscheid tussen het beschrijven en het uitvoeren van het graafmodel heb ik na een tijd laten vallen. Dat is het verschil tussen *designtime* (ontwerpmodus, ontwerptijd) en *runtime* (runmodus, looptijd) geworden. Ik heb de implementatie, waarmee een graafmodel kon beschreven worden, toen aangepast om het ook effectief uit te voeren. Ik heb daar geen bijkomende module voor moeten maken. Iets later ben ik een `BeanContext`framework beginnen gebruiken, omdat die een groot stuk van de functionaliteit bood die ik nodig had. Bovendien zorgt het gebruiken van componententechnologie ervoor dat het model makkelijker kan samenwerken met andere software.

Er zijn elementen die terugkomen uit de scheiding tussen DOM en motor. Dataknopen doen zich verschillend voor in ontwerp- en runmodus. Het doorgeven van data en de mogelijkheid om een dataknoop te verbinden met een ingang of een uitgang moeten los van elkaar geïmplementeerd worden.

Er verandert vrij veel en de veranderingen gebeuren vrij snel na elkaar. Het is nodig om af en toe naar *the big picture* te kijken, om te zien welke volgende stappen belangrijk zijn en welke minder.

6.2.4 Conclusie

Na het herlezen van mijn oude nota's over allerlei ontwerpen, merk ik dat ik dikwijls resoluut voor een bepaalde oplossing koos. Ik stel vast dat ik vaak verkeerde keuzes heb gemaakt, doordat ik beslissingen nam zonder goed te weten wat de impact ervan zou zijn. Naar het einde toe is het ontwerp sterk verbeterd. Door het effectief implementeren heb ik inzichten verworven die ik voordien niet kon hebben.

6.3 Transformaties en varianten

Al van in het begin dacht ik transformaties nodig te hebben met meerdere ingangen en uitgangen, zoals ze nu nog altijd worden voorgesteld. Toch is ook het concept van transformaties geëvolueerd.

Na het bedenken van nieuwe concepten, zoals bijvoorbeeld instroom en uitstroom, dacht ik dat ik voor elk soort nieuw concept een nieuw soort knoop zou nodig hebben en dat een transformatie gewoon één van die soorten knopen was. Uiteindelijk werden alle speciale knopen als transformaties gezien.

6.4 Voorstelling van data

De manier waarop gegevens voorgesteld worden is een van de belangrijkste aspecten van het ontwerp van ***Longbow***, want het bepaalt onder meer hoe de interface tussen transformaties er kan uitzien.

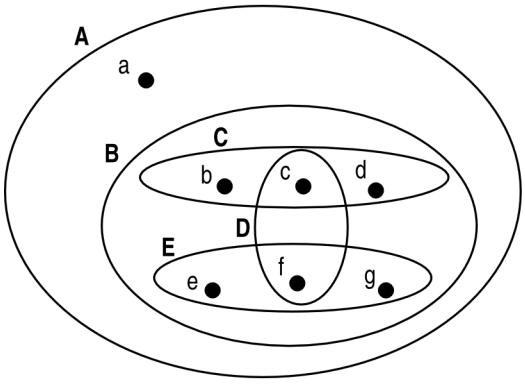
6.4.1 Centrale data

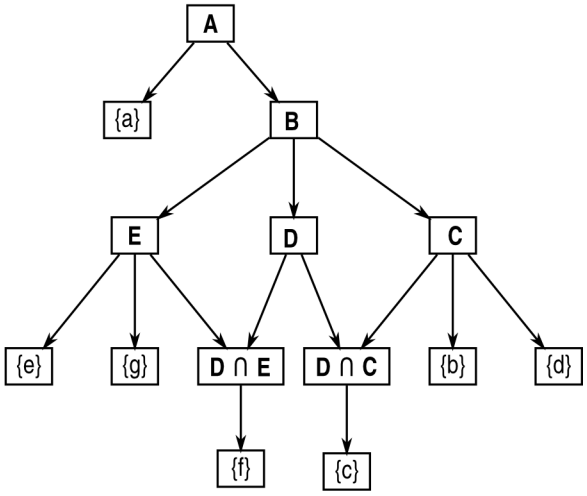
Eén aspect van de voorstelling van data waar ik mij een tijd mee heb beziggehouden, was of de data moeten gezien worden als verdeeld over het graafmodel of centraal.

In de centrale benadering worden alle data gezien als één verzameling, waarvan de inhoud grotendeels behouden blijft. Transformaties doen iets met de data, waardoor die op sommige plaatsen worden vervangen door andere gegevens, of waardoor gegevens worden toegevoegd of verwijderd. Toevoegen, verwijderen en wijzigen van data kan alleen het gevolg zijn van het uitvoeren van een transformatie. Opvragen ervan kan ofwel ook enkel via transformaties, ofwel rechtstreeks.

De verzameling die alle data bevat kan voorgesteld worden als een *dependency graph*. In deze context is een verzameling geen deelverzameling van zichzelf. Een element kan voorgesteld worden als het (unieke) singleton dat dat element bevat. De relatie tussen verzamelingen met als voorschrift *"is deelverzameling van"* is een partiële orderelatie. Als elke verzameling wordt voorgesteld door een knoop van een graaf en elk element van de relatie met als voorschrift *"is deelverzameling van"* als een verbinding, dan ontstaat er bijgevolg een *dependency graph*. Deze graaf is duidelijk niet dezelfde als de graaf die de transformaties in zijn knooppunten bevat.

Niet alle verzamelingen zijn weergegeven. **C** U **D** is bijvoorbeeld een deelverzameling van **B.** Alle onderlinge relaties kunnen zowel uit het Venn-diagram als uit de dependency graph afgeleid worden.





Niet alle verbindingen zijn weergegeven in de dependency graph. Er kunnen verbindingen bestaan van elke knoop naar al zijn opvolgers. Er mogen geen verbindingen worden weggenomen als dat de partiële orde zou wijzigen.

Deze structuur kan als dusdanig als datastructuur in een applicatie gebruikt worden. Een verbinding kan een referentie naar een andere verzameling zijn. In die zin mogen er verbindingen toegevoegd en verwijderd worden om de performantie te bevorderen.

Figuur 6.2: Verzamelingen: Venn-diagrammen en *dependency graph*

Figuur 6.2 laat een voorbeeld van zo'n eenvoudige *dependency graph* zien, samen met de Venn-diagrammen. De verzamelingen hoeven niet weergegeven te worden tot op het niveau van de singletons, als duidelijk is welke verzamelingen er zijn en wat de onderlinge relaties ertussen zijn.

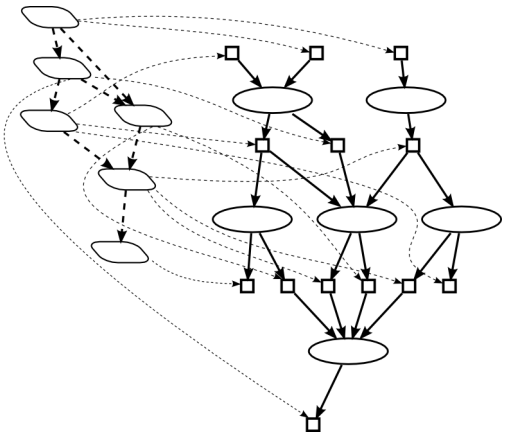
Voor het opstellen van een datastructuur op basis van dit principe moet nagedacht worden over waar de data – de elementen – effectief worden opgeslagen. Er zijn tal van mogelijkheden:

- In alle verzamelingen waartoe ze behoren (als referentie)
- Enkel in de grote verzameling. De verzamelingen waartoe ze behoren zijn eigenschappen die bij het element horen
- Enkel in de meest specifieke verzameling waartoe ze behoren
- Een combinatie van bovenstaande mogelijkheden

Verder moet er nog een beslissing genomen worden over de voorstelling van de interfaces tussen de transformaties worden. Intern zullen transformaties natuurlijk met de echte data, dus de elementen van de verzamelingen werken. Tussen de transformaties onderling kan gekozen worden of er met verzamelingen of met elementen gewerkt wordt. Anders gezegd: er moet gekozen worden op welke manier beide *dependency graphs* met elkaar kunnen verweven worden. Deze benaderingen worden besproken in de volgende twee paragrafen.

Elementgebaseerd benadering

Op basis van deze benadering ontstond het eerste werkende prototype dat gebruik maakte van mark-and-sweep. De interface tussen transformaties wordt bepaald door de types van de elementen die tussen twee transformaties worden doorgegeven, zoals weergegeven in figuur 6.3.



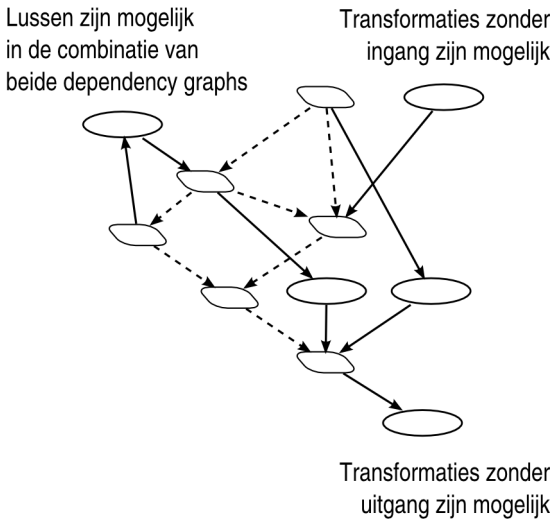
Figuur 6.3: Verweven van het datamodel met het graafmodel, volgens de elementgebaseerde benadering

De interface die in de huidige versie van het prototype gebruikt wordt lijkt hierop, maar er wordt niet meer uitgegaan van een centraal datamodel.

Het is niet eenvoudig om te werken met gegevens die andere data bevatten. Als een bepaalde transformatie bijvoorbeeld een matrix nodig heeft, een andere transformatie een rij en nog een andere enkele kolommen van die matrix, kan dat problemen veroorzaken. Een wijziging van een rij van de matrix wijzigt de matrix zelf en minstens één kolom. In de praktijk moeten verzamelingen als elementen bekeken kunnen worden, wat tegenstrijdig is met de opzet van deze benadering.

Het matrixvoorbeeld was de struikelsteen voor de elementgebaseerde benadering. Er kan alleen met atomaire data gewerkt worden. Wijzigen van een element mag geen wijzigingen in andere elementen veroorzaken, als die niet rechtstreeks het gevolg zijn van het uitvoeren van een transformatie. Als dat mogelijk is voor een bepaalde toepassing, dan kan het elementgebaseerd model zeker overwogen worden. Maar ik stelde mij niet tevreden met het mogelijk maken van slechts enkele datatypes.

Verzamelinggebaseerde benadering



Figuur 6.4: Voorbeeld van een graafmodel volgens de verzamelinggebaseerde benadering

Figuur 6.4 toont een voorbeeld van hoe een graafmodel met een verzamelinggebaseerd datamodel er kan uitzien. De transformaties moeten één of meerdere *dependency graphs* vormen, zonder daarbij rekening te houden met de verbindingen tussen de verzamelingen onderling. Elke verzameling kan in zo'n graaf meermaals voorkomen. In de figuur wordt dat weergegeven door twee transformaties die aan hun uitgang met dezelfde verzameling verbonden zijn. Het is zelfs mogelijk dat een transformatie met dezelfde verzameling verbonden is, zowel aan de ingang als aan de uitgang.

Voor elke verzameling moet er een signatuur gedefinieerd worden. Ook de ingang en de uitgang van elke transformatie moeten een signatuur hebben. Opdat een ingang of een uitgang met een verzameling zouden kunnen verbonden worden moeten de signaturen ervan compatibel zijn. De signaturen vormen de interface tussen verzamelingen en transformaties.

Telkens een verzameling gewijzigd wordt, moet eerst het datamodel consistent zijn en daarna het graafmodel (de *dependency graph* met de transformaties). Het consistent maken van het graafmodel kan er weer voor zorgen dat het datamodel inconsistent wordt. Zo kan dat een tijdje, of zelfs oneindig lang blijven doorgaan. Het datamodel en het graafmodel vormen samen een dynamisch systeem, mogelijk met stabiele en instabiele evenwichtspunten en oscillatoren. Er zijn gebieden waarin convergentie mogelijk is en andere waarin het systeem enkel kan divergeren.

Als een verzameling wordt gewijzigd, wijzigen automatisch alle verzamelingen waar die verzameling deel van uitmaakt. Die kunnen makkelijk worden teruggevonden via de *dependency graph* van het datamodel. Potentieel worden er ook deelverzamelingen gewijzigd.

De gewijzigde verzamelingen geven mogelijk aanleiding tot inconsistenties in het graafmodel. Alle transformaties waarvan een gewijzigde verzameling verbonden is met de ingang moeten opnieuw uitgevoerd worden. Alle transformaties waarvan een – door het consistent maken van het datamodel – gewijzigde verzameling verbonden is met de uitgang moeten ook opnieuw uitgevoerd worden, om er zeker van te zijn dat de wijziging van de verzameling consistent is met het graafmodel.

Deze twee stappen moeten herhaald worden tot het graafmodel consistent is geworden. Het blijft dus belangrijk dat er in het graafmodel geen lussen voorkomen. In voorkomend geval zou

consistentie niet mogelijk zijn. Dat er in het datamodel ook geen lussen mogen voorkomen is natuurlijk triviaal.

Mocht het mogelijk zijn met de verzamelinggebaseerde benadering een bepaalde klasse van dynamische systemen te modelleren dan is dit een bijzonder krachtig concept. Het kan de moeite waard zijn om dit idee verder uit te werken.

De gegevens waarmee in deze benadering kunnen gewerkt worden zijn niet meer beperkt tot atomaire data. Nu stelt zich de eis dat alle referenties tussen de data onderling gekend moeten zijn. Doordat objecten dikwijls referenties naar andere objecten inkapselen, is het niet mogelijk om met alle datatypes te werken. Dit datamodel werd dus ook niet toegepast voor ***Longbow***.

Een aangepast relationeel model lijkt een goede oplossing te zijn voor de implementatie van dit concept. Men kan voorlopig experimenteren met een lichte relationele databank, die kan ingebed worden in een toepassing, zoals JavaDB.

Continue verzamelingen en constraints Continue verzamelingen zijn verzamelingen die gedefinieerd worden door een voorschrift en niet door een opsomming van de elementen. Het zou interessant kunnen zijn om ook met dit soort verzamelingen te kunnen werken. Als dit wordt vergeleken met SQL-databanken, dan zou een `CREATE TABLE` statement een continue verzameling kunnen voorstellen.

Continue verzamelingen bepalen de *constraints* (beperkingen) die van toepassing zijn op de elementen die het bevat. Een deelverzameling van een continue verzameling erft die *constraints* en kan enkel meer beperkingen opleggen aan zijn elementen. Die elementen moeten immers ook tot de continue verzameling behoren.

Bedenkingen over centrale data

Ongetwijfeld biedt het voorstellen van data in een *dependency graph* interessante mogelijkheden. Er zou op basis van dit idee bijvoorbeeld een toepassing kunnen gemaakt worden die controleert of bepaalde gegevens elkaar tegenspreken. Zo'n toepasing zou kunnen bestaan uit een *dependency graph* als datamodel. Transformaties zouden tussen de verzamelingen kunnen geplaatst worden. Er kunnen data, verzamelingen en transformaties toegevoegd worden. Een evenwichtstoestand zou er dan op wijzen dat het nieuwe feit niet in tegenspraak is met de reeds bekende feiten. Als er een oneindige lus ontstaat of als het systeem divergeert, wijst dat op een inconsistentie in de data of de transformaties ertussen.

Het zou waarschijnlijk geen groot probleem zijn om bijkomende gegevens toe te voegen aan een bestaand systeem, zodat tegenspraken efficiënt kunnen gevonden worden. Een dergelijk systeem zou kunnen toegepast worden om bijvoorbeeld:

- Beweringen van getuigen in een moordzaak te controleren tijdens een verhoor.
- Fouten te zoeken in een script van een film of een toneelstuk
- Te zoeken naar inconsistenties in de wetgeving van een land of streek
- Te kunnen antwoorden op de vraag of een bepaalde handeling legaal is, of na te gaan of de handeling of de wetgeving erover dubbelzinnig is
- Beschrijven welke tests zullen uitgevoerd worden om software, die nog in ontwikkeling is, te testen. Bovendien worden de verwachte testresultaten vermeld. Als bovendien een formeel model van de te maken software wordt toegevoegd, kan er in het geheel van het formeel model en en tests met resultaten gezocht worden naar inconsistenties.

Deze toepassingen zouden kunnen gerealiseerd worden door feiten formeel voor te stellen en te classificeren in een verzameling. Transformaties zouden verbanden tussen deze verzamelingen beschrijven. Een goede formele voorstelling van feiten vinden is waarschijnlijk het moeilijkste probleem in al deze toepassingen.

Classificeren in een verzameling kan door feiten te labelen met sleutelwoorden. Het toekennen van een sleutelwoord aan een feit is equivalent met het onderbrengen van een feit in een bepaalde verzameling. Ook het toekennen van een regel aan een feit is hiermee equivalent. De regels waaraan een bepaald feit moet voldoen om geen andere feiten tegen te spreken bepalen dus de verzamelingen en transformaties die ermee verband houden. Verder kan met een feit een beschrijving gepaard gaan, zodat er voor zowel mensen als computers interpreteerbare informatie wordt voorgesteld.

Door het toevoegen van een feit wordt er meer bekend dan alleen dat feit zelf. Kennis bestaat niet alleen uit feiten, maar ook uit hun onderlinge relaties, net als relaties tussen feiten en relaties en relaties tussen relaties onderling. Al die kennis moet op een zo efficiënt mogelijke manier worden weergegeven, wat een grote uitdaging op zich is.

De verzamelingen en transformaties kunnen opgebouwd worden door de feiten één voor één toe te voegen. Als er na het toevoegen van een bepaald feit geen evenwichtstoestand kan gevonden worden, dan duidt dat op een tegenspraak. Als er een oscillatie ontstaat, vormt die daarvan een bewijs (uit het ongerijmde). Dat bewijs kan voorgelegd worden aan een gebruiker, die er eventuele fouten kan uithalen, of bevestigen dat er een tegenspraak is.

6.4.2 Gedistribueerd datamodel

Gegevens worden nu niet meer gezien als iets centraals dat deels verwerkt wordt en door elke bewerking voor het grootste deel behouden blijft. Bij verwerking van gegevens wordt het resultaat telkens gekopieerd. Het grote consistentieprobleem wordt daarmee van tafel geveegd.

Het grote voordeel van de elementgebaseerde benadering was de eenvoud. Maar waar dat model veronderstelde dat alle gegevens atomair zijn en dat ze geen deel uitmaken van andere data, is dat hier niet het geval.

Het voordeel van de verzamelinggebaseerde benadering was dat er met ingewikkelde geneste datastructuren zou kunnen gewerkt worden, terwijl de ontwerper daar weinig of geen rekening mee zou moeten houden. Een groot nadeel ervan is dat de verzamelinggebaseerde benadering niet performant zou zijn.

Een gedecentraliseerde oplossing kan de voordelen van beide benaderingen combineren. Gegevens worden bij verwerking gekopieerd. Programmeurs zijn ervoor verantwoordelijk dat dit effectief gebeurt.

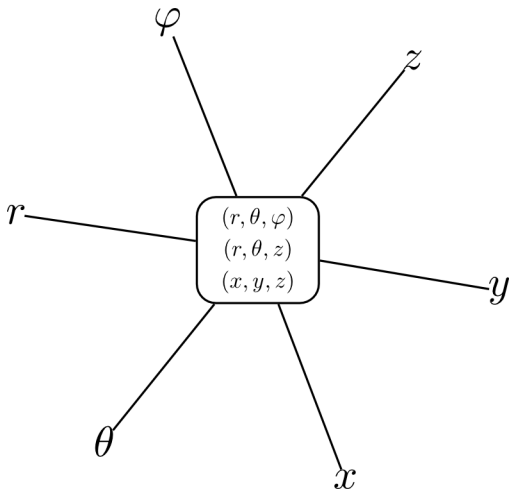
In de praktijk is kopiëren niet altijd nodig en kan soms vermeden worden, bijvoorbeeld bij het gebruik van *immutable* objecten of singletons.

Het grootste voordeel van een gedistribueerd datamodel is dat het eenvoudig is en toch alle datatypes toelaat. Dat wordt aangetoond in het geïmplementeerde prototype. Bijkomende voordelen zijn dat sommige van de toepassingen en uitbreidingen uit deel IV waarschijnlijk onmogelijk zouden zijn met een centraal datamodel. Onder meer het gebruik van contextafhankelijke regels voor het beschrijven van de interface (zie 7.4.3) zou erin problemen kunnen opleveren.

Een nadeel van een gedistribueerd model is dat het moeilijker is om centraal gegevens bij te houden, die relevant zijn op verschillende plaatsen in het graafmodel. Het aanbieden van diensten (zie 9.2) biedt hier een oplossing. .

Nog een nadeel is dat er door het kopiëren van data waarschijnlijk meer geheugen zal verbruikt worden dan met een centraal datamodel. Een gedistribueerd model biedt evenwel de mogelijkheid om de verwerking van data te spreiden over meerdere computers (zie 9.6.3).

6.4.3 Objecten met meerdere voorstellingsvormen



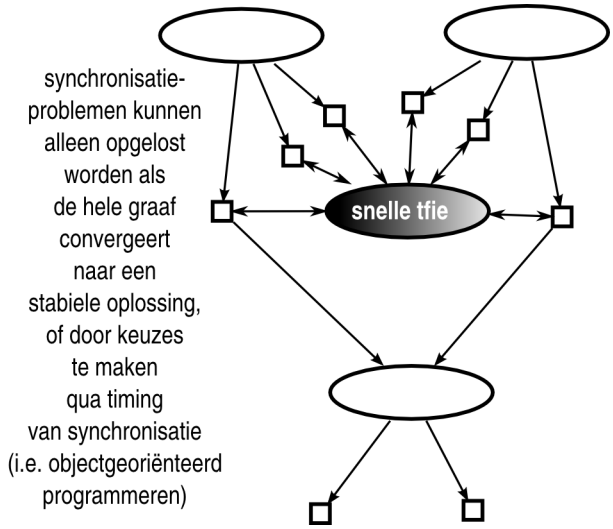
Figuur 6.5: Verschillende voorstellingsvormen van een punt in een driedimensionale ruimte

Sommige objecten kunnen op verschillende manieren gemanipuleerd worden om hetzelfde te bereiken. Een voorbeeld van zo'n object is een punt in een driedimensionale ruimte. Zo'n punt kan in verschillende coördinatensystemen worden weergegeven, bijvoorbeeld cartesiaans (x, y, z), met cilindercoördinaten (r, θ, z) en met bolcoördinaten (r, θ, φ). Voor verschillende toepassingen kunnen verschillende voorstellingsvormen nuttig zijn. De vraag is hoe zo'n punt consistent in alle coördinatensystemen kan worden weergegeven en aangepast. Dit geval is vrij braaf. Het zou kunnen voorgesteld worden zoals op figuur 6.5. Elke coördinaat kan ondubbelzinnig aangepast worden, als geëist wordt dat r nooit 0 kan worden.

In een graaf kan het punt voorgesteld worden als een knoop. Elke coördinaat kan ook als een knoop voorgesteld worden, met een verbinding naar het punt. De verbindingen zijn nu niet meer gericht, want elke coördinaat kan aangepast worden, waardoor het punt wijzigt en als het punt wijzigt, kunnen de coördinaten aangepast worden. In feite komt dit neer op een graafvoorstelling van een object dat een punt in een driedimensionale ruimte voorstelt, in een objectgeoriënteerde taal. Een objectgeoriënteerd programma kan namelijk als een graaf bekeken worden. De coördinaten kunnen als verschillende eigenschappen van het punt gezien worden. De interne voorstelling van het object is ingekapseld.

Een ingewikkelder object dan een punt zou een venster kunnen zijn, waarvan de positie en de grootte op verschillende manieren kunnen worden weergegeven. Als de linkerkant van het venster wordt verplaatst, kan het ofwel verbreden, ofwel opschuiven naar links. Er moet een *keuze* gemaakt worden. Wie *The Matrix* trilogie gezien heeft weet dat computers daar problemen mee kunnen hebben. Het kan zijn dat er geen convergentie van de graaf naar een evenwichtssituatie mogelijk is als die keuze automatisch wordt gemaakt. De inconsistentie wordt hier niet veroorzaakt door het datamodel, maar door het feit dat er in de graaf verbindingen in twee richtingen werden geïnjecteerd.

Snelle transformaties zouden een object met meerdere verschijningsvormen kunnen voorstellen. Ze zorgen ervoor dat verschillende representaties van hetzelfde object consistent worden voorgesteld. Figuur 6.6 toont hoe dit visueel zou kunnen voorgesteld worden. Het moet duidelijk zijn dat hier synchronisatieproblemen kunnen optreden.



Figuur 6.6: Synchronisatieprobleem bij snelle transformaties

Data moeten ondubbelzinnig kunnen voorgesteld worden onder één enkele vorm. Meerdere voorstellingen van dezelfde data zijn interessant en dikwijls nodig, maar overgangen tussen verschillende voorstellingen moeten expliciet gemodelleerd worden door transformaties om synchronisatieproblemen te vermijden.

Longbow implementeert een dynamisch model, terwijl onmiddellijke interne synchronisatie van objecten enkel mogelijk is in een statisch model.

6.5 Persistentie

De structuur van het graafmodel moet kunnen worden opgeslagen. Er moet een graafmodel kunnen ingeladen worden, zodat een ontwerper niet voortdurend dezelfde graaf moet ineenpuzzelen.

Er moet onderscheid gemaakt worden tussen persistentie van de structuur enerzijds en van de data anderzijds. Verwerkte gegevens kunnen opgeslagen worden als er kan van uitgegaan worden dat ze niet meer zullen wijzigen. Het zou handig zijn om een systeem te vinden dat de balans in evenwicht houdt tussen plaats en reconstructietijd. Dat is nog altijd een openstaand probleem.

In de loop van de ontwikkeling werd er vooral nagedacht over de persistentie van de structuur. Er werd slechts op het laatst een oplossing gevonden voor persistente opslag van verwerkte data (zie 9.5).

In het eerste ontwerp (6.2.1) werd er nog niet zo goed nagedacht over persistentie. Toch zou dit geen probleem hebben opgeleverd. De graafstructuur werd voorgesteld in een script of in een XML-bestand, wat in se persistente voorstellingen zijn. Het opslaan van data gebeurde via uitstroomknopen, wat allerlei mogelijke vormen van dataopslag mogelijk maakte. Terug inlezen van de structuur zou bijvoorbeeld gebeuren door een script te interpreteren. Opgeslagen data zouden terug opgehaald kunnen worden via instroomknopen.

Het laten varen van de beschrijving van het graafmodel als kern van de applicatie heeft als negatief effect gehad dat het ontwikkelen van een persistente vorm moeilijker is geworden. Na heel wat lees- en opzoekwerk denk ik dat JAXB 2.0 de meest gepaste technologie hiervoor is. Om deze technologie te gebruiken, moeten annotaties aangebracht worden in de broncode die gepersisteerd moet worden. Aangezien de huidige implementatie afhankelijk is van de (oude) implementatie van `BeanContextSupport`, die de nodige annotaties niet bevat, kan JAXB 2.0 momenteel nog niet gebruikt worden. De meest voor de hand liggende oplossing is het herimplementeren van de interface `BeanContext`, *met* annotaties in de broncode.

Als voorlopige oplossing kan een graafmodel opgeslagen worden door middel van serialisatie. Alle objecten waaruit het graafmodel is opgebouwd moeten dus van een klasse zijn die de interface `Serializable` implementeert, of als `transient` (vergankelijk) gemarkeerd zijn langs alle paden van waaruit ze bereikbaar zijn. Alle vergankelijke attributen van een geserialiseerd object moeten tijdens het deserialiseren opnieuw geïnitialiseerd worden. Dat gebeurt in de methode `readResolve()`.

6.6 GUI

Ik heb meer tijd besteed aan het ontwikkelen van de GUI dan ik oorspronkelijk wilde, maar ik kreeg daardoor belangrijke inzichten in functie van de ontwikkeling van het model. Door het ontwikkelen van de GUI ben ik erin geslaagd metamodellen te bedenken. Een GUI ontwikkelen betekent immers onder meer dat er moet nagdacht worden over hoe een model kan ingebed worden in een toepassing. Het ontwikkelen van de editors voor de dataknopen zonder voorlopers heeft mij het doorslaggevend inzicht gegeven dat interfaces voor transformaties moeten beschreven worden, zodat ze rekening kunnen houden met *constraints*.

6.7 NetBeans Platform

Voor het maken van een GUI wilde ik gebruik maken van het *Netbeans Platform*[2]. Dat biedt de mogelijkheid een GUI samen te stellen uit modules die apart kunnen geïmplementeerd worden. Een NetBeansmodule kan in meerdere applicaties hergebruikt worden. Alle modules die voor de IDE gemaakt zijn kunnen hergebruikt worden in een nieuwe GUI. Een eenvoudige manier om een HTML-editor te maken op basis van het NetBeans platform is bijvoorbeeld de IDE als basis te nemen en alle overbodige modules weg te laten.

Ik was van plan om de volgende items uit de IDE te hergebruiken:

- Ondersteuning voor het werken met projecten. Een project zou onder andere bestaan uit een bibliotheek en een graafmodel.
- Maken van nieuwe bestandstypen. In het begin legde ik nogal de nadruk op de mogelijkheid om het werk van een ontwerper te kunnen opslaan. Ondersteuning voor het maken van nieuwe bestandstypen zou goed van pas komen.
- Er kunnen meerdere editorvensters geopend worden in verschillende tabbladen. Die kunnen naast en onder elkaar gezet worden. Dat leek me handig voor het editeren van een graaf op verschillende plaatsen, of voor het editeren van meerdere grafen tegelijk.
- Het runtimevenster toont de resources die gebruikt worden. Dat komt in de IDE goed van pas en zou bijvoorbeeld kunnen laten zien welke grafen aan het runnen zijn.

2. [NetBeans Platform, 2007]

- De "palette" is de component van waaruit beans kunnen gesleept worden in Matisse, of van waaruit html-componenten gesleept kunnen worden om in te voegen in de html-editor. Met een beetje metselwerk zou ik de palette kunnen aanpassen om er een bibliotheek van transformaties van te maken.
- Het propertiesvenster zou de functie van een editor overnemen. Na aanklikken van een dataknoop zouden in het propertiesvenster alle eigenschappen ervan kunnen worden weergegeven en eventueel aangepast.
- Enkele handigheidjes die toch al beschikbaar zijn in de IDE hadden ook in de GUI van pas kunnen komen: To do, User tasks, Cvs & subversion, Favorieten en eventueel ook een module manager.

Ook handig is de integratie van JavaHelp, een online documentatiesysteem.

Ik ben een week bezig geweest met het maken van een prototype op basis van het NetBeans Platform. In die week bleek dat het gemak waarmee applicaties kunnen samengesteld worden iets rooskleuriger werd voorgesteld dan het in werkelijkheid is. Herhaaldelijk stootte ik op fouten. Een zoektocht op het internet leverde een bug report op en gelijkaardige problemen zijn ook terug te vinden op fora. Volgens testers is de bug nochtans niet reproduceerbaar op andere machines. Vaak kon de fout opgelost worden door opnieuw te beginnen, maar de echte oorzaak ervan bleef een raadsel. NetBeansontwikkelaars beweren dat zulke fouten aan een configuratieprobleem kunnen liggen. NetBeans herinstalleren had dit euvel moeten verhelpen, maar dit bleek niet het geval te zijn. Gezien het beperkte resultaat dat ik ermee heb behaald, heb ik besloten het NetBeans Platform links te laten liggen.

6.7.1 JGraph

Ik ben begonnen met het maken van een prototype om te leren werken met JGraph. Dat zou immers een belangrijk onderdeel vormen van het maken van de GUI. Het heeft vrij lang geduurd voordat ik JGraph een goede plaats heb kunnen geven in ***Longbow***. Dat komt waarschijnlijk vooral door het feit dat de interface voor het JGraphmodel sterk verschilt van de interface van het ***Longbow***model.

Uiteindelijk heb ik het *Proxy Pattern* toegepast voor het model van JGraph. Ik heb de interface `GraphModel` geïmplementeerd, waarbij de meeste methoden rechtstreeks werden gedelegeerd naar een `DefaultGraphModel`, de standaardimplementatie van die interface. De overige methoden sluisde ik door naar het ***Longbow***model. Die vormde samen met de `GraafModelProxy` ongeveer een MVC-structuur.

Het omleiden van sommige functies van het model heeft nefaste gevolgen gehad op bepaalde functionaliteit die JGraph normaal gezien biedt. Ongedaan maken lukte niet meer nadat de GUI aan het model gekoppeld was. Groeperen van knopen leidde keer op keer tot het crashen van de Java Virtual Machine. Soms ging dit gepaard met een stacktrace waarin niets van mijn code te bespeuren was en soms met een disassembly van *glibc*, een van de belangrijkste onderdelen vn C voor Linux. Het was mijns inziens onmogelijk te achterhalen waar de fout precies zat.

6.7.2 Drag-and-drop

Drag-and-drop had ik nodig om ontwerpers de mogelijkheid te bieden transformaties te slepen uit een lijst – de bibliotheek – en ze neer te zetten op het graafmodel. Sinds Java 1.4 biedt Swing een eenvoudige interface voor het implementeren van drag-and-drop.

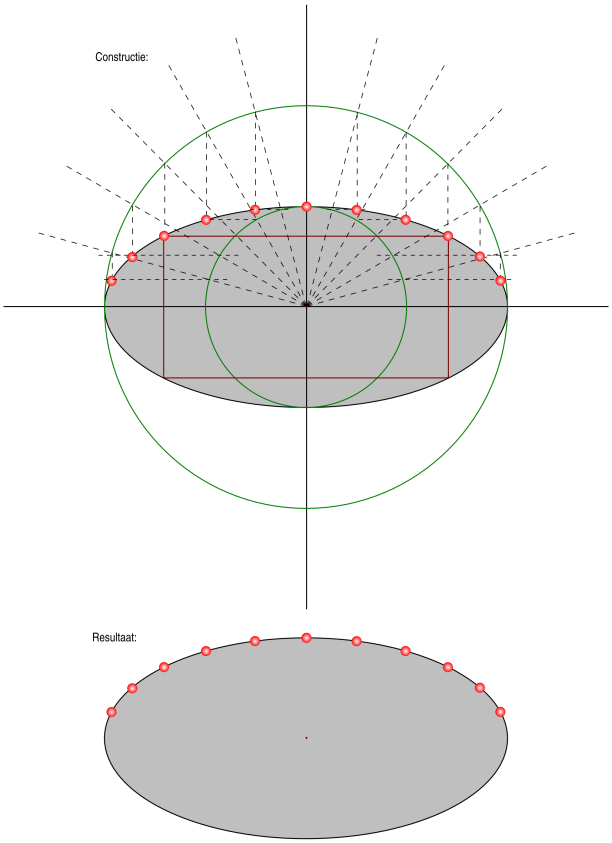
De JGraph-handleiding vermeldt twee verschillende manieren om drag-and-drop toe te passen. Ik heb beide geprobeerd zonder succes. Ook op het JGraph-forum heb ik posts gelezen van mensen

die er niet in slaagden dit aan het werk te krijgen. Uiteindelijk vond ik een eigen oplossing, buiten de handleiding van JGraph.

6.7.3 Ellipsen

Plaatsen van poorten op de transformaties

Ik heb enkele mogelijheden geprobeerd om mooie spreiding te krijgen van poorten op transformaties. De mooiste spreiding vond ik die volgens gelijke excentrische anomalie, zoals op figuur 6.7. Ik heb uiteindelijk de poorten niet weergegeven op de rand van de transformaties.



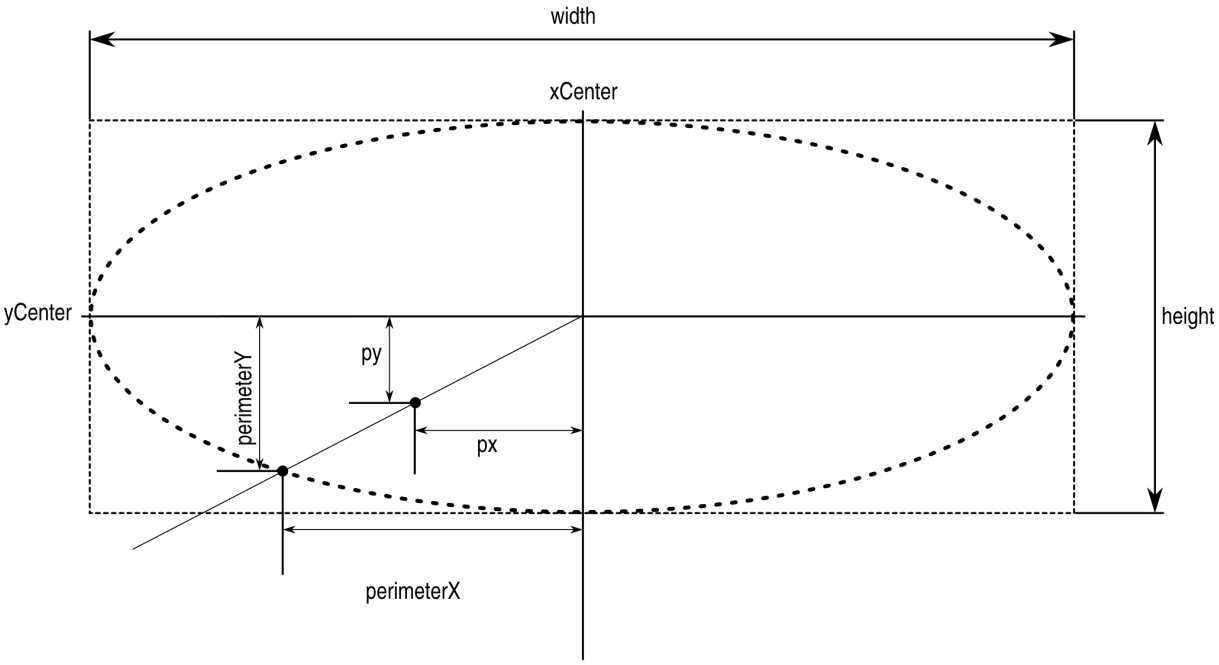
Figuur 6.7: Spreiding van poorten op een transformatie volgens gelijke excentrische anomalie

Punt op de omtrek van een ellips

Figuur 6.8 dient als documentatie bij de methode `getPerimeterPoint()` in de klasse *TransformatieRenderer*. In die methode wordt de positie van een punt op de omtrek van een ellips bepaald.

6.7.4 Tekortkomingen

De GUI kan slechts een beperkt deel van de mogelijkheden van ***Longbow*** benutten. Er zou een GUI moeten ontworpen worden die gelijke tred kan houden met de ontwikkeling van het model. De ontwikkeling van die GUI zou heel snel moeten kunnen gebeuren, wat een sterk contrast zou betekenen met de ontwikkeling van de huidige GUI.



Figuur 6.8: Bepalen van een punt op de omtrek van een ellips

Ik denk dat de meest voordelige manier om zo'n GUI – zij het een desktop- of een webapplicatie – te ontwikkelen is, om die zo veel mogelijk op basis van ***Longbow*** te maken. De voordelen daarvan zijn:

- De GUI is flexibel: makkelijk aan te passen en uitbreidbaar. Het is heel waarschijnlijk dat grote delen ervan kunnen hergebruikt worden, als de GUI moet veranderen,
- Het is een goede test voor het maken van metamodellen
- Het is een goede test voor het implementeren van MVC met behulp van ***Longbow***.
- Er kan makkelijk gebruik gemaakt van hierna beschreven *features*, zoals versiebeheer, sessiebeheer, ...

6.8 Enkele tekortkomingen aan Alpha

Altijd dataknopen rond transformaties

Transformaties altijd omringen met dataknopen is een manier om transformaties altijd uitvoerbaar te maken en om makkelijk verbindingen te maken in de GUI. De oorspronkelijke bedoeling was dat ingangen en uitgangen *facultatief* kunnen zijn. Een transformatie zou niet uitvoerbaar zijn als er nog ingangen of uitgangen zouden zijn die zowel niet verbonden als niet facultatief zijn. Dit begrip is verloren gegaan door ten allen tijde alle ingangen en uitgangen van alle transformaties te verbinden met dataknopen.

Interface voor het maken van verbindingen

De interface voor het maken van verbindingen in het model zelf kan beter. Ik zie een framework voor samenstelling van componenten in een *dependency graph* in Java als een uitbreiding van de BeanContextServices specificatie. De uitbreiding van de interface moet ervoor zorgen dat er een *dependency graph* kan opgebouwd worden.

In de huidige *user interface* zijn er voor het maken van verbindingen dataknopen nodig. Die zou ik het liefst zo veel mogelijk verbergen. De componenten waar het om gaat in de samenstelling zijn de transformaties.

DEEL IV

FORMELE BESCHRIJVING

Hoofdstuk 7

Bedenkingen vooraf

Het doel van dit deel is het ontwerpen van een formeel model voor een framework dat kan dienen voor het samenstellen van componenten. Het framework moet kunnen dienen om software te ontwikkelen zonder te programmeren in een klassieke programmeertaal. De componenten die daarvoor kunnen gebruikt worden moeten natuurlijk wel al bestaan.

Aan de hand van het geïmplementeerde prototype werd een formeel model voor ***Longbow*** ontwikkeld. De bedoeling is dat het model zo algemeen mogelijk toepasbaar is voor het ontwikkelen van software met behulp van bestaande (of nog te implementeren) componenten.

7.1 Classificatie van objecten en transformaties

7.1.1 Objecten

Mutable en immutable

Er zijn *mutable* (*veranderlijke*) en *immutable* (*onveranderlijke*) objecten. Veranderlijke objecten zijn objecten waarvan de waarde, of de waarde van een van de eigenschappen ervan, kan veranderen nadat het object is aangemaakt. Een object met een veranderlijke eigenschap is altijd veranderlijk. Onveranderlijke objecten moeten door nieuwe worden vervangen als eenzelfde soort object nodig is met een andere waarde.

Gebonden en ongebonden

De veranderlijke objecten kunnen ingedeeld worden in *gebonden* (*bound*), *gedeeltelijk gebonden* (*partially bound*) en *ongebonden* (*unbound*) objecten. Een gebonden object is een object dat in staat is al zijn wijzigingen te melden aan minstens één transformatie. Een gedeeltelijk gebonden object is een object dat in staat is welbepaalde wijzigingen te melden aan een transformatie. Als de wijzigingen die niet kunnen gemeld worden niet relevant zijn in de context van het graafmodel waar de transformatie zich in bevindt, kan het gedeeltelijk gebonden object in die context als een gebonden object beschouwd worden. De meeste, zoniet alle componententechnologieën voorzien in de mogelijkheid om *events* te sturen. *Events* kunnen gebruikt worden om een object geheel of gedeeltelijk te binden.

Een gebonden of gedeeltelijk gebonden object is altijd veranderlijk, want een onveranderlijk object kan nooit een wijziging van zichzelf melden.

7.1.2 Transformaties

Interactief

Een transformatie is *interactief* als ze communiceert met een (in de context van het graafmodel) gebonden object dat kan wijzigen op minstens één andere manier dan door impulsen vanuit het graafmodel zelf. Het object waar de transformatie mee communiceert is een *interactieve component*.

Een eenvoudig voorbeeld van een interactieve component is een component in een GUI, waar een mens acties op kan uitvoeren.

Een interactieve transformatie kan bijvoorbeeld gebruikt worden om een printer te bedienen. Er kunnen documenten naar de printer gestuurd worden om af te drukken en de staat van de wachtrij van de printer kan opgevraagd worden. Een printerdriver kan m.a.w. gezien worden als een interactieve component.

Monitor

Een *monitor* is een transformatie die van alle *relevante* wijzigingen van een object op de hoogte wordt gehouden, zonder dat vanuit het graafmodel berichten naar dat object gestuurd worden. De tijd waarop berichten naar de monitor gestuurd worden is onbepaald.

Het monitoren van een component kan ruim opgevat worden. In sommige omstandigheden is het mogelijk een component te monitoren door op geregelde tijdstippen de staat ervan op te vragen. Dat is slechts een lapmiddel, bij gebrek aan een betere oplossing. De melding van het ontvangen van een e-mail is bijvoorbeeld handiger dan periodieke controles.

Staatloos

Een transformatie is *staatloos* als het enkel gegevens van zijn ingangen verwerkt en verwerkte data naar de uitgangen stuurt. Er wordt geen gebruik gemaakt van gegevens die reeds berekend werden voordat de data aan de ingangen bekend waren. Dat geldt zowel voor gegevens in het graafmodel zelf, als voor data in componenten die door de transformatie gebruikt worden.

De eenvoudigste transformaties zijn staatloos, maar staatloze transformaties kunnen ook complexe bewerkingen uitvoeren, bijvoorbeeld gegevens ophalen uit een databank.

Een staatloze transformatie kan een bepaalde bewerking uitvoeren, bijvoorbeeld het berekenen van het gemiddelde van een lijst van getallen.

Stateful

Een *stateful* transformatie kan bewerkingen uitvoeren aan de hand van reeds bepaalde gegevens en gegevens die via de ingangen kunnen verkregen worden. De reeds bepaalde gegevens kunnen via de uitgangen verkregen worden, of via een component.

Een voorbeeld van het gebruik van *stateful* transformaties werd gegeven in de beschrijving van het *State Pattern* (zie 4.4.2).

Een *stateful* transformatie moet overweg kunnen met een initiële situatie, wanneer er nog geen vooraf bepaalde data beschikbaar zijn. Er moet een *initiële staat* zijn.

Visueel

Een *visuele* transformatie is een transformatie die iets kan laten zien, doordat het een *grafische component* bevat. Hij kan interactief zijn, maar is dat niet noodzakelijk. Hij is afhankelijk van de *grafische context* van het graafmodel (zie 9.1.4). Een visuele transformatie moet de grafische context van het graafmodel ondersteunen om eraan te kunnen worden toegevoegd. Als de transformatie meerdere visuele contexten ondersteunt, moet hij in staat zijn de visuele context van het graafmodel te bepalen, om zo te beslissen welke van de grafische componenten actief worden. Een graafmodel ondersteunt niet altijd een grafische omgeving.

Als voorbeeld van een grafische context kan Swing genomen worden. Een grafische component die kan weergegeven worden is dan een `java.awt.Component`. Als een transformatie zowel een `JComponent` als een Java Server Faces-component bevat, dan moet de `JComponent` geselecteerd worden voor weergave.

Vaak zullen visuele transformaties ook interactief zijn. Het eenvoudigste voorbeeld hiervan is een knop of een tekstveld die kan worden weergegeven. Een grafiek kan een voorbeeld zijn van een transformatie die niet interactief is.

Een niet interactieve visuele transformatie is bijvoorbeeld: een transformatie die grafiek met resultaten toont, waarmee met betrekking tot het graafmodel verder niets kan worden gedaan.

Zoals hierboven al vermeld, bestaan er ook interactieve transformaties die niet visueel zijn. Er is dus geen direct verband tussen het visueel zijn en het interactief zijn van een transformatie.

7.2 Component Composition

In 1968[1] werd er al gewag gemaakt van het feit dat software voornamelijk op ambachtelijke manier wordt ontwikkeld en dat er geen gebruik wordt gemaakt van massaproductietechnieken *voor het ontwerp* van software. M.D. McIlroy beweerde in zijn lezing dat een industrialisering van het ontwerpen betekent dat software uit componenten opgebouwd wordt.

Sindsdien is er veel veranderd. Objectgeoriënteerd programmeren is tegenwoordig de meest gebruikte techniek voor het maken van software. Voor het werken met softwarecomponenten bestaan er diverse technologieën.

Componenten

Er bestaat geen standaarddefinitie van een component. Sommigen beweren dat een klasse een component is, anderen beweren dat een component een module is die voldoet aan de eisen van een of andere componentenarchitectuur.

In deze tekst worden componenten in een ruime zin gezien. Transformaties en al wat erdoor kan vertegenwoordigd worden, is een component. Componenten uit bestaande componentenarchitecturen voldoen daaraan. Een combinatie van componenten zou in deze betekenis soms als één component kunnen gezien worden.

Samenstellen van componenten

Component Composition is het samenstellen van componenten, zodat de samenstelling uitvoerbare (of toch op zijn minst bruikbare) software vormt. Bij voorkeur is een samenstelling van

1. [Naur and Randell (Eds.), 1969]

componenten op zich ook een component, die weer kan samengesteld worden met andere componenten.

Het samenstellen van componenten kan ongecontroleerd zijn. In dat geval is de vraag of een samenstelling correct werkende software oplevert moeilijk op te lossen, zelfs als de componenten afzonderlijk correct werken. Het samenstellen kan ook op een meer gecontroleerde manier, met behulp van een framework. Als componenten op een gecontroleerde manier worden samengesteld, dan beperkt dit het aantal vrijheidsgraden van het probleem. Bijgevolg wordt het makkelijker om componenten *automatisch* samen te stellen. Dat komt neer op een meer 'industriële manier' voor het samenstellen van componenten.

Longbow tracht dit te verwezenlijken door de componenten te laten vertegenwoordigen door een transformatie en de transformaties samen te stellen. Twee transformaties met elkaar verbinden, zou tot gevolg kunnen hebben dat er een verbinding gelegd wordt tussen de componenten die ze vertegenwoordigen.

7.3 Semantic Web

Een semantisch web is een vorm van artificiële intelligentie. Het *Semantic Web* is een uitbreiding van het www, die poogt een wereldwijd semantisch web te implementeren. De mogelijkheden daarvan zijn gigantisch. Een van de mogelijkheden is helpen bij het samenstellen van componenten: *Component Composition* en *Web Service Composition*.

Meer uitleg over het *Semantic Web* is te vinden in [Berners-Lee et al., 2001], een gevulgariseerd artikel. Het artikel is geschreven door Tim Berners-Lee, de bedenker van het www en van het *Semantic Web*. Hier volgt een beknopte en onvolledige samenvatting van wat het *Semantic Web* inhoudt.

7.3.1 Ontologieën

Wie de dag van vandaag wil zoeken op het Web naar het goedkoopste ramgeheugen dat in een straal van 10 kilometer verkocht wordt, wacht een lange zoektocht. Als er gegoogled wordt naar de termen 'ramgeheugen', 'Gent' en 'prijs' is de kans klein dat de juiste resultaten gevonden worden. Google snapt niet waarnaar gezocht moet worden.

Opdat software dat wel zou kunnen begrijpen, moeten er metadata worden opgeslagen in webpagina's. Die metadata kunnen genoteerd worden in RDF (Resource Description Framework), een set geordende drietallen. Elk drietal bevat een onderwerp (bv. 'een mens'), een relatie (bv. 'is een') en een voorwerp (bv. 'zoogdier'). Alle drie worden ze voorgesteld door een URL. De betekenis van die URL's wordt door mensen gegeven en kan gebruikt worden om software te schrijven, die met de annotaties (of regels) kan redeneren. Met behulp van RDF kan er bijvoorbeeld beschreven worden hoe de prijs, de fabrikant, de technologie en de verkoopplaats van een plaatje ramgeheugen uit een website kan gehaald worden, zodanig dat de informatie op de bewuste webpagina door software interpreteerbaar is.

Een verzameling mogelijke regels is een *ontologie*. Ontologieën en relaties ertussen moeten gedeclareerd kunnen worden. OWL[2] (Web Ontology Language) kan daarvoor dienen.

2. OWL en niet WOL: dit is geen typfout

7.3.2 Agents

Met behulp van het *Semantic Web* kunnen zogenaamde *software-agents* informatie verzamelen en samenstellen. *Software-agents* zijn componenten die in staat zijn te redeneren aan de hand van bepaalde regels. Ook informatie die niet rechtstreeks is af te leiden uit één enkele bron kan met behulp van *agents* achterhaald worden. Er kan een bewijs van een bepaald feit geformuleerd worden, dat er bijvoorbeeld kan uitzien als een opeenvolging van webpagina's.

7.3.3 Nut voor *Longbow*

Als er bij transformaties op een gepaste manier aantekeningen gemaakt worden, kunnen ze zodanig beschreven worden, dat een *question answering system* vragen erover zou kunnen beantwoorden. De nodige annotaties zouden kunnen gegenereerd worden aan de hand van annotaties in de broncode van de transformaties.

Longbow kan samenwerken met het *Semantic Web*. Transformaties kunnen op die manier makkelijker teruggevonden worden. Daardoor verkleint de kans op herimplementatie van software met hetzelfde doel. Softwareontwikkelaars zullen zich dus beter kunnen bezighouden met het ontwikkelen van nieuwe software, in plaats van het wiel opnieuw uit te vinden. Het gebrek aan dat soort informatie bemoeilijkt bijvoorbeeld in Unix-systemen het terugvinden van een programma met een bepaald doel.

Als er daarvoor een *agent* bestaat die intelligent genoeg is, kan er software gemaakt worden, enkel door te beschrijven wat die software moet doen. Er kan gezocht worden naar de transformaties of componenten die kunnen gebruikt worden voor het verwezenlijken van het uiteindelijke doel. Het zoeken naar de verschillende modules kan met behulp van een bibliotheek die gebruik maakt van het *Semantic Web* (zie 9.11). Het samenstellen ervan kan met behulp van een framework zoals beschreven in dit deel. Intelligente *agents* kunnen redeneren over hoe de samenstelling moet gebeuren.

7.4 Interfaces: geen identificatie, maar beschrijving

7.4.1 Probleemstelling

Stel dat de interface van transformaties zou gedefinieerd zijn zoals in objectgeoriënteerde talen: door een identificatie (de naam van een methode) en een signatuur. Het op die manier declareren van een interface kan naar transformaties vertaald worden. Ingangen en uitgangen hebben een identificatie en zouden een bepaald gegevenstype als signatuur kunnen hebben.

Als de signatuur van zowel de uitgang van een transformatie als de ingang van een andere transformatie zou overeenkomen met het type `File`, zou dat problemen kunnen veroorzaken. Het overeenkomen van de signatuur zou betekenen dat de uitgang en de ingang met elkaar mogen verbonden worden. Het is echter onmogelijk om na te gaan over welk soort bestanden het gaat in beide signaturen. Als de uitgang het over een binair bestand heeft en de ingang bedoelt een ascii-bestand, dan zijn fouten tijdens het uitvoeren van de transformaties bijna verzekerd.

Er zijn verschillende mogelijkheden om dit probleem op te lossen:

- Transformaties kunnen tijdens het uitvoeren altijd controleren welke gegevens ze binnenkrijgen. Zo kunnen fouten voorkomen worden tijdens het uitvoeren. Dit vereist telkens extra werk tijdens het uitvoeren en dat kan heel vaak zijn. De performantie gaat er bijgevolg sterk op achteruit. Dit is dus geen optie.

- Er kan een subtype van `File` gemaakt worden, zodat het overgedragen bestand zeker van het juiste type is.
  Deze methode zal problemen tijdens het uitvoeren vermijden, maar een goede oplossing is het niet. Er kan voor een bepaald bestandstype een subklasse van `File` gemaakt worden, maar niets belet dat er voor hetzelfde bestandstype ook nog een andere subklasse van `File` wordt gemaakt. Een uitgang zou als signatuur het ene subtype van `File` kunnen hebben en een ingang het andere. Hoewel het verbinden van de uitgang met de ingang in principe geen problemen zou opleveren, is dat toch onmogelijk, omdat hun signatuur niet overeenkomt.
  Om dit nieuwe probleem op te lossen zou er een transformatie kunnen tussengevoegd worden, die in staat is het ene type om te zetten in het andere (een *adapter*). Zo'n transformatie zou moeten toegevoegd worden in alle gelijkaardige gevallen. Dat impliceert dat ofwel dergelijke transformaties automatisch moeten toegevoegd worden, ofwel dat ze door een ontwerper moeten worden toegevoegd. Het toevoegen door een ontwerper is geen optie, want een ontwerper kan in principe niet programmeren en beseft niet noodzakelijk wat het probleem is. Het automatisch toevoegen is ook geen optie, want dat vereist dat er een formele beschrijving bestaat van alle paren gelijkwaardige datatypes, of dat die beschrijving ergens uit kan afgeleid worden. Dit is ook geen optie, want:
  - Zo'n beschrijving bestaat niet.
  - Voor het beschrijven van interfaces bestaat er een betere oplossing (zie verder).
- Een derde mogelijkheid is dat zowel de ingang als de uitgang het type `File` gebruiken om aan te duiden dat zij met een bestand werken. De signatuur wordt uitgebreid met bijkomende eisen, die duidelijk maken wat voor een bestand het moet zijn: te openen of al geopend, te lezen of te schrijven, ...
- De vorige mogelijkheid kan verder uitgebreid worden. Het kan handig zijn dat zowel een bestand als een gegevensstroom doorgegeven kunnen worden om te lezen. Bestanden en stromen worden in Java door verschillende klassen weergegeven. Een ingang of een uitgang zou dus bijvoorbeeld als eis moeten kunnen stellen dat ofwel een bestand om te lezen ofwel een invoerstroom moet worden doorgegeven, en dat de data die ze bevatten moet bestaan uit ascii-tekens.

Het vergelijken van datatypes is niet voldoende voor het aflijnen van een interface tussen transformaties. Het creëren van bijkomende types lost het probleem niet op zonder nieuwe problemen te creëren.

Voor het samenstellen van componenten is het bepalen van hoe de interface tussen die componenten eruitziet een van de belangrijkste problemen. Die interface bepaalt in een controlerend framework grotendeels de mogelijkheden voor het onderling verbinden van de componenten.

Er moet voor kunnen gezorgd worden dat de interface niet te beperkend is en bovendien mag de interface geen verbindingen toelaten die tot fouten kunnen leiden tijdens de uitvoering.

7.4.2 Interface door definitie

Interface door definitie is een interface zoals die bekend is van objectgeoriënteerde talen.

In OO talen worden interfaces gezien als een verzameling van methoden. Methoden worden gezien als een koppel van een identificatie (de naam van de methode) en een signatuur (inclusief return). Bij elke interface hoort een contract, waar een programmeur zich aan zou moeten houden. Dit heeft enkele gevolgen, waaronder de volgende:

- Twee methoden die dezelfde signatuur en hetzelfde contract, maar een verschillende identificatie hebben, kunnen niet door elkaar gebruikt worden.

- Het contract is niet bindend. Het zich niet houden aan een contract – bijvoorbeeld een negatief getal doorgeven waar dat niet mag – is mogelijk en kan fouten tot gevolg hebben. De interface is in deze gevallen te vrij en dat kan niet verholpen worden. De achterliggende reden is dat het contract in menselijke taal is opgesteld en bedoeld voor programmeurs. Het contract is niet interpreteerbaar door software, dus kan het er niet door worden afgedwongen.
- Het contract negeren kan moeilijk te begrijpen broncode opleveren (wat in zekere zin ook als een fout kan beschouwd worden).
- Sommige mogelijkheden van sturen van berichten kunnen niet benut worden, omdat de identificatie van de interface het niet toelaat. De interface is in deze gevallen te sterk gebonden aan regels en er zijn geen middelen om dat rechtstreeks te verhelpen. Dat kan bijvoorbeeld wel via een omweg verholpen worden door het *Adapter Pattern* toe te passen, maar dan kunnen er zich problemen voordoen zoals hierboven beschreven.

7.4.3 Interface door omschrijving

De interface van een transformatie bepaalt welke andere transformaties ermee verbonden kunnen worden. Verbinden van transformaties gebeurt via de ingangen en de uitgangen (zie volgend hoofdstuk). De interface van een transformatie bestaat dus uit een beschrijving die op elk moment bepaalt welke ingangen en uitgangen een transformatie heeft en beschrijft bovendien op elk moment hoe die ingangen en uitgangen met andere ingangen en uitgangen verbonden kunnen worden. De interface is een *formele* beschrijving, die interpreteerbaar is door software.

Het doel van dit soort interfaces is tweeledig:

1. Als twee transformaties met elkaar verbonden kunnen worden, dan moeten ze gegarandeerd samen kunnen uitgevoerd worden. Dit is het analogon van het contract in interface door definitie.
2. Tegelijk moet ervoor gezorgd worden dat elke transformatie met zoveel mogelijk andere transformaties moet kunnen verbonden worden. Dat resulteert in een zo groot mogelijke bruikbaarheid van de transformatie.

Dit soort interface omvat interface door definitie.

Garanderen van het contract

Om te kunnen garanderen dat het verbinden van twee transformaties geen problemen zal veroorzaken tijdens de uitvoering, moet er een contract opgesteld worden, waaraan beide uiteinden van de verbinding moeten voldoen. Dit contract moet kunnen afgedwongen worden. Het is de bedoeling dat een transformatie nooit data te verwerken krijgt waar het niet mee overweg kan.

Opstellen van het contract. Een contract bestaat uit één of meer regelverzamelingen. Een regelverzameling bepaalt een soort data dat aanvaard wordt. Een regelverzameling die geen enkele soort data toelaat is zinloos. Ook het opstellen van een contract zonder regelverzamelingen is zinloos, want dat zou betekenen dat er geen verbindingen mogelijk zijn.

De basis van een regelverzameling is een beschrijving van een categorie van data die kunnen aanvaard worden, bijvoorbeeld een datatype. Er kunnen regels toegevoegd worden aan de verzameling, die de mogelijkheden om data te aanvaarden beperken. Zo'n regel kan complex zijn en zelfs contextgevoelig (zie 8.5.3).

De regels die bepaald worden in een contract, worden ook *constraints* of *beperkingen* genoemd.

Een regel toevoegen aan een regelverzameling kan er niet voor zorgen dat het aantal mogelijkheden om data te aanvaarden toeneemt. Regels zijn altijd beperkend. Om meer soorten data toe te laten, moet er een nieuwe regelverzameling toegevoegd worden aan het contract. Regels verwijderen kan in principe niet, want dat beperkt de mogelijkheden voor versiebeheer van transformaties (zie 9.3.1).

Het contract bepaalt de verzameling van alle mogelijke data die het kan aanvaarden. Contracten kunnen aan elkaar toegevoegd worden, door de regelverzamelingen van het ene contract toe te voegen aan het andere contract. Dat komt overeen met het bepalen van de unie van de verzamelingen van mogelijke data. Analoog moeten contracten van elkaar kunnen worden afgetrokken, zodat het resultaat het verschil van de verzamelingen mogelijke data bepaalt.

Er is verder onderzoek nodig om een efficiënte manier te vinden waarop de doorsnede van de verzamelingen bepaald door twee contracten kan bepaald worden. Er moet ook gezocht worden naar een efficiënte manier om te bepalen of zo'n verzameling deelverzameling is van (of gelijk is aan) een andere. Eventueel moeten er bijkomende voorwaarden gesteld worden aan het opstellen van de contracten, zodat deze operaties efficiënt kunnen verlopen.

Er is verder onderzoek nodig om te bepalen waaraan contextgevoelige regels moeten voldoen opdat dergelijke verbanden met verzamelingentheorie behouden blijven.

Afdwingen van de regels. Regels op zich helpen niet veel als ze niet worden toegepast. Het contract moet op twee manieren kunnen afgedwongen worden: voor het toevoegen van data aan een dataknoop zonder voorlopers en voor het maken van een verbinding.

Controleren van data. Dit gebeurt wanneer data aan een constante worden toegevoegd in ontwerptijd (zie 8.8.12). Er moet dan gecontroleerd worden of een bepaald object voldoet aan het contract.

Controleren van metadata. Dit gebeurt telkens als een verbinding gemaakt wordt. De contracten aan het begin en aan het eind van de verbinding moeten compatibel zijn. Dat betekent dat als een uitgang van een transformatie wordt verbonden met een ingang van een andere transformatie, die laatste nooit data mag ontvangen van de eerste, waar ze niet mee overweg kan. Het contract van de uitgang moet dus een deelverzameling bepalen van de verzameling bepaald door het contract van de ingang. Bovendien mag de doorsnede van die twee verzamelingen niet ledig zijn.

Mogelijke implementaties voor contracten

Ad-hocimplementatie. In een ad-hocimplementatie wordt een contract hard gecodeerd. Er worden twee methoden geïmplementeerd: een voor het controleren van data en een voor het controleren van metadata.

XML Schema. XML Schema biedt uitgebreide mogelijkheden voor het bepalen van regels waaraan datatypes moeten voldoen. In veel gevallen zal XML Schema voldoende zijn voor het opstellen van contracten. Een beschrijving kan bovendien op een ontologie gemapt worden.

Resource Description Framework. Regels kunnen m.b.v. RDF beschreven worden. Op die manier kan van de mogelijkheden van het *Semantic Web* gebruik gemaakt worden, bijvoorbeeld om transformaties te zoeken op het web, die kunnen verbonden worden met bepaalde ingangen of uitgangen van andere transformaties. Dat zou een grote stap zijn in de richting van het automatisch samenstellen van transformaties.

7.4.4 Contextgevoelig aanpassen van de transformatie

Er kunnen regels opgesteld worden om te bepalen in welke gevallen een transformatie bepaalde ingangen en uitgangen bevat. Als er geen contextgevoelige regels zijn, moet er minstens een beschrijving zijn van welke ingangen en uitgangen er zijn.

Mogelijke implementaties voor deze regels zijn ad hoc en met behulp van een ontologie.

Hoofdstuk 8

Formeel model voor een framework voor samenstelling van componenten in een dependency graph

Dit formeel model is een verbeterde versie van Alpha, het laatst geïmplementeerde proof-of-concept. Wat hier beschreven staat kan als basis dienen voor een implementatie. Het is eigenlijk het resultaat van een uitgebreide behoefteanalyse.

8.1 Gebruikersrollen en modi

Er zijn enkele duidelijk afgelijnde gebruikersrollen. Voor het grootste deel vallen die rollen samen met modi waarin ***Longbow*** kan uitgevoerd worden.

Er is een sterk verband tussen gebruikersrollen en modi. Ze vallen echter niet volledig samen, want een overgang tussen verschillende modi kan geïnitieerd worden door een gebruiker die een bepaalde rol vervult.

Gebruikers kunnen meerdere rollen vervullen, maar voor één specifiek graafmodel vervult een gebruiker op elk moment juist één rol.

8.1.1 Frameworkprogrammeurs

Beschrijving

Frameworkprogrammeurs zijn de programmeurs die dit framework implementeren, of er uitbreidingen aan toevoegen. Uitbreidingen worden beschreven in het volgende hoofdstuk en zijn niet te verwarren met transformaties.

Voor de implementatie van het framework wordt van een bepaalde programmeertaal gebruik gemaakt. Die programmeertaal wordt in het vervolg *de programmeertaal van het framework* genoemd.

Modus

Aan deze rol is geen modus verbonden. Frameworkprogrammeurs implementeren het hier beschreven model, of de uitbreidingen uit het volgende hoofdstuk. Ze moeten daarbij met alle modi rekening houden.

8.1.2 Programmeurs

Beschrijving

Programmeurs kunnen nieuwe transformaties implementeren. Er bestaat een duidelijk onderscheid tussen programmeurs en frameworkprogrammeurs.

Modus

Programmeurs werken in de *statische modus*. Ze beschikken over een implementatie van dit model en maken ervoor een toepassing. Ze wijzigen het framework niet. In principe weten ze zelfs niet met welke implementatie van het framework ze te maken hebben.

De statische modus onderstelt dat een werkende implementatie van het framework beschikbaar is, zonder dat bekend is welke strategieën (zie 9.1) zullen gebruikt worden. Transformaties die afhankelijk zijn van een bepaalde strategie worden dus door frameworkprogrammeurs ontwikkeld en niet door programmeurs. Het is beter om dit soort transformaties te vermijden. Er is immers geen garantie dat het gebruik ervan door een graafmodel ook mogelijk is bij het toepassen van een andere strategie.

8.1.3 Ontwerpers

Beschrijving

Ontwerpers kunnen het graafmodel wijzigen (zie 8.8). Ze hoeven hiervoor geen gebruik te maken van de programmeertaal van het framework. Dit gebeurt door berichten te sturen naar en te ontvangen van het graafmodel en zijn onderdelen.

Een ontwerper hoeft niet altijd een mens te zijn. Er moet naar gestreefd worden om het ontwerpen zo veel mogelijk automatisch te laten verlopen.

Er mag van een ontwerper niet geëist worden dat hij kan programmeren in de programmeertaal van het framework.

Modus

Ontwerpers werken in *ontwerpmodus*. Synoniemen daarvan zijn *designtime* en *ontwerptijd*.

8.1.4 Eindgebruikers

Beschrijving

Eindgebruikers maken gebruik van een door een ontwerper samengestelde toepassing. Soms interageren ze met de toepassing, maar soms hebben ze alleen maar het resultaat ervan nodig. Ze voeren het graafmodel uit.

Modus

Eindgebruikers werken in *uitvoeringsmodus*, ook *runmodus*, *runtime* of *uitvoeringstijd* genoemd.

8.2 Data

Data die in een graafmodel worden doorgegeven tussen transformaties, worden ingekapseld in een dataobject.

8.3 Metadata

Een metadataobject is een *immutable* object dat een contract beschrijft, zoals beschreven in 7.4.3. Metadata bepalen het type en de *constraints* van data. De termen *metadata* en *contract* zijn onderling uitwisselbaar.

Het is ook mogelijk dat er geen direct verband is tussen metadata en data die worden doorgegeven in het graafmodel. Het is bijvoorbeeld mogelijk dat metadata een verbindingsmogelijkheid tussen componenten buiten het graafmodel om beschrijven, die via het graafmodel kan gecontroleerd worden.

Metadataobject M_1 is compatibel met metadataobject M_2 als de verzameling van aanvaarde data bepaald door M_1 een deelverzameling is van de verzameling van aanvaarde data bepaald door M_2. Er kan ook gezegd worden dat M_1 aanvaard wordt door M_2.

Er wordt een methode voorzien om te controleren of een object voldoet aan het contract en een methode om te controleren of de metadata compatibel zijn met die van een gegeven metadataobject.

Dataknopen en poorten bevatten metadata. Compatibiliteit van metadata is transitief, dus als de metadata van een uitgang en een dataknoop compatibel zijn en de metadata van die dataknoop en een ingang zijn compatibel, dan zijn de metadata van die uitgang en die ingang compatibel. Compatibiliteit van metadata is een partiële orderelatie.

8.4 Koppeling tussen metadata en data

Een dataobject kan aangemaakt worden aan de hand van een metadataobject. Hiervoor wordt een *Factory Method Pattern* gebruikt. De *factory* heeft één publieke methode, die een metadataobject als argument heeft. De returnwaarde is een dataobject. Verschillende metadata kunnen vereisen dat er verschillende soorten dataobjecten aangemaakt worden. Dankzij het gebruik van een *factory*, kunnen dataobjecten uniform via hun interface benaderd worden. Voor meer informatie over *design patterns* wordt verwezen naar de uitgebreide literatuur over dat onderwerp.

8.5 Transformaties

8.5.1 Beschrijving

Een transformatie is een component die ingangen en uitgangen kan hebben. Via de ingangen en uitgangen kunnen verbindingen gemaakt worden. Transformaties worden geïmplementeerd als klassen afgeleid van een abstracte klasse die deel uitmaakt van het framework. Ze kunnen geïmplementeerd worden in de programmeertaal van het framework.

Een transformatie kan dienen als een *placeholder* van een component. Die component kan een component in een bepaalde componententechnologie zijn. Het kan ook een *webservice*, of nog een andere soort softwaremodule zijn. De transformatie kan dienen als communicatiemiddel tussen het graafmodel en de externe component. In die zin is het mogelijk om een graafmodel te gebruiken als framework woor *Component Composition* of *Web Service Composition*.

Er moet gestreefd worden naar de mogelijkheid transformaties niet te programmeren, maar formeel te beschrijven. Aan de hand van de beschrijving kan er dan dynamisch een transformatie aangemaakt worden. Het intensief gebruik maken van bijvoorbeeld *annotations* in Java kan een eerste stap in die richting zijn. Het uiteindelijke doel is om zo veel mogelijk werk uit handen te

nemen van de programmeurs. Programmeurs moeten zich kunnen concentreren op de applicatielogica, en niet op het schrijven van *boilerplate code* die nodig is om een transformatie in een framework te laten passen.

Een bijkomend voordeel van het beschrijven van transformaties, in plaats van het programmeren ervan, is dat ze bruikbaar kunnen worden in meerdere implementaties van het framework, zelfs in verschillende programmeertalen. Dit doet denken aan de beschrijving van *webservices*, met behulp van WSDL.

8.5.2 Statische modus

Implementatie van transformaties

Transformaties kunnen geïmplementeerd worden door een klasse te maken, afgeleid van een abstracte klasse of interface die een transformatie voorstelt.

Dynamische transformaties zijn transformaties die aangemaakt worden wanneer ze nodig zijn, zonder dat ervoor bijvoorbeeld een specifieke klasse is geïmplementeerd. Ze worden aangemaakt aan de hand van een beschrijving, bijvoorbeeld in XML, of aan de hand van een component, waarvan de gepaste eigenschappen kunnen achterhaald worden. Dynamische transformaties zijn eigenlijk transformaties waar geen programmeur meer aan te pas hoeft te komen. Ze worden aangemaakt in de schemerzone tussen statische en ontwerpmodus.

8.5.3 Ontwerpmodus

Context

Een transformatie bevindt zich tijdens ontwerptijd en uitvoeringstijd altijd in een bepaalde context. Die context wordt bepaald door alle eigenschappen van het graafmodel die wel tijdens ontwerptijd, maar niet tijdens uitvoeringstijd kunnen wijzigen en die zich in de onmiddellijke omgeving van de transformatie bevinden. De context kan dus enkel tijdens ontwerptijd gewijzigd worden, niet tijdens uitvoeringstijd.

De onmiddellijke omgeving van een transformatie bestaat uit de verbindingen van zijn poorten en de dataknopen die met die poorten verbonden zijn. De context ruimer definiëren maakt inkapseling (zie 8.5.7) onmogelijk.

Contextgevoelige regels kunnen waarschijnlijk beschreven worden met behulp van RDF. Ze kunnen ook rechtstreeks geprogrammeerd (hard gecodeerd) worden. Ze moeten ondubbelzinnig zijn en op elk ogenblik een transformatie beschrijven ten opzichte van zijn context. Contextgevoelige regels mogen niet afhankelijk zijn van de geschiedenis van wijzigingen aan de transformatie of aan de context. De toestand van een transformatie moet dus op elk ogenblik kunnen afgeleid worden uit de context en de contextgevoelige regels, zonder dat er rekening gehouden wordt met de geschiedenis van wijzigingen van de context.

Er is meer onderzoek nodig om precies te bepalen welke contextgevoelige regels mogelijk zijn en welke niet.

Poorten

Een poort is een ingang of een uitgang van een transformatie. Een ingang kan nooit een uitgang zijn en een uitgang nooit een ingang. Een poort behoort tot één en slechts één transformatie. Een transformatie kan meerdere poorten bevatten. Een poort kan verbonden worden met één en slechts één dataknoop.

Voor een gegeven poort moet kunnen achterhaald worden tot welke transformatie de poort behoort. Een poort wordt door de volgende drie factoren geïdentificeerd:

- De transformatie waartoe de poort behoort
- Het is ofwel een ingang ofwel een uitgang
- Een identificatie (bijvoorbeeld een URL), zodanig dat als voor twee verschillende poorten de vorige twee eigenschappen gelijk zijn, deze identificatie verschillend moet zijn.

Intern houdt een poort de volgende informatie bij:

- Metadata. Die bepaalt met welke dataknopen de poort kan verbonden worden. Een poort kan met hoogstens één dataknoop verbonden worden. Deze informatie kan niet gewijzigd worden, maar is wel contextgevoelig.
- Of de poort verbonden is met een dataknoop en zo ja, met welke. Deze informatie behoort tot de context van een transformatie.
- Of de poort facultatief is. Deze eigenschap kan door de transformatie waartoe de poort behoort, aangepast worden. Dit mag enkel gebeuren ten gevolge van een wijziging van de context. Er moeten, indien van toepassing, contextgevoelige regels bestaan die op elk moment ondubbelzinnig beschrijven in welke omstandigheden welke poorten van een transformatie facultatief zijn.

Toevoegen van een poort aan een transformatie gebeurt door de transformatie zelf. Het toevoegen van een poort kan enkel het gevolg zijn van het aanmaken van de transformatie, of van het veranderen van de context van de transformatie.

Verwijderen van een poort kan enkel door de transformatie zelf gebeuren. Het kan alleen in ontwerptijd en bovendien ten gevolge van een wijziging van de context waarin de transformatie zich bevindt. Verwijderen van een poort is onmogelijk als de poort verbonden is met een dataknoop. Merk op dat deze laatste eis een inperking is van de mogelijkheden voor contextgevoelige regels.

Het enige attribuut van een poort dat kan wijzigen is het al dan niet facultatief zijn ervan en enkel de transformatie waartoe de poort behoort kan die eigenschap wijzigen.

Facultatieve poorten

Een facultatieve poort is een poort waarvan het niet verbonden zijn met een dataknoop niet tot gevolg heeft dat de transformatie waartoe de poort behoort niet uitvoerbaar (zie 8.5.5) is. Met andere woorden: een facultatieve poort mag, maar hoeft niet verbonden te worden met een dataknoop. Om te eisen dat een poort niet mag verbonden worden, moet die poort verwijderd worden.

Zowel ingangen als uitgangen kunnen facultatief zijn.

Zichtbaarheid van poorten

Poorten kunnen volledig verborgen worden voor ontwerpers. Zij moeten kunnen weten hoe poorten kunnen geïdentificeerd worden, om verbindingen te kunnen maken. Enkel de identificatie van poorten moet dus voor ontwerpers bekend zijn.

Gegeven de identificatie van een poort, moet een ontwerper een overzicht kunnen krijgen van alle verbindingsmogelijkheden van die poort.

Ter herinnering: ontwerpers moeten niet kunnen programmeren en hebben dus bitter weinig aan bijvoorbeeld een datatype om te zien waar verbindingen kunnen gelegd worden.

Aangezien het niet strikt nodig is om poorten of hun eigenschappen openbaar te maken, is het ook wenselijk om dat niet te doen.

Ingangen

Een ingang is een poort en kan dus verbonden worden met juist één dataknoop. De richting van de eventuele verbinding loopt van de dataknoop naar de ingang.

Meestal kan een transformatie via een ingang data binnentrekken. Een ingang hoeft echter niet noodzakelijk te dienen om data naar een transformatie over te brengen.[1] Een ingang heeft een ruimere betekenis, die wordt ingevuld door de implementatie van de transformatie. Die implementatie moet conform zijn met de metadata die bij de ingang horen.

Facultatieve ingangen kunnen dienen om opties mee te geven, zoals dikwijls het geval is bij consoleapplicaties.

Uitgangen

Een uitgang is een poort. De richting van de eventuele verbinding van een uitgang loopt van de uitgang naar de dataknoop.

Meestal dient een uitgang om data af te geven aan een dataknoop. Dat hoeft echter niet altijd zo te zijn. De implementatie van een transformatie waartoe de uitgang behoort bepaalt waar een uitgang voor dient. Die implementatie moet wel conform zijn met de metadata van de uitgang.

Facultatieve uitgangen kunnen nuttig zijn als er meerdere uitgangen zijn, waarvan er mogelijk slechts één nodig is.

8.5.4 Runmodus

Gedurende de runmodus wordt er geen rekening gehouden met de poorten. Performantie is van belang, dus mogen er geen dure controles uitgevoerd worden. Dure controles kunnen bijvoorbeeld nodig zijn voor het wijzigen van een verbinding. Het controleren van contextgevoelige regels kan inefficiënt zijn en moet dus vermeden worden tijdens het "echte werk".

Het uitvoeren van een transformatie gebeurt in het eenvoudigste geval in drie stappen. Eerst worden de ingangen verwerkt. Daarna gebeurt er wat er moet gebeuren en dan worden de uitgangen verwerkt. Dat gebeurt elke keer dat de transformatie *gesweept* wordt (zie 8.9).

Het uitvoeren van transformaties hangt sterk samen met mark-and-sweep. Dat is zo belangrijk voor het framework dat er daarvoor een aparte sectie is voorzien.

8.5.5 Overgang van ontwerpmodus naar runmodus

Wat een transformatie betreft, kan er overgegaan worden van ontwerpmodus naar runmodus als de transformatie *uitvoerbaar* is. Een transformatie is uitvoerbaar als alle niet facultatieve poorten zijn verbonden met een dataknoop.

De overgang gebeurt door een methode van de transformatie op te roepen. Door het uitvoeren van die methode bereidt de transformatie zichzelf voor om uitgevoerd te worden. Eén van de handelingen die moet gebeuren is het registreren voor het ontvangen van *mark-* en *sweepevents* (zie ook 9.1.2). Andere handelingen zijn afhankelijk van datgene waarvoor de transformatie dient. Eventueel is compileren en dynamisch laden van een deel van de transformatie mogelijk.

In het geval dat een verbinding tussen een poort en een dataknoop betekent dat er data moeten doorgesluisd worden, wordt er in deze fase een referentie opgeslagen naar het gedeelte van de dataknoop dat die data effectief bevat. Op die manier worden de poorten in runmodus omzeild.

---

1. In die zin is "dataknoop" een slechte naam geworden voor datgene waar het voor staat.

De meeste van deze acties kunnen transparant gemaakt worden voor een programmeur. Een programmeur zou enkel heel specifieke acties zelf moeten kunnen implementeren.

8.5.6 Overgang van runmodus naar ontwerpmodus

Stoppen met uitvoeren van een transformatie betekent dat alle referenties naar dataobjecten uit dataknopen op `null` moeten ingesteld worden, om geheugenlekken te voorkomen.

Wat mark-and-sweep betreft, moeten alle luisteraars gederegistreerd worden.

Eventueel moeten er bronnen worden vrijgegeven, of moet het geheugen opgeruimd worden, als er geen gebruik gemaakt wordt van garbage collection.

8.5.7 Inkapselen van transformaties

Een ontwerper kan meerdere transformaties samenvoegen tot één enkele transformatie. Deze actie heet *inkapselen van transformaties*. Het resultaat van de actie is de *omhullende transformatie* of de omhullende, de oorspronkelijke transformaties worden *ingekapselde transformaties* genoemd.

Inkapselen van transformaties gebeurt in ontwerpmodus. Een verzameling transformaties kan ingekapseld worden als ze samen een graafmodel vormen dat uitvoerbaar is, of als de oorzaak voor het niet uitvoerbaar zijn kan gelokaliseerd worden in niet verbonden ingangen en uitgangen van transformaties van dat graafmodel. Eventuele data worden niet behouden.

De omhullende heeft als poorten de niet verbonden poorten van alle transformaties in het graafmodel. Ze worden geïdentificeerd door hun oorspronkelijke identificatie, eventueel aangevuld met een bijkomende identificatie als er een conflict bestaat.

Het feit dat de omhullende opgebouwd is uit ingekapselde transformaties moet transparant zijn voor ontwerpers. De omhullende moet voor alle hier besproken facetten kunnen beschouwd worden als één enkele transformatie. Contextafhankelijke eigenschappen worden afgeleid van de ingekapselde transformaties. Daarvoor moet alleen naar de 'buitenste' transformaties gekeken worden.

Als de ingekapselde transformaties staatloos zijn, dan is de omhullende dat ook. Als de omhullende *stateful* is, moet de initiële staat zodanig opgebouwd worden dat er geen gebruik gemaakt wordt van data uit ingekapselde interne dataknopen.

Er zijn verschillende voordelen aan het inkapselen van transformaties:

1. Een graafmodel kan overzichtelijker worden, doordat het minder transformaties bevat.
2. Een transformatie kan snel opgebouwd worden door inkapselen en later efficiënter geïmplementeerd worden door een programmeur.
3. Een graafmodel kan efficiënter opgebouwd worden, doordat niet meer moet gecontroleerd worden of verbindingen tussen ingekapselde transformaties kunnen gemaakt worden. Dit is echter een optimalisatietechniek die optioneel kan toegepast worden.
4. ...

In feite is het inkapselen van transformaties hetgeen soms met *Component Composition* wordt bedoeld, alleen worden de componenten zelf niet samengesteld, maar wel hun vertegenwoordigers in het graafmodel: de transformaties.

Als transformaties formeel kunnen beschreven worden, moet een formele beschrijving van de omhullende kunnen gegenereerd worden uit de formele beschrijvingen van de ingekapselde transformaties.

De instanties van ingekapselde transformaties die als instantie deel uitmaken van een omhullende, kunnen op zich niet tot een graafmodel behoren. Sterker nog: de instanties van ingekapselde transformaties behoren enkel en alleen tot hun omhullende en zijn onzichtbaar voor objecten daarbuiten.

8.6 Dataknopen

8.6.1 Beschrijving

Dataknopen zijn knopen die als hulpmiddel dienen voor het doorgeven van data of andere berichten tussen transformaties. Ze bevatten een metadataobject en een dataobject. De metadata dienen om verbindingsmogelijkheden te controleren in ontwerpmodus, de data voor het echte werk in runmodus.

De data nemen deel aan het mark-and-sweepmechanisme tijdens het uitvoeren van het graafmodel. Transformaties maken geen gebruik van referenties naar dataknopen, maar enkel van referenties naar de data in uitvoeringsmodus.

Een dataknoop kan aangemaakt worden aan de hand van een metadataobject. De data worden verkregen via een *factory* (zie 8.4).

Dataknopen kunnen nooit als luisteraar voor een *event* van een component geregistreerd worden.

8.6.2 Statische modus

Programmeurs kunnen eigenschappen van dataknopen gebruiken om contextgevoelige regels op te stellen waar transformaties aan moeten voldoen. Voor de rest mogen ze niets met dataknopen kunnen aanvangen. Ze mogen bijvoorbeeld geen dataknopen kunnen aanmaken.

8.6.3 Ontwerpmodus

Dataknopen zijn van belang in ontwerpmodus. In uitvoeringsmodus zijn enkel de data die ze eventueel bevatten belangrijk.

Verbindingsmogelijkheden

Een dataknoop kan met nul of één uitgang verbonden zijn en met nul, één of meerdere ingangen. De verbindingsmogelijkheden kunnen beperkt worden door contextgevoelige regels in de metadata. Er kan bijvoorbeeld slechts één verbinding met een ingang toegestaan worden.

Dataknopen zijn nodig. Er zou kunnen gedacht worden dat uitgangen de taken kunnen uitvoeren die nu door dataknopen worden uitgevoerd die verbonden zijn met een uitgang. Dat klopt niet, want de dataknoop moet met een andere uitgang kunnen verbonden worden terwijl de data behouden blijven.

Uitvoerbaarheid

Een dataknoop is uitvoerbaar als hij verbonden is met een uitgang, of als hij data bevat en verbonden is met een uitgang.

Metadata mogen niet op een zodanige manier opgesteld zijn dat een dataknoop nooit uitvoerbaar kan zijn.

Wijzigen van data in ontwerpmodus

Een dataknoop heeft drie mogelijke uitvoerbare staten in ontwerpmodus. De meest voorkomende staat is als verbindingsmiddel tussen twee of meer transformaties. Dan is de dataknoop met zowel een uitgang als met een of meerdere ingangen verbonden. Deze dataknopen worden *interne dataknopen* genoemd. Verder kan een dataknoop voorkomen als *constante* en als *silo*.

Constanten Een constante is een dataknoop die verbonden is met één of meerdere ingangen en niet met een uitgang. Bovendien bevat de dataknoop gegevens die bruikbaar zijn voor de transformaties waarmee hij verbonden is.

De metadata kunnen verbieden dat een dataknoop als constante voorkomt. Dat kan nuttig zijn voor "vluchtige" data, zoals iteratoren in Java.

Als een dataknoop verbonden is met een of meer ingangen, maar niet met een uitgang en geen data bevat, dan is hij niet uitvoerbaar.

De data van een constante kunnen gewijzigd worden in ontwerpmodus, maar niet in uitvoeringsmodus. Dit is een tegenstelling met de andere uitvoerbare staten van dataknopen.

Silo's Een silo is een dataknoop die met een uitgang, maar met geen enkele ingang verbonden is. Een silo is een opslagplaats voor data die bekomen worden door het uitvoeren van het graafmodel. In een latere fase zou de dataknoop kunnen verbonden worden met een ingang en de verbinding met de uitgang losgemaakt, zodat de dataknoop een constante wordt.

8.6.4 Runmodus

Dataknopen zijn passieve modules in de runmodus. Ze doen niets, behalve af en toe gewijzigd worden en gelezen worden. Ze spelen geen rol in het mark-and-sweepmechanisme.

8.7 Verbindingen

Verbindingen werden tussendoor al besproken. Hier volgt een kort overzicht.

8.7.1 Statische modus

Verbindingen komen niet voor in statische modus. Ze worden niet als object gemodelleerd. Er bestaat m.a.w. geen klasse `Verbinding`.

8.7.2 Ontwerpmodus

Er kunnen verbindingen gemaakt worden tussen dataknopen en transformaties: van transformaties naar dataknopen via de uitgangen en omgekeerd via de ingangen. Er kunnen geen verbindingen gemaakt worden tussen dataknopen onderling en ook niet tussen transformaties onderling.

Om te kunnen verbinden moeten de metadata van de poort en van de dataknoop compatibel zijn.

Een dataknoop kan één inkomende verbinding hebben en meerdere uitgaande. Een ingang kan één inkomende verbinding hebben, een uitgang één uitgaande verbinding. Een transformatie kan geen verbinding hebben die geen verbinding is van een poort van die transformatie.

Er mag geen verbinding gemaakt worden als het gevolg daarvan zou zijn dat er een lus ontstaat in het graafmodel. Het graafmodel moet altijd een DAG blijven.

8.7.3 Uitvoeringsmodus

Verbindingen kunnen niet toegevoegd, verwijderd of gewijzigd worden tijdens het uitvoeren van het graafmodel. Of ze kunnen opgevraagd worden is een implementatiedetail. Dat onmogelijk maken is extra werk.

8.8 Graafmodel

8.8.1 Beschrijving

Een graafmodel is een `BeanContext`[2] waarin transformaties kunnen samengesteld worden in een *dependency graph*. De dataknopen en transformaties die deel uitmaken van de DAG behoren tot juist één graafmodel. Het is niet zo dat dataknopen en transformaties samen met hun verbindingen het graafmodel vormen. Het graafmodel vormt een laag boven de dataknopen en transformaties.

Metamodellen

Een graafmodel is op zich een interactieve component en kan als dusdanig ingebed worden in een transformatie. Een graafmodel dat een of meerdere graafmodellen bevat in een of meer van zijn transformaties, is een *metamodel*. Een graafmodel dat vervat zit in een metamodel, is een *ingebed graafmodel*. Een graafmodel kan tegelijk een metamodel en een ingebed graafmodel zijn.

Een graafmodel en een metamodel bevinden zich niet noodzakelijk in dezelfde modus. Het is bijvoorbeeld mogelijk dat het metamodel zich in uitvoeringsmodus bevindt en een ingebed model in ontwerpmodus.

Let op het verschil tussen metamodellen en ingekapseling van transformaties. Een omhullende is een transformatie, terwijl een ingebed graafmodel een graafmodel is. Er moet nog een (bij voorkeur dynamische) transformatie aangemaakt worden opdat een graafmodel zou kunnen ingebed worden in een ander graafmodel.

8.8.2 Statische modus

Programmeurs kunnen niets aanvangen met het graafmodel. Ze kunnen hooguit het opbouwen ervan hard coderen, wat eigenlijk neerkomt op het maken van een automatische ontwerper.

8.8.3 Ontwerpmodus

Over het algemeen moeten de operaties voor het aanpassen van het graafmodel eenvoudig zijn. In deze sectie worden die operaties besproken op het niveau van het graafmodel.

Er moet van uitgegaan worden dat meerdere ontwerpers tegelijk het graafmodel kunnen aanpassen. Het graafmodel bevindt zich dus in een *multithreaded* omgeving. Quasi alle hierna besproken acties moeten gesynchroniseerd worden. Dat wordt verder niet meer vermeld. Een implementatie van `BeanContext` biedt voldoende mogelijkheden voor synchronisatie.

Er kunnen ruwweg drie niveaus aangeduid worden in het framework:

- Het hoogste niveau is het graafmodel

2. Gelijkaardige concepten gelden natuurlijk voor andere componententechnologieën. Deze opmerking geldt voor heel deze sectie.

- Daaronder bevinden zich de dataknopen en transformaties, waartussen verbindingen kunnen gelegd worden.
- Op het laagste niveau bevinden zich de metadata, die deel uitmaken van de dataknopen en van de poorten.

Waar welk niveau voor verantwoordelijk is wordt verder niet besproken. Het is de bedoeling dat ontwerpers aan de slag kunnen door enkel methodes uit het graafmodel te gebruiken.

De volgende secties beschrijven wat er met een graafmodel kan gedaan worden in ontwerpmodus.

8.8.4 Toevoegen van transformaties

Transformaties kunnen tot hoogstens één graafmodel behoren. Het zijn beans die de interface `BeanContextChild` implementeren.

Interne werking

Er moet eerst gecontroleerd worden of het graafmodel zich in ontwerpmodus bevindt.

Als een referentie naar een transformatie wordt doorgegeven, moet gecontroleerd worden of die transformatie al tot een `BeanContext` behoort. Het beancontextframework regelt eventueel het verplaatsen van de transformatie.

Het graafmodel houdt een collectie van transformaties bij die het bevat. Toevoegen van een transformatie aan het graafmodel betekent toevoegen van die transformatie aan die collectie.

Er moet na toevoegen snel kunnen gecontroleerd worden of een transformatie al dan niet tot het graafmodel behoort. Een `HashSet` is een goede datastructuur voor het bijhouden van de collectie transformaties, maar dat is eigenlijk een implementatiedetail.

Initieel is een transformatie met geen enkele dataknoop verbonden.[3]

Interface voor ontwerper

Een ontwerper moet bij voorkeur op verschillende manieren een transformatie kunnen toevoegen aan het graafmodel:

- Een instantie van een transformatie toevoegen
- De klasse van een transformatie doorgeven
- Een beschrijving van de transformatie doorgeven
- Een URL van een beschrijving van de transformatie
- ...

Al deze mogelijkheden kunnen geïmplementeerd worden door een overladen methode van het graafmodel.

8.8.5 Verwijderen van transformaties

Interne werking

- Er wordt een referentie naar een te verwijderen transformatie doorgegeven.
- Er moet eerst gecontroleerd worden of het graafmodel zich in ontwerpmodus bevindt.
- Er moet gecontroleerd worden of de transformatie tot het graafmodel behoort. Als dat niet zo is, gebeurt er verder niets.

3. Dit is een verbetering ten opzichte van Alpha.

- Er moet gecontroleerd worden of de transformatie verbindingen heeft met dataknopen. Als dat zo is, worden die verbindingen eerst verwijderd.
- Uiteindelijk wordt de transformatie effectief verwijderd.

Interface voor ontwerper

Als een ontwerper beschikt over een referentie naar een transformatie, dan kan hij die transformatie verwijderen. Het graafmodel moet dus een methode hebben, met een referentie naar een transformatie als argument, waarmee een transformatie kan verwijderd worden.

8.8.6 Wijzigen van transformaties

Wijzigen van een transformatie kan enkel door zijn context te wijzigen en is dus geen losstaande operatie.

8.8.7 Opvragen van de transformaties

Een ontwerper moet kunnen weten welke transformaties er in een graafmodel zitten.

Interne werking

Er wordt een collectie van referenties naar de transformaties doorgegeven, op een zodanige manier dat de ontwerper via die collectie het graafmodel niet kan wijzigen.

In Java kan dit in $O(1)$ met behulp van een van de volgende methoden:

- `Collections.unmodifiableSet()` voor een `Set`
- `Collections.unmodifiableCollection()` voor een `Collection`

Een van deze methoden kan uitgevoerd worden op de collectie die de transformaties bevat.

Een veiliger manier is om geen referenties naar de transformaties zelf, maar enkel naar *proxies* terug te geven (zie 9.7).

Interface voor ontwerper

Er moet een methode voorzien worden waarmee een ontwerper een collectie van transformaties kan opvragen.

8.8.8 Wijzigingen van een transformatie

Ontwerpers moeten op de hoogte kunnen gehouden worden van wijzigingen van transformaties, ten gevolge van wijzigingen van hun context.

Interne werking

Ontwerpers kunnen zich registreren als luisteraar voor wijzigingen van transformaties. Als luisteraar kunnen ze te weten komen wanneer poorten worden toegevoegd of verwijderd en wanneer ze wel of niet facultatief worden.

Interface voor ontwerper

Ontwerpers kunnen zich interesseren voor het wijzigen van een specifieke transformatie, of voor het wijzigen van alle transformaties in het graafmodel. Zowel de transformaties als het graafmodel moeten dus bron zijn van *events* die wijzigingen aan transformaties adverteren.

8.8.9 Op de hoogte gehouden worden van toevoegen en verwijderen van transformaties

Het is mogelijk dat meerdere ontwerpers hetzelfde graafmodel wijzigen. Alle ontwerpers moeten op de hoogte gehouden worden van hoe het graafmodel op elk moment is opgebouwd.

Interne werking

Dit wordt geregeld door de implementatie van `BeanContext`.

Interface voor ontwerper

Er zijn *events* die het toevoegen of verwijderen van transformaties adverteren.

8.8.10 Toevoegen van dataknopen

Interne werking

Dataknopen kunnen enkel toegevoegd worden als dat toevoegen samengaat met het maken van een verbinding tussen die dataknoop en een poort. Het toevoegen van een dataknoop en het verbinden ervan met een poort moet als een atomair geheel kunnen beschouwd worden.

Dataknopen worden niet in een aparte datastructuur bijgehouden, maar er wordt vanuit de transformaties naar gerefereerd via de verbindingen.

Het graafmodel registreert zich bij de pas toegevoegde dataknoop voor een *event* dat optreedt op het moment dat de laatste verbinding van de dataknoop verwijderd wordt. In talen met garbage collection is dat niet nodig.

Toevoegen van constanten. Een ontwerper kan dataknopen toevoegen, waarvan hij beweert dat het constanten zullen worden. De dataknoop wordt aangemaakt aan de hand van de metadata van de ingang waarmee de constante zal verbonden worden. De dataknoop blijft zichtbaar voor de ontwerper zolang hij niet wordt verbonden met een uitgang.

Het op deze manier toevoegen van dataknopen is enkel mogelijk als de desbetreffende metadata dat toestaan.

Toevoegen van silo's. Een ontwerper kan silo's toevoegen. De silo wordt verbonden met een welbepaalde uitgang en krijgt dezelfde metadata als die uitgang. Ze blijven zichtbaar zolang het geen interne dataknopen worden, of totdat ze verwijderd worden.

Het aanmaken van een silo is enkel mogelijk als dat is toegestaan door de desbetreffende metadata.

Toevoegen van interne dataknopen is transparant voor de ontwerper.

Interface voor ontwerper

Er moet een methode bestaan waarmee constanten kunnen toegevoegd worden. Als argumenten voor die methode zijn een referentie naar een transformatie nodig, samen met de identificatie van een ingang.

Analoog moet er een methode zijn waarmee silo's kunnen toegevoegd worden. Als argumenten heeft die methode een referentie naar een transformatie en de identificatie van een uitgang.

Er wordt in geen geval een rechtstreekse referentie naar de aangemaakte dataknoop teruggegeven.

8.8.11 Verwijderen van dataknopen

Interne werking

Automatisch verwijderen. Een dataknoop wordt automatisch verwijderd als er geen verbindingen meer zijn met die dataknoop.

Het graafmodel is geregistreerd als luisteraar voor *events* die adverteren wanneer de laatste verbinding van een dataknoop verwijderd wordt.

Als reactie op dat *event* verwijdert het graafmodel de dataknoop uit het geheugen. In talen met een garbage collector is het voldoende dat er geen referenties naar de dataknoop meer bestaan en wordt heel dit systeem automatisch verwezenlijkt.

Om de automatische verwezenlijking in talen met een garbage collector vlot te laten verlopen, mag geen enkel onderdeel van een dataknoop tijdens ontwerptijd geregistreerd zijn als luisteraar voor een *event*. Dat zou immers inhouden dat er nog referenties bestaan naar de dataknoop, of naar onderdelen ervan. Meer specifiek kan er best voor gezorgd worden dat alle *mark-* en *sweepevents* na het uitvoeren van het graafmodel gederegistreerd worden.

Expliciet verwijderen. Constanten en silo's zijn zichtbaar voor ontwerpers en zouden expliciet kunnen verwijderd worden. Achter de schermen zou dit kunnen neerkomen op het verwijderen van de verbindingen naar de dataknoop.

Interface voor ontwerper

Een methode voor het expliciet verwijderen van constanten en silo's is niet strikt nodig, omdat verbindingen kunnen verwijderd worden.

8.8.12 Wijzigen van dataknopen

Zichtbare dataknopen zijn dataknopen die ofwel met geen enkele ingang, ofwel met geen enkele uitgang verbonden zijn. Enkel zichtbare dataknopen kunnen expliciet gewijzigd worden. Wijzigen van verbindingen wordt later besproken. Data wijzigen kan voor dataknopen zonder voorlopers.

Interne werking

Data toevoegen. Gebeurt door data toe te voegen aan het dataobject van de dataknoop. De toe te voegen data worden eerst gecontroleerd aan de hand van de metadata.

Metadataobjecten zijn, zoals reeds geformuleerd, *immutable*. De metadata van een constate moeten kunnen opgevraagd worden, om componenten te kunnen voorzien waarin de data geëditeerd kunnen worden.

Voor alle duidelijkheid: controle van data aan de hand van metadata gebeurt enkel in ontwerptijd. Tijdens het uitvoeren van het graafmodel moet alles wat sneller gaan. Deze controle wordt dan achterwege gelaten. Toch zijn de data zeker betrouwbaar, want de metadata werden gecontroleerd tijdens het maken van verbindingen (zie verder).

Data verwijderen. Dat gebeurt door data te verwijderen uit het dataobject van de dataknoop. Hierdoor wordt de dataknoop en dus ook het graafmodel, onuitvoerbaar.

Data wijzigen. Komt op hetzelfde neer als toevoegen van data. Hier moet een ontwerpkeuze gemaakt worden. Als de nieuwe data niet kunnen toegelaten worden, kunnen ofwel de oude data behouden blijven, ofwel verwijderd worden. Een derde mogelijkheid is het wijzigen van data niet toe te laten, maar enkel het toevoegen en verwijderen.

Data opvragen. Dit wijzigt de dataknoop niet, maar moet ook mogelijk zijn. Componententechnologieën kunnen de mogelijkheid bieden de opgevraagde data op een gepaste manier weer te geven.

Interface voor ontwerper

De hierna beschreven methoden moeten een dataknoop kunnen identificeren. Dat kan door de combinatie van een referentie naar een transformatie en de identificatie van de ingang die met de dataknoop verbonden is. Er moet voor gezorgd worden dat er nooit een rechtstreekse referentie naar een dataknoop wordt vrijgegeven.

Toevoegen. Er moet een methode voorzien worden om data toe te voegen. Indien mogelijk in de programmeertaal van het framework, kan er gebruik gemaakt worden van templates of Generics om aan die methode een argument van het juiste type door te geven. Dit type omvat echter waarschijnlijk niet alle regels van de metadata van de dataknoop. De controle van de data aan de hand van de metadata kan dus in geen geval achterwege gelaten worden.

Er moet een methode voorzien worden om de metadata van een constante op te vragen.

Verwijderen. Er moet een methode voorzien worden om de data uit een constante te verwijderen.

Wijzigen. Er kan eventueel een methode voorzien worden voor het wijzigen van de data van een constante. Eventueel gaat het hier om dezelfde methode als voor toevoegen van data.

Opvragen. Er moet een methode zijn voor het weergeven van data. Merk op dat dit risico's kan inhouden. Indien mogelijk moeten de data *immutable* zijn, of moet er een kopie teruggegeven worden.

8.8.13 Opvragen van dataknopen

Er kunnen nooit rechtstreeks referenties naar dataknopen verkregen worden. Slechts van enkele dataknopen kan er een glimp opgevangen worden: silo's en dataknopen die niet met een ingang verbonden zijn.

Die glimp kan bestaan uit de identificatie van de poorten die met de dataknopen verbonden zijn.

Interne werking

Via diepte- of breedte-eerst zoeken. Er wordt – zeker bij gebruik van garbage collection – geen collectie van zichtbare dataknopen bijgehouden, Referenties naar dataknopen zijn te vermijden.

Eventueel kunnen de transformaties in partiële orde gerangschikt worden, waardoor dit iets sneller gaat.

Interface voor ontwerper

Er moet een methode voorzien worden voor het opvragen van de silo's. De teruggeefwaarde van die methode bestaat uit de identificatie van de uitgangen die met de silo's verbonden zijn.

Er moet een methode voorzien worden voor het opvragen van de dataknopen die niet verbonden zijn met een uitgang. De teruggeefwaarde van die methode kan bestaan uit de collecties van ingangen die met die dataknopen verbonden zijn.

8.8.14 Op de hoogte gehouden worden van een wijziging van een dataknoop

Wijzigen van staat van een zichtbare dataknoop moet via een *event* van het graafmodel kunnen waargenomen worden. Hetzelfde geldt voor wijzigen van data (indien mogelijk) en verwijderen van een zichtbare dataknoop.

Een luisteraar voor een dergelijk *event* wordt gederegistreerd als de dataknoop onzichtbaar wordt, of verwijderd.

Interface voor ontwerper

Via het graafmodel moeten luisteraars zich voor *events* kunnen registreren bij silo's. Die *events* adverteren het veranderen van de staat en het verwijderen van de dataknoop. Op het ogenblik dat de staat verandert wordt er eerst een *event* gestuurd naar alle luisteraars, waarna de luisteraars gederegistreerd worden. Analoog bij verwijderen.

Via het graafmodel moeten luisteraars zich voor *events* kunnen registreren bij dataknopen die niet verbonden zijn met een uitgang. Dergelijke *events* adverteren het toevoegen, verwijderen en wijzigen van de data en het veranderen van de staat en het verwijderen van de dataknoop. Als de dataknoop wordt verbonden met een uitgang, of verwijderd wordt, worden alle luisteraars gederegistreerd.

Deze *events* moeten als bron het graafmodel vermelden. Als bijkomende informatie moeten ze duidelijk maken van welke dataknoop het *event* afkomstig is. Via een *event* mag er geen referentie naar een dataknoop bekomen worden.

8.8.15 Op de hoogte gehouden worden van wijzigingen van dataknopen

Dit is in het algemeen onmogelijk. Dataknopen moeten zo veel mogelijk verborgen worden. Indien mogelijk moeten ze als *package private* klassen gedeclareerd worden, eventueel als *inner class* van de klasse die het graafmodel voorstelt. Voor zichtbare dataknopen is het op de hoogte gehouden worden wel in beperkte mate mogelijk, zoals beschreven in de vorige paragraaf.

Er kunnen in het graafmodel methoden voorzien worden waarmee luisteraars zich kunnen registreren en deregistreren voor alle waarneembare wijzigingen van dataknopen.

8.8.16 Toevoegen van verbindingen

In geen geval mag een nieuwe verbinding een lus veroorzaken in de *dependency graph*. Een speciaal geval is het verbinden van een uitgang met een ingang van dezelfde transformatie, wat onmogelijk moet zijn. Dit kan gecontroleerd worden door diepte-eerst zoeken. Die controle moet uitgevoerd worden vóórdat de verbinding gemaakt wordt.

Een verbinding mag maar één keer toegevoegd worden. Een tweede keer toevoegen van een verbinding heeft geen effect. Merk op dat het wel mogelijk is twee verbindingen te maken tussen dezelfde dataknoop en transformatie, maar via verschillende poorten.

Mogelijke verbindingen

Met een dataknoop die niet met een uitgang verbonden is. Dit werd al gedeeltelijk besproken, bij het aanmaken van zo'n dataknoop. De dataknoop kan bijkomend met andere ingangen verbonden worden. Voor het maken van zo'n verbinding moeten de metadata van de dataknoop en van de ingang compatibel zijn. De dataknoop kan geïdentificeerd worden door een ingang waarmee deze verbonden is.

Een dergelijke dataknoop verbinden met een uitgang heeft als gevolg dat de eventuele data die de dataknoop bevat verwijderd worden. Zonder het graafmodel uit te voeren is het immers onmogelijk te controleren of die data geldig kunnen zijn.

Tussen een uitgang en een ingang. Een ontwerper kan ervoor kiezen om een uitgang met een ingang te verbinden. Als de uitgang nog met geen enkele dataknoop verbonden was, wordt er een nieuwe dataknoop aangemaakt, met de metadata van de uitgang. De ingang en de uitgang worden dan met de nieuwe dataknoop verbonden, na controle van de metadata. Als de uitgang al met een dataknoop verbonden was, dan wordt de ingang met die dataknoop verbonden, na controle van de metadata.

Als de bewuste ingang al verbonden is met een dataknoop, gaat het maken van de nieuwe verbinding niet door.

Met een silo. Verbinden van een uitgang met een silo gebeurt door toevoegen van een silo en werd al besproken.

Een silo kan met een ingang verbonden worden, waardoor de staat ervan verandert. De data blijven behouden. Hiervoor moet geen aparte methode voorzien worden, want de hiervoor beschreven methode volstaat.

Een silo kan niet met een bijkomende uitgang verbonden worden.

Interne voorstelling van verbindingen

Intern wordt een verbinding voorgesteld door een wederzijdse referentie tussen een dataknoop en een poort. Aanmaken van een verbinding bestaat uit het opslaan van twee referenties.

Interface voor ontwerper

Er moet een methode bestaan waarmee een ingang en een uitgang met elkaar kunnen verbonden worden.

Er moet een methode zijn waarmee een verbinding tussen een ingang en een dataknoop die met geen enkele uitgang verbonden is, kan gemaakt worden.

8.8.17 Verwijderen van verbindingen

Verwijderen van verbindingen is eenvoudig. Het bestaat uit het verwijderen van de wederzijdse referenties tussen een dataknoop en een poort. Alle bestaande verbindingen kunnen verwijderd worden.

Als de metadata van een dataknoop niet toestaan dat die een constante wordt, dan worden alle verbindingen van de dataknoop verwijderd als de verbinding met de uitgang verwijderd wordt.

Analoog, als de metadata niet toestaan dat de dataknoop een silo wordt, dan wordt de verbinding met een uitgang verwijderd, onmiddellijk na het verwijderen van de laatste verbinding met een ingang.

Interface voor ontwerper

Er moet een methode bestaan om de verbinding tussen een ingang en een dataknoop die niet verbonden is met een uitgang te verwijderen. Als argument moet enkel de identificatie van de ingang worden meegegeven. Het is mogelijk dat de dataknoop wordt verwijderd als gevolg van het uitvoeren van deze methode.

Er moet een methode zijn waarmee de verbinding tussen een silo en een uitgang kan worden verwijderd. Als argument van deze methode is enkel de identificatie van de uitgang nodig. Dit is exact hetzelfde als expliciet verwijderen van een silo.

Er moet een methode zijn om verbindingen tussen een ingang en een uitgang te verwijderen. De interne dataknoop blijft bestaan na uitvoeren van deze methode. Enkel de verbinding tussen de dataknoop en de bewuste ingang wordt verwijderd. Het is mogelijk dat de dataknoop hierdoor een silo wordt. Aangezien de dataknoop data kan bevatten die nodig kunnen zijn voor het uitvoeren van het graafmodel, mag deze niet zonder meer verwijderd worden.

8.8.18 Wijzigen van verbindingen

Wijzigen van een verbinding komt neer op het achtereenvolgens verwijderen en toevoegen van een verbinding. De verwijderde verbinding is niet dezelfde als de toegevoegde. Ofwel de poort, ofwel de dataknoop is voor beide verbindingen dezelfde.

Interface voor ontwerper

Het aanbieden van methoden voor het wijzigen van verbindingen is enkel nuttig in een omgeving met concurrerende ontwerpers.

8.8.19 Opvragen van verbindingen

De verbindingen worden in de graaf bijgehouden door de referenties waaruit ze bestaan. De graaf kan dus in diepte- of breedte-eerst volgorde overlopen worden om alle verbindingen te zoeken.

Een verbinding kan in een tijdelijke structuur – die geen referenties naar dataknopen, maar wel identificaties van poorten bevat – teruggeven worden. Deze tijdelijke structuur kan hergebruikt worden voor het adverteren van het toevoegen of verwijderen van verbindingen.

Interface voor ontwerper

Een ontwerper kan een collectie of (bij voorkeur) iterator opvragen van de verbindingen van het graafmodel.

8.8.20 Op de hoogte gehouden worden van de staat van een verbinding

Verbindingen hebben geen staten en worden zelfs niet als een object voorgesteld, dus is dit niet zinvol.

8.8.21 Op de hoogte gehouden worden van wijzigingen in de verbindingen

Wat betreft het wijzigen van verbindingen, kan dat achteraf voorgesteld worden als een combinatie van het verwijderen en toevoegen van een verbinding (in die volgorde). Eventueel kan het wijzigen van een verbinding ook als een aparte actie gezien worden.

Interne werking

Er wordt gebruik gemaakt van dezelfde tijdelijke datastructuur als voor het opvragen van de verbindingen.

Tijdelijke objecten zorgen in omgevingen met een *generational garbage collector* (zoals moderne Java Virtual Machines) nagenoeg niet voor overlast en zijn zelfs te verkiezen boven het continu voorstellen van dezelfde informatie als object.

Interface voor ontwerper

Luisteraars kunnen zich registreren voor *events* die het toevoegen en verwijderen van verbindingen in het graafmodel adverteren.

8.8.22 Runmodus

Dit wordt uitvoerig besproken in 8.9.

8.8.23 Overgang van ontwerpmodus naar runmodus

Interne werking

Het graafmodel moet uitvoerbaar zijn. Als dat zo is, wordt er een lock op ingesteld. Vanaf dat ogenblik kunnen ontwerpers het graafmodel niet meer wijzigen. Er moet nu opnieuw gecontroleerd worden of het graafmodel uitvoerbaar is (*double-checked locking*).

Alle transformaties worden overlopen. Van alle transformaties wordt de initialisatiemethode opgeroepen. Als dat gebeurd is, is het graafmodel klaar om uitgevoerd te worden.

Uitvoeren van een graafmodel kan een dure operatie zijn en wordt in een aparte thread uitgevoerd, om een eventuele GUI of andere controlestructuur niet onhandelbaar te maken.

Interface voor ontwerper

De ontwerper moet een methode ter beschikking hebben waarmee het graafmodel kan uitgevoerd worden. Aangezien het om een overgang gaat, zou ook kunnen gezegd worden dat een eindgebruiker een methode nodig heeft om het graafmodel uit te voeren. Beide benaderingen komen op hetzelfde neer.

8.8.24 Uitvoerbaarheid van het graafmodel

Het graafmodel is uitvoerbaar als alle dataknopen en alle transformaties uitvoerbaar zijn. Na elke wijziging aan het graafmodel wordt dit gecontroleerd.

Om te vermijden dat er veel werk moet gebeuren na elke wijziging van het graafmodel, worden de transformaties die niet uitvoerbaar zijn, of die verbonden zijn met een dataknoop die niet uitvoerbaar is, opgeslagen in een hashtabel. Als die hashtabel leeg is, is het graafmodel uitvoerbaar. Na elke wijziging aan het graafmodel moet gecontroleerd worden of die hashtabel moet gewijzigd worden.

Interface voor ontwerper

Een ontwerper moet kunnen controleren of het graafmodel uitvoerbaar is.

Als het graafmodel niet uitvoerbaar is, moet een ontwerper kunnen controleren waarom dat zo is. Hij moet een collectie van alle problemen met de uitvoerbaarheid kunnen opvragen van het graafmodel. Die problemen kunnen eenvoudig beschreven worden in tijdelijke objecten.

Een ontwerper moet op de hoogte kunnen gehouden worden van de uitvoerbaarheid van een graafmodel. Dat kan door bij het graafmodel een luisteraar te registreren die wijzigingen van de uitvoerbaarheid van het graafmodel adverteert.

8.8.25 Overgang van runmodus naar ontwerpmodus

Deze overgang gebeurt ofwel automatisch, ofwel moet het uitvoeren expliciet gestopt worden door een ontwerper of een eindgebruiker.

De manier waarop gestopt wordt is afhankelijk van het soort transformaties dat in het graafmodel zit (zie 7.1).

Interne werking

Alle transformaties worden overlopen en hun methode voor het stoppen met uitvoeren wordt opgeroepen. De transformaties regelen het opruimen van eventregistraties, ook bij de dataknopen. Er moet ook zeker voor gezorgd worden dat een transformatie in ontwerptijd geen referentie bevat naar een dataobject van een dataknoop.

Interface voor ontwerper of eindgebruiker

Er moet kunnen gecontroleerd worden wanneer het graafmodel uitgevoerd wordt en wanneer niet. Er moeten zich dus luisteraars kunnen registreren voor een *event* die dat adverteert.

In sommige gevallen moet het graafmodel expliciet kunnen gestopt worden door een ontwerper of een eindgebruiker. Hiervoor moet een methode voorzien worden. In een `BeanContext` kan dat de methode `setDesignTime(boolean)` zijn.

8.9 Mark-and-sweep

Mark-and-sweep is het mechanisme dat wordt toegepast om te bepalen welke transformaties op welk moment moeten uitgevoerd worden. Er zijn verschillende mogelijkheden om dit mechanisme te implementeren. Hier worden enkel de specificaties vermeld waaraan de implementaties moeten voldoen.

8.9.1 Geldigheid van transformaties

In runmodus kunnen transformaties en dataknopen geldig of ongeldig zijn. Een transformatie kan alleen geldig zijn als de dataknopen die ermee zijn verbonden aan de ingangen geldig zijn. Bovendien is een transformatie enkel geldig na uitvoeren ervan. Een transformatie kan nooit uitgevoerd worden als de dataknopen die aan de ingangen ermee verbonden zijn ongeldig zijn.

8.9.2 Geldigheid van dataknopen

Dataknopen spelen geen rol in uitvoeringsmodus. Het zijn enkel de dataobjecten die ertoe behoren die een rol spelen. Dataobjecten van constanten zijn altijd geldig. Dataobjecten van dataknopen die met een uitgang van een transformatie verbonden zijn, zijn geldig nadat die transformatie is uitgevoerd en totdat die transformatie ongeldig wordt. De geldigheid van een dataobject in een (niet constante) dataknoop wordt dus bepaald door de transformatie waarvan een uitgang verbonden is met die dataknoop.

> In Alpha maken de dataobjecten deel uit van het mark-and-sweepmechanisme. Dat is niet strikt nodig. In Alpha houdt een transformatie een collectie van ongeldige dataobjecten aan zijn ingangen bij. De geldigheid van die dataknopen wordt bepaald door het uitvoeren van de transformaties waarvan een uitgang met die dataknopen is verbonden. Er kan dus even goed een collectie van transformaties bijgehouden worden. Constanten bevatten altijd geldige data in uitvoeringsmodus.

8.9.3 Mark

Als een transformatie een *mark* ontvangt, wordt hij ongeldig. Een transformatie verstuurt een mark naar zijn opvolgers voordat, tijdens of nadat hij wordt uitgevoerd en na ontvangen van een andere mark.

Een ontvangen mark wordt door een transformatie verder gepropageerd naar zijn opvolgers. Als een transformatie ongeldig wordt, dan moeten al zijn opvolgers ongeldig worden. De volgorde waarin dit gebeurt wordt beschouwd als een implementatiedetail.

Een ongeldige transformatie die een mark ontvangt, moet die mark niet verder propageren, omdat de opvolgers van een ongeldige transformatie ook ongeldig zijn, of dat binnenkort zullen worden.

Eventueel moet het uitvoeren van een transformatie stopgezet worden als die transformatie een mark ontvangt. Voor eenvoudige transformaties die niet veel tijd vragen om uitgevoerd te worden zal dit minder nuttig zijn.

Een mark kan eventueel slechts een gedeelte van de data in de dataknopen die met een uitgang van een transformatie verbonden zijn, ongeldig maken. Dit is in veel gevallen efficiënter dan alle data te moeten herberekenen. Deze mogelijkheid moet kunnen afgeleid worden uit de metadata.

8.9.4 Sweep

Voordat een transformatie kan uitgevoerd worden, moet gecontroleerd worden of zijn voorlopers geldig zijn. Dat kan door ongeldige voorlopers bij te houden in een collectie. Hoe dat precies gebeurt, is een implementatiedetail.

Ongeldige voorlopers kunnen geldig gemaakt worden door het sturen van een *sweep*. Na ontvangen van een sweep, wordt geprobeerd de transformatie uit te voeren. Als dat onmogelijk is, doordat er ongeldige voorlopers zijn, wordt de sweep verder gepropageerd naar de voorlopers.

Uitvoeren van een transformatie bestaat uit achtereenvolgens voorbereidend werk, het effectief uitvoeren van de transformatie en werk achteraf.

Voorbereidend werk kan bestaan uit data ophalen uit:

- dataobjecten van dataknopen die met de ingangen verbonden zijn;
- dataobjecten van dataknopen die met de uitgangen verbonden zijn;
- componenten waarvan de transformatie een vertegenwoordiger is.

Uitvoeren van een transformatie mag niet direct tot gevolg hebben dat data in een van de dataknopen die met de ingangen verbonden zijn, gewijzigd worden.

Impulsen om een sweep te genereren kunnen komen van buiten het graafmodel, bijvoorbeeld voor het weergeven van een grafische component. Ze kunnen ook komen vanuit transformaties zelf. Er kan bijvoorbeeld een transformatie gemaakt worden die altijd onmiddellijk na een mark een sweep verstuurt naar zichzelf.

8.9.5 Monitoren van de uitvoering

Externe listeners kunnen zich registreren voor *mark*- en *sweepevents*. Die *events* adverteren het sturen van marks en sweeps. Ook het geldig en ongeldig worden van transformaties kan door *events* geadverteerd worden.

Voor interne marks en sweeps kunnen dezelfde soort *events* gebruikt worden, maar er moet onderscheid kunnen gemaakt worden tussen interne marks en sweeps en *events* waarop externe listeners geregistreerd zijn. Referenties die gemaakt worden voor interne doeleinden, moeten bij voorkeur gemaakt worden vlak voor het uitvoeren en na het uitvoeren vewijderd worden.

8.9.6 Uitvoeren van het graafmodel

Als er in het graafmodel geen monitors en geen interactieve transformaties voorkomen, wordt mark-and-sweep in batchmode uitgevoerd. Uitvoeren begint met het ongeldig maken van alle transformaties, door het sturen van een mark naar de transformaties zonder voorlopers. Het uitvoeren stopt automatisch als alle transformaties geldig zijn. Dat gebeurt door het sturen van een sweep naar alle transformaties zonder opvolgers.

Het graafmodel kan ook in interactieve modus uitgevoerd worden. Dat is het geval als het graafmodel monitors of interactieve transformaties bevat. In dat geval stopt het uitvoeren van het graafmodel niet, ook niet als alle transformaties geldig zijn. Uitvoeren van het graafmodel in interactieve modus begint ook met het ongeldig maken van alle transformaties.

8.10 Terugkoppeling

8.10.1 Probleemstelling

De mogelijkheid bestaat dat er in een bepaald graafmodel terugkoppeling optreedt. Dat betekent dat tijdens het uitvoeren van het graafmodel transformaties uitgevoerd worden ten gevolge van hun eigen acties.

Terugkoppeling is onmogelijk volledig uit te sluiten, dus moeten er manieren gevonden worden om ermee om te gaan, in plaats van het tot elke prijs proberen te vermijden. Het treedt altijd op in de uitvoeringsmodus.

Er zijn verschillende manieren waarop terugkoppeling kan optreden, de ene wat subtieler dan de andere. Twee transformaties kunnen dezelfde component vertegenwoordigen, waardoor ze altijd tegelijk ongeldig worden. Ze kunnen ook beiden componenten vertegenwoordigen, waarbij de ene component wijzigingen teweegbrengt in de andere.

Een ontwerper is zich mogelijk niet bewust van het feit dat terugkoppeling zal optreden in een graafmodel. Indien mogelijk, kan hij er rekening mee houden en het gebruiken in zijn voordeel. Een manier om opzettelijk terugkoppeling teweeg te brengen worden besproken in het volgende hoofdstuk.

Het mogelijk optreden van terugkoppeling moet zo goed mogelijk gedocumenteerd worden, zodat ontwerpers hun voorzorgen kunnen nemen. Annotaties kunnen ook automatische ontwerpers in staat stellen om te gaan met terugkoppeling.

Hoofdstuk 9

Uitbreidingen en toepassingen

Er zijn diverse uitbreidingsmogelijkheden voor het formeel model. Deze zijn nog niet in de praktijk gebracht. Het is dus mogelijk dat het implementeren van al deze uitbreidingen problemen oplevert. Elke uitbreiding moet rekening houden met andere uitbreidingen. Het zal waarschijnlijk niet makkelijk zijn om alle uitbreidingen in hetzelfde model te gieten.

9.1 Strategy Pattern

Het *Strategy Pattern* kan gebruikt worden om bepaalde facetten van de implementatie van ***Longbow*** aan te passen tijdens ontwerptijd. Een strategie is een implementatie van een bepaald facet. Een ontwerper kan controle krijgen over strategieën, of ze kunnen op een andere manier geconfigureerd worden. Het wijzigen van strategieën tijdens uitvoeringstijd is in de meeste gevallen waarschijnlijk moeilijk te realiseren.

Er kan voor veel verschillende aspecten mogelijk gemaakt worden dat er verschillende strategieën gebruikt worden. Het kan zijn dat de strategieën elkaar nodig hebben om goed te kunnen functioneren, maar ze mogen niet in elkaars vaarwater zitten. Voor deze toepassingen is een goed ontwerp noodzakelijk.

Voor meer informatie over het *Strategy Pattern* wordt verwezen naar de uitgebreide literatuur die over *design patterns* bestaat. [Freeman et al., 2004] is een goede inleiding, met een duidelijke uitleg over het *Strategy Pattern* in het eerste hoofdstuk.

De subsecties bespreken enkele mogelijke gevallen waarvoor strategieën kunnen gebruikt worden.

9.1.1 Threads

Interactieve transformaties worden in een aparte thread uitgevoerd. Het uitvoeren van het graafmodel gebeurt ook in een andere thread dan het wijzigen ervan. Het doorgeven van marks en sweeps kan in dezelfde thread als het uitvoeren van transformaties, of er kan een systeem voorzien worden, gelijkaardig aan de EDT (*Event Dispatching Thread*) in AWT (Abstract Windowing Toolkit).

Er kan een *threadpool* voorzien worden waaruit de nodige threads gehaald worden, of threads kunnen aangemaakt worden wanneer ze nodig zijn. Er zijn experimenten nodig om na te gaan wat de beste manier is om te werken met threads in bepaalde situaties. Toepassen van het *Strategy Pattern* voor verschillende manieren om met threads om te gaan is een goede manier om een vergelijkend onderzoek uit te voeren voor verschillende implementaties.

Zo'n vergelijkend onderzoek zou kunnen uitgevoerd worden door een metamodel. Om dat mogelijk te maken, moeten verschillende graafmodellen die tegelijk worden uitgevoerd, verschillende strategieën kunnen gebruiken. Als het *Strategy Pattern* op een goede manier geïmplementeerd wordt, is dat geen probleem.

9.1.2 Mark-and-sweep

Er zijn verschillende mogelijkheden voor de implementatie van mark-and-sweep. De prioriteit voor het ongeldig maken van opvolgers kan verschillen. Er kan een intelligentere manier gevonden worden om de volgorde te bepalen waarin transformaties kunnen uitgevoerd worden. Er moet weer vergelijkend onderzoek mogelijk zijn om de beste strategie voor een bepaalde opbouw van het graafmodel te bepalen.

In Alpha werd de implementatie van mark-and-sweep al apart gehouden van die van de transformaties. Een dergelijke manier van werken maakt de implementatie van het *Strategy Pattern* voor dat aspect gemakkelijk.

9.1.3 Toevoegen transformaties beperken

Er kan bepaald worden welke transformaties mogen toegevoegd worden aan het graafmodel en welke niet. Dit kan met verschillenden doeleinden gebeuren:

- Enkel transformaties toelaten waarvan bepaalde informatie beschikbaar is. Herschikken van het graafmodel om het te optimaliseren is bijvoorbeeld niet met alle transformties mogelijk.
- Het soort componenten dat dient voor grafische weergave kan beperkt worden tot één enkele soort. Dat soort componenten is afhankelijk van de grafische omgeving. Er kunnen bijvoorbeeld enkel Swingcomponenten, of enkel servlets gebruikt worden.
- Enkel met bepaalde soorten gegevens werken, door transformaties die met andere soorten gegevens werken, niet toe te laten in het graafmodel.
- ...

Dit is eigenlijk geen losstaande strategie, maar kan een deel zijn van andere uitbreidingen.

9.1.4 Grafische omgeving

Als software aan de man moet gebracht worden, moet het niet alleen goed werken, maar het moet er ook mooi uitzien. Een toepassing die gemaakt is met behulp van ***Longbow***, kan er op verschillende manieren uitzien.

Er bestaan tal van technologieën die als doel hebben software te visualiseren. Transformaties kunnen gebruikt worden om bepaalde grafische componenten weer te geven. In hetzelfde graafmodel werken de transformaties meestal samen om een gemeenschappelijk doel te bereiken.

Voor het maken van GUI's bestaan er toepassingen waarmee de grafische componenten op een goede plaats kunnen gezet worden, waarna er nog wat moet geprogrammeerd worden, zodat er iets zinnigs gebeurt. Het maken van een GUI met behulp van ***Longbow*** verloopt eigenlijk omgekeerd. Eerst wordt ervoor gezorgd dat het graafmodel goed in elkaar zit. Daarvoor kan gebruik gemaakt worden van grafische componenten, vertegenwoordigd door een transformatie.

Eenmaal de applicatielogica in elkaar zit, kunnen de componenten op een zinnige manier weergegeven worden. Er kan daarvoor een nieuwe modus voorzien worden. Van ontwerpmodus kan er naar die modus overgegaan worden, voordat er naar uitvoeringsmodus wordt overgegaan. De

schikking van visueel weergegeven componenten moet bewaard blijven in een datastructuur, ook na uitvoeren van het graafmodel.

Direct overgaan van ontwerp- naar uitvoeringsmodus moet ook mogelijk zijn, op voorwaarde dat de grafische componenten een plaats gekregen hebben.

Die nieuwe modus kan voor Swingcomponenten met behulp van een `BeanContext` gerealiseerd worden. De grafische componenten behoren dan tot die `BeanContext`.

9.2 Aanbieden van diensten aan transformaties

Componentenframeworks zoals `BeanContext` bij JavaBeans bieden de mogelijkheid dat er diensten worden aangeboden aan de componenten van het framework. Een graafmodel zou op diverse manieren van dergelijke diensten gebruik kunnen maken.

Een verlener van diensten kan, net zoals een transformatie, toegevoegd worden aan het graafmodel. In ***Longbow*** zou het er kunnen uitzien als een transformatie zonder ingangen en uitgangen, of zelfs als een gewone transformatie, die bovendien in staat is diensten aan te bieden.

Aanvragen van een dienst wordt in een `BeanContextServices` gerealiseerd via het doorgeven van een `Class`-object aan een *factory* methode. In de plaats daarvan kan een contract gebruikt worden.

Intern wordt een verlener van diensten dus voorgesteld als een *factory*, die aan de hand van een contract een referentie naar een dienst, een bepaald object, kan teruggeven. Het is mogelijk dat een dienst telkens wordt aangemaakt wanneer het nodig is, of een bepaalde soort dienst kan altijd naar hetzelfde object verwijzen, zodat een object uniek kan zijn voor een gegeven graafmodel.

Transformaties kunnen gebruik maken van diensten. De uitvoerbaarheid van een transformatie kan afhankelijk zijn van het beschikbaar zijn van een dienst in het graafmodel. Net zoals bij poorten, moet het gebruik van diensten facultatief kunnen zijn. Er zouden ook contextgevoelige regels kunnen bestaan, die bepalen van welke diensten een transformatie kan gebruik maken, of het gebruik van die diensten facultatief is, ...

Ingangen, uitgangen en de mogelijkheid om van diensten gebruik te maken, zouden uniform als poorten kunnen voorgesteld worden. Er zou een nieuw soort verbinding kunnen geïntroduceerd worden, die een verlener van een dienst verbindt met een verbruiker ervan. Die verbindingen worden automatisch gelegd en verbroken, zonder dat er een ontwerper aan te pas komt. Ontwerpers kunnen enkel beslissen welke diensten er beschikbaar zijn en welke transformaties er worden toegevoegd, die van die diensten gebruik kunnen maken. Het moet duidelijk zijn dat diensten een invloed zullen hebben op de uitvoerbaarheid van het graafmodel.

Een referentie naar een dienst kan al beschikbaar zijn in ontwerpmodus. Telkens als er iets verandert voor de verlener van een dienst, bijvoorbeeld door het aanpassen van een constante, moeten de referenties naar de diensten opnieuw ingesteld worden. Er moet geprobeerd worden om zowel het wijzigen, toevoegen en verwijderen van diensten efficiënt te maken, als het controleren van de uitvoerbaarheid van het graafmodel.

9.2.1 Configuratie van een graafmodel

Er kunnen diensten gebruikt worden om een graafmodel te configureren. Een dienst zou bijvoorbeeld een *connectionpool* kunnen zijn, die verbindingen met een databank levert. Om dezelfde

functionaliteit te hergebruiken, maar met een andere databank, volstaat het om de verlener van de dienst te vervangen.

Er zouden systeemafhankelijke instellingen, zoals bijvoorbeeld de locatie van bepaalde lokale bestanden, kunnen beschikbaar gesteld worden via een dienst. Een graafmodel zou dan, behalve die dienst, systeemonafhankelijk kunnen zijn.

In combinatie met het *Strategy Pattern* (zie 4.4.2), zouden in een graafmodel systeemafhankelijke functies kunnen uitgevoerd worden.

De dienst zou ook kunnen aangepast worden, door hem voor te stellen als een transformatie met een ingang, waar bijvoorbeeld de URL van een databank kan ingegeven worden. Er kan via de metadata van die ingang geëist worden dat die URL constant moet zijn tijdens het uitvoeren van het graafmodel.

9.2.2 Unieke objecten

Objecten kunnen uniek zijn voor een graafmodel. Een pseudorandomgenerator is een goed voorbeeld van een dergelijk object dat (meestal) uniek moet zijn. Als een graafmodel dient om het gedrag van randomgeneratoren te onderzoeken, kunnen er natuurlijk nog altijd meerdere generatoren als data beschikbaar gesteld worden.

Diensten zijn niet strikt nodig om objecten te verkrijgen die uniek zijn voor een graafmodel, maar ze zijn een handig hulpmiddel om de uniciteit van een object te verzekeren. Ze vermijden dat er moet gesteund worden op het consequent werken van een ontwerper.

9.2.3 Werken met centrale data in een gedistribueerd datamodel

Een dienst kan een dataobject beschikbaar stellen, dat op meerdere plaatsen in het graafmodel kan gebruikt worden. Een eenvoudig voorbeeld waarvoor dit nodig is, is portefeuillebeheer. Als een graafmodel wordt gebruikt om op de beurs te spelen, of om op een weloverwogen manier te gokken in een casino, moet er een saldo kunnen worden bijgehouden. Er moet bijvoorbeeld kunnen gecontroleerd worden wanneer al het geld op is. Zonder gebruik te maken van diensten is zoiets onmogelijk met een gedistribueerd datamodel. Zodoende kan er een grote schuldenberg ontstaan.

Werken met een mengeling van een centraal en een gedistribueerd datamodel wordt mogelijk door gebruik te maken van diensten. Centrale data worden hierbij als dienst beschikbaar gesteld. Merk op dat een dienst een mark moet kunnen sturen naar een transformatie die ervan gebruik maakt.

9.3 Versiebeheer

De enige constante in softwareontwikkeling is de voortdurende verandering. Om het leven voor veel mensen makkelijker te maken, wordt er niet alleen maar nietwe software uitgebracht, maar ook nieuwe versies van bestaande software. Wat ***Longbow*** betreft, komt dit neer op nieuwe versies van transformaties en nieuwe versies van de structuur van het graafmodel.

9.3.1 Versiebeheer van transformaties

Alle transformaties moeten achterwaarts compatibel zijn met hun vorige versies:

1. De ingangen moeten minstens dezelfde data kunnen aanvaarden. Hun metadata mogen dus uitgebreid worden met nieuwe regelverzamelingen, maar niet met nieuwe *constraints* in bestaande regelverzamelingen.
2. Onder bepaalde voorwaarden kunnen er ingangen of uitgangen toegevoegd worden. Er moet minstens één configuratie zijn van de toegevoegde ingangen, zodanig dat de nieuwere versie, zonder rekening te houden met de nieuwe ingangen en uitgangen, functioneel identiek is aan de oude. Dit betekent dat er een blackboxtest moet kunnen gemaakt worden, zodanig dat niet kan achterhaald worden of in de testopstelling gebruik werd gemaakt van de nieuwe dan wel van de oude versie.
3. Een uizondering op regel 2 is als er fouten verholpen zijn in de nieuwe versie.
4. Er mogen ingangen toegevoegd worden in een nieuwe versie als daarbij voldaan is aan regel 2
5. Er mogen uitgangen toegevoegd worden.
6. De uitgangen mogen elk hoogstens dezelfde data kunnen afgeven als in de vorige versie

Als transformaties op een goede manier formeel beschreven zijn, kunnen al deze eisen min of meer automatisch getest worden. Er kan minstens automatisch een checklist gegenereerd worden om te controleren of de nieuwe versie achterwaarts compatibel is met de oudere.

Het updaten van een graafmodel, door transformaties te vervangen door nieuwere versies, kan dus minstens gedeeltelijk automatisch gebeuren.

Er kan ook checklist gegenereerd worden voor het aanpassen van de documentatie na het vergelijken van twee versies van een transformatie.

9.3.2 Versiebeheer van een graafmodel

Een graafmodel kan gewijzigd worden door transformaties eraan toe te voegen, te verwijderen of te vervangen door transformaties met dezelfde interface. Verbindingen tussen transformaties kunnen toegevoegd, verwijderd of gewijzigd worden. Als meerdere ontwerpers samenwerken om hetzelfde graafmodel op te bouwen, moeten ze van elkaars wijzigingen op de hoogte gebracht worden. Ook voor een enkele ontwerper is het handig om terug te kunnen kijken naar welke wijzigingen hij heeft gemaakt en waarom. Wijzigingen moeten indien nodig kunnen ongedaan gemaakt worden.

Versiebeheer kan op verschillende manieren geïmplementeerd worden. Er zijn enkele ontwerpbeslissingen die moeten genomen worden om te komen tot een goed versiebeheersysteem voor een graafmodel:

- Het algoritme om de verschillen tussen opeenvolgende versies te bepalen kan verschillend zijn. Er kan gebruik gemaakt worden van een diff-algoritme, speciaal ontworpen en geschikt voor een graafmodel, of er kan gebruik gemaakt worden van een *Command Pattern*.
- Het tijdstip waarop er sprake is van een nieuwe versie moet bepaald worden. Ofwel beslist de ontwerper of er sprake is van een nieuwe versie, ofwel is er altijd sprake van een nieuwe versie wanneer het graafmodel uitvoerbaar is. Beide benaderingen hebben hun voor- en nadelen en kunnen gecombineerd worden.
- Er moet beslist worden hoe het beheer gebeurt: centraal beheer, delen van het graafmodel door verschillende ontwerpers laten beheren, concurrente versies, ...
- Het is mogelijk alleen maar rekening te houden met de structuur van de graaf. Er kan bovendien ook rekening gehouden worden met de data in de constanten. Er moet ook gekozen worden of er met diensten rekening moet gehouden worden.
  Eventueel kan de ontwerper kiezen welke factoren in rekening worden gebracht.

- Hoe wordt het versiebeheersysteem beveiligd?
- ...

Verschillende vormen van versiebeheer kunnen hun nut hebben. Een configureerbaar versiebeheersysteem voor graafmodellen, geïmplementeerd als een metamodel zou al deze vormen in één graafmodel kunnen gieten.

Er zou voor kunnen gezorgd worden dat er altijd met de meest recente versie van transformaties gewerkt wordt, als er periodiek naar nieuwe versies van transformaties gezocht wordt. Er kunnen ook andere criteria dan de meest recente gebruikt worden, bv. de veiligste, de snelste, die met de hoogste kwaliteit, ...

Diff-algoritme

Er kan een diff-algoritme ontwikkeld worden die alle wijzigingen tussen twee graafmodellen opspoort en die compact kan opslaan. Het opslaan van een versie betekent het opslaan van de verschillen tussen de huidige en de vorige versie. Aan de hand van de verschillen tussen twee versies moet, gegeven de oudste versie, de nieuwste versie kunnen opgebouwd worden. In feite betekent dit dat een diff als een reeks commando's moet kunnen gezien worden, die op een oudere versie van een graafmodel kunnen uitgevoerd worden en waarvan het resultaat een nieuwere versie is. Als een diff-algoritme gebruikt wordt om het verschil tussen twee versies te bepalen, is het dus nog altijd nuttig om het *Command Pattern* te implementeren.

Command Pattern

Telkens een ontwerper iets aan het graafmodel wil wijzigen, wordt er een *commando* aangemaakt. Een commando kan uitgevoerd worden, ongedaan gemaakt en als het ongedaan gemaakt is kan het heruitgevoerd worden. Er wordt een gelinkte lijst van commando's bijgehouden worden, zodat een ontwerper op elk moment op zijn stappen kan terugkeren. Een andere mogelijkheid is dat een ontwerper niet op elke stap kan terugkeren, maar slechts enkele stappen, wat meestal genoeg is. Dankzij het versiebeheersysteem kan de ontwerper ook terugkeren naar alle vorige werkende of opgeslagen versies.

Aangezien het aantal mogelijke uit te voeren acties op een graafmodel door een ontwerper beperkt is (ten opzichte van het wijzigen van een broncodebestand), is het veel gemakkelijker om een reeks commando's compact te maken, i.e. zoveel mogelijk commando's te verwijderen om toch nog hetzelfde resultaat te bekomen. Dat is eenvoudig als de commando's per soort gegroepeerd worden:

- verwijderen transformaties
- toevoegen transformaties
- verwijderen verbindingen
- toevoegen verbindingen
- ...

Er moet wel op gelet worden dat de laatste commando's in dezelfde volgorde behouden worden, zodat de ontwerper nog op zijn stappen kan terugkeren.

Een opgeslagen lijst van commando's, waarmee het graafmodel kan opgebouwd worden, stelt een eenvoudige persistente vorm van het graafmodel voor.

Op het ogenblik dat beslist wordt om een nieuwe versie op te slaan, wordt de volledige lijst van commando's, inclusief de laatste, compact gemaakt. Er kan verder onderzocht worden of er een efficiënt *randomized* algoritme bestaat om de lijst van commando's compacter te maken.

Het komt er eigenlijk op neer dat een ontwerper, terwijl hij bezig is het graafmodel te wijzigen, tegelijk bezig is met een diff te maken. Het verschil met de vorige paragraaf is dat het nu niet nodig is om twee grafen te doorzoeken om een diff te maken. Werken met het *Command Pattern* is dus waarschijnlijk efficiënter als versies van het graafmodel moeten beheerd worden.

9.4 Automatische documentatie

Documentatie van transformaties in een bepaalde context kan automatisch gegenereerd worden. Daarvoor is het nodig dat er een algemene, contextonafhankelijke documentatie bestaat voor de transformatie, die rekening houdt met alle mogelijke contexten. Als die algemene documentatie op een gepaste manier geannoteerd wordt, bijvoorbeeld met behulp van RDF, kan een *software-agent* daaruit afleiden wat de documentatie moet zijn *in de huidige context*.

Een *software-agent* kan helpen voor het opstellen van de algemene documentatie, aan de hand van de beschrijving van de contextgevoelige regels van de transformatie. De *agent* zou bijvoorbeeld kunnen ontdekken dat de documentatie niet volledig is.

Ook de waarde van data in constanten die met de transformatie verbonden zijn, zou kunnen gebruikt worden voor het genereren van contextafhankelijke documentatie.

Documentatie voor de samenstelling van componenten kan samengesteld worden uit de documentatie van de afzonderlijke componenten. Deze documentatie zal waarschijnlijk rudimentair zijn. Het is voor software mogelijk te achterhalen wat de bedoeling is van een bepaalde samenstelling van transformaties. Die bedoeling klaar en duidelijk in menselijke taal formuleren is echter een ander paar mouwen.

Het *Semantic Web* biedt de mogelijkheid tot het aanmaken van documentatie, enkel door het samenstellen van componenten. Als machines kunnen 'begrijpen' *hoe* componenten samengesteld worden, wat de *betekenis* is van een bepaalde koppeling van een ingang met een uitgang, dan kunnen ze ook documentatie genereren die ondubbelzinnig beschrijft wat een bepaalde samenstelling van componenten *doet*. Hiervoor is het uiteraard absoluut noodzakelijk dat componenten, evenals hun ingangen en uitgangen geannoteerd worden met behulp van een bepaalde ontologie.

Het idee van automatische documentatie stamt al uit 1968 en is te lezen in een reactie op de lezing van McIlroy.[1]

9.5 Sessiebeheer

Om sessiebeheer toe te voegen zijn enkele aanpassingen nodig:

- Data moeten bij een bepaalde sessie kunnen behoren. Dat kan opgelost worden door van een dataobject een hashtabel te maken, met als sleutel de sessie en als waarde de data zelf. Er zijn twee mogelijke visies hieromtrent. Het nut van de tweede bestaat zeker, maar is moeilijk te begrijpen:
  1. Eigenlijk moet er hiervoor niets veranderen aan het formeel model, want er kan worden afgedwongen dat een dataobject een dergelijke hashtabel moet bevatten. Het nadeel van deze methode is dat het mogelijk is het werken met sessies te omzeilen door de metadata uit te breiden.

1. [Naur and Randell (Eds.), 1969]

2. Er moet een dataobject gemaakt worden die met sessies kan omgaan. Dat dataobject bestaat op zich uit een hashtabel en kan data bevatten voor verschillende sessies. Als er in een graafmodel niet met sessies gewerkt wordt, kan er bijvoorbeeld een ingang met een standaardsleutel gemaakt worden.

- Marks en sweeps moeten duidelijk maken voor welke sessie ze bedoeld zijn. Hiervoor moet er weinig veranderen. Het werd al vermeld dat het een goed idee zou zijn om marks en sweeps zo weinig mogelijk data tegelijk ongeldig te laten maken.
- Niet alle transformaties zullen met sessies kunnen omgaan. Dat moet blijken uit de interface van de transformatie.

Het uitvoeren van een gedeelte van een graafmodel kan afhankelijk zijn van de huidige sessie, een ander deel kan onafhankelijk zijn voor sessies en dus voor alle sessies gelden. Op die manier kan er gecommuniceerd worden tussen sessies (en kan er dus bijvoorbeeld een chatruimte op een webpagina gemaakt worden). Een ontwerper moet greep kunnen hebben over concepten zoals:

- de huidige sessie
- alle huidige sessies van het graafmodel
- alle sessies in een bepaald gebied van de graaf
- alle huidige sessies in een bepaald gebied
- alle sessies
- ...

Al deze concepten kunnen op zich als een sessie bekeken worden. Een sessie kan afhankelijk zijn van andere sessies. Als een sessie op een bepaalde plaats geen specifieke data heeft, kan er gebruik gemaakt worden van data van een sessie waarvan deze afhankelijk is. Alle sessies zijn bijvoorbeeld afhankelijk van de sessie *'alle sessies'*, behalve die sessie zelf.

De sessies vormen – voor de verandering – een *dependency graph* en kunnen dus opgesplitst worden in lagen.

9.5.1 Persistentie van structuur en data

Het invoeren van sessies geeft een nieuwe dimensie aan het persisteren van data. Een sessie moet onafhankelijk van andere sessies kunnen opgeslagen worden. Bij het opslaan van een sessie moeten enkel de wijzigingen die invloed hebben op die welbepaalde sessie en niet op sessies waarvan het afhankelijk is, opgeslagen worden.

Persistentie van het graafmodel kan nu in verschillende niveaus gebeuren: de structuur van het model is het laagste niveau. Daarbovenop komen dataniveaus. Het laagste dataniveau bevat sessies die afhankelijk zijn van de structuur van het graafmodel, maar niet van andere sessies. Elk hoger niveau bevat gegevens van sessies die afhankelijk zijn van sessies waarvan de data in een lager niveau worden opgeslagen.

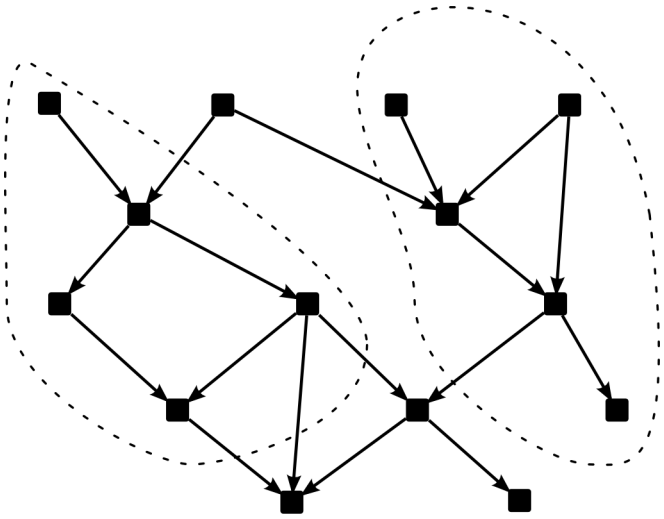
9.5.2 Concurrente verwerking

Er kan onderzocht worden of de uitvoering van verschillende sessies van het graafmodel kan gesplitst worden over verschillende threads of processoren. Dezelfde vraag kan gesteld worden over het spreiden van de verwerking over verschillende computers. In die zin kunnen sessies ook bekeken worden als iets dat eerder samenhangt met een bepaalde computer of thread, dan met een bepaalde gebruiker.

9.6 Parallelle verwerking

Er zijn verschillende mogelijkheden om het uitvoeren van het graafmodel door meerdere processoren, of kernen van processoren te laten gebeuren. In Java kan dat gebeuren door de verwerking te spreiden over meerdere threads.

9.6.1 Langs elkaar niet kruisende paden



Figuur 9.1: Twee onafhankelijke gebieden in een *dependency graph*

In figuur 9.1 worden twee gebieden op een *dependency graph* aangeduid. Als de vierkantjes transformaties voorstellen, zou dit betekenen dat de aangeduide gebieden zonder problemen in verschillende threads kunnen uitgevoerd worden. Een goede verdeling van de transformaties in zulke gebieden kan het uitvoeren van het graafmodel aanzienlijk versnellen. Op computersystemen waar dat mogelijk is, kunnen meerdere processoren, of meerdere kernen van processoren aangewend worden om het graafmodel uit te voeren.

Statische verdeling in gebieden

Er kan, voordat het graafmodel wordt uitgevoerd, een verdeling gemaakt worden in gebieden, die in aparte threads worden uitgevoerd. Tijdens de verwerking kunnen die gebieden niet wijzigen.

Ongeveer hetzelfde zou kunnen verwezenlijkt worden door de verschillende gebieden in te kapselen en daarna het graafmodel uit te voeren, waarbij elke transformatie in zijn eigen thread wordt uitgevoerd.

Dynamische verdeling in gebieden

Er zijn verschillende manieren om een *dependency graph* te verdelen in onafhankelijke gebieden. Het is mogelijk dat een bepaalde thread altijd op dezelfde andere thread moet wachten, omdat het uitvoeren van de transformaties in zijn gebied langer duurt.

Als er statistieken over de performantie worden bijgehouden, is het mogelijk om de verdeling in onafhankelijke gebieden dynamisch aan te passen, zodanig dat de uitvoering van het graafmodel versneld wordt.

9.6.2 Pipelining

In plaats van de *dependency graph* te verdelen in gebieden, kan ook dynamisch gecontroleerd worden of transformaties van elkaar afhankelijk zijn. Het basisidee is als volgt.

Er kan op elk moment een wachtrij bestaan van transformaties die moeten uitgevoerd worden. Die wachtrij kan een *priority queue* zijn, met de partiële orde van het graafmodel als prioriteit. Transformaties hebben dus een hogere prioriteit dan hun opvolgers.

Verder is er ook een *threadpool*, die threads beheert waarin transformaties kunnen uitgevoerd worden.

Het beheren van de wachtrij gebeurt ook in een thread. Dat kan een thread zijn van de *threadpool*. Als er een thread vrij is om gebruikt te worden, wordt de thread met de wachtrij geactiveerd. De top van de wachtrij wordt eraf gehaald. Als dat een transformatie is die onmiddellijk kan uitgevoerd worden, dan gebeurt dat. Als de transformatie niet kan uitgevoerd worden doordat een van zijn voorlopers nog wordt uitgevoerd, of doordat een voorloper ongeldig is, dan wordt de transformatie terug op de wachtrij gezet. Dat gebeurt tot er een transformatie van de wachtrij wordt gehaald die onmiddellijk uitvoerbaar is.

Het mark-and-sweepmechanisme moet hiervoor nauwelijks aangepast worden. In plaats van een transformatie onmiddellijk uit te voeren, moet die nu toegevoegd worden aan de wachtrij.

9.6.3 Grid computing

Wat pipelining is voor een lokale machine, is *grid computing* voor een netwerk van computers. In plaats van transformaties uit te voeren in verschillende threads van een lokale machine, zouden ze ook kunnen uitgevoerd worden op verschillende threads van verschillende computers. Voor het verdelen van de transformaties over de computers kan er rekening gehouden worden met de rekenkracht van die computers en de kost van het uitvoeren van een transformatie. Wat de betreft de hoeveelheid data die moet overgedragen worden, kan er ook rekening gehouden worden met de capaciteit van de netwerkverbindingen.

Opdat dit in combinatie met diensten mogelijk zou zijn, zouden diensten via het netwerk beschikbaar moeten zijn.

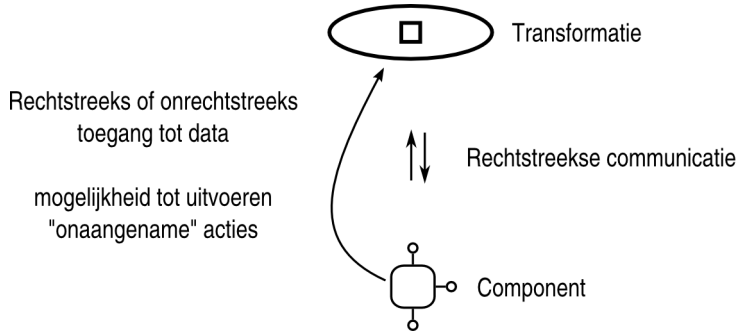
9.7 Beveiliging

9.7.1 Autorisatie en authenticatie

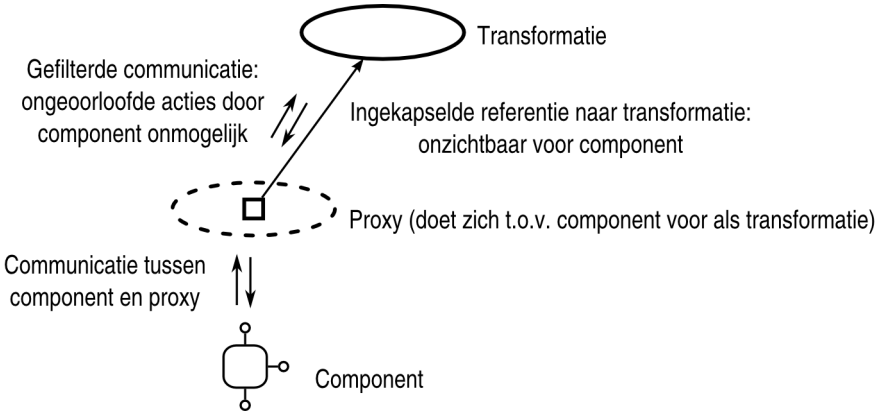
Longbow kan gekraakt worden. Stel dat een component gemonitord wordt door een transformatie. De transformatie moet op de hoogte gehouden worden van wijzigingen in de interactieve component, dus moet er rechtstreeks of onrechtstreeks een referentie zijn van de interactieve component naar zijn monitor.

Die component kan dus volledige toegang krijgen tot de transformatie, en langs die weg de normale werking van het graafmodel verstoren, of er op een ongeoorloofde manier informatie uithalen.

Een manier om transformaties te beveiligen is gebruik te maken van *proxies*: een component communiceert niet rechtstreeks met zijn monitor, maar doet dat via een tussenliggend object, een proxy. De proxy kan instaan voor authenticatie en autorisatie. De proxy kan er ook voor zorgen dat de interactieve component nooit rechtstreeks toegang krijgt tot het graafmodel en zijn componenten: de referentie naar de eigenlijke transformatie wordt ingekapseld.



Figuur 9.2: Mogelijk veiligheidsprobleem door rechtstreekse toegang tot transformaties



Figuur 9.3: Communicatie tussen een transformatie en een component via een *proxy*

Beveiliging van transformaties kan geautomatiseerd worden. Er kan voor de beveiliging een strategie toegepast worden, die ervoor zorgt dat alle transformaties vertegenwoordigd worden door een proxy die de toegang tot het graafmodel regelt.

Data zelf kunnen beveiligd worden door te encrypteren. Voor het opstellen van een beveiligingsbeleid moet exact nagegaan worden waar encryptie nodig is. Automatiseren van beveiliging is wenselijk, omdat het menselijke fouten helpt vermijden.

9.8 Mogelijkheden van *Longbow*

De vraag wat er met ***Longbow*** op het vlak van software kan gedaan worden kan in twee woorden beantwoord worden: *bijna alles*. Dat kan gaan van *Hello, World!* over het uitvoeren van complexe simulaties, tot het gebruik ervan als systeem om de software in een bedrijf aan elkaar te koppelen. Het is belangrijk te begrijpen dat de eigenlijke functionaliteit niet in ***Longbow*** zelf vervat zit, maar in de transformaties, of in de componenten waarvoor die een vertegenwoordiger zijn.

Longbow kan de lijm vormen tussen componenten die zelf eenvoudige of complexe acties uitvoeren. Reeds bestaande softwarecomponenten kunnen, mits het beschikbaar stellen van de gepaste transformaties, samen acties uitvoeren.

9.8.1 Beperkingen

Realtimetoepassingen zijn niet zonder meer mogelijk. Als er garanties moeten kunnen gegeven worden op het vlak van hoeveel tijd er kan gespendeerd worden aan het uitvoeren van een transformatie, of hoeveel tijd er mag zijn tussen het ontvangen van een sweep en het geldig worden van een transformatie, moeten er speciale voorzieningen getroffen worden. Wat er allemaal moet veranderen om ze mogelijk te maken, kan het onderwerp zijn van verder onderzoek.

Een graafmodel kan alleen uitgevoerd worden in een omgeving die voldoet aan bepaalde systeemeisen. Embedded systemen beschikken bijvoorbeeld over het algemeen niet over genoeg processorkracht en geheugen om een framework op te laten draaien. Ze kunnen echter wel communiceren met een interactieve transformatie, zodat ze met behulp van een graafmodel beheerd kunnen worden.

9.8.2 Uitvoeren van programma's

Als iemand ooit een programma zou schrijven dat de wereldvrede bevordert, dan zou ***Longbow*** de wereldvrede kunnen bevorderen. Er kunnen immers transformaties gemaakt worden die programma's kunnen uitvoeren. Programma's kunnen op gezette tijdstippen uitgevoerd worden, zodat de mogelijkheden van bijvoorbeeld *cron* in een graafmodel kunnen gevat worden. Er kan gewacht worden tot het programma beëindigd wordt, of het kan asynchroon gestart worden.

9.8.3 Gebruik maken van bibliotheken

Als ***Longbow*** in Java geïmplementeerd wordt, kunnen er bibliotheken gebruikt worden, die in diverse programmeertalen geschreven zijn. Via *native* methoden kunnen er functies en procedures uit bibliotheken in machinetaal gebruikt worden. Bibliotheken kunnen ook beschikbaar gesteld worden via *webservices* of RMI. Er zijn ook nog andere mogelijkheden.

9.8.4 Geïnterpreteerde talen

Er bestaan diverse *scripting engines* die in Java geïmplementeerd zijn. Die kunnen talen zoals bijvoorbeeld *JRuby*, *Jython*, *Jawk* en *JavaScript* interpreteren. Dat betekent dat deze scripttalen kunnen gebruikt worden om transformaties te implementeren. De methoden die beschikbaar gesteld worden voor het implementeren van transformaties, kunnen immers gebruikt worden in scripttalen. Dat is zelfs vrij eenvoudig, met behulp van de Scripting API in Java 1.6. Het is dus mogelijk om heel snel eenvoudige transformaties te ontwikkelen met behulp van een scripttaal.

Scripttalen kunnen ook gebruikt worden om een graafmodel samen te stellen in ontwerptijd. Hiervoor volstaat het dat een *scripting engine* controle krijgt over een graafmodel. In combinatie met de vorige toepassing van scripttalen, kan een graafmodel dus gebruikt worden voor het uitvoeren van een ingewikkelde collectie van scripts in diverse talen.

9.8.5 Diverse soorten componenten

Componenten van allerlei soorten kunnen met behulp van ***Longbow*** samengesteld worden. Het begrip *component* wordt heel ruim opgevat, zodat zelfs *embedded systems* die op afstand kunnen beheerd worden als een component kunnen gezien worden. Ook drivers van bepaalde hardware kunnen soms als een component gezien worden.

De vraag is eigenlijk niet: *"Waaraan moet een component voldoen om door een transformatie vertegenwoordigd te kunnen worden?"* maar het antwoord is: *"Alles wat door een transformatie kan vertegenwoordigd worden, inclusief een transformatie zelf, is een component."*

9.9 Metamodellen

Een metamodel is een graafmodel dat minstens één graafmodel bevat. Veel toepassingen kunnen daarvan gebruik maken. Er zijn een aantal verschillende manieren waarop metamodellen kunnen voorkomen.

Het kunnen voorkomen van metamodellen is geen uitbreiding van het formeel model. Ze bestaan doordat graafmodellen interactieve componenten zijn.

9.9.1 Graafmodel in transformatie

Een graafmodel kan beheerd worden door een interactieve transformatie, die geen verlener is van een dienst. Die transformatie kan gebruikt worden als ontwerper en eindgebruiker ervan.

Transformatie beheert ander graafmodel dan metamodel

In dit geval is een interactieve transformatie de vertegenwoordiger van een ingebed graafmodel, dat niet het graafmodel is waar het zelf toe behoort. Het ingebed graafmodel kan zich op afstand bevinden en de communicatie tussen de transformatie en dat graafmodel kan bijvoorbeeld via *webservices* verlopen.

Als het ingebed model zich in ontwerpmodus bevindt, kan het aangepast worden aan de hand van informatie die de transformatie in zijn omringende dataknopen en in het ingebedde graafmodel vindt. Er kan dus een automatische ontwerper gemaakt worden door een metamodel te ontwerpen.

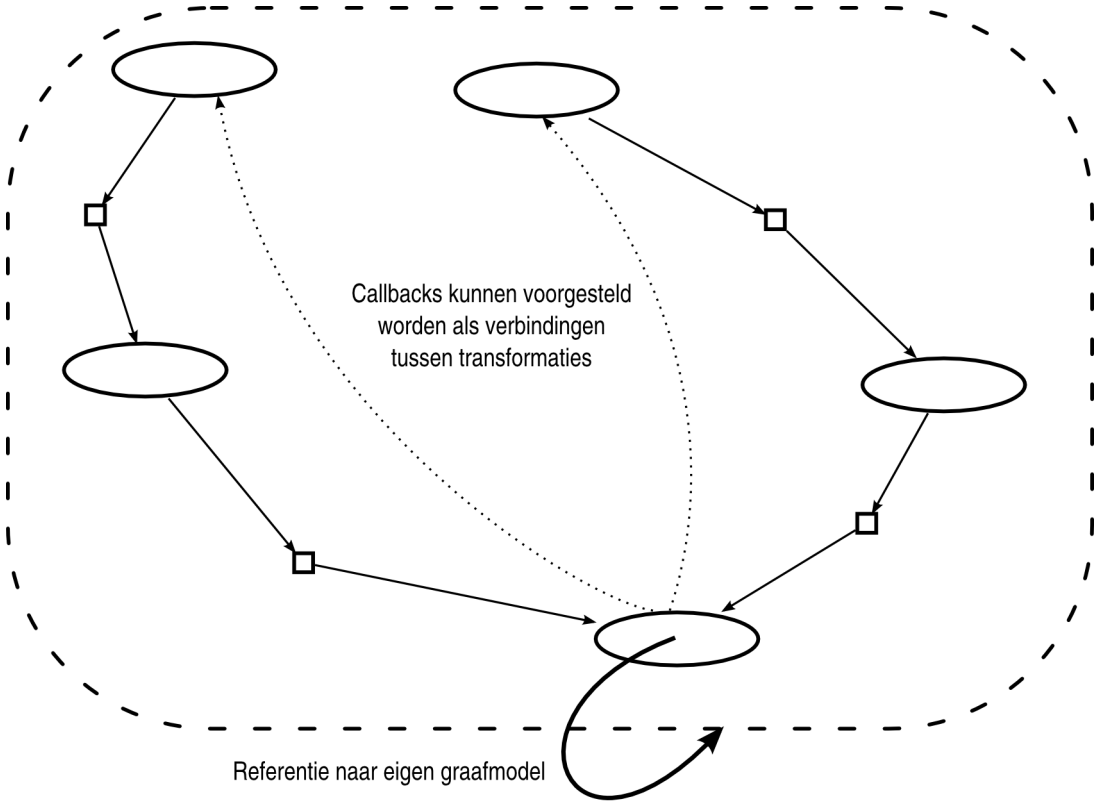
Er kan ook een metamodel gemaakt worden dat enkel controleert wanneer een ander graafmodel moet uitgevoerd worden.

Transformatie bevat metamodel

Een transformatie kan een vertegenwoordiger zijn van het graafmodel waar hij zelf toe behoort. Zo'n graafmodel kan zichzelf niet ontwerpen, want het kan zich niet tegelijk in ontwerptijd bevinden als ingebed graafmodel en in uitvoeringstijd als metamodel.

Zichzelf stoppen. Een metamodel dat zichzelf bevat, kan zichzelf stoppen wanneer het wordt uitgevoerd. Dat kan bijvoorbeeld gebruikt worden als veiligheidsmechanisme voor een graafmodel dat bepaalde hardware beheert. Het is ook een manier om bijvoorbeeld een graafmodel dat een interactieve component bevat na een bepaalde tijd te stoppen.

Callbacks. Een graafmodel dat zichzelf bevat, kan marks en sweeps sturen naar gelijk welke transformatie van het graafmodel. Het sturen van een mark naar een voorloper van de interactieve transformatie kan een *callback* genoemd worden. Er ontstaat op die manier een lus. Die lus kan oneindig zijn, bijvoorbeeld als er periodiek telkens hetzelfde gebeurt. Er kan ook een stopvoorwaarde ingebouwd worden, door enkel marks te sturen naar andere transformaties als er aan een bepaalde voorwaarde voldaan is.



Figuur 9.4: Implementatie van *callbacks* via een metamodel

Dergelijke *callbacks* kunnen controleerbaar zijn door een ontwerper. Er kan een nieuw soort verbinding ingevoerd worden, die bepaalt welke transformaties ongeldig worden ten gevolge van het ongeldig worden van andere transformaties. Dergelijke verbindingen zijn weergegeven in figuur 9.4.

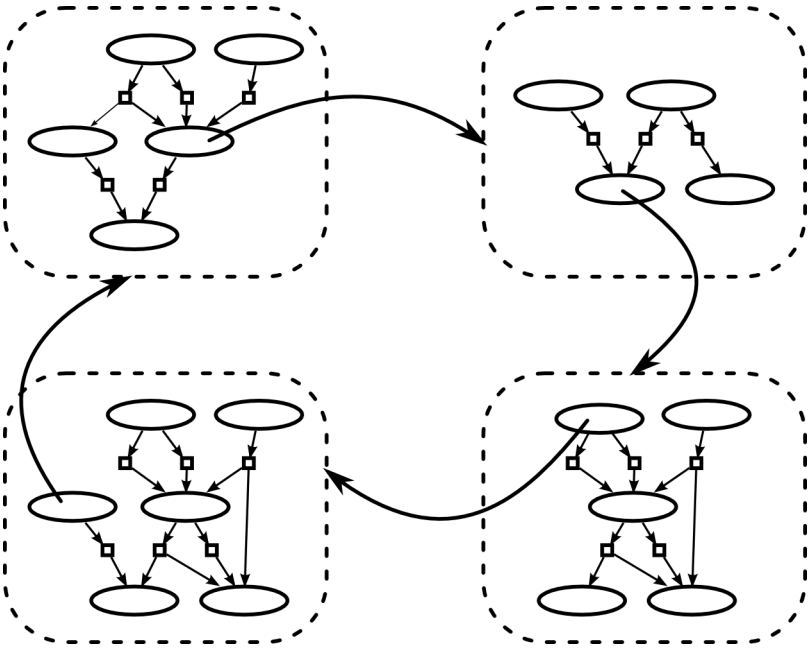
Het is opmerkelijk dat *callbacks* de implementatie van parallelle verwerking (zie 9.6) niet in de weg staan. Dat is te verklaren doordat ze geen onmiddellijke uitvoering of verplaatsing van data tot gevolg hebben, maar enkel transformaties als ongeldig markeren. Er moeten geen onoverkomelijke synchronisatieproblemen opgelost worden.

Stel dat er een heleboel statistieken moeten berekend worden over mensen van verschillende nationaliteiten, verschillende leeftijdsgroepen en voor zowel mannen als vrouwen. Het kan zijn dat alle statistieken moeten bepaald worden voor elke mogelijke combinatie van nationaliteiten, ge-

slachten en leeftijdsgroepen. Dat is perfect realiseerbaar met behulp van *callbacks*. Backtracking kan op bijna dezelfde manier gerealiseerd worden.

De combinatie van *callbacks* en diensten bieden de mogelijkheid methodes zoals *tabu search*, *simulated annealing* en *genetische algoritmen* toe te passen. Toepassen van zulke methodes in combinatie met *grid computing* biedt interessante mogelijkheden.

Keten van graafmodellen



Figuur 9.5: Een keten van graafmodellen

Er kan een keten van graafmodellen gemaakt worden, zoals in figuur 9.5. Dat kan nuttig zijn, bijvoorbeeld als voor het controleren van de stopvoorwaarde van een *callback* op afstand ingewikkelde controles moeten uitgevoerd worden.

9.9.2 Graafmodel als dienst

Een graafmodel kan als dienst beschikbaar gesteld worden, waardoor alle transformaties van het metamodel er toegang toe hebben. Ongeveer dezelfde toepassingen als in de vorige paragraaf zijn mogelijk.

9.10 Diverse toepassingen

9.10.1 Testframework

Er kan een framework gemaakt worden om transformaties te testen, naar analogie met JUnit. De geteste transformatie moet werken in alle omstandigheden, bijvoorbeeld door alle mogelijke combinaties van strategieën voor het graafmodel te testen.

Het testen zelf kan in een ingebed graafmodel gebeuren. Zo kunnen de tests achter elkaar uitgevoerd worden. Het metamodel moet kunnen beschikken over de beschrijvingen van de testopstellingen. Na het uitvoeren van een test, kan het testresultaat weergegeven worden.

Testcases kunnen vrij ingewikkeld zijn, wanneer rekening moet worden gehouden met parallelle verwerking, gedistribueerde verwerking, ...

Transformaties waarvan niet alle tests slagen, zouden niet mogen gebruikt worden in een graafmodel. Het toevoegen van een component kan daarom, in kwaliteitsvolle toepassingen, gepaard gaan met het testen ervan. De actie die een component toevoegt aan een graafmodel kan zelf een metamodel van dat graafmodel zijn, die eerst de component test voordat hij ze toevoegt en de component weigert als er een test faalt.

9.10.2 Model View Controller

MVC kan gerealiseerd worden met behulp van ***Longbow***, als er gebruik gemaakt wordt van diensten en *callbacks*. Er zijn meerdere modellen en views mogelijk in hetzelfde graafmodel. Modellen kunnen als dienst beschikbaar gesteld worden. In tegenstelling tot huidige toepassingen voor het grafisch samenstellen van GUI's, moet er achteraf geen code meer geschreven worden voor het afhandelen van *events*.

Door gebruik te maken van de combinatie van ***Longbow*** en MVC kunnen door ontwerpers GUI's gemaakt worden zonder ook maar één lijn code te moeten typen.

9.10.3 Onderlinge controle

Computersystemen zijn niet altijd even betrouwbaar. Voor kritieke toepassingen, bijvoorbeeld in computersystemen in operatiekwartieren, gebeurt het dat meerdere computers elkaar controleren. Met ***Longbow*** is het opbouwen van een dergelijk controlesysteem ook mogelijk. De implementatie wordt weergegeven in figuur 9.6.

De communicatie tussen de verschillende systemen kan gebeuren volgens een client-servermodel. De transformatie die het resultaat bekendmaakt is een server en de transformaties die de resultaten van de andere systemen controleren zijn clients.

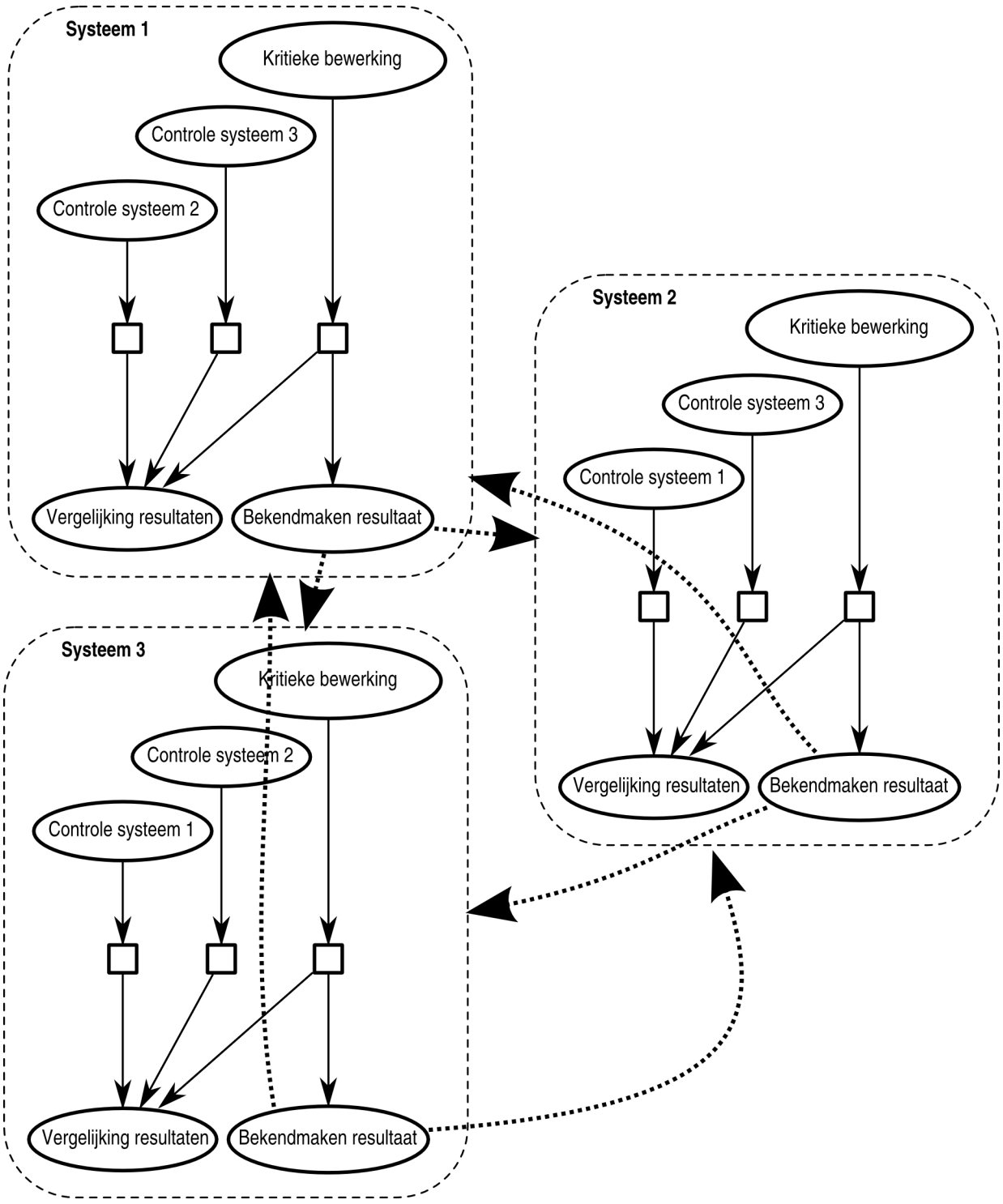
9.11 Bibliotheek

Een ontwerper kan alleen gebruik maken van ***Longbow*** als hij beschikt over de nodige transformaties. De bibliotheek van ***Longbow*** is een component die dient om op een efficiënte manier transformaties terug te vinden. De bibliotheek moet ook in staat zijn transformaties te genereren voor componenten die op een gepaste manier geannoteerd zijn (met behulp van RDF).

Een *agent* (zie 7.3.2) die in staat is een graafmodel samen te stellen, moet, net als een menselijke ontwerper, in staat zijn de bibliotheek te gebruiken.

De gebruiksvriendelijkheid van ***Longbow*** ten opzichte van zowel programmeurs als ontwerpers hangt af van de manier waarop de bibliotheek georganiseerd is. De bibliotheek moet hierbij gezien worden als een medium met twee kanten. Aan de ene kant zijn er de programmeurs, die hun transformaties op een efficiënte manier willen beschikbaar stellen. Aan de andere kant staan de ontwerpers, die graafjes ineen steken en daarvoor transformaties willen gebruiken die ze vlot terugvinden.

Er kunnen op het web *magazijnen* (*repositories*) van transformaties gemaakt worden die kunnen gedownload worden. Een magazijn kan beschikbaar gesteld worden door een uitgever. Eventueel kan een uitgever certificaten uitdelen, die bevestigen dat de transformaties uit het magazijn van hem afkomstig zijn.



Figuur 9.6: Systeem voor onderlinge controle

Een ontwerper moet zijn bibliotheek in bepaalde magazijnen kunnen laten zoeken. Eventueel kan er ook *in het wild* gezocht worden naar transformaties, met behulp van zoekrobots.

Een magazijn moet worden ingedeeld zodat een bibliotheek makkelijk de transformatie die hij zoekt terugvindt. Dit moet de manier waarop ontwerpers naar transformaties zoeken weerspiegelen.

Stel dat er een hiërarchische structuur gebruikt wordt voor het indelen van de transformaties, zoals dat bijvoorbeeld ook gebeurt voor de beschikbare functies in spreadsheetprogramma's. Als een ontwerper zijn weg kent in een boomstructuur van transformaties, kan hij snel de transformatie terugvinden die hij nodig heeft. Dikwijls zal een ontwerper echter, voordat hij begint te zoeken, niet weten welke transformatie hij nodig heeft. Aangezien er na verloop van tijd ook veel meer transformaties zullen bestaan dan functies in een spreadsheetprogramma, kan een boomstructuur voor het organiseren van een magazijn niet efficiënt zijn.

Een betere structuur voor magazijnen kan ontwikkeld worden door aan transformaties en componenten bepaalde eigenschappen toe te kennen. Een beschrijving van een transformatie bestaat uit regels, die met behulp van RDF genoteerd kunnen worden.

Misschien is het mogelijk om voor magazijnen het centrale datamodel gebaseerd op verzamelingen te gebruiken, dat werd besproken in 6.4.1. Doordat regels en verzamelingen in dat model equivalent zijn, zouden transformaties vrij snel, aan de hand van gegeven regels, kunnen teruggevonden worden.

De bibliotheek kan aan de ontwerper de transformaties aanbieden die *relevant* zijn. De relevantie kan bepaald worden aan de hand van de context waarin de ontwerper aan het werken is. Interpreteren van de context is relatief eenvoudig, door het beperkt aantal vrijheidsgraden van het ontwerp. De context kan aangevuld worden door zoektermen die door de ontwerper zijn geselecteerd.

9.12 Eisen voor componenten

9.12.1 Kwaliteit

Uiteraard zijn componenten van hoge kwaliteit het meest waardevol. Maar kwantiteit kan ook van grote waarde zijn, zelfs al is de kwaliteit van sommige van de grote hoeveelheid componenten iets minder. Soms is het belangrijk dat er snel resultaten zijn, vooral voor toepassingen waarbij componenten samengesteld worden om een *fast prototype* te maken. Bij het maken van een prototype kan het dat er componenten nodig zijn die nog niet bestaan. Die moeten dan ter plekke aangemaakt worden en liefst zo snel mogelijk, wat het risico van mindere kwaliteit inhoudt.

Over het algemeen kan worden aangenomen dat:

- Hoe kleiner een component is, hoe groter de kans is dat de kwaliteit goed is
- Hoe langer de component in gebruik is, hoe groter de kans dat de kwaliteit goed is, op voorwaarde dat er af en toe verbeteringen worden aangebracht
- Hoe meer unit tests er bestaan voor een component, hoe groter de kans dat een component robuust is.
- ...

Er zijn waarschijnlijk veel eigenschappen waarmee de kwaliteit van een component kan gemeten worden. Het vinden van een effectief, objectief systeem waarmee duidelijk de kwaliteit van een component wordt aangegeven, zou het onderwerp kunnen zijn van verder onderzoek.

De kwaliteit van componenten moet in gunstige zin evolueren. In een bibliotheek kunnen componenten gefilterd worden op basis van hun kwaliteit.

9.12.2 Performantie

Een flessenhals op één plaats kan een reden zijn om een heel softwarepakket niet te gebruiken. Er moet dus een objectief, effectief en duidelijk systeem bestaan waarmee kan gemeten worden hoe performant een component is. Het uitbouwen van zo'n systeem kan het onderwerp zijn van verder onderzoek.

Slotbeschouwing

Voordat dit werk zijn uiteindelijke vorm kreeg, heb ik een lange weg afgelegd. Eerst naar hoe een dataverwerkingssysteem kan opgebouwd worden. Ik heb veel nagedacht, gelezen, onderzocht en technologie bestudeerd.

Na een tijd waren de ideeën rijp genoeg om een prototype te ontwikkelen. De ontwikkeling gebeurde stap voor stap, want het was zoeken naar hoe het verder kon. Elke stap moest grondig overwogen en getest worden, opdat de juiste conclusies konden getrokken worden.

Aan de hand van de opgedane ideeën bedacht ik een formeel model, dat de oorspronkelijke mogelijkheden uitbreidt. Oude ideeën kwamen terug en vonden hun plaats in een nieuwe omgeving.

Ik heb een proof-of-concept gemaakt, die in principe de oorspronkelijke bedoeling realiseert. Ik vond het onderzoeken van de verdere mogelijkheden belangrijk. In de plaats van concrete, uitgewerkte toepassingen heb ik enkele eenvoudige modules ontworpen, die model kunnen staan voor meer ingewikkelde. Ik maak een onderscheid tussen programmeurs, die het raamwerk ontwikkelen en zij die op basis daarvan feitelijke toepassingen creëren. In die zin is het de bedoeling dat toepassingen ontwikkeld worden door mensen met meer achtergrondkennis, of specialisten in een bepaald vakgebied.

Voor het ontwikkelen van het raamwerk moet er theoretisch over nagedacht worden. Op die manier wordt de kans op toepasbaarheid voor ruimere mogelijkheden groter. Om na te gaan of ze mogelijk zijn, kunnen concrete toepassingen bedacht worden. De aandacht kan echter van de hoofdzaken afgeleid worden als steeds dezelfde aanwendingen in gedachten worden gehouden.

De gemaakte software werd in een ruimere context geplaatst. Ik zag meer mogelijkheden dan ik in het begin had kunnen voorzien. Het documenteren daarvan vond ik belangrijk. In de lijn van de resultaten van deze tekst, moet er immers ook aan de toekomst gedacht worden.

Opvolging

In de tekst worden tal van mogelijkheden voor verder onderzoek besproken. Voor verder onderzoek is het belangrijk bepaalde zaken in gedachten te houden. Er zijn diverse krachten die het automatiseren van softwareontwikkeling mogelijk maken.

Het scheiden van verantwoordelijkheden tussen programmeurs, ontwerpers en andere gebruikersgroepen zorgt ervoor dat iedereen zijn taak eenvoudiger kan vervullen.

Het aantal vrijheidsgraden voor het samenstellen van transformaties wordt beperkt. Door gebruik te maken van een raamwerk, kunnen regels voor het samenstellen afgedwongen worden.

Het is mogelijk heel eenvoudige tot heel complexe transformaties te implementeren. Gebruik maken van bestaande software als componenten is in principe vaak mogelijk.

Beschrijven van interfaces, in plaats van identificeren ervan zorgt ervoor dat er strenger kan gecontroleerd worden of componenten eraan voldoen. Het bepalen van een interface kan voor een groot deel zelfs de implementatie vormen.

Mogelijkheden voor gedistribueerde en multithreaded verwerking zorgen voor meer efficiëntie.

Samenwerking met met het *Semantic Web* biedt ongekende mogelijkheden.

Ik denk dat het belangrijk is dat verder onderzoek dicht aanleunt bij de realiteit. Er moet een opeenvolging zijn van het onderzoeken van bepaalde ideeën, een ontwerp op basis van die ideeën en een implementatie van dat ontwerp. Zowel frameworkprogrammeurs als programmeurs en ontwerpers moeten bij de ontwikkeling betrokken worden. Software die onder meer kan dienen om samenwerking te bevorderen, moet ook door samenwerking tussen verschillende mensen tot stand komen.

Toekomstmuziek

Of grote softwarepakketten zullen blijven bestaan, kan moeilijk voorspeld worden. Wat vrijwel zeker is, is dat ze op termijn de concurrentie zullen moeten aangaan met op maat gemaakte software. Softwarebedrijven zullen vooral het verschil moeten maken door hun specialisatie, die zich kan uiten in de componenten die ze aanbieden. Hun klanten zullen het gewoon worden dat ze op maat gemaakte producten krijgen – niet alleen wat software betreft.

Het uiteindelijke doel van systemen voor *Component Composition* kan zijn om met behulp van het *Semantic Web* software automatisch op te bouwen. Systemen voor samenstelling van componenten kunnen dus een grote stap voorwaarts betekenen op het gebied van softwareontwikkeling. Dit opent voor velen een nieuwe wereld, maar kan over enkele decennia gemeengoed zijn.

> *Component Composition* en het *Semantic Web* vormen een prachtig koppel
> en er is hen nog een lang en gelukkig leven beschoren...

Literatuurlijst

[ArgoUML, 2007] *ArgoUML* (0.24). [Software]. `http://argouml.tigris.org`, BSD licentie.

[Batik, 2007] *Batik*. [Software]. `http://xmlgraphics.apache.org/batik/`.

[Beck, 2004] Beck, K. (2004). *JUnit Pocket Guide*. O'Reilly Media, Inc.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*.

[Bloch, 2001] Bloch, J. (2001). *Effective Java: Programming Language Guide*. Addison-Wesley Professional.

[Bloch and Gafter, 2005] Bloch, J. and Gafter, N. (2005). *Java(TM) Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley Professional.

[Easycomp, 2007] Easycomp (2007). [WWW]. `http://www.info.uni-karlsruhe.de/projects.php/id=44&lang=en`.

[FindBugs, 2007] *FindBugs*. [Software]. `http://findbugs.sourceforge.net`, LGPL licentie.

[Flanagan and McLaughlin, 2004] Flanagan, D. and McLaughlin, B. (2004). *Java 5.0 Tiger: A Developer's Notebook*. O'Reilly Media, Inc.

[Freeman et al., 2004] Freeman, E., Freeman, E., Bates, B., and Sierra, K. (2004). *Head First Design Patterns*. O'Reilly Media, Inc.

[Inkscape, 2007] *Inkscape* (0.45). [Software]. `http://www.inkscape.org`.

[Instruments, 2007] National Instruments. (2007). *LabVIEW* (8.20). [Software]. `http://www.ni.com/labview/`.

[Jakarta Commons, 2007] Jakarta Commons (2007). [WWW]. `http://jakarta.apache.org/commons`.

[JavaBeans, 2007] JavaBeans (2007). [WWW]. `http://java.sun.com/products/javabeans/index.jsp`.

[JavaCC, 2006] JavaCC 4.0. [Software]. `https://javacc.dev.java.net`.

[Jetbox, 2007] Jetbox CMS 2.1. [Software]. `http://jetbox.streamedge.com`, GPL of Professional licentie.

[Jfig, 2006] Jfig 3. [Software]. `http://tams-www.informatik.uni-hamburg.de/applets/jfig/`.

[JFreeChart, 2007] *JFreeChart* (1.0.4). [Software]. `http://www.jfree.org/jfreechart/, LGPL licentie`.

[JGraph, 2006] *JGraph* (5.9.2.1). [Software]. `http://www.jgraph.com`, LGPL, Mozilla en JGraph licenties.

[kdesvn, 2007] *kdesvn* (0.12.0). [Software]. `http://www.kde-apps.org/content/show.php?content=26589`, LGPL licentie.

[McLaughlin and Edelson, 2006] McLaughlin, B. and Edelson, J. (2006). *Java & XML*. O'Reilly Media, Inc., 3rd edition.

[Naur and Randell (Eds.), 1969] Naur, P. and Randell, B. (Eds.). (1969). *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968, Section 8.2*. Scientific Affairs Division, NATO (1969), Brussels.

[NetBeans Platform, 2007] NetBeans Platform (2007). [WWW]. `http://platform.netbeans.org`.

[Oetiker et al., 2003] Oetiker, T. et al. (2003). De niet zo korte inleiding tot LaTeX 2_ε. versie 1.3, GNU GPL licentie.

[Omniscient Debugger, 2007] *Omniscient Debugger*. [Software]. `http://www.lambdacs.com/debugger/`, GPL licentie.

[PMD, 2007] *PMD* (3.9). [Software]. `http://pmd.sourceforge.net`, BSD licentie.

[Robert Simmons, 2004] Robert Simmons, J. (2004). *Hardcore Java: Secrets of the Java Masters*. O'Reilly Media, Inc.

[Scalable Vector Graphics (SVG) 1.1 Specification, 2003] Scalable Vector Graphics (SVG) 1.1 Specification (2003). [WWW]. `http://www.w3.org/tr/svg11/`.

[Subversion, 2007] *Subversion* (1.4.3). [Software]. `http://subversion.tigris.org`, Open Source: Apache/BSD licentie.

[Tortoise, 2007] *TortoiseSVN* (1.4.3). [Software]. `http://tortoisesvn.tigris.org`, vrij te gebruiken.

[xfig, 2007] *Xfig* (3.2.5). [Software]. `http://www.xfig.org`.

[Zukowski, 2006] Zukowski, J. (2006). *Java 6 Platform Revealed*. Apress.