Sketch-Based User Interface Design

Kathleen Renders

promotor :
Prof. dr. Karin CONINX

co-promotor :
Prof. dr. Kris LUYTEN

Eindverhandeling voorgedragen tot het bekomen van de graad
Master in de informatica human-computer interaction

Universiteit Maastricht

universiteit hasselt

ABSTRACT

Tijdens het user interface design proces ontwikkelen designers prototypes van de toekomstige user interface. Enerzijds om verschillende design ideeën snel uit te proberen en anderzijds om de communicatie tussen klant en stakeholders te bevorderen. Sketch-based user interface design tools bieden ondersteuning tijdens dit proces door het designers mogelijk te maken de toekomstige user interface op een natuurlijke en informele manier te schetsen. De geproduceerde schets is, in tegenstelling tot een papieren schets, interactief en beschikt dus over tal van mogelijkheden. Hoewel de layout van de toekomstige user interface erg belangrijk is, bieden deze tools weinig of geen ondersteuning voor de specificatie van layout eigenschappen zoals bijvoorbeeld alignment en adjacency van widgets. In deze thesis wordt daarom een manier beschreven om layout eigenschappen op een informele manier aan een geschetste interface gaan toe te voegen. Deze informele manier maakt gebruik van gestures en scribbles om layout commando's op user interface elementen uit te voeren. De layout van de interface wordt daarbij voorgesteld als een verzameling constraints, zodat de gespecificeerde layout eigenschappen blijven behouden en niet éénmalig gedefinieerd worden.

WOORD VOORAF

Met deze thesis zet ik een punt achter mijn studies Master Informatica aan de Universiteit Hasselt. Het tot stand komen van deze thesis was onmogelijk geweest zonder de steun en medewerking van een aantal mensen. Ik wil van deze gelegenheid gebruik maken om deze personen te bedanken.

Eerst en vooral wil ik de mensen bedanken die mij tijdens het maken van deze thesis begeleid hebben namelijk mijn promotor *Prof. dr. K. Coninx*, mijn co-promotor *Prof. dr. K. Luyten* en mijn begeleider *Yves Vandriessche*. Bedankt voor de begeleiding, interesse en talloze suggesties.

Ten tweede wil ik graag een aantal mensen bedanken die mij tijdens mijn studies hebben bijgestaan. Als eerste wil ik mijn vriend *Maarten* bedanken voor zijn aanmoediging en begrip van de voorbije jaren. Ook wil ik een woord van dank richten aan mijn *moeder* en broer *Bart* voor hun steun, geduld en waardevolle adviezen. In het bijzonder wil ik mijn *moeder* bedanken, zonder haar zou het immers niet mogelijk geweest zijn deze studies te kunnen volgen.

Tenslotte wil ik nog een aantal bijzondere medestudenten bedanken. Zonder hen zou het studentenleven er een stuk onaangenamer hebben uitgezien. Bedankt *Pascal*, *Bart* en *Jochem* voor het vele plezier en de leuke samenwerking van de voorbije jaren.

Kathleen Renders

INHOUDSOPGAVE

Hoofdstuk 1 Inleiding 12
1.1 Probleembeschrijving 12
1.2 Doel thesis 13
1.3 Opbouw document 13

Hoofdstuk 2 Prototyping 16
2.1 Overview 16
2.2 User Interface Design 16
2.3 Low-fidelity prototypes 17
2.4 Medium-fidelity prototypes 18
2.5 High-fidelity prototypes 20
2.6 Vergelijking prototypes 22
2.7 Perceptual User Interfaces 23
2.8 Conclusie 24

Hoofdstuk 3 Informele Sketching Tools 26
3.1 Overview 26
3.2 SILK 26
3.3 DENIM 28
3.4 SketchiXML 31
3.5 JavaSketchIt 33
3.6 Canonical Abstract Prototypes 34
3.7 Vergelijking sketch-based applicaties 37
3.8 Conclusie 38

Hoofdstuk 4 Layout Management 40
4.1 Overview 40
4.2 Layout proces 41
4.3 Layout management in Qt 42
4.3.1 Absolute positionering 43
4.3.2 Manuele layout 43
4.3.3 Layout managers 43
QBoxLayout 43
QGridLayout 44
QStackedLayout 45
4.3.4 Overige layout elementen 45
QSpacer 45
4.4 Layout Management in JFC 46
4.4.1 Absolute positionering 46
4.4.2 Layout managers 47
BorderLayout 47
BoxLayout 48
CardLayout 48
FlowLayout 49
GridLayout 49

GridBagLayout 50
4.4.3 Overige layout elementen 50
Rigid area 50
Glue 51
Custom filler 51
4.5 Layout Management in GTK+ 52
4.5.1 Packing widgets 52
Packing boxes 52
Packing met tabellen 52
4.5.2 Containers 53
4.6 Automated layout technieken 53
4.6.1 Learning technieken 53
4.6.2 Constraint-based technieken 55
4.6.2.1 Constraint satisfaction problem 56
4.6.2.2 One-way en multi-way constraints 57
4.6.2.3 Constraint hiërarchieën 58
4.6.2.4 Constraint solvers 59
4.6.2.5 Constraint visualisatie 60
4.7 Conclusie 61

Hoofdstuk 5 Constraint-based Systemen 62
5.1 Overview 62
5.2 Constraint-based tekenprogramma's 62
5.2.1 Sketchpad 62
5.2.2 Briar 64
5.2.3 Unidraw en QOCA 66
5.2.4 Chimera 67
5.2.5 Pegasus 68
5.3 Constraint-based window layout systemen 69
5.3.1 Scheme Constraints Window Manager 69
5.4 Constraint-based UI widget toolkits 71
5.4.1 Amulet 71
5.4.2 Bramble 72
5.4.3 OPUS 73
5.5 Vergelijking tussen constraint-based systemen 76
5.6 Conclusie 78

Hoofdstuk 6 Gestures 79
6.1 Overview 79
6.2 Gestures-based interactie 79
6.3 Structuur gesture 81
6.4 Gesture-based systemen 81
6.5 Gesture recognition algoritmen 82
6.5.1 Xstroke 82
6.5.2 Rubine 83
6.5.3 CALI library 84
6.6 Conclusie 85

Hoofdstuk 7 Software Architectuur 88
7.1 Overview 88
7.2 Overzicht doelstellingen 88
7.3 Implementatie doelstellingen 89
7.3.1 Doelstelling 1 en 2 89
7.3.2 Doelstelling 4 tot en met 7 91
7.3.3 Doelstelling 8 en 9 91
7.3.4 Doelstelling 3 en 10 93
7.4 Werking prototype 95

Hoofdstuk 8 Conclusie 97
8.1 Algemene conclusie 97
8.2 Future work 99

LIJST VAN FIGUREN

Figuur 1 - Voorbeeld van een low-fidelity prototype 18
Figuur 2 - Prototyping dimensies 18
Figuur 3 - Voorbeeld van een medium-fidelity prototype in Microsoft Visio 19
Figuur 4 - Voorbeeld van een high-fidelity prototype in Microsoft Visual Studio 21
Figuur 5 - SILK storyboard, uit [Land95] 27
Figuur 6 - Voorbeeld dialoogmodel 28
Figuur 7 - DENIM, uit [Lin02] 30
Figuur 8 - Agent oriented architectuur van SketchiXML, uit [Coye06] 32
Figuur 9 - SketchiXML 33
Figuur 10 - Links JavaSketchIt en rechts JavaSketchIt 2 34
Figuur 11 - Canonical Abstract Prototype, uit [Cons03] 35
Figuur 12 - Links QVBoxLayout, rechts QHBoxLayout 44
Figuur 13 - QGridLayout 44
Figuur 14 - QstackedLayout met een QListWidget om tussen de widgets te switchen 45
Figuur 15 - QSpacer, links in Qt Designer, rechts in uiteindelijke interface 45
Figuur 16 - Links desktop pane met interne frames, rechts split pane, uit [5] 47
Figuur 17 - BorderLayout, uit 47
Figuur 18 - BoxLayout, uit [5] 48
Figuur 19 - CardLayout, uit [5] 48
Figuur 20 - FlowLayout, uit [5] 49
Figuur 21 - GridLayout, uit [5] 49
Figuur 22 - GridBagLayout, uit [5] 50
Figuur 23 - Rigid area 51
Figuur 24 - Glue 51
Figuur 25 - Custom filler 52
Figuur 26 - Mondrian, uit [Lieb93] 54
Figuur 27 - Voorbeelden van constraints, uit [Lok01] 55
Figuur 28 - Voorbeeld van een constraint systeem, uit [Badr98b] 57
Figuur 29 - Boven one-way constraint graaf, onder multi-way constraint graaf, uit [Badr98b] 58
Figuur 30 - Grafische voorstelling van constraints in Sketchpad, uit [Suth63] 63
Figuur 31 - Constraints en alignment objects in Briar, uit [Glei94] 65
Figuur 32 - Layout constraints die opgelost kunnen worden in QOCA, uit [Helm92] 66
Figuur 33 - Scwm constraint toolbar, uit [Badr00] 70
Figuur 34 - Manipulatie van een control in Bramble, uit [Glei93] 72
Figuur 35 - Penguims line interactor object, uit [Huds93] 74

Figuur 36 - Twee OPUS UI. Frames worden voorgesteld door rechthoeken, referentie lijnen door lijnen en constraints door pijlen, uit [Huds90] 74
Figuur 37 - Xstroke full recognition, uit [Wort03] 83
Figuur 38 - Grid feature in Xstroke 83
Figuur 39 - Overzicht van vormen die herkend worden door CALI, uit [Fons02] 85
Figuur 40 - CALI herkenningsproces, uit [Fons02] 85
Figuur 41 - Prototype 90
Figuur 42 - WACOM Volito2 Tablet 90
Figuur 43 - Koppeling Cassowary 92
Figuur 44 - Koppeling CALI 95
Figuur 45 - Werking prototype: links toevoegen van UI elementen, midden selecteren dmv lasso, rechts constraint toepassen 96

LIJST VAN TABELLEN

Tabel 1 - Vergelijking tussen prototypes 22
Tabel 2 - Vergelijkend overzicht van sketch-based applicaties 37
Tabel 3 - Overzicht Chimera constraints, uit [Kurl93] 68
Tabel 4 - Overzicht Scwm constraints 70
Tabel 5 - Vergelijkend overzicht van constraint-based applicaties 77
Tabel 6 - Overzicht prototype doelstellingen 89
Tabel 7 - Overzicht koppeling tussen commando's en gestures 94

DEEL 1 INLEIDING

Hoofdstuk 1

Inleiding

1.1 Probleembeschrijving

De titel van deze thesis is “sketch-based user interface design”. Zoals in de volgende hoofdstukken uitgebreid zal worden besproken, bieden sketch-based user interfaces de mogelijkheid om een user interface op een snelle en flexibele manier te schetsen. Deze sketch-based user interfaces maken daarvoor meestal gebruik van pen-based interactietechnieken zoals gestures en scribbles. Op die manier kan de designer zich op een informele manier gaan uitdrukken, en kan het schetsen op een even natuurlijke manier gebeuren als het schetsen met pen en papier. In tegenstelling tot een papieren schets, is de elektronische schets interactief waardoor de capaciteiten van de computer ten volle benut kunnen worden en de elektronische schets dus over een groter aantal mogelijkheden beschikt. Tal van sketch-based user interface zijn reeds met succes ontwikkeld. Een aantal daarvan zullen verder in deze thesis worden behandeld.

Spijtig genoeg merken we dat huidige sketch-based user interfaces over weinig of zelfs geen mogelijkheden beschikken om de layout eigenschappen van de geschetste user interface en bijhorende UI elementen te definiëren. Toch zou dit een meerwaarde voor de huidige sketch-based user interfaces kunnen betekenen. Wanneer immers gekeken wordt naar toolkits en user interface builders, die nog door veel designers worden gebruikt voor de ontwikkeling van high-fidelity prototypes, merken we op dat deze uitgerust zijn met tal van mogelijkheden voor het specificeren van layout eigenschappen die reeds hun succes hebben bewezen. Er wordt hier bijvoorbeeld gedacht aan het uitlijnen van elementen, het definiëren van adjacency-constraints en het gebruik van layout managers en spacer widgets.

Wanneer deze layout mogelijkheden in huidige sketch-based user interfaces geïntegreerd zouden kunnen worden, zou het design proces nog beter ondersteund kunnen worden dan reeds het geval is. Via deze thesis zullen de mogelijkheden van een dergelijke integratie onderzocht worden.

1.2 Doel thesis

Met het schrijven van deze thesis wens ik twee doelen te bereiken. Enerzijds wil ik via een uit te voeren *literatuurstudie* een overzicht geven van de huidige stand van zaken wat betreft sketch-based user interfaces. Hierbij wil ik de werking en de mogelijkheden tot het definiëren van layout eigenschappen van naderbij bekijken.

Via de literatuurstudie wil ik eveneens een beeld geven van de mogelijkheden die momenteel bestaan op gebied van layout management. Hierbij zal vooral aandacht worden besteed aan het definiëren van een layout door middel van constraints. Er wordt eveneens een overzicht gegeven van een aantal constraint-based systemen. Als laatste wil ik ook de werking van gestures verder bekijken en onderzoeken op welke manier zij een rol kunnen spelen in de koppeling tussen layout eigenschappen en constraints.

Het tweede doel dat ik met deze thesis wens te bereiken, is de ontwikkeling van een *prototype*. Via dit prototype moeten de verschillende technieken, die in de uitgevoerde literatuurstudie worden besproken, met elkaar worden gekoppeld. Meerbepaald zal het hier gaan om de koppeling tussen layout commando's en constraints door middel van gestures. Het is echter niet de bedoeling een volledige informele sketching tool te creëren, maar eerder de mogelijkheden van een combinatie tussen gestures, constraints en layout eigenschappen te testen. De vereisten waaraan het te ontwikkelen prototype moet voldoen, zullen in de loop van de volgende hoofdstukken geformuleerd worden.

1.3 Opbouw document

Deze thesis kan worden opgesplitst in twee grote onderdelen. In het eerste gedeelte wordt een samenvatting gegeven van de uitgevoerde literatuurstudie. Deze samenvatting loopt van hoofdstuk 2 tot en met hoofdstuk 6.

Hoofdstuk 2 geeft een inleiding tot het ontstaan van perceptuele informele applicaties. Daarnaast wordt het gebruik van low-fidelity, medium-fidelity en high-fidelity prototypes tijdens de ontwikkeling van user interfaces besproken. De voor- en nadelen van elk soort prototype worden opgesomd.

In *hoofdstuk 3* wordt een overzicht gegeven van de informele sketching tools die zich momenteel op de markt bevinden. Elke sketch-based ontwikkelingstool wordt in het kort besproken. Hierbij wordt eveneens aandacht gegeven aan de layout mogelijkheden waarover de tool beschikt. Op het einde van het hoofdstuk wordt een vergelijking gemaakt tussen de verschillende tools.

In *hoofdstuk 4* wordt een overzicht gegeven van de verschillende mogelijkheden op het gebied van layout management. Het specificeren van layout eigenschappen binnen toolkits zoals Qt, JFC en GTK+ wordt uitvoerig besproken. Hierbij wordt een overzicht gegeven van de beschikbare layout managers en hun werking. Daarnaast wordt er ook een inleiding gegeven tot automated layout systemen. De werking van constraint-based systemen en learning techniques wordt besproken.

Hoofdstuk 5 gaat dieper in op de werking van constraint-based systemen en geeft een overzicht van een aantal constraint-based drawing systemen en hun mogelijkheden. Op het einde van het hoofdstuk wordt een vergelijkend overzicht gegeven.

Hoofdstuk 6 geeft een inleiding tot gestures en het gebruik ervan in huidige applicaties. Er wordt in dit hoofdstuk eveneens gekeken naar de werking van een aantal gesture recognition algoritmen.

Het tweede onderdeel van deze thesis heeft betrekking op de ontwikkeling van het prototype. In *hoofdstuk 7* zal enerzijds de werking en de mogelijkheden van het ontwikkelde prototype worden uitgelegd. Anderzijds zal de integratie met de verschillende onderdelen, zoals de gesture recognizer en de constraint solver, verder worden besproken.

De thesis wordt tenslotte afgesloten met *hoofdstuk 8* waarin de belangrijkste conclusies geformuleerd worden. In dit hoofdstuk wordt ook een overzicht gegeven van mogelijke uitbreidingen en toekomstige mogelijkheden.

DEEL 2 LITERATUURSTUDIE

Hoofdstuk 2

Prototyping

2.1 Overview

Tijdens het user interface design proces ontwikkelen designers prototypes van de toekomstige user interface. Enerzijds om snel verschillende ideeën uit te proberen, anderzijds om te communiceren met de klant en de overige teamleden. Deze prototypes kunnen in drie groepen worden ingedeeld naargelang hun gelijkenis met de uiteindelijke user interface namelijk low-fidelity, medium-fidelity en high-fidelity prototypes. Deze prototypes, en hun rol binnen het design proces, worden in dit hoofdstuk verder besproken. Eveneens wordt een overzicht gegeven van hun voor- en nadelen. Tenslotte wordt een inleiding gegeven tot informele sketching tools, waarin de voor- en nadelen van low-fidelity en high-fidelity technieken gecombineerd worden.

2.2 User Interface Design

In [Coye06] wordt user interface design gedefinieerd als een *iteratief*, *open* en *onvolledig* proces. *Iteratief* omdat het ontwerpen van een user interface meestal in een aantal cycli gebeurt om uiteindelijk tot het gewenste resultaat te komen. *Open* omdat de ontwikkeling van een interactieve applicatie niet altijd een voorgedefinieerd proces volgt, en tenslotte *onvolledig* omdat niet alle requirements waaraan de applicatie moet voldoen op elk ogenblik gekend zijn. Daarnaast zijn de requirements ook vaak onvolledig, dubbelzinnig en hebben ze geen specifiek doel voor ogen [Coye05].

Tijdens dit proces maken designers prototypes van de toekomstige user interface. Een prototype is een model dat gebouwd wordt om design ideeën te ontwikkelen en te

testen [Walk02]. Het heeft als doel de algemene layout en structuur van componenten uit te werken. Eveneens worden prototypes in User Centered Design gebruikt om de eindgebruiker te betrekken bij het ontwikkelingsproces en zo feedback van hen te krijgen [Coye05] [Land96].

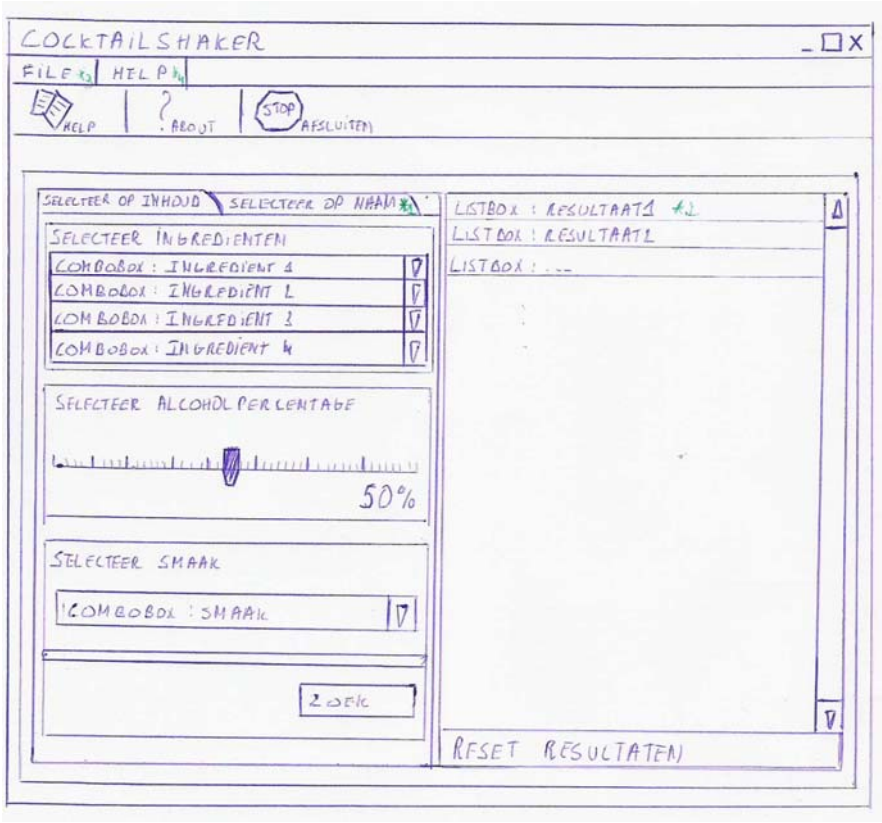
Prototypes kunnen worden onderverdeeld in drie klassen [Coye06]: high-fidelity, medium-fidelity en low-fidelity prototypes. Deze indeling gebeurt op basis van de gelijkenis met de uiteindelijke user interface. In de volgende paragrafen worden de verschillende klassen besproken.

2.3 Low-fidelity prototypes

Low-fidelity prototypes worden gebruikt om de algemene structuur van de user interface te schetsen. De focus wordt gelegd op scherm layout, design concepten en alternatieven zonder daarbij aandacht te schenken aan onnodige details zoals bijvoorbeeld kleur of lettertype. Niet de details en de achterliggende functionaliteit zijn belangrijk, wel het algemene beeld van de user interface. Voorbeelden van low-fidelity prototyping technieken zijn pen en papier, whiteboard of blackboard en Post-it® notes.

Het gebruik van low-fidelity prototypes in het design proces heeft een aantal voordelen. Ze kunnen snel en op eender welk ogenblik worden ontworpen. Ze ondersteunen eveneens de creativiteit van de designer door de aandacht te vestigen op belangrijke kenmerken in plaats van op details. Hierdoor hebben designers meer tijd om alternatieven te bekijken en iteraties te doorlopen. Daarnaast zijn low-fidelity prototypes ook bevorderlijk voor de communicatie. Ze kunnen collaboratief worden gebruikt en ze vergemakkelijken eveneens de validatie en het afsluiten van een contract met de klant [Coye06]. In figuur 1 wordt een voorbeeld van een Lo-Fi prototype gegeven.

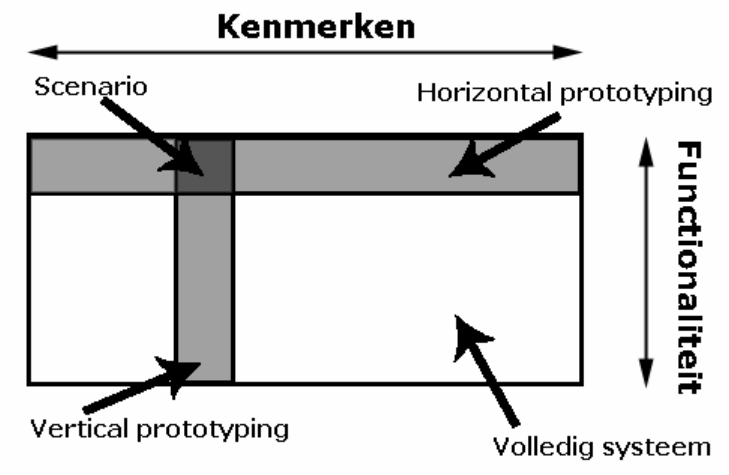
Toch zijn er ook een aantal nadelen verbonden aan het gebruik van low-fidelity prototypes [Coye05] [Coye06] [Land96]. Wanneer het geproduceerde prototype niet kan worden omgezet naar een elektronische versie op computer, moet de designer het volledige prototype gaan herimplementeren. Hierbij is er geen zekerheid dat de consistentie tussen beide prototypes blijft bestaan. Daarnaast hebben de meeste low-fidelity prototyping technieken ook niet de mogelijkheid om gemeenschappelijke elementen in schermen te gaan kopiëren of te gaan hergebruiken. Eveneens is het moeilijk om veranderingen aan te brengen en is het daardoor vaak nodig schetsen te hertekenen. Er is ook een gebrek aan “design memory”: er kunnen wel aantekeningen op de schetsen worden gemaakt maar deze kunnen bijvoorbeeld niet worden doorzocht. Sketches op papier zijn over het algemeen moeilijk te doorzoeken, te organiseren of te hergebruiken. Er is ook geen interactie met de gebruiker mogelijk.



Figuur 1 - Voorbeeld van een low-fidelity prototype

2.4 Medium-fidelity prototypes

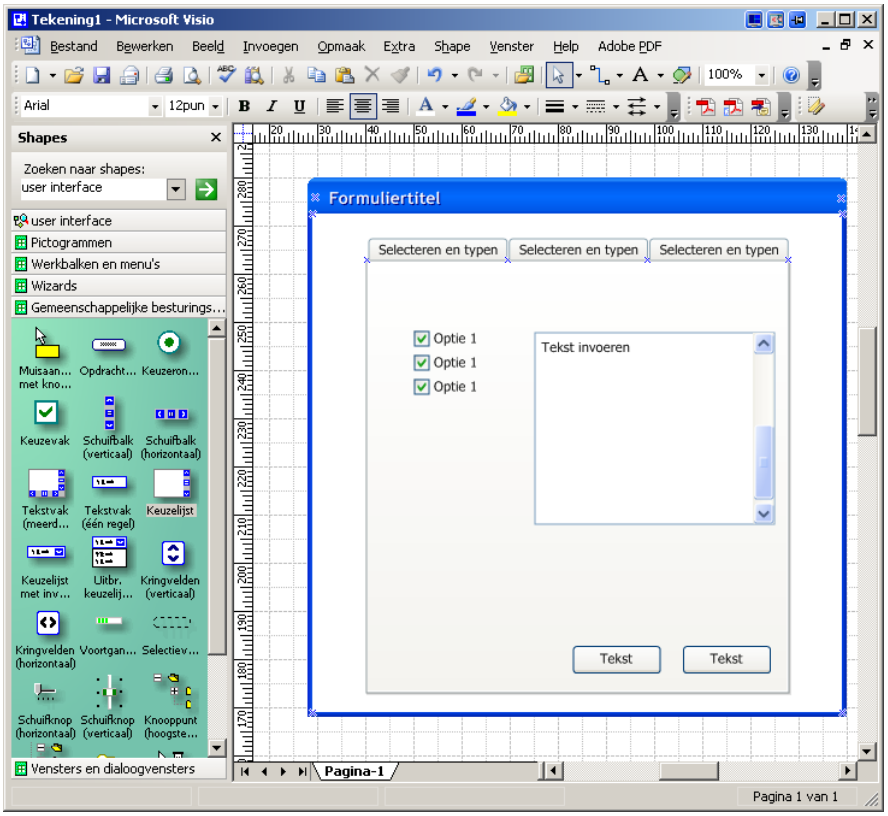
Medium-fidelity prototypes [Petr06] zijn een meer verfijnde vorm van low-fidelity prototypes die gecreëerd worden op computer en waarin een aantal kenmerken van de toekomstige user interface worden gesimuleerd. Ze worden vaak gebruikt om testen bij gebruikers uit te voeren. Hier zijn drie verschillende aanpakken mogelijk: *vertical prototyping*, *horizontal prototyping* en *scenarios* [Niel93].



Figuur 2 - Prototyping dimensies

Bij *vertical prototyping* worden slechts een beperkt aantal kenmerken van het systeem geïmplementeerd maar dit wel met diepgaande functionaliteit. Een aantal taken van de toekomstige user interface kunnen hierdoor worden getest. Bij *horizontal prototyping* daarentegen wordt de volledige user interface geïmplementeerd zonder enige functionaliteit. Ze bieden de mogelijkheid om de totale “look and feel” van de interface aan de eindgebruiker te presenteren. *Scenarios* beperken beide, zowel het aantal geïmplementeerde kenmerken als de onderliggende functionaliteit. Ze kunnen gebruikt worden om de toekomstige user interface te testen zolang de testgebruiker zich aan een voorgedefinieerd parcours houdt. Figuur 2 geeft een overzicht van de verschillende prototyping dimensies [Niel93].

Een typisch voorbeeld van een medium-fidelity prototyping tool is Microsoft Visio waarvan een voorbeeld wordt gegeven in figuur 3. In Visio is het enkel mogelijk het type, de grootte en de inhoud van user interface widgets grafisch te specificeren. Er wordt verder geen aandacht besteed aan typografie en kleurenschema’s. Andere voorbeelden van medium-fidelity prototyping tools zijn HyperCard en Director.



Figuur 3 - Voorbeeld van een medium-fidelity prototype in Microsoft Visio

Medium-fidelity prototypes hebben een aantal voordelen ten opzichte van low-fidelity prototypes. Omdat ze enige vorm van functionaliteit implementeren, worden ze interactief. Daardoor kunnen ze gemakkelijk gebruikt worden bij testen en evaluaties. In

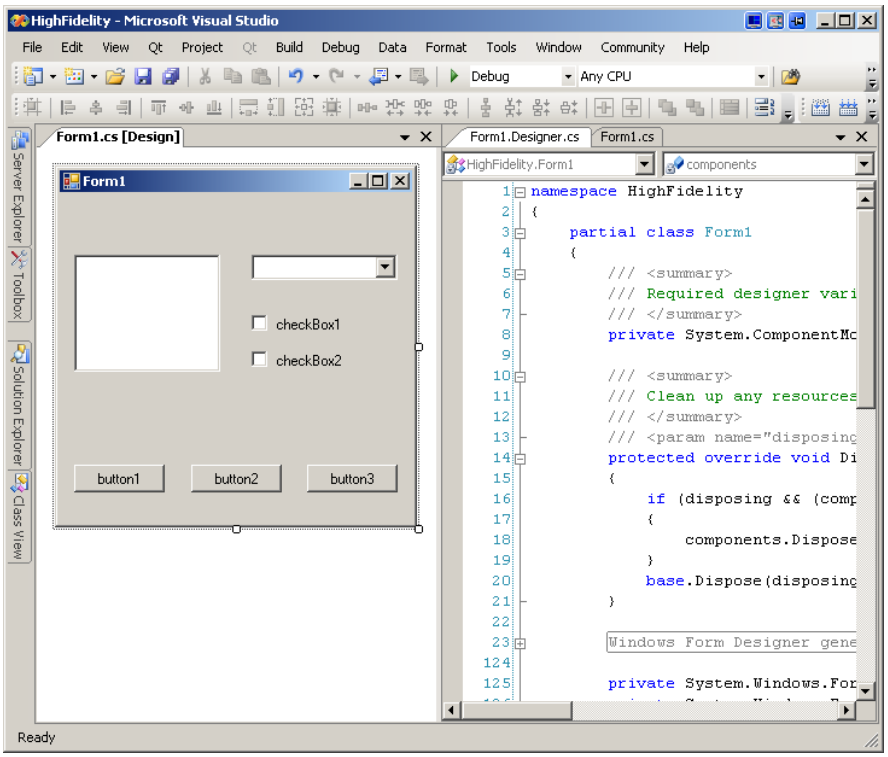
sommige gevallen worden medium-fidelity prototypes ook gebruikt als marketing of sales tool. Het gevaar is dan wel dat de aandacht van de eindgebruiker of manager zich gaat vestigen op onbelangrijke details waardoor ze zich een verkeerd beeld van de toekomstige user interface kunnen vormen. Een tweede nadeel is dat medium-fidelity prototyping tools geen code generatie ondersteunen waardoor het gecreëerde prototype niet hergebruikt kan worden in een later stadium van het design proces. Een laatste nadeel is het feit dat voor het produceren van een medium-fidelity prototype meer tijd nodig is dan voor de creatie van een low-fidelity prototype [Niel93].

2.5 High-fidelity prototypes

High-fidelity prototypes geven een volledig beeld van hoe de toekomstige user interface er zal uit zien. Ze worden vaak met dezelfde methode ontwikkeld als de uiteindelijke user interface en zijn daarom meestal uitgerust met dezelfde interactietechnieken en hetzelfde uiterlijk als de uiteindelijke user interface. High-fidelity prototyping tools ondersteunen dus de ontwikkeling van een volledige user interface. Ze bieden de mogelijkheid om een afbeelding (bijvoorbeeld via MS Powerpoint of Adobe Photoshop) of code (bijvoorbeeld via Macromedia Dreamweaver of MS Visual Basic) van de geconstrueerde user interface te genereren zodat deze in een later stadium van het design proces hergebruikt kan worden [Coye06] [Walk02].

High-fidelity prototypes hebben volgende voordelen. De prototypes die gecreëerd worden zijn volledig interactief en geven een goed beeld van hoe de totale “look and feel” van de uiteindelijke user interface er zal uit zien. Ze kunnen dus net zoals medium-fidelity prototypes gebruikt worden bij testen, evaluaties en marketing doeleinden. Een tweede grote voordeel is de mogelijkheid om code of een andere presentatie van de user interface te genereren, die dan in een later stadium van het design proces hergebruikt kan worden [Coye05].

Eveneens kunnen de eigenschappen van user interface elementen zoals uitlijning en locatie heel gemakkelijk gespecificeerd worden. Vaak bestaan UI construction tools uit een canvas en een palet waarbij componenten van het palet naar het canvas worden gesleept waardoor de exacte locatie en grootte van de objecten onmiddellijk is gekend [Land02]. Daarnaast zijn deze tools ook meestal uitgerust met een aantal editing functies voor user interface widgets zoals undo, move, erase,… zodat het prototype gemakkelijk kan worden aangepast.



Figuur 4 - Voorbeeld van een high-fidelity prototype in Microsoft Visual Studio

Toch is het in high-fidelity prototyping tools moeilijk om het gedrag van de interface te specificeren [Land95]. Vaak is een programmeer- of scripting taal nodig om acties te definiëren. Over het algemeen is er ook meer tijd nodig voor de ontwikkeling van Hi-Fi prototypes en ligt de kost ervan hoger dan die van Lo-Fi en Me-Fi prototypes [Walk02] [Coye05]. Het grootste nadeel is het feit dat deze tools geen natuurlijke vorm van communicatie toelaten. In tegenstelling tot low-fidelity technieken is het hier niet mogelijk de toekomstige user interface op een natuurlijke manier te gaan schetsen [Land02].

2.6 Vergelijking prototypes

In tabel 1 worden de voor- en nadelen van elk prototype nogmaals opgesomd.

	Lo-Fi Prototypes	**Me-Fi Prototypes**	**Hi-Fi Prototypes**
Voordelen	- Snel - Goedkoop - Ondersteunt natuurlijke vorm van communicatie tussen stakeholders - Kan op elk ogenblik gecreëerd worden - Geen speciale kennis vereist - Aandacht wordt gevestigd op belangrijke elementen zoals structuur,… - Collaboratief	- Interactief - Bruikbaar voor testen en evaluaties - Bruikbaar als marketing tool	- Volledig beeld van UI - Interactief - Generatie van code of andere presentatie - Layout eigenschappen makkelijk te definiëren - Bruikbaar voor testen en evaluaties - Bruikbaar als marketing tool

Nadelen	- Niet interactief
	- Geen functionaliteit
	- Geen mogelijkheid tot kopiëren,…
	- Moeilijk veranderingen aan te brengen
	- Gebrek aan design memory
	- Geen code generatie
		- Hogere ontwikkelingskost:
		  - meer tijd nodig om te ontwikkelen
		- Focus op details
		- Geen code generatie
			- Gedrag moeilijk te specificeren
			- Hogere ontwikkelingskost:
			  - kennis van programmeer- of scripting taal vereist
			  - meer tijd nodig om te ontwikkelen
			- Geen natuurlijke vorm van interactie mogelijk
			- Focus op details

Tabel 1 - Vergelijking tussen prototypes

2.7 Perceptual User Interfaces

In het begin van het ontwikkelingsproces maken designers voornamelijk gebruik van low-fidelity technieken. Deze technieken laten een snelle exploratie van verschillende design ideeën toe en bieden een goede ondersteuning bij brainstorming sessies. Op een bepaald ogenblik wordt er echter overgeschakeld naar medium- of high-fidelity technieken waardoor het reeds ontwikkelde Lo-Fi prototype verloren gaat. De laatste jaren wordt er daarom veel onderzoek verricht naar een soort hybride aanpak die de voordelen van low-fidelity en high-fidelity technieken gaat combineren en die gebruikt kan worden tijdens alle stadia van het ontwikkelingsproces. Deze hybride aanpak worden "sketch-based user interfaces" genoemd en zijn een vorm van informele *Perceptual User Interfaces* [Land96].

Perceptual User Interfaces (PUIs) trachten de beperkingen van huidige user interfaces te overkomen door human-computer interaction natuurlijker te maken. Deze interfaces trachten de acties van de gebruiker waar te nemen via computer vision, speech recognition en sketch recognition. PUIs kunnen worden verdeeld in twee groepen: informele en formele user interfaces. In formele user interfaces wordt de menselijke input onmiddellijk geïnterpreteerd. Deze onmiddellijke herkenning legt echter een structuur vast op de input, die in de weg kan komen te staan bij creatieve of communicatiegerichte taken, zoals bijvoorbeeld brainstorming of design. Een aantal domeinen, waaronder *sketching*, *speech* en *handwriting*, zijn echter beter af met een meer informele aanpak. Daarom wordt er de laatste jaren veel onderzoek gedaan naar informele user interfaces die een natuurlijke, ambigue vorm van human-computer interaction ondersteunen zonder al te veel up-front recognition of transformatie van de input [Land02].

Design is zo een domein waar een informele aanpak meer gepast zou zijn. Huidige design applicaties vragen over het algemeen de gebruiker objecten van een palet op een canvas te slepen waar deze dan gemanipuleerd kunnen worden met de muis. De objecten worden op een exacte plaats met een exacte grootte geplaatst. Die precisie zorgt ervoor dat ze gemakkelijk door de computer kunnen worden voorgesteld. Bij het ontwerpen van CAD tekeningen is die precisie ook belangrijk. Maar deze manier van tekenen is echter totaal verschillend van hoe wij gewend zijn onze tekeningen te maken zonder computers [Land02].

Sketchen is een voorbeeld van een informele, perceptuele interactievorm die heel waardevol kan zijn bij creatieve design taken. Belangrijk hierbij is dat tijdens het schetsen ambiguïteit wordt toegestaan zodat de designer de mogelijkheid krijgt om verschillende design ideeën te verkennen. Het is mogelijk dat een designer een aantal interpretaties van een bepaald object in gedachten heeft, maar dat deze te weinig gedetailleerd zijn om al

een vaste representatie op te leggen. De ambiguïteit van een schets zorgt ervoor dat hij die beslissing kan uitstellen tot een later stadium in het design proces [Land02] [Gros96].

Een ander voorbeeld van een perceptuele manier van interactie is *spraak*. Hoewel spraak iets typisch menselijk en natuurlijk is, hebben huidige speech recognition systemen geen informele user interface. De meeste systemen falen zelfs wanneer ze in contact komen met ambiguïteit. Ze trachten menselijke spraak te interpreteren en om te zetten in een precieze representatie waarmee machines overweg kunnen. Daarom eisen ze vaak tussenkomst van de gebruiker om foute interpretaties te verbeteren. Dit alles staat een natuurlijke vorm van interactie in de weg. Een voorbeeld van een speech recognition systeem met een informele interface is SUEDE [Land02]. De werking van deze tool valt echter buiten de scope van de thesis en zal hier dus niet verder worden besproken.

Handwriting is een derde voorbeeld van perceptuele interactie. Ook deze systemen kampen met dezelfde problemen als speech recognition systemen. Vele handwriting recognition systemen trachten een onmiddellijke herkenning te doen van wat de gebruiker schrijft. Daardoor wordt de aandacht van de gebruiker van de oorspronkelijke taak afgeleidt [Land02].

2.8 Conclusie

In dit hoofdstuk werd een overzicht gegeven van de verschillende prototypes die tijdens het ontwikkelingsproces worden ontwikkeld. Low-fidelity prototypes zijn snel en gemakkelijk te ontwikkelen maar hebben het nadeel dat ze geen computerondersteuning bieden, waardoor het genereren van code of het ondersteunen van typische computermogelijkheden zoals kopiëren onmogelijk worden. Medium-fidelity prototypes bieden wel die computerondersteuning, maar beschikken niet over de mogelijkheid om code te genereren. High-fidelity prototypes kunnen dan wel weer worden omgezet in code, maar maken daarvoor gebruik van een programmeer- of scripting taal waarvoor een specifieke kennis vereist is. Ze laten daardoor geen natuurlijke vorm van communicatie toe. Daarom worden er informele perceptuele user interface tools ontwikkeld die de voor- en nadelen van de verschillende prototyping technieken met elkaar tracht te combineren. In het kader van deze thesis zijn we vooral geïnteresseerd in de werking van sketch-based user interfaces. Dit zijn informele tools die ontwikkeld zijn om ondersteuning te bieden tijdens het user interface design proces.

Op het einde van elk hoofdstuk worden op basis van de besproken literatuur een aantal doelstellingen geformuleerd waaraan het te ontwikkelen prototype moet gaan voldoen. Een eerste doelstelling die hier vooropgesteld kan worden is de volgende:

> “Doelstelling 1: Het prototype moet een natuurlijke vorm van interactie ondersteunen. De gebruiker moet in staat zijn user interface elementen te gaan schetsen met behulp van een stylus en elektronisch pad.”

De doelstellingen die in de loop van de hoofdstukken geformuleerd worden, worden in hoofdstuk 7 samengevat. Daar wordt vervolgens per doelstelling besproken hoe deze in het prototype werd geïntegreerd.

Hoofdstuk 3

Informele Sketching Tools

3.1 Overview

Zoals reeds besproken werd in het vorige hoofdstuk is “schetsen” een voorbeeld van een informele, perceptuele interactietechniek. De tools die in dit hoofdstuk worden besproken, zijn ontwikkeld om deze vorm van interactie mogelijk te maken. Via deze tools kan een designer een toekomstige user interface op een natuurlijke manier gaan schetsen met behulp van een muis of een stylus en elektronisch pad. De getekende schets is, in tegenstelling tot een papieren schets, interactief. In de volgende alinea’s zullen een aantal tools besproken worden. Op het einde van het hoofdstuk wordt een vergelijkend overzicht gegeven.

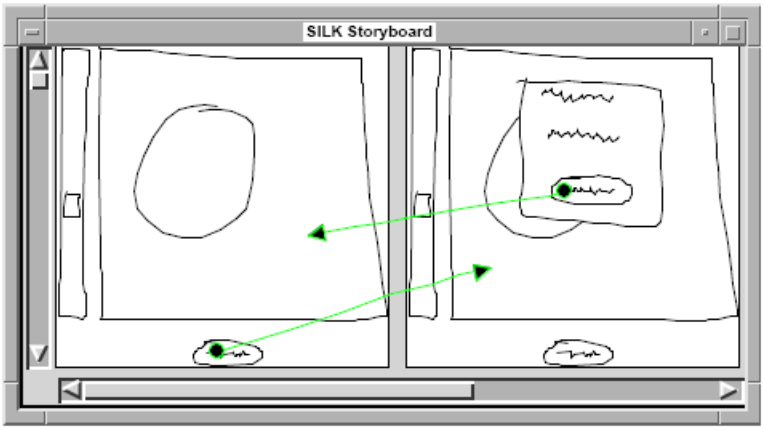
3.2 SILK

SILK [Land95] [Land96] is een elektronische sketching tool die het designers mogelijk maakt de algemene layout en structuur van de toekomstige interface op een snelle manier te schetsen door gebruik te maken van een elektronisch pad en stylus. Daarnaast biedt SILK eveneens de mogelijkheid om ook het gedrag van de interface al in een zeer vroeg stadium te specificeren. SILK maakt hiervoor gebruik van een storyboard mechanisme.

Dit mechanisme werkt met het aanbrengen van visuele notaties op de interface, die gelijkaardig zijn aan de notaties die worden gebruikt bij het schetsen van een interface op papier. Het storyboard wordt opgebouwd door geschetste schermen vanuit het sketch window naar het storyboard te kopiëren. De designer kan vervolgens notaties aanbrengen

tussen kopieën van schermen om zo de reacties van het systeem op een actie van de gebruiker weer te geven [Land95].

In figuur 5 wordt een voorbeeld gegeven van een SILK storyboard. Er wordt gebruik gemaakt van pijlen en schermen om het dynamische gedrag weer te geven. Een scherm is een schets van een interface in een bepaalde state. Pijlen verbinden verschillende objecten met elkaar. Figuur 5 geeft het oproepen van een nieuwe dialoog weer.

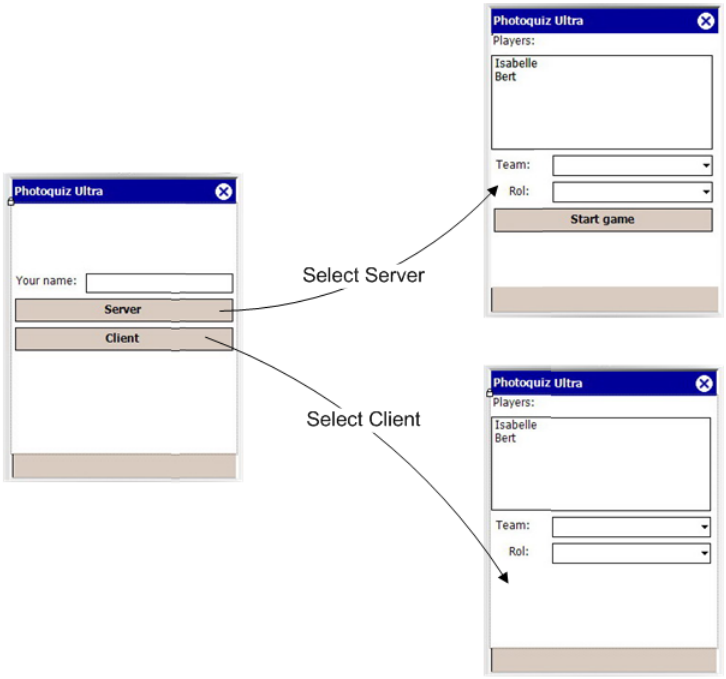


Figuur 5 - SILK storyboard, uit [Land95]

Het storyboard mechanisme van SILK vertoont een groot aantal gelijkenissen met het dialoogmodel dat tijdens model-based user interface design wordt ontwikkeld. Model-based UI design maakt gebruik van een aantal semi-formele representaties om de toekomstige user interface gaan voor te stellen. Elk model belicht een verschillend perspectief van de toekomstige user interface [Traet02]. Het dialoogmodel is één van de modellen dat gebruikt kan worden. In dit model wordt de toekomstige user interface voorgesteld als een graaf, waarin elke node een toekomstige dialoog van de user interface voorstelt. De dialogen worden met elkaar verbonden door lijnen die de transities tussen de dialogen weergeven. Figuur 6 geeft een voorbeeld van een dialoogmodel.

SILK ondersteunt vier modi: sketch, run, annotate en decorate mode. In sketch mode kan de designer objecten aanmaken en bewerken. De getekende objecten worden onmiddellijk herkend, hoewel de gebruiker dit enkel zal merken bij het uittesten van de interface. In run mode kan de designer de getekende interface gaan testen. Getekende knoppen kunnen worden gebruikt en de overgang tussen verschillende schermen wordt eveneens weergegeven. In annotate mode is het mogelijk om aantekeningen toe te voegen aan de schetsen. De aantekeningen kunnen getekend, geschreven of ingetypt worden. Ze worden in een blauwe kleur weergegeven op een speciale laag die verborgen kan worden.

Decorate mode werkt gelijkaardig als annotate mode. In decorate mode kan de gebruiker tekst en afbeeldingen toevoegen die hij niet als een UI widget wenst te laten herkennen. Deze decoraties worden eveneens op een aparte laag weergegeven in een groene kleur [Land96].



Figuur 6 - Voorbeeld dialoogmodel

SILK biedt de mogelijkheid om de geschetste user interface te transformeren naar Visual Basic 5 of Common Lisp code. Eveneens kan de user interface getransformeerd worden naar een standaard grafische user interface zoals Motif, Windows of Macintosh. Deze transformatie gebeurt niet volledig automatisch. Er wordt een kleine tussenkomst van de designer vereist voor het specificeren van een aantal details zoals tekst labels, kleuren, ed … [Land01].

3.3 DENIM

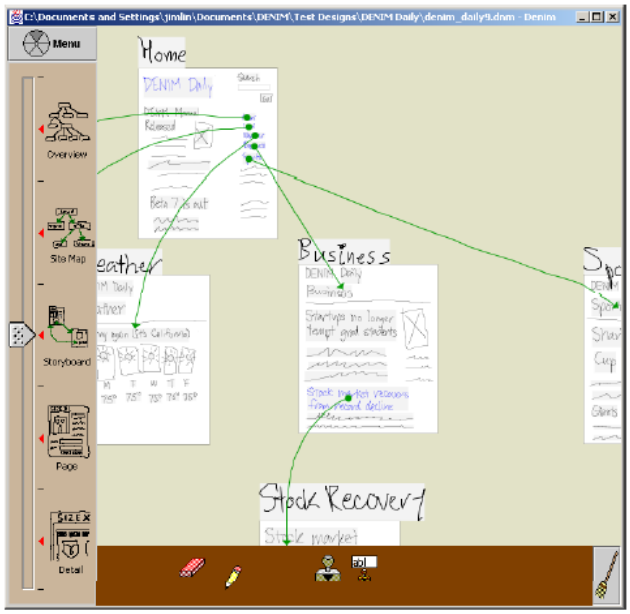
DENIM [Lin00] [Lin02] is een informele sketching tool die gebruikt kan worden bij het ontwerpen van websites. De tool is eveneens ontworpen door Landay en is gebaseerd op dezelfde principes als SILK. DENIM beschikt echter over een geavanceerdere visuele taal, die het mogelijk maakt componenten te gaan hergebruiken en transities te specificeren tussen interface elementen. Daardoor wordt het mogelijk complexere en meer interactieve interfaces te creëren.

In SILK werd reeds gebruik gemaakt van een visuele taal om het gedrag van de interface te schetsen in een storyboard. Die taal maakt gebruik van pagina's en pijlen. DENIM blijft daar eveneens gebruik van maken, maar de taal wordt uitgebreid met een aantal extra elementen waaronder *components*, *enhanced arrows* en *conditionals*. *Components* maken het designers mogelijk herbruikbare widgets en storyboard fragmenten te creëren. Er zijn twee types: intrinsic components en custom components. Intrinsic components zijn standaard user interface elementen zoals bijvoorbeeld text fields. Custom components kunnen door de designer zelf gedefinieerd worden zodat het voor mogelijk wordt eigen "building blocks" te gaan creëren. *Enhanced arrows* maken het mogelijk om in het storyboard gebruik te maken van pijlen die meerdere events ondersteunen. Daarvoor was het enkel mogelijk een linker muisklik te definiëren. Via enhanced arrows kunnen meerdere events ondersteund worden zoals bijvoorbeeld dubbelklikken en rollovers. Tenslotte zorgen *conditionals* ervoor dat transities kunnen afhangen van de status van bepaalde interface elementen zoals bijvoorbeeld van de status van een check box [Lin02].

Tijdens het ontwerpen van een website doorlopen designers een proces van verfijning [Lin00]. Daarin worden visualisaties gecreëerd met verschillende niveaus van detail. DENIM ondersteunt dit proces door vijf zoom levels te introduceren, die elk een bepaald detailniveau visualiseren. Het eerste en meest abstracte level is het *overview* level. Hier wordt een algemene structuur van de site weergegeven, zonder verdere details. Het tweede level, de *site map*, geeft eveneens een algemeen overzicht van de gehele site, maar met iets meer details, zo zijn bijvoorbeeld de namen van de geschetste pagina's zichtbaar. Het derde level is het *storyboard*. Hier worden verschillende pagina's, met hun onderlinge relaties, gelijktijdig weergegeven. Het *sketch* level toont één pagina op realistische schaal en maakt het op die manier mogelijk om de inhoud van de pagina te schetsen. Als laatste is er het *detail* level dat een nog fijner zicht op een individuele pagina toelaat om zo een preciezere vorm van sketching mogelijk te maken. De designer kan tussen de verschillende levels manoeuvreren via zooming. DENIM legt, integendeel tot SILK, dus niet alleen de aandacht op de creatie van individuele schermen maar maakt het mogelijk een gehele website te creëren. Daarnaast zijn ook alle levels met elkaar geïntegreerd via een vorm van semantisch zoom.

DENIM's interface is opgebouwd uit drie gebieden [Lin02]. In het midden bevindt zich een canvas waar de gebruiker de pagina's en hun inhoud kan gaan schetsen. Het linkergedeelte bestaat uit een slider die gebruikt kan worden om het huidige zoom level in te stellen. Onderaan is er een toolbox met tools voor erasing, panning,…. Een interessante tool in deze toolbox is de "Grouping Pencil" tool. Deze tool maakt het mogelijk geschreven woorden tekst met elkaar te gaan groeperen. De tekst kan achteraf terug worden opgesplitst in twee delen door een horizontale of verticale beweging te maken met de stylus. Deze vorm van groepering kan eveneens worden toegepast op radio buttons

[Deni]. Naast deze tools maakt DENIM voornamelijk gebruik van pen-based interactietechnieken zoals gestures en pie menus [Lin00] [Lin02]. Er zijn gestures geïmplementeerd voor panning, undo, redo, group, select, cut, copy en paste via een aangepaste versie van gdt [Long99] en Rubine's recognizer [Rubi91a]. Commando's die niet gemakkelijk via gestures uitgevoerd kunnen worden, werden geïmplementeerd via pie menus. De gebruiker kan het pie menu activeren door met de pen op het scherm te tappen.



Figuur 7 - DENIM, uit [Lin02]

In tegenstelling tot SILK, die een onmiddellijke herkenning van de input uitvoert, stelt DENIM deze herkenning zo lang mogelijk uit om een meer vrije vorm van sketching mogelijk te maken. DENIM doet een minimale vorm van herkenning. Woorden worden herkend als nieuwe page labels en lijnen tussen twee pagina's worden herkend als links tussen deze pagina's. Daarnaast worden een aantal voorgedefinieerde gestures herkend.

Het uittesten van de schermen in run mode gebeurt op dezelfde manier als in SILK. Zodra de gebruiker een aantal schermen geschetst heeft, heeft hij de mogelijkheid deze schermen uit te testen. In run mode verschijnt er een simpele browser op het scherm die één enkele pagina toont. De getekende pagina's kunnen uiteindelijk geëxporteerd worden naar HTML [Lin00].

DENIM werd geïmplementeerd met behulp van Java 2 SDK versie 1.3 en SATIN [Hong02], een toolkit die gebruikt kan worden bij het ontwerpen van informele, pen-based applicaties.

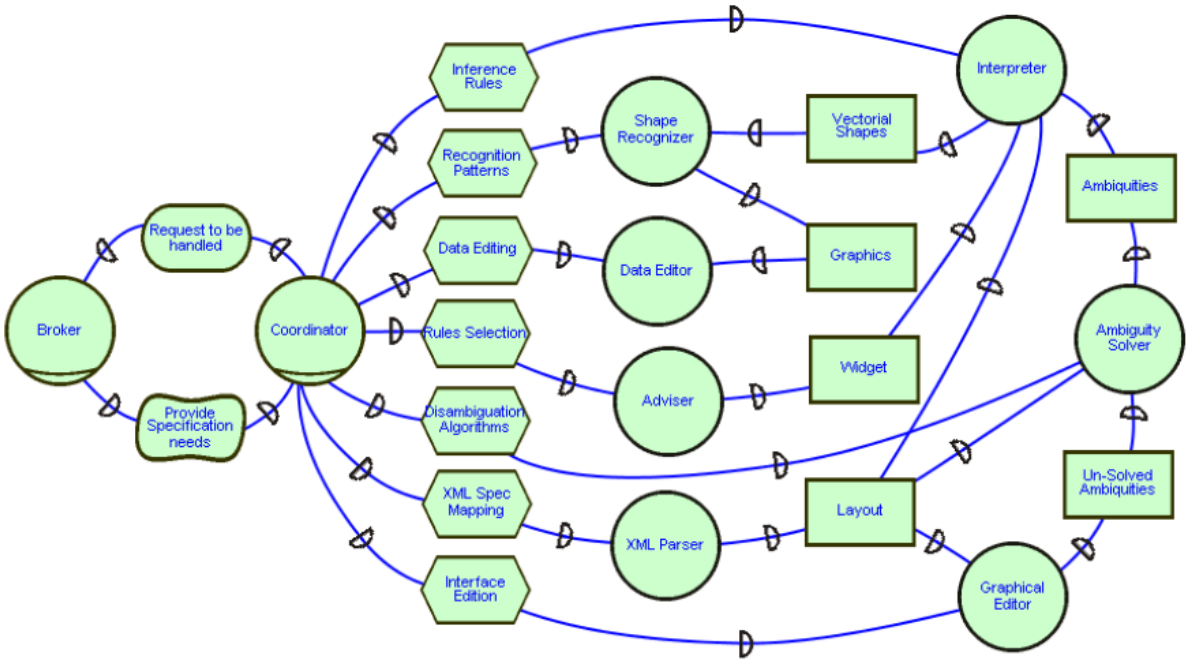
3.4 SketchiXML

In *SketchiXML* [Coye05] [Coye06] is het mogelijk een user interface te schetsen op verschillende detailniveaus. SketchiXML maakt daarvoor gebruik van UsiXML, een User Interface Description Language (UIDL) die het mogelijk maakt een complete user interface te definiëren onafhankelijk van onderliggende technologieën. Het gebruik van een UIDL heeft als doel tijdens de gehele ontwikkelingscyclus gebruik te maken van éénzelfde algemeen formaat. In tegenstelling tot andere UIDLs, zoals UIML [Ali02] en XIML [Eise01], maakt UsiXML gebruik van vier abstractieniveaus die gedefinieerd worden door het Cameleon reference framework [Calv03].

Het hoogste niveau in dit framework is *"Taken en Concepten"*. Op dit niveau worden de verschillende interactieve taken, die door de gebruiker uitgevoerd moeten worden, en de domeinobjecten, die door deze taken gemanipuleerd zullen worden, beschreven. Het tweede niveau *"Abstract User Interface"* geeft een definitie van de toekomstige user interface die onafhankelijk is van een bepaalde modaliteit of een bepaalde interactietechniek. Een AUI bestaat uit Abstract Containers (AC) en Abstract Individual Components (AICs) waartussen abstracte relaties gedefinieerd worden. AICs zijn abstracties van widgets die in GUIs kunnen worden terugvonden zoals bijvoorbeeld buttons. Er kunnen twee soorten relaties tussen AICs worden weergegeven: ten eerste een navigatie transitie en ten tweede een aantal fysieke constraints. Het derde niveau *"Concrete User Interface"* transformeert een AUI naar een bepaalde context door gebruik te maken van Concrete Interaction Objects (CIOs). Een CIO stelt een entiteit voor die door gebruikers kan waargenomen en gemanipuleerd worden zoals een push button, een check box, een vocal menu,…. Op het vierde en laatste niveau wordt de CIO omgezet naar een *"Final User interface"*. De FUI is de uiteindelijke operationele user interface.

Het grote voordeel van het gebruik van een UIDL is dat de output die gegenereerd wordt in SketchiXML gebruikt kan worden in andere softwarepakketten die eveneens gebruik maken van dit formaat zodat het werk uit de designfase niet verloren gaat.

De architectuur van SketchiXML is agent-oriented wat wil zeggen dat de verschillende taken worden uitgevoerd door agents. Een agent is een entiteit die een bepaalde verantwoordelijkheid van het design proces voor zijn rekening neemt. In figuur 8 wordt de architectuur van SketchiXML weergegeven. Deze is gestructureerd in de vorm van een graaf.



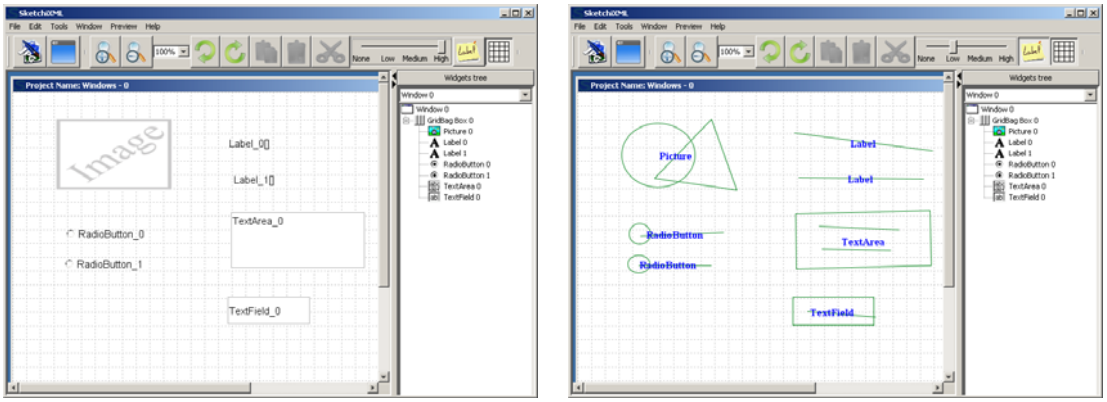
Figuur 8 - Agent oriented architectuur van SketchiXML, uit [Coye06]

De Broker agent fungeert als een tussenpersoon tussen de externe gebruiker en het interne systeem. Van zodra een nieuw project wordt opgestart, zal de Broker agent informatie van de gebruiker verzamelen en deze doorgeven aan de Coordinator agent. Hoe het sketching proces dan precies verloopt is afhankelijk van de parameters die de gebruiker hier opgeeft. Er kunnen parameters worden ingesteld voor: het gewenste platform (mobiele telefoon, PDA, TabletPC, laptop, desktop,…), de mate waarin recognition moet worden uitgevoerd (manueel of volledig automatisch), de mate waarin de UsiXML-specificatie moet gegenereerd worden (laag of hoog) en de kwaliteit van de output (laag of hoog). De Coordinator zal de ingegeven parameters verwerken en op basis daarvan de nodige agents activeren en coördineren.

In het volgende voorbeeld stellen we dat de gebruiker ervoor gekozen heeft real time recognition te laten uitvoeren en gebruik te maken van een pen als input device. De Data Editor agent zal vervolgens een leeg canvas tonen waarop de gebruiker de toekomstige interface kan gaan schetsen. De gebruiker kan door middel van een aantal voorgedefinieerde gestures user interface elementen op het canvas gaan toevoegen. De geschetste elementen komen automatisch in een container terecht die is uitgelijnd met een gridbaglayout.

Alle gestures en lijnen die de gebruiker tekent, worden verzameld en doorgegeven aan de Shape Recognizer agent. De recognition engine van deze agent zal de lijnen trachten om te zetten in vormen. Deze recognition engine is gebaseerd op de CALI

library. De herkende vormen worden vervolgens doorgegeven aan de Interpreter agent die de vormen zal trachten te combineren tot een layout. Hij maakt hiervoor gebruik van fuzzy spatial relations[1]. Wanneer de Interpreter er echter niet in slaagt alle componenten te herkennen, wordt de Ambiguity Solver agent geactiveerd. Deze agent zal trachten de ambiguïteiten op te lossen door middel van een aantal algoritmen toe te passen. Indien hij de ambiguïteiten niet krijgt opgelost, worden deze doorgegeven aan de Graphical Editor agent. Deze agent zal alle tot op die moment herkende widgets weergeven. De widgets met een te lage recognition rate worden gekenmerkt. Van zodra de layout een voldoende grote zekerheid heeft, zal de interface worden doorgestuurd naar de XML Parser agent die vervolgens een UsiXML specificatie van de interface genereert.



Figuur 9 - SketchiXML

SketchiXML beschikt over een gesture trainer en een grammar editor die de gebruiker de mogelijkheid geven om de voorgedefinieerde gestures te trainen, hun patroon te veranderen en nieuwe gestures toe te voegen.

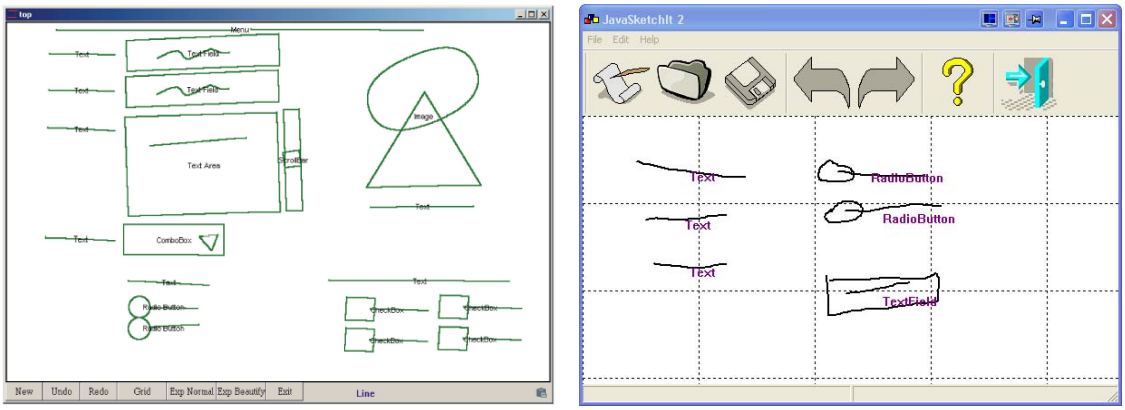
3.5 JavaSketchIt

JavaSketchIt [Caet02] is een prototype dat ontstaan is uit een onderzoek naar het gebruik van een visuele grammatica tijdens het schetsen van interfaces. Er werd onderzocht welke vormen en combinaties van vormen designers gebruiken tijdens het schetsen van interfaces. De resultaten werden geïmplementeerd via JavaSketchIt waar een designer een toekomstige interface kan schetsen door gebruik te maken van gestures en shapes. De shape recognizer die gebruikt wordt voor de herkenning is, net zoals die van SketchiXML, gebaseerd op de CALI library. Recognition vindt onmiddellijk plaats en zodra een vorm wordt herkend, wordt er geprobeerd deze samen te voegen met reeds herkende vormen tot samengestelde widgets. Er wordt gebruik gemaakt van fuzzy

[1] Fuzzy spatial relations worden gebruikt om ruimtelijke relaties te beschrijven die niet perfect gedefinieerd zijn

relational grammar om ambiguïteiten tussen vormen op te lossen. De geschetste interface kan uiteindelijk worden omgezet in een Java interface. Het uiteindelijke resultaat kan nog verbeterd worden door gebruik te maken van een aantal a posteriori grammar rules.

Momenteel is er ook een tweede versie van JavaSketchIt uitgebracht. Daarin wordt het voor gebruikers mogelijk een eigen visuele taal te creëren. Eveneens is het in JavaSketchIt 2 mogelijk om de geschetste user interface te exporteren naar een Java, HTML, GTK, C# of XML formaat. Geen van beide applicaties beschikken echter over de mogelijkheid om layout eigenschappen te specificeren.



Figuur 10 - Links JavaSketchIt[2] en rechts JavaSketchIt 2[3]

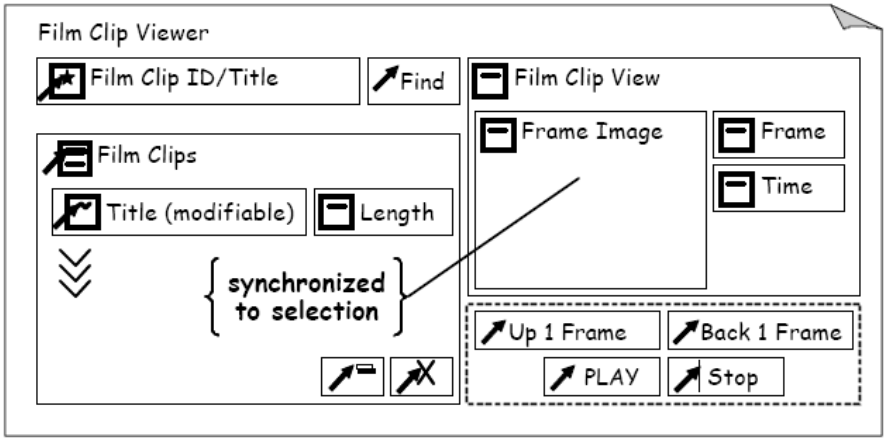
3.6 Canonical Abstract Prototypes

De sketch-based user interface tools die tot nu toe werden besproken, maken gebruik van een aantal specifieke user interface elementen om de toekomstige user interface voor te stellen. Een andere, meer abstracte manier die gebruikt kan worden om een user interface voor te stellen is via *Canonical Abstract Prototypes* [Cons03]. In een CAP is een prototype opgebouwd uit een verzameling universele, abstracte componenten. Elke component stelt een abstracte interactieve functie voor zoals bijvoorbeeld het weergeven van informatie, het veranderen van een status,…. Daarnaast kunnen de componenten ook gebruikt worden om de positie, grootte, layout en compositie van de user interface te definiëren.

Een component wordt voorgesteld door een notatie die is opgebouwd uit twee symbolen: een vierkant en een pijl. Het vierkant stelt een *materiaal* of een *container* voor en wordt gebruikt om inhoud, informatie, data of andere user interface objecten die gemanipuleerd kunnen worden tijdens het uitvoeren van de taak, te modelleren. De pijl stelt een *tool* of een *actie* voor en wordt gebruikt om operatoren, mechanismen of

[2] JavaSketchIt project: http://immi.inesc-id.pt/project_page.php?project_id=21
[3] JavaSketchIt 2 project: http://immi.inesc-id.pt/projects/javasketchit2/index.html

controllers te presenteren die gebruikt kunnen worden om materialen te creëren, te transformeren of te manipuleren. Deze twee klassen van componenten kunnen gecombineerd worden tot een derde hybride klasse die bestaat uit componenten die gelijktijdig een container en een actieve control voorstellen. Een voorbeeld van deze laatste klasse is een text entry box, waar enerzijds informatie kan worden weergegeven en anderzijds deze informatie gemanipuleerd kan worden of nieuwe informatie kan worden toegevoegd [Cons03] [Camp04].



Figuur 11 - Canonical Abstract Prototype, uit [Cons03]

In de volgende alinea's worden twee tools beschreven die een user interface voorstellen door middel van abstracte componenten. Een eerste tool die wordt beschreven is *CanonSketch* [Camp04]. CanonSketch biedt ondersteuning voor het tekenen, groeperen, labelen en resizen van abstracte componenten. De componenten bevinden zich op een palet en kunnen van daaruit op het canvas worden toegevoegd. De tool ondersteunt eveneens een aantal commando's waarvan undo en redo, een gridlayout, tool tips en het naar voor- en achterbrengen van componenten.

In CanonSketch kan er gebruik worden gemaakt van drie soorten views: de Wisdom view waar het mogelijk is een Wisdom presentation model [Nune01] te creëren, een Canonical view waar het palet met abstracte componenten wordt weergegeven en de HTML view waar een concreet prototype in HTML kan worden gegenereerd.

Een tweede tool die de creatie van CAPs ondersteunt is *DialogSketch* [Nóbr05]. In DialogSketch worden Canonical Abstract Prototypes gecombineerd met de ConcurTaskTree notatie [Mori02]. Het doel van deze combinatie is een link te creëren tussen het dialoogmodel, voorgesteld door middel van ConcurTaskTrees en het presentatiemodel dat wordt voorgesteld via Canonical Abstract Prototypes. Beide notaties worden complementair gebruikt voor het creëren van effectieve abstracte prototypes. DialogSketch ondersteunt de creatie en modificatie van de UML versie van deze CAPs en CTTs.

Net zoals CanonSketch beschikt DialogSketch over de mogelijkheid een gridlayout te specificeren. Daarnaast worden nog een aantal andere features geïmplementeerd zoals een multiple undo en een mogelijkheid tot zoomen. In DialogSketch bestaat er echter geen mogelijkheid om code te genereren van het gecreëerde prototype.

3.7 Vergelijking sketch-based applicaties

		Mogelijkheden?	Layout ondersteuning?	Code generatie?	Gesture recognition algoritme?
	SILK	Creatie van UI; storyboard mechanisme; annotaties	Groeperen van componenten in widgets op basis van ruimtelijke relaties: check box bestaat uit vierkant met rechts tekst	Visual Basic 5, Common Lisp	Rubine
	DENIM	Creatie van website; storyboard mechanisme; annotaties; zooming	Groeperen van radio buttons en tekst	HTML	Rubine
	SketchiXML	Creatie van UI; ondersteuning UsiXML	Groeperen van componenten in widgets; elementen worden automatisch in een gridbaglayout container geplaatst	Nee	CALI library
	JavaSketchIt	Creatie van UI	Groeperen van componenten in widgets	Java	CALI library
	JavaSketchIt 2	Creatie van UI; creatie van eigen visuele taal	Groeperen van componenten in widgets	Java, HTML, XML, C#, GTK	CALI library
CAP	**CanonSketch**	Creatie van UI; ondersteuning CAPs en Wisdom model	Plaatsen van elementen in containers; toepassen gridlayout	HTML	-
	DialogSketch	Creatie van UI; ondersteuning CAPs en CTTs	Plaatsen van elementen in containers; toepassen gridlayout	Nee	-

Tabel 2 - Vergelijkend overzicht van sketch-based applicaties

3.8 Conclusie

In dit hoofdstuk werd een overzicht gegeven van de verschillende sketch-based applicaties die zich momenteel op de markt bevinden.

SILK en DENIM werden als eerste besproken. Beide tools leggen vooral de aandacht op het specificeren van het gedrag van de toekomstige user interface. Ze zijn daarvoor uitgerust met een storyboard mechanisme waar de gebruiker door middel van pijlen en lijnen de transities tussen de verschillende schermen kan gaan weergeven. Voor de herkenning van shapes en gestures maken beide tools gebruik van een aangepaste versie van Rubine's algoritme. SILK is in staat een aantal verschillende user interface elementen te herkennen, maar voert deze herkenning niet onmiddellijk uit. DENIM is iets beperkter op dat gebied en herkent minder widgets dan SILK. DENIM is dan ook specifiek ontworpen voor de ontwikkeling van websites.

De volgende twee tools die werden besproken, waren SketchiXML en JavaSketchIt. Beiden hebben ze de mogelijkheid een groot aantal user interface elementen te herkennen. De herkenning gebeurt ook vlot, met een recogntion rate tot zelfs 97 procent[4]. Hun recognition engine is gebaseerd op de CALI library. Net zoals DENIM en SILK, is het in SketchiXML en JavaSketchIt mogelijk de user interface op een informele manier te gaan schetsen door gebruik te maken van scribbles en gestures. Deze vorm van interactie is zeer intuïtief voor de gebruiker en daarom zal deze interactievorm ook in het te ontwikkelen prototype geïntegreerd worden. De geschetste user interface elementen moeten kunnen worden toegevoegd door middel van pen-based interactietechnieken. Voor de herkenning van de user interface elementen zal er gekeken worden naar de werking van het recognition algoritme dat in SILK en DENIM wordt gebruikt en eveneens naar de library die in SketchiXML en JavaSketchIt wordt gebruikt. Deze algoritmen worden besproken in hoofdstuk 6. De volgende twee doelstellingen worden aan de lijst met doelstellingen toegevoegd:

"Doelstelling 2: Het prototype moet pen-based interactie technieken ondersteunen."

"Doelstelling 3: Het prototype moet in staat zijn geschetste user interface elementen te herkennen."

Als laatste in dit hoofdstuk werden Canonical Abstract Prototypes besproken. Via CAPs wordt het mogelijk de toekomstige user interface op een abstracte manier voor te stellen. Twee tools die het gebruik van CAPs ondersteunen werden besproken, namelijk CanonSketch en DialogSketch.

[4] CALI project page: http://immi.inesc-id.pt/cali/index.html

Na het bekijken van de verschillende tools kan er besloten worden dat deze weinig mogelijkheden bieden op gebied van specificatie van layout eigenschappen. CanonSketch en DialogSketch bieden op dit vlak nog het grootste aantal mogelijkheden, namelijk het specificeren van een gridlayout en het onderbrengen van elementen in een container. Zoals reeds in het begin van deze thesis werd aangegeven, zou de mogelijkheid om layout eigenschappen te definiëren en layout commando's uit te voeren toch een meerwaarde kunnen betekenen voor bestaande sketch-based user interfaces. Daarom zal in het volgende hoofdstuk worden gekeken naar welke mogelijkheden er zijn om layout eigenschappen van user interface widgets te definiëren. De volgende doelstelling kan ook nog worden toegevoegd aan de doelstellingenlijst:

> "Doelstelling 4: Het prototype moet de gebruiker in staat stellen elementen te laten groeperen in containers."

Hoofdstuk 4

Layout Management

4.1 Overview

Zoals reeds in hoofdstuk 2 besproken werd, maken designers in het begin van het ontwikkelingsproces gebruik van low-fidelity technieken zoals pen en papier om de algemene structuur van de toekomstige user interface te schetsen en om snel verschillende ideeën uit te proberen. In dit beginstadium is de exacte positie en grootte van de user interface elementen nog van geen enkel belang. Later in het ontwikkelingsproces zal de interface echter meer vorm en structuur krijgen. Op een bepaald ogenblik schakelen designers dan ook over naar high-fidelity technieken om de interface verder te gaan ontwerpen. Toolkits en user interface builders zijn voorbeelden van zulke Hi-Fi technieken. Meestal beschikken toolkits over een grafische versie zodat de ontwikkeling van de user interface niet louter bestaat uit het schrijven van code. Deze grafische versie bestaat meestal uit een canvas met toolbox, waarbij de gebruiker elementen vanuit de toolbox naar het canvas kan slepen. Het grote voordeel van deze aanpak is dat de positie en grootte van elk element onmiddellijk gekend is.

In dit hoofdstuk worden de belangrijkste user interface toolkits besproken. Er wordt hierbij vooral aandacht besteed aan de manier waarop deze toolkits omgaan met het specificeren van layout eigenschappen zoals grootte en positie. Alvorens een overzicht te geven van de layout mogelijkheden binnen de verschillende toolkits wordt eerst kort het layout proces omschreven.

Daarnaast wordt er in dit hoofdstuk ook een inleiding gegeven tot constraint-based layout systemen. In deze systemen wordt een layout voorgesteld als een verzameling constraints. Een aantal van deze systemen zijn ook in staat een layout automatisch te

genereren. Ze maken daarvoor gebruik van automated layout technieken waarbij de grootte en positie van de objecten in de representatie voor een gedeelte of helemaal automatisch door het systeem gebeurd.

4.2 Layout proces

Het layout proces is een proces waarin de grootte en positie van elk visueel object in een informatie presentatie wordt bepaald. In deze definitie wordt met “presentatie” verwezen naar het materiaal dat door de gebruikers ervan kan worden bekeken en kan worden gemanipuleerd. Voorbeelden van materialen zijn grafische of tekstuele user interfaces, World Wide Web documenten,…. Het opbouwen van de layout maakt deel uit van een groter design proces waarin eveneens beslissingen worden gemaakt omtrent het aantal visuele objecten, de informatie die ze overbrengen en het formaat van deze informatie. Onder formaat wordt de manier verstaan waarop visuele objecten worden gerealiseerd, zoals tekst en user interface elementen, en eveneens de attributen van deze realisaties zoals kleur, lettertype en textuur.

De meeste user interfaces worden vandaag de dag ontworpen met behulp van een user interface toolkit. Toolkits bieden een aantal voorgedefinieerde grafische elementen, zoals push buttons en combo boxen, waarmee een user interface kan worden opgebouwd. De positie en grootte van deze elementen kan op twee manieren worden bepaald: via *absolute positionering* of via *layout managers* [Lok01].

Bij *absolute positionering* wordt de grootte en positie van elk element in de code gedefinieerd. Het gebruik van absolute positionering heeft daardoor een groot aantal nadelen. De positie en grootte moet van elk element handmatig worden uitgerekend. Hierdoor wordt de constructie van een layout niet alleen een langdurig maar ook een foutgevoelig proces. Bovendien is dit ook erg nadelig voor de onderhoudbaarheid. Een tweede nadeel is dat de user interface elementen niet over de mogelijkheid beschikken zich aan te passen aan veranderingen in het venster of container waarin ze zijn ondergebracht. Wanneer bijvoorbeeld het gebruikte lettertype of de algemene stijl van het hoofdvenster wordt veranderd, passen de elementen zich niet automatisch aan waardoor bepaalde informatie verborgen kan worden [Blan06].

Toolkits zijn daarom uitgerust met *layout managers*. Een layout manager is een component dat de grootte en positie van de grafische elementen at run time gaat bepalen. Deze waarden worden bepaald door twee factoren: enerzijds door de beperkingen die door de gekozen layout manager worden opgelegd en anderzijds door een aantal parameters die door de gebruiker gedefinieerd kunnen worden [Lok01].

Een layout manager zal de verschillende componenten onderbrengen in een container. De manier waarop deze componenten in de container worden gerangschikt, wordt enerzijds dus bepaald door het type layout manager. Zo zal een horizontale layout manager de componenten op een horizontale lijn plaatsen, terwijl een gridlayout manager de componenten zal verdelen over een rooster. Door het plaatsen van de componenten in een container lost een layout manager een vorm van het *two dimensional bin packing problem* op. In het two dimensional bin packing problem wordt een gegeven verzameling van n rechthoekige items met elk een breedte w_j en hoogte h_j verdeeld over een onbeperkt aantal identieke, begrensde, rechthoekige bins, waarbij elke bin een breedte W en hoogte H heeft. De items moeten in zo weinig mogelijk bins worden ondergebracht. Hierbij moet er rekening gehouden worden dat de items elkaar niet overlappen en de randen van de items parallel worden uitgelijnd met de randen van de bin [Lodi99].

Bij het plaatsen van de componenten in de container wordt er anderzijds ook rekening gehouden met de *parameters* die door de gebruiker gedefinieerd worden. Voorbeelden van zulke parameters zijn minimum, maximum en gewenste hoogte en breedte, de ruimte tussen objecten onderling en de ruimte tussen objecten en hun containers.

De layout wordt uiteindelijk at run time samengesteld. Dit heeft als grote voordeel dat de layout zich kan aanpassen aan de grootte van het scherm [Lok01]. In de volgende alinea's wordt een overzicht gegeven van de belangrijkste grafische toolkits en hun bijhorende layout managers.

4.3 Layout management in Qt

Qt is een grafisch framework dat kan gebruikt worden bij de ontwikkeling van grafische C++ applicaties. Het framework is opgebouwd uit vier grote onderdelen: Qt Class Library, Qt Assistant, Qt Linguist en tenslotte Qt Designer. De core van het framework wordt gevormd door de Class Library. Deze bestaat uit een vierhonderdtal klassen die naast grafische functionaliteit ook ondersteuning bieden voor threading, database toegang, XML, OpenGL integratie, en dergelijke. Het tweede belangrijkste onderdeel van het framework is de Qt Designer. Qt Designer is een user interface builder, die het toelaat de look-and-feel van de toekomstige user interface op een grafische manier te construeren. De Designer beschikt hiervoor over een aantal voorgedefinieerde componenten, widgets genaamd, die gebruikt kunnen worden om de user interface op te bouwen. De ontwikkelde user interface kan op eender welk ogenblik worden omgezet in code. Tenslotte bestaat het framework ook nog uit Qt Linguist en Qt Assistant. Qt Linguist is een interface die ondersteuning biedt bij de vertaling van user interface tekst

naar andere talen. Qt Assistant is een document reader die gebruikt kan worden bij het samenstellen van online help documentatie [Qt1].

In Qt zijn er drie verschillende manieren om layout eigenschappen toe te voegen aan widgets en forms waaronder *absolute positionering*, *manuele layout* en *layout managers*.

4.3.1 Absolute positionering

De eerste manier om layout eigenschappen toe te voegen aan widgets is via *absolute positionering*. De grootte en positie van elk widget wordt via de `setGeometry()` functie toegekend [Blan06].

4.3.2 Manuele layout

Een tweede methode om een layout toe te voegen is via *manuele layout*. Bij deze methode worden er nog steeds absolute posities aan de widgets gegeven, maar de groottes worden proportioneel ingesteld ten opzichte van het venster waarin de widgets zich bevinden. In Qt wordt manuele layout toegevoegd door middel van herimplementatie van de `resizeEvent()` functie. Omdat de posities van de widgets nog steeds in de code worden gedefinieerd, heeft het gebruik van manuele layout nog steeds dezelfde nadelen als absolute positionering [Blan06].

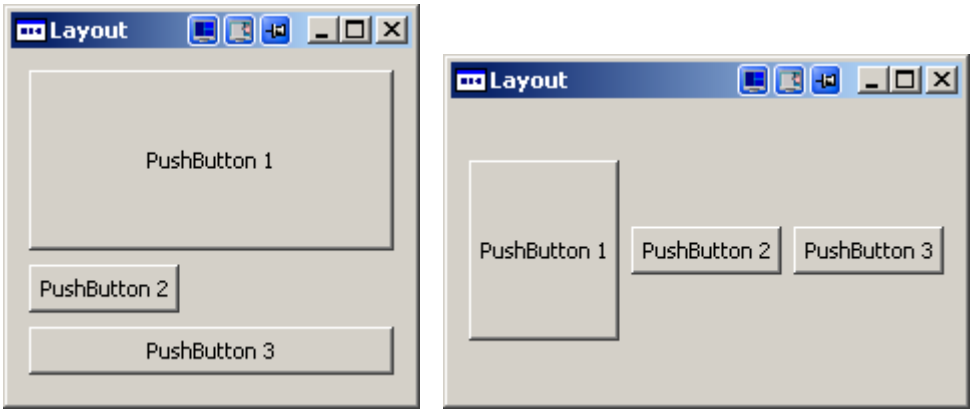
4.3.3 Layout managers

Tenslotte kan er gebruik worden gemaakt van *layout managers* om de layout van widgets en forms te specificeren. Qt verziet een aantal voorgedefinieerde layout managers. Deze worden in de volgende alinea's besproken.

QBoxLayout

De QBoxLayout heeft twee mogelijke uitvoeringen: QVBoxLayout en QHBoxLayout. QVBoxLayout zal de verschillende widgets op een verticale lijn naast elkaar plaatsen, QHBoxLayout op een horizontale lijn onder elkaar. Er kunnen nieuwe widgets aan het einde van de layout worden toegevoegd met behulp van de functie `addWidget`. Hierbij neemt deze functie drie argumenten. Het eerste argument specificeert het widget dat aan de layout moet worden toegevoegd. Het tweede argument kan worden

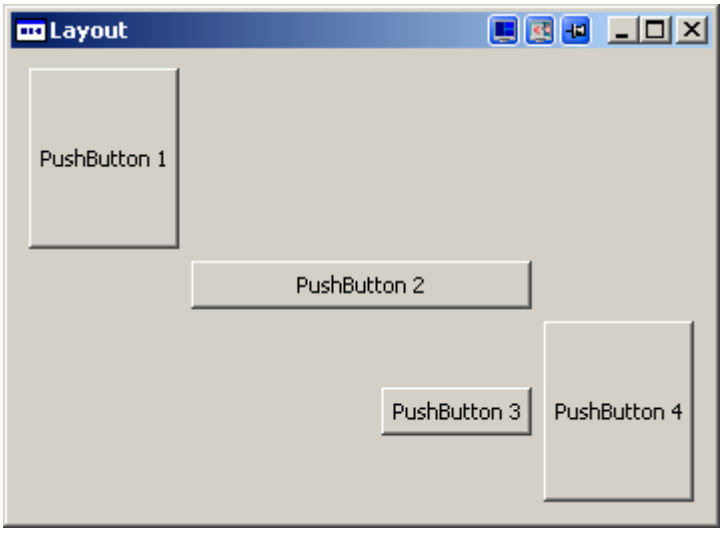
gebruikt om een stretch factor op te geven. Deze stretch factor bepaalt hoe snel het widget “groeit” bij een toename van de beschikbare plaats. Hoe groter de opgegeven stretch factor, hoe sneller het widget zal groeien. Via het laatste argument kan een individuele uitlijning per widget worden opgegeven. Bij het plaatsen van widgets in de layout wordt er eveneens rekening gehouden met de opgegeven minimum, maximum en gewenste grootte van elk widget [Blan06][Qt2].



Figuur 12 - Links QVBoxLayout, rechts QHBoxLayout

QGridLayout

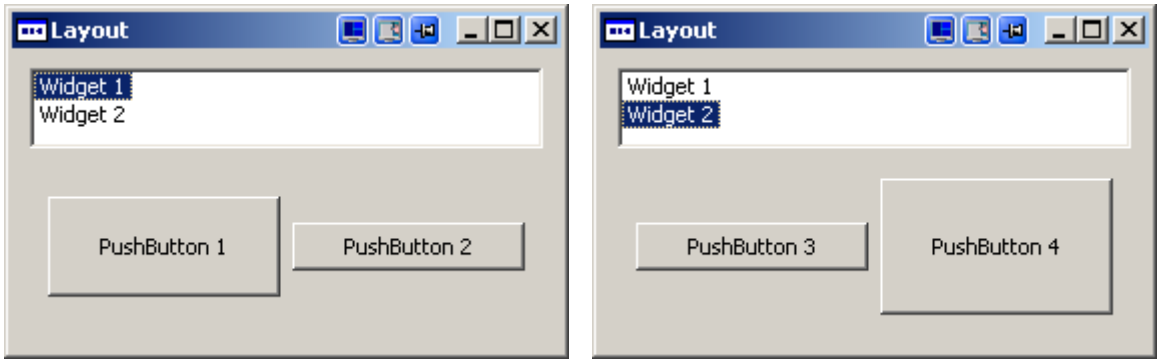
Een QGridLayout verdeelt de beschikbare ruimte op in rijen en kolommen en zal vervolgens de verschillende widgets onderbrengen in het rooster. Normaal gezien wordt er één widget per cel ondergebracht, maar het is ook mogelijk om een widget over meerdere cellen te spreiden [Blan06] [Qt2].



Figuur 13 - QGridLayout

QStackedLayout

De QStackedLayout rangschikt de verschillende widgets als een stapel kaarten. Enkel het bovenste widget is zichtbaar voor de gebruiker, de rest van de widgets worden verborgen. Om toegang te krijgen tot de overige widgets, dient een mechanisme te worden toegevoegd om tussen de widgets te wisselen. Hiervoor kan gebruik worden gemaakt van een QComboBox of een QListWidget [Blan06] [Qt2].

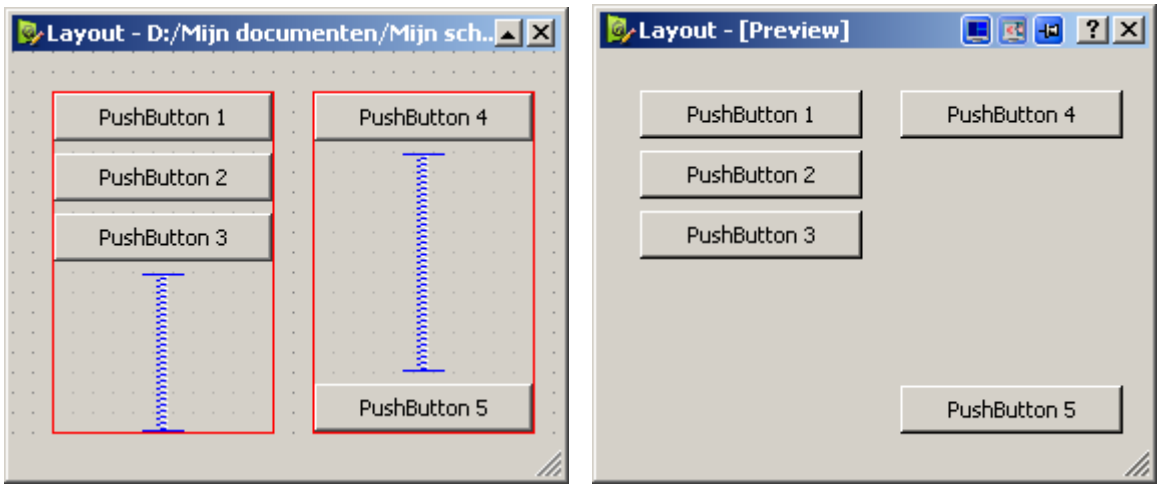


Figuur 14 - QstackedLayout met een QListWidget om tussen de widgets te switchen

4.3.4 Overige layout elementen

QSpacer

Een *spacer item* (ook wel “stretch” genaamd) zorgt ervoor dat een bepaalde ruimte wordt opgevuld. Een spacer item wordt voorgesteld in de vorm van een blauwe veer en kan horizontaal of verticaal georiënteerd zijn. De QSpacer wordt enkel getoond in de Qt Designer. In de uiteindelijke user interface is deze onzichtbaar.



Figuur 15 - QSpacer, links in Qt Designer, rechts in uiteindelijke interface

4.4 Layout Management in JFC

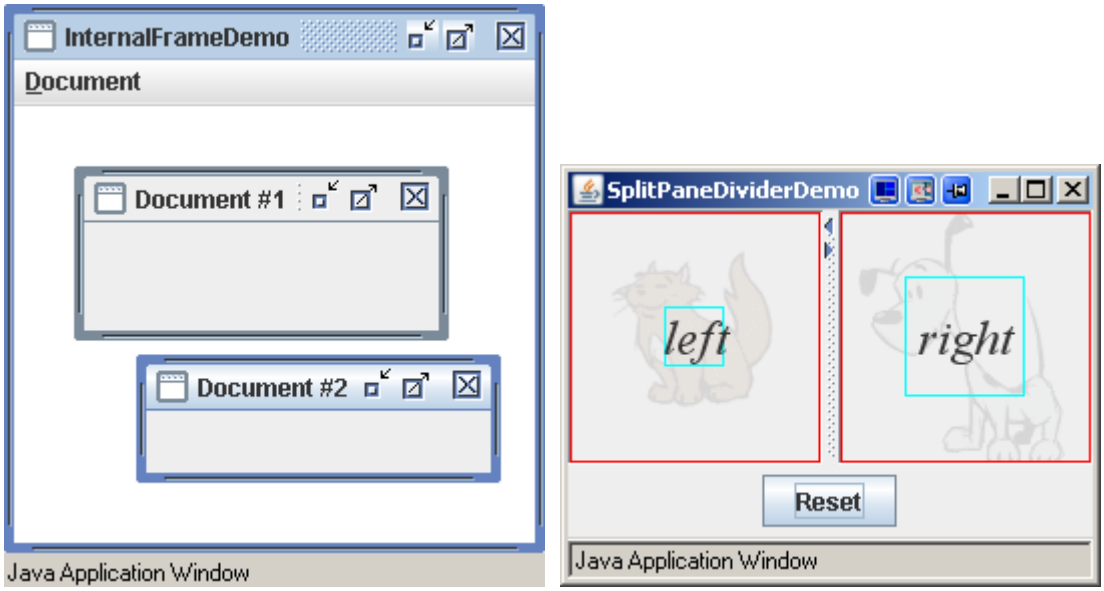
JFC is de afkorting voor Java Foundation Classes, een verzameling van klassen die gebruikt kunnen worden bij de creatie van grafische user interfaces in Java. De Java Foundation Classes bestaan uit een aantal API's en toolkits die grafische functionaliteit en interactiviteit aan Java applicaties kunnen toevoegen. De originele grafische toolkit is AWT, wat staat voor Abstract Windows Toolkit. AWT is een simpele toolkit die een beperkt aantal grafische elementen (componenten genaamd), events en layout managers definieert. Het biedt bijvoorbeeld geen ondersteuning voor de creatie van complexere componenten zoals bijvoorbeeld tabellen, trees,…. Daarom werd in een later stadium de Swing toolkit aan de JFC toegevoegd. Swing is gebaseerd op AWT, maar bevat een uitgebreider aantal componenten en layout managers. Via Swing wordt het mogelijk complexere user interfaces te creëren. Hier tegenover staat wel dat de toolkit complexer is, en dus ook moeilijker te leren. Naast AWT en Swing voorzien de JFC nog een aantal andere functionaliteiten zoals een 2D API voor het creëren en onderhouden van tweedimensionale tekeningen, een 3D API voor driedimensionaal tekenen en een Accessibility API voor het toevoegen van ondersteunende technologieën zoals Braille displays en drag-and-drop support [Knud99] [Java1].

4.4.1 Absolute positionering

In JFC kan eveneens gebruik worden gemaakt van absolute positionering of layout managers om layout eigenschappen van componenten te definiëren [Swin]. Zoals reeds besproken heeft het gebruik van absolute positionering een groot aantal nadelen. Slechts in een aantal gevallen wordt het gebruik ervan in JFC toch niet afgeraden. Wanneer in een container componenten zijn opgenomen die toch niet worden beïnvloed door de container kan er gebruik worden gemaakt van absolute positionering. Voorbeelden van zulke componenten zijn desktop panes. Een desktop pane bevat interne frames die niet direct beïnvloed worden door de grootte van het pane.

Ook bij custom containers, die zelf de berekening van grootte en positie uitvoert zoals bijvoorbeeld bij split panes, kan gebruik worden gemaakt van absolute positionering. Een voorbeeld van een desktop pane en split panes wordt gegeven in figuur 16.

In alle andere gevallen wordt echter het gebruik van layout managers aangeraden. In de volgende alinea's zullen kort de belangrijkste layout managers en hun werking worden besproken [Swin].

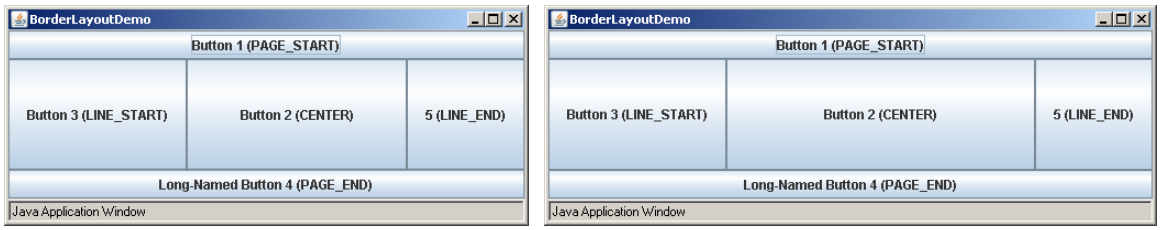


Figuur 16 - Links desktop pane met interne frames, rechts split pane, uit [5]

4.4.2 Layout managers

BorderLayout

Een BorderLayout [Swin] zal de grafische componenten in vijf verschillende gebieden verdelen namelijk boven (PAGE_START), onder (PAGE_END), links (LINE_START), rechts (LINE_END) en midden (CENTER). Het middelste gedeelte neemt zoveel mogelijk van de beschikbare schermruimte in. Wanneer het scherm wordt vergroot, zal vooral het middelste gebied vergroten. De overgebleven gebieden vullen de rest van het scherm op.



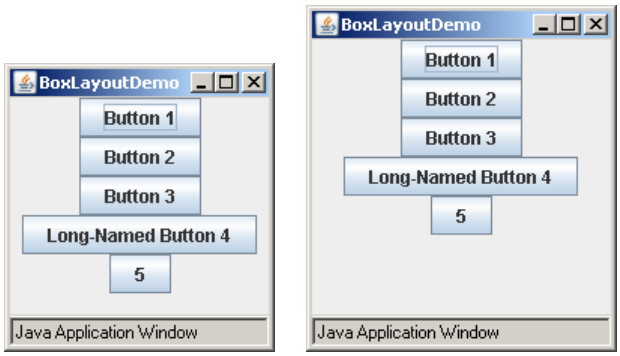
Figuur 17 - BorderLayout, uit [5]

De BorderLayout is zowel in AWT als in Swing beschikbaar [Java2] en wordt best gekozen wanneer een component zoveel mogelijk plaats moet krijgen. Een voorbeeld waar best gebruik wordt gemaakt van een BorderLayout is een scherm dat is opgebouwd met in het midden een tekencanvas en daar rond enkele toolboxen [Swin].

[5] Java Tutorials: Laying Out Components Within a Container: Examples: http://java.sun.com/docs/books/tutorial/uiswing/examples/layout/index.html

BoxLayout

Een BoxLayout [Swin] plaatst alle componenten op éénzelfde rij of in éénzelfde kolom. Daarbij wordt er rekening gehouden met de afzonderlijke uitlijning en grootte van elke component. In figuur 18 wordt een voorbeeld gegeven van een top-to-bottom BoxLayout. Wanneer er niet voldoende ruimte beschikbaar is om de verschillende componenten te plaatsen, zal de grootte van de componenten worden aangepast. Indien er voldoende plaats aanwezig is, wordt de overige ruimte onderaan het scherm toegevoegd.



Figuur 18 - BoxLayout, uit [5]

De BoxLayout is gelijkaardig aan Qt's QBoxLayout en is enkel beschikbaar in de Swing toolkit [Java2]. Deze layout wordt het best gebruikt indien er een klein aantal componenten met verschillende uitlijning en grootte in een rij of kolom moeten worden geplaatst [Swin].

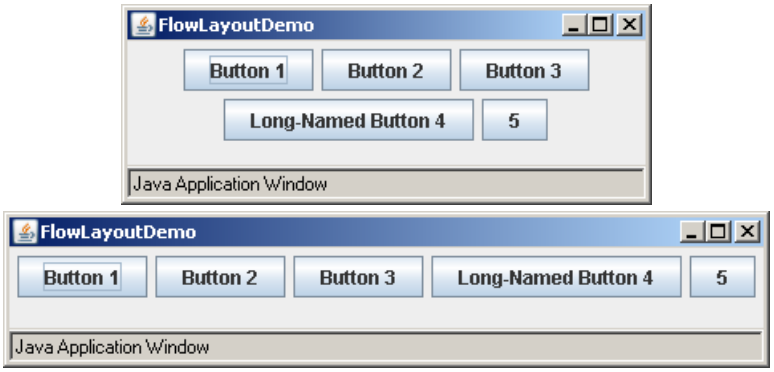
CardLayout

De CardLayout [Swin] werkt op dezelfde manier als Qt's QStackedLayout. De verschillende componenten worden ook hier op elkaar geplaatst zoals in een stapel kaarten. De componenten delen éénzelfde schermruimte waarbij enkel de bovenste component zichtbaar is voor de gebruiker. Omdat de onderliggende componenten niet zichtbaar zijn, moet ook hier een mechanisme worden toegevoegd waarmee de gebruiker toegang krijgt tot de onderliggende componenten. Dit kan via een combo box of tabblad. De CardLayout is in AWT en Swing beschikbaar [Java2].

Figuur 19 - CardLayout, uit [5]

FlowLayout

De FlowLayout [Swin] brengt de verschillende componenten onder in een rij. Wanneer niet alle componenten in eenzelfde rij passen, worden ze verdeeld over verschillende rijen. Wanneer er plaats over is, zal de rij gecentreerd worden. Indien gewenst kan er dan gekozen worden voor een rechtse of linkse uitlijning. De componenten kunnen in grootte van elkaar verschillen.

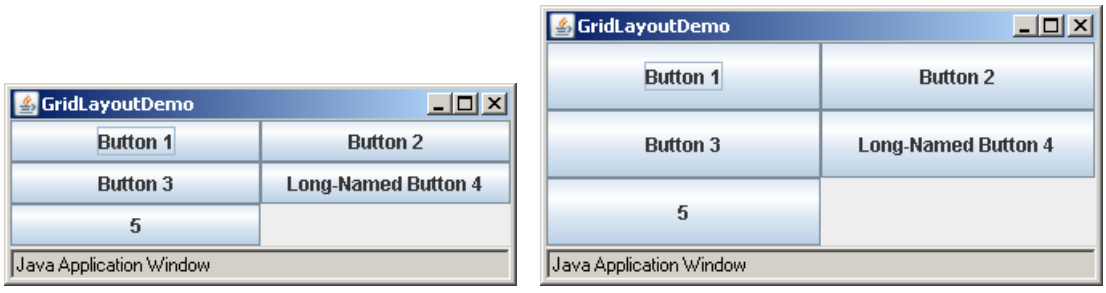


Figuur 20 - FlowLayout, uit [5]

De FlowLayout is geschikt voor het weergeven van een klein aantal componenten [Swin] en kan zowel in Swing als in AWT worden gebruikt [Java2].

GridLayout

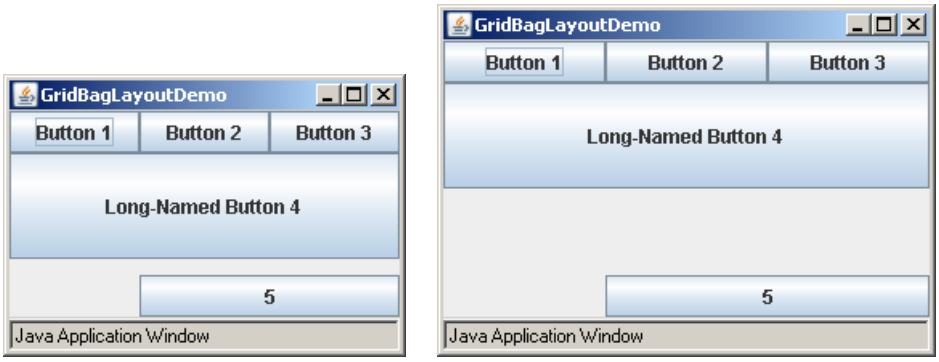
Wanneer gebruik wordt gemaakt van een GridLayout, wordt het scherm onderverdeeld in een rooster. Elke cel in het rooster heeft dezelfde grootte en bevat één component. Wanneer het scherm wordt vergroot, zal ook de grootte van de cellen toenemen [Swin]. De GridLayout is zowel in AWT als in Swing beschikbaar [Java2].



Figuur 21 - GridLayout, uit [5]

GridBagLayout

Net zoals bij een GridLayout worden de verschillende componenten onder gebracht in een rooster. Er zijn echter twee grote verschillen met de GridLayout. Ten eerste is het in een GridBagLayout mogelijk componenten over meerdere kolommen en rijen te verdelen. Ten tweede wordt er ook rekening gehouden met de gewenste grootte van ieder individueel component, met als gevolg dat niet iedere cel in het rooster even groot is [Swin]. De GridBagLayout heeft dezelfde werking als Qt's QGridLayout en is zowel in AWT als in Swing beschikbaar [Java2].



Figuur 22 - GridBagLayout, uit [5]

4.4.3 Overige layout elementen

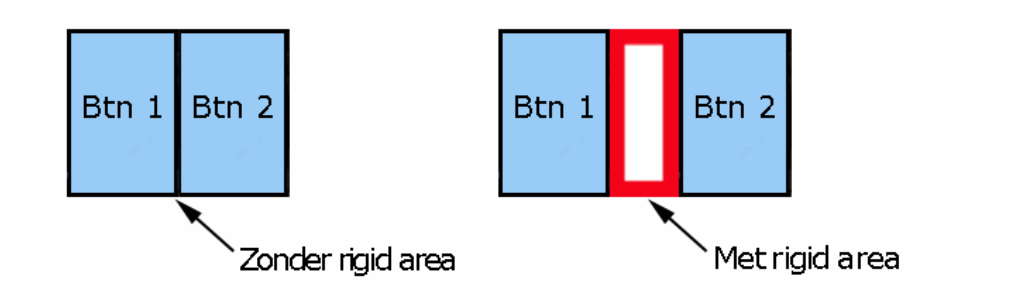
In JFC kan er gebruik worden gemaakt van de Box klasse om in een layout componenten te introduceren die ruimte opvullen. Deze klasse heeft drie verschillende componenten: rigid area, glue en custom filler. Deze componenten zijn vergelijkbaar met de spacer widget in Qt. Ze zijn onzichtbaar in de uiteindelijke user interface.

Rigid area

Wanneer een vaste ruimte moet worden opgevuld, kan er gebruik worden gemaakt van een *rigid area*. Onderstaand code voorbeeld geeft aan hoe een rigid area kan worden gecreëerd. De exacte grootte van het gebied kan worden opgegeven.

```
container.add(button1);
container.add(Box.createRigidArea(new Dimension(5,0)));
container.add(button2);
```

Dit voorbeeld zal een rigid area van 5 pixels breed tussen twee buttons plaatsen.

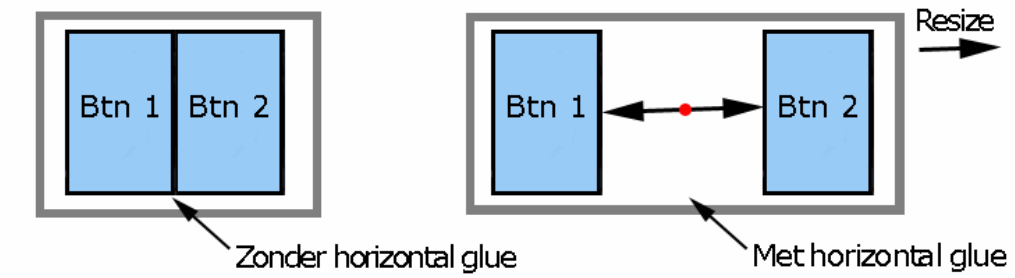


Figuur 23 - Rigid area

Glue

Via *glue* kan worden aangegeven dat beschikbare ruimte tussen componenten moet worden opgevuld in plaats van rechts ervan. Aan glue kan een horizontale of verticale oriëntatie worden gegeven.

```
container.add(button1);
container.add(Box.createHorizontalGlue());
container.add(button2);
```

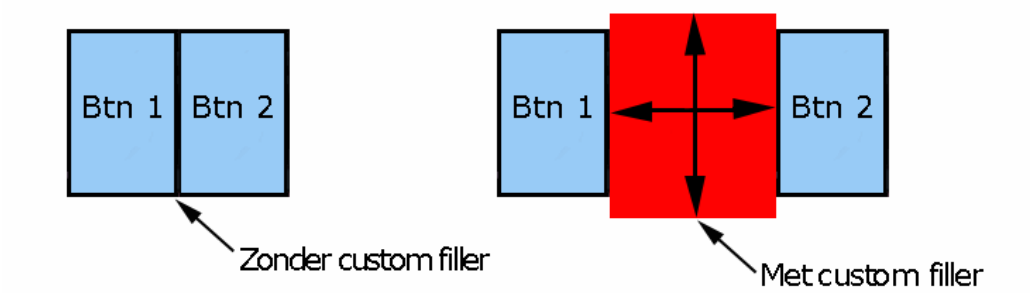


Figuur 24 - Glue

Custom filler

Tenslotte is er nog de *custom filler*. Aan dit component kan een minimum, maximum en preferred dimensie worden gegeven.

```
container.add(button1);
Dimension minSize = new Dimension(5, 100);
Dimension prefSize = new Dimension(5, 100);
Dimension maxSize = new Dimension(10, 200);
container.add(new Box.Filler(minSize, prefSize, maxSize));
container.add(button2);
```



Figuur 25 - Custom filler

4.5 Layout Management in GTK+

GTK+ is een multi-platform library die gebruikt kan worden voor de creatie van grafische user interfaces. GTK+ is gebaseerd op drie libraries: GLib, Pango en ATK. GLib is de core library en vormt de basis voor GTK+. Pango is de library voor het renderen van tekst en lettertypes en tenslotte is er nog de ATK library die een aantal accessibility interfaces voorziet zoals screen readers en magnifiers [GTK1].

4.5.1 Packing widgets

Packing boxes

Een *packing box* is vergelijkbaar met de horizontale en verticale layout managers uit Qt en JFC. De box is een onzichtbare container waarin user interface widgets horizontaal of verticaal gerangschikt kunnen worden. Bij het aanmaken van de packing box kan door middel van een aantal parameters het gedrag van de componenten in de container worden bepaald. Via de parameter “expand” kan worden aangegeven of de widgets die in de box zijn ondergebracht alle beschikbare plaats innemen of dat de grootte van de box kleiner wordt gemaakt zodat alle widgets er in passen. Indien voor de laatste mogelijkheid wordt gekozen is het mogelijk een linkse of rechtse uitlijning van de widgets te kiezen. De parameter “fill” geeft aan of eventueel beschikbare ruimte wordt verdeeld over de widgets zelf of over de padding in de box [GTK2].

Packing met tabellen

Widgets kunnen ook ondergebracht worden in *tabellen*. Deze manier van packing vertoont veel gelijkenis met de gridlayout in Qt en de gridbaglayout in JFC. De widgets kunnen onderverdeeld worden in een aantal rijen en kolommen. Bij het creëren van de tabel kan worden aangegeven of de cellen zich automatisch dienen aan te passen aan het grootste widget dat zich in de tabel bevindt, of dat de grootte van de cellen bepaald wordt door het grootste widget in diezelfde rij en kolom. Net zoals in Qt’s gridlayout en JFC’s

gridbaglayout kan er per widget worden aangegeven hoeveel cellen deze dient in te nemen en hoe het widget in de cel moet worden uitgelijnd. Eveneens kan via de parameters “xoptions” en “yoptions” worden opgegeven of het widget de hele cel moet innemen, of widgets verkleind mogen worden indien blijkt dat de tabel te klein is en of de tabel overgebleven ruimte mag innemen of niet [GTK2].

4.5.2 Containers

In GTK+ kan ook gebruik worden gemaakt van containers om een widget binnen een window te plaatsen. Er zijn drie verschillende soorten containers: *alignment widgets*, *fixed containers* en *layout containers*.

Een *alignment widget* geeft de mogelijkheid om een widget binnen een window te plaatsen met een grootte en positie relatief ten opzichte van de grootte van het alignment widget. Alignment widgets kunnen gebruikt worden om widgets te centreren binnen een window. Via een *fixed container* wordt het mogelijk een widget op een vaste positie in een window te plaatsen. Een *layout container* heeft dezelfde mogelijkheden als een fixed container maar een layout container is uitgerust met een oneindig scrolling area [GTK2].

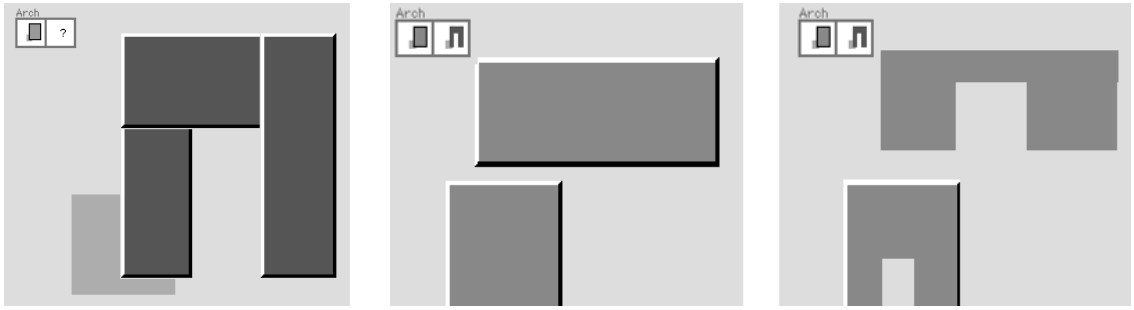
4.6 Automated layout technieken

Hoewel er vele CASE tools en toolkits ter beschikking zijn, worden nog een groot deel van de layouts vandaag de dag met de hand gecreëerd. Toch gaan bij dit manuele proces een groot aantal nadelen gepaard. Ten eerste is de creatie van een layout een langdurig en kostbaar proces. In sommige complexe gevallen kan de creatie van een layout voor een enkel scherm of een enkele presentatie enkele uren tot zelfs dagen in beslag nemen. Ten tweede is het aanleren van effectieve layout technieken een proces dat veel tijd in beslag neemt. Daarom wordt er de laatste jaren veel onderzoek gedaan naar de automatisering van dit proces. Er wordt hier dan gesproken van *automated layout techniques*, technieken die gebruik maken van een computerprogramma om het hele layout proces of een gedeelte ervan te automatiseren. Het onderzoek naar automated layout systems gebeurt in twee verschillende gebieden: *learning technieken* en *constraint-based technieken* [Lok01].

4.6.1 Learning technieken

Automated layout systemen die gebruik maken van *learning technieken* worden ook wel programming by demonstration of programming by example systemen genoemd. Deze systemen trachten op basis van een aantal voorbeelden, die door de gebruiker

gedefinieerd worden, een algemene actie of een algemeen object af te leiden. De gebruiker demonstreert de verschillende stappen van een bepaalde handeling of de opbouw van een bepaald object op een concreet voorbeeld. Het systeem zal de verschillende stappen opnemen en door middel van machine learning technieken relaties tussen objecten en afhankelijkheden tussen operaties trachten op te sporen. De opgenomen handeling wordt vervolgens gegeneraliseerd zodat deze op een later ogenblik op gelijkaardige voorbeelden kan worden toegepast [Lieb93].



Figuur 26 - Mondrian, uit [Lieb93]

In figuur 26 wordt een voorbeeld gegeven van het *Mondrian* systeem [Lieb93]. Mondrian is een grafische editor waar door middel van programming by demonstration technieken nieuwe grafische primitieven en procedures kunnen worden toegevoegd. In figuur 26 wordt links een nieuw grafisch object gedefinieerd, namelijk een boog. De gebruiker transformeert een gewone rechthoek in een boog en demonstreert de opbouw ervan aan het systeem. Het systeem neemt de verschillende stappen op en generaliseert het proces. Wanneer de gebruiker op een later ogenblik een rechthoek wil transformeren in een boog, dan kan hij het boog commando op een willekeurige rechthoek toepassen. Dit proces wordt weergegeven in het middelste en rechtergedeelte van figuur 26.

Een tweede systeem dat wordt besproken is Marquise. *Marquise* [Myer93] is een systeem voor de ontwikkeling van user interfaces voor grafische applicaties. Het legt de nadruk op de ontwikkeling van het hoofdcanvas, waar de eindgebruiker objecten kan creëren en manipuleren met de muis. In Marquise is het mogelijk het grafische uitzicht van de interface te tekenen en het gedrag ervan te specificeren door middel van demonstraties. Hiervoor wordt gebruik gemaakt van vier verschillende modi: build, run, train en show. In “build mode” kan de designer de toekomstige user interface schetsen. Hij tekent de statische onderdelen waaruit de user interface zal bestaan zoals een label, een command button of een palet met mogelijke kleuren. “Train mode” wordt gebruikt om de acties van de eindgebruiker te definiëren. De designer maakt hiervoor gebruik van het toetsenbord en de muis om te demonstreren hoe de eindgebruiker de toekomstige user interface zal gebruiken. Marquise zal de verschillende demonstraties evalueren en deze resultaten veralgemenen om vervolgens de user interface te creëren. In “show mode”

worden daarna de reacties van het systeem op deze acties gedemonstreerd. In “run mode” kan de ontwikkelde user interface uiteindelijk worden getest.

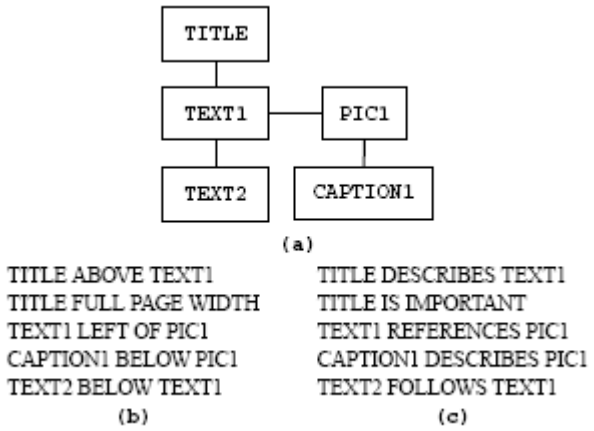
Het grote voordeel van programming by demonstration technieken is het feit dat er geen nood is aan een programmeertaal om een bepaalde procedure of een bepaald object te definiëren. De gebruiker kan door middel van een aantal voorbeelden aan het systeem demonstreren wat een bepaald commando moet doen of hoe een bepaald object moet worden opgebouwd. De systemen kunnen dus gebruikt worden door mensen zonder enige programmeerervaring.

4.6.2 Constraint-based technieken

Via constraint-based technieken wordt het mogelijk een layout voor te stellen als een verzameling constraints [Lok01]. Constraints worden daarom vooral in grafische applicaties gebruikt voor het definiëren en behouden van bepaalde layout eigenschappen.

De meeste conventionele grafische programma’s maken geen gebruik van constraints. Het is wel mogelijk om een aantal relaties tussen objecten te specificeren, zoals bijvoorbeeld het uitlijnen van objecten, maar deze relatie wordt echter maar éénmaal gedefinieerd. Van zodra één van de objecten wordt verplaatst, zullen de overige objecten zich niet mee verplaatsen en zal dus niet meer aan de gewenste relatie worden voldaan. Door gebruik te maken van constraints kan er voor worden gezorgd dat deze relaties wel blijven bestaan. Constraints vergemakkelijken daardoor het layout proces en zorgen ervoor dat de gewenste precisie blijft behouden [Badr98b].

Een constraint drukt een relatie uit waaraan moet worden voldaan [Cons] en kan worden opgedeeld in twee groepen namelijk *abstracte* en *ruimtelijke* constraints [Lok01]. In figuur 27 zijn een aantal voorbeelden van constraints terug te vinden.



Figuur 27 - Voorbeelden van constraints, uit [Lok01]

Een *abstracte* constraint beschrijft een hogere relatie van één of meerdere componenten zoals bijvoorbeeld “title is important” of “text1 references pic1”. *Ruimtelijke* constraints daarentegen hebben betrekking op de positie en grootte van user interface elementen. Voorbeelden van ruimtelijke constraints zijn “title above text1” en “text1 left of pic1”.

Constraints worden uiteindelijk opgelost door een constraint solver. Ruimtelijke constraints kunnen onmiddellijk worden doorgegeven aan een constraint solver, abstracte constraints moeten echter eerst geconverteerd worden naar ruimtelijke constraints alvorens ze kunnen worden opgelost.

Het doel van een constraint-based automated layout systeem is de verzameling van gedefinieerde constraints, ook wel het constraint systeem genaamd, om te zetten in een juiste grootte en positie voor elke component in de user interface. Automated layout technieken lossen dus een vorm van het constraint satisfaction problem op [Lok01].

4.6.2.1 Constraint satisfaction problem

Een constraint satisfaction problem (CSP) kan gedefinieerd worden als een verzameling variabelen X_1, X_2, … , X_n en een verzameling constraints C_1, C_2, …, C_m. Elke variabele X_i heeft een niet-leeg domein D_i met daarin een aantal mogelijke waarden. Elke constraint C_i heeft betrekking op een deelverzameling van de variabelen en specificeert de mogelijke combinaties voor de waarden van die deelverzameling.

Het vierkleurenprobleem is een voorbeeld van een CSP. Wanneer bovenstaande definitie wordt toegepast op het vierkleurenprobleem dan kan dit als volgt worden voorgesteld. Elke regio uit het land stelt een variabele X_n voor met n het totaal aantal regio’s in het land. Het domein D_i van elke variabele bestaat uit de mogelijke kleuren namelijk {rood, groen, blauw, geel}. De constraint die wordt opgelegd is dat twee aangrenzende regio’s niet in dezelfde kleur mogen worden ingekleurd. Deze constraint specificeert de volgende verzameling van mogelijke combinaties voor twee aangrenzende regio’s: {(rood, groen), (rood, blauw), (rood, geel), (groen, rood), (groen, blauw), (groen, geel), (blauw, rood), (blauw, groen), (blauw, geel), (geel, groen), (geel, rood), (geel, blauw)}.

Een constraint satisfaction problem kan ook gevisualiseerd worden door middel van een constraint graaf. In deze graaf worden variabelen voorgesteld als nodes, en constraints als bogen [Russ03].

4.6.2.2 One-way en multi-way constraints

Zoals reeds beschreven kunnen constraints worden onderverdeeld in abstracte en spatiale constraints, naargelang de relatie die ze definiëren. Daarnaast kan er ook nog een tweede opsplitsing worden gemaakt namelijk tussen *one-way* en *multi-way constraints*.

One-way constraints zijn gemakkelijk te begrijpen en gedragen zich voorspelbaar, maar beschikken over minder mogelijkheden om relaties uit te drukken dan multi-way constraints [Huds90].

Multi-way constraints zijn krachtiger dan one-way constraints. Relaties kunnen ook op een duidelijkere en meer uniforme manier gedefinieerd worden. Daarnaast zijn multi-way constraint solvers ook flexibeler dan one-way constraint solvers. Ze beschikken immers over de mogelijkheid een keuze te maken over welke functie ze zullen gebruiken om een waarde verandering van een variabele door te voeren in de rest van het constraint systeem. Hier tegenover staat wel dat multi-way constraint systemen moeilijker te controleren en te voorspellen zijn dan one-way constraint systemen [Sann93].

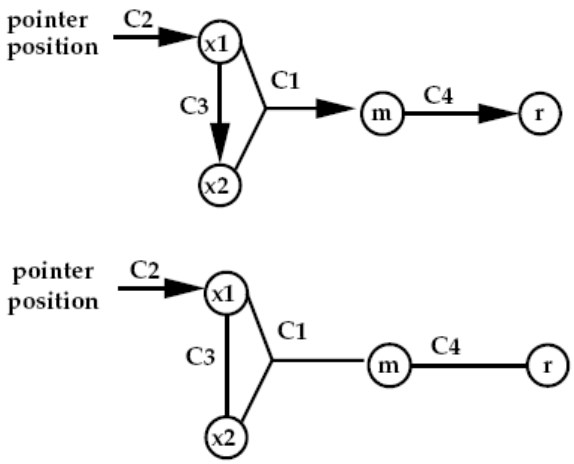
$$
\begin{array}{lllll}
C_1: & m & = & \frac{(x_1+x_2)}{2} \\
C_2: & x_1 & = & \text{pointer position} \\
C_3: & x_2 & = & x_1 + 6 \\
C_4: & r & = & m^2
\end{array}
$$

Figuur 28 - Voorbeeld van een constraint systeem, uit [Badr98b]

In figuur 28 wordt een voorbeeld gegeven van een constraint systeem. In figuur 29 worden bijhorende one-way en multi-way constraint grafen weergegeven waarbij de nodes in de graaf variabelen voorstellen en de bogen constraints.

Wanneer in het one-way constraint systeem de waarde van een variabele verandert, moet de solver de waarden van de overige variabelen gaan herberekenen zodat aan de gespecificeerde constraints blijft voldaan. Het systeem dient de overgebleven constraints in de graaf in topologische volgorde op te lossen [Badro98b]. Wanneer in figuur 29 bijvoorbeeld de waarde van x_1 verandert, wordt de rest van de graaf doorlopen en worden nieuwe waarden berekend voor x_2, m en r [Verm05].

One-way constraints hebben slecht één enkele output variabele: in C_3 kan de solver bijvoorbeeld enkel de waarde van x_2 veranderen. Een multi-way constraint solver is in staat om x_1 te berekenen door x_2 - 6.



Figuur 29 - Boven one-way constraint graaf, onder multi-way constraint graaf, uit [Badr98b]

4.6.2.3 Constraint hiërarchieën

Een constraint solver moet dus de waarden voor de variabelen in het constraint systeem gaan bepalen. Wanneer een variabele van waarde verandert, dan moeten de overige variabelen geüpdate worden zodat aan de gedefinieerde constraints blijft voldaan. In een one-way constraint systeem is het duidelijk hoe deze nieuwe waarden voor de variabelen berekend moeten worden. Wanneer het constraint systeem gebruik maakt van multi-way constraints of wanneer er cycles in de constraint graaf verschijnen, bestaan er echter meerdere mogelijkheden om de status van het systeem te updaten en dus ook meerdere juiste oplossingen van het constraint systeem. Het systeem is dan underconstrained [Sann93].

Constraint hiërarchieën kunnen gebruikt worden om het probleem van underconstrained systemen op te lossen. Ze helpen beslissen in welke volgorde de constraints opgelost dienen te worden en kunnen aangeven welke oplossing van de verschillende mogelijke oplossingen de meest gewenste is. In een hiërarchie kunnen constraints op verschillende niveaus gedefinieerd worden namelijk als “required” en als “preferential”. Aan required constraints moet te allen tijde worden voldaan. Indien mogelijk moet het systeem ook trachten te voldoen aan preferential constraints, maar er zal geen fout optreden wanneer dat niet mogelijk is [Sann93].

Om een constraint hiërarchie op te lossen wordt er gebruik gemaakt van comparators. Deze zullen de beste oplossing van de hiërarchie proberen te vinden waarbij er wordt voldaan aan alle required constraints en waarbij eveneens wordt getracht te voldoen aan de gedefinieerde preferential constraints. De twee meest voorkomende comparators zijn locally-predicate-better en locally-error-better [Badr98b].

4.6.2.4 Constraint solvers

Een constraint systeem wordt uiteindelijk opgelost door een constraint solver. Er zijn verschillende constraint solvers beschikbaar die elk andere soorten van constraint solving technieken implementeren. Eén van de eerste constraint solving technieken die werden ontwikkeld, zijn *local-propagation* (LP) technieken. Bij local-propagation wordt een constraint gebruikt om de waarde van een variabele te bepalen. Wanneer de waarde van deze variabele wordt gevonden, kan het systeem gebruik maken van een andere constraint om de waarde van een volgende variabele te vinden, enzovoort…. Een constraint is daarvoor uitgerust met een aantal functies. Een functie krijgt een aantal input variabelen als argument, en berekent vervolgens de waarde van een output variabele waarbij er wordt gezorgd dat er aan de constraint wordt voldaan [Sann93].

Het voordeel van local-propagation technieken is dat ze heel simpel, efficiënt en algemeen zijn [Sann93] [Badr98b]. Ze hebben ook de mogelijkheid om niet-numerische constraints op te lossen. Het nadeel van local-propagation technieken is dat ze niet in staat zijn alle constraint systemen op te lossen [Sann93]. Ze beschikken bijvoorbeeld niet over de mogelijkheid om meerdere constraints gelijktijdig af te handelen en zijn dus niet bruikbaar voor systemen waar meerdere constraints gelijktijdig gemanipuleerd moeten worden [Badr98b].

Local-propagation technieken kunnen gebruikt worden voor het oplossen van one-way of multi-way constraints. Bij one-way constraints wordt er één enkele functie gebruikt die een waarde berekent voor één enkele output variabele. Bij multi-way constraints wordt er gebruik gemaakt van meerdere functies voor het berekenen van de waarden van de variabelen die door de constraint beperkt worden [Sann93].

Een tweede soort constraint solvers die gebruikt kunnen worden, zijn *numeric solvers*. In tegenstelling tot local propagation technieken zijn deze enkel toepasbaar op numerieke waarden. Numeric solvers kunnen worden opgedeeld in *iterative numeric solvers* en *direct numeric solvers*. *Iterative numeric solvers* zijn niet zo geschikt voor het gebruik in grafische applicaties omwille van een aantal redenen. Ten eerste wordt er gebruik gemaakt van iteratieve methoden voor het oplossen van het constraint systeem, waardoor deze methoden erg traag zijn. Ten tweede zijn iteratieve methoden erg afhankelijk van de initiële situatie. Enkele kleine veranderingen aan de startsituatie kunnen een totaal ander resultaat opleveren. Een laatste reden is het feit dat iteratieve methoden niet gemakkelijk te implementeren zijn [Badr98b]. Omwille van deze redenen worden ze vaak enkel gebruikt in situaties waar local propagation technieken falen [Howe95].

Direct numeric solvers proberen de nadelen van iterative numeric solving technieken te overkomen door één exacte oplossing voor het constraint systeem te vinden. Cassowary en QOCA zijn voorbeelden van direct numeric solvers. Ze zijn beide gebaseerd op een incrementele versie van het simplex algoritme. Het simplex algoritme kan worden opgesplitst in twee fasen: in fase één wordt een initiële oplossing voor het constraint systeem gezocht, in fase twee een optimale oplossing. De werking van de twee algoritmen is gelijkaardig tot aan fase twee. In fase twee is er een verschil in de manier waarop ze de beste oplossing voor het constraint systeem kiezen. In Cassowary worden gewichten op constraints geplaatst om zo de optimalisatie functie te controleren. In QOCA daarentegen worden gewichten aan variabelen gekoppeld. Elke variabele is daarvoor uitgerust met een gewenste locatie en een gewicht dat aangeeft hoe sterk deze voorkeur is. Deze aanpak maakt QOCA vooral bruikbaar voor geometrische applicaties, omdat de objecten zich zo dicht mogelijk bij de gewenste locatie trachten te plaatsen. QOCA's methode is hierdoor wel complexer op het gebied van implementatie en berekening. De aanpak van Cassowary is iets algemener omdat deze het mogelijk maakt voorkeuren uit te drukken op willekeurige constraints, wat in QOCA niet mogelijk is [Badr98b].

Een laatste soort methode die gebruikt kan worden zijn *differential methods*. In tegenstelling tot de vorige technieken, waarin de solver verantwoordelijk is voor het definiëren van een initiële oplossing en het behouden van deze oplossing, worden in differential methods enkel constraints gedefinieerd waaraan reeds is voldaan. Op die manier wordt het probleem van constraint solving sterk vereenvoudigd [Badr98b].

4.6.2.5 Constraint visualisatie

Een laatste belangrijk punt in een constraint-based systeem is constraint visualisatie. Om de gebruiker te allen tijde op de hoogte te houden van de aanwezige constraints, is het belangrijk dat constraints gevisualiseerd worden. Er zijn drie mogelijke visualisaties: *tekstuele beschrijving*, *schematische voorstelling* en *grafische representaties*.

Wanneer de constraints door middel van *tekst* worden beschreven dan heeft dit het grote voordeel dat de constraints makkelijk aanpasbaar zijn. Het nadeel echter is dat de tekst en de tekening van elkaar gescheiden zijn en het dus voor de gebruiker moeilijker wordt deze twee aan elkaar te koppelen. Ditzelfde probleem geldt ook voor *schematische voorstellingen*.

Bij *grafische representaties* worden constraints rechtstreeks op de tekening uitgetekend. Het voordeel is dat de gebruiker een overzicht krijgt van welke constraints

welke objecten beïnvloeden. Grafische representaties hebben echter ook een aantal nadelen. Zo moeten elke constraint op een éénduidige manier worden voorgesteld zodat het voor de gebruiker duidelijk is wat de constraint doet en welke objecten daardoor worden beïnvloed. Dit kan vooral een probleem zijn bij systemen die veel verschillende soorten constraints implementeren. Een tweede nadeel is dat een grafische representatie moeilijk te editeren is, zeker wanneer er clustering optreedt [Glei94].

4.7 Conclusie

In dit hoofdstuk werden een aantal grafische user interface toolkits uitvoerig besproken. Zoals in de inleiding van dit hoofdstuk al werd vermeld, maken designers meestal in een later stadium van de design cyclus gebruik van toolkits om de interface te creëren en de layout ervan op te bouwen. Er kan besloten worden dat deze toolkits over tal van mogelijkheden beschikken voor de specificatie van layout eigenschappen en commando's die reeds hun succes bewezen hebben. Wanneer we echter terugkijken naar de sketch-based user interface design tools die in het vorig hoofdstuk besproken werden, moeten we opmerken dat geen van deze tools over zulke mogelijkheden beschikt, hoewel dit toch een meerwaarde zou kunnen betekenen. Daarom zullen de volgende algemene layout commando's in het prototype worden toegevoegd:

> "Doelstelling 5: Het prototype moet de gebruiker in staat stellen de grootte en positie van elementen te laten aanpassen."

> "Doelstelling 6: Het prototype moet het voor de gebruiker mogelijk maken enkele simpele layout managers toe te passen."

> "Doelstelling 7: Het prototype moet de gebruiker in staat stellen elementen toe te voegen die het mogelijk maken een bepaalde ruimte in een layout te reserveren."

Het nadeel van de specificatie van layout commando's in de besproken toolkits is het feit dat ze maar éénmalig gedefinieerd worden. Wanneer de designer bijvoorbeeld een aantal widgets links uitlijnt, en daarna één van de widgets verplaatst, dan zal er niet meer aan de gedefinieerde uitlijning worden voldaan. Daarom worden layouts vaak voorgesteld als een verzameling constraints, waarbij de constraints worden opgelost door een constraint solver die er op toe ziet dat aan de gedefinieerde constraints blijft voldaan.

Hoofdstuk 5

Constraint-based Systemen

5.1 Overview

In het vorige hoofdstuk werd reeds een inleiding gegeven tot het gebruik van constraint-based technieken voor het specificeren van een layout. In dit hoofdstuk worden een aantal constraint-based systemen van naderbij bekeken. Hierbij wordt vooral gekeken naar de manier waarop constraints gedefinieerd worden, hoe deze gevisualiseerd worden en op welke manier ze uiteindelijk worden opgelost. Het overzicht is gebaseerd op het werk dat gepresenteerd wordt in [Badr98b]. In dit hoofdstuk worden echter alleen de systemen besproken die relevant zijn voor deze thesis. Het gaat hier om een aantal tekenprogramma's, meer bepaald Sketchpad, Briar, Unidraw, Chimera, Pegasus, de Scheme window manager en tenslotte een aantal toolkits waaronder Garnet, Bramble en OPUS. Voor een volledig overzicht van constraint-based systemen wordt echter verwezen naar [Badr98b]. Op het einde van dit hoofdstuk wordt tenslotte een vergelijkend overzicht van de besproken systemen gegeven.

5.2 Constraint-based tekenprogramma's

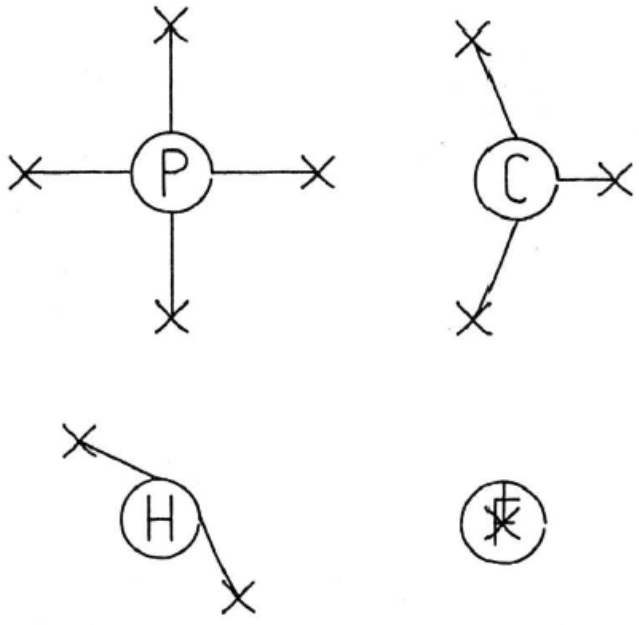
5.2.1 Sketchpad

Sketchpad [Suth63] werd in 1964 ontwikkeld door Sutherland en was het eerste interactieve constraint-based systeem. Met de ontwikkeling van Sketchpad ontstond een nieuwe vorm van mens-machine interactie. Tot op dat moment bestond de communicatie tussen mens en computer immers uit het intypen van geschreven statements. In een aantal

applicatiedomeinen, zoals bijvoorbeeld het uittekenen van elektronische circuits en het beschrijven van mechanische onderdelen bleken die geschreven statements geen goed communicatiemiddel. Daarnaast nam het uitschrijven van zulke statements veel tijd in beslag. Sketchpad maakte het voor de gebruiker mogelijk met de computer te communiceren door middel van getekende objecten en lijnen. Deze objecten bestaan uit primitieve vormen zoals cirkels en lijnen.

Door gebruik te maken van een lichtpen in combinatie met buttons, kan de gebruiker in Sketchpad een tweedimensionale grafische scène gaan opbouwen. De lichtpen wordt gebruikt om objecten te positioneren en te selecteren. De buttons worden gebruikt om commando's en constraints op de objecten of op de scène toe te passen. Wanneer de gebruiker bijvoorbeeld twee lijnen parallel wil uitlijnen, dient hij eerst de lijnen aan te duiden met de lichtpen om vervolgens het constraint commando toe te passen. Andere voorbeelden van mogelijke commando's zijn: move, delete, copy, circle center,…. Voor een volledig overzicht wordt verwezen naar Appendix B van [Suth63].

Het Sketchpad systeem houdt informatie bij over de structuur van de opgebouwde scène. In die structuur wordt eveneens informatie opgeslagen over opgelegde constraints in de vorm van constraint blocks. Een constraint block houdt de verscheidene eigenschappen van het bijhorende constraint type bij. Wanneer de gebruiker een object in de tekening verplaatst, zal de rest van de getekende scène ook worden aangepast. Wanneer bijvoorbeeld een punt in een polygoon wordt verplaatst, zullen de aanliggende zijden zich mee verplaatsen.



Figuur 30 - Grafische voorstelling van constraints in Sketchpad, uit [Suth63]

Opdat de gebruiker deze constraints kan gaan manipuleren, bestaat er een mogelijkheid om een grafische presentatie ervan weer te geven. Door die presentatie is de gebruiker eveneens op de hoogte van de reeds opgelegde constraints en kan hij deze ook gaan selecteren met de lichtpen om deze vervolgens te gaan wijzigen of te verwijderen. De visuele presentatie van de constraints kan aan en uit worden gezet. In figuur 30 worden een aantal voorbeelden gegeven van zulke visuele representaties. Voor een volledig overzicht wordt ook hier verwezen naar Appendix A in [Suth63].

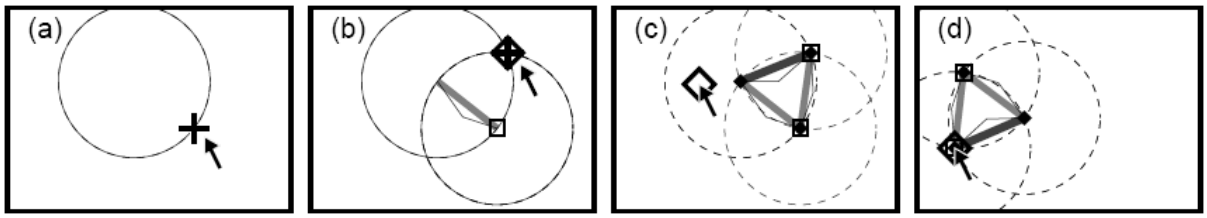
In Sketchpad kan gebruik worden gemaakt van multi-way constraints. Voor het oplossen van deze constraints wordt een local propagation based solver gebruikt, die hier ook wel “one-pass method” wordt genoemd. De werking van local propagation solvers werd reeds besproken in sectie *“4.6.2.4 Constraint solvers”*. Wanneer de LP solver er echter niet in slaagt een juiste oplossing voor het constraint systeem te vinden, wordt teruggevallen op de “relaxation method”. Deze relaxation method is een voorbeeld van een iterative numeric techniek die eveneens in sectie *“4.6.2.4 Constraint solvers”* werd besproken. De relaxation method zal trachten de fouten, die in het systeem geïntroduceerd werden door de gespecificeerde constraints, te verminderen. Elke variabele wordt opnieuw geëvalueerd om zo geïntroduceerde fouten te verwijderen. Deze methode is zeer betrouwbaar, maar traag, vooral wanneer een groot aantal variabelen geëvalueerd moeten worden [Suth63].

5.2.2 Briar

Een tweede constraint-based tekenprogramma dat wordt besproken is *Briar*. In Briar worden constraints gedefinieerd met behulp van “augmented snap-dragging”, een uitgebreidere vorm van snap-dragging. Snap-dragging [Bier88] is een techniek die ontwikkeld is voor het snel en precies ontwerpen van twee- en driedimensionale vormen. Deze techniek maakt gebruik van twee cursors: een hardware en software cursor. De hardware cursor wordt rechtstreeks door de gebruiker aangestuurd door middel van een muis, stylus of ander pointing device. In de hardware cursor bevindt zich de software cursor, die ook wel caret wordt genoemd. De positie en oriëntatie van de caret worden bepaald door twee factoren: enerzijds door de grafische objecten die zich in de scène bevinden en anderzijds door de positie van de hardware cursor. Van zodra de hardware cursor dicht genoeg in de buurt van een object komt, zal de caret de hardware cursor verlaten en zich met het object verbinden [Bier88].

Augmented snap-dragging maakt gebruik van snap-dragging voor het tot stand brengen van relaties tussen objecten. Deze relaties worden echter blijvend gemaakt door middel van constraints. De gebruiker heeft hiervoor de keuze uit twee verschillende soorten constraints: point-coincident en point-on-object. Point-coincident maakt het

mogelijk de cursor met een punt te verbinden, point-on-object zal de cursor verbinden met een zijde of curve van het object. Andere relaties kunnen gespecificeerd worden door middel van alignment objects. Een alignment object is een object dat geen deel uitmaakt van de scène, maar wel gebruikt kan worden om snapping op uit te voeren. Om bijvoorbeeld [Badr98b] een punt p op een afstand d van het punt q te plaatsen, kan er een alignment cirkel met diameter d rond het punt p worden getekend [Glei94].



Figuur 31 - Constraints en alignment objects in Briar, uit [Glei94]

In tegenstelling tot de meeste constraint-based systemen, wordt er hier enkel gebruik gemaakt van constraints om reeds gedefinieerde relaties te onderhouden. De constraint solver is dus steeds op de hoogte van de initiële situatie en dient dus nooit constraints op te lossen vanuit een willekeurige status. Op die manier kunnen er geen conflicten tussen de constraints optreden, waardoor een aantal problemen worden voorkomen. De taak van de solver wordt dus voor een groot deel vereenvoudigd. Deze methode van constraint maintenance wordt ook wel “differential method” genoemd, en werd reeds besproken in *“4.6.2.4 Constraint solvers”* [Glei94].

De constraints in het systeem worden op een grafische manier weergegeven. De point-coincident en de point-on-object constraint worden respectievelijk voorgesteld als een leeg vierkant en een opgevulde ruit. Alignment objects worden met een stippellijn weergegeven. Constraints kunnen op twee manieren verwijderd worden. Via de eerste manier kunnen constraints worden verwijderd door ze van het object of punt af te “scheuren”. Wanneer tijdens het slepen een speciale toets wordt ingehouden, zullen de constraints die verbonden zijn met dat punt verwijderd worden. Wanneer men niet alle constraints in één keer wenst te verwijderen, kan het systeem ook in constraint-free mode worden gezet. In deze modus kan de gebruiker de objecten gaan manipuleren alsof er geen constraints aan verbonden zijn. Wanneer terug naar normale modus wordt overgeschakeld, worden de constraints waaraan niet meer wordt voldaan verwijderd [Glei94].

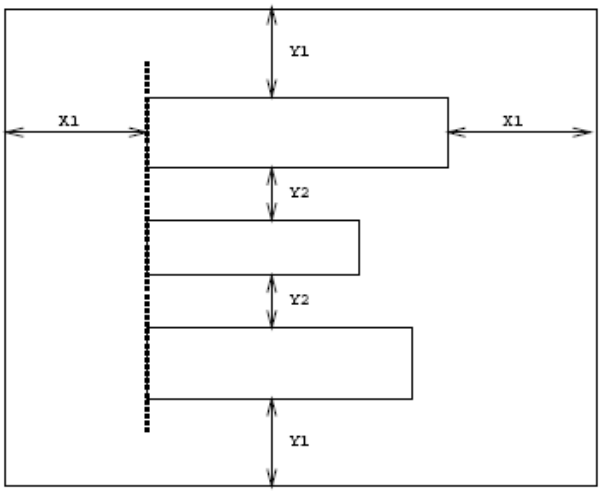
De objecten in de scène kunnen verplaatst worden door ze te verslepen. Wanneer een object wordt verplaatst, zullen de overige objecten eventueel ook worden aangepast zodat steeds aan de gedefinieerde relaties blijft voldaan. Bij elk input event worden de

constraints dus opnieuw opgelost door de solver. Deze vorm van directe manipulatie wordt constraint-based direct manipulation genoemd [Glei94].

5.2.3 Unidraw en QOCA

In [Helm92] wordt het grafische editing framework *Unidraw* geïntegreerd met de *QOCA constraint solving toolkit* tot een constraint-based, direct manipulation grafische editor.

Unidraw is een framework dat de creatie van grafische editors ondersteunt. Unidraw maakt daarvoor gebruik van een aantal abstracties waarmee de editor kan worden opgebouwd namelijk components, tools, commands en external representations. *Components* definiëren het gedrag en uiterlijk van een object, *tools* maken directe manipulatie op components mogelijk, *commands* zorgen ervoor dat operaties op components gedefinieerd kunnen worden en tenslotte zorgen *external representations* voor een mapping tussen components en een bestand of database. In Unidraw werden reeds een beperkt aantal connection constraints voorzien [Vlis89]. De constraint mogelijkheden worden uitgebreid door toevoeging van de QOCA constraint solving toolkit.



Figuur 32 - Layout constraints die opgelost kunnen worden in QOCA, uit [Helm92]

In [Helm92] werden drie mogelijke uitbreidingen besproken. Een eerste voorbeeld dat wordt gegeven is het correct weergeven van connectiviteit tussen objecten. Om ervoor te zorgen dat lijnen en objecten correct aan elkaar aansluiten kan in QOCA een objective function gedefinieerd worden die ervoor zorgt dat de lijn steeds de kortste afstand tussen twee objecten inneemt en daardoor mooi aan de objecten aansluit. Een tweede uitbreiding die wordt besproken, wordt weergegeven in figuur 32. In dit voorbeeld worden drie

objecten links met elkaar uitgelijnd. Het bovenste object wordt in de container gecentreerd. Via verticale constraints wordt aangegeven dat er ruimte moet worden vrijgehouden tussen de objecten en de container. Deze verzameling constraints vormen een systeem van lineaire gelijkheden met drie onbekenden. QOCA is in staat dit constraint systeem op te lossen doordat het over de mogelijkheden beschikt lineaire (on)gelijkheden gelijktijdig op te lossen. Een laatste uitbreiding die wordt besproken, is de mogelijkheid om gebruik te gaan maken van een aantal primitieve operaties om de stabiliteit van constraints uit te testen. Deze operaties kunnen door de designer gebruikt worden om fout gedefinieerde constraints op te sporen en te testen.

Eveneens werd Unidraw uitgebreid met een aantal andere mogelijkheden. Zo werden undo en redo operaties toegevoegd, de mogelijkheid om constraints in en uit te schakelen en de mogelijkheid om de status van het systeem op te slaan en in te laden. Net zoals in Briar wordt eveneens een vorm van constrained direct manipulation voorzien [Helm92].

5.2.4 Chimera

In *Chimera* [Kurl93] wordt een andere methode gebruikt voor constraint specificatie. Constraints kunnen er impliciet gedefinieerd worden aan de hand van één of meerdere voorbeelden. Chimera maakt hiervoor gebruik van een snapshot mechanisme. De gebruiker dient een initiële scène met objecten op te bouwen waarin aan alle gewenste constraints wordt voldaan. Vervolgens wordt van deze scène een snapshot genomen. De gebruiker kan de scène daarna wijzigen, waarna er terug één of meerdere snapshots van worden gemaakt.

Elk snapshot stelt een correcte oplossing van het constraint systeem voor. De constraints worden afgeleid vanuit de snapshots en toegepast op de objecten in de scène. Met het nemen van één snapshot kunnen dus een groot aantal constraints gelijktijdig gespecificeerd worden. Tabel 3 geeft een overzicht van de constraints die in Chimera gedefinieerd kunnen worden. Absolute constraint drukken één enkele geometrische relatie uit, terwijl relatieve constraints verschillende geometrische relaties met elkaar vergelijken. Gelijkaardige constraints die van toepassing zijn op éénzelfde verzameling hoekpunten worden ondergebracht in groepen. Hierdoor wordt het aantal constraints die door de solver opgelost moeten worden met een groot aantal verminderd.

Het voordeel van het snapshot mechanisme is dat de gebruiker niet meer rechtstreeks in contact komt met individuele low-level constraints. Het specificatieproces wordt op die manier een stuk intuïtiever en simpeler. Een tweede voordeel is de mogelijkheid om door middel van één snapshot een groot aantal constraints gelijktijdig te

specificeren. Chimera is daardoor vooral bruikbaar voor scènes die gedefinieerd worden door een groot aantal constraints.

Absolute constraints	**Relatieve constraints**
Fixed vertex location	Coincident vertices
Distance between two vertices	Relative distance between two pairs of vertices
Distance between parallel lines	Relative distance between two pairs of parallel lines
Slope between two vertices	Relative slopes between two pairs of two vertices
Angle defined by three vertices	Equality between two angles, each defined by three vertices

Tabel 3 - Overzicht Chimera constraints, uit [Kurl93]

Het snapshot mechanisme heeft echter ook een aantal nadelen. Tijdens het nemen van een snapshot kunnen een aantal ongewenste constraints gedefinieerd worden. De gebruiker dient dan de verschillende constraints te doorlopen en ongewenste constraints uit de scène te verwijderen. Vaak is het nemen van meerdere snapshots ook noodzakelijk. Eveneens is het mogelijk dat tijdens het nemen van een snapshot een aantal constraints gedefinieerd worden die overbodig zijn. Deze constraints zullen enkel de kost, die nodig is om het constraint systeem op te lossen, verhogen. Wanneer maar een klein aantal constraints gedefinieerd moeten worden, is een meer traditionelere aanpak daarom ook beter.

5.2.5 Pegasus

Pegasus [Igar98] is een tekenprogramma dat gebruikt kan worden bij het construeren van geometrische diagrammen. Tijdens het tekenen worden er door het systeem constraints opgelegd door middel van twee technieken: *interactive beautification* en *predictive drawing*.

De *interactive beautification* techniek zal op de door de gebruiker geschetste vormen constraints trachten te plaatsen. Het type constraint dat wordt geplaatst, wordt gekozen op basis van de geometrische relaties van de schets. De schets wordt vervolgens verbeterd en zal uiteindelijk aan de gebruiker worden getoond. Indien het systeem meer

dan één mogelijke kandidaat voor de geschetste vorm vindt, zullen deze allemaal aan de gebruiker gepresenteerd worden. De gebruiker dient hieruit dan de juiste vorm te kiezen.

De tweede techniek *predictive drawing* tracht de input van de gebruiker te voorspellen. Wanneer de gebruiker een object tekent waarvan de vorm gelijkaardig is aan die van een reeds getekende vorm, dan zal het systeem automatisch de volgende segmenten aan de gebruiker presenteren. De gebruiker kan de segmenten dan selecteren [Igar98].

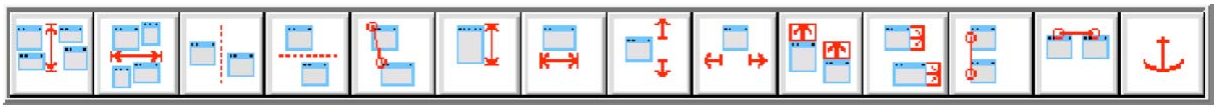
In totaal ondersteunt Pegasus de volgende zeven constraints: connection, parallellism, perpendicularity, alignment, congruence, symmetry en interval equality [Badr95].

5.3 Constraint-based window layout systemen

5.3.1 Scheme Constraints Window Manager

Scwm [Badr00] is een window manager die ontwikkeld werd voor het X Window System. In tegenstelling tot de meeste window managers, die enkel gebruik maken van directe manipulatie voor het positioneren van vensters, is het in Scwm mogelijk hiervoor gebruik te maken van constraints.

Voor het toepassen van constraints op vensters maakt Scwm gebruik van een toolbar. Elke knop in de toolbar stelt een constraint klasse voor. De gebruiker kan een constraint op een venster toepassen door het venster en de gewenste constraint te selecteren. In figuur 33 wordt een overzicht gegeven van de constraint toolbar. Bij sommige constraints, zoals bijvoorbeeld bij het uitlijnen van vensters, is het nodig aan te geven op welk gedeelte van het venster de constraint moet worden toegepast. Elk venster is daarom opgedeeld in negen even grote gebieden, nonants genaamd. De gebruiker dient het gewenste gebied te selecteren door het met de muis aan te klikken. Wanneer de gebruiker bijvoorbeeld de rechterkant van een venster wenst uit te lijnen met de linkerkant van een ander venster, dient hij eerst één van de drie oostelijke gebieden van het eerste venster te selecteren, om vervolgens op één van de westelijke gebieden van het tweede venster te klikken. Deze techniek heeft als voordeel dat er minder knoppen in de toolbar moeten worden opgenomen en dat sommige constraint klassen, zoals uitlijning, op een meer algemene manier gebruikt kunnen worden [Badr00].



Figuur 33 - Scwm constraint toolbar, uit [Badr00]

Een overzicht van de reeds toegevoegde constraints verschijnt in de “constraint investigation interface”. Wanneer de gebruiker met de muis over een constraint beweegt, zal de representatie van de constraint worden uitgetekend op het scherm. Deze visuele representaties komen sterk overeen met de afbeeldingen die in de toolbar worden gebruikt. Via de investigation interface is het eveneens mogelijk constraints tijdelijk in en uit te schakelen door bijhorende check boxen aan en uit te vinken. Constraints kunnen eveneens verwijderd worden met behulp van de delete-knop. De wijzigingen die in de investigator worden aangebracht, worden onmiddellijk doorgevoerd naar het hoofdvenster. Om de gebruiker niet in de war te brengen wordt de verplaatsing van een venster naar een eventuele nieuwe locatie geanimeerd [Badr00]. In tabel 4 worden de verschillende constraints omschreven.

Constraint	**Omschrijving**
Constant Height/Width Sum	De totale hoogte/breedte van twee vensters moet constant blijven
Horizontal/Vertical Separation	Het venster moet zich altijd boven of links van een andere venster bevinden
Strict Relative Position	De relatieve posities van twee vensters moeten worden behouden
Vertical/Horizontal Maximum Size	De hoogte/breedte van een venster moet onder een bepaalde grens liggen
Vertical/Horizontal Minimum Size	De hoogte/breedte van een venster moet boven een bepaalde grens liggen
Vertical/Horizontal Relative Size	De verandering in hoogte/breedte van twee vensters moet constant blijven. Bijvoorbeeld: beide vensters moeten met dezelfde hoeveelheid worden vergroot of verkleind
Vertical/Horizontal Alignment	Een hoekpunt of middelpunt van een venster moet worden uitgelijnd met een hoekpunt of middelpunt van een ander venster
Anchor	Het venster moet zijn huidige positie behouden

Tabel 4 - Overzicht Scwm constraints

Soms is het echter nodig een aantal constraints toe te passen alvorens het gewenste resultaat wordt bereikt. In [Badr00] wordt het voorbeeld van tiling gegeven. Wanneer men tiling wil toepassen op twee vensters zijn hiervoor drie tot vijf constraints nodig.

Indien men tiling wenst toe te passen op meerdere vensters wordt dit een erg tijdrovend proces. Dit probleem wordt opgelost door de gebruiker de mogelijkheid te geven zelf constraints gaan samen te stellen. Dit gebeurt door middel van een programming-by-demonstration techniek. De constraints en relaties die de gebruiker toepast op de vensters in de workspace worden opgenomen en vervolgens omgezet in een nieuw constraint klasse object. De nieuwe constraint klasse wordt eveneens in de toolbar ondergebracht zodat deze op een later ogenblik nogmaals kan worden gebruikt.

Scwm ondersteunt ook snapping. Wanneer twee vensters tegen elkaar zijn geplaatst, is het eveneens mogelijk deze snapping om te zetten naar een blijvende constraint. Deze uitgebreidere vorm van snapping wordt augmented snapping genoemd. De gedefinieerde constraints worden in Scwm opgelost door de Cassowary constraint solving toolkit [Badr98a].

5.4 Constraint-based UI widget toolkits

5.4.1 Amulet

Amulet [Myer97] is een user interface development toolkit waarmee interactieve grafische user interfaces ontworpen kunnen worden voor het Unix, Windows en Mactintosh systeem. Amulet bevat een aantal voorgedefinieerde objecten op een palet die gebruikt kunnen worden om de user interface op te bouwen. De objecten zijn uitgerust met een aantal slots. Ieder slot heeft een naam en kan een waarde van eender welk type bevatten. Deze waarde kan een constante waarde zijn of kan berekend worden door één of meerdere constraints. Twee soorten one-way constraints kunnen hiervoor gebruikt worden: *formula constraints* en *indirecte constraints*.

Formula constraints werken volgens hetzelfde principe als een spreadsheet formule. Ze kunnen aan een slot van een object worden toegevoegd, en berekenen er vervolgens een waarde die gebaseerd kan zijn op andere waarden afkomstig uit andere slots van datzelfde of andere objecten. Daarnaast kan er ook gebruik worden gemaakt van een tweede soort constraints namelijk *indirecte constraints*. Een voorbeeld van een indirecte constraint is de breedte van een object die gelijk gesteld wordt aan de maximum breedte van alle andere objecten. Constraints kunnen C++ code bevatten.

Eén van de voorgedefinieerde objecten is uitgerust met gesture recognition ondersteuning. De gesture recognition engine, Agate genaamd, is gebaseerd op Rubine's algoritme dat in hoofdstuk 6 zal worden besproken. De gebruiker dient een tiental voorbeelden van het gesture te specificeren waarna het gekoppeld kan worden aan een operatie.

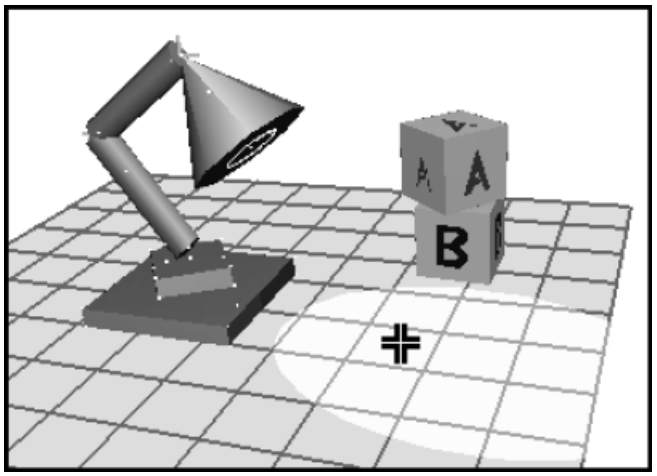
Amulet was het eerste systeem waarin het gelijktijdig gebruik van meerdere constraint solvers ondersteund werd. De constraints worden opgelost door middel van local propagation technieken (zie *"4.6.2.4 Constraint solvers"*). Eveneens bestaat de mogelijkheid om nieuwe constraint solvers toe te voegen waaronder een *web constraint solver* en een *animation constraint solver*.

De *web constraint solver* wordt gebruikt om de consistentie tussen lijnen en polygonen te behouden. De *animation constraint solver* laat het toe animaties aan objecten toe te voegen. Wanneer een animation constraint aan een slot wordt toegevoegd, zal deze het slot invullen met waarden die tussen de oude en de nieuwe gedefinieerde waarde liggen.

5.4.2 Bramble

Bramble [Glei93] is een toolkit waarmee grafische applicaties ontwikkeld kunnen worden. Bramble legt hierbij vooral de aandacht op de ondersteuning van grafische manipulatietechnieken en maakt daarvoor gebruik van differentiële methoden (zie *"4.6.2.4 Constraint solvers"*). Briar, dat eerder besproken werd in sectie *"5.2.2 Briar"*, is een grafisch tekenprogramma dat ontworpen werd met de Bramble toolkit.

In Bramble hebben grafische objecten een aantal "aspects" waarin de gebruiker geïnteresseerd kan zijn. Bij een lijn zijn deze aspects bijvoorbeeld de posities van de eindpunten, de positie van het middelpunt, de lengte van de lijn, de oriëntatie van de lijn,…. Het basisidee is dat de gebruiker de aspects van een grafisch object kan gaan manipuleren. Aspects die door de gebruiker direct gemanipuleerd kunnen worden, worden "controls" genoemd.



Figuur 34 - Manipulatie van een control in Bramble, uit [Glei93]

In [Glei93] wordt het voorbeeld gegeven van het verplaatsen van de lichtstraal van een bureaulamp. In een traditioneel driedimensionaal systeem zou de gebruiker dit bereiken door de hoeken van de arm aan te passen. In Bramble kan de gebruiker de richting van de lichtstraal direct manipuleren door de plaats waar de lichtstraal op de grond schijnt (zie figuur 34) te verplaatsen. De constraint engine zal automatisch de positie en oriëntatie van de bureaulamp updaten. De lichtstraal op de vloer is dus één van de controls die gebruikt kan worden om de parameters van de lamp te manipuleren.

Een tweede belangrijk element in Bramble is het feit dat het systeem een continu idee heeft van tijd. Grafische objecten blijven continu bestaan en verplaatsen zich met een constante beweging. Op het einde van elk tijdsinterval wordt de status van het systeem geüpdate waardoor op elk ogenblik een correct beeld van de systeem status wordt gegeven.

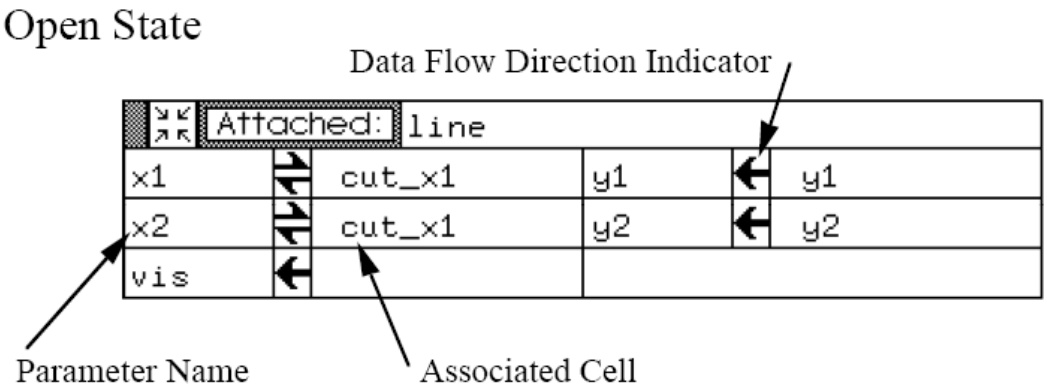
5.4.3 OPUS

OPUS [Huds90] staat voor On-line Penguims-based user interface en is de grafische user interface designer van Penguims.

Penguims (Programmable Environment for Graphical User Interface Management and Specification) [Huds93] is een interactieve omgeving die gebruikt maakt van een geavanceerde vorm van spreadsheets voor de ontwikkeling van grafische user interfaces. In tegenstelling tot “gewone” spreadsheets, is de Penguims spreadsheet niet opgebouwd volgens een vast rooster. Het is mogelijk gerelateerde cellen te groeperen in “objects”, en gerelateerde objects kunnen op hun beurt gegroepeerd worden in “groups”.

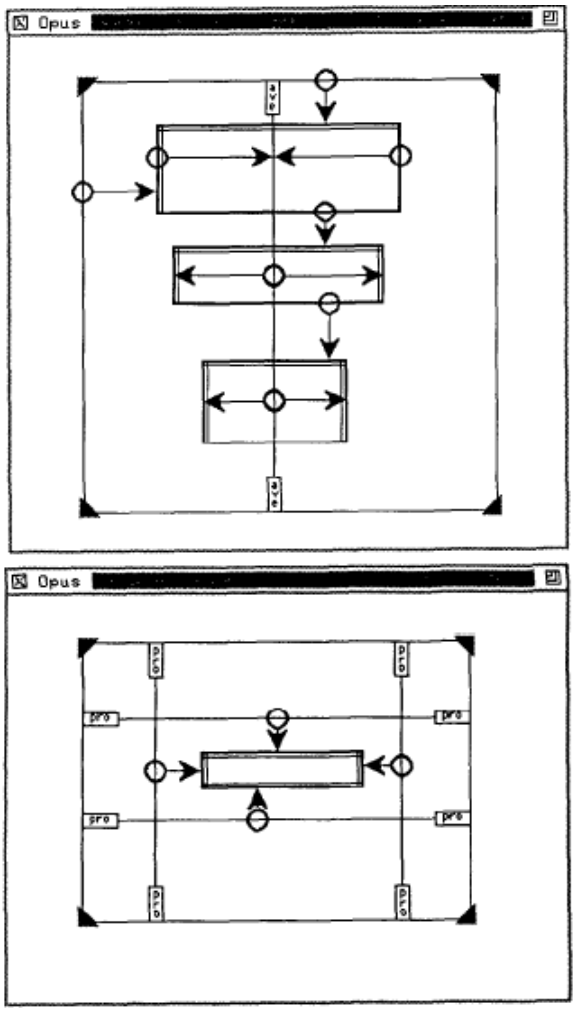
Een cel bestaat uit twee onderdelen: een waarde en een optionele vergelijking of formule. Deze vergelijking beschrijft hoe de waarde van de cel is samengesteld. Wanneer de waarde of de vergelijking verandert, zal ook de waarde van alle cellen die daardoor rechtstreeks of onrechtstreeks beïnvloed worden, worden aangepast. Deze vergelijkingen stellen one-way constraints voor. De constraints worden opgelost door local propagation technieken (zie *“4.6.2.4 Constraint solvers”*) en bepalen de zichtbaarheid, positie en andere kenmerken van elk object.

De cellen kunnen uiteindelijk worden verbonden met objecten van een grafische user interface. Deze objecten kunnen interactor objects (widgets zoals bijvoorbeeld een lijn) of application objects (application data) zijn. Elk object heeft een aantal parameters waar elke parameter een cel controleert of gecontroleerd wordt door de waarde van een andere cel [Huds93].



Figuur 35 - Penguims line interactor object, uit [Huds93]

OPUS kan gebruikt worden om een user interface op een grafische manier op te bouwen. OPUS kan hiervoor gebruik maken van vier basis bouwblokken namelijk frames, reference lines, constraints en interactor objects.



Figuur 36 - Twee OPUS UI. Frames worden voorgesteld door rechthoeken, referentie lijnen door lijnen en constraints door pijlen, uit [Huds90]

Frames zijn rechthoeken die gebruikt kunnen worden om objecten in de interface te groeperen. Het frame zelf wordt niet in de uiteindelijke user interface toegevoegd, maar wordt enkel gebruikt als een referentiepunt om relaties tussen objecten in de interface op te leggen. Van zodra objecten in een frame worden geplaatst, krijgen ze constraints opgelegd ten opzichte van de zijden van het frame. Er zijn twee soorten frames: open frames en hierarchical composition frames. Een open frame kan gebruikt worden om objecten te groeperen, een hierarchical composition frame wordt gebruikt om andere groepen in te plaatsen.

Reference lines worden gebruikt bij de uitlijning van objecten. Net zoals frames zijn ze niet zichtbaar in de uiteindelijke user interface, maar worden ze enkel gebruikt als referentiepunt. Er zijn verschillende types reference lines. Free reference lines zijn referentielijnen waarvan de positie wordt bepaald door één enkele constraints. Daarnaast zijn er ook min, max en average reference lines waarvan de positie wordt bepaald door de berekening van minima, maxima en gemiddelden. Ten laatste zijn er nog proportional reference lines die gebruikt kunnen worden om een proportionele afstand aan te houden. Reference lines kunnen verbonden worden met frames.

Constraints stellen vergelijkingen voor en worden op de tekening als pijlen voorgesteld. De pijl start vanuit een anchor of fixed point en loopt tot een object, waarbij een relatie wordt gecreëerd tussen een coördinaat van het (anchor) punt en een coördinaat van het object. Constraints die vertrekken vanuit een punt beginnen met een cirkel, constraints die vertrekken vanuit een vast anchor point starten met een gekleurde ruit. Wanneer op de constraint wordt geklikt, wordt de bijhorende vergelijking weergegeven en kan deze vervolgens worden gewijzigd.

Een *interactor object* is een voorstelling van een zichtbaar component in de uiteindelijke user interface. Er worden acht verschillende soorten componenten ondersteund: horizontale en verticale schuifbalken, rechthoeken, afbeeldingen, labels, text fields, push buttons en radio buttons. Elk interactor object heeft een bounding rectangle die de positie en grootte van het object bepalen. Aan elke zijde van de rechthoek kan een constraint worden gekoppeld. Wanneer de zijde “vrij” wordt gelaten, dan krijgt deze een beginwaarde toegewezen. Per interactor object kunnen ook nog een aantal andere kenmerken gedefinieerd worden. Dit gebeurt door middel van een property list die wordt opgeroepen van zodra op een interactor object wordt geklikt [Huds90].

5.5 Vergelijking tussen constraint-based systemen

		Mogelijkheden?	Constraints?	Constraint solving techniek?	Constraint visualisatie?
Drawing	**Sketchpad**	Objecten positioneren en selecteren met pen, buttons om constraints toe te passen	Geometric (multi-way)	Local propagation, relaxation	Symbool
	Briar	Augmented snap-dragging; alignment objects om snapping op uit te voeren; constraints worden enkel behouden, niet initieel gespecificeerd; constraint-based direct manipulation	Geometric: points-on-object, points-on-coincident	Differential methods	Points-on-object dmv ruit, points-on-coincident dmv vierkant, alignment object dmv stippellijn
	Unidraw / QOCA	Undo / redo mogelijkheden; opslaan en inladen van system state; constrained direct manipulation	Linear (in)equalities (multi-way)	Direct numeric: QOCA	Spacer widgets dmv pijlen, alignment constraints dmv stippellijnen
	Chimera	Snapshot-mechanisme	Geometric: zie tabel 3	Numeric	Geen informatie
	Pegasus	Interactive beautification, predictive drawing	Geometric: connection, alignment parallellism, perpendicularity, congruence, symmetry interval equality (multi-way)	Numerical equality solver (modification of CLP(R))	Stippellijnen

		Mogelijkheden?	Constraints?	Constraint solving techniek?	Constraint visualisatie?
Window layout	**Scheme window manager**	Augmented snap-dragging; zelf constraints samenstellen dmv PBD	Linear (in)equalities: zie tabel 4 (multi-way)	Direct numeric: Cassowary	Pijlen, stippellijnen (zie figuur 33)
UI toolkit	**Amulet**	Constraints koppelen aan slots; object met gesture recognition ondersteuning; web constraint solver en animation constraint solver	Formula constraints, indirect constraints (one-way)	Local propagation	Geen informatie
	Bramble	Controls manipuleren (cfr. bureaulamp); notie van tijd	Geometric	Differential methods	Geen informatie
	OPUS / Penguims	Geavanceerde vorm van spreadsheets	Geometric (one-way)	Local propagation	Pijlen met cirkel (vanuit punt), pijlen met gekleurde ruit (vanuit anchor)

Tabel 5 - Vergelijkend overzicht van constraint-based applicaties

5.6 Conclusie

In dit hoofdstuk werden een aantal constraint-based systemen besproken. Hierbij werd er vooral gekeken naar de manier waarop deze systemen constraints specificeren, van welke constraint solving techniek gebruikt wordt gemaakt en hoe de constraints gevisualiseerd worden. Hoewel deze tools ontwikkeld zijn voor een bepaald applicatiedomein, worden constraints binnen éénzelfde domein op verschillende manieren gedefinieerd en gebruikt.

Vooral de manier waarop constraints worden gebruikt in Unidraw en de Scheme window manager lijkt interessant. In Unidraw kunnen een aantal layout commando's op user interface elementen worden toegepast die ook kunnen worden terugvonden in de toolkits die in het vorige hoofdstuk werden besproken, zoals het uitlijnen van elementen, het toevoegen van spacer widgets, het onderbrengen van elementen in een container,…. Diezelfde commando's zullen ook geïntegreerd worden in het prototype. Ook in de Scheme window manager kan gebruik worden gemaakt van een aantal interessante constraints zoals alignment, separation en anchors. Hoewel deze constraints worden toegepast op windows, zijn ze ook bruikbaar voor user interface elementen. Beide tools maken gebruik van een direct numeric solver: Unidraw werd geïntegreerd met de QOCA constraint solving toolkit en Scwm maakt gebruik van de Cassowary constraint solving toolkit.

Een ander interessant element dat in Briar en Unidraw werd gebruikt is constrained direct manipulation. Deze vorm van directe manipulatie zorgt ervoor dat tijdens de verplaatsing van objecten aan de gedefinieerde relaties blijft voldaan.

Op basis van de systemen die in dit hoofdstuk werden besproken, kunnen de volgende doelstellingen geformuleerd worden voor het te ontwikkelen prototype:

> "Doelstelling 8: Het prototype moet de layout van de toekomstge user interface definiëren in de vorm van constraints en deze moeten worden opgelost door een constraint solver.
>
> "Doelstelling 9: In het prototype moet gebruik worden gemaakt van een vorm van constrained direct manipulation."

Hoofdstuk 6

Gestures

6.1 Overview

In de loop van deze literatuurstudie zijn we het begrip “gesture” al verschillende keren tegengekomen. In hoofdstuk 3 werden een aantal informele sketching tools besproken die het gebruik van gestures ondersteunen. In deze tools worden gestures enerzijds gebruikt om nieuwe user interface elementen aan de scène toe te voegen en anderzijds om commando’s zoals delete, copy,… op de user interface elementen uit te voeren. Ook in het vorige hoofdstuk werd het begrip “gesture” aangehaald bij de bespreking van de constraint-based toolkit Amulet. In deze toolkit is één van de voorgedefinieerde objecten uitgerust met gesture recognition waardoor het voor de gebruiker mogelijk wordt zelfgedefinieerde gestures toe te voegen en deze aan een bepaald commando te koppelen. In dit laatste hoofdstuk van deze literatuurstudie wordt een verklaring gegeven aan het begrip “gesture”. Er wordt eveneens gekeken naar het gebruik en de werking ervan. Daarnaast wordt een overzicht gegeven van een aantal gesture recognition algoritmen.

6.2 Gestures-based interactie

In Longman Dictionary wordt de volgende verklaring aan het woord gesture gegeven:

> *“a movement of part of your body, especially your hands or head, to show what you mean or how you feel.”*

In het dagelijkse leven maken mensen vaak gebruik van gebaren om met elkaar te communiceren. Sinds de ontwikkeling van invoerapparaten zoals de muis en stylus, is er veel onderzoek gedaan om deze vorm van communicatie ook te gebruiken in de interactie tussen mens en computer. In [Kara05] wordt er een overzicht gegeven van de interactietechnieken die de voorbije veertig jaar werden ontwikkeld. In het kort worden hier de belangrijkste applicatiedomeinen opgesomd. Voor een gedetailleerd overzicht wordt echter verwezen naar [Kara05].

Een eerste domein is *virtual en augmented reality*. Hier worden gestures gebruikt om de bewegingen en expressies van avatars[6] aan te sturen. De gestures kunnen op een tablet device worden uitgetekend waarop ze vervolgens worden omgezet in verschillende lichaamsbewegingen en gelaatsuitdrukkingen. Indien gebruik wordt gemaakt van sensoren kunnen de bewegingen van de gebruiker rechtstreeks worden toegepast op de avatar.

Gestures kunnen in dit domein eveneens gebruikt worden voor navigatie en manipulatie van objecten in een 3D scène. Ook hier wordt meestal gebruik gemaakt van sensoren om de locatie en houding van het menselijk lichaam te volgen. Robotica applicaties maken ook gebruik van gestures. Enerzijds voor het controleren van de snelheid en de richting van de bewegingen van de robots, anderzijds voor het aansturen van en de controle over hand- en armbewegingen.

Een tweede domein waar gebruik wordt gemaakt van gesture-based interactie zijn *ubiquitous computing en smart environments*. Hier voert de gebruiker voornamelijk hand en arm gestures uit om controle te krijgen over apparaten en displays die zich op een afstand bevinden. Voorbeelden hiervan zijn de controle en bediening van lichten in een kamer, de controle van muziek en andere entertainment interfaces,…. *Tangible computing*, waarbij de gebruiker een fysieke presentatie van digitale user interface objecten kan gaan manipuleren, valt ook in dit domein.

Ook in *games* worden gestures gebruikt. Enerzijds om input over de oriëntatie en positie van objecten aan het spel door te geven, anderzijds voor de controle en navigatie van de avatar.

In *pervasive en mobile interfaces* hebben gestures het doel de gebruiker een “eyes-free style of interaction” te bezorgen. Hiermee wordt getracht de aandacht van de interactietechniek af te leiden zodat de gebruiker zich kan concentreren op andere taken. Een voorbeeld hiervan is het gebruik van een GPS systeem op PDA in de auto. Dit systeem kan worden aangestuurd door middel van vinger gestures. De gebruiker moet dus

[6] Een virtueel object dat gebruikt wordt om een deelnemer of een fysisch object in de virtuele wereld voor te stellen

minder aandacht besteden aan de bediening van het apparaat en kan zich volledig concentreren op het autorijden.

Een laatste domein waar gebruik wordt gemaakt van gesture-based interactie zijn *desktop en tablet PC applicaties*. Hier bieden gestures een alternatief voor muis- en keyboardinteracties en maken op die manier een meer natuurlijke vorm van interactie mogelijk. Gestures worden in tal van systemen gebruikt onder andere in grafische programma's, CSCW systemen en multimodale interfaces.

In het kader van deze thesis zijn we vooral geïnteresseerd in toepassingen in dit laatste domein. De gestures waarheen in deze thesis wordt verwezen, hebben dus betrekking op de handbewegingen die door de gebruiker via een muis of stylus aan de computer worden doorgegeven.

6.3 Structuur gesture

Een gesture is opgebouwd uit één of meerdere strokes. Een stroke is een onafgebroken beweging die met een stylus, muis of een ander invoerinstrument wordt uitgevoerd. Een stroke bestaat dus uit een opeenvolging van punten en wordt meestal geïnstantieerd en beëindigd door een actie van de gebruiker zoals het neerzetten en opheffen van de stylus of het indrukken en loslaten van een muistoets. Wanneer een gesture uit één enkele stroke bestaat, spreekt men van een *single-stroke* gesture. Een gesture dat uit meerdere strokes bestaat, is een *multi-stroke* gesture [Rubi91b].

Een gesture kan dus eigenlijk worden voorgesteld als een pad van verschillende punten over een bepaald tijdsinterval. Op die manier kan "pointing" worden gezien als de meest simpele gesture-vorm waarbij één enkele positie op een bepaald tijdstip wordt gedefinieerd. Gestures waarbij één enkel punt in een bepaald tijdsinterval wordt gedefinieerd, worden *single-path gestures* genoemd. Wanneer gebruik wordt gemaakt van een invoerapparaat dat in staat is meer dan één positie gelijktijdig aan te geven, dan spreekt men van *multi-path gestures*. Voorbeelden van invoerapparaten die het mogelijk maken multi-path gestures te definiëren zijn Sensor Frames, multi-finger touch pads en DataGloves [Rubi91b].

6.4 Gesture-based systemen

Een gesture-based systeem is een applicatie die is uitgerust met een gesture-based interface. Deze interface laat de gebruiker toe commando's op te roepen en uit te voeren door middel van gestures. Hoewel in de vorige paragraaf werd aangegeven dat pointing

de meest simpele gesture vorm is, worden interfaces die enkel uitgerust zijn met pointing als interactietechniek niet beschouwd als gesture-based interfaces. Interfaces die dus enkel gebruik maken van click-and-drag interactie behoren niet tot de groep van gesture-based interfaces [Rubi91b].

Een belangrijk element in een gesture-based interface is de *gesture recognizer* of *gesture classifier*. Deze heeft als taak het getekende gesture te herkennen en in de juiste gesture klasse te classificeren zodat vervolgens het juiste commando kan worden opgeroepen. In de volgende alinea's worden een aantal gesture recognition algoritmen besproken waarmee tijdens de uitvoeren van deze literatuurstudie in contact werd gekomen.

6.5 Gesture recognition algoritmen

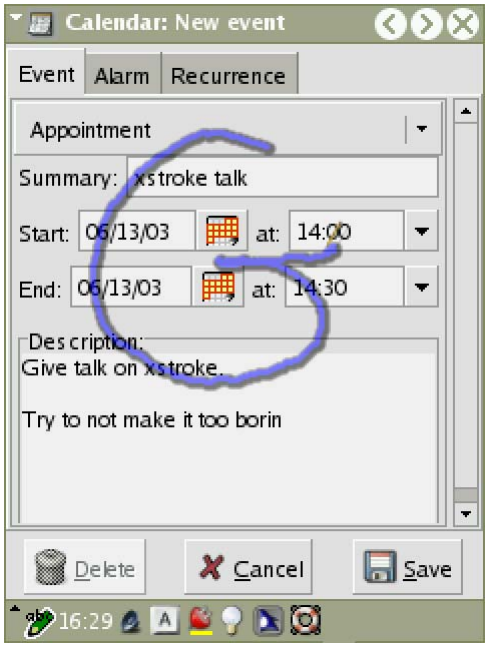
6.5.1 Xstroke

Een eerste gesture recognition algoritme dat wordt besproken is *Xstroke* [Wort03], een gesture recognition systeem voor het X Window System, de standaard grafische user interface van UNIX systemen. Via Xstroke kunnen gestures worden toegevoegd aan de desktop omgeving van UNIX en UNIX-achtige besturingssystemen.

Het Xstroke systeem is in de eerste plaats in staat full-screen recognition uit te voeren. Hiermee kan de gebruiker overal op de desktop, ongeacht van de onderliggende applicatie, gestures op het scherm uitvoeren. Het getekende gesture wordt geïnterpreteerd en in doorzichtige inkt bovenop de geopende applicatie getoond. Een voorbeeld hiervan wordt weergegeven in figuur 37. Het systeem is in staat geroteerde versies van de gestures te herkennen. Eveneens kan het recognition systeem worden uitgebreid met meer gesofisticeerde recognition engines.

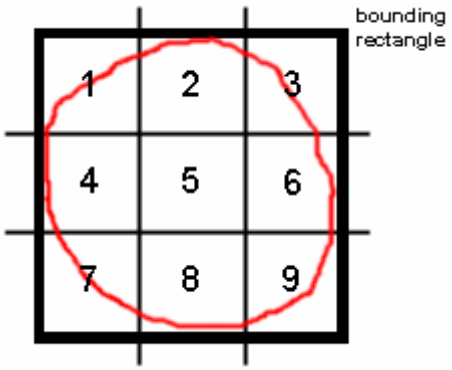
Ten tweede beschikt het Xstroke systeem over een gesture alfabet dat gebruikt kan worden in plaats van het toetsenbord. Aan elke gesture letter in het alfabet is een actie gekoppeld. De gebruiker kan andere acties aan de letters koppelen indien hij dat wenst.

Het recognition algoritme maakt gebruik van een grid feature geïnspireerd op het algoritme gebruikt in libstroke [Will97]. Het grid bestaat uit 9 cellen, verdeeld over 3 rijen en 3 kolommen, die genummerd zijn van 1 tot en met 9. Van zodra de gebruiker een gesture tekent, wordt de bounding box van dit gesture berekend. Het grid wordt uitgelijnd met de X- en Y-as van deze bounding box en gecentreerd op het gesture geplaatst.



Figuur 37 - Xstroke full recognition, uit [Wort03]

De muisposities van de gebruiker worden tijdens het tekenen bijgehouden. Van zodra het grid correct op het gesture is geplaatst, wordt voor elke muispositie berekend in welke cel het zich bevindt. De code van de cel wordt voor elke muispositie bijgehouden. Wanneer de gebruiker een cirkel gesture tekent, zal dit bijvoorbeeld in de volgende sequentie resulteren: 3333…2222…1111…4444…7777…8888…9999…6666. De codes worden gefilterd zodat volgende reeks overblijft: 32147896. Deze reeks kan tenslotte getest worden om te controleren welk gesture werd uitgevoerd.



Figuur 38 - Grid feature in Xstroke

6.5.2 Rubine

Een tweede recognition algoritme dat wordt besproken is *Rubine's algoritme* [Rubi91b]. Het algoritme maakt deel uit van de GRANDMA toolkit, een toolkit die ontworpen is voor het toevoegen van gestures aan direct manipulation interfaces. Het

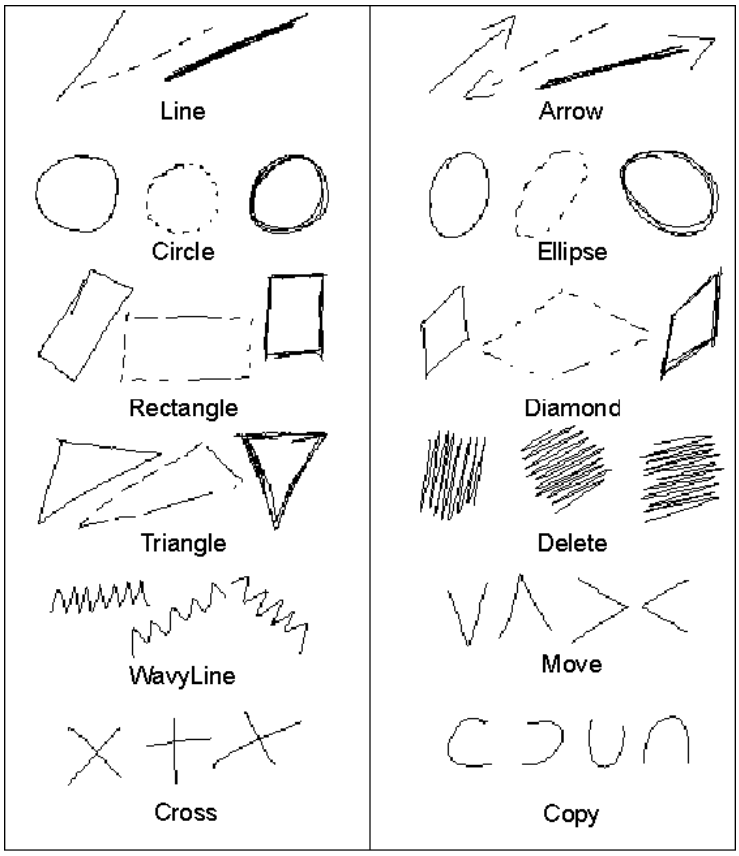
algoritme is in staat single-stroke gestures te herkennen die door de gebruiker gedefinieerd worden aan de hand van een aantal gesture voorbeelden. De recognition engines in DENIM en SILK, en het AGATE recognition algoritme in Amulet zijn gebaseerd op dit algoritme.

Het herkenningsproces van het algoritme is gebaseerd op twee principes: enerzijds *classificatie* en anderzijds *training*. Het algoritme maakt gebruik van een aantal gesture klassen waarbij elke klasse bestaat uit een aantal voorbeeld gestures. Deze voorbeelden kunnen door de gebruiker gedefinieerd worden. Het doel van het algoritme is een gegeven gesture te *classificeren* in de juiste klasse. Deze classificatie gebeurt op basis van een aantal kenmerken die van het input gesture worden berekend en tenslotte worden vergeleken met de kenmerken van de voorbeeld gestures uit elke gesture klasse. Een gesture klasse is daarvoor uitgerust met een evaluatiefunctie. De evaluatiefuncties berekenen voor welke classificatie de evaluatie van de kenmerken het grootste is. Elk kenmerk wordt daarom geassocieerd met een bepaald gewicht. Het bepalen van deze gewichten gebeurt door het *training* principe.

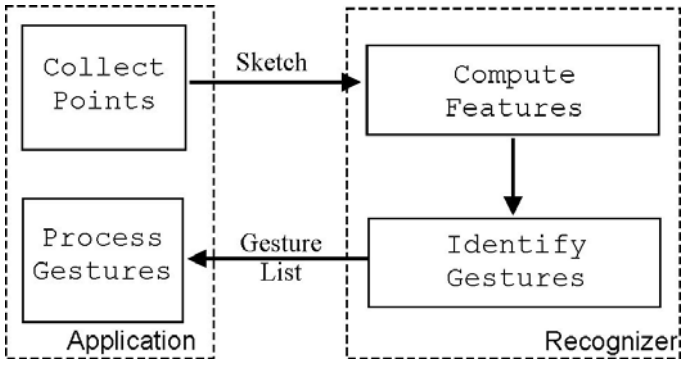
6.5.3 CALI library

Het laatste gesture recongition algoritme dat wordt besproken is *CALI* [Fons02]. Deze library wordt in JavaSketchIt en SketchiXML gebruikt voor het herkennen van commando's en user interface elementen. CALI is een software library die in staat is multi-stroke gestures en vormen te herkennen. Met multi-stroke gestures worden gestures bedoelt die uit meerdere penbewegingen bestaan zoals een pijl of een kruis. In figuur 39 wordt een overzicht gegeven van de verschillende vormen die door de CALI library ondersteund worden. Alle vormen hebben een gestippelde of vette variant die eveneens herkend kan worden. Elke vorm en zijn varianten kunnen ook gedraaid worden onder een willekeurige hoek.

Het recognition proces van CALI verloopt in drie verschillende stappen. In de eerste stap ontvangt de recognizer de scribbles van de applicatie. Van elke scribble worden een aantal geometrische kenmerken berekend. In de tweede stap worden een aantal filters op deze kenmerken toegepast om uiteindelijk de correcte vorm te identificeren. Tijdens de derde stap worden ambiguïteiten weggewerkt door gebruik te maken van fuzzy logic. Aan elke vorm wordt tenslotte een herkenningspercentage toegevoegd. De vormen en hun bijhorende herkenningspercentage worden uiteindelijk gerangschikt in een lijst van groot naar klein. Deze "gesture list" wordt terug doorgegeven naar de applicatie. Figuur 40 geeft dit proces weer.



Figuur 39 - Overzicht van vormen die herkend worden door CALI, uit [Fons02]



Figuur 40 - CALI herkenningsproces, uit [Fons02]

6.6 Conclusie

Naast de structuur en het gebruik van gestures, werden in dit hoofdstuk een aantal gesture recognition algoritmen behandeld. Het eerste algoritme dat besproken werd, was het Xstroke algoritme. Dit algoritme maakt gebruik van een grid dat over het gesture wordt geplaatst om vervolgens te kijken welk pad het gesture in het grid volgt. Het gevolgde pad kan dan vergeleken worden met een aantal voorgedefinieerde sequenties om vervolgens het juiste gesture te bepalen. De werking van het algoritme is simpel, maar

heeft als nadeel dat het niet geschikt is voor de herkenning van multi-stroke gestures. Daarnaast is het ook moeilijk om een onderscheid te maken tussen een cirkel- en rechthoekbeweging. Beide bewegingen volgens immers dezelfde sequentie binnen het grid. Omwille van deze redenen werd het algoritme als niet geschikt bevonden voor het te ontwikkelen prototype.

Een tweede gesture recognition algoritme dat werd besproken was Rubine's algoritme. Het voordeel van dit algoritme is dat het gebruikers in staat stelt zelf gestures te gaan definiëren. Het nadeel is wel dat het algoritme enkel single-stroke gestures kan herkennen. Daarnaast is het voor een goede werking van het algoritme ook aangeraden dat de gebruiker van elk gesture een aantal verschillende voorbeelden definieert.

Het laatste algoritme dat werd besproken was de CALI library. Deze library is in staat om multi-stroke gestures te herkennen met een recognition rate van 97%. Het nadeel van deze library is dat de gebruiker niet over de mogelijkheid beschikt zelf gestures te definiëren. Er dient dus gebruik te worden gemaakt van de voorgedefinieerde verzameling gestures die momenteel door de library herkend worden.

Na deze verzameling onderzocht te hebben, kan er worden vastgesteld dat deze voldoende ondersteuning biedt voor de layout commando's die in het prototype gekoppeld zullen worden. Zo lijkt de pijl gesture ook ideaal voor het specificeren van adjacency constraints. Daarnaast worden de widgets ook vlot herkend. De volgende doelstelling kan dus worden toegevoegd aan de lijst met reeds gedefinieerde doelstellingen:

> "Doelstelling 10: Het prototype maakt gebruik van de CALI library voor de herkenning van vormen en gestures."

DEEL 3 IMPLEMENTATIE

Hoofdstuk 7

Software Architectuur

7.1 Overview

Zoals reeds werd vermeld in de inleiding heeft de uitvoering van deze thesis twee concrete doelen voor ogen: enerzijds via een literatuurstudie een duidelijk beeld te geven van de huidige stand van zaken wat betreft sketch-based user interfaces en de mogelijkheden die op gebied van layout en constraint-based layout systemen bestaan. Anderzijds de ontwikkeling van een prototype waarin de koppeling tussen layout commando's en constraints door middel van gestures wordt getest. In dit laatste hoofdstuk zal de architectuur en ontwikkeling van het geïmplementeerde prototype worden besproken. Tijdens de literatuurstudie werden een aantal doelstellingen vooropgesteld waaraan het te ontwikkelen prototype moest voldoen. Er wordt een overzicht gegeven van deze doelstellingen waarbij per doel wordt besproken hoe dit werd verwezenlijkt.

7.2 Overzicht doelstellingen

De volgende doelstellingen werden tijdens de literatuurstudie geformuleerd:

#	Hoofdstuk	Doelstelling
1	2	Het prototype moet een natuurlijke vorm van interactie ondersteunen. De gebruiker moet in staat zijn user interface elementen te gaan schetsen met behulp van een stylus en elektronisch pad.
2	3	Het prototype moet pen-based interactie technieken ondersteunen.

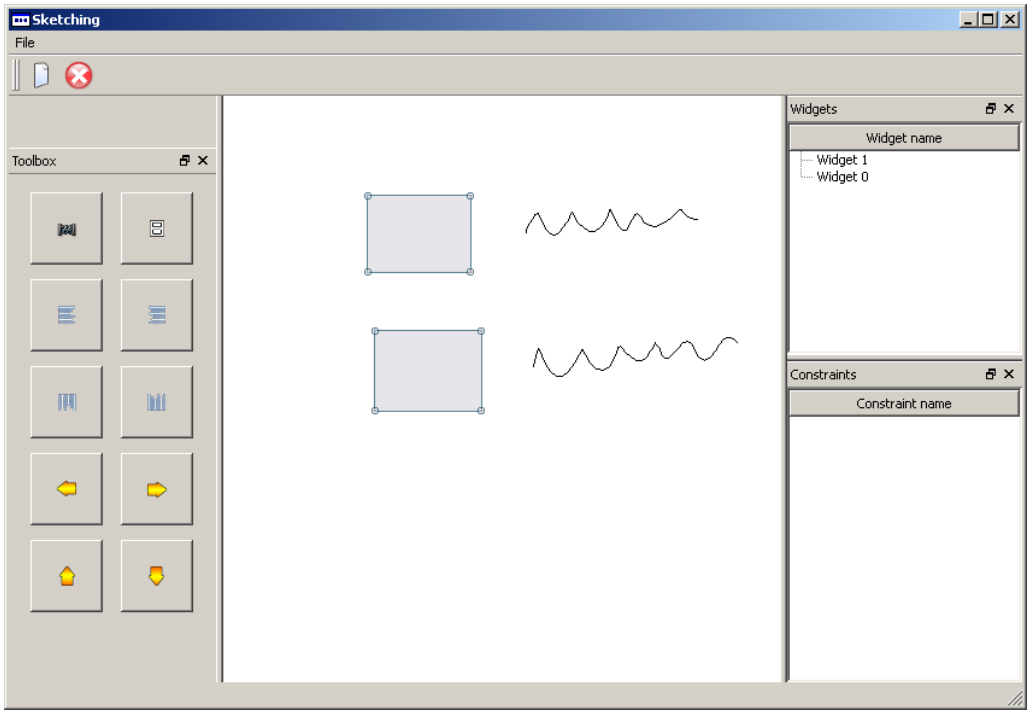
#	Hoofdstuk	Doelstelling
3	3	Het prototype moet in staat zijn geschetste user interface elementen te herkennen.
4	3	Het prototype moet de gebruiker in staat stellen elementen te laten groeperen in containers.
5	4	Het prototype moet de gebruiker in staat stellen de grootte en positie van elementen te laten aanpassen.
6	4	Het prototype moet het voor de gebruiker mogelijk maken enkele simpele layout managers toe te passen.
7	4	Het prototype moet de gebruiker in staat stellen elementen toe te voegen die het mogelijk maken een bepaalde ruimte in een layout te reserveren.
8	5	Het prototype moet de layout van de toekomstge user interface definiëren in de vorm van constraints en deze moeten worden opgelost door een constraint solver.
9	5	In het prototype moet gebruik worden gemaakt van een vorm van constrained direct manipulation
10	6	Het prototype maakt gebruik van de CALI library voor de herkenning van vormen en gestures.

Tabel 6 - Overzicht prototype doelstellingen

7.3 Implementatie doelstellingen

7.3.1 Doelstelling 1 en 2

In hoofdstuk 3 van deze thesis werden zes verschillende sketching tools besproken. Wanneer er wordt gekeken naar de manier waarop in deze tools een user interface geconstrueerd kan worden, kunnen deze tools in twee groepen worden verdeeld. Enerzijds zijn er SILK, DENIM, SketchiXML en JavaSketchIt waar de designer door middel van gestures en scribbles de toekomstige user interface kan gaan schetsen. Anderzijds zijn er CanonSketch en DialogSketch, waar de gebruiker de user interface kan opbouwen door middel van een aantal voorgedefinieerde user interface elementen vanuit een palet naar een canvas te slepen. Vooral de eerste groep trekt hier onze interesse. Door gebruik te maken van scribbles en gestures kan de gebruiker immers op een heel natuurlijke en flexibele manier een user interface gaan opbouwen. Daarom werd gekozen deze vorm van interactie in het prototype te integreren.



Figuur 41 - Prototype

In figuur 41 wordt een screen shot van het ontwikkelde prototype weergegeven. Net zoals de besproken tools is het ontwikkelde prototype ook opgebouwd uit een canvas waarop de gebruiker kan gaan schetsen. Door middel van gestures kan de gebruiker user interface elementen aan het canvas toevoegen. Eveneens kunnen gestures gebruikt worden om layout commando's op de user interface elementen toe te passen. De werking van gestures wordt verder besproken in *"7.3.4 Doelstelling 3* en 10*"*.

Doordat het opbouwen van de user interface en het specificeren van layout eigenschappen door middel van gestures gebeurt, kan er best gebruik worden gemaakt van een stylus en elektronisch pad om de interface te schetsen. Tijdens de implementatie van het prototype werd er gebruik gemaakt van een WACOM elektronisch pad met bijhorende stylus.

Figuur 42 - WACOM Volito2 Tablet [7]

[7] WACOM website: http://www.wacomshop.be/products.html

7.3.2 Doelstelling 4 tot en met 7

Zoals reeds werd aangegeven in het begin van deze thesis, is het doel van de ontwikkeling van het prototype een koppeling tussen layout commando's en gestures te testen. In hoofdstuk 4 werden daarom een aantal veelgebruikte grafische toolkits van naderbij bekeken om zo te achterhalen over welke layout mogelijkheden deze beschikken. Na deze toolkits grondig bestudeerd te hebben, kon worden vastgesteld dat ze een aantal algemene, gelijkaardige layout eigenschappen en commando's delen, zoals bijvoorbeeld horizontale en verticale layout managers, uitlijning van elementen,…. Deze basis layout commando's werden ook in het prototype geïntegreerd. Het gaat om de volgende commando's:

- aanpassen van de positie van elementen
- aanpassen van de grootte van elementen
- definiëren van de positie van elementen ten opzichte van andere elementen (adjacency)
- elementen horizontaal of verticaal ten opzichte van elkaar uitlijnen (alignment)
- componenten groeperen in een container
- toepassen van verticale en horizontale layout managers
- toevoegen van elementen die in staat zijn een bepaalde ruimte in de layout te reserveren

Hoewel het adjacency-commando niet teruggevonden kon worden in één van de toolkits, leek dit toch een handig commando. Daarnaast werden ook nog onderstaande commando's toegevoegd om de constructie van de user interface te vergemakkelijken:

- selectie van elementen
- elementen verwijderen

7.3.3 Doelstelling 8 en 9

Om de layout voor te stellen door middel van constraints is er gekozen om gebruik te maken van de Cassowary constraint solving toolkit [Badr98a]. Deze keuze is gebaseerd op een aantal redenen die hier verder besproken zullen worden.

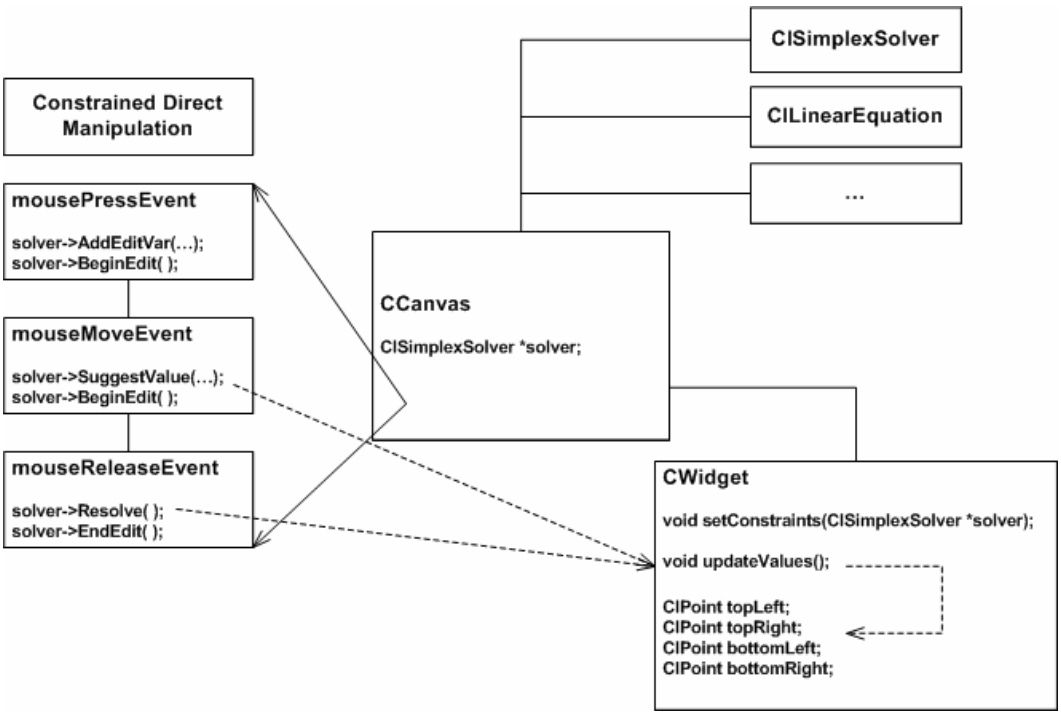
Ten eerste is Cassowary in staat om zowel lineaire gelijkheden als ongelijkheden op te lossen. Beide soorten constraints ontstaan bij het definiëren van een layout en andere geometrische relaties. Het is dus belangrijk dat het algoritme overweg kan met deze soorten constraints [Badr98a].

Ten tweede is het in Cassowary mogelijk om meerdere constraints gelijktijdig af te handelen. Het algoritme kan dus overweg met cycli in de constraint graaf [Badr98a].

Een derde reden is het feit dat het Cassowary algoritme incrementeel is waardoor het bruikbaar wordt voor interactieve user interface applicaties. In interactieve applicaties worden op korte tijd vele kleine aanpassingen gedaan zoals bijvoorbeeld het bewegen van een object met de muis. Het constraint algoritme moet dus in staat zijn om snel gelijkaardige problemen herhaaldelijk op te lossen. In Cassowary wordt die incrementaliteit op twee manieren bereikt. Wanneer een object wordt bewogen met de muis, dan wordt dit voorgesteld door middel van een one-way constraint. Anderzijds wordt er telkens zoveel mogelijk van de vorige oplossing hergebruikt [Badr98a].

Een laatste reden voor deze keuze is de mogelijkheid om in Cassowary een gewicht aan constraints toe te kennen. Cassowary maakt gebruik van een constraint hiërarchie, waarin elke constraint een bepaalde “strength” krijgt toegekend. Op die manier kunnen constraints “required” gemaakt worden waardoor kan worden aangegeven dat aan deze constraint te allen tijde moet worden voldaan. Constraints met een hogere strength domineren zwakkere constraints [Badr98a].

Onderstaande figuur geeft een overzicht van hoe de Cassowary toolkit in het prototype wordt gebruikt.

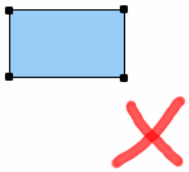
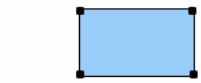
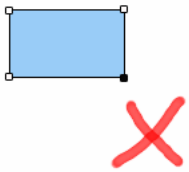
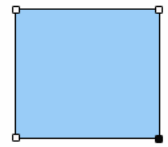
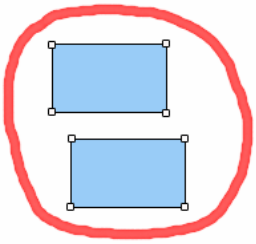
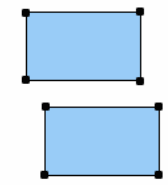


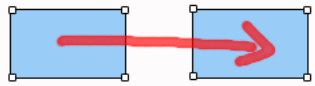
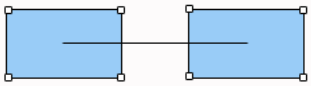
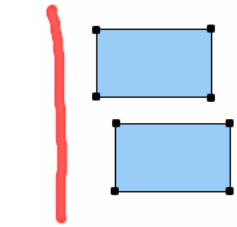
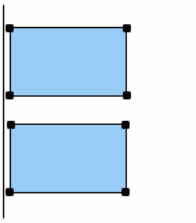
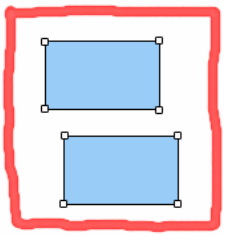
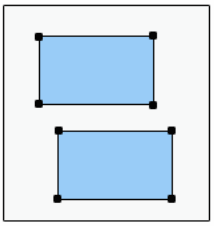
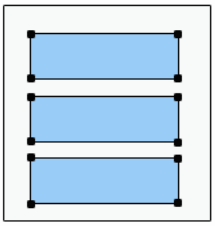
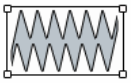
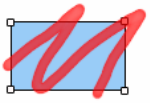
Figuur 43 - Koppeling Cassowary

In het canvas wordt een instantie naar ClSimplexSolver aangemaakt, de hoofdklasse van de Cassowary toolkit. Elk user interface element bevat vier Cassowary variabelen van het type ClPoint, die de hoekpunten van de bounding box van het element voorstellen. Wanneer de gebruiker een widget of een hoekpunt van een widget wil verslepen, zullen deze vier hoekpunten bij de constraint solver geregistreerd worden als edit variabelen. Edit variabelen werden speciaal aan de toolkit toegevoegd om directe manipulatie te ondersteunen. Ze worden gebruikt om aan te geven dat deze constraints snel moeten worden opgelost. Tijdens het verslepen van de elementen worden de gewenste waarden (huidige muispositie) van de variabelen doorgegeven aan de solver door middel van de SuggestValue functie. Daarna wordt de Resolve functie aangeroepen die een nieuwe oplossing van het constraint systeem zal berekenen die zo dicht mogelijk bij de oude oplossing en de gewenste waarden van de edit variabelen ligt [Marr02].

7.3.4 Doelstelling 3 en 10

In punt *7.3.2* werd reeds een opsomming gegeven van de layout commando's die het prototype dient te ondersteunen. Tabel 7 geeft de overeenkomstige gestures weer die aan deze commando's gekoppeld werden. In deze representatie stellen de vierkanten user interface elementen voor. Wanneer het element geselecteerd is, worden de hoekpunten ervan zwart gekleurd. De gesture die de gebruiker dient uit te voeren, wordt weergegeven door middel van een dikke rode lijn.

Commando	**Gesture**	**Resultaat gesture**
Positie van elementen aanpassen		
Grootte van elementen aanpassen		
Elementen selecteren		

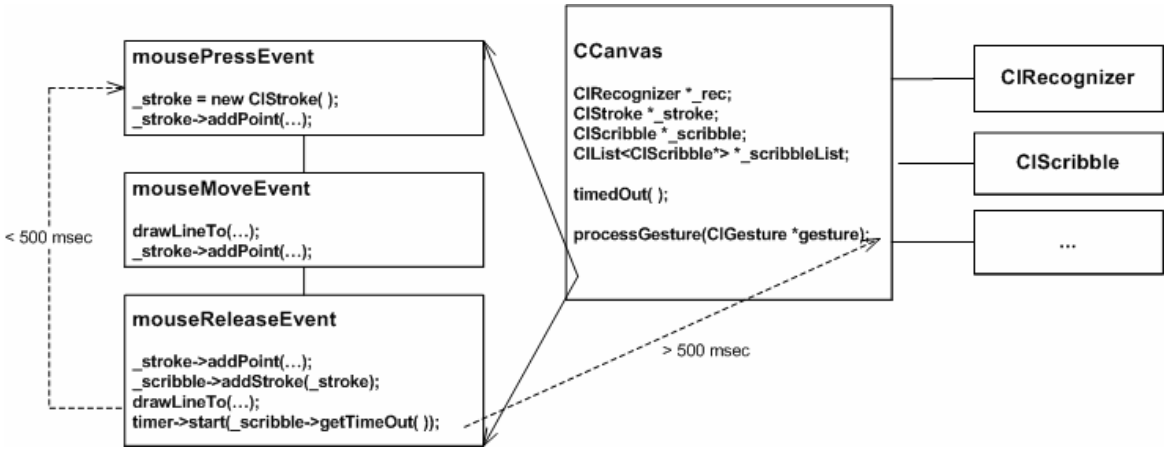
Commando	Gesture	Resultaat gesture
Adjacency (left-of)		
Alignment (left)		
Elementen groeperen in een container		
Layout manager toevoegen (verticaal)		
Spacer element toevoegen		
Element verwijderen		

Tabel 7 - Overzicht koppeling tussen commando's en gestures

Er werd gekozen om deze gestures in het prototype te implementeren door middel van de CALI library. Deze keuze is gebaseerd op twee redenen. Enerzijds werd er voor deze library gekozen omdat het recognition algoritme in staat is gestures te herkennen die uit meerdere strokes zijn opgebouwd zoals bijvoorbeeld een pijl of een kruis. Een tweede reden voor deze keuze is het feit dat deze library een hoge recognition rate haalt, namelijk een rate van 97 procent.

Zoals reeds besproken werd in sectie *"6.5.3 CALI library"* bestaat het herkenningsproces uit drie verschillende stappen: het verzamelen van geometrische constraints van scribbles, het identificeren van de juiste vorm en als laatste het wegwerken van eventuele ambiguïteiten.

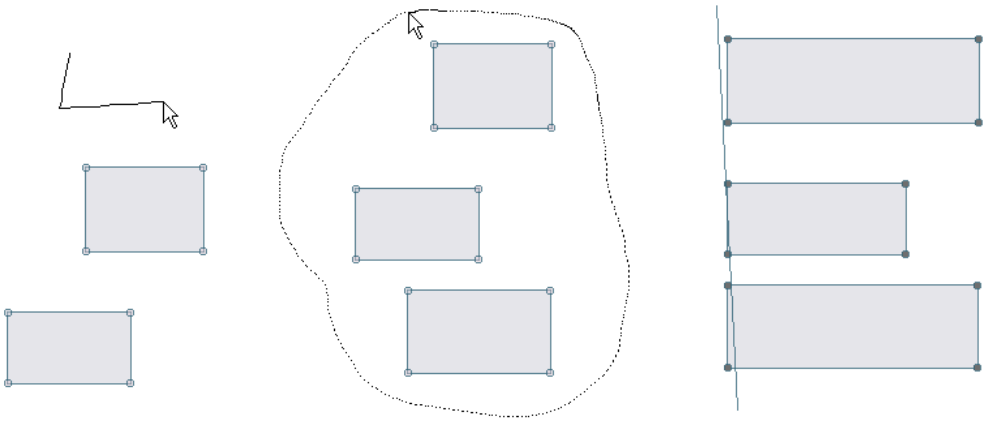
Allereerst worden dus de strokes verzameld die de gebruiker tekent. Bij het verzamelen van een stroke wordt telkens een timer gestart. De waarde van deze timer staat momenteel ingesteld op 500 msec. Wanneer de gebruiker binnen het verlopen van deze timer een tweede stroke tekent, zal deze worden toegevoegd aan dezelfde scribble. Op die manier wordt het mogelijk multi-stroke gestures samen te stellen. Van zodra de timer afloopt, zullen er geen strokes meer aan de scribble worden toegevoegd en wordt de scribble doorgegeven aan de recognizer. Vervolgens zullen een aantal geometrische kenmerken van de scribble worden berekend. Deze kenmerken worden gebruikt om de scribble te classificeren als een bepaalde vorm. Wanneer het algoritme er niet in slaagt één enkele vorm te herkennen, wordt een lijst teruggeven met daarin mogelijke kandidaten en een bijhorend herkenningspercentage.



Figuur 44 - Koppeling CALI

7.4 Werking prototype

Voor de specificatie van layout eigenschappen op user interface elementen wordt steeds dezelfde techniek gehanteerd. Eerst dient de gebruiker de UI elementen op het canvas te creëren. Dat doet hij door een rechthoek te tekenen op de plaats waar hij het element wil toevoegen. Vervolgens dient de gebruiker de elementen, waarop hij het layout commando wil gaan uitvoeren, te selecteren. Dat kan hij doen door middel van de rechtermuisknop ingedrukt te houden, en een cirkel-beweging rond de elementen te maken. Van zodra de gebruiker de rechtermuisknop indrukt, zal een lasso-tool geactiveerd worden die het toelaat meerdere elementen te selecteren. Zodra de elementen geselecteerd zijn, worden de hoekpunten ervan gekleurd. De gebruiker kan dan het gewenste commando gaan toepassen. In figuur 45 wordt dit proces weergegeven.



Figuur 45 - Werking prototype: links toevoegen van UI elementen, midden selecteren dmv lasso, rechts constraint toepassen

Hoofdstuk 8

Conclusie

8.1 Algemene conclusie

Tijdens het user interface ontwikkelingsproces maken designers gebruik van sketches om enerzijds snel nieuwe design ideeën uit te testen en anderzijds te communiceren met de overige stakeholders. Sketch-based user interfaces ondersteunen de designer tijdens dit proces door hem de mogelijkheid te geven de toekomstige user interface op een informele manier te gaan creëren en deze schets uit te rusten met tal van interactieve mogelijkheden. Hoewel niet alle besproken sketch-based user interface design tools uitgerust zijn met de mogelijkheid om code te genereren, is het hoofddoel van deze tools wel ervoor te zorgen dat de tijd en het werk, die de designer in de ontwikkeling van het prototype investeert, niet verloren gaan. Ze zorgen ervoor dat het ontwikkelde prototype of een gedeelte ervan op één of andere manier in een later stadium van het design proces hergebruikt kan worden.

In hoofdstuk 3 werden verschillende sketch-based user interfaces besproken. Na de werking van deze tools te analyseren, kon er besloten worden dat deze tools over weinig of zelfs geen mogelijkheden beschikken voor de specificatie van layout eigenschappen. In het beste geval is de sketch-based UI design tool in staat componenten onder te brengen in een container of widgets te rangschikken in een gridlayout. Wanneer in hoofdstuk 4 de layout mogelijkheden van een aantal user interface toolkits werden bekeken, kon thans besloten worden dat deze uitgerust zijn met tal van commando’s zoals het toepassen van verschillende types layout managers, het uitlijnen van objecten, het toevoegen van spacer widgets,….

Het nadeel van deze layout commando's is wel dat ze slechts éénmalig gedefinieerd worden. Wanneer de designer één van de user interface elementen verplaatst, zal er niet meer aan de gedefinieerde layout eigenschappen worden voldaan. Daarom werd er in hoofdstuk 5 gekeken naar de werking van constraint-based systemen. In deze systemen wordt een layout voorgesteld als een verzameling constraints die uiteindelijk worden opgelost door een constraint solver. De constraint solver zorgt ervoor dat aan de gedefinieerde constraints blijft voldaan. De gebruiker kan dus user interface elementen gaan verplaatsen, zonder dat layout commando's daarbij opnieuw gespecificeerd moeten worden.

Het doel van deze thesis bestaat uit het vinden van een mogelijkheid om layout eigenschappen op een informele manier te gaan specificeren en deze uit te testen in een prototype. Hierbij werd er rekening gehouden met het feit dat de meeste informele user interface design tools gebruik maken van pen-based interactietechnieken. Om diezelfde flexibiliteit te behouden, werd ervoor gekozen om voor de specificatie van layout commando's gebruik te maken van gestures. Om het meest gepaste recognition algoritme te vinden, werden een aantal algoritmen in hoofdstuk 6 van naderbij bekeken. De CALI library leek voor het te ontwikkelen prototype het meest geschikt te zijn. Enerzijds omdat deze library in staat is multi-stroke gestures te herkennen, anderzijds omdat deze een hoge recognition rate heeft. Het nadeel van de CALI library is wel dat deze enkel in staat is een aantal voorgedefinieerde gestures te herkennen. Voor de ontwikkeling van het prototype leek deze verzameling gestures voldoende te zijn.

De uiteindelijke layout wordt voorgesteld als een verzameling constraints. Voor het voorstellen en oplossen van het constraint systeem werd er gekozen gebruik te maken van de Cassowary constraint solving toolkit. De keuze voor deze toolkit berust op een aantal redenen. Ten eerste is de toolkit geschikt voor interactieve applicaties aangezien deze gebaseerd is op een incrementele versie van het simplex algoritme en daardoor vele kleine aanpassingen op korte tijd kan verwerken. Ten tweede is Cassowary in staat om zowel lineaire gelijkheden als ongelijkheden gelijktijdig op te lossen. Ten derde bestaat er in Cassowary de mogelijkheid om gewichten aan constraints toe te kennen. Op die manier kan gespecificeerd worden aan welke constraints te allen tijde moet worden voldaan, en welke constraints minder belangrijk zijn.

Na het prototype te hebben ontwikkeld, kan geconcludeerd worden dat de specificatie van layout eigenschappen door middel van gestures een goede en flexibele manier is. De designer kan tijdens het schetsen gewoon gebruik blijven maken van gestures om layout eigenschappen toe te voegen en dient geen menu op te roepen of button te selecteren om het commando uit te voeren. Toch moet worden vastgesteld dat de koppeling met onderliggend constraint systeem moeilijker verliep dan verwacht. Constraint-based layout systemen kampen immers nog met een aantal problemen. Bij het

definiëren van simpele layout commando's, zoals bijvoorbeeld het uitlijnen van twee elementen, werden weinig problemen ondervonden. Zodra de layout echter complexer werd, kon in sommige gevallen niet meer aan de gewenste situatie worden voldaan wat tot onvoorspelbare resultaten leidde.

De onvoorspelbare aard van constraints is één van de redenen waarom constraints vandaag de dag nog niet intensief in user interface toolkits worden gebruikt. Daarnaast zijn constraint-based systemen ook moeilijker te implementeren: het opbouwen van een juiste constraint verzameling en hiërarchie is immers een moeilijke taak. In [Myer00] worden ook nog een aantal andere redenen gegeven waaronder de declaratieve aard van constraints die programmeurs verplicht het probleem op een ander niveau te bekijken, de beperkingen die bepaalde constraint solvers op het definiëren van constraints opleggen en de moeilijkheid om bugs in constraint methoden op te sporen.

Daarnaast moeten constraint-based systemen ook over mogelijkheden bezitten om constraints te specificeren en op een duidelijke manier aan de gebruiker te presenteren. Eveneens moet er een mogelijkheid zijn om constraints te gaan editeren. In de meeste constraint-based systemen worden deze mogelijkheden geïmplementeerd in verschillende modi waartussen de gebruiker kan gaan switchen. Een aantal systemen maken voor de specificatie van constraints gebruik van een snapshot mechanisme. Met de ontwikkeling van dit prototype werd een andere mogelijkheid uitgeprobeerd namelijk de specificatie van layout commando's door middel van gestures. Er kan besloten worden dat dit een gemakkelijke en flexibele manier is.

8.2 Future work

Het ontwikkelde prototype kan op verschillende gebieden worden uitgebreid met een aantal mogelijkheden.

Een eerste mogelijke uitbreiding is het prototype verder uit te bouwen tot een complete informele sketching tool. Het belangrijkste hierbij is dat er een mogelijkheid moet worden toegevoegd waarmee de gecreëerde interface getransformeerd kan worden naar code of een andere representatie. Daarnaast dient ook de visualisatie van constraints en de mogelijkheid om constraints te editeren verder te worden uitgewerkt. Momenteel worden de constraints tekstueel weergegeven in een lijst en worden deze gevisualiseerd op het canvas door middel van lijnen en rechthoeken. Deze representaties kunnen verder worden uitgebreid. Daarnaast moet ook een mechanisme worden toegevoegd om constraints te editeren. Het lijkt handig hiervoor gebruik te maken van een tweede modus waarin het systeem geschakeld kan worden en waar de gebruiker de mogelijkheid krijgt gedefinieerde constraints aan te passen. Daarna kan het systeem terug in "gewone modus"

worden geschakeld waar de gebruiker verder kan gaan met het schetsen van de user interface.

Het prototype zelf dient ook nog met een aantal opties te worden uitgebreid. Zo moet er bijvoorbeeld een mogelijkheid worden voorzien waarmee de gecreëerde user interface kan worden opgeslagen en op een later tijdstip terug kan worden ingeladen. Ook undo en redo operaties kunnen worden toegevoegd.

Een tweede mogelijke aanvulling is een uitbreiding van de huidige layout commando's en het toevoegen van nieuwe commando's. Er wordt daarbij vooral gedacht aan het toevoegen van layout managers zoals bijvoorbeeld een gridlayout of borderlayout. Wanneer er nieuwe layout commando's worden toegevoegd, moeten deze ook gekoppeld worden aan een passend gesture. Er moet hierbij rekening gehouden worden dat de CALI recognition engine maar een beperkt aantal gestures bevat, waarvan momenteel al een groot deel door de huidige layout commando's worden gebruikt. Daarom moeten andere mogelijkheden voor de herkenning van gestures worden bekeken.

BIBLIOGRAFIE

[Ali02]

Ali M.F., Pérez-Guiñones M.A., Abrams M., *Building Multi-Platform User Interfaces with UIML*, In Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces, CADUI 2002, France, 2002, p.225-236.

[Badr98a]

Badros Greg J., Borning Alan, *The Cassowary Linear Arithmetic Constraint Solving Algorithm: Interface and Implementation*, Technical Report UW-CSE-98-06-04, University of Washington, Department of Computer Science and Engineering, Seattle, Washington, June 1998.

[Badr98b]

Badros Greg J., *Constraints in Interactive Graphical Applications*, Ph.D. Thesis, University of Washington, Department of Computer Science and Engineering, Seattle, Washington, December 1998.

[Badr00]

Badros Greg J., Nichols Jeffrey, Borning Alan, *Scwm: An Extensible Constraint-Enabled Window Manager*, 2000.

[Bier88]

Bier Eric A., *Snap-Dragging: Interactive Geometric Design in Two and Three Dimensions*, Technical Report CSD-88-416, University of California, Berkeley, 1988.

[Blan06]

Blanchette Jasmine, Summerfield Mark, *C++ GUI Programming with Qt 4*, Prentice Hall, 2006.

[Caet02]

Caetano Anabela, Goulart Neri, Fonseca Manuel, Jorge Joaquim, *JavaSketchIt: Issues in Sketching the Look of User Interfaces,* AAAI Spring Symposium on Sketch Understanding, 2002.

[Calv03]

Calvary G., Coutaz J., Thevenin D., Limbourg Q., Bouillon L., Vanderdonckt J., *A Unifying Reference Framework for Multi-Target User Interfaces*, Interacting with Computers, 15(3), 2003, p.289-308.

[Camp04]

Campos Pedro F., Nunes Nuno J., *CanonSketch: A User-Centered Tool for Canonical Abstract Prototyping*, In Proceedings of the International Conference on Engineering Human-Computer Interaction / International Workshop on Design, Specification and Verification of Interactive Systems, EHCI/DSV-IS 2004, Hamburg, Germany, 2004.

[Cons]

UW Constraint-Based Systems, http://www.cs.washington.edu/research/constraints/ .

[Cons03]

Constantine Larry L., *Canonical Abstract Prototypes for Abstract Visual and Interaction Design*, University of Technology, Sydney, Australia, July 2003.

[Coye05]

Coyette Adrien, *SketchiXML: A Sketching Tool for Designing User Interfaces for Information Systems*, Ph.D. Thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2005.

[Coye06]

Coyette Adrien, Vanderdonckt Jean, Limbourg Quentin, *SketchiXML: An Informal Design Tool for User Interface Early Prototyping*, In Proceedings of RISE 2006 Workshop on Rapid User Interface Prototyping Infrastructures Applied to Control Systems, RUIPICAS 2006, Geneva, September 2006.

[Deni]

DENIM help file, Denim Project Page: http://dub.washington.edu/projects/denim/ .

[Eise01]

Eisenstein Jacob, Vanderdonckt Jean, Puerta Angel, *Model-Based User-Interface Development Techniques for Mobile Computing*, In Proceedings of ACM International Conference on Intelligent User Interfaces, IUI 2001, Santa Fe, January 2001.

[Fons02]

Fonseca Manuel J., Pimentel César, Jorge Joaquim A., *CALI: An Online Scribble Recognizer for Calligraphic Interfaces*, In AAAI Spring Symposium on Sketch Understanding, AAAI Technical Report SS-02-08, 2002, p.51-58.

[Glei93]

Gleicher Michael, *A Graphics Toolkit Based on Differential Constraints*, In Proceedings of the 1993 ACM Symposium on User Interface Software and Technology, UIST 1993, 1993, p.109-120.

[Gros96]

Gross Mark D., Do Ellen Yi-Luen, *Ambiguous Intentions: A Paper-like Interface for Creative Design*, In Proceedings of the 9th Annual Symposium for User Interface Software and Technology, UIST 1996, 1996, p.183-192.

[GTK1]

GTK+ - The GIMP Toolkit, http://www.gtk.org/ .

[GTK2]

GTK+ 2.0 Tutorial, http://www.gtk.org/tutorial/index.html .

[Helm92]

Helm Richard, Huynh Tien, Marriott Kim, Vlissides John, *An Object-Oriented Architecture for Constraint-Based Graphical Editing*, Workshops on Object-Oriented Graphics, 1992, p.217-238.

[Hong02]

Hong Jason I., Landay James A., *SATIN: A Toolkit for Informal Ink-based Applications*, In Proceedings of the 2000 ACM Symposium on User Interface Software and Technology, UIST 2000, CHI Letters 2(2), 2000, p.63-72.

[Howe95]

Hower Walter, Graf Winfried H., *Research in Constraint-Based Layout, Visualization, CAD, and Related Topics: A Bibliographical Survey*, December 1995.

[Huds90]

Hudson Scott E., Mohamed Shamim P., *Interactive Specification of Flexible User Interface Displays*, ACM Transactions on Information Systems, 8(3), July 1990, p.269-288.

[Huds93]

Hudson Scott E., *User Interface Specification Using an Enhanced Spreadsheet Model*, ACM Transactions on Graphics, 13(3), 1993, p.209-239.

[Igar98]

Igarashi Takeo, Matsuoka Satoshi, Kawachiya Sachiko, Tanaka Hidehiko, *Pegasus: A Drawing System for Rapid Geometric Design*, In Proceedings of ACM Conference on Human Factors in Computing Systems, CHI 1998, Los Angelos, USA, April 1998, p.24-25.

[Java1]

Java SE Desktop Technologies, http://java.sun.com/javase/technologies/desktop/techoverview.jsp#awt .

[Java2]

SWT, Swing or AWT: Which is right for you?, http://www-128.ibm.com/developerworks/grid/library/os-swingswt/ .

[Kara05]

Karam Maria, Schraefel M.C., *A Taxonomy of Gestures in Human Computer Interaction*, ACM Transactions on Computer-Human Interactions, 2005.

[Kurl93]

Kurlander David, Feiner Steven, *Inferring Constraints from Multiple Snapshots*, ACM Transactions on Graphics, 12(4), October 1993, p.277-304.

[Land95]

Landay James A., Myers Brad A., *Just Draw It! Programming by Sketching Storyboards*, Technical Report CMU-CS-95-199, School of Computer Science, Carnegie Mellon University, Pittsburgh, November 1995.

[Land96]

Landay James A., *Interactive Sketching for the Early Stages of User Interface Design*, Ph.D. Thesis, Technical Report CMU-HCII-96-105, School of Computer Science, Carnegie Mellon University, Pittsburgh, December 1996.

[Land01]

Landay James A., Myers Brad A., *Sketching Interfaces: Toward More Human Interface Design*, In Computer, 34(3), March 2001, p.56-64.

[Land02]

Landay James A., Hong Jason I., Klemmer Scott R., Lin James, Newman Mark W., *Informal PUIs: No Recognition Required*, In Proceedings of AAAI 2002 Spring Symposium (Sketch Understanding Workshop), Stanford, CA, March 2002, p.86-90.

[Lieb93]

Lieberman Henry, *Mondrian: A Teachable Graphical Editor*, In Proceedings of the SIGCHI conference on Human Factors in Computing Systems, Amsterdam, The Netherlands, 1993, p.144.

[Lin00]

Lin James, Newman Mark W., Hong Jason I., Landay James A., *DENIM: Finding a Tighter Fit Between Tools and Practice for Web Site Design*, In CHI Letters: Human Factors in Computing Systems, CHI 2000, 2000, 2(1), p.510-517.

[Lin02]

Lin James, Thomsen Michael, Landay James A., *A Visual Language for Sketching Large and Complex Interactive Designs*, In CHI Letters: Human Factors in Computing Systems, CHI 2002, 2002, 4(1), p.307-314.

[Lodi99]

Lodi Andrea, *Algorithms for Two-Dimensional Bin Packing and Assingment Problems*, Ph.D. Thesis, Universita di Bologna, 1999.

[Lok01]

Lok Simon, Feiner Steven, *A Survey of Automated Layout Techniques for Information Presentations*, In Proceedings of SmartGraphics 2001, March 2001.

[Long99]

Long Allan.Christian, Jr., Landay James A., Rowe Lawrence A., *Implications For a Gesture Design Tool*, In Proceedings of Human Factors in Computer Systems, CHI 1999, Pittsburgh, PA, May 1999, p.40-47.

[Marr02]

Marriott Kim, Chok Sitt Sen, *QOCA: A Constraint Solving Toolkit for Interactive Graphical Applications*, Constraints Archive, 7(3-4), July-October 2002, p.229-254.

[Mori02]

Mori Giulio, Paternò Fabio, Santoro Carmen, *CTTE: Support for Developing and Analyzing Task Models for Interactive System Design*, In IEEE Transactions on Software Engineering, 28(9), September 2002.

[Myer97]

Myers Brad A., McDaniel Rich, Miller Rob, Ferrency Alan, Faulring Andrew, Borison Ellen, Kyle Bruce, Mickish Andy, Klimovitski Alex, Doane Patrick, *The Amulet Environment: New Models for Effective User Interface Software Development*, In Software Engineering, May 1997, 23(6), p.347-365.

[Myer93]

Myers Brad A., McDaniel Richard G., Kosbie David S., *Marquise: Creating Complete User Interfaces by Demonstration*, In Proceedings of the SIGCHI conference on Human Factors in Computing Systems, Amsterdam, The Netherlands, 1993, p.293-300.

[Myer00]

Myers Brad A., Hudson Scott E., Pausch Randy, Past, *Present, and Future of User Interface Software Tools*, ACM Transactions on Computer-Human Interaction, 7(1), March 2000, Special Issue on Human-Computer Interaction in the New Millennium part 1, p.3-28.

[Niel93]

Nielsen Jakob, *Usability Engineering*, Academic Press, 1993.

[Nóbr05]

Nóbrega Leonel, Nunes Nuno Jardim, Coelho Helder, *DialogSketch: Dynamics of the Canonical Prototypes*, In Proceedings of the 4th International Workshop on TAsk MOdels and DIAgrams for User Interface Design: For Work and Beyond, TAMODIA 2005, Gdansk, Poland, September 2005.

[Nune01]

Nunes Nuno Jardim, Falcao e Cunha Joao, *Wisdom: A UML Based Architecture for Interactive Systems*, Lecture Notes in Computer Science, 2001.

[Knud99]

Knudsen Jonathan, Java 2D Graphics, O'Reilly. 1999.

[Petr06]

Petrie Jennifer N., Schneider Kevin A., *Mixed-Fidelity Prototyping of User Interfaces*, In Proceedings of the 13th International Workshop on Design, Specification and Verification of Interactive Systems, DSVIS 2006, Dublin, Ireland, July 2006, p.195-208.

[Qt1]

Qt Product Features, http://www.trolltech.com/products/qt/features/index.

[Qt2]

Qt's Classes, http://doc.trolltech.com/4.1/classes.html.

[Rubi91a]

Rubine Dean, *Specifying Gestures by Example*, Computer Graphics: ACM SIGGRAPH 1991, July 1991, p.329-337.

[Rubi91b]

Rubine Dean, *The Automatic Recognition of Gestures*, Ph.D. Thesis, Technical Report CMU-CS-91-202, School of Computer Science, Carnegie Mellon University, Pittsburgh, December 1991.

[Sann93]

Sannella Michael, Maloney John, Borning Alan, *Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm*, In Software-Practice and Experience, May 1993, 23(5), p.529-566.

[Suth63]

Sutherland Ivan Edward, *Sketchpad: A Man-machine Graphical Communication System*, Ph.D. Thesis, Massachusetts Institute of Technology, January 1963.

[Swin]

Swing tutorial, http://java.sun.com/docs/books/tutorial/uiswing/index.html.

[Traet02]

Traetteberg Hallvard, *Model Based Design Patterns*, In Proceedings of the Workshop on User Interface Design Patterns, CHI 2000, 2000.

[Verm05]

Vermeulen Jo, *Widget Set Independent Layout Management for UIML*, Thesis, Universiteit Hasselt, Belgium, June 2005.

[Vlis89]

Vlissides John M., Linton Mark A., *Unidraw: A Framework for Building Domain-Specific Graphical Editors*, In Proceedings of the 2nd Annual ACM SIGGRAPH Symposium on User interface Software and Technology, Williamsburg, Virginia, US, 1989, p.158-167.

[Walk02]

Walker Miriam, Takayama Leila, Landay James A., *High-Fidelity or Low-Fidelity, Paper or Computer? Choosing Attributes When Testing Web Prototypes*, In Proceedings of the Human Factors and Ergonomics Society, Annual Meeting: HFES2002, September-October 2002, p.661-665.

[Will97]

Willey Mark, *Design and Implementation of a Stroke Interface Library*, IEEE Region 4 Student Paper Contest, 1997.

[Wort03]

Worth Carl D., *Xstroke: Full-Screen Gesture Recognition for X*, In Proceedings of the USENIX 2003 Annual Technical Conference, 2003, p. 187-197.

Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen, de gevraagde informatie in te vullen (en de overeenkomst te ondertekenen en af te geven).

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Sketch-Based User Interface Design
Richting: **Master in de informatica** Jaar: **2007**
in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Ik ga akkoord,

Kathleen Renders

Datum: **23.05.2007**