IT-Universitetet i København
Softwareudviklingslinjen
Kandidatafhandling
Vejledt af Kasper Østerbye
Marts 2005

The Hierarchical Graphical Library

Design og udvikling af et bibliotek til GUI programmering

Skrevet af:
Thomas Quistgaard
cprnr: 200677-2989
e-mail: tquistgaard@itu.dk

Abstract

The Hierarchical Graphical Library (HGL) is an alternative to the Swing API for Java. HGL provides an easier way of programming graphical user interfaces (GUI), in form of a hierarchical component structure, combined with new ways of handling component layout information and events.

In this master thesis at ITU in Copenhagen I'm developing an alternative way to construct GUI with the sole purpose of simplifying GUI programming for Java and more specific the Swing library. The motivation for the thesis is based on my own frustrations with creating Swing applications and the result, or intension, hereof is a library which eliminates these frustrations, is easier to use and more intuitive. The target audience for the library is mainly newcomers to object oriented programming (OOP) with Java. The reason why the library targets newcomers is the huge amount of constrains associated with usage hereof, opposite to Swing where you have all the flexibility you could wish for. Programmers are strongly limited in what they can do, for example you are not allowed to places panels in buttons as you would in Swing, these constrains and other elements are part of the simplification process.

HGL uses a hierarchical way of structuring components which is not seen in Java or any other newer object oriented programming language. The principle behind this hierarchical approach is declarative instead of imperative. A GUI will be created in a nested hierarchical structure based on anonymous inner classes, which I think is more intuitive than the current way of constructing GUI. The hierarchical structure clearly reflects the parent/child relationship between components as opposite to Swing.

Layout information which among other things determines the actually position of a component in its container, is removed from layout managers and into a new feature in Java 1.5 called annotations – in fact HGL hides the use of layout managers from the user, so the only thing the user have to be concerned with is using the appropriat annotations.

Event handling in HGL is carried out only by use of methods, no classes, interfaces or nested structures are needed. HGL provides a set of highly intuitive event methods for the different components.

Because of the programming language Java, HGL does not fully satisfy the technical design criteria's, but still it comply with the main goal for the thesis, that is it simplifies GUI programming with Swing – though if Java had been a more comprehensive language, and thereby been able to solve all the technical criteria's, the goal could have been satisfied even more.

Indhold

Kapitel 1

Forord

Denne rapport udgør mit speciale på linjen for softwareudvikling ved IT-Universitetet i København.

1.1 Deltagerforudsætninger

Jeg er studerende på SWU (softwareudvikling) linjen ved ITU, og har en baggrund som datamatiker fra Lyngby Uddannelsescenter (LUC), en bachelor i software konstruktion fra Aalborg Universitet, og har som studiejob arbejdet i konsulenthuset Basset A/S. Studiejobbet har udelukket omhandlet web implementeret i Microsoft teknologi som ASP og ASP.NET med C# som programmeringssprog.

Da jeg påbegyndte specialet, var mit eneste konkrete kendskab til Java fra kurset OOP (Objekt Orienteret Programmering), hvoraf én kursusdag omhandlede Swing, samt $2\frac{1}{2}$ dags introduktion til Java 1.5.

Oprindeligt var vi to personer der startede med at skrive specialet. Martin valgte dog at søge andre veje fire måneder henne i forløbet, mens jeg besluttede at forsætte. Gennem diverse tutorials og eksperimenter med Swing dannede vi tilsammen dele af tankesættet, og ideerne omkring det framework vi ønskede at designe. På trods af at Martin valgte at gå andre veje, vil jeg takke ham for de mange inspirerende diskussioner gennem de fire måneder.

1.2 Rapportens struktur

Rapportens struktur som gennemgåes i følgende afsnit, afspejler til dels tilgangsvinklen til forløbet, hvorfor et metode afsnit ikke indgår i rapporten.

Grunden til at nedenstående kun til dels afspejler forløbet, skyldes at processen var mere iterativ end vandfaldsagtig, udgangspunktet for forløbet var dog den opstillede struktur.

Abstract Et kort engelsk resumé af rapporten og dens konklusioner.

Kapitel 1 Forord Dette kapitel.

Kapitel 2 Motivation Her begrundes motivationen for specialet på et overordnet niveau, der efterfølgende konkretiseres i problemformuleringen i kapitel 4.

Kapitel 3 Metoder til konstruktion af grafiske brugergrænseflader Kapitlet opsummerer research fra fire kendte biblitoteker AWT, Swing, Windows Forms og Web Forms, og ser kort på GUI builders. Tilsidst gennemgåes to eksisterende bibliotekter, der har forsøgt at simplificere Swing (CookSwing: XML to Swing GUI og Buoy: A Better User Interface Toolkit). Research arbejdet i kapitlet benyttes efterfølgende, sammen med motivationen, til at danne problemformuleringen og give input til designet og de konkrete tiltag.

Kapitel 4 Problemformulering Med udgangspunkt i de to forgående kapitler klassificeres en række tiltagsområder. Retning og afgrænsninger defineres, og den endelige overordnet målsætning dannes.

Kapitel 5 Designmål På baggrund af researcharbejdet fra kapitel 3 og problemformuleringen opstilles, en række enkeltstående designmål for arkitekturen, der hver især bidrager til den overordnet målsætning. Kapitlet beskriver ligeledes de problemer jeg ser med Swing, i forhold til en simpel konstruktion af brugergrænseflader.

Kapitel 6 Design Med udgangspunkt i designmålene skabes en arkitektur, der udmunder i et konkret bibliotek kaldet HGL (The Hierarchical Graphical Library). Kapitlet beskriver arkitekturen dannet på baggrund af disse designmål, samt de implementeringskonstruktioner der benyttes for at nå målet. Kapitlet indeholder uddrag fra koden, den komplette kode findes i bilag D.

Kapitel 7 Eksempel Kapitlet præsenterer dele af en komplet applikation udviklet i både HGL og Swing, hvor fokus er at sammenligne GUI opbygningen for de to biblioteker. Formålet er at vise et praktisk eksempel på, hvordan HGL blandt andet har simplificeret opbygningen af GUI.

Kapitel 8 Konklusion og videre arbejde I kapitlet konkluderes resultatet af den overordnede målsætning: *Har jeg gjort GUI programmering med Swing nemmere for begyndere?*, samt hvilke muligheder der er for et videre arbejdsforløb med HGL.

1.3 Hjemmeside

I tilknytning til specialet har jeg oprette en hjemmeside hos SourceForge.net, hvorfra det er muligt at hente koden til den implementeret prototype af biblioteket HGL, samt nogle eksempler på brugen heraf. Derudover er det også muligt at få en række generelle informationer omkring biblioteket. Alle informationerne vil dog findes i rapporten her, hvorfor det ikke er nødvendigt at læse indholdet af hjemmesiden i forbindelsen med specialet, den skal blot anses for at være en online informationskilde for HGL.

Adressen til hjemmesiden er http://hgl.sourceforge.net

1.4 Begrebsafklaring

I det følgende gives forklaring på begreber anvendt i rapporten, hvis betydning der kan være tvivl om, eller som er særligt centrale.

HGL

HGL er forkortelsen for biblioteket der udformes, og står for The Hierarchical Graphical Library.

Grafisk brugergrænseflade

Betegnelse for den måde, hvorpå et computerprogram kommunikerer med brugeren via vinduer, ikoner og dialogbokse mv. Den engelske betegnelse graphical user interface, GUI, anvendes ofte for den beskrevne brugergrænseflade.

Bruger/Nybegynder

Refererer til den målgruppe biblioteket tænkes brugt af. Målgruppen defineres som værende nybegyndere i programmering inden for det objekt orienterede paradigme. Målgruppen tager ligeledes udgangspunkt i min egne erfaringer med Swing, før jeg påbegyndte specialet, dvs. folk med erfaring i objekt orienterede programmering, men som ikke har udviklet GUI med Swing og Java.

Komponent/Element

Ordet komponent/element bruges i flæng gennem rapporten, og refererer til grafiske elementer som knap, liste, vindue, panel osv. med mindre andet er angivet.

Java 1.5 og Java 5.0

Gennem rapporten refererer jeg til den nyeste version af Java som "Java 1.5". Mange kender den også som Java 5.0. Java 1.5 og Java 5.0 er dog én og samme ting.

Metode erklæringer i teksten

Metode erklæringer i teksten har to former, afhængigt af om metoden har parametre eller ej. En metode uden parametre angives med metodenavn efterfulgt af start og slut parentes, eks. `myMethod()`. En metode med parametre angives derimod med metodenavn efterfulgt af start parentes, tre punktummer og slut parentes, eks. `myMethod(...)`.

Kapitel 2

Motivation

Mange computersystemer har i dag en grafisk brugergrænseflade. Der er selvfølgelig undtagelser, men grafiske brugergrænseflader er blevet den almindelige måde at interagere med computerprogrammer på. Grafiske brugergrænseflader får derfor også større og større opmærksomhed. I de seneste årtier har der bl.a. været stigende fokus på programmers brugervenlighed. Der har været arbejdet med udviklingen af værktøjer og biblioteker der skal understøtte konstruktion af grafiske brugergrænseflader. GUI programmering er blevet udbredt til en større skarre af programmører end tidligere, og indgår nu typisk som del af pensum for studerende, der skal lære programmering for første gang.

Programmering af grafiske brugergrænseflader fylder en stor del af den samlede udviklingsindsats i forbindelse med softwareudvikling. Empiriske undersøgelser viser, at op mod 50% af den samlede programkode der skrives, er kode der vedrører et programs grafiske brugergrænseflade [27], hvilket afspejler den store arbejdsbyrde i GUI programmering.

2.1 Problematik ved GUI programmering

Der er flere ting der springer i øjnene, når man giver sig i kast med programmering af grafiske brugergrænseflader. Grundlæggende adskiller programmering af grafiske brugergrænseflader sig fra anden programmering, i det man i tekstuel form beskriver, hvordan noget skal præsenteres visuelt. Man programmerer/skriver, hvordan visuelle elementer skal præsenteres og opføre sig. Det kræver et højt abstraktionsniveau, idet tekst ikke er oplagt til at beskrive en visuel konstruktion. Det er derimod en skitse, et billede, en fysisk model el. lign. Det kan være en af de overordnede årsager til, at GUI programmering kan virke mere kompliceret, end man måske regner med.

Sproget eller biblioteket som anvendes til GUI konstruktion kan således være afgørende for, hvor stort springet er mellem kode og den endelige visuelle præsentation, altså hvor let det er at skrive en GUI.

Omfanget af de biblioteker man skal anvende for at konstruere grafiske brugergrænseflader kan virke overvældende. Swing består eksempelvis af flere hundrede klasser og interfaces, hertil kommer AWT som Swing er baseret på. En komponent i Swing har mere end 300 metoder osv. Det kræver lang tid at få et overblik over Swing, og endnu længere tid at blive fortrolig med den indeholdte funktionalitet, alene grundet omfanget heraf. Indlæringskurven for udnyttelse af biblioteker til konstruktion af grafiske brugergrænseflader er typisk stejl.

GUI programmering skal være lettere tilgængeligt, og det skal ikke være overraskende kompliceret hvordan den ønskede grafiske brugergrænseflade skal programmeres. Abstraktionsniveauet fra kode til det færdige skærmbillede og dets opførsel skal være lavere end det er nu.

2.2 Min motivation

Min motivation for at skrive dette speciale stammer konkret fra mine egne oplevelse med GUI programmering i Java – nærmere bestemt Swing. Som ny og entusiastisk programmør kaster man sig i dag hurtigt over programmering af grafiske brugergrænseflader. Det er dog min oplevelse, at man bliver overrasket over omfanget af og kompleksiteten forbundet med konstruktionen af grafiske brugergrænseflader i Swing. Det gjorde jeg i hvert fald. Hvad der virkede ligetil og intuitivt, man skulle jo bare bruge et vindue med nogle knapper og en dialogboks i, blev ret hurtigt overraskende besværligt og tidskrævende. Dette skal ses i lyset af min baggrund som Microsoft udvikler, hvor tingene virker mere simpelt og ligetil, hvilket førte til en stor forundring over Swing's kompleksitet.

2.3 Min overordnede målsætning

Med udgangspunkt i ovenstående, vil jeg gerne forstå kompleksiteten forbundet med programmering af grafiske brugergrænseflader i Swing. Herunder undersøge hvordan programmering af grafiske brugergrænseflader adskiller sig fra anden programmering, samt hvordan GUI programmering understøttes af moderne programmeringssprog, biblioteker og værktøjer.

Med udgangspunkt i Swing som bibliotek til programmering af grafiske brugergrænseflader og Java som programmeringssprog, vil jeg gerne gøre GUI

programmering mere intuitivt og mere simpelt at gå til. Resultatet af specialet udmunder i en arkitektur for et selvstændig Java bibliotek, samt en proof-of-concept implementering heraf.

Kapitel 3

Metoder til konstruktion af grafiske brugergrænseflader

Der findes et hav af forskelligt software, der skal støtte konstruktionen af grafiske brugergrænseflader. Man kan opdele og kategorisere disse systemer i forskellige lag som vist i tabel 3.1. Ikke alle systemer kan dog placeres i denne model, for nogle systemer giver det mere mening at udvide modellen med flere lag, og for andre kan lag være slået sammen. Opdelingen tjener dog fint som en klassificering og et udgangspunkt for at undersøge de forskellige tilgange og metoder, der findes til konstruktion af grafiske brugergrænseflader.

Tabel 3.1: Hovedbestanddelene i systemer til udvikling af grafiske brugergræseflader.

Applikation
Værktøjer
Biblioteker
Windowing system
Operativ system

I dag findes der til de fleste operativsystemer basis funktionalitet, der muliggør udvikling af grafiske brugergrænseflader. Denne funktionalitet og dets interface kaldes for et *windowing system*. Windows indeholder et integreret windowing system, til Unix varianter er X [10] et eksempel på et windowing system. Den kommende version af Windows, kaldet Longhorn [5], indeholder et nyt windowing system, der hedder Avalon [17]. Et windowing system kan betragtes som værende noget, der kombinerer output fra forskellige applikationer, og realiserer det på brugerens skærme i deres respektive vinduer. Et windowing system er typisk også ansvarligt for at dirigere input fra brugeren

(fra keyboard, mus osv.) videre til de relevante applikationer (via bibliotekerne de måtte være baseret på).

Ovenpå windowing systemet findes en række biblioteker og væktøjer, der skal støtte og effektivisere udviklingen af grafiske brugergrænseflader. Jeg koncentrerer mig om bibliotekerne og værktøjerne, hvilket er den del af systemet, der giver det højeste niveau af abstraktion, og som typisk anvendes direkte af programmører af grafiske brugergrænseflader.

Da mængden af værktøjer og biblioteker er så stor som den er, vil jeg generalisere disse, og kun beskrive dem i det omfang, de bidrager med nye måder at støtte konstruktionen af grafiske brugergrænseflader på, set udfra en applikationsprogrammørs perspektiv. Fokusområdet for kapitlet (og specialet) er dermed en applikationsprogrammørs tilgangsvinkel til konstruktion af GUI programmer. Jeg har udvalgt følgende biblioteker og værktøjer, som jeg vil behandle:

Tabel 3.2: Udvalgte biblioteker og værktøjer

Navn	**Type**	**Platform**
Abstract Window Toolkit (AWT)	Bibliotek	Java
Swing	Bibliotek	Java
Standard Widget Toolkit (SWT)	Bibliotek	Java
Windows Forms	Bibliotek	Microsoft .NET
Web Forms	Bibliotek	Microsoft .NET
JDeveloper	Værktøj	Java
JBuilder	Værktøj	Java
Visual Studio .NET	Værktøj	Microsoft .NET
Delphi 2005 alias C#Builder	Værktøj	Microsoft .NET
CookSwing	Bibliotek	Java
Buoy	Bibliotek	Java

Formålet med dette afsnit er, på baggrund af de eksisterende systemer at konkretisere specialets overordnede målsætning. Hvilke tiltag, der ikke allerede eksisterer, kan jeg tage for at gøre GUI programmering lettere? Hvilke idéer kan jeg eventuelt benytte fra de eksisterende systemer? Hvilke problemer eksisterer der med de nuværende systemer?

3.1 Biblioteker

Der findes biblioteker til udvikling af grafiske brugergrænseflader til de fleste moderne programmeringssprog. I det store hele minder de forskellige biblioteker meget om hinanden. De består typisk som minimum af følgende dele

jf. tabel 3.3:

Tabel 3.3: Biblioteker basal funktionalitet

Et predefineret sæt af komponenter og mekanismer til kontrol af deres visuelle egenskaber.
Mekanismer til tegning og gentegning af komponenter.
Mekanismer til opbygning og placering af komponenter.
Mekanismer til kommunikation mellem GUI og program.

Brugen af bibliotekerne er forskellig og bestemt af bibliotekets arkitektur, designkritierer og programmeringssproget de anvendes fra. Antallet af komponeneter, måden hvorpå komponenterne placeres og hvordan biblioteket inteagerer med windowing systemet er forskelligt. For de udvalgte biblioteker i tabel 3.2 disktures disse forskelligheder og fordele/ulemper herved i det følgende.

3.1.1 AWT

Abstract Window Toolkit (AWT) er det originale bibliotek i Java til konstruktion af grafiske brugergrænseflader [29]. Med AWT 1.0, som er en del af JDK 1.0, kan der skrives simple grafiske brugergrænseflader til programmer og applets. Mængden af funktionalitet i AWT 1.0 er begrænset, men dækker over alle områderne listet i 3.3. Sættet af predefinerede komponenter i AWT er begrænset til kun at inkludere de mest gængse grafiske komponenter. Tabel 3.4 viser hvilke komponenter AWT 1.1, som er den seneste version af AWT, indeholder. Som det fremgår er komponenter, som i dag er kendt fra mange applikationer, såsom træer, tabeller osv. ikke inkluderet. Simple ting som tooltips (beskrivende tekst der kommer frem, når man bevæger musen over eksempelvis en label) er heller ikke en del af sættet af komponeneter.

Tabel 3.4: AWT 1.1 komponenter

Button	Canvas	Checkbox
Choice	Dialog	FileDialog
Frame	Label	List
Panel	ScrollPane	Scrollbar
TextArea	TextField	Window

Den begrænsede mængde af funktionalitet, skyldes at man med AWT ville have et bibliotek, hvor også GUI koden var platformsuafhængig – intentinen om “Write Once, Run Anywhere” skulle også gælde for konstruktionen af grafiske brugergrænseflader. Måden hvorpå dette er implementeret i AWT,

er ved at have platformsspecifikke implementeringer, også kaldet *peers*, af de forskellige komponeneter, for hver platform som ønskes understøttet. Disse peers er typisk skrevet i C eller C++, og delegerer konstruktionen af de forskellige komponenter videre til eksempelvis det underliggende windowing system jvf. tabel 3.1. Dette betyder at en knap vil ligne en typisk Windows knap, hvis programmet afvikles på en Windows platform, og knappen vil ligne en Mac knap, hvis programmet køres på en Mac. API'en og Java implementeringen der udgør AWT er den samme, men den native implementering af de forskellige komponenters peers er forskellig fra platform til platform.

Der er fordele og ulemper ved AWT's peer model. Den primære motivation for designet er at gøre GUI koden platformsuafhængig ("Write Once, Run Anywhere"), ved at anvende et komponentsæt der er tilgængeligt på samtlige platforme. Programmøren behøver således ikke bekymre sig om platform forskelligheder. Ulempen er at sættet af komponenter og deres funktionalitet bliver "laveste fællesnævner". En anden ulempe er, at logikken der genererer den grafiske brugergrænseflade, er gemt væk i native implementeringer. Således bliver det vanskeligt at overskrive eller udbygge de eksisterende implementeringer af komponenterne. De forskellige peers skal løbende holdes up-to-date, for at sikre konsistens de forskellige platforme imellem. Udover at sørge for at de forskellige komponenter er tilgængelige på alle platforme, skal man også sikre sig, at de opfører sig ens (eksempelvis reagerer ens på input osv.). Vedligeholdes native koden ikke ryger platformsuafhængigheden. Ønsker man med AWT 1.0 at udvide samlingen af komponenter med egne komponenter, kan dette gøres ved at specialisere `Canvas` eller `Panel` klassen jvf. tabel 3.4. Problemet med dette er, at man altid vil få en *heavyweight* komponent (et vindue eller et lignenede område til at tegne i fra windowing systemet). Heavyweight komponenter vil typisk være rektangulære og ikke transparente, afhængig af mulighederne i det underliggende windowing system og implementeringen af peers.

Disse ulemper førte til, at Sun med JDK 1.1 og AWT 1.1 introducerede muligheden for anvendelse af *lightweight* komponenter. Lightweight komponenter har ingen native peers, dvs. de har ingen relation til windowing systemet som sådan, udover at lightweight komponenter altid vil være placeret i en heavyweight komponent. Lightweight komponenter er rene Java komponenter, der tegnes vha. geometriske primitive. Ingen af komponenter fra tabel 3.4 er lightweight komponenter, men man kan oprette egne lightweight komponenter ved at specialisere AWT's `Component` klassen.

AWT 1.0's event model har også gennemgået en opgradering i AWT 1.1. Den gamle event model baseres på arv. For at fange og håndtere events, skulle man nedarve fra `AWT.Component` og enten overskrive `action()` eller `handleEvent()` metoderne. Ved at returnere sand (`true`) fra en

af disse metoder slutter, eventet hvor det ellers ville forsætte opad i GUI hierarkiet, indtil det blev stoppet eller roden af hierakiet nået.

Event modellen kan sammenlignes med en kædereaktion. Tag eksempelvis et klik på en knap. Knappens peer modtager besked fra operativ systemet og kalder knappens `deliverEvent()` metode, som kalder knappens `postEvent()`, som kalder `handleEvent()` metode, der kalder `action()` metoden. Såfremt knappens `action()` metode returnere sand, vil dens `handleEvent()` også returnere sand, og dens `postEvent()` stopper kædereaktionen. Hvis `action()` metoden derimod havde returnerede falsk, ville `handleEvent()` også returnere falsk og `postEvent()` ville forsætte kædereaktionen til dens forældre, som er komponenten der indeholder knappen. Som sagt ville samme procedure forsætte indtil der enten returneres sand eller roden af hierakiet nåes.

Resulatet af AWT 1.0's event model giver programmøren to muligheder til at håndtere events. (1) Hver individuel komponent nedarves til at håndtere dets events, hvilket giver en overflod af klasser, eller (2) en mængde af events håndteres af én container, hvilket kræver en stor og kompleks betinget erklæring.

I event modellen i AWT 1.1 erstattes kædereaktion med lyttere (listeners). Events sendes fra en kilde til en lytter, ved at kalde en metode med instansen af eventet på lytteren. Princippet bag dette er identisk med Oberserver designmønstret [13]. En lytter er et objekt som implementerer et specifikt `EventListener` interface. Interfacet definerer en eller flere metoder, som aktiveres af event kilden afhængigt af event typen. Kilden er et objekt hvorfra eventet opstår og sendes. Typisk vil kilden i GUI programmering være grafiske elementer aktiveret af slutbrugere, men kan også sagtens være "interne" komponenter. Alt eventhåndtering, såvel som tegning af elementer, håndteres af en tråd kaldet *event-dispatching*.

Placering af komponenter på skærmen foregår i AWT ved anvendelse af såkaldte *layout managers*. Man tilknytter en layout manager til en container, og denne bestemmer så komponenternes størrelse og hvordan de placeres i containeren, nedenstående tabel 3.5 viser de forskellige layout managers AWT tilbyder.

Tabel 3.5: AWT 1.1 layout managers

BorderLayout
CardLayout
FlowLayout
GridBagLayout
GridLayout

AWT tilbyder et rimeligt sæt klasser til at designe GUI med, men er man vant til andre GUI biblioteker, kan tilbuddet af klasser i AWT forekomme en smule begrænset, men medfører derimod et mere overskuelig API.

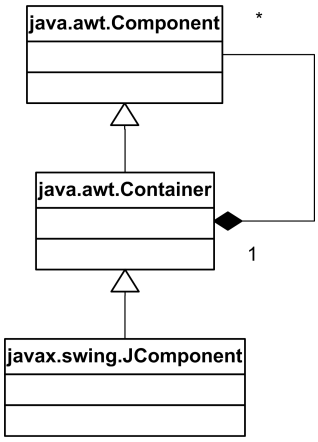
AWT er ikke længere det primære Java bibliotek til konstruktion af grafiske brugergrænseflader, det er er erstattet af Swing, man kan dog stadig benytte AWT komponenter om man vil.

3.1.2 Swing

Det officielle overordnede designmål for Swing er: *"To build a set of extensible GUI components to enable developers to more rapidly develop powerful Java front ends for commercial applications."* [11]

Swing er udviklet af Sun Microsystems, og er det primære bibliotek til udvikling af grafiske brugergrænseflader til Java applikationer. Swing blev en del af Java fra version 1.1 af Java Development Kit (JDK), som en del af Java Foundation Classes (JFC) [25]. Swing er baseret på dele af Abstract Window Toolkit (AWT) [23], og benytter geometriske primitiver og metoder fra Java 2D biblioteket [24] til at tegne sine komponenter med, hvilket i forhold til AWT, gør dem til lightweight komponenter. Swing er skrevet i ren Java.

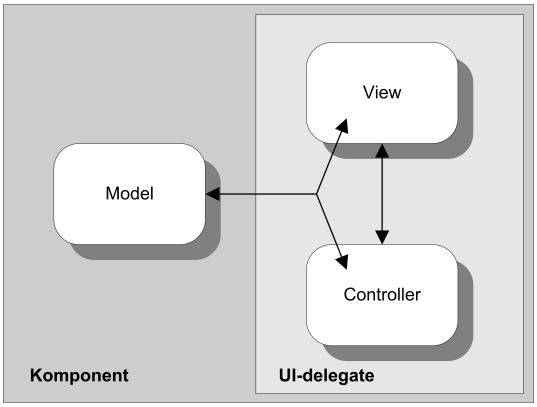
Swing er et stort bibliotek, som er udgjort af mere end 500 klasser og interfaces (klasser og interfaces defineret i javax.swing pakken og underpakker alene). Swing er et meget komplet bibliotek til konstruktion af grafiske brugergrænseflader og det understøtter et væld af komponenter. Man kan specialisere de eksisterende komponenter, og konstruere egne nye komponenter [20].



Figur 3.1: Swing og AWT relation. Komponent composite design mønster.

Swing biblioteket har et meget generelt design der gør, at biblioteket sjældent vil sætte grænser for, hvad der kan lade sig gøre at lave, men som til gengæld kræver, at man er bekendt og fortrolig med Swing's arkitektur og grundidéerne i dets opbygning. I Swing, jf. figur 3.1 nedarver alle komponenter fra `JComponent` klassen. `JComponent` nedarver fra AWT's `Container` klasse, som igen arver fra `Component` klassen. En container er en komponent, der kan indeholde andre komponenter.

Alle Swing komponenter er opdelt i en model- og en User Interface-del (UI) jf. figur 3.2. Modellen indeholder de data, der beskriver den enkelte komponents tilstand (eksempelvis elementerne i en liste, teksten på en knap osv.). UI-delen varetager tegning af komponenten, samt bestemmer hvordan den skal reagere på input. Ved ændringer til modellen opdateres visningen af komponenten og omvendt. Adskillelse i model- og UI-del betyder, at visningen af komponenten kan ændres uafhængigt af modellen. Endvidere kan en tilstrækkelig generel model tilknyttes forskellige visninger.



Figur 3.2: Opdeling af komponenter i model og UI del

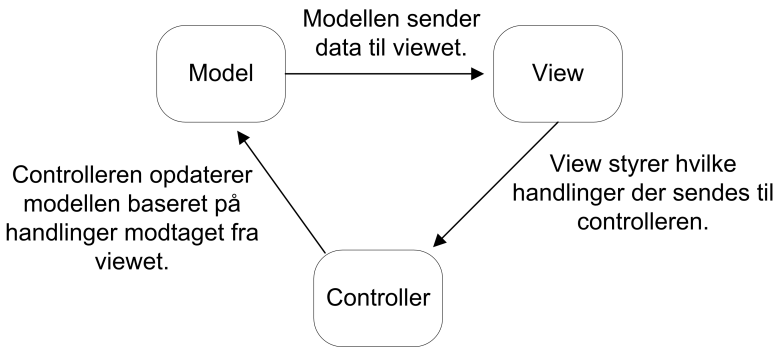
Teknikken baseres på designmønstret model-view-controller (MVC), der stammer fra SmallTalk [18]. MVC mønstret består af følgende dele og kommunikationen derimellem ses på figur 3.3:

Model Data applikationen manipulerer.

View Den grafiske repræsentationen af modellen (der kan eksistere flere views for samme model).

Controller Håndterer bruger input der modificerer modellen.

Forskellen mellem Swing's MVC designmønster og det originale MVC designmønster fra SmallTalk, er relationsforholdet elementerne imellem (model, view og controller). Hvor der i Swing's version eksisterer en stærk kobling elementerne imellem, er koblingen i SmallTalk's version mindre stærk.



Figur 3.3: Model-View-Controller kommunikation

Ligesom i AWT sker placering af komponenter ved brug af layout managers. Swing tilbyder dog to nye, så sammenlagt med AWT er der ialt syv forskellige layout managers jf. tabel 3.6, hvoraf de to markeret med fed skrift stammer fra Swing.

Tabel 3.6: Swing standard layout managers

BorderLayout
BoxLayout
CardLayout
FlowLayout
GridBagLayout
GridLayout
SpringLayout

Som i AWT 1.1 baseres events i Swing ligeledes på lyttere.

3.1.3 SWT

Standard Widget Toolkit [8] (SWT) er et andet platformsuafhængigt bibliotek til Java, skabt af Eclipse til brug for udvikling af grafiske brugergrænseflader. Udover at SWT er platformsuafhængigt er det ydermere uafhængigt af Swing og AWT.

SWT er skrevet i Java, og benytter JNI (Java Native Interfaces) som interface til kald til operativsystemet. Den primære forskel mellem Swing og SWT er, at SWT anvender platformsspecifikke komponenter hvor det er muligt, og kun hvis en platformsspecifik komponent ikke eksisterer på den pågældende platform, benyttes en lightweight udgave af komponenten. Det sidste er præcis forskellen på AWT og SWT. I AWT måtte man anvende "laveste fællesnævner", altså kun inkludere de komponenter og funktioner i API'et som var tilgængelige på alle platforme. SWT placerer sig således et sted mellem AWT og Swing.

Motivationen for brugen af SWT er at få Java applikationer til at ligne platformsspecifikke programmer, både hvad angår udseende og performance. SWT er et alternativ til Swing for dem der ikke bryder sig som udseendet af Swing's lightweight komponenter, eller som ikke er tilfreds med Swing's performance. Der er performancetests af SWT og Swing, der svinger begge veje. Det virker som om, SWT's fordel ved at trække på de platformsspecifikke resourcer udjævnes, hvis GUI'ens datamodel bliver tilstrækkelig stor. Da skal denne jo kopieres i SWT, mellem operativsystemet og JVM (Java Virtual Machine).

SWT tilbyder med få undtagelser de samme typer komponenter som Swing, og SWT anvender også layout managers til placering af komponenterne. Funktionaliteten i SWT er dog mere begrænset end i Swing. Ting som *cell renders* og data modeller er ikke en del af SWT, men er istedet inkluderet i et ekstra bibliotek til SWT kaldet JFace. JFace er et bibliotek, der udvider funktionaliteten af SWT, og gør trivielt og hyppigt anvendt GUI funktionalitet hurtigere at programmere.

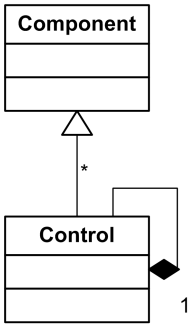
Har man været vant til at benytte Swing og skal skifte til SWT, vil man umiddelbart savne en del funktionalitet, særligt MVC designet fra Swing. Dette problem kan dog løses med `JFrame` og `ContentProviders`. Funktionaliteten af SWT mere begrænset, og gør SWT lettere at gå til end Swing primært på grund af den mere simple API, hvor der er lagt vægt på at inkludere og supportere kun typisk anvendt funktionalitet.

3.1.4 Windows Forms

Microsofts nye platform .NET tilbyder to bibliotekter til at lave GUI med, Windows.Forms og Web.Forms. Afsnittet beskriver førstnævnte bibliotek, og det efterfølgende afsnit beskriver forskellen imellem de to, da ligheden er stor.

Windows Forms benyttes til at programmere Microsoft Windows applikationer (WIN32) med, og kræver at .NET frameworket er installeret (der er dog nogle igangværende projekter der forsøger at implementere .NET til andre platforme end Microsoft Windows).

Ligesom i Swing indeholder Windows Form et stort udvalg af grafiske komponenter, til brug for konstruktion af grafiske brugergrænseflader, og også her findes Composite strukturen (tildels). Alle grafiske elementer arver fra Windows Forms `Control` klasse, som igen arver fra en `Component` klasse, hvilket gør alle grafiske elementer til kontroller, jf. figur 3.4. Grafiske elementer tilhører som minimium en kontrol (det yderste vindue), men kan sagtens indlejres i en række andre kontroller. Kontrollen elementet ligger i kaldes containeren.



Figur 3.4: Sammenhæng mellem Component og Control

Placering af grafiske komponenter sker ved direkte angivelse af koordinator (absolutpositionering med to punkter, der angiver komponentens start position øverst i venstre hjørne, i den kontrol komponenten ligger), der tilbydes ingen form for layout managers som i AWT og Swing. Muligheden for selv at udarbejde layout managers til Windows Forms eksisterer, idet layout managers blot er et lag over den absolut positionering. Grundet de værktøjer Microsoft og andre leverandører stiller til rådighed, vil det dog i de fleste tilfælde ikke være relevant (mere om dette i afsnit 3.2).

Windows Forms komponenter besidder to eksplicitte egenskaber til at håndtere resizing af vinduer, i modsætning til Java, hvor dette er en del af layout manageren. Den første egenskab er *anchors*, der benyttes til at fastlåse komponentets sider i forhold til dets container. Mulighederne er top, bund, højre og venstre – man kan benytte nul eller flere af disse. Som standard er top og venstre valgt, hvilket betyder at elementet ikke flytter sig hvis vinduet resizes. Den anden egenskab er en *dock*, der benyttes til at placere komponenten, som ved et `BorderLayout` i Java. Dette betyder at komponenten enten kan placeres i containerns venstre side, højre side, top eller bund.

Sammenlignet med Java, er ansvaret for komponents placering og opførelse flyttet fra containeren til komponenten selv.

En anden væsentlig forskel mellem de to platforme, er den manglende model-view-controller implementering i Windows Forms. Manglen ligger i synkroniseringen mellem modellen og viewet. Når eksempelvis en liste (`ListBox`) fyldes op med objekter, genereres et array med strenge på baggrund af objekternes `ToString()` metode til brug for visning i listen. Arrayet ændrer ikke tilstand på trods af at objekternes tilstand ændres, `ToString()` metoden genererer altså derfor et andet output end tidligere. Selv om en `ListBox` er ekstrem nem at benytte set i forhold til Swing's `JList`, er konsekvensen af den manglende MVC effekt under al kritik. Ændres et objekt fra modellen, (indholdet af listen) har det ingen effekt på viewet (præsentationen af listen), hvilket betyder at det visuelle indhold af listen ikke nødvendigvis

er korrekt, såfremt indholdet heraf er blevet ændret. Dog er det muligt, at tvinge listen til at opdatere viewet så den er, synkron med modellen, ved at indsætte objekterne i listen på ny. Konkret går kritikken til `ListBox` på view og model ikke automatisk holdes synkron eller kan synkroniseres via en simple `updateView()` metode, men at objekterne istedet skal genindsættes i listen for at få den ønskede effekt.

Events kan anskues fra tre perspektiver, implementering, anvendelse og informationen omkring eventet selv. Hvor første- og andennævnte konstrueres anderledes i .NET (Windows Forms og Web Forms) end i Java, er ligheden i sidstnævnte stor. I Java gemmes alt information omkring eventet i en underklasse til enten `java.awt.AWTEvent` eller `javax.swing.event`. Herigennem kan der skabes en reference til selve komponenten (kilden til eventet), og det kan f.eks. afgøres, hvilken museknap der blev trykket på. Samme information findes i .NET, men splittes her op i to objekter. Der hvor de to platforme adskiller sig, er på implementerings- og anvendelsesniveau, hvor sidstnævte jf. specialets formål, er den mest interessante. De grafiske komponenter har en række felter, tilsvarende de events de kan udfører, hvorpå man kan tildele metoder. Ved udførelse af et event kaldes den tildelte metode, jf. kodeeksempel 3.1. .NET implementerer event mekanismen ved brug af delegater (delegates). Et delegat er en referencetype, der bruges til at indkapsle metoder med bestemte signaturer og retur typer [31][19].

Kode 3.1: Håndtering af klik på knap i .NET

```
1    MyButton.Click += new System.EventHandler(this.MyButton_Click);
2
3    private void MyButton_Click(object sender, System.EventArgs e)
4        {
5            // Do something
6        }
```

3.1.5 Web Forms

Web Forms, også kendt som ASP.NET, den næste generation af ASP (Active Server Pages), benyttes til at udarbejde web baseret browser applikationer. Udover det miljø som de to platforme afvikles på er forskellig, ligger den hovedsaglige forskel, set udfra et GUI perspektiv, i opbygningen af den grafiske brugergrænseflade. Web Forms består i sin nuværende version af to dele, en designdel der indeholder den grafiske brugergrænseflade og en kodedel (også kaldet *code behind*), der indeholder logikken. GUI'en laves udfra klassisk HTML kode [33], hvor man ved hjælp af specielle mærker (på engelsk *tags*), jf. nedenstående kodeeksempel 3.2, kan indsætte Web Forms komponenter.

Kode 3.2: HTML tag for Web Forms komponent (Button)

```
1    <asp:Button id="MyButton" runat="server" Text="Click me"></asp:Button>
```

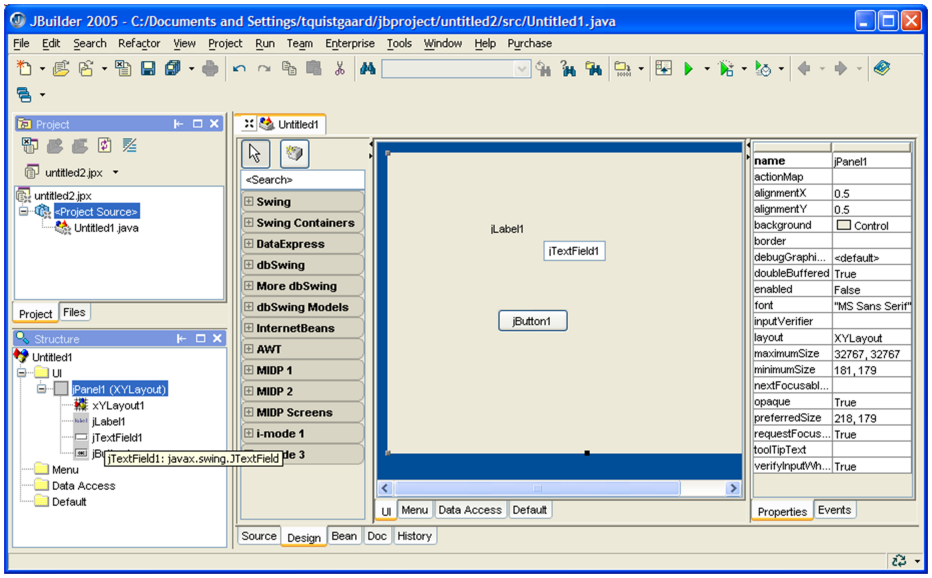
Fra kodedelen er det dermed muligt at arbejde med komponenten, samt tilføje og håndtere eventuelle events.

3.2 Værktøjer

Ved brug af værktøjer stiger abstraktion, jf. tabel 3.1, væk fra programmering af GUI til automatisk opbygning heraf. Jeg har valgt kun at undersøge decideret GUI builders, hvilket refererer til værktøjer hvorved den grafiske brugergrænsefalde hovedsagligt laves ved *"drag and drop"*.

3.2.1 JDeveloper og JBuilder

Jeg har undersøgt to GUI builders til Java som begge minder om hinanden; JDeveloper fra Oracle [28] og JBuilder fra Borland [3] hvoraf et screendump for JBuilder ses på figur 3.5.



Figur 3.5: JBuilder

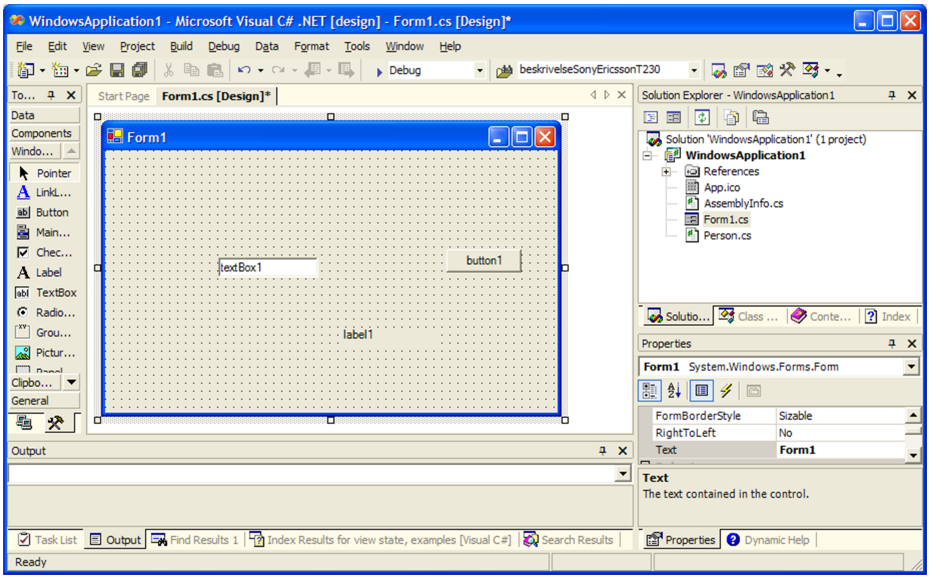
Begge værktøjer anvender de eksisterende komponenter fra Swing og AWT til at konstruere GUI'en med, samtidig med at de understøtter et "drag and drop" miljø for en række af de inkorporerede layout managers i Java. Efter min mening er sammenlægning af de eksisterende layout managers med et "drag and drop" miljø ikke anvendeligt. Et "drag and drop" miljø skal placere komponenter præcist hvor de trækkes hen, og ikke fylde hele toppen, bunden, højreside, venstreside eller midten som ved anvendelse af `BorderLayout`.

Heldigvis har begge værktøjer implementeret deres egen layout manager, som anvender absolutpositionering til at placere komponenter med, hvilket gør værktøjerne og "drag and drop" funktionen anvendelige. Ulempen herved er, at applikationen der udvikles ved brug af værktøjernes egne layout managers, nu ligeledes er afhængige af disse og kan ikke afvikles uden dem.

Udover at værktøjerne fokuserer på at simplificere GUI opbygning, simplificerer de ligeledes eventhåndtering, ved eksempelvis at lade værktøjet danne skeletkoden til håndtering af events, ved blot at dobbeltklikke på de grafiske komponenter. Derudover gør værktøjerne det nemt at ændre egenskaberne for de grafiske komponenter via en manipulerbar visualisering af disse.

3.2.2 Visual Studio .NET og Delphi 2005

Som ved de to forgående værktøjer er ligheden mellem Visual Studio .NET fra Microsoft [22] og Delphi fra Borland [2] tilsvarende, figur 3.6 viser et screendump fra Visual Studio .Net. De to værktøjer som her er beskrevet, bruges til at programmere Windows Forms og Web Froms applikationer.



Figur 3.6: Visual Studio .NET

Forskel mellem Visual Studio .Net og Delphi, set i forhold til de to tidligere værktøjer 3.2.1, er layoutdelen. Hvor Visual Studio .Net og Delphi kun understøtter absolutpositionering af komponenter, understøtter JDeveloper og JBuilder derudover en række layout managers. I kraft af at Windows Forms biblioteket kun arbejder med stategi omkring absolutpositionering og ikke anvende Java's strategi omkring layout managers, er de to beskrevne værk-

tøjer, et nødvendig onde for at programmere Windows Forms applikationer (uden brugen heraf ville placering af komponenter være en langvarig proces). Situationen er en anden ved programmering af Web Forms, idet der er tale om web applikationer og dermed HTML.

3.3 Eksisterende tiltag

Jeg har undersøgt to biblioteker, henholdsvis CookSwing [34] og Buoy [6], hvilke begge forsøger at simplificere GUI konstruktionsprocessen forbundet med Swing. De to biblioteker er lavet som open source projekter, og kan begge findes på sourceforge.net. Udover de to biblioteker eksisterer der et hav af andre forsøg på at simplificere GUI programmering med Swing, såsom JellySwing [9], Luxor [1], osv. hvoraf jeg kun har undersøgt CookSwing og Buoy.

3.3.1 CookSwing: XML to Swing GUI

CookSwing er et bibliotek, som bygger Java Swing GUI udfra et XML dokument. Frem for at skrive GUI kode i selve klassen, skrives den i et separat XML dokument. Syntaksen for XML'en og de dertilhørende tags og attributter er Swing's klassenavne, offentlige feltnavne og set metoder omdannet til XML-tags (hvor J'et er fjernet). Attributterne der kan sættes for et objekt og dermed benyttes i XML'en, svarer direkte til de eksisterende offentlige felter og set metoder for objektet. Kodeeksempel 3.3 viser et simpelt XML dokument for et "hello world" eksempel, hvor der benyttes en `JFrame` og en `JLabel`.

Kode 3.3: XML syntaks for CookSwing

```
1    <frame title="Hello World!" size="640,480">
2        <label text="Hello World!" horizontalalignment="CENTER"
3               foreground="#ff0000" font="Serif,bold italic,20" />
4    </frame>
```

Måden hvorpå XML dokumentet omdannes til konkrete Swing komponenter foretages i CookSwing bibliotekets `render(...)` metode, hvilket betyder at XML'en gennemløbes og behandles runtime. Kodeeksempel 3.4 viser erklæringerne som omdanner et XML dokument til et faktisk GUI vindue.

Kode 3.4: Initialisering af XML dokument

```
1    CookSwing cookSwing = new CookSwing();
2    cookSwing.render("helloworld.xml").setVisible(true);
```

CookSwing's strategi til at håndtere fejl i XML'en når det gennemløbes runtime, er at ignorere fejlen, så hvis en attribut er stavet forkert eller attributten ikke eksisterer på objektet ignoreres denne blot. Det samme gælder

for hele XML-tags, støder CookSwing på et tag som ikke svarer til et Swing objekt ignoreres dette.

XML'en som benyttes i CookSwing minder meget om Swing (en næsten direkte kopi), hvilket også gælder for layout managers. Imens XML'en giver en form for strukturering af komponenterne, styres deres fysiske placering af layout manager XML-tags, der virker præcist som i Swing. Kodeeksempel 3.5 viser hvordan et `BorderLayout` benyttes i XML'en.

Kode 3.5: Layout manager XML-tag

```
1  <frame>
2      <panel>
3          <borderlayout>
4              <constraint location="North">
5                  <button text="North" />
6              </constraint>
7          </borderlayout>
8      </panel>
9  </frame>
```

Sidste element i CookSwing er eventhåndtering som består af to dele, registrering af eventet på de grafiske komponenter og oprettelsen af selve eventet. Kodeeksempel 3.6 og 3.7 viser hvordan de to dele erklæres.

Kode 3.6: Eventregistrering i XML'en

```
1  <button text="action button" actionlistener="buttonAction" />
```

Kode 3.7: Eventregistrering i klassen

```
1  public ActionListener buttonAction = new ActionListener () {
2      public void actionPerformed (ActionEvent e) {
3          System.out.println("button action");
4      }
5  };
```

I klassen registreres en offentlig `ActionListener` (lyttere der erklæres i en klasse og som skal tilknyttes komponenter i XML'en, skal erklæres offentlige gundet implementeringen af CookSwing) som forbindes med knappen i XML'en via referencevariablen `buttonAction` og XML-attributten `actionlistener`. Yderligere kræves det at `CookSwing` instansen jf. kodeeksempel 3.4 linje 1 initialiseres med objektet, hvorpå den offentlige lytter (`ActionListener`) findes se kodeeksempel 3.8.

Kode 3.8: Initialisering af XML dokument med lyttere

```
1  CookSwing cookSwing = new CookSwing (this);
```

"The fundamental goal of CookSwing is to provide specific supports for GUI packages and free up programmers from doing tedious GUI hard coding and cleanup the code, WITHOUT compromising functionality." [34]

Ovenstående citat er målet bag CookSwing, hvilket jeg ikke mener realiseres. Derimod skabes et mere trættende GUI kodningsforløb for programmører, da fejlfindingsprocessen flyttes til runtime fremfor compiletime. For overhovedet

at kunne gå på kompromis med runtime fejl fremfor compiletime, burde CookSwing have fundet en simplere GUI syntaks for XML'en, istedet for blot af "XMLificere" Swing. Grundlæggende synes jeg dog konceptet omkring XML, og dermed det klart afspejlede forhold mellem komponenterne er en god ide, samtidig med at det åbner muligheder for, at skifte kørende GUI ud med nyt uden genkompilering er nødvendig.

3.3.2 Buoy: A Better User Interface Toolkit

Ligesom CookSwing bygger Buoy ovenpå Swing, men benytter ikke en XML-orienteret tilgang til GUI konstruktion. Buoy benytter samme tilgang som Swing til at konstruere GUI med, men bruger et sæt Buoy komponenter istedet for Swing komponenter. Forskellen mellem Buoy og Swing komponenter er antallet af deres egenskaber (felter og metoder), hvor der i Buoy forsøges at minimere disse i forhold til Swing.

Kodeeksempel 3.9 viser konstruktionen af et "hello world" eksempel med Buoy. Som det fremgår af koden minder konstruktionen meget om en Swing konstruktion, men er forskellig på tre måder; (1) udover at komponenterne hedder noget andet (og har et færre egenskaber), håndterer Buoy også (2) layout managers og (3) events anderledes end Swing.

Kode 3.9: Buoy kode

```
1  public class HelloWorld
2  {
3    public static void main(String args[])
4    {
5      BFrame f = new BFrame();
6      BorderContainer content = new BorderContainer();
7      f.setContent(content);
8      content.add(new BLabel("Hello World!"), BorderContainer.NORTH);
9      BButton b = new BButton("OK");
10     content.add(b,BorderContainer.SOUTH);
11     b.addEventLink(CommandEvent.class, new Object() {
12       void processEvent() {
13         System.exit(0);
14       }
15     });
16     f.pack();
17     f.setVisible(true);
18   }
19 }
```

Buoy har sammenlagt begrebet layout manager med containere, så frem for at tilføje layout managers til containere, styres placeringen af komponenter af containere. I kodeeksemplet linje 6 benyttes en `BorderContainer`, der implicit er sammenlagt med en `BorderLayout` layout manager, til at indeholde og styre placeringen af labelen og knappen.

Den sidste forskel mellem Swing og Buoy er eventhåndtering, hvor der i Swing benyttes en metodeorienteret tilgang, til at definere både eventtypen samt det specifikke event, benyttes der i Buoy en klasseorienteret tilgang. Istedet for at have en række metoder, der muliggører at tilføje diverse lyttere

og adaptere, benyttes en generisk metode i Buoy `addEvent(...)`, der som parameter tager en type, hvorfra det specifikke event bestemmes. Metoden findes i tre former, hvoraf den vist i eksemplet kræver at anden parameter skal implementere en `processEvent()` metode, hvilket er metoden der eksekveres ved udførelse af eventet.

"Why would you want to use Buoy instead of the standard Swing API? Quite simply, because it's better. It is easier to learn and easier to use. It lets you do what you want to do with fewer lines of code. It produces programs which are simpler, more readable, and easier to maintain." [6]

Citatet er taget fra en introduktionstekst til Buoy, hvor grunden til at anvende Buoy fremfor Swing ikke underbygges yderligere videnskabeligt. Hvem siger for eksempel at færre antal linjer kode formindsker kompleksitet? Dog mener jeg at Buoy giver et reelt alternativ til Swing, som jeg personligt mener er en smule simplere at anvende, men ikke simpelt nok idet Buoy stadig minder for meget om Swing og dermed måde at opbygge GUI på.

Kapitel 4

Problemformulering

4.1 Klassifikation af problemer med GUI programmering

Et typisk programmeringsforløb af grafiske brugergrænsefalder programmer indebærer (1) placering og gruppering af grafiske komponenter i forhold til hinanden, (2) tildeling af de rigtige farver, størrelser og placering af komponenter i forhold til de paneler de ligge i, (3) sammenkobling af handling og programlogik der skal udføres ved klik og valg af komponenter og (4) håndtering af tilstande mellem objekter og deres præsentation (gennem komponenter).

De fire punkter afspejler en opdeling for GUI programmeringen og udgør derigennem en klassifikation af problemer som alle GUI bibliotketer skal tage højde for.

Ser vi på de fire nævnte klassifikationer i forhold til bibliotekerne undersøgt i forrige kapitel og vurderer anvendelsen heraf, kan følgende tabel 4.1 opstilles. Tabellen viser min subjektive vurdering af hvor nemt det er at anvende biblioteksterne i forhold til de fire opstillet klassifikationer, hvor AWT, Swing og SWT bibliotekerne er slået sammen til et, idet anvendelsen heraf næsten er identisk.

Tabel 4.1: Anvendelsesorienteret vurdering af bibliotekerne

Bibliotek	(1) Struktur	(2) Layout	(3) Event	(4) MVC
AWT, Swing, SWT	÷	÷	÷	÷
Windows Forms	÷	+	+	÷
Web Forms	+	+	+	÷
CookSwing	+	÷	÷	÷
Buoy	÷	÷	÷	+

Som det fremgår af tabllen kan Swing vurderes som ikke let anvendelig, hvorimod Web Forms kan vurderes som let anvendelig. Grunden til den høje grad af anvendelighed for Web Forms, skyldes at det er beregnet til Web, hvilket betyder at der anvendes HTML til at opbygge GUI'en med. HTML giver en klar gruppering af grafiske elementer i form af dens XML agtige opbygning og gennem Cascading Style Sheets (CSS) [32] er det let at angive farver, størrelser og præcise placeringer for de grafiske elementer. CookSwing benytter sig ligeledes af XML til gruppering af komponenter, men er svagt stillet i de tre andre kriterier, hvorfor jeg blandt andet mener at CookSwing ikke formår at bidrage til en simplificering af GUI konstruktion. I modsætning til CookSwing gør Buoy et konrekt forsøg på at simplificere Swing gennem indpakningen heraf, men trods tiltaget mener jeg stadig at Buoy's bibliotek minder for meget om Swing set udfra et anvendelsessynspunkt. Windows Forms og Web Forms er de eneste to biblioteker der formår at håndtere events på sådan en måde, at alle kan være med. Anvendelse heraf er intuitiv og ligetil.

Efter at have arbejdet med syv forskellige biblioteker og set på både de positive og negative sider udfra fire kriterier, kan det konkluderes at ingen af bibliotkerne i sig selv er perfekte (lette at anvende). En kombination af disse vil derimod skabe et let anvendeligt bibliotek til brug for konstruktion af grafiske brugergrænseflader.

4.2 Problemformulering og overordnet målsætning

Jeg vil med anvendelse af Java 1.5 reducere kompleksiteten forbundet med programmering af grafiske brugergrænseflader for Swing. Konkrekt vil jeg udarbejde en arkitektur, der baseres ovenpå Swing, hvilket resulterer i et biblioteksdesign og en proof-of-concept implementering. Den overordnet målsætning for biblioteket er "simplicity", hvorfor biblioteket skal kunne indsættes i tabel 4.1 og overholde samtlige kriterier.

Målgruppen for biblioteket, jf. min egen baggrund 1.1, er begyndende/nyudklækket Java programmører, der som jeg selv, ikke kunne forstå hvorfor det skulle være så kompliceret at lave GUI med Swing.

Mit fokus vil være områderne jeg har analyseret (de fire klassifikationer jf. afsnit 4.1) og som bidrager væsentligt til den eksisterende kompleksitet i Swing. (1) Strukturering af komponenter, (2) layout af komponenter, (3) eventhåndtering samt (4) MVC strukturen. Med viden samt inspiration baseret udfra analysen, vil jeg i næste kapitel konkretisere en række problemer med Swing. Udfra de forskellige problemer opstiller jeg en række mål, hvis formål tjener til reducering af kompleksiteten for GUI programmering med Swing.

4.3 Afgrænsning

Kun Java 1.5

Alt udvikles indenfor rammerne af Java 1.5. Hvilket betyder at det eneste krav, for at arbejde med biblioteket, udover biblioteket selv, er JDK 5.0 (J2SE Development Kit). Specialet tager dog udgangspunkt i muligheder i Java 1.5 beta 1, idet introduktionen jf. mine forudsætninger lå heri.

Værktøjer

Da antallet og ligheden mellem de forskellige værktøjer (GUI builders) er stor, afgrænser jeg mig fra at undersøge og behandle disse yderligere.

Kun GUI konstruktion

Givet min målsætning og målgruppe, vælger jeg ikke at analysere og behandle problemer med Swing, hvis natur ligger udenfor området GUI konstruktion. Herunder hører performance, der ellers er et af højdepunkterne indenfor Swing debatter.

Delmængde af Swing elementer

Jeg har valgt at behandle følgende grafiske Swing elemeter jf. tabel 4.2. Valget af netop disse komponenter skyldes at de repræsenterer de mest basale dele i Swing. `JFrame` og `JPanel` er de komponenter der danner rammerne for en GUI, hvor `JTextField`, `JLabel` og `JButton` blot er valgt for at

kunne skabe en prototype, hvormed der kan skabes en nogenlunde meningsfuld GUI, elementer som `JCheckBox` og `JRadioButton` kunne have været tilsvarende kandidater. Til sidst har jeg valgt `JList` som en repræsentation for et element, der kan rumme og præsentere objekter.

Tabel 4.2: Swing elementer.

JFrame
JPanel
JTextField
JLabel
JButton
JList

Kapitel 5

Designmål

I kapitel 4 blev der opstillet fire fokusområder: Struktur, Layout, Events og MVC. På baggrund af fokusområderne, opstilles i følgende kapitel en række designmål, som arkitekturen for HGL skal indfri. Designmålene baseres på en række problemområder jeg har fundet i Swing, og skal fungere som retningslinjer for reduceringen af kompleksitet forbundet med GUI programmering i Swing. De opsatte mål skal derfor ligeledes fungere som en succesindikator for den overordnet målsætning. Udover de fire fokusområder opstilles en række generelle designmål gældende for hele arkitekturen.

Alle designmål i kapitlet angives med fed skrift i hver delafsnit.

5.1 Generelle

Jeg har opstillet en række generelle designmål for hele arkitekturen. Disse gælder for et eller flere af efterfølgende klassificeringer, hvorfor de betragtes som generelle.

Biblioteket og brugen heraf skal ske inden for de eksisterende rammer af Java 1.5

Arkitekturen der udformes og brugen heraf, skal ske indenfor rammerne af Java 1.5 hvilket betyder, at eksterne biblioteker og værktøjer ikke tages i brug. Designmålet er todelt, da det samtidig afspejler en afgrænsning jf. afsnit 4.3.

API skal minimaliseres

Swing er et meget omfattende bibliotek med et tilsvarende stort API, og som bruger af Swing kan mængden af funktionalitet forekomme uoverskuelig. Tag blot en `JLabel` der indeholder ca. 340 public metoder. For en garvet Java GUI programmør er dette intet problem, da en `JLabel` jo arver fra `JComponent`, der arver fra `Container`, der arver fra `Component` som til sidst arver fra `Object`, hvorfor en `JLabel` selvfølgelig rummer 340 metoder. For en nybegynder kan dette måske virke forvirrende, hvorfor jeg ønsker at skabe et bibliotek med et minimalistisk API. Udover et mere overskueligt API vil resultatet som sideeffekt kunne minimere antallet af kodefejl, da antallet af mulige kodekonstruktioner reduceres.

Alle fejl skal fanges ved kompileringstid

At alle fejl skal fanges ved kompilering er ikke et specielt kritikpunkt for Swing men blot god kodeskik, hvorfor jeg stræber efter dette. Ideen med at fange fejl ved kompilering frem for runtime, betyder at fejl fanges på udviklerens computer frem for på slutbrugerens (runtime fejl).

5.2 Struktur

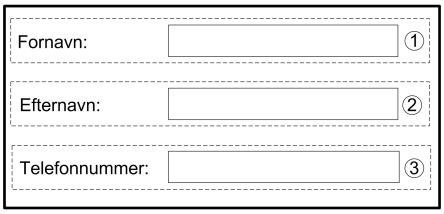
Struktur definerer, hvordan grafiske komponenter placeres i forhold til hinanden. Struktur definerer ikke, hvor komponenterne præcist placeres, hvilket håndteres i layoutdelen, men danner de ydre rammer for, hvor de kan placeres. Den væsentlige pointe her er tilhørsforholdet komponenterne imellem, hvilket er det struktur omhandler.

I forhold til strukturfænomenet ser jeg to problemer med Swing.

Forholdet mellem komponenterne skal fremgå klart og logisk i koden

For mange grafiske brugergrænseflader er det nemmest at bryde designet op i mindre dele, for til sidst at samle disse. I Swing er den primære ressource til dette et panel. Paneler er usynlige enheder, hvis formål er at gruppere komponenter. Nedenstående figur 5.1 viser tre paneler (de stiplede kasser), der hver især indeholder en label og et tekstfelt. Selvom eksemplet er simpelt viser det idéen om anvendelse af paneler som en container for komponenter.

Jeg synes Swing's (og AWTs) ide omkring container er god og den korrekte vej frem. Der hvor problemet opstår er på implementeringsniveauet.



Figur 5.1: Brug af paneler

Abstraktionsgabet mellem hvordan koden ser ud, i forhold til hvordan en programmør forestiller sig komponenter grupperet er stor. En mulig implementering af figur 5.1 kunne realiseres som kodeeksempel 5.1. Her afspejles forskellen klart mellem ens intuition og en konkrete implementering. I linje 2 oprettes det første panel og i linje 6 og 10 oprettes henholdsvis den label og det tekstfelt, panelet skal indeholde. Men de tilføjes først i linje 14 og 17. Et panel symboliserer en logisk gruppering, men kan realiseres på en måde der er langt fra logisk. Eksemplet viser nærmere en spredning frem for en gruppering.

Kode 5.1: Implementering af panel eksempel

```
1         ...
2         JPanel panel1 = new JPanel();
3         JPanel panel2 = new JPanel();
4         JPanel panel3 = new JPanel();
5
6         JLabel label1 = new JLabel("Fornavn:");
7         JLabel label2 = new JLabel("Efternavn:");
8         JLabel label3 = new JLabel("Telefonnummer:");
9
10        JTextField text1 = new JTextField();
11        JTextField text2 = new JTextField();
12        JTextField text3 = new JTextField();
13
14        panel1.add(label1);
15        panel2.add(label2);
16        panel3.add(label3);
17        panel1.add(text1);
18        panel2.add(text2);
19        panel3.add(text3);
20        ...
```

Hvorfor denne spredning er mulig, skyldes at Swing kodes imperativt, hvilket kan resultere i ikke-intuitive kodekonstruktioner. Endvidere, udvides kodeeksemplet jf. kodeeksempel 5.2 vil den intuitive kompleksitet stige og en ikke direkte synlig hierarkisk struktur dannes. Hvilke komponenter og container der henholdsvis er forældre og børn til hinanden, afspejles svagt i koden.

Kode 5.2: Implementering af panel eksempel - udvidelse

```
1         panel1.add(panel2);
2         panel2.add(panel3);
3         ...
```

Selvfølgelig kan koden opbygges anderledes jf. kodeeksempel 5.3, hvor der ved hjælp af tabulering dannes det ønskede hierarki. Dog løser tabuleringen ikke målet, idet muligheden for at skrive kode, som i det tidligere eksempel,

stadig er tilstede. Det ønskværdige er et GUI bibliotek med en syntaks, hvor det fremgår klart og tydeligt, hvordan de forskellige grafiske komponenter hører sammen, såvel intuitivt som kodemæssigt.

Kode 5.3: Implementering af panel eksempel - anderledes

```
1         JPanel panel1 = new JPanel();
2             JLabel label1 = new JLabel("Fornavn:");
3             panel1.add(label1);
4             JTextField text1 = new JTextField();
5             panel1.add(text1);
6             JPanel panel2 = new JPanel();
7             panel1.add(panel2);
8                 JLabel label2 = new JLabel("Efternavn:");
9                 panel2.add(label2);
10                JTextField text2 = new JTextField();
11                panel2.add(text2);
12                JPanel panel3 = new JPanel();
13                panel2.add(panel3);
14                    JLabel label3 = new JLabel("Telefonnummer:");
15                    panel3.add(label3);
16                    JTextField text3 = new JTextField();
17                    panel3.add(text3);
```

Målet er altså, at mindske abstraktionen mellem koden og det intuitive og dermed danne en én-til-én relation mellem disse, samt at give et klart overblik, over hvilke komponenter der er henholdsvis forældre og børn til hinanden.

Fordelene ved sådan et mål er indlysende. En deklarativ tilgang fremfor en imperativ hjælper brugeren til nemt og hurtig at kunne gennemskue den grafiske opbygning udfra koden, uden at skulle se det visuelle resultat først.

Strukturen skal være rigid. Placering af komponenter ulogiske steder skal ikke tillades

Alt i Swing (her menes grafisk komponenter som tekstfelter, labels, mm.) er både AWT `Component's` og `Container's`, hvilket betyder at der er mulighed for, at lave en grænseflade bestående af knapper inden i knapper. Selvom dette ikke er et reelt problem, er princippet om denne fleksible opbygning, i mine øjne et problem. Såvel dette som ovenstående problem med Swing, kan "løses" ved at stille æstetiske kodekrav. Dog mener jeg ikke, at dette er tilstrækkeligt, da sådanne krav afhænger af menneskelige faktorer. Hvis biblioteket istedet tvang brugere til at skrive "korrekt" kode, kunne både abstraktionen og fleksibiliteten derimod styres på maskinniveau frem for på det menneskelige.

Målet er et strammere regelsæt, hvor placering af komponenter inde i komponenter kun tillades, hvor det giver mening. F.eks. giver en knap placeret i en knap ikke mening. En rigid struktur sætter faste rammer og udelukker muligheder. Fordelen er en stor formindskelse af antallet af kodefejl, jo færre muligheder, jo færre fejl.

5.3 Layout

Layout definerer komponenternes udseende og deres placering indenfor strukturen. Dette kunne eksempelvis være om en given komponent højrestilles i det panel den ligger placeret.

Layout managerene skal elimineres for brugerne

Swing's svar på layout er layout managers, som i forlængelse med struktur fuldender måden at placere grafiske komponenter på. Layout managers bestemmer, hvordan komponenter placeres i den beholder (`Container`) de ligger i ved at kalde metoden `setLayout(...)` på beholderen. Igen spøger den enorme fleksibilitet Swing tilbyder. Jf. afsnit 5.2 er det muligt at placere `JComponent's` i `JComponent's`, hvorfor det også kan lade sig gøre at tilknytte layout managers til knapper, tekstfelter, mm. (da `JComponent` arver fra `Container`).

Komponenter tilføjes til `Container` med den samme metode, uanset hvilken layout manager der benyttes. Konsekvensen heraf stiller krav til en ekstrem generisk metode: `add(Component comp,Object constraints)`. Metodens første argument er komponenten som ønskes tilføjet. Andet argument bestemmer layoutinformationerne for komponenten. Her er der intet der forhindre, at et ikke-beregnet objekt angives som parameter. Resultatet udmunder i runtime fejl, idet den valgte layout manager ikke kan behandle parameteret. Swing har dog en overloaded `add(Component comp)` metode, der kun tager komponenten som argument, hvilket er tilstrækkelig for nogle få layout managers.

Udover den overvældende fleksibilitet tilbyder Java også hele syv layout managers. Udfra Swing's ensartede måde at arbejde med komponenter og beholdere på ville det være naturligt, om denne ligeså gjaldt for layout managers. Efter at have arbejdet med de forskellige layout managers, oplever jeg det som om Sun har haft syv forskellige udviklingshold, der på ingen måde har kommunikeret indbyrdes, til at implementere disse. Hver layout manager virker forskellig fra hinanden, lærer man eksempelvis `BorderLayout`, er det ikke ensbetydende med, at man nu forstår `SpringLayout` og igen `GridBagLayout`.

Som eksempel kan vises de tre nedenstående kodeeksempler der håndterer layoutinformationen på hver deres måde. I første eksempel 5.4 linje 8 og 9 bliver de to knapper tilføjet til panelet sammen med layoutinformationen. I andet eksempel 5.5 tilføjes knapperne i linje 9 og 10, hvor layoutinformationen konstrueres seperat i linje 13 og 14. I sidste eksempel 5.6 oprettes layoutinformationen for første knap i linje 12 og 13 og knappen tilføjes i linje

14 sammen med layoutinformationen. Proceduren gentages igen for anden knap.

Kode 5.4: BorderLayout

```
1     JFrame frame = new JFrame();
2     JPanel panel = new JPanel();
3     JButton button1 = new JButton("MyButton");
4     JButton button2 = new JButton("YourButton");
5
6     panel.setLayout(new BorderLayout());
7     frame.add(panel);
8     panel.add(button1,BorderLayout.NORTH);
9     panel.add(button2,BorderLayouy.SOUTH);
```

Kode 5.5: SpringLayout

```
1     JFrame frame = new JFrame();
2     JPanel panel = new JPanel();
3     JButton button1 = new JButton("MyButton");
4     JButton button2 = new JButton("YourButton");
5     SpringLayout layout = new SpringLayout();
6
7     panel.setLayout(layout);
8     frame.add(panel);
9     panel.add(button1);
10    panel.add(button2);
11
12    // specify the constraints
13    layout.putConstraint(SpringLayout.WEST, button1,10,SpringLayout.WEST, panel);
14    layout.putConstraint(SpringLayout.NORTH, button2,100,SpringLayout.NORTH,
          panel);
```

Kode 5.6: GridBagLayout

```
1     JFrame frame = new JFrame();
2     JPanel panel = new JPanel();
3     JButton button1 = new JButton("MyButton");
4     JButton button2 = new JButton("YourButton");
5
6     panel.setLayout(new GridBagLayout());
7     frame.add(panel);
8
9     GridBagConstraints c = new GridBagConstraints();
10
11    // for each component to add, specify the constraints
12    c.gridx = 0;
13    c.gridy = 0;
14    panel.add(button1,c);
15
16    c.gridx = 1;
17    c.gridy = 1;
18    panel.add(button2,c);
```

De tre eksempler viser tre forskellige måder at håndtere layoutinformation, hvoraf ingen er ens. Første eksempel benytter containerens `add(...)` metode og tilføjer komponenten sammen med layoutinformation. I andet eksempel introduceres en nye måde at tilføje layoutinformation på, i form af en metode på layout manageren. Sidste eksempel minder om tilgangen fra det første, men er langt fra hvad man forbinder med objekt orienteret (OO) programmering. Der oprettes et `GridBagConstraints` objekt, hvorpå en række felter tildeles layoutinformationsværdier. Objekter og komponenten tilføjes derefter ved brug af `add(...)` metoden. Problemet opstår idet det sammen `GridBagConstraints` objekt benyttes til den næste komponent, nye værdier tildeles og komponent og `GridBagConstraints` objekt tilføjes til panelet. Ud fra sund OO fornuft burde objektets nye tilstand slå igennem for den første knap, men nej.

Ikke nok med at syntaksen er forskellige for de mange layout managers, komponenternes opførelse er også dybt afhængig af hvilken layout manager der benyttes. Nogle layout managers respekterer størrelserne man giver komponenter, andre er ligeglade, og pludselig har man en knap der fylder hele panelet, hvilket man som programmør måske ikke helt forventer.

Layout managers er en kompleks del og er typisk en showstopper for nybegyndere, hvorfor jeg ønsker at gøre den transparent for brugeren.

Antallet af muligheder at angive layoutinformationer på skal tydeliggøres og minimeres

I Swing tilføjes layoutinformation og komponenter, i de fleste tilfælde, i en og samme erklæring. I `SpringLayout` sker dette dog i to separate erklæringer, som vist i kodeeksempel 5.5 linje 9 og 13. Denne inkonsistens omkring hvordan layoutinformation kontruktioner angives, ønsker jeg at fjerne. Målet er en klar adskillelse af hvordan kodekonstruktioner for henholdsvis placering af komponenter i container (struktur) og layoutinformation skrives.

5.4 Events

Overordnet er events handlinger/reaktioner der kobler brugergrænsefladen sammen med logikken. Dog kan events deles op i to typer, slutbruger-event og system-events. Slutbruger-events afspejler de handlinger der kan udføres i den grafiske brugergrænseflade, og system-events er de interne handlinger. F.eks. et tryk på en knap afspejler et slutbruger-event, og et event der afvikles som respons på resizing af et vindue viser et system-event.

Følgende designmål tager udgangspunkt i slutbruger-events.

Minimer antallet af muligheder og linjer kode det kræves for at implementere eventhåndtering

Eventhåndtering i Java kan implementeres enten ved brug af klasser eller anonyme indreklasser. Klassetilgangen favoriserer genbruglighed, hvilket anonyme indreklasser ikke gør. Skal samme kode afvikles for flere events, kunne klasseilgangen måske være at foretrække, hvorimod hvis et givent styk kode afvikles én gang, ved ét event, vil de anonyme indreklasser være mere passende.

Selvom man som programmør har to valgmuligheder, er mængde af kode der skal skrives, for blot et tryk på en knap, overvældende – især som nybegyn-

der. Jf. kodeeksempel 5.7 kræver dette 7 linjer kode. Som tidligere beskrevet kan dette separeres ud i en klasse og genbruges om nødvendigt.

Kode 5.7: Tryk på en knap

```
1  button.addActionListener(
2      new ActionListener(){
3          public void actionPerformed(ActionEvent ev) {
4              doSomething();
5          }
6      }
7  );
```

Graver vi videre i event værktøjskassen finder vi muse events. Musen er et uundværligt redskab i grafiske brugergrænseflader, en applikation hvori museeffekter ikke optræder er usandsynlig. For at implementere en mouseover effekt ved brug af anonyme indreklasser, skal der kodes 16 linjer jf. kodeeksempel 5.8. Grunden hertil skyldes, at `MouseListener` interfacet har fem metoder der skal implementeres hvad enten de bruges eller ej.

Kode 5.8: Mouseover på en label

```
1  label.addMouseListener(
2      new MouseListener(){
3          public void mouseClicked(MouseEvent ev) {
4          }
5          public void mouseEntered(MouseEvent ev) {
6              doSomething();
7          }
8          public void mouseExited(MouseEvent ev) {
9              doSomething();
10         }
11         public void mousePressed(MouseEvent ev) {
12         }
13         public void mouseReleased(MouseEvent ev) {
14         }
15     }
16 );
```

Heldigvis tilbyder Java igen to strategier der mindsker antallet af kodelinjer. AWT biblioteket indeholder en klasse kaldet `MouseAdapter`.

"An abstract adapter class for receiving mouse events. The methods in this class are empty. This class exists as convenience for creating listener objects." [26].

Ved at arve fra denne abstrakte klasse, behøver man kun overskrive de nødvendige metoder, da der er tale om en abstrakt klasse og ikke et interface. Det tidligere eksempel kan nu reduceres fra 16 til 10 linjer jf. kodeeksempel 5.9. Men idet `MouseAdapter` skal nedarves, kan man komme ud for situationer, hvor man tvinges til at benytte anonyme indreklasser, til at håndtere eventet. Hvis ens klasse, der ellers skulle håndtere eventet, allerede arver fra en anden klasse, er det ikke muligt at benytte adapteren (Java understøtter ikke multipel nedarving). Derfor har Java både et interface og en abstrakt klasse, der udføre samme funktionalitet samt anonyme indreklasser.

Kode 5.9: Mouseover på en label - MouseAdapter

```
1   label.addMouseListener(
2       new MouseAdapter(){
3           public void mouseEntered(MouseEvent ev) {
4               doSomething();
5           }
6           public void mouseExited(MouseEvent ev) {
7               doSomething();
8           }
9       }
10  );
```

Igen ses den enorme fleksibilitet der tilbydes, hvilket jeg ønsker at reducere. Ovenstående præsenterer fire valgmuligheder, hvoraf en programmør skal vælge én til at håndtere events. En programmør kan enten vælge at benytte anonyme indreklasser med interfaces, anonyme indreklasser med adapters, en klasse med interface eller en klasse med adapter (om muligt).

Målet er ikke blot at reducere antallet af linje kode, de fire muligheder skal også skæres ned til en.

Kompleksiteten i eventhåndtering skal tydeliggøres og reduceres

I forlængelse af forgående designmål, skal ikke blot antallet af muligheder nedskæres, men også kompleksiteten forbundet hermed. Udover de mange linjer koder, er brugen af indreklasser, til håndtering af events ikke intuitiv for nybegyndere. Udover at events i sig selv, i form af lytter begrebet, er en kompleks størrelse, bidrager anonyme indreklasser yderligere til summen af kompleksitet, hvorfor tiltag tilsvarende .NETs event håndteringsteknik er interessante.

Intuitionen omkring anvendelsen af events i .NET er enorm. Alle grafiske komponenter har felter, tilsvarende handliger, der kan udføres. Ved at parre metoder og felterne via delegater, opnåes en tydelig og simple tilgang til event håndtering. Ønskes eksempelvis håndtering af klik for en knap, tildeles knappens felt `onClick`, blot metoden som ønskes udført ved klik heraf (kodeeksempel 3.1). Lytter begrebet (`ActionListener`, `MouseListener`, osv.) skjules og der skabes en direkte og tydelig relation, mellem den udførte handling og kodekonstruktionen, på langt færre linjer koder.

Konkret er målet, at (1) skjule lytter begrebet, (2) undgå anonyme indreklasser og (3) tydeliggøre én-til-én relationen mellem handling og kodekonstruktion.

5.5 MVC

Komponenter i Swing, hvis formål er at indeholde data som eksempelvis `JList`, `JComboBox`, `JSpinner`, ect. benytter sig af MVC mønstret. Kom-

ponenten (View) indeholder en passende model (Model), hvor ændringer i modellen opdateres automatisk (Control) i komponenten.

Selv om jeg kun har valgt at implementere tiltaget for `JList`, vil følgende gælde for alle komponenter af samme opbygning.

Antallet af forskellige kodekonstruktioner der kræves for at arbejde med JList skal minimeres

`JList` er en grafisk komponent, hvis formål er at præsentere en liste af elementer. Der er to måder, hvorpå indholdet for en `JList` kan sættes. (1) Man kan sætte hele indholdet ved at give et array eller en vector med som argument til konstruktøren jf. kodeeksempel 5.10 eller (2) lave et objekt, der implementerer interfacet `ListModel` og sætte `JList'ens` model til dette jf. kodeeksempel 5.11 og derfra manipulere listens indhold.

Kode 5.10: Sæt indhold for JList via konstruktør

```
1    Vector v = new Vector();
2    v.add("Foo");
3    v.add("Bar");
4    JList list = new JList(v);
```

Kode 5.11: Sæt indhold for JList via ListModel

```
1    // MyListModel is a class that extends DefaultListModel
2    MyListModel myListModel = new MyListModel();
3    JList list = new JList();
4    list.setModel(myListModel);
5    myListModel.addElement("Foo");
6    myListModel.addElement("Bar");
```

Første eksempel er en simpel tilgang, hvor MVC mønstret ikke slår igennem. Hvis der tilføjes et element til vectoren efter linje 4, opdateres listen ikke. I andet eksempel arbejdes der "korrekt" med Swing. Ved tilføjelse af elementer til `myListModel` notificeres listen og opdateres med ændringerne.

For at få udbytte af MVC effekten, skal man benytte `ListModel` tilgangen. Dette betyder at man som programmør skal udføre følgende skridt:

- Oprette en klasse, der implementerer `ListModel` interfacet.
- Sætte `JList'ens` model til en instans af den oprettede klasse.
- Tilføje elementer til instansen af den oprettede klasse.

Udover kendskabet til interfacet `ListModel`, kræves der ligeledes en smule kendskab til klassen `AbstractListModel` og klassen `DefaultListModel`. Hvis der ønskes events tilknyttet listen, skal endnu et interface og klasse involveres (`ListSelectionListener` og `ListSelectionEvent`).

Målet er (1) at reducere de tre skridt til et, (2) minimere antallet af kodekonstruktioner der benyttes i `JList`, og (3) stadig bibeholde effekten af MVC.

5.6 Opsummering

Jeg har opstillet i alt 10 designmål, jf. nedenstående liste for arkitekturen, som hver især bidrager til simplificering af GUI programmering i Java. Jeg vil i næste kapitel beskrive, hvordan jeg har designet arkitekturen til at efterstræbe de opsatte mål. I kapitlet vil jeg referere til designmålene ud fra nedenstående liste. F.eks. vil D10 være "Gør MVC transparent".

D1 Biblioteket og brugen heraf skal ske inden for de eksisterende rammer af Java 1.5

D2 API skal minimaliseres

D3 Alle fejl skal fanges ved kompileringstid

D4 Forholdet mellem komponenterne skal fremgå klart og logisk i koden

D5 Strukturen skal være rigid. Placering af komponenter ulogiske steder skal ikke tillades

D6 Layout managerene skal elimineres for brugerne

D7 Antallet af muligheder at angive layoutinformationer på skal tydeliggøres og minimeres

D8 Antallet af muligheder og linjer kode det kræves for at implementere eventhåndtering skal minimeres

D9 Kompleksiteten i eventhåndtering skal tydeliggøres og reduceres

D10 Antallet af forskellige kodekonstruktioner der kræves for at arbejde med JList skal minimeres

Kapitel 6

Design

Arkitekturen for det udarbejdede bibliotek HGL og uddrag af proof-of-concept implementeringen beskrives i følgende kapitel. Det diskuteres hvordan de forskellige designmål søges realiseret og overholdt, samt hvilke komplikationer de medfører. Arkitekturen (samt designmålene) deles op i fem dele, hvoraf de generelle dækker et eller flere områder.

Kapitlet afrundes med en opsummering af de restriktioner der har ligget i Java, i forbindelse med implementeringerne af HGL.

6.1 Generelt

Kort fortalt er HGL et selvstændig bibliotek der bruges til at programmere brugergrænseflader med, og kan i den betragtning sammenlignes med Swing og AWT. Anvendelse af HGL kræver intet kendskab til hverken Swing eller AWT men kun til HGL selv.

Teknisk set kan HGL betragtes som en erstatning for Swing og AWT – hvilket så alligevel ikke er helt sandt. HGL bygges ovenpå Swing, men benytter de egentlige komponenter i Swing, dette sker dog transparent. Ved oprettelse af HGL knapper oprettes rent faktisk Swing knapper osv, så i lyset heraf er en bedre opfattelse af HGL måske nærmere, en alternativ tilgang til Swing – et alternativt Swing API om man vil.

Arkitekturen og proof-of-concept implementeringen beskrevet i kapitlet, baseres på HGL's nuværende byggeklodser vist i tabel 6.1.

Tabel 6.1: Byggekoldser i HGL

Frame
Panel
TextField
Button
Label
List

Hele arkitekturen for HGL skal overholde tre generelle designkriterier D1, D2 og D3.

6.1.1 Implementering af D1 og D3

"Biblioteket og brugen heraf skal ske inden for de eksisterende rammer af Java 1.5 og alle fejl skal fanges ved kompileringstid"

Designmål D1 og D3 er strategiske implementeringsbeslutninger truffet fra starten, og har intet med problemerne i Swing at gøre. Begge spiller dog en kritisk rolle i forhold til den overordnet målsætning – men kan være modsigende. Ønsket om at alle fejl skal fanges på kompileringstidspunktet imødekommes ikke, idet HGL realiseres ved brug af refleksion og annoteringer, hvorimod kravet der stilles i D1 overholdes. Hvilke af de to designmål (D1 og D3) der spiller den største rolle i forhold til målet kan diskuteres, jeg har valgt at vægte D1 som det højeste.

6.1.2 Implementering af D2

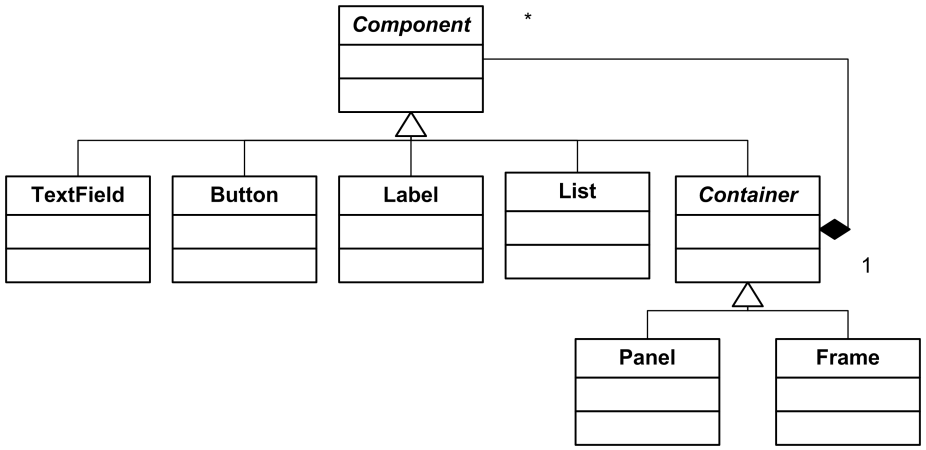
"API skal minimaliseres"

Hvordan HGL indkapsler Swing, og dermed skjuler de underliggende metoder, felter, konstruktører og klasser, således at designmål D2 kan overholdes beskrives i det følgende.

Ønsket om et minimalt API, som ligeledes er et af målene i Buoy [6], realiseres ved indkapsling af Swing gennem komposition. Ideen er, at for hver grafisk komponent jf. tabel 6.1 laves en tilsvarende klasse, som indkapsler den originale klasse fra Swing. Inden løsningen præsenteres, ser vi først på fundamentet for arkitekturen. Figur 6.1 viser opbygningen af de grafiske elementer.

Som i AWT er strukturen opbygget udfra Composite designmønstret [13]. Biblioteket indeholder en `Component` klasse, der afspejler de grafiske elemeter, og en `Container` klasse der udgør kompositionen til at indeholde

disse elementer. Udfra denne struktur eksisterer der to mulige implementeringsstrategier for designmål D2.



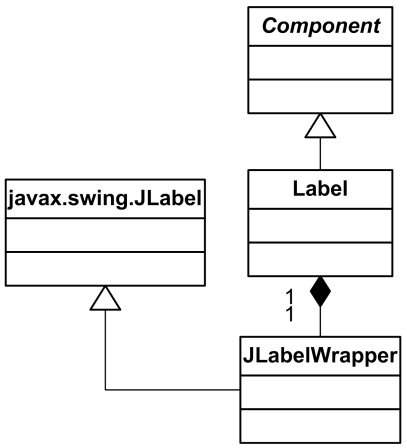
Figur 6.1: Fundamentet bag arkitekturen, uden felter, metoder og type erklæringer

Wrapper klasser

Før mulighederne i Java 1.5 ville en implementering jf. figur 6.2, have været en mulighed (hvilket faktisk var et af mine første udkast til arkitekturen, findes i komplet format i bilag A). `Label` klassen vil gennem en privat komposition til `JLabelWrapper`, som arver fra `JLabel`, skjule alle ellers offentlige felter og metoder. Strategien ville indebærer et ekstra lag i form af wrapper klasser, én for hver komponent, hvilket ville skabe overflod af klasser. Udvides arkitekturen med et element, vil en tilsvarende wrapper klasse skulle implementeres. Hvorfor `Label` klassen ikke blot har kompositionen til `JLabel`, skyldes første udkast til layoutdelen (mere om dette i afsnit 6.3).

Generics

Med de nye muligheder omkring *Generics* [4] i Java 1.5, kan man ved brug af formel type parameter erklæring, fjerne de mange wrapper klasser og lave en generisk. Resultatet er en generisk `Component` klasse, se kodeuddrag 6.1, som har ansvaret for indpakning af de originale Swing komponenter. Den fulde implementeringen af `Component` findes i bilag D.1. Ved at konstruere klassen med parameter typen `<T extends java.awt.Container>` og det beskyttet felt `swingcomponent` med typen `<T>`, sikres det ved opretelse af grafiske komponenter som `Label`, at feltet `swingcomponent` får den korrekte tilsvarende Swing komponents type. Dette opretholdes ved, at



Figur 6.2: Implementeringsstrategir for D2 før Java 1.5

nedarvede klasser fra `Component` tvinges til at angive en type parameter der nedarver fra `java.awt.Container`. De nedarvede klasser sætter derefter, i konstruktøren, feltet `swingcomponent` til en ny instans af Swing komponenten `<T>`.

Kode 6.1: Del af Component og Label klasserne

```
1       abstract class Component<T extends java.awt.Container> implements
            IComponent {
2          protected T swingcomponent;
3            ...
4       }
5
6       protected class Label extends Component<JLabel> implements ILabel {
7          public Label() {
8              this("");
9          }
10
11         public Label(String text) {
12             swingcomponent = new JLabel(text);
13             ...
14         }
15         ...
16      }
```

Eksempelvis er klassen `Label` typeparamatiseret med Swing komponenten `JLabel`, hvor i konstruktøren for `Label`, tildeles en ny instans af `JLabel` til feltet `swingcomponent`, der via. typeparamatiseringen har typen `JLabel`, se kodeuddrag 6.1 (bilag D.3 for den fulde implementering).

Samme konstruktion gør sig gældende for `Container` klassen og dermed dens underklasser (`Frame` og `Panel`).

Hvorfor denne indpakning er nødvendig skyldes som nævnt, at HGL bygges ovenpå Swing, og det er Swing som har ansvaret for, at tegne de grafiske komponenter. Dette betyder at biblioteket blot kalder `setPack()` og `setVisible()` metoderne, for den yderste Swing komponent (hvilket er klassen `Frame's` indpakket `JFrame`), hvorefter Swing overtager kontrollen.

Swing anvender naturligvis tildelte layout managers til at styre placeringen af de forskellige komponenter, dette beskrives i layoutdelen 6.3.

6.2 Struktur

HGL benytter indreklasser til at realisere struktur designmålene, hvoraf navnet The Hierarchical Graphical Library. Inspirationen til dette stammer dels fra min vejleder Kasper Østerbye, Lidskjalv biblioteket til BETA [16], CookSwing [34] og generelt den moderne XML tid vi lever i. Betragter man eksempelvis hvordan GUI laves i Avalon eller CookSwing, benyttes en XML orienteret tilgang. Ideerne bag disse er gode og opfylder faktisk første designmål (D4) for strukturen. Ser vi på kombinationen af designmål D5 og de generelle designmål, vil en adaption af de to frameworks (Avalon og CookSwing) dog ikke være tilstrækkelig. Begge falder igennem på flere områder.

Da jeg bestræber mig på at fange alle fejl på kompileringstidspunktet og samtidig udvikle alt indenfor rammerne af Java 1.5, vil benyttelsen af en seperat XML fil til opbygning af GUI ikke være en mulighed – hvilket er teknikkerne bag blandt andet Avalon og CookSwing. Havde jeg ønsket at benytte en sådan teknik, skulle jeg have haft en form for prævalidering af XML filen op imod selve kode, for at undgå runtime fejl. Det samme gælder for ønsket om den rigide struktur (D5). Set i forhold til eksempelvis CookSwing, der ligesom Swing tillader en fleksible placering af komponenter, ville behovet for en prævalidering igen være til stede. Det ville være nødvendigt at lave et reglsæt for forældre- og børneroller, som blev tjekket inden afvikling af programmet.

Ved brug af indreklasser til håndtering af sturkturen, får jeg de positive træk fra XML (første designmål for struktur D4) og jeg sikrer at placering af komponenter kan styres (andet designmål for struktur D5), samtidig med at de fleste fejl håndteres compile tid.

6.2.1 Et simpelt eksempel

HGL tilbyder en række forskellige byggeklodser jf. tabel 6.1, hvoraf kodeeksempel 6.2 benytter `Frame`, `Panel`, `Label` og `TextField`.

Eksemplet viser hvordan den hierakiske struktur opbygges ved brug af indreklasser. En `Frame` indeholder (som indreklasse) et `Panel`, der indeholder (som indreklasser) henholdsvis en `Label` og et `TextField`.

Kode 6.2: Anvendelse af biblioteket for strukturen

```
1        Frame frame = new Frame() {
2            Panel panel = new Panel() {
3                Label label = new Label("My Label");
4                TextField text = new TextField("Write Here");
5            };
6        };
```

6.2.2 HGL's hierakiske struktur

HGL benytter to typer af indreklasser til at realisere den hierakiske struktur. Hidtil har jeg blot kategoriserede begge som indreklasser, men der er reelt tale om to typer af indreklasser. Én til brug af konstruktion af biblioteket og én til anvendelse heraf. Anvendelse af biblioteket realiseres gennem anonyme indreklasser, og for at muligøre anvendelsesscenarier som i kodeeksempel 6.2, skal biblioteket designes ved brug af normale indreklasser. Distinktionen mellem de to typer af indreklasser spiller en væsentlige rolle som jeg gentagende gange vil komme ind på.

Ved brug af byggeklodserne fra tabel 6.1 er det med HGL muligt at skabe GUI strukturere, indenfor rammerne af tidligere viste klassediagram 6.1. Dog eksisterer der en række restriktioner som følge af designmål D5. HGL definerer følgende reglsæt (restriktioner) for byggeklodserne, som klasseopbygningen fra figur 6.1 skal implementere:

- Der må kun (og skal) eksistere én frame, som samtidig skal være det yderste element.
- Paneler må kun placeres i frames eller andre paneler.
- Knapper, tekstfelter, lister og labels må kun placeres i paneler.
- Knapper, tekstfelter, lister og labels må ikke indeholde andre elementer.

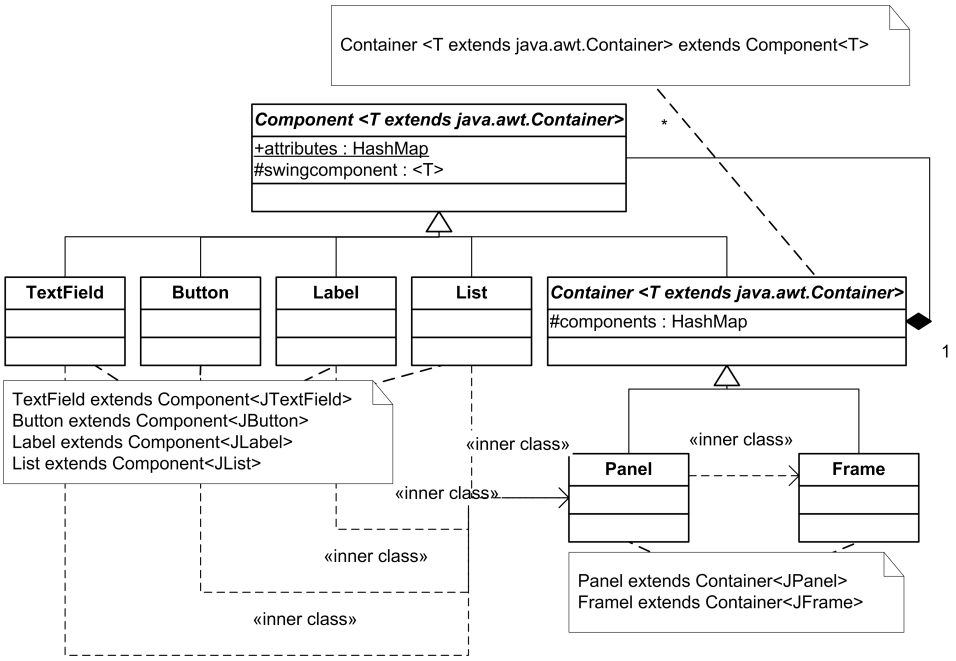
I de følgende to delafsnit beskrives hvordan ovenstående restriktioner (D5) og klasseopbygningen (D4), som tillader brugere at lave hierakiske GUI strukturere konstrueres, samt hvilke komplikationer brugen af indreklasser medfører.

6.2.3 Implementering af D4

"Forholdet mellem komponenterne skal fremgå klart og logisk i koden"

Som nævnt implementeres HGL med indreklasser, for at muliggøre GUI opbyggelse med anonyme indreklasser.

Figur 6.3 udvider tidligere vist arkitektur jf. figur 6.1 med indreklasser og afspejler den komplette klasseopbygning for arkitekturen. Som det fremgår af klassediagrammet, klassificerer HGL alt som værende enten en komponent eller en container, hvoraf containere kan indeholde andre containere og komponenter. Udover dette forhold defineres samtidig et andet "indre" forhold. `Panel` oprettes som indreklasse i `Frame`, og har selv `TextField`, `Button`, `Label` og `List` som indreklasser. Resultatet af dette "indre" forhold er en hierakisk strukturopbyggelse af GUI, som blandt andet muliggøre eksempler jf. kodeeksempel 6.2.



Figur 6.3: Den komplette arkitektur uden interfaces

Oprettelse af de underliggende Swing komponenter

Når en brugere anvender HGL, og dermed opbygger en hierakisk GUI struktur, skal denne omdannes til en tilsvarende Swing struktur, idet HGL bygger på Swing. Det betyder for at kunne anvende Swing mekanismen, skal de tilsvarende Swing komponenter forbindes til hinanden. De underliggende `JButton's`, `JLabel's`, mm. skal placeres i et eller flere `JPanel's`, som evt. selv skal placeres i et `JPanel`, for til sidst at blive placeret i en `JFrame`.

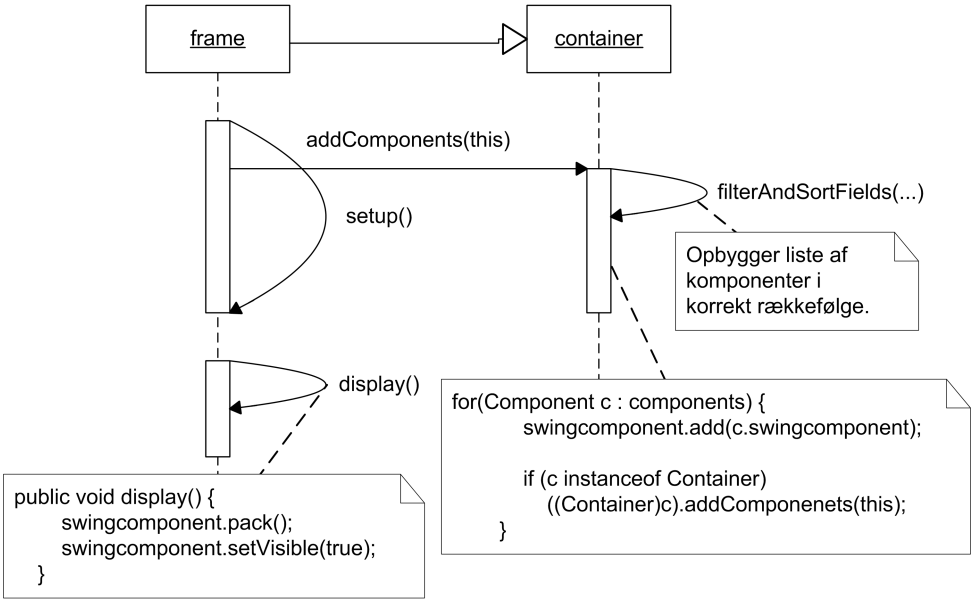
Rutine der udfører denne transformering skal eksplicit aktiveres af brugeren, hvilket istedet burde ske automatisk ved oprettelse af de enklet komponenter. Grundet manglende konstruktioner i Java og HGL's anvendelse af refleksion, er det dog ikke muligt at udfører transformeringen uden brugerens

indblanding (hvilket så alligvel ikke er helt sandt og diskuteres i slutningen af kapitlet).

Figur 6.4 viser sekvensen for transformeringen, og dermed hvordan de underliggende Swing komponenter tilknyttes hinanden. Klasserne der implementerer dette er `Component`, `Container` og `Frame` og kan findes i bilag D.1, D.2 og D.3.

Inden forløbet beskrives diskuteres først problemstillingen forbundet hermed.

Idet HGL benytter refleksion til at finde de forskellige komponenter som udgør GUI'en, gennem refleksionskaldet `getDeclaredFields()`, kan rækkefølgen ikke garanteres overholdt. Placeres eksempelvis en knap og en label i et panel, hvoraf knappen er første element, garanterer refleksionskaldet ikke at knappen hentes frem som det første element. Løsningen på dette problem, som det vil fremkomme i nedenstående, er at holde styr på komponentrækkefølgen i forhold til placeringen i hierakiet.



Figur 6.4: Sekvensdiagram for indkapsling af Swing

Transformeringen starter som sagt ved kald af metoden `setup()`, men før forklaringen af dette, er det nødvendigt at se på løsningen for tidligere nævnte problem.

Alle de grafiske komponenter arver fra klassen `Component`, hvilket betyder at ved objektoprettelse kaldes både `Component` konstruktøren og klassens egen konstruktør. I `Component` klassens konstruktør anvendes to felter til

at holde styr på oprettelsesrækkefølgen af komponenter. En statisk komponenttæller og et komponentnummer der begge jf. kodeeksempel 6.3 tildeles unikke værdier.

Kode 6.3: Konstruktøren for Component

```
1        public Component() {
2            fieldCounter++;
3            fieldNumber = fieldCounter;
4        }
```

Rækkefølgen for oprettelsen af komponenter spiller en kritisk rolle, idet der benyttes refleksion til at forbinde Swing komponenterne til hinanden. Dette vil fremgå af forklaringen af sekvensforløbet for figur 6.4. Efter udførelse af `Component` konstruktøren, udføres klassens egen konstruktør, hvori feltet `swingcomponent` tildeles, se kodeeksempel 6.1, en instans af den tilsvarende Swing komponent.

Tilbage til transformeringen. Efter kaldet til metoden `setup()` kaldes her i, med `Frame` instansen selv som argument, metoden `addComponenets(...)` hvor i følgende to skridt udføres (én eller flere gange):

1. Opret sorteret liste af børnekomponenter for containeren (metodens argumentet).
2. For hver komponent i listen, tilføjes komponentens underliggende Swing komponent til containerens underliggende Swing komponent. Hvis komponenten der tilføjes selv er af typen `Container`, kaldes metoden `addComponenets(...)` med komponenten selv som argument.

Ved hjælp af en filtrerings- og sorteringsmetode i skridt 1, findes alle felter i containeren gennem refleksionskaldet `getDeclaredFields()`. Metoden opbygger en liste af komponenter (`Component's`), baseret udfra felterne i `Container` klassen. Metoden sikrer ligeledes, at selve containeren ikke tilføjes til listen og at kun felter af typen `Component` tilføjes. Grundet det anvendte refleksionskald, hvilket ikke garanterer en sorteret rækkefølge, sorterer metoden til sidst listen udfra komponenternes tildelte komponentnummer fra tidligere (hvorfor komponentnummeret spillede en kritisk rolle).

Efter brugeren har kaldt `setup()` metoden på `Frame` instansen, kaldes `display()` metoden for at få framen vist. `display()` metoden videredelegere kontrollen til Swing, ved at kalde de underliggende Swing metoder `pack()` og `setVisible(true)`.

6.2.4 Implementering af D5

"Strukturen skal være rigid. Placering af komponenter ulogiske steder skal ikke tillades"

HGL's design i form af indreklasser løser designmål D4, men stiller en række udfordringer til designmål D5. Udfra klassediagrammet på figur 6.3 er det muligt at sammensætte elementer ulovligt, jf. de fire punkter i starten af afsnittet. Det betyder at klasseopbygningen giver en medfødte fordel, i form af den hierakiske struktur gennem anonyme indreklasse, og ligeledes afleder den også nogle problemer.

Hvordan reglsættet (de fire restriktionerne fra listen i starten af afsnittet) søges implementeret og dermed løser problemerne for designmål D5, beskrives i det følgende.

Der må og skal kun eksistere én frame, som samtidig skal være det yderste element

Idet biblioteket bygges ovenpå Swing, og dermed anvender de tilsvarende komponenter, tillades det ikke, at placere en frame inden i en frame eller en frame inden i et panel. Resultatet vil være samme runtime fejl som ved Swing.

"*java.lang.IllegalArgumentException: adding a window to a container*"

I bibliotekets nuværende tilstand, fanges fejlen som sagt i den underliggende mekanisme. Ved en smule modifikation heraf, kunne fejlen fanges af selve biblioteket og kaste en mere beskrivende fejlbesked.

Paneler må kun placeres i frames eller andre paneler og knapper, tekstfelter, lister og labels må kun placeres i paneler

For at overholde det ønskede Composite design mønster med restriktionerne, er klasserne `Panel`, `TextField`, `Button`, `Label` og `List` angivet med beskyttet adgangsbegrænsning. Et beskyttet objekt er tilgængeligt for andre objekter i samme pakke (package), eller for objekter udenfor pakken, som har nedarvet fra det beskyttede objekts klasse. Effekten heraf gør, at konstruktioner som i kodeeksempel 6.4 ikke er gyldige uden for bibliotekets pakke.

Kode 6.4: Ikke gyldige konstruktioner grundet beskyttet adgangsbegrænsning for Panel og Button

```
1   // some other package than dk.itu.tq
2
3   Frame frame_one = new Frame();
4   Frame.Panel panel_one = frame.new Panel(); // not valid
5
6   Frame frame_two = new Frame(){
7       Panel panel_two = new Panel(){
8           //...
9       };
10
11      Panel.Button button = panel_two.new Button(); // not valid
12  };
```

I linje 4 forsøges det at skabe et `Panel` objekt, men grundet adgangsbegrænsningen vil compileren brokker sig, idet `Panel` klassen ikke må benyttes her. Havde eksemplet været indenfor bibliotekets pakke, var `Panel` objektet blevet oprettet. Samme situation går igen i linje 11, hvor der prøves at oprette en `Button`.

Grunden til at der tillades oprettelse af et `Panel` i den sidste `Frame`, linje 7, skyldes mekanismen bag anonyme indreklasser. Ved oprettelse af `frame_two` angives der istedet for et semikolon, en ny klasse definition, hvor i `panel_two` oprettes. Denne syntaks betyder, der oprettes et objekt af en anonym klasse som arver fra `Frame`. Idet adgangsbegrænsningen for `Panel` er beskyttet, betyder det at det anonyme objekt nu har adgang til `Frame` klassens beskyttet felter, herunder `Panel`.

Igennem den beskyttet adgangsbegrænsning sikres det delvist, at panel, knapper, lister, mm. ikke oprettes uhensigtsmæssige steder, jf. designmål D5. Hvordan paneler placeret i knapper eller knapper placeret i knapper forhindres, og dermed fuldender designmål D5, beskrives i næste punkt.

Knapper, tekstfelter, lister og labels må ikke indeholde andre elementer

Logikken omkring den beskyttede adgangsbegrænsning forhindrer eksempelvis ikke, at panler placeres i knapper, men gør det derimod muligt. I kodeeksempel 6.5 placeres der et panel inden i en knap, hvilket umiddelbart virker korrekt syntaksmæssigt, men ved brug af nøgleordet `final`, erklæres klasserne `TextField`, `Button`, `Label` og `List` endelige, hvilket betyder at der ikke kan nedarves fra disse. Resultatet for kodeeksemplet er en kompileringsfejl, idet der forsøges at skabe et anonymt objekt, som nedarver fra klassen `Button`.

Kode 6.5: Ikke gyldige konstruktion grundet endelig deklarering af Button

```
1         Frame frame = new Frame() {
2             Panel panel_one = new Panel(){
3                 Button b = new Button(){
4                     Panel panel_two = new Panel();
5                 };
6             };
7         };
```

De tre punkter som netop er gennemgået, viser hvordan implementering af bibliotektet efterlever struktur designmål D5, og hvorfor det ikke er muligt at fange alle fejl under kompilering.

6.2.5 Reference til komponenterne

Strukturen kan opbygges og overholde de to designmål (D4 og D5), men hvordan opnår en brugere referencer til de indlejrede anonyme klasser?

Typeproblem med anonyme indreklasser

Intuitivt, hvilket dog ikke er korrekt, vil man "prikke" sig til referencen som vist i kodeeksempel 6.6 linje 8, hvor der dannes en reference til den indlejrede label gennem hierakiet.

Kode 6.6: Ikke gyldig reference til label

```
1         Frame frame = new Frame() {
2             Panel panel = new Panel(){
3                 Label label = new Label("My label");
4             };
5         };
6
7         void someMethod() {
8             ILabel label = frame.panel.label;
9             label.setText("Got a reference to my label");
10        }
```

Grunden til at sådan en tilgang ikke er lovlig, udmunder i et typeproblem. Umiddelbart ser koden korrekt ud, men faktisk er den anvendte notation for de indre anonyme klasser, en forkortelse for koden i kodeeksempel 6.7 [7]. Som tidligere beskrevet arver det anonyme objekt blot fra typen der skabes, og definerer selv en ny anonym klasse. I eksemplet ses de to anonyme klasser `MyFrame` og `MyPanel` og anvendelsen heraf eksplicit. Framen og panelet der oprettes er af hendholdvis typen `MyFrame` og `MyPanel`, mens referencetypen hertil er `Frame` og `Panel`. Grundet denne "forkerte" referencetype, hvori de oprettede felter ikke eksisterer, er denne løsningsmodel ikke mulig.

Kode 6.7: Typeproblem for anonyme indreklasser

```
1  class MyFrame extends Frame {
2      // implementation...
3      class MyPanel extends Frame.Panel{
4          // implementation...
5      }
6  }
7
8  Frame frame = new MyFrame() {
9      Panel panel = new MyPanel() {
10         Label l = new Label("My Label");
11     };
12 };
```

Grundet typeproblemerne med anonyme indreklasser, understøtter Java ikke den "mest" korrekte løsningen, hvorfor en alternativ tilgang søges.

Løsning

Typeproblemet imødekommes som beskrevet i følgende.

Første skridt er at definere en tilgængelig referencetype for de forskellige grafiske komponenter, idet adgangsbegrænsning for disse er beskyttet. For hver enkelt komponent oprettes et interface som implementeres, jf. tabel 6.2.

Tabel 6.2: Interfaces.

Frame implements **IFrame**
Panel implements **IPanel**
TextFiled implements **ITextField**
Button implements **IButton**
Label implements **ILabel**
List implements **IList**

Andet skridt er at tildele referencetypen en værdi.

Via en simpel `get(...)` metode, som via et sammensat referencevariabelnavn og en type, kan returnere referencer til komponenter, er det muligt at få tildelt værdien til referencetypen. Med udgangspunkt i samme strukturopbygning fra tidligere kodeeksempel, vises der i kodeeksempel 6.8 et gyldigt anvendelsesscenarie til opnåelse af reference for den indlejrede label.

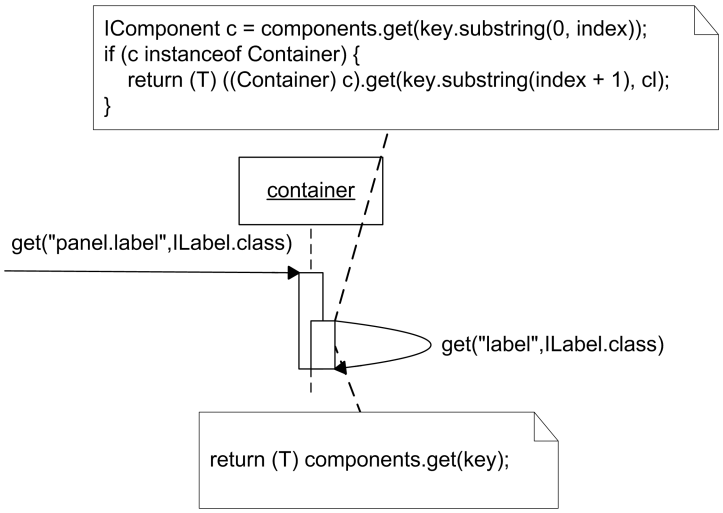
Metoden tager som argument tekststrengen `"panel.label"`, der symboliserer "prikke" tilgangen fra tidligere, og `ILabel` klassens type. Sidste argument definerer metodens returtype, hvorfor ingen *downcasting* af returværdien er nødvendig.

Kode 6.8: Gyldig reference til label

```
1   Frame frame = new Frame() {
2       Panel panel = new Panel(){
3           Label label = new Label("My label");
4       };
5   };
6
7   void someMethod() {
8       ILabel label = frame.get("panel.label",ILabel.class);
9       label.setText("Got a reference to my label");
10  }
```

Løsningen realiseres gennem en udvidet kompositionsrelation, i form af et `HashMap` mellem `Container` og `Component` klasserne (figur 6.3), hvori både komponenterne og deres referencenavne gemmes. Komponentregistreringen heraf udføres i filtrerings- og sorteringsmetoden som tidligere nævnt. Udover komponenterne blev sorteret, gemmes ligeledes komponenternes feltnavne og værdier i containerens `HashMap`.

Hver `Container` instans har således et register over alle sine børne og deres referencenavne, hvorfor det er muligt at finde frem til disse i `get(...)` metoden. Via opslag i registre baseret på de enkelte dele af den sammensatte tekststreng og rekursive kald, ledes efter den søgte komponent.



Figur 6.5: Sekvensdiagram for get(...) metoden

Sekvensdiagrammet i figur 6.5 viser forløbet for kaldet af `get(...)` metoden i kodeeksempel 6.8. Metoden analyserer tekststrengen, finder første komponent (panelet) og kalder derefter sig selv med tekststrengen `"label"`. Idet tekststrengen ikke indeholder yderligere komponenter (i form af flere punktummer), laves der et opslag efter `"label"`, hvor værdien af returneres tilbage til første kald af `get(...)` metoden, som så igen returnerer det endelige svar.

Den fulde implementeringen af `get(...)` metoden findes i bilag D.2.

6.2.6 Opsummering

Resultat af arkitekturen for strukturdelen giver en klar afspejlning af tilhørsforholdet mellem de grafiske komponenter (D4), og tvinger brugeren til at overholde en streng kodestil for strukturering af disse(D5). Bivirkningen ved Java som programmeringssprog resulterer i muligheden for flere runtime fejl end ønsket. Grundet typeproblematikken ved anonyme indreklasser, vil der eksempelvis opstå runtime fejl, ved forkert angivelse af tekststreng i `get(...)` metoden. Havde den konkrete type derimod kunne findes, ville "prikke" syntaksen have været mere ligetil og tættere på den overordnede målsætning.

6.3 Layout

Layout managers er komplekse, modsigende og svære at arbejde med for nybegyndere, hvorfor jeg i min løsning har valgt, at gøre disse transparente. Brugere af HGL skal på intet tidspunkt kende til fænomenet layout managers, men blot via den hierakiske komponentopbygningen og en række metatags, styre placering og layout af elementerne, der tilsammen danner den grafiske brugergrænseflade.

Ved brug af nye muligheder i Java 1.5 realiseres de to opsatte designmål D6 og D7 gennem annoteringer. Valget af og inspirationer hertil udmunder i formålet med annoteringer. Annoteringer er en konstruktion, der anvendes til at beskrive konstruktører, felter, lokale variabler, metoder, pakker, parametre, typer eller annoteringer selv. I forhold til kodekommentare (JavaDoc tags), kan annoteringer findes og anvendes runtime gennem refleksion, hvilket gør dem ideelle som løsningsmodel for D7, og derigennem også for D6. Ligeledes adskilles syntaksen for den hierakiske opbygning og layoutinformationer i to, hvilket gør koden lettere læselig. Ønskes der udviklet et GUI builder værktøj for HGL, gør annoteringerne derudover også det konceptuelle arbejdet nemmere.

I resten af afsnittet opstilles et simpelt anvendelsesscenarie, præsentation af de forskellige annoteringer HGL tilbyder og implementeringerne af D6 og D7.

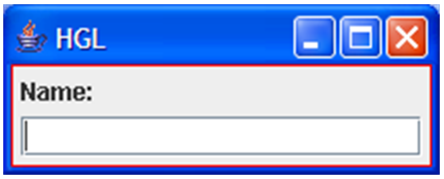
6.3.1 Et simpelt eksempel

HGL tilbyder en række forskellige annoteringer, hvoraf `@Layout` og `@Width` annoteringerne benyttes i kodeeksempel 6.9 (annoteringer præfikses med @ foran navnet). Resten af annoteringerne i HGL beskrives i næste afsnit.

Kode 6.9: Brug af Layout og Width annoteringer

```
1  Frame frame = new Frame() {
2      @Layout(Layout.Orientation.VERTICAL)
3      Panel panel = new Panel() {
4          Label label = new Label("Name:");
5          @Width(200)
6          TextField text = new TextField();
7      };
8  };
```

Eksemplet angiver et metatag for panelet, som bestemmer at alle komponenter der placeres heri, placeres vertikalt fremfor horisontalt. Måden hvorpå dette angives kan ses i linje 2. Den anden annotering i eksemplet benyttes til at bestemme bredden for tekstfeltet, hvilket ses i linje 5. Det grafiske resultat af kodeeksemplet kan ses i figur 6.6.



Figur 6.6: Grafisk resultat af kodeeksempel 6.9

Annoteringer angives, som vist i eksemplet, over komponenten der ønskes berørt. Hvis en komponent har behov for flere layoutinformationer, listes disse blot over hinanden eller ved siden af hinanden.

6.3.2 HGL's annoteringer

Faktorerne layoutinformationerne skal varetage, er komponenternes størrelse, placering, resizing egenskaber og afstand komponenterne imellem. Til at håndtere dette tilbyder HGL, i nuværende stadie, seks forskellige annoteringer, alle vist i tabel 6.3.

Tabel 6.3: Annoteringer i HGL

Annotering	**Værdi**	**Fuktion**
@Height	int	Højde
@Width	int	Bredde
@Layout	Orientation	Placering i containeren
@Hlock	boolean	Horisontal resizing egenskab
@Vlock	boolean	Vertikal resizing egenskab
@Padding	int	Afstand komponenterne imellem

Størrelse

Annoteringerne `@Height` og `@Width` styrer komponenternes størrelse, hvilket respekteres ved resizing (det er ikke mulig at angive en størrelse for en frame, i den nuværende version af layout manageren).

Placering

`@Layout` styrer placeringen af komponenterne i container, hvorpå annoteringen benyttes, hvorfor det kun giver mening at anvende denne på containere. De mulige værdier er horisontal og vertikal, som henholdsvis placerer komponenter fra venstre til højre og fra top til bund. Standard er horisontal placering af komponenterne.

Resizing

`@Hlock` og `@Vlock` benyttes til at kontrollere resizing af applikationen, men ignoreres henholdsvis af `@Height` og `@Width`. Hvis et panel tildeles `@Hlock(false)` og `@Vlock(false)`, resizes panelet både horisontalt og vertikalt i takt med framen, men er der samtidig angivet en bredde, resizes panelet ikke horisontalt. Standard er begge låse slået til, hvilket betyder at komponenter ikke resizes i takt med framen.

Afstand

`@Padding` definerer afstanden mellem komponenterne i en container, hvorfor det også her kun giver mening, at anvende denne på containere. Præcist er det afstanden fra forrige komponent der defineres, standard er afstanden sat til 5px.

6.3.3 Implementering af D6

"Layout managerene skal elimineres for brugerne"

Som vist i strukturdelen, oprettes der internt i HGL et tilsvarende Swing komponenthieraki for den hierakiske HGL komponentopbygningen, hvilket benyttes til at tegne GUI'en. Idet mekanismen der opbygger GUI'en, er den samme som i Swing, skal der enten anvendes absolutpositionering eller en layout manager til at placere komponenterne (strategierne som anvendes i Swing).

Udfra de tidligere erfaringer med Windows Forms 3.1.4 og Visual Studio .NET 3.2.2, som indirekte kræver anvendelsen af et GUI builder værktøj (grundet absolutpositionering), implementerer HGL en specialkonstrueret layout manager, der baseres på de nævnte annoteringer og den hierakiske opbygning.

Implementeringen af HGL's layout manager kan ses i bilag D.2 (klassen `LayoutManager` findes som indreklasse i `Container`).

Måden hvorpå HGL's layout manager anvendes og dermed gøres transparent for brugerne, sker i forbindelse med oprettelsen af de underliggende Swing komponenter. Kodeeksempel 6.10 er uddrag af konstruktørerne for henholdsvis `Frame` og `Panel` klasserne, og viser hvordan layout manageren tildeles den tilsvarende Swing komponent (linje 3 og 8). Idet det kun er containere som kan indeholde komponenter, er det kun `Frame` og `Panel` klasserne som benytter layout manageren.

Kode 6.10: Uddrag af Frame og Panel konstruktørerne

```
1     public Frame(String title) {
2         ...
3         swingcomponent.setLayout(new LayoutManager());
4     }
5
6     public Panel() {
7         ...
8         swingcomponent.setLayout(new LayoutManager());
9     }
```

6.3.4 Implementering af D7

"Antallet af muligheder at angive layoutinformationer på skal tydeliggøres og minimeres"

Hvordan annoteringsinformationerne stilles til rådighed for HGL's layout manager og dermed inkorporeres i arkitekturen, beskrives i de to følgende punkter, hvoraf det første er et tidligt udkast og det sidste er løsningen.

Første løsningsmodel

I første udkast til tilgængeligheden af annoteringsinformationerne, blev felter placeret i wrapper klassen fra tidligere arkitekturudkast jf. bilag A (udkastet håndterer kun `@Vlock` og `@Hlock` annoteringerne). Ideen var at lade wrapper klassen indeholde layoutinformationen, og basere implementeringen af layout manageren på referencetypen `IWrapper`, som er det interface wrapper klasserne implementerer. Implementeringen af layout manageren for sådan en løsningsmodel havde krævet en række typetests, for at bestemme de forskellige komponenters specifikke type.

Årsagen til at felterne og Swing indkapslingen ikke blev placeret direkte i komponenten, skyldes at i layout managerens daværende stadie, kun arbejdede med `java.awt.Container` objekter (hvilket de forskellige wrapper klasser var gennem nedarvning).

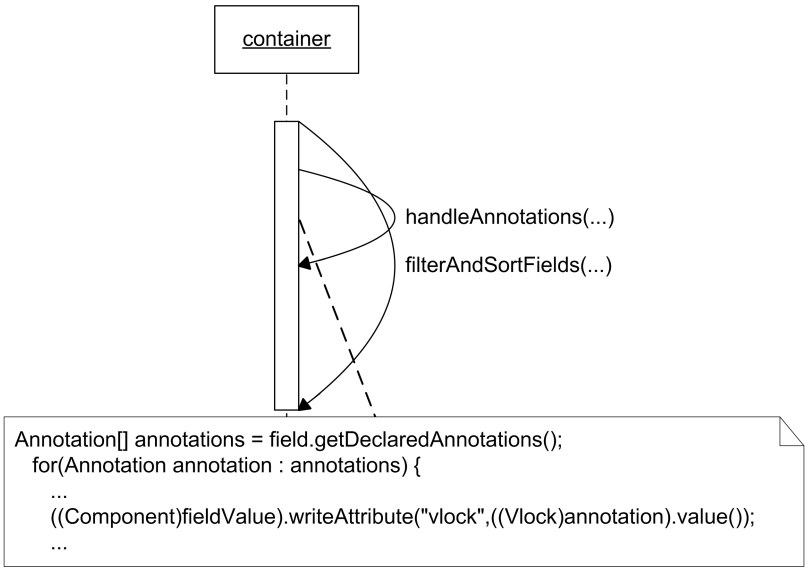
Endelig løsningsmodel

I den endelige løsning arbejder layout manageren stadig med objekter af typen `java.awt.Container`.

Alle layoutinformationer registreres i et statisk paramatiseret `HashMap` jf. kodeeksempel 6.11, hvoraf nøglen er Swing komponenten, og værdien selv er et nyt paramatiseret `HashMap` hvori annoteringens navn og tildelte værdi gemmes.

Kode 6.11: Layoutinformation register

```
1    static protected HashMap<Object, HashMap<String, Object>> attributes =
2    new HashMap<Object, HashMap<String, Object>>();
```



Figur 6.7: Sekvensdiagram for registrering af annoteringer

Registreringen af de forskellige annoteringer gennemføres, via den tidligere omtalte sorterings- og filtreringsmetode, ved brug af to hjælpemetoder. Forløbet fremgår i figur 6.7. Samtidig med at alle felterne i den hierakiske struktur gennemløbes i `filterAndSortFields(...)` metoden, kaldes `handleAnnotations(...)` metoden. Herigennem findes og registreres samtlige layoutinformationsannoteringer i `HashMap'et` via komponentens

`writeAttribute(...)` metode. For klassen `Frame` er dette dog en undtagelse. Grundet at en `Frame` er det yderste element og at det rent faktisk er et indlejret panel i en frame, der indeholder komponenterne der tilføjes til framen , kan den samme mekanisme ikke bruges til at registre annoteringerne herfor som ved `Panel`. Layoutinformationsannoteringer findes der for direkte i framens konstruktør. For implementeringen af layoutinformationerne henvises til bilag D.3.

Den komplette kode kan ses i bilag D.1 og D.2.

Måden hvorpå layoutinformationer hentes frem til anvendelse af layout manageren, håndteres af en statisk opslagsrutine, som via Swing komponentobjektet finder den angivet layoutinformation.

6.3.5 Opsummering

Gennem brugen af Java's annoteringsmekanisme og den hierakiske opbygning af komponenterne, fjernes layout managers (D6) og erstattes med metatags (D7).

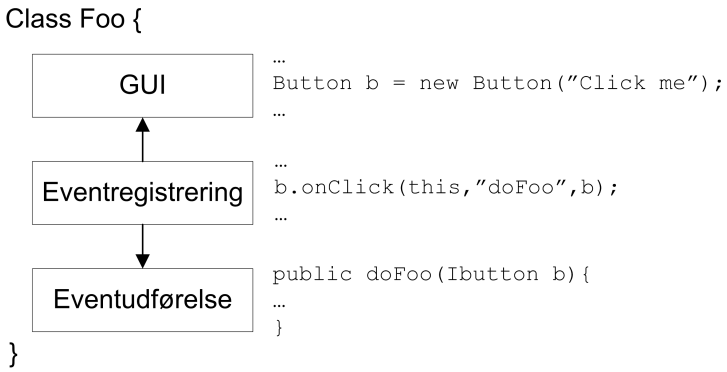
6.4 Events

Eventhåndteringen i HGL er dramatisk anderledes end i Swing. Udover at Swing, jf. designmål D8, tilbyder fire implementeringsstrategier og indirekte kræver kendskab til et eller to designmønstre (Observer og Mediator [13]), er lytter- og eventklassehierakiet også komplekse enheder.

HGL derimod anvender intet lytter-/eventklassehieraki, men benytter metoder direkte til at repræsentere events, samtidig med at det indirekte kendskab til de to designmønstre er skrumpet til et (Mediator).

Eksempelvis skal programmøren kun kende til en metode, for at registrere klik på en knap, modsat Swing hvor det kræver kendskab til to metoder og to klasser.

Udover at HGL simplificerer eventhåndteringsprocessen og derigennem også sætter en række (ønskede) begrænsninger, er ønsket også at separere struktur- og layoutkode fra eventkode, hvilket betyder at eventkode ikke længere implementeres via anonyme indreklasser. Separeringen heraf udmønter sig i en tredelsorganisering til at realisere HGL's eventhåndtering, en organisering som også benyttes i ASP.NET. Figur 6.8 viser hvordan GUI (struktur og layout), eventregistrering og eventudførelse (metoder som eksekveres i forbindelse med eventet) forbindes som helhed til at håndtere events i HGL.



Figur 6.8: Tredelsorganisering for HGL eventhåndtering

I resten af afsnittet vises hvordan man benytter events i HGL, og hvordan disse er implementeret.

6.4.1 Et simpelt eksempel

Hvis man i HGL ønsker at udfører en bestemt rutine ved klik på en knap, benyttes blot metoden `onClick(...)`. HGL forlanger ikke at brugere skal kende til metoderne `addActionListener` og `actionPerformed` samt klasserne `ActionListener` og `ActionEvent`. Kodeeksempel 6.12 viser hvordan realiseringen heraf udformes.

Kode 6.12: Eventet onClick(...) for en knap

```
1   Frame frame = new Frame("HGL"){
2       Panel panel = new Panel() {
3           @Width(200)
4           TextField text = new TextField();
5           Button button = new Button("Click");
6           @Width(200)
7           Label label = new Label("");
8       };
9   };
10
11  void doGUI() {
12      IButton b = frame.get("panel.button",IButton.class);
13      ILabel l = frame.get("panel.label",ILabel.class);
14      ITextField t = frame.get("panel.text",ITextField.class);
15
16      b.onClick(this,"handleButtonClick",l,t);
17  }
18
19  void handleButtonClick(ILabel l, ITextField t) {
20      l.setText(t.getText());
21  }
```

Først dannes en reference til knappen hvorpå eventet registreres, derefter kaldes metoden `onClick(...)` med fire argumenter. Første argument er objektet hvorpå en metode ønskes kaldt, andet argument er navnet på metoden og de sidste argumenter er en liste af argumenter til metoden. I eksempelt, linje 16, kaldes metoden `handleButtonClick(...)` med to argumenter på objektet (`this`) hvori komponenthierakiet er defineret.

Som nævnt er målet i HGL at separere eventkode fra struktur- og layoutkode, og grundet Javas initialiseringsblokke (*instance initialization cluse*), er det muligt at angive eventkode heri, dog med begrænsninger. Konsekvensen ved dette er at tredelsorganiseringen bliver til en todeling, idet det miderste lag i figur 6.8 fjernes og flyttes op til GUI laget. Kodeeksempel 6.13 viser hvordan forrige eksempel kan implementeres via en initialiseringsblok.

Kode 6.13: Eventet onClick(...) for en knap indlejret i strukturheirakiet

```
1   Frame frame = new Frame("HGL"){
2       Panel panel = new Panel() {
3           @Width(200)
4           TextField text = new TextField();
5
6           Button button = new Button("Click");
7           {
8               button.onClick(this,"handleButtonClick",null);
9           }
10          void handleButtonClick() {
11              label.setText(text.getText());
12          }
13
14          @Width(200)
15          Label label = new Label("");
16      };
17  };
```

I linje 7 til 9 oprettes en initialiseringsblok, hvori eventet for knappen registres. Registreringen er altså flyttet ud af `doGUI()` metoden fra tidligere, og op under selve knappen og ligeledes er `handleButtonClick()` metoden flyttet (for at vise at det er muligt). Typeproblemerne i forbindelse med anonyme indreklasser har ingen indflydelse her, hvis metoden `handleButtonClick()` placeres i klassen som bygger GUI hierakiet op, idet man kan skrive klassens navn punktum `this` (`klassenavn.this`), men placeres den i et andet niveau i hierakiet kan den ikke findes grundet typeproblemerne.

Begrænsningerne heri skyldes igen typeproblemerne. Hvis lablen lå placeret i et andet panel, som var i samme niveau med det tidligere panel, ville det ikke have været muligt at få fat i referencen til labelen, som i linje 11. Løsningen ville være at benytte `get(...)` metoden fra forrige kodeeksempel.

Muligheden for at angive eventhåndteringskode i både den hierakiske struktur og adskilt herfra, strider ikke imod designmål D8 som siger, at eventhåndteringsmulighederne skal skæres ned til én. Reduceringen i D8 går på interface versus adapters og anonyme indreklasser versus klasser, hvor HGL blot giver programmører mulighed for at skrive eventhåndteringskode to forskellige steder.

6.4.2 HGL's events

I HGL's nuværende stadie tilbydes fire events fordelt på komponenterne vist i tabel 6.4.

Tabel 6.4: Events i HGL

Event	Komponenter
onMouseOver	TextField, Button, Label
onTextChanged	TextField, Label
onClick	Button
onSelectedItemChanged	List

onMouseOver

Aktiveres når musecursoren trækkes ind på et tekstfelt, knap eller label.

onTextChanged

Aktiveres når tekstsindholdet for et tekstfelt eller label ændres.

onClick

Aktiveres når der klikkes på knappen.

onSelectedItemChanged

Aktiveres når det valgte element i listen ændres til et andet.

6.4.3 Implementering af D8 og D9

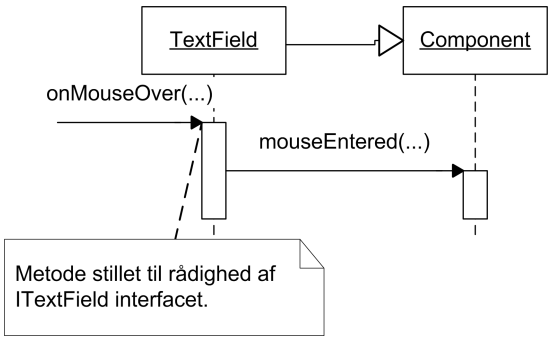
"Antallet af muligheder og linjer kode det kræves for at implementere eventhåndtering skal minimeres og kompleksiteten i eventhåndtering skal tydeliggøres og reduceres"

Som vist stiller HGL en række events til rådighed gennem metoder på komponenterne, men under overfladen er det igen Swing (AWT) som rent faktisk håndterer og implementerer eventene. I det følgende vises hvordan HGL implementerer Swing's eventmekanisme, og stiller den til rådighed for brugere i en simplere facon.

Indkapslingen af eventmekanisme for Swing

HGL's eventmetoder stilles som vist, til rådighed gennem komponenterne og deres interfaces. Flere af komponenterne tilbyder samme eventmetode,

hvorfor en Swing implementeringen heraf søges genbrugt. Indkapslingen benytter derfor konceptet omkring designmønstret Template Method [13] til at indkapsle Swing's eventhåndtering, hvor det er muligt. `Component` klassen som alle de grafiske komponenter arver fra, implementerer en række beskyttet metoder, hvori eventhåndteringen indkapsles. De grafiske komponenter implementerer en række offentlige metoder, jf. tabel 6.4, som videredelegere kaldet til de implementerede metoder i `Component` klasse (de offentlige metoder stilles tilrådig via komponenternes interface). Sekvensdiagrammet på figur 6.9 viser det simple forløb.



Figur 6.9: Sekvensdiagram for kald af onMouseOver(...)

Swing indkapslingen af henholdsvis eventet `onClick` for `Button` og eventet `onSelectedItemChanged` for `List`, implementeres dog ikke i `Component` men i `Button` og `List` klasserne selv. Grunden hertil skyldes at `Component` klassen er paramatiseret med `<T extends java.awt.Container>`, hvilket som bekendt benyttes til at bestemme typen for den tilsvarende Swing komponent. Konsekvensen heraf er, at lytterne og adapterne som bruges til at implementere events med, skal være tilgængelige i `java.awt.Container`, hvilket de ikke er for henholdsvis eventet `onClick` for `Button` og eventet `onSelectedItemChanged` for `List`, som henholdsvis implementeres af `ActionListener` og `ListSelectionListener`.

Implementeringen af eventmekanisme for Swing

Indkapslingen af Swing implementeres ved, at tilføje en passende lytter eller adapter til komponentens tilsvarende Swing komponent, og derefter implementere de af lytterens/adapterens nødvendige metoder til at håndtere HGL eventet. Tabel 6.5 viser hvilke Swing lyttere/adaptere og metoder der benyttes for HGL komponenterne og deres events. Eksempelvis benyttes en `java.awt.event.MouseAdapter()` lytter og en `mouseEntered(...)` metode til, at implementere HGL eventet `onMouseOver` for alle komponenterne.

Tabel 6.5: Brugte Swing lyttere og adaptere i HGL

HGL Komponent	**HGL Event**	**Swing Lytter/Adapter**	**Swing Metode**
TextField	onTextChanged	KeyAdapter	keyReleased
TextField	onMouseOver	MouseAdapter	mouseEntered
Button	onClick	ActionListener	actionPerformed
Button	onMouseOver	MouseAdapter	mouseEntered
Label	onMouseOver	MouseAdapter	mouseEntered
Label	onTextChanged	PropertyChange-Listener	propertyChange
List	onSelected-ItemChanged	ListSelectionListener	valueChanged

Internt i lyttermetoden som implementerer eventet, benyttes refleksion til at finde og eksekvere metoden, der gives med som argument i form af tre parametre. Kodeeksempel 6.14 viser implementeringen for HGL eventet `onMouseOver`.

Kode 6.14: Implementering af onMouseOver

```
1   // TextField class
2   public void onMouseOver(final Object target, final String targetmethod, final
        Object... targetmethodargs) {
3       mouseEntered(target, targetmethod,targetmethodargs);
4   }
5
6   // Component class
7   void mouseEntered(final Object target, final String targetmethod, final
        Object... targetmethodargs) {
8       swingcomponent.addMouseListener(new java.awt.event.MouseAdapter() {
9           public void mouseEntered(java.awt.event.MouseEvent ev) {
10              try {
11                  Method m = findMethod(target,targetmethod,targetmethodargs);
12                  m.setAccessible(true);
13                  Object result = m.invoke(target,targetmethodargs);
14              } catch (NoSuchMethodException e) { System.out.println(e); }
15                catch (IllegalAccessException e) { System.out.println(e); }
16                catch (InvocationTargetException e) { System.out.println(e); }
17          }
18      });
19  }
```

Et interessant men problematisk element i eksemplet er `findMethod(...)`, som jeg selv har måtte skrive, i linje 11. I refleksionsbiblioteket i Java, kan en klasses metoder findes på baggrund af metodenavn og parametertypernes klasse. Dog tager rutinen ikke højde for arv i forbindelse med parametertyperne. Kaldes eksempelvis refleksionsrutinen med en nedarvet type, og klassen der søges i indeholder metoden men med super typen for den nedarvet type, findes metode ikke. Grunden hertil er at Java søger efter en metode med "*the same formal parameter types*" [26], hvilket ikke tager højde for arv, men med `findMethod(...)` metoden er dette ikke et problem. Metoden finder alle metoder med samme navn og antal parametre, hvorefter metoden med flest specifikke antal parametre benyttes (metoden findes

i bilag D.1). Såfremt metoden finder to lige specifikke metoder, og de er de mest specifikke, kastes en fejl. Returværdien i forbindelse med metoden der søges ignoreres, idet den i forbindelse med eventhåndtering anses for at være irrelevant.

6.4.4 Næste generations eventhåndtering i HGL

Ovenstående implementering samt brugen heraf, efterlader en række åbenlyse *nice-to-have* egenskaber. Underafsnittet her belyser en række punkter, endnu ikke implementeret i HGL, hvilke på designniveau bidrager til HGL's eventhåndtering. Grundet specialets og dermed HGL's målgruppe er det dog usikkert, hvorvidt nedenstående udvidelse vil være passende.

Fejlhåndtering

HGL kaster ikke en undtagelse, såfremt metoder der søges efter ikke findes, og ej heller hvis metodeeksekveringen fejler. Brugerens eneste informationskildes hertil er et output til konsollen. HGL kunne udvides til at kaste undtagelser, og derigennem tvinge brugere til at håndtere disse (kompileringstvunget undtagelser), eller som ved opbygningen af hierakiet og referenceopnåelse af komponenter, kaste en runtime undtagelse.

HGL Method klasse

I HGL's nuværende eventmetoder angives tre parametre, hvoraf det sidste evalueres til en værdi, og ikke selv behandles som en metode der skal gennem refleksionsprocessen. Angives eksempelvis en `getText()` metode hvor intensionen er at hver gang eventet afvikles, afvikles ligeledes `getText()` metoden, overholdes intensionen ikke idet resultatet af `getText()` metoden afvikles inden selve eventmetodekaldet, hvorfor det sidste argument altid evaluere til samme værdi.

Ønsket er et dynamisk afviklet sidste argument for eventhåndteringen.

HGL udvides med en offentlig klasse `Method`, jf. kodeeksempel 6.15, som indkapsler de nuværende tre argumenter.

Kode 6.15: HGL Method klasse

```
1     public class Method {
2         Object target;
3         String targetmethod;
4         Object[] targetmethodargs;
5
6         Method(Object target, String targetmethod){
7         }
8
9         Method(Object target, String targetmethod, Object... targetmethodargs){
10        }
11
12        public addArguments(Object arg) {
13        }
14    }
```

Ideen er at sidste argument i sig selv, kan være et `Method` objekt. Dette betyder at afviklingen heraf sker dynamisk, og at der dermed ikke tilknyttes en statisk værdi til metoden, som eventet skal eksekvere (første og andet argument i eventet). Kodeeksempel 6.16 viser hvordan ideen anvendes.

Kode 6.16: Anvendelse af HGL Method klasse

```
1     IButton b = get.("panel.button", IButton.class);
2     ITextField t = get("panel.text", ITextField.class);
3
4     b.onClick(new Method(this, "doSomething", new Method(t,"getText")));
```

Eksemplet består af en knap og et tekstfelt, hver gang der trykkes på knappen, skal en metode `doSomething(...)` afvikles med værdien af tekstfeltet. Eventet `onClick` registreres på knappen med `doSomething(...)`, og med et nyt `Method` objekt som argument hertil.

En sådan udvidelse af designet for HGL kræver ligeledes en udvidelse i indkapslingen af Swing. Rutinen som håndterer lytter begrebet, skal ændres til at kunne tage højde for `Method` objekter (som igen kan indeholde `Method` objekter).

Avanceret events

Grundet den store grad af indkapsling, har brugere af HGL ikke adgang til selve Java eventet, hvilket i særlige situationer ikke er hensigtsmæssigt. Normalt bruges Java eventet til at finde ekstra oplysninger, såsom hvilken museknap eller tegn på tastaturet der blev trykket på. Første situation kan løses med et dedikeret event for hver museknap, hvorimod sidste situation kræver en lidt anden tilgang. Ideen er at oprette et event som returnerer værdier, hvilket så efterfølgende kan anvendes i andre HGL events. Kodeeksempel 6.17 viser tankesættet her bag.

Kode 6.17: Event med retur værdi

```
1     public String getKeyPressed()  {
2         EventInfo eventinfo = new EventInfo();
3         keyReleased(eventinfo);
4         return eventinfo.getKeypressed();
5     }
6
7     public String onKeyPressed(Method method)  {
8         keyReleased(Method method);
9     }
```

HGL stiller et `getKeyPressed` event til rådighed for brugeren, som returnerer hvilken knap på tastaturet der er trykket på. Brugeren kan benytte resultat heraf i eventet `onKeyPressed`, som vist i kodeeksempel 6.18 (eksemplet benytter `Method` klassen fra forrige punkt).

Kode 6.18: Anvendelse af event med retur værdi

```
1     {
2         TextField t = frame.get("panel.text", ITextField);
3
4         Method m1 = new Method(t, "getKeyPressed");
5         Method m2 = new Method(this, "someMethod", m1);
6
7         t.onKeyPressed(m2);
8     }
9
10    void someMethod(String s) {
11        System.out.println(s);
12    }
```

Eksemplet opretter to `Method` objekter `m1` og `m2`, linje 4 og 5, hvor `m1` indkapsler `getKeyPressed` eventet, der benyttes som argument i `m2`. På tekstfeltet `t`, linje 7, registreres eventet `onKeyPressed` med `m2`. Ved indtastning af tekst i tekstfeltet, kaldes metoden `someMethod(...)` med værdien af tasterne der trykkes på.

6.4.5 Opsummering

HGL har skåret antallet af kodelinjer ned for selve registreringen af eventet og indskrænket mulighederne herfor (D8), samt indført en mere intuitvit tilgang (D9). Ved at skjule lytter begrebet og derigennem Observer designmønstret, er kompleksiteten faldet (D9). Fra at skulle forstå et abstrakt og generelt mønster, er eventhåndteringen nu blevet mere konkret og jordnær.

Grundet den store grad af refleksion, overholdes designmål D3 ikke. Derudover er det eneste konkrete problem typeproblemet fra tidligere. En positiv sideeffekt implementeringen har givet, er en større indkapsling af de metoder eventene afvikler, set i forhold til Swing. Hvis man eksempelvis vælger at lade sin klasse implementerer eller nedarver fra henholdsvis lyttere og adaptere, vil eventmetoderne heri være offentlige, hvilket ikke er god indkapslingsskik.

6.5 MVC

HGL implementerer en simpel, let anvendelig liste, hvori Swing kompleksiteten skjules og gøres transparent for brugerne, samtidig med at MVC effeketen stadig slår igennnem. Inspirationskilden for sådan en løsning stammer dels fra Windows Forms (dog overholder denne ikke helt MVC konceptet) og dels fra Buoy. Begge har implementeret en lignende liste, hvor anvendelse heraf sammenlignet med `JList` er mange gange simplere. Grunden til jeg ønsker at simplificere `JList` skyldes jf. desginmål D10, at en programmør skal kende til alt for mange konstruktioner for at udnytte `JList` korrekt, og med det menes at bibeholde MVC effekten.

I det følgende beskrives hvordan listen benyttes, de forskellige funktioner listen stiller til rådighed, samt en sammenligning mellem HGL's liste (`List`) og Swing's liste (`Jlist`), og hvordan implementeringen er realiseret.

6.5.1 Et simpelt eksempel

I kodeeksempel 6.19 benyttes tre metoder til at manipulere en HGL liste. I linje 2 oprettes en reference til listen, og i linje 4 og 5 tilføjes to tekststrenge. I linje 7 registreres et `onSelectedItemChanged` event, som kalder metoden `handleList(...)` med listen som argument, hvor det valgte element udskrives til konsollen.

Kode 6.19: Anvendelse af HGL List

```
1     {
2         IList list = frame.get("panel.list",IList.class);
3
4         list.addItem("item 1");
5         list.addItem("item 2");
6
7         list.onSelectedItemChanged(this,"handleList",list);
8     }
9
10    public void handleList(IList list) {
11        System.out.println("Selected item was: "+ list.getSelectedItem());
12    }
```

6.5.2 HGL's List

En HGL `List` indeholder otte offentlige metoder til at manipulere indholdet med, alle vist i tabel 6.6.

Tabel 6.6: Metoder for HGL List

Metode	Formål
void addItem(Object item)	Tilføjer et element til listen
Object getSelectedItem()	Returnerer det valgte element
ArrayList getItems()	Returnerer alle elementerne
void setItems(ArrayList items)	Fylder listen med elementer
void removeItem(Object item)	Fjerner et element
void update()	Opdaterer GUI'en for listen
void onSelectedItemChanged(...)	onSelectedItemChanged event
void clear()	Tømmer listen
int size()	Antallet af elementer i listen
void setSelectedItemIndex(int i)	Vælger element i

Som det fremgår af tabellen, kan brugere benytte et `ArrayList` til at styre indholdet af listen med (`setItems(...)` metoden tager som argument et `ArrayList` og fylder listen med indholdet heraf) og samtidig med at MVC effekten bibeholdes, ingen `ListModel` eller `DefaultListModel` er nødvendig. Ændringer i listens antal af elementer opdateres automatisk mens ændringer i enkelte elementer kræver et `update()` kald, hvorfor listen implementerer en `update()` metode.

Sammenligning af HGL's List og Swing's JList

Målet med HGL's `List` er at skabe en simplere version af Swing's `JList`, hvilket jeg i det følgende vil vises er opnået.

Sammenligningen tager udgangspunkt i en liste, hvori der tilføjes et element og der registreres et event, som udskriver værdien af elementer når de vælges fra listen. Fokus for sammenligning, jf. designmål D10, er antallet af forskellige kodekonstruktioner der benyttes, for at implementere eksemplet med henholdsvis `List` og `JList`. Tabel 6.7 opsummerer resultatet heraf og implementeringen af sammenligningen findes i bilag B.

Tabel 6.7: Sammenligning af konstruktioner i JList og List

Konstruktion	Swing's JList	HGL's List
Metode	12	6
Klasse	6	3
Interface	1	1
Annotering	0	2
I alt	**19**	**12**

Som det klart fremgår af tabellen, er målet med HGL's `List` opnået. I forhold til `JList` indeholder `List` syv konstruktioner færre – en reducering på ca. 37%.

6.5.3 Implementering af D10

"Antallet af forskellige kodekonstruktioner der kræves for at arbejde med JList skal minimeres "

Implementeringen af designmål D10, og dermed listen, arbejder internt med en `DefaultListModel`, hvorfor MVC effekten bibeholdes. Idet en ny liste oprettes, oprettes ligeledes en ny `DefaultListModel` som indholder elementer for listen. Metoderne der manipulerer listens indhold, manipulerer derfor med den interne `DefaultListModel` som er skjult for brugerne. Udover arbejdet med elementerne i listen, implementeres ligeledes en metode til håndtering af eventet `onSelectedItemChanged`. For at kunne håndtere dette event, kræves der en implementering af interfacet `ListSelectionListener`, hvilket `List` klassen selv implementerer som en privat indreklasse.

6.5.4 Opsummering

Som det fremgår af implementeringen af listen, er dette arbejde forholdsvis ligetil, såfremt det laves én gang. Skal man benytte flere lister i det program man laver, og vælger man at implementere et nyt `JList` konstruktionssæt (`JList`, `ListModel` og eventhåndtering) pr. liste, er dette et større arbejde. Sammenligningen mellem `List` og `JList` viser tydeligt en forskel mellem de to, og viser at designmålet herfor er opnået. Et enkelt kritik punkt til HGL's `List` er dog, at den ikke er generisk, da muligheden herfor eksisterer i den nye version af Java.

6.6 Opsummering af restriktioner med Java som implementeringssprog

Designet og implementeringen af HGL overholder alle de opstillet designmål på nær et enkelt. Grundet manglende konstruktioner i Java er det ikke muligt for HGL, at overholde designmålet D3, som siger at alle fejl skal fanges ved kompileringstid. Udover brud på D3 eksisterende en mindre problemstillingen, men som stadig er en problemstilling. Projektets formål og målgruppen herfor, tager udgangspunkt i simplificeret GUI konstruktion, hvorfor `Frame` klassens metode `setup()` er et irritationsmoment. Brugere

af HGL er tvunget til eksplicit at kalde metoden, hvor den burde blive kaldt internt i biblioteket af biblioteket selv. En sidste problemstilling som ikke skyldes Java's manglende konstruktioner, men designet for arkitekturen, er problemstillingen omkring en frame placret i en frame. Brud herpå resulterer ikke i kompileringsfejl med derimod runtime fejl.

Idet følgende vi jeg opsummere problemerne forbundet med designmål D3 og `setup()` metoden, og derigennem beskrive hvilke konstruktioner Java mangler, og som kunne have løst problemerne. Tilsidst vil jeg diskutere konsekvensen af det eksisterende design, i forhold til problematikken omkring frames placeret i frames.

Typeproblem med anonyme indreklasser

Grundet Java's design af anonyme indreklasser er det som bekendt ikke muligt at finde den præcise type. Konsekvensen heraf resulterer i en `get(...)` metode, der returnerer referencer til de anonyme indreklasser, som bekendt udgør de grafiske komponenter i GUI'en. Havde Java derimod understøttet type inferens [30], som eksempelvis Standard ML [14], ville den anonyme indreklasses type ikke være ukendt. Pseudokoden vist i kodeeksempel 6.20 viser ideen bag dette.

Kode 6.20: Pseudokode for type inferens

```
1     val frame = new Frame() {
2         val panel = new Panel() {
3             val label = new label("my label");
4         }
5     }
6
7     void someMethod() {
8         frame.panel.label.setText("your label");
9     }
```

Gennem et nøgleord `val`, som bestemmer at type inferens skal benyttes for den givende variabel, er det muligt at finde den korrekte type for henholdsvis `frame` og `panel`. Hvilket resulterer i at `get(...)` metoden kan fjernes og at designmålet D3 på dette punkt kan overholdes.

Tvunget eksplicit kald til setup() metoden

Formålet med `setup()` metoden er at stille komponenternes værdi og referencenavn til rådighed for `get(...)` metoden i `Frame` klassen, samt at registrere komponenternes layoutinformationer (annoteringerne) til brug for layout manageren.

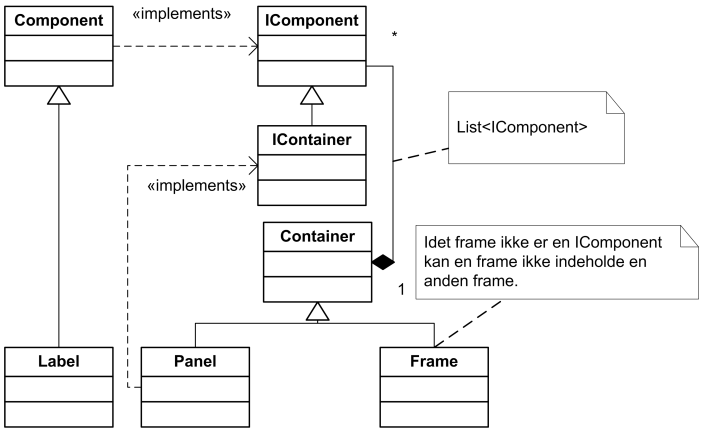
Ønsket er at skjule kaldet af `setup()` metoden, så brugere ikke er tvunget til eksplicit at kalde denne. Et naturligt sted at indkapsle metoden ville være i `Frame` klassens konstruktør, idet et GUI program altid indeholder en

`Frame` instans. Men grundet initialiseringsrækkefølgen af objekter i Java, vil Java starte med at oprette det yderste element i den hierakiske struktur og derefter arbejde sig indad, hvorfor det ikke er mulig at placere metoden i konstruktøren for `Frame`. Havde det derimod været muligt, at tvinge Java til at oprette sine børneklasser først og herefter kalde metoden, ville sådan en løsning være mulig. I programmeringssproget BETA [21] eksisterer et ligende koncept, rettet mod arv mellem metoder. Ved brug af en konstruktion kaldet `inner` [15], er det muligt for en supermetode, at aktivere eksekveringen for den nedarvet metode. Men da der i den hierakiske opbygning ikke er tale om et arve forhold komponenterne imellem, kan `inner` konstruktionen ikke overføres direkte, men konceptet heromkring kunne måske.

En reel løsningsmodel (som er mulig i Java) ville være at placere `setup()` metoden i selve `get(...)` metoden og dermed skabe en *lazy* initialisering af informationerne, så første gang `get(...)` metoden blev kaldt, skulle komponenters værdi og referencenavn registreres og anden gang skulle intet ske (idet komponenterne allerede var registreret). Løsningsmodellen her ville dog samtidig kræve, at registreringen af layoutinformationerne blev flytte til `display()` metoden, for at sikre at informationerne er tilgængelige for layout managerne, såfremt `get(...)` metoden aldrig blev kaldt. En anden løsningen ville være at separere initialiseringen af `HasMap'et`, som netop indeholder komponenters værdi og referencenavn, ud i en selvstændig metode, en såkaldt *Extract Method refactoring* [12], der så blev kaldt fra `get(...)` metoden af.

Konsekvensen ved brugen af det eksisterende Composite designmøsnter

Grundet den "direkte" brug af Composite designmønstret er det muligt, at placere en frame i en frame. Resultatet heraf er Swings's runtime fejl, såfremt dette prøves. En alternativ arkitektur for Composite mønstret, hvor `Frame` klassen adskilles fra `Component`, så `Frame` ikke er en underklasse eller implementering heraf, ville kunne fange frames placeret i frames på kompileringstidspunktet. Figur 6.10 viser ideen bag dette.



Figur 6.10: Alternativ Composite mønster

`Container` klassen der udgør kompositionen til `IComponent`, benytter en paramatiseret liste `List<IComponent>` der definerer, at kun objekter af typen `IComponent` kan tilføjes hertil. Og idet `Frame` ikke implementerer interfacet `IComponent` og `Panel`, samt de resterende komponenter (hvor kun `Label` er vist på figuren) gør, sikres det at en `Frame` ikke kan tilføjes andre steder.

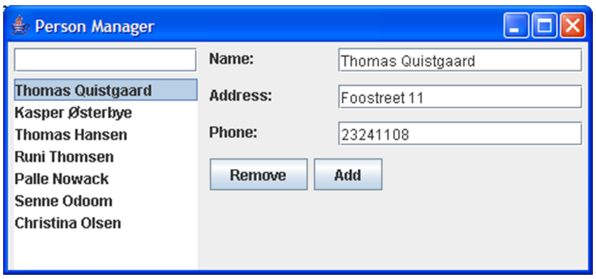
Kapitel 7

Eksempel

Indtil nu har jeg beskrevet designet for HGL, og opstillet brudstykker af HGL kode. I følgende kapitel vil jeg udfra en komplet implementeret applikation, kodet i både Swing og HGL, vise hvordan HGL især har simplificeret opbygningen af selve den grafiske brugergrænseflade sammenlignet med Swing. Dette betyder ikke at HGL ikke også har simplificeret eventhåndtering og MCV effekten for `JList`, men blot at GUI opbygningen anses for, at være mere banebrydende end eventhåndtering og MCV. Udover at kapitlet sammenligner strukturen og layoutet for GUI programmering, præsenteres der også en generel sammenligning heraf i form af antal forskellige kodekonstruktioner.

Implementeringen af eksemplet diskuteret i kapitlet, kan ses i komplet format i bilag C.

7.1 Personmanager applikation



Figur 7.1: Person Manager.

Applikation der ligger til grund for sammenligningen er en simple personoversigtsapplikation, hvorfra det er muligt at vælge personer fra en liste eller

sortere på disse via et sorteringsfelt, og dermed få informationerne tilknyttet personen vist i en række tekstfelter. Et screendump fra applikationen fremgår af figur 7.1.

7.1.1 HGL implementering af GUI-delen

En HGL implementering for GUI-delen er vist i kode eksempel 7.1.

Eksemplet benytter en frame, hvori der placeres to paneler til at håndtere henholdsvis venstre- og højreside. I panelet til venstre sættes et tekstfelt og en liste, og i panelet til højre indsættes et panel for hver linje/par af personinformation der vises. Der anvendes en `@Width` annotering på labelen til at sikre, at tekstfelterne står præcist under hinanden.

Kode 7.1: HGL GUI kode

```
1  Frame frame = new Frame("Person Manager") {
2
3      @Layout(Layout.Orientation.VERTICAL)
4      Panel leftpanel = new Panel() {
5          @Width(150)
6          TextField searchtextfield = new TextField();
7          @Width(150)
8          @Height(300)
9          List list = new List();
10     };
11
12     @Layout(Layout.Orientation.VERTICAL)
13     @Padding(0)
14     Panel rightpanel = new Panel() {
15
16         Panel namepanel = new Panel() {
17             @Width(100)
18             Label namelabel = new Label("Name:");
19             @Width(200)
20             TextField nametextfield = new TextField();
21         };
22
23         Panel addresspanel = new Panel() {
24             @Width(100)
25             Label addresslabel = new Label("Address:");
26             @Width(200)
27             TextField addresstextfield = new TextField();
28         };
29
30         Panel phonepanel = new Panel() {
31             @Width(100)
32             Label phonelabel = new Label("Phone:");
33             @Width(200)
34             TextField phonetextfield = new TextField();
35         };
36
37         Panel addpanel = new Panel() {
38             IButton removebutton = new Button("Remove");
39             IButton addbutton = new Button("Add");
40         };
41     };
42 };
```

Som det klart fremgår af koden, bygges GUI'en deklarativt frem for imperativt. Koden giver et klart overblik over de to paneler (højre og venstre panel), samt over hvilke komponenter disse indeholder. HGL giver dermed et overskueligt overblik over tilhørsforholdet imellem komponenter, hvor vi i Swing eksemplet 7.2 vil se det modsatte.

7.1.2 Swing implementering af GUI-delen

En Swing implementering for GUI-delen er vist i kode eksempel 7.2.

Eksemplet benytter ligeledes en frame, samt et venstre og højre panel til at danne rammen for GUI'en. Inde i både højre og venstre panel placeres et ekstra panel, som indeholder komponenterne, for at sikre at resizing af vinduet ikke ændrer på komponenternes placering og størrelse. Derudover laves en specialiseret panel klasse, til at håndtere persondetaljerne, idet denne skal lytte på listen og vise informationerne for en person ved valg heraf.

Kode 7.2: Swing GUI kode

```
1     JFrame frame = new JFrame("Swing Person Manager");
2
3     JPanel leftpanel = new JPanel();
4
5     final JTextField searchtext = new JTextField();
6     searchtext.setPreferredSize(new Dimension(150,20));
7     searchtext.setMaximumSize(new Dimension(150,20));
8
9     final MyJList list = new MyJList();
10    fillList(list);
11    list.setPreferredSize(new Dimension(150,300));
12    list.setMaximumSize(new Dimension(150,300));
13
14    JPanel innerleft = new JPanel();
15    leftpanel.add(innerleft);
16    innerleft.setLayout(new BorderLayout(5,5));
17    innerleft.add(searchtext, BorderLayout.NORTH);
18    innerleft.add(list, BorderLayout.CENTER);
19
20    JPanel rightpanel = new JPanel();
21    final InfoPanel infopanel = new InfoPanel();
22    rightpanel.add(infopanel);
23
24    frame.setLayout(new BoxLayout(frame.getContentPane(), BoxLayout.X_AXIS));
25    frame.add(leftpanel);
26    frame.add(rightpanel);
27
28    class InfoPanel extends JPanel implements javax.swing.event.
          ListSelectionListener {
29        JLabel namelabel = new JLabel("Name:");
30        JLabel addresslabel = new JLabel("Address:");
31        JLabel phonelabel = new JLabel("Phone:");
32        JTextField nametextfield = new JTextField();
33        JTextField addresstextfield = new JTextField();
34        JTextField phonetextfield = new JTextField();
35        JButton removebutton = new JButton("Remove");
36        JButton addbutton = new JButton("Add");
37
38        public InfoPanel() {
39            setPreferredSize(new Dimension(300,100));
40            setLayout(new GridLayout(4,2,5,5));
41            add(namelabel);
42            add(nametextfield);
43            add(addresslabel);
44            add(addresstextfield);
45            add(phonelabel);
46            add(phonetextfield);
47            add(removebutton);
48            add(addbutton);
49        }
50    }
```

Swing koden for GUI'en delen er rodet, og viser ikke klart forhold imellem komponenterne. Den imperative tilgang giver ikke det samme klare overblik som den deklarative.

7.1.3 Sammenligning af Swing og HGL

De to kodeeksempler 7.1 og 7.2 viser tydeligt forskellen mellem en implementering i HGL og Swing. Hvor GUI opbygning fremgår klart i HGL implementeringen, fremgår den mindre klart i Swing implementeringen, og virker nærmest rodet. Swing versionen involverer flere layout managers til at placere komponenterne, hvilket gør koden svær at gennemskue, med mindre man kender sine layout managers godt. HGL versionen derimod, kræver blot at man kender forskellen mellem horisontal og vertikal placering samt en `@Width` annotering.

En yderligere analyse, jf. tabel 7.1, af de komplette eksempler som findes i bilag C, viser at antallet af forskellige kodekonstruktioner der benyttes til implementeringen, reduceres med 37% i HGL versionen, set i forhold til Swing versionen.

Tabel 7.1: Sammenligning af konstruktioner i SwingPersonManager og HGLPersonManager

Konstruktion	**SwingPersonManager**	**HGLPersonManager**
Metode	31	18
Klasse	17	7
Interface	1	4
Annotering	0	4
Konstant	4	1
I alt	**53**	**34**

Selv om kapitlet kun fokuserer på forskellen mellem HGL og Swing GUI opbygning, vil forskellen og dermed simplificeringen af eventhåndtering og den implementerede liste (HGL `List`), tydeligt fremgå ved inspektion af koden i bilaget.

Kapitel 8

Konklusion og videre arbejde

Jeg har i specialet undersøgt en række kendte GUI biblioteker, og belyst deres positive såvel som negative sider, i forbindelse med konstruktion af brugergrænseflader. Derudover har jeg også undersøgt to alternative biblioteker Buoy og CookSwing, der begge forsøger at bidrage med alternative og mere simple tilgange til GUI konstruktion med Swing.

Gennem analysen af de forskellige biblioteker, og særligt Swing, har jeg belyst en række problemer i forbindelse med GUI konstruktion med Swing. Ved brug af viden omkring problemerne med Swing, og mulighederne i de andre undersøgte biblioteker, har jeg opstillet en række designmål for et alternativt Swing bibliotek til Java. Et bibliotek hvis intension og målsætning er, at simplificere GUI konstruktion for Swing.

I forbindelse med mine kritikpunkter og analyse af Swing, samt den endelige konklusion af biblioteket, opstår et kritisk og manglende element. En videnskabelig tilgang til vurdering af problemerne og kompleksiteten forbundet med Swing, er ikke tilstede. Hvor jeg blot har benyttet mine personlige erfaringer, samt beklagende fortællinger fra andre objekt orienterede programmerings (OOP) studerende, som målestok for hvad der er simpelt, burde der som fundament istedet ligge en række videnskabeligt underbyggede retningslinjer for hvad simpel og intuitiv kode er.

Udfra de opsatte designmål har jeg designet et bibliotek, hvorfra jeg kan anlægge to overordnede synspunkter, når jeg skal konkludere på resultatet heraf.

Første synspunkt er at betragte designet af biblioteket udfra de opsatte designmål isoleret. Jeg mener at designet af biblioteket, samt taget i betragtning hvad der er muligt i Java som programmeringssprog, er realiseret indenfor rammerne af de opsatte designmål med en enkelt undtagelse. Designmål D3 der siger “at alle fejl skal fanges ved kompileringstid” overholdes

ikke, grundet angivelsen af metodenavne som tekststrenge i forbindelse med events, og brugen af anonyme indreklasser og den dertilhørende typeproblematik i forbindelse med strukturen.

Et andet og mere interessant synspunkt er, at betragte biblioteket udfra den overordnede målsætning. Jeg mener at intensionen om at gøre GUI programmering med Swing nemmere og mere intuitiv er opnået. Jeg har implementeret tiltag for de forskellige kritikpunkter jf. kapitel 5, der som resultat har simplificeret GUI konstruktion med HGL sammenlignet med Swing.

8.1 Videre arbejde

Jeg ser fire retninger hvormed der kan arbejdes videre på specialet og dermed HGL; (1) en produktorienteret tilgang, (2) en sprogorienteret tilgang, (3) en udvidelsesorienteret tilgang og (4) en målsætningsorienteret tilgang. De fire retninger afspejler de tanker og ideer jeg igennem specialet ikke har nået at realisere, eller som har ligget uden for min problemformulering.

Produktorienteret tilgang

Hvis jeg havde haft mere tid til specialet, ville jeg efter proof-of-concept implementeringen anvende en produktorienteret tilgang til færdiggørelse af biblioteket. En sådan tilgang ville involvere følgende punkter:

- Implementering af en række Swing komponenter såsom `JMenuBar`, `JCheckBox`, osv.
- Angivelse af en række "ordentlige" standard værdier for layoutinformationerne.
- Udarbejdelse af diverse tutorials og slutbruger JavaDocs.
- Implementering af en gennemarbejdet fejlhåndteringsmekanisme.
- Definering og udarbejdelse af et testmiljø for biblioteket.

HGL ville skulle udvides med en række ekstra komponenter, og dermed indeholde en række komponenter, der gør det muligt at konstruere en mere beriget GUI, end blot et vindue med et tekstfelt, label, knap og en liste. Samtidig ville de forskellige layoutinformationsannoteringer skulle tildeles en fornuftig standard værdi, så brugerne i første omgang ikke behøvede, at fokusere på både struktur og layout men blot struktur. Derudover ville der skulle udarbejdes en række tutorials, samt dokumentation der lærer nybegyndere, og dermed målgruppen, at anvende biblioteket. Et manglende element som en

gennemarbejdet fejlhåndteringsmekanisme, ville ligeledes skulle indarbejdes i HGL, hvilket er et område som kun er behandlet overfladisk i specialet. Til sidst ville der skulle defineres og udarbejdes en måde hvorpå HGL kunne testes på, et punkt som specialet overhovedet ikke berører.

Sprogorienteret tilgang

Igennem specialets forløbet har jeg udviklet en større interesse for biblioteksdesign, og dermed programmeringssproget der benyttes til at implementere biblioteket. Jeg har med Java som programmeringssprog, igennem designet og implementeringen af HGL, oplevet at sproget (dvs. dets konstruktioner) sætter begrænsninger for designmulighederne. Konkret er jeg stødt på typeproblemetikken med anonyme indreklasser, og mangel på kontrol af objektoprettelsesrækkefølgen, og gennem et eventuelt videre forløb med design og implementering af HGL, ville der sikkert dukke flere sprogbegrænsninger op. I den forbindelse og i forlængelse af artiklen "On the Role of Language Constructs for Framework Design"[15], kunne et separat studie heri, eventuelt med HGL som case, være et spændende videre forløb.

Udvidelsesorienteret tilgang

Med en udvidelsesorienteret tilgang forstås, hvordan biblioteket i brugerscenarier kan udvides eller specialiseres med brugerdefineret komponenter. Ønskes eksempelvis en specialiseret knap, kræver den nuværende arkitektur at brugeren specialiserer hele klassehierarkiet, dvs. både `Frame`, `Panel` og de komponenter der indsættes heri. Selvom målgruppen ikke ligger inden for rammerne af en udvidelsesorienteret tilgang, idet nybegyndere højest sandsynligt ikke har i sinde at udvide og specialisere HGL komponenter, er emnet relevant og spændende. Hvordan smidiggøres et så fastlåst og restriktionsfyldt bibliotek, og skal udvidelser overhovedet være mulige, samt kan implementeringen ved brug af indreklasser, overhovedet anvendes i forbindelse med udvidelser, er alle centrale spørgsmål i forbindelse med udvidelsesmuligheder for HGL.

Målsætningsorienteret tilgang

Hvis man anskuer målsætningen udfra et mere generelt perspektiv, og fjerner fokus specifikt fra Swing og Java, og dermed ser på simplificering af GUI programmering uafhængigt af sprog og biblioteker, mener jeg ikke at HGL er simpelt nok. Trods det at HGL simplificerer GUI programmering med Swing, tror jeg at GUI i fremtiden søges konstrueret ved brug af kompileringssikkert HTML, XHTML eller XML, med dertilhørende XSLT eller CCS

til brug for layout behandling. Jeg mener at Microsoft i deres nye version af ASP.NET kaldet ASP.NET 2.0, har taget et korrekt tiltag, et tiltag som sikrer at kode angivet i HTML, præ-kompiles inden afvikling af programmet. Hvor der i den tidligere version ville opstår runtime fejl, opstår fejl nu på kompileringstidspunktet.

Udover kompileringssikkert XML ville en sammensmeltning af udvikling for GUI til web og GUI til applikationer (eks. windows applikationer), så programmører ikke behøver at kende til flere forskellige platforme og biblioteker, være et centralt emne for en mere generaliseret målsætning.

Litteratur

[1] Asberry, D., Bauer, G., Cardon, D., Carr, D., Kvassov, V., Mejdi, A., and Phalip, J. Luxor - xml ui language (xul) toolkit. http://luxor-xul.sourceforge.net/.

[2] Borland. Delfi 2005. http://www.borland.com/delphi/.

[3] Borland. Jbuilder. http://www.borland.com/jbuilder/.

[4] Bracha, G. Generics in the java programming language. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[5] Corporation, M. Longhorn developer center. http://msdn.microsoft.com/longhorn/.

[6] Eastman, P. Buoy: A better user interface toolkit. http://buoy.sourceforge.net/AboutBuoy.html.

[7] Eckel, B. *Thinking in Java 3rd edition*. Prentice Hall, PTR, 2003.

[8] eclipse. Swt standard widget toolkit. http://www.eclipse.org/swt/.

[9] Foundation, A. S. Jellyswing. http://jakarta.apache.org/commons/jelly/libs/swing/.

[10] foundation, X. X.org. http://x.org.

[11] Fowler, A. A swing architecture overview. http://java.sun.com/products/jfc/tsc/articles/architecture/.

[12] Fowler, M. *Refactoring Improving the Design of Existing Code*. Addison-Wesley, 2000.

[13] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison-Wesley, 1995.

[14] Hansen, M. R., and Rischel, H. *Introduction to Programming Using SML*. Addison-Wesley, 1999.

[15] Hedin, G., and Knudsen, J. L. On the role of language constructs for framework design. *ACM* (2000).

[16] Informatics, M. Lidskjalv: User interface framework - tutorial. `http://www.daimi.au.dk/~beta/Manuals/latest/lidskjalv-tut/`.

[17] Januszewski, K., Sneath, T., and Cohen, A. Avalon november 2004 community technology preview. `http://msdn.microsoft.com/Longhorn/understanding/pillars/avalon/avnov04ctp/default.aspx`.

[18] Krasner, G. E., and Pope, S. T. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program. 1*, 3 (1988), 26–49.

[19] Liberty, J. *Programming C# 3rd edition*. O'Reilly, 2003.

[20] Loy, M., Eckstein, R., Wood, D., Elliott, J., and Cole, B. *Java Swing*, 2nd ed. O'Reilly & Associates, Inc., 2003.

[21] Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[22] Microsoft. Visual studio .net. `http://msdn.microsoft.com/vstudio/`.

[23] Microsystems, S. The awt in 1.0 and 1.1. `http://java.sun.com/products/jdk/awt/`.

[24] Microsystems, S. Java 2d api. `http://java.sun.com/products/java-media/2D/index.jsp`.

[25] Microsystems, S. Java foundation classes (jfc/swing). `http://java.sun.com/products/jfc/index.jsp`.

[26] Microsystems, S. Javatm 2 platform standard edition 5.0 api specification. `http://java.sun.com/j2se/1.5.0/docs/api/`.

[27] Myers, B. A. *Languages for Developing User Interfaces*. Jones and Bartlett Publishers, Inc., 1992.

[28] Oracle. Jdeveloper. `http://www.oracle.com/technology/products/jdev/index.html`.

[29] Rowe, G. W. *An Introduction to Data Structures and Algorithms with Java*. Prentice Hall, 1998.

[30] Sestoft, P. *Programming Language Concepts*. Royal Veterinary and Agricultural University and IT University of Copenhagen, 2003.

[31] Sestoft, P., and Hansen, H. I. *C# Precisely*. Massachusetts Institute of Technology, 2004.

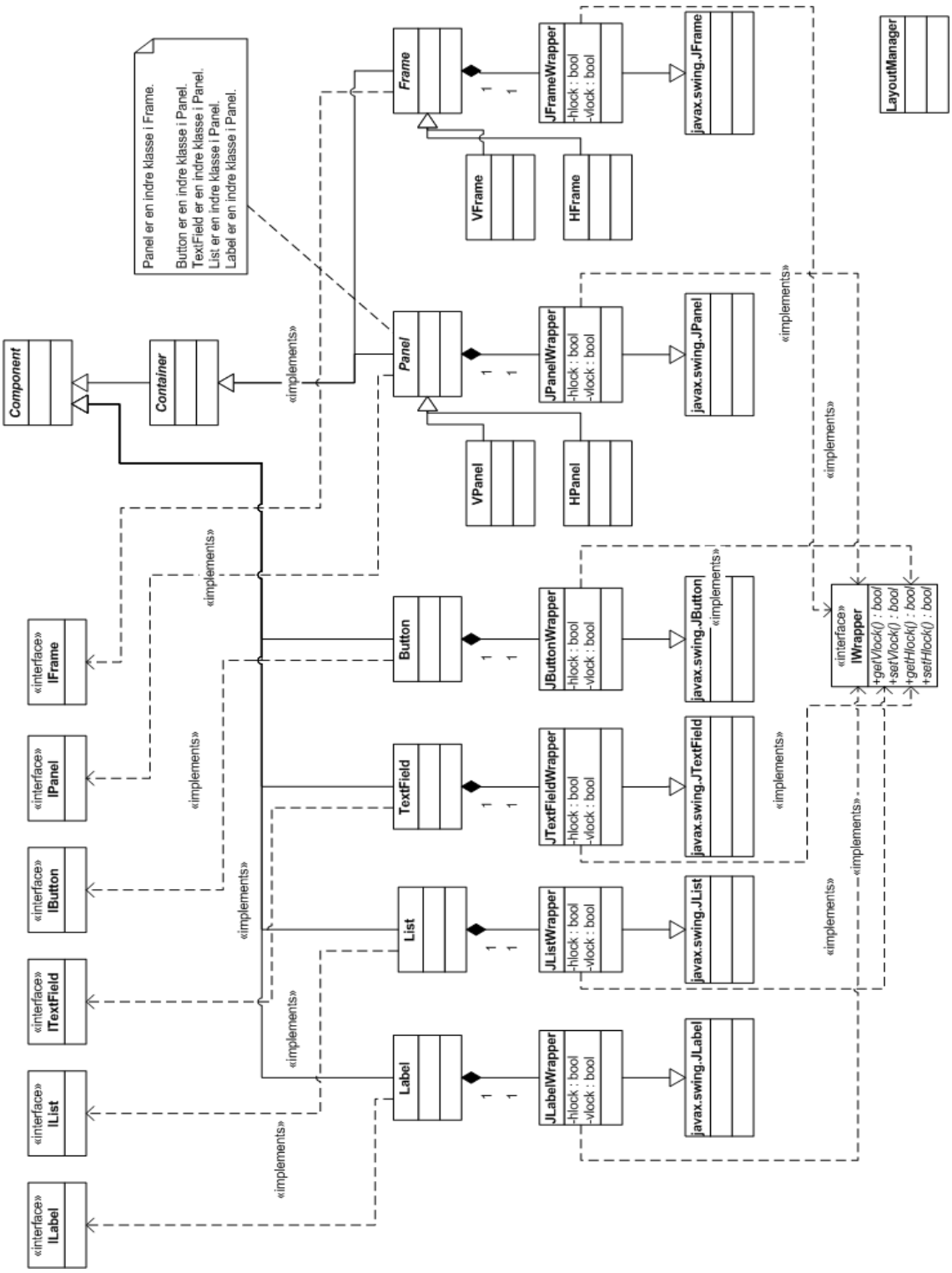
[32] W3C. Cascading style sheets. `http://www.w3.org/Style/CSS/`.

[33] W3C. Hypertext markup language (html). `http://www.w3.org/MarkUp/`.

[34] Yuan, H. Cookswing: Xml to swing gui. `http://cookxml.sourceforge.net/cookswing/index.html`.

Bilag A

Arkitektur version 1


Component
Container
Panel er en indre klasse i Frame.
Button er en indre klasse i Panel.
TextField er en indre klasse i Panel.
List er en indre klasse i Panel.
Label er en indre klasse i Panel.
Frame
Panel
Button
TextField
List
Label
VFrame
HFrame
VPanel
HPanel
JFrameWrapper
JPanelWrapper
JButtonWrapper
JTextFieldWrapper
JListWrapper
JLabelWrapper
-hlock : bool
-vlock : bool
javax.swing.JFrame
javax.swing.JPanel
javax.swing.JButton
javax.swing.JTextField
javax.swing.JList
javax.swing.JLabel
LayoutManager
«interface»
IWrapper
+getVlock() : bool
+setVlock() : bool
+getHlock() : bool
+setHlock() : bool
«interface»
IFrame
«interface»
IPanel
«interface»
IButton
«interface»
ITextField
«interface»
IList
«interface»
ILabel
«implements»

Bilag B

JList versus List

Kode B.1: JList

```
1         public class SimpelJList {
2             private abstract class MyListListener implements
                  ListSelectionListener {
3                 public void valueChanged(ListSelectionEvent event) {}
4             }
5             JFrame frame = new JFrame();
6             JPanel panel = new JPanel();
7             JList list = new JList();
8             DefaultListModel model = new DefaultListModel();
9
10            public static void main(String[] args) {
11                SimpelJList test = new SimpelJList();
12                test.run();
13            }
14
15            public void run() {
16                frame.add(panel);
17                panel.add(list);
18                list.addListSelectionListener(new SimpelJList.MyListListener(){
19                    public void valueChanged(ListSelectionEvent event) {
20                        if (!event.getValueIsAdjusting())
21                            System.out.println(((JList)event.getSource()).
                                  getSelectedValue());
22                    }
23                });
24                model.addElement("item");
25                list.setModel(model);
26                list.setSize(100,300);
27                frame.pack();
28                frame.setVisible(true);
29            }
30        }
```

Kode B.2: List

```
1         public class SimpelList{
2             Frame frame = new Frame("HGL"){
3                 Panel panel = new Panel(){
4                     @Width(100)
5                     @Height(300)
6                     List list = new List();
7                 };
8             };
9
10            public static void main(String[] args) {
11                SimpelList test = new SimpelList();
12                test.run();
13            }
14
15            public void run() {
16                frame.setup();
17                IList list = frame.get("panel.list",IList.class);
18                list.addItem("item");
19                list.onSelectedItemChanged(this,"print",null);
20                frame.display();
21            }
22
23            public void print() {
24                IList list = frame.get("panel.list",IList.class);
25                System.out.println(list.getSelectedItem());
26            }
27        }
```

Bilag C

Personmanager

C.1 SwingPersonManager.java

```
1
2  import javax.swing.*;
3  import java.awt.*;
4  import java.awt.event.KeyAdapter;
5  import java.awt.event.KeyEvent;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.util.ArrayList;
9
10 public class SwingPersonManager {
11
12     ArrayList<Person> persons = Person.getPerson();
13     ArrayList<Person> filteredpersons = Person.getPerson();
14
15     public SwingPersonManager() {
16         JFrame frame = new JFrame("Swing Person Manager");
17
18         JPanel leftpanel = new JPanel();
```

```
19
20  final JTextField searchtext = new JTextField();
21  searchtext.setPreferredSize(new Dimension(150,20));
22  searchtext.setMaximumSize(new Dimension(150,20));
23
24  final MyJList list = new MyJList();
25  fillList(list);
26  list.setPreferredSize(new Dimension(150,300));
27  list.setMaximumSize(new Dimension(150,300));
28
29  JPanel innerleft = new JPanel();
30  leftpanel.add(innerleft);
31  innerleft.setLayout(new BorderLayout(5,5));
32  innerleft.add(searchtext, BorderLayout.NORTH);
33  innerleft.add(list, BorderLayout.CENTER);
34
35  JPanel rightpanel = new JPanel();
36  final InfoPanel infopanel = new InfoPanel();
37  rightpanel.add(infopanel);
38
39  frame.setLayout(new BoxLayout(frame.getContentPane(), BoxLayout.X_AXIS));
40  frame.add(leftpanel);
41  frame.add(rightpanel);
42
43  searchtext.addKeyListener(new KeyAdapter(){
44      public void keyReleased(KeyEvent event) {
45          handleSearch(searchtext.getText(), list);
46      }
47  });
48
49  list.addListSelectionListener(infopanel);
50
51  infopanel.addbutton.addActionListener(new ActionListener(){
52      public void actionPerformed(ActionEvent e) {
53          String name = infopanel.nametextfield.getText();
54          String address = infopanel.addresstextfield.getText();
55          String phone = infopanel.phonetextfield.getText();
56          Person p = new Person(name, address, phone);
57          list.addElement(p);
58          persons.add(p);
59      }
60  });
61
62  infopanel.removebutton.addActionListener(new ActionListener(){
63      public void actionPerformed(ActionEvent e) {
64          list.removeElement(list.getSelectedValue());
65      }
66  });
67
68  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
69          frame.pack();
70          frame.setVisible(true);
71      }
72
73      void handleSearch(String search, MyJList list) {
74          filteredpersons.clear();
75          for (Person p : persons)
76              if (p.name.toLowerCase().startsWith(search.toLowerCase()))
77                  filteredpersons.add(p);
78
79          fillList(list);
80
81          if (list.numerOfElements() == 1)
82              list.setSelectedIndex(0);
83      }
84
85      public void fillList(MyJList list) {
86          list.clear();
87          for (Person p : filteredpersons)
88              list.addElement(p);
89      }
90
91      class MyJList extends JList {
92          private DefaultListModel model = new DefaultListModel();
93
94          public MyJList() {
95              super.setModel(model);
96          }
97
98          public void addElement(Object o) {
99              model.addElement(o);
100         }
101
102         public void clear() {
103             model.clear();
104         }
105
106         public int numerOfElements() {
107             return model.size();
108         }
109
110         public void removeElement(Object o) {
111             model.removeElement(o);
112         }
113     }
114
115     class InfoPanel extends JPanel implements javax.swing.event.ListSelectionListener {
116         JLabel namelabel = new JLabel("Name:");
117         JLabel addresslabel = new JLabel("Address:");
118         JLabel phonelabel = new JLabel("Phone:");
```

```
119         JTextField nametextfield = new JTextField();
120         JTextField addresstextfield = new JTextField();
121         JTextField phonetextfield = new JTextField();
122         JButton removebutton = new JButton("Remove");
123         JButton addbutton = new JButton("Add");
124
125         public InfoPanel() {
126             setPreferredSize(new Dimension(300,100));
127             setLayout(new GridLayout(4,2,5,5));
128             add(namelabel);
129             add(nametextfield);
130             add(addresslabel);
131             add(addresstextfield);
132             add(phonelabel);
133             add(phonetextfield);
134             add(removebutton);
135             add(addbutton);
136         }
137
138         public void valueChanged(javax.swing.event.ListSelectionEvent event) {
139             if (!event.getValueIsAdjusting()) {
140                 Person p = (Person)((MyJList)event.getSource()).getSelectedValue();
141                 if (p != null) {
142                     nametextfield.setText(p.name);
143                     addresstextfield.setText(p.address);
144                     phonetextfield.setText(p.phone);
145                 } else {
146                     nametextfield.setText("");
147                     addresstextfield.setText("");
148                     phonetextfield.setText("");
149                 }
150             }
151         }
152     }
153
154     public static void main(String[] args) {
155         SwingPersonManager spm = new SwingPersonManager();
156     }
157 }
```

C.2 HGLPersonManager.java

```
1  import dk.itu.hgl.*;
2
```

```
3  import java.util.ArrayList;
4
5  public class HGLPersonManager {
6
7      Frame frame = new Frame("Person Manager") {
8
9          @Layout(Layout.Orientation.VERTICAL)
10         Panel leftpanel = new Panel() {
11             @Width(150)
12             TextField searchtextfield = new TextField();
13             @Width(150)
14             @Height(300)
15             List list = new List();
16         };
17
18         @Layout(Layout.Orientation.VERTICAL)
19         @Padding(0)
20         Panel rightpanel = new Panel() {
21
22             Panel namepanel = new Panel() {
23                 @Width(100)
24                 Label namelabel = new Label("Name:");
25                 @Width(200)
26                 TextField nametextfield = new TextField();
27             };
28
29             Panel addresspanel = new Panel() {
30                 @Width(100)
31                 Label addresslabel = new Label("Address:");
32                 @Width(200)
33                 TextField addresstextfield = new TextField();
34             };
35
36             Panel phonepanel = new Panel() {
37                 @Width(100)
38                 Label phonelabel = new Label("Phone:");
39                 @Width(200)
40                 TextField phonetextfield = new TextField();
41             };
42
43             Panel addpanel = new Panel() {
44                 IButton removebutton = new Button("Remove");
45                 IButton addbutton = new Button("Add");
46             };
47         };
48     };
49
50     ArrayList<Person> persons = Person.getPerson();
51     ArrayList<Person> filteredpersons = Person.getPerson();
52
```

```
53  public HGLPersonManager() {
54      frame.setup();
55
56      ITextField search = frame.get("leftpanel.searchtextfield",ITextField.class);
57      IList list = frame.get("leftpanel.list",IList.class);
58      IButton additem = frame.get("rightpanel.addpanel.addbutton",IButton.class);
59      IButton removeitem = frame.get("rightpanel.addpanel.removebutton",IButton.class);
60
61      list.setItems(filteredpersons);
62
63
64      search.onTextChanged(this,"handleSearch",search,list);
65
66      list.onSelectedItemChanged(this,"fillRightPanel",list);
67
68      additem.onClick(this,"add",list);
69
70      removeitem.onClick(this,"remove",list);
71
72      frame.display();
73  }
74
75  public void handleSearch(ITextField search, IList list) {
76      filteredpersons.clear();
77      for (Person p : persons) {
78          if (p.name.toLowerCase().startsWith(search.getText().toLowerCase()))
79              filteredpersons.add(p);
80      }
81
82      list.setItems(filteredpersons);
83
84      if (list.size() == 1)
85          list.setSelectedItemIndex(0);
86  }
87
88  public void fillRightPanel(IList list) {
89      ITextField name = frame.get("rightpanel.namepanel.nametextfield",ITextField.class);
90      ITextField address = frame.get("rightpanel.addresspanel.addresstextfield",ITextField.class);
91      ITextField phone = frame.get("rightpanel.phonepanel.phonetextfield",ITextField.class);
92
93      Person p = (Person)list.getSelectedItem();
94      if (p != null) {
95          name.setText(p.name);
96          address.setText(p.address);
97          phone.setText(p.phone);
98      } else {
99          name.setText("");
100         address.setText("");
101         phone.setText("");
102     }
```

```
103     }
104
105     public void add(IList list) {
106         ITextField name = frame.get("rightpanel.namepanel.nametextfield",ITextField.class);
107         ITextField address = frame.get("rightpanel.addresspanel.addresstextfield",ITextField.class);
108         ITextField phone = frame.get("rightpanel.phonepanel.phonetextfield",ITextField.class);
109         Person p = new Person(name.getText(), address.getText(), phone.getText());
110         list.addItem(p);
111         persons.add(p);
112     }
113
114     public void remove(IList list) {
115         list.removeItem(list.getSelectedItem());
116     }
117
118     public static void main(String[] args) {
119         HGLPersonManager test = new HGLPersonManager();
120     }
121 }
```

Bilag D

Kode

D.1 Component.java

```
1  package dk.itu.hgl;
2
3  import java.util.HashMap;
4  import java.util.*;
5  import java.lang.reflect.Method;
6  import java.lang.reflect.InvocationTargetException;
7
8  abstract class Component<T extends java.awt.Container> implements IComponent {
9      // HashMap for storing layoutinformations (annotations).
10     // The key in the first hashmap is the Swing component and the value is
11     // a hashmap containg the name of the layoutinformation and the value.
12     static protected HashMap<Object, HashMap<String, Object>> attributes = new HashMap<Object, HashMap<String, Object>>(100);
13     // The incapsulated Swing component
14     protected T swingcomponent;
15     private static int fieldCounter = 0;
16     protected int fieldNumber;
17
18     public Component() {
```

```
19          fieldCounter++;
20          fieldNumber = fieldCounter;
21      }
22
23      synchronized protected void writeAttribute(String name, Object value) {
24          writeAttribute(name, value, null);
25      }
26
27      synchronized protected void writeAttribute(String name, Object value, javax.swing.JFrame jframe) {
28          Object swingcomponenttowrite = jframe == null ? swingcomponent : jframe.getContentPane();
29
30          if (name == null || value == null)
31              throw new IllegalArgumentException();
32          HashMap<String, Object> attributeMap = attributes.get(swingcomponenttowrite);
33          if (attributeMap != null) {
34              attributeMap.put(name, value);
35          } else {
36              attributeMap = new HashMap<String, Object>(5);
37              attributeMap.put(name, value);
38              attributes.put(swingcomponenttowrite, attributeMap);
39          }
40      }
41
42      static public Object readAttribute(Object o, String attributeName) {
43          HashMap<String, Object> attributeMap = attributes.get(o);
44          if (attributeMap != null) {
45              Object value = attributeMap.get(attributeName);
46              return value;
47          }
48          return null;
49      }
50
51      private static void printAllAttributes() {
52          Collection<HashMap<String,Object>> c = attributes.values();
53      }
54
55      // Event handling
56
57      void mouseEntered(final Object target, final String targetmethod, final Object... targetmethodargs) {
58          swingcomponent.addMouseListener(new java.awt.event.MouseAdapter() {
59              public void mouseEntered(java.awt.event.MouseEvent ev) {
60                  try {
61                      Method m = findMethod(target, targetmethod, targetmethodargs);
62                      m.setAccessible(true);
63                      Object result = m.invoke(target,targetmethodargs);
64                  } catch (NoSuchMethodException e) {
65                      System.out.print(e.toString());
66                  } catch (IllegalAccessException e) {
67                      System.out.print(e.toString());
68                  } catch (InvocationTargetException e) {
```

```
69                  System.out.print(e.toString());
70              }
71          }
72      });
73  }
74
75  void keyReleased(final Object target, final String targetmethod, final Object... targetmethodargs) {
76      swingcomponent.addKeyListener(new java.awt.event.KeyAdapter() {
77          public void keyReleased(java.awt.event.KeyEvent ev) {
78              try {
79                  Method m = findMethod(target, targetmethod, targetmethodargs);
80                  m.setAccessible(true);
81                  Object result = m.invoke(target,targetmethodargs);
82              } catch (NoSuchMethodException e) {
83                  System.out.print(e.toString());
84              } catch (IllegalAccessException e) {
85                  System.out.print(e.toString());
86              } catch (InvocationTargetException e) {
87                  System.out.print(e.toString());
88              }
89          }
90      });
91  }
92
93  void propertyChange(final Object target, final String targetmethod, final Object... targetmethodargs) {
94      swingcomponent.addPropertyChangeListener(new java.beans.PropertyChangeListener(){
95          public void propertyChange(java.beans.PropertyChangeEvent evt) {
96              try {
97                  Method m = findMethod(target, targetmethod, targetmethodargs);
98                  m.setAccessible(true);
99                  Object result = m.invoke(target,targetmethodargs);
100             } catch (NoSuchMethodException e) {
101                 System.out.print(e.toString());
102             } catch (IllegalAccessException e) {
103                 System.out.print(e.toString());
104             } catch (InvocationTargetException e) {
105                 System.out.print(e.toString());
106             }
107         }
108     });
109 }
110
111 // Helper methods for event handling
112
113 Class[] convertObjectArrayToClassArray(Object... target) {
114     if (target == null)
115         return new Class[0];
116
117     int length = target.length;
118
```

```
119
120         if (length == 0)
121             return new Class[0];
122
123         Class[] types = new Class[length];
124         for (int i = 0; i < length; i++) {
125             types[i] = target[i].getClass();
126         }
127
128         return types;
129     }
130
131     Method findMethod(Object target, String targetmethod, Object... targetmethodargs) throws NoSuchMethodException {
132         ArrayList<Method> candidateMethods = new ArrayList<Method>();
133         Class[] targetmethodargsTypes = convertObjectArrayToClassArray(targetmethodargs);
134         Class targetclass = target.getClass();
135         for(Method m : targetclass.getDeclaredMethods()) {
136             if (m.getName() == targetmethod && m.getParameterTypes().length == targetmethodargsTypes.length) {
137                 // Check to see if the parameters are legal subclasses
138                 Boolean match = true;
139                 for(int i=0; i < m.getParameterTypes().length; i++) {
140                     Class c1 = m.getParameterTypes()[i];
141                     Class c2 = targetmethodargsTypes[i];
142
143                     if (c1.isAssignableFrom(c2) == false) {
144                         match = false;
145                         break;
146                     }
147                 }
148                 if (match)
149                     candidateMethods.add(m);
150             }
151         }
152
153         // If we have more than one candidate method find the most specific one
154         Method correctMethod = candidateMethods.size() > 0 ? candidateMethods.get(0) : null;
155         boolean twoIdenticalMethods = false;
156         if (candidateMethods.size() > 1) {
157             int compare = 0;
158             for (int i=0; i < candidateMethods.size()-1; i++) {
159                 compare = compareMethod(correctMethod, candidateMethods.get(i+1));
160
161                 switch(compare) {
162                     case 0:
163                         twoIdenticalMethods = true;
164                         correctMethod = candidateMethods.get(i);
165                         break;
166                     case 1:
167                         twoIdenticalMethods = false;
168                     case 2:
```

```
169                         twoIdenticalMethods = false;
170                         correctMethod = candidateMethods.get(i+1);
171                     }
172                 }
173             }
174
175             if (candidateMethods.size() == 0 || twoIdenticalMethods == true)
176                 throw new NoSuchMethodException("Method "+ targetmethod +" could not be found in class "+ target);
177             else
178                 return correctMethod;
179         }
180
181         // Compares two methods by comparing their types.
182         // 0 is returned if m1 equals m2.
183         // 1 is returned if m1 is more specific.
184         // 2 is returned if m2 is more specific.
185         int compareMethod(Method m1, Method m2){
186             int m1Counter = 0;
187             int m2Counter = 0;
188
189             Class[] m1Types = m1.getParameterTypes();
190             Class[] m2Types = m2.getParameterTypes();
191
192             for (int i=0; i<m1Types.length; i++) {
193                 if (m1Types[i].isAssignableFrom(m2Types[i]))
194                     m2Counter++;
195                 if (m2Types[i].isAssignableFrom(m1Types[i]))
196                     m1Counter++;
197             }
198
199             if (m1Counter == m2Counter)
200                 return 0;
201             else
202                 return m1Counter > m2Counter ? 1 : 2;
203         }
204 }
```

D.2 Container.java

```
1  package dk.itu.hgl;
2
3  import java.util.*;
4  import java.lang.reflect.Field;
5  import java.lang.annotation.Annotation;
```

```
6
7  abstract class Container<T extends java.awt.Container> extends Component<T> {
8      protected HashMap<String, IComponent> components = new HashMap<String, IComponent>();
9
10     private class ComponentReflectiveAddException extends RuntimeException {
11         ComponentReflectiveAddException(String s) {
12             super(s);
13             System.out.println(s);
14         }
15     }
16
17     private class ComponentNotFoundException extends RuntimeException {
18         ComponentNotFoundException(String s) {
19             super(s);
20             System.out.println(s);
21         }
22     }
23
24     private Vector<Component> filterAndSortFields(Field[] fields, Container caller) {
25         Object fieldValue = null;
26         Vector<Component> v = new Vector<Component>();
27         for(Field field : fields) {
28             field.setAccessible(true);
29             try {
30                 fieldValue = field.get(this);
31                 //System.out.println("Looking for "+ field.getName() +" in "+this);
32             } catch (IllegalAccessException e) {
33                 throw new ComponentReflectiveAddException("Unable to get field value for field: " + field.getName());
34             }
35             if (fieldValue != caller && fieldValue instanceof Component) {
36                 // used for sorting components
37                 v.add((Component)fieldValue);
38                 // used for containers hashmap which is used by the get(...) method
39                 this.components.put(field.getName(), (Component) fieldValue);
40                 // used for layoutinformations
41                 handleAnnotations(field, fieldValue);
42             }
43         }
44         Collections.sort(v, new Comparator<Component>() {
45             public int compare(Component c, Component c2) {
46                 if (c.fieldNumber > c2.fieldNumber)
47                     return 1;
48                 if (c.fieldNumber < c2.fieldNumber)
49                     return -1;
50                 else
51                     return 0;
52             }
53         });
54         return v;
55     }
```

```
56
57  protected void addComponenets(Container caller) throws ComponentReflectiveAddException {
58      // A vector containing all children components
59      Vector<Component> components = filterAndSortFields(getClass().getDeclaredFields(), caller);
60
61      // Adding all children components to the original Java swingcomponent, ex. JFrame
62      for(Component c : components) {
63          swingcomponent.add(c.swingcomponent);
64
65          // If the swingcomponent it self is a Container make a recursive call to addComponents(...)
66          if (c instanceof Container)
67              ((Container)c).addComponenets(this);
68      }
69  }
70
71  void handleAnnotations(Field field, Object fieldValue) {
72      handleAnnotations(field, fieldValue, null);
73  }
74
75  void handleAnnotations(Field field, Object fieldValue, javax.swing.JFrame jframe) {
76      Annotation[] annotations = field.getDeclaredAnnotations();
77      for(Annotation annotation : annotations) {
78          Class type = annotation.annotationType();
79          if (type == Hlock.class) {
80              ((Component)fieldValue).writeAttribute("hlock", ((Hlock)annotation).value(), jframe);
81              continue;
82          }
83          if (type == Vlock.class) {
84              ((Component)fieldValue).writeAttribute("vlock", ((Vlock)annotation).value(), jframe);
85              continue;
86          }
87          if (type == Layout.class) {
88              ((Component)fieldValue).writeAttribute("layout", ((Layout)annotation).value(), jframe);
89              continue;
90          }
91          if (type == Width.class) {
92              if (((Width)annotation).value() > 0 )
93                  ((Component)fieldValue).writeAttribute("width", ((Width)annotation).value(), jframe);
94              continue;
95          }
96          if (type == Height.class) {
97              if (((Height)annotation).value() > 0 )
98                  ((Component)fieldValue).writeAttribute("height", ((Height)annotation).value(), jframe);
99              continue;
100         }
101         if (type == Padding.class) {
102             ((Component)fieldValue).writeAttribute("padding", ((Padding)annotation).value(), jframe);
103             continue;
104         }
105     }
```

```
106     }
107
108     public <T extends IComponent> T get(String key, Class<T> cl) {
109         int index = key.indexOf(".");
110         if (index > -1) {
111             IComponent c = components.get(key.substring(0, index));
112             if (c instanceof Container) {
113                 return (T) ((Container) c).get(key.substring(index + 1), cl);
114             }
115             throw new ComponentNotFoundException("Component: " + key + " was not found or is not container");
116         }
117         IComponent c = components.get(key);
118         if (c == null) {
119             throw new ComponentNotFoundException("Unknown name: " + key);
120         }
121         if (!cl.isAssignableFrom(c.getClass())) {
122             throw new ComponentNotFoundException("Component found does not match parameterized type");
123         }
124         return (T) components.get(key);
125     }
126
127     class LayoutManager implements java.awt.LayoutManager {
128
129         public void addLayoutComponent(String name, java.awt.Component comp) {
130         }
131
132         public void removeLayoutComponent(java.awt.Component comp) {
133         }
134
135         // Calculates the preferred size dimensions for the specified
136         // panel given the components in the specified parent container.
137         public java.awt.Dimension preferredLayoutSize(java.awt.Container parent) {
138             return getLayoutSize(parent, true);
139         }
140
141         // Calculates the minimum size dimensions for the specified
142         // panel given the components in the specified parent container.
143         public java.awt.Dimension minimumLayoutSize(java.awt.Container parent) {
144             return getLayoutSize(parent, false);
145         }
146
147         public void layoutContainer(java.awt.Container parent) {
148             int width = 0;
149             int height = 0;
150             int padding = Component.readAttribute(parent,"padding") == null ? 0 :
151                           (Integer)Component.readAttribute(parent,"padding");
152             int x = padding;
153             int y = padding;
154
155             // Handle Layout annotation.
```

```
156             Layout.Orientation orientation = Component.readAttribute(parent,"layout") == null ?
157                                 Layout.Orientation.HORISONTAL :
158                                 (Layout.Orientation)Component.readAttribute(parent,"layout");
159
160             // Handle VLock and HLock annotations, including the Size annotation when
161             // calculating the width and height. Size overrules the preferred size
162             // and anything else.
163             int noOfComponentsWithoutVLock = 0;
164             int noOfComponentsWithoutHLock = 0;
165             int totalWidthOfComponentsWithHLock = 0;
166             int totalHeightOfComponentsWithVLock = 0;
167             int totalSizeWidth = 0;
168             int totalSizeHeight = 0;
169
170             for (java.awt.Component c : parent.getComponents()) {
171                 boolean isVLock = (Boolean)Component.readAttribute(c,"vlock");
172                 boolean isHLock = (Boolean)Component.readAttribute(c,"hlock");
173                 int sizeWidth = Component.readAttribute(c,"width") == null ? 0 :
174                                 (Integer)Component.readAttribute(c,"width");
175                 int sizeHeight = Component.readAttribute(c,"height") == null ? 0 :
176                                  (Integer)Component.readAttribute(c,"height");
177
178                 totalSizeWidth += sizeWidth;
179                 totalSizeHeight += sizeHeight;
180
181                 if (isVLock) {
182                     totalHeightOfComponentsWithVLock += (sizeHeight > 0 ? 0 : c.getPreferredSize().height);
183                 }
184                 else {
185                     noOfComponentsWithoutVLock += sizeHeight > 0 ? 0 : 1;
186                 }
187
188                 if (isHLock) {
189                     totalWidthOfComponentsWithHLock += (sizeWidth > 0 ? 0 : c.getPreferredSize().width);
190                 }
191                 else {
192                     noOfComponentsWithoutHLock += sizeWidth > 0 ? 0 : 1;
193                 }
194             }
195
196             // Loops through all the components in the container and places them.
197             for (java.awt.Component c : parent.getComponents()) {
198                 boolean isVLock = (Boolean)Component.readAttribute(c,"vlock");
199                 boolean isHLock = (Boolean)Component.readAttribute(c,"hlock");
200
201                 // Calculate the size of the swingcomponent based on the locks and orientation.
202                 if (orientation == Layout.Orientation.VERTICAL) {
203                     width = isHLock ? c.getPreferredSize().width : parent.getSize().width - (padding*2);
204                     height = isVLock ? c.getPreferredSize().height :
205                                        (parent.getSize().height - (padding*(1+parent.getComponents().length))
```

```
206                                 - totalSizeHeight - totalHeightOfComponentsWithVLock) /
207                                 Math.max(noOfComponentsWithoutVLock,1);
208             }
209             else {
210                 width = isHLock ? c.getPreferredSize().width :
211                                 (parent.getSize().width - (padding*(1+parent.getComponents().length))
212                                 - totalSizeWidth - totalWidthOfComponentsWithHLock) /
213                                 Math.max(noOfComponentsWithoutHLock,1);
214                 height = isVLock ? c.getPreferredSize().height  : parent.getSize().height - (padding*2);
215             }
216
217             // Handle Size annotation. Override width and height is Size exists for the swingcomponent.
218             int sizeWidth = Component.readAttribute(c,"width") == null ? 0 :
219                             (Integer)Component.readAttribute(c,"width");
220             int sizeHeight = Component.readAttribute(c,"height") == null ? 0 :
221                             (Integer)Component.readAttribute(c,"height");
222             width = sizeWidth > 0 ? sizeWidth : width;
223             height = sizeHeight > 0 ? sizeHeight : height;
224
225             // Place the swingcomponent.
226             c.setBounds(x,y,width,height);
227
228             // Increase the coordinates for the next swingcomponent to be added.
229             y += (orientation == Layout.Orientation.VERTICAL ? height + padding : 0);
230             x += (orientation == Layout.Orientation.HORISONTAL ? width + padding : 0);
231         }
232     }
233
234     private java.awt.Dimension getLayoutSize(java.awt.Container parent, boolean isPreferred) {
235         int width = 0;
236         int height = 0;
237
238         Object o = Component.readAttribute(parent,"layout");
239         Layout.Orientation orientation = o == null ? Layout.Orientation.HORISONTAL : (Layout.Orientation)o;
240         for (java.awt.Component c : parent.getComponents()) {
241             // Handle Size and Layout annotation.
242             // Handle Layout annotation.
243             int sizeWidth = Component.readAttribute(c,"width") == null ? 0 : (Integer)Component.readAttribute(c,"width");
244             int sizeHeight = Component.readAttribute(c,"height") == null ? 0 : (Integer)Component.readAttribute(c,"height");
245
246             if (orientation == Layout.Orientation.HORISONTAL) {
247                 width += (sizeWidth > 0 ? sizeWidth : c.getPreferredSize().width);
248                 height = (Math.max(sizeHeight > 0 ? sizeHeight : c.getPreferredSize().height, height));
249             }
250             else
251             {
252                 width = (Math.max(sizeWidth > 0 ? sizeWidth : c.getPreferredSize().width, width));
253                 height += (sizeHeight > 0 ? sizeHeight : c.getPreferredSize().height);
254             }
255         }
```

```
256
257             // Calculate the padding
258             int padding = Component.readAttribute(parent,"padding") == null ? 0 :
259                           (Integer)Component.readAttribute(parent,"padding");
260             int hpadding = 0;
261             int wpadding = 0;
262             if (orientation == Layout.Orientation.HORISONTAL) {
263                 wpadding = padding * (parent.getComponents().length + 1);
264                 hpadding = padding * 2;
265             }
266             else {
267                 wpadding = padding * 2;
268                 hpadding = padding * (parent.getComponents().length + 1);
269             }
270
271             return new java.awt.Dimension ( width + wpadding, height + hpadding );
272         }
273     }
274 }
```

D.3 Frame.java

```
1  package dk.itu.hgl;
2
3  import javax.swing.*;
4  import java.awt.event.*;
5  import java.lang.reflect.Method;
6  import java.lang.reflect.InvocationTargetException;
7  import java.lang.reflect.Field;
8
9  public class Frame extends Container<JFrame> implements IFrame {
10
11     public Frame() {
12         this("");
13     }
14
15     public Frame(String title) {
16         swingcomponent = new JFrame(title);
17         swingcomponent.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18         swingcomponent.setLayout(new LayoutManager());
19
20         // Handle annotations for frame. This is a result of an issue discovered
21         // late in the process and because of changes in the design. The problem was
22         // that frame was the outer class and that the layout manager uses the content
```

```
23      // pane which is actualy a panel.
24      for(Field field : this.getClass().getEnclosingClass().getDeclaredFields()) {
25          if (field.getType() == Frame.class)
26              handleAnnotations(field,this,swingcomponent);
27      }
28  }
29
30  public String toString() {
31      return "frame: " + super.toString() + "("+ swingcomponent.getTitle() +")";
32  }
33
34  public void setSize(int width, int height) {
35      swingcomponent.setSize(width, height);
36  }
37
38  public void setup() {
39      addComponenets(this);
40  }
41
42  public void setTitle(String title) {
43      swingcomponent.setTitle(title);
44  }
45
46  public void display() {
47      swingcomponent.pack();
48      swingcomponent.setVisible(true);
49  }
50
51
52  protected class Panel extends Container<JPanel> implements IPanel {
53      public Panel() {
54          swingcomponent = new JPanel();
55          writeAttribute("hlock", true);
56          writeAttribute("vlock", true);
57          writeAttribute("padding",5);
58          swingcomponent.setLayout(new LayoutManager());
59          //swingcomponent.setBorder(BorderFactory.createLineBorder(new Color(255,0,0)));
60      }
61
62      public String toString() {
63          return "panel: " + super.toString() + "(no. "+ fieldNumber +")";
64      }
65
66      final protected class Button extends Component<JButton> implements IButton {
67          public Button() {
68              this("");
69          }
70
71          public Button(String text) {
72
```

```
73                  swingcomponent = new JButton(text);
74                  writeAttribute("hlock", true);
75                  writeAttribute("vlock", true);
76              }
77
78              public void setText(String text) {
79                  swingcomponent.setText(text);
80              }
81
82              // addActionListener is implemented directly in the button class,
83              // even though JMenuItem, JToggleButton uses this as well.
84              public void onClick(final Object target, final String targetmethod, final Object... targetmethodargs) {
85                  swingcomponent.addActionListener(new ActionListener() {
86                      public void actionPerformed(ActionEvent ev) {
87                          try {
88                              Method m = findMethod(target, targetmethod, targetmethodargs);
89                              m.setAccessible(true);
90                              Object result = m.invoke(target,targetmethodargs);
91                          } catch (NoSuchMethodException e) {
92                              System.out.print(e.toString());
93                          } catch (IllegalAccessException e) {
94                              System.out.print(e.toString());
95                          } catch (InvocationTargetException e) {
96                              System.out.print(e.toString());
97                          }
98                      }
99                  });
100             }
101
102             public void onMouseOver(final Object target, final String targetmethod, final Object... targetmethodargs)  {
103                 mouseEntered(target, targetmethod,targetmethodargs);
104             }
105         }
106
107         final protected class Label extends Component<JLabel> implements ILabel {
108             public Label() {
109                 this("");
110             }
111
112             public Label(String text) {
113                 swingcomponent = new JLabel(text);
114                 Panel.this.swingcomponent.add(swingcomponent);
115                 writeAttribute("hlock", true);
116                 writeAttribute("vlock", true);
117             }
118
119             public void setText(String text) {
120                 swingcomponent.setText(text);
121             }
122             public String getText() {
```

```
123         return swingcomponent.getText();
124     }
125
126     public void onMouseOver(final Object target, final String targetmethod, final Object... targetmethodargs) {
127         mouseEntered(target, targetmethod,targetmethodargs);
128     }
129
130     public void onTextChanged(final Object target, final String targetmethod, final Object... targetmethodargs) {
131         propertyChange(target, targetmethod,targetmethodargs);
132     }
133 }
134
135 final protected class TextField extends Component<JTextField> implements ITextField {
136     public TextField() {
137         this("");
138     }
139
140     public TextField(String text) {
141         swingcomponent = new JTextField(text);
142         writeAttribute("hlock", true);
143         writeAttribute("vlock", true);
144     }
145
146     public void setText(String text) {
147         swingcomponent.setText(text);
148     }
149
150     public String getText() {
151         return swingcomponent.getText();
152     }
153
154     public void onTextChanged(final Object target, final String targetmethod, final Object... targetmethodargs) {
155         keyReleased(target, targetmethod,targetmethodargs);
156     }
157
158     public void onMouseOver(final Object target, final String targetmethod, final Object... targetmethodargs) {
159         mouseEntered(target, targetmethod,targetmethodargs);
160     }
161
162 }
163
164
165 final protected class List extends Component<JList> implements IList {
166     private DefaultListModel model = new DefaultListModel();
167
168     public List() {
169         swingcomponent = new JList();
170         swingcomponent.setModel(model);
171         writeAttribute("hlock", true);
172         writeAttribute("vlock", true);
```

```
173 }
174
175 public void addItem(Object item) {
176     model.addElement(item);
177 }
178
179 public java.util.ArrayList getItems() {
180     java.util.ArrayList items = new java.util.ArrayList();
181     for (Object o :model.toArray())
182         items.add(o);
183     return items;
184 }
185
186 public void setItems(java.util.ArrayList items) {
187     model.clear();
188     for(Object o : items)
189         model.addElement(o);
190 }
191
192 public void removeItem(Object o) {
193     model.removeElement(o);
194 }
195
196 public Object getSelectedItem() {
197     return swingcomponent.getSelectedValue();
198 }
199
200 public void update() {
201     swingcomponent.updateUI();
202 }
203
204 public void clear() {
205     model.clear();
206 }
207
208 public int size(){
209     return model.size();
210 }
211
212 public void setSelectedItemIndex(int index) {
213     swingcomponent.setSelectedIndex(index);
214 }
215
216 public void onSelectedItemChanged(final Object target, final String targetmethod, final Object... targetmethodargs) {
217     swingcomponent.addListSelectionListener(new MyListListener() {
218         public void valueChanged(javax.swing.event.ListSelectionEvent event) {
219             if (!event.getValueIsAdjusting()) {
220                 try {
221                     Method m = findMethod(target, targetmethod, targetmethodargs);
222                     Object result = m.invoke(target,targetmethodargs);
```

```
223                             } catch (NoSuchMethodException e) {
224                                 System.out.print(e.toString());
225                             } catch (IllegalAccessException e) {
226                                 System.out.print(e.toString());
227                             } catch (InvocationTargetException e) {
228                                 System.out.print(e.toString());
229                             }
230                         }
231                     }
232                 });
233             }
234
235             private abstract class MyListListener implements javax.swing.event.ListSelectionListener {
236                 public void valueChanged(javax.swing.event.ListSelectionEvent event) {}
237             }
238         }
239     }
240 }
```

D.4 Height.java

```
1  package dk.itu.hgl;
2
3  import java.lang.annotation.Target;
4  import java.lang.annotation.ElementType;
5  import java.lang.annotation.Retention;
6  import java.lang.annotation.RetentionPolicy;
7
8  @Target(ElementType.FIELD)
9  @Retention(RetentionPolicy.RUNTIME)
10 public @interface Height {
11     int value() default -1;
12 }
```

D.5 Hlock.java

```
1  package dk.itu.hgl;
2
3  import java.lang.annotation.Target;
4  import java.lang.annotation.ElementType;
5  import java.lang.annotation.Retention;
6  import java.lang.annotation.RetentionPolicy;
7
8  @Target(ElementType.FIELD)
9  @Retention(RetentionPolicy.RUNTIME)
10 public @interface Hlock {
11     boolean value();
12 }
```

D.6 IButton.java

```
1  package dk.itu.hgl;
2
3  public interface IButton extends IComponent {
4      void setText(String text);
5
6      void onClick(Object target, String targetmethod, Object... targetmethodargs);
7      void onMouseOver(Object target, String targetmethod, Object... targetmethodargs);
8  }
```

D.7 IComponent.java

```
1  package dk.itu.hgl;
2
3  // In the current version of HGL the is not used.
4  interface IComponent {
5  }
```

D.8 IFrame.java

```
1 package dk.itu.hgl;
2
3 public interface IFrame extends IComponent {
4     String toString();
5     void setSize(int width, int height);
6     void setup();
7     void setTitle(String title);
8     void display();
9 }
```

D.9 ILabel.java

```
1 package dk.itu.hgl;
2
3 public interface ILabel extends IComponent {
4     void setText(String text);
5     String getText();
6
7     void onMouseOver(Object target, String targetmethod, Object... targetmethodargs);
8     void onTextChanged(Object target, String targetmethod, Object... targetmethodargs);
9 }
```

D.10 IList.java

```
1 package dk.itu.hgl;
2
3 public interface IList extends IComponent {
4     void addItem(Object item);
5     Object getSelectedItem();
6     java.util.ArrayList getItems();
7     void setItems(java.util.ArrayList items);
8     void removeItem(Object item);
9     void update();
```

```
10      void clear();
11      int size();
12      void setSelectedItemIndex(int index);
13
14      void onSelectedItemChanged(Object target, String targetmethod, Object... targetmethodargs);
15  }
```

D.11 IPanel.java

```
1   package dk.itu.hgl;
2
3   public interface IPanel extends IComponent {
4       String toString();
5   }
```

D.12 ITextField.java

```
1   package dk.itu.hgl;
2
3   public interface ITextField extends IComponent {
4       void setText(String text);
5       String getText();
6
7       void onTextChanged(Object target, String targetmethod, Object... targetmethodargs);
8       void onMouseOver(Object target, String targetmethod, Object... targetmethodargs);
9   }
```

D.13 Layout.java

```
1  package dk.itu.hgl;
2
3  import java.lang.annotation.Target;
4  import java.lang.annotation.ElementType;
5  import java.lang.annotation.Retention;
6  import java.lang.annotation.RetentionPolicy;
7
8  @Target(ElementType.FIELD)
9  @Retention(RetentionPolicy.RUNTIME)
10 public @interface Layout {
11     enum Orientation {VERTICAL,HORISONTAL};
12
13     Orientation value();
14 }
```

D.14 Vlock.java

```
1  package dk.itu.hgl;
2
3  import java.lang.annotation.Target;
4  import java.lang.annotation.ElementType;
5  import java.lang.annotation.Retention;
6  import java.lang.annotation.RetentionPolicy;
7
8  @Target(ElementType.FIELD)
9  @Retention(RetentionPolicy.RUNTIME)
10 public @interface Vlock {
11     boolean value();
12 }
```

D.15 Width.java

```
1  package dk.itu.hgl;
2
3  import java.lang.annotation.Target;
4  import java.lang.annotation.ElementType;
```

```
5  import java.lang.annotation.Retention;
6  import java.lang.annotation.RetentionPolicy;
7
8  @Target(ElementType.FIELD)
9  @Retention(RetentionPolicy.RUNTIME)
10 public @interface Width {
11     int value() default -1;
12 }
```