Hochschule
Ravensburg-Weingarten
Technik | Wirtschaft | Sozialwesen

Studiengang Angewandte Informatik(Bachelor)
Fakultät Elektrotechnik und Informatik

Bachelor-Arbeit

Erweiterung von Layout-Manager-Konzepten für Jo Widgets

zur Erlangung des akademischen Grades
Bachelor of Science

vorgelegt von:

Nikolaus Moll

17. Juni 2011

1. Gutachter: Prof. Dr. rer. nat. Martin Zeller
2. Gutachter: Dipl.-Inf. Michael Grossmann

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben. Die Arbeit wurde bisher, in gleicher oder ähnlicher Form, keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Unterschrift

Ort, Datum

Abstract

Thema:	Erweiterung von Layout-Manager-Konzepten für Jo Widgets
Verfasser:	Nikolaus Moll
Betreuer:	Prof. Dr. rer. nat. Martin Zeller Dipl.-Inf. Michael Grossmann
Abgabedatum:	17. Juni 2011

Diese Bachelor-Arbeit beschäftigt sich mit der Frage, wie Jo Widgets um Layout-Manager-Mechanismen erweitert werden kann. Im Grundlagenteil dieser Arbeit werden einige Layout Manager, das Framework Jo Widgets und daneben Layout-Konzepte anderer GUI-Frameworks vorgestellt. Auf Basis der Grundlagen werden danach die Anforderungen an einen Layout-Mechanismus in Jo Widgets definiert.

Der Layout-Mechanismus erweitert Jo Widgets um eine Schnittstelle für Layout Manager. Daneben werden einige einfache und mit MiGLayout ein mächtiger Layout Manager implementiert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Im Bereich von Enterprise-Anwendungen wird schon länger auf Programmiersprachen wie Java gesetzt, die eine gewisse Plattformunabhängigkeit bieten. Doch dieser Plattformunabhängigkeit sind Grenzen gesetzt. Neben der Tatsache, dass für jede zu unterstützende Plattform eine entsprechende Laufzeitumgebung vorhanden sein muss[1], schränkt auch die Bindung des Anwendungscodes an eine UI-Technologie[2] diese Unabhängigkeit ein.

Durch den Boom des Internets und der Smartphones spielt auch das Internet als Plattform eine wichtigere Rolle. Der Wunsch nach Anwendungen, die relativ einfach mit verschiedenen UI-Technologien zusammenarbeiten können, wird zunehmend größer. Die Portierung und die parallele Entwicklung für mehrere Zielplattformen sind nicht nur zeitaufwendig und somit teuer, sondern führen auch noch zu Code-Redundanz, die wiederum die Wartbarkeit des Quellcodes reduziert und den allgemeinen Implementierungsaufwand erhöht. *Single Sourcing* ist ein Weg, Anwendungen ohne Anpassung am eigentlichen Anwendungscode auf verschiedene Plattformen bzw. UI-Technologien zu bringen (siehe 2.1). Mit *Jo Widgets* entwickelt die innoSysTec GmbH ein Single-Sourcing-Framework für Java, das Anwendungen von der verwendeten UI-Technologie entkoppelt (siehe 3).

Das Thema Layouting fand in Jo Widgets lange keine große Beachtung, da hierfür ein für die beiden wichtigen Plattformen *Swing* und *SWT* verfügbarer Layout Manager verwendet wurde.

1.1 Zielsetzung

Jo Widgets soll um einen Layouting-Mechanismus erweitert werden, der nicht auf GUI-Framework-spezifische Layout Manager zurückgreift. Dazu wird die Vorgehensweise etablierter GUI-Frameworks untersucht und miteinander verglichen. Da für eine genaue Zielsetzung Kenntnisse über Jo Widgets vorausgesetzt werden, folgt diese erst in Kapitel 5.

1.2 Gliederung

Zunächst sollen in Kapitel 2 einige Grundbegriffe und Technologien im Zusammenhang mit Layout Managern erläutert werden, die für das weitere Verständnis dieser Arbeit notwendig sind. Dabei werden auch einige Layout Manager kurz vorgestellt. In Kapitel 3 wird

[1] wie z.B. das Java Runtime Environment

[2] wie Swing oder SWT

das Framework Jo Widgets vorgestellt. Neben der Entstehung seiner Architektur wird auch noch auf den aktuellen Stand bezüglich des Layoutings eingegangen. Anschließend wird in Kapitel 4 ein Vergleich zwischen verschiedenen GUI-Frameworks in Bezug auf Layout Manager durchgeführt. Dabei stehen die vorhandenen Schnittstellen und die Arbeitsweise im Vordergrund, nicht die verfügbaren Layout Manager selbst. In Kapitel 5 wird dann die Zielsetzung auf Basis der Grundlagen nochmals konkreter formuliert und einige Anforderungen benannt. Darauf folgt mit Kapitel 6 die Realisierung der Schnittstellen und die Implementierung einiger Layout Manager. Kapitel 7 bildet den Abschluss dieser Arbeit. Neben einer Zusammenfassung enthält es einen kurzen Ausblick auf mögliche Verbesserungen.

Kapitel 2

Grundlagen

Dieses Kapitel geht auf grundlegende Themen ein, die für das weitere Verständnis dieser Arbeit notwendig sind. Dabei wird der Begriff *Single Sourcing* definiert. Anschließend wird das Framework *Qt Jambi* kurz vorgestellt. Abschließend werden Grundbegriffe zum Thema Layouting erläutert und einige Layout Manager vorgestellt.

2.1 Single Sourcing

Der Begriff *Single Sourcing* wird von Chris Aniszczyk in [1, Folie 25] folgendermaßen beschrieben:

> Single source publishing, also known as single sourcing, allows the same source to be used in different runtime environments.

Single Sourcing ist demzufolge die Möglichkeit, denselben Quellcode in verschiedenen Laufzeitumgebungen zu verwenden. Dies ist bereits ein Ziel der modularen Entwicklung. Einschränkungen gibt es dann, wenn Module Abhängigkeiten auf ein konkretes GUI-Framework haben. An dieser Stelle geht Single Sourcing einen Schritt weiter. Mit Hilfe generischen GUI-Codes soll die Anwendung von der UI-Technologie unabhängig gemacht werden. Häufig wird als Beispiel für Single Sourcing *SWT* herangeführt, für das es mit *RWT*[1] eine (noch unvollständige) Web-Implementierung auf Basis des SWT-API gibt.

Die wichtigsten Gründe für Single Sourcing sind:

- **Multi-Plattform-Entwicklung**: Die Bedeutung von Web-Anwendungen steigt durch den Boom des Internets kontinuierlich und damit auch Wunsch nach Anwendungen, die auf verschiedenen Plattformen nutzbar sind. Dieser Trend wird auch durch die immer leistungsfähigeren Smartphones und Tablet-PCs begünstigt, die fast überall den Zugriff auf das Internet ermöglichen.
- **Unabhängigkeit**: Die Entwicklungszeit und die Nutzungsdauer einer Enterprise-Anwendung summiert sich auf mehrere Jahre. Die Entscheidung für eine UI-Technologie wird dabei früh gefällt. In diesem Zeitraum können neue Betriebssysteme und neue GUI-Frameworks erscheinen, oder die Weiterentwicklung eines GUI-Frameworks eingestellt werden. Single Sourcing kann die Abhängigkeit auf eine UI-Technologie reduzieren.

[1] RAP Widget Toolkit

- **Synergie-Effekte**: Synergie-Effekte zwischen Anwendungen, die auf verschiedenen UI-Technologien basieren, begrenzen sich normalerweise auf die Anwendungslogik. Generischer GUI-Code kann hier zu weiteren Synergie-Effekten führen.

2.2 Qt Jambi

Qt[2] ist ein für viele Plattformen zur Verfügung stehendes Anwendungsframework, das neben GUI-Bibliotheken u.a. auch Unterstützung für XML, für Netzwerkprogrammierung und für SQL-Datenbanken bietet. Das für die Entwicklung verantwortliche Unternehmen Trolltech wurde 2008 von Nokia aufgekauft. Bei Qt handelt es sich eigentlich um ein C++-Framework, es existieren allerdings auch Implementierungen für C, C#, Java, Perl, Python und Ruby. *Qt Jambi* ist die Java-Implementierung für Qt. Anstatt den gesamten Framework-Quellcode zu portieren, wurden Adapter-Klassen entwickelt, die mit Hilfe von JNI[3] die C++-Qt-Bibliotheken nutzen. [10]

Qt erlaubt die Entwicklung von Multi-Plattform-Anwendungen, die auf allen Betriebssystemen eine native Erscheinung haben. Mit Unterstützung für Microsoft Windows, Linux und Apple Mac OS X deckt es die verbreiteten Desktop-Betriebssysteme ab. Zu den bekanntesten Qt-Anwendungen gehören die VoIP-Software Skype, Google Earth, Oracle VirtualBox und VLC media player.

2.3 Layout Manager

Layout Manager sind Klassen, die die Positionierung und Größenberechnung von Kind-Elementen in einem Container übernehmen. Sie stellen ein Mittel dar, die Programmoberfläche nicht über absolute Angaben zu definieren, sondern anhand von Regeln[4]. Dadurch sollen die Benutzeroberflächen von Fenstergröße und verwendetem Zeichensatz[5] unabhängig bleiben. Vor allem in Java findet dieses Konzept großen Anklang, und mit Swing und SWT setzen die beiden dort bekanntesten GUI-Frameworks darauf.

Die Regeln hängen vom verwendeten Layout Manager und - sofern möglich - von seiner Konfiguration ab.

Im Kontext von Layout Managern sind drei Größenangaben von GUI-Elementen von großer Bedeutung:

- **Minimum Size**: Sie gibt die Mindestgröße an, die für die Darstellung des GUI-Elements notwendig ist.
- **Preferred Size**: Sie gibt die bevorzugte Größe eines GUI-Elements an
- **Maximum Size**: Sie gibt die maximale Größe an, die ein GUI-Element annehmen kann

[2] Qt wird ausgesprochen wie das englische Wort *cute* ([kju:t], übersetzt pfiffig bzw. nett).
[3] Java Native Interface, ein API um Funktionen einer C/C++-Bibliothek aufzurufen und umgekehrt
[4] Nichtsdestotrotz gibt es sogenannte *NullLayouts*, die die absolute Positionierung erlauben
[5] Schriftart, Schriftgröße, Schriftstil

Mit Hilfe seines Regelwerks und anhand dieser Größen kann ein Layout Manager den zur Verfügung stehenden Platz auf die Kind-Elemente des Containers aufteilen. Ob und wie diese Größen berücksichtigt werden, hängt vom Layout Manager selbst und den verwendeten Regeln ab. Da Container selbst Kind-Elemente darstellen können, gehört es zu den Aufgaben der Layout Manager, die Mindest-, die bevorzugte und die Maximalgröße eines Containers zu berechnen.

2.3.1 Wichtige Begriffe

Client-Area und Margins

Als Client-Area wird der für Kind-Elemente nutzbare Bereich eines Containers bezeichnet.

Im Kontext der GUI-Programmierung ist ein Margin ein Innenrand, eine Art Einzug. Dieser Rand hat direkten Einfluss auf die Größe der Client-Area. Je nach Seite wird der Rand als *margin-top*, *margin-bottom*, *margin-left* oder *margin-right* bezeichnet. Die Größen der vier Seitenränder kann unterschiedlich sein. Abbildung 2.1 zeigt das Konzept:

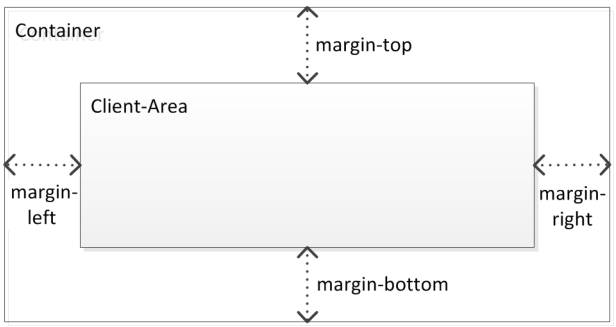


Abbildung 2.1: Die Margins in einem Container und seine Client-Area

Gap

Ein Gap[6] ist ein Zwischenraum zwischen zwei benachbarten GUI-Elementen im selben Container. Es wird zwischen horizontalen und vertikalen Gaps unterschieden.

Grid

Ein Grid ist ein Gitter. Ein Grid-basierter Layout Manager ordnet die Komponenten zellenweise in Zeilen und Spalten an. Es wird aber keine Aussage gemacht, wie die einzelnen Zellen dimensioniert werden oder wie viele Kind-Elemente in einer Zelle aufgenommen werden können. Die einzelnen Zellen des Gitters werden auch Kacheln genannt.

Constraints

Constraints[7] sind Anweisungen bzw. Informationen für den Layout Manager. Mit Hilfe von Constraints lässt sich ein Layout anpassen. Ob es Constraints gibt und wie sie aussehen, hängt vom Layout Manager ab.

[6] übersetzt u.a. Abstand, Lücke
[7] übersetzt u.a. Auflage, Bedingung bzw. Beschränkung

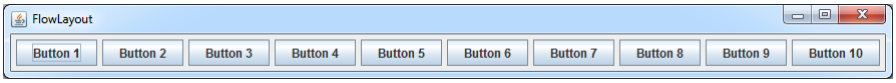
2.3.2 Beispiele für Layout Manager

In diesem Abschnitt werden einige Layout Manager beschrieben, die entweder Swing selbst mitbringt oder unter Swing sehr verbreitet sind. Oftmals sind die Layout Manager in gleicher oder ähnlicher Form auch für andere GUI-Frameworks verfügbar, oder lassen sich portieren.

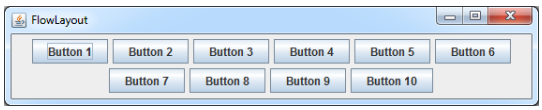
FlowLayout

Das *FlowLayout* ist einer der einfachsten Layout Manager und ist beim Swing-Container *JPanel* voreingestellt. Die Komponenten werden horizontal nebeneinander dargestellt. Ist in einer Zeile kein Platz mehr für die nächste Komponente, werden die nachfolgenden Komponenten in einer neuen Zeile angeordnet. Dieser Zeilenumbruch wird *Wrapping* genannt. Die Abstände zwischen den Komponenten (jeweils horizontal und vertikal) lassen sich genau festlegen so wie auch die Ausrichtung (linksbündig, zentriert, rechtsbündig). Das FlowLayout nutzt lediglich die bevorzugte Größe und ignoriert die Mindest- und Maximalgrößen von Komponenten für das Layouting, selbst wenn nicht genug Platz zur Verfügung steht. Das führt dann dazu, dass Komponenten in einen nicht-sichtbaren Bereich des Containers gelegt werden. [13]

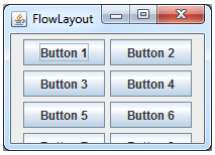
Abbildung 2.2 zeigt eine Beispielanwendung in unterschiedlichen Fenstergrößen:



(a) bevorzugte Fenstergröße



(b) veränderte Fenstergröße (1)



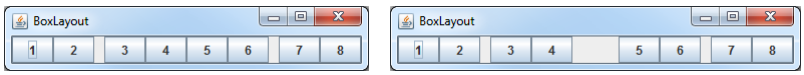
(c) veränderte Fenstergröße (2)

Abbildung 2.2: FlowLayout-Beispielanwendung

BoxLayout

Das *BoxLayout* ähnelt dem FlowLayout sehr. Anders als das FlowLayout kann es allerdings nicht nur horizontal, sondern auch vertikal verwendet werden. Außerdem gibt es kein Wrapping von Komponenten, wenn nicht genug Platz zur Verfügung steht. BoxLayout berücksichtigt alle Größenangaben für das Layouting. Bei horizontaler Ausrichtung versucht es, alle Komponentenhöhen mit der höchsten Komponente anzugleichen, und bei vertikaler Ausrichtung geht es analog mit den Komponentenbreiten vor. Zwischen die Komponenten lassen sich individuell Gaps fester oder variabler Größe einfügen.[12]

Die folgenden Abbildungen zeigen ein Beispielprogramm mit acht Buttons und zwei festen und einem variablen Zwischenraum:



(a) bevorzugte Fenstergröße (b) veränderte Fenstergröße

Abbildung 2.3: BoxLayout-Beispielanwendung

BorderLayout

Beim *BorderLayout* wird die Client-Area in fünf Bereiche aufgeteilt: *Center*, *North*, *South*, *West* und *East*. Jeder dieser Bereiche kann höchstens eine Komponente aufnehmen, die allerdings selbst ein Container sein kann. Abbildung 2.4 zeigt den schematischen Aufbau des Layouts. BorderLayout ist standardmäßig bei der Klasse *JFrame*[8] gesetzt. Beim Layouting berücksichtigt BorderLayout die bevorzugte Größe von Komponenten und hält sich auch an die Mindest- und Maximalwerte. [11]

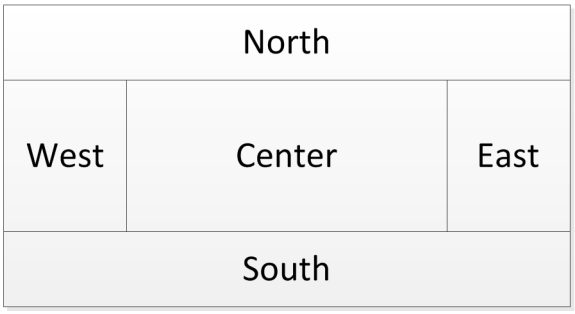


Abbildung 2.4: Schematische Aufteilung der Client-Area im BorderLayout

GridLayout

GridLayout ist ein Grid-basierter Layout Manager, der die Client-Area in gleich große, rechteckige Kacheln unterteilt. Die Zahl der Kacheln lässt sich zeilen- und spaltenweise festlegen. Werden mehr Komponenten hinzugefügt als Kacheln zur Verfügung stehen, wird jeweils eine weitere Spalte hinzugefügt. Untypisch für einen Layout Manager ist, dass er sämtliche Größenangaben der Kindkomponenten ignoriert. Alle Komponenten füllen immer eine ganze Kachel aus. Es ist möglich, den horizontalen und den vertikalen Abstand der Kacheln festzulegen.[14]

Abbildung 2.5: GridLayout-Beispielanwendung

Die Beispiel-Anwendung in Abbildung 2.5 zeigt zehn Buttons in einem GridLayout. Die Beschriftungen der Buttons zeigen die festgelegten bevorzugten Größen. Das Grid wurde

[8] Eigentlich ist BorderLayout der Standard-Layout-Manager der *Content Pane* eines JFrames.

mit drei Zeilen und drei Spalten genau eine Kachel zu klein definiert, so dass eine weitere Spalte hinzugefügt wurde.

GridBagLayout

Mit den bisher aufgeführten Layout Managern ist eine komplexe Oberfläche nur durch Verschachtlung von Layouts realisierbar. Dieses Problem versucht das *GridBagLayout* zu lösen. Mit ihm lassen sich auch Layouts realisieren, die sich mit den einfacheren Layout Managern nicht umsetzen lassen würden. Im Gegensatz zum namensverwandten GridLayout sind beim GridBagLayout nicht zwangsläufig alle Container-Elemente gleich groß. Stattdessen wird die Größe einer Komponente anhand ihrer Größenvorgaben und dem zur Verfügung stehenden Platz errechnet.

GridBagLayout erlaubt keine explizite Definition seines Grids. Tatsächlich erfolgt sie implizit über die Constraints der Komponenten. Diese werden von der Klasse *GridBagLayoutConstraints* repräsentiert.

Der Umgang mit den Constraints von GridBagLayout ist allerdings etwas gewöhnungsbedürftig und der Layout-Code nicht immer auf Anhieb für Menschen verständlich (siehe „Beispiel und Vergleich mit anderen Layout Managern“ in Abschnitt 2.3.3).

FormLayout (JGoodies)

FormLayout von JGoddies[9] ist einer von zwei externen Layout Managern, die in der offiziellen Java-Dokumentation von Oracle als verbreitete Layout Manager aufgeführt werden[17]. Obwohl seine Stärke in der Darstellung von Formularen liegt, kann FormLayout als universeller Layout Manager bezeichnet werden, der für eine Vielzahl von Aufgaben verwendet werden kann. Die Motivation hinter FormLayout ist es, einfacheren, menschenlesbaren und vor allem für Menschen vorstellbaren Layout-Code zu erlauben. FormLayout gehört zu den Grid-basierten Layout Managern[9, Seite 2f]

Der Layout-Code ist kompakter und wirkt leichter lesbar für Menschen (siehe „Beispiel und Vergleich mit anderen Layout Managern“ in Abschnitt 2.3.3).

MiGLayout

Der andere von Oracle genannte Layout Manager ist MiGLayout, der im folgenden Abschnitt näher behandelt wird.

2.3.3 MiGLayout

Ein verbreiteter Layout Manager unter Java ist *MiGLayout* der Firma MiG InfoCom AB [17]. Es kann unter der *BSD*-Lizenz genutzt werden[10] und ist somit auch für kommerzielle Anwendungen geeignet.

MiGLayout ist ein mächtiger, flexibler, aber vor allem einfach zu nutzender Layout Manager. Aufgrund seiner Flexibilität ist MiGLayout universell anwendbar, da sich damit nicht

[9] http://www.jgoodies.com/

[10] Neben der BSD-Lizenz ist MiGLayout auch unter der *GPL*-Lizenz nutzbar

nur alle Standard-Layouts aus Swing nachbauen lassen, sondern auch verschachtelte Layouts, für die sonst mehrere einfache Layout Manager verwendet werden mussten, aber auch Layouts, die mit den einfachen Layout Managern nicht zu bewerkstelligen sind. Zwar ist MiGLayout nicht der einzige Layout Manager, der diese Flexibilität bietet, doch gibt es kaum einen anderen Layout Manager, der dabei eine ähnlich einfache Nutzung erlaubt.

Die Flexibilität zeigt sich aber auch in den unterstützten GUI-Frameworks. MiGLayout wurde so entwickelt, dass es sich trotz seiner Komplexität vergleichsweise leicht für andere GUI-Frameworks portieren lässt. Mit Implementierungen für Swing und SWT unterstützt es standardmäßig die beiden wichtigsten GUI-Frameworks unter Java. Daneben gibt es Implementierungen für Qt Jambi[11] und für das Web-Framework Vaadin[12].

Diese Anpassungsfähigkeit hebt MiGLayout von FormLayout ab. Außerdem ist MiGLayout noch intuitiver zu verwenden und der Code lesbarer (siehe „Beispiel und Vergleich mit anderen Layout Managern“ weiter unten).

Arbeitsweise

MiGLayout verwendet ein zweidimensionales Grid, also ein Gitter, das für die Anordnung der Controls verwendet wird. Dieses Grid wird über Constraints definiert. Das Grid kann allerdings im Gegensatz zu manchen anderen Grid-basierten Layouts in beide Richtungen wachsen, d.h. es können mehr Spalten oder Zeilen genutzt werden als in der Definition angegeben. Außerdem ist es möglich, einzelne Zellen nochmals zu unterteilen. Zusätzlich kann die gesamte Grid-Definition nachträglich verändert werden.

Constraints

Constraints bilden das Regelwerk von MiGLayout. Es gibt innerhalb von MiGLayout vier verschiedene Kategorien von Constraints, die über drei Klassen realisiert werden:

- **Layout Constraints**: Mit Hilfe der Layout Constraints lässt sich das Verhalten des Layout Managers steuern. Außerdem können auch Einstellungen zum Grid gemacht werden. Die Layout Constraints werden durch die Klasse *LC* repräsentiert.
- **Column Constraints** und **Row Constraints**: Sie legen die Eigenschaften der Spalten bzw. der Zeilen im Layout fest und definieren zusammen das Grid. In der MiGLayout-Dokumentation werden diese Constraints auch als *Axis Constraints* bezeichnet [6], entsprechend trägt die Klasse für beide Typen den Namen *AC*.
- **Component Constraints**: Mit Hilfe der Component Constraints kann das Layouting einzelner Komponenten individuell gesteuert werden.

MiGLayout erlaubt die Angabe der Constraints auf zwei Arten:

- **Constraints-Klassen (AC, CC, LC)**: Die Constraints-Klassen enthalten Setter-ähnliche-Methoden, die wie beim Builder-Pattern auf das Präfix *set* verzichten und die Instanz der Klasse selbst zurückliefern. Dies erlaubt verkettete Methodenaufrufe und damit kürzeren Code.

[11] z.B. http://code.google.com/p/miglayout-qt/

[12] Diese ist Teil des Joodin-Projekts (http://code.google.com/p/joodin/)

- **Strings**: Ursprünglich sah MiGLayout nur Constraints in Textform vor. Bei der Syntax der Text-Constraints wurde darauf Wert gelegt, dass die Constraints lesbar und verständlich für Menschen bleiben. Die Strings werden mit Hilfe der Klasse *ConstraintParser* überprüft und in die entsprechenden Klassen umgewandelt.

Zu den Besonderheiten von MiGLayout zählt die Möglichkeit, sogenannte *Component Links* (oder auch nur *Links* genannt) zu verwenden. Ein solcher Link stellt eine Bindung einer Komponente zu einer anderen her, die sich dann relativ zu dieser anordnen lässt. Außerdem lassen sich Größen in verschiedenen Einheiten angeben, wie z.B. in Pixel, in dpi[13], im metrischen System und in Punkt.

Beispiel und Vergleich mit anderen Layout Managern

Anhand eines kleinen Adressformulars soll hier MiGLayout mit dem Layout Manager GridBagLayout und FormLayout verglichen werden. Dabei geht es weniger um den Leistungsumfang, sondern viel mehr um den Implementierungsaufwand und die Lesbarkeit des Quellcodes.

Die folgenden Abbildungen zeigen die umgesetzten Formulare:

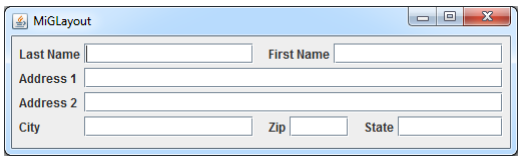


Abbildung 2.6: MiGLayout-Beispielformular

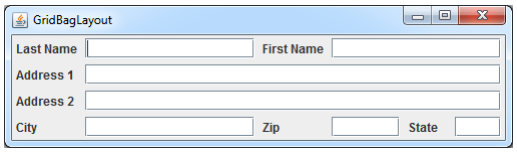


Abbildung 2.7: GridBagLayout-Beispielformular

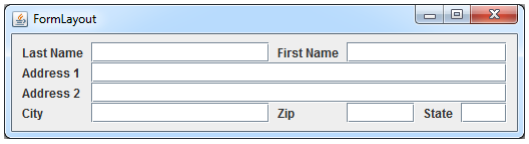


Abbildung 2.8: FormLayout-Beispielformular

Der folgende Quellcode zeigt die Implementierung mit GridBagLayout:

```
1 setLayout(new GridBagLayout());
2 GridBagConstraints c = new GridBagConstraints();
3 c.insets = new Insets(3, 4, 3, 4);
4
5 c.fill = GridBagConstraints.HORIZONTAL;
6 c.weightx = 1;
7
8 add(new JLabel("Last␣Name"), c);
```

[13] Dots [Pixel] per Inch, ein Inch entspricht 2,54 cm. Die Pixeldichte wird in dpi angegeben.

```
9  add(new JTextField(15), c);
10 add(new JLabel("First␣Name"), c);
11 c.gridwidth = GridBagConstraints.REMAINDER; // wrap
12 add(new JTextField(15), c);
13 c.gridwidth = 1;
14
15 add(new JLabel("Address␣1"), c);
16 c.gridwidth = GridBagConstraints.REMAINDER; // wrap
17 add(new JTextField(30), c);
18 c.gridwidth = 1;
19
20 add(new JLabel("Address␣2"), c);
21 c.gridwidth = GridBagConstraints.REMAINDER; // wrap
22 add(new JTextField(30), c);
23 c.gridwidth = 1;
24
25 add(new JLabel("City"), c);
26 add(new JTextField(15), c);
27 add(new JLabel("Zip"), c);
28 add(new JTextField(5), c);
29 add(new JLabel("State"), c);
30 add(new JTextField(3), c);
```

Listing 2.1: Layout-Beispiel mit GridBagLayout

Die Verwendung der Constraints ist relativ umständlich. Das Constraints-Objekt muss bereits vor dem eigentlichen *add*-Aufruf existieren und entsprechend konfiguriert sein[14]. Häufig wird nur ein solches Objekt verwendet und die Attribute für die jeweilige Komponente angepasst.

FormLayout stellt in dieser Hinsicht eine deutliche Verbesserung dar. Der etwas kürzere Code des FormLayout-Beispiels:

```
1  setLayout(new FormLayout(
2      "5dlu,␣pref,␣4dlu,␣pref:grow,␣4dlu,␣" +
3      "pref,␣4dlu,␣pref:grow,␣4dlu,␣" +
4      "pref,␣4dlu,␣pref:grow,␣5dlu", // columns
5      "5dlu,␣pref,␣pref,␣pref,␣pref,␣5dlu"  // rows
6      ));
7
8  add(new JLabel("Last␣Name"), CC.xy(2, 2));
9  add(new JTextField(15), CC.xy(4, 2));
10 add(new JLabel("First␣Name"), CC.xy(6, 2));
11 add(new JTextField(15), CC.xyw(8, 2, 5));
12
13 add(new JLabel("Address␣1"), CC.xy(2, 3));
14 add(new JTextField(30), CC.xyw(4, 3, 9));
15
16 add(new JLabel("Address␣2"), CC.xy(2, 4));
17 add(new JTextField(30), CC.xyw(4, 4, 9));
18
19 add(new JLabel("City"), CC.xy(2, 5));
20 add(new JTextField(15), CC.xy(4, 5));
21 add(new JLabel("Zip"), CC.xy(6, 5));
22 add(new JTextField(5), CC.xy(8, 5));
23 add(new JLabel("State"), CC.xy(10, 5));
24 add(new JTextField(3), CC.xy(12, 5));
```

Listing 2.2: Layout-Beispiel mit FormLayout

[14] Alternativ kann auch ein Konstruktor für die Constraints verwendet werden, der elf Parameter erwartet

Die Definition des Grids im Konstruktor ist allerdings relativ komplex. In den Strings stehen die Definitionen der Spalten und Zeilen, d.h. die Größenangaben und die Ausrichtung. Die einzelnen Zeilen bzw. Spalten werden durch Kommata voneinander getrennt. In dem Beispiel wird jede zweite Spalte für einen Gap verwendet, genauso die erste und die letzte Zeile. Die Abkürzung *dlu*, die in der Definition auftaucht, steht für *Dialog Unit*[15]. Spalten (oder auch Zeilen) mit der Definition *pref* nehmen die bevorzugte Größe ein, eine zusätzliche Angabe von *grow* bedeutet, dass sie wächst, wenn genug Platz zur Verfügung steht.

Über die statische Klasse *CC* werden *Component Constraints* erstellt. In diesem Beispiel werden nur die Grid-Positionen und gegebenenfalls die Anzahl der Zellen einer Komponente angegeben.

Das selbe Beispiel in MiGLayout kann wie in Listing 2.3 implementiert werden:

```
1  setLayout(new MigLayout("", "[][grow,fill]15[grow]"));
2
3  add(new JLabel("Last Name"));
4  add(new JTextField(15));
5  add(new JLabel("First Name"), "split");
6  add(new JTextField(15), "growx, wrap");
7
8  add(new JLabel("Address 1"));
9  add(new JTextField(30), "span, growx");
10
11 add(new JLabel("Address 2"));
12 add(new JTextField(30), "span, growx");
13
14 add(new JLabel("City"));
15 add(new JTextField(15), "");
16 add(new JTextField(15), "");
17 add(new JLabel("Zip"), "split");
18 add(new JTextField(5), "");
19 add(new JLabel("State"), "gap 15, split");
20 add(new JTextField(3), "growx, wrap");
```

Listing 2.3: Layout-Beispiel mit MiGLayout

Die Version in MiGLayout ist nicht nur kompakter als die beiden anderen Varianten, sondern auch verständlicher. Die Grid-Definition erfolgt im Konstruktor von MiGLayout. Der erste Parameter stellt die allgemeinen Layout Constraints dar, diese sind leer. Danach folgen die Column Constraints zur Definition der Spalten. Dabei wird eine Spalte von einem Paar eckiger Klammern repräsentiert. Innerhalb dieser Klammern kann das Verhalten der Spalte angegeben werden, z.B. die Breite. Die erste Spalte hat keine Angaben, in diesem Fall wird versucht, die bevorzugte Größe der darin liegenden Komponenten zu ermöglichen. Die zweite Spalte enthält die Angaben *grow* und *fill*. Während *grow* dazu führt, dass die Spalte wachsen kann, sorgt *fill* dazu, dass die Komponenten die Spalte ausfüllen. Eine Zahl zwischen zwei Spaltendefinitionen gibt den Gap zwischen beiden an. Die Definition der Zeilen erfolgt analog zu den Spalten, aber es wurde im Beispiel darauf verzichtet, Zeilen zu definieren.

Die Komponenten-Constraints sind hier in Form von Strings angegeben - sofern überhaupt notwendig. MiGLayout füllt standardmäßig eine Zelle nach der anderen mit hinzugefügten Komponenten, sofern die Constraints kein anderes Verhalten vorschreiben. Die Constraint

[15] Eine *Dialog Unit* ist eine von FormLayout eingeführte Einheit, die die Bildschirmpixeldichte (dpi) und die Fensterschriftgröße berücksichtigt, und so unabhängig von diesen Einflüssen eine ähnliche Wirkung auf dem Bildschirm erzielt. [9, Seite 6]

wrap veranlasst MiGLayout, nach dieser Komponente in der nächsten Zeile mit dem Einfügen von Komponenten fortzufahren. Mit Hilfe von *split* lässt sich eine Zelle nochmals unterteilen. Dadurch wird erreicht, dass in der ersten Zeile vier Komponenten sind, obwohl die Grid-Definition nur drei Spalten vorsieht. Die Constraint *growx* sorgt dafür, dass die Komponente horizontal wachsen kann und den zur Verfügung stehenden Platz auch ausnutzt. Wenn sich eine Komponente über mehrere Zellen erstrecken soll, wird die Constraint *span* verwendet. Ihr kann eine Zahl als Parameter angefügt werden, die die Anzahl an Zellen angibt. Ohne Parameter bedeutet, dass die Komponente alle verbleibenden Zellen der Zeile einnimmt. Ein explizites *wrap* ist hierbei nicht notwendig. Über *gap* lässt sich ein vertikaler Abstand angeben.

Eine umfangreichere Beschreibung der verfügbaren MiGLayout-Constraints findet man in [7].

Architektur

MiGLayout besteht aus einem Kern-Bereich, zu dem alle Klassen gehören, die unabhängig von der UI-Technologie sind. Zu diesen Kern-Klassen gehören:

- **AC**, **CC** und **LC**: Die bereits vorher angesprochenen Constraints-Klassen.
- **BoundSize**: Diese Klasse repräsentiert ein Größentupel. Sie nimmt Mindest-, bevorzugte und Maximalgröße auf.
- **ConstraintParser**: Eine Hilfsklasse zur Umwandlung der String-Repräsentation von Constraints in die jeweilige Klasse.
- **DimConstraint**: Diese Klasse nimmt die Werte einer Constraint in einer Richtung (Dimension) auf.
- **Grid**: Diese Klasse repräsentiert das Layout-Grid. Sie enthält den größten Teil der MiGLayout-Logik und führt die Layout-Berechnungen durch.
- **LayoutUtil**: Diese Hilfsklasse enthält Methoden für die Layout-Berechnung.
- **LinkHandler**: Bei der Hilfsklasse *LinkHandler* werden alle Component Links registriert.
- **PlatformDefaults**: Diese Hilfsklasse enthält diverse plattformabhängige Einstellungen für MiGLayout. Dezu gehören z.B. Abstände, die unter Mac OS traditionell größer ausfallen als unter Windows oder Linux. Es gibt Voreinstellungen für Windows, Mac OS und Linux.
- **ResizeConstraint**: Die zweite Klasse, die sich mit der Repräsentation von Constraints beschäftigt. *ResizeConstraint* enthält Informationen darüber, wie sich die Größe einer Komponente im Vergleich zu anderen verhält, wenn sich die Containergröße ändert.
- **UnitValue**: Die Klasse repräsentiert eine Größe. Sie kann mit einer der von MiGLayout unterstützten Größeneinheiten initialisiert werden und erlaubt die Umrechnung in Pixel.

Daneben gibt es drei Klassen, die für jeden GUI-Framework individuell implementiert werden müssen:

- **MigLayout**[16]: Diese Klasse stellt den Layout Manager dar. Sie implementiert das jeweilige Layout-Interface bzw. erweitert die entsprechende abstrakte Layout-Klasse des GUI-Frameworks.
- **MigComponentWrapper**: Sie stellt einen Adapter für den MiGLayout-Kern auf die Komponenten dar. Dazu implementiert sie das Interface *ComponentWrapper*, das im Kern-Bereich definiert ist.
- **MigContainerWrapper**: Analog zum MigComponentWrapper handelt es sich bei dieser Klasse um einen Adapter auf Container, die das *ContainerWrapper*-Interface implementiert. Da Container selbst Komponenten sind, erweitert die Klasse die MigComponentWrapper-Klasse.

[16]Die Klasse *MigLayout* wird gemäß der Namenskonvetion in Java mit kleinem „g“ geschrieben[5, Seite 147f]

Kapitel 3

Jo Widgets

Jo Widgets ist ein Widget[1]-basiertes Open-Source-Framework für Java, das federführend von der innoSysTec GmbH entwickelt wird. Es steht unter der *New-BSD*-Lizenz und wird auf Google Code gehostet[2]. Diese Arbeit basiert auf der gegenwärtig aktuellen Jo-Widgets-Version 0.4.0-SNAPSHOT.

3.1 Ziele

Die Hauptziele von Jo Widgets sind vor allem:

- **Single Sourcing**: Das wichtigste Ziel von Jo Widgets ist Single Sourcing (siehe 2.1). Es soll eine einheitliche Schnittstelle für verschiedene GUI-Frameworks bieten und dabei die Anwendung unabhängig von der genutzten UI-Technologie machen.

- **High-Level-API**[3]: Jo Widgets soll eine umfangreiche API bieten. Aufgaben, die häufig gemacht werden, sollen so weit möglich direkt im API untergebracht werden. Dadurch kann Jo Widgets dabei helfen, die Menge an GUI-Code zu reduzieren.

- **Erweiterbarkeit**: Es soll möglich sein, Jo Widgets sowohl um die Unterstützung neuer GUI-Frameworks, als auch um neue Widgets zu erweitern.

3.2 Architektur

Dieser Abschnitt erklärt, welche Überlegungen zu der jetzigen Jo-Widgets-Architektur geführt haben. Dabei werden nur die Teile der Architektur behandelt, die für das Verständnis wichtig sind.

[1] Ein Widget stellt eine Schnittstelle zwischen Benutzer und Anwendung dar
[2] http://www.jowidgets.org
[3] Das Application Programming Interface ist die Schnittstelle eines Frameworks zur Anwendung

3.2.1 Entstehung der Architektur

Um das Hauptziel zu verwirklichen, wurde eine Architektur entwickelt, die Anwendungscode von GUI-Framework-spezifischem Code trennt. Ein erster Ansatz sieht eine allgemeine Programmierschnittstelle (API) für alle GUI-Frameworks vor. Daneben gibt es Implementierungen dieser API für Swing und SWT. Diese Implementierungen sind Plugins und werden *Swing Impl* bzw. *SWT Impl* genannt.

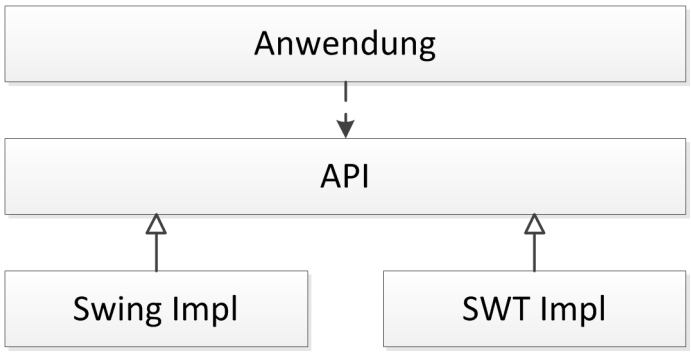


Abbildung 3.1: Eine einfache Version eines Single-Sourcing-Frameworks

Auch wenn mit der Entkopplung des Business-Codes vom GUI-Framework das Hauptziel erreicht wird, gibt es einige Probleme. Dieser Ansatz führt zu einem hohen Grad an Code-Redundanz, da für jeden GUI-Framework der komplette Funktionsumfang der API implementiert werden muss. Vor allem sogenannte Convenience-Methoden[4] bestehen häufig aus Code, der mit API-Aufrufen auskommt und selbst nicht vom GUI-Framework abhängt. Selbst wenn die API lediglich um Convenience-Methoden erweitert wird, müssen alle Plugins entsprechend angepasst werden. Code-Redundanz sorgt für geringere Wartbarkeit, da der Quellcode unübersichtlicher wird und bei Änderungen am Code diese an mehreren Stellen durchgeführt werden müssen.

Diese Probleme lassen sich durch eine zusätzliche Schicht zwischen der API und den Plugins lösen, der sogenannten *Common Impl*[5], siehe Abbildung 3.2.

Die Common Impl besteht aus Klassen, die den gemeinsamen Basis-Code enthält. Einige dieser Klassen könnten abstrakt sein und die Plugins darauf aufbauen. Die Plugins würden dann direkt von der *Common Impl* abhängen. Abstrakte Klassen bringen allerdings Nachteile mit sich. So wären zur Entwicklung der Plugins Implementierungsdetails der *Common Impl* notwendig und Änderungen an der *Common Impl* erschwert. In [4, Seite 27] wird ein Prinzip des objektorientierten Entwurfs folgendermaßen definiert: „Ziehe Objektkomposition der Klassenverbertung vor.“ Mit Hilfe von Komposition kann die *Common Impl* Klassen aus den Plugins nutzen. Dies wird durch eine zusätzliche Schnittstelle zwischen *Common Impl* und den Plugins erreicht, dem *Service Provider Interface (SPI)*. Jedes Plugin, das das SPI implementiert, kann von der *Common Impl* verwendet werden. Da die *Common Impl* in dieser Architektur nun direkt die API implementiert, wird sie nun nur noch *Impl* genannt. Die mit dieser Schicht fertige Architektur zeigt Abbildung 3.3[6].

[4] „convenience“ bedeutet übersetzt Komfort, Bequemlichkeit. Convenience-Methoden sind Methoden, die die Programmierung für einen Entwickler bequemer machen. Bei der Methode *JFrame.add()* handelt es sich um eine solche, da ihre einzige Aufgabe darin besteht, die Arbeit an die *Content Pane* des JFrames zu delegieren[16].

[5] common bedeutet in diesem Kontext allgemein bzw. gemeinsam

[6] frei nach http://jowidgets.org/pic/jowidgets_modules_no_workbench.gif

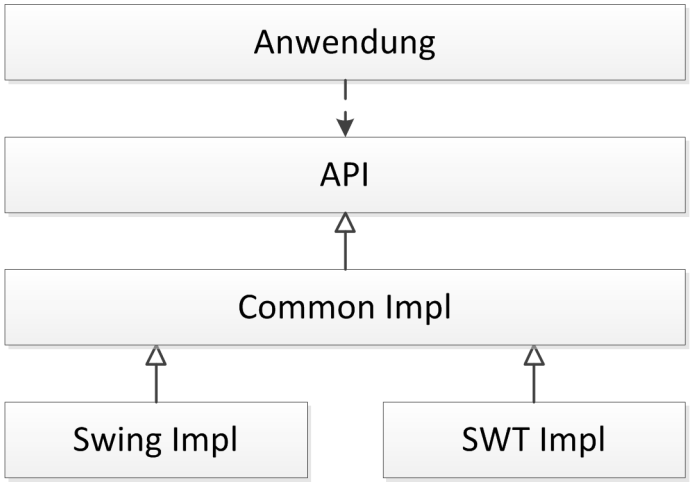


Abbildung 3.2: Erweiterung der Architektur

3.2.2 Die Architektur

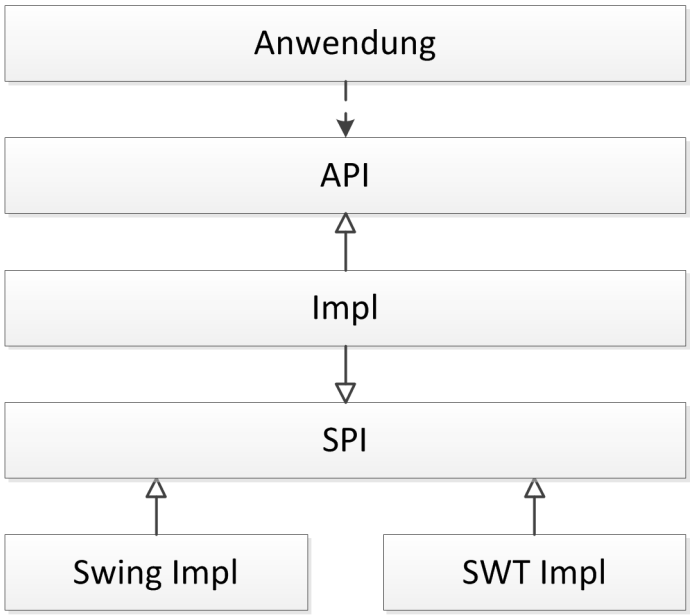


Abbildung 3.3: Die fertige Architektur von Jo Widgets

API

Das *API* ist die Schnittstelle des Frameworks zur Anwendung. Es definiert den Standard-Funktionsumfang des Frameworks und lässt sich von einer Anwendung erweitern. Das API besteht überwiegend aus Interfaces. Die meisten davon repräsentieren die Widgets und die dazugehörigen Setups und Descriptoren[7]. Neben diesen Interfaces enthält das API auch die *Toolkit*-Klasse (siehe 3.4) und einige Aufzählungstypen.

[7] Setups enthalten die Initialkonfiguration von Widgets. Descriptoren erweitern diese um Typinformationen, also um welchen Widget-Typ es sich handelt. Mit Hilfe eines solchen Descriptors wird ein Widget erzeugt.

Impl

Die *Impl* ist die Implementierung der API.

SPI

Das *SPI*[8] ist die Schnittstelle der API-Implementierung zu den GUI-Framework-Plugins. Das SPI und das API können orthogonal zueinander sein, auch wenn sie stellenweise auf gemeinsamen Interfaces basieren. Das SPI wurde möglichst schlank definiert, d.h. es wurde so weit möglich auf Convenience-Methoden verzichtet.

Plugins

Das Plugin-Konzept stellt die Erweiterbarkeit von Jo Widgets in Bezug auf UI-Technologien sicher. Die Plugins stellen die Verbindung zu den GUI-Frameworks her. Sie basieren auf dem *Adapter Pattern*[9]. Grundsätzlich ist es dadurch möglich, Jo Widgets mit beliebigen Java-UI-Technologien zu verwenden, dazu gehören Web-Frameworks wie GWT und Vaadin, selbst ein Plugin für Android ist denkbar.

Ein Plugin wird auch als *Service-Provider* bezeichnet, da es das SPI implementiert.

Jo Widgets bringt Service-Provider für Swing und SWT mit, weitere Plugins wie für Qt Jambi[10] und dem Web-Framework Vaadin[11] befinden sich in der Entwicklung.

3.2.3 Vorteile dieser Architektur

Eine mit gängigen GUI-Frameworks entwickelte Enterprise-Anwendung lässt sich nur sehr aufwändig auf ein anderes Framework portieren. Wurde die selbe Anwendung mit Jo Widgets implementiert, reduziert sich der Portierungsaufwand enorm. Selbst wenn es noch keinen Service-Provider für das Ziel-Framework gibt und dieser implementiert werden muss, ist bei umfangreichen Projekten der Aufwand geringer. Der Grund liegt daran, dass die Menge an Code eines Service-Providers im Vergleich zu einer Enterprise-Anwendung deutlich geringer ausfällt.

Das Framework selbst muss nicht an neue UI-Technologien angepasst werden. Durch die Architektur reicht es aus, einen neuen Service-Provider zu entwickeln. Durch das schlanke SPI beläuft sich die Menge an Code eines Service-Providers momentan auf etwa 7.000 Zeilen, für den vollständigen Funktionsumfang von Jo Widgets wird sie auf bis zu 10.000 Zeilen geschätzt[12]. Dabei besteht der Code eines Service-Providers überwiegend aus verhältnismäßig einfach zu entwickelnden Wrapper-Methoden, die Aufgaben an die Klassen des GUI-Frameworks delegieren. Demgegenüber besteht die *Impl* (also die Standard-Implementierung der Jo-Widgets-API) bereits aus über 27.000 Codezeilen und es wird geschätzt, dass sie auf mindestens 60.000 Zeilen wächst.

[8] Service Provider Interface

[9] Ein Adapter (oder auch Wrapper) erlaubt die Kommunikation zweier Komponenten, die normalerweise nicht miteinander kommunizieren können

[10] Jombi, http://code.google.com/p/jombi/

[11] Joodin, http://code.google.com/p/joodin/

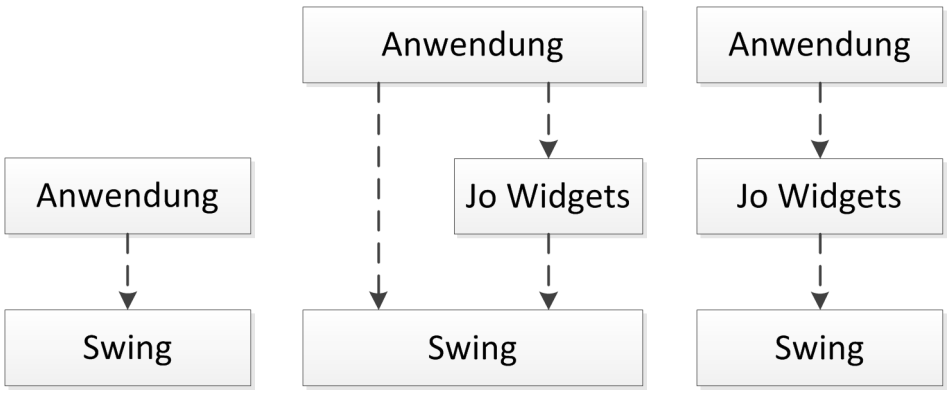
[12] Die Zahlen zu den Code-Zeilen basieren auf dem Eclipse-Plugin *Metrics*, http://sourceforge.net/projects/metrics/. Die Schätzungen stammen aus [8].

Ohne die Unterteilung in *Impl* und Service-Provider würde das Framework letztendlich aus über 60.000 Code-Zeilen bestehen, die für jede UI-Technologie angepasst werden müssten. Für drei Technologien wären das bereits knapp 200.000 Zeilen Code. Dieser Code würde große Mengen an redundantem Code enthalten, der die Wartung erschwert. Demgegenüber stehen bei der hier vorgestellten Architektur dann etwa 90.000 Code-Zeilen[13], das ist knapp weniger als die Hälfte. Bei zunehmender Anzahl von unterstützten Technologien fällt der Vorteil noch deutlicher aus.

3.3 Anwendungsmöglichkeiten

Das Framework Jo Widgets kann ergänzend zu bisherigem GUI-Code verwendet werden. Somit ist es möglich, einzelne Fenster und Komponenten mit Jo Widgets zu erstellen und in einem bestehenden Programm zu nutzen (siehe Abbildung 3.4b). Dies erlaubt es, Anwendungen progressiv nach Jo Widgets zu portieren. Ein mit Jo Widgets implementierter Dialog kann in verschiedenen Anwendungen mit unterschiedlichen GUI-Frameworks genutzt werden.

Abbildung 3.4 zeigt am Beispiel von Swing die Anwendungsmöglichkeiten von Jo Widgets:



(a) klassische Anwendung (b) hybride Anwendung (c) Jo-Widgets-Anwendung

Abbildung 3.4: Verschiedene Anwendungsmöglichkeiten von Jo Widgets

3.4 Toolkit

Jo Widgets enthält ein sogenanntes *Toolkit*[14]. Das Toolkit ist der Einstiegspunkt für die Nutzung von Jo Widgets, denn es bietet den Zugriff auf die verschiedenen Hilfsklassen von Jo-Widgets. Der wichtigste Teil der Hilfsklassen sind Klassen mit den Fabrik-Methoden[15], über die auch die Widgets erstellt werden. Aus Convenience-Gründen sind alle Methoden der Toolkit-Klasse statisch. Diese statischen Methoden delegieren die Aufrufe an eine Klasse, die das Interface *IToolkit* implementiert. Jo Widgets bringt eine Standard-Implementierung für dieses Interface mit, das sogenannte *DefaultToolkit*, das für die meisten Anwendungsfälle zweckmäßig ist. Auch hier zeigt sich die Offenheit von Jo Widgets, denn neben dem DefaultToolkit kann auch ein anderes Toolkit verwendet werden.

[13] 60.000 für die *Impl*, jeweils 10.000 für die drei Service-Provider

[14] übersetzt Werkzeugsatz

[15] Eine *Factory Method* bzw. Fabrik-Methode erzeugt eine Instanz einer Klasse.

Für die Instantiierung der *IToolkit*-Klasse ist eine Klasse, die das Interface *IToolkitProvider* implementiert, verantwortlich. Dieses Interface enthält nur eine *get*-Methode, die die Toolkit-Instanz zurückliefert. Sofern bei der statischen Toolkit-Klasse nicht explizit anders angegeben, wird der *DefaultToolkitProvider* verwendet, der schließlich das DefaultToolkit instantiiert.

Es gibt Situationen, in denen das DefaultToolkit nicht sinnvoll ist. So ist in einer *Tomcat*-Umgebung[16] der Einsatz von statischen Klassenelementen nicht unproblematisch, da ein solches Element nur einmal pro Server und nicht einmal pro Session existiert. Da jedes Programm bzw. jede Session auf dem Webserver in einem eigenen Thread gestartet wird, kann sich ein spezielles für Web-Umgebungen angepasster Toolkit-Provider die Klasse *InheritableThreadLocal* zunutze machen, um pro Session genau ein Toolkit zu instantiieren und immer das korrekte zurückzuliefern. [15]

3.5 Layouting in Jo Widgets bisher

Das Thema Layouting fand in der Anfangsphase der Entwicklung von Jo Widgets keine große Beachtung. Mit MiGLayout gibt es einen Layout Manager, der für die beiden wichtigsten Jo-Widgets-UI-Technologien Swing und SWT nutzbar ist. Aufgrund seiner Flexibilität lassen sich mit MiGLayout die verschiedenen Standard-Layout-Manager nachbilden. Deshalb wurde MiGLayout zum Standard-Layout-Manager von Jo Widgets. Das Layouting fand ausschließlich innerhalb der Service-Provider statt, weshalb jeder Service-Provider eine passende Implementierung von MiGLayout benötigt.

Die nachfolgende Aufzählung fasst die Nachteile zusammen:

- **Implementierungsaufwand**: Einen Layout Manager im Service-Provider vorauszusetzen erhöht den Implementierungsaufwand der Service-Provider. Es entspricht nicht der Philosophie, die Service-Provider so schlank wie möglich zu halten.
- **Eingeschränkte Erweiterbarkeit**: Zwar sah das API bereits vor, Jo Widgets um weitere Layout Manager zu ergänzen, doch führt jeder hinzugefügte Layout Manager dazu, dass alle Service-Provider angepasst werden müssen.
- **Keine eigenen Layout Manager**: Es ist nicht möglich, innerhalb einer Anwendung einen eigenen Layout Manager zu verwenden, der für eine Aufgabe besser geeignet ist als die in Jo Widgets vorhandenen.

[16] Apache Tomcat ist eine Open-Source-Implementierung der Java-Servlet- und JavaServer-Pages-Technologien. Er ermöglicht die Ausführung von Java-Code auf einem Webserver. [3]

Kapitel 4

Vergleich der Layout-Manager-Schnittstellen

Dieses Kapitel vergleicht die Arbeitsweise von Layout Managern in den drei GUI-Frameworks Swing, SWT und Qt Jambi. Auch wenn die allgemeine Arbeitsweise von Layout Managern in den verschiedenen Frameworks große Ähnlichkeit hat, gibt es im Detail einige Unterschiede. Die gewonnenen Erkenntnisse sollen bei der Konzeption der Layout-Manager-Schnittstelle von Jo Widgets einfließen.

4.1 Swing

Abbildung 4.1 zeigt die Schnittstelle, die ein Layout Manager unter Swing implementieren muss[1]. Die Abbildung wurde auf die wichtigen Methoden reduziert.

Swing ermöglicht es, eine Instanz eines Layout Managers für mehrere Container zu nutzen, dementsprechend erwarten viele Methoden den Container als Parameter:

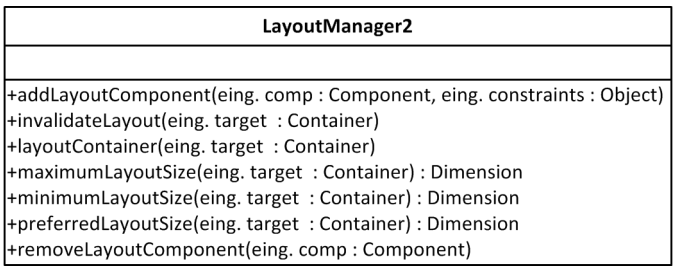


Abbildung 4.1: Das Interface *LayoutManager2* von Swing

Das Interface bringt einige nicht unbedingt notwendige Methoden mit, die für eine einfachere Implementierung eines Layout Managers sorgen. Dazu gehören die Methoden *addLayoutComponent* und *removeLayoutComponent*. Diese Methoden informieren den Layout Manager über Änderungen an Komponenten. Zu diesen Änderungen gehört, dass eine neue Komponente hinzugefügt wurde oder sich die Constraints einer Komponente verändert haben und dass Komponenten entfernt wurden.

[1] Grundsätzlich reicht das einfachere Interface *LayoutManager* für Swing-Layout-Manager aus, es bietet allerdings deutlich wengier Funktionalität.

Sinnvollerweise führt der Layout Manager eine Liste von Komponenten und speichert auch die Constraints zwischen. Hier zeigt sich der Nachteil daran, dass zwischen Container und Layout Manager keine 1:1-Beziehung vorgeschrieben ist, also dass ein Layout Manager genau zu einem Container gehört. Es ist mit einem gewissen Aufwand verbunden, eine saubere Trennung der Caches für die einzelnen Container zu implementieren. Deshalb verbieten viele Swing-Layout-Manager die Nutzung einer Instanz für mehrere Container.

Das Swing-Framework ruft bei jeder Größenänderung eines Containers die *invalidate*-Methode des zugehörigen Layout Managers auf. Das ist ein relativ defensives Verhalten um sicher zu stellen, dass ein Layout Manager beim nächsten Layout-Vorgang korrekt arbeitet. Da der Layout Manager die Ursache für den Invalidate-Aufruf nicht erfährt, hat der Layout Manager eigentlich keine Alternative zum Leeren des Caches, auch wenn es vielleicht nicht notwendig gewesen wäre.

4.2 SWT

Alle Layout Manager in SWT müssen von der abstrakten Klasse *Layout* abgeleitet werden. Wie auch bei Swing kann eine Instanz eines SWT-Layout-Managers für mehrere Container verwendet werden. Das Interface ist im Gegensatz zu Swing allerdings deutlich schlanker. Dies ist auch darin begründet, dass es ein schlichteres Größenkonzept verfolgt. Auf Mindest- und Maximalgrößen wird verzichtet, und ein Layout Manager ist nur in der Lage, die bevorzugte Größe zu berechnen. Die bevorzugte Größe einer Komponente in SWT entspricht der Größe, die mindestens benötigt wird, um den Komponenteninhalt darzustellen.

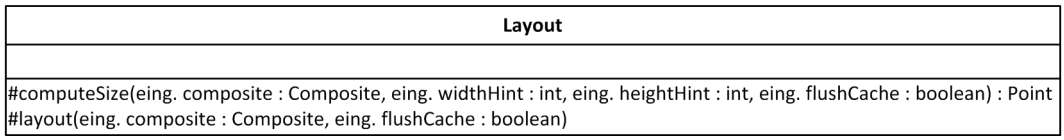


Abbildung 4.2: Die abstrakte Klasse *Layout* von SWT

In SWT kennt jedes Control seine eigenen Layout-Constraints[2]. Wie bei Swing verzichten viele SWT-Layout-Manager auf die Möglichkeit, eine Instanz für mehrere Container zu nutzen.

SWT verlangt von den Layout Managern mehr Eigenverantwortung als Swing. So teilt das Framework dem Layout Manager weder mit, ob Controls hinzugefügt oder gelöscht wurden, noch ob sich Layout-Constraints eines Controls geändert haben. Es bringt deshalb auch keinen Leistungsvorteil, Caches für Controls und deren Constraints zu führen, da diese bei jedem Layout-Vorgang[3] mit den tatsächlichen Werten abgeglichen werden müssen. Allerdings stellt dieser Abgleich eine Möglichkeit dar, die Gültigkeit zwischengespeicherter Layout-Berechnungen zu validieren.

Auf den ersten Blick mag der Abgleich teurer erscheinen als die Neuberechnung des Layouts. Bei genauerer Betrachtung relativiert sich dies jedoch. So gilt für den Abgleich und die Neuberechnung gleichermaßen, dass über alle Controls iteriert und alle Constraints abgefragt werden müssen. Stellt sich beim Abgleich heraus, dass keine Änderungen im Vergleich zu den letzten Berechnungen erfolgt sind, kann auf die Neuberechnung und das eigentliche Layouten verzichtet werden. Unnötige Layout-Vorgänge führen unter SWT auf

[2]Die Layout-Constraints können über die Methode *Control.getLayoutData()* abgefragt werden[18].

[3]Neben dem Layouting selbst kann auch das Berechnen der Mindest-, bevorzugten und Maximalgröße als Layout-Vorgang angesehen werden.

der Windows-Plattform mitunter zu einem leichten Flackern, weshalb es sinnvoll ist, die Controls nur dann neu anzuordnen, wenn es notwendig ist.

Das eigentliche Layouting findet in der *layout*-Methode statt. Die Methode *computeSize* dient der Berechnung der bevorzugten Größe. Anstatt einer *invalidate*-Methode gibt es die Möglichkeit, den beiden Methoden mittels des Parameters *flushCache* mitzuteilen, die zwischengespeicherten Werte zu verwerfen. Im Gegensatz zu Swing, das relativ häufig Gebrauch von *invalidate* macht, geschieht dies bei SWT relativ selten. So muss der Layout Manager selbstständig überprüfen, ob sich die Größe des Containers geändert hat und eine Neuberechnung des Layouts notwendig ist.

4.3 Qt Jambi

Alle Layout Manager in Qt Jambi werden von der abstrakten Klasse *QLayout* abgeleitet. Diese Klasse implementiert selbst zwei Interfaces und enthält eine Vielzahl an Methoden. Um einen Layout Manager zu implementieren, muss allerdings nur ein kleiner Teil der Methoden umgesetzt werden (siehe Abbildung 4.3). Die Layout-Manager-Schnittstelle von Qt Jambi unterscheidet sich in einigen Punkten grundlegend von Swing und SWT. So sieht es Qt Jambi nicht vor, dass ein Layout Manager für mehrere Container verwendet werden kann. Neue Widgets werden nicht dem Container selbst hinzugefügt, sondern seinem Layout Manager. Auf diese Weise erfährt der Layout Manager wie unter Swing, wann Widgets hinzugefügt oder entfernt werden. Das Constraints-Konzept ist in Qt Jambi nicht verankert. Typischerweise implementieren Layout Manager dann überladene Methoden zum Hinzufügen von Widgets, bei denen auch Constraints übergeben werden können.

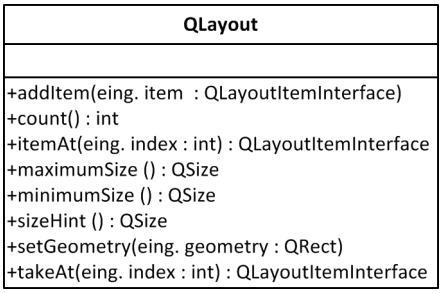


Abbildung 4.3: Die abstrakte Klasse *QLayout* von Qt Jambi

Über die Methode *addItem* werden dem Layout Manager neue Widgets hinzugefügt. Der Parameter ist vom Typ *QWidgetItem*, einer Klasse, die das *QLayoutItemInterface* implementiert und als Adapter für Widgets dient. Ein Layout Manager unter Qt Jambi muss selbstständig eine geordnete Liste dieser Adapter führen. Die Anzahl der verwalteten Elemente lässt sich über die Methode *count* abfragen. Zum Abrufen und Entfernen von Widgets dienen die Methoden *itemAt* und *takeAt*. Beide erwarten den Index des Widgets als Eingangsparameter und beide liefern den entsprechenden Adapter auf das Widget zurück, *takeAt* entfernt den Adapter zusätzlich noch aus seiner Liste.

Der Layout Manager wird über die Methode *setGeometry* angestoßen[4]. Dabei sollte Layout Manager wie auch unter SWT die Gültigkeit seines Layouts eigenständig überprüfen, da er nicht zuverlässig über Größenänderungen des zugehörigen Containers informiert wird. *QLayout* enthält zwar eine *invalidate*-Methode und diese wird auch aufgerufen, wenn die

[4]Die Position und Größe eines Widgets wird unter Qt als *Geometry* bezeichnet.

Fenstergröße verändert wird. Liegt der Container aber in einem Splitter[5] und ändert sich durch diesen seine Größe, wird sie allerdings nicht aufgerufen.

Die übrigen Methoden dienen der Abfrage der Mindest-, der bevorzugten und der Maximalgröße des Layouts (*minimumSize*, *sizeHint* und *maximumSize*).

4.4 Zusammenfassung

Aus der Betrachtung der verschiedenen Ansätze lassen sich wertvolle Schlussfolgerungen ziehen. So bringt es keine nennenswerten Vorteile, eine Instanz eines Layout Managers für mehrere Container zu verwenden. Im Gegenteil, die Nachteile überwiegen hier. Zwar wird dies sowohl unter Swing als auch unter SWT grundsätzlich ermöglicht, aber es führt bei steigender Komplexität des Layout Managers und zunehmendem Einsatz von Caching für die Layout-Berechnungen zu Problemen. Die Einstellungen und die Caches der einzelnen zu layoutenden Container lassen sich nicht sauber innerhalb einer Instanz voneinander trennen. Tatsächlich sind sogar einige Standard-Layout-Manager in Swing nicht in der Lage, mit mehreren Containern gleichzeitig umzugehen (z.B. *BorderLayout*). Ein Verzicht auf diese Möglichkeit hat den positiven Nebeneffekt, dass die Schnittstelle einfacher wird, da dann nicht jede Methode den Container als Parameter benötigt.

Der Grad an Eigenverantwortung variiert stark zwischen Swing, SWT und Qt Jambi. Er wirkt sich direkt auf den Umfang der Schnittstelle aus. Swing hält die Eigenverantwortung sehr gering, die Folge sind viele Methoden. SWT stellt das andere Extrem dar, da der Layout Manager hier sehr selbstständig arbeiten muss - dementsprechend wenig Methoden sind hier zu implementieren.

[5] Ein Splitter ist ein Container, der seine Client-Area mehreren Kindern zur Verfügung stellt. Die Trenner zwischen den Kindern können vom Benutzer verschoben werden um die Größe der Elemente zu verändern.

Kapitel 5

Zielsetzung

5.1 Zielsetzung

- **Layouting von SPI-Ebene auf API-Ebene**: Layouting auf SPI-Ebene hat den Nachteil, dass jeder Service-Provider den entsprechenden Layout Manager implementieren muss. Um das zu vermeiden, soll ein Layout-Mechanismus auf API-Ebene realisiert werden. Dadurch soll ebenfalls vermieden werden, dass Service-Provider Kenntnisse über verwendete Layout Manager benötigen.
- **Erweiterbarkeit in Bezug auf Layout Manager**: Anwendungen oder Programmkomponenten sollen eigene Layout Manager definieren und diese mit jedem Service-Provider nutzen können.
- **Layout Manager implementieren** MiGLayout soll innerhalb der Jo-Widgets-Impl implementiert werden. Dadurch muss nicht mehr für jeden Service-Provider MiGLayout verfügbar sein. Wenn ein Service-Provider MiGLayout nativ enthält, soll dieses weiterhin genutzt werden können. Bei die Implementierung von MiGLayout in der *Impl* sollen die MiGLayout-Kern-Klassen (also die Klassen, die unabhängig von der UI-Technologie sind), möglichst unverändert bleiben. Neben MiGLayout sollen noch einige weitere Layout Manager hinzugefügt werden.

5.2 Anforderungen an die Layout-Schnittstelle

Ein Interface für Layout Manager ist einer abstrakten Basisklasse vorzuziehen. In [4, Seite 25] empfehlen die Autoren „auf eine Schnittstelle hin [zu programmieren], nicht auf eine Implementierung“. Eine abstrakte Klasse wäre nur vorzuziehen, wenn die Layout Manager eine gemeinsame Implementierungsbasis benötigen würden. Für gemeinsame Logik sieht Jo Widgets allerdings die *Impl* vor, so dass es keine gewichtigen Argumente gibt, die für eine abstrakte Klasse sprechen. Weitere Anforderungen an die Schnittstelle sind:

- **Layouting**: Die wichtigste Aufgabe eines Layout Managers ist das Positionieren und Dimensionieren von Komponenten.
- **Größenberechnung**: In Abschnitt 2.3 wurde bereits erklärt, dass ein Layout Manager in der Lage sein sollte, die Mindest-, die bevorzugte und die Maximalgröße eines Containers zu berechnen.

- **Leeren des Caches**: Es soll möglich sein, dem Layout Manager mitzuteilen, seine Caches zu leeren.
- **Schlankes Interface** Das Layout-Interface soll relativ schlank bleiben und keine nicht unbedingt notwendigen Methoden oder Parameter enthalten.

Kapitel 6

Realisierung

6.1 Layout-Schnittstelle von Jo Widgets

Das API von Jo Widgets wurde um das Interface *ILayouter* für Layout Manager erweitert. Auf Ebene der Service-Provider musste jeweils eine Adapter-Klasse eingeführt werden, die Layout-Anfragen vom GUI-Framework an das *ILayouter*-Interface weitergeben.

Abbildung 6.1 zeigt das fertige *ILayouter*-Interface.

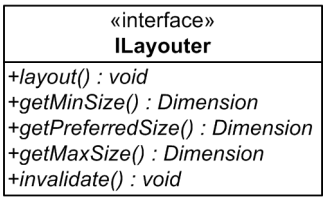


Abbildung 6.1: Das Interface *ILayouter*

- **layout()**: In dieser Methode positioniert und dimensioniert der Layout Manager die von ihm verwalteten GUI-Elemente.
- **invalidate()**: Mit Hilfe dieser Methode kann Jo Widgets dem Layout Manager mitteilen, dass er seine gecachten Informationen zum Layout verwerfen soll.
- **getMinSize(), getPreferredSize(), getMaxSize()**: Diese Methoden ermitteln die minimale Größe, die bevorzugte Größe und die maximale Größe des Layouts.

6.2 LayoutFactoryProvider

Layout Manager, die Jo Widgets selbst mitbringt, werden wie üblich im API in Form von Interfaces definiert und in der Impl implementiert. Für die Instantiierung dieser Layout Manager werden Fabrik-Methoden genutzt. Diese Methoden werden im Interface *ILayoutFactoryProvider* definiert. Die entsprechende Fabrik-Methode für den LayoutFactoryProvider selbst ist standesgemäß im Toolkit untergebracht. Über das Toolkit wird sichergestellt, dass es pro Session nur eine Instanz des LayoutFactoryProviders gibt.

Dabei werden die im API integrierten Layouts über ein Builder-Pattern erzeugt. Beim Builder-Pattern wird eine Hilfsklasse verwendet, die alle Konstruktorparameter aufnimmt und die Klasse instantiiert. Es eignet sich vor allem dann, wenn ein Konstruktor mehrere optionale Parameter hat, um ein Überladen des Konstruktors zu vermeiden. [2, Seite 14f] Außerdem bietet der LayoutFactoryProvider für jedes Layout eine Convenience-Methode an, die das Layout mit Standardwerten erzeugt.

6.3 Implementierte Layout Manager

6.3.1 NullLayout

Beim *NullLayout* handelt es sich um einen Layout Manager, der die absolute Positionierung und Dimensionierung von Komponenten erlaubt. Aufgrund der Tatsache, dass es dem Entwickler die vollständige Kontrolle beim Layouting überlässt, ist die Implementierung sehr schlicht. Die Methoden *layout* und *invalidate* sind leer, die Methoden zur Abfrage der Größe werden an den zugehörigen Container delegiert.

6.3.2 PreferredSizeLayout

Eine Erweiterung des NullLayouts stellt das *PreferredSizeLayout* dar und verwendet ebenfalls eine absolute Positionierung von Komponenten. Beim Setzen der Größe unterscheidet es sich allerdings vom NullLayout, das selbst keine Größen setzt. Das PreferredSizeLayout setzt alle Komponentengrößen auf ihre bevorzugten Größen. Der Layout Manager kann seine bevorzugte Größe errechnen, dabei werden die Koordinaten der äußersten Seiten rechts und unten berechnet. Die auf das Wesentliche vereinfachte Implementierung sieht wie folgt aus:

```
1  private Dimension calcPreferredSize() {
2    int maxX = 0;
3    int maxY = 0;
4    for (IControl control : container.getChildren()) {
5      Dimension controlSize = control.getPreferredSize();
6      Position controlPos = control.getPosition();
7      maxX = Math.max(maxX, controlPos.getX() +
           controlSize.getWidth());
8      maxY = Math.max(maxY, controlPos.getY() +
           controlSize.getHeight());
9    }
10   return new Dimension(maxX, maxY);
11 }
```

Listing 6.1: Methode zur Berechnung der bevorzugten Größe der Client-Area

Da alle Komponenten unabhängig von der Container-Größe immer in ihrer bevorzugten Größe gelayoutet werden, sind die minimale und die maximale Größen des Layouts identisch mit seiner bevorzugten Größe.

6.3.3 FillLayout

Das *FillLayout* ist ein Layout Manager, der den zur Verfügung stehenden Platz mit (höchstens) einer Komponente füllt. Es kann ein Rand (Margin) um die Komponente definiert

werden (siehe Abschnitt 2.3.1). Die Komponente wird nie kleiner angezeigt als es ihre Mindestgröße vorschreibt, wenn dennoch weniger Platz zur Verfügung steht, befindet sich ein Teil der Komponente im nicht-sichtbaren Bereich.

6.3.4 FlowLayout

Das *FlowLayout* stellt eine Mischung aus den Swing-Layout-Managern FlowLayout und BoxLayout dar (siehe 2.3.2). Die Möglichkeit, eine Orientierung (horizontal oder vertikal) anzugeben hat es mit dem BoxLayout gemein, genauso das fehlende Wrapping bei zu wenig zur Verfügung stehendem Platz. Die aktuelle Implementierung nutzt lediglich die bevorzugten Größen, in einer verbesserten Version können auch Mindest- und Maximalgrößen berücksichtigt werden.

6.3.5 BorderLayout

Bereits in Abschnitt 2.3.2 wurde der Layout Manager *BorderLayout* vorgestellt. Die Jo-Widgets-Implementierung orientiert sich sehr daran, auch wenn die Constraints dieser Implementierung die Bezeichnungen *TOP*, *BOTTOM*, *LEFT*, *RIGHT* und *CENTER* tragen. Ähnlich wie FlowLayout berücksichtigt die BorderLayout-Implementierung noch nicht alle Größenangaben von Komponenten. Der Layout Manager weist den bis zu vier Randkomponenten immer ihre bevorzugte Größe zu, das Element im Zentrum erhält den Rest der Client-Area.

6.3.6 MiGLayout

Dieser Abschnitt beschreibt die Implementierung von MiGLayout in das Jo-Widgets-Framework.

Einbinden der Kern-Klassen

Jo Widgets nutzt das Build-Werkzeug *Apache Maven*[1]. Dieses ist in der Lage, Projektabhängigkeiten aufzulösen. Einzelne Maven-Projekte bzw. -Module werden als Artefakte bezeichnet.

MiGLayout liegt in Form von Maven-Artefakten vor. Es gibt jeweils ein Artefakt für Swing, für SWT und für beide zusammen. Im gemeinsamen Artefakt sind außerdem einige Beispielanwendungen enthalten. Leider gibt es kein Artefakt, das nur die Kern-Klassen von MiGLayout enthält. Um die MiGLayout-Klassen innerhalb der Jo-Widgets-Implementierung zu nutzen, gibt es zwei Möglichkeiten. Die erste ist eine Abhängigkeit in Form eines Maven-Artefakts, die zweite ist die Integration der MiGLayout-Klassen in die *Impl* selbst.

Für die erste Möglichkeit kommen das SWT-Artefakt und das gemeinsame Artefakt aufgrund ihrer Abhängigkeiten auf SWT nicht in Frage. Neben der Verwendung des Swing-Artefakts ist auch die Erstellung eines eigenen Artefakts denkbar, das lediglich aus den Kern-Klassen von MiGLayout besteht.

[1] http://maven.apache.org/

Aufgrund der Tatsache, dass MiGLayout einige statische Klassen nutzt, die angepasst werden müssen, ist die Integration der Klassen die bessere Variante. Die angepassten Klassen werden aufgrund der Anpassungen direkt vom Toolkit von Jo Widgets abhängen, so dass ein eigenes Modul keine Vorteile bringt.

Die Klassen MigLayout, JoMigComponentWrapper und JoMigContainerWrapper

Die Original-Implementierungen von MiGLayout für Swing und SWT enthalten überladene Konstruktoren, um die Angabe von Constraints optional zu machen und sie dann mit Standardwerten zu belegen. Da die hier implementierten Layout Manager über das Builder Pattern erzeugt werden (siehe Abschnitt 6.2), reicht ein einziger Konstruktor aus. Die Standardwerte werden in der Hilfsklasse festgelegt. Der Code zum Erzeugen von einer MiGLayout-Instanz ist über das Builder Pattern zwar länger, aber auch lesbarer. Listing 6.2 zeigt die klassische Instantiierung:

```
1 container.setLayout(new MigLayout("", "[grow][][]"));
```

Listing 6.2: Instantiierung von MiGLayout mit überladenem Konstruktor

Hier werden nur die allgemeinen Layout Constraints und die Column Constraints übergeben, da die Angabe der Row Constraints optional ist. Das selbe Beispiel über das Builder Pattern ist allerdings intuitiver:

```
1 ILayoutFactoryProvider LFP = Toolkit.
     getLayoutFactoryProvider();
2 container.setLayout(LFP.migLayoutBuilder().
     columnConstraints("[grow][][]").build());
```

Listing 6.3: Instantiierung von MiGLayout über Builder Pattern

Die Hauptaufgabe der Klasse MigLayout neben dem eigentlichen Layouting ist es, die Constraints zum Layout und den Kind-Elementen zu verwalten. Sie erzeugt die entsprechenden Komponenten- bzw. Container-Wrapper. Die Wrapper werden so in einer Hash-Map abgelegt, um vom Wrapper direkt auf die entsprechenden Component Constraints zugreifen zu können.

Die Implementierung der MigLayout-Klasse orientiert sich stark an der SWT-Klasse. Der Grund liegt darin, dass der Layout-Mechanismus von Jo Widgets dem von SWT sehr ähnelt. Wie unter SWT kennt jedes Control von Jo Widgets seine eigenen Constraints, und wie bei SWT gibt es unter Jo Widgets noch keinen Mechanismus der einen Layout Manager darüber informieren kann, ob Container-Elemente hinzugefügt oder entfernt wurden. So prüft auch die MiGLayout-Implementierung von Jo Widgets bei jedem Layout-Vorgang, ob der Cache noch gültig ist.

Bei der Prüfung des Caches werden alle Container-Elemente abgefragt und geprüft, ob es bereits einen Wrapper für sie gibt. Falls nicht, wird ein Wrapper erzeugt und das Grid verworfen. Anschließend wird geprüft, ob es Wrapper gibt, deren Kind-Elemente nicht mehr Teil des Containers sind, also ob Kind-Elemente entfernt wurden. Wenn das der Fall ist, werden sie gelöscht und das Grid ebenfalls als ungültig markiert.

Die Wrapper-Klassen bestehen hauptsächlich aus Methoden, die die Arbeit delegieren. Zu den besonderen Methoden gehört die Methode zur Berechnung des Layout-Hashs. Diese Methode berechnet aus den Layout-Vorgaben[2] einen Hash-Wert und wird dazu verwendet,

[2] wie z.B. tatsächliche, Mindest-, bevorzugte und Maximalgröße

um Änderungen daran festzustellen. Diese Methode hat großen Einfluss auf die Performance des Layout Managers, da sie sehr häufig aufgerufen werden können. Die Abfrage der Größen kann dabei sehr performancelastig werden, weshalb die aktuelle Implementierung der Wrapper-Klassen darauf verzichtet, diese Größen mit in die Berechnung einfließen zu lassen. Bei Änderungen dieser Größen muss dann der Layout Manager über ein *invalidate* darauf hingewiesen werden, dass er seine Layout-Berechnungen neu durchführen muss.

Zwei Methoden, die eher untypisch für die Wrapper sind, sind die Methoden *getHorizontalScreenDPI* und *getVerticalScreenDPI*, die zur Abfrage der Pixeldichten des Ausgabegeräts in dpi. Dafür sieht Jo Widgets momentan noch keine Möglichkeit vor, so dass sie gegenwärtig einen Standardwert von 72 dpi zurückliefern.

Realisierung der Constraints

Wie bereits in 2.3.3 beschrieben, gibt es in MiGLayout zwei Möglichkeiten, Constraints zu definieren: Über Strings oder über Constraints-Klassen. Beide Varianten sollen vom Jo-Widgets-API unterstützt werden. Dementsprechend müssen die verwendeten Parametertypen für die Constraints dem API bekannt sein. Im Gegensatz zu den Constraints-Klassen sind Strings überall bekannt und sind deshalb problemlos zu verwenden. Aufgrund ihrer Implementierung können die Constraints-Klassen nicht einfach ins Jo-Widgets-API verschoben werden, da sie Abhängigkeiten auf weitere MiGLayout-Klassen haben. Die Folge wäre, dass der gesamte MiGLayout-Kern im API enthalten sein müsste.

Die Lösung stellen die zusätzlichen im API definierten Interfaces dar, die die Constraints-Klassen abstrahieren: *IAC*, *ICC* und *ILC*. Diese enthalten sämtliche Methoden, die der Konfiguration der Constraints dienen. Alle Methoden haben das jeweilige Interface selbst als Rückgabetyp, um die gewohnten verketteten Constraints-Definitionen zu ermöglichen.

Anstatt die Original-MiGLayout-Klassen anzupassen und sie die drei Interfaces implementieren zu lassen, werden drei Constraints-Adapter-Klassen verwendet. Diese Adapter-Klassen implementieren jeweils eines der Interfaces. Sie erzeugen ein Original-Constraints-Objekt von MiGLayout und delegieren alle Aufrufe an dieses. Abbildung 6.2 stellt das Klassendiagramm beispielhaft und stark vereinfacht dar. Sie zeigt eine fiktive Adapter-Klasse (*ConstraintsWrapper*), zu der eine Instanz der tatsächlichen Constraints gehört (*Constraints*) und die das Constraints-Interface (*IConstraints*) implementiert.

MigLayoutToolkit und IMigLayoutToolkit

Das Interface *IMigLayoutToolkit* definiert die Fabrik-Methoden zur Instantiierung der Constraints-Klassen. Die Methodenbezeichnungen *ac*, *cc* und *lc* entsprechen den bekannten Klassennamen, darüber hinaus gibt es zur leichteren Lesbarkeit die Convenience-Methoden *columnConstraints*, *componentConstraints*, *layoutConstraints* und *rowConstraints*.

Die in der Jo-Widgets-Impl realisierte Klasse *MigLayoutToolkit* implementiert das *IMigLayoutToolkit*-Interface, bietet aber noch weitere Methoden. MiGLayout enthält einige komplett oder teilweise statische Klassen, die z.B. im Umfeld eines Tomcat-Servers zu Problemen führen können (siehe Abschnitt 3.4). Da das MigLayoutToolkit über das Jo-Widgets-Toolkit erstellt wird, ist sichergestellt, dass es pro Session nur ein einziges Mal erstellt wird. Es eignet sich somit für den Zugriff auf statische Klassen, die in nicht-statische umgewandelt werden (siehe Abschnitt „Statische Klassen“).

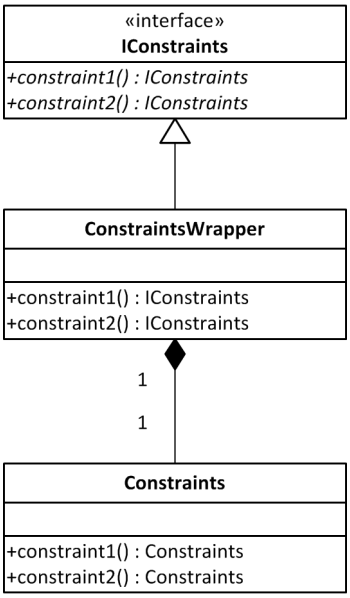


Abbildung 6.2: Schematische Hierarchie der Constraints-Adapter

Statische Klassen

Die in MiGLayout enthaltenen statischen Klassen sind *ConstraintParser*, *PlatformDefaults*, *LinkHandler* und *LayoutUtil*. Daneben gibt es noch die Klasse *UnitValue*, die einen statischen und einen nicht-statischen Teil enthält. Die Klasse *ConstraintParser* enthält nur statische Hilfsmethoden und muss deshalb nicht verändert werden.

Die übrigen Klassen erfordern Anpassungen. Die Idee ist es, aus den statischen Klassen Singleton[3]-ähnliche Klassen zu machen. Dazu wird die Klasse UnitValue in zwei Klassen aufgeteilt. Der nicht-statische Teil bleibt in der Klasse UnitValue, alle statischen Elemente und Methoden sind nun in der Klasse *UnitValueToolkit* untergebracht, mit der dann wie mit den vollständig statischen Klassen verfahren wird. In diesen werden alle statischen Elemente und Methoden in nicht-statische umgewandelt. Statische Initialisierungen werden in gegebenenfalls hinzuzufügenden Konstruktoren untergebracht. Um dafür zu sorgen, dass pro Klasse und pro Session nur eine Instanz erstellt wird, übernimmt nicht die Klasse selbst die Instantiierung sondern das MigLayoutToolkit, das auch den Zugriff auf die Klassen ermöglicht. Die Konstruktoren der Klassen sind darüber hinaus *package private*, um keine unbeabsichtigte Instantiierung zu ermöglichen.

Das MigLayoutToolkit führt die Instantiierung *lazy* durch, also nur bei Bedarf bzw. Zugriff auf die jeweilige Klasse. Dies ist notwendig, da eine Instantiierung innerhalb des MigLayoutToolkit-Konstruktors nicht möglich ist, da sich die einzelnen Klassen während ihrer Initialisierung mitunter gegenseitig benötigen. Aber erst wenn der Konstruktor von MigLayoutToolkit abgeschlossen ist, kann es über das Jo-Widgets-Toolkit abgefragt werden.

Der Umfang dieser Änderungen und die damit verbundenen Anpassungen erschweren den Austausch der MiGLayout-Kernklassen im Falle eines neuen MiGLayout-Releases.

[3]Das Entwurfsmuster Singleton stellt sicher, „dass eine Klasse genau ein Exemplar besitzt“ und „einen globalen Einstiegspunkt darauf“ bietet. [4, Seite 157]

Funktionale Abweichungen von Original-MiGLayout und Probleme

Die Möglichkeit, das Grid nachträglich zu ändern, ist in der Jo-Widgets-Implementierung von MiGLayout nicht vorgesehen. Zwar erlaubt auch die Jo-Widgets-Implementierung dies, doch sieht das Jo-Widgets-API keine Möglichkeit vor, den Layout Manager eines Containers abzufragen.

Im Vergleich zur Original-Version nutzt die Implementierung ein aggressiveres Caching. Normalerweise verhält sich MiGLayout sehr defensiv und prüft bei jedem Layout-Vorgang, ob sich irgendwelche Constraints oder Größenangaben von Komponenten geändert haben und berechnet dann gegebenenfalls das Grid neu. Das Abfragen der verschiedenen Größenangaben von Komponenten kann relativ zeitaufwendig sein. Eine Abfrage bewegt sich zwar im Nano-Sekundenbereich, finden aber viele Layout-Anfragen in sehr kurzen Intervallen statt, die alle eine Neuberechnung des Grids anstoßen (wie beim Vergrößern/Verkleinern eines Fensters), kann es zu einer ruckeligen Darstellung führen.

Es wurde in Betracht gezogen, einen Mechanismus zu implementieren, der einzelne Layout-Vorgänge bei häufigen Anfragen überspringt. Dabei traten allerdings einige Probleme auf, so dass davon Abstand genommen wurde. Die Zeitmessung ist nicht unproblematisch, da je nach zu verrichtender Arbeit ein Layout-Vorgang länger oder kürzer dauern kann. Außerdem wird ein Layout Manager nicht kontinuierlich aufgerufen, sondern nur bei Bedarf. Es ist nicht abzusehen, bei welcher Anfrage es sich um die vorerst letzte handelt. Somit muss der Layout Manager einen Thread erzeugen, der den Layout-Vorgang nach der letzten (verworfenen) Anfrage auslöst. Neben dem Overhead kann es hier zu dem Problem kommen, dass der zu layoutende Container während der Thread läuft bereits zerstört wurde.

Die Klasse *LinkHandler* fasst normalerweise alle Links eines Layout Managers in einer Map zusammen. Das API von Jo Widgets sieht es allerdings nicht vor, den Layout Manager eines Containers abzufragen. Deshalb wird für die Zuordnung nicht die Instanz des Layout Managers selbst, sondern die des Containers verwendet.

MiGLayout-Beispielanwendung

MiGLayout enthält zwei Versionen einer Beispielanwendung. Eine Version ist für Swing, die andere für SWT. Obwohl das Beispiel die Fähigkeiten von MiGLayout demonstrieren soll, eignet sie sich auch als Testanwendung. Leider bietet Jo Widgets noch nicht alle verwendeten Widgets an, so dass die Jo-Widgets-Version anstelle einer Auswahlliste einen Baum verwendet, außerdem ist das Beispiel noch nicht vollständig portiert, so werden stellenweise noch falsche Schriftarten bzw. -stile verwendet.

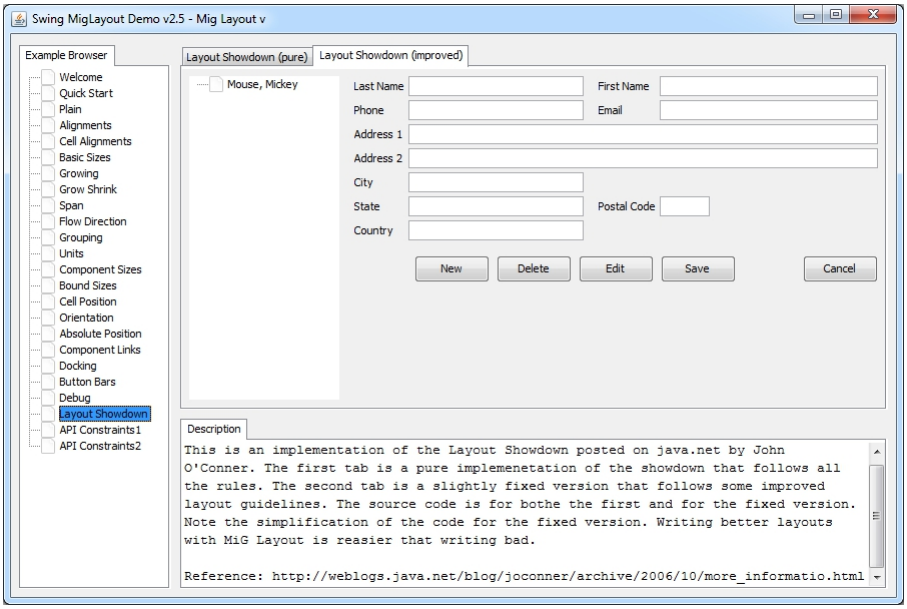


Abbildung 6.3: Demo-Anwendung mit Jo Widgets realisiert mit Swing als UI-Technologie

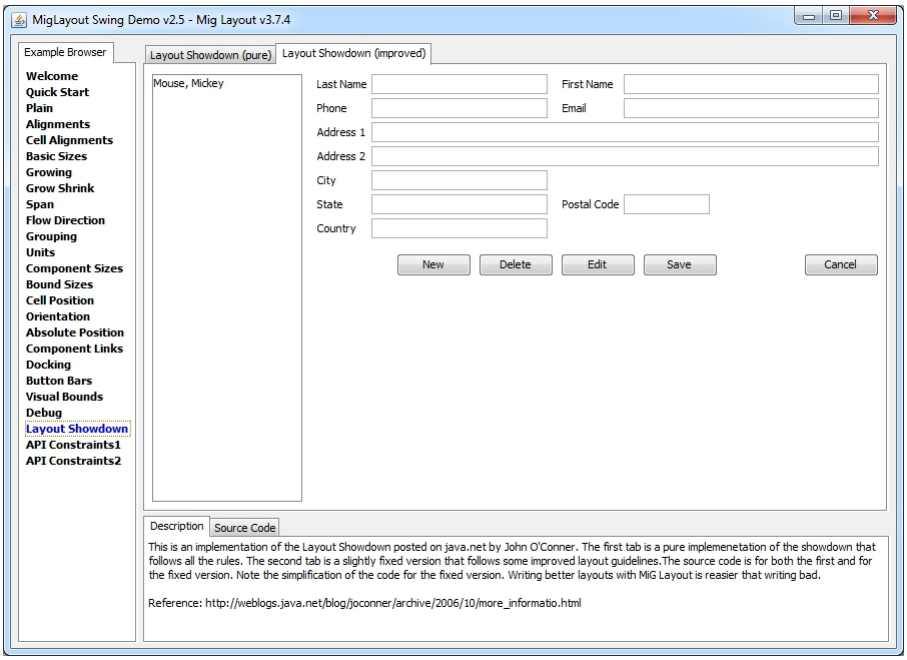


Abbildung 6.4: Original Swing-Demo-Anwendung von MiGLayout

Kapitel 7

Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, das Framework Jo Widgets um einen Layout-Manager-Mechanismus zu erweitern, der nicht auf nativem GUI-Code basiert. Jo Widgets ist ein Framework, dessen Hauptziel das Single-Sourcing von grafischen Benutzeroberflächen ist. Dazu verwendet es Plugins, die Adapter zu den entsprechenden GUI-Frameworks darstellen. Die bisherige Möglichkeit, das Layouting innerhalb der Plugins durchzuführen, bringt einige Probleme mit sich. So lässt sich das Framework nur schwer um neue Layout Manager erweitern, weil sie einerseits fest im Jo-Widgets-API verankert, andererseits noch von jedem Plugin implementiert werden müssen.

Es wurden verschiedene verbreitete Layout-Manager-Schnittstellen miteinander verglichen und die Vor- und Nachteile der jeweiligen Realisierung beschrieben. Mit Hilfe dieser Erkenntnisse wurde ein Interface für Layout Manager innerhalb dem Jo-Widgets-API definiert. Dieses umfasst fünf parameterlose Methoden und ist relativ schlank gehalten. Drei von ihnen dienen der Berechnung der Mindest-, der bevorzugten bzw. der Maximalgröße des Containers. Die beiden anderen Methoden stoßen den Layout-Vorgang an bzw. veranlassen den Layout Manager, seine gecachten Informationen zu verwerfen.

Durch die Implementierung von einfacheren Layout Managern und dem mächtigen MiG-Layout auf API-Ebene innerhalb von Jo Widgets wurde gezeigt, dass sich die Schnittstelle für den praktischen Einsatz eignet. Trotz der Tatsache, dass es sich bei MiGLayout um einen relativ leicht portierbaren Layout Manager handelt, traten bei der Implementierung einige Probleme auf.

Der Einsatz statischer Klassen bereitet vor allem in Web-Service-Umgebungen Schwierigkeiten. Um diesem Umfeld dennoch gerecht zu werden, wurden problematische statische Klassen in nicht-statische Klassen umgewandelt bzw. durch solche ersetzt. Im Vergleich zum Original-MiGLayout führt die implementierte Version ein aggressiveres Caching durch.

Für die korrekte Arbeitsweise der MiGLayout-Implementierung muss Jo Widgets noch um die Möglichkeit, die Pixeldichte in dpi vom Anzeigegerät abzufragen, ergänzt werden. Da Layout Manager mitunter komplexe Grid-Berechnungen durchführen, wäre eine Erweiterung des Jo-Widgets-API um Benachrichtigungen bei Änderungen an Containern sinnvoll. Dazu könnte der Container um entsprechende Events erweitert werden, an die sich die Layout Manager bei Bedarf anmelden. Außerdem wäre es sinnvoll, wenn ein Layout automatisch invalidiert würde, wenn sich die Constraints eines Controls ändern.

Noch sieht Jo Widgets nicht vor, dass eine MiGLayout-Implementierung innerhalb der SPI optional ist. Es fehlt ein Mechanismus, über den ein Service-Provider mitteilen kann, ob er MiGLayout nativ unterstützt oder eben nicht. Die Implementierung von Jo Widgets müsste dies dann berücksichtigen und bei Bedarf automatisch die eigene Version von MiGLayout verwenden.

Abbildungsverzeichnis

Listings

Literatur

[1] Chris Aniszczyk. *Single Sourcing RAP and RCP*. Eclipse Source. 2009. URL: http://www.slideshare.net/caniszczyk/single-sourcing-rcp-and-rap (besucht am 05. 06. 2011).

[2] Joshua Bloch. *Effective Java Second Edition*. The Java Series. Addison-Wesley, 2008. ISBN: 9780321356680. URL: http://books.google.com/books?id=ka2VUBqHiWkC&pg.

[3] The Apache Software Foundation. *Apache Tomcat - Welcome!* The Apache Software Foundation. 2011. URL: http://tomcat.apache.org/ (besucht am 15. 06. 2011).

[4] Erich Gamma u. a. *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*. Pearson Deurschland GmbH, 1996. ISBN: 3827318629. URL: http://books.google.com/books?id=-GXxUV0L6XsC.

[5] James Gosling u. a. *The Java Language Specification (Third Edition)*. The Java Series. Addison-Wesley, 2005. ISBN: 0321246780. URL: http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf.

[6] Mikael Grev. *AC (MiGLayout 3.7.4 API)*. MiG InfoCom AB. URL: http://www.migcalendar.com/miglayout/javadoc/index.html (besucht am 07. 06. 2011).

[7] Mikael Grev. *MiG Layout Cheat Sheet*. MiG InfoCom AB. 2011. URL: http://www.migcalendar.com/miglayout/cheatsheet.pdf (besucht am 15. 06. 2011).

[8] Michael Grossmann. *Interne Dokumentation*. innoSysTec GmbH. 2011.

[9] Karsten Lentzsc. *The JGoodies Forms Framework*. JGoodies. 2004. URL: http://www.jgoodies.com/articles/forms.pdf (besucht am 14. 06. 2011).

[10] Nokia. *Qt - A cross-plattform application and UI framework*. Nokia. URL: http://qt.nokia.com/products (besucht am 15. 04. 2011).

[11] Oracle. *BorderLayout (Java Platform SE 7 b141)*. Oracle. 2011. URL: http://download.oracle.com/javase/7/docs/api/java/awt/BorderLayout.html (besucht am 11. 06. 2011).

[12] Oracle. *BoxLayout (Java Platform SE 7 b141)*. Oracle. 2011. URL: http://download.oracle.com/javase/7/docs/api/javax/swing/BoxLayout.html (besucht am 11. 06. 2011).

[13] Oracle. *FlowLayout (Java Platform SE 7 b141)*. Oracle. 2011. URL: http://download.oracle.com/javase/7/docs/api/java/awt/FlowLayout.html (besucht am 11. 06. 2011).

[14] Oracle. *GridLayout (Java Platform SE 7 b141)*. Oracle. 2011. URL: http://download.oracle.com/javase/7/docs/api/java/awt/GridLayout.html (besucht am 11. 06. 2011).

[15] Oracle. *InheritableThreadLocal (Java Platform SE 6)*. Oracle. 2011. URL: http://download.oracle.com/javase/6/docs/api/java/lang/InheritableThreadLocal.html (besucht am 15. 06. 2011).

[16] Oracle. *JFrame (Java Platform SE 6)*. Oracle. 2011. URL: http://download.oracle.com/javase/6/docs/api/javax/swing/JFrame.html (besucht am 05. 06. 2011).

[17] Oracle. *Using Layout Managers (The Java(TM) Tutorials > Creating a GUI With JFC/Swing > Laying Out Components Within a Container)*. Oracle. URL: http://download.oracle.com/javase/tutorial/uiswing/layout/using.html (besucht am 05. 06. 2011).

[18] Eclipse contributors u. a. *Help - Eclipse SDK*. Eclipse contributors u. a. 2010. URL: http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/widgets/Control.html (besucht am 05. 06. 2011).